Tasks:

Task 1:
(See pre-lab)

Task 2:

In order to implement each I-Type instruction, I modified the control, ALU_control, and ALU modules. Inside the control module, I extended the ALUOp wire and func wire to 4 bits to leave space for more instructions/operations.

I then extended the conditionals to account for the unqiue opcodes for each I-Type instruction. Inside the sub-considitonals, I set the relevant ALUOp, MemRead, MemtoReg, RegDst, Branch, ALUSrc, MemWrite, and RegWrite bits. ALUSrc is set to 1 for all I-Type instructions, because we require the immediate as the second input to the ALU. RegWrite is also set to 1 for all I-Type instructions, so we can store the result of the operation in a register.

Inside the ALU_control module, I added more conditionals to parse for the new 4-bit ALUOps I made to map the I-Type instructions to the relevant ALU "func". For each I-Type instruction, I set the ALU func, J, and JR flag bits (J and JR flag bits are 0 for all I-Type instructions).

Lastly, inside the ALU module, I double checked that each I-Type instruction was being mapped to the proper ALU function inside the conditionals.

Tasks 3 & 4:

To implement the J-Type instructions, I implemented two additional MUXes in series after the "PC_Input_MUX". See diagram below for visualization of this. The first MUX (J_MUX) I implemented, immediately after the existing PC_Input_MUX, takes the output of that MUX as well as the immediate wire, with the J flag I implemented inside of ALU_control. When the J flag is 1 (meaning when we have fetched a Jump instruction), the MUX will send the immediate value forward to the PC.

This is because the next MUX (JR_MUX) takes the output of the J_MUX as one input, with Read_Data_1 as the second input, and the JR flag as select. When the J flag is 1, the JR flag is 0, so this MUX will forward the immediate from the previous MUX (J_MUX) to the PC.

Alternatively, when a JR instruction is being performed, the J flag/select will be 0, and JR flag/select will be 1, so the final JR_MUX will output the contents of the Read_Data_1 wire to the PC.

Task 5:

I implemented BNE by adding its OPCODE to the conditional inside the control module, setting the branch wire/flag to 1, and assigning it to ALUOp 0111. Then, inside ALU_control, I set up another conditional to check for this ALUOp, and map it to ALU func 7.

Inside the ALU module, I set up func 7 with a ternary that outputs 1 if inputs a and b are equal, or 0 if they are not equal. I do this because we want the zero_flag to output 1 for a branch instruction to properly execute with the given gate logic, so I want the output of the ALU to be 0 in order to trip this flag when the BNE comparison is "true".

Task 6:

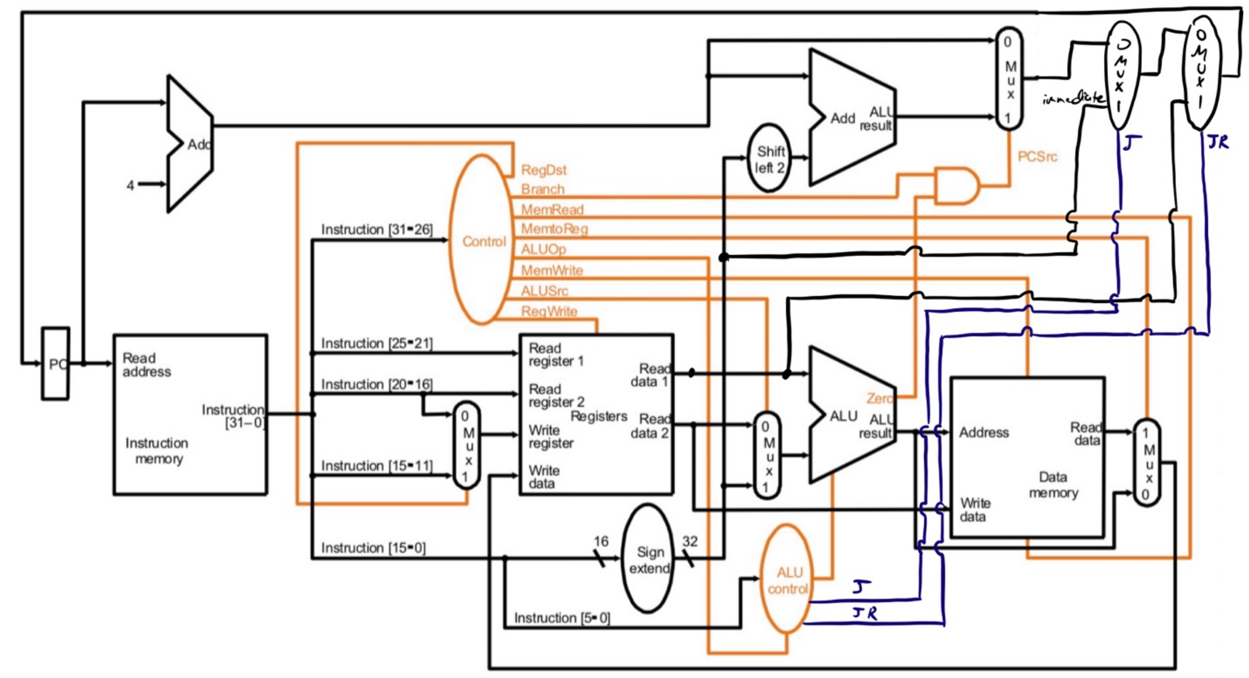
I implemented LUI by (again) adding another conditional to the control module, using OPCODE 1111 for LUI, ALUOP 1000, setting ALUSrc to 1 so we input the immediate into the ALU, and RegWrite to 1 to save the result to the specified register.

Then, inside ALU_control, I map LUI's ALUOP to ALU func 8, which corresponds to the following operation that I implemented inside the ALU module:

```
out[size-1:size-16] = b;        // lui
out[size-17:0] = 16'd0;
```

This operation is parameterized to take a 16-bit immediate and move it to the first 16 bits of any size register.

Modified Diagram:

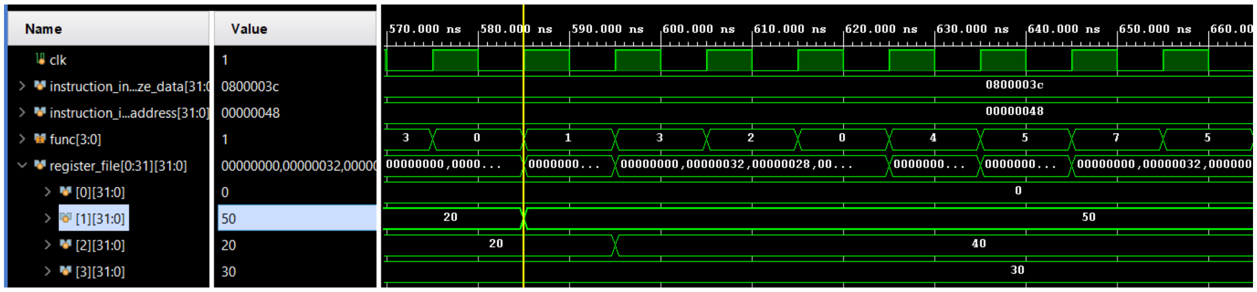


Testbench & Waveforms:

Format:

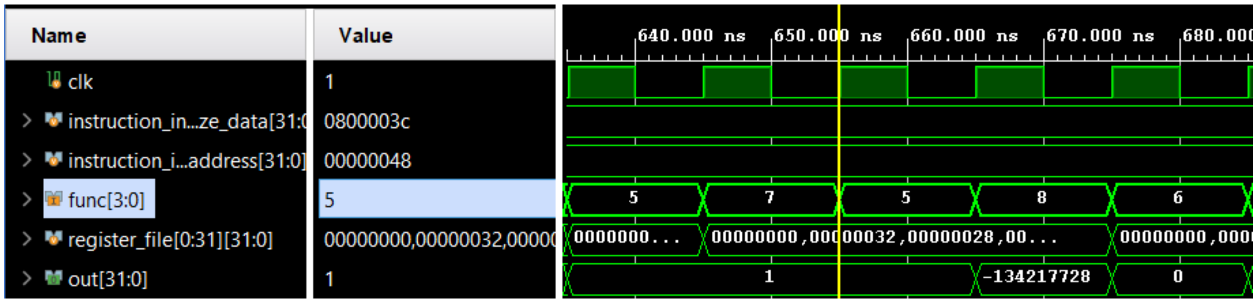
```
		Instruction_initialize_address = <32'b>
		Instruction_initialize_data = <32'b>
// <INSTRUCTION NAME>

		instruction_initialize_address = 0;
		instruction_initialize_data = 32'b000000_00000_00010_00001_00000_10_0000;
// ADD R1, R0, R2
		#20
		instruction_initialize_address = 4;
		instruction_initialize_data = 32'b000000_00100_00100_01000_00000_10_0010;
// SUB R8, R4, R4
		#20
		instruction_initialize_address = 8;
```

```
                    instruction_initialize_data = 32'b000000_00101_00110_00111_00000_10_0101;
// OR R7, R5, R6
                    #20
                    instruction_initialize_address = 12;
                    instruction_initialize_data = 32'b101011_00000_01001_00000_00000_00_1100;
// SW R9, 12(R0)
                    #20
                    instruction_initialize_address = 16;
                    instruction_initialize_data = 32'b100011_00000_01100_00000_00000_00_1100;
// LW R12, 12(R0)
                    #20
        instruction_initialize_address = 20;
                    instruction_initialize_data = 32'b000100_00000_00001_00000_00000_00_0001;
// BEQ R0, R1, 1
                    #20
                    instruction_initialize_address = 24;
                    instruction_initialize_data = 32'b000000_00101_00110_00111_00000_10_0101;
// OR R8, R4, R7
                    #20
                    instruction_initialize_address = 28;
                    instruction_initialize_data = 32'b001000_00011_00001_00000_00000_01_0100;
// ADDI R1, R3, 20
```



This waveform shows the ADDI instruction executing at time ~585 ns, with func 0 mapping to my behavioral add operation inside the ALU. The ADDI instruction is taking the value is R3 (30), adding it with the immediate value (20), and storing it in R1. As we can see, the value in R1 successfully changes to the sum (50) at the end of this instruction.

```
                    #20
                    instruction_initialize_address = 32;
                    instruction_initialize_data = 32'b001001_00001_00010_00000_00000_00_1010;
// SUBI R2, R1, 10
                    #20
                    instruction_initialize_address = 36;
                    instruction_initialize_data = 32'b001101_00101_00101_00000_00000_00_0000;
// ORI R5, R5, 0
                    #20
                    instruction_initialize_address = 40;
                    instruction_initialize_data = 32'b001100_00110_00110_11111_11111_11_1111;
// ANDI R6, R6, 16'b1111111111111111
                    #20
                    instruction_initialize_address = 44;
                    instruction_initialize_data = 32'b100000_00110_00101_00000_00000_00_0000;
// MOV R5, R6
                    #20
                    instruction_initialize_address = 48;
                    instruction_initialize_data = 32'b000000_00001_00001_00110_00000_10_0111;
// NOT R6, R1, R1
                    #20
                    instruction_initialize_address = 52;
                    instruction_initialize_data = 32'b000000_00011_00010_01000_00000_10_1010;
// SLT R8, R3, R2
                    #20
       instruction_initialize_address = 56;
                    instruction_initialize_data = 32'b000101_10001_10001_11111_11111_11_1111;
// BNE R17, R17, -1
```

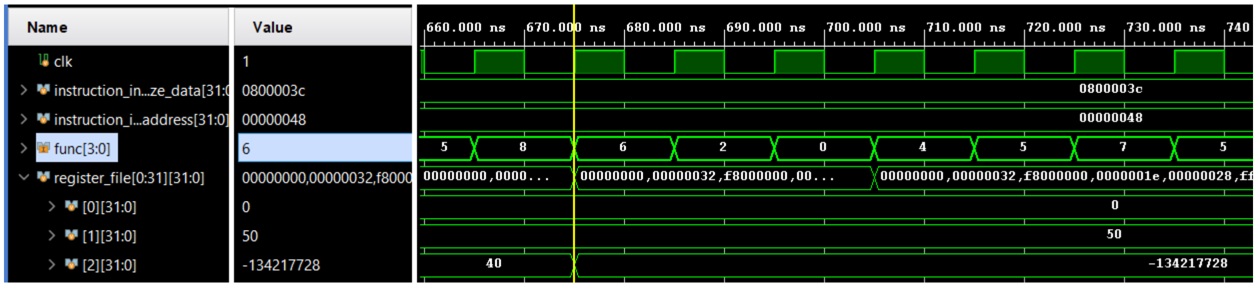


```
Name                              Value
clk                               1
instruction_in...ze_data[31:0]    0800003c
instruction_i...address[31:0]     00000048
func[3:0]                         5
register_file[0:31][31:0]         00000000,00000032,0000
out[31:0]                         1
```

```
640.000 ns    650.000 ns    660.000 ns    670.000 ns    680.000
func:  5 | 7 | 5 | 8 | 6
register_file:  0000000... | 00000000,00000032,00000028,00... | 00000000,000
out:  1 | -134217728 | 0
```

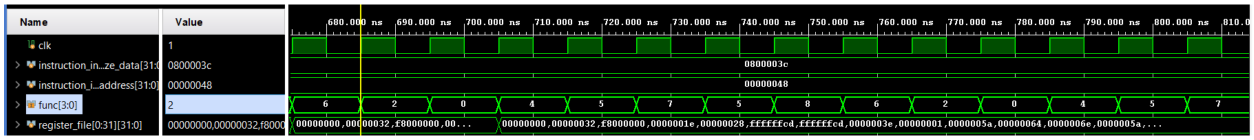
This waveform shows the BNE instruction properly executing at time ~655 ns, with func 7 mapping to the BNE operation inside my ALU. The instruction is comparing R17 with itself, so the ALU outputs 1 because the comparison is equal. Since the ALU output is not zero, the zero flag will remain 0, and the branch will not execute. Instead, the PC increments to the next instruction and continues on to the SLTI operation ahead.

```
                    #20
       instruction_initialize_address = 60;
                    instruction_initialize_data = 32'b001010_00100_01000_00000_00000_11_0010;
// SLTI R8, R4, 50
                    #20
       instruction_initialize_address = 64;
                    instruction_initialize_data = 32'b001111_00000_00010_11111_00000_00_0000;
// LUI R2, 63488
```



This waveform shows the LUI instruction executing at time ~675 ns. This operation maps to func 8 in the ALU, which takes the 16 bit immediate and loads it into the left-most bits of R2 (since registers are 32 bits in this data pipeline, it loads it into the upper 16 bits). As we can see, the immediate value of 63488 (1111100000000000) gets shifted left to the upper 16 bits of the register R2 at time 675 ns, updating the register contents to -134217728 (11111000000000000000000000000000).

```
			#20
	instruction_initialize_address = 68;
			instruction_initialize_data = 32'b000000_00100_00000_00000_00000_00_1000;
// JR R4
```


Name
Value
clk 1
instruction_in...ze_data[31:0] 0800003c
instruction_i...address[31:0] 00000048
func[3:0] 2
register_file[0:31][31:0] 00000000,00000032,f8000
680.000 ns
690.000 ns
700.000 ns
710.000 ns
720.000 ns
730.000 ns
740.000 ns
750.000 ns
760.000 ns
770.000 ns
780.000 ns
790.000 ns
800.000 ns
810.0
0800003c
00000048
6 2 0 4 5 7 5 8 6 2 0 4 5 7
00000000,00000032,f8000000,00...
00000000,00000032,f8000000,0000001e,00000028,ffffffcd,ffffffcd,0000003e,00000001,0000005a,00000064,0000006e,0000005a,...
This waveform shows a repeated jump to the address stored in R4 (40). As we can see, the next ALU func that executes after the jump is func 2, which is an AND/ANDI instruction. If we look back in the testbench, we can see that the ANDI instruction has address 40. The next sequential instructions are fetched again until we reach the JR instruction, where we jump back to the ANDI instruction (func 2) once again.

```
			#20
	instruction_initialize_address = 72;
			instruction_initialize_data = 32'b000010_00000_00000_00000_00000_11_1100;
// J 60
			#20
	instruction_initialize_address = 76;
			instruction_initialize_data = 32'b000000_00001_00001_00010_00000_10_0100;
// AND R2, R1, R1
```