時序電路設計及應用

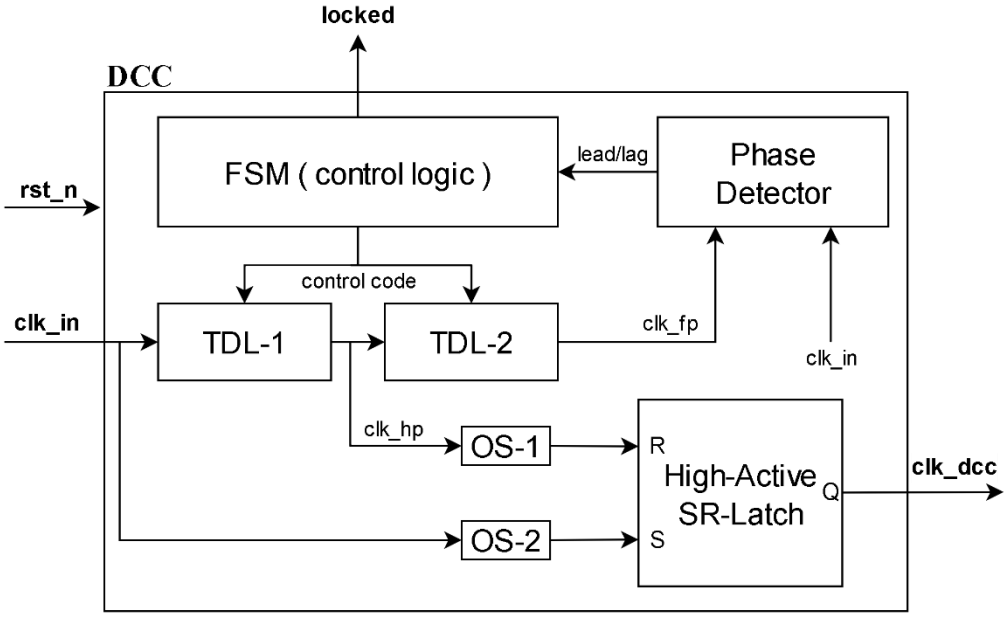
Homework 3

姓名：王品然

學號：113063572

Consider a design process for a duty-cycle corrector for a 1GHz input clock signal, denoted as *clk_in*. The output clock signal is denoted as *clk_dcc*. Once locked, *clk_dcc* will have a duty cycle very close to 50%.

(a) (20%) Draw a **block diagram** of your design and summarizing how you plan to achieve the duty-cycle correction with a paragraph.

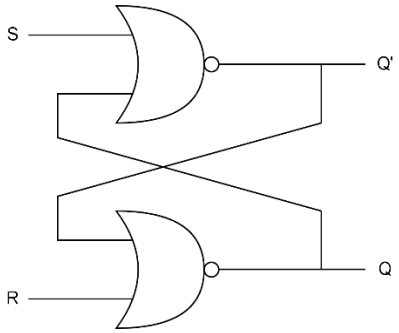


使用如上所示的 block diagram 可以實現 duty cycle 的修正，上半部的電路架構如同 Delay Locked Loop (DLL)，clk_in 訊號會經過兩個相同的 Tunable Delay Line (TDL)，且兩 TDL 由相同的 control code 所控制，從第二個 TDL 輸出的訊號(clk_fp)與 clk_in 當作 Phase Detector (PD)的輸入訊號，PD 會比較出現在 clk_fp 的 phase 是領先的或是落後的，並輸出 lead/lag 訊號做為指示 controller 做出 control code 改變的依據，在這個疊代的過程中，clk_in 和 clk_fp 的 phase error 會越來越小，直到 controller 鎖定 control code 確定 clk_fp 是 clk_in **延遲一個 cycle** 的訊號。這裡需要特別注意一般的 DLL 可以鎖定在 clk_fp 為 clk_in 的整數倍周期的延遲，但這邊一定要是鎖定在一個週期，原因是要讓 clk_hp 剛好延遲**半個**週期。在 controller 做到以上工作後，讓 clk_in 與 clk_hp 分別經過一個 one shot circuit，以避免 clk_in 的 duty cycle 過大造成輸入 active high SR-latch 時，出現兩個輸入都為 1 的不合理情況。如此一來在最後一級的 SR-latch 會出現規律的(S,R)=(1,0)和(S,R)=(0,1)，每隔半周期交替出現，並在剩餘期間維持(S,R)=(0,0)，也就是說每半周期 output(Q)會 toggle 一次並維持訊號直到下次 toggle，如此就完成了 duty cycle correction 的任務了。

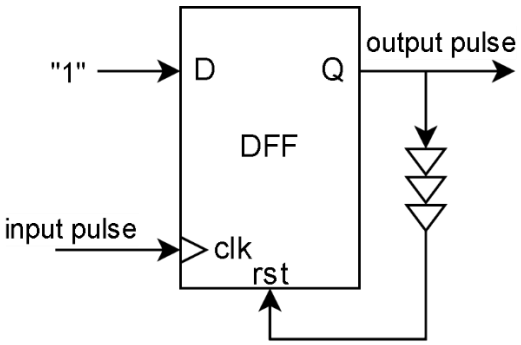
(b) (40%) **Realize the design as a synthesizable Verilog code**. (Your design can be a mixed format containing both netlists and RTL codes). (Hint: you may need Tunable Delay Lines, A Phase Detector, and a Controller, etc.) In this homework, we only need to use simple TDLs, e.g., path-selection-based TDLs, and some cell-based Phase Detector). Estimate your time resolution.

將(a)的 block diagram 中的子電路一一實現，以下展示電路圖或其架構，最後再附上 code：

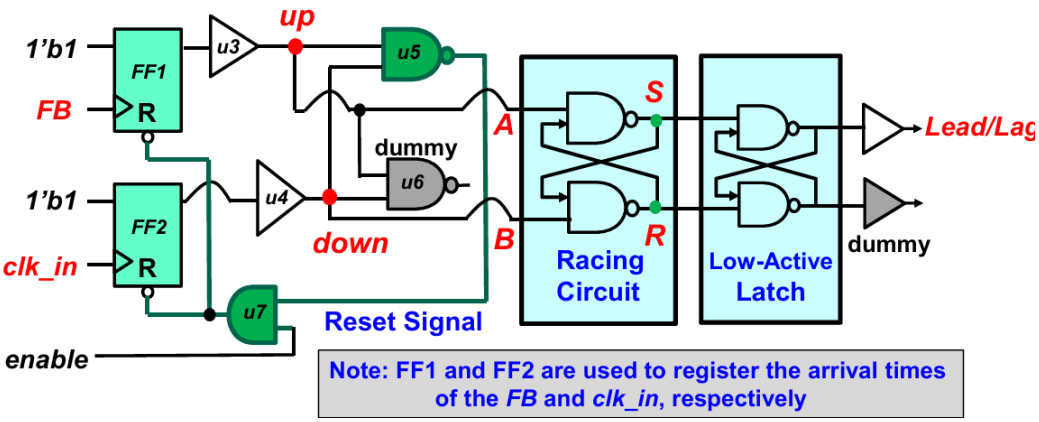
(1) High-Active SR-Latch



(2) One Shot Based Oscillator

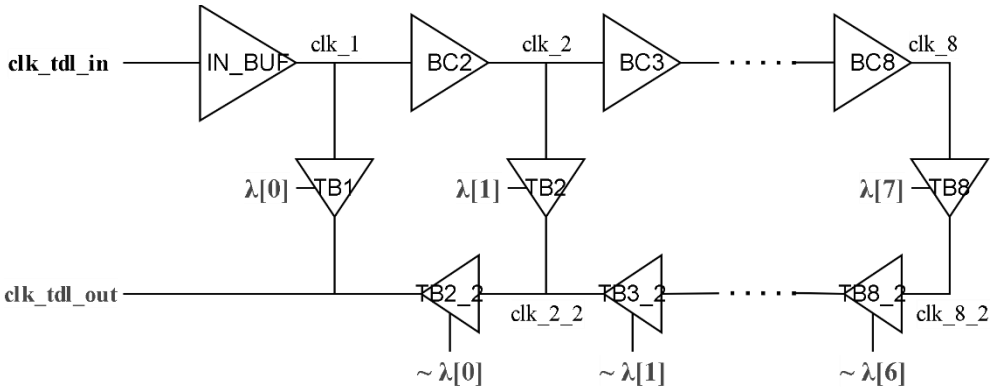


(3) Phase Detector



此電路圖截自講義

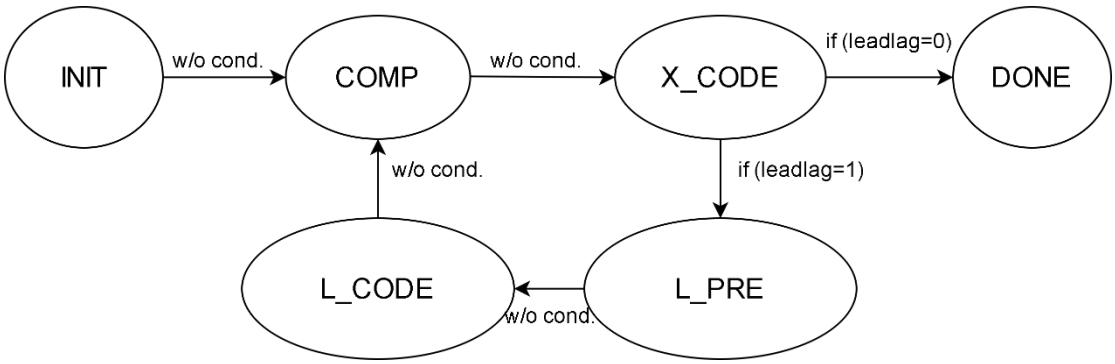
(4) Tunable Delay Line



最終規格如下：IN_BUF 是 4 個 X1 的 CLKBUF 的 buffer chain；BC 是 2 個 X1 的 CLKBUF 的 buffer chain；TB 是一個 TBUFX4。

(5) Controller (FSM)

這次電路的起始狀態會在 λ[7:0]=8'b0000_0001 也就是 x[2:0]=3'b000 的情況，目的是為了讓 TDL 的 delay 先設定到最小，再慢慢(一次+1)尋找 lead/lag 訊號由 1 變 0 的時候，即為目標的 control code，得到 clk_hp 接近是 clk_in 的半周期延遲訊號，並在此時讓 locked=1。以此為中心思想，又因為考慮到輸入時脈訊號為 1GHz，不算低頻的訊號，為避免後續可能有 setup time violation，因此先想辦法讓每個 state 做的事越少越好，所以將 control code 的決定分成了兩個步驟，並且還加入了一個中間 state (下圖中的 L_PRE)，目的是對即將轉變的通道做預先開啟的動作，以避免 unknown 的狀態發生，在上次作業中有較詳細解釋過理由。以下即是作為 controller 的 FSM 架構：



INIT state: DCC reset 後會到這個 state，主要負責設定電路內的初始值。

COMP state: 拉起 enable PD 的訊號，並等待 lead/lag 訊號穩定確定下來，接著在下一個 state 讀取。

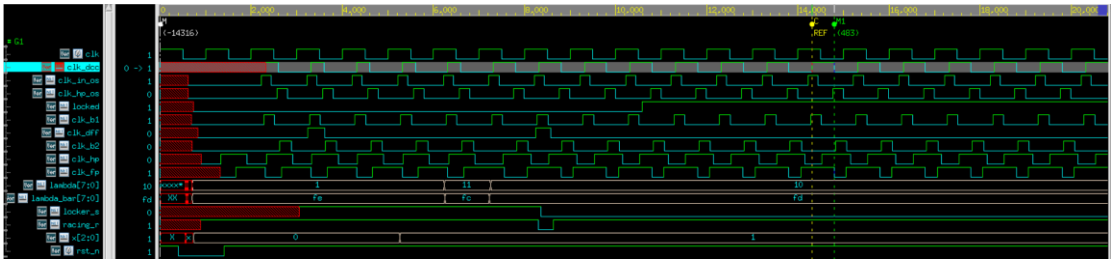
X_CODE state: 依據 lead/lag 訊號，決定 x[2:0]，並停下 PD。

L_PRE state: 在更新 λ[7:0]之前，先預先打開通道為未知節點做充放電，避免 unknown signal 的出現。

L_CODE state: 將 x[2:0]轉換成的 one-hot code 更新 λ[7:0]。

DONE state: 表示電路已經準備好完成 dcc 之 clock 訊號。

在驗證電路功能時發現 input clock 的 duty cycle 為 50%時，還是功能正常的電路，如下圖：



到了 input clock 的 duty cycle 為 70%時，卻發生 timing violation，波型圖與 error message 如下圖所示：

```
"/usr/cadtool/ee5216/CBDK_TSMC90GUTM_Arm_f1.0/CIC/Verilog/tsmc090.v", 11018: Timing violation in testbench.u_dcc.clk_hp_os_reg
    $width( negedge CK &&& (flag == 1'b1):8351,  : 8532, limit: 186 );
```

可以發現是因為 clk_fp 為 0 的時間太短了，儘管已經選用 CLKBUF 作為 buffer，但還是因為經過許多 buffer 和 tri-state buufer 後 skew 得越來越嚴重，為解決此問題我選擇減少 buffer 數量，並同時調小 buffer 的尺寸，以獲得差不多的 delay time 但較少的 skew，最終規格是如上 TDL 架構時提過的數量及尺寸。

TDL netlist:

```
module delay_buffer_chain ( out, in );
  input in;
  output out;
  wire n1,n2,n3;
  CLKBUFX1 U0 ( .A(in), .Y(n1)  );
  CLKBUFX1 U1 ( .A(n1), .Y(n2)  );
  CLKBUFX1 U2 ( .A(n2), .Y(n3)  );

  CLKBUFX1 U4 ( .A(n3), .Y(out)  );

endmodule

module delay_buffer ( out, in );
  input in;
  output out;
  wire n1;
  CLKBUFX1 U0 ( .A(in), .Y(n1)  );
  CLKBUFX1 U4 ( .A(n1), .Y(out)  );

endmodule



module tdl (clk_in, lambda, lambda_bar, clk_out);
  input [7:0] lambda;
  input [7:0] lambda_bar;
  input clk_in;
  output clk_out;
  wire   clk_1, clk_2, clk_3, clk_4, clk_5, clk_6, clk_7, clk_8,
         clk_2_2, clk_3_2, clk_4_2, clk_5_2,clk_6_2, clk_7_2, clk_8_2, clk_out;


  delay_buffer_chain bc1 ( .out(clk_1), .in(clk_in) );
  delay_buffer bc2 ( .out(clk_2), .in(clk_1) );
  delay_buffer bc3 ( .out(clk_3), .in(clk_2) );
  delay_buffer bc4 ( .out(clk_4), .in(clk_3) );
  delay_buffer bc5 ( .out(clk_5), .in(clk_4) );
  delay_buffer bc6 ( .out(clk_6), .in(clk_5) );
  delay_buffer bc7 ( .out(clk_7), .in(clk_6) );
  delay_buffer bc8 ( .out(clk_8), .in(clk_7) );

  TBUFX4 buf_1 (.A(clk_1), .Y(clk_out), .OE(lambda[0]));
  TBUFX4 buf_2 (.A(clk_2), .Y(clk_2_2), .OE(lambda[1]));
  TBUFX4 buf_2_2 (.A(clk_2_2), .Y(clk_out), .OE(lambda_bar[0]));
  TBUFX4 buf_3 (.A(clk_3), .Y(clk_3_2), .OE(lambda[2]));
  TBUFX4 buf_3_2 (.A(clk_3_2), .Y(clk_2_2), .OE(lambda_bar[1]));
  TBUFX4 buf_4 (.A(clk_4), .Y(clk_4_2), .OE(lambda[3]));
  TBUFX4 buf_4_2 (.A(clk_4_2), .Y(clk_3_2), .OE(lambda_bar[2]));
  TBUFX4 buf_5 (.A(clk_5), .Y(clk_5_2), .OE(lambda[4]));
  TBUFX4 buf_5_2 (.A(clk_5_2), .Y(clk_4_2), .OE(lambda_bar[3]));
  TBUFX4 buf_6 (.A(clk_6), .Y(clk_6_2), .OE(lambda[5]));
  TBUFX4 buf_6_2 (.A(clk_6_2), .Y(clk_5_2), .OE(lambda_bar[4]));
  TBUFX4 buf_7 (.A(clk_7), .Y(clk_7_2), .OE(lambda[6]));
  TBUFX4 buf_7_2 (.A(clk_7_2), .Y(clk_6_2), .OE(lambda_bar[5]));
  TBUFX4 buf_8 (.A(clk_8), .Y(clk_8_2), .OE(lambda[7]));
  TBUFX4 buf_8_2 (.A(clk_8_2), .Y(clk_7_2), .OE(lambda_bar[6]));


endmodule
```

DCC RTL code:

```
`include "tdl.v"

module dcc (
    output  clk_dcc,
    output  locked,
    input   clk,
    input   rst_n
);

//////////////////////
//DECLARATION
//////////////////////
//TDL signals
wire clk_fp, clk_hp;
reg  [7:0] lambda;
reg  [7:0] lambda_bar;
reg  [7:0] lambda_next, lambda_prenext;
wire [7:0] lambda_bar_next, lambda_bar_prenext;
reg  [2:0]  x;

//PD signals
wire direction;
reg  clk_dff;
reg  clk_fp_dff;
wire rst_pd;
wire rst_pd_flag;
wire pd_en;
wire pd_dummy;
wire racing_s, racing_r;
wire locker_s, locker_r;

//one shot circuit
reg clk_in_os, clk_hp_os;
wire rst_os1, rst_os2;
wire clk_b1, clk_b2;

//output SR latch
wire q, q_bar;

//state
localparam ST_INIT    = 3'b000;
localparam ST_X_CODE  = 3'b001;
localparam ST_L_PRE   = 3'b010;
localparam ST_L_CODE  = 3'b011;
localparam ST_COMP    = 3'b100;

localparam ST_DONE    = 3'b110;

reg   [2:0] state;
reg   [2:0] state_nx;

//////////////////////
//FSM
//////////////////////
//setting next state
always@(posedge clk or negedge rst_n)begin
   if(~rst_n)  state <= ST_INIT;
   else        state <= state_nx;
end

//decision of next state
always@(*)begin
   state_nx = state;
   case(state)
   ST_INIT  : state_nx = ST_COMP;
   ST_X_CODE :
   begin
       if(direction) state_nx = ST_L_PRE;
       else state_nx = ST_DONE;
   end
   ST_L_PRE : state_nx = ST_L_CODE;
   ST_L_CODE : state_nx = ST_COMP;
```

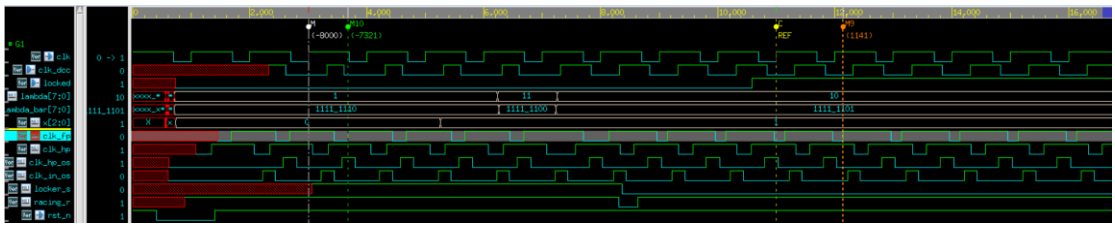
```
   ST_COMP  : state_nx = ST_X_CODE;

   ST_DONE  : state_nx = state;
   default  : state_nx = state;
   endcase
end

//output controlled by FSM
assign locked = (state == ST_DONE);
assign pd_en = (state==ST_COMP);   //for PD

//////////////////////
//Searching process
//////////////////////
//lambda code decision - x value
always@(posedge clk or negedge rst_n) begin
    if(~rst_n)    x <= 3'b000;
    else begin
        case(state)
            ST_INIT  : x <= 3'b000;
            ST_X_CODE  :
                begin
                    if(direction) x <= x + 1;
                    else x <= x;    //or x-1
                end
            ST_L_PRE, ST_L_CODE, ST_COMP, ST_DONE  : x <= x;
            default  : x <= x;
        endcase
    end
end


//lambda comb. logic
always@(*) begin
    case(x)
        3'b000: lambda_next = 8'b00000001;
        3'b001: lambda_next = 8'b00000010;
        3'b010: lambda_next = 8'b00000100;
        3'b011: lambda_next = 8'b00001000;
        3'b100: lambda_next = 8'b00010000;
        3'b101: lambda_next = 8'b00100000;
        3'b110: lambda_next = 8'b01000000;
        3'b111: lambda_next = 8'b10000000;
        default: lambda_next = 8'b00000010;
    endcase
end

assign lambda_bar_next = ~ lambda_next;

always@(*) begin
    case(x)
        3'b000: lambda_prenext = 8'b00000001;
        3'b001: lambda_prenext = 8'b00000011;
        3'b010: lambda_prenext = 8'b00000110;
        3'b011: lambda_prenext = 8'b00001100;
        3'b100: lambda_prenext = 8'b00011000;
        3'b101: lambda_prenext = 8'b00110000;
        3'b110: lambda_prenext = 8'b01100000;
        3'b111: lambda_prenext = 8'b11000000;
        default: lambda_prenext = 8'b00000011;
    endcase
end

assign lambda_bar_prenext = ~ lambda_prenext;


//lambda update
always@(posedge clk or negedge rst_n) begin
    if(~rst_n) begin
        lambda <= 8'b00000001;
        lambda_bar <= 8'b11111110;
    end
    else begin
        case(state)
            ST_INIT          : begin
            lambda <= 8'b00000001;
            lambda_bar <= 8'b11111110;
            end
            ST_X_CODE, ST_COMP, ST_DONE : begin
            lambda <= lambda;
            lambda_bar <= lambda_bar;
            end
            ST_L_PRE  : begin
            lambda <= lambda_prenext;
            lambda_bar <= lambda_bar_prenext;
            end
            ST_L_CODE  : begin
            lambda <= lambda_next;
            lambda_bar <= lambda_bar_next;
            end
            default          : begin
            lambda <= lambda;
            lambda_bar <= lambda_bar;
            end
        endcase
    end
end


//////////////////////
//Phase Detector
//////////////////////
//PD
always @(posedge clk_fp or negedge rst_pd) begin
    if(~rst_pd) clk_fp_dff <= 0;
    else clk_fp_dff <= 1;
end

always @(posedge clk or negedge rst_pd) begin
    if(~rst_pd) clk_dff <= 0;
    else clk_dff <= 1;
end

assign rst_pd_flag = ~ (clk_fp_dff & clk_dff);
assign pd_dummy = ~(clk_dff & clk_fp_dff);
assign rst_pd = pd_en & rst_pd_flag;

//SR latch as racing circuit
assign racing_s = ~ (racing_r & clk_fp_dff);
assign racing_r = ~ (racing_s & clk_dff);

//SR latch as signal locker
assign locker_s = ~ (locker_r & racing_s);
assign locker_r = ~ (locker_s & racing_r);

//direction = 1 if clk_fp is faster, which means we need to tune a
assign direction = locker_s;

//////////////////////
//One Shot Circuit
//////////////////////
//one-shot-based osc. 1
always @(posedge clk or negedge rst_os1 or negedge rst_n) begin
    if(~rst_n) clk_in_os <= 0;
    else if(~rst_os1) clk_in_os <= 0;
    else clk_in_os <= 1;
end

delay_buffer ub_1 (.out(clk_b1), .in (clk_in_os));
assign rst_os1 = ~clk_b1;

//one-shot-based osc. 2
always @(posedge clk_hp or negedge rst_os2 or negedge rst_n) begin
    if(~rst_n) clk_hp_os <= 0;
```

```
    else if(~rst_os2) clk_hp_os <= 0;
    else clk_hp_os <= 1;
end

delay_buffer ub_2 (.out(clk_b2), .in (clk_hp_os));
assign rst_os2 = ~clk_b2;


//////////////////////
//Tunable Delay Line
//////////////////////
//instantiate
tdl tdl_1 (
    .clk_out (clk_hp),
    .clk_in  (clk),
    .lambda  (lambda),
    .lambda_bar (lambda_bar)
);
```

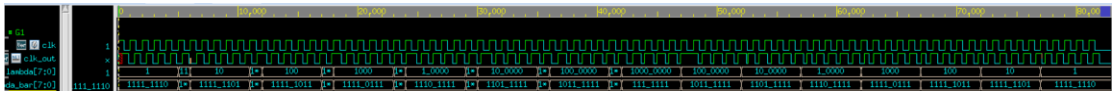
```
tdl tdl_2 (
    .clk_out (clk_fp),
    .clk_in  (clk_hp),
    .lambda  (lambda),
    .lambda_bar (lambda_bar)
);

//////////////////////
//Output Stage SR Latch
//////////////////////
assign q_bar = ~ (clk_in_os | q);
assign q = ~ (clk_hp_os | q_bar);

assign clk_dcc = q;


endmodule
```

Time resolution estimation:

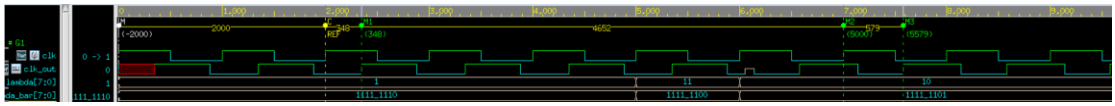
在模擬時可以看到當 lambda code 從 8'b0000_0001 到 8'b0000_0010，兩條 TDL 的 delay time 總和在兩個情況下相差 1141-679=462ps，波形如下圖：



對單條 TDL 做 gate-level simulation，得到波形如下圖：



局部放大圖：



可以發現當 lambda code 從 8'b0000_0001 到 8'b0000_0010，單條 TDL 的 delay time 在兩個情況下相差 579-348=231ps (兩條就會是 462ps)，與實際應用在 DCC 電路中的結果是相符的。之後的 lambda code 每增加 1 格，TDL 的 delay time 就會增加 230ps，篇幅關係就不貼上局部的波形圖。

因此結論是單個 TDL 的 time resolution 大約在 230 到 231ps 之間，若看兩個 TDL 則 time resolution 大約在 460 到 462ps 之間。

(c) (15%) Try to use Design Compiler to **synthesize your design into standard-cell netlists**. Report the gate count of your netlist.

合成時使用的.tcl 檔：

```
set ref_cycle 1
set DESIGN_TOP dcc
#/*-------------------------------------------------------------*/
#/*----------------------- 1.Read files ------------------------*/
#/*-------------------------------------------------------------*/
analyze -format verilog {../1.RTL_simulation/dcc.v}
elaborate $DESIGN_TOP
set_operating_conditions -min_library fast -min fast \
                         -max_library slow -max slow
#/*-------------------------------------------------------------*/
#/*--------------- 2. Set design constraints -------------------*/
#/*-------------------------------------------------------------*/
create_clock -name CLK -period $ref_cycle [get_ports clk]
set_dont_touch_network [get_ports clk]
set_fix_hold [get_clocks CLK]
# input drive / output load
set_drive 1 [all_inputs]
set_load  [load_of slow/CLKBUFX20/A] [all_outputs]
#/*-------------------------------------------------------------*/
#/*------- 2.5 保護 TDL（不要讓 DC 動裡面的 gate） ------------*/
#/*-------------------------------------------------------------*/
set_dont_touch [get_designs tdl]
set_dont_touch [get_cells -hierarchical tdl_1]
set_dont_touch [get_cells -hierarchical tdl_2]
#/*-------------------------------------------------------------*/
#/*----------------- 3.Check and Link Design -------------------*/
#/*-------------------------------------------------------------*/
link
check_design
uniquify
set_fix_multiple_port_nets -all -buffer_constants
#/*-------------------------------------------------------------*/
#/*----------------------- 4.Compile ---------------------------*/
#/*-------------------------------------------------------------*/
# Setting DRC Constraint
set_max_fanout 20.0 $DESIGN_TOP
# Area Constraint
set_max_area   0
# before synthesis settings
set case_analysis_with_logic_constants true
set_fix_multiple_port_nets -all -buffer_constants
set_clock_gating_style -max_fanout 10
compile -map_effort medium
# remove dummy ports
remove_unconnected_ports [get_cells -hierarchical *]
remove_unconnected_ports [get_cells -hierarchical *] -blast_buses
check_design
# rename / bus style
set bus_inference_style {%s[%d]}
set bus_naming_style {%s[%d]}
set hdlout_internal_busses true
set verilogout_show_unconnected_pins true
#/*-------------------------------------------------------------*/
#/*----------------------- 5.Write out files -------------------*/
#/*-------------------------------------------------------------*/
report_area                              >  ./dcc_syn.report
report_timing -path full -delay max      >> ./dcc_syn.report
report_power                             >> ./dcc_syn.report
write -format verilog -hierarchy -output ./dcc_syn.v
write_sdf -version 2.1 -context verilog -load_delay cell ./dcc_syn.sdf
write_sdc ../3.Post_layout_Simulation/APR/design_data/dcc_syn.sdc
exit
```

合成出的 netlist 就不附上了，實在有點冗長。合成後的面積如下：

```
Number of ports:                          76
Number of nets:                          249
Number of cells:                         210
Number of combinational cells:           133
Number of sequential cells:               56
Number of macros/black boxes:              0
Number of buf/inv:                        57
Number of references:                     47

Combinational area:                497.448013
Buf/Inv area:                      162.993604
Noncombinational area:             643.507208
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:                  1140.955220
```

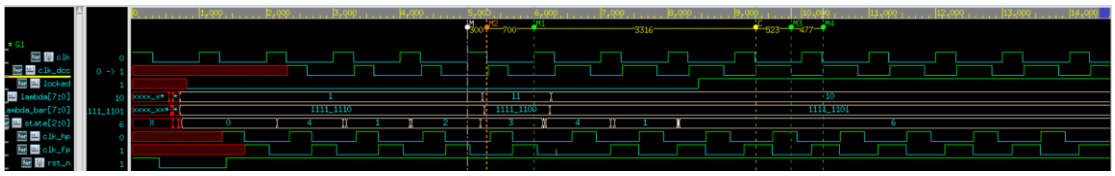
使用 total cell area 1140.95522μm^2，可以除以 2-input NAND gate 的面積 2.82μm^2，換算成 gate count 約為 404.6。

(d) (15%) Verify the correctness of your design by running gate-level Verilog simulation using as accurate delay model as possible. Show the waveforms of *clk_in* and *clk_dcc*. Try the following input duty cycle samples for *clk_in* – {30%, 70%}. **Report the final duty cycle for each input samples.**

Testbench 的部分很簡單，只有給予 clk 和 rst_n，並等待 locked 後結束模擬：

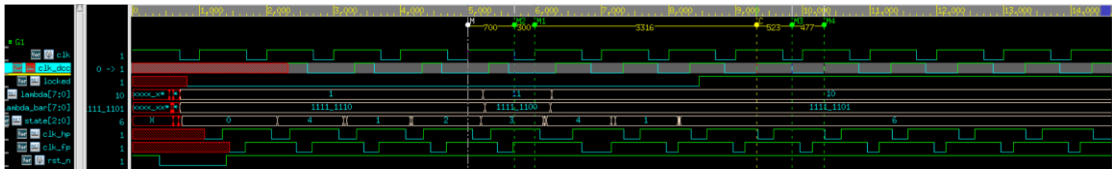
```
`timescale 100ps / 10ps
module testbench;

    localparam period = 10;
    localparam delay = 4;

    reg       clk;
    wire      clk_dcc;
    reg       rst_n;
    wire      locked;

    dcc u_dcc (
        .clk_dcc(clk_dcc),
        .locked(locked),
        .clk(clk),
        .rst_n(rst_n)
    );

    initial begin
        $sdf_annotate("./dcc_syn.sdf", u_dcc);
        $fsdbDumpfile("../4.Simulation_Result/dcc_syn.fsdb");
        $fsdbDumpvars;
    end

  initial begin
    clk = 1;
    forever begin
      #(3)  clk = 0;
      #(7) clk = 1;
    end
  end

  initial begin
        rst_n = 1;
        #(delay) rst_n = 0;
        #(period) rst_n = 1;
        wait(locked);

        #(period * 6) $finish;
    end

    // Automatically finish
    initial begin
        #30000;
        $finish;
    end

endmodule
```

輸入時脈訊號 **duty cycle 為 30%**的模擬結果：



可以看到 clk_dcc 在 DCC 鎖定後的 duty cycle=$\frac{523\ ps}{1000\ ps}$=52.3%。

輸入時脈訊號 **duty cycle 為 70%**的模擬結果：



可以看到 clk_dcc 在 DCC 鎖定後的 duty cycle=$\frac{523\ ps}{1000\ ps}$=52.3%。