

電路設計說明

第14組

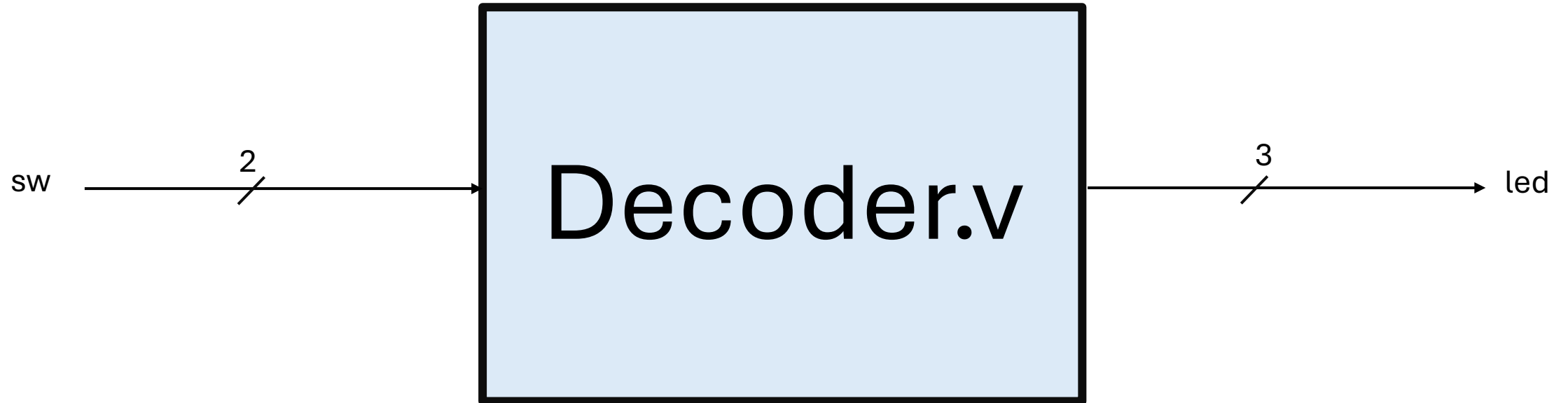
黃偉峰 E34106010

陳識博 E94106096

黃芊 F74104040

Problem 1 - RGB LED

Block diagram




Decoder implementation

- 右方這段RTL code將會產生出2 to 4 decoder with 3-bit output，透過case語句根據sw選擇出rgb的輸出值。

```
module Decoder(sw, rgb);  
    input [1:0] sw;  
    output reg [2:0] rgb;  
  
    always @(*)begin  
        case(sw)  
            2'b00:    rgb = 3'b111;  
            2'b01:    rgb = 3'b100;  
            2'b10:    rgb = 3'b010;  
            2'b11:    rgb = 3'b110;  
            default : rgb = 3'b111;  
        endcase  
    end  
end
```

Constraints

switchs 作為輸入sw 訊號



```
##Switches
```

```
set_property -dict { PACKAGE_PIN M20  IOSTANDARD LVCMOS33 } [get_ports { sw[0] }]; #IO_L7N_T1_AD2N_35 Sch=sw[0]
```

```
set_property -dict { PACKAGE_PIN M19  IOSTANDARD LVCMOS33 } [get_ports { sw[1] }]; #IO_L7P_T1_AD2P_35 Sch=sw[1]
```

```
##RGB LEDs
```


```
set_property -dict { PACKAGE_PIN L15  IOSTANDARD LVCMOS33 } [get_ports { rgb[0] }]; #IO_L22N_T3_AD7N_35 Sch=led4_b
```

```
set_property -dict { PACKAGE_PIN G17  IOSTANDARD LVCMOS33 } [get_ports { rgb[1] }]; #IO_L16P_T2_35 Sch=led4_g
```

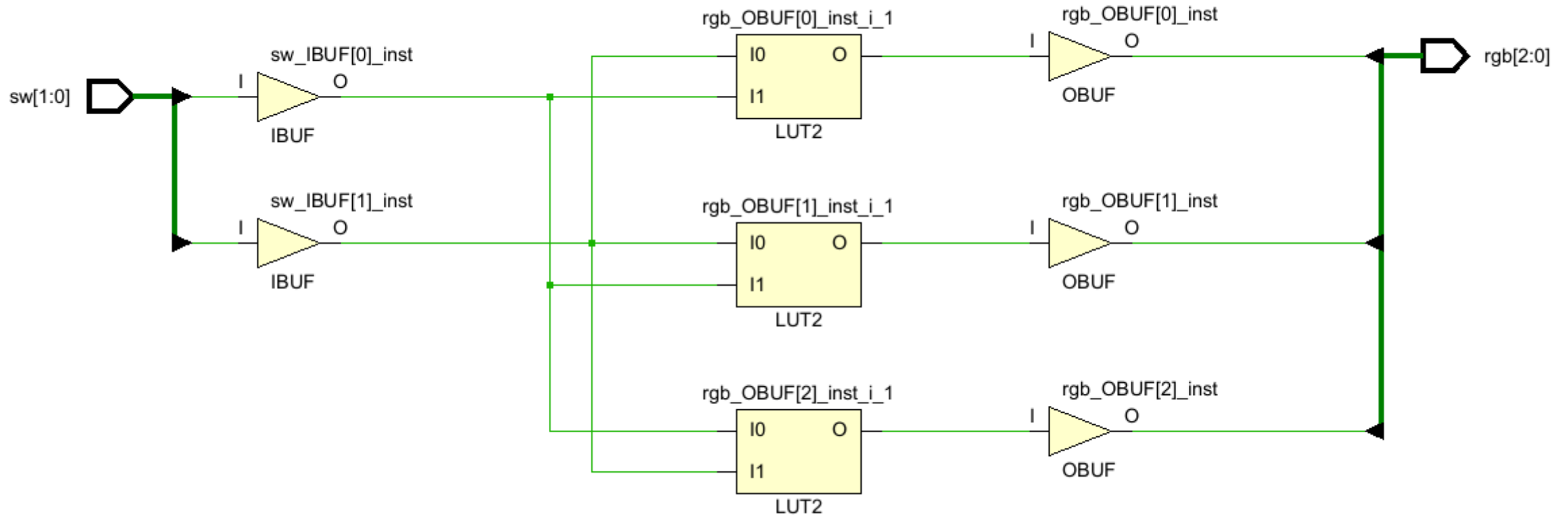
```
set_property -dict { PACKAGE_PIN N15  IOSTANDARD LVCMOS33 } [get_ports { rgb[2] }]; #IO_L21P_T3_DQS_AD14P_35 Sch=led4_r
```

```
#set_property -dict { PACKAGE_PIN G14  IOSTANDARD LVCMOS33 } [get_ports { led5_b }]; #IO_0_35 Sch=led5_b
```

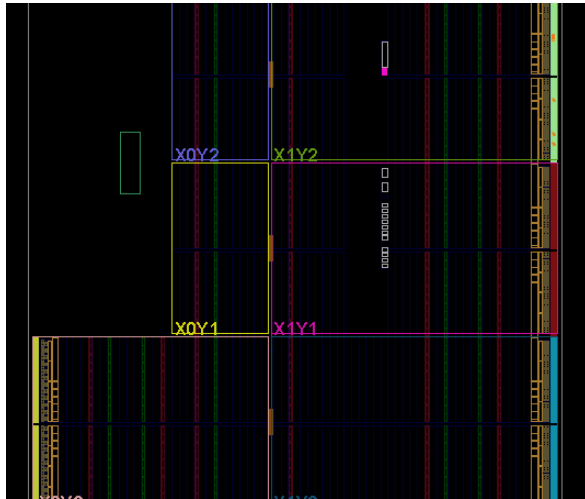
rgb led 輸出訊號



Gate-Level netlist

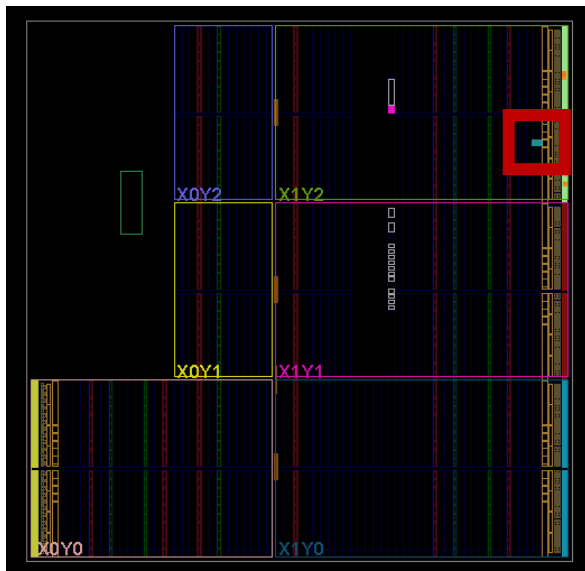


Synthesis and Implementation



Floorplan
After synthesis

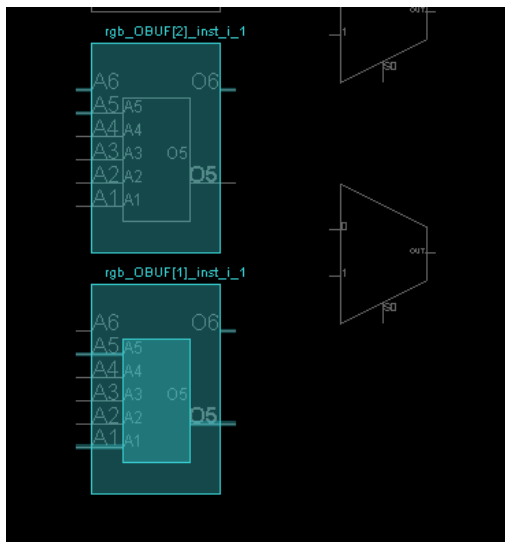
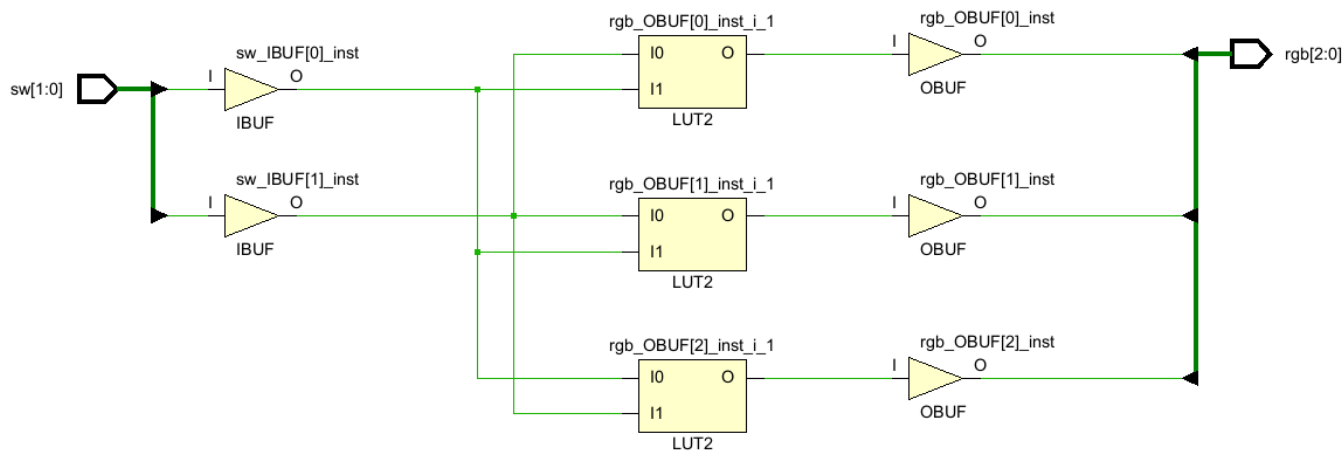
FPGA 設計流程中，Synthesis是將 RTL轉換為 Gate-Level Netlist 的過程。在這個階段，EDA 工具會進行多項優化，包括刪除冗餘邏輯、Boolean Optimization，並將邏輯映射到 LUT、FF、MUX等 FPGA 資源。然而，Synthesis 過程並不涉及Placement與Routing，因此在左圖中，尚未顯示具體的CLB被使用的情況。



Floorplan
After implementation

Implementation過程中執行了Placement和Routing，左圖可以看到 CLB已經被分配。

CLB跟Gate Level netlist關係



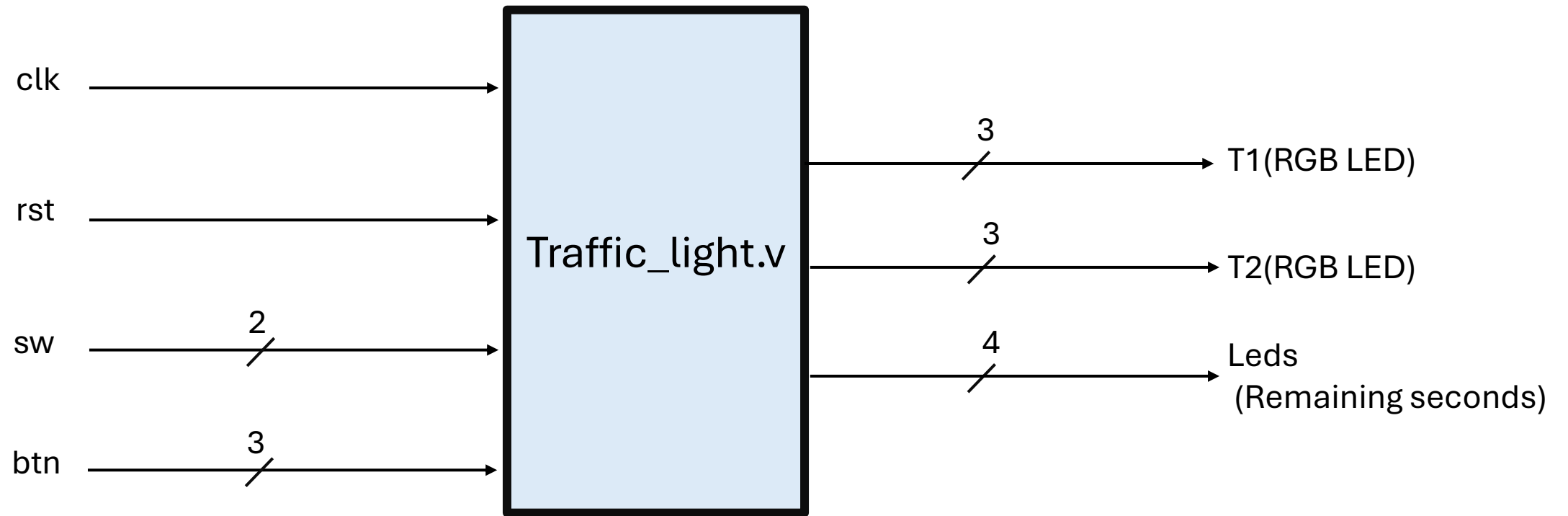
上圖為Gate-Level netlist，左圖為Implementation在FPGA上的結果，Pynq板的每個CLB裡面具有兩個Slice每個Slice又有四個6-input的LUT而每個6-input的LUT又可以當作兩個5-input的LUT使用，從netlist可以看到我們有三個output，因此tool幫我們implement成使用了兩個6-input的LUT，而這兩個6-input的LUT實際上都被用成了5-input的LUT，下方的LUT提供兩個output，上方的LUT提供一個output，剛好可以對應到netlist的三個input。

Video Time code

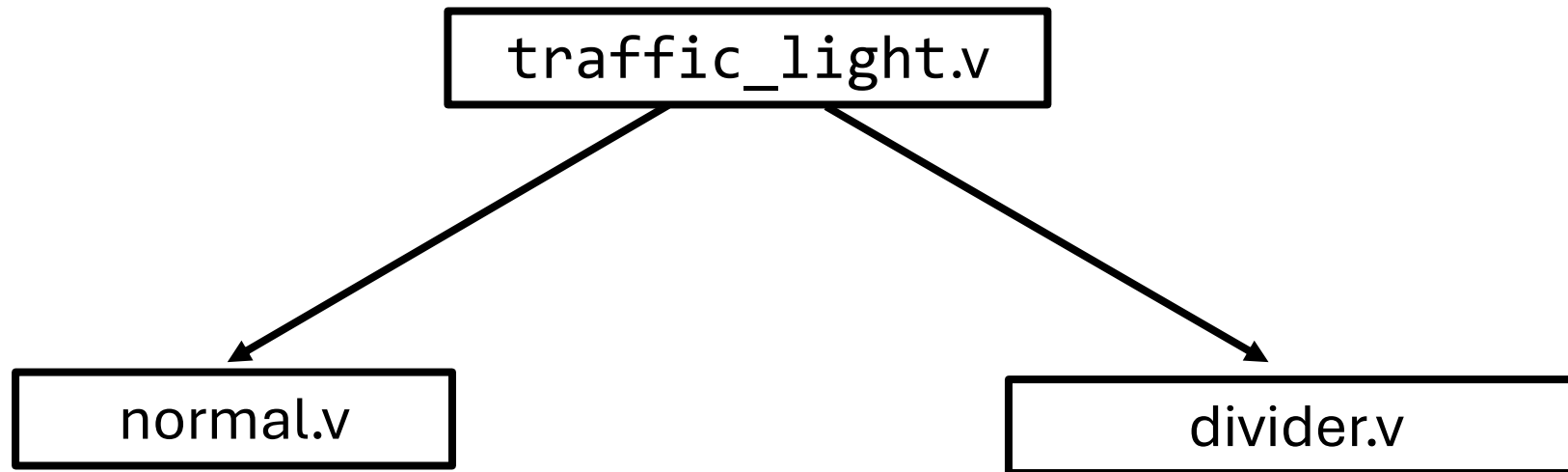
- Demo_Video
- [00:03] Problem1 RGB LED

Problem 2 - Traffic Light

Top Module Block diagram



Module hierarchy

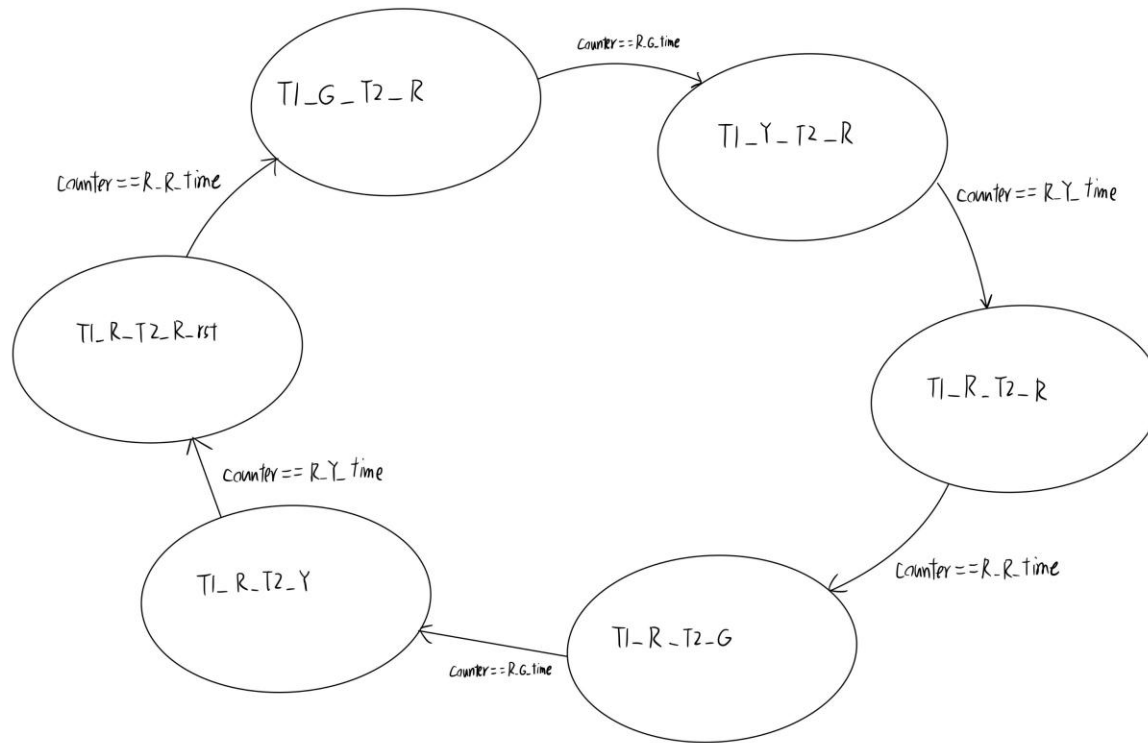


當 `sw = 2'b00` 時的正常紅綠燈模式。

除頻器
將125Mhz的CLK頻率
降到1Hz。

normal.v - FSM

若 $sw \neq 2'b00$ 時,
則維持在初始 $T1_G_T2_R$ 狀態



FSM

```
//next_state_logic
always @(*)
    if(rst)
        next_state = 0;
    else
        case(curr_state)
            T1_G_T2_R : if(counter==R_G_time)
                          next_state = T1_Y_T2_R;
                        else
                          next_state = T1_G_T2_R;
            T1_Y_T2_R : if(counter==R_Y_time)
                          next_state = T1_R_T2_R;
                        else
                          next_state = T1_Y_T2_R;
            T1_R_T2_R : if(counter==R_R_time)
                          next_state = T1_R_T2_G ;
                        else
                          next_state = T1_R_T2_R;
            T1_R_T2_G : if(counter==R_G_time)
                          next_state = T1_R_T2_Y ;
                        else
                          next_state = T1_R_T2_G ;
            T1_R_T2_Y : if(counter==R_Y_time)
                          next_state = T1_R_T2_R_rst;
                        else
                          next_state = T1_R_T2_Y;
            T1_R_T2_R_rst : if(counter==R_R_time)
                              next_state = T1_G_T2_R;
                            else
                              next_state = T1_R_T2_R_rst;
            default : next_state = T1_G_T2_R;
```

normal.v – I/O

normal模組用來跑當 sw 訊號為 2'b00 時的正常紅綠燈狀態。

T1、T2 輸出到 RGB leds, remain訊號作為leds的倒數計時輸出。

```
module normal(  
    input clk,  
    input rst,  
    input [3:0] R_G_time,  
    input [3:0] R_Y_time,  
    input [3:0] R_R_time,  
    output reg [2:0] T1,  
    output reg [2:0] T2,  
    output reg [3:0] remain);
```

I/O port

```
//output T1_T2  
always @(*)  
    if(rst)  
        T1 = 3'b111;  
    else  
        case(curr_state)  
            T1_G_T2_R : T1 = Green;  
            T1_Y_T2_R : T1 = Yellow;  
            T1_R_T2_R : T1 = Red;  
            T1_R_T2_G : T1 = Red;  
            T1_R_T2_Y : T1 = Red;  
            T1_R_T2_R_rst : T1 = Red;  
            default : T1 = 3'b111;  
        endcase  
  
    if(rst)  
        T2 = 3'b111;  
    else  
        case(curr_state)  
            T1_G_T2_R : T2 = Red;  
            T1_Y_T2_R : T2 = Red;  
            T1_R_T2_R : T2 = Red;  
            T1_R_T2_G : T2 = Green;  
            T1_R_T2_Y : T2 = Yellow;  
            T1_R_T2_R_rst : T2 = Red;  
            default : T2 = 3'b111;  
        endcase
```

```
//remain  
always @(*)  
    if(rst)  
        remain = 4'b1111;  
    else  
        case(curr_state)  
            T1_G_T2_R : remain = R_G_time - counter;  
            T1_Y_T2_R : remain = R_Y_time - counter;  
            T1_R_T2_R : remain = R_R_time - counter;  
            T1_R_T2_G : remain = R_G_time - counter;  
            T1_R_T2_Y : remain = R_Y_time - counter;  
            T1_R_T2_R_rst : remain = R_R_time - counter;  
            default : remain = 4'b1111;  
        endcase
```

traffic_light.v

```
//RGB_time

//Red
always @(posedge clk_div or posedge rst)
    if(rst)
        R_time <= 4'd2;
    else
        case(sw)
            2'b00 : R_time <= R_time;
            2'b01 : R_time <= R_time;
            2'b10 : R_time <= R_time;
            2'b11 : if(btn[0])
                        R_time <= 4'd2;
                    else if(btn[1])
                        R_time <= R_time + 4'd1;
                    else if(btn[2])
                        R_time <= R_time - 4'd1;
                    else
                        R_time <= R_time;
        endcase
```

根據 switch 決定紅綠燈目前的控制狀態，以左圖為例，R_time 代表在兩個RGB皆為紅燈時的維持秒數，按下 btn[1]時加一秒，按下 btn[2]時減一秒。

另有 Y_time 以及 G_time 分別記錄一紅一黃以及一紅一綠的燈號時長。

traffic_light.v – output(leds)

```
//leds
always @(posedge clk_div or posedge rst)
    if(rst)
        leds <= 4'b0000;
    else
        case(sw)
            2'b00 : leds <= remain;
            2'b01 : leds <= G_time;
            2'b10 : leds <= Y_time;
            2'b11 : leds <= R_time;
        endcase
```

leds 輸出使用二進位制來呈現距離RGB 燈號轉換的剩餘秒數。

若 $sw \neq 2'b00$ ，則表示當前燈號的維持的秒數。

traffic_light.v – output(T1 and T2)

```
//T1
always @(posedge clk_div or posedge rst)
  if(rst)
    T1 <= 3'b111;
  else
    case(sw)
      2'b00 : T1 <= normal_T1;
      2'b01 : T1 <= 3'b010;
      2'b10 : T1 <= 3'b110;
      2'b11 : T1 <= 3'b100;
    endcase

always @(posedge clk_div or posedge rst)
  if(rst)
    T2 <= 3'b111;
  else
    case(sw)
      2'b00 : T2 <= normal_T2;
      2'b01 : T2 <= 3'b100;
      2'b10 : T2 <= 3'b110;
      2'b11 : T2 <= 3'b100;
    endcase
```

```
wire [3:0] remain;
normal_switch_00(.clk(clk_div), .rst(rst), .R_G_time(G_time), .R_Y_time(Y_time),
                 .R_R_time(R_time), .T1(normal_T1), .T2(normal_T2), .remain(remain));
```

不同 sw 狀態下 T1 以及 T2 的 RGB 燈號輸出。

SW	RGB LEDs	state
00	正常運作	正常運作
01	T1 綠 T2 紅	調一綠一紅時長
10	T1 黃 T2 黃	調一黃一紅時長
11	T1 紅 T2 紅	調兩紅時長

Constraints

```
set_property -dict { PACKAGE_PIN H16    IOSTANDARD LVCMOS33 } [get_ports { clk }]; #IO_L13P_T2_MRCC_35 Sch=sysclk
#create_clock -add -name sys_clk_pin -period 8.00 -waveform {0 4} [get_ports { sysclk }];

##Switches

set_property -dict { PACKAGE_PIN M20    IOSTANDARD LVCMOS33 } [get_ports { sw[0] }]; #IO_L7N_T1_AD2N_35 Sch=sw[0]
set_property -dict { PACKAGE_PIN M19    IOSTANDARD LVCMOS33 } [get_ports { sw[1] }]; #IO_L7P_T1_AD2P_35 Sch=sw[1]

##RGB LEDs

set_property -dict { PACKAGE_PIN L15    IOSTANDARD LVCMOS33 } [get_ports { T1[0] }]; #IO_L22N_T3_AD7N_35 Sch=led4_b
set_property -dict { PACKAGE_PIN G17    IOSTANDARD LVCMOS33 } [get_ports { T1[1] }]; #IO_L16P_T2_35 Sch=led4_g
set_property -dict { PACKAGE_PIN N15    IOSTANDARD LVCMOS33 } [get_ports { T1[2] }]; #IO_L21P_T3_DQS_AD14P_35 Sch=led4_r
set_property -dict { PACKAGE_PIN G14    IOSTANDARD LVCMOS33 } [get_ports { T2[0] }]; #IO_0_35 Sch=led5_b
set_property -dict { PACKAGE_PIN L14    IOSTANDARD LVCMOS33 } [get_ports { T2[1] }]; #IO_L22P_T3_AD7P_35 Sch=led5_g
set_property -dict { PACKAGE_PIN M15    IOSTANDARD LVCMOS33 } [get_ports { T2[2] }]; #IO_L23N_T3_35 Sch=led5_r

##LEDs

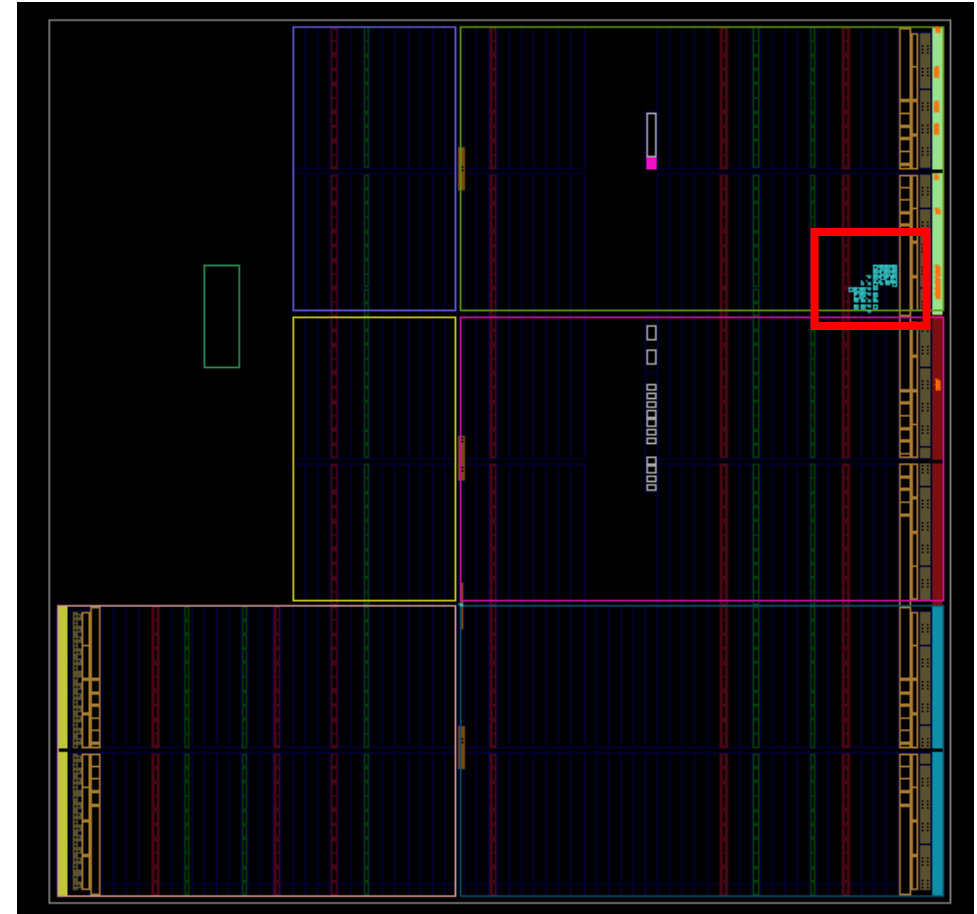
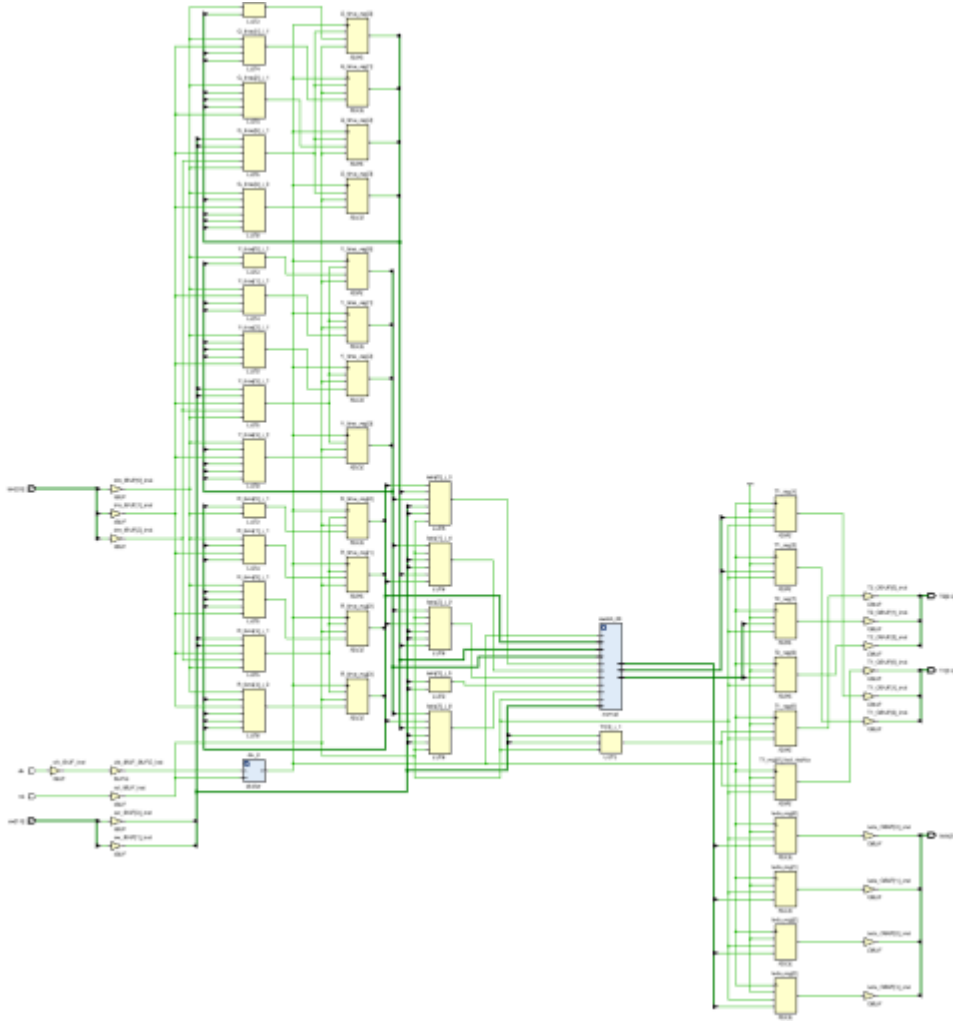
set_property -dict { PACKAGE_PIN R14    IOSTANDARD LVCMOS33 } [get_ports { leds[0] }]; #IO_L6N_T0_VREF_34 Sch=led[0]
set_property -dict { PACKAGE_PIN P14    IOSTANDARD LVCMOS33 } [get_ports { leds[1] }]; #IO_L6P_T0_34 Sch=led[1]
set_property -dict { PACKAGE_PIN N16    IOSTANDARD LVCMOS33 } [get_ports { leds[2] }]; #IO_L21N_T3_DQS_AD14N_35 Sch=led[2]
set_property -dict { PACKAGE_PIN M14    IOSTANDARD LVCMOS33 } [get_ports { leds[3] }]; #IO_L23P_T3_35 Sch=led[3]

##Buttons

set_property -dict { PACKAGE_PIN D19    IOSTANDARD LVCMOS33 } [get_ports { btn[0] }]; #IO_L4P_T0_35 Sch=btn[0]
set_property -dict { PACKAGE_PIN D20    IOSTANDARD LVCMOS33 } [get_ports { btn[1] }]; #IO_L4N_T0_35 Sch=btn[1]
set_property -dict { PACKAGE_PIN L20    IOSTANDARD LVCMOS33 } [get_ports { btn[2] }]; #IO_L9N_T1_DQS_AD3N_35 Sch=btn[2]
set_property -dict { PACKAGE_PIN L19    IOSTANDARD LVCMOS33 } [get_ports { rst }]; #IO_L9P_T1_DQS_AD3P_35 Sch=btn[3]
```

Global reset

Floorplan and Gate-Level netlist



Video Time code

- Demo_Video
- [00:24] Problem2 Traffic Light
- [00:26] Switch切換模式示範並以預設時間跑一輪
- [01:16] Button加減時間以及重置示範並以修改後的時間跑一輪
- [03:23] Global Reset示範

Problem 3

1. 為什麼要加入 blinky. xdc ?

- Xdc檔主要用於Synthesis 和Implementation階段，可以使 FPGA 設計能夠達到正確的Timing與Placement&Routing要求。
- Xdc 檔主要包含兩大類約束：

1) 時序約束 (Timing Constraints)

確保 FPGA 設計能夠滿足時序要求，在STA時正確計算Setup time跟 Hold time避免 violation發生。

2) 物理約束 (Physical Constraints)

確保 FPGA 設計的 I/O 腳位分配、Placement、Routing能夠符合 FPGA 硬體限制。

blinky.xdc

```
1 create_clock -period 8.000 -name sys_clk_pin -waveform {0.000 4.000} -add [get_ports clk];  
2 create_generated_clock -name clk_div -divide_by 125000000 -source [get_ports clk] [get_pins div_0/clk_div_reg/Q];
```

- 第一行定義了一個 8ns週期 (125 MHz)、Duty cycle 50%的clk，並將其綁定到clk這個 Port。
- 第二行建立了一個由clk除頻125000000倍的generated clk(clk_div)，並將其來源設為 div_0/clk_div_reg/Q。
 - div_0代表著Clock divider的instance name
 - clk_div_reg為內部的一個D Flip-Flop，用於儲存clk除頻的結果
 - Q這代表D型 Flip-Flop 的輸出端，通常對應於 clk_div_reg 的輸出

create_clock 定義了Primary Clock，用於計算 Setup/Hold 時序。

create_generated_clock 定義了衍生時脈 (Generated Clock)，可以確保 EDA工具正確分析除頻後的時序關係。

2. Vivado 的開發流程中 Synthesis 和 Implementation 的結果差異在哪？

▣ Synthesis 主要是將 RTL code 轉換為 Gate-Level netlist。

輸出結果包含：

- Netlist, 但未映射到具體的FPGA佈局。
- Synthesis後的時序報告, 但不包含Placement與Routing影響。

▣ Implementation 則是將 Synthesis 產生的 Netlist, 根據 FPGA 的實際電路, 進行 Placement 與 Routing。

輸出結果包含：

- Placed & Routed Netlist。
- Timing Report, 這時的時序分析是真正考慮了 FPGA 內部的 Placement 與 Routing 的延遲。

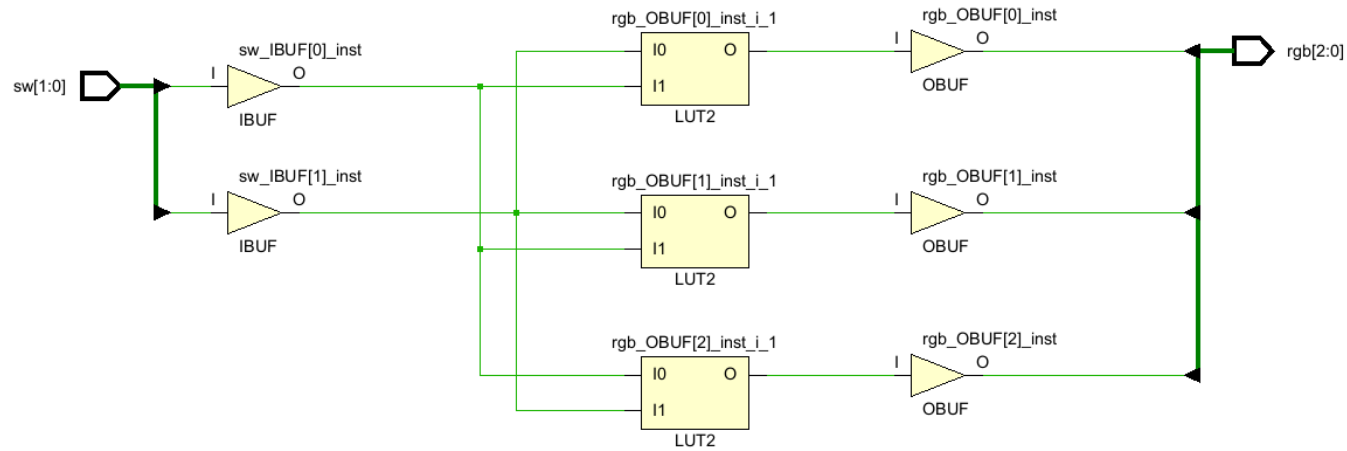
總結來說， Synthesis和Implementation的結果差異為：

Synthesis出來的Gate-Level Netlist， 已經會有LUT、 FF、 MUX等等FPGA具有的電路， 但未根據實際使用的FPGA內部電路做Placement。

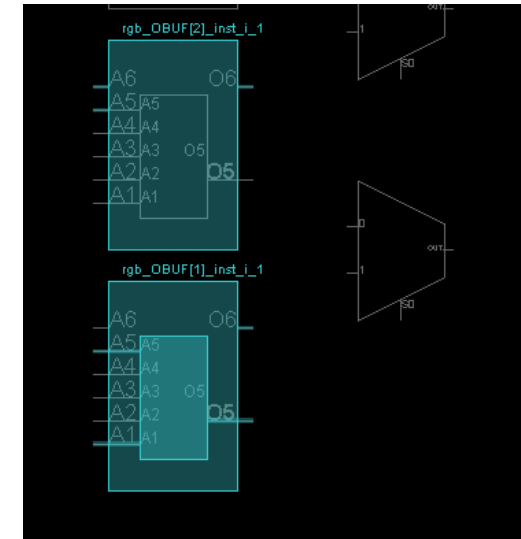


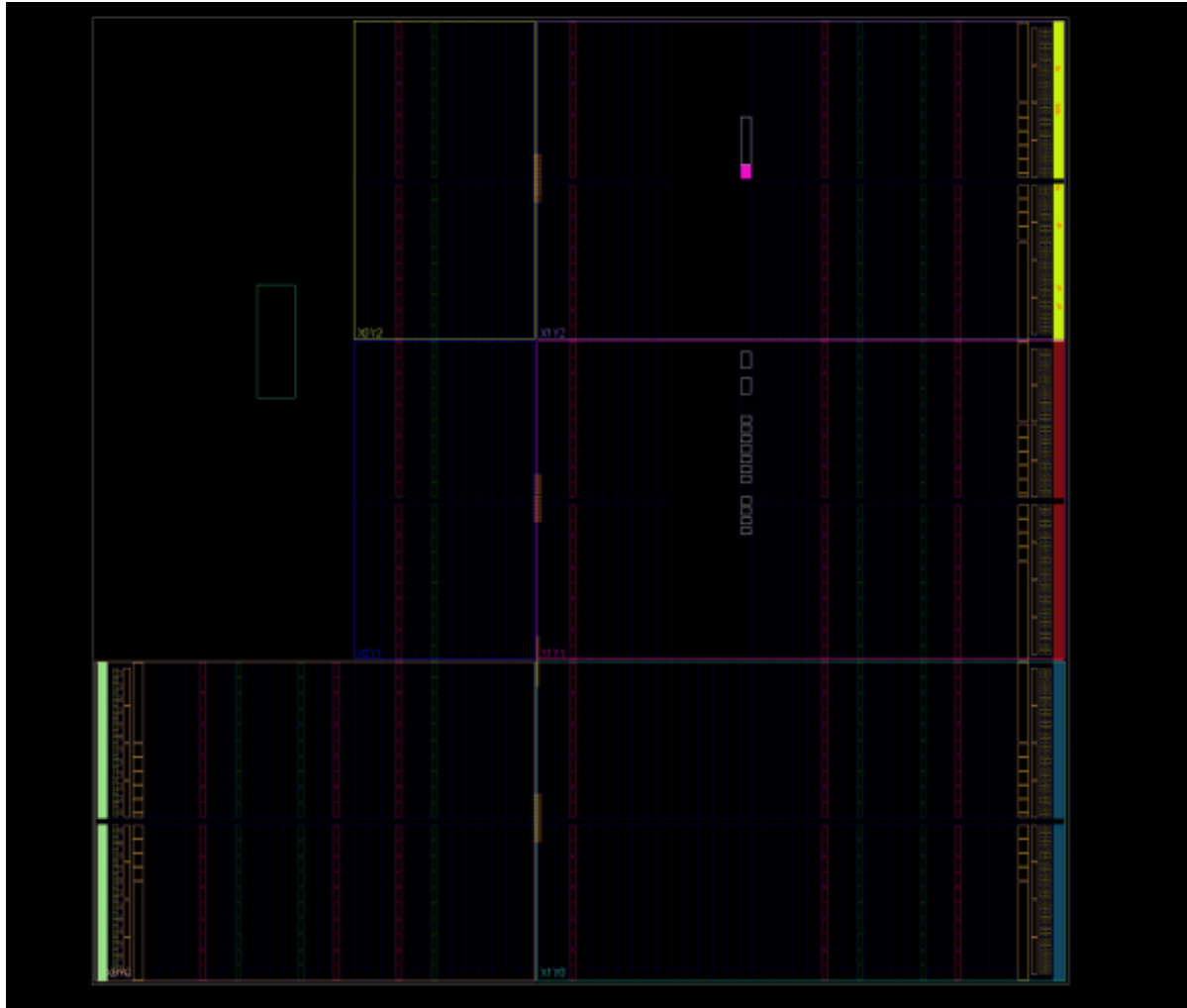
Implementation則是將Synthesis出來的Netlist進行Placement&Routing， 並根據你實際使用的FPGA做LUT的拆分配置等等。

Synthesis後得到的 Gate-Level netlist

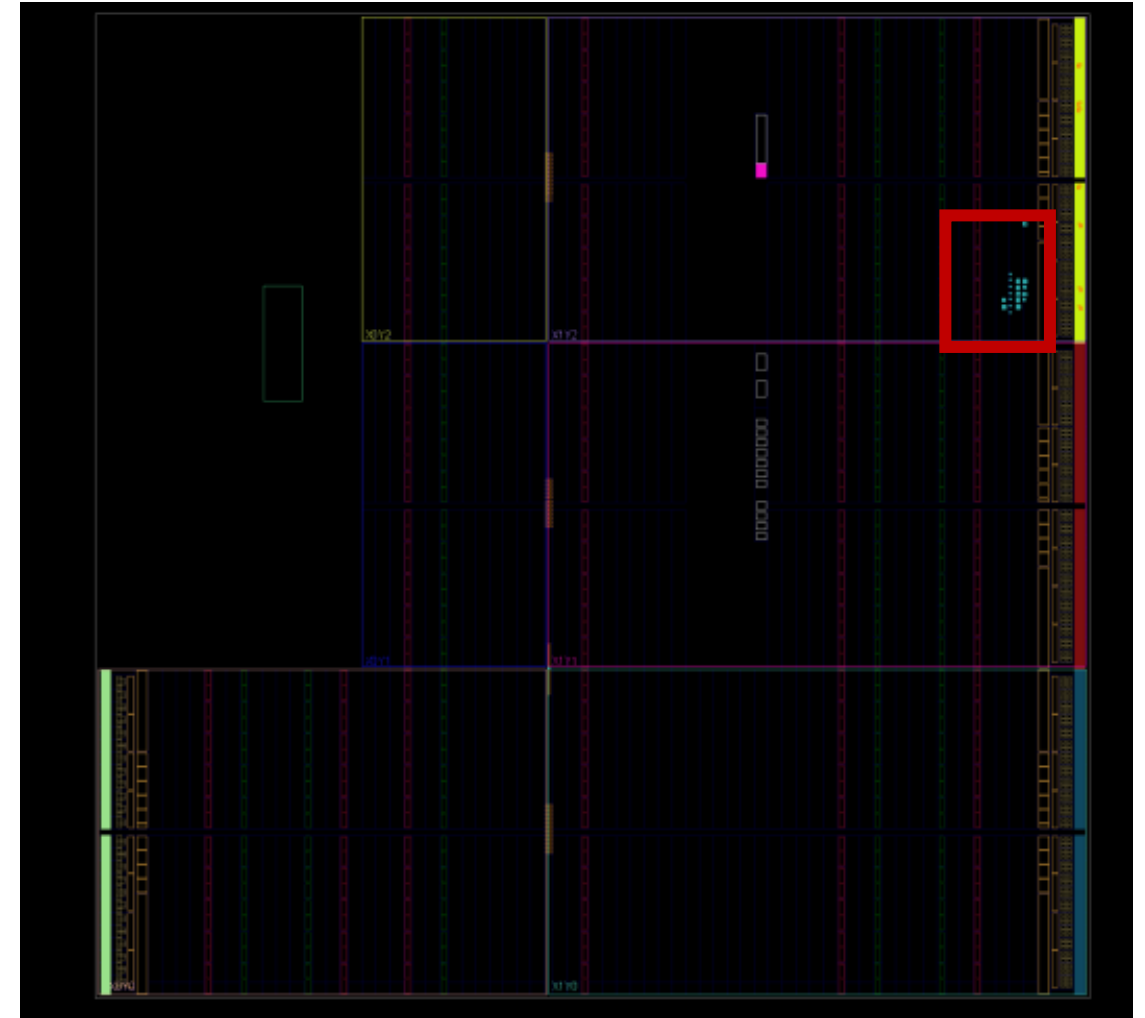


Implementation後得到FPGA內部電路 實際的使用圖





Floorplan
After synthesis



Floorplan
After implementation