Project 1

CDA 5155: Spring 2012

Due Date: Feb 22, 11:55 PM (UF EDGE Students: Feb 25, 11:55 PM)

Project1 webpage: http://www.cise.ufl.edu/class/cda5155sp12/assign.html

You are not allowed to take or give help in completing this project. **No late submission will be accepted**. Please include the following sentence on top of your source file: "**On my honor, I have neither given nor received unauthorized aid on this assignment**".

In this project you will create a simple MIPS simulator which will perform the following two tasks:

- The first component of your implementation should be capable of loading a specified MIPS text file[1] and generating the assembly code equivalent to the input file (**disassembler**).
- The second component should generate the instruction-by-instruction simulation of the MIPS code (**simulator**). It should also produce/print the contents of *registers* and *data memories* after execution of each instruction.

You do not have to implement any exception/interrupt handling during simulation for this project.

You can use C, C++ or Java to implement your project. In any case, TA should be able to build and run your simulator in CISE Linux environment (e.g., *thunder.cise.ufl.edu*). If your source code is more than one file, please provide a Makefile which will allow TA to correctly build your project on CISE linux machines. Please provide (on top of your source code as comments) any special notes/assumptions you made about the project that the TA should take into consideration. If you have multiple source files:

- Please submit the **source files** necessary to build and run your project, in a single jar/tar file in e-Learning website. Please do not submit any executables or intermediate files such as .exe or .class files.
- You should make a separate directory to work on your project (e.g., P1) and then use jar -cvf P1.jar * to create your jar file.

Instructions

For reference, please use the MIPS Instruction Set Architecture PDF (available from class project1 webpage) to see the format for each instruction **and pay attention to the following changes.**

Your disassembler/simulator need to support the following two categories of MIPS instructions:

Category-1	Category-2
* J, JR, BEQ, BLTZ, BGTZ * BREAK * SW, LW * SLL, SRL, SRA * NOP	* ADD, SUB * MUL * AND, NOR * SLT

[1] This is a text file consisting of 0/1's (not a binary file). See the sample input file sample.txt in the Project1 webpage.

The format of each category will be described in the following. The instruction format for Category-1 remains the same as the MIPS Instruction Set Architecture PDF as shown in Table 1.

31 — 0
Same as MIPS instructions (with Opcode 6 bits etc.)

Table 1: Format of Instructions in Category-1

If the instruction belongs to Category-2, the first bit (MSB) called the **immediate** bit. The immediate bit decides whether an instruction (with the same mnemonic like "ADD") is a **register-register version** or a **register-immediate version**. That means, if the immediate bit is 0, all the operands are from registers (rd ← rs op rt), which means the instruction format also remains the same as the MIPS Instruction Set Architecture PDF as shown in Table 2.

31	30 — 26	25 — 21	20 — 16	15 — 11	10 — 6	5 — 0
Imm 0 (1 bit)	Opcode (5 bits)	rs (5 bits)	rt (5 bits)	rd (5 bits)	0 00000	Function (6 bits)

Table 2: Format of Instructions in Category-2 in Register-Register Mode

On the other hand, if the immediate bit is 1, the second operand is an immediate value of 16 bits (rt ← rs op immediate), with the modification that **the last 6-bit Function code** is moved after the immediate bit and **its last bit is cut off**, and overwrites the opcode (which you can find in the MIPS Instruction Set Architecture PDF, most of the time it consists only 0s). The 16 bits at the rightmost side serves as the immediate value. **Note that for Category-2 instructions in register-immediate mode, the operation performed is now determined by the 5 bits Function code as shown in Table 3**.

31	30 — 26	25 — 21	20 — 16	15 — 0
Imm 1 (1 bit)	Function (5 bits)	rs (5 bits)	rt (5 bits)	Immediate (16 bits)

Table 3: Format of Instructions in Category-2 in Register-Immediate Mode

All Category-2 instructions in register-immediate mode can be differentiated by the 5 bits Function code as shown below:

	Function code
ADD	10000
SUB	10001
MUL	00001
AND	10010
NOR	10011
SLT	10101

Hence, if the first bit is known to be 0, then we can safely treat the instruction in the same manner as the MIPS format in the MIPS Instruction Set Architecture PDF, and can decode it by checking the first 6 bits (1-bit '0' + 5-bit opcode) and probably the last 6 bits as well. If the first bit is 1, then we should check the next 5 bits Function code. For example, if they are 10000, then we know it is a

register-immediate ADD instruction with the last 16 bits acting as the immediate value. The semantic is rt ← rs + immediate.

Your program will be given a binary (text) input file. This file will contain a sequence of 32-bit instruction words which begin at address **"64"**. The final instruction in the sequence of instructions is always BREAK. Following the BREAK instruction (immediately after BREAK) is a sequence of 32-bit 2's complement signed integers for the program data up to the end of the file.

Your MIPS simulator (with executable name as **MIPSsim**) should accept an input file (**inputfilename.txt**) in the following command format and produce two output files in the same directory: **disassembly.txt** (contains disassembled output) and **simulation.txt (**contains the simulation trace). You can hardcode the names of the output files.

MIPSsim inputfilename.txt

Correct handling of the sample input file (with possible different data values) will be used to determine 60% of the credit. The remaining 40% will be determined from other valid test cases that you will not have access prior to grading. It is recommended that you construct your own sample input files with which to further test your disassembler/simulator.

The disassembler output file should contain 3 columns of data with each column separated by one tab character ('\t' or chr(9)). See the sample disassembly file in the class Project1 webpage.

1. The text (e.g., 0's and 1's) string representing the 32-bit data word at that location. For instructions you should split this into six groups of digits: a group of 6 bits representing opcode/function code, 4 groups of 5 bits, and a final group of 6 bits.
2. The address (in decimal) of that location
3. The disassembled instruction opcode.

Note, if you are displaying an instruction, the third column should contain every part of the instruction, with each argument separated by a comma and then a space (", ").

The simulation output file should have the following format.

* 20 hyphens and a new line
* Cycle: < cycleNumber > < tab >< instr_Address >< tab >< instr_string >
* < blank_line >
* Registers
* R00:< tab >< int(R0) >< tab >< int(R1) >..< tab >< int(R15) >
* R16:< tab >< int(R16) >< tab >< int(R17) >..< tab >< int(R31) >
* < blank_line >
* Data
* < firstDataAddress >:< tab >< display 8 data words as integers with tabs in between >
* ..... < continue until the last data word >

The instructions and instruction arguments should be in capital letters. Display all integer values in decimal. Immediate values should be preceded by a "#" symbol. **Note that some instructions take**

signed immediate values while others take unsigned immediate values. You will have to make sure you properly display a signed or unsigned value depending on the context.

Because we will be using "**diff -b**" to check your output versus ours, exactly follow the output formatting. TA may not be able to debug in case of any mismatch. In other words, mismatches can be treated as wrong output.

Sample Data

The course project webpage contains the following sample programs/files to test your disassembler/simulator.

- sample.c : This contains the C source code for the test program. This is for your reference only.
- sample.txt : This is the input to your program.
- disassembly.txt : This is what your program should produce as disassembled output.
- simulation.txt : This is what your program should output as simulation trace.

Grading Policy

This project is worth **5 points** (10 points for EDGE students).

- 1 point will be given if source files look reasonably correct and can be complied.
- 2 points will be given if your project correctly handles the sample input file:
    - 1 point for disassembly
    - 1 point for simulation (0.5 if only 50% of the cycles are correct)
- 2 points will be given if your project works correctly with our internal test cases. Students will not have access to these internal test cases prior to their submission.
- During re-grading, a student is allowed to make minor changes to their code to reclaim some of the lost points. A minor change is equivalent to changing a symbol such as changing "a > b" to "a <b". Every minor change will cost you 10% of the lost score. In other words, if you make more than 10 minor changes during regarding, you will not get back any points.