Ellis Chen
ECPE 173
Project 3: Beta 1 Report

PC Design:

The clock and reset signals are both rising edge triggered in my design. If the reset signal is a logical 1, then the PC is reset to a 32-bit 0. Else, the program counter increments by decimal value of 4 on the rising edge of the clock. A potential issue with this design could be related to the clock driving the reset. The reset is supposed to be asynchronous, but in my design, I have the reset signal occur on the rising edge of the clock.

```
module pc(input logic clk, reset, output logic [31:0] ia);

always_ff@(posedge clk, posedge reset)
begin

if (reset == 1'b1)
        begin
                ia <= 32'd0;
        end
else
        begin
                ia <= (ia + 32'd4);
        end

end

endmodule
```

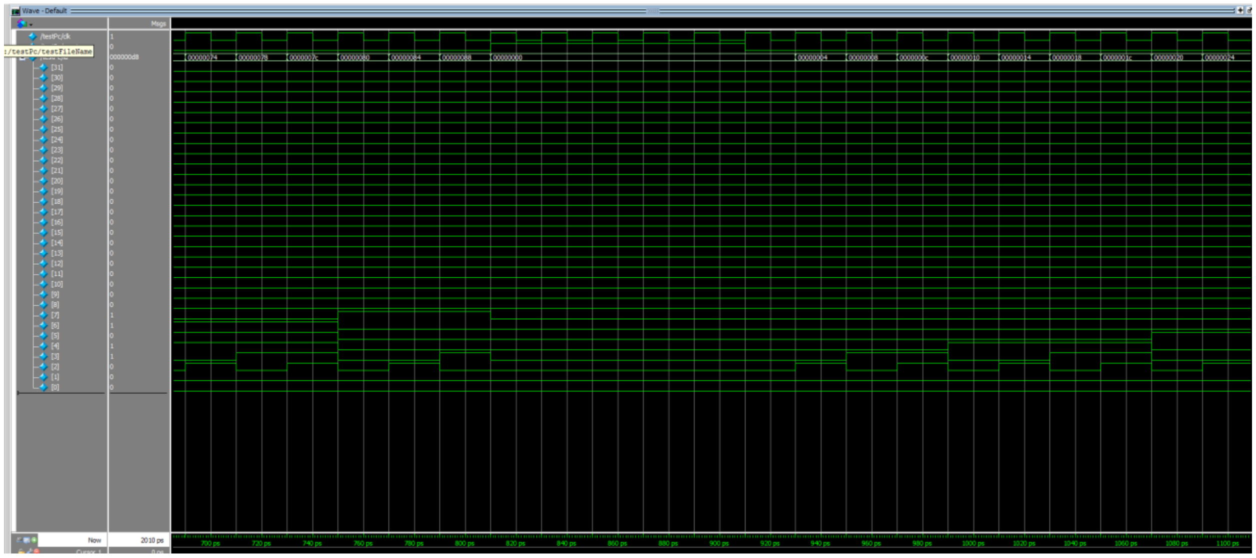

Figure 1: Waveform diagram for PC

Register File Design:

I created an array of 32 registers, each with 32 bits for the design of the register block. I initialized all the bits of each of the 32 registers to have a value of 0. In the first always_comb block, I checked if the read register 1 (Rs) was not zero. If Rs was zero, then the read data 1 would output a 32-bit 0; else whatever was stored in Rs would be assigned to the read data 1 output. The same procedure applied to the read register 2 (Rt) and its respective read data 2 output. The reading from the registers part was all done before the clock went high. Whenever the RegWrite control signal has a value of 1 on the positive edge of the clock, incoming data to write is assigned to Rt or Rd registers depending on the RegDst control signal. One potential issue with my design may be the use of two always_comb blocks with an always_ff block might cause unexpected results in certain test cases, where the clock may trigger high and write to memory before any data would be read in.

```
module regfile(input logic clk, RegWrite, RegDst,
                  input logic [4:0] ra, rb, rc,
                  input logic [31:0] wdata,
                  output logic [31:0] radata, rbdata);

logic [31:0] memory[31:0];

initial begin
for (int i = 0; i <32; i++)
memory[i] <= 0;
end

always_comb
begin
      if (ra == 5'd0) begin
      radata <= 32'd0; end
      else begin
      radata <= memory[ra];
      end
end

always_comb
begin
      if (rb == 5'd0) begin
      rbdata <= 32'd0; end
      else begin
      rbdata <= memory[rb];
      end
end

always_ff@(posedge clk)
begin

      if (RegWrite == 1'b1)
      begin
            if (RegDst == 1'b1)
            begin
                  memory[rb] <= wdata;
            end
            else begin
                  memory[rc] <= wdata;
            end
      end
end

endmodule
```

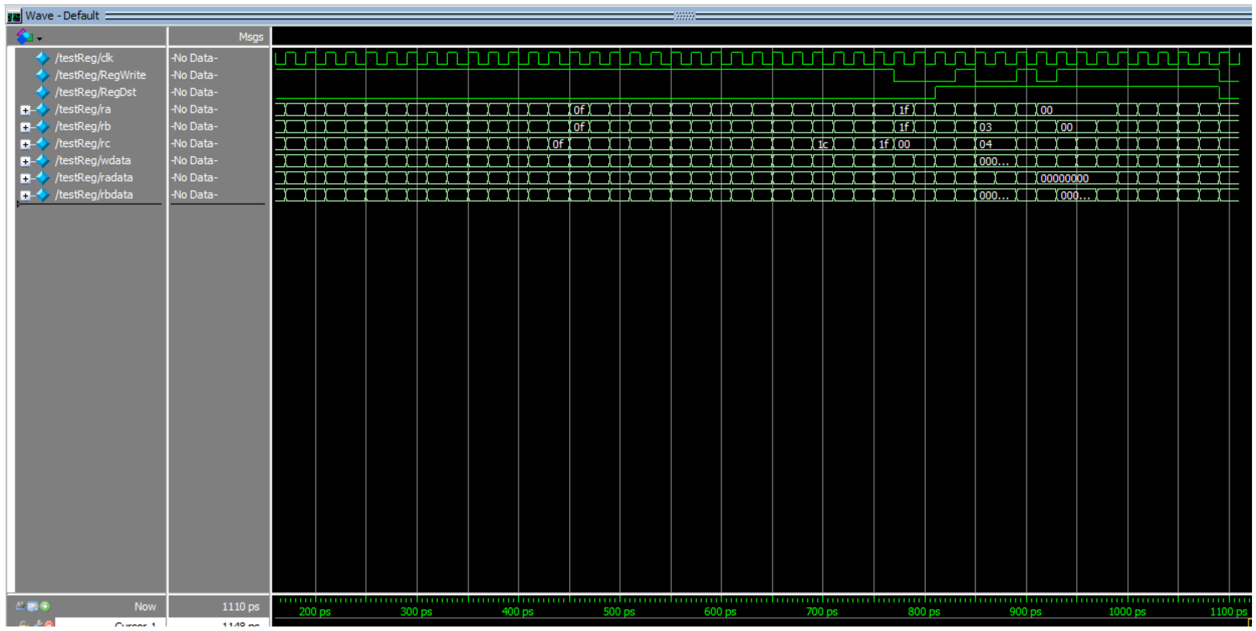

Figure 2: Waveform diagram for register file

Control Section:

My design first checks if the reset control signal is high, if true then I disabled all control signals by assigning a value of 0. My design then checks the 6-bit Opcode; if the opcode was 6'b000000, then it would be classified as an r-type and the design would have to check the bit field for funct. All the control signals were determined using: worksheet 6, singleCycleControlTable.xlsx, project 2: ALU description page, and MIPS instruction format pdf. Since there was a lot of repetitive control signal assignments that could have been avoided, going back to this design and modifying it would not be a pleasant task. For example, most of the r-type instructions have the same control signals, so there were many opportunities to reduce hardcoding the values.

```
module ctl(input logic reset,
                  input logic [5:0] opCode,
                  input logic [5:0] funct,
                  output logic RegDst, ALUSrc, RegWrite,
                  output logic MemWrite, MemRead, MemToReg,
                  output logic [4:0] ALUOp);

always_comb
begin
if (reset == 1'b1) begin
      begin
      MemWrite <= 1'b0;
      MemRead <= 1'b0;
```

```
                MemToReg <= 1'b0;
                RegDst <= 1'b0;
                ALUSrc <= 1'b0;
                RegWrite <= 1'b0;
                ALUOp <= 5'b00000;
                end
        end
        else begin
                case(opCode)
                6'b000000:  //ADD, SUB, AND, NOR, OR, XOR, SLL, SRL, SRA, SLT
                                begin
                                        case(funct)
                                        6'b100000:  begin //ADD
                                                        MemWrite <= 1'b0;
                                                        MemRead <= 1'b0;
                                                        MemToReg <= 1'b0;
                                                        RegDst <= 1'b0;
                                                        ALUSrc <= 1'b0;
                                                        RegWrite <= 1'b1;
                                                        ALUOp <= 5'b00000;
                                                        end
                                        6'b100010:  begin //SUB
                                                        MemWrite <= 1'b0;
                                                        MemRead <= 1'b0;
                                                        MemToReg <= 1'b0;
                                                        RegDst <= 1'b0;
                                                        ALUSrc <= 1'b0;
                                                        RegWrite <= 1'b1;
                                                        ALUOp <= 5'b00001;
                                                        end
                                        6'b100100:  begin //AND
                                                        MemWrite <= 1'b0;
                                                        MemRead <= 1'b0;
                                                        MemToReg <= 1'b0;
                                                        RegDst <= 1'b0;
                                                        ALUSrc <= 1'b0;
                                                        RegWrite <= 1'b1;
                                                        ALUOp <= 5'b11000;
                                                        end
                                        6'b100111:  begin //NOR
                                                        MemWrite <= 1'b0;
                                                        MemRead <= 1'b0;
                                                        MemToReg <= 1'b0;
                                                        RegDst <= 1'b0;
                                                        ALUSrc <= 1'b0;
                                                        RegWrite <= 1'b1;
                                                        ALUOp <= 5'b10001;
                                                        end
                                        6'b100101:  begin //OR
                                                        MemWrite <= 1'b0;
                                                        MemRead <= 1'b0;
                                                        MemToReg <= 1'b0;
                                                        RegDst <= 1'b0;
                                                        ALUSrc <= 1'b0;
                                                        RegWrite <= 1'b1;
                                                        ALUOp <= 5'b11110;
                                                        end
```

```
                        6'b100110:  begin //XOR
                                          MemWrite <= 1'b0;
                                          MemRead <= 1'b0;
                                          MemToReg <= 1'b0;
                                          RegDst <= 1'b0;
                                          ALUSrc <= 1'b0;
                                          RegWrite <= 1'b1;
                                          ALUOp <= 5'b10110;
                                          end
                        6'b000000:  begin //SLL
                                          MemWrite <= 1'b0;
                                          MemRead <= 1'b0;
                                          MemToReg <= 1'b0;
                                          RegDst <= 1'b0;
                                          ALUSrc <= 1'b1;
                                          RegWrite <= 1'b1;
                                          ALUOp <= 5'b01000;
                                          end
                        6'b000010:  begin //SRL
                                          MemWrite <= 1'b0;
                                          MemRead <= 1'b0;
                                          MemToReg <= 1'b0;
                                          RegDst <= 1'b0;
                                          ALUSrc <= 1'b1;
                                          RegWrite <= 1'b1;
                                          ALUOp <= 5'b01001;
                                          end
                        6'b000011:  begin //SRA
                                          MemWrite <= 1'b0;
                                          MemRead <= 1'b0;
                                          MemToReg <= 1'b0;
                                          RegDst <= 1'b0;
                                          ALUSrc <= 1'b1;
                                          RegWrite <= 1'b1;
                                          ALUOp <= 5'b01011;
                                          end
                        6'b101010:  begin //SLT
                                          MemWrite <= 1'b0;
                                          MemRead <= 1'b0;
                                          MemToReg <= 1'b0;
                                          RegDst <= 1'b0;
                                          ALUSrc <= 1'b0;
                                          RegWrite <= 1'b1;
                                          ALUOp <= 5'b00111;
                                          end
                        default:    begin
                                          MemWrite <= 1'b0;
                                          MemRead <= 1'b0;
                                          MemToReg <= 1'b0;
                                          RegDst <= 1'b0;
                                          ALUSrc <= 1'b0;
                                          RegWrite <= 1'b0;
                                          ALUOp <= 5'b00000;
                                          end
                        endcase
                  end
6'b001000:  begin //ADDI
```

```
                  MemWrite <= 1'b0;
                  MemRead <= 1'b0;
                  MemToReg <= 1'b0;
                  RegDst <= 1'b1;
                  ALUSrc <= 1'b1;
                  RegWrite <= 1'b1;
                  ALUOp <= 5'b00000;
                  end
6'b001100:  begin //ANDI
                  MemWrite <= 1'b0;
                  MemRead <= 1'b0;
                  MemToReg <= 1'b0;
                  RegDst <= 1'b1;
                  ALUSrc <= 1'b1;
                  RegWrite <= 1'b1;
                  ALUOp <= 5'b11000;
                  end
6'b001101:  begin //ORI
                  MemWrite <= 1'b0;
                  MemRead <= 1'b0;
                  MemToReg <= 1'b0;
                  RegDst <= 1'b1;
                  ALUSrc <= 1'b1;
                  RegWrite <= 1'b1;
                  ALUOp <= 5'b11110;
                  end
6'b001110:  begin //XORI
                  MemWrite <= 1'b0;
                  MemRead <= 1'b0;
                  MemToReg <= 1'b0;
                  RegDst <= 1'b1;
                  ALUSrc <= 1'b1;
                  RegWrite <= 1'b1;
                  ALUOp <= 5'b10110;
                  end
6'b100011:  begin //LW
                  MemWrite <= 1'b0;
                  MemRead <= 1'b1;
                  MemToReg <= 1'b1;
                  RegDst <= 1'b1;
                  ALUSrc <= 1'b1;
                  RegWrite <= 1'b1;
                  ALUOp <= 5'b00000;
                  end
6'b101011:  begin //SW
                  MemWrite <= 1'b1;
                  MemRead <= 1'b0;
                  MemToReg <= 1'b0;
                  RegDst <= 1'b0;
                  ALUSrc <= 1'b1;
                  RegWrite <= 1'b0;
                  ALUOp <= 5'b00000;
                  end
default:    begin
                  MemWrite <= 1'b0;
                  MemRead <= 1'b0;
                  MemToReg <= 1'b0;
```

```
                        RegDst <= 1'b0;
                        ALUSrc <= 1'b0;
                        RegWrite <= 1'b0;
                        ALUOp <= 5'b00000;
      end
      endcase
end
end

endmodule
```

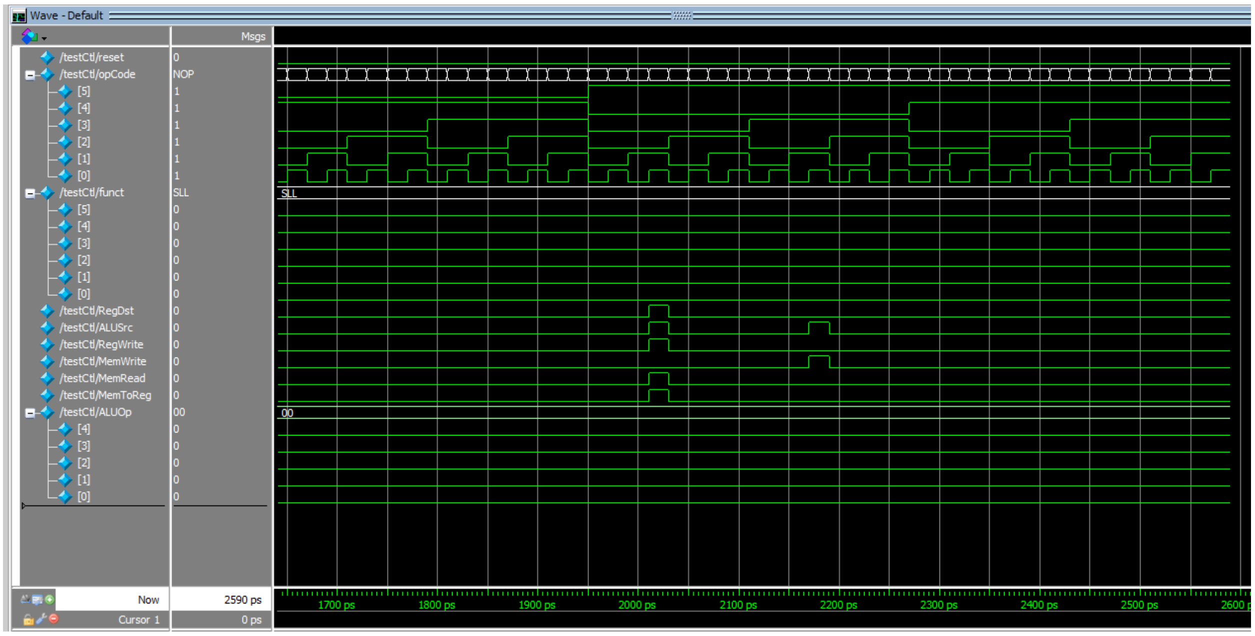

Figure 3: Waveform diagram for control

Lab 3 Section:

The first thing to do was declare any logic that was not instantiated by the input and output logics. Then I wired the submodules for pc, ctl, and regfile; I would wire the alu subcircuit later. The wiring for the pc submodule was straightforward because the names of the wires were the same. For the ctl submodule, I had to specify the bitfields that would correspond to the opcode and funct input logics. For regfile submodule, that was the most different. The three inputs for Rt, Rs, and Rd had to be wired with the correct bitfields from instruction memory. Also, the input logic that writes data to the register was modified, as well as the second output for read register data. The second input that would go into the alu submodule would be determined by muxes. If ALUSrc control signal was equal to 1, then we either zero pad or sign extend the corresponding bit fields of the instruction memory. For zero padding fields [10:6], we check if the opcode is a 6-bit zero (r-type) and that the funct codes were related to logical shifting. For zero padding fields [15:0], we check if the opcode corresponds to Boolean operations using immediate. All other cases when ALUSrc is still equal to 1, sign extension occurs for bits [15:0] for operations like addi, load word, or store word. If ALUSrc was equal to 0, the output from the read data 2 port goes into the second input of the ALU. The alu submodule is declared after the second input for the ALU was declared. Finally, depending on the value of the MemToReg control signal, either the address computed by the ALU or the read data from memory would be fed back into the register block.

One issue that was seen in development was that we cannot assign a value to input logics declared at the beginning of the module. I think this issue is implying that no port can have two or more wires reading and writing to it. The workaround to the problem was to declare new logics that stored the temporary values for later use. I also found that we cannot declare submodules inside always_comb blocks since system Verilog needs to build all the hardware at once; we either have that subcircuit or we don't kind of thing.

```
module beta(input logic clk, reset,
                  input logic [31:0] id, memReadData,
                  output logic [31:0] ia, memAddr, memWriteData,
                  output logic MemRead, MemWrite);

logic RegDst, RegWrite, ALUSrc, MemToReg, z, v, n;
logic [4:0] ALUOp;
logic [31:0] radata, rbdata, registerWriteData;

pc xpc(.clk(clk),.reset(reset),.ia(ia));

ctl xctl(.reset(reset),.opCode(id[31:26]),.funct(id[5:0]),.RegDst(RegDst),
      .ALUSrc(ALUSrc),.RegWrite(RegWrite),.MemWrite(MemWrite),
      .MemRead(MemRead),.MemToReg(MemToReg),.ALUOp(ALUOp));

regfile xregfile(.clk(clk),.RegWrite(RegWrite),.RegDst(RegDst),
      .ra(id[25:21]),.rb(id[20:16]),.rc(id[15:11]),.wdata(registerWriteData),
      .radata(radata),.rbdata(memWriteData));

logic [31:0] aluinput2;

always_comb
begin
      if (ALUSrc == 1'b1) begin
            if ((id[31:26] == 6'b000000) && ((id[5:0] == 6'b000000)
                  || (id[5:0] == 6'b000010) || (id[5:0] == 6'b000011)))
            begin
                  aluinput2 <= {27'd0,id[10:6]};       // zero pad [10:6]

            end
            else if ((id[31:26] == 6'b001100) || (id[31:26] == 6'b001101)
                  || (id[31:26] == 6'b001110))
            begin
                  aluinput2 <= {16'd0,id[15:0]};       // zero pad [15:0]
            end
            else
                  aluinput2 <= {{16{id[15]}},id[15:0]};      // sign extend
[15:0]
      end
      else begin
            aluinput2 <= memWriteData;
      end
end

alu
xalu(.A(radata),.B(aluinput2),.ALUOp(ALUOp),.Y(memAddr),.z(z),.v(v),.n(n));

always_comb
```

```
begin
        if (MemToReg == 1'b0) begin
        registerWriteData <= memAddr; end
        else begin
        registerWriteData <= memReadData; end
end

endmodule
```

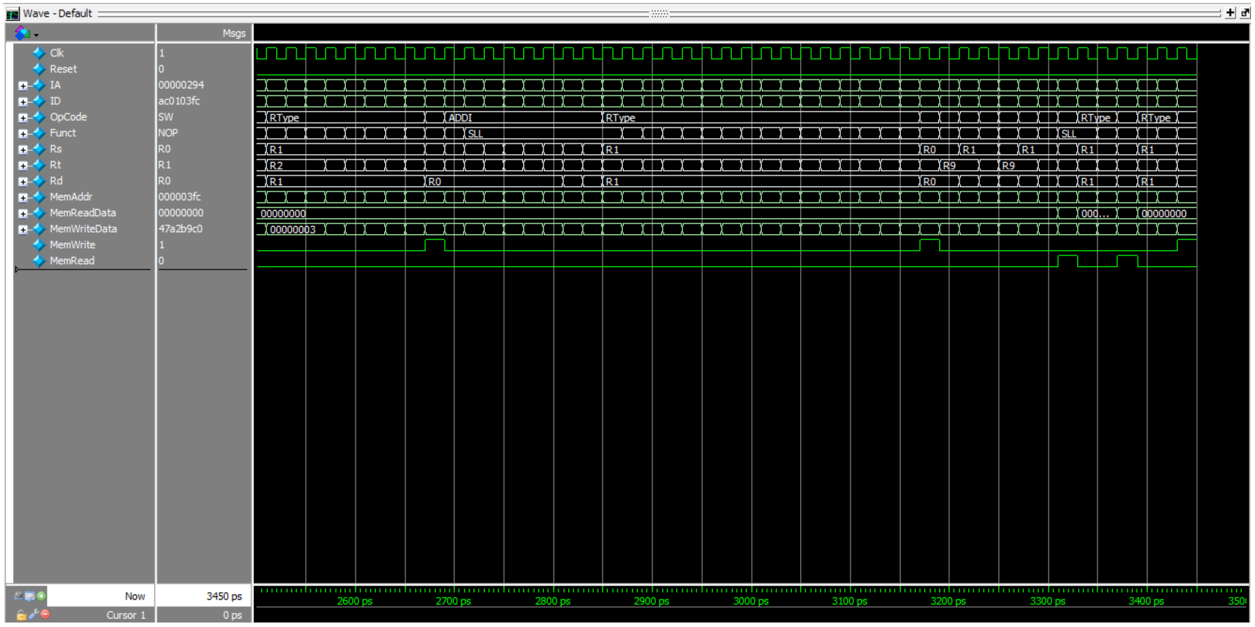

Figure 4: Waveform diagram for lab 3

I believe that during the development of my design. Many of the hardcoding inside the ctl.sv could have been mitigated. Although I hardcoded all the test cases that were required, additional modification or error correcting regarding opcodes and funct control signals would most likely be an issue for further development in the future. On the other hand, the way I have structured my read and write designs in regfile.sv would allow for easy additions for new input logic. If we wanted to add more read and write register ports, than the process of coding in that logic would not be very challenging because I can copy and paste most of the code. Also, for beta.sv, the area that may have the most potential for improvement is the implementation of the muxes. I think there is a more efficient way to check all the conditions from the opcodes and funct bitfields to determine if we need to sign extend or zero pad for an operation, especially if we have to deal with all test cases. My implementation only covers what was required, so if there is an alternative method without hardcoding for every case, that would be ideal. Overall, simulating the whole process was successful, so all test cases covered by the script were valid.