Procedural Statements

Verilog provides *procedural statements* (also called *sequential statements*). Whereas concurrent statements are executed in parallel, procedural statements are evaluated in the order in which they appear in the code. Verilog syntax requires that procedural statements be contained inside an **always** block.

The Conditional Operator

conditional_expression ? true_expression : false_expression

If the conditional expression evaluates to 1 (true), then the value of true_expression is chosen; otherwise, the value of false_expression is chosen.

For example, the statement   A= (B < C) ? (D + 5) : (D + 2);
means that if B is less than C, the value of A will be D + 5 or else A will have the value D+2.

Parentheses are not necessary. The conditional operator can be used both in continuous assignment statements and in procedural statements inside an always block.

```
module mux2to1 (w0, w1, s, f);
input w0, w1, s;
output f;
assign f = s ? w1 : w0;
endmodule
```

```
module mux2to1 (w0, w1, s, f);
input w0, w1, s;
output f;
reg f;
always @(w0 or w1 or s)
f = s ? w1 : w0;
endmodule
```

```
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
```

The If-Else Statement

if-else statement has the syntax

if (conditional_expression) statement;
else statement;

If the expression is evaluated to true then the first statement (or a block of statements delineated by **begin** and **end** keywords) is executed or else the second statement (or a block of statements) is
executed.

```
module mux2to1 (w0, w1, s, f);
input w0, w1, s;
output f;
reg f;
always @(w0 or w1 or s)
if (s == 0)
f = w0;
else
f = w1;
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
```

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

```
module mux16to1 (W, S16, f);
input [0:15] W;
input [3:0] S16;
output f;
wire [0:3] M;
mux4to1 Mux1 (W[0:3], S16[1:0], M[0]);
mux4to1 Mux2 (W[4:7], S16[1:0], M[1]);
mux4to1 Mux3 (W[8:11], S16[1:0], M[2]);
mux4to1 Mux4 (W[12:15], S16[1:0], M[3]);
mux4to1 Mux5 (M[0:3], S16[3:2], f);
endmodule
```

The Case Statement

```
case (expression)
alternative1: statement;
alternative2: statement;
·
·
·
alternativej: statement;
[default: statement;]
endcase
```

The controlling expression and each alternative are compared bit by bit. When there is one or more matching alternative, the statement(s) associated with the first match (only)
is executed. When the specified alternatives do not cover all possible valuations of the controlling expression, the optional **default** clause should be included.

```
module mux4to1 (W, S, f);
input [0:3] W;
input [1:0] S;
output f;
reg f;
always @(W or S)
case (S)
0: f = W[0];
1: f = W[1];
2: f = W[2];
3: f = W[3];
endcase
endmodule
```

The four values that the select vector *S* can have are given as decimal numbers, but they could also be given as binary numbers.

```
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
```

```
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
```

```
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
```

```
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
```

The Casex and Casez Statements

Verilog provides two variants of the **case** statement that treat the *z* and *x* values in a different way. The **casez** statement treats all *z* values in the case alternatives and the controlling expression as don’t cares. The **casex** statement treats all *z* and *x* values as don’t cares.

```
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
```

```
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
```

Verilog Operators

Bitwise Operators

Bitwise operators operate on individual bits of operands. The ~ operator forms the 1's complement of the operand such that the statement C = ~A;
produces the result $c2 = a2'$, $c1 = a1'$, and $c0 = a0'$, where ai and ci are the bits of the vectors A and C.

Other bitwise operators operate on pairs of bits. The statement
C =A& B;
generates $c2 = a2\&b2$, $c1 = a1\&b1$, and $c0 = a0\&b0$.

A scalar function may be assigned a value as a result of a bitwise operation on two vector operands. In this case, it is only the least-significant bits of the operands that are involved in the operation. Hence the statement f =A^B; yields $f = a0 \oplus b0$.

The bitwise operations may involve operands that include the unknown logic value *x*.
if P = 4'b101x and Q = 4'b1001, then P& Q = 4'b100x while P | Q = 4'b1011.

^	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

The ! operator has the same effect on a scalar operand as the ~ operator.
Thus, f = !w = ~w.
But the effect on a vector operand is different, namely if f = !A;
then $f = (a2 + a1 + a0)'$.
The && operator implements the AND operation such that
f =A&& B; produces $f = (a2 + a1 + a0) \& (b2 + b1 + b0)$. Similarly, using the
|| operator in f =A|| B;
gives $f = (a2 + a1 + a0) \mid (b2 + b1 + b0)$.

Reduction Operators

The reduction operators perform an operation on the bits of a single vector operand and produce a one-bit result. Using the & operator in
f = &A; produces $f = a2 \& a1 \& a0$.

Arithmetic Operators

C =A+ B; puts the three-bit sum of *A* plus *B* into *C*.
The operation C = −A; places the 2’s complement of *A* into *C*.
The addition, subtraction, and multiplication operations are supported by most CAD synthesis tools. However, the division operation is often not supported. When the Verilog compiler encounters an arithmetic operator, it usually synthesizes it by using an appropriate module from a library.

Relational Operators

The relational operators are typically used as conditions in **if-else** and **for** statements.
These operators function in the same way as the corresponding operators in the C programming language. An expression that uses the relational operators returns the value 1 if it is evaluated as true, and the value 0 if evaluated as false. If there are any *x* (unknown) or *z*
bits in the operands, then the expression takes the value *x*.

```
module compare (A, B, AeqB, AgtB, AltB);
input [3:0] A, B;
output AeqB, AgtB, AltB;
reg AeqB, AgtB, AltB;
always @(A or B)
begin
AeqB = 0;
AgtB = 0;
AltB = 0;
if (A == B)
AeqB = 1;
else if (A > B)
AgtB = 1;
else
AltB = 1;
end
endmodule
```

Equality Operators

The expression (A == B) is evaluated as true if *A* is equal to *B* and false otherwise.
The != operator has the opposite effect. The result is ambiguous (*x*) if either operand contains *x* or *z* values.

Shift Operators

A vector operand can be shifted to the right or left by a number of bits specified as a constant. When bits are shifted, the vacant bit positions are filled with 0s. For example,
B =A<< 1; results in b2 = a1, b1 = a0, and b0 = 0. Similarly,
B =A>> 2; yields b2 = b1 = 0 and b0 = a2.

Concatenate Operator

This operator concatenates two or more vectors to create a larger vector. For example, D = {A, B}; defines the six-bit vector D = a2a1a0b2b1b0. Similarly, the concatenation E = {3'b111, A, 2'b00};
produces the eight-bit vector E = 111$a_2a_1a_0$00.

Replication Operator

This operator allows repetitive concatenation of the same vector, which is replicated the number of times indicated in the replication constant. For example, {3{A}} is equivalent to writing {A, A, A}. The specification {4{2'b10}} produces the eight-bit vector 10101010.
The replication operator may be used in conjunction with the concatenate operator. For instance, {2{A}, 3{B}} is equivalent to {A, A, B, B, B}.