Lab 6 – Tiny MIPS Simulator

Problem Overview

You're going to write a MIPS simulator, that is, a program that can simulate the execution of MIPS instructions. More precisely, your simulator handles exactly seven instructions: **add**, **sub**, **or**, **xor**, **addi**, **beq,** and **halt**. (The last is not an actual MIPS instruction, but is useful for obvious reasons. No worries, it will be implemented for you.)

Your program takes as input a program written using only those instructions. You execute each of the instructions, in order, halting when you encounter a halt instruction. As you execute each instruction, you update values stored in your simulator's registers. That's pretty much all there is to it.

Your simulator happens to be written in MIPS assembler as well. For this assignment, you will be using the MARS simulator, which can be found in the simulator skeleton package (see below).

Program Input

Input to your program is a list of instructions, encoded as decimal integer representations of the 32-bit machine instructions. For example:

```
5
537395212
554237972
554303505
19552293
1073741848
```

The first integer indicates how many instructions follow. The integers that come next are instructions, written in decimal (because the only routine available to read an integer from assembler requires decimal input). They correspond to the assembler program, and the hex encoded machine instructions, shown in the following output:

```
Address     Code (hex)    Code
00000000    0x2008000c    addi   t0,$0, 12
00000004    0x21090014    addi   t1, t0, 20
00000008    0x210a0011    addi   t2, t0, 17
0000000c    0x012a5825    or     t3, t1, t2
00000010    0x40000018    halt          # not MIPS, but needed for the simulator
```

Note that the last instruction should be a halt instruction to tell us when to stop the simulation.

Program Output

None. We'll run your program in the simulator and examine the contents of the simulated registers when it halts.

Skeleton Code

Having a look at the simulator skeleton code will probably make things somewhat less confusing. At the top, it allocates space to hold the program to be simulated and to keep the values in the simulated registers. In the *.text* section, the code begins by reading in the input: first the number of instructions in the input and then the instructions themselves. It then initializes the simulated PC, as a pointer into the memory holding the program to be simulated and goes into a loop fetching instructions and updating the simulated PC to mimic sequential flow control. The skeleton doesn't actually simulate the effect of each instruction it fetches, though - *that's what you'll implement.* Instead, it simply checks whether or not the instruction just fetched is a halt. If not, it fetches the next instruction. Otherwise, the skeleton program itself halts.

You may consider it useful to consult the MIPS instruction formats here or the MIPS green sheet.

Important Notes

You can (and should!) assume in your code that the input is always error free - there are 50 or fewer instructions, and each is one of the seven your simulator supports.

We know that this is terrible programming practice, but makes life a lot easier, especially when hand coding assembler.

You must use only the s0-s7 and t0-t9 registers in your input programs.

Preparing Test Input & Running your Simulator

It's a bit clunky to prepare the input to run your simulator. The basic process is

1. Create a small assembler program (testProgram.s) – one is provided in the skeleton - that uses only the instructions your simulator supports.
2. Assemble it using MARS and dump the instructions in hex (details here)
3. Convert the hex to decimal.
4. Run your simulator and type in the input.

That's so painful that we're providing some automation to help. The distribution (see below) comes with those simple Linux Bash scripts.

1. $ ./gen-input.sh <testProgram> <inputFile>
   will assemble <testProgram>, run MARS on it to get the hex coding of the instructions, then extract the hex instructions and turn them into decimal, and finally create a file <inputFile> that is suitable for feeding as input to your simulator.

2. $ ./run-sim.sh <mipsSim.s> <inputFile>
   will run your simulator (mipsSim.s) using MARS, providing it with input from <inputFile>.

Downloading Starter Files

Download mips-sim.zip, and save it in the directory you want to work in. After unzipping the file, it produces files mipsSim.s, gen-input.sh, run-sim.sh, testProgram.s, and Mars_2194_a.jar (the MIPS simulator to be used in this assignment).

Acknowledgment

Adapted from John Zahorjan & UW CSE 378 winter 2007