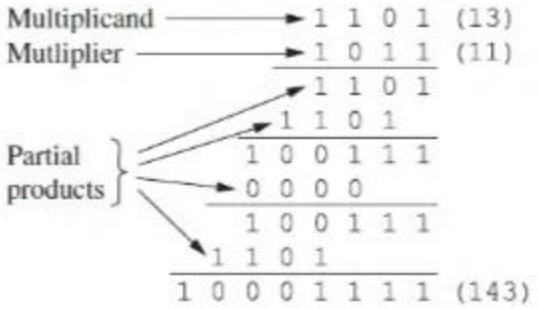
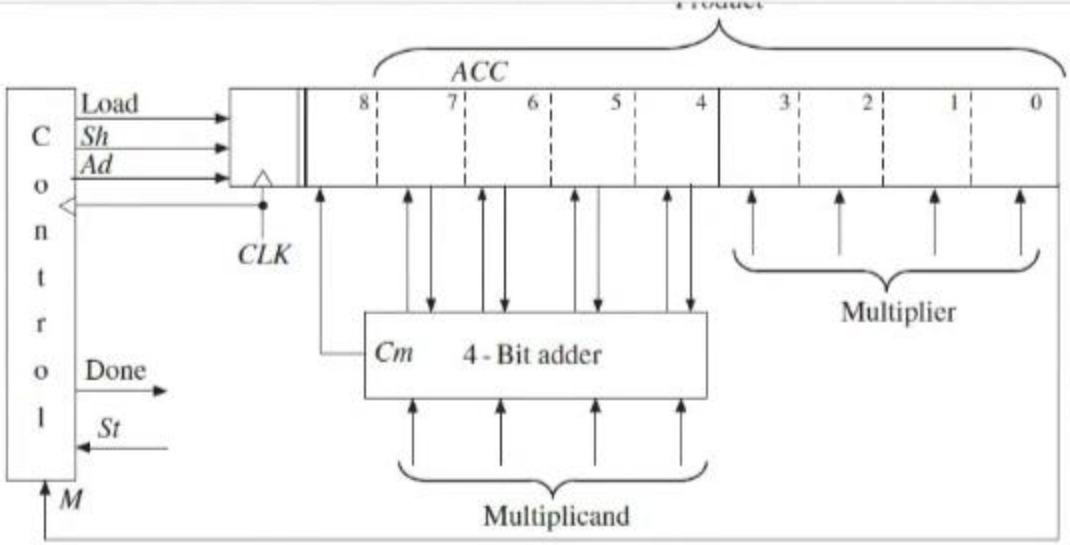
# 4.8 A Shift-and-Add Multiplier

In this section, we will design a multiplier for unsigned binary numbers. When we form the product $A \times B$, the first operand ($A$) is called the *multiplicand* and the second operand ($B$) is called the *multiplier*. As illustrated here, binary multiplication requires only shifting and adding. In the following example, we multiply $13_{10}$ by $11_{10}$ in binary




Note that each partial product is either the multiplicand (1101) shifted over by the appropriate number of places or zero. Instead of forming all the partial products first and then adding, each new partial product is added in as soon as it is formed, which eliminates the need for adding more than two binary numbers at a time.

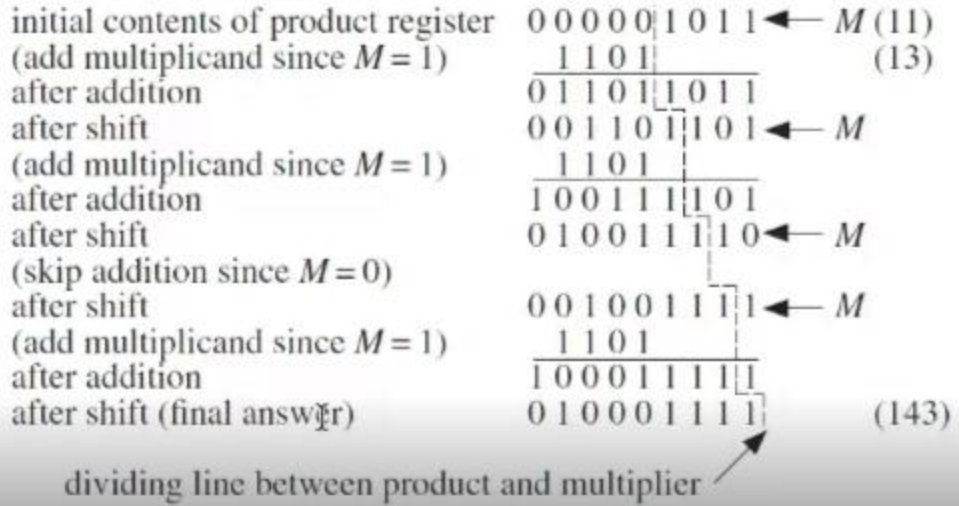



This type of multiplier is sometimes referred to as a serial-parallel multiplier, since the multiplier bits are processed serially, but the addition takes place in parallel. As indicated by the arrows on the diagram, 4 bits from the accumulator (ACC) and 4 bits from the multiplicand register are connected to the adder inputs; the 4 sum bits and the carry output from the adder are connected back to the accumulator. When an add signal (*Ad*) occurs, the adder outputs are transferred to the accumulator by the next clock pulse, thus causing the multiplicand to be added to the accumulator. An extra bit at the left end of the product register temporarily stores any carry that is generated when the multiplicand is added to the accumulator. When a shift signal

This type of multiplier is sometimes referred to as a serial-parallel multiplier, since the multiplier bits are processed serially, but the addition takes place in parallel. As indicated by the arrows on the diagram, 4 bits from the accumulator (ACC) and 4 bits from the multiplicand register are connected to the adder inputs; the 4 sum bits and the carry output from the adder are connected back to the accumulator. When an add signal (*Ad*) occurs, the adder outputs are transferred to the accumulator by the next clock pulse, thus causing the multiplicand to be added to the accumulator. An extra bit at the left end of the product register temporarily stores any carry that is generated when the multiplicand is added to the accumulator. When a shift signal (*Sh*) occurs, all 9 bits of ACC are shifted right by the next clock pulse.

Since the lower 4 bits of the product register are initially unused, we will store the multiplier in this location instead of in a separate register. As each multiplier bit is used, it is shifted out the right end of the register to make room for additional product bits. A shift signal (*Sh*) causes the contents of the product register (including the multiplier) to be shifted right one place when the next clock pulse occurs. The control circuit puts out the proper sequence of add and shift signals after a start signal ($St = 1$) has been received. If the current multiplier bit ($M$) is 1, the multiplicand is added to the accumulator followed by a right shift; if the multiplier bit is 0, the addition is skipped and only the right shift occurs. The multiplication example (13 ⊠ 11) is reworked as follows showing the location of the bits in the registers at each clock time:

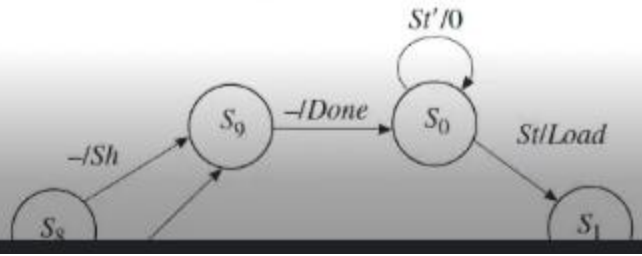
bit is used, it is shifted out the right end of the register to make room for additional product bits. A shift signal ($Sh$) causes the contents of the product register (including the multiplier) to be shifted right one place when the next clock pulse occurs. The control circuit puts out the proper sequence of add and shift signals after a start signal ($St = 1$) has been received. If the current multiplier bit ($M$) is 1, the multiplicand is added to the accumulator followed by a right shift; if the multiplier bit is 0, the addition is skipped and only the right shift occurs. The multiplication example (13 ⊠ 11) is reworked as follows showing the location of the bits in the registers at each clock time:



The control circuit must be designed to output the proper sequence of add and shift signals. Figure 4-26 shows a state graph for the control circuit. In Figure 4-26, $S_0$ is the reset state, and the circuit stays in $S_0$ until a start signal ($St = 1$) is received. This generates a *Load* signal, which causes the multiplier to be loaded into the lower 4 bits of the accumulator (ACC) and the upper 5 bits of the accumulator to be cleared. In state $S_1$, the low-order bit of the multiplier ($M$) is tested. If $M = 1$, an add signal is generated, and if $M = 0$, a shift signal is generated. Similarly, in states $S_3$, $S_5$, and $S_7$, the current multiplier bit ($M$) is tested to determine whether to generate an add or shift signal. A shift signal is always generated at the next clock time following an add signal (states $S_2$, $S_4$, $S_6$, and $S_8$). After four shifts have been generated, the control network goes to $S_9$ and a done signal is generated before returning to $S_0$.

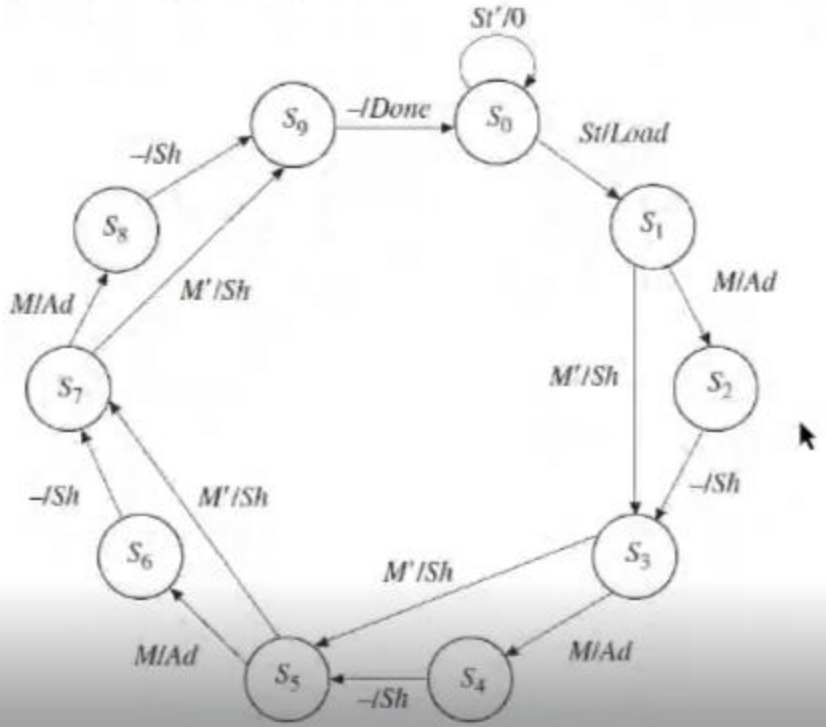
4-26: State Graph
ry Multiplier

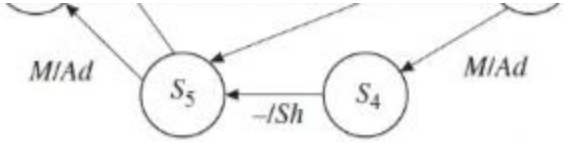
HURA R



whether to generate an add or shift signal. A shift signal is always generated at the next clock time following an add signal (states $S_2$, $S_4$, $S_6$, and $S_8$). After four shifts have been generated, the control network goes to $S_9$ and a done signal is generated before returning to $S_0$.

ate Graph
iplier






The behavioral Verilog model (Figure 4-27) corresponds directly to the state graph. Since there are 10 states, we have declared an integer ranging from 0 to 9 for the state signal. The signal *ACC* represents the 9-bit accumulator output. The statement

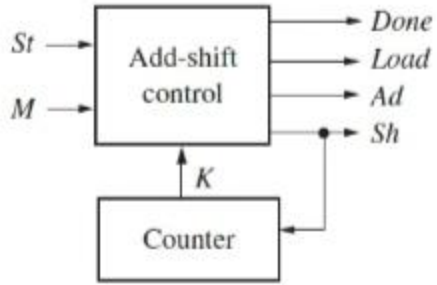
```
`define M ACC[0]
```

allows us to use the name *M* in place of *ACC(0)*. The notation 1, 3, 5, 7 : means when the state is 1 or 3 or 5 or 7, the action that follows occurs. All register operations and state changes take place on the rising edge of the clock. For example, in state 0, if *St* is 1, the multiplier is loaded into the accumulator at the same time the state changes to 1. The expression `{1'b0, ACC[7:4]} + Mcand` is used to compute the sum of two 4-bit unsigned vectors to give a 5-bit result. This represents the adder output, which is loaded into *ACC* at the same time the state counter is incremented. The right shift on *ACC* is accomplished by loading *ACC* with 0 concatenated with the upper 8 bits of *ACC*. The expression `{1'b0, ACC[8:1]}` could be replaced with `ACC >> 1`.

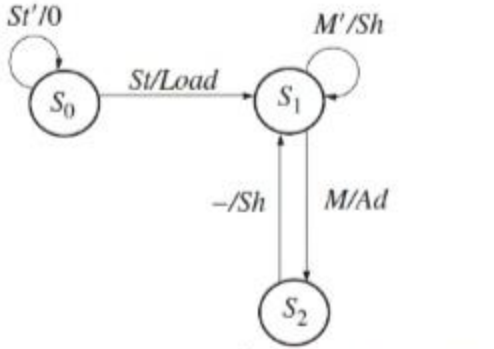
```
// This is a behavioral model of a multiplier for unsigned
// binary numbers.  It multiplies a 4-bit multiplicand
// by a 4-bit multiplier to give an 8-bit product.

// The maximum number of clock cycles needed for a
// multiply is 10.

`define M ACC[0]

module mult4X4 (Clk, St, Mplier, Mcand, Done, Result);

    input Clk;
    input St;
    input[3:0] Mplier;
    input[3:0] Mcand;
    output Done;
    output[7:0] Result;

    reg[3:0] State;
    reg[8:0] ACC;
    initial
    begin
```

```
always @(posedge Clk)
begin
        case (State)
           0 :
                      begin
                         if (St ==1'b1)
                         begin
                            ACC[8:4] <= 5'b00000 ;
                            ACC[3:0] <= Mplier ;
                            State <= 1 ;
                         end
                      end
           1, 3, 5, 7 :
                      begin
                       if (`M == 1'b1)
                       begin
                          ACC[8:4] <= {1'b0, ACC[7:4]} + Mcand ;
                            State <= State + 1 ;
                         end
                         else
                         begin
                           ACC <= {1'b0, ACC[8:1]} ;
```

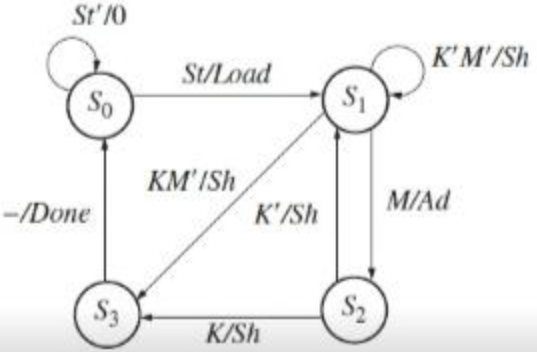
GURE 4-28: Multiplier
ontrol with Counter




(a) Multiplier control




(b) State graph for add-shift control




(c) Final state graph for add-shift control

HURA R

more shift signal and then go to $S_3$. The last shift signal will increment the counter

# 4.9 Array Multiplier

An array multiplier is a parallel multiplier that generates the partial products in a parallel fashion. The various partial products are added as soon as they are available. Consider the process of multiplication as illustrated in Table 4-3. Two 4-bit unsigned numbers, $X_3X_2X_1X_0$ and $Y_3Y_2Y_1Y_0$, are multiplied to generate a product that is possibly 8 bits. Each of the $X_iY_j$ product bits can be generated by an AND gate. Each partial product can be added to the previous sum of partial products using a row of adders. The sum output of the first row of adders, which adds the first two partial products, is $S_{13}\,S_{12}\,S_{11}\,S_{10}$, and the carry output is $C_{13}C_{12}C_{11}C_{10}$. Similar results occur for the other two rows of adders. (We have used the notation $S_{ij}$ and $C_{ij}$ to represent the sums and carries from the $i$th row of adders.)

-3: 4-Bit
r Partial
rc o ucts

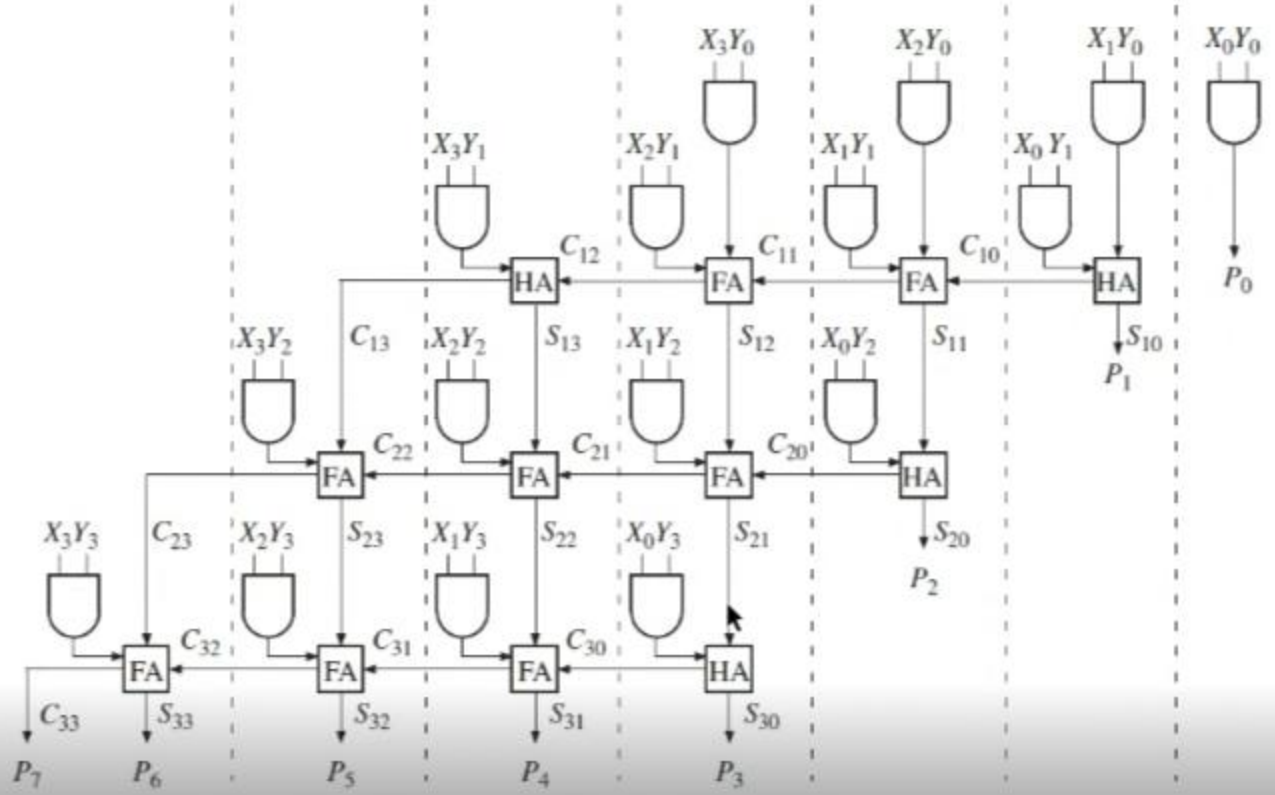
| | | | | | |
|---|---|---|---|---|---|
| | $X_3$ | $X_2$ | $X_1$ | $X_0$ | Multiplicand |
| | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ | Multiplier |
| | $X_3Y_0$ | $X_2Y_0$ | $X_1Y_0$ | $X_0Y_0$ | partial product 0 |
| $X_3Y_1$ | $X_2Y_1$ | $X_1Y_1$ | $X_0Y_1$ | | partial product 1 |

TABLE 4-3: 4-Bit Multiplier Partial Products

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | $X_3$ | $X_2$ | $X_1$ | $X_0$ | Multiplicand |
| | | | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ | Multiplier |
| | | | $X_3Y_0$ | $X_2Y_0$ | $X_1Y_0$ | $X_0Y_0$ | partial product 0 |
| | | $X_3Y_1$ | $X_2Y_1$ | $X_1Y_1$ | $X_0Y_1$ | | partial product 1 |
| | | $C_{12}$ | $C_{11}$ | $C_{10}$ | | | 1st row carries |
| | $C_{13}$ | $S_{13}$ | $S_{12}$ | $S_{11}$ | $S_{10}$ | | 1st row sums |
| | $X_3Y_2$ | $X_2Y_2$ | $X_1Y_2$ | $X_0Y_2$ | | | partial product 2 |
| | $C_{22}$ | $C_{21}$ | $C_{20}$ | | | | 2nd row carries |
| $C_{23}$ | $S_{23}$ | $S_{22}$ | $S_{21}$ | $S_{20}$ | | | 2nd row sums |
| $X_3Y_3$ | $X_2Y_3$ | $X_1Y_3$ | $X_0Y_3$ | | | | partial product 3 |
| $C_{32}$ | $C_{31}$ | $C_{30}$ | | | | | 3rd row carries |
| $C_{33}$ | $S_{33}$ | $S_{32}$ | $S_{31}$ | $S_{30}$ | | | 3rd row sums |
| $P_7$ | $P_6$ | $P_5$ | $P_4$ | $P_3$ | $P_2$ | $P_1$ | $P_0$ final product |

Figure 4-29 shows the array of AND gates and adders to perform this multiplication. If an adder has three inputs, a full adder (FA) is used, but if an adder has only two inputs, a half-adder (HA) is used. A half-adder is the same as a full adder with one of the inputs set to 0. This multiplier requires 16 AND gates, 8 full adders,

4-29: Block
m of 4 × 4 Array
lier




URA R

FIGURE 4-30: Verilog Code for 4 × 4 Array Multiplier

```
module Array_Mult (X, Y, P);

    input[3:0] X;
    input[3:0] Y;
    output[7:0] P;

    wire[3:0] C1;
    wire[3:0] C2;
    wire[3:0] C3;
    wire[3:0] S1;
    wire[3:0] S2;
    wire[3:0] S3;
    wire[3:0] XY0;
    wire[3:0] XY1;
    wire[3:0] XY2;
    wire[3:0] XY3;

    assign XY0[0] = X[0] & Y[0] ;
    assign XY1[0] = X[0] & Y[1] ;
    assign XY0[1] = X[1] & Y[0] ;
    assign XY1[1] = X[1] & Y[1] ;
```

```
assign XY0[0] = X[0] & Y[0] ;
assign XY1[0] = X[0] & Y[1] ;
assign XY0[1] = X[1] & Y[0] ;
assign XY1[1] = X[1] & Y[1] ;
assign XY0[2] = X[2] & Y[0] ;
assign XY1[2] = X[2] & Y[1] ;
assign XY0[3] = X[3] & Y[0] ;
assign XY1[3] = X[3] & Y[1] ;
assign XY2[0] = X[0] & Y[2] ;
assign XY3[0] = X[0] & Y[3] ;
assign XY2[1] = X[1] & Y[2] ;
assign XY3[1] = X[1] & Y[3] ;
assign XY2[2] = X[2] & Y[2] ;
assign XY3[2] = X[2] & Y[3] ;
assign XY2[3] = X[3] & Y[2] ;
assign XY3[3] = X[3] & Y[3] ;

FullAdder FA1 (XY0[2], XY1[1], C1[0], C1[1], S1[1]);
FullAdder FA2 (XY0[3], XY1[2], C1[1], C1[2], S1[2]);
FullAdder FA3 (S1[2], XY2[1], C2[0], C2[1], S2[1]);
FullAdder FA4 (S1[3], XY2[2], C2[1], C2[2], S2[2]);
FullAdder FA5 (C1[3], XY2[3], C2[2], C2[3], S2[3]);
FullAdder FA6 (S2[2], XY3[1], C3[0], C3[1], S3[1]);
FullAdder FA7 (S2[3], XY3[2], C3[1], C3[2], S3[2]);
```

```
    HalfAdder HA2 (XY1[3], C1[2], C1[3], S1[3]);
    HalfAdder HA3 (S1[1], XY2[0], C2[0], S2[0]);
    HalfAdder HA4 (S2[1], XY3[0], C3[0], S3[0]);

    assign P[0] = XY0[0] ;
    assign P[1] = S1[0] ;
    assign P[2] = S2[0] ;
    assign P[3] = S3[0] ;
    assign P[4] = S3[1] ;
    assign P[5] = S3[2] ;
    assign P[6] = S3[3] ;
    assign P[7] = C3[3] ;
endmodule

// Full Adder and half adder modules
// should be in the project

module FullAdder (X, Y, Cin, Cout, Sum);
    input X;
    input Y;
```

```
    assign P[6] = S3[3] ;
    assign P[7] = C3[3] ;
endmodule

// Full Adder and half adder modules
// should be in the project

module FullAdder (X, Y, Cin, Cout, Sum);

    input X;
    input Y;
    input Cin;
    output Cout;
    output Sum;

    assign Sum = X ^ Y ^ Cin ;
    assign Cout = (X & Y) | (X & Cin) | (Y & Cin) ;
endmodule

module HalfAdder (X, Y, Cout, Sum);

    input X;
    input Y;
    output Cout;
```

```
    input Y;
    input Cin;
    output Cout;
    output Sum;

    assign Sum = X ^ Y ^ Cin ;
    assign Cout = (X & Y) | (X & Cin) | (Y & Cin) ;
endmodule

module HalfAdder (X, Y, Cout, Sum);

    input X;
    input Y;
    output Cout;
    output Sum;

    assign Sum = X ^ Y ;
    assign Cout = X & Y ;
endmodule
```