EECS 2021 PROJECT MILESTONE #1

PART A: Overview

This whole project is based on how an instruction in RISC-V processes into the CPU using six states: **Fetching, Decoding, Control, Executing, Writeback, and Change PC**. The project is outlined in the CPU_top module and we have control module as the brain of the project. This project gives us an entire idea from how an instruction is fetched to its execution. In this project we have total of eight modules namely: pc, registers, ALU, inst_ROM, inst_decoder, control, CPU_top and data_RAM. We will see a detail description of each module later in this report.

PART B: Module-Wise Explanation

- **pc module**:

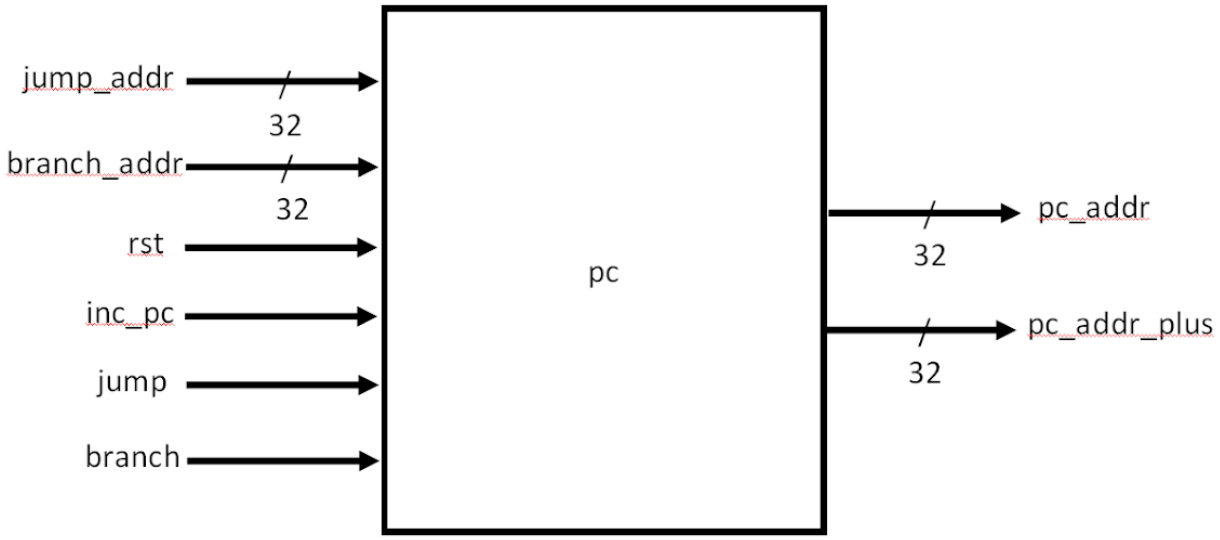
This module consists a total of 6 inputs and 2 outputs. Inputs are:

a. rst (reset).
b. Inc_pc (increment pc)
c. Jump
d. Branch
e. jump_addr (jumping address)
f. branch_addr (branch address)

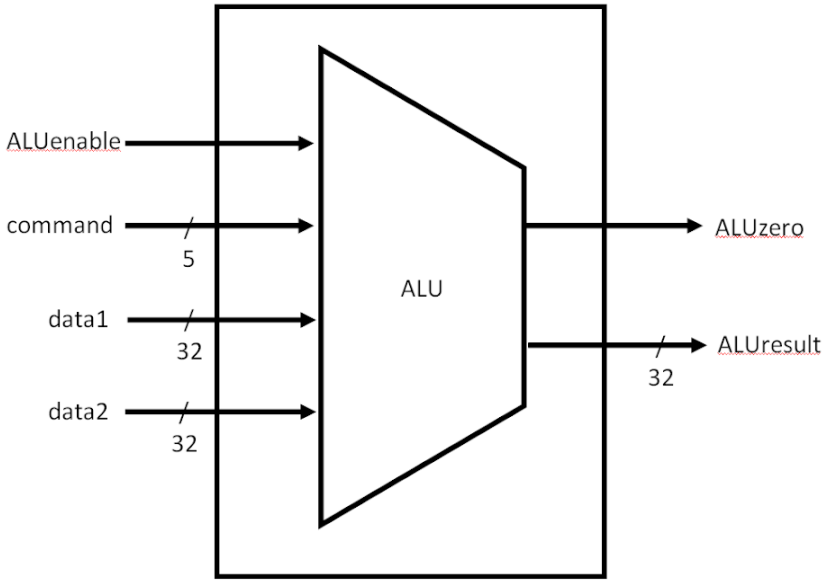
Based on these inputs we get two outputs:

g. pc_addr (pc address)
h. pc_addr_plus (increments pc address by 1)

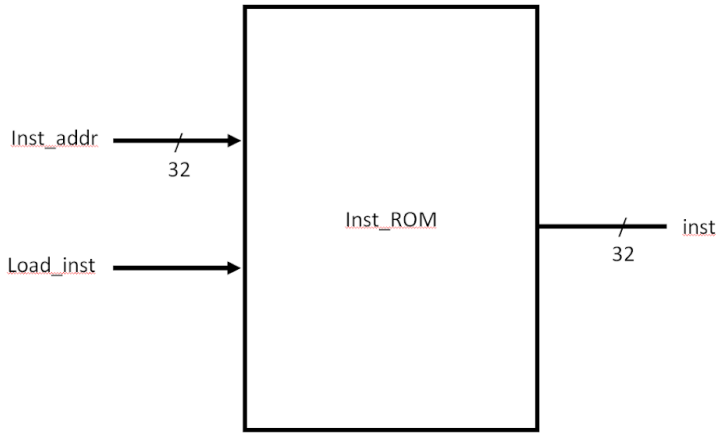
This module determines the address of PC (Program Counter) whether it is a branch or jump instruction or neither of the instructions. If the program is resetting, the PC address will be set to 0 (32 bits). If not then, it will see whether a branch or jump instruction is implemented. If it is branch, the PC address will be updated to branch_addr with a target address. If its branch instruction, then PC address will be updated to branch_addr (branching address). But if the current instruction is neither of the two, the PC address will increment by 1 and will be updated to pc_addr_plus. Here is an illustration of this module:



- **ALU module:**
  This is out logical operational unit of the CPU, where all the logical operations are performed (Arithmetic Logic Unit). In this unit, we have 4 inputs and 2 outputs. Two data for the operation (32 bits), one command (5 bits) to determine which operation we need to perform on those data's and an ALUenable. Based on the type of command i.e., ADD, SUB, SLL, XOR, OR the result will be will be updated into ALU_result. How does it determines which command it has? It is determined from the control unit of our CPU. In the control unit, when the state is "control" and it is executing the current instruction, it determines the type of execution and from there it triggers the ALU… operation whichever command it is. For example: if during the control state, the execution instruction is of addition, then in the execution state it will perform ALUADD operation into the ALU module and update its result in ALU_result wire. Each command is of 5 bits, with each of them uniquely stored for different operations. If the result is zero, then ALU_zero will store 32'b00000000 in it. Diagrammatically we can show this module as:

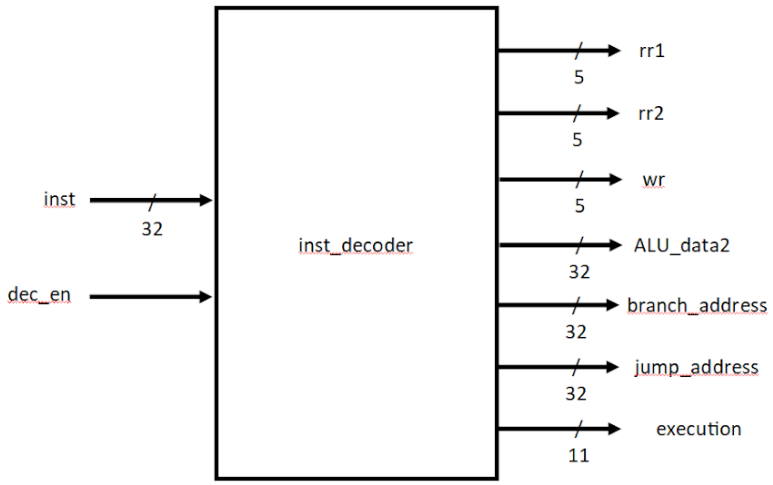


- **Inst_ROM module**
  This module is read-only so it is ROM. If the load instruction control signal is 1 into the control unit, then it will get the instruction indexed by the inst_addr from the pc module. Else if it is not the load instruction then inst will be updated by the current instruction based on its address. Therefore, this module reads the instruction indexed on the instruction address.



- **Inst_decoder**

  It is important to decode the values to do the operation in the ALU and which type of instruction it is. This module takes the inst (instruction) as the input from the inst_ROM module after it reads the instruction based on its address. As we know that any instruction has a 32-bit value when converted to machine language. And we also know how those 32 bits are partitioned in opcode, rs1, rs2, rd, functional code, etc. based on the instruction. So firstly, it takes the first 7 bits of the instruction which is the opcode of the instruction and based on it, it determines whether its I-type, S-type, R-type, UJ-type or SH-type. After it checks the opcode, the decoder checks the 12^{th} – 14^{th} bit which are the functional code of the operation. After determining the opcode and its functional code it is easier to get the rs1, rs2, rd, or imm values of the particular instruction. Inst_decoder has a total of 7 outputs: rr1, rr2, wr, ALU_data2, branch_address, jump_address, execution. Both rr1 and rr2 are rs1 and rs2, wr is the target register or rd in which the answer value is stored from the operations (ADD, SUB, LW, SLLI, SLL, XOR, OR, JAL). Branch address and jump address are triggered only when the opcode is 1100011(branch) or 1101111(jump and link). The netlist viewer has a complicated diagram of this module so in a simpler manner here is a diagram:

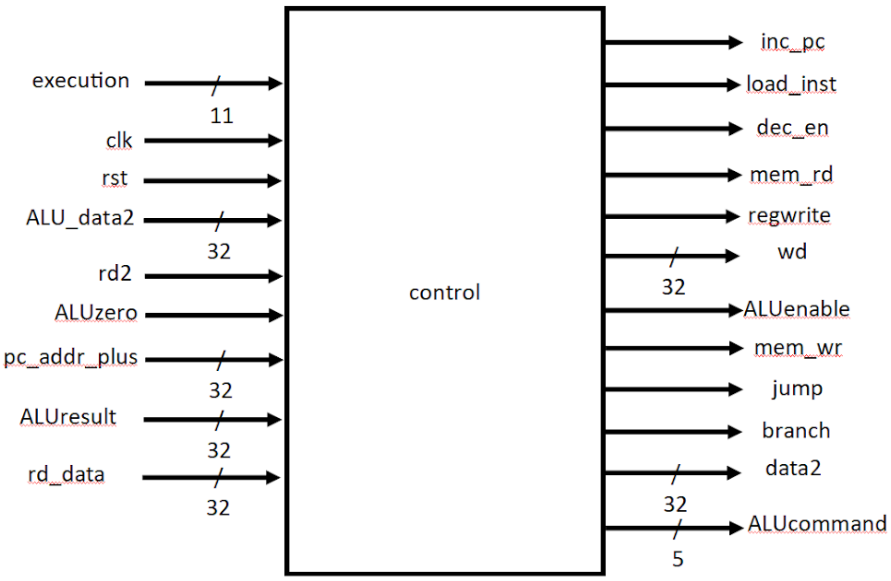


- **Control module**

  This module is also called the brain of the CPU we designed. It has a total of 9 inputs:

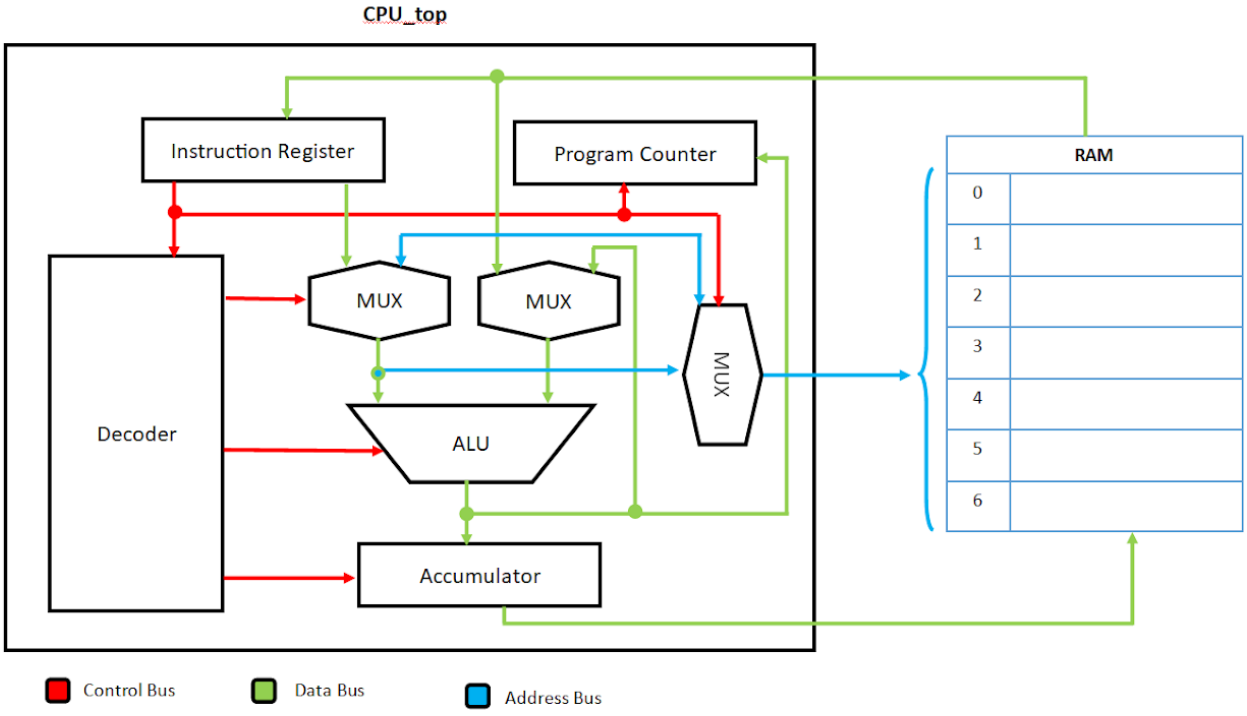
  a. execution (10 bits)
  b. clk (clock)
  c. rst (reset)
  d. ALU_data2
  e. rd2
  f. ALUzero
  g. pc_addr_plus
  h. ALUresult
  i. rd_data

  The control unit handles and processes each state of the CPU. So, it starts with checking whether the program is resetting or not. If yes, then everything becomes zero or else if the state is fetch: then the control unit will load one instruction from the inst_ROM module and change its state to decoding. When the state is decoding, the op_reg will be updated to 110000000 because it will start to decode the fetched instruction using the inst_decoder module and the state will be updated to control. Now during the control state, as we have determined the type of instruction in decoder module, based on that instruction it performs. For example, if the instruction is LW or SW, ALUsrc will be triggered to 1 because LW takes second input from ALU_data2. Also, it will update the ALU command to specific command. After determining the inputs from the control state, the execution state executes the instruction using the instruction to be executed and operation to be made using ALUcommand in the ALU module. It is also similar for the Writeback state, based on the executed data and the instruction, the WB state begins to read and write into the register writing. Finally, the change pc state increments the Program Counter for the next instruction changing the state again to fetch. Also, in the change pc state op_reg will write the result into register file. This is how our control unit works.



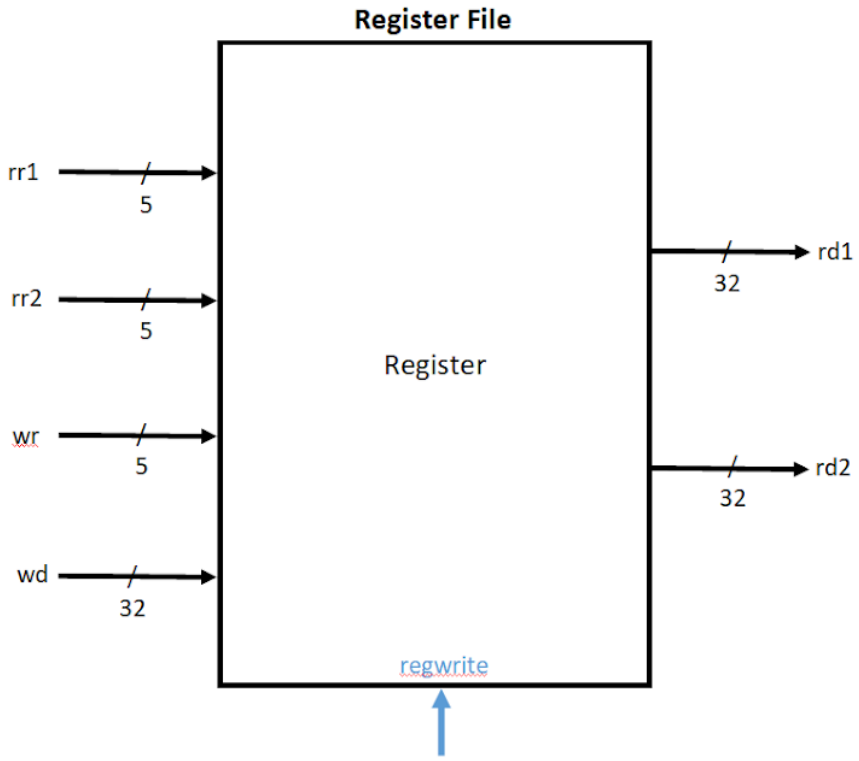
- **CPU_top module**

  This explains the entire design of the project we are working on. The RTL netlist viewer gives us an entire idea of how every module of the project is connected to each other. There is no code inside it, instead all seven module instances are created into it. Like the java interface, where it just consists of method headers, similarly these instances are like headers of various modules which calls them and processes them accordingly.

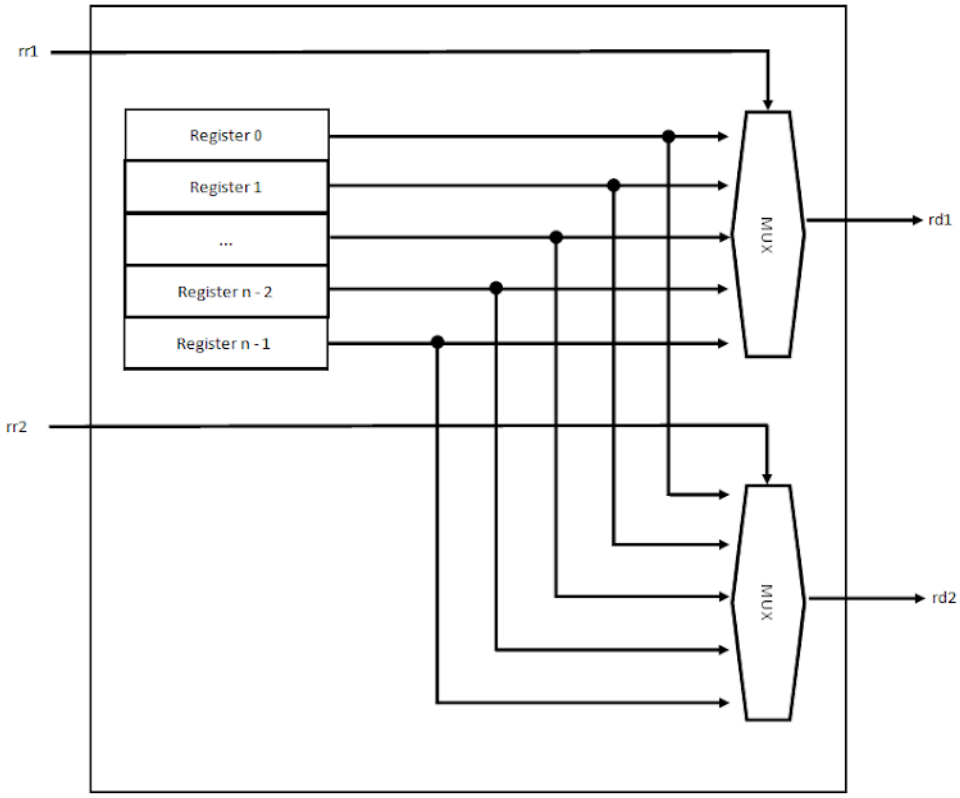


- **register module**

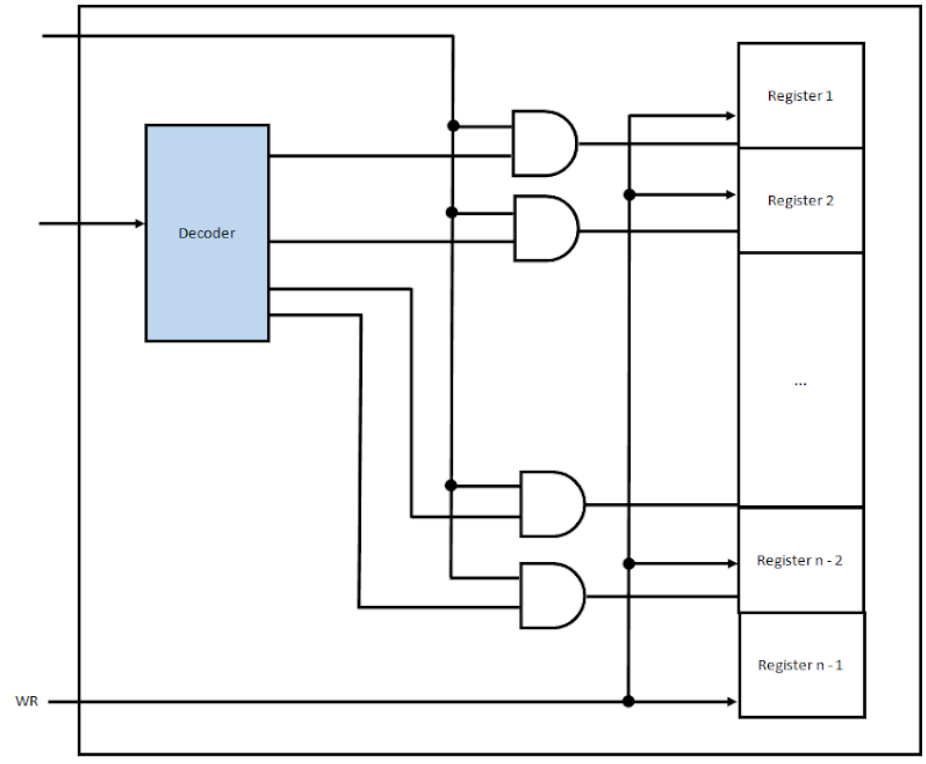
  This module creates a registerfile to store registers. So, it acts as a combination of 2-mux's: which means based on the input of rr1 and rr2, the registers are stored into the destination registers rd1 and rd2.



Register File: Read

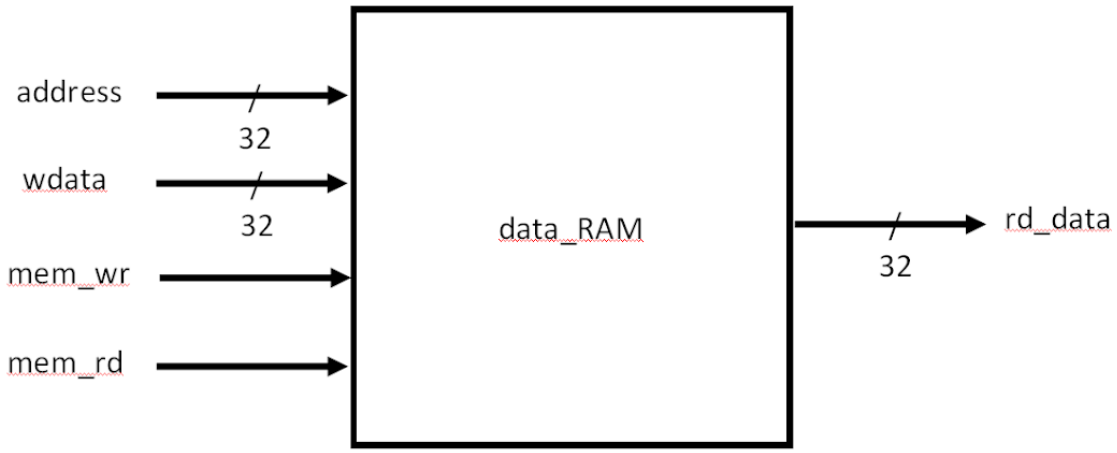


Register File: Write



- **data_RAM module**

  This module will trigger when we have to function regwrite or when we have to write back the result of the operation to the target register. In this module, the address of the value and the writing data will be fed to RAM module including the control signals mem_wr (memory write) and mem_rd (memory read). So if the instruction requires mem_rd as 1 then rd_data (reading data) will be assigned based on the address of the wdata fed to this module. Also, whenever the mem_wr has the rising edge, RAM will update its size with address as its index.



PART C: Explanation of the System

Creating a processor requires putting together different components as we have learned in the course. Using abstraction we can break the whole system into smaller subsystems to help understand and read the code better while also giving up the ability to add more functions in future cases.

The following is a simplified overview of the CPU and how it behaves:

Firstly, the control module is the “brain” of our CPU because it tells our modules to what and when to process.

When the machine code is entered into the system it is in the ROM module, the control unit tells the **decoder** the process that instruction. Here the decoder extracts the registers, type of instruction and branch address from the machine code and send it to different parts of the CPU.

Some of these parts like the registers are sent to the register module. This module stores the registers so they can accessed later.

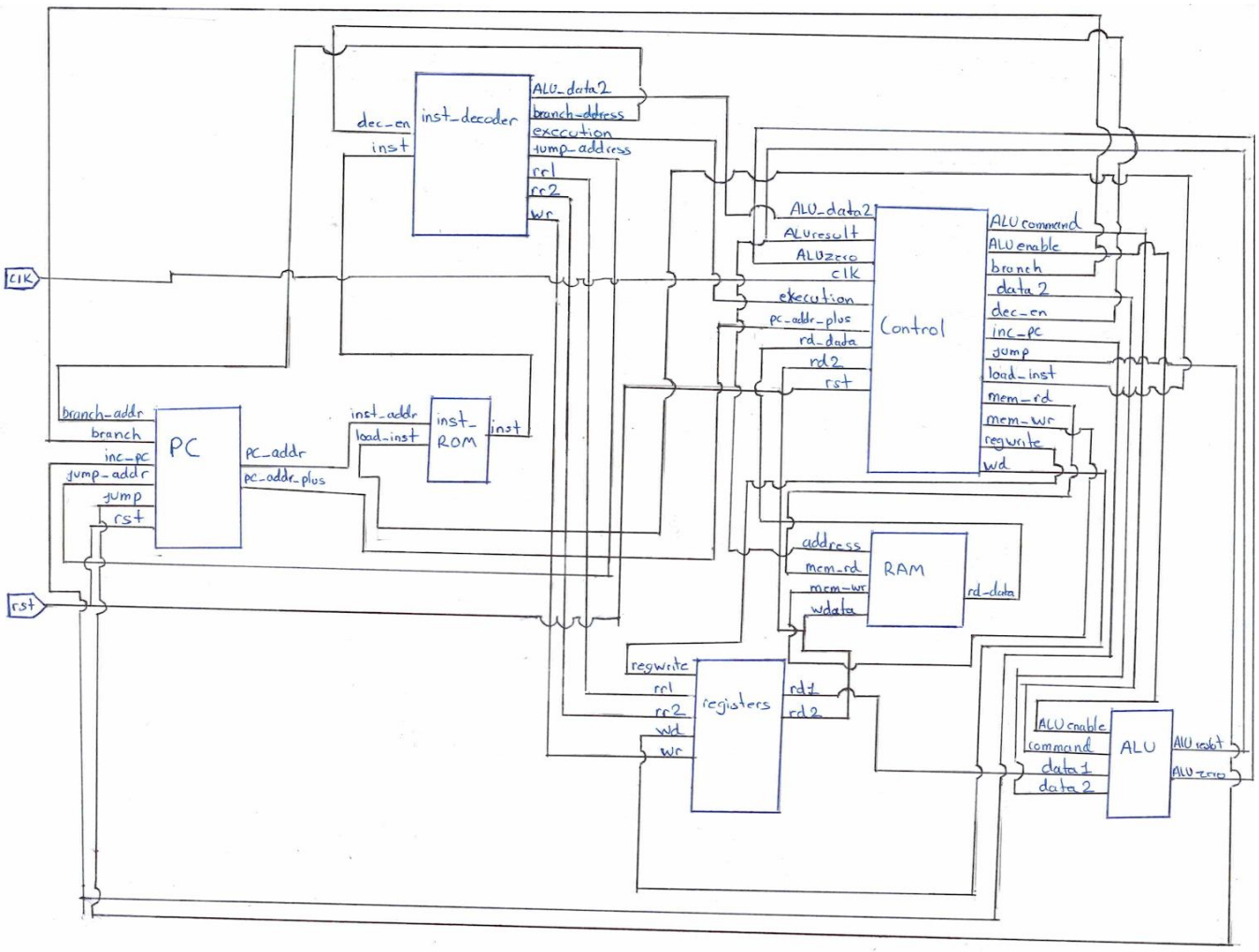
Jump address and branch address are sent the PC (Program counter) which basically acts as memory operator. It can move through the memory and provide us with memory address we need depending on the instruction.

The next step depends on the control, based on the instruction read it will update the ALU command to a specific command that was in the instruction and also send the data required by the ALU.

The ALU module is where the arithmetic actually happens, as mentions the ALU takes the its command and the data from the control module and the register module and it conducts the arithmetic and the results back to the control.

Back in the control module, we have many different that can be performed, like store the result into the register module, jump/branch to different memory address, perform an ALU command again or simply output the result.

The following is the full diagram which shows the connectivity of all the modules.



PART D: Implementation of RISC-V Instructions

The instructions that will be implemented are of the R type instructions.

The format for the R-type instruction is:

7-bit funct | 5-bit-rs2 | 5-bit-rs1 | 3-bit-funct | 5-bit rd | 7-bit opcode

NOTE: The reg [8:0]op_reg in the control unit consists of all the 9 control signals used in our project.

The first instruction that we will implement is:

xor x9, x5, x6……(rd, rs1, rs2)

We will first generate its machine code:

0000000_00110_00101_100_01001_0110011

So, when the instruction will be fed to inst_decdoder, it will determine the type of instruction and the type of execution it is doing. Here, first it will see the first 7 least significant bytes (LSB) and check the type of instruction. we have **0110011** as first 7 LSB, and that's an R-type instruction. Next we figure out what are its function 3 bits (12th - 14th bit). We have **100** as our function 3 bits and also check the **0000000** function 7 bits (25th- 31st bit) those determine the type of execution we are gonna have, and that is **XOR** in this case. So, till now we know its an XOR execution and an R-type instruction. As we know XOR has a target register, and two source registers (read register 1 and read register 2) on which XOR performs its operation. So, after decoding the instruction, the control unit will move towards the control state. The control state gives ALU module its data and the command to perform calculation. The control state has cases based on execution, so we have XOR as execution, will locate XOR case. For doing XOR we do not need any immediate value, instead we need the second input as the source register, so it will select rd2 as its second input. It will also assign the Alu command and update the state to executing. In the execution state, it will trigger ALUenable control signal. Now the ALU module will start operating XOR operation when the ALUenable has the rising edge, and store the result in ALUresult (32 bits). Now we have done our execution, the state becomes writeback. As the result is in ALUresult, the writeback state will select the output of ALUresult as the input of the register writing by triggering the regwrite control signal. Now remember, the output of inst_decoder is fed as input of the register module (rr1, rr2, wr). And also the state in the control will be updated to change pc because we have to process the remaining code. Again based on the type of execution, the change pc state will now write the result to the register file when it update the mem_wr control signal to 1. So, this is how XOR instruction is implemented into the CPU.

The Second instruction that we will implement is:

add x5, x6, x7…… (rd, rs1, rs2)

Firstly, we will generate its 32 bit machine code which is as follows:

0000000_00111_00110_000_00101_0110011

So, right now our pc address is at this instruction (assume this is the first instruction of the code). Also the state will initially be **fetch**. We also assume that the above instruction is being fetched from the inst_ROM module based on the value of our PC and the state is updated to decoding. Now the decoding state will trigger inst_decoder module to decode the above instruction by changing the value of op_reg to 110000000. The decoder module is designed perfectly to determine the instruction and its type of execution. Firstly, it has a case statement to read the first seven bits to check the opcode of the instruction, based on that, it will then check the 3-bit functional code of the instruction (12th - 14th bit). Since we have gotten the

opcode and the 3-bit functional code of the instruction, it is easy to say whether the second input is of the immediate or register-type. In this case:

opcode: 0110011 -> R-type instruction.

funct3: 000 & funct7 : 0000000 -> ADD type operation.

Therefore the instruction will be ADD, the bits from 15-19 will be rs1, the bits from 20-24 will be our rs2 and the bits from 7-11 will be the targeted register. After determining the type of instruction and the operation, the control state will determine what values it has to take in order to execute the ADD operation i.e, whether the immediate is used or register value is used. Here, ALUsrc will be 0, why? Because, the ADD operation does not require an immediate value. And state will be updated to execution. Now the execution state performs the ADD operation using the ALU module. ALUresult will store the result of the operation. Next state is WriteBack, which first reads the data content from ram and then acts as an input to the register module. So in our case of the ADD instruction, it will select the output of the ALUresult ram as the input of register writing. This is how this instruction is implemented throughout the project.

The third instruction that we will implement is :

sub x12, x15, x14 .....(rd, rs1, rs2)

The machine code for this instruction is as follows:

0100000_01110_01111_000_01100_0110011

This is also an R-type instruction, so it is similar case as the previous two instructions.

- First the Control unit will **fetch** the code as op_reg will trigger the load_inst control signal. So, it will fetch instruction from ROM module and update its state to decoding.
- Secondly, after fetching the instruction, it will start decoding the instruction using inst_decoder module. And do it in the same way as we saw in previous instructions implementations. So from the opcode and function 3 bit we get SUB execution and R-type instruction.
- After decoding the instruction, it executes based upon the execution type. The control state gives SUB command to the ALU module and it performs subtraction operation on the two registers. Also, the control state will be updated to executingstate.
- The executing state will then trigger ALUenable control signal as 1 and it will make the subtraction happen into the ALU module saving the result in ALUresult and changing the state to writeback.
- Based on the SUB execution, the output of the ALUresult will act as an input of register writing. And change its state to change pc.
- The change pc state actually writes back into the register file and then updates the pc triggered by the positive edge of inc_pc (increment pc).

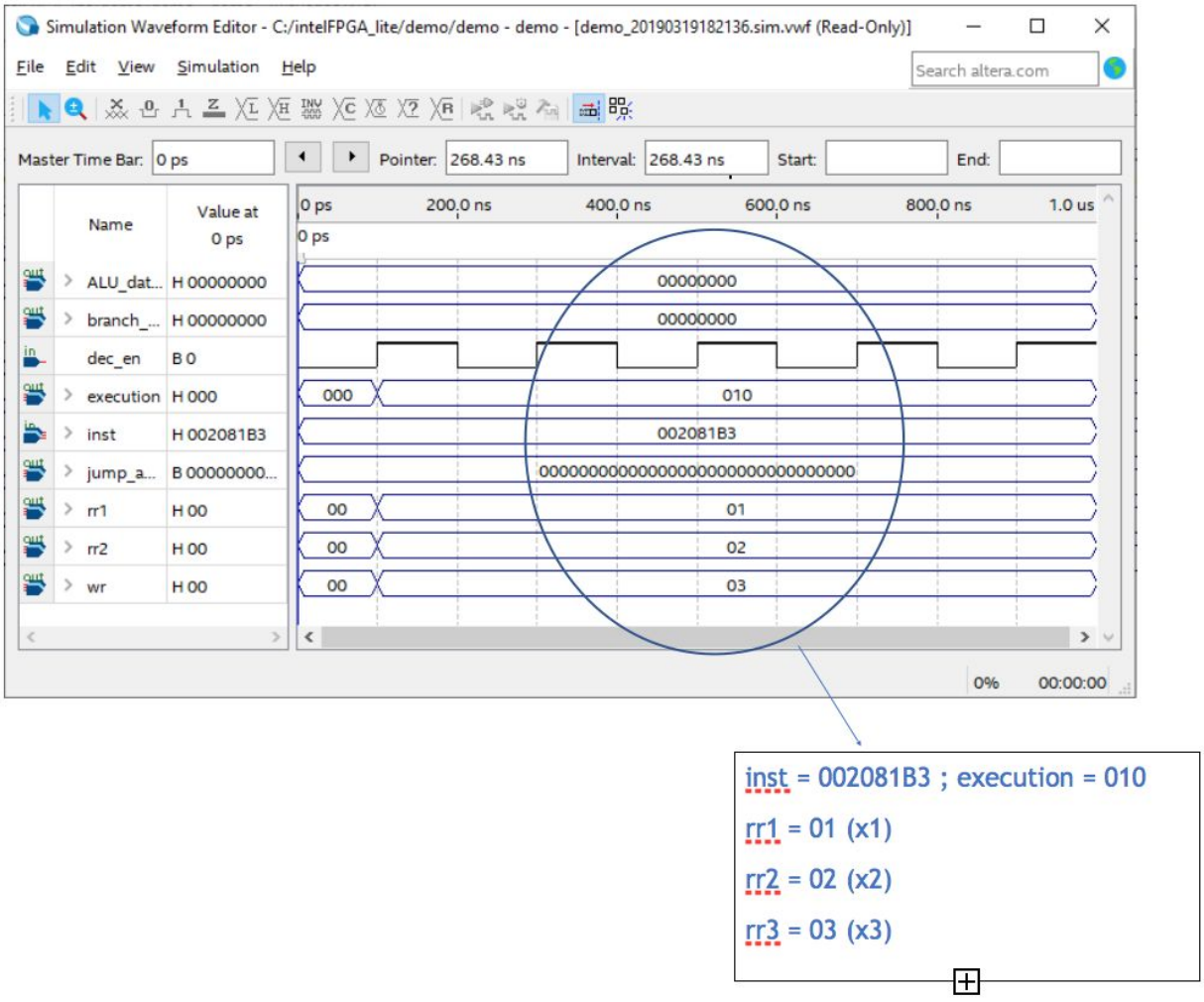
PART E: FUNCTIONALITY OF RELEVANT MODULES.

Functionality of relevant modules are shown below:

- **Inst_decoder module functionality:**

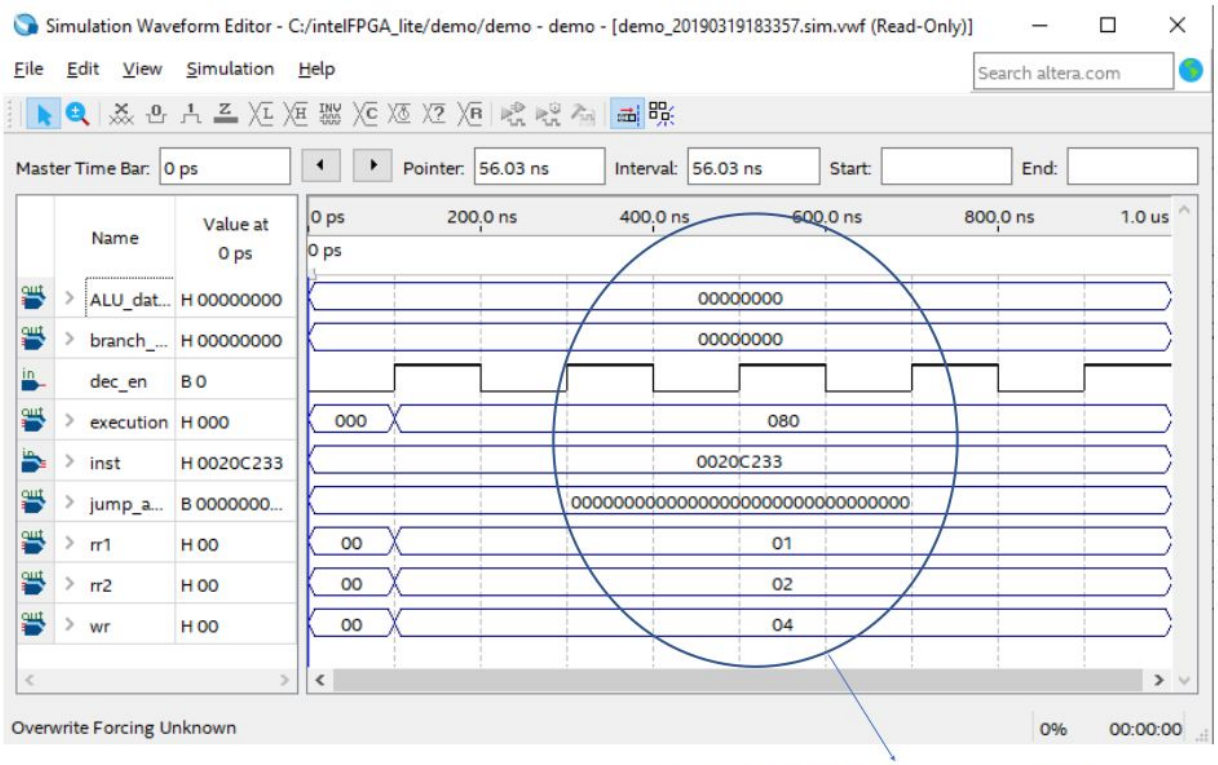
From time 0 to 100ns the value of execution is 0 because the rising edge of dec_en starts from 100ns and therefore from that clock cycle time 100ns the always block in inst_decoder module will execute.

a.



BINARY	HEXADECIMAL
0000000_00010_00001_000_00011_0110011	002081B3
0000_0001_0000	010

b.



inst = 0020C233; execution = 080

rr1 = 01 (x1)

rr2 = 02 (x2)

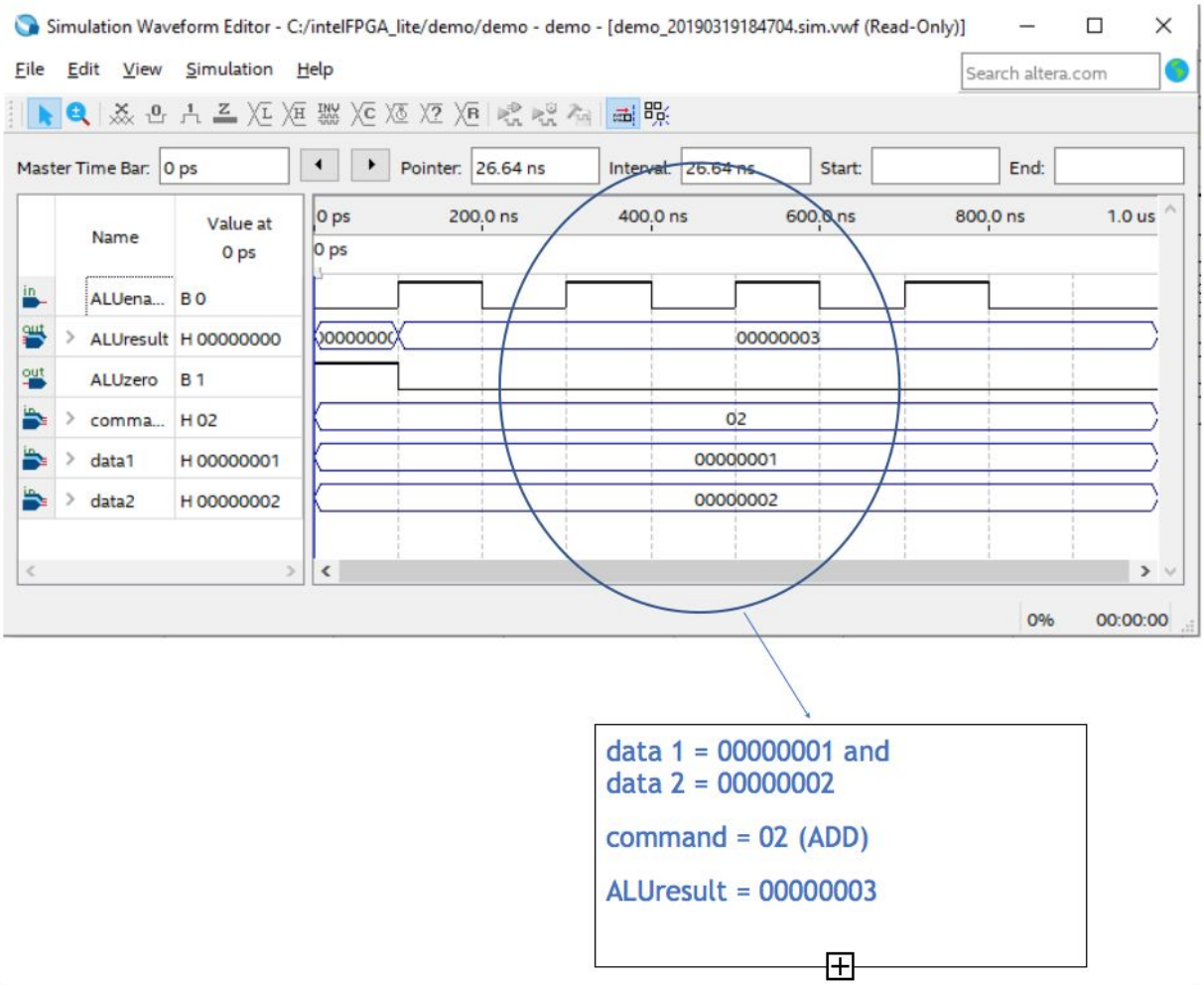
rr3 = 04 (x4)

BINARY	HEXADECIMAL
0000000_00010_00001_100_00100_0110011	0020C233
1000_0000	080

- **ALU module functionality:**

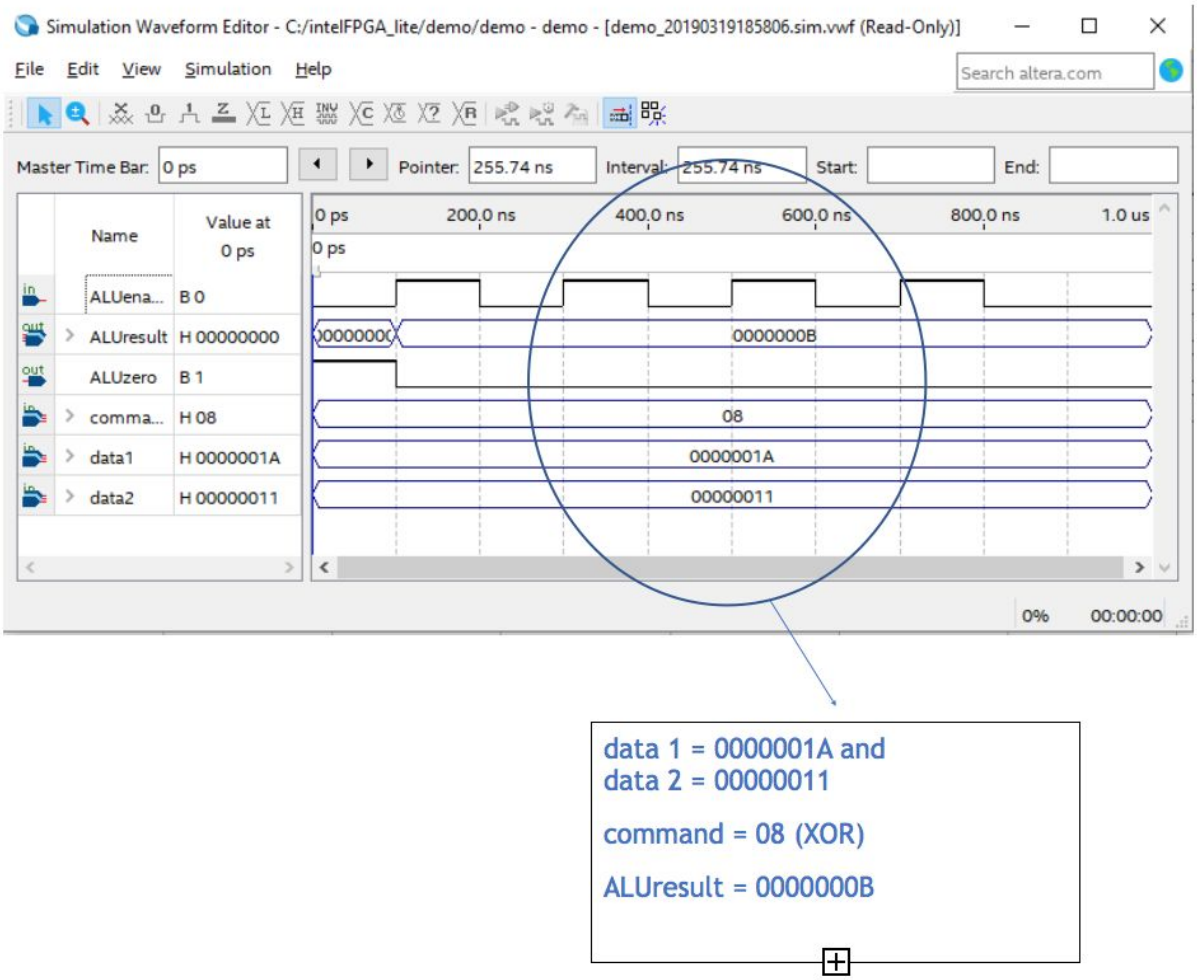
From time 0 to 100ns the value of ALUresult is 0 because the rising edge starts from 100ns and therefore from that clock cycle time 100ns the always block in ALU module will execute.

a.



BINARY	HEXADECIMAL
0000_0000_0000_0000_0000_0000_0000_0001	00000001
0000_0000_0000_0000_0000_0000_0000_0010	00000002
00010	02
0000_0000_0000_0000_0000_0000_0000_0101	00000003

b.



data 1 = 0000001A and
data 2 = 00000011

command = 08 (XOR)

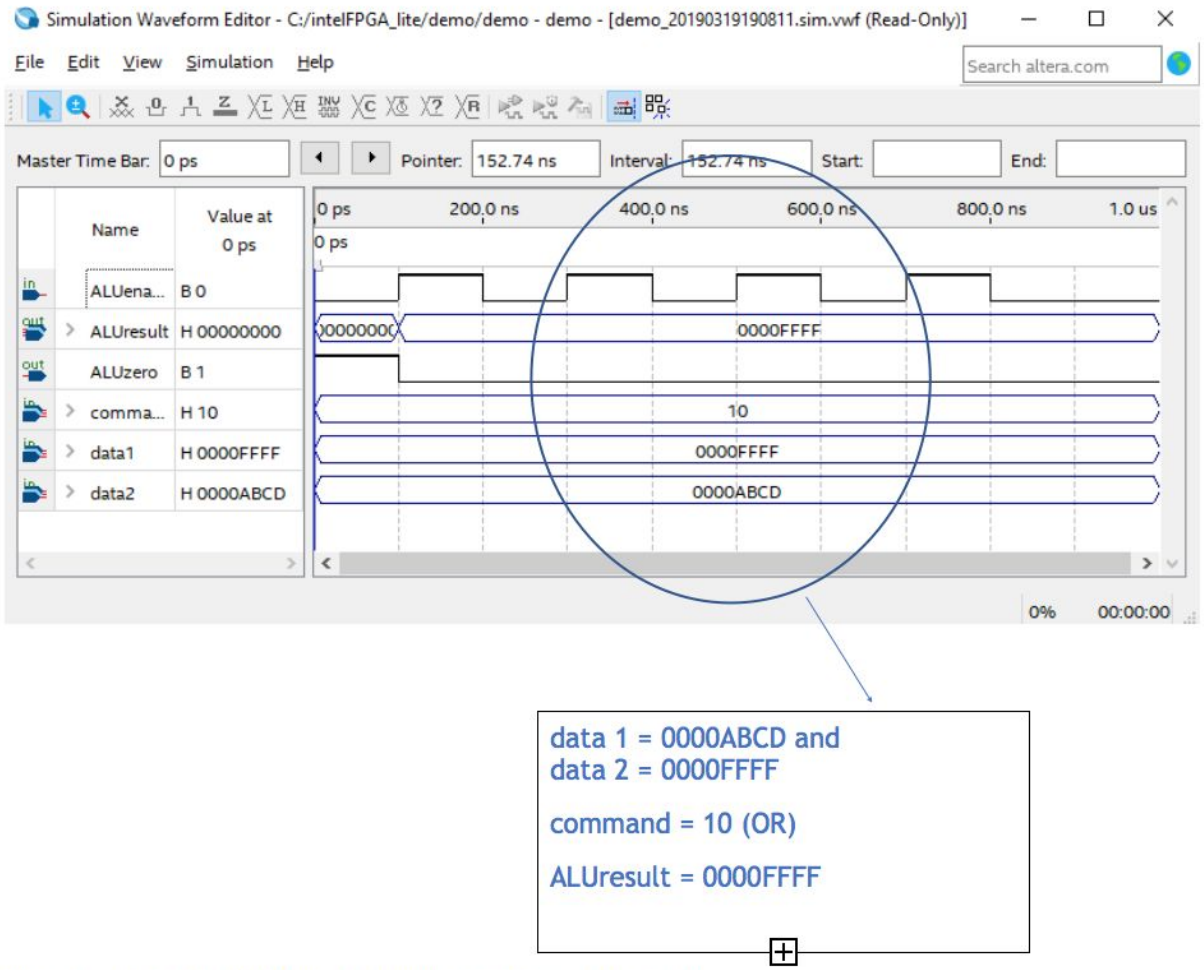
ALUresult = 0000000B

The result is B because:

XOR of 11010 and 10001 is 01011 and converting 01011 to hexadecimal is B.

BINARY	HEXADECIMAL
0000_0000_0000_0000_0000_0000_0001_1010	0000001A
0000_0000_0000_0000_0000_0000_0001_0001	00000011
00100	08
0000_0000_0000_0000_0000_0000_0000_1011	0000000B

c.



data 1 = 0000ABCD and
data 2 = 0000FFFF

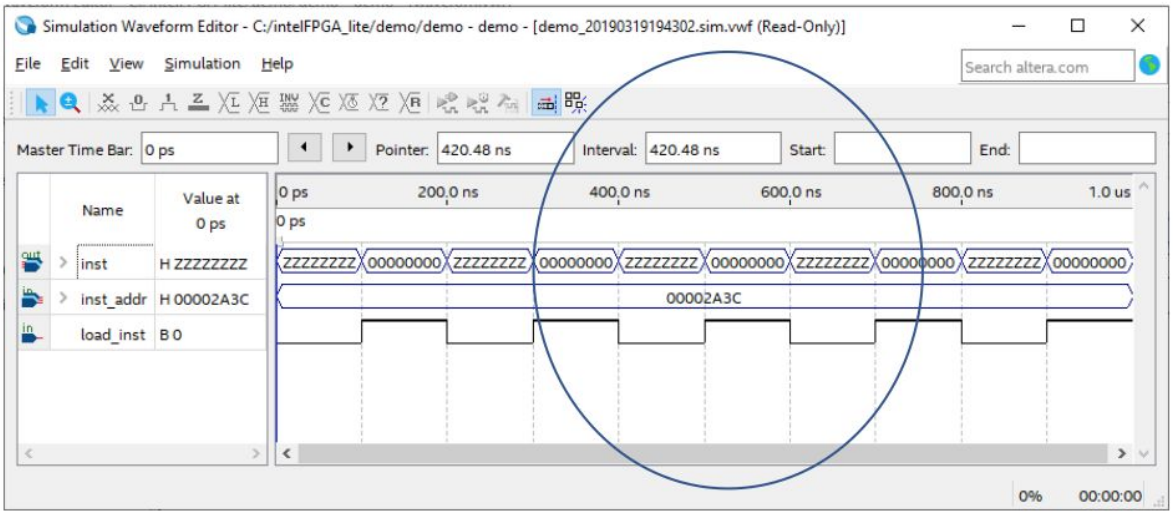
command = 10 (OR)

ALUresult = 0000FFFF

We are doing OR of ABCD with FFFF so the result is FFFF.

BINARY	HEXADECIMAL
0000_0000_0000_0000_1010_1011_1100_1101	0000ABCD
0000_0000_0000_0000_1111_1111_1111_1111	0000FFFF
10000	10
0000_0000_0000_0000_0000_0000_0000_1011	0000FFFF

- **inst_ROM functionality:**



Here, in the ROM module we have to input the instructions address and it gives the instruction based on the index of the address. When the load_inst has a rising edge, then it will give us the instruction from the ROM file based on the address, or else if it has a negative edge it will give “zzzzzzzz” in hexadecimal. Here in the above illustration we inputted a random Hexadecimal instruction address so when load_inst has a positive edge, it will give us the specific instruction in 32 bits and when it is a negative edge it gives “zzzzzzzz”.

BINARY	HEXADECIMAL
0000_0000_0000_0000_0010_1010_0011_1100	00002A3C

PART F: Project Breakdown

Priyank Patel	- RISC-V instructions implementation - module explanations
Kishan Patel	- module diagrams - functionality of relevant modules
Harpreet Goraya	- explanation of a system - full design diagram
Kairav Naik	- RISC-V instructions implementation - conclusion

Part G: Conclusion

In conclusion, through this project and milestone we have learned how the aforementioned modules are used to be able to process a RISC-V instruction into the CPU. The CPU itself is an interwoven web of different components/modules which as a team creates a system. With the control module acting as the brain, it decides what the modules will process. The different parts of an instruction is decoded and all the components (i.e registers, branch) are split off into different modules. Finally ending up in the ALU depending on the information that it needs which the control module can then use to perform an action. All in all this milestone introduced us to how a CPU is created.