



國立台灣科技大學
電機工程系

計算機組織

(EE3005301)

Project 1

班級：四電機三甲

學號：B10507004

指導老師：陳雅淑

姓名：游照臨

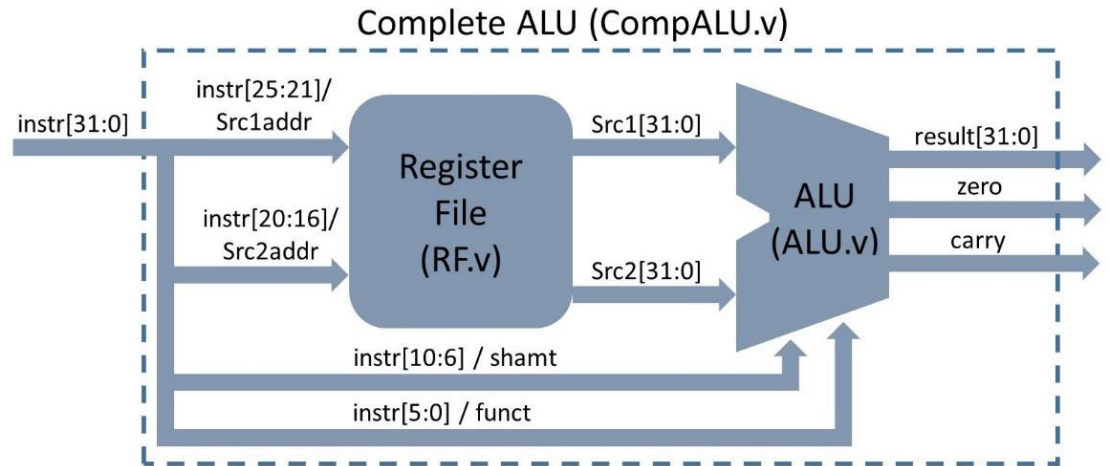
中華民國 108 年 4 月 10 日

壹、Part 1

一、題目

Implement a 32-bit Complete ALU

二、架構圖



三、功能描述

(一) 32-bit ALU

在此部分需實作具有六種運算的 ALU 元件，運算元如下表所示。依照運算類型將正確結果由 result 訊號輸出，當 result 為 0 時 zero=1，反之，當 result 不為 0 時 zero=0；而 carry 記錄運算是否有進位的情形發生，例如兩輸入訊號 MSB 皆為 1 作相加，將產生一位元的進位結果，此時 carry=1。

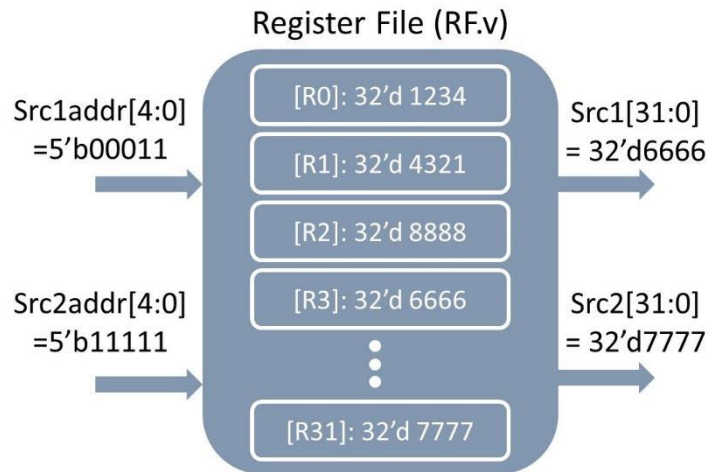
1. 六種功能表

Operation	Name	funct
ADD	Add	27
SUB	Subtract	28
AND	And	29
OR	Or	30
SRL	Shift Right Logical	31
SLL	Shift Left Logical	32

(二) 32-bit Register File (Read Only)

為了提供 ALU 運算值的來源，在此部分將實作一組 32 位元且唯讀的 Register File，首先輸入兩個 5 位元的暫存器地址，接著將選到的暫存器內容輸出，如圖二所示。假設在 Src1addr 輸入 5' b00011=3(10)，則 Src1 將輸出 R3 暫存器的內存值；與此同時在 Src2addr 輸入 5' b11111=31(10)，則 Src2 將輸出 R31 暫存器的內存值。

(Shift operation 對 Src1 訊號作位移，並輸出至 result。)

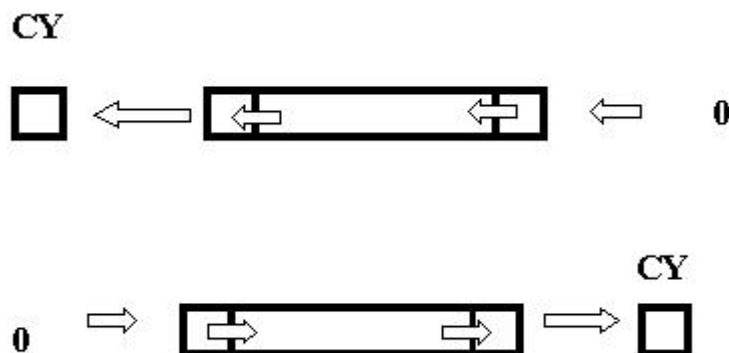


(三) Complete ALU:

opcode	rs	rt	rd	shamt	funct
opcode(6 bits): partially specifies what instruction it is (Note: 0 for all RFormat instructions) funct(6- bits): combined with opcode to specify the instruction rs (5-bits)(Source Register): <i>generally</i> used to specify register containing first operand rt (5-bits)(Target Register): <i>generally</i> used to specify register containing second operand rd (5-bits)(Destination Register): <i>generally</i> used to specify register which will receive result of computation shamt(5-bits) : the amount of bits to shift. Used in shift instructions.					

(四) Carry

- 雖然根據題目敘述，僅描述了加法的 Carry，但是事實上減法與位移也都有 Carry，根據定義，減法的 Carry 為：當被減數小於減數時，也就是會產生負值的狀況下就會有 Carry，而 Shift 的 Carry 定義則依照各參考書的定義而有所不同，我設計的 Shift Carry 則是依照下圖定義。



四、程式碼

(一) RF

1. 程式碼

```
1. module RF(  
2.     input  [4:0]Src1addr,  
3.     input  [4:0]Src2addr,  
4.     output [31:0]Src1,  
5.     output [31:0]Src2  
6. );  
7.     REGISTER_SEL src1_sel(Src1addr,Src1);  
8.     REGISTER_SEL src2_sel(Src2addr,Src2);  
9.  
10. endmodule  
11.  
12. module REGISTER_SEL(  
13.     input [4:0]Srcaddr,  
14.     output reg [31:0]src  
15. );  
16.     always @(Srcaddr)begin  
17.         case (Srcaddr[4:0])  
18.             5'd0: src <= 0;  
19.             5'd1: src <= 1;  
20.             5'd2: src <= 2;  
21.             5'd3: src <= 3;  
22.             5'd4: src <= 4;  
23.             5'd5: src <= 5;  
24.             5'd6: src <= 6;  
25.             5'd7: src <= 7;  
26.             5'd8: src <= 8;  
27.             5'd9: src <= 9;  
28.             5'd10: src <= 10;  
29.             5'd11: src <= 11;  
30.             5'd12: src <= 12;  
31.             5'd13: src <= 13;  
32.             5'd14: src <= 14;  
33.             5'd15: src <= 15;  
34.             5'd16: src <= 16;  
35.             5'd17: src <= 17;
```

```

36.         5'd18: src <= 18;
37.         5'd19: src <= 19;
38.         5'd20: src <= 20;
39.         5'd21: src <= 21;
40.         5'd22: src <= 22;
41.         5'd23: src <= 23;
42.         5'd24: src <= 24;
43.         5'd25: src <= 25;
44.         5'd26: src <= 26;
45.         5'd27: src <= 27;
46.         5'd28: src <= 28;
47.         5'd29: src <= 29;
48.         5'd30: src <= 30;
49.         5'd31: src <= 31;
50.     endcase
51. end
52. endmodule

```

RF 的 Module 主要 Call 了 REGISTER_SEL 這個自己定義的 module，由於題目要求的標準輸出 output 沒有 reg，而且題目為雙輸入、雙輸出的需求，因此透過 REGISTER_SEL 的 module 來使用 always 執行，並在 RF 內建立兩次。

REGISTER_SEL 模組內，我透過 Behavioral model，以 Srcaddr 的變動觸發條件，將數字指定到輸出端。因為該題的輸入值恰好等於輸出值僅是特例，因此我依然把所有的狀況給條列出來，同樣的內容打需要打 32 次是一件很麻煩的事情，所以我使用了 python 腳本來自動產生部分的程式碼片段。

```

for i in range(32):
    print("5'd{0}: src <= {0};".format(i))

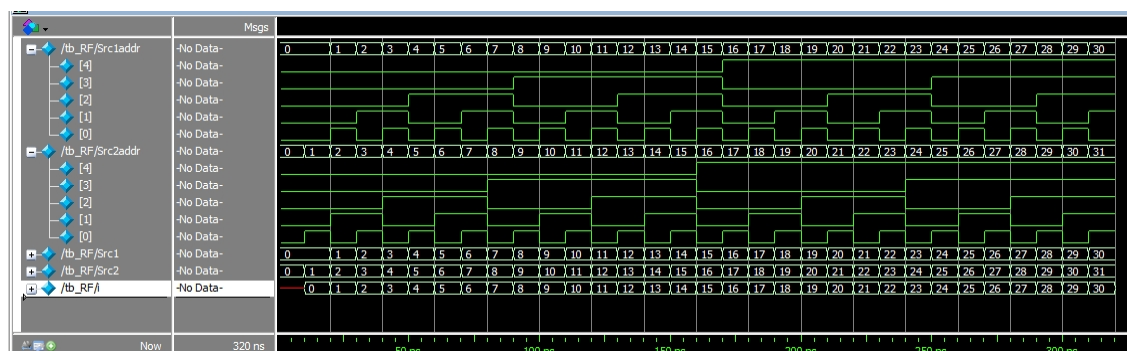
```

2. Testbench

```
1. `timescale 1ns/ 1ps
2. module tb_RF;
3.
4. reg [4:0] Src1addr;
5. reg [4:0] Src2addr;
6. wire [31:0] Src1;
7. wire [31:0] Src2;
8. integer i;
9.
10. RF registerf(Src1addr,Src2addr,Src1,Src2);
11.
12. initial begin
13.     Src1addr = 0;
14.     Src2addr = 0;
15.     #10;
16.     for(i=0;i<31;i=i+1)begin
17.         Src1addr = i;
18.         Src2addr = i+1;
19.         #10;
20.     end
21. end
22.
23. endmodule
```

此 testbench 透過 for 迴圈的方式，依序從 0 開始測試，並使 i=0 至 31，每 10ns 變化一次，使 Src1addr 的值從 0 依序增加到 30、Src2addr 的值從 1 依序增加到 31 為止。

3. Testbench 模擬結果圖



4. 模擬結果分析

由於本 RF 的規格預先知道輸入信號等於輸出信號，因此 Src1、Src2 的輸出剛好等於 Src1addr、Src2addr 的輸入信號。

(二) ALU

1. 程式碼

```
1. module ALU(  
2.     input [31:0] Src1,  
3.     input [31:0] Src2,  
4.     input [5:0] funct,  
5.     input [4:0] shamt,  
6.     output reg [31:0] result,  
7.     output zero,  
8.     output carry  
9. );  
10. wire [31:0]result1;  
11.  
12. initial begin  
13.     result = 0;  
14. end  
15.  
16. // when result=0 , zero = 1  
17. assign zero = result == 0 ? 1:0;  
18.  
19. ALU1 alu(  
20.     Src1,  
21.     Src2,  
22.     funct,  
23.     shamt,  
24.     result1,  
25.     carry  
26. );  
27. always @(result1)begin // 因為題目的 result 需要是 reg  
28.     result = result1;  
29. end  
30.  
31. endmodule  
32.
```

```

33. module ALU1(
34.     input [31:0] Src1,
35.     input [31:0] Src2,
36.     input [5:0] funct,
37.     input [4:0] shamt,
38.     output reg[31:0] result,
39.     output reg carry //因為題目 carry 沒有 reg，所以用另外
        一個 model 來連
40. );
41.
42. always@(Src1 or Src2 or funct or shamt)begin
43.     case (funct[5:0])
44.         6'd27: {carry,result} <= Src1 + Src2;
45.         6'd28: {carry,result} <= Src1 - Src2;
46.         6'd29: {carry,result} <= Src1 & Src2;
47.         6'd30: {carry,result} <= Src1 | Src2;
48.         6'd31: begin
49.             result <= (Src1 >> shamt);
50.             carry <= Src1[shamt-1];
51.         end
52.         6'd32: {carry,result} <= (Src1 << shamt);
53.         default: {carry,result} <= {1'b0,Src2};
54.     endcase
55. end
56.
57. endmodule

```

由於題目的 result 需求是 reg 的，而 zero、carry 則不是，但我的設計構想皆在 Behavioral model 上運行，因此我在 ALU 內多建立了一個 ALU1 的 Module 來實現功能，但題目預設輸出的 result 則是 reg 型態，因此在 ALU model 內需要再透過 wire 來轉型回 reg 型態。

ALU1 內定義了加法、減法、and、or、shift 的 result 與 carry 運算方式，carry 的定義如上所示。

2. Testbench

```
1. `timescale 1ns/ 1ps
2. module tb_ALU;
3. // inputs
4. reg [31:0] Src1;
5. reg [31:0] Src2;
6. reg [5:0] funct;
7. reg [4:0] shamt;
8.
9. // Outputs
10. wire [31:0]result;
11. wire zero;
12. wire carry;
13.
14. ALU uut(
15.     .Src1(Src1),
16.     .Src2(Src2),
17.     .funct(funct),
18.     .shamt(shamt),
19.     .result(result),
20.     .zero(zero),
21.     .carry(carry)
22. );
23.
24. initial begin
25.     Src1 = 0;
26.     Src2 = 0;
27.     funct = 0;
28.     shamt = 0;
29.
30. #10
31. Src1 = 32'd7;
32. Src2 = 32'd6;
33. funct = 6'd27;
34. // 7 + 6 = 13
35.
```

```

36. #10
37. funct = 6'd28;
38. // 7 - 6 = 1
39.
40. #10
41. funct = 6'd29;
42. // 7 and 6 = 6
43.
44. #10
45. funct = 6'd30;
46. // 7 or 6 = 7
47.
48. #10
49. funct = 6'd31;
50. shamt = 5'd3;
51. // 7 >> 3 = 0
52. // carry = 1
53.
54. #10
55. funct = 6'd32;
56. // 7 << 3 = 56
57.
58. // test carry
59. #20;
60. Src1 = 32'd4294967295;
61. Src2 = 32'd3;
62. funct = 6'd27;
63. // 4294967295 + 3 = 2
64. // carry = 1
65.
66. #10;
67. Src1 = 32'd3;
68. Src2 = 32'd4;
69. funct = 6'd28;
70. // 3-4 = -1 = 4294967295
71. // carry = 1
72.
73. #10;

```

```

74. Src1 = 32'd4294967295;
75. shamt = 32'd2;
76. funct = 6'd32;
77. // 4294967295 << 2 = 4294967292
78. // carry = 1
79.

80. #10;
81.
82. end
83. endmodule

```

Testbench 的實作方式為，每格 10ns 依序送入不同的指令與數字，預期的結果皆已於註解上。

3. Testbench 模擬結果圖

Signal	Value
Src1	4294967295
Src2	4
shamt	2
funct	32
zero	0
carry	1
Result	4294967292

4. 模擬結果分析

根據註解上標明，依序送入了加、減、乘、除、位移的數據，並測試 Carry 的功能皆達成了預期的結果。

(三) CompALU

1. 程式碼

```

1. module CompALU(
2.     input [31:0]instr,
3.     output [31:0] result,
4.     output zero,
5.     output carry
6. );
7. //input to RF
8. wire [4:0]Src1addr,Src2addr;
9.
10. //input to ALU
11. wire [5:0]shamt,funct;
12. wire [31:0]Src1,Src2;
13.

```

```

14. assign Src1addr[4:0] = instr[25:21];
15. assign Src2addr[4:0] = instr[20:16];
16. assign shamt[5:0] = instr[10:6];
17. assign funct[5:0] = instr[5:0];
18.
19. RF regi(Src1addr,Src2addr,Src1,Src2);
20. ALU alu(Src1,Src2,funct,shamt,result,zero,carry);
21.
22. endmodule

```

CompALU 模組依照題目的圖一設計，輸入為一個 instr 的 32bit 數值，透過 MIPS 的 ISA-R Format 把值分割成記憶體位址、運算符號、位移量...等功能。

2. Testbench

```

1. `timescale 1ns/ 1ps
2. module tb_CompALU;
3. //input
4. reg [31:0]instr;
5.
6. //output
7. wire [31:0] result;
8. wire zero;
9. wire carry;
10.
11. CompALU ca(instr,result,zero,carry);
12. initial begin
13.     instr[31:26] = 0;
14.     instr[25:21] = 13;
15.     instr[20:16] = 17;
16.     instr[15:11] = 0;
17.     instr[10:6] = 0;
18.     instr[5:0] = 27;
19.     // 13 + 17 = 30;
20.     #10;
21.
22.     instr[5:0] = 28;
23.     // 13 - 17 = 4294967292
24.     #10;

```

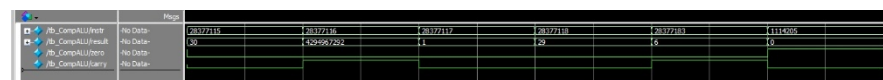
```

25.
26.     instr[5:0] = 29;
27.     // 13 and 17 = 1
28.     #10;
29.
30.     instr[5:0] = 30;
31.     // 13 or 17 = 29
32.     #10;
33.
34.     instr[5:0] = 31;
35.     instr[10:6] = 1;
36.     // 13 >> 1 = 6
37.     #10;
38.
39.     instr[5:0] = 29;
40.     instr[25:21] = 0;
41.     // 0 and 17 = 0
42.     #10;
43.
44. end
45. endmodule

```

Testbench 依序送入了不同的指令做運算，詳細的運算數字與結果皆表示於上述程式碼的註解當中。

3. Testbench 模擬結果圖



Signal	Value
No_Datap	28377115
No_Datap	28377116
No_Datap	28377117
No_Datap	28377118
No_Datap	28377119
No_Datap	1114506

4. 模擬結果分析

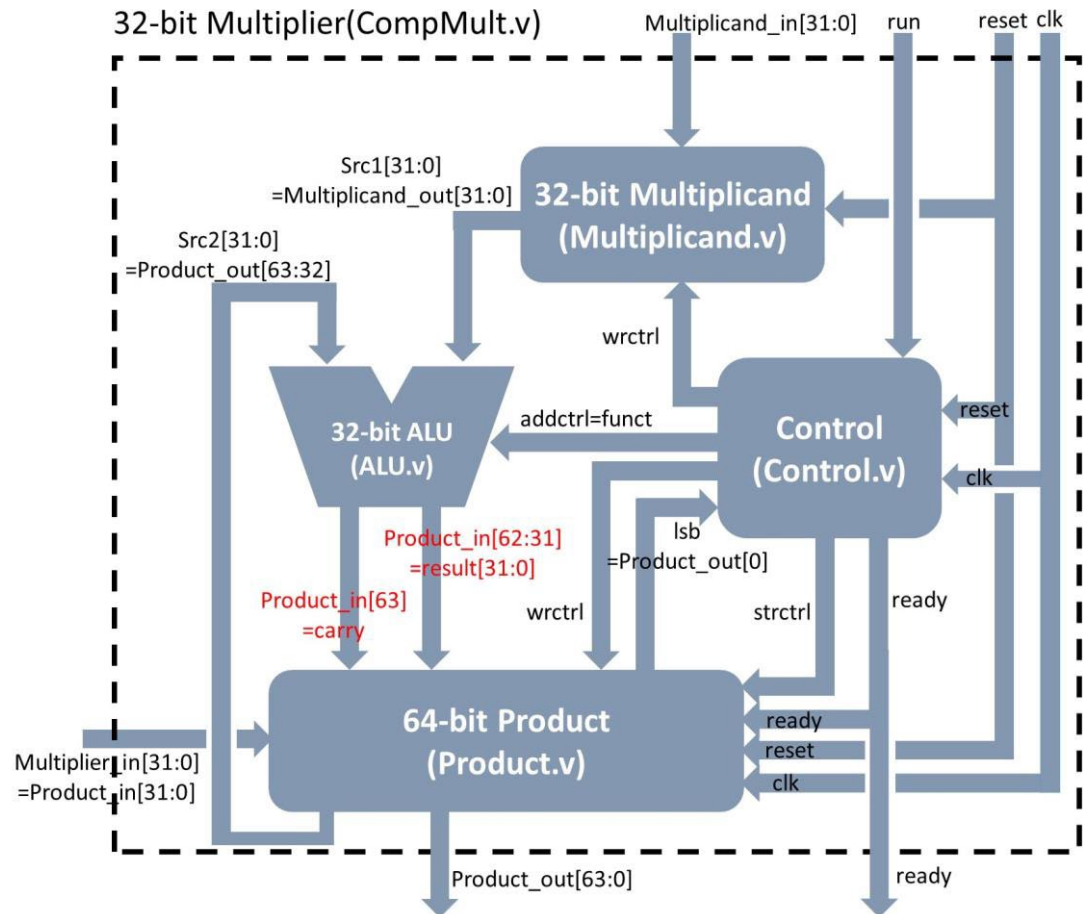
根據模擬結果，可以看出給予指定的 instruction，這個 CompALU 確實可以實作出數學的運算。

貳、Part2

一、題目

Implement a 32-bits multiplier

二、架構圖

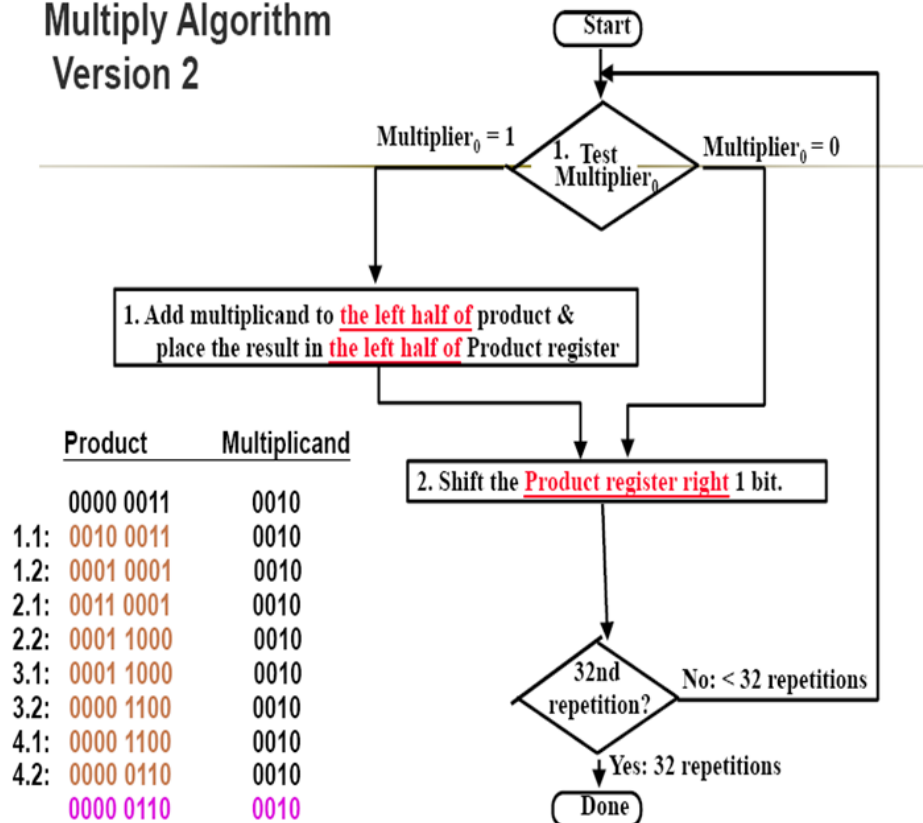


三、功能描述

rst 為 1 時，各信號初始化為 0。run 為 1，使 Multiplicand 以及 Multiplier 讀取輸入值 (mtplier_in 及 mtpcand_in) 進行運算，運算完畢 product 可讀時，設 ready 為 1。

四、演算法流程圖

Multiply Algorithm Version 2



五、程式碼

(一) CompMul

1. 程式碼

```

1. module CompMult(
2.     input [31:0] Multiplicand_in,
3.     input [31:0] Multiplier_in,
4.     input run,
5.     input reset,
6.     input clk,
7.     output ready,
8.     output [63:0] Product_out
9.     // output [31:0]Multiplicand_out
10.    // output [63:0] Product_inA
11.    // output [63:0]Product_in,
12.    // output [5:0]addctrl,
13.    // output strctrl,
14.    // output [31:0]counter
15.);

```

```

16.
17. wire strctrl;
18. wire wrctrl;
19. wire ze;
20. wire [5:0]addctrl;
21. wire [31:0]Multiplicand_out;
22. wire [63:0]Product_in;
23. wire [63:0] Product_inA;
24.
25. wire [32:0]alurest;
26.
27. assign Product_in[63:32] = alurest[32:1];
28. assign Product_in[31] = wrctrl ? Multiplier_in[31] :
    alurest[0];
29. assign Product_in[30:0] = Multiplier_in[30:0];
30.

31. Control ctrl(.run(run),
32.             .reset(reset),
33.             .clk(clk),
34.             .lsb(Product_out[0]),
35.             .ready(ready),
36.             .strctrl(strctrl),
37.             .wrctrl(wrctrl),
38.             .addctrl(addctrl)
39. );
40.
41. Multiplicand mul(.Multiplicand_in(Multiplicand_in),
42.                 .reset(reset),
43.                 .wrctrl(wrctrl),
44.                 .Multiplicand_out(Multiplicand_out)
45. );
46.
47. ALU alu(.Src1(Multiplicand_out),
48.         .Src2(Product_out[63:32]),
49.         .funct(addctrl),
50.         .shamt(5'd0),
51.         .result(alurest[31:0]),

```



```

52.         .zero(ze),
53.         .carry(alurestult[32])
54. );
55.
56. Product pro(.Product_in(Product_in),
57.             .wrctrl(wrctrl),
58.             .strctrl(strctrl),
59.             .ready(ready),
60.             .reset(reset),
61.             .clk(clk),
62.             .Product_out(Product_out)
63. );
64.
65. endmodule

```

此 CompMul 的 Module 依照題目架構圖的敘述建立而成，其中比較複雜的設計點為程式碼 27 至 29 行內的 assign 值，因為 Product 模組接收到的 Product_in 的值中，第 31 個 bit 是 ALU 輸出與 Multiplier_in 是共用的，且大多數時間下，皆為 ALU 的輸出值，僅有 wrctrl 為高電位(即程式剛起動，載入資料時)，會將 Multiplier 讀入。

2. Testbench

```

1. // 被乘數、乘數在第 51、52 行定義
2. `timescale 1ns/ 1ps
3. module tb_CompMult;
4. reg [31:0] Multiplicand_in;
5. reg [31:0] Multiplier_in;
6. reg run;
7. reg reset;
8. reg clk;
9.
10. wire ready;
11. wire [63:0] Product_out;
12. // wire wrctrl;
13. // wire [31:0]Multiplicand_out;
14. // wire [63:0] Product_inA;

```

```

15. // wire [63:0]Product_in;
16. // wire [5:0]addctrl;
17. // wire strctrl;
18. // wire [31:0]counter;
19.
20. CompMult cm(
21.     Multiplicand_in,
22.     Multiplier_in,
23.     run,
24.     reset,
25.     clk,
26.     ready,
27.     Product_out
28.     // Multiplicand_out
29.     // Product_inA
30.     // Product_in,
31.     // addctrl,
32.     // strctrl,
33.     // counter
34. );
35.
36. integer i;
37. initial begin
38.     clk <= 0;
39.     reset <= 0;
40.     run <= 0;
41.     Multiplicand_in <= 0;
42.     Multiplier_in <= 0;
43.
44.     #1
45.     clk = 1;
46.
47.     // initial
48.     reset <= 1;
49.     run <= 0;
50.     Multiplicand_in <= 32'd15;
51.     Multiplier_in <= 32'd19;
52.

```

```

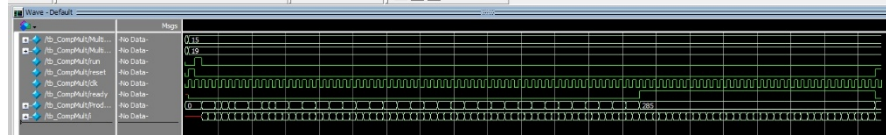
53.     #1
54.     clk = 0;
55.     #1
56.     clk = 1;
57.
58.     reset <= 0;
59.     run <= 1;
60.
61.     #1
62.     clk = 0;
63.     #1
64.     clk = 1;
65.
66.     run <= 0;
67.
68.     for(i=0;i<100;i=i+1)begin
69.         #1
70.         clk = 0;
71.         #1
72.         clk = 1;
73.     end
74.
75.     reset <= 1;
76.
77.     #1
78.     clk = 0;
79.     #1
80.     clk = 1;
81.
82. end
83.
84. endmodule

```

Testbench 中先於第 50、51 行的位置指定好被乘數與乘數，並且拉低 reset 電位，提高 run 電位 1 個 clock 後降低，程式開始運算，透過一個 for 迴圈不斷的送 clock 後計算完畢，計算完畢後可於 Product_out 顯示出計算值，並且使

ready 為高電位。當計算完畢後可以送入 reset 訊號把數值歸零並重新運算。

3. Testbench 模擬結果圖



4. 模擬結果分析

輸入被乘數為 15、乘數為 19，經過多個 Clock 運算後可取得計算結果為 285。

(二) Product

1. 程式碼

```
1. module Product(  
2.     input [63:0] Product_in,  
3.     input wrctrl,  
4.     input strctrl,  
5.     input ready,  
6.     input reset,  
7.     input clk,  
8.     output reg[63:0] Product_out  
9. );  
10. reg [64:0]Product_reg;  
11. reg a;  
12. initial begin  
13.     Product_reg = 0;  
14.     Product_out = 0;  
15.     a = 0;  
16. end  
17.  
18. always @(posedge clk or posedge reset)begin  
19.     if(reset)begin  
20.         Product_reg = 0;  
21.         Product_out = 0;  
22.     end  
23.     if(!ready)begin  
24.         if(wrctrl)begin // load data and start  
25.             Product_reg[31:0] = Product_in[31:0];
```

```

26.         end
27.
28.
29.         else if(strctrl && a == 0) begin //收加法
30.             Product_reg[64] = Product_in[63];
31.             Product_reg[63:32] = Product_in[62:31];
32.             a = 1;
33.         end
34.
35.         //shift
36.         else if(!strctrl && !reset && !wrctrl && clk &&
a == 1)begin
37.             Product_reg = (Product_reg >> 1);
38.             a = 0;
39.         end
40.
41.     end
42.
43.     Product_out[63:0] = Product_reg[63:0];
44.
45. end
46. endmodule

```

Product 程式碼中，當 wrctrl 為高電位時，將資料載入 Product_reg，當收到 strctrl 訊號時，接收 ALU 所送來的加法數值，並設定暫存器 a = 1，透過暫存器 a 作為狀態機使用，由不同的 clock 可以判斷出此時需要做接收加法的動作或是位移的動作。

2. Testbench

```

1. `timescale 1ns/ 1ps
2. module tb_Product;
3.     reg [63:0] Product_in;
4.     reg wrctrl;
5.     reg strctrl;
6.     reg ready;
7.     reg reset;
8.     reg clk;
9.     wire[63:0] Product_out;

```

```

10.
11. Product pro(
12.     Product_in,
13.     wrctrl,
14.     strctrl,
15.     ready,
16.     reset,
17.     clk,
18.     Product_out
19. );
20.
21. initial begin
22.
23.     Product_in[63:32] <= 32'd0;
24.
25.     //////////////////////////////////////
26.     // 初始訊號
27.     clk <= 1;
28.     reset <= 1;
29.     wrctrl <= 0;
30.     strctrl <= 0;
31.     ready <= 0;
32.     Product_in[31:0] <= 32'b0;
33.     // out = 0
34.
35.     #1
36.     clk <= 0;
37.     #1
38.     clk <= 1;
39.
40.     // load multiplier
41.     reset <= 0;
42.     wrctrl <= 1;
43.     Product_in[31:0] <= 32'b11;
44.     //out = 11
45.
46.     #1

```

```

47.    clk <= 0;
48.    #1
49.    clk <= 1;
50.
51.    wrctrl <= 0;
52.    if(Product_out) begin
53.        strctrl <= 1;
54.        Product_in[62:31] <= 32'b1111 +
Product_out[63:32];
55.        Product_in[63] <= 0;
56.    end
57.    // 加法
58.    // out 左半 1111 + 0000 右半 1111
59.
60.    #1
61.    clk <= 0;
62.    #1
63.    clk <= 1;
64.
65.    strctrl <= 0;
66.    //位移
67.
68.
69.    #1
70.    clk <= 0;
71.    #1
72.    clk <= 1;
73.
74.    if(Product_out) begin
75.        strctrl <= 1;
76.        Product_in[62:31] <= 32'b1111 +
Product_out[63:32];
77.        Product_in[63] <= 0;
78.    end
79.    //加法
80.
81.    #1
82.    clk <= 0;

```

```

83.     #1
84.     clk <= 1;
85.
86.     strctrl <= 0;
87.     //位移
88.
89.     #1
90.     clk <= 0;
91.     #1
92.     clk <= 1;
93.
94.     if(Product_out) begin
95.         strctrl <= 1;
96.         Product_in[62:31] <= 32'b1111 +
Product_out[63:32];
97.         Product_in[63] <= 0;
98.     end
99.     //加法
100.
101.     #1
102.     clk <= 0;
103.     #1
104.     clk <= 1;
105.
106.     strctrl <= 0;
107.     //位移
108.
109.     #1
110.     clk <= 0;
111.     #1
112.     clk <= 1;
113.
114.     if(Product_out) begin
115.         strctrl <= 1;
116.         Product_in[62:31] <= 32'b1111 +
Product_out[63:32];
117.         Product_in[63] <= 0;

```



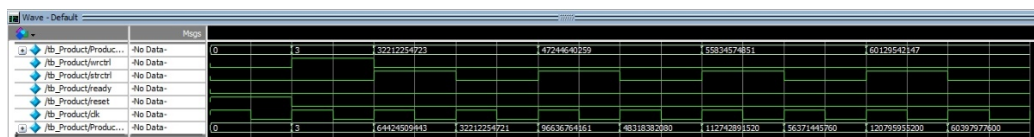
```

118.     end
119.     //加法
120.
121.     #1
122.     clk <= 0;
123.     #1
124.     clk <= 1;
125.
126.     strctrl <= 0;
127.     //位移
128.
129.     #1
130.     clk <= 0;
131.     #1
132.     clk <= 1;
133.
134. end
135.
136. endmodule

```

在 Testbench 中剛開始先載入了初始訊號，透過 wrctrl 信號將 Product_in 載入模組中，接下來透過交替 clock 與 strctrl 的方法模擬加法與位移，詳細動作流程已於上方程式註解處敘述。

3. Testbench 模擬結果圖



4. 模擬結果分析

透過觀察信號模擬的結果，可以看出輸出信號確實有依照上方 tb 的預期內容實行載入數字、加法、位移等動作。

(三) Control

1. 程式碼

```

1. module Control(
2.     input run,

```

```

3.     input reset,
4.     input clk,
5.     input lsb,
6.     output reg ready,
7.     output reg strctrl,
8.     output reg wrctrl,
9.     output reg [5:0]addctrl
10. );
11. reg runs,a;
12.
13. integer counter;
14.
15. initial begin
16.     runs = 0;
17.     a = 0;
18.     counter = 0;
19.     strctrl = 0;
20. end
21.
22. always @(lsb)begin
23.     addctrl = lsb ? 5'd27 : 5'd0;
24. end
25.
26. always @(posedge run)begin
27.     if (run && clk)begin
28.         runs = 1;
29.         wrctrl = 1;
30.     end
31. end
32.
33. always @(posedge reset)begin
34.     if(reset)begin
35.         ready <= 0;
36.         strctrl <= 0;
37.         wrctrl <= 0;
38.         runs <= 0;
39.     end
40. end

```

```

41.
42. always @(posedge clk)begin
43.     if (runs == 1)begin
44.         if(wrctrl == 1)begin
45.             wrctrl = 0;
46.         end
47.         else if(a == 0 && counter < 32)begin // test mul
48.             strctrl = 1;
49.             a = 1;
50.         end
51.
52.         else if (a == 1)begin //shift
53.             strctrl = 0;
54.             a = 0;
55.             counter = counter + 1;
56.         end
57.
58.         else begin // 終止條件
59.             runs <= 0;
60.             ready <= 1;
61.             counter <= 0;
62.         end
63.     end
64.
65. end
66.
67. endmodule

```

Control Module 的程式碼可以算是整個乘法器的控制中樞，接收到 run 信號時，需要觸發 wrctrl 使被乘數與乘數被載入。當收到 reset 信號時，也需要把 ready、strctrl、wrctrl 給全部重置。當城市開始執行時，需要不斷的測試 lsb，也就是 Product_out[0] 的訊號，由此決定是否需要做加法。判斷完畢後再發出位移訊號。透過一個 conter 來判斷何時該進入終止條件。

2. Testbench

```
1. `timescale 1ns/ 1ps
2.
3. module tb_Control;
4. reg run;
5. reg reset;
6. reg clk;
7. reg lsb;
8.
9. wire ready;
10. wire strctrl;
11. wire wrctrl;
12. wire [5:0]addctrl;
13.
14. Control ctrl(
15.     run,
16.     reset,
17.     clk,
18.     lsb,
19.     ready,
20.     strctrl,
21.     wrctrl,
22.     addctrl
23. );
24.
25. integer i;
26. initial begin
27.     // 初始狀態
28.     clk <= 1;
29.     run <= 0;
30.     reset <= 1;
31.     lsb <= 0;
32.     // 全部輸出都 0
33.
34.     //0
35.     #1
36.     clk <= 0;
37.     #1
```

```
38.    clk <= 1;
39.
40.    //2
41.    reset <= 0;
42.    run <= 1;
43.    // 開始
44.    // 載入資料
45.    // wrctrl = 1
46.
47.    #1
48.    clk <= 0;
49.    #1
50.    clk <= 1;
51.
52.    //4
53.    run <= 0;
54.    // load data done
55.    // wrctrl = 0
56.
57.    #1
58.    clk <= 0;
59.    #1
60.    clk <= 1;
61.
62.    //6
63.    lsb <= 1;
64.    // 判斷 lsb 做加法
65.    // str = 1 add = 27
66.
67.
68.    #1
69.    clk <= 0;
70.    #1
71.    clk <= 1;
72.
73.    //8
74.    // nothing
```

```
75.    // shift (str = 0)
76.    // 1
77.
78.    #1
79.    clk <= 0;
80.    #1
81.    clk <= 1;
82.
83.    //10
84.    // nothing
85.    // add (str = 1, add = 27)
86.
87.    #1
88.    clk <= 0;
89.    #1
90.    clk <= 1;
91.
92.    //12
93.    // nothing
94.    // shift (str = 0, add = 0)
95.    // 2
96.
97.    #1
98.    clk <= 0;
99.    #1
100.   clk <= 1;
101.
102.   //14
103.   lsb <= 0;
104.   // no add (str = 1 , add = 0)
105.
106.   #1
107.   clk <= 0;
108.   #1
109.   clk <= 1;
110.
111.   //16
```

```

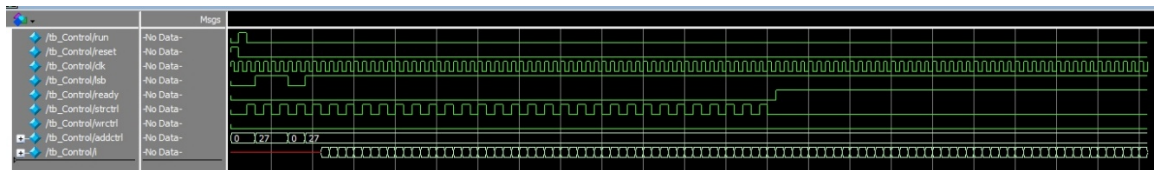
112.    //nothing
113.    // shift (str=0,add=0)
114.    // 3
115.
116.    #1
117.    clk <= 0;
118.    #1
119.    clk <= 1;
120.
121.    //18
122.    lsb <= 1;
123.    // add (str = 1 , add = 27)
124.
125.    #1
126.    clk <= 0;
127.    #1
128.    clk <= 1;
129.
130.    //20
131.    //nothing
132.    //shif(str = 0, add = 0)
133.
134.    //done!!!! so ready = 1
135.
136.
137.    #1
138.    clk <= 0;
139.    #1
140.    clk <= 1;
141.
142.    for(i=0;i<100;i=i+1)begin
143.        #1
144.        clk <= 0;
145.        #1
146.        clk <= 1;
147.    end
148.    //測試 ready
149. end

```

```
150.
151.endmodule
```

本 Testbench 模擬了原本的初始狀態、reset、run 等各種信號的輸入，測試了透過 lsb 觸發是否進行加法的動作，以及需要位移的時機，最後透過一個迴圈來測試運算完畢後會使 run 拉為高電位，動作過程詳見程式碼中的敘述。

3. Testbench 模擬結果圖



4. 模擬結果分析

透過此模擬結果可以看出，此 Model 可以接收到 run、reset 的訊號，並透過 lsb 的訊號判斷出該次的執行過程是否需要進行加法，並統一做位移的動作，直到運算完畢後，reset 會為 1。

(四) Multiplicand

1. 程式碼

```
1. module Multiplicand(
2.     input [31:0]Multiplicand_in,
3.     input reset,
4.     input wrctrl,
5.     output reg [31:0] Multiplicand_out
6. );
7.
8. reg [31:0]Multiplicand_out_reg;
9. initial begin
10. Multiplicand_out_reg = 0;
11. end
12. always @(reset or wrctrl)begin
13.
14.     if(reset)
15.         Multiplicand_out_reg = 32'd0;
16.     else if (wrctrl)
17.         Multiplicand_out_reg = Multiplicand_in;
```



```

18.
19.     Multiplicand_out = Multiplicand_out_reg;
20.
21. end
22.
23. endmodule

```

Multiplicand model 的功能是全部裡面最單純的，只負責接收輸入的資料並暫存，收到 reset 訊號後將資料歸零。

2. Testbench

```

1. `timescale 1ns/ 1ps
2. module tb_Multiplicand;
3.
4.     // input
5.     reg [31:0]Multiplicand_in;
6.     reg reset;
7.     reg wrctrl;
8.
9.     // output
10.    wire [31:0] Multiplicand_out;
11.
12.    Multiplicand mul(
13.        Multiplicand_in,
14.        reset,
15.        wrctrl,
16.        Multiplicand_out
17.    );
18.
19.    initial begin
20.        Multiplicand_in = 0;
21.        reset = 0;
22.        wrctrl = 0;
23.        #10
24.        // zzzzzz
25.
26.        Multiplicand_in = 32'd123;
27.        wrctrl = 1;

```

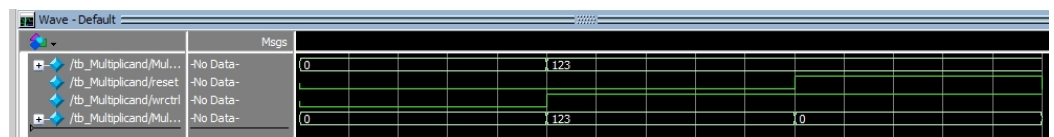
```

28.    #10
29.    // 123
30.
31.    reset = 1;
32.    // 00000
33.
34.    #10
35.    reset = 0;
36.
37. end
38.
39. endmodule

```

Testbench 測試將資料由 Multiplicand_in 讀入，此時可使用 Multiplicand_out 做輸出，並使用 reset 清空。

3. Testbench 模擬結果圖



4. 模擬結果分析

經過測試，Multiplicand 可以順利的讀取資料，送出給 ALU，也可以 reset。

(五) ALU

1. 程式碼

```

1. module ALU(
2.     input [31:0] Src1,
3.     input [31:0] Src2,
4.     input [5:0] funct,
5.     input [4:0] shamt,
6.     output reg [31:0] result,
7.     output zero,
8.     output carry
9. );
10. wire [31:0] result1;
11.
12. initial begin

```

```

13.  result = 0;
14.  end
15.
16.  // when result=0 , zero = 1
17.  assign zero = result == 0 ? 1:0;
18.
19.  ALU1 alu(
20.    Src1,
21.    Src2,
22.    funct,
23.    shamt,
24.    result1,
25.    carry
26.  );
27.  always @(result1)begin // 因為題目的 result 需要是 reg
28.    result = result1;
29.  end
30.
31. endmodule
32.
33. module ALU1(
34.   input [31:0] Src1,
35.   input [31:0] Src2,
36.   input [5:0] funct,
37.   input [4:0] shamt,
38.   output reg[31:0] result,
39.   output reg carry //因為題目 carry 沒有 reg，所以用另外
    一個 model 來連
40. );
41.
42. always@(Src1 or Src2 or funct or shamt)begin
43.   case (funct[5:0])
44.     6'd27: {carry,result} <= Src1 + Src2;
45.     6'd28: {carry,result} <= Src1 - Src2;
46.     6'd29: {carry,result} <= Src1 & Src2;
47.     6'd30: {carry,result} <= Src1 | Src2;
48.     6'd31: begin
49.       result <= (Src1 >> shamt);

```

```

50.         carry <= Src1[shamt-1];
51.         end
52.         6'd32: {carry,result} <= (Src1 << shamt);
53.         default: {carry,result} <= {1'b0,Src2};
54.     endcase
55. end
56.
57. endmodule

```

由於題目的 result 需求是 reg 的，而 zero、carry 則不是，但我的設計構想皆在 Behavioral model 上運行，因此我在 ALU 內多建立了一個 ALU1 的 Module 來實現功能，但題目預設輸出的 result 則是 reg 型態，因此在 ALU model 內需要再透過 wire 來轉型回 reg 型態。

並且由於考慮到 Control 程式 data path 的一致性，在程式碼的第 53 行新增一個例外情形，也就是非規定的 27 至 31 function Code 時，就會將第二個輸入的值回傳回來，方便乘法的流程進行。

2. Testbench

```

1. `timescale 1ns/ 1ps
2. module tb_ALU;
3.     // inputs
4.     reg [31:0] Src1;
5.     reg [31:0] Src2;
6.     reg [5:0] funct;
7.     reg [4:0] shamt;
8.
9.     // Outputs
10.    wire [31:0]result;
11.    wire zero;
12.    wire carry;
13.
14.    ALU uut(
15.        .Src1(Src1),
16.        .Src2(Src2),
17.        .funct(funct),
18.        .shamt(shamt),
19.        .result(result),

```

```

20.     .zero(zero),
21.     .carry(carry)
22. );
23.
24. initial begin
25.     Src1 = 0;
26.     Src2 = 0;
27.     funct = 0;
28.     shamt = 0;
29.
30. #10
31. Src1 = 32'd7;
32. Src2 = 32'd6;
33. funct = 6'd27;
34. // 7 + 6 = 13
35.
36. #10
37. funct = 6'd28;
38. // 7 - 6 = 1
39.
40. #10
41. funct = 6'd29;
42. // 7 and 6 = 6
43.
44. #10
45. funct = 6'd30;
46. // 7 or 6 = 7
47.
48. #10
49. funct = 6'd31;
50. shamt = 5'd3;
51. // 7 >> 3 = 0
52. // carry = 1
53.
54. #10
55. funct = 6'd32;

```

```
56. // 7 << 3 = 56
57.
58. // test carry
59. #20;
60. Src1 = 32'd4294967295;
61. Src2 = 32'd3;
62. funct = 6'd27;
63. // 4294967295 + 3 = 2
64. // carry = 1
65.
66. #10;
67. Src1 = 32'd3;
68. Src2 = 32'd4;
69. funct = 6'd28;
70. // 3-4 = -1 = 4294967295
71. // carry = 1
72.
73. #10;
74. Src1 = 32'd4294967295;
75. shamt = 32'd2;
76. funct = 6'd32;
77. // 4294967295 << 2 = 4294967292
78. // carry = 1
79.
80. #10;
81.
82. end
83. endmodule
```

Testbench 的實作方式為，每格 10ns 依序送入不同的指令與數字，預期的結果皆已於註解上。

3. Testbench 模擬結果圖

[illegible]

4. 模擬結果分析

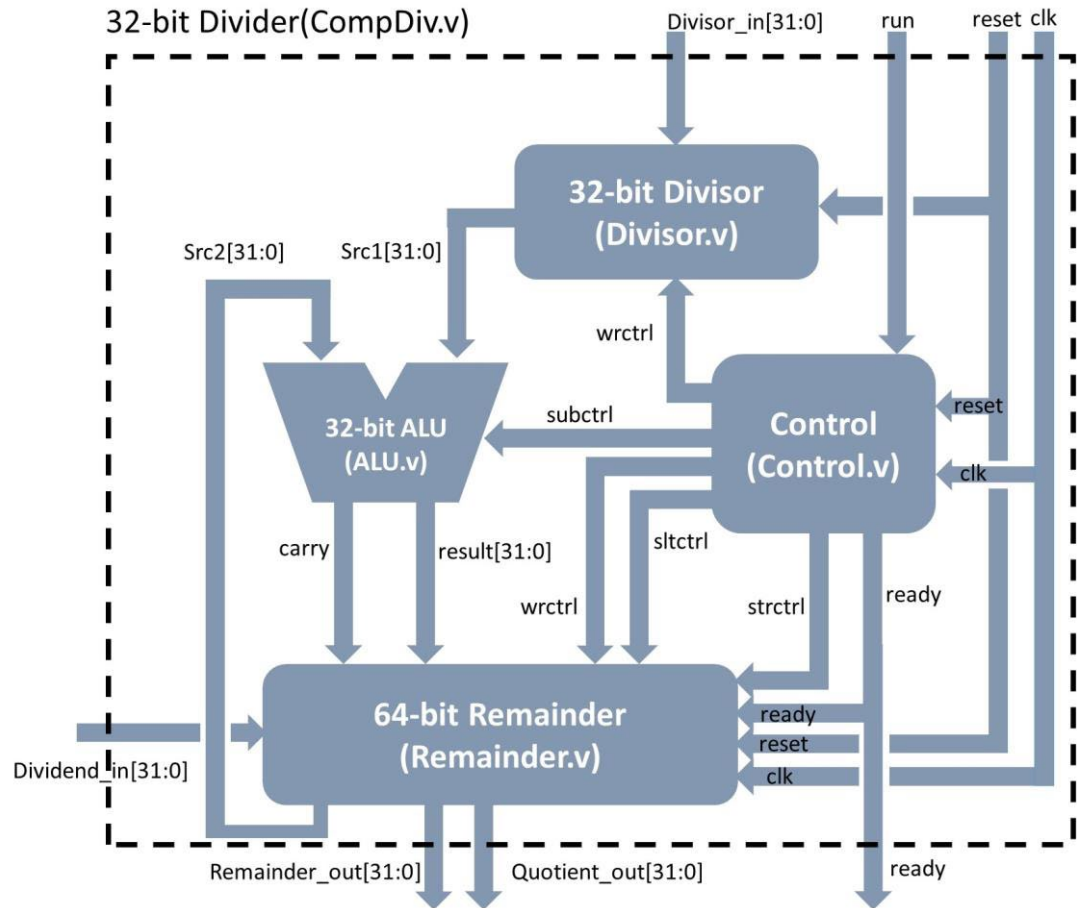
根據註解上標明，依序送入了加、減、乘、除、位移的數據，並測試 Carry 的功能皆達成了預期的結果。

參、Part3

一、題目

Implement a 32-bits divider

二、架構圖



三、功能描述

實作一個除法器，除了 `CompDiv` 外，其他模組可自行定義，但需在報告中詳細說明，內部必須有 `ALU.v`、`Remainder.v`、`Divisor.v`、`Control.v` 幾個檔案。

四、I/O 定義

```
1. module CompDiv(  
2.     input [31:0] Divisor_in,  
3.     input [31:0] Dividend_in,  
4.     input run,  
5.     input reset,  
6.     input clk,  
7.     output ready,  
8.     output [31:0] Quotient_out,
```



```
9.      output [31:0] Remainder_out  
10. )
```

五、程式碼
未完成

肆、心得

在開發的過程中，我發覺 ModelSim 程式拿來做開發的效率非常不佳，軟體介面也有一點跟不上時代的感覺，經過了多方的詢問與查詢，發現 ModelSim 其實也有支援 Command Line 的運作方式，可以直接下指令，主要參考了「ModelSim® Command Reference Manual」文件的內容，在 ModelSim 的 IDE 外面做編譯與模擬的動作。因此我最終的開發介面使用了 Visual Studio Code，透過這種方式感覺開發的效率增加了非常的多！

在實作的過程中，讓我真正的了解到了 MIPS 指令中 ALU 在背後的計算方式，經過了明確的實作，就不會有教科書永遠只在講理論，永遠都在紙上談兵的感覺，經過自己開發了這次專案後，我也對各晶片底層的開發者趕到了深深的敬佩，一個簡單的乘法除法在系統底層竟然也運行的如此複雜。

剛開始選這堂課的時候，就聽說到了這堂課的 Project 非常的硬，內容非常的複雜與困難，由於我原本對於 Verilog 的了解程度非常有限，也很擔心無法成功完成這次的作業，經過了幾天徹夜的研究後，終於順利的完成了這一個 Project，感謝老師與助教在課堂上的協助，也感謝這次 Project 實作過程中與我討論的同學們。