Leanorine Lorenzana-Garcia

EC413

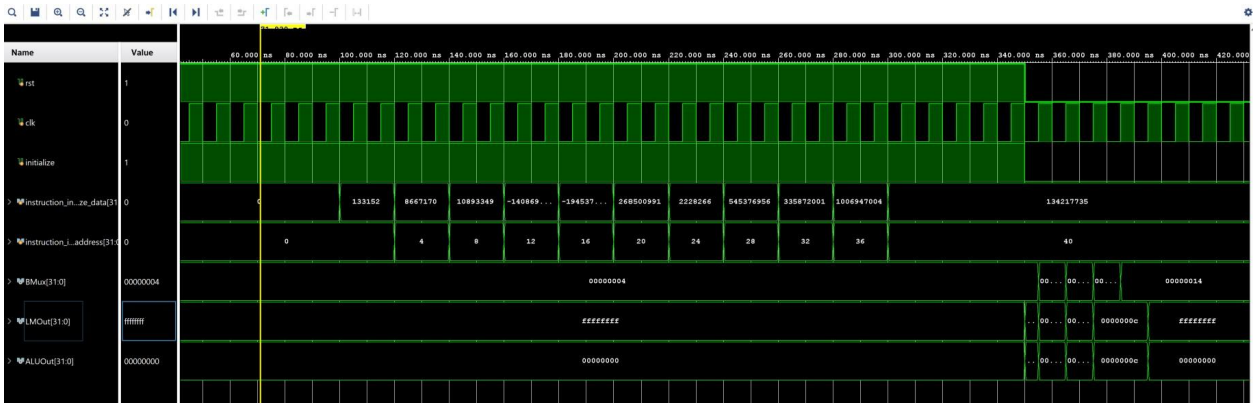
Lab 6 Report

Task 1:

For Task 1, we were asked to simulate the project and generate outputs for the instruction sequences given. These instructions include:

000000_00000_00010_00001_00000_10_0000	ADD R1, R0, R2
000000_00100_00100_01000_00000_10_0010	SUB R8, R4, $4
000000_00101_00110_00111_00000_10_0101	OR R5, R6, R7
101011_00000_01001_00000_00000_00_1100	SW R9, 12(R0)
100011_00000_01100_00000_00000_00_1100	LW R12, 12(R0)
000100_00000_00000_00000_00000_00_0010	BEQ R0, R0, -1

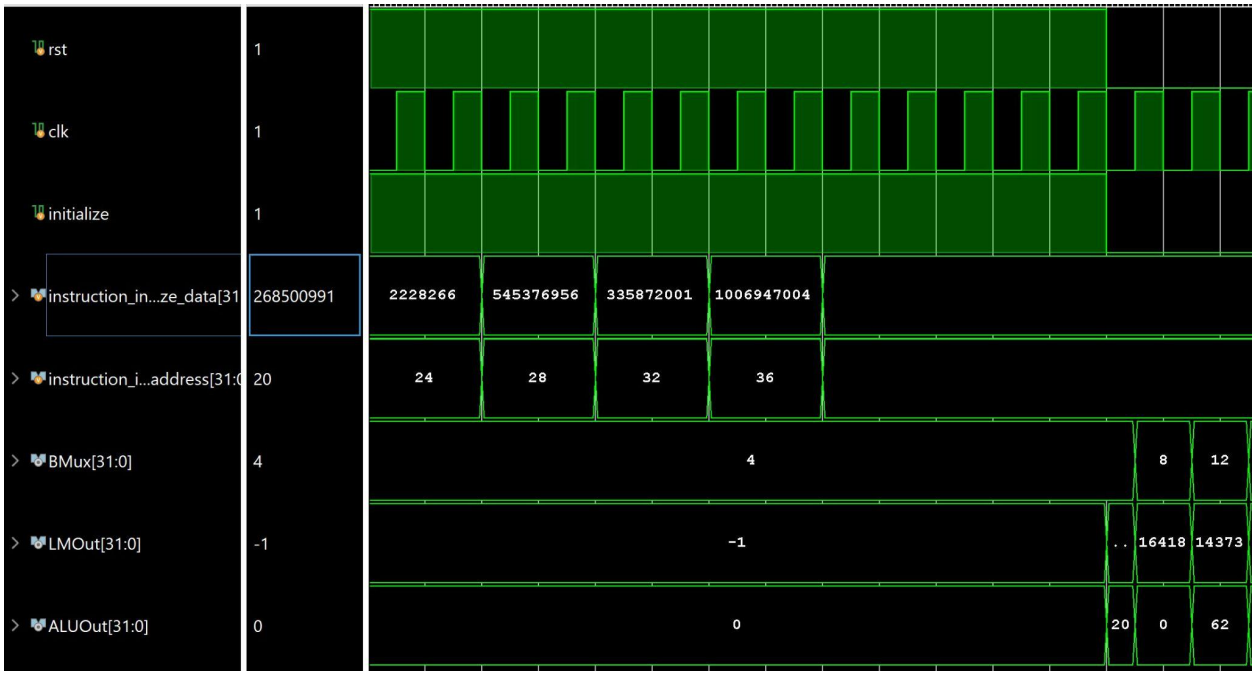
The Results of this task can be seen in the final wave form:



Task 2 SLT:

To implement the SLT ALU operation into the CPU I added a conditional statement to the `ALU_Control.v` file that checks if the 6-bit instruction code is 42 or 101010 in binary, the function number to which SLT corresponds to according to the MIPS table. If the input

instruction is found to be of the type SLT, it sets the output to 3'd5. I then added another conditional statement into the `ALU.v` file that checks the input function resulting from `ALU_control.v` is 3'd5 and performs the SLT operation. It does this by setting the output to 1 if the first input is less than the second input, otherwise it is set to 0.



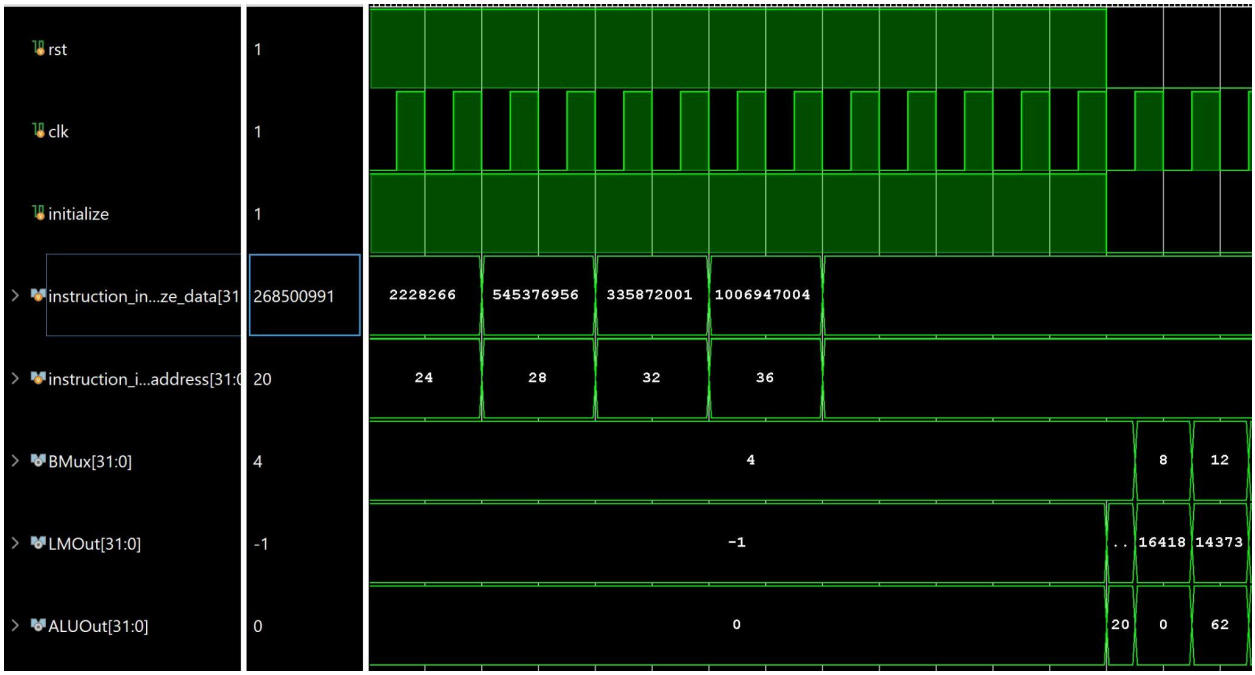
*Note SLT starts where instruction_initialize_address = 24;

Task 3 ADDI:

To add the ADDI function to the single-cycle CPU, I started by adding a conditional statement to the `ALU_control.v` file and it checks if the opcode corresponds to that of the ADDI function defined in the MIPS Table (001000). Then in the control file, if the correct opcode is passed in it sets the following outputs, which correspond to that of what is needed for an ADDI:

```
ALUOp = 2'b10 //to perform the ALU Addition
MemRead = 1'b0; // Not needed for immediate
MemtoReg = 1'b0; // not needed fop immediate
```

```
RegDst = 1'b0; // write result to readReg2

Branch = 1'b0;

ALUSrc = 1'b1;

MemWrite = 1'b0; // not writing to memory

RegWrite = 1'b1; //write result to a reg
```



*Note ADDI starts where instruction_initialize_address = 28;

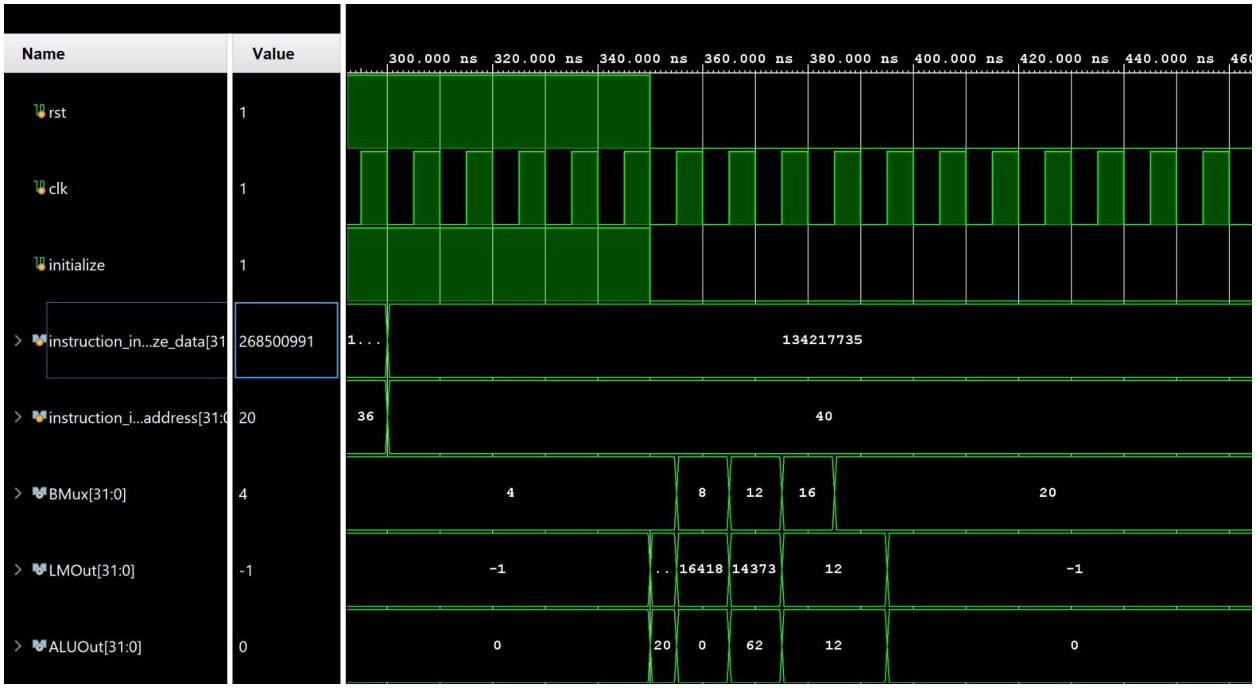
Task 4 J:

To implement the jump function I followed the textbooks implementation. To do this, I started like the other modules by adding a conditional statement to the `control.v` file that the opcode corresponds to the jump instruction according to the table (000010 = (2)_10). In that, I created an additional output signal J to be set to 1 in the case of the jump task. Along with that the outputs are as follows:

```
ALUOp = 2'b11; // don't care

MemRead = 1'b0;
```

```
MemtoReg = 1'b0;

RegDst = 1'b0;

Branch = 1'b0;

ALUSrc = 1'b0;

MemWrite = 1'b0;

RegWrite = 1'b0;

J = 1'b1;
```

You will notice almost every input is set to 0, this is because we do not use them and ALUOp is set to a don't care value. After this, in the main CPU module I used the `shift_left_2.v` to shift bits 25-0 of the instruction, since that is the offset specified by the J-type instruction, left by 2 bits as given in the diagram in class. Then to get the new address it is a combination of bits 31 to 28 of PC-Out, the new shifted instructions, and 2'b0. I then passed this value into a mux with select as the J input, in_0 as the output of the branch Mux and input 1 as the new address we computed. If the J input signal is 1 then we use the new address as our PC_in otherwise we continue with the previous address.



*Note J starts where instruction_initialize_address = 40;

Task 5 BNE:

To start implementing the BNE function, I started similar to the other modules in the control.v file. There I checked whether the corresponding opcode for BNE (000101) was inputted and set the various outputs. These are:

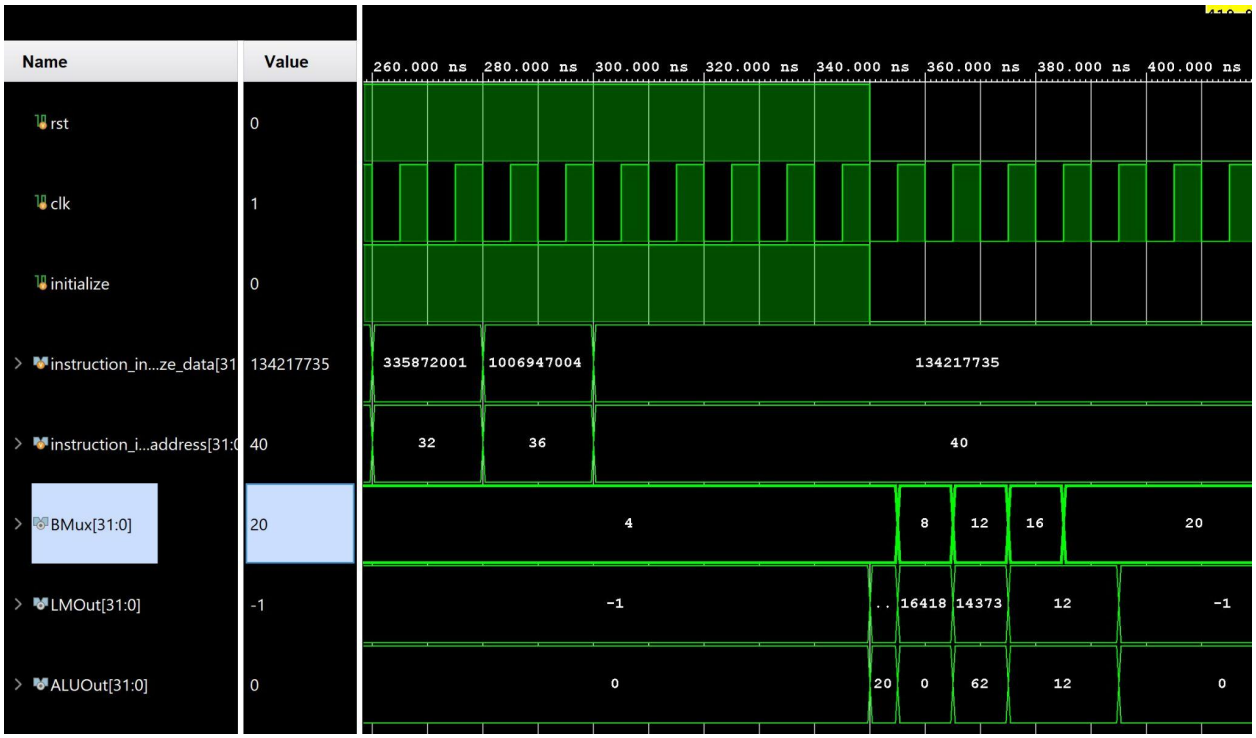
```
ALUOp = 2'b01; // similar to BEQ - using SUB
MemRead = 1'b0;
MemtoReg = 1'b0;
RegDst = 1'b0;
Branch = 1'b1; //BNE is a branch function
ALUSrc = 1'b0;
MemWrite = 1'b0;
RegWrite = 1'b0;
J = 1'b0;
```

```
BNE = 1'b1; // set to 1
```

Similar to the J function, you'll notice I created a BNE output and set it to 1 in the case of the BNE instruction. In addition, the branch output is also set to 1. Then using a model from online and watching a youtube video I was able to implement the BNE operation through the following:

- Using an AND gate to and the zero_flag and Branch input
- Using a NOT gate to get the inverse of zero_flag
- Using an AND gate to and the BNE from control with the result of the previous not gate
- Lastly, ORing the results of the two-AND gates and storing that into PCSrc

The addition of the logic gates can be found in the final updated diagram of the CPU.



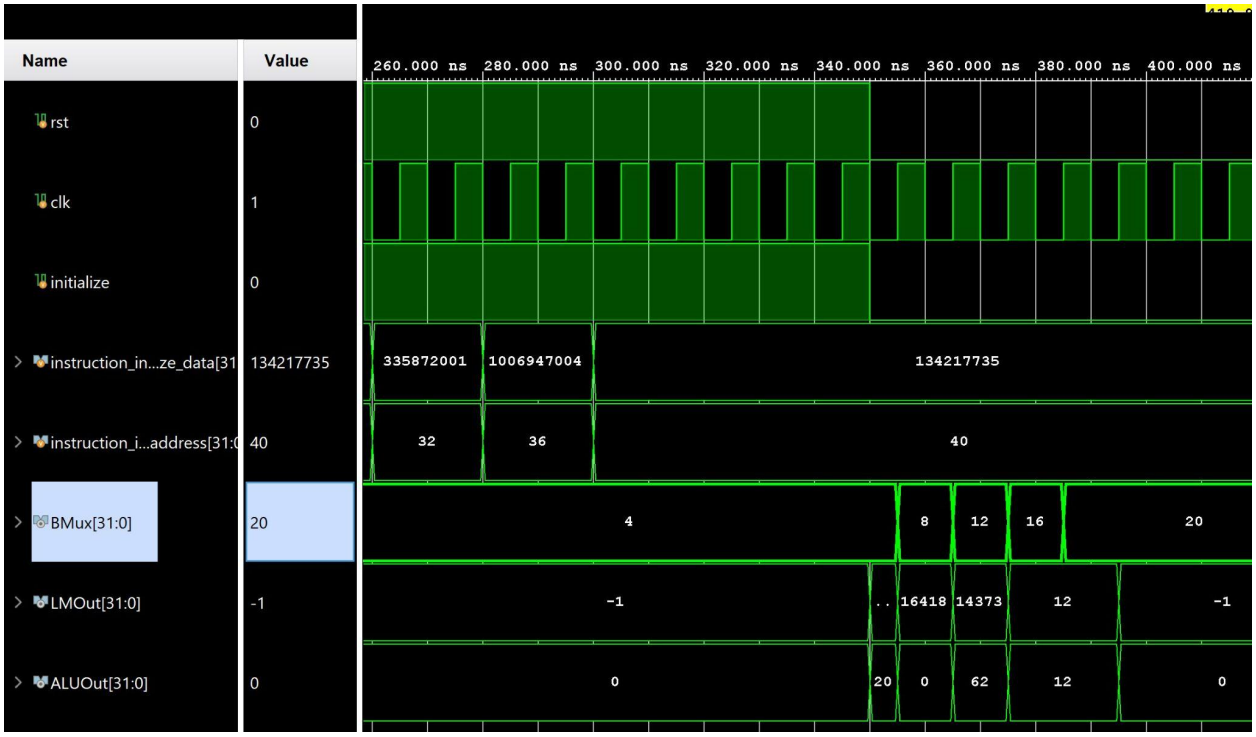
*Note BNE starts where instruction_initialize_address = 32;

Task 6 LUI:

Like previous instructions, to start implementing the LUI function I added a conditional statement to `control.v`. This statement looks to see if the proper opcode for LUI is input and sets the various outputs, accordingly:

```
ALUOp = 2'b10; // want to use ADD
MemRead = 1'b1; //read 4 upper bits from memory
MemtoReg = 1'b0;
RegDst = 1'b0;
Branch = 1'b0; //BNE is a branch function
ALUSrc = 1'b1;
MemWrite = 1'b0;
RegWrite = 1'b1; //Writing result to memory
J = 1'b0;
BNE = 1'b0;
LUI = 1'b1; //Performing LUI Function
```

Similar to tasks 4 and 5 I added the output LUI to be used in the overall CPU. To perform this function, in the cpu.v file I started by taking the first 16 bits of the instruction set and then adding 16 0's to make the proper 32 bit immediate instruction. I then fed that through a mux along with the sign-extended immediate where the resulting LUI from control decides which output to use.



*Note LUI starts where instruction_initialize_address = 36;

Testbench (`tb_cpu.v`):

To test the CPU module, I used the given file and added instructions to it. To test the CPU is working I output the results of the LUI Mux, ALUOut, and the Branch Mux. I tested the following instructions:

000000_00000_00010_00001_00000_10_0000 ADD R1, R0, R2

000000_00100_00100_01000_00000_10_0010 SUB R8, R4, $4

000000_00101_00110_00111_00000_10_0101 OR R5, R6, R7

101011_00000_01001_00000_00000_00_1100 SW R9, 12(R0)

100011_00000_01100_00000_00000_00_1100 LW R12, 12(R0)

000100_00000_00000_00000_00000_00_0010 BEQ R0, R0, -1

000000_00001_00010_00000_00000_10_1010; SLT R0, R1 , R2

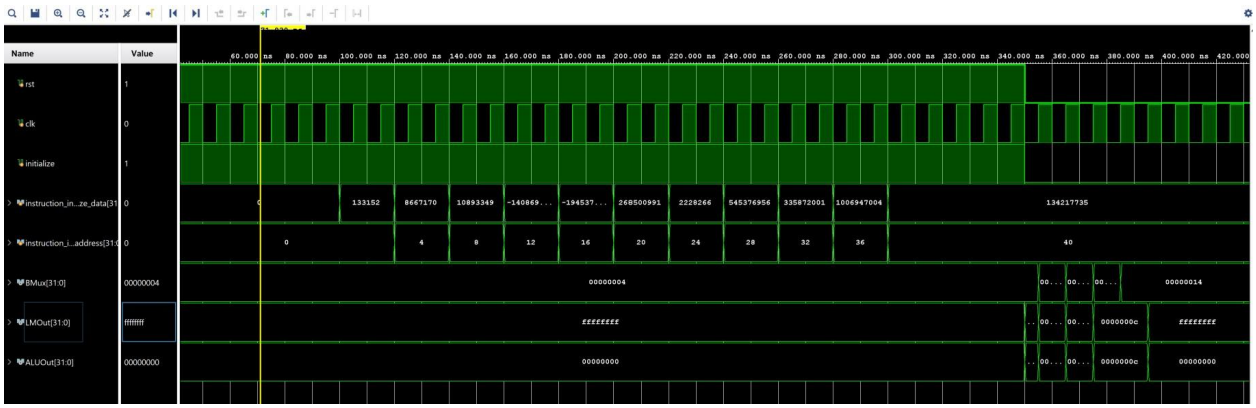
001000_00100_00001_1100101010111100; ADDI R1 , R4, 4'hcabc

000101_00000_00101_00000_00000_00_0001; BNE R0, R5, 1

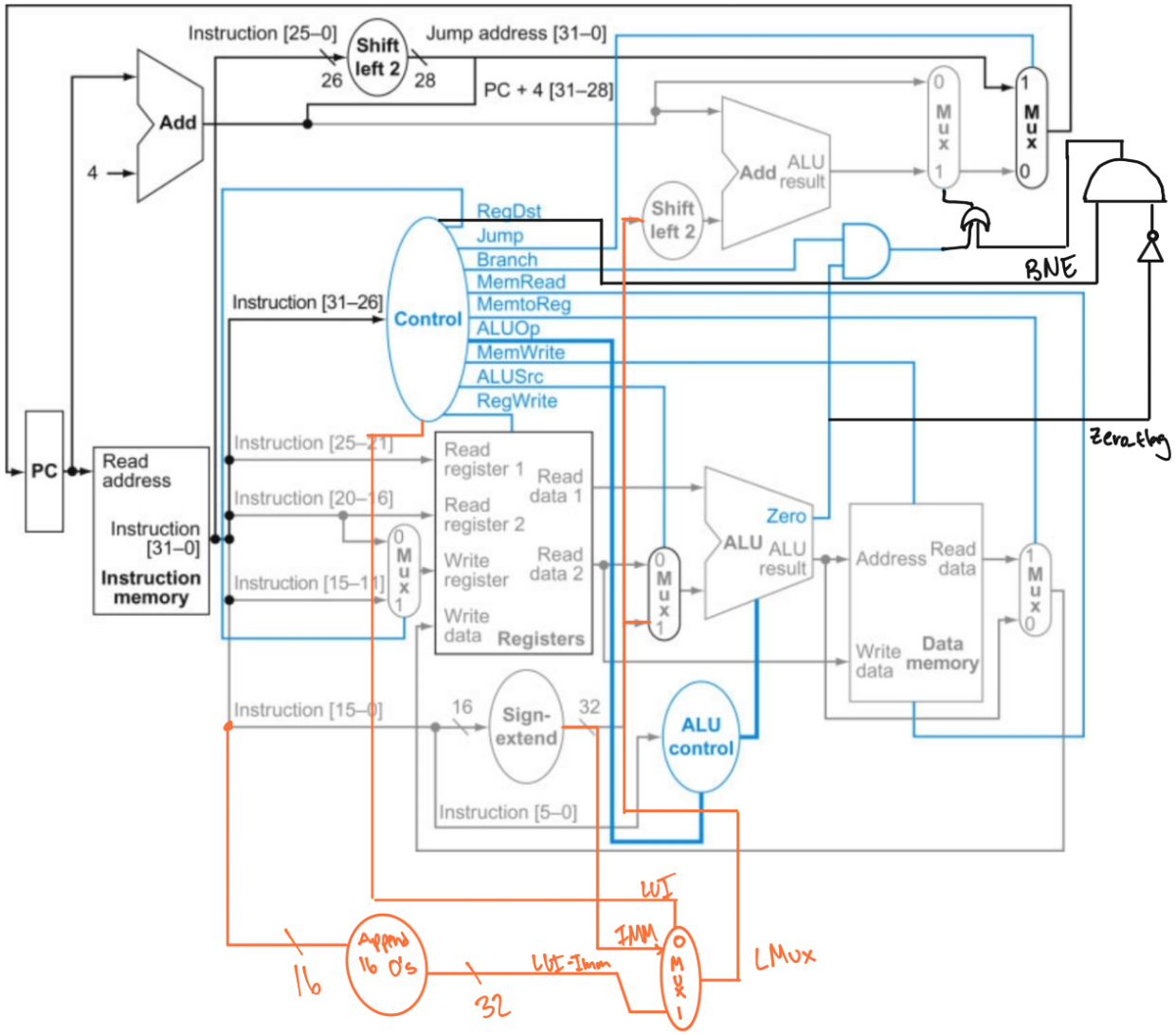
001111_00000_00100_1100101010111100; LUI R4, 4'hcabc

000010_00000_00000_00000_00000_00_0111; J 7

The Results of this task can be seen in the final wave form:



Modified Diagram:



Design Hierarchy:

```
cpu.v
    InstrMem.v
    control.v
    mux.v
    nbit_register_file.v
```

sign_extend.v

ALU_control.v

ALU.v

Memory.v

PC.v

Adder.v

shift_left_2.v