A Generic Tool Set for Application Specific Processor Architectures *

Frank Engel, Johannes Nührenberg, Gerhard P. Fettweis
Mannesmann Mobilfunk Chair for Mobile Communication Systems
Dresden University of Technology, 01062 Dresden, Germany
{engel, fettweis}@in.tu-dresden.de

ABSTRACT
Retargetability allows an easy adoption of a simulator on different processor architectures without a time consuming redesign of all tools. This is evident for an efficient HW/SW codeign.

In this paper we describe a tool set for fast and easy simulation of processor architectures based on a retargetable simulator core. This approach helps to reduce the development time for designing and validating System-on-a-chip (SoC) applications based on a processor core. The use of ANSI C avoids an expensive development of a modeling language.

Our main focus in this paper is on conceptual decisions we made and on the structure of the tool set.

1. INTRODUCTION
The progress in chip technology allows designers to create more and more complex designs known as System-on-a-chip (SoC). This means, systems like printed circuit boards so far assembled of discrete units (e.g. RS232 interface, microcontroller, etc.) are now integrated on one silicon die. Synthesizable models of this units are required. Often they are configurable to allow a customization onto the application needs. Especially in the case of a processor core, modifying the instruction set leads to lower power consumption and smaller die size. These are major aspects in system design. Thus, the popularity of application-specific processors has raised constantly in recent years.

Ways of modification are the customization of instruction set and architecture of existing standard cores [6] or the design of a new processor based on a modular instruction set architecture and customizable data paths [4].

Currently a lot of powerful tools like HDL simulators, silicon compilers or physical layout generators are available. But all of these tools address the HW related part of the design flow. The research for an automatic development of SW tools like an assembler, a simulation model or even a simulation environment is still at the beginning. Hence, this part becomes more and more the critical path of the framework.

In this paper we present a fast and retargetable simulator for processor architectures. Together with an also retargetable assembler one can easily customize an existing processor toward system needs or design a new core.

In our example, the target processor for concept validation was the In/Out-Processor (IOP) of the M3-DSP chipset [12]. This processor supports the DSP core by managing data transfer, data pre- and post-processing and interrupt handling.

The paper is structured as follows: In Section 2 we describe simulator concepts and explain basic decisions. The next section deals with a deeper description of main features. Section 4 resumes all parts to a design flow. Finally results are presented and an outlook on future work is given.

2. SIMULATOR CONCEPTS
A lot of Simulators or even emulators from different vendors are available for a specific processor. But they focus on fixed processors ready to use on a printed board. As depicted above, in the field of SoC customizable HW models are required. Thus, even the simulators need to be customizable. Unfortunately the number of research projects dealing with retargetable simulators is limited. They all use a special developed modeling language.

One example is MIMOLA developed at Dortmund University of Technology [1]. MIMOLA is a special language to describe the structure of a processor. It is similar to common hardware description languages (HDL) like VHDL or Verilog but on a higher level of abstraction. The advantage is an easy generation of a synthesizable HDL description. Drawback is the slow simulation speed similar to HDL simulators. To achieve fast simulations an extraction of a behavioral model is necessary. Unfortunately this extraction is restricted to a small class of fixed arithmetic DSPs.

Another example is Sim-nML of IIT, a language for behavioral description of a processor [3]. This is an extension of nML of Target Compiler Technologies [11]. The language

*This project is sponsored in part by the German NFS (DFG) under contract no. SFB358/TP6.
Figure 1: Types of Processor Models

allows the simulation of the instruction-pipeline. Although Sim-ML provides a compact description of a processor, the simulation speed is still limited [10].

The language LISA from Aachen University of Technology allows the modeling of the instruction set by dividing it into cycles [8]. The focus is on an exact pipeline representation. Thus, a cycle-accurate simulation is implemented.

An easier and more efficient way we decided for is the use of an existing language like ANSI C. There are a wide range of tools like compilers, linkers, debuggers for all platforms available. Furthermore it is a widely spread and well accepted programming language. We adapted it by defining a sufficient set of keywords for describing the instruction set and the processor behavior.

A simulator concept strongly depends on the required accuracy of the processor model. There are two major groups:

1. **HW oriented models** are used for designing structural models. The modeling of physical characteristics like gate-delays is required. Based on these descriptions synthesis tools generate gate-level structures for fabrication.

2. **SW oriented models** are needed for designing application software. Three subgroups can be defined:
   (a) *Statement level accuracy* is needed for testing influences on the algorithms. The focus is on problems like arithmetic overflow etc.
   (b) *Instruction level accuracy* is used for simulating the processor instructions to obtain benchmarks.
   (c) *Cycle level accuracy* is used for investigating correct timing. Often these models are used as a golden reference to validate the HW model.

As depicted in Figure 1 an increasing amount of modeling accuracy results in a decrease of simulation speed [14].

Hence, a tradeoff between maximum performance and the lowest acceptable accuracy level is required. A way to improve simulation speed is the use of compiled instead of interpreted simulation. Since the program for the target processor is a fixed sequence of instructions, processor actions like fetch and decode can be executed before the simulation run. They are translated into a sequence of simulator statements. This is possible as long as the program running on the target processor is not self-modifying. But since the majority of embedded applications run firmware stored in a ROM or an EPROM this type of programs can be neglected. An exception are data dependent instructions like conditional jumps. No preprocessing is possible. During simulation a jump to the succeeding instruction sequence is necessary.

For proof of concept our first target architecture was the IOP. As part of the M3D-DSP chip set this processor is responsible for pre- and post-processing of data. The DSP-core itself is a highly parallel SIMD architecture which needs aligned data sets for full use of its processing power [12]. Thus one task of the IOP is the mapping of data streams onto aligned data sets for effective signal processing. Additionally this processor is responsible for scheduling of this data stream manipulation together with asynchronous events like interrupts in a multitasking manner. Hence to simulate correct response times a cycle accurate model with full control of the pipeline behavior is required. Thus our simulator classifies to the cycle-level-accurate group in Figure 1.

To ensure platform independence we decided for Java and ANSI C as the programming and simulation languages.

3. DETAILS

3.1 Compiled Simulation

The preprocessing of a compiled simulation is divided into the following steps (Figure 2):

1. **Mapping of instructions** and its parameters onto the corresponding simulator statements. This is equivalent to the decoding step within a processor. But instead of repetitive decoding each time the instruction is fetched, it is only done once. This is comparable to a reorganization of the instruction memory.

2. **Building a sequence of simulator statements.** This is possible since the common way of executing a program is the incrementing of the program counter. Thus the execution order of the instructions is known, except for conditional jumps.

3. **Dividing into pipeline steps and merging** with steps of other statements follows. This ensures the capability of simulating pipeline behavior.

3.2 Mapping of Concurrent Functionality

Besides the instruction decoder and the data path a processor contains separate units like timer, interrupt logic etc. All these modules are running in parallel from the instruction execution. Hence a mechanism to integrate them into simulation is needed. We introduced pre-cycle and post-cycle statements. These statements serve as an interface. Parameters are the status of the processor and the resources (re-
Figure 2: Steps of Compiled Simulation

The PΔ-File is divided into the following main categories:

1. **Init**: This section defines global values necessary for simulation.
2. **Helpers**: Here general C code can be directly included. This is useful for defining multiple used C routines and macros to describe the behavior of instructions.
3. **Resources**: This part defines all processor resources. The use of ‘unsigned’ C types is recommended to ensure a correct representation of bit vectors.
4. **Intercycle**: The concurrent functionalities are introduced here. Since they are directly linked into the simulator base they need to be implemented in ANSI C.
5. **Instruction_set**: The definition of instructions follows. This section is divided into the following sub-sections:
   (a) **Instruction**: This represents the mnemonic name of the instruction.
   (b) **Syntax**: The next line describes a disassembly string used in the debugger.
   (c) **Coding**: This section defines the semantic of each bit in the byte-code. Values in square brackets are fixed. Bits represented by the same numeral are grouped into one parameter. Furthermore this number denotes the parameter order in the assembler mnemonic. E.g. 2222 in Figure 3 denotes the second parameter by a 4 bit value. Up to 9 parameters can be defined.
   (d) **Behavior**: This part describes the behavior of an instruction. Here ANSI C code is required. The cycles of the pipeline are divided by the keyword <cycle n>.

Figure 3 gives an example on how to describe the behavior of an instruction.

The processor is divided into two parts:

1. **Architecture independent**: This represents the frame of the simulator. It includes all routines for compiled simulation, handling of concurrent functionality, interfaces to debugger, etc. It is called the simulator base.
2. **Architecture dependent**: This part is derived from the Processor Description File (PΔ-File). A Java parser generates all necessary C routines from this information.

The change table is also used to observe multiple manipulations of variables (e.g. the content of a register). A warning message is generated if more than one entry for updating the same variable is found. This feature is helpful during the development of a new processor architecture.

3.3 Processor Description

To achieve retargetability the simulator is divided into two parts:

1. **Architecture independent**: This represents the frame of the simulator. It includes all routines for compiled simulation, handling of concurrent functionality, interfaces to debugger, etc. It is called the simulator base.
2. **Architecture dependent**: This part is derived from the Processor Description File (PΔ-File). A Java parser generates all necessary C routines from this information.
4. SIMULATION FLOW

In a second step the architecture dependent part of the simulator is generated from the PD-File. After that, according to Section 3.1, the preprocessing step of the compiled simulation is executed. It uses the byte-code produced by the assembler.

Now a standard C compiler is used to compile and link all three parts (simulator base, precompiled program and architecture dependent C routines) to an executable representing the simulator ‘SimuC’.

Currently the free C debugger ‘DDD’ is used as a graphical frontend during simulation [13].

5. RESULTS

The tool set was developed on SUN workstations using the GNU compiler ‘gcc’, Java and the Java parser ‘javacc’. Simulation speed up to 300,000 instructions per second are achieved. The performance is measured using different types of programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Instructions per Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>201,000</td>
</tr>
<tr>
<td>OFDM</td>
<td>255,000</td>
</tr>
<tr>
<td>Transport</td>
<td>300,000</td>
</tr>
</tbody>
</table>

Table 1: Speed Results

As depicted in Table 1 the lowest speed is achieved with programs only containing arithmetic and logical operations. Since the IOP can operate on variable bits within a data element this value results from cost intensive C modeling of bit operations. A better benchmark is the OFDM-based Hiperlan wireless ATM modem, the target application of the M3-DSP chipset [5]. This program represents a profile of arithmetic/logical, transport and control instructions the IOP is designed for. With a benchmark mostly consisting of transport instructions up to 300,000 instructions per second are achieved. This is comparable to results of other retargetable simulators (240,000 .. 300,000 [7], 160,000 [9]).
6. SUMMARY AND FUTURE WORK
In this paper we presented an easy approach for a retargetable processor simulator together with an assembler. Instead of designing a new modeling language to describe the processor architecture we choose ANSI C and Java. The advantage is a great variety of reliable tools like a compiler and the wide spread acceptance of these languages. The processor modeling is controlled by a set of keywords and constructs linked with self written C code (see Section 3.3). The user can still fully explore the capacity of ANSI C. Java is used for parsing the target program and the processor description.

Further investigations will deal with the design of a graphical front-end to replace the currently used DDD front-end. Also required is an interface for co-simulation with other tools like VHDL simulators or evaluation boards to integrate it into a complete system modeling environment [2].

7. REFERENCES