Retargetable Compiled Simulation of Embedded Processors Using a Machine Description Language

Stefan Pees, Andreas Hoffmann and Heinrich Meyr
Integrated Signal Processing Systems, RWTH Aachen, Germany

Association for Computing Machinery, Inc., 1515 Broadway, New York, NY 10036, USA
Tel: (212) 555-1212; Fax: (212) 555-2000

Fast processor simulators are needed for the software development of embedded processors, for HW/SW cosimulation systems and for profiling and design of application specific processors. Such fast simulators can be generated based on the machine description language LISA. Using this language to model processor architectures enables the generation of compiled simulators on various abstraction levels, assemblers and compiler back-ends. The article discusses the requirements of software development tools on processor models and presents the approach based on the LISA language. Furthermore, the implementation of a retargetable environment consisting of compiled simulator, debugger and assembler is presented. Measurements for a verified, cycle-based LISA model of the TI TMS320C62x DSP show that this approach achieves between 37x and 170x higher simulation speed compared to a commercial simulator using a standard technique and the same accuracy level.

Categories and Subject Descriptors: I.6.5 [**Simulation and Modeling**]: Model Development—*Modeling Methodologies*; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Embedded Systems*; B.1.2 [**Control Structures and Microprogramming**]: Control Structure Performance Analysis and Design Aids—*Simulation*

General Terms: Design, Languages, Performance, Verification

Additional Key Words and Phrases: Machine Description Languages, Compiled Simulation, DSP Processors, Instruction Set Simulators, Processor Modeling and Simulation, HW/SW cosimulation, System-on-Chip

1. INTRODUCTION

In consumer electronics and telecommunications two major trends are the increasing system complexity and shorter design cycles due to time-to-market constraints. Driven by the advances in semiconductor technology combined with the need for new applications like digital TV and wireless broadband communications, the amount of system functionality realized on a single chip is growing enormously.

Higher integration and thus increasing miniaturization have led to a shift from using distributed hardware components towards heterogeneous system-on-chip (SOC) designs [Birnbaum and Sachs 1999]. Due to the complexity introduced by such SOC designs and time-to-market constraints, the designer's productivity has become the vital factor for successful products. For this reason a growing amount of system functions and signal processing algorithms is implemented in software rather then in hardware by employing programmable processor cores.

Integrating embedded processors into a hardware environment on a single chip raises new challenges in the area of verification based on cosimulation. Among others, many HW/SW cosimulation systems use instruction set based processor models which are wrapped in bus-interface models but modern processor architectures with complex pipelines can hardly be modeled based on this approach. For the system-level verification of real-time systems, cycle-based processor models are required [Guerra, L. et *al.* 1999; Earnshaw et al. 1997]. At the same time, *simulation speed* is critical for the verification of such systems and thus an important issue in simulator design [Olukotun et al. 1998; Hartoog, M. et *al.* 1997; Rowson 1994].

The principle of compiled simulation is to take advantage of a priori knowledge and move frequent operations from simulation run-time to compile-time with the goal of providing the highest possible simulation speed. In contrast to interpretive simulators, this approach requires a translation step to be performed before the simulation can be run. Algorithms used in embedded systems typically consist of many loops and code has a high locality. Therefore, the additional effort of translating the application code to a compiled simulation pays off. This is because frequent operations such as fetching, dispatching and decoding instruction words and determination of operands and execution modes are performed only once at compile-time instead of every time the respective instruction is executed at run-time of the simulation. In figure 1, both principles are compared for a four-stage pipelined processor with the stages instruction fetch (IF), instruction decode (ID), operand fetch (OF), and execute (EX). The simulation compiler produces an application specific simulation program consisting of configured execute-operations. The IF, ID and OF operations are eliminated at simulation run-time.

Compiled simulation of programmable DSP architectures was introduced to speed up the instruction set simulation [Živojnović et al. 1995] and was extended to cycle-accurate models of pipelined processors [Pees et al. 1997]. So far, the approaches addressing the particular requirements of compiled simulation of DSPs are targeted to a specific processor architecture using a handwritten simulation compiler. However, 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 a machine description [Barbacci 1981; Fauth and Knoll 1993]. This article explores principles of retargetable software development tools with focus on compiled simulation. Furthermore, an implementation based on the machine description language LISA [Živojnović et al. 1996; Pees et al. 1999] is presented.

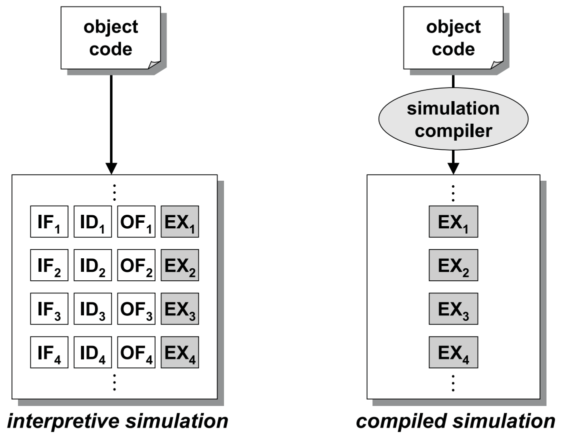


Fig. 1. Compiled vs. interpretive simulation.

2. PROCESSOR MODELS FOR SOFTWARE DEVELOPMENT TOOLS

All embedded processors like DSPs, microcontrollers and ASIPs need a complete software development tool suite consisting of compiler, assembler, linker, simulator, and software debugger [Flynn et al. 1999]. However the model requirements of tasks performed by these tools are quite different.

	memory model	*resource model*	*instruction set model*	*behavioral model*	*timing model*
compiler	register allocation	instruction scheduling	instruction selection	-	instruction scheduling
assembler	-	-	instruction translation	-	-
linker	memory allocation	-	-	-	-
simulator	simulation of storage	hazard detection	decoder/ disassembler	operation simulation	operation scheduling
debugger	display configuration	profiling	-	-	-

Fig. 2. Model requirements of SW development tools.

The process of generating program development tools requires information on architecture properties and the instruction set definition as depicted in figure 2. A suitable processor description for simulator generation should provide information consisting of the following model components.

—The *memory model* lists the registers and memories of the system with their respective bit widths, ranges, and aliasing. The compiler gets information on available registers and memory spaces. The memory configuration is provided to perform object code linking. During simulation, the entirety of storage elements represents the state of the processor which can be displayed in the debugger.

—The *resource model* describes the available hardware resources and the resource requirements of operations. Resources reproduce properties of hardware structures which can be accessed exclusively by one operation at a time. The instruction scheduling of the compiler depends on this information.

—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. Compilers and assemblers can identify instructions based on this model. The same information is used at the reverse process of decoding and disassembling.

—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.

—The *timing model* specifies the activation sequence of hardware operations and units. The instruction latency information lets the compiler find an appropriate schedule and provides timing relations between operations during simulation.

3. REQUIREMENTS AND LIMITATIONS

This approach is based on processor descriptions in the LISA language which is designed for the formalized description of programmable architectures, their peripherals, and interfaces [Živojnović et al. 1996; Pees et al. 1999]. Its development was motivated by the lack of approaches which are able to produce cycle-accurate models of modern embedded processor architectures (for example the TI TMS320C62xx [Texas Instruments 1998]) and to cover the instruction-set. The language enables designers to describe DSPs and microcontrollers with SISD, SIMD, and MIMD execution structure and architectures with deep pipelines.

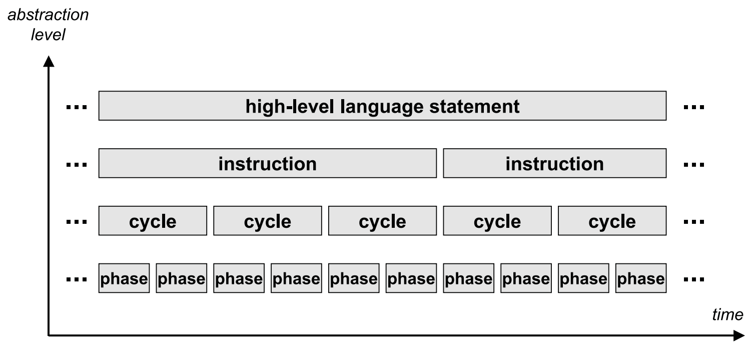


Fig. 3. Abstraction levels of LISA processor descriptions.

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. For example, one instruction of a processor may be represented by just one operation in case of an instruction set model (see figure 3). In another case, it may be described by a whole sequence of operations which represent the separate actions between clock cycles in case of a phase-accurate model. Common to all models described in LISA is the underlying zero-delay model. This means that all transitions are provided correctly at each control step. Control steps may be clock phases, clock cycles, instruction cycles or even higher levels as illustrated in figure 3. Events between these control steps are not regarded. However, this property meets requirements of current cosimulation environments [Synopsys 1999; Cadence 1999; Mentor Graphics 1999] on processor simulators to be used for HW/SW co-design [Guerra, L. et *al.* 1999; Earnshaw et al. 1997].

4. 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 cycle-based or instruction-level processor simulation has a number of disadvantages. They cover a huge amount of hardware implementation details which are not needed for performance evaluation, cycle-based simulation and software verification. Moreover, the description of detailed hardware structures has a significant impact on simulation speed [Olukotun et al. 1998; Rowson 1994]. Another problem is that the extraction of the instruction set is a highly complex, manual task and some instruction set information, like e.g. assembly syntax cannot be obtained from HDL descriptions at all. The simulator of the FlexWare project [Paulin, P. et *al.* 1995] is based on a partially reconfigurable VHDL model which is configured by an instruction set specification using the Insulin formalism. However, the reported simulation speed of 500-800 instructions on a Sparc 2 workstation is rather low for application software design. An approach of translating event-driven VHDL models into C++ simulators is reported [Krishnaswamy, V. et *al.* 1999], but the achieved simulation speed-up is comparatively low and absolute speed far from our requirements.

There are many publications on machine description languages providing instruction-set models. Most approaches using such models are addressing retargetable code generation [Stallman 1993; Araujo et al. 1996; Liem, C. et *al.* 1995; Engler 1996]. Other approaches address retargetable code generation and simulation. The approaches of Maril [Bradlee et al. 1991] as part of the Marion environment and a system for VLIW compilation [Rau 1996] are both using latency annotation and reservation tables for code generation. But models based on operation latencies are too coarse for *cycle-accurate* simulation.

The language nML was developed at TU Berlin [Freericks 1991][Fauth et al. 1995] and adopted in several projects [Hartoog, M. et *al.* 1997][Geurts, W. et *al.* 1996][Van Praet, J. et *al.* 1996] and also extended [Rajesh and Moona 1999]. However, the underlying instruction sequencer does not allow to describe the mechanisms of pipelining as required for cycle-based models. The main reason is the simple underlying instruction sequencer. Processors with more complex execution schemes and instruction-level parallelism like the Texas Instruments TMS320C6x cannot be described, even at the instruction-set level, because of the numerous combinations. The same restriction applies to ISDL [Hadjiyiannis et al. 1997] which is very similar to nML.

The EXPRESSION language [Halambi, A. et *al.* 1999] allows the cycle-accurate processor description based on a mixed behavioral/structural approach. But no results are published on simulation speed. The language RADL [Siska 1998] is derived from earlier work on LISA [Živojnović et al. 1996] and allows detailed pipeline description, but no results are provided on realized simulators based on this language.

The tool set of the SimpleScalar architecture provides five fast simulators with different accuracy levels [Burger and Austin 1997]. But the retargetability of this tool set is restricted to derivatives of the MIPS architecture. In the SimOS project [Rosenblum et al. 1995; Witchel and Rosenblum 1996] processor models on three different abstraction levels are used to simulate complete operating systems. However, the strategy of abstraction by direct execution requires binary code compatibility between host and target which usually does not apply to software development for embedded processors.

Our interest in addressing retargetable, compiled processor simulation [Živojnović et al. 1995] based on cycle-accurate models for a wide range of embedded processor architectures motivated the introduction of the language LISA which is used in our approach [Živojnović et al. 1996; Pees et al. 1999].

5. 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 embedded processors and the generation of compiled cycle-accurate simulators. For this reason, LISA includes improvements in the following areas:

—Capability to provide cycle-accurate processor models, including constructs to specify pipelines and their mechanisms including stalls, flushes, operation injection, etc;

—Extension of the target class of processors including SIMD, VLIW, and superscalar architectures of real world processor architectures;

—Explicit language statements addressing compiled simulation techniques;

—Distinction between the detailed bit-true description of operation behavior including side-effects for the simulation on the one hand and assignment to arithmetical functions for the instruction selection task of the compiler on the other hand which allows to freely determine the abstraction level of the behavioral part of the processor model;

—Strong orientation on the programming languages C/C++; LISA is a framework which encloses pure C/C++ behavioral operation description;

—Support for instruction aliasing and complex instruction coding schemes.

5.1 Language Overview

LISA descriptions are composed of *resources* and *operations*. The declared resources represent the storage objects of the hardware architecture (e.g. registers, memories, pipelines) which capture the state of the system. 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 which are defined in several sections[1].

[1]A complete reference of the language is provided here: [LISA Homepage 2000].

—The CODING section describes the binary image of the instruction word.

—The SYNTAX section describes the assembly syntax of instructions, operands, and execution modes.

—The SEMANTICS section specifies the transition function of the instruction.

—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 admissible operands or execution modes.

5.2 Resources

The resource section lists the definitions of all objects which are required to build the memory model and the resource model. A sample resource section of a simplified version of the DLX processor described in [Hennessy and Patterson 1996] is shown in example 1.

```
RESOURCE {
  PROGRAM_COUNTER int pc;
  REGISTER int R[0..31];

  PROGRAM_MEMORY char pmem[0..0x100000];
  DATA_MEMORY char dmem[0..0x100000];

  PIPELINE pipe = { IF; ID; EX; MEM; WB };
  PIPELINE_REGISTER IN pipe {
    ir_t ir;
    int npc, reg_a, reg_b, imm, alu;
    REGISTER bool cond;
  };
}
```

Example 1: Resource declaration of a simple DLX

The resource section begins with the keyword RESOURCE followed by (curly) braces enclosing all object definitions. The definitions are made in C-style and can be attributed with keywords like e.g. `REGISTER`, `PROGRAM_COUNTER`, etc. These keywords are not mandatory but they are used to classify the definitions in order to configure the debugger display. The resource section in example 1 shows the declaration of program counter, register file, memories, the five-stage instruction pipeline, and pipeline-registers.

The LISA language provides designated mechanisms to model pipelines of a processor architecture. The principle of this pipeline model is that operations are explicitly assigned to pipeline stages. Thus, the respective pipelines must be defined in the RESOURCE section. The declaration starts with the keyword PIPELINE, followed by an identifying name and the list of stages. The registers defined in the the `PIPELINE_REGISTER` statement produce multiple instances for each pipeline stage. As the pipeline is advanced, the pipeline registers are shifted as well.

5.3 Operations

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

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

Enclosed in braces, the operation body contains the different sections which describe the properties of the instruction set model, the behavioral model and the timing model. Operations are assigned to pipeline stages by using the keyword `IN` and providing the name of the pipeline and the identifier of the respective stage, such as:

```
OPERATION name_of_operation in pipe.EX
```

The ACTIVATION section in the operation description allows to activate other operations in the context of the current instruction. The activated operations are launched as soon as the instruction enters the pipeline stage the activated operation is assigned to. Non-assigned operations are launched in the pipeline stage of their activation. This mechanism is called *spatial activation*. In figure 4, this mechanism is illustrated for a the instruction `ADDI` of the DLX processor.

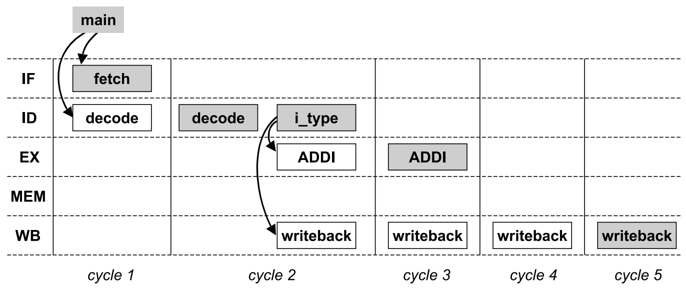


Fig. 4. Spatial activation of pipeline operations.

```
OPERATION main {
 ACTIVATION { fetch, decode }
}
OPERATION fetch IN pipe.FE {
  BEHAVIOR {
    ir.word = pmem[pc++];
    if ( PIPELINE(pipe.MEM).cond )
      npc = pc = PIPELINE(pipe.MEM).alu;
    else
      npc = ++pc;
  }
}
OPERATION decode IN pipe.DC {
  DECLARE { GROUP instruction = { i_type || r_type || j_type };}
  CODING AT(pc) { 0bx[32] => instruction }
  SYNTAX { instruction }
  BEHAVIOR { instruction(); }
}
```

Example 2: Fetch and decode stages of the DLX pipeline

The respective description of the cycle-based model is shown in the code examples 2 (describing activations of the first cycle) and 3 (activations in the second cycle).

```
OPERATION i_type IN pipe.DC {
  DECLARE {
    GROUP opcode = { ADDI || ADDUI || SUBI ||    [truncated...] };
    GROUP rs1, rd = { fix_register }; }
  CODING { opcode  rs1  rd  immediate }
  SYNTAX { opcode  rd  ","  rs1  ","  immediate }
  BEHAVIOR { reg_a = rs1;  imm = immediate;  cond = 0; }
  ACTIVATION { opcode, writeback }
}
OPERATION ADDI IN pipe.EX {
  CODING { 0b001000 }
  SYNTAX { "ADDI" }
  BEHAVIOR { alu = reg_a + imm; }
}
OPERATION writeback IN pipe.WB {
  DECLARE { REFERENCE rd; }
  BEHAVIOR { rd = alu; }
}
```

Example 3: The ADDI instruction of DLX

From operation `main`, the operations `fetch` and `decode` are activated (white box) and `fetch` is executed (shaded box) in the same cycle. In the following cycle, `decode` and then `i_type` is executed which in turn activates the operations `ADDI` and `writeback`. These operations execute during the next three cycles in their respective pipeline stage. In order to introduce a forwarding mechanism to the pipeline, the following operation can be introduced to the model:

```
OPERATION forwarding IN pipe.EX {
  BEHAVIOR {
    if ( PIPELINE(pipe.MEM).ir.itype.rd == ir.itype.sr1 )
      reg_a = PIPELINE(pipe.MEM).alu;
  }
}
```

Example 4: Forwarding mechanism

By accessing the pipelined registers such as the instruction register `ir`, the state of other instructions currently in the pipeline can be obtained. In the condition of the forwarding logic the bit fields of operand coding fields are compared.

Interrupts and exceptions typically may occur in every clock cycle. For this reason, it is most useful to place the check for exceptions into the `main` operation:

```
OPERATION main {
 ACTIVATION {
  if ( page_fault )
    { exception_handler }
  else
    { fetch, decode }
 }
}
```

Example 5: Exception handling

In addition, the operation execution can be delayed by multiples of control steps using the mechanisms of *temporal activation* which is denoted using semicolons. All operations listed in the ACTIVATION section which are separated by commas are launched with the same (or without) delay. Each semicolon delays following operations by one control step. Beyond the ordinary operation execution schemes, pipelines can be controlled explicitly by the designer.

In the BEHAVIOR section, pipelines are controlled by means of predefined functions *stall, shift, flush, insert*, and *execute* which are automatically provided by the LISA environment for each pipeline declared in the resource section. All these pipeline control functions can be applied to single stages as well as whole pipelines, for example:

```
PIPELINE(pipe.EX).stall();
```

The pipeline control function stall can be introduced in the DLX processor description to model an interlocking mechanism activated in case of data hazard between load instructions and immediately following instructions reading the destination register (see [Hennessy and Patterson 1996], pp. 152). The interlocking mechanism and the necessary changes to the operation `i_type` are described in example 6.

```
OPERATION interlocking IN pipe.ID {
  BEHAVIOR {
    // check for load instruction in EX stage...
    if ( ( PIPELINE(pipe.EX).ir.itype.opcode & 0x3F ) == 0x20 )
      // if data hazard then stall
      if ( PIPELINE(pipe.EX).ir.itype.rd == ir.itype.sr1 ) {
        PIPELINE(pipe.IF).stall();
        PIPELINE(pipe.ID).stall();
        PIPELINE(pipe.EX).stall();
      }
  }
}
OPERATION i_type IN pipe.DC {
    [...]
  ACTIVATION { forwarding, interlocking, opcode, writeback }
}
```

Example 6. Forwarding and interlocking

6. COMPILED SIMULATION

The objective of compiled simulation is to reduce the simulation time. In general, efficient run-time reduction is achieved by accelerating frequent operations. Here, the technique for accelerating operations is to use a priori knowledge during the translation of target program code into simulation code for the host.

The principle of compiled simulation for DSPs corresponds to the ideas that are already successfully implemented in the simulation of synchronous VLSI circuits [Barzilai, Z. et *al.* 1987], constant propagation in high-level language compilers [Aho et al. 1986], and that are used for static multi-processor scheduling [Lee and Messerschmitt 1987]. Such compiled simulators for DSPs have been realized for specific processor architectures [Pees et al. 1997]. Re-using the efforts for the implementation of the compiled techniques is extremely difficult since the compiled techniques are implemented in the so-called *simulation compiler* which is highly architecture dependent.

The processing of the simulation compiler can be split into three major steps which are depicted in figure 5.

—The step of **instruction decoding** determines the instructions, operands and modes from the respective instruction word. The pipeline structures found in modern embedded processors make it obvious that the simulation of these operations consumes a significant amount of simulation time. If we take for example the Texas Instruments TMS320C62x DSP, most instructions actually execute within only one pipeline stage (or cycle), whereas fetching, dispatching, and decoding

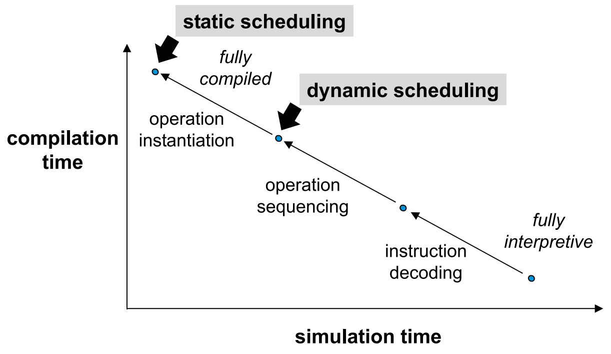


Fig. 5. Levels of compiled simulation.

operations require six pipeline stages (or cycles).

—The step of **operation sequencing** determines the sequence of operations to be executed for each instruction of the application program. This step can be implemented in a compiled simulator by generating a two dimensional table (see figure 6). One dimension (ordinate) of this table represents the instructions of the application program, the other (abscissa) contains pointers to functions that contribute to the transition function which drives the simulator into the next control step. In case of using the LISA language, these simulator functions correspond to the behavior code of operations. We call this compiled level *dynamic scheduling* because the scheduling of operations from overlapping instructions in the pipeline is performed at run-time of the simulation.

address	*simulator function*	*simulator function*	*simulator function*	*simulator function*
80561	&fetch	&decode	&ADDI	...
80562	&fetch	&decode	&SW	...
80563	&fetch	&decode	&MULT	...
...	...	...	...	

Fig. 6. Simulation table.

—**Operation instantiation and simulation loop unfolding** unfolds the simulation loop that drives the simulation into the next state and instantiates the respective simulation code for each instruction of the application program. This is implemented in the compiled simulator by generating individual behavioral code for each instruction of the DSP program. For cycle-accurate models of pipelined processor architectures, all possible traces of pipeline operations at every address of the application program have to be generated. During simulation execution, the valid trace is executed. This compiled level is called *static scheduling* because the overlapping of operations are statically scheduled at compile-time.

Between the two extremes of fully compiled and fully interpretive simulation, partial realization of the compiled principle is possible by implementing only some of these steps. Higher levels of compiled simulation can be achieved by investing substantially more effort in simulator design and exploiting highly architecture-specific properties.

A disadvantage of the highest level of compilation is that the static schedule cannot be changed at simulation run-time. Although there is not principal limitation to simulate exceptions or interrupts, self-modifying code cannot be handled with static scheduling. However the lower levels of compilation such as dynamic scheduling which is based on tables that can be (partially) updated (re-compiled) at simulation run-time if changes to program memory areas are detected. But self-modifying code appears very infrequently in the software for embedded processors – predominantly in the context of memory paging or boot loading.

7. SIMULATOR IMPLEMENTATION

7.1 Retargetable Environment

The implementation of our retargetable environment currently comprises simulator, debugger, assembler, and linker (see figure 7. Due to the open issues in the generation of production-quality code using customized compilers for DSP processors [Willems and Živojnović 1996], a retargetable compiler back-end is not yet implemented. In the following, we will discuss the generation of compiled simulators in detail.

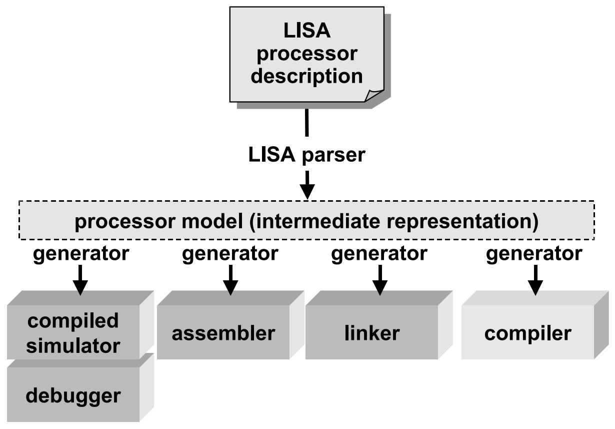


Fig. 7. Retargetable SW development tools.

In order to evaluate the applicability and efficiency of our approach, a compiled simulator based on dynamic scheduling is implemented in our experimental tool suite. As shown in Figure 8, a LISA compiler takes the processor description and translates it into two components – the processor-specific simulation compiler on the one hand and the simulation library on the other hand. The library consists of the variables representing the processor state and the transition functions which are described in the LISA operations of the processor description. The transition functions are implemented in C code which is generated from the behavior sections of all LISA operations. Executing an appropriate sequence of transition functions drives the processor into a new state. The sequences of transition functions are defined in the application specific simulation table (see figure 6). The generic processor model contains the arithmetic for any bit-width and the functionality of pipeline operations and control.

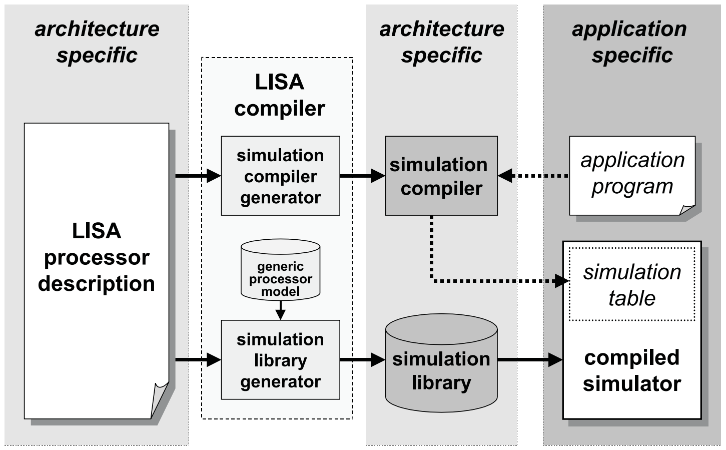


Fig. 8. Retargetable, compiled simulation tools.

The generated source code of the simulation compiler is compiled to an executable program that translates application programs into the simulation table which is part of the compiled simulator. The simulation table is an evaluated representation of the program memory which provides the particular configuration of each instruction. It contains the references to simulation library functions required for the respective instruction of the application program and for each pipeline stage. In addition, the simulation library is compiled and becomes a part of the final object file.

7.2 Processor Models

We have completely described several processor models in LISA to explore the applicability of the LISA language for modeling different processor architectures and generating the respective software development tools with a particular focus on compiled simulation. Figure 1 lists the processors which have been successfully described and which can be simulated in our environment. Because of the processor complexity, the different accuracy levels and significant differences in the designer's modeling experience, the numbers of lines required for the LISA description varies substantially.

processor	*accuracy level*	*pipeline*	*LISA description*
ARM 7	**instruction-set**	**no**	**3700 lines**
DLX	**cycle-based**	**5 stages**	**843 lines**
Texas Instruments C62x	**cycle-based**	**11 stages**	**10002 lines**
Texas Instruments C54x	**cycle-based**	**6 stages**	**12850 lines**

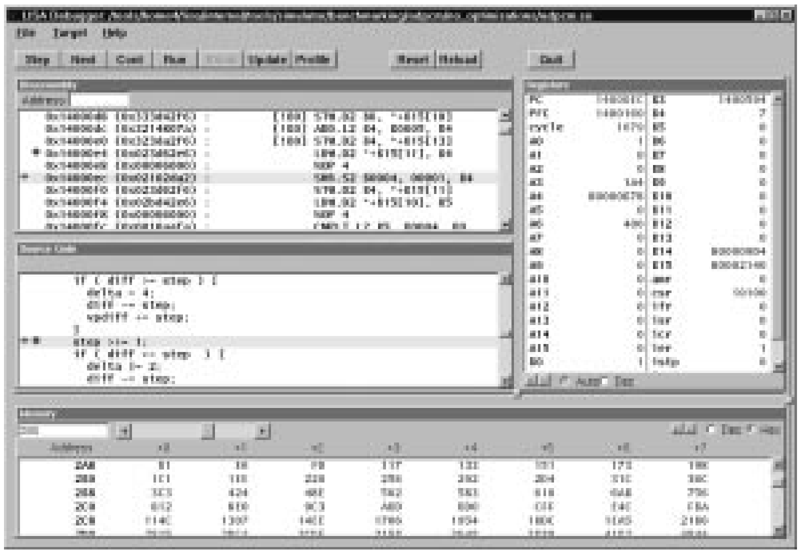
Table 1. LISA processor descriptions.

From the processor models described in LISA, we have chosen the Texas Instruments TMS320C62x DSP for a complete experimental analysis of the achievable simulation speed and a comparison to the current simulation technology. For the TI C54x DSP and the ARM 7 only the simulation speed was measured. The TMS320C62x including its memory interface was described in LISA as a cycle-based model. Although the architecture of this processor with two pipelines consisting of eleven pipeline stages is very complex, the LISA description was realized by one designer in 6 weeks. The complete translation of this model with the LISA compiler and the whole compilation of all architecture-specific components takes only 44 seconds (measured on a Sparc Ultra 5 workstation). About the same time was spent for a description of the TMS320C54x (six-stage pipeline). As a comparison, the custom implementation of a cycle-accurate, compiled simulator took more than 16 man-months.

The C62x model was verified against the commercial simulator from Texas Instruments `sim62x` by dumping traces of the processor state after each clock cycle for a large set of application programs. The traces of both simulators were compared on a cycle-by-cycle basis. For all algorithms of our test suite, the verification was successfully completed. In general, the verification procedure exceeds by far the design effort of the processor description. Improving this procedure will be an important topic of our future research.

7.3 Retargetable Debugger

Interaction with the compiled simulation is provided by a target-independent debugger. Program execution and the full processor state can be observed by loading the application program code and a dynamically linked library object which contains the compiled simulator for the application. The debugger displays are configured by a dedicated table in the linked library object. The table lists for each resource to be displayed the name, type (register, program counter, memory, etc.), size, and bit width. Figure 9 depicts a screen-shot of the debugger.



```
    if ( diff >= step ) {
      delta = 4;
      diff -= step;
      vpdiff += step;
    }
    step >>= 1;
    if ( diff >= step  ) {
      delta |= 2;
      diff -= step;
```

Fig. 9. Target-independent debugger.

The debugger controls simulation execution through an API to step through the program, run freely, handle breakpoints, and read and write registers and memory contents. Furthermore, the debugger allows to collect detailed profiling data in order to observe operation activity, use of address spaces, and resource accesses such as registers, memories and buses.

8. IMPLEMENTATION RESULTS

8.1 Measurement Conditions

In order to evaluate the simulation speed of our generated, compiled simulator of the TI C6201 we used the `sim62x`, version 2.0 which is part of the TI's software development tools for our reference. The measurements presented here are based on three typical DSP algorithms with different complexity, a FIR filter kernel, the ADPCM G.721 speech codec (encoder/decoder) pair, and the complete GSM speech encoder and codec. The GSM codec was the largest application filling the complete internal (on-chip) memory. For the FIR filter and the ADPCM codec, different implementations were investigated to explore the impact on operation density and instruction-level parallelism on the results. Except one of the FIR filter implementations, the applications were compiled from C source code with different levels of optimization (-O3, -O0, non-optimized).

All measurements were made on a Sun Ultra Sparc 5 with 333 MHz and 256 MB main memory. The execution time was determined by adding user and system time consumed by the process. For C compilation on the host the GNU C/C++ compiler, version 2.91.60 was used.

8.2 Simulation Speed

Simulation speed was quantified by running an application on the respective simulator and relating the simulation time to the processed number of cycles. The measurement results are listed in table 2.

application	*implementation*	*LISA simulator simulation speed [instructions/s]*	*TI simulator simulation speed [instruction/s]*	*speed-up factor*
FIR	no opt	**228.968,40**	**3.227,10**	**70,95**
FIR	-O3	**402.727,52**	**8.538,14**	**47,17**
FIR	asm	**437.424,97**	**11.901,29**	**36,75**
ADPCM	no opt	**259.833,84**	**3.214,34**	**80,84**
ADPCM	-O0	**307.245,53**	**4.747,87**	**64,71**
ADPCM	-O3	**266.742,51**	**4.363,13**	**61,14**
GSM	encoder	**256.715,24**	**3.225,46**	**79,59**
GSM	codec	**287.647,83**	**1.692,76**	**169,93**

Table 2. Measured simulation speed.

The reference simulator from TI achieved between 1.6k and 11.9k cycles/s whereas our generated compiled simulator runs with speeds between 228k and 437k cycles/s at the same accuracy level. These boundary values are found for two implementations of the FIR filter kernel. For full applications – the ADPCM and the GSM speech codecs – simulation speeds in the range of 250k to more than 300k instructions/s are achieved.

The resulting speed-up is shown in figure 10. The compiled simulator runs with factors between 36.8x to 170x faster than the interpretive reference.

It is remarkable that higher optimization during application compilation results in lower speed-ups. This effect is particular noticeable with the C62x. It can be explained by higher instruction-level parallelism in case of optimized code. Since the compiled technique profits from removing fetch and decoding operations, higher

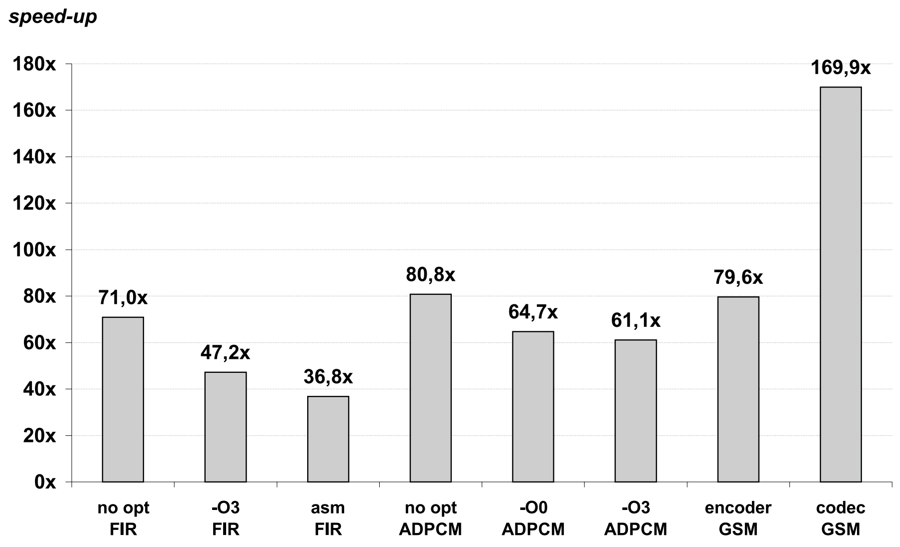


Fig. 10. Speed-up: LISA simulator vs. TI sim62x.

shares of execute stage operations reduce the freedom of the compiled technique to achieve speed-ups. The extraordinary speed-up of the GSM codec is caused by the low performance of the reference simulator when simulating large applications.

processor	*application*	*implementation*	*LISA simulator simulation speed [instructions/s]*
C54x	**ADPCM**	-O3	**267,000**
ARM 7	**ADPCM**	-O3	**2,130,000**

Table 3. Simulation speed.

The simulation speeds measured for the C54x DSP and the ARM 7 are listed in table 3. Although the C54x has a smaller pipeline, its complex mechanisms allow no higher simulation speed than the C62x model. However, the instruction set model of the ARM 7 runs at more than 2 million instructions per second. Considering the results from [Pees et al. 1997], we expect that using static simulation scheduling would even greatly improve this result.

8.3 Simulator Compilation

The first step of generating a simulation is to run the simulation compiler. Since it generates C/C++ source code, simulations have to be compiled with a C++ compiler afterwards. The required time for the simulation compiler, C++ compiler and linker are listed in table 4 for our set of application programs.

In order to predict the required preprocessing time for a given application program, an analysis of the compilation speed is more interesting which is displayed in figure 11.

Compilation speed is currently dominated by the processing time of the simulation compiler (which is an subject of future optimization). Otherwise, the

application	implementation	simulation compiler	C compiler	linker	total
FIR	no opt	0,60 s	0,11 s	0,21 s	0,92 s
FIR	-O3	0,70 s	0,14 s	0,21 s	1,05 s
FIR	asm	0,65 s	0,12 s	0,24 s	1,01 s
ADPCM	no opt	0,81 s	0,15 s	0,24 s	1,20 s
ADPCM	-O0	0,67 s	0,12 s	0,20 s	0,99 s
ADPCM	-O3	0,65 s	0,11 s	0,22 s	0,98 s
GSM	encoder	14,56 s	2,09 s	0,93 s	17,58 s
GSM	codec	15,72 s	2,20 s	1,00 s	18,92 s

Table 4. Generation, compilation and linkage time.

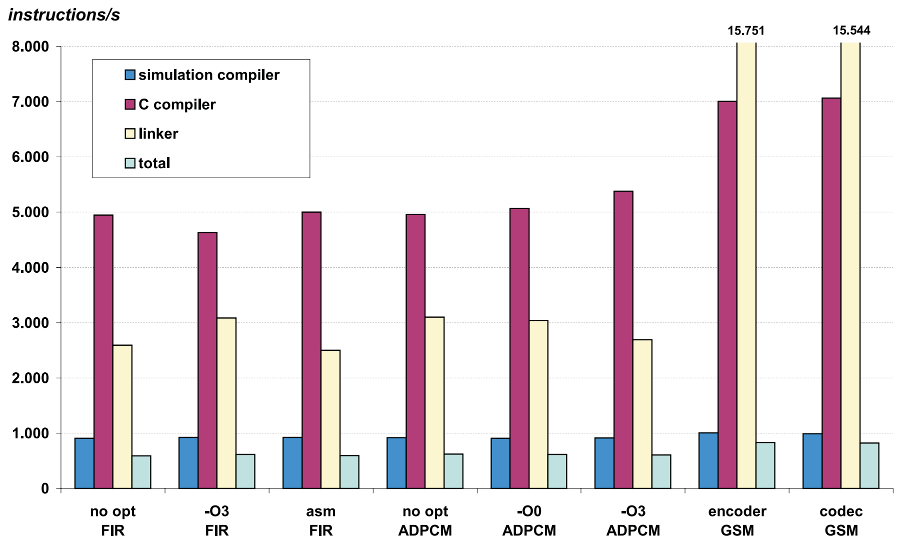


Fig. 11. Simulator compilation speed.

shares of the linker can nearly be neglected. The lowest total compilation speed of 592 instruction words/s is achieved for the smallest application – the assembly-implemented FIR filter. It increases for larger applications – the GSM codec – up to 821 instruction words/s. But in general, the very high simulation speed-ups are well worth investing short compilation time.

9. CONCLUSION AND FUTURE WORK

LISA is a language which aims at the formal description of embedded processors, their peripherals, and interfaces. The language supports different description styles and models at various abstraction levels. This article discussed the LISA modeling approach and the implementation of a retargetable environment of software development tools including compiled processor simulators. Using the LISA processor description significantly reduces the efforts of designing such environments and makes processor specification transparent and understandable to others than the authors. The high simulation speed of our approach helps to improve the productivity of processor and application designers.

Our future work will focus on an efficient verification methodology which compares LISA models against HDL descriptions. The generation of abstract models on the instruction set level (and higher) from a given cycle-based specification is part of our current research. Another subject of the ongoing research work is the integration of software simulators into HW/SW cosimulation environments. Furthermore, the goal of the ongoing language design is to address retargetable compiler back-ends as well.

REFERENCES

Aho, A., Sethi, R., and Ullman, J. 1986. *Compilers, Principles, Techniques and Tools.* Addison-Wesley.

Araujo, G., A., S., and Malik, S. 1996. Instruction set design and optimization for address computation in DSP architectures. In *Proc. of the Int. Symposium on System Synthesis (ISSS)* (1996).

Barbacci, M. 1981. Instruction set processor specifications (ISPS): The notation and its application. *IEEE Transactions on Computers C-30*, 1 (Jan.), 24–40.

Barzilai, Z. et *al.* 1987. HSS - A high speed simulator. *IEEE Trans. on CAD CAD-6*, 601–616. 1987.

Birnbaum, M. and Sachs, H. 1999. How VSIA answers the SOC dilemma. *IEEE Computer 32*, 8 (June), 42–50.

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

Burger, D. and Austin, T. 1997. The SimpleScalar tool set, version 2.0. *Computer Architecture News 25*, 3 (June), 13–25.

Cadence. 1999. *Cierto*
http://www.cadence.com/technology/hwsw. Cadence.

Earnshaw, R., Smith, L., and Welton, K. 1997. Challanges in cross-development. *IEEE Micro 17*, 4 (July/Aug.), 28–36.

Engler, D. 1996. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proc. of the Int. Conf. on Programming Language Design and Implementation (PLDI)* (May 1996).

Fauth, A. and Knoll, A. 1993. Automatic generation of DSP program development tools using a machine description formalism. In *Proc. of the ICASSP - Minneapolis, Minn.* (1993).

Fauth, A., Van Praet, J., and Freericks, M. 1995. Describing instruction set processors using nML. In *Proc. European Design and Test Conf., Paris* (Mar. 1995).

Flynn, M., Hung, P., and Rudd, K. 1999. Deep-submicron microprocessor design issues. *IEEE Micro 19*, 4 (July/Aug.), 11–22.

Freericks, M. 1991. The nML machine description formalism. Technical Report 1991/15, Technische Universität Berlin, Fachbereich Informatik, Berlin.

Geurts, W. et *al.* 1996. Design of DSP systems with Chess/Checkers. In *2nd Int. Workshop on Code Generation for Embedded Processors* (Leuven, Mar. 1996).

Guerra, L. et *al.* 1999. Cycle and phase accurate DSP modeling and integration for HW/SW co-verification. In *Proc. of the Design Automation Conference (DAC)* (Jun. 1999).

Hadjiyiannis, G., Hanono, S., and Devadas, S. 1997. ISDL: An instruction set description language for retargetability. In *Proc. of the Design Automation Conference (DAC)* (Jun. 1997).

Halambi, A. et *al.* 1999. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)* (Mar. 1999).

Hartoog, M. et *al.* 1997. Generation of software tools from processor descriptions for hardware/software codesign. In *Proc. of the Design Automation Conference (DAC)* (Jun. 1997).

HENNESSY, J. AND PATTERSON, D. 1996. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc. Second Edition.

KRISHNASWAMY, V. ET *al.* 1999. A procedure for software sythesis from VHDL models. In *Proc. of the Int. Conf. on Measurement and Modeling of Computer Systems* (Aug. 1999).

LEE, E. AND MESSERSCHMITT, D. 1987. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers C-36*, 24–35.

LIEM, C. ET *al.* 1995. Industrial experience using rule-driven retargetable code generation for multimedia applications. In *Proc. of the Int. Symposium on System Synthesis (ISSS)* (Sep. 1995).

LISA HOMEPAGE. 2000. *http://www.ert.rwth-aachen.de/lisa.* ISS, RWTH Aachen.

Mentor Graphics. 1999. *Seamless*
http://www.mentor.com/seamless. Mentor Graphics.

OLUKOTUN, K., HEINRICH, M., AND OFELT, D. 1998. Digital system simulation: Methodologies and examples. In *Proc. of the Design Automation Conference (DAC)* (Jun. 1998).

PAULIN, P. ET *al.* 1995. FlexWare: A flexible firmware development environment for embedded systems. In P. MARWEDEL AND G. GOOSENS Eds., *Code Generation for Embedded Processors* (1995). Kluwer Academic Publishers.

PEES, S., HOFFMANN, A., ZIVOJNOVIC, V., AND MEYR, H. 1999. LISA – Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures. In *Proc. of the Design Automation Conference (DAC)* (New Orleans, June 1999).

PEES, S., ŽIVOJNOVIĆ, V., ROPERS, A., AND MEYR, H. 1997. Fast Simulation of the TI TMS 320C54x DSP. In *Proc. of the Int. Conf. on Signal Processing Applications and Technology (ICSPAT)* (San Diego, Sep. 1997), pp. 995–999.

RAJESH, V. AND MOONA, R. 1999. Processor modeling for hardware software codesign. In *Int. Conf. on VLSI Design* (Goa, India, Jan. 1999).

RAU, B. 1996. VLIW compilation driven by a machine description database. In *Proc. 2nd Code Generation Workshop, Leuven, Belgium* (1996).

ROSENBLUM, M., HERROD, S., WITCHEL, E., AND GUPTA, A. 1995. Complete computer system simulation: The SimOS approach. *IEEE Parallel & Distributed Technology 3*, 4 (Winter), 34–43.

ROWSON, J. 1994. Hardware/Software co-simulation. In *Proc. of the Design Automation Conference (DAC)* (1994).

SISKA, C. 1998. A processor description language supporting retargetable multi-pipeline DSP program development tools. In *Proc. of the Int. Symposium on System Synthesis (ISSS)* (Dec. 1998).

STALLMAN, R. 1993. *Using and Porting GNU CC* (2.4 ed.). Cambridge, MA: Free Software Foundation.

Synopsys. 1999. *Eaglei*
http://www.synopsys.com/products/hwsw. Synopsys.

Texas Instruments. 1998. *TMS320C62x/C67x CPU and Instruction Set Reference Guide.* Texas Instruments.

VAN PRAET, J. ET *al.* 1996. A graph based processor model for retargetable code generation. In *Proc. of the European Design and Test Conference (ED&TC)* (Mar. 1996).

WILLEMS, M. AND ŽIVOJNOVIĆ, V. 1996. DSP-Compiler: Product Quality for Control-Dominated Applications? In *Proc. of the Int. Conf. on Signal Processing Applications and Technology (ICSPAT)* (Boston, Oct. 1996).

WITCHEL, E. AND ROSENBLUM, M. 1996. Embra: Fast and flexible machine simulation. In *Proc. of the Int. Conf. on Measurement and Modeling of Computer Systems* (May 1996).

ŽIVOJNOVIĆ, V., PEES, S., AND MEYR, H. 1996. LISA – machine description language and generic machine model for HW/SW co-design. In *Proc. of the IEEE Workshop on VLSI Signal Processing* (San Francisco, Oct. 1996).

ŽIVOJNOVIĆ, V., TJIANG, S., AND MEYR, H. 1995. Compiled simulation of programmable DSP architectures. In *Proc. of IEEE Workshop on VLSI Signal Processing* (Sakai, Osaka, Oct. 1995), pp. 187–196.