Darrian Johnson
ELEC 5200 – Project Part 3
3/17/2021

Program Counter

Main Code
See Appendix A

Testbench
See Appendix B

Simulation

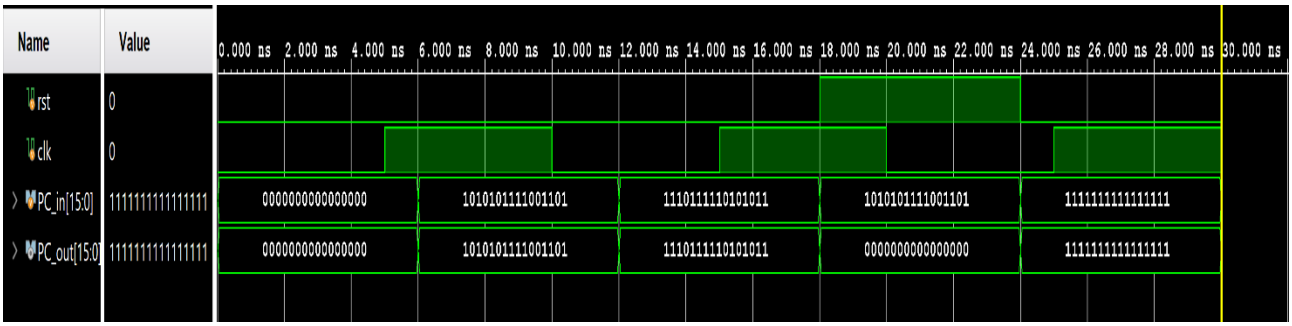

Figure 1. Simulation of Program counter over 30 ns.

The Program Counter (PC) takes in a 16-bit input (PC_in) and immediately sends that same value to a 16-bit output (PC_out). Therefore, at 6ns when PC_in = 1010_1011_1100_1101, PC_out outputs the same value. However, when reset is high, PC_out will reset to 0000_0000_0000_0000, no matter what value is input to the PC as shown in **Figure 1** at 18ns.

Adders

Main Code

See Appendix C

Testbench

See Appendix D

Simulation

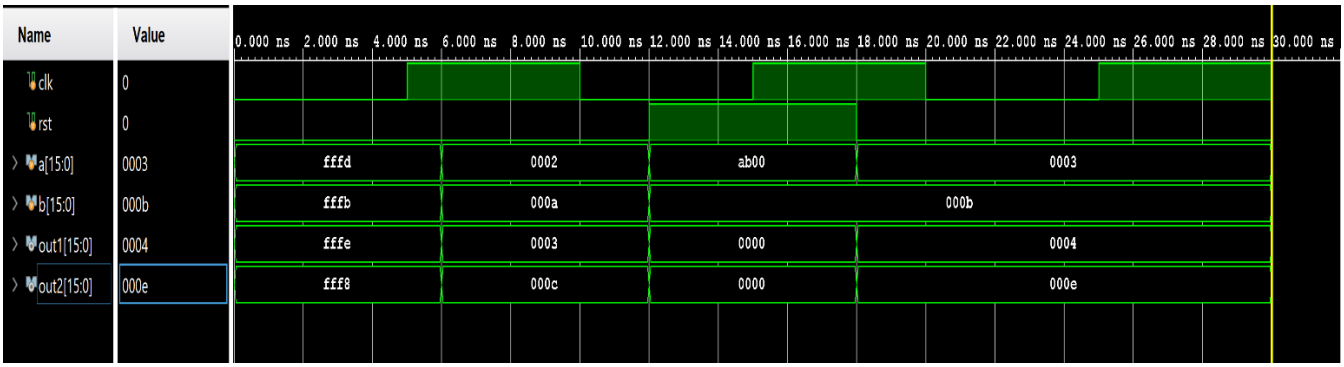

Figure 2. Simulation of Adders over 30 ns.

The Adder module has two different functions which is written to two different 16-bit outputs (out1 & out2). One adder takes a 16-bit input (a), adds 1 to it, and writes it to output (out1); whereas the other adder takes two 16-bit inputs (a & b), adds them together, and writes it to output (out2). At the start of the simulation, the functionality of the adders was tested using signed negative numbers -5 (1111_1111_1111_1101) and -3 (1111_1111_1111_1011). The simulation gave the following output:

$$a = -5$$
$$b = -3$$
$$out1 = a + 1'b1 = 1111111111111101 = \text{fffe}$$
$$out2 = a + b = 1111111111111101 + 1111111111111011 = \text{fff8}$$

However, when reset is high both outputs are reset to 0000_0000_0000_0000 as shown in **Figure 2** at 12ns.

2:1 MUX

Main Code

See Appendix E

Testbench

See Appendix F

Simulation

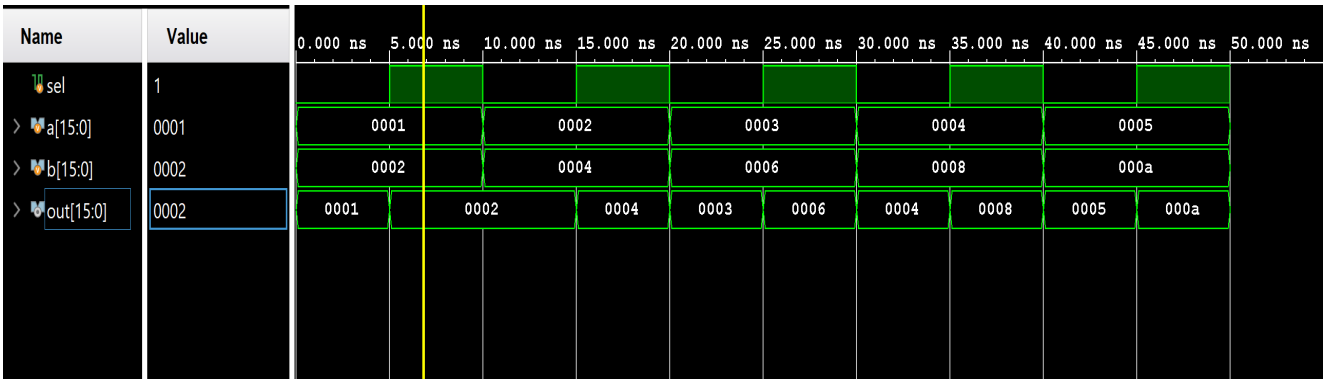

Figure 3. Simulation of 2-to-1 MUX over 50ns.

The MUX module takes in two 16-bit inputs (a & b) and a 1-bit select line (sel). One of the inputs is then written to a 16-bit output (out) depending on the value of the select line. When the select line is 0, then the output is a; however, when the select line is 1, then the output is b. In **Figure 3** from 0-5ns when the select line is 0, then the output is 0000_0000_0000_0001 (a); however, from 5-10ns when the select line is 1, then the output is 0000_0000_0000_0010 (b).

AND Gate

Main Code

See Appendix G

Testbench

See Appendix H

Simulation

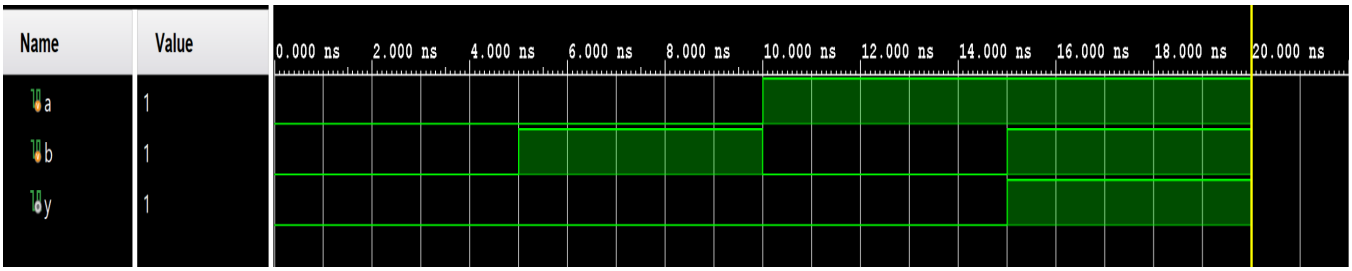

Figure 4. Simulation of AND gate over 25ns.

The AND gate module takes two 1-bit inputs (a & b), compares them using an AND gate, and writes the result to a 1-bit output (y). The simulation in Figure 4 shows the following results:

- 0 - 5ns, a = 0, b = 0, y = 0
- 5 - 10ns, a = 0, b = 1, y = 0
- 10 - 15ns, a = 1, b = 0, y = 0
- 15 – 20ns, a = 1, b = 1, y = 1

Control Unit

Main Code

See Appendix I

Testbench

See Appendix J

Simulation

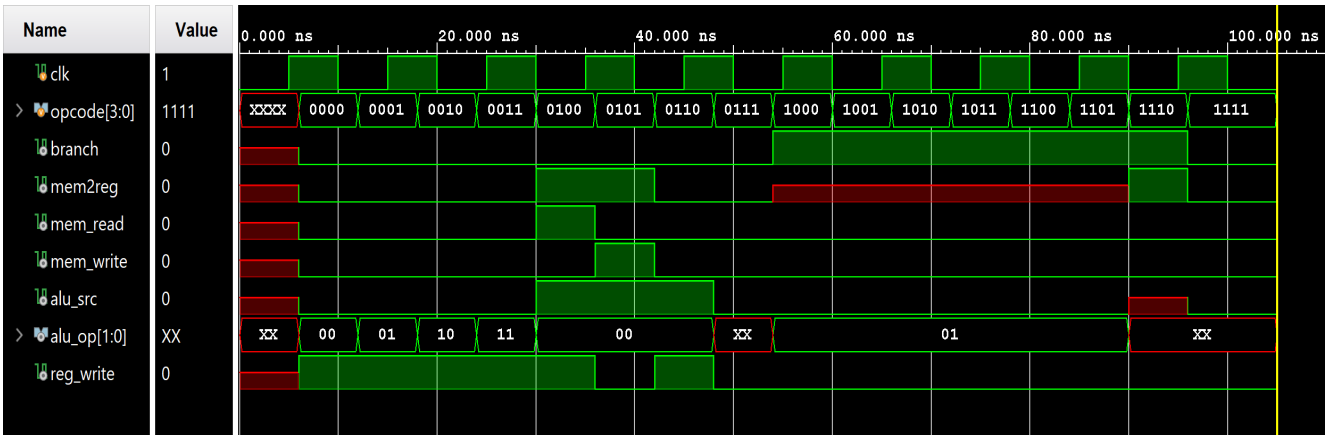

Figure 5. Simulation of the Control Unit over 105ns.

The Control Unit takes in a 4-bit input (opcode) and assigns it to several control signal outputs (branch, mem2reg, mem_read, mem_write, alu_src, alu_op, & reg_write). This portion of the project may vary based on the student's ISA. The control signals are shown below in **Table 1.** When observing the simulation in **Figure 5,** the opcode input 1110 (jmp instruction) matches the expected output values as shown in **Table 1.** The table can be used to verify the correct output of other input values in the simulation.

Table 1. Control Signals for Control Unit.

Instruction	Branch	Mem2Reg	MemRead	MemWrite	ALU Source	ALU Opcode	RegWrite
plus	0	0	0	0	0	00	1
min	0	0	0	0	0	01	1
and	0	0	0	0	0	10	1
or	0	0	0	0	0	11	1
ldw	0	1	1	0	1	00	1
stw	0	1	0	1	1	00	0
plusi	0	0	0	0	1	00	1
lui	0	0	0	0	0	xx	0
beq	1	x	0	0	0	01	0
bne	1	x	0	0	0	01	0
bgt	1	x	0	0	0	01	0
blt	1	x	0	0	0	01	0
bge	1	x	0	0	0	01	0
blte	1	x	0	0	0	01	0
jmp	1	1	0	0	x	xx	0
stop	0	0	0	0	0	xx	0

Register File

Main Code

See Appendix K

Testbench

See Appendix L

Simulation

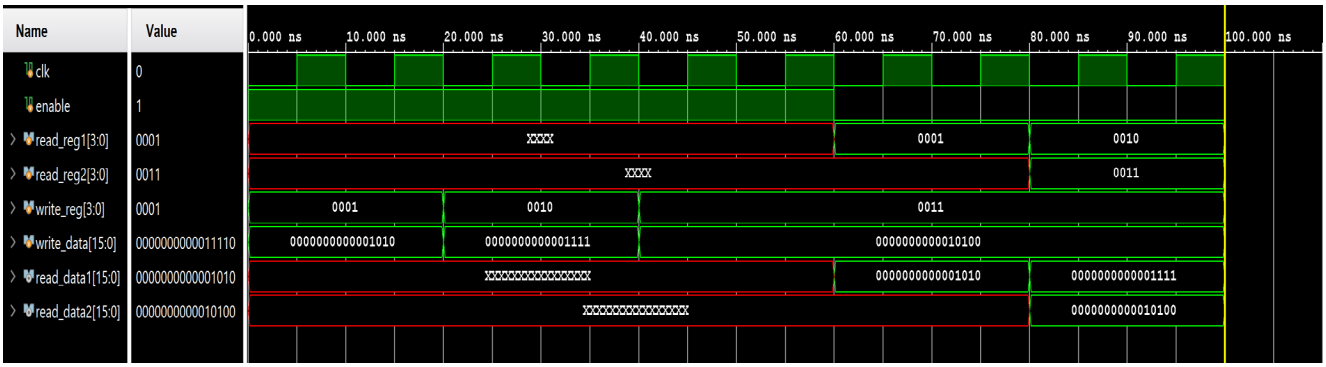

Figure 6. Simulation of Register File over 100ns.

The Register File module has three 4-bit inputs read_reg1, read_reg2, & reg_write),one 16-bit input (write_data), and an enable. The 1-bit enable line is used to signal a data write operation on the Register File. The enable line is active high. When there are valid register numbers in the output selection ports read_reg1 and read_reg2 then the Register File outputs the values in the corresponding registers to the output ports read_data1 and read_data2. When the enable line is high and there is a valid register number in the input selection port write_reg then the Register File writes the value on the input port write_data to the corresponding register in the Register File as shown in **Figure 6.**

ALU

Main Code

See Appendix M

Testbench

See Appendix N

Simulation

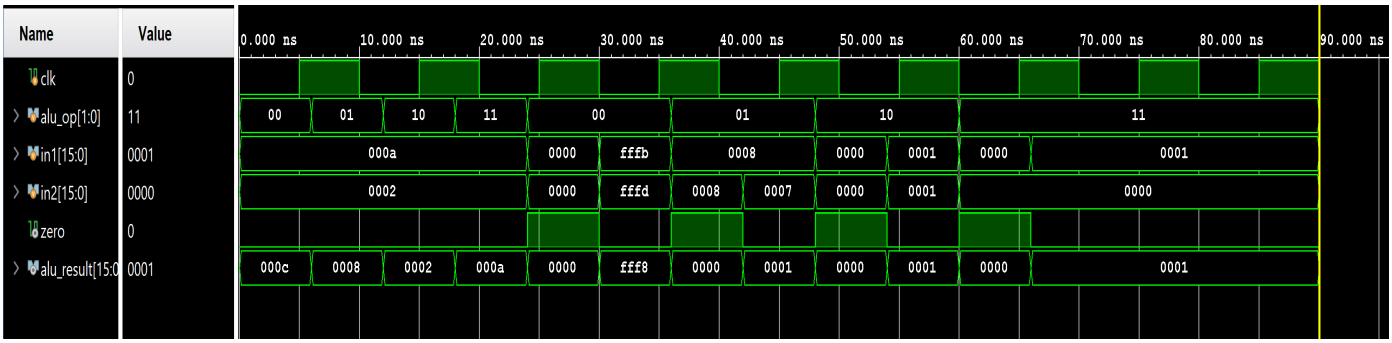

Figure 7. Simulation of ALU over 90ns.

The ALU takes in two 16-bit values (in1 & in2) and a 2-bit input (alu_op). The input alu_op performs one of the following four functions on inputs in1 & in2: ADD, SUB,AND,OR. The result is then output to a 16-bit output (alu_result) and a 1-bit output (zero). The output zero only goes high when alu_result equals 0. The following test was performed on the simulation to test functionality at 25ns as shown in **Figure 7:**

$$in1 = 0000$$
$$in2 = 0000$$
$$alu_{op} = 00 = ADD$$
$$alu_{result} = in1 + in2 = 0000$$
$$0 = 0, so\ zero = 1$$

The following test was performed on the simulation to test functionality at 30ns as shown in **Figure 7:**

$$in1 = -3 = \text{fffb}$$
$$in2 = -5 = \text{fffd}$$
$$alu_{op} = 00 = ADD$$
$$alu_{result} = in1 + in2 = -8 = = \text{fffe}$$
$$-8 \neq 0, so\ zero = 0$$

Immediate Generator

Main Code

See Appendix O

Testbench

See Appendix P

Simulation

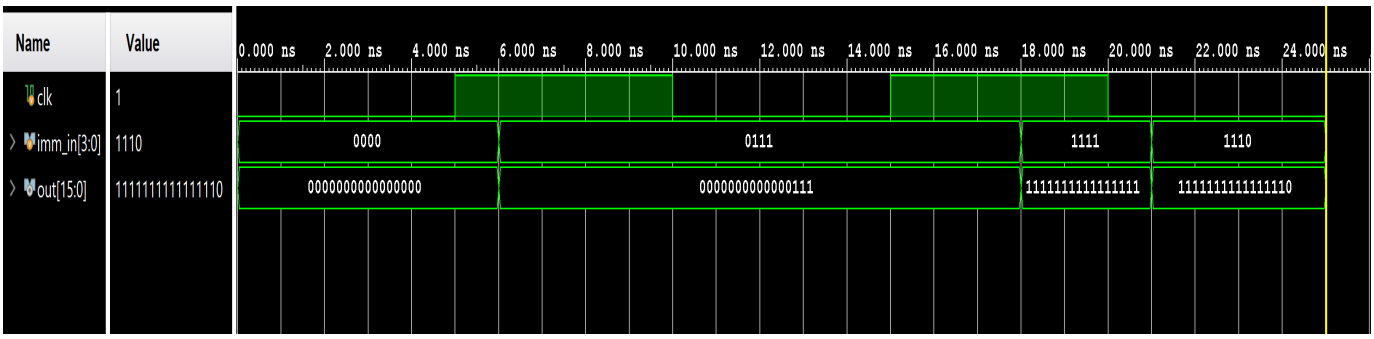

Figure 8. Simulation of Immediate Generator over 25ns.

The Immediate Generator takes in one 4-bit input (imm_in) and extends the MSB of that input by 12-bits to produce a 16-bit output (out). As seen in **Figure 8**, the following outputs are produced in the simulation:

- Imm_in = 0000, out = 0000000000000000
- Imm_in = 0111, out = 0000000000000111
- Imm_in = 1111, out = 1111111111111111
- Imm_in = 1110, out = 1111111111111110

The MSB is highlighted in green, and the copied bits are highlighted in red.

Appendix A

```
module PC(
    input rst,clk,
    input [15:0] PC_in,
    output reg [15:0] PC_out
    );
initial
    begin
        PC_out <= 16'h0000;
    end
always @ (*)
    begin
        if(rst) PC_out <= 16'h0000;
        else PC_out = PC_in;
    end
endmodule
```

Appendix B

```
module PC_tb;
    reg rst, clk;
    reg [15:0] PC_in;
    wire [15:0] PC_out;

PC uut(.rst(rst),.PC_in(PC_in),.PC_out(PC_out));
initial
    begin
        clk = 0;
        rst = 0;
        forever
        #5 clk = ~clk;
    end
initial
    begin
        PC_in = 16'h0;
        #6;
        PC_in = 16'habcd;
        #6;
        PC_in = 16'hefab;
        #6;
        PC_in = 16'habcd;
        rst = 1;
        #6;
        PC_in = 16'hffff;
        rst = 0;
        #6;
    end
endmodule
```

Appendix C

```
module Adders(
    input rst, clk,
    input [15:0] a, b,
    output reg [15:0] out1, out2
);
always @ (clk or rst)
    begin
        if(rst) begin
            out1 <= 16'h0000;
            out2 <= 16'h0000;
        end
        else if (clk)
            begin
                out1 <= a + 1'b1;
                out2 <= a + b;
            end
    end
endmodule
```

Appendix D

```
module Adders_tb;
    reg clk,rst;
    reg [15:0] a, b;
    wire [15:0] out1,out2;

Adders uut(.clk(clk),.rst(rst),
           .a(a),.b(b),
           .out1(out1),.out2(out2));
initial
    begin
        clk = 0;
        rst = 0;
        forever
        #5 clk = ~clk;
    end
initial
    begin
        a = 16'h1;
        b = 16'h000f;
        #6;
        a = 16'h2;
        b = 16'h000a;
        #6;
        a = 16'hab00;
        b = 16'h000b;
        rst = 1;
        #6
        a = 16'h3;
        b = 16'h000b;
        rst = 0;
        #6;
    end
endmodule
```

Appendix E

```
module Mux(
    input [15:0] a,b,
    input wire sel,
    output reg [15:0] out
    );
always @ (a or b or sel)
    begin
        case (sel)
            1'b0: out <= a;
            1'b1: out <= b;
        endcase
    end
endmodule
```

Appendix F

```
module Mux_tb;
    reg sel;
    reg [15:0] a,b;
    wire [15:0] out;
    reg count = 1'b1;

Mux uut(.a(a),.b(b),.sel(sel),.out(out));
initial
    begin
        a = 16'h0; b = 16'h0; sel = 1'b0;
        for(count = 1'b0;count<256;count = count+1'b1) begin
            a = a + 16'h01;
            b = b + 16'h02;
            #10;
        end
    end
 always #5 sel = ~sel;
endmodule
```

Appendix G

```
module AND_gate(
    input a, b,
    output reg y
    );
always @ (a or b)
    begin
        if(a == 1'b1 && b == 1'b1) begin
            y = 1'b1;
        end
        else
            y = 1'b0;
    end
endmodule
```

Appendix H

```
module AND_gate_tb;
    reg a, b;
    wire y;

AND_gate AND1(.a(a),.b(b),.y(y));

initial
    begin
        a = 0; b = 0;
        a = 0; b = 0; #5;
        a = 0; b = 1; #5;
        a = 1; b = 0; #5;
        a = 1; b = 1; #5;
    end
endmodule
```

Appendix I

```
module Control_Unit(
    input [3:0] opcode,
    output reg [1:0] alu_op,
    output reg branch,mem2reg,mem_read,mem_write,
               alu_src,reg_write
    );
always @ (*)
    begin
        case(opcode)
            4'b0000:    //plus
                begin
                     branch = 1'b0;
                     mem2reg = 1'b0;
                     mem_read = 1'b0;
                     mem_write = 1'b0;
                     alu_src = 1'b0;
                     alu_op = 2'b00;
                     reg_write = 1'b1;
                end
            4'b0001:    //min
                begin
                     branch = 1'b0;
                     mem2reg = 1'b0;
                     mem_read = 1'b0;
                     mem_write = 1'b0;
                     alu_src = 1'b0;
                     alu_op = 2'b01;
                     reg_write = 1'b1;
                end
            4'b0010:    //and
                begin
                     branch = 1'b0;
                     mem2reg = 1'b0;
                     mem_read = 1'b0;
                     mem_write = 1'b0;
                     alu_src = 1'b0;
                     alu_op = 2'b10;
                     reg_write = 1'b1;
                end
            4'b0011:    //or
                begin
                     branch = 1'b0;
                     mem2reg = 1'b0;
                     mem_read = 1'b0;
                     mem_write = 1'b0;
                     alu_src = 1'b0;
                     alu_op = 2'b11;
                     reg_write = 1'b1;
                end
            4'b0100:    //ldw
                begin
                     branch = 1'b0;
                     mem2reg = 1'b1;
                     mem_read = 1'b1;
                     mem_write = 1'b0;
```

```
            alu_src = 1'b1;
            alu_op = 2'b00;
            reg_write = 1'b1;
        end
    4'b0101:    //stw
        begin
            branch = 1'b0;
            mem2reg = 1'b1;
            mem_read = 1'b0;
            mem_write = 1'b1;
            alu_src = 1'b1;
            alu_op = 2'b00;
            reg_write = 1'b0;
        end
    4'b0110:    //plusi
        begin
            branch = 1'b0;
            mem2reg = 1'b0;
            mem_read = 1'b0;
            mem_write = 1'b0;
            alu_src = 1'b1;
            alu_op = 2'b00;
            reg_write = 1'b1;
        end
    4'b0111:    //lui
        begin
            branch = 1'b0;
            mem2reg = 1'b0;
            mem_read = 1'b0;
            mem_write = 1'b0;
            alu_src = 1'b0;
            alu_op = 2'bxx;
            reg_write = 1'b0;
        end
    4'b1000:    //beq
        begin
            branch = 1'b1;
            mem2reg = 1'bx;
            mem_read = 1'b0;
            mem_write = 1'b0;
            alu_src = 1'b0;
            alu_op = 2'b01;
            reg_write = 1'b0;
        end
    4'b1001:    //bne
        begin
            branch = 1'b1;
            mem2reg = 1'bx;
            mem_read = 1'b0;
            mem_write = 1'b0;
            alu_src = 1'b0;
            alu_op = 2'b01;
            reg_write = 1'b0;
        end
    4'b1010:    //bgt
        begin
            branch = 1'b1;
```

```
            mem2reg = 1'bx;
            mem_read = 1'b0;
            mem_write = 1'b0;
            alu_src = 1'b0;
            alu_op = 2'b01;
            reg_write = 1'b0;
        end
    4'b1011:    //blt
        begin
            branch = 1'b1;
            mem2reg = 1'bx;
            mem_read = 1'b0;
            mem_write = 1'b0;
            alu_src = 1'b0;
            alu_op = 2'b01;
            reg_write = 1'b0;
        end
    4'b1100:    //bge
        begin
            branch = 1'b1;
            mem2reg = 1'bx;
            mem_read = 1'b0;
            mem_write = 1'b0;
            alu_src = 1'b0;
            alu_op = 2'b01;
            reg_write = 1'b0;
        end
    4'b1101:    //blte
        begin
            branch = 1'b1;
            mem2reg = 1'bx;
            mem_read = 1'b0;
            mem_write = 1'b0;
            alu_src = 1'b0;
            alu_op = 2'b01;
            reg_write = 1'b0;
        end
    4'b1110:    //jmp
        begin
            branch = 1'b1;
            mem2reg = 1'b1;
            mem_read = 1'b0;
            mem_write = 1'b0;
            alu_src = 1'bx;
            alu_op = 2'bxx;
            reg_write = 1'b0;
        end
    4'b1111:    //stop
        begin
            branch = 1'b0;
            mem2reg = 1'b0;
            mem_read = 1'b0;
            mem_write = 1'b0;
            alu_src = 1'b0;
            alu_op = 2'bxx;
            reg_write = 1'b0;
        end
```

Appendix J

```
module Control_Unit_tb;
    reg clk;
    reg [3:0] opcode;
    wire [1:0] alu_op;
    wire branch,mem2reg,mem_read,mem_write,
         alu_src,reg_write;

Control_Unit uut(.opcode(opcode),.alu_op(alu_op),
                 .branch(branch),.mem2reg(mem2reg),
                 .mem_read(mem_read),.mem_write(mem_write),
                 .alu_src(alu_src),.reg_write(reg_write));
initial
    begin
        clk = 0;
        forever
        #5 clk = ~clk;
    end
initial
    begin
        #6 opcode = 4'b0000;
        #6 opcode = 4'b0001;
        #6 opcode = 4'b0010;
        #6 opcode = 4'b0011;
        #6 opcode = 4'b0100;
        #6 opcode = 4'b0101;
        #6 opcode = 4'b0110;
        #6 opcode = 4'b0111;
        #6 opcode = 4'b1000;
        #6 opcode = 4'b1001;
        #6 opcode = 4'b1010;
        #6 opcode = 4'b1011;
        #6 opcode = 4'b1100;
        #6 opcode = 4'b1101;
        #6 opcode = 4'b1110;
        #6 opcode = 4'b1111;
    end
endmodule
```

Appendix K

```
module RegFile(
    input enable,clk,
    input [3:0] read_reg1,read_reg2,write_reg,
    input [15:0] write_data,
    output reg [15:0] read_data1,read_data2
    );
    reg [15:0] registers [0:15];    //16 16-bit registers
    integer i;
initial
    begin
        for(i=0;i<16;i=i+1)
            registers[i] <= 16'd0;
    end
always @ (posedge clk)
    begin
        if(enable) begin
            registers[write_reg] <= write_data;
        end
         //output selected registers
        assign read_data1 = registers[read_reg1];
        assign read_data2 = registers[read_reg2];
    end
endmodule
```

Appendix L

```
module regfile_tb;
//inputs
reg clk,enable;//rst;
reg [3:0] read_reg1,read_reg2,write_reg;
reg [15:0] write_data;
//outputs
wire [15:0] read_data1,read_data2;

RegFile UUT(.clk(clk),
            .enable(enable),
            .read_reg1(read_reg1),
            .read_reg2(read_reg2),
            .write_reg(write_reg),
            .write_data(write_data),
            .read_data1(read_data1),
            .read_data2(read_data2));
initial
    begin
        clk = 0;
        enable = 0;
        //rst = 0;
        forever
        #5 clk = ~clk;
    end
   initial
     begin
        write_reg = 1;
        enable = 1;
        write_data = 10;
        #20;
        write_reg = 2;
        enable = 1;
        write_data = 15;
        #20;
        write_reg = 3;
        enable = 1;
        write_data = 20;
        #20;
        read_reg1 = 1;
        enable = 0;
        #20;
        read_reg1 = 2;
        read_reg2 = 3;
        #20;
        read_reg1 = 1;
        write_reg = 1;
        enable = 1;
        write_data = 30;
        #20;
    end
endmodule
```

Appendix M

```
module ALU(
    input [15:0] in1,in2,
    input [1:0] alu_op,
    output reg [15:0] alu_result,
    output reg zero
    );
    wire [16:0] temp;

    //ALU Opcodes
    localparam ADD = 2'b00;
    localparam SUB = 2'b01;
    localparam AND = 2'b10;
    localparam OR = 2'b11;
always @ (*)
    begin
        case(alu_op)
            ADD: alu_result = in1 + in2;   //ADD
            SUB: alu_result = in1 - in2;   //SUB
            AND: alu_result = in1 & in2;   //AND
             OR: alu_result = in1 | in2;   //OR
             default: alu_result = in1 + in2; //default
        endcase
        if(alu_result == 0)
            zero = 1'b1;
        else
            zero = 1'b0;
    end
endmodule
```

Appendix N

```
module ALU_tb;
    reg clk;
    reg [1:0] alu_op;
    reg [15:0] in1,in2;
    wire zero;
    wire [15:0] alu_result;
    integer i = 2'd0;
ALU uut(.in1(in1),.in2(in2),
        .alu_op(alu_op),.alu_result(alu_result),
        .zero(zero));
initial
    begin
        clk = 0;
        forever
        #5 clk = ~clk;
    end
initial
    begin
        in1 = 16'h000a; in2 = 16'h0002; alu_op = 2'b0;
        for(i=0;i<=3;i=i+1'b1) begin
            alu_op = i;
            #6;
        end
//Test ADD: first add sets zero flag high and second add resets zero flag
        in1 = 16'h0;
        in2 = 16'h0;
        alu_op = 2'b00;
        #6;
        in1 = 16'hfffb;
        in2 = 16'hfffd;
        alu_op = 2'b00;
        #6;
//Test SUB: first sub sets zero flag high and second sub resets zero flag
        in1 = 16'h8;
        in2 = 16'h8;
        alu_op = 2'b01;
        #6;
        in1 = 16'h8;
        in2 = 16'h7;
        alu_op = 2'b01;
        #6;
//Test AND: first AND sets zero flag high and second AND resets zero flag
        in1 = 16'h0;
        in2 = 16'h0;
        alu_op = 2'b10;
        #6;
        in1 = 16'h1;
        in2 = 16'h1;
        alu_op = 2'b10;
        #6;
//Test OR: first OR sets zero flag high and second OR resets zero flag
        in1 = 16'h0;
        in2 = 16'h0;
        alu_op = 2'b11;
        #6;
```

```
        in1 = 16'h1;
        in2 = 16'h0;
        alu_op = 2'b11;
        #6;
    end
endmodule
```

Appendix O

```
module ImmGen(
    input clk,
    input [3:0] imm_in,
    output reg [15:0] out
    );
    wire [3:0] imm;
    assign imm = imm_in;
//sign extension of input assigned to output
always @ (*)
    begin
        out [15:0] <= {{12{imm[3]}}, imm[3:0]};
    end
endmodule
```

Appendix P

```
module ImmGen_tb;
//inputs
reg clk;
reg [3:0] imm_in;
//outputs
wire [15:0] out;

ImmGen UUT(.clk(clk),.imm_in(imm_in),.out(out));
    initial
        begin
            clk = 0;
            forever
            #5 clk = ~clk;
        end
    initial
        begin
            imm_in = 4'd0;
        end
    initial
        begin
            #6 imm_in = 4'b0111;
            #12 imm_in = 4'b1111;
            #3  imm_in = 4'b1110;
        end
endmodule
```