# Generation of Interpretive and Compiled Instruction Set Simulators

Rainer Leupers, Johann Elste, Birger Landwehr*
University of Dortmund
Dept. of Computer Science 12
44221 Dortmund, Germany
email: leupers@ls12.cs.uni-dortmund.de

*Abstract– Due to the large variety of different embedded processor types, retargetable software development tools, such as compilers and simulators, have received attention recently. Retargetability allows to handle different target processors with a single tool. In this paper, we present a system for automatic generation of instruction set simulators for a class of embedded processors. Retargetability is achieved by automatic generation of simulators from processor descriptions, given as behavioral or RT-level HDL models. The presented system is capable of bit-true simulation for arbitrary processor word lengths, and it generates both interpretive or compiled simulators. Experimental results for different processors indicate comparatively high simulation speed.*

## 1 Introduction

Today, the core functionality of many embedded systems is implemented by software running on embedded programmable processor cores, while dedicated ASIC hardware is mainly used for accelerating the execution of time-critical functions. Due to its high flexibility and its potential for reuse, embedded software usually provides a shorter time-to-market as compared to ASIC implementations.

The trend towards software implementation of embedded systems creates a need for appropriate development tools, e.g., C compilers and processor simulators. Since embedded software is typically developed on a workstation or PC host, such tools have to be designed as cross-compilers or cross-simulators, respectively, which generate and simulate code for target processors different from the host CPU. One major problem in this context is the large variety of embedded processor architectures that might need to be considered as a target machine for a particular application. While there is already a large amount of off-the-shelf microcontrollers, RISCs, and DSP cores available, many applications require even more customized architectures (application-specific instruction-set processors, ASIPs), so as to meet given performance or power consumption constraints.

In order to keep pace with the increasing number of different embedded processors, *retargetable* software development tools have been proposed. Retargetability denotes the capability of supporting a whole family of embedded processors with a single tool. An example are retargetable optimizing compilers, which have received significant interest during the last years (see [1, 2] for overviews). Retargetable tools enable the study of hardware-software trade-offs at the processor level. This is important, since many ASIPs may still be configured or parameterized, e.g., with respect to the word length, the number of functional units, or the register file sizes. Obviously, retargetable compilers are very useful to identify a good processor configuration for a particular application.

However, less effort has been spent so far on retargetable instruction set simulators, which are necessary for validation of embedded software. Besides flexibility, also high simulation speed is very important for such tools. Currently, there are two main techniques for processor cross-simulation: The classical *interpretive* simulation approach, which in each step decodes a target processor instruction and simulates it by host instructions, and the more recent *compiled* simulation approach [3], which generates an executable simulation program for a fixed application program, thereby moving the instruction decoding overhead to simulator generation time. While the compiled approach achieves significantly higher speed, its disadvantage is that the simulation program needs to be re-generated after each change in the application program. Furthermore, it excludes the simulation of self-modifying code, which is, however, rather rare in practice.

In this paper, we present a retargetable instruction-level simulator generator for a family of fixed-point DSPs, which provides several improvements as compared to previous work:

- Simulators are automatically generated from HDL processor models. There is no need to write target-specific simulator functions or to model the target processor in a simulator-specific language.
- Both interpretive and compiled simulators can be

*Publication: ASP-DAC, Hong Kong, Jan 1999, ©IEEE generated.

- Simulators can be generated for arbitrary target processor word lengths. This allows for bit-true simulation independent of the simulation host word length.

The organization of the paper is as follows. In section 2, we discuss related work in more detail. In section 3, a system overview is given, which also outlines the cooperation with a retargetable compiler. Section 4 describes the generation of simulators from processor models. Experimental results are provided in section 5.

## 2 Related work

Cross-simulators/debuggers are available from semiconductor vendors for many standard DSPs. Since these simulators are based on the interpretive simulation approach, they achieve a comparatively low simulation speed, typically in the order of several kilo-instructions per host CPU second. However, this speed is insufficient for bit-true evaluation of DSP algorithms on large test data sequences, which may require simulation times in the order of hours and days [4]. Furthermore, current commercial tools do not allow for retargetable processor simulation. In the following, we mention recent approaches to accelerate simulation and to provide retargetability.

The Insulin instruction-set simulator is part of the FlexWare system [5]. It is based on a configurable VHDL model of a "generic" processor. The target processor machine code is translated into assembly code for the generic processor. Execution of the generic assembly code is simulated by a standard VHDL simulator. This approach provides retargetability to a certain extent, but permits only rather slow simulation, typically less than 1 K instructions per second.

There are several approaches to retargetable simulation based on the nML processor modeling language. The Sigh/Sim system [6] automatically generates interpretive simulators from nML models. This is done by composing simulation functions for basic operations to more complex functions for complete machine instructions. Each time an instruction is fetched, its corresponding simulation functions are inserted into a queue of events to be simulated. This work has been extended in the CHECKERS system [7]. However, due to the interpretive simulation technique, both systems suffer from limited simulation speed. In [8], generation of compiled simulators from nML models has been proposed, so as to achieve higher simulation speed. For an ARM7 RISC, for instance, a speed of 150 K instructions per second has been achieved, as opposed to 22 K instructions obtained through the interpretive technique. However, several shortcomings of the nML language have been identified, e.g., missing support for specification of signed/unsigned arithmetic operators and local registers.

The SuperSim technique described in [3, 4] aims at maximizing simulation speed through compiled simulation. The target machine program is translated into an equivalent C simulation program, which is then compiled and executed on a simulation host. Since the instruction decoding is completely performed at simulator generation time, the simulation speed is very high and ranges from several hundred K instructions to several million instructions per second. The disadvantage, however, is the necessity to re-generate and re-compile the (possibly large) simulation program after each change in the application code. Therefore, compiled simulation is more useful, if the application code is already largely fixed. Furthermore, the generation of the C simulation program is done by processor-specific tools. Therefore, SuperSim cannot be classified as a retargetable system.

## 3 System overview

In order to overcome some limitations of previous work, we have implemented the Jacob system, which generates compiled and interpretive instruction-level simulators from processor models given in the MIMOLA HDL [9], which is similar to structural VHDL. The HDL model contains an integrated description of the processor controller and the data path. Jacob cooperates with the Record system, a retargetable compiler for a class of fixed-point DSPs (fig. 1). This processor class

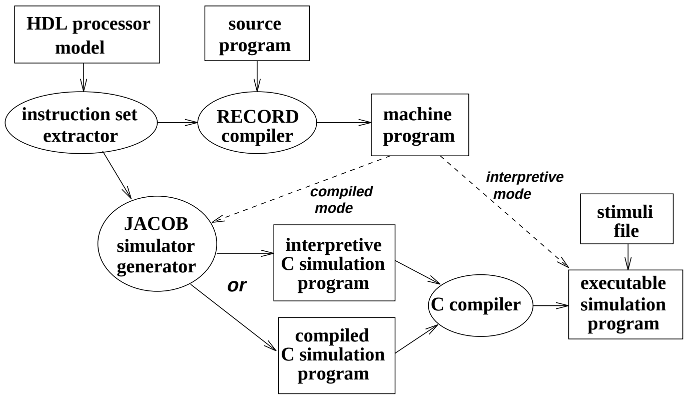
HDL processor model
source program
instruction set extractor
RECORD compiler
machine program
compiled mode
interpretive mode
JACOB simulator generator
or
interpretive C simulation program
compiled C simulation program
C compiler
stimuli file
executable simulation program

Figure 1: *System overview*

is characterized by a VLIW-like controller, single-cycle instructions, and a DSP-specific generic address generation unit (AGU) architecture (cf. [10] for a more detailed description of Record and the accepted processor class), while the data path and instruction format, including instruction-level parallelism, can be arbitrarily modeled by the user. The HDL processor model, which may comprise structural components like registers, functional units, or busses, is first preprocessed, so as to extract an internal instruction-set (i.e. purely behavioral) model of the target processor, while eliminating unnecessary structural details. This model is also used for code generation in the Record compiler [11].

In interpretive mode, Jacob reads the instruction-set model and generates a C simulation program, which is compiled onto the simulation host with a standard C compiler. The simulation program in turn reads a compiled or manually written target machine program and simulates it. In compiled mode, Jacob reads both the instruction-set model and a target machine program, and generates a program-specific C simulation program, which, as opposed to the interpretive mode, does not incur any instruction decoding overhead at simulation time. In the following, we describe the generation of simulators in more detail.

## 4 Simulator generation

It is assumed that the target processor executes one instruction per cycle. Each instruction is a set of one ore more parallel register-transfers (RTs). The instruction-set model extracted from the HDL model consists of *RT templates*, that capture all different RTs the target processor can execute. An RT template specifies the assignment of some value to a destination (register, memory, or I/O port). The value itself is typically an (arithmetic or logic) expression on register contents or constants. For instance, a multiply-accumulate (MAC) operation could be described as:

```
ACCU := ACCU + (X * Y)
```

The expressions are in turn composed of primitive HDL operations (*, +, −, AND, OR, >>, etc.). The RT templates also contain information about argument and destination bit widths.

### 4.1 Primitive operations

While the number of different RT templates that may occur in instruction-set models is infinite, the number of primitive operations is limited. Therefore, a key component in Jacob is a fixed library of simulation functions written in C, each of which simulates one primitive operation. The simulation functions are generic with respect to the target processor word length, which may be different from (and in particular may exceed) the host word length. For each target processor word, a sufficiently long list of 32-bit simulation host words are allocated. The primitive simulation functions traverse the list of subwords and perform the correct actions on these.

If a fixed maximum word length can be assumed for all target processors to be simulated, the simulation functions can be replaced by (faster) macros, since in this case also the list length is fixed, so that the required number of simulation host words can be statically allocated.

### 4.2 Register transfers

When reading the instruction-set model, Jacob generates a C macro for each RT template, that consists of calls to the appropriate simulation functions. In case of the above MAC operation, for instance, the simulation macro would be

```
#define MAC Mult(X,Y,z); Add(ACCU,z,ACCU)
```

where `Mult` and `Add` are the generic simulation functions for "*" and "+", respectively, and "z" is an auxiliary variable.

All macros for RT templates are stored in a C header file, which is included when compiling the generated simulation program. The macro file needs to be generated only once for each target processor model. This speeds up the generation of the simulation program, which mainly consist of macro instances.

### 4.3 Instructions

The machine program to be simulated is given as a list of instructions. When reading the machine program, each instruction is decoded, i.e., the set of RTs to be simulated in the current instruction are determined. Depending on the selected simulation mode (interpretive or compiled), decoding takes place at simulation time or simulator generation time. Since the parallel RTs must be simulated by a sequential program, first a correct sequential simulation schedule has to be found, which does not violate possible read/write dependency constraints between RTs executed in the same instruction. The schedule is determined by topological sorting of the parallel RTs according to the dependencies. If a valid schedule does not exist due to cyclic dependencies, auxiliary variables are inserted that temporarily store register contents, so as to break the cycles.

### 4.4 Interpretive simulation

The generated C simulation program in both interpretive and compiled mode is a loop that terminates after a given amount of instruction cycles. In interpretive mode, each iteration of the main loop first determines a valid sequential simulation schedule for all RTs in the current instruction (i.e., the instruction currently pointed to by the program counter), and then iteratively passes control to the appropriate RT simulation macros until all RTs in the schedule have been processed. The general scheme looks as follows:

```
while (simulation not finished)
{ currentInstr = programMemory[PC];
  sch = DecodeAndSchedule(currentInstr);
  for (all RTs r in sch)
  { switch(r)
    { ...
      case "multiply_accumulate": MAC; break;
      ...
      case "pc_increment": INCR_PC; break;
      ...
    }
  }
}
```

Note, that the modification of the program counter `PC` (increment or jump) is also simulated by a special macro in each instruction, so that after termination of the for loop the correct next instruction is fetched and simulated.

### 4.5 Compiled simulation

Similar to the above case, the C simulation program in compiled mode also consists of a while loop and a switch statement. However, the switch statement directly selects simulation code based on the current program counter contents. Each case label tags a sequence of RT simulation macros, i.e., the simulation schedule for the current instruction. Suppose, an instruction executing a MAC, an address register increment, and a PC increment in parallel is located at program address 0x0005. Then, the corresponding fragment of the simulation program would look as follows:

```
while (simulation not finished)
{ switch(PC)
  { ...
    case 0x0005: MAC; INCR_AR;
                 INCR_PC; break;
    ...
  }
}
```

Besides the pure simulation code, some additional code is inserted, which counts the simulation cycles and checks for breakpoints. Both in interpretive and compiled mode, the generated simulation programs are compiled onto the simulation host with a regular C compiler.

## 5 Results

The current version of the JACOB system is implemented in C++ on a UNIX workstation environment. The generated simulation programs are linked to a common graphical user interface, which supports usual simulator functions, e.g. stepping through a machine program, setting breakpoints, or editing register contents. JACOB has been applied to different ASIPs and a Texas Instruments TMS320C25 standard DSP. For the 'C25, we have simulated several DSP programs, such as digital filters, from the DSPStone benchmark suite [12] on a SPARCstation 20 with 256 MB main memory.

In interpretive mode, an average simulation speed of 115K instructions per CPU second has been achieved. In compiled mode, the measured simulation speed was 240K instructions per second. Using macros instead of calls to the primitive simulation functions (see section 4.1) lead to about 300K instructions per second. For ASIPs architectures less complex than the 'C25, a speed of up to 500K instructions per second could be achieved in compiled simulation mode.

## 6 Conclusions

We have presented a retargetable instruction-set simulation system for a class of embedded DSPs. One main aspect of this work is that bit-true simulators are directly generated from behavioral or RT-level HDL processor models. Since HDL models can also be used for processor synthesis and code generation, the amount of different models of the same processor required during the system design process is reduced. The presented system is flexible with respect to different instruction sets and processor word lengths, and it permits to select between interpretive and compiled simulation, dependent on the intended application. Experimental results indicate a comparatively high simulation speed. Furthermore, the system offers a large degree of flexibility which – in combination with a retargetable compiler – supports the fine-tuning of processor architectures towards a certain application program.

Future work will focus on further practical applications of the compiler/simulator tool set, as well as extensions of the supported processor class.

## References

[1] P. Marwedel, G. Goossens (eds.): *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995

[2] C. Liem: *Retargetable Compilers for Embedded Core Processors*, Kluwer Academic Publishers, 1997

[3] V. Zivojnovic, S. Tjiang, H. Meyr: *Compiled Simulation of Programmable DSP Architectures*, IEEE Workshop on VLSI Signal Processing, 1995

[4] V. Zivojnovic, H. Meyr: *Compiled HW/SW Co-Simulation*, Design Automation Conference (DAC), 1996

[5] P. Paulin, C. Liem, T. May, S. Sutarwala: *FlexWare: A Flexible Firmware Development Environment*, in [1], 1995

[6] A. Fauth: *Beyond Tool-Specific Machine Descriptions*, in [1], 1995

[7] W. Geurts, D. Lanneer, G. Goossens, et al.: *Design of DSP Systems with CHESS/CHECKERS*, Handouts of the 2nd Int. Workshop on Code Generation for Embedded Processors, Leuven/Belgium, 1996

[8] M. Hartoog, J. Rowson, P. Reddy, et al.: *Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign*, Design Automation Conference (DAC), 1997

[9] S. Bashford, U. Bieker, et al.: *The MIMOLA Language V4.1*, Technical Report, University of Dortmund, Dept. of Computer Science, September 1994

[10] R. Leupers: *Retargetable Code Generation for Digital Signal Processors*, Kluwer Academic Publishers, 1997

[11] R. Leupers, P. Marwedel: *Retargetable Generation of Code Selectors from HDL Processor Models*, European Design & Test Conference (ED & TC), 1997

[12] V. Zivojnovic, J.M. Velarde, C. Schläger, H. Meyr: *DSPStone – A DSP-oriented Benchmarking Methodology*, Int. Conf. on Signal Processing Applications and Technology (ICSPAT), 1994