

Generating Production Quality Software Development Tools Using a Machine Description Language

Andreas Hoffmann, Achim Nohl, Stefan Pees, Gunnar Braun, Heinrich Meyr
Integrated Signal Processing Systems (ISS), RWTH Aachen
Templergraben 55, 52056 Aachen, Germany
hoffmann[nohl,pees,braun,meyr]@iss.rwth-aachen.de
<http://www.iss.rwth-aachen.de/lisa>

Abstract

This paper presents a methodology to automatically generate production quality software development tools for programmable architectures using the machine description language LISA. Various architectures presenting diverse architectural originalities will be presented and the feasibility of automatically generating simulator, assembler, linker and graphical debugger frontend will be discussed. The presented approach is not limited to a fixed abstraction level – case studies of the Texas Instruments C62x and C54x, the Analog Devices ADSP2101 as well as the ARM7 will show the applicability of the methodology from cycle/phase to instruction accurate models.

1 Introduction

All embedded processors like Digital Signal Processors (DSP) and micro-controllers (μ C) need a complete tool set consisting of code-generation and simulation tools. However, building simulator, assembler, linker and graphical debugger frontend *manually* for new architectures is extremely error-prone and tedious. The lengthy process of matching the simulator to an abstract model of the processor architecture might be performed several times within the development of a programmable System-On-Chip (SOC) design. Moreover, co-simulation of hardware and software puts specific requirements on the simulation accuracy on the programmable side, while simulator performance is still an important factor. Hence, processor models on different levels of abstraction are demanded to provide increased simulation accuracy as well as outstanding fast simulation performance. The efforts of writing software development tools can be reduced significantly by using a retargetable approach based on a machine description.

The Language for Instruction Set Architectures (LISA) [5] was developed for the automatic generation of 100% consistent development tools. The LISA language is designed for the formalized description of programmable ar-

chitectures, their peripherals, and interfaces. A LISA processor description covers the instruction-set, the behavioral and the timing model of the underlying hardware, thus providing all essential information for the generation of a complete set of development tools including compiler, assembler, linker and simulator. Changes in the hardware are easily transferred to the LISA model and are automatically applied to the generated tools. Moreover, the speed and the functionality of the generated tools allow usage after the product development has been finished. Therefore there is no need to rewrite the tools in order to upgrade them to production quality standard.

2 Related 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 simulation 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 [7]. Another problem is that the extraction of the instruction set is a highly complex, manual task and instruction set information, like e.g. assembly syntax cannot be obtained from HDL descriptions.

The machine description language nML was developed at TU Berlin [2] and adopted in several projects [4]. 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.

These restrictions also apply to the approach of ISDL [3] which is very similar to nML. The approach based on the language EXPRESSION [1] incorporates particular mechanisms for the description of memory hierarchies and focuses on retargeting high level language compilers. However, no results are published that indicate the applicability for cycle-accurate simulation purposes.

To summarize the review, none of the approaches above does support modeling of cycle/phase-accurate architectures including pipelines or the generation of very fast, production quality tools.

3 Software development tools

The LISA tool-suite is a set of development tools, which is automatically generated from LISA machine descriptions. It includes assembler, linker, simulation compiler and simulator as well as a graphical debugger frontend. Providing these tools, a complete software development environment is available which ranges from the assembly source file up to simulation within a comfortable graphical debugger frontend. Figure 1 shows the components of the LISA tool-suite.

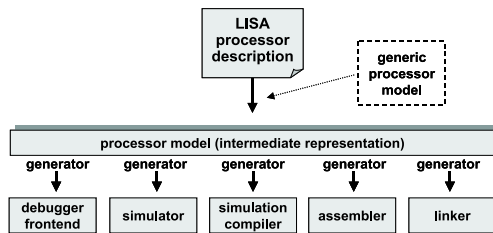


Figure 1. Automatic tool-chain generation.

LISA Simulator and Simulation Compiler The LISA simulator utilizes the technique of compiled simulation for outstanding simulation performance [8]. Compiled simulators are application-specific simulators, which are generated from the target application file by inserting a translation step before the simulation is run. The translation of the application is performed by a tool called simulation compiler.

A major task in compiled processor simulation is to determine the temporal order of executed operations – in particular for pipelined architectures. The simulation compiler utilizes three scheduling principles to generate the most efficient simulator for the underlying architecture model, namely dynamic scheduling, static scheduling, and instruction-based code translation. While the former two techniques are used for pipelined processor models, the latter is a technique that may be applied to instruction-set accurate models and cycle accurate models without an instruction pipeline, resulting in an enormous increase in simulation speed. Principles and implementation aspects of these simulation techniques are discussed in [9].

LISA Assembler The LISA assembler translates meaningful text-based instructions into object code for the respective programmable architecture. Symbolic names for opcodes, memory-contents and branch addresses simplify the programming enormously. Beside the processor specific instruction-set, the LISA assembler provides a set of

pseudo-instructions to control the assembling process (directives). This concerns data initialization, reasonable separation of the program into sections, handling of symbolic identifiers for numeric values and branch addresses. The re-targetability of the LISA assembler requires support for unrestricted instruction word-sizes and the handling of complex assembly syntax.

LISA Linker Large programs do consist of multiple modules, where each module is a unit of logically grouped functions, variables and declarations. The modules can be assembled separately. The LISA linker combines these multiple object files into a single executable object.

The linking process is controlled by a linker command file which keeps a detailed model of the target memory environment and an assignment table of the module sections to their respective target memories.

Graphical debugger frontend The LISA debugger frontend is a generic GUI for the generated LISA simulator (see figure 2). It visualizes the internal state of the simulation process. Both the C-source code and the disassembly of the application as well as all configured memories and (pipeline) registers are displayed. All contents can be changed in the frontend at runtime of the application. The process of the simulator can be controlled by stepping and running through the application and setting breakpoints.

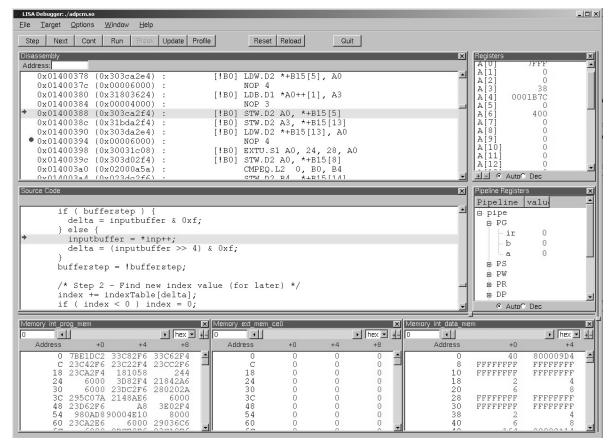


Figure 2. Graphical debugger frontend

4 Considered Architectures

To examine and analyze the modeling abilities of LISA as well as the feasibility of generating software development tools, four different architectures have been considered. The architectures were carefully chosen to cover a broad range of architectural characteristics and are widely used in the field of digital signal processing (DSP) and micro-controllers (μC). Moreover, the abstraction level of

the models ranges from phase accuracy (TMS320C62x) to instruction-set accuracy (ARM7).

- **ARM7** The ARM7 core is a 32 bit micro-controller of Advanced RISC Machines Ltd. The realization of a LISA model of the ARM7 μ C at instruction-set accuracy took approx. two weeks.
- **ADSP2101** The Analog Devices ADSP2101 is a 16 bit fixed-point DSP with 20 bit instruction-word width. The realization of the LISA model of the ADSP2101 at cycle accuracy took approx. 3 weeks.
- **TMS320C54x** The Texas Instruments TMS320C54x is a high performance 16 bit fixed-point DSP with a five stage instruction pipeline. The realization of the model at cycle accuracy (including pipeline behavior) took approx. 8 weeks.
- **TMS320C62x** The Texas Instruments TMS320C62x is a general-purpose fixed-point DSP based on a very long instruction-word (VLIW) architecture containing an eleven stage pipeline. The realization of the model at phase accuracy (including pipeline behavior) took approx. 6 weeks.

The characteristics of the LISA description (coding, syntax, behavior) of these architectures are discussed in the following. Besides, it will be shown that the generated tools are working accurately and with a satisfactory level of speed.

5 Architectural characteristics

Every architecture has its characteristics particularly with regard to the instruction-set or the structure. However, even for complex architectures the high modeling efficiency of LISA allows to realize a model of the chosen architecture in a reasonable amount of time. This section focuses on the modeling of one outstanding characteristic for each presented architecture.

5.1 VLIW

The C62x DSP of Texas Instruments is a VLIW architecture with 256 bit instruction word width. The instruction word is fetched as a whole from memory and then partitioned into eight micro-instructions which are dispatched into the execution pipeline. For the modeling of word-sizes greater than the maximum word size of the simulating host, LISA provides a dedicated type `bit` which is parameterized by the resources bit-width.

```
RESOURCE
{
    ...
    unsigned bit[256] insn_register;
    ...
}
```

is an excerpt of the resource-declaration of the C62x LISA model showing the declaration of the instruction register.

The `bit` data type in LISA contains a set of overloaded operators and can thus be used in the behavioral code of the LISA model as any other C-type can. For modeling of signed and unsigned operations on the processor resources, the `bit` data-type can be attributed with a `signed` or `unsigned` keyword.

```
OPERATION Dispatch IN pipe.DP
{
    DECLARE
    {
        GROUP micro_insn_1, micro_insn_2, micro_insn_3,
              micro_insn_4, micro_insn_5, micro_insn_6,
              micro_insn_7, micro_insn_8 =
              { decode_instruction };
    }
    CODING
    {
        insn_register ==
            (micro_insn_1) || (micro_insn_2)
            || (micro_insn_3) || (micro_insn_4)
            || (micro_insn_5) || (micro_insn_6)
            || (micro_insn_7) || (micro_insn_8)
    }
}
```

Example 1: Formal expression of parallel instructions

Once the VLIW instruction-word is in the dispatcher it is split into eight micro-instructions which are all of the same type, i.e. are processed by identical decoders in hardware. To prevent writing separate LISA-code for all eight instructions, they are merged onto the same coding-tree. Example 1 shows the respective LISA-code taken from the C62x model. The logical OR between the micro-instruction indicates parallel execution.

5.2 Multiple instruction words

Frequently, architectures are employed which utilize instructions made up of multiple instruction words. The TMS320C54x DSP contains instructions that can be composed of either one, two or even three instruction words. The second/third instruction word mostly carries immediate values or operands but can also be part of the opcode of the instruction.

The correlation between the different instruction types and the decoders is established in the coding-root. Example 2 shows the mapping of each instruction type described in the coding-root of the LISA description onto the respective coding-tree – `insn_reg_DC`, `insn_reg_FE` and `insn_reg_PF` are thereby processor resources carrying the instruction words whereas `Type1`, `Type2` and `Type3` represent the three coding-trees for the respective instruction type. The branching into the three coding-trees is controlled by a SWITCH-CASE statement which is part of the LISA control structure. The selection of the appropriate instruction type is based on an enumeration type with the effect that all coding-trees are searched until the currently processed instruction word is identified.

```

OPERATION Decode IN pipe.DC
{
  DECLARE {
    ENUM InsType = { Type1, Type2, Type3 };
    GROUP Decode_16 = { ALU || Branch };
    GROUP Decode_32 = { ALU_op || Branch_op };
    GROUP Decode_48 = { ALU_op1_op2 };
    SWITCH (InsType)
    {
      CASE Type1: /* 16 bit instruction */
        CODING { insn_reg_DC == Decode_16 }
        /* 16 bit instruction with 16 bit address */
      CASE Type2:
        CODING { (insn_reg_DC == Decode_32)
          && (insn_reg_FE == Operand) }
        /* 32 bit instruction with 16 bit address */
      CASE Type3:
        CODING { (insn_reg_DC == Decode_48)
          && (insn_reg_FE == Operand1)
          && (insn_reg_PF == Operand2) }
    }
  }
}

```

Example 2: Formal expression of multiple insn-words

5.3 Non-coherent coding elements

Especially in low-power architectures an optimal usage of the instruction word is required. Here, coherent coding-fields are split into multiple pieces spread over the instruction word. In the ARM7 μ C, this is the case for the operand in ALU-instructions.

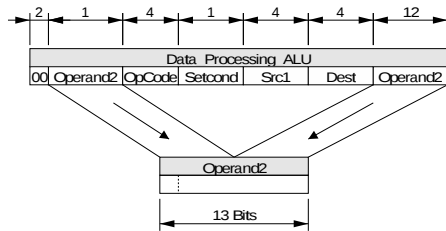


Figure 3. Non-coherent coding in the ARM7

The coding of Operand2 is distributed over the coding of the operation Data.Processing.ALU. Example 3 shows how the merging of the coding-tree is modeled in LISA. The distributed coding element is attributed with the position in the coding of the LISA operation the element is referring to.

```

OPERATION Data_Processing_ALU
{
  DECLARE {
    INSTANCE Operand2;
    REFERENCE { Src1, Dest, Setcond };
  }
  CODING
  {
    00b Operand2=[12..12] OpCode Setcond
      Src1 Dest Operand2=[0..11]
  }
}

```

Example 3: Modeling non-coherent coding in LISA

5.4 Algebraic instruction syntax

Sometimes, architectures programmed primarily in assembly use a C like assembly instruction syntax to ease programming. The Analog Devices ADSP2101 features such an algebraic programming syntax. This means that the instruction syntax is not fragmented into a mnemonic and a list of operands but formulated as an algebraic expression. An example for an algebraic instruction would be :

ADD Y,X,Z $\iff$ Z=X+Y

The LISA control-flow syntax can be used to express the syntax dependency on the coding of the instruction-word. Example 4 shows an excerpt of the model of the ADSP2101.

```

OPERATION ALU_Instructions
{
  DECLARE {
    REFERENCE Destination, Xoperand, Yoperand;
    GROUP Opcode = { ADD || SUB || AND };
    CODING { 0011b Opcode }
    SWITCH(Opcode) {
      CASE ADD:
        SYNTAX { Dest "=" Xop "+" Yop }
      CASE SUB:
        SYNTAX { Dest "=" Xop "-" Yop }
      CASE AND:
        SYNTAX { Dest "=" Xop "&" Yop }
    }
  }
}

```

Example 4: Algebraic instruction syntax in LISA

Here, depending on the Opcode of the instruction word the respective syntax is chosen via the SWITCH-CASE control structure.

6 Efficiency of the generated tools

To evaluate the applicability and efficiency of the generated tools, we compared them to the commercially available tools provided by the semiconductor vendors. Measurements took place on a AMD Athlon system with a clock frequency of 800 MHz. The system is equipped with 256 MB of RAM and is part of the networking system. It runs under the operating system Linux, kernel version 2.2.14. Tool compilation was performed with GNU gcc, version 2.92.

The generation of the complete tool-suite (simulator, assembler, linker and debugger frontend) takes, depending on the complexity of the considered model, between 12 sec (ARM7 μ C instruction-set accurate) and 67 sec (C6x DSP phase accurate).

6.1 Performance of simulator

Figures 4 and 5 show the speed of the generated simulators in instructions per second/cycles per second respectively. Simulation speed was quantified by running an ap-

plication on the respective simulator and counting the number of proprocessed cycles. The simulated application on all architectures is an ADPCM G.721 (Adaptive Differential Pulse Code Modulation) coder/decoder.

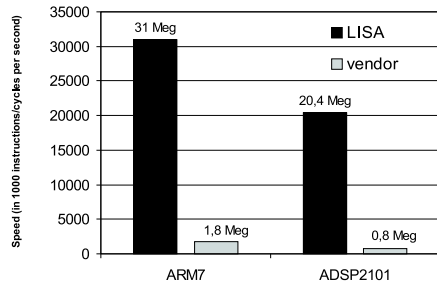


Figure 4. Speed of ARM7 and ADSP2101

As expected, the compiled simulation technique applied by the generated LISA simulators outperforms the vendor simulators by one to two orders in magnitude.

For the ARM7, ADSP2101 and the C54x, static scheduling was applied which is the highest possible grade of prediction in compiled simulation. Considering an ARM7 μ C running at a frequency of 25 MHz, the software simulator running at 31 MIPS even outperforms the real hardware. This makes application development suitable before the actual silicon is at hand. Due to its superscalar instruction dispatching mechanism the simulator for the C62x DSP uses compiled simulation with dynamic scheduling.

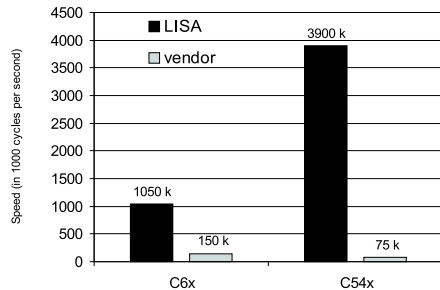


Figure 5. Speed of C62x and C54x

6.2 Performance of assembler and linker

The generated assembler and linker are not as time critical as the simulator is. It shall be mentioned though that the performance (i.e. the number of assembled/linked instructions per second) of the automatically generated tools is comparable to that of the vendor tools.

7 Conclusion and Future Work

In this paper, we presented a methodology to automatically generate software development tools for programmable architectures using the machine description lan-

guage LISA. To enable modeling and automatic tool generation for a wide range of architectures featuring structural originalities as VLIW, multiple instruction words and non-coherent coding fields as well as instruction-set originalities as algebraic assembly syntax the LISA language contains special syntax elements.

In case studies, LISA models were realized and tools successfully generated for the ARM7 μ C, the Analog Devices ADSP2101, the Texas Instruments C62x and the Texas Instruments C54x on instruction-set/cycle/phase accuracy respectively. Due to the usage of the compiled simulation principle, the generated simulators run by one to two orders in magnitude faster than the vendor simulators. Moreover, the generated assembler and linker can compete well in speed with the vendor tools.

Our future work will focus on modeling further real-life processor architectures and retargetable compiled simulators that provide the third step of compilation [6] – operation instantiation. 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 VHDL-code synthesis for the control-path and the instruction decoder of the modeled architecture.

References

- [1] A. Halambi, P. Grun, et al. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 1999.
- [2] A. Fauth, M. Freericks, and A. Knoll. Generation of hardware machine models from instruction set descriptions. In *Proc. of the IEEE Workshop on VLSI Signal Processing*, 1993.
- [3] 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.
- [4] M. Hartoog, J. Rowson, et al. Generation of software tools from processor descriptions for hardware/software codesign. In *Proc. of the Design Automation Conference (DAC)*, Jun. 1997.
- [5] S. Pees, A. Hoffmann, V. Živojnović, and H. Meyr. LISA – Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures. In *Proceedings of the Design Automation Conference (DAC)*, pages 933–938, New Orleans, June 1999.
- [6] 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)*, pages 995–999, San Diego, Sep. 1997.
- [7] J. Rowson. Hardware/Software co-simulation. In *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 1994.
- [8] S. Pees, A. Hoffmann, and H. Meyr. Retargeting of compiled simulators for digital signal processors using a machine description language. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 2000.
- [9] V. Živojnović and H. Meyr. Compiled HW/SW co-simulation. In *Proc. of the DAC 1996 – Las Vegas*, pages 690–695, June 1996.