LISA – Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures

Stefan Pees[1], Andreas Hoffmann[1], Vojin Zivojnovic[2], Heinrich Meyr[1]

[1]Integrated Signal Processing Systems
Aachen University of Technology
Aachen, Germany
pees,hoffmann,meyr@ert.rwth-aachen.de

[2]AXYS Design Automation, Inc.
135 Santa Louisa
Irvine, CA 92606, USA
vz@axysdesign.com

Abstract – This paper presents the machine description language LISA for the generation of bit- and cycle accurate models of DSP processors. Based on a behavioral operation description, the architectural details and pipeline operations of modern DSP processors can be covered. Beyond the behavioral model, LISA descriptions include other architecture-related information like the instruction set. The information provided by LISA models enables automatic generation of simulators and assemblers which are essential elements of DSP software development environments. In order to proof the applicability of our approach, a realized model of the Texas Instruments TMS320C6201 DSP is presented and derived LISA code examples are given.

1 Introduction

Designers of today's telecommunication products such as cellular phones, modems, and networking devices are facing a rapidly growing system complexity. Driven by the advances in semiconductor technology and the need for new applications, the amount of system functionality that is realized on a single chip is increasing enormously. Due to the complexity and time-to-market constraints, the designer's productivity has become an important factor for successful products. For this reason, off-the-shelf digital signal processors (DSPs), application-specific instruction-set processors (ASIPs), and other programmable components are increasingly employed into systems and a growing amount of system functions is implemented in software instead of hardware. The programmability helps to rise the designer's productivity and the flexibility of software allows late design changes, thus shortening design cycles.

Integrating complete systems consisting of hardware and software components on a single chip raises new challenges in the area of verification. Since hardware based models and prototypes cannot be used, the complete system must be verified by means of simulation. Co-simulation of hardware and software puts specific requirements on the simulation accuracy of the programmable side. Generally, it is essential for designers of these systems to use cycle-accurate processor models because modern DSPs and microcontrollers implement deep pipelines. The complex pipeline mechanisms have a significant impact on the performance and cannot be covered by models which just accumulate instruction latencies since they have to cope with the requirements of cycle-based or even event-driven simulation of hardware components.

All embedded processors like DSPs and ASIPs need a complete tool set consisting of code-generation and simulation tools. Especially the task of building a custom simulator for new architectures is extremely error-prone and tedious. It is a very lengthy process of matching the simulator to an abstract model of the processor architecture. These efforts can be reduced significantly by using a retargetable simulator which is generated from machine descriptions.

The task of retargeting different tools like the compiler, the assembler, and the simulator of a given processor requires different types of architecture related information. This information can be described in different models.

- The *memory model* lists the registers and memories of the system with their respective bit widths, ranges, and aliasing.
- The *resource model* describes the available hardware resources and the resource requirements of operations. Resources reproduce the properties of hardware structures which can be used exclusively by one operation at a time.
- In the *behavioral model*, the activities of hardware structures are abstracted to operations changing the state of the system. The abstraction level of this model can range widely between the hardware implementation level and the level of high-level language (HLL) statements.
- The *instruction set model* collects all instructions as combinations of hardware operations which are permitted by the CPU controller. Furthermore, it comprises

the instruction semantics and their formal representation, such as admissible operands, binary representation and the assembly syntax.

- In the *timing model*, the activation sequence of hardware operations and units is specified.

1.1 LISA Language Approach

LISA is a language designed for the formalized description of programmable architectures, their peripherals, and interfaces [1]. Its development was necessary since existing approaches are not able to produce cycle-accurate models of modern DSP architectures (for example the TI TMS320C6x [2]) and to cover their instruction-set. The language as presented in this paper, is able to support DSP and embedded SIMD and VLIW processor architectures with deep pipelines, like the TMS320C62xx of TI which was used as the test case.

LISA supports different description styles and models at various abstraction levels. Similar to other programming languages, the user has a high degree of freedom to describe his view of the architecture. Furthermore, hierarchical modeling style is supported to allow structuring and easy maintenance of the code. Due to its C-like syntax, LISA can be easily and intuitively used by designers and exchanged as a non-ambiguous specification of the target architecture. Such a specification is a very valuable replacement for the textual documentation written by designers which is usually faulty and not up-to-date. The approach of using the LISA language even enables the automatic generation of such text book documentation.

2 Previous Work

Hardware description languages (HDLs) like VHDL or Verilog are widely used to model and simulate processors, but mainly with the goal of developing hardware. Using these models for instruction-level processor design has a number of disadvantages. They cover hardware implementation details which are not needed for performance evaluation and software verification. Moreover, the description of detailed hardware structures has a significant impact on simulation speed [3]. Another problem is that the extraction of the instruction set is a highly complex task and instruction set information, like e.g. assembly syntax cannot be obtained from HDL descriptions.

Many publications on machine description languages are focused on retargetable compilation for embedded processors. The approaches of Maril [4] as part of the Marion environment and a system for VLIW compilation [5] are both using reservation tables for code generation. But reservation tables cannot be used to describe pipeline operations like flushes or to model pipeline hazards.

Several publications address retargetable compilation and simulation. The language nML was developed at TU Berlin [6] and adopted in several projects, [7, 8]. While retargetable assemblers and disassemblers can be generated for some DSP processors, it is not possible to produce cycle-accurate simulators for pipelined processor architectures. The main reason is the simple underlying instruction sequencer which does not support pipeline operations like e.g. flushes. Processors with more complex execution schemes like the Texas Instruments TMS320C6x [2] cannot be described, even at the instruction-set level, because of the numerous combinations of parallel and sequential instructions within a fetch packet. Another significant restriction of nML is that the same information (action attribute) is used for the instruction selection task of the compiler and the description of the operation behavior for the simulator. Consequently, instruction semantics and instruction behavior are merged and described at the same abstraction level. These restrictions also apply to the approach of ISDL [9] which is similar to nML and to the machine description MDES which parameterizes the experimental PlayDoh architecture [10]. A similar approach is the language EXPRESSION [11] which incorporates particular mechanisms for the description of memory hierarchies. However, no results are published that indicate the applicability for cycle-accurate simulation purposes.

The language RADL [12] is derived from earlier work on LISA [1] and extended to support multiple pipelines. But again, no results are provided on realized simulators based on this language.

To summarize the review, none of the approaches above does support cycle-accurate simulation or fast processor simulators that are based on compiled techniques [13]. Our interest in supporting this technique and the issue of realizing cycle-accurate processor models motivated the introduction of the language LISA.

3 LISA Language

In many aspects, LISA incorporates ideas which are similar to nML. However, it turned out from our experience with different DSP architectures that significant limitations of existing machine description languages must be overcome to allow the description of modern commercial DSPs. For this reason, LISA includes improvements in the following areas:

- Support of cycle-accurate processor models, including constructs to specify pipelines and their mechanisms,
- The target class of processors includes SIMD, VLIW, and superscalar architectures of real products currently on the market,
- Direct support for compiled simulation techniques,
- Strong orientation on the programming language C,
- Distinction between behavior and semantics to freely determine the abstraction level of the processor model,
- Support for instruction aliasing and complex instruction coding schemes.

LISA descriptions are composed of *resource declarations* on the one hand and of *operations* on the other hand. The declared resources represent the storage objects of the hardware architecture (e.g. registers, memories, pipelines) which capture the state of the system and which can be used to model the limited availability of resources for operation access. Section 3.1 provides a discussion of resource declarations.

Operations are the basic objects in LISA. They represent the designer's view of the behavior, the structure, and the instruction set of the programmable architecture. Operation definitions collect the description of different properties of the system, i.e. operation behavior, instruction set information, and timing. These operation attributes are defined in several sections.

- The CODING section describes the binary image of the instruction word which is part of the instruction set model.
- The SYNTAX section describes the assembly syntax of instructions and their operands which is part of the instruction set model.
- The SEMANTICS section specifies the instruction semantics.
- The BEHAVIOR and EXPRESSION sections describe components of the behavioral model. During simulation, the operation behavior is executed and modifies the values of resources which drives the system into a new state.
- The ACTIVATION section describes the timing of other operations relative to the current operation.
- The DECLARE section contains local declarations of identifiers and lists of alternative elements (GROUP).

Beyond these sections, which are predefined in LISA, the designer may add further sections in order to describe other attributes, like e.g. power consumption.

The designer has the full freedom to determine the abstraction level of his model based on the operations. For example, one instruction of a processor may be represented by just one operation in case of an instruction set model. In another case, it may be described by a whole set of operations which represent the separate actions between clock cycles in case of a cycle-accurate model. A presentation of the operation sections is given in section 3.2.

3.1 Resources

The resource section lists the definitions of all objects which are required to build the memory model and the resource model. Object definitions in the resource section contribute to both models, automatically receiving the properties of a memory element and a resource element. It depends on the operation's functionality if both or only one of these properties is used. A register and a pipeline stage for example may both have the properties of a resource but only the register is used in statements of the operation behavior. A detailed discussion of the role of resources in our generic machine model can be found in [1].

```
RESOURCE {
  PROGRAM_COUNTER  int pc;
  CONTROL_REGISTER int instruction_register;
  REGISTER         bit[48] accu;
  REGISTER         bit carry;
  DATA_MEMORY      int data_mem1[0x80000];
  DATA_MEMORY      int data_mem2[4]([0x20000]);
  PROGRAM_MEMORY   int prog_mem[0x100..0xFFFF];
}
```

Example 1: Declaration of resources.

The resource section begins with the keyword RESOURCE followed by braces enclosing all object definitions. The definitions are made in C-style and can be attributed with one of the keywords `REGISTER`, `CONTROL_REGISTER`, `PROGRAM_COUNTER`, `DATA_MEMORY`, or `PROGRAM_MEMORY`. These keywords are not mandatory but they are used to classify the definitions. Example 1 shows the declaration of several resources.

The LISA language provides designated mechanisms to model pipelines of a processor architecture. The principle of this pipeline model is that operations are assigned to pipeline stages. But before this mechanisms can be used, the respective pipelines must be defined in the RESOURCE section. The declaration starts with the keyword PIPELINE, followed by an identifier as the name of the pipeline and the list of stages as shown in example 2.

```
RESOURCE {
  PIPELINE fetch_pipe = { PG; PS; PW; PR; DP };
  PIPELINE execute_pipe = { DC; E1; E2; E3; E4; E5 };
}
```

Example 2: Pipeline definition.

The stage identifiers are enclosed in braces and separated by semicolons. They are ordered with the first stage first. Operations are assigned to a certain pipeline stage by using the name of the pipeline followed by a dot and the identifier of the respective stage, such as `pipe.DC` for example.

3.2 Operations

Operations are formed by a header line and the operation body. The header line consists of the keyword OPERATION, the identifying name of the operation, and possible options:

```
OPERATION name_of_operation [options]
{
  sections...
}
```

Enclosed in (curly) braces, the operation body contains the different sections which describe the properties of the instruction set model, the behavioral model, the timing model, and required declarations.

3.2.1 Instruction Set Model

In the CODING section the elements are specified as a sequence of coding objects. The coding objects can be either a sequence of binary code or a reference to the coding section of another operation. Binary code is specified as a sequence composed of `0`, `1`, and `x` which is preceded by a `0b`. So, the binary sequence for the decimal value 28 would be written as `0b11100`.

During decoding, the bit pattern must match the provided instruction word to select a specific operation or resource. During encoding, the same pattern is used to generate the respective instruction word. The `x` matches always (don't care bit).

Object references are the identifiers of other operations which include a coding section as well. Thus, the coding information of the referenced operation is inserted at the respective position. A sample coding section could look like this:

```
CODING { 0b001011101 x_operand y_operand result }
```

In order to identify instructions, the coding sequences of all defined operations must be compared to the actual value of the current instruction word. This is specified in the coding section by comparing one or more resource values to the coding sequence. The compare operator == separates the identifier of the resource which shall be compared (on the left hand side) from the coding sequence (on the right hand side). This comparison represents the top object in the coding tree.

```
OPERATION {
  DECLARE {
    GROUP Instruction = { abs || add || and || cmp || ld
                          || mul || mv || norm || not ||
                          or || sat || sub || st || xor };
  }
  CODING { instruction_register == Instruction }
  SYNTAX { Instruction }
  BEHAVIOR { Instruction }
}
```

Example 3: Root of the coding tree.

The purpose of the SYNTAX section is to describe the syntax of assembly instructions. It is specified as a textual description of the instruction mnemonics, the operands, and the numeric parameters evaluating to a string of characters. Instruction mnemonics are specified as strings enclosed in quotation marks. Operands may be specified by referencing other operations or as immediate values.

During assembly, the string pattern must match the provided assembly statement to select a specific operation or resource. During disassembly, the same pattern is used to generate the respective assembly statement.

Object references are the identifiers of other operations which include a syntax section as well. Thus, the syntax information of the referenced operation is inserted at the respective position. A sample syntax declaration could look like this:

```
SYNTAX { "ADD" x_operand "," y_operand:#S
         "," result ";" }
```

3.2.2 Declare Section

Similar to programming languages, LISA requires symbol declarations for all objects used in operations. The DECLARE section collects four types of symbols which are introduced here:

- declaration of operation references,
- definition of operation groups,
- declaration of group references,
- and declaration of inter-section references (labels).

The purpose of groups is to list alternative operations which are used in the same context. They correspond to the mechanism of or-rules in nML. The group name replaces the reference to a specific operation which is part of this group. A typical application for the use of groups are admissible source and destination operands of instructions.

Group definitions are located in the DECLARE section and identified by their name which is followed by the group members. Each member of the group must be an operation identifier. Group members are selected based on the coding or syntax information provided in the respective operations which has to match the current instruction coding or the current assembly statement (see example 4).

The groups `src1`, `src2`, and `dest` are instantiations of the same operation group consisting of only one element – the operation `register`. They are the admissible source and destination operands for operation `add_d` and used in the coding and in the syntax section. All operations listed in the group declaration must provide coding as well as syntax information in order to enable operation selection.

```
OPERATION add_d
{
  DECLARE  { GROUP  Dest, Src1, Src2 = { register }; }
  CODING   { Dest  Src2  Src1  0b010000  0b10000 }
  SYNTAX   { "ADD"  ".D"  Src1  ","  Src2  ","  Dest }
  BEHAVIOR { Dest = Src1 + Src2; }
}

OPERATION register
{
  DECLARE    { LABEL  index; }
  CODING     { 0bx  index:0bx[4] }
  SYNTAX     { "A"  index:#U }
  EXPRESSION { A[index] }
}
```

Example 4: Operation groups.

The declaration of the label `index` in operation `register` is an inter-section reference. It is used to link elements of different sections. The last four bits of the coding are linked to the numerical parameter in the syntax. This combination forms a translation rule to be used by the assembler or the disassembler. For example, the assembler statement

```
ADD .D A4, A3, A15;
```

would be translated into the binary code

```
01111 00011 00100 010000 10000.
```

3.2.3 Behavioral Model

The BEHAVIOR section describes the behavior of operations based on the programming language C. The whole section can be seen as the implementation body of a function. As in any basic block in C, local variables can be declared here. Within the behavioral code, groups and direct references to other operations are permitted. The referenced operations either provide further behavior code or expressions which allow to access resources.

The EXPRESSION section identifies an object which is accessed by the behavior part of a referencing operation. These expressions are typically used for operands and other resource accesses. In effect, they represent the mechanism of modes in nML. Example 4 depicts the operation `add_d` which accesses the expressions identified in operation `register`. According to this example, the assembly statement

```
ADD .D A3, A4, A0
```

would cause the following behavioral code to be executed during simulation:

```
A[0] = A[3] + A[4];
```

3.2.4 Timing Model

In cycle-accurate machine models, all effects of pipelines in the processor architecture become visible and have to be described. LISA assumes all operations to be executed synchronously to control steps. The designer has the freedom to determine if these control steps correspond to instruction cycles, clock cycles or even phases. Based on these control steps, LISA incorporates a generic pipeline model with two major mechanisms:

- operation assignment to the stage of a pipeline and
- activation of operations with or without delay.

In order to deal with the complex pipelines of modern DSP processors, LISA allows the description of multiple pipelines and supports typical pipeline operations, such as stalls and flushes. Beyond these mechanisms, the generic pipeline is open for user-defined operations which supplement the pipeline control mechanisms.

Operations can be assigned to those pipelines defined in the RESOURCE section (see Example 2). The assignment is made in the header line of the respective operation by appending the keyword IN and the identifier of the pipeline stage such as

```
OPERATION add IN execute_pipe.E1
```

The purpose of the ACTIVATION section is to describe the activation timing of operations in the respective stage of a pipeline. It supplements the behavior section which can only call other operations in the same control step. In the activation section, operations can be activated in the same or in subsequent control steps. This lets the designer e.g. concatenate operations which belong to the same instruction as shown in Example 5.

```
OPERATION Prog_Address_Generate IN fetch_pipe.PG {..}
OPERATION Prog_Address_Send IN fetch_pipe.PS {..}
OPERATION Prog_Access_Ready_Wait IN fetch_pipe.PW {..}
OPERATION Prog_Fetch_Packet_Receive IN fetch_pipe.PR {..}

OPERATION main {
  ACTIVATION {
    if (dispatch_complete && !multicycle_nop) {
      Prog_Address_Generate, Prog_Address_Send,
      Prog_Access_Ready_Wait,
      Prog_Fetch_Packet_Receive, Dispatch
    }
    if (multicycle_nop) {
      fetch_pipe.DP.stall ();
      execute_pipe.DC.stall ();
    }
    fetch_pipe.shift();
    execute_pipe.shift();
  }
}
```

Example 5: Activation of operations.

Here, the operation `main` activates operations (e.g. `Program_Address_Generate`) under a certain condition. The activation time is directly determined by the number of stages (spatial distance) in the pipeline. Activation of operations which are assigned to the same pipeline stage are activated in the same control step.

Besides this delay caused by the spatial distance, it is also possible to add delays which are measured in control steps. In general, the activation section consists of a list of operations which are separated by the activation operators. There are two types of operators:

- concurrent activation operator: comma (`,`) and
- delayed activation operator: semicolon (`;`).

The same operators can be found in Maril for the notation of operations in a pipeline. However, we allow the activation to be embedded in control structures such as if-then-else and switch-case.

3.2.5 Conditional Operation Structuring

One of the most crucial issues in the development of processor simulators is simulation speed [3]. It turned out from our research studies that the technique of compiled simulation can achieve speed-ups of more than two orders of magnitude over interpretive processor simulators [13]. In order to support the generation of compiled simulators, LISA features conditional structures on the operation-level that evaluate at compile-time:

- IF-THEN-ELSE statements and
- SWITCH-CASE statements.

These conditional structures allow to select different blocks of LISA code. They enclose one or more operation sections. The selection is made based on the selection of group members.

```
OPERATION register {
  DECLARE { GROUP  Side = { side1  ||  side2 };
            LABEL  index; }
  CODING { 0bx  index:0bx[4] }
  SWITCH (Side) {
    CASE side1: { SYNTAX     { "A"  index:#U }
                  EXPRESSION { A[index] } }
    CASE side2: { SYNTAX     { "B"  index:#U }
                  EXPRESSION { B[index] } } }
}

OPERATION side1 {
  CODING { 0b }
  SYNTAX { "1" }
}

OPERATION side2 {
  CODING { 1b }
  SYNTAX { "2" }
}
```

Example 6: Conditional operation structuring.

Example 6 depicts the operation `register`, which includes a SWITCH-CASE statement to select between two register sets. The selection of the respective syntax and expression can already be determined at compile-time thus avoiding to check the bit at run-time of the simulation. More details on the LISA language can be found at [14].

4 TI TMS320C6201 Simulator

In order to check the applicability and the use of the LISA language, we realized models of different DSPs. Among these, the most complex architecture is TI's TMS320C6201. In cooperation with Texas Instruments, a cycle-based LISA model of this processor was realized by one designer within less than two month. The description of the processor core including memory interface and interrupt controller consists of

- 54 resources and
- 256 operations comprising the full set of 156 real instructions and 8 instruction aliases

which adds up to 5362 lines of LISA code at an average of approximately 21 lines of code per operation. It turned out from our modeling experience that most issues and the hardest problems in the description of processor architectures only appear if processor models are realized completely, including all details *and* verified successfully.

4.1 Retargetable Development Tools

We have developed an environment of retargetable development tools that is configured by LISA descriptions. A parser reads the LISA models and translates them into an intermediate data base which is accessed by all other tools. We implemented a retargetable compiled simulator that is generated from this data base. The translation of the TMS320C6201 processor model into the simulator takes only 30 seconds on a Sparc Ultra 10 workstation.

The realized simulator was successfully verified against the simulator `sim62x` from Texas Instruments based on a number of typical DSP applications.

5 Conclusion and Future Work

LISA is a language which aims at the formal description of programmable architectures, their peripherals, and interfaces. The language supports different description styles and models at various abstraction levels. Its development was necessary since existing approaches are not able to produce cycle-accurate models of pipelined DSP architectures and to cover their instruction-set. Furthermore, LISA enables the principle of fast compiled simulation of embedded processors. This paper provides an overview of the LISA language and discusses modeling issues with sample LISA code derived from a realized model of the TI TMS320C6201.

Our future work will focus on modeling further real-life processor architectures and the generation of fast simulators. We will systematically investigate retargetable fast simulators based on compiled techniques and report on the simulation speed of this technology. Another issue is the integration of software simulators into HW/SW co-simulation environments. Furthermore, the goal of the ongoing language design is to address retargetable compiler back-ends as well.

6 Acknowledgement

The authors would like to thank Wanda Gass, Dan Pickens, David Shoemaker, and Dave Thomas for their technical and organizational support for this project.

References

[1] V. Živojnović, S. Pees, and H. Meyr, "LISA — machine description language and generic machine model for HW/SW co-design," in *Proceedings of the IEEE Workshop on VLSI Signal Processing*, (San Francisco), Oct. 1996.

[2] Texas Instruments, *TMS320C62x/C67x CPU and Instruction Set Reference Guide*, Mar. 1998.

[3] J. Rowson, "Hardware/Software co-simulation," in *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 1994.

[4] D. Bradlee, R. Henry, and S. Eggers, "The Marion system for retargetable instruction scheduling," in *Proc. ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, Toronto, Canada*, pp. 229–240, 1991.

[5] B. Rau, "VLIW compilation driven by a machine description database," in *Proc. 2nd Code Generation Workshop, Leuven, Belgium*, 1996.

[6] A. Fauth, J. Van Praet, and M. Freericks, "Describing instruction set processors using nML," in *Proc. European Design and Test Conf., Paris*, Mar. 1995.

[7] M. Hartoog, J. Rowson, *et al.*, "Generation of software tools from processor descriptions for hardware/software codesign," in *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, Jun. 1997.

[8] W. Geurts, D. Lanneer, et *al.*, "Design of DSP systems with Chess/Checkers," in *2nd Int. Workshop on Code Generation for Embedded Processors*, (Leuven), Mar. 1996.

[9] G. Hadjiyiannis, S. Hanono, and S. Devadas, "ISDL: An instruction set description language for retargetability," in *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, Jun. 1997.

[10] V. Kathail, M. Schlansker, and B. Rau, "HPL PlayDoh Architecture Specification: Version 1.0," in *HP Laboratories Technical Report HPL-93-80*, Mar. 1994.

[11] A. Halambi, P. Grun, et *al.*, "EXPRESSION: A language for architecture exploration through compiler/simulator retargetability," in *Proceedings of the European Conference on Design, Automation and Test (DATE)*, Mar. 1999.

[12] C. Siska, "A processor description language supporting retargetable multi-pipeline dsp program development tools," in *Proceedings of the International Symposium on System Synthesis (ISSS)*, Dec. 1998.

[13] S. Pees, V. Živojnović, A. Ropers, and H. Meyr, "Fast Simulation of the TI TMS 320C54x DSP," in *Proc. Int. Conf. on Signal Processing Application and Technology (ICSPAT)*, (San Diego), pp. 995–999, Sep. 1997.

[14] *http://www.ert.rwth-aachen.de/lisa/lisa.html.*