

計算機組織 PA1

電機三甲 B11107056 黃敏聖

Part1 :

```
47     Chip area for module '\CompMultiplier': 1366.442000
48         of which used for sequential elements: 388.360000 (28.42%)
609 finish critical path slack
610 -----
611 4.5072|-----|
612
```

Chip Area : 1366. Worst Slack : 4.507

Part2 :

```
49     Chip area for module '\CompDivider': 1489.600000
50         of which used for sequential elements: 403.522000 (27.09%)
617 finish critical path slack
618 -----
619 4.4213|-----|
```

Chip Area : 1489. Worst Slack : 4.421

壹、Multiplier

一、程式碼

CompMultiplier:

```
1  module CompMultiplier(
2      // Outputs
3      output [63:0]Product,
4      output Ready,
5      // Inputs
6      input [31:0] Multiplicand,
7      input [31:0] Multiplier,
8      input Run,
9      input Reset,
10     input clk
11 );
12   wire [31:0] Multiplicand_out, product_out, ALU_result;
13   wire W_ctrl, Addu_ctrl, LSB, SRL_ctrl, ALU_carry;
14
15   assign product_out = Product[63:32];
16   assign LSB = Product[0];
17
18   Multiplicand_Register m1(
19     .Reset(Reset),
20     .W_ctrl(W_ctrl),
21     .Multiplicand_in(Multiplicand),
22     .Multiplicand_out(Multiplicand_out)
23 );
24   ALU a1(
25     .Src_1(product_out), //product out from Product_Register
26     .Src_2(Multiplicand_out), //Multiplicand
27     .Addu_ctrl(Addu_ctrl), //this exp use 1bit to reduce area
28     .ALU_carry(ALU_carry),
29     .ALU_result(ALU_result)
30 );
31   Control c1(
32     .Run(Run),
33     .Reset(Reset),
34     .clk(clk),
35     .LSB(LSB), //LSB is Product[0] from Product_Register
36     .W_ctrl(W_ctrl),
37     .SRL_ctrl(SRL_ctrl),
38     .Ready(Ready),
39     .Addu_ctrl(Addu_ctrl) //this exp use 1bit to reduce area
40 );
41   Product_Register p1(
42     .SRL_ctrl(SRL_ctrl),
43     .W_ctrl(W_ctrl),
44     .Reset(Reset),
45     .clk(clk),
46     .ALU_carry(ALU_carry),
47     .ALU_result(ALU_result),
48     .Multiplier_in(Multiplier),
49     .Product(Product)
50 );
51
52 endmodule
```

將每個模組連接在一起，將最低有效位 (LSB) 連接到輸出的最低位元，而 Product out 則連接到輸出的上半部分。

Multiplicand:

```
≡ Multiplicand.v
1 module Multiplicand_Register (
2     input Reset,
3     input W_ctrl,
4     input [31:0] Multiplicand_in,
5     output reg [31:0] Multiplicand_out
6 );
7     always @(*) begin
8         if (Reset) begin
9             Multiplicand_out = 32'd0; //reset
10        end
11        else if (W_ctrl) begin
12            Multiplicand_out = Multiplicand_in; //load
13        end
14        else begin
15            Multiplicand_out = Multiplicand_out; // not change
16        end
17    end
18 endmodule
```

當 W_ctrl 為 1 時，將 in 的數據加載到 out，當 Reset 為 1 時，out 會被重置；在其他情況下，out 的值保持不變。

ALU:

```
≡ ALU.v
1 module ALU (
2     input [31:0] Src_1, //product out from Product_Register
3     input [31:0] Src_2, //Multiplicand
4     input Addu_ctrl, //this exp use 1bit to reduce area
5     output reg ALU_carry,
6     output reg [31:0] ALU_result
7 );
8     always @(*) begin
9         if (Addu_ctrl) begin
10            {ALU_carry , ALU_result} = Src_1 + Src_2; // add
11        end
12        else begin
13            ALU_carry = 0;
14            ALU_result = Src_1;
15        end
16    end
17 endmodule
```

當 Addu_ctrl 為 1 時，將兩個輸入的數值相加，而在其餘情況下，則將 Src_1 (即 Product out) 的值傳遞到輸出。

Product:

```
≡ Product.v
1 module Product_Register (
2     input SRL_ctrl, W_ctrl, Reset, clk, ALU_carry,
3     input [31:0] ALU_result, Multiplier_in,
4     output reg [63:0] Product
5 );
6     always @(posedge clk or posedge Reset) begin
7         if (Reset) begin
8             Product <= 0;
9         end
10        else if (W_ctrl) begin
11            Product[63:32] <= ALU_result;
12            Product[31:0] <= Multiplier_in;
13        end
14        else if (SRL_ctrl) begin
15            Product[63:32] <= {ALU_carry , ALU_result[31:1]}; //use nonblocking shift
16            Product[31:0] <= {ALU_result[0] , Product[31:1]};
17        end
18        else begin
19            Product <= Product;
20        end
21    end
22 endmodule
```

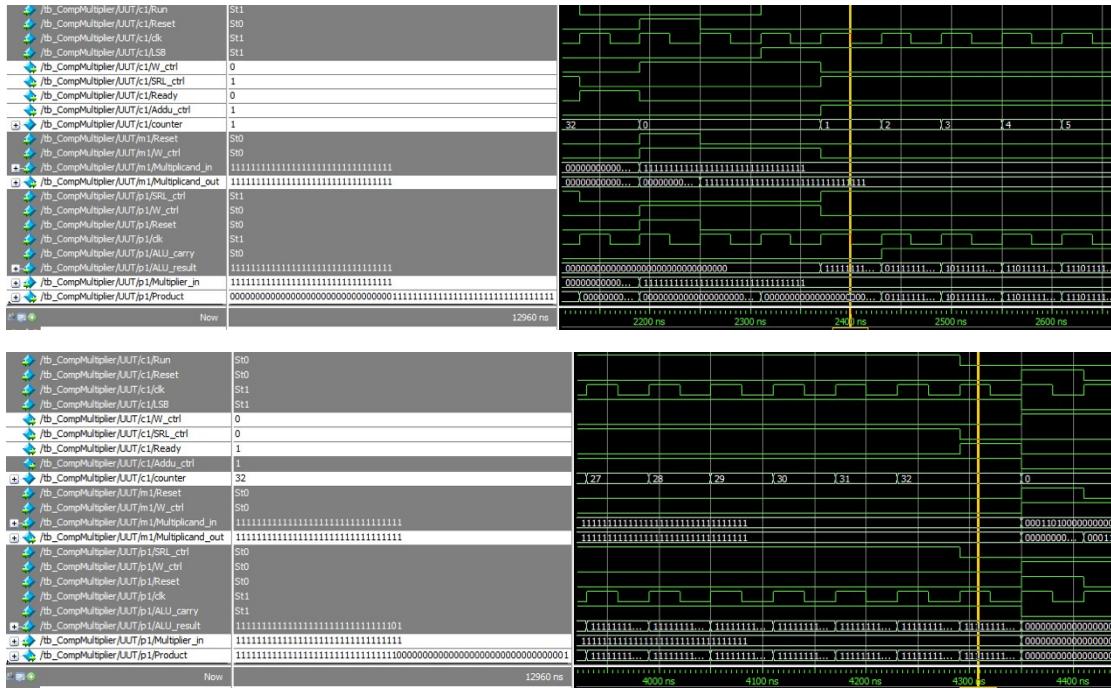
Reset 時重置輸出，W_ctrl 為 1 時 load，SRL_ctrl 為 1 時做 shift，用 nonblocking 的方式，因為時序電路最好是採用同步的設計方式

Control:

```
1 module Control (
2     input Run, Reset, clk,
3     input LSB, //LSB is Product[0] from Product_Register
4     output reg W_ctrl, SRL_ctrl, Ready,
5     output reg Addu_ctrl //this exp use 1bit to reduce area
6 );
7     reg [5:0] counter;
8
9     always @(posedge clk or posedge Reset) begin
10         if (Reset) begin
11             counter <= 0;
12             W_ctrl <= 1; // when reset high Multiplicand and Multiplier alr
13             SRL_ctrl <= 0;
14             Ready <= 0;
15         end
16         else if (Run) begin
17             W_ctrl <= 0;
18             if (counter <= 6'd31) begin
19                 SRL_ctrl <= 1;
20                 counter <= counter + 1;
21             end
22             else begin
23                 Ready <= 1;
24                 SRL_ctrl <= 0;
25             end
26         end
27     end
28
29     always @(*) begin
30         if (~W_ctrl && LSB) begin//Avoid add once more
31             Addu_ctrl = 1;
32         end
33         else begin
34             Addu_ctrl = 0;
35         end
36     end
37 endmodule
```

當 Reset 信號啟動時，會將 W_ctrl 設為 1，其他控制信號設為 0，以確保重置後的初始化狀態。當 Run 為 1 時，開始計算 counter 並執行位移操作，該過程會持續 32 個時鐘週期。Addu_ctrl 會通過組合電路來確保 LSB 能及時反應是否需要進行加法操作，而 \sim W_ctrl 的作用是防止在計數器尚未開始計數時進行加法運算，確保加法操作只會在正確的時間點執行。這樣的設計能夠保證電路的同步性與正確性，避免未經過計數器的加法操作。

二、 testbench



在當 counter 為 0 時，乘數和被乘數會載入各自的暫存器。當 counter 為 1 時，將各控制信號設為 1，此時 ALU 已經決定好輸出結果。隨著 counter 從 1 進到 2 的瞬間，會開始執行 shift 操作，並持續進行，直到 counter 達到 32。在 counter 為 32 時，會進行最後一次的 shift 操作，並且 Ready 信號變為為 1，完成整個運算過程。這樣的設計可以保證運算過程中的每一步都按時執行，並確保結果在最後一個時鐘週期正確產生。

三、測資

```
# Multiplicand:      1003, Multiplier:      20
# result:           20060
# Multiplicand:4294967295, Multiplier:4294967295
# result:18446744065119617025
# Multiplicand: 436225289, Multiplier:      10553
# result:        4603485474817
# Multiplicand: 19080257, Multiplier: 179027151
# result:       3415884051057807
# Multiplicand:      1, Multiplier:      1
# result:           1
# Multiplicand:     1343, Multiplier:      8
# result:          10744
```

	testbench > tb_CompMultiplier.in	testbench > tb_CompMultiplier.out
1	000003EB_00000014	0000000000004e5c
2	ffffffff_ffffffff	fffffffffffe00000001
3	1a004509_00002939	0000042fd51dd001
4	01232441_0aabccf	000c22ba7b750c8f
5	00000001_00000001	0000000000000001
6	00000053f_00000008	00000000000029f8

	被乘數	乘數	積
1.	32 ' h000003eb	32 ' h00000014	64 ' h00000000000000004e5c
2.	32 ' hfffffff	32 ' hfffffff	64 ' hffffffe00000001
3.	32 ' h1a004509	32 ' h00002939	64 ' h00000042fd51dd001
4.	32 ' h01232441	32 ' h0aabccf	64 ' h000c22ba7b750c8f
5.	32 ' h00000001	32 ' h00000001	64 ' h00000000000000000001
6.	32 ' h00000053f	32 ' h00000008	64 ' h000000000000000029f8

第一個測資是預設的，我沒有進行任何修改。第二個測資是 32-bit 乘法器處理的最大案例，這是為了測試乘法器處理極端情況的能力。第三個和第四個測資則是隨機生成的數字，用來測試在運算過程中會涉及到 carry 的情況，這樣可以檢查乘法器在進行進位處理時的準確性。第五個測資是為了測試最小的情況，主要是檢查乘法器在處理極小數值時的運算結果。第六個測資是隨機設置的數字，這樣可以進一步測試乘法器對於不同數字組合的處理能力。這些測資的輸出結果都與我預想的相符。第二個測資我無法用計算機直接得出結果，因此我選擇手算來檢查結果；而第三和第四個測資，我則是使用計算機計算並觀察能顯示的位數，然後根據尾數來核對計算是否正確。例如，對於測資三， $9*3=27$ ，輸出的尾數為 7，這是正確的，符合預期的結果。這樣的測試方式確保了乘法器在各種情境下的準確性與穩定性。

貳、 Divider

一、 程式碼

CompDivider:

```
CompDivider.v
1 module CompDivider(
2     // Outputs
3     output [31:0] Quotient,
4     output [31:0] Remainder,
5     output Ready,
6     // Inputs
7     input [31:0] Dividend,
8     input [31:0] Divisor,
9     input Run,
10    input Reset,
11    input clk
12 );
13 wire [31:0] ALU_result, Divisor_out;
14 wire ALU_carry, W_ctrl, SLL_ctrl, SRL_ctrl, subu_ctrl;
15 Divisor_Register d1(
16     .Reset(Reset),
17     .W_ctrl(W_ctrl),
18     .Divisor_in(Divisor),
19     .Divisor_out(Divisor_out)
20 );
21 ALU a1 (
22     .Src_1(Remainder), //remainder out from Remainder_Register
23     .Src_2(Divisor_out), //divisor
24     .subu_ctrl(subu_ctrl), //this exp use 1bit to reduce area
25     .ALU_carry(ALU_carry),
26     .ALU_result(ALU_result)
27 );
28 Control c1(
29     .Run(Run),
30     .Reset(Reset),
31     .clk(clk),
32     .W_ctrl(W_ctrl),
33     .SLL_ctrl(SLL_ctrl),
34     .SRL_ctrl(SRL_ctrl),
35     .Ready(Ready),
36     .subu_ctrl(subu_ctrl) //this exp use 1bit to reduce area
37 );
38 Remainder_Register r1(
39     .SLL_ctrl(SLL_ctrl),
40     .SRL_ctrl(SRL_ctrl),
41     .W_ctrl(W_ctrl),
42     .Reset(Reset),
43     .clk(clk),
44     .ALU_carry(ALU_carry),
45     .ALU_result(ALU_result),
46     .Dividend_in(Dividend),
47     .Remainder({Remainder , Quotient})
48 );
49 endmodule
```

將每個模組連接在一起，Quotient 連接到輸出的下半部，Remainder 連接到輸出的上半部。

Divisor:

```
 1 module Divisor_Register (
 2   input Reset,
 3   input W_ctrl,
 4   input [31:0] Divisor_in,
 5   output reg [31:0] Divisor_out
 6 );
 7   always @(*) begin
 8     if (Reset) begin
 9       |   Divisor_out <= 32'd0; //reset
10     end
11     else if (W_ctrl) begin
12       |   Divisor_out <= Divisor_in; //load
13     end
14     else begin
15       |   Divisor_out <= Divisor_out; // not change
16     end
17   end
18
19 endmodule
```

當 W_ctrl 為 1 時，將 in 的數據加載到 out，當 Reset 為 1 時，out 會被重置；在其他情況下，out 的值保持不變。

ALU:

```
 1 module ALU (
 2   input [31:0] Src_1, //remainder out from Remainder_Register
 3   input [31:0] Src_2, //divisor
 4   input subu_ctrl, //this exp use 1bit to reduce area
 5   output reg ALU_carry,
 6   output reg [31:0] ALU_result
 7 );
 8   always @(*) begin
 9     if (subu_ctrl && (Src_1 >= Src_2))begin
10       |   ALU_result <= Src_1 - Src_2;//if is positive, sub
11       |   ALU_carry <= 1; //shift in 1
12     end
13     else begin
14       |   ALU_result <= Src_1;
15       |   ALU_carry <= 0;
16     end
17   end
18 endmodule
```

如果 Src_1 - Src_2 為正數，也就是說，當 Src_1 大於或等於 Src_2，則將 Src_1 - Src_2 的結果送至輸出；反之，如果 Src_2 大於 Src_1 時，則將 Src_1 的值送至輸出。在其餘情況下，輸出保持不變。

Control:

```
1  module Control (
2      input Run, Reset, clk,
3      output reg W_ctrl, SLL_ctrl, SRL_ctrl, Ready,
4      output reg subu_ctrl //this exp use 1bit to reduce area
5  );
6      reg [5:0] counter;
7
8      always @(posedge clk or posedge Reset) begin
9          if (Reset) begin
10              counter <= 0;
11              W_ctrl <= 1;
12              SLL_ctrl <= 1;
13              SRL_ctrl <= 0;
14              Ready <= 0;
15              subu_ctrl <= 0;
16      end
17      else if (Run) begin
18          W_ctrl <= 0;
19          if (counter <= 6'd32) begin
20              subu_ctrl <= 1;
21              SLL_ctrl <= 1;
22              counter <= counter + 1;
23          end
24          else if (counter == 6'd33) begin
25              subu_ctrl <= 0;
26              SLL_ctrl <= 0;
27              SRL_ctrl <= 1;
28              counter <= counter + 1;
29          end
30          else begin
31              Ready <= 1;
32              SRL_ctrl <= 0;
33          end
34      end
35  end
36 endmodule
37
```

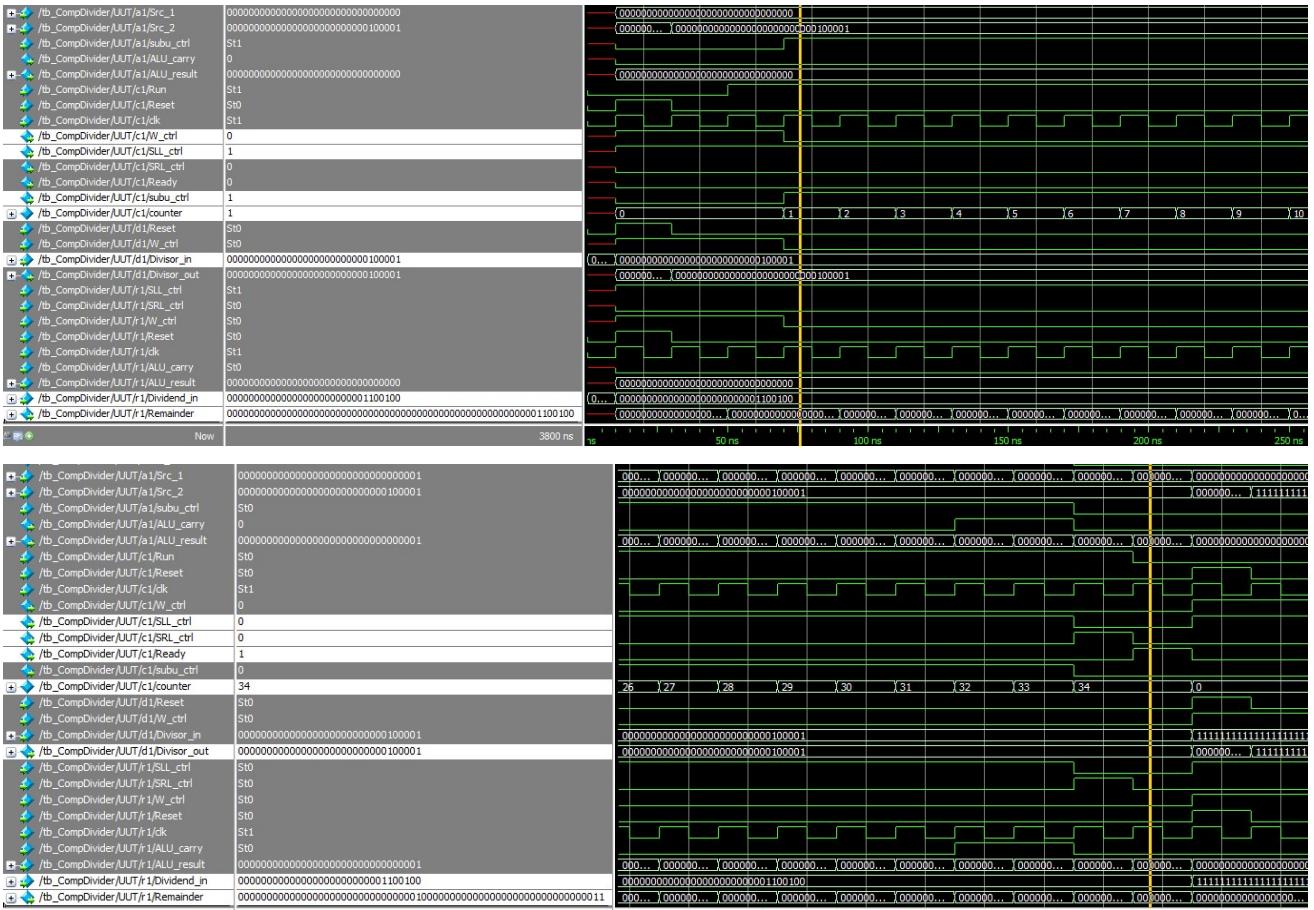
當 Reset 信號啟動時，會將 W_ctrl 和 SLL_ctrl 設為 1，同時將除數和被除數載入到暫存器中。這樣的設置是為了準備開始進行運算。SLL_ctrl 的設置為 1，因為在演算法的開始階段，需要對數據進行一次左移操作（左移是進行除法運算時的必要步驟），之後會執行 32 次的左移操作，進行逐步的運算過程。最後，當所有左移操作完成後，再進行一次右移操作，以調整最終的結果，確保除法演算法的正確性。這樣的步驟能夠確保演算法在每個階段都能夠正確地運行並得到正確的結果。

Reminder:

```
1 module Remainder_Register (
2     input SLL_ctrl, SRL_ctrl, W_ctrl, Reset, clk, ALU_carry,
3     input [31:0] ALU_result, Dividend_in,
4     output reg [63:0] Remainder
5 );
6     reg shift_left_of;
7
8     always @ (posedge clk or posedge Reset) begin
9         if (Reset) begin
10             Remainder <= 0;
11         end
12         else if (W_ctrl) begin
13             Remainder[63:32] <= ALU_result;
14             Remainder[31:0] <= Dividend_in;
15         end
16         else if (SLL_ctrl) begin
17             {shift_left_of, Remainder[63:32]} <= {ALU_result, Remainder[31]};
18             Remainder[31:0] <= {Remainder[30:0], ALU_carry};
19         end
20         else if (SRL_ctrl) begin
21             Remainder[63:32] <= {shift_left_of, Remainder[63:33]};
22         end
23         else begin
24             Remainder <= Remainder;
25         end
26     end
27 endmodule
```

W_ctrl 為 1 時，將資料載入到 Remainder 寄存器；當 SLL_ctrl 為 1 時，會執行左移操作並且考慮 carry；而當 SRL_ctrl 為 1 時，則會進行單純的右移操作，不涉及進位或其他額外處理。shift_left_of 是用來存如果餘數最高為元是一時左移後右移會被設為 0 所以需要一個暫存器來存於數的最高位。

二、 testbench



當 counter 為 0 時，除數和被除數會載入到各自的暫存器中，為接下來的運算做準備。當 counter 為 1 時，所有控制信號會設為 1，並且進行一次左移操作，這是為了初始化計算過程。接著，進行 32 次的左移操作，每次左移後還會進行減法判斷，這是為了進行逐步的除法運算。當 counter 達到 34 時，會執行右移操作，這是最後一步，並且此時會將 Ready 信號設為高電位，表示運算已經完成，準備好將結果輸出。

三、測資

```
# Dividend: 100, Divisor: 33
# Quotient: 3, Remainder: 1
# Dividend:4294967295, Divisor:4294967295
# Quotient: 1, Remainder: 0
# Dividend: 112111379, Divisor: 38228
# Quotient: 293, Remainder: 10575
# Dividend: 149, Divisor: 149
# Quotient: 1, Remainder: 0
# Dividend: 2165, Divisor: 115
# Quotient: 18, Remainder: 95
# Dividend: 5, Divisor: 20
# Quotient: 0, Remainder: 5
# Dividend: 65534, Divisor:4294967295
# Quotient: 0, Remainder: 65534
# Dividend:4294967294, Divisor:4294967295
# Quotient: 0, Remainder:4294967294
```

	testbench > tb_CompDivider.in	testbench > tb_CompDivider.out
1	00000064_00000021	00000003_00000001
2	ffffffff_ffffffff	00000001_00000000
3	00ab1273_00009554	00000125_0000294f
4	00000095_00000095	00000001_00000000
5	00000075_00000073	00000012_0000005f
6	00000005_00000014	00000000_00000005
7	0000ffffe_ffffffff	00000000_0000ffffe
8	ffffffffffe_ffffffff	00000000_fffffffe

	被除數	除數	商	餘數
1.	32 ' h00000064	32 ' h00000021	32 ' h00000003	32 ' h00000001
2.	32 ' hfffffff	32 ' hfffffff	32 ' h00000001	32 ' h00000000
3.	32 ' h00ab1273	32 ' h00009554	32 ' h00000125	32 ' h0000294f
4.	32 ' h00000095	32 ' h00000095	32 ' h00000001	32 ' h00000000
5.	32 ' h000000875	32 ' h00000073	32 ' h00000012	32 ' h0000005f
6.	32 ' h000000005	32 ' h00000014	32 ' h00000000	32 ' h00000005
7.	32 ' h0000ffffe	32 ' hfffffff	32 ' h00000000	32 ' h0000ffffe
8.	32 ' hfffffffe	32 ' hfffffff	32 ' h00000000	32 ' hfffffffe

第一個測資是預設的，我沒有做任何更動。第二個和第四個測資是用來測試整除的情況，這些案例確保在整除時，系統能夠正確地處理除法運算並給出準確的結果。第三個和第五個測資則是隨機數測試，這些測資有助於檢驗系統對不同數字組合的運算處理能力，並確保運算結果在隨機情況下仍然正確無誤，第六個和第七個測資則是用來測試如果商為 0 的情況，這些測資有助於檢測被除數小於除數的情況運算是否正確，第八個測資則是用來測試餘數最高位元為 1 時的狀況，因為如果不用暫存器存取餘數的最高為元，在最後一次左移之後右移會將餘數的最高為元變為 0，解決方法是將左移出去的那個位元存起來，右移時放回去，這樣就不會被吃掉餘數的最高位元，我用了一般計算機來確認這些結果是否正確，這些測資的輸出結果都與我預想的相符。

參、心得

這次的作業難的點是在時序的同步上，不過因為我最近為了參加 IC Contest 的關係，練習了一些相關的題目，比起那些題目，這個作業真的還算簡單。我在合成過程中遇到的問題還比較多，尤其是我電腦無法安裝 WLS，也無法升級系統，因此我選擇使用虛擬機進行合成，但這也帶來了一些額外的挑戰。之後我還遇到了一些安裝程式的问题，例如某些工具無法正常運行，需要額外查找解決方案。雖然遇到不少技術性問題，但最終還是順利完成了這次作業，這些經歷也讓我對未來的 IC 設計和合成更加熟悉。