ISSN:2231-0711

Modelling with LISA- A Powerful Machine Description Language

P.V.Bhandarkar[1] , Dr.S.S.Limaye[2],

[1]Assistant Professor ,SRCOEM,RTM Nagpur University
Nagpur, M.S ,India

[2]Principal,JIT,RTM Nagpur University
Nagpur, M.S, India

Abstract- ***The development of application specific instruction set processors comprises several design phases: architecture exploration, software tools design, system verification and design implementation. The LISA processor design platform (LPDP) based on machine descriptions in the LISA language provides one common environment for these design phases. Required software tools for architecture exploration and application development can be generated from one sole specification. This paper focuses on the implementation phase and the generation of synthesizable HDL code from a LISA model.***

Keywords-**LISA, Architecture exploration, machine description languages.**

I. INTRODUCTION

Today the majority of microprocessors are employed in embedded systems (1). This number is not surprising because a typical home today may have a laptop/PCs with a high performance microprocessor but probably dozens of embedded systems including electronic entertainment, household, and telecom devices, each of them equipped with one or more embedded processors. A modern car typically has more than 50 microprocessors. Embedded processors are most often developed by relatively small teams within short time-to-market requirements and the processor design automation is clearly a very important issue. Once a model of a new processor is available, existing hardware synthesis tools enable the path to custom VLSI implementation. However embedded processor designs typically begin at a much higher abstraction level, even far beyond an instruction set architecture (ISA) and involves several architecture exploration cycles before the optimum hardware/software partitioning is found. It turns out, that this requires a number of tools for software development and profiling. These are normally written manually - a major source of cost and inefficiency in embedded processor design so far. The CoWare Processor Designer formerly known as LISAtek processor design platform (LPDP) originally developed at RWTH Aachen (4, 5) and now a product of CoWare Inc. addresses these issues in a highly innovative and satisfactory manner. The LISA language supports profiling-based stepwise refinement of processor models down to cycle-accurate and even VHDL or Verilog RTL synthesis models for fast custom VLSI implementation. In an elegant way, it avoids model inconsistencies otherwise inevitable in traditional design flows.

II. SIMILAR WORK

A need to explore architectural tradeoffs during the design phase of embedded processors [3] has led to an increased interest in ADLs for processor design.[8] Existing processor design ADLs can be categorized into one of three categories. These categories include languages that focus on describing the processor at the architectural level (RTL or structural level), languages that abstract the design to the instruction level (behavioral level), and the languages that incorporate a joint behavioral and structural design approach.[2] [8] MIMOLA (Machine Independent Microprogramming Language) is an example of an ADL that describes the processor at the RTL level.[2] [8] The MIMOLA language is very similar to that of Verilog or VHDL. The software tool suite uses the structural definition of the processor and therefore, usually results in poor quality compilers and assemblers.[8] Languages such as MIMOLA do not support the exploration of different processor architectures and for this reason were not considered for this research.[2] [9] nML and ISDL (Instruction Set Description Language) are examples of ADLs that describe the design of an embedded processor at the behavioral level.[2] [8] nML will produce an assembler based on the defined model, however the generated simulator does not support cycle accurate pipeline or VLIW architectures.[6] The nML processor model can be used with the separate CHESS compiler to generate processor specific code from a higher level language source file.[3] Furthermore, nML must be used with a separate product, GO HDL generator, to produce synthesizable RTL code [2] [10] ISDL is similar to nML and requires a separate compiler tool to generate processor specific assembly files.[7] Since these languages model the processor

from a instruction set view, the ability to group common functionality together into dedicated hardware units is severely limited; which in return limits the amount of resource sharing that can occur.[2] For these reasons, behavioral level ADLs were not further considered. Many newer ADLs are available that describe the embedded processor in a combined behavioral and structural manner. Examples of such ADL tools include EXPRESSION, Xtensa by Tensilica, PICO (Processor In Chip Out) by HP Labs and CoWare Inc.'s LISA.[2] Xtensa is built on a predefined RISC core and is limited in the architectures that it can model.[11] Similarly, PICO is based on a set of predefined hardware components and is also limited in its ability to model arbitrary processor architectures.[12]

EXPRESSION and LISA are more flexible ADLs and allow arbitrary processor architectures and their memories to be designed and simulated. Both languages provide automatic generation of a complete tool suite and RTL code generation from the model description.[12] [10] [8] .

EXPRESSION is a public-domain product that is available http://www.ics.uci.edu/~expression/index.htm. No results have been published on the efficiency of the generated tool suite or RTL code based on an EXPRESSION model. CoWare's LISA tool suite is the leading commercially supported ADL. CoWare is the only such toolset that has both UNIX and Windows versions, unlike EXPRESSION which requires a Sun Sparcstation.[2] [8].

III. LISA LANGUAGE

The language LISA [13] is aiming at the formalized description of programmable architectures, their peripherals and interfaces. It was developed to close the gap between purely structural oriented languages (VHDL, Verilog) and instruction set languages for architecture exploration purposes.

The language syntax provides a high flexibility to describe the instruction set of various processors, such as SIMD, MIMD and VLIW-type architectures. Moreover, processors with complex pipelines can be easily modelled. The LISA machine description provides information consisting of the following model components:

1)The *memory model* lists the registers and memories of the system with their respective bit widths, ranges, and aliasing.

2)The *resource model* describes the available hardware resources, for example registers or functional units and the resource requirements of operations. Resources reproduce properties of hardware structures which can be accessed exclusively by a given number of operations at a time.

3)The *instruction set model* identifies valid combinations of hardware operations and admissible operands. It is expressed by the assembly syntax, instruction word coding, and the specification of legal operands and addressing modes for each instruction.

3) The *behavioral model* abstracts the activities of hardware structures to operations changing the state of the processor for simulation purposes. The abstraction level of this model can range widely between the hardware implementation level and the level of high-level language (HLL) statements.

4)The *timing model* specifies the activation sequence of hardware operations and units.

5)The *micro-architecture model* allows grouping of hardware operations to functional units and contains the exact micro-architecture implementation of structural components such as adders, multipliers, etc.

These various model components are sufficient for generating software tools as well as a HDL representation each with their particular requirements. Furthermore, LISA models may cover a wide range of abstraction levels. This comprises all levels starting at a pure functional sight, modeling the data path of the architecture, to register transfer level (RTL) accurate models. Besides a proper description of the structure, RTL models include detailed information about the micro-architecture model. Therefore, these models can be used to generate a HDL representation of the architecture, using the languages VHDL,Verilog or SystemC. Certainly a working set of software tools can be generated from all levels of abstraction.

LISA can be used to model any processor that is defined by an instruction set, such as an SRC or a DSP processor. The LISA tool suite can also be used to develop new application specific processors, study the effect of different computer architectures on an instruction set, as well as develop replacements for legacy processors. The LISA language allows for the easy representation of pipelined (cycle-accurate) and VLIW processors. The LISA tool suite includes Processor Designer, Processor Debugger, Processor Generator, and a C Compiler.

The full description of the hardware and instruction set of the processor can be developed with Processor Designer. This combining of hardware and software design for an embedded processor greatly reduces the complexity and time of modeling. LISA can be used to describe the instructional hierarchy, which lends itself to easy addition of instructions to an already defined design. The behavior of each instruction within LISA is coded in ANSI-C. This eliminates the need for previous knowledge of an HDL language in order to use Processor Designer. Processor Designer generates an assembler, linker, disassembler, an instruction set simulator, and a debugger, based on the LISA design description. Along with these tools,

Processor Designer can generate an instruction set user's manual. The instruction set user's manual lists the complete instruction set in the architecture along with syntax and a short description of how each instruction is implemented. The LISA language model can be tested and debugged using the Processor Debugger tool, before the HDL code is generated. Within the debugger the user has the ability to view all the hardware resources, including registers and memories, as the assembly code is executed. The debugger also keeps track of statistical information about the processor, such as the percentage of time a pipeline stage spent executing on data or how many times a block of assembly is run (keeps iteration counters on every line of the assembly code).
Once the LISA model is verified with Processor Designer, HDL code can be generated with the Processor Generator tool. The Processor Generator tool allows for generated HDL code in either Verilog or VHDL. Options are also given for the optimization of the generated HDL code; which include the degree of resource sharing between functional Units.

IV. ARCHITECTURE DESIGN

Today's standard architecture development process uses description languages in two fields for the development of new architectures: for architecture exploration, the software development tools are realized using a high level language as C/C++ to describe the target architecture from the instruction set view, whereas (low level) Hardware Description Languages (HDL) like VHDL and Verilog are used to model the underlying hardware in detail for implementation purposes. It is obvious that combining the development processes of software tools suite and HDL description is extremely benefiting.
As can be seen in figure 1 the LISA language compiler generates both and design changes only influence the LISA description. By this, consistency problems vanish and the generated software development tools and HDL code are correct by construction.

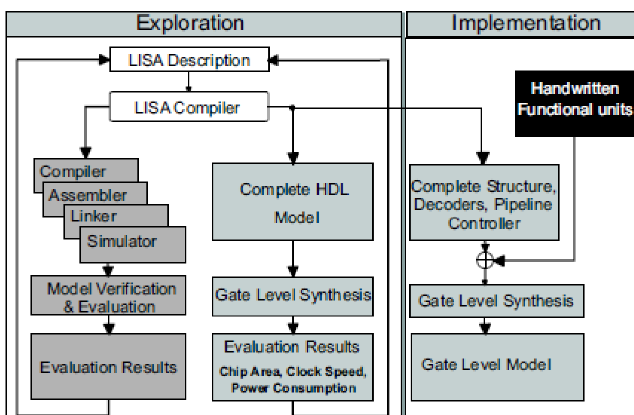

Figure 1. Exploration and implementation

The LISA processor design platform (LPDP)[14] is an environment that allows the automatic generation of software development tools for architecture exploration and application design, hardware-software co-simulation interfaces and hardware implementation, from one sole specification of the target architecture in the LISA language. The set of LISA tools comprises the following programs

1) The LISA language debugger for debugging the instruction-set as well as the behavior with a dedicated graphical debugger frontend.
2) The Assembler which translates text-based instructions into object code for the respective Programmable architecture.
3) The linker which is configured by a dedicated linker command file
4) The Instruction-set architecture (ISA) simulator for cycle accurate simulation including support for deep instruction and data pipelines.

After design exploration and application design the target architecture needs to be implemented, which will be discussed in subsequent part of this paper.

V. ARCHITECTURE IMPLEMENTATION

The LPDP platform supports the generation of a HDL representation of the architecture. Since, the generated HDL model does not consist of any predefined components, such as ALUs or basic control logic, the LISA compiler must derive all necessary information from the given LISA description. Thus, the generated HDL model components can be fully compared to the LISA model components as given in following section illustrated in figure 2:

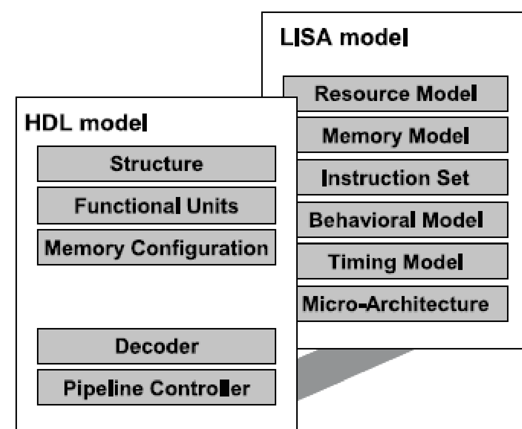

Figure 2: LISA model an correspondent HDL model components

- The memory configuration, which summarizes the register and memory sets including the bus configuration is directly derived from the LISA memory model.

- The structure of the architecture, such as pipeline stages and pipeline registers is generated. The required information is gathered from the resource model, behavioral model and the micro-architecture model.
- The functional units are generated from the microarchitecture model. Depending on the HDL language used, the functional units are either generated as empty frames or with full functionality.
- The decoders are resulting from the coding information included in the instruction set model and the timing model.
  The pipeline controller is also generated from instruction set model and the timing model.

Generally a processor model written in LISA has two sections those are Resource and Operation section [18]. Again the Operation section contains three subsections those are Coding, Syntax and Behavior Processor resources include the internal storage elements of the processor as well as dedicated input/output pins and global variables. The internal storage elements of the processor are represented by its registers and its internal memories. But in cycle accurate models there are other types of processor resources, like pipeline registers and internal signals [19]. Processor resources are generally declared in the resource section, indicated by the keyword RESOURCE. An example is shown in Example 1.

VI RESOURCES

The resource section lists the definitions of all objects[13] 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 proper-ties 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 [15].

```
RESOURCE {
  PROGRAM_COUNTER   int  pc₃
  CONTROL_REGISTER  int  instruction_register₃
  REGISTER          bit[48]  accu₃
  REGISTER          bit  carry₃
  DATA_MEMORY   int  data_mem1[0x80000]₃
  DATA_MEMORY   int  data_mem2[4]([0x20000])₃
  PROGRAM_MEMORY      int  prog_mem[0x100..0xFFFF]₃
}
```

Example 1: Declaration of resources

The resource section begins with the keyword RESOURCE followed by braces enclosing all object de finitions. 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 = { PG₃ PS₃ PW₃ PR₃ DP }₃ PIPELINE
  execute_pipe = { DC₃ E1₃ E2₃ E3₃ E4₃ E5 }₃
}
```

Example 2: Pipeline definition.

The stage identifiers are enclosed in braces and separated by semicolons. They are ordered with the rst stage rst. 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

VII OPERATIONS

Operations are formed by a header line and the operation body. The header line consists of the keyword OPERA-TION, 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.

VIII 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 f 0b001011101 x operand y operand result g
```

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 }₃

   }
   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 f "ADD" x operand "," y operand:#S ","
                 result "₃" g
```

IX 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,
- de nition 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.

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, A15₃

would be translated into the binary code

01111 00011 00100 010000 10000.

```
OPERATION add_d
{
  DECLAR    GROU
  E         { P       Dest, Src1, Src2 = { register }₃ }
  CODING    { Dest  Src2  Src1  0b010000  0b10000 }
  SYNTA
  X         { "ADD"  ".D"   Src1   ","   Src2   ","   Dest }
  BEHAVIOR  Dest
  {            =        Src1 + Src2₃ }
}

OPERATION register
{
  DECLARE { LABEL index₃ }
  CODING { 0bx index:0bx[4] }
  SYNTAX { "A" index:#U }
  EXPRESSION { A[index] }

}
```

Example 4: Operation groups

X 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 ac-cessed by the behavior part of a referencing operation. These expressions are typically used for operands and other resource accesses. 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

XI TIMING MODEL

In cycle-accurate machine models, all effects of pipelines in the processor architecture become visible and have to be de-scribed. 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 ()₃
        execute_pipe.DC.stall ()₃
     }
     fetch_pipe.shift()₃
     execute_pipe.shift()₃
   }
}
```

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 (₃).

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.

XII CONDITIONAL OPERATION STRUCTURING

One of the most crucial issues in the development of processor simulators is simulation speed [19]. 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 [17]. In order to sup-port 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.

XIII CONCLUSION AND FUTURE WORK

LISA is a language which aims at the formal description of programmable architectures, their peripherals, and inter-faces. 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 .

Our future work will focus on modeling further real-life pro-cessor architectures and the generation of fast simulators.

REFERENCE

[1] Uwe Meyer-B¨ase, Alonzo Vera, Suhasini Rao, Karl Lenk, and Marios Pattichis *FPGA Wavelet Processor Design using Language for Instruction-set Architectures (LISA).* Independent Component Analyses, Wavelets, Unsupervised Nano-Biomimetic

Sensors, and Neural Networks V, Proc. of SPIE Vol. 6576, 65760U, (2007).

[2] A. Hoffmann, F. Fiedler, A. Nohl, and Surender Parupalli, *A Methodology and Tooling Enabling Application Specific Processor Design*, IEEEConference on VLSI Design, 2005
An Evaluation of CoWare Inc.'s Processor Designer Tool Suite for the Design of Embedded ProcessorsJonathan D. Franz, M.S.E.C.E.

[3] A. Hoffmann, H. Meyr, and R. Leupers, *Architecture Exploration for Embedded Processors with LISA*, Kluwer Academic Publishers, Boston, 1 ed., 2002.

[5] P. Ienne and R. Leupers, *Customizable Embedded Processors*, Morgan Kaufmann, Boston, 1 ed., 2006

[6] A. Hoffmann, A. Nohl, G. Braun, and H. Meyr, *A Survey on Modeling Issues Using the Machine Description Language LISA*, , 2001

[7] G. Hadjiyiannis, S. Hanono, and S. Devadas, *ISDL: An Instruction Set Description Language for Retargetability*, in Proc. of the Design Automation Conference (DAC), Jun 1997

[8] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, *EXPRESSION: A Language for Architecture Exploration Compiler/Simulator Retargetability*, In Proc. of the Conference Design,Automation & Test in Europe, Mar. 1999

[9] Peter Marwedel, *The MIMOLA Design System: Tools for the Design of Digital Processors,* In Proceedings of the 21st Design Automation Conference, pages 587-593, 1983

[10] S. Yang, Y. Qian, H. Tie-Jun, S. Rui, and H. Chao-Huan, *A New HW/SW Codesign Methodology to Generate a System Level Platform Based on LISA,2005*

[11] R. Gonzales, *XTensa: A Configurable and Extensible Processor,* IEEE Micro,Mar. 2000

[12] Vincent P. Heuring and Harry F. Jordan, *Computer System Design and Architecture Second Edition*, Pearson Education Inc., 2004

[13] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA –Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures. In *Proc. of the Design Automation Conference (DAC)*, New Orleans, June 1999

[14] A. Hoffmann, A. Nohl, G. Braun, O. Schliebusch, T. Kogel, and H. Meyr. A Novel Methodology for the Design of Application Specific Instruction Set Processors (ASIP) Using a Machine Description Language. *IEEE Transactions onComputers-Aided Design*, Nov. 2001

[15] Zivojnovi, S.Pees & H.Meyr;LISA–machine description language and generic machine model for HW/SW codesign in Proceedings of the IEEE Workshop on VLSI Signal Processing,San Francisco,Oct 1996

[16] httpwww .ertrwth_aachende_lisa_lisahtml_

[17] S. Pees, V. Zivojnovic, A. Ropers, and H. Meyr, \Fast Sim-ulation of the TI TMS 320C54x DSP," in Proc. Int. Conf. on Signal Processing Application and Technology (IC-SPAT), (San Diego), pp. 995{999, Sep. 1997.

[18] Stefan Pees,Andreas Hoffmann,Vojin Zivojnovic, Heinrich Meyer Meyr LISA - Machine Description Language forCycle_Accurate Models of Programmable DSP Architectures

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