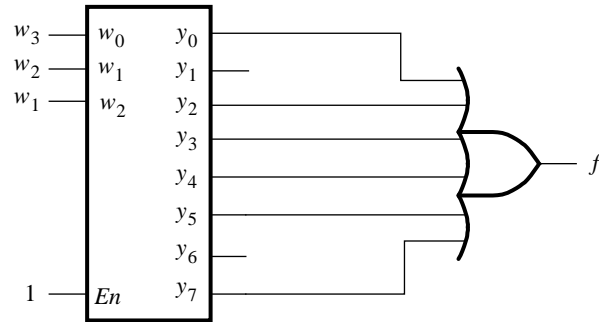
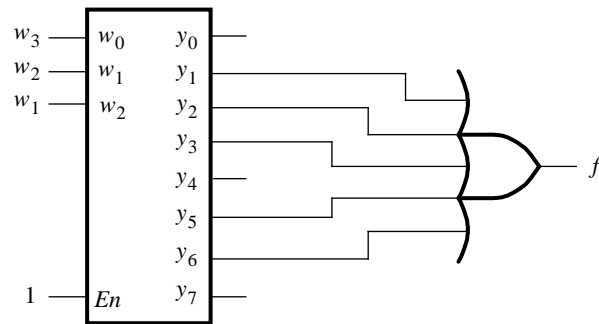


Chapter 6

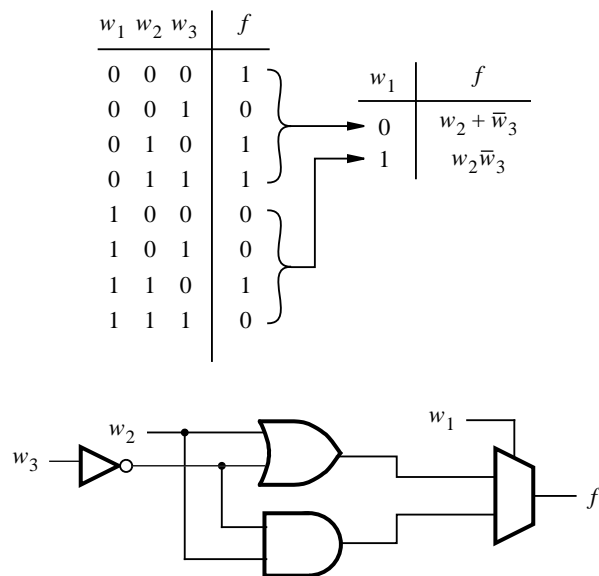
6.1.



6.2.



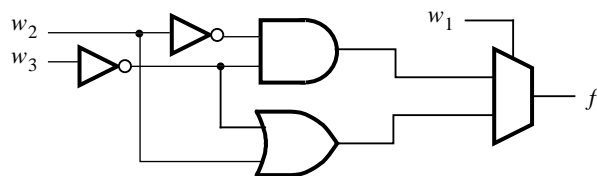
6.3.



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	$\bar{w}_2\bar{w}_3$
1	$w_2 + \bar{w}_3$



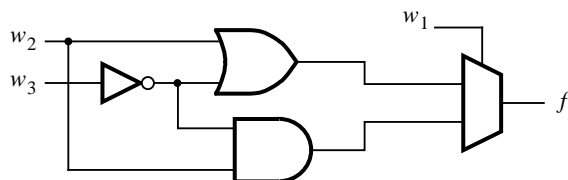
6.5. The function f can be expressed as

$$f = \bar{w}_1\bar{w}_2\bar{w}_3 + \bar{w}_1w_2\bar{w}_3 + \bar{w}_1w_2w_3 + w_1w_2\bar{w}_3$$

Expansion in terms of w_1 produces

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

The corresponding circuit is



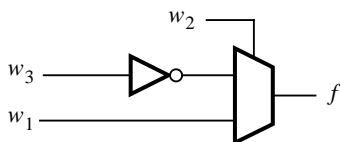
6.6. The function f can be expressed as

$$f = \bar{w}_1\bar{w}_2\bar{w}_3 + w_1\bar{w}_2\bar{w}_3 + w_1w_2\bar{w}_3 + w_1w_2w_3$$

Expansion in terms of w_2 produces

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

The corresponding circuit is



6.7. Expansion in terms of w_2 gives

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

Further expansion in terms of w_1 gives

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

Further expansion in terms of w_3 gives

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

6.8. Expansion in terms of w_1 gives

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

Further expansion in terms of w_2 gives

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

Further expansion in terms of w_3 gives

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

6.9. Proof of Shannon's expansion theorem

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

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:

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

Setting $x_1 = 1$, we have:

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

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

6.10. Derivation using \bar{f} :

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

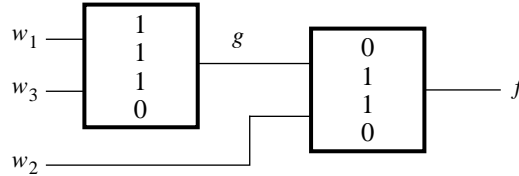
6.11. Expansion in terms of w_2 gives

$$f = \overline{w_2}(\overline{w_1} + \overline{w_3}) + w_2(w_1w_3)$$

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

$$\begin{aligned} f &= \overline{w_2}(\overline{w_1} + \overline{w_3}) + w_2(w_1w_3) \\ &= w_2 \oplus (\overline{w_1} + \overline{w_3}) \\ &= w_2 \oplus \overline{w_1w_3} \end{aligned}$$

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

$$\begin{aligned} f &= \overline{w_2}(\overline{w_3} + \overline{w_1}w_4) + w_2(w_3\overline{w_4} + w_1w_3) \\ &= \overline{w_2}(\overline{w_3} + \overline{w_1}w_4) + w_2(w_3(w_1 + \overline{w_4})) \end{aligned}$$

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 .

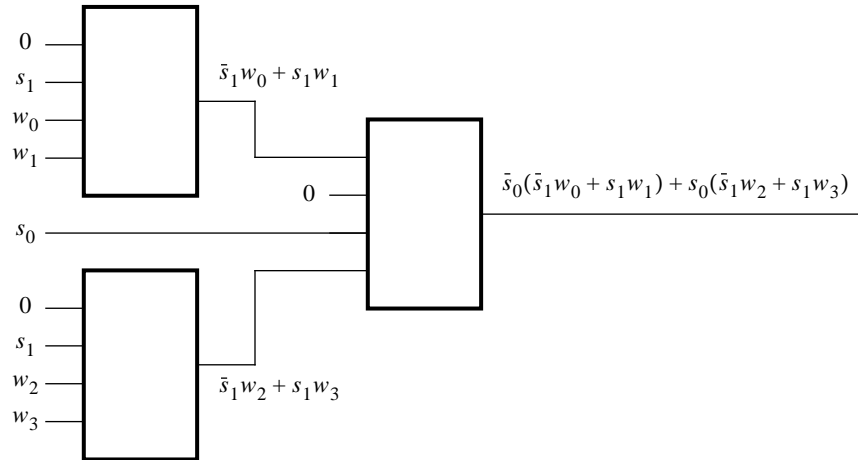
6.14. Any number of 5-variable functions can be implemented by using two 4-LUTs. For example, if we cascade the two 4-LUTs by connecting the output of one 4-LUT to an input of the other, then we can realize any function of the form

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

6.15. Expressing f in the form

$$\begin{aligned} f &= \bar{s}_1 \bar{s}_0 w_0 + s_1 \bar{s}_0 w_1 + \bar{s}_1 s_0 w_2 + s_1 s_0 w_3 \\ &= \bar{s}_0 (\bar{s}_1 w_0 + s_1 w_1) + s_0 (\bar{s}_1 w_2 + s_1 w_3) \end{aligned}$$

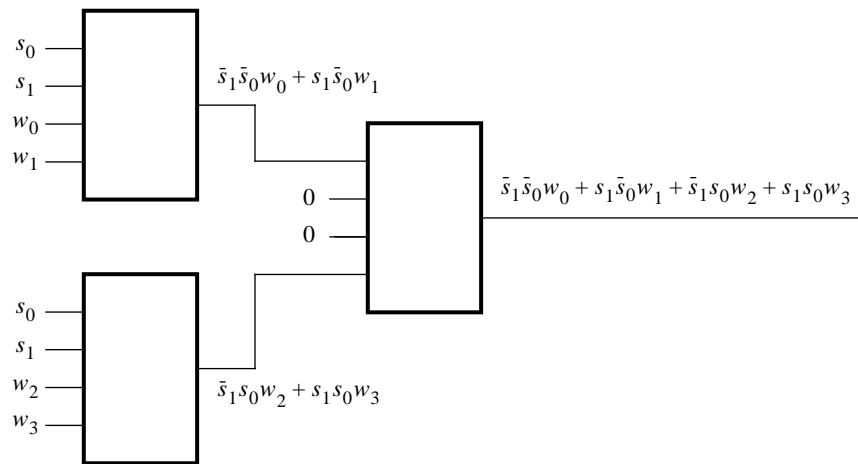
leads to the circuit.



Alternatively, directly using the expression

$$f = \bar{s}_1 \bar{s}_0 w_0 + s_1 \bar{s}_0 w_1 + \bar{s}_1 s_0 w_2 + s_1 s_0 w_3$$

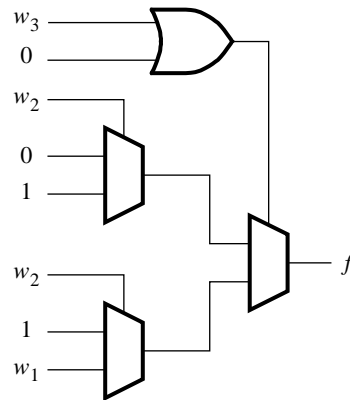
leads to the circuit.



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

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

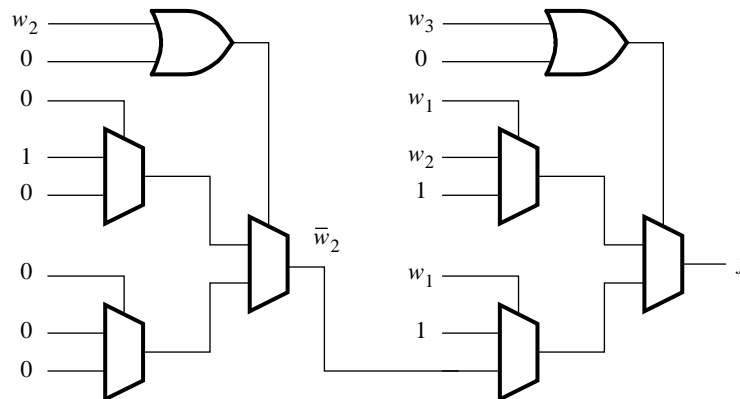
The corresponding circuit is



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

$$f = w_3(\overline{w_1} + w_1\overline{w_2}) + \overline{w_3}(w_1 + \overline{w_1}w_2)$$

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 following 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
    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

```

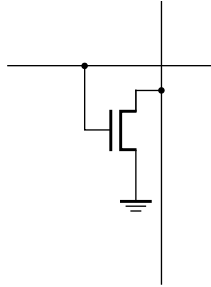
6.29.

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

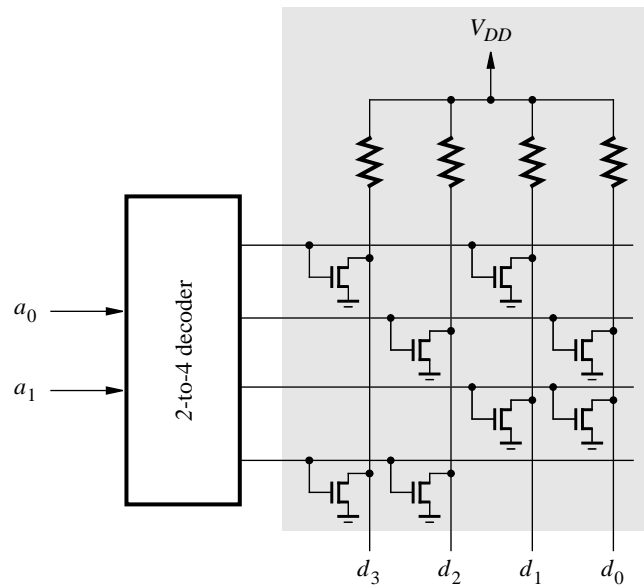
6.30.

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

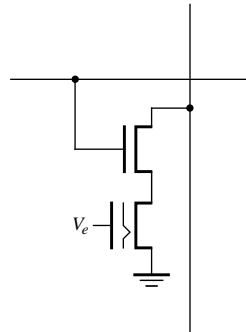
- 6.31. (a) Each ROM location that should store a 1 requires no circuitry, because the pull-up resistor provides the default value of 1. Each location that stores a 0 has the following cell



(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)

