# FPGA 系統設計實務

# FPGA System Design

**Final-term Project**

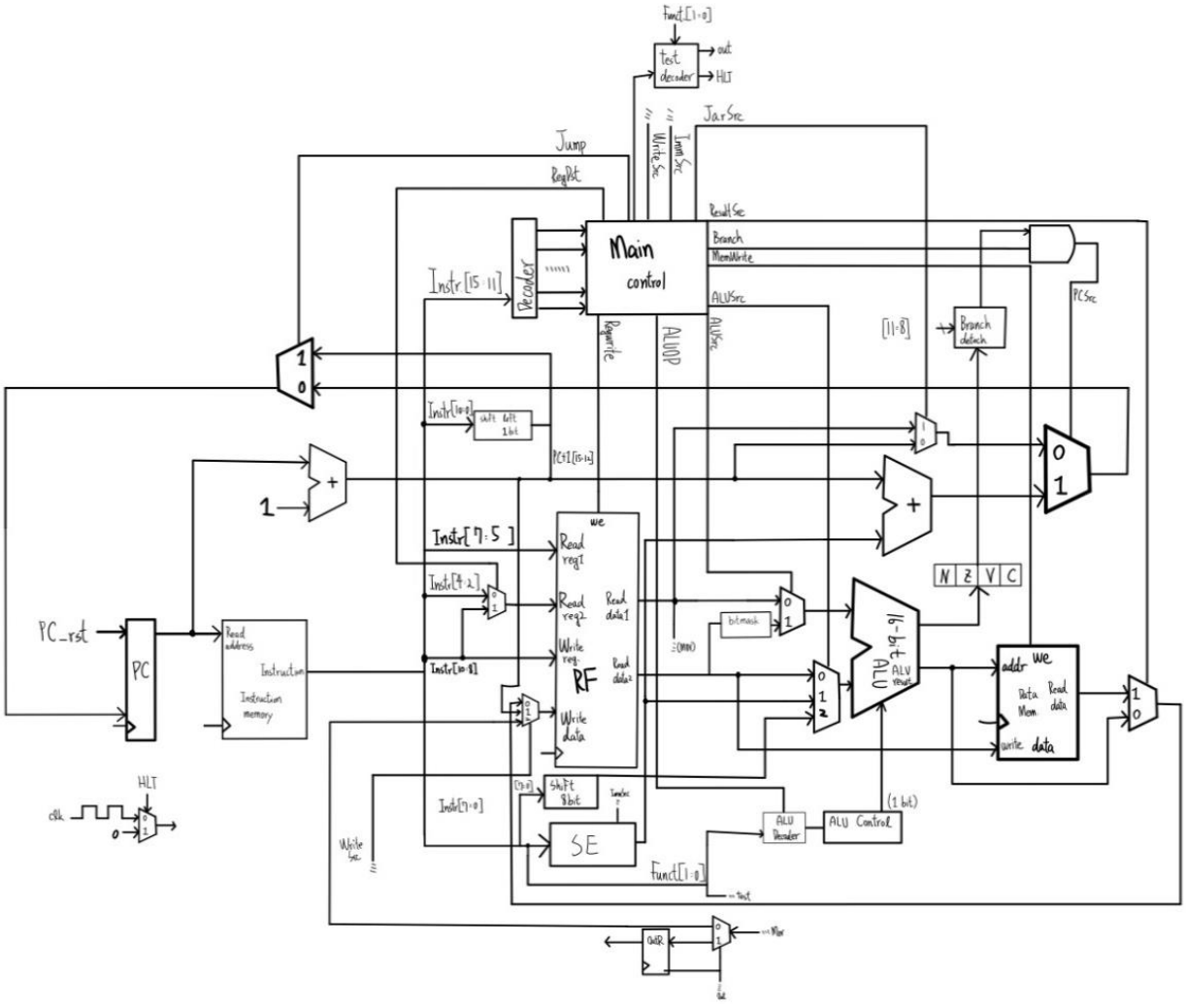
**學號:M11215075**

**姓名:胡劭**

# 1. Design:

我選擇的是 Single-cycle architecture，以下是我的設計圖，以及設計步驟說明:

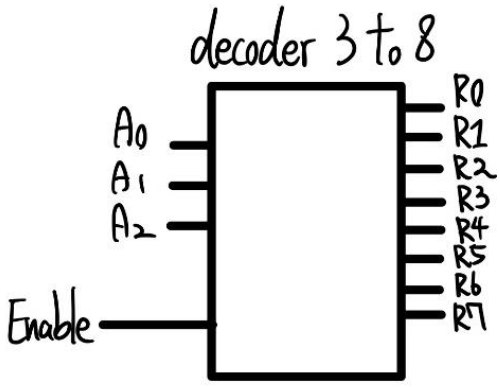
圖 1:完整設計圖



其他模組設計圖與做法:

- **A 16-Bit Eight-Register Register File**

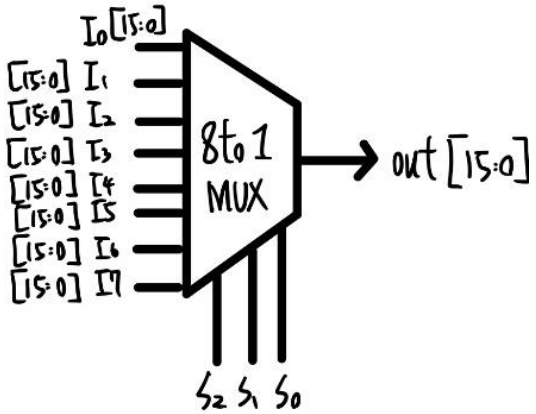
1. enabled-controlled 3-to-8 noninverting output decoder




作法:用 3 個信號線與 3 個 not gate 以及 8 顆 4and gate，其中用一條線接到每一顆 and gate 作為 enableed-controlled 作為輸出。

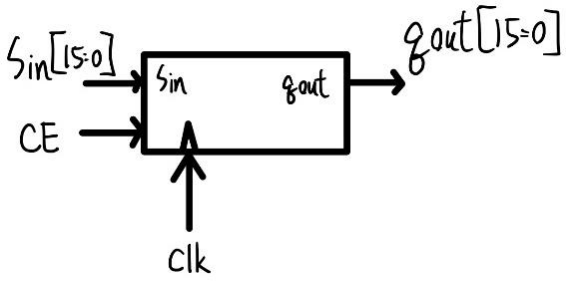
2. 16-bit 8-to-1 multiplexer

作法:把 16 顆 8 對 1 多工器疊加裡用 bus 來整合輸入輸出線



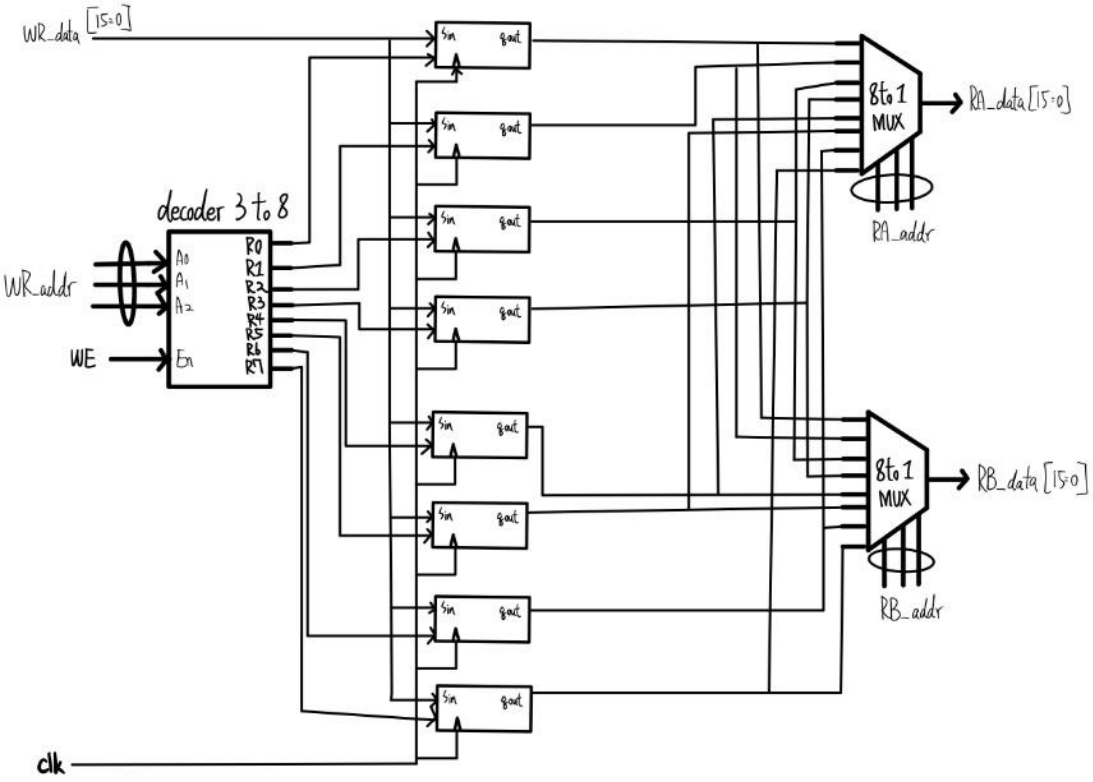
3. a 16-bit D-flip-flop register with clock-enable

作法:把 16 顆 D-flip-flop 用 bus 串接輸入與輸出，以及同步 clk 與 CE 線來控制 clock-enable




4. register file

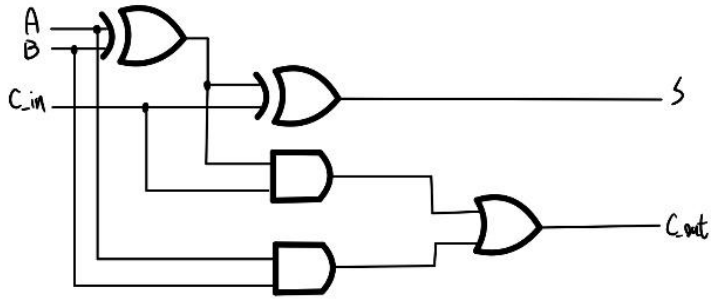
作法:把 3-to-8 與 8 顆 reg 做串接，八顆 reg 再分別跟兩個 8 對 1 多工器做串接，再去串接其他信號線，來做出一個 RF



- **A 16-Bit ALU**

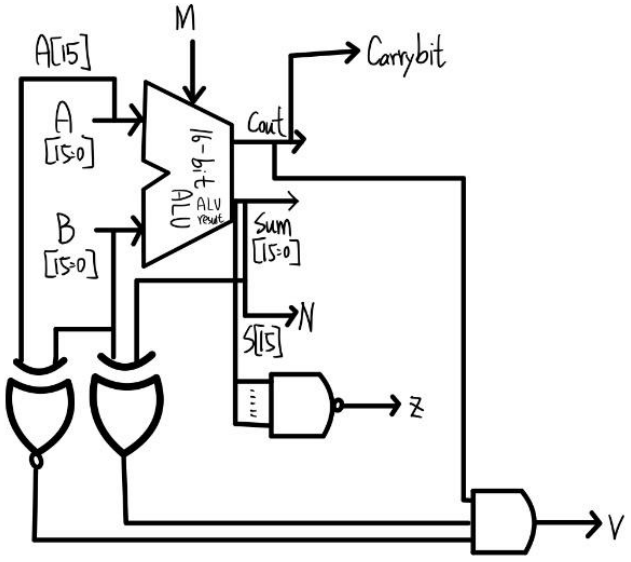
1. full adder

作法:用兩個 XOR gate 與兩個 and gate 與一個 or gate 來做



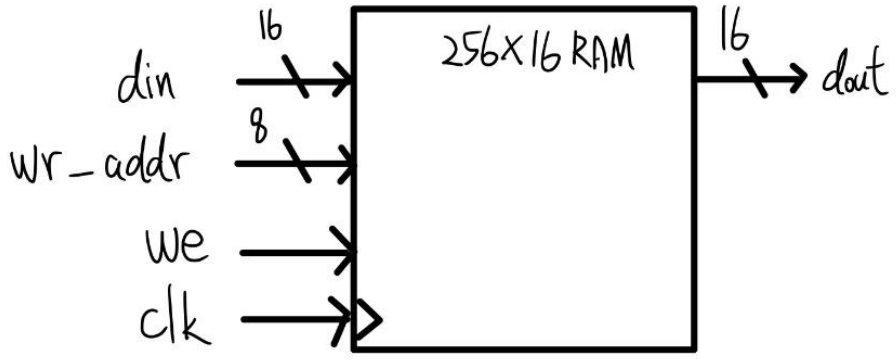

2. ALU

作法:把 fulladder 擴充為 16bit 後再利用 XOR 做成 2 補數加法器，之後要做 flag，cout 就是 flag C，sum 的 MSB 為 flag N，sum0~15bit 做 nand 為 flag Z，A 的 MSB 與 B 的 MSB 做 XNOR，B 的 MSB 與 Sum 的 MSB 做 XOR，最後再與 cout 一起做 and 為 flag V



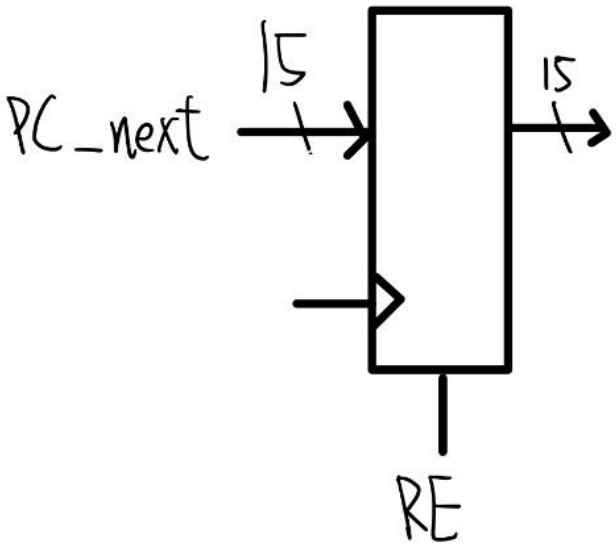
- **A 256×16 Memory Module**

  作法:只接把 address input 做出來方便跟外部的 3-to-8 decoder 做連接



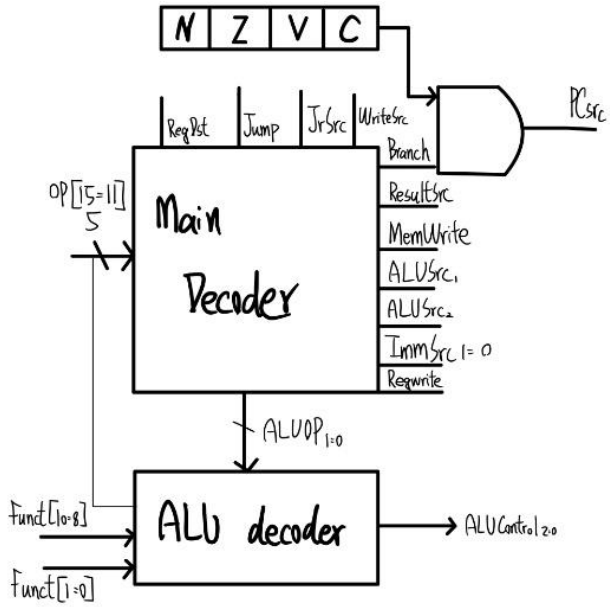

- **PC Circuitry**

  作法:用 D-flip-flop 作為 counter，每個 D-flip-flop 要有 reset，把 16 顆疊加之後就可以做成 16bit 的 counter



- **Instruction Decoder**

  作法:由 Main decoder 與 ALU decoder 一起組成




- **Complete Controller**

  以下是控制信號表:

| opcode | instr. | Funct. | RegDst | ALUSrc1 | ALUSrc2 | ResultSrc | MemWrit | RegWrite | Branch | ALUOp | WriteSrc | ImmSrc | Jump | JarSrc | Test |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00000 | ADD | 00 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 00000 | ADC | 01 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 00000 | SUB | 10 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 00000 | SBB | 11 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 00001 | LHI | xx | 0 | 1 | 10 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 00010 | LLI | xx | 0 | 0 | 01 | 0 | 0 | 1 | 0 | x | 10 | 1 | 0 | 0 | 0 |
| 00011 | LDR | xx | 0 | 0 | 01 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 00101 | STR | xx | 1 | 0 | 01 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 00110 | CMP | 01 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 00111 | ADDI | xx | 0 | 0 | 01 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 01000 | SUBI | xx | 0 | 0 | 01 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 01011 | MOV | xx | 0 | 0 | x | 0 | 0 | 1 | 0 | x | 10 | 0 | 0 | 0 | 0 |
| 10000 | JMP | xx | 0 | 0 | x | 0 | 0 | 0 | 0 | x | 0 | 0 | 1 | 0 | 0 |
| 10001 | JAL | xx | 0 | 0 | x | 0 | 0 | 1 | 0 | x | 1 | 1 | 0 | 1 | 0 |
| 10010 | JAL | xx | 0 | 0 | x | 0 | 0 | 1 | 0 | x | 1 | 0 | 0 | 0 | 0 |
| 10011 | JR | xx | 0 | 0 | x | 0 | 0 | 0 | 0 | x | 0 | 0 | 0 | 1 | 0 |
| 11000 | BCC | 011 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | x | 0 | 1 | 0 | 0 | 0 |
| 11000 | BCS | 010 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | x | 0 | 1 | 0 | 0 | 0 |
| 11000 | BNE | 001 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | x | 0 | 1 | 0 | 0 | 0 |
| 11000 | BEQ | 000 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | x | 0 | 1 | 0 | 0 | 0 |
| 11001 | B[AL] | 110 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | x | 0 | 1 | 0 | 0 | 0 |
| 11100 | OutR | 00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | x | 0 | 0 | 0 | 0 | 1 |
| 11100 | HLT | 01 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | x | 0 | 0 | 0 | 0 | 1 |

# 2. HDL Entry:

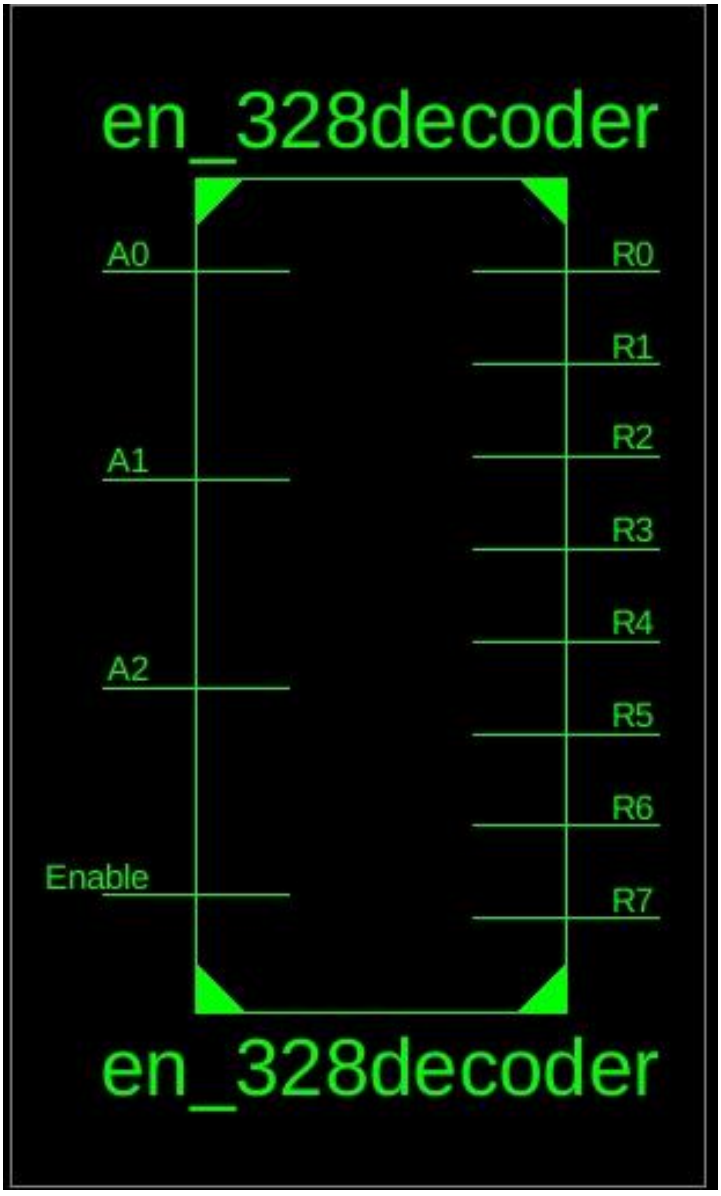
- **A 16-Bit Eight-Register Register File**

  1. enabled-controlled 3-to-8 noninverting output decoder

  Verilog HDL:

```
module en_328decoder(
    input A0,
    input A1,
    input A2,
    input Enable,
    output reg R0,
    output reg R1,
    output reg R2,
    output reg R3,
    output reg R4,
    output reg R5,
    output reg R6,
    output reg R7
    );
always @ (*) begin
    R0 = 0;
    R1 = 0;
    R2 = 0;
    R3 = 0;
    R4 = 0;
    R5 = 0;
    R6 = 0;
    R7 = 0;
    if (Enable == 1'b1 )
    begin
        case ({A2, A1, A0})
            3'b000: R0 = 1'b1;
            3'b001: R1 = 1'b1;
            3'b010: R2 = 1'b1;
            3'b011: R3 = 1'b1;
            3'b100: R4 = 1'b1;
            3'b101: R5 = 1'b1;
            3'b110: R6 = 1'b1;
            3'b111: R7 = 1'b1;
            default: begin

                R0 = 1'b0;
                R1 = 1'b0;
                R2 = 1'b0;
                R3 = 1'b0;
                R4 = 1'b0;
                R5 = 1'b0;
                R6 = 1'b0;
                R7 = 1'b0;
            end
        endcase
    end
end
endmodule
```

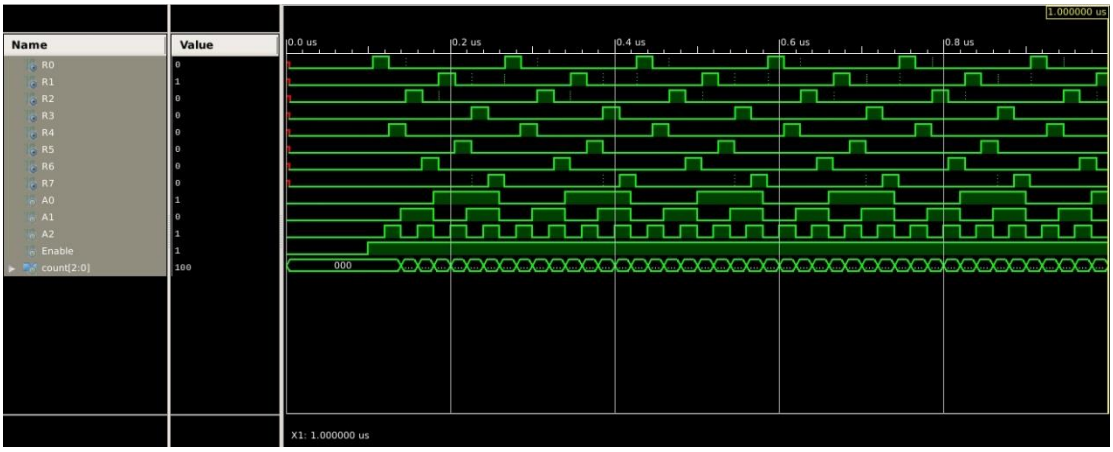
RTL


en_328decoder
A0
A1
A2
Enable
R0
R1
R2
R3
R4
R5
R6
R7
en_328decoder

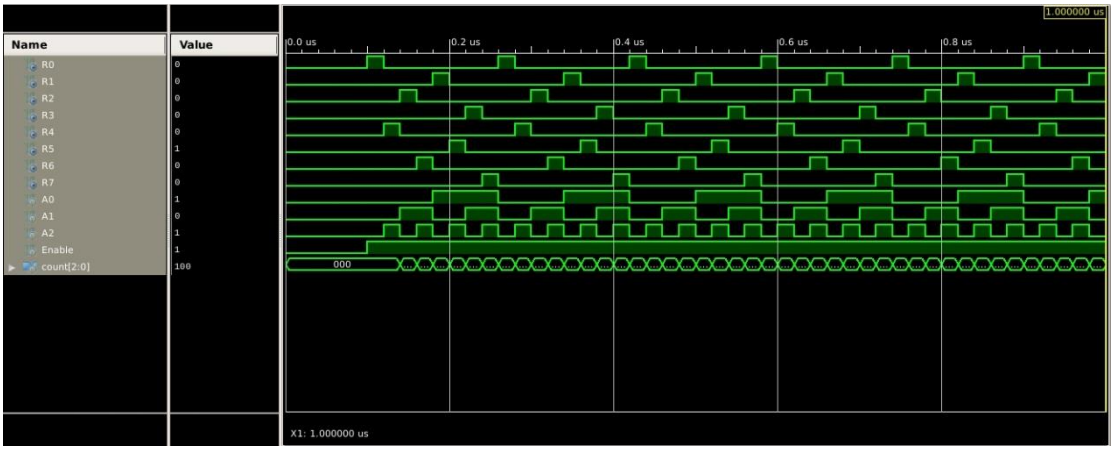
Testbench

```
module en_328decoder_tb;

    // Inputs
    reg A0;
    reg A1;
    reg A2;
    reg Enable;

    // Outputs
    wire R0;
    wire R1;
    wire R2;
    wire R3;
    wire R4;
    wire R5;
    wire R6;
    wire R7;

    reg [2:0] count = 3'd0;
    // Instantiate the Unit Under Test (UUT)
    en_328decoder uut (
        .A0(A0),
        .A1(A1),
        .A2(A2),
        .Enable(Enable),
        .R0(R0),
        .R1(R1),
        .R2(R2),
        .R3(R3),
        .R4(R4),
        .R5(R5),
        .R6(R6),
        .R7(R7)
    );
// Initialize Inputs
     initial begin
      A2 = 1'b0;
      A1 = 1'b0;
      A0 = 1'b0;
      Enable = 1'b0;
// Wait 100 ns for global reset to finish

      #100;

      Enable = 1'b1;

      #20;
      for (count = 0; count < 8; count = count + 1'b1)
          begin

              {A0,A1,A2} = {A0,A1,A2} + 1'b1;
              #20;
          end
      Enable = 1'b0;
     end
endmodule
```

## Behavioral




## Post-Route

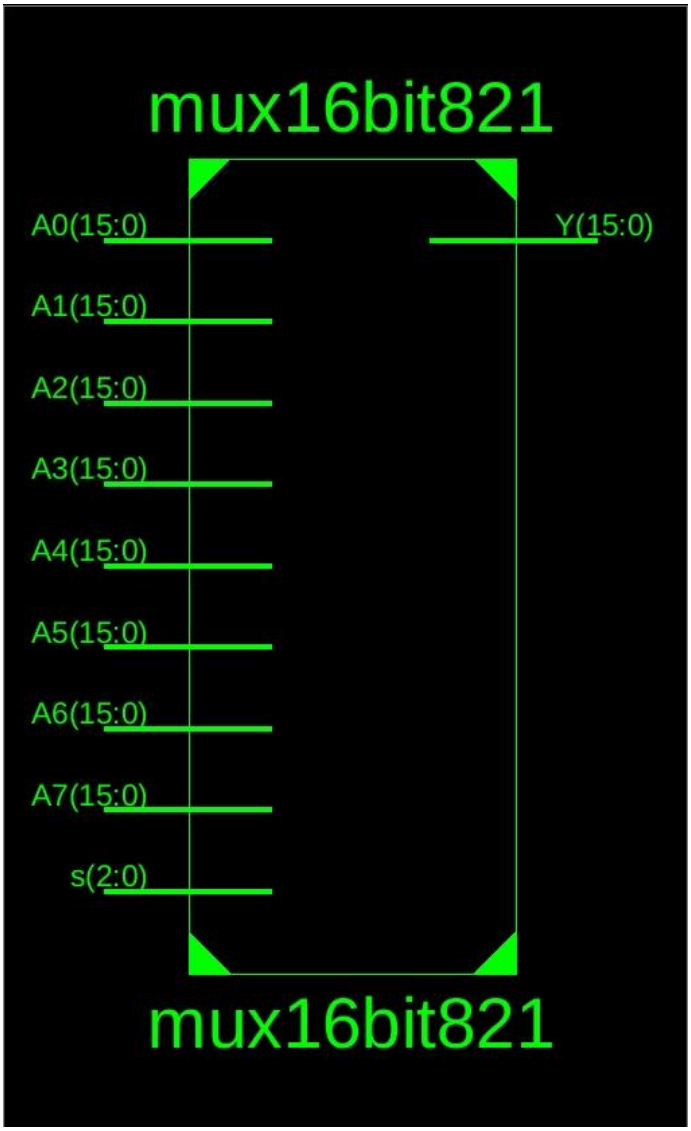


2. 16bit-8-to-1 multiplexer

Verilog HDL:

```
module mux16bit821(
    input [15:0] A7,
    input [15:0] A6,
    input [15:0] A5,
    input [15:0] A4,
    input [15:0] A3,
    input [15:0] A2,
    input [15:0] A1,
    input [15:0] A0,
    input [2:0] s,
    output reg [15:0] Y
    );
always @ (*) begin
    case(s)
        3'b000: Y = A0;
        3'b001: Y = A1;
        3'b010: Y = A2;
        3'b011: Y = A3;
        3'b100: Y = A4;
        3'b101: Y = A5;
        3'b110: Y = A6;
        3'b111: Y = A7;
        default: Y = 16'bx;
    endcase
end

endmodule
```

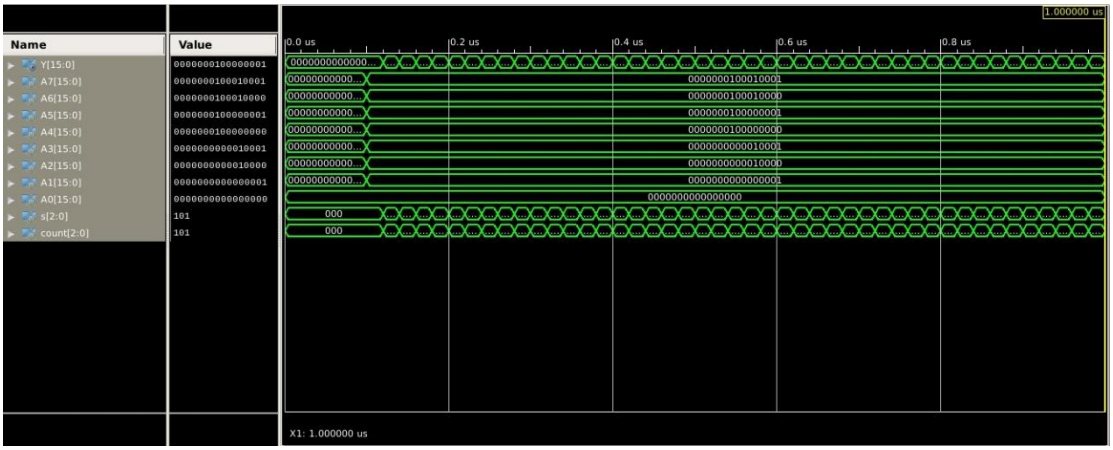
RTL


mux16bit821
A0(15:0)
A1(15:0)
A2(15:0)
A3(15:0)
A4(15:0)
A5(15:0)
A6(15:0)
A7(15:0)
s(2:0)
Y(15:0)
mux16bit821

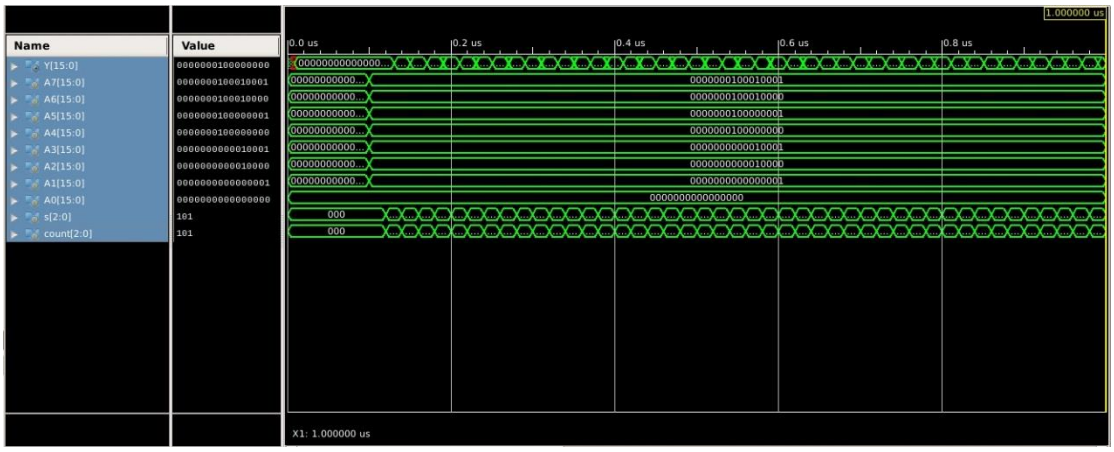
## Testbench

```
module mux16bit821_tb;

    // Inputs
    reg [15:0] A7;
    reg [15:0] A6;
    reg [15:0] A5;
    reg [15:0] A4;
    reg [15:0] A3;
    reg [15:0] A2;
    reg [15:0] A1;
    reg [15:0] A0;
    reg [2:0] s;

    // Outputs
    wire [15:0] Y;
    reg [2:0] count = 3'b000;
    // Instantiate the Unit Under Test (UUT)
    mux16bit821 uut (
        .A7(A7),
        .A6(A6),
        .A5(A5),
        .A4(A4),
        .A3(A3),
        .A2(A2),
        .A1(A1),
        .A0(A0),
        .s(s),
        .Y(Y)
    );

    initial begin
        // Initialize Inputs
        A7 = 0;
        A6 = 0;
        A5 = 0;
        A4 = 0;
        A3 = 0;
        A2 = 0;
        A1 = 0;
        A0 = 0;
        s = 0;

        // Wait 100 ns for global reset to finish
        #100;

        s = 3'b000;
        A0 = 16'h0000;
        A1 = 16'h0001;
        A2 = 16'h0010;
        A3 = 16'h0011;
        A4 = 16'h0100;
        A5 = 16'h0101;
        A6 = 16'h0110;
        A7 = 16'h0111;

        for (count = 0; count < 111; count = count + 1'b1) begin
            s = count;

            #20;
        end

    end

endmodule
```

## Behavioral




## Post-Route

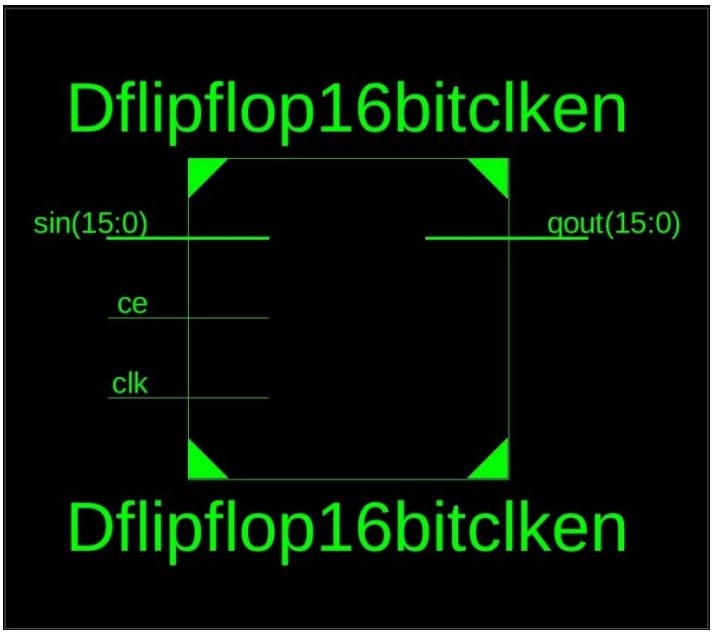


3. a 16-bit D-flip-flop register with clock-enable

Verilog HDL:

```
module Dflipflop16bitclken(
    input [15:0] sin,
    input ce,
    input clk,
    output reg [15:0] qout
    );
always @(posedge clk) begin
    if (ce == 1'b1) begin
        qout = 16'b0;
    end else begin
        qout = sin;
    end

end

endmodule
```

RTL:



Testbench

```
module Dflipflop16bitclken_tb;

    // Inputs
    reg [15:0] sin;
    reg ce;
    reg clk;

    // Outputs
    wire [15:0] qout;

    // Instantiate the Unit Under Test (UUT)
    Dflipflop16bitclken uut (
        .sin(sin),
        .ce(ce),
        .clk(clk),
        .qout(qout)
    );

    initial begin
      clk = 0;
      forever #10 clk = ~clk;
    end
    initial begin
      sin <= 16'h0000;
      #100
      sin <= 16'h0001;
      #100
      sin <= 16'h0000;
      #100
      sin <= 16'h0001;
    end

endmodule
```

## Behavioral

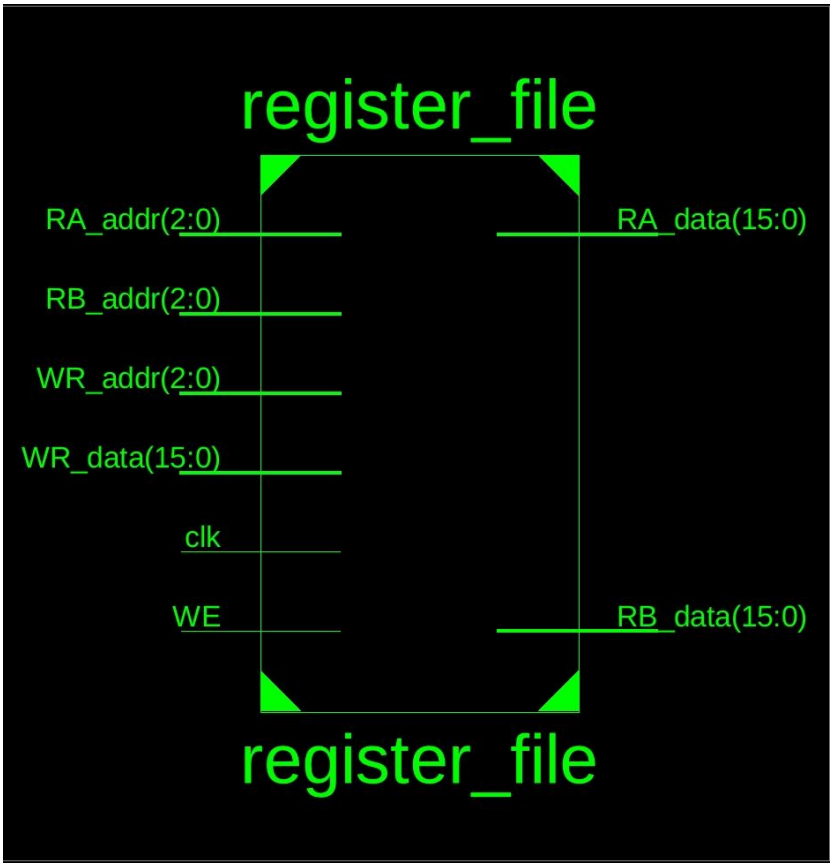

## Post-Route

## 4. register file

## Verilog HDL:

```
module register_file(
    input [15:0] WR_data,
    input [2:0] WR_addr,
    input WE,
    input clk,
    input [2:0] RA_addr,
    input [2:0] RB_addr,
    output [15:0] RA_data,
    output [15:0] RB_data
    );
    wire reg_sel[7:0]; //8 select signal of DFlipflop
    wire [15:0] reg_out[7:0]; //8 output of DFlipflop
    en_328decoder decoder_inst(
        .A0(WR_addr[0]),
        .A1(WR_addr[1]),
        .A2(WR_addr[2]),
        .Enable(WE),
        .R0(reg_sel[0]),
        .R1(reg_sel[1]),
        .R2(reg_sel[2]),
        .R3(reg_sel[3]),
        .R4(reg_sel[4]),
        .R5(reg_sel[5]),
        .R6(reg_sel[6]),
        .R7(reg_sel[7])
    );

    // 8 16bit register
    Dflipflop16bitclken reg0 (.sin(WR_data), .ce(reg_sel[0]), .clk(clk), .qout(reg_out[0]));
    Dflipflop16bitclken reg1 (.sin(WR_data), .ce(reg_sel[1]), .clk(clk), .qout(reg_out[1]));
    Dflipflop16bitclken reg2 (.sin(WR_data), .ce(reg_sel[2]), .clk(clk), .qout(reg_out[2]));
    Dflipflop16bitclken reg3 (.sin(WR_data), .ce(reg_sel[3]), .clk(clk), .qout(reg_out[3]));
    Dflipflop16bitclken reg4 (.sin(WR_data), .ce(reg_sel[4]), .clk(clk), .qout(reg_out[4]));
    Dflipflop16bitclken reg5 (.sin(WR_data), .ce(reg_sel[5]), .clk(clk), .qout(reg_out[5]));
    Dflipflop16bitclken reg6 (.sin(WR_data), .ce(reg_sel[6]), .clk(clk), .qout(reg_out[6]));
    Dflipflop16bitclken reg7 (.sin(WR_data), .ce(reg_sel[7]), .clk(clk), .qout(reg_out[7]));

    mux16bit821 mux1_inst(
      .A7(reg_out[7]),
      .A6(reg_out[6]),
      .A5(reg_out[5]),
      .A4(reg_out[4]),
      .A3(reg_out[3]),
      .A2(reg_out[2]),
      .A1(reg_out[1]),
      .A0(reg_out[0]),
      .s(RA_addr),
      .Y(RA_data)
    );
    mux16bit821 mux2_inst(
      .A7(reg_out[7]),
      .A6(reg_out[6]),
      .A5(reg_out[5]),
      .A4(reg_out[4]),
      .A3(reg_out[3]),
      .A2(reg_out[2]),
      .A1(reg_out[1]),
      .A0(reg_out[0]),
      .s(RB_addr),
      .Y(RB_data)
    );


endmodule
```

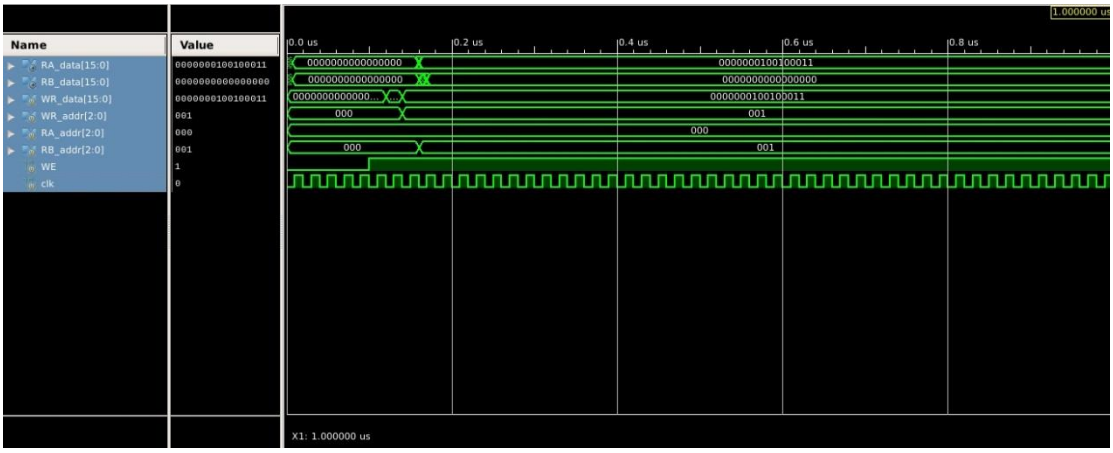
RTL


register_file
RA_addr(2:0)
RB_addr(2:0)
WR_addr(2:0)
WR_data(15:0)
clk
WE
RA_data(15:0)
RB_data(15:0)
register_file

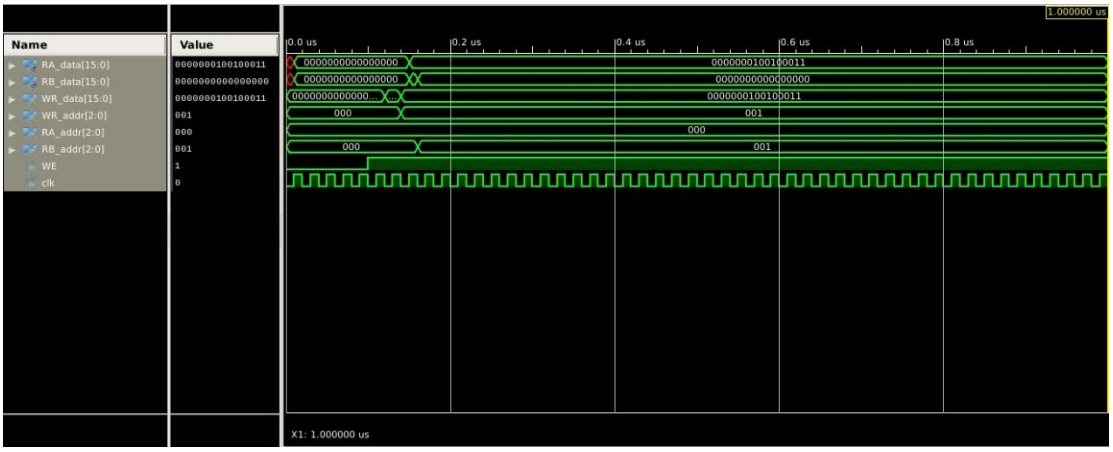
## Testbench

```
module register_file_tb();

    // Inputs
    reg [15:0] WR_data;
    reg [2:0] WR_addr;

    reg [2:0] RA_addr;
    reg [2:0] RB_addr;
    reg WE;
    reg clk;

    // Outputs
    wire [15:0] RA_data;
    wire [15:0] RB_data;

    // Instantiate the Unit Under Test (UUT)
    register_file uut (
        .WR_data(WR_data),
        .WR_addr(WR_addr),
        .WE(WE),
        .clk(clk),
        .RA_addr(RA_addr),
        .RB_addr(RB_addr),
        .RA_data(RA_data),
        .RB_data(RB_data)
    );

    initial begin
        // Initialize Inputs
        WR_data = 16'b0;
        WR_addr = 3'b000;
        WE = 1'b0;
        clk = 1'b0;
        RA_addr = 3'b000;
        RB_addr = 3'b000;

        // Wait 100 ns for global reset to finish
        #100;

        // Add stimulus here
        WE = 1'b1;
        #20;
        WR_data = 16'habcd;
        WR_addr = 3'b000;
        #20;
        WR_data = 16'h0123;
        WR_addr = 3'h001;
        #20;
        RA_addr = 3'h000;
        RB_addr = 3'h001;

    end
    always #10 clk = ~clk;

endmodule
```

## Behavioral




## Post-Route



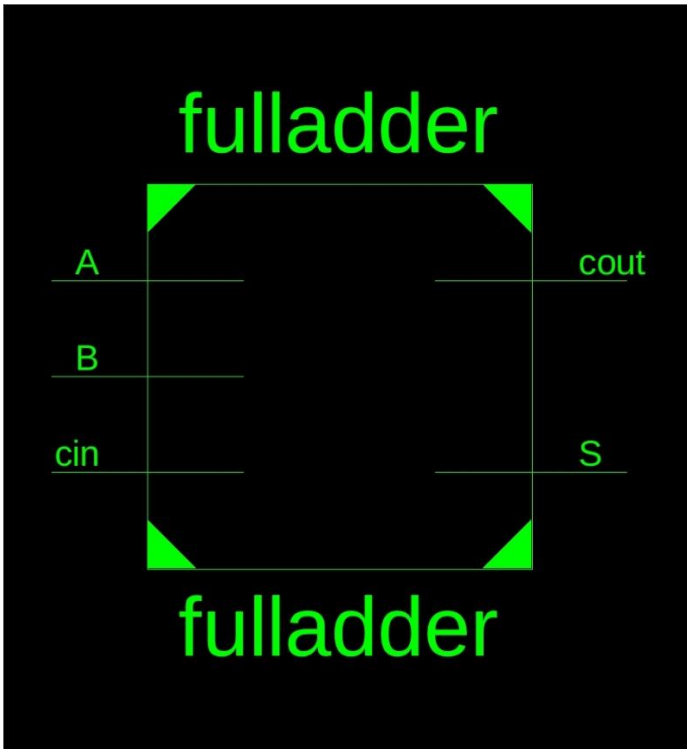
- **A 16-Bit ALU**

1. full adder

Verilog HDL:

```
module fulladder(
    input A,
    input B,
    input cin,
    output reg cout,
    output reg S
    );
always @(*) begin
    S = (A ^ B) ^ cin;
    cout = ((A ^ B) & cin) | (A & B);
end


endmodule
```

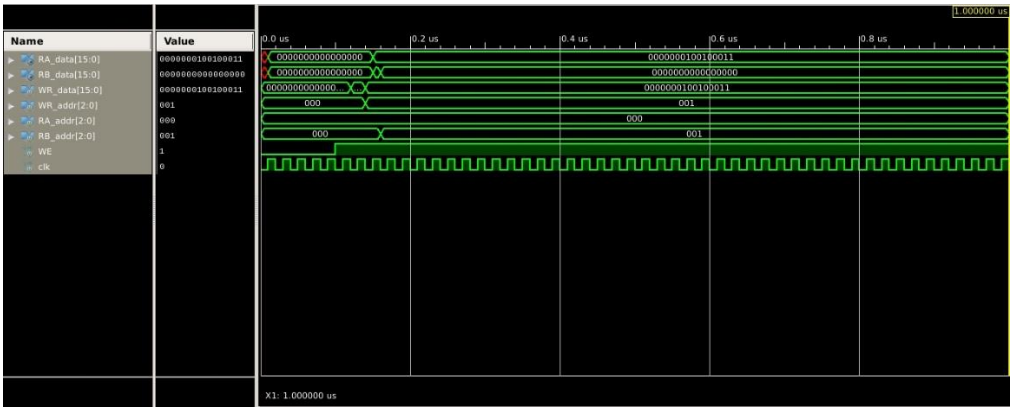
RTL



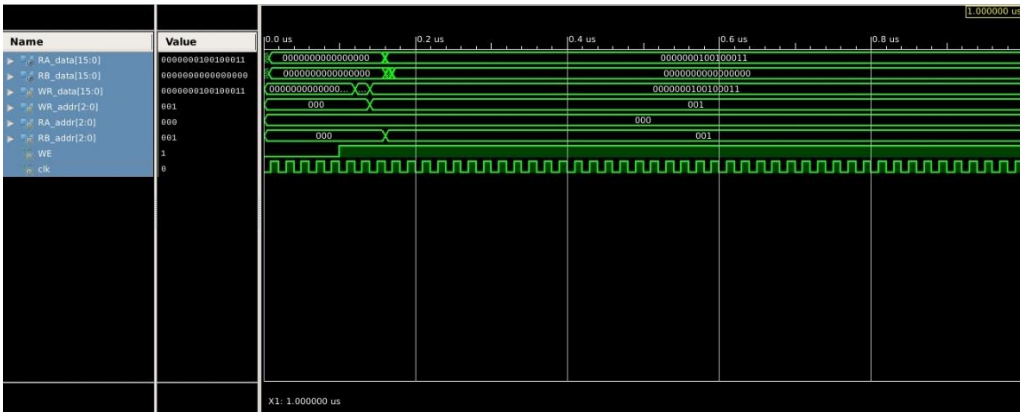
Testbench

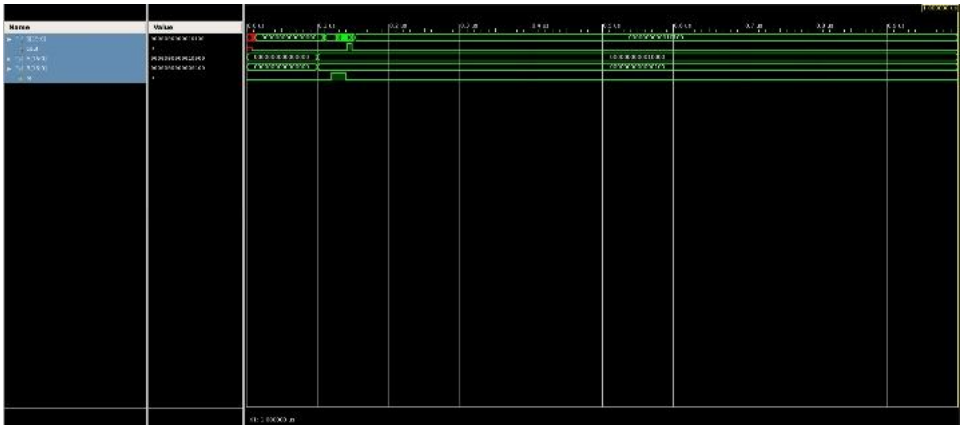
```
module fulladder_tb;

    // Inputs
    reg A;
    reg B;
    reg cin;

    // Outputs
    wire cout;
    wire S;
    reg [2:0] i = 3'd0;
    // Instantiate the Unit Under Test (UUT)
    fulladder uut (
        .A(A),
        .B(B),
        .cin(cin),
        .cout(cout),
        .S(S)
    );

    initial begin
        // Initialize Inputs
        A = 1'b0;
        B = 1'b0;
        cin = 1'b0;

        // Wait 100 ns for global reset to finish
        #100;
        for ( i = 0; i < 8; i = i + 1'b1)begin

             {A, B, cin} = {A, B, cin} + 1'b1;

        #20;

   end

        // Add stimulus here

    end

endmodule
```

## Behavioral




## Post-Route



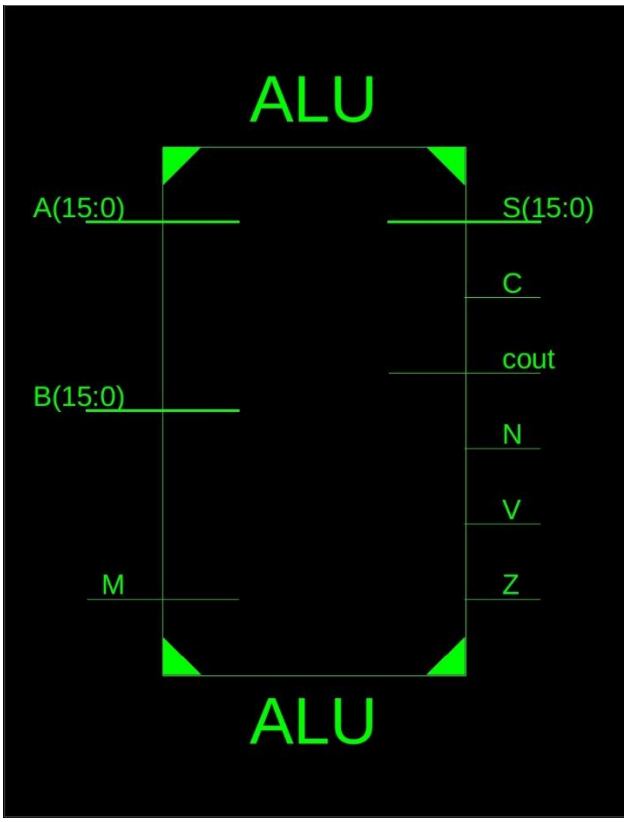



## 2. 16-bit ALU

## Verilog HDL:

```
module ALU(
    input M,
    input [15:0] A,
    input [15:0] B,

    output cout,
    output [15:0] S,
    output N,
    output Z,
    output C,
    output V
    );
    wire [14:0] cout_f; // internal carry declared as net
    wire [15:0] xb; // after xor

xor x1(xb[0],B[0],M);
xor x2(xb[1],B[1],M);
xor x3(xb[2],B[2],M);
xor x4(xb[3],B[3],M);
xor x5(xb[4],B[4],M);
xor x6(xb[5],B[5],M);
xor x7(xb[6],B[6],M);
xor x8(xb[7],B[7],M);
xor x9(xb[8],B[8],M);
xor x10(xb[9],B[9],M);
xor x11(xb[10],B[10],M);
xor x12(xb[11],B[11],M);
xor x13(xb[12],B[12],M);
xor x14(xb[13],B[13],M);
xor x15(xb[14],B[14],M);
xor x16(xb[15],B[15],M);

genvar i;
generate
    for (i = 0; i < 16; i = i + 1) begin : adders
        if (i == 0)
            fulladder adders(
            .A(A[i]),
            .B(xb[i]),
            .cin(M),
            .S(S[i]),
            .cout(cout_f[i])
            );
        else if (i == 15)
            fulladder adders(
            .A(A[i]),
            .B(xb[i]),
            .cin(cout_f[i-1]),
            .S(S[i]),
            .cout(cout)
            );
        else
            fulladder adders(
            .A(A[i]),
            .B(xb[i]),
            .cin(cout_f[i-1]),
            .S(S[i]),
            .cout(cout_f[i])
            );
    end
endgenerate

    flagN fN (.S15(S[15]));
    flagZ fZ (.S(S));
    flagC fC (.cout(cout));
    flagV fV (.a15(A[15]),.b15(B[15]),.cout(cout),.S15(S[15]));


endmodule

module flagN (
    input S15,
    output N
    );
    assign N = S15;
endmodule

module flagZ (
    input [15:0]S,
    output Z
    );
    assign Z = (~(S[15]&S[14]&S[13]&S[12]&S[11]&S[10]&S[9]&S[8]&S[7]&S[6]&S[5]&S[4]&S[3]&S[2]&S[1]&S[0]));
endmodule

module flagC (
    input cout,
    output C
    );
    assign C = cout;
endmodule

module flagV (
    input a15,
    input b15,
    input cout,
    input S15,
    output V
    );
    assign V = (b15 ^ a15) & (b15^ S15) & cout;
endmodule
```

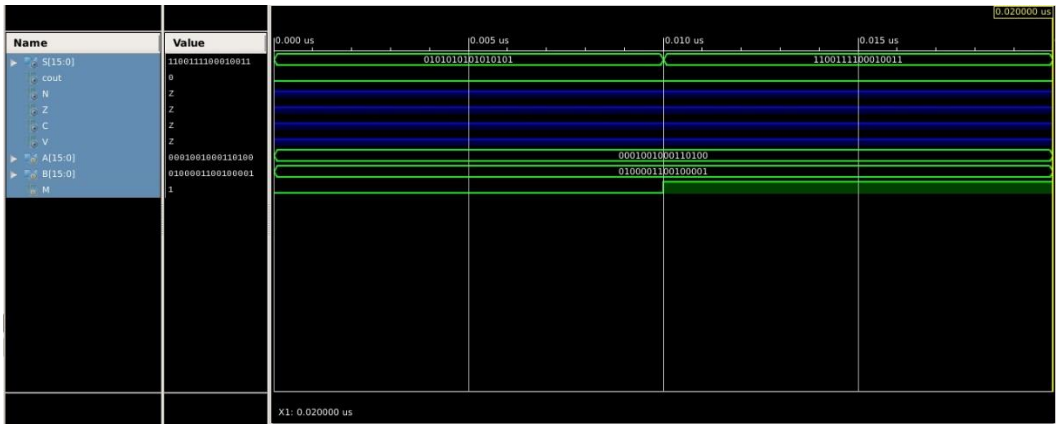
## RTL




## Testbench

```
module ALU_tb();

   // Inputs

   reg [15:0] A;
   reg [15:0] B;
   reg M;

   // Outputs
   wire [15:0] S;
   wire cout;

   wire N;
   wire Z;
   wire C;
   wire V;

   // Instantiate the Unit Under Test (UUT)
   ALU uut (

      .A(A),
      .B(B),
      .S(S),
      .cout(cout),
      .M(M),
      .N(N),
      .Z(Z),
      .C(C),
      .V(V)
   );

   initial begin
      A = 16'h1234;
      B = 16'h4321;
      M = 1'b0;  // Set the mode (0 for addition, 1 for subtraction)

      #10;
      M = 1'b1;
      #10;
      $finish;
   end
   initial begin
      $monitor("At time %d, Sum = %d, Zero = %b, Negative = %b, Carry = %b, Overflow = %b", $time, S, Z, N, C, V);
   end

endmodule
```

Behavioral




Post-Route


- **A 256×16 Memory Module**

Verilog HDL:

```
module Memory_module(
    input [15:0] data,
    input addr0, addr1, addr2, addr3, addr4, addr5, addr6, addr7,
    input WEn,
    input clk,
    output [15:0] qout
    );

    reg [15:0] mem [0:255]; //256 words of 16 bits

    always @(posedge clk) begin
        if(WEn) begin
            mem[{addr7, addr6, addr5, addr4, addr3, addr2, addr1, addr0}] <= data;
        end
    end

    assign qout = mem[{addr7, addr6, addr5, addr4, addr3, addr2, addr1, addr0}];

endmodule
```

RTL

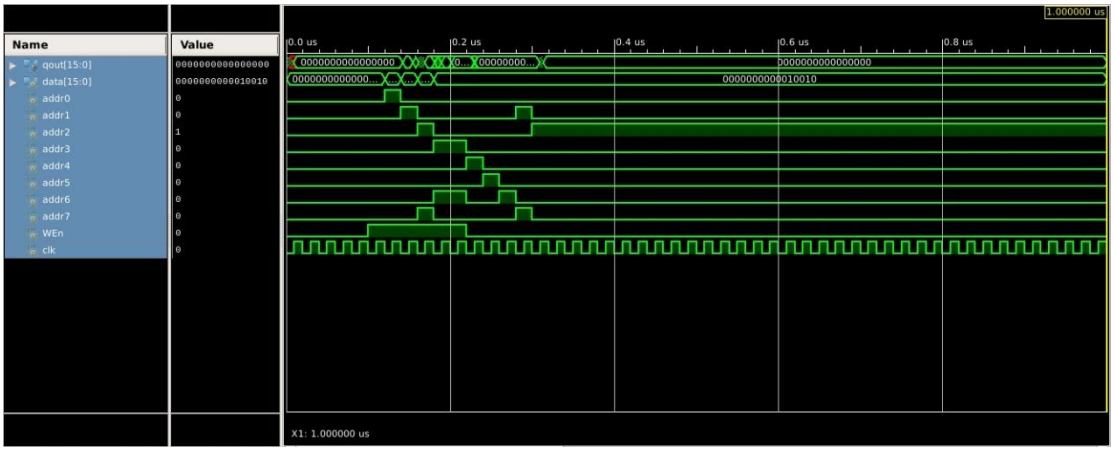

Memory_module
data(15:0)
addr0
addr1
addr2
addr3
addr4
addr5
addr6
addr7
clk
WEn
qout(15:0)
Memory_module

# Testbench

```
module Memory_module_tb;

	// Inputs
	reg [15:0] data;
	reg addr0;
	reg addr1;
	reg addr2;
	reg addr3;
	reg addr4;
	reg addr5;
	reg addr6;
	reg addr7;
	reg WEn;
	reg clk;

	// Outputs
	wire [15:0] qout;

	// Instantiate the Unit Under Test (UUT)
	Memory_module uut (
		.data(data),
		.qout(qout),
		.addr0(addr0),
		.addr1(addr1),
		.addr2(addr2),
		.addr3(addr3),
		.addr4(addr4),
		.addr5(addr5),
		.addr6(addr6),
		.addr7(addr7),
		.WEn(WEn),
		.clk(clk)
	);

	initial begin
		// Initialize Inputs
		data = 16'h0;
		addr0 = 1'b0;
		addr1 = 1'b0;
		addr2 = 1'b0;
		addr3 = 1'b0;
		addr4 = 1'b0;
		addr5 = 1'b0;
		addr6 = 1'b0;
		addr7 = 1'b0;
		WEn = 1'b0;
		clk = 1'b0;
		// Wait 100 ns for global reset to finish
		#100;

		// Add stimulus here

		data  = 16'h0;
		addr0 = 1'b0;
		addr1 = 1'b0;
		addr2 = 1'b0;
		addr3 = 1'b0;
		addr4 = 1'b0;
		addr5 = 1'b0;
		addr6 = 1'b0;
		addr7 = 1'b0;
		WEn  = 1'b1;
		#20;
		data  = 16'h1;
		addr0 = 1'b1;
		addr1 = 1'b0;
		addr2 = 1'b0;
		addr3 = 1'b0;
		addr4 = 1'b0;
		addr5 = 1'b0;
		addr6 = 1'b0;
		addr7 = 1'b0;
		#20;
		data  = 16'h10;
		addr0 = 1'b0;
		addr1 = 1'b1;
		addr2 = 1'b0;
		addr3 = 1'b0;
		addr4 = 1'b0;
		addr5 = 1'b0;
		addr6 = 1'b0;
		addr7 = 1'b0;
		#20;
		data  = 16'h6;
		addr0 = 1'b0;
		addr1 = 1'b0;
		addr2 = 1'b1;
		addr3 = 1'b0;
		addr4 = 1'b0;
		addr5 = 1'b0;
		addr6 = 1'b0;
		addr7 = 1'b1;
		#20;
		data  = 16'h12;
		addr0 = 1'b0;
		addr1 = 1'b0;
		addr2 = 1'b0;
		addr3 = 1'b1;
		addr4 = 1'b0;
		addr5 = 1'b0;
		addr6 = 1'b1;
		addr7 = 1'b0;
		#40;
		addr0 = 1'b0;
		addr1 = 1'b0;
		addr2 = 1'b0;
		addr3 = 1'b0;
		addr4 = 1'b1;
		addr5 = 1'b0;
		addr6 = 1'b0;
		addr7 = 1'b0;
		WEn  = 1'b0;
		#20;
		addr0 = 1'b0;
		addr1 = 1'b0;
		addr2 = 1'b0;
		addr3 = 1'b0;
		addr4 = 1'b0;
		addr5 = 1'b1;
		addr6 = 1'b0;
		addr7 = 1'b0;
		#20;
		addr0 = 1'b0;
		addr1 = 1'b0;
		addr2 = 1'b0;
		addr3 = 1'b0;
		addr4 = 1'b0;
		addr5 = 1'b0;
		addr6 = 1'b1;
		addr7 = 1'b0;
		#20;
		addr0 = 1'b0;
		addr1 = 1'b1;
		addr2 = 1'b0;
		addr3 = 1'b0;
		addr4 = 1'b0;
		addr5 = 1'b0;
		addr6 = 1'b0;
		addr7 = 1'b1;
		#20;
		addr0 = 1'b0;
		addr1 = 1'b0;
		addr2 = 1'b1;
		addr3 = 1'b0;
		addr4 = 1'b0;
		addr5 = 1'b0;
		addr6 = 1'b0;
		addr7 = 1'b0;
	end
	always #10 clk = ~clk;

endmodule
```

Behavioral


Post-Route

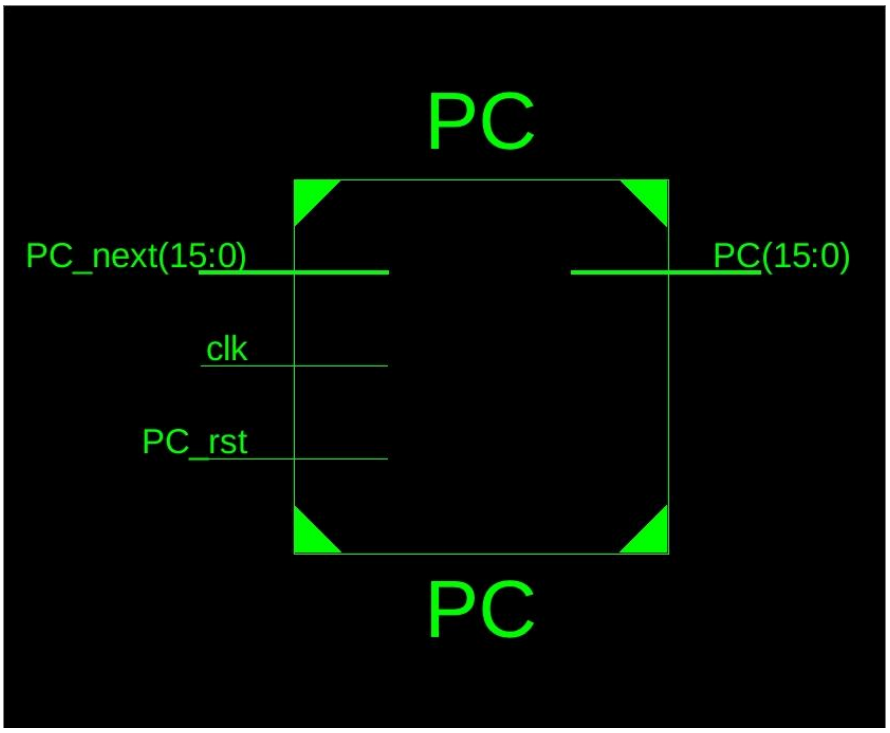



- **PC Circuitry**

Verilog HDL:

```
module PC(
    input [15:0] PC_next,
    input PC_rst,
    input clk,
    output reg [15:0] PC
    );

    always @(posedge clk or posedge PC_rst) begin
        if (PC_rst == 1'b1) begin
            // Reset the counter
            PC <= 16'b0;
        end else begin
            // Update the counter with the next value
            PC <= PC_next;
        end
    end


endmodule
```

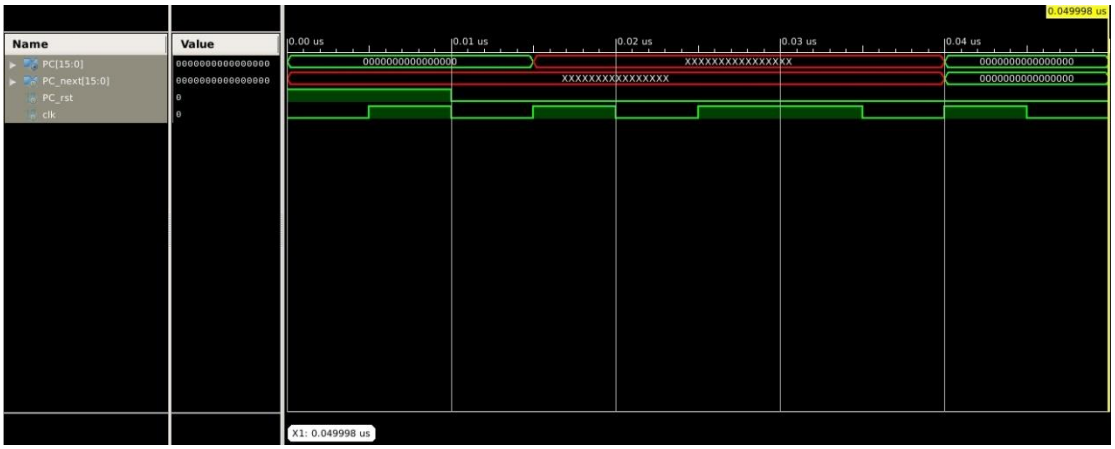
RTL:



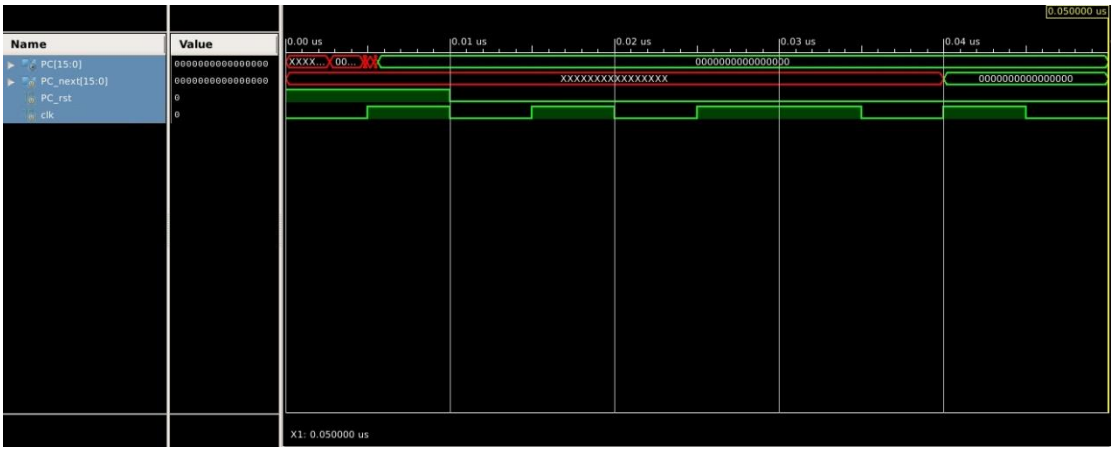
## Testbench:

```
module PC_tb;

    // Inputs
    reg [15:0] PC_next;
    reg PC_rst;
    reg clk;

    // Outputs
    wire [15:0] PC;

    // Instantiate the Unit Under Test (UUT)
    PC uut (
        .PC_next(PC_next),
        .PC_rst(PC_rst),
        .clk(clk),
        .PC(PC)
    );

// Initialize Inputs
    initial begin
        // Initialize Inputs
        PC_rst = 0;
        clk = 0;

        // Apply a reset pulse
        PC_rst = 1;
        #10;
        PC_rst = 0;
        #10;

        // Clock the design
        clk = 1;
        #10;
        clk = 0;
        #10;

    // Test different cases
    $display("Initial PC: %h", PC);
    case (PC_next)
        16'h0000: PC_next = 16'h0001;  // Increment by 1
        16'h0001: PC_next = 16'h0010;  // Increment by 2
        16'h0010: PC_next = 16'h0000;  // Wrap around to 0
        default: PC_next = 16'h0000;  // Default case
    endcase
    #10;
    $display("New PC: %h", PC);
        // Finish the simulation
        $finish;
    end
    always begin
        #5 clk = ~clk;  // Toggle clock every 5 time units
    end
endmodule
```

## Behavioral




## Post-Route

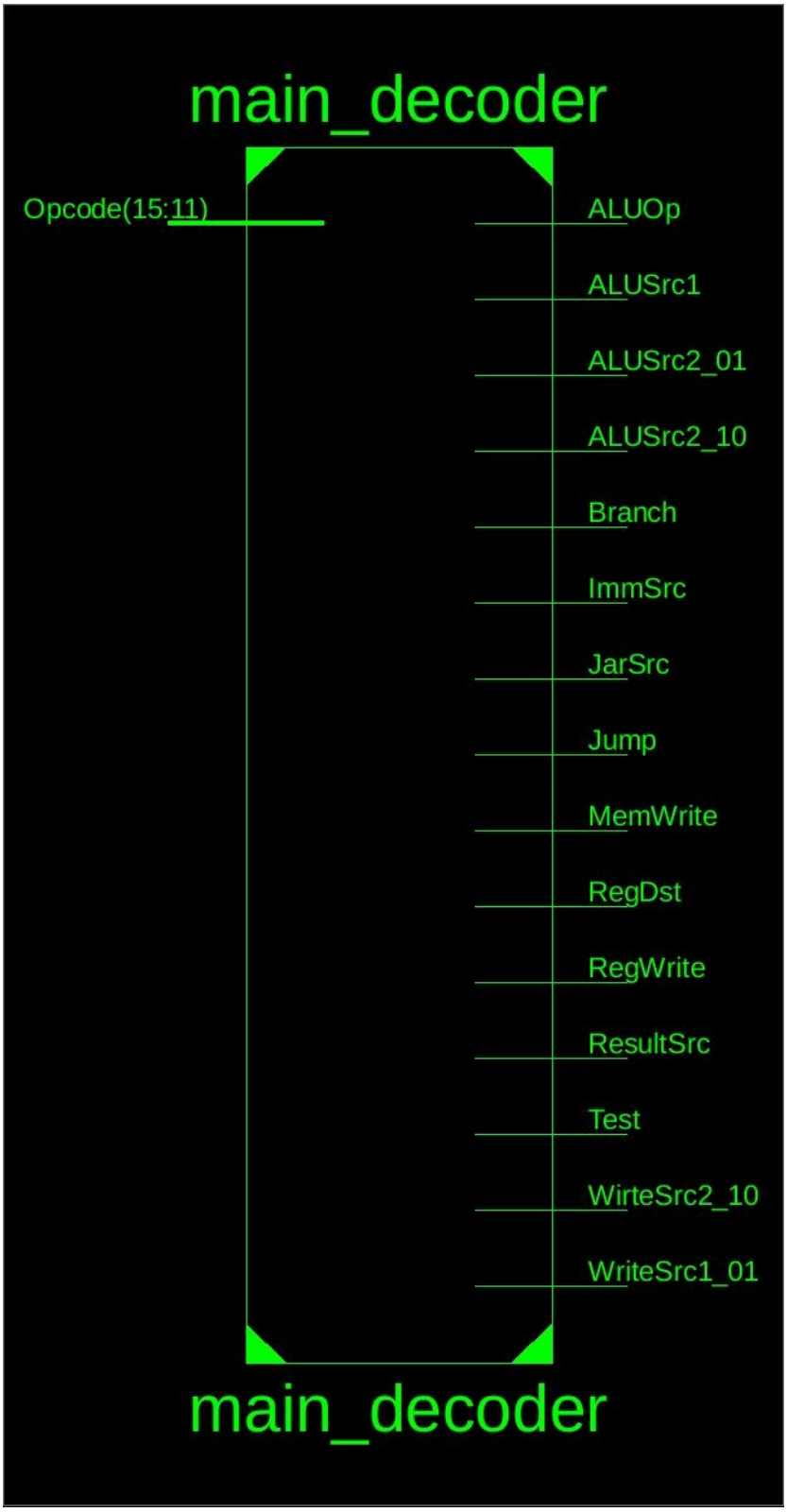


- **Instruction Decoder**

Verilog HDL:

```
module main_decoder(
    input [15:11] Opcode,
    output reg RegDst,
    output reg ALUSrc1,
    output reg ALUSrc2_01,
    output reg ALUSrc2_10,
    output reg ResultSrc,
    output reg MemWrite,
    output reg RegWrite,
    output reg Branch,
    output reg ALUOp,
    output reg WriteSrc1_01,
    output reg WirteSrc2_10,
    output reg ImmSrc,
    output reg Jump,
    output reg JarSrc,
    output reg Test
    );

always @(*) begin
    case(Opcode)
    5'b00000: begin //Rtype
        RegDst = 1'b0;
        ALUSrc1 = 1'b0;
        ALUSrc2_01 = 1'b0;
        ALUSrc2_10 = 1'b0;
        ResultSrc = 1'b0;
        MemWrite = 1'b0;
        RegWrite = 1'b1;
        Branch = 1'b0;
        ALUOp = 1'b0;
        WriteSrc1_01 = 1'b0;
        WirteSrc2_10 = 1'b0;
        ImmSrc = 1'b0;
        Jump = 1'b0;
        JarSrc = 1'b0;
        Test = 1'b0;
    end
    5'b00001: begin //LHI
        RegDst = 1'b1;
        ALUSrc1 = 1'b1;
        ALUSrc2_01 = 1'b0;
        ALUSrc2_10 = 1'b1;
        ResultSrc = 1'b0;
        MemWrite = 1'b0;
        RegWrite = 1'b1;
        Branch = 1'b0;
        ALUOp = 1'b0;
        WriteSrc1_01 = 1'b0;
        WirteSrc2_10 = 1'b0;
        ImmSrc = 1'b1;
        Jump = 1'b0;
        JarSrc = 1'b0;
        Test = 1'b0;
    end
    5'b00010: begin //LLI
        RegDst = 1'b0;
        ALUSrc1 = 1'b0;
        ALUSrc2_01 = 1'b1;
        ALUSrc2_10 = 1'b0;
        ResultSrc = 1'b0;
        MemWrite = 1'b0;
        RegWrite = 1'b1;
        Branch = 1'b0;
        ALUOp = 1'bx;
        WriteSrc1_01 = 1'b0;
        WirteSrc2_10 = 1'b1;
        ImmSrc = 1'b1;
        Jump = 1'b0;
        JarSrc = 1'b0;
        Test = 1'b0;
    end
    5'b00011: begin //LDR
        RegDst = 1'b0;
        ALUSrc1 = 1'b0;
        ALUSrc2_01 = 1'b1;
        ALUSrc2_10 = 1'b0;
        ResultSrc = 1'b1;
        MemWrite = 1'b0;
        RegWrite = 1'b1;
        Branch = 1'b0;
        ALUOp = 1'b0;
        WriteSrc1_01 = 1'b0;
        WirteSrc2_10 = 1'b0;
        ImmSrc = 1'b0;
        Jump = 1'b0;
        JarSrc = 1'b0;
        Test = 1'b0;
    end
    5'b00101: begin //STR
        RegDst = 1'b1;
        ALUSrc1 = 1'b0;
        ALUSrc2_01 = 1'b1;
        ALUSrc2_10 = 1'b0;
        ResultSrc = 1'b0;
        MemWrite = 1'b1;
        RegWrite = 1'b0;
        Branch = 1'b0;
        ALUOp = 1'b0;
        WriteSrc1_01 = 1'b0;
        WirteSrc2_10 = 1'b0;
        ImmSrc = 1'b0;
        Jump = 1'b0;
        JarSrc = 1'b0;
        Test = 1'b0;
    end
    5'b00110: begin //CMP
        RegDst = 1'b0;
        ALUSrc1 = 1'b0;
        ALUSrc2_01 = 1'b0;
        ALUSrc2_10 = 1'b0;
        ResultSrc = 1'b0;
        MemWrite = 1'b0;
        RegWrite = 1'b1;
        Branch = 1'b0;
        ALUOp = 1'b1;
        WriteSrc1_01 = 1'b0;
        WirteSrc2_10 = 1'b0;
        ImmSrc = 1'b0;
        Jump = 1'b0;
        JarSrc = 1'b0;
        Test = 1'b0;
    end
    5'b00111: begin //ADDI
        RegDst = 1'b0;
        ALUSrc1 = 1'b0;
        ALUSrc2_01 = 1'b1;
        ALUSrc2_10 = 1'b0;
        ResultSrc = 1'b0;
        MemWrite = 1'b0;
        RegWrite = 1'b1;
        Branch = 1'b0;
        ALUOp = 1'b0;
        WriteSrc1_01 = 1'b0;
        WirteSrc2_10 = 1'b0;
        ImmSrc = 1'b0;
        Jump = 1'b0;
        JarSrc = 1'b0;
        Test = 1'b0;
    end
    5'b01000: begin //SUBI
        RegDst = 1'b0;
        ALUSrc1 = 1'b0;
        ALUSrc2_01 = 1'b1;
        ALUSrc2_10 = 1'b0;
        ResultSrc = 1'b0;
        MemWrite = 1'b0;
        RegWrite = 1'b1;
        Branch = 1'b0;
        ALUOp = 1'b1;
        WriteSrc1_01 = 1'b0;
        WirteSrc2_10 = 1'b0;
        ImmSrc = 1'b0;
        Jump = 1'b0;
        JarSrc = 1'b0;
        Test = 1'b0;
    end
    5'b01011: begin //MOV
        RegDst = 1'b0;
        ALUSrc1 = 1'b0;
        ALUSrc2_01 = 1'bx;
        ALUSrc2_10 = 1'bx;
        ResultSrc = 1'b0;
        MemWrite = 1'b0;
        RegWrite = 1'b1;
        Branch = 1'b0;
        ALUOp = 1'bx;
        WriteSrc1_01 = 1'b0;
        WirteSrc2_10 = 1'b0;
        ImmSrc = 1'b0;
        Jump = 1'b1;
        JarSrc = 1'b0;
        Test = 1'b0;
    end
    5'b10000: begin //JMP
        RegDst = 1'b0;
        ALUSrc1 = 1'b0;
        ALUSrc2_01 = 1'bx;
        ALUSrc2_10 = 1'bx;
        ResultSrc = 1'b0;
        MemWrite = 1'b0;
        RegWrite = 1'b0;
        Branch = 1'b0;
        ALUOp = 1'bx;
        WriteSrc1_01 = 1'b0;
        WirteSrc2_10 = 1'b0;
        ImmSrc = 1'b0;
        Jump = 1'b1;
        JarSrc = 1'b0;
        Test = 1'b0;
    end
    5'b10001: begin //JAL1
        RegDst = 1'b0;
        ALUSrc1 = 1'b0;
        ALUSrc2_01 = 1'bx;
        ALUSrc2_10 = 1'bx;
        ResultSrc = 1'b0;
        MemWrite = 1'b0;
        RegWrite = 1'b1;
        Branch = 1'b0;
        ALUOp = 1'bx;
        WriteSrc1_01 = 1'b1;
        WirteSrc2_10 = 1'b0;
        ImmSrc = 1'b1;
        Jump = 1'b0;
        JarSrc = 1'b1;
        Test = 1'b0;
    end
    5'b10010: begin //JAL2
        RegDst = 1'b0;
        ALUSrc1 = 1'b0;
        ALUSrc2_01 = 1'bx;
        ALUSrc2_10 = 1'bx;
        ResultSrc = 1'b0;
        MemWrite = 1'b0;
        RegWrite = 1'b1;
        Branch = 1'b0;
        ALUOp = 1'bx;
        WriteSrc1_01 = 1'b1;
        WirteSrc2_10 = 1'b0;
        ImmSrc = 1'b0;
        Jump = 1'b0;
        JarSrc = 1'b0;
        Test = 1'b0;
    end
    5'b10011: begin //JR
        RegDst = 1'b0;
        ALUSrc1 = 1'b0;
        ALUSrc2_01 = 1'bx;
        ALUSrc2_10 = 1'bx;
        ResultSrc = 1'b0;
        MemWrite = 1'b0;
        RegWrite = 1'b0;
        Branch = 1'b0;
        ALUOp = 1'bx;
        WriteSrc1_01 = 1'b0;
        WirteSrc2_10 = 1'b0;
        ImmSrc = 1'b0;
        Jump = 1'b0;
        JarSrc = 1'b1;
        Test = 1'b0;
    end
    5'b11000: begin //Btype
        RegDst = 1'b0;
        ALUSrc1 = 1'b0;
        ALUSrc2_01 = 1'b0;
        ALUSrc2_10 = 1'b0;
        ResultSrc = 1'b0;
        MemWrite = 1'b0;
        RegWrite = 1'b0;
        Branch = 1'b1;
        ALUOp = 1'bx;
        WriteSrc1_01 = 1'b0;
        WirteSrc2_10 = 1'b0;
        ImmSrc = 1'b1;
        Jump = 1'b0;
        JarSrc = 1'b0;
        Test = 1'b1;
    end
    5'b11001: begin //B[AL]
        RegDst = 1'b0;
        ALUSrc1 = 1'b0;
        ALUSrc2_01 = 1'b0;
        ALUSrc2_10 = 1'b0;
        ResultSrc = 1'b0;
        MemWrite = 1'b0;
        RegWrite = 1'b0;
        Branch = 1'b1;
        ALUOp = 1'bx;
        WriteSrc1_01 = 1'b0;
        WirteSrc2_10 = 1'b0;
        ImmSrc = 1'b1;
        Jump = 1'b0;
        JarSrc = 1'b0;
        Test = 1'b1;
    end
    5'b11100: begin //test
        RegDst = 1'b0;
        ALUSrc1 = 1'b0;
        ALUSrc2_01 = 1'b0;
        ALUSrc2_10 = 1'b0;
        ResultSrc = 1'b0;
        MemWrite = 1'b0;
        RegWrite = 1'b0;
        Branch = 1'b0;
        ALUOp = 1'bx;
        WriteSrc1_01 = 1'b0;
        WirteSrc2_10 = 1'b0;
        ImmSrc = 1'b0;
        Jump = 1'b0;
        JarSrc = 1'b0;
        Test = 1'b1;
    end
    endcase
end
endmodule
```

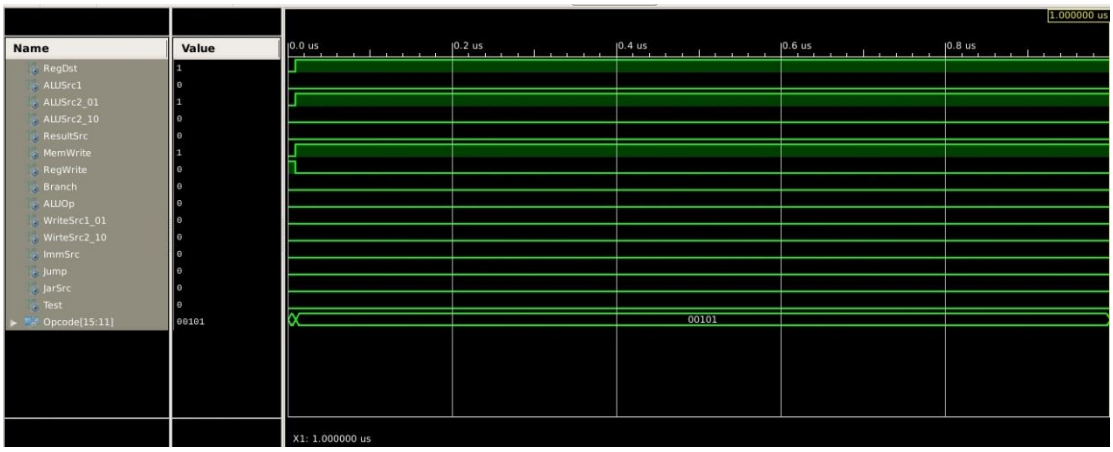
RTL


main_decoder
Opcode(15:11)
ALUOp
ALUSrc1
ALUSrc2_01
ALUSrc2_10
Branch
ImmSrc
JarSrc
Jump
MemWrite
RegDst
RegWrite
ResultSrc
Test
WirteSrc2_10
WriteSrc1_01
main_decoder

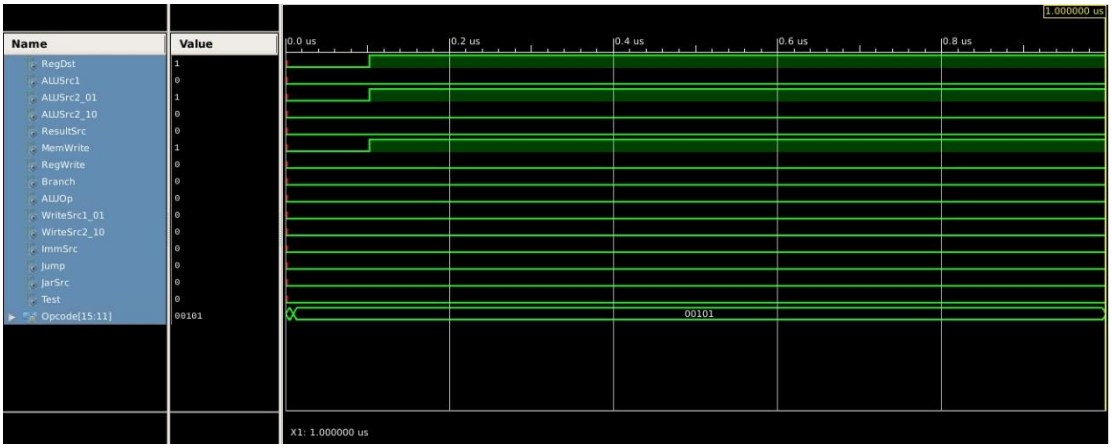
Testbench

```
module main_decoder_tb;

    // Inputs
    reg [15:11] Opcode;

    // Outputs
    wire RegDst;
    wire ALUSrc1;
    wire ALUSrc2_01;
    wire ALUSrc2_10;
    wire ResultSrc;
    wire MemWrite;
    wire RegWrite;
    wire Branch;
    wire ALUOp;
    wire WriteSrc1_01;
    wire WirteSrc2_10;
    wire ImmSrc;
    wire Jump;
    wire JarSrc;
    wire Test;

    // Instantiate the Unit Under Test (UUT)
    main_decoder uut (
        .Opcode(Opcode),
        .RegDst(RegDst),
        .ALUSrc1(ALUSrc1),
        .ALUSrc2_01(ALUSrc2_01),
        .ALUSrc2_10(ALUSrc2_10),
        .ResultSrc(ResultSrc),
        .MemWrite(MemWrite),
        .RegWrite(RegWrite),
        .Branch(Branch),
        .ALUOp(ALUOp),
        .WriteSrc1_01(WriteSrc1_01),
        .WirteSrc2_10(WirteSrc2_10),
        .ImmSrc(ImmSrc),
        .Jump(Jump),
        .JarSrc(JarSrc),
        .Test(Test)
    );

        // Initialize Inputs
    initial begin
        // Test with different opcodes
        Opcode = 5'b00000; // You can change the opcode for testing
        #10; // Wait for some time
        Opcode = 5'b00101; // Another opcode
        #10; // Wait for some time
        // Add more tests with different opcodes as needed
    end

endmodule
```

## Behavioral




## Post-Route



- **Complete Computer**

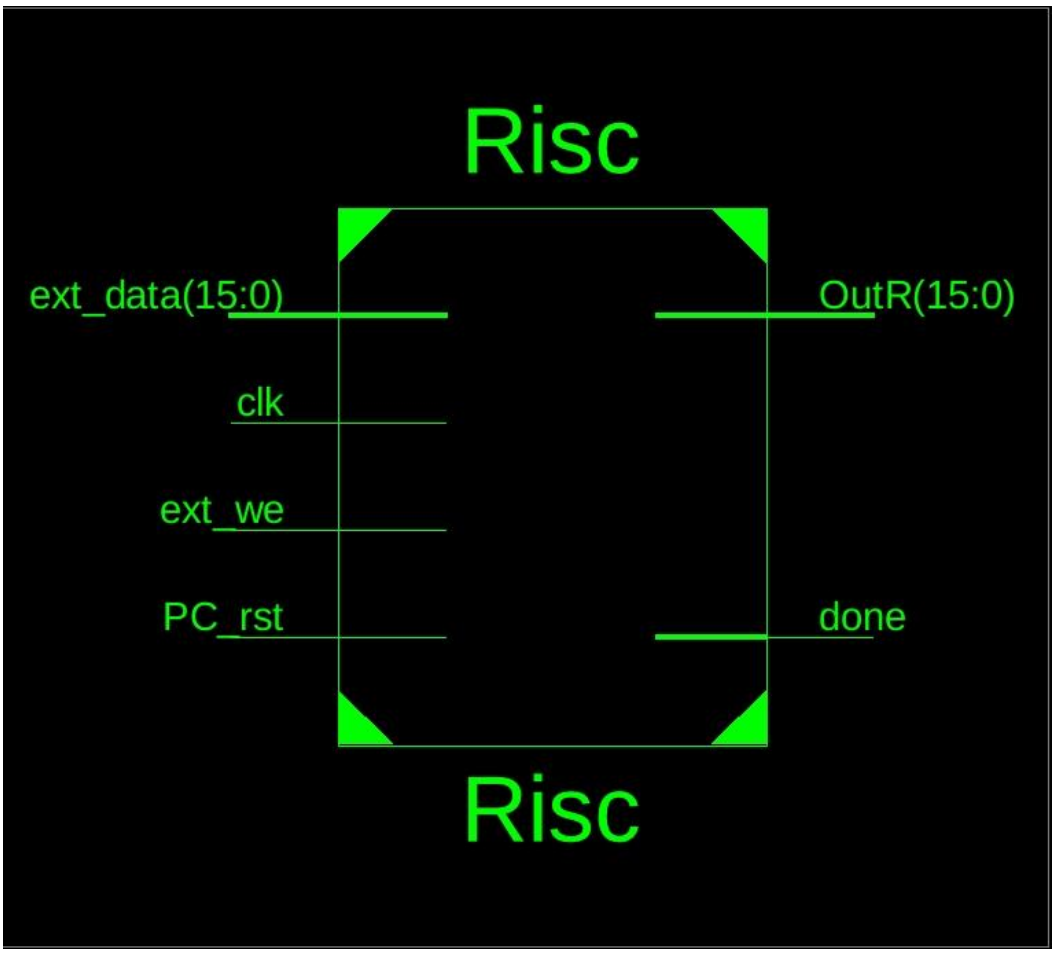
Verilog HDL:

```
module Risc(
    input clk,
    input [15:0] ext_data,
    input ext_we,
    input PC_rst,
    output [15:0] OutR,
    output done
    );

wire [15:0] PC;
wire [15:0] PC_next;
wire [15:0] pcb;
assign pcb = 16'b0000_0000_0000_0001;
wire [15:0] pc2addero;
wire [15:0] pcaddseo;

wire pcen0, pcen1, pcen2, pcen3, pcen4, pcen5, pcen6, pcen7; //8 encoder output to instruction memory
wire [15:0] Instr; // output of instruction memory
wire [2:0] muxregdsto; // mux(3bit) output
wire [15:0] RAd; // output of read_data1
wire [15:0] RBd; // output of read_data2
wire [15:0] RBb; //output of bitmask
wire [15:0] muxalusrclo; //output of muxalusrc1
wire [15:0] muxalusrc2o; //output of muxalusrc2
wire [15:0] aluo; //output of alu
wire N, Z, C, V, cout ; //flag of alu
wire C0,C1,Z0,Z1; //flag of branch detect
assign C0 = C;
assign C1 = ~C;
assign Z0 = Z;
assign Z1 = ~Z;

wire M; //Mode is output of ALU_decoder
wire aluo0, aluo1, aluo2, aluo3, aluo4, aluo5, aluo6, aluo7; //8 encoder output to data memory
wire [15:0] datam; //output of data memory
wire [15:0] muxresultsrco; //output of muxresult
wire [15:0] muxwritesrco; //output of muxwritesrc
wire [15:0] seo; //output of signextend
wire [15:0] shift8o; //output of shift8
wire [15:0] muxJarsrco; //output of muxJarsrc
wire [15:0] muxPCsrco; //output of muxPCsrc
wire RegDst, ALUSrc1, ALUSrc2_01, ALUSrc2_10, ResultSrc, MemWrite, RegWrite, Branch, ALUOp, WriteSrc1_01, WirteSrc2_10, ImmSrc, Jump, JarSrc, Test;
wire bccout, bcsout, bneout, beqout, BAL, branchdetecto;
assign branchdetecto = bccout | bcsout | bneout | beqout | BAL;
wire PCSrc;
assign PCSrc = Branch & branchdetecto;

wire [15:0] tomuxJumpB;
assign tomuxJumpB = {PC[15:12], Instr[10:0], 1'b0};

wire OutRo, HLT;
assign done = HLT & Test;
wire S;
assign S = OutRo & Test;
wire demux0utR0, demux0utR1;
wire muxHLTi;
assign muxHLTi = 1'b0;

PC pc(
    .PC_next(PC_next),
    .PC_rst(PC_rst),
    .clk(clko),
    .PC(PC)
);

adder16bit pc2adder(
    .A(PC),
    .B(pcb),
    .S(pc2addero)
);

encoder1628 encoder1(
    .datain(PC),
    .ecoutput0(pcen0),
    .ecoutput1(pcen1),
    .ecoutput2(pcen2),
    .ecoutput3(pcen3),
    .ecoutput4(pcen4),
    .ecoutput5(pcen5),
    .ecoutput6(pcen6),
    .ecoutput7(pcen7)
);

Memory_module Instructmem(
    .data(ext_data),
    .addr0(pcen0),
    .addr1(pcen1),
    .addr2(pcen2),
    .addr3(pcen3),
    .addr4(pcen4),
    .addr5(pcen5),
    .addr6(pcen6),
    .addr7(pcen7),
    .WEn(ext_we),
    .clk(clko),
    .qout(Instr)
);

mux3bit221 muxregdst(
    .A(Instr[4:2]),
    .B(Instr[10:8]),
    .sel(RegDst),
    .out(muxregdsto)
);

mux16bit321 muxwritesrc(
    .A(muxresultsrco),
    .B(pc2addero),
    .C(demux0utR0),
    .sel({WriteSrc2_10, WriteSrc1_01}),
    .out(muxwritesrco)
);
```

```
signextend se(
    .A(Instr[7:0]),
    .ImmSrc(ImmSrc),
    .out(seo)
);

shift8bit shift8(
    .A(Instr[7:0]),
    .out(shift8o)
);

register_file regfile(
    .RA_addr(Instr[7:5]),
    .RB_addr(Imout1),
    .WR_addr(Instr[10:8]),
    .WR_data(muxwritesrco),
    .WE(RegWrite),
    .clk(clko),
    .RA_data(RAd),
    .RB_data(RBd)
);

Bitmask bm(
    .data(RBd),
    .out(RBb)
);

mux16bit221 muxalusrc1(
    .A(RAd),
    .B(RBb),
    .sel(ALUSrc1),
    .out(muxalusrc1o)
);

mux16bit321 muxalusrc2(
    .A(RBd),
    .B(seo),
    .C(shift8o),
    .sel({ALUSrc2_10, ALUSrc2_01}),
    .out(muxalusrc2o)
);

adder16bit pcaddse(
    .A(pc2addero),
    .B(seo),
    .S(pcaddseo)
);

ALUdecoder alude(
    .funct(Instr[1:0]),
    .ALUOp(ALUOp),
    .M(M)
);

ALU alu(
    .M(M),
    .A(muxalusrc1o),
    .B(muxalusrc2o),
    .S(aluo),
    .N(N),
    .Z(Z),
    .C(C),
    .V(V),
    .cout(cout)
);

encoder1628 encoder2(
    .datain(aluo),
    .ecoutput0(aluo0),
    .ecoutput1(aluo1),
    .ecoutput2(aluo2),
    .ecoutput3(aluo3),
    .ecoutput4(aluo4),
    .ecoutput5(aluo5),
    .ecoutput6(aluo6),
    .ecoutput7(aluo7)
);

Memory_module datamem(
    .data(wd),
    .addr0(aluo0),
    .addr1(aluo1),
    .addr2(aluo2),
    .addr3(aluo3),
    .addr4(aluo4),
    .addr5(aluo5),
    .addr6(aluo6),
    .addr7(aluo7),
    .WEn(MemWrite),
    .clk(clko),
    .qout(datam)
);

mux16bit221 muxresultsrc(
    .A(aluo),
    .B(datam),
    .sel(ResultSrc),
    .out(muxresultsrco)
);

mux16bit221 muxPCsrc(
    .A(muxJarsrco),
    .B(pcaddseo),
    .sel(PCSrc),
    .out(muxPCsrco)
);

mux16bit221 muxJarsrc(
    .A(pc2addero),
    .B(RAd),
    .sel(JarSrc),
    .out(muxJarsrco)
);

mux16bit221 muxJump(
    .A(muxPCsrco),
    .B(tomuxJumpB),
    .sel(Jump),
    .out(PC_next)
);
```

```
branchdetech bd(
    .funct(Instr[11:8]),
    .C0(C0),
    .C1(C1),
    .Z0(Z0),
    .Z1(Z1),
    .bccout(bccout),
    .bcsout(bcsout),
    .bneout(bneout),
    .beqout(beqout),
    .BAL(BAL)
);

main_decoder control(
    .Opcode(Instr[15:11]),
    .RegDst(RegDst),
    .ALUSrc1(ALUSrc1),
    .ALUSrc2_01(ALUSrc2_01),
    .ALUSrc2_10(ALUSrc2_10),
    .ResultSrc(ResultSrc),
    .MemWrite(MemWrite),
    .RegWrite(RegWrite),
    .Branch(Branch),
    .ALUOp(ALUOp),
    .WriteSrc1_01(WriteSrc1_01),
    .WriteSrc2_10(WriteSrc2_10),
    .ImmSrc(ImmSrc),
    .Jump(Jump),
    .JarSrc(JarSrc),
    .Test(Test)
);

testdecoder td(
    .funct(Instr[1:0]),
    .OutR(OutRo),
    .HLT(HLT)
);

demux16bit122 demuxOutR(
    .A(RAd),
    .sel(S),
    .Y0(demuxOutR0),
    .Y1(demuxOutR1)
);

Dflipflop16bitclken OutRegister(
    .sin(demuxOutR1),
    .clk(clko),
    .ce(S),
    .qout(OutR)
);

mux221 muxHLT(
    .A(clk),
    .B(muxHLTi),
    .sel(done),
    .out(clko)
);
endmodule
```

RTL:




Testbench:

```
`timescale 1ns / 1ps

module RISCV16bit_tb();

// Inputs
    parameter clk_period = 20;
    parameter delay_factor = 2;
    reg clk;
    reg [15:0] ext_data;
    reg PC_rst;
    reg ext_we;

// Output
    wire [15:0] OutR;
    wire done;
// Instantiate the UUT
    RISCV16bit UUT (
        .clk(clk),
        .ext_data(ext_data),
        .ext_we(ext_we),
        .PC_rst(PC_rst),
        .OutR(OutR),
        .done(done)
    );
```

```
// Clock generation
    always begin
       #(clk_period / 2) clk <= 1'b0;
       #(clk_period / 2) clk <= 1'b1;
    end
// Initialize Inputs
    initial begin
    // Find max&min
        PC_rst <= 1'b0;
        repeat (9) @(posedge clk)
                   #(clk_period / delay_factor) PC_rst <= 1'b0;
        PC_rst <= 1'b1;

        write_mem1(16 'b0000_1000_0111_1111 ) ; // LHI R0,#127
        write_mem1(16 'b0000_1001_0000_0000 ) ; // LHI R1,#0

        write_mem1(16 'b0001_1010_1010_0000 ) ; // LDR R2,R5,#0
        loop1:
        write_mem1(16 'b0001_1011_1010_0001 ) ; // LDR R3,R5,#1

        write_mem1(16 'b0000_0100_0100_1110 ) ; // SUB R4,R2,R3
        write_mem1(16 'b1100_0010_0000_0010 ) ; // BCS next_min
        write_mem1(16 'b0101_1000_0110_0000 ) ; // MOV R0,R3
        next_min:
        write_mem1(16 'b0000_0100_0010_0010 ) ; // SUB R4,R1,R3
        write_mem1(16 'b1100_0011_0000_0010 ) ; // BCC next_max
        write_mem1(16 'b0101_1001_0110_0000 ) ; // MOV R1,R3

        next_max:
        write_mem1(16 'b0011_1101_1010_0001 ) ; // ADDI R5,R5,#1
        write_mem1(16 'b1100_0001_1111_0101 ) ; // BNE loop2

        done1:
        write_mem1(16 'b1110_0000_0000_0000 ) ; // OutR R0
        $display("Minimum Value (R0): %h", OutR);
        write_mem1(16 'b1110_0000_0010_0000 ) ; // OutR R1

        $display("Maximum Value (R1): %h", OutR);
        write_mem1(16 'b1110_0000_0000_0001 ) ; // HLT
        @(posedge clk) #(clk_period / delay_factor) ext_we = 1'b0;
        //read data from the dual-port memory

        repeat (9) @(posedge clk)
                   #(clk_period / delay_factor) PC_rst = 1'b0;
        PC_rst = 1'b1;
        wait (done);
    end
    initial begin
    // ADD 2 numbers in mem and store result in another location
        PC_rst <= 1'b0;
        repeat (9) @(posedge clk)
                   #(clk_period / delay_factor) PC_rst <= 1'b0;
        PC_rst <= 1'b1;
        write_mem2(16 'b0001_1000_1010_0000 ) ; // LDR R0,R5,#0
        write_mem2(16 'b0001_1001_1100_0000 ) ; // LDR R1,R6,#0
        write_mem2(16 'b0000_0010_0000_0100 ) ; // ADD R2,R0,R1
```

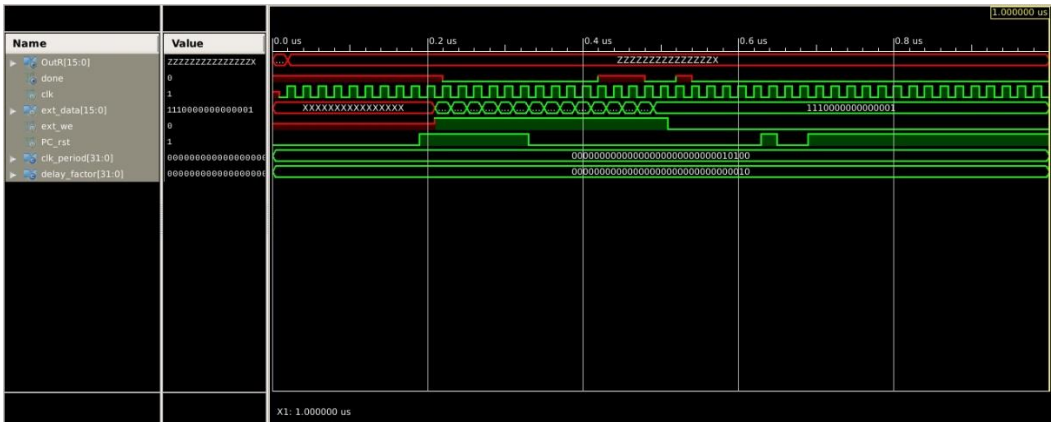
```
        write_mem2(16 'b0010_1010_1110_0000 ) ; // STR R2,R7,#0
        write_mem2(16 'b1110_0000_0000_0001 ) ; // HLT
        @(posedge clk) #(clk_period / delay_factor) ext_we = 1'b0;
        //read data from the dual-port memory

        repeat (9) @(posedge clk)
                   #(clk_period / delay_factor) PC_rst = 1'b0;
        PC_rst = 1'b1;
        wait (done);
    end
    initial begin
    // ADD 10 numbers in consecutive in mem.
        PC_rst <= 1'b0;
        repeat (9) @(posedge clk)
                   #(clk_period / delay_factor) PC_rst <= 1'b0;
        PC_rst <= 1'b1;
        write_mem3(16 'b0000_1000_0000_0000 ) ; // LHI R0,#0
        write_mem3(16 'b0000_1001_0000_0000 ) ; // LHI R1,#0
        write_mem3(16 'b0000_1011_0000_1010 ) ; // LHI R3,#10
        loop2:
        write_mem3(16 'b0001_1010_1010_0000 ) ; // LDR R2,R5,#0
        write_mem3(16 'b0000_0000_0000_1000 ) ; // ADD R0,R0,R2
        write_mem3(16 'b0011_1001_0010_0001 ) ; // ADDI R1,R1,#1
        write_mem3(16 'b0011_1101_1010_0001 ) ; // ADDI R5,R5,#1
        write_mem3(16 'b0011_0000_0010_1101 ) ; // CMP R1,R3
        write_mem3(16 'b1100_0001_1111_1010 ) ; // BNE loop2
        write_mem3(16 'b1110_0000_0000_0000 ) ; // OutR R0
        write_mem3(16 'b1110_0000_0000_0001 ) ; // HLT
```

```
    @(posedge clk) #(clk_period / delay_factor) ext_we = 1'b0;
    //read data from the dual-port memory

    repeat (9) @(posedge clk)
               #(clk_period / delay_factor) PC_rst = 1'b0;
    PC_rst = 1'b1;
    wait (done);
end
initial begin
// Mov a mem block of N words from one place to another.
    PC_rst <= 1'b0;
    repeat (9) @(posedge clk)
               #(clk_period / delay_factor) PC_rst <= 1'b0;
    PC_rst <= 1'b1;
    write_mem4(16 'b0000_1000_0000_0000 ) ; // LHI R0,#0
    write_mem4(16 'b0000_1001_0000_0010 ) ; // LHI R1,#2
    write_mem4(16 'b0000_1010_0010_0000 ) ; // LHI R2,#32
    write_mem4(16 'b0000_1011_0100_0000 ) ; // LHI R3,#64
    move_loop:
    write_mem4(16 'b0001_1100_0100_0000 ) ; // LDR R4,R2,#0
    write_mem4(16 'b0010_1100_0110_0000 ) ; // STR R4,R3,#0
    write_mem4(16 'b0011_1000_0000_0001 ) ; // ADDI R0,R0,#1
    write_mem4(16 'b0011_1010_0010_0010 ) ; // ADDI R2,R2,#2
    write_mem4(16 'b0011_1011_0011_0010 ) ; // ADDI R3,R3,#2
    write_mem4(16 'b0011_0000_0010_0001 ) ; // CMP R0,R1
    write_mem4(16 'b1100_0001_1000_0110 ) ; // BNE move_loop
    write_mem4(16 'b1110_0000_0000_0001 ) ; // HLT
    @(posedge clk) #(clk_period / delay_factor) ext_we = 1'b0;
```

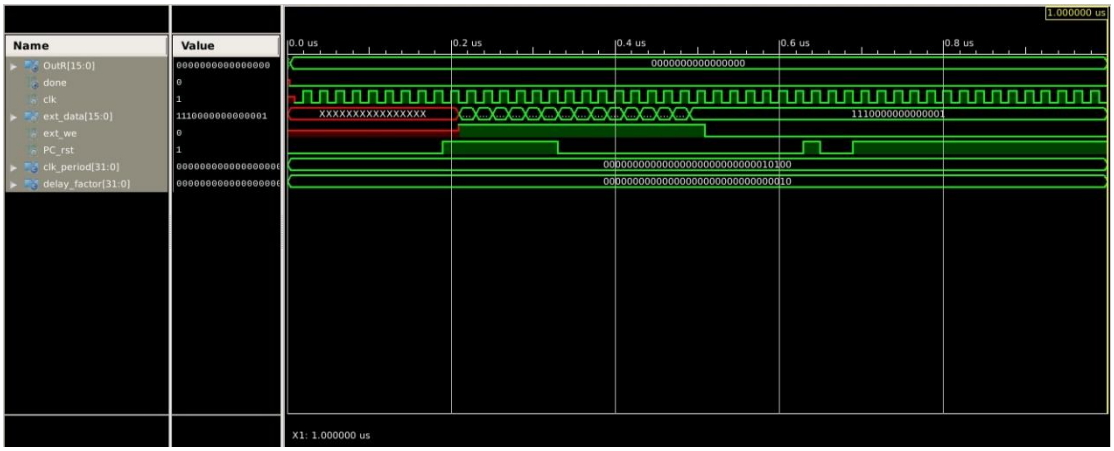
```
        repeat (9) @(posedge clk)
                    #(clk_period / delay_factor) PC_rst = 1'b0;
        PC_rst = 1'b1;
        wait (done);
    end
    task write_mem1;
    input [15:0] data;
    begin
       @(posedge clk) #(clk_period/delay_factor) begin
           ext_we = 1'b1;
           ext_data = data;
        end
    end

    endtask

    task write_mem2;
    input [15:0] data;
    begin
       @(posedge clk) #(clk_period/delay_factor) begin
           ext_we = 1'b1;
           ext_data = data;
        end
    end

    endtask
```

```
    task write_mem3;
    input [15:0] data;
    begin
       @(posedge clk) #(clk_period/delay_factor) begin
           ext_we = 1'b1;
           ext_data = data;
        end
    end

    endtask
    task write_mem4;
    input [15:0] data;
    begin
       @(posedge clk) #(clk_period/delay_factor) begin
           ext_we = 1'b1;
           ext_data = data;
        end
    end

    endtask

    initial #1000000000 $finish;
    initial
       $monitor ($realtime, "ns %h %h %h %h %h %h \n", clk, PC_rst, ext_we, ext_data, OutR, done);
endmodule
```

## Behavioral



Post-Route




# 3. Programs

1. Find the minimum and maximum from two numbers in memory.

```
write_mem1(16 'b0000_1000_0111_1111 ) ; // LHI R0,#127
write_mem1(16 'b0000_1001_0000_0000 ) ; // LHI R1,#0

write_mem1(16 'b0001_1010_1010_0000 ) ; // LDR R2,R5,#0
loop1:
write_mem1(16 'b0001_1011_1010_0001 ) ; // LDR R3,R5,#1

write_mem1(16 'b0000_0100_0100_1110 ) ; // SUB R4,R2,R3
write_mem1(16 'b1100_0010_0000_0010 ) ; // BCS next_min
write_mem1(16 'b0101_1000_0110_0000 ) ; // MOV R0,R3
next_min:
write_mem1(16 'b0000_0100_0010_0010 ) ; // SUB R4,R1,R3
write_mem1(16 'b1100_0011_0000_0010 ) ; // BCC next_max
write_mem1(16 'b0101_1001_0110_0000 ) ; // MOV R1,R3
next_max:
write_mem1(16 'b0011_1101_1010_0001 ) ; // ADDI R5,R5,#1
write_mem1(16 'b1100_0001_1111_0101 ) ; // BNE loop2

done1:
write_mem1(16 'b1110_0000_0000_0000 ) ; // OutR R0
$display("Minimum Value (R0): %h", OutR);
write_mem1(16 'b1110_0000_0010_0000 ) ; // OutR R1

$display("Maximum Value (R1): %h", OutR);
write_mem1(16 'b1110_0000_0000_0001 ) ; // HLT
```

```
    // Initialization
    LHI     R0, #32767; R0 = 32767
    LHI     R1, #0; R1 = 0

    // Load the first number from memory into R2
    LDR     R2, [R5, #0]; R2 = Mem[R5]

loop:// Load the next number from memory into R3
    LDR     R3, [R5, #1]; R3 = Mem[R5 + 1]

    // Compare R2 and R3 to find the minimum
    SUB     R4, R2, R3; R4 = R2 - R3
    BCS     next_min; Jump to next_min if R4 is not positive
    MOV     R0, R3; R0 = R3

next_min:// Compare R3 and R1 to find the maximum
    SUB     R4, R1, R3; R4 = R1 - R3
    BCC     next_max; Jump to next_max if R4 is not positive
    MOV     R1, R3; R1 = R3

next_max:
    ADDI R5, R5, #1 ; R5 = R5 + 1
    BNE loop;

done:// R0 and R1 now contain the minimum and maximum values
    // End of program
    OutR R0;
    OutR R1;
    HLT
```

2. Add two numbers in memory and store the result in another

memory location.

```
//Load the first number from memory into R0
    LDR     R0, [R5, #0];
//Load the second number from memory into R1
    LDR     R1, [R6, #0];
//Add the numbers
    ADD     R2, R0, R1; R2 = R0 + R1
//Store the result back into memory at address R7
    STR     R2, [R7, #0];
// End of program
    HLT;
```

```
write_mem2(16 'b0001_1000_1010_0000 ) ; // LDR R0,R5,#0
write_mem2(16 'b0001_1001_1100_0000 ) ; // LDR R1,R6,#0
write_mem2(16 'b0000_0010_0000_0100 ) ; // ADD R2,R0,R1
write_mem2(16 'b0010_1010_1110_0000 ) ; // STR R2,R7,#0
write_mem2(16 'b1110_0000_0000_0001 ) ; // HLT
```

2. Add ten numbers in consecutive memory locations.

```
write_mem3(16 'b0000_1000_0000_0000 ) ; // LHI R0,#0
write_mem3(16 'b0000_1001_0000_0000 ) ; // LHI R1,#0
write_mem3(16 'b0000_1011_0000_1010 ) ; // LHI R3,#10
loop2:
write_mem3(16 'b0001_1010_1010_0000 ) ; // LDR R2,R5,#0
write_mem3(16 'b0000_0000_0000_1000 ) ; // ADD R0,R0,R2
write_mem3(16 'b0011_1001_0010_0001 ) ; // ADDI R1,R1,#1
write_mem3(16 'b0011_1101_1010_0001 ) ; // ADDI R5,R5,#1
write_mem3(16 'b0011_0000_0010_1101 ) ; // CMP R1,R3
write_mem3(16 'b1100_0001_1111_1010 ) ; // BNE loop2
write_mem3(16 'b1110_0000_0000_0000 ) ; // OutR R0
write_mem3(16 'b1110_0000_0000_0001 ) ; // HLT
```

```
// Initialization
LHI R0, #0; Initialize R0 to store the sum
LHI R1, #0; Initialize R1 as a loop counter
LHI R3, #10; Set R3 to 10, indicating we want to add ten numbers

// Loop
loop:
    LDR R2, [R5, #0]; Load the first number into R2
    ADD R0, R0, R2; Add the number in R2 to the sum in R0
    ADDI R1, R1, #1; Increment the loop counter in R1
    ADDI R5, R5, #1; Move to the next memory location
    CMP R1, R3; Compare the counter to 10
    BNE loop; Continue the loop if the counter is not yet 10

// End of program
OutR R0; Display the sum
HLT; End the program
```

4. Mov a memory block of N words from one place to another.

```
write_mem4(16 'b0000_1000_0000_0000 ) ; // LHI R0,#0
write_mem4(16 'b0000_1001_0000_0010 ) ; // LHI R1,#2
write_mem4(16 'b0000_1010_0010_0000 ) ; // LHI R2,#32
write_mem4(16 'b0000_1011_0100_0000 ) ; // LHI R3,#64
move_loop:
write_mem4(16 'b0001_1100_0100_0000 ) ; // LDR R4,R2,#0
write_mem4(16 'b0010_1100_0110_0000 ) ; // STR R4,R3,#0
write_mem4(16 'b0011_1000_0000_0001 ) ; // ADDI R0,R0,#1
write_mem4(16 'b0011_1010_0010_0010 ) ; // ADDI R2,R2,#2
write_mem4(16 'b0011_1011_0011_0010 ) ; // ADDI R3,R3,#2
write_mem4(16 'b0011_0000_0010_0001 ) ; // CMP R0,R1
write_mem4(16 'b1100_0001_1000_0110 ) ; // BNE move_loop
write_mem4(16 'b1110_0000_0000_0001 ) ; // HLT
```

```
// Initialization
    LHI R0, #0;
    LHI R1, #2;
    LHI R2, #32;
    LHI R3, #64;
// Loop
move_loop:
    LDR R4, [R2, #0];
    STR R4, [R3, #0];
    ADDI R0, R0, #1;
    ADDI R2, R2, #2;
    ADDI R3, R3, #2;
    CMP RO, R1;
    BNE move_loop;

// End of program
    HLT;
```

# 4. Discussion

1. 一開始本來很擔心要花很多時間來寫 verilog，還好就如同老師說的，verilog 是硬體描述語言只要把邏輯閘、電路、整個線路佈好自然可以順利的把作業寫出來。
2. 一開始在 module 串接遇到小問題，不太知道怎麼把一些小 module 做成一個大 module，就像是在主程式中 call 副程式來用，後來在網路上看到一些作法，才知道說 verilog 的呼叫是更加簡單，只是要把一些 wire 寫清楚才不會搞混。
3. 在寫作業的時候剛好課堂上有教到有線狀態機的部分，如果有機會會想把它實作出來，會對於這塊更加清楚，上課的時候還處於能夠聽懂但在實作上還是有點不清楚，這一部分需要花一段時間來吸收。
4. 最後是一些分享，原來 FPGA 最重要的還是數位邏輯、數位系統、計算機組織等這些在大學部基礎的課程，這一部分基礎打好，在 verilog 是更加如魚得水的，因為我在期中遇到畫電路上的問題是比較大的，也花了最久的時間，期末的 verilog 對資工系的我來說稍微比較簡單，而且也透過這樣的學習更清楚的認識 verilog 與 C/C++之間的差異性。

# 5.Conclusion

## Hardware Cost:

| Risc Project Status | | | |
|---|---|---|---|
| **Project File:** | RiscV.xise | **Parser Errors:** | No Errors |
| **Module Name:** | Risc | **Implementation State:** | Placed and Routed |
| **Target Device:** | xc3s100e-4cp132 | **• Errors:** | No Errors |
| **Product Version:** | ISE 14.7 | **• Warnings:** | 543 Warnings (537 new) |
| **Design Goal:** | Balanced | **• Routing Results:** | All Signals Completely Routed |
| **Design Strategy:** | Xilinx Default (unlocked) | **• Timing Constraints:** | |
| **Environment:** | System Settings | **• Final Timing Score:** | 0 (Timing Report) |

| Device Utilization Summary | | | | [-] |
|---|---|---|---|---|
| **Logic Utilization** | **Used** | **Available** | **Utilization** | **Note(s)** |
| Number of Slices containing only related logic | 0 | 0 | 0% | |
| Number of Slices containing unrelated logic | 0 | 0 | 0% | |
| Number of bonded IOBs | 17 | 83 | 20% | |

| Performance Summary | | | | [-] |
|---|---|---|---|---|
| **Final Timing Score:** | 0 (Setup: 0, Hold: 0) | **Pinout Data:** | Pinout Report | |
| **Routing Results:** | All Signals Completely Routed | **Clock Data:** | Clock Report | |
| **Timing Constraints:** | | | | |

| Detailed Reports | | | | | [-] |
|---|---|---|---|---|---|
| **Report Name** | **Status** | **Generated** | **Errors** | **Warnings** | **Infos** |
| Synthesis Report | Current | Sun Nov 26 12:11:15 2023 | 0 | 543 Warnings (537 new) | 21 Infos (21 new) |
| Translation Report | Current | Sun Nov 26 12:11:33 2023 | 0 | 0 | 0 |
| Map Report | Current | Sun Nov 26 12:11:37 2023 | 0 | 0 | 4 Infos (2 new) |
| Place and Route Report | Current | Sun Nov 26 12:11:41 2023 | 0 | 0 | 1 Info (0 new) |
| Power Report | | | | | |
| Post-PAR Static Timing Report | Current | Sun Nov 26 12:11:44 2023 | 0 | 0 | 6 Infos (0 new) |
| Bitgen Report | | | | | |

| Secondary Reports | | | [-] |
|---|---|---|---|
| **Report Name** | **Status** | **Generated** | |
| Post-Synthesis Simulation Model Report | Current | Sun Nov 26 12:11:17 2023 | |

**Date Generated:** 11/29/2023 - 07:13:38