SimMips A MIPS System Simulator

Naoki Fujieda†, Takefumi Miyoshi†‡, and Kenji Kise†‡
†Graduate School of Information Science and Engineering, Tokyo Institute of Technology
‡Japan Science and Technology Agency (JST)
{fujieda, miyo}@arch.cs.titech.ac.jp, kise@cs.titech.ac.jp

Abstract

We have developed SimMips, a simply coded MIPS system simulator written in C++, to meet increasing demands for embedded system education. In this paper, we show the simplicity of SimMips by describing its concept and implementation. And we show the comprehensibility, taking examples of using it as a lecture material. We designed and implemented SimMips considering the hardware organization of the target computer system. We introduce a palm-top hardware embedded system named MieruPC which includes a MIPS-like soft processor based on SimMips, to reveal flexibility of SimMips.

1 Introduction

Lots of processor simulators are used as tools of processor education and research [4, 5]. Similarly, system simulators are often used, which simulate the whole of the target computer system, including not only processors but also memory controllers and I/O controllers. For educational use, key points of such simulators are simplicity, comprehensibility, and flexibility. Learners can grasp the target systems more experientially by reading and modifying their source codes.

QEMU [2], M5 [3], and Simics [9] are well-known system simulators. Their goals are dealing with various platforms and high-speed simulation, rather than simple and comprehensive source codes. Bochs [7] is famous single-platform system simulator which emulates x86 systems. Though x86 is one of the most popular architecture in commercial processors, it is too complicated for educational use. On the other hand, MIPS architecture has a straightforward instruction set and is often used as a material of computer architecture lectures [12]. So a simple MIPS system simulator is very useful.

SPIM [6] is one of the most common MIPS processor simulators. It interprets and runs assembly language directly, so it is a merit that users do not need to build a cross development environment. But now this merit gets smaller, because building the environment is not so hard work as in the past by using tools like Buildroot [1]. Instead, the demerit of not being able to accept compiled binary files is relatively growing.

We have developed a system simulator named SimMips, whose target computer system includes a MIPS32 ISA processor, as a practical simulator for embedded system education and research. SimMips is implemented simply with 4,500 lines in C++, and Linux runs on it without modifications. Though there is a tradeoff between readability and simulation speed, processor speedup now enable simulators to satisfy high readability and sufficient speed at the same time. So our primary design policy is to keep the source code simple and comprehensive.

We are also developing MieruPC, a palm-top embedded system including a MIPS-like soft processor named MipsCore, which is written in Verilog HDL and based on SimMips. High readability and flexibility enable such an application to be quickly built.

The rest of this paper is as follows. Section 2 describes the concept and the implementation of SimMips. Section 3 clarifies the effectiveness of SimMips by measuring simulation speed and taking a case study. Section 4 contains the development of MipsCore and MieruPC. Section 5 concludes this paper.

2 Concept and implementation

SimMips is implemented in C++ and the lines of code is 4,538, which is very small as a system simulator. The number of lines and summary of each file are summarized in Table 1. Our policies on simulators are high readability and hardware-aware design. We say the word hardware-aware to mean keeping the software structure close to the hardware structure of the target computer system.

---

1The word mieru stands for visible in Japanese.
Table 1. The file organization of SimMips Version 0.5.5.

<table>
<thead>
<tr>
<th>filename</th>
<th>lines</th>
<th>summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>define.h</td>
<td>747</td>
<td>definition</td>
</tr>
<tr>
<td>main.cc</td>
<td>21</td>
<td>main function</td>
</tr>
<tr>
<td>board.cc</td>
<td>622</td>
<td>simulation environment</td>
</tr>
<tr>
<td>memory.cc</td>
<td>297</td>
<td>main memory and controller</td>
</tr>
<tr>
<td>simloader.cc</td>
<td>227</td>
<td>ELF program loader</td>
</tr>
<tr>
<td>mips.cc</td>
<td>907</td>
<td>MIPS computation core</td>
</tr>
<tr>
<td>mipsinst.cc</td>
<td>767</td>
<td>attributes of MIPS instructions</td>
</tr>
<tr>
<td>cp0.cc</td>
<td>309</td>
<td>MIPS processor control(CP0)</td>
</tr>
<tr>
<td>device.cc</td>
<td>641</td>
<td>I/O controllers</td>
</tr>
<tr>
<td>- Total -</td>
<td>4,538</td>
<td></td>
</tr>
</tbody>
</table>

2.1 OS-Mode and App-Mode

SimMips reads and simulates an ELF executable file. It offers two modes, OS-Mode and App-Mode. In the OS-Mode, SimMips reads an OS kernel as an ELF executable and initializes simulation environment along machine setting files. The input in the App-Mode is a statically-linked user program. Through these modes, SimMips is used not only as a system simulator, but also as a simple processor simulator.

In addition to standard output of an application or an OS, the simulator optionally outputs the contents of the memory and the registers, the trace of the executed instructions, and the statistics of the execution.

2.2 Software Architecture

Hardware-awareness is our major design concept. Such design makes it easy for learners to understand simulator’s behavior. The hardware organization of the target system is shown in Figure 1. Deeply related elements are connected with an arrow. We call the whole system “Board”. The Board consists of a “Chip”, a main memory, an off-chip interrupt controller (IntController), and a serial I/O controller (SerialIO). The Chip includes a MIPS Processor, a processor control coprocessor (MipsCp0), and a memory interface (MemoryInterface). A serial console is used for both input and output.

The relations among the major objects of SimMips are described in Figure 2. The hierarchic structure of the target system is maintained. A solid arrow indicates that an object creates an instance (instances) of another class. A dotted arrow means reference. For example, an object of the Mips class is created by the Chip class, and refers to a MipsCp0 and a MemoryController. Objects indicated by gray boxes are not shown in Figure 1. These are additional objects to make the readability better or to offer particular functions to the simulator. In addition, SimLoader is an ELF program loader used for initialization. While all the units work in the OS-Mode, the App-Mode invalidates a MipsCp0, a IntController, and a SerialIO.

2.3 Implementation of computation core

The Mips class implements a MIPS computation core. Normally, the execution model is function-level or instruction-level, that is, SimMips executes one instruction in a single simulator cycle. This class interprets almost all the MIPS32 instructions, except floating point instructions. Objects of the MipsArchstate class, the MipsSimstate class, and the MipsInst class are generated by the Mips class. The MipsArchstate keeps architectural states such as contents of the registers. The MipsSimstate records the statistics like
1 void Mips::execute()
2 {
3     switch (inst->op) {
4         case ADDU_____:
5             rrd = rrs + rrt;
6             break;
7         case BEQ______:
8             npc = inst->pc +
9                 (exts32(inst->imm, 16) << 2) + 4;
10            cond = (rrs == rrt);
11             break;
12         case SYSCALL__:
13             if (cp0)
14                 exception(EXC_SYSCALL);
15             else
16                 syscall();
17             break;
18         case ...
19     }
20 }

Figure 3. A part of the execute method.

declares the number of instructions simulated. The MipsInst contains decode, mnemonic, and other information related to MIPS instruction. To separately implement these classes provides better readability.

Like SimCore [5] and SimCell [14], our previous processor simulators, SimMips adopts a folded description style. This style means that the operation of each instruction is divided into some stages and described gradually like a pipeline structure. SimMips has 8 stages: fetch, decode, register fetch, execute, 2 memory access stages (send a request and receive the result), and 2 write back stages (for the registers and for the program counter). Each stage is implemented in a single method. Thus, the Mips class has fetch(), decode(), regfetch(), execute(), memsendt(), memreceive(), writeback(), and setnpc() methods, corresponding to each stage.

The following is the more detailed outline of the Mips class by taking up some actual codes. A part of execute() method, corresponding to the execute stage of MIPS processors, is shown in Figure 3.

In execute() method, the value of the destination register, the target of the branch instruction, or the memory address to be accessed, is calculated according to the instruction operation and the values of the source registers gotten in the previous stages. For example, from the 4th through the 6th line describes the addu (ADD Unsigned) instruction that writes the sum of two registers rs and rt to the register rd. The variables rrs, rrt, rrd stand for the value of registers rs, rt, rd, respectively. The beq (Branch Equal) instruction, shown in the block from the 7th to the 11th line, branches to the address specified relatively by the immediate (imm) field of the instruction, only if the two registers rs and rt are equivalent.

1 void Mips::syscall()
2 {
3     switch (as->r[REG_V0]) {
4         case SYS_EXIT:
5             state = CPU_STOP;
6             break;
7         case SYS_WRITE:
8             if (as->r[REG_A0] == STDOUT_FILENO) {
9                 for (uint i = 0; i < as->r[REG_A2]; i++) {
10                     int mcid = mc->enqueue(as->r[REG_A1] + i,
11                                              sizeof(char), NULL);
12                     if (mcid < 0) break;
13                     mc->step();
14                     if (mc->inst[mcid].state == MCI_FINISH)
15                         putchar((int) mc->inst[mcid].data008);
16                 }
17             }
18             as->r[REG_V0] = as->r[REG_A2];
19             as->r[REG_A3] = 0;
20             break;
21         case ...
22     }
23 }

Figure 4. A part of the syscall method, which emulates the behavior of system call in App-Mode.

As indicated by the block from 12th through the 17th line, the implementation of the syscall (SYStem CALL) instruction varies between the OS-Mode and the App-Mode. In the OS-Mode, the exception() method in the 14th line is called and an exception is raised there. Then the program counter becomes the address of the system call handler on the target OS.

In App-Mode, the syscall() method briefed in Figure 4 is called. Since no system call handler are prepared in this mode, the syscall() method emulates the behavior of system calls in App-Mode. For example, exit system call halts the core and finishes simulation in the 5th line. The process of the write system call is described from the 7th to the 19th line. If the target is the standard output (the 8th line), the simulator reads a specified number of character from the MainMemory through the MemoryController, and write them to the standard output of SimMips (from the 9th to the 16th line). Finally the results of the system call are set (the 18th and 19th line). We have implemented limited kinds of system call so far, but we can deal with comprehensive system calls by adding description to the syscall() method.

2.4 Implementation of OS-Mode

This section describes the implementation of the characteristic part of SimMips as a system simulator. We first developed the App-Mode alone, that is, a simple proces-
sor simulator including the computation core and the main memory. Then we added the OS-Mode-specific part.

The necessary functions to realize a system simulator are dealing with exception, TLB management, address translation, I/O control, and so on. The following is the explanation of 4 classes taking important roles in OS-Mode.

**MipsCp0** implements a MIPS processor control or a CP0 (Coprocessor 0). The CP0 has control registers and TLB. It manages exceptions, TLB, and address translation. It also has an internal counter. When this counter reaches a specific value, the CP0 raises a timer interrupt.

**MemoryController** implements a memory controller. The computation core loads and stores through it. It has a memory map initialized by machine setting files. Depending on the memory map and the physical address, a proper memory-mapped unit is selected and accessed.

**IntController** provides the function of interrupt controllers. It simulates two 8259-like controllers. It integrates interrupt requests and sends to the CP0.

**SerialIO** emulates a serial I/O controller. The standard input of SimMips is polled at fixed intervals and stored to the input buffer of the serial console. Similarly, the output of the serial console is written to the standard output. When the input buffer is not empty and interrupts are enabled, the controller raises an interrupt.

Coordination among the classes described above and the Mips class realizes the functions of OS-Mode. As an example, we explain the response to an interrupt caused by serial input. The SerialIO detects input and send an interrupt through the IntController to the MipsCp0. When the MipsCp0 detects the interrupt, the Mips cancels normal execution. The MipsCp0 writes the information of the interrupt to its own control registers and sets the privilege bit. Lastly the Mips resume execution from the address of the interrupt handler.

To keep the source code simple, all classes having memory-mapped I/O and accessed through the MemoryController implement an interface MMDevice (Memory Mapped Device).

### 3 Effectiveness of SimMips

#### 3.1 Platforms

SimMips operates on many platforms. The platforms we verified the operation are as follows:

- Intel Xeon, Red Hat Enterprise 5.2, GCC 4.1.2
- Intel Xeon, Cygwin 1.5.25, GCC 3.4.4
- AMD Opteron, CentOS 4.7, GCC 3.4.6
- ARM Cortex-A8, Ubuntu 9.04, GCC 4.3.3

SimMips does not work on big-endian platforms like Cell/B.E.

#### 3.2 Evaluation of simulation

This section describes the way we verify the correctness of simulation and shows the simulation speed of SimMips. The data in this section is measured using a server with two Xeon X5365 (3 GHz, Quad-core, 4 MB L2) processors and 16 GB main memory running Red Hat Enterprise Linux 5.2.

Since SimMips is a function-level simulator, we checked that the register file and the program counter are correct, instruction by instruction. In the App-Mode, we collected and compared logs of GNU Debugger (GDB) and SimMips by printing all the registers after every stepwise execution. Programs used for this validation include a Hello World program, a quick sort, and an N-queens problem. The number of instructions for them ranges from tens of thousands to millions.

In the OS-Mode, determinacy is lost due to timing of interrupt raised. So we similarly verified about 3 million instructions with existent simulators, by the first interrupt. After that we tested the behavior of simulation and saw a...
Linux kernel booted correctly. A part of the output of booting Linux on SimMips is shown in Figure 5. The lines starting with `##` are messages of the simulator. About 400 million instructions are simulated in a minute. Some mistakes caused kernel panic or infinite loop during debug process, so the system simulation works without critical bugs.

The instruction mix of booting Linux (about 400 million instructions) and that of quick sort program (about 17.7 million) is summarized in Figure 6. Arithmetic and compare instructions account for a large part of quick sort, but logical instructions are rarely used. On the other hand, booting OS includes many logical and load instructions. It is one of the merits of SimMips that we can get statistics of instructions not only on the user level but on the privilege level like this.

As mentioned, booting Linux on SimMips only takes a minute. So the simulation speed is practical enough. The following is a detailed evaluation of simulation speed. The quick sort program is used for a benchmark, so SimMips is executed in the App-Mode. The simulator is compiled by two compilers, GCC 4.1.2 and Intel Compiler (ICC) 10.1, for evaluation. For comparison, a Malta board [10] by MIPS Technologies is used as a MIPS real machine. This board has a MIPS 4KEc core (240 MHz) and 128 MB main memory.

Execution speeds of SimMips and the real machine are shown in Figure 7. The unit of measurement is million instruction per second or MIPS. SimMips is operated fastest when using gcc with `-O3` optimization and the simulation speed is 12.1 MIPS. It is about 20% faster than using gcc with `-O2` optimization or icc with `-O3`, and 3x faster than using gcc without optimization. Still, SimMips is about 12x slower than the real machine. But this slowdown can be covered by proper choices of data set.

### 3.3 SimMips as a Lecture Material

This section shows more practical suitability for education by taking an example of using SimMips as a material of a computer architecture lecture. We gave assignments to undergraduate and master’s students. They are required to modify SimMips to measure hit rates of value prediction [8] and data cache. We also demanded submission of a report with a description of the time spent on the each assignment. Undergraduate students have more than a year of programming experience, and master’s students have at least 3 years experience. However, not all the students are familiar with C++ programming.

We received reports from 25 undergraduate students and 26 master’s students. The percentage of the students who solve the problem and the average time spent are summarized in Table 2. Most of the students understood the source code and added necessary mechanisms. The average time required on each assignment was from 5 through 8 hours. We did not compare with using other simulators, but we think the results were good enough.

Along with the reports, we received some feedback about the assignments and the simulator. To provide more comments, documents, and tutorials are most requested. Based on such opinions, we would like to improve SimMips in usability and comprehensibility.
1 void MipsInst::decode()
2 {
3     opcode = (ir >> 26) & 0x3f;
4     rs = (ir >> 21) & 0x1f;
5     ...
6     funct = ir & 0x3f;
7     ...
8     switch (opcode) {
9         case 0:
10            switch (funct) {
11                ...
12                case 33:
13                    op = ADDU_____
14                    attr = READ_RS | READ_RT | WRITE_RD;
15                end
16            end
17        end
18    }
19}
20
1 /* MipsInst::decode() */
2 always@ ( DATA_IN ) begin
3     IDOPCODE = DATA_IN[31:26];
4     IDRS = DATA_IN[25:21];
5     ...
6     IDFUNCT = DATA_IN[5: 0];
7     ...
8     case ( IDOPCODE )
9         6'd0: begin
10             6'd33: begin
11                 IDOP = 'ADDU_____
12                 IDATTR = 'READ_RS | READ_RT | WRITE_RD;
13             end
14         end
15 end
16 ...
17 ...

Figure 8. The expression of decode part of SimMips.

Figure 9. The expression of decode part of MipsCore.

4 SimMips as an Infrastructure

This section discusses a MIPS-like soft processor MipsCore and a simple palm-top embedded system MieruPC (We have called it Simplem [16] in the past).

We used SimMips as an infrastructure to develop MipsCore, that is, taking advantage of hardware-aware design and implementation of SimMips, we partially ported it to Verilog HDL. Since MipsCore is not pipelined, it takes from 8 to 40 cycles for executing an instruction (as discussed later), which differs from the simulator. Also, it does not provide all the MIPS32 instruction. Privilege instructions, floating-point instructions, and multiply-add instructions are not implemented yet.

Plasma, YACC - Yet Another CPU CPU, UCore, etc. are registered in OpenCores [11] as existent MIPS soft processors. The most famous one, Plasma [13], is pipelined and works its own OS. So MipsCore is inferior to it in functional aspect. But our processor is compactly implemented with about 1,150 lines, while the number of lines of Plasma is about 1,600.

The division of instruction on MipsCore is similar to that of SimMips (See Section 2.3). The difference is that write-back and setnpc stages are integrated into a single write back stage. Therefore MipsCore has 7 stages: fetch, decode, register fetch, execute, 2 memory access stages, and write back. The memory access stages are skipped if the instruction is not load/store. Basically each stage takes a cycle, but it spends 4 cycles to fetch and one of the memory access stages, and 32 cycles to multiply or to divide. So it takes 8, 13, or 40 cycles to execute a general instruction, a load/store, or a multiplication/division, respectively.

Figure 8 shows the expression of decoding the addu instruction in SimMips, and Figure 9 shows the same part in MipsCore. The block from 3rd through 7th line divides an instruction into several fields. While this division is done with shift and mask operations in C++, it was done by cutting bits out in Verilog HDL. Actual decode operation begins at the 9th line. The instruction is sorted out there according to its opcode and funct fields. The implementation of instructions on the decode stage and the execute stage, which accounts for more than half of the code, is done by automatic replacement like this. The other stages are not so easily implemented because of differences in interfaces with the register file or the memory controller. The verification of MipsCore is also similar to that of SimMips. That is, we drew a comparison of execution logs between the Verilog simulation of MipsCore and the simulator. The implementation and the verification are done by a master’s student and two undergraduate students. The time spent for them is about a week. Though it is natural to take a long time to develop hardware, using SimMips makes such a development much easier.

We have also developed a simple embedded system MieruPC, which includes MipsCore as a processor core. MieruPC consists of an FPGA board, a mother board, and an LCD unit. A photograph of an FPGA board on a mother board is shown in Figure 10. The FPGA board contains a Spartan-3E FPGA (XC3S250E, speed grade -4), a 4MBit SRAM, a JTAG connector, and 24 I/O pins. The mother board has some I/O connectors and switches: from left, a power connector, an MMC or multimedia card connector, a PS/2 keyboard port, a reset button, an LCD connector, and a power switch. As the LCD unit, a command interpreter LCD unit ITC-2432-035H by Integral Electronics is used.

When MieruPC is powered up or reset, an application program, which is statically linked with a startup routine, is read from fixed part of a multimedia card. Then it
runs on MipsCore and sends commands to the LCD unit through memory-mapped I/O. According to the commands, the LCD unit displays texts and graphics. A picture of a sample application running on MieruPC shows in Figure 11.

Table 3 shows the file organization of the current version of MieruPC. In addition to MipsCore, I/O controllers and an initializer are implemented on the FPGA. The total number of lines is about 2,200. MieruPC uses 2,360 slices (Xilinx’s criterion of logic size), which occupies 96% of available slices in XC3S250E, when Xilinx ISE 11.1 is used as a logic synthesis tool. Maximum frequency is about 76 MHz according to the tool, but the actual limit may be slower because of timing constraints of SRAM. We run MieruPC with 54 MHz for safety.

We verified MieruPC by comparison of architecture state (like the verification of SimMips and MipsCore), unit tests of controllers, and checking behavior of various kinds of applications. We spent about 2 weeks for basic implementation and verification of MieruPC.

We gave an optional assignment to develop an application for MieruPC in our lecture (mentioned in Section 3.3). Some eager students submit interesting programs. A shot of one of the applications is shown in Figure 12. It should be a motivation for students to see a compact embedded system “visibly” working.

5 Conclusion

We have developed a simply coded system simulator SimMips for education of computer architecture and embedded system. It satisfies high comprehensibility and sufficient simulation speed simultaneously. In this paper, we described the concept, the implementation, and the evaluation of SimMips, and clarified high effectiveness for ed-
ucation. We also discussed a palm-top embedded system MieruPC based on SimMips. SimMips Version 0.5.5 (as of October 2009) is downloadable from http://www.arch.cs.titech.ac.jp/SimMips/ and further information of MieruPC project is available from http://www.arch.cs.titech.ac.jp/mieru/. We are planning to develop a comprehensive educational platform including a system simulator, hardware, an OS, and so on. Then we would like to open the accomplishment actively.

Acknowledgement

A part of the development of SimMips was supported by Core Research for Evolutional Science and Technology (CREST), JST.

References