Architecture Description Languages for Retargetable Compilation

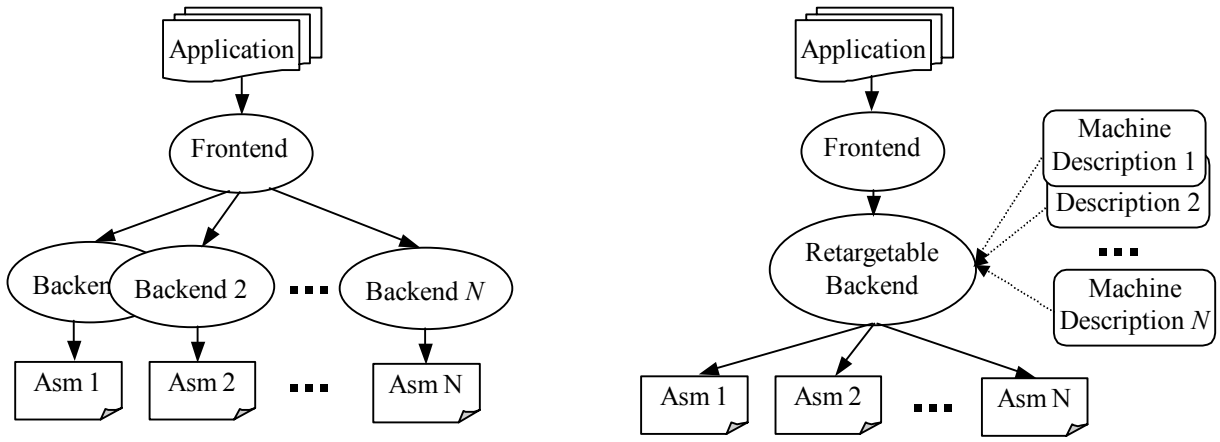
Wei Qin, Sharad Malik

Department of Electrical Engineering
Princeton University

I. INTRODUCTION

Retargetable compilation has been the subject of some study over the years. This work is motivated by the need to develop a single compiler infrastructure for a range of possible target architectures. Recent technology trends point to the growth of application specific programmable systems. This trend makes it doubly important that we develop efficient retargetable compilation infrastructures. The first need for this is in directly supporting the different target architectures that are likely to be developed to fuel this trend. The second and possibly more important need for this is in the design space exploration for the architecture and micro-architecture of the processor being developed. Any evaluation of a candidate needs a compiler to compile the application down to the candidate architecture, and a simulator to measure the performance. Since it is desirable to evaluate multiple candidates, a retargetable compiler (and simulator) is highly desirable.

The retargetability support largely needs to be provided for the back end of the compiler. Compiler back-ends may be classified into the following three types: customized, semi-retargetable and retargetable. Customized back-ends have little reusability. They are usually written for high quality proprietary compilers or developed for special architectures requiring non-standard code generation flow. In contrast, semi-retargetable and retargetable compilers aim to reuse at least part of the compiler back-end by factoring out target dependent information into machine description systems. The difference of customization and retargetability is illustrated in Figure 1.



(a) customized compilation flow (b) retargetable compilation flow

Figure 1 Illustration of different compilation flows

Semi-retargetable compilers, e.g. lcc[31] and gcc[37], share significant amount of code among different target back ends. This reduces the time to port the compilers to new target machines. However, they still require a non-trivial amount of programming effort for each target due to the idiosyncrasies of the target architectures or their calling conventions. In a semi-retargetable compiler implementation, the instruction set of the target is usually described in tree patterns as is required by popular code generation algorithms[1]. General-purpose register files and usages of the registers also need to be specified. This type of a machine description system serves as the interface between the machine independent part and the machine dependent part of the compiler implementation. It typically involves a mixture of the pattern descriptions and C interface functions or macros. It is the primitive form of an architecture description language.

A fully retargetable compiler aims at minimizing the coding effort for a range of targets by providing a more friendly machine description interface. Retargetable compilers are important for application specific instruction-set processor (ASIP) (including Digital Signal Processors, or DSP) designs. These cost-sensitive processors are usually designed for specialized application domains and have a relatively narrow market window. It is costly for vendors to customize compiler back ends and other software tools for each one of them. Moreover, for code density or manufacturing cost concerns, these architectures are often designed to be irregular. Their irregularity prevents general-purpose semi-retargetable compilers from generating good quality code, since those compilers lack optimization capability for irregular architectures. As a result, for most ASIP designs, programmers have to fall back on assembly programming, which suffers from poor productivity and portability. As architectures with higher degrees of parallelism are gaining popularity, it is increasingly hard for assembly programmers to produce high quality code. A fully retargetable optimizing compiler is desirable to alleviate this software development bottleneck.

To provide sufficient information for optimizing retargetable compiler back-ends as well as other software tools including assemblers and simulators, architecture description languages (ADL) are developed. ADLs also enable design space exploration (DSE) by providing a formalism for modeling processing elements.

DSE is important to computer architecture design since there is never a single design optimal in every aspect. Designers have to try out and quantitatively estimate or calculate the

performance and various other metrics of interest of a number of alternative designs before reaching an acceptable trade-off. Common metrics of interest include power consumption, transistor count, memory size, and implementation complexity (as it affects time to market). DSE is especially important to the design of ASIPs due to their sensitivity to cost. Figure 2 shows the Y-chart[2] of a typical ASIP DSE flow. In such a flow, a designer or possibly even an automated tool tunes the architecture description. The retargetable compiler compiles the applications, usually a set of benchmark programs, into architecture specific code. The simulator executes the code and produces performance metrics such as cycle count and power consumption. The metrics are analyzed and guide the tuning of the architecture description. The process iterates until a satisfactory cost-effective architectural tradeoff has been found. In an embedded system development environment, DSE also helps to determine the optimal combination of hardwired components and processing elements. For either purpose, DSE over a reasonably large architecture space is prohibitive if significant manual coding effort to port the compiler and simulator for each target is involved. A retargetable software tool chain driven by an ADL is indispensable for DSE.

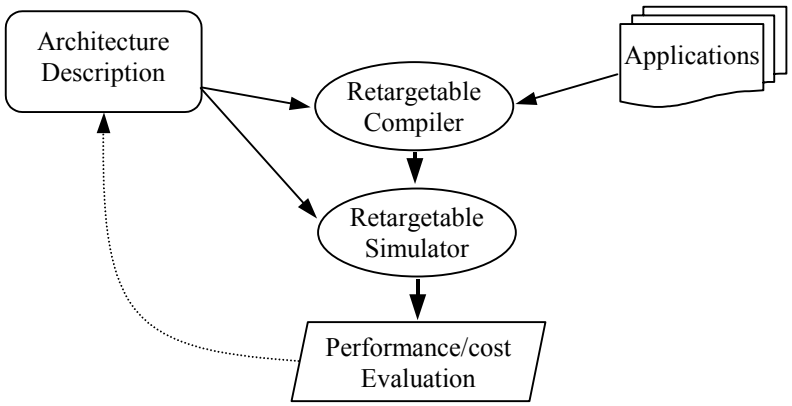


Figure 2 DSE for ASIP design

Over the past decade, a few interesting ADLs have been introduced together with their supporting software tools. These ADLs include MIMOLA[3], UDL/I[4], nML[5], ISDL[6], CSDL[34], Maril[7], HMDES[8], TDL[33], LISA[9,10], RADL[11], EXPRESSION[12] and PRMDL[13]. Usually an ADL driven tool chain contains a retargetable compiler, assembler, dis-assembler and functional simulator. In some cases, a cycle accurate simulator and even micro-architecture synthesis tools are included.

In this chapter, we will start with a survey of the existing ADLs. Next we will compare and analyze the ADLs to highlight their relative strengths and weaknesses. This will be followed by a characterization of the necessary elements that form an ADL, as well as a generic structural organization of the information within and ADL. This study will then form the basis for describing the difficulties faced by ADL designers and their users. Finally, future architecture trends and their implications on requirements for ADLs will be pointed out.

II. ARCHITECTURE DESCRIPTION LANGUAGES

Traditionally, architecture description languages have been classified into three categories: structural, behavioral and mixed. This classification is based on the nature of the information provided by the language.

1. Structural Languages

An important tradeoff in the design of an ADL is the level of abstraction vs. generality. Due to the increasingly vast diversity of computer architectures, it is very difficult to find a high level formalism to capture the interesting characteristics of all types of processors. A common way to obtain more generality is to lower the abstraction level. A lower level of representation can capture concrete structural information in more detail. Register transfer level (RT-level) is a popular abstraction level - low enough for concrete behavior modeling of synchronous digital logic, and high enough to hide gate-level implementation details. It is a formalism commonly used in hardware design. In an RT-level description, data movement and transformation between storage units is specified on a clock cycle basis. A few early ADLs are based on RT-level descriptions.

It is worth noting that the RT-level abstraction should not be confused with the register transfer lists (RT-lists), though both are often abbreviated as RTL. The later is a higher level abstraction used for operation semantics specification. The major difference between the two is the treatment of time. Cycle time is an essential element in an RT-level description. All RT-level events are associated with a specific time stamp and are fully ordered. In contrast, for RT-lists time is not a concern. Only causality is of interest and events are only partially ordered.

MIMOLA

One RT-level ADL is MIMOLA[3], an interesting "computer hardware description language" as well as "high-level programming language" developed at the University of Dortmund, Germany. It was originally proposed for micro-architecture design. Over the years, MIMOLA has undergone many revisions and a number of software tools have been developed based on it. The major advantage of a MIMOLA architecture description is that the single description can be used for hardware synthesis, simulation, test generation and code generation purposes. 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[3]. MIMOLA has also been used by the RECORD[14] compiler as its architecture representation language.

MIMOLA contains two parts: the hardware part where micro-architectures are specified in the form of a component netlist, and the software part where application programs are written in a PASCAL-like syntax.

Hardware structures in MIMOLA are described as a netlist of component modules, each of which is defined at the RT-level level. A simple arithmetic unit module, slightly modified from an example in [3] is:

```
MODULE Alu
     (IN i1, i2: (15:0); OUT outp: (15:0));
```

```
     IN ctr: (1:0));
CONBEGIN
     outp <- CASE ctr OF
          0: i1 + i2 ;
          1: i1 - i2 ;
          2: i1 AND i2 ;
          3: i1 ;
          END AFTER 1;
CONEND;
```

The module declaration style is similar to that of the VHDL hardware description language[15]. The starting line declares the module named Alu. The following two lines describe the module's port names, directions and widths. The port names can be used as variables in the behavior statements as references to the signal values on ports. If there is more than one statement between CONBEGIN and CONEND, they are evaluated concurrently during execution. The AFTER statement in above example describes timing information.

For a complete netlist, connections need to be declared to connect the ports of module instances together, for example[3],

```
CONNECTIONS    Alu.outp -> ACCU.inp
               Accu.outp -> alu.i1
```

The MSSQ code generator will extract instruction set information from the module netlist for use in code generation. It will transform the RT-level hardware structure into the connection operation graph (COG). The nodes in a COG represent hardware operators or module ports, while the edges represent dataflow through the nodes. The instruction tree (I-tree) that maintains a record of instruction encoding is also derived from the netlist and the decoder module. During code generation, the PASCAL like source code is transformed to some intermediate representation. Then pattern matching is done from intermediate code to the COG. Register allocation is done at the same time. The MSSQ compiler directly outputs binary code by looking up the I-tree.

In order for the compiler to locate some important hardware modules, linkage information needs to be provided to identify important units such as the register file, the instruction memory, the program counter, etc. The program counter and instruction memory location can be specified as follows[3]:

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

Where PC and IM are previously declared modules. However, even with the linkage information, it is still a very difficult task to extract the COG and I-trees 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.

MIMOLA's software programming model is an extension of PASCAL. It is different from other high level programming languages, as it allows programmers to designate variables to physical registers and to use hardware components like procedures calls. For example, to use an operation performed by a module called Simd, programmers can write

x: = Simd(y,z);

The special feature helps programmers to control code generation and to map intrinsics effectively. Compiler intrinsics are assembly instructions in the form of library functions. They help programmers to utilize complex machine instructions while avoiding writing inline assembly. Common instrinsics candidates include SIMD (Single Instruction Multiple Data parallel) instructions.

UDL/I

Another RTL hardware description language used for compiler generation is UDL/I[4] developed at Kyushu University in Japan. It describes the input to the COACH ASIP design automation system. A target specific compiler can be generated based on the instruction set extracted from the UDL/I description. The instruction set simulator can also be generated to supplement the cycle accurate RT-level simulator. As in the case of MIMOLA, hints need to be supplied by the designer to locate important machine states such as the program counter and register files. To avoid confusing the instruction set extractor, restrictions were imposed on the scope of supported target architectures. Superscalar and very long instruction word(VLIW) architectures were not supported by the instruction set extractor.

In general, RT-level ADLs enable flexible and precise micro-architecture descriptions. The same description can be used by a series of electronic design automation (EDA) tools including logic synthesis, test generation and verification tools; as well as software tools such as retargetable compilers and simulators at various abstraction levels. However, for users interested in retargetable compiler development only, describing a processor at the RT-level can be quite a tedious process. The instruction set information is buried under enormous micro-architecture detail that only the logic designers care about. Instruction set extraction from RT-level descriptions is difficult without restrictions on description style and target scope. As a result, the generality of the RT-level abstraction makes it very hard for use in the development of efficient compilers. The RT-level descriptions are more amicable to hardware designers than retargetable compiler writers.

2. Behavioral languages

Behavioral languages avoid the difficulty of instruction set extraction by abstracting behavioral information out of the micro-architecture. Instruction semantics are directly specified in the form of RT-lists and detailed hardware structures are ignored. A typical behavioral ADL description is close in form to an instruction set reference manual.

nML

nML is a simple and elegant formalism for instruction set modeling. It was proposed at the Technical University of Berlin, Germany. nML was used by the retargetable code generator CBC[16] and the instruction set simulator Sigh/Sim[17]. It was also used by the CHESS[18] code generator and the CHECKERS instruction set simulator at IMEC. CHESS and CHECKERS were later commercialized[38].

Designers of nML observed that in a real-life processor, usually several instructions share some common properties. Factorization of these common properties can result in a simple and compact representation. Consequently, they used a hierarchical scheme to describe instruction sets. The topmost elements of the hierarchy are instructions, and the intermediate elements are partial instructions (PI). Two composition rules can be used to specify the relationships between the elements: the AND-rule groups several PIs into a larger PI and the OR-rule enumerates a set

of alternatives for one PI. So 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.

The instruction set description of nML utilizes attribute grammars. Each element in the hierarchy has a few attributes and a non-leaf element's attribute values can be computed based on its children's attribute values. Attribute grammar was adopted by ISDL and TDL too.

An example nML instruction semantics specification is provided below:

```
op num_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 | mul | div
op add()
action = {
     temp_dst = temp_dst + temp_src
}
...
```

The num_instruction definition combines three PIs with the AND-rule. The first PI, num_action, is formed through an OR-rule. Any one of add, sub, mul and div is a valid option for num_action. The number of all possible derivations of num_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 num_instruction. Each of num_action's option should have its own *action* attribute defined as its specific behavior, which is referred by the "a.action" line. Besides the *action* attribute shown in the example, two additional attributes *image* and *syntax* can be specified in the same hierarchical manner. *Image* represents the binary encoding of the instructions and *syntax* is the assembly mnemonic.

Though classified as a behavioral language, nML is not completely free of structural information. Any structural information referred to by the instruction set architecture (ISA) must be provided in the description. For example, storage units should be declared since they are visible to the instruction set. Three storage types are supported in nML: RAM, register and transitory storage. Transitory storage refers to machine states that are retained only for a limited number of cycles, for instance, values on busses and latches. The timing model of nML is simple: computations have no delay, only storage units have delay. Instruction delay slots can be modeled by introducing storage units as pipeline registers and by propagating the result through the registers in the behavior specification.

nML models constraints between operations by enumerating all valid combinations. For instance, if a two-register-file architecture provides only one read port of register file A to I-type instructions, then in order to describe a three-source-operand I-type instruction in nML, the user should enumerate all the possible cases including ABB, BAB, BBA and BBB. 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 X cannot directly follow operation Y.

ISDL

The problem of constraint modeling is tackled by ISDL with explicit specification. ISDL stands for Instruction Set Description Language. It was developed at MIT and used by the Aviv compiler[19] and the associated assembler. It was also used to by the retargetable simulator generation system GENSIM[20]. ISDL was designed to assist hardware-software codesign for embedded systems. Its target scope is VLIW ASIPs.

ISDL mainly consists of five sections:

- storage resources,
- instruction word format,
- global definition,
- instruction set,
- constraints.

Similar to nML, storage resources are the only structural information modeled by ISDL. Register files, program counter and instruction memory must be defined for each architecture. The instruction word format section describes the composing fields of the instruction word. ISDL assumes a simple tree-like VLIW instruction model: the instruction word contains a list of fields and each field contains a list of sub-fields. Each field corresponds to one operation. For VLIW architectures, an ISDL instruction corresponds to an nML instruction and an ISDL operation corresponding to a PI of nML.

The global definition section describes the operations' addressing modes. Production rules for tokens and non-terminals can be defined in this section. 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 register operand is

```
Token "RA"[0..15] RA {[0..15];}
```

In this example, following the keyword Token is the assembly format of the operand. Here any one of "RA0" to "RA15" is a valid choice. Next, "RA" is the name of the token for later reference use. The second [0..15] is the return value of the token. It 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. It can be used to describe complex addressing modes. A non-terminal typically groups a few tokens or other non-terminals as its arguments, whose assembly formats and return values can be used when defining the non-terminal. An example non-terminal specification is:

```
Non_Terminal SRC:
     RA {$$ = 0x00000 | RA;} {RFA[RA]} {} {} {} |
     RB {$$ = 0x10000 | RB;} {RFB[RB]} {} {} {};
```

The example shows a non-terminal named SRC. It refers to token RA and RB as its two options. The first pair of braces in each line defines binary encoding as a bit-or result. The "$$" symbol indicates the return value of current non-terminal, a usage likely borrowed from Yacc[21]. The second pair of braces contains the action. Here the SRC operand refers to the data value in the register indexed by the return value of token RA as its action. The following three empty pairs of braces specify side effects, cost, and time. These three exist in the instruction set section as well.

The instruction set section of ISDL describes the instruction set in terms of its fields. For each field, a list of alternative operations can be described. Similar to non-terminals, an operation contains a name, a list of tokens or non-terminals as arguments, binary encoding definition, action definition, side effects and costs. Side effects refer to behaviors such as the setting or clearing of a carry bit. Three types of cost can be specified: execution cycles, encoding size and stall. Stall models the cycle number of pipeline stalls if the next instruction uses the result of the current instruction. The costs affect instruction selection preference. The timing model of ISDL contains two parameters: latency and usage. Latency is the number of instructions to be fetched before the result of the current operation becomes available and usage is the cycle count that the operation spends in its slot. The difference or relationship between the latency and the stall cost is not clear in available publications of ISDL.

The most interesting part of ISDL is its explicit constraint specification. In contrast to nML, which enumerates all valid combinations, ISDL defines invalid combinations in the form of Boolean expressions. This often results in a much simpler constraint specification. It also enables ISDL to capture much more irregular ILP constraints. Recall the example that instruction X cannot directly follow Y, which cannot be modeled by nML. ISDL can describe the constraint as[6]:

```
~(Y *) & ([1] X *, *)
```

The "[1]" indicates a cycle time delay of one fetch bundle. The "~" is a symbol for not and "&" for logical and. Such constraints cannot be specified by nML. Details of the Boolean expression syntax are available in ISDL publications[19].The way ISDL models constraints affects the code generation process: a constraint checker is needed to check if the selected instructions meet the constraint. Iterative code generation is required in case of checking failure.

Overall, ISDL provides the means for compact, hierarchical instruction set specification of instruction sets. Its Boolean expression based constraint specification helps to model irregular instruction level parallelism (ILP) effectively. A shortcoming of ISDL is that the simple tree-like instruction format forbids the description of instruction sets with multiple encoding formats.

CSDL

CSDL stands for computer system description languages. It is a family of machine description languages deveoloped for the Zehpyr compiler infrastructure mainly at University of Virginia. The CSDL family currently includes CCL, a function calling convention specification language, SLED, a formalism describing instruction assembly syntax and binary encoding and λ-RTL, a register transfer lists language for instruction semantics description.

SLED stands for Specification Language for Encoding and Decoding. It was developed as part of the New Jersey Machine-Code toolkit, which assists programmers to build binary editing tools. A retargetable linker mld and a retargetable debugger ldb have been reported based on the toolkit.

Similar to ISDL, SLED uses a hierarchical model for machine instruction: an instruction is composed of one or more *tokens* and each token is composed of one or more *fields*. The construct *patterns* helps to group the fields together and to bind them to binary values. The directive *constructors* helps to connect the fields into instruction words. A detailed description of the syntax can be found in SLED publications[35].

SLED does not assume a single format of the instruction set. So, it is flexible enough to describe the encoding and assembly syntnax of both reduced instruction set computer(RISC) and

complex instruction set computer(CISC) instruction sets. Same 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.

The vpo(very portable optimizer) in Zephyr system provides the capability of instruction selection, instruction scheduling and global optimization. Instruction sets are represented in RT-lists form in vpo. The raw RT-lists form is verbose. To reduce description effort, λ-RTL was developed. A λ-RTL description will be translated into RT-lists for the use of vpo.

According to the developers[36], λ-RTL is a high order, strongly typed, polymorphic, pure functional language based largely on Standard ML[40]. 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. The functional language has several elegant properties[36], which are beyond the scope of this chapter.

CSDL descriptions describe only storage units as hardware resources. Timing information such as operation latencies are not described. So the languages themselves do not supply enough information for VLIW processor code generation and scheduling. They are more suitable for conventional general-purpose processor modeling.

The behavioral languages share one common feature: hierarchical instruction set description based on attribute grammars[39]. This feature greatly helps to simplify the instruction set description by exploiting the common components between operations. However, the lack of detailed pipeline and timing information prevents the use of these languages as an extensible architecture model. Information required by resource-based scheduling algorithms cannot be obtained directly from the description. Also, it is impossible to generate cycle accurate simulators based on the behavioral descriptions without some assumptions on the architecture's control behavior, i.e. an implied architecture template has to be used.

3. Mixed languages

Mixed languages extend behavioral languages by including abstracted hardware resources in the description. As in the case of behavioral languages, RT-lists are used in mixed languages for semantics specification.

Maril

Maril is a mixed architecture description language used by the retargetable compiler Marion[7]. It contains both instruction set information as well as coarse-grained structural information. Maril descriptions were intended for use in code generation for RISC style processors only. So unlike the case with ISDL, there is no distinction between instruction and operation in Maril. The structural information contains storage units as well as highly abstracted pipeline units. Compiler back-ends for the Motorola 88000[44], the MIPS R2000[45] and the Intel i860[46] architectures were developed based on Maril descriptions.

Maril contains three sections:

1. Declaration

   The declaration section describes structural information, such as register files, memory and abstracted hardware resources. Abstracted hardware resources include pipeline stages and data buses. The resources are to be used for reservation table description. In the compiler

community, the term *reservation table* has been used as a mapping from operation to architecture resources and time. It captures the operation's resource usage at every cycle starting with the fetch. The reservation table is often a one-to-many mapping since there can be multiple alternative paths for one instruction.

Besides hardware structures, the declaration section also contains information such as the range of immediate operands or relative branch offset. This is necessary information for the correctness of generated code.

2. Runtime model

   The runtime model section specifies the conventions of generated code – this deals mostly with the function calling convention. This section is the parameter system of the Marion compiler. It is not intended to be a general framework for calling convention specification, which is complex enough to qualify for a description language itself[22]. Calling conventions are not the primary interest of this chapter and will not be discussed further.

3. Instruction

   The instruction section describes the instruction set. Each instruction definition in Maril contains five parts. The first part is the instruction mnemonics and operands, which can be used in the assembly output format. The optional second part declares data type constraints of the operation for code selection use. The third part describes, for each instruction, a single expression. The patterns used by the tree-pattern matching code-generator are derived from this expression. A limitation on the expression is that it can contain only one assignment. This constraint is reasonable since the code generator can usually handle tree-patterns only. However, it forbids Maril from describing instruction behavior such as side effects on machine flags, or post-increment memory references, since those involve multiple assignments. Consequently, a Maril description is generally not accurate enough for use by a functional simulator. The fourth part of the instruction declaration is the reservation table of the instruction. The abstracted resources used in each cycle can be described here, starting from instruction fetch. The last part of an instruction specification is a triple of (cost, latency, slot). Cost is used to distinguish actual operations from dummy instructions, which are used for some type conversions. Latency is the number of cycles before the result of the instruction can be used by other operations. Slot specifies the delay slot count. Instruction encoding information is not provided in Maril.

   An example integer Add instruction definition of Maril is[7]:

```
%instr    Add r, r, r  (int)
          {$3 = $1+$2;}
          {IF; ID; EX; MEM; WB} (1,1,0)
```

   The operands of the Add instruction are all general-purpose registers, as denoted by the r's. The ";" in the reservation table specification delimits the cycle boundary. The Add instruction will go through five pipeline stages in five cycles. It has a cost of one, takes one cycle and has no delay slot.

In general, Maril is designed for use in RISC processor code generation and scheduling. Some of its information is tool specific. It cannot describe a VLIW instruction set, nor does it provide enough information for accurate simulation. It does not utilize a hierarchical instruction set description scheme as is done by most behavioral languages. Nonetheless, compared with the behavioral languages, it carries more structural information than just storage units. The structural

resource based reservation table description enables resource-based instruction scheduling, which can bring significant performance improvement for deeply pipelined architectures.

HMDES

Another language with emphasis on scheduling support is HMDES[8], which was developed at UIUC for the IMPACT research compiler. IMPACT mainly focuses on Instruction Level Parallelism (ILP) exploration. As a result, the instruction's reservation table information is the major content of HMDES. Although there is no explicit notion of instruction or operation bundle in HMDES, it can be used for VLIW scheduling purpose by representing VLIW issue slots as artificial resources. IMPACT is interested in wide issue architectures in which a single instruction can have numerous scheduling alternatives. For example, if in an eight-issue architecture an *Add* instruction can go through any of the 8 decoding slots into any of the 8 function units, there are in total 64 scheduling alternatives for it. To avoid laboriously enumerating the alternatives, an and-or tree structure is used in HMDES to compress reservation tables. Figure 3 shows the reservation table description hierarchy of HMDES. The leaf node resource usage is a tuple of (resource, time).

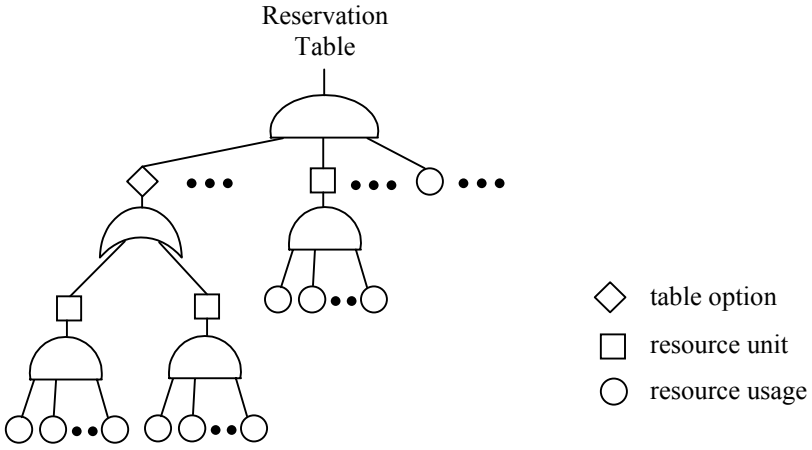


Figure 3 HMDES reservation table description hierarchy

A special feature of HMDES is its preprocessing constructs. C-like preprocessing capabilities such as file inclusion, macro expansion and conditional inclusion can be used. Complex structures such as loop and integer range expansion are supported too. The preprocessing does not provide extra description power, but helps to keep the description compact and easy to read.

Instruction semantics, assembly syntax and binary encoding information are not part of HMDES since IMPACT is not designed to be a fully retargetable code generator. It has a few manually written code generation back ends. After machine specific code is generated, IMPACT queries the machine database built through HMDES to do ILP optimization.

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. A detailed description of the interface between the machine description system and the compiler back end can be found in the documentation for Elcor[23].

TDL

TDL stands for target description language. It has been developed at Saarland University in Germany. The language is used in a retargetable postpass assembly-based code optimization system called PROPAN[33].

In PROPAN, a TDL description is transformed into a parser for target specific assembly language and a set of ANSI C files containing data structures and functions. The C files are included in the application as a means of generic access to architecture information. A TDL description contains 4 parts:

- Resource section

  Storage units such as register files and memory have built-in syntax support in this section. Moreover, TDL allows the description of the cache, which partially exposes the memory hierarchy to the compiler and allows for more optimization opportunities. Function units are also described in this section. TDL also provides flexible syntax support for the user to define generic function units which do not belong to any previous category.

- Instruction set section

  Same to behavioral languages, TDL's organization is based on attribute grammar[39]. TDL supports VLIW architectures, so it distinguishes the notions of operation and instruction. No binary encoding information is provided TDL.

  An adapted example of TDL operation description TDL publications[33] is given here:

```
DefineOp IAdd "%s=%s+%s" {
        dst1="$1" in {gpr}, src1="$2" in {gpr}, src2="$3" in {gpr}},
        {ALU1(latency=1, slots=0, exectime=1) | ALU2(latency=1, slots=0,exectime=2);
            WbBus(latency=1)},
        {dst1 := src1+src2;}.
```

  The operation definition contains a name, assembly format, a list of pre-defined attributes such as source and destination operand position and type, reservation table and semantics in RT-lists form. In this example, the destination operand $1 corresponds to the first "%s" in the assembly format and is from the register file named ireg. The operation can be scheduled on either function unit ALU1 or ALU2. It will also use the result write back bus resource. The operation performs addition. A very detailed TDL version RT-lists semantics description rule can be found in [50].

  This section also contains operation class definition which groups operation into groups for the ease of reference. Instruction word formats can be described using the operation classes. For instance, the declaration[50] "InstructionFormat ifo2 [opclass3, opclass4];" means that the instruction format ifo2 contains one operation from opclass3 and one from opclass4.

  Similar to ISDL, TDL also provides a non-terminal construct to capture common components between operations.

- Constraints section

  Recall that ISDL has a constraint specification section. The Boolean expression used in ISDL is based on lexical elements in the assembly instruction. TDL also uses Boolean

expressions for constraint modeling, but based on the explicitly declared operation attributes from the above sections.

A constraint definition includes a premise part followed by a rule part, separated by a colon. An example definition from [33] is:

```
op1 in {MultiAluFixed} & op2 {MultiMulFixed} :
        !(op1 && op2) | op1.src1 in {iregC} & op1.src2 in {iregD}
        & op2.src1 in {iregA} & op2.src2 in {iregB}
```

The example specifies the constraint for two operations from MultiAluFixed class and MultiMulFixed class, respectively. Either they do not coexist in one instruction word or their operands have to be in specific register files. The constraint specification is as powerful as that of ISDL, but in a much cleaner syntax.

The Boolean expression will be transformed to an integer linear programming constraint to be used in the PROPAN system.

- Assembly section

  This section describes the lexical elements in the assembly file including comment syntax, operation and instruction delimiters and assembly directives.

On the whole, TDL provides a well-organized formalism for VLIW DSP assembly code description. Preprocessing is supported. Description of various hardware resources including caches is supported. The RT-lists description rules are exhaustively defined. A hierarchical scheme is used to exploit common components among operations. Both resource based (function units) and explicit Boolean expression based constraint modeling mechanisms are provided. However, the timing model and reservation model of TDL seems to be restricted by the syntax. A user cannot describe a reservation table with multiple pipeline stages involved. Also there is no way for a user to flexibly specify operand latencies — the cycle time when operands are read and written. These restrictions prevent the use of TDL for accurate RISC archiecture modeling. Another limitation is that register ports or data transfer paths are not explicitly modeled in the resource section. The two are often resource bottlenecks in VLIW DSPs. But overall, the limitations are minor and can be overcome with extensions to current TDL.

A related but more restrictive machine description formalism for assembly code analysis and transformation applications can be found in the SALTO framework[48].

EXPRESSION

A problem of an explicit reservation table description in the above mixed languages is that it may not be natural and intuitive enough. A translation from pipeline structures to abstract resources has to be done by description writers. Such manual translation can be annoying in cases of DSE. The architecture description language EXPRESSION[12] avoids human effort in doing the translation. Instead, it describes a netlist of pipeline stages and storage units directly and automatically generates reservation tables based on the netlist. In contrast to MIMOLA's fine-grained netlist, EXPRESSION uses a much coarser representation. A netlist style specification is friendly to the architects and makes graphical input possible.

EXPRESSION was developed at University of California at Irvine. It is used by the research compiler EXPRESS[24], and the research simulator SIMPRESS[25] developed there. A GUI front end for EXPRESSION has also been developed[25]. Expression takes a balanced view

of behavioral and structural descriptions and consists of a distinct behavioral section and a structural section.

The behavioral section is similar to that of ISDL in that it distinguishes instructions and operations. But it does not cover assembly syntax and binary encoding. Nor does it use a hierarchical structure for instruction semantics. The behavioral section contains three subsections: operation, instruction and operation mapping. The operation subsection is in the form of RT-lists. But detailed semantics description is not clearly defined. A useful feature of EXPRESSION is that it groups similar operations together for the ease of later reference. Operation addressing modes are also defined in the operation subsection.

The instruction subsection describes the bundling of operations that can be issued in parallel. Instruction width, slot widths and function units that correspond to the slots are declared in this section. The information is essential for VLIW modeling.

The operation mapping subsection specifies the transformation rules when generating code. Two types of mapping can be specified: mapping from a compiler's intermediate generic operations to target specific operations, and mapping from target specific operations to target specific operations. The first type of mapping is used for code generation. The second type is required for optimization purposes. Predicates can be specified for conditional mappings. Providing the mapping subsection in EXPRESSION makes the code generator implementation much easier, but it also makes the EXPRESSION language tool dependent.

The most interesting part of EXPRESSION is its netlist based structural specification. It contains three subsections: component declaration, path declaration and memory sub-system declaration.

In the component subsection, coarse-grained structural units such as pipeline stages, memory controllers, memory banks, register files and latches are specified. Linkage resources including ports and connections can also be declared in this part. A pipeline stage declaration for the example architecture in Figure 4 is:

```
(DECODEUnit  ID
       (LATCHES  decodeLatch)
       (PORTS  ID_srcport1  ID_srcport2)
       (CAPACITY 1)
       (OPCODES  all)
       (TIMING  all  1)
)
```

The example shows the declaration of the instruction decoding stage. The LATCHES statement refers to the output pipeline register of the unit. The PORTS statement refers to the abstracted data ports of the unit. The ID unit has two register read ports. CAPACITY describes the number of instructions that the pipeline stage can hold at the same time. Common function units have capacity of one, while fetching and decoding stages of VLIW or superscalar processors can have capacity as wide as its issue width. OPCODES describes the operations that can go through this stage. "All" here refers to an operation group containing all operations. Timing is the cycle count that operations spend in the unit. Each operation takes one cycle. Timing can also be described on a per operation basis.

In the path subsection, pipeline paths and data paths between pipeline stages and storage units are specified. This part connects the components together into a netlist. The pipeline path declaration stitches together the pipeline stages to form a directed acyclic pipeline graph, in

which the pipeline stages are vertices and paths are directed edges. An example path declaration for a the simple DLX architecture[26] shown in Figure 4 is:

```
(PIPELINE FE ID EX MEM WB)
(PIPELINE FE ID F1 F2 F3 F4 WB)
```

Recall that the OPCODES attribute of pipeline stages declares the operations that can go through each stage. So for each operation, possible paths that it can go through can be inferred by looking for paths whose every node can accommodate the operation. Each path corresponds to a scheduling alternative. Since the time spent by each operation in each stage is specified in the TIMING declaration in the component subsection, a reservation table can be generated from the paths. Register file ports, and data transfer path usage information can also be derived from the operation's operands and the connections between the pipeline stages and storage units. The ports and data transfer paths can be resources in the reservation tables too.

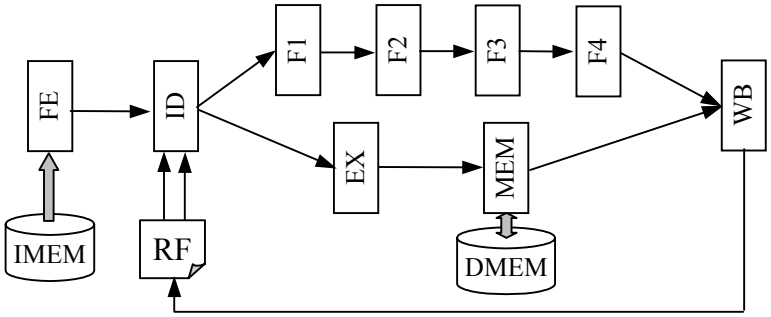


Figure 4 Example DLX pipeline

Compared with the explicit reservation table specification style of HMDES and Maril, EXPRESSION's netlist style is more attractive. However, the way that EXPRESSION derives reservation tables[12] excludes cases in which an operation can occupy two pipeline stages at the same cycle, which may happen in floating point pipelines or when artificial resources[27] need to be introduced for irregular ILP constraint modeling.

The last structural part is the memory model. A parameterized memory hierarchy model is provided in EXPRESSION. Memory hierarchy specification is important, especially as we go from single processing element descriptions toward system-on-chip description. Advanced compilers can make use of the memory hierarchy information to improve cache performance. EXPRESSION and TDL are the only ADLs that address memory hierarchy.

In general, EXPRESSION captures the data path information in the processor. As with all the above mixed languages, the control path is not explicitly modeled. Thus, it actually does not contain complete information for cycle accurate simulation. An architecture template needs to be used to supplement the control path information for simulation purposes. The behavioral model of EXPRESSION does not utilize hierarchical operation specification. This can make it tedious to specify a complete instruction set. The VLIW instruction composition model is simple. It is not clear if inter-operation constraints such as sharing of common fields can be modeled. Such constraints can be modeled by ISDL through cross-field encoding assignment.

LISA

The emphasis of the behavioral languages is mostly on instruction set specification. Beyond that, mixed languages provide coarse-grained data path structural information. However,

control path specification is largely ignored by behavioral languages, as well as by the above mixed languages. This is probably due to the lack of formalism in control path modeling. Complex control related behaviors such as speculation and zero-overhead loops are very difficult to model cleanly. Control behavior modeling is important for control code generation, e.g. branch and loop code generation, as well as for cycle-accurate simulation. As the pipeline structures of high-end processors get increasingly complicated, branching and speculation can take up a significant portion of program execution time. Correct modeling of control flow is crucial for accurate simulation of such behaviors, whose feedback can provide important guidance to ILP compiler optimizations.

The architecture description language LISA[9] was initially designed with a simulator centric view. LISA stands for Language of Instruction Set Architecture. It was developed at Aachen University of Technology in Germany. LISA's development accompanies that of a production quality cycle-accurate simulator[10]. A compiler based on LISA is undergoing development at this moment.

Compared with all above languages, LISA is closer to an imperative programming language. Control related behavior such as pipeline flush or stall can be specified explicitly. These operation primitives provide the flexibility to describe a diversity of architectural styles such as super-scalar, VLIW and SIMD.

LISA contains mainly two types of top-level declarations: resource and operation. Resource declarations cover hardware resources including register files, memories, program counters, pipeline stages, etc. The pipeline definition declares all the possible pipeline paths that the operations can go through. A pipeline description corresponding to the example shown in Figure 4 is as follows:

```
PIPELINE pipe = {FE; ID; EX; MEM; WB}
PIPELINE pipe_fp = {FE; ID; F1; F2; F3; F4; WB}
```

Similar to Maril, the ";"'s above are used to delimit the cycle boundary.

Machine operations are defined stage by stage in LISA. OPERATION is the basic unit defining operation behavior, encoding, and assembly syntax on a per stage basis. At some pipeline stages such as the instruction fetch or decoding stage, several operations share common behavior. LISA exploits the commonality by grouping the specifications of similar operations into one. As a slightly modified version from an example in [10], the decoding behavior for arithmetic operations in the DLX ID stage can be described as:

```
OPERATION arithmetic IN pipe.ID {
     DECLARE {
          GROUP opcode={ADD || ADDU || SUB || SUBU}
          GROUP rs1, rs2, rd = {fix_register};
     }
     CODING {opcode rs1 rs2 rd}
     SYNTAX {opcode rd "," rs1 "," rs2}
     BEHAVIOR {
          reg_a = rs1;
          reg_b = rs2;
          cond = 0;
     }
     ACTIVATION {opcode, writeback}
}
```

The example captures the common behavior of the four arithmetic operations ADD, ADDU, SUB and SUBU in the decoding stage, as well as their assembly syntax and binary encoding.

To get the complete behavior of an operation, its behavior definition in other pipeline stages has to be taken into account. As shown in the example, an operation declaration can contain several parts:

- DECLARE, where local identifiers are specified.
- CODING, where the binary encoding of the operation is described.
- SYNTAX, where the assembly syntax of the operation is declared.
- BEHAVIOR, where the exact instruction semantics including side effects are specified in a C-like syntax.
- ACTIVATION, where the dependence relationship of the other OPERATIONs is specified.

An extra SEMANTICS part can be defined too. While both SEMANTICS and BEHAVIOR define the function performed by an operation, SEMANTICS is reserved for use by the code generator during mapping and behavior is for use in simulation. Similar redundancy can be found in EXPRESSION too.

ACTIVATION is part of the execution model of LISA. In LISA, one OPERATION can activate another OPERATION. The firing time of the activated OPERATION is dependent on the distance between the two OPERATIONs in the pipeline path. The above example activates one of the opcodes, which is declared in the EX stage. Since EX directly follows ID, it will execute in the following cycle. One option ADD can be declared as[10]:

```
OPERATION ADD IN pipe.EX
{
      CODING {0b001000}
      SYNTAX {"ADD"}
      BEHAVIOR {alu = reg_a + reg_b;}

}
```

Each LISA description contains a special main OPERATION, which will be activated at every cycle. It serves as a kernel loop in simulation. Pipeline control functions including shift, stall and flush can be used in the main loop.

One advantage of LISA is that the user can describe detailed control path information with the activation model. Control path description is important in generating a real cycle accurate simulator. A fast and accurate simulator for Texas Instrument's TMS320C6201 VLIW DSP[28] based on LISA has been reported[10].

In order to use LISA for retargetable code generation, the instruction set has to be extracted. This is an easier job than instruction extraction from RT-level languages. The semantics of most arithmetic, logical and memory operations can be directly obtained by combining their OPERATION definitions. For complicated and hard-to-map instruction behaviors, the SEMANTICS keyword can be used as a supplement.

Miscellaneous

A language similar to LISA is RADL (Rockwell Architecture Description Language)[11]. It was developed at Rockwell, Inc. as a follow-up of some earlier work on LISA. The intention of RADL is to support cycle accurate simulation only. Control related APIs such as stall and kill can be used in RADL descriptions too. However, there is no available information on the simulator utilizing RADL.

Besides RADL, another architecture description language developed in the industry is PRMDL[13]. PRMDL stands for Philips Research Machine Description Language. It is not intended to cover a wide range of architectures. Instead, it focuses exclusively on clustered VLIW architectures.

III. ANALYSIS OF ARCHITECTURE DESCRIPTION LANGUAGES

1. Summary of ADLs

Architecture description languages as a research area is far from being mature. In contrast to the situation with other computer languages such as programming languages or hardware description languages, there is neither any kind of standard nor any dominating ADL. Most of the ADLs are designed specifically for software tools being developed by the same group of designers and are thus tightly affiliated with them. The only languages that have been used by multiple compilers are MIMOLA and nML. However, they have limited popularity, and no recent adopters. The stage is wide open for new efforts and it is not surprising that new ADLs are constantly emerging.

An apparent reason for the lack of a standard ADL is the lack of formalism in general computer architecture modeling. Modern computer architectures range from simple DSP/ASIPs with small register files and little control logic to complex out-of-order superscalar machines[26] with deep memory hierarchy and heavy control and data speculation. As semiconductor processes evolve, more and more transistors are being squeezed into a single chip. The drive for high-performance leads to the birth of even more sophisticated architectures, such as multi-threading processors[26]. It is extremely hard for a single ADL to describe the vast range of computer architectures accurately and efficiently. A practical and common approach is for the designers to use a high-level abstraction committed to a small range of architectures. Usually one or more implicit architecture templates are behind a high level ADL to bridge the gap between the abstraction and the physical architecture.

A second reason is probably the different design purposes of ADLs. Some ADLs were originally designed as a hardware description language, e.g. MIMOLA and UDL/I. The major design goal of those languages is accurate hardware modeling. Cycle accurate simulators and hardware synthesis tools are natural outcomes of such languages. It is non-trivial to extract instruction set information from these languages for use in a compiler. Some other ADLs were initially developed to be high-level processor modeling languages, such as nML, CSDL, TDL, LISA and EXPRESSION. Important design considerations of these languages are general coverage over an architecture range and support for both compilers and simulators. Due to the divergent needs of the compiler and simulator, a language usually can provide good description support for only one at a time. Furthermore, designers of the ADLs are often interested in the research value of the software tools. They have interests in different parts of the compiler or

simulator. As a result, the ADLs are usually biased toward the parts of interest. It is hard for a single ADL to satisfy the needs of all researchers.

The remaining ADLs were developed as a configuration system for its accompanying software tool, e.g. Maril, HMDES and PRMDL. In some sense those languages can be viewed as a by-product of the software tools, which were designed as compiler research infrastructure or as architecture exploration tools. The goal of these languages is flexibility within the configuration space. Generality is a secondary concern though it is desirable for extensibility considerations. However, if the generality of an ADL significantly exceeds that of its accompanying software tools, it may well become a source of ambiguity. Moreover, generality often means less efficiency in modeling. There is little value in describing things that the software tools cannot support after all.

Thus we see that ADLs can differ along many dimensions. Different ADLs are designed for different purposes, at different abstraction levels, with emphasis on different architectures of interest, and with different views on the software tools. They reflect the their designer's view of computer architecture and software tools.

The design of an ADL usually accompanies the development of software tools. The overall design effort on an ADL is often a small fraction of the development effort of the software tools utilizing the ADL. So very often the developers of new software tools tend to design new ADLs. Though so many ADLs have been introduced in the past decade, as mentioned earlier, this is not yet a mature research field. However, this is likely to change in the near future. As programmable parts are playing an increasingly important role in the development of systems, this field is attracting the attention of more and more researchers and engineers.

Table 1 summarizes and compares the important features of various ADLs. A few entries in the table were left empty, either because information is not available (no related publication) or not applicable. The ( )'s in some entries indicates support with significant limitation.

	MIMOLA	UDL/I	nML	ISDL	SLED/ λ-RTL	Maril	HMDES	TDL
category	HDL	HDL	behavioral	behavioral	behavioral	mixed	mixed	mixed
compiler	MSSQ, Record	COACH	CBC, CHESS	Aviv	vpo	Marion	IMPACT	PROPAN
simulator	MSSB/U	COACH	Sign/Sim, Checkers	GENSIM				
behavioral representation	RT-level	RT-level	RT-lists	RT-lists	RT-lists	RT-lists		RT-lists
hierarchical behavioral representation			yes	yes	yes	no	yes	yes
structural representation	netlist	netlist				resource	resource	resource
ILP compiler support			yes	yes		yes	yes	yes
cycle simulation support	yes	yes	(yes)	(yes)	no	no	(yes)	no
control path	yes	yes	no	no	no	no	no	no
constraint model				Boolean		resource	resource	resource, Boolean

other features							preprocessing support	preprocessing support	

Table 1 Comparison between ADLs

2. Basic Elements of an ADL

Modern retargetable compiler back ends normally consist of three phases: code selection, register allocation and operation scheduling. The phases and various optimization steps[41] may sometimes be performed in a different order and in different combinations. The first two phases are always necessary to generate working code. They require operation semantics and addressing mode information. Operation scheduling tries to minimize potential pipeline hazards in code execution, including data, structural and control hazards[26]. Scheduling also helps to pack operations into instruction words for VLIW processors and tries to minimize unused empty slots. It is traditionally viewed as an optimization phase. As deeper pipelines and wider instruction words are gaining more popularity in processor designs, scheduling is increasingly important to the quality of a compiler. A good ILP scheduler can improve performance as well as reduce code size greatly. So we assume that it is an essential part of the compiler back end in this chapter.

The data model of a common scheduler data model contains two basic elements: operand latency and reservation table. The first element models data and control hazards and is used to build a directed acyclic dependency graph. The second element models structural hazards between operations. The instruction word packing problem does not always fit into the data model directly. A solution is to convert the packing conflicts to artificial resources and let conflicting operations require the use of such a resource at the same time. By augmenting the reservation table with artificial resource usages, we prevent the scheduler from scheduling conflicting operations simultaneously. Thus we can unify the packing problem into normal reservation table based scheduling.

Each machine operation performs some type of state transition in the processor. A precise description of the state transition has to include three elements: what, where and when. Correspondingly, information required by a compiler back end contains three basic elements: behavior, resource and time. Here, behavior means semantic actions including reading of source operands, calculation and writing of destination operands. Resource refers to abstracted hardware resources used to model structural hazards or artificial resources used to model instruction packing. Common hardware resources include pipeline stages, register file ports, memory ports and data transfer paths. The last element, viz. time, is the cycle number when the behavior occurs and when resources are used. It is usually described relative to the operation fetch time or issue time. In some cases, phase number can be used for even more accurate modeling.

With the three basic elements, we can easily represent each machine operation in a list of triples. Each triple is in the form of (behavior, resource, time). For example, an integer Add operation in the pipeline depicted by Figure 4 can be described as:

(read operand reg[src1], through register file port a, at the 2nd cycle since fetch);
(read operand reg[src2], through register file port b, at the 2nd cycle since fetch);
(perform addition, in alu, at the 3rd cycle since fetch);
(write operand reg[dst], through register file write port, at the 5th cycle since fetch).

From the triple list, the compiler can extract the operation semantics and addressing mode by combining the first elements in order. It can also extract operand latency and reservation table information for the scheduler. Operand latency is used to schedule against data hazards, while reservation table can be used to schedule against structural hazard.

The triple list is a simple and general way of operation description. In practice, some triples may sometimes be simplified into tuples. For instance, most architectures contain sufficient register read ports so that no resource hazard can ever occur due to them. As a result, the scheduler does not need to take the ports into consideration. One can simplify the first two triples by omitting the resource elements. On the other hand, when there exist resources that may cause contention, but there is no visible behavior associated with the resource, the behavior can be omitted. In the same integer Add example, if for some reason the MEM stage becomes a source of contention, one may want to model it as a tuple of (MEM stage, at the 4^{th} cycle since fetch).

The triple/tuple list can be found in all mixed languages in some form. For instance, the HMDES language is composed of these two types of tuples: the (behavior, time) tuple for operand latency and the (resource, time) tuple for reservation table. In LISA, operations were described in terms of pipeline stages. Each pipeline stage has an associated time according to its position in the path. So a LISA description contains the triple list in some form too.

The final pieces of information left out from the triple list that are needed by the compilers are assembly syntax and binary encoding. These two can be viewed as attached attributes of the operation behavior. It is relatively straightforward to describe them.

3. Structuring of an ADL

It is possible to directly turn the aggregation of triple lists into an ADL. However, describing a processor based on such an ADL can be a daunting task if it is to be written by humans. A typical processing element can contain around 50~100 operations. Each of the operations may have multiple issue alternatives, proportional to the issue width of the architecture. Each issue alternative corresponds to one triple list, whose length approximates the pipeline depth. As a result, the total number of tuples is about the product of operation count, issue width and pipeline depth, which may be of the order of thousands. Moreover, as mentioned earlier, artificial resources can be used to model instruction packing or irregular inter-operation constraints. A raw triple list representation requires the user to do the artificial resource formation prior to the description. This process can be laborious and error-prone for humans.

So the task of an ADL design is to find a simple, concise and intuitive organization to capture the required information. Conceptually, a complete ADL should contain three parts.

- Behavioral part

This part contains operation addressing modes, operation semantics, assembly mnemonics and binary encoding. This corresponds to the first element in the triple.

Behavior information is most familiar to compiler writers and can be found directly in architecture reference manuals. As a result, in many ADLs behavioral information constitutes an independent section by itself, if not all of the ADL.

In common processor designs, operations in the same category usually share many common properties. For instance, usually all 3-operand arithmetic operations and logic

operations share the same addressing modes and binary encoding formats. They differ only in opcode and semantics. Exploitation of the commonalities among operations can make the description compact, as has been demonstrated by nML and ISDL among others. Both languages adopt hierarchical description schemes, under which common behaviors are described at the root of the hierarchy while specifics are kept in the leaves.

Besides commonality sharing, another powerful capability of hierarchical description is factorization of sub-operations. A single machine operation can be viewed as the aggregation of several sub-operations, each of which has a few alternatives. Take for example, the Load operation of TI's TMS320C6200 family DSP[26]. The operation contains two sub-operations, the load action and the addressing mode. Five versions of load action exist: load byte, load unsigned byte, load half word, load unsigned half word and load word. The addressing mode can further be decomposed into two parts: the offset mode and the address calculation mode. Two offset modes exist: register offset and constant offset. For the address calculation mode, six alternatives are possible: plus offset, minus offset, pre-increment, pre-decrement, post-increment and post-decrement. Overall, the single Load operation contains 5*2*6=60 versions. Under a flat description scheme, it could be a nightmare when one finds a typographical error in the initial version after cutting, pasting and modifying 59 times.

In contrast to the geometric complexity of a flat description, use of a hierarchical description scheme factorizes sub-operations and keeps the overall complexity linear. The resulting compact description is much easier to verify and modify. Conciseness is a very desirable feature for ADLs.

- Structural part

This part describes the hardware resources. It corresponds to the second element in the triple representation. Artificial resources may also be included in this part. The level of abstraction in this part can vary from fine-grained RT-level to coarse-grained pipeline stage level, though for use in a compiler a coarse-grained description is probably more suitable. Two schemes exist for coarse-grained structural descriptions: resource based and netlist based.

Maril, TDL and HMDES utilize the resource based description scheme. The advantage of the resource-based scheme is flexibility in creating resources. When creating a resource, the description writer does not have to worry about the corresponding physical entity and connection with other entities. The resource may comfortably be an isolated artificial resource used for constraint modeling.

In contrast, EXPRESSION and PRMDL utilize the netlist-based scheme. A significant advantage of a netlist-based description is its intuitiveness. A netlist is more familiar to an architect than a list of unrelated abstract resource names. A netlist description scheme also enables friendly GUI input. A reservation table can be extracted from the netlist through some simple automated translation. The disadvantage is that the modeling capability may not be as flexible as the resource-based scheme. Architectures with complex dynamic pipeline behaviors are hard to model as a simple coarse-grained netlist. Also, netlists of commercial superscalar architectures may not be available to researchers.

Based on this comparison, the resource-based scheme seems more suitable as an approximation to complex high-end architecture descriptions, while the netlist-based scheme is better suited as an accurate model for simple ASIP/DSP designs whose netlists are available and whose control logic is minimal.

- Linkage part

  This part completes the triple. The linkage information maps operation behavior to resources and time. It is usually not an explicit part of ADLs.

  Linkage information is represented by different ADLs in different ways. There is no obvious advantage of one way over another. In Maril, TDL or HMDES, linkage is described in the form of an explicit reservation table. For each operation, the resources it uses and the time of each use is enumerated. HMDES further exploits the commonality of resource uses through a hierarchical representation. In EXPRESSION, the linkage information is expressed in multiple ways: operations are mapped to pipeline stages by the OPCODES attribute associated with the pipeline stages, while data transfer resources are mapped to operations according to the operands and netlist connections. Grouping of operations into groups helps to simplify the mapping in EXPRESSION.

In summary, the desirable features of an architecture description include simplicity, conciseness, and generality. These features may be contradictory to each other. A major task of the ADL design process is to find a good tradeoff among the three for the interesting architecture family in consideration. A good ADL should also not be tied into internal decisions made by the software tools and also minimize redundancy.

4. Difficulties

The triple list model looks simple and general. However, real world architectures are never so simple and straightforward to describe. ADL designers are constantly plagued with the tradeoff of generality and abstraction level: low abstraction level brings more generality, while high abstraction level provides better efficiency. As for the software tools using the ADL, compilers prefer a high level of abstraction while cycle accurate simulators expect concrete detailed micro-architecture models. It is hard to find a clean and elegant representation satisfying both. Besides that, idiosyncrasies of various architectures make general ADL design an even more agonizing process. When modeling a new processor, designers have to evaluate the ability to accommodate it without disrupting the existing architecture model supported by the ADL. If this is not possible, then some new language constructs may need to be added. If the new construct is not orthogonal to the existing constructs, then the entire language may have to be revised, which also means that all written architecture descriptions and the parser need to be revised too. The emergence of new “weird” architectural features keeps the designers busy struggling with evaluations, decisions and rewriting. Comparatively speaking, tool specific ADL designers are much better off since they can comfortably exclude those architectures if the tools cannot handle them.

We list a few common problems faced by ADL designers and users below. Most of them do not have a clean solution and tradeoffs have to be made depending on the specific needs of each case.

- Ambiguity

  The most common problem of an ADL is ambiguity, or inability to define precise behavior. As a side effect of abstraction, ambiguity exists in most ADLs, especially in modeling control related behavior. Take the common control behavior of interlocking for example. Superscalar architectures have the capability to detect data and control hazards and

stall the pipeline when a hazard exists. VLIW architectures, on the other hand, may not have the interlocking mechanism. The difference among the two will obviously result in different requirements on the simulator. It also results in different requirements for the operation scheduler: for superscalar, the scheduler only needs to remove as many read-after-write(RAW) data hazards as possible; while for VLIW, the scheduler should ensure the correctness of the code by removing all hazards. For two architectures differing in interlocking policy only, it would be expected that the two result in different structural descriptions. However, for many ADLs, there is actually no difference for the two due to the lack of control path specification. Among mixed languages, only LISA and RADL can model the interlocking mechanism accurately since they have their emphasis on accurate simulator generation.

Generally speaking, dynamic pipeline behaviors are very hard to model cleanly. Behaviors like out-of-order issue and speculation are extremely hard and cumbersome to model precisely. The use of microinstructions[26] in CISC machines makes modeling even harder. Such complicated behavior can be ignored for compiler oriented ADLs. However, an ADL with support for both compiler and cycle accurate simulator may need to address them.

Another common ambiguity example is VLIW instruction word packing. Some architectures allow packing of only one issue bundle into an instruction word, i.e. only operations scheduled to issue at the same cycle can appear in the same instruction word; while other architectures allow multiple issue bundles in one instruction word. The architecture will dispatch the issue bundles at different cycles according to stop bits encoded in the instruction word[28], or according to instruction word templates[29]. Among the mixed ADLs, few can capture behavior related instruction bundling since its behavior is related to the control path. The problem of code compression[47,49] is more difficult than simple bundling and is not addressed by the ADLs at all.

Ambiguity is the result of abstraction. RT-level languages have the least amount of ambiguity while highly abstracted behavioral level ADLs have the most amount of ambiguity. Among mixed ADLs, simulator oriented ADLs are less ambiguous than compiler oriented ADLs. A good architecture description language design involves clever abstraction with minimal ambiguity.

In practice, ambiguity can be resolved by using an architecture template, i.e. the compiler or simulator presumes some architecture information left out from the ADL descriptions. This strategy has been adopted by the tool specific ADLs. The general purpose ADLs can resolve the ambiguity while preserving generality by using multiple architecture templates.

- Variable latency

  In many processors, operations or their operands may have variable latency. Many compiler oriented ADLs can only describe the nominal latency rather than accurate variable latency.

  Consider again the example of an integer Add in Figure 4. By default, the source operands will be read at the ID stage and destination written at the WB stage. So the source operand has a latency of 1, while the destination operand's latency is 4, relative to the fetch time. However, if there exists a forwarding path, which allows the operation to bypass its results from MEM to EX, then its destination operand latency is equivalent to 3. It is still fine if we fool the compiler by telling it the equivalent latency, though to the cycle accurate simulator we should tell the truth. Now consider a multi-issue version of the same

architecture in which inter-pipeline forwarding is forbidden, as is shown in Figure 5. Inside of each pipeline, forwarding can occur and we have an equivalent destination operand latency of 3. While between pipelines, no forwarding is allowed and the destination latency is 4. Here we see a variable latency for the same operand. The variable latency is hard to describe explicitly in the triple list, unless the forwarding path and its implication become part of the ADL. Reservation table based ADLs normally do not capture forwarding paths. A common practice is to inform the scheduler about only the worst-case latency, which means there is no distinction for the compiler between the intra and inter-pipeline case. As a result, some optimization opportunity is lost if the compiler has some control over operation issue.

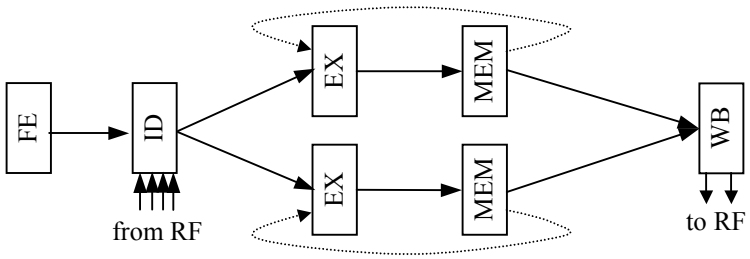


Figure 5 A two-issue integer pipeline with forwarding

Another type of variable latency resides in the operations themselves. Operations such as floating point division or square root can have a variable number of execution cycles depending on the exact source operands. In the triple list model, the variable operation latency means that the time element should be a variable, which could complicate the ADL greatly. To save the complication, usually a pessimistic latency is used in the description.

Memory operations can also result in variable latency under the presence of memory hierarchy. Most ADLs ignore the memory hierarchy by specifying a nominal latency for load and store operations since the emphasis of the ADLs is only on the processing elements. Memory hierarchy modeling is nontrivial due to the existence of various modules including ROM, SRAM, DRAM, non-volatile memory as well as numerous memory hierarchy options and memory management policies. As the speed discrepancy between digital logic and memory keeps increasing, it would be more and more important for the compiler to be aware of the memory hierarchy. So far, only EXPRESSION and TDL provide a parameterized memory hierarchy model. Research has indicated that memory aware optimization in some cases yields an average profit of 24% reduction of execution time [30].

Overall, it is useful to specify structural information such as the pipeline diagram, forwarding path and memory hierarchy for use in the compiler.

- Irregular constraints

  Constraints exist in computer architectures due to the limitation of resources, including instruction word length resources and structural resources. Common constraints include the range of constant operands and the number of issue slots. These constraints are familiar to compiler writers and can be handled with standard code generation and operation scheduling techniques.

  ASIP designs often have extra-irregular constraints, as they are extra-cost sensitive. Clustered and special register files with incomplete data transfer paths are commonly used to

conserve chip area and power. Irregular instruction word encoding is also used as a means to save instruction word size, and therefore instruction memory size.

Typical operation constraints include both intra-operation constraints and inter-operation constraints. An intra-operation constraint example from a real proprietary DSP design is that if in operation ADD D1, S0, S1, if D1 is AX, then S0 and S1 must be chosen from B0 and B1. If D1 is otherwise, there is no constraint on S0 and S1. Inter-operation constraints occur similarly relating operands in different operations. These operand-related constraints are hard to convert to artificial resource based constraints. Special compiler techniques are needed to handle them, such as an extra constraint checking phase in Aviv[19] or integer linear programming based code optimization techniques[33].

The irregular constraints create new challenges for compilers as well as ADLs. Most existing ADLs cannot model irregular constraints. ISDL and TDL utilize Boolean expressions for constraint modeling, which is an effective model to supplement the triple list.

- Operation semantics

  Both the code generator and the instruction set simulator need operation semantics information. For use in the code generator, simple tree-like semantics would be most desirable due to the popularity of tree-pattern based dynamic programming code generators. Operation side effects such as the setting of machine flags are usually not part of the tree pattern and may be omitted. Similarly, operations unsuitable for code generation can be omitted too. In contrast, for the simulator, precise operation semantics of all operations should be provided. It is not easy to unify the two semantic specifications into one, while still keeping it convenient for use in the compiler and simulator. ADLs like LISA and EXPRESSION separate the two semantic specifications at the cost of redundancy. This leads to the additional problem of ensuring consistency between the redundant parts of the description.

  For use in compilers, two schemes of operation semantics specification exist in ADLs. The first scheme is a simple mapping from machine independent generic operations to machine dependent operations. Both EXPRESSION and PRMDL use this scheme. The mapping scheme makes code generation a simple table lookup. This is a practical approach. The disadvantage is that the description cannot be reused for simulation purposes, and the description becomes dependent on compiler implementation.

  The second description scheme defines for each operation one or more statements based on a predefined set of primitive operators. nML, ISDL, Maril all use this approach. The statements can be used for simulation, and for most operations, the compiler can derive tree pattern semantics for its use. However, difficulties exist since some intermediate operations may fail to match any single operation.

  For example, in lcc's[31] intermediate representation, comparison and branch are unified in the same node. An operator EQ compares two of its children and jumps to the associated label if they are equal. Many processors do not have a single operation performing this task. Take the x86 family for example[32]. A comparison followed by a conditional branch operation is used to perform this task. The comparison operation sets machine flags and the branch operation reads the flags and alters control flow conditionally. The x86 floating point comparison and branch is even more complicated: the flags set by floating point comparison will have to be moved through an integer register to the machine flags that the branch operation can read. As a result, a total of four operations need to be used to map the single

EQ node. It would be a nontrivial job for the compiler to understand the semantics expression of each of the operations and the semantics associated with the flags in order to derive the matching. Some hints to the compiler should be provided under such cases. For such purpose, Maril[7] contains the "glue" and "*func" mechanisms. "Glue" allows user to transform the tree for easier pattern matching and "*func" maps one node to multiple operations. However, such mechanisms expose the internals of the software tools to the ADL, and makes the ADL tool dependent.

- Debugging of ADL descriptions

  Debugging support is an important feature for any programming language since code written by humans invariably contains bugs. Retargetable compilers and simulators are difficult to write and debug themselves. The bugs inside of an ADL description simply make the development process harder.

  An initial ADL description often involves thousands of lines of code and may end up with hundreds of errors. Among the errors, syntax errors can be easily detected by ADL parsers. Some others can be detected by careful case checking in the compiler, e.g., redefinition of the same operation. However, the trickiest errors can pass both tests and remain in the software for a long time. The experience of LISA developers shows that it takes longer to debug a machine description than to write the description itself[10].

  So far there is no formal methodology for ADL debugging. An ADL description itself does not execute on the host machine. It is usually interpreted or transformed into executable C code. In the interpretation case, debugging of the ADL description is actually the debugging of the retargetable software tool using the description. In the executable transformation case, debugging might be performed on the generated C, which probably looks familiar only to the tool writer.

  The task of debugging would be easier if there exists a usable target specific compiler and simulator for the architecture to be described. Comparison of emitted assembly code or simulation traces helps detect errors. Unfortunately this is impossible for descriptions of new architecture designs. The ADL writer probably will have to iterate in a trial and error process for weeks before getting a working description.

  Though it takes long to get the right description, it is still worthwhile since it would take even longer to customize a compiler. After the initial description is done, mutation can then be performed on it for the purpose of DSE.

- Others

  Other issues in ADL design includes handling of register overlap and mapping of intrinsics. These issues are not too hard and have been addressed by many ADL designs. They are worth the attention of all new ADL designers.

  Register overlap, or alias, is common to many architectures. For example, in x86 architecture, the lower 16 bits of the 32-bit register EAX can be used independently as AX. Lower and upper half of AX can also be used as 8-bit registers AH and AL. Register overlap also commonly exists in floating point register files of some architectures[45] where a single precision register is half of a double precision register.

  Most architectures contain a few operations that cannot be automatically generated by compilers or operations that cannot be expressed by predefined RT-lists operators. For

example, the TI's TMS320C6200 family implements bit manipulation operations such as reverse operation BITR. Such operations are useful to special application families. But no C operator can be mapped to such operations. Intrinsics mapping is a cleaner way to utilize these operations than inline assembly. So it is helpful for ADL to provide intrinsics support.

Description capability of operating system related operations may also be of interest to ADLs with accurate simulation support.

5. Future directions

Driven by market demands, increasingly complex applications are being implemented on semiconductor systems today. As both chip density and size increases, traditional board level functionality can now be performed on a single chip. As a result, processing elements are no longer privileged central components in electronic systems. A modern system-on-chip may contain multiple processors, busses, memories and peripheral circuits. As EDA tools start to address system-on-chip design issues, system level modeling becomes a hot research area.

A processing element model now becomes part of the system model. It is desirable that the same ADL modeling the processing element can be extended to model the whole system. Hardware description language based ADLs such as MIMOLA may be naturally used for this purpose. However, due to the large size of a typical system-on-chip, description of the entire system at the RT-level inevitably suffers from poor efficiency. When analog or mixed-signal parts are involved in the system, it is impossible to model the entire system even at the RTL level. Abstraction at different levels for different parts of the system is a more practical approach.

To reuse the existing tools and ADLs for system level description, an effective approach is to extend the ADL with communication interface models. Communication interface models include the modeling of input/output and bus drivers, memory system interface and interrupt interface.

The system level interface is important for processing element simulators to interact with system simulators. It is also important for the design of advanced compilers. For instance, a system level compiler can partition tasks and assign the tasks to multiple processing elements according to the communication latency and cost. Compilers can also schedule individual transactions to avoid conflict or congestion according to the communication model.

Among the existing mixed ADLs, LISA is capable of modeling interrupts and EXPRESSION has a parameterized memory hierarchy model. Both of these can be viewed as important steps toward specification of a system level communication interface.

On the whole, systematic modeling of the communication interface for processing elements is an interesting while difficult research direction. This can form the basis of device driver synthesis and even custom operating system synthesis.

IV. CONCLUSION

Architecture description languages provide machine models for retargetable software tools including the compiler and the simulator. They are crucial to DSE of ASIP designs.

In terms of requirements for an optimizing compiler, an tool independent ADL should contain several pieces of information:

- Behavioral information in the form of RT-lists. Hierarchical behavioral model based on attribute grammars are common and effective means. For VLIW architectures, instruction word formats should be also modeled.
- Structural information in the form of reservation table or coarse-grained netlists. Essential information provided by this section includes abstracted resources such as pipeline stages and data transfer paths.
- Mapping between the behavioral and structural information. Information in this part includes the time when semantic actions takes place and the resources used by the action.

Besides, irregular ILP constraint modeling is useful for ASIP oriented ADLs. The desire features of an ADL include simplicity for the ease of understanding, conciseness for description efficiency, generality for wide architecture range support, minimal ambiguity and redundancy, and tool independence. Tradeoff of abstraction levels needs to be made if a single ADL is used for both a re-targetable compiler and a cycle-accurate simulator for a range of architectures.

As ASIP design gains popularity in the system-on-chip era, ADLs as well as retargetable software tools sets will become an important part of EDA tools. These will encompass not just single processors, but rather complete systems.

V. REFERENCES

1. A.V. Aho, M.Ganapathi and S.W.K. Tjiang, "Code generation using tree matching and dynamic programming*," ACM Trans. On Programming Languages and Systems*, vol. 11, no. 4, Oct. 1989, pp. 491—516.
2. B. Kienhuis, *Design Space Exploration of Stream-based Dataflow Architectures*, Doctoral Thesis, Delft Univ. Technology, Jan. 1999
3. R. Leupers and P. Marwedel, "Retargetable Code Generation based on Structural Processor Descriptions," *Design Automation for Embedded Systems*, vol. 3, no. 1, Jan 1998, pp. 1-36.
4. H. Akaboshi, *A Study on Design Support for Computer Architecture Design*, Doctoral Thesis, Depart. of Information Systems, Kyushu Univ., Japan, Jan. 1996
5. A. Fauth, J. Van Praet and M. Freericks, "Describing instructions set processors using nML," *Proc. European Design and Test Conference*, Paris, France, Mar. 1995, pp. 503—507.
6. G. Hadjiyiannis, S.Hanono and S. Devadas. "ISDL: An instruction set description language for retargetability," *Proc. 34th Design Automation Conference* (DAC 97), Anaheim, CA., June 1997, pp. 299—302.
7. D.G. Bradlee, R.R. Henry and S.J. Eggers. "The Marion System for Retargetable Instruction scheduling," *Proc.ACM SIGPLAN'91 Conf. on Programming Language Design and Implementation*, Toronto, Canada, June 1991, pp. 229—240.
8. J.C.Gyllenhaal, B.R. Rau and W.W. Hwu, *Hmdes Version 2.0 Specification*, tech. report IMPACT-96-3, IMPACT Research Group, Univ. of Illinois, Urbana, IL, 1996.

9. S. Pees et al., “LISA — Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures,” *Proc. 36th Design Automation Conference (DAC 99)*, New Orleans, LA., June 1999, pp. 933—938.
10. S. Pees, A. Hoffmann and H. Meyr, “Retargetable compiled simulation of embedded processors using a machine description language*,” ACM Transactions on Design Automation of Electronic Systems*, Vol. 5, no. 4, Oct. 2000, pp. 815-834.
11. C. Siska, “A Processor Description Language Supporting Retargetable Multi-Pipeline DSP program Development Tools,” *Proc. 11th International Symposium on System Synthesis* (ISSS 98), Taiwan, China, Dec. 1998, pp.31—36.
12. A. Halambi et al., “Expression: A Language for Architecture Exploration through Compiler /Simulator Retargetability,” *Proc. Design Automation and Test in Europe* (DATE 99), Mar. 1999, Munich, Germany, 485—490.
13. A.S. Terechko, E.J.D. Pol and J.T.J. van Eijndhoven,” PRMDL: A Machine Description Language for Clustered VLIW Architectures,” *Proc. European Design and Test Conference* (DATE 01), Munich, Germany, Mar., 2001, pp. 821..
14. R. Leupers and P. Marwedel, “Retargetable Generation of Code Selectors from HDL Processor Models,” *Proc. European Design & Test Conference* (ED&TC 97), Paris, France, 1997, pp.144—140.
15. *IEEE Standard VHDL Language Reference Manual* (IEEE std. 1076-1993), IEEE, Piscataway, N.J., 1993.
16. A. Fauth and A. Knoll. *Automatic generation of DSP program development tools*. In Proc. Int’l Conf. Acoustics, Speech and Signal Processing (ICASSP 93), Minneapolis, MN., Apr.1993, vol.1, pp. 457—460.
17. F. Lohr, A. Fauth, M. Freericks, *SIGH/SIM: An environment for retargetable instruction set simulation*. tech report 1993/43, Dept. Computer Science, Tech. Univ. Berlin, Germany, 1993.
18. D. Lanneer et al., “Chess: Retargetable code generation for embedded DSP processors,” *Code Generation for Embedded Processors*, Kluwer Academic Publishers, Boston, MA, 1995, pp. 85—102.
19. S.Z. Hanono, *Aviv: A Retargetable Code Generator for Embedded Processors*, Ph.D Thesis, Massachusetts Inst. of Tech., June1999.
20. G.Hadjiyiannis, P. Russo and S. Devadas, “A Methodology for Accurate Performance Evaluation in Architecture Exploration,” *Proc. 36th Design Automation Conference* (DAC 99), New Orleans, LA., June1999, pp. 927-932.
21. S.C. Johnson, “Yacc: Yet Another Compiler-Compiler,” http://dinosaur.compilertools.net/yacc/index.html (current Nov. 2001)
22. M. Bailey and J. Davidson. “A Formal Model and Specification Language for Procedure Calling Conventions,” *Conf. Record of 22nd SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (POPL 95), New York, NY., Jan. 1995, pp. 298—310.
23. S. Aditya, V. Kathail and B. Rau. *Elcor’s Machine Description System: Version 3.0*. tech. report, HPL-98-128, Hewlett-packard Company, Sept, 1999.
24. A. Halambi et al., “A Customizable Compiler Framework for Embedded Systems,” *5th Int’l Workshop Software and Compilers for Embedded Systems* (SCOPES 2001), St. Goar, Germany, 2001.

25. A. Khare et al., "*V_SAT: A visual specification and analysis tool for system-on-chip exploration*," Proc. EUROMICRO-99,
Workshop on Digital System Design (DSD99), Milan, Italy, 1999.
26. J. Hennessy, D. Patterson, *Computer Architecture A Quantitative Approach*, 2nd edition, Morgan Kaufmann Publishers, San Francisco, CA., 1995.
27. S. Rajagopalan, M. Vachharajani and S.Malik, "Handling Irregular ILP Within Conventional VLIW Schedulers Using Artificial Resource Constraints," Proc. Int'l Conf. Compilers, Architectures, Synthesis for Embedded Systems (CASES 2000), San Jose, CA., Nov. 2000, pp 157—164.
28. *TMS320C6000 CPU and Instruction Set Reference Guide*, Texas Instrument Incorporated, Oct. 2000.
29. *Intel IA-64 Software Developer's Manual, Volume3: Instruction Set Reference*, Intel Corporation, July 2000.
30. P. Grun, N. Dutt and A. Nicolau, "Memory Aware Compilation Through Accurate Timing Extraction," *Proc. 37th Design Automation Conference*, Los Angeles, CA., June 2000, pp. 316-321.
31. C. Fraser, D. Hanson, *A Retargetable C Compiler: Design and Implementation*, Addison-Wesley Publishing Company, Menlo Park, CA., 1995.
32. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual*, Intel Corporation, 1997.
33. D. Kästner. *Retargetable Postpass Optimization by Integer Linear Programming.* Ph.D Thesis, Saarland University, Germany, 2000.
34. N. Ramsey and J. Davidson, "*Machine descriptions to build tools for embedded systems,*" ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98), vol. 1474 of LNCS, Springer Verlag, June 1998, pp. 172--188.
35. N. Ramsey, M. Fernandez, "Specifiying Representation of Machine Instructions," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 3, May 1997, pp. 492-524
36. N. Ramsey, J. Davidson, *Specifying Instructions' Semantics Using λ-RT*", interim report, University of Virginia, July, 1999
37. R. Stallman, *Using and Porting the GNU Compiler Collection(GCC)*, http://gcc.gnu.org/onlinedocs/gcc_toc.html(current Dec. 2001)
38. http://www.retarget.com (current Dec. 2001)
39. J. Paakki. "Attribute Grammar Paradigms--A High-Level Methodology in Language Implementation," *ACM Computing Surveys*, vol. 27, no. 2, June 1995, pp.196-256.
40. R. Milner, M. Tofte and R. Harper, *The Definition of Standard ML*, MIT Press, Cambridge, MA, 1989
41. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, San Francisco, CA, 1997
42. M. Fernandez, "Simple and effective link-time optimization of Modula-3 programs," *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.* June, 1995, pp. 103-115.
43. N.Ramsey, *A retargetable debugger*. Ph.D. thesis, Dept. of Computer Science, Princeton Univ., Princeton, NJ, 1992
44. *MC88100 RISC Microprocessor User's Manual*, Motorola Inc., 1988.
45. G.Kane, *MIPS R2000 RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ, 1987
46. *i860 64-bit Microprocessor Programmer's Reference Manual*, Intel Corporation, 1989

47. S.Aditya, S.A.Mahlke, B.Rau, Code Size minimization and Retargetable Assembly for Custom EPIC and VLIW Instruction Formats, *ACM Transactions on Design Automation of Electronic Systems*, vol. 5, no. 4, Oct 2000, pp. 752-773
48. E.Rohou, F.Bodin and A.Seznec, "SALTO: System for Assembly-language Transformation and Optimization," *Proc. 6th Workshop on compilers for Parallel computers*, Aachen, Germany, Dec. 1996, pp. 261-272
49. A. Wolfe and A. Chanin, "Executing Compressed Programs on an Embedded RISC Architecture," *Proc. 25th Int'l Symp. on Microarchitecture*, Portland, OR, Dec. 1992, pp. 81-91
50. D. Kastner, *TDL: A Hardware and Assembly Description Languages*, Technical Report TDL 1.4, Saarland University, Germany, 2000.
51. *The SPARC Architecture Manual, Version 8*, SPARC International, 1992