Chapter 2

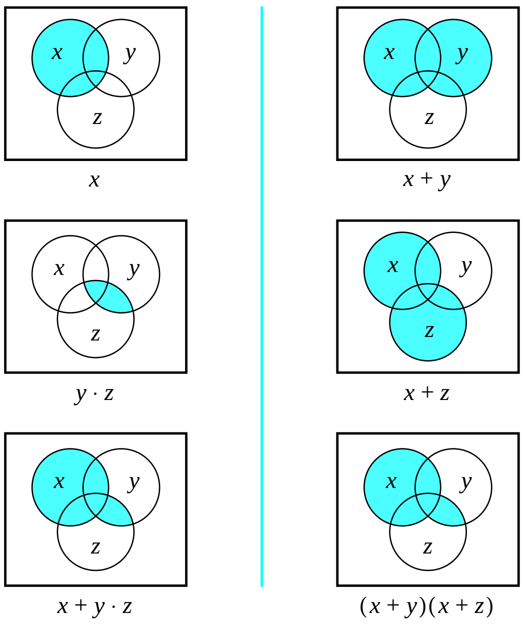
2.1. The proof is as follows:

$$\begin{aligned}
(x+y)\cdot(x+z) &= xx + xz + xy + yz \\
&= x + xz + xy + yz \\
&= x(1 + z + y) + yz \\
&= x \cdot 1 + yz \\
&= x + yz
\end{aligned}$$

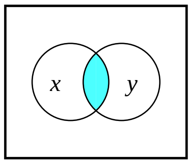
2.2. The proof is as follows:

$$\begin{aligned}
(x+y)\cdot(x+\overline{y}) &= xx + xy + x\overline{y} + y\overline{y} \\
&= x + xy + x\overline{y} + 0 \\
&= x(1 + y + \overline{y}) \\
&= x \cdot 1 \\
&= x
\end{aligned}$$

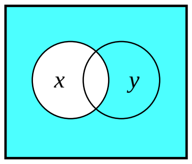
2.3. Proof using Venn diagrams:



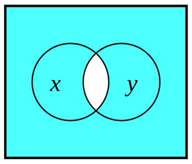
2.4. Proof of 15a using Venn diagrams:



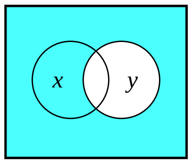
$x \cdot y$



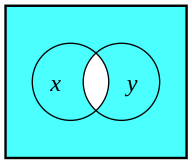
$\bar{x}$



$\overline{x \cdot y}$



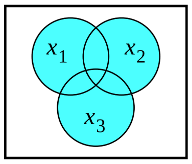
$\bar{y}$



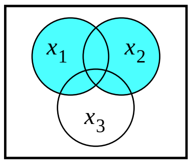
$\bar{x} \cdot \bar{y}$

A similar proof is constructed for 15b.

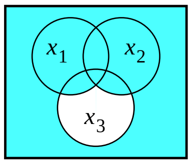
2.5. Proof using Venn diagrams:



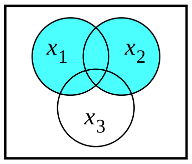
$x_1 + x_2 + x_3$



$x_1 + x_2$



$x_1 + x_2 + \bar{x}_3$

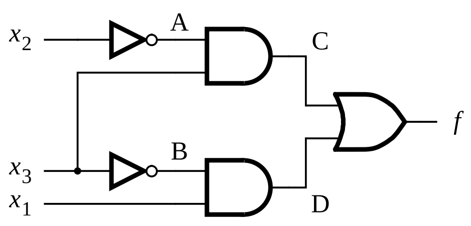


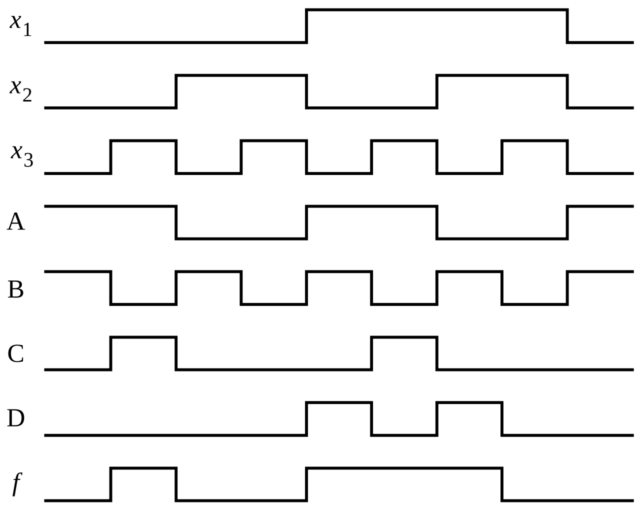
$(x_1 + x_2 + x_3) \cdot (x_1 + x_2 + \bar{x}_3)$

2.6. A possible approach for determining whether or not the expressions are valid is to try to manipulate the left and right sides of an expression into the same form, using the theorems and properties presented in section 2.5. While this may seem simple, it is an awkward approach, because it is not obvious what target form one should try to reach. A much simpler approach is to construct a truth table for each side of an expression. If the truth tables are identical, then the expression is valid. Using this approach, we can show that the answers are:

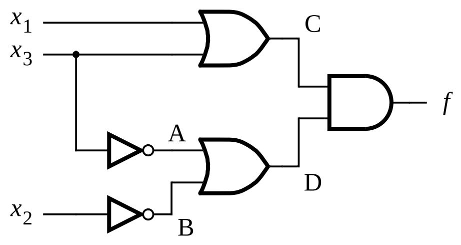
(a) Yes
(b) Yes
(c) No

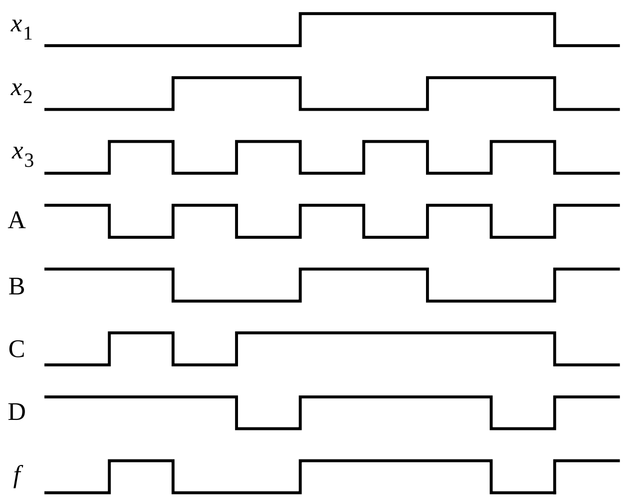
2.7. Timing diagram of the waveforms that can be observed on all wires of the circuit:





2.8. Timing diagram of the waveforms that can be observed on all wires of the circuit:





2.9. Starting with the canonical sum-of-products for f get

$$\begin{aligned}
f &= \overline{x}_1\overline{x}_2x_3 + \overline{x}_1x_2\overline{x}_3 + \overline{x}_1x_2x_3 + x_1\overline{x}_2\overline{x}_3 + x_1\overline{x}_2x_3 + x_1x_2\overline{x}_3 + x_1x_2x_3 \\
&= x_1(\overline{x}_2\overline{x}_3 + \overline{x}_2x_3 + x_2\overline{x}_3 + x_2x_3) + x_2(\overline{x}_1\overline{x}_3 + \overline{x}_1x_3 + x_1\overline{x}_3 + x_1x_3) \\
&\quad + x_3(\overline{x}_1\overline{x}_2 + \overline{x}_1x_2 + x_1\overline{x}_2 + x_1x_2) \\
&= x_1(\overline{x}_2(\overline{x}_3 + x_3) + x_2(\overline{x}_3 + x_3)) + x_2(\overline{x}_1(\overline{x}_3 + x_3) + x_1(\overline{x}_3 + x_3)) \\
&\quad + x_3(\overline{x}_1(\overline{x}_2 + x_2) + x_1(\overline{x}_2 + x_2)) \\
&= x_1(\overline{x}_2 \cdot 1 + x_2 \cdot 1) + x_2(\overline{x}_1 \cdot 1 + x_1 \cdot 1) + x_3(\overline{x}_1 \cdot 1 + x_1 \cdot 1) \\
&= x_1(\overline{x}_2 + x_2) + x_2(\overline{x}_1 + x_1) + x_3(\overline{x}_1 + x_1) \\
&= x_1 \cdot 1 + x_2 \cdot 1 + x_3 \cdot 1 \\
&= x_1 + x_2 + x_3
\end{aligned}$$

2.10. Starting with the canonical product-of-sums for f can derive:

$$\begin{aligned}
f &= (x_1 + x_2 + x_3)(x_1 + x_2 + \overline{x}_3)(x_1 + \overline{x}_2 + x_3)(x_1 + \overline{x}_2 + \overline{x}_3) \cdot \\
&\quad (\overline{x}_1 + x_2 + x_3)(\overline{x}_1 + x_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_2 + x_3) \\
&= ((x_1 + x_2 + x_3)(x_1 + x_2 + \overline{x}_3))((x_1 + \overline{x}_2 + x_3)(x_1 + \overline{x}_2 + \overline{x}_3)) \cdot \\
&\quad ((\overline{x}_1 + x_2 + x_3)(\overline{x}_1 + x_2 + \overline{x}_3))((\overline{x}_1 + \overline{x}_2 + x_3)(\overline{x}_1 + x_2 + x_3)) \\
&= (x_1 + x_2 + x_3\overline{x}_3)(x_1 + \overline{x}_2 + x_3\overline{x}_3) \cdot \\
&\quad (\overline{x}_1 + x_2 + x_3\overline{x}_3)(\overline{x}_1 + \overline{x}_2x_2 + x_3) \\
&= (x_1 + x_2)(x_1 + \overline{x}_2)(\overline{x}_1 + x_2)(\overline{x}_1 + x_3)
\end{aligned}$$

$$\begin{aligned}
&= (x_1 + x_2\overline{x}_2)(\overline{x}_1 + x_2x_3) \\
&= x_1(\overline{x}_1 + x_2x_3) \\
&= x_1\overline{x}_1 + x_1x_2x_3 \\
&= x_1x_2x_3
\end{aligned}$$

2.11. Derivation of the minimum sum-of-products expression:

$$\begin{aligned}
f &= x_1x_3 + x_1\overline{x}_2 + \overline{x}_1x_2x_3 + \overline{x}_1\overline{x}_2\overline{x}_3 \\
&= x_1(\overline{x}_2 + x_2)x_3 + x_1\overline{x}_2(\overline{x}_3 + x_3) + \overline{x}_1x_2x_3 + \overline{x}_1\overline{x}_2\overline{x}_3 \\
&= x_1\overline{x}_2x_3 + x_1x_2x_3 + x_1\overline{x}_2\overline{x}_3 + \overline{x}_1x_2x_3 + \overline{x}_1\overline{x}_2\overline{x}_3 \\
&= x_1x_3 + (x_1 + \overline{x}_1)x_2x_3 + (x_1 + \overline{x}_1)\overline{x}_2\overline{x}_3 \\
&= x_1x_3 + x_2x_3 + \overline{x}_2\overline{x}_3
\end{aligned}$$

2.12. Derivation of the minimum sum-of-products expression:

$$\begin{aligned}
f &= x_1\overline{x}_2\overline{x}_3 + x_1x_2x_4 + x_1\overline{x}_2x_3\overline{x}_4 \\
&= x_1\overline{x}_2\overline{x}_3(\overline{x}_4 + x_4) + x_1x_2x_4 + x_1\overline{x}_2x_3\overline{x}_4 \\
&= x_1\overline{x}_2\overline{x}_3\overline{x}_4 + x_1\overline{x}_2\overline{x}_3x_4 + x_1x_2x_4 + x_1\overline{x}_2x_3\overline{x}_4 \\
&= x_1\overline{x}_2\overline{x}_3 + x_1\overline{x}_2(\overline{x}_3 + x_3)\overline{x}_4 + x_1x_2x_4 \\
&= x_1\overline{x}_2\overline{x}_3 + x_1\overline{x}_2\overline{x}_4 + x_1x_2x_4
\end{aligned}$$

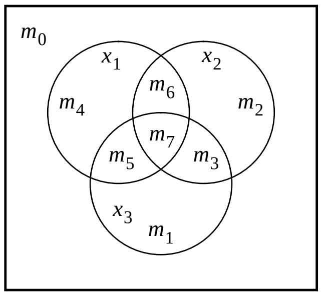
2.13. The simplest POS expression is derived as

$$\begin{aligned}
f &= (x_1 + x_3 + x_4)(x_1 + \overline{x}_2 + x_3)(x_1 + \overline{x}_2 + \overline{x}_3 + x_4) \\
&= (x_1 + x_3 + x_4)(x_1 + \overline{x}_2 + x_3)(x_1 + \overline{x}_2 + x_3 + x_4)(x_1 + \overline{x}_2 + \overline{x}_3 + x_4) \\
&= (x_1 + x_3 + x_4)(x_1 + \overline{x}_2 + x_3)((x_1 + \overline{x}_2 + x_4)(x_3 + \overline{x}_3)) \\
&= (x_1 + x_3 + x_4)(x_1 + \overline{x}_2 + x_3)(x_1 + \overline{x}_2 + x_4) \cdot 1 \\
&= (x_1 + x_3 + x_4)(x_1 + \overline{x}_2 + x_3)(x_1 + \overline{x}_2 + x_4)
\end{aligned}$$

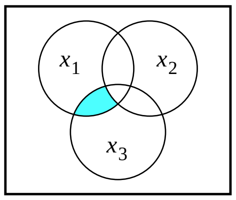
2.14. Derivation of the minimum product-of-sums expression:

$$\begin{aligned}
f &= (x_1 + x_2 + x_3)(x_1 + \overline{x}_2 + x_3)(\overline{x}_1 + \overline{x}_2 + x_3)(x_1 + x_2 + \overline{x}_3) \\
&= ((x_1 + x_2) + x_3)((x_1 + x_2) + \overline{x}_3)(x_1 + (\overline{x}_2 + x_3))(\overline{x}_1 + (\overline{x}_2 + x_3)) \\
&= (x_1 + x_2)(\overline{x}_2 + x_3)
\end{aligned}$$

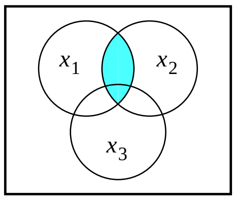
2.15. (a) Location of all minterms in a 3-variable Venn diagram:



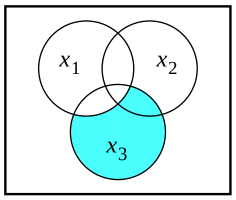
(b) For $f = x_1\overline{x}_2x_3 + x_1x_2 + \overline{x}_1x_3$ have:



$x_1 \cdot \bar{x}_2 \cdot x_3$

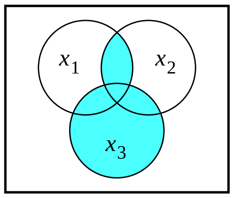


$x_1 \cdot x_2$



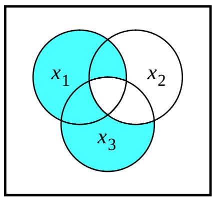
$\bar{x}_1 \cdot x_3$

Therefore, f is represented as:



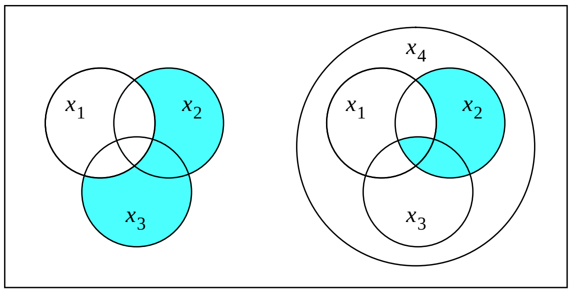
$$f = x_3 + x_1x_2$$

2.16. The function in Figure 2.18 in Venn diagram form is:



2.17. In Figure P2.1a it is possible to represent only 14 minterms. It is impossible to represent the minterms $\overline{x}_1\overline{x}_2x_3x_4$ and $x_1x_2\overline{x}_3\overline{x}_4$.
In Figure P2.1b, it is impossible to represent the minterms $x_1x_2\overline{x}_3\overline{x}_4$ and $x_1x_2x_3\overline{x}_4$.

2.18. Venn diagram for $f = \overline{x}_1\overline{x}_2x_3\overline{x}_4 + x_1x_2x_3x_4 + \overline{x}_1x_2$ is



2.19. The simplest SOP implementation of the function is

$$\begin{aligned} f &= \overline{x}_1x_2x_3 + x_1\overline{x}_2\overline{x}_3 + x_1x_2\overline{x}_3 + x_1x_2x_3 \\ &= (\overline{x}_1 + x_1)x_2x_3 + x_1(\overline{x}_2 + x_2)\overline{x}_3 \\ &= x_2x_3 + x_1\overline{x}_3 \end{aligned}$$

2.20. The simplest SOP implementation of the function is

$$\begin{aligned} f &= \overline{x}_1\overline{x}_2x_3 + \overline{x}_1x_2x_3 + x_1\overline{x}_2\overline{x}_3 + x_1x_2\overline{x}_3 + x_1x_2x_3 \\ &= \overline{x}_1(\overline{x}_2 + x_2)x_3 + x_1(\overline{x}_2 + x_2)\overline{x}_3 + (\overline{x}_1 + x_1)x_2x_3 \\ &= \overline{x}_1x_3 + x_1\overline{x}_3 + x_2x_3 \end{aligned}$$

Another possibility is

$$f = \overline{x}_1x_3 + x_1\overline{x}_3 + x_1x_2$$

2.21. The simplest POS implementation of the function is

$$\begin{aligned} f &= (x_1 + x_2 + x_3)(x_1 + \overline{x}_2 + x_3)(\overline{x}_1 + x_2 + \overline{x}_3) \\ &= ((x_1 + x_3) + x_2)((x_1 + x_3) + \overline{x}_2)(\overline{x}_1 + x_2 + \overline{x}_3) \\ &= (x_1 + x_3)(\overline{x}_1 + x_2 + \overline{x}_3) \end{aligned}$$

2.22. The simplest POS implementation of the function is

$$\begin{aligned} f &= (x_1 + x_2 + x_3)(x_1 + x_2 + \overline{x}_3)(\overline{x}_1 + x_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3) \\ &= ((x_1 + x_2) + x_3)((x_1 + x_2) + \overline{x}_3)((\overline{x}_1 + x_3) + x_2)((\overline{x}_1 + x_3) + \overline{x}_2) \\ &= (x_1 + x_2)(\overline{x}_1 + \overline{x}_3) \end{aligned}$$

2.23. The lowest cost circuit is defined by

$$f(x_1, x_2, x_3) = x_1x_2 + x_1x_3 + x_2x_3$$

2.24. The truth table that corresponds to the timing diagram in Figure P2.3 is

x_1	x_2	x_3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

The simplest SOP expression is $f = \overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1 x_2 x_3 + x_1\overline{x}_2 x_3 + x_1 x_2\overline{x}_3$.

2.25. The truth table that corresponds to the timing diagram in Figure P2.4 is

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

The simplest SOP expression is derived as follows:

$$\begin{aligned}
f &= \overline{x}_1\overline{x}_2 x_3 + \overline{x}_1 x_2\overline{x}_3 + \overline{x}_1 x_2 x_3 + x_1\overline{x}_2\overline{x}_3 + x_1 x_2 x_3 \\
&= \overline{x}_1(\overline{x}_2 + x_2)x_3 + \overline{x}_1 x_2(\overline{x}_3 + x_3) + (\overline{x}_1 + x_1)x_2 x_3 + x_1\overline{x}_2\overline{x}_3 \\
&= \overline{x}_1 \cdot 1 \cdot x_3 + \overline{x}_1 x_2 \cdot 1 + 1 \cdot x_2 x_3 + x_1\overline{x}_2\overline{x}_3 \\
&= \overline{x}_1 x_3 + \overline{x}_1 x_2 + x_2 x_3 + x_1\overline{x}_2\overline{x}_3
\end{aligned}$$

2.26. (*a*)

x_1	x_0	y_1	y_0	f
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

(*b*) The simplest POS expression is

$$f = (x_1 + \overline{y}_1)(\overline{x}_1 + y_1)(x_0 + \overline{y}_0)(\overline{x}_0 + y_0)$$

2.27. (*a*)

x_1	x_0	y_1	y_0	f
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

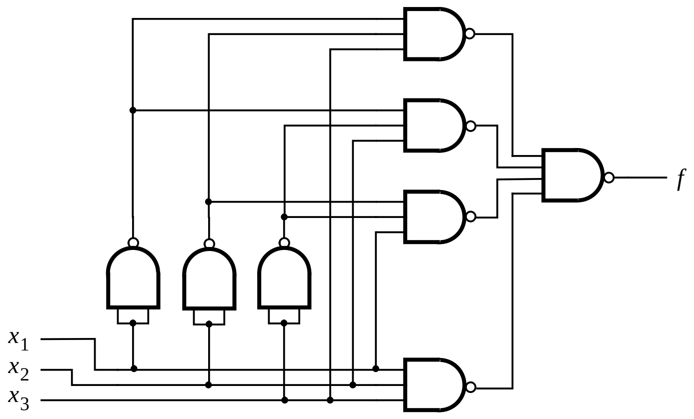
(b) The canonical SOP expression is

$$\begin{aligned} f &= \overline{x}_1\overline{x}_0\overline{y}_1\overline{y}_0 + \overline{x}_1x_0\overline{y}_1\overline{y}_0 + \overline{x}_1x_0\overline{y}_1y_0 + x_1\overline{x}_0\overline{y}_1\overline{y}_0 + x_1\overline{x}_0\overline{y}_1y_0 + x_1\overline{x}_0y_1\overline{y}_0 \\ &\quad + x_1x_0\overline{y}_1\overline{y}_0 + x_1x_0\overline{y}_1y_0 + x_1x_0y_1\overline{y}_0 + x_1x_0y_1y_0 \end{aligned}$$

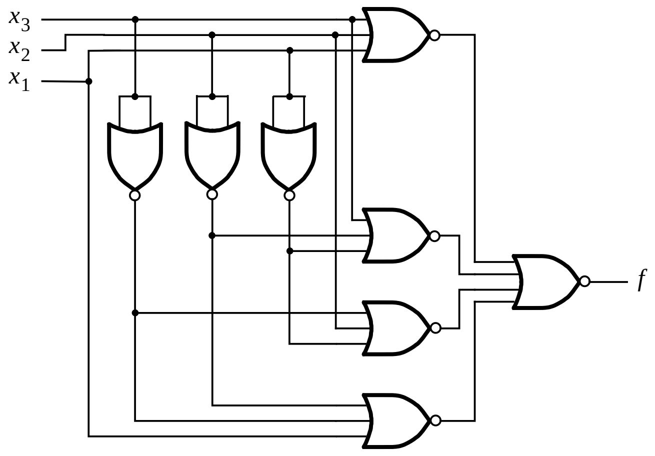
(c) The simplest SOP expression is

$$f = x_1x_0 + \overline{y}_1\overline{y}_0 + x_1\overline{y}_0 + x_0\overline{y}_1$$

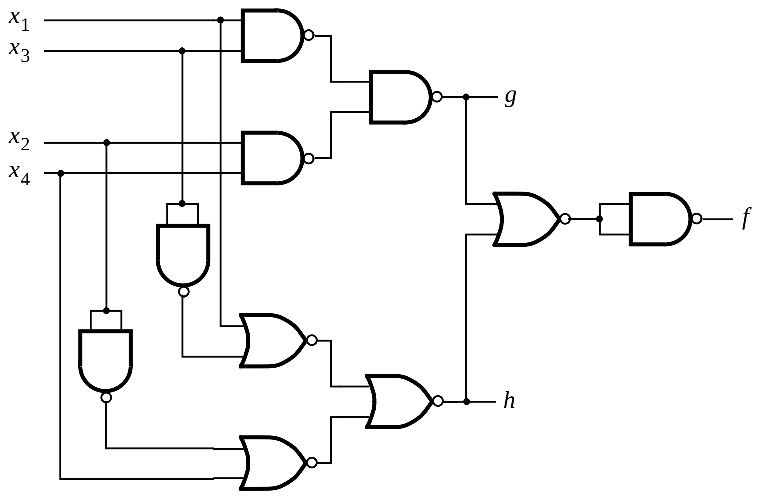
2.28. Using the ciruit in Figure 2.25a as a starting point, the function in Figure 2.24 can be implemented using NAND gates as follows:



2.29. Using the ciruit in Figure 2.25b as a starting point, the function in Figure 2.24 can be implemented using NOR gates as follows:



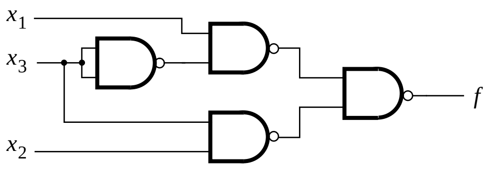
2.30. The circuit in Figure 2.33 can be implemented using NAND and NOR gates as follows:



2.31. The minimum-cost SOP expression for the function $f(x_1, x_2, x_3) = \sum m(3, 4, 6, 7)$ is

$$f = x_1\overline{x}_3 + x_2x_3$$

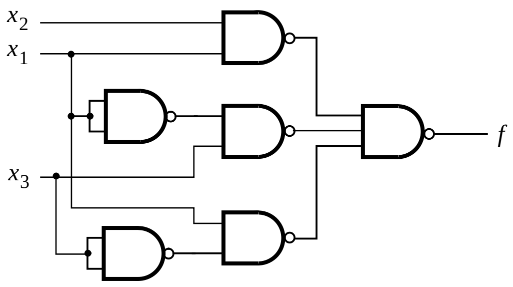
The corresponding circuit implemented using NAND gates is



2.32. A minimum-cost SOP expression for the function $f(x_1, x_2, x_3) = \sum m(1, 3, 4, 6, 7)$ is

$$f = x_1x_2 + x_1\overline{x}_3 + \overline{x}_1x_3$$

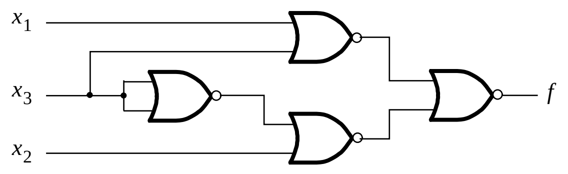
The corresponding circuit implemented using NAND gates is



2.33. The minimum-cost POS expression for the function $f(x_1, x_2, x_3) = \sum m(3, 4, 6, 7)$ is

$$f = (x_1 + x_3)(x_2 + \overline{x}_3)$$

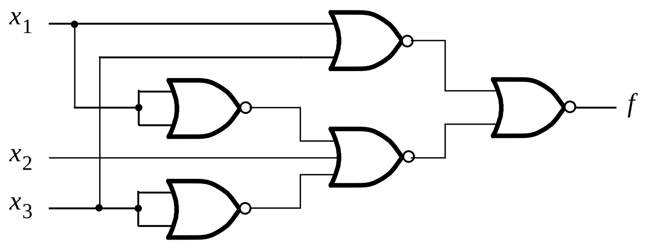
The corresponding circuit implemented using NOR gates is



2.34. The minimum-cost POS expression for the function $f(x_1, x_2, x_3) = \sum m(1, 3, 4, 6, 7)$ is

$$f = (x_1 + x_3)(\overline{x}_1 + x_2 + \overline{x}_3)$$

The corresponding circuit implemented using NOR gates is



2.37. The circuit in Figure 2.25a can be implemented using;

```
module  prob2_37 (x1, x2, x3, f);
   input  x1, x2, x3;
   output  f;

   not (notx1, x1);
   not (notx2, x2);
   not (notx3, x3);
   and (a, notx1, notx2, x3);
   and (b, notx1, x2, notx3);
   and (c, x1, notx2, notx3);
   and (d, x1, x2, x3);
   or (f, a, b, c, d);

endmodule
```

2.38. The circuit in Figure 2.25*b* can be implemented using;

```
module prob2_38 (x1, x2, x3, f);
    input x1, x2, x3;
    output f;

    not (notx1, x1);
    not (notx2, x2);
    not (notx3, x3);
    or (a, x1, x2, x3);
    or (b, notx1, notx2, x3);
    or (c, notx1, x2, notx3);
    or (d, x1, notx2, notx3);
    and (f, a, b, c, d);

endmodule
```

2.39. The simplest circuit is obtained in the POS form as

$$f = (x_1 + x_2 + x_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3)$$

Verilog code that implements the circuit is

```
module prob2_39 (x1, x2, x3, f);
    input x1, x2, x3;
    output f;

    or (g, x1, x2, x3);
    or (h, ~x1, ~x2, ~x3);
    and (f, g, h);

endmodule
```

2.40. The simplest circuit is obtained in the SOP form as

$$f = \overline{x}_2 + \overline{x}_1 x_3 + x_1 \overline{x}_3$$

Verilog code that implements the circuit is

```
module prob2_40 (x1, x2, x3, f);
    input x1, x2, x3;
    output f;

    assign f = ~x2 | (~x1 & x3) | (x1 & ~x3);

endmodule
```

2.41. The Verilog code is

```
module  prob2_41 (x1, x2, x3, x4, f1, f2);
    input  x1, x2, x3, x4;
    output  f1, f2;

    assign f1 = (x1 & ~x3) | (x2 & ~x3) | (~x3 & ~x4) | (x1 & x2) | (x1 & ~x4);
    assign f2 = (x1 | ~x3) & (x1 | x2 | ~x4) & (x2 | ~x3 | ~x4);

endmodule
```

2.42. The Verilog code is

```
module  prob2_42 (x1, x2, x3, x4, f1, f2);
    input  x1, x2, x3, x4;
    output  f1, f2;

    assign f1 = (x1 & x3) | (~x1 & ~x3) | (x2 & x4) | (~x2 & ~x4);
    assign f2 = (x1 & x2 & ~x3 & ~x4) | (~x1 & ~x2 & x3 & x4) |
            (x1 & ~x2 & ~x3 & x4) | (~x1 & x2 & x3 & ~x4);

endmodule
```

Chapter 3

3.1. (*a*)

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

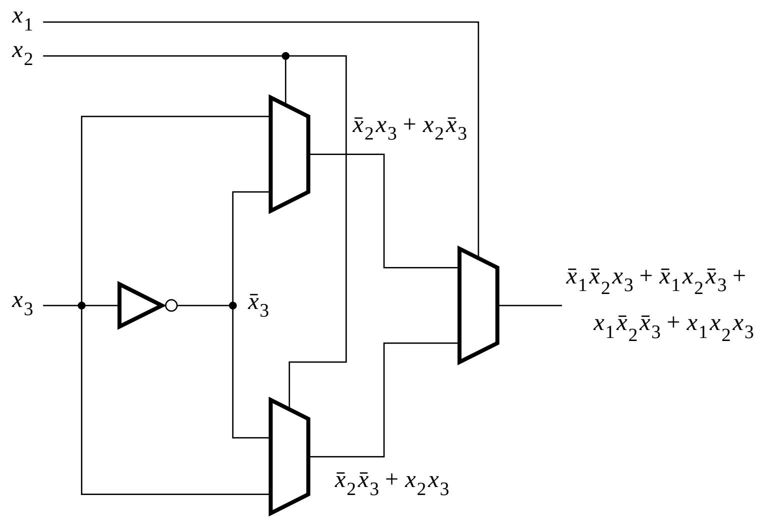
(*b*)

$$\begin{aligned}\#\text{transistors} &= \text{NOT\_gates} \times 2 + \text{AND\_gates} \times 8 + \text{OR\_gates} \\ &= 3 \times 2 + 4 \times 8 + 1 \times 10 = 48\end{aligned}$$

3.2. (*a*) In problem 3.1 the canonical SOP for f is

$$f = \overline{x}_1\overline{x}_2x_3 + \overline{x}_1x_2\overline{x}_3 + x_1\overline{x}_2\overline{x}_3 + x_1x_2x_3$$

This expression is equivalent to f in Figure P3.2, as derived below.



x_1
x_2
$\bar{x}_2x_3 + x_2\bar{x}_3$
x_3
$\bar{x}_3$
$\bar{x}_1\bar{x}_2x_3 + \bar{x}_1x_2\bar{x}_3 +$
$x_1\bar{x}_2\bar{x}_3 + x_1x_2x_3$
$\bar{x}_2\bar{x}_3 + x_2x_3$

(*b*) Assuming the multiplexers are implemented using transmission gates

$$\begin{aligned}\#\text{transistors} &= \text{NOT\_gates} \times 2 + \text{MUXes} \times 6 \\ &= 1 \times 2 + 3 \times 6 = 20\end{aligned}$$

3.3. (a) A SOP expression for f in Figure P3.3 is:

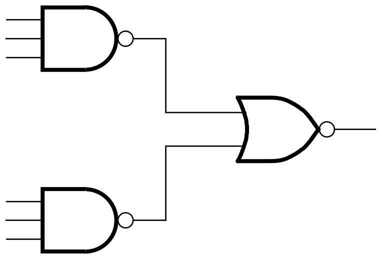
$$\begin{aligned} f &= (x_1 \oplus x_2) \oplus x_3 \\ &= (x_1 \oplus x_2)\overline{x}_3 + \overline{(x_1 \oplus x_2)}x_3 \\ &= x_1\overline{x}_2\overline{x}_3 + \overline{x}_1 x_2 \overline{x}_3 + \overline{x}_1\overline{x}_2 x_3 + x_1 x_2 x_3 \end{aligned}$$

which is equivalent to the expression derived in problem 3.2.

(b) Assuming the XOR gates are implemented as shown in Figure 3.61b

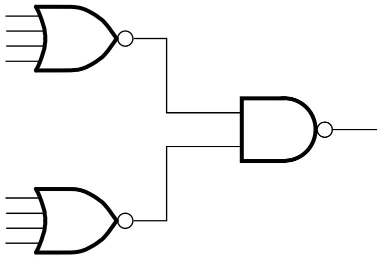
$$\begin{aligned} \#\text{transistors} &= \text{XOR\_gates} \times 8 \\ &= 2 \times 8 = 16 \end{aligned}$$

3.4. Using the circuit



The number of transistors needed is 16.

3.5. Using the circuit



The number of transistors needed is 20.

3.6. (a)

x_1	x_2	x_3	f
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

(*b*) The canonical SOP expression is

$$f = \overline{x}_1\overline{x}_2\overline{x}_3 + \overline{x}_1\overline{x}_2 x_3 + \overline{x}_1 x_2\overline{x}_3 + \overline{x}_1 x_2 x_3 + x_1\overline{x}_2\overline{x}_3$$

The number of transistors required using only AND, OR, and NOT gates is

$$\begin{aligned}\#\text{transistors} &= \text{NOT\_gates} \times 2 + \text{AND\_gates} \times 8 + \text{OR\_gates} \times 12 \\ &= 3 \times 2 + 5 \times 8 + 1 \times 12 = 58\end{aligned}$$

3.7. (*a*)

x_1	x_2	x_3	x_4	f
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

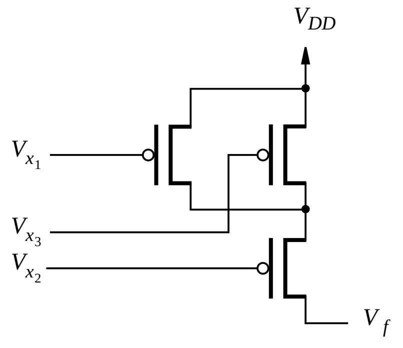
(*b*)

$$\begin{aligned}f &= \overline{x}_1\overline{x}_2\overline{x}_3\overline{x}_4 + \overline{x}_1 x_2\overline{x}_3\overline{x}_4 + x_1\overline{x}_2\overline{x}_3\overline{x}_4 \\ &= \overline{x}_1\overline{x}_3\overline{x}_4 + \overline{x}_2\overline{x}_3\overline{x}_4\end{aligned}$$

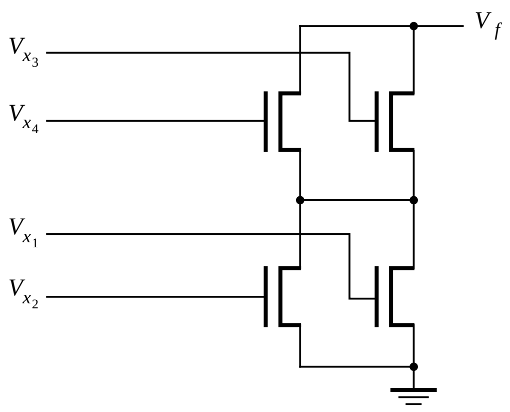
The number of transistors required using only AND, OR, and NOT gates is

$$\begin{aligned}\#\text{transistors} &= \text{NOT\_gates} \times 2 + \text{AND\_gates} \times 8 + \text{OR\_gates} \times 4 \\ &= 4 \times 2 + 2 \times 8 + 1 \times 4 = 28\end{aligned}$$

3.8.



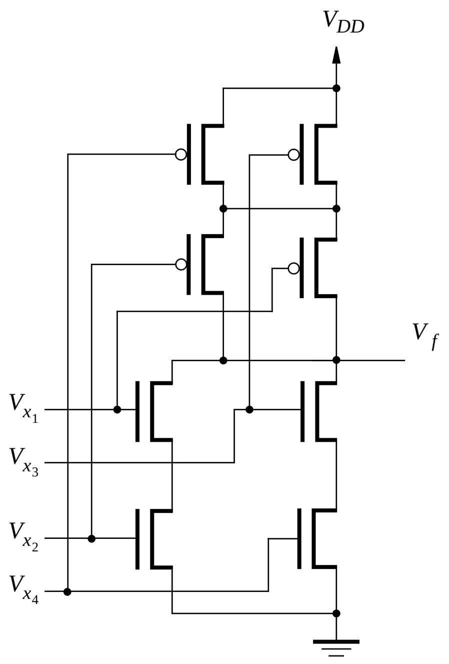
3.9.



3.10. Minimum SOP expression for f is

$$\begin{aligned} f &= \overline{x}_2\overline{x}_3 + \overline{x}_1\overline{x}_3 + \overline{x}_2\overline{x}_4 + \overline{x}_1\overline{x}_4 \\ &= (\overline{x}_1 + \overline{x}_2)(\overline{x}_3 + \overline{x}_4) \end{aligned}$$

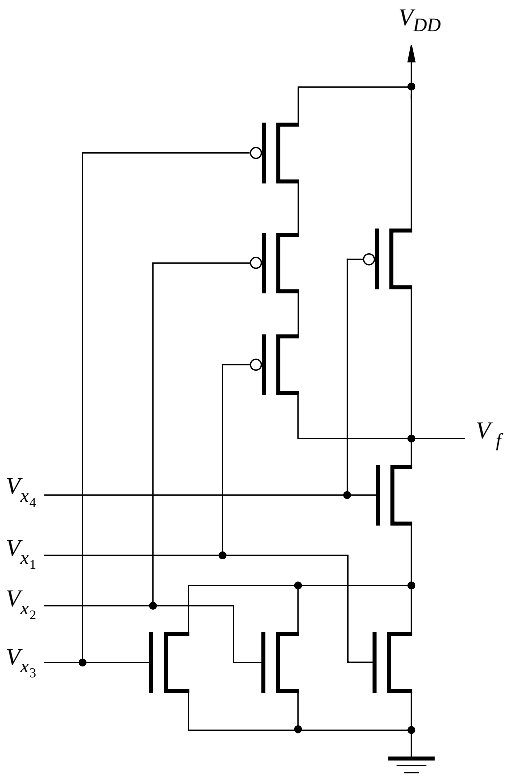
which leads to the circuit



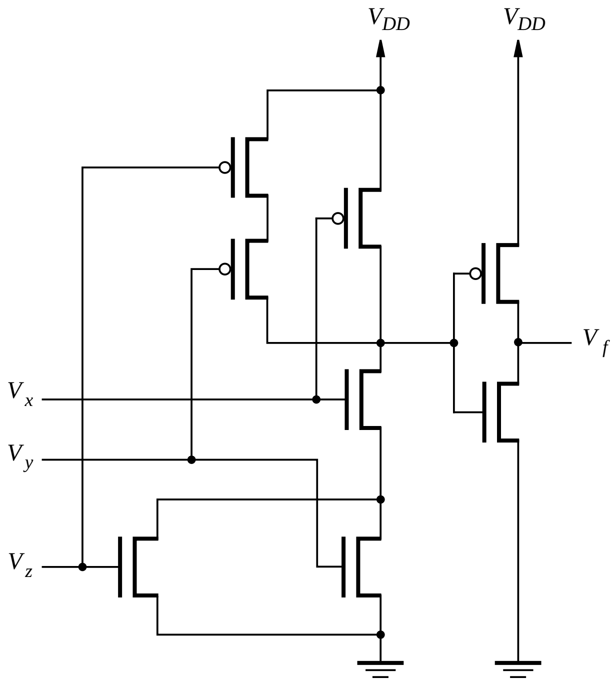
3.11. Minimum SOP expression for f is

$$f = \overline{x}_4 + \overline{x}_1\overline{x}_2\overline{x}_3$$

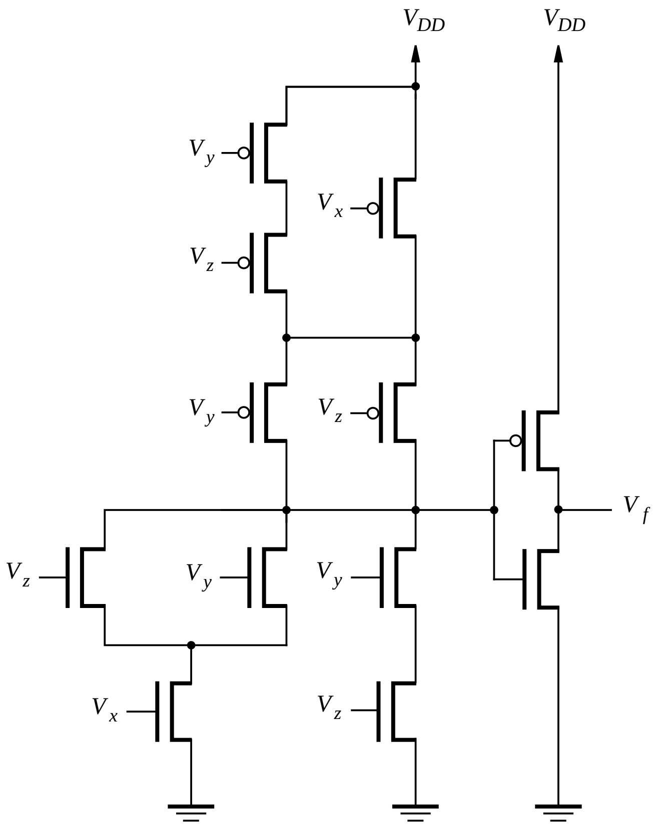
which leads to the circuit



3.12.



3.13.



3.14. (a) Since $V_{DS} \geq V_{GS} - V_T$ the NMOS transistor is operating in the saturation region:

$$\begin{aligned} I_D &= \frac{1}{2}k_n' \frac{W}{L}(V_{GS} - V_T)^2 \\ &= 10\,\frac{\mu\text{A}}{\text{V}^2} \times 5 \times (5\text{ V} - 1\text{ V})^2 = 800\,\mu\text{A} \end{aligned}$$

(b) In this case $V_{DS} < V_{GS} - V_T$, thus the NMOS transistor is operating in the triode region:

$$\begin{aligned} I_D &= k_n' \frac{W}{L}\left[(V_{GS} - V_T)V_{DS} - \frac{1}{2}V_{DS}^2\right] \\ &= 20\,\frac{\mu\text{A}}{\text{V}^2} \times 5 \times \left[(5\text{ V} - 1\text{ V}) \times 0.2\text{ V} - \frac{1}{2} \times (0.2\text{ V})^2\right] = 78\,\mu\text{A} \end{aligned}$$

3.15. (a) Since $V_{DS} \leq V_{GS} - V_T$ the PMOS transistor is operating in the saturation region:

$$\begin{aligned} I_D &= \frac{1}{2}k_p' \frac{W}{L}(V_{GS} - V_T)^2 \\ &= 5\,\frac{\mu\text{A}}{\text{V}^2} \times 5 \times (-5\text{ V} + 1\text{ V})^2 = 400\,\mu\text{A} \end{aligned}$$

(b) In this case $V_{DS} > V_{GS} - V_T$, thus the PMOS transistor is operating in the triode region:

$$\begin{aligned} I_D &= k_p' \frac{W}{L}\left[(V_{GS} - V_T)V_{DS} - \frac{1}{2}V_{DS}^2\right] \\ &= 10\,\frac{\mu\text{A}}{\text{V}^2} \times 5 \times \left[(-5\text{ V} + 1\text{ V}) \times (-0.2)\text{ V} - \frac{1}{2} \times (-0.2\text{ V})^2\right] = 39\,\mu\text{A} \end{aligned}$$

3.16.

$$\begin{aligned}R_{DS} &= 1/\left[k_n' \frac{W}{L}(V_{GS} - V_T)\right]\\ &= 1/\left[0.020\,\frac{\text{mA}}{\text{V}^2} \times 10 \times (5\text{ V} - 1\text{ V})\right] = 1.25\,\text{k}\Omega\end{aligned}$$

3.17.

$$\begin{aligned}R_{DS} &= 1/\left[k_n' \frac{W}{L}(V_{GS} - V_T)\right]\\ &= 1/\left[0.040\,\frac{\text{mA}}{\text{V}^2} \times 10 \times (3.3\text{ V} - 0.66\text{ V})\right] = 947\,\Omega\end{aligned}$$

3.18. Since $V_{DS} < (V_{GS} - V_T)$, the PMOS transistor is operating in the saturation region:

$$\begin{aligned}I_{SD} &= \frac{1}{2}k_p' \frac{W}{L}(V_{GS} - V_T)^2\\ &= 50\,\frac{\mu\text{A}}{\text{V}^2} \times (-5\text{ V} + 1\text{ V})^2 = 800\,\mu\text{A}\end{aligned}$$

Hence the value of R_{DS} is

$$\begin{aligned}R_{DS} &= V_{DS}/I_{DS}\\ &= 4.8\text{ V}/800\,\mu\text{A} = 6\,\text{k}\Omega\end{aligned}$$

3.19. Since $V_{DS} < (V_{GS} - V_T)$, the PMOS transistor is operating in the saturation region:

$$\begin{aligned}I_{SD} &= \frac{1}{2}k_p' \frac{W}{L}(V_{GS} - V_T)^2\\ &= 80\,\frac{\mu\text{A}}{\text{V}^2} \times (-3.3\text{ V} + 0.66\text{ V})^2 = 558\,\mu\text{A}\end{aligned}$$

Hence the value of R_{DS} is

$$\begin{aligned}R_{DS} &= V_{DS}/I_{DS}\\ &= 3.2\text{ V}/558\,\mu\text{A} = 5.7\,\text{k}\Omega\end{aligned}$$

3.20. The low output voltage of the pseudo-NMOS inverter can be obtained by setting $V_x = V_{DD}$ and evaluating the voltage V_f. First we assume that the NMOS transistor is operating in the triode region while the PMOS is operating in the saturation region. For simplicity we will assume that the magnitude of the threshold voltages for both the NMOS and PMOS transistors are equal, so that

$$V_T = V_{TN} = -V_{TP}$$

The current flowing through the PMOS transistor is

$$\begin{aligned}I_D &= \frac{1}{2}k_p' \frac{W_p}{L_p}(-V_{DD} - V_{TP})^2\\ &= \frac{1}{2}k_p(-V_{DD} - V_{TP})^2\\ &= \frac{1}{2}k_p(V_{DD} - V_T)^2\end{aligned}$$

Similarly, the current going through the NMOS transistor is

$$\begin{aligned} I_D &= k_n' \frac{W_n}{L_n}\left[(V_x - V_{TN})V_f - \frac{1}{2}V_f^2\right] \\ &= k_n \left[(V_x - V_{TN})V_f - \frac{1}{2}V_f^2\right] \\ &= k_n \left[(V_{DD} - V_T)V_f - \frac{1}{2}V_f^2\right] \end{aligned}$$

Since there is only one path for current to flow, we can equate the currents flowing through the NMOS and PMOS transistors and solve for the voltage V_f.

$$k_p(V_{DD} - V_T)^2 = 2k_n\left[(V_{DD} - V_T)V_f - \frac{1}{2}V_f^2\right]$$

$$k_p(V_{DD} - V_T)^2 - 2k_n(V_{DD} - V_T)V_f + k_nV_f^2 = 0$$

This quadratic equation can be solved using the standard formula, with the parameters

$$a = k_n,\ b = -2k_n(V_{DD} - V_T),\ c = k_p(V_{DD} - V_T)^2$$

which gives

$$\begin{aligned} V_f &= \frac{-b}{2a} \pm \sqrt{\frac{b^2}{4a^2} - \frac{c}{a}} \\ &= (V_{DD} - V_T) \pm \sqrt{(V_{DD} - V_T)^2 - \frac{k_p}{k_n}(V_{DD} - V_T)^2} \\ &= (V_{DD} - V_T)\left[1 \pm \sqrt{1 - \frac{k_p}{k_n}}\right] \end{aligned}$$

Only one of these two solutions is valid, because we started with the assumption that the NMOS transistor is in the triode region while the PMOS is in the saturation region. Thus

$$V_f = (V_{DD} - V_T)\left[1 - \sqrt{1 - \frac{k_p}{k_n}}\right]$$

3.21. (a)

$$\begin{aligned} I_{stat} &= \frac{1}{2}k_p'\frac{W_p}{L_p}(V_{DD} - V_T)^2 \\ &= 12\,\frac{\mu\text{A}}{\text{V}^2} \times 1 \times (5\text{ V} - 1\text{ V})^2 = 192\,\mu\text{A} \end{aligned}$$

(b)

$$\begin{aligned} R_{DS} &= 1/\left[k_n'\frac{W_n}{L_n}(V_{GS} - V_T)\right] \\ &= 1/\left[0.060\,\frac{\text{mA}}{\text{V}^2} \times 4 \times (5\text{ V} - 1\text{ V})\right] = 1.04\,\text{k}\Omega \end{aligned}$$

(c) Using the expression derived in problem 3.20

$$k_p = k_p'\frac{W_p}{L_p} = 24\,\frac{\mu\text{A}}{\text{V}^2}$$

$$k_n = k_n'\frac{W_n}{L_n} = 240\,\frac{\mu\text{A}}{\text{V}^2}$$

$$\begin{aligned} V_{OL} = V_f &= (5\text{ V} - 1\text{ V})\left[1 - \sqrt{1 - \frac{24}{240}}\right] \\ &= 0.21\text{ V} \end{aligned}$$

(d)

$$\begin{aligned} P_D &= I_{stat}V_{DD} \\ &= 192\,\mu\text{A} \times 5\text{ V} = 960\,\mu\text{W} \approx 1\text{m}W \end{aligned}$$

(e)

$$\begin{aligned} R_{SDP} &= V_{SD}/I_{SD} \\ &= (V_{DD} - V_f)/I_{stat} \\ &= (5\text{ V} - 0.21\text{ V})/0.192\text{ mA} = 24.9\text{ k}\Omega \end{aligned}$$

(f) The low-to-high propagation delay is

$$\begin{aligned} t_{p_{LH}} &= \frac{1.7C}{k'_p \frac{W_p}{L_p} V_{DD}} \\ &= \frac{1.7 \times 70\text{ fF}}{24\,\frac{\mu\text{A}}{\text{V}^2} \times 1 \times 5\text{ V}} = 0.99\text{ ns} \end{aligned}$$

The high-to-low propagation delay is

$$\begin{aligned} t_{p_{HL}} &= \frac{1.7C}{k'_n \frac{W_n}{L_n} V_{DD}} \\ &= \frac{1.7 \times 70\text{ fF}}{60\,\frac{\mu\text{A}}{\text{V}^2} \times 4 \times 5\text{ V}} = 0.1\text{ ns} \end{aligned}$$

3.22. (a)

$$\begin{aligned} I_{stat} &= \frac{1}{2}k'_p \frac{W_p}{L_p}(V_{DD} - V_T)^2 \\ &= 48\,\frac{\mu\text{A}}{\text{V}^2} \times 1 \times (5\text{ V} - 1\text{ V})^2 = 768\,\mu\text{A} \end{aligned}$$

(b)

$$\begin{aligned} R_{DS} &= 1/\left[k'_n \frac{W_n}{L_n}(V_{GS} - V_T)\right] \\ &= 1/\left[0.060\,\frac{\text{mA}}{\text{V}^2} \times 4 \times (5\text{ V} - 1\text{ V})\right] = 1.04\text{ k}\Omega \end{aligned}$$

(c) Using the expression derived in problem 3.20

$$k_p = k'_p \frac{W_p}{L_p} = 96\,\frac{\mu\text{A}}{\text{V}^2} k_n = k'_n \frac{W_n}{L_n} = 240\,\frac{\mu\text{A}}{\text{V}^2}$$

$$\begin{aligned} V_{OL} = V_f &= (5\,\mathrm{V} - 1\,\mathrm{V})\left[1 - \sqrt{1 - \frac{96}{240}}\right] \\ &= 0.90\,\mathrm{V} \end{aligned}$$

(d)

$$\begin{aligned} P_D &= I_{stat}V_{DD} \\ &= 768\,\mu\mathrm{A} \times 5\,\mathrm{V} = 3840\,\mu\mathrm{W} \approx 3.8\mathrm{m}W \end{aligned}$$

(e)

$$\begin{aligned} R_{SDP} &= V_{SD}/I_{SD} \\ &= (V_{DD} - V_f)/I_{stat} \\ &= (5\,\mathrm{V} - 0.90\,\mathrm{V})/0.768\,\mathrm{mA} = 5.34\,\mathrm{k}\Omega \end{aligned}$$

(f) The low-to-high propagation delay is

$$\begin{aligned} t_{p_{LH}} &= \frac{1.7C}{k_p' \frac{W_p}{L_p} V_{DD}} \\ &= \frac{1.7 \times 70\,\mathrm{fF}}{96\,\frac{\mu\mathrm{A}}{\mathrm{V}^2} \times 1 \times 5\,\mathrm{V}} = 0.25\,\mathrm{ns} \end{aligned}$$

The high-to-low propagation delay is

$$\begin{aligned} t_{p_{HL}} &= \frac{1.7C}{k_n' \frac{W_n}{L_n} V_{DD}} \\ &= \frac{1.7 \times 70\,\mathrm{fF}}{60\,\frac{\mu\mathrm{A}}{\mathrm{V}^2} \times 4 \times 5\,\mathrm{V}} = 0.1\,\mathrm{ns} \end{aligned}$$

3.23. (a)

$$\begin{aligned} I_{stat} &= \frac{1}{2}k_p' \frac{W_p}{L_p}(V_{DD} - V_T)^2 \\ &= 12\,\frac{\mu\mathrm{A}}{\mathrm{V}^2} \times 1 \times (5\,\mathrm{V} - 1\,\mathrm{V})^2 = 192\,\mu\mathrm{A} \end{aligned}$$

(b) The two NMOS transistors in series can be considered equivalent to a single transistor with twice the length. Thus

$$\begin{aligned} R_{DS} &= 1/\left[k_n' \frac{W_n}{L_n}(V_{GS} - V_T)\right] \\ &= 1/\left[0.060\,\frac{\mathrm{mA}}{\mathrm{V}^2} \times 2 \times (5\,\mathrm{V} - 1\,\mathrm{V})\right] = 2.08\,\mathrm{k}\Omega \end{aligned}$$

(c) Using the expression derived in problem 3.20

$$k_p = k_p' \frac{W_p}{L_p} = 24\,\frac{\mu\mathrm{A}}{\mathrm{V}^2}$$
$$k_n = k_n' \frac{W_n}{L_n} = 120\,\frac{\mu\mathrm{A}}{\mathrm{V}^2}$$

$$
\begin{aligned}
V_{OL} = V_f &= (5\text{ V} - 1\text{ V})\left[1 - \sqrt{1 - \frac{24}{120}}\right] \\
&= 0.42\text{ V}
\end{aligned}
$$

(d)

$$
\begin{aligned}
P_D &= I_{stat}V_{DD} \\
&= 192\,\mu\text{A} \times 5\text{ V} = 960\,\mu\text{W} \approx 1\text{m}W
\end{aligned}
$$

(e)

$$
\begin{aligned}
R_{SDP} &= V_{SD}/I_{SD} \\
&= (V_{DD} - V_f)/I_{stat} \\
&= (5\text{ V} - 0.42\text{ V})/0.192\text{ mA} = 23.9\text{ k}\Omega
\end{aligned}
$$

(f) The low-to-high propagation delay is

$$
\begin{aligned}
t_{p_{LH}} &= \frac{1.7C}{k_p' \frac{W_p}{L_p} V_{DD}} \\
&= \frac{1.7 \times 70\text{ fF}}{24\,\frac{\mu\text{A}}{\text{V}^2} \times 1 \times 5\text{ V}} = 0.99\text{ ns}
\end{aligned}
$$

The high-to-low propagation delay is

$$
\begin{aligned}
t_{p_{HL}} &= \frac{1.7C}{k_n' \frac{W_n}{L_n} V_{DD}} \\
&= \frac{1.7 \times 70\text{ fF}}{60\,\frac{\mu\text{A}}{\text{V}^2} \times 2 \times 5\text{ V}} = 0.2\text{ ns}
\end{aligned}
$$

3.24. (a)

$$
\begin{aligned}
I_{stat} &= \frac{1}{2}k_p' \frac{W_p}{L_p}(V_{DD} - V_T)^2 \\
&= 12\,\frac{\mu\text{A}}{\text{V}^2} \times 1 \times (5\text{ V} - 1\text{ V})^2 = 192\,\mu\text{A}
\end{aligned}
$$

(b) The two NMOS transistors in parallel can be considered equivalent to a single transistor with twice the width. Thus

$$
\begin{aligned}
R_{DS} &= 1/\left[k_n' \frac{W_n}{L_n}(V_{GS} - V_T)\right] \\
&= 1/\left[0.060\,\frac{\text{mA}}{\text{V}^2} \times 8 \times (5\text{ V} - 1\text{ V})\right] = 520\,\Omega
\end{aligned}
$$

(c) Using the expression derived in problem 3.20

$$
k_p = k_p' \frac{W_p}{L_p} = 24\,\frac{\mu\text{A}}{\text{V}^2}
$$

$$
k_n = k_n' \frac{W_n}{L_n} = 480\,\frac{\mu\text{A}}{\text{V}^2}
$$

$$
\begin{aligned}
V_{OL} = V_f &= (5\ \mathrm{V} - 1\ \mathrm{V})\left[1 - \sqrt{1 - \frac{24}{480}}\right] \\
&= 0.10\ \mathrm{V}
\end{aligned}
$$

(d)

$$
\begin{aligned}
P_D &= I_{stat} V_{DD} \\
&= 192\,\mu\mathrm{A} \times 5\ \mathrm{V} = 960\,\mu\mathrm{W} \approx 1\mathrm{m}W
\end{aligned}
$$

(e)

$$
\begin{aligned}
R_{SDP} &= V_{SD}/I_{SD} \\
&= (V_{DD} - V_f)/I_{stat} \\
&= (5\ \mathrm{V} - 0.10\ \mathrm{V})/0.192\ \mathrm{mA} = 25.5\ \mathrm{k}\Omega
\end{aligned}
$$

(f) The low-to-high propagation delay is

$$
\begin{aligned}
t_{p_{LH}} &= \frac{1.7C}{k'_p \frac{W_p}{L_p} V_{DD}} \\
&= \frac{1.7 \times 70\ \mathrm{fF}}{24\ \frac{\mu\mathrm{A}}{\mathrm{V}^2} \times 1 \times 5\ \mathrm{V}} = 0.99\ \mathrm{ns}
\end{aligned}
$$

The high-to-low propagation delay is

$$
\begin{aligned}
t_{p_{HL}} &= \frac{1.7C}{k'_n \frac{W_n}{L_n} V_{DD}} \\
&= \frac{1.7 \times 70\ \mathrm{fF}}{60\ \frac{\mu\mathrm{A}}{\mathrm{V}^2} \times 8 \times 5\ \mathrm{V}} = 0.05\ \mathrm{ns}
\end{aligned}
$$

3.25. (a)

$$
\begin{aligned}
NM_H = V_{OH} - V_{IH} = 0.5\ \mathrm{V} \\
NM_L = V_{IL} - V_{OL} = 0.7\ \mathrm{V}
\end{aligned}
$$

(b)

$$
\begin{aligned}
V_{OL} = 8 \times 0.1\ \mathrm{V} = 0.8\ \mathrm{V} \\
NM_L = 1\ \mathrm{V} - 0.8\ \mathrm{V} = 0.2\ \mathrm{V}
\end{aligned}
$$

3.26. Under steady-state conditions, for an n-input CMOS NAND gate the voltage levels V_{OL} and V_{OH} are 0 V and V_{DD}, respectively. No current flows in a CMOS gate in the steady-state. Thus there can be no voltage drop across any of the transistors.

3.27. (a)

$$
\begin{aligned}
P_{NOT\_gate} &= fCV^2 \\
&= 75\ \mathrm{MHz} \times 150\ \mathrm{fF} \times (5\ \mathrm{V})^2 = 281\,\mu\mathrm{W}
\end{aligned}
$$

(b)

$$
P_{total} = 0.2 \times 250,000 \times 281\,\mu\mathrm{W} = 14\ \mathrm{W}
$$

3.28. (a)

$$\begin{aligned} P_{NOT\_gate} &= fCV^2 \\ &= 125\,\text{MHz} \times 120\,\text{fF} \times (3.3\,\text{V})^2 = 163\,\mu\text{W} \end{aligned}$$

(b)

$$P_{total} = 0.2 \times 250,000 \times 163\,\mu\text{W} = 8.2\,\text{W}$$

3.29. (a) The high-to-low propagation delay is

$$t_{p_{HL}} = \frac{1.7C}{k'_n \frac{W_n}{L_n} V_{DD}} = \frac{1.7 \times 150\,\text{fF}}{20\,\frac{\mu\text{A}}{\text{V}^2} \times 10 \times 5\,\text{V}} = 0.255\,\text{ns}$$

(b) The low-to-high propagation delay is

$$t_{p_{LH}} = \frac{1.7C}{k'_p \frac{W_p}{L_p} V_{DD}} = \frac{1.7 \times 150\,\text{fF}}{8\,\frac{\mu\text{A}}{\text{V}^2} \times 10 \times 5\,\text{V}} = 0.638\,\text{ns}$$

(c) For equivalent high-to-low and low-to-high delays

$$\begin{aligned} t_{p_{HL}} &= t_{p_{LH}} \\ \frac{1.7C}{k'_n \frac{W_n}{L_n} V_D D} &= \frac{1.7C}{k'_p \frac{W_p}{L_p} V_D D} \\ \frac{W_p}{L_p} &= \frac{\frac{k'_n}{k'_p} W_n}{L_n} \\ &= \frac{12.5\,\mu\text{m}}{0.5\,\mu\text{m}} \end{aligned}$$

3.30. (a) The high-to-low propagation delay is

$$t_{p_{HL}} = \frac{1.7C}{k'_n \frac{W_n}{L_n} V_{DD}} = \frac{1.7 \times 150\,\text{fF}}{40\,\frac{\mu\text{A}}{\text{V}^2} \times 10 \times 3.3\,\text{V}} = 0.193\,\text{ns}$$

(b) The low-to-high propagation delay is

$$t_{p_{LH}} = \frac{1.7C}{k'_p \frac{W_p}{L_p} V_{DD}} = \frac{1.7 \times 150\,\text{fF}}{16\,\frac{\mu\text{A}}{\text{V}^2} \times 10 \times 3.3\,\text{V}} = 0.483\,\text{ns}$$

(c) For equivalent high-to-low and low-to-high delays

$$\begin{aligned} t_{p_{HL}} &= t_{p_{LH}} \\ \frac{1.7C}{k'_n \frac{W_n}{L_n} V_D D} &= \frac{1.7C}{k'_p \frac{W_p}{L_p} V_D D} \\ \frac{W_p}{L_p} &= \frac{\frac{k'_n}{k'_p} W_n}{L_n} \\ &= \frac{8.75\,\mu\text{m}}{0.35\,\mu\text{m}} \end{aligned}$$

3.31. The two PMOS transistors in a CMOS NAND gate are connected in parallel. The worst case current to drive the output high happens when only one of these transistors is turned "ON". Thus each transistor has to have the same dimensions as the PMOS transistor in the inverter, namely $\frac{W_p}{L_p} = 4$.

The two NMOS transistors are connected in series. If each one had the ratio $\frac{W_n}{L_n}$, then the two transistors could be thought of as one equivalent transistor with a $\frac{W_n}{2L_n}$ ratio. Thus each NMOS transistor must have twice the width of that in the inverter, namely $\frac{W_n}{L_n} = 4$.

3.32. The two NMOS transistors in a CMOS NOR gate are connected in parallel. The worst case current to drive the output low happens when only one of these transistors is turned "ON". Thus each transistor has to have the same dimensions as the NMOS transistor in the inverter, namely $\frac{W_n}{L_n} = 2$.

The two PMOS transistors are connected in series. If each of these transistors had the ratio $\frac{W_p}{L_p}$, then the two transistors could be thought of as one transistor with a $\frac{W_p}{2L_p}$ ratio. Thus each PMOS transistor must be made twice as wide as that in the inverter, namely $\frac{W_n}{L_n} = 8$.

3.33. The worst case path in the PMOS network contains two transistors in series. Thus each PMOS transistor must be twice as wide the transistors in the inverter. The worst case path in the NMOS network also contains two transistors in series. Similarly, each NMOS transistor must be twice as wide as those in the inverter.

3.34. The worst case PMOS path contains three transistors in series so each transistor must be three times as wide as the PMOS transistors in the inverter. The worst case NMOS path contains two transistors in series. Thus the NMOS transistors must be two times as wide.

3.35. (a) The current flowing through the inverter is equal to the current flowing through the PMOS transistor. We shall assume that the PMOS transistor is operating in the saturation region.

$$\begin{aligned} I_{stat} &= \frac{1}{2}k_p' \frac{W_p}{L_p}(V_{GS} - V_{Tp})^2 \\ &= 120\,\frac{\mu\text{A}}{\text{V}^2} \times ((3.5\text{ V} - 5\text{ V}) + 1\text{ V})^2 = 30\,\mu\text{A} \end{aligned}$$

(b) The current flowing through the NMOS transistor is equal to the static current I_{stat}. Assume that the NMOS transistor is operating in the triode region.

$$\begin{aligned} I_{stat} &= k_n' \frac{W_n}{L_n}\left[(V_{GS} - V_{Tn})V_{DS} - \frac{1}{2}V_{DS}^2\right] \\ 30\,\mu\text{A} &= 240\,\frac{\mu\text{A}}{\text{V}^2} \times \left[2.5\text{ V} \times V_f - \frac{1}{2}V_f^2\right] \\ 1 &= 20V_f - 4V_f^2 \end{aligned}$$

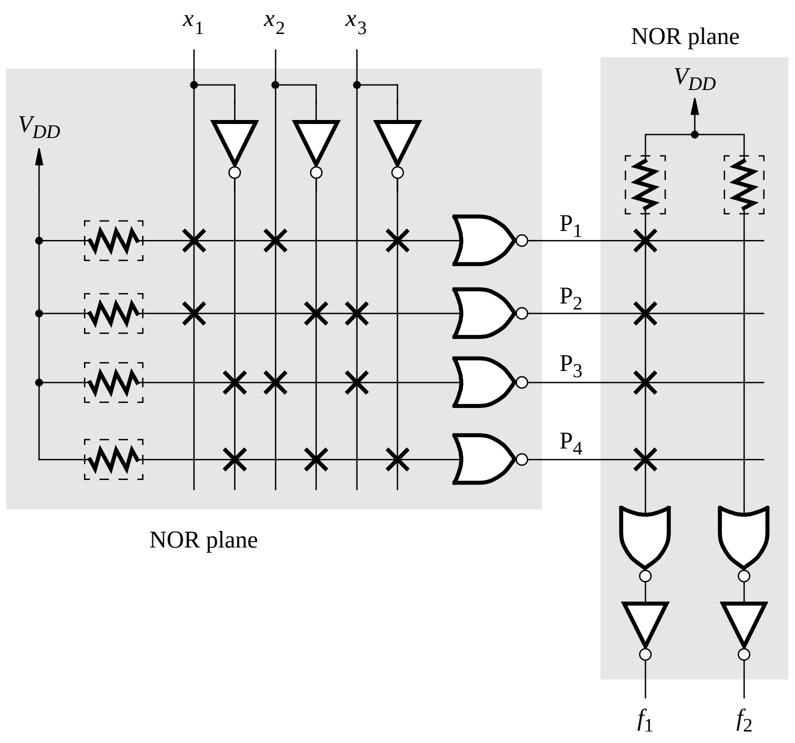
Solving this quadratic equation yields $V_f = 0.05\,\text{V}$. Note that the output voltage V_f satisfies the assumption that the PMOS transistor is operating in the saturation region while the NMOS transistor is operating in the triode region. (c) The static power dissipated in the inverter is

$$P_S = I_{stat}V_{DD} = 30\,\mu\text{A} \times 5\text{ V} = 150\,\mu\text{W}$$

(d) The static power dissipated by 250,000 inverters.

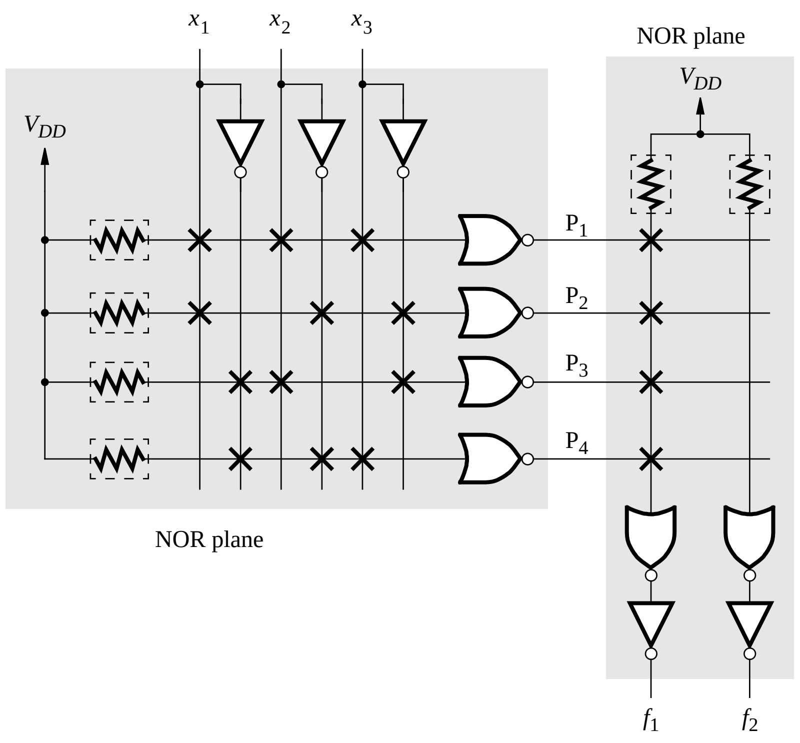
$$250,000 \times P_s = 37.5\,\text{W}$$

3.36.



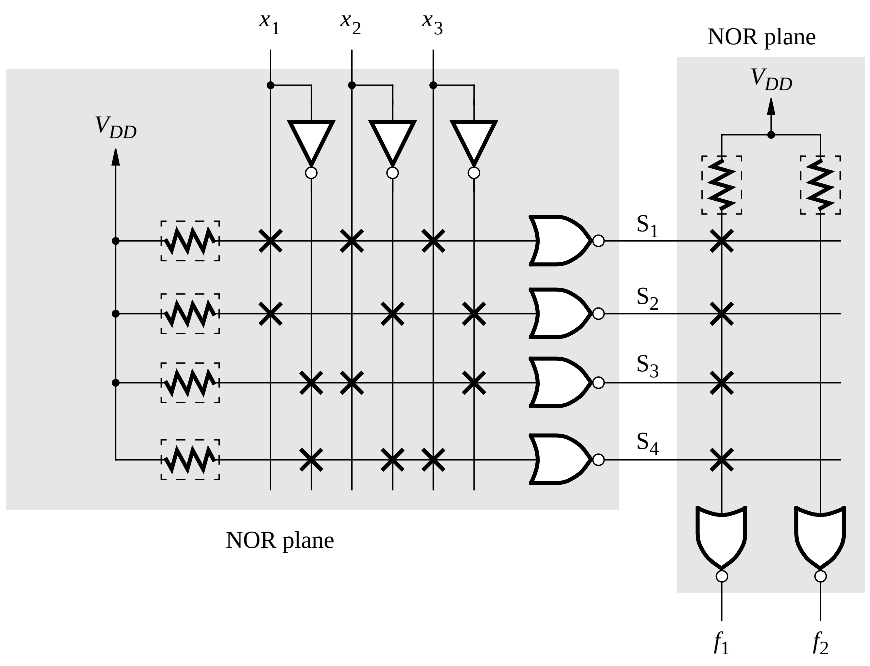
x_1
x_2
x_3
NOR plane
V_{DD}
V_{DD}
P_1
P_2
P_3
P_4
NOR plane
f_1
f_2

3.37.

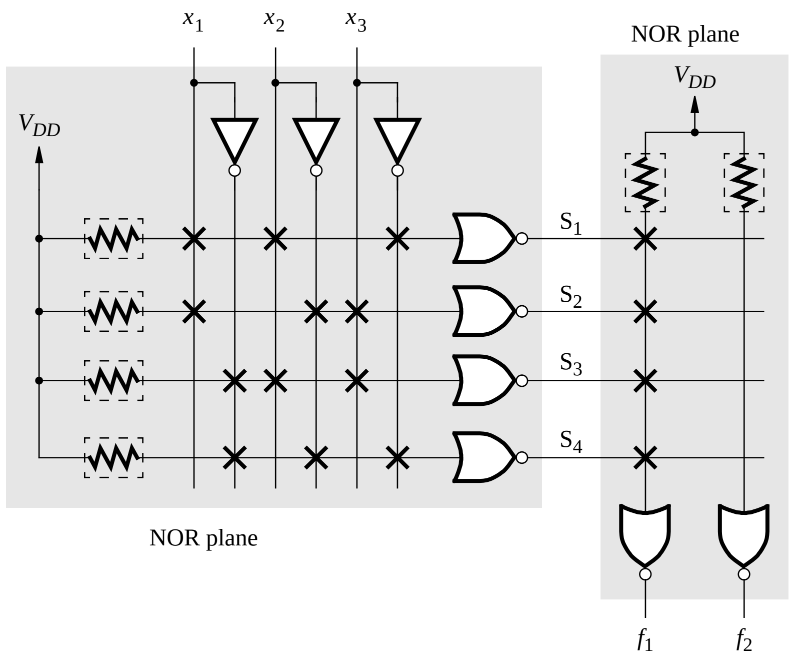


x_1
x_2
x_3
NOR plane
V_{DD}
V_{DD}
P_1
P_2
P_3
P_4
NOR plane
f_1
f_2

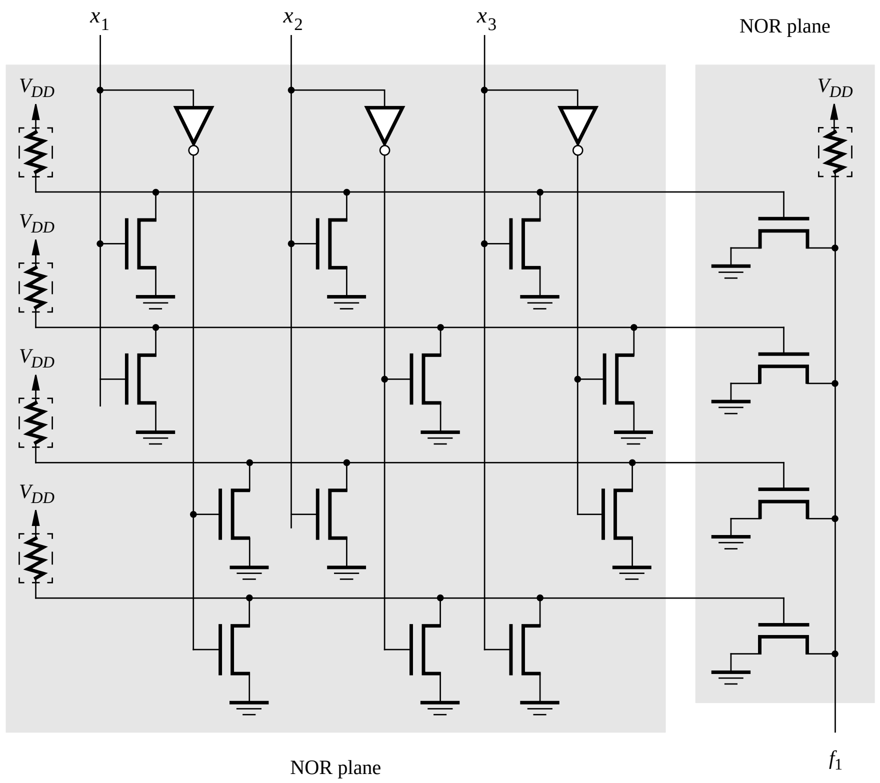
3.38.


x_1
x_2
x_3
NOR plane
V_{DD}
V_{DD}
S_1
S_2
S_3
S_4
NOR plane
f_1
f_2

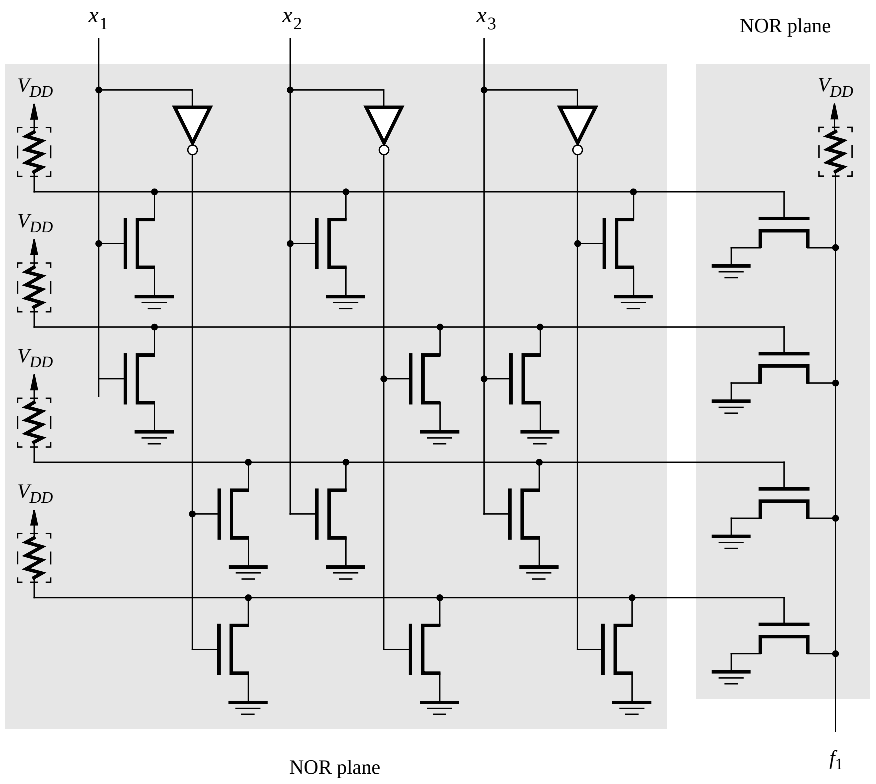
3.39.


x_1
x_2
x_3
NOR plane
V_{DD}
V_{DD}
S_1
S_2
S_3
S_4
NOR plane
f_1
f_2

3.40.


x_1
x_2
x_3
NOR plane
V_{DD}
V_{DD}
V_{DD}
V_{DD}
V_{DD}
f_1
NOR plane

3.41.


x_1
x_2
x_3
NOR plane
V_{DD}
V_{DD}
V_{DD}
V_{DD}
V_{DD}
f_1
NOR plane

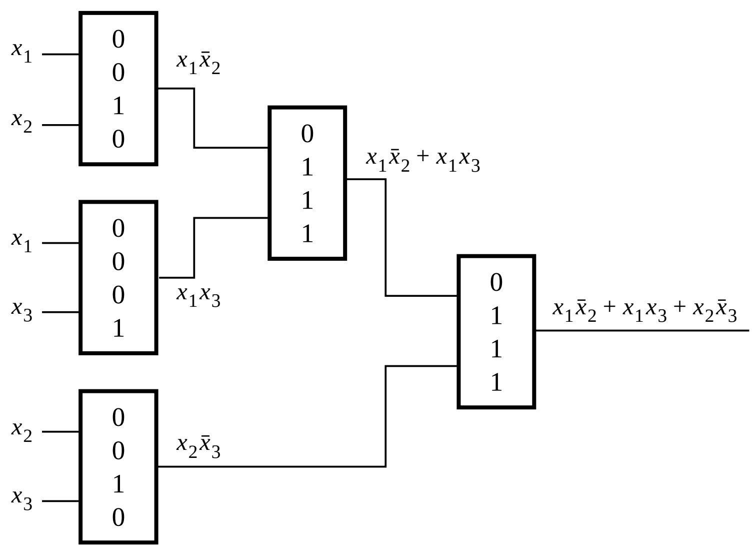
3.42.

$$\begin{aligned}
f_2 &= m_1 \\
f_2 &= m_2 \\
f_2 &= m_4 \\
f_2 &= m_7 \\
f_2 &= m_1 + m_2 \\
f_2 &= m_1 + m_4 \\
f_2 &= m_1 + m_7 \\
f_2 &= m_2 + m_4 \\
f_2 &= m_2 + m_7 \\
f_2 &= m_4 + m_7 \\
f_2 &= m_1 + m_2 + m_4 \\
f_2 &= m_1 + m_2 + m_7 \\
f_2 &= m_1 + m_4 + m_7 \\
f_2 &= m_2 + m_4 + m_7 \\
f_2 &= m_1 + m_2 + m_4 + m_7
\end{aligned}$$

3.43.

$$\begin{aligned}
f_2 &= m_0 \\
f_2 &= m_3 \\
f_2 &= m_5 \\
f_2 &= m_6 \\
f_2 &= m_0 + m_3 \\
f_2 &= m_0 + m_5 \\
f_2 &= m_0 + m_6 \\
f_2 &= m_3 + m_4 \\
f_2 &= m_3 + m_6 \\
f_2 &= m_5 + m_6 \\
f_2 &= m_0 + m_3 + m_5 \\
f_2 &= m_0 + m_3 + m_6 \\
f_2 &= m_0 + m_5 + m_6 \\
f_2 &= m_3 + m_5 + m_6 \\
f_2 &= m_0 + m_3 + m_5 + m_6
\end{aligned}$$

3.44.



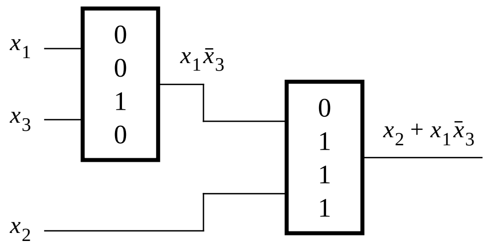
3.45. The canonical SOP for f is

$$f = \overline{x}_1x_2\overline{x}_3 + \overline{x}_1x_2x_3 + x_1\overline{x}_2\overline{x}_3 + x_1x_2\overline{x}_3 + x_1x_2x_3$$

This expression can be manipulated into

$$\begin{aligned} f &= \overline{x}_1x_2 + x_1\overline{x}_3 + x_1x_2 \\ &= x_2 + x_1\overline{x}_3 \end{aligned}$$

The circuit is



3.46. The canonical SOP for f is

$$f = x_1x_2x_4 + x_2x_3\overline{x}_4 + \overline{x}_1\overline{x}_2\overline{x}_3$$

This expression can be manipulated into

$$f = x_2 \cdot (x_1x_4 + x_3\overline{x}_4) + \overline{x}_2 \cdot (\overline{x}_1\overline{x}_3)$$

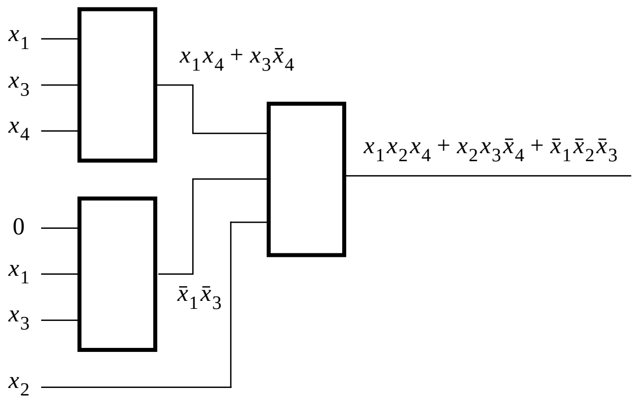
Using functional decomposition we have

$$f = x_2f_1 + \overline{x}_2f_2$$

where

$$\begin{aligned} f_1 &= x_1x_4 + x_3\overline{x}_4 \\ f_2 &= \overline{x}_1\overline{x}_3 \end{aligned}$$

The circuit is



3.47. The canonical SOP for f is

$$f = x_1x_2x_4 + x_2x_3\overline{x}_4 + \overline{x}_1\overline{x}_2\overline{x}_3$$

This expression can be manipulated into

$$f = x_2 \cdot (x_1x_4 + x_3\overline{x}_4) + \overline{x}_2 \cdot (\overline{x}_1\overline{x}_3)$$

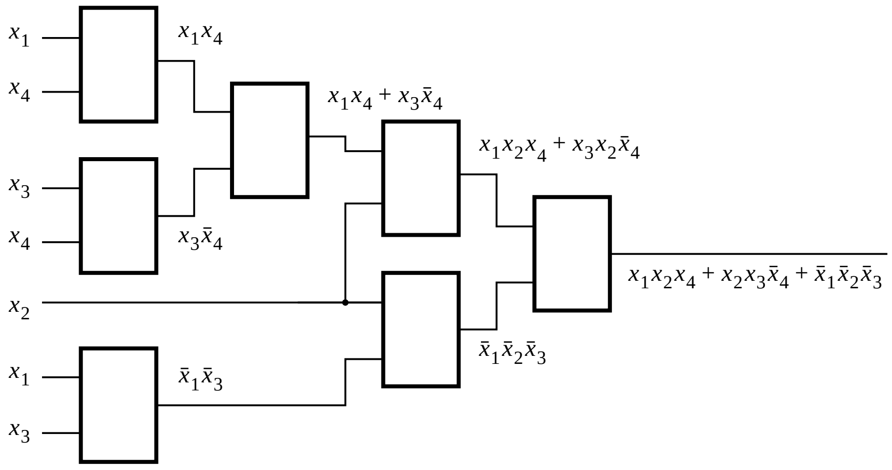
Using functional decomposition we have

$$f = x_2f_1 + \overline{x}_2f_2$$

where

$$\begin{aligned} f_1 &= x_1x_4 + x_3\overline{x}_4 \\ f_2 &= \overline{x}_1\overline{x}_3 \end{aligned}$$

The function f_1 requires one 2-LUT, while f_2 requires three 2-LUTs. We then need three additional 3-LUTs to realize f, as illustrated in the circuit

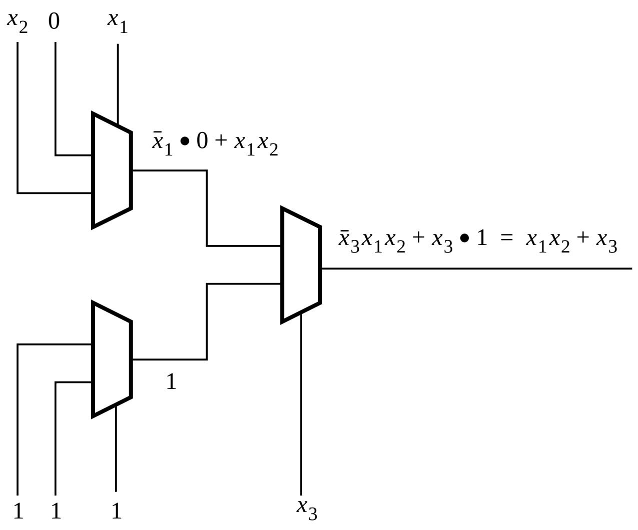


x_1
x_4
x_1x_4
$x_1x_4 + x_3\bar{x}_4$
$x_1x_2x_4 + x_3x_2\bar{x}_4$
x_3
x_4
$x_3\bar{x}_4$
x_2
$x_1x_2x_4 + x_2x_3\bar{x}_4 + \bar{x}_1\bar{x}_2\bar{x}_3$
$\bar{x}_1\bar{x}_2\bar{x}_3$
x_1
$\bar{x}_1\bar{x}_3$
x_3

3.48.

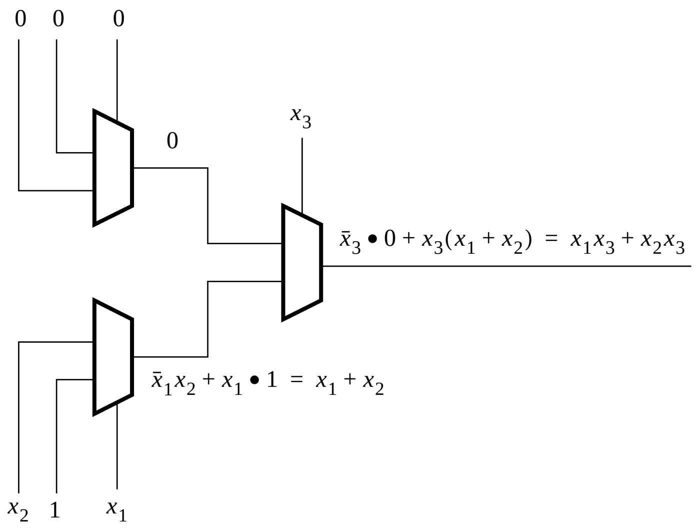
$$\begin{aligned} g &= \overline{x}_2x_3 \\ h &= x_1 \\ j &= x_2 \\ k &= x_3 \end{aligned}$$

3.49. (*a*)



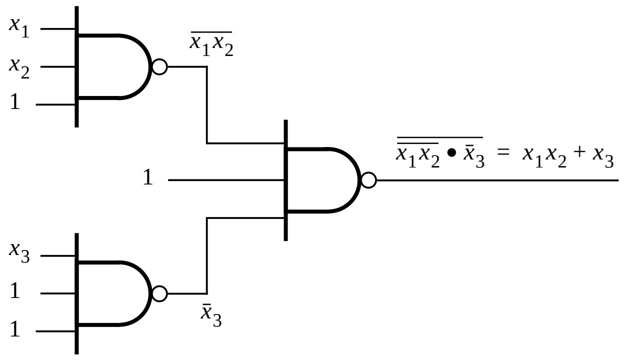
x_2
0
x_1
$\bar{x}_1 \bullet 0 + x_1x_2$
$\bar{x}_3x_1x_2 + x_3 \bullet 1 = x_1x_2 + x_3$
1
1
1
1
x_3

(*b*)



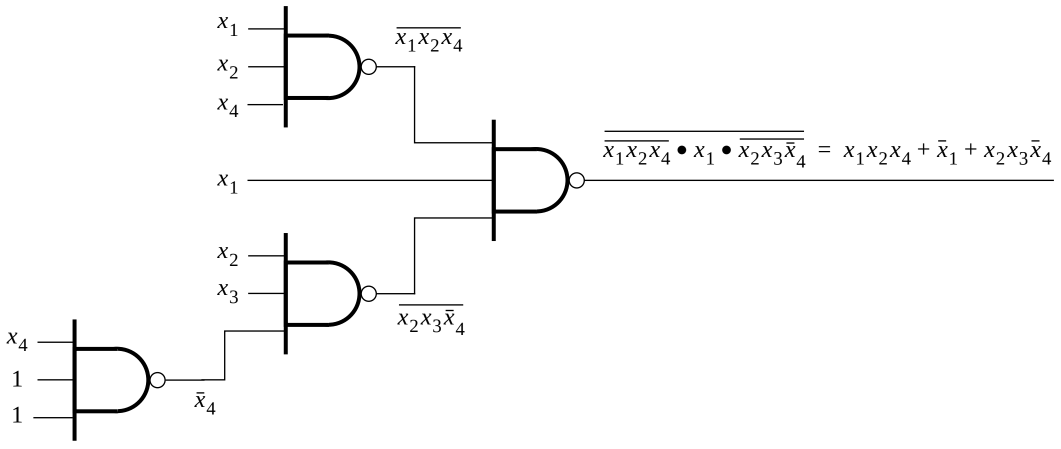
0
0
0
0
x_3
$\bar{x}_3 \bullet 0 + x_3(x_1 + x_2) = x_1x_3 + x_2x_3$
$\bar{x}_1x_2 + x_1 \bullet 1 = x_1 + x_2$
x_2
1
x_1

3.50. (*a*)



x_1
x_2
1
$\overline{x_1x_2}$
1
$\overline{\overline{x_1x_2} \bullet \bar{x}_3} = x_1x_2 + x_3$
x_3
1
1
$\bar{x}_3$

(*b*)



x_1
x_2
x_4
$\overline{x_1x_2x_4}$
x_1
$\overline{\overline{x_1x_2x_4} \bullet x_1 \bullet \overline{x_2x_3\bar{x}_4}} = x_1x_2x_4 + \bar{x}_1 + x_2x_3\bar{x}_4$
x_2
x_3
$\overline{x_2x_3\bar{x}_4}$
x_4
1
1
$\bar{x}_4$

3.51.
```
module prob2_51 (x1, x2, x3, x4, f);
    input x1, x2, x3, x4;
    output f;

    assign f = (x2 & ~x3 & ~x4) | (~x1 & x2 & x4) | (~x1 & x2 & x3) | (x1 & x2 & x3);

endmodule
```

3.52.
```
module prob2_52 (x1, x2, x3, x4, f);
    input x1, x2, x3, x4;
    output f;

    assign f = (x1 | x2 | ~x4) & (~x2 | x3 | ~x4) & (~x1 | x3 | ~x4) & (~x1 | ~x3 | ~x4);

endmodule
```

3.53.
```
module prob2_53 (x1, x2, x3, x4, x5, x6, x7, f);
    input x1, x2, x3, x4, x5, x6, x7;
    output f;

    assign f = (x1 & x3 & ~x6) | (x1 & x4 & x5 & ~x6) | (x2 & x3 & x7) | (x2 & x4 & x5 & x7);

endmodule
```

3.54. The circuit in Figure P3.10 is a two-input XOR gate. Since NMOS transistors are used only to pass logic 0 and PMOS transistors are used only to pass logic 1, the circuit does nor suffer from any major drawbacks.

3.55. The circuit in Figure P3.11 is a two-input XOR gate. This circuit has two drawbacks: when both inputs are 0 the PMOS transistor must drive f to 0, resulting in $f = V_T$ volts. Also, when $x_1 = 1$ and $x_2 = 0$, the NMOS transistor must drive the output high, resulting in $f = V_{DD} - V_T$.

Chapter 4

4.1. SOP form: $f = \overline{x}_1 x_2 + \overline{x}_2 x_3$
POS form: $f = (\overline{x}_1 + \overline{x}_2)(x_2 + x_3)$

4.2. SOP form: $f = x_1 \overline{x}_2 + x_1 x_3 + \overline{x}_2 x_3$
POS form: $f = (x_1 + x_3)(x_1 + \overline{x}_2)(\overline{x}_2 + x_3)$

4.3. SOP form: $f = \overline{x}_1 x_2 x_3 \overline{x}_4 + x_1 x_2 \overline{x}_3 x_4 + \overline{x}_2 x_3 x_4$
POS form: $f = (\overline{x}_1 + x_4)(x_2 + x_3)(\overline{x}_2 + \overline{x}_3 + \overline{x}_4)(x_2 + x_4)(x_1 + x_3)$

4.4. SOP form: $f = \overline{x}_2 \overline{x}_3 + \overline{x}_2 \overline{x}_4 + x_2 x_3 x_4$
POS form: $f = (\overline{x}_2 + x_3)(x_2 + \overline{x}_3 + \overline{x}_4)(\overline{x}_2 + x_4)$

4.5. SOP form: $f = \overline{x}_3 \overline{x}_5 + \overline{x}_3 x_4 + x_2 x_4 \overline{x}_5 + \overline{x}_1 x_3 \overline{x}_4 x_5 + x_1 x_2 \overline{x}_4 x_5$
POS form: $f = (\overline{x}_3 + x_4 + x_5)(\overline{x}_3 + \overline{x}_4 + \overline{x}_5)(x_2 + \overline{x}_3 + \overline{x}_4)(x_1 + x_3 + x_4 + \overline{x}_5)(\overline{x}_1 + x_2 + x_4 + \overline{x}_5)$

4.6. SOP form: $f = \overline{x}_2 x_3 + \overline{x}_1 x_5 + \overline{x}_1 x_3 + \overline{x}_3 \overline{x}_4 + \overline{x}_2 x_5$
POS form: $f = (\overline{x}_1 + \overline{x}_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_4)(x_3 + \overline{x}_4 + x_5)$

4.7. SOP form: $f = x_3 \overline{x}_4 \overline{x}_5 + \overline{x}_3 \overline{x}_4 x_5 + x_1 x_4 x_5 + x_1 x_2 x_4 + x_3 x_4 x_5 + \overline{x}_2 x_3 x_4 + x_2 \overline{x}_3 x_4 \overline{x}_5$
POS form: $f = (x_3 + x_4 + x_5)(\overline{x}_3 + x_4 + \overline{x}_5)(x_1 + \overline{x}_2 + \overline{x}_3 + \overline{x}_4 + x_5)$

4.8. $f = \sum m(0,7)$
$f = \sum m(1,6)$
$f = \sum m(2,5)$
$f = \sum m(0,1,6)$
$f = \sum m(0,2,5)$
etc.

4.9. $f = x_1 x_2 x_3 + x_1 x_2 x_4 + x_1 x_3 x_4 + x_2 x_3 x_4$

4.10. SOP form: $f = x_1 x_2 \overline{x}_3 + x_1 \overline{x}_2 x_4 + x_1 x_3 \overline{x}_4 + \overline{x}_1 x_2 x_3 + \overline{x}_1 x_3 x_4 + x_2 \overline{x}_3 x_4$
POS form: $f = (x_1 + x_2 + x_3)(x_1 + x_2 + x_4)(x_1 + x_3 + x_4)(x_2 + x_3 + x_4)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3 + \overline{x}_4)$
The POS form has lower cost.

4.11. The statement is false. As a counter example consider $f(x_1, x_2, x_3) = \sum m(0,5,7)$.
Then, the minimum-cost SOP form $f = x_1 x_3 + \overline{x}_1 \overline{x}_2 \overline{x}_3$ is unique.
But, there are two minimum-cost POS forms:
$f = (x_1 + \overline{x}_3)(\overline{x}_1 + x_3)(x_1 + \overline{x}_2)$ and
$f = (x_1 + \overline{x}_3)(\overline{x}_1 + x_3)(\overline{x}_2 + x_3)$

4.12. If each circuit is implemented separately:
$f = \overline{x}_1\overline{x}_4 + \overline{x}_1x_2x_3 + x_1\overline{x}_2x_4$ Cost$= 15$
$g = \overline{x}_1\overline{x}_3\overline{x}_4 + \overline{x}_2x_3\overline{x}_4 + x_1\overline{x}_3x_4 + x_1x_2x_4$ Cost $= 21$

In a combined circuit:
$f = \overline{x}_2x_3\overline{x}_4 + \overline{x}_1\overline{x}_3\overline{x}_4 + x_1\overline{x}_2\overline{x}_3x_4 + \overline{x}_1x_2x_3$
$g = \overline{x}_2x_3\overline{x}_4 + \overline{x}_1\overline{x}_3\overline{x}_4 + x_1\overline{x}_2\overline{x}_3x_4 + x_1x_2x_4$
The first 3 product terms are shared, hence the total cost is 31.

4.13. If each circuit is implemented separately:
$f = \overline{x}_1x_2x_4 + x_2x_4x_5 + x_3\overline{x}_4\overline{x}_5 + \overline{x}_1\overline{x}_2\overline{x}_4x_5$ Cost $= 22$
$g = \overline{x}_3\overline{x}_5 + \overline{x}_4\overline{x}_5 + \overline{x}_1\overline{x}_2\overline{x}_4 + \overline{x}_1x_2x_4 + x_2x_4x_5$ Cost $= 24$

In a combined circuit:
$f = \overline{x}_1x_2x_4 + x_2x_4x_5 + x_3\overline{x}_4\overline{x}_5 + \overline{x}_1\overline{x}_2\overline{x}_4x_5$
$g = \overline{x}_1x_2x_4 + x_2x_4x_5 + x_3\overline{x}_4\overline{x}_5 + \overline{x}_1\overline{x}_2\overline{x}_4x_5 + \overline{x}_3\overline{x}_5$

The first 4 product terms are shared, hence the total cost is 31. Note that in this implementation $f \subseteq g$, thus g can be realized as $g = f + \overline{x}_3\overline{x}_5$, in which case the total cost is lowered to 28.

4.14. $f = (x_3 \uparrow g) \uparrow ((g \uparrow g) \uparrow x_4)$ where $g = (x_1 \uparrow (x_2 \uparrow x_2)) \uparrow ((x_1 \uparrow x_1) \uparrow x_2)$

4.15. $\overline{f} = (((x_3 \downarrow x_3) \downarrow g) \downarrow ((g \downarrow g) \downarrow (x_4 \downarrow x_4))$, where
$g = ((x_1 \downarrow x_1) \downarrow x_2) \downarrow (x_1 \downarrow (x_2 \downarrow x_2))$. Then, $f = \overline{f} \downarrow \overline{f}$.

4.16. $f = (g \uparrow k) \uparrow ((g \uparrow g) \uparrow (k \uparrow k))$, where $g = (x_1 \uparrow x_1) \uparrow (x_2 \uparrow x_2) \uparrow (x_5 \uparrow x_5)$
and $k = (x_3 \uparrow (x_4 \uparrow x_4)) \uparrow ((x_3 \uparrow x_3) \uparrow x_4)$

4.17. $\overline{f} = (g \downarrow k) \downarrow ((g \downarrow g) \downarrow (k \downarrow k))$, where $g = x_1 \downarrow x_2 \downarrow x_5$
and $k = ((x_3 \downarrow x_3) \downarrow x_4) \downarrow (x_3 \downarrow (x_4 \downarrow x_4))$. Then, $f = \overline{f} \downarrow \overline{f}$.

4.18. $f = \overline{x}_1(x_2 + x_3)(x_4 + x_5) + x_1(\overline{x}_2 + x_3)(\overline{x}_4 + x_5)$

4.19. $f = x_1\overline{x}_3\overline{x}_4 + x_2\overline{x}_3\overline{x}_4 + x_1x_3x_4 + x_2x_3x_4 = (x_1 + x_2)\overline{x}_3\overline{x}_4 + (x_1 + x_2)x_3x_4$
This requires 2 OR and 2 AND gates.

4.20. $f = x_1 \cdot g + \overline{x}_1 \cdot \overline{g}$, where $g = \overline{x}_3x_4 + x_3\overline{x}_4$

4.21 $f = g \cdot h + \overline{g} \cdot \overline{h}$, where $g = x_1x_2$ and $h = x_3 + x_4$

4.22. Let $D(0, 20)$ be 0 and $D(15, 26)$ be 1. Then decomposition yields:
$g = x_5(\overline{x}_1 + x_2)$
$f = (\overline{x}_3\overline{x}_4 + x_3x_4)g + \overline{x}_3x_4\overline{g}$ = $x_3x_4g + \overline{x}_3\overline{x}_4g + \overline{x}_3x_4\overline{g}$
Cost $= 9 + 18 = 27$

The optimal SOP form is:
$f = \overline{x}_3 x_4 \overline{x}_5 + \overline{x}_1 x_3 x_4 x_5 + x_1 \overline{x}_2 \overline{x}_3 x_4 + \overline{x}_1 \overline{x}_3 \overline{x}_4 x_5 + x_2 \overline{x}_3 \overline{x}_4 x_5 + x_2 x_3 x_4 x_5$
Cost $= 7 + 29 = 36$

4.23. The prime implicants are generated as follows:

List 1

0	0 0 0 0	√
2	0 0 1 0	√
4	0 1 0 0	√
8	1 0 0 0	√
5	0 1 0 1	√
9	1 0 0 1	√
7	0 1 1 1	√
15	1 1 1 1	√

List 2

0,2	0 0 x 0
0,4	0 x 0 0
0,8	x 0 0 0
4,5	0 1 0 x
8,9	1 0 0 x
5,7	0 1 x 1
7,15	x 1 1 1

The initial prime implicant table is

Prime implicant	Minterm 0	2	4	5	7	8	9	15
p_1 = 0 0 x 0	√	√						
p_2 = 0 x 0 0	√		√					
p_3 = x 0 0 0	√					√		
p_4 = 0 1 0 x			√	√				
p_5 = 1 0 0 x						√	√	
p_6 = 0 1 x 1				√	√			
p_7 = x 1 1 1					√			√

The prime implicants p_1, p_5 and p_7 are essential. Removing these prime implicants gives

Prime implicant	Minterm 4	5
p_2	√	
p_3		
p_4	√	√
p_6		√

Since p_4 covers both minterms, the final cover is

$$\begin{aligned} C &= \{p_1, p_4, p_5, p_7\} \\ &= \{00\text{x}0, 010\text{x}, 100\text{x}, \text{x}111\} \end{aligned}$$

and the function is implemented as

$$f = \overline{x}_1\overline{x}_2\overline{x}_4 + \overline{x}_1x_2\overline{x}_3 + x_1\overline{x}_2\overline{x}_3 + x_2x_3x_4$$

4.24. The prime implicants are generated as follows:

List 1

0	0 0 0 0	✓
4	0 1 0 0	✓
8	1 0 0 0	✓
3	0 0 1 1	✓
6	0 1 1 0	✓
9	1 0 0 1	✓
7	0 1 1 1	✓
11	1 0 1 1	✓
13	1 1 0 1	✓
15	1 1 1 1	✓

List 2

0,4	0 x 0 0	
0,8	x 0 0 0	
4,6	0 1 x 0	
8,9	1 0 0 x	
3,7	0 x 1 1	✓
3,11	x 0 1 1	✓
6,7	0 1 1 x	
9,11	1 0 x 1	✓
9,13	1 x 0 1	✓
7,15	x 1 1 1	✓
11,15	1 x 1 1	✓
13,15	1 1 x 1	✓

List 3

3,7,11,15	x x 1 1
9,11,13,15	1 x x 1

The initial prime implicant table is

Prime implicant	Minterm 0	4	6	8	9	15
p_1 = 0 x 0 0	✓	✓				
p_2 = x 0 0 0	✓			✓		
p_3 = 0 1 x 0		✓	✓			
p_4 = 1 0 0 x				✓	✓	
p_5 = 0 1 1 x			✓			
p_6 = x x 1 1						✓
p_7 = 1 x x 1					✓	✓

There are no essential prime implicants. Prime implicant p_3 dominates p_5 and their costs are the same, so remove p_5. Similarly, p_7 dominates p_6, so remove p_6. This gives

Prime implicant	Minterm 0	4	6	8	9	15
p_1	✓	✓				
p_2	✓			✓		
p_3		✓	✓			
p_4				✓	✓	
p_7					✓	✓

Now, p_3 and p_7 are essential, which leaves

Prime implicant	Minterm 0	8
p_1	✓	
p_2	✓	✓
p_4		✓

Choosing p_2 results in the minimum cost cover

$$\begin{aligned} C &= \{p_2, p_3, p_7\} \\ &= \{\text{x000}, \text{01x0}, \text{1xx1}\} \end{aligned}$$

and the function is implemented as

$$f = \overline{x}_2\overline{x}_3\overline{x}_4 + \overline{x}_1 x_2 \overline{x}_4 + x_1 x_4$$

4.25. The prime implicants are generated as follows:

List 1

0	0 0 0 0	✓
4	0 1 0 0	✓
8	1 0 0 0	✓
3	0 0 1 1	✓
5	0 1 0 1	✓
9	1 0 0 1	✓
12	1 1 0 0	✓
7	0 1 1 1	✓
11	1 0 1 1	✓
13	1 1 0 1	✓
14	1 1 1 0	✓

List 2

0,4	0 x 0 0	✓
0,8	x 0 0 0	
4,5	0 1 0 x	✓
4,12	x 1 0 0	✓
8,9	1 0 0 x	✓
8,12	1 x 0 0	✓
3,7	0 x 1 1	
3,11	x 0 1 1	
5,7	0 1 x 1	
5,13	x 1 0 1	✓
9,11	1 0 x 1	
9,13	1 x 0 1	✓
12,13	1 1 0 x	✓
12,14	1 1 x 0	

List 3

0,4,8,12	x x 0 0
4,5,12,13	x 1 0 x
8,9,12,13	1 x 0 x

The initial prime implicant table is

Prime	Minterm						
implicant	0	3	4	5	7	9	11
p_1 = 0 x 1 1		√			√		
p_2 = x 0 1 1		√					√
p_3 = 0 1 x 1				√	√		
p_4 = 1 0 x 1						√	√
p_5 = x x 0 0	√		√				
p_6 = x 1 0 x			√	√			
p_7 = 1 x 0 x						√	
p_8 = 1 1 x 0							

Prime implicant p_5 is essential, so remove columns 0 and 4 to get

Prime	Minterm				
implicant	3	5	7	9	11
p_1	√		√		
p_2	√				√
p_3		√	√		
p_4				√	√
p_6		√			
p_7				√	

Here, p_3 dominates p_6, and p_4 dominates p_7; but costs of p_3 and p_4 are greater than the costs of p_6 and p_7, respectively. So, use branching. First choose p_3 to be in the final cover, which leads to

Prime	Minterm		
implicant	3	9	11
p_1	√		
p_2	√		√
p_4		√	√
p_6			
p_7		√	

Now, choose p_2 and p_7 (lower cost than p_4) to cover the remaining minterms. The resulting cover is

$$\begin{aligned} C &= \{p_2, p_3, p_5, p_7\} \\ &= \{\text{x011}, \text{01x1}, \text{xx00}, \text{1x0x}\} \end{aligned}$$

Next, assume that p_3 is not included in the final cover, which leads to

Prime implicant	Minterm 3	5	7	9	11
p_1	✓		✓		
p_2	✓				✓
p_4				✓	✓
p_6		✓			
p_7				✓	

Then p_6 is essential. Also, column 3 dominates 7, hence remove 3 giving

Prime implicant	Minterm 7	9	11
p_1	✓		
p_2			✓
p_4		✓	✓
p_7		✓	

Choose p_1 and p_4, which results in the cover

$$\begin{aligned} C &= \{p_1, p_4, p_5, p_6\} \\ &= \{0\text{x}11, 10\text{x}1, \text{xx}00, \text{x}10\text{x}\} \end{aligned}$$

Both covers have the same cost, so choosing the first cover the function can be implemented as

$$f = \overline{x}_2 x_3 x_4 + \overline{x}_1 x_2 x_4 + \overline{x}_3 \overline{x}_4 + x_1 \overline{x}_3$$

Observe that if we had not taken the cost of prime implicants (rows) into account and pursued the dominance of p_3 over p_6 and p_4 over p_7, then we would have removed p_6 and p_7 giving the following table

Prime implicant	Minterm 3	5	7	9	11
p_1	✓		✓		
p_2	✓				✓
p_3		✓	✓		
p_4				✓	✓

Now p_3 and p_4 are essential. Also choose p_1, so that

$$\begin{aligned} C &= \{p_1, p_3, p_4, p_5\} \\ &= \{0\text{x}11, 01\text{x}1, 10\text{x}1, \text{xx}00\} \end{aligned}$$

The cost of this cover is greater by one literal compared to both covers derived above.

4.26. Note that $X \# Y = X \cdot \overline{Y}$. Therefore,

$$\begin{aligned}(A \cdot B)\#C &= A \cdot B \cdot \overline{c}\\ (A\#C) \cdot (B\#C) &= A \cdot \overline{C} \cdot B \cdot \overline{C}\\ &= A \cdot B \cdot \overline{C}\end{aligned}$$

Similarly,

$$\begin{aligned}(A + B)\#C &= (A + B) \cdot \overline{C}\\ &= A \cdot \overline{C} + B \cdot \overline{C}\\ (A\#C) + (B\#C) &= A \cdot \overline{C} + B \cdot \overline{C}\end{aligned}$$

4.27. The initial cover is $C^0 = \{0000, 0011, 0100, 0101, 0111, 1000, 1001, 1111\}$.
Using the $*$-product get the prime implicants
P = {00x0, 0x00, x000, 010x, 01x1, 100x, x111}.
The minimum cover is $C_{minimum}$ = {00x0, 010x, 100x, x111}, which corresponds to $f = \overline{x}_1\overline{x}_2\overline{x}_4 + \overline{x}_1x_2\overline{x}_3 + x_1\overline{x}_2\overline{x}_3 + x_2x_3x_4$.

4.28. The initial cover is C^0 = {0x0x0, 110xx, x1101, 1001x, 11110, 01x10, 0x011}.
Using the $*$-product get the prime implicants
P = {0x0x0, xx01x, x1x10, 110xx, x10x0, 11x01, x1101}.
The minimum cover is $C_{mimimum}$ = {0x0x0, xx01x, x1x10, 110xx, x1101}, which corresponds to $f = \overline{x}_1\overline{x}_3\overline{x}_5 + \overline{x}_3x_4 + x_2x_4\overline{x}_5 + x_1x_2\overline{x}_3 + x_2x_3\overline{x}_4x_5$.

4.29. The initial cover is C^0 = {00x0, 100x, x010, 1111, 00x1, 011x}.
Using the $*$-product get the prime implicants P = {00xx, 0x1x, x00x, x0x0, x111}.
The minimum-cost cover is $C_{minimum}$ = {x00x, x0x0, x111}, which corresponds to $f = \overline{x}_2\overline{x}_3 + \overline{x}_2\overline{x}_4 + x_2x_3x_4$.

4.30. Expansion of $\overline{x}_1\overline{x}_2\overline{x}_3$ gives $\overline{x}_1$.
Expansion of $\overline{x}_1\overline{x}_2x_3$ gives $\overline{x}_1$.
Expansion of $\overline{x}_1x_2\overline{x}_3$ gives $\overline{x}_1$.
Expansion of $x_1x_2x_3$ gives x_2x_3.
The set of prime implicants comprises $\overline{x}_1$ and x_2x_3.

4.31. Expansion of $\overline{x}_1x_2\overline{x}_3x_4$ gives $x_2\overline{x}_3x_4$ and $\overline{x}_1x_2x_4$.
Expansion of $x_1x_2\overline{x}_3x_4$ gives $x_2\overline{x}_3x_4$.
Expansion of $x_1x_2x_3\overline{x}_4$ gives $x_3\overline{x}_4$.
Expansion of $\overline{x}_1x_2x_3$ gives $\overline{x}_1x_3$.
Expansion of $\overline{x}_2x_3$ gives $\overline{x}_2x_3$.
The set of prime implicants comprises $x_2\overline{x}_3x_4$, $\overline{x}_1x_2x_4$, $x_3\overline{x}_4$, $\overline{x}_1x_3$, and $\overline{x}_2x_3$.

4.32. Representing both functions in the form of Karnaugh map, it is easy to show that $f = g$. The minimum cost SOP expression is
$f = g = \overline{x}_2\overline{x}_3\overline{x}_5 + \overline{x}_2x_3\overline{x}_4 + x_1x_3x_4 + x_1x_2x_4x_5$.

4.33. The cost of the circuit in Figure P4.2 is 11 gates and 30 inputs, for a total of 41. The functions implemented by the circuit can also be realized as

$$\begin{aligned} f &= \overline{x}_1\overline{x}_2\overline{x}_4 + x_2\overline{x}_3\overline{x}_4 + \overline{x}_1x_3x_4 + x_1x_4 \\ g &= \overline{x}_1\overline{x}_2\overline{x}_4 + x_2\overline{x}_3\overline{x}_4 + \overline{x}_1x_3x_4 + \overline{x}_2x_4 + x_3\overline{x}_4 \end{aligned}$$

The first three product terms in f and g are the same; therefore, they can be shared. Then, the cost of implementing f and g is 8 gates and 24 inputs, for a total of 32.

4.34. The cost of the circuit in Figure P4.3 is 11 gates and 26 inputs, for a total of 37. The functions implemented by the circuit can also be realized as

$$\begin{aligned} f &= (\overline{x}_2 \uparrow x_4) \uparrow (\overline{x}_1 \uparrow x_2 \uparrow x_3) \uparrow (x_1 \uparrow \overline{x}_2 \uparrow x_3) \uparrow (\overline{x}_2 \uparrow \overline{x}_3) \\ g &= (\overline{x}_2 \uparrow x_4) \uparrow (\overline{x}_1 \uparrow x_2 \uparrow x_3) \uparrow (x_1 \uparrow \overline{x}_2 \uparrow x_3) \uparrow (\overline{x}_1 \uparrow \overline{x}_1) \end{aligned}$$

The first three NAND terms in f and g are the same; therefore, they can be shared. Then, the cost of implementing f and g is 7 gates and 20 inputs, for a total of 27.

4.35. Using gate level primitives, the circuit in Figure 4.25*b* can be implemented using the code

```
module prob4_35 (x1, x2, x3, x4, x5, f);
   input x1, x2, x3, x4, x5;
   output f;

   or (g, x1, x2, x5);
   not (notx3, x3);
   not (notx4, x4);
   and (a, x3, notx4);
   and (b, notx3, x4);
   or (k, a, b);
   and (c, g, k);
   not (notg, g);
   not (notk, k);
   and (d, notg, notk);
   or (f, c, d);

endmodule
```

4.36. Using continuous assignment, the circuit in Figure 4.25*b* can be implemented using the code

```
module prob4_36 (x1, x2, x3, x4, x5, f);
    input x1, x2, x3, x4, x5;
    output f;
    wire g, k;

    assign g = (x1 | x2 | x5);
    assign k = (x3 & ~x4) | (~x3 & x4);
    assign f = (g & k) | (~g & ~k);

endmodule
```

4.37. Using gate level primitives, the circuit in Figure 4.27*c* can be implemented using the code

```
module prob4_37 (x1, x2, x3, x4, x5, x6, x7, f);
    input x1, x2, x3, x4, x5, x6, x7;
    output f;

    nand (a, x1, x1);
    nand (b, x2, x3);
    nand (c, a, b);
    nand (d, x5, x5);
    nand (e, x6, x6);
    nand (g, d, e);
    nand (h, x4, g);
    nand (j, x7, x7);
    nand (k, h, j);
    nand (m, c, k);
    nand (f, m, m);

endmodule
```

4.38. Using continuous assignment, the circuit in Figure 4.27*c* can be implemented using the code

```
module prob4_38 (x1, x2, x3, x4, x5, x6, x7, f);
    input x1, x2, x3, x4, x5, x6, x7;
    output f;
    wire a, b;

    assign a = ~(~x1 & ~(x2 & x3));
    assign b = ~(~(x4 & ~(~x5 & ~x6)) & ~x7);
    assign f = ~(~(a & b));

endmodule
```

4.39. Using gate level primitives, the circuit in Figure 4.28*b* can be implemented using the code

```
module  prob4_39 (x1, x2, x3, x4, x5, x6, x7, f);
    input  x1, x2, x3, x4, x5, x6, x7;
    output  f;

    nor (a, x2, x2);
    nor (b, x3, x3);
    nor (c, a, b);
    nor (d, x1, c);
    nor (e, x4, x4);
    nor (g, x5, x6);
    nor (h, e, g);
    nor (k, h, x7);
    nor (f, d, k);

endmodule
```

4.40. Using continuous assignment, the circuit in Figure 4.27*c* can be implemented using the code

```
module  prob4_40 (x1, x2, x3, x4, x5, x6, x7, f);
    input  x1, x2, x3, x4, x5, x6, x7;
    output  f;
    wire a, b;

    assign a = ~(x1 | ~(~x2 | ~x3));
    assign b = ~(~(~x4 | ~(x5 | x6)) | x7);
    assign f = ~(a | b);

endmodule
```

4.41. Using the POS expression

$$f = (x_1 + x_2 + \overline{x}_3 + \overline{x}_4)(x_1 + \overline{x}_2 + \overline{x}_3 + x_4)(\overline{x}_1 + x_2 + \overline{x}_3 + x_4)(\overline{x}_1 + \overline{x}_2 + x_3 + \overline{x}_4)$$

the function can be implemented using the code

```
module prob4_41 (x1, x2, x3, x4, f);
    input x1, x2, x3, x4;
    output f;

    not (notx1, x1);
    not (notx2, x2);
    not (notx3, x3);
    not (notx4, x4);
    or (a, x1, x2, notx3, notx4);
    or (b, x1, notx2, notx3, x4);
    or (c, notx1, x2, notx3, x4);
    or (d, notx1, notx2, x3, notx4);
    and (f, a, b, c, d);

endmodule
```

4.42. Using the POS expression

$$f = (x_1 + x_2 + \overline{x}_3 + \overline{x}_4)(x_1 + \overline{x}_2 + \overline{x}_3 + x_4)(\overline{x}_1 + x_2 + \overline{x}_3 + x_4)(\overline{x}_1 + \overline{x}_2 + x_3 + \overline{x}_4)$$

the function can be implemented using the code

```
module prob4_42 (x1, x2, x3, x4, f);
    input x1, x2, x3, x4;
    output f;

    assign f = (x1 | x2 | ~x3 | ~x4) & (x1 | ~x2 | ~x3 | x4) &
               (~x1 | x2 | ~x3 | x4) & (~x1 | ~x2 | x3 | ~x4);

endmodule
```

4.43. The simplest expression is

$$f = \overline{x}_1\overline{x}_3 + x_2x_3(x_1 + x_4)$$

which can be implemented using the code

```
module prob4_43 (x1, x2, x3, x4, f);
    input x1, x2, x3, x4;
    output f;

    not (notx1, x1);
    not (notx3, x3);
    and (a, notx1, notx3);
    or (b, x1, x4);
    and (c, x2, x3, b);
    or (f, a, c);

endmodule
```

4.44. The simplest expression is

$$f = \overline{x}_1\overline{x}_3 + x_2x_3(x_1 + x_4)$$

which can be implemented using the code

```
module prob4_44 (x1, x2, x3, x4, f);
   input x1, x2, x3, x4;
   output f;

   assign f = (~x1 & ~x3) | (x2 & x3 & (x1 | x4));
endmodule
```

4.45. The simplest expression is

$$f = (\overline{x}_1 + x_3)(x_1 + \overline{x}_2 + \overline{x}_3 + x_4)$$

which can be implemented using the code

```
module prob4_45 (x1, x2, x3, x4, f);
   input x1, x2, x3, x4;
   output f;

   not (notx1, x1);
   not (notx2, x2);
   not (notx3, x3);
   or (a, notx1, x3);
   or (b, x1, notx2, notx3, x4);
   and (f, a, b);

endmodule
```

4.46. The simplest expression is

$$f = (\overline{x}_1 + x_3)(x_1 + \overline{x}_2 + \overline{x}_3 + x_4)$$

which can be implemented using the code

```
module prob4_46 (x1, x2, x3, x4, f);
   input x1, x2, x3, x4;
   output f;

   assign f = (~x1 | x3) & (x1 | ~x2 | ~x3 | x4);

endmodule
```

4.47. The simplest expression is

$$f = (x_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_3 + x_4)$$

which can be implemented using the code

```
module prob4_47 (x1, x2, x3, x4, f);
    input x1, x2, x3, x4;
    output f;

    not (notx1, x1);
    not (notx3, x3);
    or (a, x2, notx3);
    or (b, notx1, notx3, x4);
    and (f, a, b);

endmodule
```

4.48. The simplest expression is

$$f = (x_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_3 + x_4)$$

which can be implemented using the code

```
module prob4_47 (x1, x2, x3, x4, f);
    input x1, x2, x3, x4;
    output f;

    assign f = (x2 | ~x3) & (~x1 | ~x3 | x4);
endmodule
```

Chapter 5

5.1. (*a*) 478
(*b*) 743
(*c*) 2025
(*d*) 41567
(*e*) 61680

5.2. (*a*) 478
(*b*) −280
(*c*) −1

5.3. (*a*) 478
(*b*) −281
(*c*) −2

5.4. The numbers are represented as follows:

Decimal	Sign and Magnitude	1's Complement	2's Complement
73	000001001001	000001001001	000001001001
1906	011101110010	011101110010	011101110010
−95	100001011111	111110100000	111110100001
−1630	111001011110	100110100001	100110100010

5.5. The results of the operations are:

$$(a):\quad \begin{array}{r} 00110110 \\ \underline{+01000101} \\ 01111011 \end{array} \quad \begin{array}{r} 54 \\ \underline{+69} \\ 123 \end{array} \qquad (b):\quad \begin{array}{r} 01110101 \\ \underline{+11011110} \\ 01010011 \end{array} \quad \begin{array}{r} 117 \\ \underline{-\ 34} \\ 83 \end{array} \qquad (c):\quad \begin{array}{r} 11011111 \\ \underline{+10111000} \\ 10010111 \end{array} \quad \begin{array}{r} (-33) \\ \underline{+(-72)} \\ (-105) \end{array}$$

$$(d):\quad \begin{array}{r} 00110110 \\ \underline{-00101011} \\ 00001011 \end{array} \quad \begin{array}{r} 54 \\ \underline{-43} \\ 11 \end{array} \qquad (e):\quad \begin{array}{r} 01110101 \\ \underline{-11010110} \\ 10011111 \end{array} \quad \begin{array}{r} (117) \\ \underline{-(-\ 42)} \\ (159) \end{array} \qquad (f):\quad \begin{array}{r} 11010011 \\ \underline{-11101100} \\ 11100111 \end{array} \quad \begin{array}{r} (-45) \\ \underline{-(-20)} \\ (-25) \end{array}$$

Arithmetic overflow occurs in example *e*; note that the pattern 10011111 represents −97 rather than +159.

5.6. The associativity of the XOR operation can be shown as follows:

$$\begin{aligned} x \oplus (y \oplus z) &= x \oplus (\overline{y}z + y\overline{z}) \\ &= \overline{x}(\overline{y}z + y\overline{z}) + x(\overline{y} \cdot \overline{z} + yz) \\ &= \overline{x} \cdot \overline{y}z + \overline{x}y\overline{z} + x\overline{y} \cdot \overline{z} + xyz \end{aligned}$$

$$\begin{aligned} (x \oplus y) \oplus z &= (\overline{x}y + x\overline{y}) \oplus z \\ &= (\overline{x} \cdot \overline{y} + xy)z + (\overline{x}y + x\overline{y})\overline{z} \\ &= \overline{x} \cdot \overline{y}z + xyz + \overline{x}y\overline{z} + x\overline{y} \cdot \overline{z} \end{aligned}$$

The two SOP expressions are the same.

5.7. In the circuit of Figure 5.5b, we have:

$$\begin{aligned} s_i &= (x_i \oplus y_i) \oplus c_i \\ &= x_i \oplus y_i \oplus c_i \end{aligned}$$

$$\begin{aligned} c_{i+1} &= (x_i \oplus y_i)c_i + x_i y_i \\ &= (\overline{x}_i y_i + x_i \overline{y}_i)c_i + x_i y_i \\ &= \overline{x}_i y_i c_i + x_i \overline{y}_i c_i + x_i y_i \\ &= y_i c_i + x_i c_i + x_i y_i \end{aligned}$$

The expressions for s_i and c_{i+1} are the same as those derived in Figure 5.4b.

5.8. We will give a descriptive proof for ease of understanding. The 2's complement of a given number can be found by adding 1 to the 1's complement of the number. Suppose that the number has k 0s in the least-significant bit positions, $b_{k-1} \dots b_0$, and it has $b_k = 1$. When this number is converted to its 1's complement, each of these k bits has the value 1. Adding 1 to this string of 1s produces $b_k b_{k-1} b_{k-2} \dots b_0 = 100 \dots 0$. This result is equivalent to copying the k 0s and the first 1 (in bit position b_k) encountered when the number is scanned from right to left. Suppose that the most-significant $n - k$ bits, $b_{n-1} b_{n-2} \dots b_k$, have some pattern of 0s and 1s, but $b_k = 1$. In the 1's complement this pattern will be complemented in each bit position, which will include $b_k = 0$. Now, adding 1 to the entire n-bit number will make $b_k = 1$, but no further carries will be generated; therefore, the complemented bits in positions $b_{n-1} b_{n-2} \dots b_{k+1}$ will remain unchanged.

5.9. Construct the truth table

x_{n-1}	y_{n-1}	c_{n-1}	c_n	s_{n-1} (sign bit)	Overflow
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	1	1	0	0
1	1	0	1	0	1
1	1	1	1	1	0

Note that overflow cannot occur when two numbers with opposite signs are added. From the truth table the overflow expression is

$$Overflow = \overline{c}_n c_{n-1} + c_n \overline{c}_{n-1} = c_n \oplus c_{n-1}$$

5.10. Since $s_k = x_k \oplus y_k \oplus c_k$, it follows that

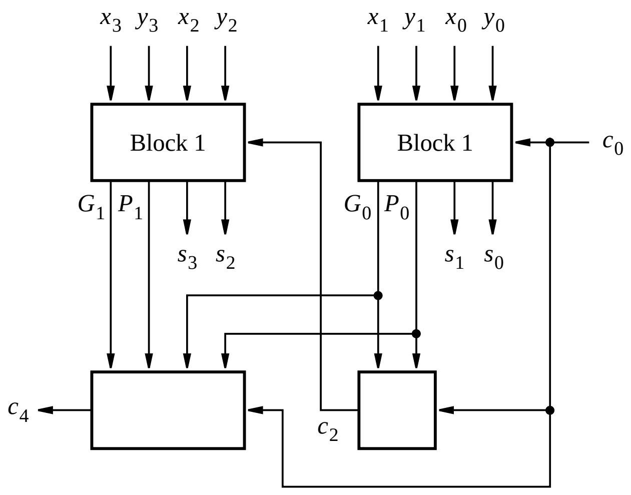
$$\begin{aligned} x_k \oplus y_k \oplus s_k &= (x_k \oplus y_k) \oplus (x_k \oplus y_k \oplus c_k) \\ &= (x_k \oplus y_k) \oplus (x_k \oplus y_k) \oplus c_k \\ &= 0 \oplus c_k \\ &= c_k \end{aligned}$$

5.11. Yes, it works. The NOT gate that produces c_i is not needed in stages where $i > 0$. The drawback is "poor" propagation of $\overline{c}_i = 1$ through the topmost NMOS transistor. The positive aspect is fewer transistors needed to produce $\overline{c}_{i+1}$.

5.12. From Expression 5.4, each c_i requires i AND gates and one OR gate. Therefore, to determine all c_i signals we need $\sum_{i=1}^{n}(i+1) = (n^2+3n)/2$ gates. In addition to this, we need $3n$ gates to generate all g, p, and s functions. Therefore, a total of $(n^2+9n)/2$ gates are needed.

5.13. 84 gates.

5.14. The circuit for a 4-bit version of the adder based on the hierarchical structure in Figure 5.18 is constructed as follows:

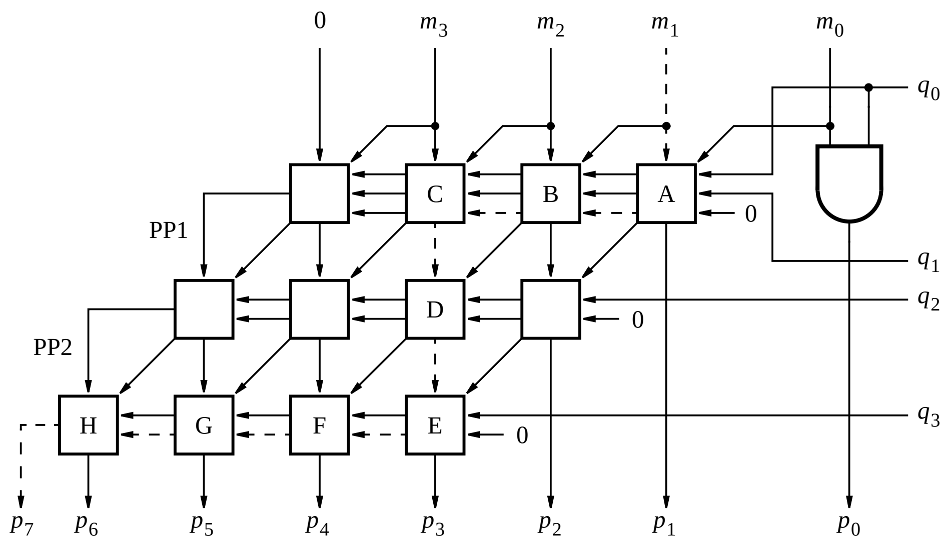


Blocks 0 and 1 have the structure similar to the circuit in Figure 5.16. The overall circuit is given by the expressions

$$\begin{aligned} p_i &= x_i + y_i \\ g_i &= x_i y_i \\ P_0 &= p_1 p_0 \\ G_0 &= g_1 + p_1 g_0 \end{aligned}$$

$$
\begin{aligned}
P_1 &= p_3 p_2 \\
G_1 &= g_3 + p_3 g_2 \\
c_2 &= G_0 + P_0 c_0 \\
c_4 &= G_1 + P_1 G_0 + P_1 P_0 c_0
\end{aligned}
$$

5.15. The longest path, which causes the critical delay, is from the inputs m_0 and m_1 to the output p_7, indicated by the dashed path in the following copy of Figure 5.33a:



Propagation through the block A involves one gate delay in the AND gate shown in Figure 5.33b and two gate delays to generate the carry-out in the full-adder. Then, in each of the blocks B, C, D, E, F, G, and H, two more gate delays are needed to generate the carry-out signals in the circuits depicted by Figure 5.33c. Therefore, the total delay along the critical path is 17 gate delays.

5.16. The 4 × 4 multiplier in Figure 5.36 can be implemented as follows:

```
module fig5_36 (M, Q, P);
    input [3:0] M, Q;
    output [7:0] P;
    wire [3:1] Ctop, Csecond, Cbottom;
    wire [5:2] PP1;
    wire [6:3] PP2;

    assign P[0] = M[0] & Q[0];
    fig5_36b toprow_stage0 (M[1], M[0], Q[1], Q[0], 0, Ctop[1], P[1]);
    fig5_36b toprow_stage1 (M[2], M[1], Q[1], Q[0], Ctop[1], Ctop[2], PP1[2]);
    fig5_36b toprow_stage2 (M[3], M[2], Q[1], Q[0], Ctop[2], Ctop[3], PP1[3]);
    fig5_36b toprow_stage3 (0, M[3], Q[1], Q[0], Ctop[3], PP1[5], PP1[4]);
    fig5_36c secondrow_stage0 (PP1[2], M[0], Q[2], 0, Csecond[1], P[2]);
    fig5_36c secondrow_stage1 (PP1[3], M[1], Q[2], Csecond[1], Csecond[2], PP2[3]);
    fig5_36c secondrow_stage2 (PP1[4], M[2], Q[2], Csecond[2], Csecond[3], PP2[4]);
    fig5_36c secondrow_stage3 (PP1[5], M[3], Q[2], Csecond[3], PP2[6], PP2[5]);
    fig5_36c bottomrow_stage0 (PP2[3], M[0], Q[3], 0, Cbottom[1], P[3]);
    fig5_36c bottomrow_stage1 (PP2[4], M[1], Q[3], Cbottom[1], Cbottom[2], P[4]);
    fig5_36c bottomrow_stage2 (PP2[5], M[2], Q[3], Cbottom[2], Cbottom[3], P[5]);
    fig5_36c bottomrow_stage3 (PP2[6], M[3], Q[3], Cbottom[3], P[7], P[6]);
endmodule

module fig5_36b (m_k1, m_k, q1, q0, Cin, Cout, s);
    input m_k1, m_k, q1, q0, Cin;
    output Cout, s;
    wire x, y;

    assign x = m_k1 & q0;
    assign y = m_k & q1;
    fulladd FA (Cin, x, y, s, Cout);
endmodule

module fig5_36c (ppi_k1, m_k, qj, Cin, Cout, s);
    input ppi_k1, m_k, qj, Cin;
    output Cout, s;
    wire y;

    assign y = m_k & qj;
    fulladd FA (Cin, ppi_k1, y, s, Cout);
endmodule

module fulladd (Cin, x, y, s, Cout);
    input Cin, x, y;
    output s, Cout;
    reg s, Cout;

    always @(x or y or Cin)
        {Cout, s} = x + y + Cin;
endmodule
```

5.17. The code in Figure P5.2 represents a multiplier. It multiplies the lower two bits of *Input* by the upper two bits of *Input*, producing the four-bit *Output*. The style of code is poor, because it is not readily apparent what is being described.

5.18. Let $Y = y_3y_2y_1y_0$ be the 9's complement of the BCD digit $X = x_3x_2x_1x_0$. Then, Y is defined by the truth table

x_3	x_2	x_1	x_0	y_3	y_2	y_1	y_0
0	0	0	0	1	0	0	1
0	0	0	1	1	0	0	0
0	0	1	0	0	1	1	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	0	0
0	1	1	0	0	0	1	1
0	1	1	1	0	0	1	0
1	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0

This gives

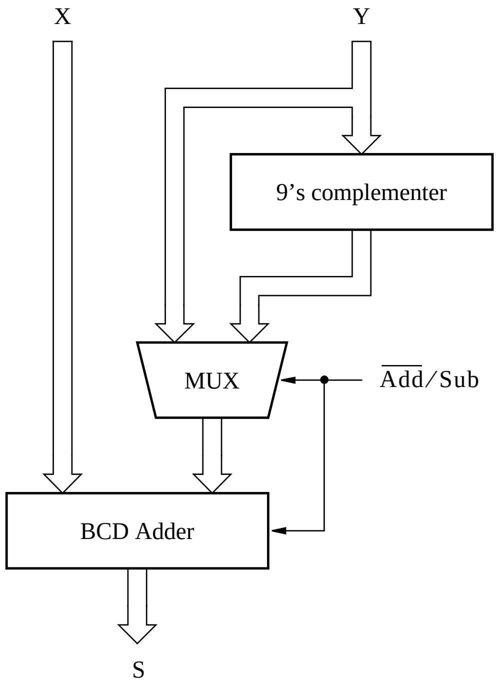
$$\begin{aligned} y_0 &= \overline{x}_0 \\ y_1 &= x_1 \\ y_2 &= \overline{x}_2x_1 + x_2\overline{x}_1 \\ y_3 &= \overline{x}_3\overline{x}_2\overline{x}_1 \end{aligned}$$

5.19. BCD subtraction can be performed using 10's complement representation, using an approach that is similar to 2's complement subtraction. Note that 10's and 2's complements are the radix complements in number systems where the radices are 10 and 2, respectively. Let X and Y be BCD numbers given in 10's complement representation, such that the sign (left-most) BCD digit is 0 for positive numbers and 9 for negative numbers. Then, the subtraction operation $S = X - Y$ is performed by finding the 10's complement of Y and adding it to X, ignoring any carry-out from the sign-digit position.

For example, let $X = 068$ and $Y = 043$. Then, the 10's complement of Y is 957, and $S' = 068 + 957 = 1025$. Dropping the carry-out of 1 from the sign-digit position gives $S = 025$.

As another example, let $X = 032$ and $Y = 043$. Then, $S = 032 + 957 = 989$, which represents -11_{10}.

The 10's complement of Y can be formed by adding 1 to the 9's complement of Y. Therefore, a circuit that can add and subtract BCD operands can be designed as follows:

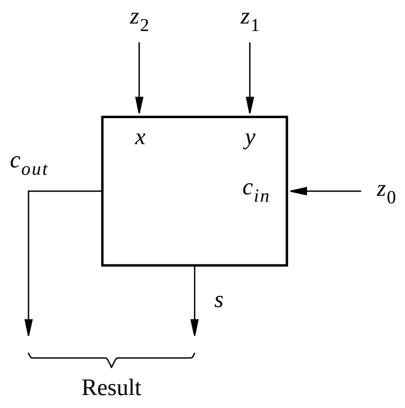


For the 9's complementer one can use the circuit designed in problem 5.18. The BCD adder is a circuit based on the approach illustrated in Figure 5.40.

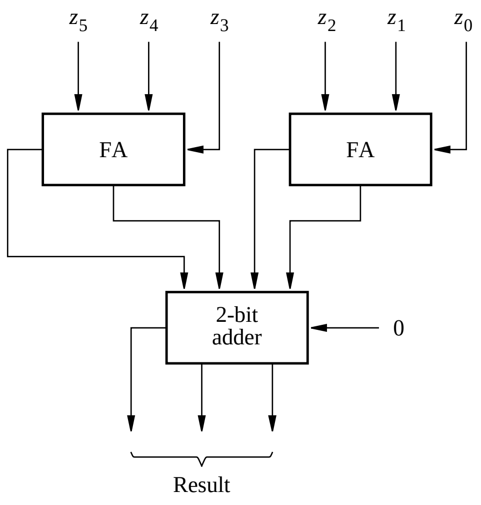
5.20. A possible Verilog code is

```
module bcdaddsubtract (A, B, D, Add_Sub, carryout);
    input [15:0] A, B;
    input Add_Sub;
    output [15:0] D;
    output carryout;
    reg [15:0] Bmux;
    wire [15:0] Bnot;
    wire [3:1] C;

    complement_digit dig0 (B[3:0], Bnot[3:0]);
    complement_digit dig1 (B[7:4], Bnot[7:4]);
    complement_digit dig2 (B[11:8], Bnot[11:8]);
    complement_digit dig3 (B[15:12], Bnot[15:12]);
    always @(B or Bnot or Add_Sub)
        if (Add_Sub == 0)  Bmux = B;
        else Bmux = Bnot;
    bcdadd stage0 (Add_Sub, A[3:0], Bmux[3:0], D[3:0], C[1]);
    bcdadd stage1 (C[1], A[7:4], Bmux[7:4], D[7:4], C[2]);
    bcdadd stage2 (C[2], A[11:8], Bmux[11:8], D[11:8], C[3]);
    bcdadd stage3 (C[3], A[15:12], Bmux[15:12], D[15:12], carryout);
endmodule

module complement_digit (W, Wnot);
    input [3:0] W;
    output [3:0] Wnot;

    assign Wnot[0] = ~W[0];
    assign Wnot[1] = W[1];
    assign Wnot[2] = (~W[2] & W[1]) | (W[2] & ~W[1]);
    assign Wnot[3] = ~W[3] & ~W[2] & ~W[1];
endmodule

module bcdadd (Cin, X, Y, S, Cout);
    input Cin;
    input [3:0] X, Y;
    output [3:0] S;
    output Cout;
    reg [3:0] S;
    reg Cout;
    reg [4:0] Z;

    always @(X or Y or Cin)
    begin
        Z = X + Y + Cin;
        if (Z < 10)  {Cout, S} = Z;
        else  {Cout, S} = Z + 6;
    end
endmodule
```

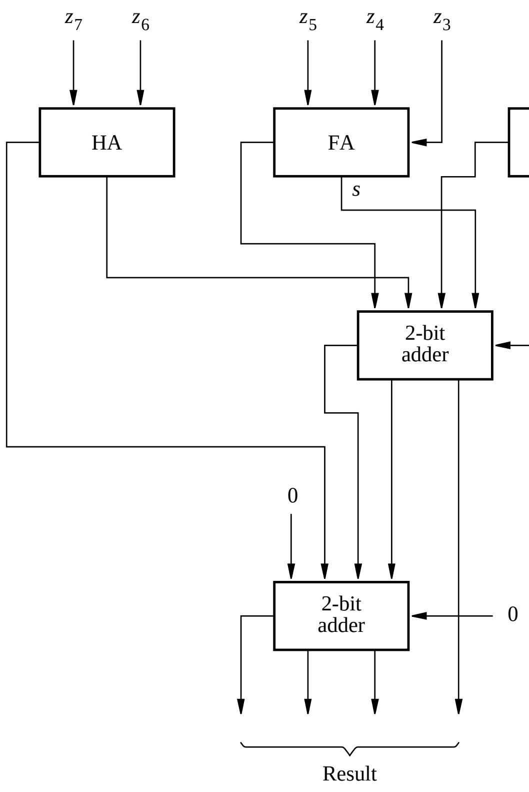
5.21. A full-adder circuit can be used, such that two of the bits of the number are connected as inputs x and y, while the third bit is connected as the carry-in. Then, the carry-out and sum bits will indicate how many input bits are equal to 1.



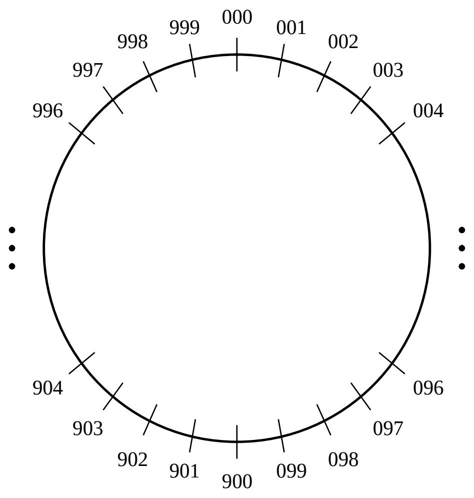
5.22. Using the approach explained in the solution to problem 5.21, the desired circuit can be built as follows:



5.23. Using the approach explained in the solutions to problems 5.21 and 5.22, the desired circuit can be built as follows:



5.24. The graphical representation is



For example, the addition $-3 + (+5) = 2$ involves starting at 997 (= -3) and going clockwise 5 numbers, which gives the result 002 (= +2). Similarly, the subtraction $4 - (+8) = -4$ involves starting at 004 (= +4) and going counterclockwise 8 numbers, which gives the result 996 (= -4).

5.25. The ternary half-adder in Figure P5.3 can be defined using binary-encoded signals as follows:

A		B		Carry	Sum	
a_1	a_0	b_1	b_0	c_{out}	s_1	s_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	1	0	0
1	0	0	0	0	1	0
1	0	0	1	1	0	0
1	0	1	0	1	0	1

The remaining 7 (out of 16) valuations, where either $a_1 = a_0 = 1$, or $b_1 = b_0 = 1$, can be treated as don't care conditions. Then, the minimum cost expressions are:

$$\begin{aligned}
c_{out} &= a_0 b_1 + a_1 b_1 + a_1 b_0 \\
s_1 &= a_0 b_0 + \overline{a}_1 \overline{a}_0 b_1 + a_1 \overline{b}_1 \overline{b}_0 \\
s_0 &= a_1 b_1 + \overline{a}_1 \overline{a}_0 b_0 + a_0 \overline{b}_1 \overline{b}_0
\end{aligned}$$

5.26. Ternary full-adder is defined by the truth table:

c_{in}	A	B	c_{out}	Sum
0	0	0	0	0
0	0	1	0	1
0	0	2	0	2
0	1	0	0	1
0	1	1	0	2
0	1	2	1	0
0	2	0	0	2
0	2	1	1	0
0	2	2	1	1
1	0	0	0	1
1	0	1	0	2
1	0	2	1	0
1	1	0	0	2
1	1	1	1	0
1	1	2	1	1
1	2	0	1	0
1	2	1	1	1
1	2	2	1	2

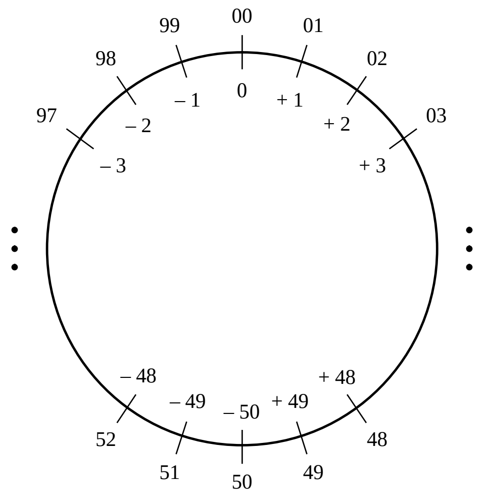
Using binary-encoded signals for this full-adder gives the following truth table:

	A		B			Sum	
c_{in}	a_1	a_0	b_1	b_0	c_{out}	s_1	s_0
0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	0	1
0	0	1	0	1	0	1	0
0	0	1	1	0	1	0	0
0	1	0	0	0	0	1	0
0	1	0	0	1	1	0	0
0	1	0	1	0	1	0	1
1	0	0	0	0	0	0	1
1	0	0	0	1	0	1	0
1	0	0	1	0	1	0	0
1	0	1	0	0	0	1	0
1	0	1	0	1	1	0	0
1	0	1	1	0	1	0	1
1	1	0	0	0	1	0	0
1	1	0	0	1	1	0	1
1	1	0	1	0	1	1	0

Treating the 14 (out of 32) valuations where either $a_1 = a_0 = 1$ or $b_1 = b_0 = 1$ as don't care conditions, leads to the minimum cost expressions

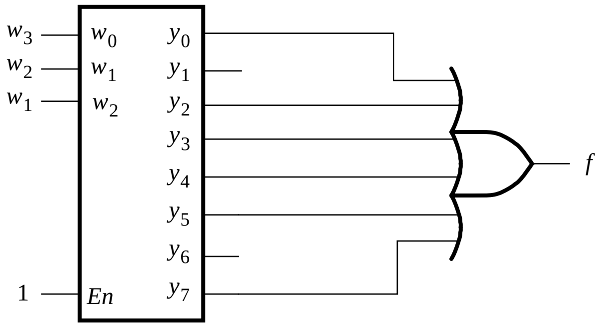
$$\begin{aligned}
c_{out} &= a_0b_1 + a_1b_0 + a_1b_1 + a_1c_{in} + b_1c_{in} + a_0b_0c_{in} \\
s_1 &= a_0b_0\overline{c}_{in} + \overline{a}_1\overline{a}_0b_1\overline{c}_{in} + a_1\overline{b}_1\overline{b}_0\overline{c}_{in} + a_1b_1c_{in} + \overline{a}_1\overline{a}_0b_0c_{in} + a_0\overline{b}_1\overline{b}_0c_{in} \\
s_0 &= a_1b_1\overline{c}_{in} + \overline{a}_1\overline{a}_0b_0\overline{c}_{in} + a_0\overline{b}_1\overline{b}_0\overline{c}_{in} + a_1b_0c_{in} + a_0b_1c_{in} + \overline{a}_1\overline{a}_0\overline{b}_1\overline{b}_0c_{in}
\end{aligned}$$

5.27. The subtractions $26 - 27 = 99$ and $18 - 34 = 84$ make sense if the two-digit numbers 00 to 99 are interpreted so that the numbers 00 to 49 are positive integers from 0 to +49, while the numbers 50 to 99 are negative integers from -50 to -1. This scheme can be illustrated graphically as follows:

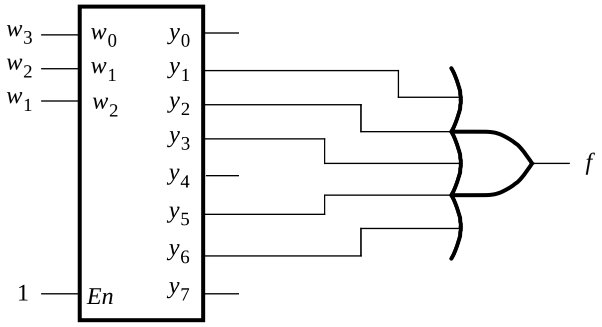


Chapter 6

6.1.



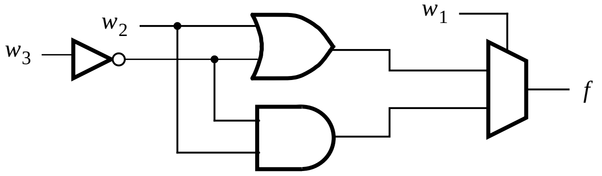
6.2.



6.3.

w_1	w_2	w_3	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

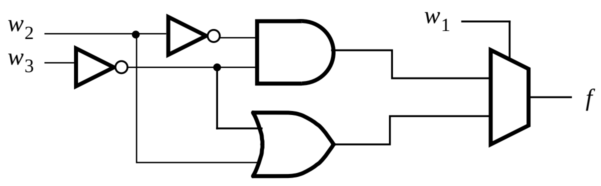
w_1	f
0	$w_2 + \overline{w}_3$
1	$w_2\overline{w}_3$



6.4.

w_1	w_2	w_3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

w_1	f
0	$\overline{w}_2\overline{w}_3$
1	$w_2 + \overline{w}_3$



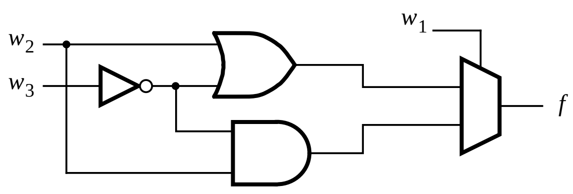
6.5. The function f can be expressed as

$$f = \overline{w}_1\overline{w}_2\overline{w}_3 + \overline{w}_1 w_2\overline{w}_3 + \overline{w}_1 w_2 w_3 + w_1 w_2\overline{w}_3$$

Expansion in terms of w_1 produces

$$f = \overline{w}_1(w_2 + \overline{w}_3) + w_1(w_2\overline{w}_3)$$

The corresponding circuit is



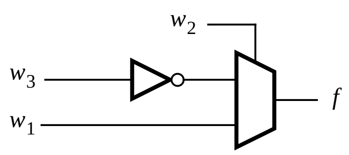
6.6. The function f can be expressed as

$$f = \overline{w}_1\overline{w}_2\overline{w}_3 + w_1\overline{w}_2\overline{w}_3 + w_1 w_2\overline{w}_3 + w_1 w_2 w_3$$

Expansion in terms of w_2 produces

$$f = \overline{w}_2(\overline{w}_3) + w_2(w_1)$$

The corresponding circuit is



6.7. Expansion in terms of w_2 gives

$$\begin{aligned} f &= \overline{w}_2(1 + \overline{w}_1\overline{w}_3 + w_1w_3) + w_2(\overline{w}_1\overline{w}_3 + w_1w_3) \\ &= \overline{w}_1\overline{w}_2\overline{w}_3 + w_1\overline{w}_2w_3 + \overline{w}_2 + \overline{w}_1w_2\overline{w}_3 + w_1w_2w_3 \end{aligned}$$

Further expansion in terms of w_1 gives

$$\begin{aligned} f &= \overline{w}_1(w_2\overline{w}_3 + \overline{w}_2\overline{w}_3 + \overline{w}_2) + w_1(w_2w_3 + \overline{w}_2w_3 + \overline{w}_2) \\ &= \overline{w}_1w_2\overline{w}_3 + \overline{w}_1\overline{w}_2\overline{w}_3 + \overline{w}_1\overline{w}_2 + w_1w_2w_3 + w_1\overline{w}_2w_3 + w_1\overline{w}_2 \end{aligned}$$

Further expansion in terms of w_3 gives

$$\begin{aligned} f &= \overline{w}_3(\overline{w}_1w_2 + \overline{w}_1\overline{w}_2 + \overline{w}_1\overline{w}_2 + w_1\overline{w}_2) + w_3(w_1w_2 + w_1\overline{w}_2 + w_1\overline{w}_2 + \overline{w}_1\overline{w}_2) \\ &= \overline{w}_1w_2\overline{w}_3 + \overline{w}_1\overline{w}_2\overline{w}_3 + w_1\overline{w}_2\overline{w}_3 + w_1w_2w_3 + w_1\overline{w}_2w_3 + \overline{w}_1\overline{w}_2w_3 \end{aligned}$$

6.8. Expansion in terms of w_1 gives

$$f = \overline{w}_1w_2 + \overline{w}_1\overline{w}_3 + w_1w_2$$

Further expansion in terms of w_2 gives

$$\begin{aligned} f &= \overline{w}_2(\overline{w}_1\overline{w}_3) + w_2(w_1 + \overline{w}_1 + \overline{w}_1\overline{w}_3) \\ &= \overline{w}_1w_2 + \overline{w}_1w_2\overline{w}_3 + \overline{w}_1\overline{w}_2\overline{w}_3 + w_1w_2 \end{aligned}$$

Further expansion in terms of w_3 gives

$$\begin{aligned} f &= \overline{w}_3(\overline{w}_1\overline{w}_2 + w_1w_2 + \overline{w}_1w_2 + \overline{w}_1w_2) + w_3(w_1w_2 + \overline{w}_1w_2) \\ &= \overline{w}_1\overline{w}_2\overline{w}_3 + w_1w_2\overline{w}_3 + \overline{w}_1w_2\overline{w}_3 + \overline{w}_1w_2w_3 + w_1w_2w_3 \end{aligned}$$

6.9. Proof of Shannon's expansion theorem

$$f(x_1, x_2, ..., x_n) = \overline{x}_1 \cdot f(0, x_2, ..., x_n) + x_1 \cdot f(1, x_2, ..., x_n)$$

This theorem can be proved using *perfect induction*, by showing that the expression is true for every possible value of x_1. Since x_1 is a boolean variable, we need to look at only two cases: $x_1 = 0$ and $x_1 = 1$.

Setting $x_1 = 0$ in the above expression, we have:

$$\begin{aligned} f(0, x_2, ..., x_n) &= 1 \cdot f(0, x_2, ..., x_n) + 0 \cdot f(1, x_2, ..., x_n) \\ &= f(0, x_2, ..., x_n) \end{aligned}$$

Setting $x_1 = 1$, we have:

$$\begin{aligned} f(1, x_2, ..., x_n) &= 0 \cdot f(0, x_2, ..., x_n) + 1 \cdot f(1, x_2, ..., x_n) \\ &= f(1, x_2, ..., x_n) \end{aligned}$$

This proof can be performed for any arbitrary x_i in the same manner.

6.10. Derivation using $\overline{f}$:

$$\begin{aligned} \overline{f} &= \overline{w}\overline{f}_{\overline{w}} + w\overline{f}_w \\ f &= \left(\overline{\overline{w}\overline{f}_{\overline{w}} + w\overline{f}_w}\right) \\ &= \left(\overline{\overline{w}\overline{f}_{\overline{w}}}\right) \cdot \left(\overline{w\overline{f}_w}\right) \\ &= (w + f_{\overline{w}})(\overline{w} + f_w) \end{aligned}$$

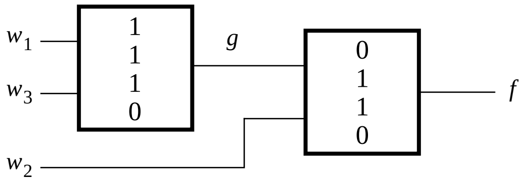
6.11. Expansion in terms of w_2 gives

$$f = \overline{w}_2(\overline{w}_1 + \overline{w}_3) + w_2(w_1 w_3)$$

Letting $g = \overline{w}_1 + \overline{w}_3$, we have

$$f = \overline{w}_2 g + w_2 \overline{g}$$

The corresponding circuit is



6.12. Expansion of f in terms of w_2 gives

$$\begin{aligned} f &= \overline{w}_2(\overline{w}_1 + \overline{w}_3) + w_2(w_1 w_3) \\ &= w_2 \oplus (\overline{w}_1 + \overline{w}_3) \\ &= w_2 \oplus \overline{w_1 w_3} \end{aligned}$$

The cost of this multilevel circuit is 2 gates + 4 inputs = 6.

6.13. Using Shannon's expansion in terms of w_2 we have

$$\begin{aligned} f &= \overline{w}_2(\overline{w}_3 + \overline{w}_1 w_4) + w_2(w_3 \overline{w}_4 + w_1 w_3) \\ &= \overline{w}_2(\overline{w}_3 + \overline{w}_1 w_4) + w_2(w_3(w_1 + \overline{w}_4)) \end{aligned}$$

If we let $g = \overline{w}_3 + \overline{w}_1 w_4$, then

$$f = \overline{w}_2 g + w_2 \overline{g}$$

Thus, two 3-LUTs are needed to implement f.

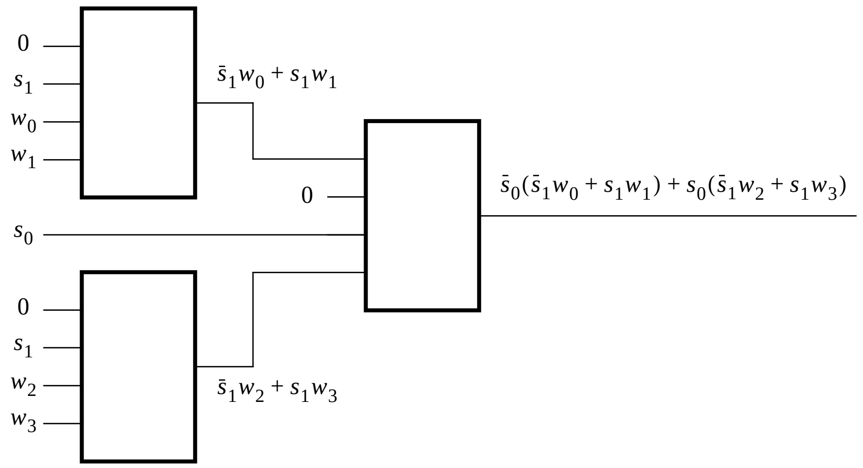
6.14. Any number of 5-variable functions can be implemented by using two 4-LUTs. For example, if we cascade the two 4-LUTs by connecting the output of one 4-LUT to an input of the other, then we can realize any function of the form

$$\begin{aligned} f &= f_1(w_1, w_2, w_3, w_4) + w_5 \\ f &= f_1(w_1, w_2, w_3, w_4) \cdot w_5 \end{aligned}$$

6.15. Expressing f in the form

$$\begin{aligned} f &= \overline{s}_1\overline{s}_0w_0 + s_1\overline{s}_0w_1 + \overline{s}_1s_0w_2 + s_1s_0w_3 \\ &= \overline{s}_0(\overline{s}_1w_0 + s_1w_1) + s_0(\overline{s}_1w_2 + s_1w_3) \end{aligned}$$

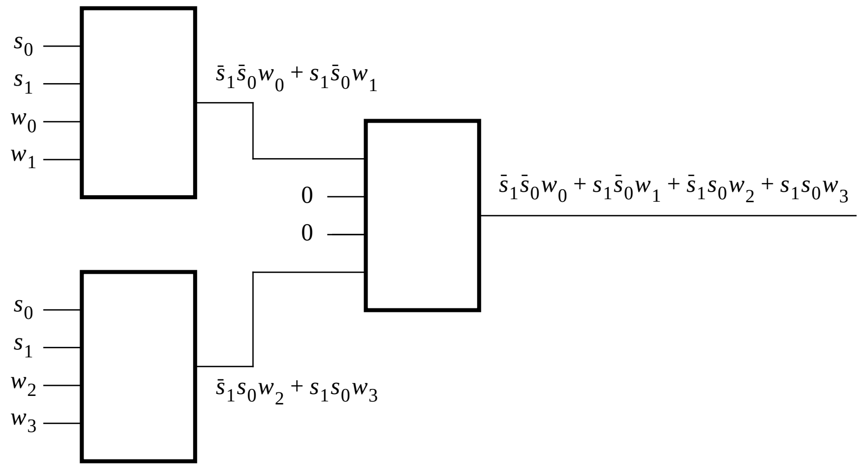
leads to the circuit.



Alternatively, directly using the expression

$$f = \overline{s}_1\overline{s}_0w_0 + s_1\overline{s}_0w_1 + \overline{s}_1s_0w_2 + s_1s_0w_3$$

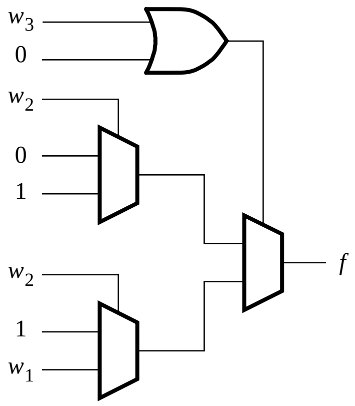
leads to the circuit.


s_0
s_1
w_0
w_1
$\bar{s}_1\bar{s}_0w_0 + s_1\bar{s}_0w_1$
0
0
$\bar{s}_1\bar{s}_0w_0 + s_1\bar{s}_0w_1 + \bar{s}_1s_0w_2 + s_1s_0w_3$
s_0
s_1
w_2
w_3
$\bar{s}_1s_0w_2 + s_1s_0w_3$

6.16. Using Shannon's expansion in terms of w_3 we have

$$\begin{aligned} f &= \overline{w}_3(w_2) + w_3(w_1 + \overline{w}_2) \\ &= \overline{w}_3(w_2) + w_3(\overline{w}_2 + w_2w_1) \end{aligned}$$

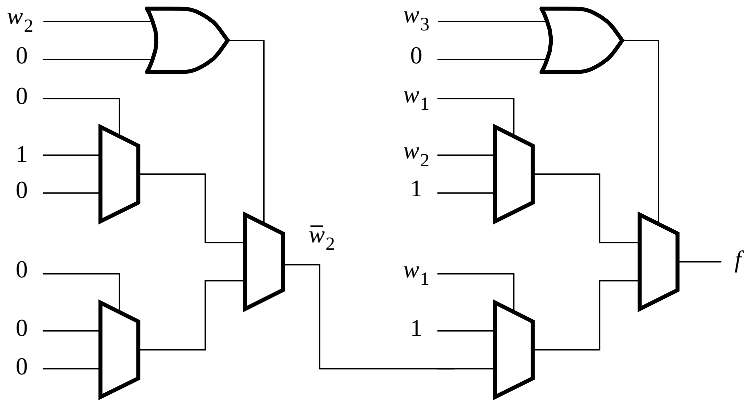
The corresponding circuit is



6.17. Using Shannon's expansion in terms of w_3 we have

$$f = w_3(\overline{w}_1 + w_1\overline{w}_2) + \overline{w}_3(w_1 + \overline{w}_1w_2)$$

The corresponding circuit is



6.18. The code in Figure P6.2 is a 2-to-4 decoder with an enable input. It is not a good style for defining this decoder. The code is not easy to read. Moreover, the Verilog compiler often turns **if** statements into multiplexers, in which case the resulting decoder may have multiplexers controlled by the En signal on the output side.

6.19. The function $f(w_1, w_2, w_3) = \sum m(1, 2, 3, 5, 6)$ can be implemented using the followung code:

```
module prob6_19 (W, f);
    input [1:3] W;
    output f;
    reg  f;

    always @(W)
        case (W)
            3'b001: f = 1;
            3'b010: f = 1;
            3'b011: f = 1;
            3'b101: f = 1;
            3'b110: f = 1;
            default: f = 0;
        endcase

endmodule
```

6.20. Using the truth table in Figure 6.23a, the 4-to-2 binary encoder can be implemented as:

```
module prob6_20 (W, Y);
    input [3:0] W;
    output [1:0] Y;
    reg  [1:0] Y;

    always @(W)
        case (W)
            4'b0001: Y = 2'b00;
            4'b0010: Y = 2'b01;
            4'b0100: Y = 2'b10;
            4'b1000: Y = 2'b11;
            default: Y = 2'bxx;
        endcase

endmodule
```

6.21. An 8-to-2 binary encoder can be implemented as:

```
module prob6_21 (W, Y);
   input [7:0] W;
   output [2:0] Y;
   reg [2:0] Y;

   always @(W)
      case (W)
         8'b00000001: Y = 3'b000;
         8'b00000010: Y = 3'b001;
         8'b00000100: Y = 3'b010;
         8'b00001000: Y = 3'b011;
         8'b00010000: Y = 3'b100;
         8'b00100000: Y = 3'b101;
         8'b01000000: Y = 3'b110;
         8'b10000000: Y = 3'b111;
         default: Y = 3'bxxx;
      endcase

endmodule
```

6.22. The code in Figure P6.3 will instantiate latches on the outputs of the decoder because the **if** statement does not specify all possibilities in a combinational circuit. It can be fixed by including the **else** clause

```
else Y[k] = 0;
```

after the **if** clause.

6.23. First define a set of intermediate variables

$$\begin{aligned}
i_0 &= \overline{w}_7\overline{w}_6\overline{w}_5\overline{w}_4\overline{w}_3\overline{w}_2\overline{w}_1 w_0\\
i_1 &= \overline{w}_7\overline{w}_6\overline{w}_5\overline{w}_4\overline{w}_3\overline{w}_2 w_1\\
i_2 &= \overline{w}_7\overline{w}_6\overline{w}_5\overline{w}_4\overline{w}_3 w_2\\
i_3 &= \overline{w}_7\overline{w}_6\overline{w}_5\overline{w}_4 w_3\\
i_4 &= \overline{w}_7\overline{w}_6\overline{w}_5 w_4\\
i_5 &= \overline{w}_7\overline{w}_6 w_5\\
i_6 &= \overline{w}_7 w_6\\
i_7 &= w_7
\end{aligned}$$

Now a traditional binary encoder can be used for the priority encoder

$$\begin{aligned}
y_0 &= i_1 + i_3 + i_5 + i_7\\
y_1 &= i_2 + i_3 + i_6 + i_7\\
y_2 &= i_4 + i_5 + i_6 + i_7
\end{aligned}$$

6.24. An 8-to-3 priority encoder can be implemented using a **case** statement as follows:

```
module  prob6_24 (W, Y, z);
    input  [7:0] W;
    output  [2:0] Y;
    output  z;
    reg   [2:0] Y;
    reg   z;

    always @(W)
    begin
        z = 1;
        case (W)
            8'b1xxxxxxx: Y = 7;
            8'b01xxxxxx: Y = 6;
            8'b001xxxxx: Y = 5;
            8'b0001xxxx: Y = 4;
            8'b00001xxx: Y = 3;
            8'b000001xx: Y = 2;
            8'b0000001x: Y = 1;
            8'b00000001: Y = 0;
            default:   begin
                           z = 0;
                           Y = 3'bx;
                       end
        endcase
endmodule
```

6.25. An 8-to-3 priority encoder can be implemented using a **for** loop as follows:

```
module  prob6_25 (W, Y, z);
    input  [7:0] W;
    output  [2:0] Y;
    output  z;
    reg   [2:0] Y;
    reg   z;
    integer   k;

    always @(W)
    begin
        Y = 3'bx;
        z = 0;
        for (k = 0; k < 8; k = k+1)
            if (W[k])
            begin
                Y = k;
                z = 1;
            end
    end
endmodule
```

6.26. The following code can be used:

```
// 3-to-8 decoder
module  h3to8 (W, Y, En);
    input  [2:0] W;
    input  En;
    output  [0:7] Y;
    wire  [0:7] Y;
    reg   En0to3, En4to7;

    always @(W or En)
    begin
        if (En == 0)
        begin
            En0to3 = 0;   En4to7 = 0;
        end
        else if (W[2] == 0)
        begin
            En0to3 = 1;   En4to7 = 0;
        end
        else if (W[2] == 1)
        begin
            En0to3 = 0;   En4to7 = 1;
        end
    end

    if2to4  lowbits  (W[1:0], Y[0:3], En0to3);
    if2to4  highbits  (W[1:0], Y[4:7], En4to7);

endmodule

// 2-to-4 decoder
module  if2to4 (W, Y, En);
    input  [1:0] W;
    input  En;
    output  [0:3] Y;
    reg  [0:3] Y;

    always @(W or En)
        if (En == 0)   Y = 4'b0000;
        else if (W == 0)   Y = 4'b0001;
        else if (W == 1)   Y = 4'b0010;
        else if (W == 2)   Y = 4'b0100;
        else if (W == 3)   Y = 4'b1000;

endmodule
```

6.27. A 6-to-64 binary decoder can be implemented by using the code:

```
module  h6to64 (W, Y, En);
    input  [5:0] W;
    input  En;
    output  [0:63] Y;
    wire  [0:63] Y;
    reg   [7:0] En3to8dec;

    always @(W or En)
    begin
        if (En == 0)
            En3to8dec = 8'b00000000;
        else
            case (W[5:3])
                0: En3to8dec = 8'b00000001;
                1: En3to8dec = 8'b00000010;
                2: En3to8dec = 8'b00000100;
                3: En3to8dec = 8'b00001000;
                4: En3to8dec = 8'b00010000;
                5: En3to8dec = 8'b00100000;
                6: En3to8dec = 8'b01000000;
                7: En3to8dec = 8'b10000000;
            endcase
    end

    h3to8  dec0  (W[2:0], Y[0:7], En3to8dec[0]);
    h3to8  dec1  (W[2:0], Y[8:15], En3to8dec[1]);
    h3to8  dec2  (W[2:0], Y[16:23], En3to8dec[2]);
    h3to8  dec3  (W[2:0], Y[24:31], En3to8dec[3]);
    h3to8  dec4  (W[2:0], Y[32:39], En3to8dec[4]);
    h3to8  dec5  (W[2:0], Y[40:47], En3to8dec[5]);
    h3to8  dec6  (W[2:0], Y[48:55], En3to8dec[6]);
    h3to8  dec7  (W[2:0], Y[56:63], En3to8dec[7]);

endmodule

//The rest of the code includes the 3-to-8 decoder
//developed in problem 6.26.
```

```
// 3-to-8 decoder
module  h3to8 (W, Y, En);
    input  [2:0] W;
    input  En;
    output  [0:7] Y;
    wire  [0:7] Y;
    reg   En0to3, En4to7;

    always @(W or En)
    begin
        if (En == 0)
        begin
                En0to3 = 0;   En4to7 = 0;
        end
        else if (W[2] == 0)
        begin
                En0to3 = 1;   En4to7 = 0;
        end
        else if (W[2] == 1)
        begin
                En0to3 = 0;   En4to7 = 1;
        end
    end

    if2to4  lowbits  (W[1:0], Y[0:3], En0to3);
    if2to4  highbits  (W[1:0], Y[4:7], En4to7);

endmodule

// 2-to-4 decoder
module  if2to4 (W, Y, En);
    input  [1:0] W;
    input  En;
    output  [0:3] Y;
    reg  [0:3] Y;

    always @(W or En)
        if (En == 0)   Y = 4'b0000;
        else if (W == 0)   Y = 4'b0001;
        else if (W == 1)   Y = 4'b0010;
        else if (W == 2)   Y = 4'b0100;
        else if (W == 3)   Y = 4'b1000;

endmodule
```

6.28. A possible code is:

```
module prob6_28 (W, S, f);
   input [0:3] W;
   input [1:0] S;
   output f;
   wire f;
   wire [0:3] Y;

   dec2to4 decoder (S, Y, 1);
   assign f = (W[0] & Y[0]) | (W[1] & Y[1]) | (W[2] & Y[2]) | (W[3] & Y[3]);

endmodule

module dec2to4 (W, Y, En);
   input [1:0] W;
   input En;
   output [0:3] Y;
   reg  [0:3] Y;

   always @(W or En)
      case (En, W)
            3'b100: Y = 4'b1000;
            3'b101: Y = 4'b0100;
            3'b110: Y = 4'b0010;
            3'b111: Y = 4'b0001;
            default: Y = 4'b0000;
      endcase

endmodule
```

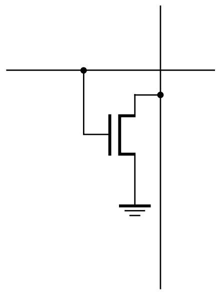
6.29.

$$\begin{aligned} a &= w_3 + w_2 w_0 + w_1 + \overline{w}_2 \overline{w}_0 \\ b &= w_3 + \overline{w}_1 \overline{w}_0 + w_1 w_0 + \overline{w}_2 \\ c &= w_2 + \overline{w}_1 + w_0 \end{aligned}$$

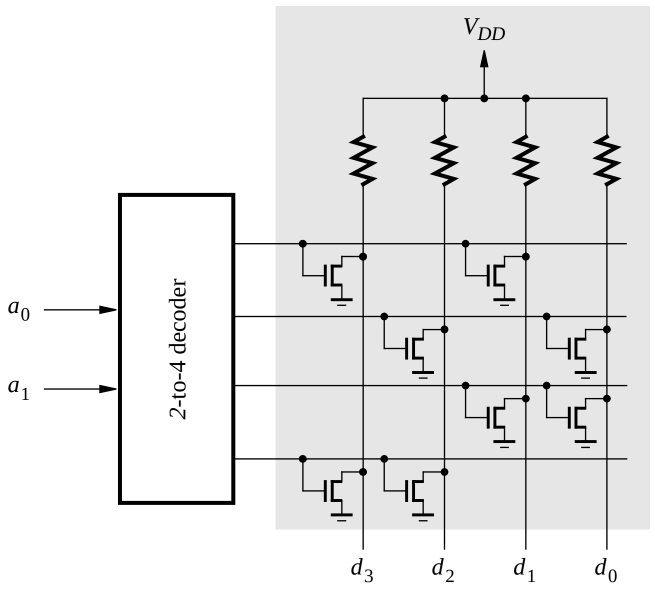
6.30.

$$\begin{aligned} d &= w_3 + \overline{w}_2 \overline{w}_0 + w_1 \overline{w}_0 + w_2 \overline{w}_1 w_0 + \overline{w}_2 w_1 \\ e &= \overline{w}_2 \overline{w}_0 + w_1 \overline{w}_0 \\ f &= w_3 + \overline{w}_1 \overline{w}_0 + w_2 \overline{w}_0 + w_2 \overline{w}_1 \\ g &= w_3 + w_1 \overline{w}_0 + w_2 \overline{w}_1 + \overline{w}_2 w_1 \end{aligned}$$

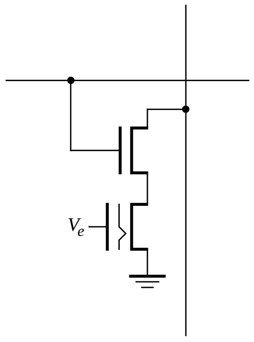
6.31. (a) Each ROM location that should store a 1 requires no circuitry, because the pull-up resistor provides the default value of 1. Each location that stores a 0 has the following cell



(b)

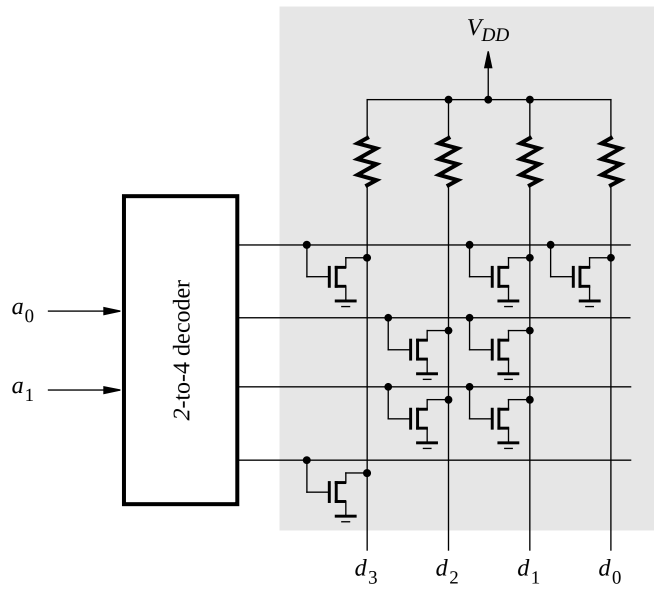


(c) Every location in the ROM contains the following cell



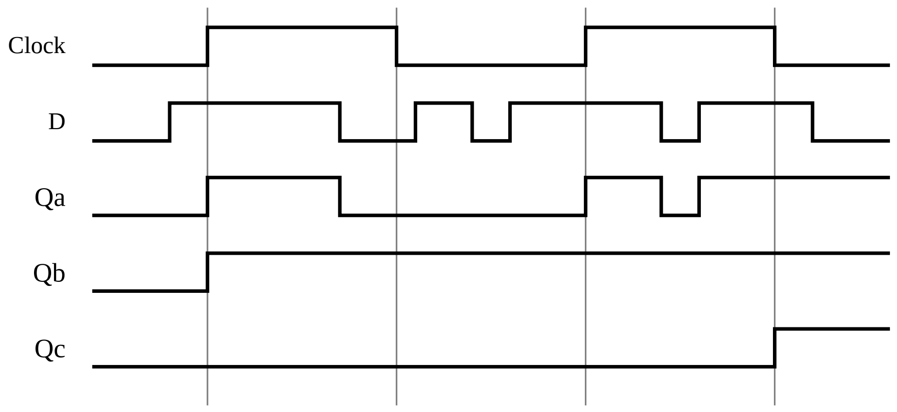
If a location should store a 1, then the corresponding EEPROM transistor is programmed to be turned off. But if the location should store a 0, then the EEPROM transistor is left unprogrammed.

(d)



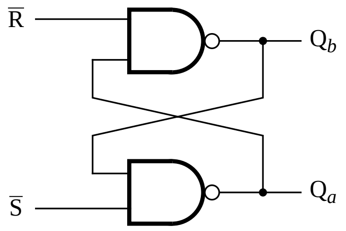
Chapter 7

7.1.



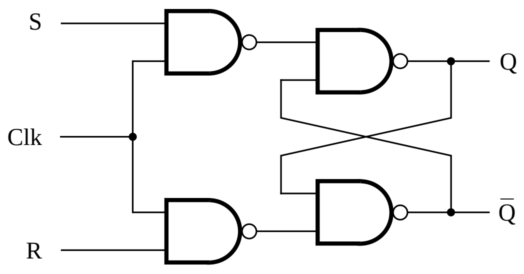
7.2. The circuit in Figure 7.3 can be modified to implement an SR latch by connecting S to the *Data* input and $S + R$ to the *Load* input. Thus the value of S is loaded into the latch whenever either S or R is asserted. Care must be taken to ensure that the *Data* signal remains stable while the *Load*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

7.3.

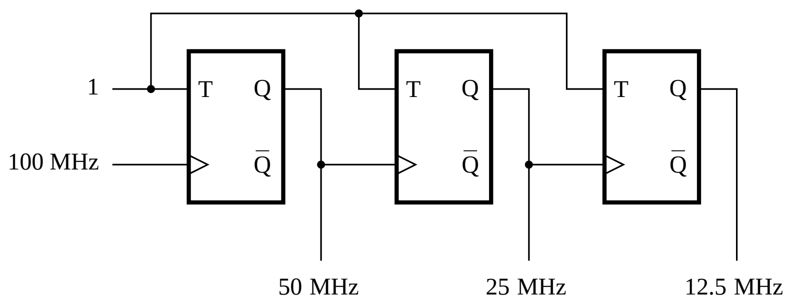


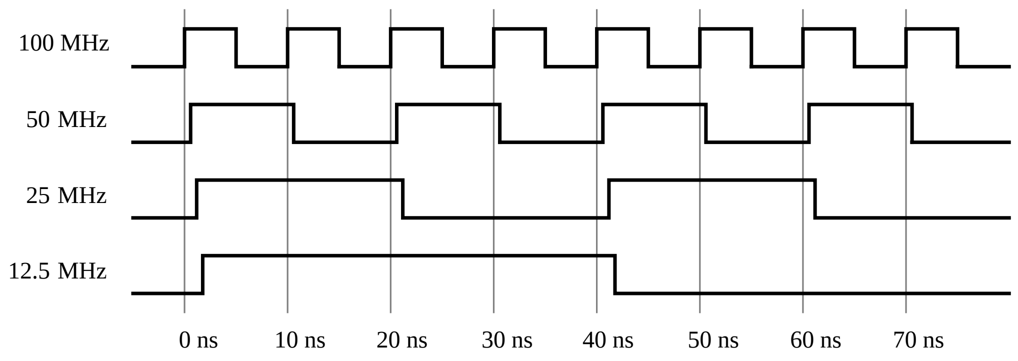
$\overline{S}$	$\overline{R}$	Q_a	Q_b	
1	1	0/1	1/0	(no change)
1	0	0	1	
0	1	1	0	
0	0	1	1	

7.4.

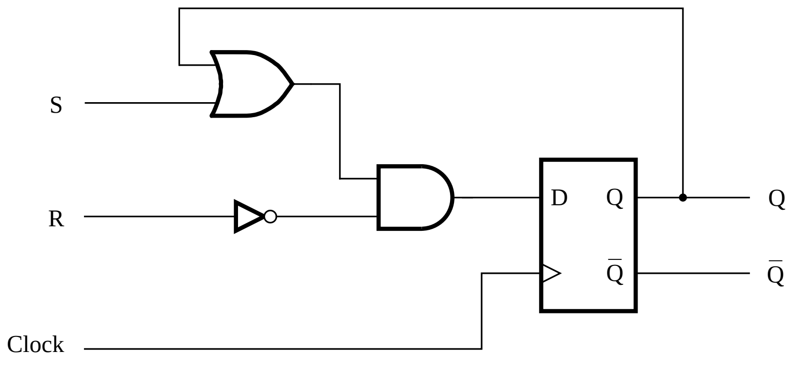


7.5.

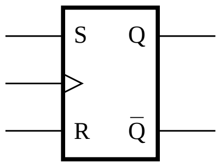




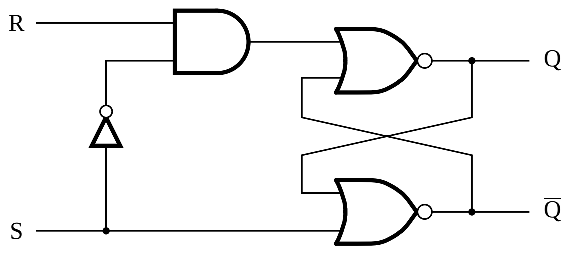
7.6.



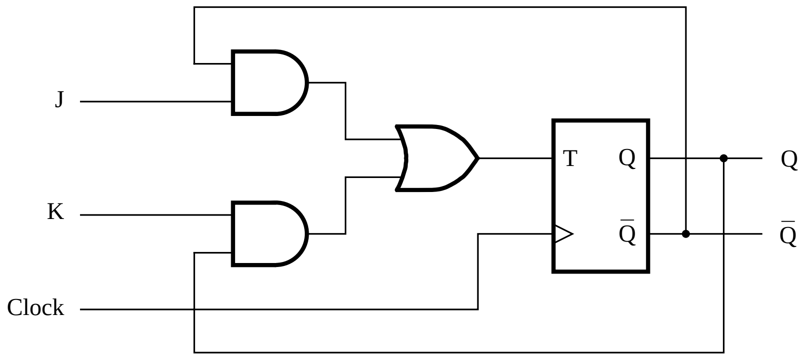
S	R	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	0



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
```

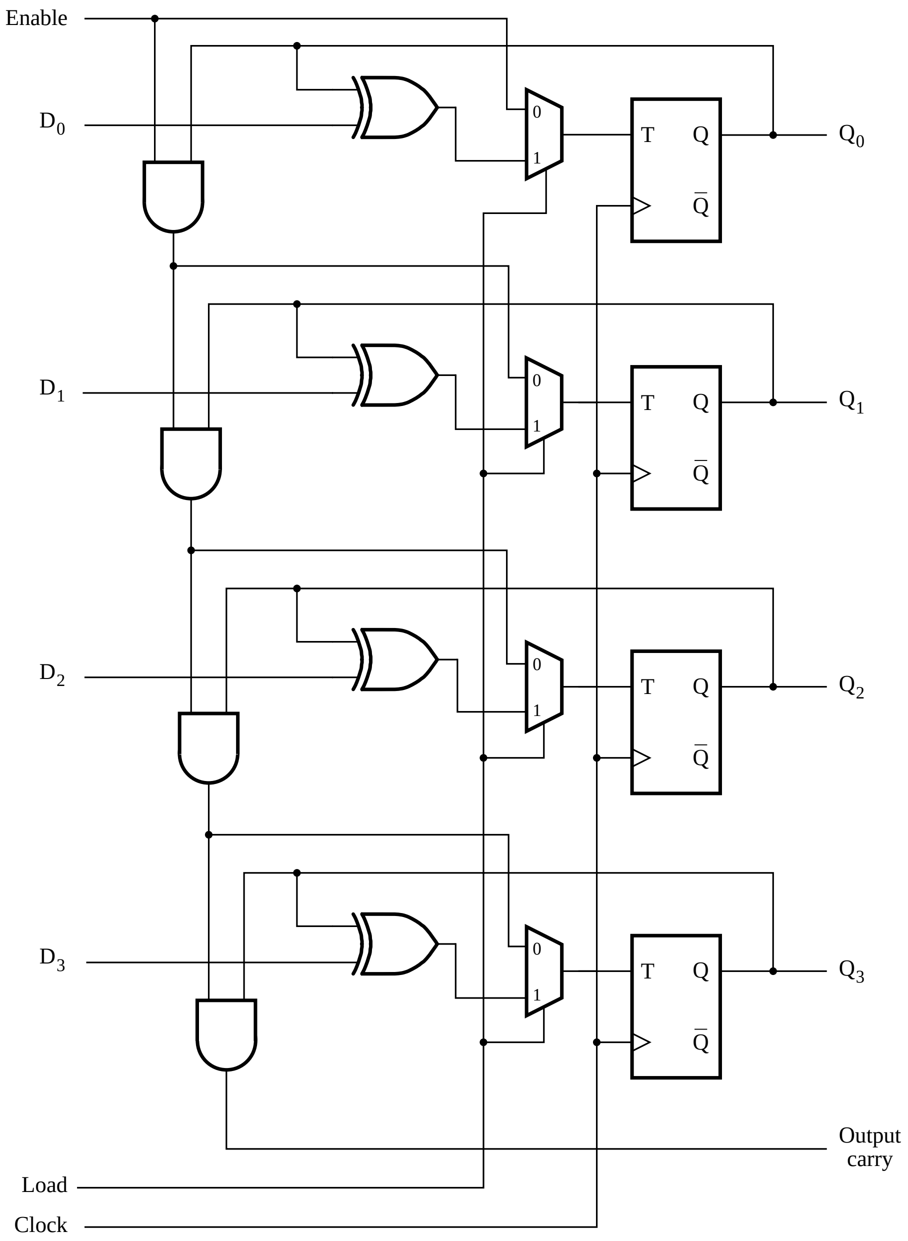
7.13. Let $S = s_1s_0$ be a binary number that specifies the number of bit-positions by which to rotate. Also let L be a parallel-load input, and let $R = r_0r_1r_2r_3$ be parallel data. If the inputs to the flip-flops are $d_0 \ldots d_3$ and the outputs are $q_0 \ldots q_3$, then the barrel-shifter can be represented by the logic expressions

$$\begin{aligned}
d_0 &= L \cdot r_0 + \overline{L} \cdot (\overline{s}_1\overline{s}_0q_0 + \overline{s}_1s_0q_3 + s_1\overline{s}_0q_2 + s_1s_0q_1) \\
d_1 &= L \cdot r_1 + \overline{L} \cdot (\overline{s}_1\overline{s}_0q_1 + \overline{s}_1s_0q_0 + s_1\overline{s}_0q_3 + s_1s_0q_2) \\
d_2 &= L \cdot r_2 + \overline{L} \cdot (\overline{s}_1\overline{s}_0q_2 + \overline{s}_1s_0q_1 + s_1\overline{s}_0q_0 + s_1s_0q_3) \\
d_3 &= L \cdot r_3 + \overline{L} \cdot (\overline{s}_1\overline{s}_0q_3 + \overline{s}_1s_0q_2 + s_1\overline{s}_0q_1 + s_1s_0q_0)
\end{aligned}$$

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

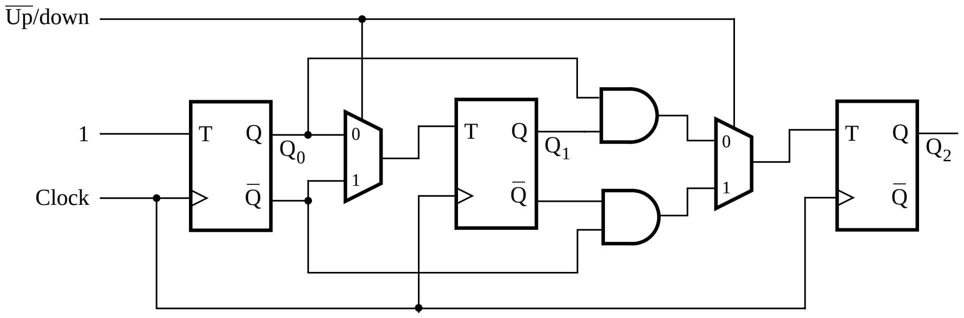
```
// Barrel shifter. If L = 1, load in parallel from R. If L = 0 and E =1
// rotate right by number of bit positions given by S.
module barrel4 (R, L, S, Clock, Q);
    input [0:3] R;
    input L, Clock;
    input [1:0] S;
    output [0:3] Q;
    reg [0:3] Q;
    wire [0:3] M;

    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.



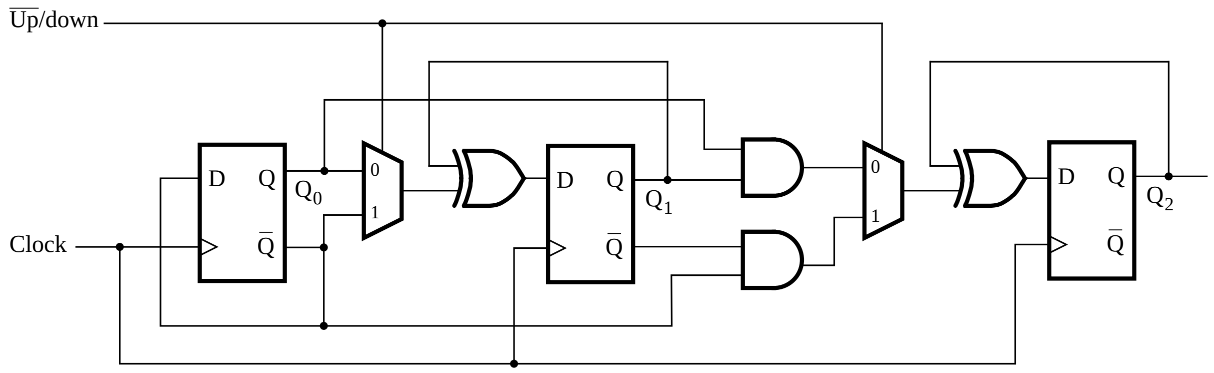
Enable
D_0
D_1
D_2
D_3
0
1
T
Q
$\overline{Q}$
Q_0
Q_1
Q_2
Q_3
Output carry
Load
Clock

7.16.



$\overline{\text{Up}}$/down
1
Clock
T
Q
$\overline{Q}$
Q_0
Q_1
Q_2
0
1

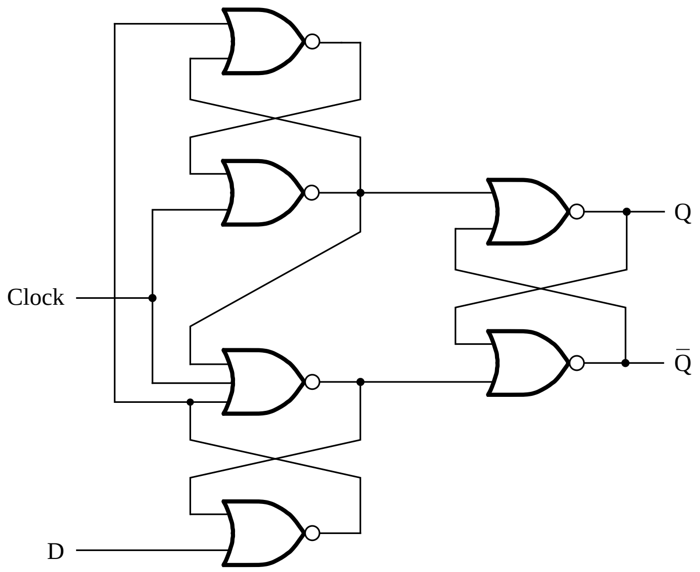
7.17.



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

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

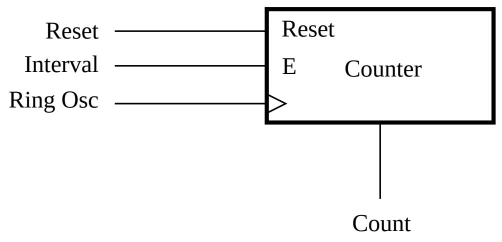
New expressions are needed for R_{in} and R_{out} to accommodate the SWAP operation:

$$\begin{aligned} Rk_{in} &= (I_0 + I_1) \cdot T_1 \cdot X_k + (I_2 + I_3) \cdot T_3 \cdot X_k + I_4 \cdot T_2 \cdot X_k + I_4 \cdot T_3 \cdot Y_k \\ Rk_{out} &= I_1 \cdot T_1 \cdot Y_k + (I_2 + I_3) \cdot (T_1X_k + T_2Y_k) + I_4 \cdot T_1X_k + I_4 \cdot T_2Y_k \end{aligned}$$

The control signals for the temporary register, T, are

$$\begin{aligned} T_{in} &= T_1I_4 \\ T_{out} &= T_3I_4 \end{aligned}$$

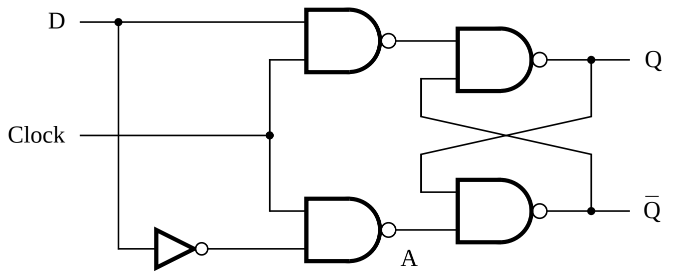
7.31. (a) $\text{Period} = 2 \times n \times t_p$
(b)

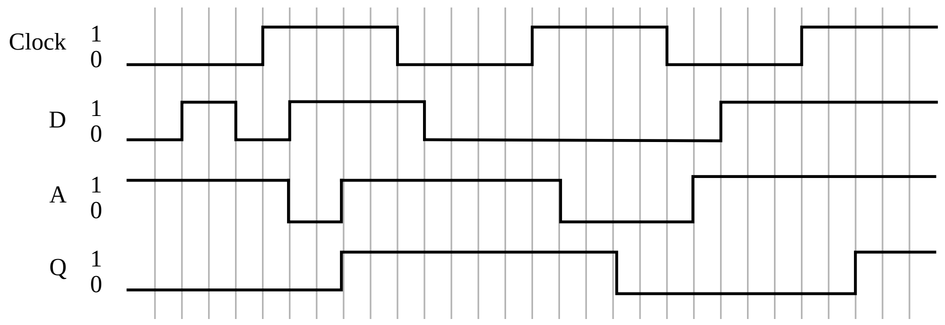


The counter tallies the number of pulses in the 100 ns time period. Thus

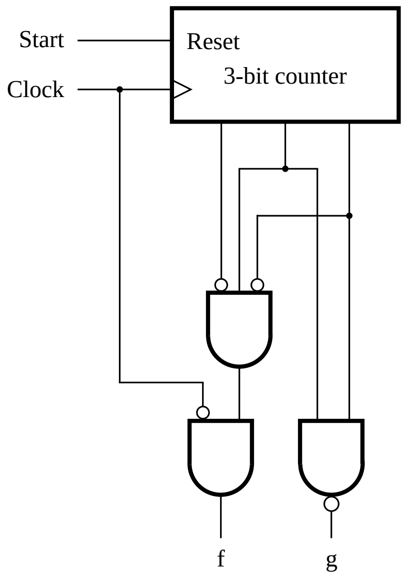
$$t_p = \frac{100 \text{ ns}}{2 \times Count \times n}$$

7.32.



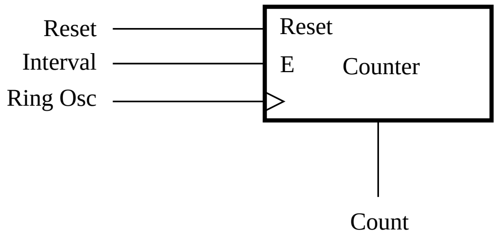


7.33.



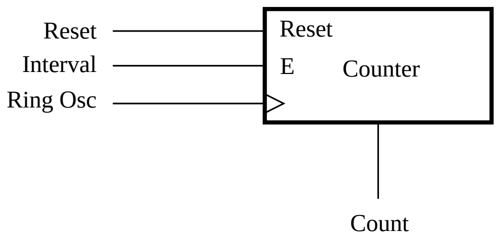
7.34. With non-blocking assignments, the result of the assignment f <= A[1] & A[0] is not seen by the successive assignments inside the **for** loop. Thus, *f* has an uninitialized value when the **for** loop is entered. Similarly, each **for** loop interation sees the unitialized value of *f*. The result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

7.35.



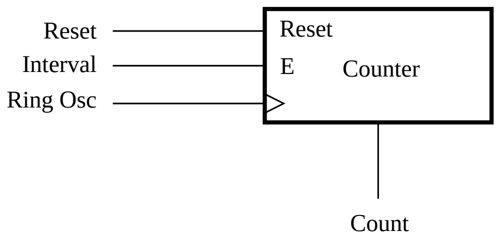
The counting sequence is: 001, 110, 011, 111, 101, 100, 010, 001

7.36.



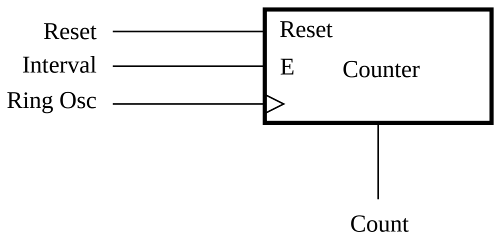
The counting sequence is: 001, 101, 111, 110, 011, 100, 010, 001

7.37.



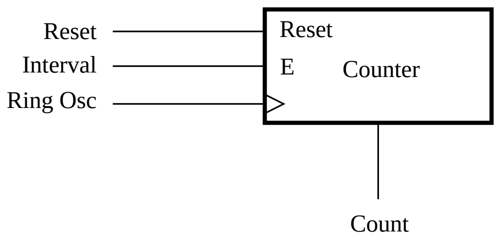
The counting sequence is: 001, 100, 000, 000, ...

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
```

Chapter 8

8.1. The expressions for the inputs of the flip-flops are

$$\begin{aligned} D_2 &= Y_2 = \overline{w}y_2 + \overline{y}_1\overline{y}_2 \\ D_1 &= Y_1 = w \oplus y_1 \oplus y_2 \end{aligned}$$

The output equation is

$$z = y_1 y_2$$

8.2. The excitation table for JK flip-flops is

Present state y_2y_1	Flip-flop inputs				Output z
	$w=0$		$w=1$		
	J_2K_2	J_1K_1	J_2K_2	J_1K_1	
00	1d	0d	1d	1d	0
01	0d	d0	0d	d1	0
10	d0	1d	d1	0d	0
11	d0	d1	d1	d0	1

The expressions for the inputs of the flip-flops are

$$\begin{aligned} J_2 &= \overline{y}_1 \\ K_2 &= w \\ J_1 &= \overline{w}y_2 + w\overline{y}_2 \\ K_1 &= J_1 \end{aligned}$$

The output equation is

$$z = y_1 y_2$$

8.3. A possible state table is

Present state	Next state		Output z	
	$w=0$	$w=1$	$w=0$	$w=1$
A	A	B	0	0
B	E	C	0	0
C	E	D	0	0
D	E	D	0	1
E	F	B	0	0
F	A	B	0	1

8.4. Verilog code for the solution given in problem 8.3 is

```
module prob8_4 (Clock, Resetn, w, z);
    input Clock, Resetn, w;
    output z;
    reg z;
    reg [3:1] y, Y;
    parameter [3:1] A = 3'b000, B = 3'b001, C = 3'b010, D = 3'b011, E = 3'b100, F = 3'b101;
    // Define the next state and output combinational circuits
    always @(w or y)
        case (y)
            A:  if (w)  begin
                            Y = B;   z = 0;
                        end
                else    begin
                            Y = A;   z = 0;
                        end
            B:  if (w)  begin
                            Y = C;   z = 0;
                        end
                else    begin
                            Y = E;   z = 0;
                        end
            C:  if (w)  begin
                            Y = D;   z = 0;
                        end
                else    begin
                            Y = E;   z = 0;
                        end
            D:  if (w)  begin
                            Y = D;   z = 1;
                        end
                else    begin
                            Y = E;   z = 0;
                        end
            E:  if (w)  begin
                            Y = B;   z = 0;
                        end
                else    begin
                            Y = F;   z = 0;
                        end
            F:  if (w)  begin
                            Y = B;   z = 1;
                        end
                else    begin
                            Y = A;   z = 0;
                        end
            default:    begin
                            Y = 3'bxxx;   z = 0;
                        end
        endcase
    // Define the sequential block
    always @(negedge Resetn or posedge Clock)
        if  (Resetn == 0)   y <= A;
        else   y <= Y;

endmodule
```

8.5. A minimal state table is

Present state	Next State		Output
	$w = 0$	$w = 1$	z
A	A	B	0
B	E	C	0
C	D	C	0
D	A	F	1
E	A	F	0
F	E	C	1

8.6. An initial attempt at deriving a state table may be

Present state	Next state		Output z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	A	B	0	0
B	D	C	0	0
C	D	C	1	0
D	A	E	0	1
E	D	C	0	0

States B and E are equivalent; hence the minimal state table is

Present state	Next state		Output z	
	$w = 0$	$w = 1$	$w = 0$	$w = 1$
A	A	B	0	0
B	D	C	0	0
C	D	C	1	0
D	A	B	0	1

8.7. For Figure 8.51 have (using the straightforward state assignment):

	Present state $y_3y_2y_1$	Next state $w=0$ $Y_3Y_2Y_1$	Next state $w=1$ $Y_3Y_2Y_1$	Output z
A	0 0 0	0 0 1	0 1 0	1
B	0 0 1	0 1 1	1 0 1	1
C	0 1 0	1 0 1	1 0 0	0
D	0 1 1	0 0 1	1 1 0	1
E	1 0 0	1 0 1	0 1 0	0
F	1 0 1	1 0 0	0 1 1	0
G	1 1 0	1 0 1	1 1 0	0

This leads to

$$\begin{aligned} Y_3 &= \overline{w}y_3 + \overline{y}_1y_2 + wy_1\overline{y}_3 \\ Y_2 &= wy_3 + w\overline{y}_1\overline{y}_2 + wy_1y_2 + \overline{w}y_1\overline{y}_2\overline{y}_3 \\ Y_1 &= \overline{y}_3\overline{w} + \overline{y}_1\overline{w} + wy_1\overline{y}_2 \\ z &= y_1\overline{y}_3 + \overline{y}_2\overline{y}_3 \end{aligned}$$

For Figure 8.52 have

	Present state y_2y_1	Next state $w=0$ Y_2Y_1	Next state $w=1$ Y_2Y_1	Output z
A	0 0	0 1	1 0	1
B	0 1	0 0	1 1	1
C	1 0	1 1	1 0	0
F	1 1	1 0	0 0	0

This leads to

$$\begin{aligned} Y_2 &= \overline{w}y_2 + \overline{y}_1y_2 + w\overline{y}_2 \\ Y_1 &= \overline{y}_1\overline{w} + wy_1\overline{y}_2 \\ z &= \overline{y}_2 \end{aligned}$$

Clearly, minimizing the number of states leads to a much simpler circuit.

8.8. For Figure 8.55 have (using straightforward state assignment):

	Present state $y_4y_3y_2y_1$	Next state DN=00	01	10	11	Output z
		$Y_4Y_3Y_2Y_1$				
S1	0 0 0 0	0 0 0 0	0 0 1 0	0 0 0 1	–	0
S2	0 0 0 1	0 0 0 1	0 0 1 1	0 1 0 0	–	0
S3	0 0 1 0	0 0 1 0	0 1 0 1	0 1 1 0	–	0
S4	0 0 1 1	0 0 0 0	–	–	–	1
S5	0 1 0 0	0 0 1 0	–	–	–	1
S6	0 1 0 1	0 1 0 1	0 1 1 1	1 0 0 0	–	0
S7	0 1 1 0	0 0 0 0	–	–	–	1
S8	0 1 1 1	0 0 0 0	–	–	–	1
S9	1 0 0 0	0 0 1 0	–	–	–	1

The next-state and output expressions are

$$\begin{aligned} Y_4 &= Dy_3 \\ Y_3 &= Dy_1 + Dy_2 + Ny_2 + \overline{D}y_3\overline{y}_2y_1 \\ Y_2 &= N\overline{y}_2 + y_3\overline{y}_1 + \overline{N}\overline{y}_3y_2\overline{y}_1 \\ Y_1 &= Ny_2 + D\overline{y}_2\overline{y}_1 + \overline{D}\overline{y}_2y_1 \\ z &= y_4 + y_1y_2 + \overline{y}_1y_3 \end{aligned}$$

Using the same approach for Figure 8.56 gives

	Present state $y_3y_2y_1$	Next state DN=00	01	10	11	Output z
		$Y_3Y_2Y_1$				
S1	0 0 0	0 0 0	0 1 0	0 0 1	–	0
S2	0 0 1	0 0 1	0 1 1	1 0 0	–	0
S3	0 1 0	0 1 0	0 0 1	0 1 1	–	0
S4	0 1 1	0 0 0	–	–	–	1
S5	1 0 0	0 1 0	–	–	–	1

The next-state and output expressions are:

$$\begin{aligned} Y_3 &= D\overline{y}_2y_1 \\ Y_2 &= y_3 + \overline{N}y_2\overline{y}_1 + N\overline{y}_2 \\ Y_1 &= \overline{D}\overline{y}_2y_1 + Ny_2\overline{y}_1 + D\overline{y}_3\overline{y}_1 \\ z &= y_3 + y_2y_1 \end{aligned}$$

These expressions define a circuit that has considerably lower cost that the circuit resulting from Figure 8.55.

8.9. To compare individual bits, let $k = w_1 \oplus w_2$. Then, a suitable state table is

Present state	Next state		Output z	
	$k=0$	$k=1$	$k=0$	$k=1$
A	B	A	0	0
B	C	A	0	0
C	D	A	0	0
D	D	A	1	0

The state-assigned table is

Present	Next State		Output	
state	$k=0$	$k=1$	$k=0$	$k=1$
y_2y_1	Y_2Y_1	Y_2Y_1	z	z
00	01	00	0	0
01	10	00	0	0
10	11	00	0	0
11	11	00	1	0

The next-state and output expressions are

$$\begin{aligned} Y_2 &= \overline{k}y_1 + \overline{k}y_2 \\ Y_1 &= \overline{k}\overline{y}_1 + \overline{k}y_2 \\ z &= \overline{k}y_1y_2 \end{aligned}$$

8.10. Verilog code for the solution given in problem 8.9 is

```
module prob8_10 (Clock, Resetn, w1, w2, z);
    input Clock, Resetn, w1, w2;
    output z;
    reg z;
    reg [2:1] y, Y;
    wire k;
    parameter [2:1] A = 2'b00, B = 2'b01, C = 2'b10, D = 2'b11;

    // Define the next state and output combinational circuits
    assign k = w1 ^ w2;
    always @(k or y)
        case (y)
            A:  if (k)  begin
                            Y = A;  z = 0;
                        end
                else    begin
                            Y = B;  z = 0;
                        end
            B:  if (k)  begin
                            Y = A;  z = 0;
                        end
                else    begin
                            Y = C;  z = 0;
                        end
            C:  if (k)  begin
                            Y = A;  z = 0;
                        end
                else    begin
                            Y = D;  z = 0;
                        end
            D:  if (k)  begin
                            Y = A;  z = 0;
                        end
                else    begin
                            Y = D;  z = 1;
                        end
        endcase

    // Define the sequential block
    always @(negedge Resetn or posedge Clock)
        if (Resetn == 0)  y <= A;
        else  y <= Y;

endmodule
```

8.11. A possible minimum state table for a Moore-type FSM is

Present state	Next state		Output z
	$w = 0$	$w = 1$	
A	B	C	0
B	D	E	0
C	E	D	0
D	F	G	0
E	F	F	0
F	A	A	0
G	A	A	1

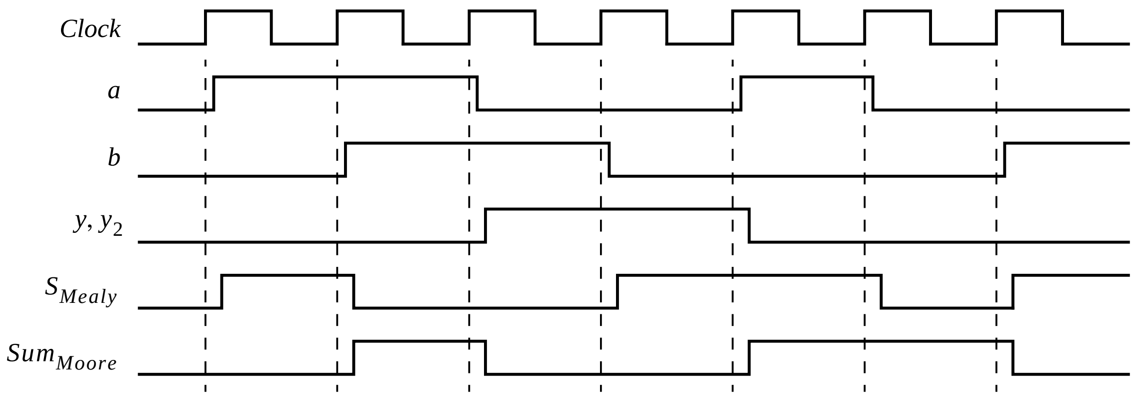
8.12. A minimum state table is shown below. We assume that the 3-bit patterns do not overlap.

Present state	Next state		Output p
	$w = 0$	$w = 1$	
A	B	C	0
B	D	E	0
C	E	D	0
D	A	F	0
E	F	A	0
F	B	C	1

8.13. Verilog code for the solution given in problem 8.12 is

```
module prob8_13 (Clock, Resetn, w, p);
   input Clock, Resetn, w;
   output p;
   reg [3:1] y, Y;
   parameter [3:1] A = 3'b000, B = 3'b001, C = 3'b010, D = 3'b011, E = 3'b100, F = 3'b101;

   // Define the next state combinational circuit
   always @(w or y)
      case (y)
         A: if (w)    Y = C;
            else      Y = B;
         B: if (w)    Y = E;
            else      Y = D;
         C: if (w)    Y = D;
            else      Y = E;
         D: if (w)    Y = F;
            else      Y = A;
         E: if (w)    Y = A;
            else      Y = F;
         F: if (w)    Y = C;
            else      Y = B;
         default:     Y = 3'bxxx;
      endcase

   // Define the sequential block
   always @(negedge Resetn or posedge Clock)
      if (Resetn == 0)   y <= A;
      else   y <= Y;

   // Define output
   assign p = (y == F);
endmodule
```

8.14. The timing diagram is



8.15. The state table corresponding to Figure P8.1 is

Present state	Next state $w=0$	Next state $w=1$	Output z
A	C	D	0
B	B	A	0
C	D	A	0
D	C	B	1

Using one-hot encoding, the state-assigned table is

	Present state $y_4y_3y_2y_1$	Next state $w=0$ $Y_4Y_3Y_2Y_1$	Next state $w=1$ $Y_4Y_3Y_2Y_1$	Output z
A	0 0 0 1	0 1 0 0	1 0 0 0	0
B	0 0 1 0	0 0 1 0	0 0 0 1	0
C	0 1 0 0	1 0 0 0	0 0 0 1	0
D	1 0 0 0	0 1 0 0	0 0 1 0	1

The next-state expressions are

$$\begin{aligned} D_4 &= Y_4 = \overline{w}y_3 + wy_1 \\ D_3 &= Y_3 = \overline{w}(y_1 + y_4) \\ D_2 &= Y_2 = \overline{w}y_2 + wy_4 \\ D_1 &= Y_1 = w(y_2 + y_1) \end{aligned}$$

The output is given by $z = y_4$.

8.16. The state-assignment given in problem 8.15 can be used, except that the state variable y_1 should be complemented. Thus, the state assignment will be $y_4y_3y_2y_1 = 0000, 0011, 0101$, and 1001, for the states A, B, C, and D, respectively. The circuit derived in problem 8.15 can be used, except that the signal for the state variable y_1 should be taken from the $\overline{\text{Q}}$ output of flip-flop 1, rather than from its Q output.

8.17. The partitioning process gives

$$\begin{aligned} P_1 &= (ABCDEFG) \\ P_2 &= (ABD)(CEFG) \\ P_3 &= (ABD)(CEG)(F) \\ P_4 &= (ABD)(CEG)(F) \end{aligned}$$

The minimum state table is

Present state	Next state		Output z	
	$w=0$	$w=1$	$w=0$	$w=1$
A	A	C	0	0
C	F	C	0	1
F	C	A	0	1

8.18. The partitioning process gives

$$\begin{aligned} P_1 &= (ABCDEFG) \\ P_2 &= (ADG)(BCEF) \\ P_3 &= (AG)(D)(B)(CE)(F) \\ P_4 &= (A)(G)(D)(B)(CE)(F) \end{aligned}$$

The minimized state table is

Present state	Next state		Output z	
	$w=0$	$w=1$	$w=0$	$w=1$
A	B	C	0	0
B	D	–	0	1
C	F	C	0	1
D	B	G	0	0
F	C	D	0	1
G	F	–	0	0

8.19. An implementation for the Moore-type FSM in Figures 8.5.7 and 8.5.6 is given in the solution for problem 8.8. The Mealy-type FSM in Figure 8.58 is described in the form of a state table as

Present state	Next state				Output z			
	DN=00	01	10	11	00	01	10	11
S1	S1	S3	S2	–	0	0	0	1
S2	S2	S1	S3	–	0	1	1	–
S3	S3	S2	S1	–	0	0	1	–

The state-assigned table is

Present	Next state				Output			
state	DN=00	01	10	11	00	01	10	11
y_2y_1	Y_2Y_1	Y_2Y_1	Y_2Y_1	Y_2Y_1	z	z	z	z
00	00	10	01	–	0	0	0	–
01	01	00	10	–	0	1	1	–
10	10	01	00	–	0	0	1	–

The next-state and output expressions are

$$\begin{aligned} Y_2 &= Dy_1 + \overline{D}y_2\overline{N} + N\overline{y}_2\overline{y}_1 \\ Y_1 &= Ny_2 + \overline{D}y_1\overline{N} + D\overline{y}_2\overline{y}_1 \\ z &= Dy_1 + Dy_2 + Ny_1 \end{aligned}$$

In this case, choosing the Mealy model results in a simpler circuit.

8.20. Use w as the clock. Then the state table is

Present state	Next state	Output z_1z_0
A	B	0 0
B	C	1 0
C	D	0 1
D	A	1 1

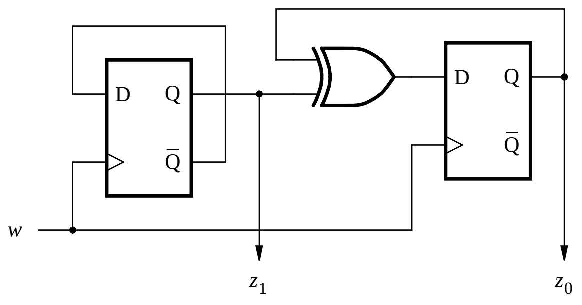
The state-assigned table is

Present state y_1y_0	Next state Y_1Y_0	Output z_1z_0
0 0	1 0	0 0
1 0	0 1	1 0
0 1	1 1	0 1
1 1	0 0	1 1

The next-state expressions are

$$\begin{aligned} Y_1 &= \overline{y}_1 \\ Y_2 &= y_1 \oplus y_2 \end{aligned}$$

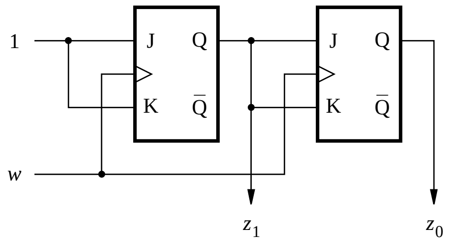
The resulting circuit is



8.21. From the state-assigned table given in the solution to Problem 8.20, the excitation table for JK flip-flops is

Present state y_1y_0	Flip-flop inputs J_1K_1	Flip-flop inputs J_0K_0	Output z_1z_0
0 0	1 d	0 d	0 0
1 0	d 1	1 d	1 0
0 1	1 d	d 0	0 1
1 1	d 1	d 1	1 1

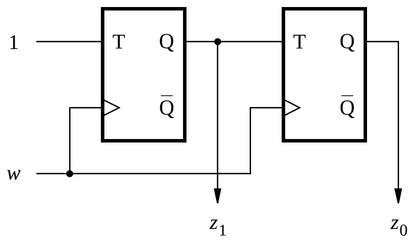
The flip-flop inputs are $J_1 = K_1 = 1$ and $J_2 = K_2 = y_1$. The resulting circuit is



8.22. From the state-assigned table given in the solution to Problem 8.20, the excitation table for T flip-flops is

Present state y_1y_0	Flip-flop inputs T_1	Flip-flop inputs T_0	Output z_1z_0
0 0	1	0	0 0
1 0	1	1	1 0
0 1	1	0	0 1
1 1	1	1	1 1

The flip-flop inputs are $T_1 = 1$ and $T_2 = y_1$. The resulting circuit is



8.23. The state diagram is

Present state	Next state		Output
	$w = 0$	$w = 1$	$z_2z_1z_0$
A	A	B	0 0 0
B	B	C	0 0 1
C	C	D	0 1 0
D	D	E	0 1 1
E	E	F	1 0 0
F	F	A	1 0 1

The state-assigned table is

Present state	Next state		Output
	$w = 0$	$w = 1$	
$y_2y_1y_0$	$Y_2Y_1Y_0$		$z_2z_1z_0$
0 0 0	0 0 0	0 0 1	0 0 0
0 0 1	0 0 1	0 1 0	0 0 1
0 1 0	0 1 0	0 1 1	0 1 0
0 1 1	0 1 1	1 0 0	0 1 1
1 0 0	1 0 0	1 0 1	1 0 0
1 0 1	1 0 1	0 0 0	1 0 1

The next-state expressions are

$$\begin{aligned} Y_2 &= \overline{y}_0y_2 + \overline{w}y_2 + wy_0y_1 \\ Y_1 &= \overline{y}_0y_1 + \overline{w}y_1 + wy_0\overline{y}_1\overline{y}_2 \\ Y_0 &= \overline{w}y_0 + w\overline{y}_0 \end{aligned}$$

The outputs are: $z_2 = y_2$, $z_1 = y_1$, and $z_0 = y_0$.

8.24. Using the state-assigned table given in the solution for problem 8.23, the excitation table for JK flip-flops is

Present state	Flip-flop inputs						Outputs
	$w = 0$			$w = 1$			
$y_2y_1y_0$	J_2K_2	J_1K_1	J_0K_0	J_2K_2	J_1K_1	J_0K_0	$z_2z_1z_0$
000	0 d	0 d	0 d	0 d	0 d	1 d	000
001	0 d	0 d	d 0	0 d	1 d	d 1	001
010	0 d	d 0	0 d	0 d	d 0	1 d	010
011	0 d	d 0	d 0	1 d	d 1	d 1	011
100	d 0	0 d	0 d	d 0	0 d	1 d	100
101	d 0	0 d	d 0	d 1	0 d	d 1	101

The expressions for the inputs of the flip-flops are

$$
\begin{aligned}
J_2 &= wy_1y_0 \\
K_2 &= wy_2y_0 \\
J_1 &= w\overline{y}_2y_0 \\
K_1 &= wy_0 \\
J_0 &= w \\
K_0 &= w
\end{aligned}
$$

The outputs are: $z_2 = y_2$, $z_1 = y_1$, and $z_0 = y_0$.

8.25. Using the state-assigned table given in the solution for problem 8.23, the excitation table for T flip-flops is

Present state	Flip-flop inputs		Outputs
	$w = 0$	$w = 1$	
$y_2y_1y_0$	$T_2T_1T_0$	$T_2T_1T_0$	$z_2z_1z_0$
000	000	001	000
001	000	011	001
010	000	001	010
011	000	111	011
100	000	001	100
101	000	101	101

The expressions for T inputs of the flip-flops are

$$
\begin{aligned}
T_2 &= wy_1y_0 + wy_2y_0 \\
T_1 &= w\overline{y}_2y_0 \\
T_0 &= w
\end{aligned}
$$

The outputs are: $z_2 = y_2$, $z_1 = y_1$, and $z_0 = y_0$.

8.26. The state diagram is

Present state	Next state $w=0$	Next state $w=1$	Count
A	H	C	0
B	A	D	1
C	B	E	2
D	C	F	3
E	D	G	4
F	E	H	5
G	F	A	6
H	G	B	7

The state-assigned table is

	Present state $y_2y_1y_0$	Next state $w=0$ $Y_2Y_1Y_0$	Next state $w=1$ $Y_2Y_1Y_0$	Output $z_2z_1z_0$
A	0 0 0	1 1 1	0 1 0	0 0 0
B	0 0 1	0 0 0	0 1 1	0 0 1
C	0 1 0	0 0 1	1 0 0	0 1 0
D	0 1 1	0 1 0	1 0 1	0 1 1
E	1 0 0	0 1 1	1 1 0	1 0 0
F	1 0 1	1 0 0	1 1 1	1 0 1
G	1 1 0	1 0 1	0 0 0	1 1 0
H	1 1 1	1 1 0	0 0 1	1 1 1

The next-state expressions (inputs to D flip-flops) are

$$\begin{aligned} D_2 &= Y_2 &=& \ w\overline{y}_2y_1 + \overline{w}y_2y_1 + wy_2\overline{y}_1 + \overline{w}y_2y_0 + \overline{y}_2\overline{y}_1\overline{y}_0w \\ D_1 &= Y_1 &=& \ w\overline{y}_1 + \overline{y}_1\overline{y}_0 + \overline{w}y_1y_0 \\ D_0 &= Y_0 &=& \ \overline{y}_0\overline{w} + y_0w \end{aligned}$$

The outputs are: $z_2 = y_2$, $z_1 = y_1$, and $z_0 = y_0$.

8.27. From the state-assigned table given in the solution to problem 8.26, the excitation table for JK flip-flops is

Present state	Flip-flop inputs						Outputs
	$w = 0$			$w = 1$			
$y_2y_1y_0$	J_2K_2	J_1K_1	J_0K_0	J_2K_2	J_1K_1	J_0K_0	$z_2z_1z_0$
000	1 d	1 d	1 d	0 d	1 d	0 d	000
001	0 d	0 d	d 1	0 d	1 d	d 0	001
010	0 d	d 1	1 d	1 d	d 1	0 d	010
011	0 d	d 0	d 1	1 d	d 1	d 0	011
100	d 1	1 d	1 d	d 0	1 d	0 d	100
101	d 0	0 d	d 1	d 0	1 d	d 0	101
110	d 0	d 1	1 d	d 1	d 1	0 d	110
111	d 0	d 0	d 1	d 1	d 1	d 0	111

The expressions for J and K inputs to the three flip-flops are

$$\begin{aligned} J_2 &= y_1w + \overline{y}_1\overline{y}_0\overline{w} \\ K_2 &= J_2 \\ J_1 &= w + \overline{y}_0 \\ K_1 &= J_1 \\ J_0 &= \overline{w} \\ K_0 &= J_0 \end{aligned}$$

The outputs are: $z_2 = y_2$, $z_1 = y_1$, and $z_0 = y_0$.

8.28. From the state-assigned table given in the solution to problem 8.26, the excitation table for T flip-flops is

Present state	Flip-flop inputs		Outputs
	$w = 0$	$w = 1$	
$y_2y_1y_0$	$T_2T_1T_0$	$T_2T_1T_0$	$z_2z_1z_0$
000	111	010	000
001	001	010	001
010	011	110	010
011	001	110	011
100	111	010	100
101	001	010	101
110	011	110	110
111	001	110	111

The expressions for T inputs of the flip-flops are

$$\begin{aligned} T_2 &= \overline{y}_1\overline{y}_0\overline{w} + y_1w \\ T_1 &= w + \overline{y}_0 \\ T_0 &= \overline{w} \end{aligned}$$

The outputs are: $z_2 = y_2$, $z_1 = y_1$, and $z_0 = y_0$.

8.29. The next-state and output expressions are

$$\begin{aligned} D_1 &= Y_1 &= w(y_1 + y_2) \\ D_2 &= Y_2 &= w(\overline{y}_1 + \overline{y}_2) \\ & \quad z &= y_1\overline{y}_2 \end{aligned}$$

The corresponding state-assigned table is

Present state y_2y_1	Next state $w=0$ Y_2Y_1	Next state $w=1$ Y_2Y_1	Output z
0 0	0 0	1 0	0
0 1	0 0	1 1	1
1 0	0 0	1 1	0
1 1	0 0	0 1	0

This leads to the state table

Present state	Next state $w=0$	Next state $w=1$	Output z
A	A	C	0
B	A	D	1
C	A	D	0
D	A	B	0

The circuit produces $z = 1$ whenever the input sequence on w comprises a 0 followed by an even number of 1s.

8.30. The Verilog code based on the style of code in Figure 8.29 is

```
module prob8_30 (Clock, Resetn, D, N, z);
    input Clock, Resetn, D, N;
    output z;
    reg [3:1] y, Y;
    wire [1:0] K;
    parameter [3:1] S1 = 3'b000, S2 = 3'b001, S3 = 3'b010, S4 = 3'b011, S5 = 3'b100;

    // Define the next state combinational circuit
    assign K = {D, N};
    always @(K or y)
        case (y)
            S1: if (K == 2'b00)   Y = S1;
                else if (K == 2'b01)   Y = S3;
                else if (K == 2'b10)   Y = S2;
                else   Y = 3'bxxx;
            S2: if (K == 2'b00)   Y = S2;
                else if (K == 2'b01)   Y = S4;
                else if (K == 2'b10)   Y = S5;
                else   Y = 3'bxxx;
            S3: if (K == 2'b00)   Y = S3;
                else if (K == 2'b01)   Y = S2;
                else if (K == 2'b10)   Y = S4;
                else   Y = 3'bxxx;
            S4: if (K == 2'b00)   Y = S1;
                else   Y = 3'bxxx;
            S5: if (K == 2'b00)   Y = S3;
                else   Y = 3'bxxx;
            default:   Y = 3'bxxx;
        endcase

    // Define the sequential block
    always @(negedge Resetn or posedge Clock)
        if (Resetn == 0)   y <= S1;
        else   y <= Y;

    // Define output
    assign z = (y == S4) | (y == S5);

endmodule
```

8.31. The Verilog code based on the style of code in Figure 8.34 is

```
module prob8_31 (Clock, Resetn, D, N, z);
    input Clock, Resetn, D, N;
    output z;
    reg [3:1] y;
    wire [1:0] K;
    parameter [3:1] S1 = 3'b000, S2 = 3'b001, S3 = 3'b010, S4 = 3'b011, S5 = 3'b100;

    assign K = {D, N};
    // Define the sequential block
    always @(negedge Resetn or posedge Clock)
        if (Resetn == 0)   y <= S1;
        else
            case (y)
                S1: if (K == 2'b00)   y <= S1;
                      else if (K == 2'b01)   y <= S3;
                      else if (K == 2'b10)   y <= S2;
                      else   y <= 3'bxxx;
                S2: if (K == 2'b00)   y <= S2;
                      else if (K == 2'b01)   y <= S4;
                      else if (K == 2'b10)   y <= S5;
                      else   y <= 3'bxxx;
                S3: if (K == 2'b00)   y <= S3;
                      else if (K == 2'b01)   y <= S2;
                      else if (K == 2'b10)   y <= S4;
                      else   y <= 3'bxxx;
                S4: if (K == 2'b00)   y <= S1;
                      else   y <= 3'bxxx;
                S5: if (K == 2'b00)   y <= S3;
                      else   y <= 3'bxxx;
                default:   y <= 3'bxxx;
            endcase

    // Define output
    assign z = (y == S4) | (y == S5);

endmodule
```

8.32. The Verilog code based on the style of code in Figure 8.29 is

```
module prob8_32 (Clock, Resetn, D, N, z);
    input Clock, Resetn, D, N;
    output z;
    reg z;
    reg [2:1] y, Y;
    wire [1:0] K;
    parameter [2:1] S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;
```

cont'd

```
    // Define the next state and output combinational circuits
    assign K = {D, N};
    always @(K or y)
        case (y)
            S1:  if (K == 2'b00)            begin
                                                Y = S1;   z = 0;
                                            end
                 else if (K == 2'b01)       begin
                                                Y = S3;   z = 0;
                                            end
                 else if (K == 2'b10)       begin
                                                Y = S2;   z = 0;
                                            end
                 else                       begin
                                                Y = 2'bxx;   z = 1'bx;
                                            end
            S2:  if (K == 2'b00)            begin
                                                Y = S2;   z = 0;
                                            end
                 else if (K == 2'b01)       begin
                                                Y = S1;   z = 1;
                                            end
                 else if (K == 2'b10)       begin
                                                Y = S3;   z = 1;
                                            end
                 else                       begin
                                                Y = 2'bxx;   z = 1'bx;
                                            end
            S3:  if (K == 2'b00)            begin
                                                Y = S3;   z = 0;
                                            end
                 else if (K == 2'b01)       begin
                                                Y = S2;   z = 0;
                                            end
                 else if (K == 2'b10)       begin
                                                Y = S1;   z = 1;
                                            end
                 else                       begin
                                                Y = 2'bxx;   z = 1'bx;
                                            end
            default:                        begin
                                                Y = 2'bxx;   z = 1'bx;
                                            end
        endcase

    // Define the sequential block
    always @(negedge Resetn or posedge Clock)
        if  (Resetn == 0)   y <= S1;
        else   y <= Y;
endmodule
```

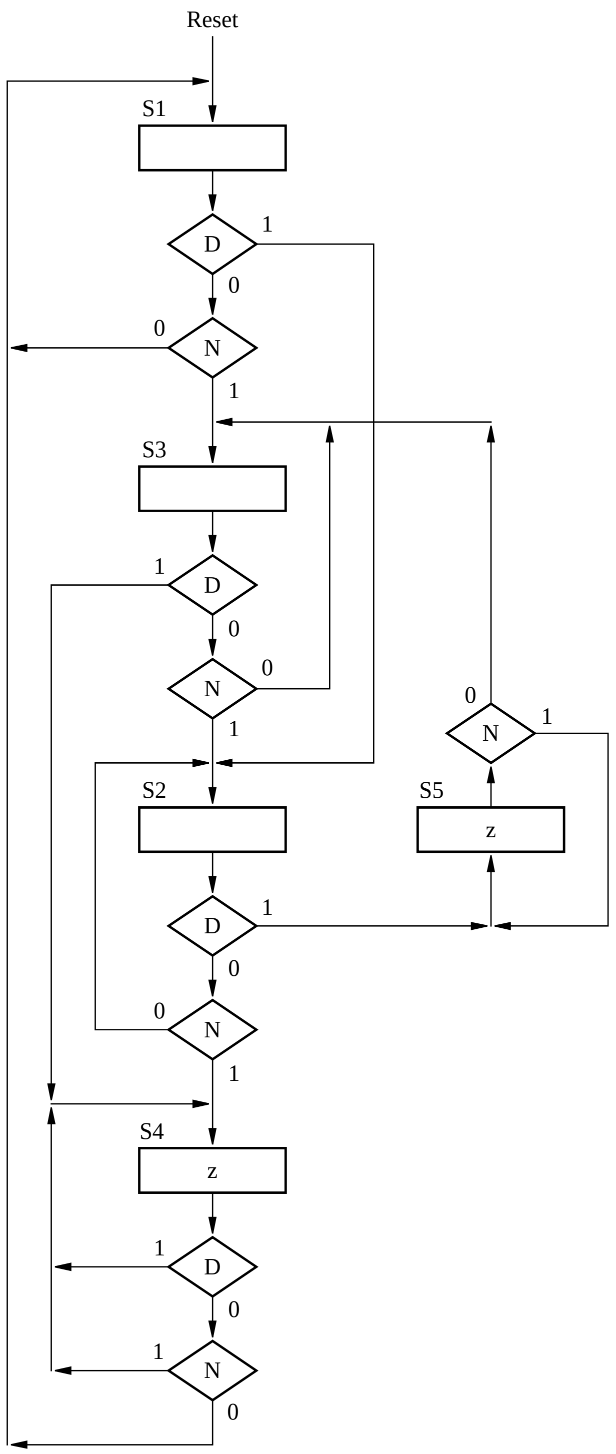
8.33. The Verilog code based on the style of code in Figure 8.34 is

```
module prob8_33 (Clock, Resetn, D, N, z);
    input Clock, Resetn, D, N;
    output z;
    reg [2:1] y;
    wire [1:0] K;
    parameter [2:1] S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;

    assign K = {D, N};
    // Define the sequential block
    always @(negedge Resetn or posedge Clock)
        if (Resetn == 0)   y <= S1;
        else
            case (y)
                S1: if (K == 2'b00)   y <= S1;
                     else if (K == 2'b01)   y <= S3;
                     else if (K == 2'b10)   y <= S2;
                     else   y <= 2'bxx;
                S2: if (K == 2'b00)   y <= S2;
                     else if (K == 2'b01)   y <= S1;
                     else if (K == 2'b10)   y <= S3;
                     else   y <= 2'bxx;
                S3: if (K == 2'b00)   y <= S3;
                     else if (K == 2'b01)   y <= S2;
                     else if (K == 2'b10)   y <= S1;
                     else   y <= 2'bxx;
                default:   y <= 2'bxx;
            endcase

    // Define output
    assign z = ((y == S2) & ((K == 2'b01) | (K == 2'b10))) | ((y == S3) & (K == 2'b10));

endmodule
```

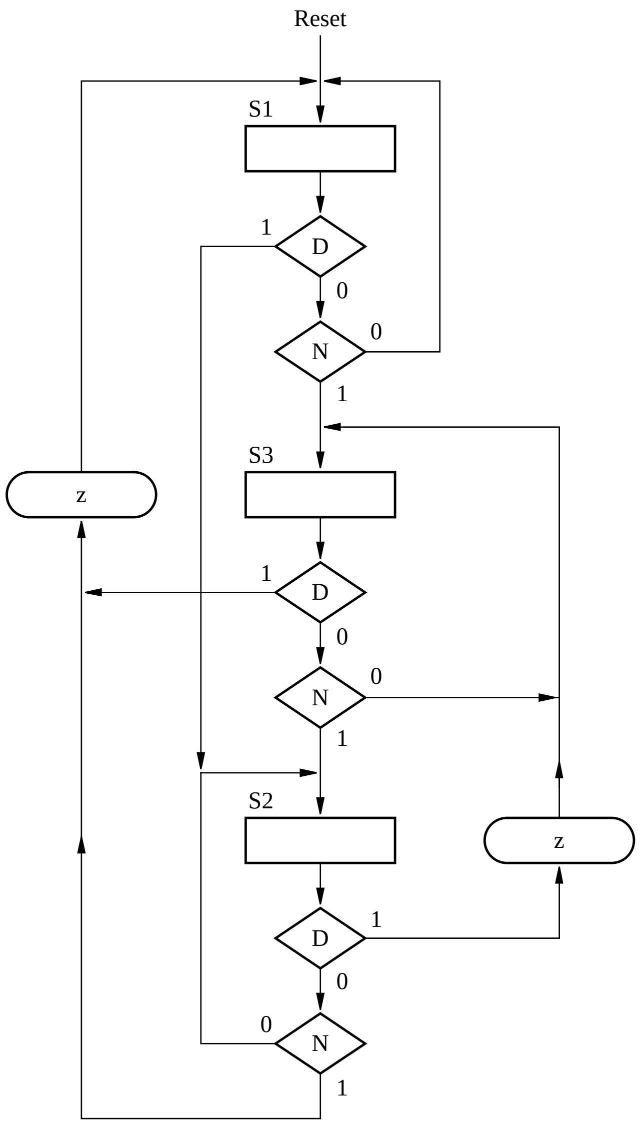
8.34. Verilog code for the FSM in Figure P8.2 is

```
module  prob8_34 (Clock, Resetn, w, z);
    input  Clock, Resetn, w;
    output  z;
    reg  [2:1] y, Y;
    parameter  [2:1] A = 2'b00, B = 2'b01, C = 2'b10, D = 2'b11;

    // Define the next state combinational circuit
    always @(w or y)
        case (y)
            A:  if (w)    Y = C;
                else      Y = A;
            B:  if (w)    Y = D;
                else      Y = A;
            C:  if (w)    Y = D;
                else      Y = A;
            D:  if (w)    Y = B;
                else      Y = A;
        endcase

    // Define the sequential block
    always @(negedge Resetn or posedge Clock)
        if  (Resetn == 0)   y <= A;
        else   y <= Y;

    // Define output
    assign  z = (y == B);
endmodule
```

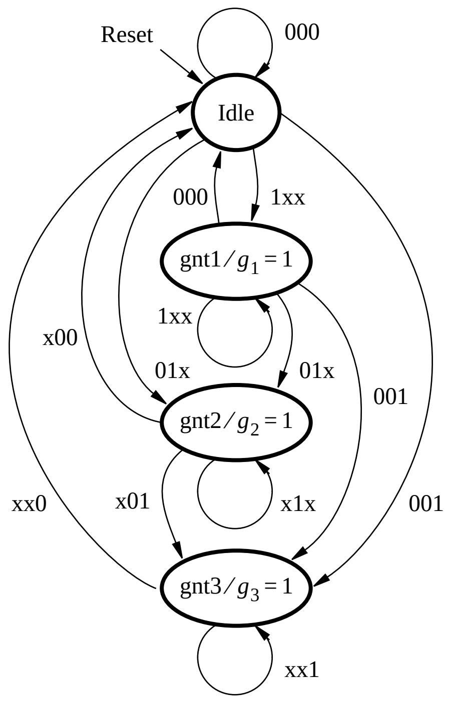
8.35. An ASM chart for the FSM in Figure 8.57 is



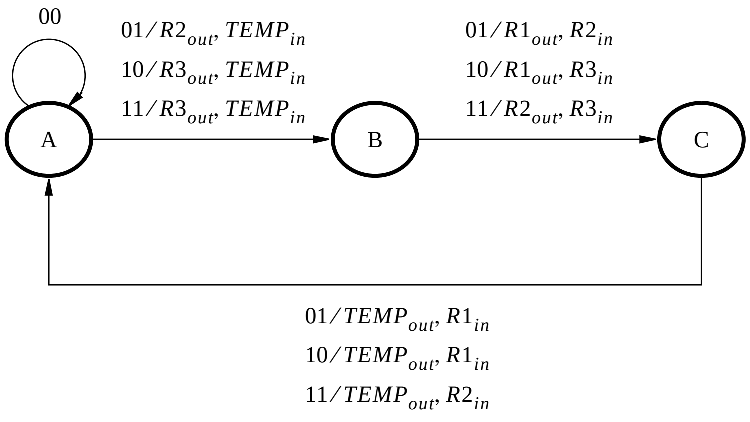
8.36. An ASM chart for the FSM in Figure 8.58 is



8.37. To ensure that the device 3 will get serviced the FSM in Figure 8.72 can be modified as follows:



8.39. The required control signals can be generated using the following FSM:



00
01/$R2_{out}$, $TEMP_{in}$
10/$R3_{out}$, $TEMP_{in}$
11/$R3_{out}$, $TEMP_{in}$
01/$R1_{out}$, $R2_{in}$
10/$R1_{out}$, $R3_{in}$
11/$R2_{out}$, $R3_{in}$
A
B
C
01/$TEMP_{out}$, $R1_{in}$
10/$TEMP_{out}$, $R1_{in}$
11/$TEMP_{out}$, $R2_{in}$

Let $k = w_2 + w_1$. Then the next-state transitions can be defined as

Present state	Next state	
	$k=0$	$k=1$
A	A	B
B	B	C
C	C	A

Using one-hot encoding, the state-assigned table becomes

Present state	Next state	
	$k=0$	$k=1$
$y_3y_2y_1$	$Y_3Y_2Y_1$	$Y_3Y_2Y_1$
001	001	010
010	010	100
100	100	001

The next-state expressions are

$$\begin{aligned} Y_3 &= \overline{k}y_3 + ky_2 \\ Y_2 &= \overline{k}y_2 + ky_1 \\ Y_1 &= \overline{k}y_1 + ky_3 \end{aligned}$$

The output expressions are

$$\begin{aligned} TEMP_{in} &= ky_1 \\ TEMP_{out} &= ky_3 \\ R1_{out} &= y_2(w_2 \oplus w_1) \\ R1_{in} &= y_3(w_2 \oplus w_1) \\ R2_{out} &= y_1\overline{w}_2w_1 + y_2w_2w_1 \\ R2_{in} &= y_2\overline{w}_2w_1 + y_3w_2w_1 \\ R3_{out} &= y_1w_2 \\ R3_{in} &= y_2w_2 \end{aligned}$$

Chapter 9

9.1. The next-state and output expressions for the circuit in Figure P9.1 are

$$\begin{aligned}
Y_1 &= \overline{w}_1 + y_1\overline{y}_2 \\
Y_2 &= \overline{w}_2 + \overline{y}_1 + w_1 y_2 \\
z_1 &= \overline{y}_1 \\
z_2 &= \overline{y}_2
\end{aligned}$$

This gives the excitation table

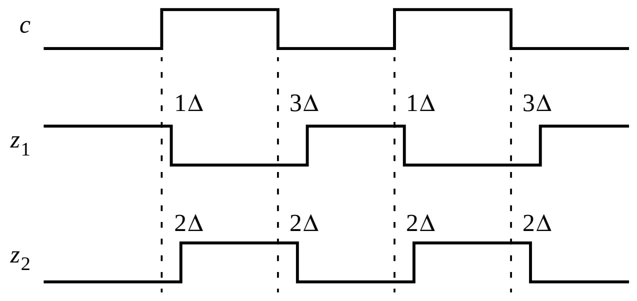
	Present state	Next state				z_2z_1
	state	$w_2w_1 = 00$	01	10	11	
	y_2y_1	Y_2Y_1				
A	00	11	10	11	10	11
B	01	11	11	$\textcircled{01}$	$\textcircled{01}$	10
C	10	11	$\textcircled{10}$	11	$\textcircled{10}$	01
D	11	$\textcircled{11}$	10	01	10	00

The resulting flow table is

Present state	Next state				z_2z_1
	$w_2w_1 = 00$	01	10	11	
A	D	C	D	C	11
B	D	D	$\textcircled{B}$	$\textcircled{B}$	10
C	D	$\textcircled{C}$	D	$\textcircled{C}$	01
D	$\textcircled{D}$	C	B	C	00

The behavior is the same as described in the flow table in Figure 9.21a, if the state interchanges A $\leftrightarrow$ D and B $\leftrightarrow$ C are made.

9.2. The waveforms are



The flow table is

Present state	Next state		Outputs, z_2z_1	
	$C=0$	1	0	1
0	1	⓪	00	10
1	①	0	01	00

The circuit generates a non-overlapping 2-phase clock.

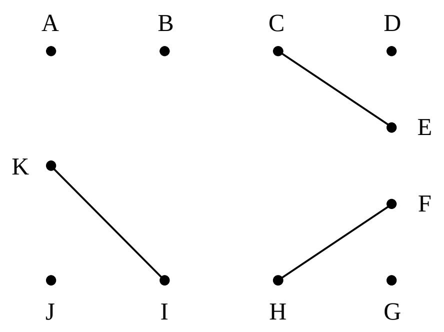
9.3. The partitioning procedure gives

$$\begin{aligned}
P_1 &= (ADGJMPT)(BEHR)(CF)(ILOSV)(KNU) \\
P_2 &= (AD)(GP)(JMT)(B)(E)(HR)(C)(F)(ILOSV)(KNU) \\
P_3 &= (A)(D)(GP)(JMT)(B)(E)(HR)(C)(F)(ILOSV)(KNU) \\
P_4 &= P_3
\end{aligned}$$

This gives the flow table

Present state	Next state				z
	$w_2w_1 = 00$	01	10	11	
A	Ⓐ	B	C	–	0
B	D	Ⓑ	–	–	0
C	G	–	Ⓒ	–	0
D	Ⓓ	E	F	–	0
E	G	Ⓔ	–	–	0
F	J	–	Ⓕ	–	0
G	Ⓖ	H	I	–	0
H	J	Ⓗ	–	–	0
I	A	–	Ⓘ	–	1
J	Ⓙ	K	I	–	0
K	A	Ⓚ	–	–	1

The corresponding merger diagram is



This leads to the reduced flow table

Present state	Next state				z
	$w_2w_1 = 00$	01	10	11	
A	Ⓐ	B	C	–	0
B	D	Ⓑ	–	–	0
C	G	Ⓒ	Ⓒ	–	0
D	Ⓓ	C	F	–	0
F	J	Ⓕ	Ⓕ	–	0
G	Ⓖ	F	I	–	0
I	A	Ⓘ	Ⓘ	–	1
J	Ⓙ	I	I	–	0

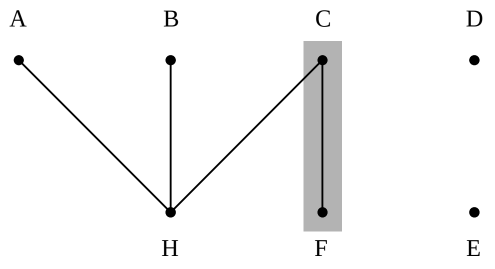
9.4. The partitioning procedure gives

$$\begin{aligned}
P_1 &= (AF)(BEGL)(CJ)(DK)(HM)\\
P_2 &= (AF)(BG)(EL)(CJ)(DK)(HM)\\
P_3 &= (A)(F)(BG)(EL)(CJ)(DK)(HM)\\
P_4 &= P_3
\end{aligned}$$

Replacing states B and G, E and L, C and J, D and K, and H and M with new states B, E, C, D, and H, respectively, produces the following flow table:

Present state	Next state				Output
	$w_2w_1 = 00$	01	10	11	z
A	Ⓐ	B	C	–	0
B	D	Ⓑ	–	H	0
C	F	–	Ⓒ	H	0
D	Ⓓ	E	C	–	1
E	A	Ⓔ	–	H	0
F	Ⓕ	E	C	–	0
H	–	B	C	Ⓗ	1

The merger diagram is



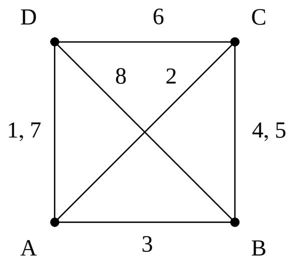
Only C and F can be merged if the Moore model is to be preserved. Therefore, the reduced flow table is

Present state	Next state				z
	$w_2w_1 = 00$	01	10	11	
A	Ⓐ	B	C	–	0
B	D	Ⓑ	–	H	0
C	Ⓒ	E	Ⓒ	H	0
D	Ⓓ	E	C	–	1
E	A	Ⓔ	–	H	0
H	–	B	C	Ⓗ	1

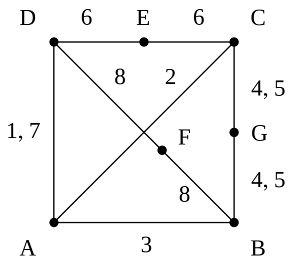
9.5. Relabel the flow table as

Present state	Next state				Output z			
	$w_2w_1 = 00$	01	10	11	00	01	10	11
A	$\textcircled{1}$	3	7	$\textcircled{2}$	0	–	1	1
B	5	$\textcircled{3}$	$\textcircled{4}$	8	0	1	0	0
C	$\textcircled{5}$	$\textcircled{6}$	4	2	0	1	0	1
D	1	6	$\textcircled{7}$	$\textcircled{8}$	–	–	1	0

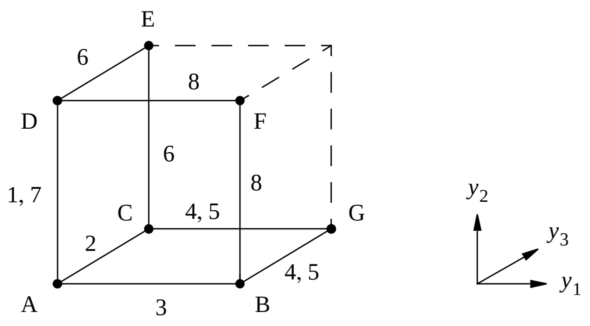
The transition diagram is



The diagonal transitions cannot be avoided without introducing additional states. A possible modification is



This transition diagram can be embedded onto a 3-cube as follows:



Then the modified flow table is

Present state	Next state				Output z			
	$w_2w_1 = 00$	01	10	11	00	01	10	11
A	$\textcircled{A}$	B	D	$\textcircled{A}$	0	–	1	1
B	G	$\textcircled{B}$	$\textcircled{B}$	F	0	1	0	0
C	$\textcircled{C}$	$\textcircled{C}$	G	A	0	1	0	1
D	A	E	$\textcircled{D}$	$\textcircled{D}$	–	–	1	0
E	–	C	–	–	–	1	–	–
F	–	–	–	D	–	–	–	0
G	C	–	B	–	0	–	0	–

The excitation table is

	Present state	Next state				Output			
		$w_2w_1 = 00$	01	10	11	00	01	10	11
	$y_3y_2y_1$	$Y_3Y_2Y_1$				z			
A	000	$\textcircled{000}$	$\textcircled{001}$	010	$\textcircled{000}$	0	–	1	1
B	001	101	$\textcircled{001}$	$\textcircled{001}$	011	0	1	0	0
C	100	$\textcircled{100}$	$\textcircled{100}$	101	000	0	1	0	1
D	010	000	110	$\textcircled{010}$	$\textcircled{010}$	–	–	1	0
E	110	–	100	–	–	–	1	–	–
F	011	–	–	–	010	–	–	–	0
G	101	100	–	001	–	0	–	0	–

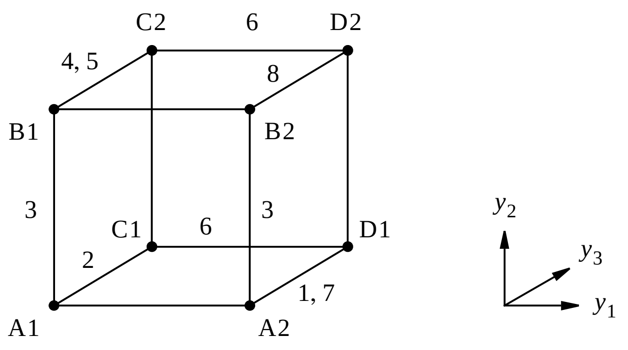
The next-state and output expressions are

$$
\begin{aligned}
Y_3 &= \overline{w}_2 y_3 + \overline{w}_2\overline{w}_1 y_1 + \overline{w}_1 y_3 y_1 \\
Y_2 &= w_2\overline{w}_1\overline{y}_3\overline{y}_1 + w_1\overline{y}_3 y_2\overline{y}_1 + w_2\overline{y}_3 y_2\overline{y}_1 + w_2 w_1\overline{y}_3\overline{y}_2 y_1 \\
Y_2 &= \overline{y}_3\overline{y}_2 y_1 + \overline{w}_2 w_1\overline{y}_3\overline{y}_2 + w_2\overline{w}_1 y_3\overline{y}_2 + w_2\overline{w}_1\overline{y}_2 y_1 \\
z &= \overline{w}_2 w_1 + w_1\overline{y}_2\overline{y}_1 + w_2\overline{w}_1\overline{y}_3\overline{y}_1
\end{aligned}
$$

9.6. Relabel the flow table in Figure 9.42 as

Present state	Next state				Output z			
	$w_2w_1 = 00$	01	10	11	00	01	10	11
A	$\textcircled{1}$	3	7	$\textcircled{2}$	0	–	1	1
B	5	$\textcircled{3}$	$\textcircled{4}$	8	0	1	0	0
C	$\textcircled{5}$	$\textcircled{6}$	4	2	0	1	0	1
D	1	6	$\textcircled{7}$	$\textcircled{8}$	–	–	1	0

Using pairs of equivalent states gives the following transition diagram:



Therefore, the modified flow table is

Present state	Next state				Output z			
	$w_2w_1 = 00$	01	10	11	00	01	10	11
A1	$\textcircled{A1}$	B1	A2	$\textcircled{A1}$	0	–	1	1
A2	$\textcircled{A2}$	B2	D1	$\textcircled{A2}$	0	–	1	1
B1	C2	$\textcircled{B1}$	$\textcircled{B1}$	B2	0	1	0	0
B2	B1	$\textcircled{B2}$	$\textcircled{B2}$	D2	0	1	0	0
C1	$\textcircled{C1}$	$\textcircled{C1}$	C2	A1	0	1	0	1
C2	$\textcircled{C2}$	$\textcircled{C2}$	B1	C1	0	1	0	1
D1	A2	C1	$\textcircled{D1}$	$\textcircled{D1}$	–	–	1	0
D2	D1	C2	$\textcircled{D2}$	$\textcircled{D2}$	–	–	1	0

The excitation table is

	Present state	Next state				Output z			
	$y_3y_2y_1$	$w_2w_1 = 00$	01	10	11	00	01	10	11
		$Y_3Y_2Y_1$				z			
A1	000	000	010	001	000	0	–	1	1
A2	001	001	011	101	001	0	–	1	1
B1	010	110	010	010	011	0	1	0	0
B2	011	010	011	011	111	0	1	0	0
C1	100	100	100	110	000	0	1	0	1
C2	110	110	110	010	100	0	1	0	1
D1	101	001	100	101	101	–	–	1	0
D2	111	101	110	111	111	–	–	1	0

The next-state and output expressions are

$$\begin{aligned}
Y_3 &= \overline{w}_2\overline{w}_1 y_2 \overline{y}_1 + \overline{w}_1 y_3 \overline{y}_2 \overline{y}_1 + w_2 \overline{w}_1 \overline{y}_2 y_1 + w_2 w_1 y_2 y_1 + \\
&\quad y_3 y_2 y_1 + \overline{w}_2 w_1 y_3 + w_1 y_3 y_2 + w_1 y_3 y_1 \\
Y_2 &= \overline{y}_3 y_2 + \overline{w}_2 w_1 y_3 + \overline{w}_1 y_2 \overline{y}_1 + \overline{w}_2 w_1 y_2 + w_2 y_2 y_1 + w_2 \overline{w}_1 y_3 \overline{y}_2 \\
Y_1 &= w_2 y_1 + \overline{w}_1 \overline{y}_2 y_1 + w_1 \overline{y}_3 y_1 + w_2 \overline{w}_1 \overline{y}_3 \overline{y}_2 \\
z &= \overline{w}_2 y_1 + w_2 \overline{y}_3 \overline{y}_2 + w_1 y_3 \overline{y}_2 + \overline{w}_1 y_3 y_1
\end{aligned}$$

9.7. Using the one-hot encoding, the FSM in Figure 9.42 can be implemented as

State assignment	Present state	Next state				Output z			
		$w_2w_1 = 00$	01	10	11	00	01	10	11
0001	A	A	E	F	A	0	–	1	1
0010	B	G	B	B	H	0	1	0	0
0100	C	C	C	G	I	0	1	0	1
1000	D	F	J	D	D	–	–	1	0
0011	E	–	B	–	–	–	1	–	–
1001	F	A	–	D	–	0	–	1	–
0110	G	C	–	B	–	0	–	0	–
1010	H	–	–	–	D	–	–	–	0
0101	I	–	–	–	A	–	–	–	1
1100	J	–	C	–	–	–	1	–	–

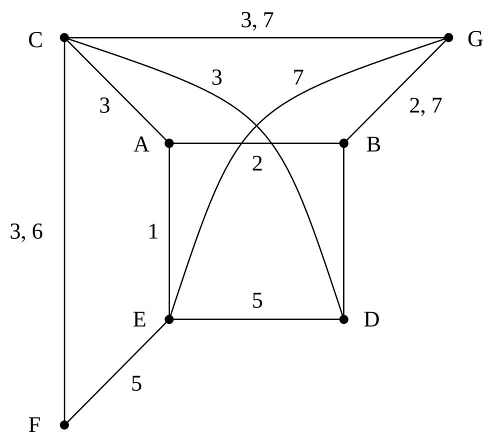
9.8. Using the merger diagram in Figure 9.40a, the FSM in Figure 9.39 becomes

Present state	Next state $w_2w_1 = 00$	01	10	11	Output z
A	$\textcircled{A}$	G	E	–	0
B	$\textcircled{B}$	C	$\textcircled{B}$	D	0
C	B	$\textcircled{C}$	E	$\textcircled{C}$	1
D	–	C	E	$\textcircled{D}$	0
E	A	–	$\textcircled{E}$	D	1
G	B	$\textcircled{G}$	–	D	1

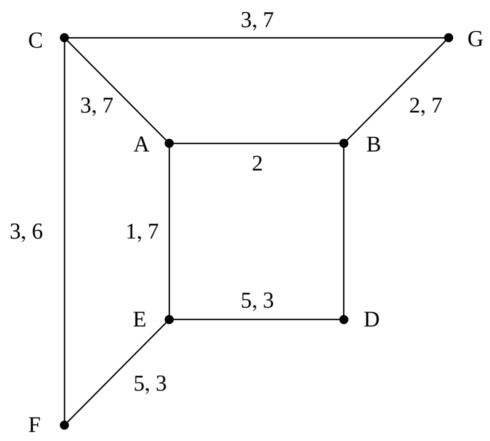
9.9. The relabeled flow table is

Present state	Next state $w_2w_1 = 00$	01	10	11	Output z
A	$\textcircled{1}$	2	3	–	
B	4	$\textcircled{2}$	–	7	
C	6	–	$\textcircled{3}$	7	
D	$\textcircled{4}$	5	3	–	
E	1	$\textcircled{5}$	–	7	
F	$\textcircled{6}$	5	3	–	
G	–	2	3	$\textcircled{7}$	

The corresponding transition diagram is as follows. Note that the diagonal transitions are shown only when they involve a transition to a stable state.



The diagonal transition $D \to C$ labeled 3 can be removed by using the unspecified entry in row E, such that the required transition is performed as $D \to E \to F \to C$; this involves placing a label 3 on the paths from D to E and E to F. Similarly, the diagonal transition $E \to G$ labeled 7 can be removed by using the unspecified entry in row A, such that the required transition is performed as $E \to A \to C \to G$. These modifications produce the following transition diagram:



Then the modified flow table is

Present state	Next state $w_2w_1 = 00$	01	10	11	Output z
A	Ⓐ	B	C	C	0
B	D	Ⓑ	–	G	0
C	F	–	Ⓒ	G	0
D	Ⓒ	E	E	–	1
E	A	Ⓔ	F	A	0
F	Ⓕ	E	C	–	0
G	–	B	C	Ⓖ	1

Thus, a possible state assignment is: $A = 000$, $B = 001$, $C = 100$, $D = 011$, $E = 010$, $F = 110$, and $G = 101$. Then, the state-assigned table is

	Present state	Next state				Output
	state	$w_2w_1 = 00$	01	10	11	
	$y_3y_2y_1$	$Y_3Y_2Y_1$				z
A	000	000	001	100	100	0
B	001	011	001	–	101	0
C	100	110	–	100	101	0
D	011	011	010	010	–	1
E	010	000	010	110	000	0
F	110	110	010	100	–	0
G	101	–	001	100	101	1

The next-state and output expressions are

$$\begin{aligned}
Y_3 &= w_2\overline{y}_2 + \overline{w}_1 y_3 + w_2\overline{w}_1\overline{y}_1 \\
Y_2 &= \overline{w}_1\overline{y}_3 y_1 + \overline{w}_2 y_3\overline{y}_1 + \overline{w}_2 w_1 y_2 + w_2\overline{w}_1\overline{y}_3 y_2 + y_2 y_1 \\
Y_1 &= w_1 y_3\overline{y}_2 + \overline{w}_2 w_1\overline{y}_2 + \overline{w}_2\overline{w}_1 y_1 + \overline{y}_3 y_2 y_1 \\
z &= y_2 y_1 + y_3 y_1
\end{aligned}$$

9.10. The minimum-cost hazard-free implementation is

$$f = \overline{x}_1\overline{x}_3\overline{x}_4 + x_1x_2x_4 + x_1x_3x_4$$

9.11. The minimum-cost hazard-free implementation is

$$f = \overline{x}_1\overline{x}_2\overline{x}_4\overline{x}_5 + \overline{x}_1\overline{x}_2x_3\overline{x}_4 + x_1x_2\overline{x}_3\overline{x}_4 + x_1x_2\overline{x}_4x_5$$

9.12. The minimum-cost hazard-free POS implementation is

$$f = (x_1 + x_2 + x_4)(x_1 + x_2 + \overline{x}_3)(x_1 + \overline{x}_3 + \overline{x}_4)(x_2 + \overline{x}_3 + x_4)$$

9.13. The minimum-cost hazard-free POS implementation is

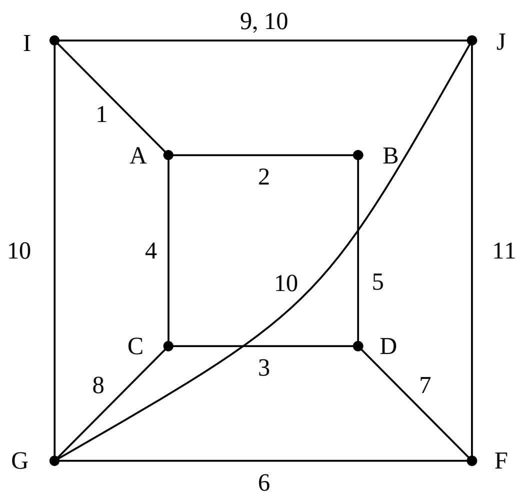
$$f = (x_1 + x_2 + \overline{x}_4 + x_5)(x_1 + x_2 + \overline{x}_3 + \overline{x}_4)(\overline{x}_1 + \overline{x}_2 + x_4)$$

9.14. If $A = B = D = E = 1$ and C changes from 0 to 1, then f changes $0 \rightarrow 1 \rightarrow 0$ and g changes $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$. Therefore, there is a static hazard on f and a dynamic hazard on g.

9.16. The flow diagram in Figure P9.3 meets the vending machine specification if $w_2 = D$ and $w_1 = N$. Therefore, the reduced flow table is the same as the same as the answer to Problem 9.3. The relabeled flow table is

Present state	Next state				Output
	DN=00	01	10	11	z
A	①	2	4	–	0
B	5	②	–	–	0
C	8	③	④	–	0
D	⑤	3	7	–	0
F	11	⑥	⑦	–	0
G	⑧	6	10	–	0
I	1	⑨	⑩	–	1
J	⑪	9	10	–	0

The transition diagram is



A suitable state assignment is: $A = 000$, $B = 001$, $C = 010$, $D = 011$, $F = 111$, $G = 110$, $I = 100$, and $J = 101$. Then the state-assigned table is

	Present state	Next state				Output
		DN=00	01	10	11	
	$y_3y_2y_1$	$Y_3Y_2Y_1$				z
A	000	$\textcircled{000}$	001	010	–	0
B	001	$\textcircled{011}$	$\textcircled{001}$	–	–	0
C	010	110	$\textcircled{010}$	$\textcircled{010}$	–	0
D	011	$\textcircled{011}$	010	111	–	0
F	111	101	$\textcircled{111}$	$\textcircled{111}$	–	0
G	110	$\textcircled{110}$	111	100	–	0
I	100	000	$\textcircled{100}$	$\textcircled{100}$	–	1
J	101	$\textcircled{101}$	100	100	–	0

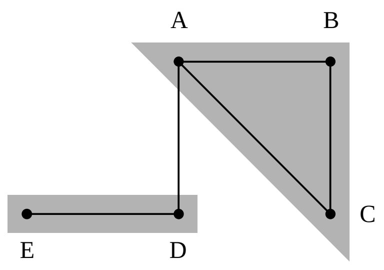
The next-state and output expressions are

$$\begin{aligned}
Y_3 &= Dy_3 + Ny_3 + y_3y_1 + y_3y_2 + Dy_1 + \overline{D}y_2\overline{y}_1\overline{N} \\
Y_2 &= D\overline{y}_3 + \overline{y}_3y_2 + \overline{N}\overline{y}_3y_1 + Ny_2 + Dy_2y_1 + \overline{D}y_2\overline{y}_1 \\
Y_1 &= \overline{N}\overline{y}_3y_1 + \overline{D}y_1\overline{N} + N\overline{y}_3\overline{y}_2 + Ny_3y_2 + \overline{y}_3\overline{y}_2y_1 + y_3y_2y_1 \\
z &= y_3\overline{y}_2\overline{y}_1
\end{aligned}$$

9.17. A possible Moore-type flow table is

Present state	Next state				Output
	$wc = 00$	01	10	11	z
A	$\textcircled{A}$	B	D	–	0
B	A	$\textcircled{B}$	–	C	0
C	–	B	D	$\textcircled{C}$	0
D	A	–	$\textcircled{D}$	E	0
E	A	$\textcircled{E}$	D	$\textcircled{E}$	1

A merger diagram for this flow table is



Merging states A, B and C into a new state A, and states D and E into a new state E, gives the Mealy-type flow table

Present state	Next state				Output z			
	$wc = 00$	01	10	11	00	01	10	11
A	Ⓐ	Ⓐ	D	Ⓐ	0	0	0	0
D	A	Ⓓ	Ⓓ	Ⓓ	0	1	0	1

Then, the excitation table is

Present state y	Next state				Output			
	$wc = 00$	01	10	11	00	01	10	11
	Y				z			
0	⓪	⓪	1	⓪	0	0	0	0
1	0	①	①	①	0	1	0	1

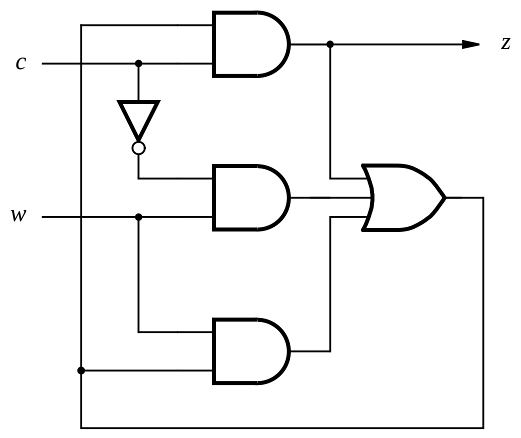
The next-state expression is

$$Y = w\bar{c} + cy + wy$$

Note that the term wy is included to prevent a static hazard. The output expression is

$$z = cy$$

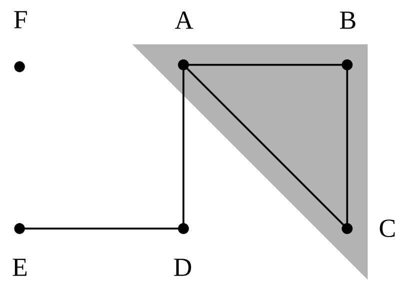
The resulting circuit is



9.18. A possible Moore-type flow table is

Present state	Next state				Output z
	$wc = 00$	01	10	11	
A	$\textcircled{A}$	B	D	–	0
B	A	$\textcircled{B}$	–	C	0
C	–	B	D	$\textcircled{C}$	0
D	A	–	$\textcircled{D}$	E	0
E	A	$\textcircled{E}$	F	$\textcircled{E}$	1
F	A	B	$\textcircled{F}$	$\textcircled{F}$	0

The corresponding merger diagram is



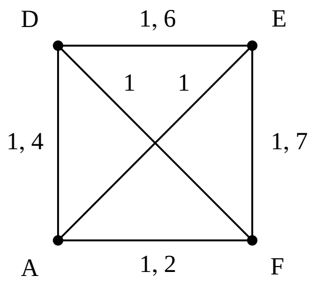
Merging rows A, B, and C into a new row A gives the reduced flow table

Present state	Next state				Output z
	$wc = 00$	01	10	11	z
A	$\textcircled{A}$	$\textcircled{A}$	D	$\textcircled{A}$	0
D	A	–	$\textcircled{D}$	E	0
E	A	$\textcircled{E}$	F	$\textcircled{E}$	1
F	A	A	$\textcircled{F}$	$\textcircled{F}$	0

To determine a suitable state assignment, relabel the flow table as follows:

Present state	Next state				Output z
	$wc = 00$	01	10	11	
A	$\textcircled{1}$	$\textcircled{2}$	4	$\textcircled{3}$	0
D	1	–	$\textcircled{4}$	6	0
E	1	$\textcircled{5}$	7	$\textcircled{6}$	1
F	1	2	$\textcircled{7}$	$\textcircled{8}$	0

The transition diagram is



The flow table is
Both diagonal transitions, under the label 1, can be omitted because there exist alternate paths along the edges for this label. Let the transition from E to A take place via D. Then, a possible state assignment is $A = 00$, $D = 01$, $E = 11$, and $F = 10$, which leads to the excitation table:

	Present	Next state				Output
	state	$wc = 00$	01	10	11	
	y_2y_1	Y_2Y_1	Y_2Y_1	Y_2Y_1	Y_2Y_1	z
A	00	$\textcircled{00}$	$\textcircled{00}$	01	$\textcircled{00}$	0
D	01	00	–	$\textcircled{01}$	11	0
E	11	01	$\textcircled{11}$	10	$\textcircled{11}$	1
F	10	00	00	$\textcircled{10}$	$\textcircled{10}$	0

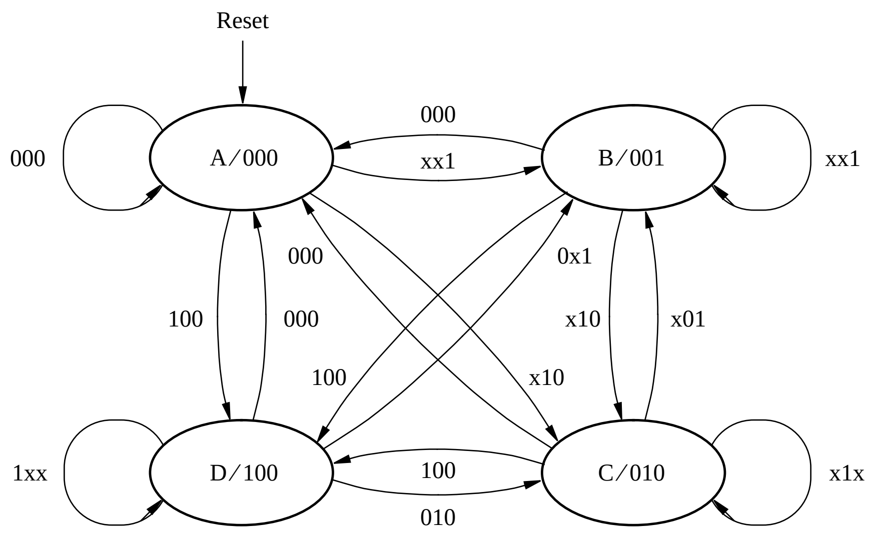
The resulting next-state expressions are

$$\begin{aligned} Y_2 &= wy_2 + cy_1 \\ Y_1 &= cy_1 + \overline{w}y_1y_2 + w\overline{y}_2\overline{c} + wy_1\overline{y}_2 \end{aligned}$$

The product term $wy_1\overline{y}_2$ is included to avoid a static hazard.

The output expression is $z = y_1y_2$.

9.19. A possible state diagram for the three-input arbiter is



9.20. Using the mutual exclusion element, the input valuation $r_2r_1 = 11$ cannot occur. Hence, the flow table is

Present state	Next state				Output
	$r_2r_1 = 00$	01	10	11	g_2g_1
A	$\textcircled{A}$	B	C	–	00
B	A	$\textcircled{B}$	C	–	01
C	A	B	$\textcircled{C}$	–	10

The excitation table is

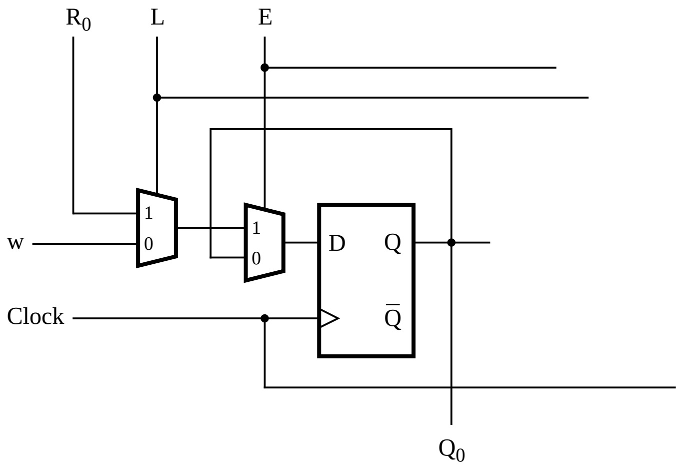
	Present state	Next state				Output
		$r_2r_1 = 00$	01	10	11	
	y_2y_1	Y_2Y_1				g_2g_1
A	00	$\textcircled{00}$	01	10	–	00
B	01	00	$\textcircled{01}$	10	–	01
C	10	00	01	$\textcircled{10}$	–	10
D	11	–	–	–	–	–

The resulting next state and output equations are

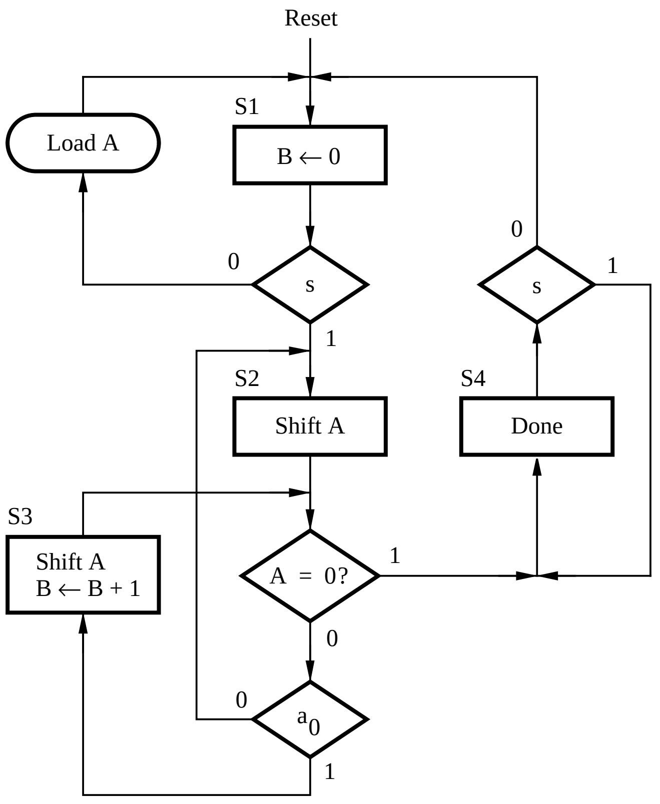
$$\begin{aligned} Y_1 &= r_1 \\ Y_2 &= r_2 \\ g_1 &= y_1 \\ g_2 &= y_2 \end{aligned}$$

Chapter 10

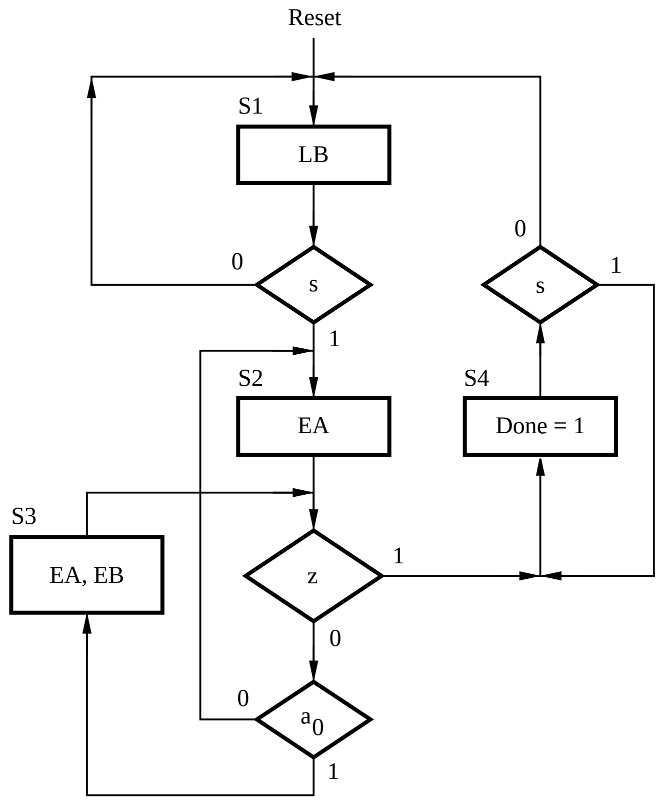
10.1. In the modified shift register the order of the multiplexers that perform the load and enable operations are reversed from the order in Figure 10.4. Bit zero of the modified register is show below.



10.2. (a) A modified ASM chart that has only Moore-type outputs in state S2 is given below.



(*b*)

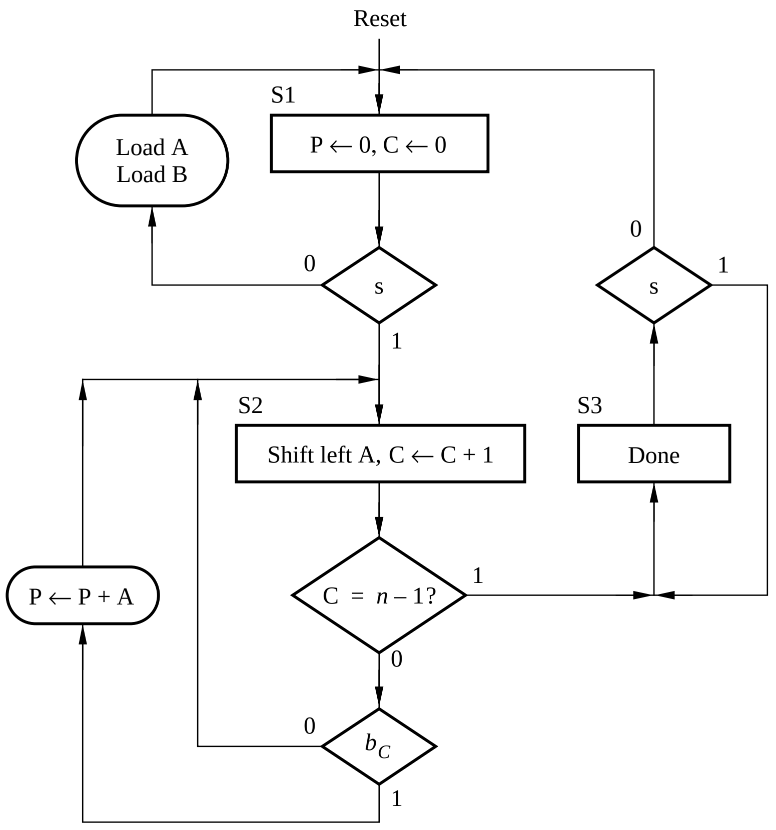


(*c*)

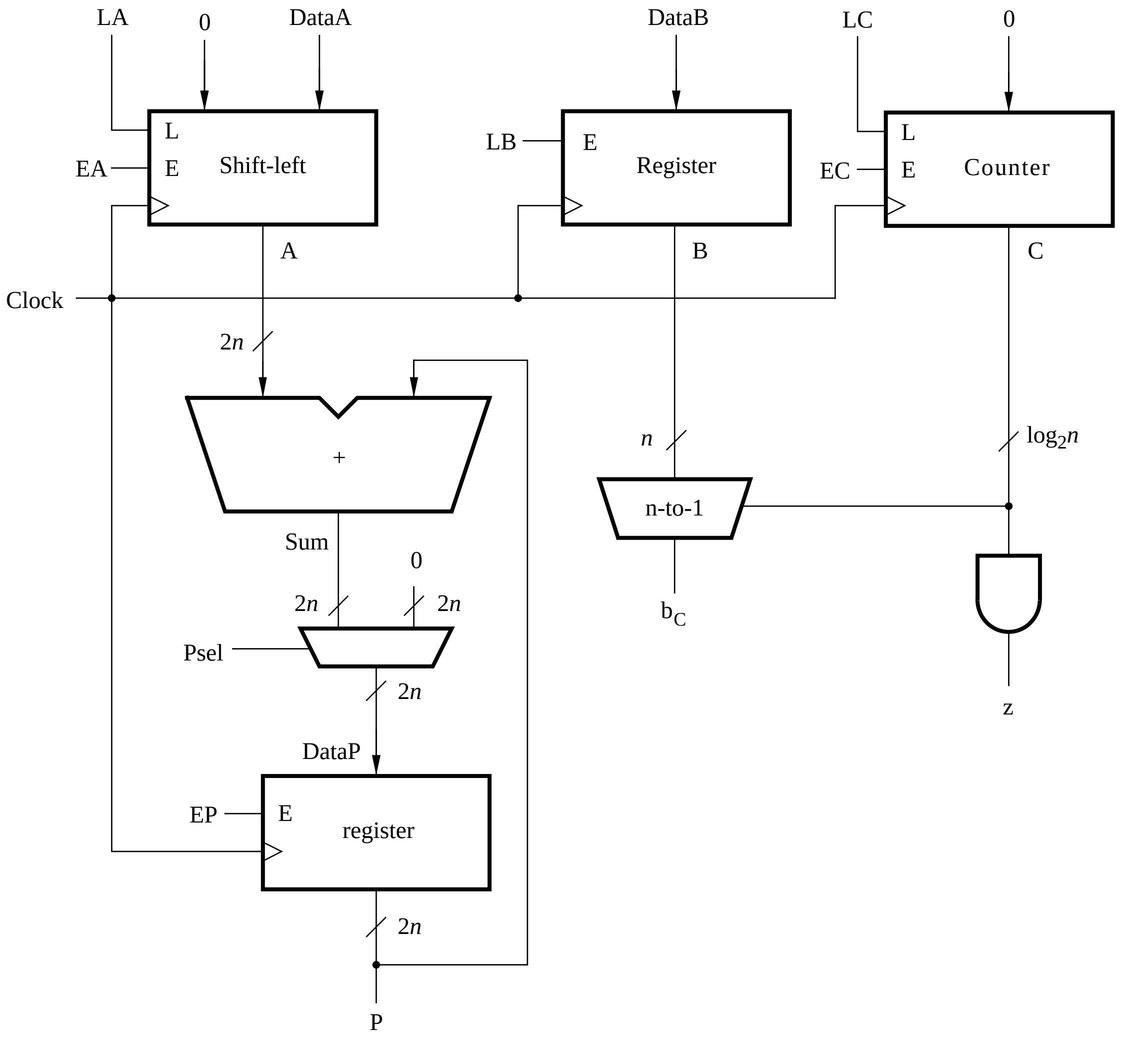
```
module  bitcount (Clock, Resetn, LA, s, Data, B, Done);
    input  Clock, Resetn, LA, s;
    input  [7:0] Data;
    output  [3:0] B;
    output  Done;
    wire  [7:0] A;
    wire  z;
    reg  [1:0] Y, y;
    reg  [3:0] B;
    reg  Done, EA, EB, LB;

// control circuit
    parameter  S1 = 2'b00, S2 = 2'b01, S3 = 2'b10, S4 = 2'b11;

    always @(s or y or z)
    begin: State_table
        case (y)
            S1:      if (s == 0)  Y = S1;
                     else  Y = S2;
            S2,S3:   if (!z && !A[0])  Y = S2;
                     else if (!z && A[0])  Y = S3;
                     else  Y = S4;
            S4:      if (s == 1)  Y = S4;
                     else  Y = S1;
        endcase
    end

    always @(posedge Clock or negedge Resetn)
    begin: State_flipflops
        if (Resetn == 0)  y <= S1;
        else  y <= Y;
    end

    always @(y or A[0])
    begin: FSM_outputs
        EA = 0; LB = 0; EB = 0; Done = 0;  // defaults
        case (y)
            S1:      LB = 1;
            S2:      EA = 1;
            S3:      begin
                         EA = 1; EB = 1;
                     end
            S4:      Done = 1;
        endcase
    end
```

```
// datapath circuit

    // counter B
    always @(negedge Resetn or posedge Clock)
        if (!Resetn)  B <= 0;
        else if (LB)  B <= 0;
        else if (EB)  B <= B + 1;

    shiftrne ShiftA (Data, LA, EA, 0, Clock, A);
    assign z = ∼|A;

endmodule
```

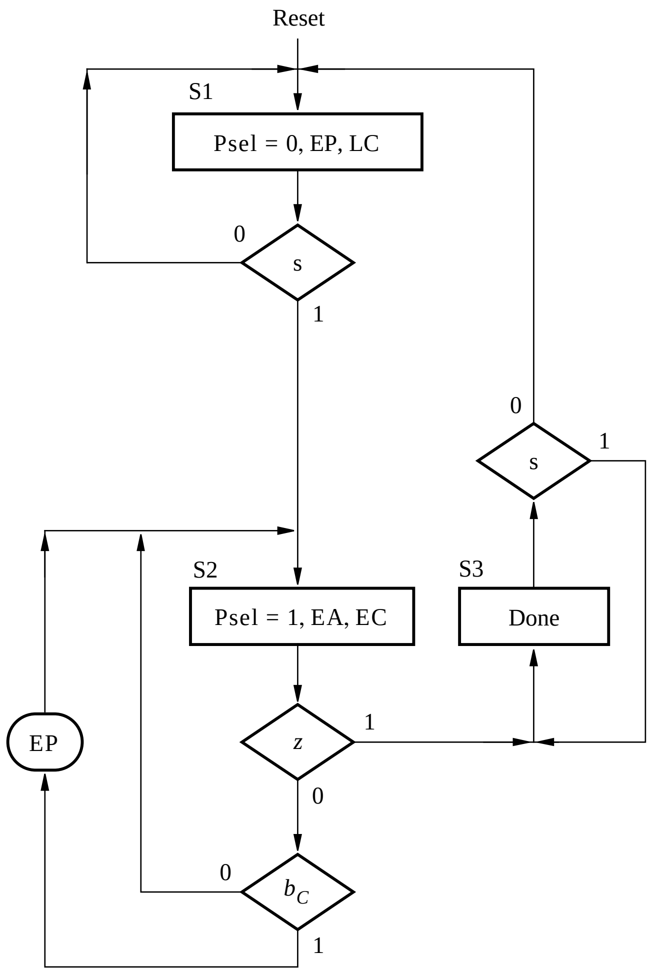
10.3. (*a*)



(*b*)



(*c*) The ASM chart for the control circuit is shown below. Note that we assume the EB signal is controlled by external logic.



(*d*)

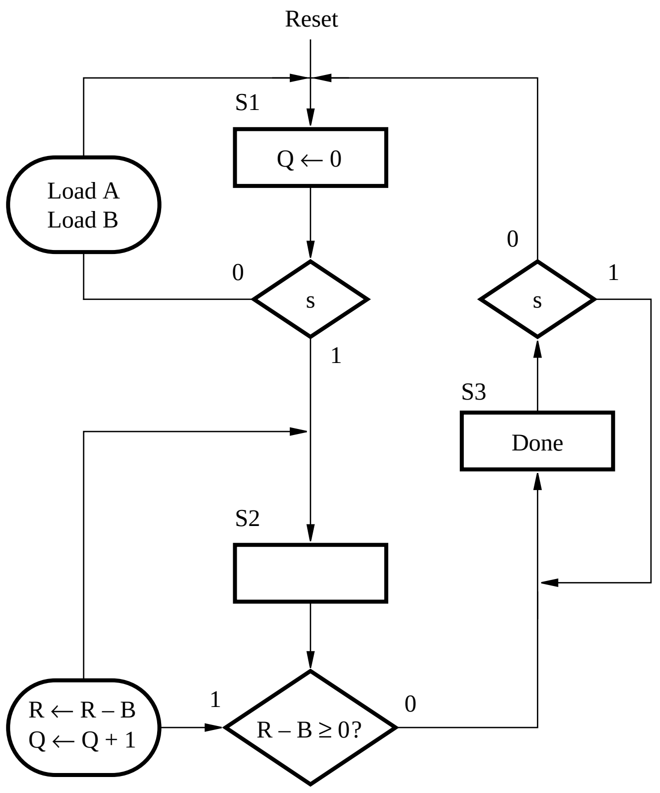
```
module multiply (Clock, Resetn, LA, LB, s, DataA, DataB, P, Done);
   parameter n = 8;
   parameter m = 3;
   input Clock, Resetn, LA, LB, s;
   input [n−1:0] DataA, DataB;
   output [n+n−1:0] P;
   output Done;
   wire bc, z;
   reg [n+n−1:0] DataP;
   wire [n+n−1:0] A, Sum;
   reg [1:0] y, Y;
   wire [n−1:0] B;
   wire [m−1:0] C;
   reg Done, EA, EP, Psel, LC, EC;
   integer k;
```

```
// control circuit
    parameter  S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;

    always @(s or y or z)
    begin: State_table
        case (y)
            S1:    if (s == 0)  Y = S1;
                   else  Y = S2;
            S2:    if (z)  Y = S3;
                   else  Y = S2;
            S3:    if (s == 1)  Y = S3;
                   else  Y = S1;
            default: Y = 2'bxx;
        endcase
    end

    always @(posedge Clock or negedge Resetn)
    begin: State_flipflops
        if (Resetn == 0)  y <= S1;
        else  y <= Y;
    end

    always @(y or bc)
    begin: FSM_outputs
        EA = 0; EP = 0; Done = 0; Psel = 0; EC = 0; LC = 0;  // defaults
        case (y)
            S1:    begin
                       EP = 1; EC = 1; LC = 1;
                   end
            S2:    begin
                       EA = 1; Psel = 1; EC = 1; LC = 0;
                       if (bc)  EP = 1;
                       else  EP = 0;
                   end
            S3:    Done = 1;
        endcase
    end

// datapath circuit
    regne  RegB  (DataB, Clock, Resetn, LB, B);
        defparam  RegB.n = 8;
    shiftlne  ShiftA  ({{n{1'b0}}, DataA}, LA, EA, Clock, A);
        defparam  ShiftA.n = 16;
    upcount  Counter  (LC, Clock, EC, C);
        defparam  Counter.n = m;
    assign  bc = B[C];
    assign  z = &C;
    assign  Sum = A + P;

    // define the 2n 2-to-1 multiplexers
    always @(Psel or Sum)
        for (k = 0; k < n+n; k = k+1)
            DataP[k] = Psel ? Sum[k] : 0;

    regne  RegP  (DataP, Clock, Resetn, EP, P);
        defparam  RegP.n = 16;

endmodule
```

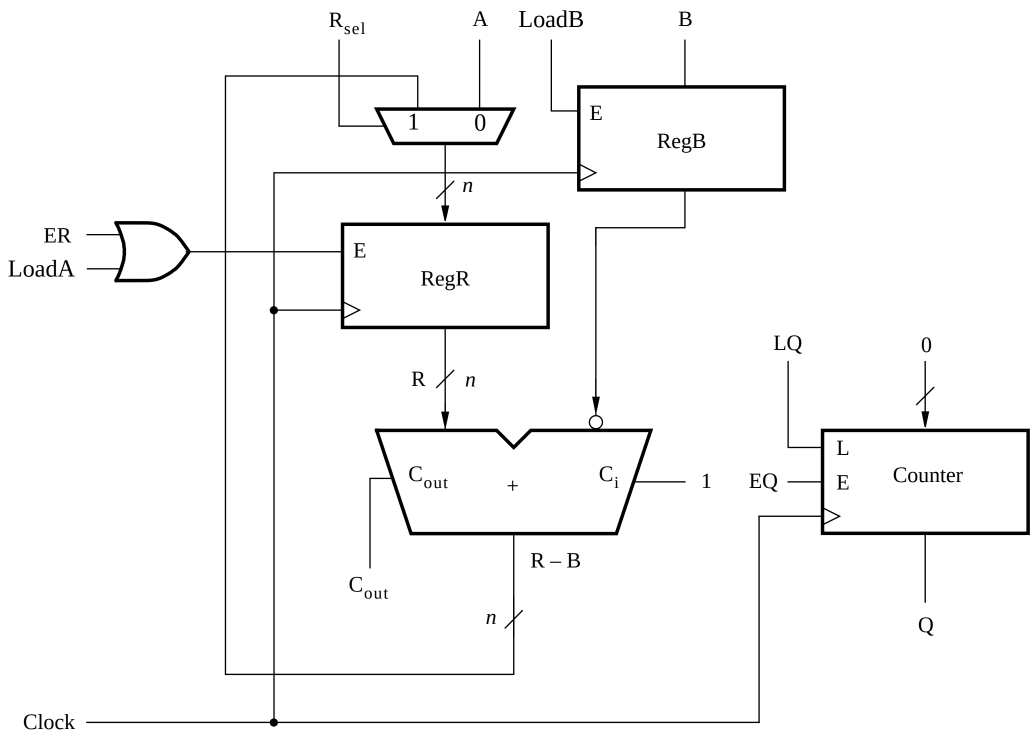
10.4.

```
module divider (Clock, Resetn, s, LA, EB, DataA, DataB, R, Q, Done);
    parameter  n = 8, logn = 3;
    input  Clock, Resetn, s, LA, EB;
    input  [n−1:0] DataA, DataB;
    output  [n−1:0] R, Q;
    output  Done;
    wire  Cout, z;
    wire  [n−1:0] DataR;
    wire  [n−1:0] Sum;
    reg  [1:0] y, Y;
    wire  [n−1:0] A, B, Q;
    wire  [logn−1:0] Count;
    reg  Done, EA, Rsel, LR, ER, LC, EC, EQ;

// control circuit
    parameter  S1 = 2'b00, S2 = 2'b01, S3 = 2'b10, S4 = 2'b11;

    always @(s or y or z)
    begin: State_table
        case (y)
            S1:    if (s == 0)  Y = S1;
                   else  Y = S2;
            S2:    Y = S3;
            S3:    if (z == 1)  Y = S4;
                   else  Y = S2;
            S4:    if (s == 1)  Y = S4;
                   else  Y = S1;
        endcase
    end
    always @(posedge Clock or negedge Resetn)
    begin: State_flipflops
        if (Resetn == 0)  y <= S1;
        else  y <= Y;
    end

    always @(y or s or Cout or z)
    begin: FSM_outputs
        LR = 0; ER = 0; LC = 0; EC = 0; EA = 0;  // defaults
        EQ = 0; Rsel = 0; Done = 0;  // defaults
        case (y)
            S1:    begin
                        LC = 1; LR = 1; Rsel = 0;
                   end
            S2:    begin
                        ER = 1; EA = 1;
                   end
            S3:    begin
                        Rsel = 1; EQ = 1; EC = 1;
                        if (Cout)  LR = 1;
                        else  LR = 0;
                        if (z == 0) EC = 1;
                        else  EC = 0;
                   end
            S4:    Done = 1;
        endcase
    end

// datapath circuit
    regne  RegB  (DataB, Clock, Resetn, EB, B);
        defparam  RegB.n = n;

    shiftlne  ShiftR  (DataR, LR, ER, A[n−1], Clock, R);
        defparam  ShiftR.n = n;

    shiftlne  ShiftA  (DataA, LA, EA, 0, Clock, A);
        defparam  ShiftA.n = n;

    shiftlne  ShiftQ  (0, 0, EQ, Cout, Clock, Q);
        defparam  ShiftQ.n = n;

    downcount  Counter  (Clock, EC, LC, Count);
        defparam  Counter.n = logn;

    assign  z = (Count == 0);
    assign  {Cout, Sum} = R + {0, ~B} + 1;

    // define the n 2-to-1 multiplexers
    assign  DataR = Rsel ? Sum : 0;

endmodule
```

10.5. (*a*)

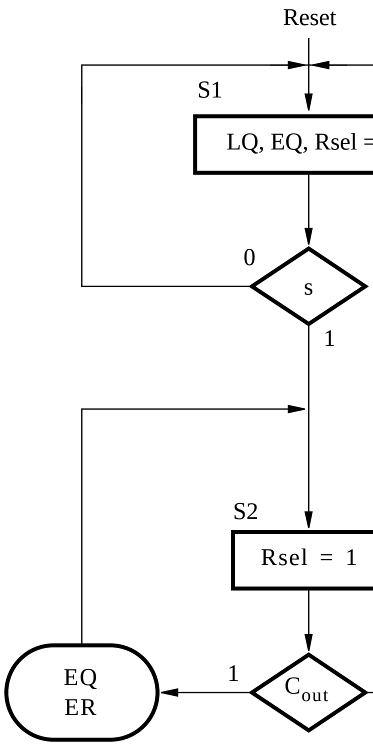


(*b*)



R_{sel}
A
LoadB
B
1
0
E
RegB
n
ER
LoadA
E
RegR
R
n
LQ
0
C_{out}
+
C_i
1
EQ
L
E
Counter
R – B
C_{out}
n
Q
Clock

(*c*)



Reset
S1
LQ, EQ, Rsel = 0
0
s
1
0
s
1
S3
Done
S2
Rsel = 1
EQ
ER
1
C_{out}
0

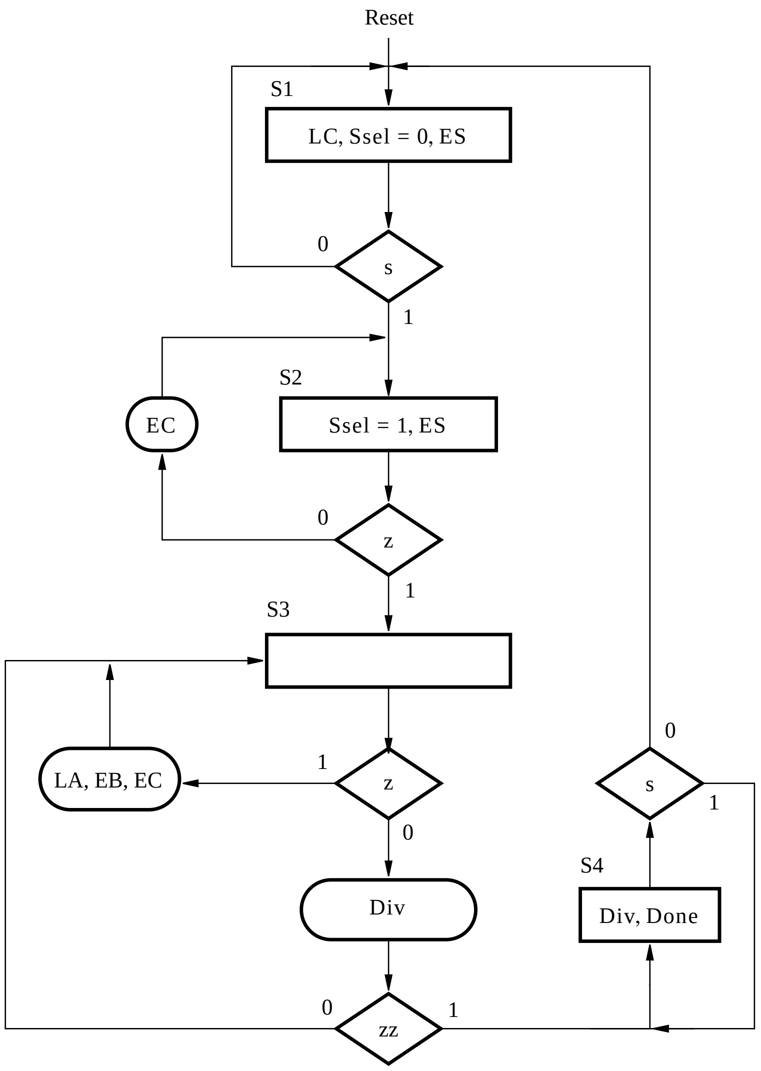
(*d*)

```
module divider (Clock, Resetn, s, EA, EB, DataA, DataB, R, Q, Done);
    parameter n = 8;
    input Clock, Resetn, s, EA, EB;
    input [n−1:0] DataA, DataB;
    output [n−1:0] R, Q;
    output Done;
    wire Cout, ERegR;
    wire [n−1:0] DataR;
    wire [n−1:0] Sum;
    reg [1:0] y, Y;
    wire [n−1:0] A, B, Q;
    reg Done, Rsel, ER, LQ, EQ;

// control circuit
    parameter S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;

    always @(s or y or Cout)
    begin: State_table
        case (y)
            S1:   if (s == 0) Y = S1;
                  else Y = S2;
            S2:   if (Cout == 0) Y = S3;
                  else Y = S2;
            S3:   if (s == 1) Y = S3;
                  else Y = S1;
            default: Y = S1;
        endcase
    end

    always @(posedge Clock or negedge Resetn)
    begin: State_flipflops
        if (Resetn == 0) y <= S1;
        else y <= Y;
    end

    always @(y or s or Cout)
    begin: FSM_outputs
        ER = 0; LQ = 0; EQ = 0; Rsel = 0; Done = 0; // defaults
        case (y)
            S1:   begin
                      LQ = 1; EQ = 1; Rsel = 0;
                  end
            S2:   begin
                      Rsel = 1;
                      if (Cout)
                      begin
                          EQ = 1; ER = 1;
                      end
```

```
                else
                begin
                    EQ = 0; ER = 0;
                end
            end
        S3:  Done = 1;
    endcase
  end
// datapath circuit
  regne  RegB  (DataB, Clock, Resetn, EB, B);
    defparam  RegB.n = n;

  regne  RegR  (DataR, Clock, Resetn, ERegR, R);
    defparam  RegR.n = n;

  upcount  Counter  (Clock, EQ, LQ, Q);
    defparam  Counter.n = n;

  assign  ERegR = ER | EA;
  assign  {Cout, Sum} = {1'b0, R} + {1'b0, ~B} + 1;

  // define the n 2-to-1 multiplexers
  assign  DataR = Rsel ? Sum : DataA;

endmodule
```

(e) This implementation of a divider is less efficient in the worst case when compared to the other implementations shown. The efficient algorithms presented are able to perform division in n cycles for n-bit inputs. However, the method of repeated subtraction takes 2^n cycles for the worst case, which is when dividing by 1. On the other hand, if the two numbers A and B are close in size, then repeated subtraction is an efficient approach.

10.6. State S3 is responsible for loading the operands into the divider, while state S4 starts the division operation. These states can be combined into a single state. We can use the z flag to indicate the first time that we've entered the new combined state. When $z = 1$ a mealy output is produced which loads the operands and decrements the counter. Thus, the z flag changes to a 0. The combined state now produces a mealy output which starts the division, on the condition that $z = 0$. This control circuit ASM chart is shown below.

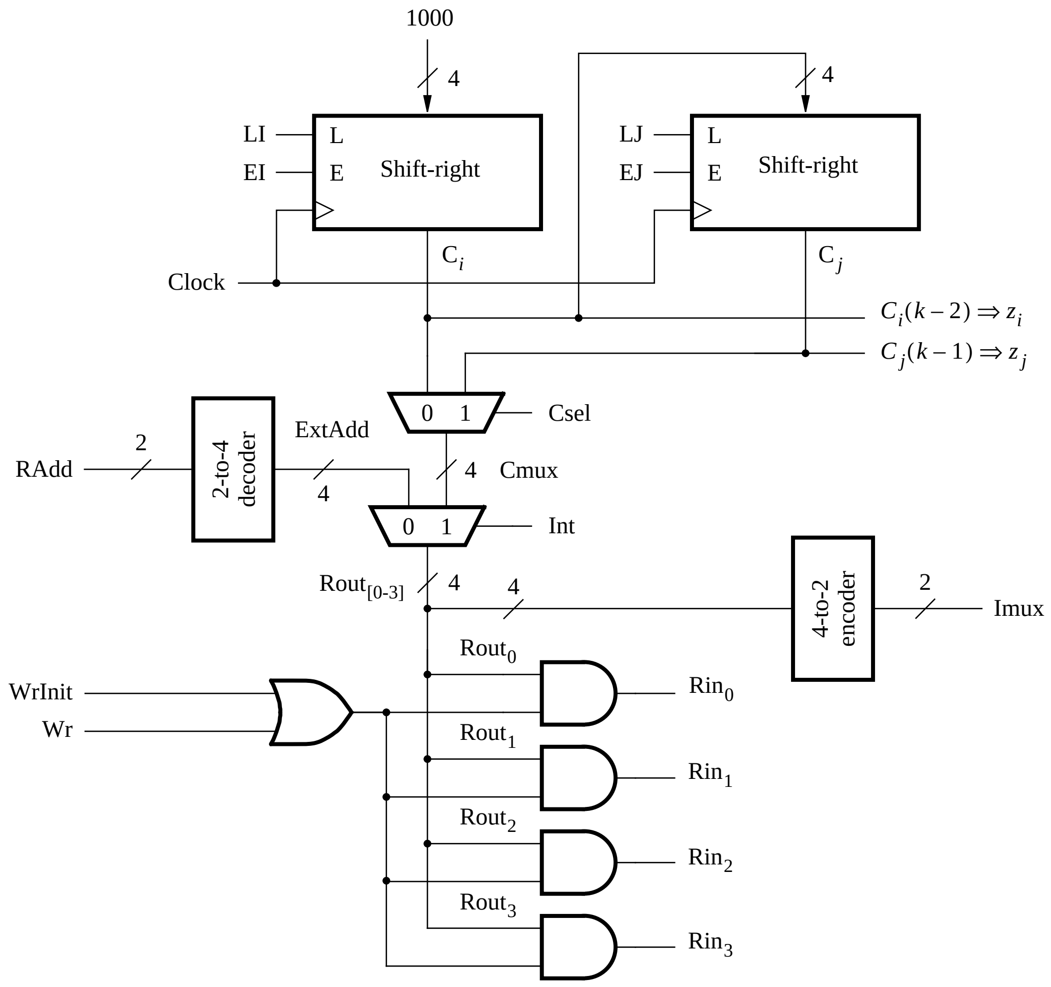


10.7.

```
module  meancntl (Clock, Resetn, s, z, zz, EC, LC, Ssel, ES, LA, EB, Div, Done);
    input  Clock, Resetn, s, z, zz;
    output  EC, LC, Ssel, ES, LA, EB, Div, Done;
    reg  EC, LC, Ssel, ES, LA, EB, Div, Done;
    reg  [1:0] y, Y;
    parameter  S1 = 2'b00, S2 = 2'b01, S3 = 2'b10, S4 = 2'b11;
    always @(s or y or z or zz)
    begin: State_table
        case (y)
            S1:  if (s == 0)  Y = S1;
                 else  Y = S2;
            S2:  if (z == 0)  Y = S2;
                 else  Y = S3;
        S3:  if (zz == 1)  Y = S3;
             else  Y = S4;
        S4:  if (s == 1)  Y = S4;
             else  Y = S1;
    endcase
end

always @(posedge Clock or negedge Resetn)
begin: State_flipflops
    if (Resetn == 0)  y <= S1;
    else  y <= Y;
end

always @(s or y or z or zz)
begin: FSM_outputs
    LC = 0; EC = 0; ES = 0; LA = 0; EB = 0; Div = 0; Done = 0; Ssel = 0;  // defaults
    case (y)
        S1:  begin
                 LC = 1; ES = 1;
             end
        S2:  begin
                 Ssel = 1; ES = 1;
                 if (z == 0)  EC = 1;
                 else  EC = 0;
             end
        S3:  if (z == 0)
             begin
                 Div = 1; LA = 0; EB = 0; EC = 0;
             end
             else
             begin
                 LA = 1; EB = 1; EC = 1;
             end
        S4:  begin
                 Div = 1; Done = 1;
             end
    endcase
end

endmodule
```

10.8. The states $S2$ and $S3$ can be merged into a single state by performing the assignment $C_j = C_i + 1$. The circuit would require an adder to increment C_i by 1 and the outputs of this adder would be loaded in parallel into the counter C_j. If instead of using counters to implement C_i and C_j we used shift registers, then the effect of producing $C_i + 1$ could be efficiently implemented by wiring C_i to the parallel-load data inputs on C_j such that the bits are shifted by one position.

10.9. (a) The part of the datapath circuit that needs to be modified is shown below. The rest of the datapath is the same as the circuit shown in Figure 10.37.



(*b*)

```
module  sort (Clock, Resetn, s, WrInit, Rd, DataIn, RAdd, DataOut, Done);
   parameter  n = 4;
   input  Clock, Resetn, s, WrInit, Rd;
   input  [n−1:0] DataIn;
   input  [1:0] RAdd;
   output  [n−1:0] DataOut;
   output  Done;
   wire  [0:3] Ci, Cj, Cmux;
   reg  [1:0] Imux;
   wire  [n−1:0] R0, R1, R2, R3, A, B;
   wire  [n−1:0] RData, ABMux;
   wire  BltA, zi, zj;
   reg  Int, Csel, Wr, Ain, Bin, Aout, Bout, LI, LJ, EI, EJ, Done;
   reg  [3:0] y, Y;
   reg  Rin0, Rin1, Rin2, Rin3;
   reg  [n−1:0] ABData;
   wire  [0:3] Rout, ExtAdd;
   wire  [0:3] Addr0; //Parallel load for Ci
```

```
// control circuit
    parameter  S1 = 4'b0000, S2 = 4'b0001, S3 = 4'b0010, S4 = 4'b0011;
    parameter  S5 = 4'b0100, S6 = 4'b0101, S7 = 4'b0110, S8 = 4'b0111, S9 = 4'b1000;

    always @(s or BltA or zj or zi or y)
    begin: State_table
        case (y)
            S1:  if (s == 0)  Y = S1;
                 else  Y = S2;
            S2:  Y = S3;
            S3:  Y = S4;
            S4:  Y = S5;
            S5:  if (BltA)  Y = S6;
                 else  Y = S8;
            S6:  Y = S7;
            S7:  Y = S8;
            S8:  if (!zj)  Y = S4;
                 else if (!zi)  Y = S2;
                 else  Y = S9;
            S9:  if (s)  Y = S9;
                 else  Y = S1;
            default: Y = 4'bx;
        endcase
    end

    always @(posedge Clock or negedge Resetn)
    begin: State_flipflops
        if (Resetn == 0)  y <= S1;
        else  y <= Y;
    end

    always @(y or zj or zi)
    begin: FSM_outputs
        Int = 1; Done = 0; LI = 0; LJ = 0; EI = 0; EJ = 0;  // defaults
        Csel = 0; Wr = 0; Ain = 0; Bin = 0; Aout = 0; Bout = 0;  // defaults
        case (y)
            S1:  begin
                     LI = 1; Int = 0;
                 end
            S2:begin
                     Ain = 1; LJ = 1;
                 end
            S3:  EJ = 1;
            S4:  begin
                     Bin = 1; Csel = 1;
                 end
            S5:  ; // no ouputs asserted in this state
            S6:  begin
                     Csel = 1; Wr = 1; Aout = 1;
                 end
        S7:  begin
                 Wr = 1; Bout = 1;
             end
        S8:  begin
                 Ain = 1;
                 if (!zj)  EJ = 1;
                 else
                 begin
                     EJ = 0;
                     if (!zi)  EI = 1;
                     else  EI = 0;
             end
        S9: Done = 1;
    endcase
end

// datapath circuit
regne  Reg0  (RData, Clock, Resetn, Rin0, R0);
    defparam  Reg0.n = n;
regne  Reg1  (RData, Clock, Resetn, Rin1, R1);
    defparam  Reg1.n = n;
regne  Reg2  (RData, Clock, Resetn, Rin2, R2);
    defparam  Reg2.n = n;
regne  Reg3  (RData, Clock, Resetn, Rin3, R3);
    defparam  Reg3.n = n;
regne  RegA  (ABData, Clock, Resetn, Ain, A);
    defparam  RegA.n = n;
regne  RegB  (ABData, Clock, Resetn, Bin, B);
    defparam  RegB.n = n;

assign  BltA = (B < A) ? 1 : 0;
assign  ABMux = (Bout == 0) ? A : B;
assign  RData = (WrInit == 0) ? ABMux : DataIn;
assign  Addr0 = 4'b1000;
shiftrne  Outerloop  (Addr0, LI, EI, 0, Clock, Ci);
shiftrne  Innerloop  (Ci, LJ, EJ, 0, Clock, Cj);
dec2to4  Decoder  (RAdd, ExtAdd);
assign  Rout = Int ? Cmux : ExtAdd;
assign  Cmux = (Csel == 0) ? Ci : Cj;

always @(WrInit or Wr or Rout or R3 or R2 or R1 or R0)
begin
    case (Rout)
        4'b1000: Imux = 0;
        4'b0100: Imux = 1;
        4'b0010: Imux = 2;
        4'b0001: Imux = 3;
        default: Imux = 0;
    endcase
```

```
        if (WrInit || Wr)
            case (Rout)
                4'b1000: {Rin3, Rin2, Rin1, Rin0} = 4'b0001;
                4'b0100: {Rin3, Rin2, Rin1, Rin0} = 4'b0010;
                4'b0010: {Rin3, Rin2, Rin1, Rin0} = 4'b0100;
                4'b0001: {Rin3, Rin2, Rin1, Rin0} = 4'b1000;
                default: {Rin3, Rin2, Rin1, Rin0} = 4'bx;
            endcase
        else {Rin3, Rin2, Rin1, Rin0} = 4'b0000;

        case (Imux)
            0: ABData = R0;
            1: ABData = R1;
            2: ABData = R2;
            3: ABData = R3;
        endcase
    end

    assign  zi = Ci[2];
    assign  zj = Cj[3];
    assign  DataOut = (Rd == 0) ? 'bz : ABData;
endmodule
```

(c) The major drawback of using shift-registers instead of counters is that the number of flip-flops is increased. Each counter uses $log_2 n$ flip-flops while each shift register contains n flip-flops. However, the shift-register requires no combinational logic to perform tests such as whether the count value $k-2$ has been reached — in the shift register we directly access bit $k-2$ of the register to perform this test. It should also be possible to clock the datapath at a higher maximum clock frequency when using shift-registers, because they are simpler than counters.

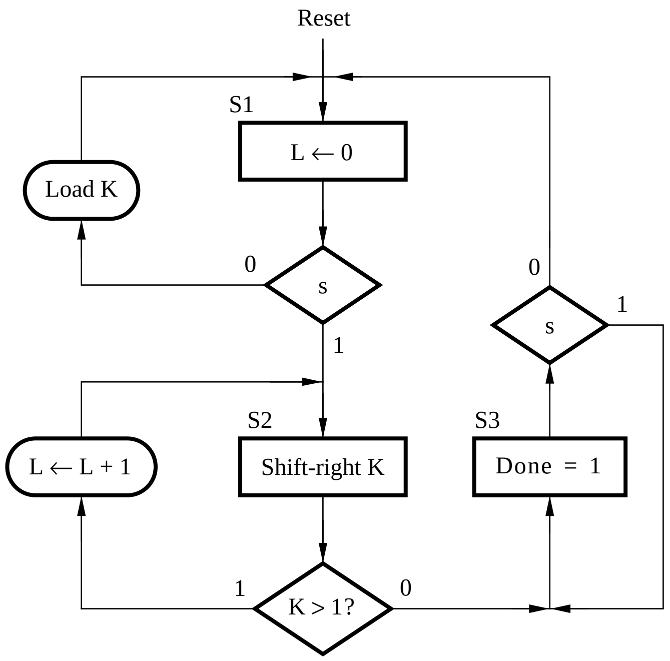
10.11. The Verilog code below shows the changes needed in the datapath so that an SRAM block can be used instead of registers. The SRAM block is clocked on the negative edge of the Clock signal, hence changes in the outputs produced by the other datapath elements must be stable before the negative edge; the clock period must be long enough to accommodate this constraint.

```
module sort (Clock, Resetn, s, WrInit, Rd, DataIn, RAdd, DataOut, Done);
    parameter n = 4;
    input Clock, Resetn, s, WrInit, Rd;
    input [n−1:0] DataIn;
    input [1:0] RAdd;
    output [n−1:0] DataOut;
    output Done;
    wire [1:0] Ci, Cj, CMux, IMux, MemAdd;
    wire [n−1:0] A, B, RData, ABMux;
    wire BltA, zi, zj, WE, NClock;
    reg Int, Csel, Wr, Ain, Bin, Aout, Bout;
    reg LI, LJ, EI, EJ, Done;
    reg [3:0] y, Y;
    reg [n−1:0] ABData;

// control circuit
    parameter S1 = 4'b0000, S2 = 4'b0001, S3 = 4'b0010, S4 = 4'b0011;
    parameter S5 = 4'b0100, S6 = 4'b0101, S7 = 4'b0110, S8 = 4'b0111, S9 = 4'b1000;

    always @(s or BltA or zj or zi)
    begin: State_table
        . . . code not shown: see Figure 10.40
    end

    always @(posedge Clock or negedge Resetn)
    begin: State_flipflops
        if (Resetn == 0)
            y <= S1;
        else
            y <= Y;
    end

    always @(y or zj or zi)
    begin: FSM_outputs
        . . . code not shown: see Figure 10.40
    end

    regne RegA (ABData, Clock, Resetn, Ain, A);
        defparam RegA.n = n;
    regne RegB (ABData, Clock, Resetn, Bin, B);
        defparam RegB.n = n;

    assign BltA = (B < A) ? 1 : 0;
    assign ABMux = (Bout == 0) ? A : B;
    assign RData = (WrInit == 0) ? ABMux : DataIn;

    upcount OuterLoop (0, Resetn, Clock, EI, LI, Ci);
    upcount InnerLoop (Ci, Resetn, Clock, EJ, LJ, Cj);

    assign CMux = (Csel == 0) ? Ci : Cj;
    assign IMux = (Int == 1) ? CMux : RAdd;

    assign MemAdd = IMux;
    assign WE = WrInit | Wr;
    assign NClock = ∼Clock;
    lpm_ram_dq mem_block (.address(MemAdd), .we(WE), .q(ABData),
        .inclock(NClock), .data(RData));
        defparam mem_block.lpm_width = 4;
        defparam mem_block.lpm_widthad = 2;
        defparam mem_block.lpm_address_control = "registered";
        defparam mem_block.lpm_indata = "registered";
        defparam mem_block.lpm_outdata = "unregistered";

    assign zi = (Ci == 2);
    assign zj = (Cj == 3);
    assign DataOut = (Rd == 0) ? 'bz : ABData;

endmodule
```

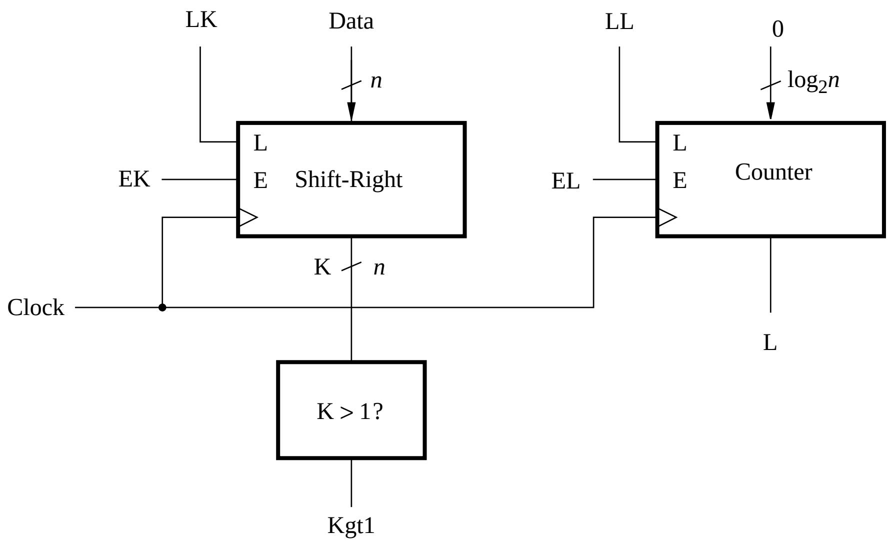
10.12. Pseudo-code that represents the log_2 operation is

```
-- assume that $K \geq 1$
$L = 0$ ;
while $(K > 1)$ do
    $K = K \div 2$ ;
    $L = L + 1$ ;
end while ;
-- $L$ now has the largest value such that $2^L < K$
```

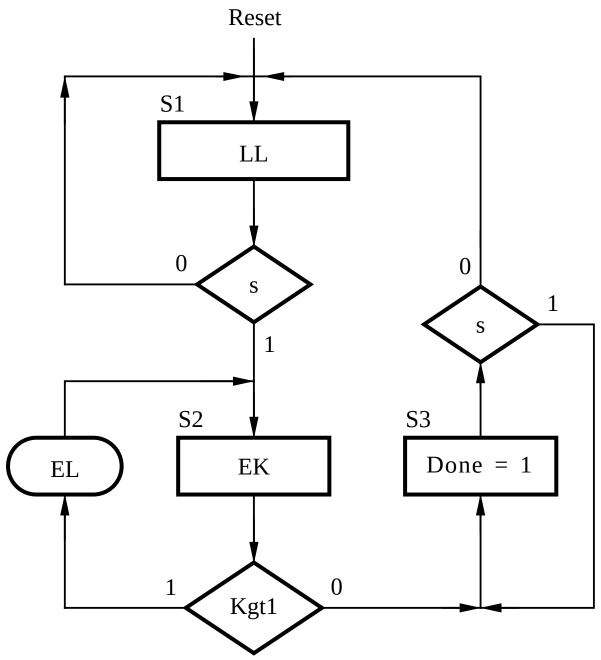
An ASM chart that corresponds to the pseudo-code is



From the ASM chart, a shift-register is needed to divide K by 2, and a counter is needed for L. An appropriate datapath circuit is



An ASM chart for the control circuit is



Complete Verilog code for this circuit is shown below.

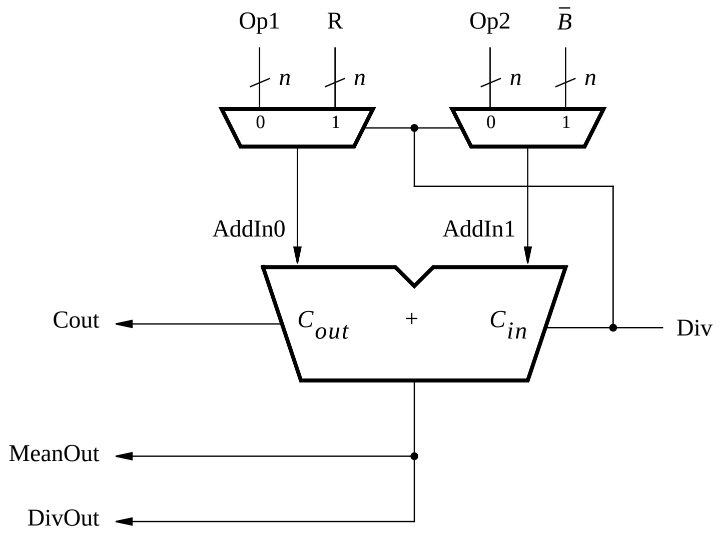
```
module log2k (Clock, Resetn, LData, s, Data, L, Done);
    input Clock, Resetn, LData, s;
    input [7:0] Data;
    output [3:0] L;
    output Done;

    wire [7:0] K;
    wire Kgt1;
    reg [1:0] y, Y;
    reg Done, EL, LL, EK;

// control circuit

    parameter S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;

    always @(s or y or Kgt1)
    begin: FSM_transitions
        case (y)
            S1:    if (s == 0) Y = S1;
                   else Y = S2;
            S2:    if (Kgt1 == 1) Y = S2;
                   else Y = S3;
            S3:    if (s == 1) Y = S3;
                   else Y = S1;
            default: Y = 2'bxx;
        endcase
    end

    always @(posedge Clock or negedge Resetn)
    begin: State_flipflops
        if (Resetn == 0)
            y <= S1;
        else
            y <= Y;
    end

    always @(y or s or LData or Kgt1)
    begin: FSM_outputs
        EL = 0; LL = 0; EK = 0; Done = 0; // defaults
        case (y)
            S1:    begin
                       EL = 1; LL = 1;
                       if (LData == 1) EK = 1;
                       else EK = 0;
                   end
            S2:    begin
                       EK = 1;
                       if (Kgt1) EL = 1;
                       else EL = 0;
                   end
            S3:    Done = 1;
        endcase
    end
```

```
//datapath circuit

    shiftrne ShiftK (Data, LData, EK, 1'b0, Clock, K);
        defparam ShiftK.n = 8;

    // upcount is in Figure 7.57
    upcount CntL (4'b0, Resetn, Clock, EL, LL, L);

    assign Kgt1 = (K > 1) ? 1 : 0;

endmodule
```

10.13.

```
module mean (Clock, Resetn, Data, RAdd, s, ER, M, Done);
    parameter n = 8;
    input Clock, Resetn;
    input [n−1:0] Data;
    input [1:0] RAdd;
    input s, ER;
    output [n−1:0] M;
    output Done;

    reg LC, EC, Ssel, ES, LA, EB, LB, Div, Done;
    wire z, zz;
    reg [0:3] Dec_RAdd;
    wire [0:3] Rin;
    wire [1:0] C;
    wire [n−1:0] R0, R1, R2, R3, SR, Sin, Sum, Remainder, K;
    reg [n−1:0] Ri;
    reg [2:0] y, Y;
    parameter S1 = 3'b000, S2 = 3'b001, S3 = 3'b010, S4 = 3'b011, S5 = 3'b100;

    always @(s or y or z or zz)
    begin: State_table
        case (y)
            S1:   if (s == 0) Y = S1;
                  else Y = S2;
            S2:   if (z == 0) Y = S2;
                  else Y = S3;
            S3:   Y = S4;
            S4:   if (zz == 0) Y = S4;
                  else Y = S5;
            S5:   if (s == 1) Y = S5;
                  else Y = S1;
            default: Y = 3'bxxx;
        endcase
    end
    always @(posedge Clock or negedge Resetn)
    begin: State_flipflops
        if (Resetn == 0)
            y <= S1;
        else
            y <= Y;
    end

    always @(y or s or z or zz)
    begin: FSM_outputs
        LC = 0; EC = 0; ES = 0; LA = 0; EB = 0; // defaults
        Div = 0; Done = 0; Ssel = 0; // defaults
        case (y)
            S1:  begin
                     LC = 1; ES = 1;
                 end
            S2:  begin
                     Ssel = 1; ES = 1;
                     if (z == 0) EC = 1;
                     else EC = 0;
                 end
            S3:  begin
                     LA = 1; EB = 1;
                 end
            S4:  Div = 1; // not really used in this circuit
            S5:  begin
                     Div = 1; Done = 1;
                 end
        endcase
    end

// Datapath

    always @(RAdd)
    begin
        case (RAdd)
            0: Dec_RAdd = 4'b1000;
            1: Dec_RAdd = 4'b0100;
            2: Dec_RAdd = 4'b0010;
            3: Dec_RAdd = 4'b0001;
        endcase
    end

    assign Rin = (ER == 1) ? Dec_RAdd : 4'b0000;

    regne Reg0 (Data, Clock, Resetn, Rin[0], R0);
        defparam Reg0.n = n;
    regne Reg1 (Data, Clock, Resetn, Rin[1], R1);
        defparam Reg1.n = n;
        regne Reg2 (Data, Clock, Resetn, Rin[2], R2);
            defparam Reg2.n = n;
        regne Reg3 (Data, Clock, Resetn, Rin[3], R3);
            defparam Reg3.n = n;

        downcount Counter (Clock, EC, LC, C);
            defparam Counter.n = 2;
        assign z = (C == 0) ? 1 : 0;
        assign Sin = (Ssel == 1) ? Sum : 0;

        regne RegS (Sin, Clock, Resetn, ES, SR);
            defparam RegS.n = n;

        always @(C)
        begin
            case (C)
                0: Ri = R0;
                1: Ri = R1;
                2: Ri = R2;
                3: Ri = R3;
            endcase
        end

        assign Sum = SR + Ri;
        //divide by 4
        assign M = 2'b00, SR[n−1:2];
        assign zz = 1;

    endmodule
```

10.14. From Figures 10.26 and 10.27, we can see that the divider subcircuit does not use its adder while in state S1. Since the control circuit for the divider stays in S1 while $s = 0$, it is possible to lend the adder to another circuit while we are in S1 and $s = 0$. The Figure below shows the changes needed in the divider circuit: a multiplexer is added to each data input on the adder. The multiplexer select line is driven by the divider's s input — this signal is called *Div* in the figure, because *Div* is the signal in the Mean circuit that drives the *s* input on the divider subcircuit. When *Div* $= 1$ the adder is provided with the normal data used in the division operation. But when *Div* $= 0$ the adder is provided with the external data inputs named *Op1* and *Op2*, which come from the Mean circuit. Note that the C_{in} input on the adder is controlled by *Div*. This feature is needed because the divider uses its adder to perform subtraction.



10.15. Verilog code for the modified divider circuit is shown below.

```
module divider (Clock, Resetn, s, LA, EB, DataA, DataB, R, Q, Done, Op1, Op2, Result);
    parameter n = 8, logn = 3;
    input Clock, Resetn, s, LA, EB;
    input [n−1:0] DataA, DataB;
    output [n−1:0] R, Q;
    output Done;
    input [n−1:0] Op1, Op2; // new ports
    output [n−1:0] Result; // new port

    wire Cout, z;
    wire [n−1:0] DataR, AddIn1, AddIn2;
    wire [n−1:0] Sum;
    reg [1:0] y, Y;
    reg [n−1:0] A, B;
    reg [logn−1:0] Count;
    reg Done, EA, Rsel, LR, ER, ER0, LC, EC, R0;
    integer k;

// control circuit

    parameter S1 = 2'b00, S2 = 2'b01, S3 = 2'b10;

    always @(s or y or z)
    begin: State_table
        . . . code not shown: see Figure 10.28
    end
    always @(posedge Clock or negedge Resetn)
    begin: State_flipflops
        if (Resetn == 0)
            y <= S1;
        else
            y <= Y;
    end

    always @(y or s or Cout or z)
    begin: FSM_outputs
        . . . code not shown: see Figure 10.28
    end

//datapath circuit

    regne RegB (DataB, Clock, Resetn, EB, B);
        defparam RegB.n = n;

    shiftlne ShiftR (DataR, LR, ER, R0, Clock, R);
        defparam ShiftR.n = n;

    muxdff FF_R0 (0, A[n−1], ER0, Clock, R0);

    shiftlne ShiftA (DataA, LA, EA, Cout, Clock, A);
        defparam ShiftA.n = n;
        assign Q = A;

    downcount Counter (Clock, EC, LC, Count);
        defparam Counter.n = logn;

    assign z = (Count == 0);

    // new code for the divider
    assign AddIn1 = (s == 1) ? {R, R0} : Op1;
    assign AddIn2 = (s == 1) ? ~B : Op2;
    assign {Cout, Sum} = AddIn1 + AddIn2 + s;

    // define the n 2-to-1 multiplexers
    assign DataR = Rsel ? Sum : 0;
    assign Result = Sum;

endmodule
```

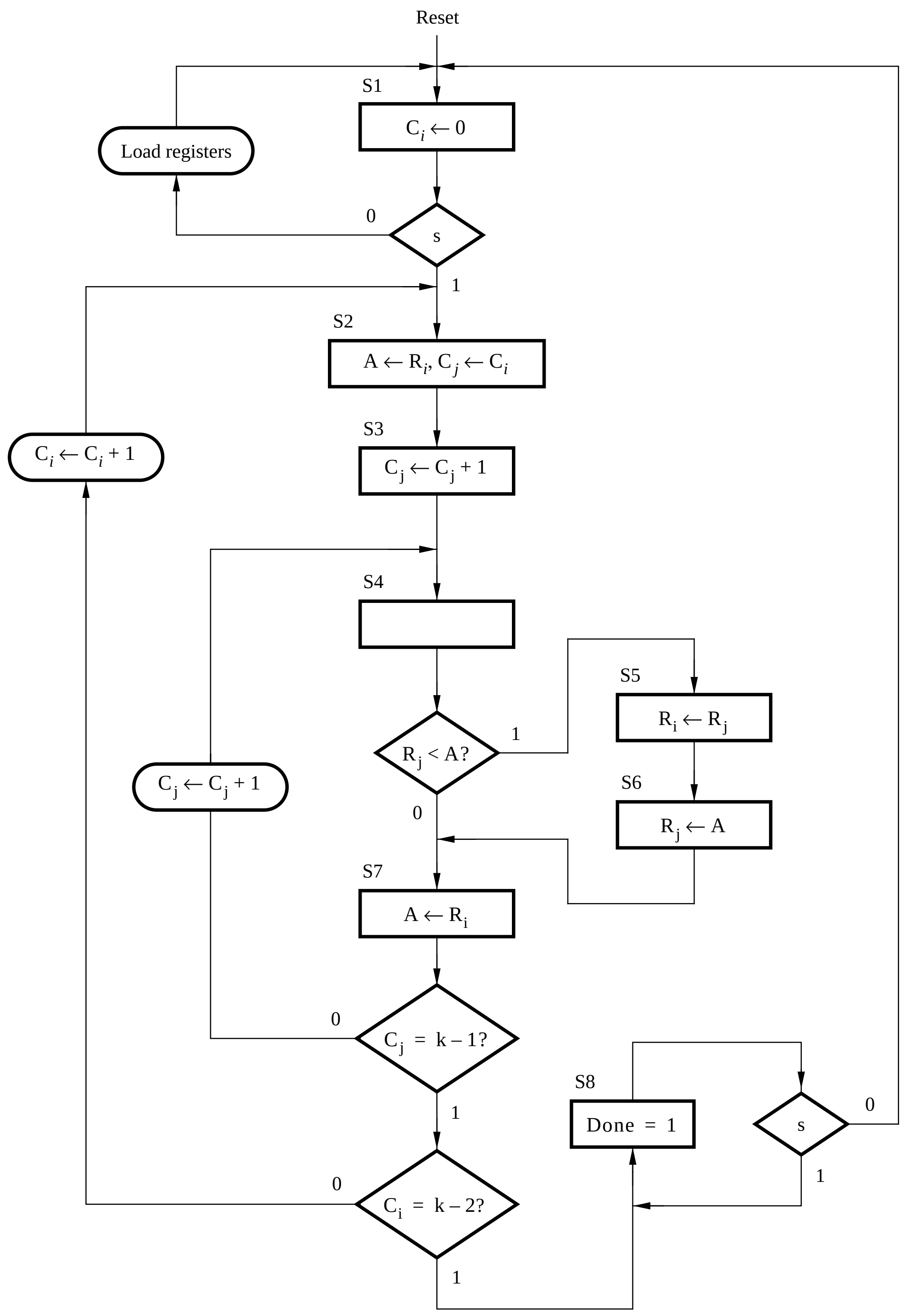
Code for the modified Mean circuit is shown below.

```
module mean (Clock, Resetn, Data, RAdd, s, ER, M, Done);
    parameter n = 8;
    input Clock, Resetn;
    input [n−1:0] Data;
    input [1:0] RAdd;
    input s, ER;
    output [n−1:0] M;
    output Done;

    reg LC, EC, Ssel, ES, LA, EB, LB, zz, Div, Done;
    wire z;
    reg [0:3] Dec_RAdd;
    wire [0:3] Rin;
    wire [1:0] C;
    wire [n−1:0] R0, R1, R2, R3, SR, Sin, Sum, Remainder, K;
    reg [n−1:0] Ri;
    reg [2:0] y, Y;
    parameter S1 = 3'b000, S2 = 3'b001, S3 = 3'b010, S4 = 3'b011, S5 = 3'b100;

    always @(s or y or z or zz)
    begin: State_table
        case (y)
            S1:   if (s == 0) Y = S1;
                  else Y = S2;
            S2:   if (z == 0) Y = S2;
                  else Y = S3;
            S3:   Y = S4;
            S4:   if (zz == 0) Y = S4;
                  else Y = S5;
            S5:   if (s == 1) Y = S5;
                  else Y = S1;
            default: Y = 3'bxxx;
        endcase
    end

    always @(posedge Clock or negedge Resetn)
    begin: State_flipflops
        if (Resetn == 0)
            y <= S1;
        else
            y <= Y;
    end

    always @(y or s or z or zz)
    begin: FSM_outputs
        LC = 0; EC = 0; ES = 0; LA = 0; EB = 0; // defaults
        Div = 0; Done = 0; Ssel = 0; // defaults
        case (y)
            S1:   begin
                      LC = 1; EC = 1; ES = 1;
                  end
            S2:  begin
                     Ssel = 1; ES = 1;
                     if (z == 0) EC = 1;
                     else EC = 0;
                 end
            S3:  begin
                     LA = 1; EB = 1;
                 end
            S4:  Div = 1;
            S5:  begin
                     Div = 1; Done = 1;
                 end
        endcase
    end

// Datapath

    always @(RAdd)
    begin
        case (RAdd)
            0: Dec_RAdd = 4'b1000;
            1: Dec_RAdd = 4'b0100;
            2: Dec_RAdd = 4'b0010;
            3: Dec_RAdd = 4'b0001;
        endcase
    end

    assign Rin = (ER == 1) ? Dec_RAdd : 4'b0000;

    regne Reg0 (Data, Clock, Resetn, Rin[0], R0);
        defparam Reg0.n = n;
    regne Reg1 (Data, Clock, Resetn, Rin[1], R1);
        defparam Reg1.n = n;
    regne Reg2 (Data, Clock, Resetn, Rin[2], R2);
        defparam Reg2.n = n;
    regne Reg3 (Data, Clock, Resetn, Rin[3], R3);
        defparam Reg3.n = n;

    downcount Counter (Clock, EC, LC, C);
        defparam Counter.n = 2;
    assign z = (C == 0) ? 1 : 0;
    assign Sin = (Ssel == 1) ? Sum : 0;

    regne RegS (Sin, Clock, Resetn, ES, SR);
        defparam RegS.n = n;
    always @(C)
    begin
        case (C)
            0: Ri = R0;
            1: Ri = R1;
            2: Ri = R2;
            3: Ri = R3;
        endcase
    end

    divider DivideBy4 (.Clock(Clock), .Resetn(Resetn), .s(Div), .LA(LA), .EB(EB),
        .DataA(SR), .DataB(4), .R(Remainder), .Q(M), .Done(zz), .Op1(SR),
        .Op2(Ri), .Result(Sum));

endmodule
```

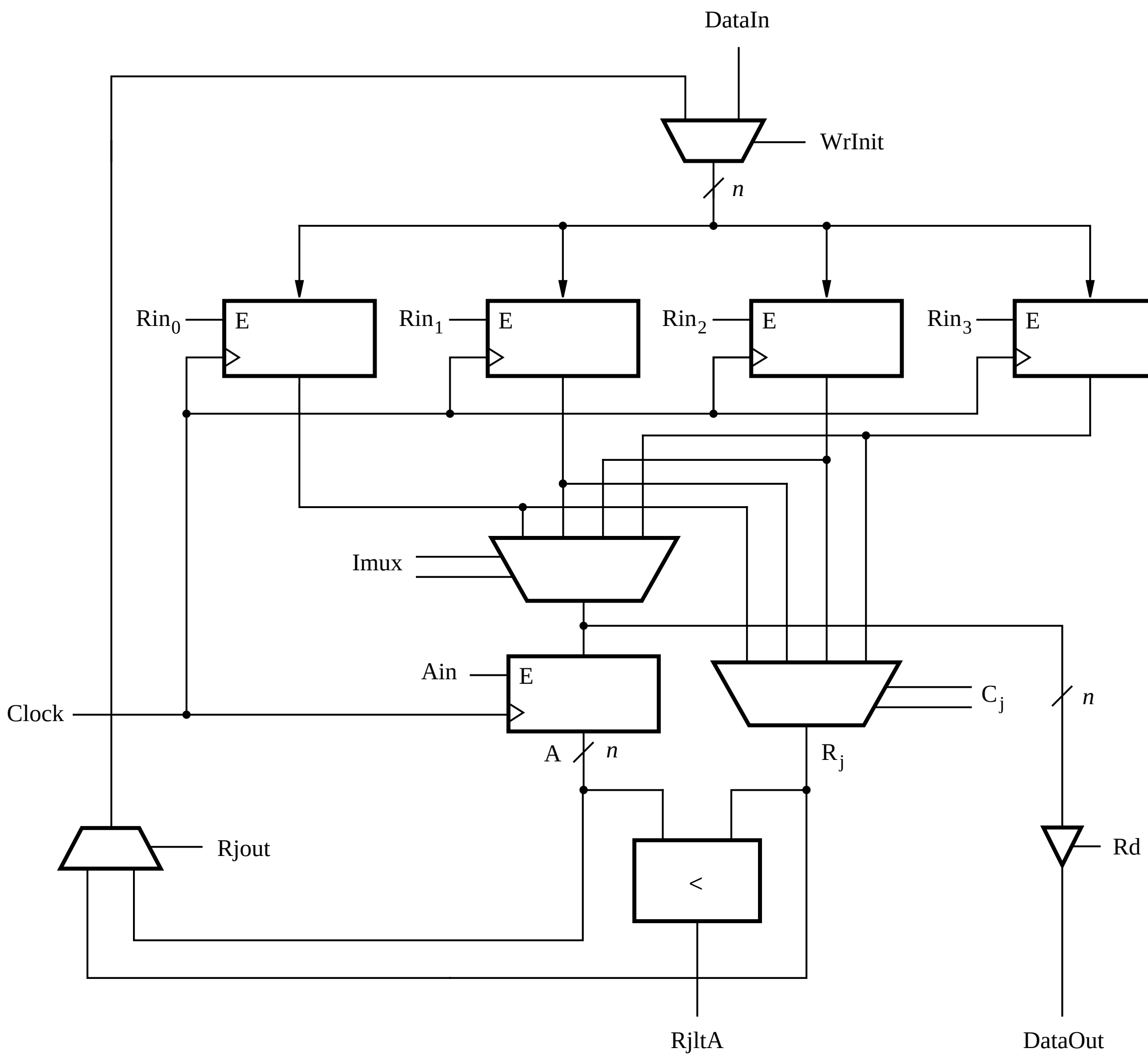
10.16. The modified pseudo-code is

```
for $i = 0$ to $k - 2$ do
    $A = R_i$ ;
    for $j = i + 1$ to $k - 1$ do
        if $R_j < A$ then
            $R_i = R_j$ ;
            $R_j = A$ ;
        end if ;
        $A = R_i$ ;
    end for ;
end for ;
```

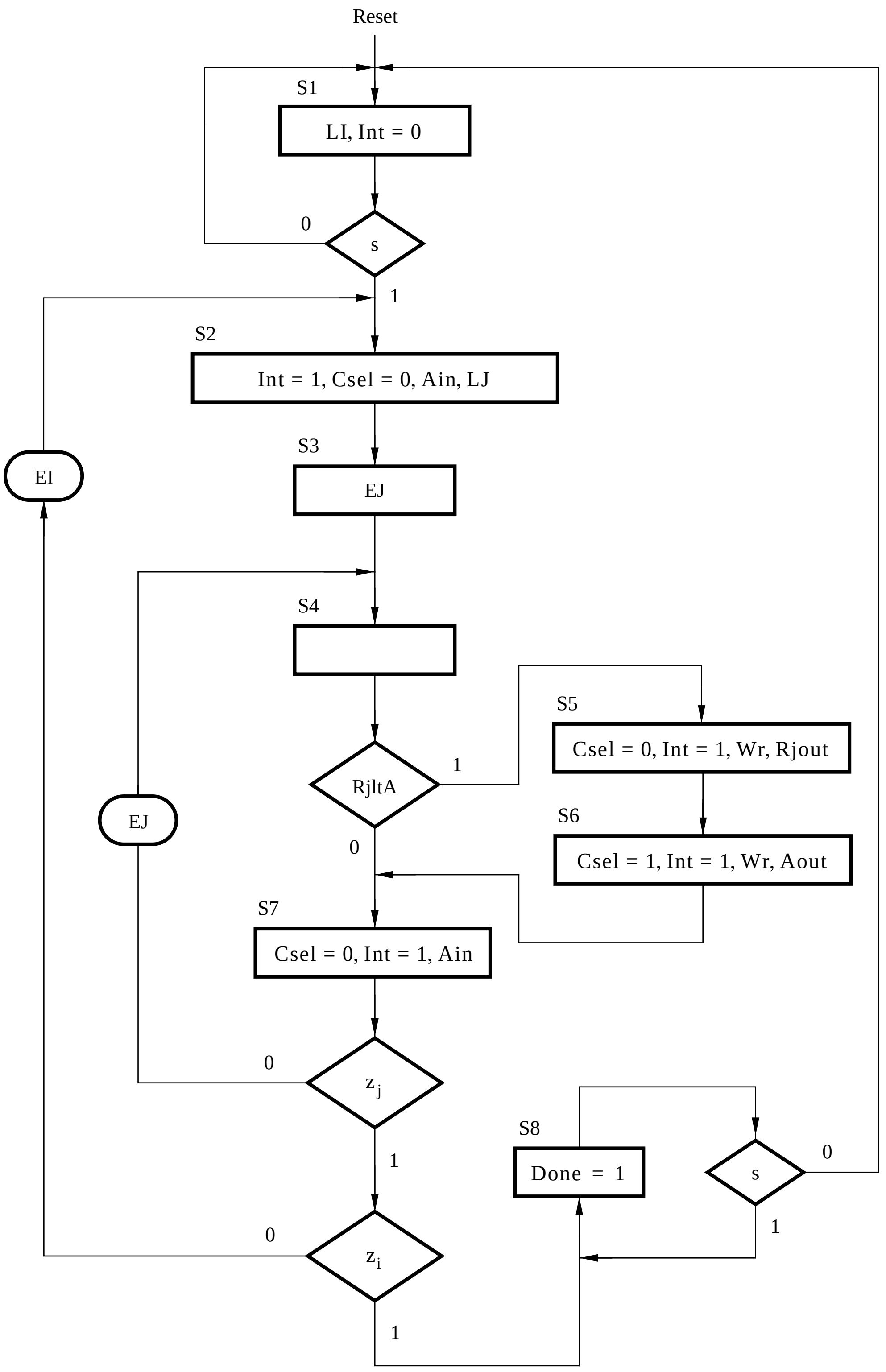
An ASM chart that corresponds to the pseudo-code is


Reset
S1
$C_i \leftarrow 0$
Load registers
s
0
1
S2
$A \leftarrow R_i, C_j \leftarrow C_i$
S3
$C_j \leftarrow C_j + 1$
$C_i \leftarrow C_i + 1$
S4
S5
$R_i \leftarrow R_j$
$R_j < A?$
1
0
S6
$R_j \leftarrow A$
$C_j \leftarrow C_j + 1$
S7
$A \leftarrow R_i$
0
$C_j = k - 1?$
1
S8
Done = 1
s
0
1
0
$C_i = k - 2?$
1

From the ASM chart, we can see that the datapath circuit needs a multiplexer to allow the operation $R_i \leftarrow R_j$. An appropriate datapath is shown below.



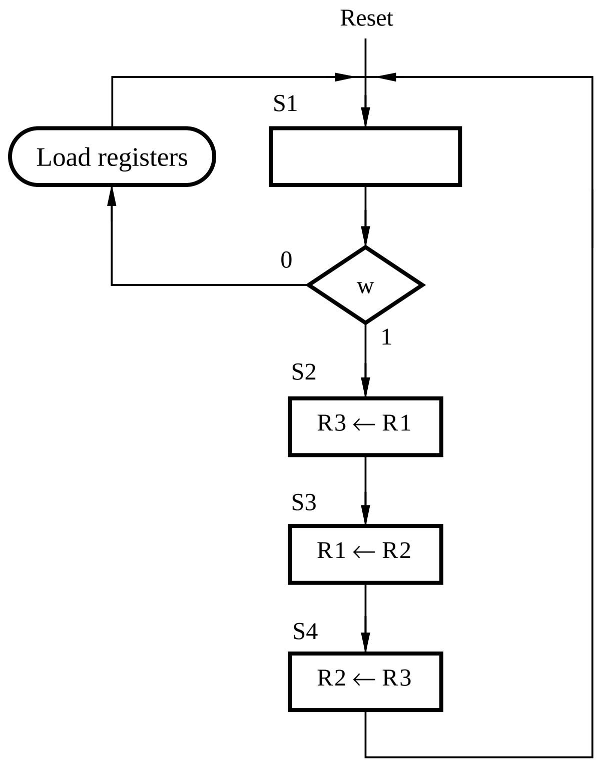
An ASM chart for the control circuit is


Reset
S1
LI, Int = 0
s
0
1
S2
Int = 1, Csel = 0, Ain, LJ
S3
EJ
EI
S4
S5
Csel = 0, Int = 1, Wr, Rjout
RjltA
1
0
S6
Csel = 1, Int = 1, Wr, Aout
EJ
S7
Csel = 0, Int = 1, Ain
z_j
0
1
S8
Done = 1
s
0
1
z_i
0
1

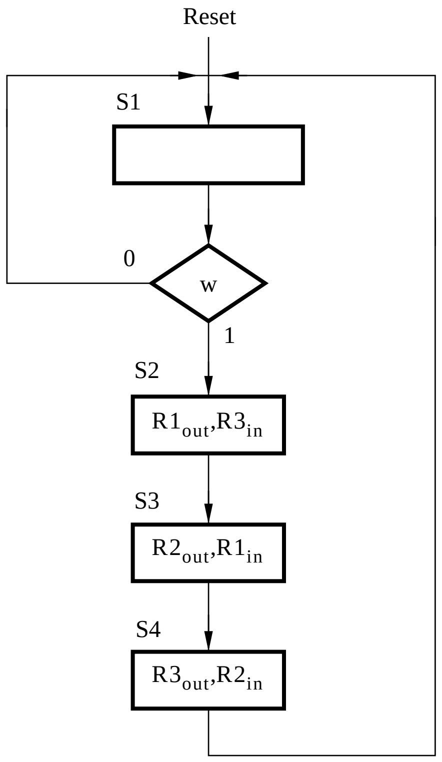
10.17.

```
module sort (Clock, Resetn, s, WrInit, Rd, DataIn, RAdd, DataOut, Done);
    parameter n = 4;
    input Clock, Resetn, s, WrInit, Rd;
    input [n−1:0] DataIn;
    input [1:0] RAdd;
    output [n−1:0] DataOut;
    output Done;

    wire [1:0] Ci, Cj, CMux, IMux;
    wire [n−1:0] R0, R1, R2, R3, A;
    wire [n−1:0] RData, ARjMux;
    wire RjltA;
    wire zi, zj;
    reg Int, Csel, Wr, Ain;
    reg LI, LJ, EI, EJ, Done, RjOut;
    reg [n−1:0] Rj;
    reg [2:0] y, Y;
    reg Rin0, Rin1, Rin2, Rin3;
    reg [n−1:0] AData;

// control circuit

    parameter S1 = 3'b000, S2 = 3'b001, S3 = 3'b010, S4 = 3'b011;
    parameter S5 = 3'b100, S6 = 3'b101, S7 = 3'b110, S8 = 3'b111;

    always @(s or RjltA or zj or zi)
    begin: State_table
        case (y)
            S1:   if (s == 0) Y = S1;
                  else Y = S2;
            S2:   Y = S3;
            S3:   Y = S4;
            S4:   if (RjltA == 1) Y = S5;
                  else Y = S7;
            S5:   Y = S6;
            S6:   Y = S7;
            S7:   if (!zj) Y = S4;
                  else if (!zi) Y = S2;
                  else Y = S8;
            S8:   if (s) Y = S8;
                  else Y = S1;
            default: Y = 4'bx;
        endcase
    end
    always @(posedge Clock or negedge Resetn)
    begin: State_flipflops
        if (Resetn == 0)
            y <= S1;
        else
            y <= Y;
    end

    assign Int = (y != S1);
    assign Done = (y == S8);

    always @(y or zj or zi)
    begin: FSM_outputs
        LI = 0; LJ = 0; EI = 0; EJ = 0; Csel = 0; // defaults
        Wr = 0; Ain = 0; RjOut = 0; // defaults
        case (y)
            S1:  begin
                     LI = 1;
                 end
            S2:  begin
                     Ain = 1; LJ = 1;
                 end
            S3:  EJ = 1;
            S4:  ;
            S5:  begin
                     RjOut = 1; Wr = 1;
                 end
            S6:  begin
                     Wr = 1; Csel = 1;
                 end
            S7:  begin
                     Ain = 1;
                     if (!zj) EJ = 1;
                     else
                     begin
                         EJ = 0;
                         if (!zi) EI = 1;
                         else EI = 0;
                     end
                 end
            S8:  ;
        endcase
    end

//datapath circuit

    regne Reg0 (RData, Clock, Resetn, Rin0, R0);
        defparam Reg0.n = n;
    regne Reg1 (RData, Clock, Resetn, Rin1, R1);
        defparam Reg1.n = n;
    regne Reg2 (RData, Clock, Resetn, Rin2, R2);
        defparam Reg2.n = n;
    regne Reg3 (RData, Clock, Resetn, Rin3, R3);
        defparam Reg3.n = n;
    regne RegA (AData, Clock, Resetn, Ain, A);
        defparam RegA.n = n;

    assign RjltA = (Rj < A) ? 1 : 0;
    assign ARjMux = (RjOut == 0) ? A : Rj;
    assign RData = (WrInit == 0) ? ARjMux : DataIn;

    upcount OuterLoop (0, Resetn, Clock, EI, LI, Ci);
    upcount InnerLoop (Ci, Resetn, Clock, EJ, LJ, Cj);

    assign CMux = (Csel == 0) ? Ci : Cj;
    assign IMux = (Int == 1) ? CMux : RAdd;

    always @(WrInit or Wr or IMux or Cj)
    begin
        case (IMux)
            0: AData = R0;
            1: AData = R1;
            2: AData = R2;
            3: AData = R3;
        endcase

        case (Cj)
            0: Rj = R0;
            1: Rj = R1;
            2: Rj = R2;
            3: Rj = R3;
        endcase

        if (WrInit || Wr)
            case (IMux)
                0: {Rin3, Rin2, Rin1, Rin0} = 4'b0001;
                1: {Rin3, Rin2, Rin1, Rin0} = 4'b0010;
                2: {Rin3, Rin2, Rin1, Rin0} = 4'b0100;
                3: {Rin3, Rin2, Rin1, Rin0} = 4'b1000;
            endcase
        else {Rin3, Rin2, Rin1, Rin0} = 4'b0000;
    end

    assign zi = (Ci == 2);
    assign zj = (Cj == 3);
    assign DataOut = (Rd == 0) ? 'bz : AData;

endmodule
```

10.18.



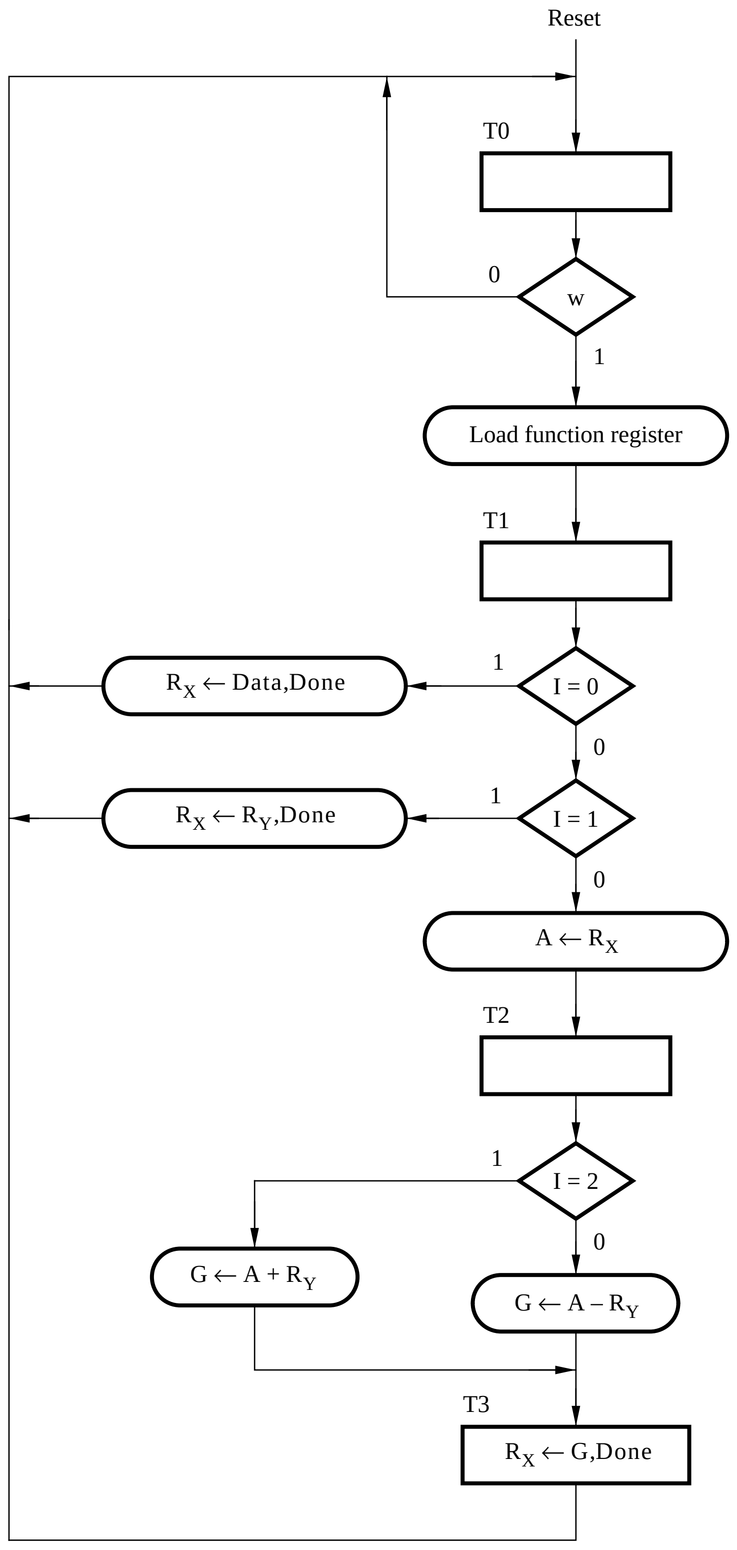
10.19. (*a*) An ASM chart for the control circuit is



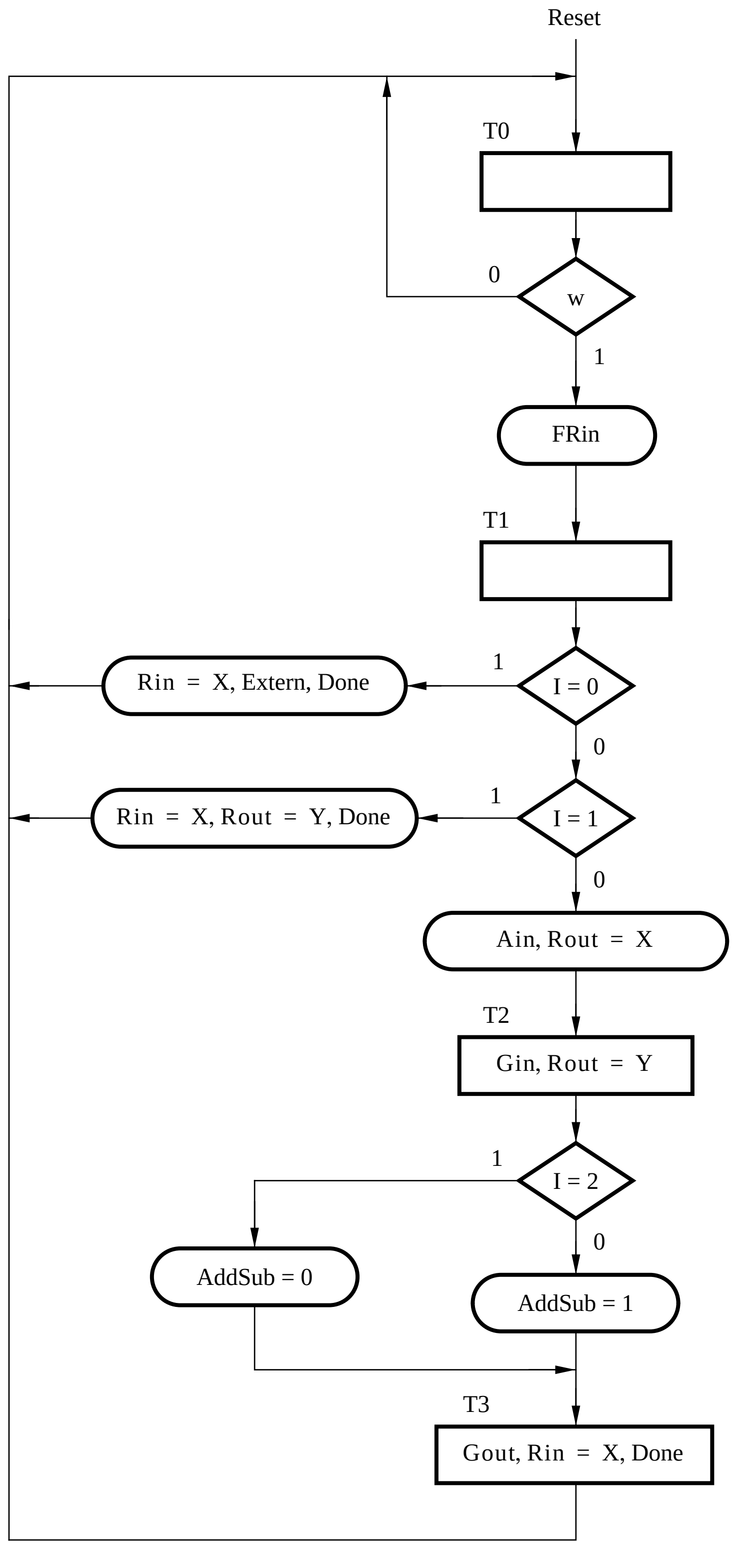
(*b*)

```
module swapmux (Data, Resetn, w, Clock, RinExt, BusWires);
    input   [7:0] Data;
    input   Resetn, w, Clock;
    input   [1:3] RinExt;
    output   [7:0] BusWires;
    reg   [7:0] BusWires;
    wire   [1:3] Rin;
    reg   [1:3] RinCntl, Rout;
    wire   [7:0] R1, R2, R3;
    reg   [1:0] y, Y;

// control circuit

    parameter S1 = 2'b00, S2 = 2'b01, S3 = 2'b10, S4 = 2'b11;

    always @(y or w)
    begin: State_table
        case (y)
            S1:   if (w == 0) Y = S1;
                  else Y = S2;
            S2:   Y = S3;
            S3:  Y = S4;
            S4:  Y = S1;
            default: Y = 3'bxxx;
        endcase
    end

    always @(posedge Clock or negedge Resetn)
    begin: State_flipflops
        if (Resetn == 0)
            y <= S1;
        else
            y <= Y;
    end

    always @(y)
    begin: FSM_outputs
        RinCntl = 3'b000; Rout = 3'b000; // defaults
        case (y)
            S1:  ;
            S2:  begin
                     Rout[1] = 1; RinCntl[3] = 1;
                 end
            S3:  begin
                     Rout[2] = 1; RinCntl[1] = 1;
                 end
            S4:  begin
                     Rout[3] = 1; RinCntl[2] = 1;
                 end
        endcase
    end

// datapath circuit
    assign Rin = RinExt | RinExt;
    regn reg_1 (BusWires, Rin[1], Clock, R1);
    regn reg_2 (BusWires, Rin[2], Clock, R2);
    regn reg_3 (BusWires, Rin[3], Clock, R3);

    always @(Rout or Data or R1 or R2 or R3)
    begin
        if (Rout == 3'b100)
            BusWires = R1;
        else if (Rout == 3'b010)
            BusWires = R2;
        else if (Rout == 3'b001)
            BusWires = R3;
        else
            BusWires = Data;
    end

endmodule
```

10.20. An ASM chart for the processor is



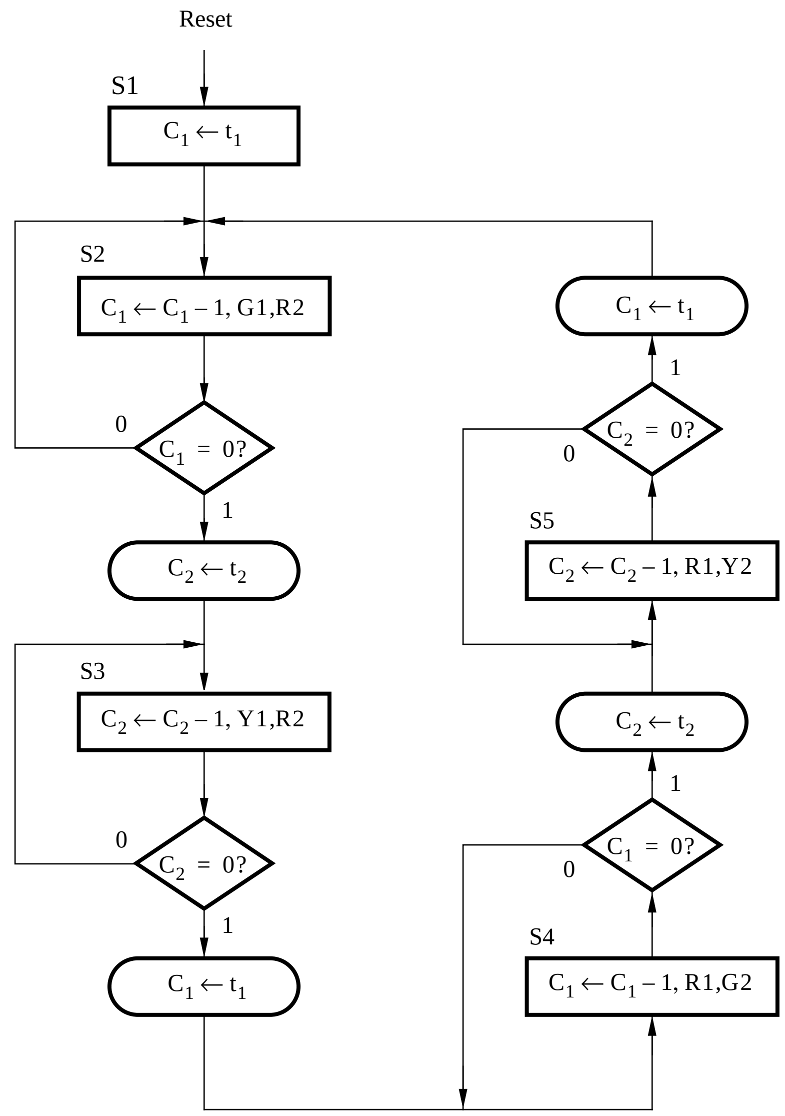
10.21. (a) An ASM chart for the control circuit is



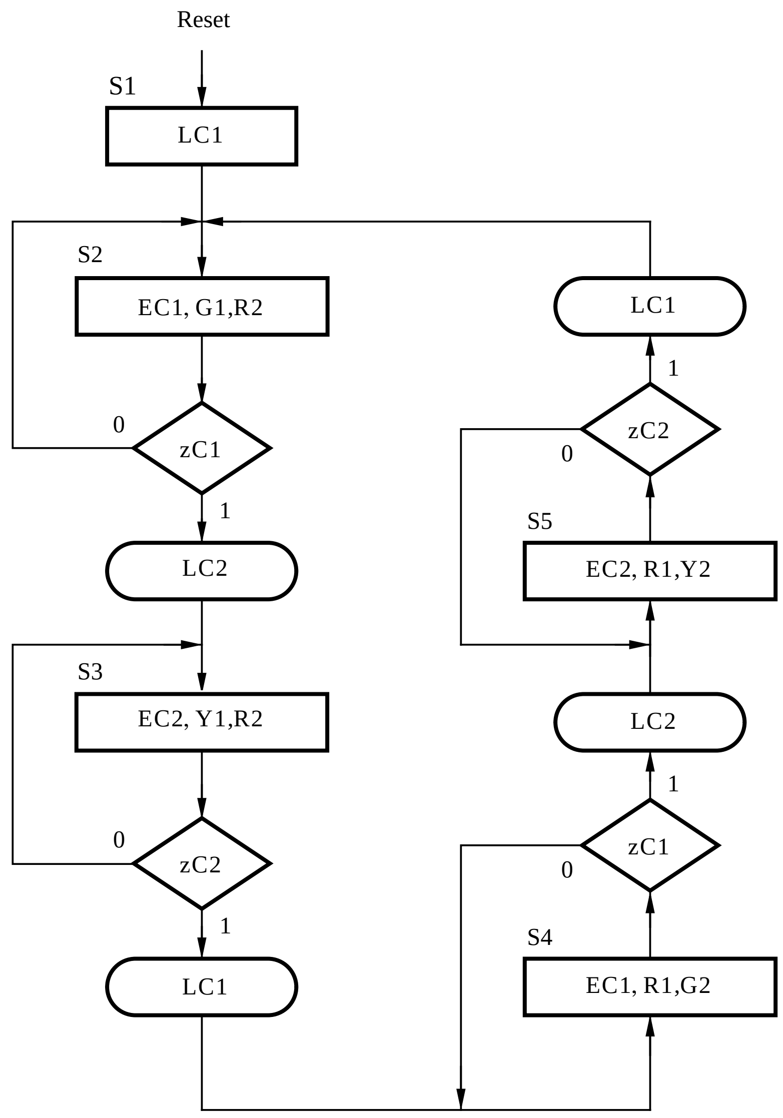
(*b*)

```
module proc (Data, Reset, w, Clock, F, Rx, Ry, Done, BusWires);
    input [7:0] Data;
    input Reset, w, Clock;
    input [1:0] F, Rx, Ry;
    output [7:0] BusWires;
    output Done;
    reg [7:0] BusWires;
    reg [0:3] Rin, Rout;
    reg [7:0] Sum;
    reg Extern, Done, Ain, FRin, Gin, Gout, AddSub;
    wire [1:0] Count, I;
    wire [0:3] Xreg, Y;
    wire [7:0] R0, R1, R2, R3, A, G;
    wire [1:6] Func, FuncReg, Sel;

    reg [1:0] t, T;

// control circuit

    parameter T0 = 2'b00, T1 = 2'b01, T2 = 2'b10, T3 = 2'b11;

    always @(t or w or I)
    begin: State_table
        case (t)
            T0:   if (w == 0) T = T0;
                  else T = T1;
            T1:   if (I == 2'b00 || I == 2'b01) T = T0;
                  else T = T2;
            T2:   T = T3;
            T3:   T = T0;
            default: T = 2'bxx;
        endcase
    end

    always @(posedge Clock or posedge Reset)
    begin: State_flipflops
        if (Reset == 1)
            t <= T0;
        else
            t <= T;
    end
    always @(t or w or I)
    begin: FSM_outputs
        FRin = 0; Rin = 4'b0000; Rout = 4'b0000; Done = 0; // defaults
        Gin = 0; Gout = 0; Extern = 0; Ain = 0; AddSub = 0; // defaults
        case (t)
            T0: if (w == 1) FRin = 1;
                else FRin = 0;
            T1: begin
                    Ain = 1; // doesn't matter if we load A when not needed
                    if (I == 2'b00)
                    begin
                        Done = 1; Rin = Xreg; Rout = 4'b0000; Extern = 1;
                    end
                    else if (I == 2'b01)
                    begin
                        Done = 1; Rin = Xreg; Rout = Y; Extern = 0;
                    end
                    else
                    begin
                        Done = 0; Rin = 4'b0000; Rout = Xreg; Extern = 0;
                    end
                end
            T2: begin
                    Gin = 1; Rout = Y;
                    if (I == 2'b10) AddSub = 0;
                    else AddSub = 1;
                end
            T3: begin
                    Gout = 1; Rin = Xreg; Done = 1;
                end
        endcase
    end

//datapath circuit
    assign Func = {F, Rx, Ry};
    regn functionreg (Func, FRin, Clock, FuncReg);
        defparam functionreg.n = 6;
    assign I = FuncReg[1:2];
    dec2to4 decX (FuncReg[3:4], 1, Xreg);
    dec2to4 decY (FuncReg[5:6], 1, Y);

    regn reg_0 (BusWires, Rin[0], Clock, R0);
    regn reg_1 (BusWires, Rin[1], Clock, R1);
    regn reg_2 (BusWires, Rin[2], Clock, R2);
    regn reg_3 (BusWires, Rin[3], Clock, R3);
    regn reg_A (BusWires, Ain, Clock, A);
    // alu
    always @(AddSub or A or BusWires)
    begin
        if (!AddSub)
            Sum = A + BusWires;
        else
            Sum = A − BusWires;
    end

    regn reg_G (Sum, Gin, Clock, G);
    assign Sel = {Rout, Gout, Extern};

    always @(Sel or R0 or R1 or R2 or R3 or G or Data)
    begin
        if (Sel == 6’b100000)
            BusWires = R0;
        else if (Sel == 6’b010000)
            BusWires = R1;
        else if (Sel == 6’b001000)
            BusWires = R2;
        else if (Sel == 6’b000100)
            BusWires = R3;
        else if (Sel == 6’b000010)
            BusWires = G;
        else BusWires = Data;
    end

endmodule
```

10.22. (a) An ASM chart for the traffic controller is



(b). The two counters, C_1 and C_2, each require clock enable and parallel-load inputs. Assuming that the clock enables signals are called *EC1* and *EC2* and the parallel-load inputs are called *LC1* and *LC2*, an ASM chart for the control circuit is

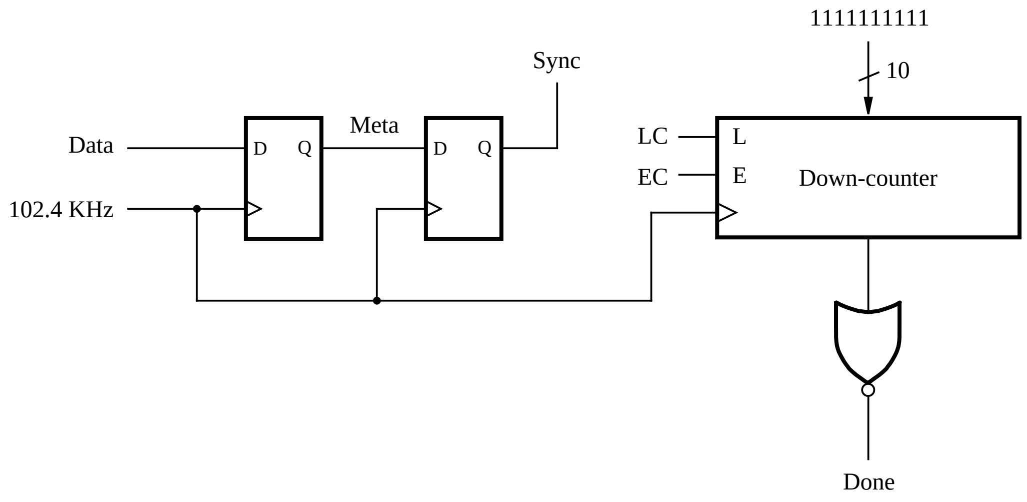


(*c*)

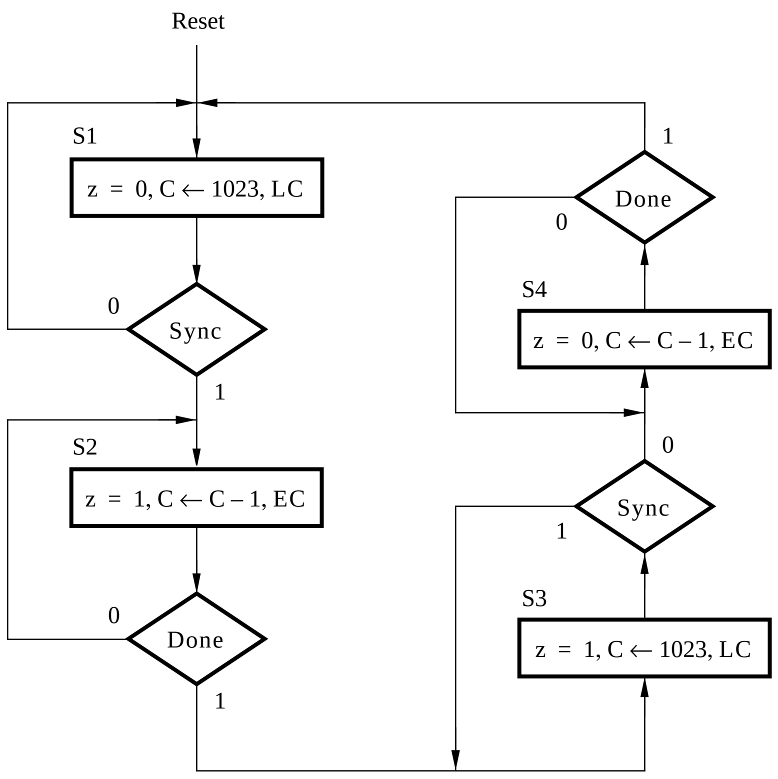
```
module traffic (Clock, Resetn, G1, Y1, R1, G2, Y2, R2);
  input Clock, Resetn;
  output G1, Y1, R1, G2, Y2, R2;
  reg G1, Y1, R1, G2, Y2, R2;

  reg [2:0] y, Y;
  reg EC1, EC2, LC1, LC2;
  reg [3:0] C1, C2;
  wire zC1, zC2;
  parameter Ticks1 = 4'b0011; // 4 ticks for C1
  parameter Ticks2 = 4'b0001; // 2 ticks for C2
```

```
// control circuit

    parameter S1 = 3'b000, S2 = 3'b001, S3 = 3'b010, S4 = 3'b011, S5 = 3'b100;

    always @(y or zC1 or zC2)
    begin: State_table
        case (y)
            S1:  Y = S2;
            S2:  if (zC1 == 0) Y = S2;
                 else Y = S3;
            S3:  if (zC2 == 0) Y = S3;
                 else Y = S4;
            S4:  if (zC1 == 0) Y = S4;
                 else Y = S5;
            S5:  if (zC2 == 0) Y = S5;
                 else Y = S2;
            default: Y = 3'bxxx;
        endcase
    end

    always @(posedge Clock or negedge Resetn)
    begin: State_flipflops
        if (Resetn == 0)
            y <= S1;
        else
            y <= Y;
    end

    always @(y or zC1 or zC2)
    begin: FSM_outputs
        G1 = 0; Y1 = 0; R1 = 0; G2 = 0; Y2 = 0; R2 = 0; // defaults
        LC1 = 0; EC1 = 0; LC2 = 0; EC2 = 0; // defaults
        case (y)
            S1:  LC1 = 1;
            S2:  begin
                     EC1 = 1; G1 = 1; R2 = 1;
                     if (zC1) LC2 = 1;
                     else LC2 = 0;
                 end
            S3:  begin
                     EC2 = 1; Y1 = 1; R2 = 1;
                     if (zC2) LC1 = 1;
                     else LC2 = 0;
                 end
            S4:  begin
                     EC1 = 1; R1 = 1; G2 = 1;
                     if (zC1) LC2 = 1;
                     else LC2 = 0;
                 end
        S5: begin
                EC2 = 1; R1 = 1; Y2 = 1;
                if (zC2) LC1 = 1;
                else LC2 = 0;
            end
    endcase
end

//datapath circuit

assign zC1 = (C1 == 0);
assign zC2 = (C2 == 0);

always @(posedge Clock)
    if (LC1)
        C1 <= Ticks1;
    else if (EC1)
        C1 <= C1 − 1;

always @(posedge Clock)
    if (LC2)
        C2 <= Ticks2;
    else if (EC2)
        C2 <= C2 − 1;

endmodule
```

10.23. The debounce circuit has three parts, as shown below. The *Data* signal from the switch has to be synchronized to the 102.4 KHz signal using two flip-flops. The synchronized signal called *Sync* is fed to an FSM. The FSM also uses the counter shown, which counts for 1024 cycles of the 102.4 KHz signal, providing a 10 msec delay.



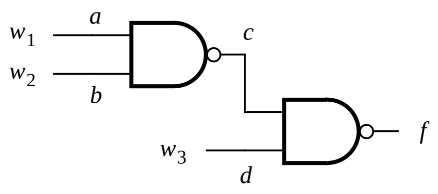
An ASM chart for the FSM is given below. The FSM provides the *z* output, which is the debounced version of the *Data* signal.



10.24. (a) If we set $C_1 = 1$ pF, then $R_a = 0$ and $R_b = 1.43$ kΩ
(b) If we set $C_1 = 1$ pF, then $R_a = 1.42$ kΩ and $R_b = 0.71$ kΩ

Chapter 11

11.1. Label the wires in the circuit of Figure P11.1 as follows:

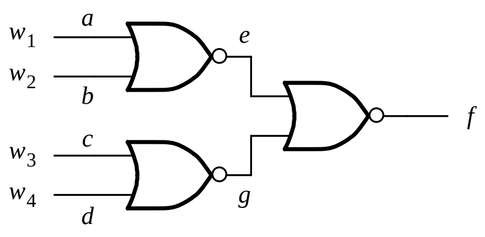


A complete fault table is

Test $w_1w_2w_3$	Fault detected									
	$a/0$	$a/1$	$b/0$	$b/1$	$c/0$	$c/1$	$d/0$	$d/1$	$f/0$	$f/1$
000								$\surd$	$\surd$	
001					$\surd$		$\surd$			$\surd$
010								$\surd$	$\surd$	
011		$\surd$			$\surd$		$\surd$			$\surd$
100								$\surd$	$\surd$	
101				$\surd$	$\surd$		$\surd$			$\surd$
110									$\surd$	
111	$\surd$		$\surd$			$\surd$			$\surd$	

A minimal test set must include the tests $w_1w_2w_3 = 011, 101$, and 111, which cover all faults except $d/1$. The latter fault can be detected by choosing one of $000, 010$, or 100.

11.2. Label the wires in the circuit of Figure P11.2 as follows:



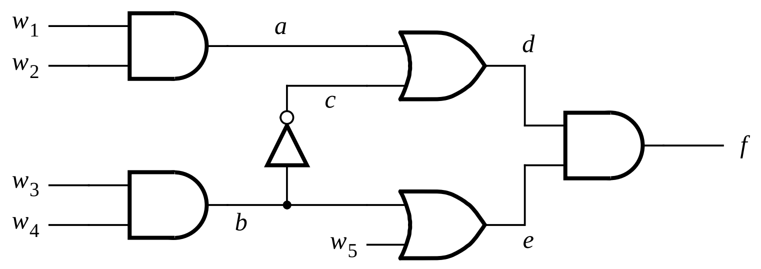
A complete fault table is

Test	Fault detected													
$w_1w_2w_3w_4$	$a/0$	$a/1$	$b/0$	$b/1$	$c/0$	$c/1$	$d/0$	$d/1$	$e/0$	$e/1$	$g/0$	$g/1$	$f/0$	$f/1$
0000														$\surd$
0001		$\surd$		$\surd$					$\surd$					$\surd$
0010		$\surd$		$\surd$					$\surd$					$\surd$
0011		$\surd$		$\surd$					$\surd$					$\surd$
0100						$\surd$		$\surd$			$\surd$			$\surd$
0101			$\surd$				$\surd$			$\surd$		$\surd$	$\surd$	
0110			$\surd$		$\surd$					$\surd$		$\surd$	$\surd$	
0111			$\surd$							$\surd$			$\surd$	
1000						$\surd$		$\surd$			$\surd$			$\surd$
1001	$\surd$						$\surd$			$\surd$		$\surd$	$\surd$	
1010	$\surd$				$\surd$					$\surd$		$\surd$	$\surd$	
1011	$\surd$									$\surd$			$\surd$	
1100						$\surd$		$\surd$			$\surd$			$\surd$
1101							$\surd$					$\surd$	$\surd$	
1110					$\surd$							$\surd$	$\surd$	
1111										$\surd$		$\surd$	$\surd$	

A possible minimal test set consists of $w_1w_2w_3w_4$ = $0001, 0110, 1000$, and 1001.

11.3. The two functions differ only in the vertex $x_1x_2x_3x_4 = 0111$. Therefore, the circuits can be distinguished by applying this input valuation.

11.4. Label the wires in the circuit of Figure P11.3 as follows:



Path $w_1 - a - d - f$	is sensitized with $w_2w_3w_4w_5$ = 111x
Path $w_2 - a - d - f$	is sensitized with $w_1w_3w_4w_5$ = 111x
Path $w_3 - b - c - d - f$	is sensitized with $w_1w_2w_4w_5$ = 0x11
Path $w_3 - b - e - f$	is sensitized with $w_1w_2w_4w_5$ = 1110
Path $w_4 - b - c - d - f$	is sensitized with $w_1w_2w_3w_5$ = 0x11
Path $w_4 - b - e - f$	is sensitized with $w_1w_2w_3w_5$ = 1110
Path $w_5 - e - f$	is sensitized with $w_1w_2w_3w_4$ = xx0x

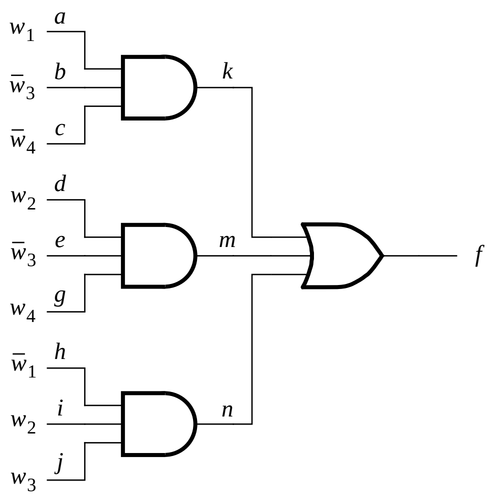
As an input signal to each path it is necessary to apply both 0 and 1 to give two tests. A possible test set is $w_1w_2w_3w_4w_5$ = 01111, 11110, 1011x, 0x011, 11010, 0x101, and 11100

11.5. The tests are $w_1w_2w_3w_4$ = 1111, 1110, 0111, and 1111.

11.6. Test 0100 detects $w_1/1,\ c/1,\ d/1,\ w_4/1$, and $f/1$.
Test 1010 detects $b/0,\ d/0,\ w_3/0$, and $f/0$.
Test 0011 detects $f/0$.
Test 1111 detects $f/0$.
Test 0110 detects $w_1/1,\ w_2/0,\ b/1,\ c/1,\ d/1,\ w_4/1$, and $f/1$.
Thus 11 different single faults can be detected using these four tests. Since the circuit has 8 wires, there can be 16 single $s/0$ or $s/1$ faults. Therefore, the tests cover 69% of single faults.

11.7. Test 0100 detects $w_1/1,\ b/0,\ c/0,\ g/1,\ h/0,\ k/0$, and $f/1$.
est 1010 detects $w_2/1,\ w_4/1,\ b/0,\ c/0,\ g/1,\ h/0,\ k/0$, and $f/1$.
Test 0011 detects $w_3/0,\ w_4/0,\ b/0,\ c/1,\ g/0,\ h/1$, and $f/0$.
Test 0110 detects $w_1/1,\ w_4/1,\ b/0,\ c/0,\ g/1,\ h/0,\ k/0$, and $f/1$.
Thus 15 different single faults can be detected using these four tests. Since the circuit has 10 wires, there can be 20 single $s/0$ or $s/1$ faults. Therefore, the tests cover 75% of single faults.

11.8. Label the wires in the circuit of Figure 11.5 as follows:



Test 0100 detects $a/1,\ g/1,\ j/1,\ k/1,\ m/1,\ n/1$, and $f/1$.
Test 1010 detects $b/1,\ k/1,\ m/1,\ n/1$,, and $f/1$.
Test 0011 detects $i/1,\ k/1,\ m/1,\ n/1$, and $f/1$.
Test 0110 detects $h/0,\ i/0,\ j/0,\ n/0$, and $f/0$.

Thus 14 different single faults can be detected using these four tests. Since the circuit has 13 wires, there can be 26 single $s/0$ or $s/1$ faults. Therefore, the tests cover 54% of single faults.

11.9. Cannot detect if the input wire w_1 is stuck-at-1. The reason is that this circuit is highly redundant. It realizes the function $f = w_3(\overline{w}_1 + \overline{w}_2)$, which can be implemented with a simpler circuit.

11.10. In a circuit in which all gates have a fan-out of 1 there exists a single path from any primary input to the output of the circuit. A test for a fault on a primary input sensitizes the path that leads from this input to the output of the circuit, thus testing for faults along this path. Therefore, a test set that tests all faults on the primary inputs, will also test all faults on the sensitized paths.

11.11. Test set = $\{0000, 0111, 1111, 1000\}$. It would work with XORs implemented as shown in Figure 4.28c.

For n bits, the same patterns can be used; thus
Test set = $\{00\cdots00, 011\cdots1, 11\cdots1, 100\cdots0\}$.

11.12. In the decoder circuit in Figure 6.16c the four AND gates are enabled only if the En signal is active. The required test set has to include all four valuations of w_1 and w_2 when $En = 1$. It is also necessary to test if the En wire is stuck at 1, which can be accomplished with the test $w_1w_2En = 000$. Therefore, a complete test set comprises w_1w_2En = 000, 001, 011, 101, and 111.

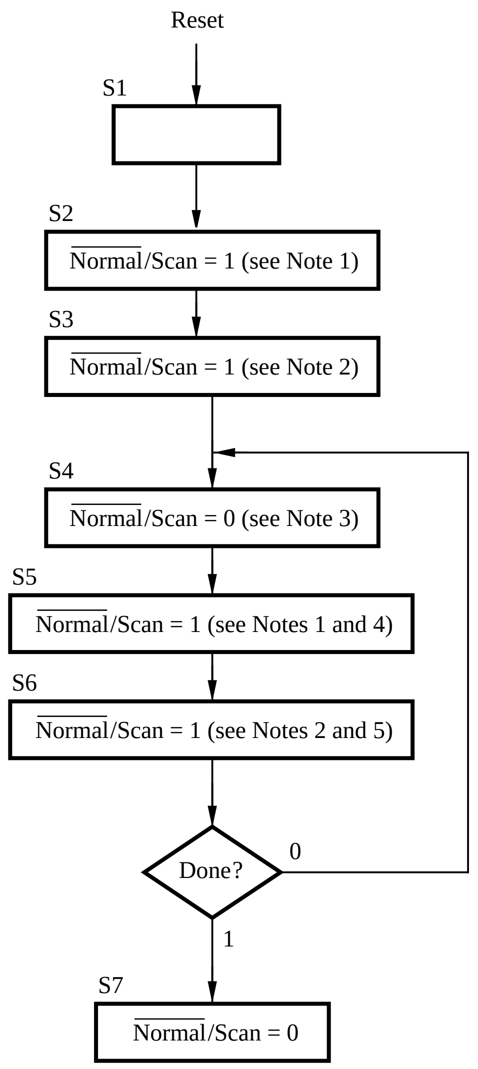
11.13. Test 1100 detects $w_1/0,\ w_2/0,\ b/1,\ c/0,\ g/0,\ k/1$, and $f/0$.
Test 0010 detects $w_4/1,\ b/0,\ c/0,\ g/1,\ h/0,\ k/0$, and $f/1$.
Test 0110 detects $w_1/1,\ w_4/1,\ b/0,\ c/0,\ g/1,\ h/0,\ k/0$, and $f/1$.

11.14. Label the output wires of the top three AND gates in Figure 11.12 as a, b, and c, respectively. Then the paths in the combinational part of the circuit are sensitized as follows.

Path $\overline{y}_1 - a - Y_1$	is sensitized with $w = 1$ and $y_2 = 0$.
Path $w - a - Y_1$	is sensitized with $y_1 = 0$ and $y_2 = 0$.
Path $w - b - Y_1$	is sensitized with $y_1 = 1$ and $y_2 = 1$.
Path $w - b - Y_2$	is sensitized with $y_1 = 0$ and $y_2 = 1$.
Path $y_2 - b - Y_1$	is sensitized with $w = 1$ and $y_1 = 1$.
Path $y_2 - b - Y_2$	is sensitized with $w = 1$ and $y_1 = 0$.
Path $w - c - Y_2$	is sensitized with $y_1 = 1$ and $y_2 = 0$.
Path $y_1 - c - Y_2$	is sensitized with $w = 1$ and $y_2 = 0$.
Path $y_1 = z$	is sensitized with $y_2 = 1$.
Path $y_2 = z$	is sensitized with $y_1 = 1$.

All 8 valuations of signals w, y_1 and y_2 have to be applied to sensitize these paths. It takes 26 clock cycles to perform the tests.

11.15. For simplicity, in the ASM chart it is assumed that testing begins one clock cycle after *Resetn* is de-asserted. States S2 to S6 depict the actions listed on page 666. We assume that external circuitry places the test data values on the *Scan-in* and *w* ports, and checks the generated results at *Scan-out* and *z*.


Reset
S1
S2
$\overline{\text{Normal}}$/Scan = 1 (see Note 1)
S3
$\overline{\text{Normal}}$/Scan = 1 (see Note 2)
S4
$\overline{\text{Normal}}$/Scan = 0 (see Note 3)
S5
$\overline{\text{Normal}}$/Scan = 1 (see Notes 1 and 4)
S6
$\overline{\text{Normal}}$/Scan = 1 (see Notes 2 and 5)
Done?
0
1
S7
$\overline{\text{Normal}}$/Scan = 0

Notes

Note 1: Scan-in has test value for y_2

Note 2: Scan-in has test value for y_1

Note 3: w has test value

Note 4: Scan-out has test result for y_2

Note 5: Scan-out has test result for y_1

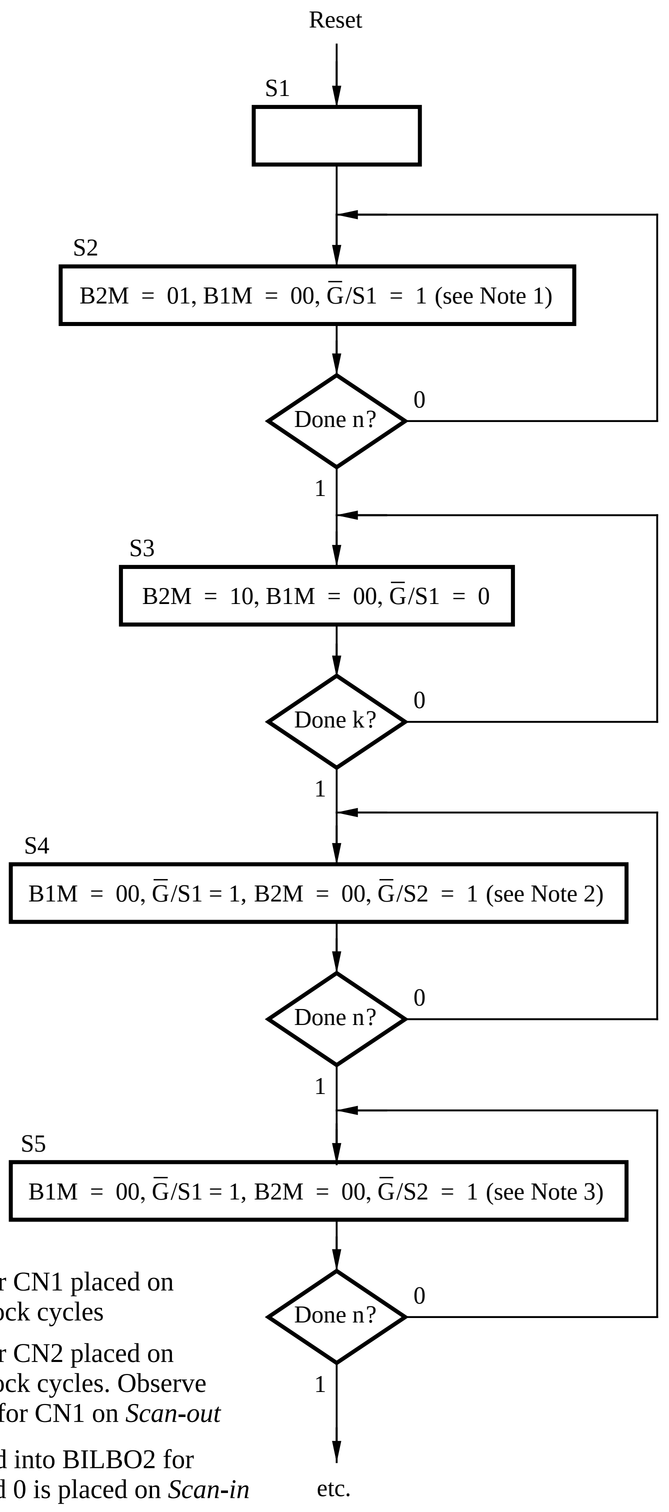
11.16. Assume that the circuit has been reset by applying *Resetn* = 0. Then, let *Resetn* = 1 and observe the behavior indicated in the following table.

Clock cycle	$\overline{\text{Normal}}$/ Scan	Scan-in	Scan-out	w	z	Transition tested
1	1	0	0	x	x	Reset
2	1	0	0	x	x	
3	0	x	x	0	0	A $\rightarrow$ A
4	1	0	0	x	x	
5	1	1	0	x	x	
6	0	x	x	0	0	B $\rightarrow$ A
7	1	0	0	x	x	
8	1	0	0	x	x	
9	0	x	x	1	0	A $\rightarrow$ B
10	1	0	0	x	x	
11	1	1	1	x	x	
12	0	x	x	1	0	B $\rightarrow$ C
13	1	1	1	x	x	
14	1	0	0	x	x	
15	0	x	x	0	0	C $\rightarrow$ A
16	1	1	0	x	x	
17	1	0	0	x	x	
18	0	x	x	1	0	C $\rightarrow$ D
19	1	1	1	x	x	
20	1	1	1	x	x	
21	0	x	x	0	1	D $\rightarrow$ A
22	1	1	0	x	x	
23	1	1	0	x	x	
24	0	x	x	1	1	D $\rightarrow$ D
25	1	x	1	x	x	
26	1	x	1	x	x	

11.17. The Verilog code for Figure 11.12 is

```
module prob11_17 (w, scanin, norm_scan, z, scanout, Resetn, Clock);
   input w, scanin, norm_scan, Resetn, Clock;
   output z, scanout;
   reg z, scanout;
   reg [2:1] y, Y, D;

   // Define the combinational circuitry
   always @(w or y or scanin or norm_scan)
   begin
      Y[1] = (w & ~y[1]) | (w & y[2]);
      Y[2] = (w & y[2]) | (w & y[1]);
      z = y[1] & y[2];
      if (norm_scan == 0)
      begin
         D[1] = Y[1];
         D[2] = Y[2];
      end
      else
      begin
         D[1] = scanin;
         D[2] = y[1];
      end
      scanout = y[2];
   end

   // Define the flip-flops
   always @(negedge Resetn or posedge Clock)
      if (Resetn == 0)  y <= 0;
      else  y <= D;

endmodule
```

11.18. For simplicity, it is assumed in the ASM chart that testing begins when the reset signal is de-asserted. The ASM chart corresponds to the first three steps listed on page 675; the other steps are similar and are not shown. In the ASM chart, B1M represents the two-bit signal M_1M_2 for BILBO1 and B2M represents M_1M_2 for BILBO2. Similarly, $\overline{G}/S1$ and $\overline{G}/S2$ represent the $\overline{G}/S$ signals for BILBO1 and BILBO2. Assume that there are n flip-flops in each BILBO register and that k clock cycles are used when running each BILBO circuit as a PRBS generator.


Reset
S1
S2
B2M = 01, B1M = 00, $\overline{G}$/S1 = 1 (see Note 1)
Done n?
0
1
S3
B2M = 10, B1M = 00, $\overline{G}$/S1 = 0
Done k?
0
1
S4
B1M = 00, $\overline{G}$/S1 = 1, B2M = 00, $\overline{G}$/S2 = 1 (see Note 2)
Done n?
0
1
S5
B1M = 00, $\overline{G}$/S1 = 1, B2M = 00, $\overline{G}$/S2 = 1 (see Note 3)
Done n?
0
1
etc.

Notes

Note 1: Initial test data for CN1 placed on *Scan-in* over *n* clock cycles

Note 2: Initial test data for CN2 placed on *Scan-in* over *n* clock cycles. Observe generated results for CN1 on *Scan-out*

Note 3: BILBO1 is shifted into BILBO2 for *n* clock cycles and 0 is placed on *Scan-in*