BEE 271 Digital circuits and systems

Spring 2017

Lecture 11: Sequential circuits

Nicole Hamilton

https://faculty.washington.edu/kd1uj

Topics

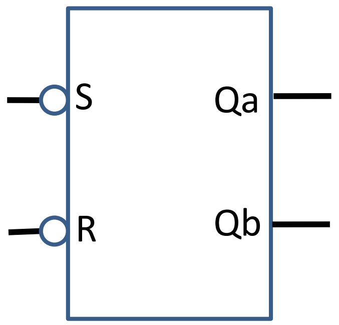
1. Introduction to sequential circuits

Terminology

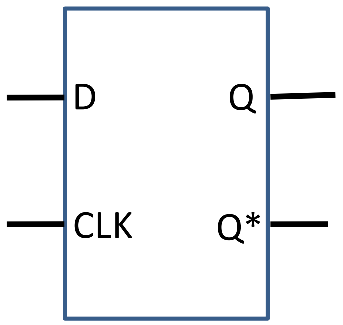
Basic latch: A feedback connection of two NORs or two NANDs that can store 1 bit. Set using the S input and reset using the R input.

Gated latch: A basic latch that includes input gating and a control input signal. Retains its value when the control signal is 0, changes state when the control signal is 1.

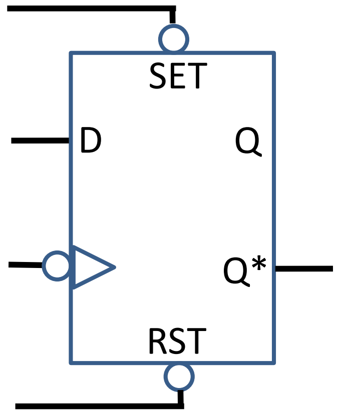
Flip-flop: A storage element whose output changes only on the edge of a controlling clock. Can be either positive or negative edge-triggered.



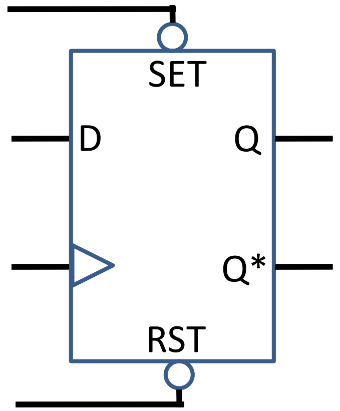
Latch



Gated Latch

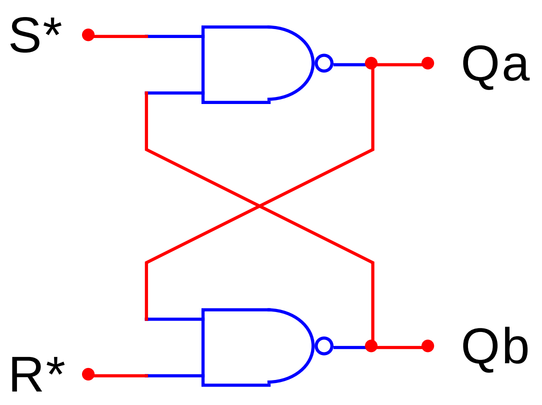


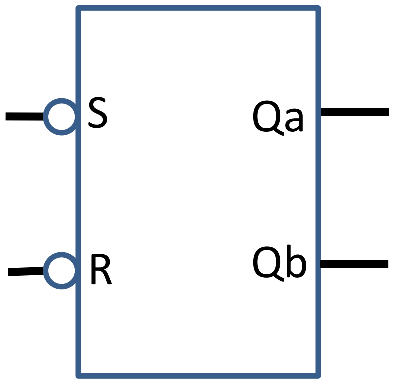
Master-slave



Edge-triggered

Set/reset latches





S'R' latch

S*	**R***	**Qa**	**Qb**
0	0	1	1
0	1	1	0
1	1	no change	no change
1	0	0	1

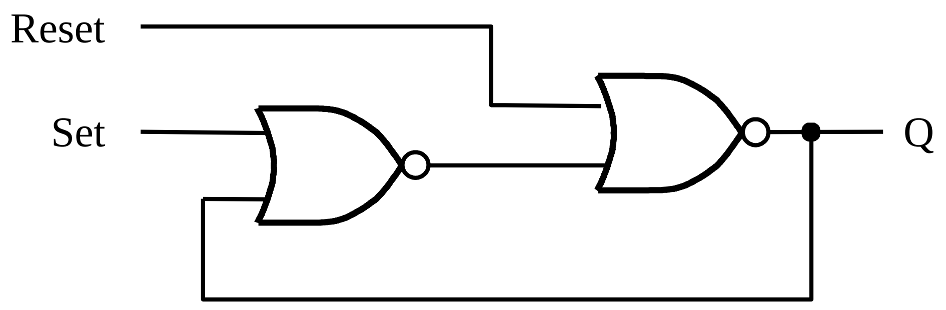
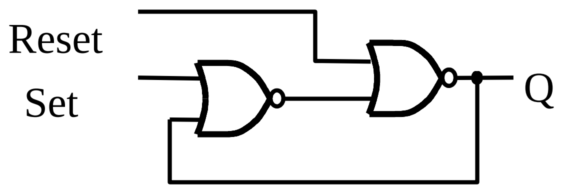
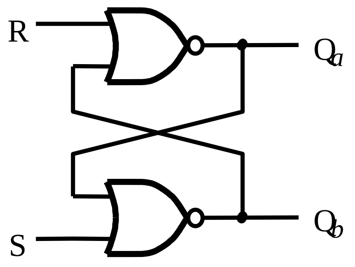


Figure 5.3. A memory element with NOR gates.

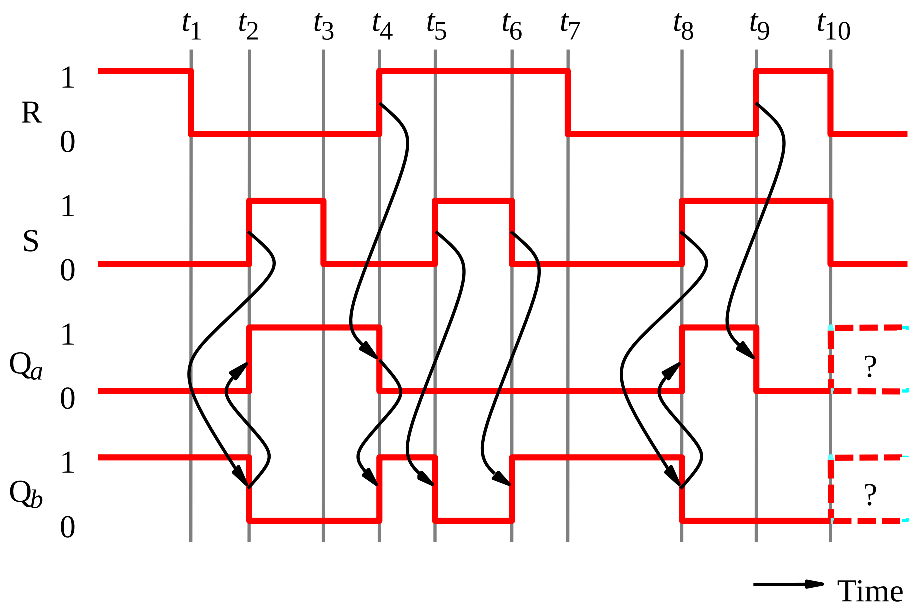




(a) Circuit

S	R	Q_a	Q_b	
0	0	0/1	1/0	(no change)
0	1	0	1	
1	0	1	0	
1	1	0	0	

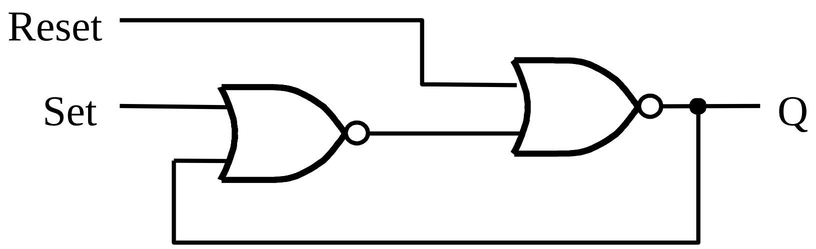
(b) Truth table



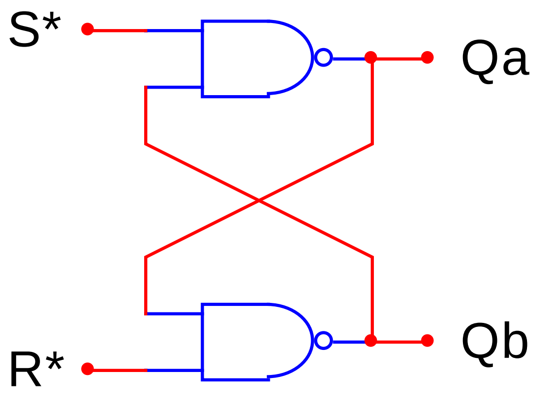
(c) Timing diagram

Figure 5.4. A basic latch built with NOR gates.

```
module SetResetLatch( input set, reset,
        output Q, Qn );

    nor ( Qn, set,   Q  );
    nor ( Q,  reset, Qn );

endmodule
```



```
module SetnResetnLatch( input setn, resetn,
        output Q, Qn );

    nand ( Q,  setn,   Qn );
    nand ( Qn, resetn, Q  );

endmodule
```



```
module SRLatch( input s, r, output reg Q );

   always @( * )
      casex ( { s, r } )
         'b1x: Q = 1;
         'b01: Q = 0;
      endcase

endmodule
```

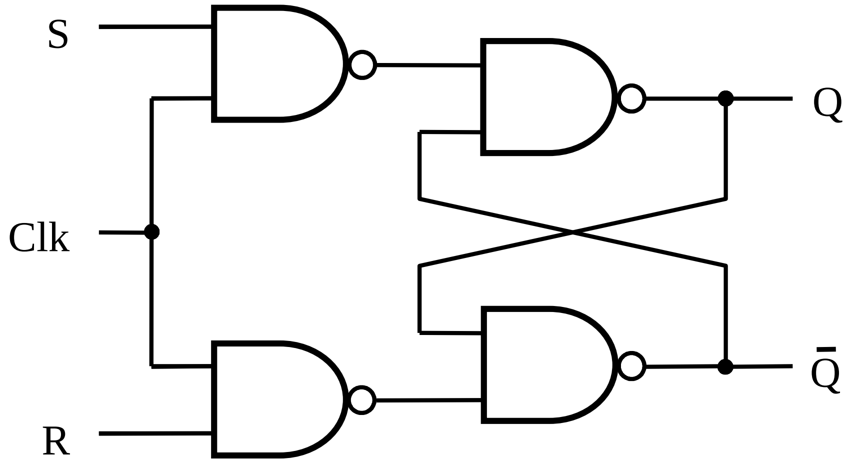
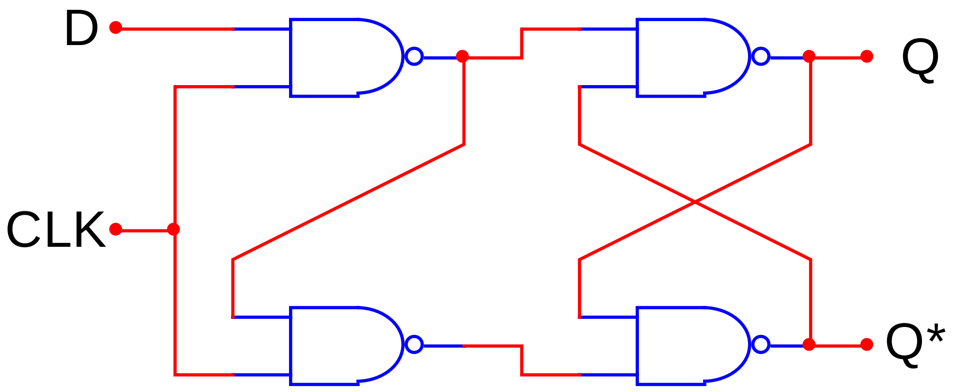
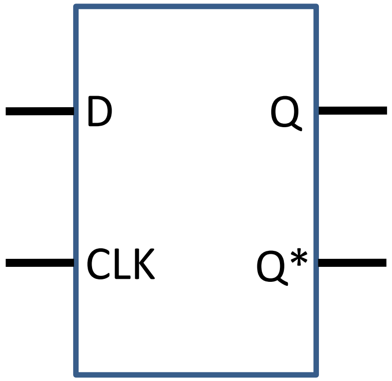


Figure 5.6. Gated SR latch with NAND gates.

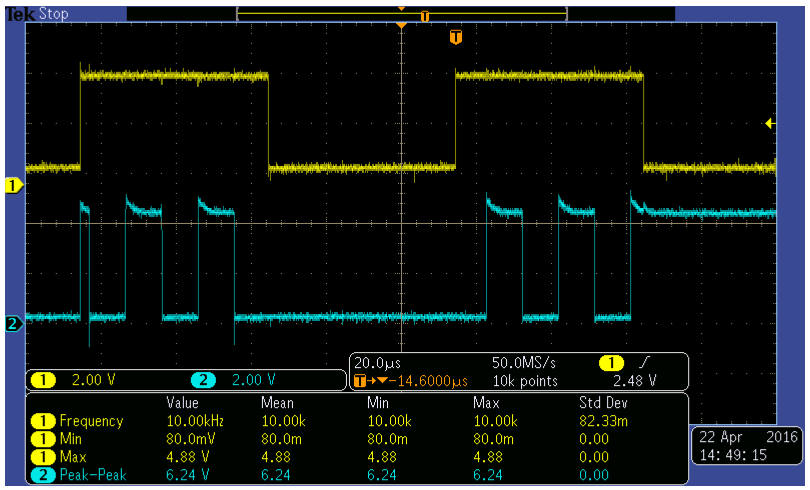
Gated latches





Gated D latch

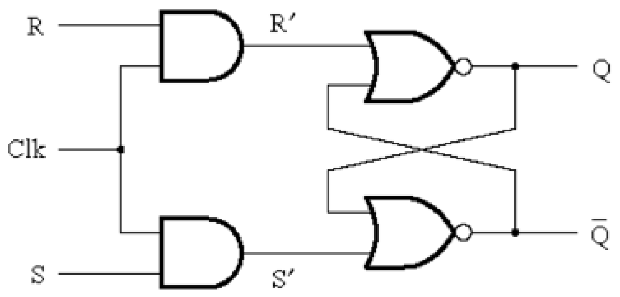
Clock used to sample the input when the clock is high and hold it when the clock is low.



```
1  2.00 V          2  2.00 V       20.0µs              50.0MS/s     1  ⟋
                                   T→▼−14.6000µs       10k points      2.48 V
                 Value      Mean      Min       Max       Std Dev
1 Frequency      10.00kHz   10.00k    10.00k    10.00k    82.33m
1 Min            80.0mV     80.0m     80.0m     80.0m     0.00
1 Max            4.88 V     4.88      4.88      4.88      0.00
2 Peak-Peak      6.24 V     6.24      6.24      6.24      0.00
                                                                     22 Apr   2016
                                                                     14: 49: 15
```

clock = 10 KHz / D = 52 KHz

```
module DLatch1( input clock, D, output reg Q );
   always @( * )
      if ( clock )
         Q = D;      // Implied memory
endmodule


module DLatch2( input clock, D, output reg Q );
   always @( * )
      Q = clock ? D : Q;
endmodule
```

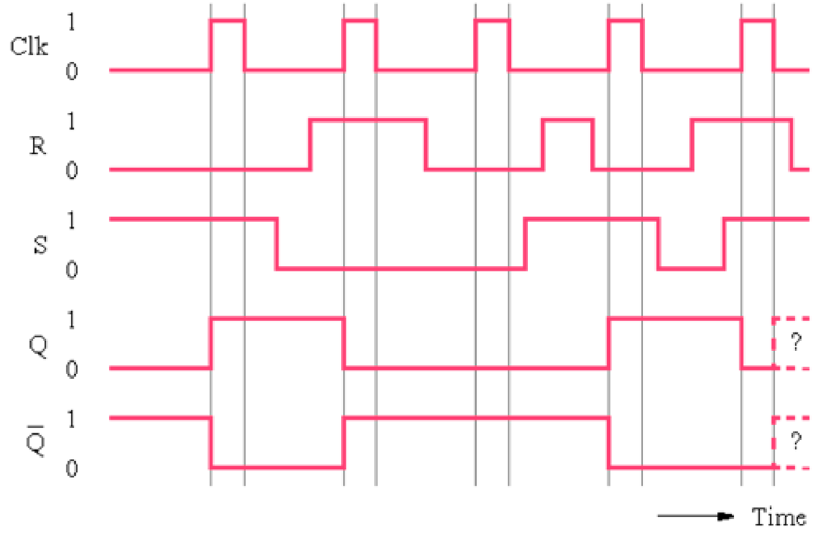


(a) Circuit

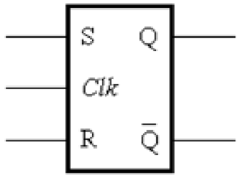
Clk	S	R	$Q(t+1)$
0	x	x	$Q(t)$ (no change)
1	0	0	$Q(t)$ (no change)
1	0	1	0
1	1	0	1
1	1	1	x

(b) Characteristic table

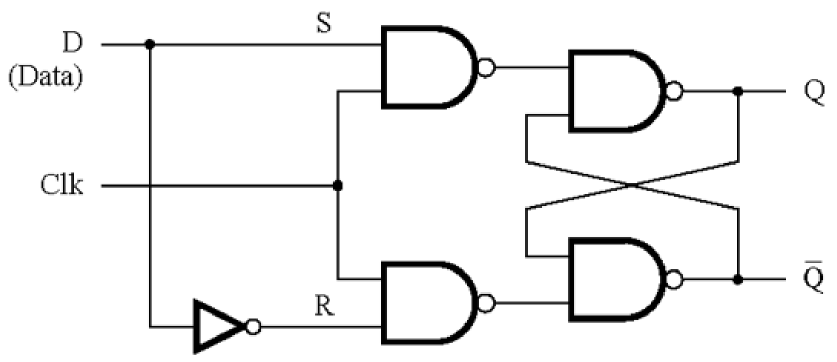
Figure 5.5. Gated SR latch.



(c) Timing diagram



(d) Graphical symbol

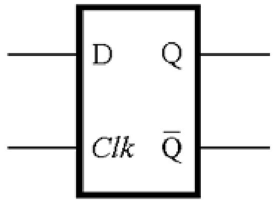


(a) Circuit

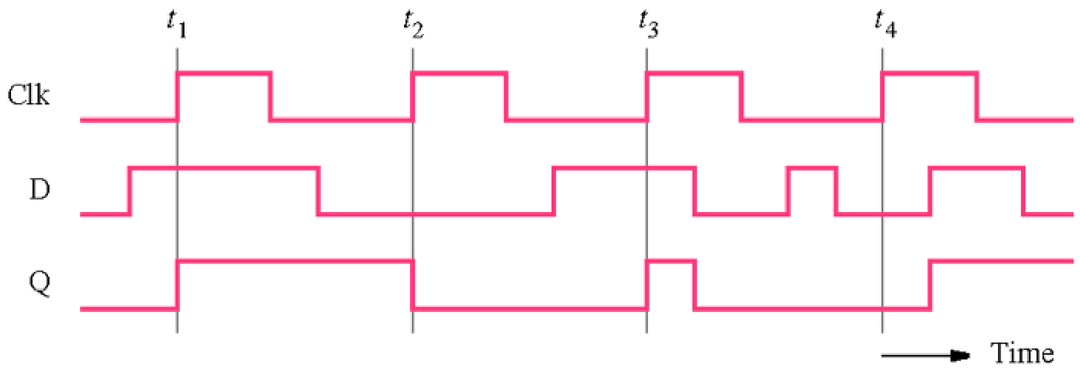
Figure 5.7. Gated D latch.

Clk	D	$Q(t+1)$
0	x	$Q(t)$
1	0	0
1	1	1

(b) Characteristic table



(c) Graphical symbol

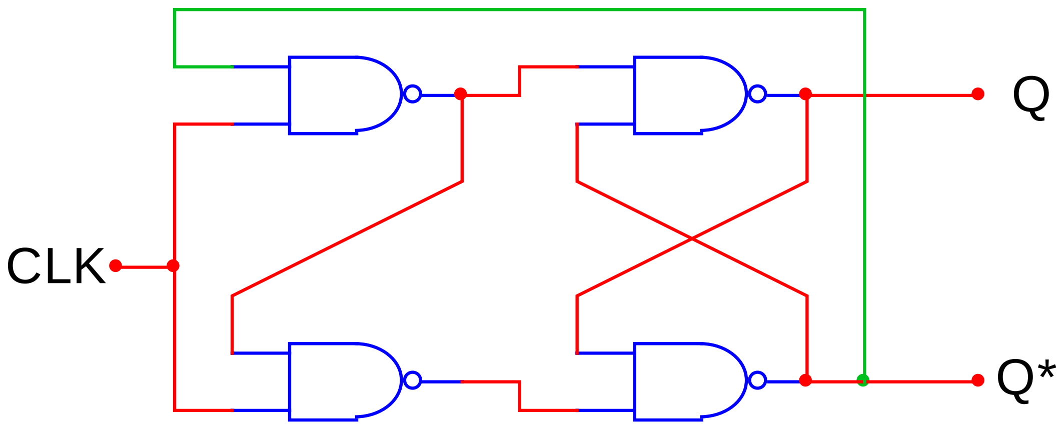


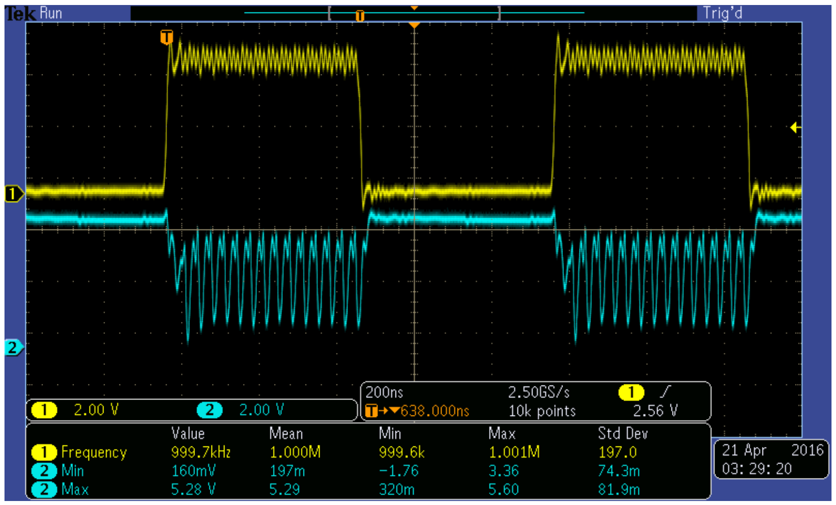
(d) Timing diagram

Flip Flops

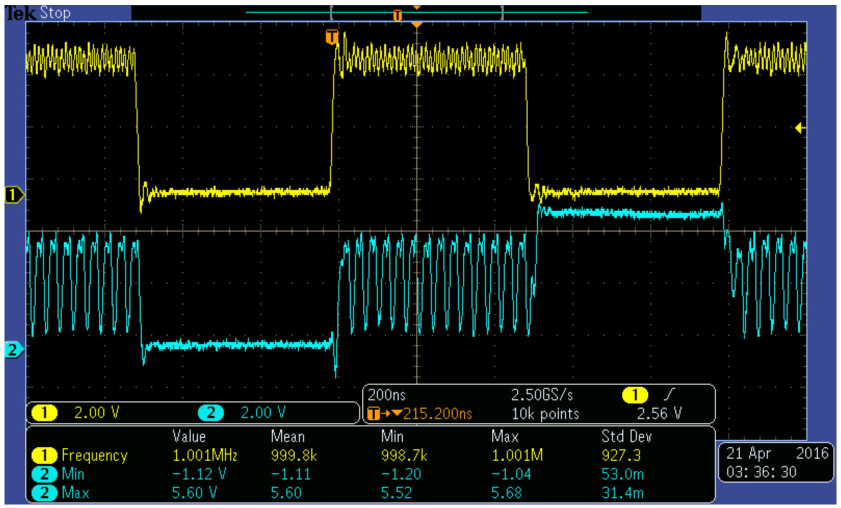
- Problem: latches are sensitive to any changes that occur with the input while the clock or control signal is high
  - Glitches/Hazards
  - Unsynchronized changes
- Solution: use flip-flops, devices that react only on the clock edge

But we cannot build reliable counters or anything else requiring feedback with latches.





```
 1  2.00 V          2  2.00 V     200ns          2.50GS/s     1  ∫
                                  T→▼638.000ns   10k points     2.56 V
              Value      Mean      Min       Max       Std Dev
 1 Frequency  999.7kHz   1.000M    999.6k    1.001M    197.0        21 Apr   2016
 2 Min        160mV      197m      -1.76     3.36      74.3m        03: 29: 20
 2 Max        5.28 V     5.29      320m      5.60      81.9m
```



```
 1  2.00 V          2  2.00 V     200ns          2.50GS/s     1  ∫
                                  T→▼215.200ns   10k points     2.56 V
              Value      Mean      Min       Max       Std Dev
 1 Frequency  1.001MHz   999.8k    998.7k    1.001M    927.3        21 Apr   2016
 2 Min        -1.12 V    -1.11     -1.20     -1.04     53.0m        03: 36: 30
 2 Max        5.60 V     5.60      5.52      5.68      31.4m
```

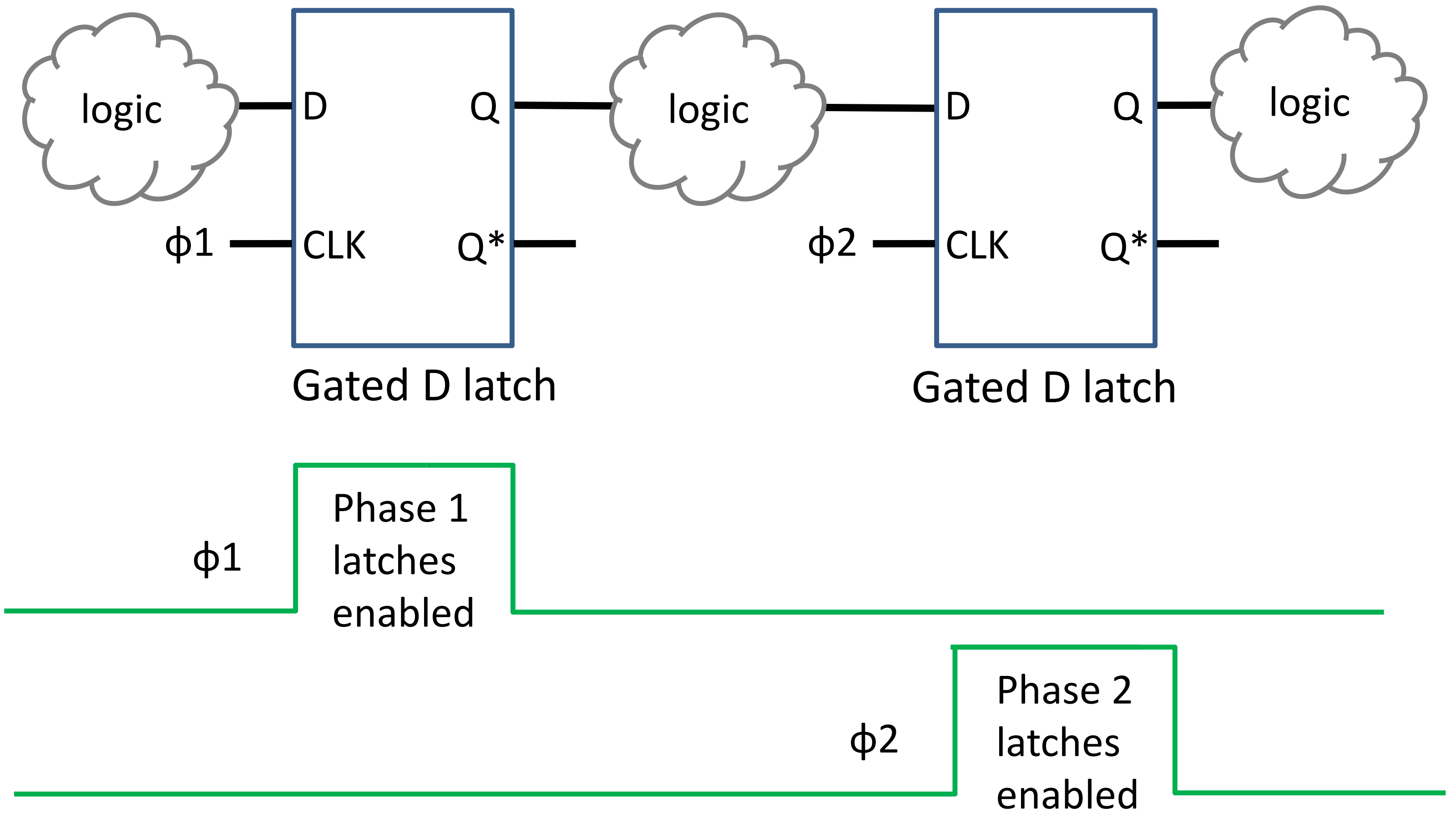
What you get depends on small differences in the length of the green wire.

Solution is to isolate inputs from outputs

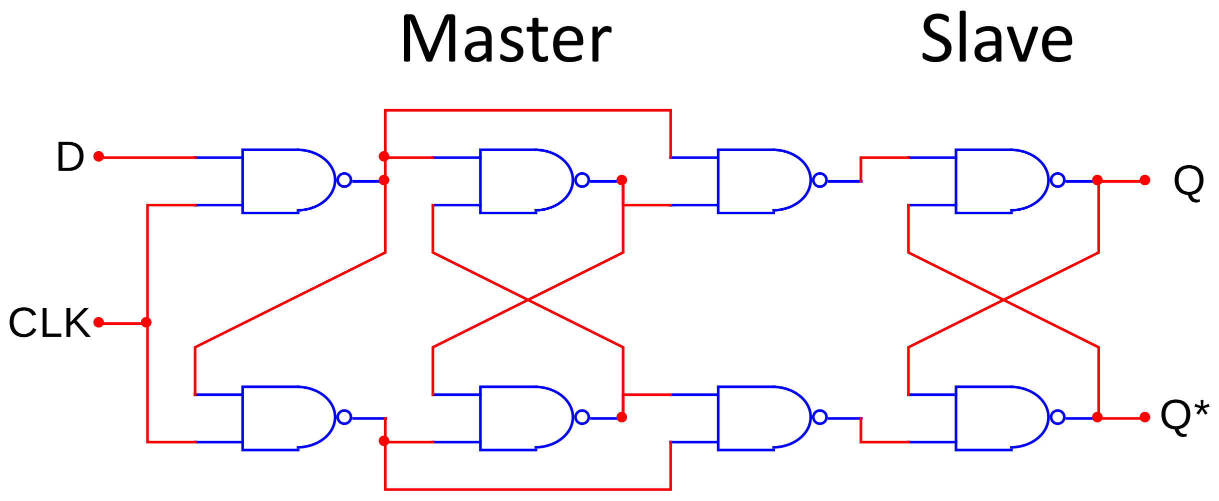
Three popular alternatives:

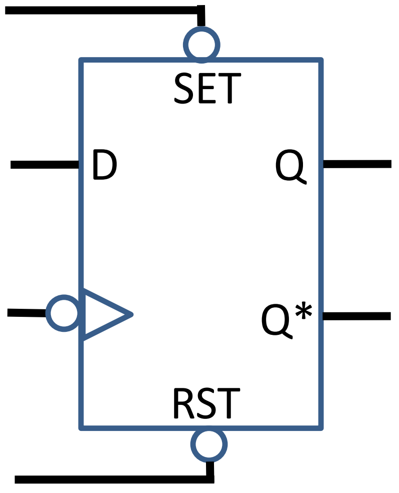
1. Two-phase clocking
2. Master-slave
3. Edge-triggered

Two-phase clock

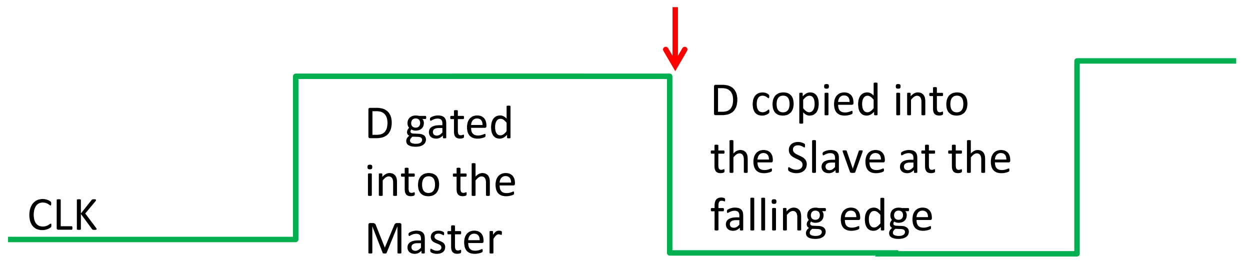


Non-overlapping clocks for phases 1 and 2. Advantages are fewer gates per bit and that you can put logic between both phases so long phase 1 latches only use phase 2 inputs and vice versa.



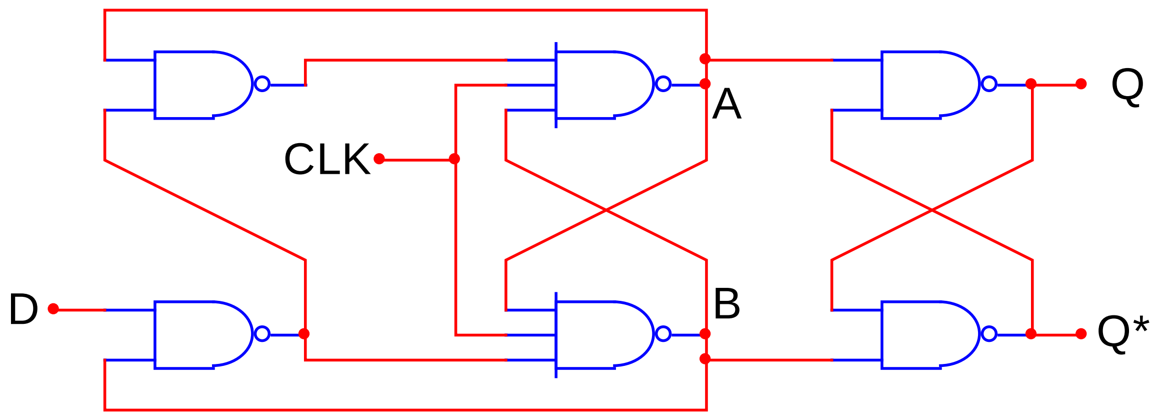


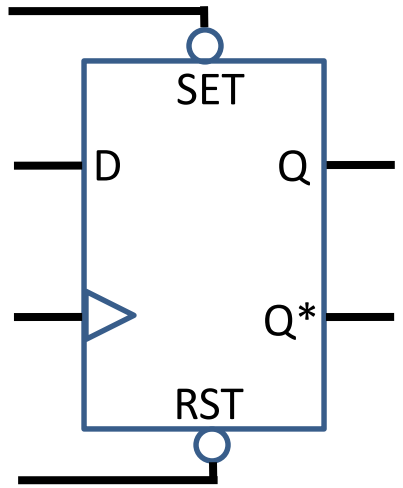
Master-slave D flip-flop



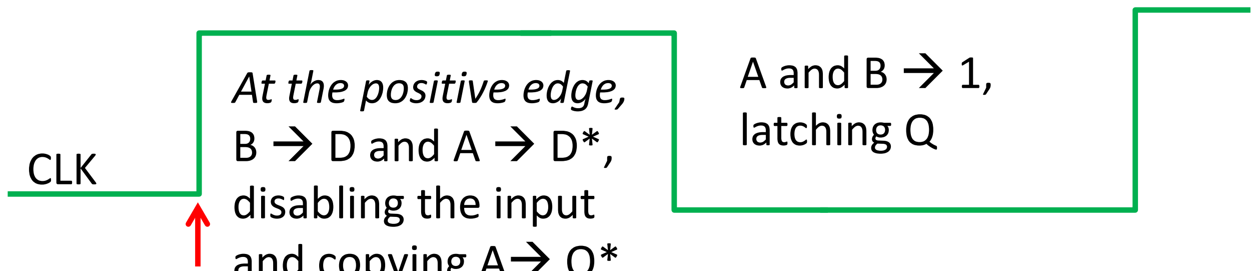
Because of the signal path is longer to the slave, we can guarantee the input to the master will be disabled before the input to the slave is enabled.

Edge-triggered





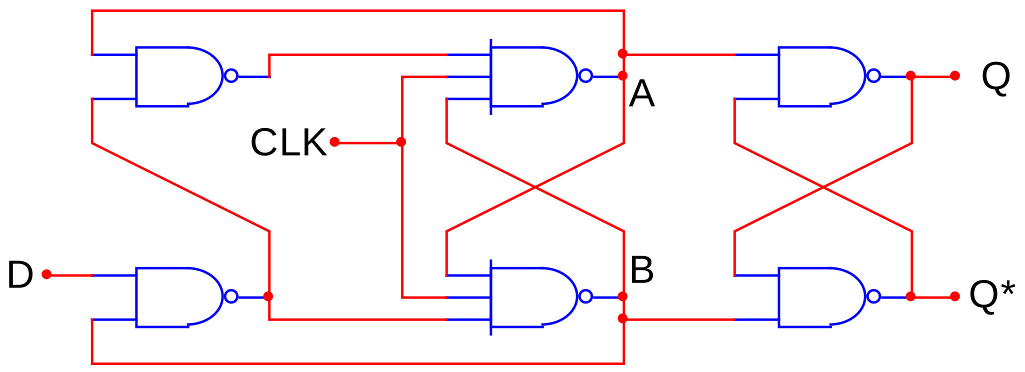
Edge-triggered D flip-flop

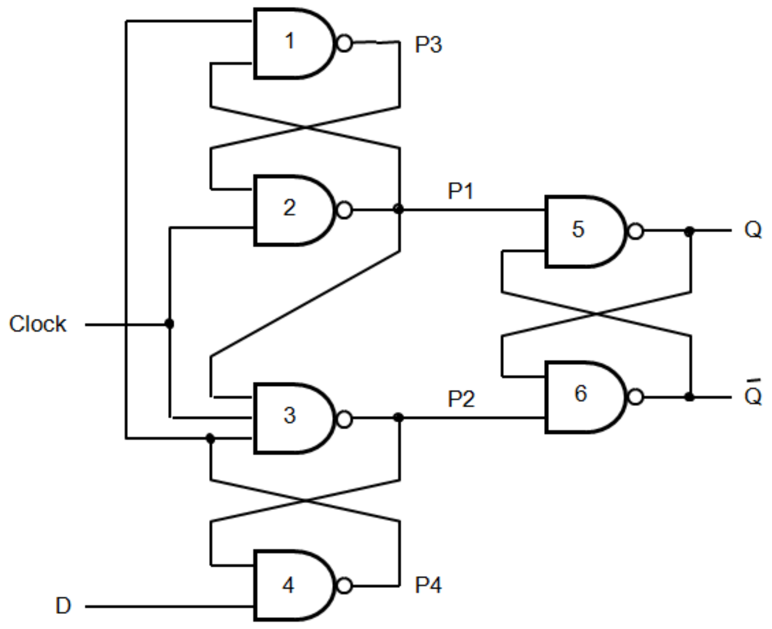


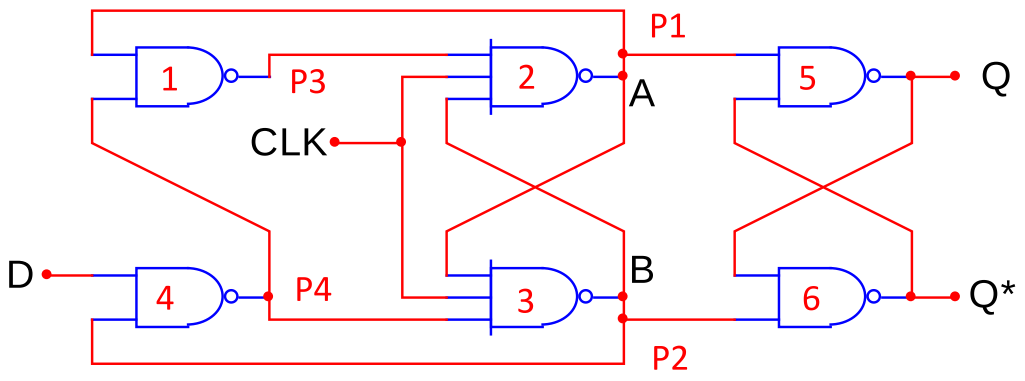
At the positive edge, B → D and A → D*, disabling the input and copying A→ Q* and B → Q

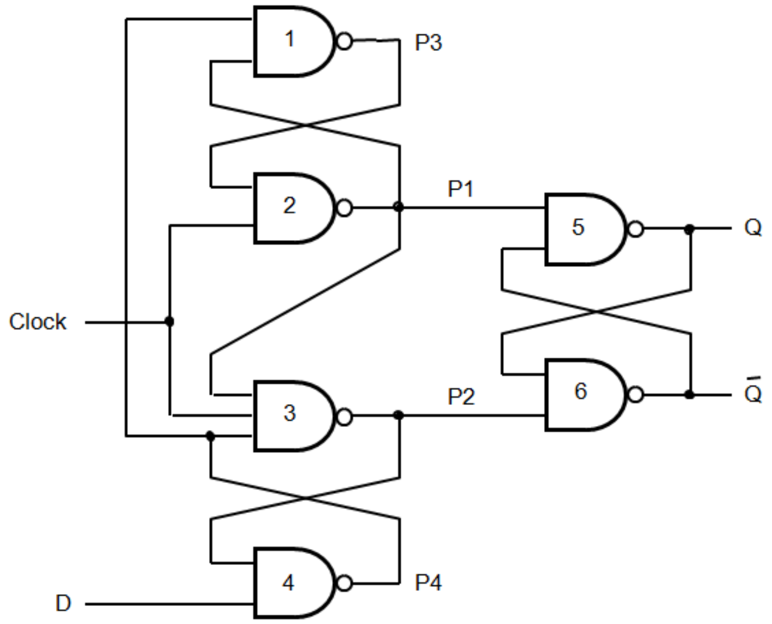
A and B → 1, latching Q

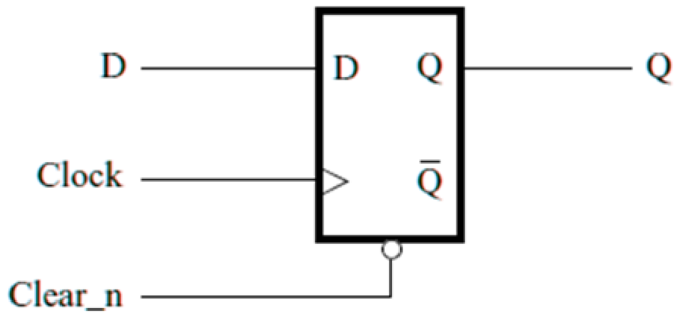
Requires a hold time: D is not allowed to change until the rising edge of the clock has propagated through A or B back to the input NANDs, latching whichever side was a zero.


CLK
D
A
B
Q
Q*

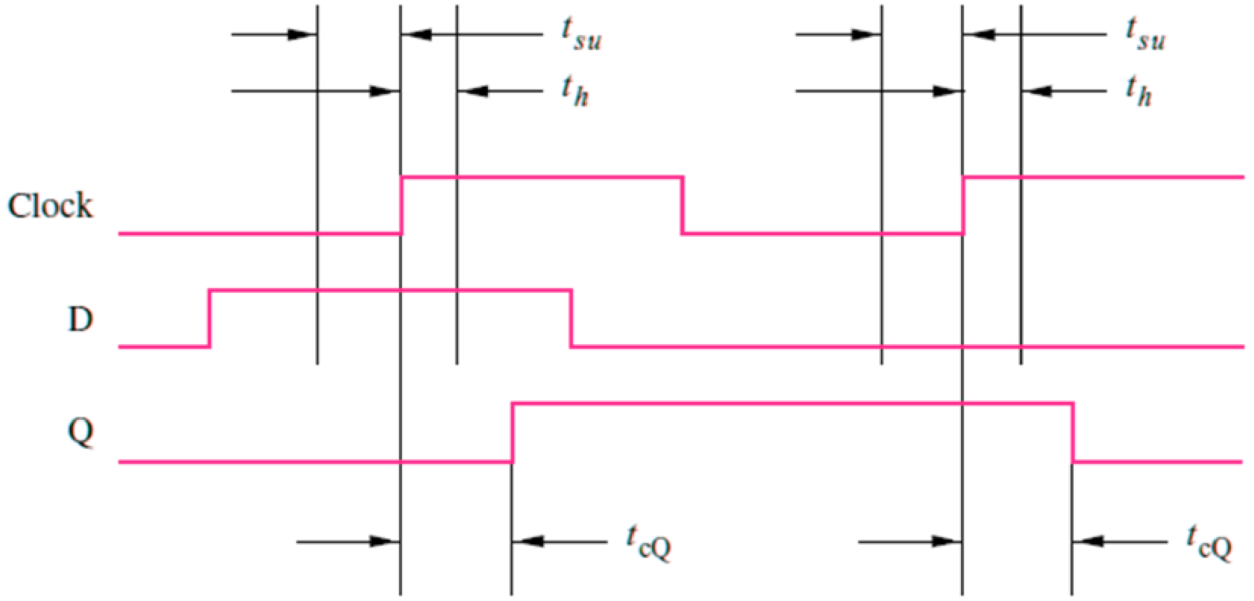

1
2
3
4
5
6
P3
P1
P2
P4
Clock
D
Q
$\bar{Q}$


P1
1
P3
2
A
5
Q
CLK
D
4
P4
3
B
6
Q*
P2


1
P3
2
P1
5
Q
Clock
3
P2
6
$\bar{Q}$
4
P4
D

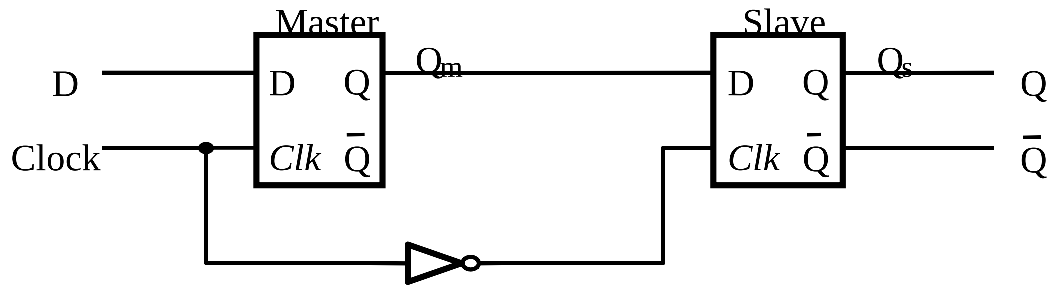


(a) D flip-flop with asynchronous clear

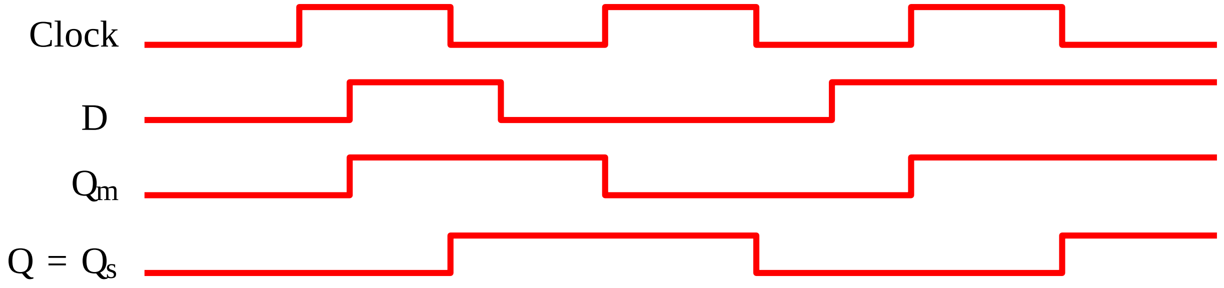


(b) Timing diagram

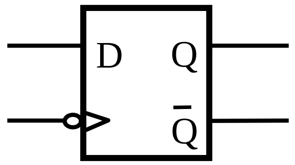
$t_{su} + t_h$ is called the “aperture” or “window” when the input must be good.



(a) Circuit



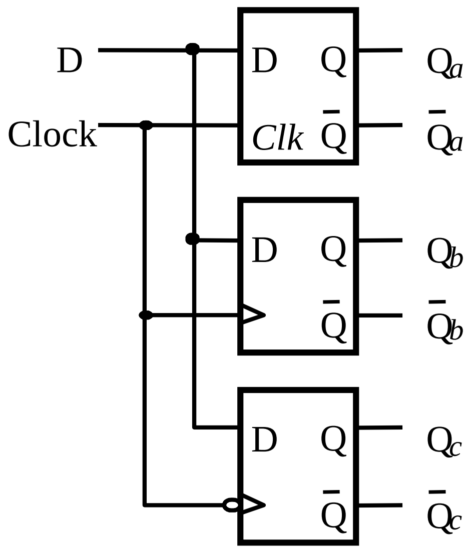
(b) Timing diagram



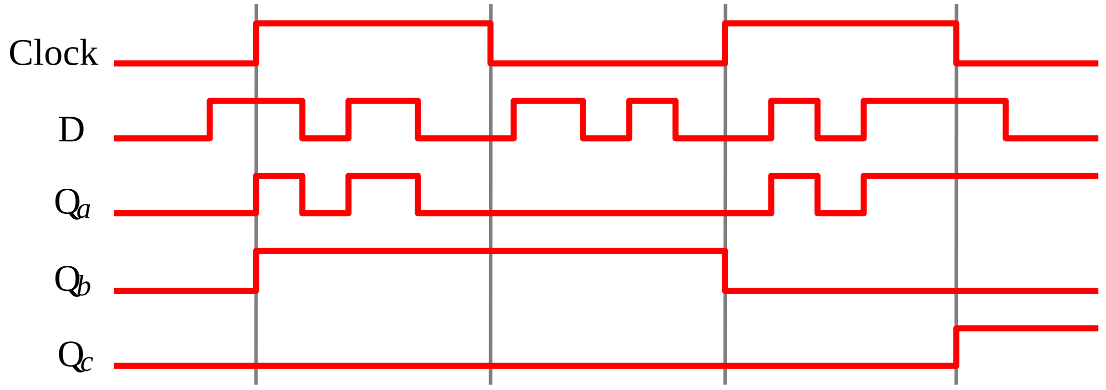
(c) Graphical symbol

Figure 5.9. Master-slave D flip-flop.

Figure 5.10. Comparison of level-sensitive and edge-triggered D storage elements.



(a) Circuit



(b) Timing diagram

```
module DMasterSlave1( input clock, D, output reg Q );

   wire Qm;
   DLatch1 master ( clock,  D,  Qm );
   DLatch1 slave  ( ~clock, Qm, Q  );

endmodule

module DMasterSlave2( input clock, D, output reg Q );

   reg Qm;
   always @( * )
      if ( clock )
         Qm = D;
      else
         Q = Qm;

endmodule
```

```
module DMasterSlave3( input clock, D, output reg Q );

   reg Qm;
   always @( negedge clock )
      Q <= D;

endmodule
```

```
module DMasterSlave3( input clock, D, output reg Q );

   reg Qm;
   always @( negedge clock )
      Q <= D;

endmodule

module DEdgeTriggered( input clock, D, output reg Q );

   always @( posedge clock )
      Q <= D;


endmodule
```

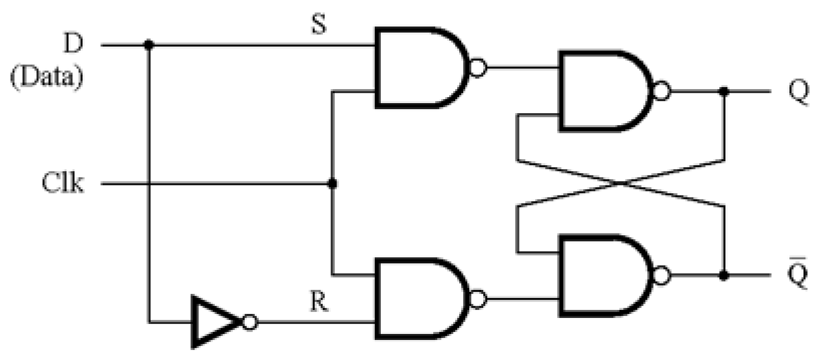
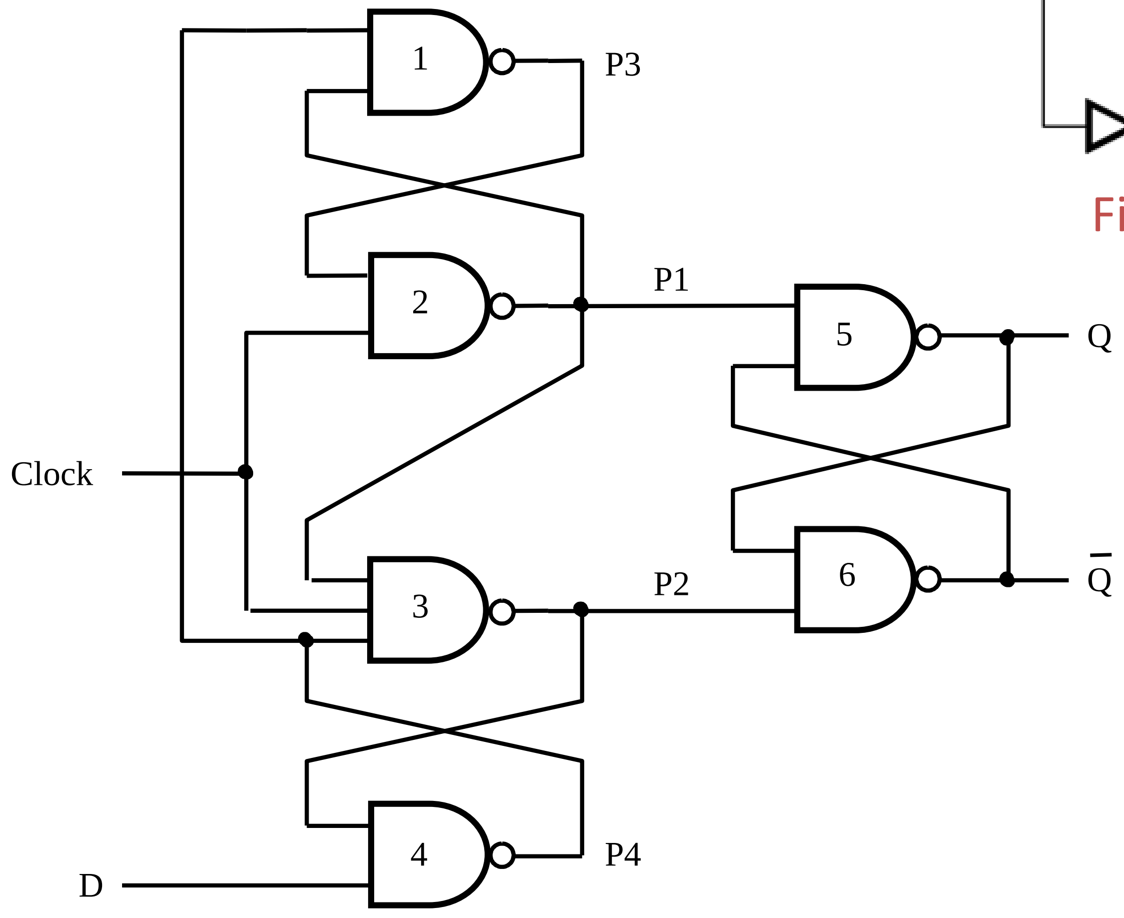
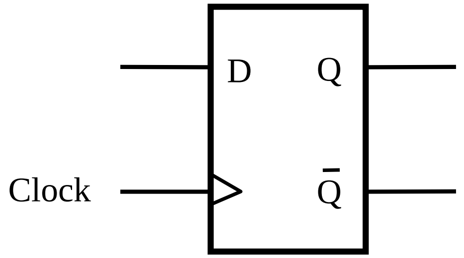


Figure 5.7. Gated D latch.

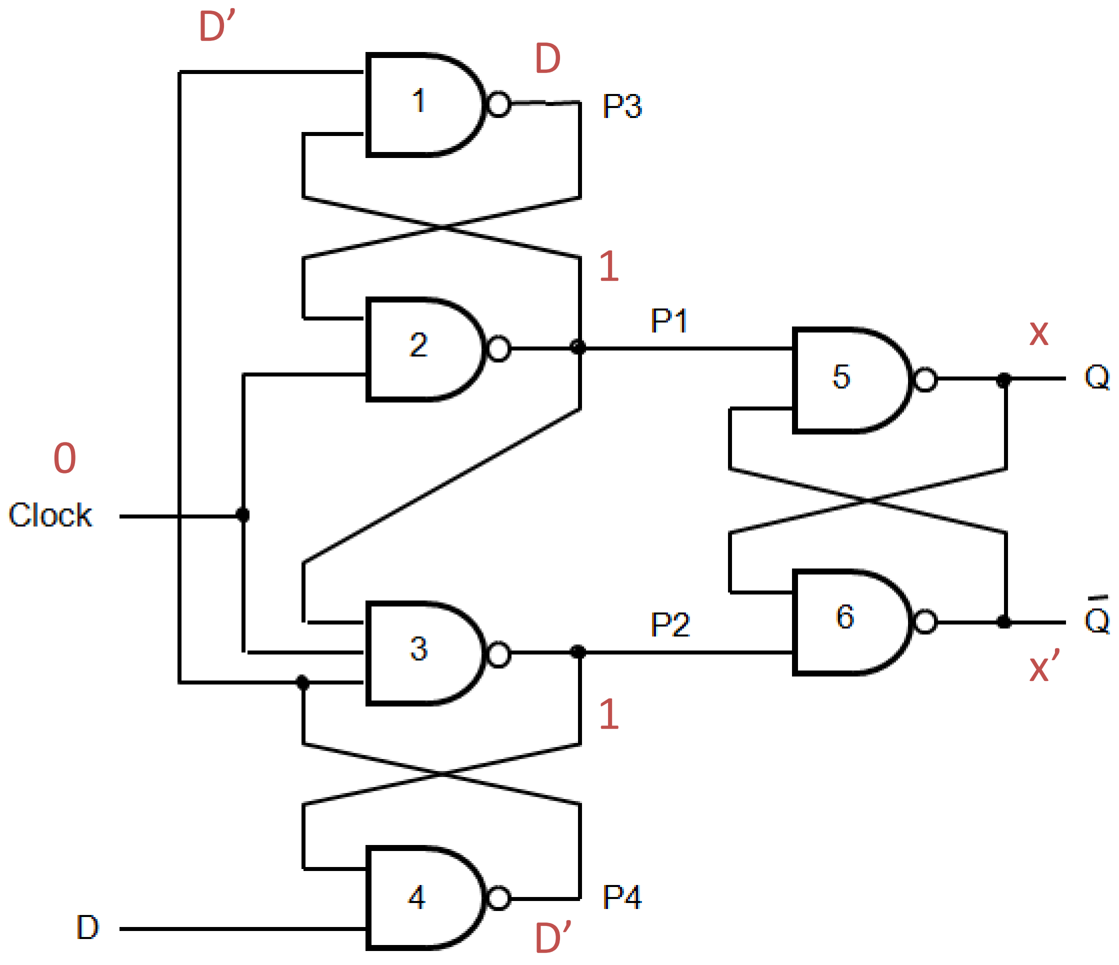




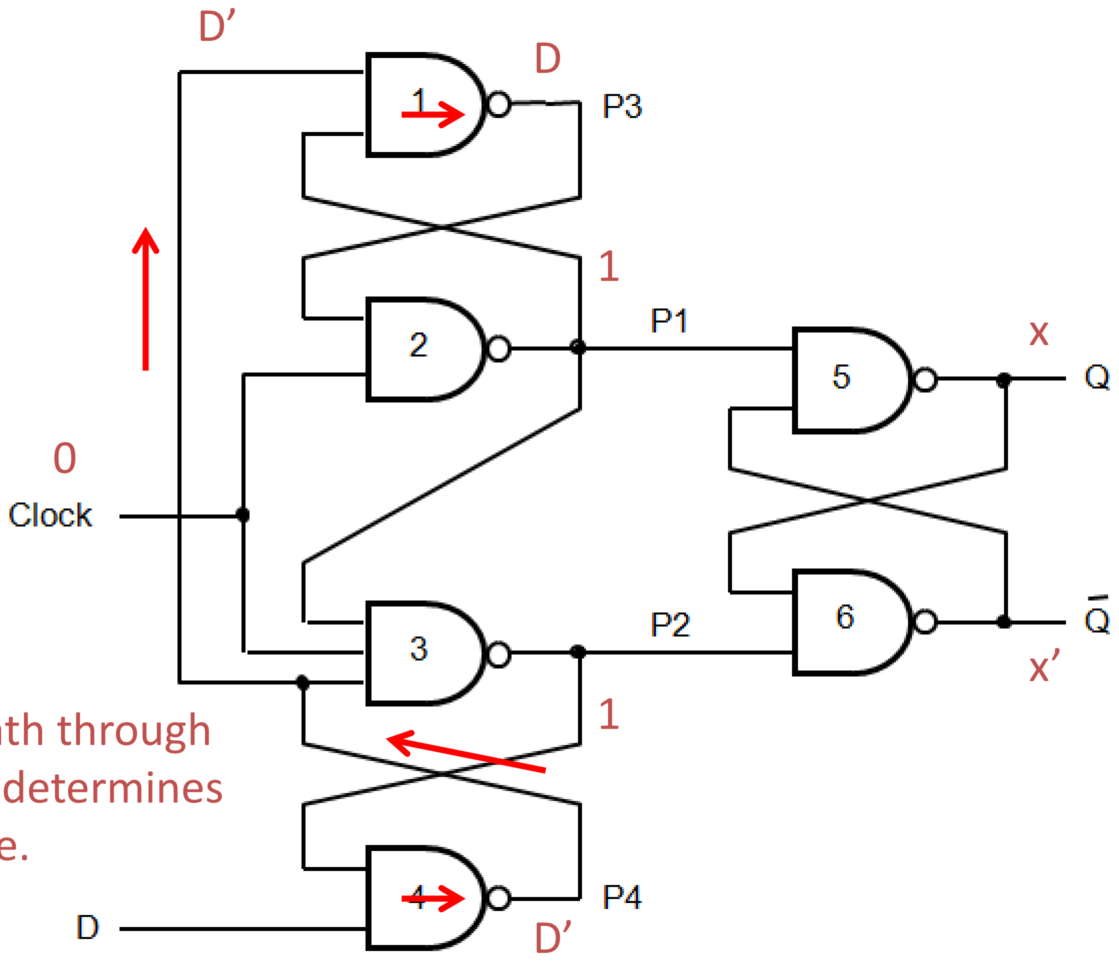
(a) Circuit

(b) Graphical symbol

Figure 5.11. A positive-edge-triggered D flip-flop.

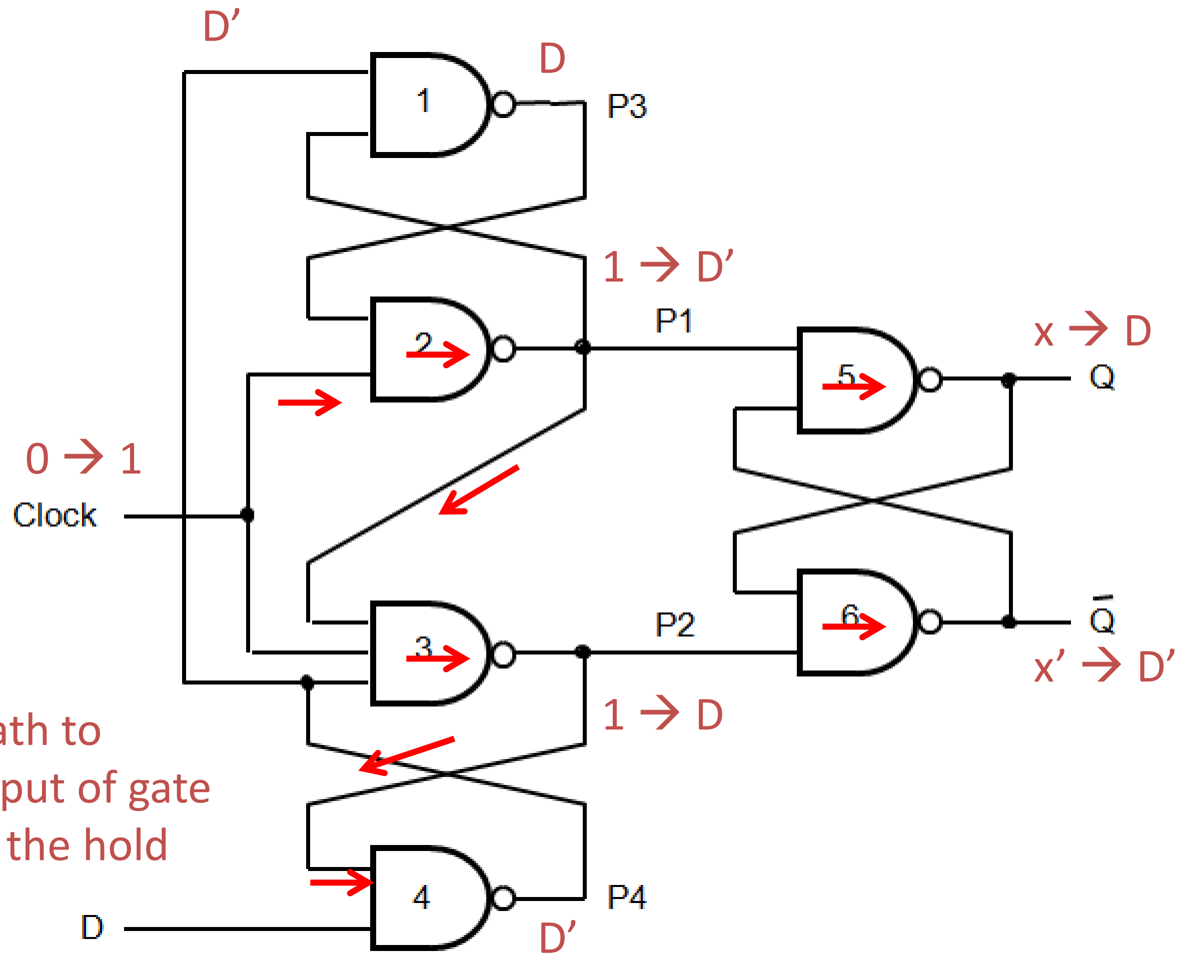


Steady-state clock = 0.



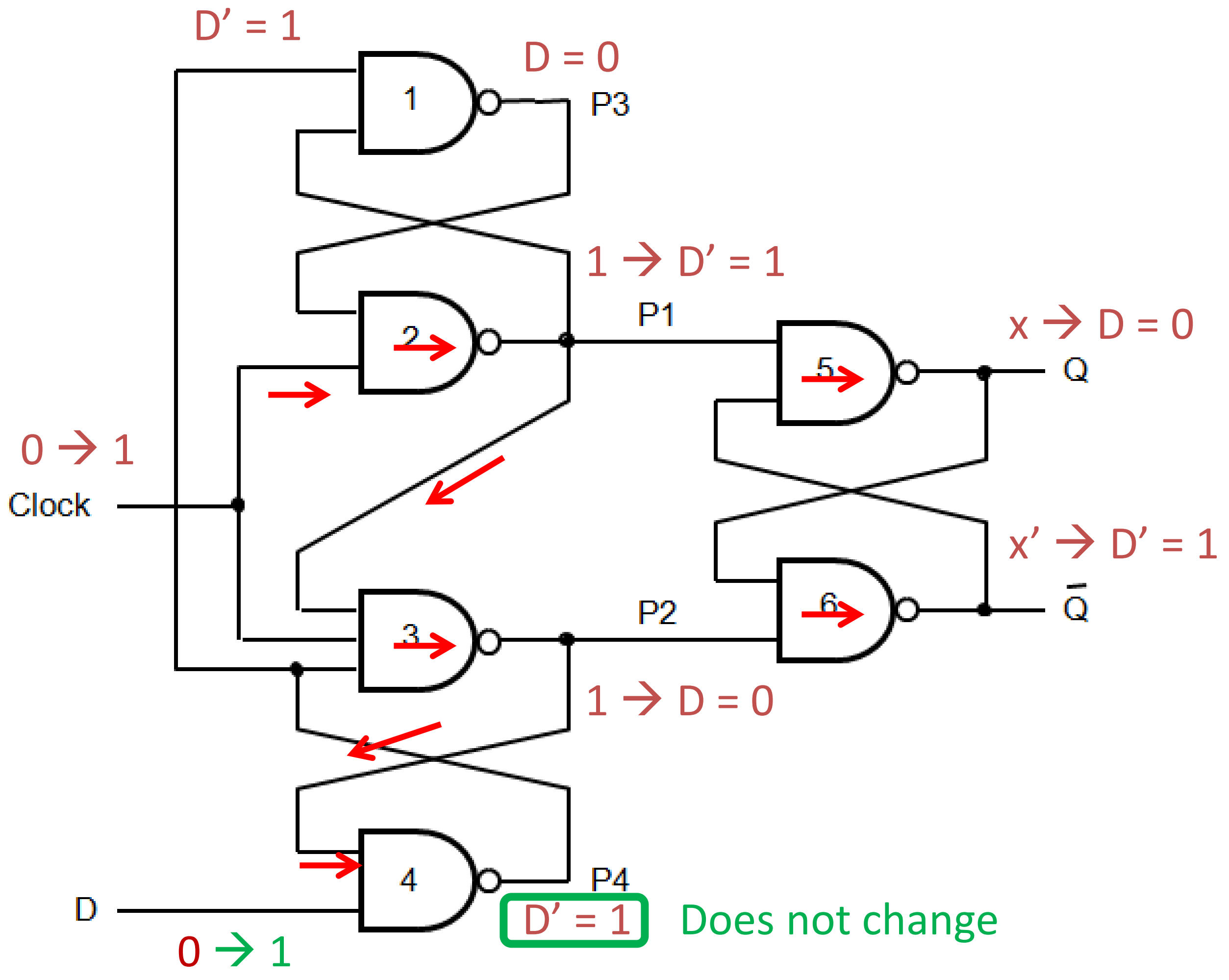
The critical path through gates 4 and 1 determines the setup time.

Steady-state clock = 0.

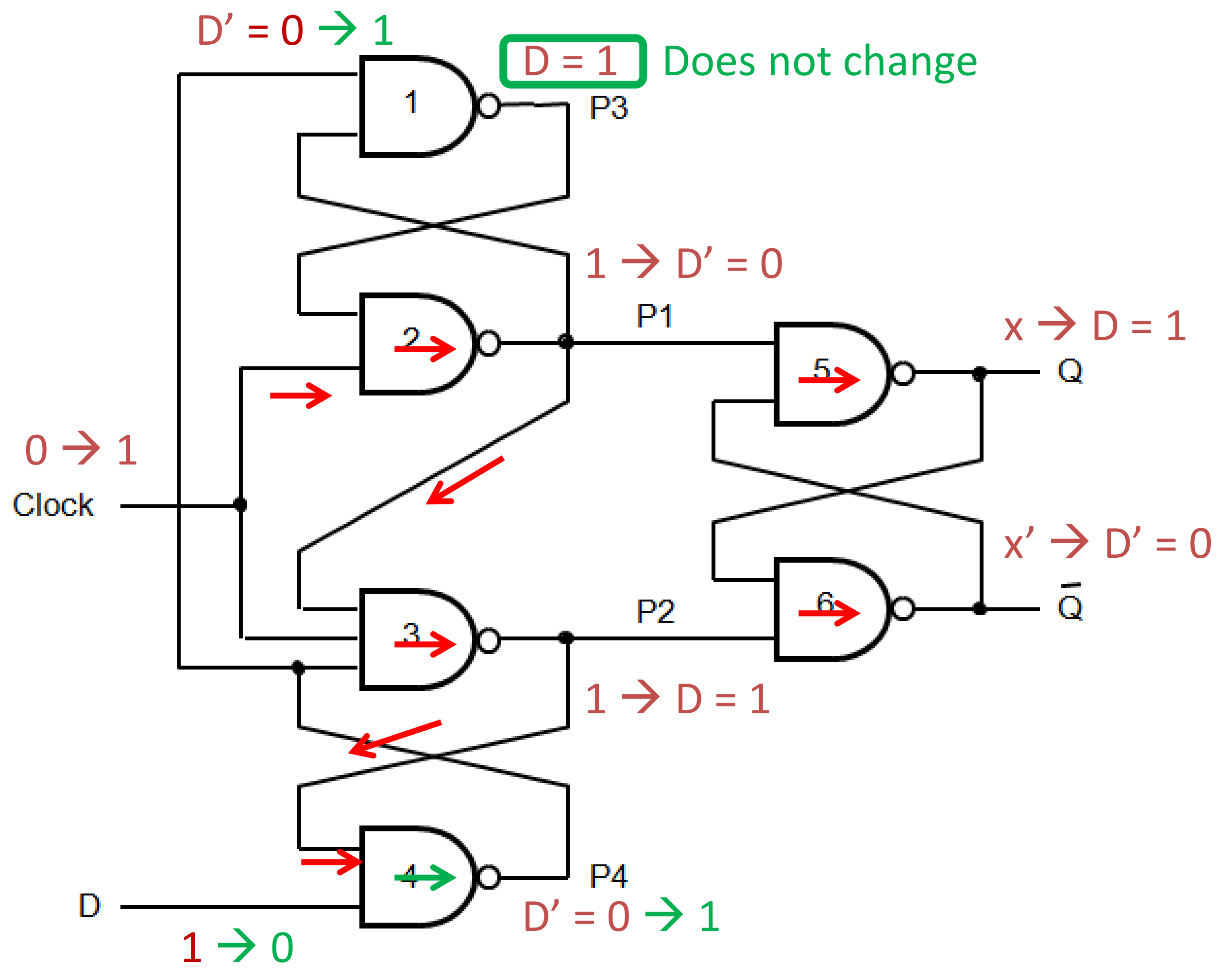


The critical path to disable the input of gate 4 determines the hold time.

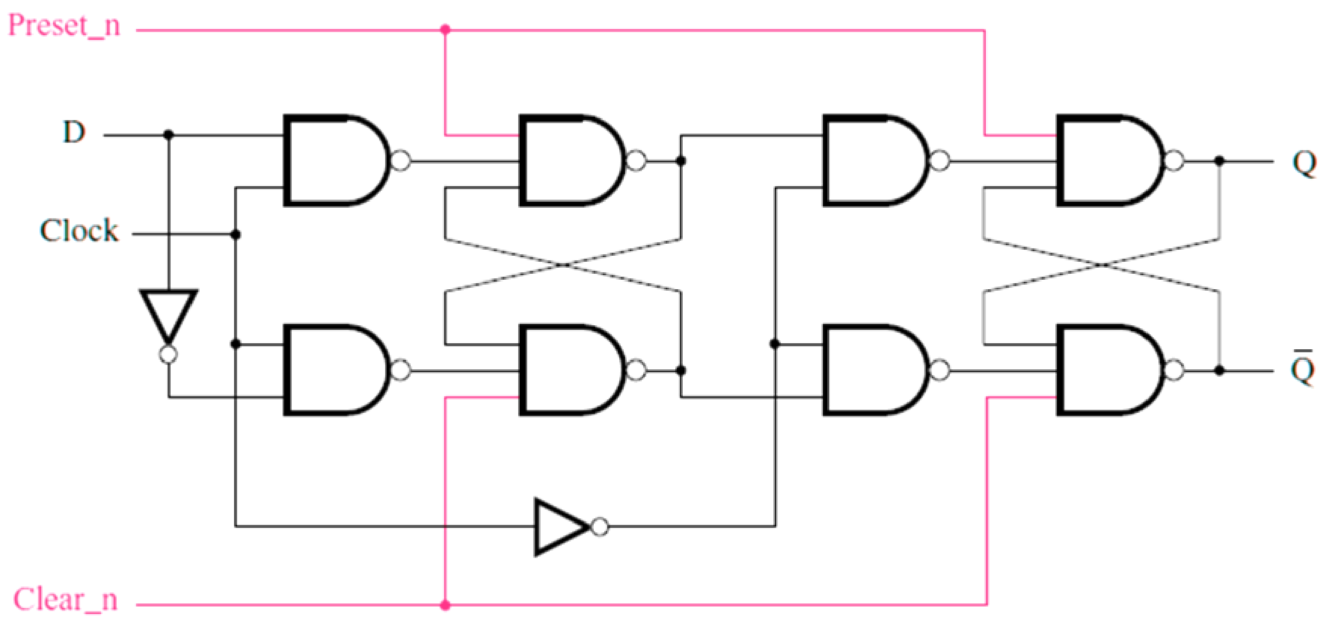
Clock transition 0 → 1



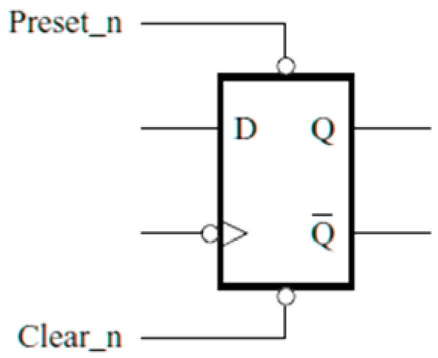
D transitions 0 → 1 after the hold time



D transitions 1 → 0 after the hold time

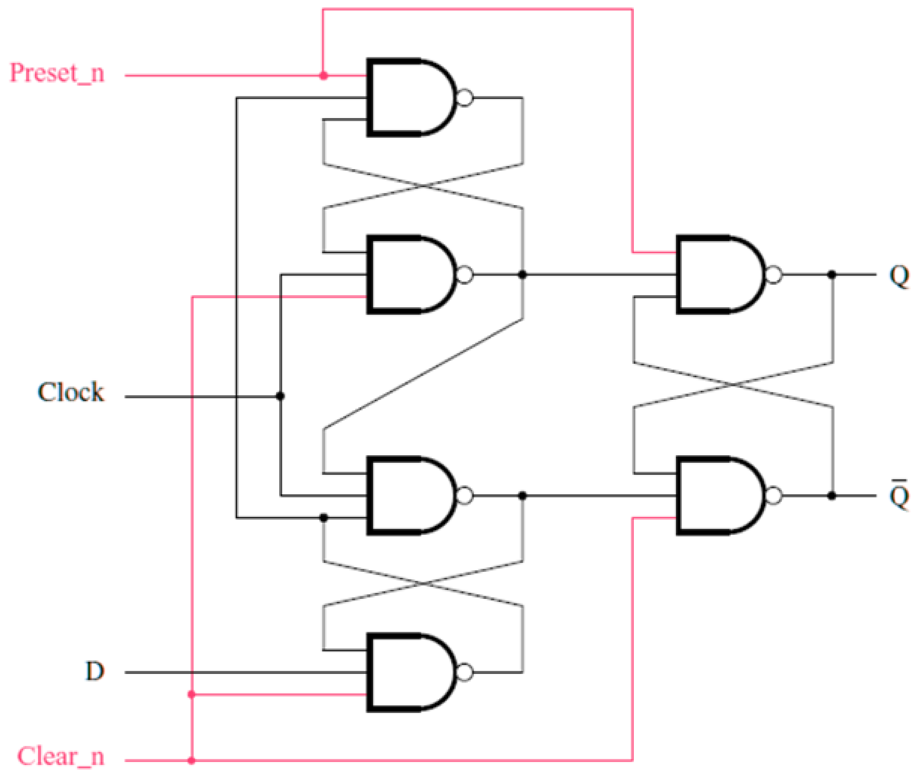


(a) Circuit

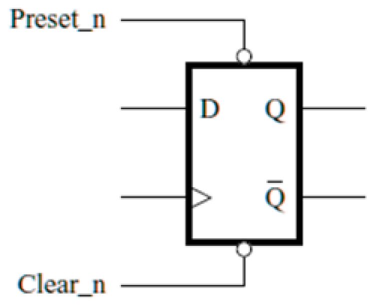


(b) Graphical symbol

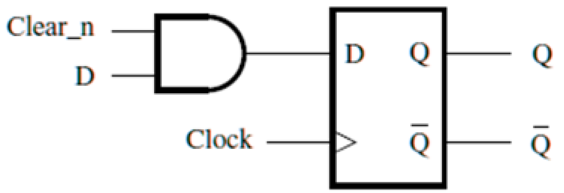
Figure 5.12. Master-slave D flip-flop with *Clear* and *Preset*.



(a) Circuit

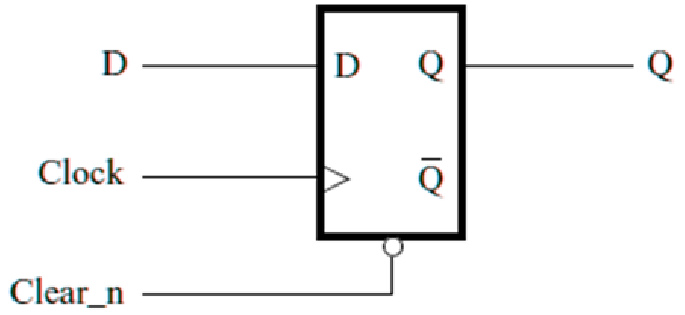


(b) Graphical symbol

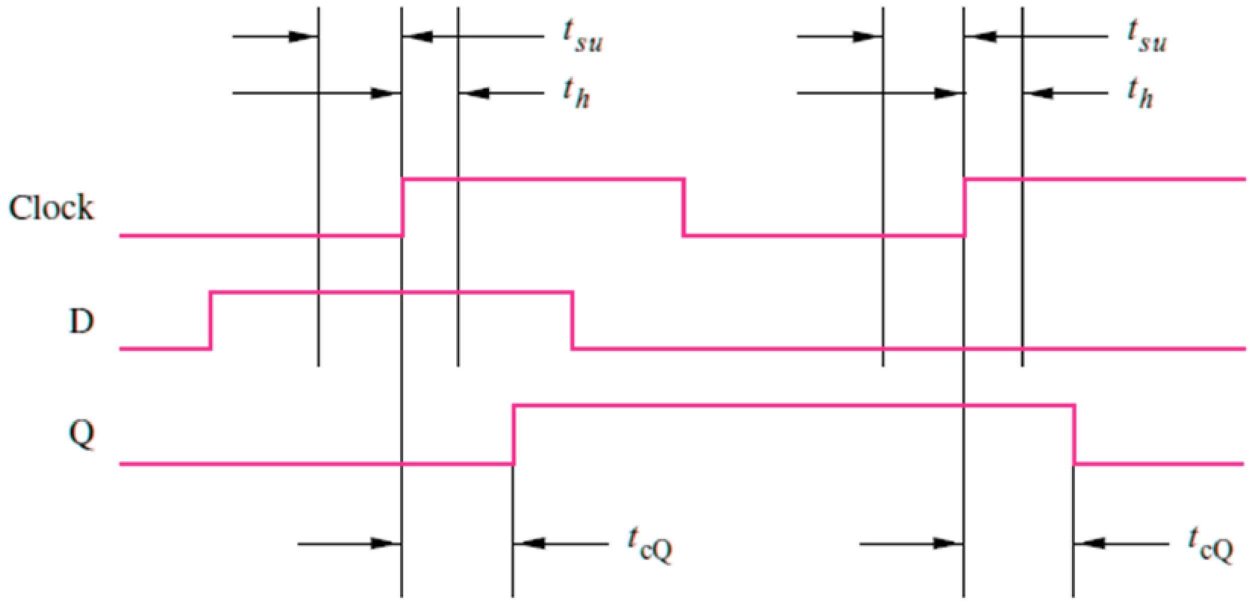


(c) Adding a synchronous clear

Figure 5.13. Positive-edge-triggered D flip-flop with *Clear* and *Preset*.



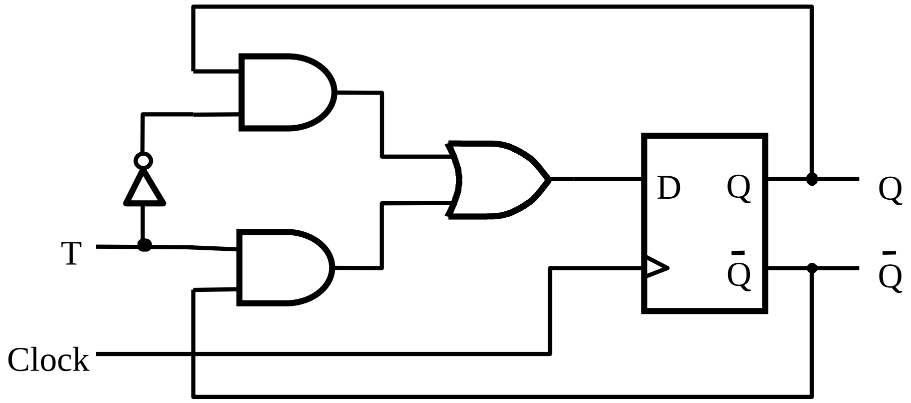
(a) D flip-flop with asynchronous clear



(b) Timing diagram

Figure 5.14. Timing for a flip-flop.

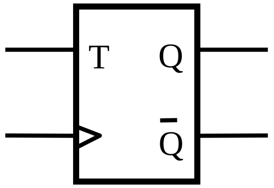
Figure 5.15. T flip-flop.



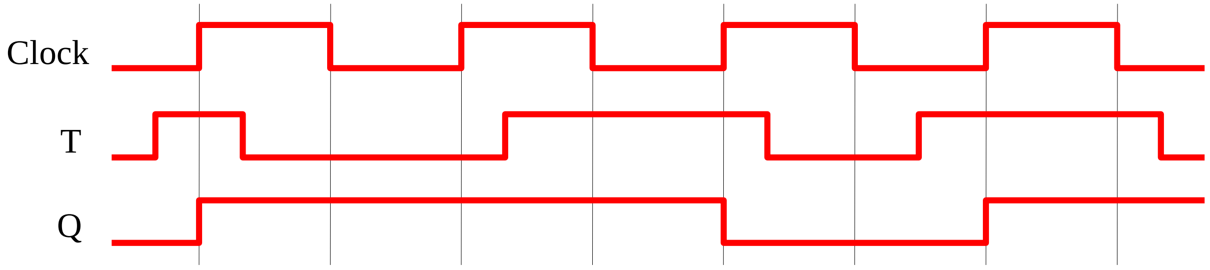
(a) Circuit

T	$Q(t+1)$
0	$Q(t)$
1	$\overline{Q}(t)$

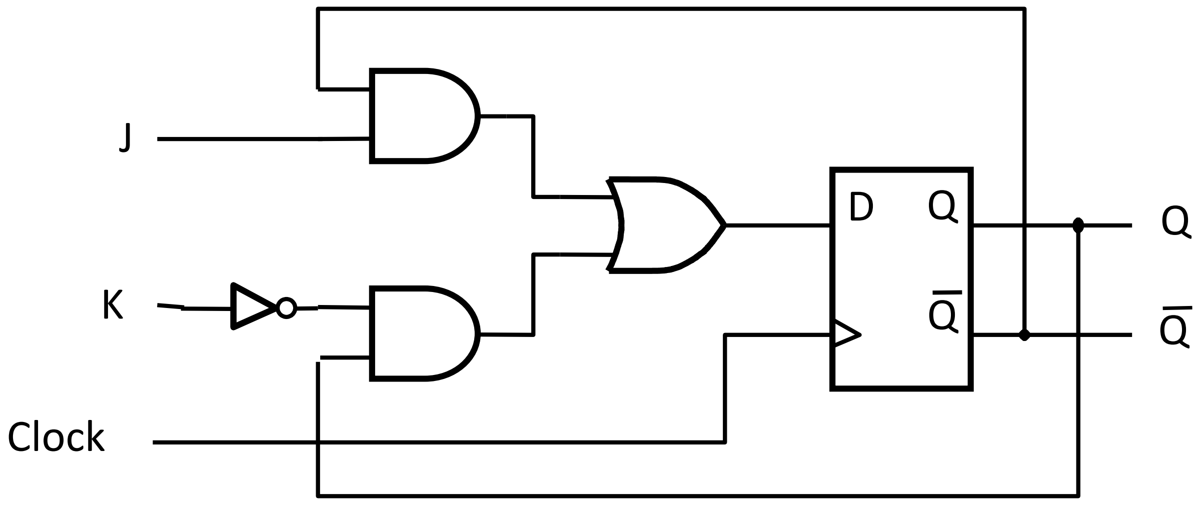
(b) Truth table



(c) Graphical symbol



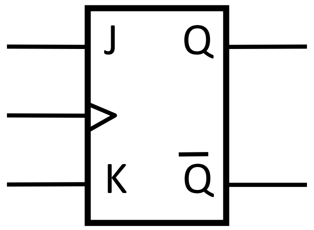
(d) Timing diagram



(a) Circuit

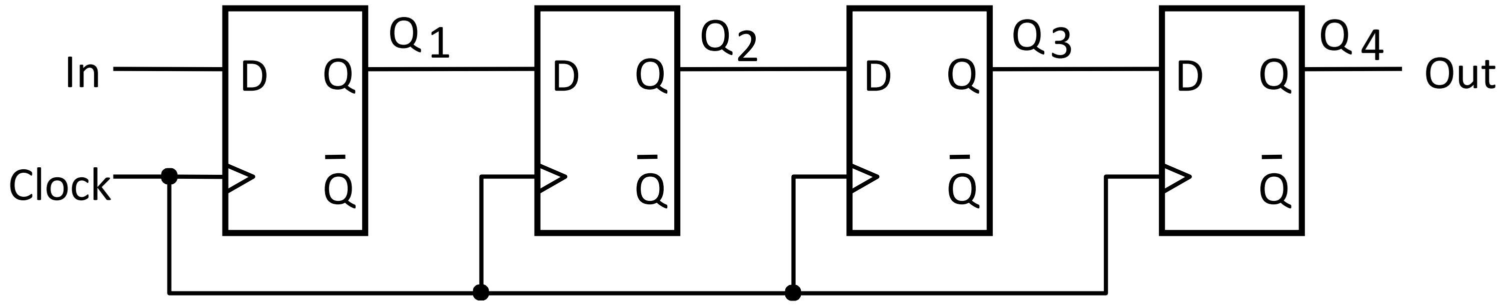
J	K	Q (t+1)
0	0	Q (t)
0	1	0
1	0	1
1	1	$\overline{Q}$ (t)

(b) Truth table



(c) Graphical symbol

Figure 5.16. JK flip-flop.



(a) Circuit

	In	Q_1	Q_2	Q_3	Q_4 = Out
t_0	1	0	0	0	0
t_1	0	1	0	0	0
t_2	1	0	1	0	0
t_3	1	1	0	1	0
t_4	1	1	1	0	1
t_5	0	1	1	1	0
t_6	0	0	1	1	1
t_7	0	0	0	1	1

(b) A sample sequence

Figure 5.17. A simple shift register.

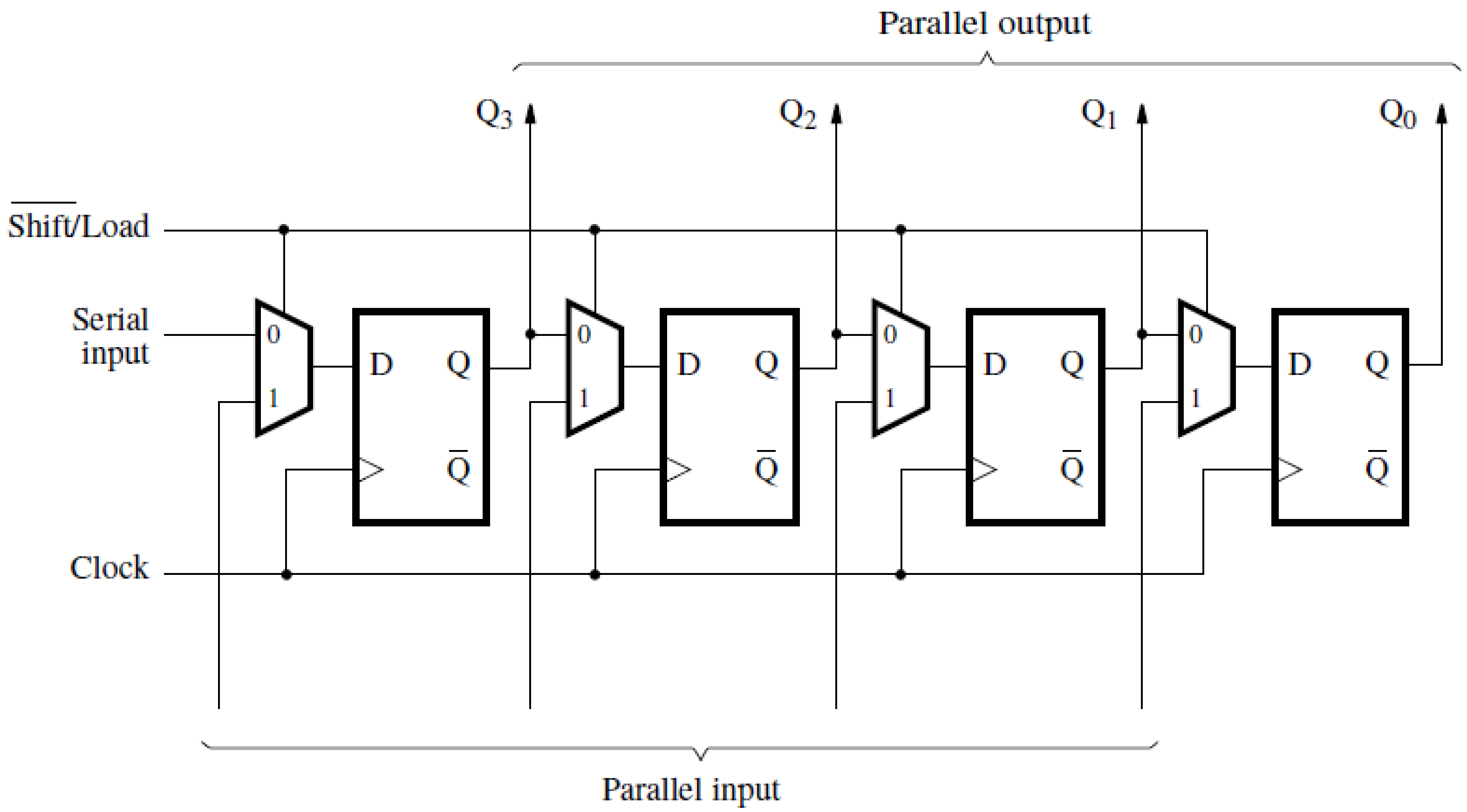
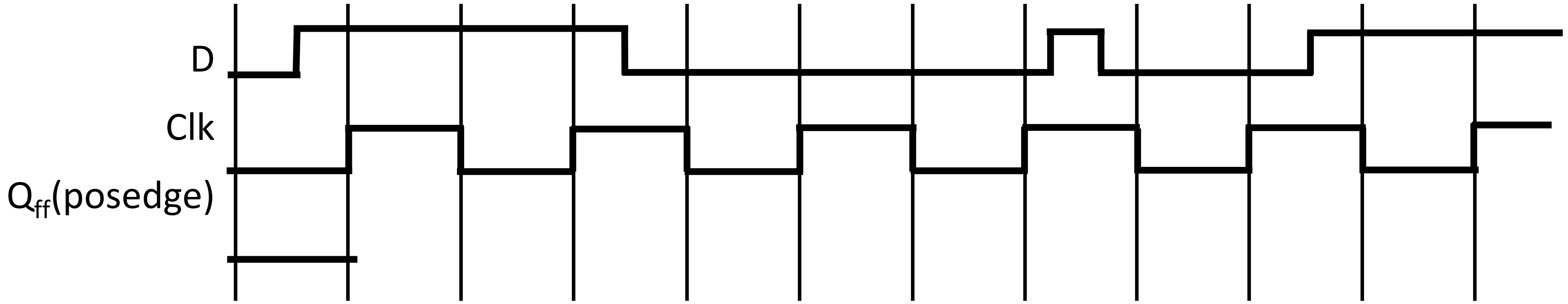


Figure 5.18. Parallel-access shift register.

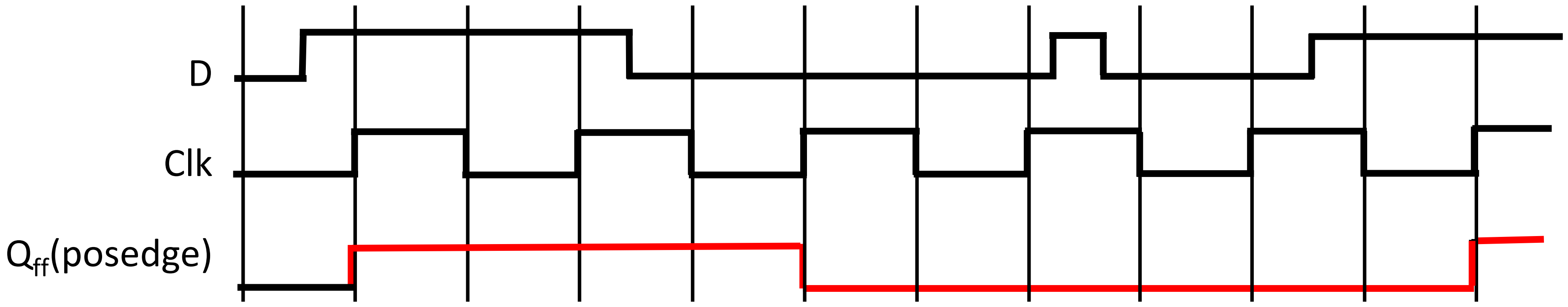
Basic D Flip Flop

```
module DFlipFlop( input clock, D, output reg Q );

   always @( posedge clock )
      Q <= D;


endmodule
```



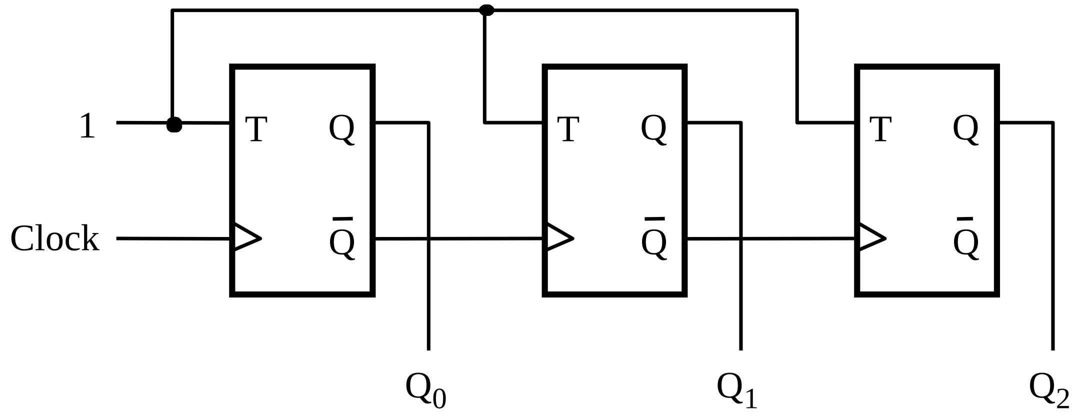
Basic D Flip Flop

```
module DFlipFlop( input clock, D, output reg Q );

   always @( posedge clock )
      Q <= D;


endmodule
```

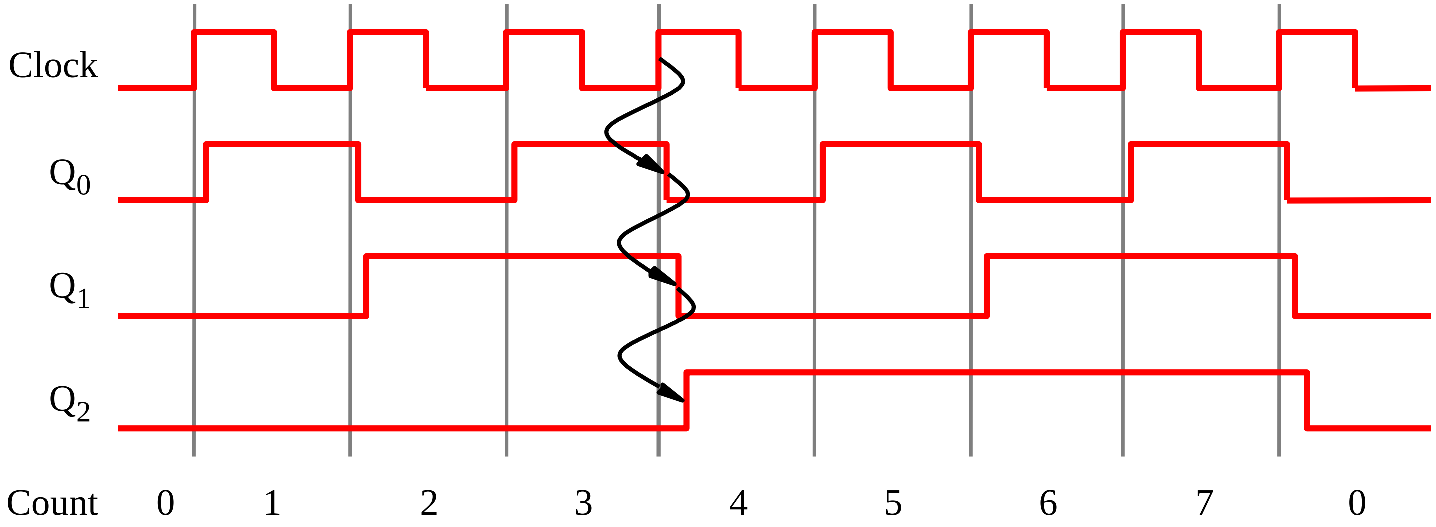


D Flip Flop w/Synchronous Reset

```
module DFlipFlop2( input clock, D, reset, output reg Q );

   always @( posedge clock )
      Q <= reset ? 0 : D;

endmodule
```

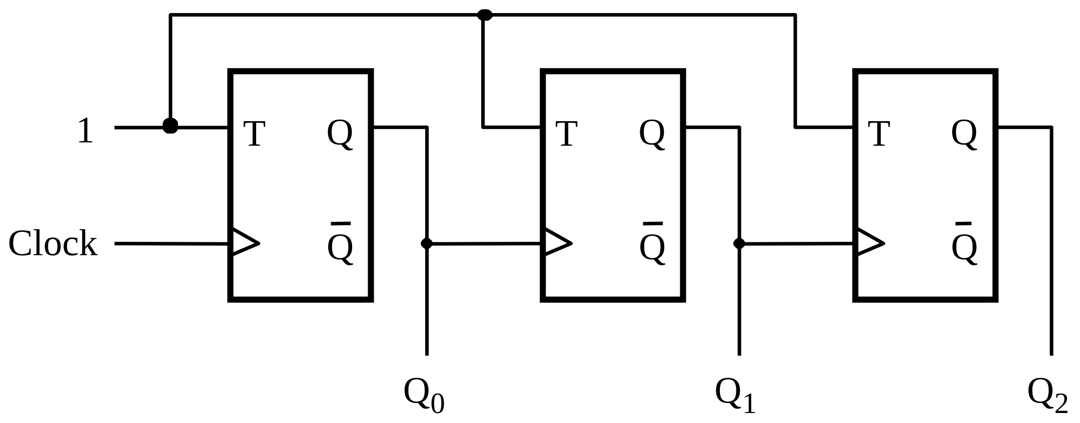


(a) Circuit

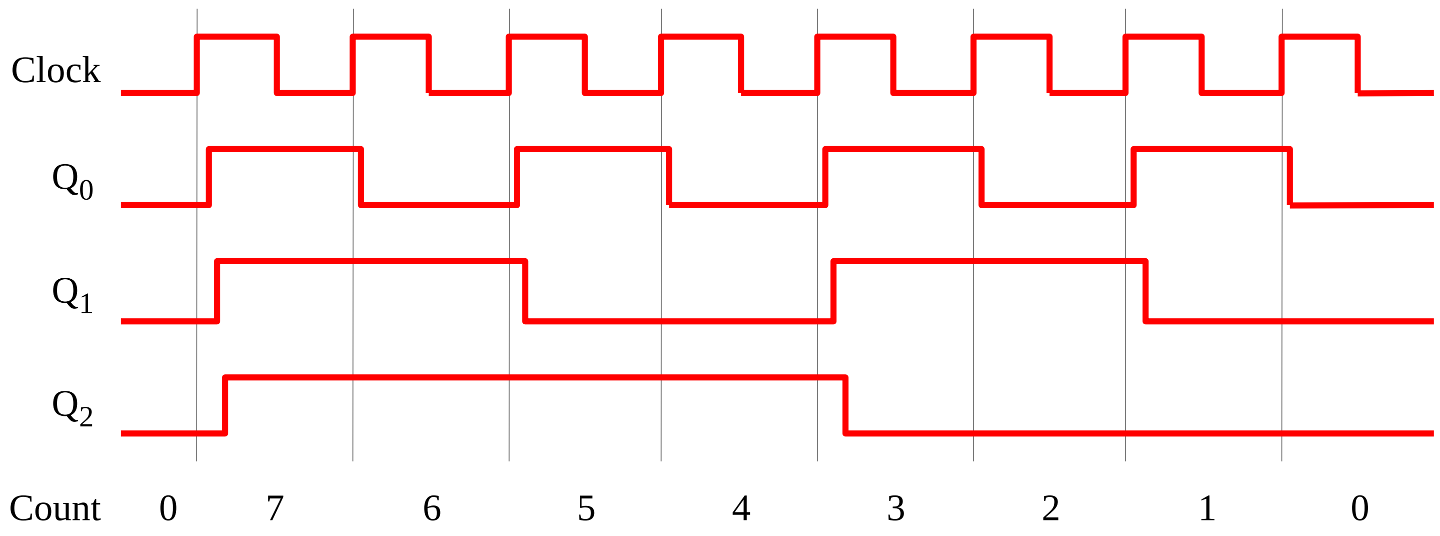


(b) Timing diagram

Figure 5.19. A three-bit up-counter.



(a) Circuit



(b) Timing diagram

Figure 5.20. A three-bit down-counter.

Clock cycle	Q_2	Q_1	Q_0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
8	0	0	0

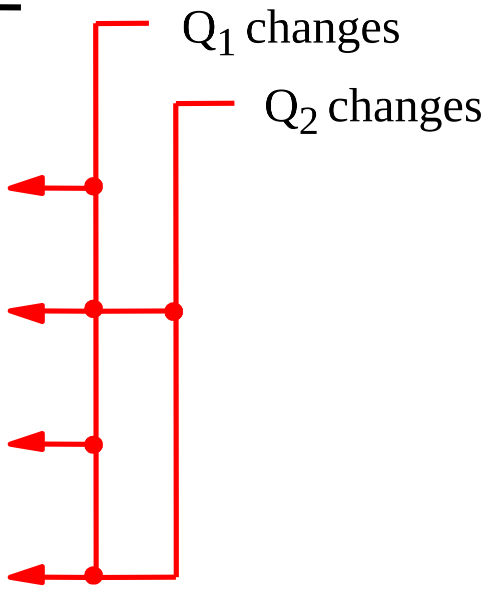
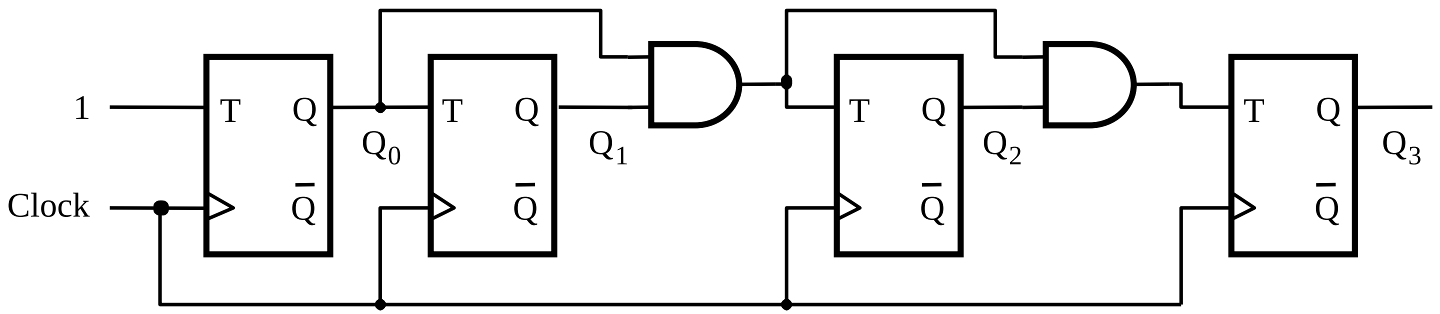
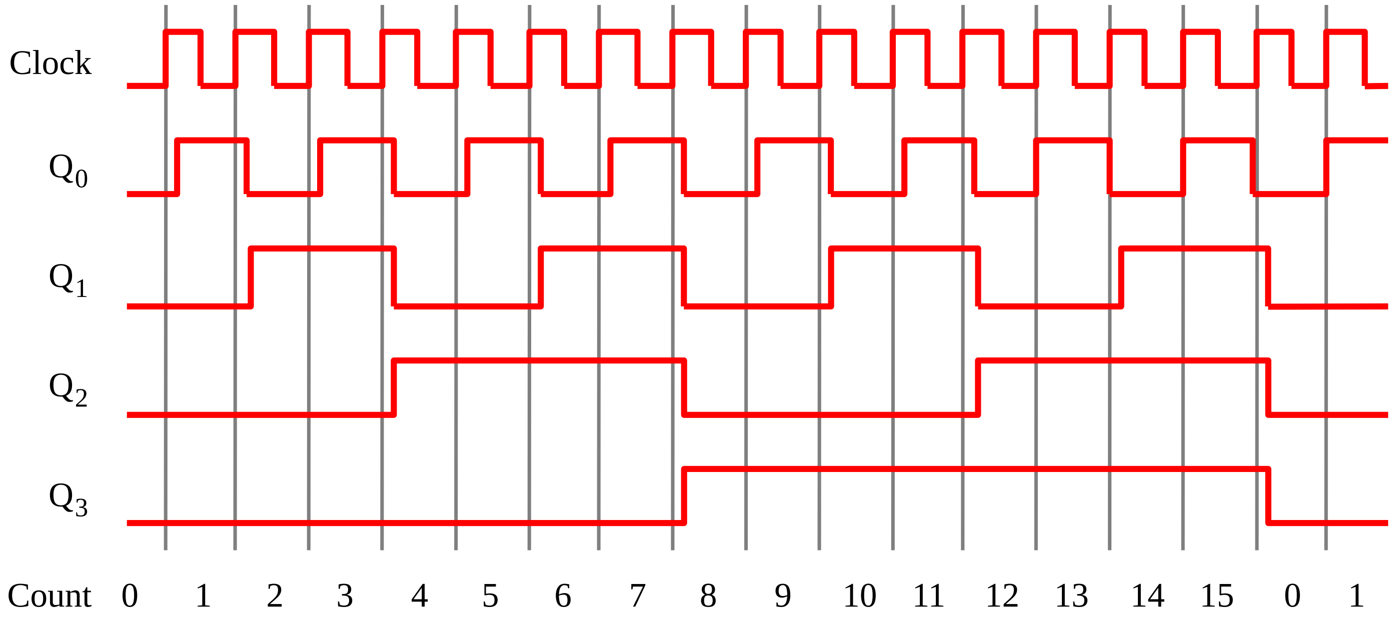


Table 5.1. Derivation of the synchronous up-counter.



(a) Circuit



(b) Timing diagram

Figure 5.21. A four-bit synchronous up-counter.

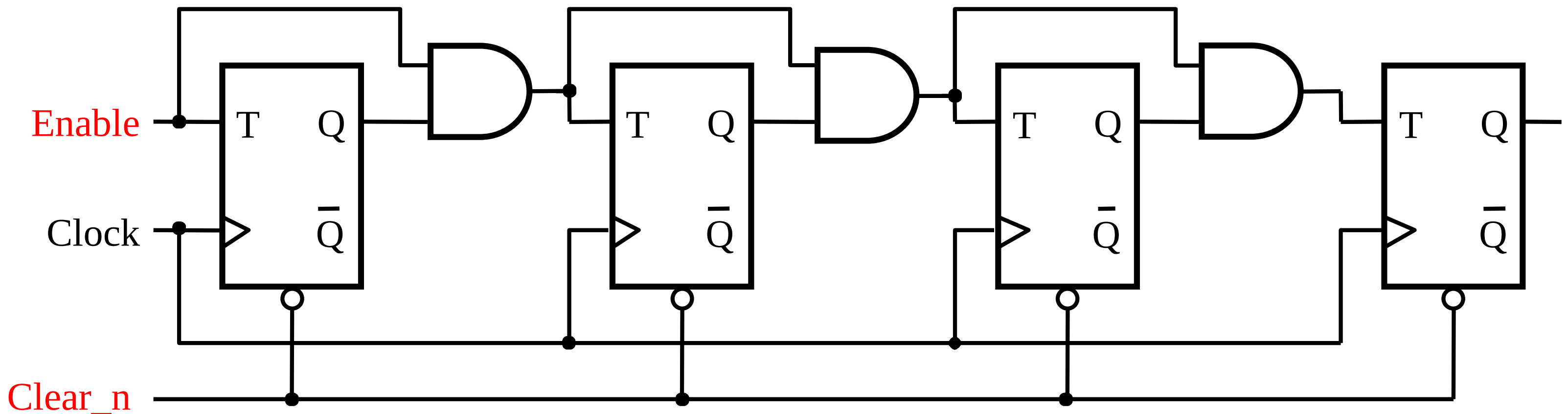


Figure 5.22. Inclusion of Enable and Clear capability.

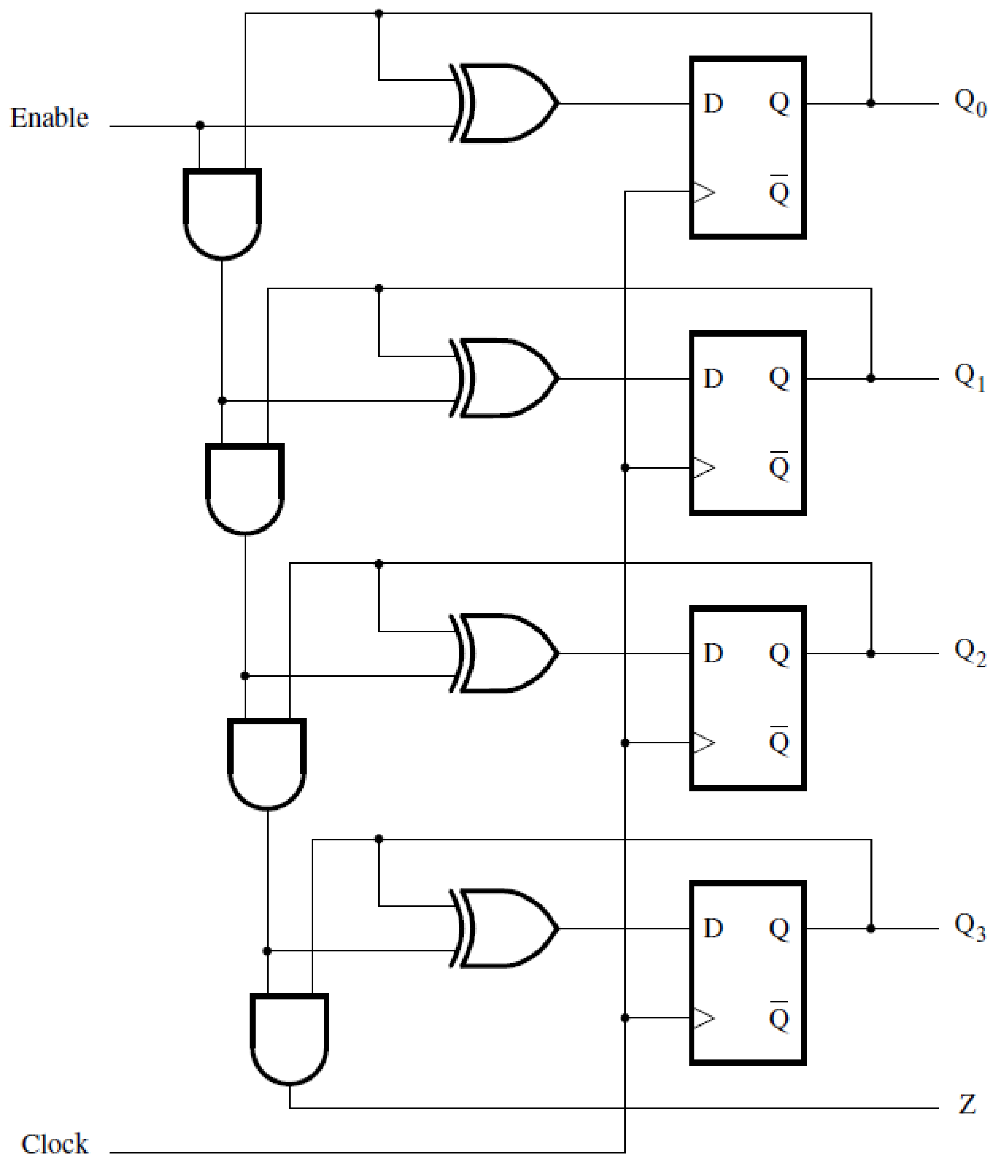


Figure 5.23. A four-bit counter with D flip-flops.

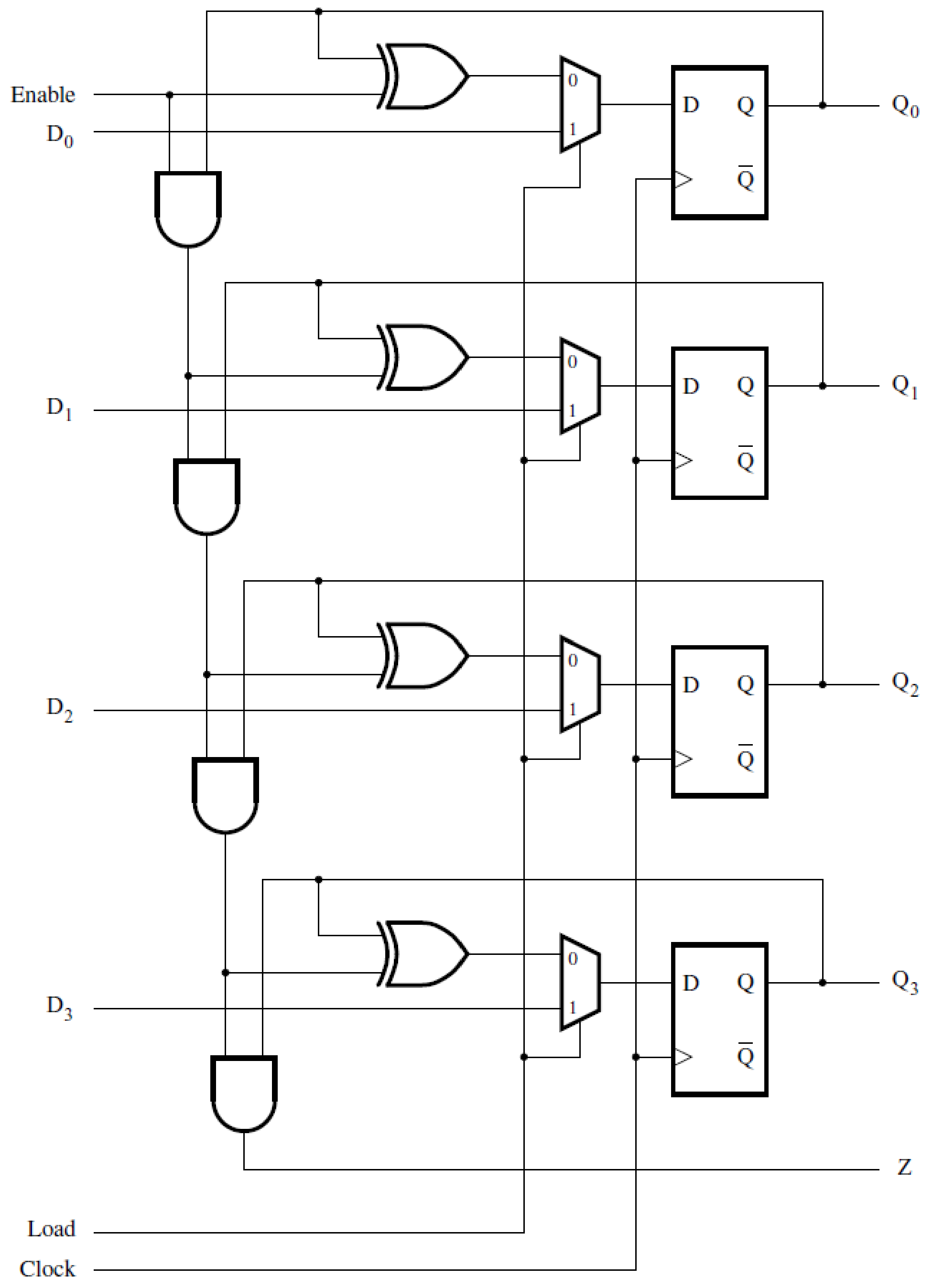
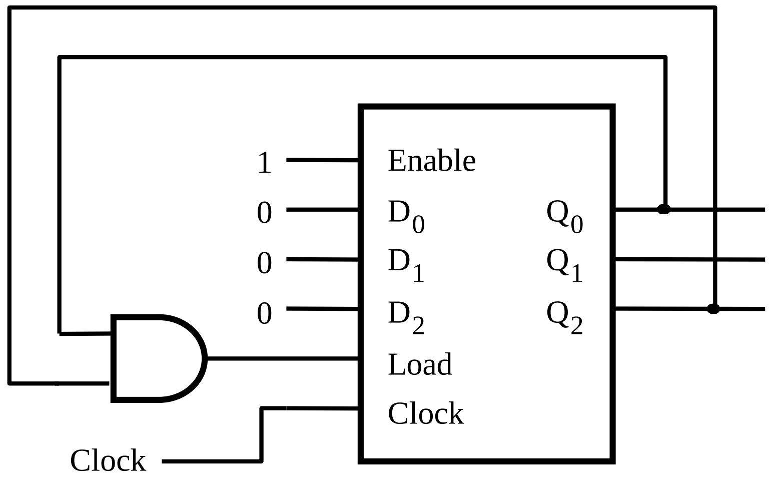
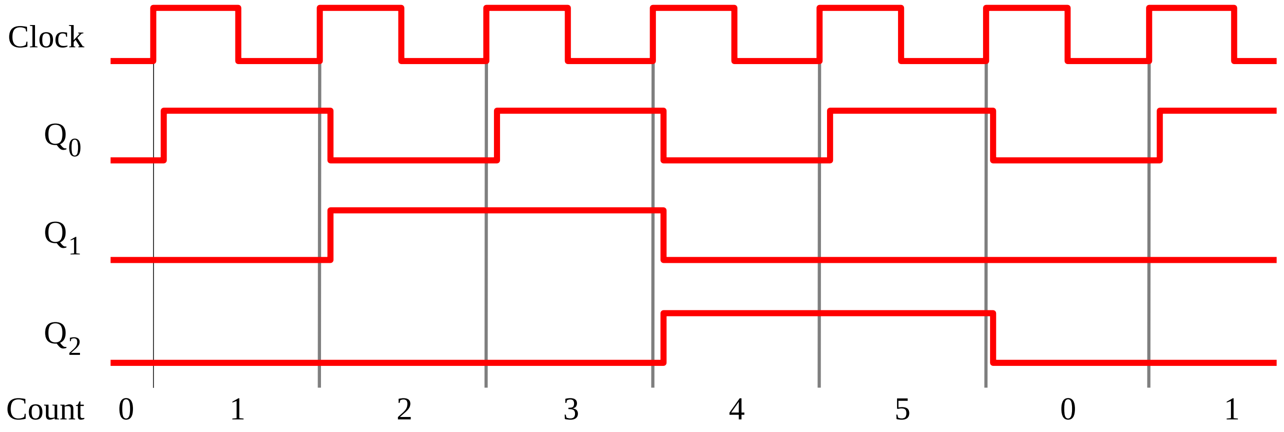


Figure 5.24. A counter with parallel-load capability.

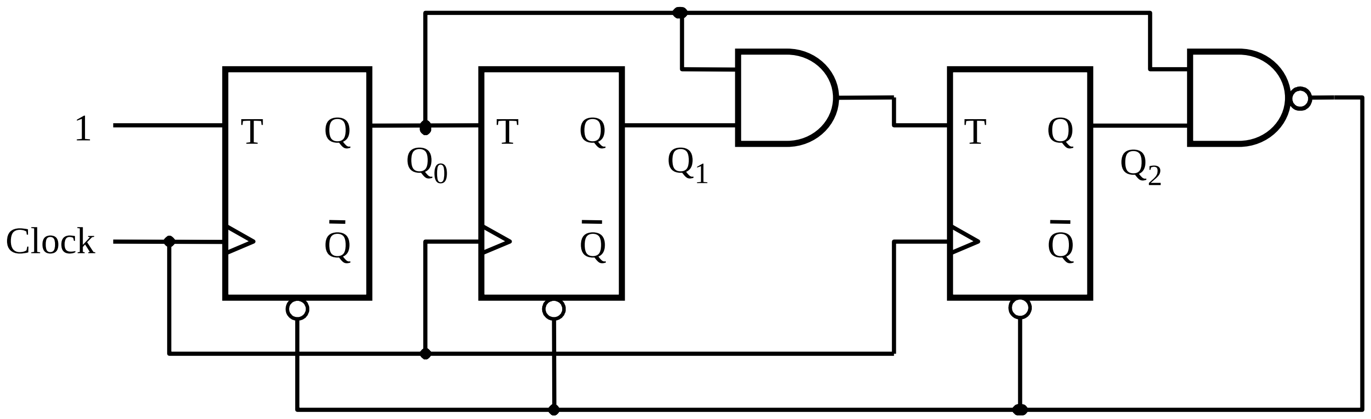


(a) Circuit

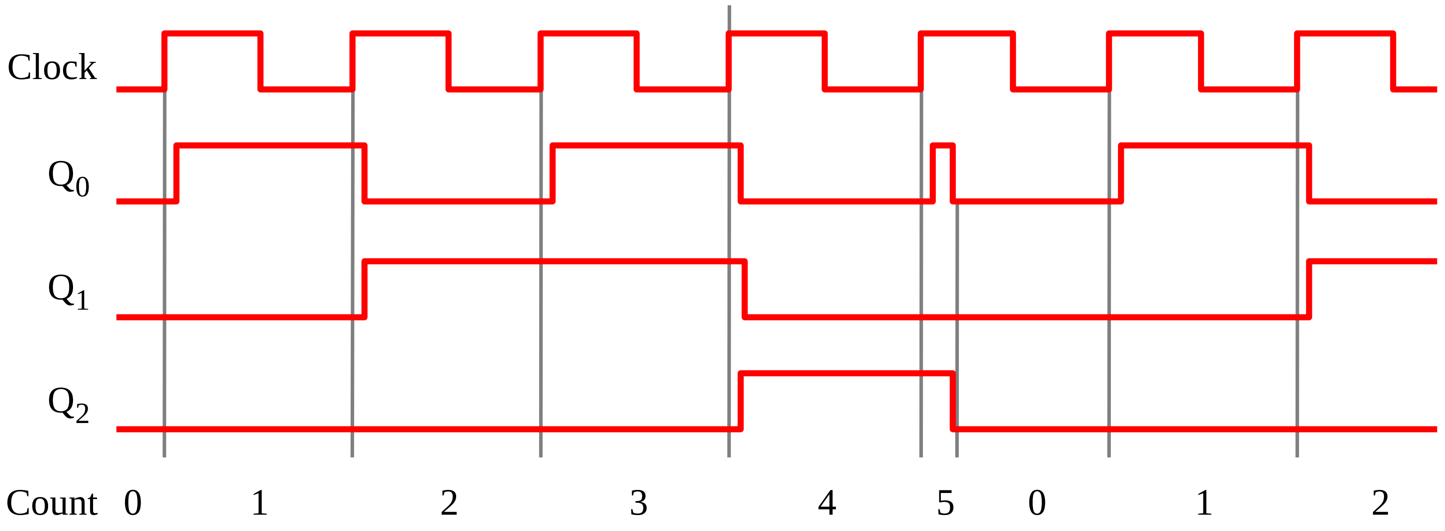


(b) Timing diagram

Figure 5.25. A modulo-6 counter with synchronous reset.



(a) Circuit



(b) Timing diagram

Figure 5.26. A modulo-6 counter with asynchronous reset.

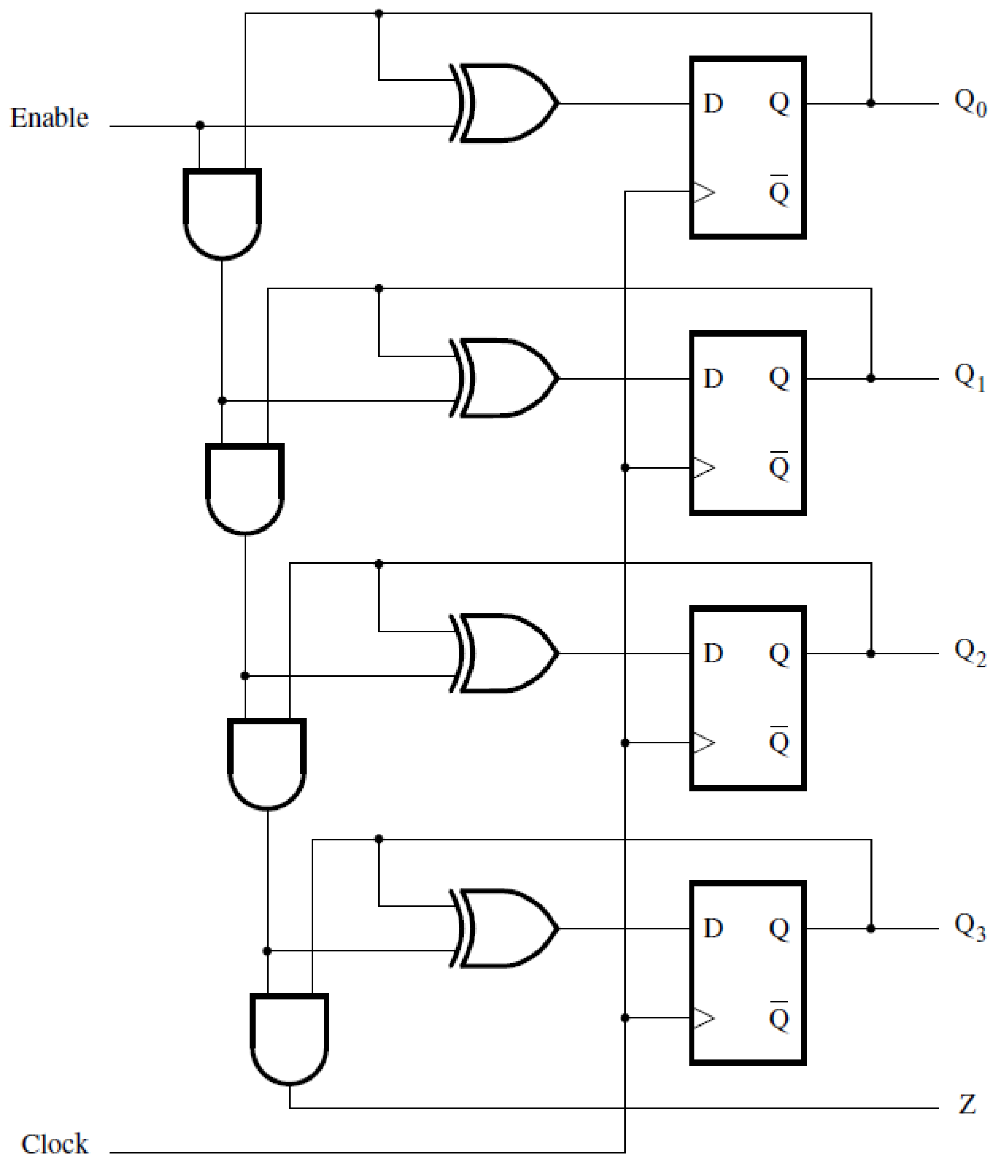


Figure 5.23. A four-bit counter with D flip-flops.

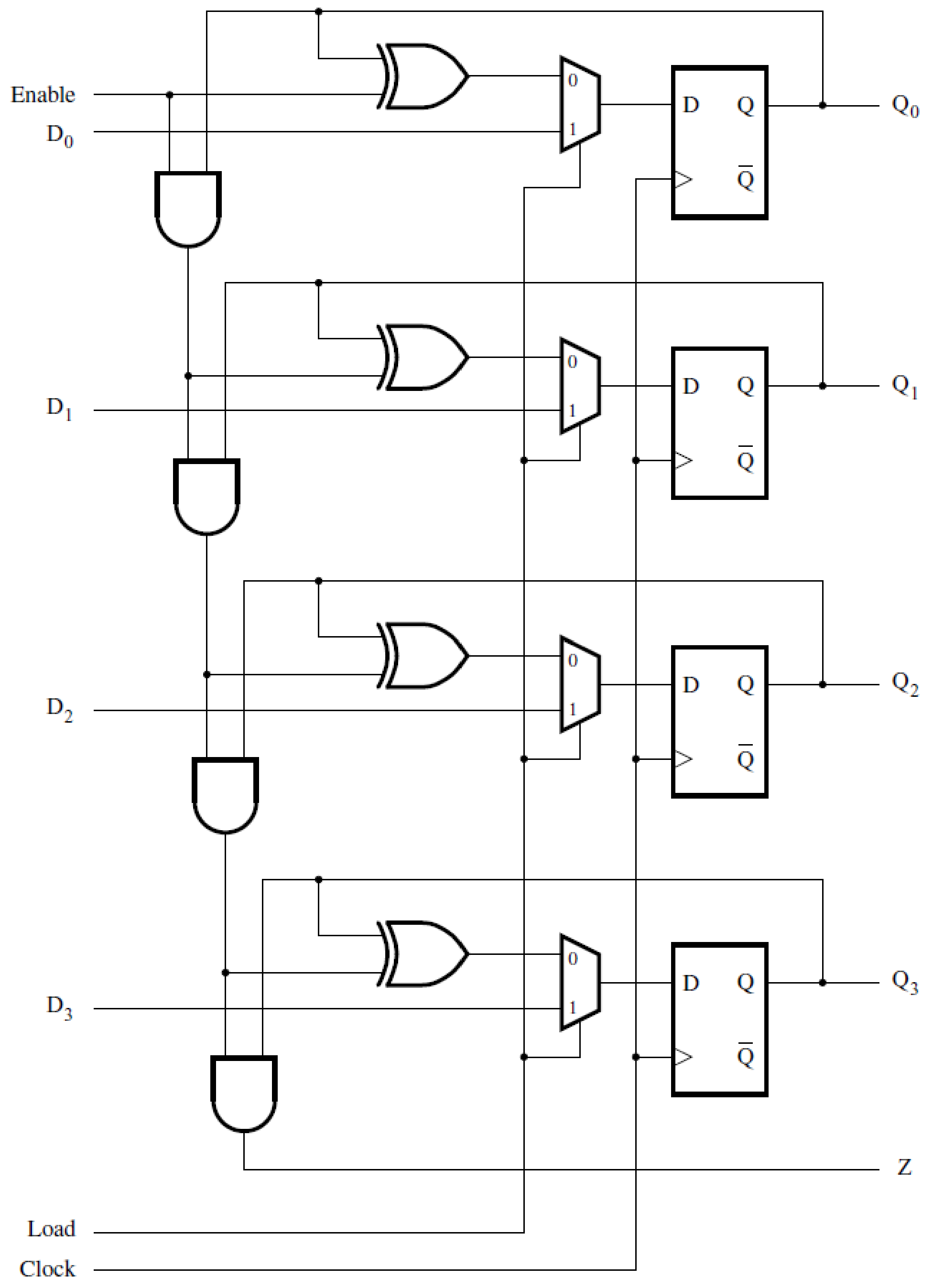
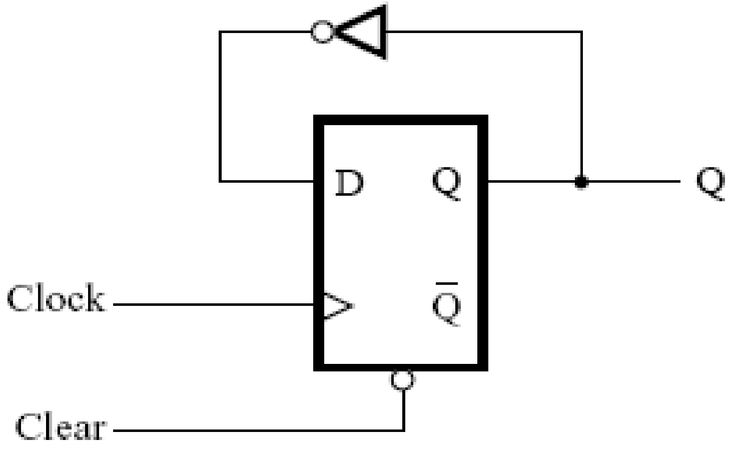


Figure 5.24. A counter with parallel-load capability.



Assume:

$t_{su} = 0.6$ ns

$t_h = 0.4$ ns

$0.8 \text{ ns} <= t_{cQ} <= 1.0 \text{ ns}$

$t_{gate} = 1.0 + 0.1k$

where

k = number of inputs

$T_{min} = t_{cQmax} + t_{NOT} + t_{su}$

$= 1.0 + 1.1 + 0.6 = 2.7$ ns

$F_{max} = 1/2.7$ ns $= 370.37$ MHz.

Figure 5.66. A simple flip-flop circuit.

Checking hold time

Assume:

$t_{su} = 0.6$ ns

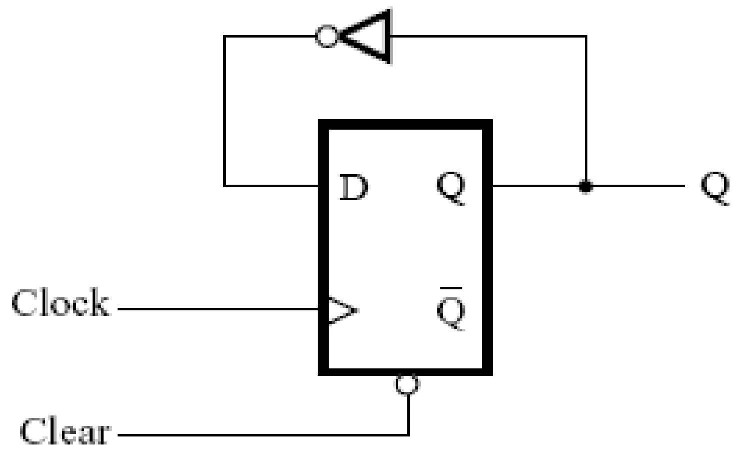
$t_h = 0.4$ ns

$0.8 \text{ ns} <= t_{cQ} <= 1.0$ ns

$t_{gate} = 1.0 + 0.1k$

where

k = number of inputs



Shortest delay = $t_{cQmin} + t_{NOT}$
$= 0.8 + 1.1 = 1.9$ ns

Since 1.9 ns > $t_h = 0.4$ ns,

no hold violation.

Figure 5.66. A simple flip-flop circuit.

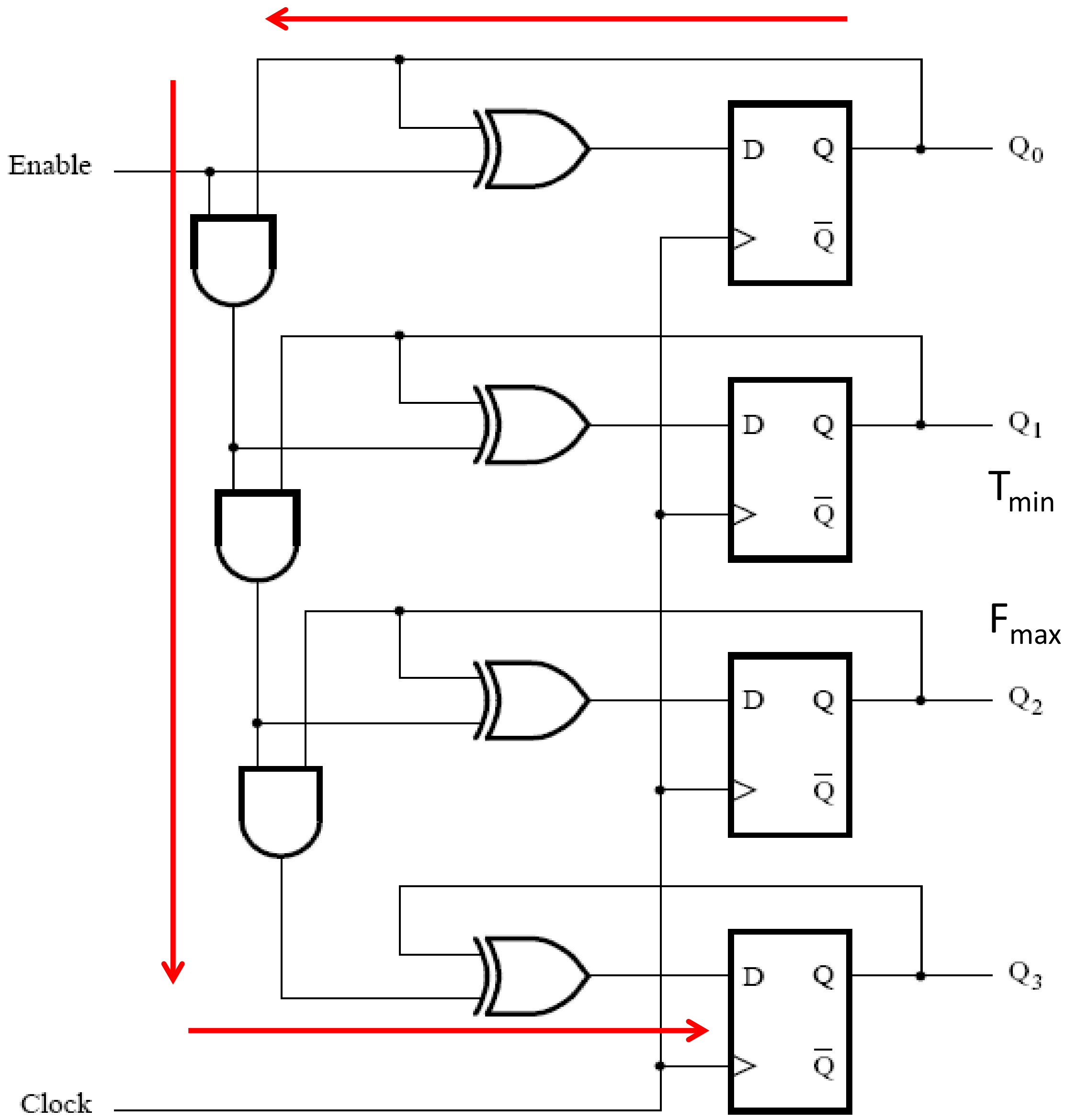
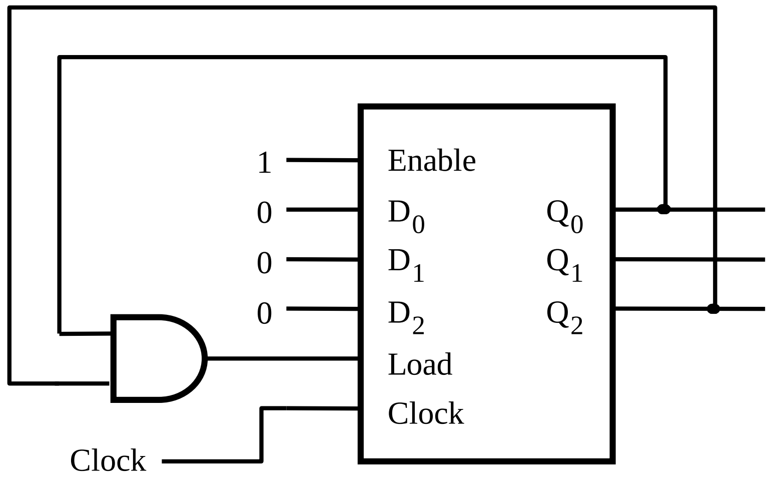


Figure 5.67. A 4-bit counter *critical path*.

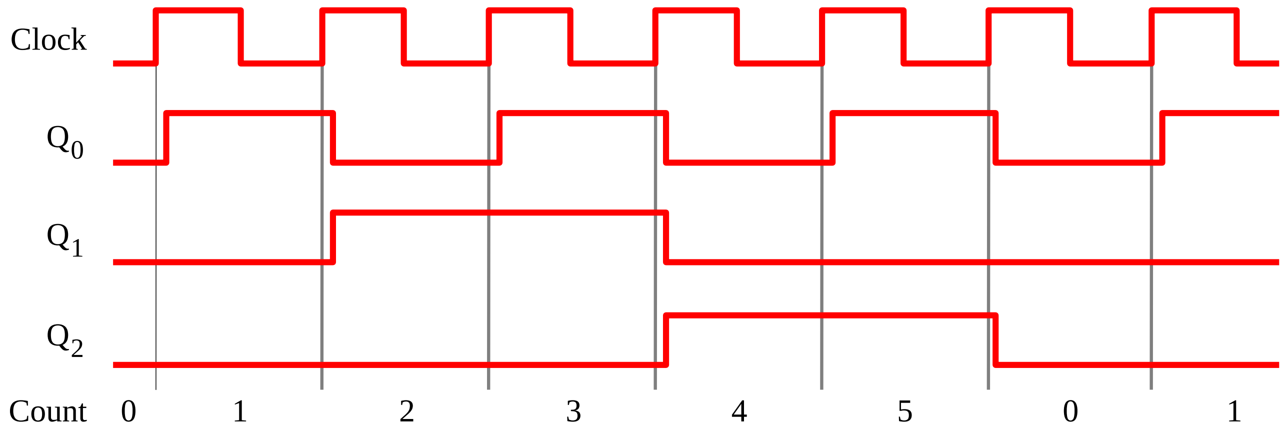
$T_{min} = t_{cQmax} + 3\ t_{AND} + t_{XOR} + t_{su}$

$= 1.0 + 3(1.2) + 1.2 + 0.6 = 6.4$ ns

$F_{max} = 1/6.4$ ns $= 156.25$ MHz.

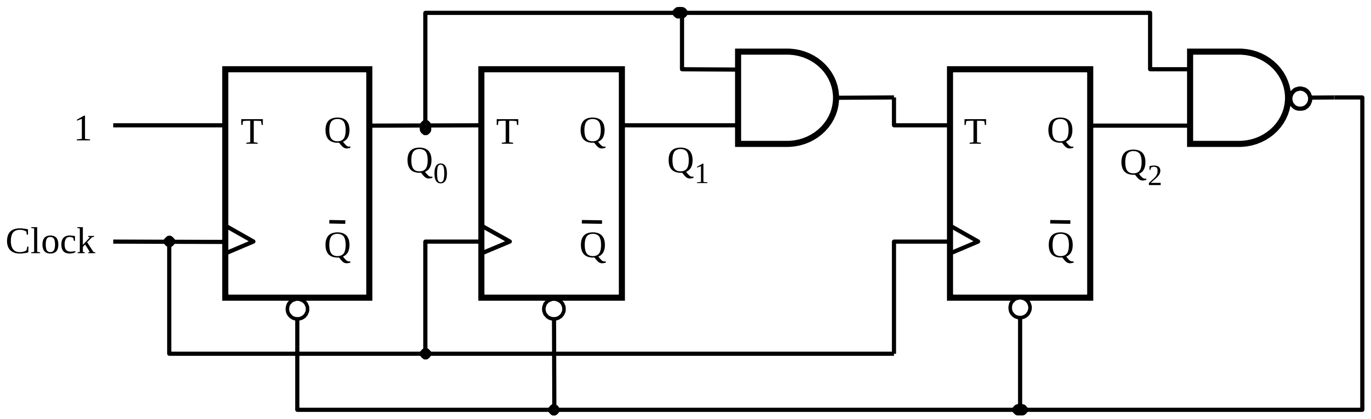


(a) Circuit

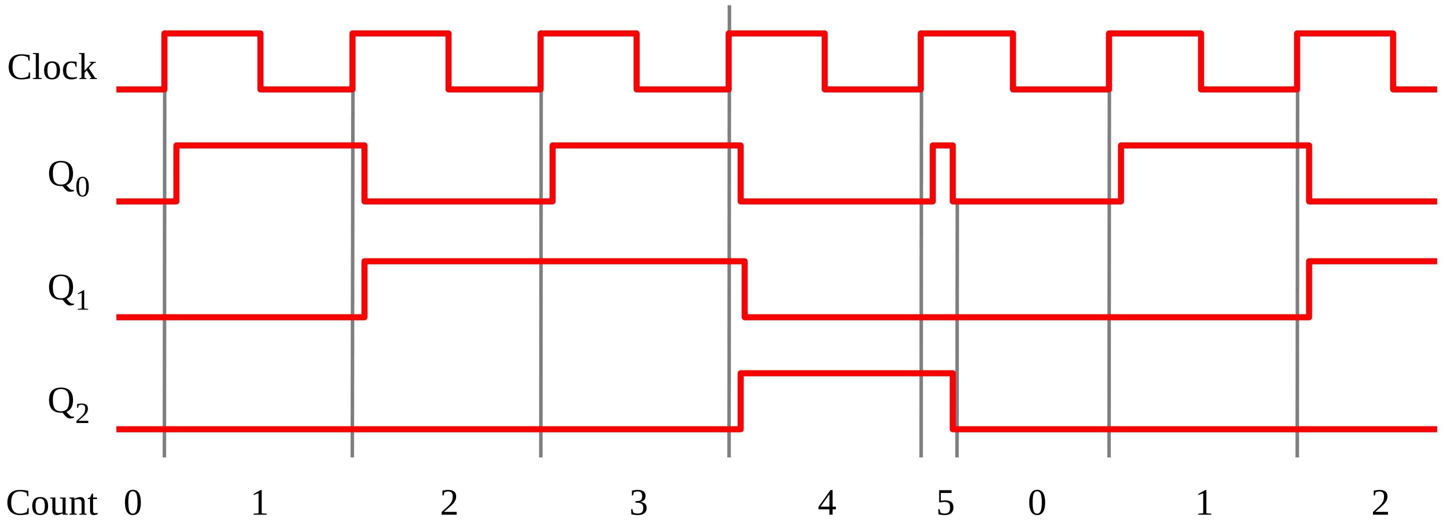


(b) Timing diagram

Figure 5.25. A modulo-6 counter with synchronous reset.



(a) Circuit



(b) Timing diagram

Figure 5.26. A modulo-6 counter with asynchronous reset.

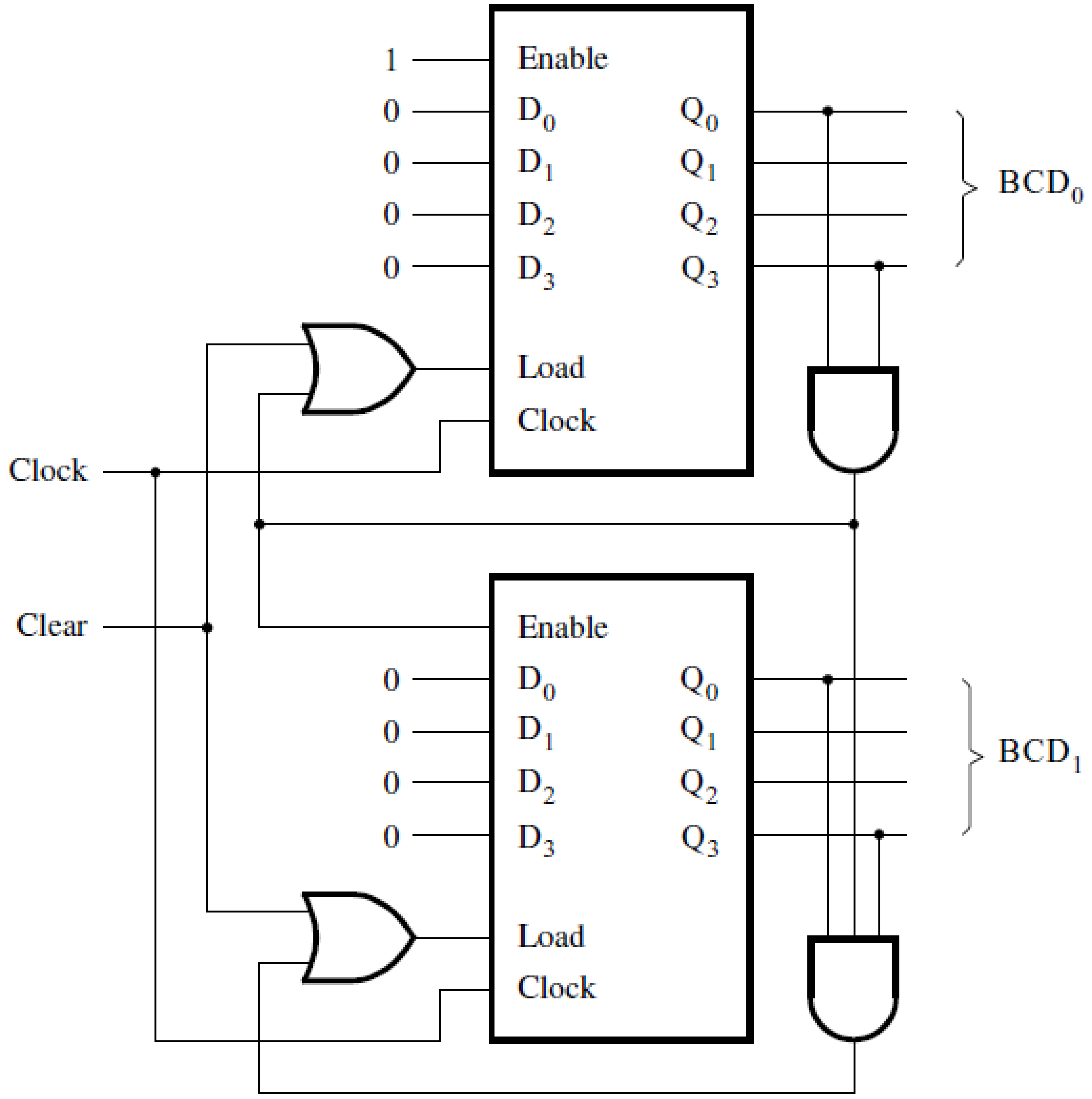
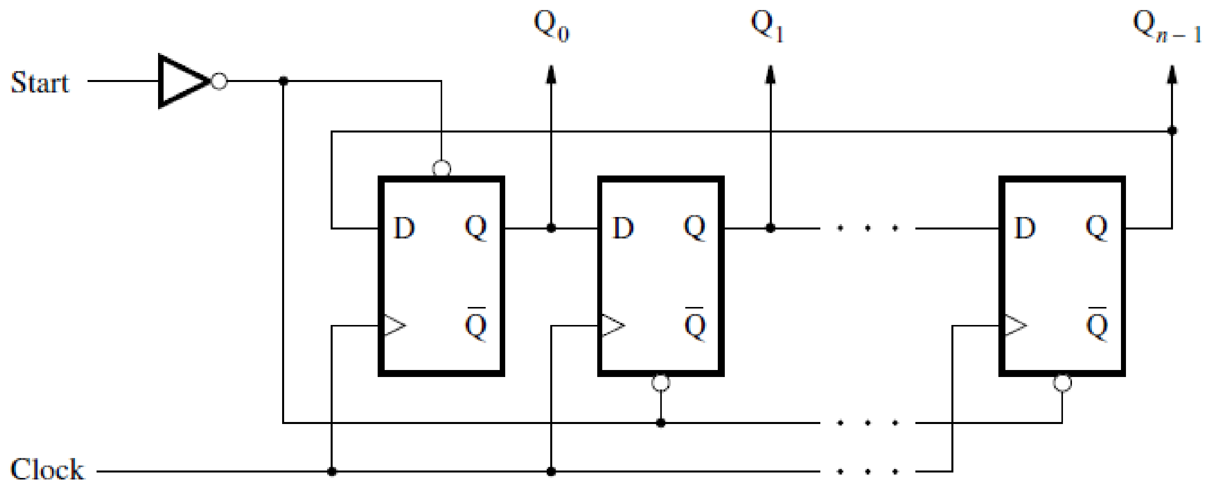
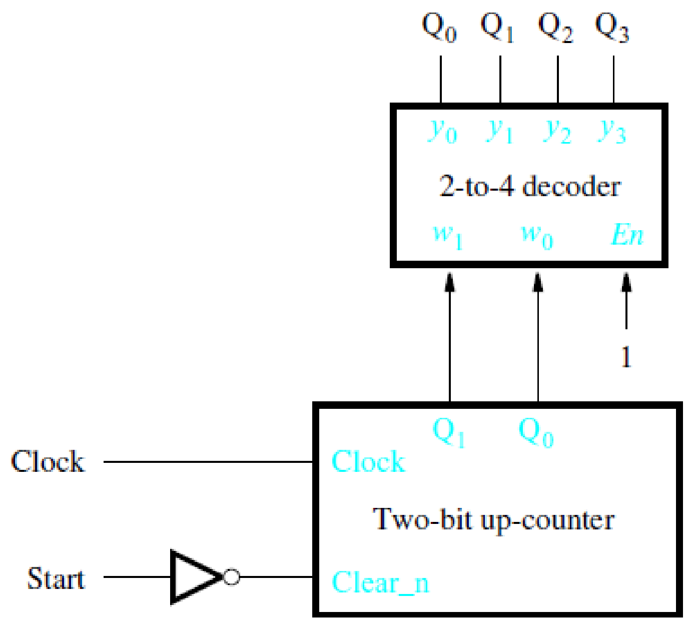


Figure 5.27. A two-digit BCD counter.



(a) An *n*-bit ring counter

Figure 5.28. Ring counter.



Produces a sequence of length 2n.

0000 → 1000 → 1100 → 1110 → 1111 → 0111 → etc.

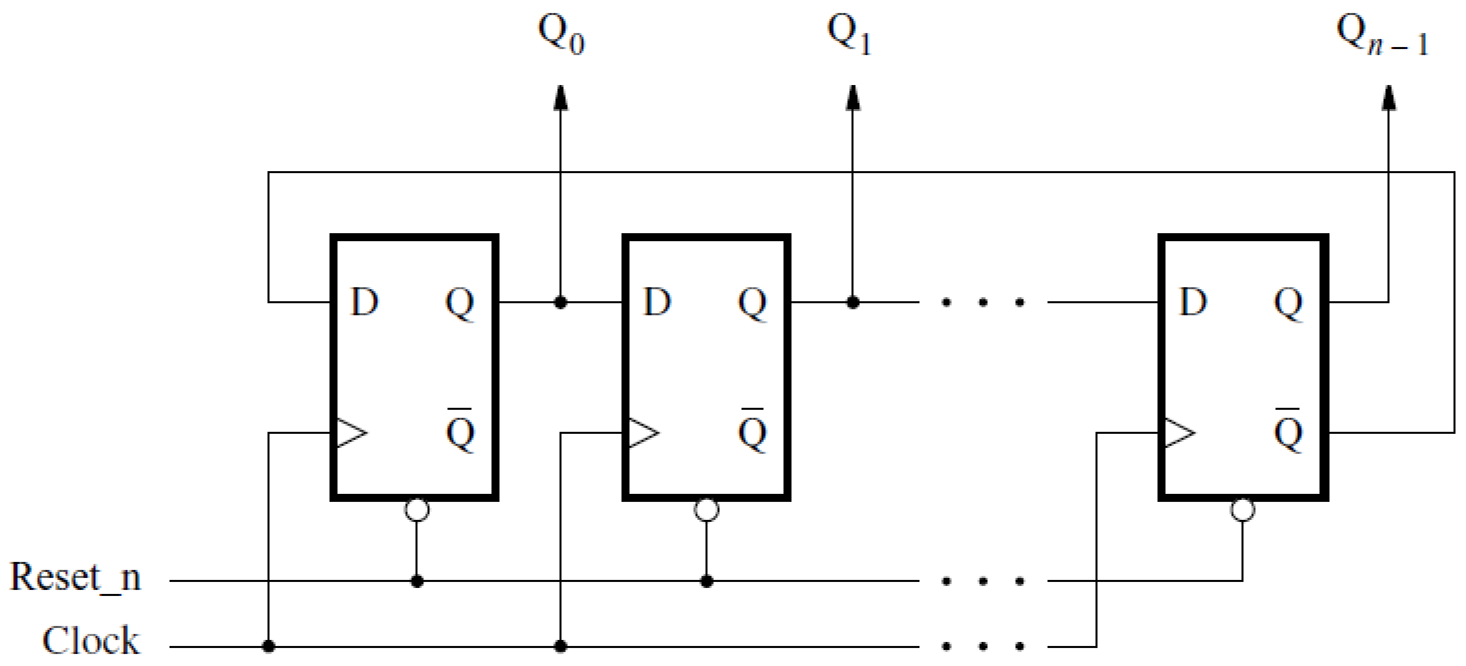
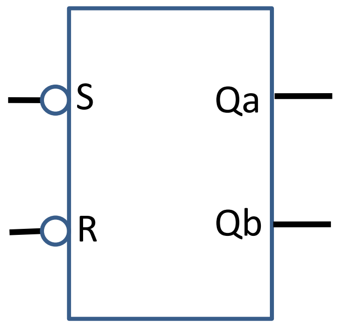
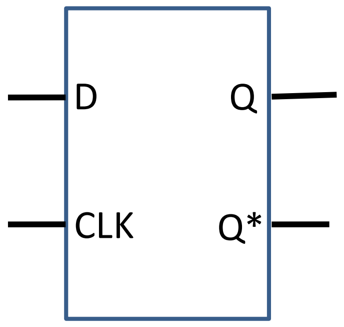


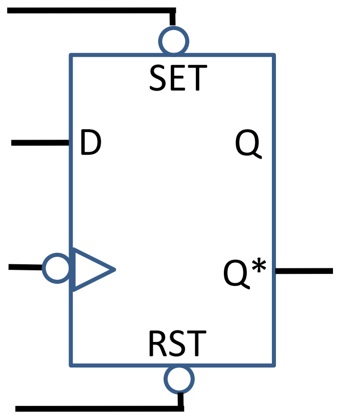
Figure 5.29. Johnson counter.



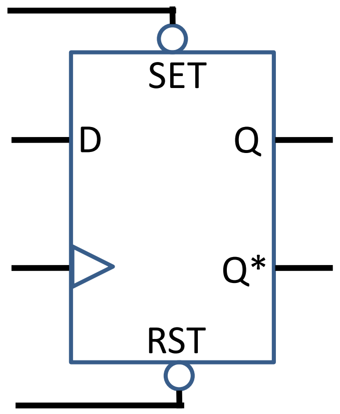
Latch



Gated Latch

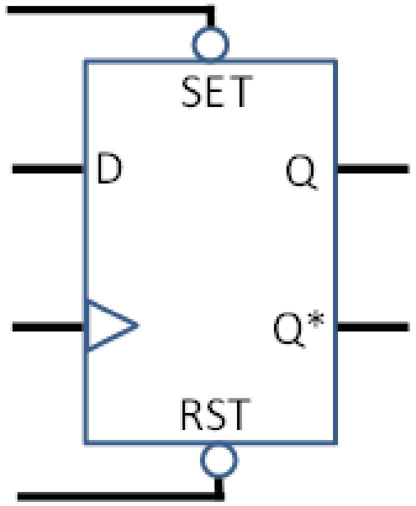


Master-slave



Edge-triggered

```
module Dflipflop( input clock,
      reset, D, output reg Q );

   always @( posedge clock )
      Q <= reset ? 0 : D;

endmodule
```



Edge-triggered

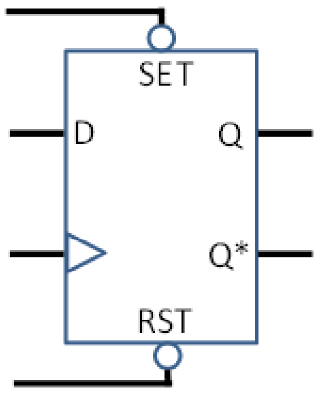
Code for a D flip-flop using a <= non-blocking assignment.

```
module Dflipflop( input clock,
      reset, D, output reg Q );

   always @( posedge clock )
      Q <= reset ? 0 : D;

endmodule
```

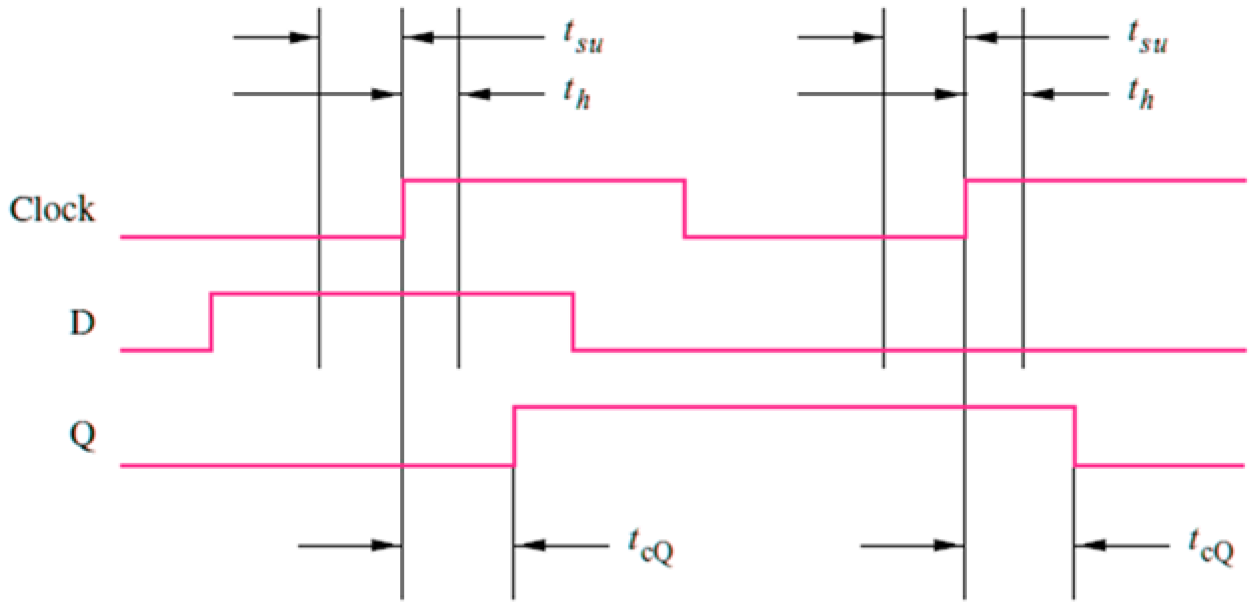
The RHS of the <= operator is evaluated just before the clock edge and the assignment is made just after the clock edge.

Code for a D flip-flop using a <= non-blocking assignment.

```
module Dflipflop( input clock,
      reset, D, output reg Q );

   always @( posedge clock )
      Q <= reset ? 0 : D;

endmodule
```



Edge-triggered

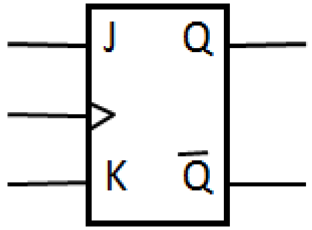


(b) Timing diagram

```
module JK( input clock, J, K, reset,
      output reg Q );

   always @( posedge reset,
         posedge clock )
      casex ( { reset, J, K } )
         'b1xx: Q <= 0;
         'b000: Q <= Q;
         'b001: Q <= 0;
         'b010: Q <= 1;
         'b011: Q <= ~Q;
      endcase

endmodule
```

J	K	Q (t+1)
0	0	Q (t)
0	1	0
1	0	1
1	1	$\bar{Q}$ (t)

(b) Truth table



(c) Graphical symbol

```
module IncorrectShiftRegister(
      input clock, D,
      output reg Q1, Q2 );

   // Incorrect code.
   // Both Q1 and Q2 always get
   // the same value.

   always @( posedge clock )
      begin
      Q1 = D;
      Q2 = Q1;
      end

endmodule
```

Should use non-blocking assignments with clocked always!

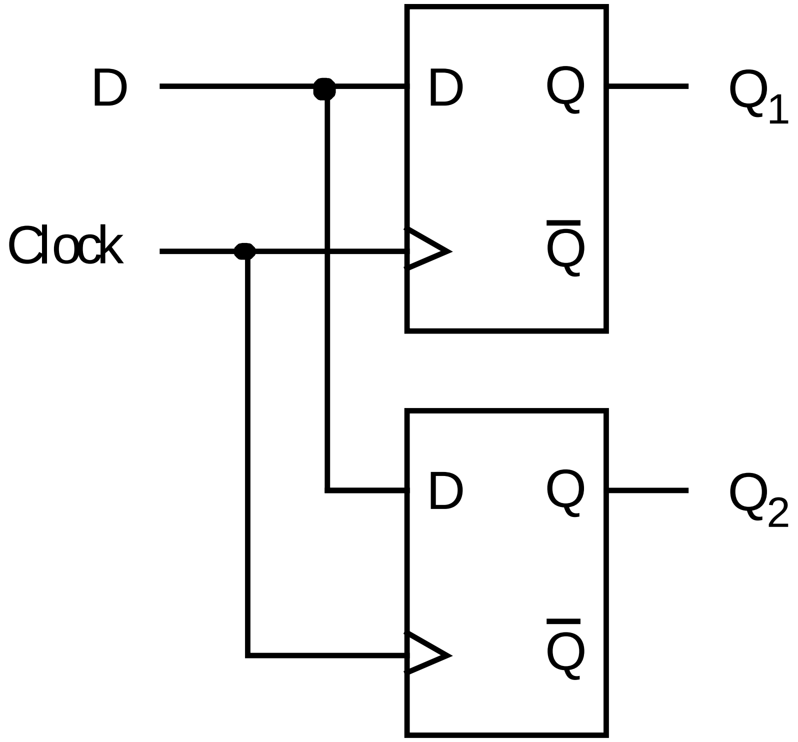


Figure 5.36. **Incorrect** code for two cascaded flip-flops.

```
module CorrectShiftRegister( input clock, D,
     output reg Q1, Q2 );

   // Correct code.
   // tCQ after the clock edge, Q1 and Q2
   // take the values D and Q1 had tsetup
   // before the clock.

   always @( posedge clock )
      begin
      Q1 <= D;
      Q2 <= Q1;
      end

endmodule
```

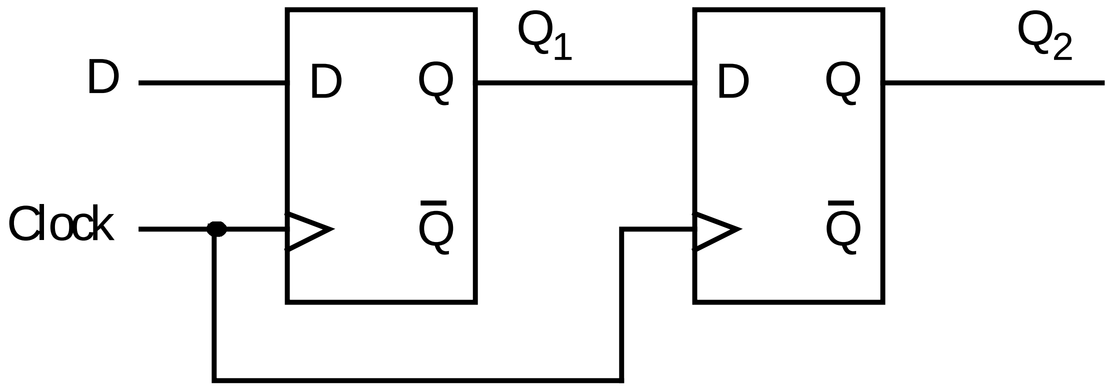


Figure 5.38. **Correct** code for two cascaded flip-flops.

```
module BadCode( input clock, x1, x2, x3,
      output reg f, g );

   always @( posedge clock )
      begin
      f = x1 & x2;
      g = f | x3;
      end

endmodule
```

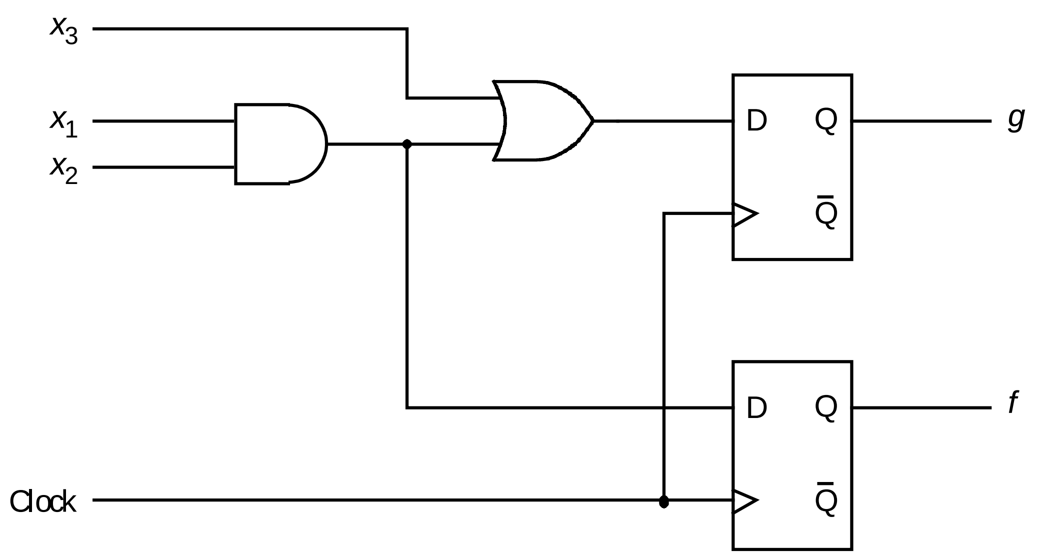


Figure 5.40. Another example of ***incorrect*** code

```
module CorrectCode( input clock, x1, x2, x3,
      output reg f, g );

   always @( posedge clock )
      begin
      f <= x1 & x2;
      g <= f | x3;
      end

endmodule
```

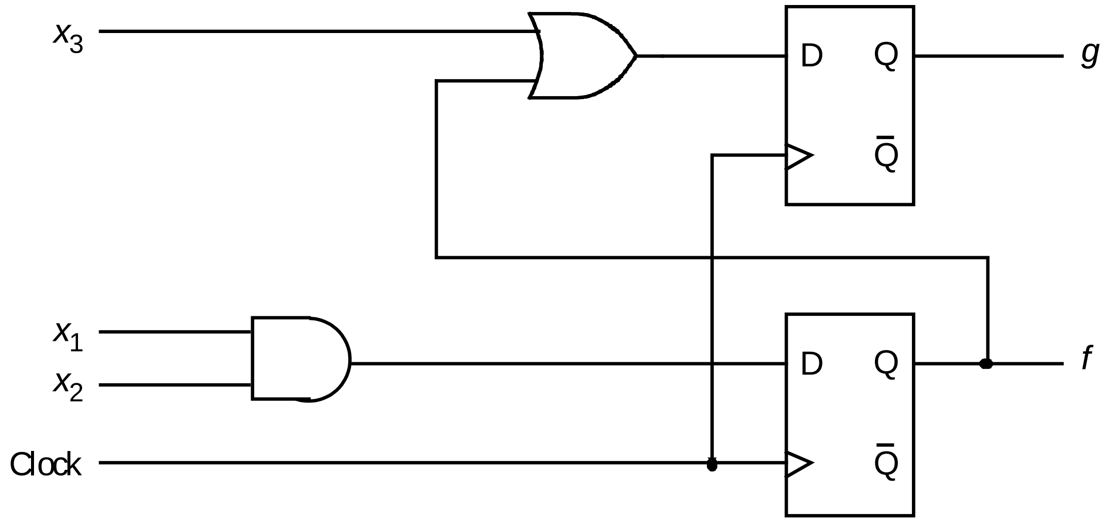


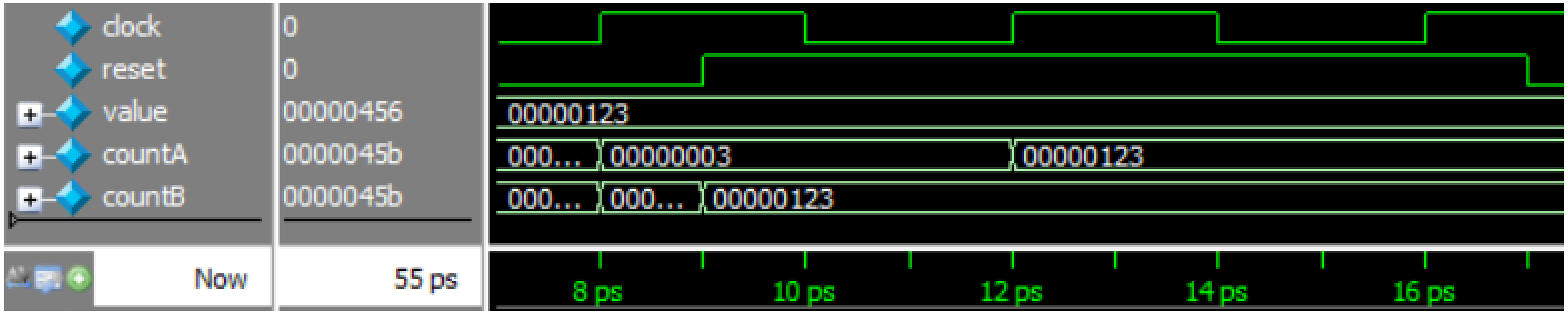
Figure 5.42. **Correct** code for Example 5.6.

```
module SynchReset( input clock, reset, D, output Q );

   // Synchronous reset (synchronized to the clock)
   always @( posedge clock )
      Q <= reset ? 0 : D;

endmodule


module AsyncReset( input clock, reset, D, output Q );

   // Asynchronous reset (not synchronized to the clock)
   always @( posedge reset, posedge clock )
      Q <= reset ? 0 : D;

endmodule
```

D flip-flops with synchronous and asynchronous resets.

```
module CounterA(
      input clock, reset,
      input [ 31:0 ] resetValue,
      output reg [ 31:0 ] count = 0 );

   // Synchronous reset (synchronized to the clock)
   always @( posedge clock )
      count <= reset ? resetValue : count + 1;

endmodule


module CounterB(
      input clock, reset,
      input [ 31:0 ] resetValue,
      output reg [ 31:0 ] count = 0 );

   // Asynchronous reset (not synchronized to the clock)
   always @( posedge reset, posedge clock )
      count <= reset ? resetValue : count + 1;

endmodule
```

From simulation, reset in *synchronous* in CounterA, changing only *with* the clock, and *asynchronous* in CounterB.



CounterA

```
always @( posedge clock )
   count <= reset ?
      resetValue : count + 1;
```

CounterB

```
always @( posedge reset, posedge clock )
   count <= reset ?
      resetValue : count + 1;
```

```
module MuxDFF1( input clock, s, D0, D1,
      output reg Q );

   always @( posedge clock )
      if ( s )
         Q <= D1;
      else
         Q <= D0;

endmodule
```

Figure 5.47. Code for a D flip-flop with a 2-to-1 multiplexer on the D input.

```
module MuxDFF2( input clock, s, D0, D1,
      output reg Q );

   always @( posedge clock )
      Q <= s ? D1 : D0;

endmodule
```

Figure 5.48. Alternative code for a D flip-flop with a 2-to-1 multiplexer on the D input.

```
module MuxDFF3( input clock, s, D0, D1,
      output reg Q );

   wire D;
   assign D = s ? D1 : D0;

   always @( posedge clock )
      Q <= D;

endmodule
```

Yet another alternative code for a D flip-flop with a 2-to-1 multiplexer on the D input.

```
module FourBitRightShift1( input clock, reset, D,
      input [ 0:3 ] resetValue, output [ 0:3 ] Q );

   MuxDFF1 ff0( clock, s, D,      resetValue[ 0 ], Q[ 0 ] );
   MuxDFF1 ff1( clock, s, Q[ 0 ], resetValue[ 1 ], Q[ 1 ] );
   MuxDFF1 ff2( clock, s, Q[ 1 ], resetValue[ 2 ], Q[ 2 ] );
   MuxDFF1 ff3( clock, s, Q[ 2 ], resetValue[ 2 ], Q[ 3 ] );

endmodule
```

Figure 5.49. Hierarchical code for a four-bit shift register.

```
module FourBitRightShift2( input clock, reset, D,
      input [ 0:3 ] resetValue, output reg [ 0:3 ] Q );

   always @( posedge clock )
      if ( reset )
         Q <= resetValue;
      else
         begin
         Q[ 0 ] <= D;
         Q[ 1 ] <= Q[ 0 ];
         Q[ 2 ] <= Q[ 1 ];
         Q[ 3 ] <= Q[ 2 ];
         end

endmodule
```

Another four-bit shift register.

```
module FourBitShiftReg3( input clock, reset, D,
      input [ 0:3 ] resetValue, output reg [ 0:3 ] Q );

   always @( posedge clock )
      if ( reset )
         Q <= resetValue;
      else
         Q <= { D, Q[ 0:2 ] };

endmodule
```

Another four-bit shift register.

```
module NBitRightShift1 #( parameter n = 4 )
      ( input clock, reset, D,
      output reg [0 : n - 1] Q );

   always @( posedge clock )
      if ( reset )
         Q <= 0;
      else
         begin
         integer i;
         for ( i = n - 1; i != 0; i = i - 1 )
            Q[ i ] <= Q[ i - 1 ];
         Q[ 0 ] <= D;
         end

endmodule
```

An n-bit right shift register.

```
module NBitRightShift2 #( parameter n = 4 )
      ( input clock, reset, D,
      output reg [ 0 : n - 1 ] Q );

   always @( posedge clock )
      if ( reset )
         Q <= 0;
      else
         Q <= { D, Q[ 0 : n - 2 ] };
endmodule
```

An n-bit right shift register.

```
module UpCount( input clock, reset, load, enable,
      input [ 3:0 ] loadValue, output reg [ 3:0 ] Q );

   always @( posedge reset, posedge clock )
      if ( reset )
         Q <= 0;
      else
         if ( load )
            Q <= loadValue;
         else
            if ( enable )
               Q <= Q + 1;

endmodule
```

```
module Scan( input CLOCK_50,
      output [ 9:0 ] LEDR );

   reg [ 31:0 ] counter;

   assign LEDR = counter[ 31:22 ];

   always @( posedge CLOCK_50 )
      counter <= counter + 1;
endmodule
```

```
module Scan( input CLOCK_50,
      output [ 9:0 ] LEDR );

   reg  [ 31:0 ] counter;
   reg  [  3:0 ] onehot;
   wire [  1:0 ] columnNumber;

   assign columnNumber = counter[ 26:25 ];
   assign LEDR = { onehot, columnNumber };

   always @( posedge CLOCK_50 )
      counter <= counter + 1;

   always @( * )
      case ( columnNumber )
         0: onehot = 'b1000;
         1: onehot = 'b0100;
         2: onehot = 'b0010;
         3: onehot = 'b0001;
      endcase
endmodule
```

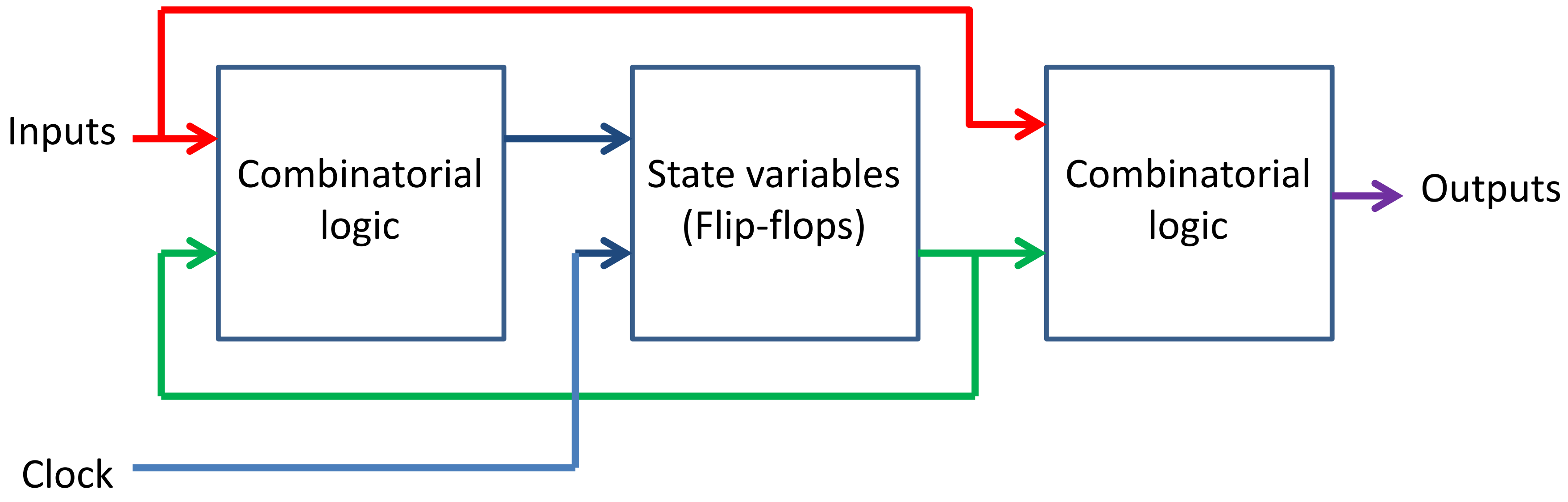
```verilog
module Scan( input CLOCK_50,
      output [ 9:0 ] LEDR );

   reg [ 31:0 ] counter;
   reg [  3:0 ] onehot;
   reg [  1:0 ] columnNumber;

   parameter desiredFrequency = 1.0/3.0,
      divisor = 50_000_000 / desiredFrequency;

   assign LEDR = { onehot, columnNumber };

   always @( posedge CLOCK_50 )
      if ( counter == 0 )
         begin
         counter <= divisor;
         columnNumber <= columnNumber + 1;
         case ( columnNumber )
            0: onehot <= 4'b1000;
            1: onehot <= 4'b0100;
            2: onehot <= 4'b0010;
            3: onehot <= 4'b0001;
         endcase
         end
      else
         counter <= counter - 1;

endmodule
```

Finite state machines

Machines whose next state depends on the inputs and the previous state.

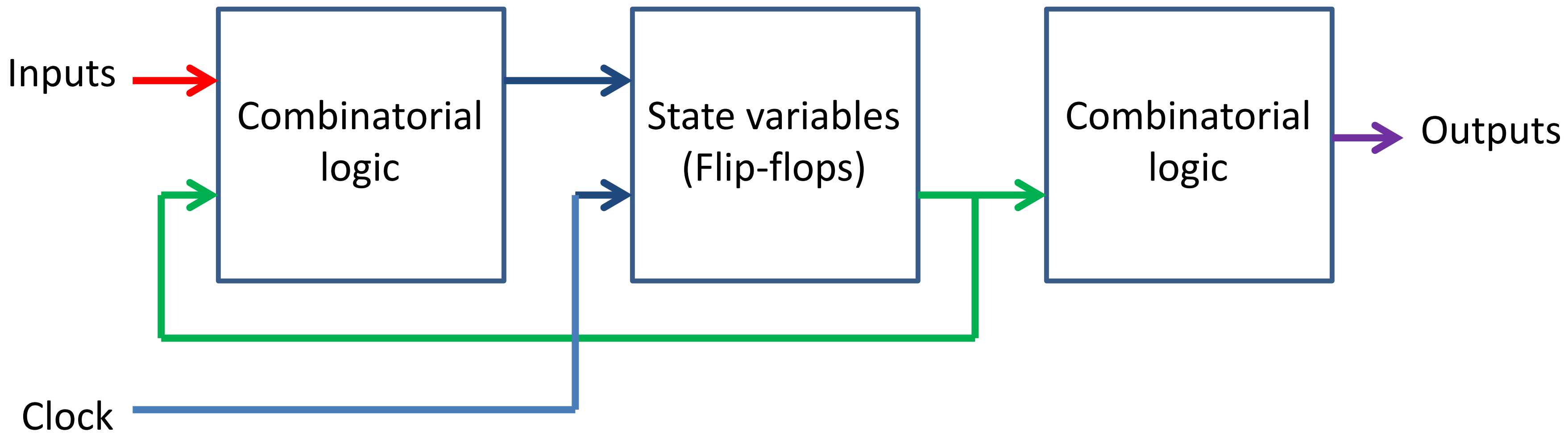
Called finite state machines because they have only a ***finite*** number of states.

Generalized form of a sequential circuit

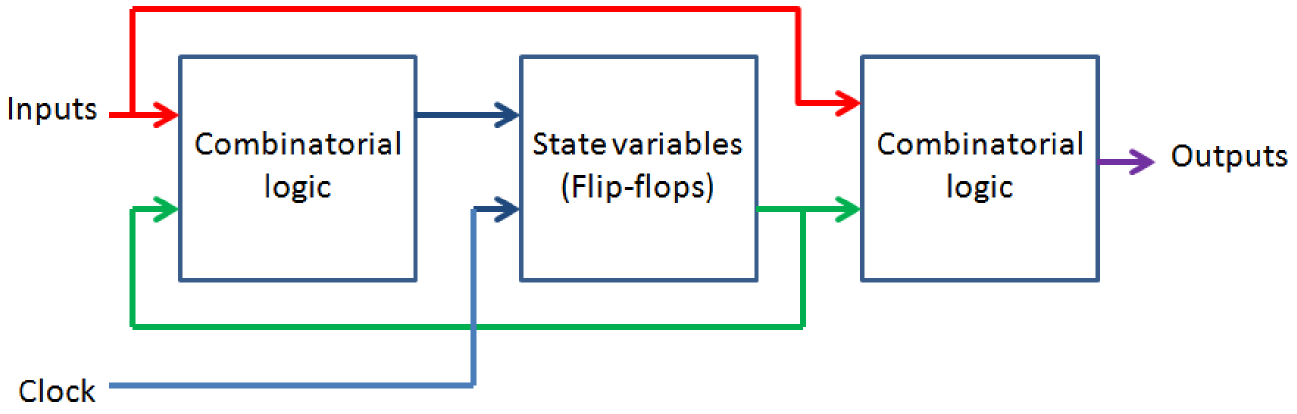


If the outputs depend on both the current state and the current inputs, it is called a ***Mealy*** machine, named after George Mealy, who invented the concept in 1955.

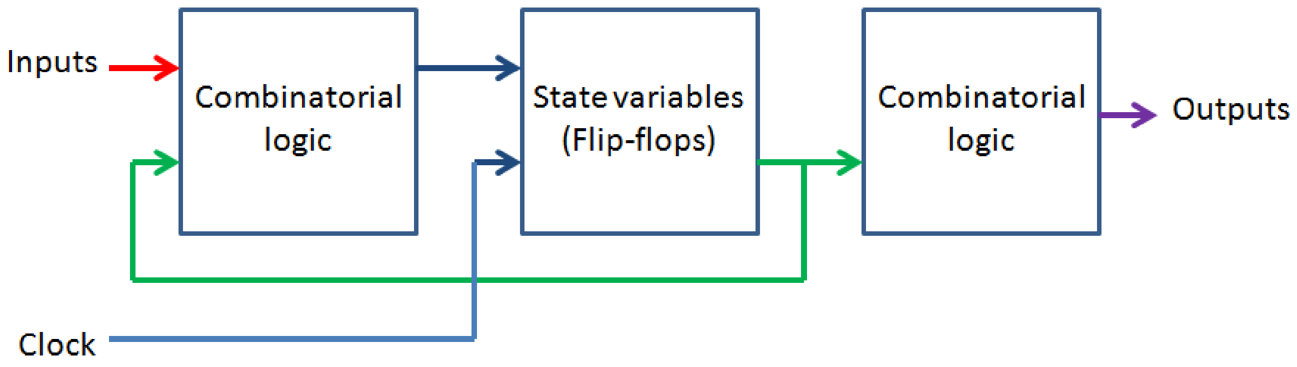
Moore machine



If the outputs depend only on the current state, it is called a ***Moore*** machine. *("Moore is less.")* It's named after Edward Moore, who invented the concept in 1956.

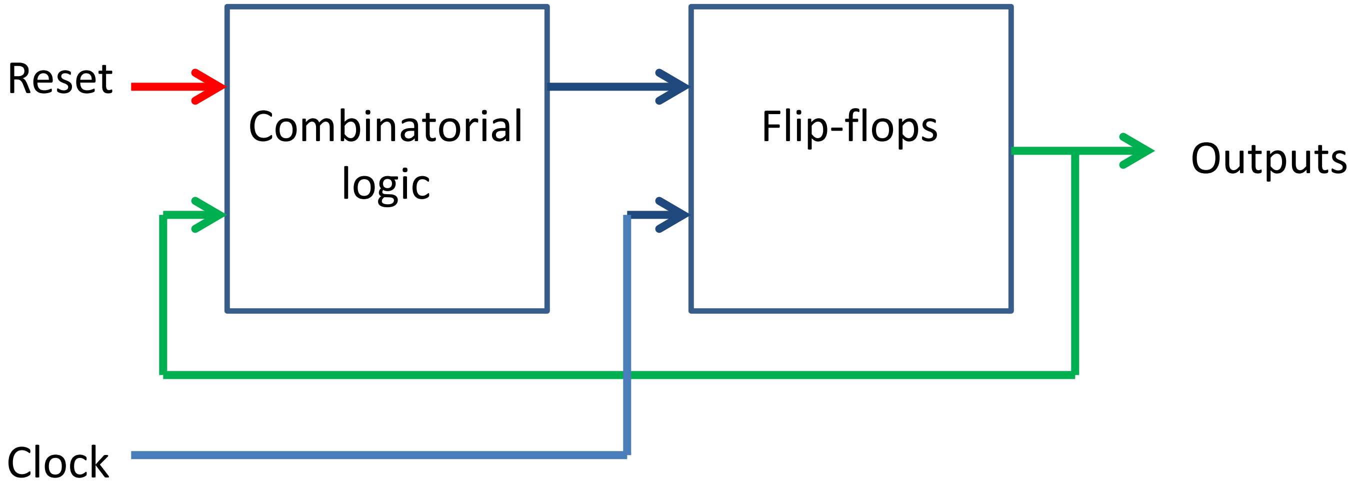


Mealy machines require fewer states but if the inputs change asynchronously, the outputs can change asynchronously as well.



Moore machines require more state variables and the outputs are delayed by one clock. But all the outputs are guaranteed to be synchronous.

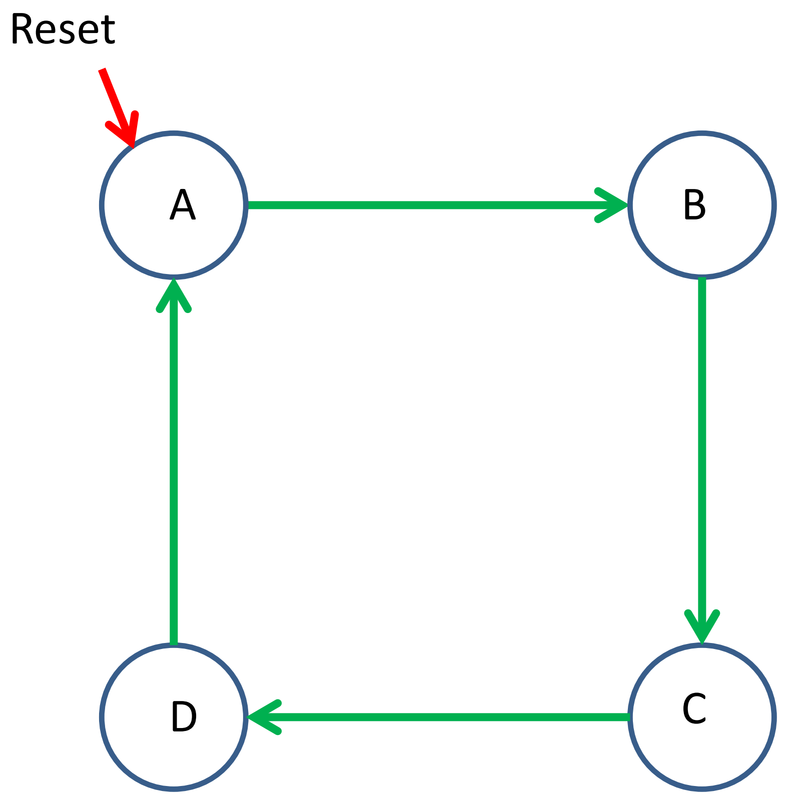
Simplest example: A counter

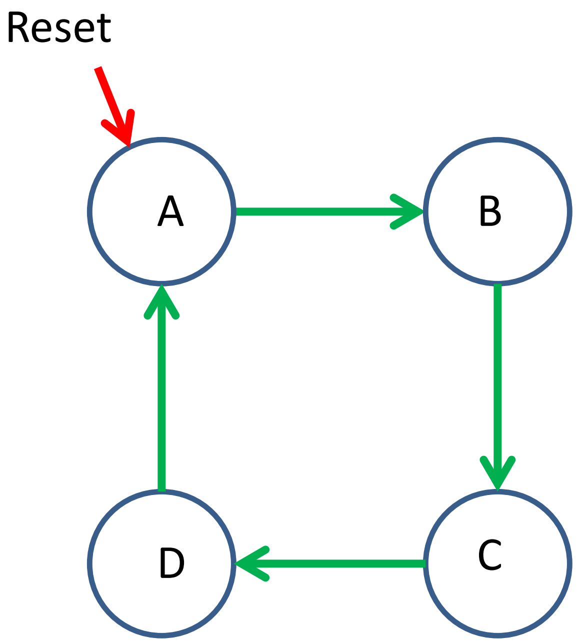


State diagram for a counter with 4 states.

At each clock, it always moves to the next state.

The arrows between states are called ***edges*** or ***transitions***.





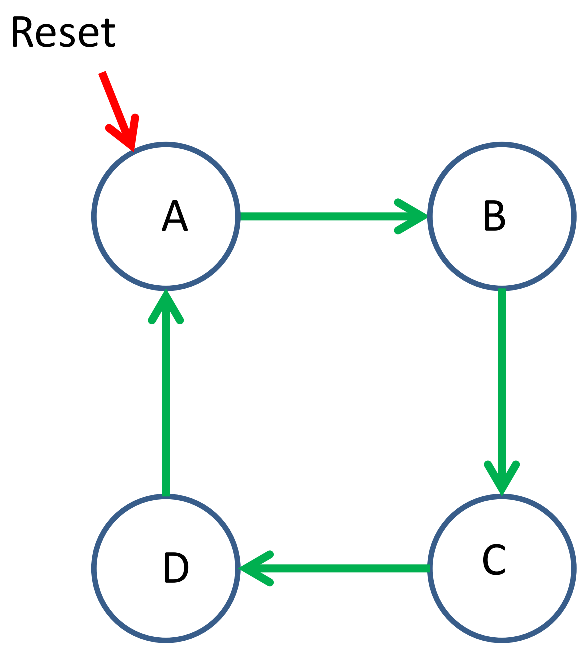
State table

Present state	Next state
A	B
B	C
C	D
D	A

State-assigned **table**

	Present state	Next state
	y2 y1	Y2 Y1
A	00	01
B	01	10
C	10	11
D	11	00

We pick any state assignments we like, though some could be better than others.



Having picked the assignments, we can use Karnaugh maps to derive the equations for the next state variables.

State-assigned **table**

	Present state y2 y1	Next state Y2 Y1
A	00	01
B	01	10
C	10	11
D	11	00

Y1		y1 0	1
y2	0	1	0
	1	1	0

Y1 = y1'

Y2		y1 0	1
y2	0	0	1
	1	1	0

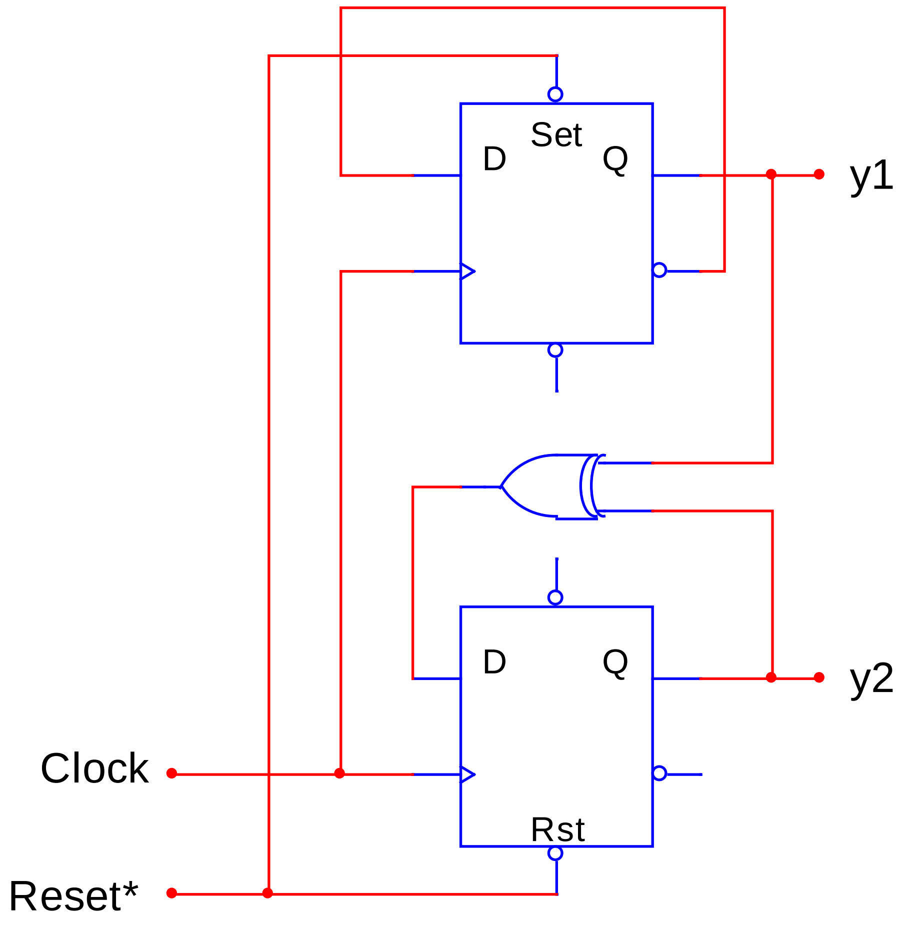
Y2 = y1 ^ y2

Y1		y1	
		0	1
y2	0	1	0
	1	1	0

Y1 = y1'

Y2		y1	
		0	1
y2	0	0	1
	1	1	0

Y2 = y1 ^ y2



Y1		y1	
		0	1
y2	0	1	0
	1	1	0

Y1 = y1'

Y2		y1	
		0	1
y2	0	0	1
	1	1	0

Y2 = y1 ^ y2

```
module Counter( input clock, reset,
          output reg y1, y2 );
      always @( posedge reset, posedge clock )
          if ( reset )
              begin
              y1 <= 0;
              y2 <= 0;
              end
          else
              begin
              y1 <= ~y1;
              y2 <= y1 ^ y2;
              end
endmodule
```

In Verilog, the <= “non-blocking” assignment means all the assignments happen synchronously at exit from the always block.

In Verilog, we would probably skip the Karnaugh maps write the code directly from the state-assigned table.

***State-assigned* table**

	Present state y2 y1	Next state Y2 Y1
A	00	01
B	01	10
C	10	11
D	11	00

```
module Counter2( input clock, reset,
          output reg [ 1:0 ] y );
     always @( posedge reset, posedge clock )
          y <= reset ? 0 : y + 1;
endmodule
```

If the cases were more complex or not in order, we might write it like this with a case statement.

State-assigned **table**

	Present state	Next state
	y2 y1	Y2 Y1
A	00	01
B	01	10
C	10	11
D	11	00

```
module Counter3( input clock, reset,
        output reg [ 1:0 ] y );
    always @( posedge reset, posedge clock )
        if ( reset )
            y <= 0;
        else
            case ( y )
                0:  y <= 1;
                1:  y <= 2;
                2:  y <= 3;
                3:  y <= 0;
            endcase
endmodule
```

Or we might parameterize the assignments.

***State-assigned* table**

	Present state y2 y1	Next state Y2 Y1
A	00	01
B	01	10
C	10	11
D	11	00

```
module Counter4( input clock, reset,
        output reg [ 1:0 ] y );
    parameter A = 0, B = 1, C = 2, D = 3;
    always @( posedge reset, posedge clock )
        if ( reset )
            y <= 0;
        else
            case ( y )
                A:  y <= B;
                B:  y <= C;
                C:  y <= D;
                D:  y <= A;
            endcase
endmodule
```

Verilog makes it really easy to pick any assignments you like and walk from one arbitrary state to another.

Example: Create a 3-bit counter in Verilog that cycles through this sequence: 4, 7, 0, 3, 2, 6, 1, 5.

Example: Create a 3-bit counter in Verilog that cycles through this sequence: 4, 7, 0, 3, 2, 6, 1, 5.

```
module PseudoRandom( input clock,
        output reg [ 2:0 ] Q );
    always @( posedge clock )
        case ( Q )
            4:  Q <= 7;
            7:  Q <= 0;
            0:  Q <= 3;
            3:  Q <= 2;
            2:  Q <= 6;
            6:  Q <= 1;
            1:  Q <= 5;
            5:  Q <= 4;
        endcase
endmodule
```