Computer Organization

PA1

(Unsigned Multiplier and Unsigned Divider)

Student

B11107157 電機三乙 林明宏

RTL Simulator & Synthesis Tool

ModelSim-Intel FPGA Standard Edition, Version 20.1.1, windows

The OpenROAD-flow-scripts from github

Slack & Area Report

Unsigned Multiplier | Critical Path Slack : 4.5237 | Area : 1223.334 um^2

Unsigned Divider | Critical Path Slack : 4.4499 | Area : 1223.068 um^2

1. Descriptions of how you implement each module2

Part 1 . Unsigned Multiplier

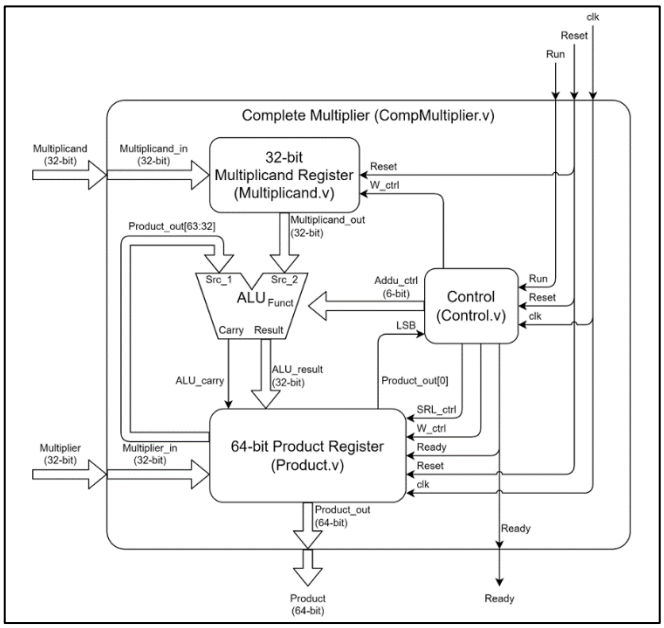
a. CompMultiplier.v

這個 Module 基本上就是把 Multiplicand、ALU、Product、Control 這些模塊做一些整合，把各自的線路照著架構圖上接起來，程式碼以及架構圖如下所示。

```
module CompMultiplier(
    // Outputs
    output  [63:0]  Product,
    output      Ready,
    // Inputs
    input   [31:0]  Multiplicand,
    input   [31:0]  Multiplier,
    input       Run,
    input       Reset,
    input       clk
);
    wire W_ctrl,Carry;
    wire [31:0] Multiplicand_out,Result;
    Multiplicand m0(Multiplicand,Multiplicand_out);
    ALU u0(Product[63:32],Multiplicand_out,Product[0],Carry,Result);
    Product p0(Carry,Result,Multiplier,W_ctrl,clk,Product);
    Control l0(Run,Reset,clk,W_ctrl,Ready);

endmodule
```

圖一、CompMultiplier.v



圖二、Unsigned Multiplier 架構圖

b. Multiplicand.v

這個 Module 主要的功能是用來將 Multiplicand_in 送到 Multiplicand_out，在這邊我把本來應該存在的 Reset 與 W_ctrl 皆移除，可以有效降低電路面積，並且也可實現電路功能。
然而，觀察程式後可以發現，目前的這個 Module 並沒有存在的必要性，我們可以直接將 Multiplicand_in 直接送到 ALU 即可，雖然 Slack 會稍微下降，但 Area 下降的相對幅度更大。

```
/*
module Multiplicand(
    input W_ctrl,
    input [31:0] Multiplicand_in,
    output [31:0] Multiplicand_out
);
    // Assignment ----------------------------------------------
    assign Multiplicand_out = Multiplicand_in;
endmodule
*/
```

圖三、Multiplicand.v(以詢問助教後並允許)

c. ALU.v

這個 Module 負責在 Control.v 判斷目前需要 ADD 時進行，把 Product [63:32] + Multplicand 不需要時則維持輸出 Product [63:32]。
注意在這邊需要引入 Carry 接腳，因為後面 Product.v 做向右位移時，需要將 Carry 位移進來。另外可以注意的是，在這個設計下我將 Funct 定義為 1 個 bit，是因為該題目只需要做加法而已，因此將其設為 1 個 bit 即可完成功能。

```
1   module ALU(
2       input [31:0] Src_1, //Product
3       input [31:0] Src_2, //Multiplicand_out
4       input Funct,
5       output Carry,
6       output [31:0] Result
7   );
8       // Assignment ------------------------------------------
9       assign {Carry,Result} = (Funct)? Src_1 + Src_2 : Src_1;
10
11  endmodule
12
```

圖四、ALU.v

d. Product.v

這個 Module 負責處理 Product 目前是否需要右移，或是維持不變，以及將做完加法的結果放入到 Product 當中，可以注意的是這個模塊我一樣沒有引入 Reset，和 a.部分相同，因為 W_ctrl 是由 Reset 驅動，因此就不在重複引入，else 中即包含了 Reset 時的情況。

在這個模塊中，我也將架構圖中的 Ready 也去除，因為 Ready 是由 Control.v 上的 cnt 驅動，當計數器數了 32 次時自然會將 Product_out 送出，根據該題目設計電路行為，Ready 並不需要引入 Product.v 中即可完成。

而 SRL_ctrl 也被我移除這個 Module，因為當 W_ctrl 拉起時，自然也必須要做向右位移的動作，因此不需要再額外判斷，可以減少面積。

```
1   module Product(
2       input ALU_carry,
3       input [31:0] ALU_result,
4       input [31:0] Multiplier_in,
5       input W_ctrl,
6       input clk,
7       output reg [63:0] Product_out
8   );
9
10      // Set Product_out ------------------------------------------------------------
11      always @(posedge clk) begin
12          if(W_ctrl) Product_out <= {ALU_carry,ALU_result,Product_out[31:1]};
13          else Product_out <= {32'd0,Multiplier_in}; // Include Reset and else condition
14      end
15
16  endmodule
```

圖五、Product.v

e. Control.v

這個 Module 定義了許多控制線，例如 Addu_ctrl、SRL_ctrl、W_ctrl……等，通過這些控制線去控制每個模塊在每個時間點上做對應的事情，在這個模塊裡，所有控制線皆由 Reset 信號歸 0，當 Reset 是 High Level 時，即把控制線歸 0，包含 cnt，而 Ready 則是由 cnt 驅動的，因此也會被 Reset 到 0，並且當 cnt 數 32 次時即把 Ready 信號拉起。
而在這個 Module 裡面，我將 Addu_ctrl、SRL_ctrl 等兩條控制線皆移除，因為該作用由 W_ctrl 就可以達成，不需要額外的控制線，以減少電路面積。

```
module Control(
    input Run,
    input Reset,
    input clk,
    output W_ctrl,
    output Ready
);
    // Set Register ------------------------------------
    reg [4:0] cnt;

    // Assignment --------------------------------------
    assign Ready = (cnt==5'd31)? 1'b1 : 1'b0;
    assign W_ctrl = (Reset)? 1'b0 : 1'b1;

    // Set Counter -------------------------------------
    always @(posedge clk) begin
        cnt <= (Run)? cnt + 5'd1 : 5'd0;
    end

endmodule
```

圖六、Control.v

Part2. Unsigned Divider

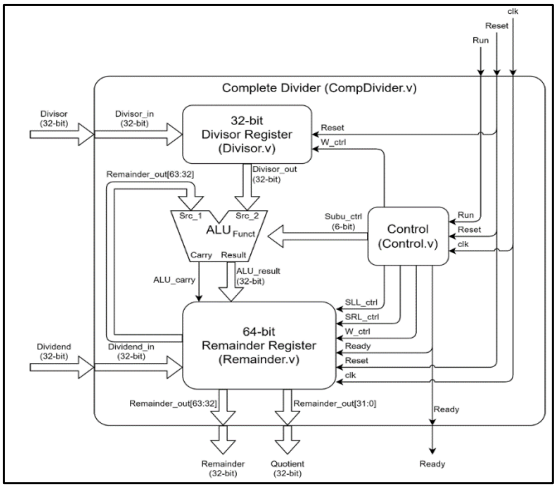
a. CompDivider.v

這個 Module 基本上就是把 Divisor、ALU、Remainder、Control 這些模塊做一些整合，把各自的線路照著架構圖上接起來，程式碼以及架構圖如下所示。
和乘法器稍微不一樣的地方是需要把 Remainder_out 分成 Quotient 和 Remainder 兩個部分。並且需要把 Remainder_out 左半部分做右移才能送給 Remainder。

```
module CompDivider(
    // Outputs
    output  [31:0]  Quotient,
    output  [31:0]  Remainder,
    output          Ready,
    // Inputs
    input   [31:0]  Dividend,
    input   [31:0]  Divisor,
    input           Run,
    input           Reset,
    input           clk
);
    wire W_ctrl;
    wire [31:0] Divisor_out,Result;
    wire [63:0] Remainder_out;
    Divisor d0(Divisor,Divisor_out);
    ALU u0(Remainder_out[63:32],Divisor_out,W_ctrl,Result);
    Remainder r0(Result,Dividend,W_ctrl,clk,Remainder_out);
    Control l0(Run,Reset,clk,W_ctrl,Ready);
    // Assignment --------------------------------------------------------------
    assign {Remainder,Quotient} = {1'b0,Remainder_out[63:33],Remainder_out[31:0]};

endmodule
```

圖七、CompDivider.v



圖八、Unsigned Divider 架構圖

b. Divisor.v

這個 Module 主要的功能是用來將 Divisor_in 送到 Divisor_out，這裡和乘法器一樣，為了減少電路面積這個 Module 沒有引入其他額外的控制線，純粹做一個 Buffer 使用。

然而，觀察程式後可以發現，目前的這個 Module 並沒有存在的必要性，我們可以直接將 Divisor_in 直接送到 ALU 即可，雖然 Slack 會稍微下降，但 Area 下降的相對幅度更大。

```
/*
module Divisor(
    input [31:0] Divisor_in,
    output [31:0] Divisor_out
);
    // Assignment ------------------------------------
    assign Divisor_out = Divisor_in;

endmodule
*/
```

圖九、Divisor.v(以詢問助教後並允許)

c. ALU.v

這個 Module 用來判斷目前是否需要將 Remainder[63:32] 減掉 Divisor，如果不用就輸出原來的 Remainder[63:32]，在這邊一樣將 Funct 設定為 1 個 bit 是因為該設計只需要做減法而已。

```
module ALU(
    input [31:0] Src_1, //Remainder
    input [31:0] Src_2, //Divisor_out
    input Funct,
    output Carry,
    output [31:0] Result
);
    // Assignment ----------------------------------------
    assign {Carry,Result} = (Funct)? Src_1 - Src_2 : Src_1;

endmodule
```

圖十、ALU.v

d. Remainder.v

這個 Module 相對其他模塊來說比較複雜，設計的步驟基本上就是按照設計流程圖對 Remainder_out 做指定的操作(詳細可以看下面的流程圖)，最後的右移步驟則是在 CompDivider 中實現。

在這個設計中我一樣沒有引入 Reset，因為該模塊的控制線皆在 Control.v 中被 Reset 所驅動，因此沒有引入 Reset 接腳。

而 Ready 也沒有引入至該模塊，在這個設計中，Ready 會在完成右移後被拉起，此時就可以輸出正確的 Remainder 和 Quotient，Ready 不需要引入該模塊即可完成正確的電路行為。

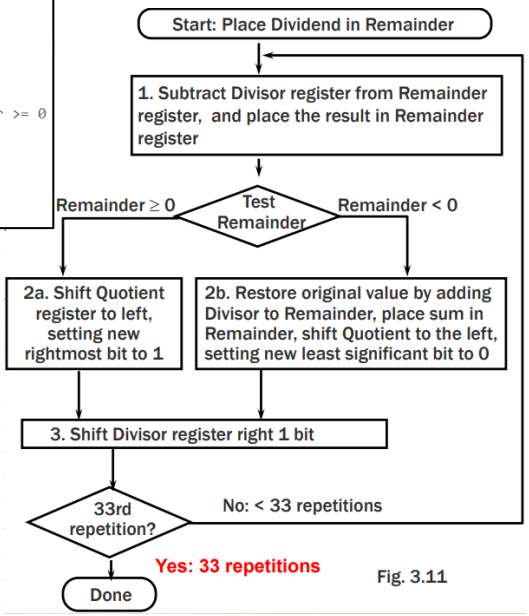
並且和前面乘法器一樣，因為 W_ctrl 只要拉起就會做向左位移的動作，因此 SLL_ctrl 即使沒有引入該 Module 中，也可以完成正確的電路行為。

以下是程式碼和詳細的流程步驟 ：

```
module Remainder(
    input [31:0] ALU_result,
    input [31:0] Dividend_in,
    input W_ctrl,
    input clk,
    input Carry,
    output reg [63:0] Remainder_out
);

    always @(posedge clk) begin
        if(W_ctrl) begin
            if(Carry) Remainder_out <= {Remainder_out[62:0],1'b0}; // New Remainder < 0
            else Remainder_out <= {ALU_result[30:0],Remainder_out[31:0],1'b1}; // New Remainder >= 0
        end
        else Remainder_out <= {31'd0,Dividend_in,1'd0};
    end

endmodule
```



圖十一、Remainder & 除法流程圖

e. Control.v

這個 Module 定義了許多控制線，例如 Subu_ctrl、SRL_ctrl、SLL_ctrl、W_ctrl……等，通過這些控制線去控制每個模塊在每個時間點上做對應的事情，在這個模塊裡，所有控制線皆由 Reset 信號歸 0，當 Reset 是 High Level 時，即把控制線歸 0，包含 cnt。
Ready 則是由 cnt 驅動的，因此也會被 Reset 到 0，而數到第 32 次時則把 Ready 信號拉起。

然而我們可以發現 Subu_ctrl 和 SLL_ctrl 皆可以被 W_ctrl 所取代，而 SRL_ctrl 由於在 CompDivider 中有定義向右位移行為，因此該控制線也可以移除。

```
module Control(
    input Run,
    input Reset,
    input clk,
    output W_ctrl,
    output Ready
);
    // Set Register ---------------------------------------
    reg [4:0] cnt;

    // Assignment -----------------------------------------
    assign Ready = (cnt==5'd31)? 1'b1 : 1'b0;
    assign W_ctrl = (Reset)? 1'b0 : 1'b1;

    // Set Counter ----------------------------------------
    always @(posedge clk) begin
        cnt <= (Run)? cnt + 5'd1 : 5'd0;
    end

endmodule
```

圖十二、Control.v

2. Descriptions of how you test your modules

Part1. Unsigned Multiplier

我設計了以下 10 個 Test Pattern，分別測試不同結果來驗證功能是否符合預期。
以下表格皆用 10 進制作為表示 ：

Test Pattern	Multiplicand	Multiplier	Result	測試目的
1	1003	20	20060	乘法功能是否正確
2	51	17	867	乘法功能是否正確
3	170	593	100810	乘法功能是否正確
4	16780544	2	33561088	乘法功能是否正確
5	43520	5	217600	乘法功能是否正確
6	268435456	1	268435456	乘法功能是否正確(當乘數為 1)
7	1	10	10	乘法功能是否正確(當被乘數為 1)
8	0	0	0	乘法功能是否正確(當兩數皆為 0)
9	1145324612	0	0	乘法功能是否正確(當被乘數為 0)
10	0	17476	0	乘法功能是否正確(當被乘數為 0)

表一、Test Patterns of Unsigned Multiplier

Part2. Unsigned Divider

我設計了以下 10 個 Test Pattern，分別測試不同結果來驗證功能是否符合預期。
此外該設計不考慮除 0，因此 Test Pattern 沒有設計除 0 情況
以下表格皆用 10 進制作為表示 ：

Test Pattern	Dividend	Divisor	Quotient	Reminder	測試目的
1	10	2	5	0	除法功能是否正確
2	43690	10	4369	0	除法功能是否正確
3	57344	8	7168	0	除法功能是否正確
4	16869	17	992	5	除法功能是否正確
5	21521	46	467	39	除法功能是否正確
6	349525	3822	91	1723	除法功能是否正確
7	1048576	2	524288	0	除法功能是否正確
8	100	33	3	1	除法功能是否正確
9	33	100	0	33	除法功能是否正確(當商為 0)
10	120	1	120	0	除法功能是否正確(當除數為 1)

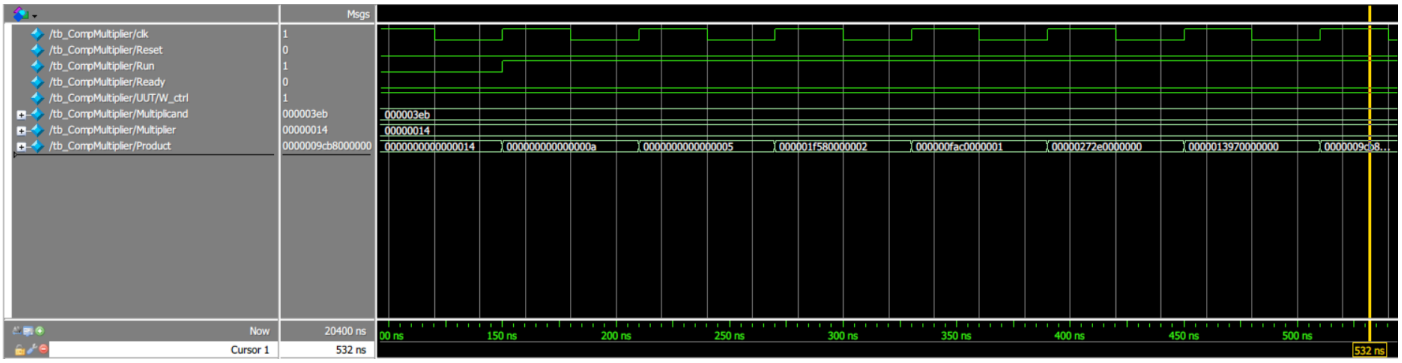
表二、Test Patterns of Unsigned Divider

上述兩個 Test Pattern 將會以 16 進制分別寫入 tb_CompMultiplier.in 和 tb_CompDivider.in 中。
該報告就不再截圖該.in 檔上來做展示。

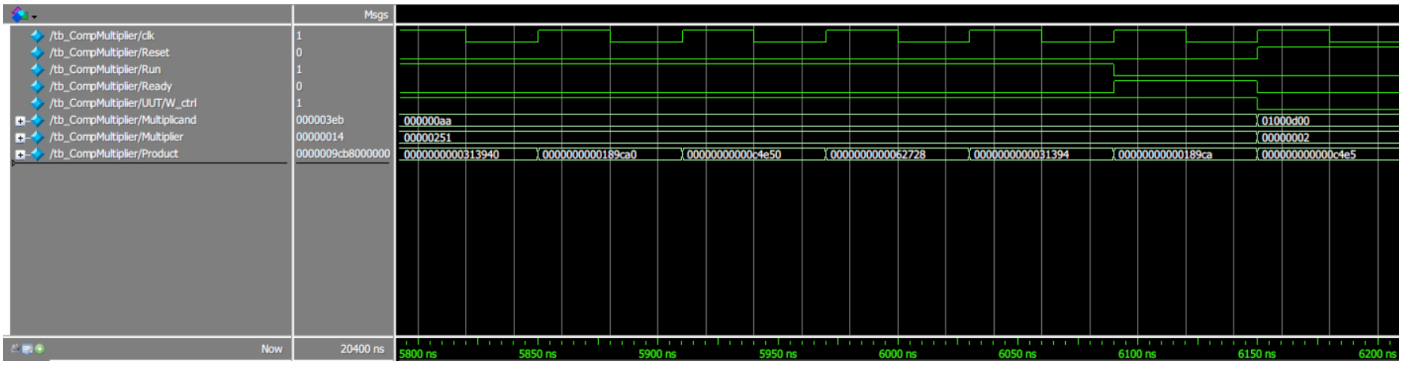
3. Descriptions of the test results (Hexadecimal Waveforms) for your modules

由於本次設計波型圖較長，篇幅有限，不太可能全部放上來，因此以下波型與描述只針對 Test Pattern 1 做討論，其他 Test Pattern 結果皆符合 2.列出的 Test Pattern 表格。

Part1. Unsigned Multiplier



Reset 歸 0，乘法器開始動作，根據流程圖執行乘法動作，直到數至 32 次後將 Ready 拉起為 1，TestBench 觀測輸出結果。



如上波型圖所示，最後 Product_out 輸出 00000000_000004e5c

代表 Product = $4e5c_{(hex)} = 20060_{(dec)}$，其結果符合預期(預期結果如下所示)。

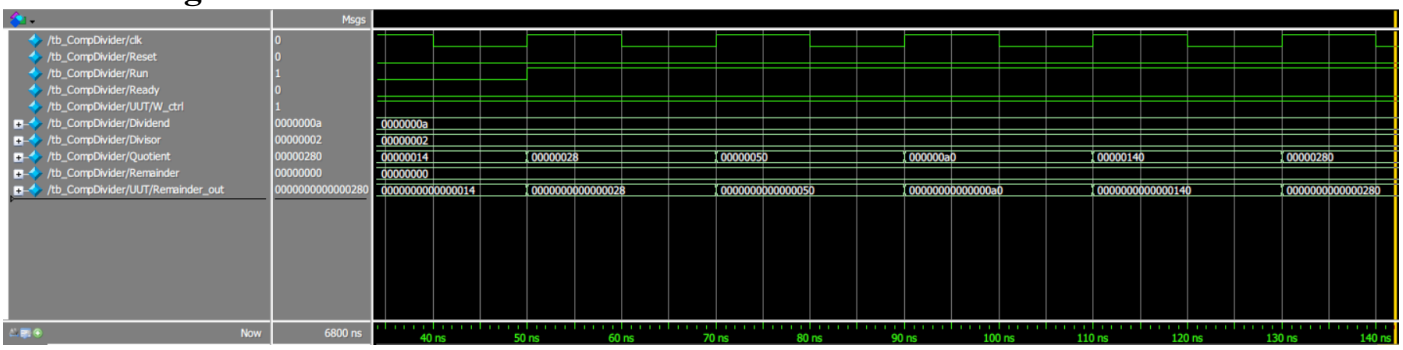
Test Pattern	Multiplicand	Multiplier	Result	測試目的
1	1003	20	20060	乘法功能是否正確

由下圖可知，其他 Pattern 皆符合 2.的預期結果。

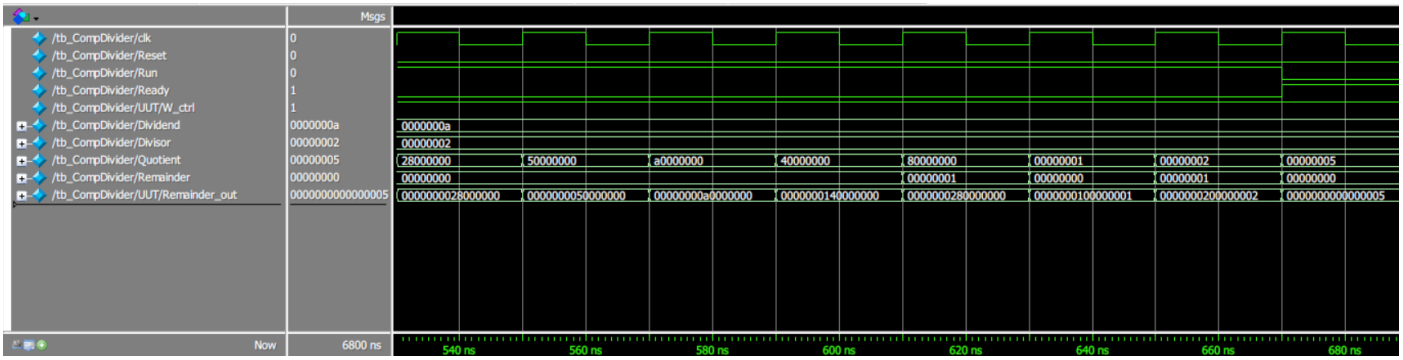
```
VSIM 35> run -all
# Multiplicand:      1003, Multiplier:        20
# result:             20060
# Multiplicand:        51, Multiplier:        17
# result:               867
# Multiplicand:       170, Multiplier:       593
# result:             100810
# Multiplicand:  16780544, Multiplier:         2
# result:          33561088
# Multiplicand:     43520, Multiplier:         5
# result:             217600
# Multiplicand: 268435456, Multiplier:         1
# result:          268435456
# Multiplicand:         1, Multiplier:        10
# result:                10
# Multiplicand:         0, Multiplier:         0
# result:                0
# Multiplicand:1145324612, Multiplier:         0
# result:                0
# Multiplicand:         0, Multiplier:     17476
# result:                0
# ** Note: $stop    : D:/Project/Computer_Organization/PA1/Part1/tb_CompMultiplier.v(118)
#    Time: 20400 ns  Iteration: 0  Instance: /tb_CompMultiplier
```

圖十三、輸出結果

Part2. Unsigned Divider



Reset 歸 0 後除法器開始動作，根據流程圖執行除法動作，直到數至 32 次後將 Ready 拉起為 1，TestBench 觀測輸出結果。



如上波型圖所示，最後 Remainder_out 輸出 00000000_000000005
代表 Quotient 為 5，而 Remainder 為 0，其結果符合預期(預期結果如下所示)。

Test Pattern	Dividend	Divisor	Quotient	Reminder	測試目的
1	10	2	5	0	除法功能是否正確

由下圖可知，其他 Test Patterns 如 2.表格所示，皆和預期結果符合。

```
VSIM 32> run -all
# Dividend:        10, Divisor:         2
# Quotient:         5, Remainder:         0
# Dividend:     43690, Divisor:        10
# Quotient:      4369, Remainder:         0
# Dividend:     57344, Divisor:         8
# Quotient:      7168, Remainder:         0
# Dividend:     16869, Divisor:        17
# Quotient:       992, Remainder:         5
# Dividend:     21521, Divisor:        46
# Quotient:       467, Remainder:        39
# Dividend:    349525, Divisor:      3822
# Quotient:        91, Remainder:      1723
# Dividend:   1048576, Divisor:         2
# Quotient:    524288, Remainder:         0
# Dividend:       100, Divisor:        33
# Quotient:         3, Remainder:         1
# Dividend:        33, Divisor:       100
# Quotient:         0, Remainder:        33
# Dividend:       120, Divisor:         1
# Quotient:       120, Remainder:         0
# ** Note: $stop    : D:/Project/Computer_Organization/PA1/Part2/tb_CompDivider.v(120)
#    Time: 7 us  Iteration: 0  Instance: /tb_CompDivider
```

圖十四、輸出結果

4. Conclusion and insights

我覺得這次的 Project 還滿特別的，由於最近在準備 IC Contest 競賽的緣由，因此練習了大量的考古題，每一年的都是新的題目、新的挑戰，然而這次的 Project 卻不像我以為在練習比賽那樣，單一只用一個模塊，建立好相應的 FSM 符合題目的需求就行了。

在這次的 Project 上，我們需要分別使用 5 個模塊，來完成整體的功能，這讓我一開始非常不適應，不知從何下手，但一樣通過之前練習比賽的經驗，我知道我需要先做的事情，就是分析好每個模塊之間的溝通該如何處理，處理好每個信號應該在甚麼時間點 High 以及 Low，分析好後，我便很快速地照著老師課堂上所教的乘法器與除法器的演算法下去做實作並完成模擬，比較特別的是這次的 Project 有使用 OpenROAD 來進行合成，這是我第一次使用這個 Synthesis Tool，以往在實驗室以及練習比賽的時候都是使用 Design Complier，不過就第一次使用的經驗來講，還不錯。

這次的 Project 除了處理每個模塊之間的溝通以外，還有另一個重點就是優化時間和面積，這個對於我來說其實算是滿常在處理的事情，畢竟在練習 IC Contest 的考古題時，做出功能並不代表完成，重要的時間跟面積都必須要在指定的規格裡面，才有辦法達標或是取得勝利。

因此，在這個 Project 中我有用到幾個優化小技巧，首先是 Slack，我認為只需要分配好每個 clock 中該做的事情就好，並且每個 clock 裡面通常我會盡量只處理一件事情，這次的 Project 我就是如此設計，如果實在沒有辦法的話就做 Pipeline 來分散單周期工作量的壓力，再來是面積的部分，在這次 Project 中，我對於跟 clock 無關的信號皆用 Assignment 來描寫，並且會合成出相對應複雜電路的語法也一概不寫入(例如: 寫”＞”跟”＜”會合成出比較器…)，以此來減少面積。

總體而言，雖然本次 Project 對我來說難度不高，但剛開始還是有點不熟悉，不過從本次實作過後我更加了解了要怎麼處理每個模塊之間的溝通，畢竟在下一次的 Project 2、Project 3 甚至是未來到了外面工作，每個模塊裡面肯定會越來越複雜，要怎麼正確的連結每個模塊，是我覺得我在這堂課上面最重要的學習。