Digital Harware Systems at low logic level I

Nikolaos Karapatsias 10661

Part 1

1) Arithmetic Logical Unit Module (alu.v)

- *Verilog:*

  - Starting by implementing our module's input output ports as requested:

```
module alu (
    input wire [31:0] op1, op2,    // Input number 1 and 2
    input wire [3:0] alu_op,    // alu operation
    output reg zero,            //output zero
    output reg [31:0] result// Output result
);
```

  - Then we are decoding each of 9 functions we want to execute with our module using "parameter" :

```
parameter [3:0] ALUOP_AND=4'b0000, ALUOP_OR=4'b0001, ALUOP_ADD=4'b0010, ALUOP_SUB=4'b0110, ALUOP_LESS_THAN=4'b0100,
ALUOP_SLIDE_R=4'b1000, ALUOP_SLIDE_L=4'b1001, ALUOP_ASLIDE=4'b1010, ALUOP_XOR=4'b0101;
```

  - From there, we are creating a 9 to 1 MUX using "case" loop which is expecting as an input the 4bit operation code, and sets the output as the result of the desired function. Note that all the reg updates of "Zero" and "Result" are done synchronously whenever an input signal changes.

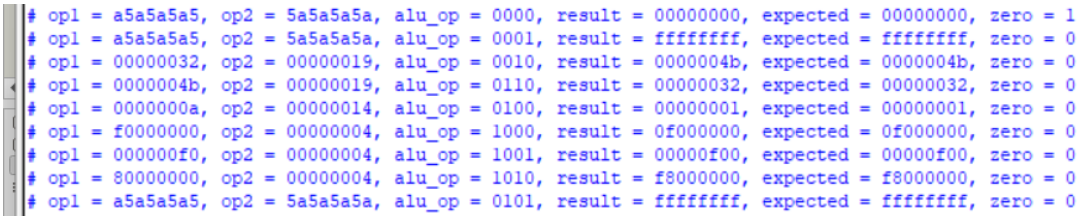
```
always @(*) begin
    result <= 32'b0;
    zero <= 1'b0;
    case(alu_op)
        ALUOP_AND: result <= (op1&op2); //AND
        ALUOP_OR: result <= (op1|op2); //OR
        ALUOP_ADD: result <= (op1+op2); //ADDITION
        ALUOP_SUB: result <= (op1-op2); //SUBTRACT
        ALUOP_LESS_THAN: result <= (op1 < op2) ? 32'b1 : 32'b0; //OP1<OP2
        ALUOP_SLIDE_R: result <= (op1>>op2[4:0]); // op1>>op2[4:0]
        ALUOP_SLIDE_L: result <= (op1<<op2[4:0]); //op1<<op2[4:0]
        ALUOP_ASLIDE: result <= ($unsigned($signed(op1)>>>op2[4:0])); //op1>>>op2[4:0]
        ALUOP_XOR:  result <= (op1^op2); //op1^op2
        default: result <= 32'b0;
    endcase
```

- Last but not least we are setting the "Zero" register either to 0 or 1 following the current result:

```
zero <= (result == 32'b0) ? 1'b1 : 1'b0;
```

- *Results:*

As we can see, our module produces the expected values:



```
# op1 = a5a5a5a5, op2 = 5a5a5a5a, alu_op = 0000, result = 00000000, expected = 00000000, zero = 1
# op1 = a5a5a5a5, op2 = 5a5a5a5a, alu_op = 0001, result = ffffffff, expected = ffffffff, zero = 0
# op1 = 00000032, op2 = 00000019, alu_op = 0010, result = 0000004b, expected = 0000004b, zero = 0
# op1 = 0000004b, op2 = 00000019, alu_op = 0110, result = 00000032, expected = 00000032, zero = 0
# op1 = 0000000a, op2 = 00000014, alu_op = 0100, result = 00000001, expected = 00000001, zero = 0
# op1 = f0000000, op2 = 00000004, alu_op = 1000, result = 0f000000, expected = 0f000000, zero = 0
# op1 = 000000f0, op2 = 00000004, alu_op = 1001, result = 00000f00, expected = 00000f00, zero = 0
# op1 = 80000000, op2 = 00000004, alu_op = 1010, result = f8000000, expected = f8000000, zero = 0
# op1 = a5a5a5a5, op2 = 5a5a5a5a, alu_op = 0101, result = ffffffff, expected = ffffffff, zero = 0
```

2) Calc_enc Module (calc_enc.v):

- *Verilog:*
  - In this module we are implementing the creation of the operation code used to control alu module using structural verilog. We are doing that by creating 4 different gate logic circuit's for each of the 4 bits of alu_op. To achieve that, we are using 3 buttons named btnc, btnl and btnr as declared below:

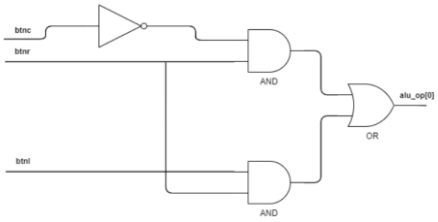
```
module calc_enc (
input wire btnc, btnl, btnr,
output wire [3:0] alu_op);
```

  - Then we are making the required declarations and assignments of wires in order for our project to work properly.

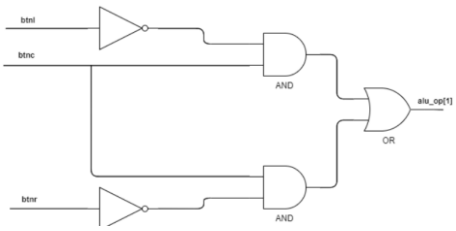
```
wire n1_result_0, a1_result_0, a2_result_0, result_0; //result declarations for alu_op[0] gates
wire n1_result_1, n2_result_1, a1_result_1, a2_result_1, result_1; //result declarations for alu_op[1] gates
wire n1_result_2, n2_result_2, a1_result_2, a2_result_2, a3_result_2, result_2; //result declarations for alu_op[2] gates
wire n1_result_3, n2_result_3, a1_result_3, a2_result_3, a3_result_3, a4_result_3, result_3; //result declarations for alu_op[3] gates


assign alu_op[0] = result_0;
assign alu_op[1] = result_1;
assign alu_op[2] = result_2;
assign alu_op[3] = result_3;
```

- From there, we are ready to implement logic for each of the 4 bits:

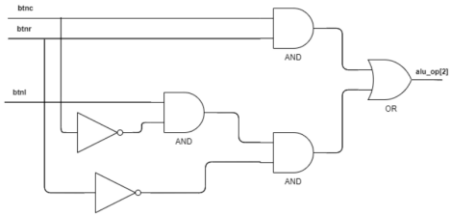
```
// alu_op[0]

not n1_0 (n1_result_0, btnc);
and a1_0 (a1_result_0, n1_result_0, btnr);
and a2_0 (a2_result_0, btnr, btnl);
or o1_0 (result_0, a1_result_0, a2_result_0);
```



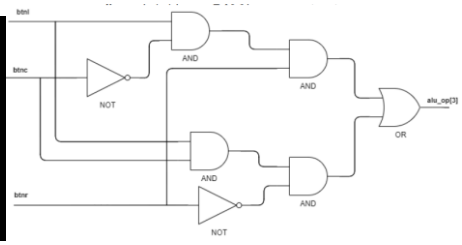

```
// alu_op[1]

not n1_1 (n1_result_1, btnl);
and a1_1 (a1_result_1, n1_result_1, btnc);
not n2_1 (n2_result_1, btnr);
and a2_1 (a2_result_1, n2_result_1, btnc);
or o1_1 (result_1, a1_result_1, a2_result_1);
```




```
// alu_op[2]

and a1_2 (a1_result_2, btnc, btnr);
not n1_2 (n1_result_2, btnc);
and a2_2 (a2_result_2, n1_result_2, btnl);
not n2_2 (n2_result_2, btnr);
and a3_2 (a3_result_2, n2_result_2, a2_result_2);
or o1_2 (result_2, a1_result_2, a3_result_2);
```




```
// alu_op[3]
not n1_3 (n1_result_3 ,btnc);
and a1_3 (a1_result_3, n1_result_3, btnl);
and a2_3 (a2_result_3, a1_result_3, btnr);
and a3_3 (a3_result_3, btnc, btnl);
not n2_3 (n2_result_3,  btnr);
and a4_3 (a4_result_3, a3_result_3, n2_result_3);
or o1_3  (result_3, a2_result_3, a4_result_3);
```



- *Result table:*

BTNL	BTNC	BTNR	ALU_OP
0	0	0	0000 AND
0	0	1	0001 OR
0	1	0	0010 ADDITION
0	1	1	0110 SUBTRACT
1	0	0	0100 LESS THAN
1	0	1	1001 SLL
1	1	0	1010 SRL
1	1	1	0101 XOR

3) Calc Module (calc.v):

- *Verilog:*
  - Firstly, we are declaring our input/output ports that are requested.

```
module calc ( input wire clk, btnc, btnl, btnu, btnr, btnd, input wire [15:0] sw, output wire [15:0] led );
```

  - From there, we are setting some registers and wires that will help us to implement this module. Note that result, zero, op1 and op2 are the wires that connects the alu to our current module. The accumulator register is used as described in the pdf file.

```
reg [15:0] accumulator = 16'b0;
wire [31:0] result;
wire [31:0] op1, op2;
wire [3:0] alu_op;
wire zero;
```

  - Then, we are making some assignments as described. *Op1* gets the value stored in the accumulator using a sign extension to transpose it from 16bits to 32bits. The same tactic is used for *Op2* which gets the value of switch using sign extension again.

```
assign op1 = {{16{accumulator[15]}}, accumulator};
assign op2 = {{16{sw[15]}}, sw};
assign led = accumulator; //Led is assigned with the accumulators value
```

  - In the next step, we are installing the submodules alu and calc_enc (described later)

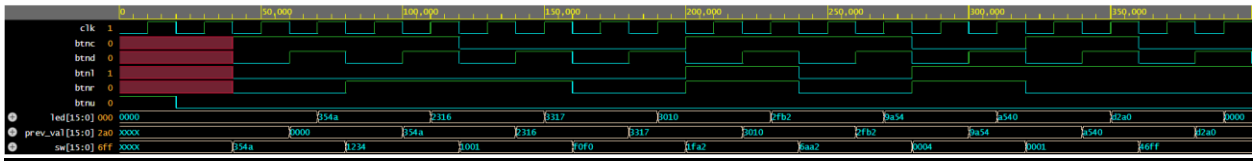
```
calc_enc my_calc_enc (.btnc(btnc), .btnl(btnl), .btnr(btnr), .alu_op(alu_op));

alu my_alu(.op1(op1), .op2(op2), .alu_op(alu_op), .zero(zero), .result(result));
```

- Last but not least, we are updating our accumulator value synchronously with a positive edge of the clock, either with 0 when btnu button is pressed or the result of alu when btnd one is pressed.

```
always @(posedge clk) begin
    if (btnu) begin
        accumulator<=16'b0; //Accumulator zeroed synchronously with btnu
        end

    if (btnd) begin
        accumulator <= result[15:0]; //updating accumulator synchronously
        end                          //with alu result

end
```

- *Result table and waveform:*

Input(btnl, btnc, btnr)	PREVIOUS VALUE(ACC)	SWITCHES(INPUT)	RESULT
xxx	xxxx	xxxx	0000
010	0000	354a	354a
011	354a	1234	2316
001	2316	1001	3317
000	3317	f0f0	3010
111	3010	1fa2	2fb2
010	2fb2	6aa2	9a54
101	9a54	0004	a540
110	a540	0001	d2a0
100	d2a0	46ff	0000



4) Regfile Module (regfile.v):

- *Verilog:*
  - Fristly we are initializing our input/output ports of our module as described in the instructions

```
module regfile #(parameter DATAWIDTH = 32)
( input wire clk , write,
  input wire [4:0] readReg1, readReg2, writeReg,
  input wire [DATAWIDTH-1:0] writeData,
  output reg [DATAWIDTH-1:0] readData1,readData2
);
```

- Then we are creating a system of 32 registers where each can store 32 bits using a for loop inside an initial begin block

```
integer i;
reg [DATAWIDTH-1:0] registers [31:0];

initial begin
    for(i=0; i<32; i=i+1)
        registers[i] = 32'b0;
end
```

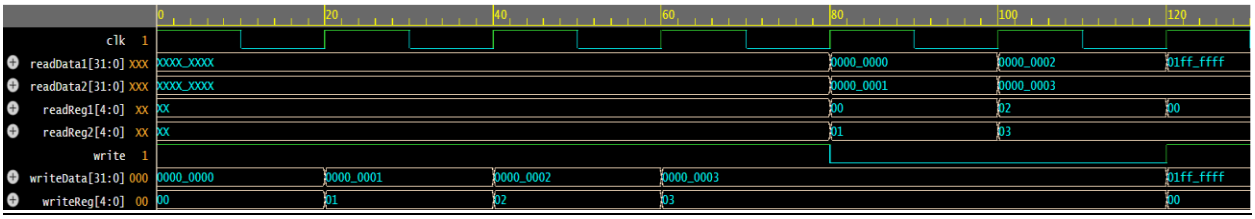
- And finally we are implementing the logic of our registers file. Firstly, we are always reading synchronously the two values of the given "readRegisters" 1 and 2 if the write signal is 1 we are also allowing our module to write "writeData" to the location specified by "writeReg". Note that if there is a conflict between the read and write registers, we are prioritizing the writing function by setting it asynchronously (=) and not synchronous (<=) as the other registers. In that case, the output of the requested read data will be the input of "writeData" port of the same clock cycle.

```
always @(posedge clk) begin
    if (write)
        begin
            if (writeReg == readReg1) //Giving priority to writing data
                begin
                registers[writeReg] = writeData; // Asychronous write as the lecturer noted
                readData1 <= registers[readReg1];
                readData2 <= registers[readReg2];
                end
            else if (writeReg == readReg2) //Giving priority to writing data
                begin
                registers[writeReg] = writeData;
                readData1 <= registers[readReg1];
                readData2 <= registers[readReg2];
                end
            else
                begin
                registers[writeReg] <= writeData;
                readData1 <= registers[readReg1];
                readData2 <= registers[readReg2];
                end

        end
    else
        begin
            readData1 <= registers[readReg1];
            readData2 <= registers[readReg2];
        end
end
```

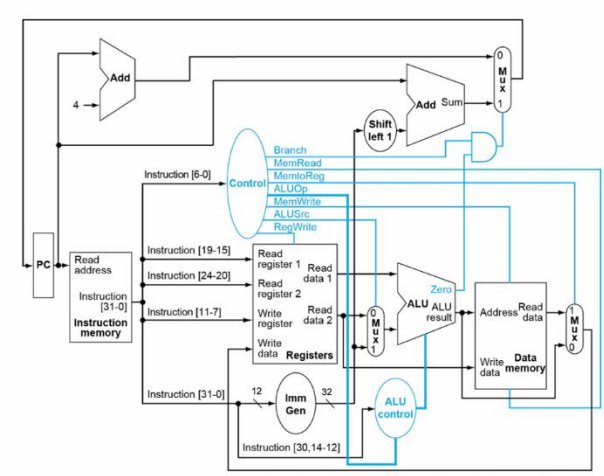
- *Results:*

The output waveform is shown in the below image. Firstly, we enable write signal and setting the first 4 registers of the regfile module as 32'b0, 32'b1, 32'b10, and 32'b11 respectively. This is done in the first 4 cycles (60 ns). Afterwards, we disable the write data signal and read the values stored in the first 4 registers. As you can see, the values match the ones we loaded in the previous step. Finally, on the 7th cycle (120 ns) we are enabling again writing, and on the same cycle we try to read from the same register. The value of the

register we read is the same as the value of “writeData” we loaded on the same cycle, 01ff_ffff as expected.



Part2



1) Datapath Module (datapath.v):

- *Verilog:*
  - Firstly we are declaring our modules input/output ports and also some registers an wires to help us implement the datapath logic.

```
module datapath #(parameter [31:0] INITIAL_PC = 32'h00400000, parameter DATAWIDTH = 32)
(
input wire clk, rst,
input wire [31:0] instr,
input wire PCSrc, ALUSrc, RegWrite, MemToReg,
input wire [3:0] ALUCtrl,
input wire loadPC,
output reg [31:0] PC,
output wire Zero,
output reg [31:0] dAddress, dWriteData, WriteBackData,
input wire [31:0] dReadData
);


reg [31:0] imm, op2_reg, op1_reg;
wire [31:0] alu_result, readData1, readData2;
reg [4:0] readReg1, readReg2, writeReg;
```

  - After that, we are installing the required submodules Alu and Regfile analyzed previously.

```
regfile my_regfile ( .clk(clk) , .write(RegWrite), .readReg1(readReg1), .readReg2(readReg2),
.writeReg(writeReg), .writeData(WriteBackData),
.readData1(readData1) ,.readData2(readData2)); // Register Instatation


alu my_alu (.op1(op1_reg), .op2(op2_reg), .alu_op(ALUCtrl)  , .zero(Zero), .result(alu_result)); //ALU Instatation
```

- Also we are initializing the Program counter to 0x00400000.

```
initial
    begin
        PC <= INITIAL_PC; // Program counter initialization
    end
```

- Now we are ready to start implementing the datapath logic. Firstly, we are setting a restet parameter to restart our system whenever we want to. We are also loading the Programm counter with the next address of Rom whenever is asked from the signal "loadPC". "PCSrc" signal defines the step by which we will update the value of Programm Counter. That signal is high "1" only on BEQ type of instructions and sets the step equal to the immediate value specified by the instruction. In all the other instructions that signal remains low.

```
always @(posedge clk) //Program Counter (PC),  Branch Target
begin
    if (rst)
        begin
            PC <= INITIAL_PC;
        end

    else
        if (loadPC)
            begin
                if(PCSrc)
                    begin
                        PC = PC + {{20{instr[31]}}, instr[31], instr[7], instr[30:25], instr[11:8]}; //PC + branch_offset (we know that is B type of insrt and so the imm)
                    end
                else
                    begin
                        PC = PC+4; //PC+4
                    end
            end
    end
```

- In that step we are calculating the immediate value of the given instruction. Note that there are four basic types of instructions in RiscV architecture: R, I, B and S and each format is shown below:

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type

Following that, we are calculating the immediate value of the current instruction each time using sign extension:

```
always @(posedge clk) // Immediate Generation-ALU
  begin
    if (instr[6:0] == 7'b0010011) // I type of instructions
      begin
        imm <= {{20{instr[31]}},instr[31:20]};
      end

    if (instr[6:0] == 7'b0000011) // LW type of instructions
      begin
        imm <= {{20{instr[31]}},instr[31:20]};
      end
    if (instr[6:0] == 7'b0100011) // S type of instructions
      begin
        imm <= {{20{instr[31]}}, instr[31:25], instr[11:7]};
      end

    if (instr[6:0] == 7'b1100011) // B type of instructions
      begin
        imm <= {{20{instr[31]}}, instr[31], instr[7], instr[30:25], instr[11:8]};
      end
```

- Then, we are updating the Operators of the ALU following the main graph in the beginning of Part 2. We are assigning the output of readData1 of registers to the first operator (Op1) of ALU. Also we are using a 2 to 1 MUX to define which signal to load on the second operator (Op2). This is decided by the value of signal ALUSrc. Whenever this signal is low we are loading the second output of registers (readData2) while on a high value we are loading the immediate value calculated in the previous step. Also in order to complete this section, we are connecting the result received from ALU, to the dAddress register which later will be assigned as the address of the RAM memory

```
    op1_reg<= readData1; //ALU
    dWriteData <= readData2;
    dAddress = alu_result;

    if(ALUSrc == 0)
      begin
        op2_reg <= readData2;
      end
    else if (ALUSrc == 1)
      begin
        op2_reg <= imm;
      end
  end
```

- In the next step, we need to feed some parts of the instruction register to the 3 inputs of the registers file.

```
always @(posedge clk) // Register File
    begin
        readReg1 = instr[19:15];
        readReg2 = instr[24:20];
        writeReg = instr[11:7];

    end
```

- Finally, we need to create the logic that writes the generated data back to the registers file. These data can be transferred from two different sources: Result of alu or output data of RAM. This is decided by the signal MemtoReg which controls a 2 to 1 MUX and sets the operation of MUX either to 0 for Alu_result passthrough or 1 for RAM data feed (dReadData)

```
always @(posedge clk) //  Write Back
    begin
        if(MemToReg == 0)
            begin
                WriteBackData <= alu_result;
            end
        else if (MemToReg == 1)
            begin
                WriteBackData <= dReadData;
            end
    end
```

Note: As the reader may already realized, all registers are updated synchronously. An exception is made for dAddress, readReg1, readReg2 and writeReg. This is done cause all of these registers are working as a connection between the datapath module and the future module they are

gonna be used in. But bearing in mind that all modules are working synchronously with clock positive edge signal, setting these registers to synchronous update will result to +1 clock cycle delay until these signals reach their destination, which leads to system instabilities as we tested. An exception is also made for alu where we set all of its input registers to synchronous update since the ALU module doesn't contain any clock activation system on its own, since its output ports are updated automatically whenever the input signals are changed. Of course, another way of implementing this logic is by using wires instead of registers, to work in exactly the same way.

2) Top processor Module (top_proc.v):

- *Verilog:*

  - Firstly we are defining our module's inputs and outputs and also declaring some required registers and wires.

```
module top_proc #(parameter [31:0] INITIAL_PC = 32'h00400000)
(
input wire clk, rst,
input wire [31:0] instr, dReadData,
output wire [31:0] PC, dAddress, dWriteData,
output reg MemRead, MemWrite,
output wire [31:0] WriteBackData
);


reg [3:0] ALUCtrl;
reg ALUSrc, RegWrite, MemtoReg, Zero, PCSrc, loadPC;
reg [2:0] state; // States of the FSM
wire Zero_w;
assign Zero_w = Zero;
```

  - Also we are decoding the Finite State Machine's states in order to use them later in the project

```
localparam IF  = 3'b000,
ID  = 3'b001,
EX  = 3'b010,
MEM = 3'b011,
WD  = 3'b100;
```

- Now we are ready to start implementing the systems logic. We will start by generating the "AluCtrl" signal which would be passed to the ALU operation code. This can be separated in 3 different sections based on the Op code of the current instruction:

1) *Sw* or *Lw* instructions trigger the alu opcode 0010 which corresponds to addition.
2) *BEQ* instruction triggers the alu opcode 0110 which corresponds to subtract
3) *R* and *I* type of instructions trigger the alu opcode corresponding to the current action needed. For example, *ADD* and *ADDI* will both trigger 0010 while *SLL* and *SLLI* will trigger 100.

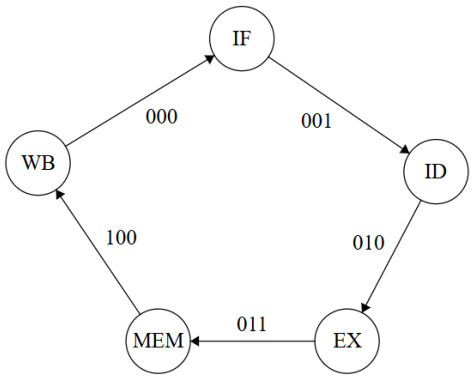
```
always @(posedge clk) //AluCtrl
  begin

    if (instr[6:0] == 7'b0100011 || instr[6:0] == 7'b0000011) //S type instruction (SW) or LW
      begin
        ALUCtrl <= 4'b0010; //Decoding of addition
      end
    else if (instr[6:0] == 7'b1100011) //B type of instruction (BEQ)
      begin
        ALUCtrl <= 4'b0110; //Decoding of subtract
      end
    else
      if (instr[6:0] == 7'b0110011) //R type of instructions
        begin
          case ({instr[31:25], instr[14:12]})
            10'b0000000000: ALUCtrl <= 4'b0010; //ADD
            10'b0100000000: ALUCtrl <= 4'b0110; //SUB
            10'b0000000001: ALUCtrl <= 4'b1001; //SLL
            10'b0000000010: ALUCtrl <= 4'b0100; //SLT
            10'b0000000100: ALUCtrl <= 4'b0101; //XOR
            10'b0000000101: ALUCtrl <= 4'b1000; //SRL
            10'b0100000101: ALUCtrl <= 4'b1010; //SRA
            10'b0000000110: ALUCtrl <= 4'b0001;//OR
            10'b0000000111: ALUCtrl <= 4'b0000;//AND
          endcase
        end
      if (instr[6:0] == 7'b0010011) //I type of instructions
        begin
          case (instr[14:12])
            3'b000: ALUCtrl <= 4'b0010; //ADDI
            3'b010: ALUCtrl <= 4'b0100; //SLTI
            3'b100: ALUCtrl <= 4'b0101; //XORI
            3'b110: ALUCtrl <= 4'b0001; //ORI
            3'b111: ALUCtrl <= 4'b0000; //ANDI
            3'b001: ALUCtrl <= 4'b1001; //SLLI
          endcase
          if ({instr[31:25], instr[14:12]} == 10'b0000000101) //SRLI
            begin
              ALUCtrl <= 4'b1000;
            end
          if({instr[31:25], instr[14:12]} == 10'b0100000101) //SRAI
            begin
              ALUCtrl <= 4'b1010;
            end
        end
  end
```

- Next, we define the value of "ALUSrc" signals which as we said selects the input of the second operator (Op2) of ALU. Only LW, SW and I type of instructions need the immediate value as an

operator so that's the only case where the "ALUSrc" is set to High.

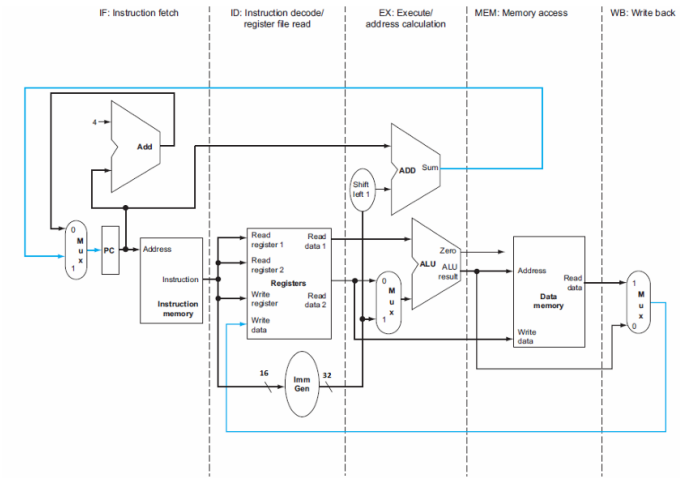
```
always @(posedge clk) //ALUSrc
    begin
        if (instr[6:0] == 7'b0000011 || instr[6:0] == 7'b0100011 || instr[6:0] == 7'b0010011)
            begin
                ALUSrc <= 1'b1;
            end
        else
            begin
                ALUSrc <= 1'b0;
            end
    end
```

- Finally, we are implementing the *Finite State Machine* which has 5 consecutive states: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM) and Write Back (WB) as shown below. Of course, Reset button press and default case, which both lead to IF state, are not represented for simplicity reasons.



- To implement this FSM we used a case loop with 5 different states changing one after the other with a positive edge clock signal and a "state" register which stores the decoding of the current state. To proceed in the next state, each time before a state is finished we are assigning synchronously to "state" register

the value of the next state. For example before the IF state ends we got `state <= ID;` to store the value of the next state. In this way our system loops again and again in these 5 states endlessly to form an FSM. The required parts needed to be executed on each state are shown in the below picture.



1) IF state: In this state, the Program counter has already been loaded with the address of the next instruction so in this step we are feeding its value to the rom module address input. In that way loadPC signal needs to be set to low as we are already on our way to receive the new instruction. The same applies to RegWrite and MemWrite, remain 0. Note that in all states the FSM is checking the current instruction and if it receives a LW instruction it sets MemtoReg to "1" to choose the source of WriteBackData. Also, it is constantly checking if the instruction is BEQ and the alu result is 0, in order to feed the immediate value to op2 by setting PCSrc to 1. Otherwise, this signal remains 0. Finally, changes the state to ID to proceed in the next FSM state in the incoming clock cycle

```
IF:
    begin
        loadPC <= 0;
        MemtoReg <= 1'b0;
        RegWrite <= 1'b0;
        PCSrc <= 1'b0;
        MemWrite <= 1'b0;
        if (instr[6:0] == 7'b1100011 && Zero == 1'b1)
            begin
                PCSrc <= 1'b1;
            end

        if(instr[6:0] == 7'b0000011)
            begin
                MemtoReg <= 1'b1;
            end

        state <= ID;
    end
```

2) 2) ID state: In this state the FSM is decoding the currenet instruction and feeds the required data to the Registers inputs. The same as previous applies for RegWrite, MemWrite, MemtoReg and PCSrc signals. Finally, changes the state to EX to proceed in the next FSM state in the incoming clock cycle

```
ID:
    begin
        MemtoReg <= 1'b0;
        RegWrite <= 1'b0;
        PCSrc <= 1'b0;

        if(instr[6:0] == 7'b0000011)
            begin
                MemtoReg <= 1'b1;
            end
        if (instr[6:0] == 7'b1100011 && Zero == 1'b1)
            begin
                PCSrc <= 1'b1;
            end

        state <= EX;
    end
```

3) EX state: In that state, the FSM executes the calculation of the alu result based on the inputs we forwarded on the previous states. The same as previous applies for RegWrite, MemWrite, MemtoReg and PCSrc signals. Finally, changes the state to MEM to proceed in the next FSM state in the incoming clock cycle.

```
EX:
    begin
        MemtoReg <= 1'b0;
        RegWrite <= 1'b0;
        PCSrc <= 1'b0;
        MemWrite <= 1'b0;
        if(instr[6:0] == 7'b0000011)
            begin
                MemtoReg <= 1'b1;
            end

        if (instr[6:0] == 7'b1100011 && Zero == 1'b1)
            begin
                PCSrc <= 1'b1;
            end

        state <= MEM;
    end
```

4) MEM state: In this state, the goal is to enable write and read signals of RAM module considering the instruction. If the current instruction is SW then the system needs to set MemWrite to "1" in order to store the data outputted from ReadData2 in the address specified by the result of the ALU. Note that this specific RAM module doesn't use read enable signal so as long as the MemWrite signal is low we can read data from it at any time. The same as previous applies for RegWrite, MemtoReg and PCSrc signals. Finally, changes the state to WB to proceed in the next and final FSM state in the incoming clock cycle.

```
MEM:
    begin
        PCSrc <= 1'b0;
        MemtoReg <= 1'b0;
        RegWrite <= 1'b0;

        if(instr[6:0] == 7'b0000011) // LW Instruction
            begin
                MemtoReg <= 1'b1;
                MemRead <= 1'b1;
            end

        if(instr[6:0] == 7'b0100011) // SW Instruction
            begin
                MemWrite <= 1'b1;
            end

        else
            begin
                MemRead <= 1'b0;
                MemWrite <= 1'b0;
            end

        if (instr[6:0] == 7'b1100011 && Zero == 1'b1)
            begin
                PCSrc <= 1'b1;
            end

        state <= WD;
    end
```

5) WB state: This final state , is responsible for writing back the data produced either from ram (read) or from ALU calculation which is specified by the signal MemtoReg. So as we did in all states, MemtoReg's value is updated to "1" only if we got a LW instruction. This is the only state of the FSM that MemtoReg signal matter. After all of these, the loadPC signal is turned on allowing the program counter to point (asynchronously for system synchronization purposes) to the next instruction from ROM module, in order for the instruction register to be ready in the IF state. Last but not least, the state register receives the code of IF state.

```
WD:
    begin
        PCSrc <= 1'b0;
        MemtoReg <= 1'b0;
        RegWrite <= 1'b1;
        loadPC <= 1;
        MemWrite <= 1'b0;

        if(instr[6:0] == 7'b0000011)
            begin
                MemtoReg <= 1'b1;
            end

        if (instr[6:0] == 7'b1100011 && Zero == 1'b1)
            begin
                PCSrc <= 1'b1;
            end

        state <= IF;
    end
```

Note that for safety reasons we are adding a default value which corresponds to ID state.

```
default: state <= IF;
```

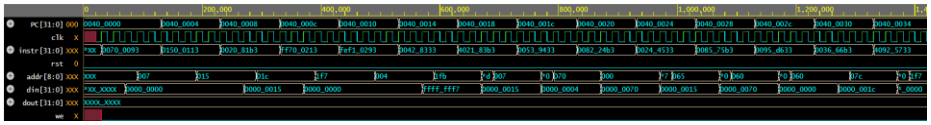
- *Instruction list included in rom_bytes.data*

N/N	Bit Code (32 bits)	Instruction
1	00000000011100000000000001 0010011	ADDI
2	00000001010100000000000010 0010011	ADDI
3	00000000001000000100000011 0110011	ADD
4	11111111101111000000000100 0010011	ADDI
5	11111110111110001000000101 0010011	ADDI
6	00000000010000101000000110 0110011	ADD
7	01000000001000011000000111 0110011	SUB
8	00000000010100111001010000 0110011	SLL
9	00000000100000100010010001 0110011	SLT
10	00000000001001000100010010 0110011	XOR
11	00000000100000101011101011 0110011	AND
12	00000000100101011110101100 0110011	SRL
13	00000000001101100110011101 0110011	OR
14	01000000100100100101011110 0110011	SRA
15	00000000101100101010000000 0100011	SW
16	00000000000000010101001111 0000011	LW
17	11111101001101000111100000 0010011	ANDI
18	00000001011010000110100001 0010011	ORI
19	00000000000010110110110010 0010011	SRLI
20	00000000101101111100010000 1100011	BEQ
21	00000000000000000000000000 0000000	-
22	00000000000000000000000000 0000000	-
23	00000000000000000000000000 0000000	-
24	00000000000000000000000000 0000000	-
25	00000000111110010010010010 0010011	SLTI
26	00000011101001000100100011 0010011	XORI
27	00000000000011000100110100 0010011	SLLI
28	01000000001001111110100101 0010011	SRAI
29	00000000000000000000000000 0000000	-

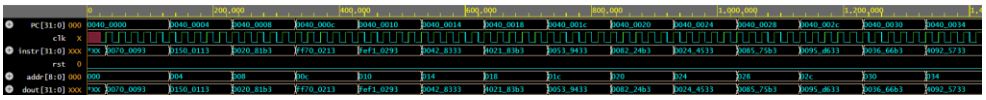
- Final waveforms for each component:

  Note: Hex scale used for clearer visualization.

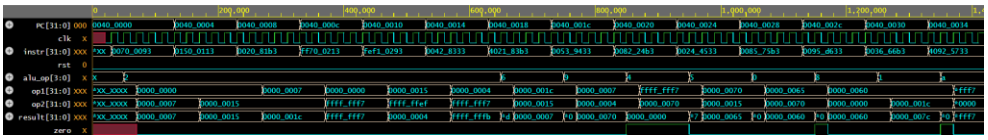
A) RAM



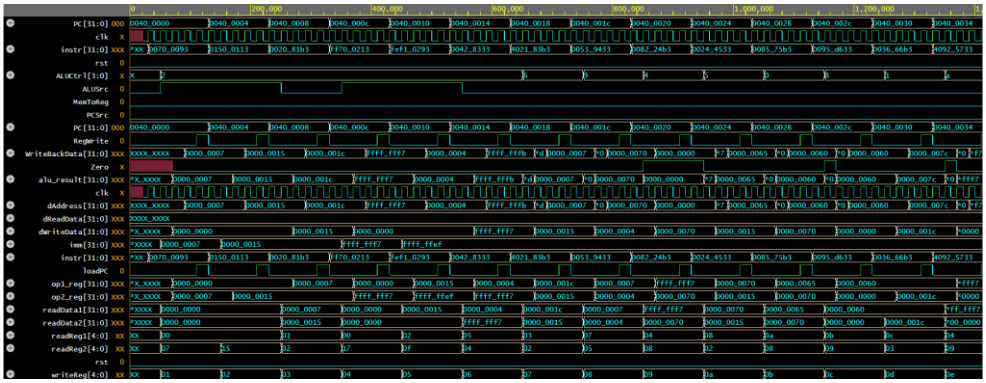
B) ROM



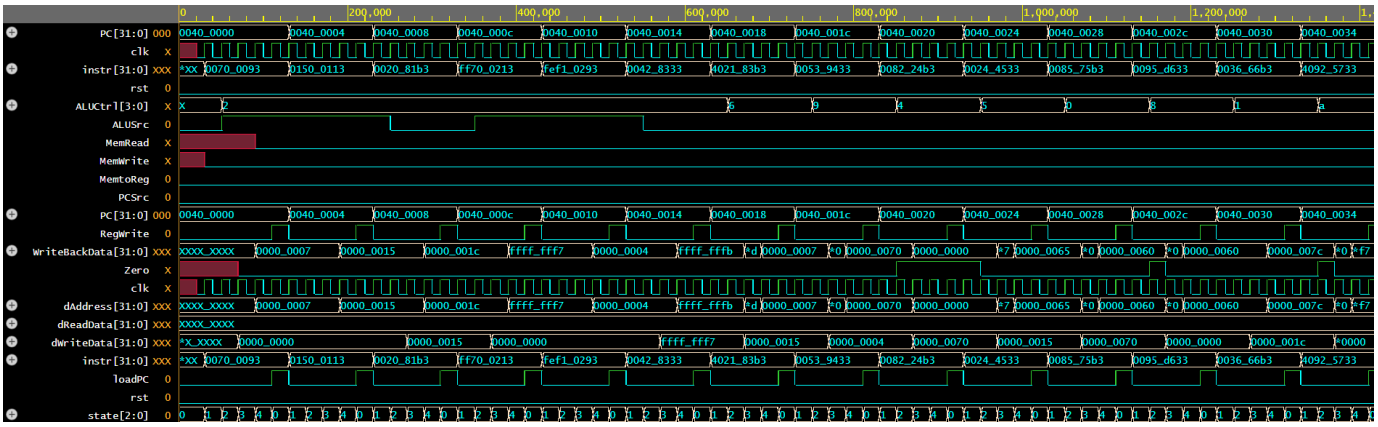
C) ALU



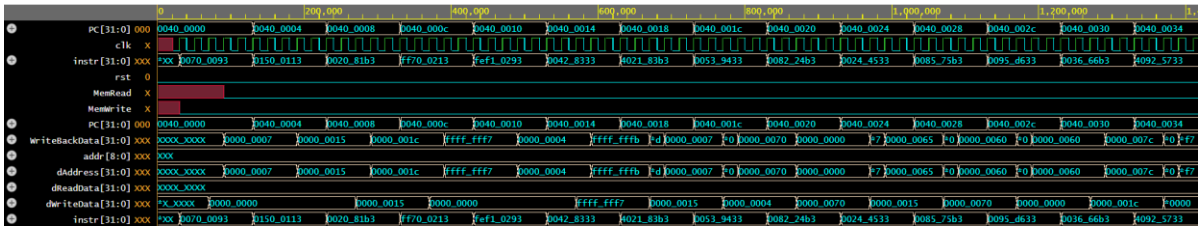
D) DATAPATH



E) TOP_PROC



F) TESTBENCH



Waveforms for example instructions:

- *ADDI:*

The first instruction we are gonna analyze is 1st in the row:

00000000011100000000000010010011

Lets break it down step by step:

- Op code: 001 0011 (It's an I type of instruction)
  - r_source: 10010
  - r_destination: 01001
  - Imm: 0000 0000 1111

In the picture above we can see that by the end of IF state (000) the instruction register is already loaded. Also by the end of the ID state (001) the imm register containing the immediate value of the instruction: 00000000000000000000000000000111 with sign extension. Also we can see that in the same time the Read register 1 is having the [19-15] and [24-20] bits of the instruction as shown in the part 2 main picture. Note also that the op2 is chosen from imm as the system recognized its an I instruction and updated ALUSrc to 1 (Note that ALUop is set to addition 0010). By the end of the state EX (010) we see that the op1 and 2 are loaded and the alu calculated the addition between the contents of ReadReg1 (which is 32'b0 as expected) and the imm register's value. In the MEM state nothing happens as the instruction is not LW or SW so the Memwrite signal remains 0. Finally, in the end of the WB state (100) we can see that MemtoReg signal remains 0 as we don't have any LW instruction.Also the loadPC is enable and the PC gets the second address on the next cycle. The RegWrite signal is enabled, writeReg is equal to [11,7] bits of the instruction and. The result of these two is that Writebackdata is loaded with the result of alu and through WriteData (which has the same value) the data are

stored in the registers file. The same operation applies for the second instruction which is also an ADDI.

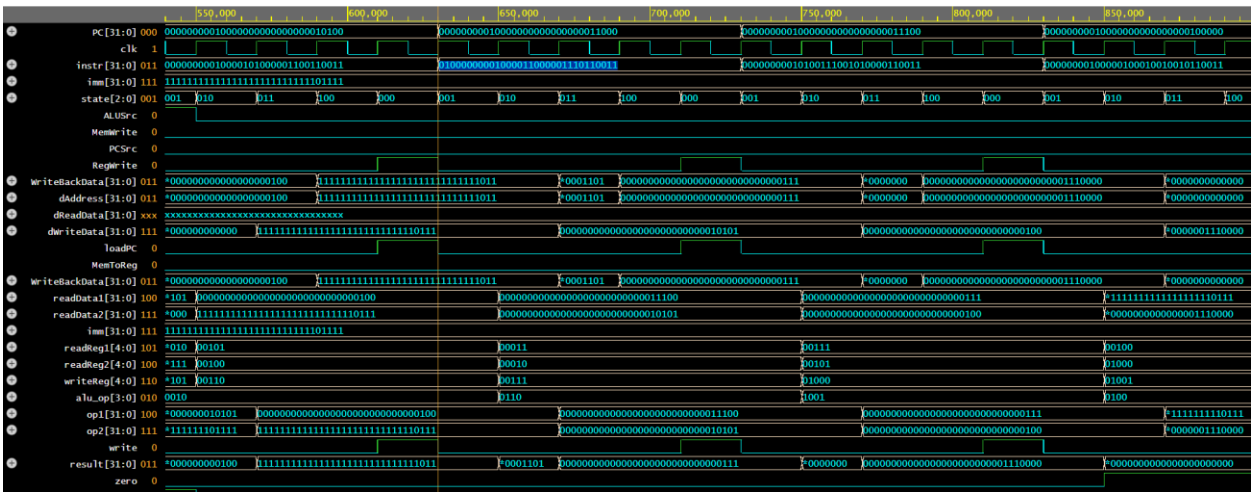
- *SUB:*

The second instruction we are going to analyze is 7th in the row:

01000000001000011000001110110011

Lets break it down step by step:

- Op code: 0110011 (It's an R type of instruction)
  - r_source: 00111
    - rs1: 00011
    - rs2: 0 0010
  - Sub = rs1-rs2



Note: Sub instruction is the highlighted one

As we can see IF (000) works perfectly and the instruction is loaded in the end of it. Also readReg1 and readReg2 have the address of the two

operants we decoded above (rs, rs2) by the end of ID (001). From there in EX(010), their contents loaded into the Op1 and Op2 as the ALUSrc remains 0 as long as we don't have any LW, SW or I instruction. From there, again the MEM(011) state is not used as we don't have any SW or LW. MemWrite remains 0 and the writeback data register gets the value of alu result (where aluop is 0110 equal to subtract) which is directly transferred to writeData in the beginning of the WB cycle. Finally, in the end of the WB the data are stored in the address 00111 shown by the rd register decoded from the instruction. Note also that the PCSrc (as zero is also 1) is enabled which means the next value of PC is the current plus the imm

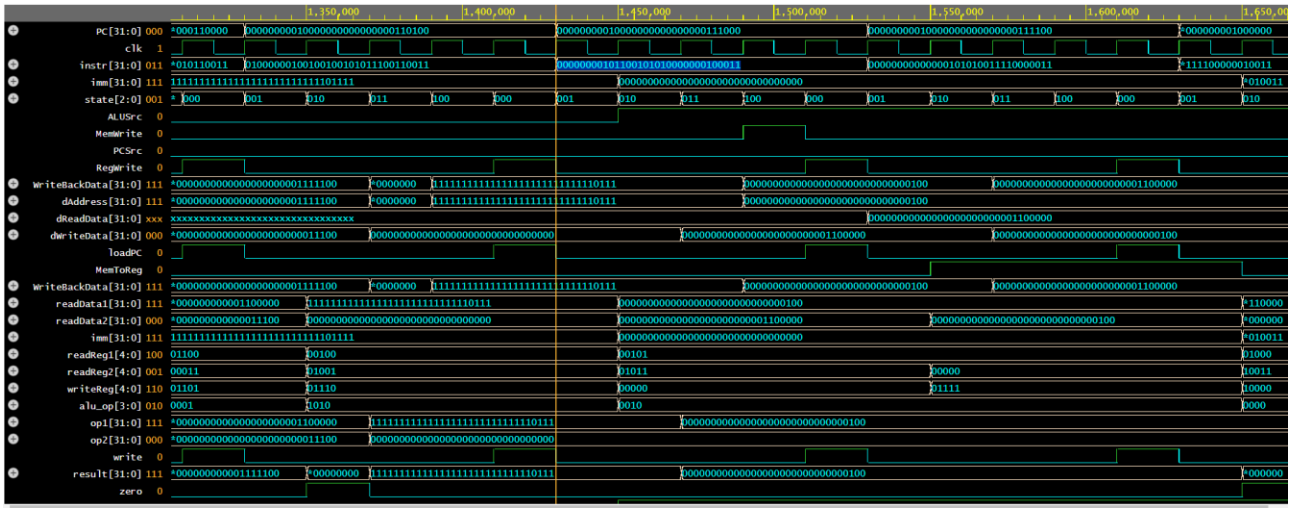
- *SW:*

The third instruction we are going to analyze is 15th in the row:

00000000101100101010000000100011

Lets break it down step by step:

- Op code: 0100011 (S type of instruction)
  - r_source (rss2): 01011
  - r_address(rs1): 00101
- imm: 000000000000 (offset)



Note: SW instruction is the highlighted one

Again IF (000) works perfectly so we move to the next state. We can see that by the end of the ID (001) the readReg1 is equal to rs1: 00101 (address register) and the readReg2 is equal to rs2: 00101 (source register). Note also that by the end of ID the alu_op is showing addition 0010 as expected. As we have a SW instruction the ALUSrc is equal to 1 giving the op2 operant the value of imm (0). Then in the EX (010) state the system calculates the result and passes the value to the address of the RAM module (addr). The MemWrite and so the we (in ram) is 1, and the din gets the value of dreadData2 to store it in the specified address.

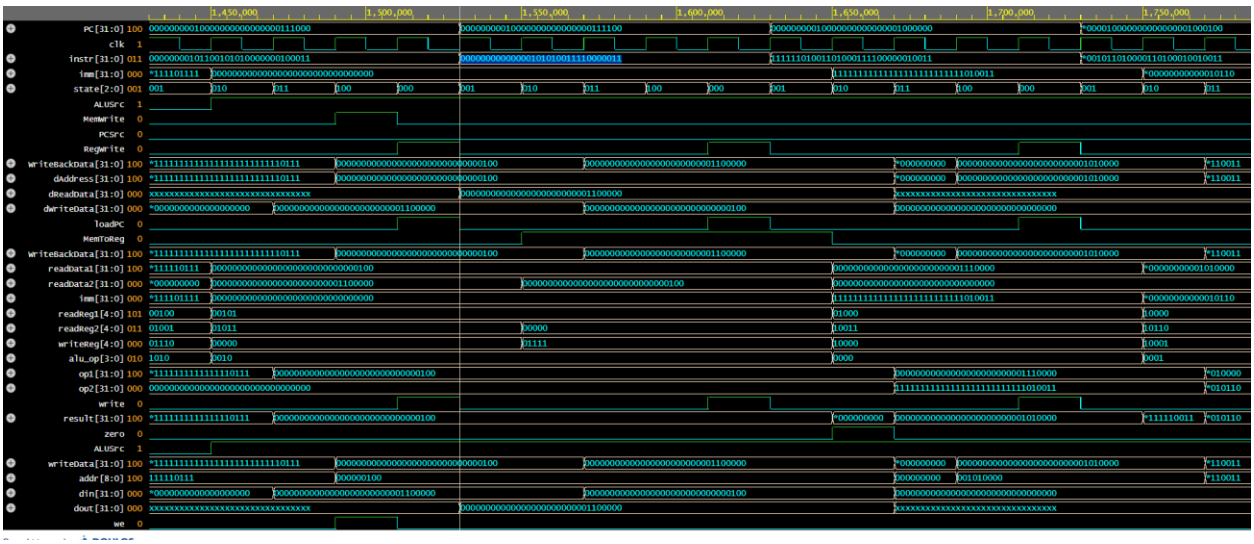
- *LW:*

The fifth instruction we are going to analyze is 19th in the row:

00000000101100101010000000100011

Lets break it down step by step:

- Op code: 100011 (S type of Instruction)
  - rd: 01111
  - rs1: 00101
  - imm: 000000000000



Note: LW instruction is the highlighted one

In exactly the same way as before we can see that all signals working perfectly, the addresses and the operants of alu are correctly set, the same with aluop and in dout we get the value we stored before in the same address which is: 32'b1100000

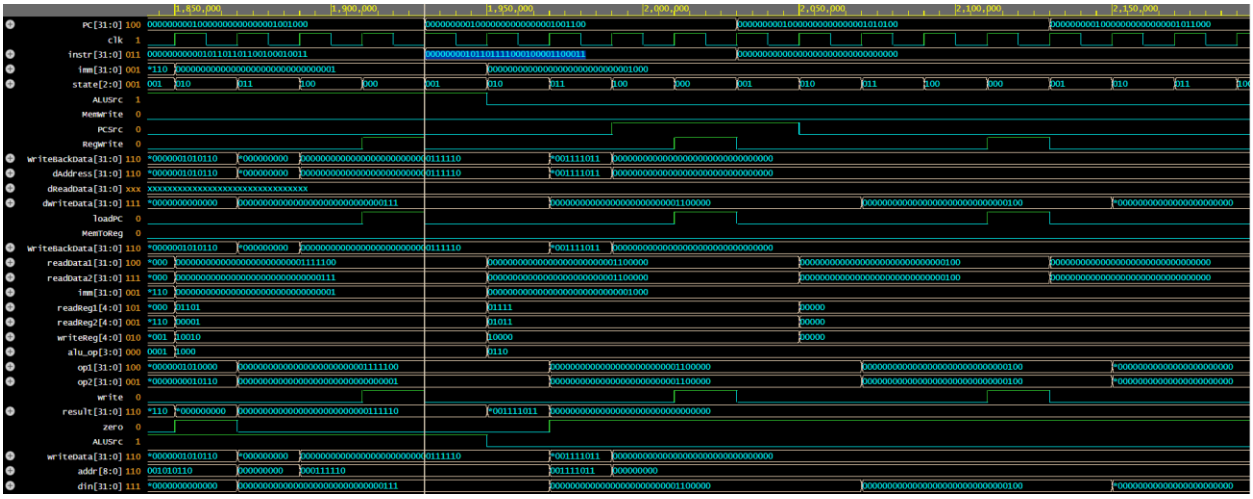
- *BEQ:*

The fourth and final instruction we are going to analyze is 15th in the row:

00000000101101111000100001100011

Lets break it down step by step:

- Op code: 1100011 (B type of instruction)
  - rs2: 01011
  - rs1: 01111
  - imm: 000000001000



Note: BEQ instruction is the highlighted one

We can also see that the FSM is working properly. The aluop is equal to subtract 0110. The imm is also calculated correctly. Also readreag2 is equal to rs2 (01011) and readreg1 is equal to rs1 (01111). And finally the address of the next program counter is calculated by PC+imm=PC+8.

FINAL WAVEFORM