ENGG3380 - Project

Pre-Knowledge

To perform this project, you will need to understand the function of the data memory block, instruction memory block, and the program counter. At this point in the course, you should have a full understanding of how the CPU will operate in order to complete the project.

Part one

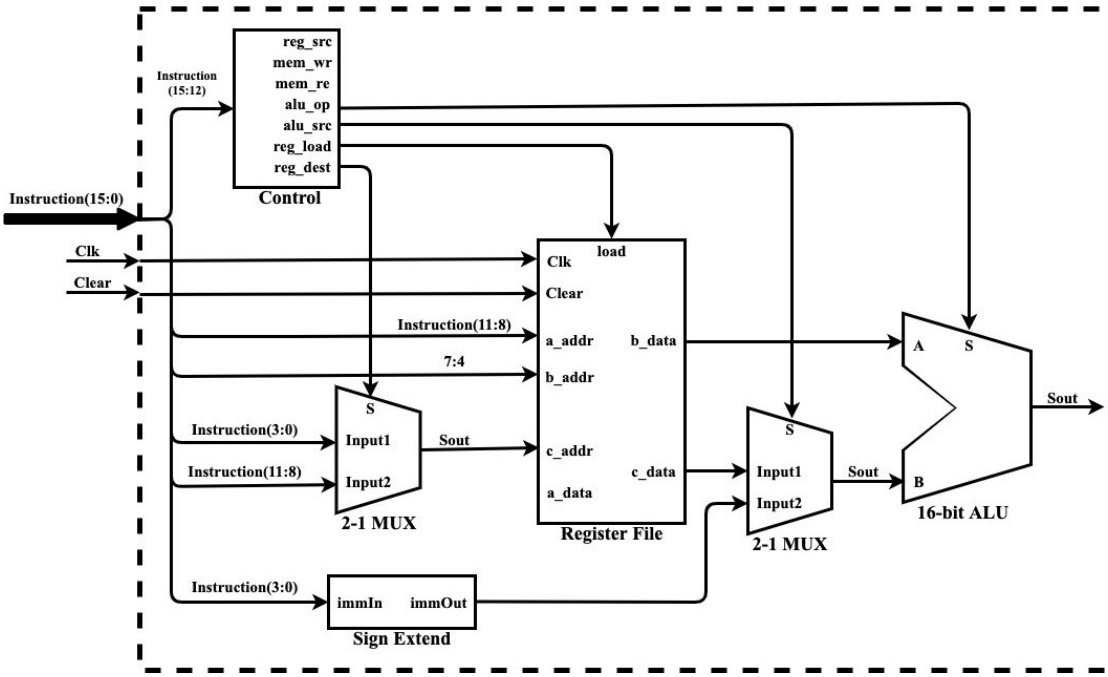
Objective

In this part of the project the students will build off the existing CPU design that should have been completed in Lab 5. A data memory block, 3-1 MUX, and instruction memory block, and a program counter will be implemented. In this project the student design a data memory block and a program counter to automatically feed in instructions to the CPU. The students need to provide a test bench that can be run in order to test the CPU.

Overview

In lab5, we have assembled a CPU up to the following level of functionality below:

CPU_3380.vhd - Top Module
(Previous design)

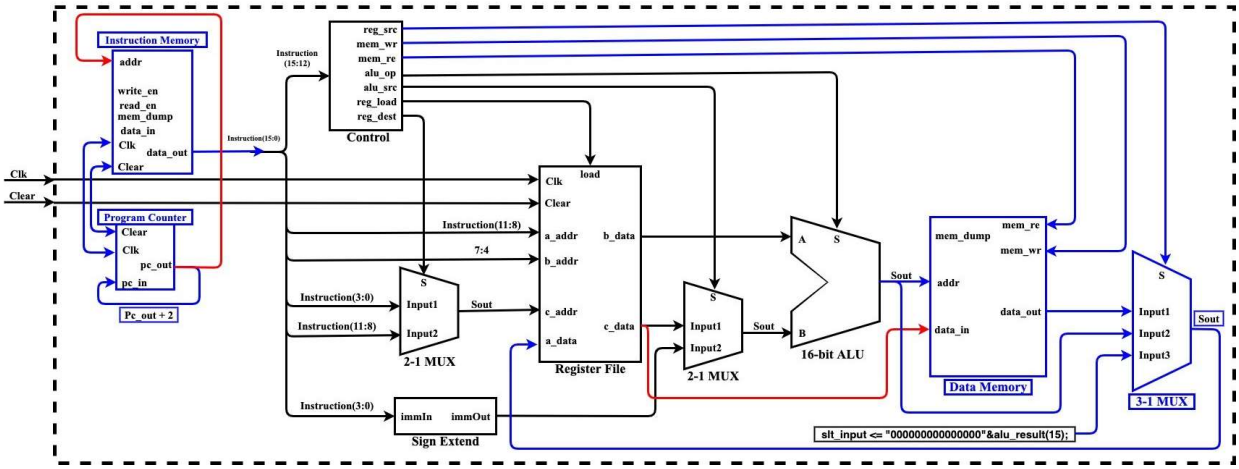


For the project, the VHDL codes are partially provided to you: ALU_16Bit.vhd, ALU.vhd, and_gate.vhd, Control.vhd, CPU_3380_Test.vhd, full_adder.vhd, mux2_1.vhd, mux3_1.vhd, MUX21_4Bit.vhd, MUX21_16Bit.vhd, MUX31.vhd, or_gate.vhd, PC_REG.vhd, Registers.vhd, Memory.vhd, and Signextend.vhd.

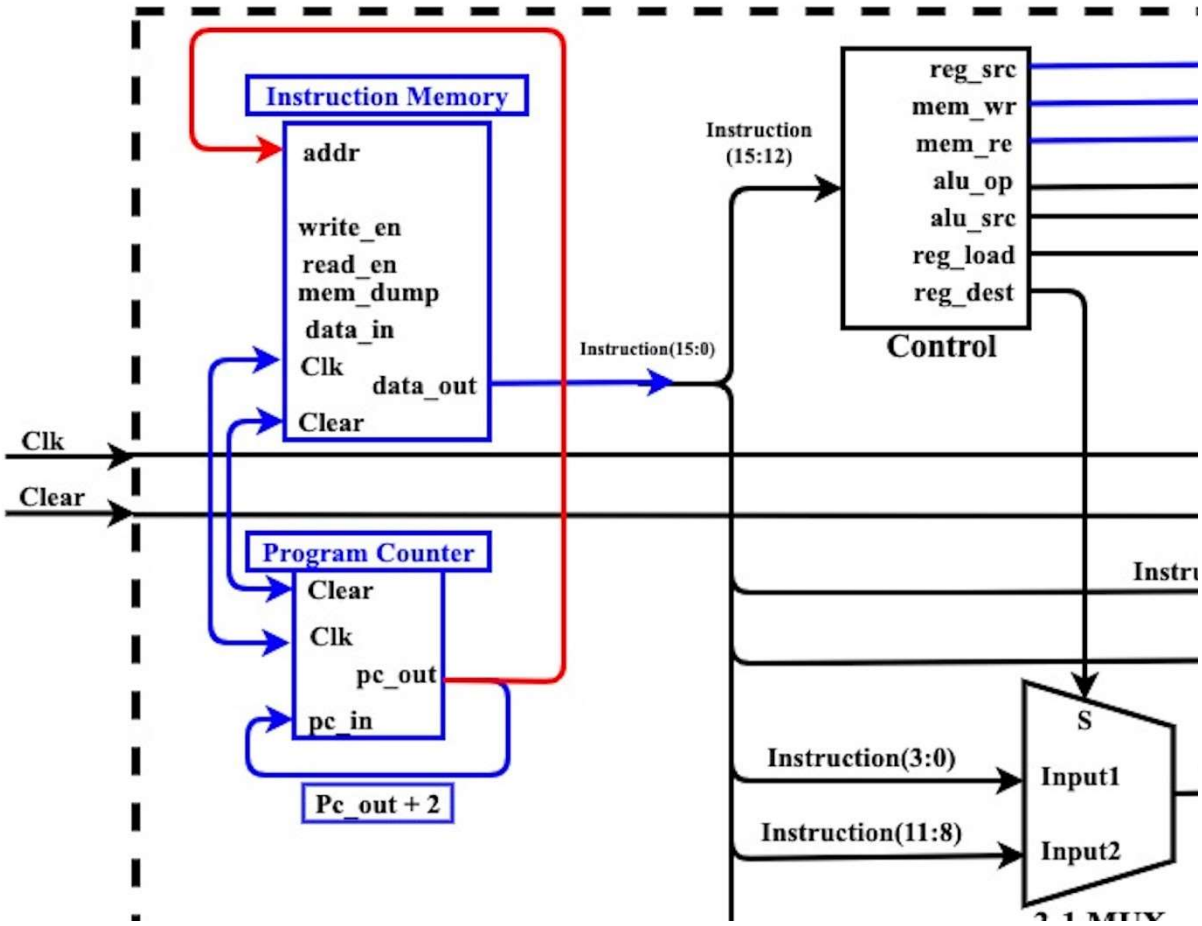
Your need to complete the codes.

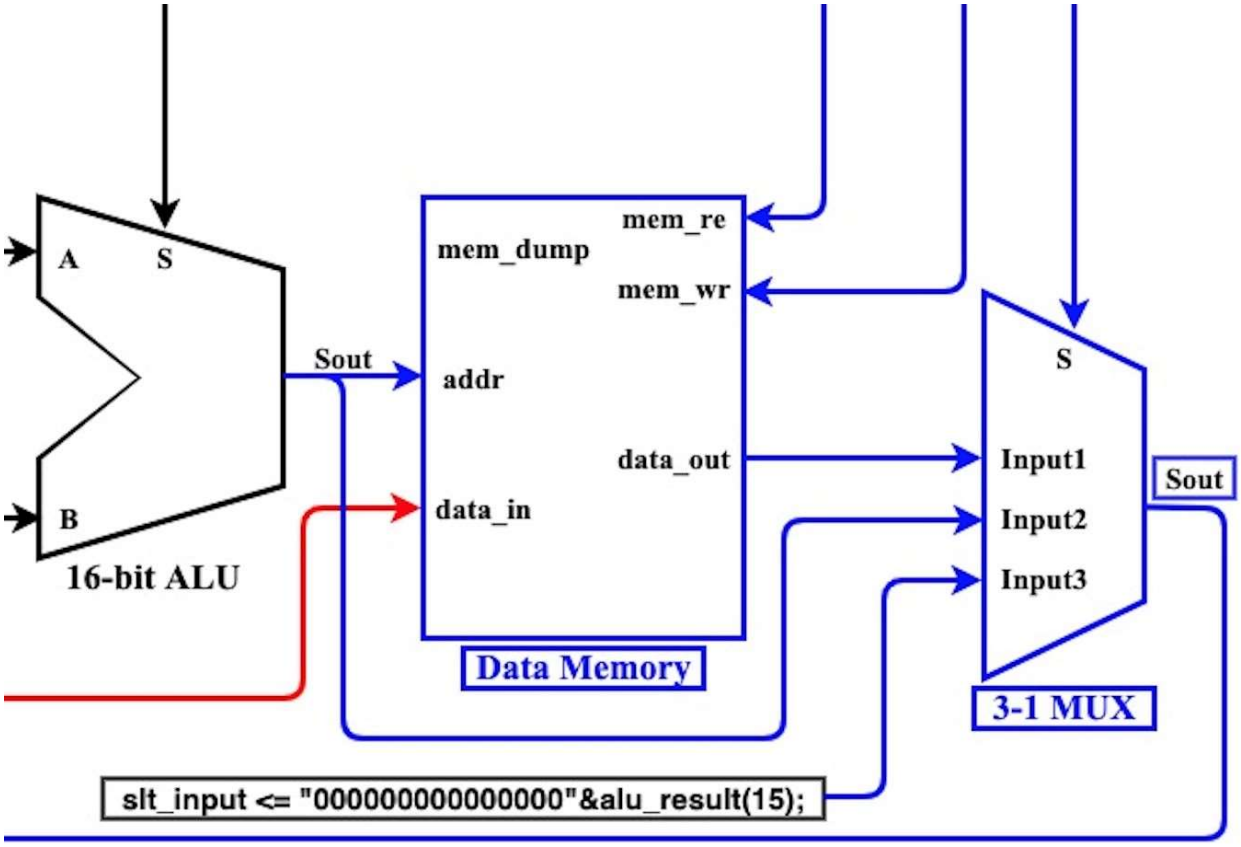
For this lab, we will implement a 3-1 MUX, the data memory block, the instruction memory block, and the program counter. The figure below can show you the full diagram for the CPU:

CPU_3380.vhd - Top Module
(Updated design)



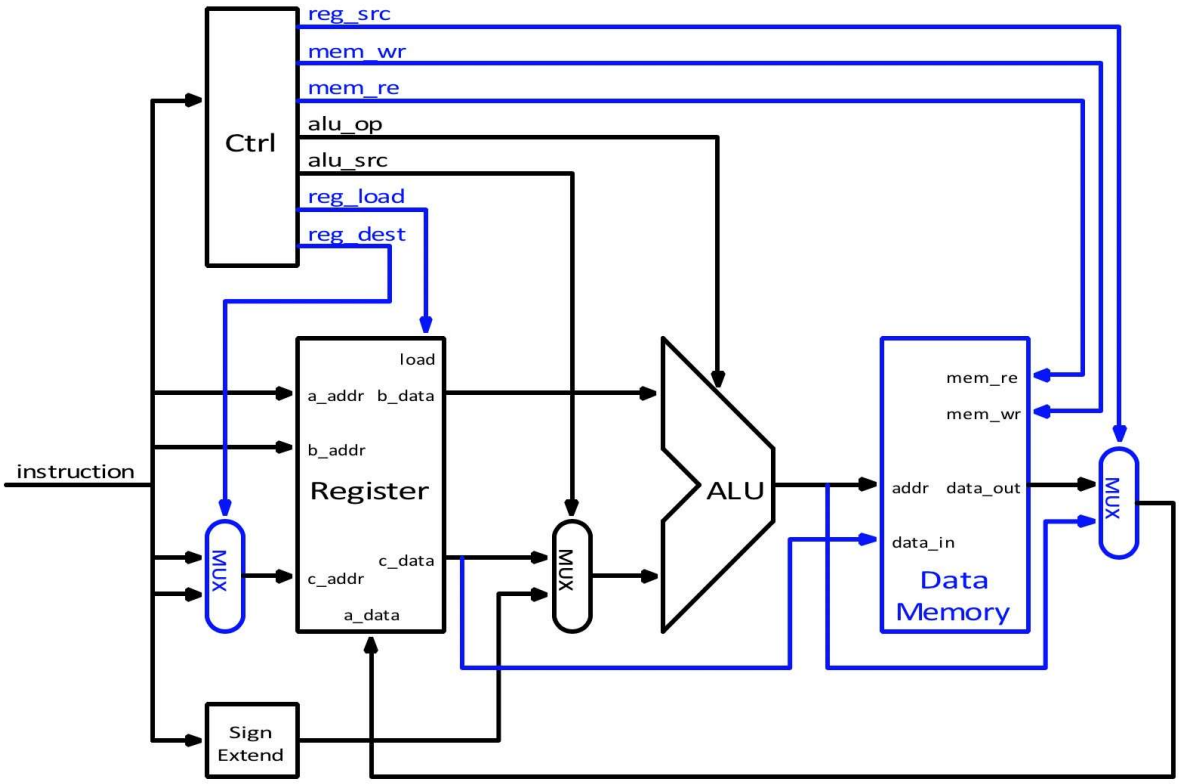
Based on the design that was developed in Lab 5, these next two figures should clearly highlight the components that will be implemented for this project:




A
S
B
16-bit ALU
Sout
mem_dump
mem_re
mem_wr
addr
data_out
data_in
Data Memory
S
Input1
Input2
Input3
Sout
3-1 MUX
slt_input <= "000000000000000"&alu_result(15);

Lab Execution

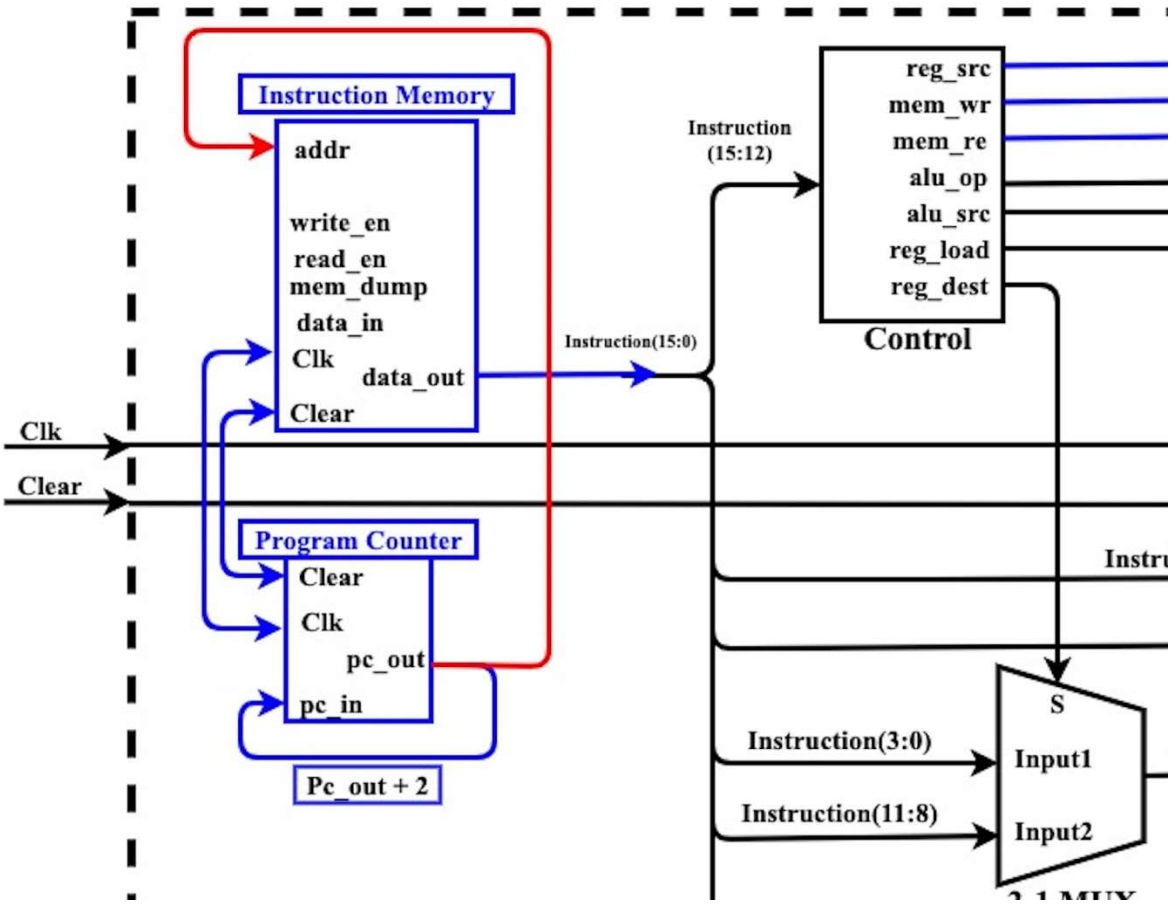
Data Memory Block: First, we will start with the implementation of the data memory block, the 3-1 MUX, and using the additional control signals that were added during Lab 5. The figure below should help you isolate what we are looking at, but note that the MUX used after the data memory block will be a 3-1 MUX, not a 2-1 MUX:



The data memory block takes in the output of the ALU into the addr (address) input, and the c_data output of the register file into the data_in signal. The mem_re input signal on the data memory will be mapped to the mem_re signal from the control block. Note that the predefined signal names in the VHDL files will differ. The mem_wr output signal of the control block will map to the mem_wr input of the data memory. Also note that the memory blocks have an input defined as "mem_dump" that isn't shown here; these values will be hardcoded with a '0' by default. The memory blocks will also be connected to a clock, the input clock of the top-level CPU design.

3-1 MUX: Next, we will implement the 3-1 MUX. Again, the figure directly above only depicts a 2-1 MUX, but the figures above that show a 3-1 MUX. We will implement the 3-1 MUX here. The output data of the data memory block, data_out, will be mapped to the first input of the 3-1 MUX. The second input of the 3-1 MUX will be the output of the ALU, the same signal that is connected to the addr signal in the data memory. The third signal is a little bit tricky; it will be called slt_input. This signal will only grab the 16th bit from the ALU result and append 15 zeros to it. The signal looks something like: "slt_input <= "0000000000000000"&alu_result(15);". This will be manually defined in the top-level CPU file. The selection signal for the MUX, S, will be controlled by the control.vhd file from the output signal reg_src. Lastly, the output of the 3-1 MUX, Sout, will be mapped back to the register file as the a_data input.

The figure below will highlight the next steps after this:



Program Counter: The program counter is very simple. Its inputs are a clock, reset, and an input from itself with a value of plus two from its output. Its output will feed into the address of the instruction memory block as an addr (address) signal, this allows the CPU to keep track of the instructions it executes. The output will also be fed back into the input of the program counter but adding two to the outputs original value. This simple module acts as a register that stores the value of its input (PC + 2) on the rising edge of the clock. Basically, storing the value here simply means that at the rising edge of the clock, the input signal is going to be placed on the output line. Also, this module should have a reset signal which helps to reset the PC value to zero whenever it is asserted low. This module does not increment the value of the counter; it merely stores it. You will implement a way to increment the value in the CPU top-level entity. The PC value is incremented by 2 every clock cycle (and not by 4 as in general cases) because each instruction has been encoded inside the instruction file on 2 bytes (16 bits). Each line of the instruction file holds a half-instruction, starting with the Least Significant Byte (LSB).

Instruction Memory: This memory is implemented in the same way as Data memory (it is basically the same component). The only difference is that we will always only read (mem_re signal is always going to be 1) from instruction memory and never write to it. Instruction Memory will initialize from a file called Instr.txt (provided in the Starter Project) which must be included in the same directory as the Memory.vhd file. This instruction file contains all the instructions to run (so that they do not need to be hardcoded in a testbench).

Testing the Design: Your testbench file for the simulation will only reset off and on the whole system. Input stimuli will be provided through the instruction file. To test your design, the instructions listed in the table below will be used.

Instruction	op	r_d	r_s	r_t/immediate	Value (r_d)
ADDI R3, R0, 5					
ADDI R4, R0, 2					
SLT R11, R3,R4					
SW R3, 0(R0)					
SW R4, 4(R0)					
ADDI R6, R0, 4					
LW R7, 0(R6)					
LW R8, 0(R0)					
ADD R9, R7, R8					
SLT R10, R0,R1					
SLT R10, R1,R0					
OR R5, R10, R9					
SUBI R10, R5, 7					
SUB R11, R10, R7					
SW R11, 5(R8)					

Part Two

In this part you need to design a CPU that performs all the functions introduced in the following table. Use the knowledge that you achieved during this course and the labs and design a CPU that performs all the functions of CPU of part one and also it performs the two last functions of the following table, i.e. BNE and Jump.

Instruction	Description	Opcode bits 15:12	Rd bits 11:8	Rs bits 7:4	Rt bits 3:0
ADD Rd, Rs, Rt	Rd := Rs + Rt	0	0-15	0-15	0-15
ADDI Rd, Rs, Imm	Rd := Rs + SignExt(Imm)	4	0-15	0-15	Imm
SUB Rd, Rs, Rt	Rd := Rs - Rt	1	0-15	0-15	0-15
SUBI Rd, Rs, Imm	Rd := Rs - SignExt(Imm)	5	0-15	0-15	Imm
AND Rd, Rs, Rt	Rd := Rs and Rt	2	0-15	0-15	0-15
OR Rd, Rs, Rt	Rd := Rs or Rt	3	0-15	0-15	0-15
SLT Rd, Rs, Rt	if Rs < Rt then Rd := 1 else Rd := 0	7	0-15	0-15	0-15
LW Rd, off(Rs)	Rd := M[off + Rs]	8	0-15	0-15	offset
SW Rd, off(Rs)	M[off + Rs] := Rd	C	0-15	0-15	offset
BNE Rd, Rs, Imm	if Rd != Rs then pc := pc + 2 + addr [1]	9	0-15	0-15	offset
JMP Address	pc := JumpAddress [2]	B	12-bit offset		

For BNE function, the address of the next instruction will be the output of the PC plus the value that has been defined as “Immediate” value.

For JMP function, the address of the next instruction will be the concatenation result of the four most significant bits of the output of the PC and the 12-bit “jump address”.

For this part, the block diagram has not been provided. You need to build the block diagram. You can use the 8-bit ALU that you designed in Lab4 and modify it to a 16-bit ALU. You can reuse any VHDL codes that you had in the labs. You can add any extra blocks that you may need for building a CPU with all functions introduced in the above-mentioned table.

Design a testbench that can verify all the functions introduced in the above-mentioned table.

Report and Demonstration:

During the demonstration the functionality of the design of part one and part two will be verified. You need to have complete testbenches for verifying all functions of the CPU’s.

For the report you need to follow the general guidance for the lab reports

Waveform: Your waveform needs to include the **instruction memory output**, the **register** bus, and the **data mem** bus. These signals are called data_out, reg, and mem, respectively. data_out is found inside of CPU_Instr_MEM. reg is found inside of CPU_Registers_0. mem is found inside of CPU_MEM_0.

To find these signals, make sure your waveform window, ISim, is using the default layout by selecting Layout -> Restore Default Layout. In the far-left window, expand cpu_3380_test, then expand uut. Selecting uut will show the signals defined inside that module. Under the uut will be a list of all the modules uut uses which includes the instruction memory, registers, and data mem. Select each of those to see their objects/signals contained within those modules.

Add the instruction memory output first, then registers, then data mem. Make sure data_out is collapsed. Make sure reg and mem are expanded. Set data_out to hexadecimal radix. Set reg and all the child signals, and mem and all the child signals to signed decimal. Take the screenshot such that the entire name column, value column, and the waveform itself are visible. Include reset, clock, mem_dump, tick, run_time, data_out, reg, and mem in the screenshot. You do not need to take multiple pictures to include beyond MEM 11.

Ensure that the waveform simulates for the minimum necessary time to display all the above instructions and their results.

This project will require a project report in which you should be able to describe the entire CPU and all of the signals. You need to explain your design in detail and include a section about all difficulties you faced in your design and obstacles that you had to overcome. 25% (twenty five percent) of the project mark is for this discussion.

Please note that part one and part two have one third (1/3) and two third (2/3) of the project mark respectively.