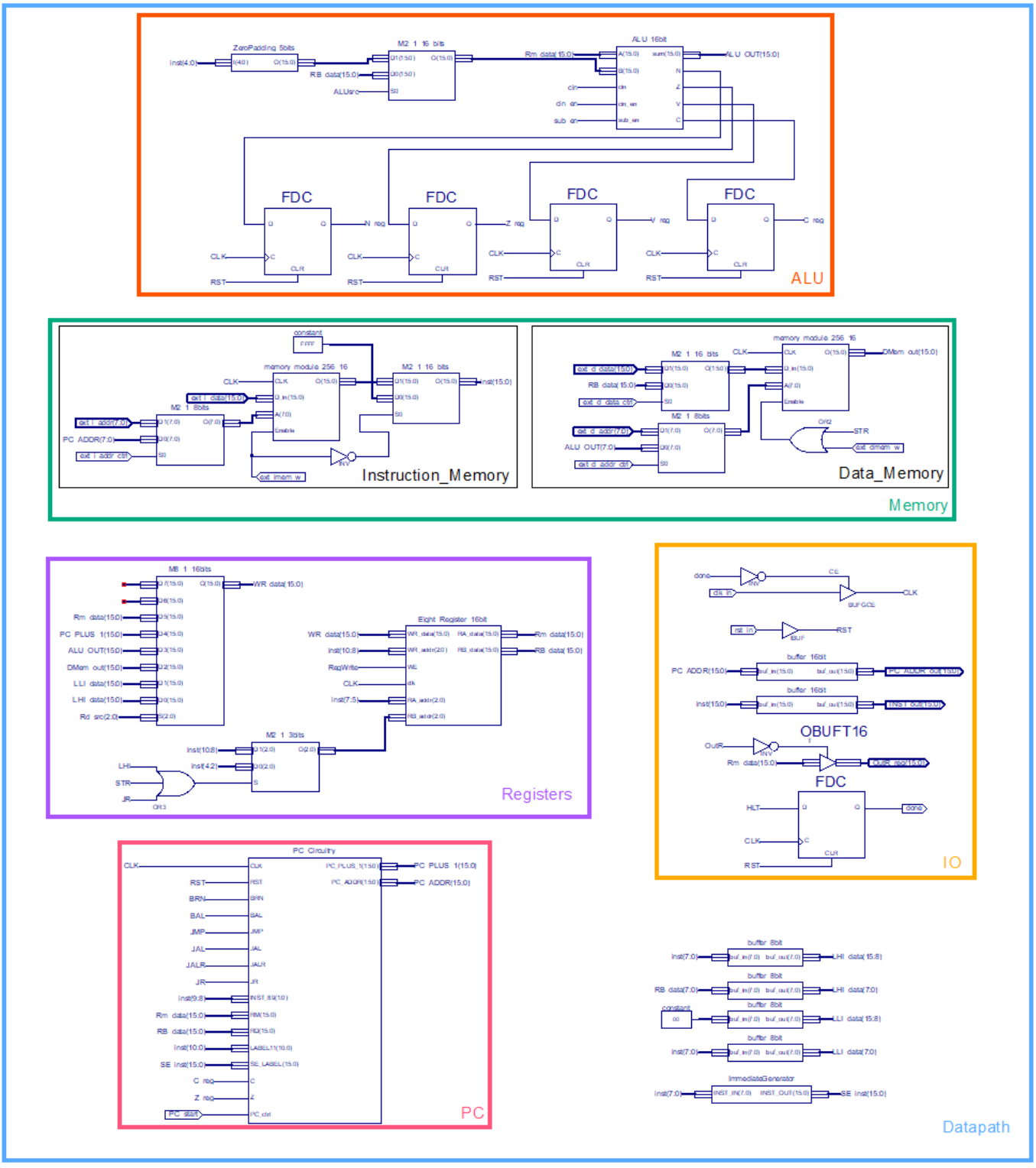
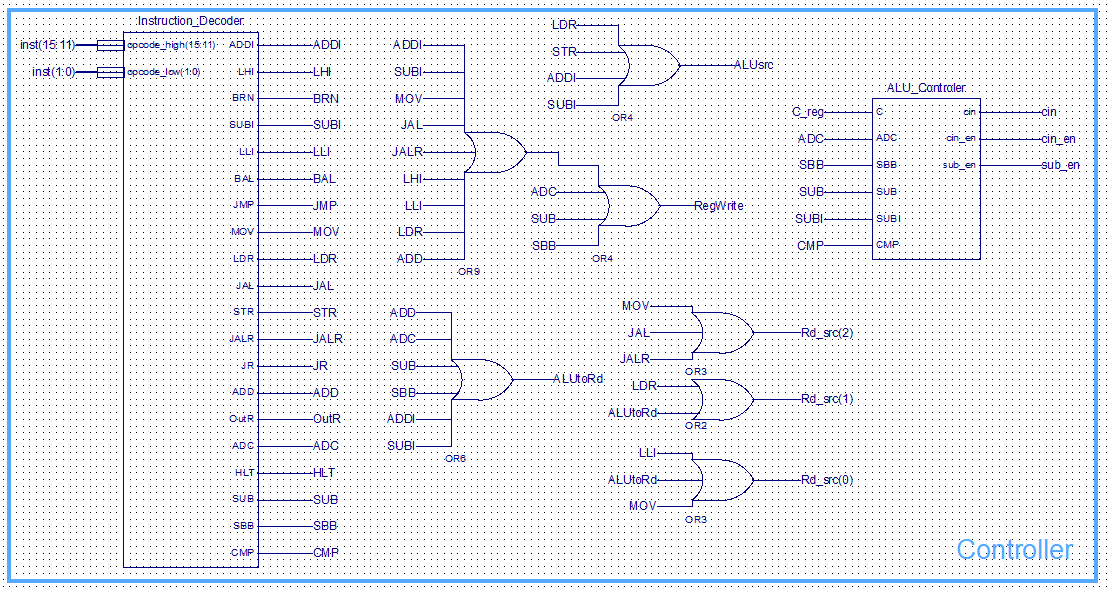
Design and Schematic/HDL Entry

- Datapath



Controller



ALU

```
module ALU
(
    A,
    B,
    Cin,
    Sum,
    Sub,
    Cin_en,
    NZVC
);
    input  [16-1:0] A;
    input  [16-1:0] B;
    input           Cin;
    input           Cin_en;
    input           Sub;
    output [16-1:0] Sum;
    output [ 4-1:0] NZVC;
    wire   [  16:0] result;
    wire            Ci;
    assign Sum = result[15:0];
    assign NZVC[3]   = result[15];
    assign NZVC[2] = ~(|result[15:0]);
    assign NZVC[1] = (A[15] & B[15] &  ~result[15]) (~A[15] & ~B[15] &  result[15]);
    assign NZVC[0] = result[16];
    assign Ci = (Cin_en) ? Cin : 1'b0;
    assign result = (Sub) ? ((Cin_en) ? ({1'b0, A} + {1'b0, ~B} + 1'b1 + {16{~Ci}})
                                      : ({1'b0, A} + {1'b0, ~B} + 1'b1))
                          : ((Cin_en) ? ({1'b0, A} + {1'b0,  B} + {{15{1'b0}}, Ci})
                                      : ({1'b0, A} + {1'b0,  B}));
endmodule
```

Design:

ALU 的設計主要就是由幾個控制腳來決定要做甚麼工作，如果 Cin_en 為 1 則代表電路會有 Cin 輸入，如果 Sub 為 1 則代表 ALU 要做減法。NZVC 的 V

代表 overflow，而當 A,B 的 sign bit 為 0，運算結果的 sign bit 為 1 或是 A,B 的 sign bit 為 1，運算結果的 sign bit 為 0，也就是正+正=負,負+負=正時，代表 APU 溢位了。

- ALU_Controller

```
module ALU_Conrtoller
(
    opcode,
    ALU_op,
    Sub,
    Cin_en
);
    input  [4:0] opcode;
    input  [1:0] ALU_op;
    output reg   Sub;
    output wire  Cin_en;

    assign Cin_en = (opcode == 5'b00000) ? ALU_op[0] : 1'b0;
    always @(*)
    begin
        case(opcode)
            5'b00000:
            begin
                case(ALU_op[1:0])
                    2'b00: Sub = 1'b0;
                    2'b01: Sub = 1'b0;
                    2'b10: Sub = 1'b1;
                    2'b11: Sub = 1'b1;
                    default:Sub = 1'b0;
                endcase
            end
            5'b00110: Sub = 1'b1;
            5'b01000: Sub = 1'b1;
            default:        Sub = 1'b0;
        endcase
    end
endmodule
```

Design:

透過 opcode 來判斷什麼時候要做減法以及會有 Cin 等等控制訊號

- Memory

```
module Memory
(
    clk,
    addr,
    idata,
    w_enable,
    odata
);
    input            clk;
    input  [ 8-1:0] addr;
    input  [16-1:0] idata;
    input            w_enable;
    output [16-1:0] odata;

    wire   [16-1:0] odata;
    reg      [15:0] memory[0:255];

    assign odata = memory[addr];
    always@(posedge clk) begin
        if(w_enable) memory[addr] <= idata;
    end

endmodule
```

Design:

設計 Memory 供 Data 和 instruction 使用，當 write enable 致能時才會將輸入資料寫入。

RegisterFile

```
module RegisterFile
(
    clk,
    rst_n,
    we,
    ra_addr,
    rb_addr,
    wr_addr,
    wr_data,
    ra_data,
    rb_data
);
    input            clk;
    input            rst_n;
    input            we;
    input  [ 3-1:0] ra_addr;
    input  [ 3-1:0] rb_addr;
    input  [ 3-1:0] wr_addr;
    input  [16-1:0] wr_data;
    output [16-1:0] ra_data;
    output [16-1:0] rb_data;
    reg signed [15:0] register [0:7];

    assign ra_data = register[ra_addr];
    assign rb_data = register[rb_addr];
    always @(posedge clk or negedge rst_n)
    begin
        if (~rst_n) // Initial
        begin
            register[0] <= 16'd0; register[1] <= 16'd0; register
            [2] <= 16'd0; register[3] <= 16'd0;
            register[4] <= 16'd0; register[5] <= 16'd0; register
            [6] <= 16'd0; register[7] <= 16'd0;
        end
        else
        begin
            if (we) // Write
                register[wr_addr] <= wr_data;
        end
    end

endmodule
```

Design:

設計 8 個 general purpose register，並且有兩個 address，給定特定的 address，register 就會將值讀出來，而當寫入致能時才會寫入資料

Controller

```
module Controller
(
    inst,
    ALU_src,
    RegWrite,
    MemWrite,
    RD_src,
    Mem_src,
    PC_src,
    Jmp,
    Jalr,
    Jr,
    OutR,
    Hlt,
    NZVC
);

    localparam LHI  = 5'b00001;
    localparam LLI  = 5'b00010;
    localparam LDR  = 5'b00011;
    localparam STR  = 5'b00101;
    localparam ALU  = 5'b00000;
    localparam CMP  = 5'b00110;
    localparam ADDI = 5'b00111;
    localparam SUBI = 5'b01000;
    localparam MOV  = 5'b01011;
    localparam BRN  = 5'b11000;
    localparam BAL  = 5'b11001;
    localparam JMP  = 5'b10000;
    localparam JAL  = 5'b10001;
    localparam JALR = 5'b10010;
    localparam JR   = 5'b10011;
    localparam OUT  = 5'b11100;
```

```
input  wire [16-1:0] inst;
input  wire [   3:0] NZVC;
output reg ALU_src;
output reg RegWrite;
output reg MemWrite;
output reg RD_src;
output reg [2:0] Mem_src;
output reg PC_src;
output reg Jmp;
output reg Jalr;
output reg Jr;
output reg OutR;
output reg Hlt;

always @(*)
begin
    case (inst[15:11]) // Acturally use 5 bits
        LHI :
        begin
            ALU_src   = 1'b0;
            RegWrite  = 1'b1;
            MemWrite  = 1'b0;
            RD_src    = 1'b1;
            Mem_src   = 3'b000;
            PC_src    = 1'b0;
            Jmp       = 1'b0;
            Jalr      = 1'b0;
            Jr        = 1'b0;
            OutR      = 1'b0;
            Hlt       = 1'b0;
        end
        LLI :
        begin
            ALU_src   = 1'b0;
            RegWrite  = 1'b1;
            MemWrite  = 1'b0;
            RD_src    = 1'b0;
            Mem_src   = 3'b001;
```

```
        PC_src    = 1'b0;
        Jmp       = 1'b0;
        Jalr      = 1'b0;
        Jr        = 1'b0;
        OutR      = 1'b0;
        Hlt       = 1'b0;
    end
    LDR :
    begin
        ALU_src   = 1'b1;
        RegWrite  = 1'b1;
        MemWrite  = 1'b0;
        RD_src    = 1'b0;
        Mem_src   = 3'b010;
        PC_src    = 1'b0;
        Jmp       = 1'b0;
        Jalr      = 1'b0;
        Jr        = 1'b0;
        OutR      = 1'b0;
        Hlt       = 1'b0;
    end
    STR :
    begin
        ALU_src   = 1'b1;
        RegWrite  = 1'b0;
        MemWrite  = 1'b1;
        RD_src    = 1'b1;
        Mem_src   = 3'b000;
        PC_src    = 1'b0;
        Jmp       = 1'b0;
        Jalr      = 1'b0;
        Jr        = 1'b0;
        OutR      = 1'b0;
        Hlt       = 1'b0;
    end
    ALU :
    begin
        ALU_src   = 1'b0;
```

```
        RegWrite  = 1'b1;
        MemWrite  = 1'b0;
        RD_src    = 1'b0;
        Mem_src   = 3'b011;
        PC_src    = 1'b0;
        Jmp       = 1'b0;
        Jalr      = 1'b0;
        Jr        = 1'b0;
        OutR      = 1'b0;
        Hlt       = 1'b0;
    end
    CMP :
    begin
        ALU_src   = 1'b0;
        RegWrite  = 1'b0;
        MemWrite  = 1'b0;
        RD_src    = 1'b0;
        Mem_src   = 3'b000;
        PC_src    = 1'b0;
        Jmp       = 1'b0;
        Jalr      = 1'b0;
        Jr        = 1'b0;
        OutR      = 1'b0;
        Hlt       = 1'b0;
    end
    ADDI:
    begin
        ALU_src   = 1'b1;
        RegWrite  = 1'b1;
        MemWrite  = 1'b0;
        RD_src    = 1'b0;
        Mem_src   = 3'b011;
        PC_src    = 1'b0;
        Jmp       = 1'b0;
        Jalr      = 1'b0;
        Jr        = 1'b0;
        OutR      = 1'b0;
        Hlt       = 1'b0;
```

```
end
SUBI:
begin
    ALU_src   = 1'b1;
    RegWrite  = 1'b1;
    MemWrite  = 1'b0;
    RD_src    = 1'b0;
    Mem_src   = 3'b011;
    PC_src    = 1'b0;
    Jmp       = 1'b0;
    Jalr      = 1'b0;
    Jr        = 1'b0;
    OutR      = 1'b0;
    Hlt       = 1'b0;
end
MOV :
begin
    ALU_src   = 1'b0;
    RegWrite  = 1'b1;
    MemWrite  = 1'b0;
    RD_src    = 1'b0;
    Mem_src   = 3'b100;
    PC_src    = 1'b0;
    Jmp       = 1'b0;
    Jalr      = 1'b0;
    Jr        = 1'b0;
    OutR      = 1'b0;
    Hlt       = 1'b0;
end
BRN :
begin
    ALU_src   = 1'b0;
    RegWrite  = 1'b0;
    MemWrite  = 1'b0;
    RD_src    = 1'b0;
    Mem_src   = 3'b000;
    Jmp       = 1'b0;
    Jalr      = 1'b0;
```

```
        Jr        = 1'b0;
        OutR      = 1'b0;
        Hlt       = 1'b0;
        case (inst[9:8])
            2'b00:  PC_src = ( NZVC[2]) ? 1'b1 : 1'b0;
            2'b01:  PC_src = (!NZVC[2]) ? 1'b1 : 1'b0;
            2'b10:  PC_src = ( NZVC[0]) ? 1'b1 : 1'b0;
            2'b11:  PC_src = (!NZVC[0]) ? 1'b1 : 1'b0;
            default:PC_src = 1'b0;
        endcase
    end
    BAL :
    begin
        ALU_src   = 1'b0;
        RegWrite  = 1'b0;
        MemWrite  = 1'b0;
        RD_src    = 1'b0;
        Mem_src   = 3'b000;
        PC_src    = 1'b1;
        Jmp       = 1'b0;
        Jalr      = 1'b0;
        Jr        = 1'b0;
        OutR      = 1'b0;
        Hlt       = 1'b0;
    end
    JMP :
    begin
        ALU_src   = 1'b0;
        RegWrite  = 1'b0;
        MemWrite  = 1'b0;
        RD_src    = 1'b0;
        Mem_src   = 3'b000;
        PC_src    = 1'b0;
        Jmp       = 1'b1;
        Jalr      = 1'b0;
        Jr        = 1'b0;
        OutR      = 1'b0;
        Hlt       = 1'b0;
```

```
end
JAL :
begin
    ALU_src   = 1'b0;
    RegWrite  = 1'b1;
    MemWrite  = 1'b0;
    RD_src    = 1'b0;
    Mem_src   = 3'b101;
    PC_src    = 1'b1;
    Jmp       = 1'b0;
    Jalr      = 1'b0;
    Jr        = 1'b0;
    OutR      = 1'b0;
    Hlt       = 1'b0;
end
JALR:
begin
    ALU_src   = 1'b0;
    RegWrite  = 1'b1;
    MemWrite  = 1'b0;
    RD_src    = 1'b0;
    Mem_src   = 3'b101;
    PC_src    = 1'b0;
    Jmp       = 1'b0;
    Jalr      = 1'b1;
    Jr        = 1'b0;
    OutR      = 1'b0;
    Hlt       = 1'b0;
end
JR  :
begin
    ALU_src   = 1'b0;
    RegWrite  = 1'b0;
    MemWrite  = 1'b0;
    RD_src    = 1'b1;
    Mem_src   = 3'b000;
    PC_src    = 1'b0;
    Jmp       = 1'b0;
```

```
            Jalr      = 1'b0;
            Jr        = 1'b1;
            OutR      = 1'b0;
            Hlt       = 1'b0;
        end
        OUT :
        begin
            ALU_src   = 1'b0;
            RegWrite  = 1'b0;
            MemWrite  = 1'b0;
            RD_src    = 1'b0;
            Mem_src   = 3'b000;
            PC_src    = 1'b0;
            Jmp       = 1'b0;
            Jalr      = 1'b0;
            Jr        = 1'b0;
            OutR      = ~inst[0];
            Hlt       =  inst[0];
        end
        default: // None
        begin
            ALU_src   = 1'b0;
            RegWrite  = 1'b0;
            MemWrite  = 1'b0;
            RD_src    = 1'b0;
            Mem_src   = 3'b000;
            PC_src    = 1'b0;
            Jmp       = 1'b0;
            Jalr      = 1'b0;
            Jr        = 1'b0;
            OutR      = 1'b0;
            Hlt       = 1'b0;
        end
    endcase
end
endmodule
```

Design:

透過解析 opcode 的方式來設定各個 controller signal 的訊號，設計方法就是透過 ISA 上所對應的指令集來解碼，並寫成 switch case 的方式

Single_Cycle_CPU

```
module Single_Cycle_CPU
(
    clk_in,
    rst_n,
    Out_R,
    PC_start,
    done,
    ext_imem_w,
    ext_i_addr,
    ext_i_data,
    ext_i_addr_ctrl,
    ext_dmem_w,
    ext_d_addr_ctrl,
    ext_d_data_ctrl,
    ext_d_addr,
    ext_d_data
);
    input           clk_in;
    input           rst_n;
    output [16-1:0] Out_R;
    output          done;
    input           PC_start;
    input           ext_i_addr_ctrl;
    input           ext_imem_w;
    input   [8-1:0] ext_i_addr;
    input  [16-1:0] ext_i_data;
    input           ext_d_data_ctrl;
    input           ext_dmem_w;
    input           ext_d_addr_ctrl;
    input   [8-1:0] ext_d_addr;
```

```
input  [16-1:0] ext_d_data;
wire            clk;
wire [16-1:0]   PC_next;
reg  [16-1:0]   PC;
wire [16-1:0]   PC_plus;
wire [16-1:0]   PC_plus_label;
wire [16-1:0]   PC_branch;
wire [16-1:0]   PC_jump;
wire [16-1:0]   PC_jalr;
wire [16-1:0]   inst;
wire [16-1:0]   idata;
wire [ 8-1:0]   iaddr;
wire ALU_src;
wire RegWrite;
wire MemWrite;
wire RD_src;
wire [2:0] Mem_src;
wire PC_src;
wire Jmp;
wire Jalr;
wire Jr;
wire OutR;
wire [ 3-1:0] rm_addr;
wire [ 3-1:0] rn_addr;
wire [ 3-1:0] rd_addr;
wire [16-1:0] rm_data;
wire [16-1:0] rn_data;
wire [16-1:0] rd_data;
wire [ 3-1:0] rb_addr;
wire [16-1:0] B;
wire [16-1:0] ALU_result;
wire [ 4-1:0] NZVC;
wire          Sub;
wire [16-1:0] data;
wire [16-1:0] ddata;
wire [ 8-1:0] daddr;
wire          dwe;
reg [ 4-1:0] NZVC_reg;
```

```
assign clk = (done) ? 1'b0 : clk_in;
assign Out_R = (OutR) ? rm_data : {16{1'bz}} ;
always @(posedge clk or negedge rst_n) begin
    if(~rst_n||~PC_start) PC <= 16'd0;   // Next PC value
    else       PC <= PC_next; // Present PC value
end
assign rd_addr = inst[10:8];
assign rm_addr = inst[7:5];
assign rn_addr = inst[4:2];
assign dwe = MemWrite|ext_dmem_w;
assign PC_plus       = PC + 1'b1;
assign PC_plus_label = PC + {{8{inst[7]}}, inst[7:0]};
always@(posedge clk) NZVC_reg <= NZVC;
MUX_2to1
#(
    .DATA_WIDTH (8 )
)
u_Instruction_Memory_MUX_2to1(
    .s  (ext_i_addr_ctrl  ),
    .i0 (PC[7:0] ),
    .i1 (ext_i_addr ),
    .o  (iaddr  )
);
MUX_2to1
#(
    .DATA_WIDTH (16 )
)
u_Instruction_Latch_MUX_2to1(
    .s  (ext_imem_w ),
    .i0 ( idata  ),
    .i1 (16'd1 ),
    .o  (inst  )
);
Memory u_Instruction_Memory(
    .clk   (clk        ),
    .addr  (iaddr      ),
    .idata (ext_i_data   ),
    .wr    (ext_imem_w     ),
```

```
    .odata (idata      )
);
MUX_2to1
#(
    .DATA_WIDTH (8 )
)
u_Data_Memory_ADDR_MUX_2to1(
    .s  (ext_d_addr_ctrl  ),
    .i0 (ALU_result[7:0] ),
    .i1 (ext_d_addr ),
    .o  (daddr  )
);
MUX_2to1
#(
    .DATA_WIDTH (16)
)
u_Data_Memory_Data_MUX_2to1(
    .s  (ext_d_data_ctrl   ),
    .i0 (rn_data  ),
    .i1 (ext_d_data ),
    .o  (ddata    )
);
Memory u_Data_Memory(
    .clk   (clk            ),
    .addr  (daddr          ),
    .idata (ddata          ),
    .wr    (dwe            ),
    .odata (data           )
);
Controller u_Controller(
    .inst    (inst    ),
    .NZVC     (NZVC_reg  ),
    .ALU_src   (ALU_src   ),
    .RegWrite  (RegWrite  ),
    .MemWrite  (MemWrite  ),
    .RD_src    (RD_src    ),
    .Mem_src   (Mem_src   ),
    .PC_src    (PC_src    ),
```

```
    .Jmp      (Jmp      ),
    .Jalr     (Jalr     ),
    .Jr       (Jr       ),
    .OutR     (OutR     ),
    .Hlt      (done )
);
RegisterFile u_RegisterFile(
    .clk    (clk     ),
    .rst_n  (rst_n   ),
    .we     (RegWrite ),
    .ra_addr (rm_addr  ),
    .rb_addr (rb_addr  ),
    .wr_addr (rd_addr  ),
    .wr_data (rd_data  ),
    .ra_data (rm_data  ),
    .rb_data (rn_data  )
);
MUX_2to1
#(
    .DATA_WIDTH(3)
)
u_RB_MUX_2to1
(
    .s  (RD_src  ),
    .i0 (rn_addr ),
    .i1 (rd_addr ),
    .o  (rb_addr )
);

MUX_2to1 u_ALU_MUX_2to1(
    .s  (ALU_src          ),
    .i0 (rn_data          ),
    .i1 ({11'd0,inst[4:0]}),
    .o  (B                )
);
ALU u_ALU(
    .A     (rm_data    ),
    .B     (B          ),
```

```
    .Cin    (NZVC_reg[0] ),
    .Sub    (Sub         ),
    .Cin_en (Cin_en      ),
    .Sum    (ALU_result  ),
    .NZVC   (NZVC        )
);
ALU_Conrtoller u_ALU_Conrtoller(
    .opcode (inst[15:11] ),
    .ALU_op (inst[1:0]   ),
    .Sub    (Sub         ),
    .Cin_en (Cin_en      )
);
MUX_8to1 u_MUX_8to1(
    .s  (Mem_src                  ),
    .i0 ({inst[7:0], rn_data[7:0]} ),
    .i1 ({{8{1'b0}} ,   inst[7:0]} ),
    .i2 (data                     ),
    .i3 (ALU_result               ),
    .i4 (rm_data                  ),
    .i5 (PC_plus                  ),
    .i6 ({16{1'b0}}               )
    .i7 ({16{1'b0}}               )
    .o  (rd_data                  )
);
MUX_2to1 u_PC_0_MUX_2to1(
    .s  (PC_src       ),
    .i0 (PC_plus      ),
    .i1 (PC_plus_label ),
    .o  (PC_branch    )
);
MUX_2to1 u_PC_1_MUX_2to1(
    .s  (Jmp                    ),
    .i0 (PC_branch               ),
    .i1 ({PC[15:11], inst[10:0]} ),
    .o  (PC_jump                 )
);
MUX_2to1 u_PC_2_MUX_2to1(
    .s  (Jalr                   ),
```

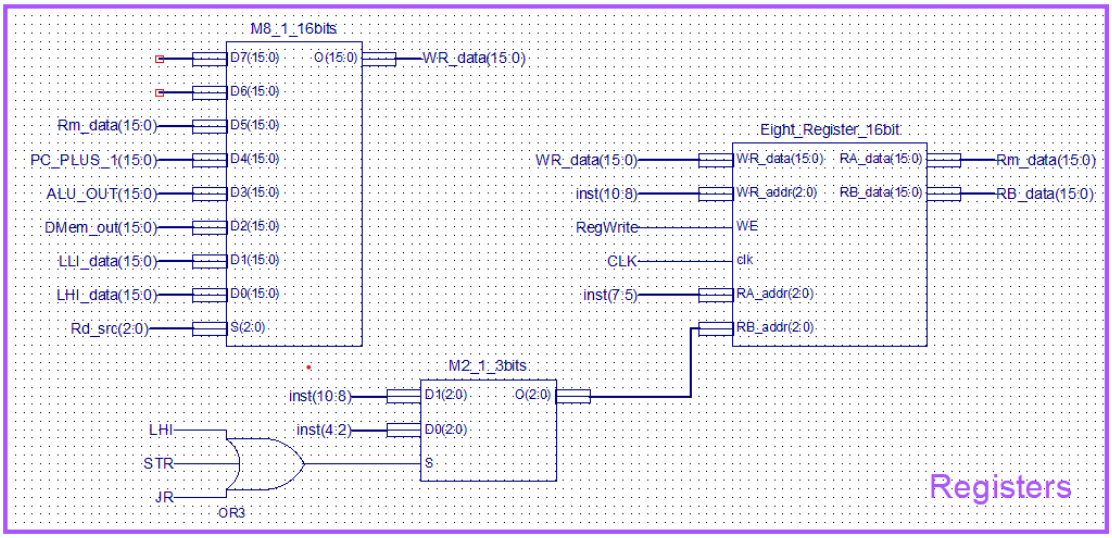
```
        .i0 (PC_jump                ),
        .i1 (rm_data                ),
        .o  (PC_jalr                )
    );
    MUX_2to1 u_PC_3_MUX_2to1(
        .s  (Jr                   ),
        .i0 (PC_jalr                ),
        .i1 (rn_data                ),
        .o  (PC_next                )
    );
endmodule
```

Design:

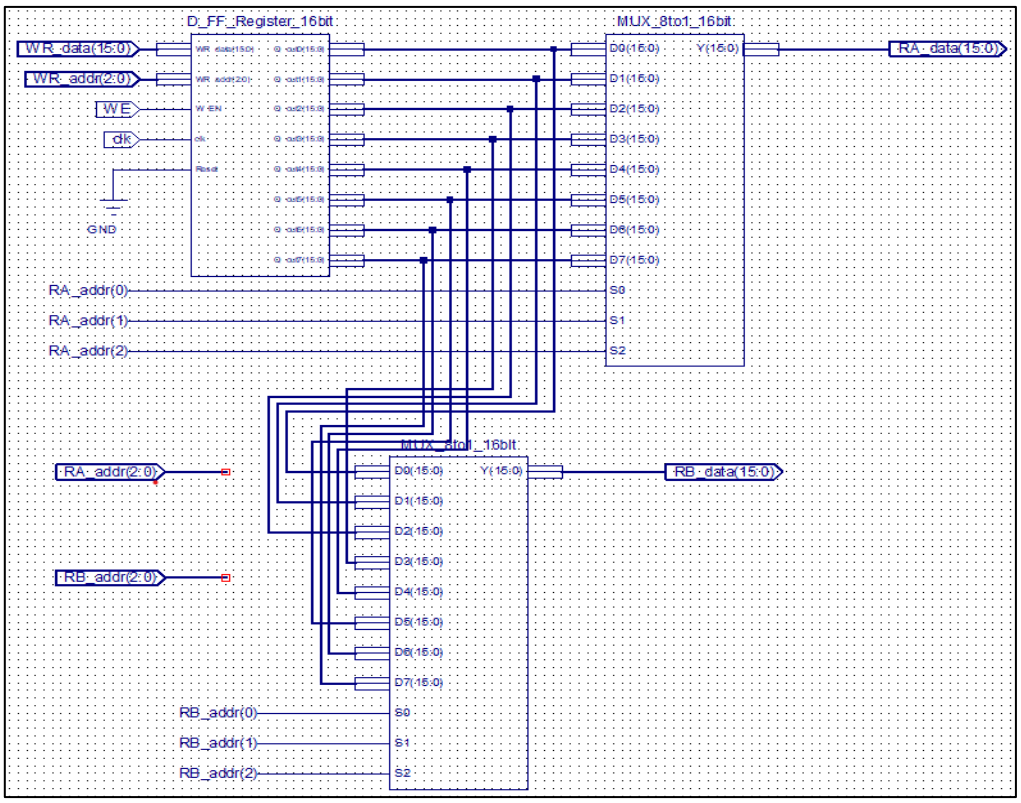
根據我們期中 Schematic 所做的電路將每個 submodule 接線接起來成為 top module，在硬體上實現 divide and conquer 的作法，除了比較好偵錯外，把每個大電路拆分成獨立功能的小電路也有助於理解電路功能。

Schematic Entry

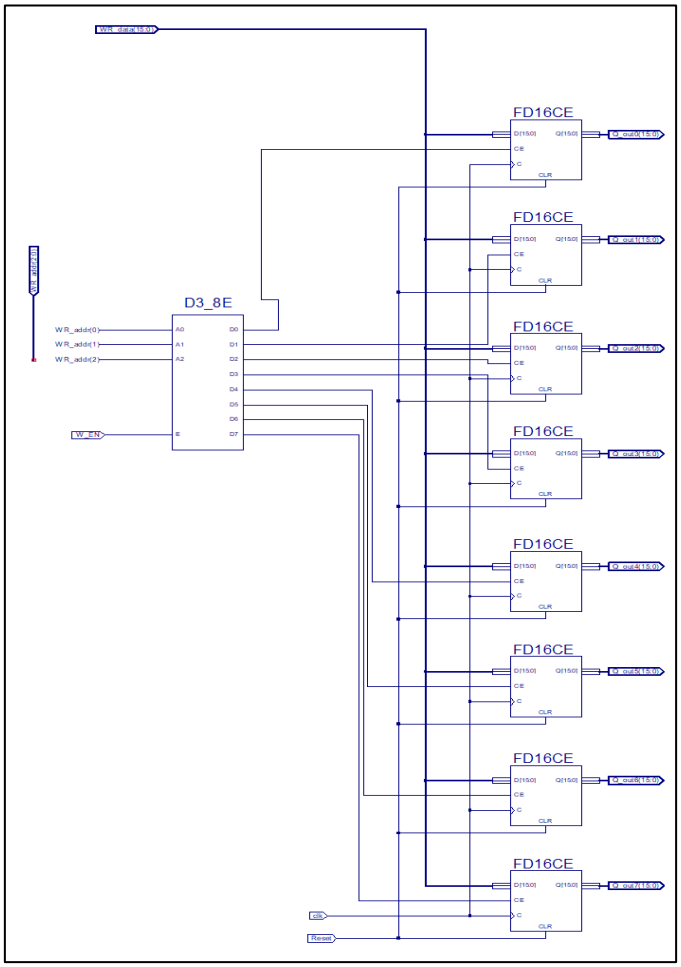
- Registers



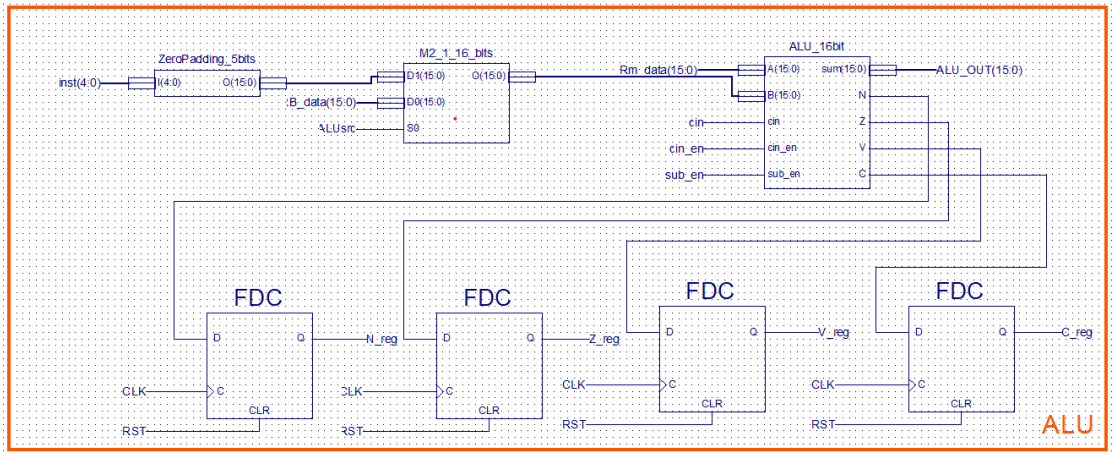
- Eight_Register_16bit



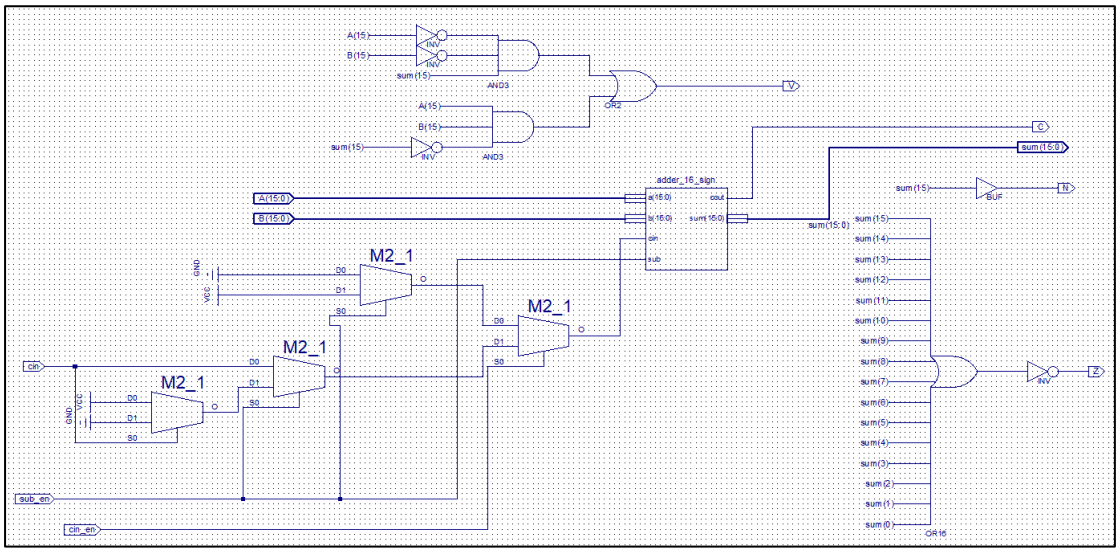
- D_ff_Register_16bit



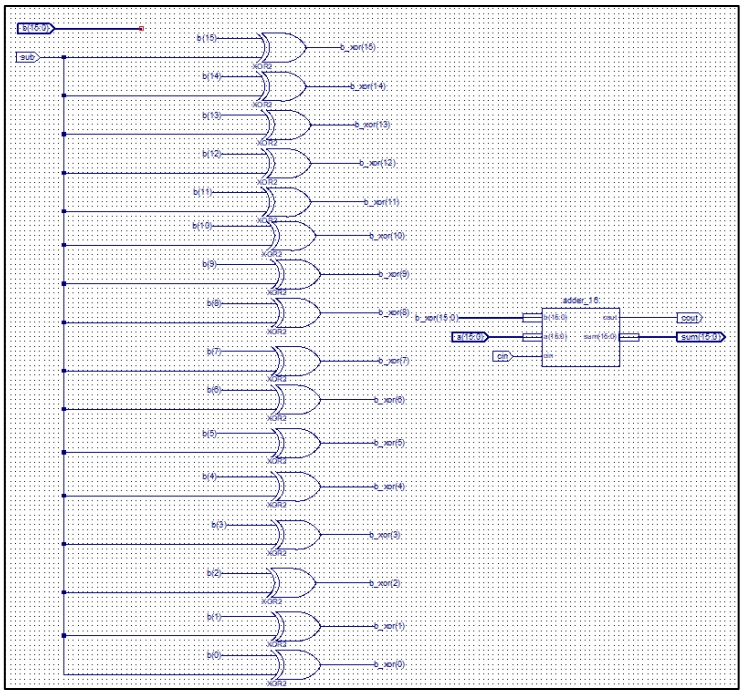
- ALU



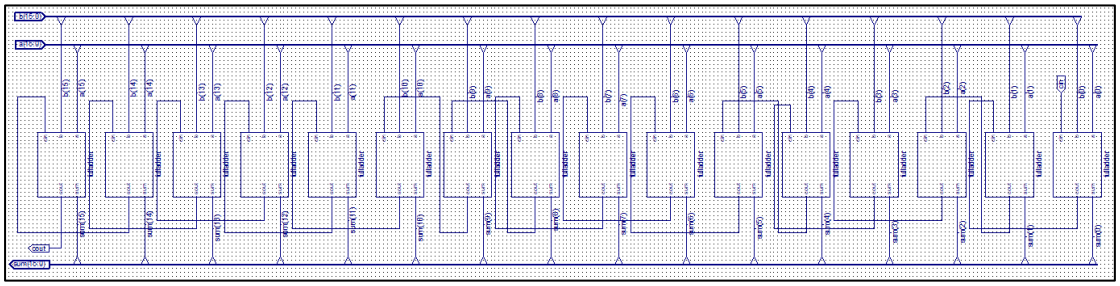
- ALU_16bit



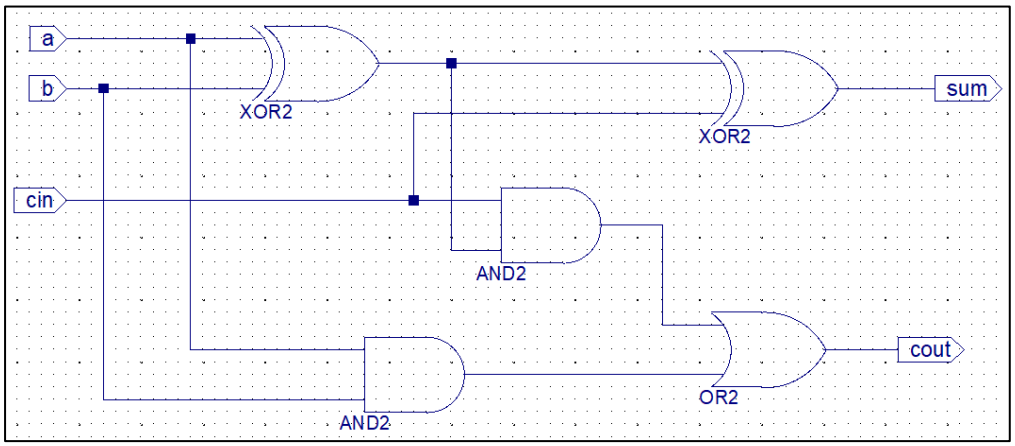
- Adder_16_sign



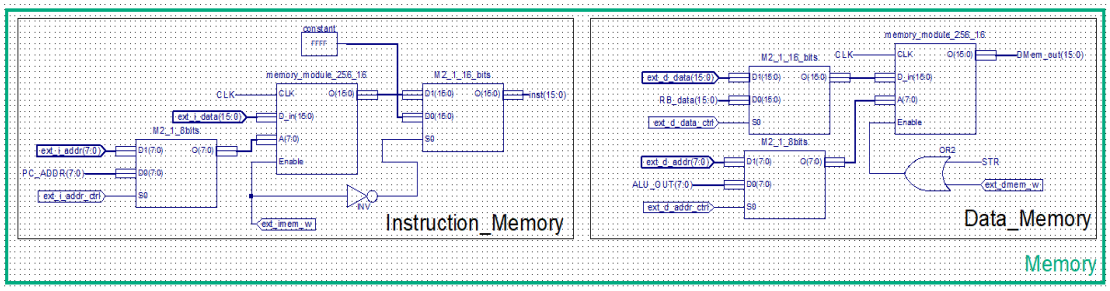
- Adder_16



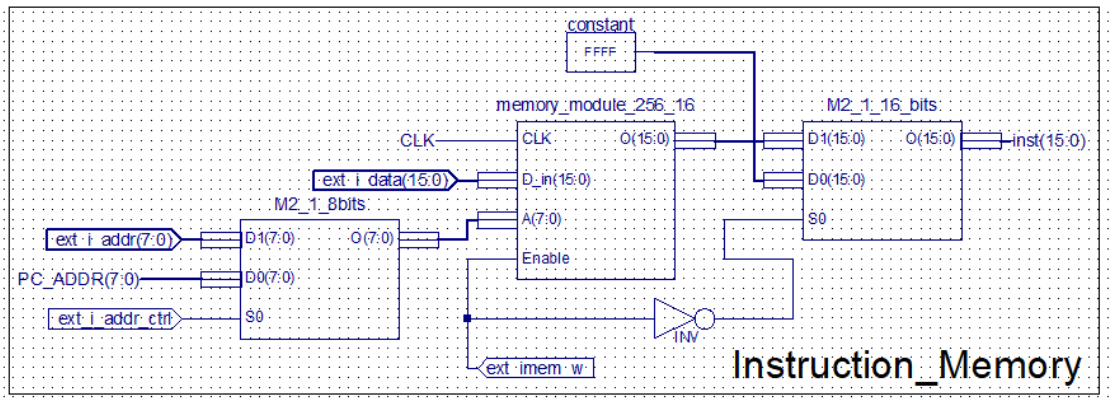
- fulladder



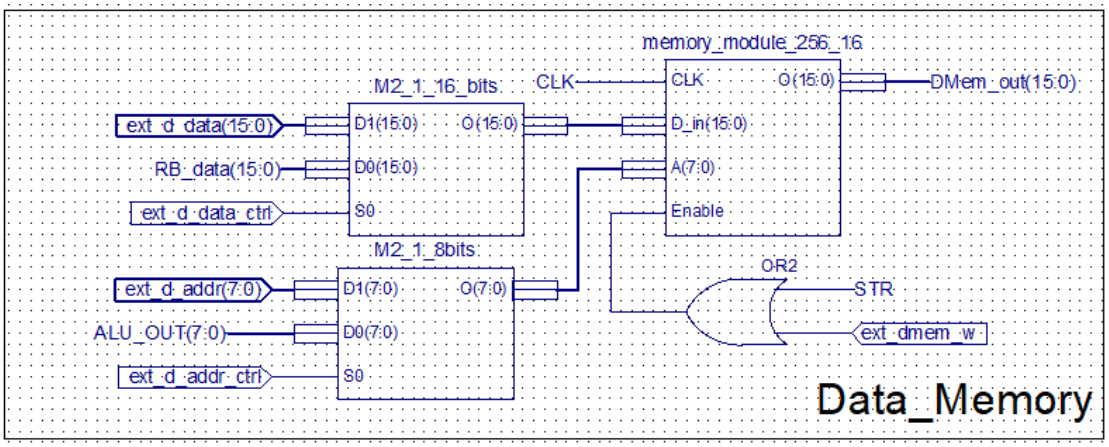
- Memory



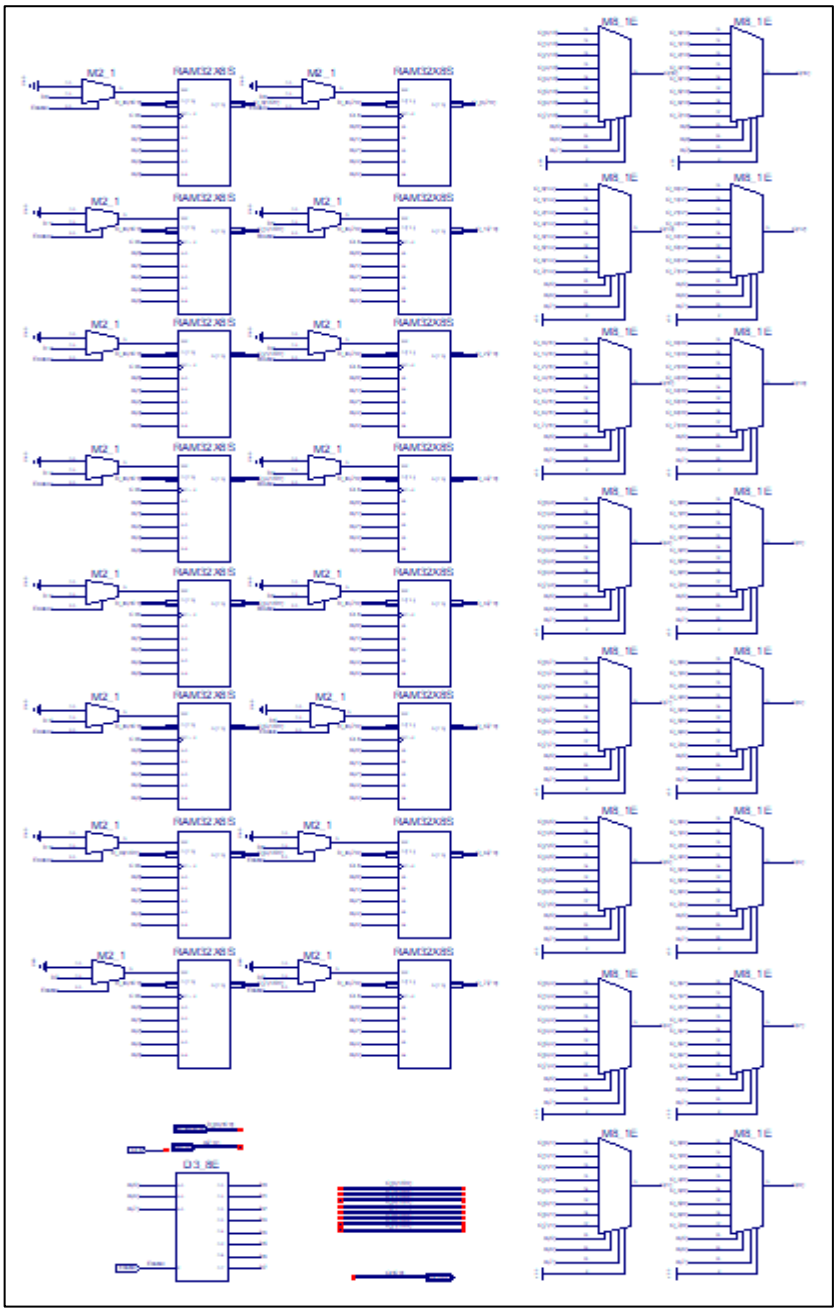
- Instruction_Memory



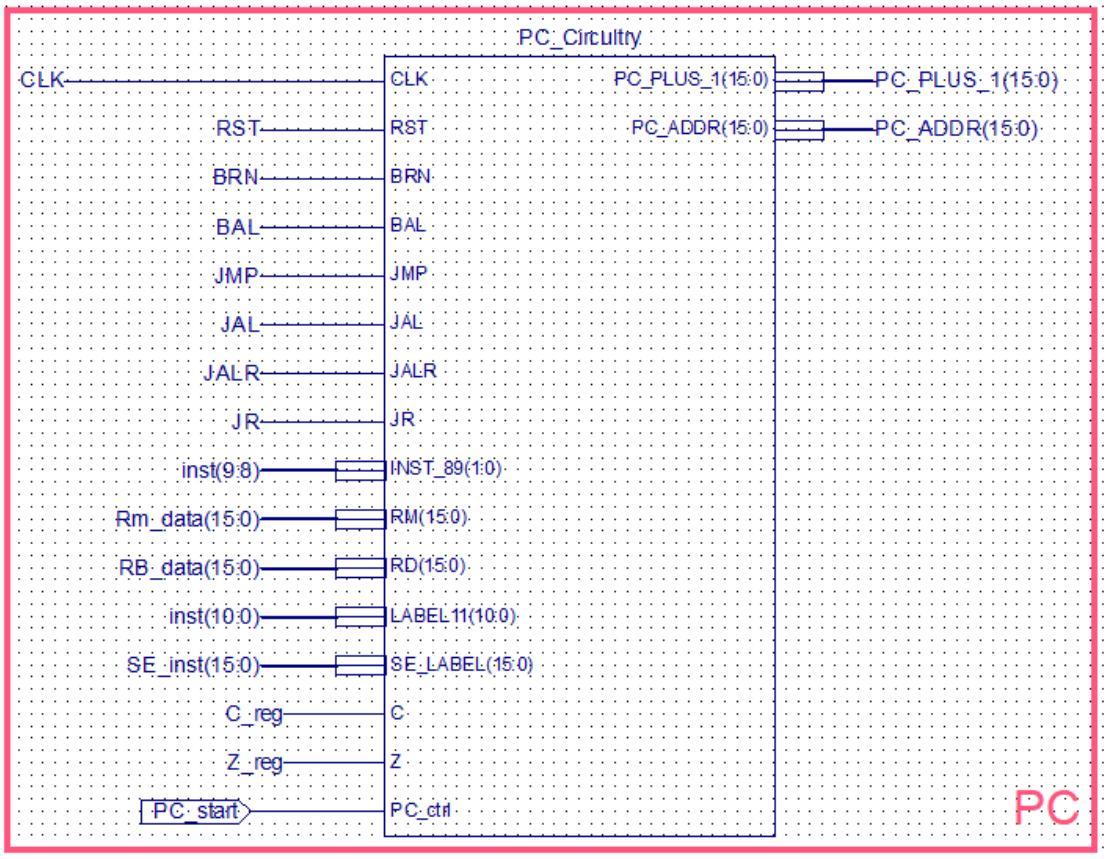
- Data_Memory



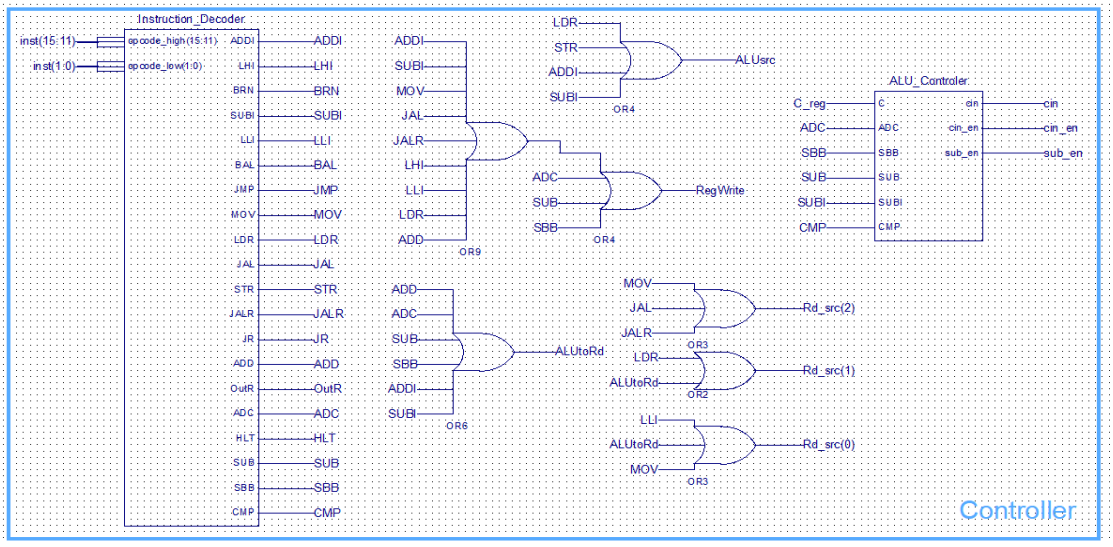
- Memory_module_256_16



- PC



- Controller



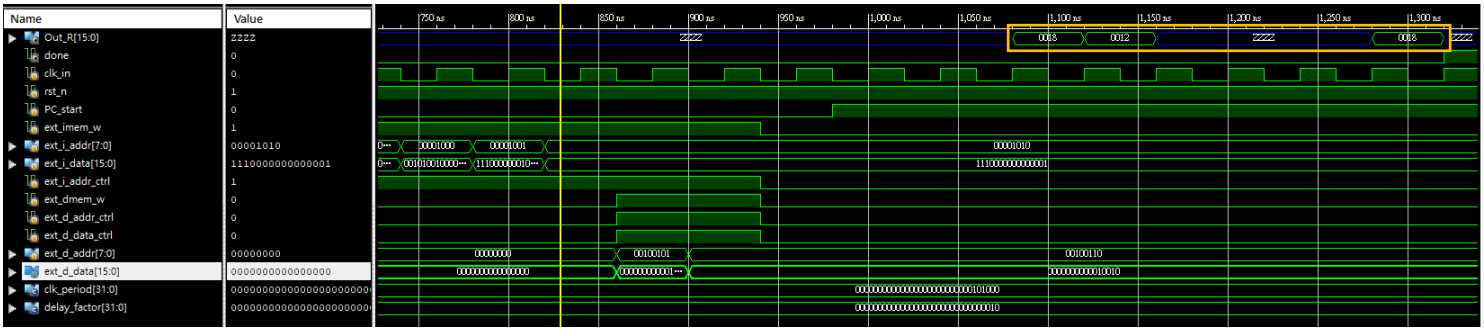
Verification

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

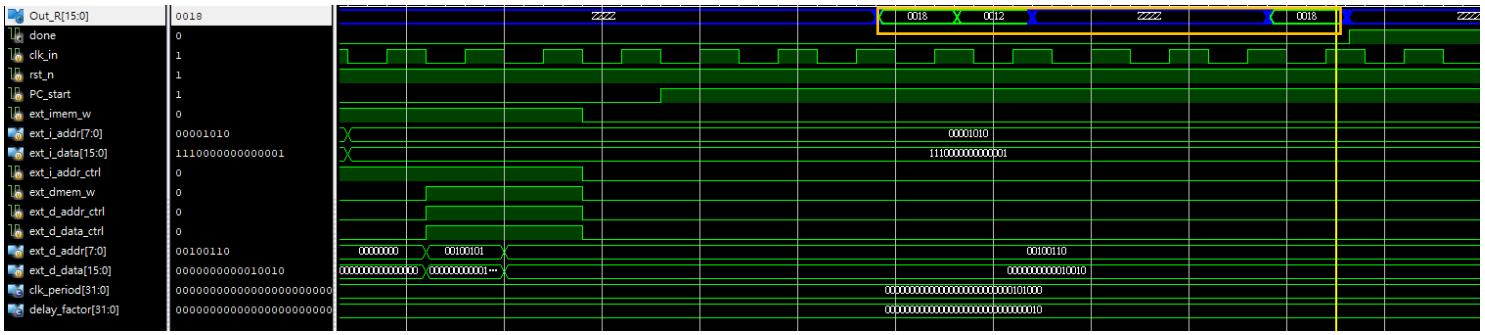
- Assembly code

```
Main:
00    LLI R0, #25
02    LDR R1, R0, #0
04    LDR R2, R0, #1
06    OUTR R1
08    OUTR R2
0A    CMP R1, R2
0C    BCS R1_bigger
0E    MOV R1, R2
R1_bigger:
10  STR R1, R0, #2
12  OUTR R1
14  HLT
```

- Behavioral



- Post-Route



可以觀察到一開始記憶體裡面的值為 12H 和 18H，而在指令執行完後會找到最大值 18H 並輸出。

Testbench

```
`timescale 1ns / 1ps
module max_min_test;
    parameter clk_period = 40;
    parameter delay_factor = 2;
    // Inputs
    reg clk_in;
    reg rst_n;
    reg PC_start;
    reg ext_imem_w;
    reg [7:0] ext_i_addr;
    reg [15:0] ext_i_data;
    reg ext_i_addr_ctrl;
    reg ext_dmem_w;
    reg ext_d_addr_ctrl;
    reg ext_d_data_ctrl;
    reg [7:0] ext_d_addr;
    reg [15:0] ext_d_data;
    // Outputs
    wire [15:0] Out_R;
    wire done;
    // Instantiate the Unit Under Test (UUT)
    Single_Cycle_CPU uut (
        .clk_in(clk_in),
        .rst_n(rst_n),
        .Out_R(Out_R),
        .PC_start(PC_start),
        .done(done),
        .ext_imem_w(ext_imem_w),
        .ext_i_addr(ext_i_addr),
        .ext_i_data(ext_i_data),
        .ext_i_addr_ctrl(ext_i_addr_ctrl),
        .ext_dmem_w(ext_dmem_w),
        .ext_d_addr_ctrl(ext_d_addr_ctrl),
        .ext_d_data_ctrl(ext_d_data_ctrl),
        .ext_d_addr(ext_d_addr),
        .ext_d_data(ext_d_data)
    );
    initial begin
        // Initialize Inputs
        clk_in = 0;
        rst_n = 0;
        PC_start = 0;
        ext_imem_w = 0;
        ext_i_addr = 0;
        ext_i_data = 0;
        ext_i_addr_ctrl = 0;
        ext_dmem_w = 0;
        ext_d_addr_ctrl = 0;
        ext_d_data_ctrl = 0;
        ext_d_addr = 0;
        ext_d_data = 0;

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

        // Add stimulus here
    end
    always begin
        #(clk_period/2) clk_in <= 1'b0;
        #(clk_period/2) clk_in <= 1'b1;
    end
```

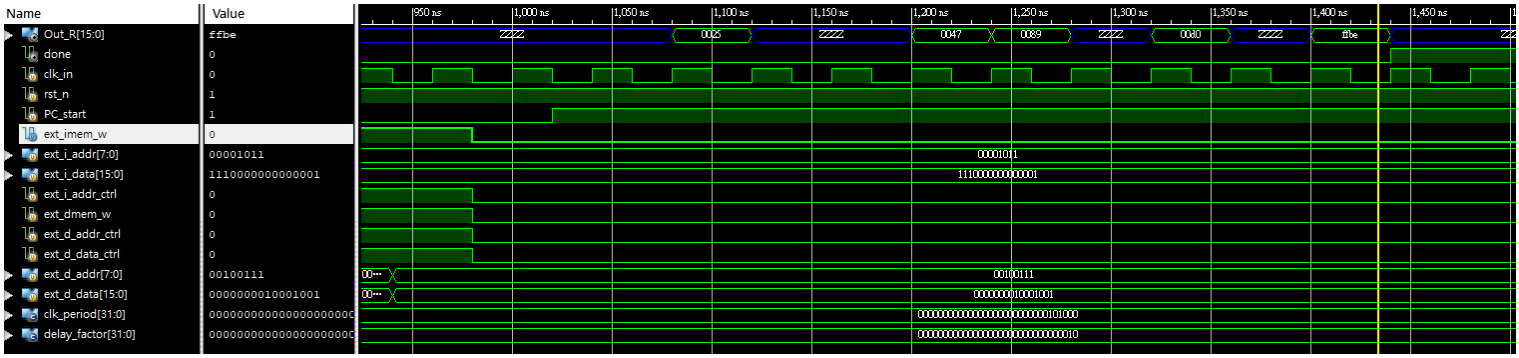
```
    initial begin
        rst_n <=1'b1;
        repeat(9)@(posedge clk_in)
        #(clk_period/delay_factor) rst_n <= 1'b0;
        rst_n <=1'b1;

        write_imem(8'h0, 16'b00010_000_00100101) ; //LLI R0,#25h
        write_imem(8'h1, 16'b00011_001_000_00000) ; //LDR R1, R0, #0
        write_imem(8'h2, 16'b00011_010_000_00001) ; //LDR R2, R0, #1
        write_imem(8'h3, 16'b11100_000_001_000_00) ; //OUTR R1 (20h)
        write_imem(8'h4, 16'b11100_000_010_000_00) ; //OUTR R2 (10h)
        write_imem(8'h5, 16'b00110_000_001_010_01) ; //CMP R1, R2
        write_imem(8'h6, 16'b1100_0010_00000010) ; //BCS R1_bigger
        write_imem(8'h7, 16'b01011_001_010_000_00) ; //MOV R1, R2.
        write_imem(8'h8, 16'b00101_001_000_00010) ; //STR R1, R0, #2
        write_imem(8'h9, 16'b11100_000_001_000_00) ; //OUTR R1 (20h)
        write_imem(8'hA, 16'b11100_0000_00000_01) ; //HLT
        write_dmem(8'h25, 16'h18 ) ; // data (25h, 18h)
        write_dmem(8'h26, 16'h12 ) ; // data (26h, 12h)
        // delay one clock to ensure the proper write to memory
        @(posedge clk_in) #(clk_period/delay_factor) begin
            ext_imem_w = 1'b0;
            ext_dmem_w = 1'b0;
            ext_i_addr_ctrl = 1'b0;
            ext_d_addr_ctrl = 1'b0;
            ext_d_data_ctrl = 1'b0;
        end
        // start the cpu to execute the program in memory
        @(posedge clk_in) #(clk_period/delay_factor) PC_start=1'b1;
        wait(done) ;
    end
    task write_imem;
        input [7:0] addr;
        input [15:0] data;
        begin
            @(posedge clk_in) #(clk_period/delay_factor) begin
                ext_imem_w = 1'b1;
                ext_i_addr_ctrl = 1'b1;
                ext_i_addr = addr;
                ext_i_data = data;
            end
        end
    endtask
    task write_dmem;
        input [7:0] addr;
        input [15:0] data;
        begin
            @(posedge clk_in) #(clk_period/delay_factor) begin
                ext_dmem_w = 1'b1;
                ext_d_addr_ctrl = 1'b1;
                ext_d_addr = addr;
                ext_d_data_ctrl = 1'b1;
                ext_d_data = data;
            end
        end
    endtask

    initial #10000 $finish ;
    initial
        $monitor($realtime,"ns %h %h %h %h %h %h %h %h %h \n",
        clk_in , rst_n, ext_imem_w, ext_dmem_w, ext_i_addr_ctrl, ext_i_addr, ext_i_data, ext_d_

endmodule
```

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

- Assembly code

```
Main:
00      LLI R0, #25
02      LDR R1, R0, #0
04      LDR R2, R0, #1
06      OUTR R1
08      OUTR R2
0A      ADD R3, R1, R2
0C      STR R3, R0, #2
0E      OUTR R3
10      HLT
```

- Behavioral



- Post-Route

把 47H 存在記憶體位址 0025H，0089H 存在記憶體位址 0026H，將兩數相加得到結果 00D0H，相減得到結果 FFBEH。

- Testbench

```
module CPU_testbench;
   parameter clk_period = 40;
   parameter delay_factor = 2;
   // Inputs
   reg clk_in;
   reg rst_n;
   reg PC_start;
   reg ext_imem_w;
   reg [7:0] ext_i_addr;
   reg [15:0] ext_i_data;
   reg ext_i_addr_ctrl;
   reg ext_dmem_w;
   reg ext_d_addr_ctrl;
   reg ext_d_data_ctrl;
   reg [7:0] ext_d_addr;
   reg [15:0] ext_d_data;
   // Outputs
   wire [15:0] Out_R;
   wire done;
   // Instantiate the Unit Under Test (UUT)
   Single_Cycle_CPU uut (
      .clk_in(clk_in),
      .rst_n(rst_n),
      .Out_R(Out_R),
      .PC_start(PC_start),
      .done(done),
      .ext_imem_w(ext_imem_w),
      .ext_i_addr(ext_i_addr),
      .ext_i_data(ext_i_data),
      .ext_i_addr_ctrl(ext_i_addr_ctrl),
      .ext_dmem_w(ext_dmem_w),
      .ext_d_addr_ctrl(ext_d_addr_ctrl),
      .ext_d_data_ctrl(ext_d_data_ctrl),
      .ext_d_addr(ext_d_addr),
      .ext_d_data(ext_d_data)
   );
   initial begin
      // Initialize Inputs
      clk_in = 0;
      rst_n = 0;
      PC_start = 0;
      ext_imem_w = 0;
      ext_i_addr = 0;
      ext_i_data = 0;
      ext_i_addr_ctrl = 0;
      ext_dmem_w = 0;
      ext_d_addr_ctrl = 0;
      ext_d_data_ctrl = 0;
      ext_d_addr = 0;
      ext_d_data = 0;
      // Wait 100 ns for global reset to finish
      #100;
      // Add stimulus here
   end
   always begin
      #(clk_period/2) clk_in <= 1'b0;
      #(clk_period/2) clk_in <= 1'b1;
   end
```

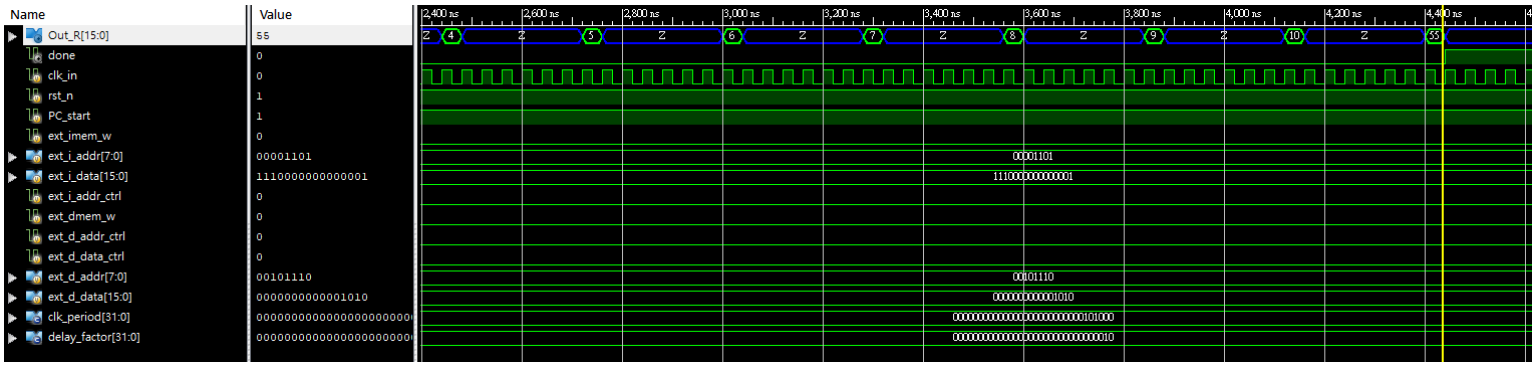
```
   initial begin
      rst_n <=1'b1;
      repeat(9)@(posedge clk_in)
      #(clk_period/delay_factor) rst_n <= 1'b0;
      rst_n <=1'b1;
      write_imem(8'h00, 16'b0001_0000_0010_0101 ) ; // LLI R0,#25
      write_imem(8'h01, 16'b0000_1000_0000_0000 ) ; // LHI R0,#00
      write_imem(8'h02, 16'b1110_0000_0000_0000 ) ; // OUT R0 (0025H)
      write_imem(8'h03, 16'b0001_1001_0000_0000 ) ; // LDR R1,R0,#0
      write_imem(8'h04, 16'b0001_1010_0000_0010 ) ; // LDR R2,R0,#2
      write_imem(8'h05, 16'b1110_0000_0010_0000 ) ; // OUT R1 (47H)
      write_imem(8'h06, 16'b1110_0000_0100_0000 ) ; // OUT R2 (89H)
      write_imem(8'h07, 16'b0000_0011_0010_1000 ) ; // ADD R3,R1,R2
      write_imem(8'h8, 16'b1110_0000_0110_0000 ) ; // OUT R3 (D0H)
      write_imem(8'h9, 16'b0000_0011_0010_1010 ) ; // SUB R3,R1,R2
      write_imem(8'hA, 16'b1110_0000_0110_0000 ) ; // OUT R3 (FFBEH)
      write_imem(8'hB, 16'b1110_0000_0000_0001 ) ; // HLT
      write_dmem(8'h25, 16'h47 ) ; // data (25h, 47h)
      write_dmem(8'h27, 16'h89 ) ; // data (27h, 89h)
      // delay one clock to ensure the proper write to memory
      @(posedge clk_in) #(clk_period/delay_factor) begin
         ext_imem_w = 1'b0;
         ext_dmem_w = 1'b0;
         ext_i_addr_ctrl = 1'b0;
         ext_d_addr_ctrl = 1'b0;
         ext_d_data_ctrl = 1'b0;
      end
      // start the cpu to execute the program in memory
      @(posedge clk_in) #(clk_period/delay_factor) PC_start=1'b1;
      wait(done) ;
   end
   task write_imem;
      input [7:0] addr;
      input [15:0] data;
      begin
         @(posedge clk_in) #(clk_period/delay_factor) begin
            ext_imem_w = 1'b1;
            ext_i_addr_ctrl = 1'b1;
            ext_i_addr = addr;
            ext_i_data = data;
         end
      end
   endtask
   task write_dmem;
      input [7:0] addr;
      input [15:0] data;
      begin
         @(posedge clk_in) #(clk_period/delay_factor) begin
            ext_dmem_w = 1'b1;
            ext_d_addr_ctrl = 1'b1;
            ext_d_addr = addr;
            ext_d_data_ctrl = 1'b1;
            ext_d_data = data;
         end
      end
   endtask
   initial #10000 $finish ;
   initial
      $monitor($realtime,"ns %h %h %h %h %h %h %h %h %h \n",
      clk_in , rst_n, ext_imem_w, ext_dmem_w, ext_i_addr_ctrl, ext_i_addr,ext_i_data,
      ext_d_addr_ctrl, ext_d_addr, ext_d_data_ctrl, ext_d_data, Out_R, done);
endmodule
```

3. Add ten numbers in consecutive memory locations.

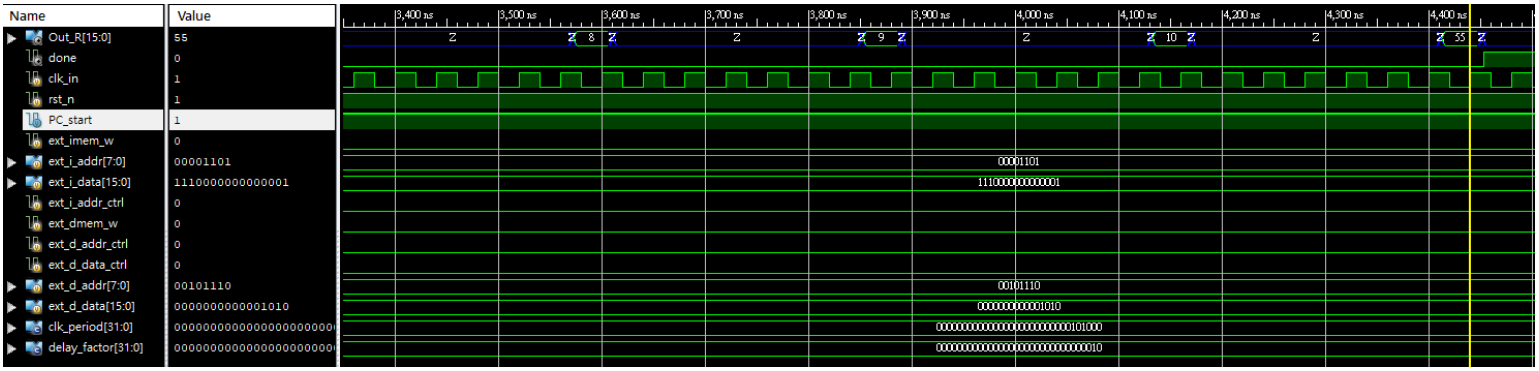
- Assembly code

```
Main:
00      LLI R0, #25
02      LLI R3, #10
04      LLI R4, #1
06      LLI R2, #0
Loop:
08      LDR R1, R0, #0
0A      OUTR R1
0C      ADD R2, R2, R1
0E      ADDI R0, R0, 1

10      SUBI R3, R3, 1
12      CMP R3, R4
14      BCS Loop
16      STR R2, R0, #5
18      OUTR R2
1A      HLT
```

- Behavioral



- Post-Route



存在記憶體的值為 1~10，可以看到結果會將每次運算的值輸出出來，也就是會計算(1+2+3…..9+10)=55，並且得到 55 的正確答案。

Testbench

```
module ten_acc;
   parameter clk_period = 40;
   parameter delay_factor = 2;
   // Inputs
   reg clk_in;
   reg rst_n;
   reg PC_start;
   reg ext_imem_w;
   reg [7:0] ext_i_addr;
   reg [15:0] ext_i_data;
   reg ext_i_addr_ctrl;
   reg ext_dmem_w;
   reg ext_d_addr_ctrl;
   reg ext_d_data_ctrl;
   reg [7:0] ext_d_addr;
   reg [15:0] ext_d_data;
   // Outputs
   wire [15:0] Out_R;
   wire done;
   // Instantiate the Unit Under Test (UUT)
   Single_Cycle_CPU uut (
      .clk_in(clk_in),
      .rst_n(rst_n),
      .Out_R(Out_R),
      .PC_start(PC_start),
      .done(done),
      .ext_imem_w(ext_imem_w),
      .ext_i_addr(ext_i_addr),
      .ext_i_data(ext_i_data),
      .ext_i_addr_ctrl(ext_i_addr_ctrl),
      .ext_dmem_w(ext_dmem_w),
      .ext_d_addr_ctrl(ext_d_addr_ctrl),
      .ext_d_data_ctrl(ext_d_data_ctrl),
      .ext_d_addr(ext_d_addr),
      .ext_d_data(ext_d_data)
   );
   initial begin
      // Initialize Inputs
      clk_in = 0;
      rst_n = 0;
      PC_start = 0;
      ext_imem_w = 0;
      ext_i_addr = 0;
      ext_i_data = 0;
      ext_i_addr_ctrl = 0;
      ext_dmem_w = 0;
      ext_d_addr_ctrl = 0;
      ext_d_data_ctrl = 0;
      ext_d_addr = 0;
      ext_d_data = 0;
      // Wait 100 ns for global reset to finish
      #100;
      // Add stimulus here
   end
   always begin
      #(clk_period/2) clk_in <= 1'b0;
      #(clk_period/2) clk_in <= 1'b1;
   end
```

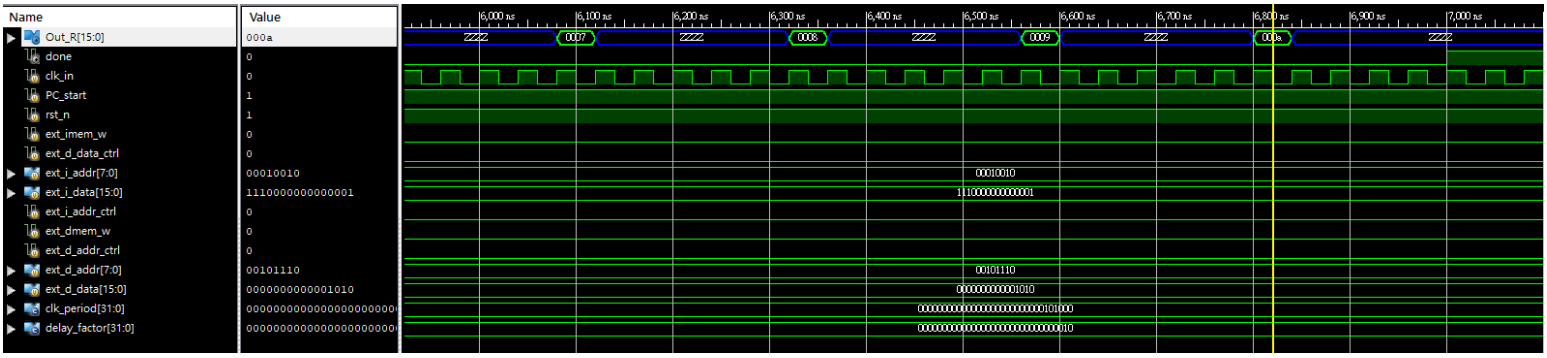
```
    initial begin
        rst_n <=1'b1;
        repeat(9)@(posedge clk_in)
        #(clk_period/delay_factor) rst_n <= 1'b0;
        rst_n <=1'b1;
        write_imem(8'h0, 16'b00010_000_00100101) ; //LLI R0,#25h
        write_imem(8'h1, 16'b00010_011_00001010) ; //LLI R3, #10
        write_imem(8'h2, 16'b00010_100_00000001) ; //LLI R4, #1
        write_imem(8'h3, 16'b00010_010_00000000) ; //LLI R2, #0
        write_imem(8'h4, 16'b00011_001_000_00000) ; //LDR R1, R0, #0
        write_imem(8'h5, 16'b11100_000_001_000_00) ; //OUTR R1 (1,2,3,4,5,6,7,8,9)
        write_imem(8'h6, 16'b00000_010_010_001_00) ; //ADD R2, R2, R1
        write_imem(8'h7, 16'b00111_000_000_00001) ; //ADDI R0, R0, 1
        write_imem(8'h8, 16'b01000_011_011_00001) ; //SUBI R3, R3, 1
        //write_imem(8'h9, 16'b11100_000_011_000_00) ; //OUTR R3 /////
        write_imem(8'h9, 16'b00110_000_011_100_01) ; //CMP R3, R4
        write_imem(8'hA, 16'b1100_0010_11111010) ; //BCS Loop /////
        write_imem(8'hB, 16'b00101_010_000_00101) ; //STR R2, R0, #5
        write_imem(8'hC, 16'b11100_000_010_000_00) ; //OUTR R2 (45)
        write_imem(8'hD, 16'b11100_0000_00000_01) ; //HLT
        write_dmem(8'h25, 16'h1 ) ; // data (25h, 1h)
        write_dmem(8'h26, 16'h2 ) ; // data (26h, 2h)
        write_dmem(8'h27, 16'h3 ) ; // data (27h, 3h)
        write_dmem(8'h28, 16'h4 ) ; // data (28h, 4h)
        write_dmem(8'h29, 16'h5 ) ; // data (29h, 5h)
        write_dmem(8'h2A, 16'h6 ) ; // data (2Ah, 6h)
        write_dmem(8'h2B, 16'h7 ) ; // data (2Bh, 7h)
        write_dmem(8'h2C, 16'h8 ) ; // data (2Ch, 8h)
        write_dmem(8'h2D, 16'h9 ) ; // data (2Dh, 9h)
        write_dmem(8'h2E, 16'hA ) ; // data (2Eh, Ah)
        // delay one clock to ensure the proper write to memory
        @(posedge clk_in) #(clk_period/delay_factor) begin
            ext_imem_w = 1'b0;
            ext_dmem_w = 1'b0;

            ext_i_addr_ctrl = 1'b0;
            ext_d_addr_ctrl = 1'b0;
            ext_d_data_ctrl = 1'b0;
        end
        // start the cpu to execute the program in memory
        @(posedge clk_in) #(clk_period/delay_factor) PC_start=1'b1;
        wait(done) ;
    end
    task write_imem;
        input [7:0] addr;
        input [15:0] data;
        begin
            @(posedge clk_in) #(clk_period/delay_factor) begin
                ext_imem_w = 1'b1;
                ext_i_addr_ctrl = 1'b1;
                ext_i_addr = addr;
                ext_i_data = data;
            end
        end
    endtask
    task write_dmem;
        input [7:0] addr;
        input [15:0] data;
        begin
            @(posedge clk_in) #(clk_period/delay_factor) begin
                ext_dmem_w = 1'b1;
                ext_d_addr_ctrl = 1'b1;
                ext_d_addr = addr;
                ext_d_data_ctrl = 1'b1;
                ext_d_data = data;
            end
        end
    endtask
    initial #10000 $finish ;
    initial
        $monitor($realtime,"ns %h %h %h %h %h %h %h %h %h \n",
        clk_in , rst_n, ext_imem_w, ext_dmem_w, ext_i_addr_ctrl, ext_i_addr, ext_i_data,
        ext_d_addr_ctrl, ext_d_addr, ext_d_data_ctrl, ext_d_data, Out_R, done);

endmodule
```

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

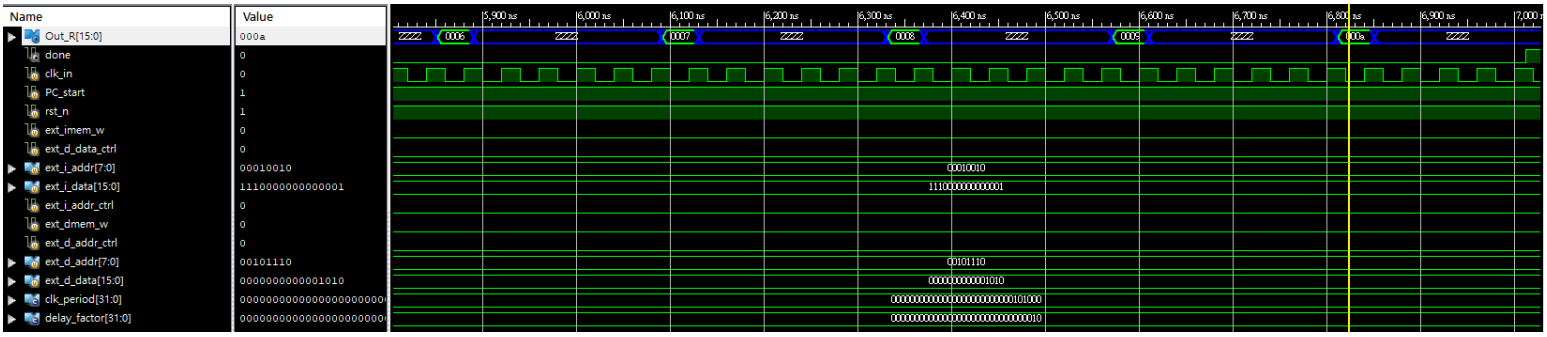
- Assembly code

```
Main:
00      LLI R0, #25h
02      LLI R2, #10
04      LLI R3, #1
06      LLI R4, #39
Loop:
08      LDR R1, R0, #0
0A      OUTR R1
0C      STR R1, R0, #30
0E      ADDI R0, R0, #1
10      SUBI R2, R2, #1
12      CMP R2, R3
14      BCS Loop
16      LLI R2, #10
Loop_check:
18      LDR R1, R4, #0
1A      OUTR R1
1C      ADDI R4, R4, #1
1E      SUBI R2, R2, #1
20      CMP R2, R3
22      BCS Loop_check
24      HLT
```

- Behavioral



- Post-Route



可以看到最後的輸出是 8…9….10，因為我們是將 memory 裡面的 1~10，搬到另一個記憶體位址，我們再搬完後去查看記憶體的位址裡面是否為 1~10，可以發現 move 成功。

- Testbench

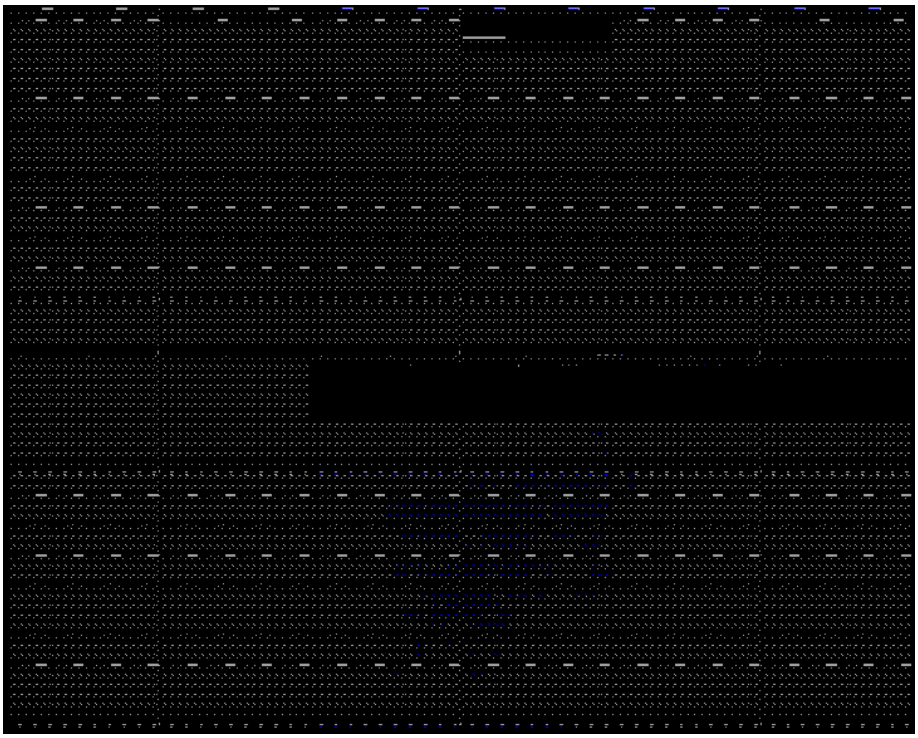
```
module mov_10;
    parameter clk_period = 40;
    parameter delay_factor = 2;
    // Inputs
    reg clk_in;
    reg PC_start;
    reg rst_n;
    reg ext_imem_w;
    reg ext_d_data_ctrl;
    reg [7:0] ext_i_addr;
    reg [15:0] ext_i_data;
    reg   ext_i_addr_ctrl;
    reg ext_dmem_w;
    reg ext_d_addr_ctrl;
    reg [7:0] ext_d_addr;
    reg [15:0] ext_d_data;
    // Outputs
    wire [15:0] Out_R;
    wire done;
    // Instantiate the Unit Under Test (UUT)
    Single_Cycle_CPU uut (
        .clk_in(clk_in),
        .rst_n(rst_n),
        .Out_R(Out_R),
        .done(done),
        .ext_imem_w(ext_imem_w),
        .PC_start(PC_start),
        .ext_i_addr(ext_i_addr),
        .ext_i_data(ext_i_data),
        .ext_d_data_ctrl(ext_d_data_ctrl),
        .ext_i_addr_ctrl(ext_i_addr_ctrl),
        .ext_d_addr_ctrl(ext_d_addr_ctrl),
        .ext_dmem_w(ext_dmem_w),
        .ext_d_addr(ext_d_addr),
        .ext_d_data(ext_d_data)
    );
    initial begin
        // Initialize Inputs
        clk_in = 0;
        rst_n = 1;
        ext_imem_w = 0;
        ext_i_addr = 0;
        ext_i_data = 0;
        ext_dmem_w = 0;
        ext_d_addr = 0;
        ext_d_data = 0;

        // Wait 100 ns for global reset to finish
        #100;
        // Add stimulus here
    end
    // generate the clock signal
    always begin
        #(clk_period/2) clk_in <= 1'b0;
        #(clk_period/2) clk_in <= 1'b1;
    end
```

```
  initial begin
  rst_n <=1'b0;
  repeat(9)@(posedge clk_in)
  #(clk_period/delay_factor) rst_n <= 1'b0;PC_start <= 1'b0;
  #(clk_period/delay_factor) rst_n <=1'b1;
  write_imem(8'h0, 16'b00010_000_00100101) ; //LLI R0,#25h
  write_imem(8'h1, 16'b00010_010_00001010) ; //LLI R2, #10
  write_imem(8'h2, 16'b00010_011_00000001) ; //LLI R3, #1
  write_imem(8'h3, 16'b00010_100_00111001) ; //LLI R4, #39h
  write_imem(8'h4, 16'b00011_001_000_00000) ; //LDR R1, R0, #0
  write_imem(8'h5, 16'b11100_000_001_000_00) ; //OUTR R1 (1,2,3,4,5,6,7,8,9,10)
  write_imem(8'h6, 16'b00101_001_000_10100) ; //STR R1, R0, #20
  write_imem(8'h7, 16'b00111_000_000_00001) ; //ADDI R0, R0, 1
  write_imem(8'h8, 16'b01000_010_010_00001) ; //SUBI R2, R2, 1
  write_imem(8'h9, 16'b00110_000_010_011_01) ; //CMP R2, R3
  write_imem(8'hA, 16'b1100_0010_11111010) ; //BCS Loop
  write_imem(8'hB, 16'b00010_010_00001010) ; //LLI R2, #10
  write_imem(8'hC, 16'b00011_001_100_00000) ; //LDR R1, R4, #0
  write_imem(8'hD, 16'b11100_000_001_000_00) ; //OUTR R1 (1,2,3,4,5,6,7,8,9,10)
  write_imem(8'hE, 16'b00111_100_100_00001) ; //ADDI R4, R4, #1
  write_imem(8'hF, 16'b01000_010_010_00001) ; //SUBI R2, R2, #1
  write_imem(8'h10, 16'b00110_000_010_011_01) ; //CMP R2, R3
  write_imem(8'h11, 16'b1100_0010_11111011) ; //BCS Loop_check
  write_imem(8'h12, 16'b11100_0000_00000_01) ; //HLT
  write_dmem(8'h25, 16'h1 ) ; // data (25h, 1h)
  write_dmem(8'h26, 16'h2 ) ; // data (26h, 2h)
  write_dmem(8'h27, 16'h3 ) ; // data (27h, 3h)
  write_dmem(8'h28, 16'h4 ) ; // data (28h, 4h)
  write_dmem(8'h29, 16'h5 ) ; // data (29h, 5h)
  write_dmem(8'h2A, 16'h6 ) ; // data (2Ah, 6h)
  write_dmem(8'h2B, 16'h7 ) ; // data (2Bh, 7h)
  write_dmem(8'h2C, 16'h8 ) ; // data (2Ch, 8h)
  write_dmem(8'h2D, 16'h9 ) ; // data (2Dh, 9h)
  write_dmem(8'h2E, 16'hA ) ; // data (2Eh, Ah)

 // delay one clock to ensure the proper write to memory
 @(posedge clk_in) #(clk_period/delay_factor) begin
  ext_imem_w = 1'b0;
  ext_dmem_w = 1'b0;
  ext_i_addr_ctrl = 1'b0;
  ext_d_addr_ctrl = 1'b0;
  ext_d_data_ctrl = 1'b0;
 end
 // start the cpu to execute the program in memory
 @(posedge clk_in) #(clk_period/delay_factor) PC_start=1'b1;
 wait(done) ;
end
  task write_imem;
     input [7:0] addr;
     input [15:0] data;
     begin
        @(posedge clk_in) #(clk_period/delay_factor) begin
           ext_imem_w = 1'b1;
           ext_i_addr_ctrl = 1'b1;
           ext_i_addr = addr;
           ext_i_data = data;
        end
     end
  endtask
  task write_dmem;
     input [7:0] addr;
     input [15:0] data;
     begin
        @(posedge clk_in) #(clk_period/delay_factor) begin
           ext_dmem_w = 1'b1;
           ext_d_addr_ctrl = 1'b1;
           ext_d_addr = addr;
           ext_d_data_ctrl = 1'b1;
           ext_d_data = data;
        end
     end
  endtask
  initial #10000 $finish ;
  initial
     $monitor($realtime,"ns %h %h %h %h %h %h %h %h %h \n",
     clk_in , rst_n, ext_imem_w, ext_dmem_w, ext_i_addr_ctrl, ext_i_addr, ext_i_data,
     ext_d_addr_ctrl, ext_d_addr, ext_d_data_ctrl, ext_d_data, Out_R, done);
endmodule
```

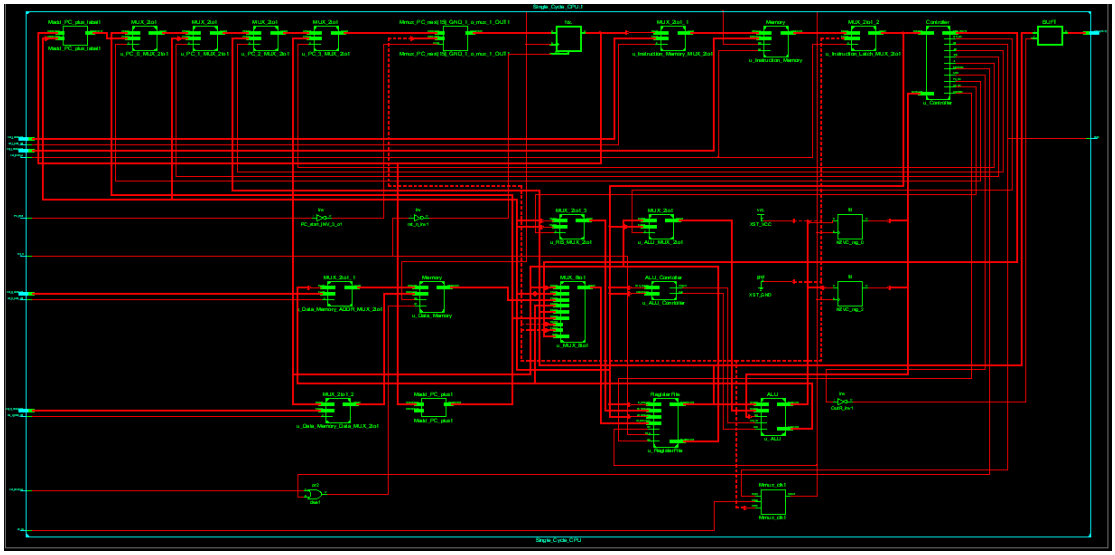
Hardware

```
Device Utilization Summary:

Slice Logic Utilization:
  Number of Slice Registers:                    148 out of  93,120    1%
    Number used as Flip Flops:                  146
    Number used as Latches:                       0
    Number used as Latch-thrus:                   0
    Number used as AND/OR logics:                 2
  Number of Slice LUTs:                         889 out of  46,560    1%
    Number used as logic:                       754 out of  46,560    1%
      Number using O6 output only:              661
      Number using O5 output only:               29
      Number using O5 and O6:                    64
      Number used as ROM:                         0
    Number used as Memory:                      132 out of  16,720    1%
      Number used as Dual Port RAM:               0
      Number used as Single Port RAM:           132
        Number using O6 output only:            132
        Number using O5 output only:              0
        Number using O5 and O6:                   0
      Number used as Shift Register:              0
    Number used exclusively as route-thrus:       3
      Number with same-slice register load:       0
      Number with same-slice carry load:          3
      Number with other load:                     0

Slice Logic Distribution:
  Number of occupied Slices:                    343 out of  11,640    2%
  Number of LUT Flip Flop pairs used:           889
    Number with an unused Flip Flop:            741 out of     889   83%
    Number with an unused LUT:                    0 out of     889    0%
    Number of fully used LUT-FF pairs:          148 out of     889   16%
    Number of slice register sites lost
      to control set restrictions:                0 out of  93,120    0%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  OVERMAPPING of BRAM resources should be ignored if the design is
  over-mapped for a non-BRAM resource or if placement fails.

IO Utilization:
  Number of bonded IOBs:                         73 out of     240   30%

Specific Feature Utilization:
  Number of RAMB36E1/FIFO36E1s:                   0 out of     156    0%
  Number of RAMB18E1/FIFO18E1s:                   0 out of     312    0%
  Number of BUFG/BUFGCTRLs:                       1 out of      32    3%
    Number used as BUFGs:                         1
    Number used as BUFGCTRLs:                     0
  Number of ILOGICE1/ISERDESE1s:                  0 out of     360    0%
  Number of OLOGICE1/OSERDESE1s:                  0 out of     360    0%
  Number of BSCANs:                               0 out of       4    0%
  Number of BUFHCEs:                              0 out of      72    0%
  Number of BUFIODQSs:                            0 out of      36    0%
  Number of BUFRs:                                0 out of      18    0%
  Number of CAPTUREs:                             0 out of       1    0%
  Number of DSP48E1s:                             0 out of     288    0%
  Number of EFUSE_USRs:                           0 out of       1    0%
  Number of FRAME_ECCs:                           0 out of       1    0%
  Number of GTXE1s:                               0 out of       8    0%
  Number of IBUFDS_GTXE1s:                        0 out of       6    0%
  Number of ICAPs:                                0 out of       2    0%
  Number of IDELAYCTRLs:                          0 out of       9    0%
  Number of IODELAYE1s:                           0 out of     360    0%
  Number of MMCM_ADVs:                            0 out of       6    0%
  Number of PCIE_2_0s:                            0 out of       1    0%
  Number of STARTUPs:                             1 out of       1  100%
  Number of SYSMONs:                              0 out of       1    0%
  Number of TEMAC_SINGLEs:                        0 out of       4    0%
```

FPGA VIEW



RTL VIEW:



Discussion

這次的作業就是根據期中的 Schematic 電路，使用 verilog 硬體描述語言來實現，而由於期中的使用已經做過 Schematic 的版本，因此實作起來並沒有想像中困難，反而覺得 Schematic 電路在設計上想得比較多，這可能就是銘波老師上課常說的 Hardware mind 也就是因為要有硬體思維，寫 verilog 時心中必須有那個電路的架構，而不是使用平常寫 c 語言等等軟體思維再進行，用軟體思維在實作的話除了可能會寫出無法合成的電路外，時常自己可能也不知道合成出來的電路是什麼。

因此很感謝老師這一學期的教導，除了讓我學會很多作為數位 IC 工程師該有的知識與技能外，也培養了我解決問題的能力與該具備的硬體思維，老師上課所使用的課本也寫了許多很重要的概念，可能到未來面試或工作都還會使用到，希望未來能夠繼續在此精進自己。