Homework Assignment #4 - CS U380 - Fall, 2008

Due date

Turn in your solution to this programming assignment at the beginning of the final exam, 8 am, Thursday, December 18.

General instructions about homework

Your answer will be a script which shows your version of the file "computer.c", a successful compilation, and a few excerpts from the simulator's output. Show that you can execute the first dozen or so instructions properly, and that the register contents when the program terminates are the right ones. Look at the program file "gcd.s" and you will see that the final results of the calculations are in registers 15, 8, 9, and 10. Once you have a good script, edit out all the extra stuff in the middle. I should be able to see your answers without going through hundreds of pages of output.

The software and test files for the assignment are in the directory "/course/csu380jc/.www/hw4"

Longish exercise: A miniature MIPS simulator

Turn in your answer to this question by modifying the framework files listed below. You should not need to create any new files. If you do, you are responsible for modifying the build scripts (in "Makefile") appropriately (but you are most definitely allowed to leave the "Makefile" exactly as it was supplied to you!)

Background

In Homework 3, you created a disassembler for a subset of the MIPS assembly language. In this homework, you will create an instruction interpreter for a subset of MIPS instructions. Similarly to SPIM, your interpreter will fetch, decode and execute MIPS machine instructions. Your disassembler is almost the decode part, but for this project, instead of printing the instruction, you need to perform the operation it specifies on the machine state. There is one important difference, though– the MARS takes in ".s" files containing assembly language, and then it assembles them into binary machine language; your simulator will simply load ".dump" files containing binary machine language.

Description

The files "sim.c", "computer.h", "dis.c", and "computer.c" comprise the framework for a MIPS simulator. Complete the program by filling in appropriate procedures in "computer.c". For full credit, your simulator must be able to simulate the machine code versions of the following MIPS instructions: addu, addiu, subu, sll, srl, and, andi, or, ori, lui, slt, slti, sltu, sltiu, beq, bne, j, jr, jal, lw, and sw.

The framework code does the following:

1. The framework code provides simulated data memory starting at address 0x00401000 and ending at address 0x00404000. It stores instructions beginning at 0x00400000. (See the notes below about assembler directives and the data segment.) The framework code starts by reading the machine code into the simulated memory, starting at address 0x00400000. (Similarly to the behavior of SPIM, addresses from 0x0000000 to 0x00400000 are unused.) We assume that the program will be no more than 1024 words long. The name of the file that contains the code is given as a command-line argument, as in homework 3.

2. Then, the stack pointer is initialized to 0x00404000, all other registers are initialized to 0x00000000, and the program counter is set to 0x00400000.

3. Then, the framework code enters a loop that repeatedly fetches and executes instructions. Your job is to provide the code for the SimulateInstr( ) function, plus any auxiliary functions that are appropriate; this function should simulate the execution of exactly one instruction by reading, modifying, and writing machine state information in the machine data structure, whose type (struct SimulatedComputer) is described in "computer.h".

If the program encounters an instruction that accesses memory illegally, it should exit with failure status by calling exit(1). If the program encounters an instruction that's not one of the ones listed above, it should exit with failure status, by calling exit(1). If any error whatsoever occurs, your code should exit with failure status by calling exit(1). (You get the idea...)

4. You should also simulate program termination. The two instructions

```
ori  $v0, $0, 10
syscall
```

should cause your program to terminate normally,

so what you should do is this: if the syscall instruction is executed AND $v0(2)$ contains the value 10, you should quit by calling exit(0) to signal a successful completion of your program. Any other invocation of the syscall instruction is to be treated as an illegal instruction, as in item 3 above.

Framework code

The framework code supports several command line options:

sim -i runs the program in "interactive mode". In this mode, the program prints a ">" prompt and waits for you to type a return before simulating each instruction. If you type a "q" (for "quit") followed by a return, the program exits. If this option isn't specified, the only way to terminate the program is to interrupt it (with a control-C), or to have it simulate an instruction that's not one of those listed in the previous section.

sim -r prints all registers after the execution of an instruction. If this option isn't specified, only the register that was affected by the instruction should be printed; for a branch, a jump, or a store, which doesn't affect any registers, the framework code prints a message saying that no registers were affected. (Your code can signal when a simulated instruction does or doesn't affect any registers by returning -1 in the changedReg argument to SimulateInstr( ).)

sim -m prints all data memory locations that contain nonzero values after the execution of an instruction. If this option isn't specified, only the memory location that was affected by the instruction should be printed; for any instruction that's not sw, the framework code prints a message saying that no memory locations were affected. (Your code can signal when a simulated instruction does or doesn't affect a location in memory by returning -1 in the changedMem argument to SimulateInstr( ).)

sim -d is a debugging flag that you might find useful. All the printing in the framework code is done in the PrintInfo function. A Disassemble( ) function that calls the sample solution to the disassembler exercise from homework 3 is also provided in the file "dis.o".

Testing your program; important caveats

A test file is available in the directory "/course/csu380jc/.www/hw4/tests"; please read the "README" file included therein. Programs that run under your interpreter also run under the MARS sinulator; but note that MARS accepts only the assembly-language form, and your program will accept only the machine-language form. These are real MIPS programs. If your code runs in MARS but not in your instruction interpreter, it either means you didn't follow the rules below (we think that's all there are!) or your interpreter has a bug in it, or both. However, there are a few rules you need to follow if you'd like to write your own MIPS programs that run both under this small interpreter and under SPIM:

Assembler directives

Your test code may not depend on any assembler directives. In particular, you can't use directives to set up memory. Instead, you must write MIPS code yourself to store up any values in memory that you wish to use. This inconsistency is an artifact of our "dump" files not being real object files, which contain the necessary information to reconstruct the data segment.

Data segment

You won"t be able to use spim's normal data-memory section (0x10000000 to 0x10001000) or rely on the fact that the stack is between 0x7fff0000 and 0x80000000. Instead, the framework code will point $sp ($29) at the upper edge of memory when the simulator starts. If you need "global" data memory, use the global pointer ($gp) by setting $gp = $sp - 0x2000 and then using addresses offset from $gp.

Labels and addresses

You won"t be able to use any absolute addresses or any labels except for those labels which point to instructions, because, as noted above, MARS uses a slightly different memory layout than your interpreter. For example, MARS's assembler

will assume a location for the data segment that simply won't exist when your interpreter runs the program, so any labels that point into the ".data" segment will be invalid pointers.

Unimplemented instructions

Obviously, your test code may not use any instructions that aren't implemented in the instruction interpreter. The one exception is in the case where you want the program to end; the done SPIM macro expands into an ori instruction followed by a syscall, so it provides a convenient way to stop your program in your simulator as well as in SPIM. Watch out for pseudoinstructions, which often expand to multiple MIPS instructions; you may not have implemented some of the expanded instructions. You can load the assemblylanguage source file into xspim to see how the instructions are translated.

Careful!!!! The green card is wrong about the jal instruction. The truth is that it puts $PC + 4$ in register 31, not $PC + 8$.