Final Programming Project

Due Wednesday, April 19, at 5:00 pm

Logistics

This assignment should be completed in groups of 3. This is not optional (i.e., you are not allowed to complete it on your own). You may only discuss it with your group members and the instructor.

A Caveat

This is the second time I've assigned this project. The first time any project is assigned, issues arise. I am pretty sure we found them all last year, but it may be that something was missed. If that is the case, please let me know early.

Simulation of a Non-Pipelined Processor

For this assignment, you will write code that simulates the operation of a five stage single cycle non-pipelined processor.

Your main program should accept the name of a single input configuration file. The input configuration file will contain information related to the input assembly code, the contents of the main memory module (you need not consider caches, etc., here), whether debugging informaton should be output, and whether the simulation is to operate in single step (single instruction at a time) or batch mode (execute the entire program at once). Details on the contents of the format and contents of the configuration file wil be included below.

General Rules for Processing the Various Input Files

Your code for processing the configuration and other input files must be robust in the following sense:

i. your code must ignore any blank lines or lines that start with the pound (#) sign. In addition, all content, on any line, that follows a pound sign is to be ignored. In particular, this means that though a data memory input file could contain lines for 100 data words, it might only contain, say, 50 data words. Processing this shorter file should not break your code.
ii. When readin a hex value, your code should process successfully both values preceeded with a "0x", and values without it.

Format and Contents of Configuration File

The configuration file is an ASCII text file. It contains entries of the form

```
parameter=value
```

Each <parameter, value> pair must be on a line by itself. Lines containing only white space should be ignored, as should lines that begin with the pound (#) sign, which is used to indicate a comment. A valid configuration file MUST contain each of the following parameters (these are **exact** names, and are case-sensitive):

- program_input
- memory_contents_input
- register_file_input
- output_mode
- debug_mode
- print_memory_contents
- output_file

You may add additional parameters if you wish, but your program must correctly read and operate from any configuration file that contains only the five parameters above (that is, if you add extra parameters, you must set default values in your code so that your program can run even when given a configuration file that does not provide values for the new parameters).

Given the necessary parameter names, it is necessary to describe the form of the values.

program_input: The value of this parameter is a string that provides the name of the input file containing MIPS assembly code. So, for example, if your input code was contained in a file called `Input.asm` in the current directory, this <parameter, value> pair would be listed as

```
program_input=Input.asm
```

in the configuration file.

memory_contents_input: The value of this parameter is a string that provides the name of the input file that contains the contents of the main memory module at the beginning of execution. Details on the format of this file will be provided below.

register_file_input: The value of this parameter is a string that provides the name of the input file that contains the contents of the register file at the beginning of execution. Details on the format of this file will be provided below.

output_mode: This parameter can have only two values: `single_step` and `batch`, according to whether the user wishes to step through the simulation one instruction at a time, or simply execute the simulation of the entire assembly program all at once. To be clear about what I mean by one instruction at a time, when in single_step mode, your code should execute one instruction, print the results of the execution of that instruction, and then pause, waiting for the user to press a key on the keyboard. When the user presses the key, the next instruction should execute and then the code should pause once again, etc.

debug_mode: This parameter is either `true` or `false`. If true, any debugging output should be displayed (whatever the programmer decides that debugging output is). If false, no debugging output should be displayed.

print_memory_contents: This parameter is either `true` or `false`. If true, each output event should print out the **current** contents of the entire register file and the entirety of memory. The format for the register file printout and the format for the memory file output should be identical to the formats given for those two files below.

write_to_file: This parameter is either `true` or `false`. If true, all output for this run of the program should be written to the file specified by the value of the parameter **output_file**.

output_file: This parameter specifies the name of the file to which output should be written, provided the value of **write_to_file** is true.

Format and Content of the Program Input File

The program input file is simply a file that contains MIPS assembly code, one instruction per line. The format is exactly the same as the assembly input files used in Lab 5 and Project 1, though incorporating a few changes. First, the assembly code used in this project handles only a subset of MIPS instructions. In particular, the instructions that your processor must successfully handle are:

- ADD
- SUB
- ADDI
- SLT
- LW
- SW
- BEQ
- J

Branch and jump instructions will contain no labels, but instead will contain hexadecimal strings (preceded by a 0x) that are to be treated as the value of the immediate field in the given instruction. These hex values can represent both positive and negative values (via the twos complement representation). The offset value in ADDI, LW, and SW instructions is represented **in decimal**, and may be either positive or negative. Registers will be encoded as their number preceded by a "$".

You need not check that the assembly file is correctly formatted. You may assume that no input file will contain more than 100 lines of assembly code (though execution of the program might involve more than 100 cycles (e.g., the code contains a loop)).

Format and Content of the Memory Input File

The memory input file is an ASCII text file that represents the contents of the memory module **at the beginning of code execution**. Each line of the file represents a single 4 byte word of memory, stored in big endian order (the address of the word is the address of the most significant byte in the word). The format of each line is <address: data value> (where the colon is included as the separator), where both are hexadecimal values representing 32 bit quantities (because addresses and a word of data both have 32 bits). So a line of the memory input file would look like this:

```
44578220:a7c31002
```

Hexadecimal symbols can be either lower or upper case. Also, the contents of a memory input file will always list consecutive addresses, so the next line in a memory file following the one above would have address 0x44578224. Of course this is overkill (I could just give you the first address contained in the memory module and let you go from there). Instead, I give you the address of each data word, to make lookup easier. Finally, all hexadecimal values can either be written with a leading "0x" or without, so your program must be able to handle either.

You need not check that the memory input file is correctly formatted. You may assume that no memory input file will contain more than 100 words of data.

Format and Content of the Register Input File

The register input file is an ASCII text file that represents the contents of the register file **at the beginning of code execution**. Each line of the file represents a single 4 byte word stored in a register. The format of each line is <register number: data value> (where the colon is included as the separator), where the register number a decimal number between 0 and 31, and the data value is a hexadecimal value representing the 32 bit contents of the register. So, a line of the register input file will look like this

```
21:a7c3be21
```

Hexadecimal symbols can be either lower or upper case. Also, the contents of a memory input file will always list the contents of every integer register, in order, so there will always be 32 lines, beginning with the contents of register 0 and ending with the contents of register 31.

You need not check that the register input file is correctly formatted.

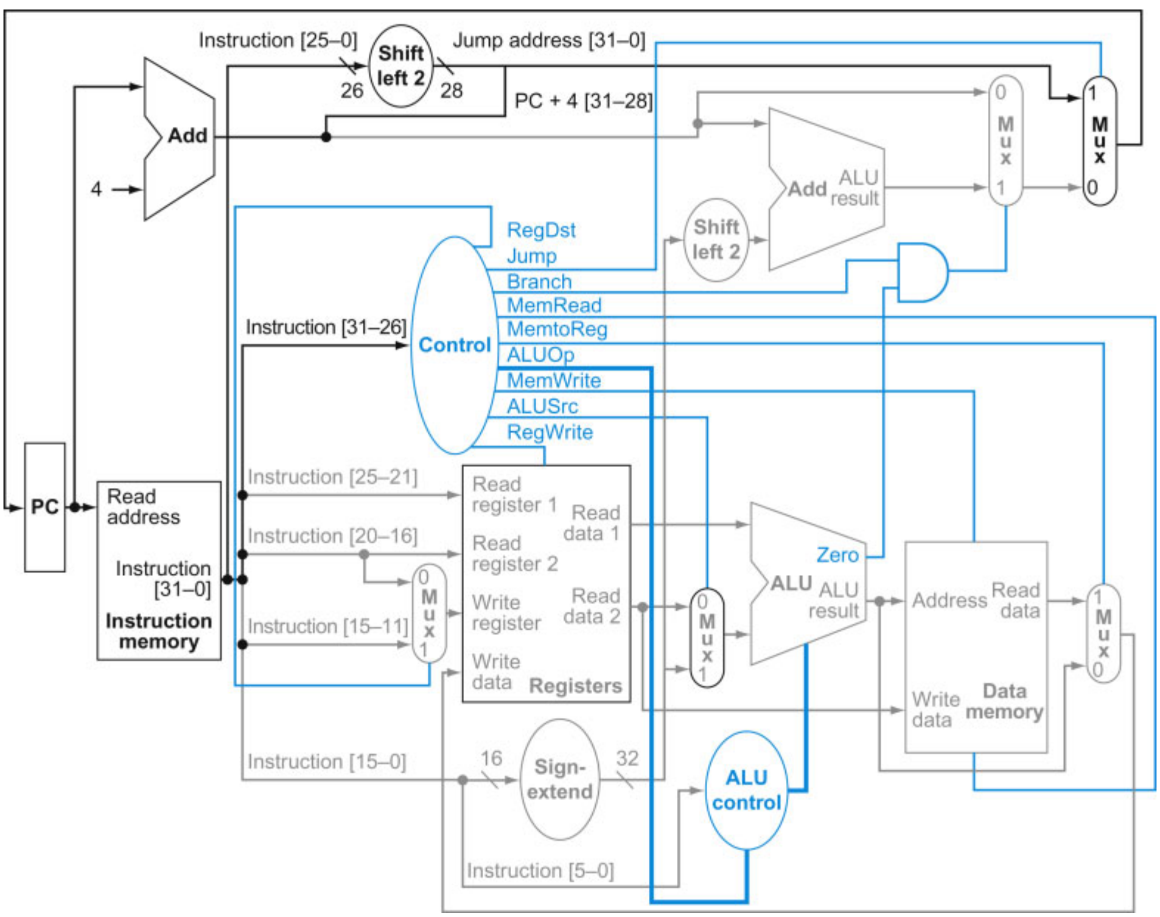
Important Note: Your program MUST NOT modify, in any way, the configuration file, or the contents of the register input, memory input, or program input files. While certainly the memory and register contents will change during program execution, these configuration files are for **initialization** only! Your program will be tasked with keeping track of the contents of each of these modules during execution. Note that this does not preclude you from having your program create files into which it writes information. Only that your program must not change the contents, in any way, of the input files.

What Your Program Must Output

Your program must output **all** control signals on **all** control lines listed in the processor diagram above. In addition, your program must output all input values to all components, all output values provided by a component, and the values of all control signals into and out of all components. This is a minimal output. When print_memory_contents is set to true, your output should include, in addition to the above, the entire contents of the data memory, instruction memory, and register file. Finally, prior to listing the various contents and control values, your program must list, in assembly (not machine code), the instruction that is being executed. To be specific about control signals, your program must display at least the following: RegDst, Jump, Branch, MemRead, MemToReg, ALUOp0, ALUOp1, MemWrite, ALUSrc, RegWrite, as well as the four digit code that is output from the ALUControl to the main ALU. **ALL OUTPUT MUST BE IN HEXADECIMAL FORMAT WITH A LEADING 0x!** So, even if you are outputing a single bit (e.g., the value of the zero line) it must be output in hex (e.g, 0x1 or 0x0).

Objects That Your Program Must Include

I will not specify the exact object oriented design of your program (though I do specify that it must be written in an object-oriented style and in the C++ language). I do, however, specify some of the **objects** (not **classes**) that your program must include. The simulation you are writing must simulate the processor we built in lecture, the diagram of which is shown below.



The diagram above contains 17 functional units:

- PC
- Instruction Memory
- Register File
- Data Memory
- 3 ALUs
- 5 Multiplexors
- 2 “shift left 2” units
- a main control unit
- an ALU control unit
- a sign extend unit

Your program must have an object representing each of these. (Again, I reiterate that I did not say a “class” for each of these, since, for example, five object instances of a single multiplexor class could be used.) The class for a given element must include members for each of the inputs into and each of the outputs out of a specific element.

Your program must, for each instruction executed (if in single step mode) output the values, in hexadecimal form, of all input and outputs to EVERY one of the processor components listed above. Note that for a control unit, this requirement effectively means that your program must know (and output) the values of all control signals! The table below (along with other such tables in Chapter 4) help with this. It shows what control lines must be set to for almost all possible opcodes in this assignment (you’ll have to figure out ADDI and J on your own).

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Starter Code

I will not provide starter code. What I will provide are sample program, register, and memory input files for your use. These are simply provided for the purposes of writing your file reading code. In particular, the program file I supply does not necessarily (and likely will not) represent a correct working MIPS assembly program (so don't assume it does).

Moreover, I will not provide you with correct input/output pairs. When building commercial code, one does not ask the client to provide sample correct input/output pairs. Rather, it is the job of the programmer(s) to test that their code functions as required. You know what each instruction is supposed to accomplish, as well as what control signals should be set for each instruction. Thus you know what your code should do.

A Warning

This project is nontrivial, to say the least. You are being given six weeks to complete it, and are allowed to work in teams, because this requires a lot of time and effort. Moreover, the project takes time to code even if you know exactly what you need to code and how to code it. I estimate that it took me well over 40 hours to code, and I never once stopped to consider what I had to code or how I had to code it – it simply took that long to write the code, because it is a lot of classes, each with several accessor, mutator, and other methods. DO NOT WAIT TO GET STARTED ON THIS!!!! There will be absolutely no extension to the stated deadline. Consider yourself warned.

Deliverables and grading

Your grade will be based 50% on correctness, 25% on design, 15% on testing, 10% on style (including commenting).

- You need NOT create a design document. Rather, you will explain, in some detail, your design when you demonstrate your program for me.
- Your code needs to compile with the –Wall flag without any warnings or errors. Additionally, you must use a Makefile.
- Grading will be done by presentation: I will schedule, via a doodle poll, grading appointments for each group. These 20 minute appointments will take place on the final class day of the semester. During this time **your entire group** will demonstrate your code for me, using configuration, program, and data and register input files that will be supplied at the time of grading. You will also discuss with me your design, as well as the testing code and/or test examples you used to ensure that your program works as it should. Bottom line, you will use your own test examples to convince me, the client, that your code works to specification. After that, I will supply my own code to double check program operation.

Submitting Your Work

Usual method, using the email address

Final_P.qojkm5grt6skbc3j@u.box.com

Your group should only submit a single tar file, and the tar file should be named name-name-name-final_project.tar, where name-name-name are the last names of each of your group members (those who have repeat last names please also include your first name initial). You must submit your project **by Wednesday, April 19, at 5:00 pm.** For this project, late submissions will not be accepted. As usual, code that does not appear to be a genuine attempt to complete the project, or that does not compile, will be given a grade of 0 points.