Introduction to Digital Logic Design Lab
EECS 31L

Lab 4

Kyle Zyler Cayanan
80576955

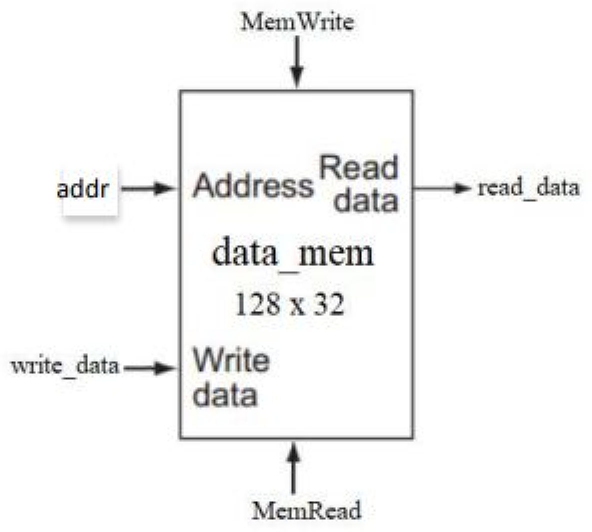
02/27/2021

1 - Objective

For this lab, we were tasked with designing a RISC-V Single Cycle Processor. All modules of the processor we had designed in previous labs. For this lab, we focused on building the Data Memory and Datapath modules. The other modules designed previously were the Flip Flop, Instruction Memory, Register File, Muxs and an Adder.

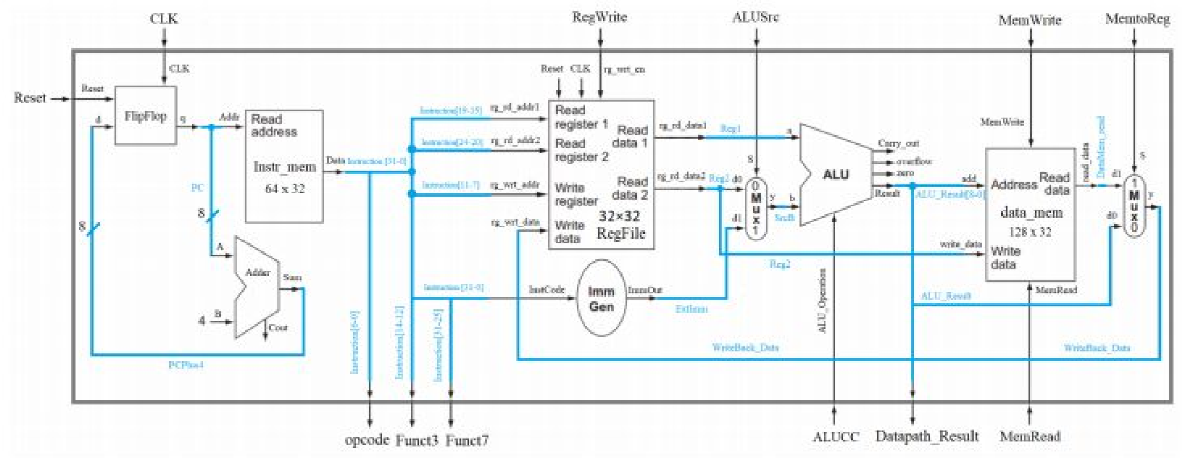
1.1 - Data Memory

The Data Memory Module is similar to the Instruction Memory module. It will read an address and write a 32 bit number to that address. There are 128 addresses available for use in the Data Memory Module. There are also inputs MemRead and MemWrite which enables the module to read and write to and from the address. The output simply reads the data written from the specified address. Where the output goes will be described in the Data Path module. Below is the diagram of the module.



1.2 - Data Path

The Data Path is essentially the entire processor as a module. It will control the data flow of each input. The output cycles back through several modules and gets split up into different sections as an instruction.



1.3 - 2:1 MUX

The MUX is the same as before, it has just been adjusted to handle 32 bit inputs and outputs.

1.4 - Adder

The adder used is a half adder and increments the Program Counter by 4 each time.

2 - Procedure

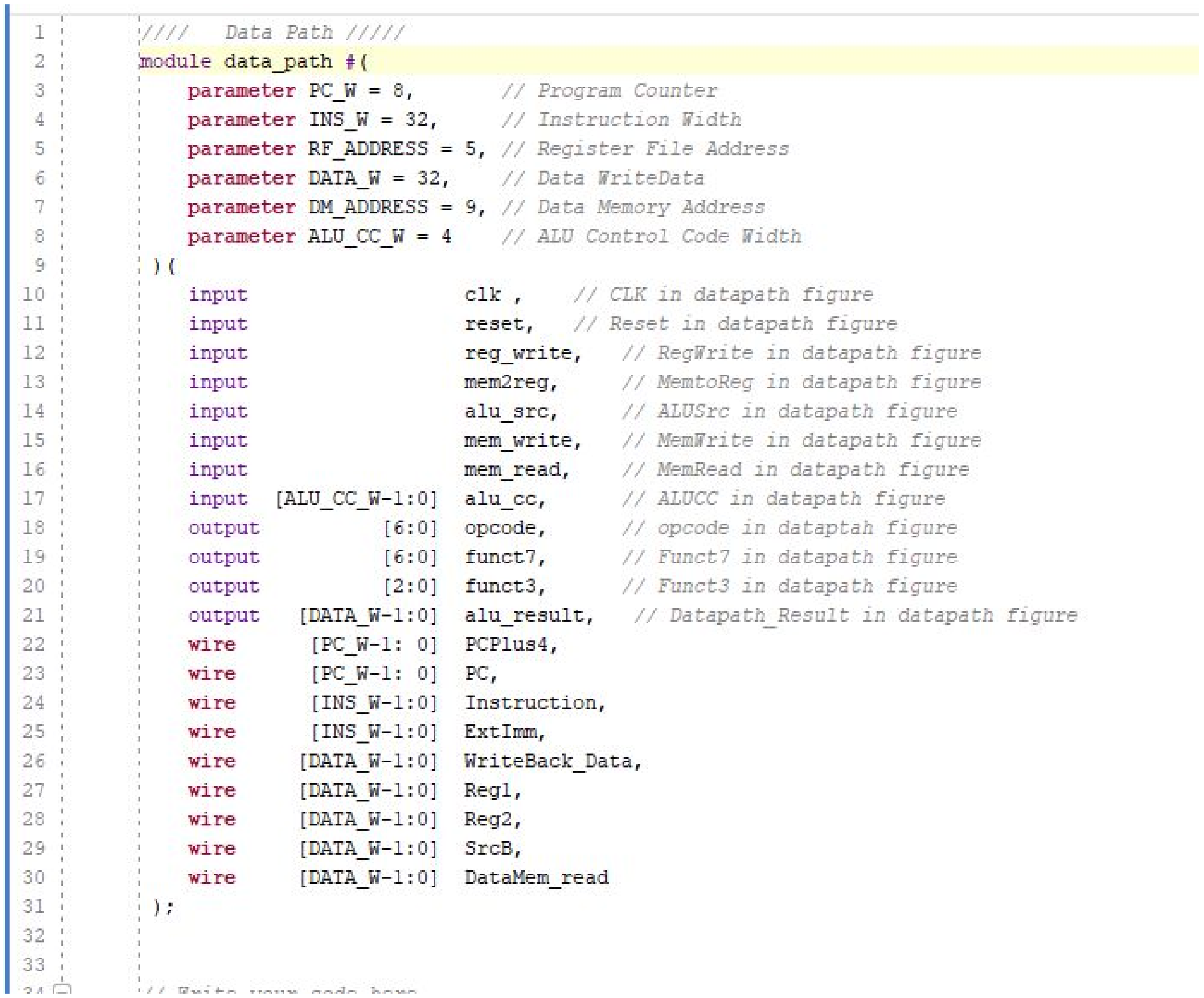
The procedure is a mixture of behavioral and structural description. Behavioral description was utilized primarily to design the Data Memory module while the DataPath module used structural description.

2.1 - Data Memory

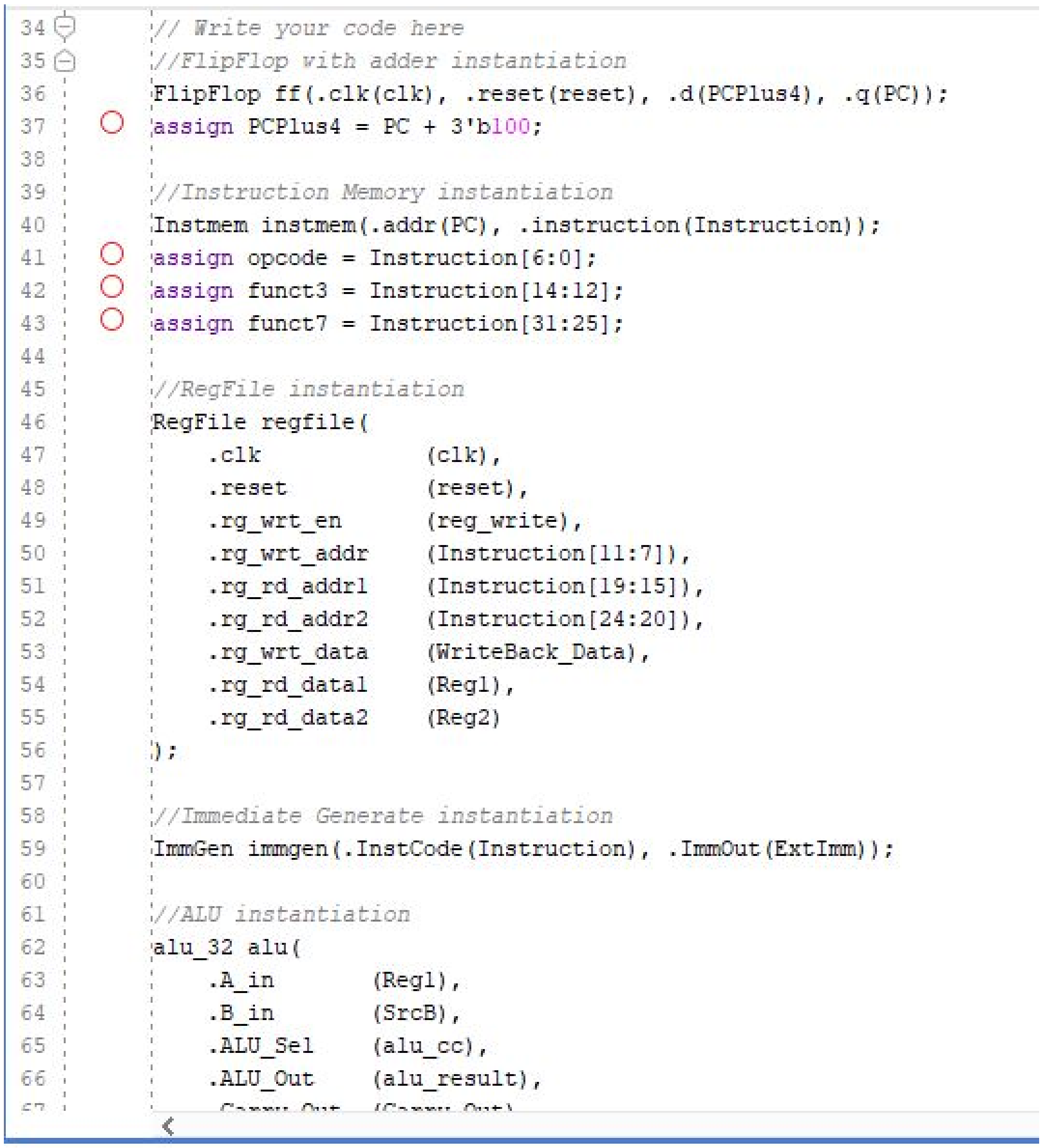
Since the Data Memory can be described as similar to Instruction Memory. The only difference is that it has the ability to read and write data to and from an address. As for the code, it consists of an always block that writes to the address if the MemWrite input is enabled and reads data from the address if MemRead is enabled. As opposed to Instruction memory where the data for each address is hard coded, Data Memory uses the ALU results to write to the address.

2.2 - DataPath

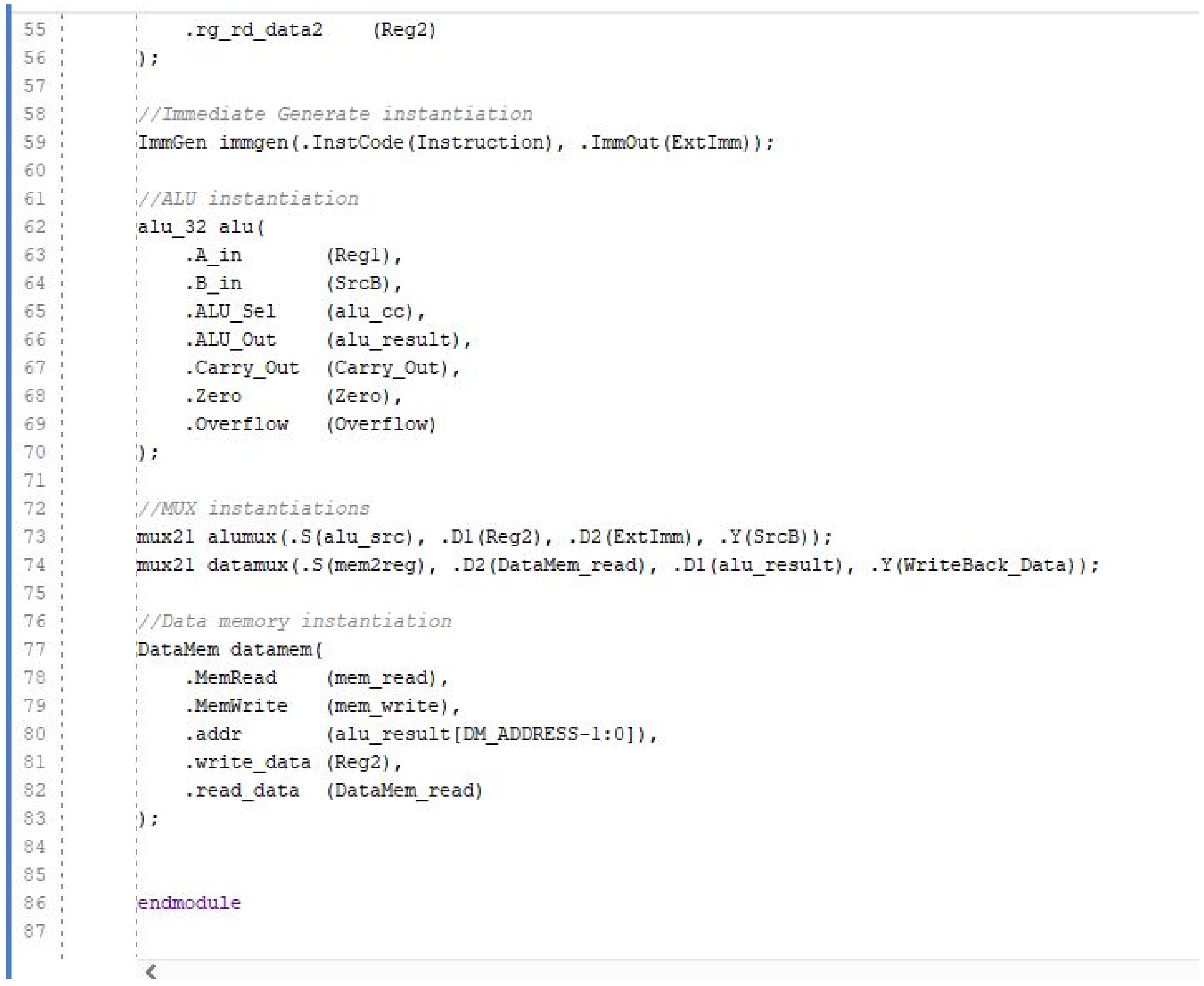
The Data Path uses structural description by instantiating each module. Instead of using the half adder module, we instead replace it with simply performing the operation of adding 4 to the Program counter. Each module besides the half adder has been instantiated to form the data path of that of a machine cycle. The four steps to a machine cycle are Fetch, Decode, Execute, Store. Starting with the Flip Flop module, it stores the input and outputs it as an address for the Instruction Memory module. While this is happening, the PC address increments by 4 to indicate which instruction the processor is on. These three modules make up the Fetch Cycle. The Decode cycle relies on the data from Instruction Memory and splits it up into sections and assigns each section to either opcode, register, or data. The Execution Cycle is mainly handled by the ALU which essentially carries out the instruction. Data Memory is instantiated so that it handles the Store Cycle. The code is shown below.



```verilog
////   Data Path /////
module data_path #(
    parameter PC_W = 8,       // Program Counter
    parameter INS_W = 32,     // Instruction Width
    parameter RF_ADDRESS = 5, // Register File Address
    parameter DATA_W = 32,    // Data WriteData
    parameter DM_ADDRESS = 9, // Data Memory Address
    parameter ALU_CC_W = 4    // ALU Control Code Width
 )(
    input                  clk ,    // CLK in datapath figure
    input                  reset,   // Reset in datapath figure
    input                  reg_write,   // RegWrite in datapath figure
    input                  mem2reg,     // MemtoReg in datapath figure
    input                  alu_src,     // ALUSrc in datapath figure
    input                  mem_write,   // MemWrite in datapath figure
    input                  mem_read,    // MemRead in datapath figure
    input   [ALU_CC_W-1:0]  alu_cc,      // ALUCC in datapath figure
    output           [6:0]  opcode,      // opcode in dataptah figure
    output           [6:0]  funct7,      // Funct7 in datapath figure
    output           [2:0]  funct3,      // Funct3 in datapath figure
    output    [DATA_W-1:0]  alu_result,   // Datapath_Result in datapath figure
    wire       [PC_W-1: 0]  PCPlus4,
    wire       [PC_W-1: 0]  PC,
    wire       [INS_W-1:0]  Instruction,
    wire       [INS_W-1:0]  ExtImm,
    wire      [DATA_W-1:0]  WriteBack_Data,
    wire      [DATA_W-1:0]  Reg1,
    wire      [DATA_W-1:0]  Reg2,
    wire      [DATA_W-1:0]  SrcB,
    wire      [DATA_W-1:0]  DataMem_read
 );


// Write your code here
```



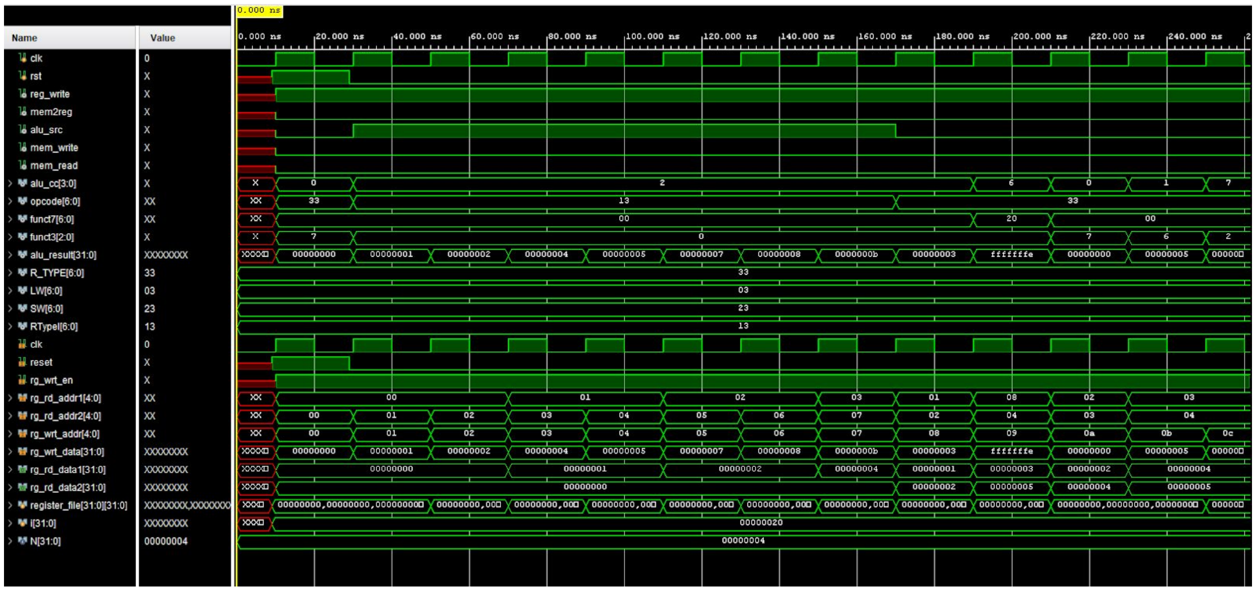
```
// Write your code here
//FlipFlop with adder instantiation
FlipFlop ff(.clk(clk), .reset(reset), .d(PCPlus4), .q(PC));
assign PCPlus4 = PC + 3'b100;

//Instruction Memory instantiation
Instmem instmem(.addr(PC), .instruction(Instruction));
assign opcode = Instruction[6:0];
assign funct3 = Instruction[14:12];
assign funct7 = Instruction[31:25];

//RegFile instantiation
RegFile regfile(
    .clk            (clk),
    .reset          (reset),
    .rg_wrt_en      (reg_write),
    .rg_wrt_addr    (Instruction[11:7]),
    .rg_rd_addr1    (Instruction[19:15]),
    .rg_rd_addr2    (Instruction[24:20]),
    .rg_wrt_data    (WriteBack_Data),
    .rg_rd_data1    (Reg1),
    .rg_rd_data2    (Reg2)
);

//Immediate Generate instantiation
ImmGen immgen(.InstCode(Instruction), .ImmOut(ExtImm));

//ALU instantiation
alu_32 alu(
    .A_in       (Reg1),
    .B_in       (SrcB),
    .ALU_Sel    (alu_cc),
    .ALU_Out    (alu_result),
```

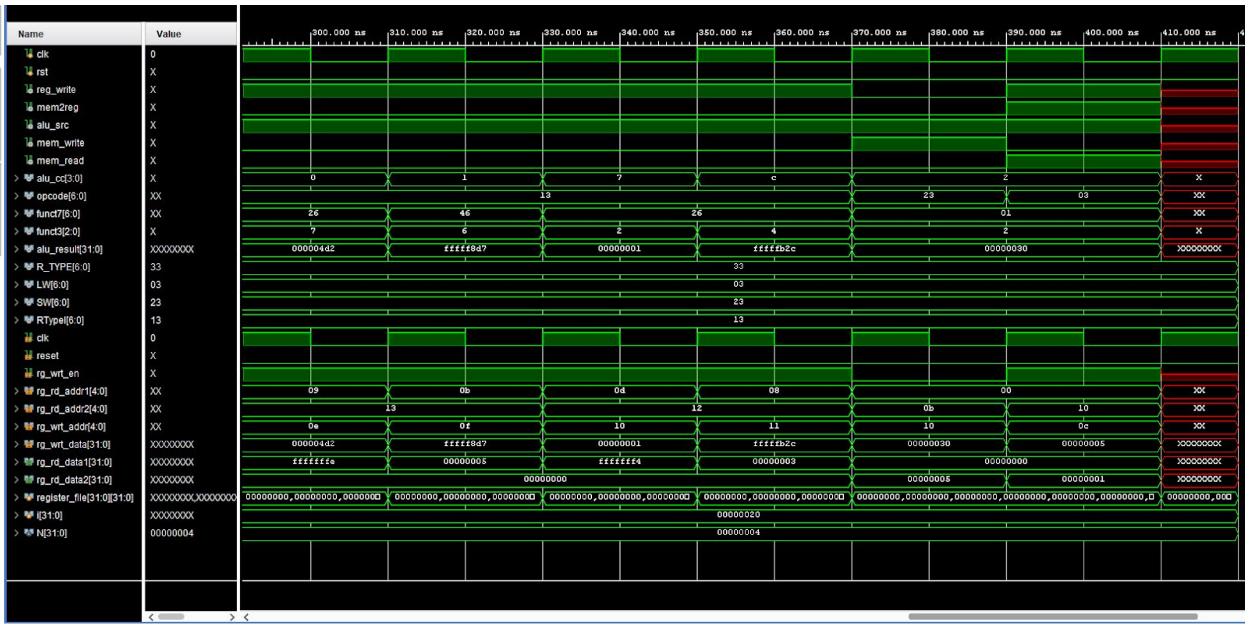


```
        .rg_rd_data2    (Reg2)
);

//Immediate Generate instantiation
ImmGen immgen(.InstCode(Instruction), .ImmOut(ExtImm));

//ALU instantiation
alu_32 alu(
    .A_in       (Reg1),
    .B_in       (SrcB),
    .ALU_Sel    (alu_cc),
    .ALU_Out    (alu_result),
    .Carry_Out  (Carry_Out),
    .Zero       (Zero),
    .Overflow   (Overflow)
);

//MUX instantiations
mux21 alumux(.S(alu_src), .D1(Reg2), .D2(ExtImm), .Y(SrcB));
mux21 datamux(.S(mem2reg), .D2(DataMem_read), .D1(alu_result), .Y(WriteBack_Data));

//Data memory instantiation
DataMem datamem(
    .MemRead    (mem_read),
    .MemWrite   (mem_write),
    .addr       (alu_result[DM_ADDRESS-1:0]),
    .write_data (Reg2),
    .read_data  (DataMem_read)
);


endmodule

```

3 - Simulation Results

At each clock cycle, the ALU will read data from the Register File and perform the respective instruction. All outputs came out as expected when compared to the provided waveform in the lab manual. Each ALU result corresponds correctly to its operation as well. The screenshots of the waveform are pictured below.


Name
Value
clk
rst
reg_write
mem2reg
alu_src
mem_write
mem_read
alu_cc[3:0]
opcode[6:0]
funct7[6:0]
funct3[2:0]
alu_result[31:0]
R_TYPE[6:0]
LW[6:0]
SW[6:0]
RTypeI[6:0]
clk
reset
rg_wrt_en
rg_rd_addr1[4:0]
rg_rd_addr2[4:0]
rg_wrt_addr[4:0]
rg_wrt_data[31:0]
rg_rd_data1[31:0]
rg_rd_data2[31:0]
register_file[31:0][31:0]
i[31:0]
N[31:0]


Name
Value
clk
rst
reg_write
mem2reg
alu_src
mem_write
mem_read
alu_cc[3:0]
opcode[6:0]
funct7[6:0]
funct3[2:0]
alu_result[31:0]
R_TYPE[6:0]
LW[6:0]
SW[6:0]
RTypeI[6:0]
clk
reset
rg_wrt_en
rg_rd_addr1[4:0]
rg_rd_addr2[4:0]
rg_wrt_addr[4:0]
rg_wrt_data[31:0]
rg_rd_data1[31:0]
rg_rd_data2[31:0]
register_file[31:0][31:0]
i[31:0]
N[31:0]