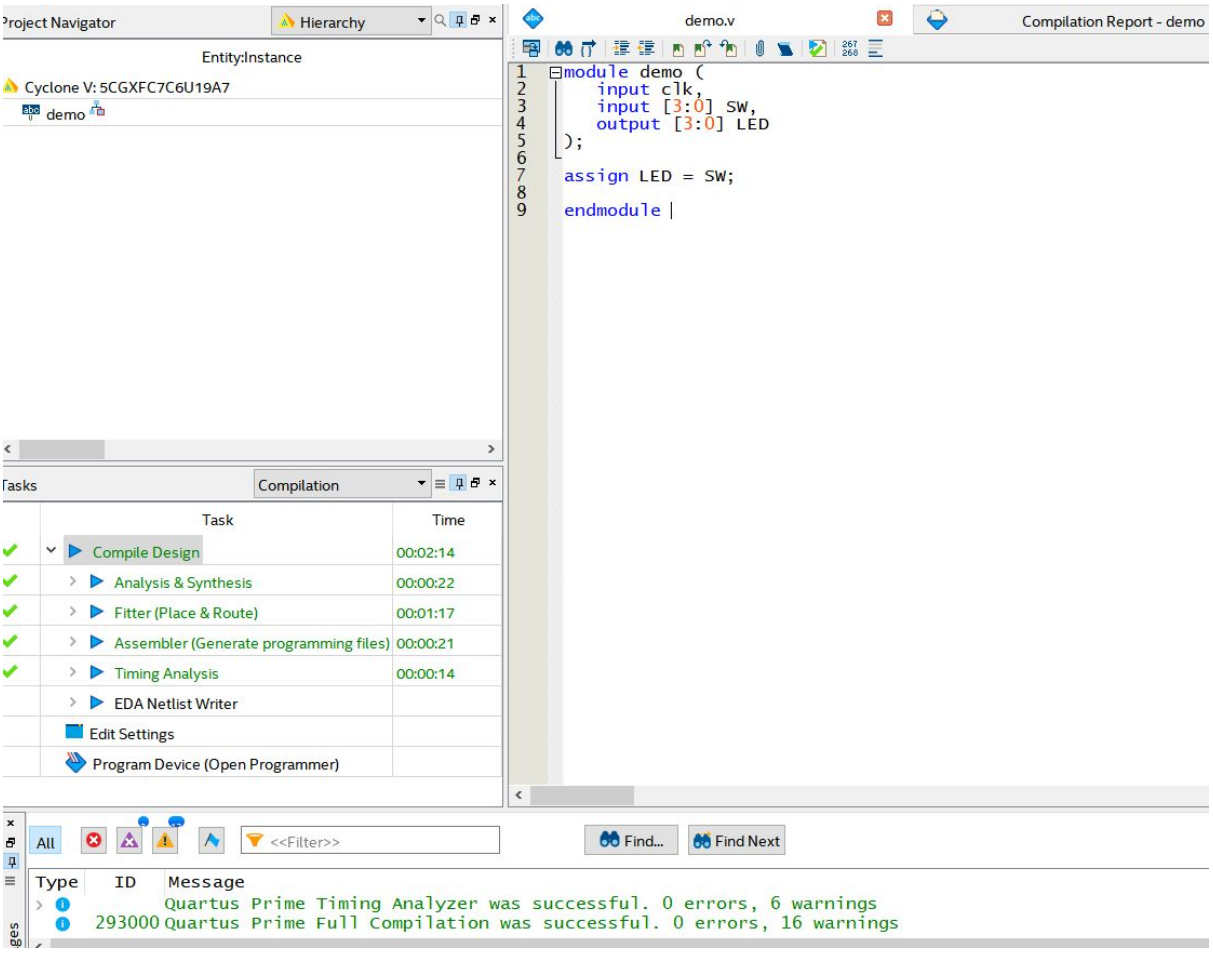
Group R

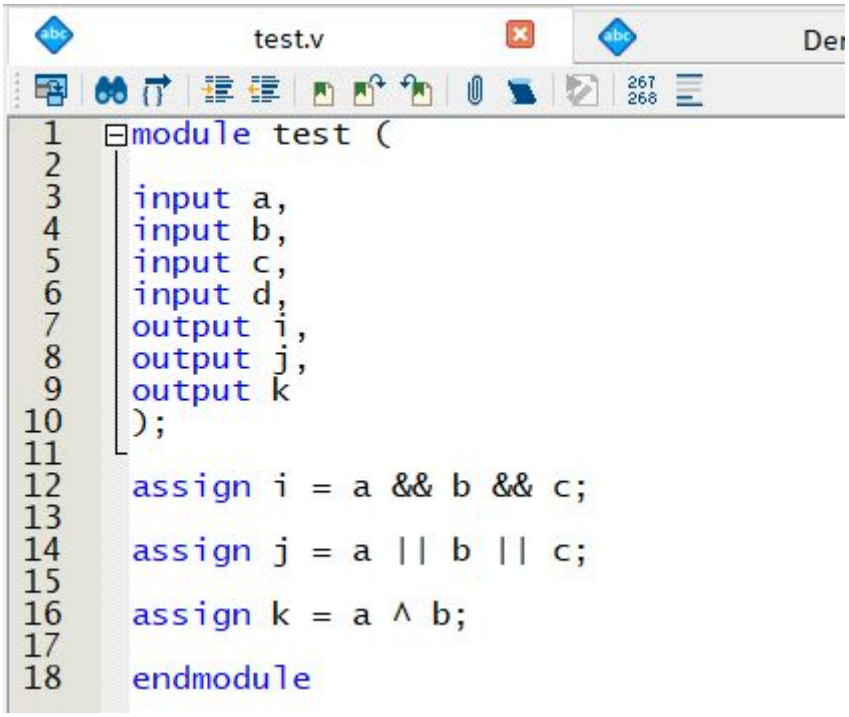
Project Milestone 1:

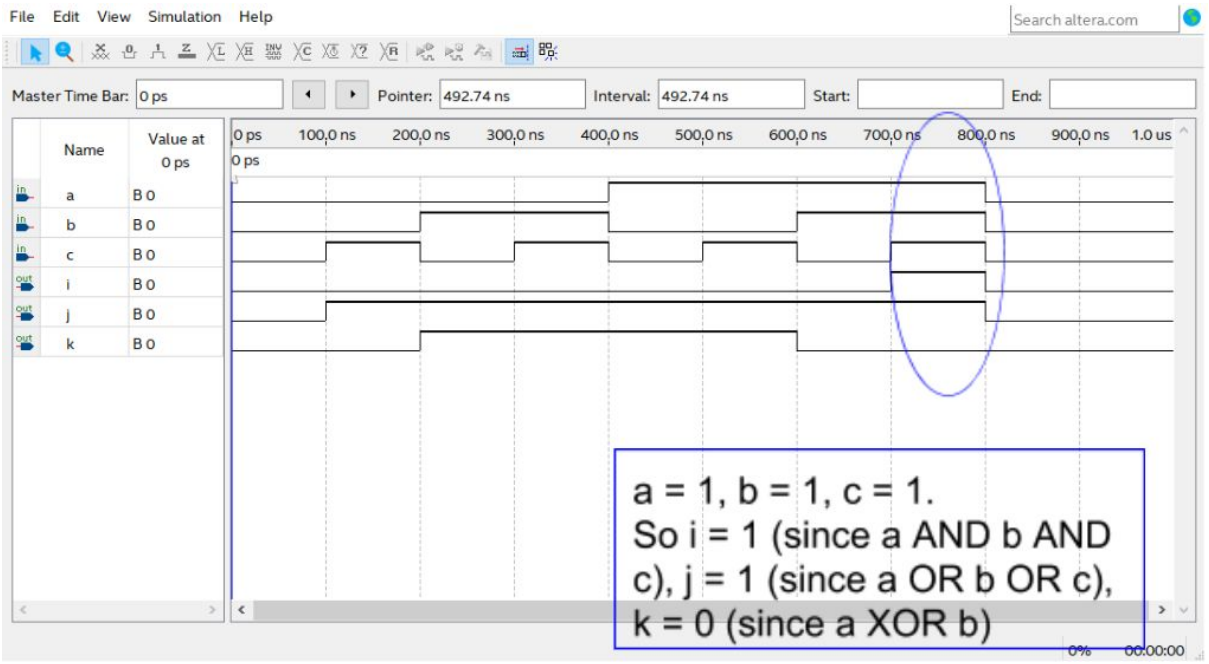
Simulation of RISC CPU using Verilog Hardware Design

Part A.

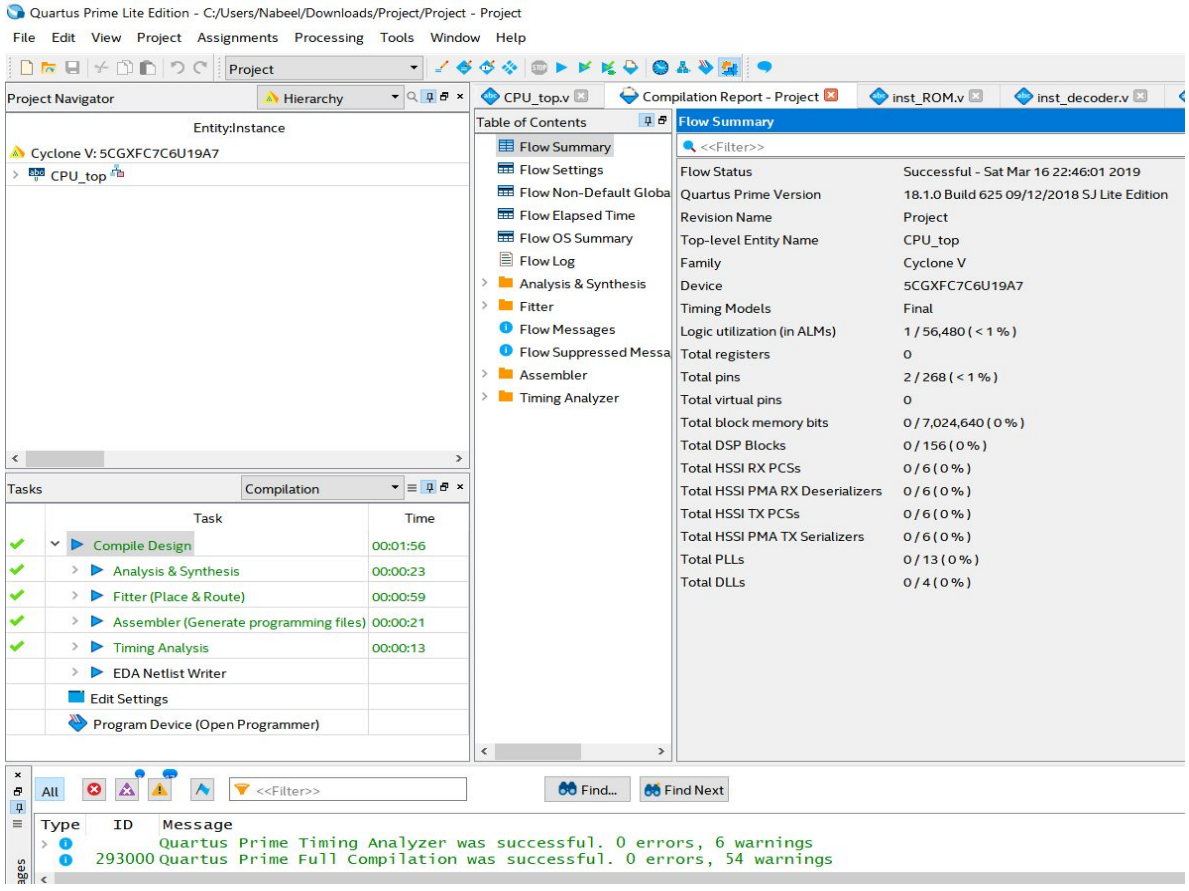

Project Navigator
Hierarchy
Entity:Instance
Cyclone V: 5CGXFC7C6U19A7
demo
demo.v
Compilation Report - demo
module demo (
input clk,
input [3:0] SW,
output [3:0] LED
);
assign LED = SW;
endmodule
Tasks
Compilation
Task
Time
Compile Design 00:02:14
Analysis & Synthesis 00:00:22
Fitter (Place & Route) 00:01:17
Assembler (Generate programming files) 00:00:21
Timing Analysis 00:00:14
EDA Netlist Writer
Edit Settings
Program Device (Open Programmer)
All
<<Filter>>
Find...
Find Next
Type ID Message
Quartus Prime Timing Analyzer was successful. 0 errors, 6 warnings
293000 Quartus Prime Full Compilation was successful. 0 errors, 16 warnings

Part B.


test.v
module test (
input a,
input b,
input c,
input d,
output i,
output j,
output k
);
assign i = a && b && c;
assign j = a || b || c;
assign k = a ^ b;
endmodule


File Edit View Simulation Help
Search altera.com
Master Time Bar: 0 ps
Pointer: 492.74 ns
Interval: 492.74 ns
Start:
End:
Name
Value at 0 ps
a
b
c
i
j
k
0 ps
100.0 ns
200.0 ns
300.0 ns
400.0 ns
500.0 ns
600.0 ns
700.0 ns
800.0 ns
900.0 ns
1.0 us
a = 1, b = 1, c = 1.
So i = 1 (since a AND b AND c), j = 1 (since a OR b OR c), k = 0 (since a XOR b)

Part C.


Quartus Prime Lite Edition - C:/Users/Nabeel/Downloads/Project/Project - Project
File Edit View Project Assignments Processing Tools Window Help
Project Navigator
Hierarchy
Entity:Instance
Cyclone V: 5CGXFC7C6U19A7
CPU_top
Tasks
Compilation
Task
Time
Compile Design 00:01:56
Analysis & Synthesis 00:00:23
Fitter (Place & Route) 00:00:59
Assembler (Generate programming files) 00:00:21
Timing Analysis 00:00:13
EDA Netlist Writer
Edit Settings
Program Device (Open Programmer)
CPU_top.v
Compilation Report - Project
inst_ROM.v
inst_decoder.v
Table of Contents
Flow Summary
Flow Settings
Flow Non-Default Globa
Flow Elapsed Time
Flow OS Summary
Flow Log
Analysis & Synthesis
Fitter
Flow Messages
Flow Suppressed Messa
Assembler
Timing Analyzer
Flow Summary
Flow Status Successful - Sat Mar 16 22:46:01 2019
Quartus Prime Version 18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name Project
Top-level Entity Name CPU_top
Family Cyclone V
Device 5CGXFC7C6U19A7
Timing Models Final
Logic utilization (in ALMs) 1 / 56,480 (< 1 %)
Total registers 0
Total pins 2 / 268 (< 1 %)
Total virtual pins 0
Total block memory bits 0 / 7,024,640 (0 %)
Total DSP Blocks 0 / 156 (0 %)
Total HSSI RX PCSs 0 / 6 (0 %)
Total HSSI PMA RX Deserializers 0 / 6 (0 %)
Total HSSI TX PCSs 0 / 6 (0 %)
Total HSSI PMA TX Serializers 0 / 6 (0 %)
Total PLLs 0 / 13 (0 %)
Total DLLs 0 / 4 (0 %)
All
Find...
Find Next
Type ID Message
Quartus Prime Timing Analyzer was successful. 0 errors, 6 warnings
293000 Quartus Prime Full Compilation was successful. 0 errors, 54 warnings

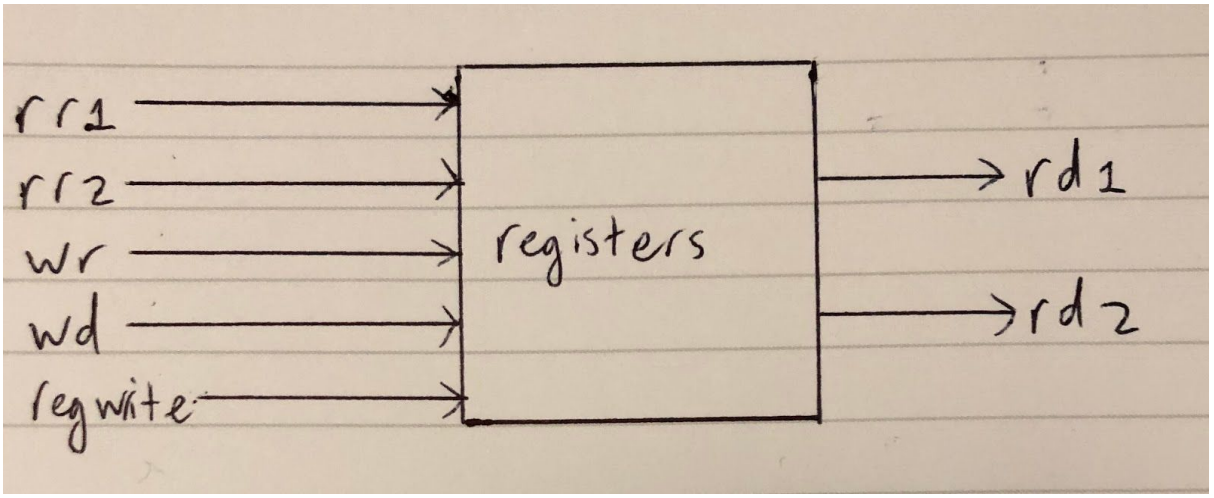
Part D.

Part A: Overview

This whole project is a RISC processor, made up of 8 modules. There are two inputs (clk and rst). The modules that make up this processor are pc, inst_ROM, inst_decoder, data_RAM, CPU_top, registers, control and ALU. The CPU carries out the instructions of a computer program by performing arithmetic, logic, control, and input/output. A more detailed overview of how the system works continues in Section E.

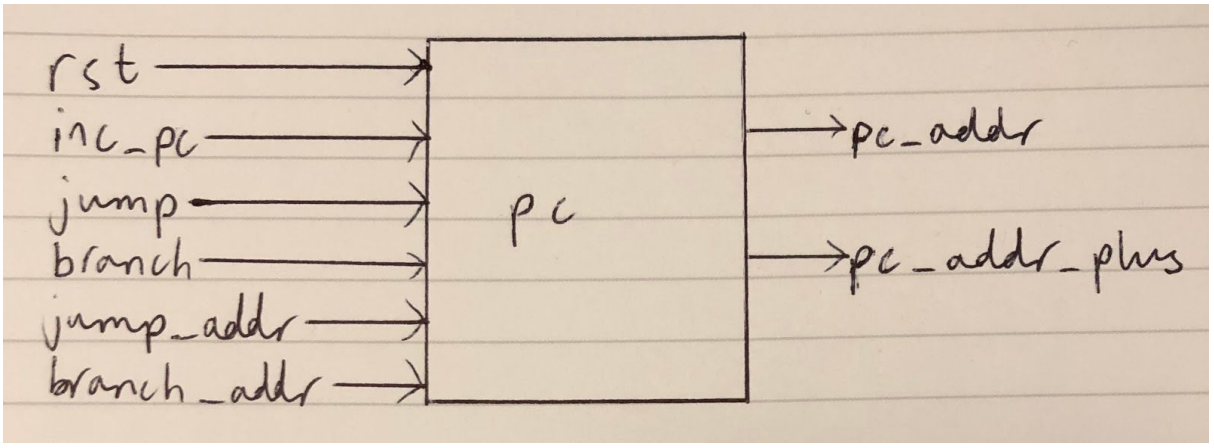
Part B: Each Module

Registers



The register module has 5 inputs and 2 outputs. The input regwrite triggers either the read or write operations (read is faster than write in the register). The register is a fast access storage location for the CPU.

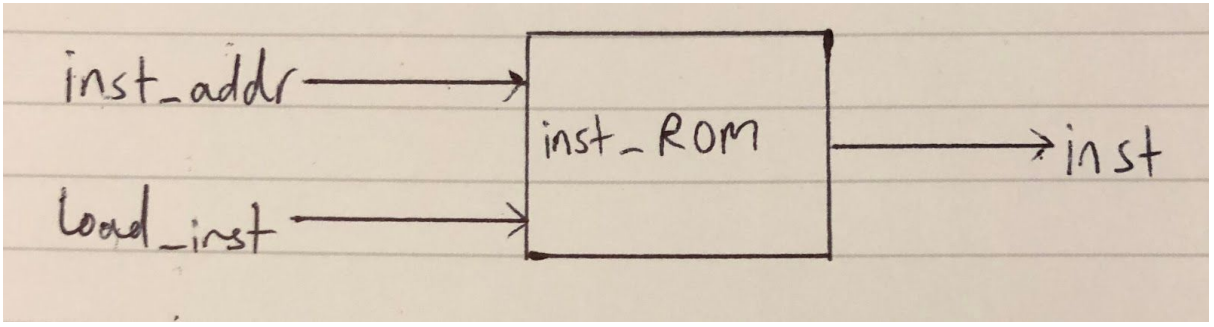
PC



The PC module has 6 inputs and 2 outputs. The output is generated based on the input. If rst is 1, the pc_address is reset to zero. If a jump instruction is used, then the pc_addr is set to the jump target address. However if its a branch instruction, then the pc_addr is set to the target

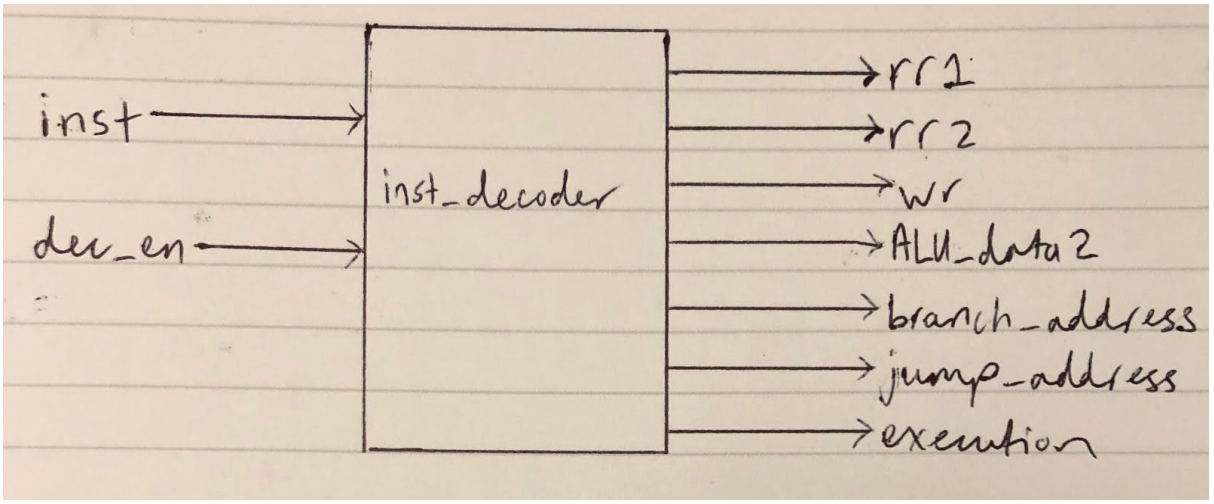
branch_addr. A normal instruction (i.e. 0 for branch and jump) will result in neither jump or branch. The PC module sets the pc_addr to either the jump_addr, branch_addr, or pc_addr_plus depending on the binary values of the jump and branch inputs.

Inst ROM



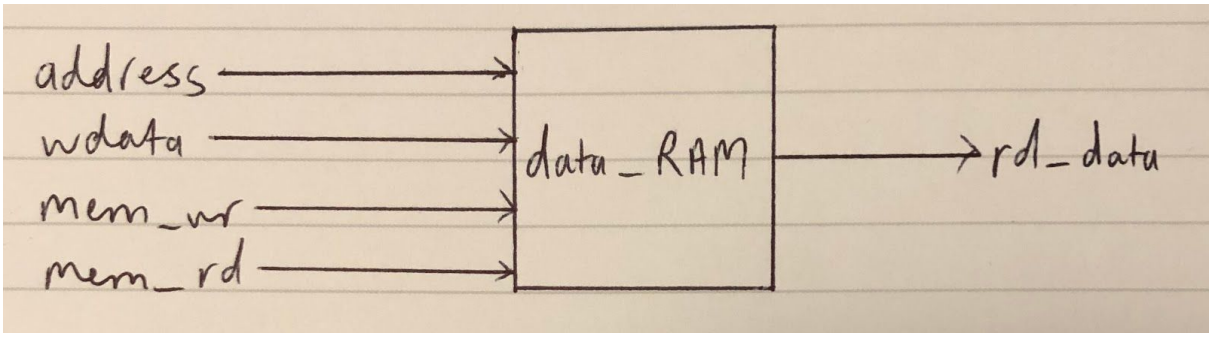
The PC module has 2 inputs and 1 output. It is a ROM because it is read-only instruction memory. If a load_inst is received, set the output inst to the inst_addr. This inst_addr instruction comes the the PC module before it; which was the output pc_addr.

Inst Decoder



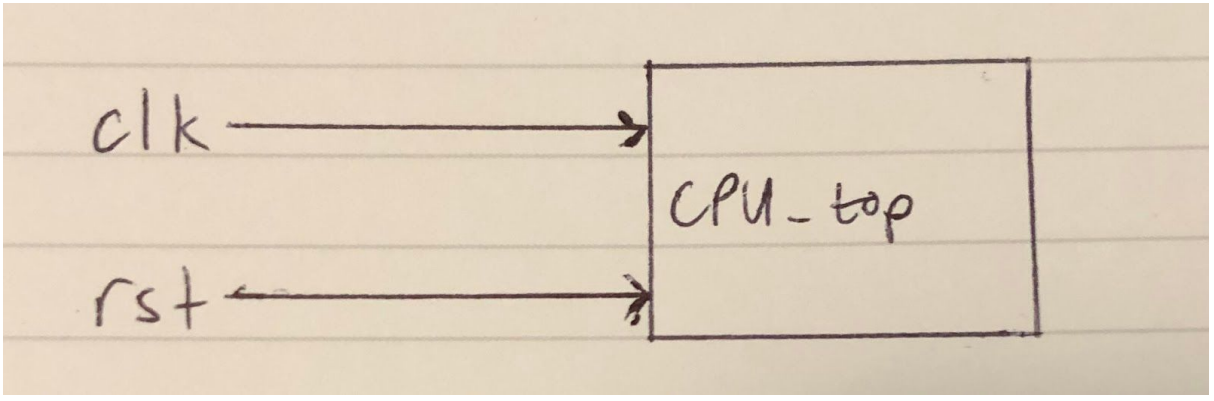
The inst_decoder module has 2 inputs and 7 outputs. It receives 1 instruction from the inst_ROM and 1 from the control. It decodes these instructions and supplies the register module with 3 of its inputs. Depending on what the instruction is each of the outputs will be set to a binary number for its next destination. The input (inst) can be I1, I2, S1, S2, R, UJ, or SH; each of these are represented by a binary number. The outputs are set to a certain instructions (i.e. BEQ (branch if equal to), ADD, SUB, XOR, ect.). The inst input is used to determine which RISC instruction will take place, once decoded. These instructions will be used as input in other modules such as the control module.

Data RAM



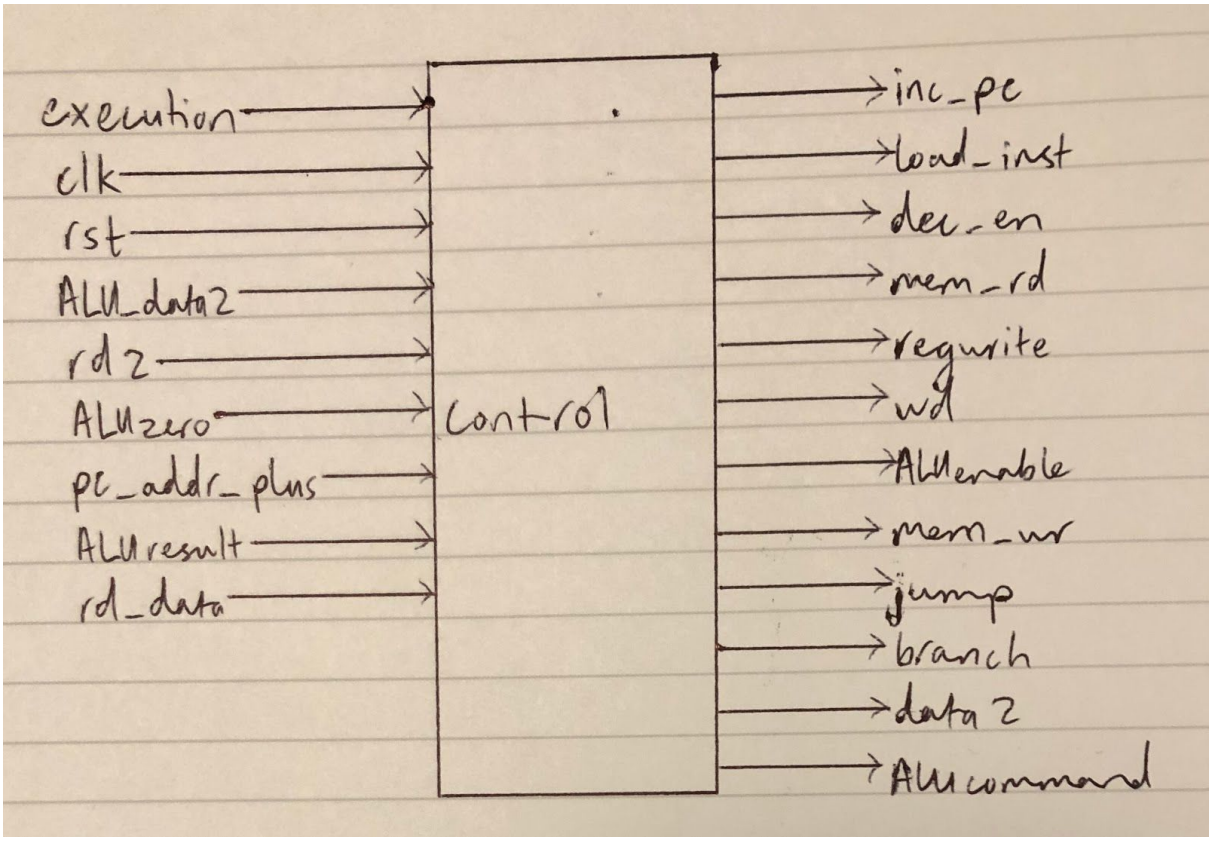
The data_RAM module has 4 inputs and 1 output. This module executes the read and write operation. This module gets its inputs (instructions) from the control module (wdata from the register/control module) and sends its output back to the control module. If the mem_rd input is a binary number 1 the output is set to the instruction that was input from the address input.

CPU top



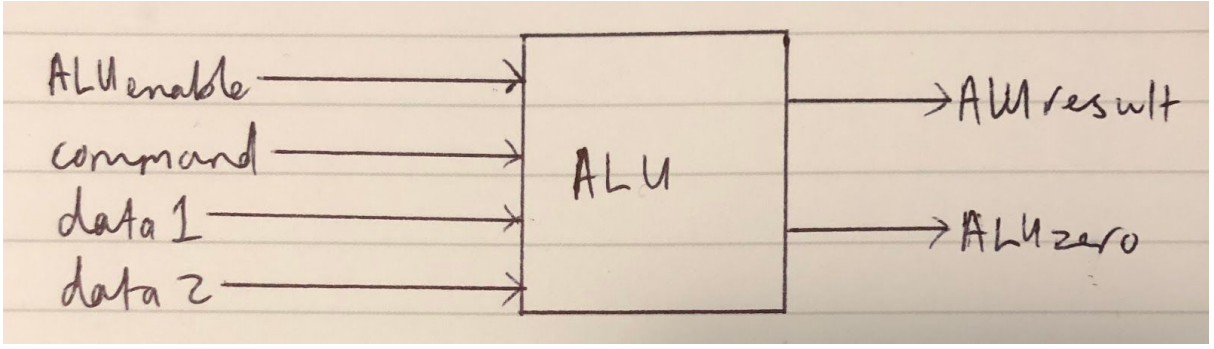
The CPU_top is the main component that contains the other modules. It only has 2 inputs and no outputs. The inputs are clock (clk) and reset (rst). This component creates all the wires used and assigns the wires (by name) to each module (i.e. the inst_ROM called rom1 gets the inst_addr, load_inst and inst wires). The same is done for the other module's wires.

Control



The control module has 9 inputs and 12 outputs. Based on an input instruction the control module can send a signal (an input) to the ALU, when necessary, which can then perform an arithmetic/logic operation on data1 and data2. Each execution can only take place once the next clock cycle starts. This module directs the operation of the CPU, deciding what to do/where to send instructions.

ALU



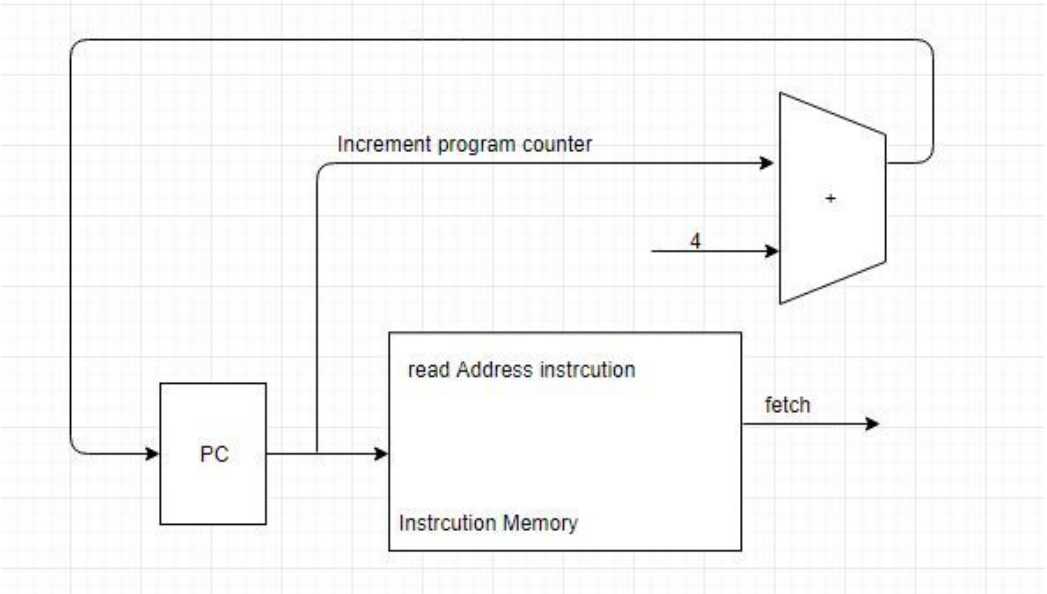
This ALU (Arithmetic Logic Unit) module's main function is to perform arithmetic and logic operations. It has 4 inputs and 2 outputs. The ALUenable input needs to be on/enabled (binary 1) for the following operations to take place. A posedge needs to be encountered for ALUenable to perform an operation or change to perform a different operation. The operations the ALU can perform, in this module, are SUB, ADD, SL (shift left), XOR (exclusive OR), and OR. The result of these operations on the inputs data1 and data2 are

output in ALUresult. If the ALUresult equals 32'h00000000, then the ALUzero is set to 1'b1 (binary 1), otherwise it is set to binary 0. The ALUzero and ALUresult will be sent to the control module but data_RAM will also get the ALUresult as one of its inputs.

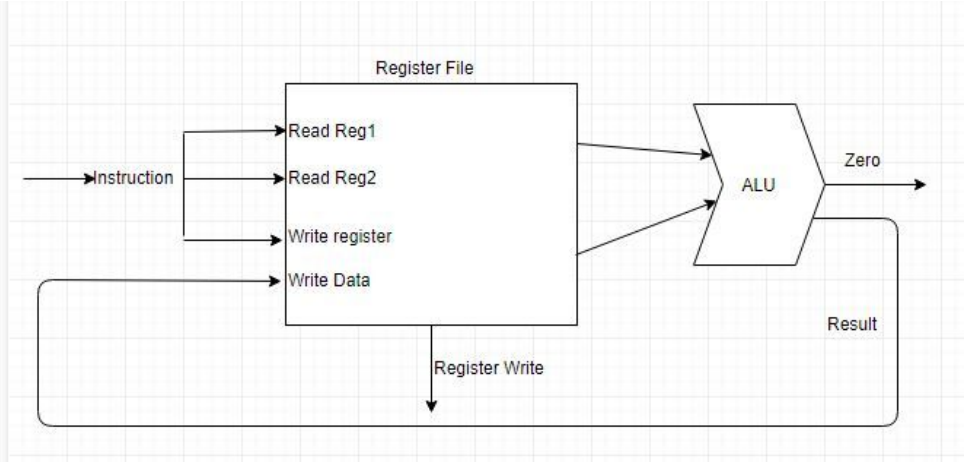
Part E.

In this section we see how this system works.

All starts with CPU_top module it is working as main method and call other module for execution.
All instructions start by Program counter (PC) to supply the instruction address to the instruction memory (instructions are read from rom.txt)

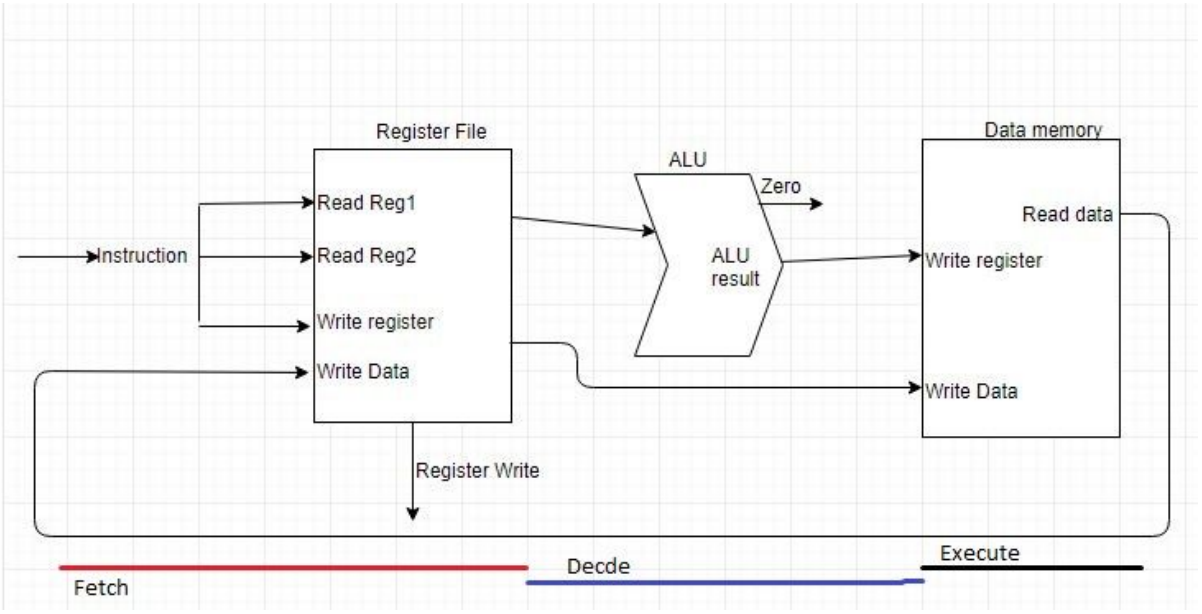


Implementation of the data path for R-format instruction is fairly straightforward the register file and the ALU are all that is required the ALU accepts its input from the DataRead ports of the register file and the register file is written into by the ALUresult output of the ALU in combination with the RegWrite signal.



For load /store, datapath uses those instructions from memory and writes into registers. Register access takes input from the register file, to implement the instruction, data, or address fetch step of the fetch-decode-execute cycle.
Instant decoder decodes the base address and offset and offset, combining them to produce the actual memory address.

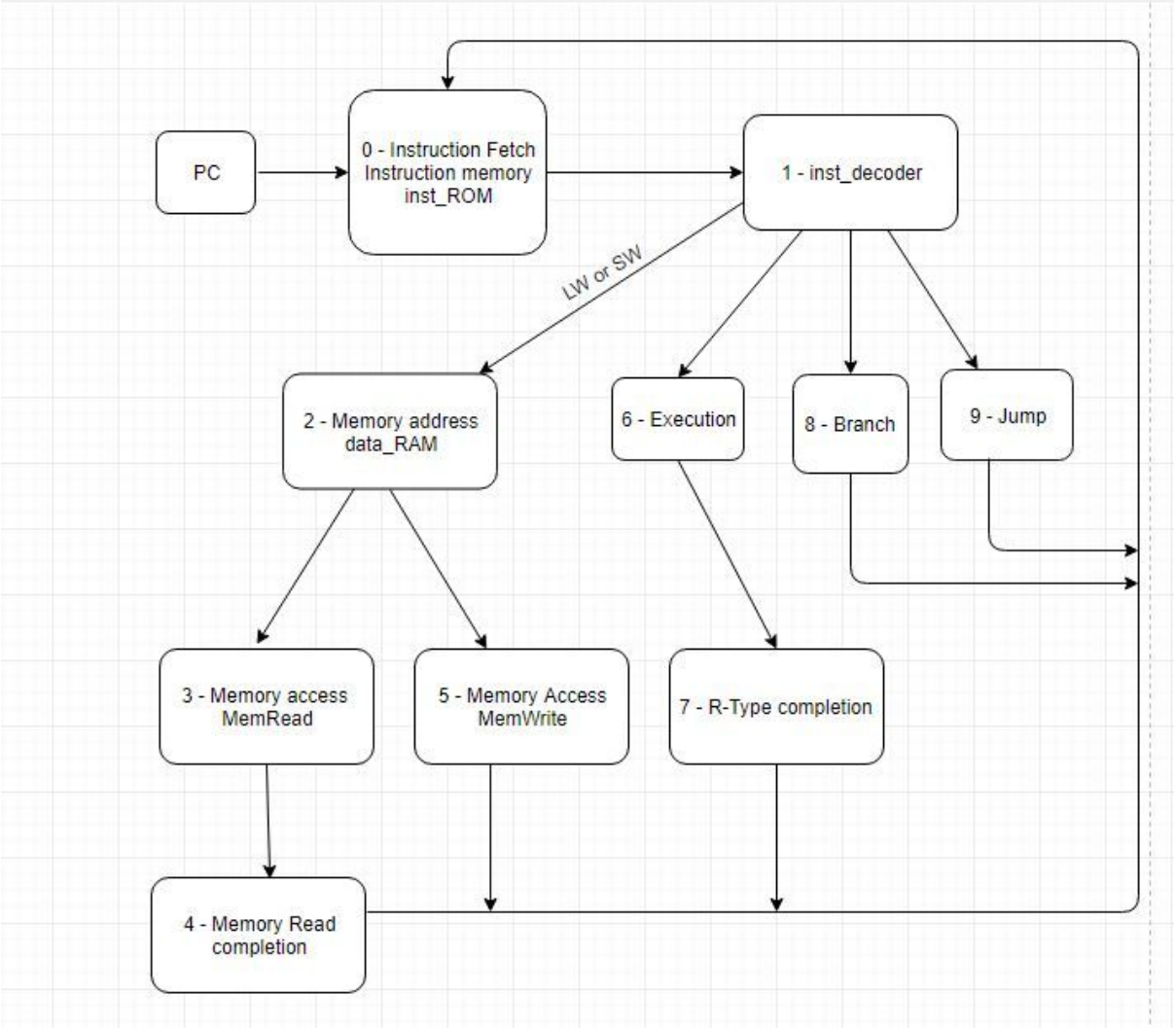
Read/Write from memory takes data instruction from the data memory, and implements the first part of the execute step of the fetch/decode/execute cycle.
Write into Register File puts data or instruction into the data memory, implementing the second part of the execute step of the fetch/decode/execute cycle.



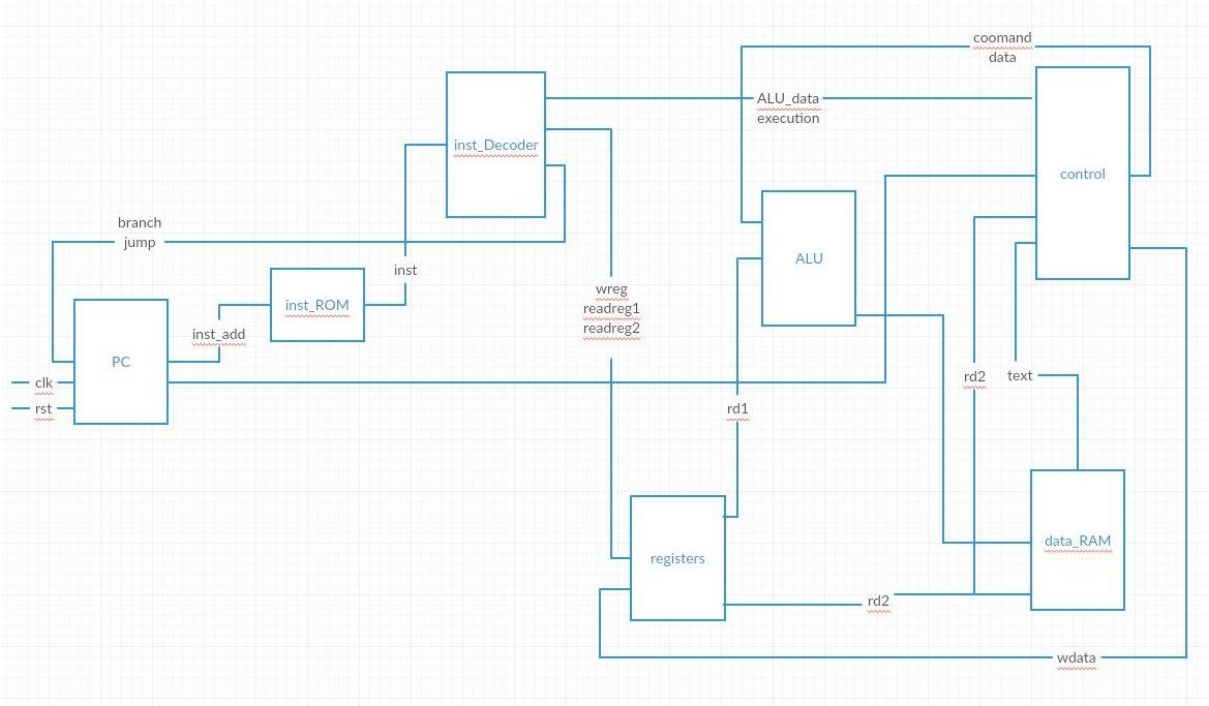
For branch/jump Register access takes input from the register file to implement the instruction fetch or data fetch step of the fetch-decode-execute cycle. Concurrent with ALU first evaluate of the branch condition, second calculates the branch target address, to be ready for the branch if it is taken. This completes the decode step of the fetch-decode-execute cycle.

Evaluate branch condition and jump to branch target address or PC+4 uses ALU first determine whether or not the branch should be taken jump to branch or PC+4 uses control logic hardware to transfer control to the instruction referenced by the branch target address. This effectively change the PC to the branch target address, and completes the execute step of the fetch-decode-execute cycle.

State Diagram



Module Diagram



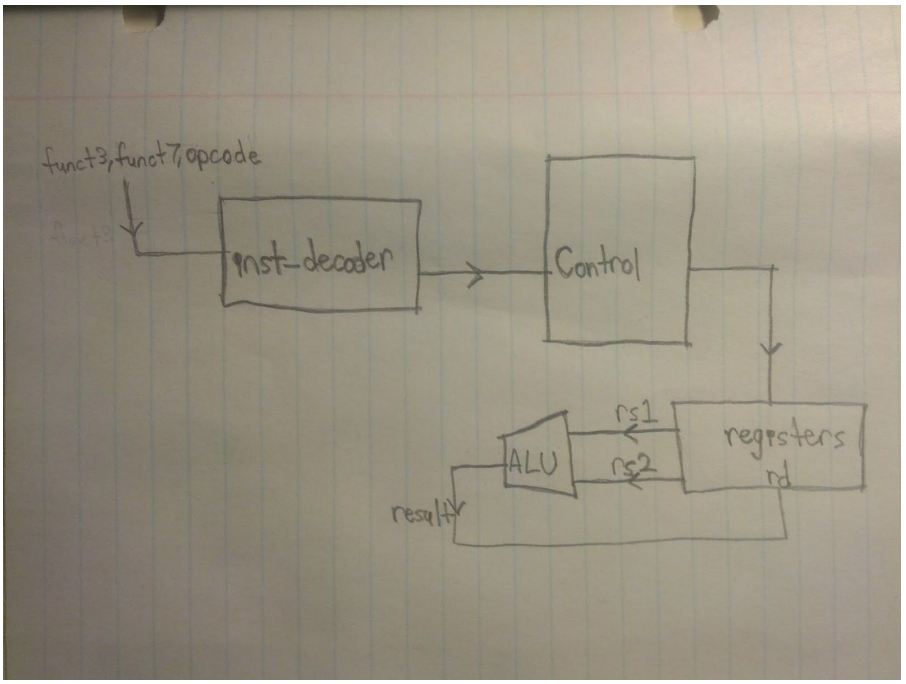
Part F.

ADD RD, RS2, RS1: This is the structure of the add instruction which adds the values in source register one(RS1) and source register two(RS2), and stores the sum in the destination register(RD). This add instruction is an R-format instruction.

funct7	rs2	rs1	funct3	rd	opcode

0000000	rs2	rs1	000	rd	0110011

The combination of funct7, funct3 and opcode are read by the instruction decoder to determine what type of instruction is meant to run. Then the cpu branches to the register module and reads the values stored in rs1 and rs2. Next, the values of rs1 and rs2 will be used as input for the ALU and added together. Lastly, their sum will be be written into the register designated by the the register rd.



LD RD,IMM(RS1): This is the structure of the load double word instruction which loads the value from the memory address to the destination register rd. The memory address mentioned is equal to the source register plus the offset (rs1 + imm).

immediate	rs1	funct3	rd	opcode
00000	rs1	011	rd	0000011

The combination of funct3 and opcode are read by the instruction decoder to determine what part of the cpu to branch to. Next, the register module read the value stored in source register one (rs1) and sends it to the ALU as the first input with the immediate value as the second input. The ALU's addition function would activate and return the sum . This summation would be used as input for the memory module and return the value at that memory address. Finally, the value at that memory address would be loaded into the destination register.

SUB (RD, RS2, RS1): The structure of the sub instruction in this cpu is done identically to how it's handled in risc. The RD is the register on which the value (RS2 - RS1) is stored, RD is stored in ALUresult_r as well as RS2 and RS1 being stored in data2 and data1, all of which are 32 bit registers. From CPU_top.v we go to inst_decoder which decodes the execution the user wants to perform then sends the correct inputs through registers rr1_r and rr2_r to control.v. It's important to note that data2 and data1 are both passed as inputs from CPU_top.v into ALU.v and CPU.v which does the arithmetic operations and stores the value into the correct register. In control.v for subtract, under the control case we are checking for BEQ and SUB executions as if the execution is branch when equal, we can find BEQ which is same with SUB in this stage. From control we go to the executing stage once the clock

comes, which executes the ALU when the command comes. We then go to writeback which selects the output of ALU result ram as the input of register writing, followed by the change_pc case which finally writes the result to the register file determined in
This SUB instruction is done under the R-type instruction format, which contains:

7-bit funct, 5-bit-rs2, 5-bit-rs1, 3-bit-funct, 5-bit rd, and a 7-bit opcode

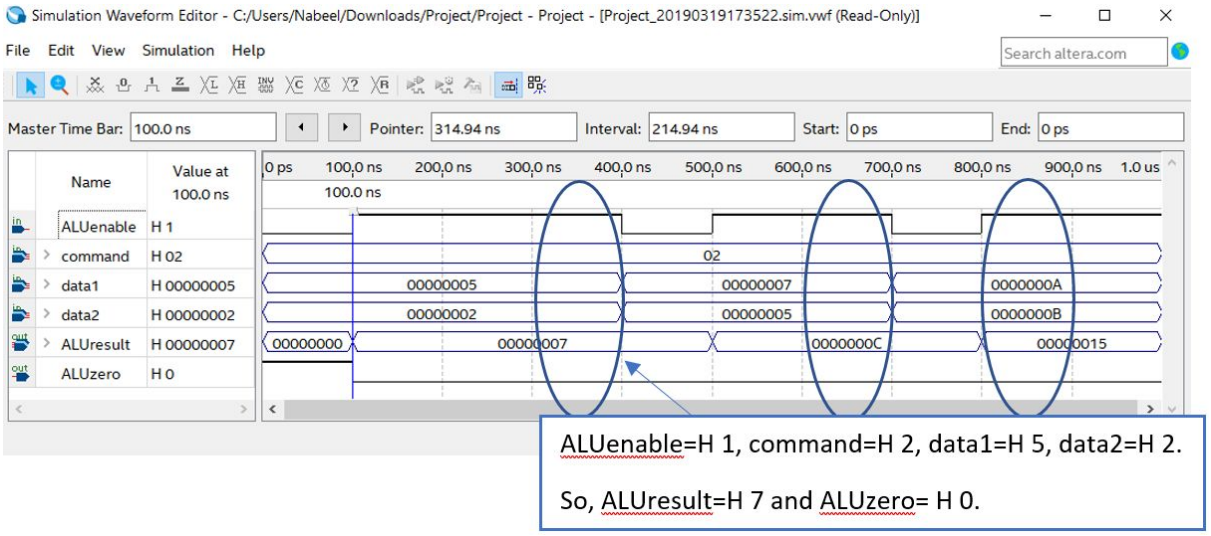
funct7	rs2	rs1	funct3	rd	opcode

BEQ (RS1, RS2, L): The structure of the beq instruction in this cpu is done identically to how it's handled in risc. From CPU_top.v we go to inst_decoder which decodes the execution the user wants to perform then sends the correct inputs through registers rr1_r and rr2_r to control.v. Both inputs are passed from CPU_top.v into ALU.v and CPU.v which does the arithmetic operations and decides whether or not to branch or not. In control.v under the control case we are checking for BEQ and SUB executions as if the execution is branch when equal, we can find BEQ which is same with SUB in this stage. From control we go to the executing stage once the clock comes, which executes the ALU when the command comes. We then go to writeback which uses the line:
op_reg[1]<=(ALUzero==1'b1)?1'b1:1'b0;
In order to check if the rs1 and rs2 values are the same then we set branch to l.
This BEQ instruction is done under the b-type instruction format.

Part G.

ALU

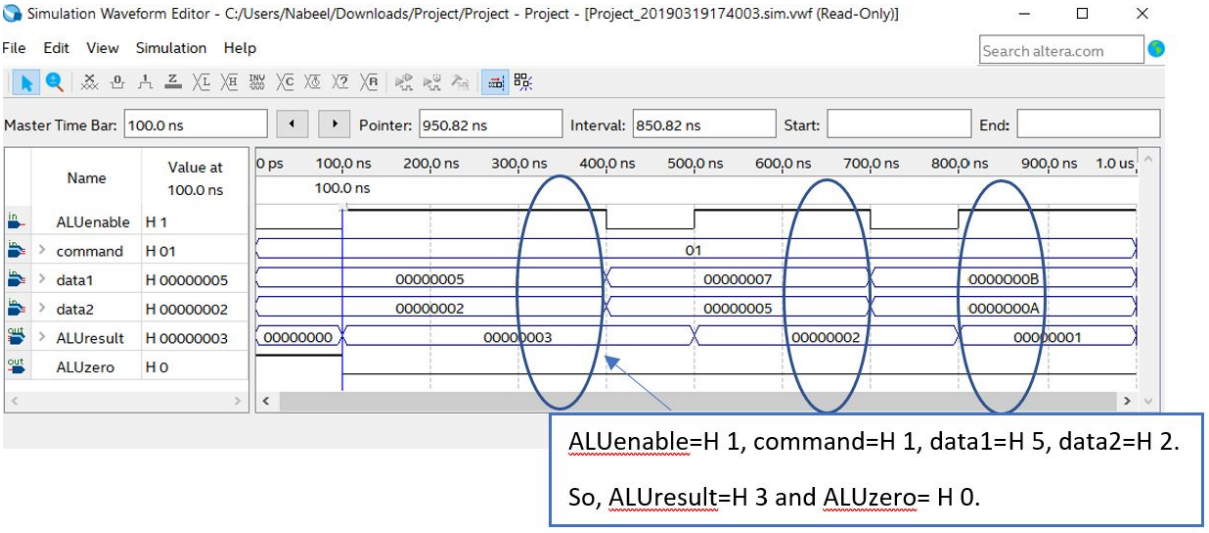
ALU ADD



Input/Output	Hexadecimal	Binary
ALUenable (input)	0 1	0 1
command (input)	2	10
data1 (input)	5 7 A	101 111 1010
data2 (input)	2 5 B	10 101 1011
ALUresult (output)	7 C 15	111 1100 10101
ALUzero (output)	0	0

These inputs and outputs show that the ALU ADD command works correctly; as H 5 + H 2 = H 7 and H 7 + H 5 = H C as expected.

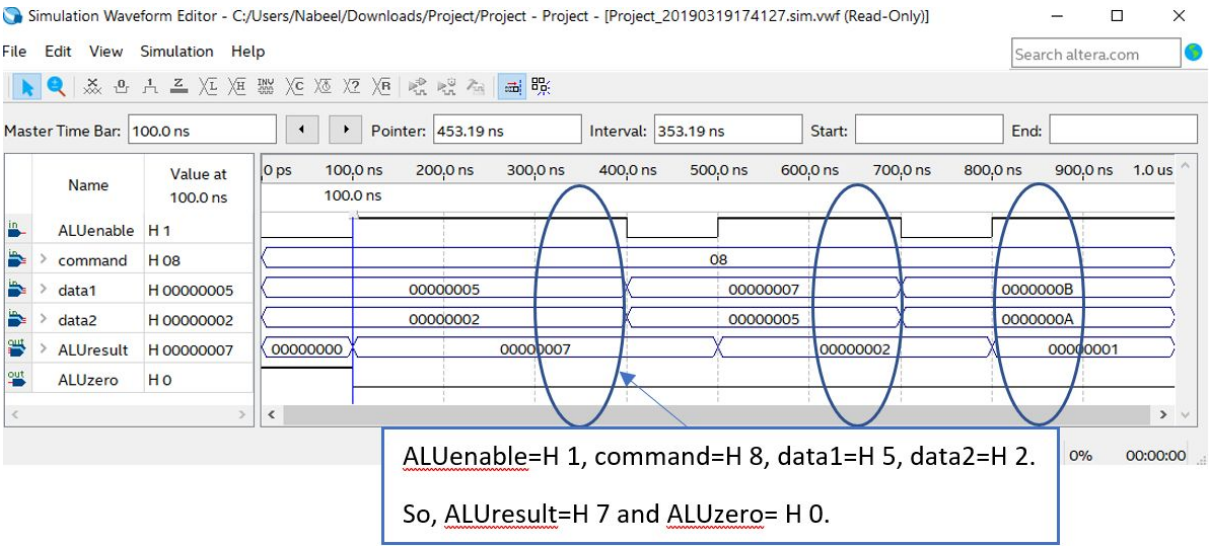
ALU SUB



Input/Output	Hexadecimal	Binary
ALUenable (input)	0 1	0 1
command (input)	1	1
data1 (input)	5 7 B	101 111 1011
data2 (input)	2 5 A	10 101 1010
ALUresult (output)	3 2 1	11 10 1
ALUzero (output)	0	0

These inputs and outputs show that the ALU SUB command works correctly; as H 5 - H 2 = H 3 and H 7 - H 5 = H 2 as expected.

ALU XOR

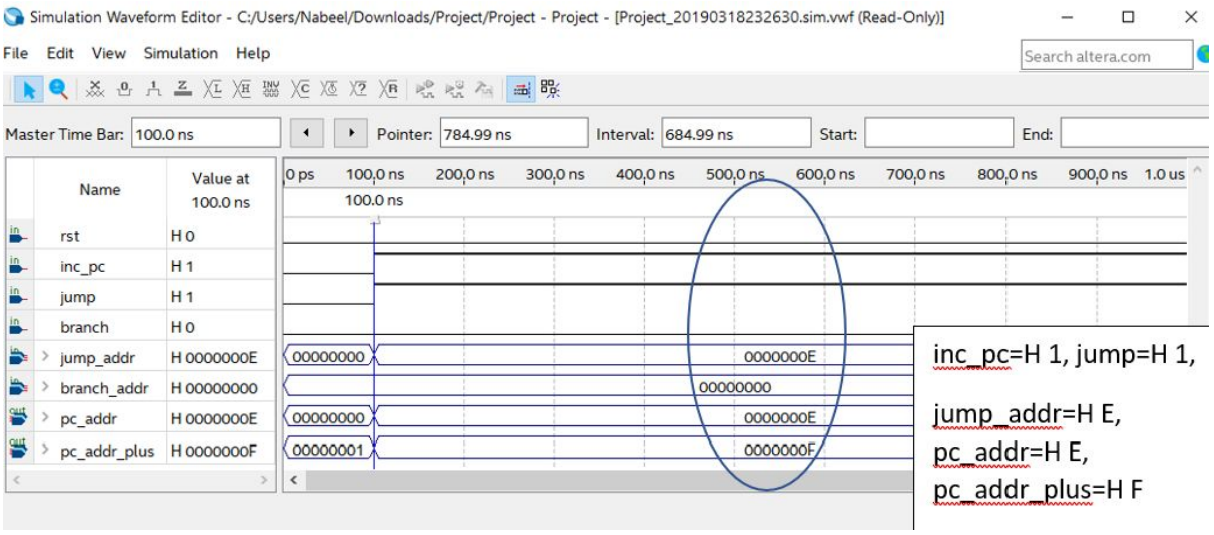


Input/Output	Hexadecimal	Binary
ALUenable (input)	0 1	0 1
command (input)	8	1000
data1 (input)	5 7 B	101 111 1011
data2 (input)	2 5 A	10 101 1010
ALUresult (output)	7 2 1	111 10 1
ALUzero (output)	0	0

These inputs and outputs show that the ALU XOR command works correctly; as H 5 ^ H 2 = H 7 and H B ^ H A = H 1 as expected.

PC

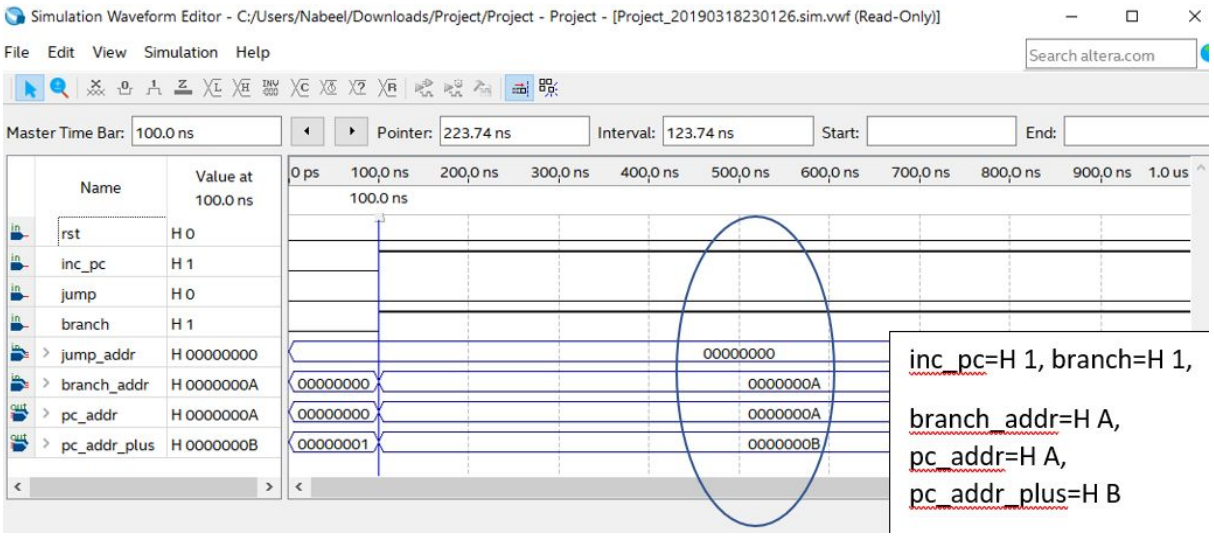
jump



Input/Output	Hexadecimal	Binary
inc_pc (input)	1	1
jump (input)	1	1
jump_addr	E	1110
pc_addr	E	1110
pc_addr_plus	F	1111

When inc_pc is set to H 1, rst does not equal H 1 and jump equals H 1; then pc_addr is set to the jump_addr which is H E and and pc_addr_plus is incremented by 1 (to H F). And this is exactly what occurs when these numbers are input into the waveform generator.

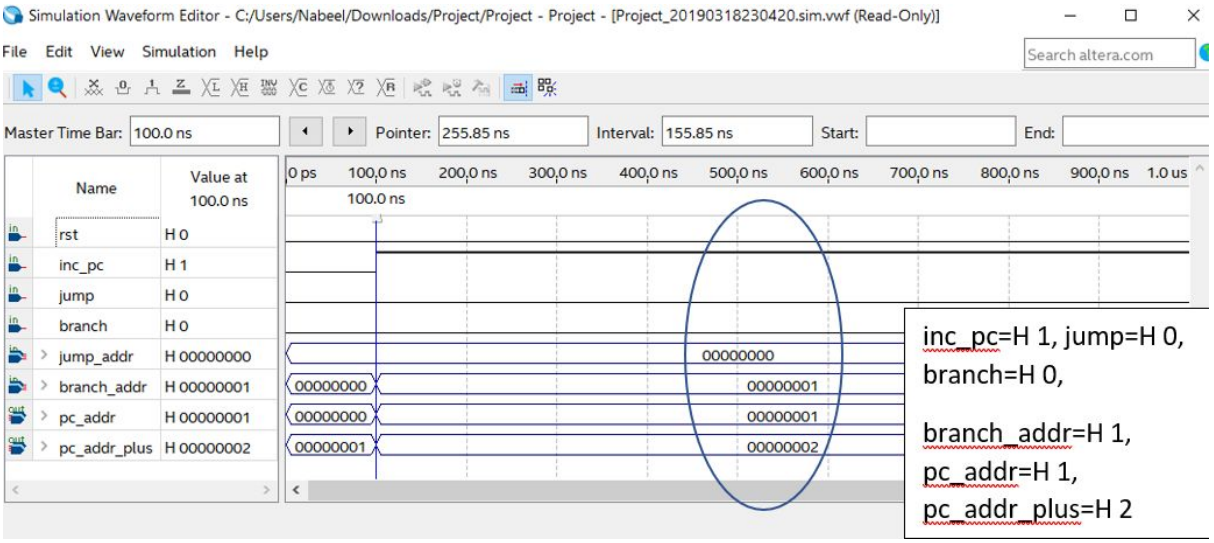
branch



Input/Output	Hexadecimal	Binary
inc_pc (input)	1	1
branch (input)	1	1
branch_addr	A	1010
pc_addr	A	1010
pc_addr_plus	B	1011

When inc_pc is set to H 1, rst does not equal H 1 and branch equals H 1; then pc_addr is set to the branch_addr and branch_addr_plus is the pc_addr+1. And this is exactly what occurs when these numbers are input into the waveform generator.

jump and branch

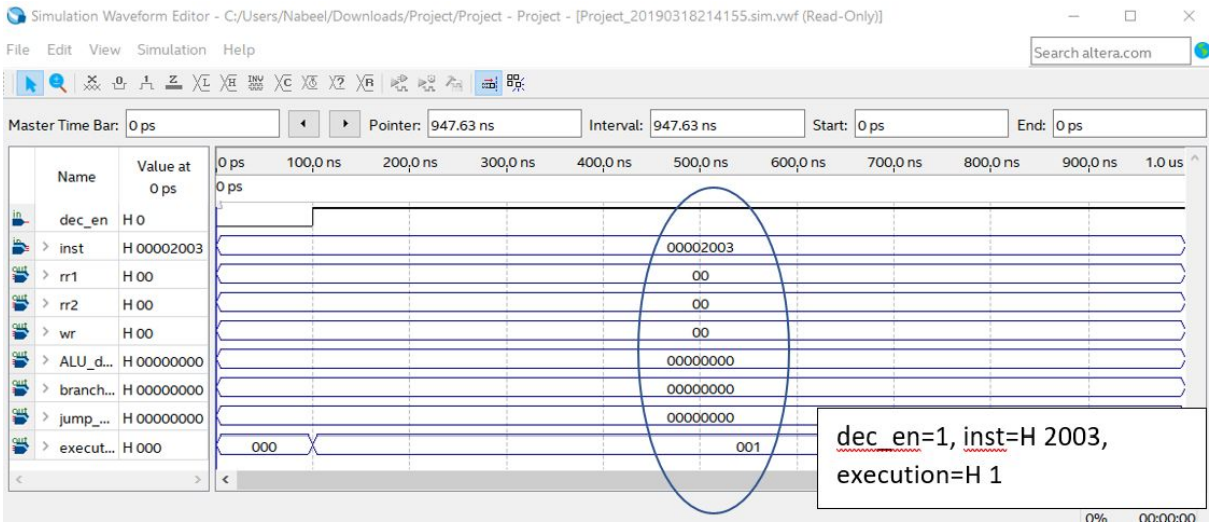


Input/Output	Hexadecimal	Binary
inc_pc (input)	1	1
jump (input)	0	0
Branch	0	0
branch_addr	1	1
pc_addr	1	1
pc_addr_plus	2	10

When inc_pc is set to H 1, rst does not equal H 1, jump and branch equal H 0; then pc_addr is set to the pc_addr_plus and pc_addr_plus is incremented by 1. And this is exactly what occurs when these numbers are input into the waveform generator.

INST_DECODER

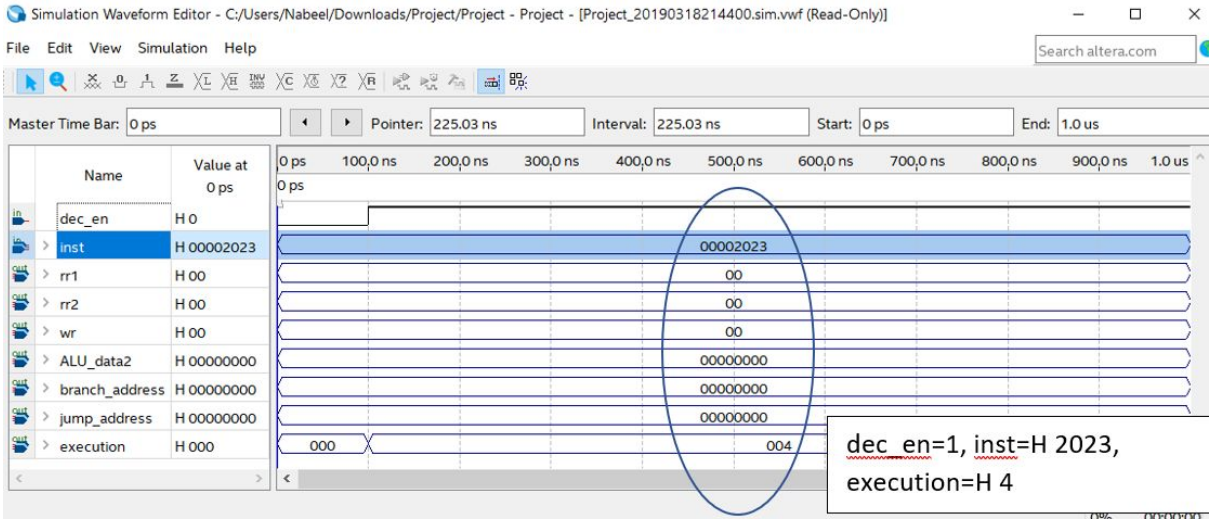
I1



Input/Output	Hexadecimal	Binary
dec_en (input)	1	1
inst (input)	2003	10000000000011
execution (output)	1	1

This is accurate since when inst=B 10000000000011, execution should be set to B 1 since this is the binary code for LW in this module.

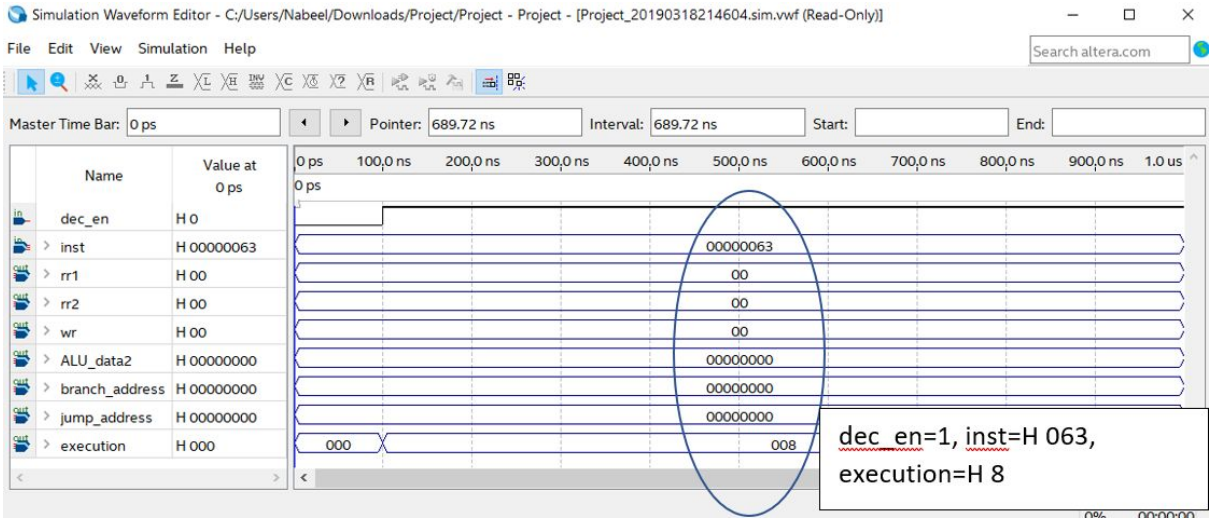
S1



Input/Output	Hexadecimal	Binary
dec_en (input)	1	1
inst (input)	2023	10000000100011
execution (output)	4	100

This is accurate since when inst=B 10000000100011, execution should be set to B 100 since this is the binary code for SW in this module.

S2



Input/Output	Hexadecimal	Binary
dec_en (input)	1	1
inst (input)	63	1100011
execution (output)	8	1000

This is accurate since when inst=B 1100011, execution should be set to B 1000 since this is the binary code for BEQ in this module.

Part H.

Members	Tasks
Nabeel Amod	- Completed Parts A, B, C, D, G - Contributed to H and I - Created a Google Doc for all members to contribute to.
Jeyaprashanth S.	- Completed Part E
Donny Miller	- Part F - Contributed to Part I
Emmanuel I.	- Part F

Part I.

We learned how to use a new software in Quartus, where we learned how to create new projects. We used Verilog HDL Files as well as University Program VWF to make waveforms. It was our first time using this program so every step was a new learning experience. A lot of trial and error took place throughout this first Milestone project in order to complete it. We learned how to compile and test our files using Quartus as well as create projects using established ALU's and files. Overall, was a good learning experience and gave us a more hands on view on how computers process information, interact with inputs/outputs, and the interaction between modules.