CSE 30 - Computer Organization and Systems Programming
Programming Assignment #6
Due 2PM, Thursday December 3, 2009

This project involves creating a MARS LITE simulator for a subset of the MIPS instructions, which we call MIPS LITE. It is an instruction set with fewer calories and low carbs. Best of all, it tastes great and is less filling.

To make it interesting, MARS LITE must be written in MIPS assembly language. In essence, you are using MIPS and MARS to create a mini-MARS written in MIPS. Your program will be called by a shell that establishes the environment and will execute until it does a return back to the shell.

MIPS LITE
The MIPS LITE instruction set consists of the following MIPS instructions:

- Loads and stores: lw, sw
- ALU operations: add, addi, sub, and, andi, sll, slt, slti, lui, mul*, mac*
- Branches and jumps: jr, jal, beq, bne

MUL and MAC are not real MIPS instructions. Their functions are described as follows:

MAC (opcode=0, function=0x17) (Multiply and accumulate)

How to call: mac $t1, $t2, $t3

Function: $t1 acts as an accumulator. This instruction takes $t2 * $t3 and then adds it to $t1, and then stores it back to $t1, e.g. $t1 = $t1 + $t2*$t3. You should count MAC as an R-type instruction.

MUL (opcode=0, function=0x18)

How to call: mul $t1, $t2, $t3

Function: Multiply $t2 and $t3, directly just take the least significant 32 bits from the product to store in $t1, e.g. $t1=$t2*$t3. Count MUL as an R-type instruction.

Environment
Your routine will be called by the given shell routine. The shell allocates memory, loads in a program, and passes you the following values:

- $a0 -- pointer to an area of memory allocated to hold your virtual machine's registers. You will use this allocated memory to simulate your virtual registers. For example, if the loaded program has an instruction which uses register $s0, your simulator will access that register via 64($a0). Recall that $s0 is actually

register 16. But because registers are 1 word (4 bytes) in length, $s0 is found at an offset of 4*16 off of $a0.

- $a1 -- pointer to the *lowest* word within the static memory area (area to be used by programs running on your virtual machine). For example, if the loaded program uses the memory address 0x0004, your simulator accesses that address via 4($a1).
- $a2 -- pointer to the top of the program's stack area. Per the MIPS convention, it is the *highest* memory address within the static memory area.
- $a3 -- pointer to the first instruction in the program. This also happens to be a pointer to the lowest memory position. In other words, registers $a3 and $a1 contain the same address. This is done for your convenience. You may want to keep $a1 constant throughout your simulator's operation, as a reference to your memory space. $a3 can then be used to track where your virtual machine's program is executing.
- Be sure to refer to the memory map for further explanation.

As usual, $ra will have the return address. Execution halts on the simulator's (jr) to the address in $ra. The simulator procedure should return these values:

- $v0 -- the return value for the program you are running
- $v1 -- the total number of simulated instructions executed.

In addition, after you have completed simulation of the program, but before you return (via "jr $ra"), print out the following statistics:

- The total number of executed R-type instructions.
- The total number of executed I-type instructions.
- The total number of executed J-type instructions.

For example, if your simulator executes 100 instructions, and if 57 are R-type, 32 are I-type, and 11 are J-type ... your code should print out the following:

```
R-type: 57
I-type: 32
J-type: 11
```

For our grading convenience, we are asking you to print out your statistics in this prescribed format and only in this prescribed format! You MUST print out your statistics in the above format! Be sure to place a carriage return at the end of each line.

Programs for your Simulator

Programs for your simulator consist of integer values that represent hand-assembled MIPS instructions. The end of the program is signified by a 0. The shell that we give you currently has code1.int hardcoded in the “code” section of the memory, which is denoted by the label “static”. You can change the code you want to execute by putting in different numbers corresponding to different instructions there. Besides changing the instructions, you are only allowed to modify the lines after the sim_mips label.

Testing
We provide you with some test sequences here, but we recommend that you also create your own test sequences. While the programming is to be done individually, we encourage the group creation and sharing of test code. You need to hand compile your test code to generate a file of integers that can be put in the "static" memory section.

- Test program 1: MIPS code, integer values, hex values
- Test program 2: MIPS code, integer values, hex values
- Test program 3: MIPS code, integer values, hex values
- MUL/MAC Test program: MIPS code, integer values, hex values
- Addi Test program: MIPS code, integer values, hex values

Hints

- Be sure to distinguish carefully between the MARS simulator and your own simulator.
- Make sure to setup your system initially in the same way as a real machine would be initialized.
- The only way your machine communicates to the outside world is through $v0 and $v1 when it finishes execution. Clearly this limits the type of programs you'll be able to write.
- Look at green card of the P+H and back of Britton book for information on the instruction formats.
- Work on one instruction at a time. It is probably best to start with addi since there is already a test program with just addi instructions.
- Test your code often.
- Start early. This is not a simple assignment.

Turn in:
You are required to submit the debug_ex.s file. This includes the code given to you as well as your code in sim_mips to implement the simulator. The grading of the project will be done by changing the input program and observing the outputs. Therefore, you must output the correct format answers in $v0, $v1, which will be printed out by the skeleton file. Additionally, you must output the number of R, I and J-type instructions in the specified format (see above) in your code. This is not done for you. Failure to do this or output in the wrong format will result in loss of points.