Digital Logic

Lab13 Registers in Vivado

Lab13

- Register
  - shift left
  - shift right
  - rotate
  - shift to parallel converter
- Button debouncing
- Edge detection
- DON'TS in design
  - **Non-Synthesizable Verilog**

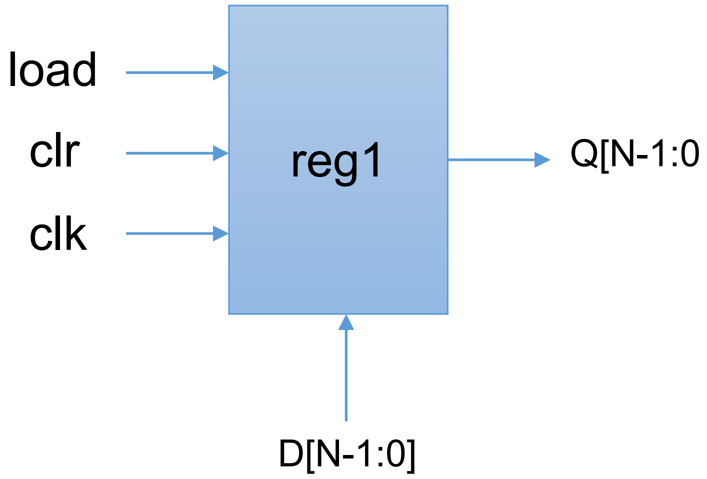
Registers

- A register consists of a group of flip-flops together with gates that affect their operation. The flip-flops hold the binary information, and the gates determine how the information is transferred into the register.

- A register capable of shifting the binary information held in each cell to its neighboring cell, in a selected direction, is called a shift register.

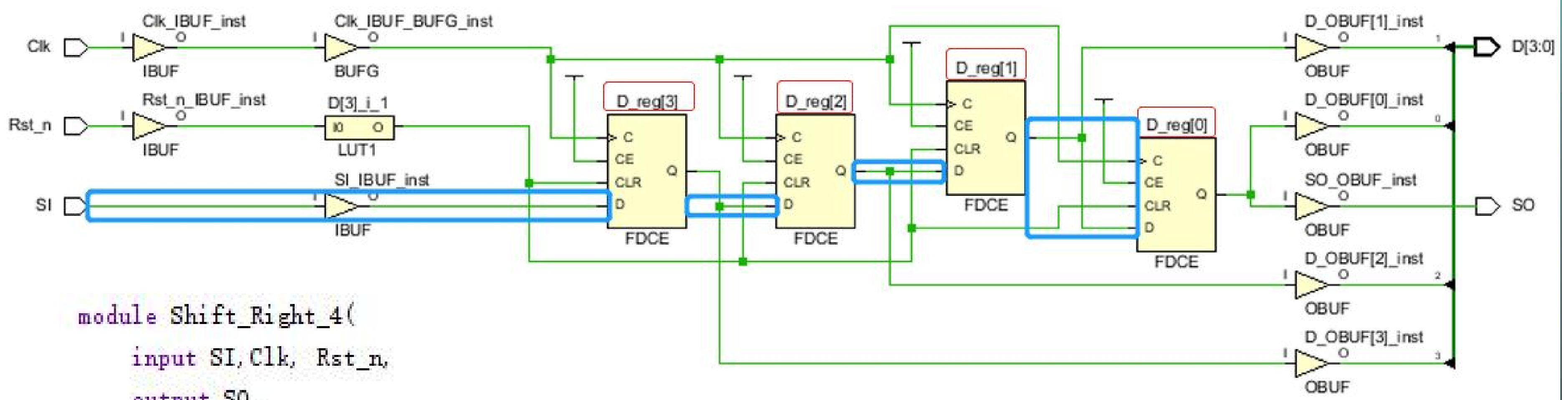
Register

- a Basic N bit register
- when clr is 0 and load is one, the input D D[N-1:0] will be transferred to Q[N-1:0] at the rising edge of clk.



```
module regN
#(parameter N = 8)
(
                input wire load, clk, clr,
                input wire [N-1:0] D,
                output reg [N-1:0] Q
);
always @(posedge clk or posedge clr)
                if(clr == 1)
                                Q <= 0;
                else if(load == 1)
                                Q <= D;
                 else           Q <= Q;
endmodule
```

Shift register(1) - shift right(1)

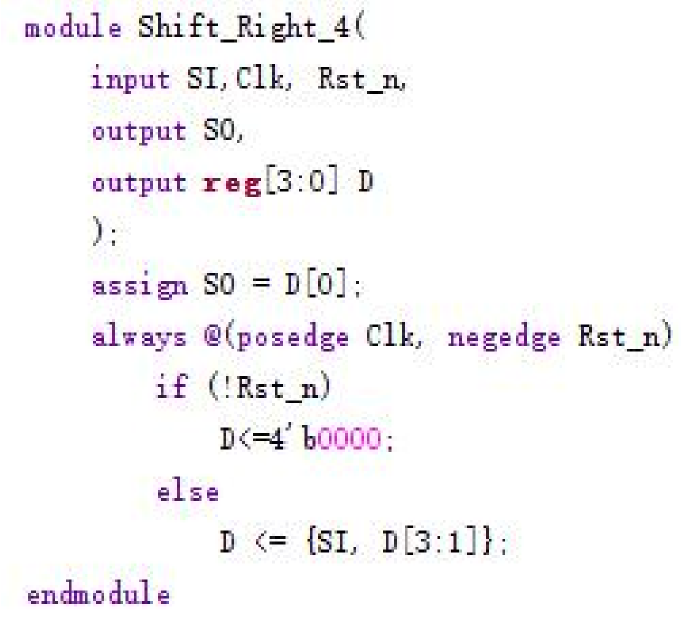


```
module Shift_Right_4(
    input SI,Clk, Rst_n,
    output SO,
    output reg[3:0] D
    );
    assign SO = D[0];
    always @(posedge Clk, negedge Rst_n)
        if (!Rst_n)
            D<=4'b0000;
        else
            D <= {SI, D[3:1]};
endmodule
```

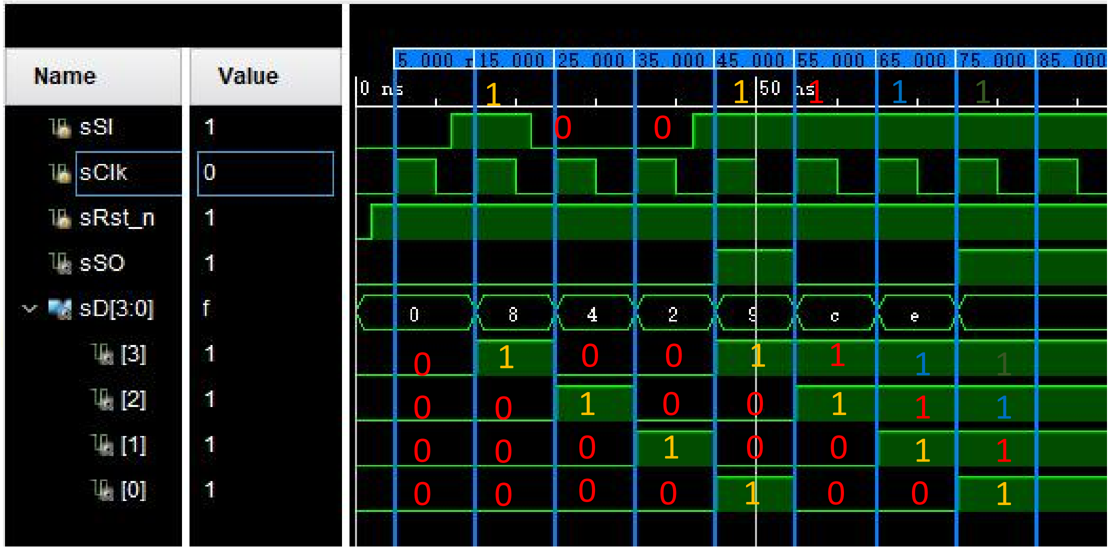
The data shift from MSB to LSB (shift right)

SI -> D[3], D[3] ->D[2], D[2]->D[1], D[1]->D[0]

Shift register(1) - shift right(2)

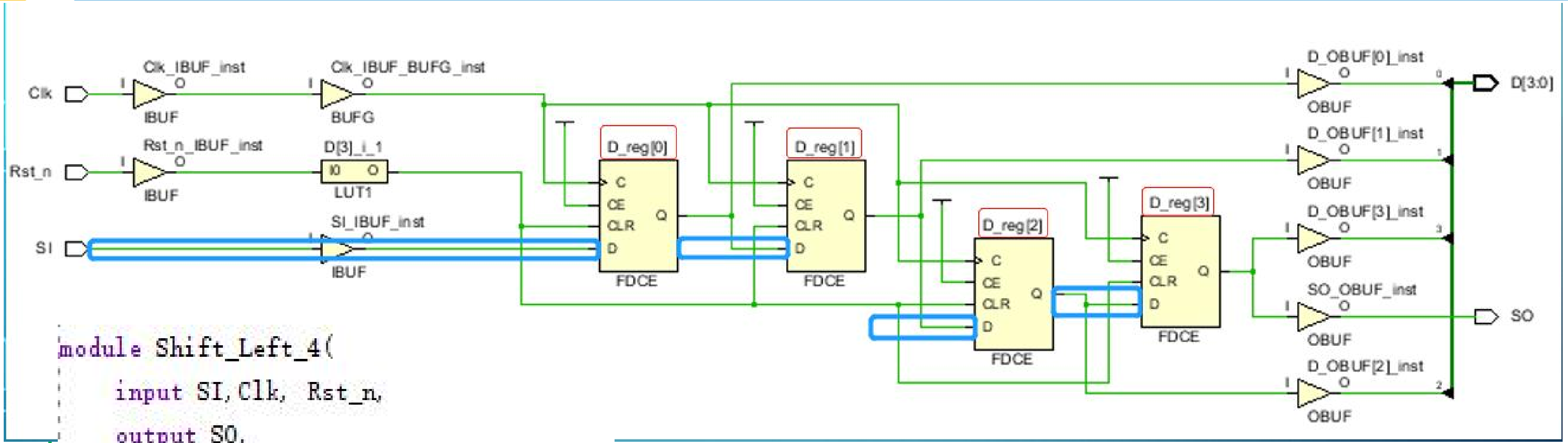


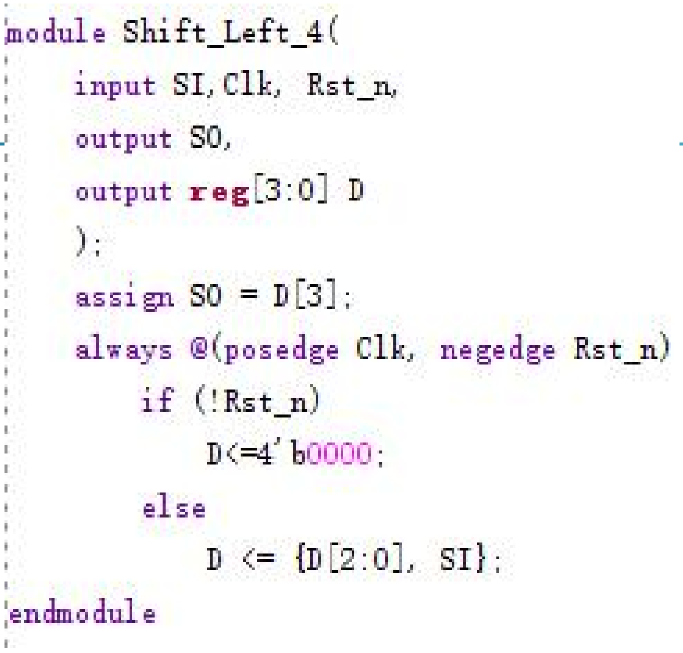
```
module Shift_Right_4(
    input SI,Clk, Rst_n,
    output SO,
    output reg[3:0] D
    );
    assign SO = D[0];
    always @(posedge Clk, negedge Rst_n)
        if (!Rst_n)
            D<=4'b0000;
        else
            D <= {SI, D[3:1]};
endmodule
```



Name	Value
sSI	1
sClk	0
sRst_n	1
sSO	1
sD[3:0]	f
[3]	1
[2]	1
[1]	1
[0]	1

Shift register(2) - shift left(1)



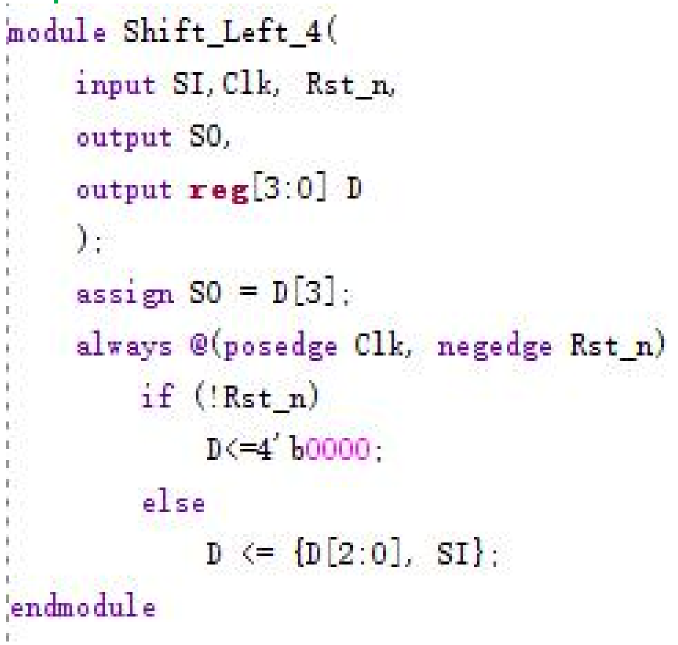


```
module Shift_Left_4(
    input SI,Clk, Rst_n,
    output SO,
    output reg[3:0] D
    );
    assign SO = D[3];
    always @(posedge Clk, negedge Rst_n)
        if (!Rst_n)
            D<=4'b0000;
        else
            D <= {D[2:0], SI};
endmodule
```

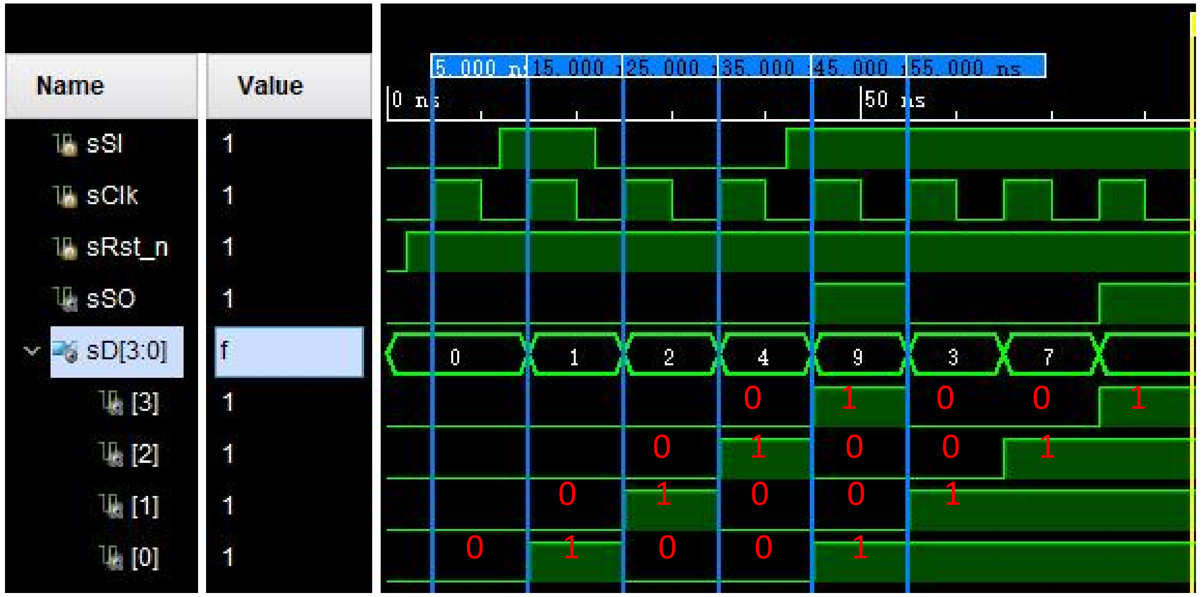
The data shift from LSB to MSB (shift left)

D[3] <- D[2], D[2] <- D[1], D[1] <- D[0] , D[0]<-S1

Shift register(2) - shift left(2)

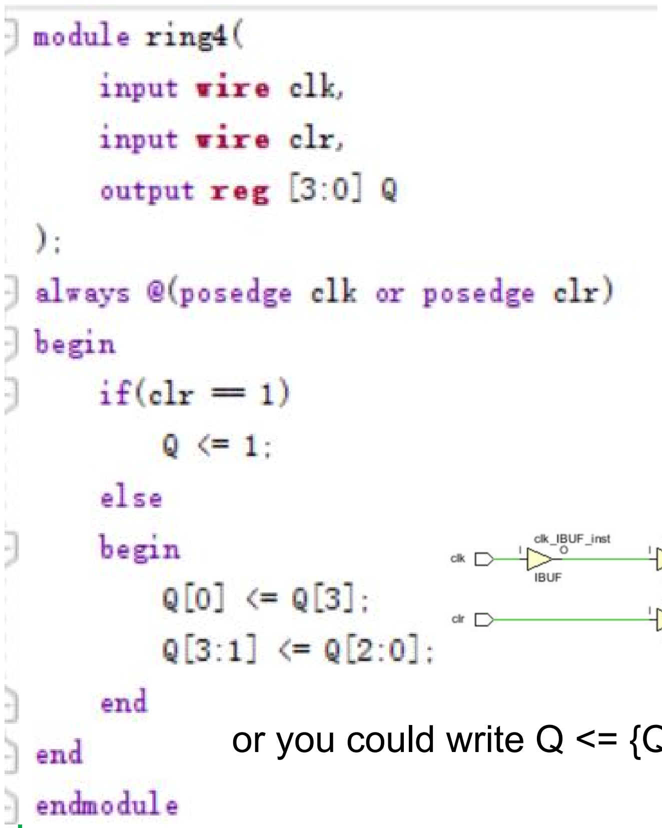


```
module Shift_Left_4(
    input SI,Clk, Rst_n,
    output SO,
    output reg[3:0] D
    );
    assign SO = D[3];
    always @(posedge Clk, negedge Rst_n)
        if (!Rst_n)
            D<=4'b0000;
        else
            D <= {D[2:0], SI};
endmodule
```

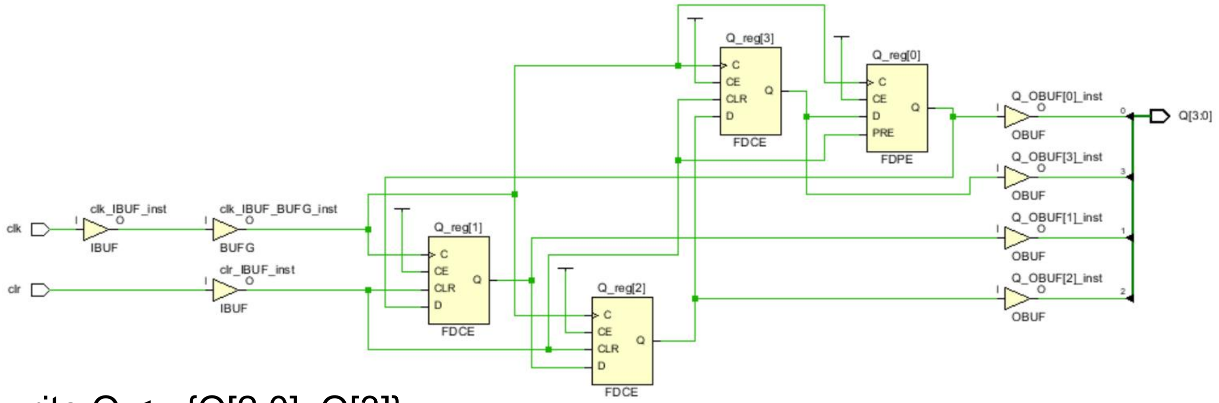


Name	Value
sSI	1
sClk	1
sRst_n	1
sSO	1
sD[3:0]	f
[3]	1
[2]	1
[1]	1
[0]	1

Rotate left



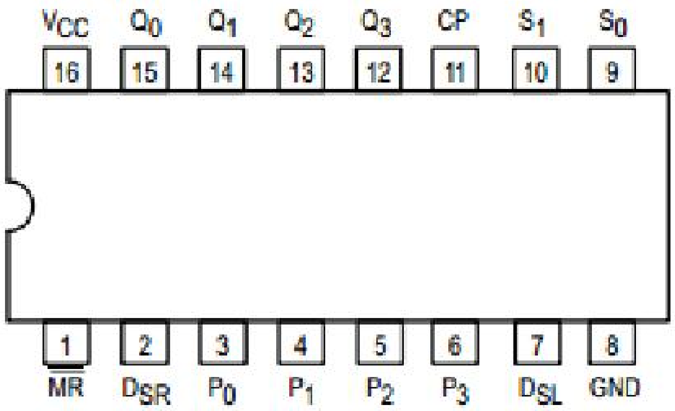
```
module ring4(
    input wire clk,
    input wire clr,
    output reg [3:0] Q
);
always @(posedge clk or posedge clr)
begin
    if(clr == 1)
        Q <= 1;
    else
    begin
        Q[0] <= Q[3];
        Q[3:1] <= Q[2:0];
    end
end
endmodule
```



or you could write Q <= {Q[2:0], Q[3]}

Shift register - 74194(1)

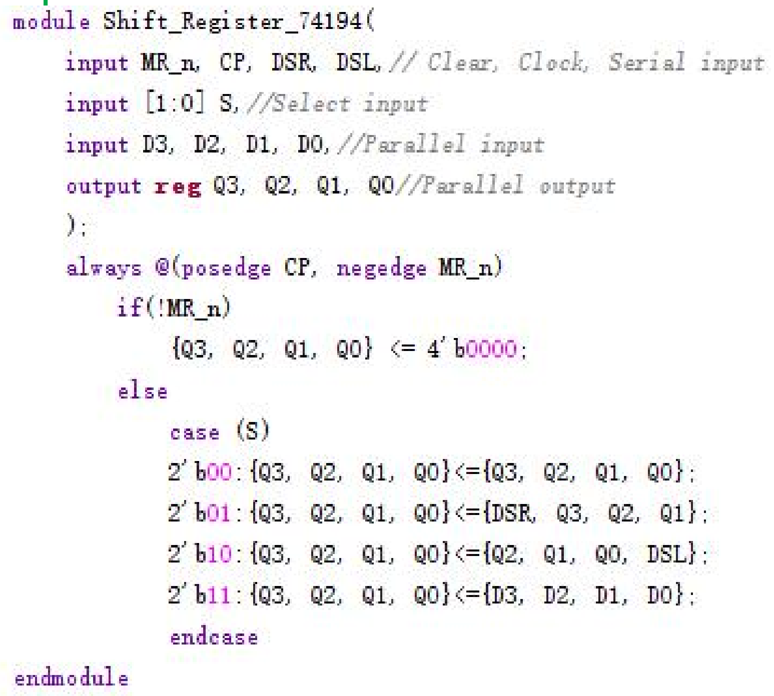
• S_0, S_1	Mode Control inputs
• $P_0 \sim P_3$	Parallel Data Inputs
• D_{SR}	Serial(Shift Right) Data Input
• D_{SL}	Serial(Shift Left) Data Input
• CP	Clock Input
• $\overline{MR}$	Master Reset Input
• $Q_0 \sim Q_3$	Parallel Outputs, Q0 is MSB



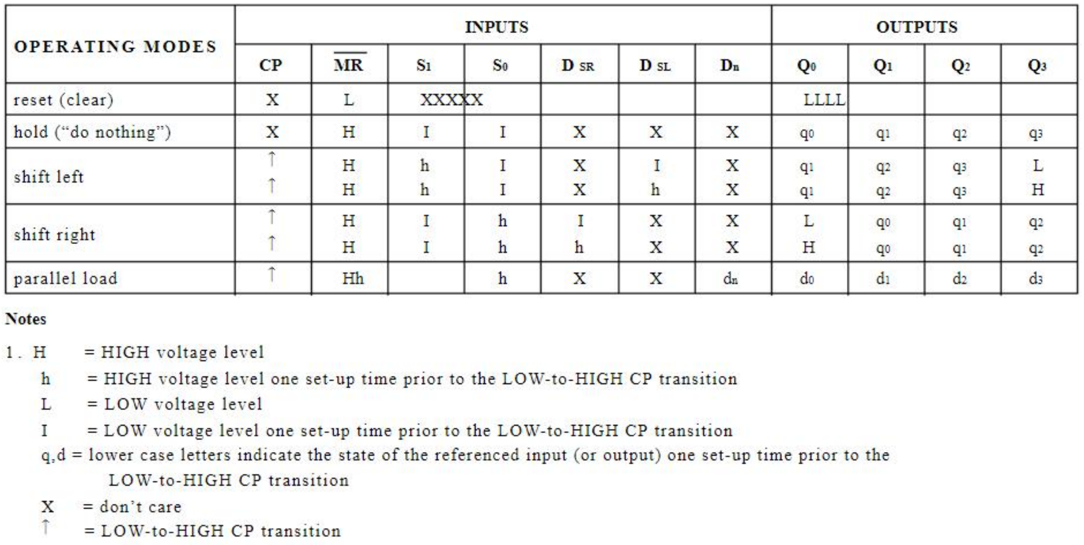
4-BIT BIDIRECTIONAL UNIVERSAL SHIFT REGISTER

The **SN54/74LS194A** is a High Speed 4-Bit Bidirectional Universal Shift Register. As a high speed multifunctional sequential building block, it is useful in a wide variety of applications. It may be used in **serial-serial**, **shift left**, **shift right**, **serial-parallel**, **parallel-serial**, and **parallel-parallel** data register transfers.

Shift register - 74194(2)



```
module Shift_Register_74194(
    input MR_n, CP, DSR, DSL,// Clear, Clock, Serial input
    input [1:0] S,//Select input
    input D3, D2, D1, D0,//Parallel input
    output reg Q3, Q2, Q1, Q0//Parallel output
    );
    always @(posedge CP, negedge MR_n)
        if(!MR_n)
            {Q3, Q2, Q1, Q0} <= 4'b0000;
        else
            case (S)
            2'b00:{Q3, Q2, Q1, Q0}<={Q3, Q2, Q1, Q0};
            2'b01:{Q3, Q2, Q1, Q0}<={DSR, Q3, Q2, Q1};
            2'b10:{Q3, Q2, Q1, Q0}<={Q2, Q1, Q0, DSL};
            2'b11:{Q3, Q2, Q1, Q0}<={D3, D2, D1, D0};
            endcase
endmodule
```



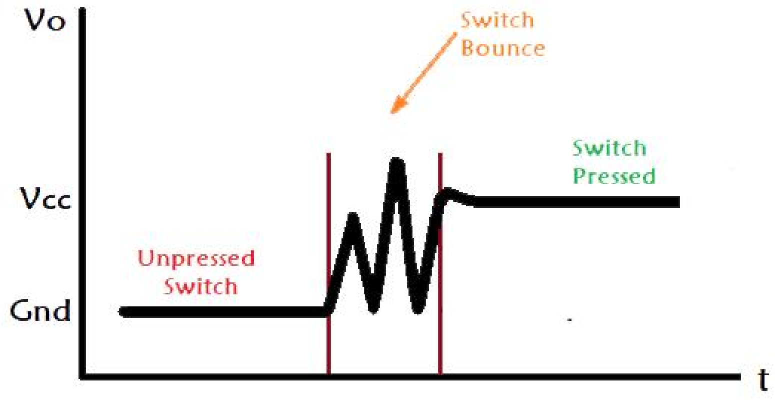
OPERATING MODES	INPUTS							OUTPUTS			
	CP	$\overline{MR}$	S_1	S_0	D_{SR}	D_{SL}	D_n	Q_0	Q_1	Q_2	Q_3
reset (clear)	X	L	XXXXX					LLLL			
hold ("do nothing")	X	H	I	I	X	X	X	q_0	q_1	q_2	q_3
shift left	↑	H	h	I	X	I	X	q_1	q_2	q_3	L
	↑	H	h	I	X	h	X	q_1	q_2	q_3	H
shift right	↑	H	I	h	I	X	X	L	q_0	q_1	q_2
	↑	H	I	h	h	X	X	H	q_0	q_1	q_2
parallel load	↑	Hh		h	X	X	d_n	d_0	d_1	d_2	d_3

Notes

1. H = HIGH voltage level
   h = HIGH voltage level one set-up time prior to the LOW-to-HIGH CP transition
   L = LOW voltage level
   I = LOW voltage level one set-up time prior to the LOW-to-HIGH CP transition
   q,d = lower case letters indicate the state of the referenced input (or output) one set-up time prior to the LOW-to-HIGH CP transition
   X = don't care
   ↑ = LOW-to-HIGH CP transition

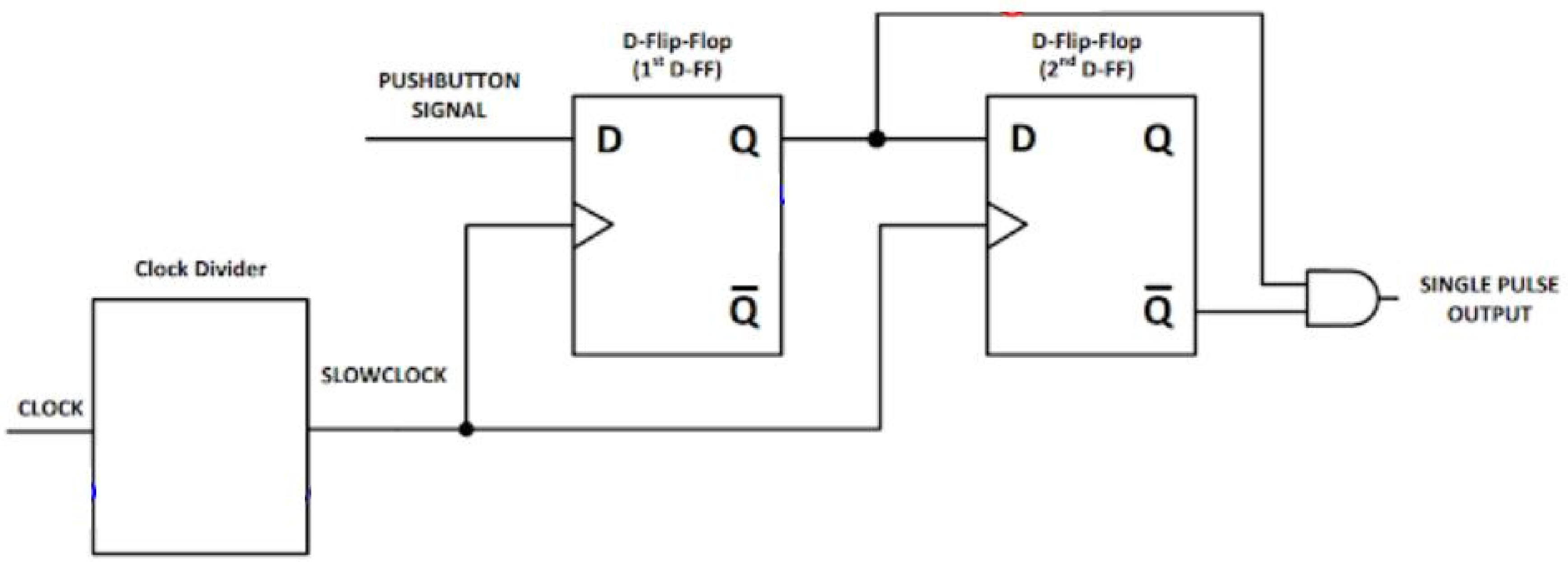
Button Debounce

- Bouncing: When we press a pushbutton or toggle switch or a micro switch, two metal parts come into contact to short the supply. But they don't connect instantly but the metal parts connect and disconnect several times before the actual stable connection is made. The same thing happens while releasing the button.
- Debouncing is removing unwanted input noise from buttons, switches or other user input. Debouncing prevents extra activations or slow functions from triggering too often.



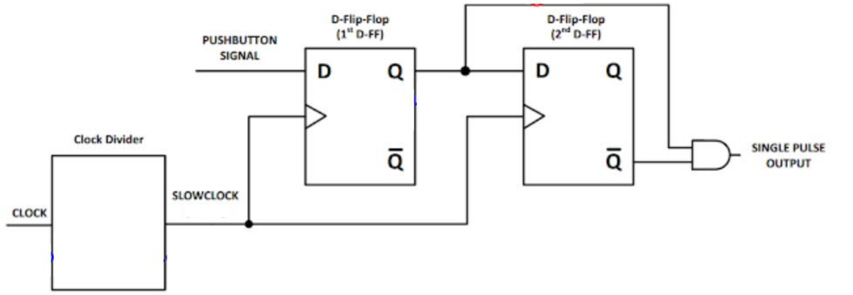
Debouncer with shift register

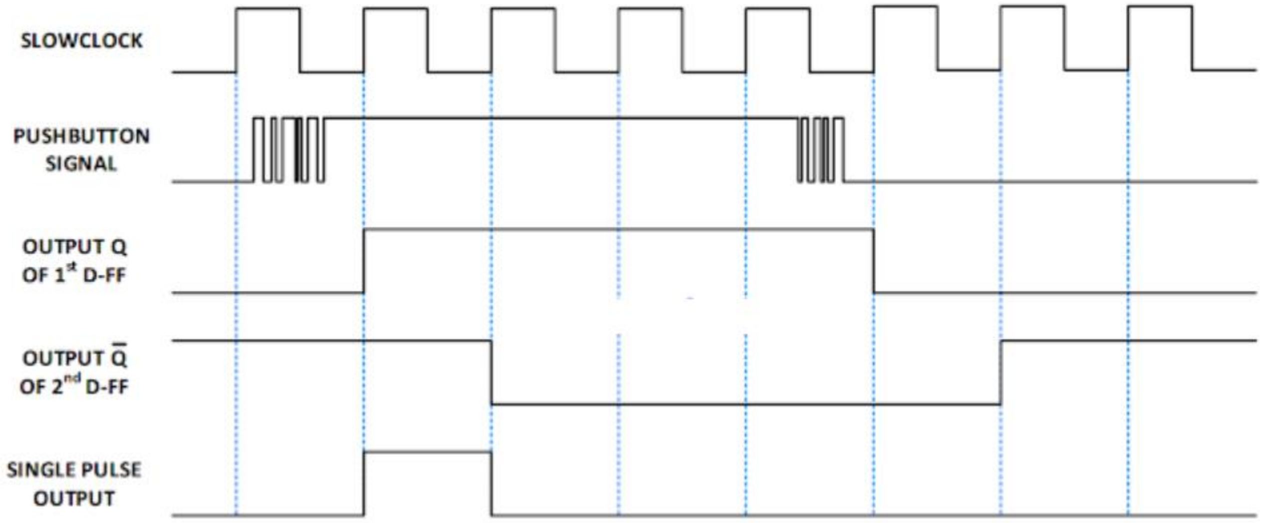
- The debouncing circuit only generates a single pulse with a period of the slow clock without bouncing as we expected.



Debouncer with shift register

- Need to use a slow clock (e.g. 100Hz) to correctly capture the trigger.
- single pulse output = Q1 & Q2'





Edge detection with shift register(1)

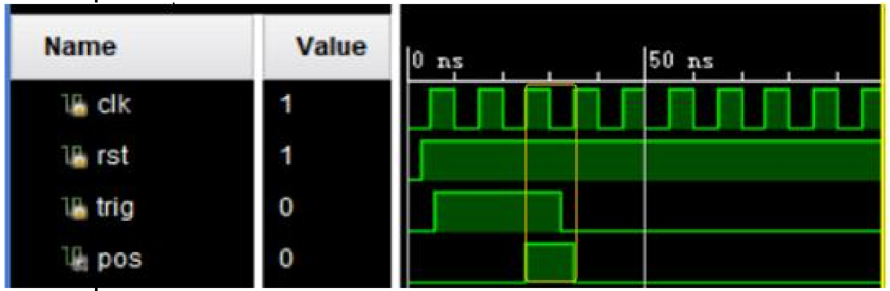
DON'T using posege or negedge of a normal signal except 'clock' or 'reset' in the sensitive list of the 'always' statement block .

DON'Ts:

```
always @(posedge trig) begin
        x < = y;
end
//here signal 'trig' is not used as clock or reset
```

Do Using following description instead

```
always @(posedge clk) begin
        if(pos)
           x <= y;
        else
           x <= x;
end
```



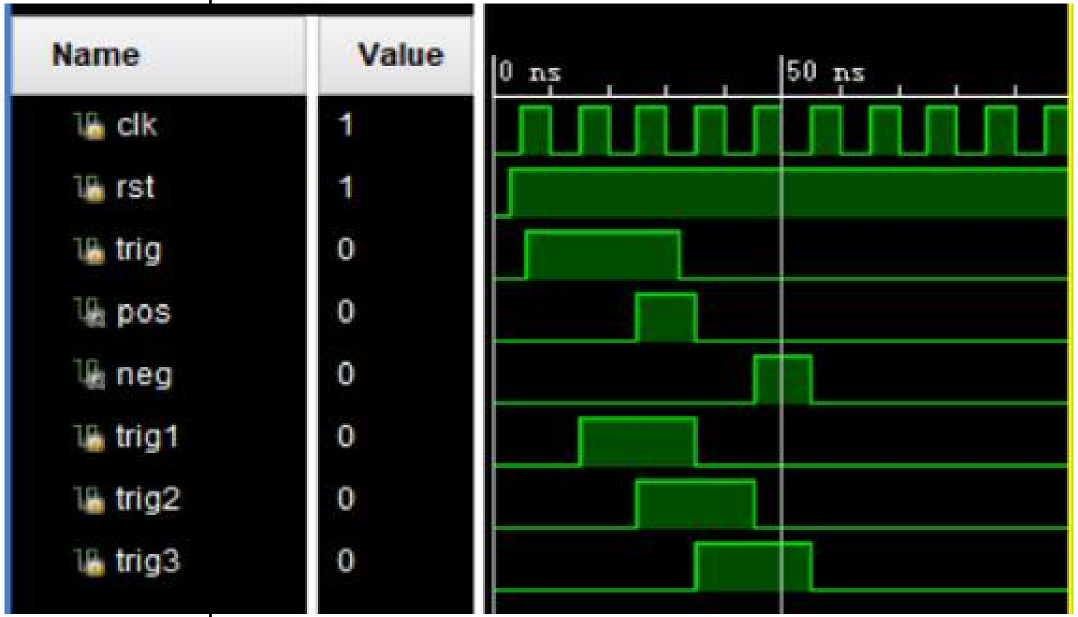
Name	Value
clk	1
rst	1
trig	0
pos	0

Here 'pos' is a signal synchronized with the clock 'clk'. When a rising edge event occurs in the 'trig' signal, the 'pos' signal is 1 and lasts for one clock cycle.

Edge detection with shift register(2)

```
module trigTest(input clk,rst,trig, output pos,neg);
reg trig1,trig2,trig3;
always @(posedge clk, negedge rst)
    if(!rst)
        {trig1,trig2,trig3} <= 3'b000;
    else begin
        trig1 <= trig;
        trig2 <= trig1;
        trig3 <= trig2;
    end
assign pos = (~trig3) & trig2;
//To be completed
endmodule
```

- Here is a demo about how to generate a “pos” signal.
- Complete the code to generate a “neg” signal.

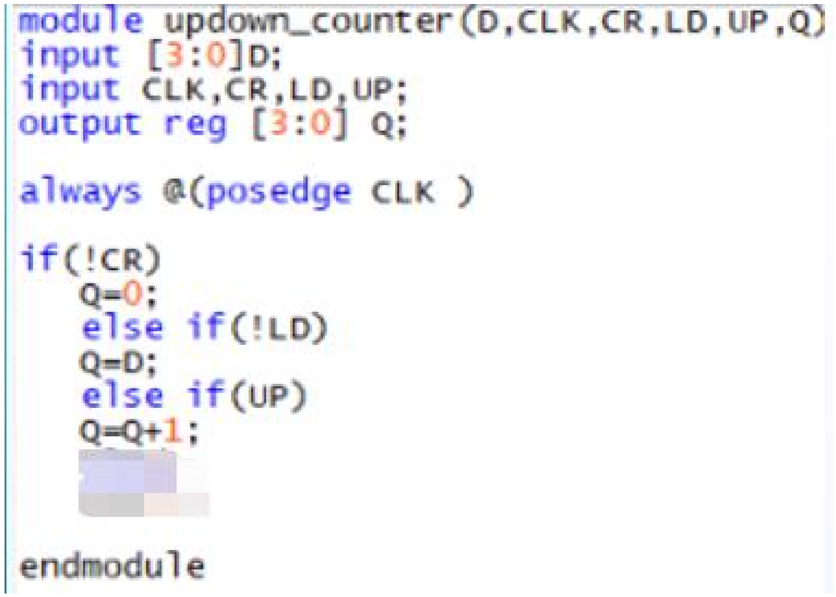


DON'Ts in Design(1)

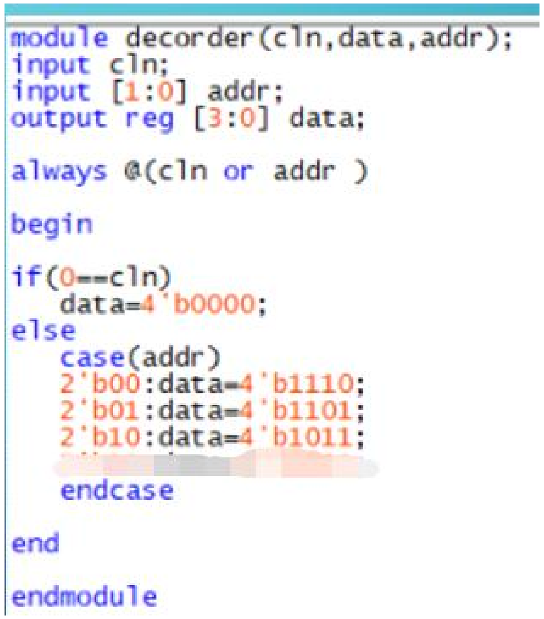
- **Non-Synthesizable Verilog** which **is NOT suggested** in design.
  - initial
  - Task, function
  - System task:$display, $monitor, $strobe, $finish
  - fork... join
  - **U**ser**D**efined**P**rimitive

DON’Ts in Design(2)

- **Incomplete “if else” block or Incomplete “case” in combinational logical block** which are **NOT suggested** in your design.
  - a unexpected latch would be generated by EDA tool while finish the systhesis, the latch is not good for the combinational logic.



```
module updown_counter(D,CLK,CR,LD,UP,Q)
input [3:0]D;
input CLK,CR,LD,UP;
output reg [3:0] Q;

always @(posedge CLK )

if(!CR)
    Q=0;
    else if(!LD)
    Q=D;
    else if(UP)
    Q=Q+1;



endmodule
```

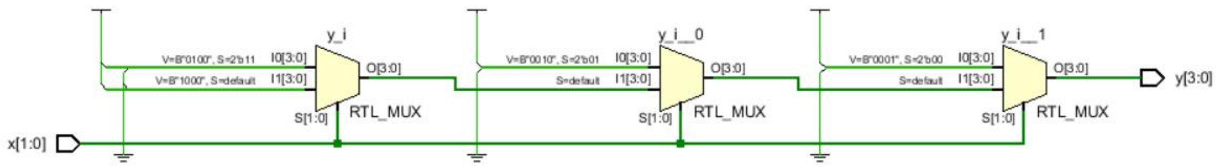


```
module decorder(cln,data,addr);
input cln;
input [1:0] addr;
output reg [3:0] data;

always @(cln or addr )

begin

if(0==cln)
    data=4'b0000;
else
    case(addr)
    2'b00:data=4'b1110;
    2'b01:data=4'b1101;
    2'b10:data=4'b1011;

    endcase

end

endmodule
```

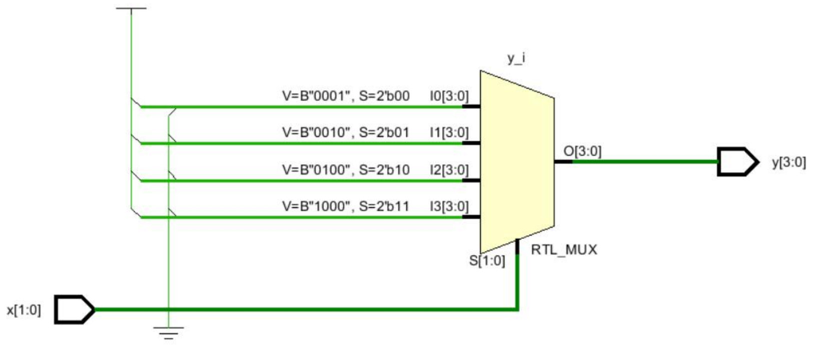
DON’Ts in Design(3)

- **NOT suggested**
  - Embedded ‘if-else’
- **Suggested**
  - Using ‘case’ instead of embedded ‘if-else’ to avoid unwanted priority and longer delay

```
always @*
    if( 2'b00 == x)
        y = 4'b0001;
    else if( 2'b01 == x)
            y = 4'b0010;
        else if( 2'b11 == x)
                y = 4'b0100;
            else
                y = 4'b1000;
```

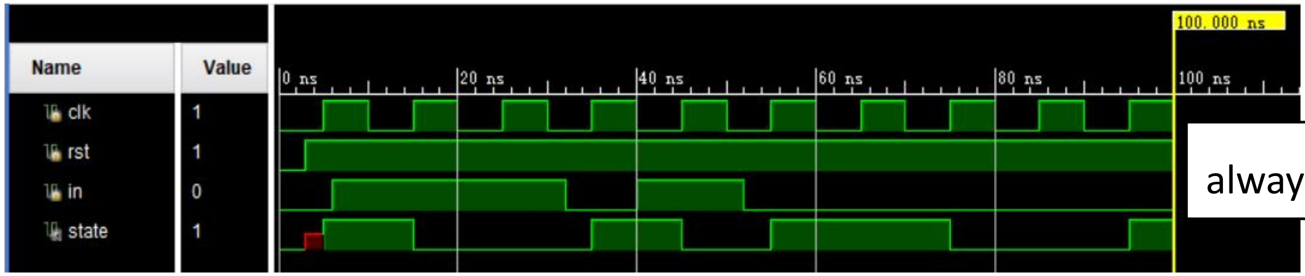


```
always @*
    case(x)
        2'b00: y=4'b0001;
        2'b01: y=4'b0010;
        2'b10: y=4'b0100;
        2'b11: y=4'b1000;
    endcase
```

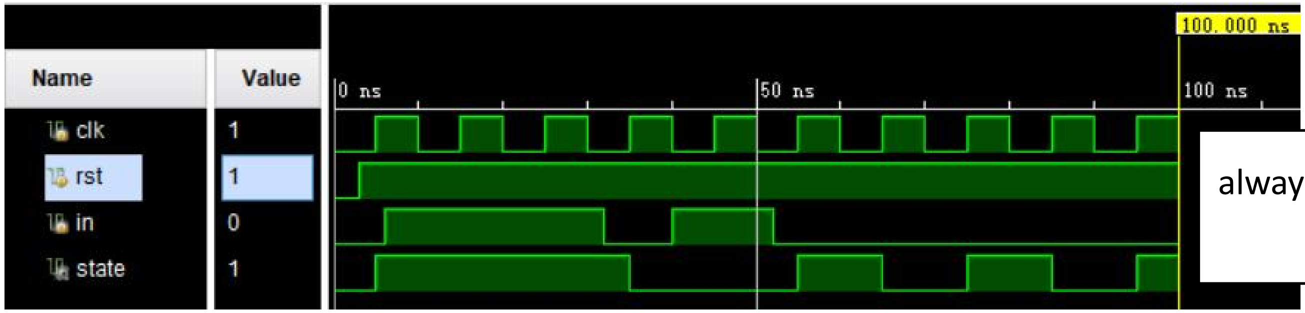


DON’Ts in Design(4)

- **NOT suggested**
  - mixed sensitive list. (e.g. always@(posedge clk, rst, in))
- **Suggested**
  - Using two stage(sequential logic + combinational logic) to replace the one stage which using mixed sensitive list



always @(posedge clk, rst, in)



always @(posedge clk, negedge rst)

always @(*)

Practice 1&2

- 1. Implement a button denouncer and toggle the state of LED after pressing each time the button.

- 2. Plan the register and memory which would be used in your project? Which modules needs register or memory? How to define, read and write them? (Parameters are suggested to be used here)

Practice 3

- Rock-Paper-Scissors game (3nd version)
  - You should modify the design implemented in Lab6 to a sequential circuit. This time, the game result can not be visualized until the evaluation button has been pressed.
  - Designs inputs are: clock(P17), reset_n(P15), button(R11), switches; outputs are: 7-segments(left most and right most), LEDs. You need to debounce the button
  - You need to design a FSM with 2 states: play, and evaluate. When reset_n(system reset) is pressed, the system enters to "play" state, when button is pressed, the system enters to "evaluate" state and the user's game results are displayed on LEDs and 7-segment tubes.
- 仿照视频中曾展示的"分歧终端机"，修改Lab6的猜拳游戏为时序逻辑：只有按过evaluation button之后，才能在LED和7段数码管上显示出两个玩家的猜拳结果。
  - 电路输入为：时钟(P17)，系统reset(P15)，evaluation button(R11)，输出为LED和数码管，按键需消抖
  - 你需要实现包含有play和evaluate两个state的FSM，在按下系统reset按键时，系统进入play状态，两个玩家使用switch输入猜拳手势，但结果不会立刻显示，直到按过evaluation button后，才在LED和数码管显示猜拳手势和结果
- You can use your own code from Lab6, in case you have lost your code, you can use the one provided in lab13
- Use P15 for reset_n (system reset, active_low)
- Use R11 for evaluation button (active high, need debouncing)