```
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.

The priority encoder's output z must be set to 1 whenever at least one of the data inputs is 1. This output is set to 1 outside the **casex** statement in the **always** block. If none of the four alternatives matches the value of W, then the **default** clause overrides the value of z and sets it to 0. The **default** clause also indicates that the Y output can be set to any pattern because it will be ignored.

4.6.4 THE FOR LOOP

If the structure of a desired circuit exhibits a certain regularity, it may be convenient to define the circuit using a **for** loop. We introduced the **for** loop in Section 3.5.4, where it was useful in a generic specification of a ripple-carry adder. The **for** loop has the syntax

for (initial_index; terminal_index; increment) statement;

A loop control variable, which has to be of type **integer**, is set to the value given as the initial index. It is used in the statement or a block of statements delineated by **begin** and **end** keywords. After each iteration, the control variable is changed as defined in the increment. The iterations end after the control variable has reached the terminal index.

Unlike **for** loops in high-level programming languages, the Verilog **for** loop does not specify changes that take place in time through successive loop iterations. Instead, during each iteration it specifies a different subcircuit. In Figure 3.25 the **for** loop was used to define a cascade of full-adder subcircuits to form an n-bit ripple-carry adder. The **for** loop can be used to define many other structures as illustrated by the next two examples.

Example 4.18 Figure 4.37 shows how the **for** loop can be used to specify a 2-to-4 decoder circuit. The effect of the loop is to repeat the **if-else** statement four times, for $k = 0, \ldots, 3$. The first loop iteration sets $y_0 = 1$ if $W = 0$ and $En = 1$. Similarly, the other three iterations set the values of y_1, y_2, and y_3 according to the values of W and En.

This arrangement can be used to specify a large n-to-2^n decoder simply by increasing the sizes of vectors W and Y accordingly, and making $n - 1$ be the terminal index value of k.

Example 4.19 The priority encoder of Figure 4.20 can be defined by the Verilog code in Figure 4.38. In the **always** block, the output bits y_1 and y_0 are first set to the don't-care state and z is cleared to 0. Then, if one or more of the four inputs $w_3, \ldots, w_0$ is equal to 1, the **for** loop will set the valuation of y_1y_0 to match the index of the highest priority input that has the value 1. Note that each successive iteration through the loop corresponds to a higher priority. Verilog semantics specify that a signal that receives multiple assignments in an **always** block retains the last assignment. Thus the iteration that corresponds to the highest priority input that is equal to 1 will override any setting of Y established during the previous iterations.

```
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.

4.6.5 VERILOG OPERATORS

In this section we discuss the Verilog operators that are useful for synthesizing logic circuits. Table 4.2 lists these operators in groups that reflect the type of operation performed. A more complete listing of the operators is given in Table A.1.

To illustrate the results produced by the various operators, we will use three-bit vectors $A[2:0]$, $B[2:0]$, and $C[2:0]$, as well as scalars f and w.

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 $c_2 = \overline{a}_2$, $c_1 = \overline{a}_1$, and $c_0 = \overline{a}_0$, where a_i and c_i are the bits of the vectors A and C.

Most bitwise operators operate on pairs of bits. The statement

```
C = A & B;
```

generates $c_2 = a_2 \cdot b_2$, $c_1 = a_1 \cdot b_1$, and $c_0 = a_0 \cdot b_0$. Similarly, the | and ^ operators perform bitwise OR and XOR operations. The ^~ operator, which can also be written as ~^, produces the XNOR such that