COMP3222/9222 Digital Circuits & Systems

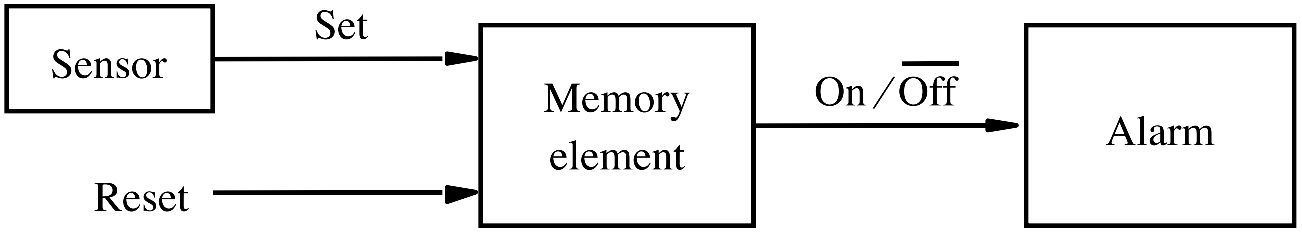
5. Flip-flops, Registers, Counters

Objectives

- Learn about logic circuits that store information
  - Flip-flops that store a single bit
  - Registers that store multiple bits
  - Shift registers that shift the contents of the register
  - Counters of various types
- VHDL constructs used to implement storage elements

Why we need circuits with memory

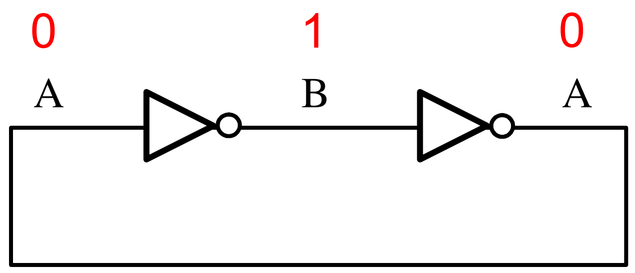
- Consider an alarm system that is required to remain activated when triggered, even when the cause for triggering has ceased



- Here, the *Reset* signal is intended to provide a means of switching off the alarm

How do we create a memory element?

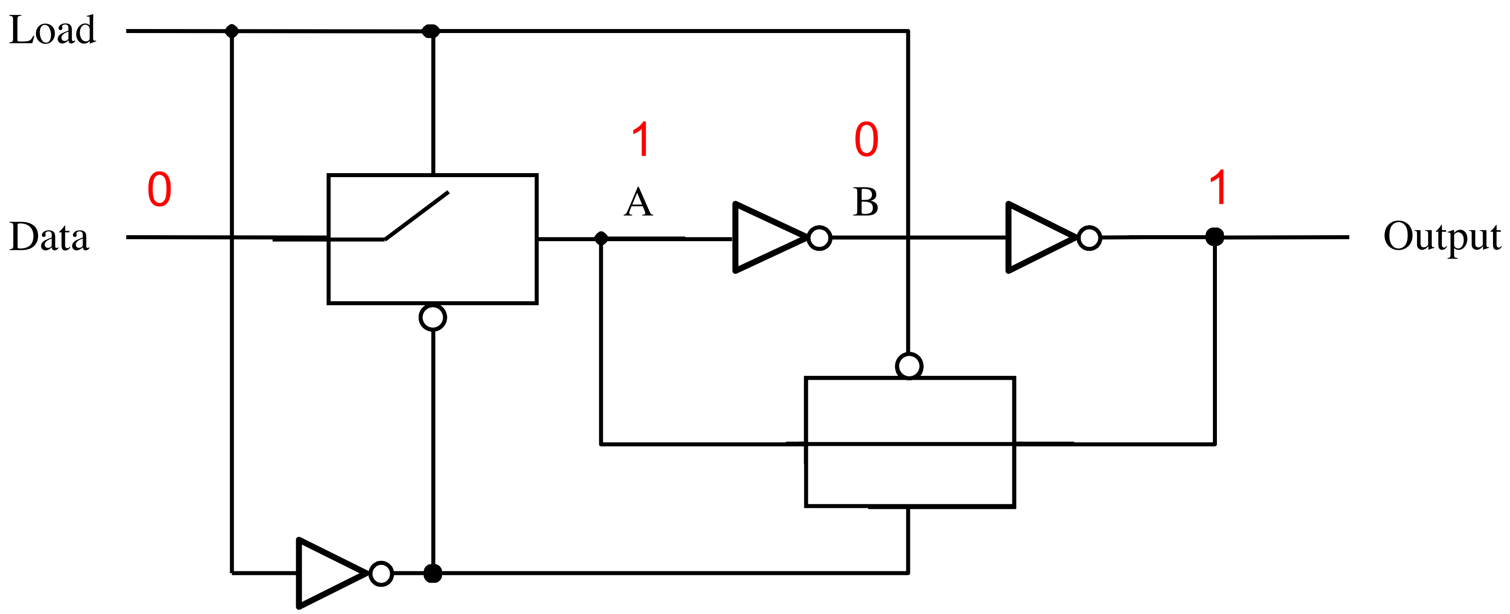
- Use feedback to “trap” a value
- Consider a simple cyclic circuit comprising two inverters

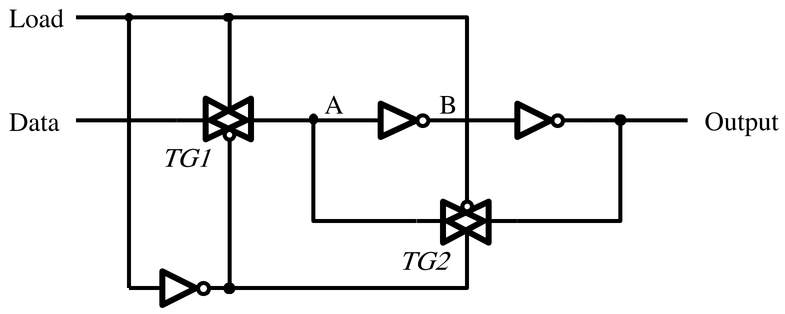


- The circuit has two stable states
- But there is no way of changing from one state to the other

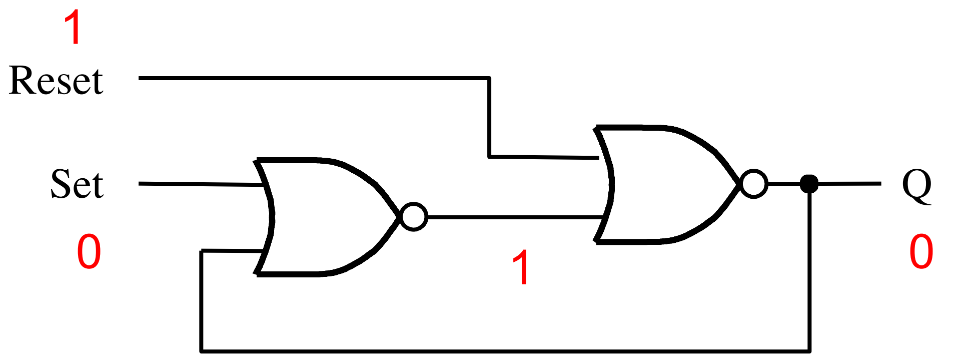
A controlled memory element

0 = preserve current state





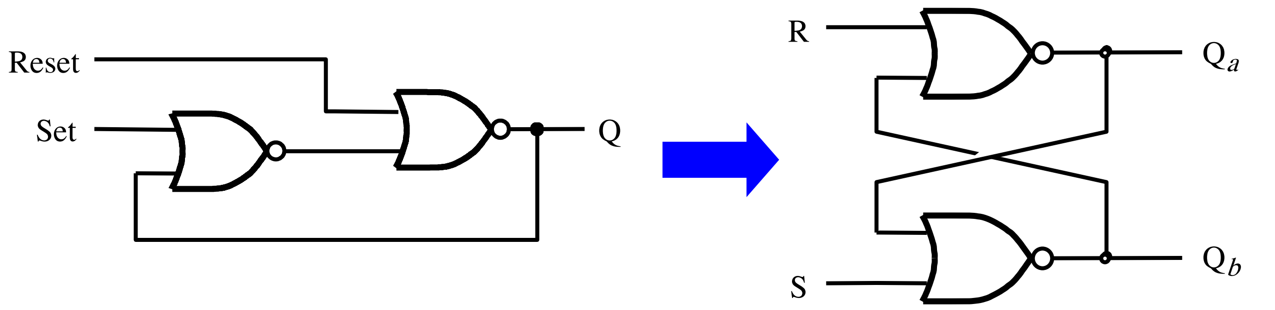
A memory element with NOR gates



A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

- When both *Set* and *Reset* are 0, the state, *Q*, is preserved
- *Set* = 1, *Reset* = 0 ⇒ *Q* = 1
- *Set* = D, *Reset* = 1 ⇒ *Q* = 0
- Known as a *latch*

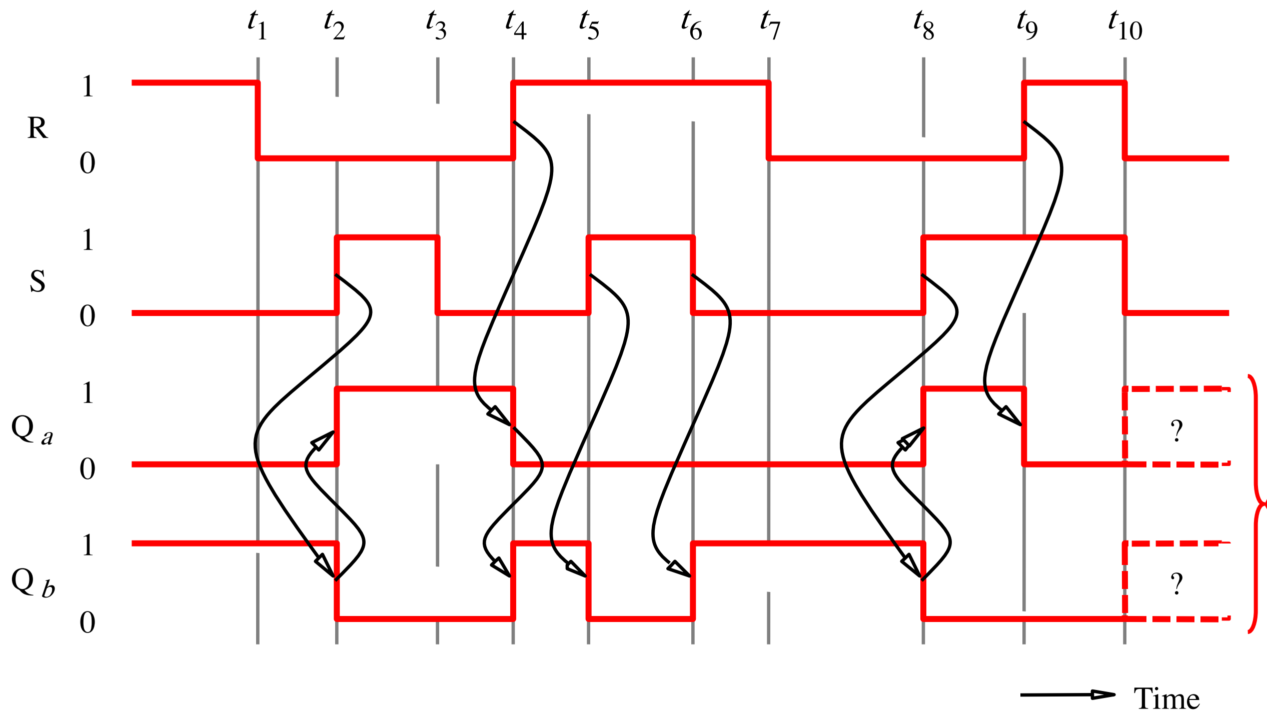
Basic latch using cross-coupled NOR gates



(a) Conventional circuit diagram

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) Characteristic table



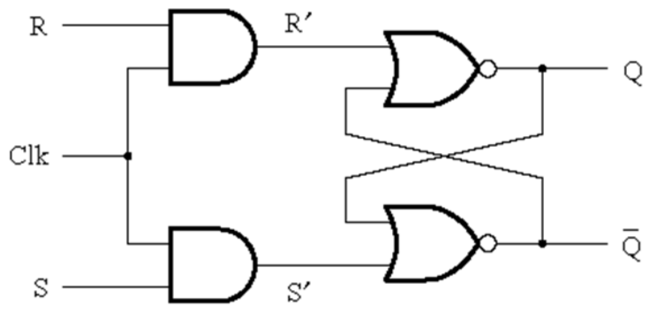
t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10}
R
S
Q_a
Q_b
1
0
?
?
Time

Oscillatory behaviour which settles to a final state that depends upon the relative speed of gates and wires

(c) Timing diagram

Gated SR latch

- A control input (*Clk*) acts to enable state changes

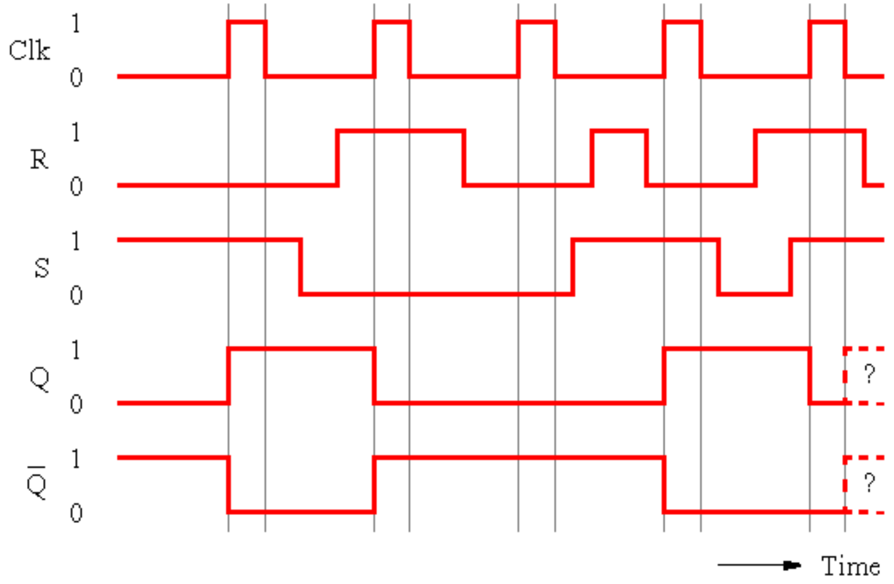


(a) Circuit

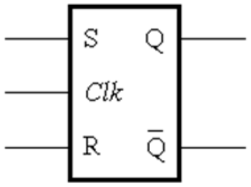
Clk	S	R	Q(t + 1)
0	x	x	Q(t) (no change)
1	0	0	Q(t) (no change)
1	0	1	0
1	1	0	1
1	1	1	x

(b) Characteristic table

Illegal input condition as outputs are supposed to be complements of each other



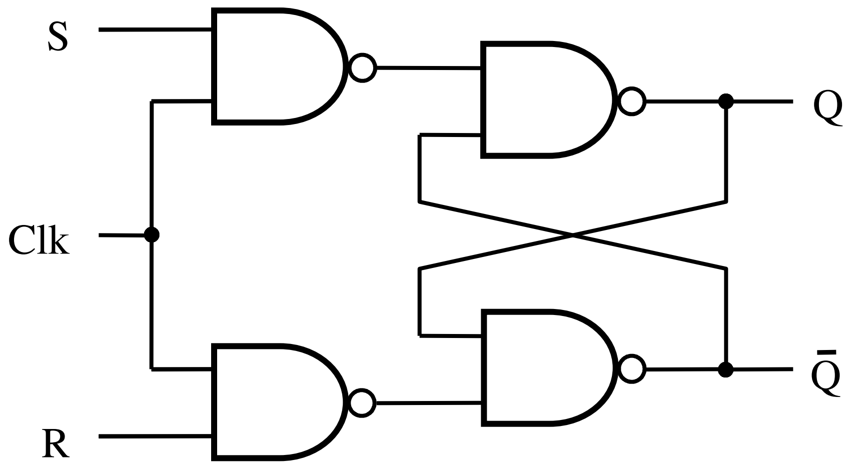
(c) Timing diagram



(d) Graphical symbol

Gated SR latch with NAND gates

- More usual configuration as it uses less transistors
  - Has exactly the same characteristic table
  - Note that S & R inputs are flipped about wrt the outputs



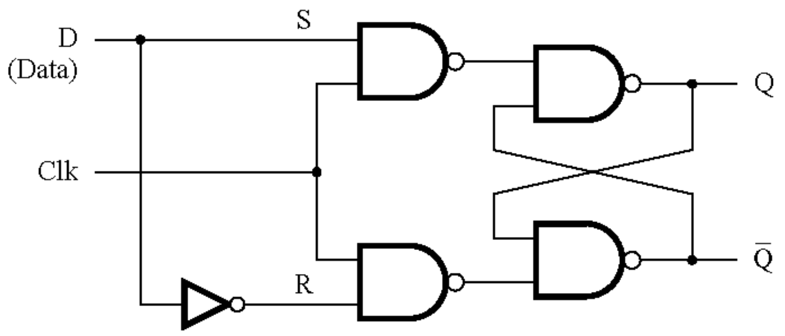
Clk	S	R	Q(t + 1)
0	x	x	Q(t) (no change)
1	0	0	Q(t) (no change)
1	0	1	0
1	1	0	1
1	1	1	x

(b) Characteristic table

A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

Gated D latch

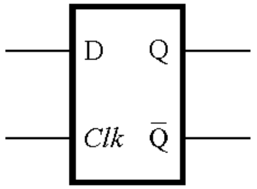
- Eliminates the illegal input combination *S = R = 1*
- Useful for storing a data bit



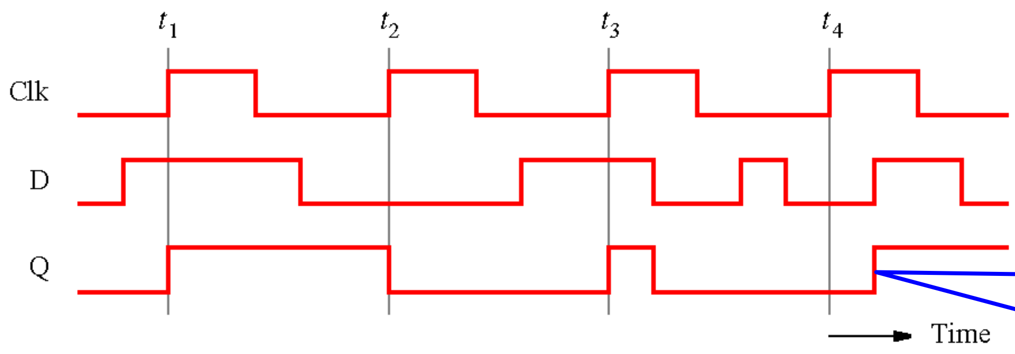
(a) Circuit

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

(b) Characteristic table



(c) Graphical symbol

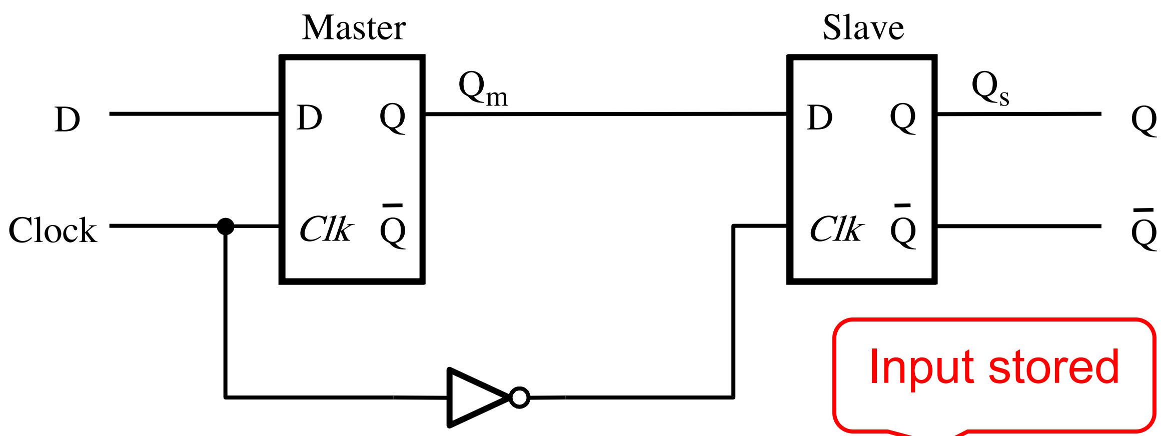


(d) Timing diagram

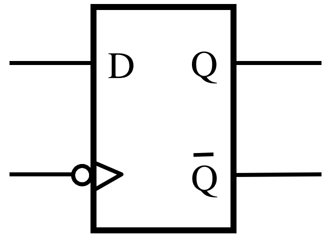
latches are said to be "transparent" to changes in the data input while the *Clk* input is high

Negative edge-triggered (Master-slave) D flip-flop

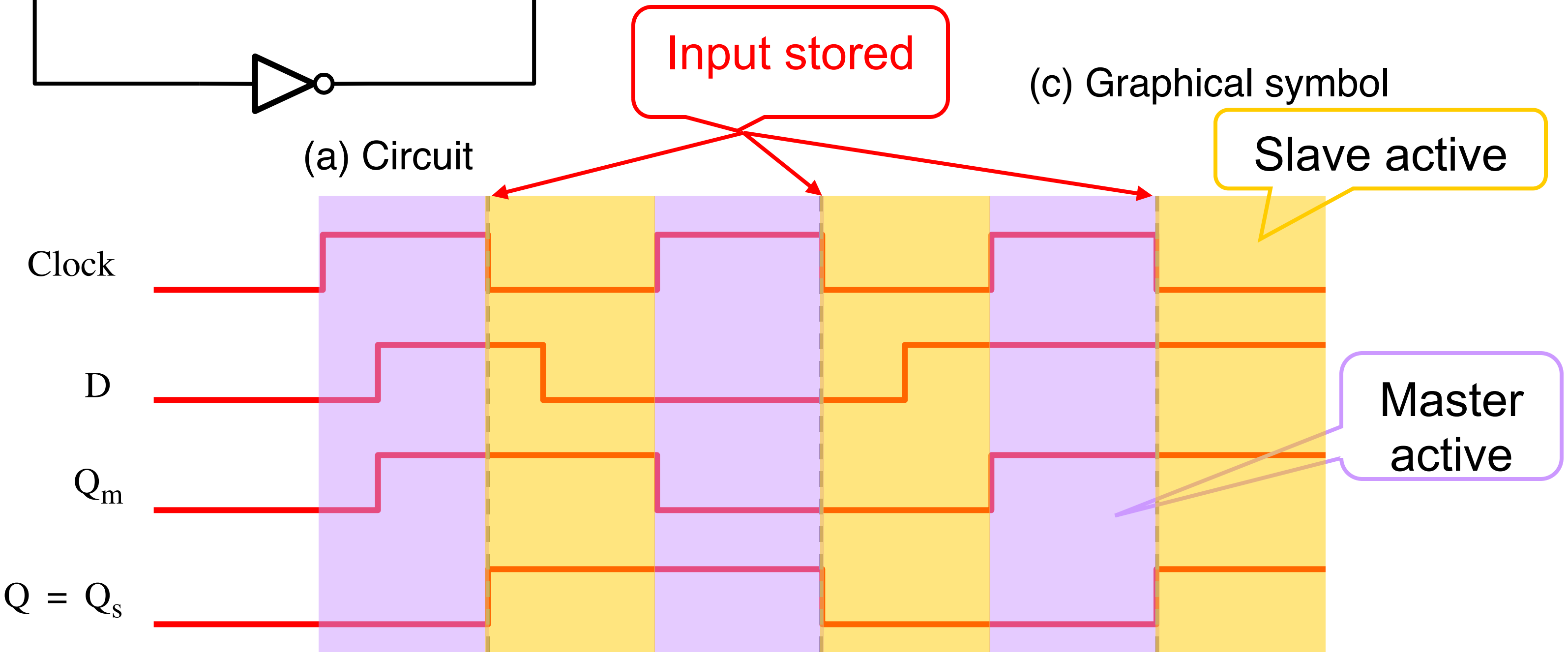
- Latches are triggered by the level of the control signal, flip-flops are triggered on control signal transitions



(a) Circuit



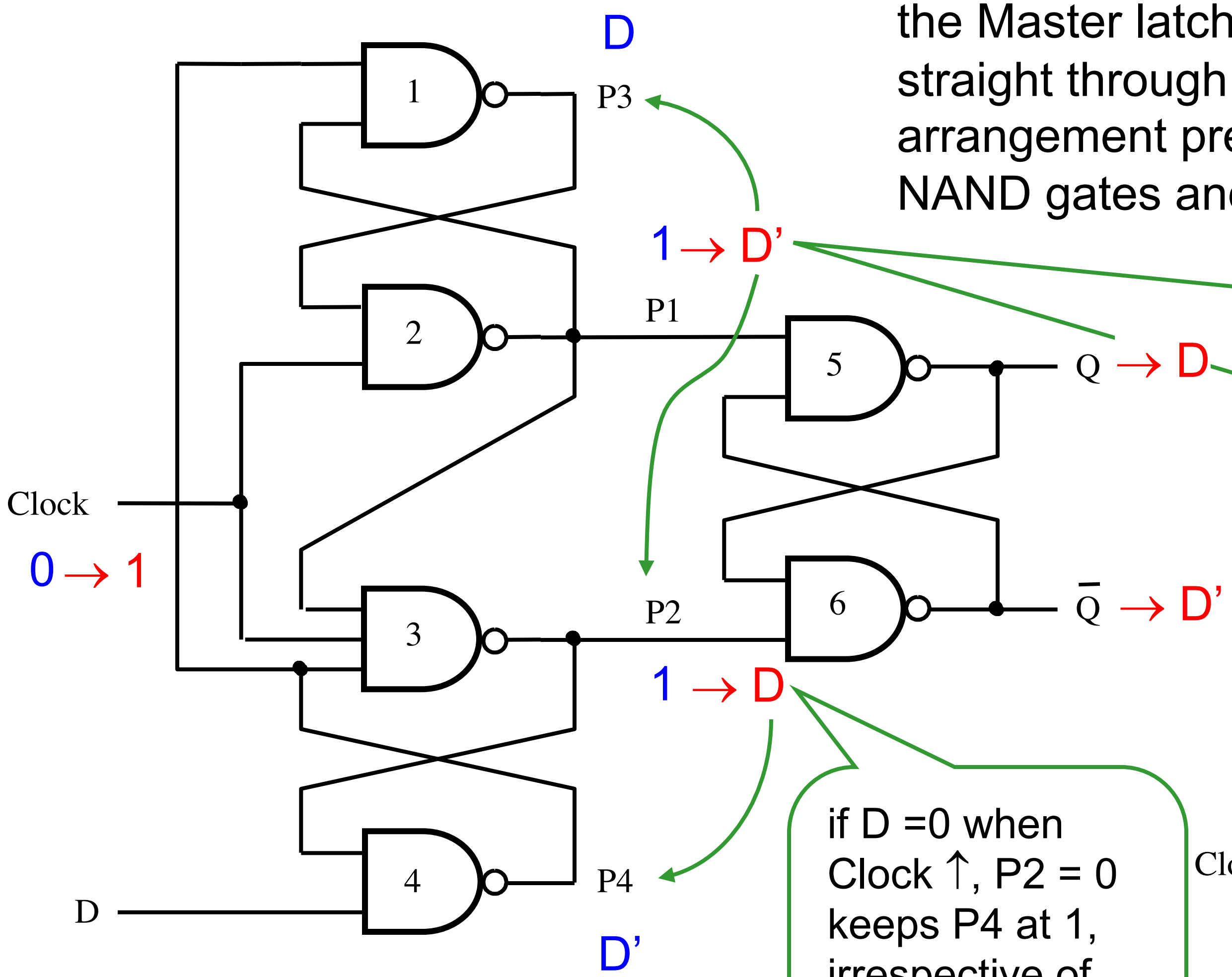
(c) Graphical symbol



(b) Timing diagram

A positive-edge-triggered D flip-flop

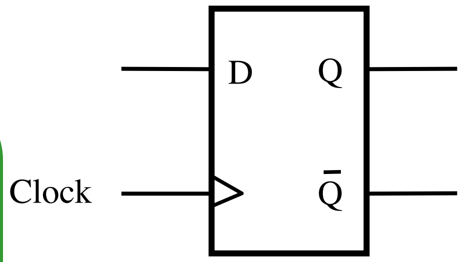
We could just invert the *Clk* input to the Master latch and feed the *Clk* signal straight through to the Slave, but the arrangement presented here saves two NAND gates and two inverters



(a) Circuit

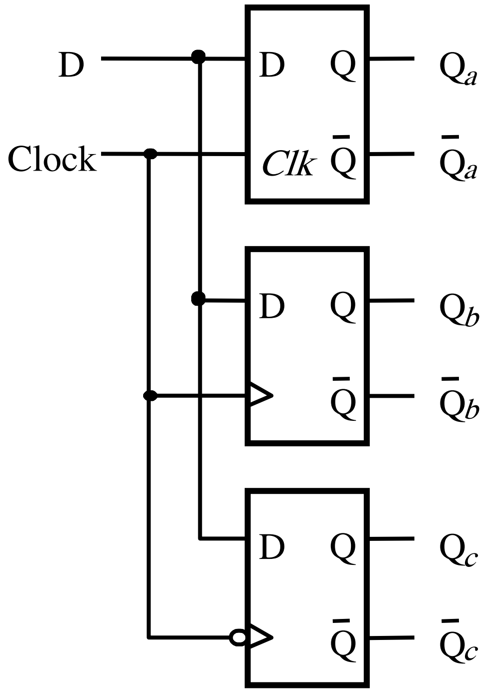
if D = 1 when Clock ↑, P1 = 0 keeps P2 & P3 at 1, irrespective of subsequent changes to D while Clock high

if D =0 when Clock ↑, P2 = 0 keeps P4 at 1, irrespective of subsequent changes to D while Clock high

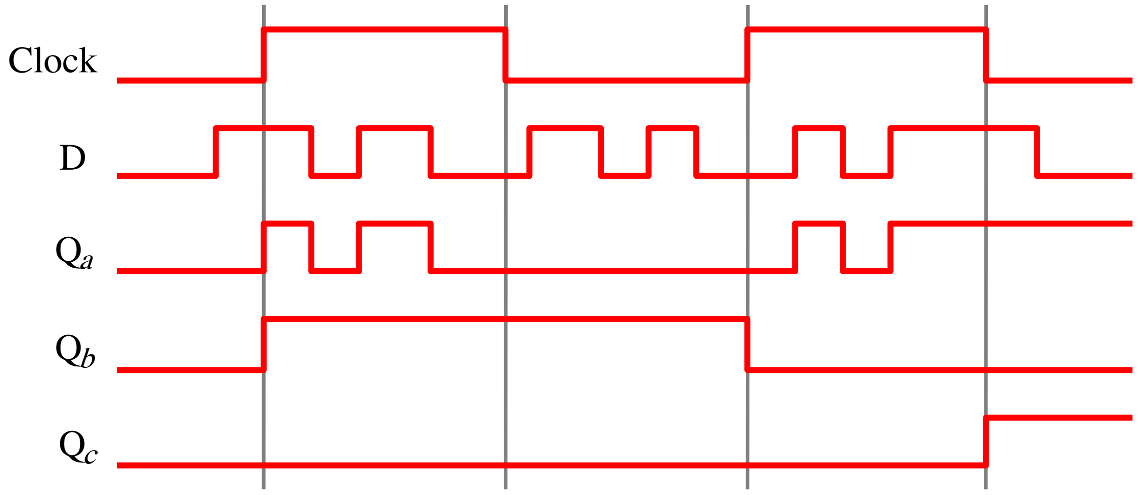


(b) Graphical symbol

Comparison of level-sensitive and edge-triggered D-type storage elements



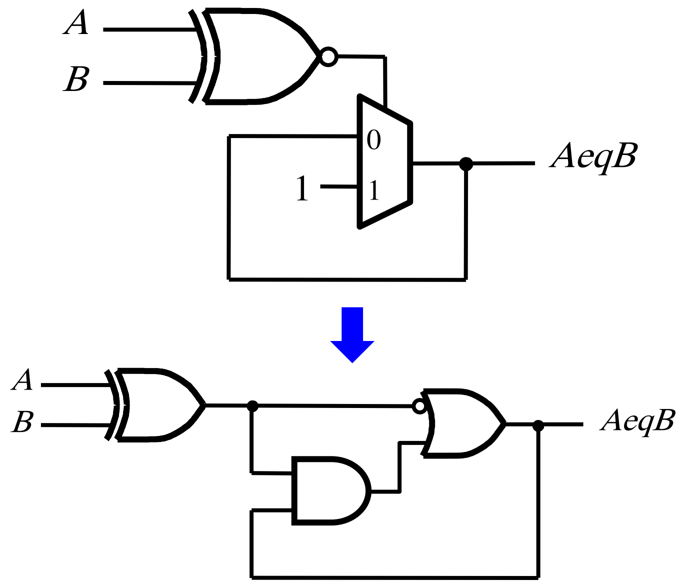
(a) Circuit



(b) Timing diagram

Recall the specification of implied memory

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY implied IS
      PORT ( A, B       : IN      STD_LOGIC ;
                  AeqB     : OUT  STD_LOGIC ) ;
END implied ;

ARCHITECTURE Behavior OF implied IS
BEGIN
      PROCESS ( A, B )
      BEGIN
            IF A = B THEN
                  AeqB <= '1' ;
            END IF ;
      END PROCESS ;
END Behavior ;
```

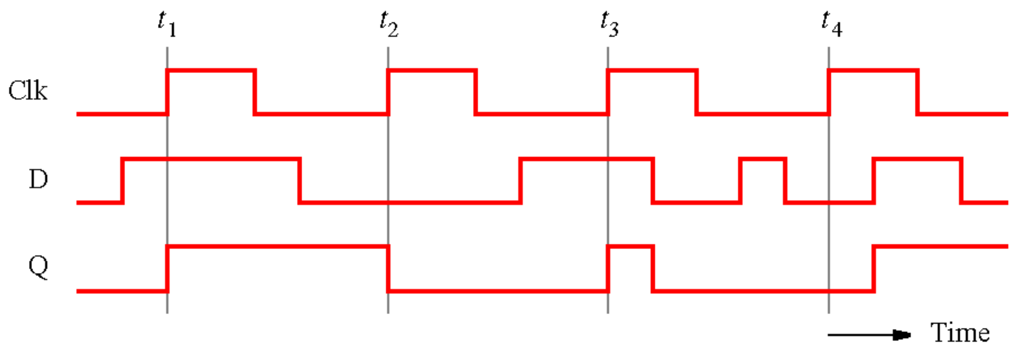


Resulting circuit has to remember the value of AeqB when A /= B

Code for a gated D latch

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY latch IS
    PORT (  D, CLK : IN     STD_LOGIC ;
            Q      : OUT    STD_LOGIC) ;
END latch ;

ARCHITECTURE Behavior OF latch IS
BEGIN
    PROCESS ( D, CLK )
    BEGIN
        IF CLK = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Note: the PROCESS describing *a latch*, while exploiting *implicit memory*, complies with the *COMBINATIONAL* design rule that all signals that can affect the output are listed in the sensitivity list; *otherwise* (if you only list the CLK signal) the compiler will think you are specifying a weird kind of flip-flop!

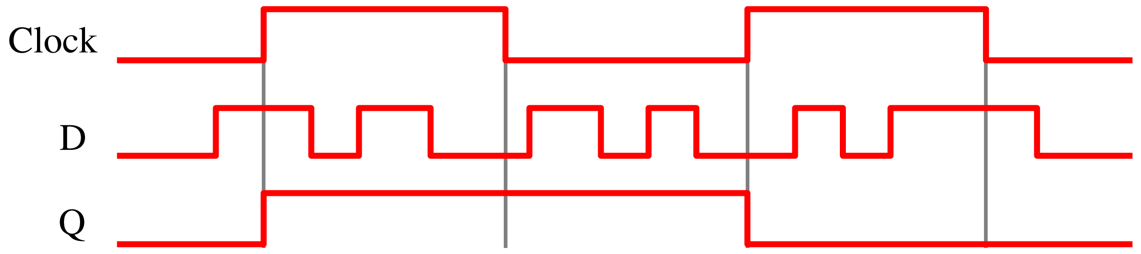


Code for a positive edge-triggered D flip-flop

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT (  D, CLK   : IN    STD_LOGIC ;
            Q        : OUT   STD_LOGIC) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS ( CLK )
    BEGIN
        IF CLK'event AND CLK = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

The boolean signal attribute *'event* is true when the signal transitions from 0→1 or 1→0

Notes: (i) *SYNCHRONOUS* PROCESSes only list the CLK signal in the sensitivity list;
(ii) All assignment statements within a synchronous process should be guarded by an IF CLK'event AND CLK='1'… condition;
(iii) Each signal on the LHS of an assignment statement guarded by an IF CLK'event AND CLK='1'… condition is the output of a flip-flop



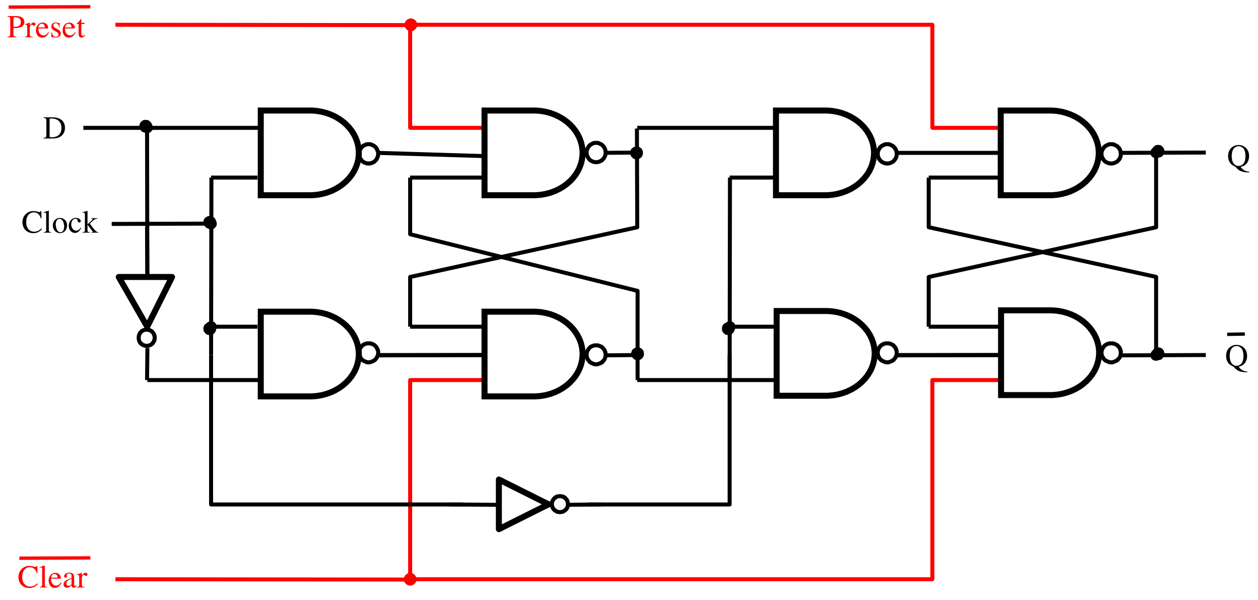
Equivalent code using a WAIT UNTIL statement

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY flipflop IS
     PORT ( D, Clock            : IN  STD_LOGIC ;
            Q          : OUT    STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
     PROCESS
     BEGIN
          WAIT UNTIL Clock'EVENT AND Clock = '1' ;
          Q <= D ;
     END PROCESS ;
END Behavior ;
```

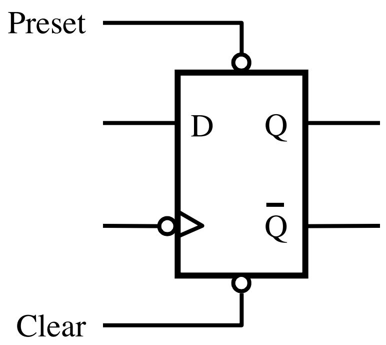
When used for synthesis, the WAIT UNTIL statement must be the first in a PROCESS block; all assignment statements that follow infer a flip-flop

Master-slave D flip-flop with *Clear* and *Preset*

- A design may call for a preset value on a FF
- Active low *Preset'* and *Clear'* inputs allow the flip-flop to be set to a given value asynchronously (independently of the *Clock*) – only one of them should be pulled low at a time



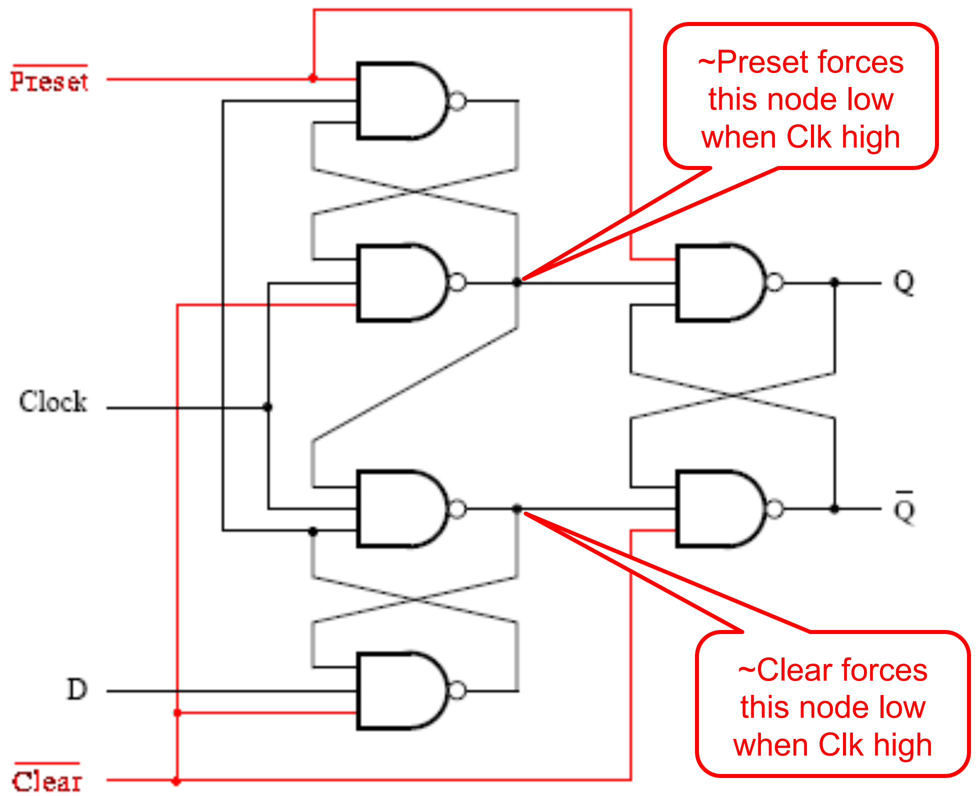
(a) Circuit



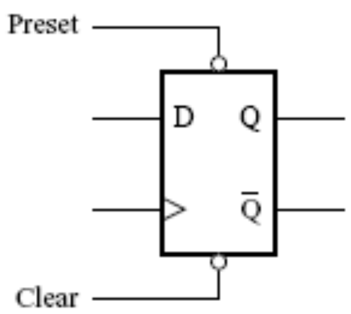
(b) Graphical symbol

- How long does the FF stay in the Clear or Preset state?

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


$\overline{\text{Preset}}$
Clock
D
$\overline{\text{Clear}}$
Q
$\overline{\text{Q}}$
~Preset forces this node low when Clk high
~Clear forces this node low when Clk high

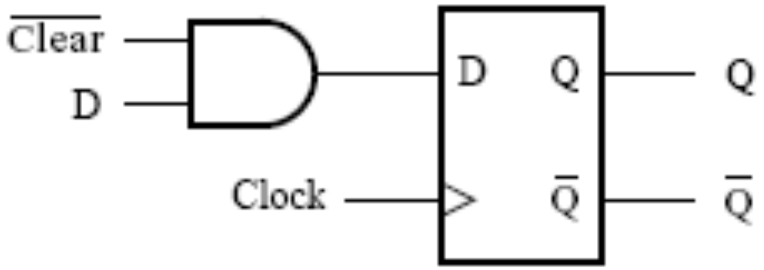
(a) Circuit



(b) Graphical symbol

Positive-edge-triggered D flip-flop with synchronous *Clear* and *Preset*

- Synchronous clear and preset is best done by gating the *D* input



D flip-flop with asynchronous reset

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT (  D, Resetn, CLK    : IN     STD_LOGIC ;
            Q                 : OUT    STD_LOGIC) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS ( Resetn, CLK )
    BEGIN
        IF Resetn = '0' THEN
            Q <= '0' ;
        ELSIF CLK'EVENT AND CLK = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Notes: (i) For a *synchronous process with an asynchronous reset/set*, both the CLK and the reset/set signal must be in the sensitivity list.
(ii) Only assign a ‘0’/‘1’ to the FF output within the reset/set condition.

D flip-flop with synchronous reset

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT (  D, Resetn, Clock  : IN      STD_LOGIC ;
            Q                 : OUT     STD_LOGIC) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock’EVENT AND Clock = ‘1’;
        IF Resetn = '0' THEN
            Q <= '0' ;
        ELSE
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Question: Which other way could you specify this behaviour?

Code for an eight-bit register with asynchronous reset

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY reg8 IS
    PORT (  D               : IN    STD_LOGIC_VECTOR(7 DOWNTO 0) ;
            Resetn, Clock   : IN    STD_LOGIC ;
            Q               : OUT   STD_LOGIC_VECTOR(7 DOWNTO 0) ) ;
END reg8;

ARCHITECTURE Behavior OF reg8 IS
BEGIN
    PROCESS (Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= “00000000”  ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Code for an *n*-bit register with asynchronous clear

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY regn IS
    GENERIC ( N : INTEGER := 16 );
    PORT (  D               : IN    STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
            Resetn, Clock   : IN    STD_LOGIC ;
            Q               : OUT   STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END regn;

ARCHITECTURE Behavior OF regn IS
BEGIN
    PROCESS (Resetn, Clock )
    BEGIN
        IF Resetn = '0' THEN
            Q <= (OTHERS => ‘0’)  ;
        ELSIF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

Parameterized component with default value of 16 for the data width parameter N

Idiom for setting all bits of a signal to 0s

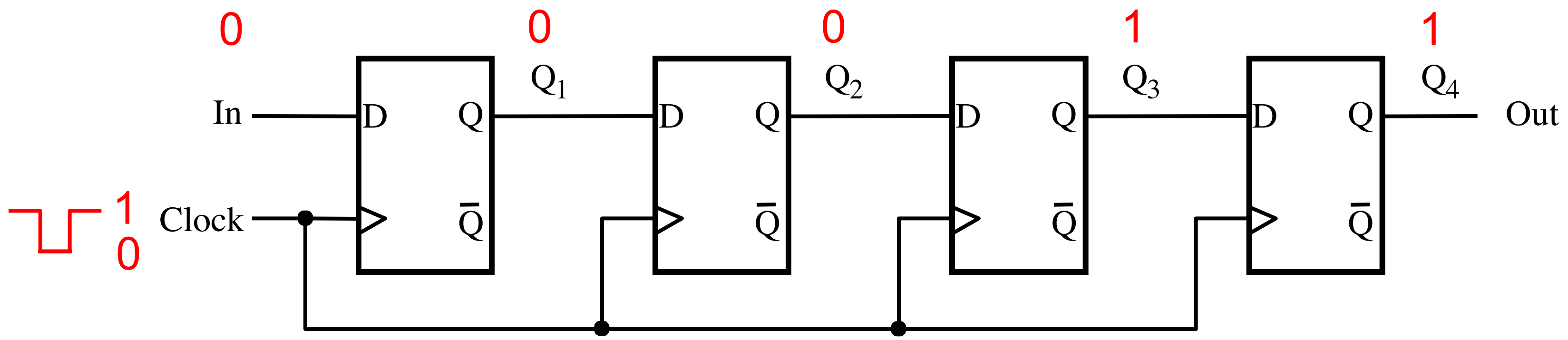
8-bit register based on *regn* component

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY reg8 IS
    PORT (  D           : IN    STD_LOGIC_VECTOR(7 DOWNTO 0) ;
            Resetn, Clk : IN    STD_LOGIC ;
            Q           : OUT   STD_LOGIC_VECTOR(7 DOWNTO 0) ) ;
END reg8 ;

ARCHITECTURE Structure OF reg8 IS
BEGIN
    reg8: regn
        GENERIC MAP ( N => 8 )
        PORT MAP ( D, Resetn, Clk, Q);
END Structure ;
```

Assumes *regn* component declared in the working directory

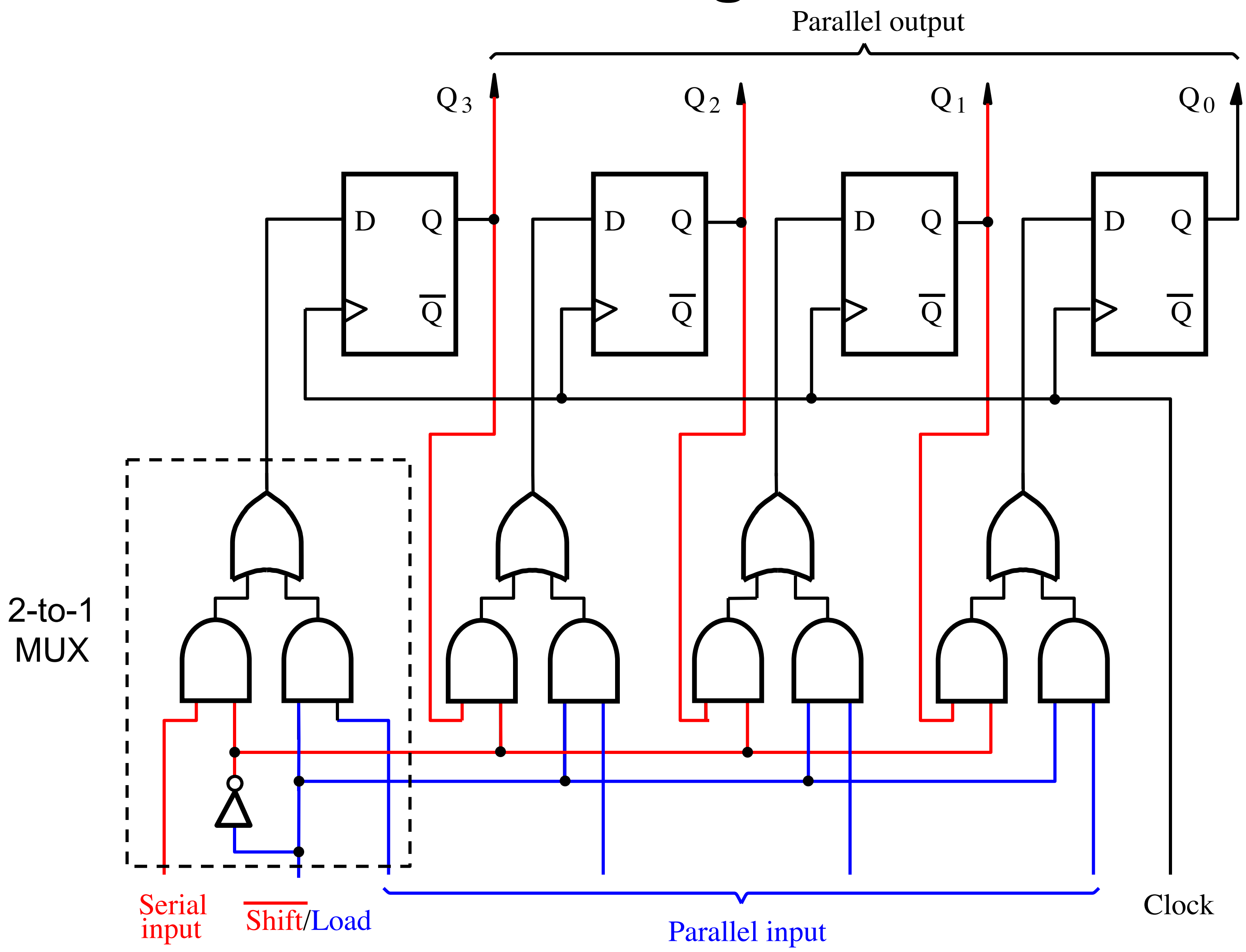
GENERIC MAP used to overwrite default parameter value

A simple shift register



	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

Parallel-access shift register



Behavioural code for a D flip-flop with a 2-to-1 multiplexer on the *D* input

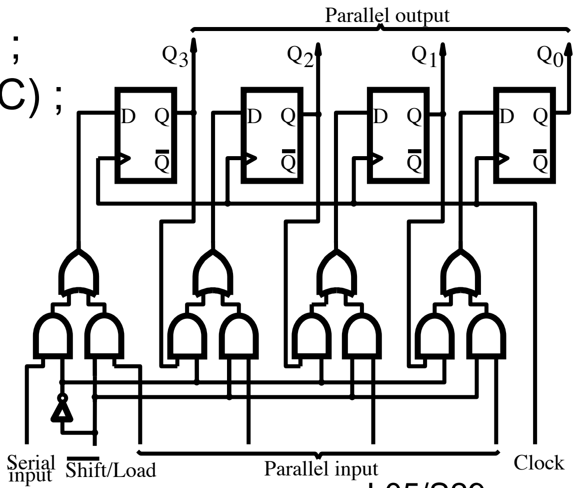
```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY muxdff IS
    PORT (  D0, D1, Sel, Clock : IN     STD_LOGIC ;
            Q                  : OUT    STD_LOGIC) ;
END muxdff;

ARCHITECTURE Behavior OF muxdff IS                        -- or:
BEGIN
    PROCESS                                               -- PROCESS (Clock)
    BEGIN                                                 -- BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1';           --      IF Clock' … THEN
        IF Sel = '0' THEN                                 --        IF Sel …
            Q <= D0 ;                                     --          etc.
        ELSE
            Q <= D1 ;                                     --        END IF;
        END IF ;                                          --    END IF;
    END PROCESS ;
END Behavior ;
```

Hierarchical code for a four-bit shift register

- Design hierarchies are recursive *structures* comprised of components, or sub-circuits, whose architectures, at the leaf level, are expressed in terms of their behaviours

BUFFER mode allows Q to be used in both IN and OUT modes

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY shift4 IS
        PORT (    R                : IN             STD_LOGIC_VECTOR( 3 DOWNTO 0) ;
                  L, w, Clock      : IN             STD_LOGIC ;
                  Q                : BUFFER         STD_LOGIC_VECTOR( 3 DOWNTO 0) ) ;
END shift4;

ARCHITECTURE Structure OF shift4 IS
        COMPONENT muxdff
                PORT ( D0, D1, Sel, Clock          : IN   STD_LOGIC ;
                       Q                           : OUT STD_LOGIC) ;
        END COMPONENT;
BEGIN
        Stage3:   muxdff PORT MAP ( w,      R(3), L, Clock, Q(3) );
        Stage2:   muxdff PORT MAP ( Q(3), R(2), L, Clock, Q(2) );
        Stage1:   muxdff PORT MAP ( Q(2), R(1), L, Clock, Q(1) );
        Stage0:   muxdff PORT MAP ( Q(1), R(0), L, Clock, Q(0) );
END Structure ;
```



Alternative (behavioural) code for a shift register

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY shift4 IS
    PORT (  R           : IN        STD_LOGIC_VECTOR( 3 DOWNTO 0) ;
            L, w, Clock : IN        STD_LOGIC ;
            Q           : BUFFER    STD_LOGIC_VECTOR( 3 DOWNTO 0) ) ;
END shift4;

ARCHITECTURE Behavior OF shift4 IS
BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL Clock'EVENT AND Clock = '1' ;
        IF L = '1' THEN
            Q <= R ;
        ELSE
            Q(0) <= Q(1) ;
            Q(1) <= Q(2) ;
            Q(2) <= Q(3) ;
            Q(3) <= w ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

BUFFER mode allows Q to appear on both the left and right sides of signal assignments

A WAIT UNTIL statement implies all signals assigned a value inside the process have to be implemented as the output of a flip-flop

Identical code, which reverses the ordering of statements 16 – 19 in L05/S30

```
1    LIBRARY ieee ;
2    USE ieee.std_logic_1164.all ;

3    ENTITY shift4 IS
4        PORT (  R              : IN        STD_LOGIC_VECTOR( 3 DOWNTO 0) ;
5                L, w, Clock    : IN        STD_LOGIC ;
6                Q              : BUFFER    STD_LOGIC_VECTOR( 3 DOWNTO 0) ) ;
7    END shift4;

8    ARCHITECTURE Behavior OF shift4 IS
9    BEGIN
10       PROCESS
11       BEGIN
12           WAIT UNTIL Clock'EVENT AND Clock = '1' ;
13           IF L = '1' THEN
14               Q <= R ;
15           ELSE
16               Q(3) <= w ;
17               Q(2) <= Q(3) ;
18               Q(1) <= Q(2) ;
19               Q(0) <= Q(1) ;
20           END IF ;
21       END PROCESS ;
22   END Behavior ;
```

IMPORTANT:
Why is the statement order immaterial?

Code for an *n*-bit left-to-right shift register

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY shiftn IS
        GENERIC ( N : INTEGER := 8 ) ;
        PORT (  R               : IN      STD_LOGIC_VECTOR( N-1 DOWNTO 0) ;
                L, w, Clock     : IN      STD_LOGIC ;
                Q               : BUFFER  STD_LOGIC_VECTOR( N-1 DOWNTO 0) ) ;
  END shiftn;

  ARCHITECTURE Behavior OF shiftn IS
  BEGIN
        PROCESS
        BEGIN
              WAIT UNTIL Clock'EVENT AND Clock = '1' ;
              IF L = '1' THEN
                    Q <= R ;
              ELSE
                    Genbits: FOR i IN 0 to N-2 LOOP
                          Q(i) <= Q(i+1) ;
                    END LOOP ;
                    Q(N-1) <= w ;
              END IF ;
        END PROCESS ;
  END Behavior ;
```

Just as the FOR GENERATE statement is used to generate a set of concurrent statements, the FOR LOOP statement is used to generate a set of sequential statements

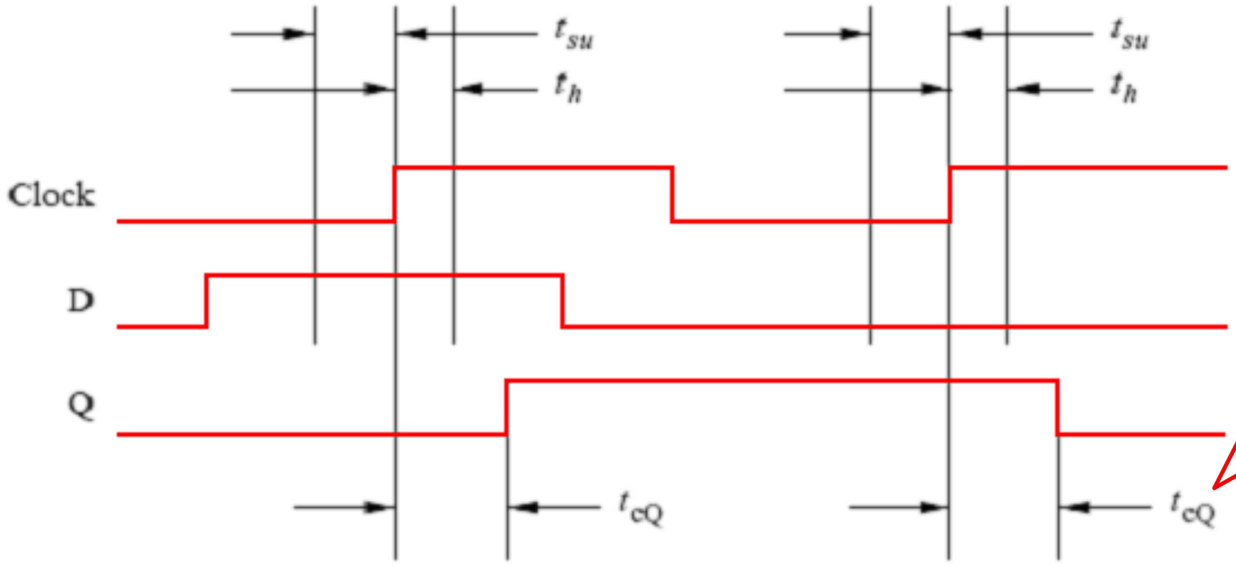
Flip-flop timing parameters

- Three important parameters that need to be considered in the design of sequential circuits:
  - *Propagation delay*, t_{cQ}, the time needed for the output of the FF to change after the triggering clock edge has occurred
  - *Setup time*, t_{su}, the time interval the input needs to be stable for *prior to* the triggering clock edge, for it to be reliably read
  - *Hold time*, t_h, the time interval the input needs to be stable for *after* the triggering clock edge, for it to be reliably read

- The magnitude of these parameters depend upon the design of the flip-flop, the process technology used to implement them, and the source voltage

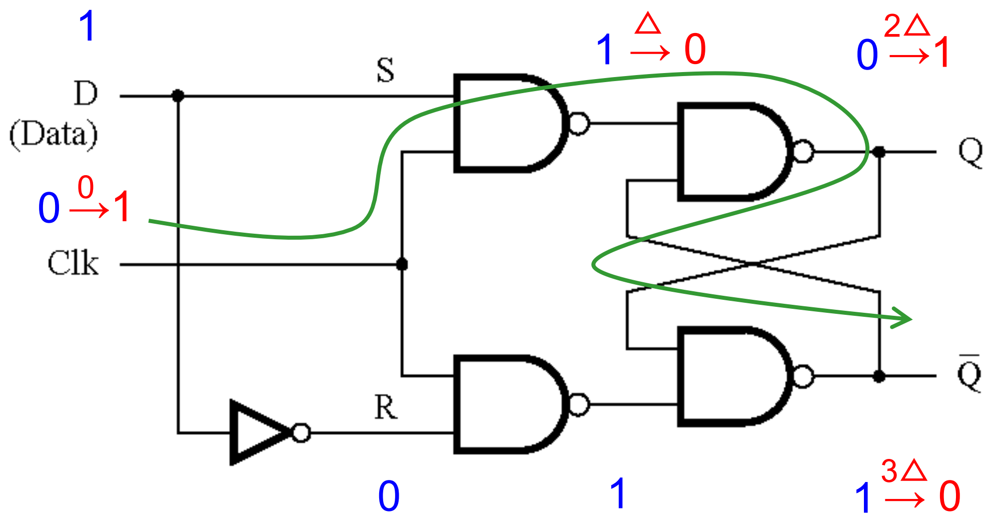
Propagation delay

- *Propagation delay is the time it takes for the new value to emerge from a flip-flop after the triggering edge*



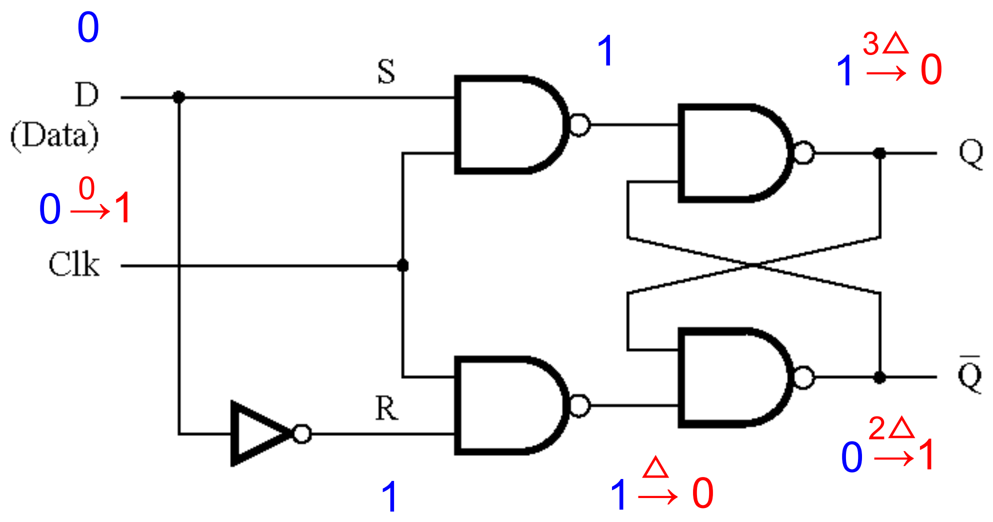
t_{cQ} (propagation delay) may not be the same for $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions, and is usually specified using min and max values

t_{cQ} for a gated D latch



1
D
(Data)
$0 \xrightarrow{0} 1$
Clk
S
R
0
$1 \xrightarrow{\Delta} 0$
1
$0 \xrightarrow{2\Delta} 1$
Q
$\bar{Q}$
$1 \xrightarrow{3\Delta} 0$

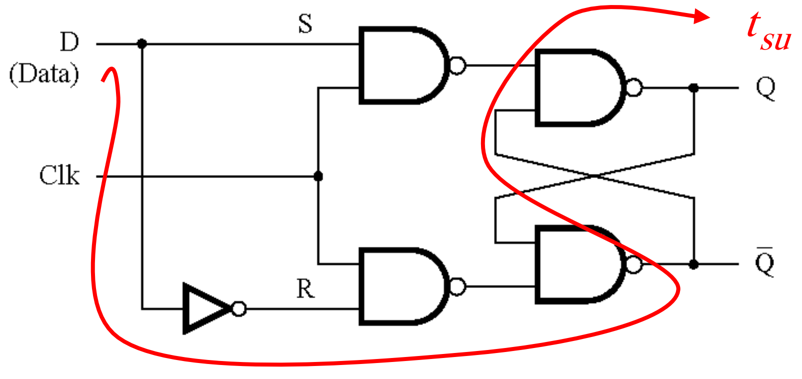
Here, t_{cQ} is 2△ for 0→1 transitions, but 3△ for 1→0 transitions



0
D
(Data)
$0 \xrightarrow{0} 1$
Clk
S
R
1
$1 \xrightarrow{3\Delta} 0$
Q
$\bar{Q}$
1
$1 \xrightarrow{\Delta} 0$
$0 \xrightarrow{2\Delta} 1$

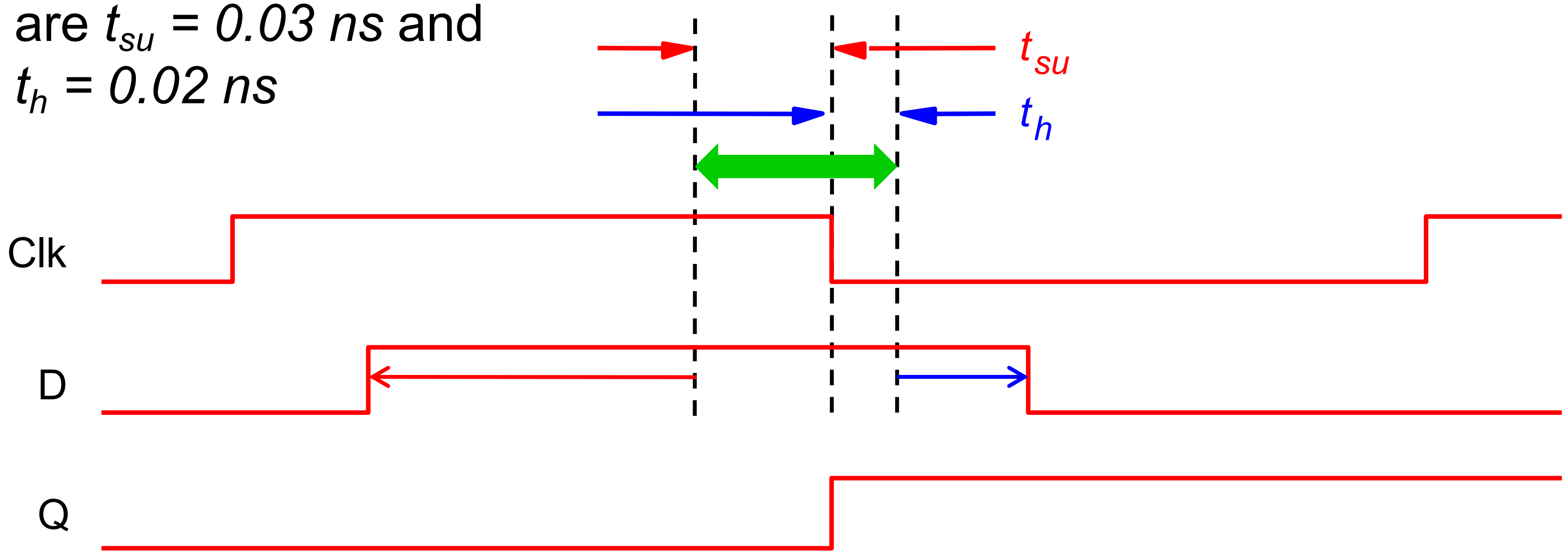
Setup and hold times

- The designer of the circuit that generates the *D* signal must ensure setup and hold times are satisfied
- Together, they define a window of time around the triggering clock edge during which *D* must be stable
- Typical values for 28nm CMOS are t_{su} = *0.03 ns* and t_h = *0.02 ns*

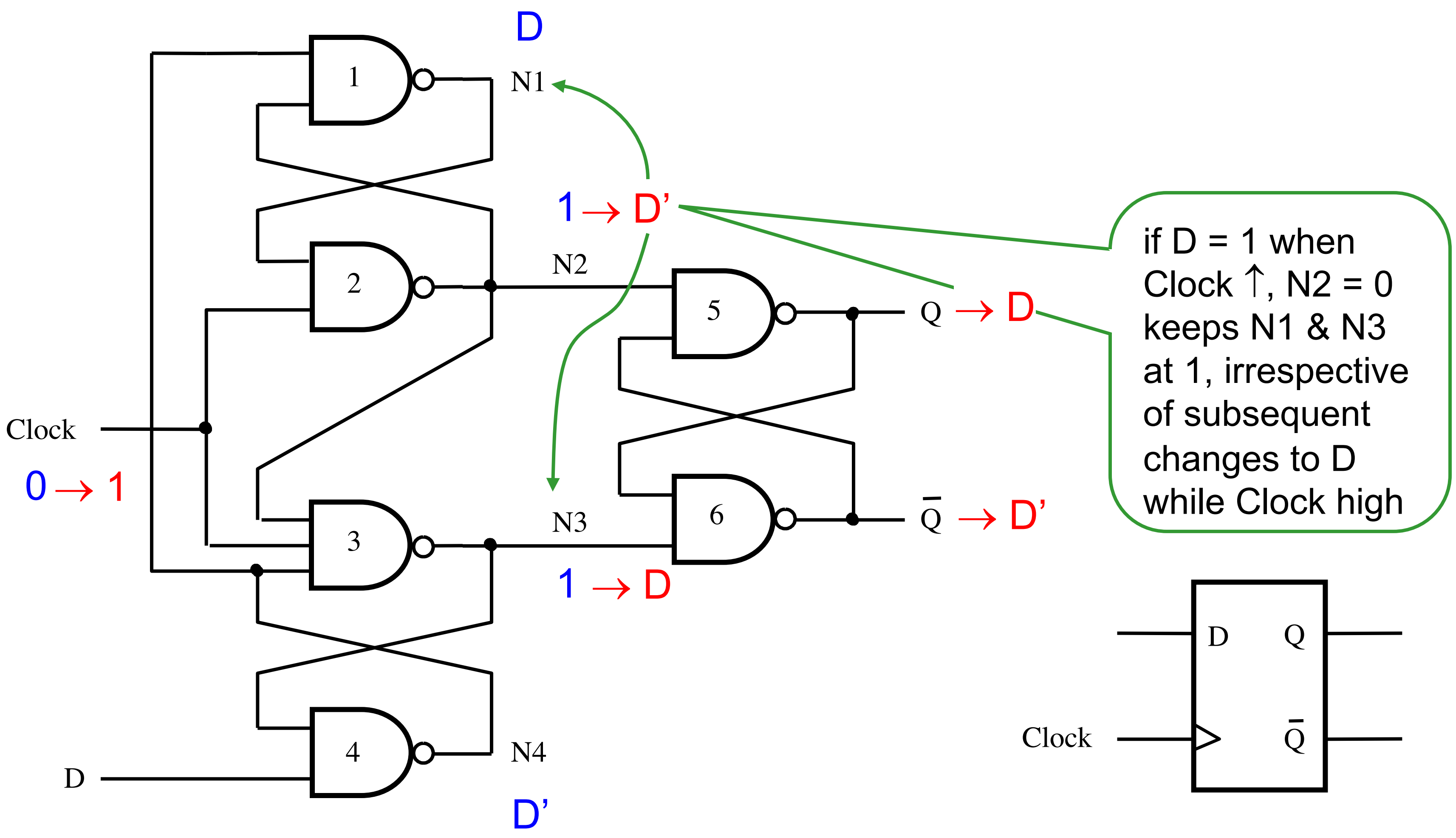


(a) Circuit

$t_{su} \Rightarrow$ a change in D has to have had time to be seen at the outputs before the negative Clk edge

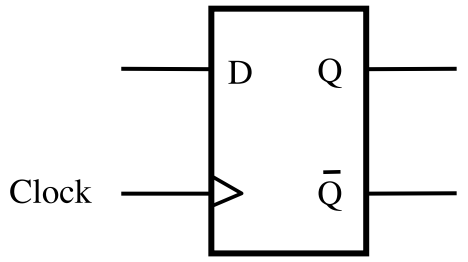


Recall positive-edge-triggered D flip-flop



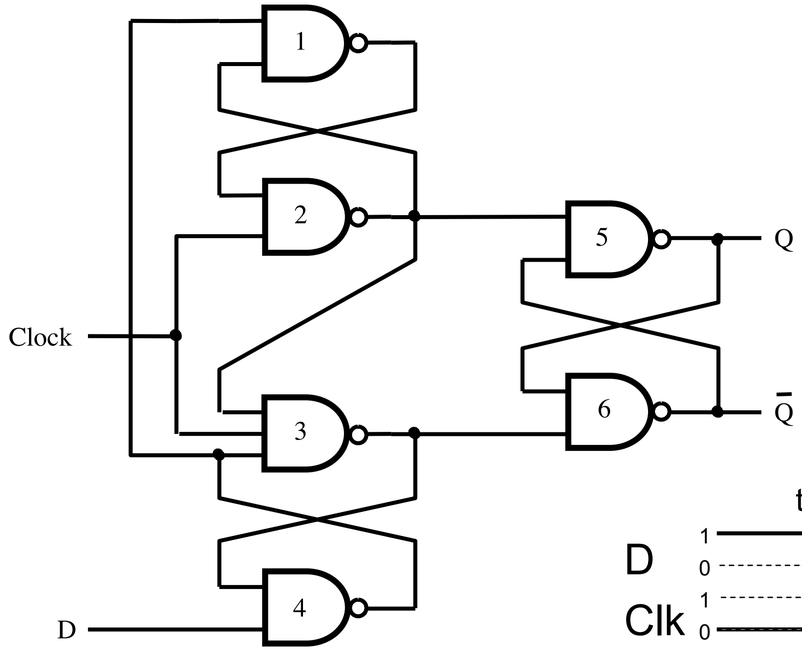
if D = 1 when Clock ↑, N2 = 0 keeps N1 & N3 at 1, irrespective of subsequent changes to D while Clock high

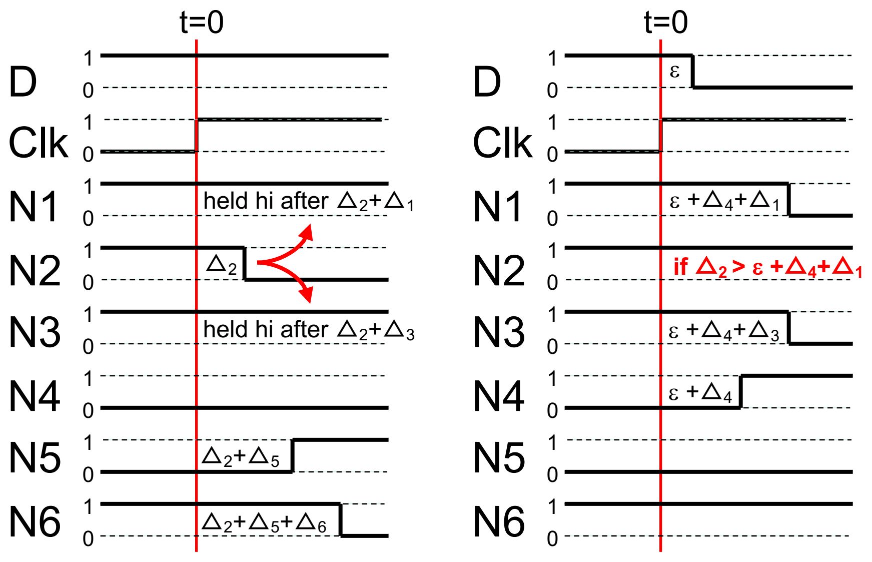
(a) Circuit



(b) Graphical symbol

t_h for a positive-edge-triggered D flip-flop

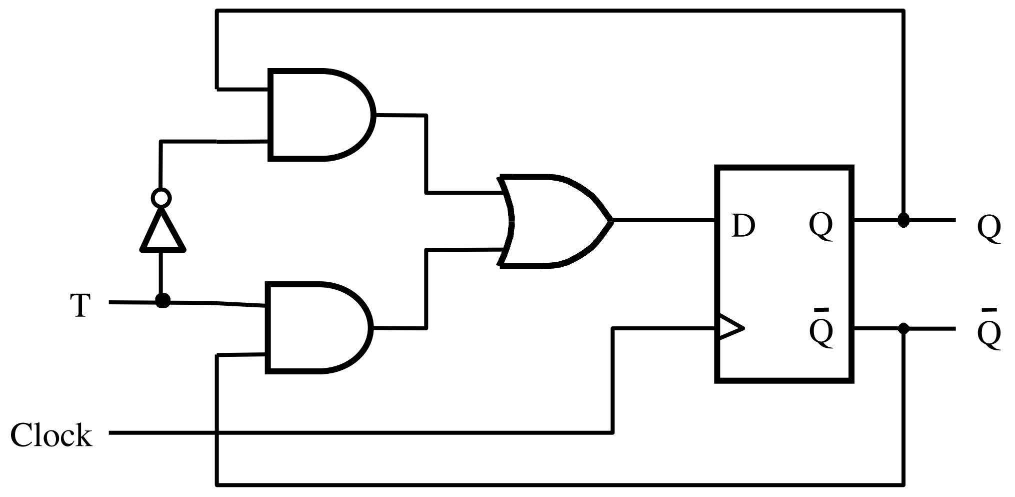




The larger ε is, the less likely it is for N2 to be held high

(a) Normal operation (b) Hold time violation

T flip-flop



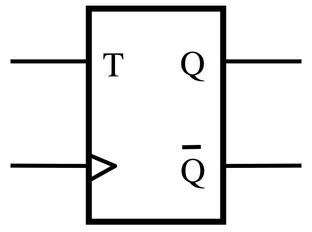
(a) Circuit

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

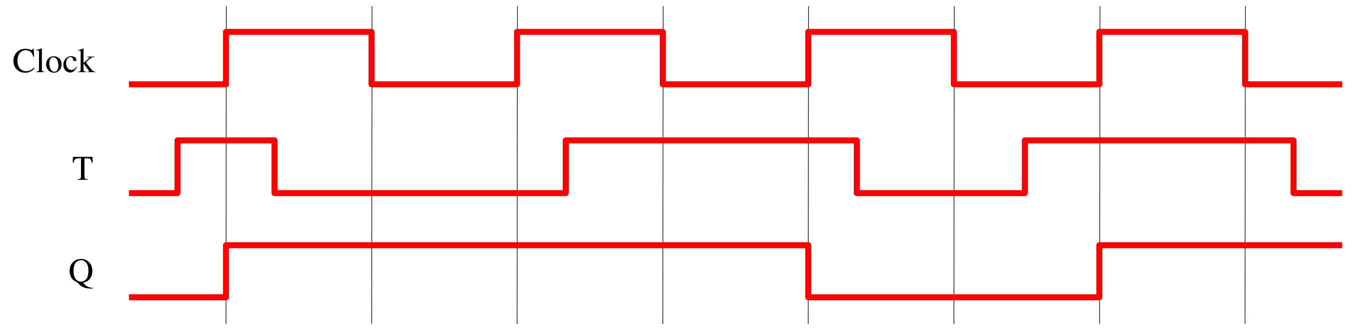
(b) Characteristic table

(c) Characteristic equation:

Q(t+1) = T.Q’(t) + T’.Q(t)
= T XOR Q(t)



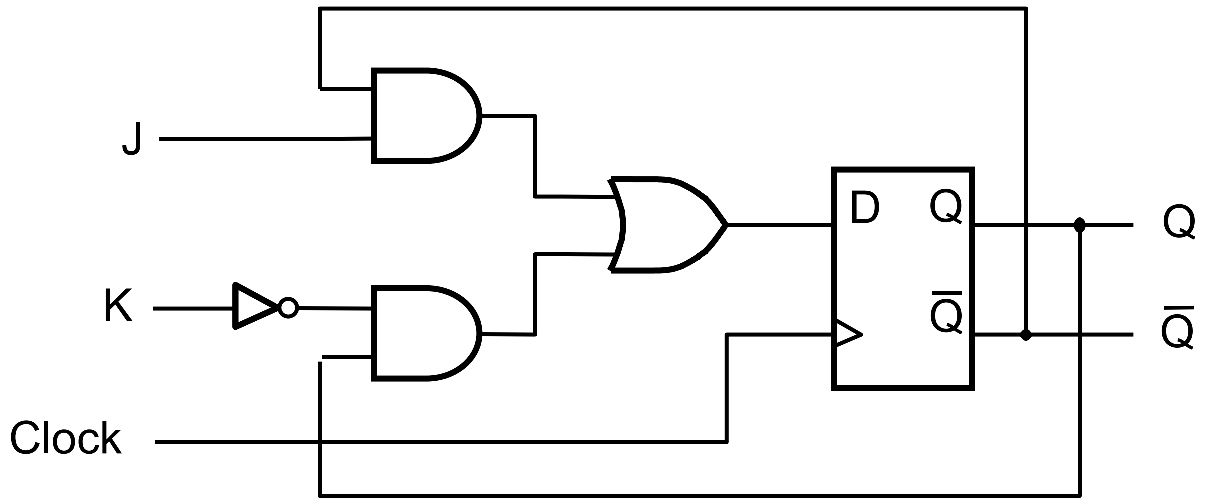
(d) Graphical symbol



(e) Timing diagram

JK flip-flop

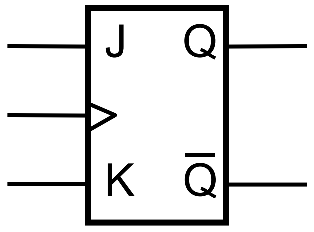
- Combines the features of an SR flip-flop and a T flip-flop



(a) Circuit

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

(b) Characteristic table

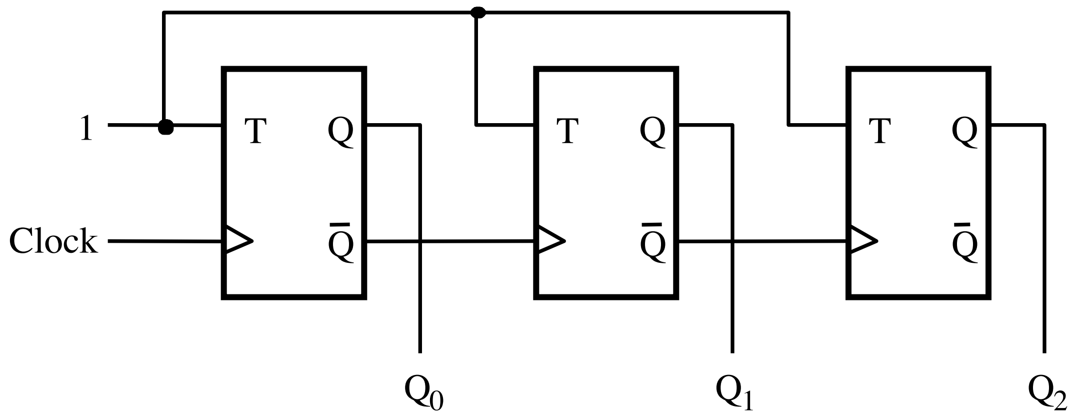


(c) Graphical symbol

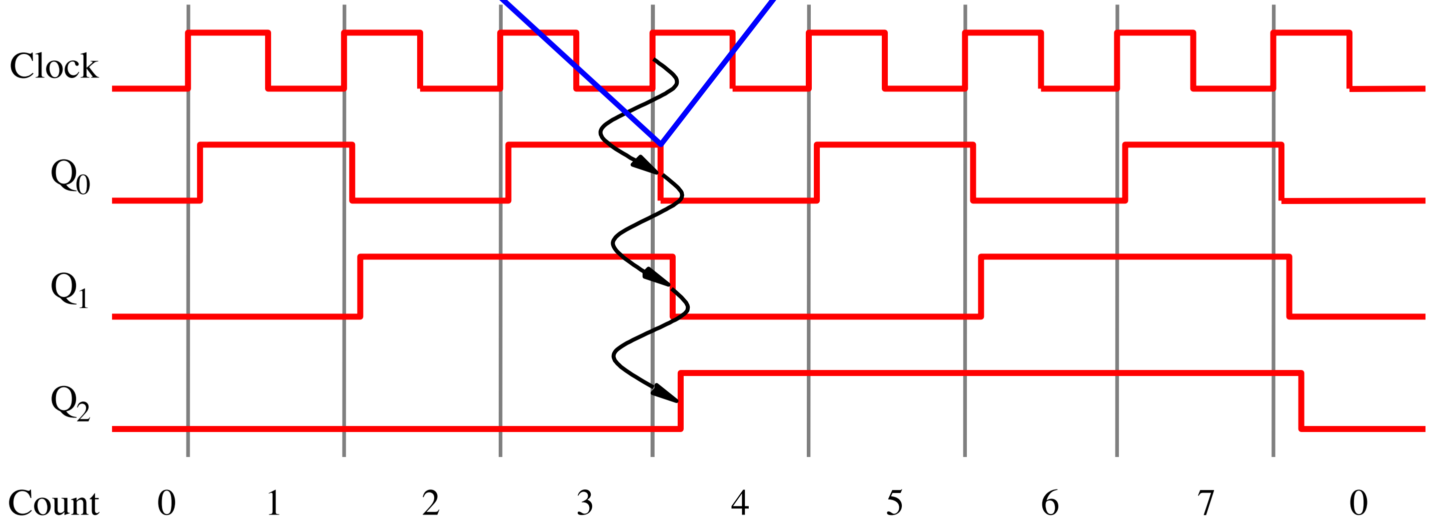
A three-bit up-counter (ripple counter)

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

Characteristic table

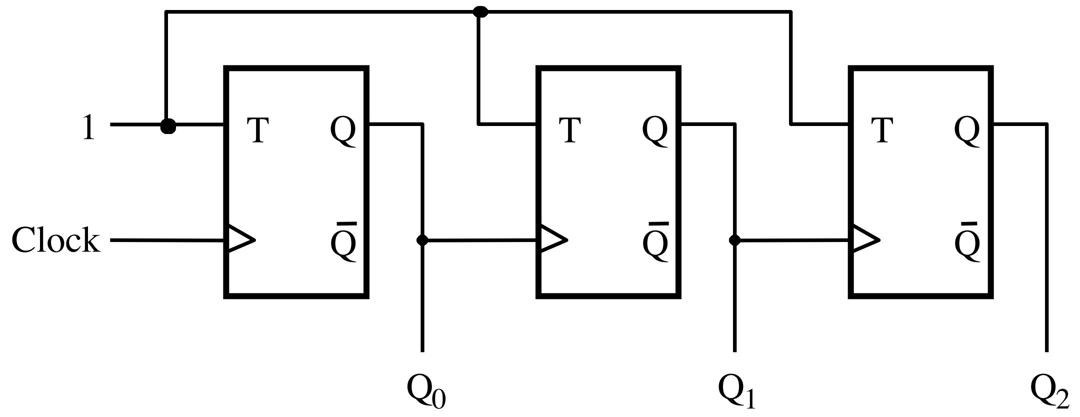


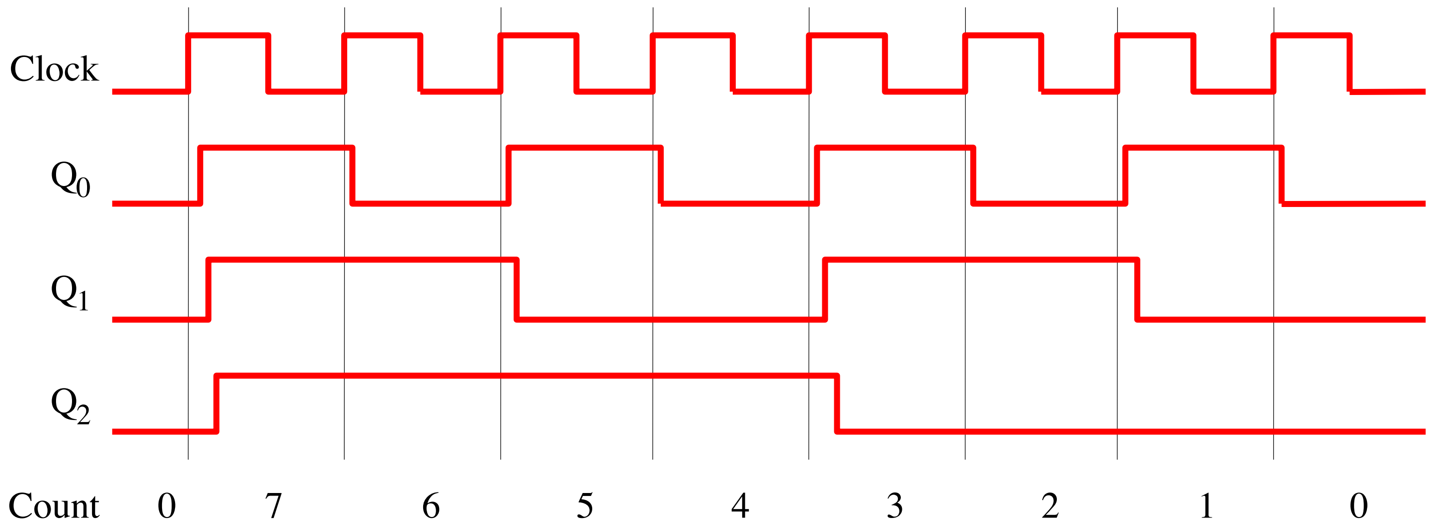
The ripple effect of the triggering edge due to t_{cQ} propagation delays causes glitching from 011 → 010 → 000 → 100



Timing diagram

A three-bit down-counter



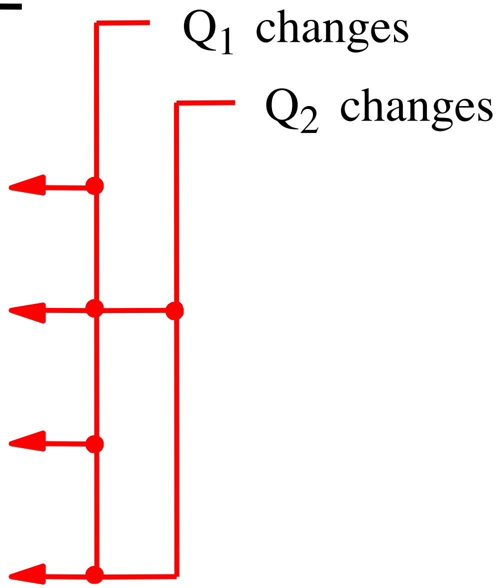


Timing diagram

Derivation of a synchronous up-counter

(based on T flip-flops triggered by the one clock signal) in which all output bits change at the same time

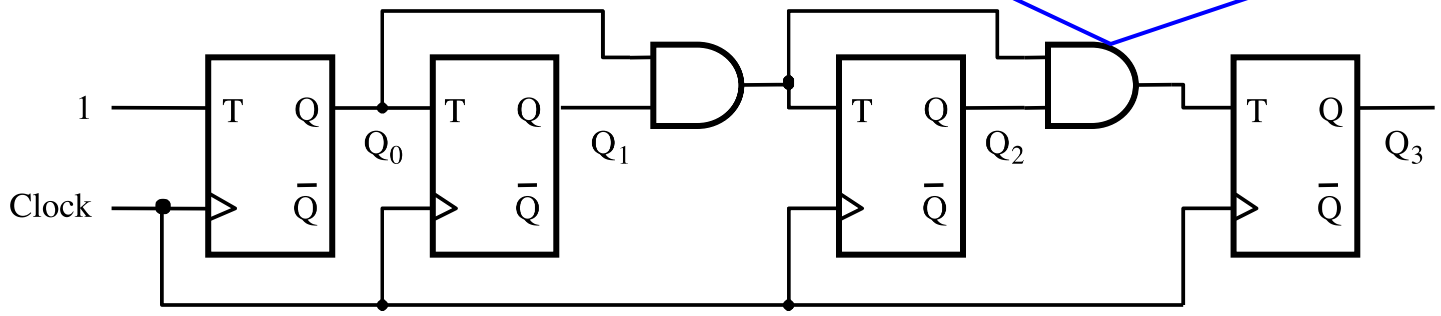
Clock cycle	Q_2	Q_1	Q_0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
8	0	0	0

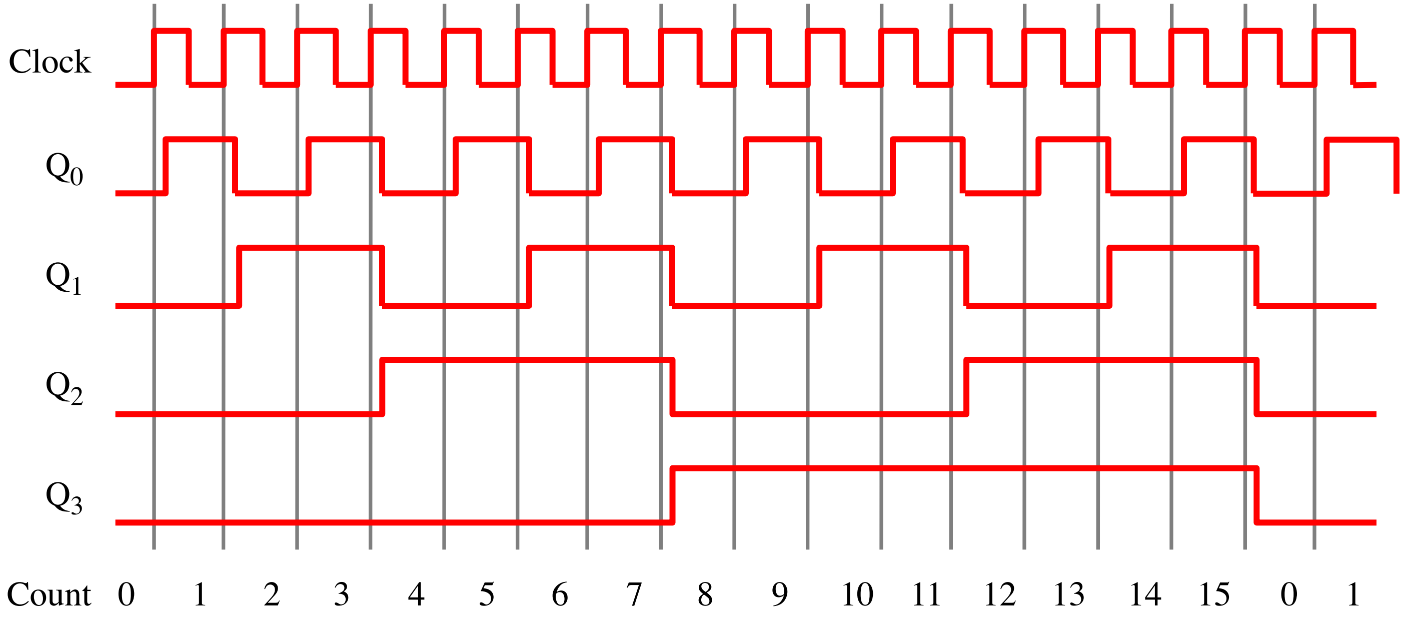


$T_0 = 1$
$T_1 = Q_0$
$T_2 = Q_0Q_1$
$T_3 = Q_0Q_1Q_2$
…
$T_n = Q_0Q_1\ldots Q_{n-1}$

A four-bit synchronous up-counter

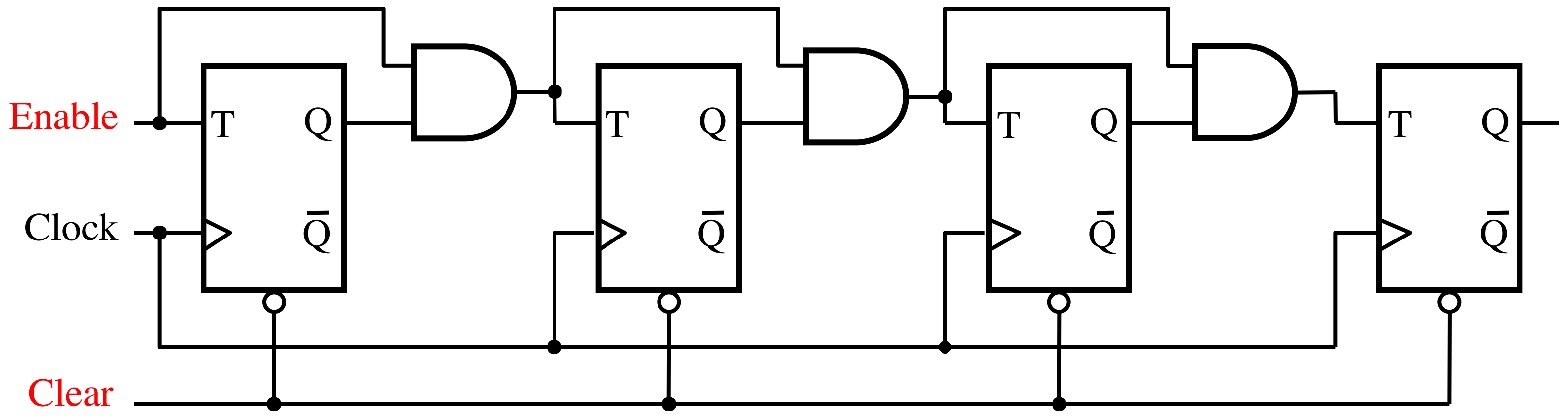
Need to ensure that the *clock_period* ≥ t_{cQ} + delay of the AND gate chain + t_{su}



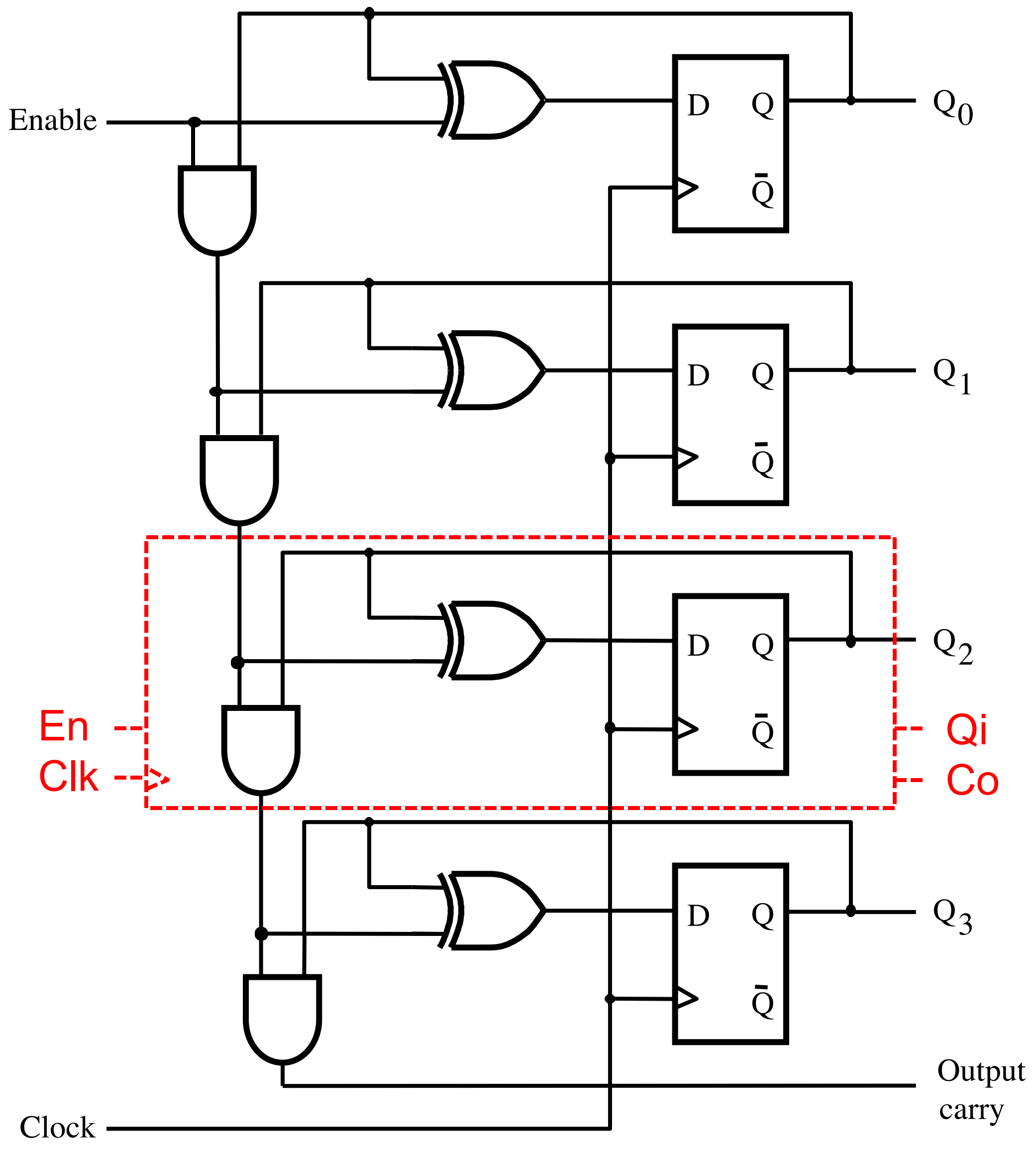


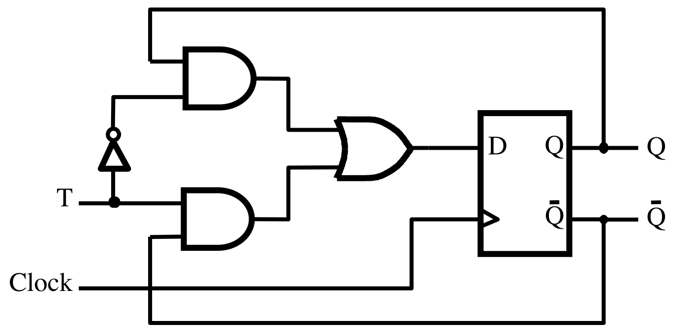
Timing diagram

Inclusion of an Enable and asynchronous Clear capability



A four-bit synchronous counter with D FFs





T flip-flop:
Q(t+1) = T XOR Q(t)

Can you see how you would describe the structure in VHDL?

Behavioural code for a four-bit up-counter with asynchronous clear

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY upcount IS
        PORT (     Clock, Resetn, E : IN          STD_LOGIC ;
                   Q                : OUT         STD_LOGIC_VECTOR( 3 DOWNTO 0) ) ;
END upcount;

ARCHITECTURE Behavior OF upcount IS
        SIGNAL Count : STD_LOGIC_VECTOR( 3 DOWNTO 0) ;
BEGIN
        PROCESS (Clock, Resetn )
        BEGIN
                IF Resetn = ‘0’ THEN
                        Count <= “0000” ;
                ELSIF (Clock’EVENT AND Clock = ‘1’) THEN
                        IF E = ‘1’ THEN
                                Count <= Count + 1 ;
                        ELSE
                                Count <= Count ;
                        END IF;
                END IF ;
        END PROCESS ;
        Q <= Count ;
END Behavior ;
```

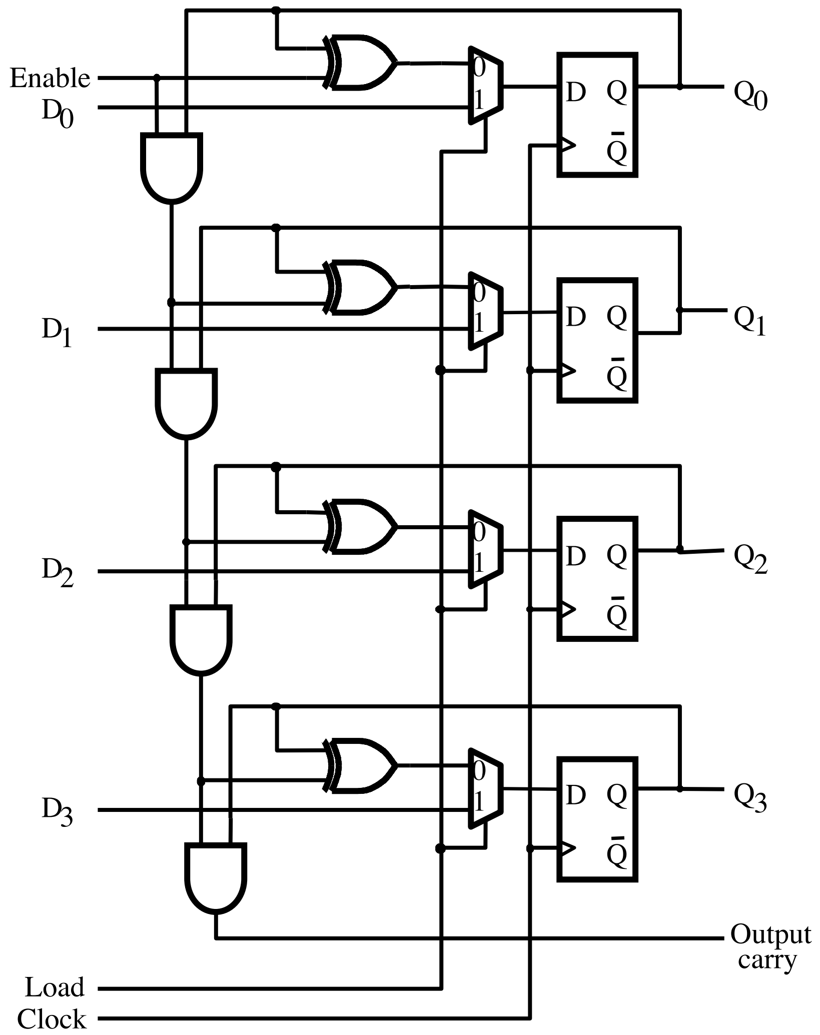
Needed to be able to increment Count

Advantages of behavioural code over structural code for this design?

While not required because of implied memory semantics, this statement is included for clarity

Starting the count from any value

- A counter with parallel-load capability

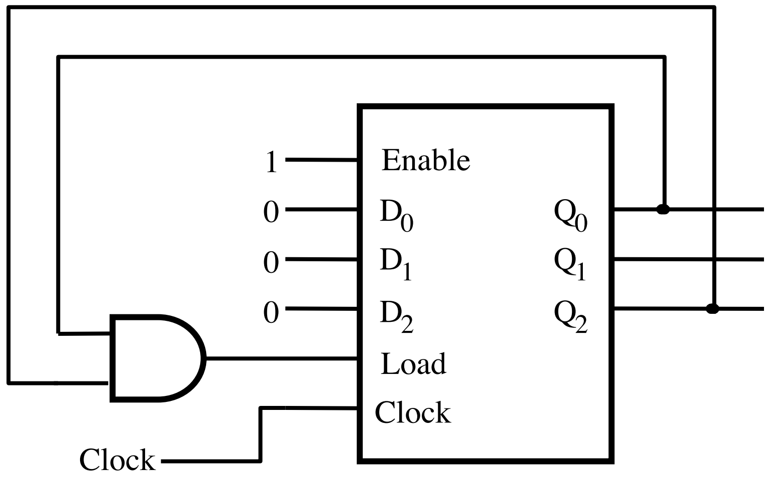


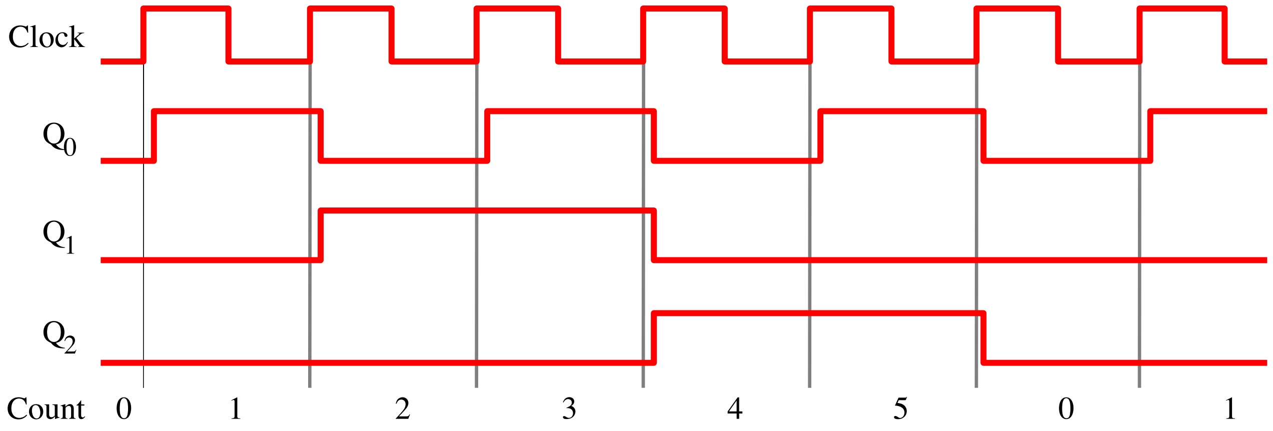
A four-bit counter with parallel load, using INTEGER signals

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY upcount IS
        PORT (    R                 : IN          INTEGER RANGE 0 TO 15 ;
                  Clock, Resetn, L  : IN          STD_LOGIC ;
                  Q                 : BUFFER      INTEGER RANGE 0 TO 15 ) ;
END upcount;

ARCHITECTURE Behavior OF upcount IS
BEGIN
        PROCESS (Clock, Resetn )
        BEGIN
                IF Resetn = ‘0’ THEN
                        Q <= 0 ;
                ELSIF (Clock’EVENT AND Clock = ‘1’) THEN`
                        IF L = ‘1’ THEN
                                Q <= R ;
                        ELSE
                                Q <= Q + 1 ;
                        END IF;
                END IF ;
        END PROCESS ;
END Behavior ;
```

Controlling the count range

- A modulo-6 counter with synchronous reset



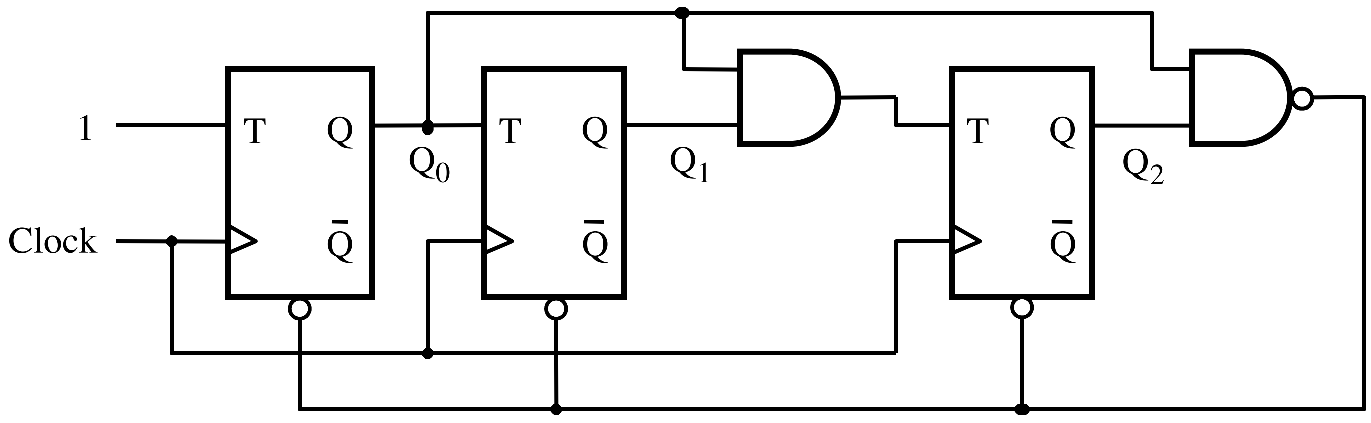


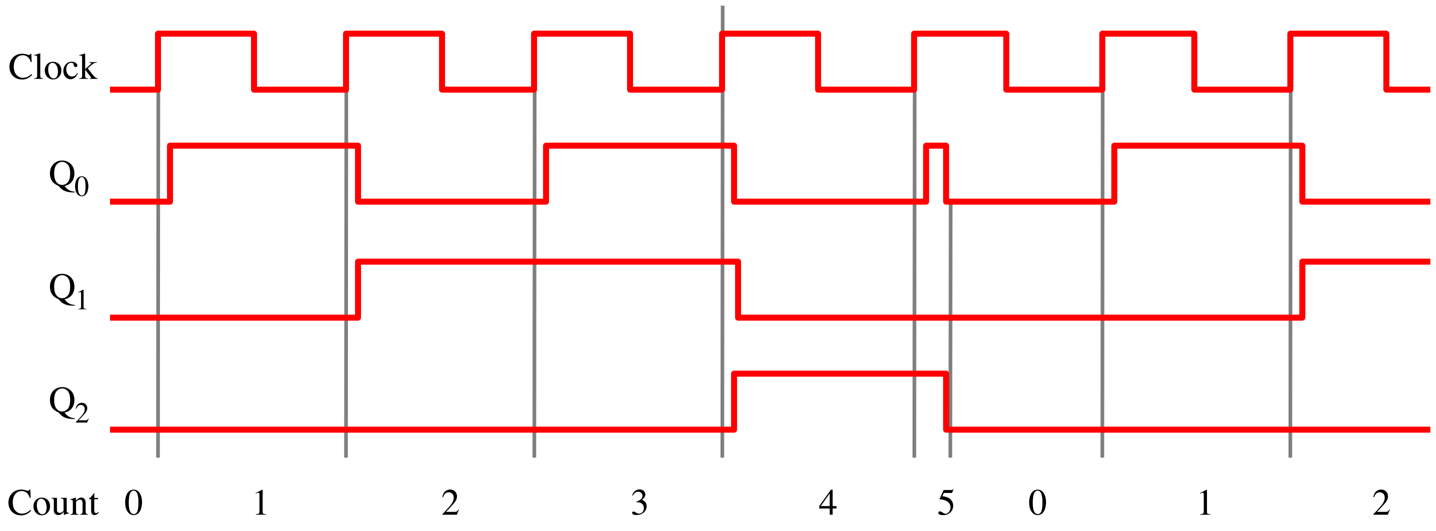
Timing diagram

Code for a down-counter

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY downcnt IS
	GENERIC ( modulus		: INTEGER := 8 ) ;
	PORT (	Clock, L, E		: IN		STD_LOGIC ;
		Q			: OUT		INTEGER RANGE 0 TO modulus-1 ) ;
END downcnt;

ARCHITECTURE Behavior OF downcnt IS
	SIGNAL Count : INTEGER RANGE 0 TO modulus-1 ;
BEGIN
	PROCESS
	BEGIN
		WAIT UNTIL (Clock'EVENT AND Clock = '1') ;
		IF L = '1' THEN
			Count <= modulus-1 ;
		ELSE
			IF E = '1' THEN
				Count <= Count-1 ;
			END IF;
		END IF ;
	END PROCESS ;
	Q <= Count ;
END Behavior ;
```

A modulo-6 counter with asynchronous reset

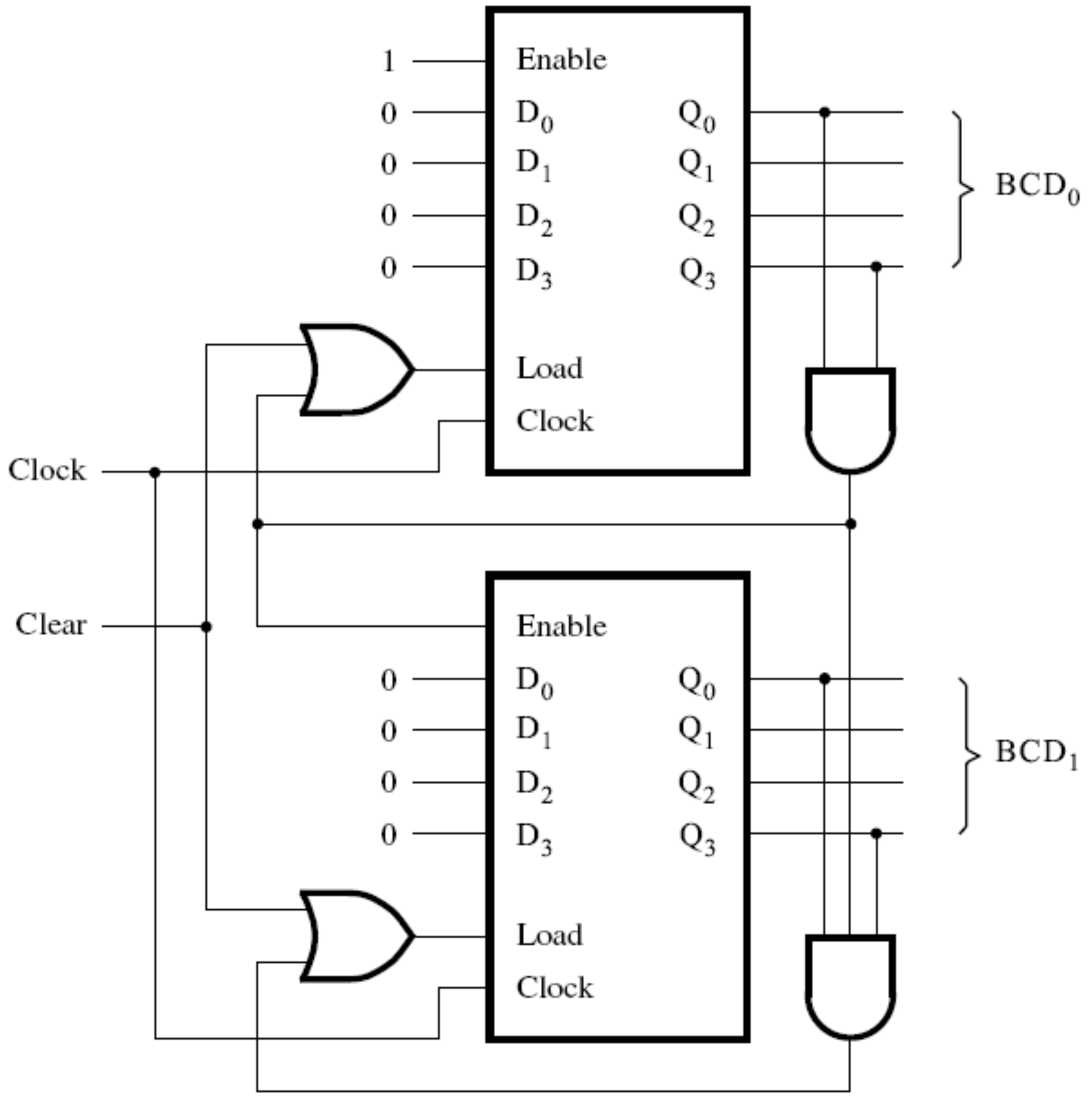




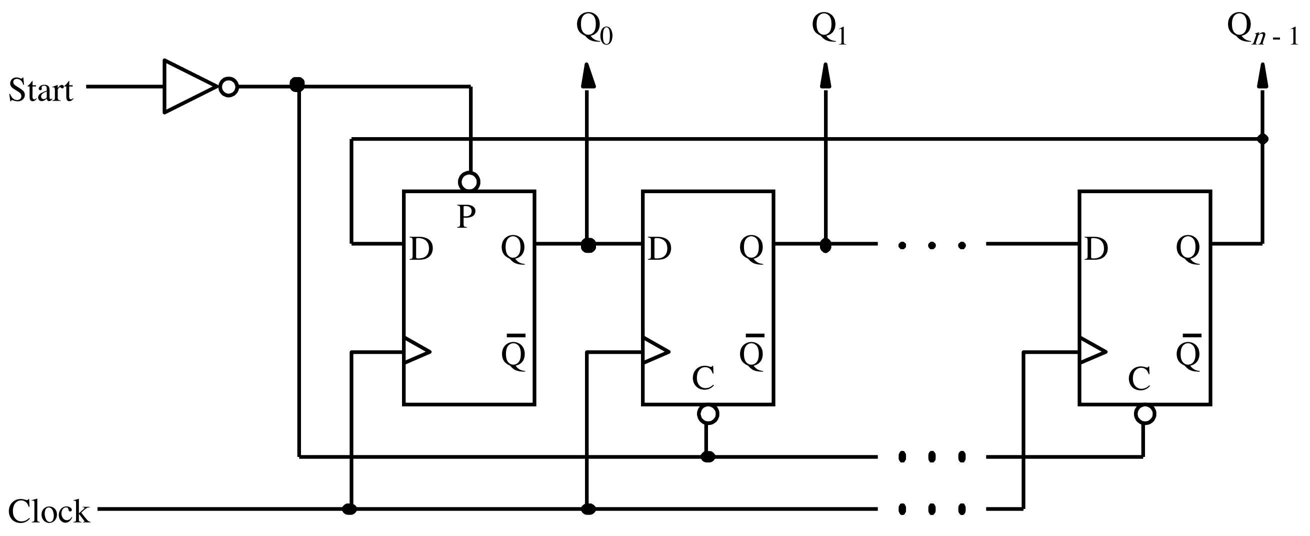
Timing diagram

Glitching is undesirable: can have unintended consequences if not designed away from the outset

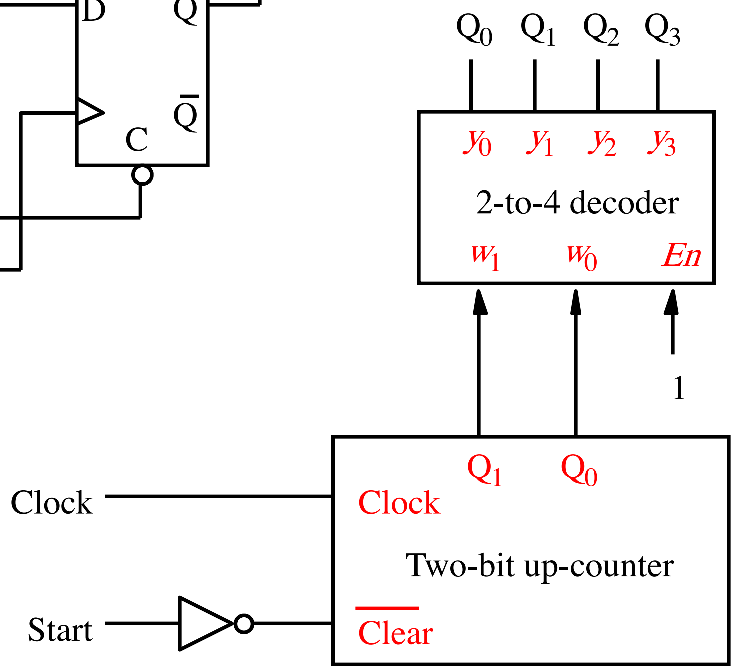
A two-digit BCD counter



Ring counter



(a) An *n*-bit ring counter



(b) A four-bit ring counter

Timing analysis of flip-flop circuits

- Usually the maximum clock frequency a circuit can be operated at, F_{max}, needs to be determined
- Whether any hold times are violated also needs to be determined
- Timing parameters of flip-flops were introduced in slides L05/S33 to L05/S38 – these include the set-up time t_{su}, the hold time t_h, and the clock-to-Q propagation delay t_{cQ}

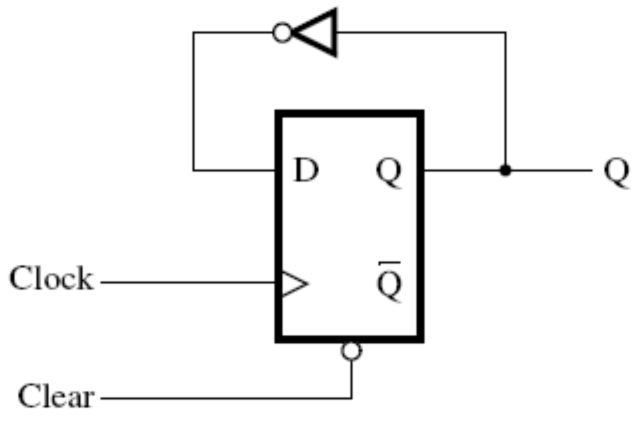
Timing analysis of a simple flip-flop circuit

- Consider the simple circuit shown, and let's assume that t_{su} = 0.6 ns, t_h = 0.4 ns, and 0.8 ns <= t_{cQ} <= 1.0 ns
- Furthermore, assume the delay of a *k*-input gate is 1 + 0.1*k* ns
- To calculate $T_{min} = 1/F_{max}$, we need to determine the longest timing path in the circuit (a.k.a. *critical path*) that starts and ends at a flip-flop
- Here:

  $T_{min} = max\{t_{cQ}\} + t_{NOT} + t_{su}$
  i.e. T_{min} = 1.0 + 1.1 + 0.6 = 2.7 ns
  and $F_{max} = 1/T_{min}$ = 370 MHz
  any faster, and t_{su} would not be satisfied

- Need to check hold time violations by considering the shortest possible delay from any +ve clock edge to any flip-flop input
- Here:

  $min\{t_{cQ}\} + t_{NOT}$ = 0.8 + 1.1 = 1.9 ns
  > t_h = 0.4 ns ∴ no violation



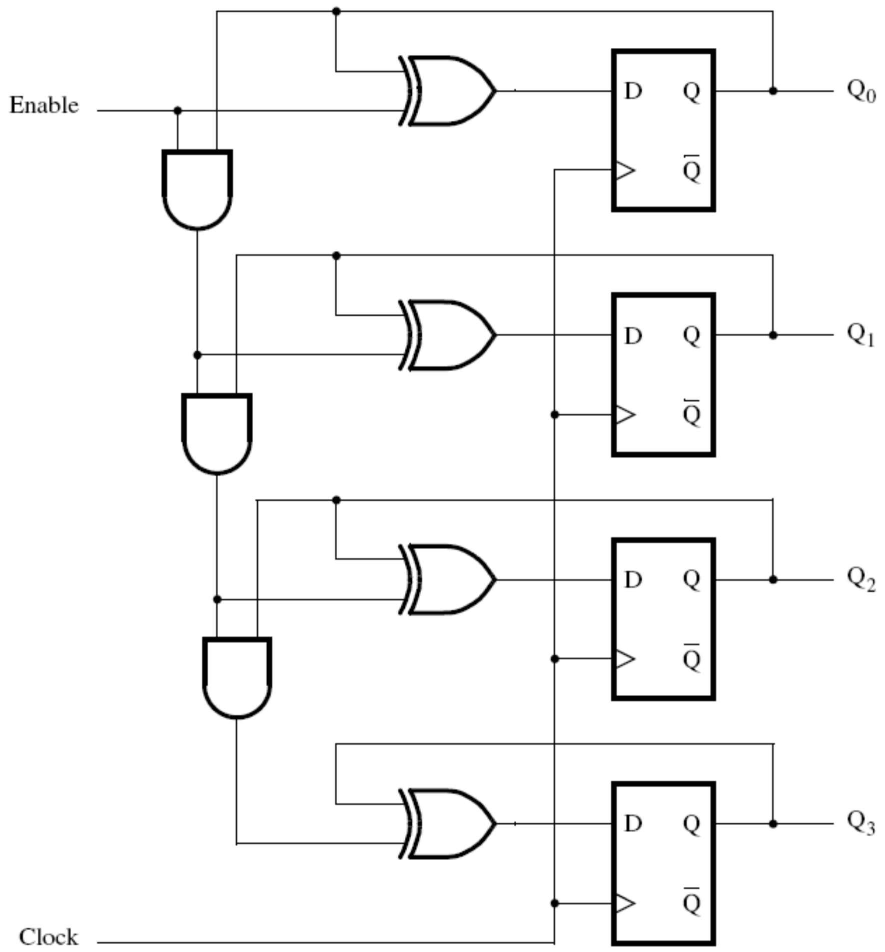
Timing analysis of a 4-bit counter

Assume the same timing parameters as in the previous example; critical path:

$T_{min} = max\{t_{cQ(Q0)}\} + 3(t_{AND}) + t_{XOR}$
$+ t_{su(Q3)}$
$= 1.0 + 3(1.2) + 1.2 + 0.6$
$= 6.4$ ns

$F_{max} = 1/6.4$ ns $= 156$ MHz (this assumes Enable is well behaved; if not, F_{max} may need to be reduced)

Shortest path from clock to D for each FF is $\min\{t_{cQ}\} + t_{XOR} = 0.8 + 1.2 = 2.0$ ns $> t_h = 0.4$ ns $\therefore$ no hold violations (given Enable is well behaved)

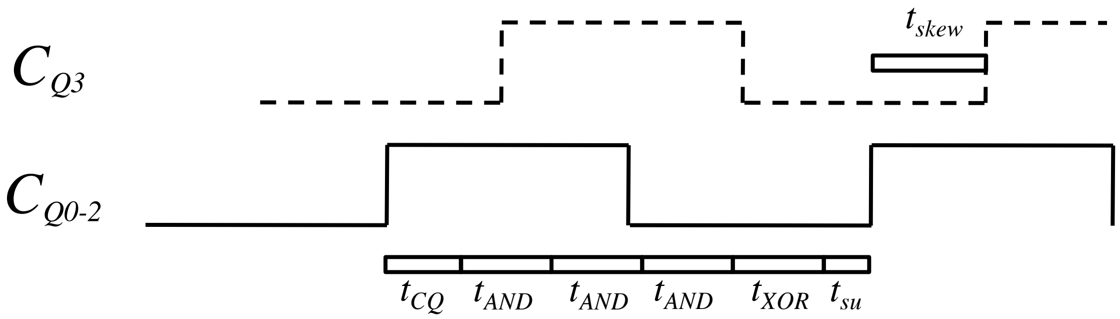


Q: When is it best to (de)assert *Enable*?

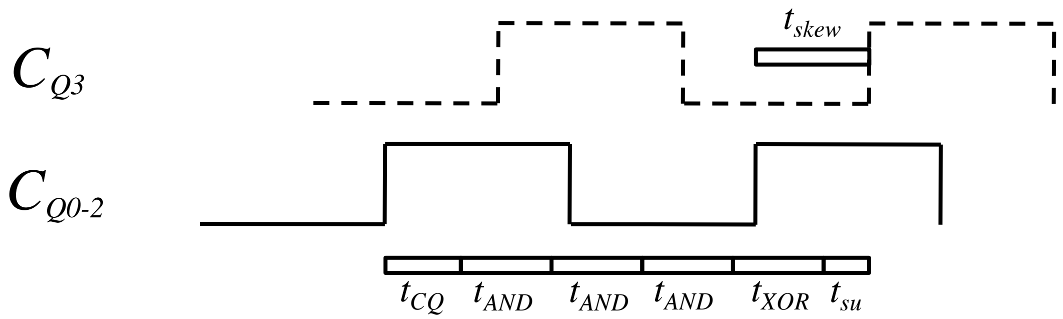
Clock skew

- Clock skew is the spread in time (relative delay) in clock edges arriving at the various synchronous components of a digital circuit
- Mostly, these are caused by wire delays
- FPGAs have special clock distribution networks, which use low-resistance (fat) wiring tracks, buffers that amplify the clock signal, and “balanced” layouts, such as H-trees with the root located at the centre of the chip, to minimize clock skew

Effect of clock skew on F_{max}



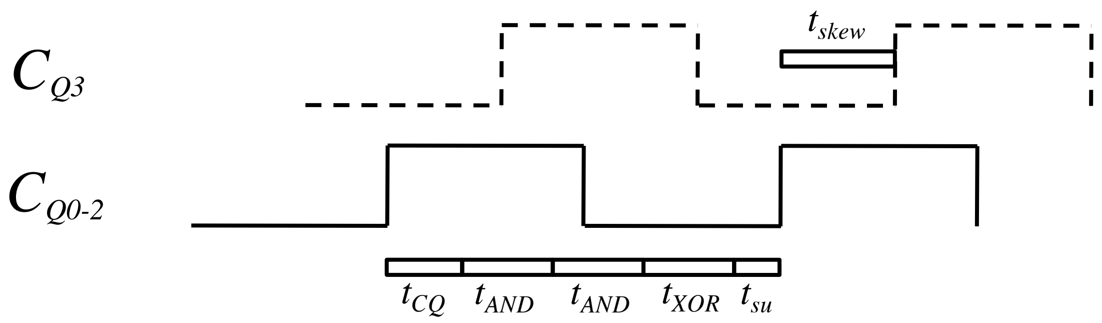
- Assume $t_{skew} = 1.5ns$ delay on clock pulses arriving at $Q3$



- Delay on path from $Q0$ to $Q3$ is then given by

  $t_{cQ} + 3(t_{AND}) + t_{XOR} + t_{su} – t_{skew} = 6.4 – 1.5 = 4.9ns$

  since the skew provides additional time before data is loaded into Q3



- However, critical path is now from $Q0$ to $Q2$, i.e.

  $$T_{min} = t_{cQ} + 2(t_{AND}) + t_{XOR} + t_{su}$$
  $$= 1.0 + 2(1.2) + 1.2 + 0.6$$
  $$= 5.2\ ns$$
  $$F_{max} = 192\ MHz$$

Negative clock skew

- A negative clock skew, i.e. clock arriving <u>earlier</u> at *Q3* than at *Q0* – *Q2* would have the opposite effect of lengthening the clock period requirement & <u>reducing clock frequency</u>

Effect of clock skew on hold times

- As positive clock skew has the effect of delaying the loading of data into FF *Q3*, it has the effect of increasing the hold time requirement of this FF to $t_h + t_{skew}$ for all paths that end at *Q3*
- The shortest such path is from FF *Q2* to *Q3* and has delay $t_{cQ} + t_{AND} + t_{XOR} = 0.8 + 1.2 + 1.2 = 3.2\ ns > t_h + t_{skew} = 1.9\ ns$ $\therefore$ no violation
- But, when $t_{skew} \geq 3.2 - t_h = 2.8\ ns$, then hold time violations will exist and the circuit will not work reliably at any frequency
- Good circuit design therefore aims to minimize, if not eliminate, clock skew