Tyler Thompson

ECE 3570 Lab 2
10-bit Instruction Set Architecture CPU - Components Implementation
February 21st, 2018

Design Description

1. **General Purpose Register File** - The general purpose register file instantiates eight registers and allows reading and writing to all registers. The reg_select input is three bits wide and selects which register to read or write two. The value of reg_select corresponds with registers zero to seven. In order to write to a register, en_write input should be set to one and a nine bit wide input data should be put on write_data. To read a register, set the reg_select input and make sure en_write is set to zero. The data output will show the value in the register selected.

2. **Program Counter Register** - The program counter is implemented and managed by the fetch unit. Any input to the fetch unit will result in and output of the program counter, after it has been updated.

3. **Arithmetic Logical unit** - The ALU is designed to do three basic operations needed by the ISA. It can perform addition, shifting, and comparing. The adder is implemented using a ripple adder and can handle two ten bit numbers. The shifter shifts the input left, unless the shift amount is detected to be negative, then it shifts the input right. The comparator will compare two inputs and determine which one is larger. If input a is larger than input b, the the output will be a one, otherwise the output is a zero. The ALU decides which operations to do based on a ALU op-code and a multiplexer. The ALU_op input is two bits wide and will default to just returning a zero if no op-code is supplied.

4. **Fetch Unit** - The fetch unit will update the program counter based on the a two bit wide op-code called pc_control. It uses a multiplexer on the pc_control to decide which operation to execute. It's supported operations are to update the program counter based on a register address, a jump address, or a branch address. By default, the current value of the program counter will be returned.

5. **Special Purpose Registers** - This design does not use any special purpose registers.

6. **Clock Signal** - The clock signal is only fed into the registers and fetch unit. During testing, a five nanosecond clock is used that has a period of ten nanoseconds.

Notes: No changes were made to the original design created in lab one.

Critical Path Delay

Register File: 6.8ns

ALU: 7.9ns (Longest Case)

Fetch Unit: 3.9ns

- All delays are represented as worst case. The clock cycle time will be the longest case, which is 7.9ns.

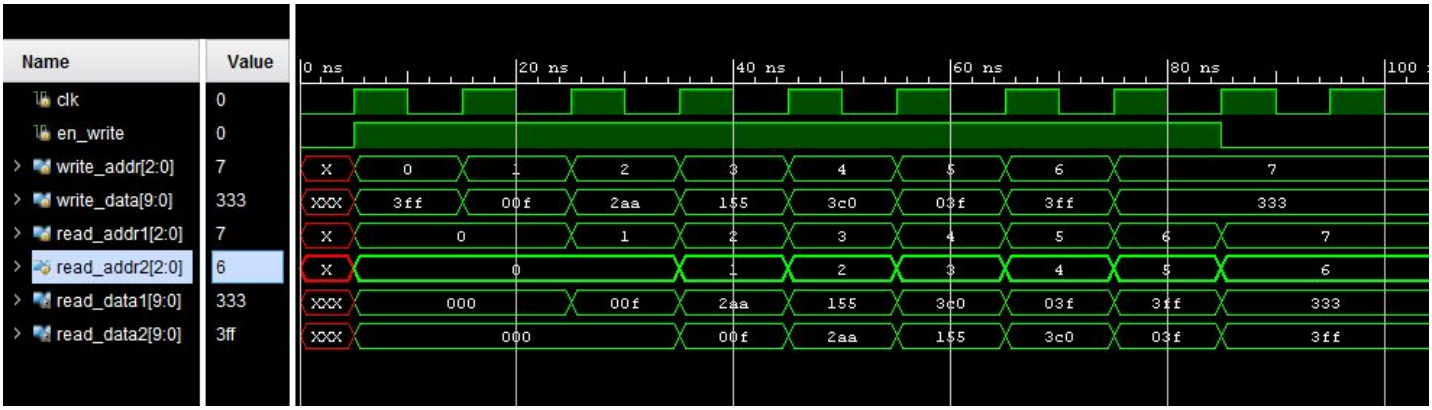
10-Bit ISA Reference Sheet

Instructions						
Name	**Description**	**Format**	**RTL**	**Syntax**	**OP Code (hex)**	**Function Code (hex)**
add	Add two values in registers together	R	if MEM[PC] == ADD rs rt rd [rs] <= [rt] + [rd] PC <= PC + 1	add $rs, $rt, $rd	0x0	
addi	Add a 3-bit signed contant to a value in a register	I	if MEM[PC] == ADDI rs rt imm [rs] <= [rt] + sign-ext(imm) PC <= PC + 1	addi $rs, $rt, imm	0x4	
sw	Store a word in memory	I	if MEM[PC] == SW rt offset(base) address = sign-extend(offset) + [base] MEM[address] <= [rt] PC <= PC + 1	sw $rt, M($rs)	0x5	
lw	Load a word from memory	I	if MEM[PC] == LW rt offset(base) address = sign-extend(offset) + [base] [rt] <= MEM[address] PC <= PC + 1	lw $rt, M($rs)	0x6	
sll	Shift left logical	R	if MEM[PC] == SLL rd rs rt [rd] <= [rs] << [rt] PC = PC + 1	sll $rd, $rs, $rt	0x1	
slt	Store less than	R	if MEM[PC] == SLT rd rs rt [rd] <= [rs] < [rt] PC = PC + 1	slt $rd, $rs, $rt	0x2	
beq	Branch equal	R	if MEM[PC] == BEQ rd rs rt if [rs] == [rt] PC <= PC + 4 + [rd]	beq $rd, $rs, $rt	0x3	
jal	Jump and link	J	if MEM[PC] == JAL address [ra] <= PC + 1 PC <= PC + address	jal Label	0x7	0x2
j	Jump	J	if MEM[PC] == J address PC <= PC + address	j Label	0x7	0x1
halt	Halt the machine	J	PC=HaltAddress if MEM[PC] == HALT PC <= 0x0	halt	0x7	0x3

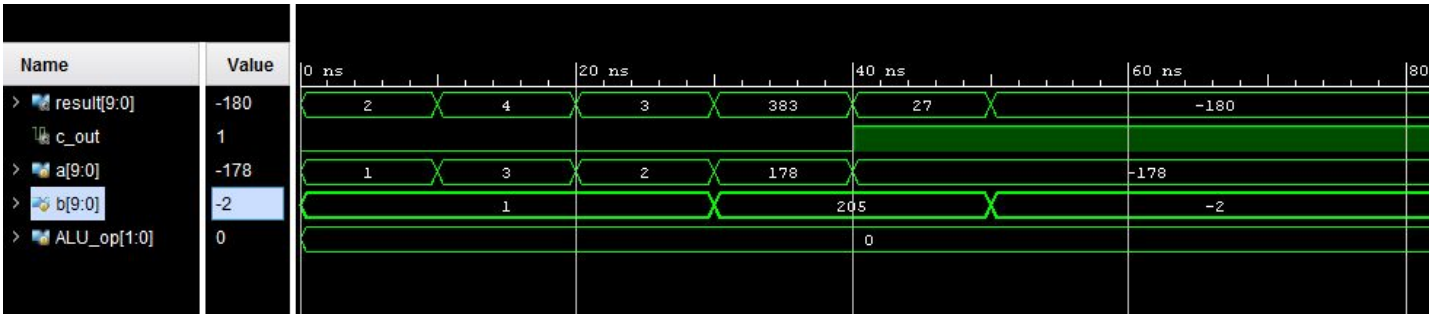
Timing Diagrams for Testing Scenarios

- See code section for testing parameters

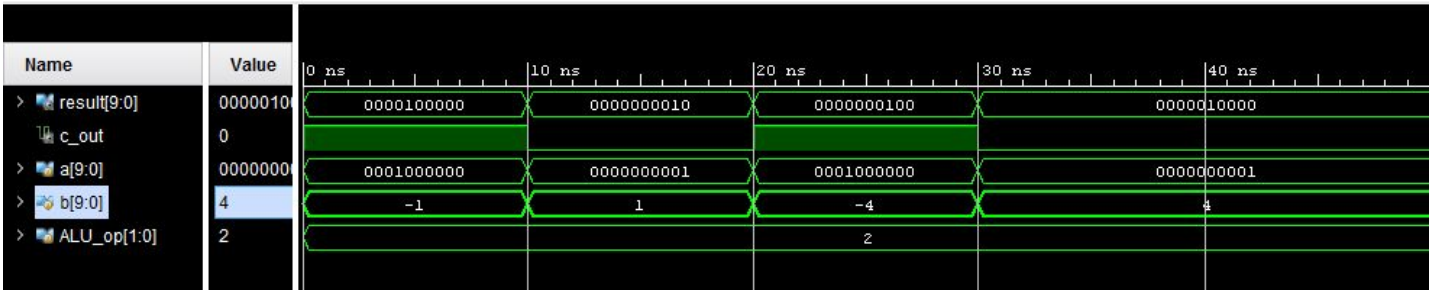
Register File Test



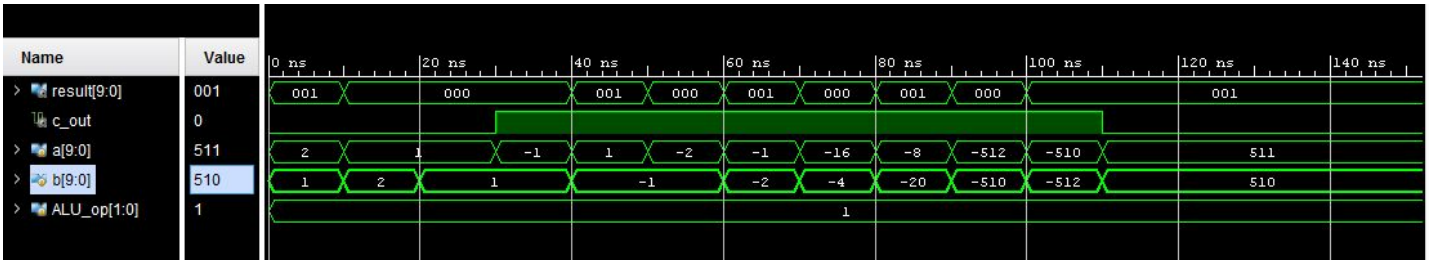
ALU - Adder Test



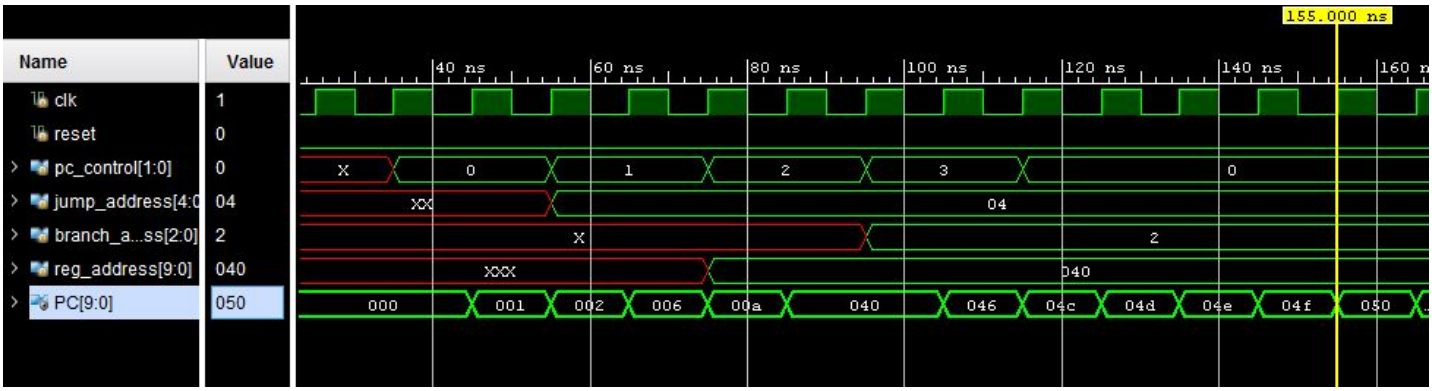
ALU - Shifter Test



ALU - Comparator Test

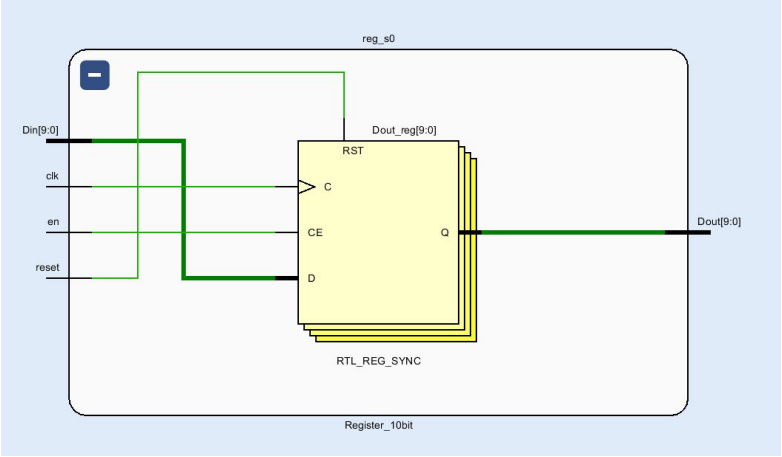


Fetch Unit Test

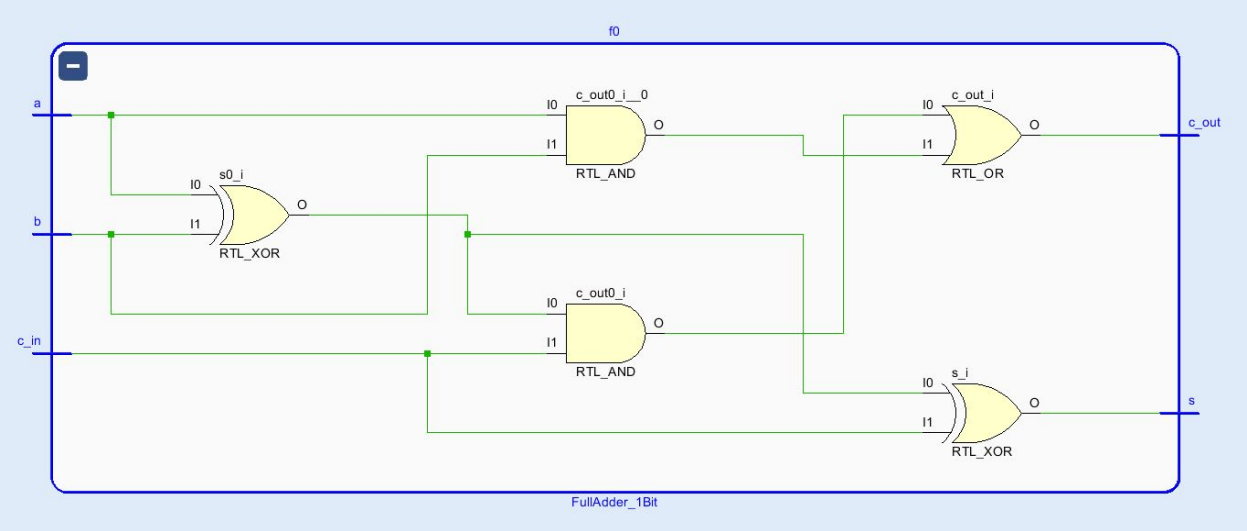


Schematics

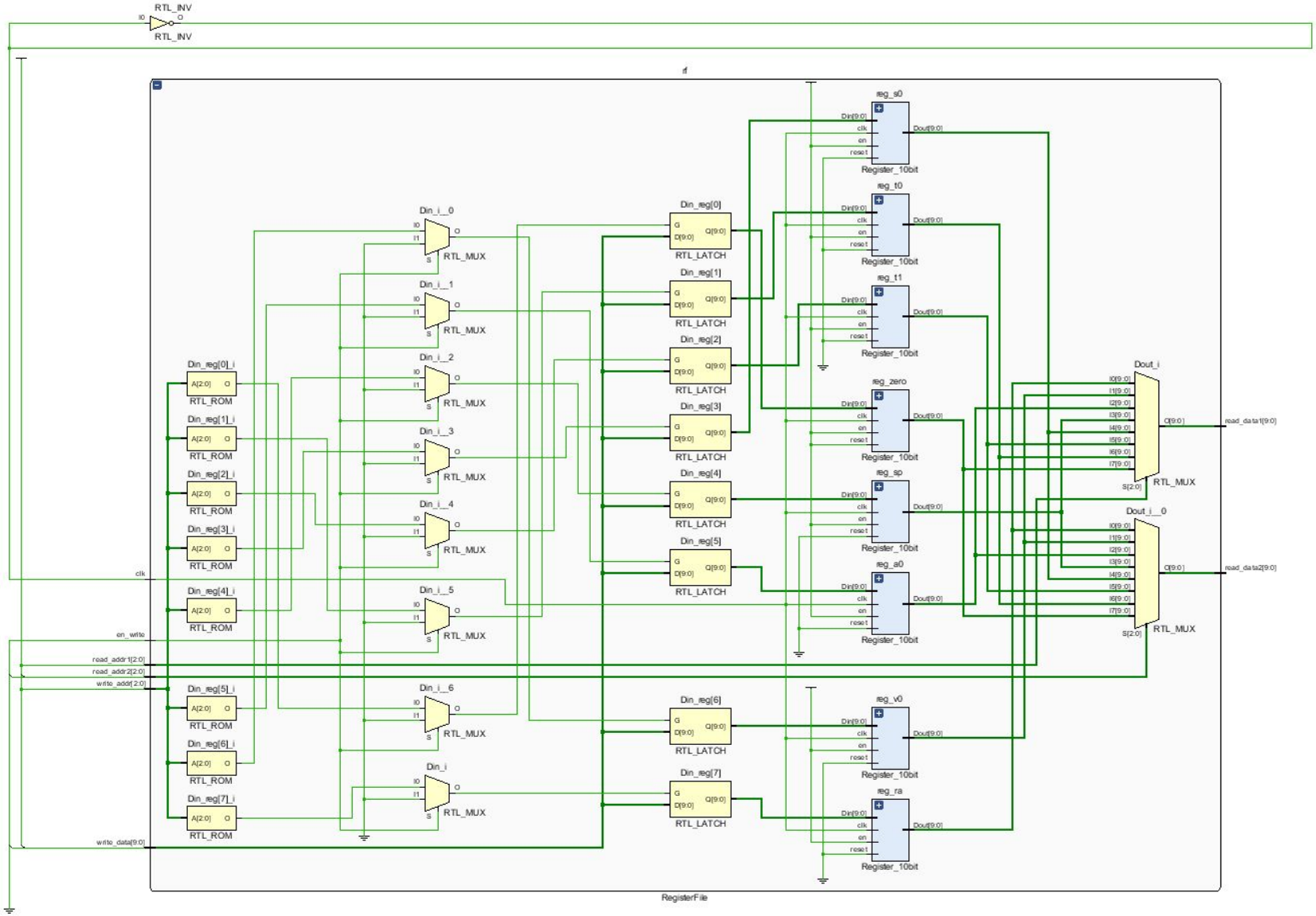
10 Bit Register



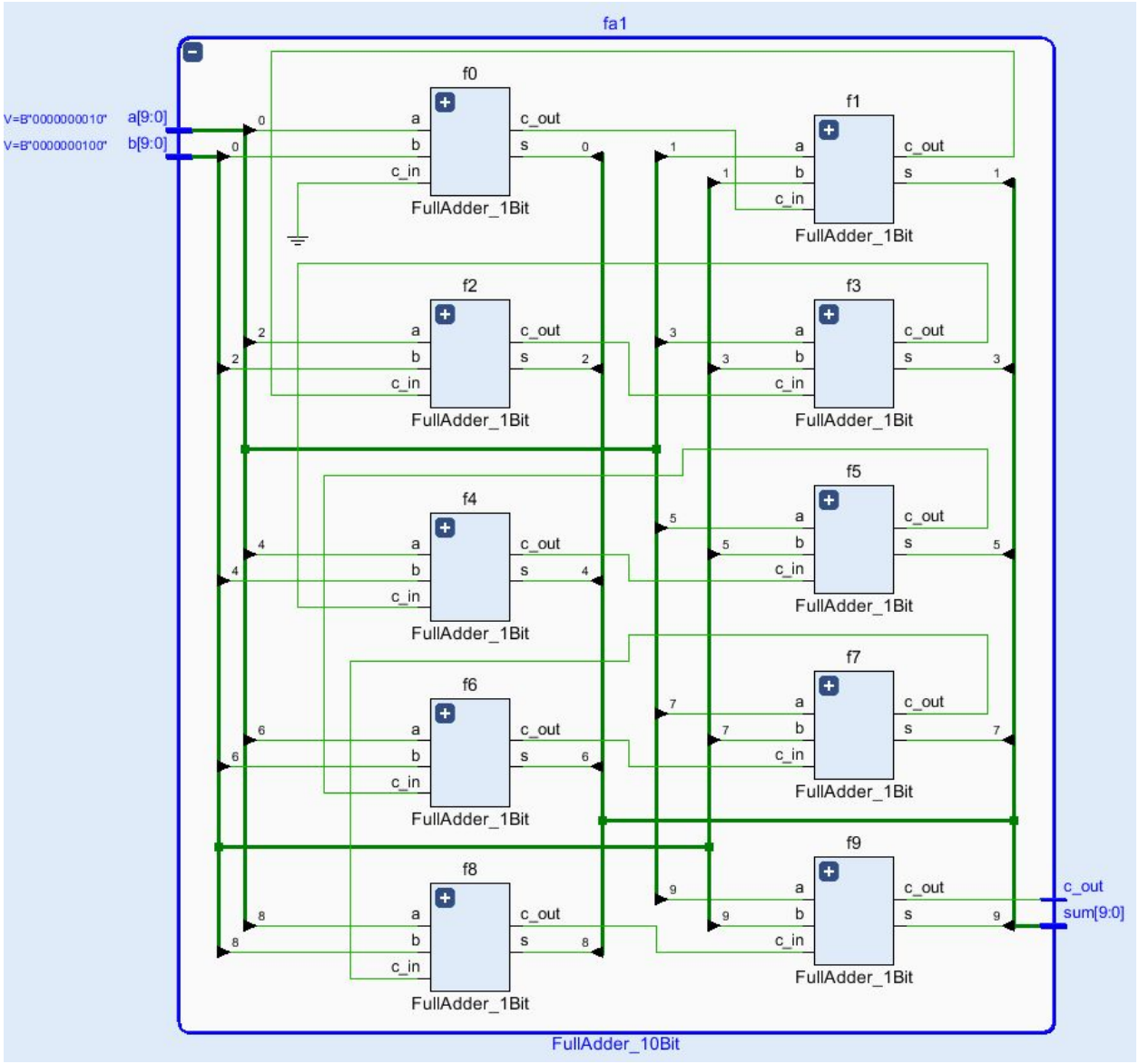
1 Bit Full Adder



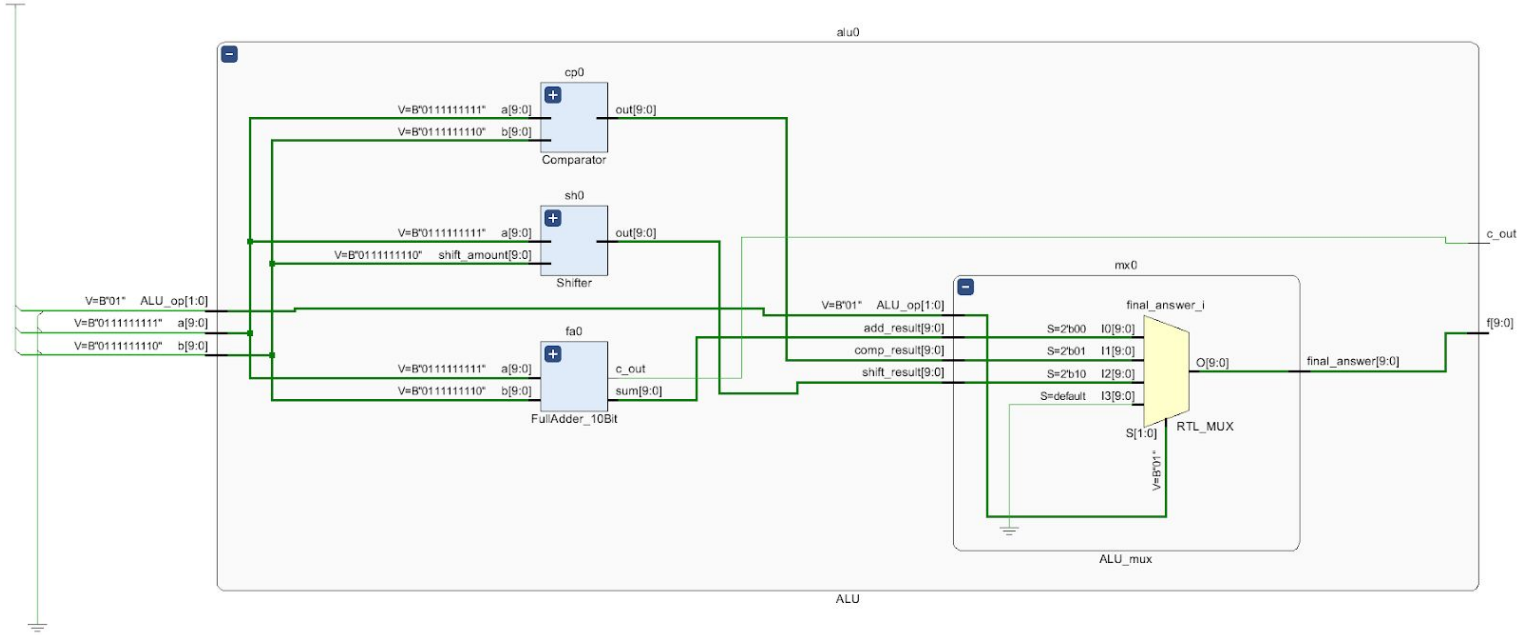
Register File



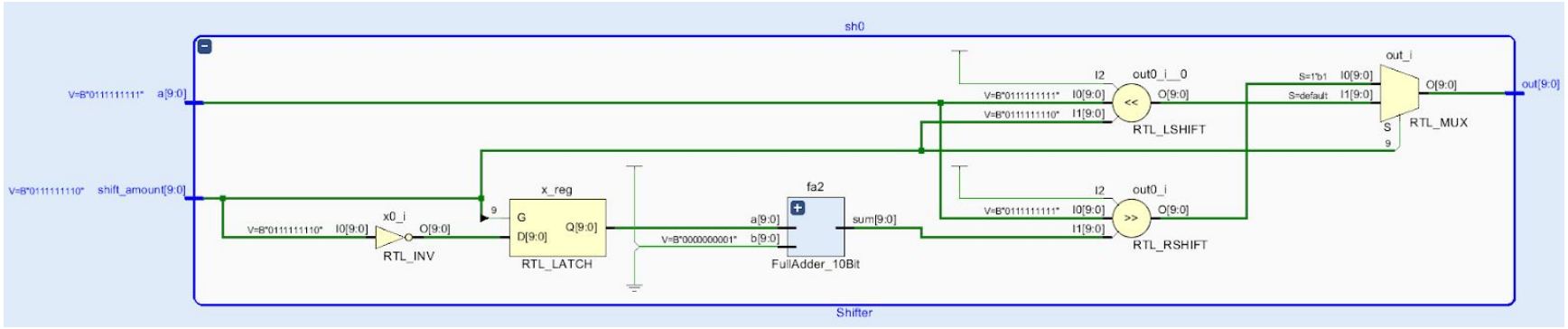
10 Bit Full Adder



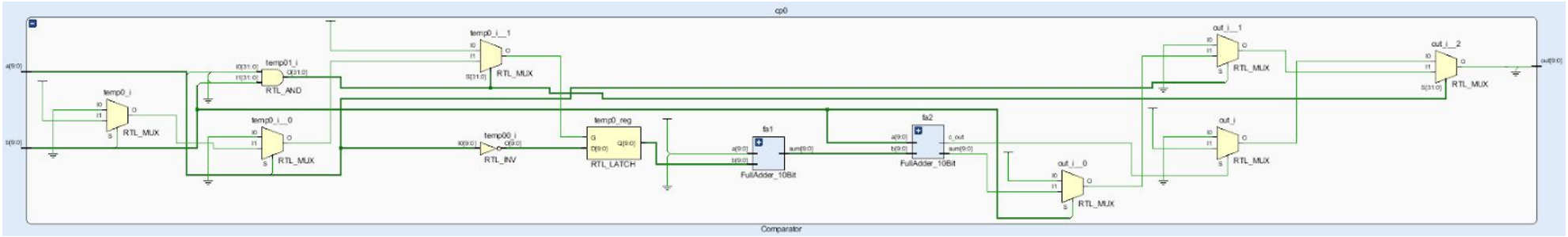
ALU


alu0
cp0
V=B"0111111111" a[9:0]
V=B"0111111110" b[9:0]
out[9:0]
Comparator
sh0
V=B"0111111111" a[9:0]
V=B"0111111110" shift_amount[9:0]
out[9:0]
Shifter
fa0
V=B"0111111111" a[9:0]
V=B"0111111110" b[9:0]
c_out
sum[9:0]
FullAdder_10Bit
c_out
mx0
V=B"01" ALU_op[1:0]
add_result[9:0]
comp_result[9:0]
shift_result[9:0]
final_answer_i
S=2'b00 I0[9:0]
S=2'b01 I1[9:0]
S=2'b10 I2[9:0]
S=default I3[9:0]
O[9:0]
S[1:0]
RTL_MUX
final_answer[9:0]
f[9:0]
ALU_mux
V=B"01" ALU_op[1:0]
V=B"0111111111" a[9:0]
V=B"0111111110" b[9:0]
ALU

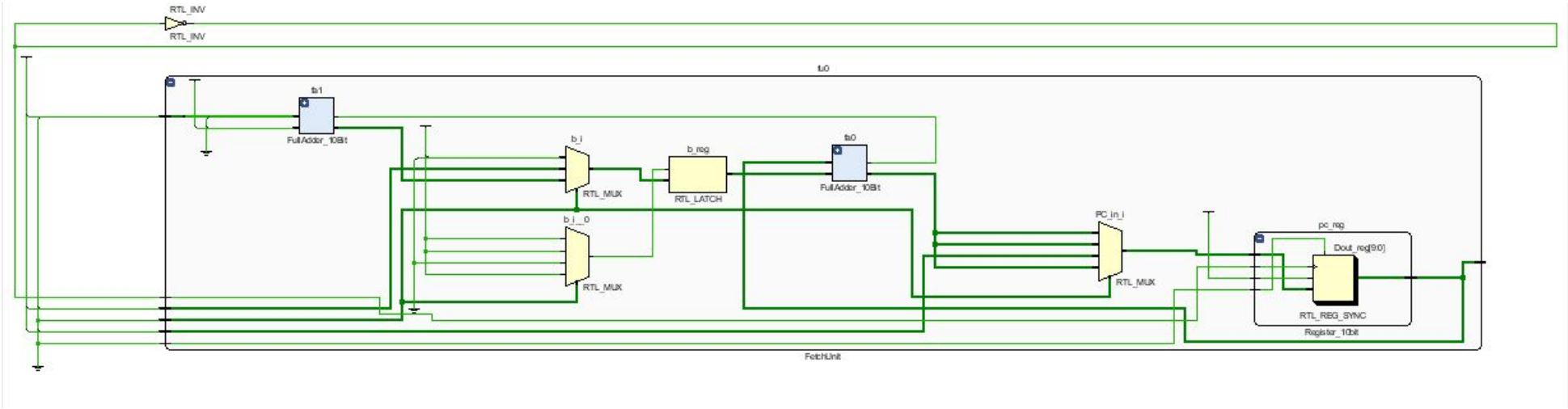
Shifter


sh0
V=B"0111111111" a[9:0]
V=B"0111111110" shift_amount[9:0]
x0_i
V=B"0111111110" I0[9:0]
O[9:0]
RTL_INV
x_reg
G
D[9:0]
Q[9:0]
RTL_LATCH
fa2
a[9:0]
b[9:0]
V=B"0000000001"
sum[9:0]
FullAdder_10Bit
I2
out0_i__0
V=B"0111111111" I0[9:0]
V=B"0111111110" I1[9:0]
<<
O[9:0]
RTL_LSHIFT
I2
out0_i
V=B"0111111111" I0[9:0]
I1[9:0]
>>
O[9:0]
RTL_RSHIFT
out_i
S=1'b1 I0[9:0]
S=default I1[9:0]
O[9:0]
S
RTL_MUX
out[9:0]
Shifter

Comparator



Fetch Unit



```
`timescale 1ns / 1ns
//////////////////////////////////////////////////////////////////////////////////
// Company: Western Michigan University
// Engineer: Tyler Thompson
//
// Create Date: 02/05/2018 07:30:26 PM
// Design Name:
// Module Name: Registers
// Project Name: ECE 3570 Lab 2
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module RegisterFileTest();
     reg clk;
     reg en_write;
     reg [2:0] write_addr;
     reg [9:0] write_data;
     reg [2:0] read_addr1;
     reg [2:0] read_addr2;
     wire [9:0] read_data1;
     wire [9:0] read_data2;

    //toggle clock every 5ns
    always #5 clk = ~clk;

    RegisterFile rf( .clk(clk), .en_write(en_write), .write_addr
(write_addr), .write_data(write_data), .read_addr1(read_addr1), .read_addr2
(read_addr2), .read_data1(read_data1), .read_data2(read_data2) );

    initial begin
        en_write = 0;
        clk = 0;
        #5;
        //prove you can't write to the zero register
        en_write = 1; write_addr = 3'b000; write_data = 10'b1111111111; read_addr1 =
3'b000; read_addr2 = 3'b000;
        #10;
        //write to $t0 and read $zero twice
        en_write = 1; write_addr = 3'b001; write_data = 10'b0000001111; read_addr1 =
3'b000; read_addr2 = 3'b000;
        #10;
        //write to $t1 and read $t0 and $zero
        en_write = 1; write_addr = 3'b010; write_data = 10'b1010101010; read_addr1 =
3'b001; read_addr2 = 3'b000;
        #10;
        //write to $s0 and read $t1 and $t0
        en_write = 1; write_addr = 3'b011; write_data = 10'b0101010101; read_addr1 =
3'b010; read_addr2 = 3'b001;
        #10;
        //write to $sp and read $s0 and $t1
        en_write = 1; write_addr = 3'b100; write_data = 10'b1111000000; read_addr1 =
3'b011; read_addr2 = 3'b010;
        #10;
         //write to $a0 and read $sp and $s0
        en_write = 1; write_addr = 3'b101; write_data = 10'b0000111111; read_addr1 =
```

```
3'b100; read_addr2 = 3'b011;
        #10;
        //write to $v0 and read $a0 and $sp
        en_write = 1; write_addr = 3'b110; write_data = 10'b1111111111; read_addr1 =
3'b101; read_addr2 = 3'b100;
        #10;
        //write to $ra and read $v0 and $a0
        en_write = 1; write_addr = 3'b111; write_data = 10'b1100110011; read_addr1 =
3'b110; read_addr2 = 3'b101;
        #10;
        //read from $ra and $v0
        en_write = 0; read_addr1 = 3'b111; read_addr2 = 3'b110;

    end
endmodule


// Register file, read and write data to registers
// addressing:  000 -> $zero
//              001 -> $t0
//              010 -> $t1
//              011 -> $s0
//              100 -> $sp
//              101 -> $a0
//              110 -> $v0
//              111 -> $ra
module RegisterFile(
    input wire clk,
    input wire en_write,
    input wire [2:0] write_addr,
    input wire [9:0] write_data,
    input wire [2:0] read_addr1,
    input wire [2:0] read_addr2,
    output reg [9:0] read_data1,
    output reg [9:0] read_data2
    );

      //Array of data input for registers
      reg [9:0]Din [7:0];

      //Array of data outputs for registers
      wire [9:0]Dout[7:0];

      wire reset = 0;
      //Initialize all general purpose registers
      Register_10bit reg_zero(.clk(clk), .reset(1'b1), .en(1'b1), .Din(Din[0]), .Dout
(Dout[0]));
      Register_10bit reg_t0(.clk(clk), .reset(reset), .en(1'b1), .Din(Din[1]), .Dout
(Dout[1]));
      Register_10bit reg_t1(.clk(clk), .reset(reset), .en(1'b1), .Din(Din[2]), .Dout
(Dout[2]));
      Register_10bit reg_s0(.clk(clk), .reset(reset), .en(1'b1), .Din(Din[3]), .Dout
(Dout[3]));
      Register_10bit reg_sp(.clk(clk), .reset(reset), .en(1'b1), .Din(Din[4]), .Dout
(Dout[4]));
      Register_10bit reg_a0(.clk(clk), .reset(reset), .en(1'b1), .Din(Din[5]), .Dout
(Dout[5]));
      Register_10bit reg_v0(.clk(clk), .reset(reset), .en(1'b1), .Din(Din[6]), .Dout
(Dout[6]));
      Register_10bit reg_ra(.clk(clk), .reset(reset), .en(1'b1), .Din(Din[7]), .Dout
(Dout[7]));

      always@(*) begin
            if ( en_write == 1 ) begin
```

```
                Din[write_addr] <= write_data;
            end
            read_data1 <= Dout[read_addr1];
            read_data2 <= Dout[read_addr2];
      end

endmodule


module RegisterTest();
  reg clk, reset;
  //toggle clock every 5ns
  always #5 clk = ~clk;

  //Array of data input for registers
  reg [9:0]Din [7:0];

  //Array of data outputs for registers
  wire [9:0]Dout[7:0];

  //Initialize all general purpose registers
  Register_10bit reg_zero(.clk(clk), .reset(1'b1), .en(1'b1), .Din(Din[0]), .Dout(Dout
[0]));
  Register_10bit reg_t0(.clk(clk), .reset(reset), .en(1'b1), .Din(Din[1]), .Dout(Dout
[1]));
  Register_10bit reg_t1(.clk(clk), .reset(reset), .en(1'b1), .Din(Din[2]), .Dout(Dout
[2]));
  Register_10bit reg_s0(.clk(clk), .reset(reset), .en(1'b1), .Din(Din[3]), .Dout(Dout
[3]));
  Register_10bit reg_sp(.clk(clk), .reset(reset), .en(1'b1), .Din(Din[4]), .Dout(Dout
[4]));
  Register_10bit reg_a0(.clk(clk), .reset(reset), .en(1'b1), .Din(Din[5]), .Dout(Dout
[5]));
  Register_10bit reg_v0(.clk(clk), .reset(reset), .en(1'b1), .Din(Din[6]), .Dout(Dout
[6]));
  Register_10bit reg_ra(.clk(clk), .reset(reset), .en(1'b1), .Din(Din[7]), .Dout(Dout
[7]));

  //run a test on the registers
  integer i;
  always @(posedge clk) begin
      for( i = 0; i < 8; i = i + 1) begin
          Din[i] = ~Dout[i];
          #5;
      end
  end

  initial begin
      clk = 0;
      reset = 0;

      //initialize all registers with zeros
      { Din[0], Din[1], Din[2], Din[3], Din[4], Din[5], Din[6], Din[7] } = { 10'b0,
10'b0, 10'b0, 10'b0, 10'b0, 10'b0, 10'b0, 10'b0 };
  end

endmodule


module Register_10bit(
    input wire clk,
    input wire reset,
    input wire en,
    input wire [9:0] Din,
```

```
    output reg [9:0] Dout
    );

    always @(posedge clk) begin
        if (reset == 1'b1) begin
            Dout <= 10'b0000000000;
        end
        else if (en == 1'b1) begin
            Dout <= Din;
        end
    end

endmodule
```

```
`timescale 1ns / 1ns
`include "ALU.v"
`include "Registers.v"

//////////////////////////////////////////////////////////////////////////////////
// Company: Western Michigan University
// Engineer: Tyler Thompson
//
// Create Date: 02/13/2018 02:47:56 PM
// Design Name:
// Module Name: FetchUnit
// Project Name: ECE 3570 Lab 2
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies: ALU.v, Registers.v
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////



module FetchUnitTest();
    reg clk, reset;
    reg [1:0] pc_control;
    reg [4:0] jump_address;
    reg [2:0] branch_address;
    reg [9:0] reg_address;
    wire [9:0] PC;
    //toggle clock every 5ns
    always #5 clk = ~clk;

    FetchUnit fu0( .clk(clk), .reset(reset), .pc_control(pc_control), .jump_address
(jump_address), .branch_address(branch_address), .reg_address(reg_address), .PC_out
(PC) );

    initial begin
        clk = 0;
        reset = 1; //initialize the program counter
        #20
        reset = 0;
        #15

        pc_control = 2'b00; // increment program counter by 1
        #20
        pc_control = 2'b01; jump_address = 5'b00100; // increment program count by 4
        #20
        pc_control = 2'b10; reg_address = 10'b0001000000; // set program counter to 64
        #20
        pc_control = 2'b11; branch_address = 3'b010; // branch to address
        #20
        pc_control = 2'b00;
    end
endmodule


// Program Counter control unit
// pc_control:  00 -> increment program count by 1
//              01 -> add jump_address to program counter
//              10 -> set program counter to reg_address
```

```
//              11 -> add 4 to program count and then add the branch_address
module FetchUnit(
    input wire clk,
    input wire reset,
    input wire [1:0] pc_control,
    input wire [4:0] jump_address,
    input wire [2:0] branch_address,
    input wire [9:0] reg_address,
    output wire [9:0] PC_out
    );
    reg [9:0]PC_in;
    reg [9:0] b;
    wire c_out;
    wire [9:0] sum;
    wire [9:0] branch_offset;
    reg [9:0] branch_ext;

    //Create 10 bit register for program counter
    Register_10bit pc_reg( .clk(clk), .reset(reset), .en(1'b1), .Din(PC_in), .Dout
(PC_out) );

    //adder for calculating branch offset
    FullAdder_10Bit fa1( .a( branch_ext ), .b(10'b0000000100), .sum
(branch_offset), .c_out(c_out) );

    //adder for calculating pc address
    FullAdder_10Bit fa0( .a(PC_out), .b(b), .sum(sum), .c_out(c_out) );

    //update the program counter on each clock cycle
    always @(*) begin
            //zero extend branch address
            branch_ext <= { {3{branch_address[2]}}, branch_address[2:0] };
            PC_in <= sum;
            case(pc_control)
                2'b00: b <= 10'b0000000001;
                2'b01: b <= jump_address;
                2'b10: PC_in <= reg_address;
                2'b11: b <= branch_offset;
                default: PC_in <= PC_out;
            endcase

    end

endmodule
```

```
`timescale 1ns / 1ns
//////////////////////////////////////////////////////////////////////////////////
// Company: Western Michigan University
// Engineer: Tyler Thompson
//
// Create Date: 02/13/2018 02:44:18 PM
// Design Name:
// Module Name: ALU
// Project Name: ECE 3570 Lab 2
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module ALUTest();

    wire [9:0] result;
    wire c_out;

    reg [9:0] a;
    reg [9:0] b;
    reg [1:0] ALU_op;

    ALU alu0( .ALU_op(ALU_op), .a(a), .b(b), .f(result), .c_out(c_out) );

    initial begin
        ALU_op = 2'b00;
        a = 10'b0000000000;
        b = 10'b0000000000;
    end

    always @(a or b) begin
        //adder test
         a = 10'b0000000001; b = 10'b0000000001; ALU_op = 2'b00; // 1 + 1
         #10
         a = 10'b0000000011; b = 10'b0000000001; ALU_op = 2'b00; // 3 + 1
         #10
         a = 10'b0000000010; b = 10'b0000000001; ALU_op = 2'b00; // 2 + 1
         #10
         a = 10'b0010110010; b = 10'b0011001101; ALU_op = 2'b00; // 178 + 205 = 383
         #10
         a = 10'b1101001110; b = 10'b0011001101; ALU_op = 2'b00; // -178 + 205 = 27
         #10
         a = 10'b1101001110; b = 10'b1111111110; ALU_op = 2'b00; // -178 + -2 = -180
         #10;

        //shifter test
        a = 10'b0001000000; b = 10'b1111111111; ALU_op = 2'b10; //shift a right by 1
        #10
        a = 10'b0000000001; b = 10'b0000000001; ALU_op = 2'b10; //shift a left by 1
        #10
        a = 10'b0001000000; b = 10'b1111111100; ALU_op = 2'b10; //shift a right by 4
        #10
        a = 10'b0000000001; b = 10'b0000000100; ALU_op = 2'b10; //shift a left by 4
        #10;
```

```
        //comparator test
        a = 10'b0000000010; b = 10'b0000000001; ALU_op = 2'b01; //test if 2 is
greater than 1
        #10
        a = 10'b0000000001; b = 10'b0000000010; ALU_op = 2'b01; //test if 1 is
greater than 2
        #10
        a = 10'b0000000001; b = 10'b0000000001; ALU_op = 2'b01; //test if 1 is
greater than 1
        #10
        a = 10'b1111111111; b = 10'b0000000001; ALU_op = 2'b01; //test if -1 is
greater than 1
        #10
        a = 10'b0000000001; b = 10'b1111111111; ALU_op = 2'b01; //test if 1 is
greater than -1
        #10
        a = 10'b1111111110; b = 10'b1111111111; ALU_op = 2'b01; //test if -2 is
greater than -1
        #10
        a = 10'b1111111111; b = 10'b1111111110; ALU_op = 2'b01; //test if -1 is
greater than -2
        #10
        a = 10'b1111110000; b = 10'b1111111100; ALU_op = 2'b01; //test if -16 is
greater than -4
        #10
        a = 10'b1111111000; b = 10'b1111101100; ALU_op = 2'b01; //test if -8 is
greater than -20
        #10
        a = 10'b1000000000; b = 10'b1000000010; ALU_op = 2'b01; //test if -512 is
greater than -510
        #10
        a = 10'b1000000010; b = 10'b1000000000; ALU_op = 2'b01; //test if -510 is
greater than -512
        #10
        a = 10'b0111111111; b = 10'b0111111110; ALU_op = 2'b01; //test if 511 is
greater than 510
    end

endmodule


// Arithmetic Logic Unit
// Supports add, shift, and compare operations
// ALU_op: 00 -> adder
//         01 -> comparator
//         10 -> shifter
module ALU(
    input wire [1:0] ALU_op,
    input wire [9:0] a,
    input wire [9:0] b,
    output wire [9:0] f,
    output c_out
    );
    wire [9:0] add_result;
    wire [9:0] shift_result;
    wire [9:0] comp_result;

    FullAdder_10Bit fa0( .a(a), .b(b), .sum(add_result), .c_out(c_out) );
    Comparator cp0( .a(a), .b(b), .out(comp_result) );
    Shifter sh0( .a(a), .shift_amount(b), .out(shift_result) );
    ALU_mux mx0( .ALU_op(ALU_op), .add_result(add_result), .comp_result
(comp_result), .shift_result(shift_result), .final_answer(f) );

endmodule
```

```
// Multiplexer for ALU output selection
// ALU_op: 00 -> adder
//         01 -> comparator
//         10 -> shifter
module ALU_mux(
     input wire [1:0] ALU_op,
         input wire [9:0] add_result,
         input wire [9:0] comp_result,
         input wire [9:0] shift_result,
         output reg [9:0] final_answer
         );

         always@(*) begin
                case(ALU_op)
                         2'b00: final_answer <= add_result;
                         2'b01: final_answer <= comp_result;
                 2'b10: final_answer <= shift_result;
                 default: final_answer <= 10'b0;
                endcase
        end
endmodule


// Compares a to b
// out = 1 if a is grater than b and 0 if a is less than or equal to b
module Comparator(
    input wire [9:0] a,
    input wire [9:0] b,
    output reg [9:0] out //1 -> a is greater than b, 0 -> a is less than or equal to b
    );

    reg [9:0]temp0;
    wire [9:0] temp1;
    wire c_out;
    wire [9:0]sum;
    wire c_out1;

    //adder for twos complement
    FullAdder_10Bit fa1( .a(10'b0000000001), .b(temp0), .sum(temp1), .c_out(c_out) );

    FullAdder_10Bit fa2( .a(b), .b(temp1), .sum(sum), .c_out(c_out1) );
    always@(*) begin
        //both a and b are negative
        if ( (a[9] & b[9]) == 1 ) begin
           //twos complement of a
           temp0 <= ~a;

           if (c_out1 == 1) begin
                out = 0;
           end
           else begin
                out = 1;
           end
        end
        //a is negative and b is positive
        else if ( a[9] == 1 ) begin
            out = 0;
        end
        //a is positive and b is negative
        else if ( b[9] == 1 ) begin
            out = 1;
        end
```

```
        //a is positive and b is positive
        else begin
            //twos completment of a
            temp0 <= ~a;

            if ( sum[9] == 1 ) begin
                out = 1;
            end
            else begin
                out = 0;
            end
         end

    end
endmodule


// Shift a number left or right
// a positive shift_amount value will shift the number left
// a negative shift_amount value will shift the number right
module Shifter(
    input [9:0] a,
    input [9:0] shift_amount,
    output reg [9:0] out
    );
    reg [9:0] x;
    wire [9:0] sum;
    wire c_out;
    FullAdder_10Bit fa2( .a(x), .b(10'b0000000001), .sum(sum), .c_out(c_out) );
   always @(*) begin
        //if shift_amount is negative, take twos complement and shift right
        if ( shift_amount[9] == 1'b1 ) begin
           x = ~shift_amount ;
            out = a >> sum;
        end
        else begin
            out = a << shift_amount;
        end
    end
endmodule

// Ripple adder for 10 bit numbers
module FullAdder_10Bit(
    input [9:0] a,
    input [9:0] b,
    output [9:0] sum,
    output c_out
    );
        wire [8:0]ripple;

        FullAdder_1Bit f0( .a( a[0] ), .b( b[0] ), .c_in( 1'b0 ), .s( sum[0]), .c_out
( ripple[0] ) );
        FullAdder_1Bit f1( .a( a[1] ), .b( b[1] ), .c_in( ripple[0] ), .s( sum
[1]), .c_out(ripple[1]) );
        FullAdder_1Bit f2( .a( a[2] ), .b( b[2] ), .c_in( ripple[1] ), .s( sum
[2]), .c_out(ripple[2]) );
        FullAdder_1Bit f3( .a( a[3] ), .b( b[3] ), .c_in( ripple[2] ), .s( sum
[3]), .c_out(ripple[3]) );
        FullAdder_1Bit f4( .a( a[4] ), .b( b[4] ), .c_in( ripple[3] ), .s( sum
[4]), .c_out(ripple[4]) );
        FullAdder_1Bit f5( .a( a[5] ), .b( b[5] ), .c_in( ripple[4] ), .s( sum
[5]), .c_out(ripple[5]) );
        FullAdder_1Bit f6( .a( a[6] ), .b( b[6] ), .c_in( ripple[5] ), .s( sum
[6]), .c_out(ripple[6]) );
```

```
        FullAdder_1Bit f7( .a( a[7] ), .b( b[7] ), .c_in( ripple[6] ), .s( sum
[7]), .c_out(ripple[7]) );
        FullAdder_1Bit f8( .a( a[8] ), .b( b[8] ), .c_in( ripple[7] ), .s( sum
[8]), .c_out(ripple[8]) );
        FullAdder_1Bit f9( .a( a[9] ), .b( b[9] ), .c_in( ripple[8] ), .s( sum
[9]), .c_out(c_out) );

    endmodule

//simple adder for adding two bits
module FullAdder_1Bit(
    input wire a,
    input wire b,
    input wire c_in, //carry in
        output wire s, //sum
        output wire c_out //carry out
    );

        assign s = a ^ b ^ c_in;
        assign c_out = ((a ^ b) & c_in) | (a & b);

endmodule
```