



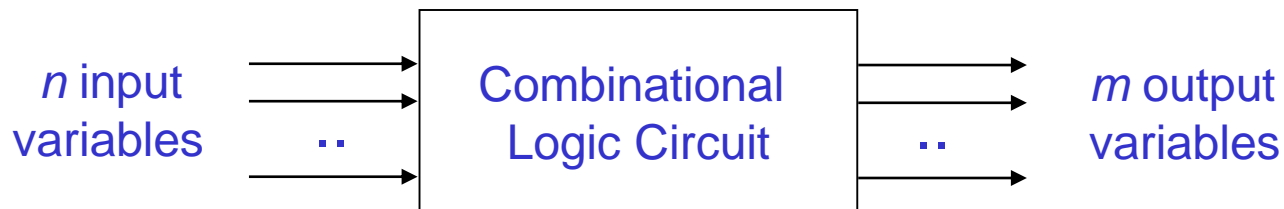
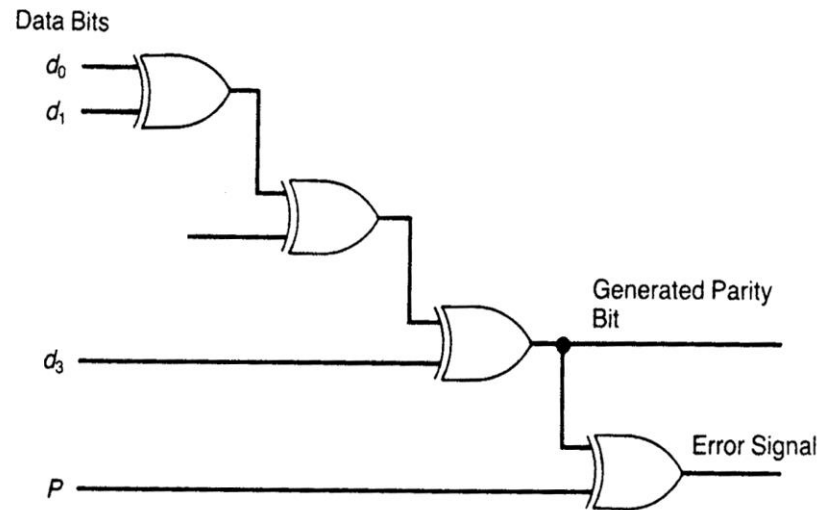
# 數位**IC**設計

---

*Digital System Design*

# Combinational Circuit

A combinational circuit consists of logic gates whose outputs at any time are determined directly from the present combination of inputs without regard to previous inputs.



# Example – Alarm (1/2)

Assume that four persons might come. Alarm is activated when (1) more than three persons come or (2) the fourth person come together with other persons

```
module four(A , B , C , D , Out);  
input A , B , C , D;  
output Out;  
reg Out , temp;  
always @(A or B or C or D)  
begin  
    case({A , B , C , D})  
        4'b0000: Out = 0;  
        4'b0001: Out = 0;  
        4'b0010: Out = 0;  
        4'b0011: Out = 1;  
        4'b0100: Out = 0;  
        4'b0101: Out = 1;  
        4'b0110: Out = 0;  
        4'b0111: Out = 1;  
        4'b1000: Out = 0;  
        4'b1001: Out = 1;  
        4'b1010: Out = 0;  
        4'b1011: Out = 1;  
        4'b1100: Out = 0;  
        4'b1101: Out = 1;  
        4'b1110: Out = 1;  
        default: Out = 1;  
    endcase  
end  
endmodule
```

A	B	C	D	Out
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

*Optimization is done by tools*

# Example – Alarm (2/2)

```
module four(A , B , C , D);  
input A , B , C , D;  
output Out;  
  
wire t1 , t2 , t3 , t4;  
  
and a1(t1 , A , D);  
and a2(t2 , B , D);  
and a3(t3 , C , D);  
and a4(t4 , A , B , C);  
or o1(Out , t1 , t2 , t3 , t4);  
endmodule
```

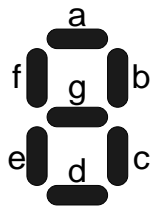
CD \ AB	00	01	11	10
00	0	0	1	0
01	0	1	1	0
11	0	1	1	1
10	0	1	1	0

$$\text{Out} = AD + BD + CD + ABC$$

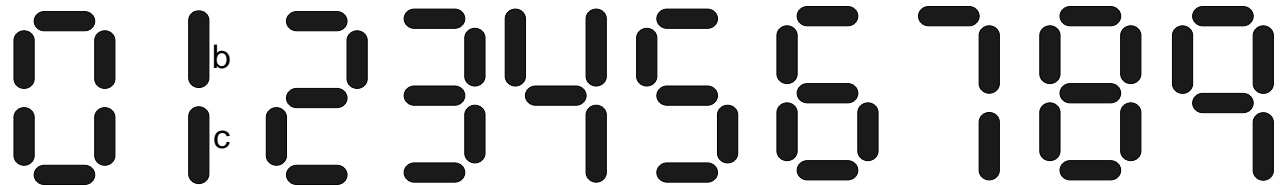
Traditional design method  
(optimization is done by hand)  
→ not suitable for HDL design

# Example - Seven Segment Display

A BCD (Binary-Coded Decimal)-to-seven-segment decoder is a combinational circuit that accepts a decimal digit in BCD and generates the appropriate output for selection of segments in a display indicator used for displaying the decimal digit. The seven outputs of the decoder (a, b, c, d, e, f, g) select the corresponding segments in the display as shown in Fig. (a). The numeric designation chosen to represent the decimal digit is shown in Fig. (b). Design the BCD-to-seven-segment decoder circuit.



(a) Segment designation



(b) Numerical designation for display

A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
.	.	.		.	.	.				

# Example – Multiplexer (1/2)

Multiplexer = selector

2 to 1 selector

```
module mux2to1a(a, b, Select, Out);  
input    a, b, Select;  
output   Out;  
reg      Out;
```

Select	Out
1	a
0	b
X	X

always @(a or b or Select)

begin

if (Select)

Out = a;

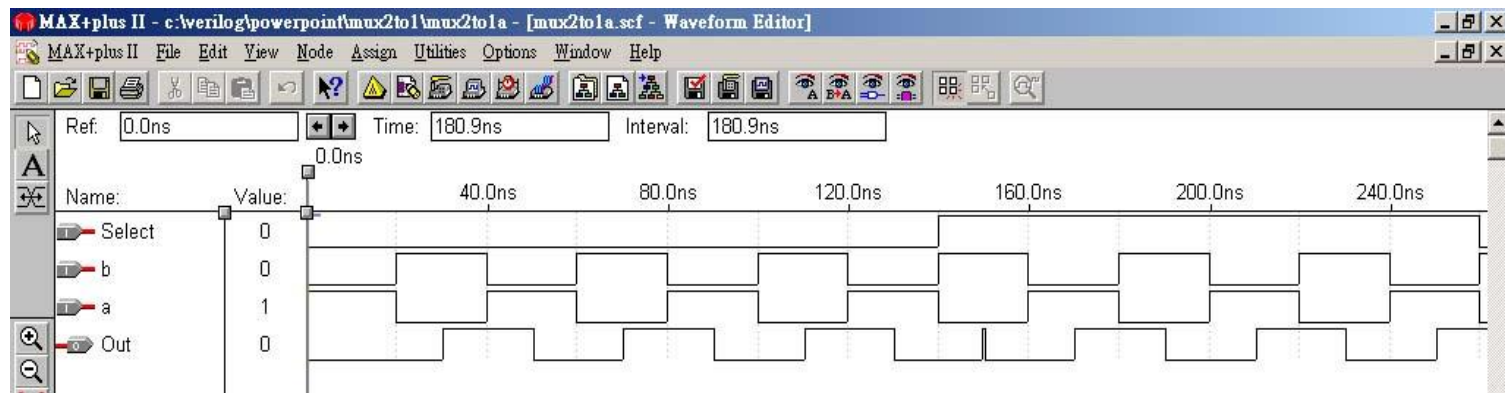
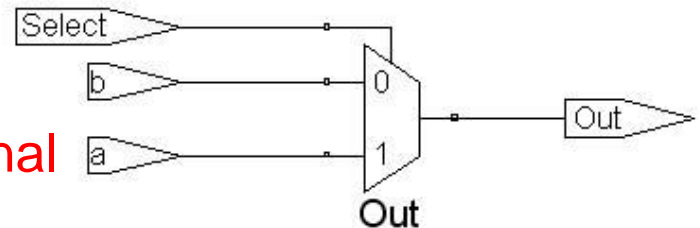
else

Out = b;

end

endmodule

Put all inputs of the combinational circuit into the sensitivity list, otherwise .. error



# Example – Multiplexer (2/2)

There are three ways to derive a 2-to-1 multiplexer

```
always @(a or b or Select)
begin
  if (Select)
    Out = a;
  else
    Out = b;
end
```

Method\_1

```
always @(a or b or Select)
begin
  Out=b;
  if (Select)
    Out = a;
end
```

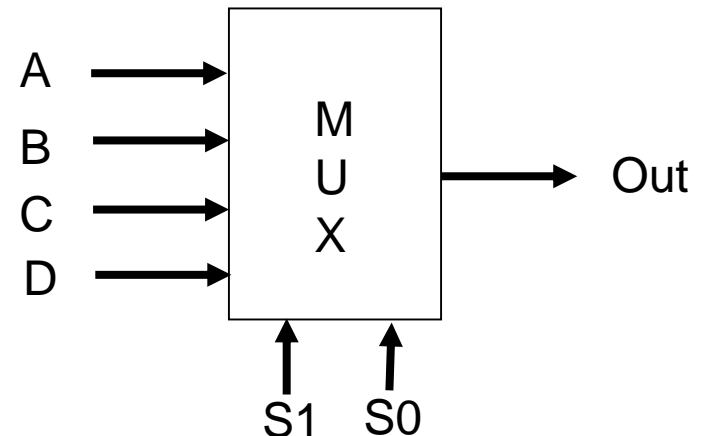
Method\_2

Method\_3

```
assign Out = Select ? a : b;
```

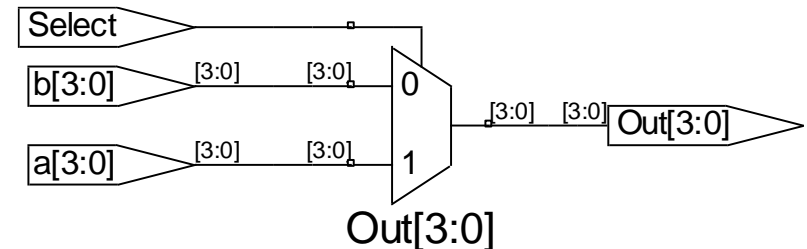
```
always @(A or B or C or D or S1 or S0)
begin
  case ({S1, S0})
    2'b00: Out = A;
    2'b01: Out = B;
    2'b10: Out = C;
    default: Out = D;
  endcase
end
```

4-to-1  
multiplexer



# Example – Multi-Bit Multiplexer

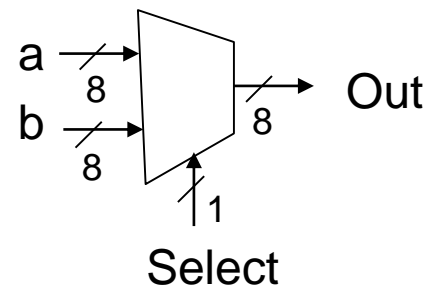
```
module multibit2(a, b, Select, Out);  
input  [3:0] a, b;  
input  Select;  
output [3:0] Out;  
wire   [3:0] Out;
```



```
    assign Out = Select ? a : b;  
endmodule
```

Actually four 1-bit 2-to-1 multiplexers are used here

```
module multibit2(a, b, Select, Out);  
parameter width=8;  
input  [width-1:0] a, b;  
input  Select;  
output [width-1:0] Out;  
wire   [width-1:0] Out;
```

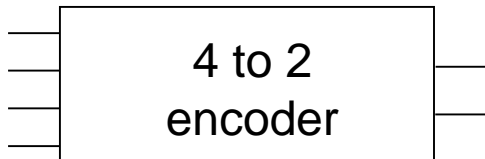


```
    assign Out = Select ? a : b;  
endmodule
```

8-bit 2-to-1 multiplexer



# Example – Encoder (4 to 2)



```
module    encoder(A,Y);  
parameter size=4;  
input     [size-1:0] A;  
output    [1:0] Y;  
reg       [1:0] Y;
```

```
always@(A)  
begin  
    case(A)  
        4'b 0001 : Y=0;  
        4'b 0010 : Y=1;  
        4'b 0100 : Y=2;  
        4'b 1000 : Y=3;  
        default: Y=2'b00;  
    endcase  
end  
endmodule
```

inputs				outputs	
A[3]	A[2]	A[1]	A[0]	Y[1]	Y[0]
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

```

module      encoder (A,Valid,Y);
input       [3:0] A;
output      Valid;
output      [1:0] Y;
reg         Valid;
reg         [1:0] Y;

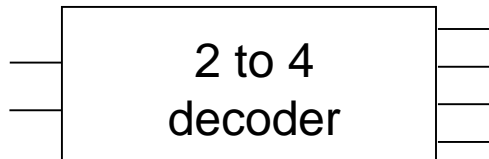
always@(A)
begin
    Valid=1;
    casex(A)
        4'b 1xxx : Y=3;
        4'b 01xx : Y=2;
        4'b 001x : Y=1;
        4'b 0001 : Y=0;
        default:
            begin Valid=0; Y=2'b00; end
    endcase
end
endmodule

```

inputs				outputs		
A[3]	A[2]	A[1]	A[0]	Y[1]	Y[0]	Valid
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

x : don't care

# Example – Decoder (2 to 4)



```
module    decoder(A,Y);  
parameter size=4;  
input     [1:0] A;  
output    [size-1:0] Y;  
reg       [size-1:0] Y;
```

```
always@(A)  
begin  
    case(A)  
        0:Y = 4'b0001;  
        1:Y = 4'b0010;  
        2:Y = 4'b0100;  
        default:Y = 4'b1000;  
    endcase  
end  
endmodule
```

inputs		outputs			
A[1]	A[0]	Y[3]	Y[2]	Y[1]	Y[0]
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

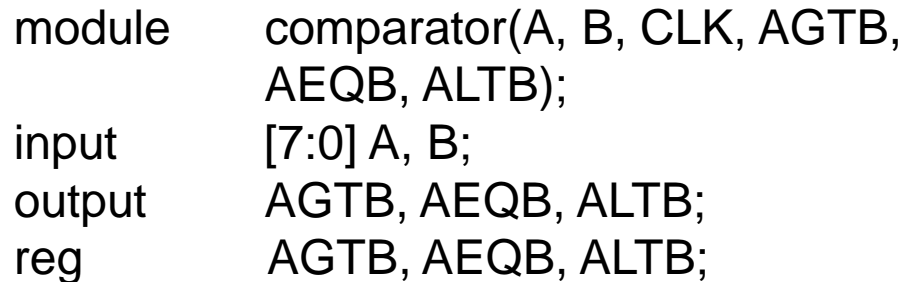
# Example – Decoder (3 to 6)

input				outputs					
En	A2	A1	A0	Y5	Y4	Y3	Y2	Y1	Y0
0	X	X	X	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	1	0
1	0	1	0	0	0	0	1	0	0
1	0	1	1	0	0	1	0	0	0
1	1	0	0	0	1	0	0	0	0
1	1	0	1	1	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0

```
module decoder3_6_case1(A,En,Y);  
input En;   input [2:0] A;  
output [5:0] Y; reg [5:0] Y;
```

```
always@(En or A)  
begin  
    if(!En)  
        Y = 6'b0;  
    else  
        case(A)  
            0:Y = 6'b000001;  
            1:Y = 6'b000010;  
            2:Y = 6'b000100;  
            3:Y = 6'b001000;  
            4:Y = 6'b010000;  
            5:Y = 6'b100000;  
            default:Y = 6'b0;  
        endcase  
    end  
end  
endmodule
```

## Combinational comparator



MAXplus II - c:\documents and settings\labellmy documents\class\max\comparator - [comparator.scf - Waveform Editor]

MAXplus II File Edit View Node Assign Utilities Options Window Help

Ref: 0.0ns Time: 0.0ns Interval: 0.0ns

Name: Value:

0.0ns

80.0ns 160.0ns 240.0ns 320.0ns 400.0ns 480.0ns

A - 00000000 01101111 11011110 01001101 10111100 00101011 10011010 00001001 01111000 11100111 01010110 11000101 -

B - 11111111 10101000 01010001 11111010 10100011 00101011 11110101 10011110 01000111 11110000 10011001 01000010 -

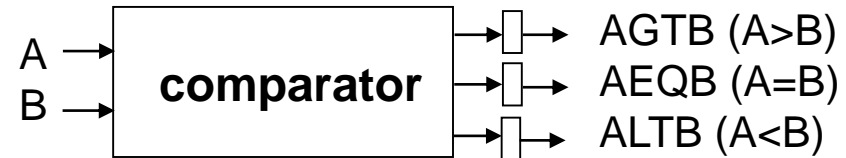
AGTB 0

AEQB 0

ALTB 1

# Comparator (2/4)

Sequential comparator

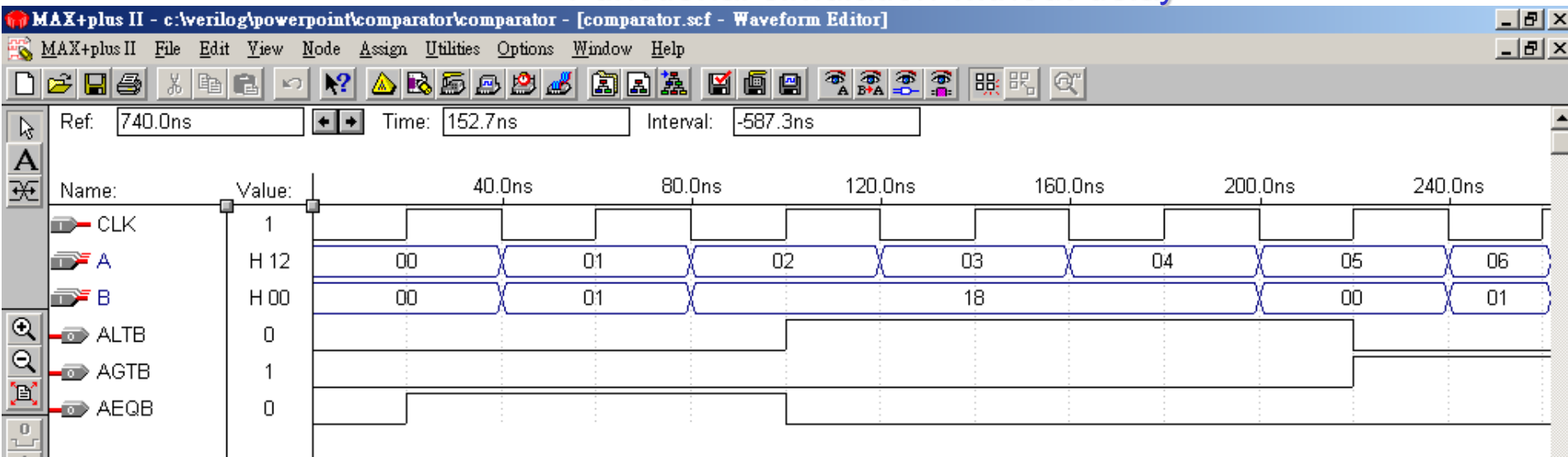


```

module    comparator(A, B, CLK, AGTB,
                    AEQB, ALTB);
parameter size = 8;
input     [size-1:0] A, B;
input     CLK;
output    AGTB, AEQB, ALTB;
reg       AGTB, AEQB, ALTB;

    always @(posedge CLK)
    begin
        AGTB = (A > B);
        AEQB = (A == B);
        ALTB = (A < B);
    end
endmodule
  
```

Functional simulation without delay



# Comparator (3/4)

Decide the biggest value among A, B, C, and D.

Method\_1:

always @(A or B or C or D)

begin

if(A >= B)

Out1=A;

else

Out1=B;

if(Out1 >= C)

Out2=Out1;

else

Out2=C;

if(Out2 >= D)

Out=Out2;

else

Out=D;

end

Method\_2: (hierarchical tree structure)

always @(A or B or C or D)

begin

if(A >= B)

Out1=A;

else

Out1=B;

if(C >= D)

Out2=C;

else

Out2=D;

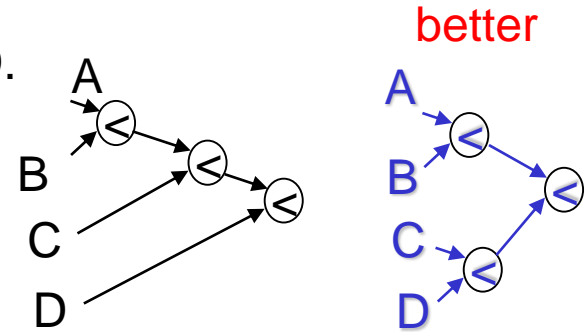
if(Out1 >= Out2)

Out=Out1;

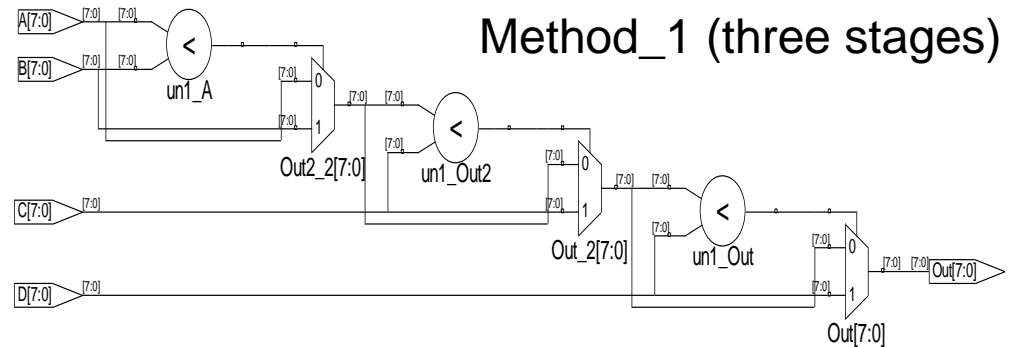
else

Out=Out2;

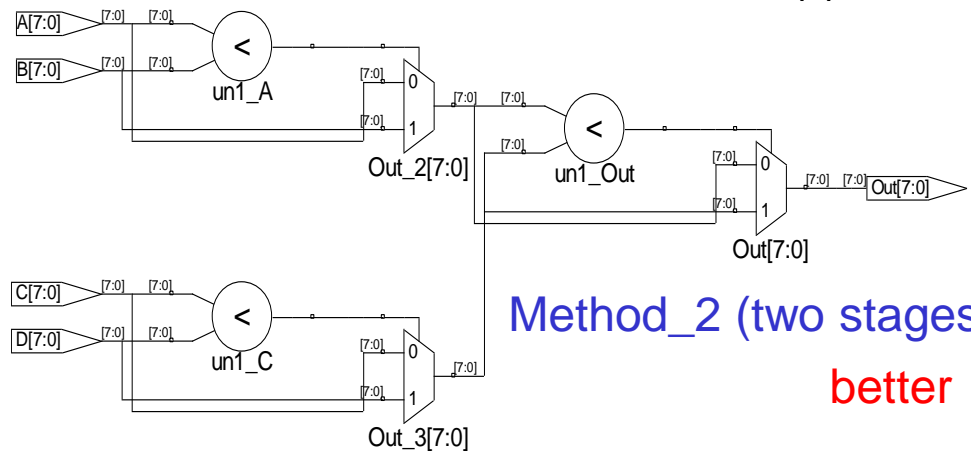
end



better



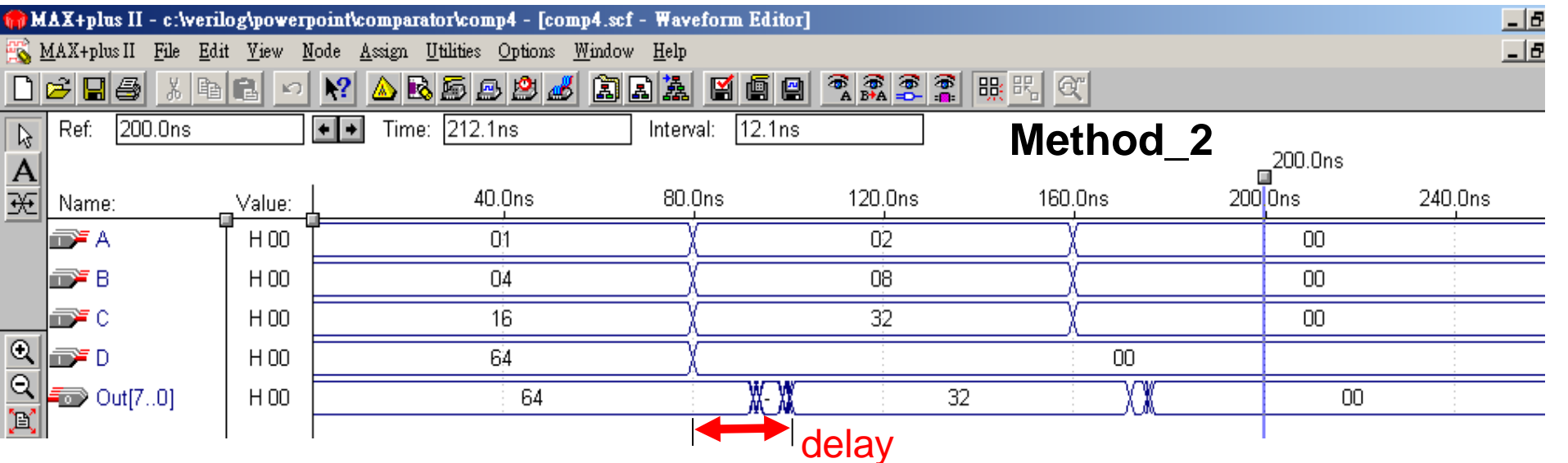
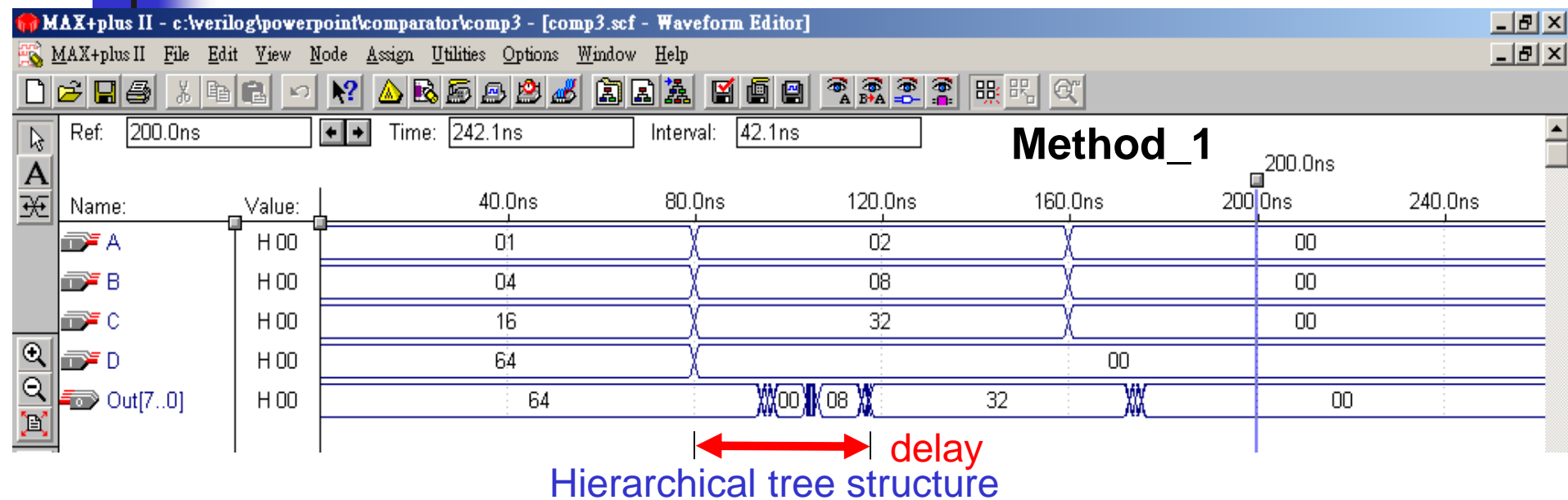
Method\_1 (three stages)



Method\_2 (two stages)

better

# Comparator (4/4)







# Arithmetic Logic Unit (1/2)

S4 S3 S2 S1 S0 Cin	Operation	Function	Implementation
0 0 0 0 0 0	$Y \leq A$	Transfer A	Arithmetic Unit
0 0 0 0 0 1	$Y \leq A + 1$	Increment A	Arithmetic Unit
0 0 0 0 1 0	$Y \leq A + B$	Addition	Arithmetic Unit
0 0 0 0 1 1	$Y \leq A + B + 1$	Add with carry	Arithmetic Unit
0 0 0 1 0 0	$Y \leq A + B_{\text{bar}}$	A plus 1's complement of B	Arithmetic Unit
0 0 0 1 0 1	$Y \leq A + B_{\text{bar}} + 1$	Subtraction	Arithmetic Unit
0 0 0 1 1 0	$Y \leq A - 1$	Decrement A	Arithmetic Unit
0 0 0 1 1 1	$Y \leq A$	Transfer A	Arithmetic Unit
0 0 1 0 0 0	$Y \leq A \text{ and } B$	AND	Logic Unit
0 0 1 0 1 0	$Y \leq A \text{ or } B$	OR	Logic Unit
0 0 1 1 0 0	$Y \leq A \text{ xor } B$	XOR	Logic Unit
0 0 1 1 1 0	$Y \leq A_{\text{bar}}$	Complement A	Logic Unit
0 0 0 0 0 0	$Y \leq A$	Transfer A	Shifter Unit
0 1 0 0 0 0	$Y \leq \text{shl } A$	Shift left A	Shifter Unit
1 0 0 0 0 0	$Y \leq \text{shr } A$	Shift right A	Shifter Unit
1 1 0 0 0 0	$Y \leq 0$	Transfer 0's	Shifter Unit



# Arithmetic Logic Unit (2/2)

---

```
module alu_case2(Sel,CarryIn,A,B,Y);  
input [4:0] Sel;  
input CarryIn;  
input [7:0] A,B;  
output [7:0] Y;  
reg [7:0] Y;
```

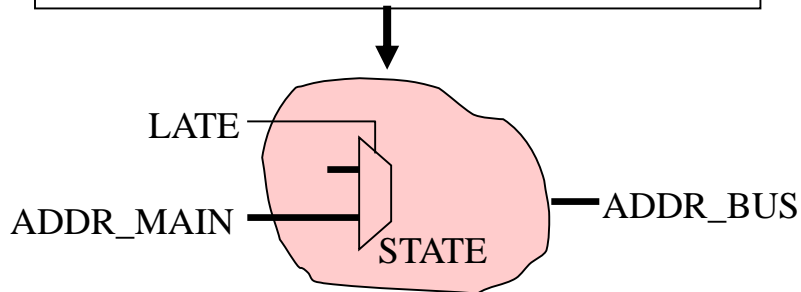
```
always@(Sel or A or B or CarryIn)  
begin  
    case({Sel[4:0],CarryIn})  
        6'b000000 : Y = A;  
        6'b000001 : Y = A + 1;  
        6'b000010 : Y = A + B;  
        6'b000011 : Y = A + B + 1;  
        6'b000100 : Y = A + !B;  
        6'b000101 : Y = A + !B + 1;  
        6'b000110 : Y = A - 1;  
        6'b000111 : Y = A;  
        6'b001000 : Y = A & B;  
        6'b001010 : Y = A | B;  
        6'b001100 : Y = A ^ B;  
        6'b001110 : Y = !A;  
        6'b010000 : Y = A << 1;  
        6'b100000 : Y = A >> 1;  
        6'b110000 : Y = 0;  
        default: Y = 8'bX;  
    endcase  
end  
endmodule
```

# Example for IF (1/4)

- Good style takes advantage of if-else priority to synthesize correct logic

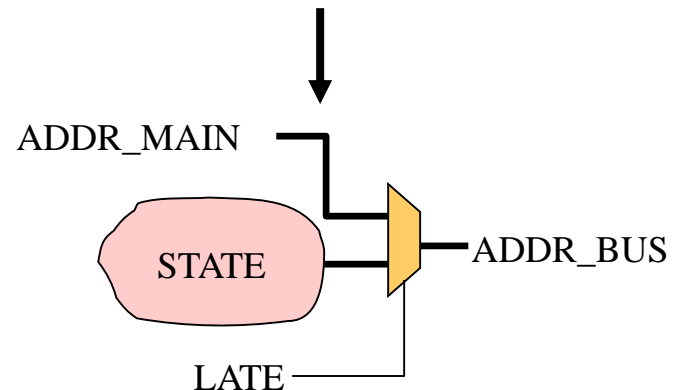
Bad

```
case (STATE)
  IDLE:
    if (LATE == 1'b1)
      ADDR_BUS <= ADDR_MAIN;
    else
      ADDR_BUS <= ADDR_CNTL;
  INTERRUPT:
    if (LATE == 1'b1)
      ADDR_BUS <= ADDR_MAIN;
    else
      ADDR_BUS <= ADDR_INT;
  ....
```



Good

```
if (LATE == 1'b1)
  ADDR_BUS <= ADDR_MAIN;
else
  case (STATE)
    IDLE:
      ADDR_BUS <= ADDR_CNTL;
    INTERRUPT:
      ADDR_BUS <= ADDR_INT;
  ....
```





# Example for IF (2/4)

```
module style_bad(In_Data, State,  
Out_Data, En);  
input En;  
input [1:0] State;  
input [2:0] In_Data;  
output [3:0] Out_Data;  
reg [3:0] Out_Data;  
parameter A1=0, A2=1, A3=2, A4=3;
```

```
always @(In_Data or State or En)  
begin  
    case(State)  
        A1:  
            begin  
                if(En)  
                    Out_Data = In_Data;  
                else  
                    Out_Data = In_Data - 1;  
            end  
    endcase  
end
```

**Bad Style**

```
A2:  
    begin  
        if(En)  
            Out_Data = In_Data;  
        else  
            Out_Data = In_Data + 1;  
        end  
A3:  
    begin  
        if(En)  
            Out_Data = In_Data;  
        else  
            Out_Data = In_Data - 2;  
        end  
A4:  
    begin  
        if(En)  
            Out_Data = In_Data;  
        else  
            Out_Data = In_Data + 2;  
        end  
    end endcase end endmodule
```



# Example for IF (3/4)

---

```
module style_good(In_Data, State, Out_Data, En);  
input  En;  
input  [1:0] State;  
input  [2:0] In_Data;  
output [3:0] Out_Data;  
reg     [3:0] Out_Data;
```

```
parameter A1=0, A2=1, A3=2, A4=3;
```

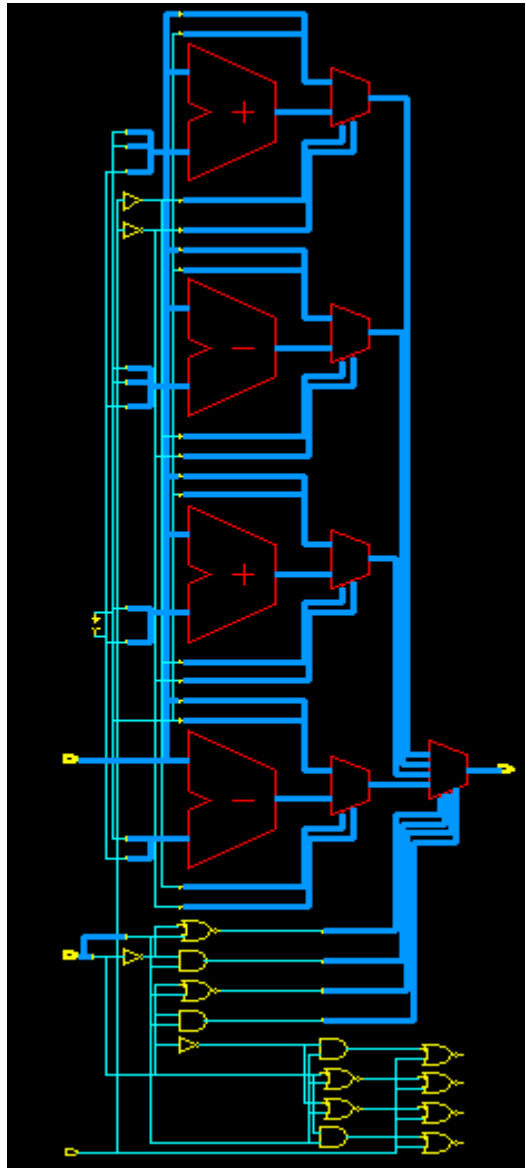
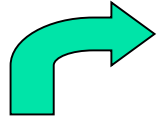
```
always @(In_Data or State or En)  
begin  
    if(En)  
        Out_Data = In_Data;  
    else  
        begin
```

**Good Style**

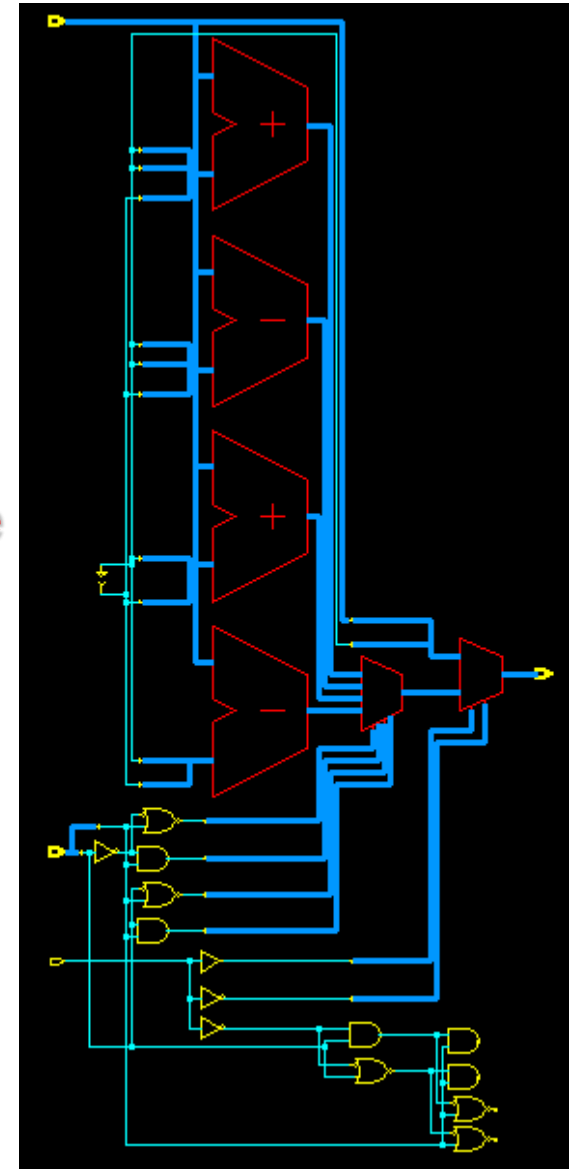
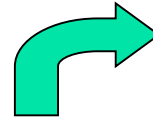
```
        case(State)  
            A1: Out_Data = In_Data - 1;  
            A2: Out_Data = In_Data + 1;  
            A3: Out_Data = In_Data - 2;  
            A4: Out_Data = In_Data + 2;  
        endcase  
    end  
end  
endmodule
```

# Example for IF (4/4)

Bad Style

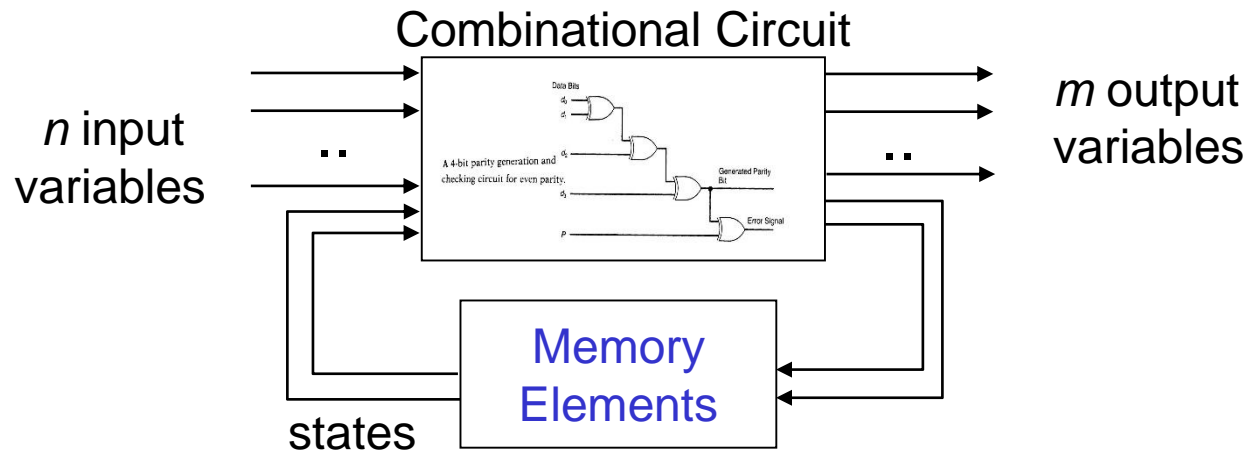


Good Style



# Sequential Circuit (1/2)

A sequential circuit is a system whose outputs at any time are determined from the present combination of inputs and the previous inputs or outputs.



- Sequential components contain memory elements
- The output values of sequential components depend on the input values and the values stored in the memory elements
- Example: Ring counter that starts the answering machine after 4 rings



# Sequential Circuit (2/2)

---

**Sequential components can be: asynchronous or synchronous**

## **Asynchronous sequential circuit:**

Change their states and outputs whenever a change in inputs occurs

## **Synchronous sequential circuit:**

Change their states and outputs at fixed points of time (specified by clock signal)

**Most circuits are synchronous circuits (easy and tool-supportable).**

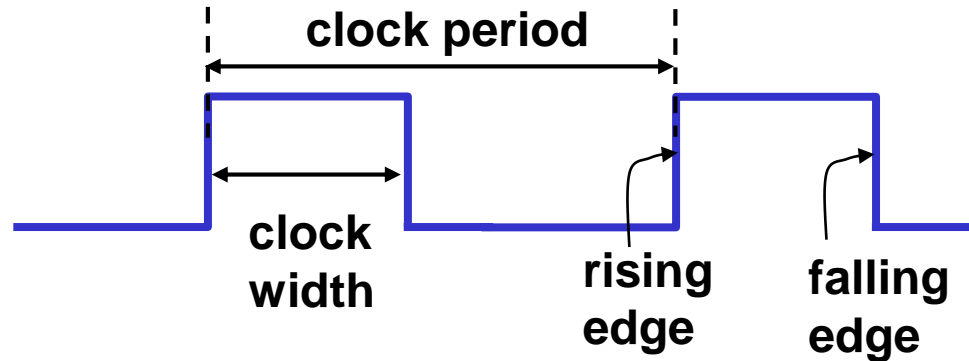
Synchronous storage components store data and perform some simple operations.

Synchronous storage components include:

- |                    |              |
|--------------------|--------------|
| (1) registers      | (2) counters |
| (3) register files | (4) memories |
| (5) queues         | (6) stacks   |

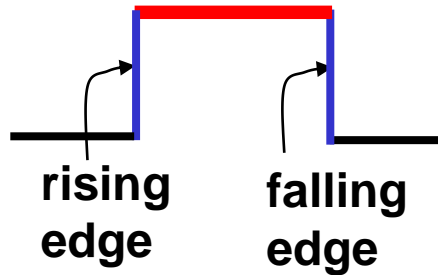


# Clock Period



- Clock period (measured in micro or nanoseconds) is the time between successive transitions in the same direction (second/cycle)
- Clock frequency (measured in MHz or GHz) is the reciprocal of clock period (cycle/second)
- Clock width is the time interval during which clock is equal to 1
- Duty cycle is the ratio of the clock width and clock period
- Clock signal is active high if the changes occur at the rising edge or during the clock width. Otherwise, it is active low

# Latch and Flip-Flop



**Latches** are level-sensitive since they respond to input changes during clock width. ➡ Latches are difficult to work with for this reason.

**Flip-Flops** respond to input changes only during the change in clock signal (the rising edge or the falling edge).

They are easy to work with though more expensive than latches.

Two basic styles of flip-flops are available:

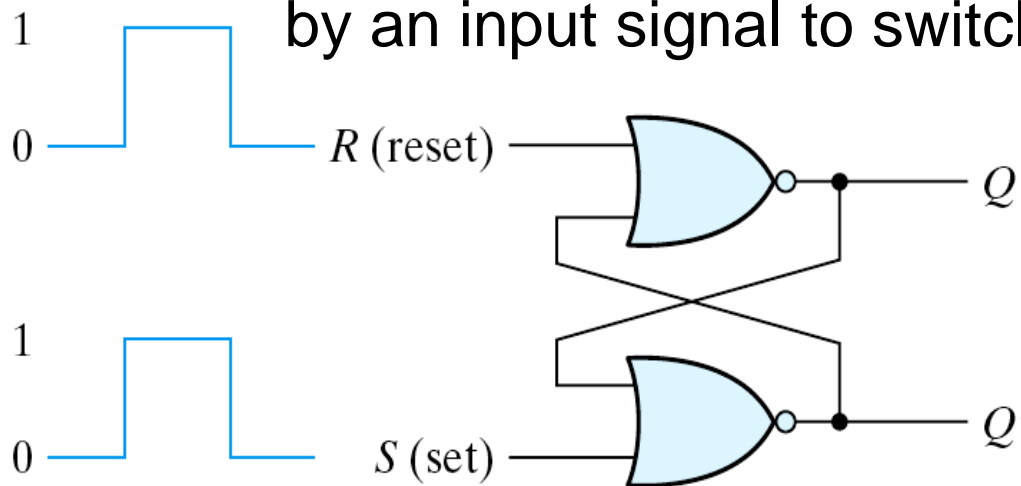
- (1) master-slave
- (2) edge-triggered

# Latches (1/2)

- The most basic types of flip-flops operate with signal levels  
→ latch
- All FFs are constructed from the latches introduced here

A FF can maintain a binary state indefinitely until directed

by an input signal to switch states



(a) Logic diagram

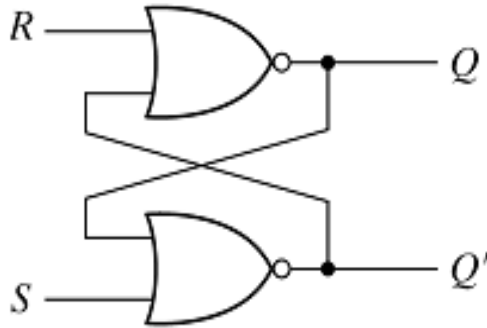
$S$	$R$	$Q$	$Q'$
1	0	1	0
0	0	1	0 (after $S = 1, R = 0$ )
0	1	0	1
0	0	0	1 (after $S = 0, R = 1$ )
1	1	0	0 (forbidden)

(b) Function table

Two NOR gates

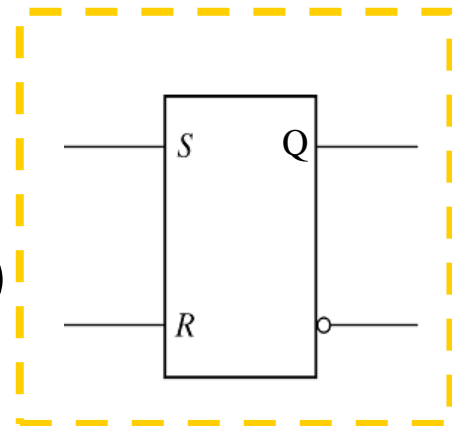
Set → 1, Reset → 0

# Latches (2/2)

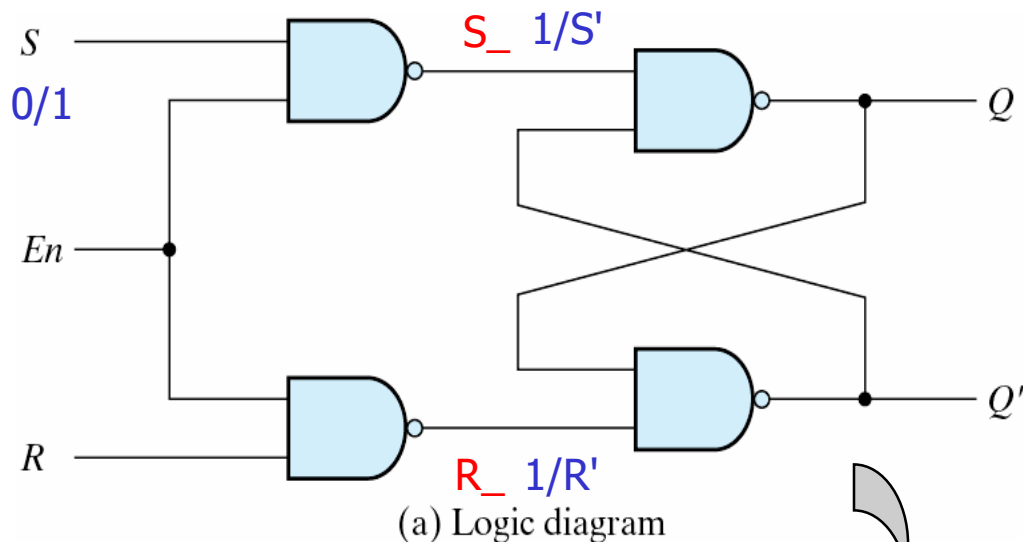


S	R	P=Q' Q		
0	0	*	*	// a stable state in the previous state
1	0	0	1	// change to another stable state "Set"
0	0	0	1	// remain in the previous state
0	1	1	0	// change to another stable state "Reset"
0	0	1	0	// remain in the previous state
1	1	0	0	// oscillate (unpredictable) if next SR=00

- more complicated types can be built upon it the condition should be avoided
- an asynchronous sequential circuit
- (S,R)= (0,0): no operation
- (S,R)=(0,1): reset (Q=0, the clear state)
- (S,R)=(1,0): set (Q=1, the set state)
- (S,R)=(1,1): indeterminate state (Q=Q'=0)
- consider (S,R) = (1,1)  $\Rightarrow$  (0,0)



# SR Latch with Control Input

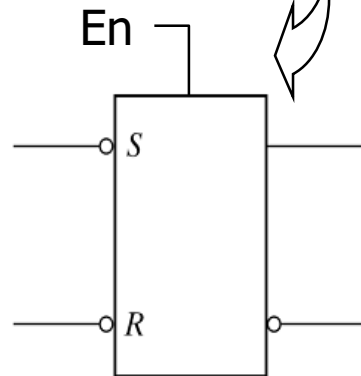


The complement output of the previous R'S' latch.

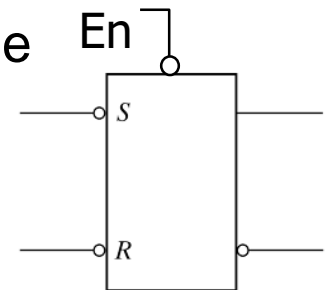
$En$	$S$	$R$	Next state of $Q$
0	X	X	No change
1	0	0	No change
1	0	1	$Q = 0$ ; reset state
1	1	0	$Q = 1$ ; set state
1	1	1	Indeterminate

(b) Function table

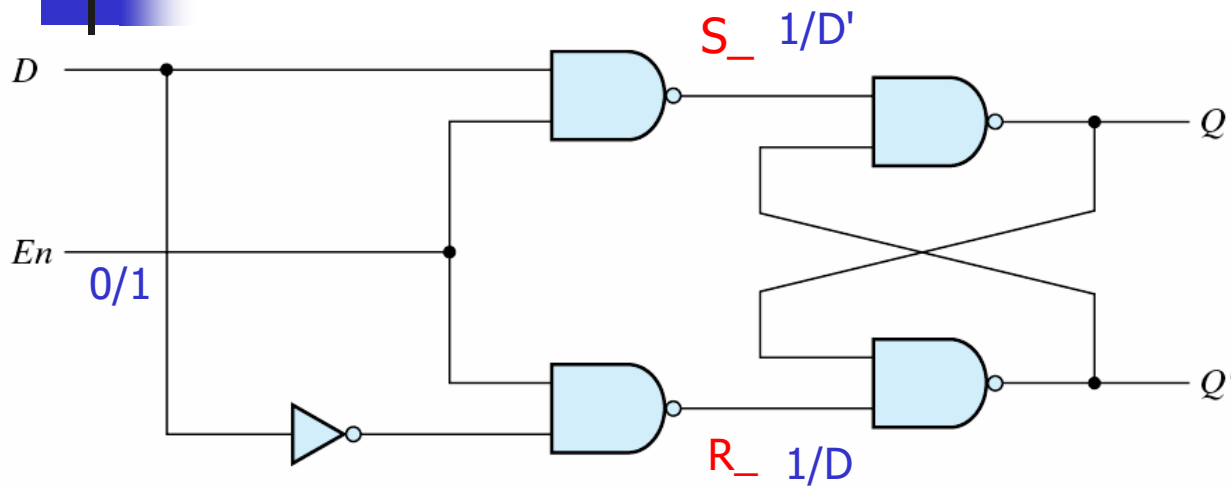
$En=0$ , no change  
 $En=1$ , enable



$En=1$ , no change  
 $En=0$ , enable



# D Latch (Transparent Latch)

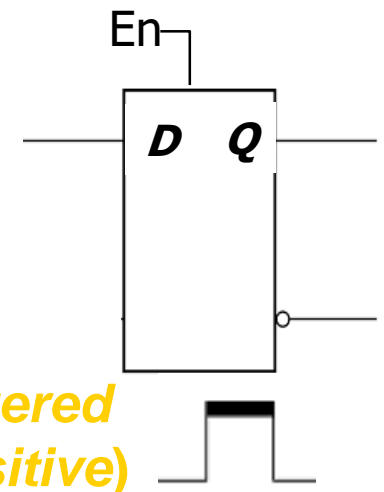


(a) Logic diagram

$En$	$D$	Next state of $Q$
0	X	No change
1	0	$Q = 0$ ; reset state
1	1	$Q = 1$ ; set state

(b) Function table

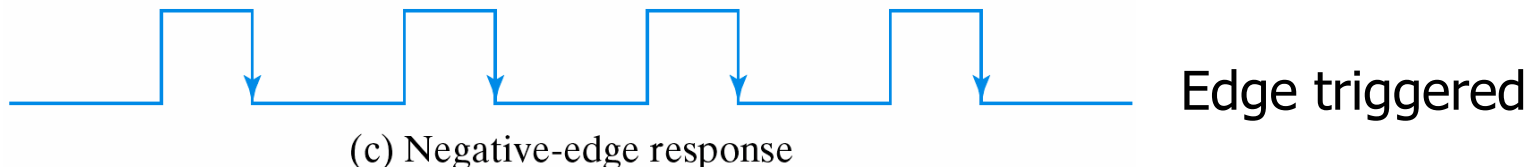
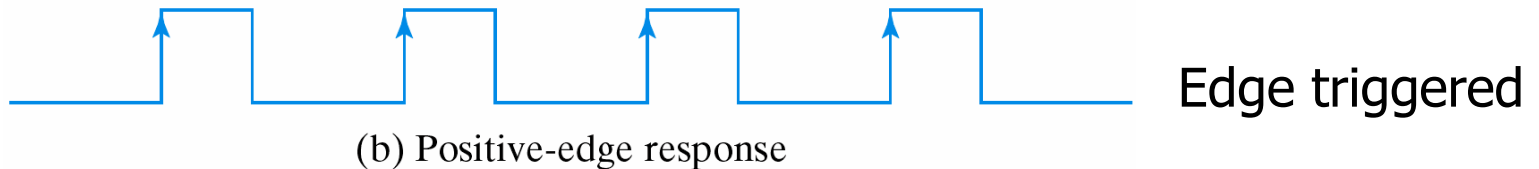
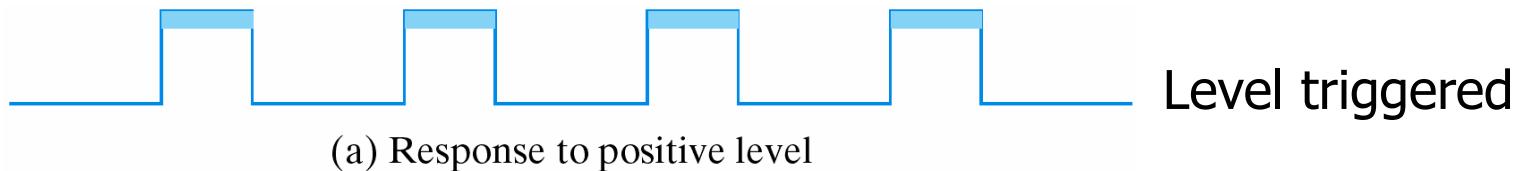
- eliminate the undesirable conditions of the indeterminate state in the RS flip-flop
- D: data
- gated D-latch
- $D \Rightarrow Q$  when  $En=1$ ; no change when  $En=0$



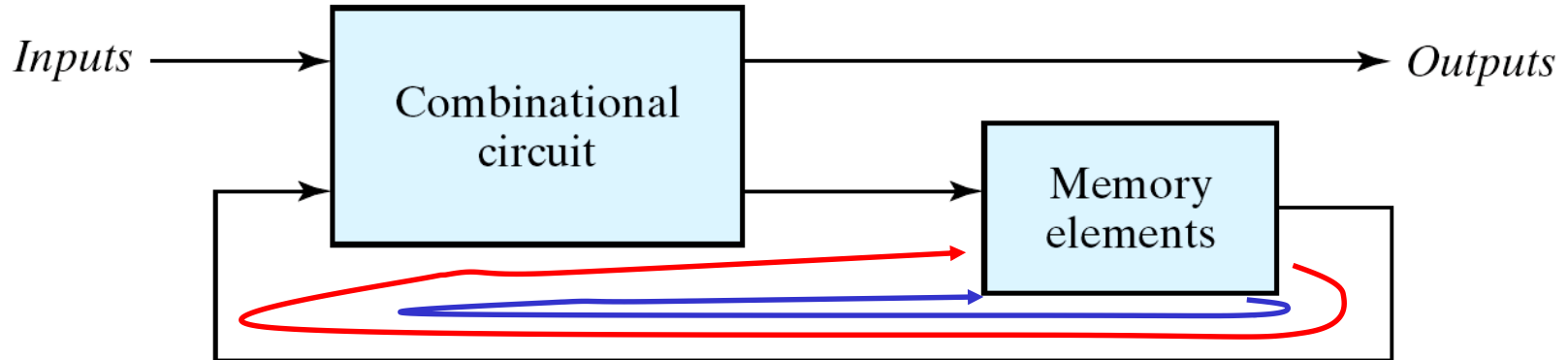
level triggered  
(level-sensitive)

# Flip-Flops

- A trigger
  - The state of a latch or flip-flop is switched by a change of the control input
- Level triggered – latches
- Edge triggered – flip-flops



# Problem of Latch



- If level-triggered flip-flops are used
  - the feedback path may cause instability problem (since the time interval of logic-1 is too long)
  - multiple transitions might happen during logic-1 level
- Edge-triggered flip-flops
  - the state transition happens only at the edge
  - eliminate the multiple-transition problem





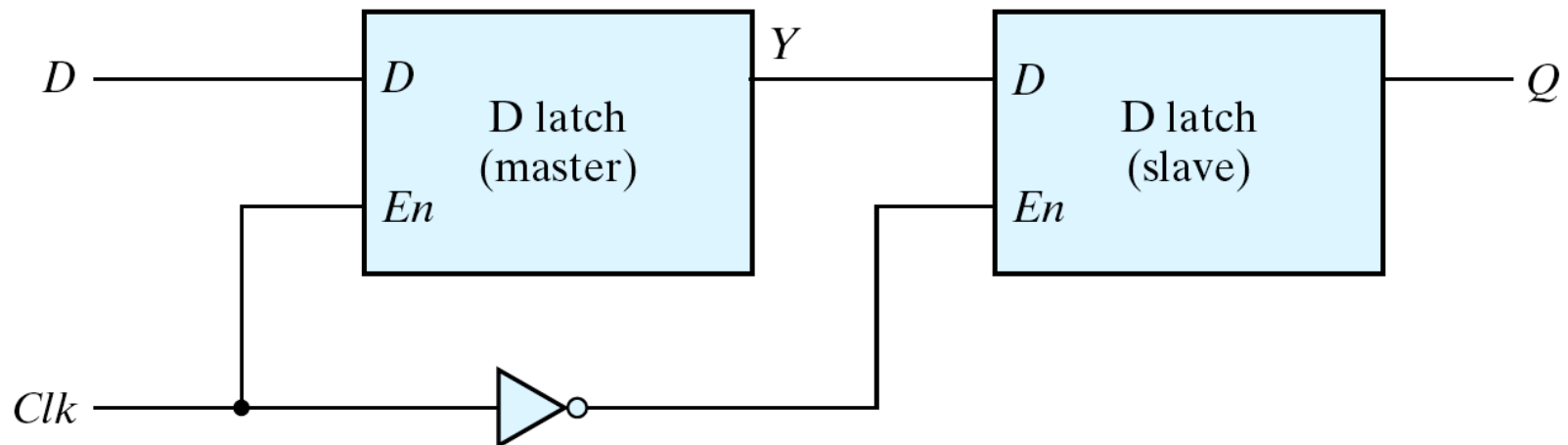
# Edge-Triggered D Flip-Flop

Two designs to solve the problem of latch:

- a. Master-slave D flip-flop
- b. Edge-trigger D flip-Flop

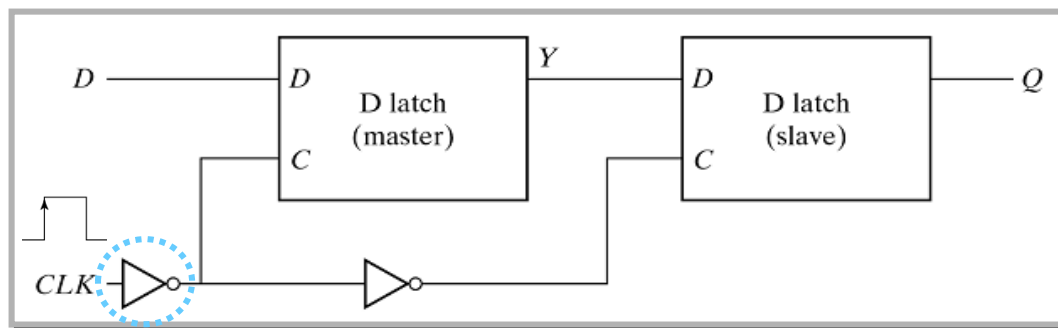
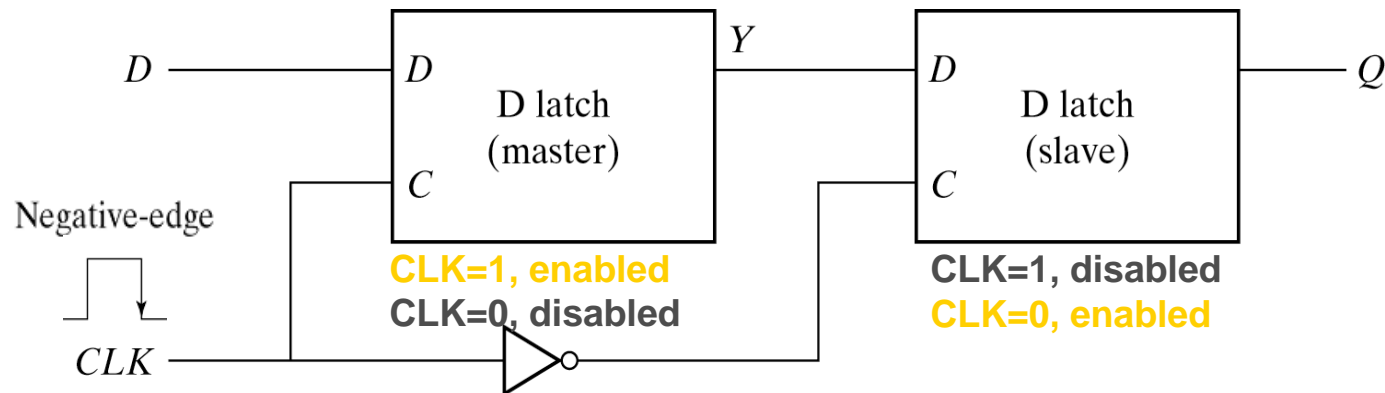
## ■ Master-slave D flip-flop

- two separate flip-flops
- a master latch (positive-level triggered)
- a slave latch (negative-level triggered)



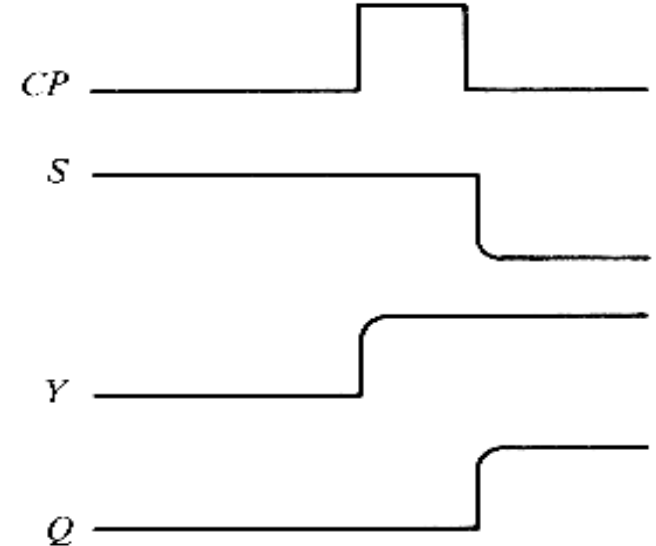
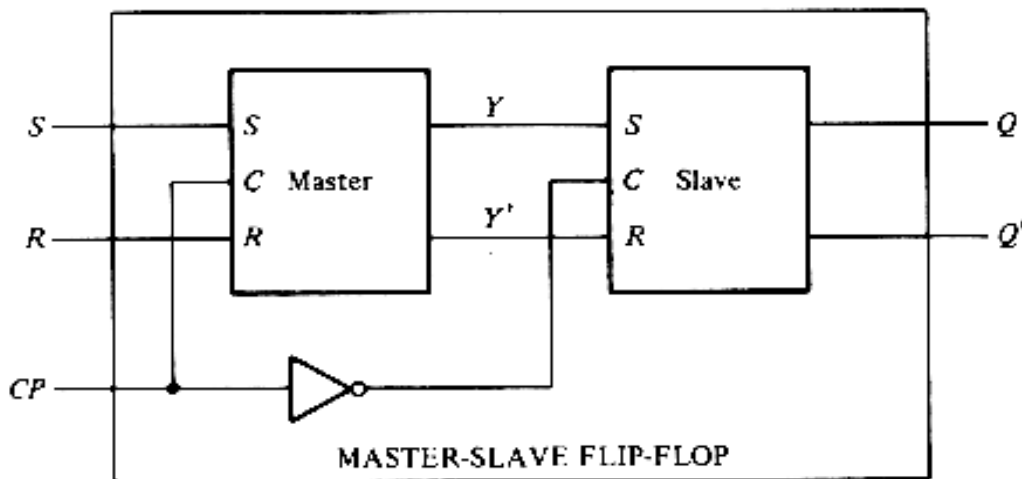
# Master-slave D flip-flop (1/2)

- Two D latches and one inverter (8 gates)
- The circuit samples D input and changes its output Q only at the negative-edge of CLK
- isolate the output of FF from being affected while its input is changing



# Master-slave D flip-flop (2/2)

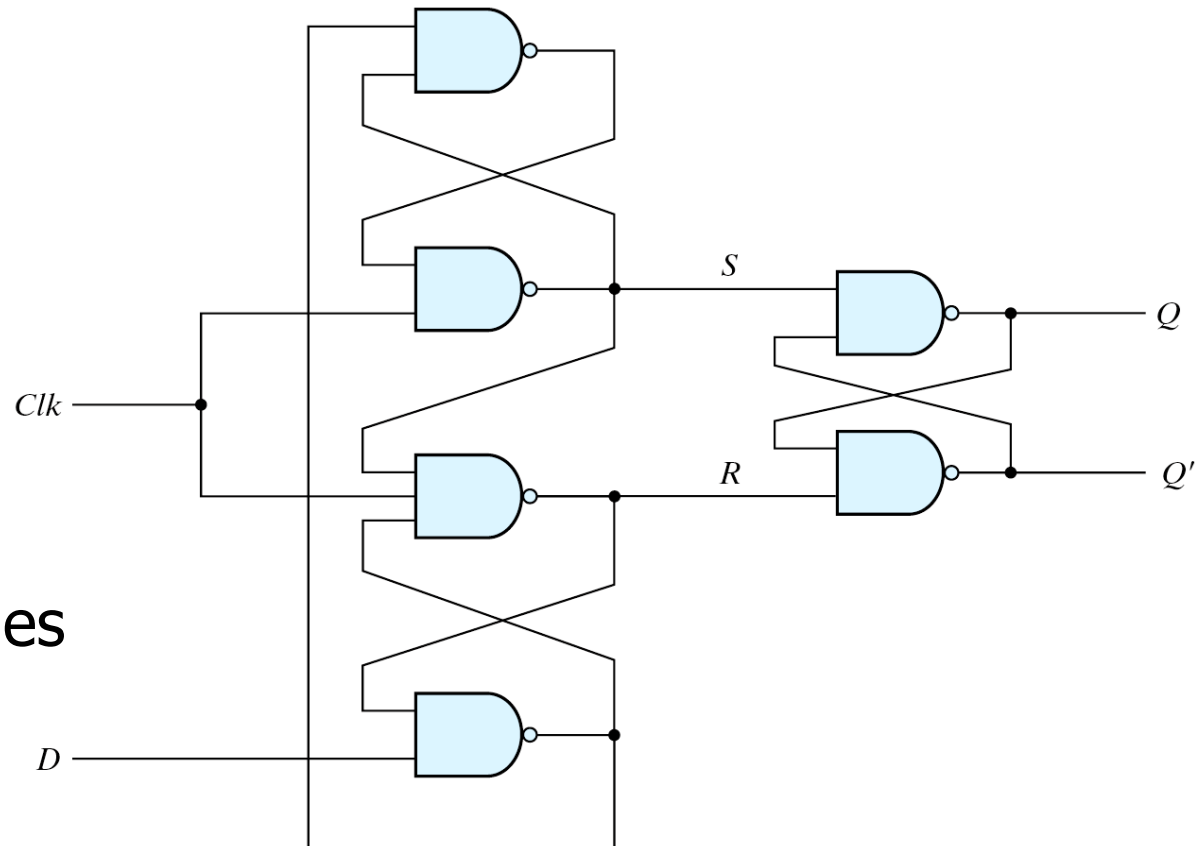
- $CP = 1$ :  $(S,R) \Rightarrow (Y,Y')$ ;  $(Q,Q')$  holds
- $CP = 0$ :  $(Y,Y')$  holds;  $(Y,Y') \Rightarrow (Q,Q')$
- $(S,R)$  could not affect  $(Q,Q')$  directly
- the state changes coincide with the negative-edge transition of  $CP$



# Edge-Triggered Flip-Flops (1/2)

the state changes during a clock-pulse transition

A D-type positive-edge-triggered flip-flop

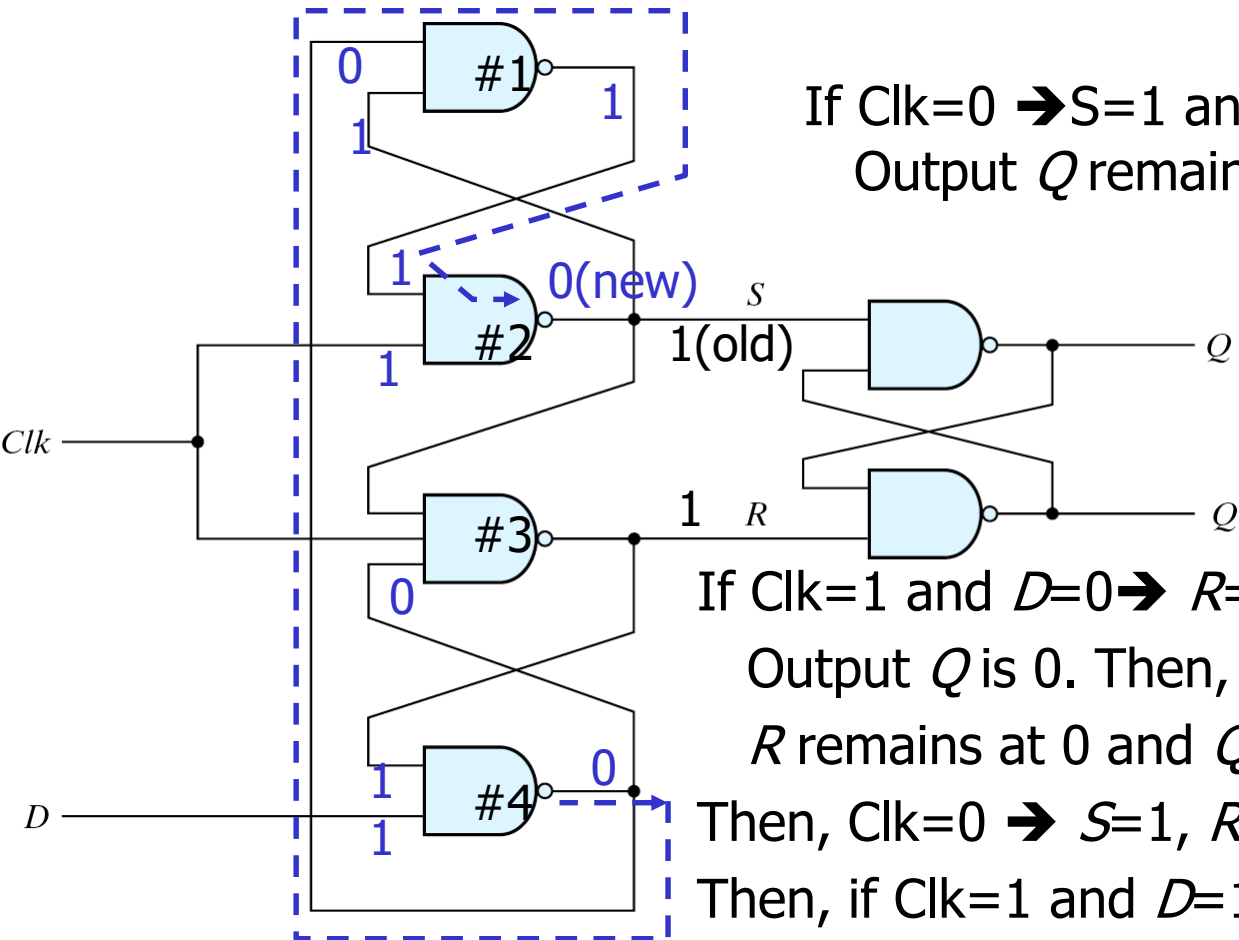


Three SR latches

# Edge-Triggered Flip-Flops (2/2)

$(S,R) = (0,1): Q = 1$        $(S,R) = (1,0): Q = 0$

$(S,R) = (1,1):$  no operation       $(S,R) = (0,0):$  should be avoided



If  $\text{Clk}=0 \rightarrow S=1$  and  $R=1 \rightarrow$  no operation.  
Output  $Q$  remains in the present state.

If  $\text{Clk}=1$  and  $D=0 \rightarrow R=0 \rightarrow$  Reset.

Output  $Q$  is 0. Then, if  $D$  changes to 1,  
 $R$  remains at 0 and  $Q$  is 0.

Then,  $\text{Clk}=0 \rightarrow S=1, R=1 \rightarrow$  no operation ( $Q=0$ )

Then, if  $\text{Clk}=1$  and  $D=1 \rightarrow S=0 \rightarrow$  Set.

Output  $Q$  is 1. (see the blue dot-line flow)

Then, if  $D$  changes to 0,  $S$  remains at 0 and  $Q=1$ ;



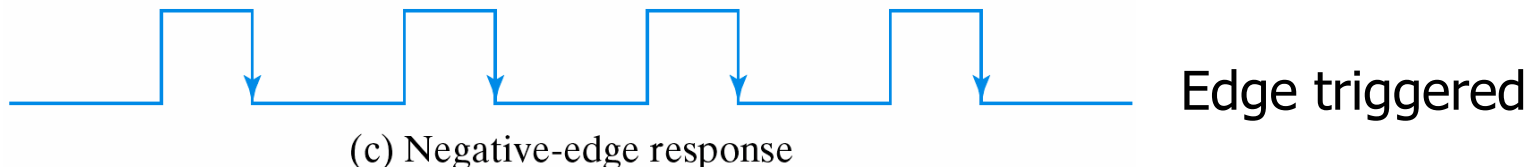
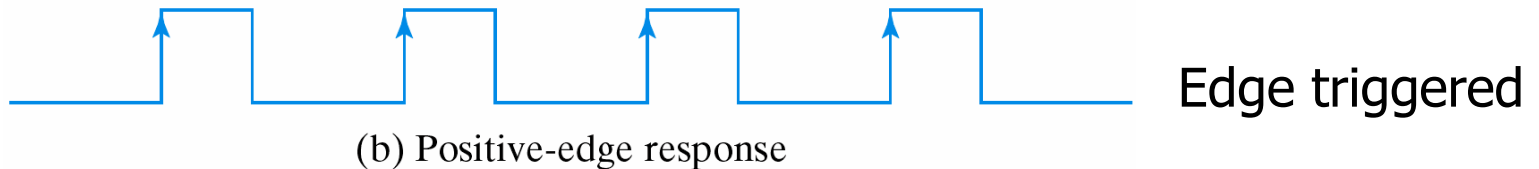
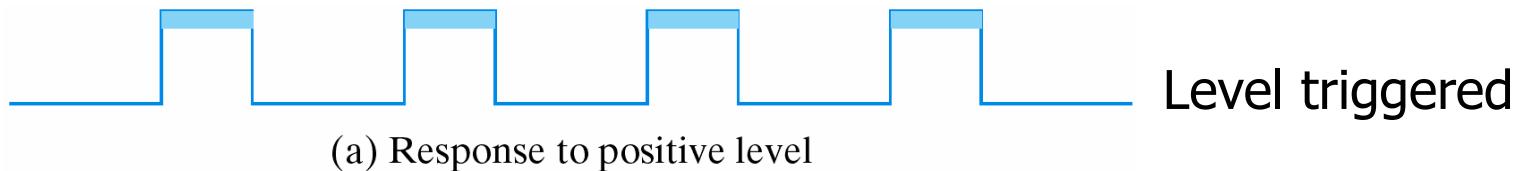
# Positive-Edge-Triggered Flip-Flops

---

- Summary
  - $\text{Clk}=0$ :  $(S,R) = (1,1)$ , no state change
  - $\text{Clk}=\uparrow$ : state change once
  - $\text{Clk}=1$ : state holds
  - eliminate the feedback problems in sequential circuits
- All flip-flops must make their transition at the same time

# Flip-Flops

- A trigger
  - The state of a latch or flip-flop is switched by a change of the control input
- Level triggered – latches
- Edge triggered – flip-flops





# Setup Time and Hold Time

---

- The setup time

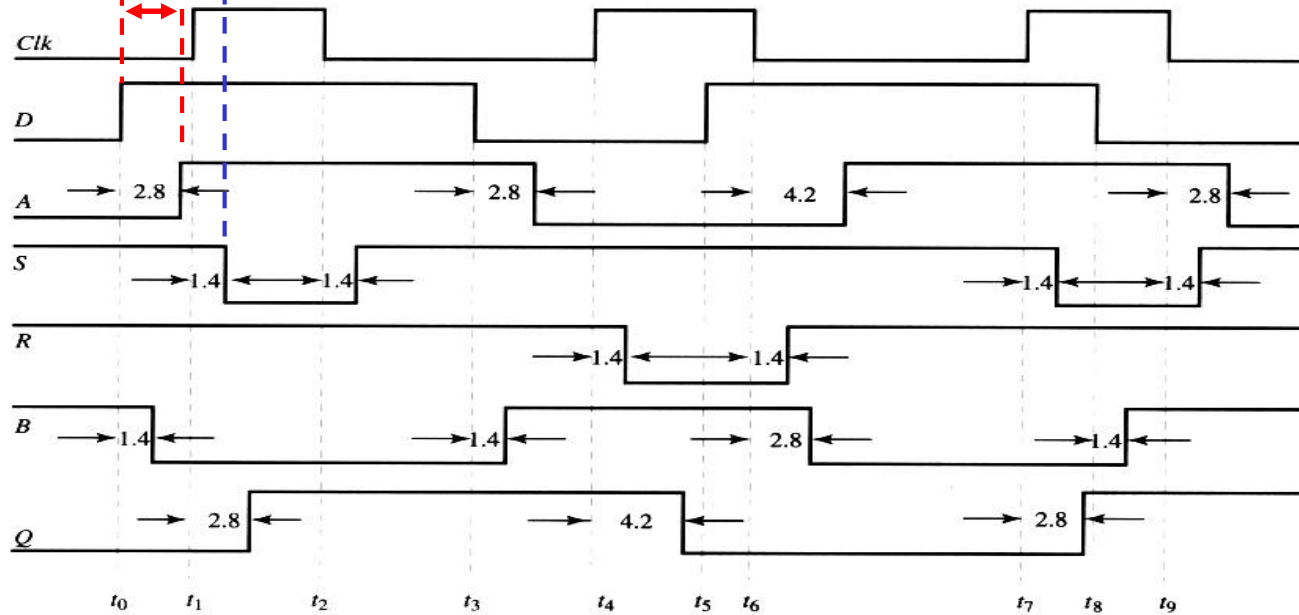
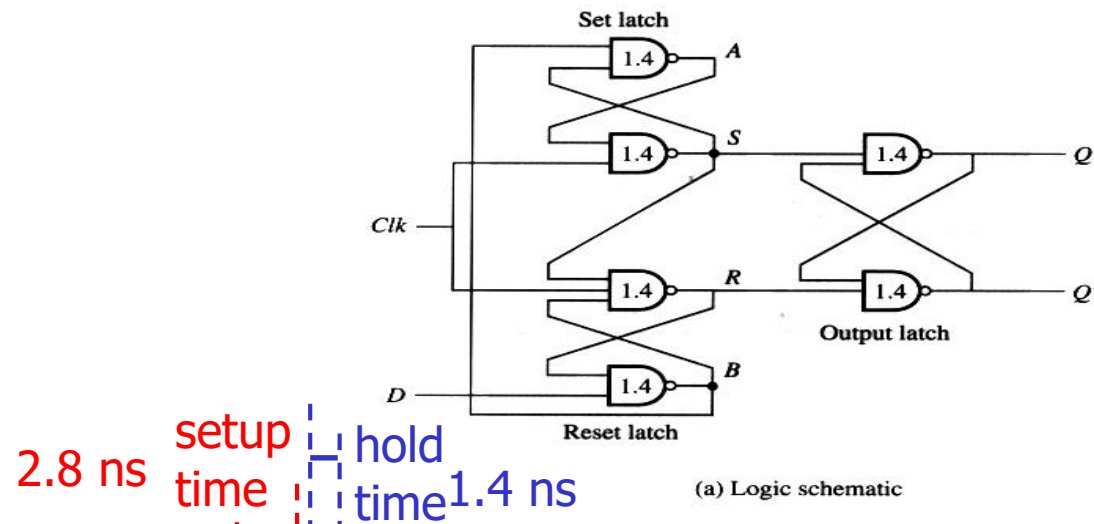
- D input must be maintained at a constant value prior to the application of the positive Clk pulse
- = the propagation delay through gates 4 and 1
- data to the internal latches

- The hold time

- D input must not changes after the application of the positive Clk pulse
- = the propagation delay of gate 3 (try to understand)
- clock to the internal latch

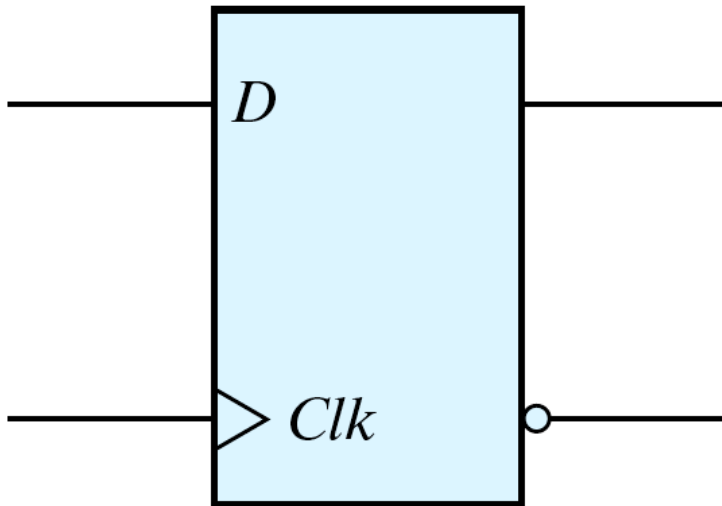


# Timing Diagram

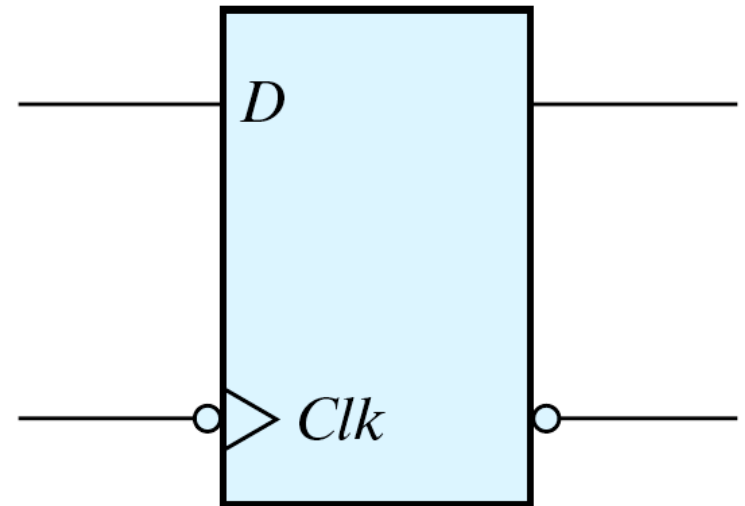


# Positive-Edge vs. Negative-Edge

- The edge-triggered D flip-flops
  - The most economical and efficient
  - Positive-edge and negative-edge



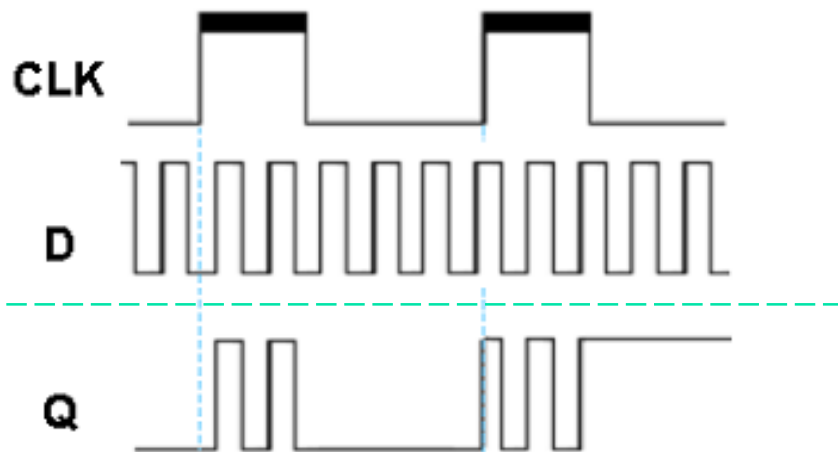
(a) Positive-edge



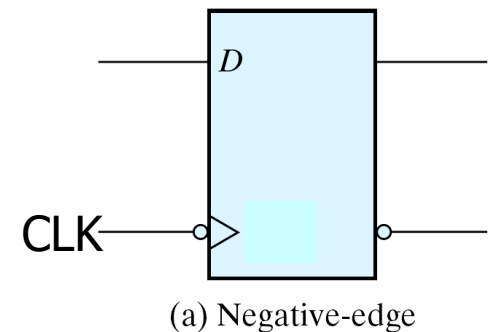
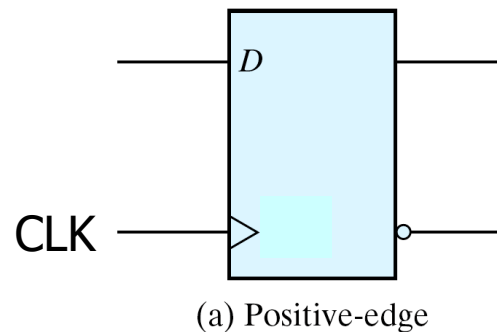
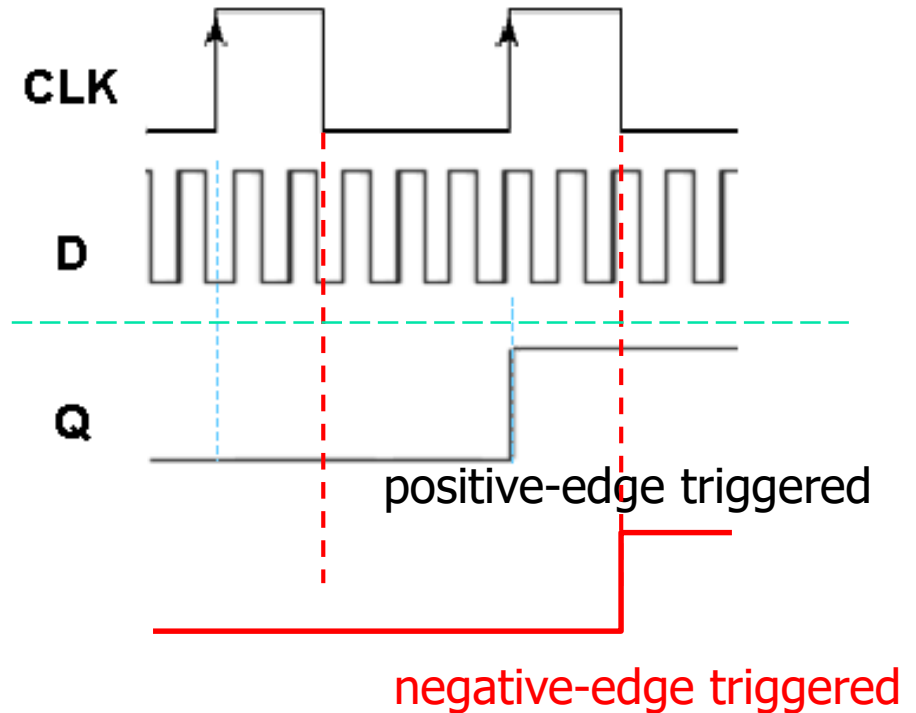
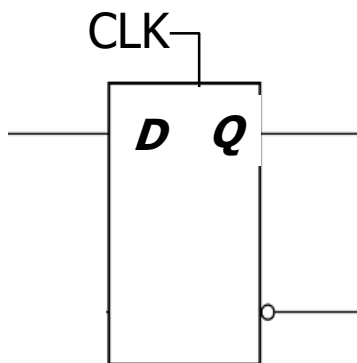
(a) Negative-edge

# Latch vs. Flip-Flop

- Level triggered



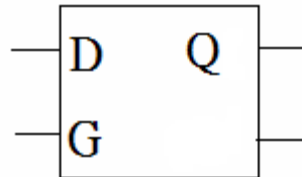
## Latch



# Latch

## D-type latch (ignoring delay)

D	G	Q(t+1)
X	0	Q (t)
0	1	0
1	1	1



```
module p163(G, D, Q);  
  input G, D;  
  output Q;  reg Q;
```

```
  always @(D or G)  
  begin
```

```
    if(G)
```

```
      Q = D;
```

```
  end
```

```
endmodule
```

/p163\_tb/G

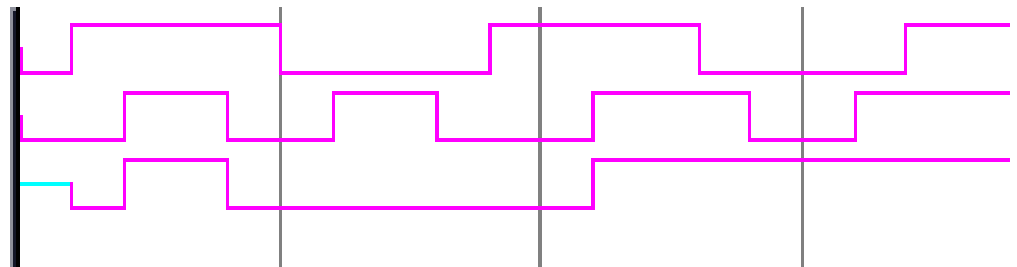
/p163\_tb/D

/p163\_tb/Q

-No Data-

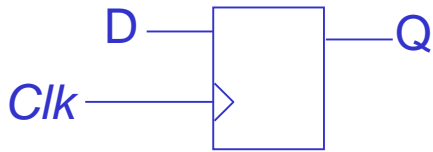
-No Data-

-No Data-

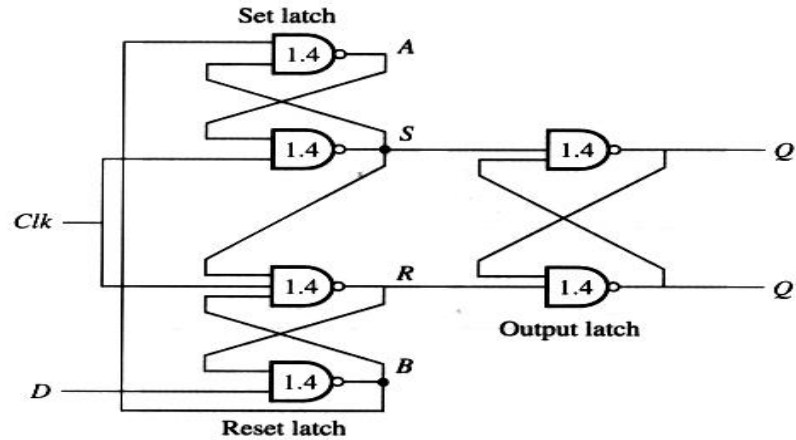


# Flip-Flop (1/2)

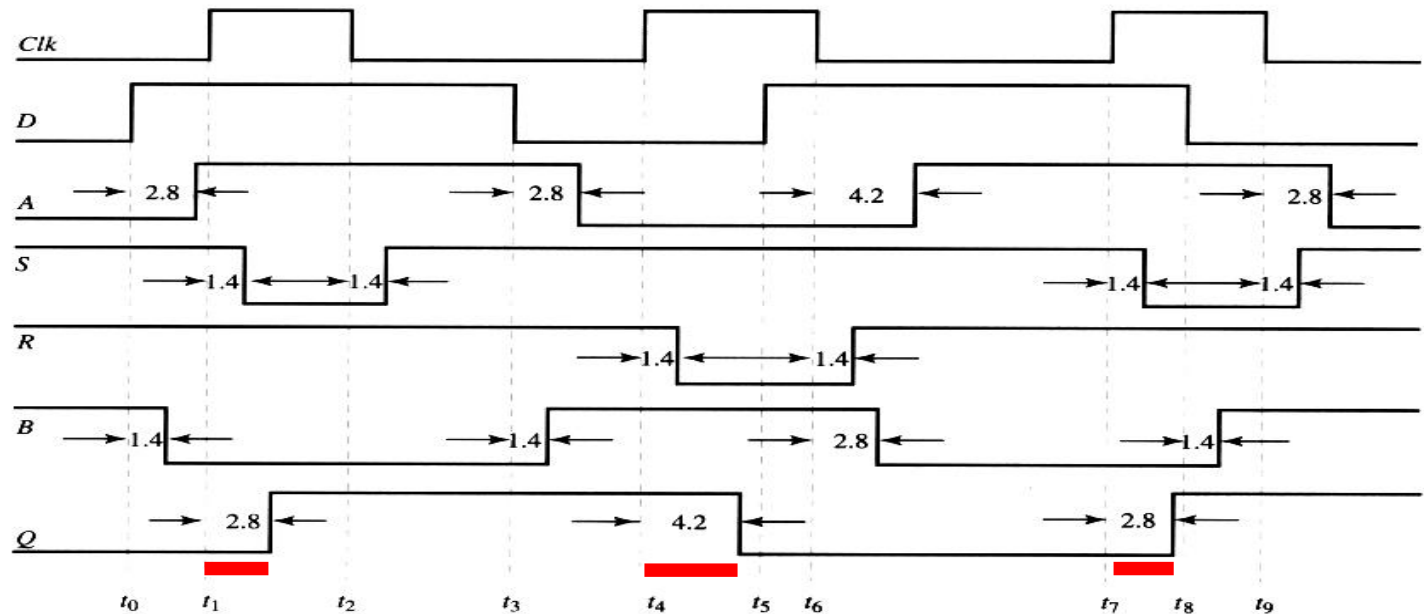
## D flip-flop



## Edge-triggered flip-flop



(a) Logic schematic

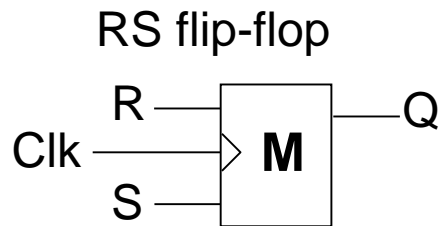


(b) Timing diagram

# Flip-Flop (2/2)

characteristic table

excitation table

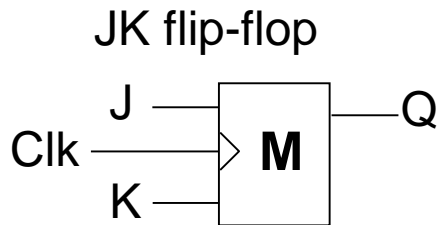


S	R	Q(t+1)
0	0	Q(t)
0	1	0
1	0	1
1	1	NA

$$Q(\text{next}) = S + R'Q$$

$$SR = 0$$

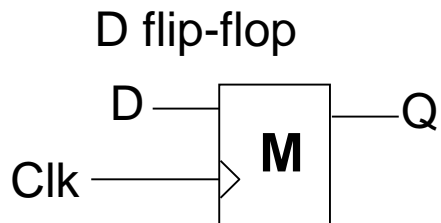
Q(t)	Q(t+1)	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0



J	K	Q(t+1)
0	0	Q(t)
0	1	0
1	0	1
1	1	Q'(t)

$$Q(\text{next}) = JQ' + K'Q$$

Q(t)	Q(t+1)	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0



D	Q(t+1)
0	0
1	1

$$Q(\text{next}) = D$$

Q(t)	Q(t+1)	D
0	0	0
0	1	1
1	0	0
1	1	1

# Flip-Flop Inference

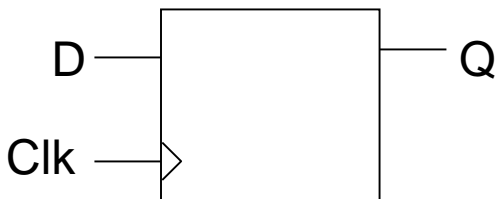
D	$Q_{(t+1)}$
0	0
1	1

D Flip-flop

```
module D_FF(Clk, D, Q);  
input Clk, D;  
output Q;  
Reg Q;
```

```
always @(posedge Clk)  
begin  
    Q=D;  
end  
endmodule
```

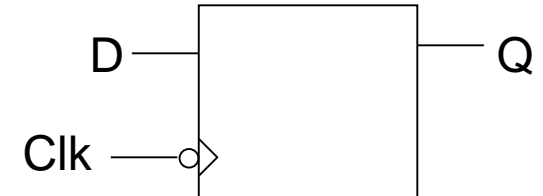
At every positive edge  
of Clk, Q is set as D



```
module D_FF(Clk, D, Q);  
input Clk, D;  
output Q;  
Reg Q;
```

At every negative edge  
of Clk, Q is set as D

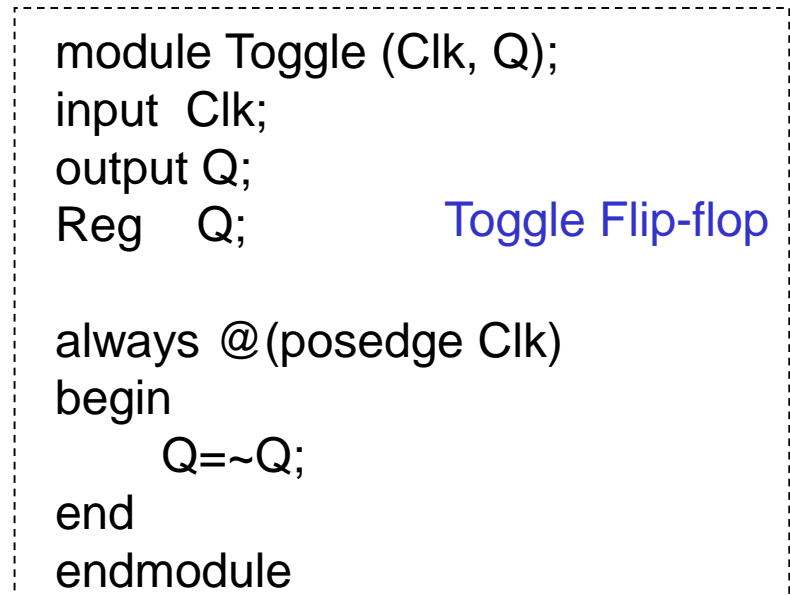
```
always @(negedge Clk)  
begin  
    Q=D;  
end  
endmodule
```



```
module Toggle (Clk, Q);  
input Clk;  
output Q;  
Reg Q;
```

Toggle Flip-flop

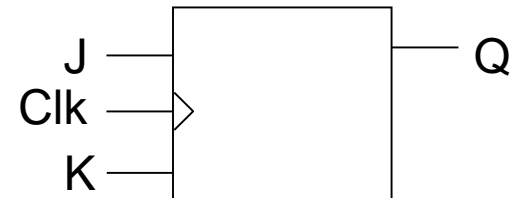
```
always @(posedge Clk)  
begin  
    Q=~Q;  
end  
endmodule
```



# JK Flip-Flop

J	K	$Q_{(t+1)}$	QB
0	0	Q	Q'
0	1	0	1
1	0	1	0
1	1	Q'	Q

JK Flip-flop



```
module JK_FF(Clk, J, K, Q);
input  Clk, J, K;
output Q, Q_Bar;
reg    Q, Q_Bar;
always @(posedge Clk)
begin
    case({J,K})
        2'b00:
            Q=Q;
```

```
        2'b01:
            Q=0;
        2'b10:
            Q=1;
        2'b11:
            Q=~Q;
    endcase
end
endmodule
```



# D Flip-flop with Reset

## D Flip-flop with asynchronous reset

If Reset changes from 1 to 0,  
then reset D flip-flop anyway.

Otherwise,  $Q=D$ .

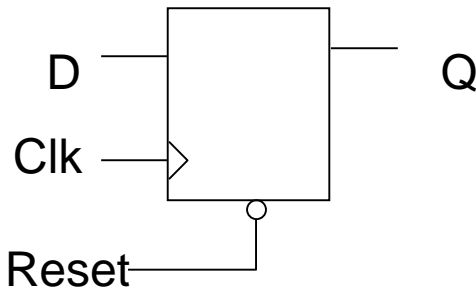
```
module DFF_AR(Clk, Reset, D, Q);  
input Clk, Reset, D;  
output Q; reg Q;
```

```
always @(posedge Clk or negedge Reset)  
begin
```

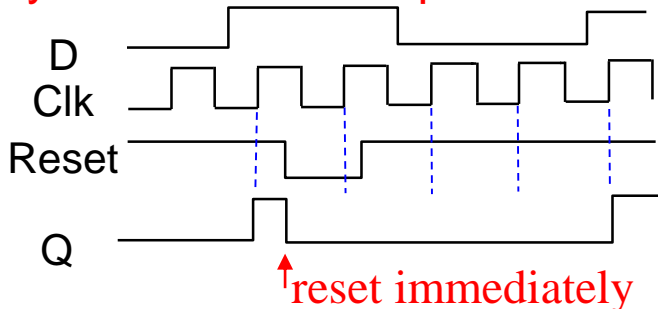
```
    if(!Reset)  
        Q=0;
```

```
    else  
        Q=D; end
```

```
endmodule
```



Asynchronous -- Respond immediately !



## D Flip-flop with synchronous reset

At every positive edge of Clk,  
if Reset==0, then reset D flip-flop  
(if Reset==1, then  $Q=D$ ).

```
module DFF_SR(Clk, Reset, D, Q);  
input Clk, Reset, D;  
output Q; reg Q;
```

```
always @(posedge Clk)  
begin
```

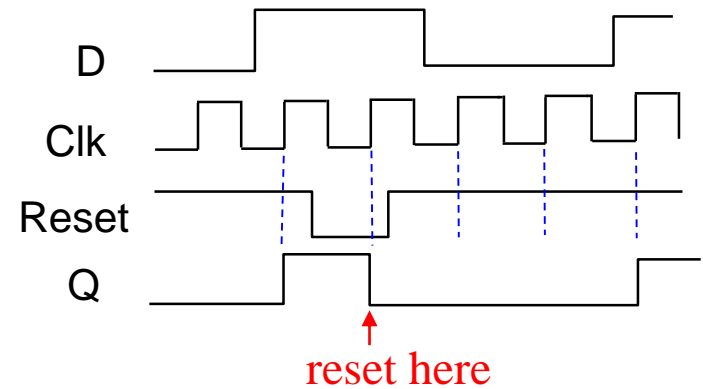
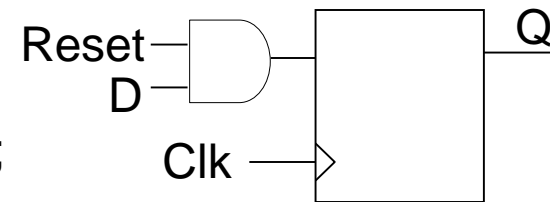
```
    if(!Reset)
```

```
        Q=0;
```

```
    else
```

```
        Q=D; end;
```

```
endmodule
```



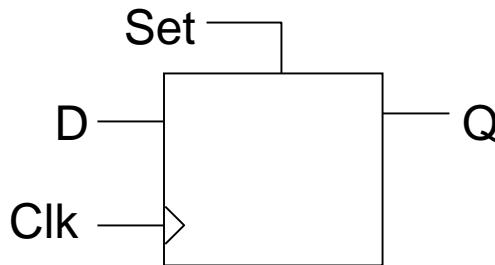
# D Flip-flop with Set

## D Flip-flop with asynchronous set

If Set changes from 0 to 1,  
then set D flip-flop to 1 anyway.  
Otherwise,  $Q=D$ .

```
module DFF_AS(Clk, Set, D, Q);  
input  Clk, Set, D;  
output Q;  
reg    Q;
```

```
always @(posedge Clk or posedge Set)  
begin  
    if(Set)  
        Q=1;  
    else  
        Q=D;  
end  
endmodule
```

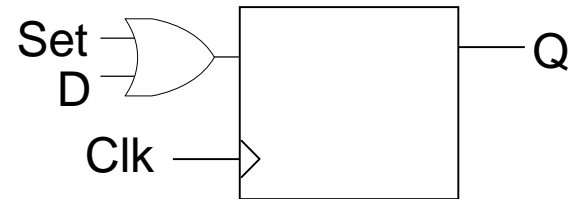


## D Flip-flop with synchronous set

At every positive edge of Clk,  
if  $Set==1$ , then set D flip-flop to 1  
(if  $Set==0$ ,  $Q=D$ ).

```
module DFF_SS(Clk, Set, D, Q);  
input  Clk, Set, D;  
output Q;  
reg    Q;
```

```
always @(posedge Clk)  
begin  
    if(Set)  
        Q=1;  
    else  
        Q=D;  
end  
endmodule
```





# D Flip-flop with Set and Reset

```
module DFF_ARS(Clk, Set,
Reset, D, Q);
input  Clk, Set, Reset, D;
output Q;
reg    Q;

always @(posedge Clk or
negedge Reset or posedge Set )
begin
    if(!Reset)
        Q=0;
    else if(Set)
        Q=1;
    else
        Q=D;
end
endmodule
```

D Flip-flop with asynchronous Set  
and asynchronous Reset

```
module DFF_SRS(Clk, Set, Reset,
D, Q, QB);
input  Clk, Set, Reset, D;
output Q, Q_Bar;
reg    Q, Q_Bar;
always @(posedge Clk)
begin
    if(!Reset)
        Q = 0;
    else if(Set)
        Q=1;
    else
        Q=D;
end
endmodule
```

D Flip-flop with synchronous Set  
and synchronous Reset

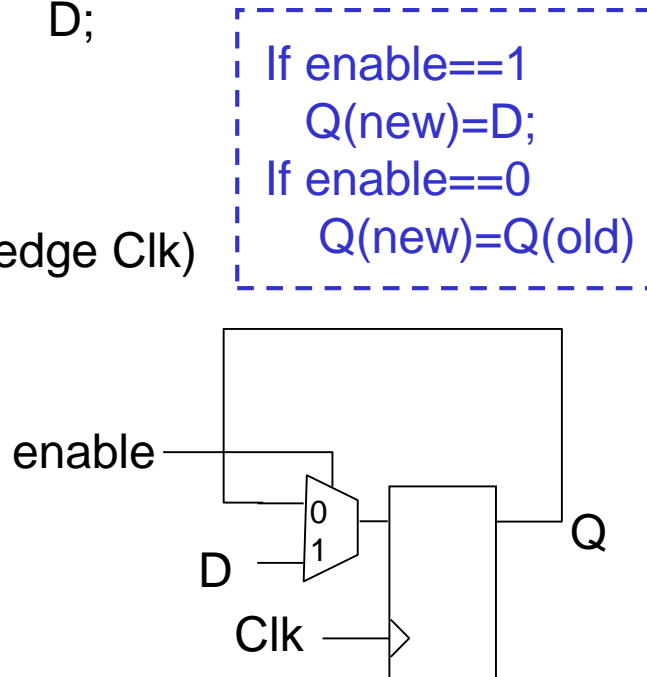
# D Flip-flop with Enable or Load

## D Flip-flop with synchronous enable

```
module DFF_MAL(Clk, enable,  
D, Q);
```

```
input      Clk, enable;  
input [3:0] D;  
output [3:0] Q;  
reg [3:0] Q;
```

```
always @(posedge Clk)  
begin  
    if(enable)  
        Q = D;  
end  
endmodule
```

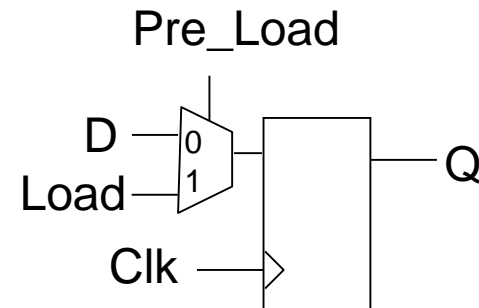


## D Flip-flop with synchronous load

```
module DFF_MSL(Clk, Pre_Load,  
Load, D, Q);
```

```
input      Clk, Pre_Load;  
input [3:0] Load, D;  
Output [3:0] Q;  
reg [3:0] Q;
```

```
always @(posedge Clk)  
begin  
    if(Pre_Load)  
        Q = Load;  
    else  
        Q = D;  
end  
endmodule
```

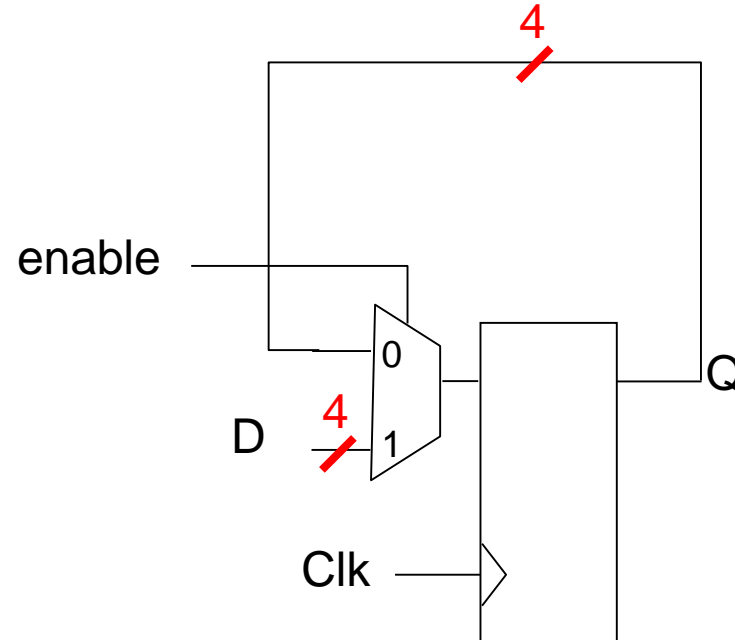


# Registers

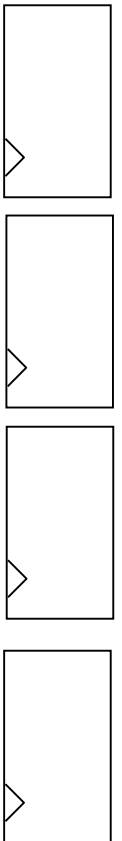
```
module DFF_MAL(Clk, enable,  
D, Q);
```

```
input      Clk, enable;  
input [3:0] D;  
output [3:0] Q;  
reg [3:0] Q;
```

```
always @(posedge Clk)  
begin  
    if (enable)  
        Q = D;  
end  
endmodule
```



4-bit register = 4 flip-flops



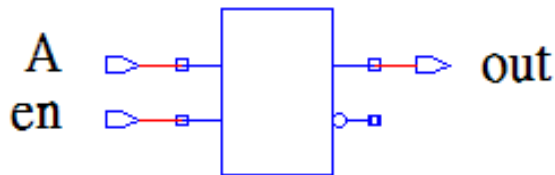
# Watch Out for Unintentional Latches (1/6)

```
module latch_if1(en,A,out);  
    input en, A;  
    output out;  
    reg out;
```

```
    always @(en)  
    begin  
        if(en)  
            out = A;
```

```
    end
```

```
endmodule
```



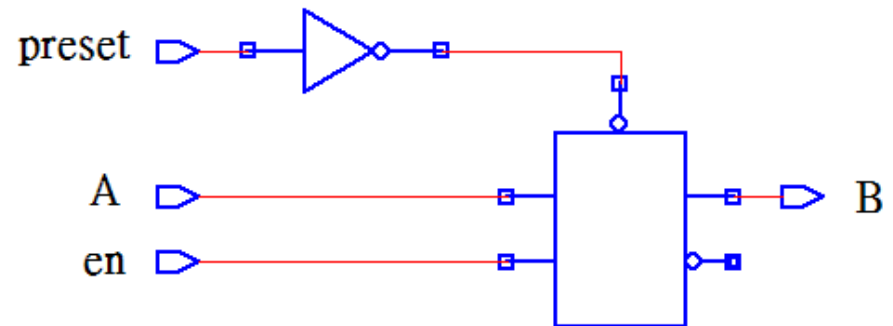
If en == 1 out = A  
else out (new) = out (old)

```
module latch_4(en, preset, A, B);  
    input en, preset, A;  
    output B;  
    reg B;  
    always @(en or preset or A)  
    begin
```

```
        if(preset)  
            B = 1;  
        else if(en)  
            B = A;
```

```
    end
```

```
endmodule
```

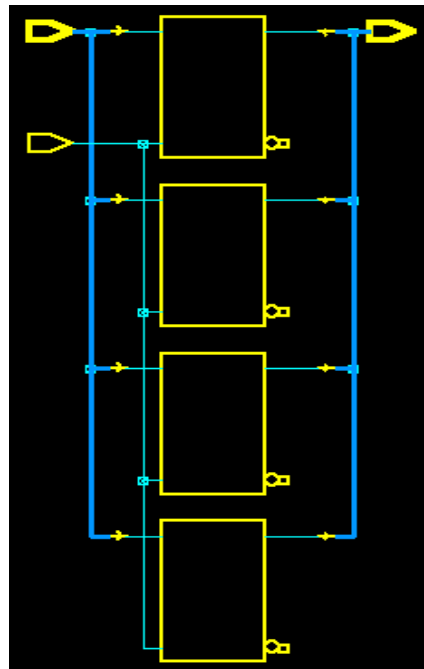


# Watch Out for Unintentional Latches (2/6)

```
Module Latch(In, Enable, Out);
input      Enable;
input [3:0] In;
output [3:0] Out;
```

```
always @(In or Enable)
begin
    if(Enable)
        Out=In;
end
endmodule
```

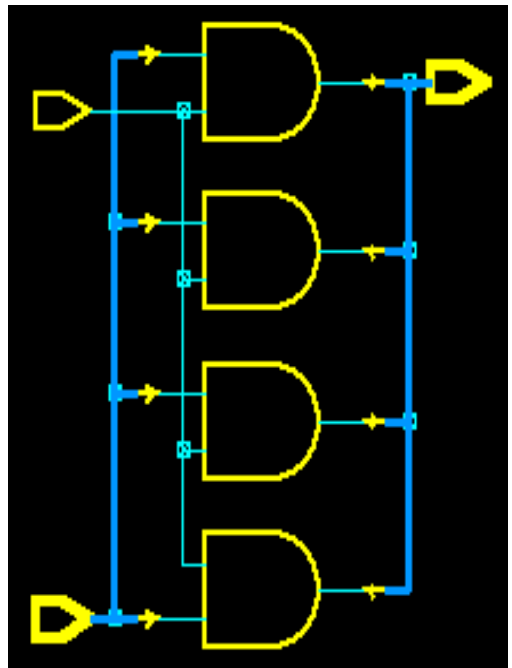
If Enable == 1  
Out (new) = In  
If Enable == 0  
Out (new) = Out (old)



```
Module Latch(In, Enable, Out);
input      Enable;
Input [3:0] In;
output [3:0] Out;
```

```
Out=0;
if(Enable)
    Out=In;
```

```
always @(In or Enable)
begin
    if(Enable)
        Out=In;
    else
        Out=0;
end
endmodule
```

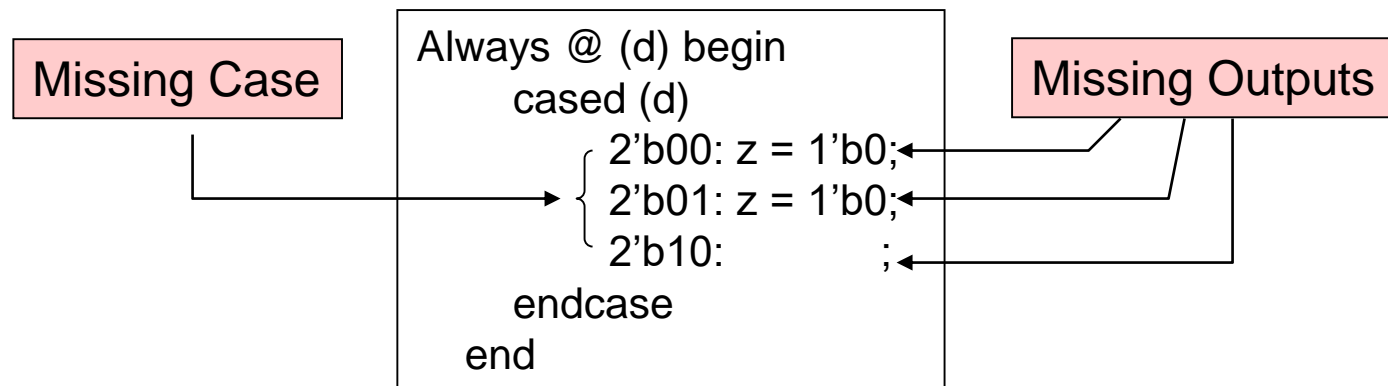


No latch inference

# Watch Out for Unintentional Latches (3/6)

## Watch Out for Unintentional Latches

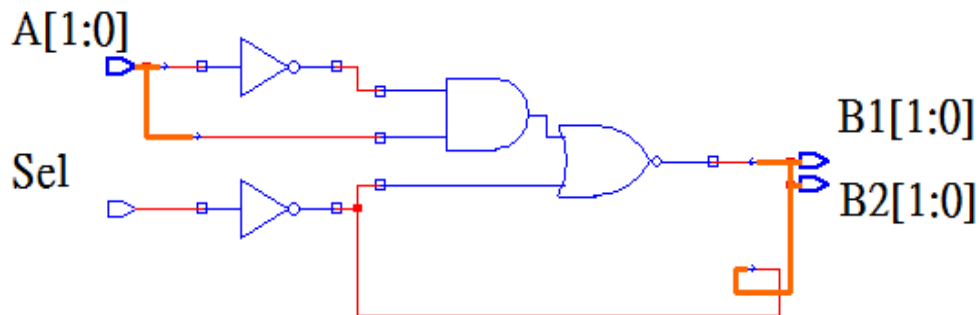
- Completely specify all clauses for every **case** and **if** statement
- Completely specify all output for every clause of each **case** or **if** statement
- Fail to do so will cause latches or flip-flops to be synthesized



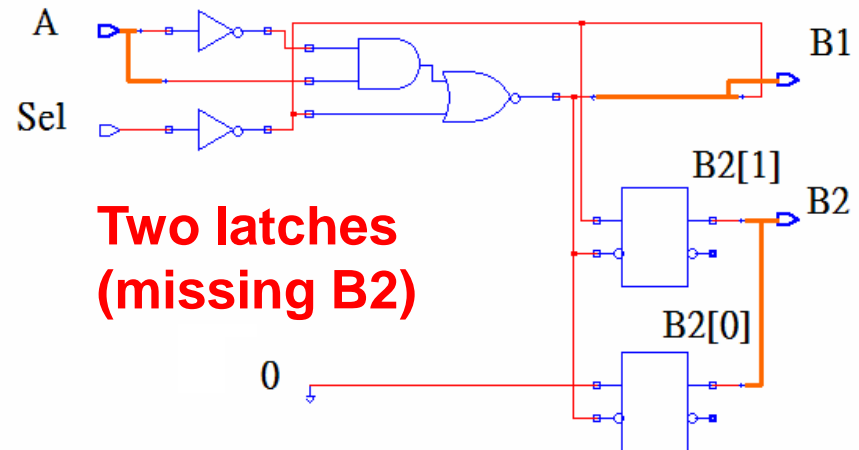


# Watch Out for Unintentional Latches (4/6)

```
module code3(Sel , A , B1, B2);  
input Sel, [1:0]A;  
output [1:0] B1, B2; reg [1:0] B1,B2;  
always @ (Sel or A)  
if(Sel)  
    if(A == 1)  
        begin B1 = 0; B2 = 0; end  
    else  
        begin B1 = 1; B2 = 1; end  
else  
    begin B1 = 2; B2 = 2; end  
endmodule
```



```
module code4(Sel , A , B1, B2);  
input Sel, input [1:0]A;  
output [1:0] B1, B2; reg [1:0] B1, B2;  
always @ (Sel or A)  
if(Sel)  
    if(A == 1)  
        begin B1 = 0; B2 = 0; end  
    else  
        begin B1 = 1; end  
else  
    begin B1 = 2; B2 = 2; end  
endmodule
```



# Watch Out for Unintentional Latches (5/6)

```
module code2(Sel , A , B1, B2);
```

```
  input Sel;
```

```
  input [1:0]A;
```

```
  output [1:0] B1, B2;
```

```
  reg [1:0] B1, B2;
```

```
  always @ (Sel or A)
```

```
  if(Sel)
```

```
    if(A == 1)
```

```
    begin
```

```
      B1 = 0;
```

```
      B2 = 0;
```

```
    end
```

```
  else
```

```
  begin
```

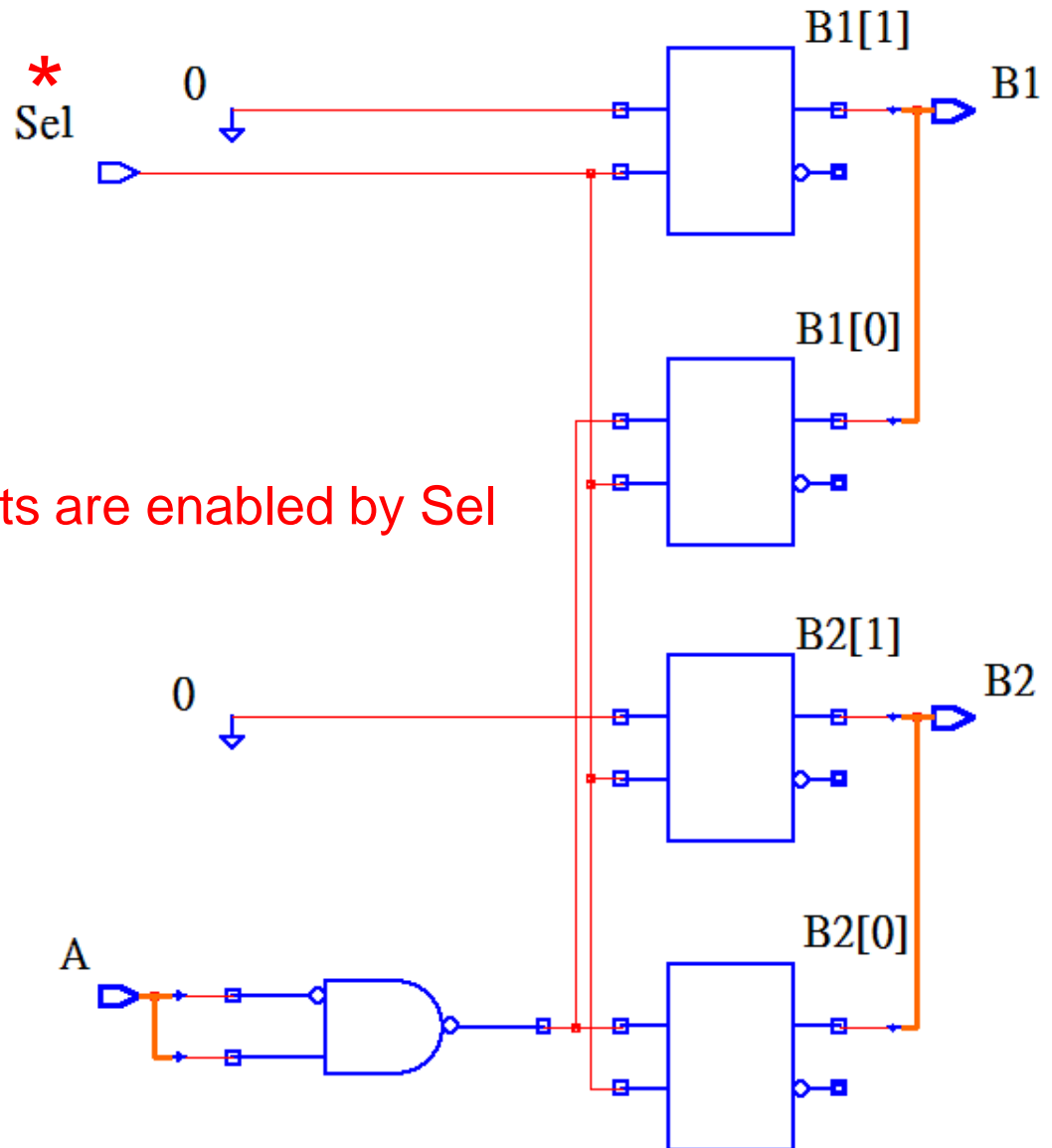
```
    B1 = 1;
```

```
    B2 = 1;
```

```
  end
```

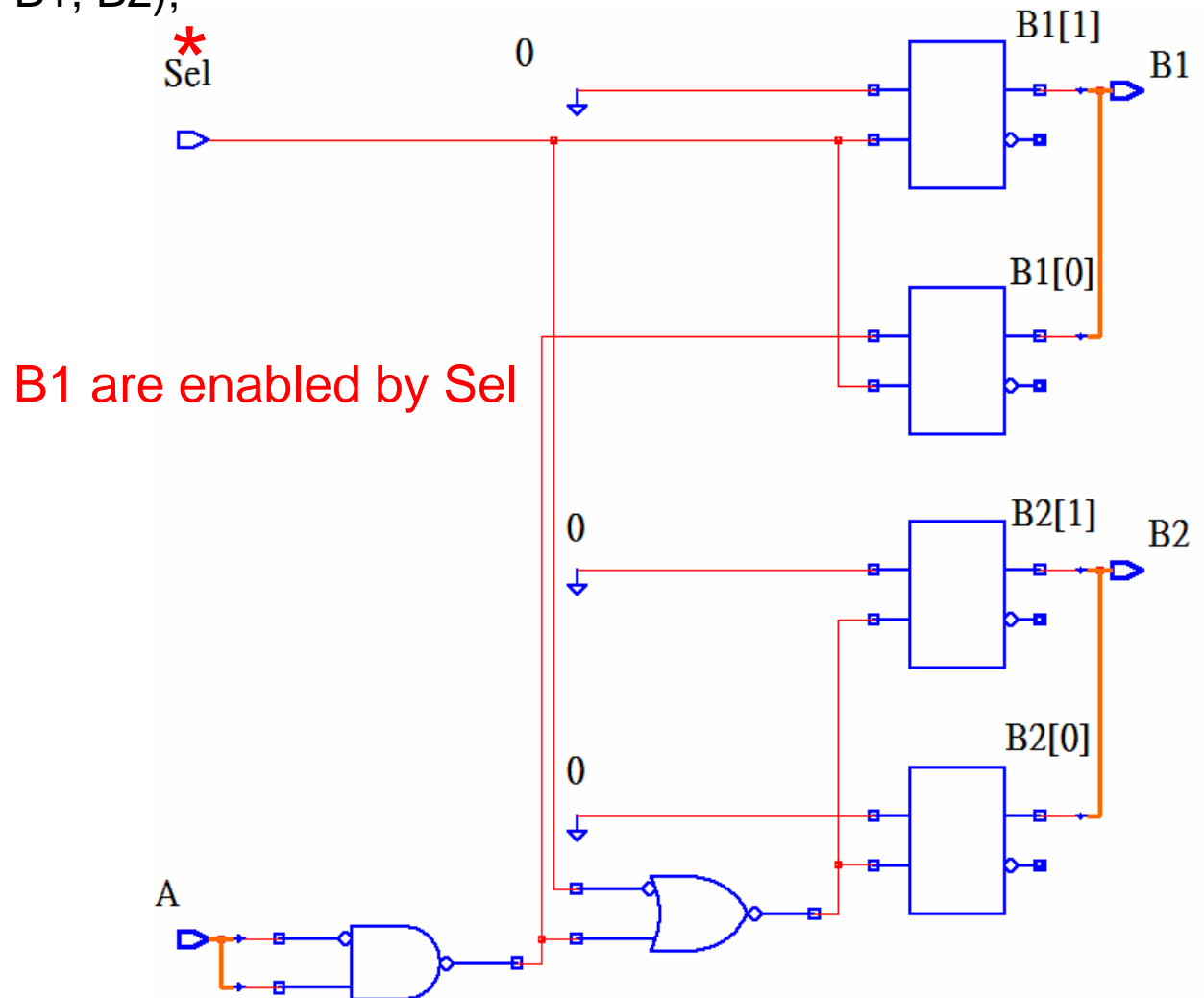
```
endmodule
```

Outputs are enabled by Sel



# Watch Out for Unintentional Latches (6/6)

```
module code1(Sel , A , B1, B2);  
input Sel, [1:0]A;  
output [1:0] B1, B2;  
reg [1:0] B1, B2;  
always @ (Sel or A)  
begin  
    if(Sel)  
        begin  
            if(A == 1)  
                begin  
                    B1= 0; B2 = 0;  
                end  
            else  
                begin  
                    B1 = 1;  
                end  
            end  
        end  
    end  
end  
endmodule
```





# Blocking vs. Non-Blocking (1/10)

- **Blocking** assignment ( **=** ) are order sensitive
- **Non-Blocking** assignment ( **<=** ) are order independent

**Blocking  
assignment**

Initial  
begin

```
a=#12 1;  
b=#3  0;  
c=#2  3;  
end
```

**Non-Blocking  
assignment**

Initial  
begin

```
d<=#12 1;  
e<=#3  0;  
f<=#2  3;  
end
```

Time-unit	a	b	c	d	e	f
0	x	x	x	x	x	x
2	x	x	x	x	x	3
3	x	x	x	x	0	3
12	1	x	x	1	0	3
15	1	0	x	1	0	3
17	1	0	3	1	0	3



# Blocking vs. Non-Blocking (2/10)

## Blocking assignment

Initial  
begin

..

A=1;

B=0;

..

A=B; // B=0 is used

B=A; // A=0 is used

Initial  
begin

..

A=1;

B=0;

..

B=A; // A=1 is used

A=B; // B=1 is used

---

## Non-Blocking assignment

Initial  
begin

..

A=1;

B=0;

..

A<=B; // B=0 is used

B<=A; // A=1 is used

Initial  
begin

..

A=1;

B=0;

..

B<=A; // A=1 is used

A<=B; // B=0 is used



# Blocking vs. Non-Blocking (3/10)

## Blocking assignment

```
module test_n(clk, a, b, c, out);  
input clk, a, b, c;  
output out;  
reg t1, t2;  
reg out;  
always @(posedge clk)  
begin  
    t1 = a&b;  
    t2 = t1&c;  
    out = t1 & t2;  
end  
endmodule
```

Blocking assignment

## Non-Blocking assignment

```
module test_n(clk, a, b, c, out);  
input clk, a, b, c;  
output out;  
reg t1, t2;  
reg out;  
always @(posedge clk)  
begin  
    t1 <= a&b;  
    t2 <= t1&c;  
    out <= t1 & t2;  
end  
endmodule
```

Non-blocking assignment

assigned in order

assigned immediately

# Blocking vs. Non-Blocking (4/10)

```
module test_n(a, b, c, d, t1, t2, out);  
input a, b, c, d;  
output out, t1, t2;  
reg t1, t2, out;
```

```
always @(a or b or c or d)  
begin
```

```
    t1 = a&b; Combinational  
    t2 = c | d; circuit  
    out = t1 & t2;
```

```
end  
endmodule
```

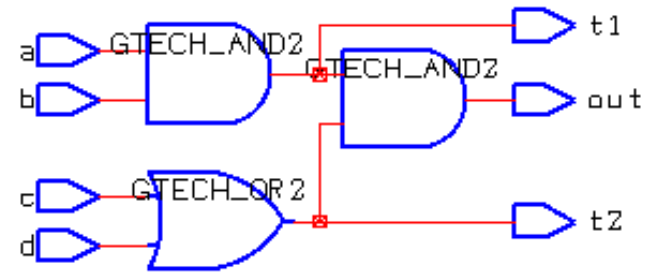
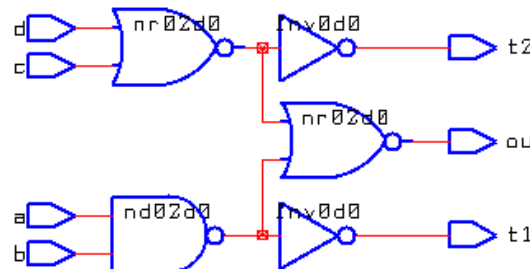
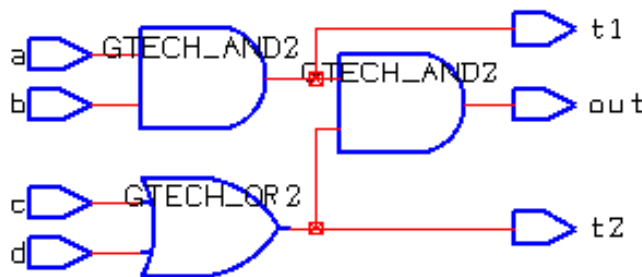
```
module test_n(a, b, c, d, t1, t2, out);  
input a, b, c, d;  
output out, t1, t2;  
reg t1, t2, out;
```

```
always @(a or b or c or d)  
begin
```

```
    t1 <= a&b; Combinational  
    t2 <= c | d; circuit  
    out <= t1 & t2;
```

```
end  
endmodule
```

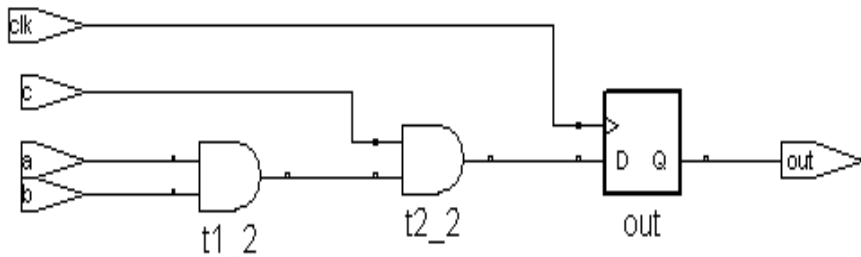
**Automatic  
optimization (DC)**



# Blocking vs. Non-Blocking (5/10)

## Blocking assignment

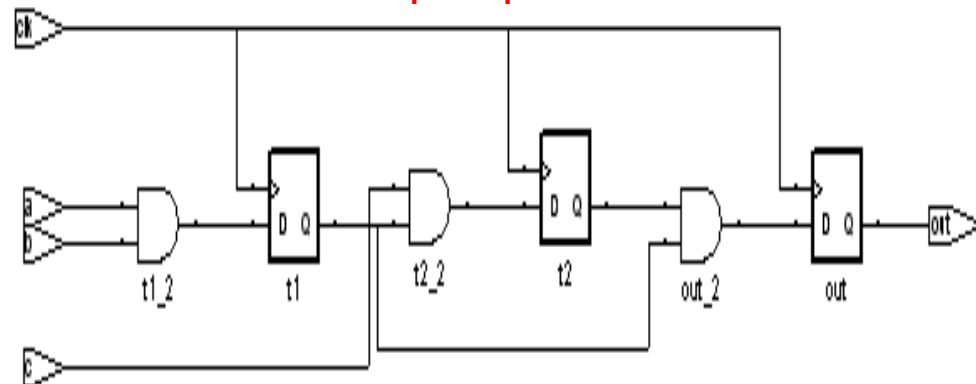
```
module test_n(clk, a, b, c, out);  
input clk, a, b, c;  
output out;  
reg t1, t2;  
reg out;  
always @(posedge clk)  
begin  
    t1 = a&b;           ①  
    t2 = t1&c;          ②  
    out = t1 & t2;      ③  
end endmodule
```



## Non-blocking assignment

```
module test_n(clk, a, b, c, out);  
input clk, a, b, c; output out;  
reg t1, t2; reg out;  
always @(posedge clk)  
begin  
    t1 <= a&b;           ①  
    t2 <= t1&c;          ① // old t1 is used  
    out <= t1 & t2;      ① // old t1 and t2  
                           are used  
end  
endmodule
```

## Three flip-flops are inferred

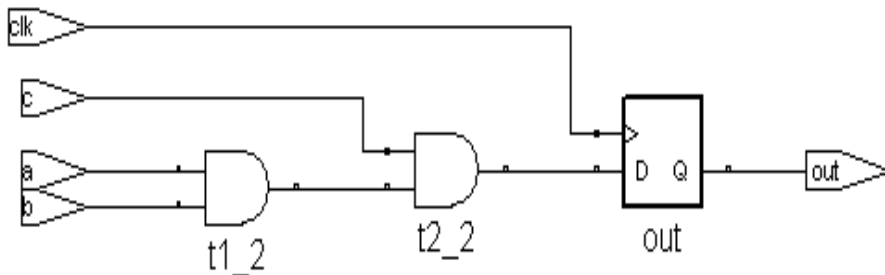




# Blocking vs. Non-Blocking (6/10)

## Blocking assignment

```
module test_n(clk, a, b, c, out);  
input clk, a, b, c;  
output out;  
reg t1, t2;  
reg out;  
always @(posedge clk)  
begin  
    t1 = a&b;           ①  
    t2 = t1&c;          ②  
    out = t1 & t2;      ③  
end  
endmodule
```

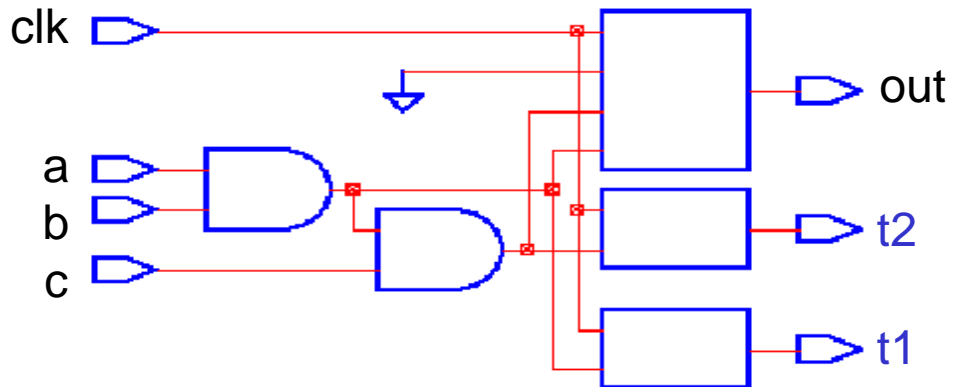


## Blocking assignment

```
module test_n(clk, a, b, c, t1, t2, out);  
input clk, a, b, c; output out, t1, t2;  
reg t1, t2; reg out;  
  
always @(posedge clk)  
begin
```

```
    t1 = a&b;           ①  
    t2 = t1&c;          ② // new t1 is used  
    out = t1 & t2;      ③ // new t1 and t2  
                        are used
```

```
end  
endmodule
```



# Blocking vs. Non-Blocking (7/10)

## Blocking assignment

```
module rtl_1(clk, data, regb);  
input data, clk;  
output regb;  
reg rega, regb;
```

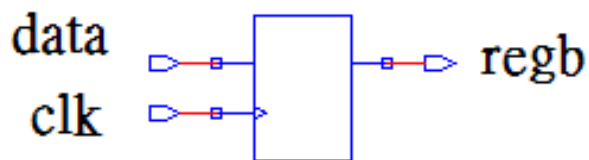
```
always @(posedge clk)  
begin
```

```
    rega = data; ①
```

```
    regb = rega; ②
```

```
end
```

```
endmodule
```



## Non-blocking assignment

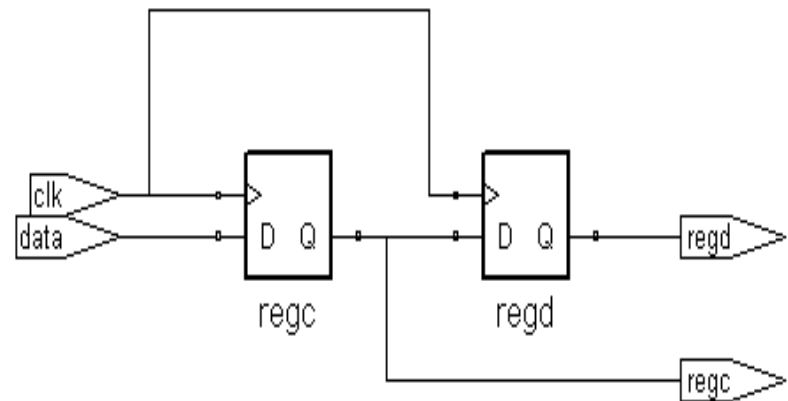
```
module rtl(clk, data, regc, regd);  
input data, clk;  
output regc, regd;  
reg regc, regd;
```

```
always @(posedge clk)  
begin
```

```
    regc <= data; ①
```

```
    regd <= regc; ① // old regc is used  
end
```

```
endmodule
```



# Blocking vs. Non-Blocking (8/10)

## Blocking assignment

```
module rtl_1(clk, data, regb);  
input data, clk;  
output regb;  
reg rega, regb;
```

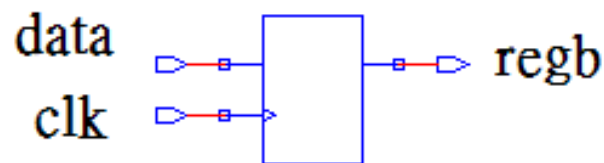
```
always @(posedge clk)  
begin
```

```
    rega = data; ①
```

```
    regb = rega; ②
```

```
end
```

```
endmodule
```



## Blocking assignment

```
module rtl_1(clk, data, regb);  
input data, clk;  
output regb;  
reg rega, regb;
```

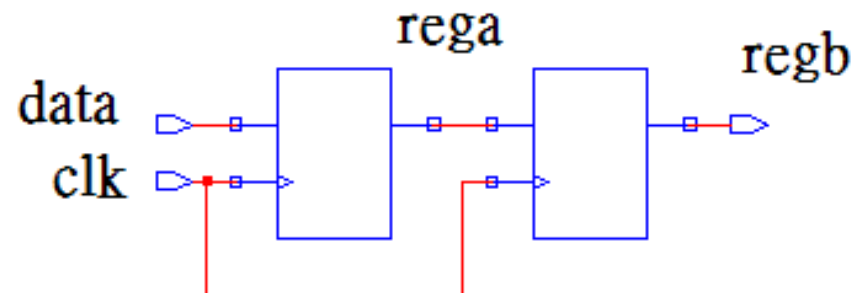
```
always @(posedge clk)  
begin
```

```
    regb = rega; ①
```

```
    rega = data; ②
```

```
end
```

```
endmodule
```



# Blocking vs. Non-Blocking (9/10)

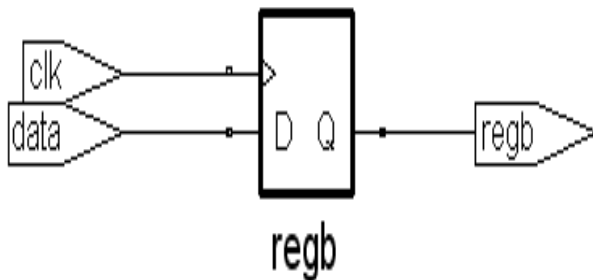
## Blocking assignment

```
module rtl_1(clk, data, regb);  
input data, clk;  
output regb;  
reg rega, regb;
```

```
always @(posedge clk)  
begin  
    rega = data; ①  
    regb = rega; ②  
end
```

endmodule

## One flip-flop is inferred



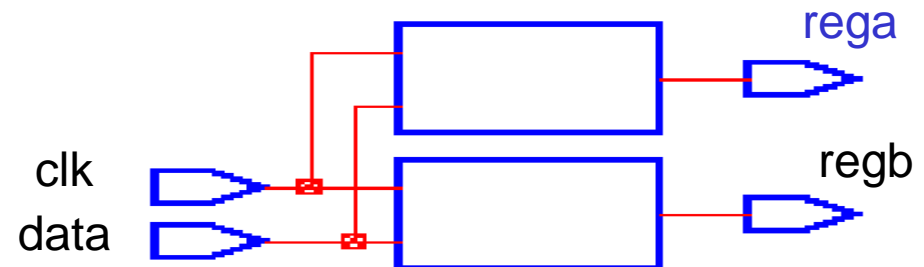
## Blocking assignment

```
module rtl_1(clk, data, rega, regb);  
input data, clk;  
output rega, regb;  
reg rega, regb;
```

```
always @(posedge clk)  
begin  
    rega = data; ①  
    regb = rega; ② // new regc is used  
end
```

endmodule

## Two flip-flops are inferred



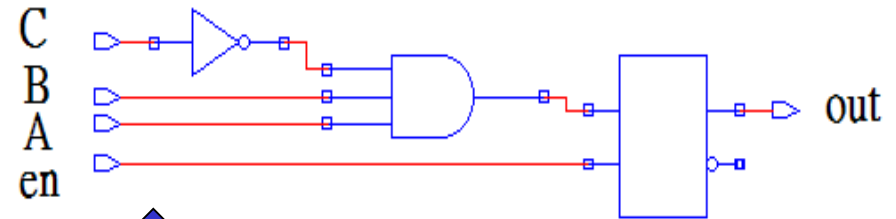
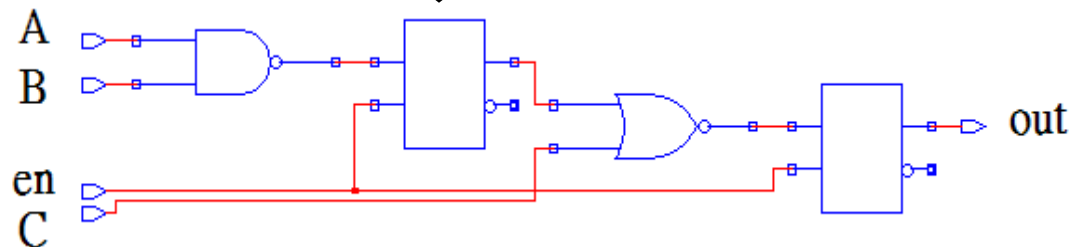
# Blocking vs. Non-Blocking (10/10)

```
module latch_if2(en, A, B, C, out);  
  input en, A, B, C;  
  output out;  
  reg K, out;
```

```
  always @(en or A or B or C)  
  if(en)  
  begin
```

```
    K <=!(A&B);  
    out <=!(K|C);
```

```
  end  
endmodule
```



One latch is inferred

```
module latch_if3(en,A,B,C,out);  
  input en, A, B, C;  
  output out;  
  reg K, out;
```

```
  always @(en or A or B or C)  
  if(en)
```

```
  begin  
    K =!(A&B);  
    out =!(K|C);
```

```
  end  
endmodule
```



# Combinational Shifter (1/2)

```
module SHIFTER (Sel, A,Y);  
input [1:0]Sel;  
input [5:0]A;  
output [5:0]Y;  
  
reg [5:0]Y;  
  
always@(Sel or A)  
begin  
    case(Sel)  
        0: Y=A;  
        1: Y=A<<1;  
        2: Y=A>>1;  
        default: Y=6'b0;  
    endcase  
end  
endmodule
```

Sel	Operation	Function
0	$Y \leftarrow A$	no shift
1	$Y \leftarrow \text{shl } A$	shift left
2	$Y \leftarrow \text{shr } A$	shift right
3	$Y \leftarrow 0$	zero outputs



# Combinational Shifter (2/2)

```
module SHIFTER_SHIFTINOUT
```

```
(Sel,ShiftLeftIn,ShiftRightIn,A,ShiftLeftOut,ShiftRightOut,Y);
```

```
    input [1:0]Sel;
```

```
    input ShiftLeftIn, ShiftRightIn;
```

```
    input [5:0]A; output [5:0]Y;
```

```
    output ShiftLeftOut,ShiftRightOut;
```

```
    reg ShiftLeftOut,ShiftRightOut;
```

```
    reg [5:0]Y; reg [7:0]A_Wide, Y_Wide;
```

```
    always@(Sel or ShiftLeftIn or  
            ShiftRightIn or A)
```

```
    begin
```

```
        A_Wide={ShiftLeftIn,A,ShiftRightIn};
```

```
        case(Sel)
```

```
            0: Y_Wide = A_Wide;
```

```
            1: Y_Wide = A_Wide<<1;
```

```
            2: Y_Wide = A_Wide>>1;
```

```
            3: Y_Wide = 8'b0;
```

```
        endcase
```

```
        ShiftLeftOut = Y_Wide[7];
```

```
        Y = Y_Wide[6:1];
```

```
        ShiftRightOut = Y_Wide[0];
```

```
    end
```

```
endmodule
```

Sel	Operation	Function
0	$Y \leftarrow A$ $\text{ShiftLeftOut} \leftarrow 0$ $\text{ShiftRightOut} \leftarrow 0$	no shift
1	$Y \leftarrow \text{shl } A$ $\text{ShiftLeftOut} \leftarrow A[5]$ $\text{ShiftRightOut} \leftarrow 0$	shift left
2	$Y \leftarrow \text{shr } A$ $\text{ShiftLeftOut} \leftarrow 0$ $\text{ShiftRightOut} \leftarrow A[0]$	shift right
3	$Y \leftarrow 0$ $\text{ShiftLeftOut} \leftarrow 0$ $\text{ShiftRightOut} \leftarrow 0$	zero outputs

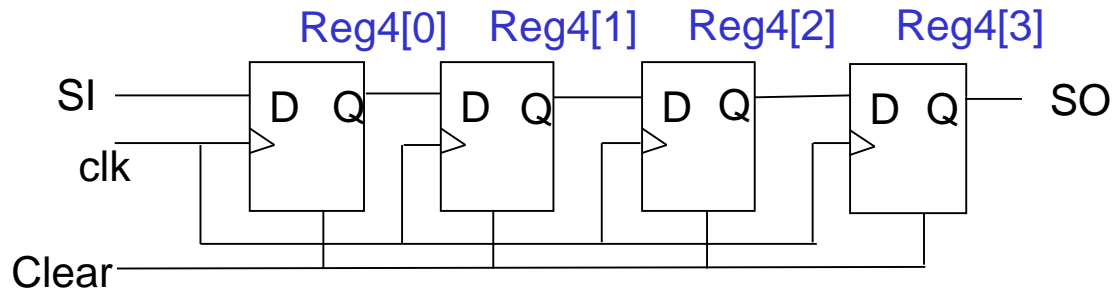
# SISO Shifter (1/4)

```
module SISO_SR(clk, Clear, SI, SO);  
input  clk, Clear, SI;  
output SO;  
reg    [3:0] Reg4;
```

**sequential shifter**

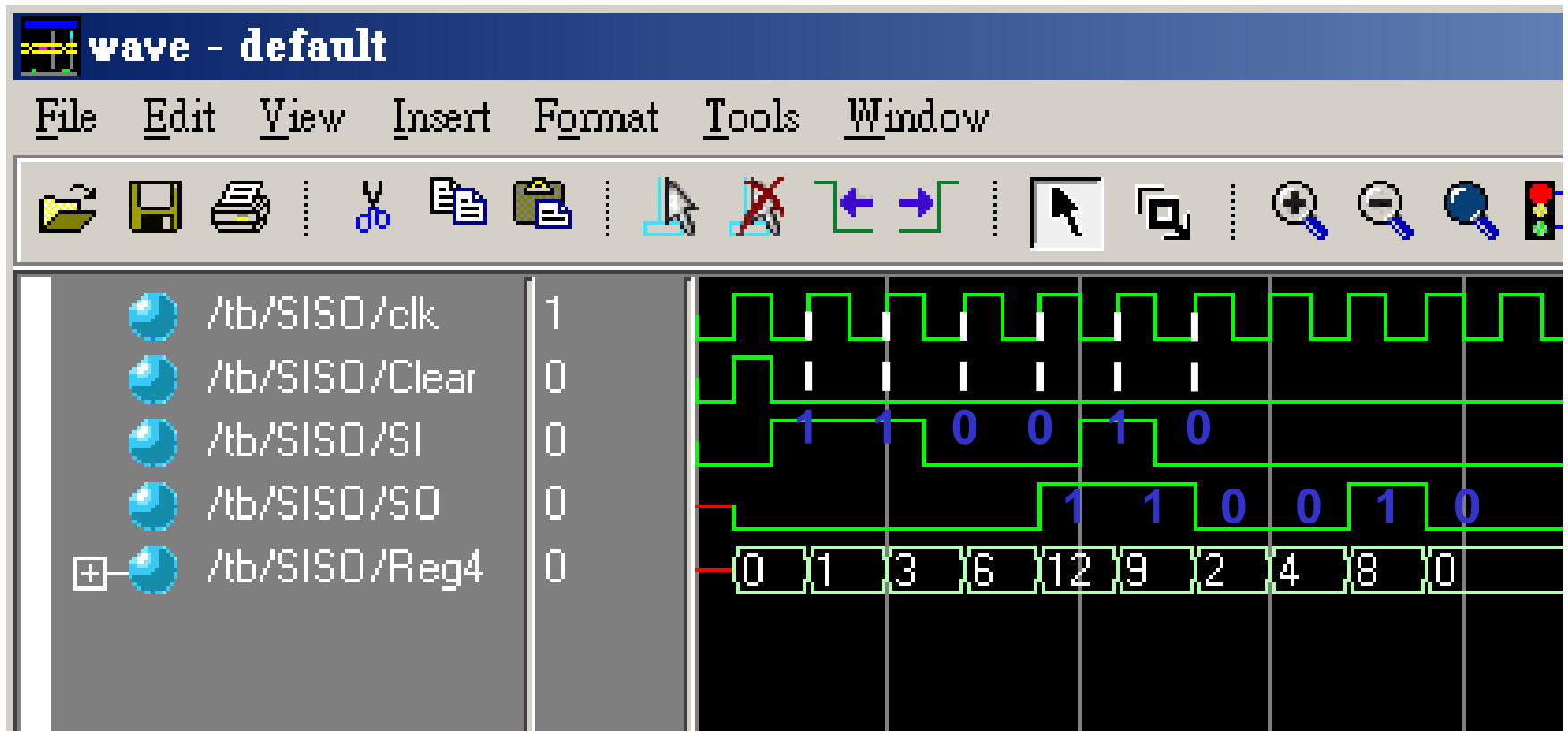
**serial in serial out**

```
always @(posedge clk or posedge Clear)  
begin  
    if (Clear)  
        Reg4 = 4'b0;  
    else begin  
        Reg4[3] = Reg4[2];  
        Reg4[2] = Reg4[1];  
        Reg4[1] = Reg4[0];  
        Reg4[0] = SI;  
    end  
end  
assign SO = Reg4[3];  
endmodule
```





# SISO Shifter (2/4)





# SISO Shifter (3/4)

```
module SISO_SR(clk, Clear, SI, SO);  
input  clk, Clear, SI;  
output SO;  
reg    [3:0] Reg4;
```

```
always @(posedge clk or posedge Clear)
```

```
begin : for_Local
```

```
integer i;
```

```
if (Clear)
```

```
    Reg4 = 4'b0;
```

```
else begin
```

```
    for (i = 3; i >= 1; i = i - 1)
```

```
        Reg4[i] = Reg4[i-1];
```

```
        Reg4[0] = SI;
```

```
    end
```

```
end
```

```
    assign SO = Reg4[3];
```

```
endmodule
```

```
Reg4[3] = Reg4[2];  
Reg4[2] = Reg4[1];  
Reg4[1] = Reg4[0];  
Reg4[0] = SI;
```



# SISO Shifter (4/4)

---

```
module SISO_SR (clk, Clear, SI, SO);
input  clk, Clear, SI;
output SO;
reg    [3:0] Reg4;

always @(posedge clk or posedge Clear)
begin
    if (Clear)
        Reg4 = 4'b0;
    else begin
        Reg4=Reg4<<1;
        Reg4[0] = SI;
    end
end

assign SO = Reg4[3];
endmodule
```



# SIPO Shifter (1/2)

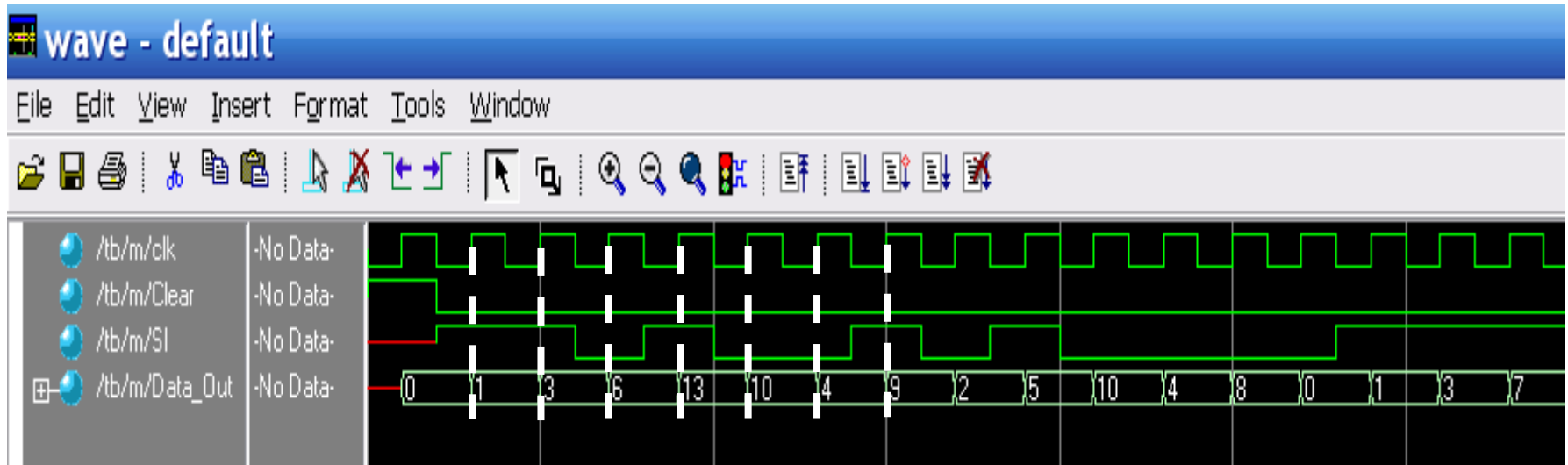
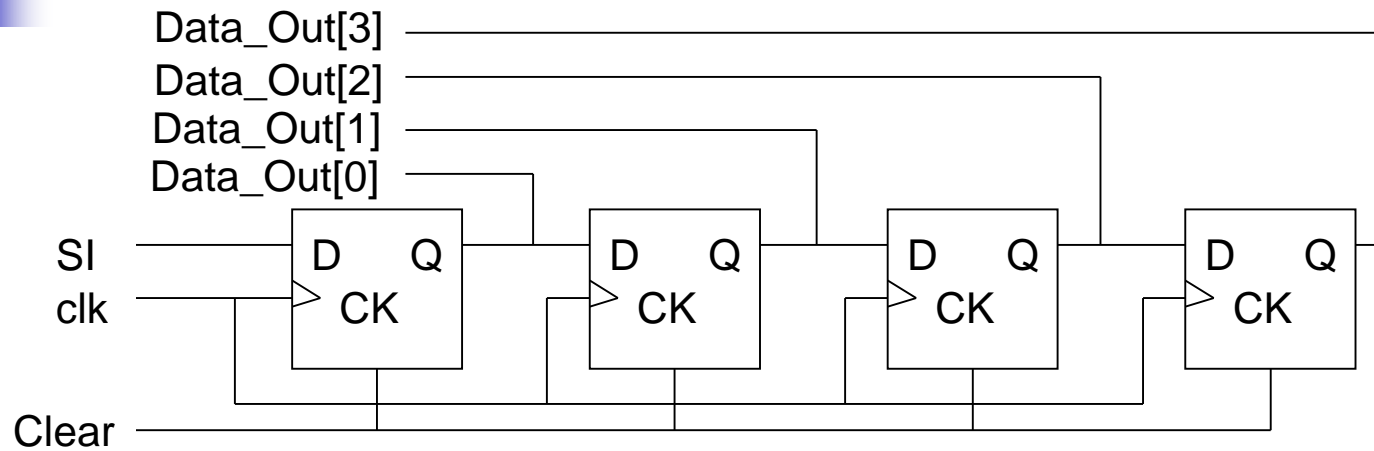
---

```
module SIPO__SR(clk, Clear, SI, Data_Out);  
input  clk, Clear, SI;  
output [3:0] Data_Out;  
reg    [3:0] Data_Out;
```

**serial in parallel out**

```
always @(posedge clk or posedge Clear)  
begin  
    if (Clear)  
        Data_Out = 4'b0;  
    else  
        begin  
            Data_Out = Data_Out << 1;  
            Data_Out[0] = SI;  
        end  
    end  
end  
endmodule
```

# SIPO Shifter (2/2)



1 1 0 1 0 0 1  
13  
10  
4



# PISO (1/2)

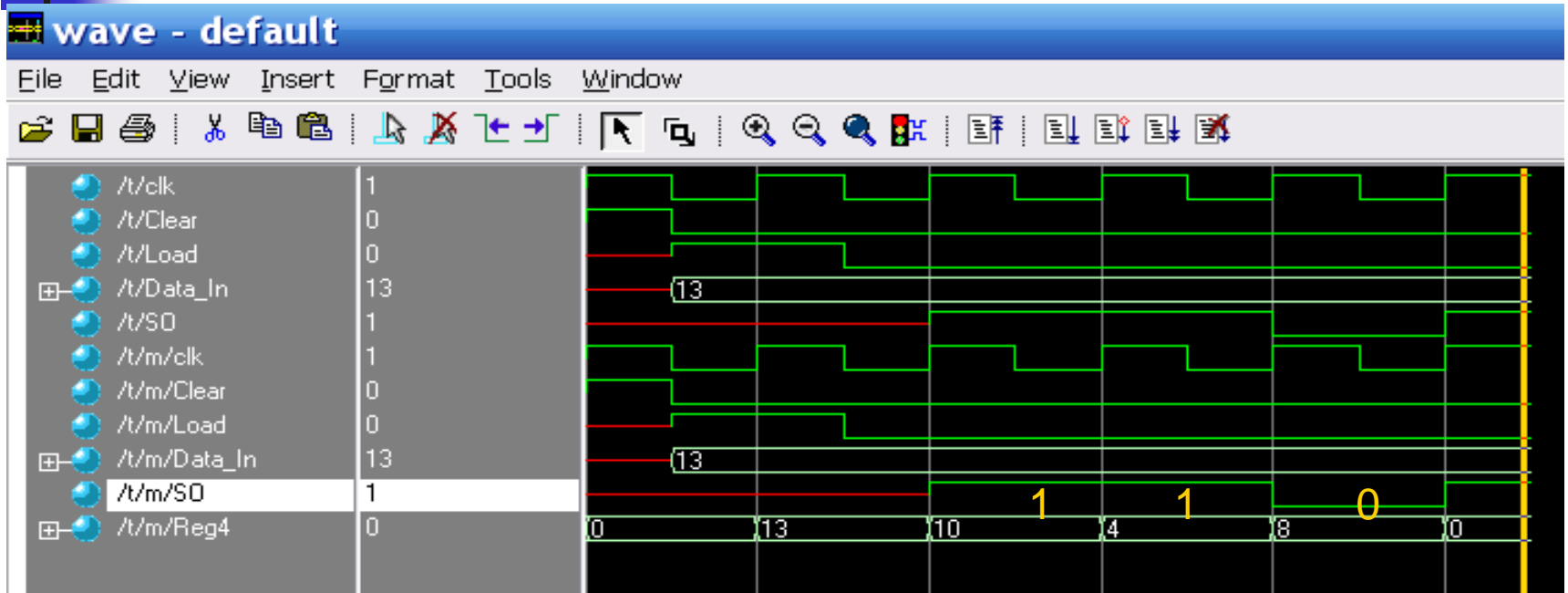
## parallel in serial out

```
module PISO__SR (clk, Clear, Load, Data_In, SO);
input      clk, Clear, Load;
input  [3:0] Data_In;
output      SO;
reg        SO;
reg  [3:0] Reg4;

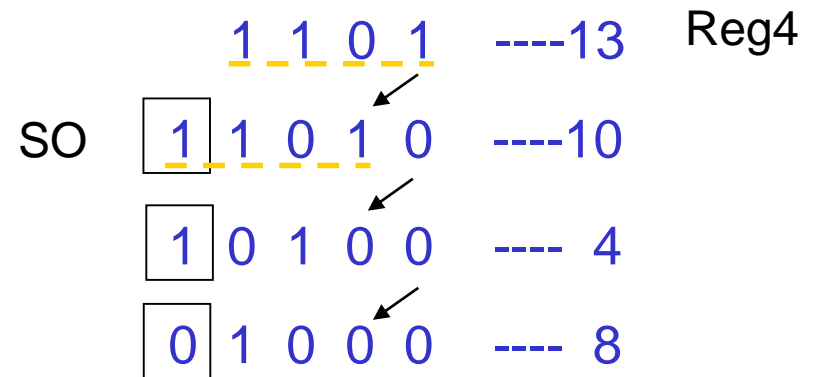
    always @(posedge clk)
    begin : for_Local
integer i;

        if (Clear)
            Reg4 = 4'b0;
        else
            if(Load)
                Reg4 = Data_In;
            else begin
                SO = Reg4[3];
                for (i = 3; i >= 1; i = i - 1)
                    Reg4[i] = Reg4[i-1];
                Reg4[0] = 0;
            end
        end
    end
endmodule
```

# PISO (2/2)



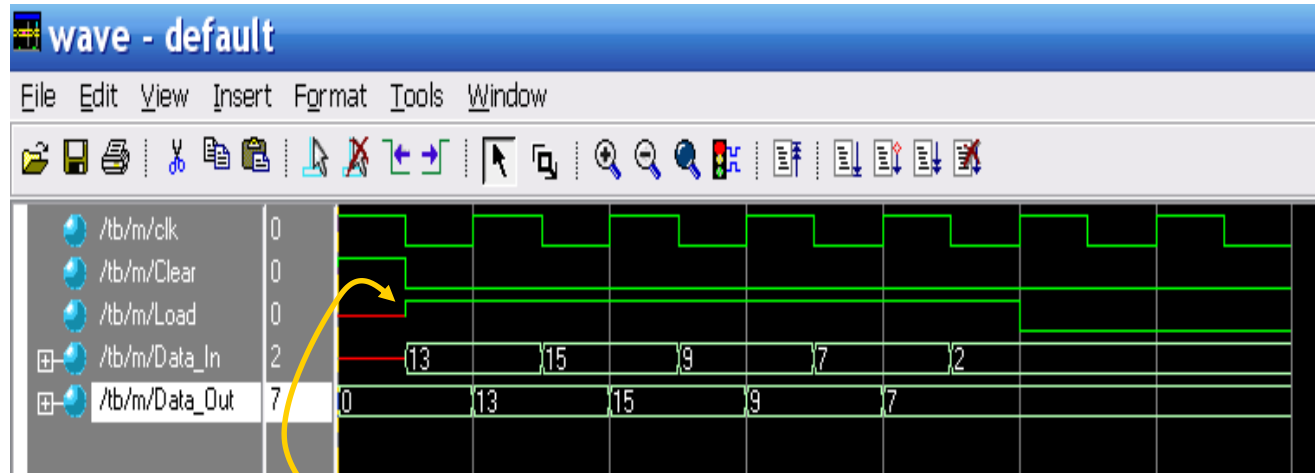
If Load=1, Data\_In=13



# PIPO

```
module PIPO__SR (clk, Clear, Load, Data_In, Data_Out);  
input  clk, Clear, Load;  
input  [3:0] Data_In; output  [3:0] Data_Out;  
reg    [3:0] Data_Out;  
always @(posedge clk)  
begin  
    if (Clear)  
        Data_Out = 4'b0;  
    else  
        begin  
            if (Load)  
                Data_Out = Data_In;  
        end  
    end  
end  
endmodule
```

**parallel in parallel out**



**Bad load !! Be Careful**

**Data\_In must be ready before posedge**



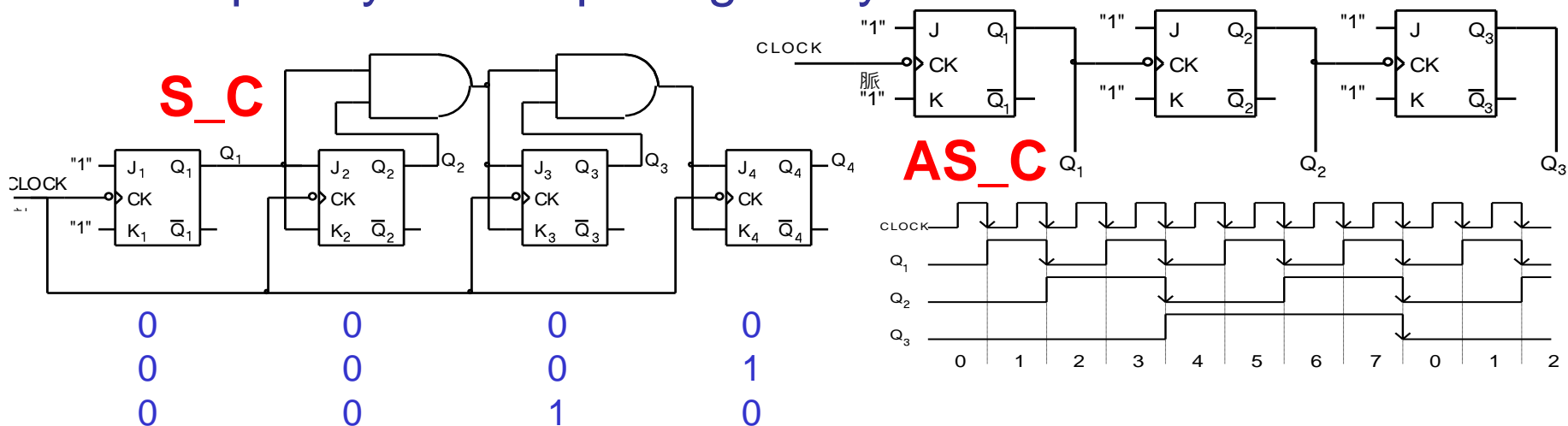
# Synchronous/Asynchronous Counter

## Synchronous counter:

All flip-flops in a synchronous counter receive the same clock pulse and so change state simultaneously.

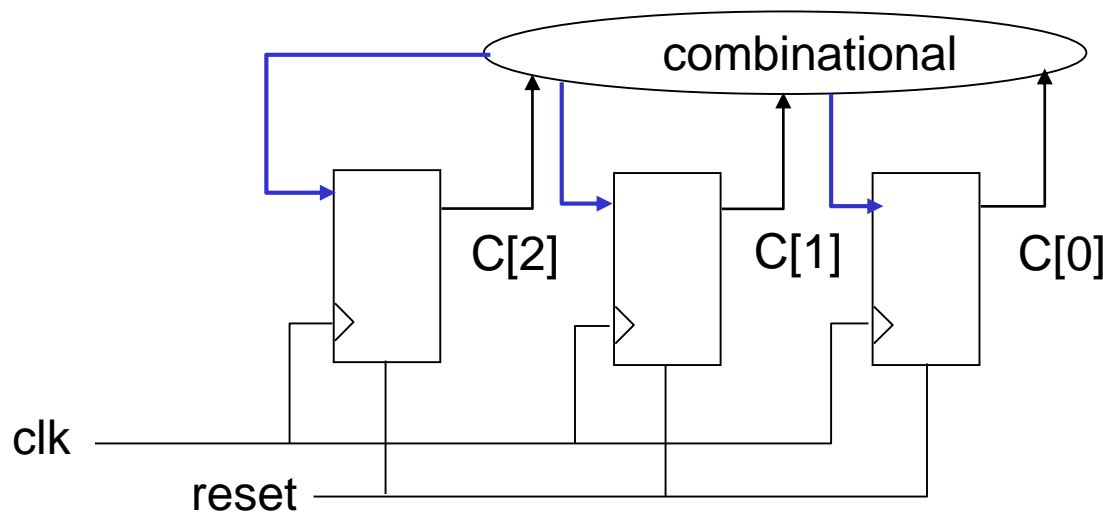
## Asynchronous (Ripple) counter:

Flip-flops transitions ripple through from one flip-flop to the next in sequence until all flip-flops reach a new stable value (state). Each single flip-flop stage divides the frequency of its input signal by two.



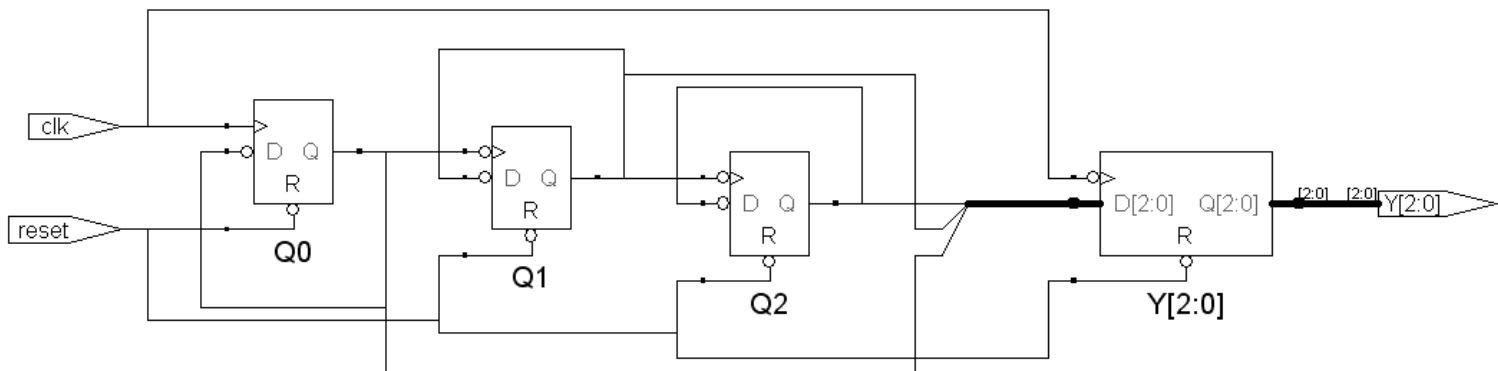
# Counter Implementation

## Synchronous counter



C(old)	C(new)
0 0 0	0 0 1
0 0 1	0 1 0
0 1 0	0 1 1
....	
1 1 0	1 1 1
1 1 1	0 0 0

## Asynchronous counter



# Synchronous Counter(1/6)

```
module Counter1(Reset, Enable,
    clk, Out);
    input    Reset, Enable, clk;
    output [2:0] Out;
    reg [2:0] Out;

    always @(posedge clk)
    begin
        if(Reset)
        begin
            Out = 3'b0;
        end
        else
```

```
        if(Enable == 1'b1)
```

```
        begin
```

```
            if(Out == 3'd7) ←
```

```
                Out = 3'b0;
```

```
            else
```

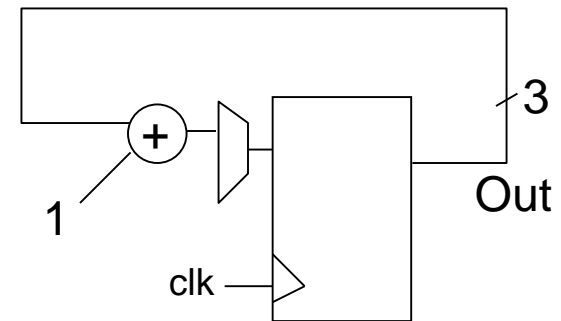
```
                Out = Out + 1'b1;
```

```
        end
```

```
    end
```

```
endmodule
```

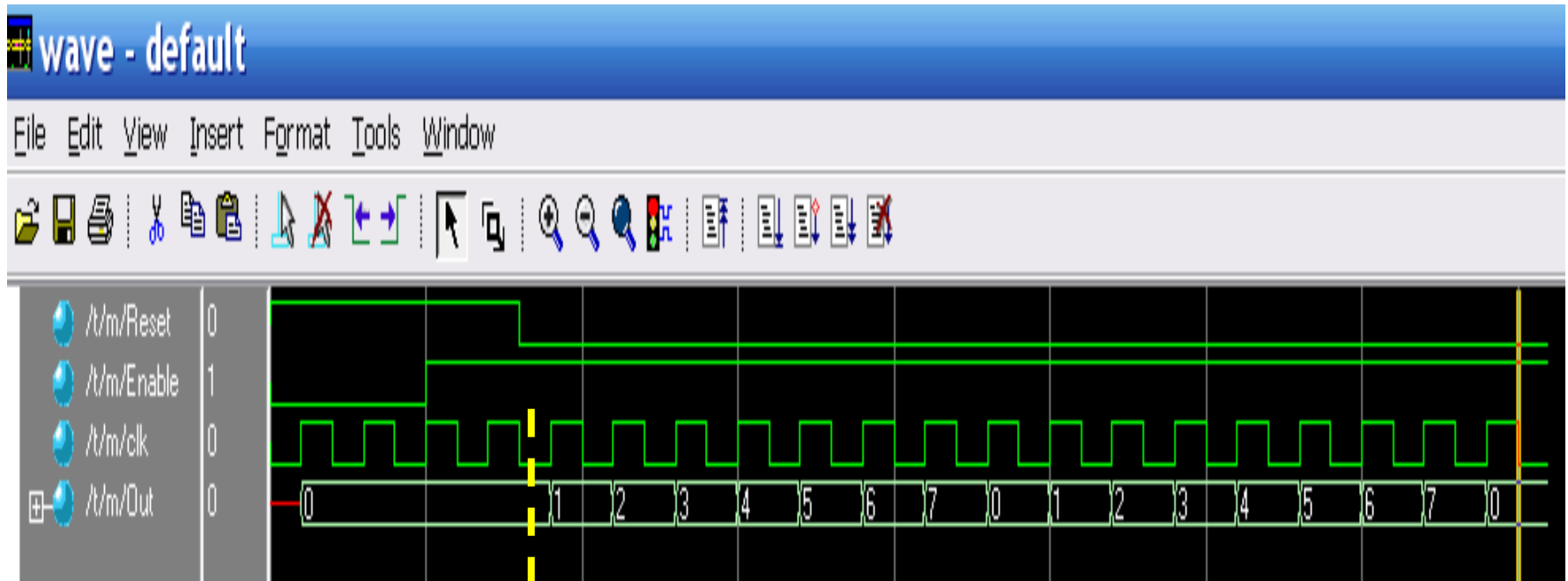
**What happens  
if Out== 3'd5 ??**



**Reset=1 Out=000**

**Reset=0, Enable==1, Out=0 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 0 → 1 → .....**

# Synchronous Counter(2/6)



Enable is active only when Reset is low. Then, the counter will begin count up.

# Synchronous Counter (3/6)

```
module Counter2 (clk, Reset, Load, Enable, Data_In, Out);
input      clk, Reset, Load, Enable;
input  [7:0] Data_In;
output [7:0] Out;
reg  [7:0] Out;
```

```
always @ (posedge clk)
```

```
begin
```

```
  if (Reset)
```

```
    Out = 0;
```

```
  else
```

```
    if (Load)
```

```
      Out = Data_In;
```

```
    else
```

```
      if (Enable)
```

```
        Out = Out + 1;
```

```
  end
```

```
endmodule
```

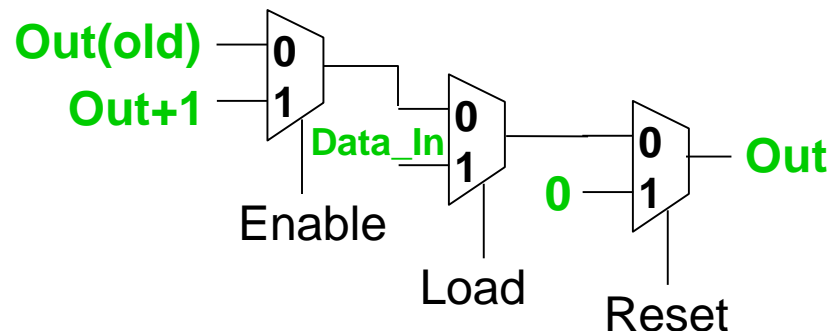
**Reset=1**

**Out=00000000**

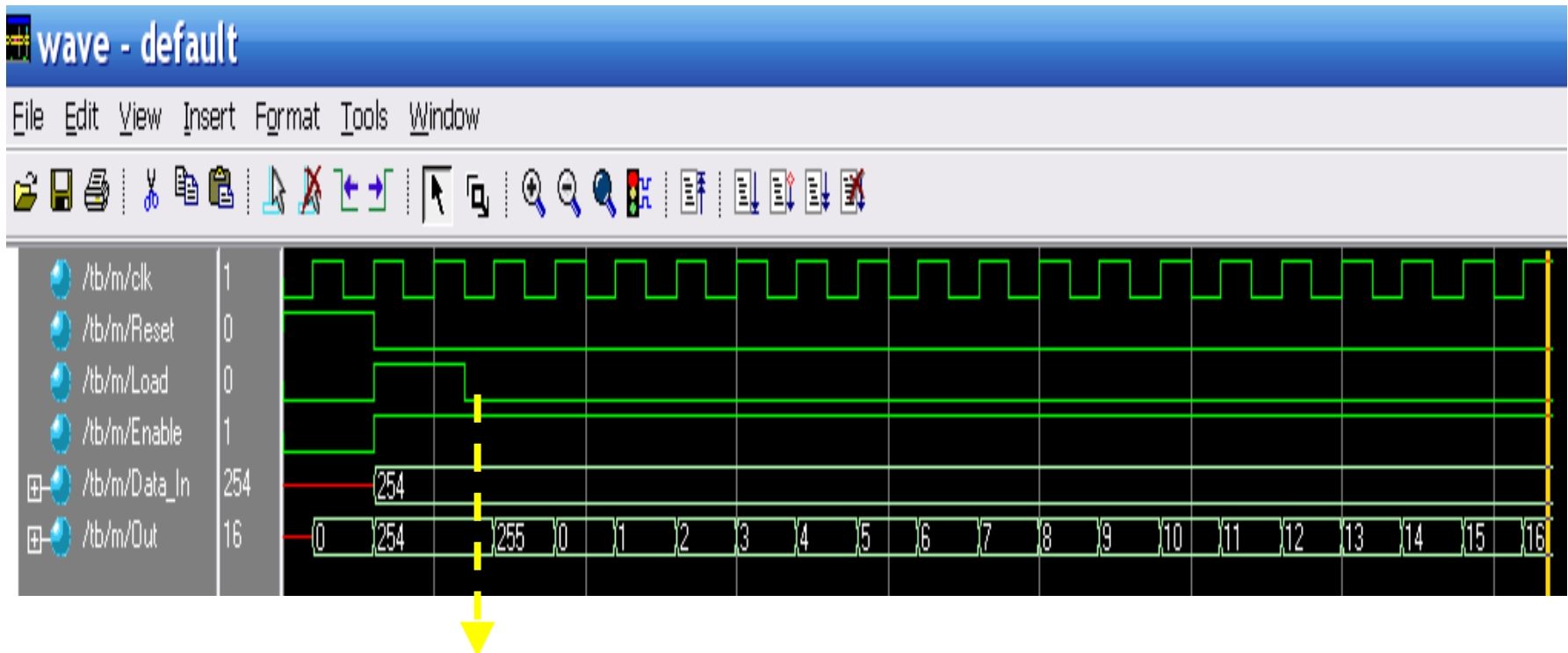
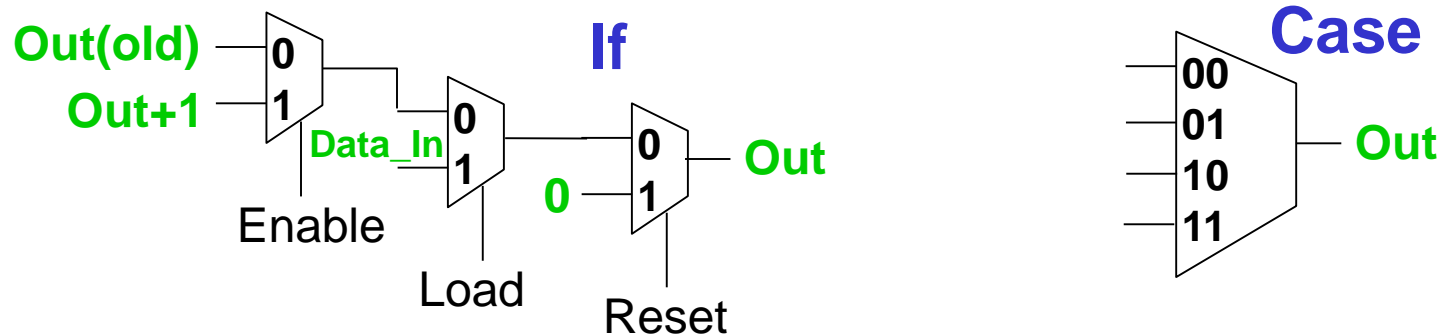
**Reset=0, Load=1, Out=Data\_In**

**Enable==1, Out=x → x+1 → ..... → 255 → 0**

**→ 1 → ..... → 255 → 0 → ...**



# Synchronous Counter(4/6)



Enable is active only when Load is low. Then, the counter will begin count up.



# Synchronous Counter(5/6)

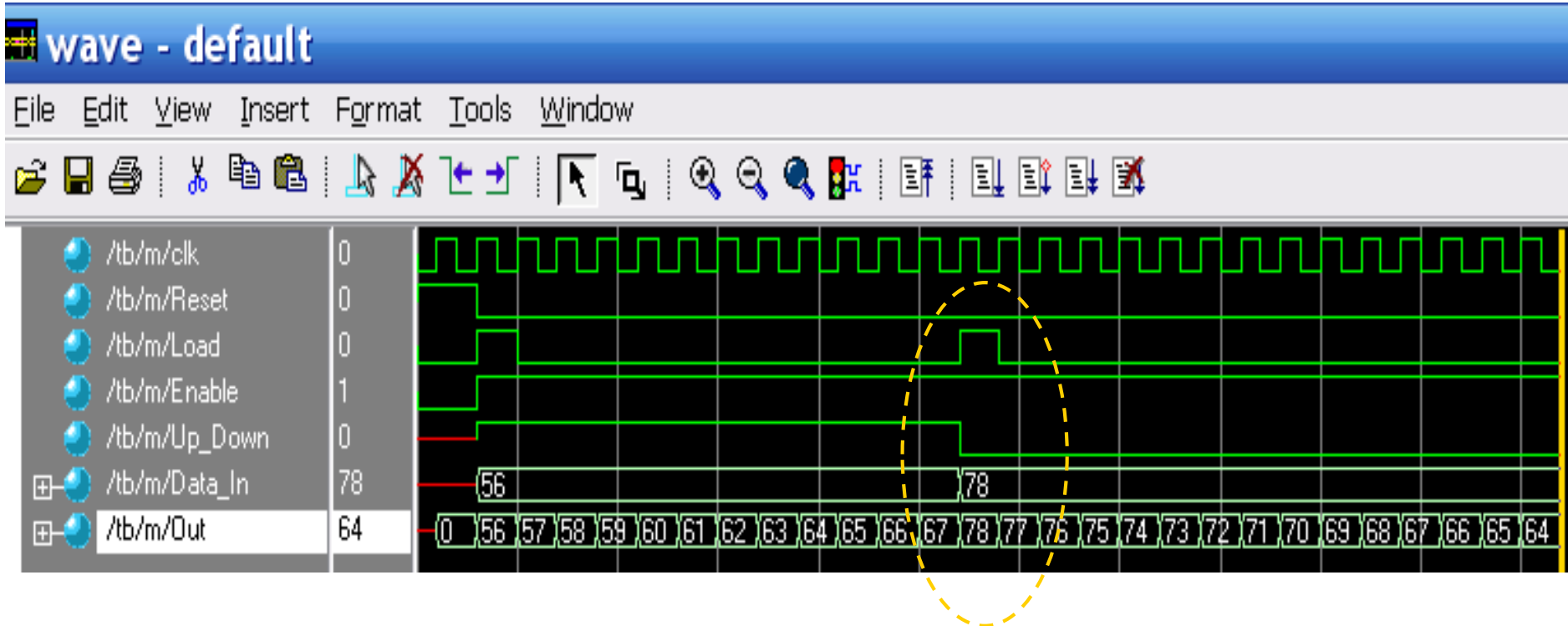
---

```
module Counter6 (clk, Reset, Load, Enable, Up_Down, Data_In,
    Out);
    input      clk, Reset, Load, Enable, Up_Down;
    input  [7:0] Data_In;
    output [7:0] Out;
    reg  [7:0] Out;

    always @ (posedge clk)
    begin
        if (Reset)
            Out = 0;
        else
            if (Load)
                Out = Data_In;
            else
                if (Enable)
                    begin
                        if (Up_Down)
                            Out = Out + 1;
                        else
                            Out = Out - 1;
                        end
                    end
                end
            endmodule

If down-by-two
Out=Out-2;
```

# Synchronous Counter(6/6)



Both Load and Up\_Down are not set properly.

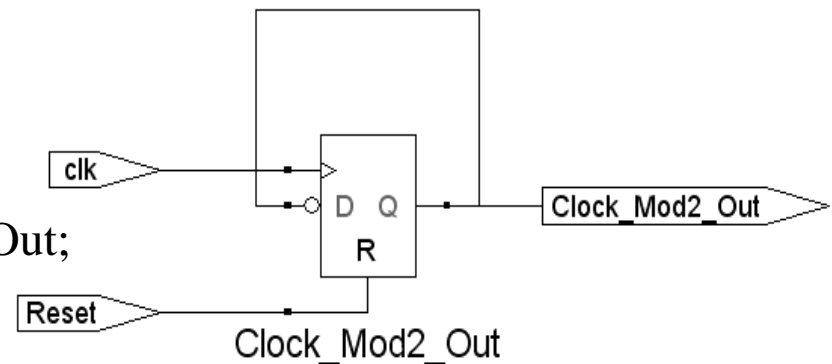


# Asynchronous Counter(1/8)

```
module FreqMod2 (Reset, clk_In, clk_Mod2_Out);  
input  Reset, clk_In; output  clk_Mod2_Out;  
reg    clk_Mod2_Out; wire    Not_clk_Mod2_Out;
```

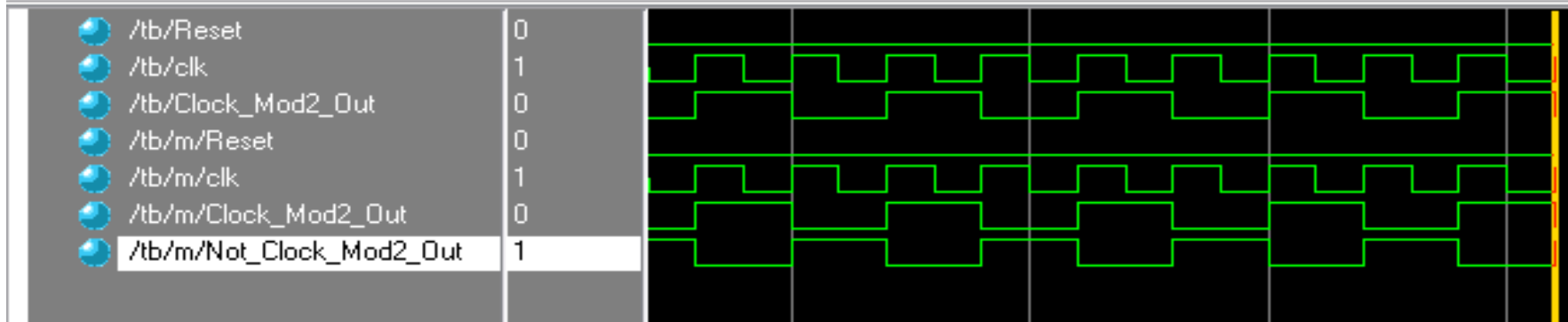
Each single flip-flop stage divides the frequency of its input signal by two.

```
assign Not_clk_Mod2_Out = !clk_Mod2_Out;  
always @(posedge Reset or posedge clk_In)  
begin  
    if (Reset)  clk_Mod2_Out = 0;  
    else        clk_Mod2_Out = Not_clk_Mod2_Out;  
end  
endmodule
```



wave - default

File Edit View Insert Format Tools Window





# Asynchronous Counter(2/8)

Divide by 16 clock divider using an asynchronous (ripple) counter

→ frequency divider

Count-down counter

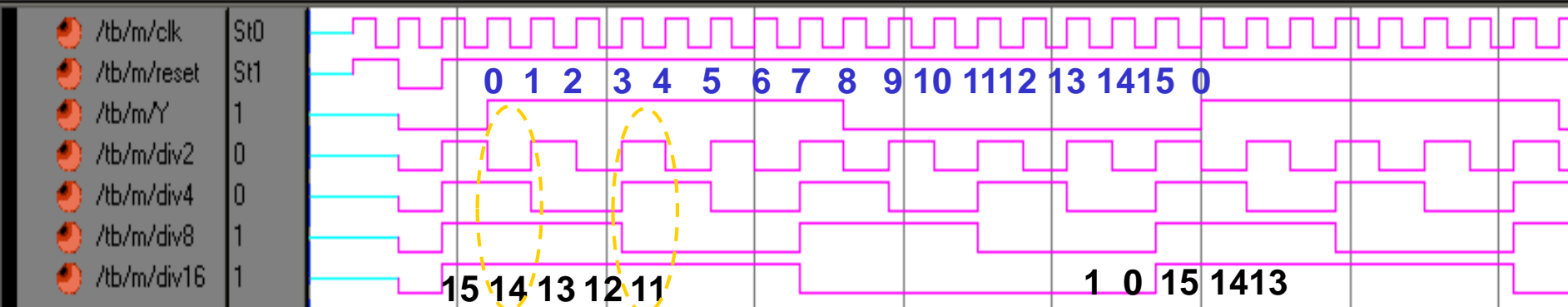
```
module CNT_ASYNC_CLK_DIV16(clk,reset,Y);
  input  clk,reset; output Y;
  reg div2,div4,div8,div16, Y;
  always@(posedge clk or negedge reset)
    if(!reset)
      div2=0;
    else
      div2=!div2;
  always@(posedge div2 or negedge reset)
    if(!reset)
      div4=0;
    else
      div4=!div4;
```

```
    always@(posedge div4 or
      negedge reset)
      if(!reset)
        div8=0;
      else
        div8=!div8;
    always@(posedge div8 or
      negedge reset)
      if(!reset)
        div16=0;
      else
        div16=!div16;
    always@(posedge clk or
      negedge reset)
      if(!reset)
        Y=0;
      else
        Y=div16;
  endmodule
```

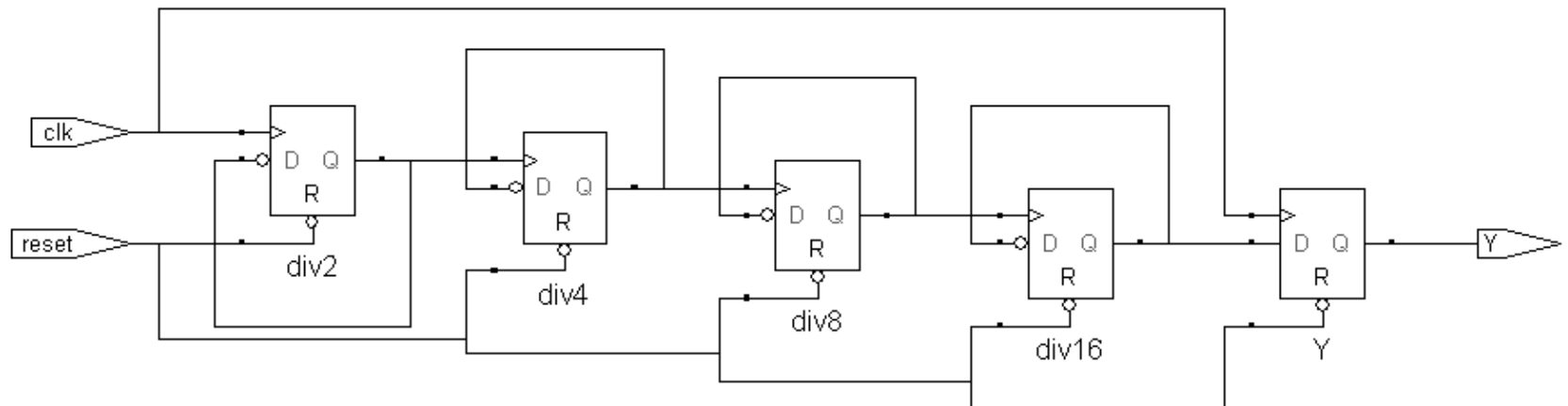
# Asynchronous Counter(3/8)

wave - default

File Edit View Insert Format Tools Window



Count-down counter 15→14→13→.....1→0→15→14→.....





# Asynchronous Counter(4/8)

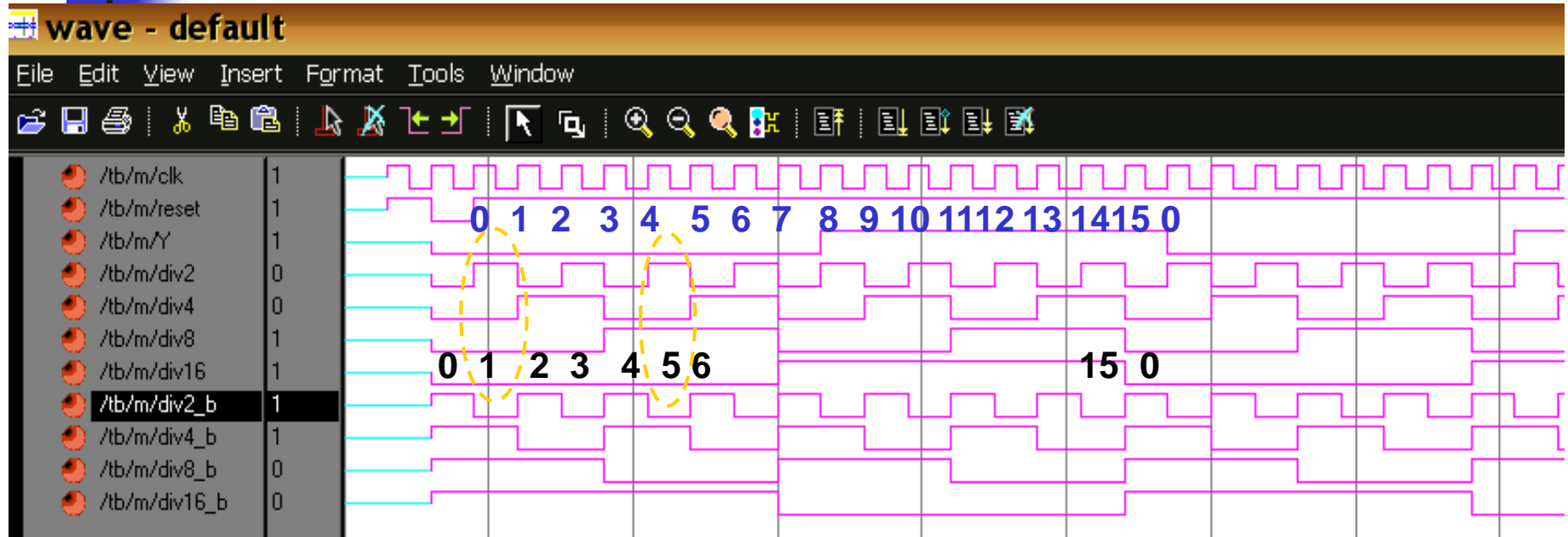
## Count-up counter

```
module DIV16(clk,reset,Y);
input  clk,reset;      output Y;
reg div2,div4,div8,div16, Y;

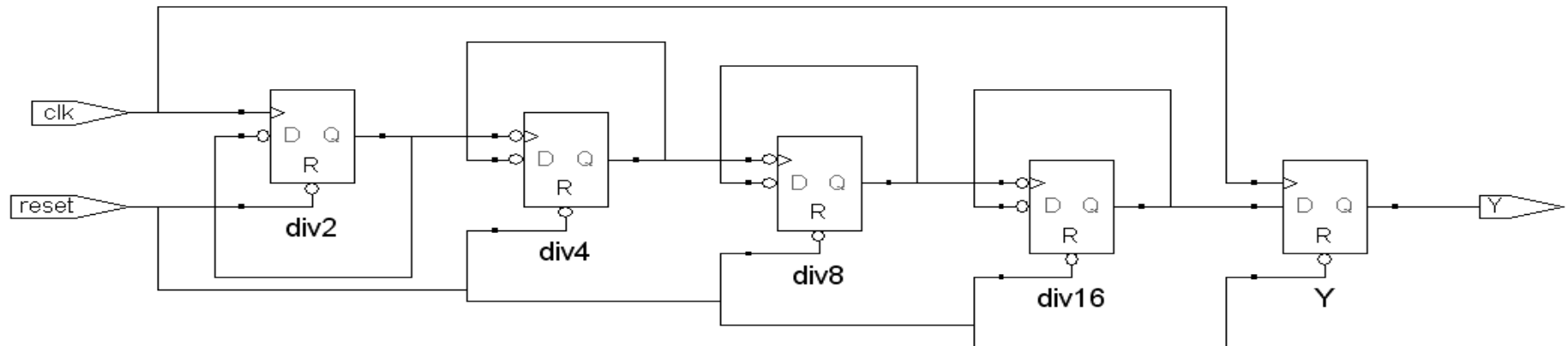
always@(posedge clk or negedge reset)
    if(!reset)
        div2=0;
    else
        div2=!div2;
    assign div2_b=!div2;
always@(posedge div2_b or negedge reset)
    if(!reset)
        div4=0;
    else
        div4=!div4;
    assign div4_b=!div4;
```

```
always@(posedge div4_b or
negedge reset)
    if(!reset)
        div8=0;
    else
        div8=!div8;
    assign div8_b=!div8;
always@(posedge div8_b or
negedge reset)
    if(!reset)
        div16=0;
    else
        div16=!div16;
always@(posedge clk or
negedge reset)
    if(!reset)
        Y=0;
    else
        Y=div16;
endmodule
```

# Asynchronous Counter(5/8)



Count-up counter  $0 \rightarrow 1 \rightarrow 2 \rightarrow \dots \rightarrow 15 \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow \dots$





# Asynchronous Counter(6/8)

## Divide by 13 clock divider using an asynchronous (ripple) counter

```
module CNT_ASYNC_CLK_DIV13(clk,reset,Y);
  input clk,reset; output Y;
  reg div2,div4,div8,div16,Y;
  wire div2_b,div4_b,div8_b,div16_b,clear;
  always@(posedge clk or negedge reset
    or posedge clear)
    if(!reset)
      div2=0;
    else if(clear)
      div2=0;
    else
      div2=!div2;

  assign div2_b=!div2;
```

```
always@(posedge div2 or negedge
  reset or posedge clear)
```

```
if(!reset)
```

```
div4=0;
```

```
else if(clear)
```

```
div4=0;
```

```
else
```

```
div4=!div4;
```

```
assign div4_b=!div4;
```

```
always@(posedge div4 or negedge
  reset or posedge clear)
```

```
if(!reset)
```

```
div8=0;
```

```
else if(clear)
```

```
div8=0;
```

```
else
```

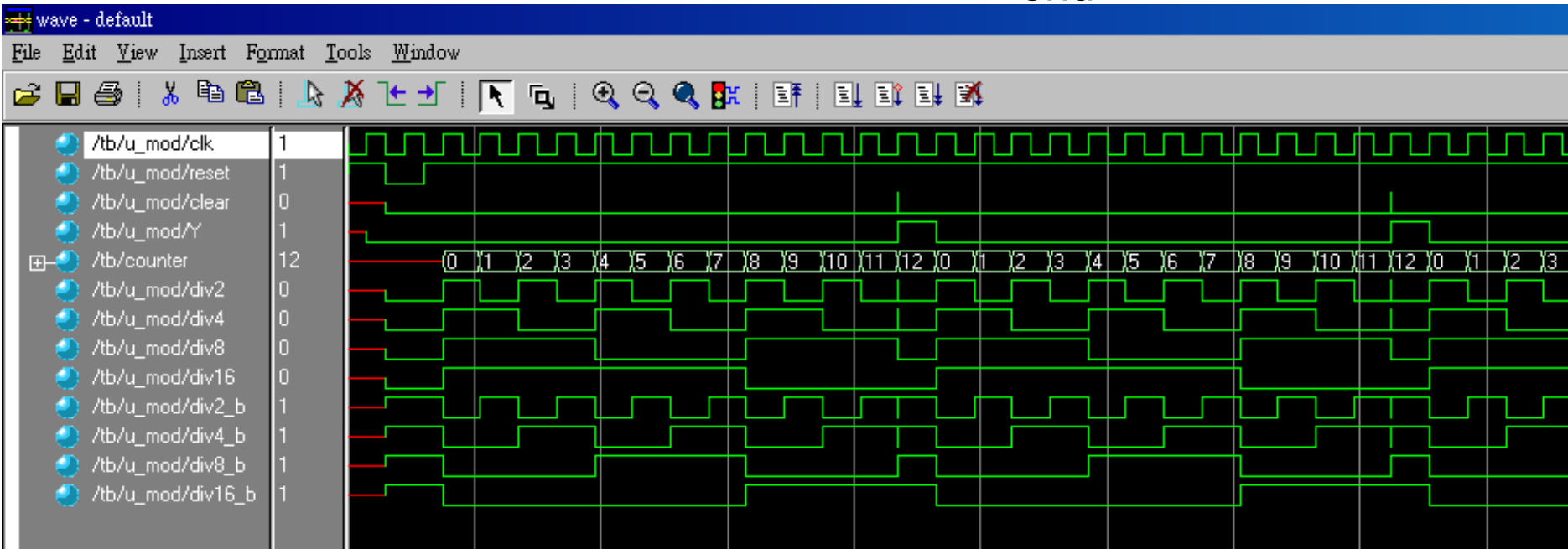
```
div8=!div8;
```

```
assign div8_b=!div8; ...
```

# Asynchronous Counter(7/8)

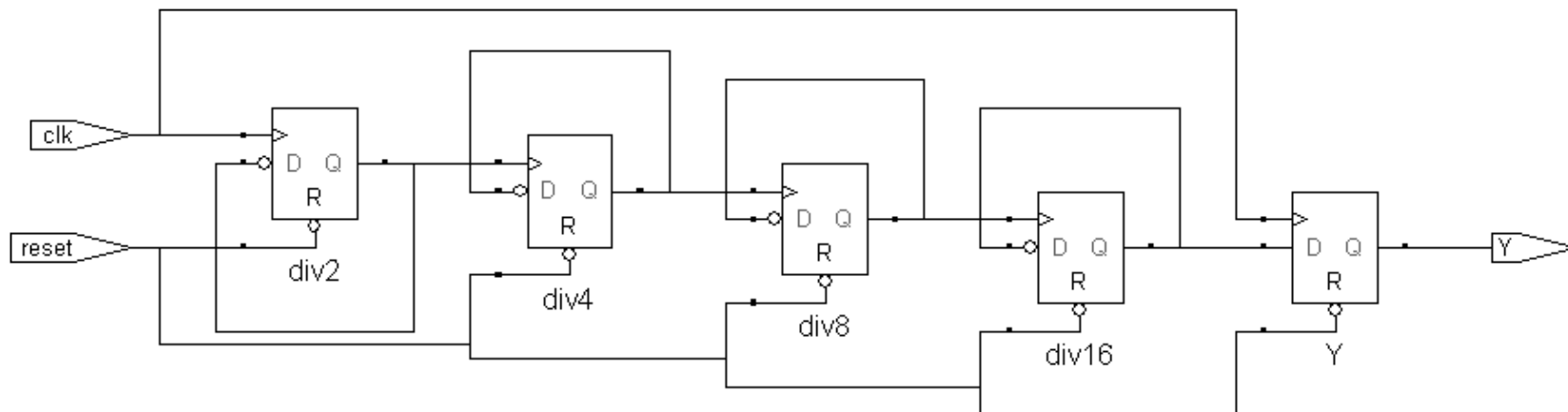
```
always@(posedge clk or negedge reset)
  if(!reset)
    Y=0;
  else if({div16_b,div8_b,div4_b,div2_b}==11)
    Y=1;
  else
    Y=0;
end
```

```
always@(div16_b or div8_b
        or div4_b or div2_b)
  begin
    if(({div16_b , div8_b ,
        div4_b , div2_b}==12))
      clear=1;
    else
      clear=0;
  end
```

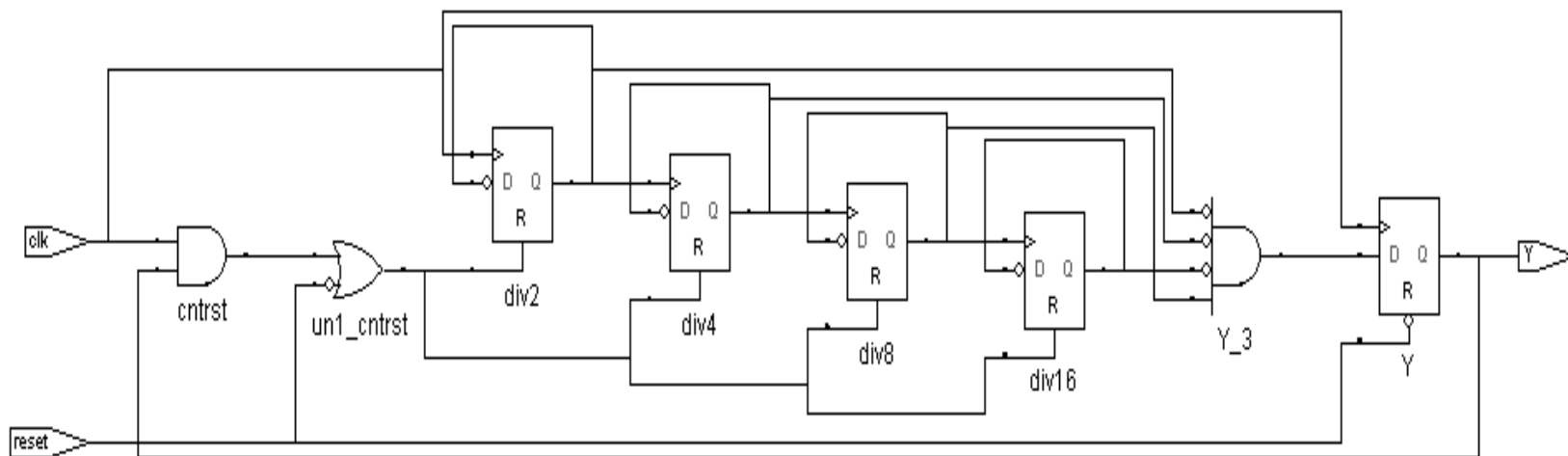


# Asynchronous Counter(8/8)

## Divide by 16



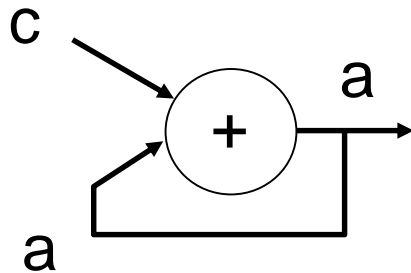
## Divide by 13





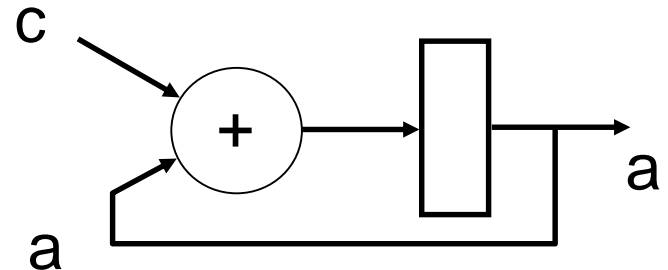
# Loop Problem (1/2)

assign a=a+c;



Error!

always @(posedge clk)  
a=a+c;

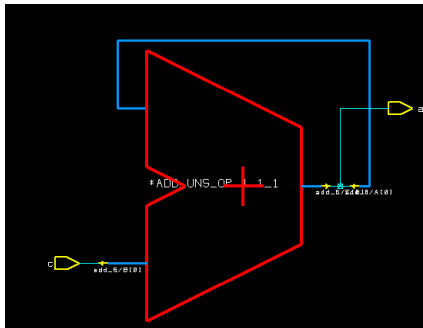
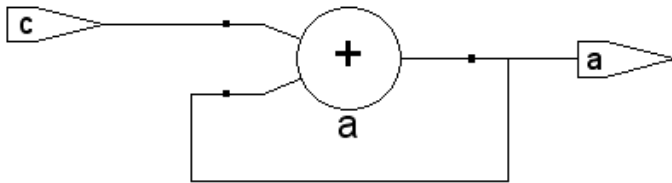


Good!

Why?

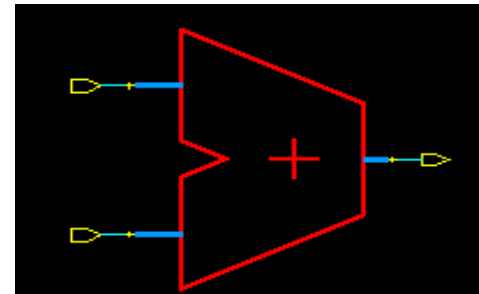
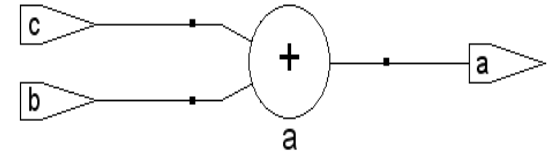
# Loop Problem (2/2)

```
module adder1(c,a);  
  input c;  
  output a;  
  assign a=a+c;  
endmodule
```



Error!

```
module adder2(c,b,a);  
  input c,b;  
  output a; reg a;  
  always@(a or c or b)  
  begin  
    a=b;  
    a=a+c;  
  end  
endmodule
```



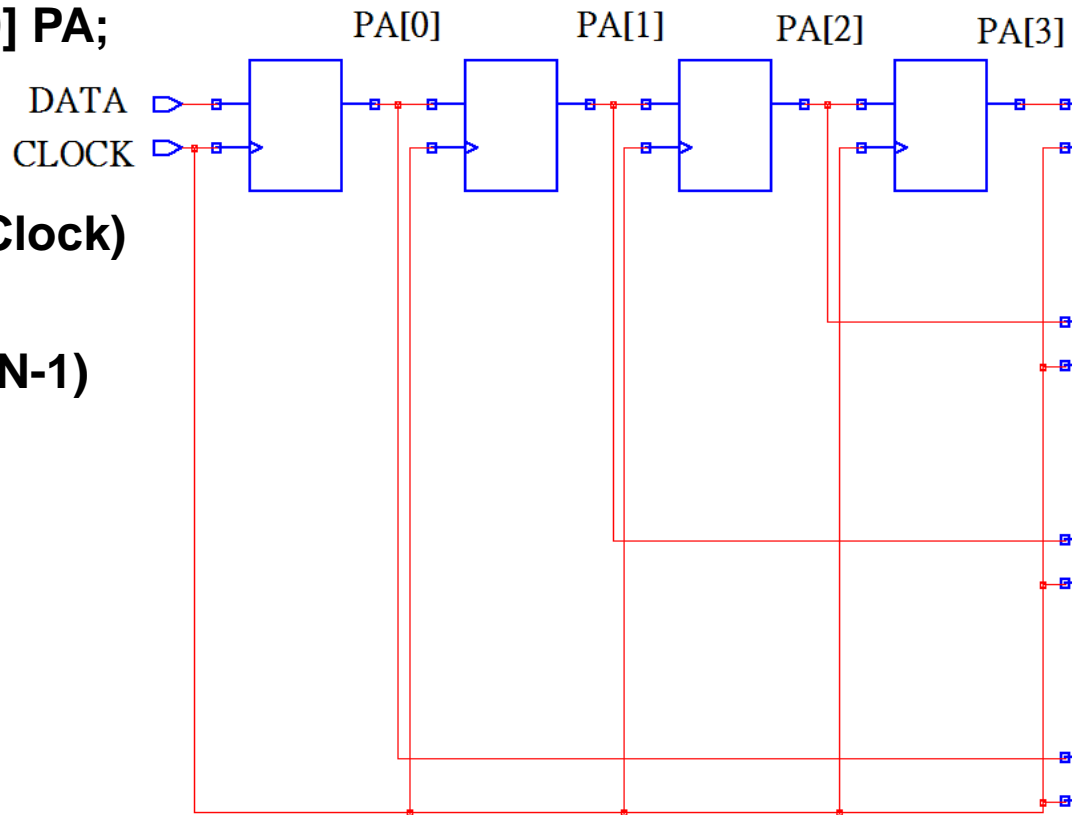
OK!

# Example of blocking and for (1/3)

```
module test3(Clock, Data, YA, YB);  
  input Clock, Data;  
  output [3:0] YA;  
  reg [3:0] YA; reg [3:0] PA;  
  integer N;
```

```
  PA[3]<=PA[2];PA[2]<=PA[1];PA[1]<=PA[0];  
  PA[0]<=Data; YA[0]<=PA[0]; YA[1]<=PA[1];  
  YA[2]<=PA[2]; YA[3]<=PA[3];
```

```
  always @(posedge Clock)  
  begin  
    for(N=3 ; N>=1 ; N=N-1)  
      PA[N] <= PA[N-1];  
    PA[0] <= Data;  
    YA <= PA;  
  end  
endmodule
```



# Example of blocking and for (2/3)

```
module test1(Clock, Data, YA, YB);
```

```
input Clock, Data;
```

```
output [3:0] YA;
```

```
reg [3:0] YA, PA;
```

```
integer N;
```

```
always@(posedge Clock)
```

```
begin
```

```
for( N=1 ; N<=3 ; N=N+1)
```

```
    PA[N] = PA[N-1];
```

```
    PA[0] = Data;
```

```
    YA = PA;
```

```
end
```

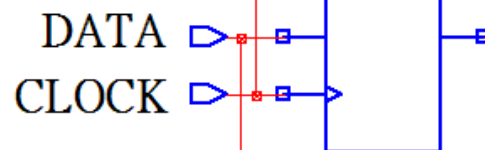
```
endmodule
```

```
PA[1]=PA[0];
```

```
PA[2]=PA[1];
```

```
PA[3]=PA[2];
```

```
PA[0]=Data;
```



```
YA[0]=PA[0]
```

```
YA[1]=PA[1]
```

```
YA[2]=PA[2]
```

```
YA[3]=PA[3]
```

```
always@(posedge Clock)
begin
    for(N=3 ; N>=1 ; N=N-1)
        PA[N] = PA[N-1];
    PA[0] = Data;
    YA = PA;
end
endmodule
```

YA[0]=PA[0]  
YA[1]=PA[1]  
YA[2]=PA[2]  
YA[3]=PA[3]

