Sequential circuit

Blocking and Non-blocking

- **Blocking Statements**: A blocking statement must be executed before the execution of the statements that follow it in a sequential block.

- **Nonblocking Statements**: Nonblocking statements allow you to schedule assignments without blocking the procedural flow. You can use the nonblocking procedural statement whenever you want to make several register assignments within the same time step without regard to order or dependence upon each other.

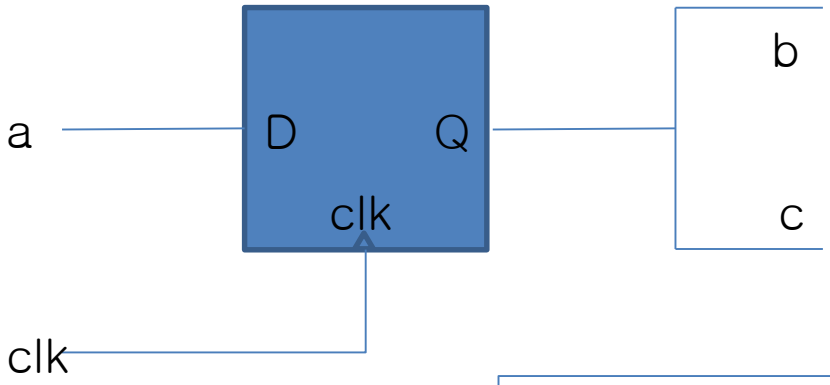
Blocking

```
module blocking;
reg clk, a, b, c;

initial begin
  $dumpfile("blocking.vcd");
  $dumpvars;
            $display("time        a        b        c");
            $monitor("%g          %b       %b       %b",$time, a,b,c);
  a = 0; b = 0; c = 0; clk = 0;
  #5 a = 1;
  #5 a = 0;
  #5 a = 0;
  #5 a = 1;
  #5 $finish;
end
always @ (posedge clk )
begin
 b = a;
 c = b;

end
always #5 clk = !clk;
endmodule
```

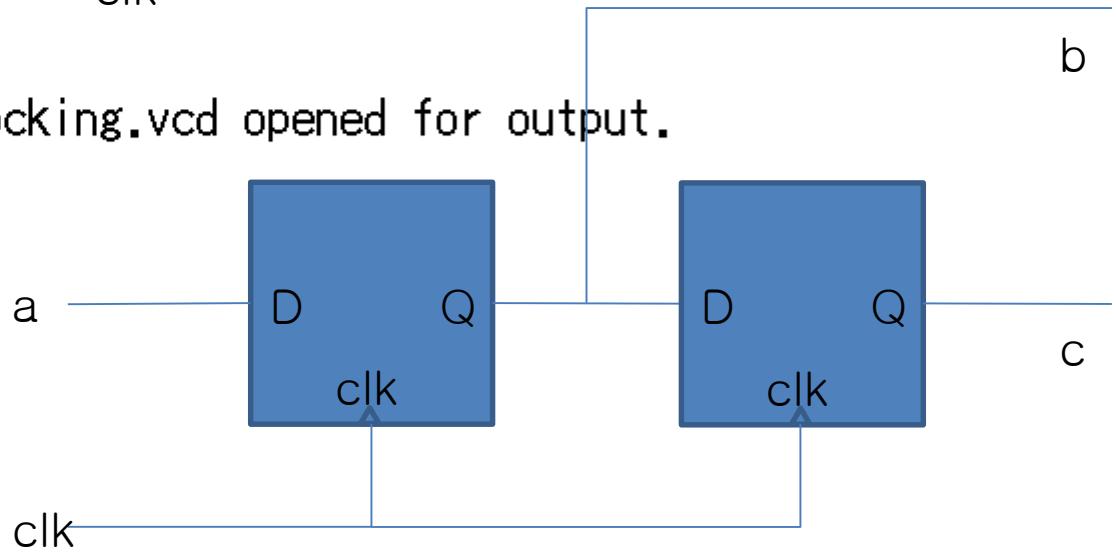
Non-Blocking

```
module blocking;
reg clk, a, b, c;

initial begin
  $dumpfile("blocking.vcd");
  $dumpvars;
            $display("time        a        b        c");
            $monitor("%g          %b       %b       %b",$time, a,b,c);
  a = 0; b = 0; c = 0; clk = 0;
  #5 a = 1;
  #5 a = 0;
  #5 a = 0;
  #5 a = 1;
  #5 $finish;
end
always @ (posedge clk )
begin
 b <= a;
 c <= b;

end
always #5 clk = !clk;
endmodule
```

```
D:¥VHDL>vvp blocking
VCD info: dumpfile blocking.vcd opened for output.
time    a       b       c
0       0       0       0
5       1       1       1
10      0       1       1
15      0       0       0
20      1       0       0
25      1       1       1
```

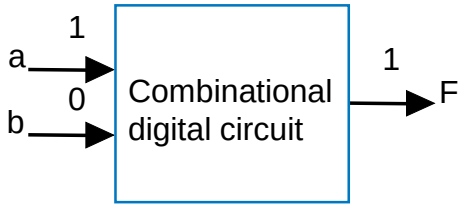


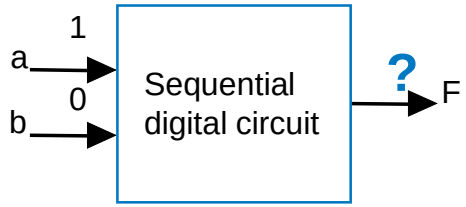

```
D:¥VHDL>vvp non-blocking
VCD info: dumpfile Non-blocking.vcd opened for output.
time    a       b       c
0       0       0       0
5       1       1       0
10      0       1       0
15      0       0       1
20      1       0       1
25      1       1       0
```



Sequential circuit

- Output depends not just on present inputs (as in combinational circuit), but on past sequence of inputs
- Stores bits, also known as having "state"





Must know sequence of past inputs to know output

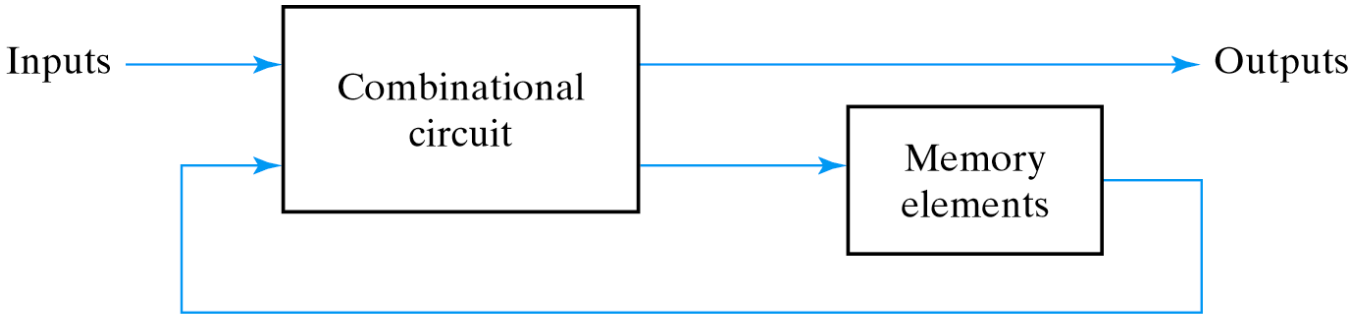
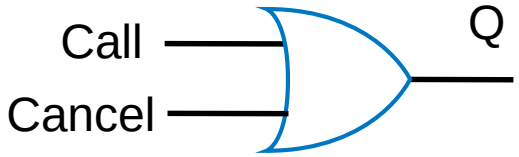


Fig. 5-1 Block Diagram of Sequential Circuit

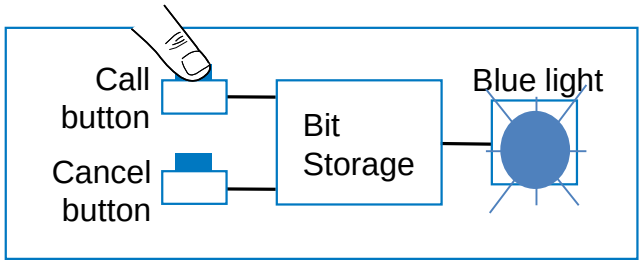
Example Needing Bit Storage

- Flight attendant call button
  - Press call: light turns on
    - *Stays on* after button released
  - Press cancel: light turns off
  - Logic gate circuit to implement this?

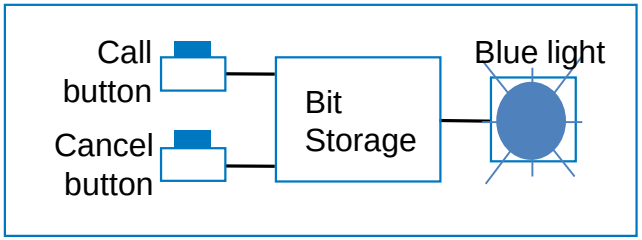


Doesn't work. Q=1 when Call=1, but doesn't stay 1 when Call returns to 0

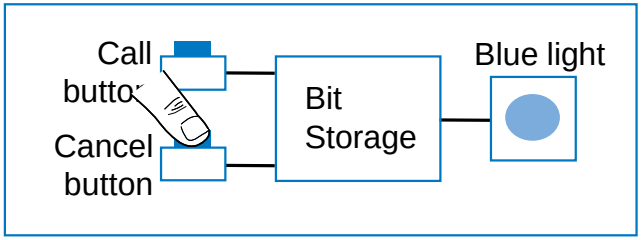
Need some form of "feedback" in the circuit



1. Call button pressed – light turns on



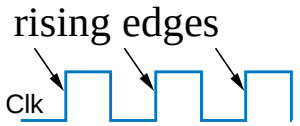
*2. Call button released – light **stays on***



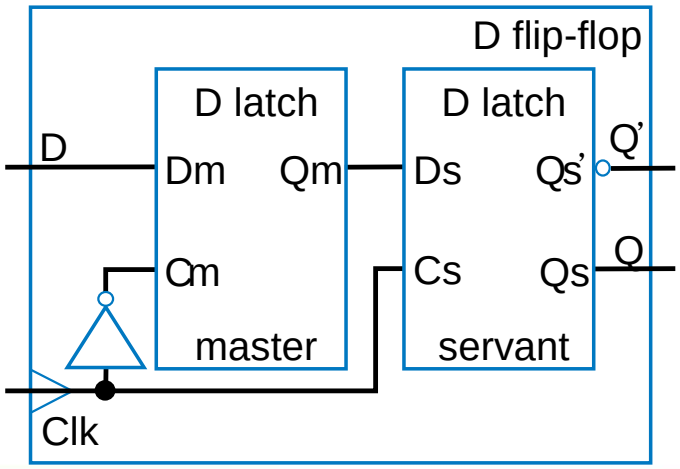
3. Cancel button pressed – light turns off

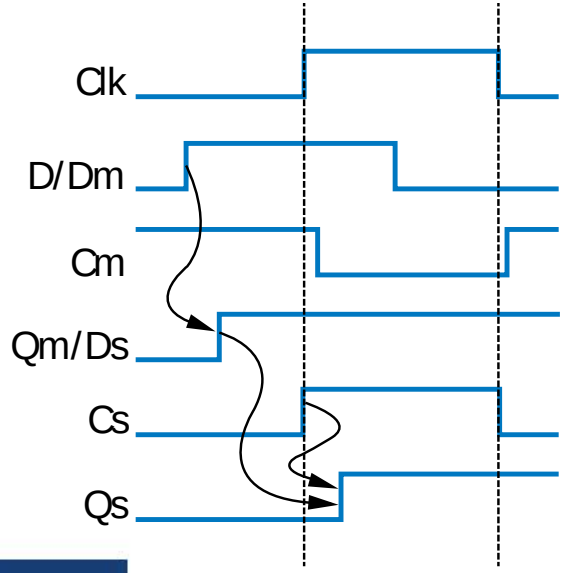
D Flip-Flop

- *Flip-flop*: Bit storage that stores on clock edge, not level
- One design -- master-servant
  - Two latches, output of first goes to input of second, master latch has inverted clock signal
  - So master loaded when C=0, then servant when C=1
  - When C changes from 0 to 1, master disabled, servant loaded with value that was at D just before C changed -- i.e., value at D during rising edge of C

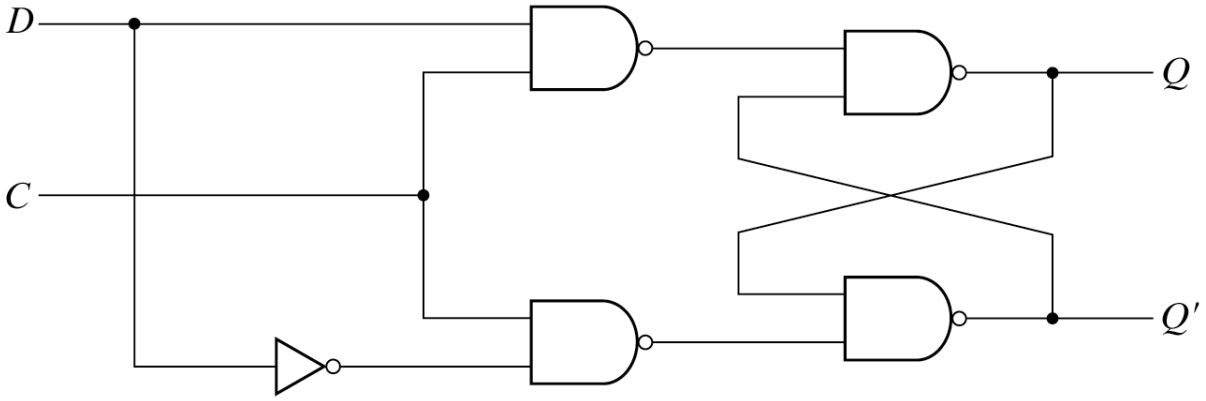


Note: Hundreds of different flip-flop designs exist





D Latch



(a) Logic diagram

C	D	Next state of Q
0	X	No change
1	0	Q = 0; Reset state
1	1	Q = 1; Set state

(b) Function table

Fig. 5-6 D Latch

```
module D_latch (output Q, input D, En);
  reg Q;
  always @ (En, D)
    if (En) Q <= D;
endmodule
```

```
module t_D_latch;
  wire     Q;
  reg      D, En;

 D_latch M0 (Q, D, En);

  initial #100 $finish;
  initial repeat (10) begin #10 D = 1; #10 D = 0; end
  initial fork
          En = 0;
    #5    En = 1;
    #15 En = 0;    // latch the output
    #35 En = 1;
    #45 En = 0;  // latch the outuput
    #75 En = 1;

  join
endmodule
```

D Flip-Flop

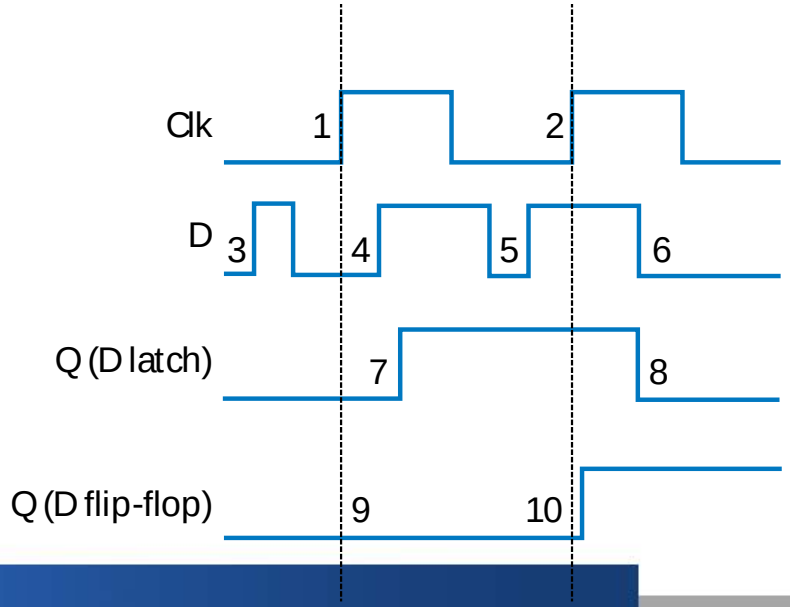
```
//리셋 없는 D플립플롭
module D_flip_flop (Q, D, CLK);
  output Q;
  input          D, CLK;
  reg            Q;

  always @ (posedge CLK)
    Q <= D;
endmodule
```

```
// Description of D flip-flop
// with active-low asynchronous reset

module D_flip_flop_AR (Q, D, CLK, RST);
  output Q;
  input          D, CLK, RST;
  reg            Q;

  always @ (posedge CLK, negedge RST)
    if (RST == 0) Q <= 1'b0;
    else Q <= D;
endmodule
```

```
module t_D_flip_flops;
  wire    Q, Q_AR;
  reg     D, CLK, RST;
  D_flip_flop M0 (Q, D, CLK);
  D_flip_flop_AR M1 (Q_AR, D, CLK, RST);
  initial #100 $finish;
  initial begin CLK = 0; forever #5 CLK = ~CLK; end
  initial fork
    D = 1;
    RST = 1;
    #20 D = 0;
    #40 D = 1;
    #50 D = 0;
    #60 D = 1;
    #70 D = 0;
    #90 D = 1;
    #42 RST = 0;
    #72 RST = 1;
  join
endmodule
```

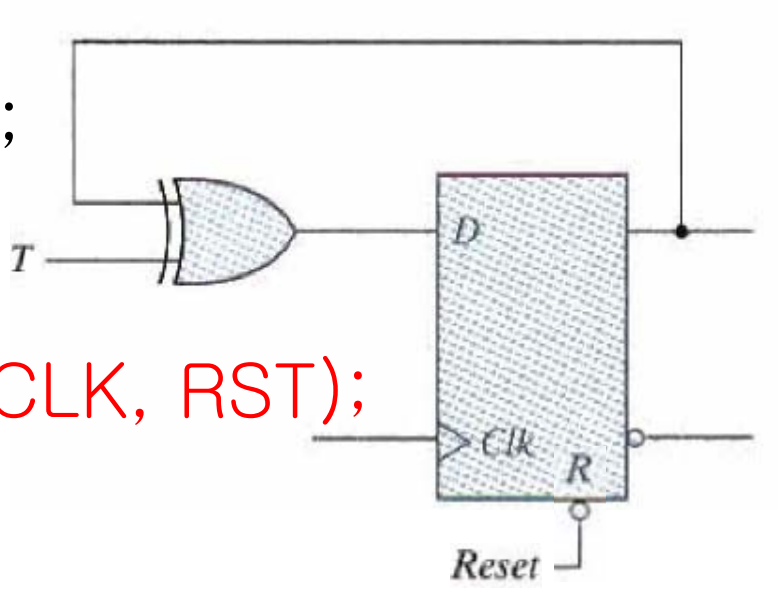
D Latch vs. D Flip-Flop

- Latch is level-sensitive: Stores D when C=1
- Flip-flop is edge triggered: Stores D when C changes from 0 to 1
  - Saying "level-sensitive latch," or "edge-triggered flip-flop," is redundant
  - Two types of flip-flops -- rising or falling edge triggered.
- Comparing behavior of latch and flip-flop:



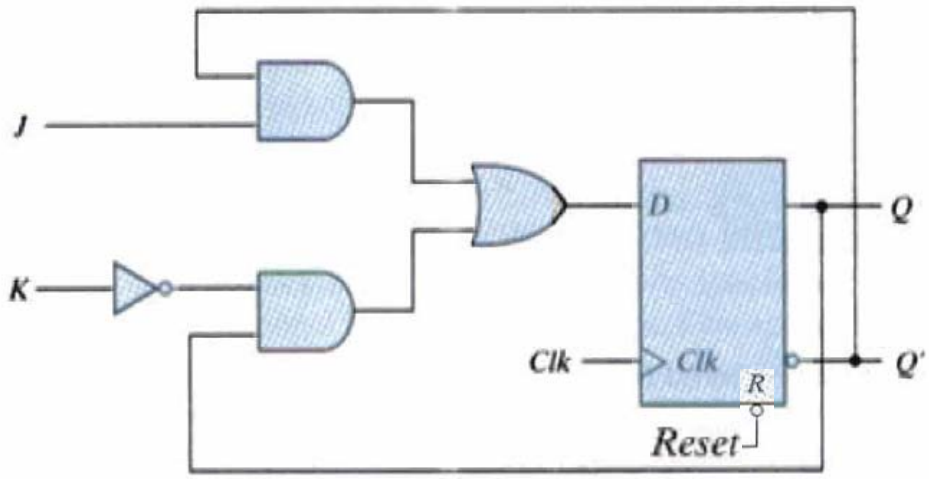
T flip-flop using D flip-flop

```
module Toggle_flip_flop (Q, T, CLK, RST);
  output     Q;
  input              T, CLK, RST;
  wire       DT;
  assign DT = T ^ Q;
  D_flip_flop_AR M0 (Q, DT, CLK, RST);

endmodule
```



JK flip-flop using D flip-flop

```
module JK_flip_flop_1 (Q, Q_not, J, K, CLK, RST_B);
  output    Q, Q_not;
  input     J, K, CLK, RST_B;
  wire      JK;
  assign    JK = (J & ~Q) | (~K & Q);
  assign    Q_not = ~Q;
  D_flip_flop_AR M0 (Q, JK, CLK, RST_B);
endmodule
```



Behavioral Description of JK flip-flop

```
module JK_flip_flop_2 (Q, Q_not, J, K, CLK);
  output      Q, Q_not;
  input       J, K, CLK;
  reg         Q;
  assign      Q_not = ~Q;
  always @ (posedge CLK)
    case ({J, K})
      2'b00:   Q <= Q;
      2'b01:   Q <= 1'b0;
      2'b10:   Q <= 1'b1;
      2'b11:   Q <= ~Q;
    endcase
endmodule
```

t_Toggle_flip_flop

```
module t_Toggle_flip_flop;
  wire  t_Q;
  reg     t_T, t_CLK, t_RST;
  Toggle_flip_flop M1  (t_Q, t_T, t_CLK, t_RST);

initial #150 $finish;
  initial begin t_CLK = 0; forever #5 t_CLK =  ~t_CLK; end
  initial fork
    t_T = 1;
    t_RST = 0;
    #22 t_RST = 1;
     #42 t_RST = 0;
    #62 t_RST = 1;
    #90 t_T = 0;
    #120 t_T = 1;
  join
endmodule
```

t_JK_flip_flop

```
module t_JK_flip_flop;
  wire     Q, Q_not;
  reg      J, K, CLK, RST_B;

  JK_flip_flop_1  M1 (Q_1, Q_1_not, J, K, CLK, RST_B);
  JK_flip_flop_2  M2 (Q_2, Q_2_not, J, K, CLK);

  initial #100 $finish;
  initial begin CLK = 0; forever #5 CLK = ~CLK; end
  initial fork
    RST_B = 0;
    RST_B = 1;
    J = 0;
    K = 0;
    #20 K = 1;
    #40 J = 1;
    #80 K = 0;
    #90 J = 0;
   join
endmodule
```