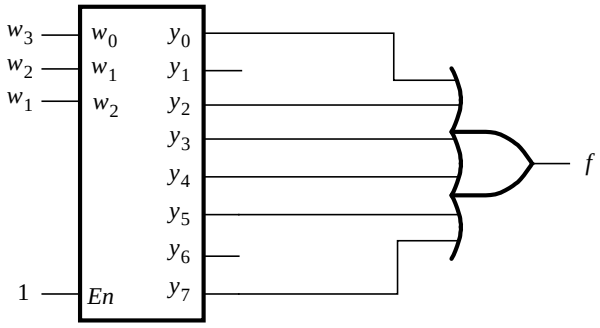
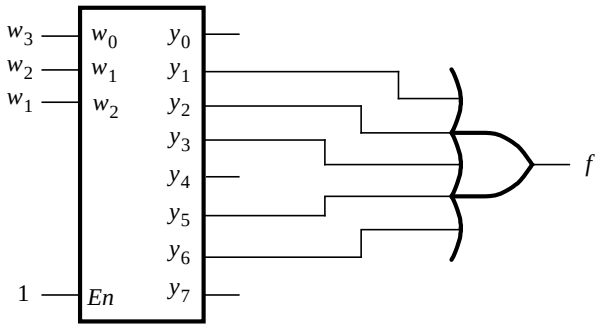
Chapter 4

4.1.



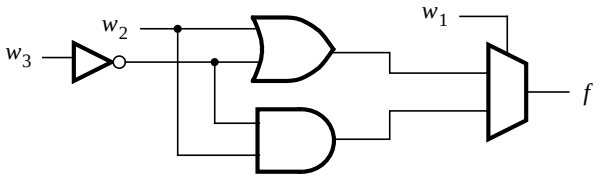
4.2.



4.3.

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

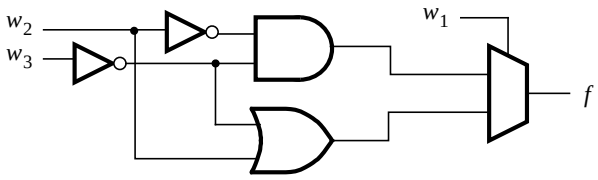
w_1	f
0	$w_2 + \overline{w}_3$
1	$w_2\overline{w}_3$



4.4.

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	$w_2 + \overline{w}_3$



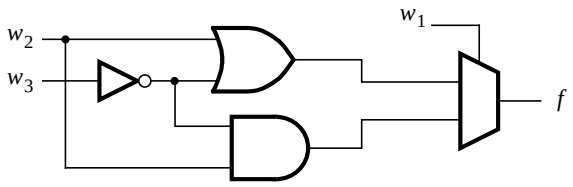
4.5. The function f can be expressed as

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

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



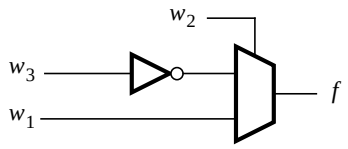
4.6. The function f can be expressed as

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

Expansion in terms of w_2 produces

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

The corresponding circuit is



4.7. 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

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

4.8. 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

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

4.9. 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.

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

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

4.11. Expansion of f in terms of w_2 gives

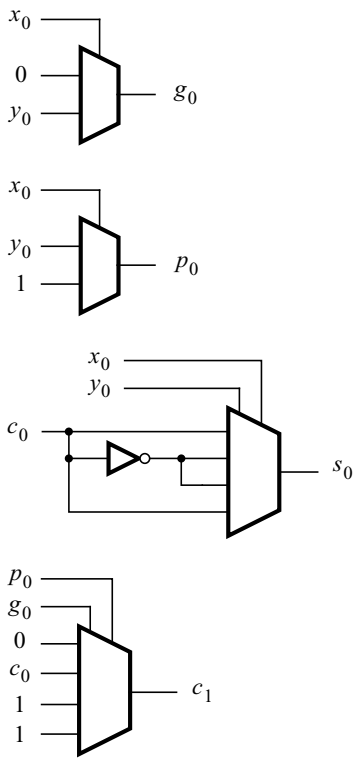
$$\begin{aligned} f &= \overline{w}_2(\overline{w}_1 + \overline{w}_3) + w_2(w_1 w_3) \\ &= w_2 \oplus (\overline{w}_1 + \overline{w}_3) \\ &= w_2 \oplus \overline{w_1 w_3} \end{aligned}$$

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

4.12. The four functions that have to be implemented are:

$$\begin{aligned} g_0 &= x_0 y_0 \\ p_0 &= x_0 \oplus y_0 \\ s_0 &= p_0 \oplus c_0 \\ c_1 &= g_0 + p_0 c_0 \end{aligned}$$

Using multiplexers, these functions can be implemented as follows:



4.13.

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

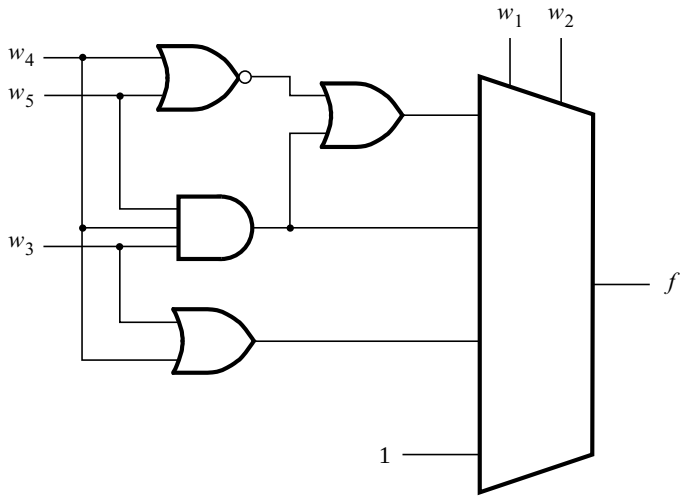
4.14.

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

4.15. Shannon's expansion with respect to w_1 and w_2 gives

$$\begin{aligned} f &= \overline{w}_1\overline{w}_2\overline{w}_4\overline{w}_5 + w_1w_2 + w_1w_3 + w_1w_4 + w_3w_4w_5 \\ &= \overline{w}_1\overline{w}_2 f_{\overline{w}_1\overline{w}_2} + \overline{w}_1 w_2 f_{\overline{w}_1 w_2} + w_1\overline{w}_2 f_{w_1\overline{w}_2} + w_1w_2 f_{w_1w_2} \\ &= \overline{w}_1\overline{w}_2(\overline{w}_4\overline{w}_5 + w_3w_4w_5) + \overline{w}_1 w_2(w_3w_4w_5) + w_1\overline{w}_2(w_3 + w_4) + w_1w_2(1) \end{aligned}$$

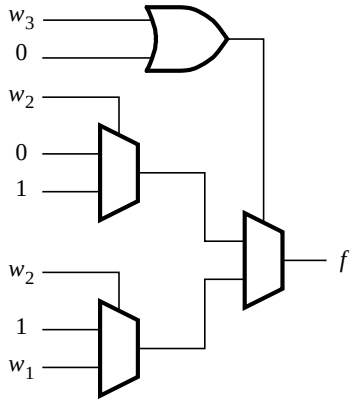
Since only uncomplemented inputs are available, the term $\overline{w}_4\overline{w}_5$ has to be implemented as $\overline{w_4 + w_5}$. The resulting circuit is



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

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

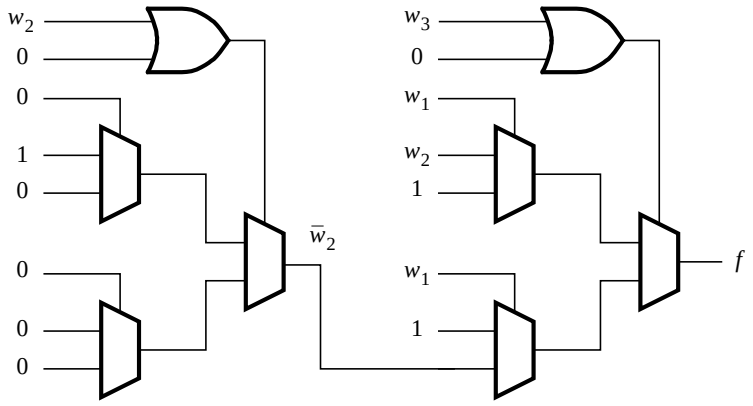
The corresponding circuit is



4.17. 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



4.18. The code in Figure P4.2 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.

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

```
module  prob4_19 (W, f);
   input  [1:3] W;
   output  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
```

4.20. Using the truth table in Figure 4.19*a*, the 4-to-2 binary encoder can be implemented as:

```
module  prob4_20 (W, Y);
   input  [3:0] W;
   output  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
```

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

```
module  prob4_21 (W, Y);
   input  [7:0] W;
   output  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
```

4.22. The code in Figure P4.3 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.

4.23. 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

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

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

```
module  prob4_24 (W, Y, z);
   input  [7:0] W;
   output  reg  [2:0] Y;
   output  reg  z;

   always @(W)
   begin
      z = 1;
      casex (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
```

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

```
module prob4_25 (W, Y, z);
   input [7:0] W;
   output reg [2:0] Y;
   output 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
```

4.26. The following code can be used:

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

    always @(W, 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  reg  [0:3] Y;

    always @(W, 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
```

4.27. 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  wire  [0:63] Y;
   reg   [7:0] En3to8dec;

   always @(W, 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 4.26.
```

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

    always @(W, 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  reg  [0:3] Y;

    always @(W, 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
```

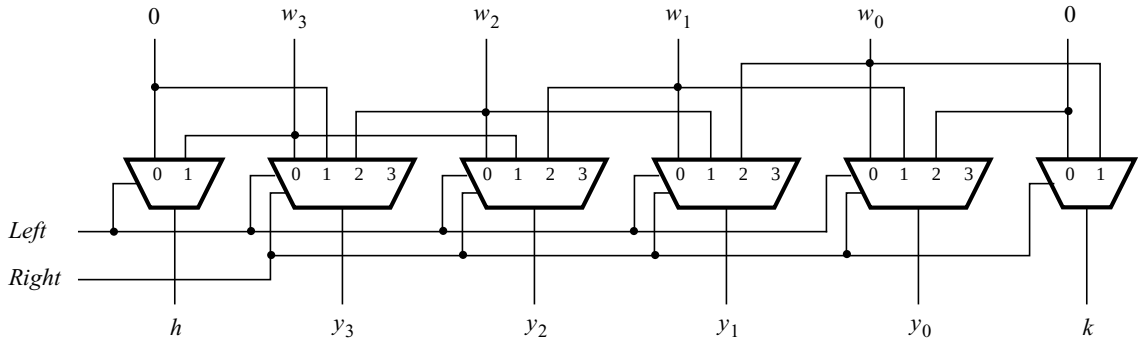
4.28. A possible code is:

```
module prob4_28 (W, S, f);
   input [0:3] W;
   input [1:0] S;
   output 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 reg [0:3] Y;

   always @(W, 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
```

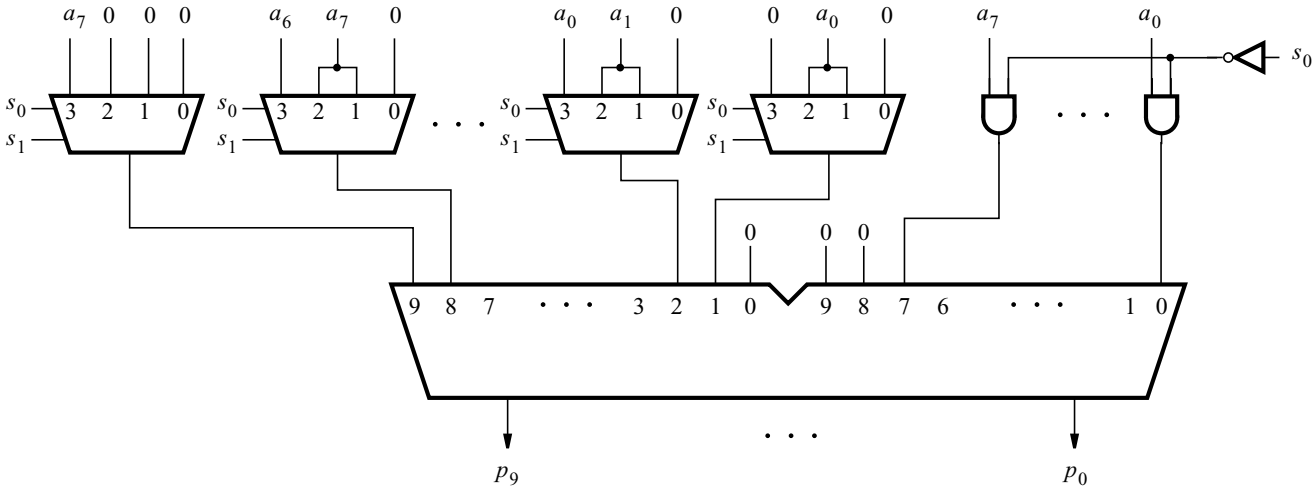
4.29. Using an arrangement similar to Figure 4.50, the desired circuit can be specified by the following truth table:

Left	*Right*	h	y_3	y_2	y_1	y_0	k
0	0	0	w_3	w_2	w_1	w_0	0
0	1	0	w_0	w_3	w_2	w_1	w_0
1	0	w_3	w_2	w_1	w_0	w_3	0

Using multiplexers, this truth table may be realized as



4.30. Let the multiplexer select inputs s_1 and s_0 represent the desired multiplication such that $s_1 s_0 = 00, 01, 10, 11$ specifies the multiplication by 1, 2, 3 and 4, respectively. Then, the required products can be generated by the following circuit.



4.32. The desired circuit is defined by the following expressions:

$$\begin{aligned} b_2 &= g_2 \\ b_1 &= g_1 \oplus g_2 \\ b_2 &= g_0 \oplus g_1 \oplus g_2 \end{aligned}$$

4.33. A possible Verilog code is

```
module  parity (X, Y, error);
   input     [7:0] Y;
   output   [7:0] X;

   assign  X = {1'b0, Y[6:0]};

   always  @(*)
   if  (^Y[6:0] == Y[7])  error = 0;
   else  error = 1;
endmodule
```