Lab 4: MIPS Simulator

Due 3/13/2014 11:59 PM

In this project you will investigate building a MIPS simulator to examine performance metrics of a given program. You are provided the support code for the simulator; you will complete the simulator and collect statistics with it.

Tasks

The simulator should handle a subset of the MIPS instruction set. The "Shang" benchmark on which you will do measurements has the following instructions that you must support: ADDIU, ORI, SW, SB, LW, LBU, LB, SLTI, SLTIU, BNE, BEQ, BLEZ, LUI, JAL, J, ADDU, SLT, JR, SLL, SRL, SRA, ADD, SUB, SUBU, AND, OR, XOR, NOR, JALR. Particularly tricky will be the byte memory operations. The C++ implementation of the simulator only reads and writes memory on a word granularity (4 bytes at a time), so make sure you work within that constraint.

Your tasks are as follows:

1. Complete the simulator so it successfully runs the Shang benchmark (details below). The simulator, as provided, implements only a handful of instructions. You need to add support for other instructions.

   Some advice:

   - DON'T start with Shang. Don't plan on running only Shang.
   - Run small tests to make sure your instructions are working. You can even write .s files to test specific instructions. Then you know exactly what you're testing.
   - Test your statistics on something other than Shang (and Shang, of course).
   - Don't procrastinate.
   - Learn how to use GDB. (resource: http://www.cs.cmu.edu/~gilpin/tutorial/)
   - Shang takes a long time to run in simulation. It's difficult to debug on a large program like Shang, and if you don't write any smaller tests to make sure your statistics and individual instructions work, you will be very frustrated the night before the assignment is due when Shang does not run correctly and you don't know why. You should particularly think about writing test code to test SB and LB. If and when you do write test code, keep in mind that you can only use integer instructions, and you can't use library routines (like printf); your best bet will be a "main" routine that does internal computation and returns a single value as a 'return' at the end of the program.

2. You also need to collect statistics on the benchmark that you run to answer the questions in the writeup. Please measure dynamic (runtime) statistics for:

   a) Number of instructions
   b) Number of R-Type, I-Type and J-Type instructions
   c) Number of Memory Reads and Memory Writes

d) Number of Register Reads and Register Writes
e) Number of Branches, both forward and backward, taken and not taken in each direction. Do not include jumps as branches.
f) Branch and jump delay slot usage. Is the branch or jump delay slot filled with a useful instruction or a useless (nop) instruction?
g) Forwarding. Count how many times a value must be forwarded: From the Ex/Mem forwarding unit, and from the Mem/WB forwarding unit.
h) Load-use hazards. Does the instruction following a load instruction use the result of the load instruction in its ALU stage?
i) Cache performance. For a 256-byte direct-mapped cache, what is the best block (line) size in bytes? You could choose to have 64 entries in the cache of 4 bytes each, or instead 1 entry of 256 bytes, or anything in between. What has the best hit rate?

You will measure these statistics for two different compilations of the Shang benchmark, one with no compiler optimizations, one with the -O3 compiler optimization. Example output from a sample simulator for the shang benchmark at level -O1 optimization is shown below:

./mipsim -s -c 256 -f shango1.sim

Starting at PC 40052c

Total number of dynamic instructions: 386447534

RType: 92393908

IType: 286747099

JType: 7306527

Number of Memory Reads: 47382214

Number of Memory Writes: 33369840

Number of Register Reads: 396532268

Number of Register Writes: 287758105

Number of Forwards:

From Ex Stage: 72262782

From Mem Stage: 33733199

Branches:

Forward:

Taken: 6104032

Not taken: 14152522

Backward:

Taken: 36047667

Not taken: 1708840

Branch delay slot:

Useful instruction: 52756579

Not useful instruction: 5256482

Jump delay slot:

Useful instruction: 8736743

Not useful instruction: 412206

Load Use Hazard:

Has load use stall: 39525516

Has load use hazard: 0

Has no load use hazard: 47382214

256 byte cache (blocksize 4 bytes): 55885096 hits, 24866958 misses (hit rate: 69.2058%)

256 byte cache (blocksize 8 bytes): 63396666 hits, 17355388 misses (hit rate: 78.5078%)

256 byte cache (blocksize 16 bytes): 65941581 hits, 14810473 misses (hit rate: 81.6593%)

256 byte cache (blocksize 32 bytes): 67874834 hits, 12877220 misses (hit rate: 84.0534%)

256 byte cache (blocksize 64 bytes): 67896111 hits, 12855943 misses (hit rate: 84.0797%)

256 byte cache (blocksize 128 bytes): 56138062 hits, 24613992 misses (hit rate: 69.5191%)

256 byte cache (blocksize 256 bytes): 2431222 hits, 78320832 misses (hit rate: 3.01072%)

Note: this took approximately 33 seconds to run when the simulator was compiled with -O3 optimization.

Deliverables

You have two deliverables. First, you will submit your source code as a single archived (zip, tar, tgz, etc.) file through handin. You will also submit a one-page PDF writeup (no longer than one page!), with text and graphs/figures/data if necessary, also through handin, that presents your conclusions to the following questions:

1. Precisely compute the CPI for the shang program at both optimization levels.
2. If you are building a processor and have to do static branch prediction (meaning you have to assume at compile time whether a branch is taken or not), how should you do it? You can make a different decision for branches that go forward or backward.
3. How good is the GCC MIPS compiler in filling the branch delay slot?
4. How good is the GCC MIPS compiler in avoiding load-use hazards?
5. If you are building a 256-byte direct-mapped cache, what should you choose as your block (line) size?
6. What conclusions can you draw about the differences between compiling with no optimization and -O3 optimization?

Compiling MIPS Programs

Here is a brief description of how to compile MIPS programs on the department machines. Remember—DON'T just compile Shang. Make your own programs.

1. Place the compiler tools in your path. They are located on CSC department Linux machines at /home/clupo/mips/bin.

   export PATH=/home/clupo/mips/bin:${PATH}

2. Compile the MIPS binary for your program. The static and soft-float options are necessary because we cannot dynamically link files during simulation, and our simulator does not support floating point operations. You will specify yourFileName and mipsBinaryName.

   mips-elf-gcc yourFileName -static -msoft-float -o mipsBinaryName

3. If you are compiling an optimized binary, add -O3 to the arguments to mips-elf-gcc.
4. Generate the simulation file from the binary (you will specify mipsbinaryname and simfilename).

   mips-elf-objdump -D mipsbinaryname | gensimcode > simfilename

5. This produces a .sim file, which contains the starting PC and a hex dump of the instructions (listed under INSTRUCTION MEMORY) and the initial data (listed under DATA MEMORY). This is the input to your simulator.

The Simulator

You get 5 source files (plus a header file). You should have to only modify two of them. Here is what they are and what you should do:

1. classify.cc associates opcodes with their string equivalents. You shouldn't need to touch it.
2. execute.cc is the major file you should change. It calls execute() each cycle, fetches the current instruction, and evaluates that instruction, changing the machine state (the data memory dmem, the register file rf, and the program counter pc). You will find a few instructions as examples in this file. You need to fill in the rest of the instructions and also capture all necessary statistics.
3. main.cc contains the main routine and parses command-line arguments which may be helpful in debugging:

   - -p dumps the parsed program at the start of the simulation.
   - -d dumps the contents of data memory (all non-zero data memory entries) and the register file after the end of the simulation.
   - -i prints every instruction as it executes.
   - -w prints every write to data memory.
   - -s prints statistics at the end of the program.
   - -c # (fill in # with a size in bytes) enables caches of size #. For this assignment, -c 256 is probably most appropriate.
   - -f simfilename runs the sim file specified. This option must be specified.

   You should not have to change this file.

4. parse.cc parses the sim file. You shouldn't need to change it.
5. mipsim.cc and mipsim.hpp contain the core data structures. You should only need to change one routine in this file, Cache::access, which currently returns a cache miss (false) for every cache access. You will need to enter tags into the cache ("entries") on a miss and check tags on every access.

   If you call "mipsim -c 256" the system automatically instantiates several caches: 256B, 4 byte cache lines; 256B, 8 byte cache lines, etc. (up to 256B, 256B cache lines). Every time you access memory (lw, sw, lbu, etc.), YOU call caches.access(addr). The system automatically calls Cache::access on each cache. YOU also need to write Cache::access(), which

   - Keeps track of cache tags (not data, not valid, ...)
   - Determines hit or miss
   - Keeps stats: hits++, misses++

   Start by copying mipsim_unfinished.cc to mipsim.cc and execute_unfinished.cc to execute.cc, then 'make' to create the mipsim executable.

Getting Started

You might want to start with the "fib" test program that is supplied. It should return the value 59 in $2. However, when you run it, it will fail because not all the instructions are implemented (it will say that OP_J [jump] is not implemented), and you should add those first until the program runs correctly.

```
$ ./mipsim -d -f fib.sim
Starting at PC 400168
DATA:
fffffe8: a
ffffffec: 59
fffffff0: 59
fffffff4: 37
RF:
0
0
59        <-- $2 = return value
22
...
```

Next, you will want to collect statistics and fix the Caches::access routine.

You should make sure your toolflow works properly by the end of the first week, 2/20 (this means you can compile and start running a program like fib, even if you don't change anything in the simulator). You should have a working fib, including statistics, by the end of week two (2/27). Alternatively, try to have a working shang (no statistics) instead by 2/27. If you don't meet these milestones, you're in danger of not finishing the assignment.

Files (in a compressed tar file)

1. Fibonacci (a small test that you might want to run to get your simulator up and running): fib.c, fib.sim
2. Shang: shang.c, shang.cc (the original C++ source; might be easier for you to play with if you're trying to get an idea how Shang works)
3. Simulator source files: classify.cc, execute.cc, main.cc, parse.cc, mipsim.cc, mipsim.hpp, Makefile.

Technical Notes About the Simulator

The simulator is written in C++ and is written for the g++ (gcc) compiler for both big and little endian machines. (You may wish to look through the source code to see how big and little endian machines are handled differently.) You are welcome to run the supplied code on any machine with any compiler, but I will only support g++, and have only tried the code on x86 and x86_64 Linux.

If you are working on a different machine, the following gcc command tells you the built-in defines for gcc:

gcc -E -dM - < /dev/null

In writing the simulator, special attention was given to the behavior of bitfields and how they are laid out by the compiler.

The Shang Benchmark

Shang is code that solves a puzzle (solution) from PuzzleHunt II. MIPS programs place their return code in $v0 (Register $2) and if your code successfully runs the Shang program, it will return a 1 in $v0; if you don't have a 1 in $v0 then you have implemented your simulator incorrectly. You should be able to locate the strip numbers that are the solution on the stack if your program successfully completes.

Collaboration

You may talk to your classmates about strategies for completing this assignment, but all work you turn in must be solely the work of you and your partner. No code or results may be shared in any way between different groups with the exception of test programs. If you use a test program from another student, credit that student in your writeup.

Project Notes

1. Here's what a successful run should look like:
2. $ /home/khaworth/mips/bin/mips-elf-gcc shang.c -static -msoft-float -o shang.mips
   /home/khaworth/mips/bin/../lib/gcc/mips-elf/4.1.2/../../../../mips-elf/bin/ld: warning: cannot find entry symbol _start; defaulting to 0000000000400040
   (and produce a shang.mips executable)
3. You may see the error "badType in ParseMem" when running the mipsim simulator on your sim file. This happens when there is no data segment in the file. I could fix the simulator and put up new simulator files, but it's much easier if you just add the following line after the last line of your sim file:
   Data Memory
4. It'll look like this:
5. $ tail fib.sim
   0x00400250 0x27bd0018
   0x00400254 0x27bdffe0
   0x00400258 0xafbf0014
   0x0040025c 0x0c100010
   0x00400260 0x00000000
   0x00400264 0x8fbf0014
   0x00400268 0x27bd0020
   0x0040026c 0x03e00008
   0x00400270 0x00000000
   Data Memory
   $