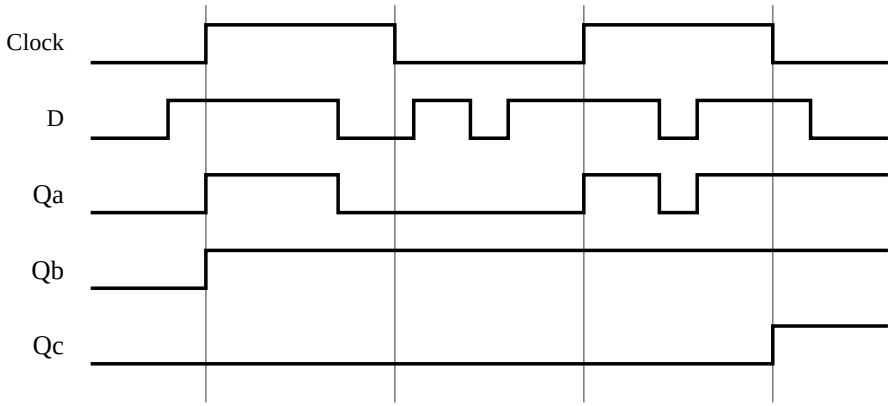
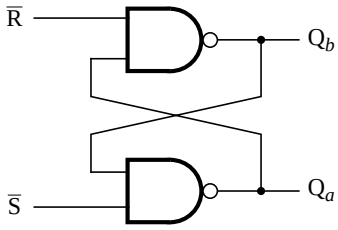
Chapter 7

7.1.



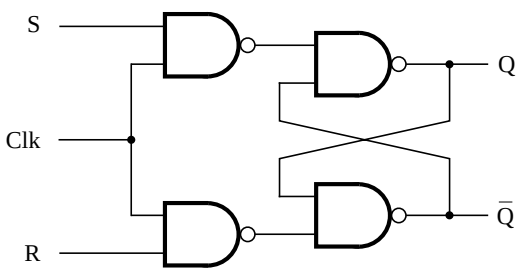
7.2. The circuit in Figure 7.3 can be modified to implement an SR latch by connecting S to the *Data* input and $S + R$ to the *Load* input. Thus the value of S is loaded into the latch whenever either S or R is asserted. Care must be taken to ensure that the *Data* signal remains stable while the *Load* signal is asserted.

7.3.

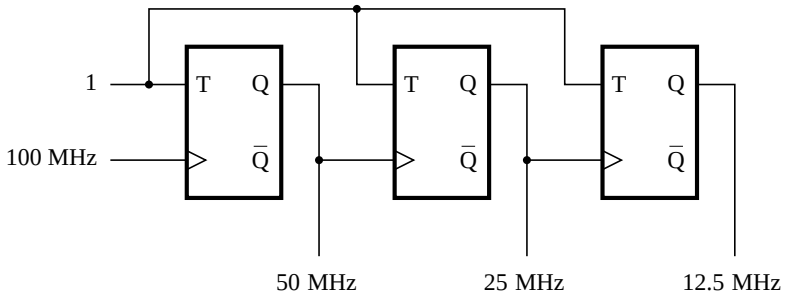


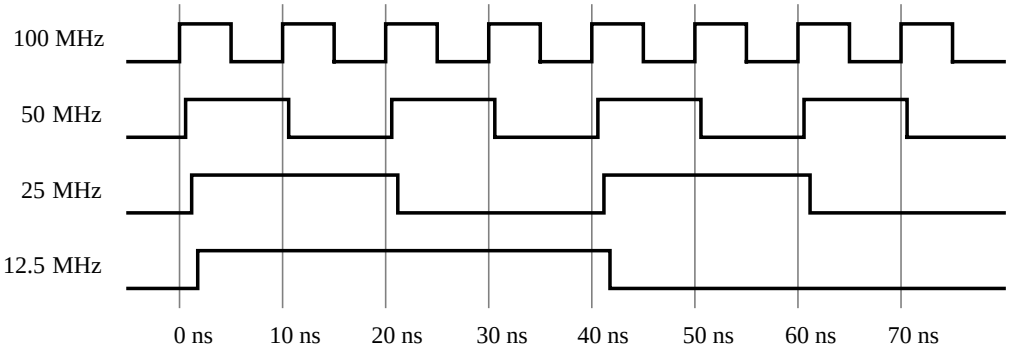
$\overline{S}$	$\overline{R}$	Q_a	Q_b	
1	1	0/1	1/0	(no change)
1	0	0	1	
0	1	1	0	
0	0	1	1	

7.4.

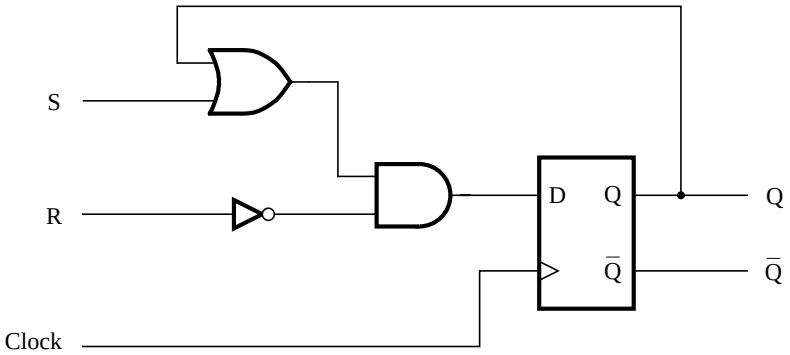


7.5.

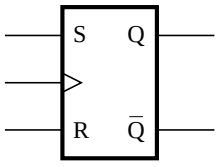




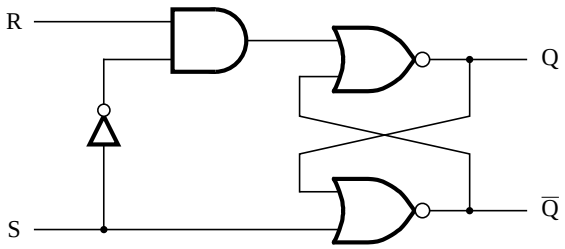
7.6.



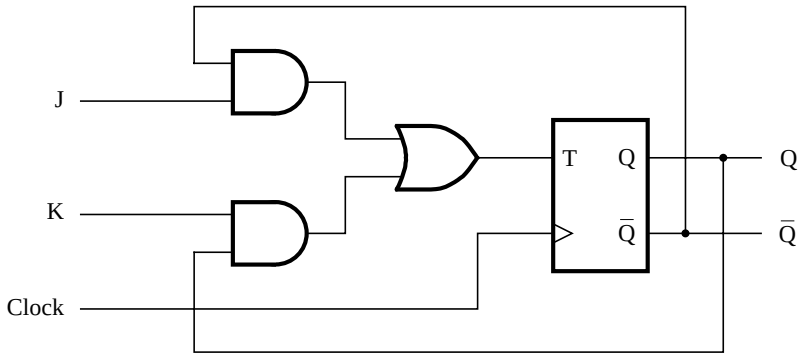
S	R	Q(t + 1)
0	0	Q(t)
0	1	0
1	0	1
1	1	0



7.7.



7.8.



7.9. This circuit acts as a negative-edge-triggered JK flip-flop, in which $J = A, K = B$, *Clock* $= C, \mathrm{Q} = D$, and $\overline{\mathrm{Q}} = E$. This circuit is found in the standard chip called 74LS107A (plus a *Clear* input, which is not shown).

7.10.

```
module tflipflop (T, Clock, Resetn, Q);
    input T, Clock, Resetn;
    output Q;
    reg Q;

    always @(negedge Resetn or posedge Clock)
        if (!Resetn)
            Q <= 0;
        else if (T)
            Q <=  Q;

endmodule
```

7.11.

```
module jkflipflop (J, K, Clock, Resetn, Q);
    input J, K, Clock, Resetn;
    output Q;
    reg Q;

    always @(negedge Resetn or posedge Clock)
        if (!Resetn)
            Q <= 0;
        else
            case (J, K)
                1'b01:    Q <= 0;
                1'b10:    Q <= 1;
                1'b11:    Q <=  Q;
                default:  Q <= Q;
            endcase

endmodule
```

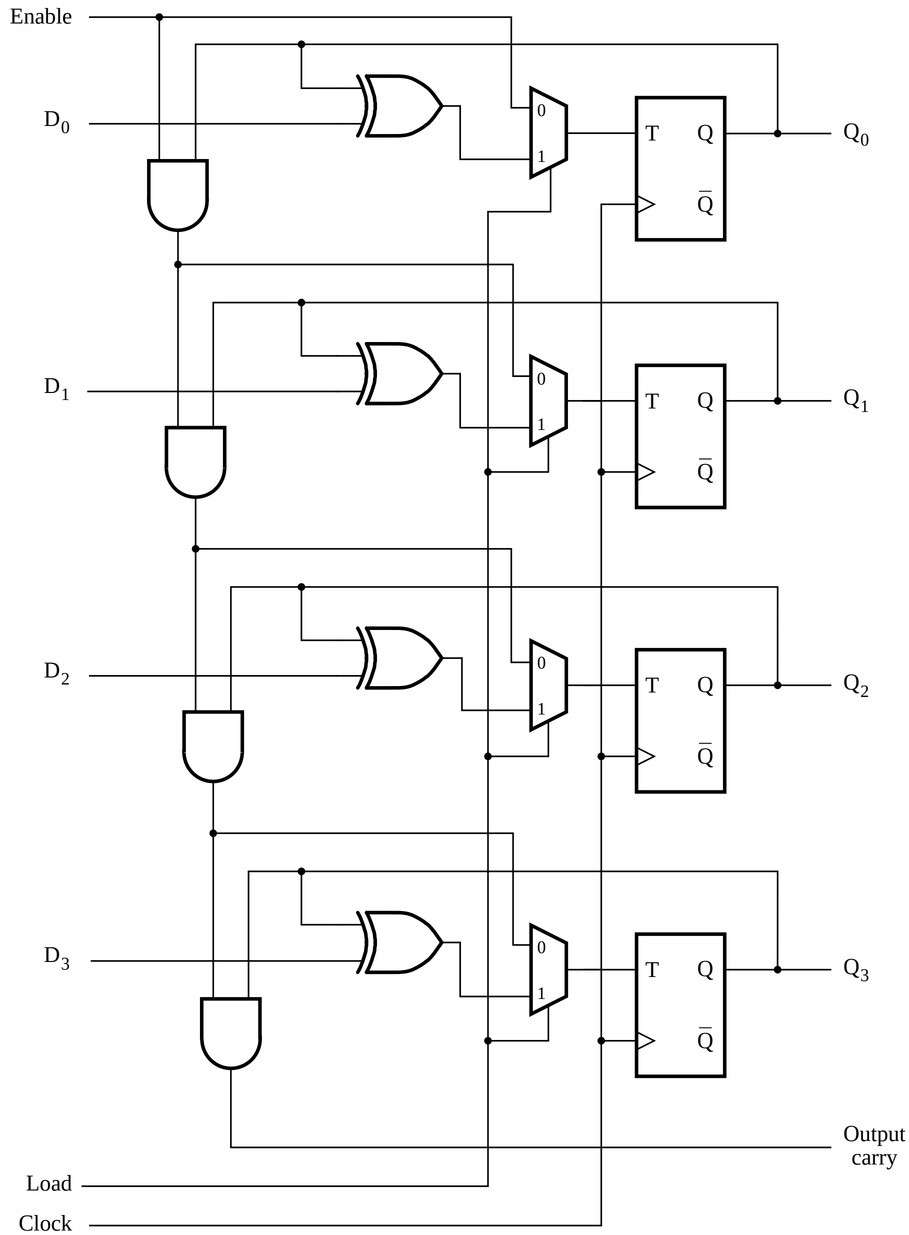
7.13. Let $S = s_1 s_0$ be a binary number that specifies the number of bit-positions by which to rotate. Also let L be a parallel-load input, and let $R = r_0 r_1 r_2 r_3$ be parallel data. If the inputs to the flip-flops are $d_0 \ldots d_3$ and the outputs are $q_0 \ldots q_3$, then the barrel-shifter can be represented by the logic expressions

$$\begin{aligned}
d_0 &= L \cdot r_0 + \overline{L} \cdot (\overline{s}_1 \overline{s}_0 q_0 + \overline{s}_1 s_0 q_3 + s_1 \overline{s}_0 q_2 + s_1 s_0 q_1) \\
d_1 &= L \cdot r_1 + \overline{L} \cdot (\overline{s}_1 \overline{s}_0 q_1 + \overline{s}_1 s_0 q_0 + s_1 \overline{s}_0 q_3 + s_1 s_0 q_2) \\
d_2 &= L \cdot r_2 + \overline{L} \cdot (\overline{s}_1 \overline{s}_0 q_2 + \overline{s}_1 s_0 q_1 + s_1 \overline{s}_0 q_0 + s_1 s_0 q_3) \\
d_3 &= L \cdot r_3 + \overline{L} \cdot (\overline{s}_1 \overline{s}_0 q_3 + \overline{s}_1 s_0 q_2 + s_1 \overline{s}_0 q_1 + s_1 s_0 q_0)
\end{aligned}$$

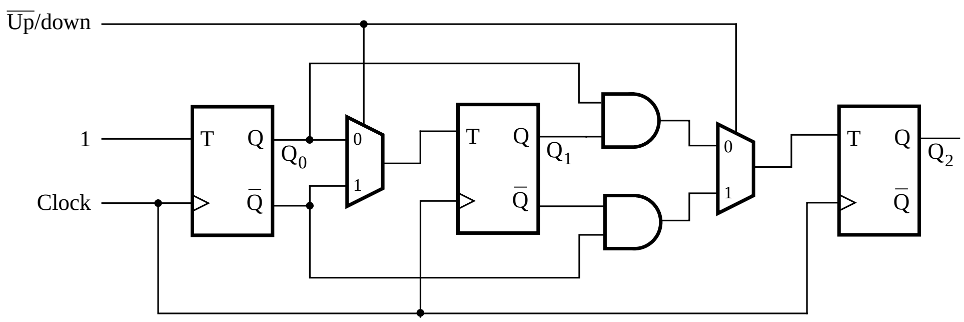
7.14. There are many ways to write the Verilog code for this problem. One solution is

```
// Barrel shifter. If L = 1, load in parallel from R. If L = 0 and E =1
// rotate right by number of bit positions given by S.
module barrel4 (R, L, S, Clock, Q);
    input [0:3] R;
    input L, Clock;
    input [1:0] S;
    output [0:3] Q;
    reg [0:3] Q;
    wire [0:3] M;

    mux4to1 Bit0 (Q[0], Q[3], Q[2], Q[1], S, M[0]);
    mux4to1 Bit1 (Q[1], Q[0], Q[3], Q[2], S, M[1]);
    mux4to1 Bit2 (Q[2], Q[1], Q[0], Q[3], S, M[2]);
    mux4to1 Bit3 (Q[3], Q[2], Q[1], Q[0], S, M[3]);

    always @(posedge Clock)
        if (L)
            Q <= R;
        else
        begin
            Q[0] <= M[0];
            Q[1] <= M[1];
            Q[2] <= M[2];
            Q[3] <= M[3];
        end

endmodule

module mux4to1 (w0, w1, w2, w3, S, f);
    input w0, w1, w2, w3;
    input [1:0] S;
    output f;
    reg f;

    always @(w0 or w1 or w2 or w3 or S)
        if (S == 2'b00)
            f = w0;
        else if (S == 2'b01)
            f = w1;
        else if (S == 2'b10)
            f = w2;
        else if (S == 2'b11)
            f = w3;

endmodule
```

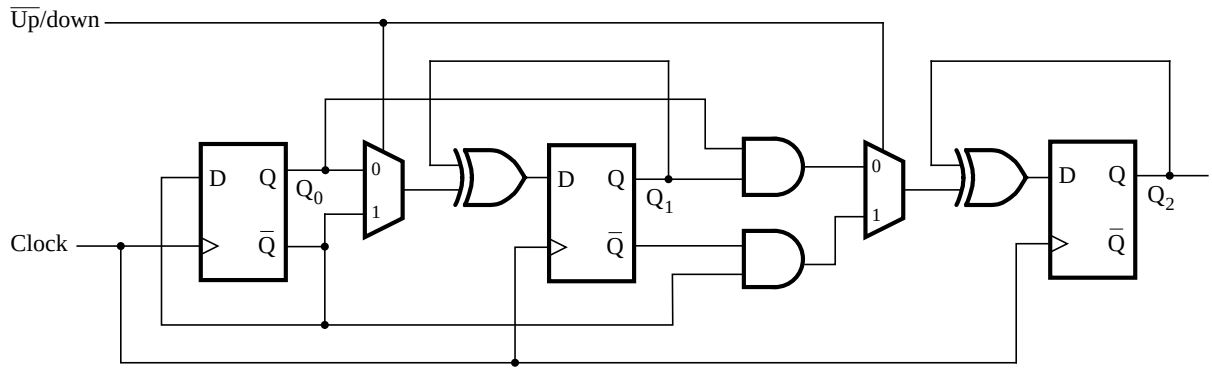
7.15.


Enable
D0
Q0
T
Q
Q̄
0
1
D1
Q1
D2
Q2
D3
Q3
Output carry
Load
Clock

7.16.


Up/down
1
Clock
T
Q
Q̄
Q0
Q1
Q2
0
1

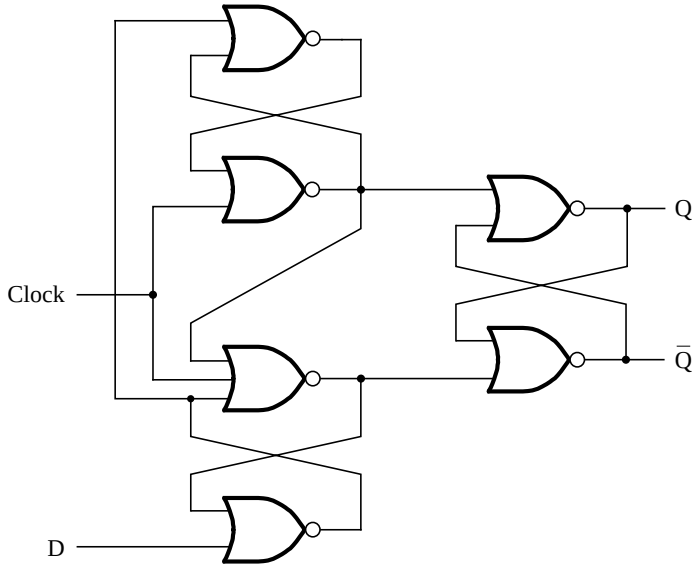
7.17.



7.18. The counting sequence is $000, 001, 010, 111$.

7.19. The circuit in Figure P7.4 is a master-slave JK flip-flop. It suffers from a problem sometimes called *ones-catching*. Consider the situation where the Q output is low, *Clock* = 0, and $J = K = 0$. Now let *Clock* remain stable at 0 while *J* change from 0 to 1 and then back to 0. The master stage is now set to 1 and this value will be incorrectly transferred into the slave stage when the clock changes to 1.

7.20. Repeated application of DeMorgan's theorem can be used to change the positive-edge triggered D flip-flop in Figure 7.11 into the negative-edge D triggered flip-flop:



7.21.

```
module upcount12 (Resetn, Clock, Q);
    input Resetn, Clock;
    output [3:0] Q;
    reg [3:0] Q;

    always @(posedge Clock)
        if (!Resetn)
            Q <= 0;
        else if (Q == 11)
            Q <= 0;
        else
            Q <= Q + 1;
endmodule
```

7.22. The longest delay in the circuit is the from the output of FF_0 to the input of FF_3. This delay totals 5 ns. Thus the minimum period for which the circuit will operate reliably is

$$T_{min} = 5 \text{ ns} + t_{su} = 8 \text{ ns}$$

The maximum frequency is

$$F_{max} = 1/T_{min} = 125 \text{ MHz}$$

7.23.

```
module johnson8 (Resetn, Clock, Q);
    input Resetn, Clock;
    output [7:0] Q;
    reg [7:0] Q;

    always @(negedge Resetn or posedge Clock)
        if (!Resetn)
            Q <= 0;
        else
            Q <= {{Q[6:0]}, {~Q[7]}};
endmodule
```

7.24.

```
// Ring counter with synchronous reset
module ripplen (Resetn, Clock, Q);
    parameter n = 8;
    input Resetn, Clock;
    output [n−1:0] Q;
    reg [n−1:0] Q;

    always @(posedge Clock)
        if (!Resetn)
        begin
            Q[7:1] <= 0;
            Q[0] <= 1;
        end
        else
            Q <= {{Q[6:0]}, {Q[7]}};
endmodule
```

7.25.

```
module accumulate(Reset, Clock, Data, Q);
    input Reset, Clock;
    input [3:0] Data;
    output [3:0] Q;
    reg [3:0] Q;

    always @(posedge Reset or posedge Clock)
        if (Reset)
            Q <= 0;
        else
            Q <= Q + Data;
endmodule
```

7.26.

```
module count32 (Clock, Reset, Q);
    input Clock, Reset;
    output [31:0] Q ;

    lpm_counter count_up (.aclr(Reset), .clock(Clock), .q(Q)) ;
        defparam count_up.lpm_width = 32;
endmodule
```

7.30.

	T_1	T_2	T_3
(Swap): I_4	$R_{out} = X, T_{in}$	$R_{out} = Y, R_{in} = X$	$T_{out}, R_{in} = Y,$ *Done*

Since the processor now has five operations a 3-to-8 decoder is needed to decode the signals f_2, f_1, f_0. The SWAP operation is represented by the code

$$I_4 = f_2\overline{f}_1\overline{f}_0$$

New expressions are needed for R_{in} and R_{out} to accommodate the SWAP operation:

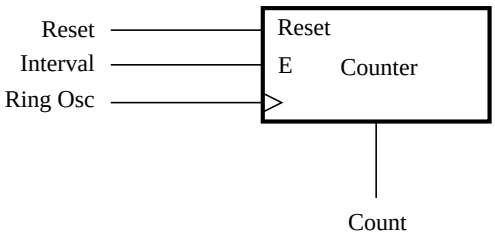
$$\begin{aligned} Rk_{in} &= (I_0 + I_1) \cdot T_1 \cdot X_k + (I_2 + I_3) \cdot T_3 \cdot X_k + I_4 \cdot T_2 \cdot X_k + I_4 \cdot T_3 \cdot Y_k \\ Rk_{out} &= I_1 \cdot T_1 \cdot Y_k + (I_2 + I_3) \cdot (T_1 X_k + T_2 Y_k) + I_4 \cdot T_1 X_k + I_4 \cdot T_2 Y_k \end{aligned}$$

The control signals for the temporary register, T, are

$$\begin{aligned} T_{in} &= T_1 I_4 \\ T_{out} &= T_3 I_4 \end{aligned}$$

7.31. (a) $\text{Period} = 2 \times n \times t_p$

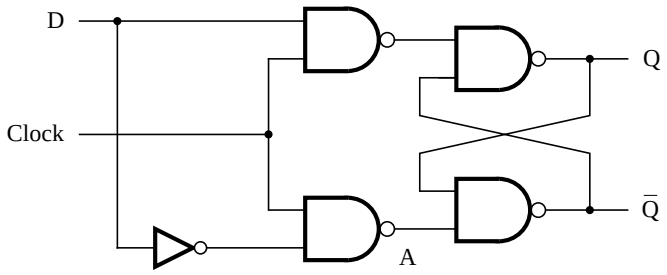
(b)

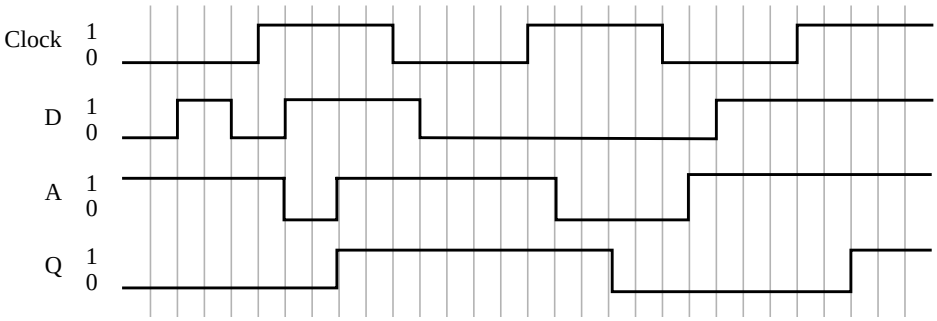


The counter tallies the number of pulses in the 100 ns time period. Thus

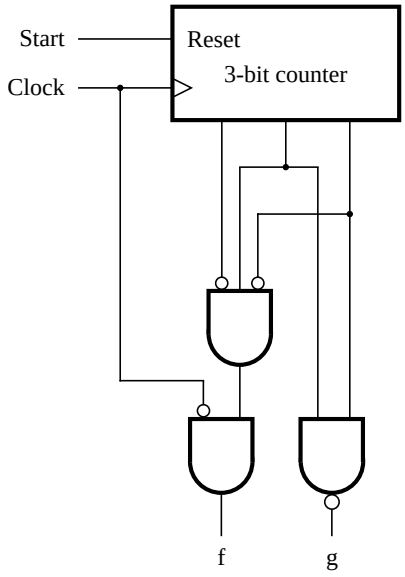
$$t_p = \frac{100 \text{ ns}}{2 \times Count \times n}$$

7.32.



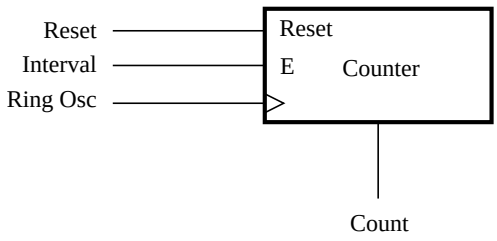


7.33.



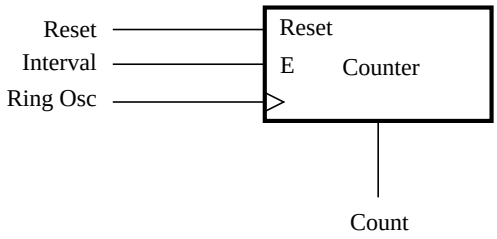
7.34. With non-blocking assignments, the result of the assignment f <= A[1] & A[0] is not seen by the successive assignments inside the **for** loop. Thus, f has an uninitialized value when the **for** loop is entered. Similarly, each **for** loop interation sees the unitialized value of f. The result of the code is the sequential circuit specified by f = f | A[n-1] A[n-2].

7.35.



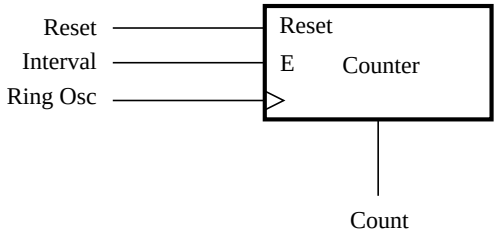
The counting sequence is: 001, 110, 011, 111, 101, 100, 010, 001

7.36.



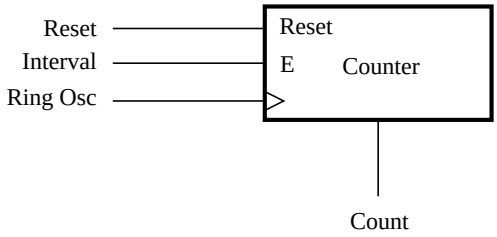
The counting sequence is: 001, 101, 111, 110, 011, 100, 010, 001

7.37.



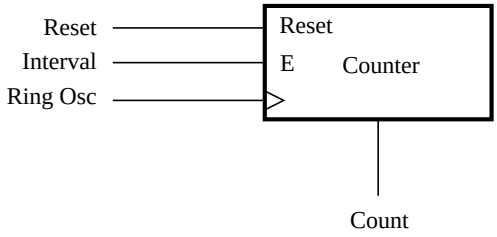
The counting sequence is: 001, 100, 000, 000, ...

7.38.



The counting sequence is: 001, 110, 000, 000, ...

7.39.



7.40.

```
// Universal shift register. If Dir = 0 shifting is to the left.
module universaln (R, L, Dir, w0, w1, Clock, Q);
    parameter n = 4;
    input [n−1:0] R;
    input L, Dir, w0, w1, Clock;
    output [n−1:0] Q;
    reg [n−1:0] Q;
    integer k;

    always @(posedge Clock)
        if (L)
            Q <= R;
        else
        begin
            if (Dir)
            begin
                for (k = 0; k < n−1; k = k+1)
                    Q[k] <= Q[k+1];
                Q[n−1] <= w0;
            end
            else
            begin
                Q[0] <= w1;
                for (k = n−1; k > 0; k = k−1)
                    Q[k] <= Q[k−1];
            end
        end

endmodule
```