In the Proceedings of ICSPAT'95, Boston, USA, pp. 1748 –1763

SUPERSIM — A NEW TECHNIQUE FOR SIMULATION OF PROGRAMMABLE DSP ARCHITECTURES

V. Živojnović, S. Pees, Ch. Schläger, R. Weber, and H. Meyr

Integrated Systems for Signal Processing
Aachen University of Technology
Templergraben 55, 52056-Aachen, Germany

ABSTRACT

This paper presents a technique for simulating DSP processors based on the principle of compiled simulation. Unlike existing, commercially available instruction set simulators for DSP processors, which are of interpretive character, the proposed technique performs instruction decoding and simulation scheduling at compile time. The technique offers up to three orders of magnitude faster simulation. The high speed allows the user to explore algorithms and hardware/software trade-offs before any hardware implementation. Moreover, the user can tailor the compiled simulation to trade speed for more accuracy. In this paper, the sources of the immense speedup are analyzed and the realization of the simulation compiler is presented.

I. Introduction

Designers of today's DSP systems — such as digital cellular phones and multimedia systems — face rising complexity and quickly changing system requirements. To deal with these challenges, designers have increasingly turned to programmable architectures as the basis for their DSP systems. The flexibility inherent in such architectures allows designers to accommodate late design changes and to fix bugs even after shipping the systems. Moreover, the flexibility promises shorter design time, thus decreasing time-to-market.

In the past, programmable processors played a peripheral role: they performed control and interface functions while ASICs provided key DSP functions. Today, with the advent of much higher performance digital signal processors, they play a much larger role, providing key DSP functions as well as control and interfacing, and relegating ASICs to the role of accelerators. Correspondingly, the amount and complexity of DSP code have increased introducing the need for new powerful software development tools.

Instruction set architecture (ISA) simulators of the processor, such as instruction set simulators are standard part of tool-sets supplied with off-the-shelf or in-house DSP processor. They enable comfortable debugging through controlled program execution and provide visibility of processor resources necessary for code development. All currently available instruction set simulators use the interpretive simulation technique. Their main disadvantage is the low simulation speed. In the sequel we shall see that this represents a serious limitation for use of ISA simulators in DSP code design and verification.

This paper presents a new approach to ISA modeling that is fast enough to enable code verification and statistical exploration of algorithms without any hardware components. Instead of the standard interpretive technique, it uses the technique of compiled simulation. We have observed three orders of magnitude improvement in speed over standard interpretive simulators on some examples. Typical speedups are between one and two orders of magnitude. The new techniques is able to close the speed-gap between simulation and real-time execution on the processor to only one order of magnitude.

Because of its ability to be tailored to each new simulation, our compiled simulator — *SuperSim* — can be used in different kinds of simulation tasks. We can write behavioral hardware models in C and attach them using subroutine calls to the compiled simulator (i.e. behavioral simulation). We can embed our simulator within system-level simulation environments such as *COSSAP*, *DSPstation*, *Ptolemy*, or *SPW*. This combination lets us verify algorithms using the exact arithmetic properties of the target processor (i.e. bit-true simulation). We can have the simulator compiler insert code that mimic instruction pipelines. Thus we can estimate the number of clock cycles that the software will require between real-time events (i.e. clock-true simulation). We can connect the compiled simulator to a VHDL or VERILOG simulator to verify hardware that interfaces directly to simulated-processor pins. The simulator compiler achieves this by inserting code that communicates pin updates to the hardware simulator (i.e. pin-accurate simulation).

Compiled simulation does not limit debugging. Our approach provides links from the compiled code to the original source program for the simulated-processor code. The links allow the use of a standard C-language debugger (such as `dbx`) on the target program. Thus, designers can debug the code just as they would with a vendor-supplied interpretive simulator.

The compiled simulation technique we use for our simulator is well known in simulation of hardware circuits, e.g. [1]. We follow the same general idea, but apply it to the simulation of the instruction set architecture. Our approach resembles binary translation used for migrating executables from one machine to another [2], or collecting run-time statistics [3]. However, clock/bit-true translation and debugging are not objectives of binary translation.

II. Fast ISA Simulation

The following three examples shall provide an insight into the role of fast ISA simulation in DSP code design, joint DSP processor-compiler design, as well as in the general *model-before-silicon* approach.

DSP Code Verification

On Fig. 1 a typical DSP software design flow is depicted. Although algorithm design and assembly code programming are probably the most complex tasks, verification is surely the most time-consuming. According to some estimates, almost 70% of the overall design time is spent on verification.

Verification of the code can be done either using a hardware or a software model of the processor. Hardware models in form of emulators, or final target system, permit fast verification, but offer only poor debugging support. For example, if debugging is done with the processor itself, pipelining effects cannot be followed. Also, the realization of the model is costly in terms of time and money, and flexibility of the model for user's interventions low.

Software models of the processor provide optimum debugging comfort. Also, model realization

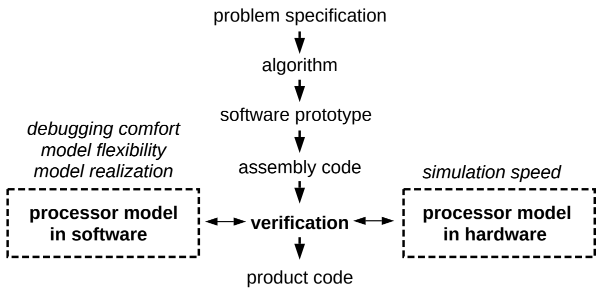


Figure 1: Code Verification.

consists in writing software which is less costly than building hardware. In the same time the flexibility of the model is very high — re-designs and extensions are done by simple re-coding of the modeling program. Using a software model of the processor can significantly shorten time-to-market of the product. The main obstacle is the relatively low simulation speed. Fig. 2 presents simulation speed as a function of the accuracy level for different processor models. Depending on the necessary accuracy level, processor simulators can simulate between 1 instruction and 100M instructions per second on a 100 MIPS machine. The ISA simulation level includes all the details of the processor which are of interest to the DSP software developer. Currently, ISA models can simulate between 2K and 20K insns/s [4]. This is mostly not fast enough for code

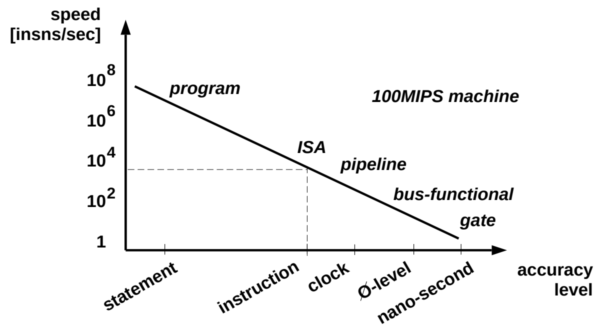


Figure 2: Modeling Accuracy vs. Simulation Speed.

verification. The following example gives an illustration. Our example arises from the development of the ADPCM G.721 and G.726 speech transcoders for the Digital European Cordless Telecommunications (DECT) and Digital Circuit Multiplication Equipment (DCME) applications using an off-the-shelf DSP processor. Off-line verification of the software implementation (~93 millions instructions) on the standard set of CCITT-ITU test sequences (13 seconds of speech signals) on the target processor takes 7 seconds. The same verification using the instruction set simulator (4K insns/s) provided by the DSP chip vendor takes approximately 6.4 hours on an 86 MIPS machine (Sparc-10).

Unfortunately, a complete statistical analysis of the algorithm required to estimate signal-to-noise ratio (SNR), bit-error-rate (BER), or word-length dependent accuracy would require far more than 13 seconds of real-time signals. When designs have to be verified using statistical methods, state-of-the-art processor simulators prove inadequate. The designer has to use a prototype or hardware emulation. Both prospects increase costs and development time enormously.

Joint Processor-Compiler Design

DSP processor and compiler have to be designed in a joint fashion if optimum results are expected. Figure 3 presents the design flow in which the processor and the compiler are simultaneously tuned in order to improve quantitative performance data measured on a set of typical DSP benchmarks. In this case

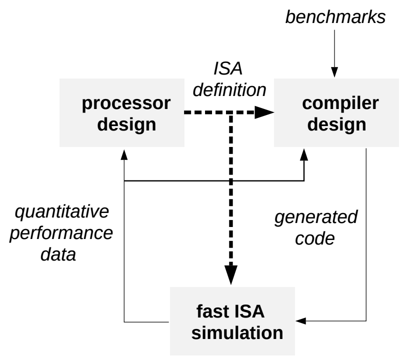


Figure 3: Joint DSP Processor-Compiler Design.

fast ISA model plays a crucial role. Moreover, it cannot be replaced by a hardware model, because the processor is still in design. If the ISA simulation is slow, most of the time of the design cycle is spent on waiting for quantitative performance data.

Model-Before-Silicon

In order to shorten time from ISA specification to final product release, DSP hardware and software suppliers are increasingly adopting the *model-before-silicon* approach well known in the field of general purpose hardware and software design. Figure 4 depicts its main idea.

After defining the ISA of the processor, the ISA simulator is built permitting the design of silicon, software tools and application to continue in parallel.

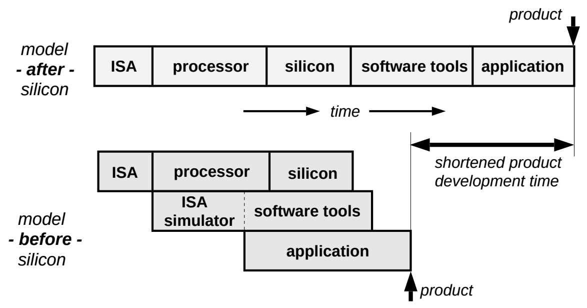


Figure 4: Model-After/Before-Silicon.

As a result, product development time is shortened. ISA simulators are the crucial part of this approach. They have to be accurate and fast enough in order to fulfill the needs of processor, tools and application designers. The standard approach to use VHDL models for ISA simulation mostly failed because of the low simulation speed.

III. Compiled Simulation of Programmable Architectures

Interpretive simulators process instructions using a software model of the target processor. A virtual processor, i.e. ISA model, is built using a data structure representing the state of the processor and a program which changes the processor state according to the stimuli — either a new instruction pointed to by the program sequencer, or some external events, such as interrupts. In general, interpretive simulators can be summarized as a loop in which instructions are fetched, decoded, and executed using a "big switch" statement, such as the one below:

```
       while(run) {
          next = fetch(PC);
          insn = decode(next);
          switch (insn) {
             ...
add:         exe_add(); break;
             ...
          }
       }
```

Our approach translates each target instruction directly to one or more host instructions. For example, if the following three target instructions

```
add r1,r2;
mov r2,mem(0x175);
mul r2,r3;
```

are interpreted, the above simulation loop iterates once for each instruction. The compiled simulation approach translates the target instructions into the following host instructions, represented here as macros:

```
ADD(_R1,_R2); SAT(_R2); ADJ_FL(_R2); PC();
MOV(_R2,MEM(0x175)); ADJ_FL(); PC();
MUL(_R2,_R3); SAT(_R3); ADJ_FL(_R3); PC();
```

The translation completely eliminates the fetch and decode steps, and loop overheads of interpretation, resulting in a faster simulation. For target processors with complex instruction encoding, the decode step can account for a significant amount of time.

Additional speedup is created because compiled simulation generates code tailored to the required accuracy level, while an interpreter provides a fixed level of accuracy. For example, if interrupts are not required, compiled simulation suppresses the simulation of the interrupt logic already during the preprocessing.

However, compiled simulation assumes that the code does not change during run-time. Therefore self-modifying programs will force us to use a hybrid interpretive-compiled scheme. Fortunately, self-modifying programs are rare. The isolated cases in DSP programming we encountered so far are limited to programs that change the target address in branch instructions. This type of self-modifying code, however, can be easily handled without interpreting.

The binary-to-binary translation process can be organized in two ways. The direct approach translates target binary to host binary directly (Fig. 5a). It guarantees fast translation and simulation times, but the translator is more complex and less portable between hosts. To simplify the translator and improve its portability, we split the translation process into two parts—compile the target code to a program written in a high-level language such as C (front-end processing), and then compile the program into host code (back-end processing) (Fig. 5b). In this way we take advantage of existing compilers on the host and we reduce the realization of the simulation compiler to building the front-end. Portability is greatly improved but with a possible loss in simulation speed.

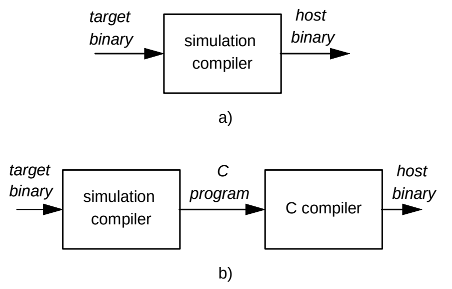


Figure 5: Two Approaches to Binary-To-Binary Translation.

Some features of machine code are difficult to represent in a high-level language. For example, in the absence of very sophisticated analysis, compiled simulation must assume that every instruction can be a target of an indirect branch statement. Therefore, every compiled instruction must have a label, and computed `goto` or `switch` statements are used to simulate indirect branching. These labels reduce the effectiveness of many compiler optimizations. For DSP architectures, indirect branches are the main problem in compiled simulation [5].

IV. Realization of the Simulation Compiler and Results

The simulation environment *SuperSim* SS-21xx has been implemented for the Analog Devices ADSP-21xx family of DSP processors. It consists of the simulation compiler (`ssc`), host C compiler (`gcc`), and C source level debugger (`dbx`). This enables cycle- and bit-true behavioral simulation of the processor in a comfortable debugging environment.

The `ssc` simulation compiler has a form of a two-pass translator with a translation speed of about 1500 target insns/s (Sun-10/64MB). Translating the whole program memory (16 Kinsns) of the ADSP-2105 into intermediate C representation takes less than 11 seconds. To enable additional trade-off between recompilation and execution speed, the simulation compiler can translate target instructions into intermediate C code using macros or function calls.

Compiling the intermediate C code to the host executable takes most of the overall translation time. For the version with function-calls the compilation speed of the `gcc-2.5.8` compiler with optimization `-O1` was about 24 target insns/s (12 target insns/s for -O3).

Table 1 presents some real-life examples of SS-21xx performance. Simulation speed measured in insns/s depends on the complexity of instructions found in the target code. The FIR filter example is generated by the C compiler of the target that generates compound instructions rarely. However, the ADPCM example is hand-coded optimally and uses complex compound instructions frequently. The results from Table 1 show that our simulator outperforms the standard simulator by almost three orders of magnitude on the FIR example and by about 200 times on the ADPCM example. The same verification which took 6.4 hours with the standard

example	simulator	opt.	insns/s	speedup
FIR filter	ADSP-21xx	-	3.9K	1
	SS-21xx	-O3	2.5M	640
	-"-	-O2	2.0M	510
	-"-	-O1	1.6M	420
	TI-C50	-	2.4K	1
	SS-C50†	-O3	0.8M	320
	SS-77016†	-O3	0.8M	-
ADPCM	ADSP-21xx	-	4.0K	1
	SS-21xx	-O3	0.8M	200
	-"-	-O2	0.6M	150
	-"-	-O1	0.4M	100
host: Sun-10/64MB; SS-21xx flags: -f compiler: gcc 2.5.8; †-preliminary results;				

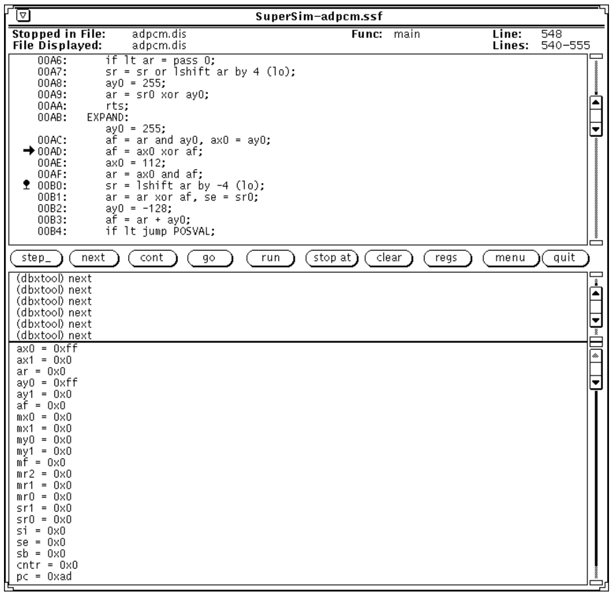
Table 1: Simulation Examples - Measurement Results.

ADSP-21xx simulator is reduced to less than 2 minutes using *SuperSim*.

The ADSP-21xx does not have a visible pipeline. In order to prove our concepts on architectures with pipeline effects we have written compiled simulation examples for the TI's TMS320C50 and NEC's μPD77016 processors. Despite overhead introduced for pipeline modeling, results from Table 1 show that our approach still achieves significant speedup. A detailed analysis of compiled simulation of pipelines is provided in [5].

Moreover, our simulator offers full debugging support using the standard C level debugger (e.g. `dbx` or `gdb`). It offers breakpoint setting and watching of registers, memory, flags, stack and pins. This is a large advantage compared to standard DSP debuggers which are highly target dependent. Figure 6 shows an example of the graphical user interface of the `dbxtool` debugger which was adapted to execute C code of the simulator, and in the same time display assembly instructions of the target or the C code of the simulator. If the attached hardware is modeled in C, the same debugger can be used for debugging of software and hardware.

In order to enable control of external events which occur in parallel with code execution, the simulation compiler is able to generate cycle-hooks, which are executed before or after each clock-cycle. Also, the user can easily insert his own hooks using simple labels in the assembly code of the target.


SuperSim-adpcm.ssf

```
Stopped in File:   adpcm.dis                Func:  main        Line:   548
File Displayed:    adpcm.dis                                   Lines:  540-555
   00A6:      if lt ar = pass 0;
   00A7:      sr = sr or lshift ar by 4 (lo);
   00A8:      ay0 = 255;
   00A9:      ar = sr0 xor ay0;
   00AA:      rts;
   00AB:   EXPAND:
              ay0 = 255;
   00AC:      af = ar and ay0, ax0 = ay0;
 → 00AD:      af = ax0 xor af;
   00AE:      ax0 = 112;
   00AF:      ar = ax0 and af;
   00B0:      sr = lshift ar by -4 (lo);
   00B1:      ar = ar xor af, se = sr0;
   00B2:      ay0 = -128;
   00B3:      af = ar + ay0;
   00B4:      if lt jump POSVAL;
```

```
(dbxtool) next
(dbxtool) next
(dbxtool) next
(dbxtool) next
(dbxtool) next
(dbxtool) next
ax0 = 0xff
ax1 = 0x0
ar = 0x0
ay0 = 0xff
ay1 = 0x0
af = 0x0
mx0 = 0x0
mx1 = 0x0
my0 = 0x0
my1 = 0x0
mf = 0x0
mr2 = 0x0
mr1 = 0x0
mr0 = 0x0
sr1 = 0x0
sr0 = 0x0
si = 0x0
se = 0x0
sb = 0x0
cntr = 0x0
pc = 0xad
```

Figure 6: Debugging with *SuperSim*.

V. HW/SW Co-Simulation

The *SuperSim* simulator can be used in a simulation of both the hardware and the software components. We have connected *SuperSim* to behavioral models of hardware components as shown in Figure 7. This example shows an A/D converter and its glue logic attached to a DSP processor. Communication

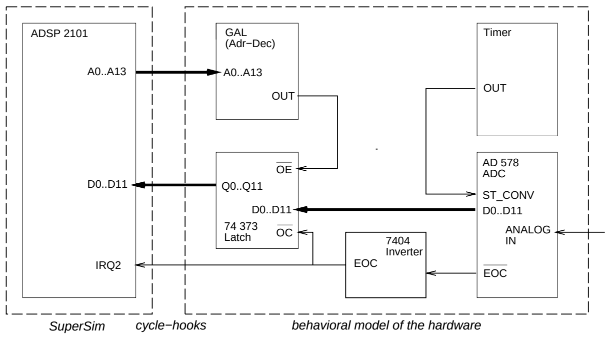


Figure 7: HW/SW Co-Simulation Using SuperSim.

between software and hardware is mediated by cycle hooks. The hooks pass control to the hardware model which is written in C. The hooks also accept data from the hardware models. When the hardware models are written in C, the hooks are simple calls. However, when the models are written in HDL, the hooks are more complicated. They must synchronize *SuperSim* to the HDL simulator and also convert data values before and after communicating with the HDL simulator.

VI. Compiled Simulation and System-Level Design Tools

System-level design tools like *COSSAP* (Synopsys), *DSPstation* (Mentor), *Ptolemy* (Berkeley University) or *SPW* (Alta Group) can easily take advantage of the compiled simulation in order to deliver bit-true simulation using the ISA model of the DSP processor. Figure 8 shows how *SuperSim* can be used to obtain bit-true simulation for one or more functional blocks. This toy example consists of three processing blocks. Whereas for the sink and source blocks the provided C functional templates are used, the functionality of the third block is described by the binary code for some target processor, in this case the ADSP21xx DSP processor from Analog Devices. In order to permit execution without using processor hardware, *SuperSim* converts the binary code into bit-equivalent C code, which is then used as the C functional template for system-level simulation.

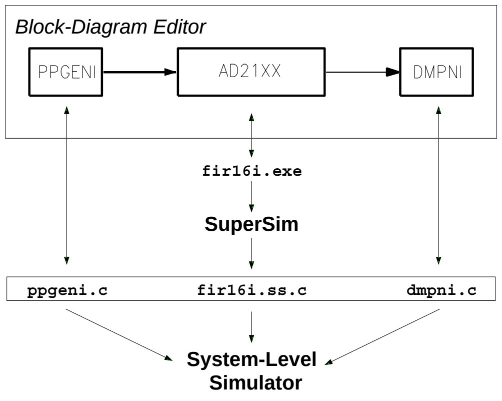


Figure 8: System-Level Simulation with *SuperSim*.

Currently system-level design tools provide the same functionality by interfacing the bit-true model to the interpretive instruction set simulator which runs as a separate process. The disadvantages of this approach are:

- low speed and flexibility of standard ISA simulators;
- multiple instantiations of the ISA simulator are necessary if multiple functional blocks are simulated in a bit-true fashion;
- interfacing the system-level and instruction-set simulator introduces a high overhead;
- different debuggers are used for system-level and instruction set simulation.

All these disadvantages disappear if *SuperSim* is used. *SuperSim* offers to the designer a fast, flexible and comfortable simulation and debugging environment for bit-, cycle-, pipeline- and pin-true simulation of the DSP processor.

VII. Conclusions

Compiled simulation provides very fast and accurate instruction set simulation. The presented *SuperSim* simulation environment generates bit-, cycle-, and pin-accurate DSP processor simulation engines that are two to three orders of magnitude faster than interpretive simulators. Moreover, standard source level debuggers offer a comfortable debugging environment and the intermediate representation in C is open for extensions by the designer. The presented compiled simulator is easily interfaced to behavioral hardware models. In addition to fast simulation, it offers a comfortable debugging environment in which hardware and software are debugged using the same debugger.

Currently, recompilations (with *SuperSim*) after design changes are relatively slow. Though recompilation will always take additional time relative to interpretation, we believe that we can reduce the time to a tolerable level by limiting recompilation only to code that has changed. Moreover, a *SuperSim*-interpreter hybrid, in addition to alleviating the problems of indirect branches, can provide fast simulation speed, as well as fast turn-around time on design changes.

VIII. References

[1] Z. Barzilai, *et al.*, “HSS - A high speed simulator,” *IEEE Trans. on CAD*, vol. CAD-6, pp. 601–616, July 1987. 1987.

[2] R. Sites, *et al.*, “Binary translation,” *Comm. of the ACM*, vol. 36, pp. 69–81, Feb. 1993.

[3] J. Davidson and D. Whalley, “A design environment for addressing architecture and compiler interactions,” *Microprocessors and Microsystems*, vol. 15, pp. 459–472, Nov. 1991.

[4] J. Rowson, “Hardware/Software co-simulation,” in *31st ACM/IEEE Design Automation Conference*, 1994.

[5] V. Živojnović, *et al.*, “Compiled simulation of programmable DSP architectures,” in *Proc. of 1995 IEEE Workshop on VLSI in Signal Processing, Osaka, Japan*, Oct. 1995.