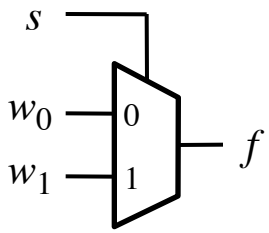
Chapter 4

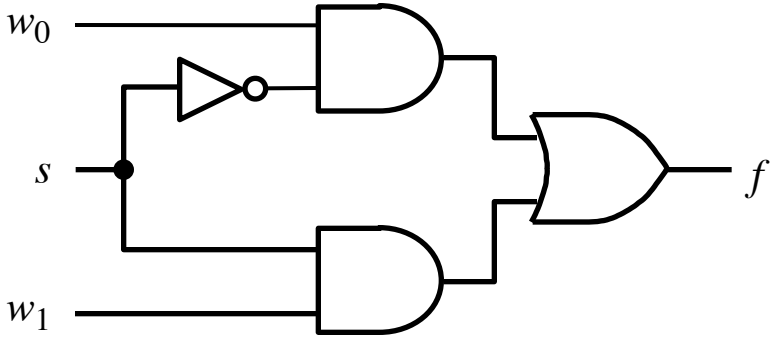
Combinational-Circuit Building Blocks



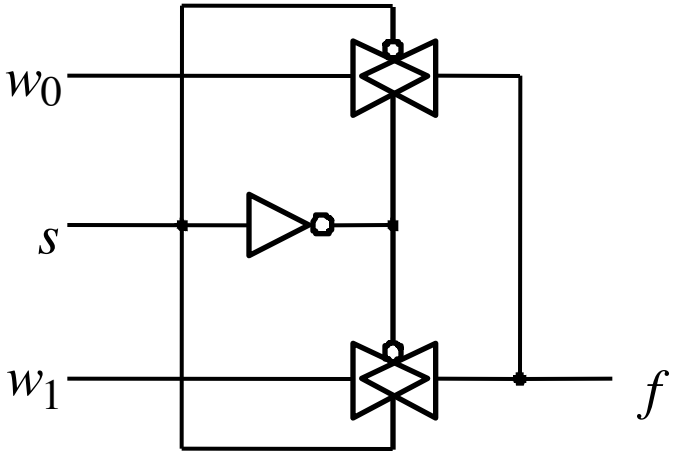
(a) Graphical symbol

s	f
0	w_0
1	w_1

(b) Truth table

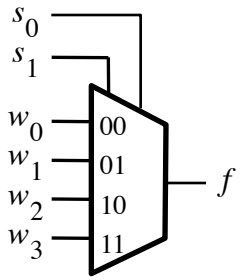


(c) Sum-of-products circuit



(d) Circuit with transmission gates

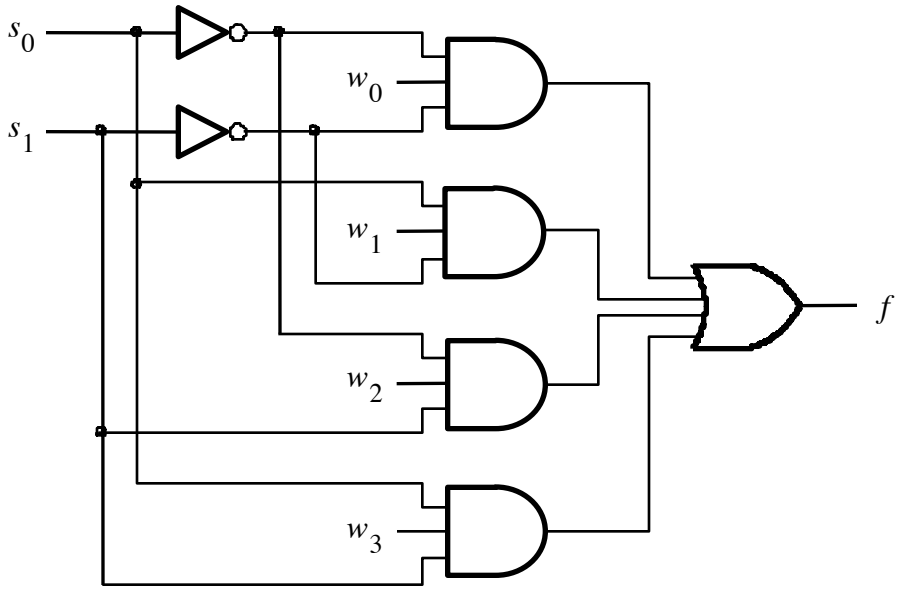
Figure 4.1. A 2-to-1 multiplexer.



(a) Graphic symbol

s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

(b) Truth table



(c) Circuit

Figure 4.2. A 4-to-1 multiplexer.

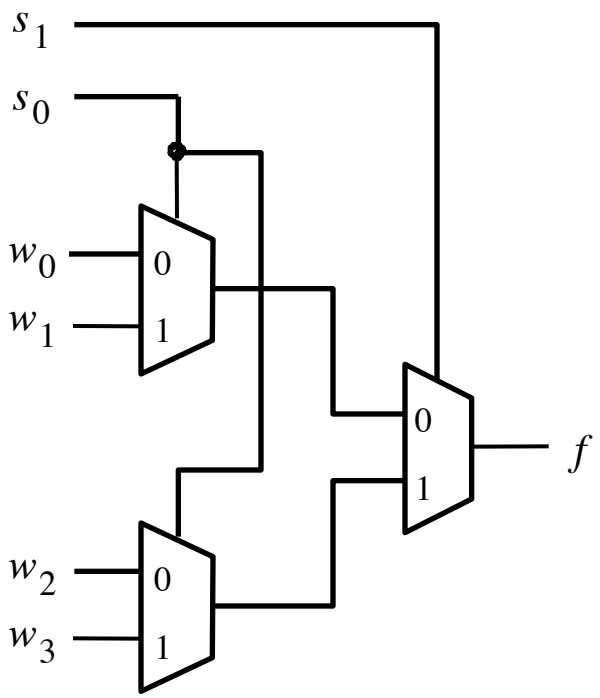


Figure 4.3. Using 2-to-1 multiplexers to build a 4-to-1 multiplexer.

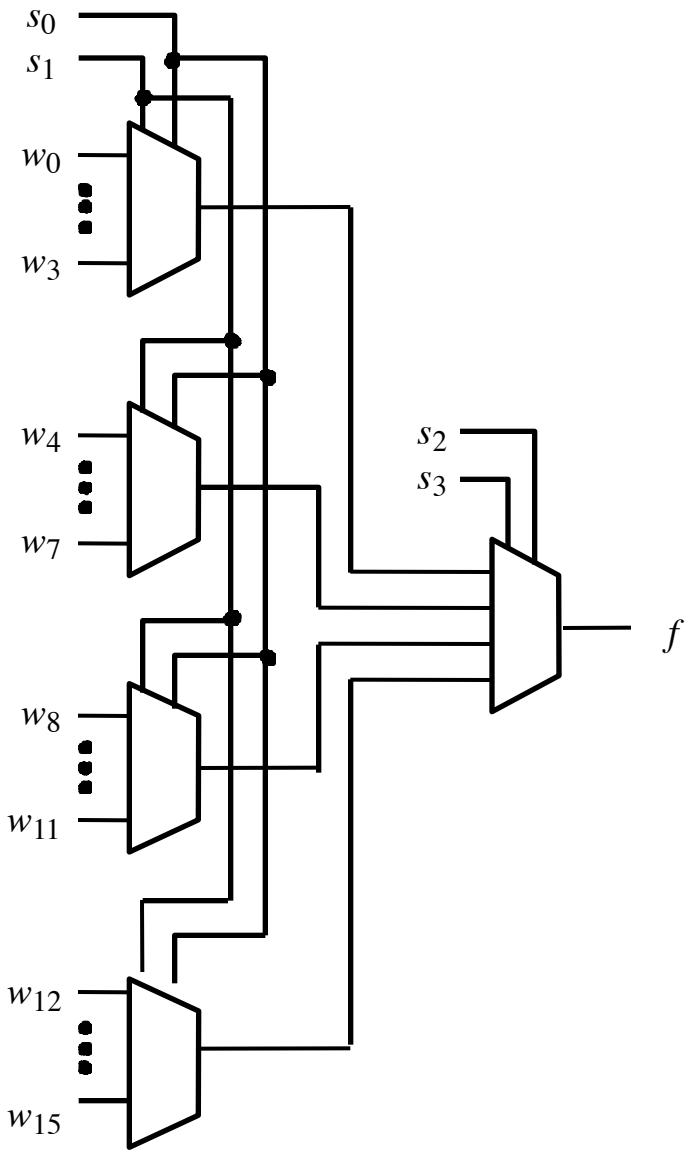
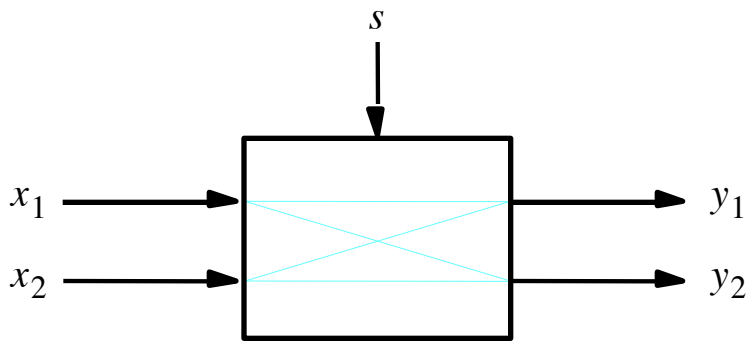
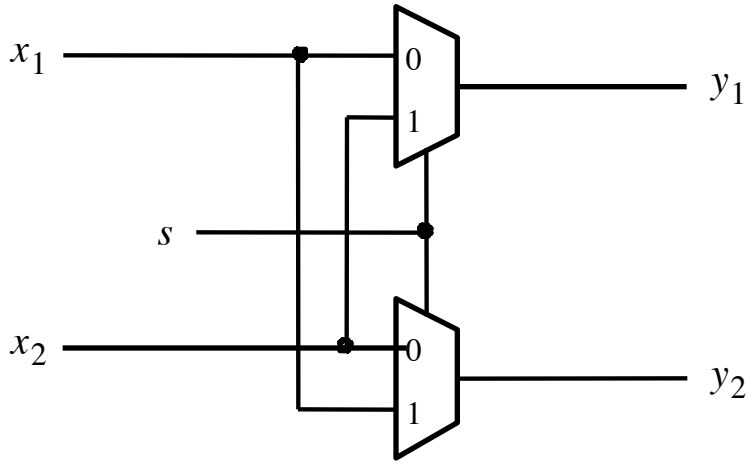


Figure 4.4. A 16-to-1 multiplexer.



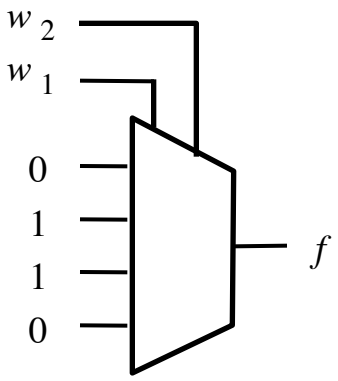
(a) A 2x2 crossbar switch



(b) Implementation using multiplexers

Figure 4.5. A practical application of multiplexers.

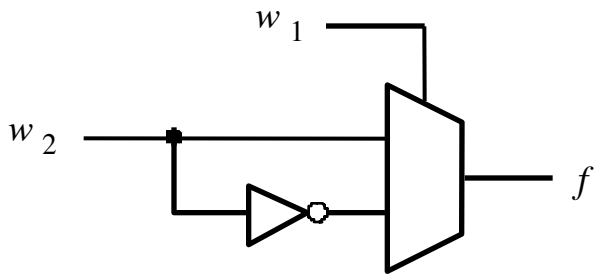
w_1	w_2	f
0	0	0
0	1	1
1	0	1
1	1	0



(a) Implementation using a 4-to-1 multiplexer

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

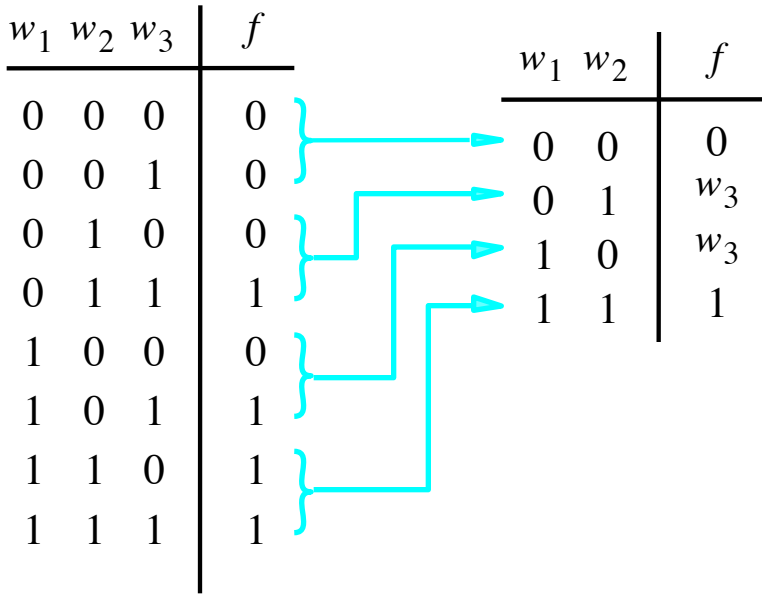
w_1	f
0	w_2
1	$\overline{w}_2$



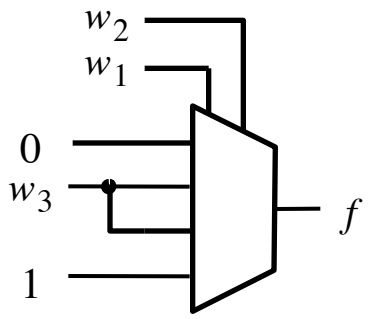
(b) Modified truth table

(c) Circuit

Figure 4.6. Synthesis of a logic function using multiplexers.



(a) Modified truth table

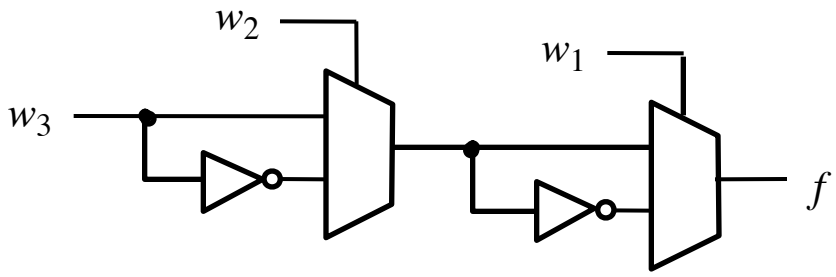


(b) Circuit

Figure 4.7. Implementation of the three-input majority function using a 4-to-1 multiplexer.

w_1	w_2	w_3	f	
0	0	0	0	
0	0	1	1	$w_2 \oplus w_3$
0	1	0	1	
0	1	1	0	
1	0	0	1	
1	0	1	0	$\overline{w_2 \oplus w_3}$
1	1	0	0	
1	1	1	1	

(a) Truth table

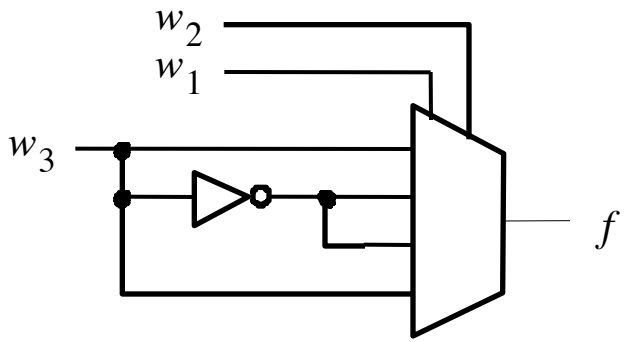


(b) Circuit

Figure 4.8. Three-input XOR implemented with 2-to-1 multiplexers.

w_1	w_2	w_3	f	
0	0	0	0	w_3
0	0	1	1	
0	1	0	1	$\bar{w}_3$
0	1	1	0	
1	0	0	1	$\bar{w}_3$
1	0	1	0	
1	1	0	0	w_3
1	1	1	1	

(a) Truth table



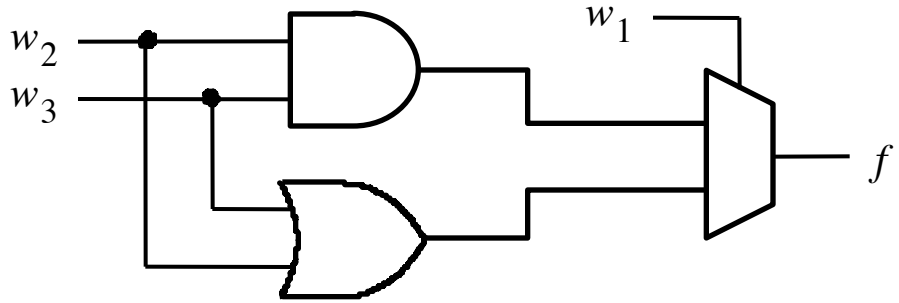
(b) Circuit

Figure 4.9. Three-input XOR function implemented with a 4-to-1 multiplexer.

w_1	w_2	w_3	f
0	0	0	0
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	1

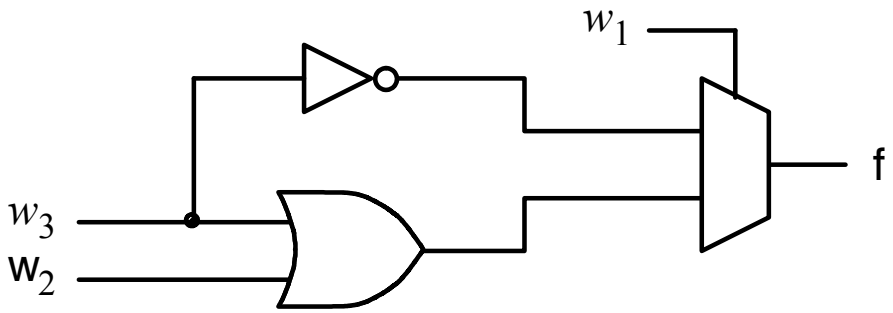
w_1	f
0	w_2w_3
1	$w_2 + w_3$

(b) Truth table

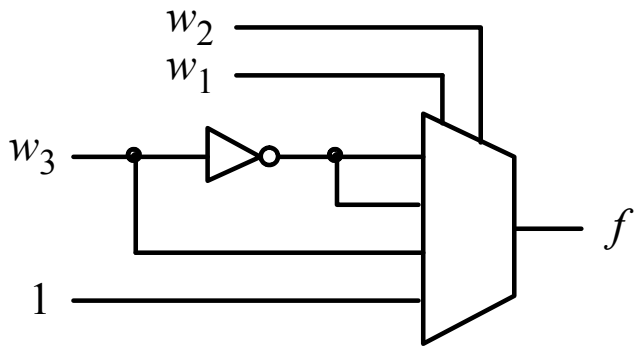


(b) Circuit

Figure 4.10. The three-input majority function implemented using a 2-to-1 multiplexer.



(a) Using a 2-to-1 multiplexer



(b) Using a 4-to-1 multiplexer

Figure 4.11. The circuits synthesized in Example 4.5.

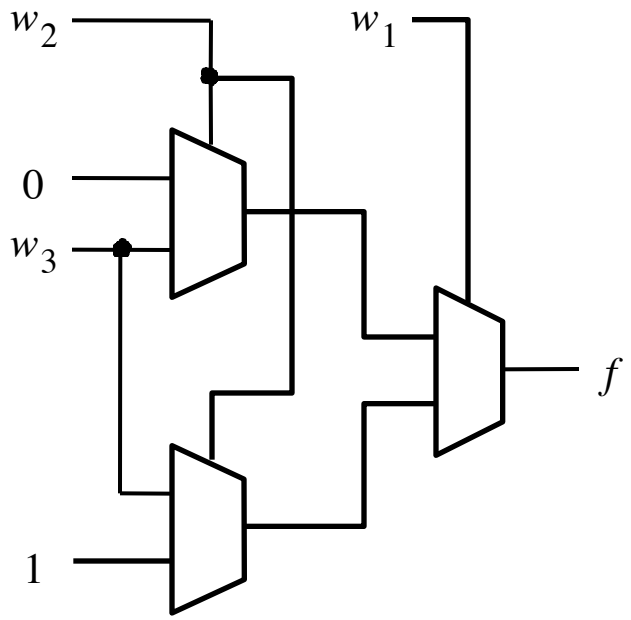
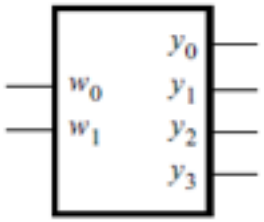


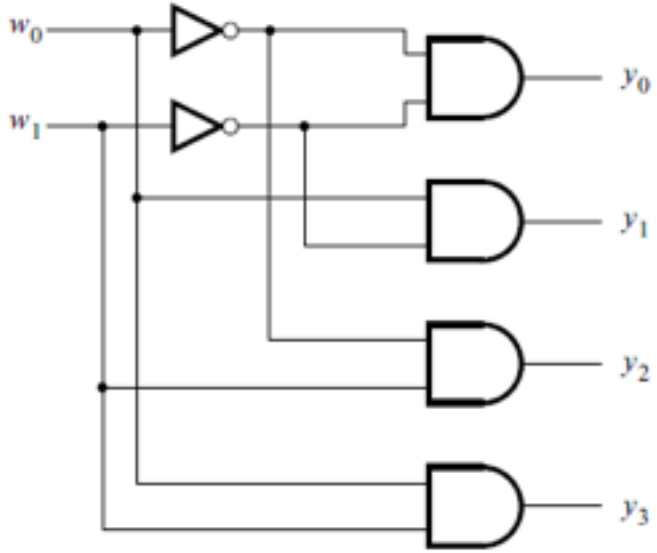
Figure 4.12. The circuit synthesized in Example 4.6.

w_1	w_0	y_0	y_1	y_2	y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

(a) Truth table



(b) Graphical symbol

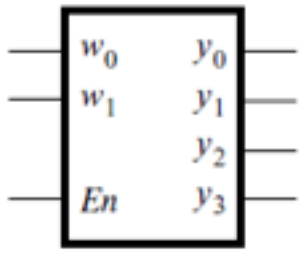


(c) Logic circuit

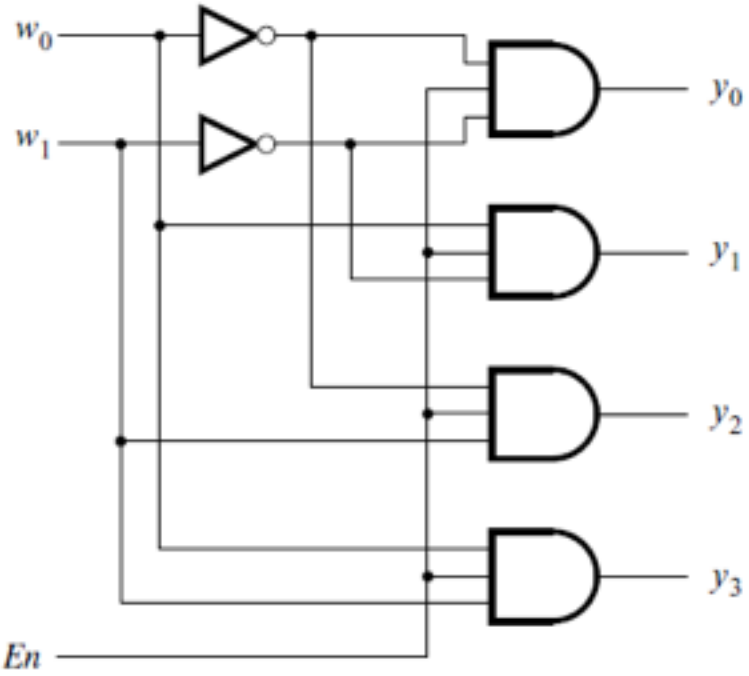
Figure 4.13. A 2-to-4 decoder.

En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

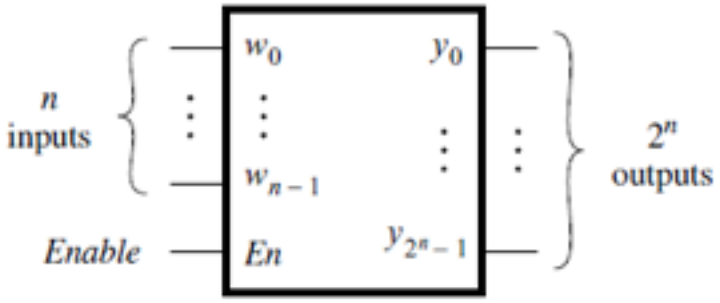
(a) Truth table



(b) Graphical symbol



(c) Logic circuit



(d) An n-to-2^n decoder

Figure 4.14. Binary decoder.

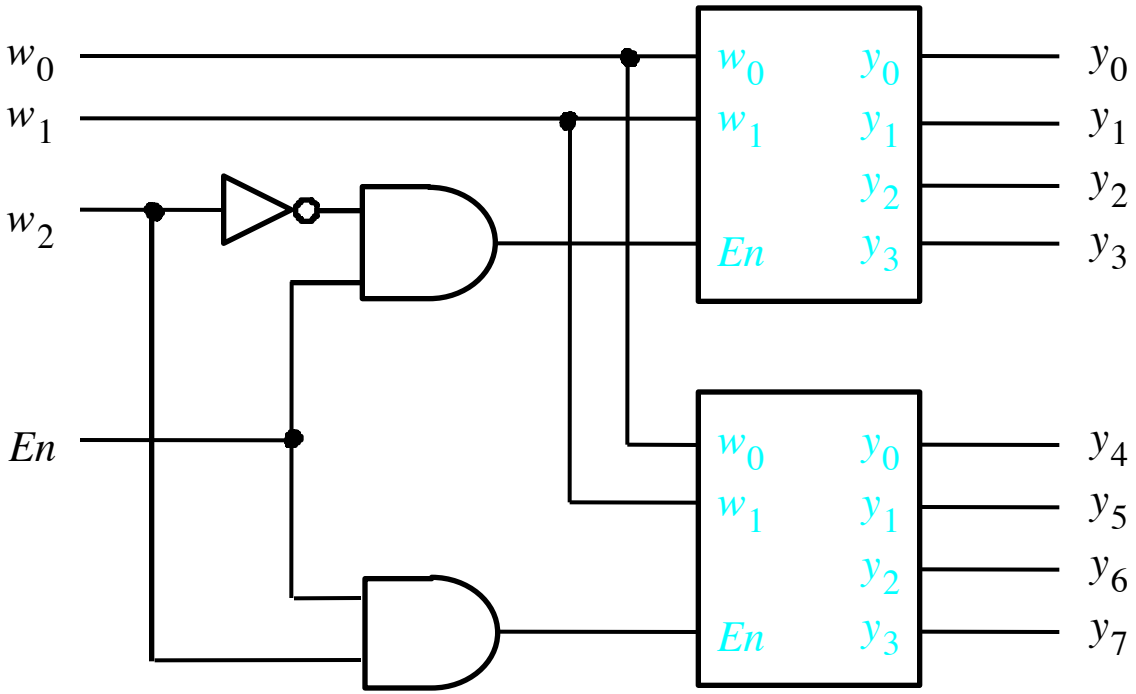


Figure 4.15. A 3-to-8 decoder using two 2-to-4 decoders.

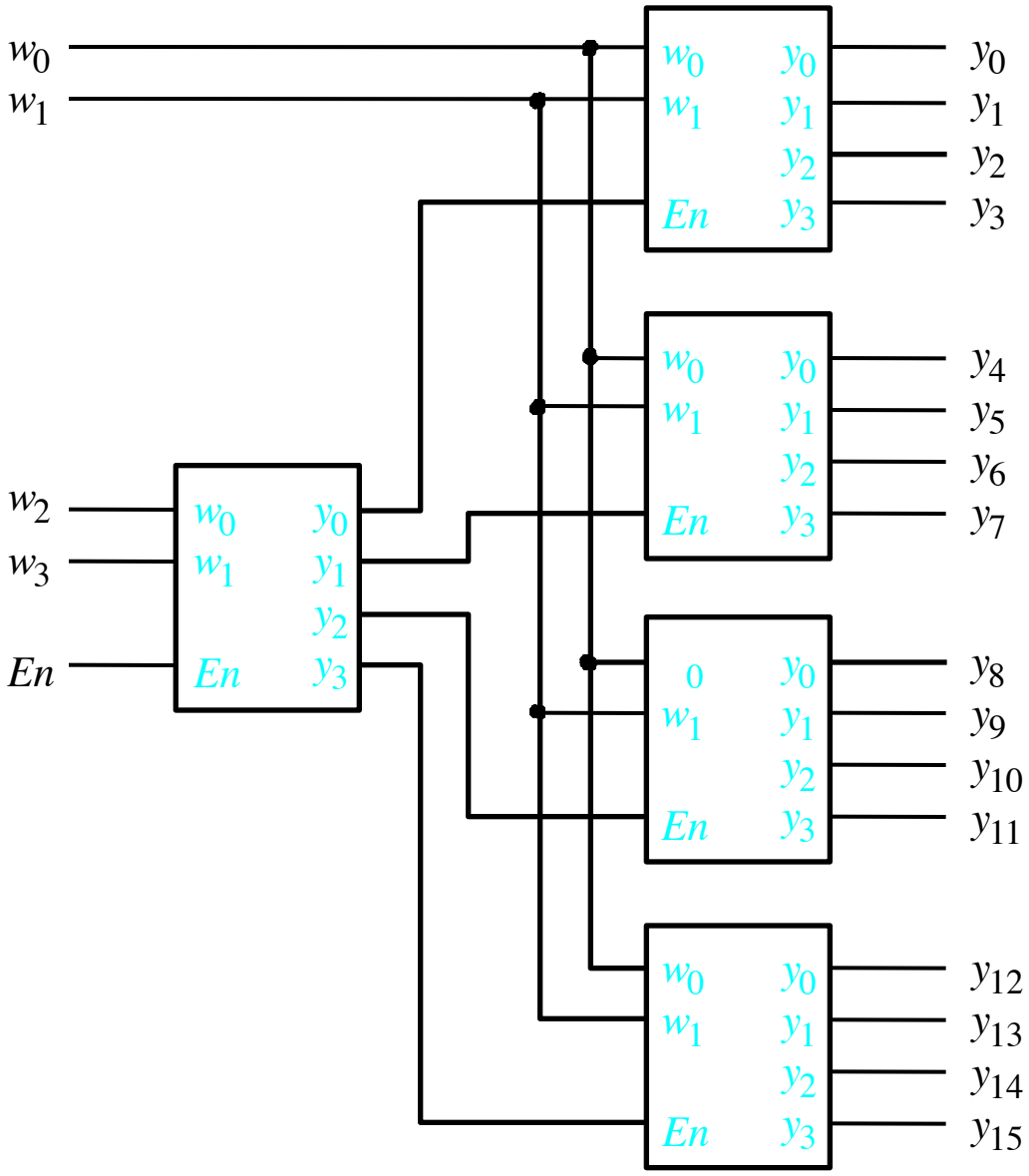


Figure 4.16. A 4-to-16 decoder built using a decoder tree.

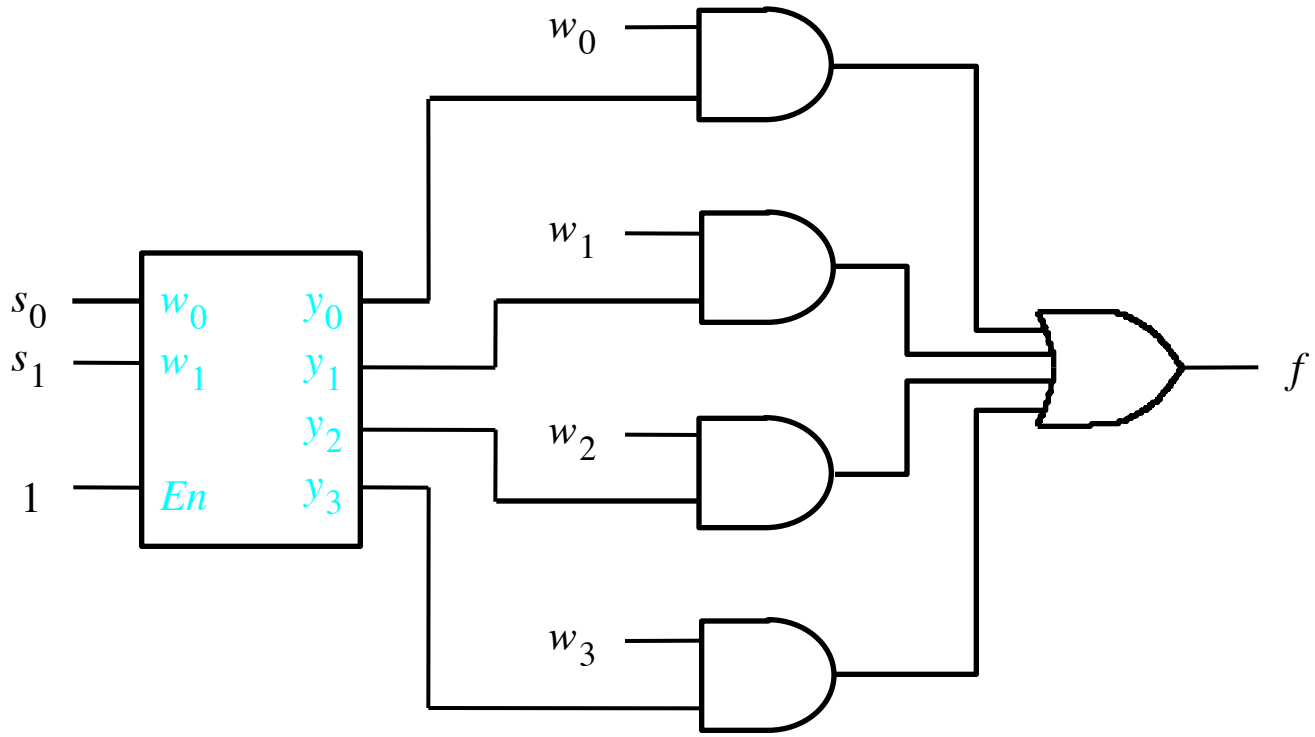


Figure 4.17. A 4-to-1 multiplexer built using a decoder.

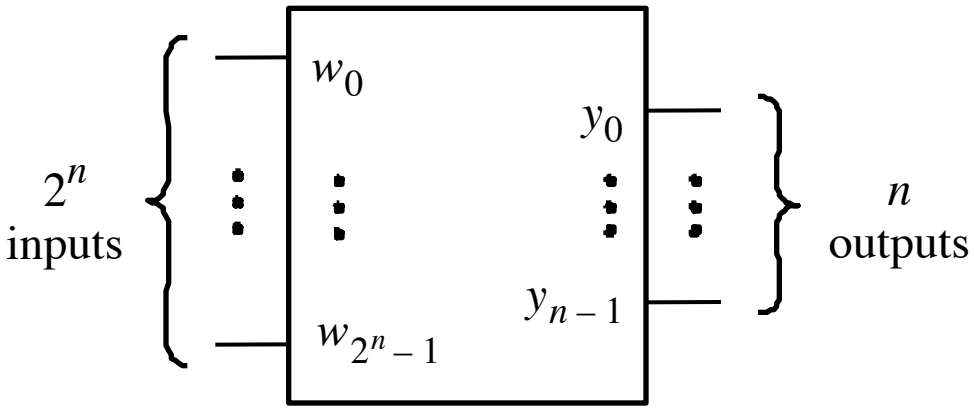
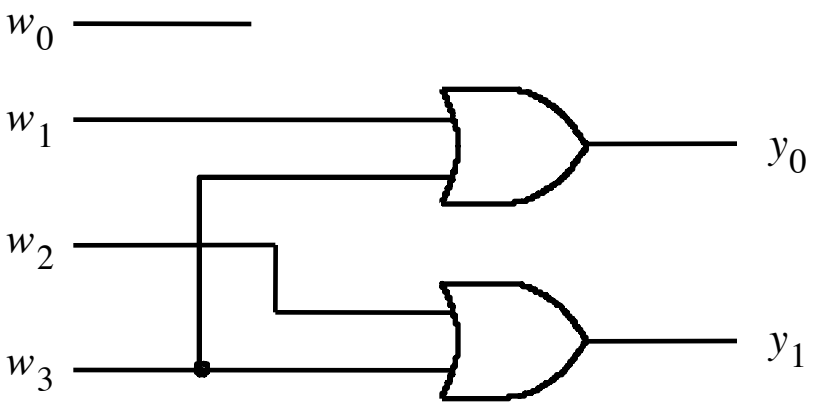


Figure 4.18. A 2^n-to-n binary encoder.

w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

(a) Truth table

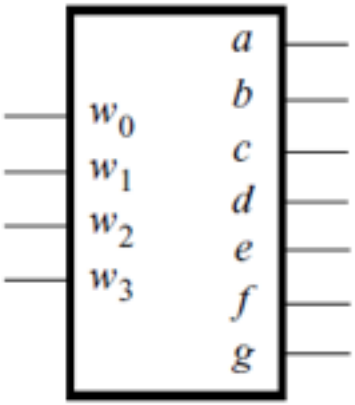


(b) Circuit

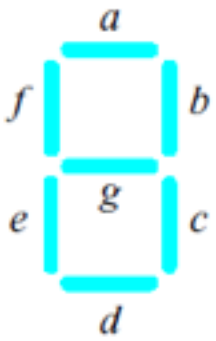
Figure 4.19. A 4-to-2 binary encoder.

w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

Figure 4.20. Truth table for a 4-to-2 priority encoder.



(a) Code converter



(b) 7-segment display

w_3	w_2	w_1	w_0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	1	1	1	0	1	1	1
1	0	1	1	0	0	1	1	1	1	1
1	1	0	0	1	0	0	1	1	1	0
1	1	0	1	0	1	1	1	1	0	1
1	1	1	0	1	0	0	1	1	1	1
1	1	1	1	1	0	0	0	1	1	1

(c) Truth table

Figure 4.21. A hex-to-7-segment display code converter.

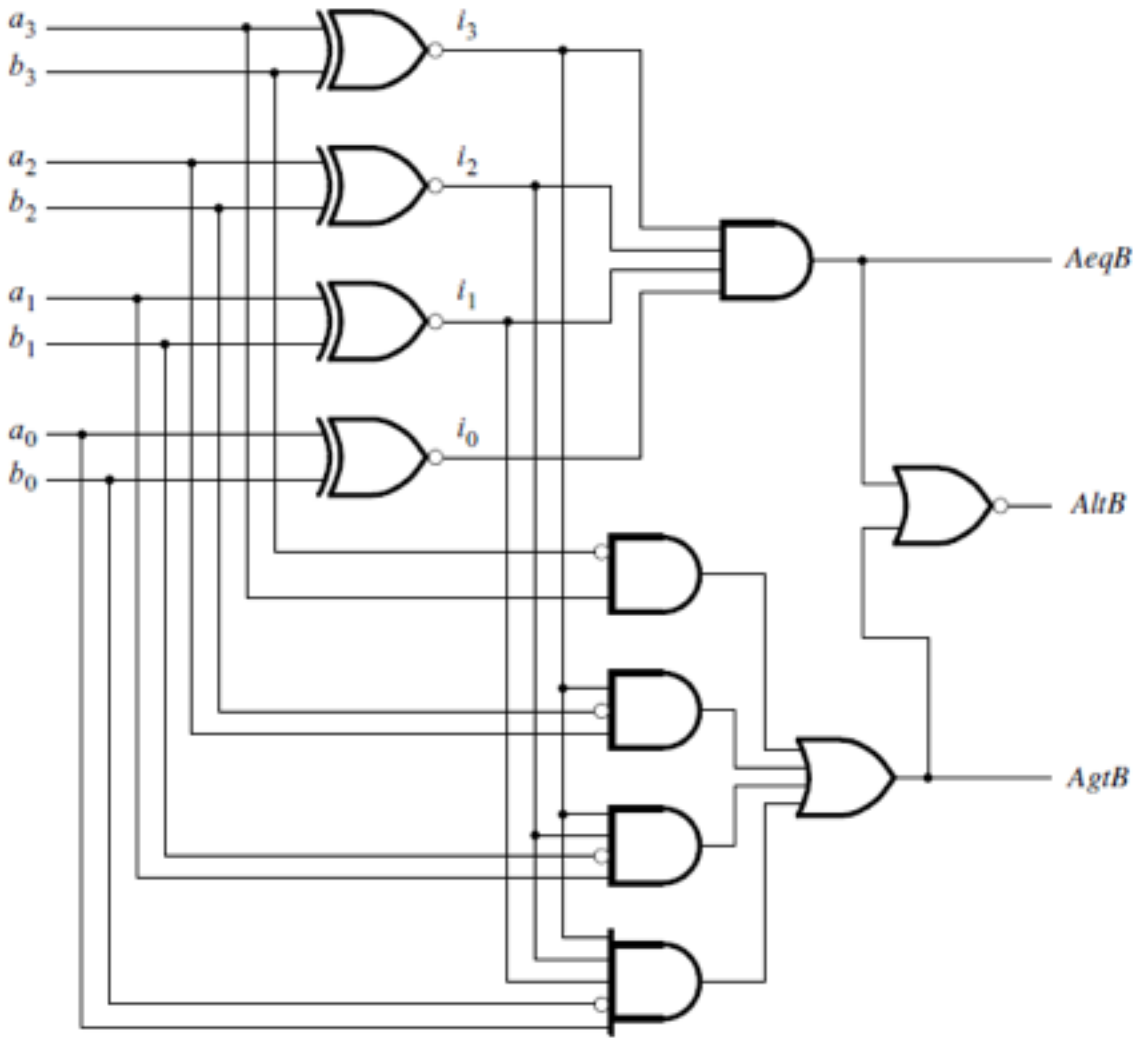


Figure 4.22. A four-bit comparator circuit.

```
module mux2to1 (w0, w1, s, f);
    input w0, w1, s;
    output f;

    assign f = s ? w1 : w0;

endmodule
```

Figure 4.23. A 2-to-1 multiplexer specified using the conditional operator.

```
module mux2to1 (w0, w1, s, f);
        input w0, w1, s;
        output reg f;

        always @(w0, w1, s)
                f = s ? w1 : w0;

endmodule
```

Figure 4.24. An alternative specification of a 2-to-1 multiplexer using the conditional operator.

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

        assign f = S[1] ? (S[0] ? w3 : w2) : (S[0] ? w1 : w0);

endmodule
```

Figure 4.25. A 4-to-1 multiplexer specified using the conditional operator.

```
module mux2to1 (w0, w1, s, f);
    input w0, w1, s;
    output reg f;

    always @(w0, w1, s)
        if (s==0)
            f = w0;
        else
            f = w1;

endmodule
```

Figure 4.26. Code for a 2-to-1 multiplexer using the **if-else** statement.

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

    always @(*)
        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
```

Figure 4.27. Code for a 4-to-1 multiplexer using the **if-else** statement.

```
module mux4to1 (W, S, f);
    input [0:3] W;
    input [1:0] S;
    output reg f;

    always @(W, S)
        if (S == 0)
            f = W[0];
        else if (S == 1)
            f = W[1];
        else if (S == 2)
            f = W[2];
        else if (S == 3)
            f = W[3];

endmodule
```

Figure 4.28. Alternative specification of a 4-to-1 multiplexer.

```
module mux16to1 (W, S, f);
        input [0:15] W;
        input [3:0] S;
        output f;
        wire [0:3] M;

        mux4to1 Mux1 (W[0:3], S[1:0], M[0]);
        mux4to1 Mux2 (W[4:7], S[1:0], M[1]);
        mux4to1 Mux3 (W[8:11], S[1:0], M[2]);
        mux4to1 Mux4 (W[12:15], S[1:0], M[3]);
        mux4to1 Mux5 (M[0:3], S[3:2], f);

endmodule
```

Figure 4.29. Hierarchical code for a 16-to-1 multiplexer.

```
module mux4to1 (W, S, f);
    input [0:3] W;
    input [1:0] S;
    output reg f;

    always @(W, S)
        case (S)
            0: f = W[0];
            1: f = W[1];
            2: f = W[2];
            3: f = W[3];
        endcase

endmodule
```

Figure 4.30. A 4-to-1 multiplexer defined using the **case** statement.

```
module dec2to4 (W, En, Y);
    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
```

Figure 4.31. Verilog code for a 2-to-4 binary decoder.

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

    always @(W, En)
    begin
        if (En == 0)
            Y = 4'b0000;
        else
            case (W)
                0: Y = 4'b1000;
                1: Y = 4'b0100;
                2: Y = 4'b0010;
                3: Y = 4'b0001;
            endcase
    end

endmodule
```

Figure 4.32. Alternative code for a 2-to4 binary decoder.

```
module dec4to16 (W, En, Y);
    input [3:0] W;
    input En;
    output [0:15] Y;
    wire [0:3] M;

    dec2to4 Dec1 (W[3:2], M[0:3], En);
    dec2to4 Dec2 (W[1:0], Y[0:3], M[0]);
    dec2to4 Dec3 (W[1:0], Y[4:7], M[1]);
    dec2to4 Dec4 (W[1:0], Y[8:11], M[2]);
    dec2to4 Dec5 (W[1:0], Y[12:15], M[3]);

endmodule
```

Figure 4.33. Verilog code for a 4-to-16 decoder.

```
module  seg7 (hex, leds);
    input  [3:0] hex;
    output  reg  [1:7] leds;

    always @(hex)
        case (hex)    //abcdefg
            0:  leds = 7'b1111110;
            1:  leds = 7'b0110000;
            2:  leds = 7'b1101101;
            3:  leds = 7'b1111001;
            4:  leds = 7'b0110011;
            5:  leds = 7'b1011011;
            6:  leds = 7'b1011111;
            7:  leds = 7'b1110000;
            8:  leds = 7'b1111111;
            9:  leds = 7'b1111011;
            10: leds = 7'b1110111;
            11: leds = 7'b0011111;
            12: leds = 7'b1001110;
            13: leds = 7'b0111101;
            14: leds = 7'b1001111;
            15: leds = 7'b1000111;
        endcase

endmodule
```

Figure 4.34. Code for a hex-to-7-segment decoder.

Operation	Inputs s_2 s_1 s_0	Outputs F
Clear	0 0 0	0 0 0 0
B−A	0 0 1	$B - A$
A−B	0 1 0	$A - B$
ADD	0 1 1	$A + B$
XOR	1 0 0	A XOR B
OR	1 0 1	A OR B
AND	1 1 0	A AND B
Preset	1 1 1	1 1 1 1

Table 4.1. The functionality of the 74381 ALU.

```
// 74381 ALU
module alu(s, A, B, F);
      input [2:0] S;
      input [3:0] A, B;
      output reg [3:0] F;

      always @(S, A, B)
          case (S)
              0: F = 4'b0000;
              1: F = B - A;
              2: F = A - B;
              3: F = A + B;
              4: F = A ^ B;
              5: F = A | B;
              6: F = A & B;
              7: F = 4'b1111;
          endcase

endmodule
```

Figure 4.35. Code that represents the functionality of the 74381 ALU chip.

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

    always @(W)
    begin
        z = 1;
        casex (W)
            4'b1xxx: Y = 3;
            4'b01xx: Y = 2;
            4'b001x: Y = 1;
            4'b0001: Y = 0;
            default: begin
                        z = 0;
                        Y = 2'bx;
                     end
        endcase
    end

endmodule
```

Figure 4.36. Verilog code for a priority encoder.

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

    always @(W, En)
        for (k = 0; k <= 3; k = k+1)
        if ((W == k) && (En == 1))
            Y[k] = 1;
        else
            Y[k] = 0;

endmodule
```

Figure 4.37. A 2-to-4 binary decoder specified using the **for** loop.

```
module priority (W, Y, z);
    input [3:0] W;
    output reg [1:0] Y;
    output reg z;
    integer k;

    always @(W)
    begin
        Y = 2'bx;
        z = 0;
        for (k = 0; k < 4; k = k+1)
            if (W[k])
            begin
                Y = k;
                z = 1;
            end
    end

endmodule
```

Figure 4.38. A priority encoder specified using the **for** loop.

Table 4.2. Verilog operators.

Operator type	Operator symbols	Operation performed	Number of operands
Bitwise	~	1's complement	1
	&	Bitwise AND	2
	|	Bitwise OR	2
	^	Bitwise XOR	2
	~^ or ^~	Bitwise XNOR	2
Logical	!	NOT	1
	&&	AND	2
	||	OR	2
Reduction	&	Reduction AND	1
	~&	Reduction NAND	1
	|	Reduction OR	1
	~|	Reduction NOR	1
	^	Reduction XOR	1
	~^ or ^~	Reduction XNOR	1

Arithmetic	+ − − * /	Addition Subtraction 2's complement Multiplication Division	2 2 1 2 2
Relational	>> < >= <=	Greater than Less than Greater than or equal to Less than or equal to	2 2 2 2
Equality	== !=	Logical equality Logical inequality	2 2
Shift	>> <<	Right shift Left shift	2 2
Concatenation	{,}	Concatenation	Any number
Replication	{{}}	Replication	Any number
Conditional	?:	Conditional	3

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

j	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

^	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

~ ^	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

Figure 4.39. Truth tables for bitwise operators.

```
module compare (A, B, AeqB, AgtB, AltB);
    input [3:0] A, B;
    output reg AeqB, AgtB, AltB;

    always @(A, B)
    begin
        AeqB = 0;
        AgtB = 0;
        AltB = 0;
        if(A == B)
          AeqB = 1;
        else if (A > B)
          AgtB = 1;
        else
          AltB = 1;
    end

endmodule
```

Figure 4.40. Verilog code for a four-bit comparator.

Operator type	Operator symbols	Precedence
Complement	! ~ -	Highest precedence
Arithmetic	* / + -	
Shift	<< >>	
Relational	< <= > >=	
Equality	== !=	
Reduction	& ~& ^ ~^ | ~|	
Logical	&& ||	
Conditional	?:	Lowest precedence

Table 4.3. Precedence of Verilog operators.

```
module addern (carryin, X, Y, S, carryout);
    parameter n=32;
    input carryin;
    input [n-1:0] X, Y;
    output [n-1:0] S;
    output carryout;
    wire [n:0] C;

    genvar k;
    assign C[0] = carryin;
    assign carryout = C[n];
    generate
        for (k = 0; k < n; k = k+1)
        begin: fulladd_stage
            wire z1, z2, z3; //wires within full-adder
            xor (S[k], X[k], Y[k], C[k]);
            and (z1, X[k], Y[k]);
            and (z2, X[k], C[k]);
            and (z3, Y[k], C[k]);
            or (C[k+1], z1, z2, z3);
        end
    endgenerate

endmodule
```

Figure 4.41. Using the **generate** loop to define an *n*-bit ripple-carry adder.

```
module mux16to1 (W, S16, f);
    input [0:15] W;
    input [3:0] S16;
    output reg f;

    always @(W, S16)
        case (S16[3:2])
            0: mux4to1 (W[0:3], S16[1:0], f);
            1: mux4to1 (W[4:7], S16[1:0], f);
            2: mux4to1 (W[8:11], S16[1:0], f);
            3: mux4to1 (W[12:15], S16[1:0], f);
        endcase

    // Task that specifies a 4-to-1 multiplexer
    task mux4to1;
        input [0:3] X;
        input [1:0] S4;
        output reg g;

        case (S4)
            0: g = X[0];
            1: g = X[1];
            2: g = X[2];
            3: g = X[3];
        endcase
    endtask

endmodule
```

Figure 4.42. Use of a task in Verilog code.

```
module mux16to1 (W, S16, f);
    input [0:15] W;
    input [3:0] S16;
    output reg f;

    // Function that specifies a 4-to-1 multiplexer
    function mux4to1;
        input [0:3] X;
        input [1:0] S4;

        case (S4)
            0: mux4to1 = X[0];
            1: mux4to1 = X[1];
            2: mux4to1 = X[2];
            3: mux4to1 = X[3];
        endcase
    endfunction

    always @(W, S16)
        case (S16[3:2])
            0: f = mux4to1 (W[0:3], S16[1:0]);
            1: f = mux4to1 (W[4:7], S16[1:0]);
            2: f = mux4to1 (W[8:11], S16[1:0]);
            3: f = mux4to1 (W[12:15], S16[1:0]);
        endcase

endmodule
```

Figure 4.43. The code from Figure 4.42 using a function.

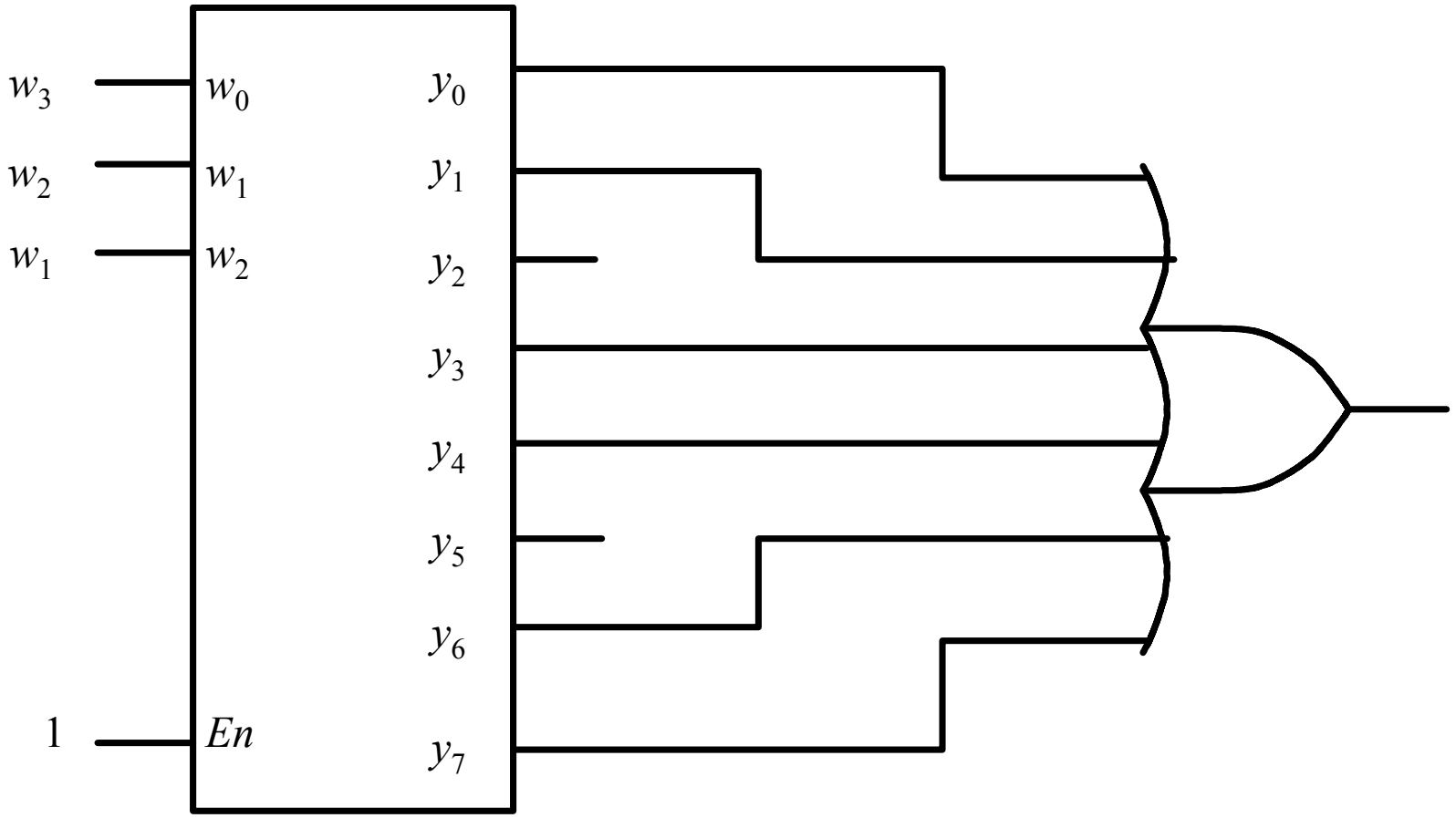


Figure 4.44. Circuit for Example 4.24.

w_7	w_6	w_5	w_4	w_3	w_2	w_1	w_0	y_2	y_1	y_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Figure 4.45. Truth table for an 8-to-3 binary encoder.

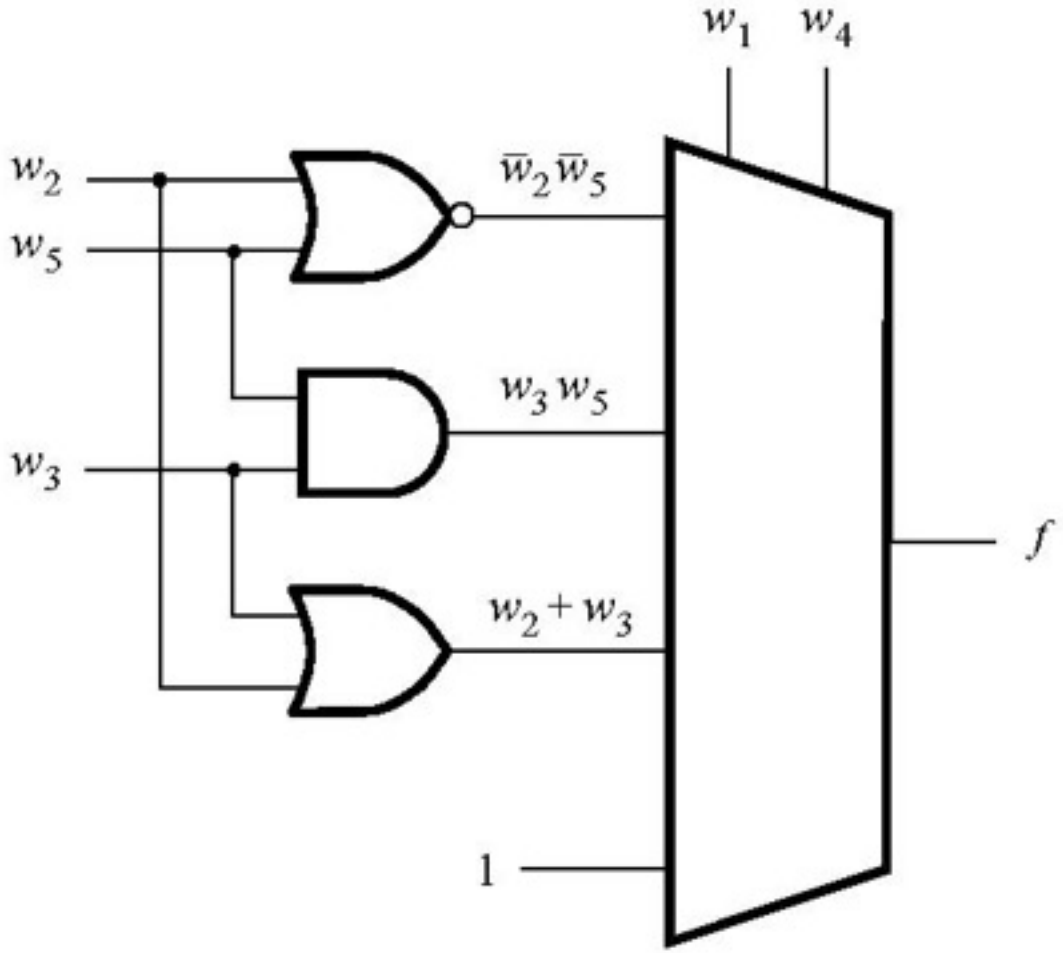


Figure 4.46. Circuit for Example 4.26.

b_2	b_1	b_0	g_2	g_1	g_0
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	0	0

Figure 4.47. Binary to Gray code conversion.

Please see “portrait orientation” PowerPoint file for Chapter 4

Figure 4.48. Circuits for Example 4.28.

Please see “portrait orientation” PowerPoint file for Chapter 4

Figure 4.49. Circuits for Example 4.29.

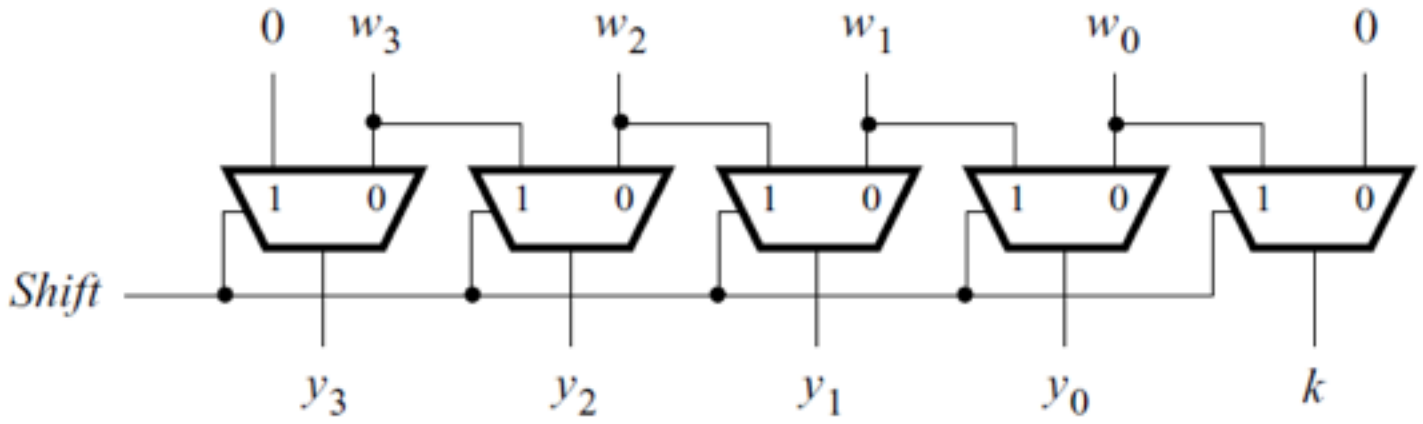
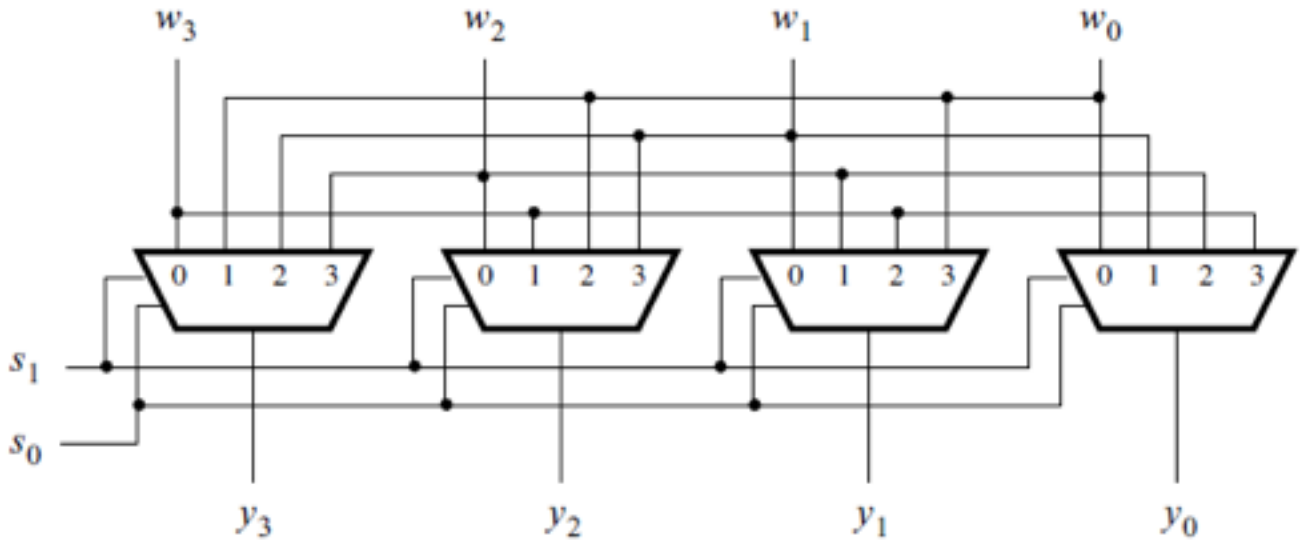


Figure 4.50. A shifter circuit.

s_1	s_0	y_3	y_2	y_1	y_0
0	0	w_3	w_2	w_1	w_0
0	1	w_0	w_3	w_2	w_1
1	0	w_1	w_0	w_3	w_2
1	1	w_2	w_1	w_0	w_3

(a) Truth table



(b) Circuit

Figure 4.51. A barrel shifter circuit.

Please see “portrait orientation” PowerPoint file for Chapter 4

Figure 4.52. Verilog code for Example 4.32.

```
module shifter (W, Shift, Y , k);
    input [3:0] W;
    input Shift;
    output reg [3:0] Y;
    output reg k;

    always @(W, Shift)
    begin
        if (Shift)
        begin
            Y[3] = 0;
            Y[2:0] = W[3:1];
            k = W[0];
        end
        else
        begin
            Y = W;
            k = 0;
        end
    end

endmodule
```

Figure 4.53. Verilog code for the circuit in Figure 4.50.

```
module shifter (W, Shift, Y , k);
    input [3:0] W;
    input Shift;
    output reg [3:0] Y;
    output reg k;

    always @(W, Shift)
    begin
        if (Shift)
        begin
            Y = W >> 1;
            k = W[0];
        end
        else
        begin
            Y = W;
            k = 0;
        end
    end

endmodule
```

Figure 4.54. Alternative Verilog code for the circuit in Figure 4.50.

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

    assign {T, Y} = {W, W} >> S;

endmodule
```

Figure 4.55. Verilog code for the barrel shifter.

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

    assign  Y =  {^X[6:0], X[6:0]};

endmodule
```

Figure 4.56. Verilog code for Example 4.35.

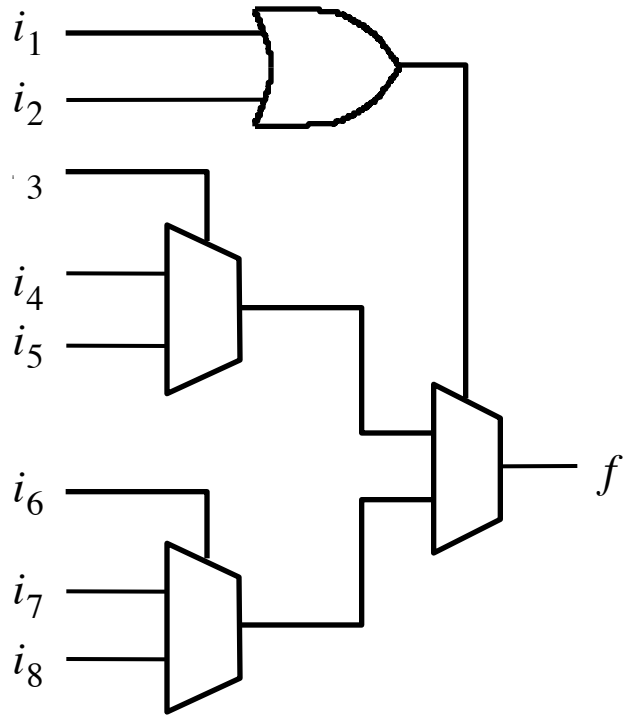


Figure P4.1. A multiplexer-based circuit.

```
module problem4_18 (W, En, y0, y1, y2, y3) ;
    input [1:0] W;
    input En;
    output reg y0, y1, y2, y3;

    always @ (W, En)
    begin
        y0 = 0;
        y1 = 0;
        y2 = 0;
        y3 = 0;
        if (En)
            if (W == 0)  y 0 = 1;
            else if (W == 1)  y1 = 1;
            else if (W == 2)  y2 = 1;
            else y3 = 1;
    end

endmodule
```

Figure P4.2. Code for Problem 4.18.

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

      always    @(W,    En)
           for  (k  =  0;  k  <=  3;  k  =  k+1)
                if  (W    ==    k)
                     Y[k]    =    En;

endmodule
```

Figure P4.3. Code for problem 4.22.