Organization of Digital Computers

EECS 112L

Lab #2: Single Cycle MIPS Processor

Miles Jennings

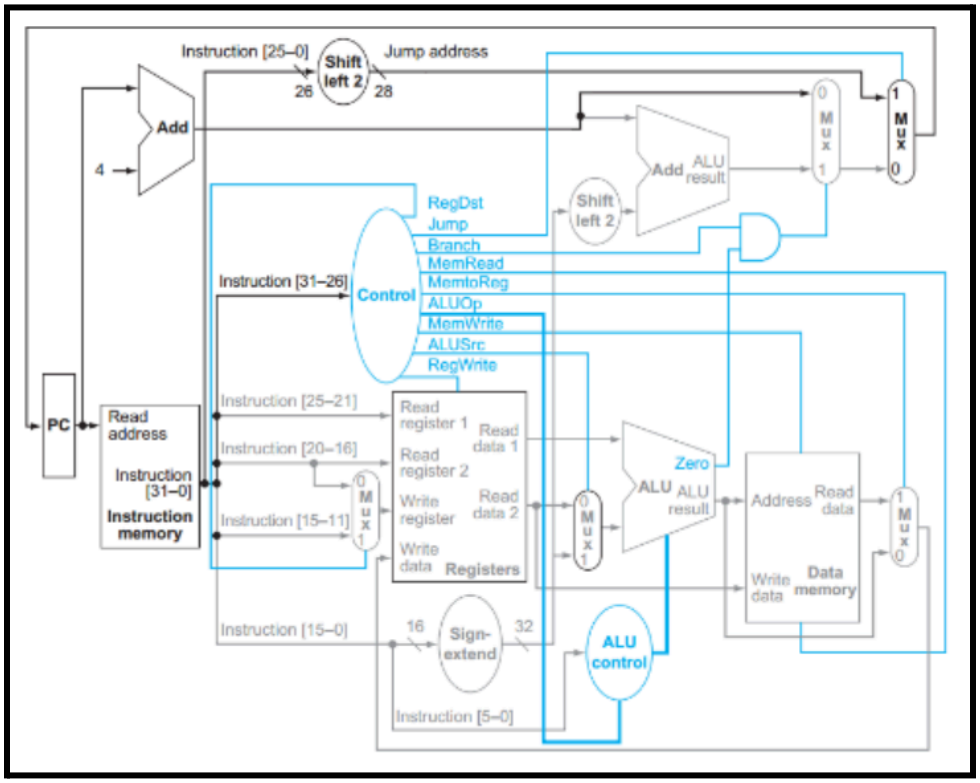
66356019

1/10/2025

1. Objective

In this lab, my objective was to design a modified MIPS processor capable of executing certain R-type and I-type instructions, as well as one J-type instruction. Using Vivado, I implemented and tested key components of the processor, including the Arithmetic Logic Unit (ALU), control, and instruction memory. Testing was conducted using Vivado's simulation tools to verify correct instruction execution.

The processor's control unit serves as the core unit, telling the system what pipelines to follow depending on the wanted instructions. This MIPS processor supports operations such as addition, subtraction, bitwise operations (AND, OR, XOR, NOR), division, multiplication, and shift operations (SLL, SRL, SRA).



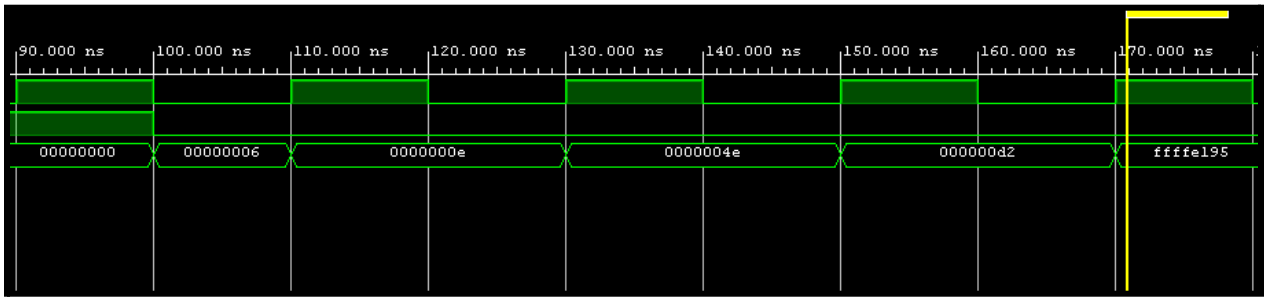
I utilized past labs, lecture notes, and the lab manual to assist in the design of the processor. Testing involved simulating R-type and I-type instructions, ensuring correct data flow between components and verifying expected outputs.

2. Procedure

The very first thing I did for this lab was check what the source code already given to us did. I ran it and saw that it had some operations running correctly.

Name	Format	op	rs	rt	rd	shamt	funct	Comments
add	R	0	reg	reg	reg	0	32	add $s1,$s2,$s3
sub	R	0	reg	reg	reg	0	34	sub $s1,$s2,$s3
addi	I	8	reg	reg	n.a.	n.a.	n.a.	addi $s1,$s2,100
lw	I	35	reg	reg	n.a.	n.a.	n.a.	lw $s1,100($s2)
sw	I	43	reg	reg	n.a.	n.a.	n.a.	sw $s1,100($s2)
and	R	0	reg	reg	reg	0	36	and $s1, $s2, $s3
or	R	0	reg	reg	reg	0	37	or $s1, $s2, $s3

What was initially given in the provided code from the lab manual



The simulation for the given code

Name	Format	op	rs	rt	rd	shamt	funct	Comments
Add	R	0	reg	reg	reg	0	32	add $s1, $s2, $s3
Addi	I	8	reg	reg	n.a.	n.a.	n.a.	addi $s1, $s2, 20
and	R	0	reg	reg	reg	0	36	and $s1, $s2, $s3
andi	I	12	reg	reg	n.a.	n.a.	n.a.	andi $s1, $s2, 20
Beq	I	4	reg	reg	n.a.	n.a.	n.a.	Beq $s1, $s0, L1
Lw	I	35	reg	reg	n.a.	n.a.	n.a.	lw $s1, 20($s2)
Nor	R	0	reg	reg	reg	0	39	nor $s1, $s2, $s3
Or	R	0	reg	reg	reg	0	37	or $s1, $s2, $s3
Slt	R	0	reg	reg	reg	n.a.	42	slt $s1, $s2, $s3
Sll	R	48	reg	reg	reg	Shift amount	0	sll $s1, $s2, 10
Srl	R	48	reg	reg	reg	Shift amount	2	srl $s1, $s2, 10
sra	R	48	reg	reg	reg	Shift amount	3	sra $s1, $s2, 10
Sw	I	43	reg	reg	n.a.	n.a.	n.a.	sw $s1, 20($s2)
Sub	R	0	reg	reg	reg	0	34	sub $s1, $s2, $s3
xor	R	0	reg	reg	reg	0	38	xor $s1, $s2, $s3
Mult	R	0	reg	reg	reg	0	24	mult $s1, $s2, $s3
div	R	0	reg	reg	reg	0	26	div $s1, $s2, $s3
jump	J	2	n.a.	n.a.	n.a.	n.a.	n.a.	J 2500

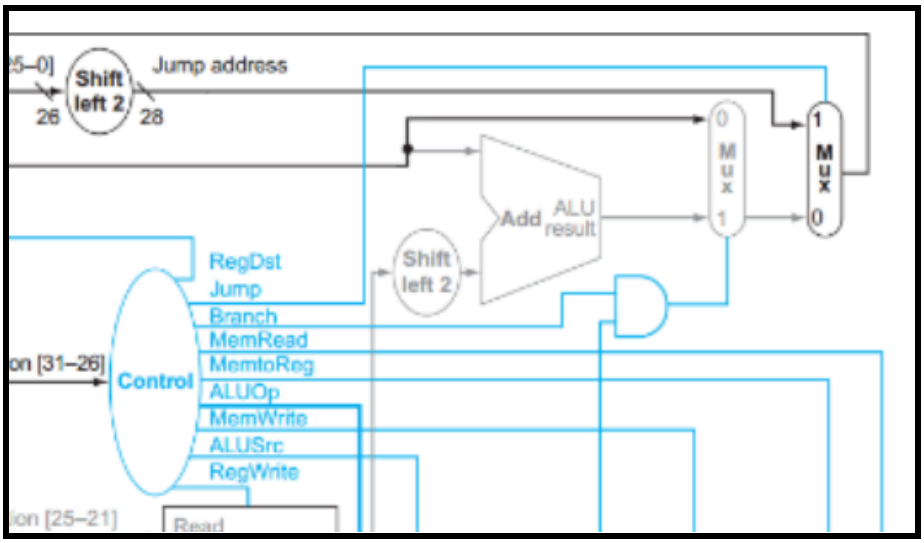
We want to modify the MIPS processor to handle all of these instructions

After reading the instructions and attending the lab, I knew what my next steps would be; I had to implement all of the operations for the MIPS processor. I wanted to get all of the operations that were the same type as the ones already given (R type format) out of the way so that I could follow the same process and better understand how the code works. I did these five first: Nor, Xor, Slt, Mult, and Div first. For these five operations, I only changed ALU.v, ALUControl.v, and Instruction_mem.v to get them to work. Since this was before the test bench was released, I had to create my own tests in instruction_mem.v to test them. I did so by reusing the prompts that were used to test the past ones but changing the instruction codes.

```
                                                    // instruction                 alu

  rom[0]  = 32'b00100000000000010000000000000110; // addi r1,r0,#6
  rom[1]  = 32'b00100000000000100000000000001110; // addi r2,r0,#e
  rom[2]  = 32'b00100000000000110000000001001110; // addi r3,r0,#4e
  rom[3]  = 32'b00100000000001000000000011010010; // addi r4,r0,#d2
  rom[4]  = 32'b00100000000001010111100001100101; // addi r5,r0,#e195
  rom[5]  = 32'b00100000000001101111111111000010; // addi r6,r0,#fe12
  rom[6]  = 32'b00000000001001000011100000100000; // add r7,r1,r4
  rom[7]  = 32'b00000000011001010100000000100000; // add r8,r3,r5
  rom[8]  = 32'b10101100001001110000000000000010; // sw mem[r1+2] <= r7
  rom[9]  = 32'b10101100100010001111111111111110; // sw mem[r4-2] <= r8
  rom[10] = 32'b00000000100000100100100000100010; // sub r9,r4,r2
  rom[11] = 32'b00000000001001010101000000100010; // sub r10,r1,r5
  rom[12] = 32'b10101101001010100000000000000000; // sw mem[r9+0] <= r10
  rom[13] = 32'b00000001001001110101100000100101; // or r11,r9,r7
  rom[14] = 32'b00000001000010100110000000100100; // and r12,r8,r10
  rom[15] = 32'b10001100001011010000000000000010; // r13 =mem[r1+2]
  rom[16] = 32'b10001100100011101111111111111110; // r14 =mem[r4-2]
  rom[17] = 32'b10001101001011110000000000000000; // r15 =mem[r9+0]
  //for all R-type instructions, bits for source registers and destination r
  //for simplicity, keep all destination register as far left register in th
  rom[18] = 32'b000000_01001_00111_10000_00000_100111; // nor r16,r9,r7
  rom[19] = 32'b000000_00001_00010_10001_00000_100110; // xor r17,r1,r2
  rom[20] = 32'b000000_00001_00010_10010_00000_101010; // slt r18,r1,r2
  rom[21] = 32'b000000_00010_00001_10011_00000_101010; // slt r19,r2,r1
  rom[22] = 32'b000000_00001_00010_10100_00000_011000; // mult r20,r1,r2
  rom[23] = 32'b000000_00011_00010_10001_00000_011010; // div r21,r3,r2
nd
```

After I did those five, I went on to do the rest of the R type instructions, the shift operations(SLL, SRL, SRA). In order to implement these instructions, I had to change ALUControl.v, ALU.v, and Control.v. I knew I had to make additions to ALUControl.v and ALU.v because the past R operations needed changes in those files, however, I also figured that since control.v did not have the opcode for these shift operations, the MIPS processor would not run correctly without it. Additionally, I was able to test it using the test bench given to us at a later date which saved a lot of time.

Finally, I implemented the last of the operations, the I-types and the one J-type. This consisted of Beq, Andi, and Jump. These were some of the hardest operations to implement and they also took me the longest. Along with the other files I have been changing for each new instruction, I had to also change datapath.v.(Cont. to next page)

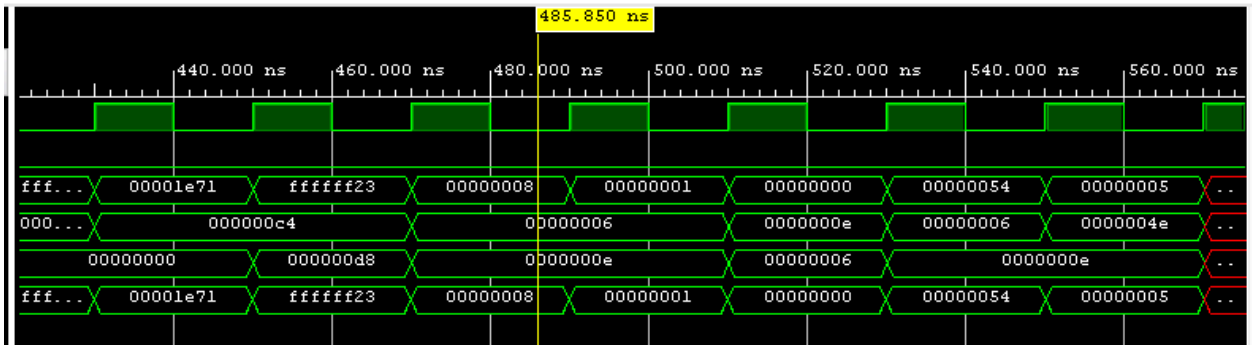


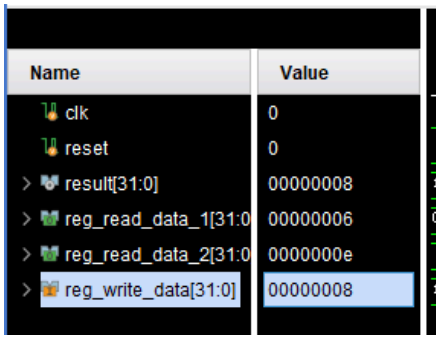
By using this datapath map, I was able to make muxes and map them to the paths needed by branch and jump instructions. I did so by following the bus wires from Branch and Jump and seeing what components they went through. I saw that we had all of the components except for the two muxes used by the two operations. By continuing to follow all of the wires entering and leaving each mux, I was able to determine the inputs, selects, and outputs needed.

3. Simulation Results

Added ALU-Operations

First 5 Added Operations: mult, div, slt, nor, xor (More details below)





Registers/Values

Changes to ALU.v – Now handles logic when the ALU control code is called

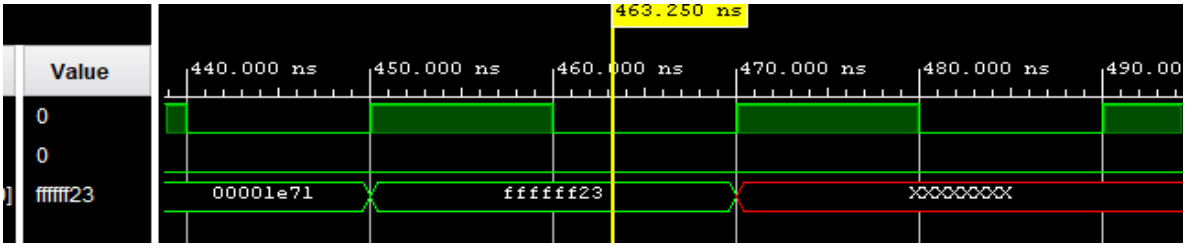
```
4'b1100: alu_result = ~(a | b); //added: nor
4'b0100: alu_result = a ^ b; //added: xor
4'b0010: alu_result = a + b; // add
4'b0110: alu_result = a - b; // sub
4'b0101: alu_result = a * b; //added: mult
4'b1011: alu_result = a / b; //added: div
```

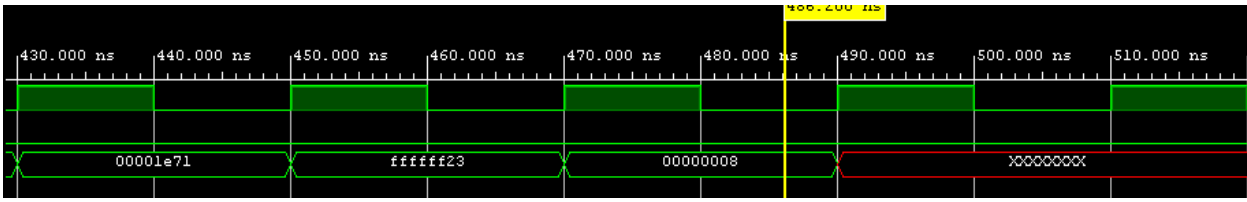
Changes to instruction_mem.v – Created Test Cases to check if the operations were doing what they were supposed to.

```
rom[18] = 32'b000000_01001_00111_10000_00000_100111; // nor r16,r9,r7          ffffff23          r16=ffffff23
rom[19] = 32'b000000_00001_00010_10001_00000_100110; // xor r17,r1,r2          00000008          r17=00000008
rom[20] = 32'b000000_00001_00010_10010_00000_101010; // slt r18,r1,r2          00000001          r18=00000001
rom[21] = 32'b000000_00010_00001_10011_00000_101010; // slt r19,r2,r1          00000000          r19=00000000
rom[22] = 32'b000000_00001_00010_10100_00000_011000; // mult r20,r1,r2         00000054          r20=00000054
rom[23] = 32'b000000_00011_00010_10001_00000_011010; // div r21,r3,r2          00000008          r21=00000008
```

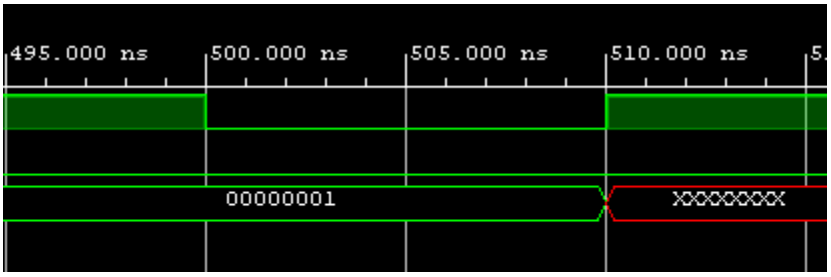
Changes to ALUControl.v – Depending on what ALUop code was given, it would then run that logic from ALU.v

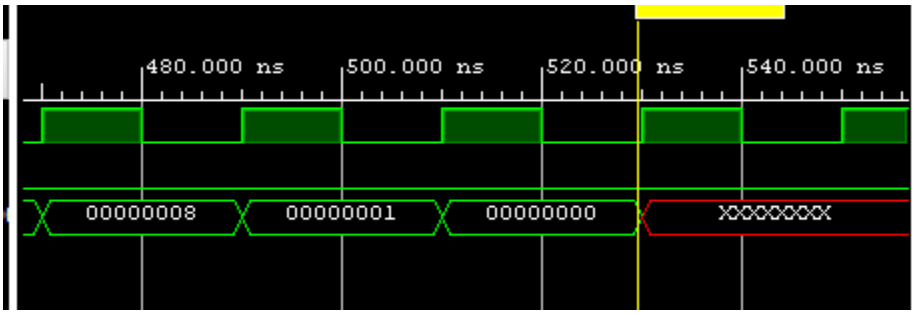
```
8'b10100111: ALU_Control=4'b1100;   //added: nor
8'b10100110: ALU_Control=4'b0100;   //added: xor
8'b00xxxxxx: ALU_Control=4'b0010;   //add imediate
8'b10100000: ALU_Control=4'b0010;   //add
8'b10100010: ALU_Control=4'b0110;   //subtract
8'b10011000: ALU_Control=4'b0101;   //added: multiply
8'b10011010: ALU_Control=4'b1011;   //added: division
```

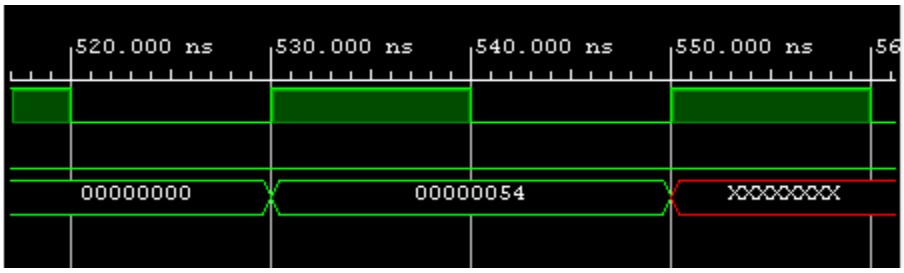
- **Nor(ALU_Control = 4'b1100):** Negates Or; Compares bit by bit between two different 32 bit registers and outputs a 1 if both corresponding bits are 0 and a 0 otherwise.
  - 

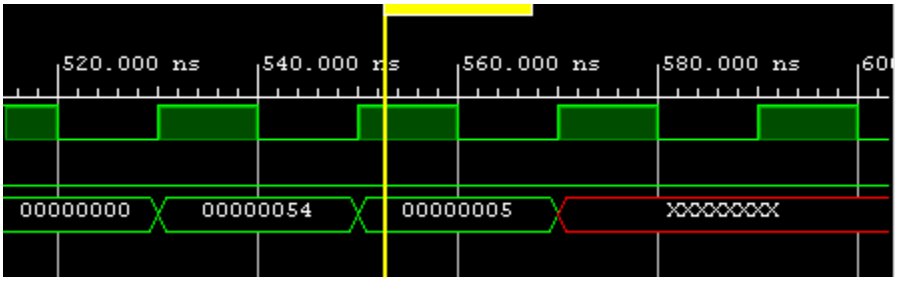
  - Inputs: r9 = C4, r7 = d8, Output: r16 = ffffff23
  - Note: For past Or operation test, Output = 000000dc
  - For comparison:
    - OR r9 r7 = 0000 0000 0000 0000 0000 0000 1101 1100
    - NOR r9 r7 = 1111 1111 1111 1111 1111 1111 0010 0011
- **Xor(ALU_Control = 4'b0100):** Exclusively OR; Compares two registers bitwise and outputs 1 when each corresponding bit is different. Outputs 0 when each corresponding bit is the same.
  - 

  - Inputs: r1 = 6, r2 = e, Output: r17 = 8
- **Slt(ALU_Control = 4'b0111):** Set less than; Compares two registers and outputs 1 if r1 < r2.

- 

  - Inputs: r1 = 6, r2 = e, Output: r18 = 1
- 

  - Inputs: r2 = e, r1 = 6, Output: r19 = 0
- **Mult(ALU_Control = 4'b0101):** Multiplies r1 and r2
- 

  - Inputs: r1 = 6, r2 = e, Output: r20 = 54(Hex)
  - 6 * 14 = 84 → 1010100 → 0101 0100 → 54
- **Division(ALU_Control = 4'b0101):** Divides r4 and r2
- 

  - Inputs: r3 =4e , r2 = e, Output: r21 = 5(Hex)
  - 4e / e = 5 remainder 8 → floor → 5

Adding the Shift operations (SLL, SRL, SRA)

I ran into an issue while implementing the shift operations. SLL and SRL would pass every test case except for the 3rd one. The values in the registers/addresses were correct, but it would still say failed.

Change in the ALUControl.v - Determines which operation is doing

```
8'b10000000: ALU_Control=4'b1000;   //added: Sll - shift left logical
8'b10000010: ALU_Control=4'b1001;   //added: Srl - shift right logical
8'b10000011: ALU_Control=4'b1010;   //added: sra - shift right arithmetic
```

Change in the ALU.v - Logic behind the operation

```
4'b1000: alu_result = a << b[10:6]; //added: Sll - shift left logical
4'b1001: alu_result = a >> b[10:6]; //added: Srl - shift right logical
4'b1010: alu_result = $signed(a) >>> b[10:6]; //added: sra - shift right arithmetic
```

Change in the control.v - Now handles the opcode of the shift operations

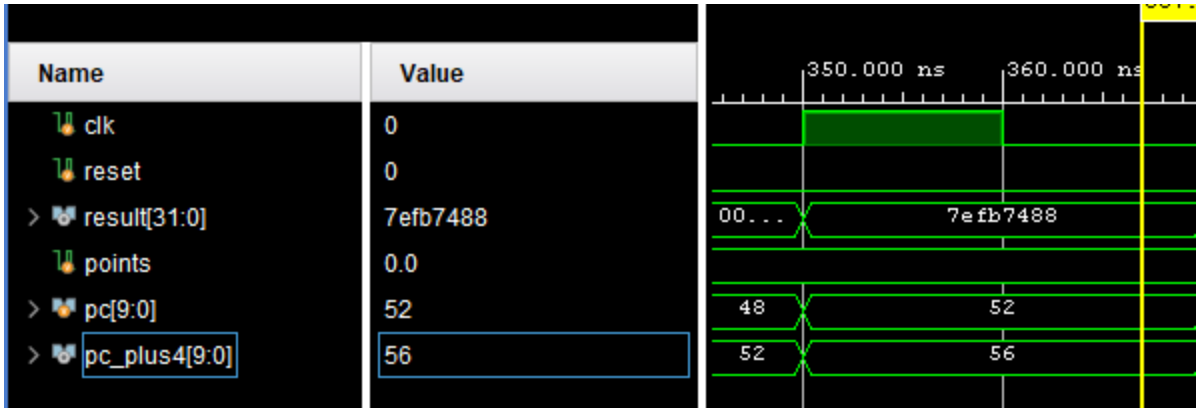
```
6'b110000:
begin //sll, srl, sra
      reg_dst = 1'b1;
      mem_to_reg = 1'b0;
      alu_op = 2'b10;
      mem_read = 1'b0;
      mem_write = 1'b0;
      alu_src = 1'b1;
      reg_write = 1'b1;
 end
```

Successful Test Cases:

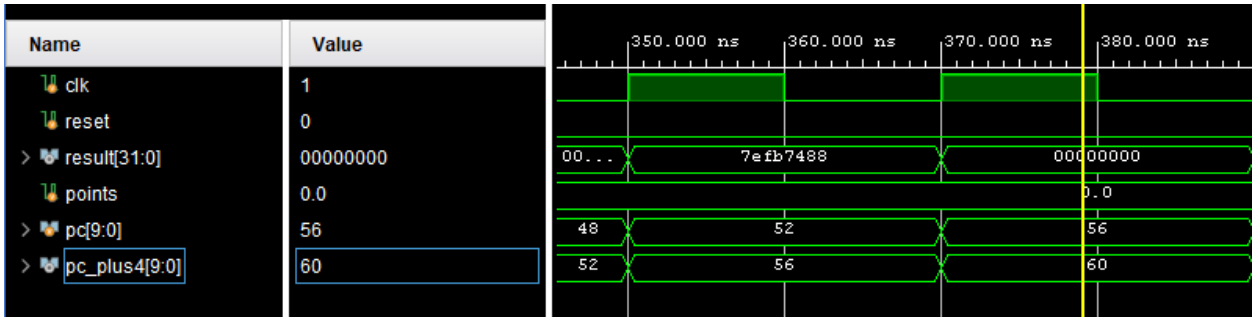
```
if(uut.datapath_unit.data_mem.ram[14]==32'h7efb7488) begin $display("SLL  1       success!\n");
if(uut.datapath_unit.data_mem.ram[15]==32'h00000000) begin $display("SRL  1       success!\n");
if(uut.datapath_unit.data_mem.ram[16]==32'hfe400000) begin $display("SRA  1       success!\n");
```

Simulation results(SLL, SRL, SRA)

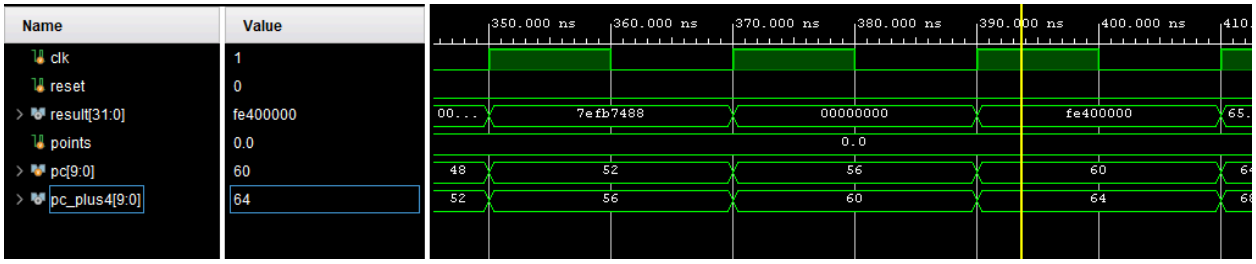
*SLL: PC = 14*4 = 56, Expected output = 32'h7efb7488*



*SRL: PC = 15*4 = 60, Expected output = 32'h00000000*



*SRA: PC = 16*4 = 64, Expected output = 32'hfe400000*



Adding the Andi operation (And-Immediate)

Change in the control.v - Now handles the opcode of the andi operation

```
6'b001100:
begin //andi
     reg_dst = 1'b0;
     mem_to_reg = 1'b0;
     alu_op = 2'b11;
     mem_read = 1'b0;
     mem_write = 1'b0;
     alu_src = 1'b1;
     reg_write = 1'b1;
end
```

Change in the ALUControl.v - Now when 11xxxxxx is called with any following immediate value(the value of the x's), And operation logic is used

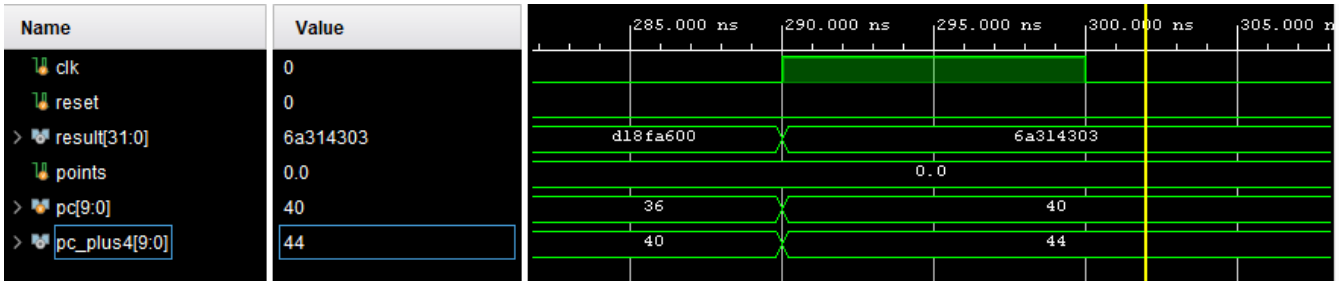
```
8'b11xxxxxx: ALU_Control=4'b0000;   //added: andi - and with an immediate value - utilizes past and logic
```

Successful Test Cases:

```
if(uut.datapath_unit.data_mem.ram[11]==32'h6a314303) begin $display("ANDI 1       success!\n");
```

Simulation results(andi)

andi:PC = 11*4 = 44, Expected output = 32'h6a314303



Adding the Branch and Jump operation (Beq and jump)

Change in the control.v - Now handles the opcode for beq and jump

```
//added New opcode
6'b000100:
begin // beq
      reg_dst = 1'b0;
      mem_to_reg = 1'b0;
      alu_op = 2'b01;
      mem_read = 1'b0;
      mem_write = 1'b0;
      alu_src = 1'b0;
      reg_write = 1'b0;
      branch = 1'b1;
      jump = 1'b0;
 end
 //added New opcode
 6'b000010:
 begin // jump
      reg_dst = 1'b0;
      mem_to_reg = 1'b0;
      alu_op = 2'b00;
      mem_read = 1'b0;
      mem_write = 1'b0;
      alu_src = 1'b0;
      reg_write = 1'b0;
      branch = 1'b0;
      jump = 1'b1;
end
```

Changes in the datapath.v - Had to create new mux components inside of datapath

I declared new wires so that the muxes I am creating have the correct inputs/outputs.

```
 //added wires for branch
 wire beq_sel;   //added to be the select value of branch
 assign beq_sel = (branch & zero);
 wire [9:0] beq_res;
 wire [9:0] branch_to;
 assign branch_to = (imm_value[9:0] << 2) + pc_plus4;

 //added wires for jump
 wire [27:0] jump_to;
 assign jump_to = (instr[25:0] << 2);
 wire [9:0] jump_res;
```

In order to create new muxes using this format(Given by lab manual download):

```
module mux2 #(parameter mux_width= 32)
(   input [mux_width-1:0] a,b,
    input sel,
    output [mux_width-1:0] y
    );

    assign y = sel ? b : a;
endmodule
```

Below are the muxes I created.

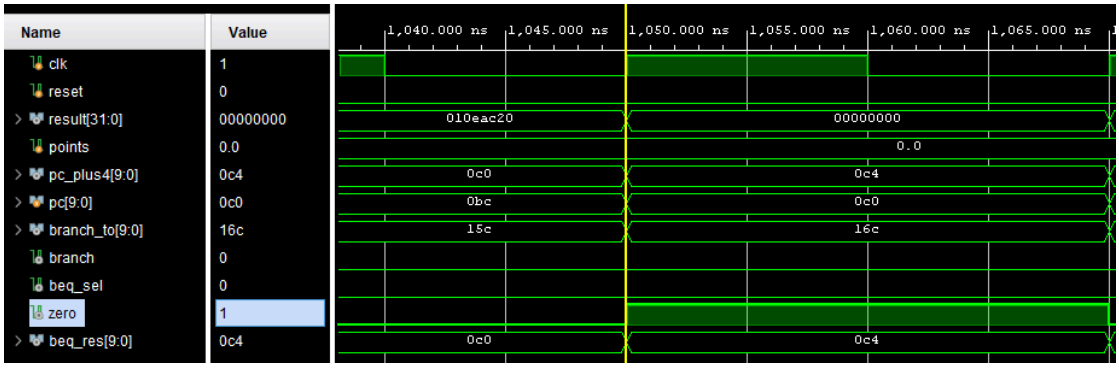
```
//add 1 mux for branch to work
mux2 #(.mux_width(10)) beq_mux
(   .a(pc_plus4),        //if sel 0
    .b(branch_to),        //if sel 1
    .sel(beq_sel),       //select based on if branch & zero = 1
    .y(beq_res));

//add 1 mux jump to work
mux2 #(.mux_width(10)) jump_mux
(   .a(beq_res),
    .b(jump_to[9:0]),
    .sel(jump),       //
    .y(jump_res));
```

Successful test cases:

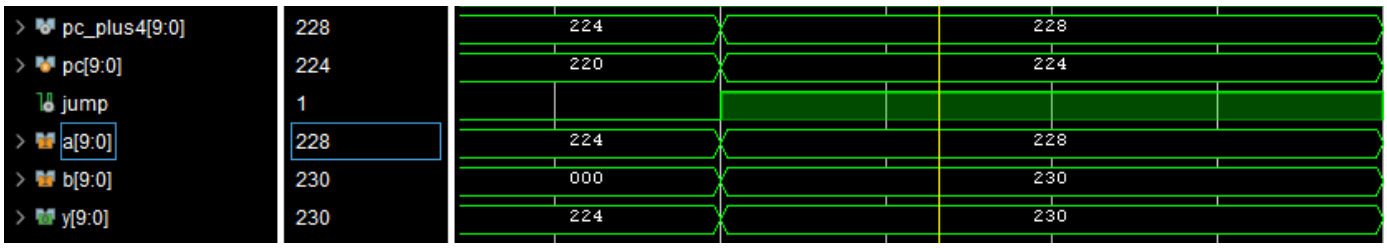
Zero = 1; Beq_sel = 0; branch = 0; pc_plus4 = beq_res

A[9:0] = 228; B[9:0] = 230; sel(Jump) = 1; y = B

Simulation results(beq and jump)



This is the correct result because since zero is 1 but beq_sel does not equal 1, it does not activate the beq_sel and because of this you can see that beq_res = pc + 4 because the branch did not trigger. This is because the inputs were not equal to each other.



This is the correct result since the output is = to the input b which is our target jump address.

Passing all test cases:

```
run 10 us
ANDI 1      success!
NOR  1      success!
SLT  1      success!
SLL  1      success!
SRL  1      success!
SRA  1      success!
XOR  1      success!
MULT 1      success!
DIV  1      success!
ANDI 2      success!
NOR  2      success!
SLT  2      success!
SLL  2      success!
SRL  2      success!
SRA  2      success!
XOR  2      success!
MULT 2      success!
DIV  2      success!
ANDI 3      success!
NOR  3      success!
SLT  3      success!
SLL  3      success!
SRL  3      success!
SRA  3      success!
XOR  3      success!
MULT 3      success!
```

```
DIV  3      success!
ANDI 4      success!
NOR  4      success!
SLT  4      success!
SLL  4      success!
SRL  4      success!
SRA  4      success!
XOR  4      success!
MULT 4      success!
DIV  4      success!
ANDI 5      success!
NOR  5      success!
SLT  5      success!
SLL  5      success!
SRL  5      success!
SRA  5      success!
XOR  5      success!
MULT 5      success!
DIV  5      success!
BEQ  1      success!
BEQ  2      success!
BEQ  3      success!
BEQ  4      success!
BEQ  5      success!
j  1        success!
j  2        success!
j  3        success!
```

```
j  4        success!
j  5        success!
points : 70
```

6 Midterm Winter 2021: Synchronous vs, Asynchronous Design

You want to design a data memory that is controlled as follows

- Write to the memory is controlled synchronously where the data at the input (write_data) is written in the location specified by the valid memory address (access_addr) when wr_en is active high. (4 points for syntax and correct design)
- At any moment, whenever read is activated (rd_en is active high) with a valid memory address (access_addr), the data is read from the memory and is outputted on read_data. (4 points for syntax and correct implementation)


```
module data memory (
        input clk ,
        input [31:0] access_addr ,
        input [31:0] write_data ,
        input wr_en ,
        input rd_en ,
        output reg [31:0] read_data);
        reg [31:0] ram [255:0]; // The size of your data memory.          //changed to reg to handle manipulation of
        indexing
        // Define your module behavior here.

//whenever rden is active high; when posedge of clock, update valid mem address with write_data
//handles synchronous write to memory
        Always @(posedge clk)
          Begin
            if (wr_en)
                begin
                ram[access_addr[7:0]] <= write_data;          //index from [7:0] bec
                                                  //ram is 256 bits→only need 8 bits to address [255:0] because 2^8 = 256
                end
          End

//handles at any moment case = asynchronous → use Always @(*)
//rd_en is active high, look at valid address and read data from there and output to read_data

        Always @(*)
          Begin
                if(rd_en)
                  Begin
```

```
                Read_data = ram[access-addr[7:0]]
            End
        else    //need to handle while rd_en is not active, we dont want it to be updating Read_data if
                                                                                    //rd_en is not on
            Begin
                Read_data = 32'b0;          //sets to 32'b0 if rd_en is disabled
            End
    end
Endmodule
```

Questions:

N/A