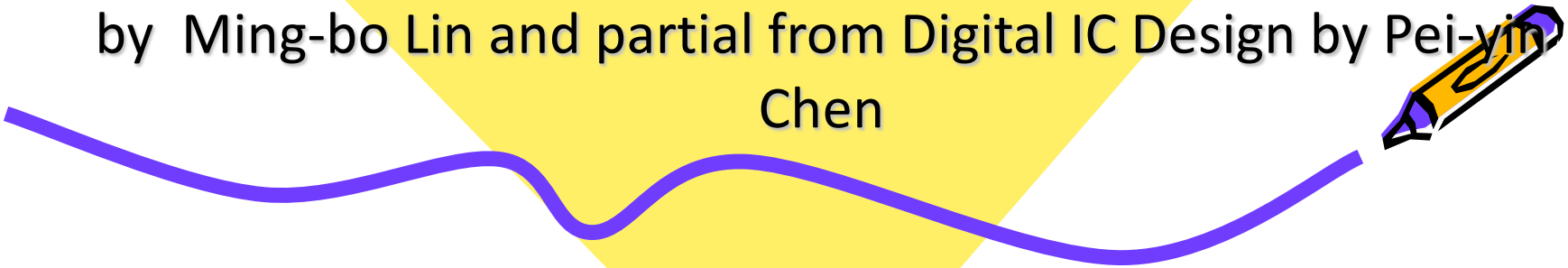
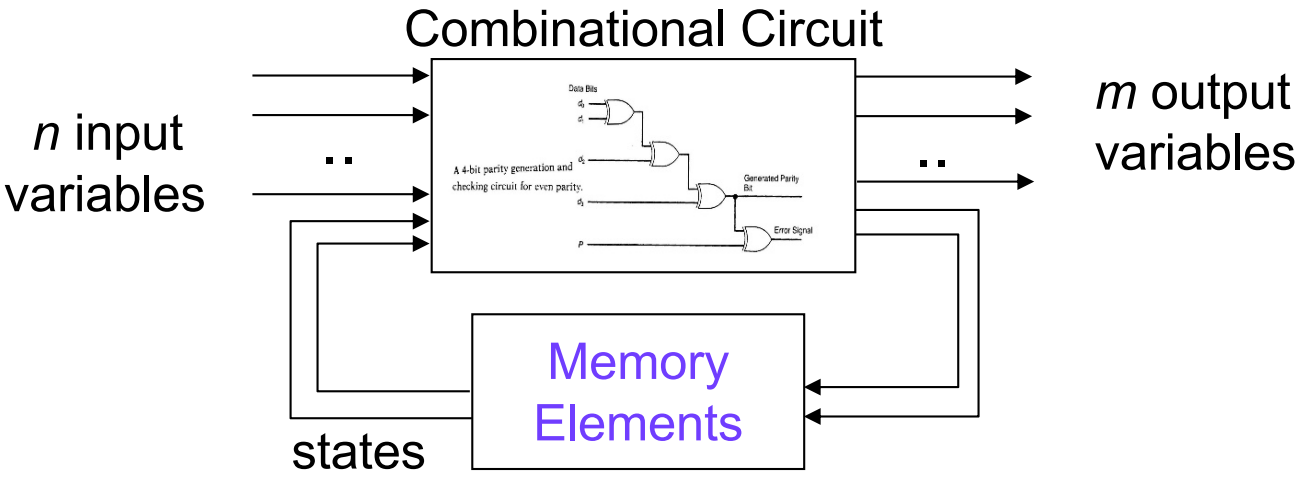
Sequential Logic Modules

Slides Partial from Digital System and Designs and Practices by Ming-bo Lin and partial from Digital IC Design by Pei-yin Chen



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.

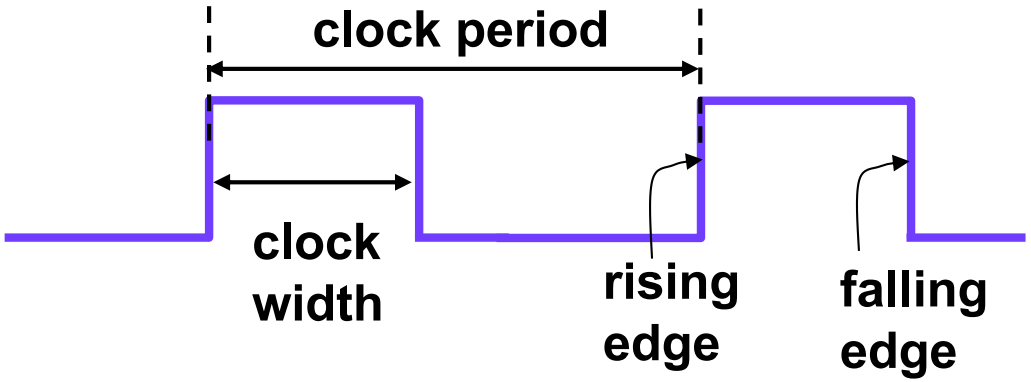
Synchronous storage components include:

(1) registers (2) counters

(3) register files (4) memories

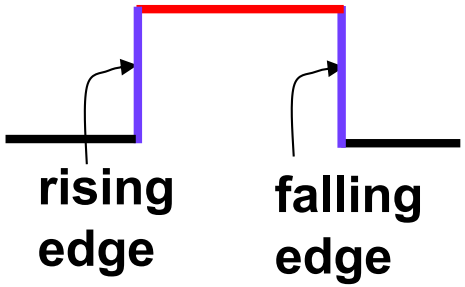
(5) queues (6) stacks

Clock Period



- Clock period (measured in micro or nanoseconds) is the time between successive transitions in the same direction
- Clock frequency (measured in MHz or GHz) is the reciprocal of clock period
- 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).

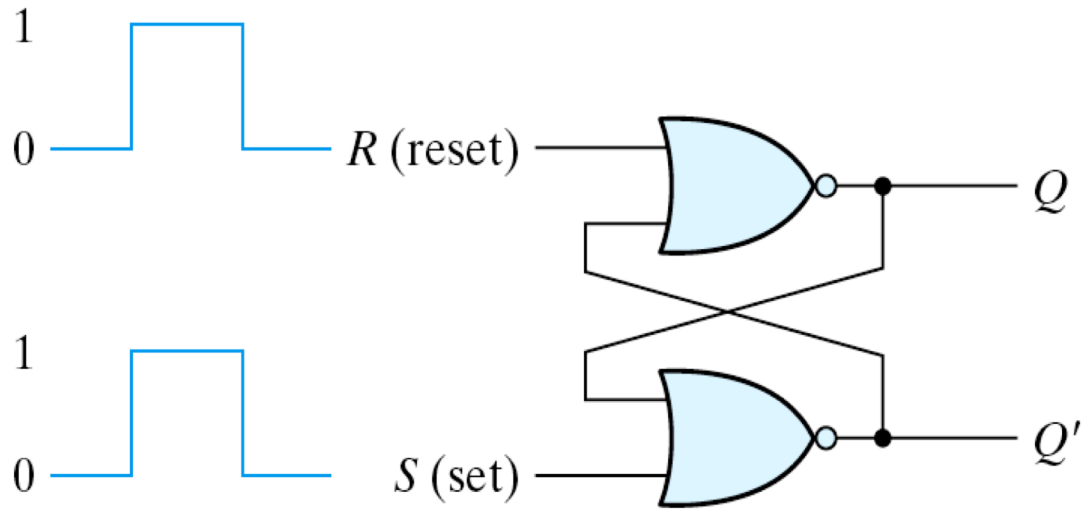
They are easy to work with though more expensive than latches.

Two basic styles of flip-flops are available:

1) Master-slave
2) Edge-triggered

Latches (1/2) (using NOR gate)

- 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 its state indefinitely until directed by an input signal to switch states



(a) Logic diagram

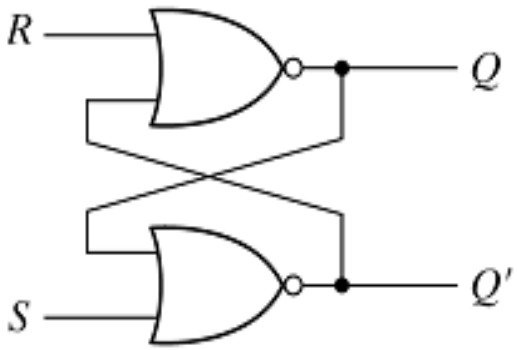
S	R	Q	Q'	
1	0	1	0	
0	0	1	0	(after $S = 1, R = 0$)
0	1	0	1	
0	0	0	1	(after $S = 0, R = 1$)
1	1	0	0	(forbidden)

(b) Function table

Two NOR gates

Set→1, Reset→0

Latches (2/2) using NOR gate



S	R	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

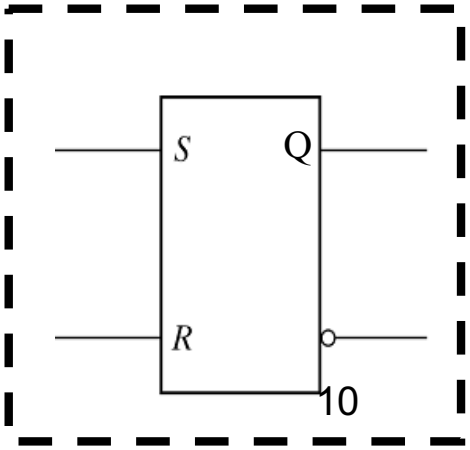
the condition should be avoided

- more complicated types can be built upon it
- an asynchronous sequential circuit
- (S,R)= (0,0): no operation

  (S,R)=(0,1): reset (Q=0, the clear state)

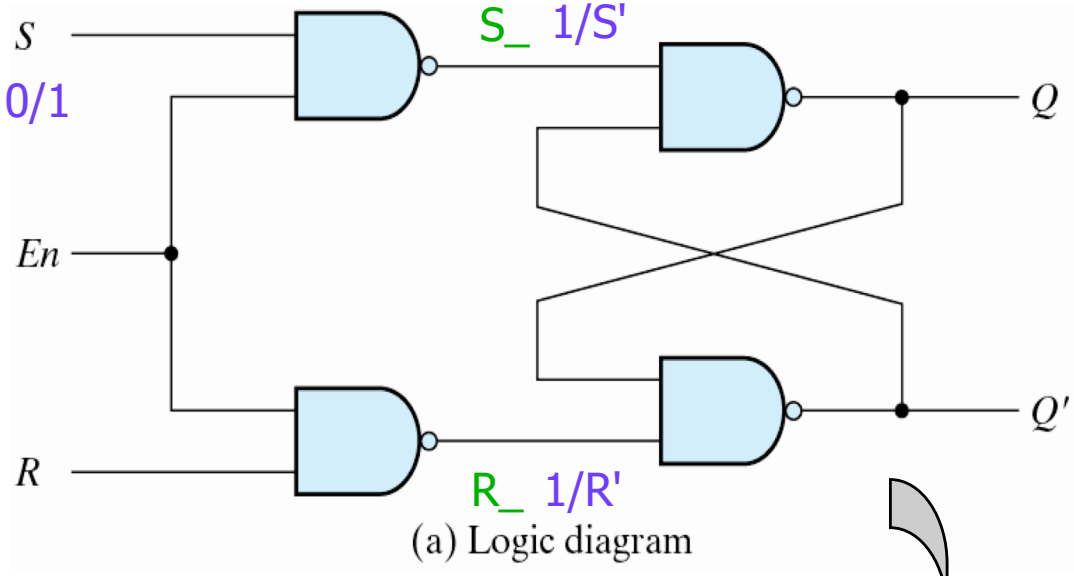
  (S,R)=(1,0): set (Q=1, the set state)

  (S,R)=(1,1): indeterminate state (Q=Q'=0)
- consider (S,R) = (1,1) $\Rightarrow$ (0,0)



SR Latch with Control Input using NAND gate

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



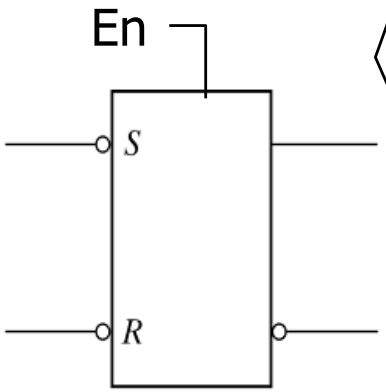
(a) Logic diagram

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

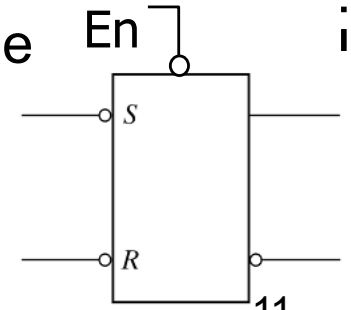
En=1, enable



(c) Graphic symbol

En=1, no change

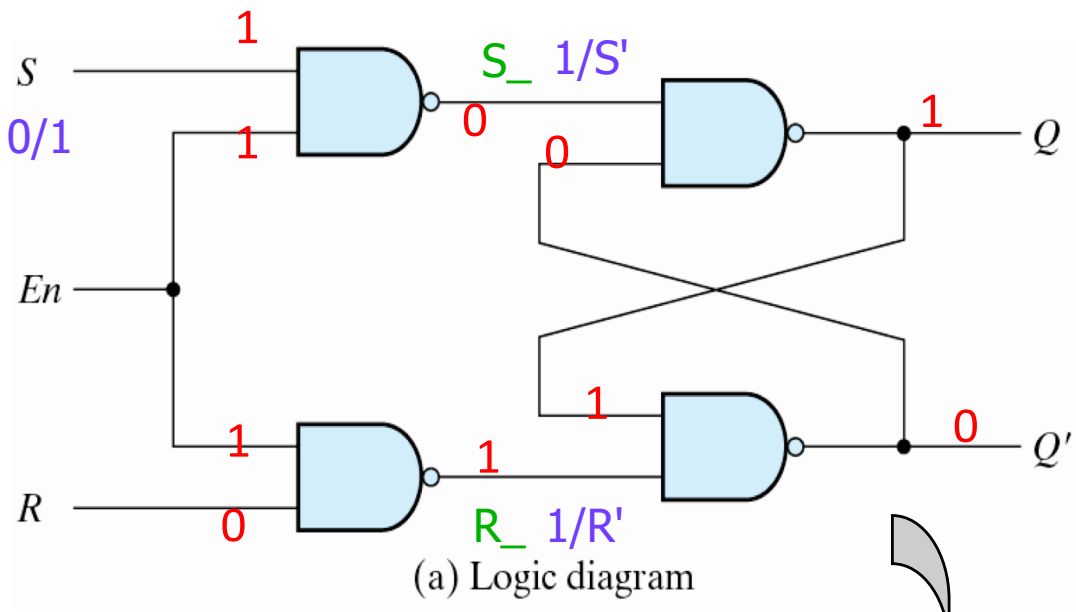
En=0, enable



(c) Graphic symbol

SR Latch with Control Input using NAND gate

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



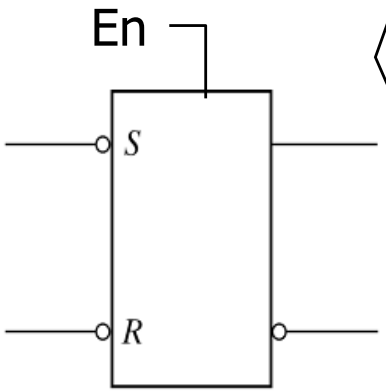
(a) Logic diagram

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

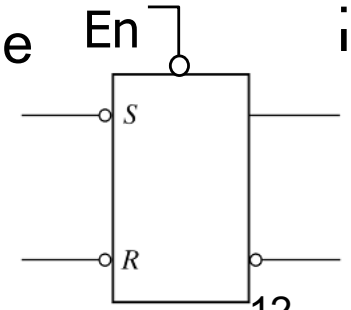
En=1, enable



(c) Graphic symbol

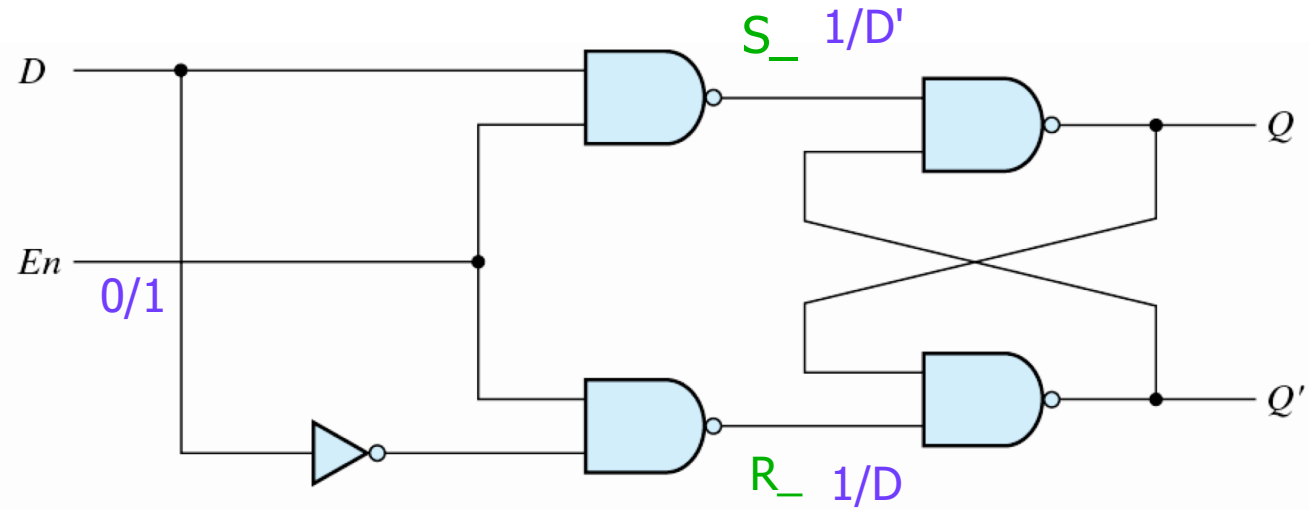
En=1, no change

En=0, enable



(c) Graphic symbol

D Latch (Transparent Latch)

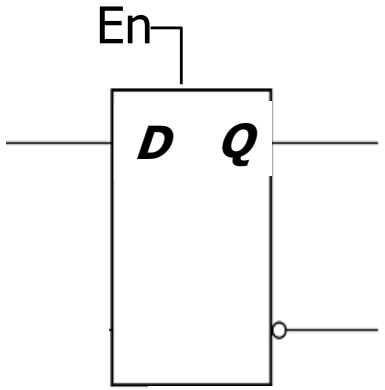


(a) Logic diagram

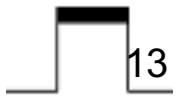
En	*D*	Next state of *Q*
0	X	No change
1	0	$Q = 0$; reset state
1	1	$Q = 1$; set state

(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

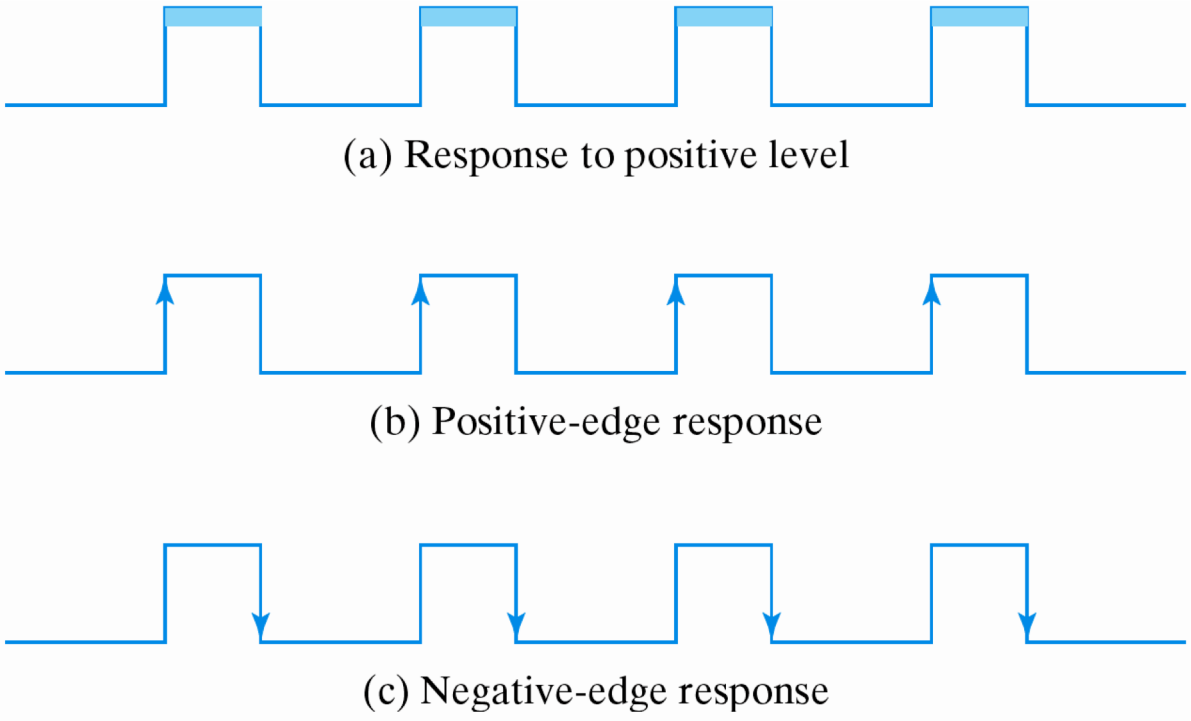


level triggered
(level-sensitive)



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

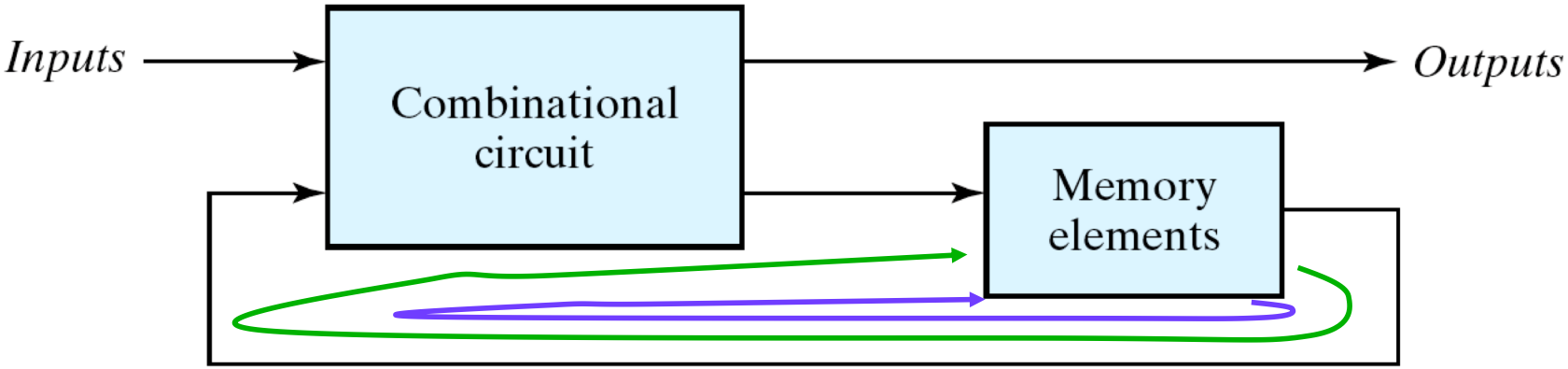


Level triggered

Edge triggered

Edge triggered

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

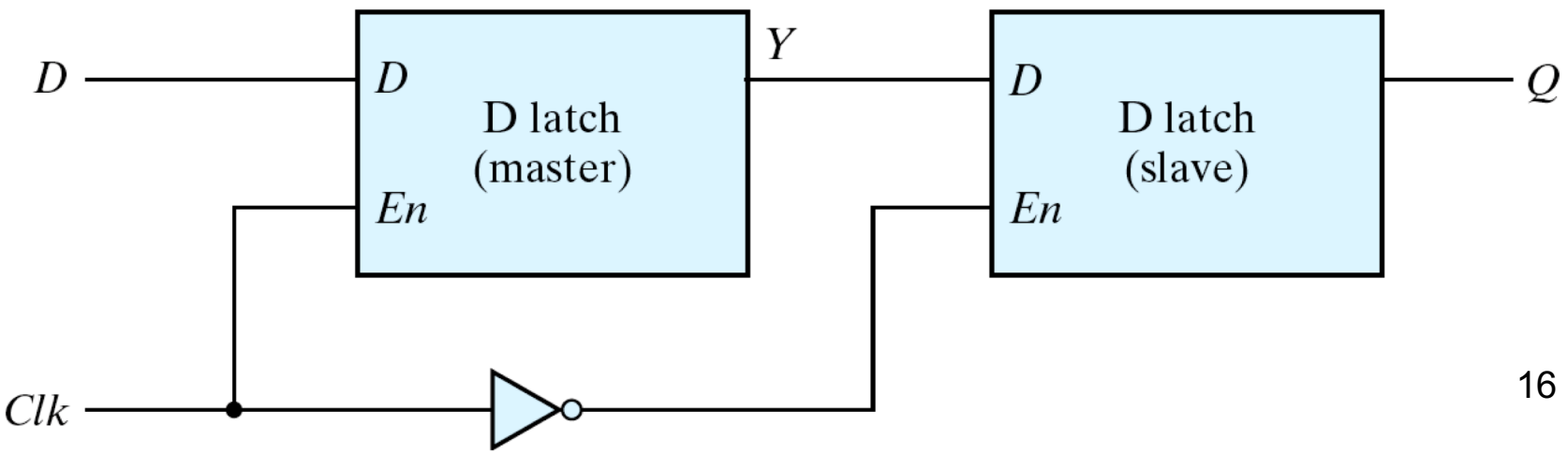
- Edge-triggered flip-flops
  - the state transition happens only at the edge
  - eliminate the multiple-transition problem

Edge-Triggered D Flip-Flop

Two D FF styles to solve the problem of latch:

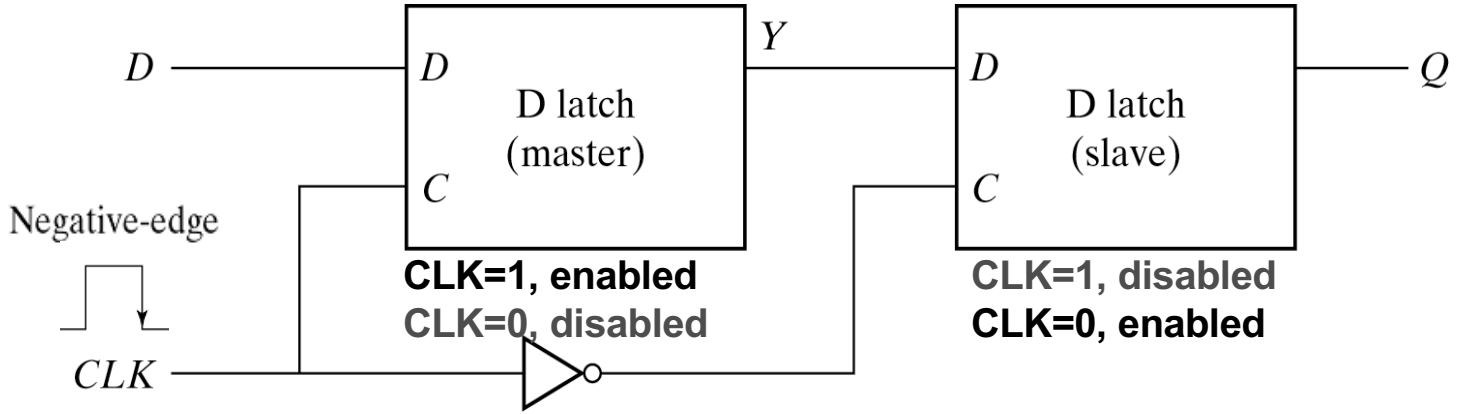
a. Master-slave
b. Edge-trigger

- Master-slave D flip-flop
  - a master latch (positive-level triggered)
  - a slave latch (negative-level triggered)

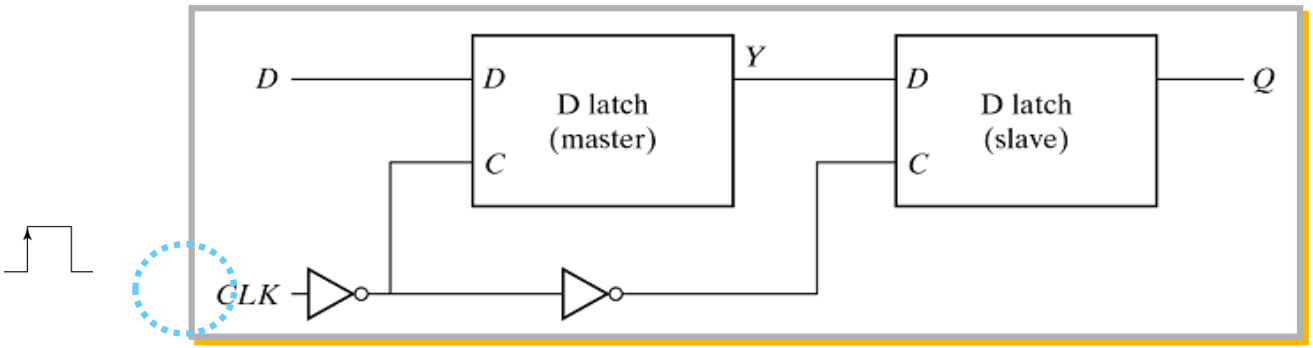


Master-slave D flip-flop (1/2)

- Two D latches and one inverter
- 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

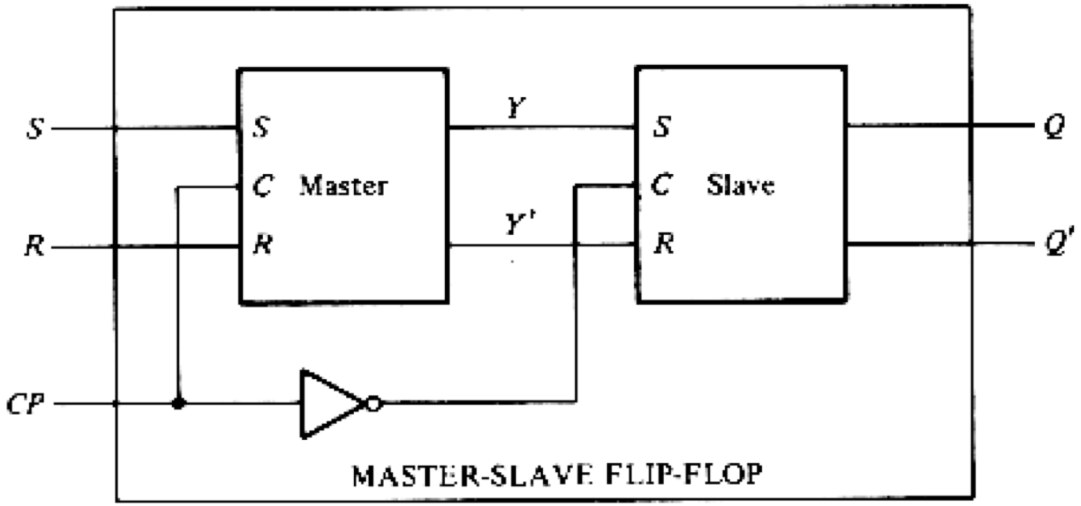


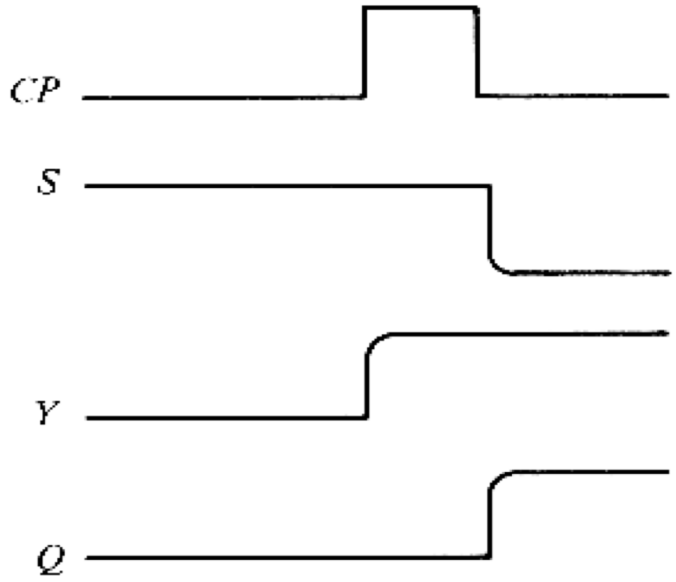
Positive edge



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



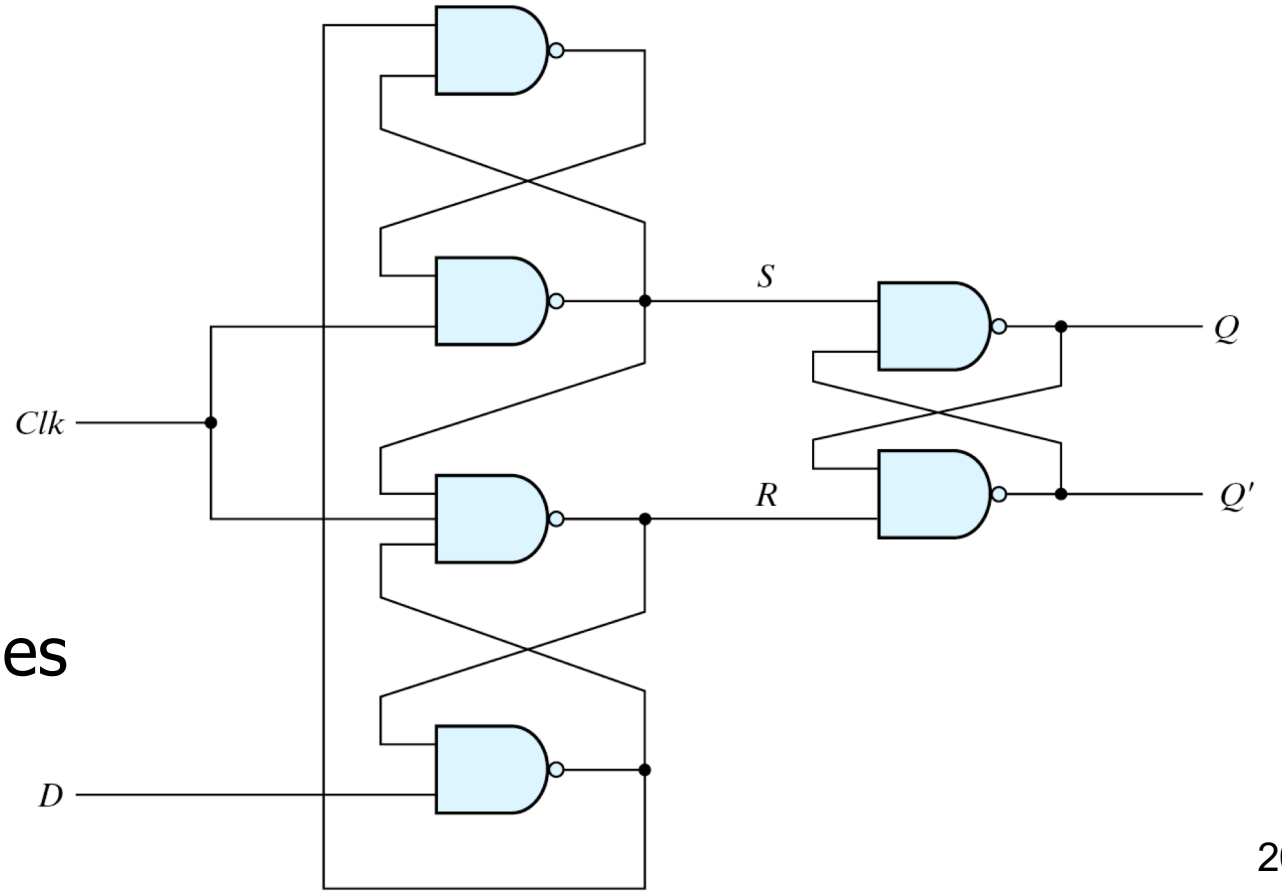


Negative Edge Trigger

Edge-Triggered Flip-Flops (1/5)

the state changes during a clock-pulse transition

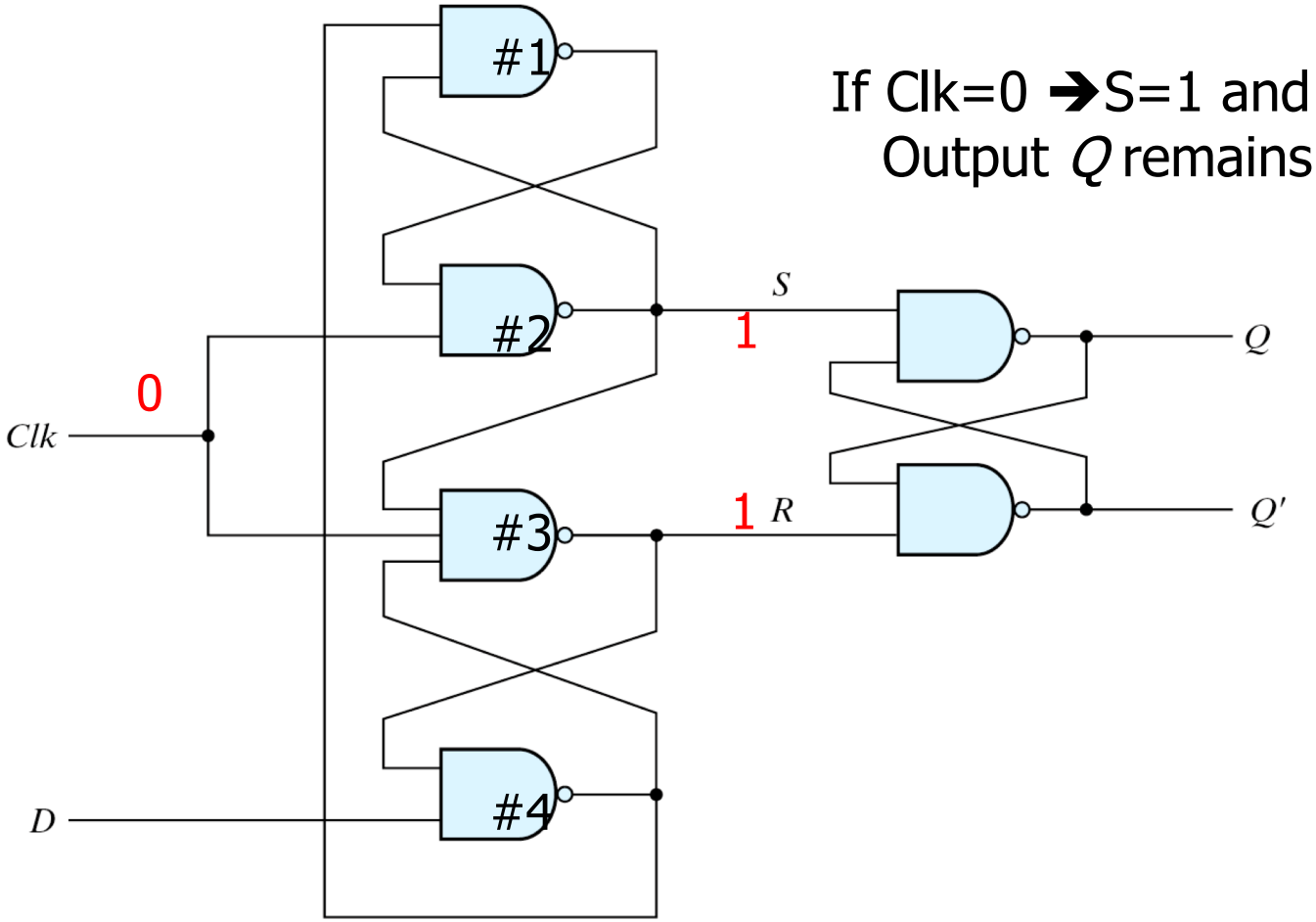
A D-type positive-edge-triggered flip-flop



Three SR latches

Edge-Triggered Flip-Flops (2/5)

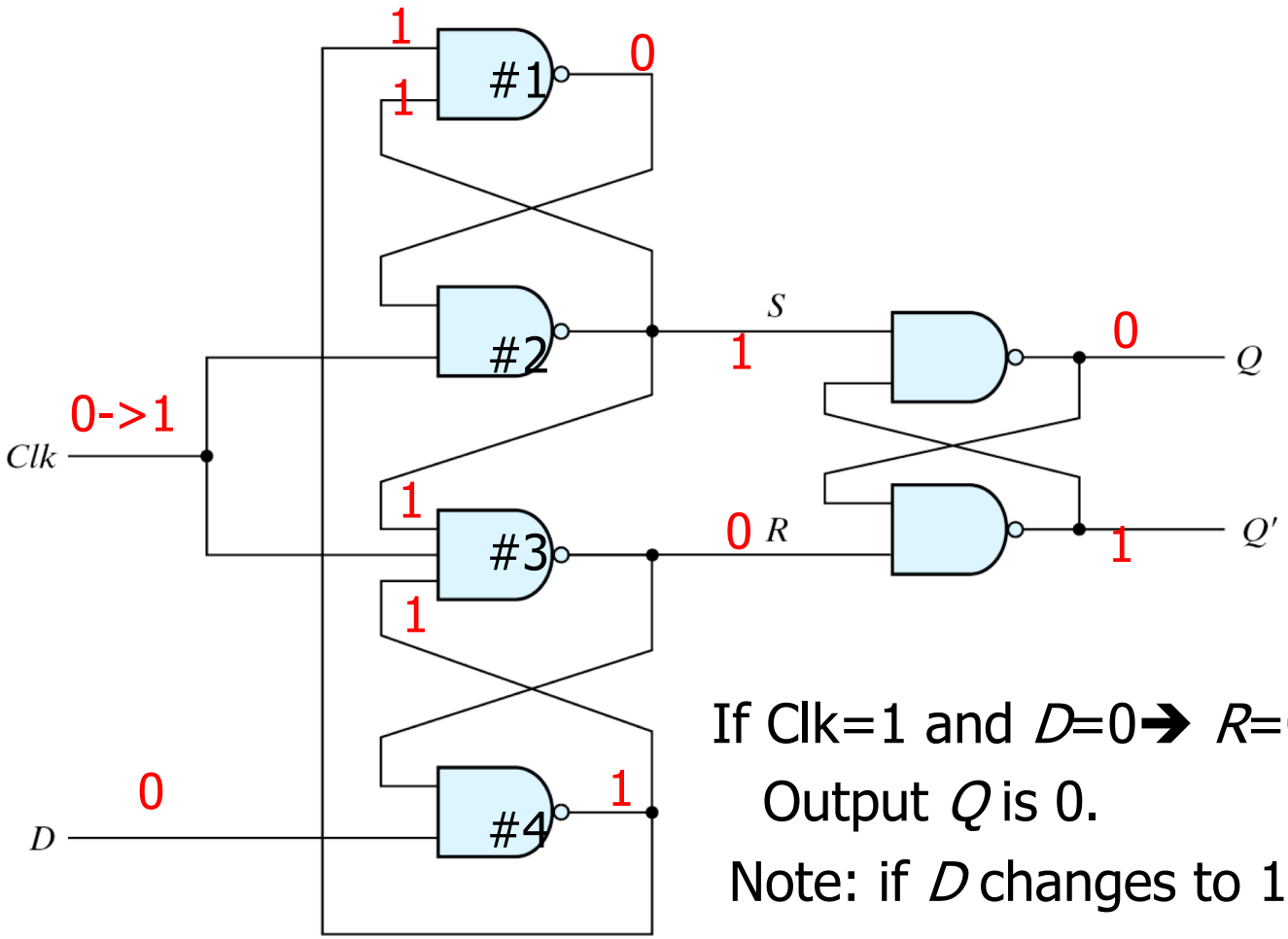
(S,R) = (0,1): Q = 1 (S,R) = (1,0): Q = 0
(S,R) = (1,1): no operation (S,R) = (0,0): should be avoided



If Clk=0 ➔S=1 and R=1 ➔ no operation.
Output *Q* remains in the present state.

Edge-Triggered Flip-Flops (3/5)

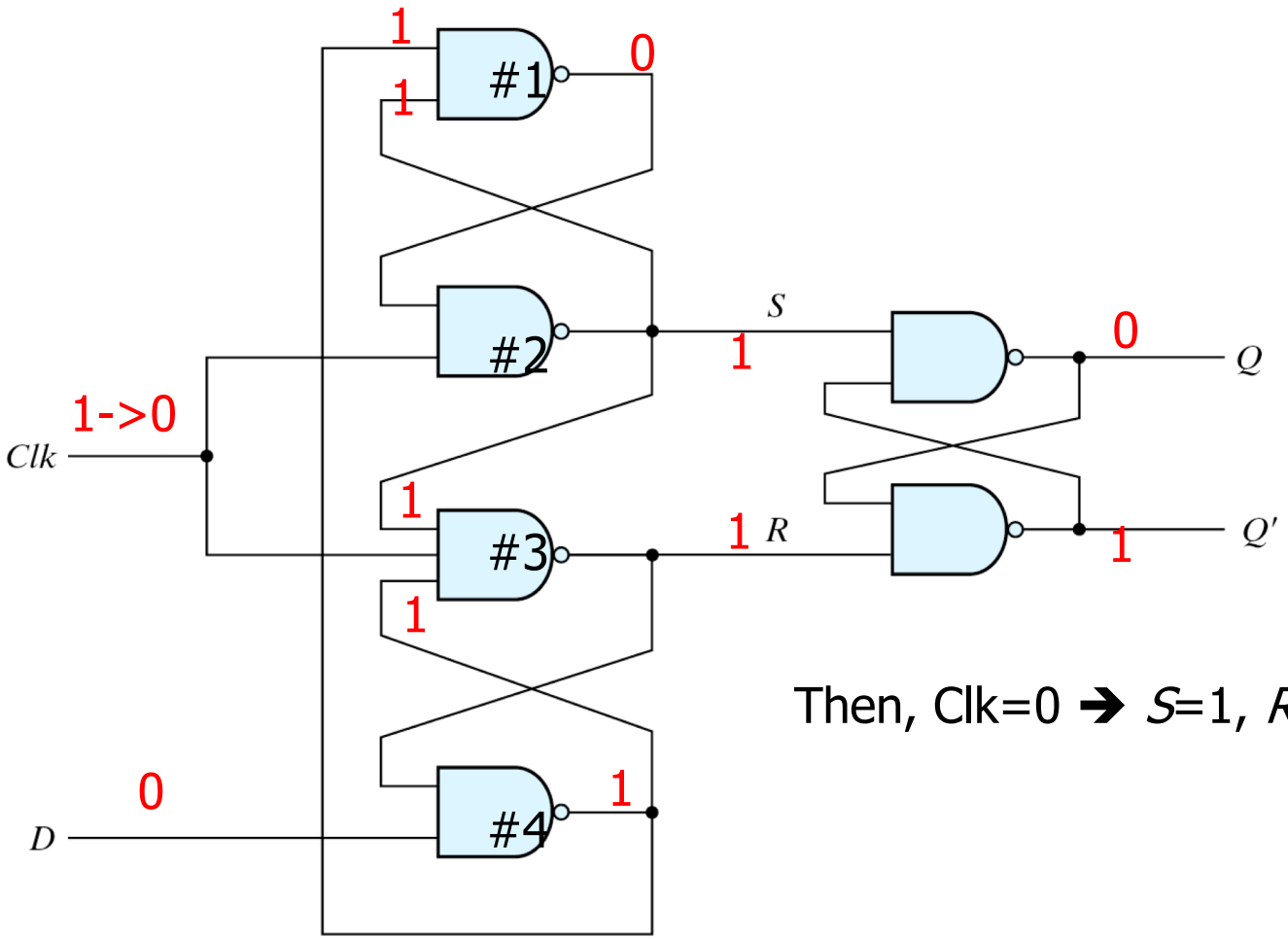
(S,R) = (0,1): Q = 1 (S,R) = (1,0): Q = 0
(S,R) = (1,1): no operation (S,R) = (0,0): should be avoided



If Clk=1 and *D*=0➔ *R*=0 ➔ Reset.
Output *Q* is 0.
Note: if *D* changes to 1, *R* remains at 0 and *Q* is 0.

Edge-Triggered Flip-Flops (4/5)

(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

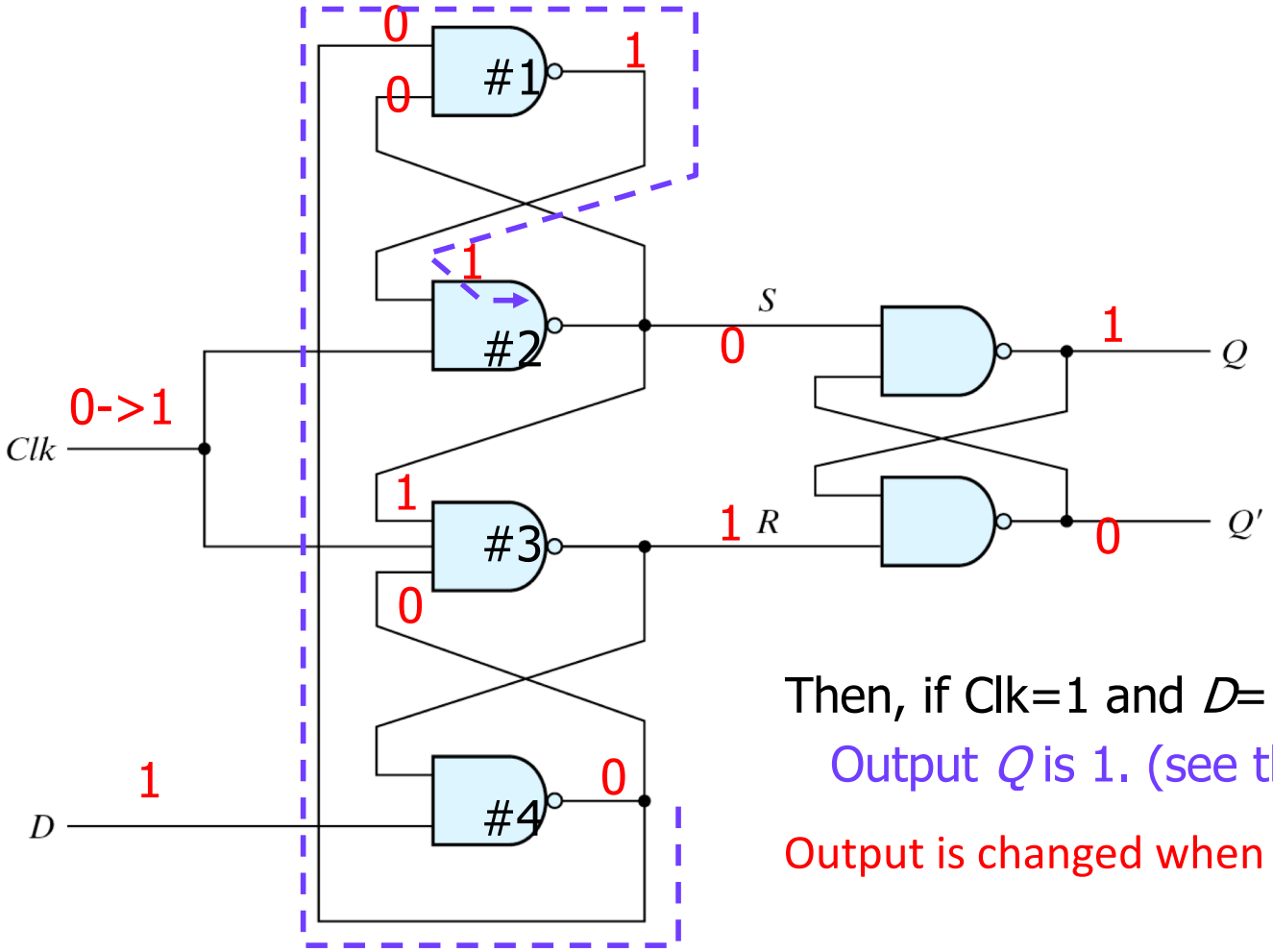


Then, Clk=0 ➔ *S*=1, *R*=1 ➔ no operation (Q=0)

Edge-Triggered Flip-Flops (5/5)

(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



Then, if Clk=1 and *D*=1➔ *S*=0➔ Set.

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

Output is changed when Clk is switching from 0 to 1

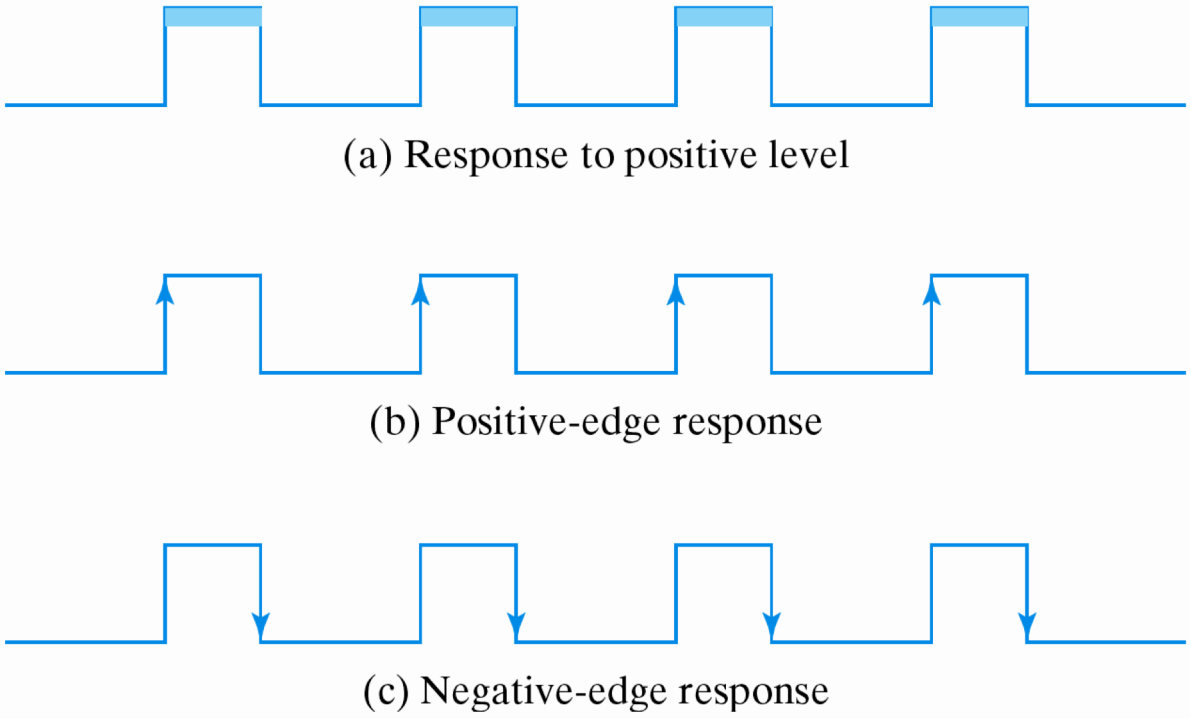
Note: Now, if *D* changes to 0, *S* remains at 0 and *Q*=1;

Summary: Positive-Edge-Triggered Flip-Flops

- Summary
  - Clk=0: (S,R) = (1,1), no state change
  - Clk=↑: state change once
  - 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



Level triggered

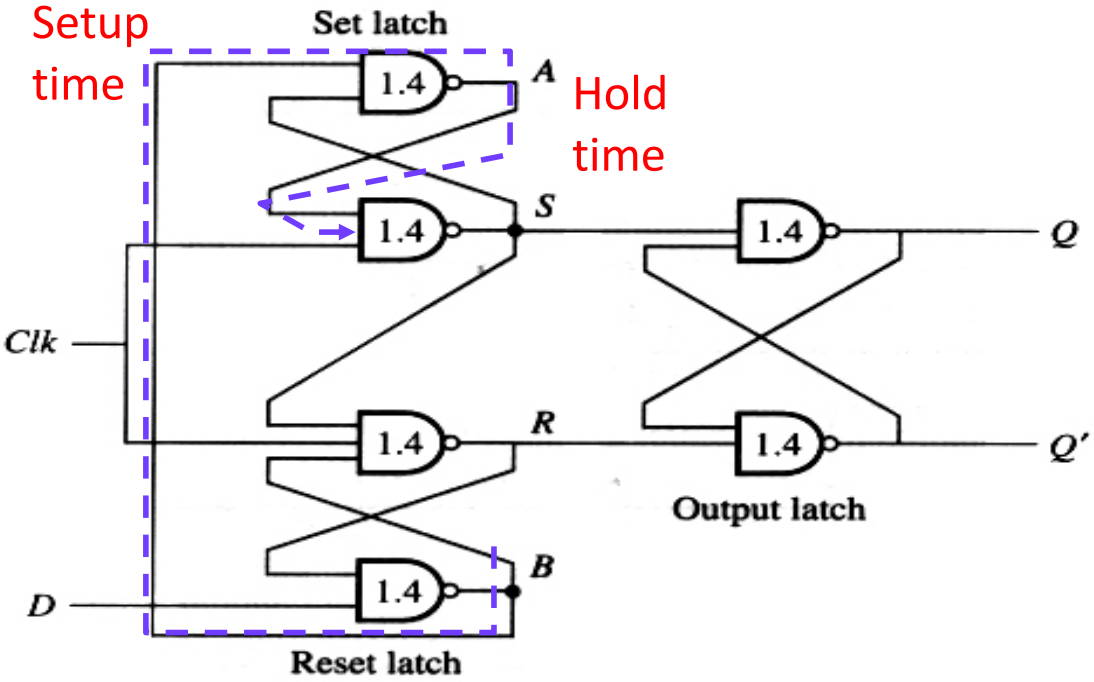
Edge triggered

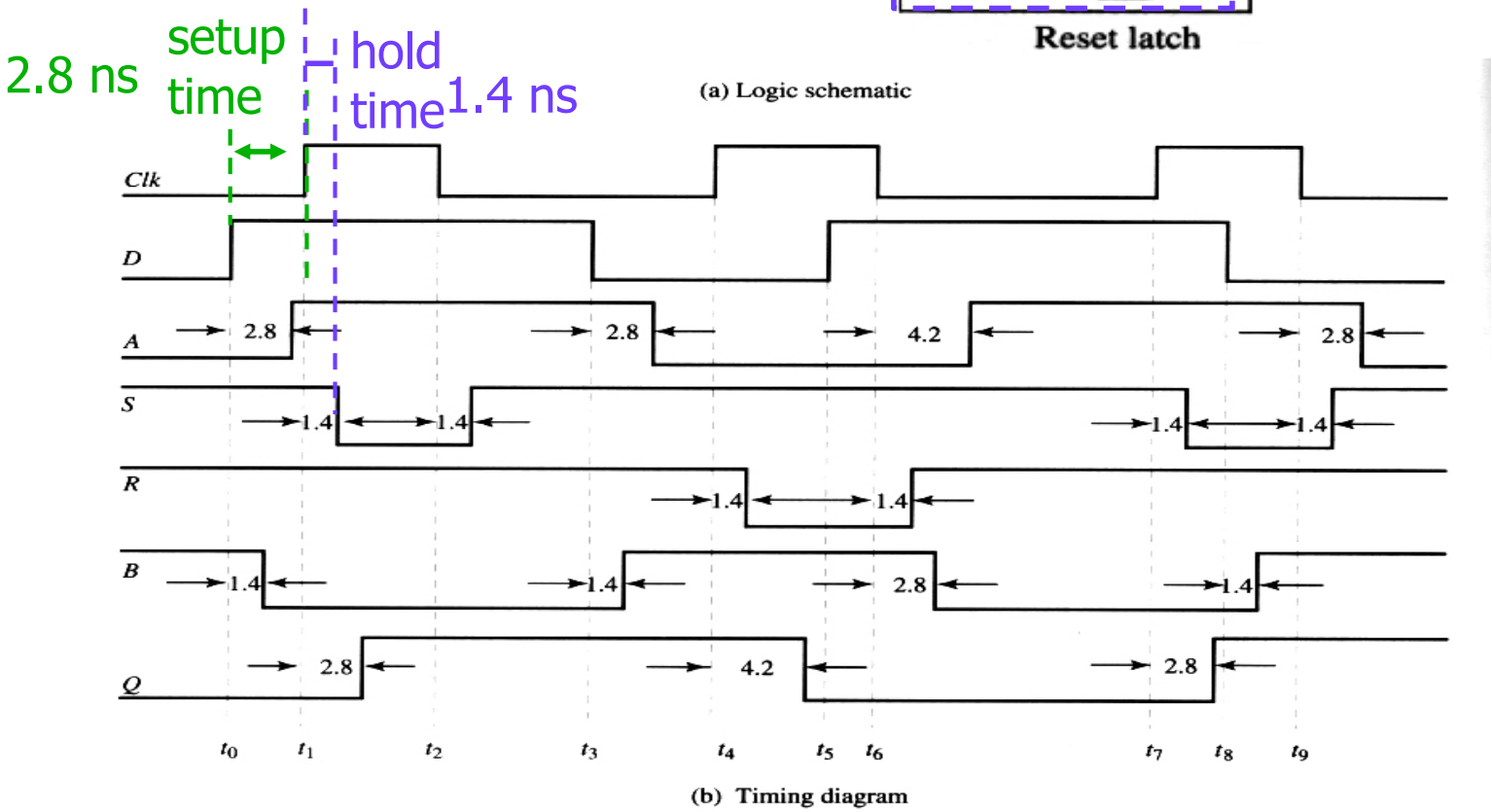
Edge triggered

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 change after the application of the positive Clk pulse
  - = the propagation delay of gate 3 (try to understand)
  - clock to the internal latch

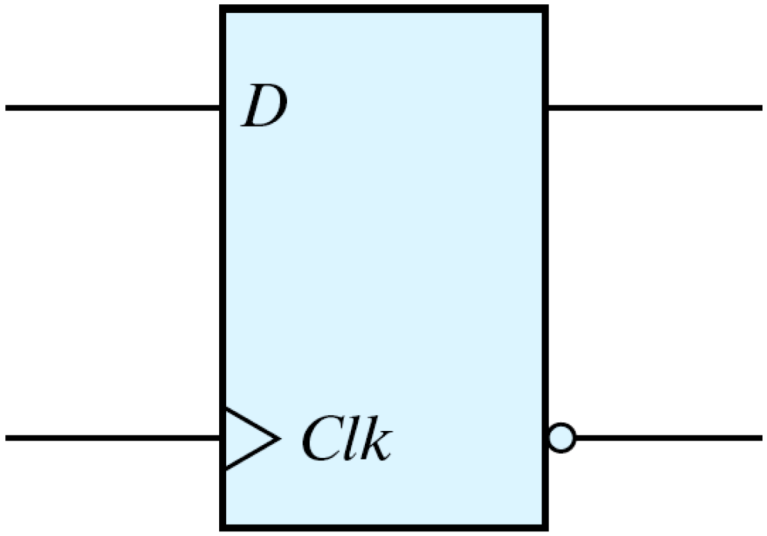
Timing Diagram



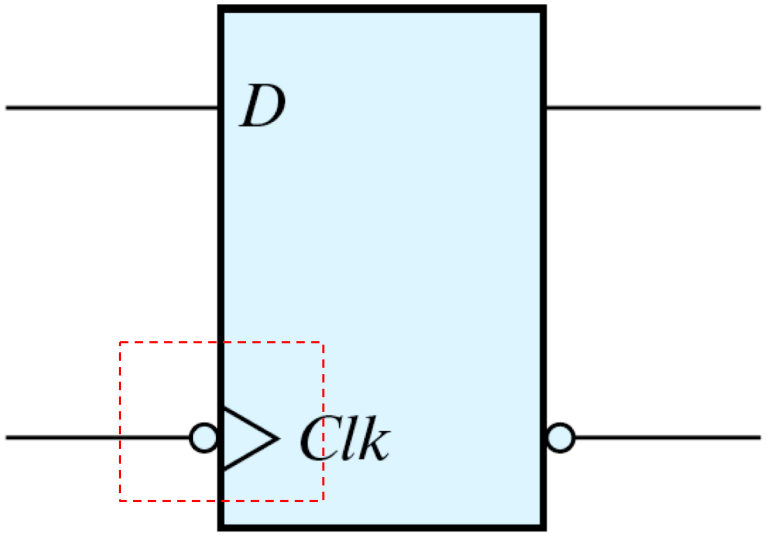


Positive-Edge vs. Negative-Edge

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



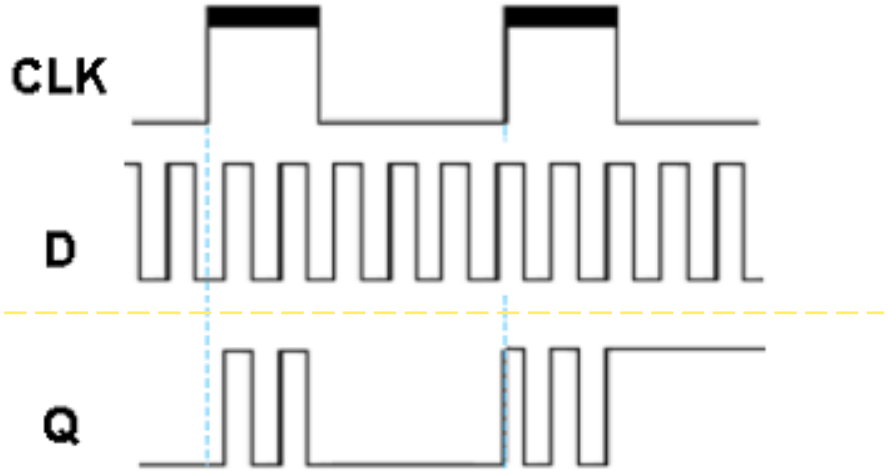
(a) Positive-edge

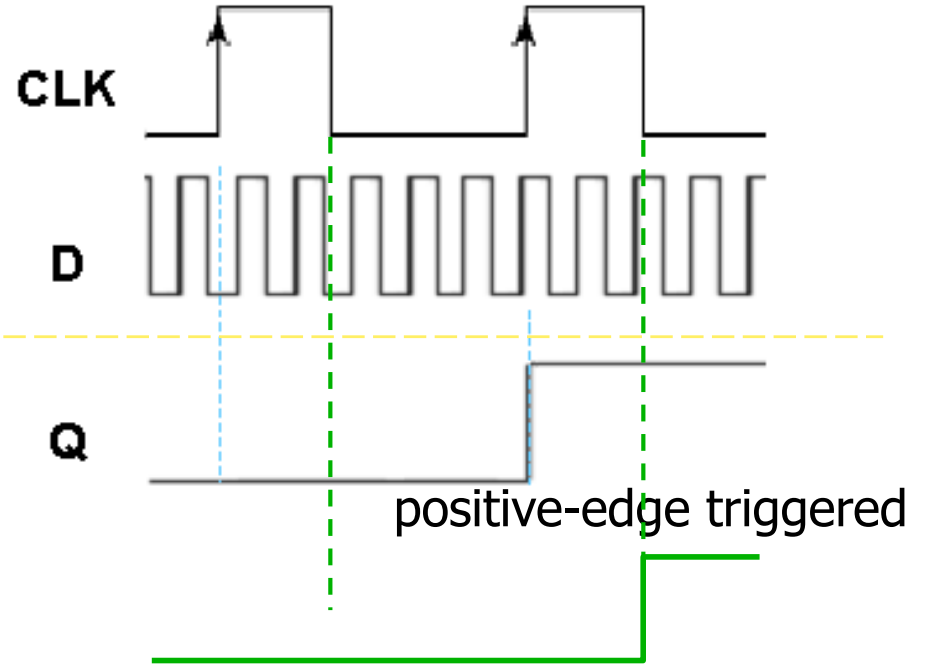


(a) Negative-edge

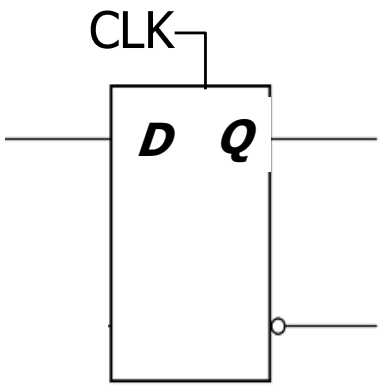
Latch vs. Flip-Flop

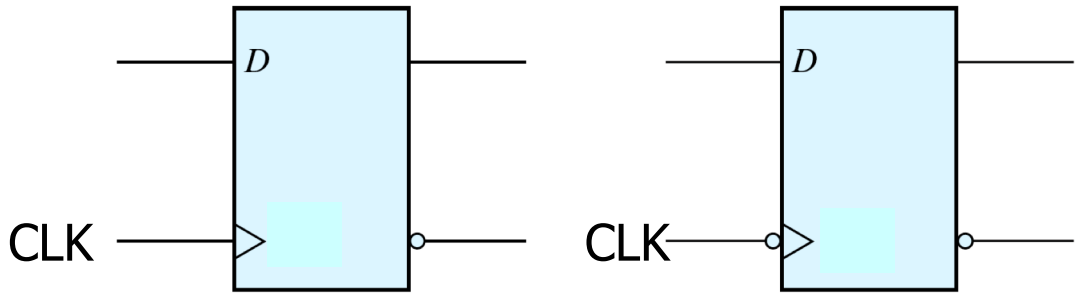
- Level triggered





Latch

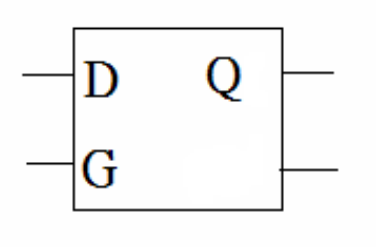




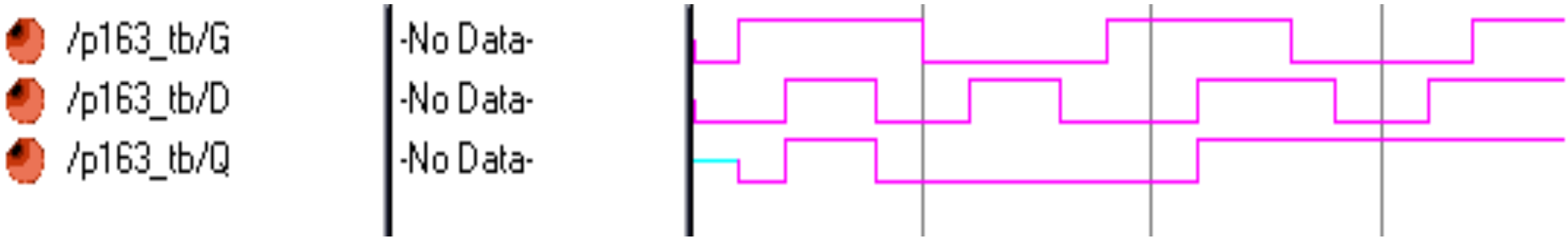
Latch

D-type latch (ignoring delay)

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

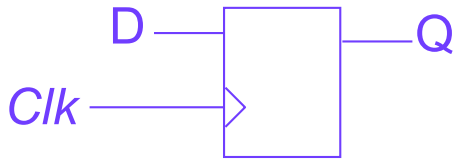



```
module p163(G, D, Q);
    input G, D;
    output Q;   reg Q;

    always @(D or G)
    begin
          if(G)
                    Q = D;
    end
endmodule
```

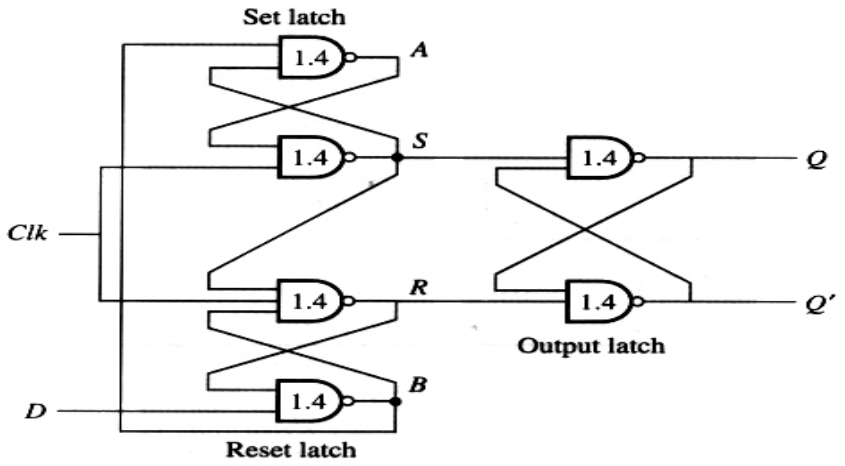


Flip-Flop (1/2)

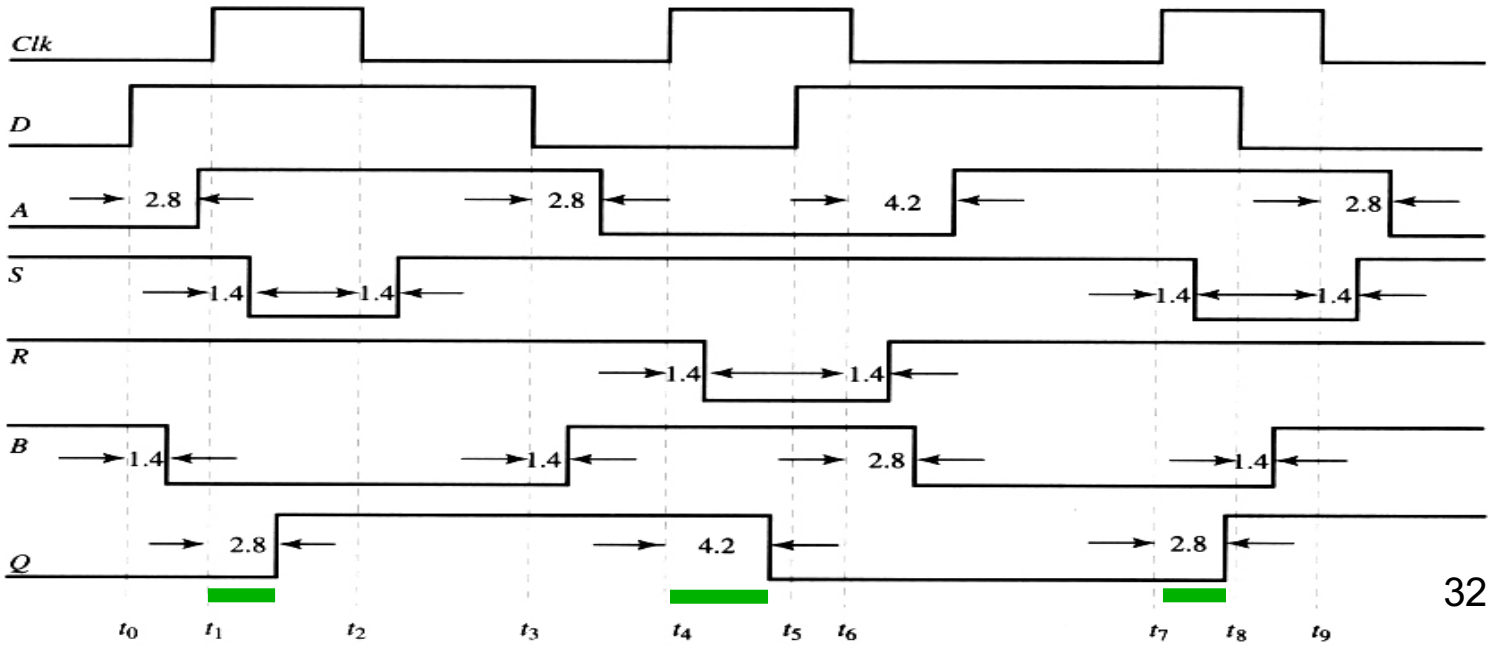
D flip-flop



Edge-triggered flip-flop



(a) Logic schematic



(b) Timing diagram

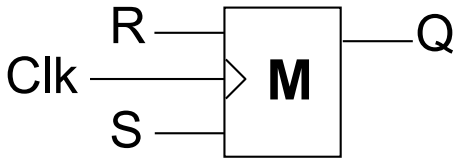
Clk-to-Q delay varies (2.8 or 4.2)

Flip-Flop (2/2)

characteristic table

excitation table

RS flip-flop

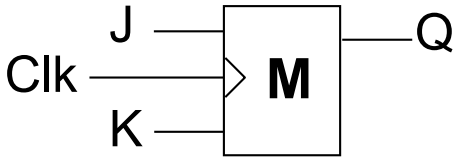


S	R	Q(t+1)
0	0	Q(t)
0	1	0
1	0	1
1	1	NA

Q(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

JK flip-flop

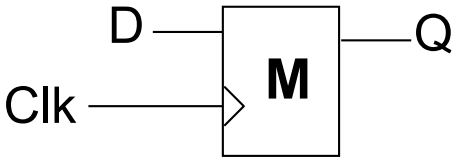


J	K	Q(t+1)
0	0	Q(t)
0	1	0
1	0	1
1	1	Q'(t)

Q(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

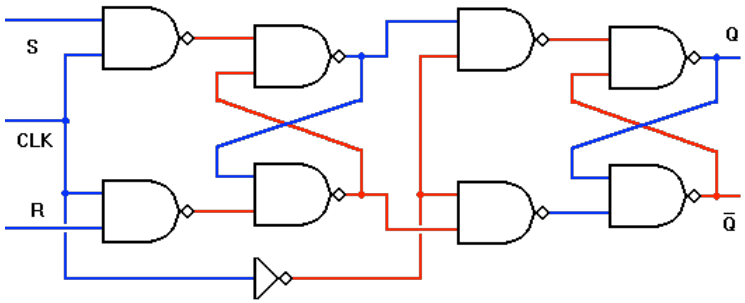
D flip-flop



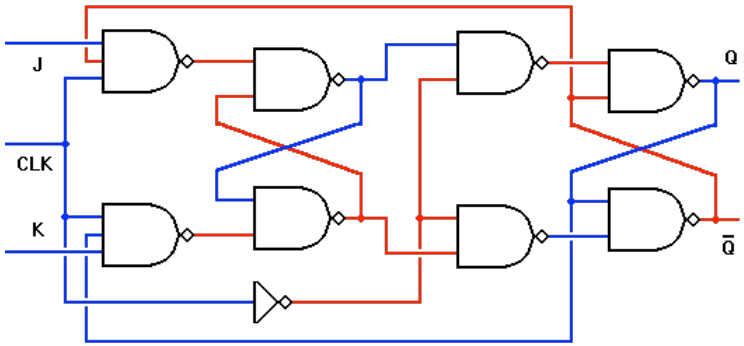
D	Q(t+1)
0	0
1	1

Q(next)=D

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



Positive Trigger SR Flip-Flop



Positive Trigger JK Flip-Flop

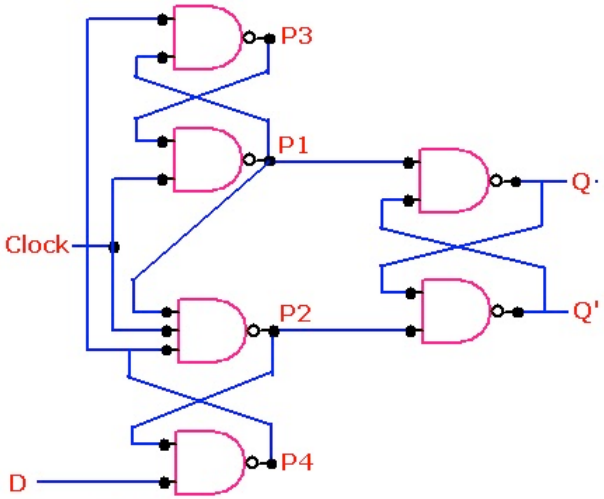


Figure 7.11
Fundamentals of Digital Logic with Verilog Design
by Stephen Brown and Zvonko Vranesic

Positive Edge Trigger D Flip-Flop

http://www.play-hookey.com/digital/sequential/d_nand_flip-flop.html

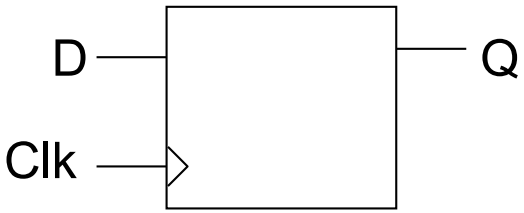
Flip-Flop Inference

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

D Flip-flop

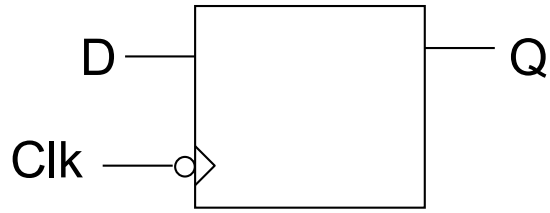
```
module D_FF(Clk, D, Q);
input  Clk, D;
output Q;
Reg    Q;

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;

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

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



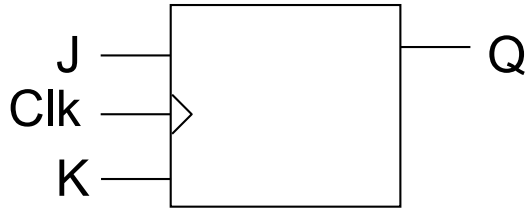

```
module Toggle (Clk, Q);
input  Clk;
output Q;
Reg    Q;

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

Toggle Flip-flop

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



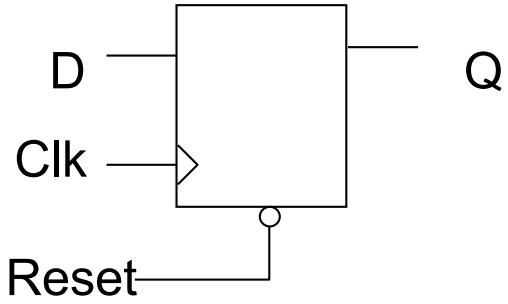

```
module JK_FF(Clk, J, K, Q);
input   Clk, J, K;
output Q, Q_Bar;
reg      Q, Q_Bar;
always @(posedge Clk)
begin
  case({J,K})
            2'b00:
                  Q=Q;
            2'b01:
                  Q=0;
            2'b10:
                  Q=1;
            2'b11:
                  Q=~Q;
    endcase
end
endmodule
```

D Flip-flop with Reset

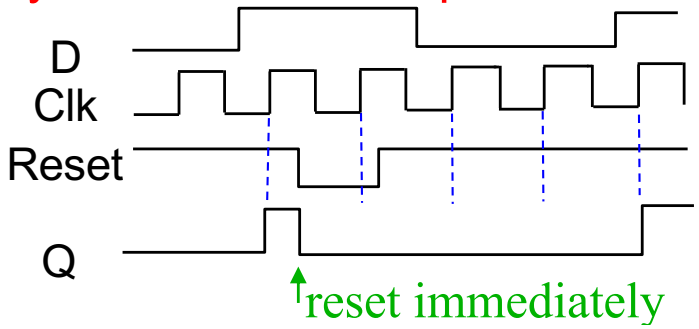
D Flip-flop with asynchronous reset

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
```



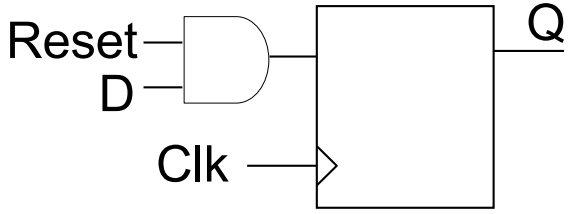
Asynchronous -- Respond immediately !

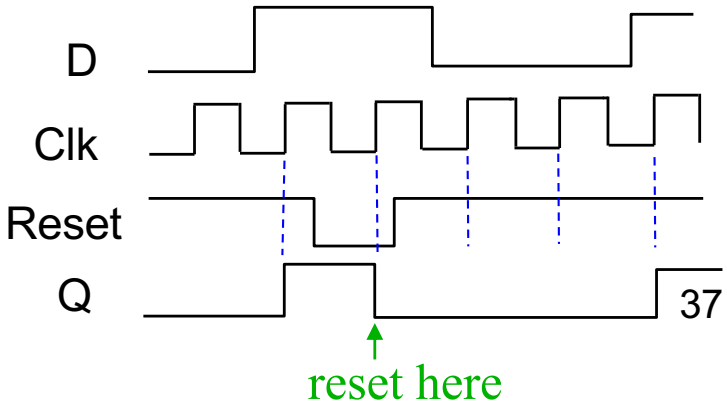


D Flip-flop with synchronous reset

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
```



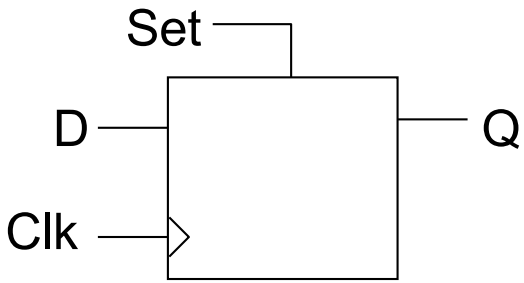


D Flip-flop with Set

D Flip-flop with asynchronous set

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

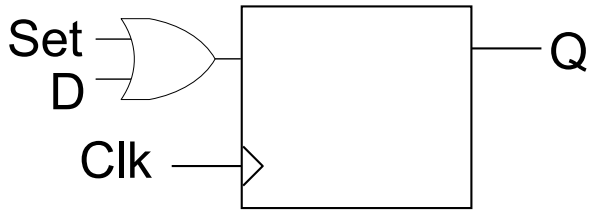
```
module DFF_AS(Clk, Set, D, Q);
input    Clk, Set, D;
output Q;
reg      Q;

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
```

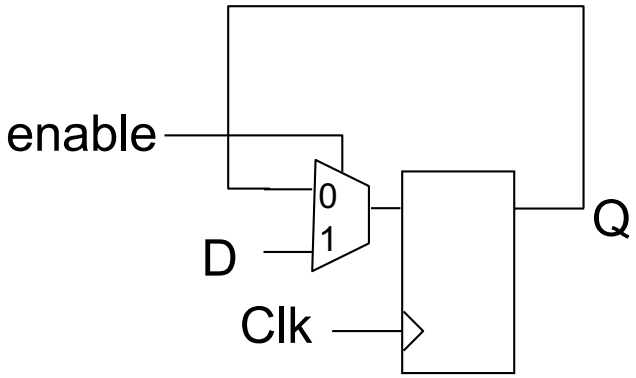
D Flip-flop with synchronous Set and synchronous Reset

D Flip-flop with Enable or Load

D Flip-flop with synchronous enable

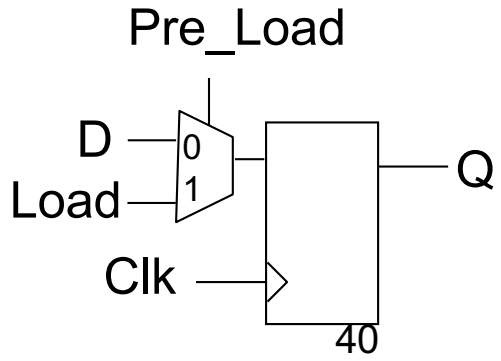
```
module DFF_MAL(Clk, enable,
D, Q);

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
```

If enable==1
Q(new)=D;
If enable==0
Q(new)=Q(old)



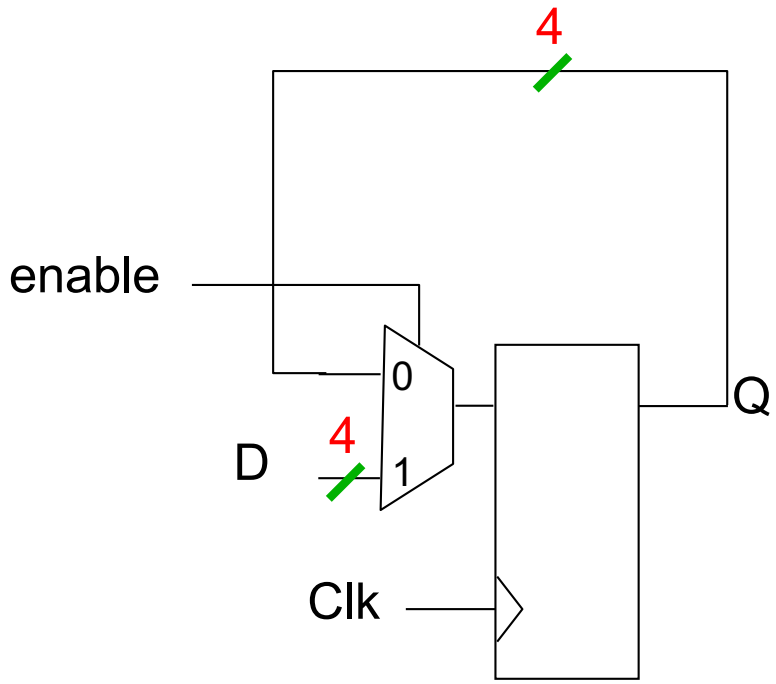
D Flip-flop with synchronous load

```
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
```

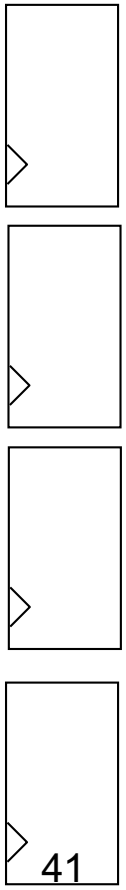


Registers

```
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
```

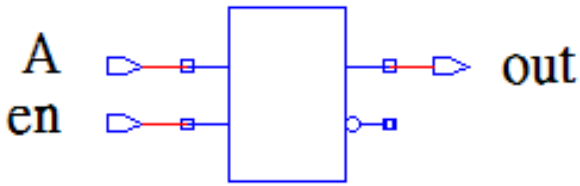


4-bit register = 4 flip-flops



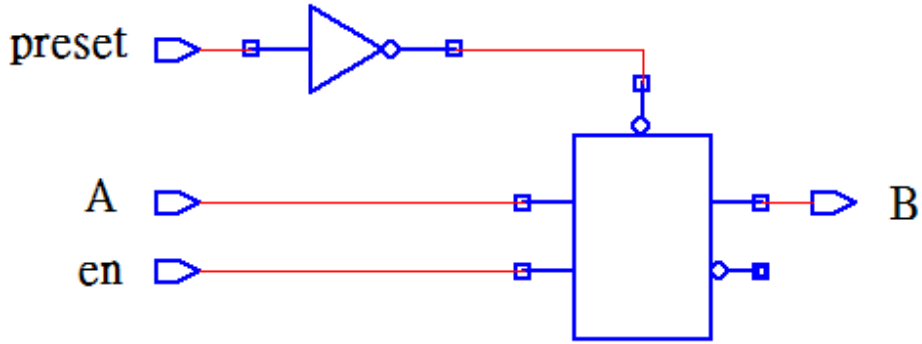
Watch Out for Unintentional Latches (1/6)

```
module latch_if1(en,A,out);
    input en, A;
    output out;  reg out;
    always @(en)
    begin
        if(en)
          out = A;
    end
endmodule
```



Latch is inferred because =>
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
```

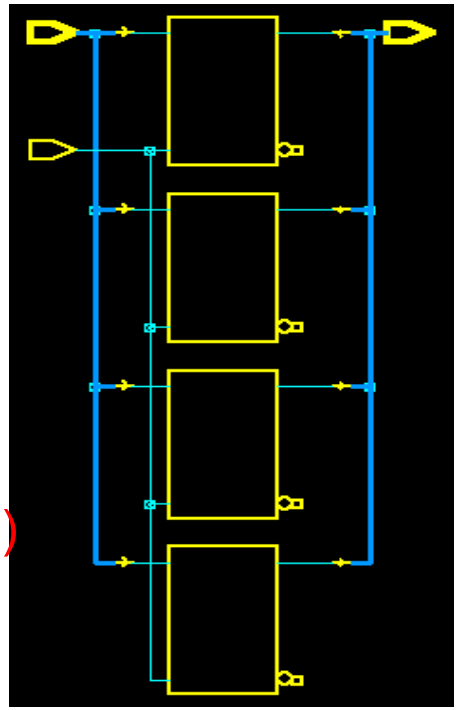


Latch is inferred because =>
If en ==1 B = A
else B (new) = B (old)

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
```

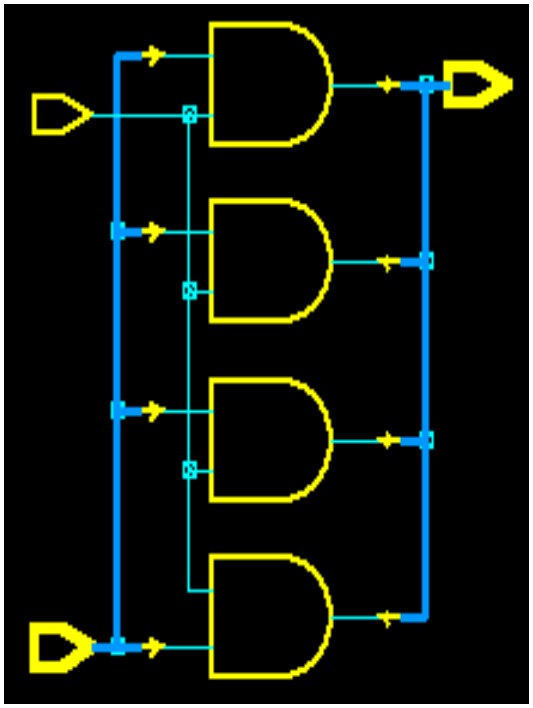
If Enable ==1
Out (new) = In
If Enable==0
Out (new) = Out (old)



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

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

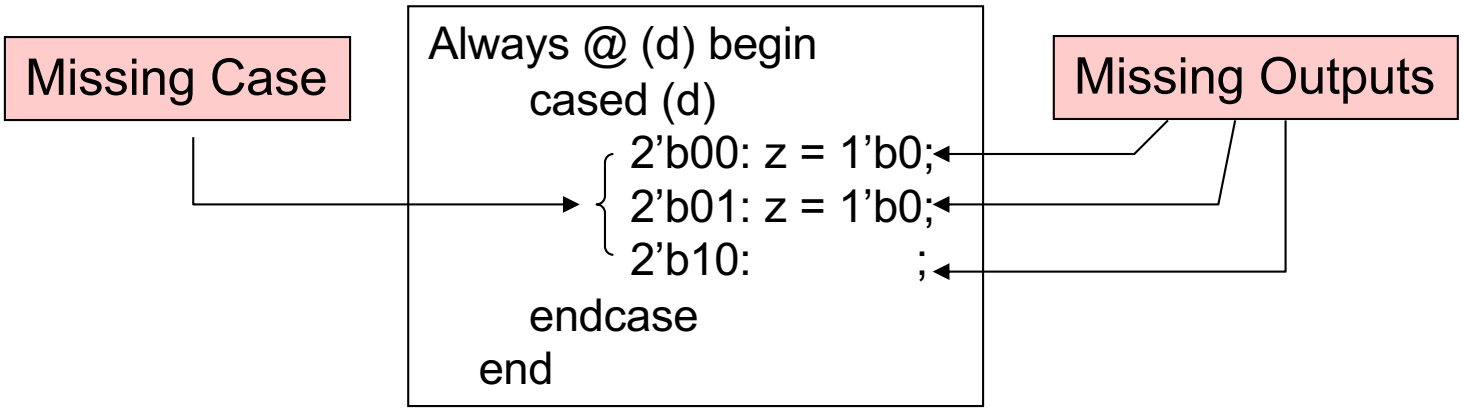
```
Out=0;
if(Enable)
          Out=In;
```

No latch inference



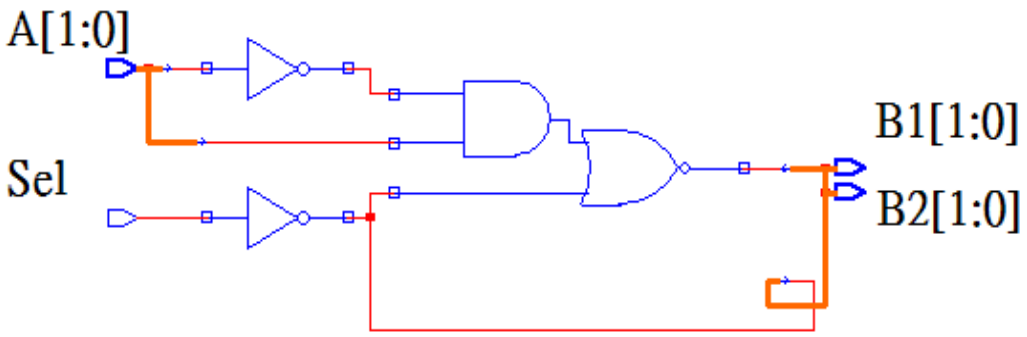
Watch Out for Unintentional Latches (3/6)

- 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

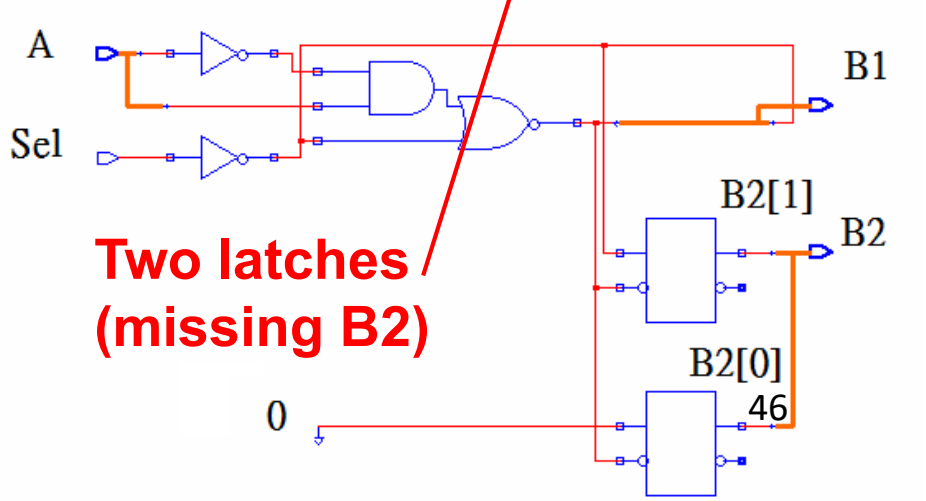


Watch Out for Unintentional Latches (4/6)

```
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
```



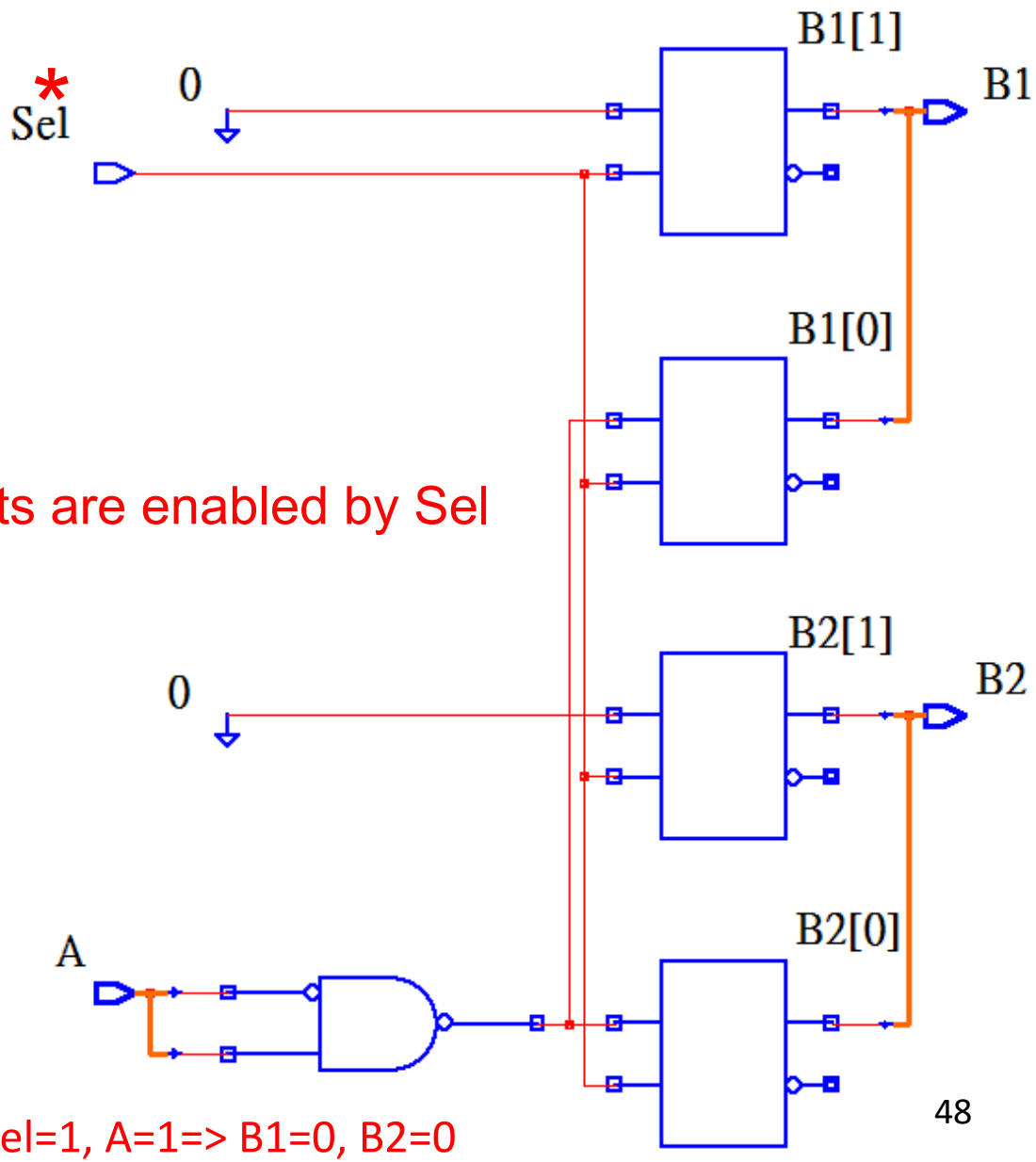

```
module code4(Sel , A , B1, B2);
 input Sel, input [1:0]A;
 output [1:0] B1, B2;  reg [1:0] B1, B2;
 always @ (Sel or A)
if(Sel)
      if(A == 1)
        begin   B1 = 0;  B2 = 0;   end
      else
        begin  B1 = 1;                end
else
     begin    B1 = 2; B2 = 2;          end
endmodule
```



Watch Out for Unintentional Latches (5/6)

* If Sel ==0 Out (new) = Out (old)

```
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

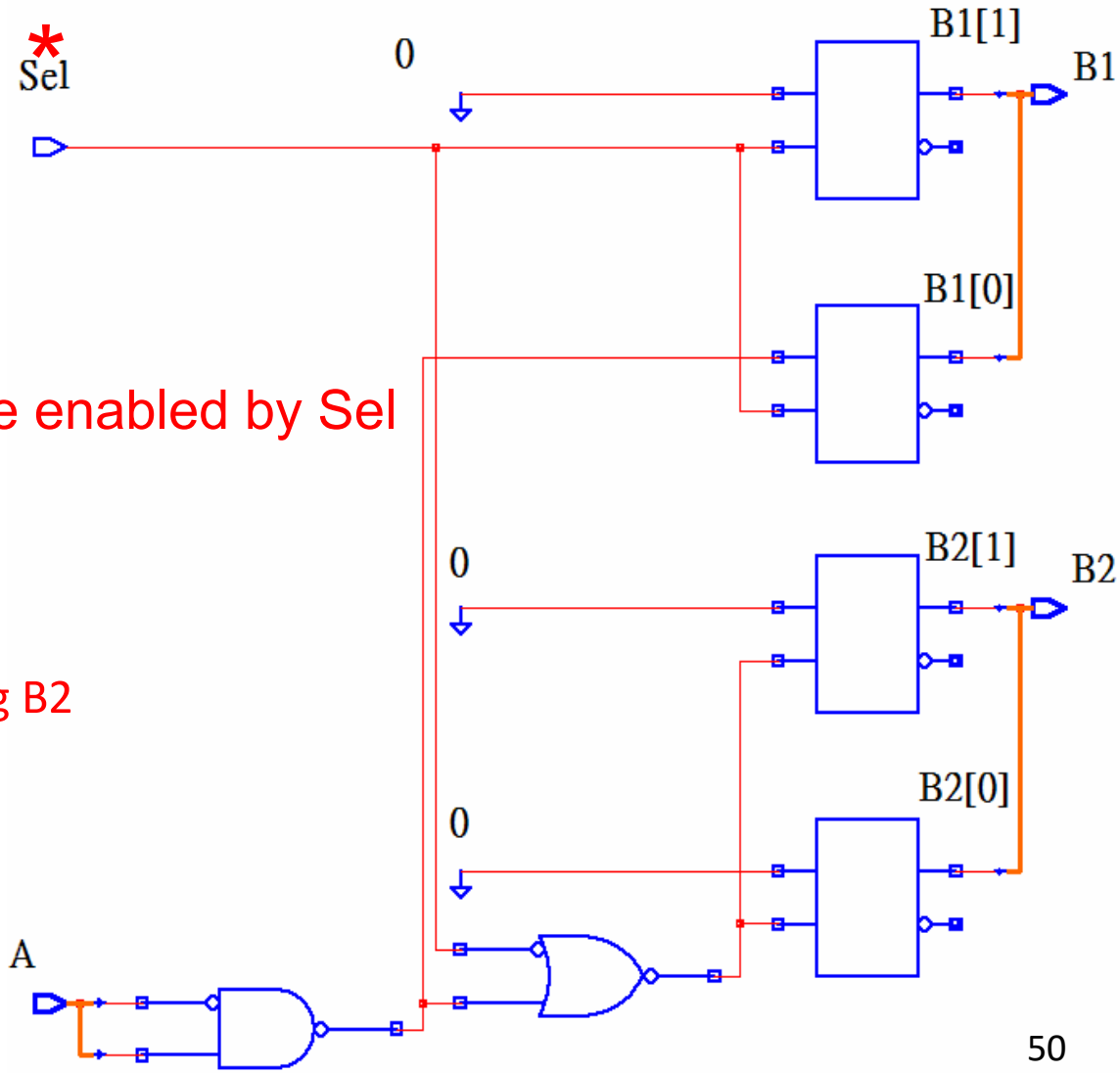
Sel=1, A=1=> B1=0, B2=0

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
endmodule
```

B1 are enabled by Sel

Missing B2



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
```

① ② ③ assigned in order

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
```

① ① ① assigned immediately

Non-blocking assignment

Blocking vs. Non-Blocking (4/10)

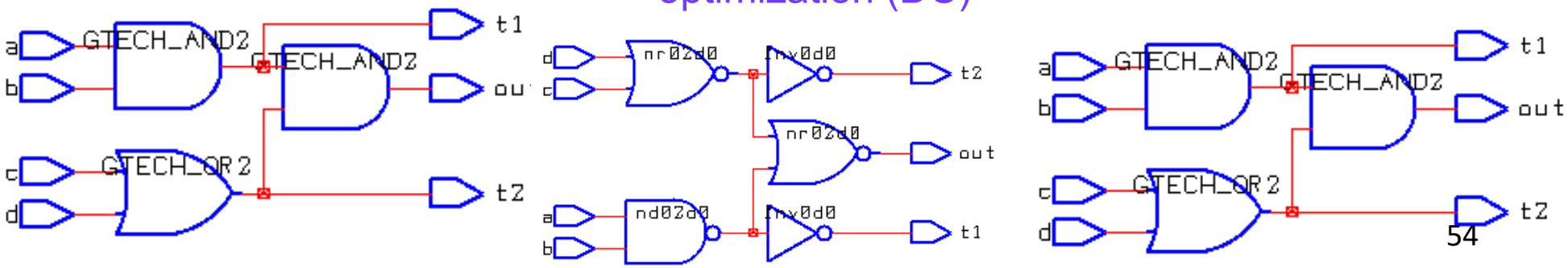
```
module test_n(a, b, c, d, t1, t2, out);
input a, b, c, d;
output out, t1, t2;
reg t1, t2, out;

always @(a or b or c or d)
begin
        t1 = a&b;
        t2 = c | d;
        out = t1 & t2;
end
endmodule
```

Combinational circuit

```
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;
        t2 <= c | d;
        out <= t1 & t2;
end
endmodule
```

Combinational circuit

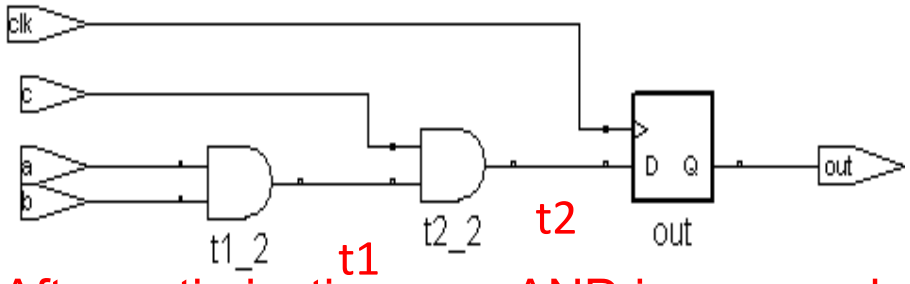
Automatic optimization (DC)



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
```

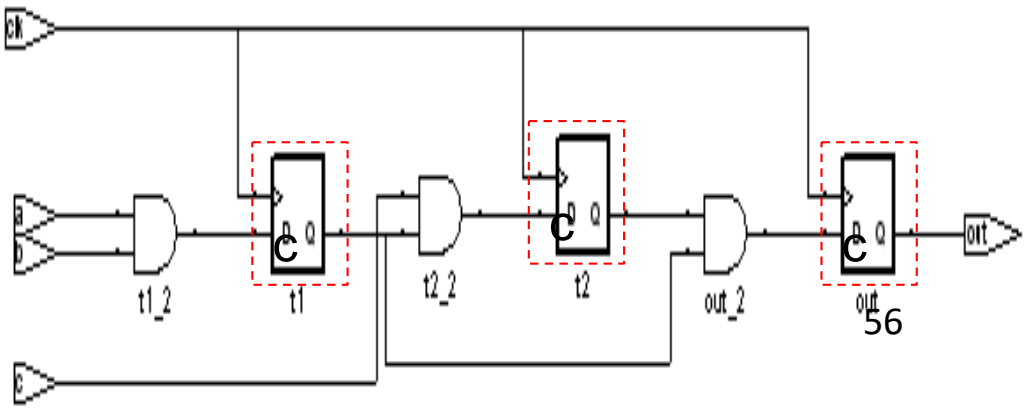


After optimization, one AND is removed

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
```

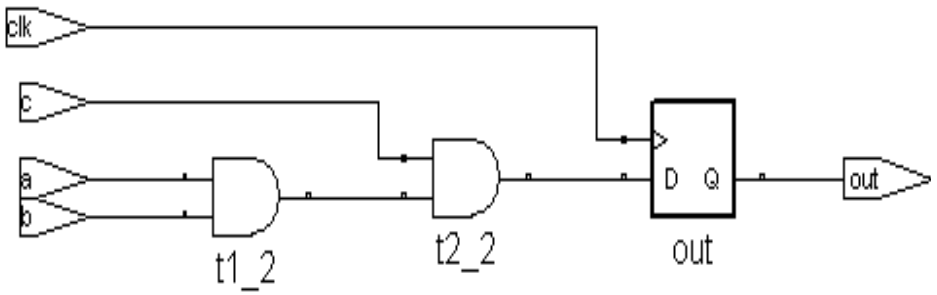
Three flip-flops are inferred



Blocking vs. Non-Blocking (6/10)

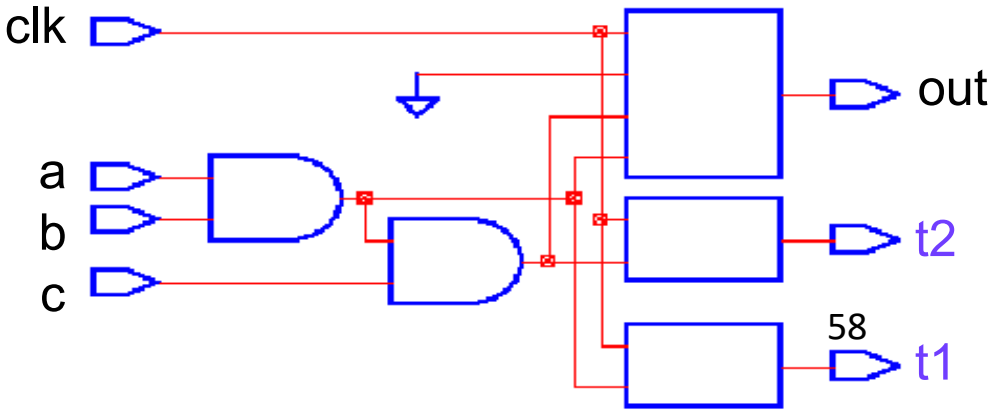
Blocking assignment

```
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 t1,t2 are output

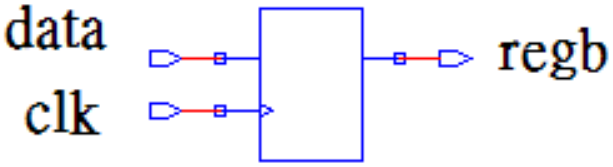
```
module test_n(clk, a, b, c, t1, t2, out);
input clk, a, b, c;  output out, t1, t2;
reg t1, t2;reg out;

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)

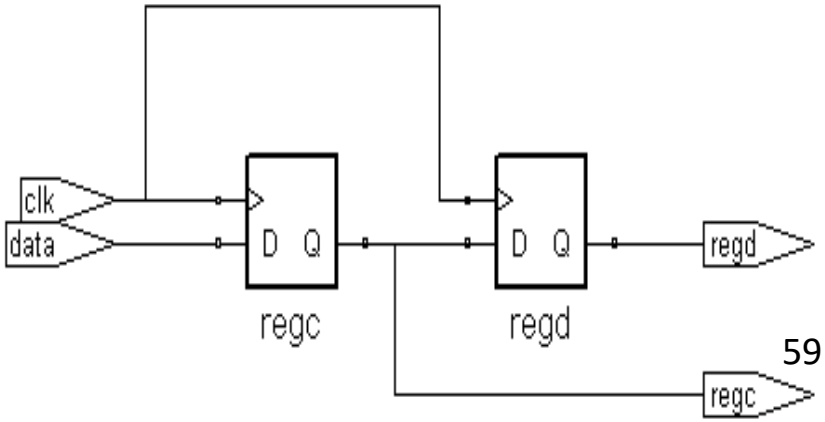
Blocking assignment

```
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

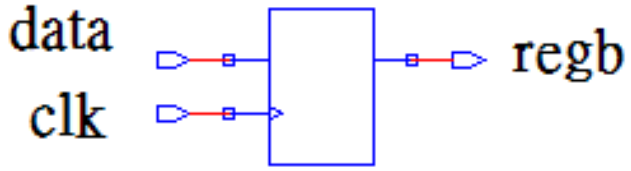
```
module rtl(clk, data, regc, regd);
input  data, clk;
output regc, regd;
reg    regc, regd;

always @(posedge clk)
begin
  regc <= data;   ①
  regd <= regc;   ①  // old regc is used
end

endmodule
```



Blocking vs. Non-Blocking (8/10)

Blocking assignment

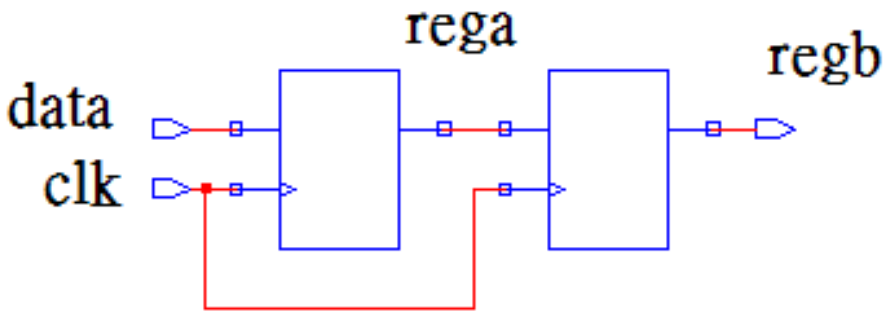
```
module rtl_1(clk, data, regb);
input  data, clk;
output regb;
reg    rega, regb;

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
```

Order Dependence



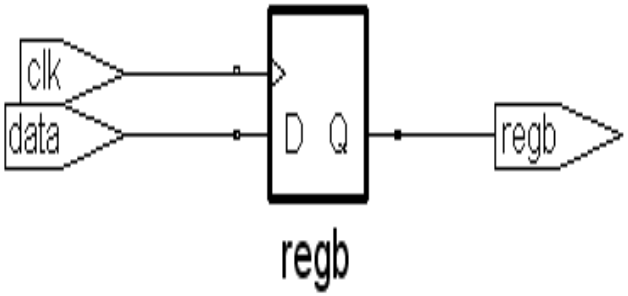
Order is important in blocking assignment

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

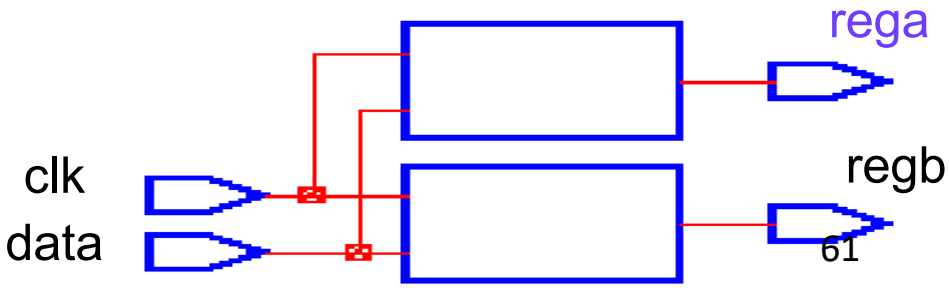


Blocking assignment

```
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  // regb=rega
end

endmodule
```

Rega is output port

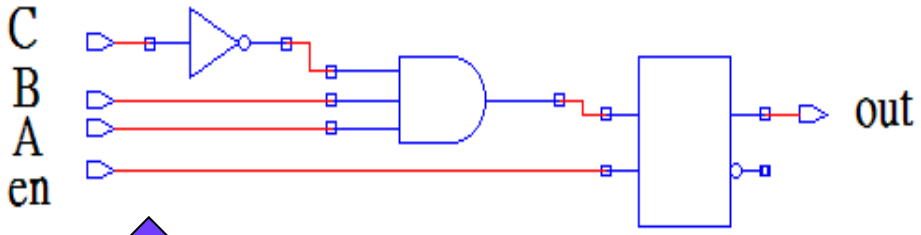
Two flip-flops are inferred



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
```

<=nonblocking

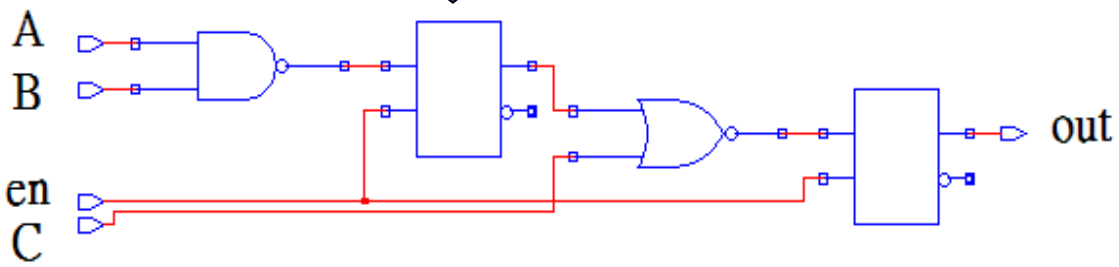


One latch is inferred

```
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
```

<=blocking

Two latches are inferred

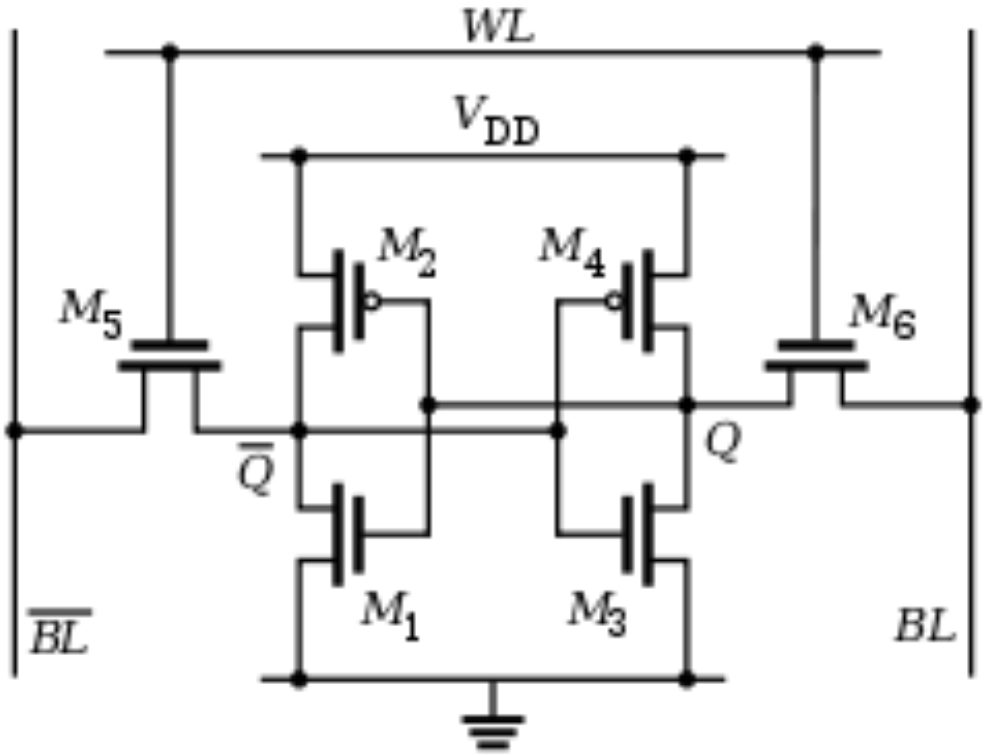


Registers

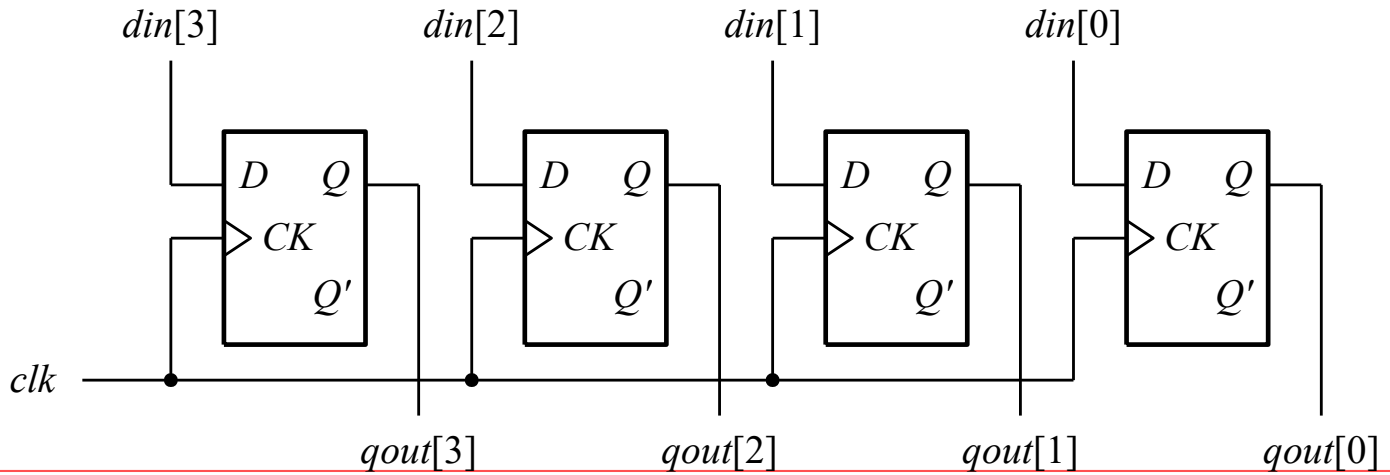
- Registers provide a means for storing information in any digital system.
- Types of registers used in digital systems:
  - Data register using flip-flops or latches.
  - Register file being used in data paths.
  - Synchronous RAM (random access memory) being used for storing moderate amount of information.
- A flip-flop may take up 10 to 20 times the area of a 6-transistor static RAM cell.

SRAM Cell

- 6-T SRAM Cell can 1-bit data (much smaller than a flip flop)



Data Registers




```
// an n-bit data register
module register(clk, din, qout);
parameter N = 4;   // number of bits
input   clk;
input   [N-1:0] din;
output reg [N-1:0] qout;
// the body of an n-bit data register
always @(posedge clk) qout <= din;
endmodule
```

4-bit register = 4 flip-flops

N-bit data register with asynchronous reset

```
module register_reset (clk, reset_n, din, qout);
parameter N = 4;   // number of bits
input   clk, reset_n;
input   [N-1:0] din;
output reg [N-1:0] qout;

// The body of an n-bit data register
always @(posedge clk or negedge reset_n)
  if (!reset_n) qout <= {N{1'b0}};
  else          qout <= din;
endmodule
```

Asyn Reset

Coding style:

- For active-low signal, end the signal name with an underscore followed by a lowercase letter b or n, such as reset_n.

N-bit data register with synchronous load and asynchronous reset

```
module register_load_reset (clk, load, reset_n, din, qout);
parameter N = 4;   // number of bits
input  clk, load, reset_n;
input  [N-1:0] din;
output reg [N-1:0] qout;

// the body of an N-bit data register
always @(posedge clk or negedge reset_n)
  if (!reset_n)    qout <= {N{1'b0}};
  else  if (load)  qout <= din;
  else qout <= qout;
endmodule
```

synchronous load

N-word register file with one-write and two-read ports

```
module register_file(clk, rd_addra, rd_addrb, wr_addr, wr_enable, din, douta, doutb);
parameter M = 4;   // number of address bits
parameter N = 16;  // number of words, N = 2**M
parameter W = 8;   // number of bits in a word
input  clk, wr_enable;
input   [W-1:0]   din;
output [W-1:0]   douta, doutb;
input   [M-1:0]   rd_addra, rd_addrb, wr_addr;
reg     [W-1:0]   reg_file [N-1:0];

// the body of the N-word register file
assign douta = reg_file[rd_addra],
         doutb = reg_file[rd_addrb];
always @(posedge clk)
  if (wr_enable) reg_file[wr_addr] <= din;
endmodule
```

Two read ports

Write port

An Synchronous RAM

- A Synchronous RAM is a random-access memory where data access is synchronism with a clock signal being applied to the memory
- Use block memory (memory micro) to reduce area

```
// a synchronous RAM module example
module syn_ram (addr, cs, din, clk, wr, dout);
parameter N = 16;   // number of words
parameter A = 4;    // number of address bits
parameter W = 4;    // number of wordsize in bits
input  [A-1:0]  addr;
input  [W-1:0] din;
input   cs, wr, clk;   // chip select, read-write control, and clock signals
output reg [W-1:0] dout;
reg    [W-1:0] ram [N-1:0];  // declare an N * W memory array

always @(posedge clk) // the body of synchronous RAM
    if (cs) if (wr) ram[addr] <= din;
            else            dout <= ram[addr];
endmodule
```

One write port

One read ports

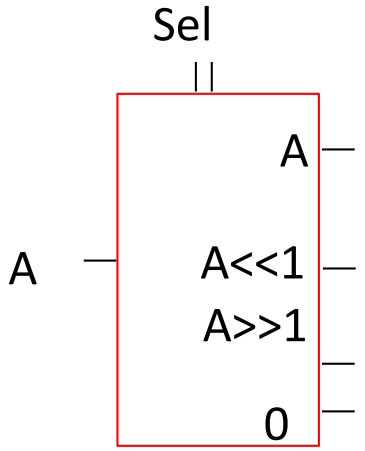
Shift Registers

- Shift registers perform left or right shift operation.
- Parallel/serial format conversion:
  - SISO (serial in serial out)
  - SIPO (serial in parallel out)
  - PISO (parallel in serial out)
  - PIPO (parallel in parallel out)

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← A	no shift
1	Y← shl A	shift left
2	Y← shr A	shift right
3	Y← 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;
```

Sel	Operation	Function
0	Y← A ShiftLeftOut←0 ShiftRightOut←0	no shift
1	Y← shl A ShiftLeftOut←A[5] ShiftRightOut←0	shift left
2	Y← shr A ShiftLeftOut←0 ShiftRightOut←A[0]	shift right
3	Y← 0 ShiftLeftOut←0 ShiftRightOut←0	zero outputs

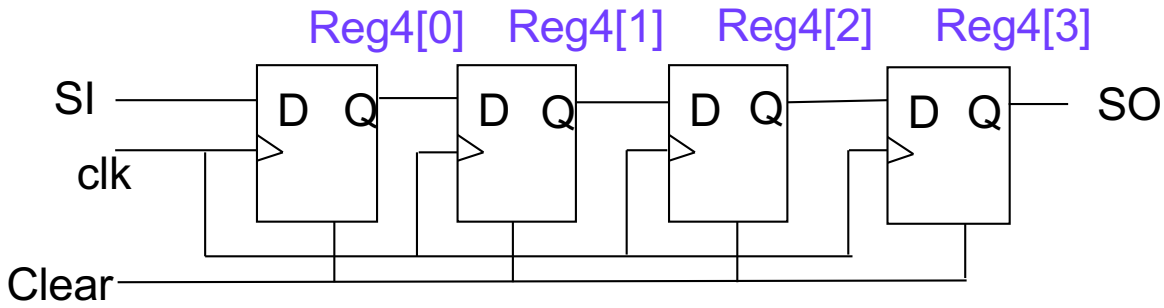
```
always@(Sel or ShiftLeftIn or
ShiftRightIn or A)
begin
    A_Wide={ShiftLeftIn,A,ShiftRightIn};

    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
```

SISO Shifter (1/4)

sequential shifter

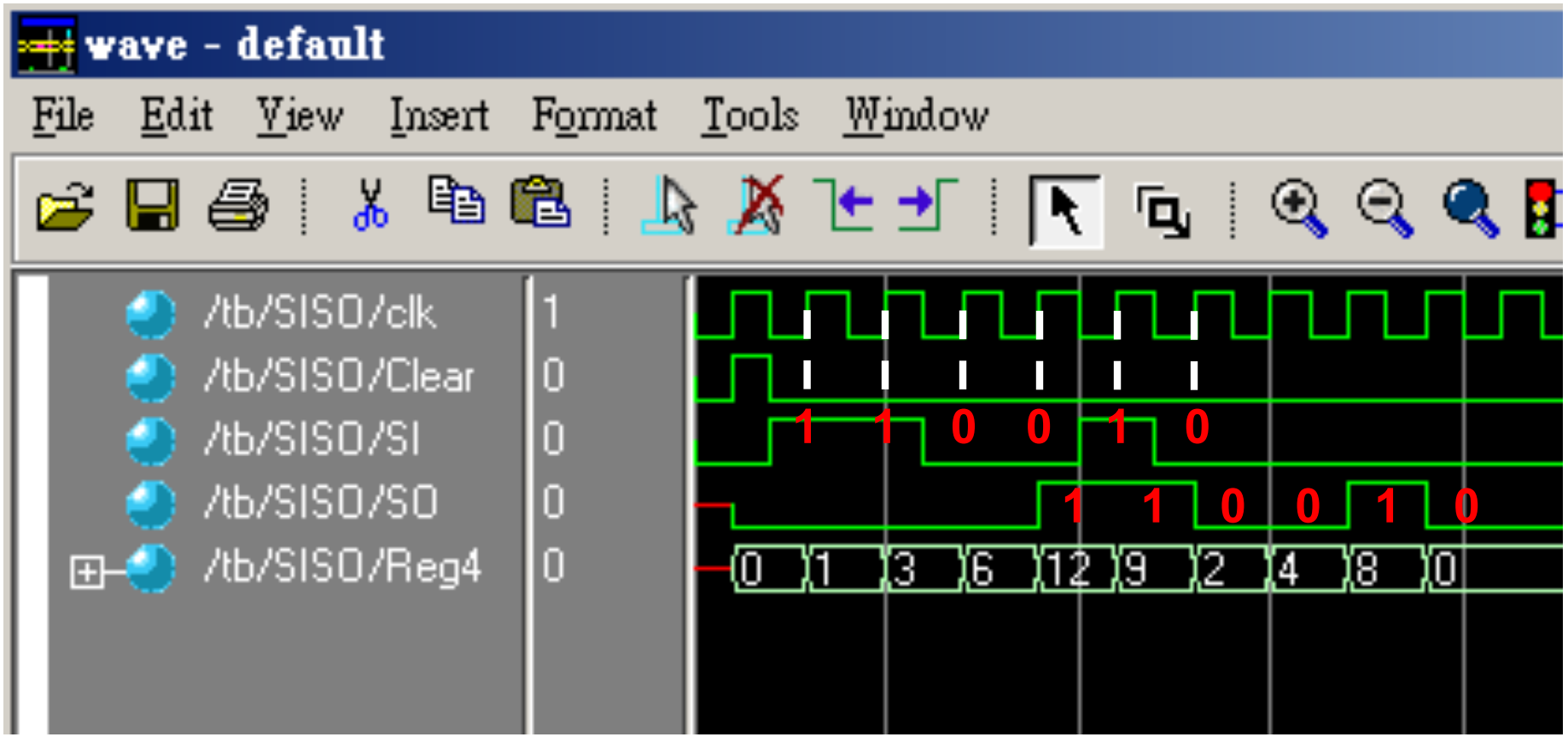
serial in serial out

```
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[3] = Reg4[2];
      Reg4[2] = Reg4[1];
      Reg4[1] = Reg4[0];
      Reg4[0] = SI;
          end
  end
  assign SO = Reg4[3];
endmodule
```



Data appear on SO after 4 clocks

SISO Shifter (2/4)



Data appear on SO after 4 clocks

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
```

Using if & for

```
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
```

Using <<

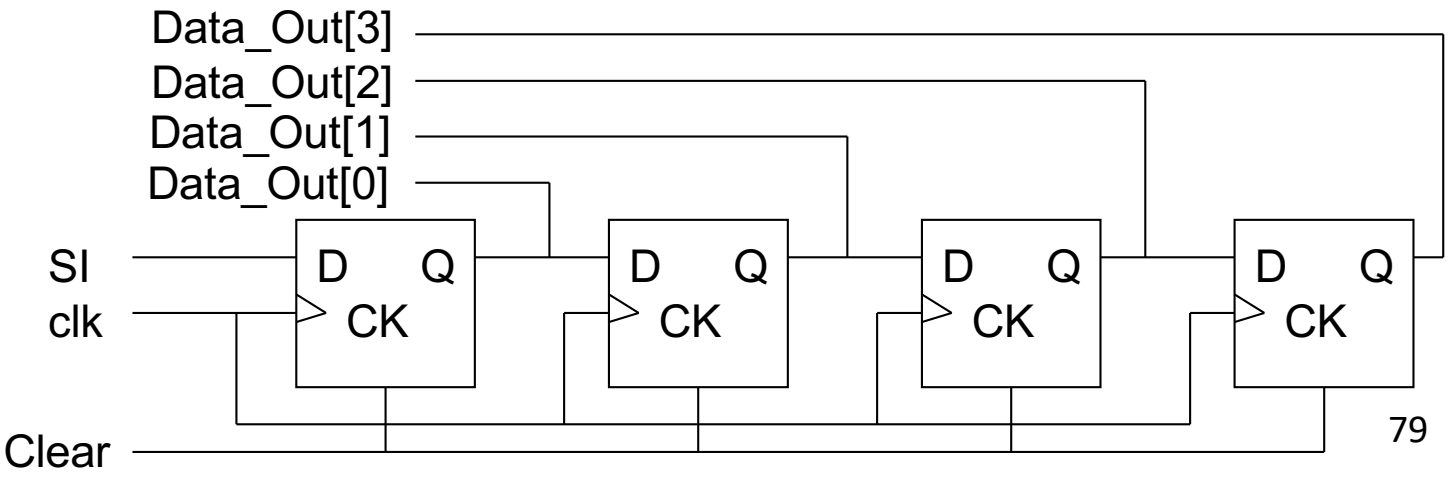
Using <<

SIPO Shifter (1/2)

serial in parallel out

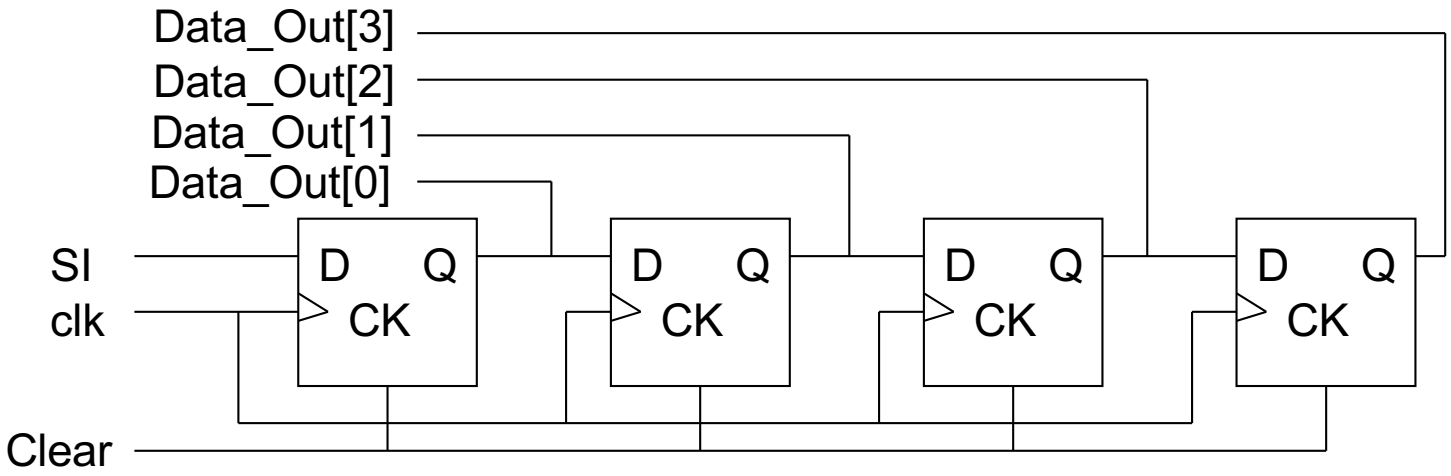
```
module  SIPO__SR(clk, Clear, SI, Data_Out);
input     clk, Clear, SI;
output    [3:0] Data_Out;
reg       [3:0] Data_Out;

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
endmodule
```

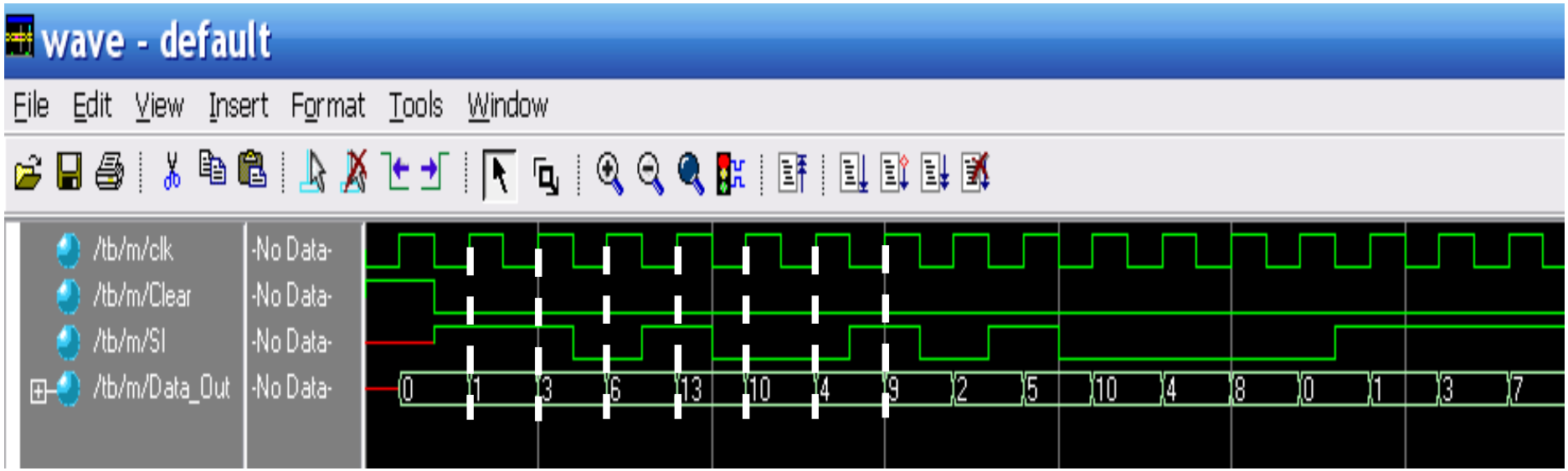
Parallel Out



Data_Out has 4 bits

SIPO Shifter (2/2)





1 1 0 1 0 0 1

13

10

4

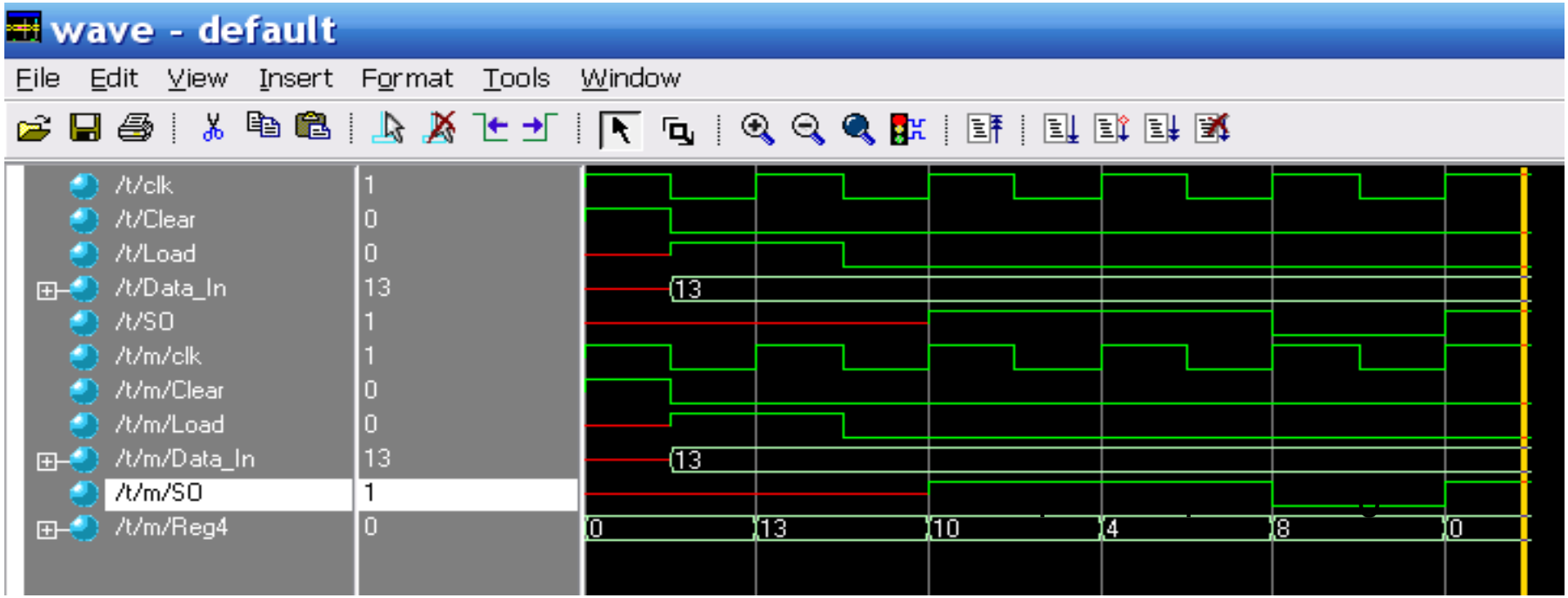
PISO (1/2)

parallel in serial out

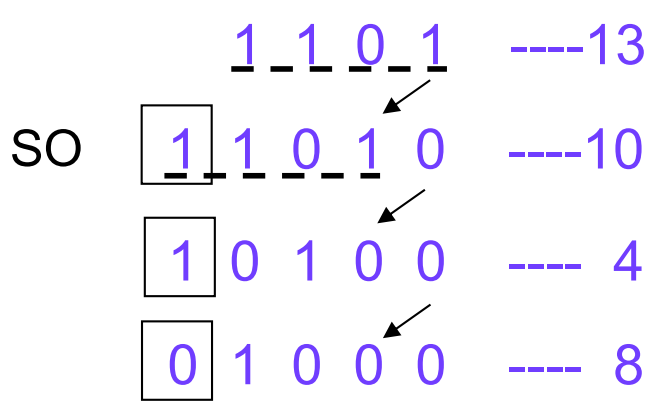
```
module  PISO__SR (clk, Clear, Load, Data_In, SO);
input          clk, Clear, Load;
input    [3:0] Data_In;
output         SO;
reg            SO;
reg      [3:0] Reg4;

  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
endmodule
```

parallel in serial out

PISO (2/2)



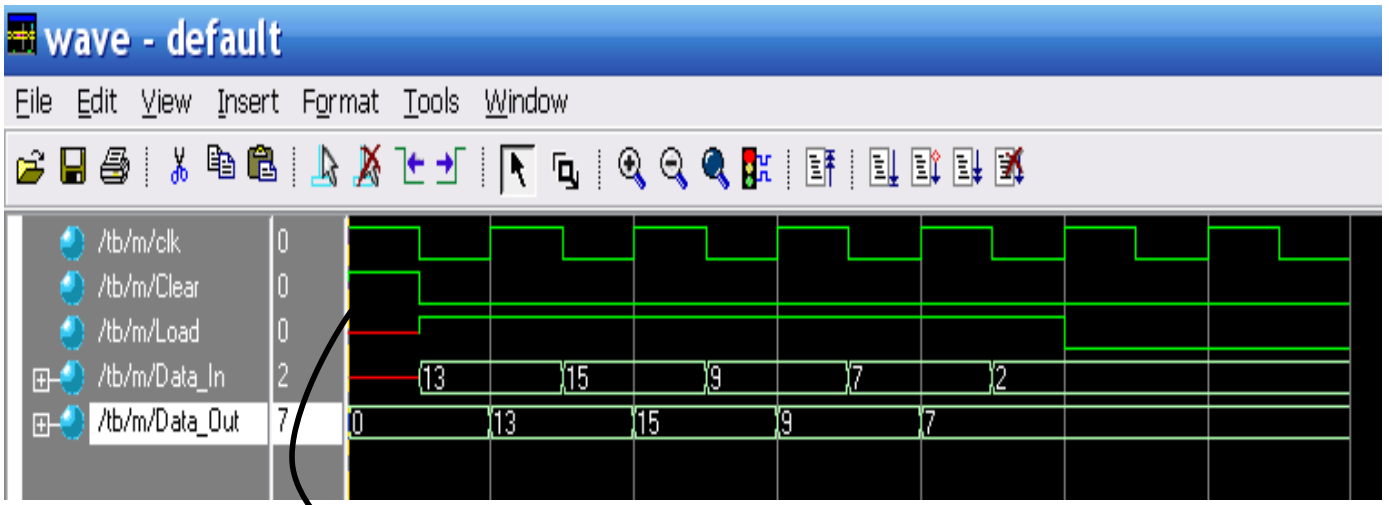
If Load=1, Data_In=13



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
endmodule
```

parallel in parallel out



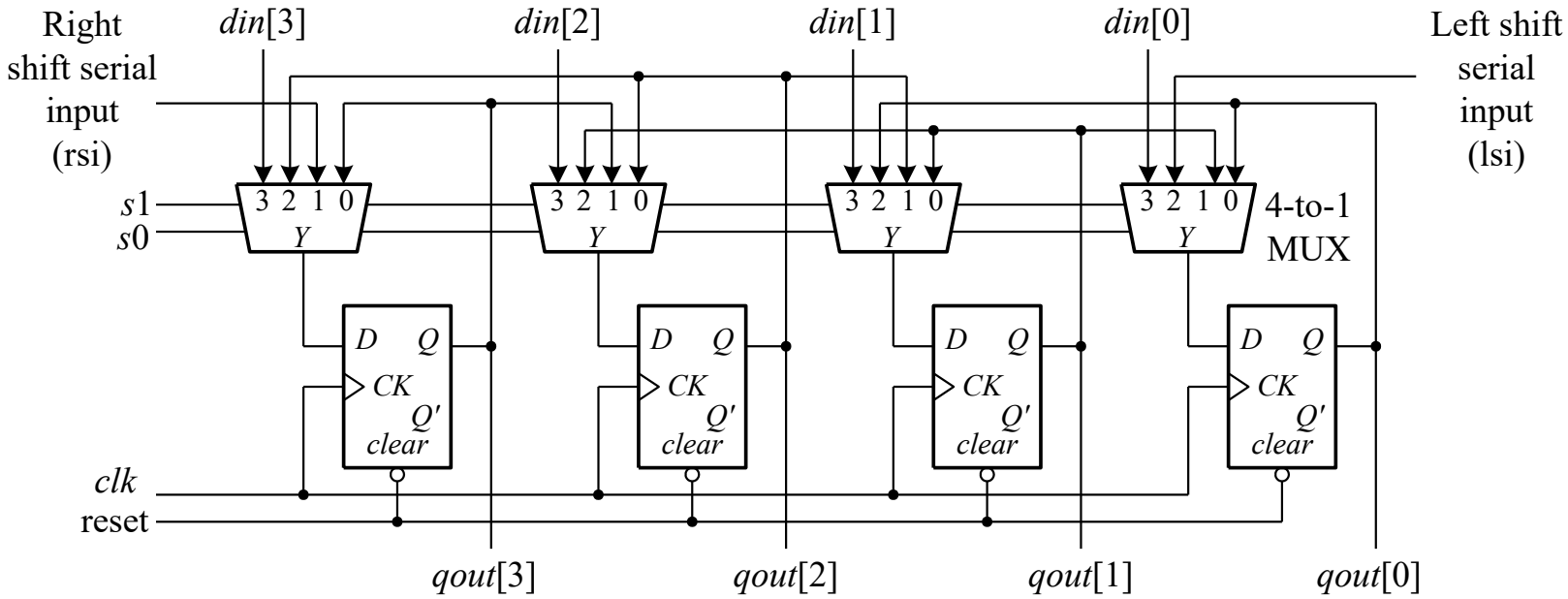
Bad load !! Be Careful

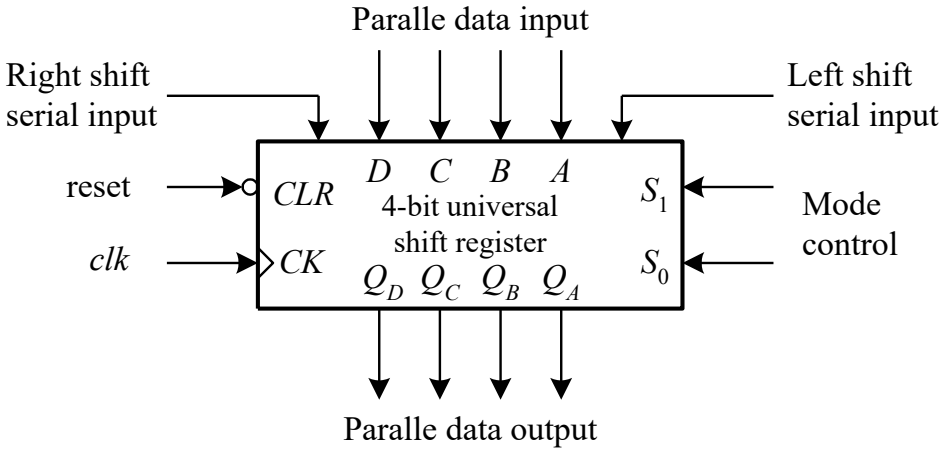
Data_In must be ready before posedge

Universal Shift Registers

- A universal shift register can carry out
  - SISO (serial in serial out)
  - SIPO (serial in parallel out)
  - PISO (parallel in serial out)
  - PIPO (parallel in parallel out)
- The register must have the following capabilities:
  - Parallel load
  - Serial in and serial out
  - Shift left and shift right

Universal Shift Registers





$s1$	$s0$	Function
0	0	No change
0	1	Right shift
1	0	Left shift
1	1	Load data

Universal Shift Registers

```
module universal_shift_register (clk, reset_n, s1, s0, lsi, rsi, din, qout);
parameter N = 4;  // define the size of the universal shift register
input  s1, s0, lsi, rsi, clk, reset_n;
input  [N-1:0] din;
output reg [N-1:0] qout;
// the shift register body
always @(posedge clk or negedge reset_n)
  if (!reset_n) qout <= {N{1'b0}};
  else case ({s1,s0})
       2'b00: ; // qout <= qout;              // No change
       2'b01: qout <= {lsi, qout[N-1:1]};   // Shift right
       2'b10: qout <= {qout[N-2:0], rsi};   // Shift left
       2'b11: qout <= din;                  // Parallel load
  endcase
endmodule
```

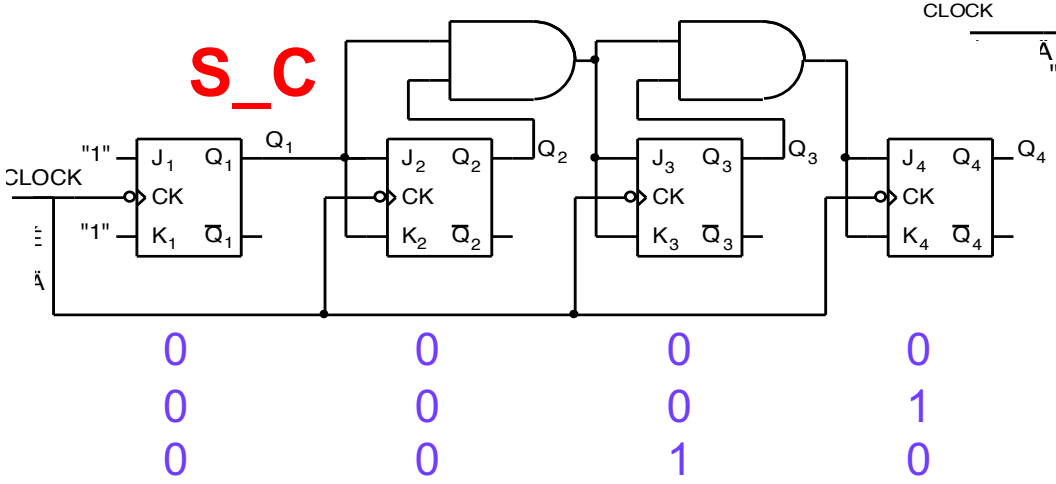
Synchronous/Asynchronous Counter

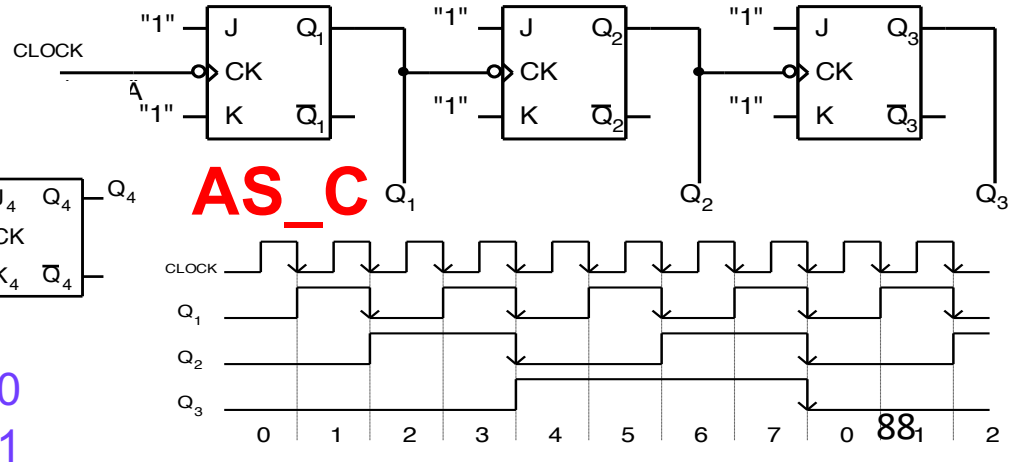
Synchronous counter:

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

Asynchronous (Ripple) counter:

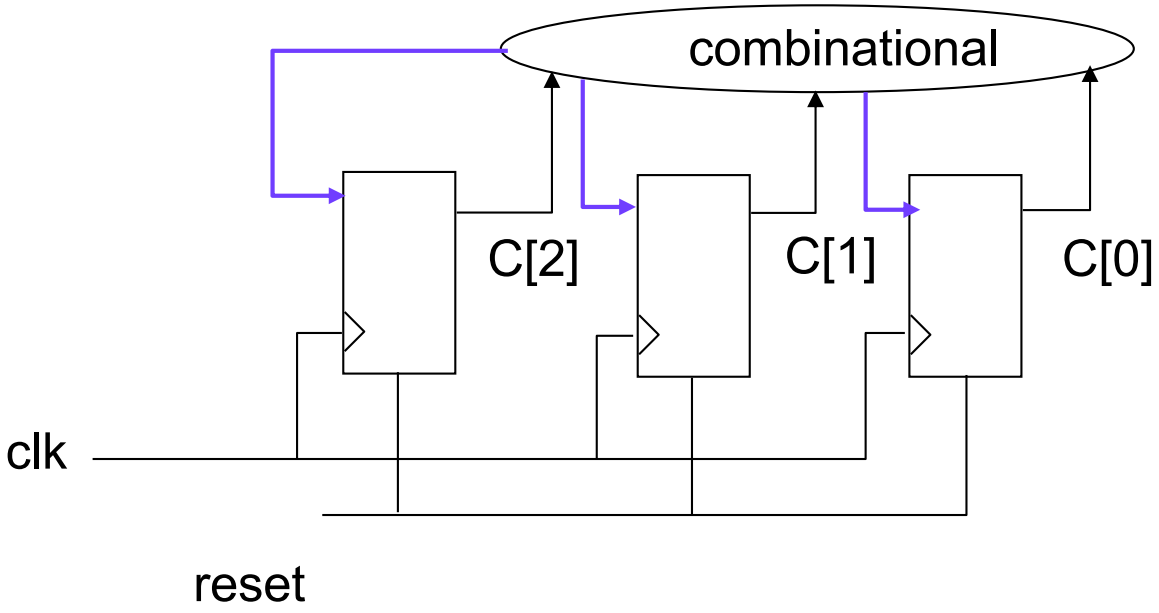
Flip-flops transitions ripple through from one flip-flop to the next in sequence until all flip-flops reach a new stable value (state). Each single flip-flop stage divides the frequency of its input signal by two.





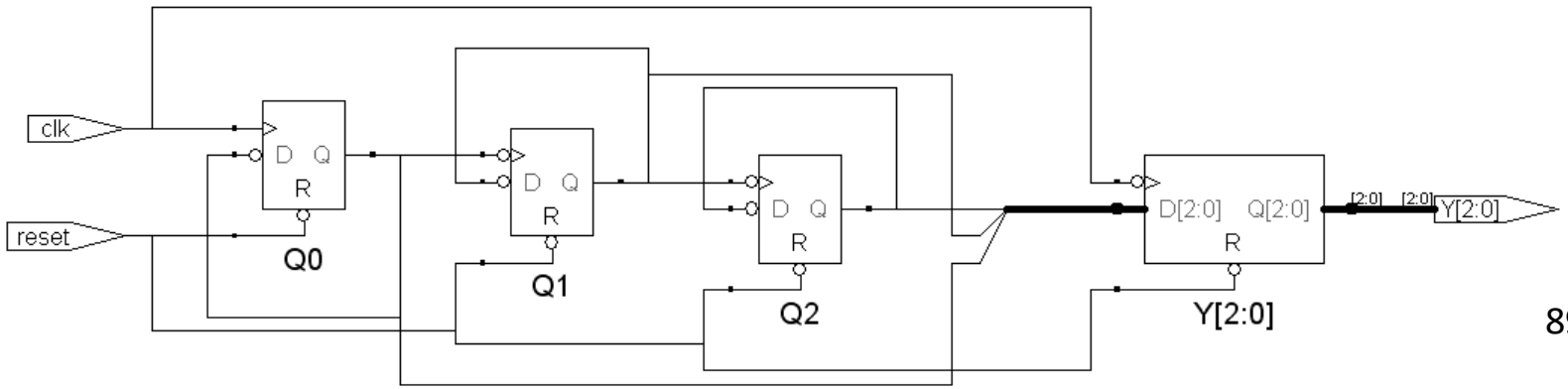
Counter Implementation

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

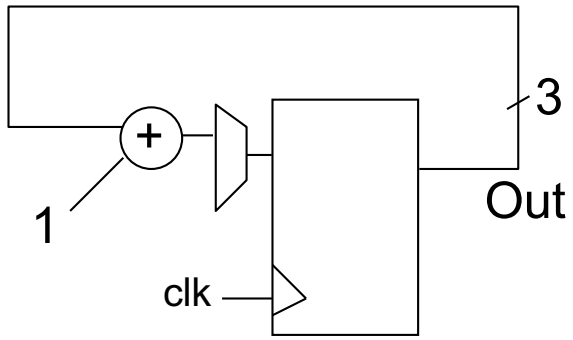


Synchronous Counter(1/6)

```
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
```

Two additional signal:
Reset, Load and Enable

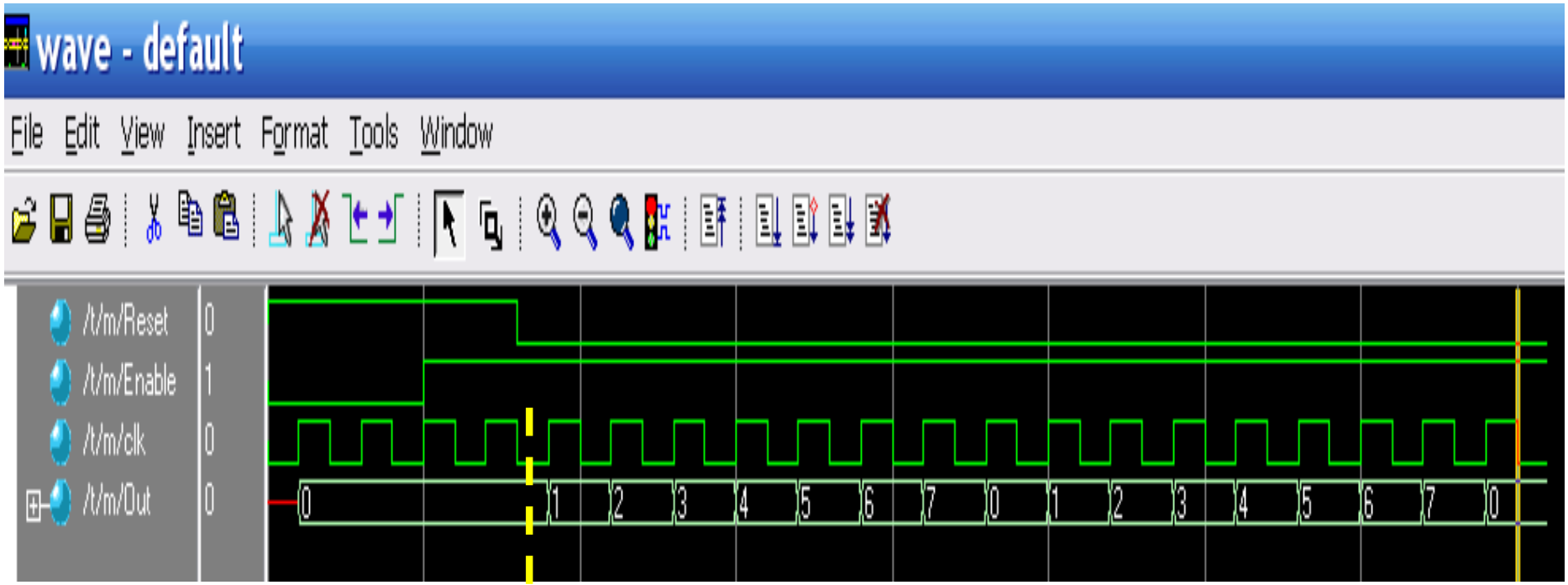
⇦ **What happens if Out== 3'd5 ??**



Reset=1 Out=000

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

Synchronous Counter(2/6)



Only when Reset is low and Enable is active, then, the counter will begin count up.

Synchronous Counter (3/6)

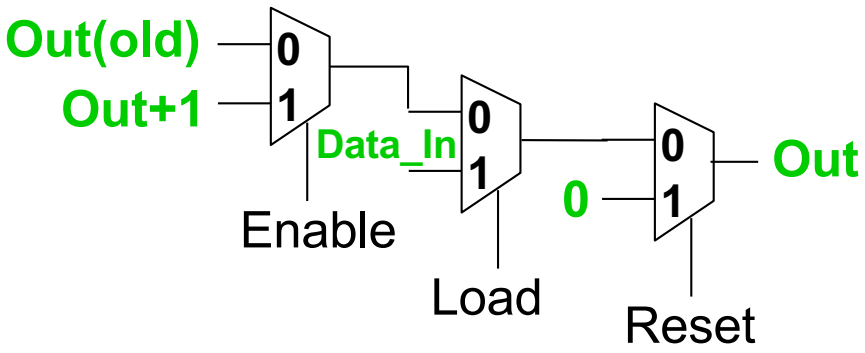
```
module  Counter2 (clk, Reset, Load, Enable, Data_In, Out);
input       clk, Reset, Load, Enable;
input    [7:0] Data_In;
output   [7:0] Out;
reg      [7:0] Out;

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

Three additional signal: Reset, Load and Enable

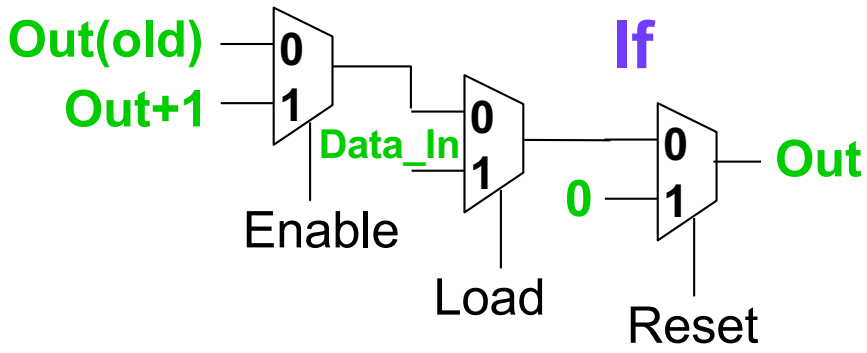
Reset=1 Out=00000000

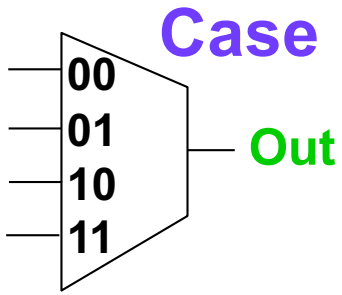
Reset=0, Load=1, Out=Data_In

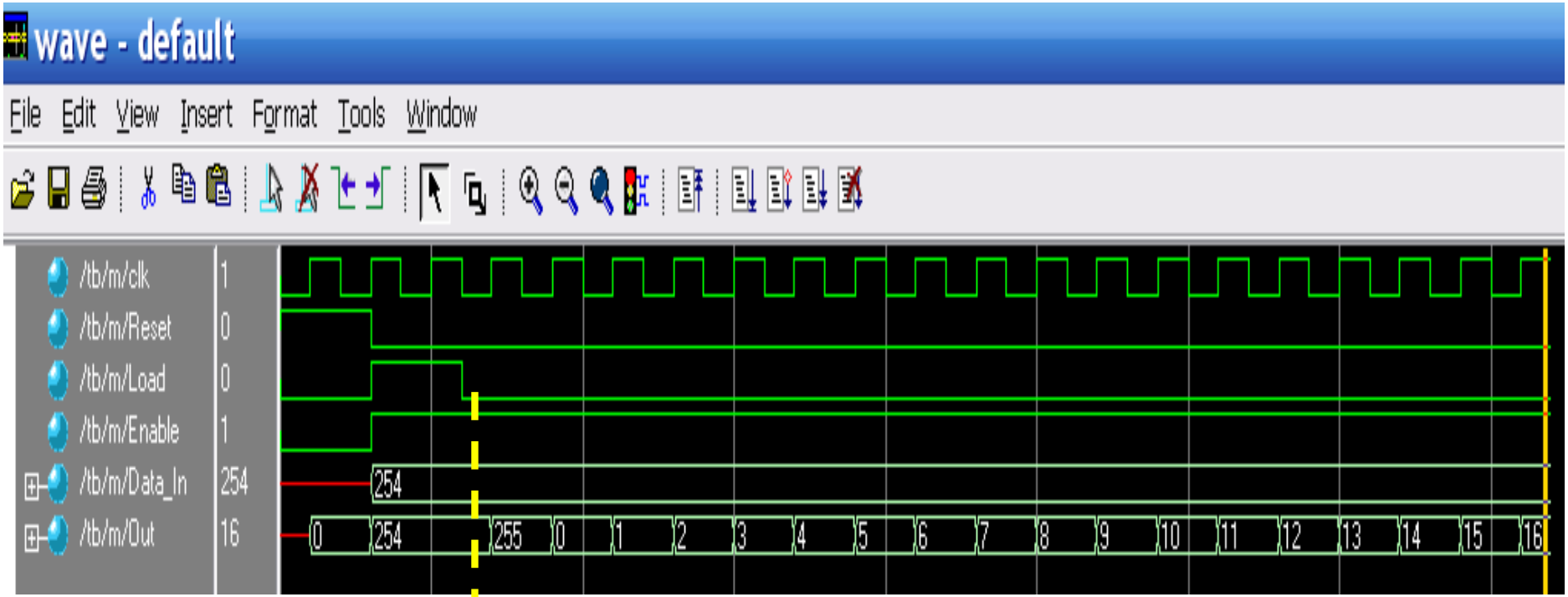
Enable==1, Out=x ➔x+1➔…..➔255➔0
➔1➔……➔255➔0➔…



Synchronous Counter(4/6)







When Reset =0, Load=0, Enable=1, the counter will begin to count up.

Synchronous Counter(5/6)

Up_Down to determine “count up” or “count down”

```
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
```

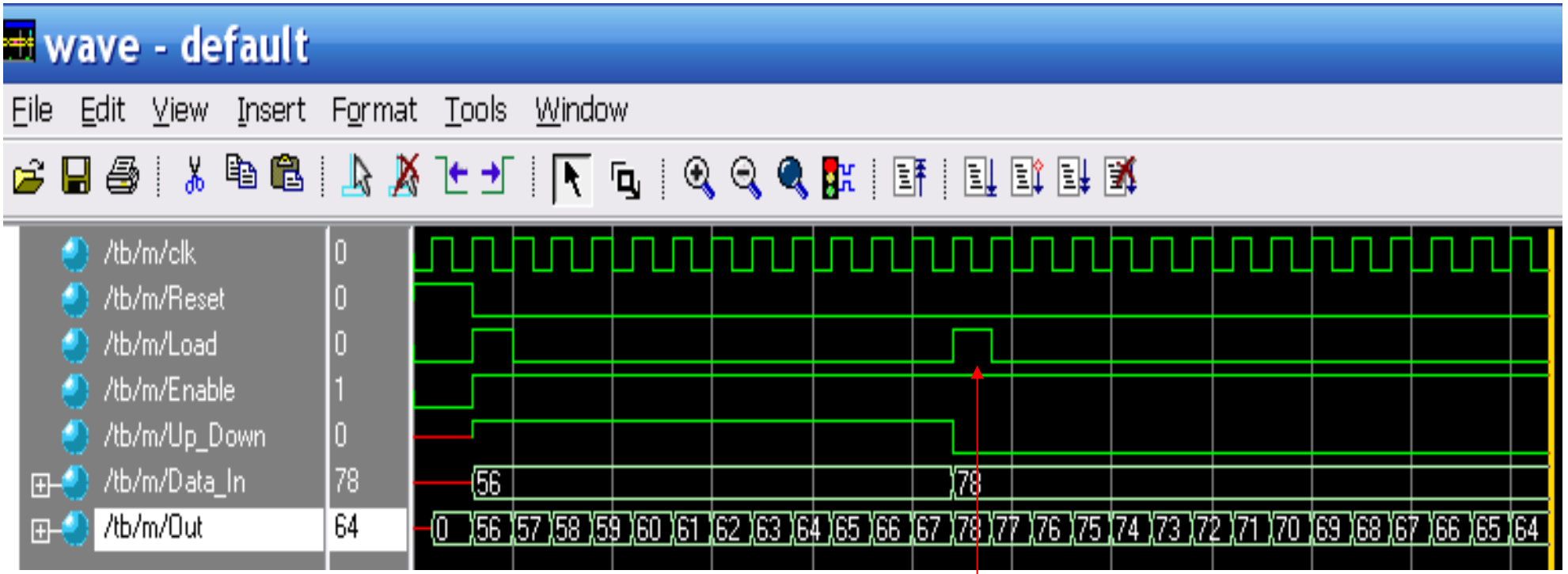
```
if (Up_Down)
          Out = Out + 1;
        else
          Out = Out - 1;
      end
  end
endmodule
```

If down-by-two

Out=Out-2;

Synchronous Counter(6/6)

Up_down=1 => increase, Up_down=0 => decrease



Up_down=1 => increase

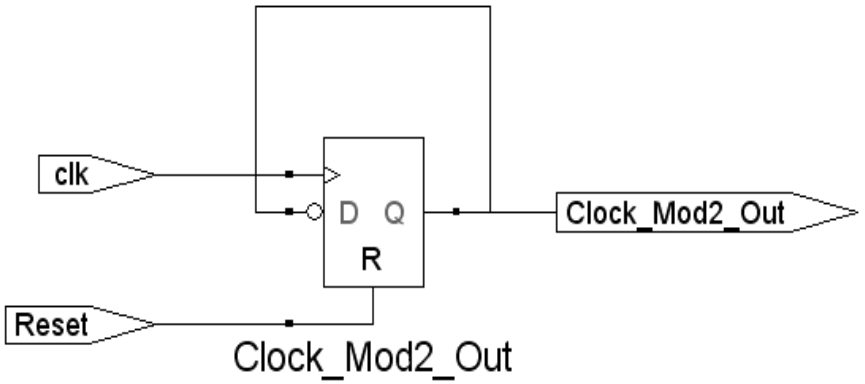
Up_down=0 => decrease

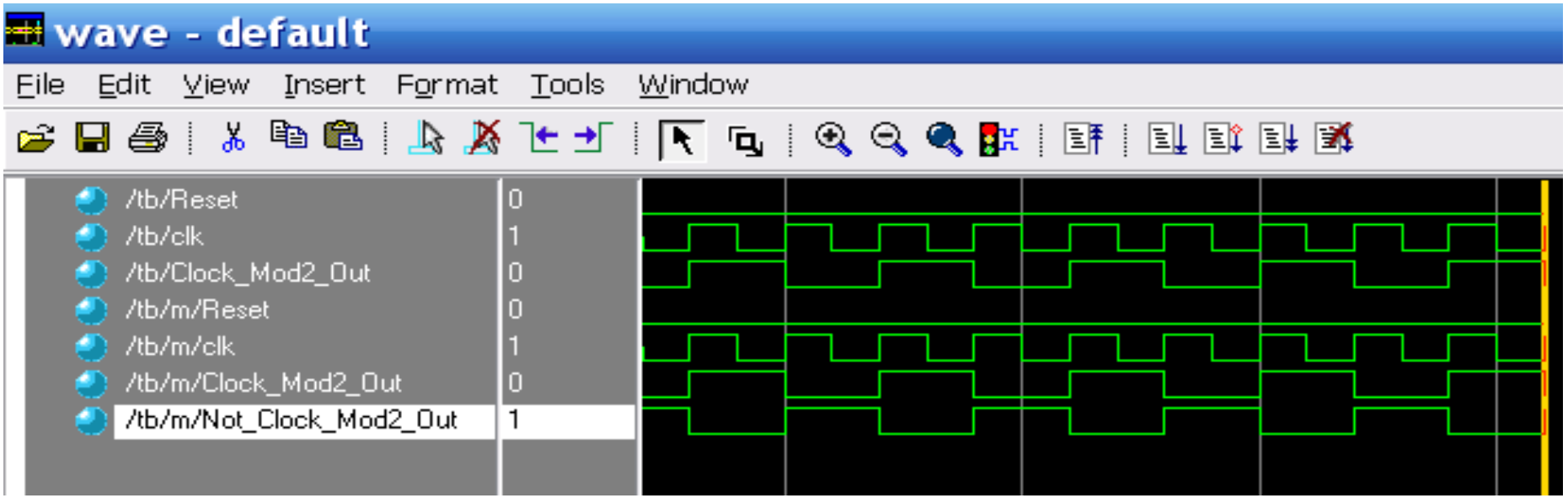
Load =1 => Load data

Asynchronous Counter(1/8)- FreqMod2

```
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;

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
```

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





Asynchronous Counter(2/8)-countsdown

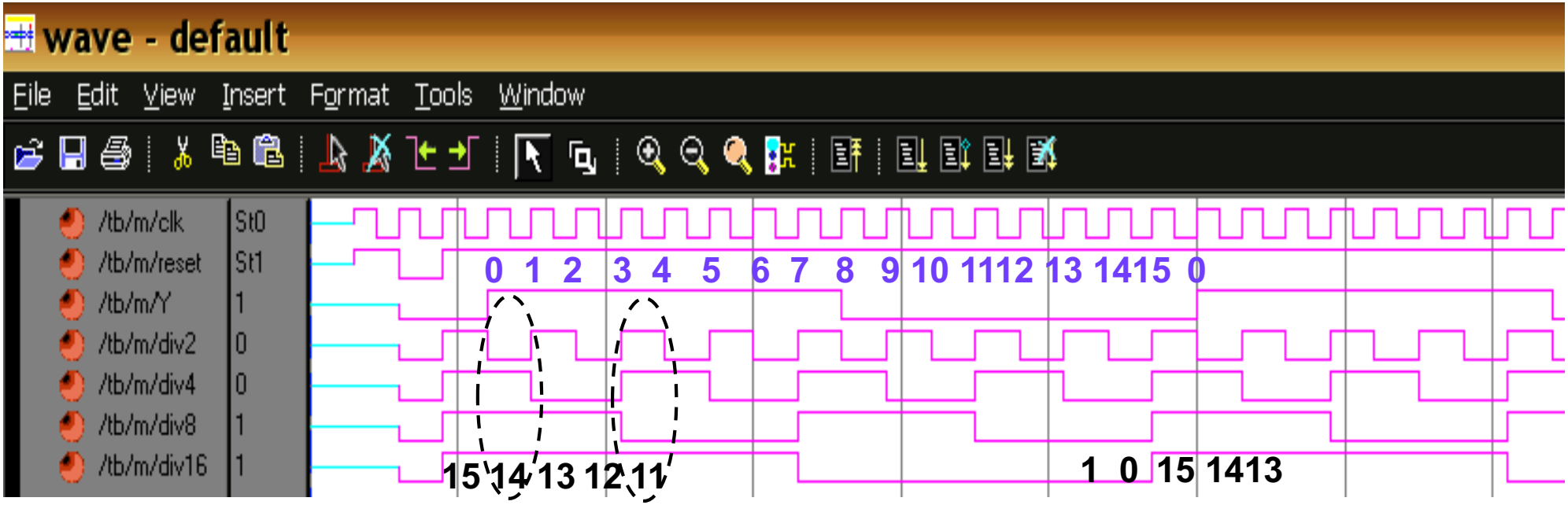
Divide by 16 clock divider using an asynchronous (ripple) counter

➔ ***frequency divider***

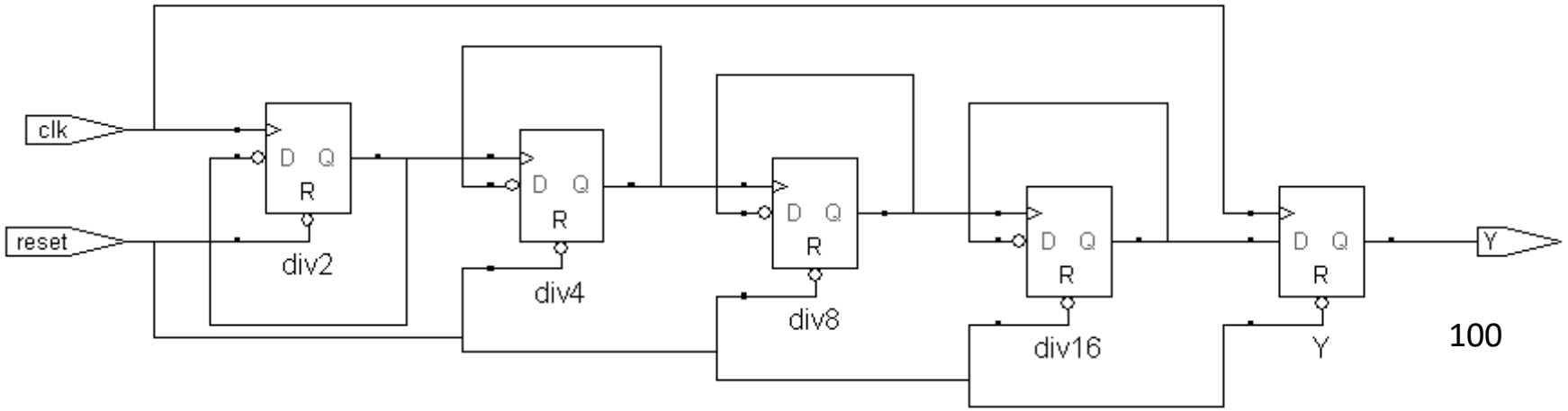
Count-down counter

```
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(2/8)-countdown



Count-down counter 15➔14➔13➔…….1➔0➔15➔14➔…..



Asynchronous Counter(6/8)-div by 13

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

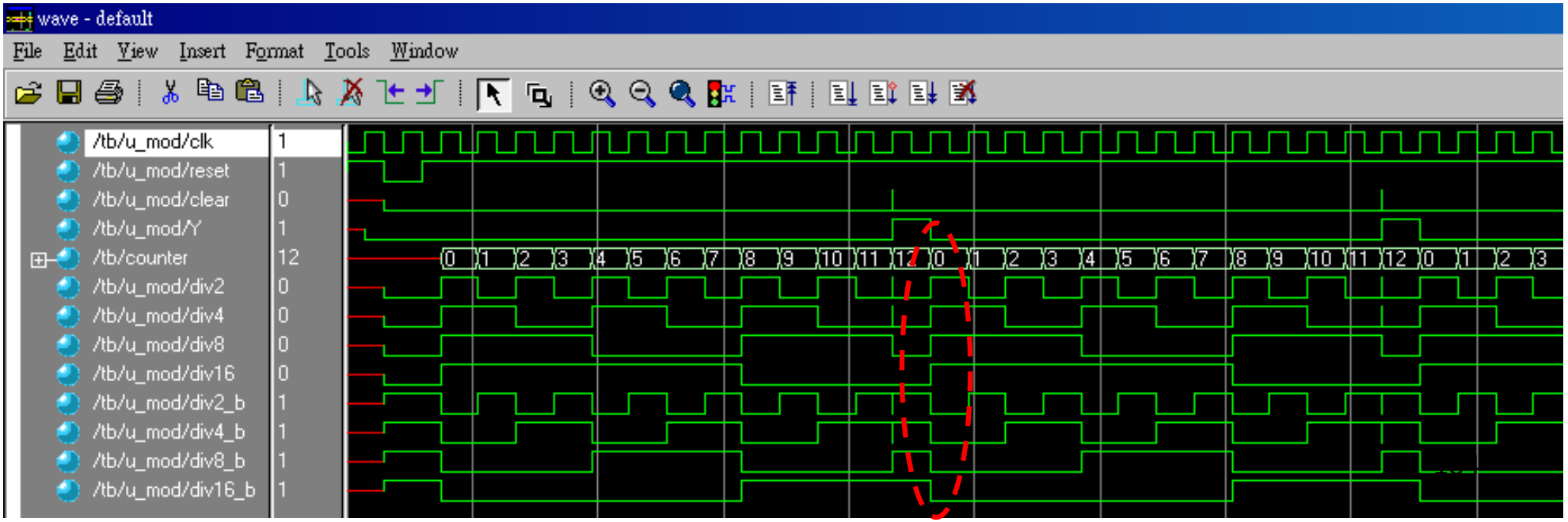
```
module CNT_ASYNC_CLK_DIV13(clk,reset,Y);
    input  clk,reset;  output Y;
    reg  div2,div4,div8,div16,Y;
    wire div2_b,div4_b,div8_b,div16_b,clear;
always@(posedge clk or negedge reset
      or posedge clear)
        if(!reset)
                div2=0;
        else if(clear)
                div2=0;
        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)-div by 13

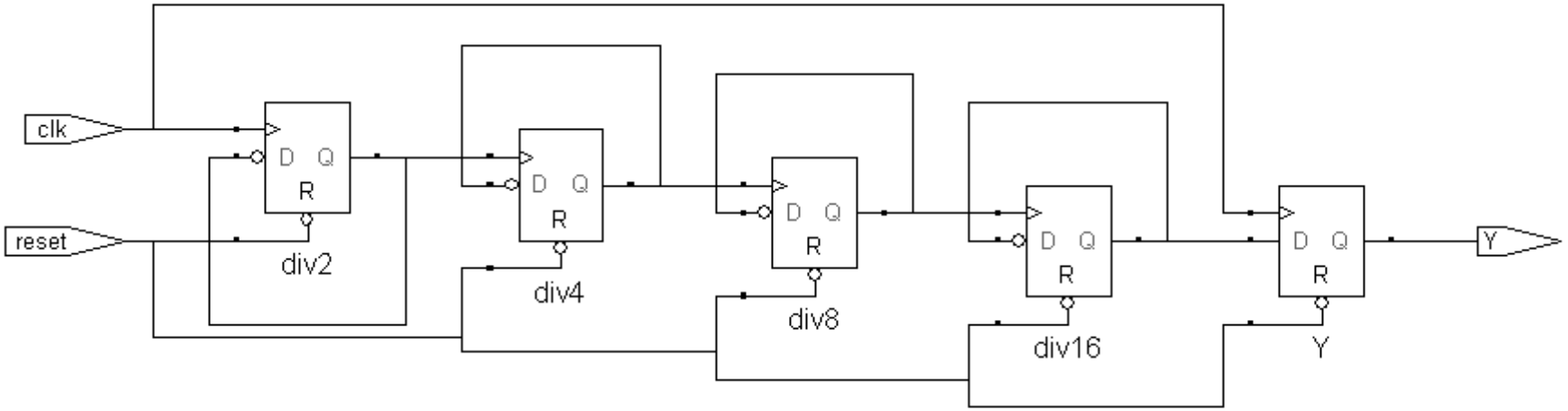
```
always@(posedge clk or negedge reset)
  if(!reset)
     Y=0;
  else if({div16_b,div8_b,div4_b,div2_b}==11)
     Y=1;
  else
     Y=0;
  end
```

```
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
```

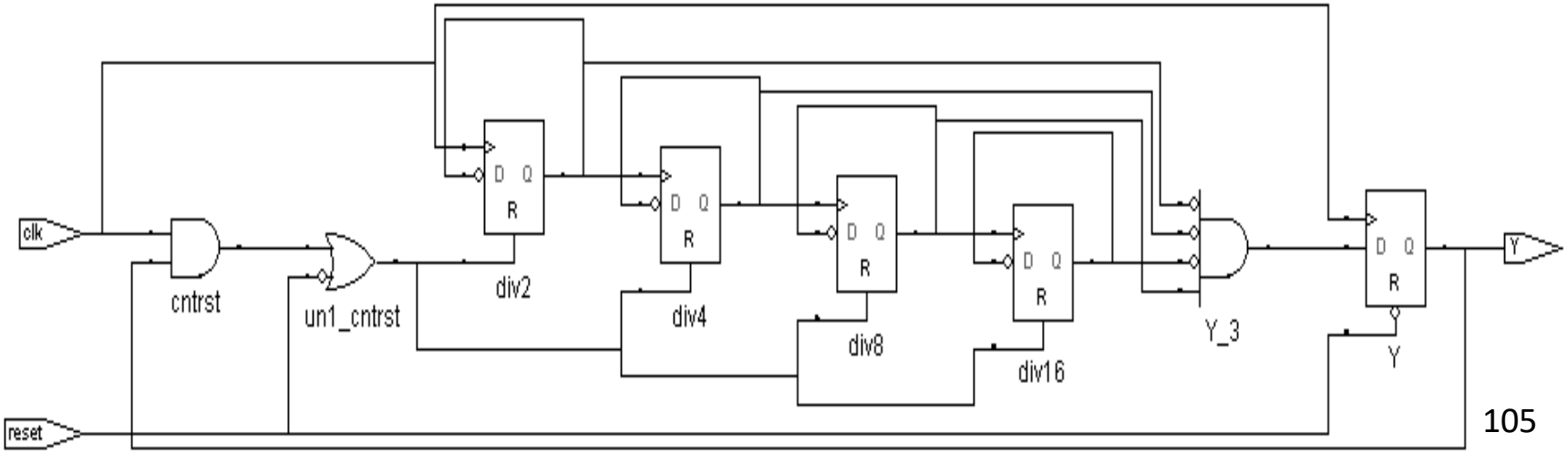


Asynchronous Counter(8/8)

Divide by 16

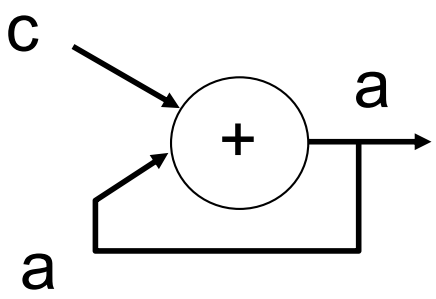


Divide by 13



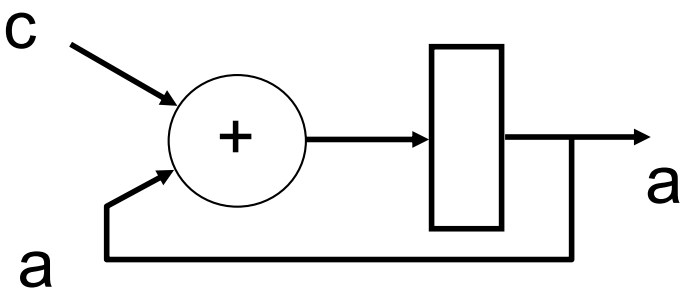
Loop Problem (1/2)

assign a=a+c;



Error!

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

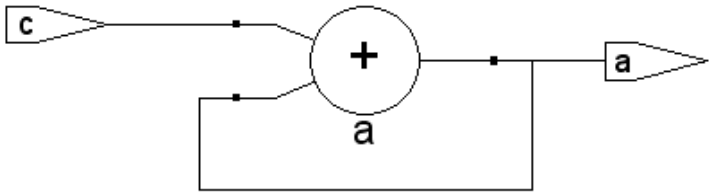


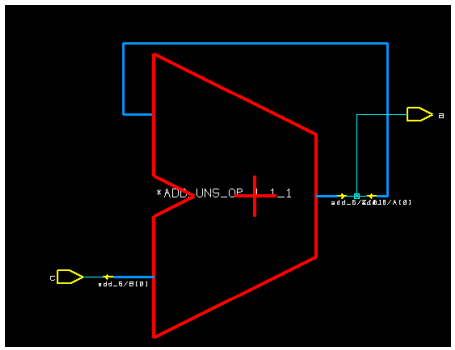
Good!

Why?

Loop Problem (2/2)

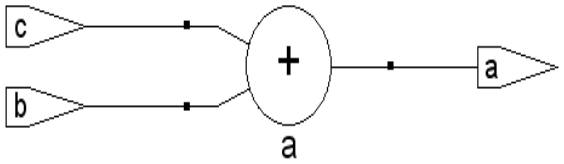
```
module adder1(c,a);
    input c;
    output a;
    assign a=a+c;
endmodule
```

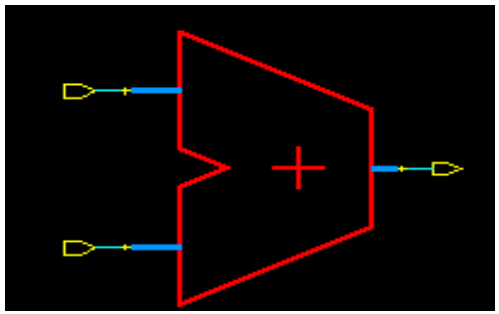




Error!

```
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
```





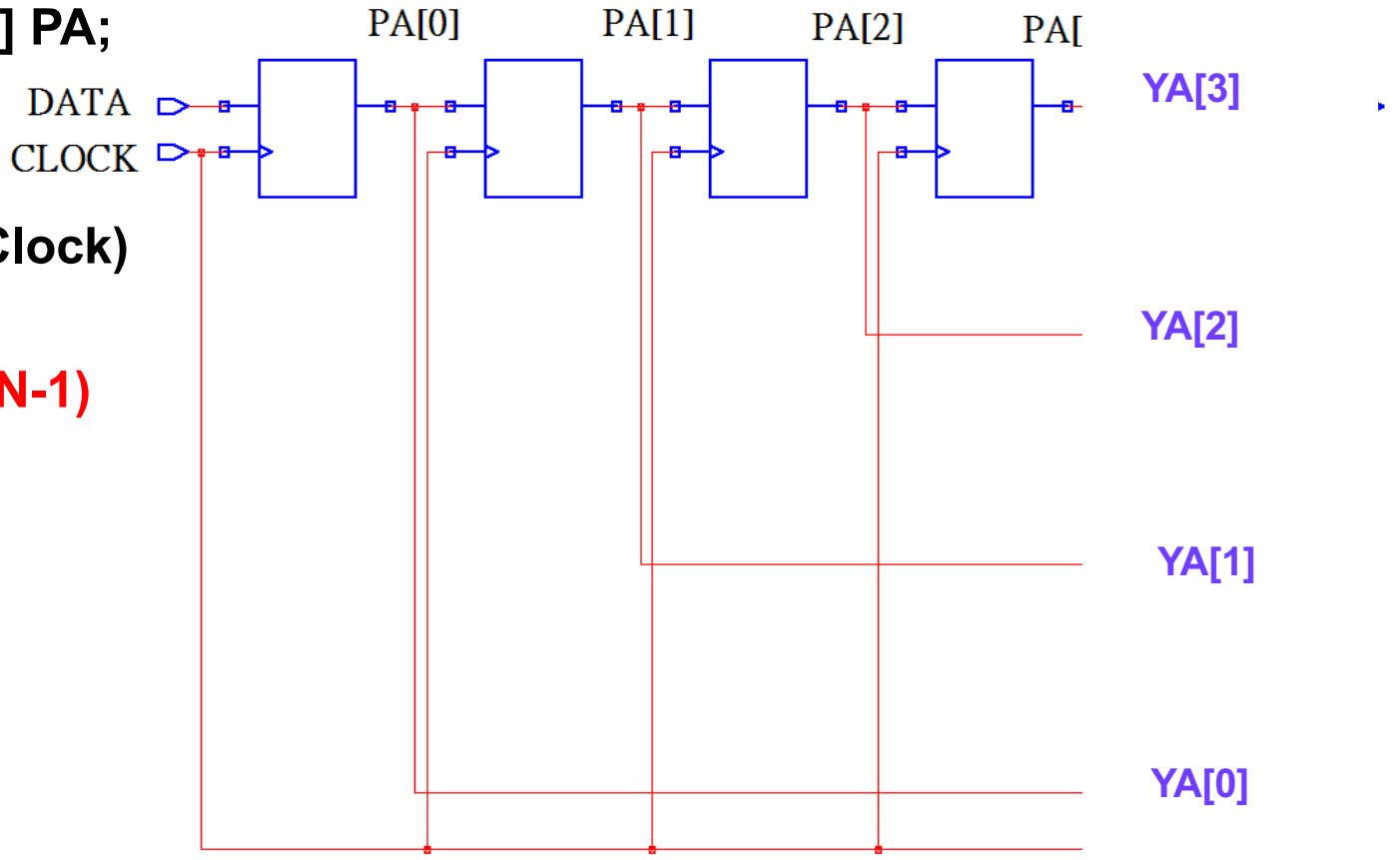
OK!

Example of blocking and for (1/3)

Generate 4 registers

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

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

```
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];
```

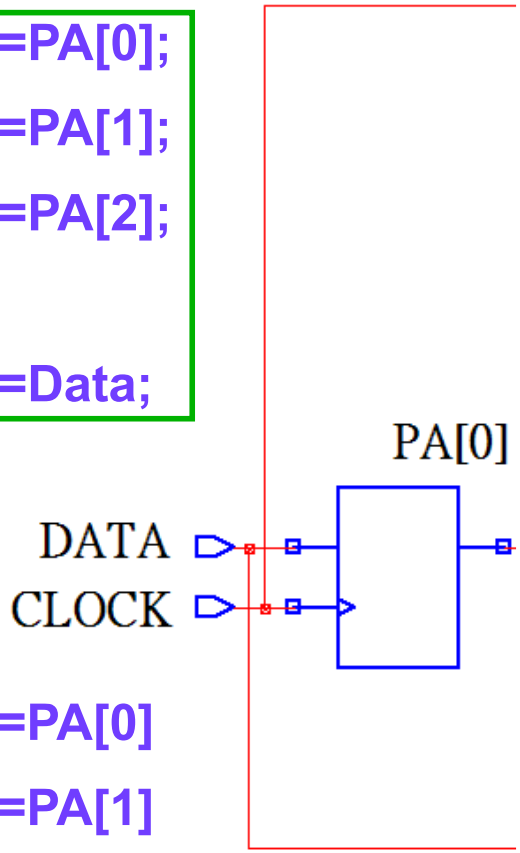


Example of blocking and for (2/3)

Generate 1 registers

```
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;
```



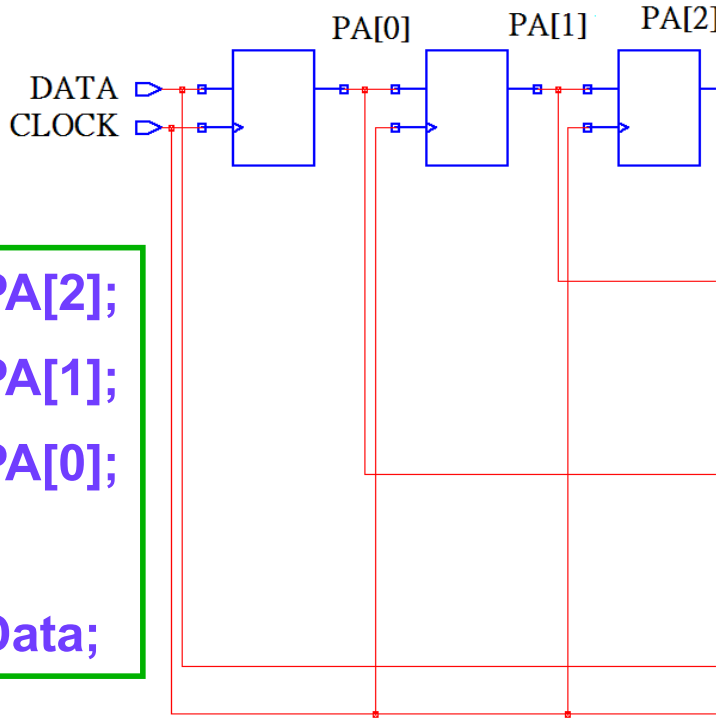

```
YA[0]=PA[0]
YA[1]=PA[1]
YA[2]=PA[2]
YA[3]=PA[3]
```

Example of blocking and for (3/3)

Generate 4 registers

```
module test2(Clock, Data, YA, YB);
input Clock, Data;
output [3:0] YA;
reg [3:0] YA, PA;
integer N;

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

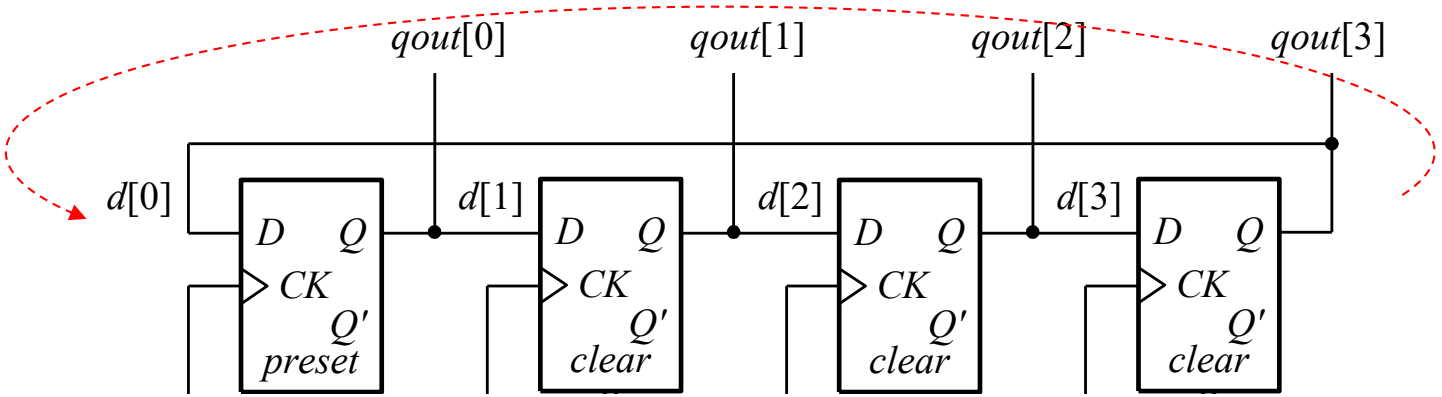
```
PA[3]=PA[2];
PA[2]=PA[1];
PA[1]=PA[0];

PA[0]=Data;
```




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

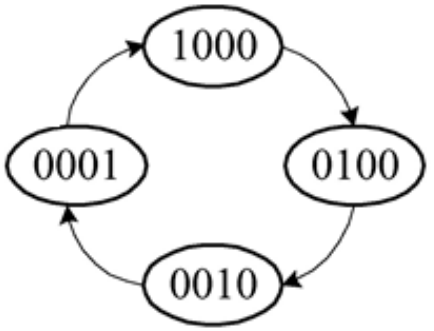
Ring Counter

- Ring counter is an n-stage shift register with the serial output feedback to the serial input



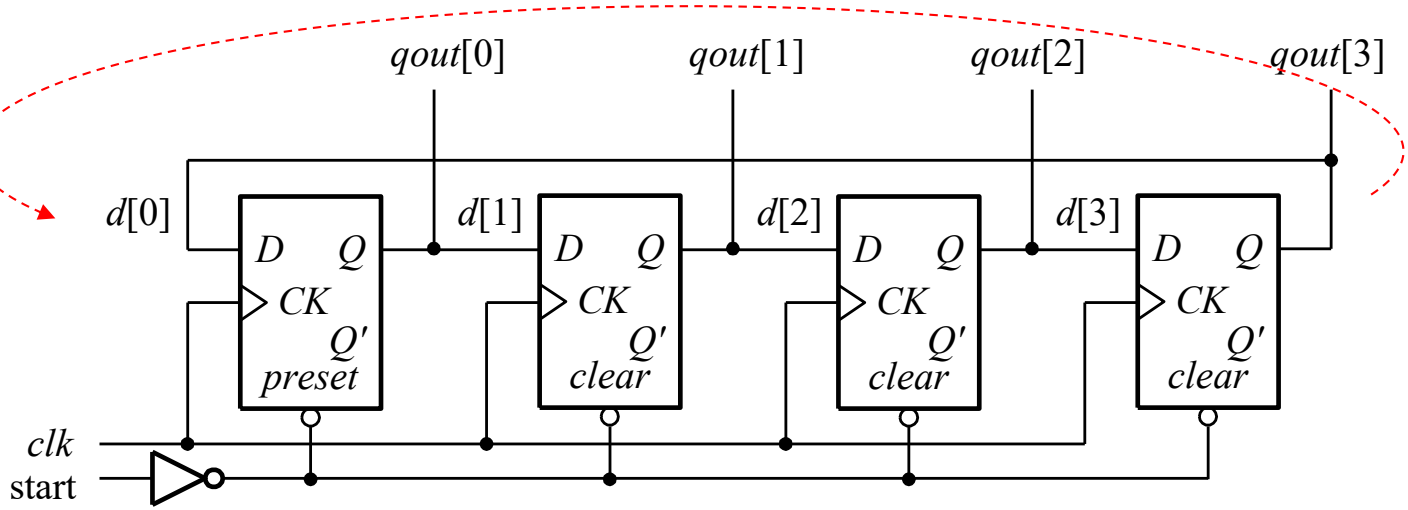
# of Valid States for n bits: n
# of Invalid States of n bits: 2^n-n

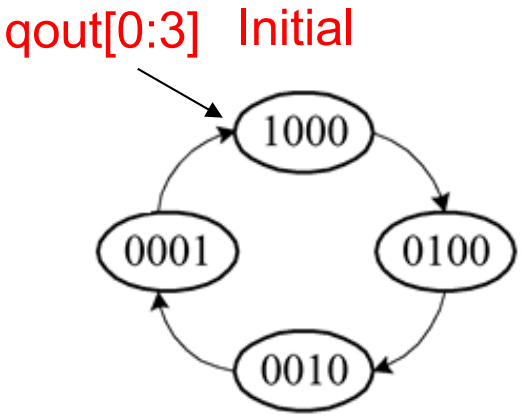
For example: 4 bits
4 valid states: 1000->0100 -> 0010 -> 0001
12 invalid states



Ring Counter - Example

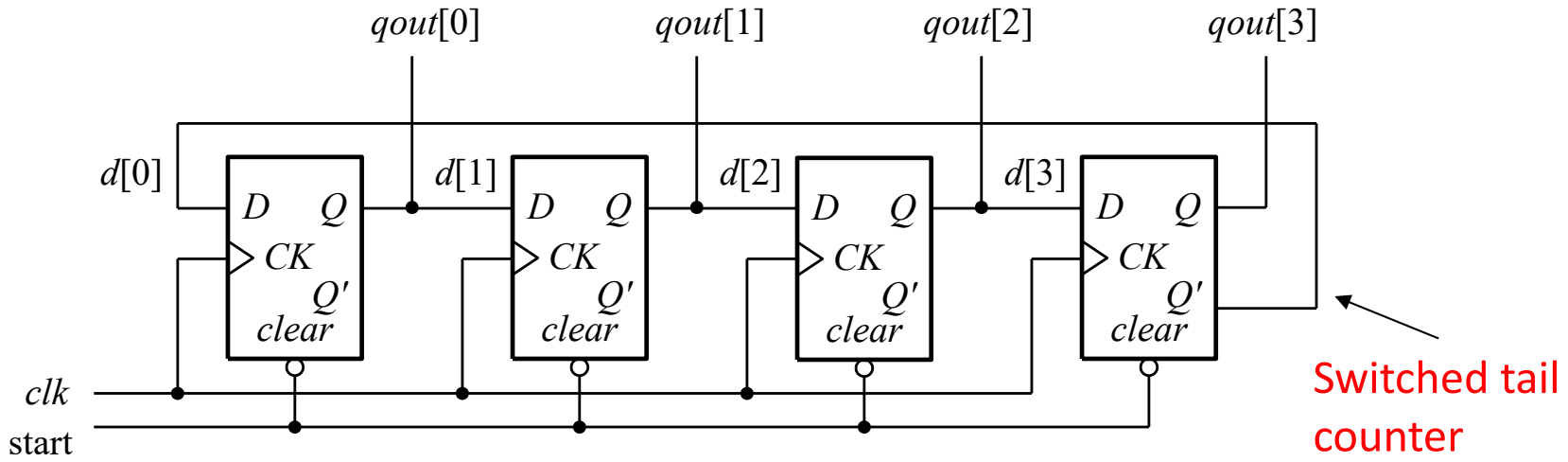
- Design a ring counter that has the following state transitions






```
// a ring counter with initial value
module ring_counter(clk, start, qout);
parameter N = 4;
input  clk, start;
output reg [0:N-1] qout;
// the body of ring counter
always @(posedge clk or posedge start)
  if (start)  qout <= {1'b1,{N-1{1'b0}}};
  else        qout <= {qout[N-1], qout[0:N-2]};
endmodule
```

Johnson Counters (Twisted ring counter)




```
// Johnson counter with initial value
module ring_counter(clk, start, qout);
parameter N = 4; // define the default size
input  clk, start;
output reg [0:N-1] qout;
// the body of Johnson counter
always @(posedge clk or posedge start)
    if (start)  qout <= {N{1'b0}};
    else        qout <= {~qout[N-1], qout[0:N-2]};
endmodule
```

qout[0:3]

0000->1000 -> 1100 -> 1110->1111 -> 0111->0011->0001->0000

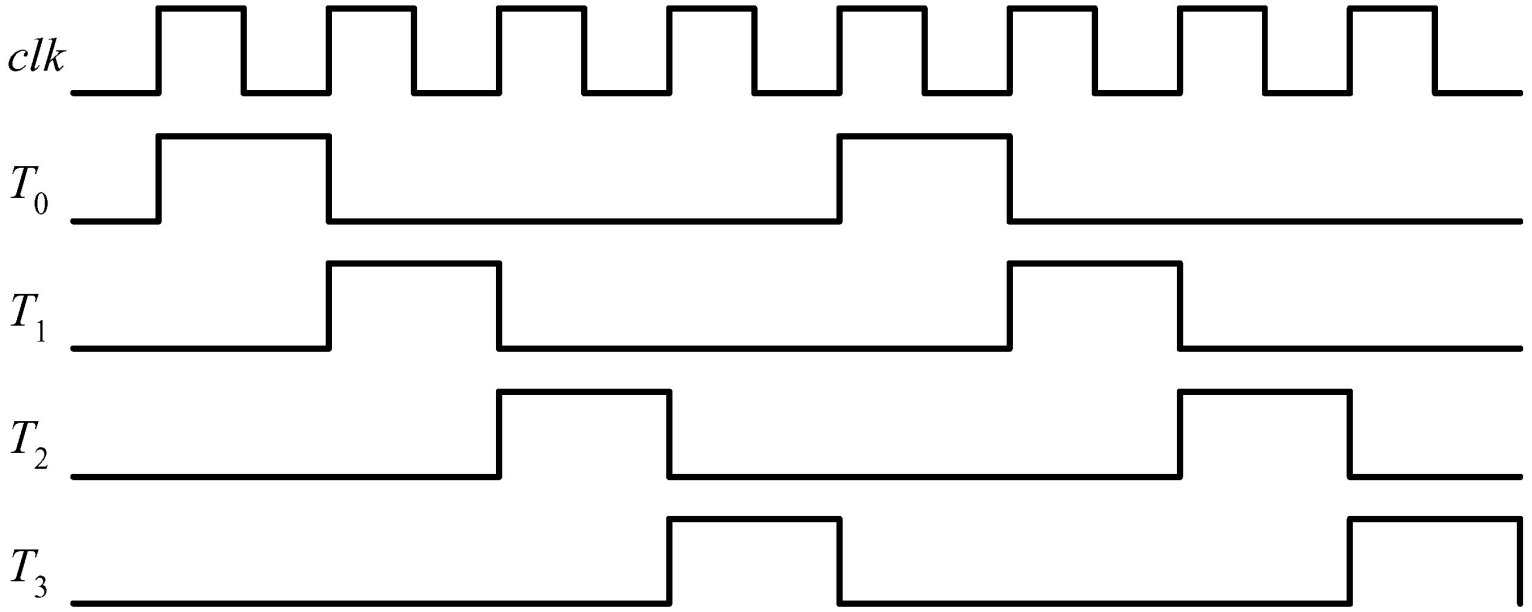
Timing Generators

- A timing generator is a device that generates timings required for specific application.
- Two timing generators: multiphase clock signal generator and digital monostable circuit.
- Multiphase clock signals generator includes:
  - Ring counter
  - Binary counter with decoder
- Digital monostable circuits includes:
  - Retriggerable
  - Nonretriggerable

A Multiphase Clock Signal

- Multiphase clock signal (4 phases) is shown below.
- Each phase lasts for a clock cycle and is repeated after a given number of clock cycles.

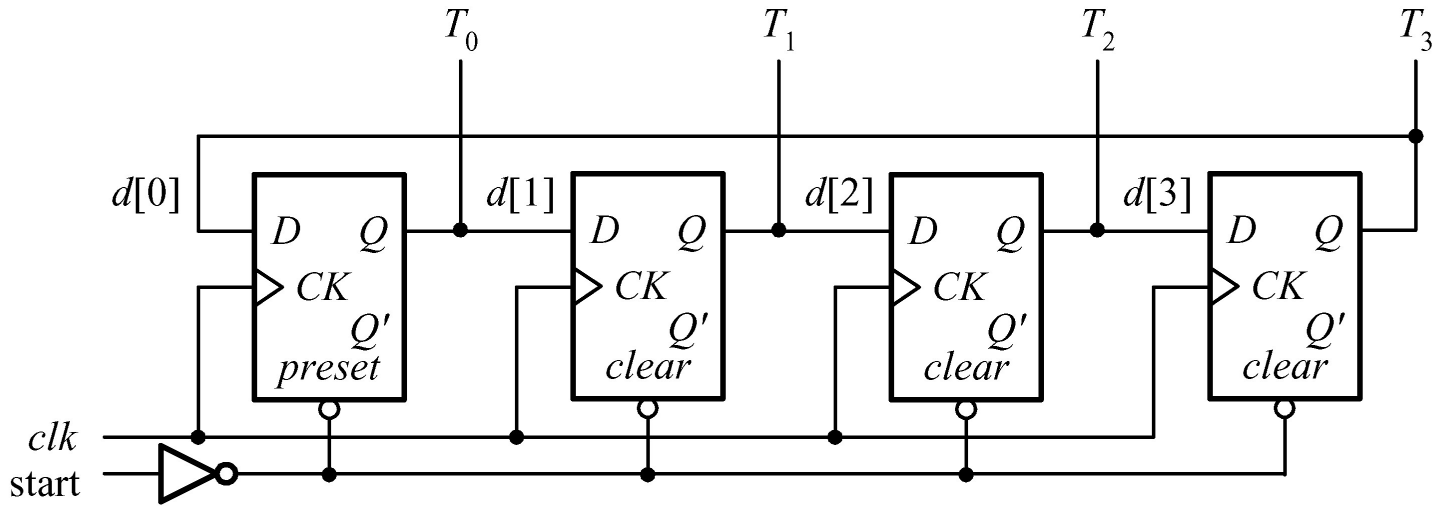
  => can be implemented by a ring counter



1000 0100 0010 0001

Multiphase Clock Generators

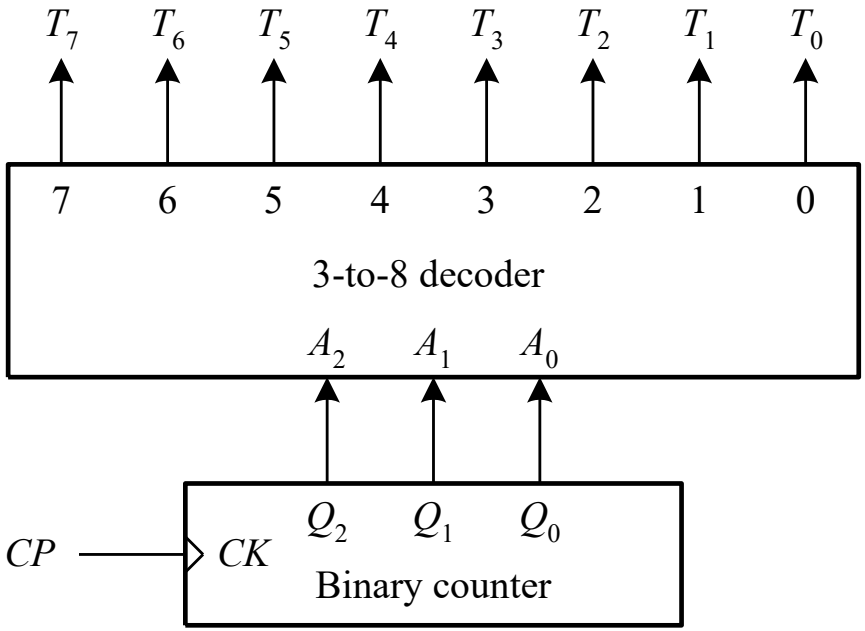
(a) Ring counter approach




```
// a ring counter with initial value serve as a timing generator
module ring_counter_timing_generator(clk, reset, qout);
parameter n = 4;  // define the counter size
input  clk, reset;
output reg [0:n-1] qout;
// the body of ring counter
always @(posedge clk or posedge reset)
  if (reset)  qout <= {1'b1,{n-1{1'b0}}};
  else        qout <= {qout[n-1], qout[0:n-2]};
endmodule
```

Multiphase Clock Generators – Using binary counter with decoder

- Generate an n-phase clock signal by using a $\log_2$n-bit binary counter with a $\log_2$n-to-n decoder



An 8 phase clock generators

3-to-8 decoder

3 bit binary counter

000 -> 00000001
001 -> 00000010
…

111 -> 10000000

Multiphase Clock Generators

a binary counter with decoder serve as a timing generator

```
module binary_counter_timing_generator(clk, reset, enable, qout);
parameter N = 8;   // define the number of phases
parameter M = 3;  // define the bit number of binary counter
input  clk, reset, enable;
output reg [N-1:0] qout;
reg    [M-1:0] bcnt_out;

// the body of binary counter
always @(posedge clk or posedge reset)
  if (reset)  bcnt_out <= {M{1'b0}};
  else if (enable) bcnt_out <= bcnt_out + 1;
// decode the output of the binary counter

always @(bcnt_out)
   qout = {N-1{1'b0},1'b1} << bcnt_out;
endmodule
```

m-bit binary counter

m-to-2^m decoder