

Part1:

a.32-bit ALU

```
module ALU(  
    input [31:0] Source1, //Register1 input  
    input [31:0] Source2, //Register2 input  
    input [5:0] funct, //R type function code  
    input [4:0] shamt, //Shift amount  
    output reg [31:0] result=32'd0, //Result output  
    output zero, //Zero flag  
    output reg carry=0 //Carry flag  
);  
  
    assign zero=(result==0)?1:0;//If the result is zero,the zero flag is 1  
  
    always@(Source1 or Source2 or funct or shamt)begin  
        case(funct[5:0])//Identify function code  
            6'd27: {carry,result}<=Source1+Source2;//Function ADD  
            6'd28: result<=Source1-Source2;//Function SUB  
            6'd29: result<=Source1>>shamt;//Function SRL  
            6'd30: result<=Source1<<shamt;//Function SLL  
            6'd31: result<=Source1^Source2;//Function XOR  
            6'd32: result<=Source1&Source2;//Function AND  
            default: result<=result;//If function code no match,maintain the result  
        endcase  
    end  
endmodule
```

Part1 的 ALU 非常的簡單，只要照著老師給的接線，照著步驟打上去，基本上就完成 9 成了，剩下的就是要考慮 **flag** 的問題而已，而我選擇用 **assign** 的方式去做 Zero Flag，然後 Carry Flag 是用 **reg** 的方式直接寫在 ADD 的 function 裡。而我的 ALU 在正確的 function 中會做它該做的事，但只要非這 6 個 function 輸入 ALU，它只會維持之前的結果，不會改變。

Testbench 的部分我測試的方式是按照 function 的方式去測試，首先去測試它的 carry 是否正常，再來測試 ADD 是否正常，第三個測試 SUB 和 zero 是否正常，再來測試左移右移(十進位的乘除 2or 乘除 4)，再來測試 XOR 和 AND 的 function，最後我將 funct 的值設 0，觀察輸出並無改變，ALU 測試就告一段落

b. 32-bit Register File (Read Only)

```
module RF(  
    input [4:0] Address1, //Source 1 address  
    input [4:0] Address2, //Source 2 address  
    output [31:0] Source1, //Source 1 value  
    output [31:0] Source2 //Source 2 value  
);  
    wire [31:0] register[31:0]; //Creat 32 registers  
  
    assign register[0]=32'd21; //Number0  
    assign register[1]=32'd444; //Number1  
    assign register[2]=32'd178; //Number2  
    assign register[3]=32'd365; //Number3  
    assign register[4]=32'd33; //Number4  
    assign register[5]=32'd89; //Number5  
    assign register[6]=32'd49; //Number6  
    assign register[7]=32'd11; //Number7  
    assign register[8]=32'd347; //Number8  
    assign register[9]=32'd44; //Number9  
    assign register[10]=32'd1000; //Number10  
    assign register[11]=32'd2000; //Number11  
    assign register[12]=32'd71; //Number12  
    assign register[13]=32'd38; //Number13  
    assign register[14]=32'd19; //Number14  
    assign register[15]=32'd51; //Number15  
    assign register[16]=32'd663; //Number16  
    assign register[17]=32'd1871; //Number17  
    assign register[18]=32'd364; //Number18  
    assign register[19]=32'd1110; //Number19  
    assign register[20]=32'd197; //Number20  
    assign register[21]=32'd180; //Number21  
    assign register[22]=32'd1; //Number22  
    assign register[23]=32'd619; //Number23  
    assign register[24]=32'd42; //Number24  
  
    assign register[25]=32'd43; //Number25  
    assign register[26]=32'd831; //Number26  
    assign register[27]=32'd39; //Number27  
    assign register[28]=32'd734; //Number28  
    assign register[29]=32'd92; //Number29  
    assign register[30]=32'd3456; //Number30  
    assign register[31]=32'd1234; //Number31  
  
    assign Source1=register[Address1]; //Put the register value which address is address1 in the Source1  
    assign Source2=register[Address2]; //Put the register value which address is address2 in the Source2  
endmodule
```

Register File 的做法也是非常的簡單，只要先把老師規定的 32 個暫存器的值，慢慢地依序打進檔案裡，剩下的就是利用 assign address 的方式去輸出結果即可

Wave - Default		Msig0																			
0	/b_RF/RF/Address1	0		2	4	6	8	10	12	14	16	18	20	22	24	26	28	30			
1	/b_RF/RF/Address2	0	0	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31		
2	/b_RF/RF/Source1	21			178	33	49	347	1000	71	19	663	364	197	1	42	831	734	3456		
21	/b_RF/RF/Source2	21	444	365	89	11	44	2000	38	51	1871	1110	180	619	43	39	92	1234			

Wave - Default		Msig0																			
0	/b_RF/RF/Address1	0		2	4	6	8	10	12	14	16	18	20	22	24	26	28	30			
1	/b_RF/RF/Address2	0	0	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31		
2	/b_RF/RF/Source1	21			178	33	49	347	1000	71	19	663	364	197	1	42	831	734	3456		
21	/b_RF/RF/Source2	21	444	365	89	11	44	2000	38	51	1871	1110	180	619	43	39	92	1234			

c. Complete ALU:

```
module CompALU(
    input [31:0] Instruction, //Instruction input
    output [31:0] result, //Arithmetic result
    output zero, //Zero flag
    output carry //Carry flag
);
    wire [31:0] Src1; //The wire of Source1
    wire [31:0] Src2; //The wire of Source2

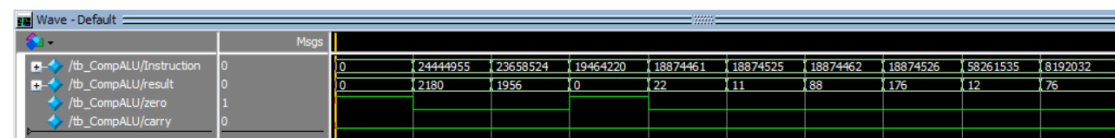
    RF RF(Instruction[25:21], Instruction[20:16], Src1, Src2); //The module of register file
    ALU uut(Src1, Src2, Instruction[5:0], Instruction[10:6], result, zero, carry); //The module of ALU
endmodule
```

Part1 的最後就是將以上兩個做出來的東西結合在一起，利用輸入指令的方式，做出一個可以接收指令去做去不同事情的 ALU+Register，在這個部分要注意的是 Instruction 的不同位置代表不同的事情，所以接線的時候要想清楚，不然輸入到 ALU 或是 Register File 裡就會出錯

```
`timescale 1ns/1ns
module tb_CompALU;
    reg [31:0] Instruction; //The signal of instruction
    wire [31:0] result; //The signal of result
    wire zero; //The signal of zero flag
    wire carry; //The signal of carry flag

    CompALU CPU(Instruction, result, zero, carry); //The module of CompALU

    initial begin //Simulation begin
        Instruction=32'b000000_00000_00000_00000_000000;
        #10 Instruction=32'b000000_01011_10101_00000_00000_011011; //Number11 register(2000) ADD Number21 register(180)
        #10 Instruction=32'b000000_01011_01001_00000_00000_011100; //Number11 register(2000) SUB Number9 register(44)
        #10 Instruction=32'b000000_01001_01001_00000_00000_011100; //Number9 register(44) SUB Number9 register(44)
        #10 Instruction=32'b000000_01001_00000_00000_00001_011101; //Number9 register(44) SRL lbit
        #10 Instruction=32'b000000_01001_00000_00000_00010_011101; //Number9 register(44) SRL 2bits
        #10 Instruction=32'b000000_01001_00000_00000_00001_011110; //Number9 register(44) SLL lbit
        #10 Instruction=32'b000000_01001_00000_00000_00010_011110; //Number9 register(44) SLL 2bits
        #10 Instruction=32'b000000_11011_11001_00000_00000_011111; //Number27 register(39) XOR Number25 register(43)
        #10 Instruction=32'b000000_00011_11101_00000_00000_100000; //Number3 register(365) AND Number29 register(92)
    end
endmodule
```



Signal	0	1	2	3	4	5	6	7	8	9
/tb_CompALU/Instruction	0	24444955	23658524	19464220	18874461	18874525	18874462	18874526	58261535	8192032
/tb_CompALU/result	0	2180	1956	0	22	11	88	176	12	76
/tb_CompALU/zero	1									
/tb_CompALU/carry	0									

而我測試的方式很簡單，就是隨機的選暫存器做 function，只有做 SUB 那裡有特別選出兩個一樣的值去執行，因為要觀察 Zero 是否正常工作，但 carry 的部分因為 Register File 裡並沒有夠大的值讓我去檢查，因此跳過那個部分

Part2:

a.32-bit ALU

```
module ALU(  
    input [31:0] Source1, //Register1 input  
    input [31:0] Source2, //Register2 input  
    input [5:0] funct, //R type function code  
    input [4:0] shamt, //Shift amount  
    output reg [31:0] result=32'd0, //Result output  
    output zero, //Zero flag  
    output reg carry=0 //Carry flag  
);  
  
    assign zero=(result==0)?1:0;//If the result is zero,the zero flag is 1  
  
    always@(Source1 or Source2 or funct or shamt)begin  
        case(funct[5:0])//Identify function code  
            6'd27: {carry,result}<=Source1+Source2;//Function ADD  
            6'd28: result<=Source1-Source2;//Function SUB  
            6'd29: result<=Source1>>shamt;//Function SRL  
            6'd30: result<=Source1<<shamt;//Function SLL  
            6'd31: result<=Source1^Source2;//Function XOR  
            6'd32: result<=Source1&Source2;//Function AND  
            default: result<=result;//If function code no match,maintain the result  
        endcase  
    end  
endmodule
```

因為這個部分是利用 Part1 的 ALU 繼續使用，所以這部分的說明就跳過，
以下附上 testbench

```
`timescale 1ns/1ns  
module tb_ALU();  
  
    reg [31:0]Src1;//The signal of Source1  
    reg [31:0]Src2;//The signal of Source2  
    reg [5:0]funct;//The signal of function code  
    reg [4:0]shamt;//The signal of shift amount  
    wire [31:0]result;//The signal of result  
    wire zero;//The signal of zero flag  
    wire carry;//The signal of carry flag  
  
    ALU uut(Src1,Src2,funct,shamt,result,zero,carry);//The module of ALU  
  
    initial begin//simulation begin  
        Src1=0;  
        Src2=0;  
        funct=0;  
        shamt=0;  
  
        #20    funct=6'd27;  
        #10    Src1=32'b10000000000000000000000000000000;  
               Src2=32'b10000000000000000000000000000000;  
  
        #10    Src1=32'd25;  
               Src2=32'd19;  
  
        #10    funct=6'd28;  
  
        #10    Src1=32'd64;  
               Src2=32'd64;
```

```
endmodule
```

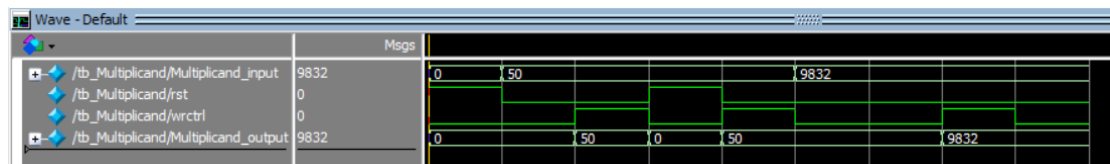
[illegible]

b.32-bit Multiplicand

```
module Multiplicand(  
    input [31:0] Multiplicand_input, //Multiplicand input  
    input rst, //Reset multiplicand  
    input wrctrl, //write the multiplicand into register  
    output reg [31:0] Multiplicand_output //Multiplicand output  
);  
  
always@(wrctrl or rst)begin  
    if(rst)Multiplicand_output=32'd0;//If reset signal is 1,put the value 0 to the output  
    else if(wrctrl)Multiplicand_output=Multiplicand_input;//If write control signal is 1,,put the value of input to the output  
end  
endmodule
```

Multiplicand 這個暫存器的寫法非常簡單明瞭，就是只要收到寫入的訊號，我們就將 input 丟到 output

```
`timescale 1ns/1ns  
module tb_Multiplicand;  
    reg [31:0] Multiplicand_input; //Multiplicand input  
    reg rst; //Reset multiplicand  
    reg wrctrl; //write the multiplicand into register  
    wire [31:0] Multiplicand_output; //Multiplicand output  
  
    Multiplicand Mult(Multiplicand_input,rst,wrctrl,Multiplicand_output);//The module of Multiplicand  
  
    initial begin//Simulation begin  
        Multiplicand_input=32'd0;  
        rst=1;  
        wrctrl=0;  
    #10  
        rst=0;  
        Multiplicand_input=32'd50;  
    #10  
        wrctrl=1;  
    #10  
        rst=1;  
        wrctrl=0;  
    #10  
        rst=0;  
        wrctrl=1;  
    #10  
        Multiplicand_input=32'd9832;  
        wrctrl=0;  
    #20  
        wrctrl=1;  
    #10  
        wrctrl=0;  
    #10;  
    end  
endmodule
```



而我 testbench 的測試想法就是 input 一個值，寫入之後清除，再寫入一次，在下次寫入之前改變 input 值看 output 會不會改變，再來寫入訊號，改變 output 結果，測試結果正常

c. 64-bit Product

```
module Product(  
    input [63:0] Product_input, //Product input  
    input wrctrl, //Write control signal  
    input strctrl, //Signal for store the ALU result into Product  
    input ready, //Ready control signal  
    input rst, //Reset signal  
    input clk, //Clock  
    output reg [63:0] Product_output //Product output  
);  
  
    reg [2:0] state; //The state  
    reg [2:0] next_state; //The next state  
  
    always@(posedge clk or posedge rst) begin  
        if(rst) begin //If reset signal is 1, put the value 0 to all output and all setting  
            Product_output=64'd0;  
            state=3'd0;  
            next_state=3'd0;  
        end  
        else begin  
            state=next_state;  
  
            case(state)  
                3'd0: begin //Put the initial value product input(32'd0, Multiplier) to product output  
                    if(wrctrl) begin  
                        Product_output[31:0]=Product_input[31:0];  
                        Product_output[63:32]=32'd0;  
                        next_state=3'd1;  
                    end  
                end  
                3'd1: next_state=3'd2; //First time need 1 cycle to load initial value  
                3'd2: begin //Shift product output  
                    Product_output=Product_output>>1;  
                    next_state=3'd3;  
                end  
                3'd3: begin //Identify put ALU result to product output(higher bits) or not  
                    if(strctrl) Product_output[63:31]=Product_input[63:31];  
                    next_state=3'd4;  
                end  
                3'd4: begin //Identify ready single  
                    if(ready) begin  
                        Product_output=Product_output;  
                        next_state=3'd5;  
                    end  
                    else next_state=3'd2;  
                end  
                3'd5: Product_output=Product_output; //maintain result until reset  
            endcase  
        end  
    end  
endmodule
```

這個部分跟 Control 一樣重要，算是計算乘法的重點之一，一開始並不是利用狀態機的方式下去寫，但因為一直出了錯誤，所以最後改成利用狀態機的方式去寫這個部分，首先狀態 0 是判斷是否有寫入訊號，再來狀態 1 非常特別，因為在第一次寫入時需要時間，所以多出一個 Cycle 的時間讓數值確定，再來狀態 2 的部分是做右移的部分，會先做右移是因為等等做加法的結果可以利用接線的方式做右移，這樣也可以確保解決 carry 的問題，狀態 3 是去判斷是否需要將 ALU 計算的結果丟入 Product，最後狀態 4 去判斷 ready 訊號，也就是乘法是否做完了，最後維持 result 直到 reset

```

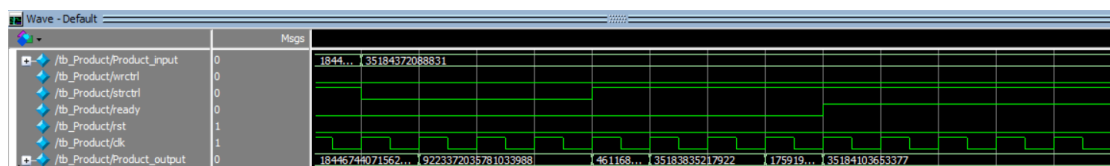
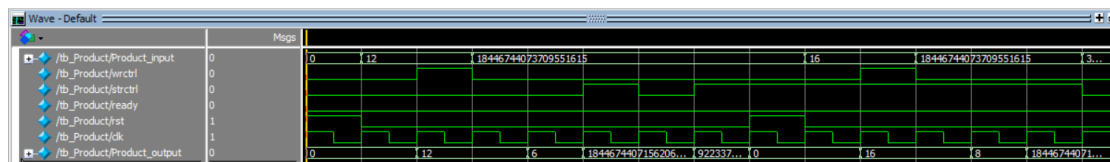
`timescale 1ns/1ns
module tb_Product;
reg [63:0] Product_input; //The signal of input
reg wrctrl; //The signal of write control
reg strctrl; //Signal for store the ALU result into Product
reg ready; //The signal of ready control
reg rst; //The signal of reset
reg clk; //The signal of clock
wire [63:0] Product_output; //The signal of output

Product Product(Product_input,wrctrl,strctrl,ready,rst,clk,Product_output); //The module of Product

initial begin //Simulation begin
    Product_input=64'd0;
    wrctrl=0;
    strctrl=0;
    ready=0;
    rst=1;
    clk=1;
#10
    rst=0;
    Product_input=64'd12;
#10
    wrctrl=1;
#10
    wrctrl=0;
    Product_input=64'b11111111_11111111_11111111_11111111_11111111_11111111_11111111_1111;
#20
    strctrl=1;
#10
    strctrl=0;
#10
    strctrl=1;
#10
    rst=1;
#10
    rst=0;
    Product_input=64'd16;
#10
    wrctrl=1;
#10
    wrctrl=0;
    Product_input=64'b11111111_11111111_11111111_11111111_11111111_11111111_11111111_1111;
#10
    strctrl=1;
#20
    strctrl=0;
    Product_input=64'b0000000000_0000000001_11111111_11111111_11111111_11111111_11111111_1111;
#40
    strctrl=1;
#40
    ready=1;
#50
    $finish;
end

always #5 begin //Creat a clock which the period is 10ns and the duty cycle is 50%
    clk = ~clk;
end
endmodule

```



Product 的 testbench 我一開始設計是測試 reset 是否正常，再來 reset 過後依序去測試它的寫入控制是否正常，從 wave 去觀察，一開始輸入 16 進去，再來去判斷是否讀取 ALU 結果，我們可以從 wave 看到 output 因為 strctrl 的關係被寫入到 output 的部分

d.Control

```
module Control(  
    input run,//Signal for start the multiplication  
    input rst,//Signal for reset  
    input clk,//Clock  
    input lsb,//The least bit of multiplier  
    output reg ready,//Result ready  
    output reg strctrl, //Signal for store the ALU result into Product  
    output reg wrctrl,//The signal of write control  
    output reg [5:0] addctrl //for ALU function  
);  
reg [2:0]state;//The state  
reg [2:0]next_state;//The next state  
reg strctrl_wait;//We need to shift first,so we creat a register to store the state  
integer i;//Creat the integer i to make sure how much counted  
  
always@(posedge clk or posedge rst)begin  
    if(rst)begin//If reset signal is 1,put the value 0 to all output and all setting  
        i=0;  
        ready=0;  
        strctrl=0;  
        strctrl_wait=0;  
        wrctrl=0;  
        addctrl=6'd0;  
        state=3'd0;  
        next_state=3'd0;  
    end  
    else begin  
        ready=0;//Everytime needs to clear previous setting  
        strctrl=0;  
        wrctrl=0;  
        addctrl=6'd0;  
  
        state=next_state;  
        case(state)  
            3'd0:begin//Identify can run or not  
                i=0;  
                if(run) begin  
                    wrctrl=1;  
                    next_state=3'd1;  
                end  
            end  
            3'd1:next_state=3'd2;  
            3'd2:begin//Identify lsb use ALU to ADD or not  
                if(lsb) begin  
                    addctrl=6'd27;  
                    strctrl_wait=1;  
                end  
                next_state=3'd3;  
            end  
            3'd3:begin//Identify strctrl_wait load ALU result or not  
                if(strctrl_wait) begin  
                    strctrl=1;  
                    strctrl_wait=0;  
                end  
                next_state=3'd4;  
            end  
  
            3'd4:begin//Check times  
                i=i+1;  
                if(i==32) begin  
                    ready=1;  
                    next_state=3'd5;  
                end  
                else next_state=3'd2;  
            end  
            3'd5:ready=1;//Maintain ready until reset  
        endcase  
    end  
end  
endmodule
```

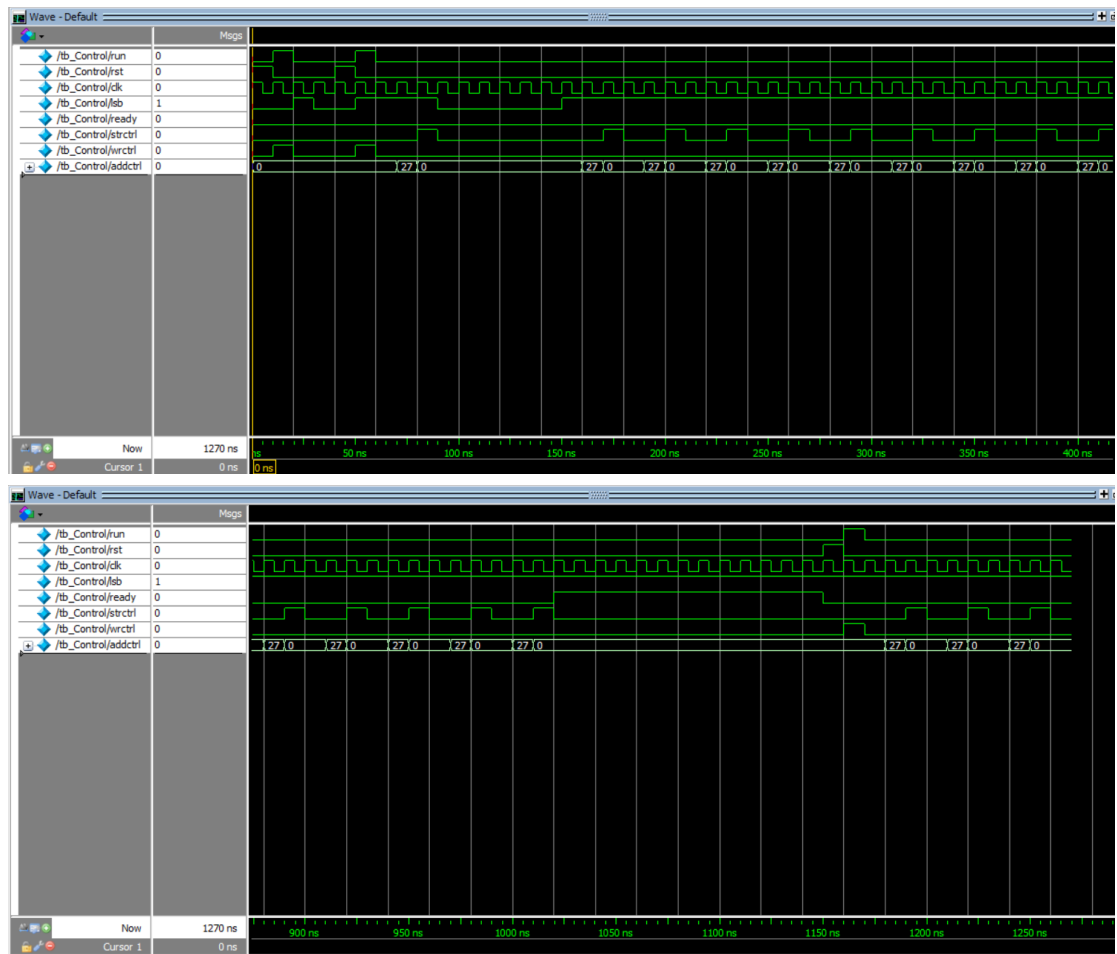
這個部分算是乘法的核心，因為這是負責所有的控制訊號的部分，首先狀態 0 就是去判斷是否有 run 的訊號，如果有就要將 input 的值丟進來，也就是寫入訊號控制，狀態 1 跟 Product 一樣就是等待輸入值，狀態 2 是判斷最小位元是否為 1，去控制下一個狀態是否將 ALU 的值寫入 Product 的最高 32 位元裡，狀態 3 是接續上一個狀態，去判斷是否將 ALU 的值寫入，狀態 4 是去判斷是否循環了 32 次，也就是乘法是否做完，最後維持 ready 訊號直到 reset

```
`timescale 1ns/1ns
module tb_Control;
    reg run;//The signal of run
    reg rst;//The signal of reset
    reg clk;//The signal of clock
    reg lsb;//The signal of the least bit of multiplier
    wire ready;//The signal of ready signal
    wire strctrl;//Signal for store the ALU result into Product
    wire wrctrl;//The signal of write control
    wire [5:0] addctrl;//The signal of function code

    Control Ctrl(run,rst,clk,lsb,ready,strctrl,wrctrl,addctrl);//The module of Controller

    initial begin//Simulation begin
        rst=1;
        run=0;
        clk=1;
        lsb=0;
        #10    rst=0;
                run=1;
        #10    run=0;
                lsb=1;
        #10    lsb=0;
        #10    rst=1;
        #10    rst=0;
                run=1;
                lsb=1;
        #10    run=0;
                lsb=1;

        #30    lsb=0;
        #30    lsb=0;
        #30    lsb=1;
        #1000  rst=1;
        #10    rst=0;
                run=1;
        #10    run=0;
        #100  $finish;
    end
    always begin//Creat a clock which the period is 10ns and the duty cycle is 50%
        #5    clk=~clk;
    end
endmodule
```



而我 testbench 的寫法是去測試 reset 是否正常，還有 lsb 是 0 是 1 時，我不會有寫入的問題，最後去找出做完的時候我的 ready 是否會正常跳起一個 Cycle 的時間，最後測試結果均正常

e. 32-bits multiplier

```

module CompMul(
    input [31:0] Multiplicand_input, //Multiplicand input
    input [31:0] Multiplier_input, //Multiplier input
    input run, //Start signal
    input rst, //Reset all the control signals
    input clk, //Clock signal
    output ready, //Result ready
    output [63:0] Product_output //Arithmetic result
);

    wire [31:0] Multiplicand_output; //The wire of multiplicand output
    wire [63:0] Product_input; //The wire of product input
    wire [32:0] Product_input_new;
    wire [5:0] addctrl; //The wire of function code
    wire [4:0] shamt=5'd0; //The wire of shift amount, but this situation don't need, then set the value is 0
    wire zero; //The wire of zero flag
    wire stroctrl; //Signal for store the ALU result into Product
    wire wrctrl; //The wire of write control
    reg state; //The state for put the initial value (multiplier input) to product
    reg next_state; //The next state

    assign Product_input=(state)?(Product_input_new[32:0],Product_output[30:0]):(32'd0,Multiplier_input); //Set product input

    Control ctr(run,rst,clk,Product_output[0],ready,stroctrl,wrctrl,addctrl); //The module of Controller
    Multiplicand Mult(Multiplicand_input,rst,wrctrl,Multiplicand_output); //The module of Multiplicand
    Product Product(Product_input,wrctrl,stroctrl,ready,rst,clk,Product_output); //The module of Product
    ALU uut(Multiplicand_output,Product_output[63:32],addctrl,shamt,Product_input_new[31:0],zero,Product_input_new[32]); //The module of ALU

    always@(posedge clk or posedge rst)begin
        if(rst)begin//If reset signal is 1, put the value 0 to all setting
            state=0;
            next_state=0;
        end
        else begin
            state=next_state;
            case(state)
                1'd0:if(run)next_state=1'd1; //If run signal is 1, start to multiply
                1'd1:if(ready) next_state=1'd0; //If the multiplication finished go back to the begin
            endcase
        end
    end
endmodule

```

這部分就是 Part2 最後的部分，按照我的想法去完成接線，但在這個過程中我也 debug 用了很久。這個部分最特別的就是 Product input 的部分，因為第一次讀入跟其他次讀入的接線不同，所以特地利用 state 去做選擇，看是要 Multiplier 的輸入還是要 ALU 的輸出。

```

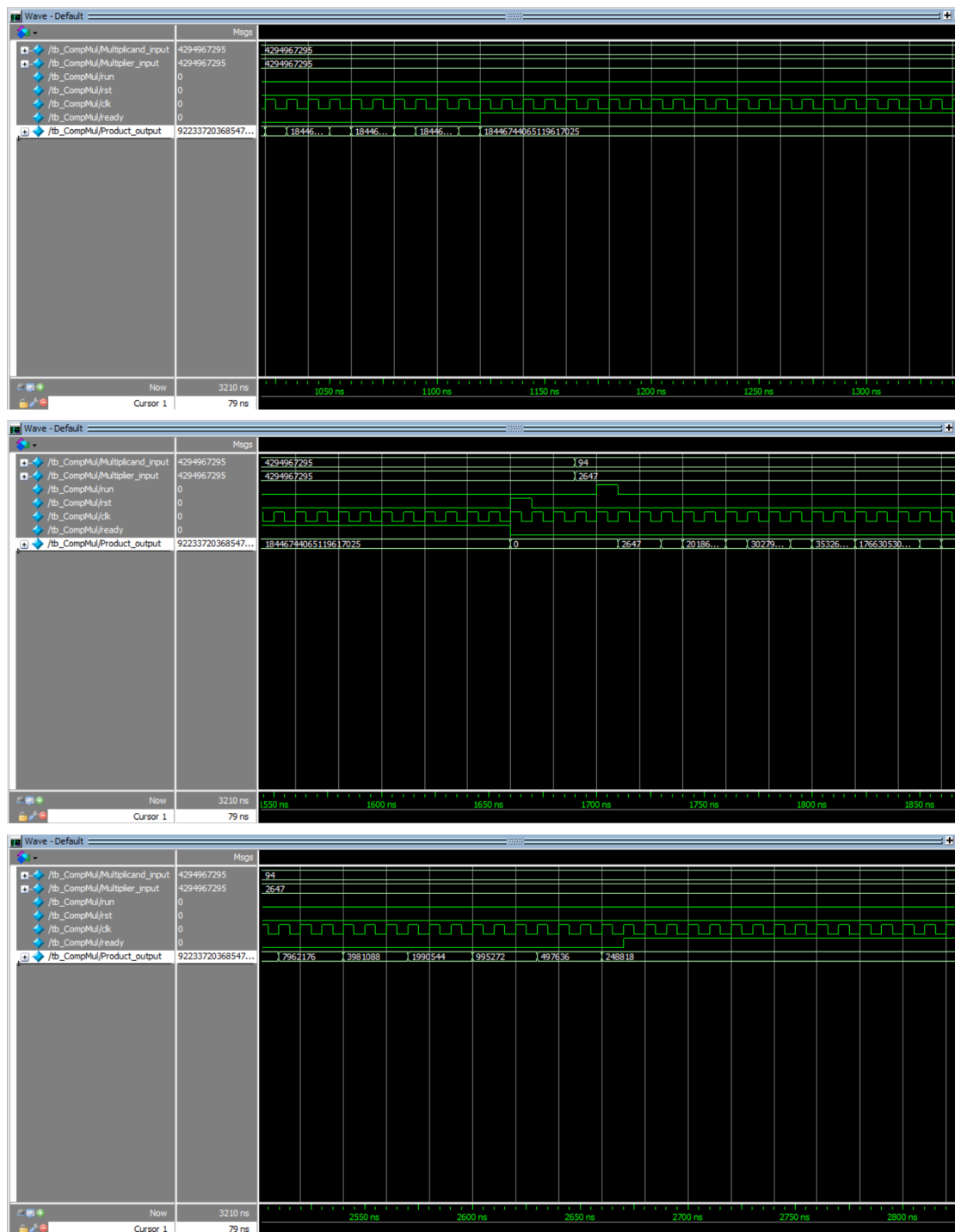
`timescale 1ns/1ns
module tb_CompMul;
    reg [31:0] Multiplicand_input; //The signal of multiplicand input
    reg [31:0] Multiplier_input; //The signal of multiplier input
    reg run; //The signal of run control
    reg rst; //The signal of reset
    reg clk; //The signal of clock
    wire ready; //The signal of ready control
    wire [63:0] Product_output; //The value of result

    CompMul Mult(Multiplicand_input,Multiplier_input,run,rst,clk,ready,Product_output); //The module of CompMul

    initial begin//Simulation begin
        Multiplicand_input=32'd4294967295; //All 32-bit are 1
        Multiplier_input=32'd4294967295; //All 32-bit are 1
        run=0;
        rst=1;
        clk=1;
        #10 rst=0;
        #10 run=1;
        #10 run=0;
        #100 rst=1;
        #20 rst=0;
        run=1;
        #10 run=0;
        #1500 rst=1;
        #10 rst=0;
        #20 Multiplicand_input=32'd94; //test
        Multiplier_input=32'd2647; //test
        ...

        #10 run=1;
        #10 run=0;
        #1500 $finish;
    end
    always begin//Creat a clock which the period is 10ns and the duty cycle is 50%
        #5 clk=~clk;
    end
endmodule

```

這是這個 Part 的 testbench，主要測試的是幾個功能，首先第一次的兩個數字都是用最大的數(32bit 全為 1)，去測試乘法器在最大數相乘是否會出錯，第一次是從 150ns 開始，跑到 1120ns 結果輸出，然後結果一直維持到 1670ns 第二次計算開始，而第二次計算結束是在 2640ns，每一次的計算都是在 960ns(96 個 cycle)做完

Part1:

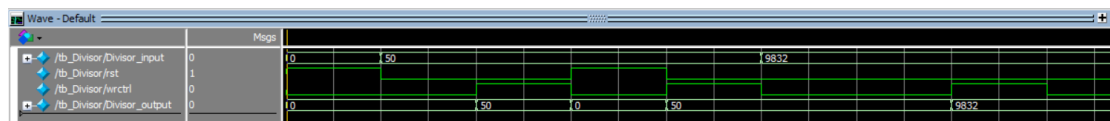
a.32-bit ALU

```
module ALU(  
    input [31:0] Source1, //Register1 input  
    input [31:0] Source2, //Register2 input  
    input [5:0] funct, //R type function code  
    input [4:0] shamt, //Shift amount  
    output reg [31:0] result=32'd0, //Result output  
    output zero, //Zero flag  
    output reg carry=0 //Carry flag  
);  
  
    assign zero=(result==0)?1:0;//If the result is zero,the zero flag is 1  
  
    always@(Source1 or Source2 or funct or shamt)begin  
        case(funct[5:0])//Identify function code  
            6'd27: {carry,result}<=Source1+Source2;//Function ADD  
            6'd28: result<=Source1-Source2;//Function SUB  
            6'd29: result<=Source1>>shamt;//Function SRL  
            6'd30: result<=Source1<<shamt;//Function SLL  
            6'd31: result<=Source1^Source2;//Function XOR  
            6'd32: result<=Source1&Source2;//Function AND  
            default: result<=result;//If function code no match,maintain the result  
        endcase  
    end  
endmodule  
  
module tb_ALU();  
  
    reg [31:0]Src1;//The signal of Source1  
    reg [31:0]Src2;//The signal of Source2  
    reg [5:0]funct;//The signal of function code  
    reg [4:0]shamt;//The signal of shift amount  
    wire [31:0]result;//The signal of result  
    wire zero;//The signal of zero flag  
    wire carry;//The signal of carry flag  
  
    ALU uut(Src1,Src2,funct,shamt,result,zero,carry);//The module of ALU  
  
    initial begin//simulation begin  
        Src1=0;  
        Src2=0;  
        funct=0;  
        shamt=0;  
  
        #20    funct=6'd27;  
        #10    Src1=32'b10000000000000000000000000000000;  
                Src2=32'b10000000000000000000000000000000;  
  
        #10    Src1=32'd3;  
                Src2=32'd4;  
  
        #10    funct=6'd28;  
  
        #10    Src1=32'd64;  
                Src2=32'd64;
```

[illegible]

b. 32-bit Divisor

```
module Divisor(  
    input [31:0] Divisor_input, //Divisor input  
    input rst, //Reset Divisor  
    input wrctrl, //write the Divisor into register  
    output reg [31:0] Divisor_output //Divisor output  
);  
  
always@(wrctrl or rst)begin  
    if(rst)Divisor_output=32'd0;//If reset signal is 1,put the value 0 to the output  
    else if(wrctrl)Divisor_output=Divisor_input;//If write control signal is 1,put the value of input to the output  
end  
endmodule  
  
`timescale 1ns/1ns  
module tb_Divisor;  
    reg [31:0] Divisor_input; //Divisor input  
    reg rst; //Reset divisor  
    reg wrctrl; //write the divisor into register  
    wire [31:0] Divisor_output; //Divisor output  
  
    Divisor Div(Divisor_input,rst,wrctrl,Divisor_output);//The module of Divisor  
  
    initial begin//Simulation begin  
        Divisor_input=32'd0;  
        rst=1;  
        wrctrl=0;  
        #10  
        rst=0;  
        Divisor_input=32'd50;  
        #10  
        wrctrl=1;  
        #10  
        rst=1;  
        wrctrl=0;  
        #10  
        rst=0;  
        wrctrl=1;  
        #10  
        Divisor_input=32'd9832;  
        wrctrl=0;  
        #20  
        wrctrl=1;  
        #10  
        wrctrl=0;  
        #10;  
    end  
endmodule
```



Part3 的這個部分與 Part2 的 Multiplicand 相同，因此說明省略

c.64-bit Remainder

```
module Remainder(
    input [63:0] Remainder_input, //Remainder input
    input wrctrl, //Write control signal
    input ozctrl, //Add 1 or 0 at the least bit
    input ready, //Ready control signal
    input ready_wait, //Ready control signal before shift right
    input rst, //Reset signal
    input clk, //Clock
    output reg [31:0] Remainder_output, //Remainder output
    output reg [31:0] Quotient_output //Quotient output
);

reg [2:0] state; //The state
reg [2:0] next_state; //The next state

always@(posedge clk or posedge rst)begin
    if(rst)begin//If reset signal is 1, put the value 0 to all output and all setting
        Remainder_output=32'd0;
        Quotient_output=32'd0;
        state=3'd0;
        next_state=3'd0;
    end
    else begin
        state=next_state;

        case(state)
            3'd0:begin//Put the initial value Remainder input[32'd0, Dividend] to Remainder output and Quotient output
                if(wrctrl) begin
                    Quotient_output[31:0]=Remainder_input[31:0];
                    Remainder_output[31:0]=32'd0;
                    next_state=3'd1;
                end
            end
            3'd1:begin//First time shift left
                {Remainder_output, Quotient_output}={Remainder_output, Quotient_output}<<1;
                next_state=3'd2;
            end
            3'd2:begin//Put ALU result(SUB) into Remainder output
                Remainder_output[31:0]=Remainder_input[63:32];
                next_state=3'd3;
            end
            3'd3:next_state=3'd4; //One cycle to load result
            3'd4:begin//Identify put 1 or 0 at the least bit
                if(ozctrl) begin
                    {Remainder_output, Quotient_output}={Remainder_output, Quotient_output}<<1;
                    {Remainder_output, Quotient_output}={Remainder_output, Quotient_output}+1;
                end
                else begin
                    Remainder_output[31:0]=Remainder_input[63:32];
                    {Remainder_output, Quotient_output}={Remainder_output, Quotient_output}<<1;
                end
                next_state=3'd5;
            end
            3'd5:begin//Identify ready single before shift right
                if(ready_wait) begin
                    Remainder_output=Remainder_output>>1;
                    next_state=3'd6;
                end
                else
                    next_state=3'd2;
            end
            3'd6:begin//maintain result until reset
                Remainder_output=Remainder_output;
                Quotient_output=Quotient_output;
            end
        endcase
    end
end
endmodule
```

Part3 的這個部分我也是利用狀態機的方式去處理，因此只要按照著順序步驟，輸入什麼 Control 的訊號，就去執行該做的事，就不會有太大的問題，但因為我的 Remainder 原本在 Control 輸出 ready 時，還需要做右移的一個步驟，所以實際上 Remainder 那時還沒做完，因此我多了一個訊號線 ready_wait 是讓 Remainder 有一個 Cycle 做最後的右移

```

|timescale 1ns/1ns
module tb_Remainder;
reg [63:0] Remainder_input; //Remainder input
reg wrctrl; //Write control signal
reg ozctrl; //Add 1 or 0 at the least bit
reg ready; //Ready control signal
reg ready_wait; //Ready control singal before shift right
reg rst; //Reset signal
reg clk; //Clock
wire [31:0] Remainder_output; //Remainder output
wire [31:0] Quotient_output; //Quotient output

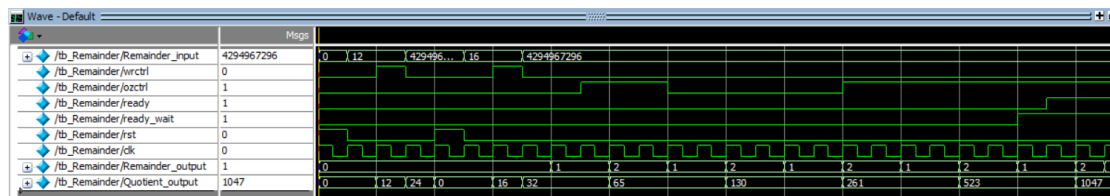
Remainder Remainder(Remainder_input,wrctrl,ozctrl,ready,ready_wait,rst,clk,Remainder_output,Quotient_output); //The module of Remainder

initial begin//Simulation begin
    Remainder_input=64'd0;
    wrctrl=0;
    ozctrl=0;
    ready=0;
    ready_wait=0;
    rst=1;
    clk=1;
#10
    rst=0;
    Remainder_input=64'd12;
#10
    wrctrl=1;
#10
    wrctrl=0;
    Remainder_input=64'd4294967296; //32-bit is 1, other are 0
#10
    rst=1;
#10
    rst=0;
    Remainder_input=64'd16;

#10
    wrctrl=1;
#10
    wrctrl=0;
    Remainder_input=64'd4294967296;
#20
    ozctrl=1;
#30
    ozctrl=0;
#60
    ozctrl=1;
#60
    ready_wait=1;
#10
    ready=1;
#30
    $finish;
end

always #5 begin//Creat a clock which the period is 10ns and the duty cycle is 50%
    clk = ~clk;
end
endmodule

```



d.Control

```
module Control(
    input run, //Signal for start the multiplication
    input rst, //Signal for reset
    input clk, //Clock
    input fsb, //The first bit of multiplier
    output reg ready, //Result ready
    output reg ready_wait, //Result ready but need to shift right
    output reg wrctrl, //The signal of write control
    output reg ozctrl, //Add one or zero at the least Remainder
    output reg [5:0] ALUfunction //for ALU function
);
    reg [2:0] state; //The state
    reg [2:0] next_state; //The next state
    integer i; //Creat the integer i to make sure how much counted

    always@(posedge clk or posedge rst)begin
        if(rst)begin //If reset signal is 1, put the value 0 to all output and all setting
            i=0;
            ready=0;
            ready_wait=0;
            wrctrl=0;
            ozctrl=0;
            ALUfunction=6'd0;
            state=3'd0;
            next_state=3'd0;
        end
        else begin
            ready=0; //Everytime needs to clear previous setting
            ready_wait=0;
            wrctrl=0;
            ALUfunction=6'd0;
            state=next_state;

            case(state)
                3'd0:begin //Identify can run or not
                    i=0;
                    if(run) begin
                        wrctrl=1;
                        next_state=3'd1;
                    end
                end
                3'd1:next_state=3'd2; //This Cycle Remainder is doing another work
                3'd2:begin //Control ALU to do SUB function
                    ALUfunction=6'd28;
                    next_state=3'd3;
                end
                3'd3:next_state=3'd4; //One Cycle to load result
                3'd4:begin //Identify the first bit to restore original value or not
                    if(fsb) begin
                        ALUfunction=6'd27;
                        ozctrl=0;
                    end
                    else begin
                        ozctrl=1;
                    end
                    next_state=3'd5;
                end
                3'd5:begin //Check times
                    i=i+1;
                    if(i==32) begin
                        ready_wait=1;
                        next_state=3'd6;
                    end
                    else next_state=3'd2;
                end
                3'd6:ready=1; //maintain ready until reset
            endcase
        end
    end
endmodule
```

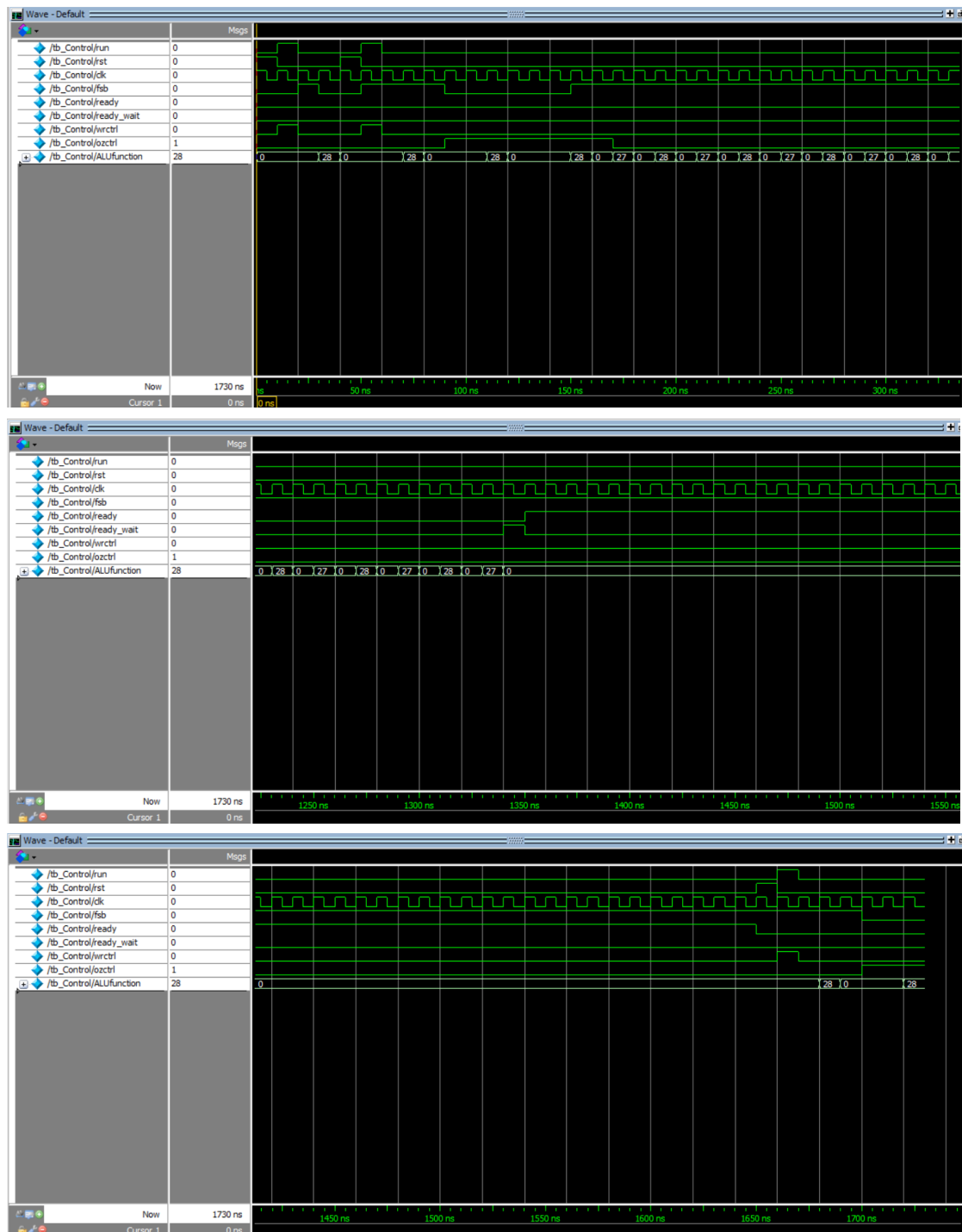

Part3 的 Control 會比 Part2 的難一點點，因為要考慮的因素變多，左移右移還有 ALU function 有兩種，因此在編寫的時候有滿多問題。首先我要解釋對這個過程的想法，一開始當然是 write 的寫入動作，再來直接做第一次的左移，我並沒有訊號線去控制左右移，因為我將這個事情直接寫在 Remainder 裡了。再來要控制 ALU 去做減法的動作，因為結果有可能為負數或正數，因此要去判斷最高位元是 1 或是 0，我在這裡設了一個 input 為 fsb 去判斷最高位元，如果為 1(負數)我要利用加法去把原本的值變回來，這樣我就不用特地將值存下來，接下來最後是去檢查是否有 32 次循環了，如果有就要輸出 ready_wait 和 ready

```
`timescale 1ns/1ns
module tb_Control;
reg run;//Signal for start the multiplication
reg rst;//Signal for reset
reg clk;//Clock
reg fsb;//The first bit of multiplier
wire ready;//Result ready
wire ready_wait;//Result ready but need to shift right
wire wrctrl;//The signal of write control
wire ozctrl;//Add one or zero at the least Remainder
wire [5:0] ALUfunction; //for ALU function

Control Ctrl(run,rst,clk,fsb,ready,ready_wait,wrctrl,ozctrl,ALUfunction);//The module of Controller

initial begin//Simulation begin
    rst=1;
    run=0;
    clk=1;
    fsb=0;
    #10    rst=0;
           run=1;
    #10    run=0;
           fsb=1;
    #10    fsb=0;
    #10    rst=1;
    #10    rst=0;
           run=1;
           fsb=1;
    #10    run=0;
           fsb=1;

    #30    fsb=0;
    #30    fsb=0;
    #30    fsb=1;
    #1500  rst=1;
    #10    rst=0;
           run=1;
           fsb=1;
    #10    run=0;
           fsb=1;
    #30    fsb=0;
    #30    $finish;
end
always begin//Creat a clock which the period is 10ns and the duty cycle is 50%
    #5     clk=~clk;
end
endmodule
```



這部分的 testbench 我的想法是一樣先去測試 reset 功能是否正常，再來是隨機測試 fsb 是 0 是 1 時的 ALU function 是否有跟著對應的判斷去更改 function code，測試結果為正常，最後去看它是否有判斷到執行完的訊號(Ready)，最後測試結果均為正常

e.32-bits divider

```
module CompDiv(
    input [31:0] Divisor_input, //Divisor input
    input [31:0] Dividend_input, //Dividend input
    input run, //Start signal
    input rst, //Reset all the control signals
    input clk, //Clock signal
    output ready, //Result ready
    output [31:0] Quotient_output, //Arithmetic result
    output [31:0] Remainder_output //Remainder
);

    wire [31:0] Divisor_output; //The wire of multiplicand output
    wire [63:0] Remainder_input; //The wire of product input
    wire [32:0] Remainder_input_new;
    wire [5:0] ALUfun; //The wire of function code
    wire [4:0] shamt=5'd0; //The wire of shift amount, but this situation don't need, then set the value is 0
    wire zero; //The wire of zero flag
    wire carry; //The wire of carry flag
    wire wrctrl; //The wire of write control
    wire ready_wait; //The wire of ready signal before shift right
    reg state; //The state for put the initial value (multiplier input) to product
    reg next_state; //The next state

    assign Remainder_input=(state)?(Remainder_input_new[31:0],Quotient_output[31:0]):(32'd0,Dividend_input); //Set remainder input

    Control ctr(run,rst,clk,Remainder_output[31],ready,ready_wait,wrctrl,ozctrl,ALUfun); //The module of Controller
    Divisor Div(Divisor_input,rst,wrctrl,Divisor_output); //The module of Divisor
    Remainder Remainder(Remainder_input,wrctrl,ozctrl,ready,ready_wait,rst,clk,Remainder_output,Quotient_output); //The module of remainder
    ALU uut(Remainder_output[31:0],Divisor_output,ALUfun,shamt,Remainder_input_new[31:0],zero,carry); //The module of ALU

    always@(posedge clk or posedge rst)begin
        if(rst)begin//If reset signal is 1, put the value 0 to all setting
            state=0;
            next_state=0;
        end
        else begin
            state=next_state;
            case(state)
                1'd0:if(run)next_state=1'd1; //If run signal is 1, start to multiply
                1'd1:if(ready)next_state=1'd0; //If the Divide finished go back to the begin
            endcase
        end
    end
endmodule
```

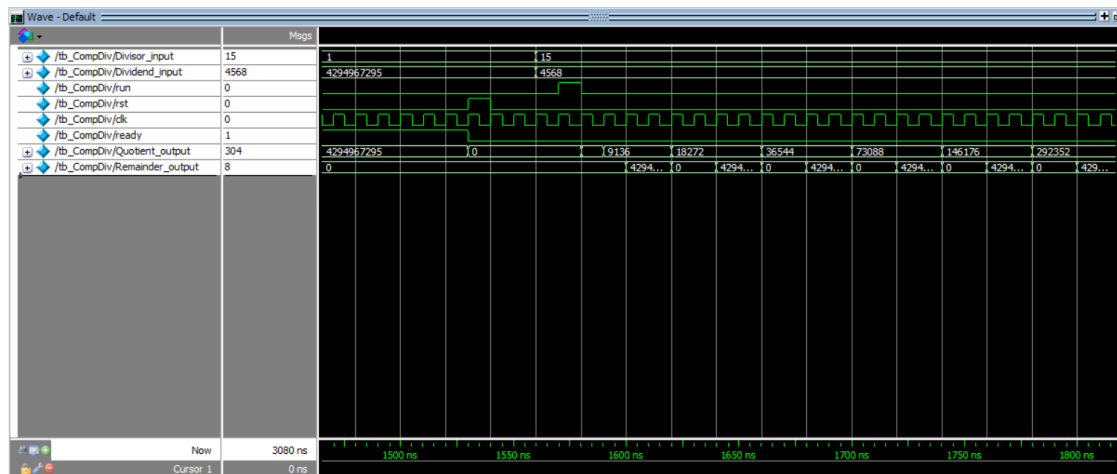
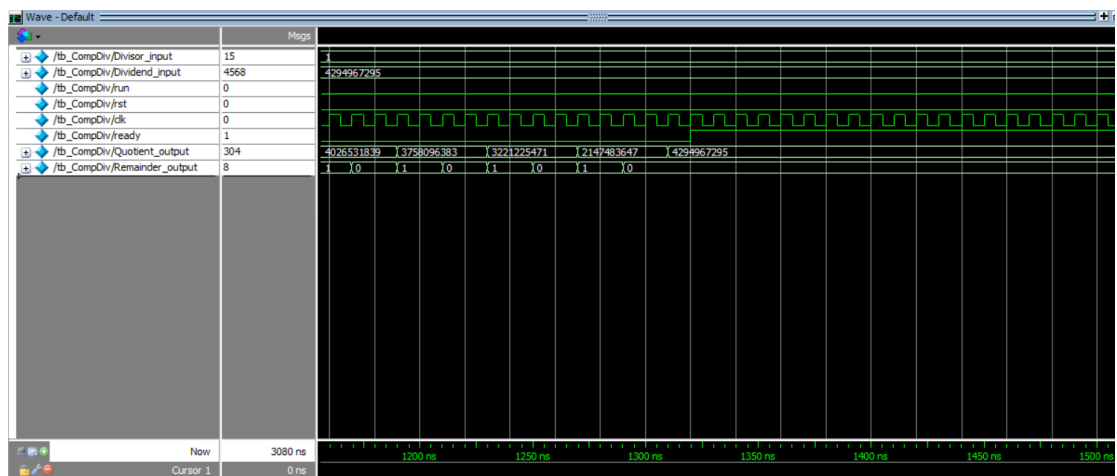
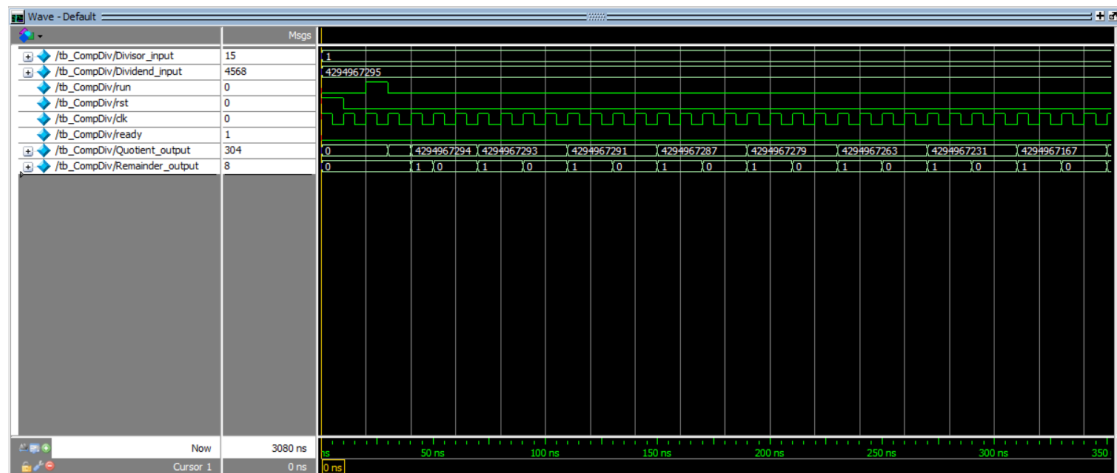
最後的這個部分就是將以上四個 module 組合在一起，當然在這個部分也會遇到跟 Part2 一樣的問題，就是在 Remainder 的輸入選擇上在會有是否是第一次的差異，因此我用的方式與 Part2 一樣，寫一個 state 去判斷是否為第一次，當然我做的這個除法器，一樣是可以執行多次，不須 reset 就可以直接持續做

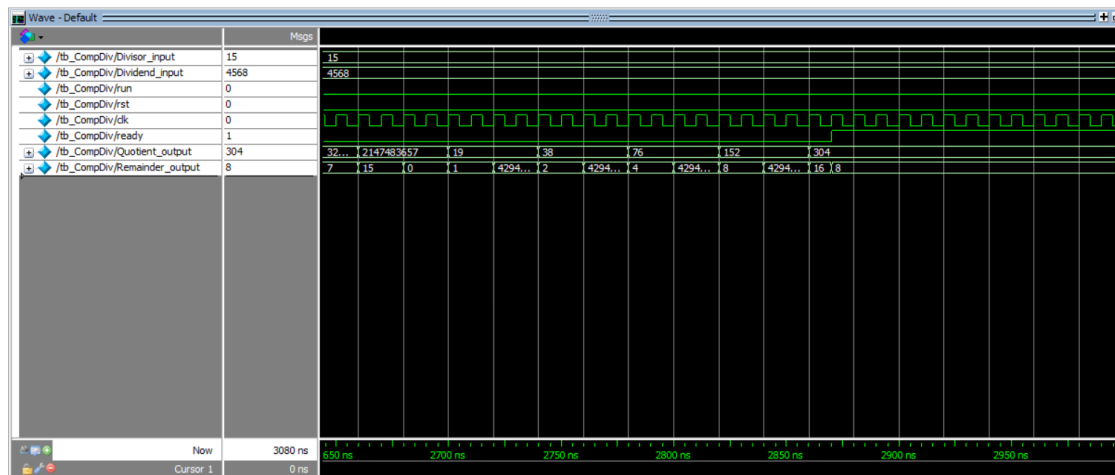
```
`timescale 1ns/1ns
module tb_CompDiv;
    reg [31:0] Divisor_input; //The signal of Divisor input
    reg [31:0] Dividend_input; //The signal of Dividend input
    reg run; //The signal of run control
    reg rst; //The signal of reset
    reg clk; //The signal of clock
    wire ready; //The signal of ready control
    wire [31:0] Quotient_output; //Arithmetic result
    wire [31:0] Remainder_output; //Remainder

    CompDiv Div(Divisor_input,Dividend_input,run,rst,clk,ready,Quotient_output,Remainder_output); //The module of CompDiv

    initial begin//Simulation begin
        Divisor_input=32'd1;
        Dividend_input=32'b11111111111111111111111111111111; //All 32-bit are 1
        run=0;
        rst=1;
        clk=1;
    #10
        rst=0;
    #10
        run=1;
    #10
        run=0;
    #1500
        rst=1;
    #10
        rst=0;
    #20
        Divisor_input=32'd15; //test
        Dividend_input=32'd4568; //test
    #10
        run=1;
    #10
        run=0;
    #1500
        $finish;
    end

    always begin//Creat a clock which the period is 10ns and the duty cycle is 50%
        #5
        clk=~clk;
    end
endmodule
```





Testbench 可以觀察到一開始我在 20ns 時開始用 32bit 全是 1 的數去除以 1，最後的結果在 1320ns 時輸出一個 Cycle 的 ready 訊號，商是 32bit 全是 1 的數，餘數為 0，結果正確。第二次在 1540ns 時開始執行 4568 除以 15 的除法，在 2840ns 時輸出 ready 訊號，商為 304 餘數為 8，結果正確。兩次除法耗時皆為 1300ns(130Cycle 的時間)