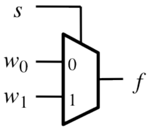
ECE/CS 3700: Fundamentals of Digital System Design

Chris J. Myers

Lecture 4: Combinational Circuit Building Blocks

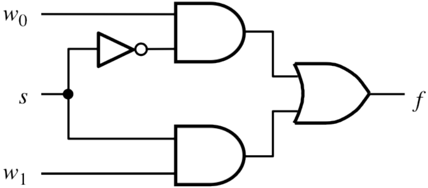
A 2-to-1 Multiplexer



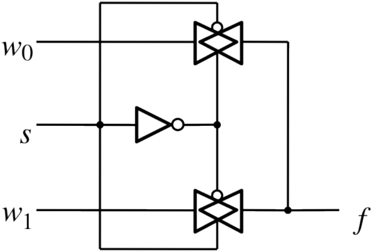
(a) Graphical symbol

s	f
0	w_0
1	w_1

(b) Truth table

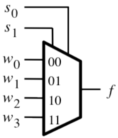


(c) Sum-of-products circuit



(d) Circuit with transmission gates

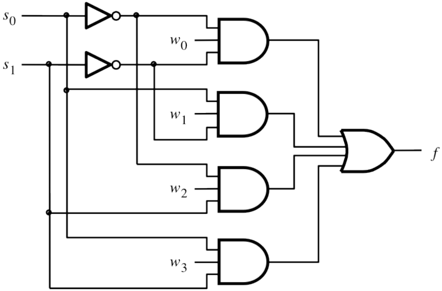
A 4-to-1 Multiplexer



(a) Graphic symbol

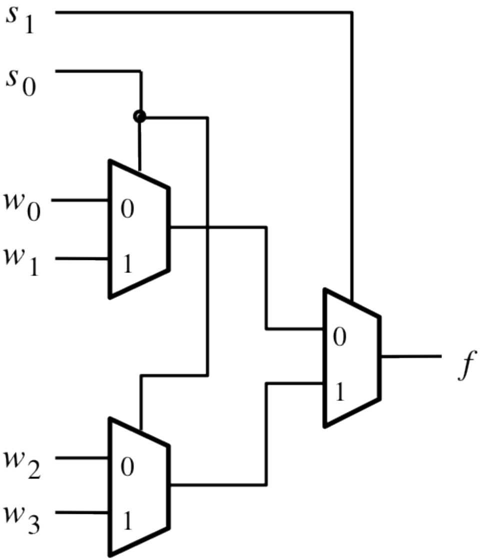
s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

(b) Truth table

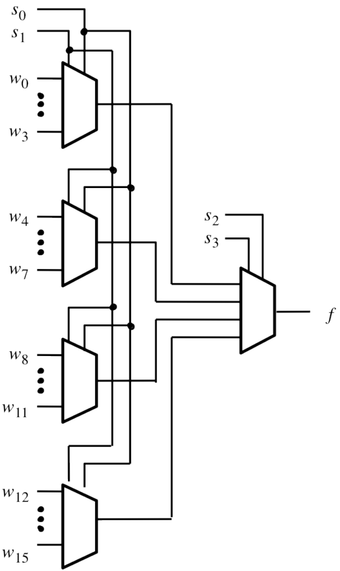


(c) Circuit

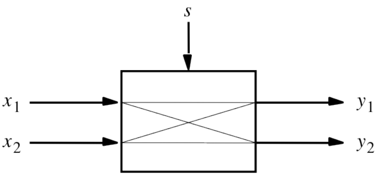
A 4-to-1 Multiplexer Using 2-to-1 Multiplexers



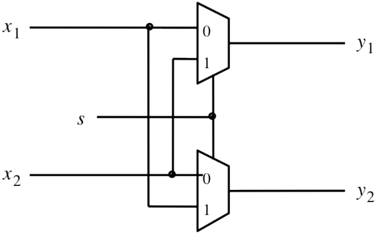
A 16-to-1 Multiplexer



A 2×2 Crossbar Switch



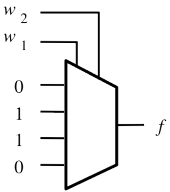
(a) A 2x2 crossbar switch



(b) Implementation using multiplexers

Synthesis of a Logic Function with Multiplexers

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

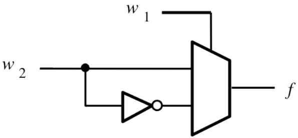


(a) Implementation using a 4-to-1 multiplexer

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



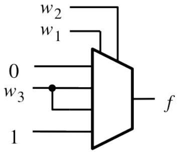
(c) Circuit

A Three-input Majority Function 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	w_2	f
0	0	0
0	1	w_3
1	0	w_3
1	1	1

(a) Modified truth table

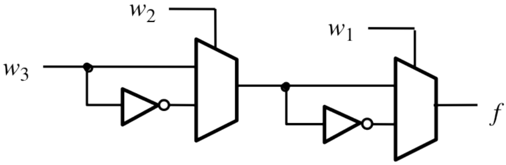


(b) Circuit

A Three-input XOR with 2-to-1 Multiplexers

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

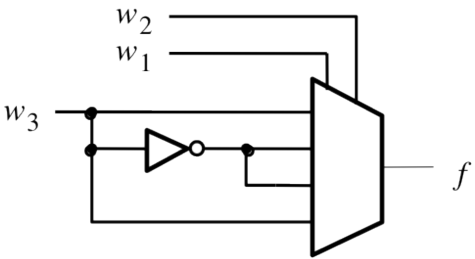


(b) Circuit

A Three-input XOR with a 4-to-1 Multiplexer

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

(a) Truth table



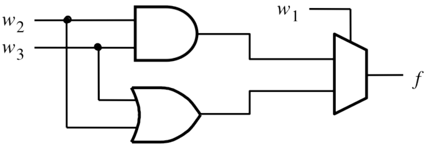
(b) Circuit

A Three-input Majority Function with a 2-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	$w_2 w_3$
1	$w_2 + w_3$

(b) Truth table



(b) Circuit

Boole's (Shannon's) Expansion

$$f(w_1, w_2, \ldots, w_n) = \overline{w}_1 \cdot f(0, w_2, \ldots, w_n) + w_1 \cdot f(1, w_2, \ldots, w_n)$$

Boole's (Shannon's) Expansion

$$
\begin{aligned}
f(w_1, w_2, \ldots, w_n) &= \overline{w}_1 \cdot f(0, w_2, \ldots, w_n) + w_1 \cdot f(1, w_2, \ldots, w_n) \\
&= \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1}
\end{aligned}
$$

Boole's (Shannon's) Expansion

$$
\begin{aligned}
f(w_1, w_2, \ldots, w_n) &= \overline{w}_1 \cdot f(0, w_2, \ldots, w_n) + w_1 \cdot f(1, w_2, \ldots, w_n) \\
&= \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1} \\
&= \overline{w}_1 \overline{w}_2 \cdot f(0, 0, w_3, \ldots, w_n) + \\
&\quad\ \overline{w}_1 w_2 \cdot f(0, 1, w_3, \ldots, w_n) + \\
&\quad\ w_1 \overline{w}_2 \cdot f(1, 0, w_3, \ldots, w_n) + \\
&\quad\ w_1 w_2 \cdot f(1, 1, w_3, \ldots, w_n)
\end{aligned}
$$

Boole's (Shannon's) Expansion

$$
\begin{aligned}
f(w_1, w_2, \ldots, w_n) &= \overline{w}_1 \cdot f(0, w_2, \ldots, w_n) + w_1 \cdot f(1, w_2, \ldots, w_n) \\
&= \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1} \\
&= \overline{w}_1 \overline{w}_2 \cdot f(0, 0, w_3, \ldots, w_n) + \\
&\quad \overline{w}_1 w_2 \cdot f(0, 1, w_3, \ldots, w_n) + \\
&\quad w_1 \overline{w}_2 \cdot f(1, 0, w_3, \ldots, w_n) + \\
&\quad w_1 w_2 \cdot f(1, 1, w_3, \ldots, w_n) \\
&= \overline{w}_1 \overline{w}_2 f_{\overline{w}_1 \overline{w}_2} + \overline{w}_1 w_2 f_{\overline{w}_1 w_2} + w_1 \overline{w}_2 f_{w_1 \overline{w}_2} + w_1 w_2 f_{w_1 w_2}
\end{aligned}
$$

A Three-input Majority Function

$$f = w_1 w_2 + w_1 w_3 + w_2 w_3$$

A Three-input Majority Function

$$\begin{aligned} f &= w_1 w_2 + w_1 w_3 + w_2 w_3 \\ &= \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1} \end{aligned}$$

A Three-input Majority Function

$$
\begin{aligned}
f &= w_1 w_2 + w_1 w_3 + w_2 w_3 \\
&= \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1} \\
&= \overline{w}_1 (0 \cdot w_2 + 0 \cdot w_3 + w_2 w_3) + w_1 (1 \cdot w_2 + 1 \cdot w_3 + w_2 w_3)
\end{aligned}
$$

A Three-input Majority Function

$$\begin{aligned}
f &= w_1 w_2 + w_1 w_3 + w_2 w_3 \\
&= \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1} \\
&= \overline{w}_1 (0 \cdot w_2 + 0 \cdot w_3 + w_2 w_3) + w_1 (1 \cdot w_2 + 1 \cdot w_3 + w_2 w_3) \\
&= \overline{w}_1 (w_2 w_3) + w_1 (w_2 + w_3 + w_2 w_3)
\end{aligned}$$

A Three-input Majority Function

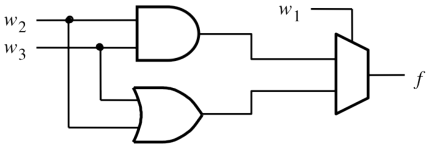
$$\begin{aligned}
f &= w_1 w_2 + w_1 w_3 + w_2 w_3 \\
&= \overline{w}_1 f_{\overline{w}_1} + w_1 f_{w_1} \\
&= \overline{w}_1 (0 \cdot w_2 + 0 \cdot w_3 + w_2 w_3) + w_1 (1 \cdot w_2 + 1 \cdot w_3 + w_2 w_3) \\
&= \overline{w}_1 (w_2 w_3) + w_1 (w_2 + w_3 + w_2 w_3) \\
&= \overline{w}_1 (w_2 w_3) + w_1 (w_2 + w_3)
\end{aligned}$$

A Three-input Majority Function with a 2-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	$w_2 w_3$
1	$w_2 + w_3$

(b) Truth table



(b) Circuit

A Three-input Majority Function

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

A Three-input Majority Function

$$
\begin{aligned}
f &= w_1 w_2 + w_1 w_3 + w_2 w_3 \\
&= \overline{w}_1 (w_2 w_3) + w_1 (w_2 + w_3) \\
&= \overline{w}_1 (g) + w_1 (h)
\end{aligned}
$$

A Three-input Majority Function

$$
\begin{aligned}
f &= w_1 w_2 + w_1 w_3 + w_2 w_3 \\
&= \overline{w}_1 (w_2 w_3) + w_1 (w_2 + w_3) \\
&= \overline{w}_1 (g) + w_1 (h) \\
g &= w_2 w_3
\end{aligned}
$$

A Three-input Majority Function

$$\begin{aligned} f &= w_1 w_2 + w_1 w_3 + w_2 w_3 \\ &= \overline{w}_1 (w_2 w_3) + w_1 (w_2 + w_3) \\ &= \overline{w}_1 (g) + w_1 (h) \\ g &= w_2 w_3 \\ &= \overline{w}_2 (0) + w_2 (w_3) \end{aligned}$$

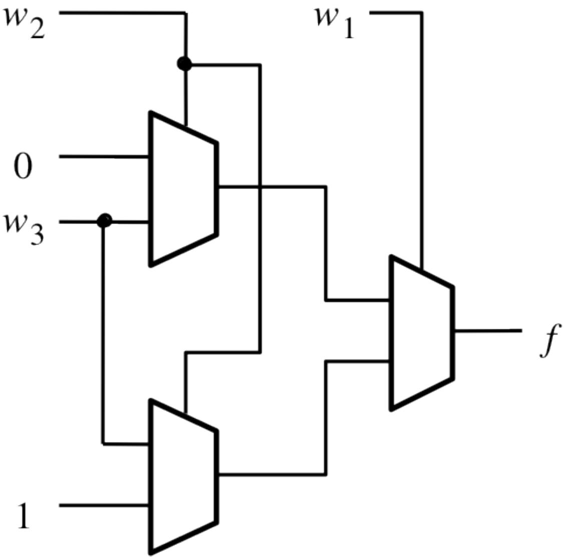
A Three-input Majority Function

$$
\begin{aligned}
f &= w_1 w_2 + w_1 w_3 + w_2 w_3 \\
&= \overline{w}_1 (w_2 w_3) + w_1 (w_2 + w_3) \\
&= \overline{w}_1 (g) + w_1 (h) \\
g &= w_2 w_3 \\
&= \overline{w}_2 (0) + w_2 (w_3) \\
h &= w_2 + w_3
\end{aligned}
$$

A Three-input Majority Function

$$
\begin{aligned}
f &= w_1 w_2 + w_1 w_3 + w_2 w_3 \\
&= \overline{w}_1 (w_2 w_3) + w_1 (w_2 + w_3) \\
&= \overline{w}_1 (g) + w_1 (h) \\
g &= w_2 w_3 \\
&= \overline{w}_2 (0) + w_2 (w_3) \\
h &= w_2 + w_3 \\
&= \overline{w}_2 (w_3) + w_2 (1)
\end{aligned}
$$

A Three-input Majority Function with two 2-to-1 Multiplexers



Another Example

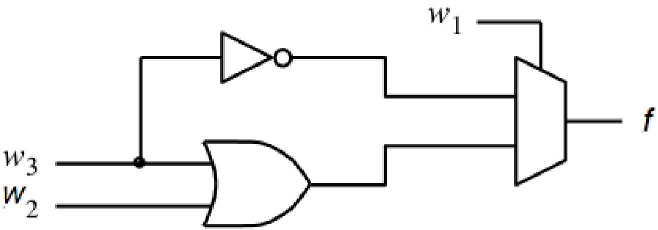
$$f = \overline{w}_1 \overline{w}_3 + w_1 w_2 + w_1 w_3$$

Another Example

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

Another Example

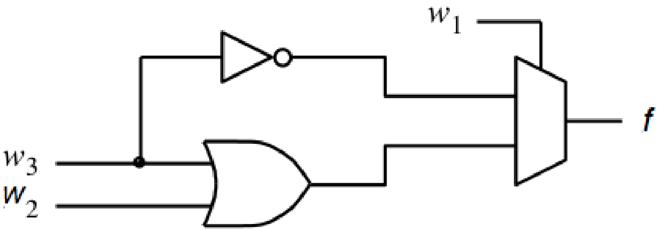
$$
\begin{aligned}
f &= \overline{w}_1 \overline{w}_3 + w_1 w_2 + w_1 w_3 \\
&= \overline{w}_1 (\overline{w}_3) + w_1 (w_2 + w_3)
\end{aligned}
$$



(a) Using a 2-to-1 multiplexer

Another Example

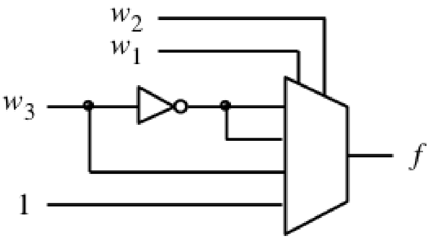
$$
\begin{aligned}
f &= \overline{w}_1\overline{w}_3 + w_1 w_2 + w_1 w_3 \\
&= \overline{w}_1(\overline{w}_3) + w_1(w_2 + w_3) \\
&= \overline{w}_1\overline{w}_2(\overline{w}_3) + \overline{w}_1 w_2(\overline{w}_3) + w_1\overline{w}_2(w_3) + w_1 w_2(1)
\end{aligned}
$$



(a) Using a 2-to-1 multiplexer

Another Example

$$
\begin{aligned}
f &= \overline{w}_1\overline{w}_3 + w_1w_2 + w_1w_3 \\
&= \overline{w}_1(\overline{w}_3) + w_1(w_2 + w_3) \\
&= \overline{w}_1\overline{w}_2(\overline{w}_3) + \overline{w}_1w_2(\overline{w}_3) + w_1\overline{w}_2(w_3) + w_1w_2(1)
\end{aligned}
$$

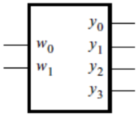


(b) Using a 4-to-1 multiplexer

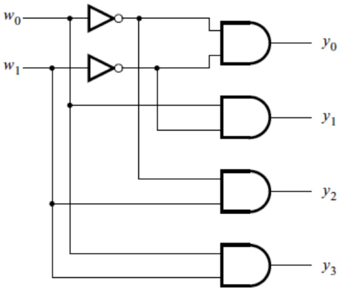
A 2-to-4 Decoder

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

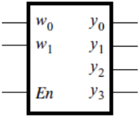


(c) Logic circuit

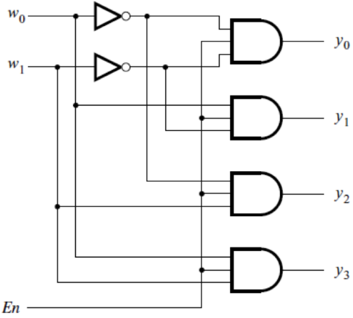
A Binary Decoder

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

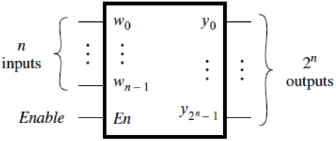
(a) Truth table



(b) Graphical symbol

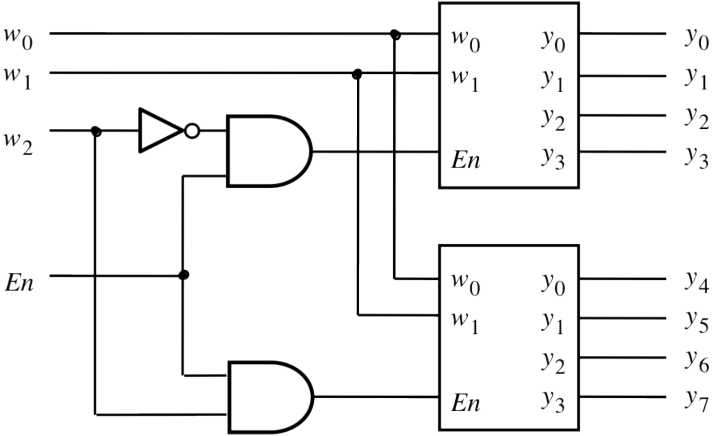


(c) Logic circuit

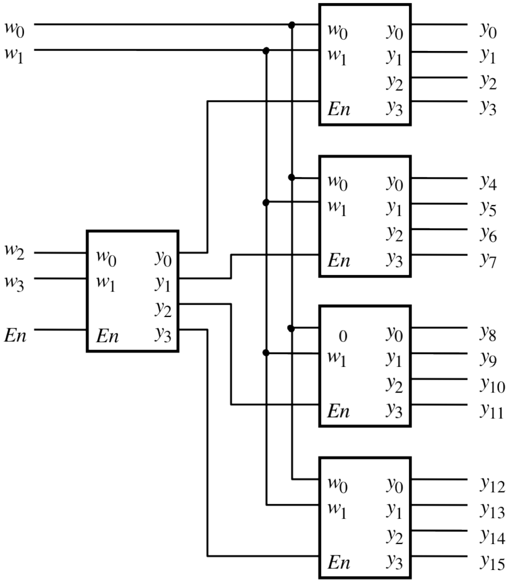


(d) An n-to-2^n decoder

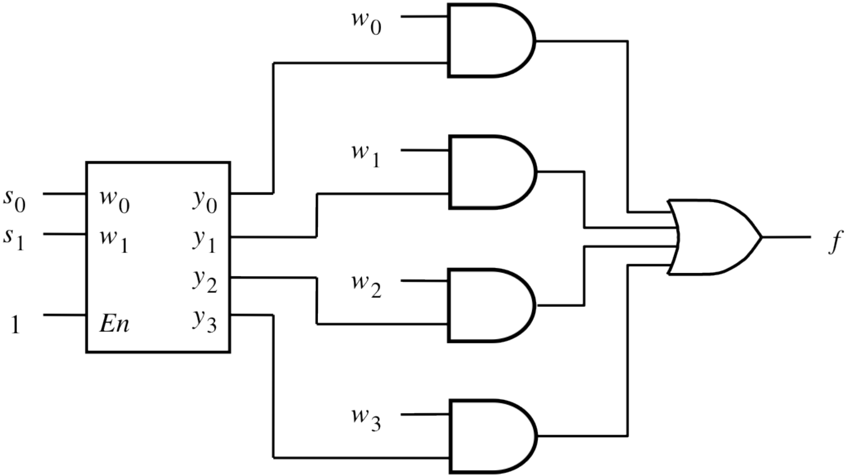
A 3-to-8 Decoder



A 4-to-16 Decoder

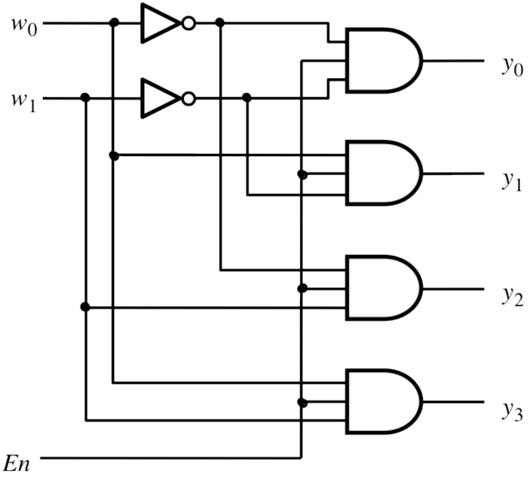


A 4-to-1 Multiplexer Built Using a Decoder

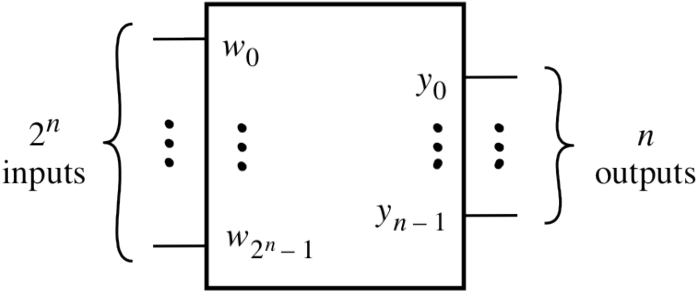


Demultiplexers

- A demultiplexer is a circuit which places the value of a single data input onto multiple data outputs is a demultiplexer.



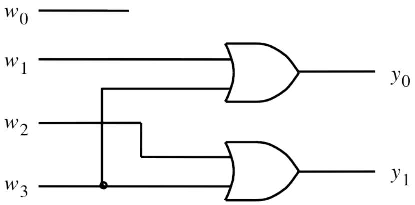
A 2^n-to-n Binary Encoder



A 4-to-2 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

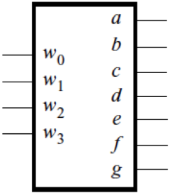
(a) Truth table



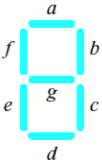
(b) Circuit

A 4-to-2 Priority 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



(a) Code converter

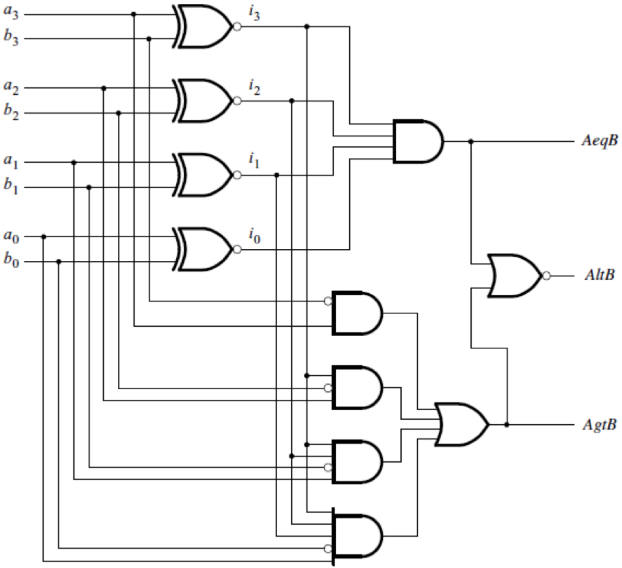


(b) 7-segment display

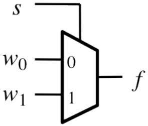
w_3	w_2	w_1	w_0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	1	1	1	0	1	1	1
1	0	1	1	0	0	1	1	1	1	1
1	1	0	0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

A Four-Bit Comparator Circuit



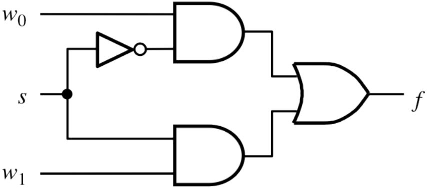
2-to-1 Multiplexer



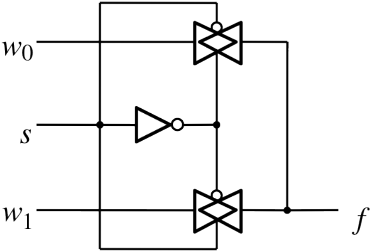
(a) Graphical symbol

s	f
0	w_0
1	w_1

(b) Truth table



(c) Sum-of-products circuit



(d) Circuit with transmission gates

2-to-1 Multiplexer: Conditional Statement

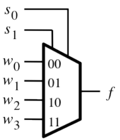
```
module mux2to1 (w0, w1, s, f);
    input w0, w1, s;
    output f;

    assign f = s ? w1 : w0;

endmodule
```

2-to-1 Multiplexer: Conditional Statement

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

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

endmodule
```

2-to-1 Multiplexer: If-else Statement

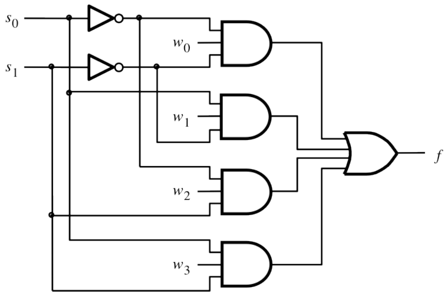
```
module mux2to1 (w0, w1, s, f);
    input w0, w1, s;
    output reg f;

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

endmodule
```



(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

4-to-1 Multiplexer: Conditional Statement

```
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
```

4-to-1 Multiplexer: 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
```

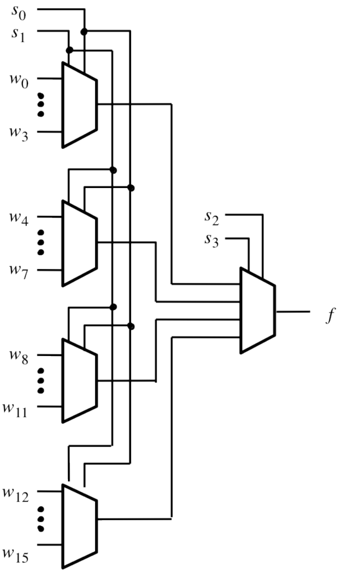
4-to-1 Multiplexer: If-else Statement

```
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
```

4-to-1 Multiplexer: Case Statement

```
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
```

A 16-to-1 Multiplexer



A 16-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
```

A 16-to-1 Multiplexer: Tasks

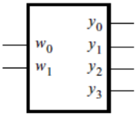
```
module mux16to1 (W, S16, f);
   input [0:15] W;
   input [3:0] S16;
   output reg f;

   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
```

A 16-to-1 Multiplexer: Functions

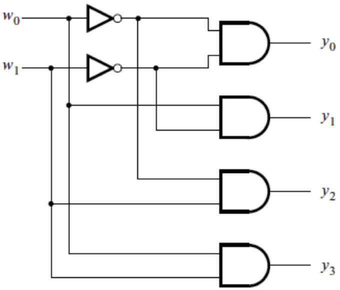
```
module mux16to1 (W, S16, f);
   input [0:15] W;
   input [3:0] S16;
   output reg f;

   // 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
```

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

A 2-to-4 Decoder

```
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
```

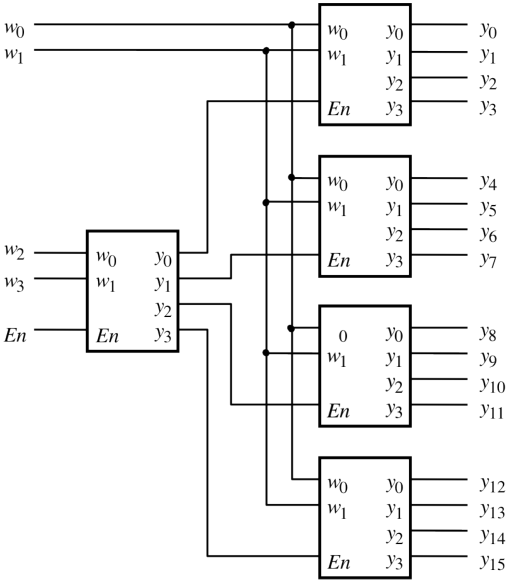
A 2-to-4 Decoder

```
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
```

A 2-to-4 Decoder

```
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
```

A 4-to-16 Decoder



A 4-to-16 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
```

A 4-to-2 Priority 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

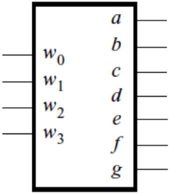
A 4-to-2 Priority Encoder

```
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
```

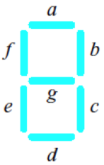
A 4-to-2 Priority Encoder

```
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
```

A Hex-to-7 Segment Display Code Converter



(a) Code converter



(b) 7-segment display

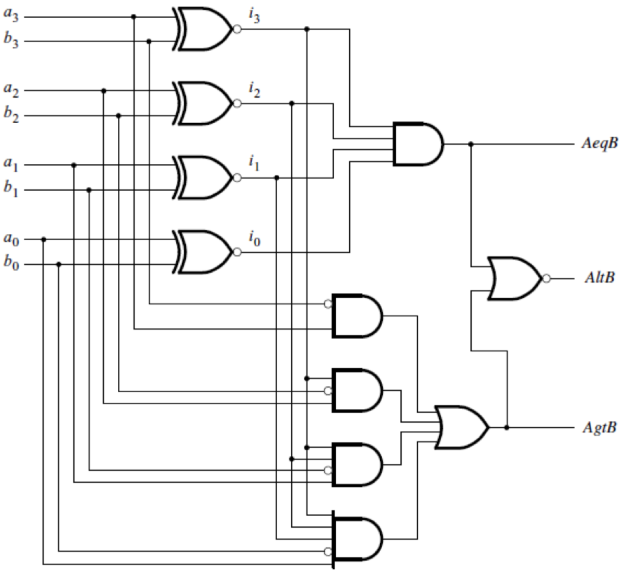
w_3	w_2	w_1	w_0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	1	1	1	0	1	1	1
1	0	1	1	0	0	1	1	1	1	1
1	1	0	0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

A Hex-to-7 Segment Display Code Converter

```
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
```

A Four-Bit Comparator Circuit



A Four-Bit Comparator Circuit

```
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
```

74381 ALU

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

Verilog for 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
```

Bitwise Operators

$C = \sim A$

Result is:

$$c_0 = \overline{a}_0$$
$$c_1 = \overline{a}_1$$
$$c_2 = \overline{a}_2$$
$$\ldots$$

$C = A \,\&\, B$

Result is:

$$c_0 = a_0 \cdot b_0$$
$$c_1 = a_1 \cdot b_1$$
$$c_2 = a_2 \cdot b_2$$
$$\ldots$$

$C = A \mid B$

Result is:

$$c_0 = a_0 + b_0$$
$$c_1 = a_1 + b_1$$
$$c_2 = a_2 + b_2$$
$$\ldots$$

$C = A \,\hat{}\, B$

Result is:

$$c_0 = a_0 \oplus b_0$$
$$c_1 = a_1 \oplus b_1$$
$$c_2 = a_2 \oplus b_2$$
$$\ldots$$

$C = A \sim\hat{}\, B$ or $C = A \,\hat{}\!\sim B$

Result is:

$$c_0 = \overline{a_0 \oplus b_0}$$
$$c_1 = \overline{a_1 \oplus b_1}$$
$$c_2 = \overline{a_2 \oplus b_2}$$
$$\ldots$$

- If operands are unequal size pads 0s to the left.

Truth Tables for Bitwise Operators

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

|	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

Logical Operators

$$
\begin{aligned}
f &= !A &\iff& \quad f = \overline{a_2 + a_1 + a_0} \\
f &= A \,\&\&\, B &\iff& \quad f = (a_2 + a_1 + a_0) \cdot (b_2 + b_1 + b_0) \\
f &= A \,||\, B &\iff& \quad f = (a_2 + a_1 + a_0) + (b_2 + b_1 + b_0)
\end{aligned}
$$

Reduction Operators

$$
\begin{aligned}
f &= \&A &\iff&& f &= a_2 \cdot a_1 \cdot a_0 \\
f &= \sim\&A &\iff&& f &= \overline{a_2 \cdot a_1 \cdot a_0} \\
f &= |A &\iff&& f &= a_2 + a_1 + a_0 \\
f &= \sim|A &\iff&& f &= \overline{a_2 + a_1 + a_0} \\
f &= \wedge A &\iff&& f &= a_2 \oplus a_1 \oplus a_0 \\
f &= \sim\wedge A &\iff&& f &= \overline{a_2 \oplus a_1 \oplus a_0}
\end{aligned}
$$

Arithmetic Operators

$$
\begin{array}{lcll}
C & = & A+B & \text{Addition} \\
C & = & A-B & \text{Subtraction} \\
C & = & -A & \text{2's Complement} \\
C & = & A*B & \text{Multiplication} \\
C & = & A/B & \text{Division}
\end{array}
$$

- Synthesis selects appropriate module from a library.
- Division not often supported by synthesis.

Relational and Equality Operators

A	$>$	B	Greater than
A	$<$	B	Less than
A	$>=$	B	Greater than or equal to
A	$<=$	B	Less than or equal to
A	$==$	B	Logical equality
A	$!=$	B	Logical inequality

- Typically used in **if-else** and **for** statements.
- Results are x when any operand contains an x.

Miscellaneous Operators

$$
\begin{array}{lcll}
C & = & A >> n & \text{Right shift} \\
C & = & A << n & \text{Left shift} \\
C & = & \{A, B\} & \text{Concatenation} \\
C & = & \{n\{A\}\} & \text{Replication} \\
C & = & e \,?\, A \;:\; B & \text{Conditional}
\end{array}
$$

Verilog Operators

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

Verilog Operators (cont)

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

Operator Precedence

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

Concluding Remarks

- Introduced a number of circuit building blocks.
- Described several Verilog constructs:
  - **if-else** statement.
  - **case** statement.
  - **for** loop.
- Statement choice is often a personal preference.
- Care must be taken to remember Verilog is not a programming language.
- Circuits should be constructed from well-defined modules.