DIGITAL SYSTEM DESIGN USING VERILOG

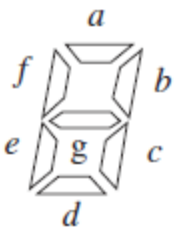
COURSE CODE: 19EC5DCDSV
(3 CREDITS)
MODULE-4B

Faculty In-Charge: Dr. Dinesha P

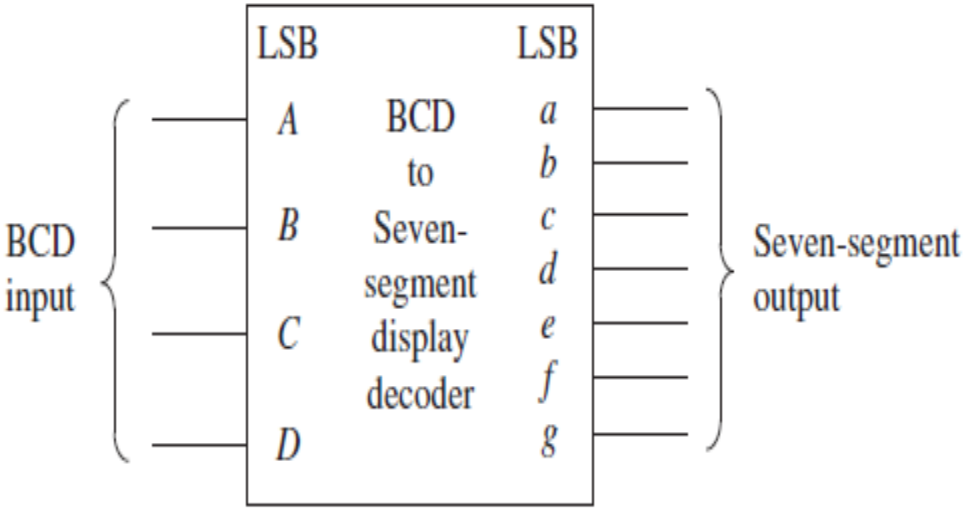
drdinesh-ece@dayanandasagar.edu

BCD to 7-Segment Display Decoder:

- Seven segment displays are often used to display digits in digital counters, watches, and clocks.



- A block diagram of the decoder is shown in Figure



Behavioral Verilog Code for BCD to 7-Segment Decoder

```
module bcd_seven (bcd, seven);
  input [3:0] bcd;
  output[7:1] seven;

  reg   [7:1] seven;

  always @(bcd)
  begin
    case (bcd)
      4'b0000 : seven = 7'b0111111 ;
      4'b0001 : seven = 7'b0000110 ;
      4'b0010 : seven = 7'b1011011 ;
      4'b0011 : seven = 7'b1001111 ;
      4'b0100 : seven = 7'b1100110 ;
      4'b0101 : seven = 7'b1101101 ;
      4'b0110 : seven = 7'b1111101 ;
      4'b0111 : seven = 7'b0000111 ;
      4'b1000 : seven = 7'b1111111 ;
      4'b1001 : seven = 7'b1101111 ;
      default : seven = 7'b0000000 ;
    endcase
  end
endmodule
```

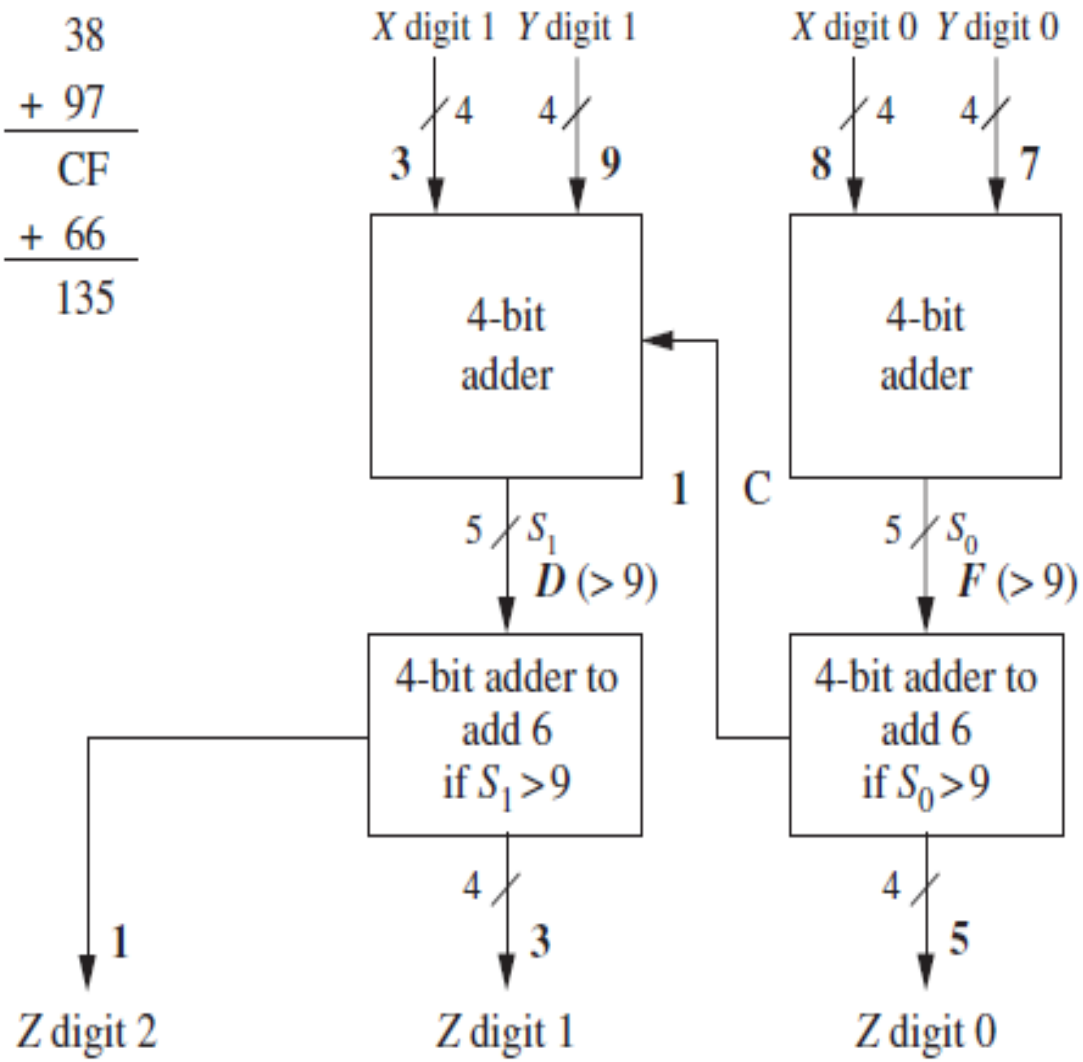
A BCD Adder:

- In 2-digit BCD adder, will add two BCD numbers and produce the sum in BCD format.
- When BCD numbers are added, each sum digit should be adjusted to skip the six unused codes.

For example, if 6 is added with 8, the sum is 14 in decimal form. A binary adder would yield 1110, but the lowest digit of the BCD sum should read 4. In order to obtain the correct BCD digit, 6 should be added to the sum whenever it is greater than 9.

- Figure illustrates the hardware that will be required to perform the addition of 2 BCD digits. A binary adder adds the least significant digits. If the sum is greater than 9, an adder adds 6 to yield the correct sum digit along with a carry digit to be added with the next digit.

Addition of Two BCD Numbers



- The Verilog code for the BCD adder is shown in Figure The input BCD numbers are represented by *X and Y. The BCD sum of two 2-digit BCD numbers* can exceed two digits and hence three BCD digits are provided for the sum, which is represented by *Z*.
- *The compiler directive 'define' can be used to denote each digit* of each BCD number. For example, the upper digit of *X can be denoted by Xdig1* by using the Verilog statement.

```
define Xdig1  X[7:4]
```

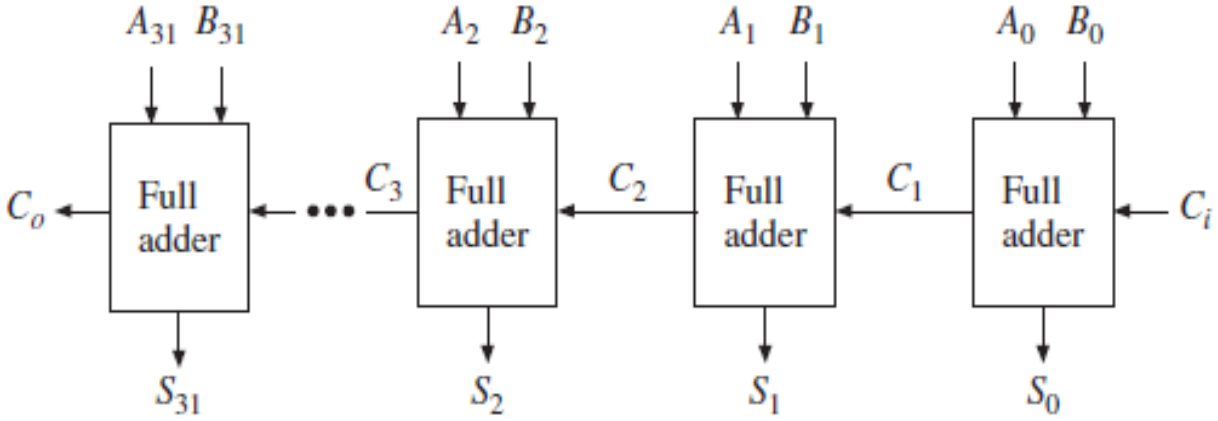
- This statement allows us to use the name *Xdig1 whenever we wish to refer to the* upper digit of *X*.

```
`define Xdig1 X[7:4]
`define Xdig0 X[3:0]
`define Ydig1 Y[7:4]
`define Ydig0 Y[3:0]
`define Zdig2 Z[11:8]
`define Zdig1 Z[7:4]
`define Zdig0 Z[3:0]

module BCD_Adder (X, Y, Z);

   input[7:0] X;
   input[7:0] Y;
   output[11:0] Z;
```

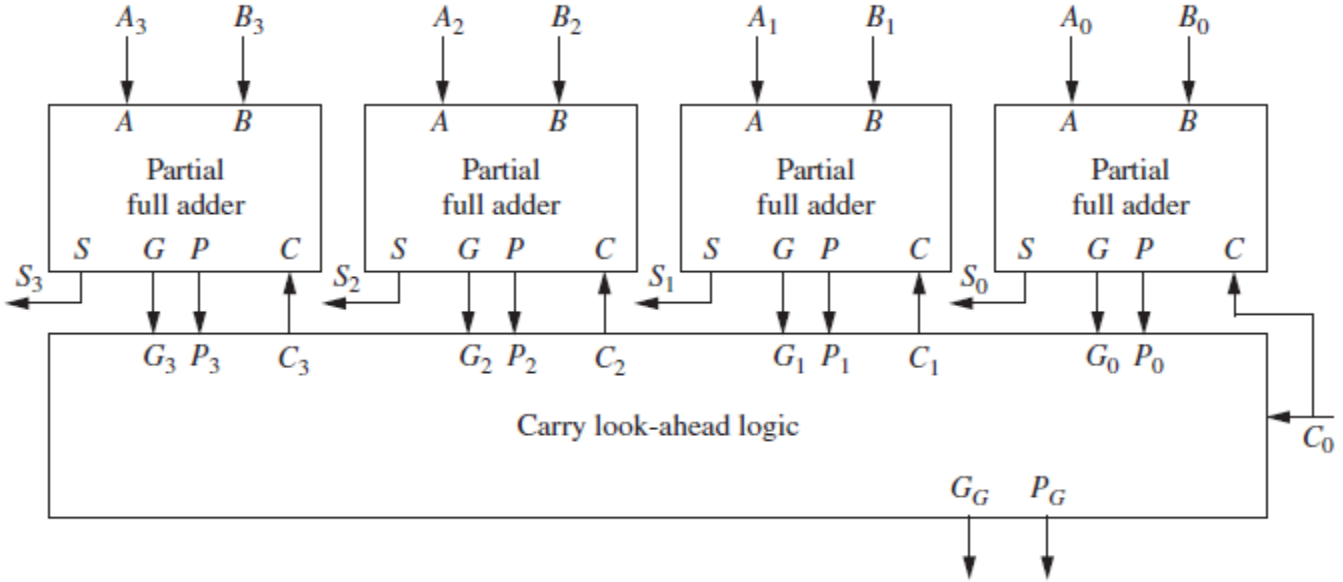
```
    wire[4:0] S0;
    wire[4:0] S1;
    wire C;

    assign S0 = `Xdig0 + `Ydig0 ;
    assign `Zdig0 = (S0 > 9) ? S0[3:0] + 6 : S0[3:0] ;
    assign C = (S0 > 9) ? 1'b1 : 1'b0 ;

    assign S1 = `Xdig1 + `Ydig1 + C ;
    assign `Zdig1 = (S1 > 9) ? S1[3:0] + 6 : S1[3:0] ;
    assign `Zdig2 = (S1 > 9) ? 4'b0001 : 4'b0000 ;

endmodule
```

32-Bit Adders:

Ripple carry adder:



Carry Look-Ahead Adders:



Verilog Description of a 4-Bit Carry Look-Ahead Adder:

```
module CLA4 (A, B, Ci, S, Co, PG, GG);
    input[3:0] A;
    input[3:0] B;
    input Ci;
    output[3:0] S;
    output Co;
    output PG;
    output GG;

    wire[3:0] G;
    wire[3:0] P;
    wire[3:1] C;
    CLALogic CarryLogic (G, P, Ci, C, Co, PG, GG);
    GPFullAdder FA0 (A[0], B[0], Ci, G[0], P[0], S[0]);
    GPFullAdder FA1 (A[1], B[1], C[1], G[1], P[1], S[1]);
    GPFullAdder FA2 (A[2], B[2], C[2], G[2], P[2], S[2]);
    GPFullAdder FA3 (A[3], B[3], C[3], G[3], P[3], S[3]);
endmodule
```

```
module CLALogic (G, P, Ci, C, Co, PG, GG);
    input[3:0] G;
    input[3:0] P;
    input Ci;
    output[3:1] C;
    output Co;
    output PG;
    output GG;

    wire GG_int;
    wire PG_int;

    assign C[1] = G[0] | (P[0] & Ci) ;
    assign C[2] = G[1] | (P[1] & G[0]) | (P[1] & P[0] & Ci) ;
    assign C[3] = G[2] | (P[2] & G[1]) | (P[2] & P[1] & G[0]) | (P[2] & P[1] &
                  P[0] & Ci) ;
    assign PG_int = P[3] & P[2] & P[1] & P[0] ;
    assign GG_int = G[3] | (P[3] & G[2]) | (P[3] & P[2] & G[1]) | (P[3] & P[2] &
                    P[1] & G[0]) ;
    assign Co = GG_int | (PG_int & Ci) ;
    assign PG = PG_int ;
    assign GG = GG_int ;
endmodule
```

```
module GPFullAdder (X, Y, Cin, G, P, Sum);
    input X;
    input Y;
    input Cin;
    output G;
    output P;
    output Sum;

    wire P_int;

    assign G = X & Y ;
    assign P = P_int ;
    assign P_int = X ^ Y ;
    assign Sum = P_int ^ Cin ;
endmodule
```

- The behavioral Verilog code for a 32-bit adder using the "1" operator is shown below. If this code is synthesized, depending on the tools used and the target technology, an adder with characteristics in between a ripple-carry adder and a fast 2-level adder will be obtained. The various topologies result in different area, power, and delay characteristics.

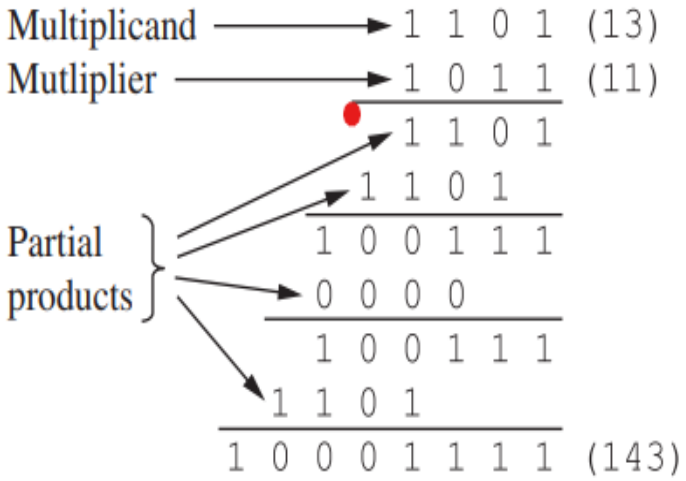
```
module Adder32 (A, B, Ci, S, Co);

    input[31:0] A;
    input[31:0] B;
    input Ci;
    output[31:0] S;
    output Co;

    wire[32:0] Sum33;

    assign Sum33 = A + B + Ci ;
    assign S = Sum33[31:0] ;
    assign Co = Sum33[32] ;
endmodule
```

Is ripple-carry adder the smallest 32-bit adder?

- A 32-bit ripple-carry adder uses 32 1-bit adders. One could design a 32-bit serial adder using a single 1-bit full adder. The input numbers are shifted into the adder, one bit at a time, and carry output from addition of each pair of bits is saved in a flip-flop and fed back to the next addition. The hardware illustrated in Figure accomplishes this

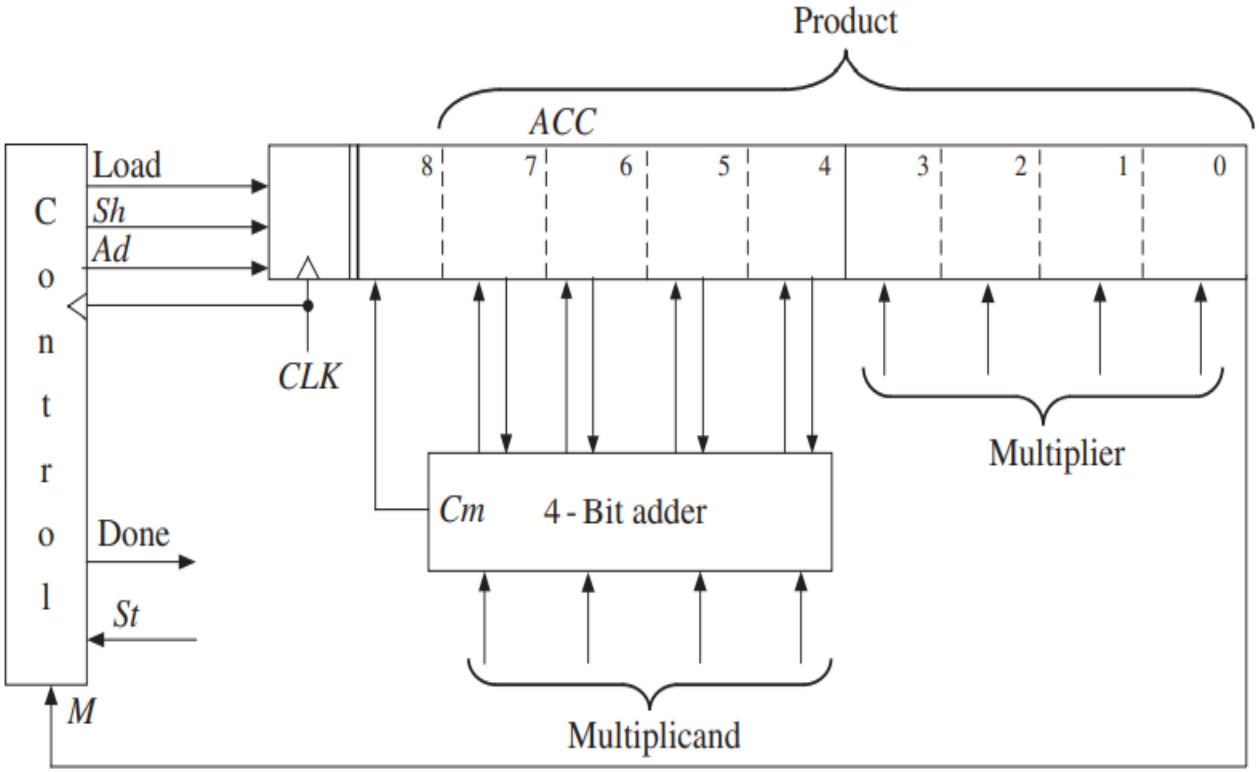
A Shift-and-Add Multiplier: (serial-parallel multiplier)

- In product A × B, the first operand (A) is called the multiplicand and the second operand (B) is called the multiplier.
- Binary multiplication requires only shifting and adding. In the following example, we multiply 13_{10} by 11_{10} in binary.



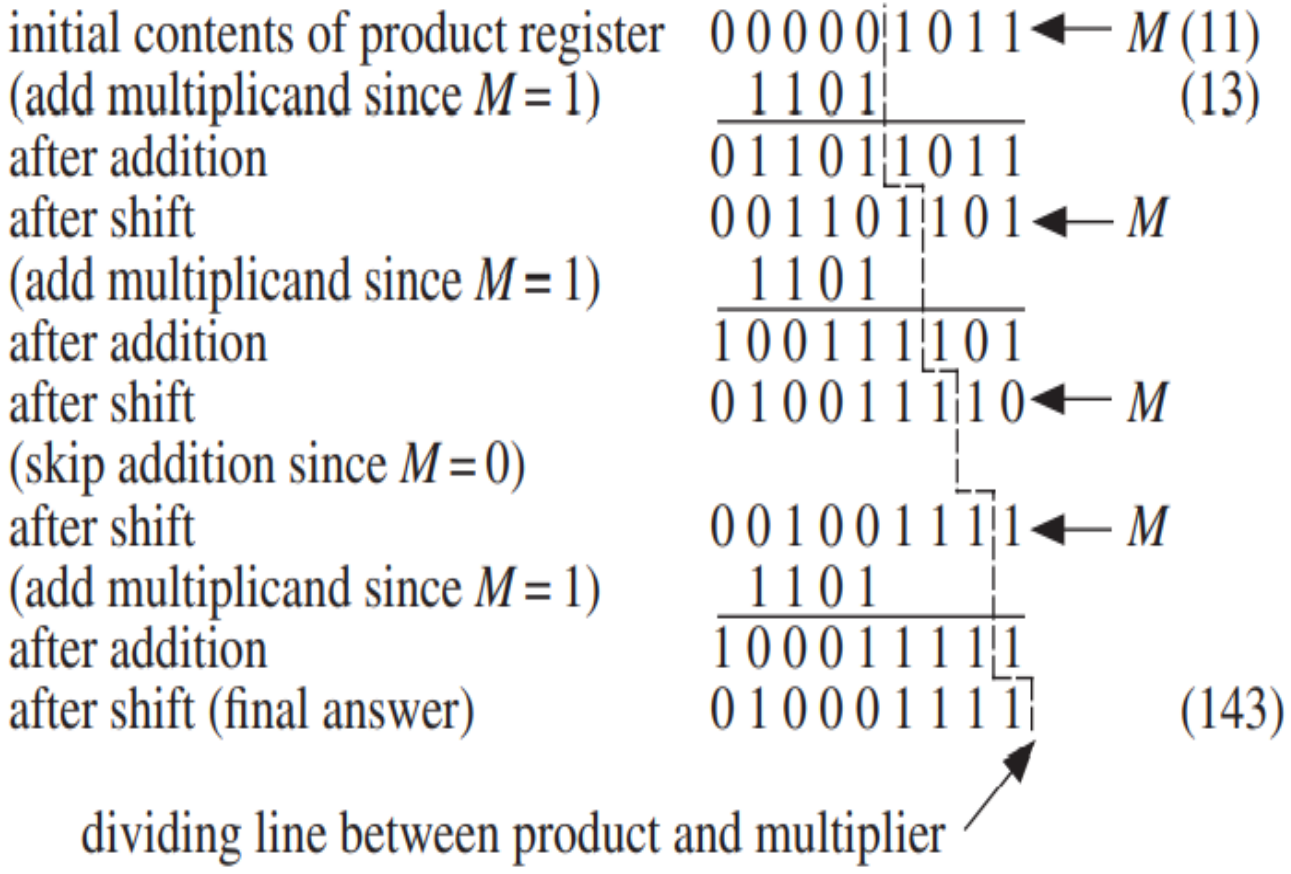
- Multiplication of two 4-bit numbers requires a 4-bit multiplicand register, a 4-bit multiplier register, a 4-bit full adder, and an 8-bit register for the product.

- The product register serves as an accumulator to accumulate the sum of the partial products. If the multiplicand were shifted left each time before it was added to the accumulator, as was done in the previous example, an 8-bit adder would be needed. Therefore, it is better to shift the contents of the product register to the right each time, as shown in the block diagram of Figure.



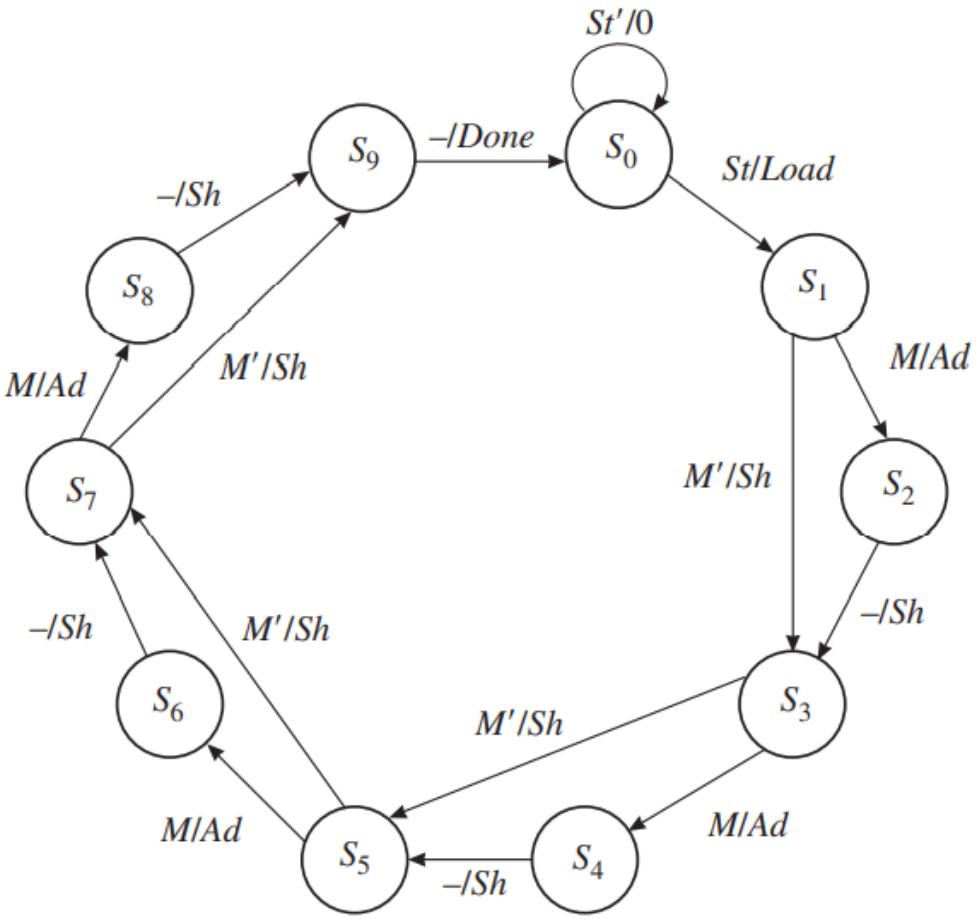
- As indicated by the arrows on the diagram, 4 bits from the accumulator (ACC) and 4 bits from the multiplicand register are connected to the adder inputs; the 4 sum bits and the carry output from the adder are connected back to the accumulator.
- When an add signal (Ad) occurs, the adder outputs are 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.
- 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.



- Figure shows a state graph for the control circuit.
- In Figure, S0 is the reset state, and the circuit stays in S0 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 S1 , 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 S3 , S5 , and S7 , 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 S2 , S4 , S6 , and S8). After four shifts have been generated, the control network goes to S9 and a done signal is generated before returning to S0

State Graph for Binary Multiplier Control



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]

Behavioral Model for 4 × 4 Binary Multiplier:

```
// 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
    State = 0;
    ACC   = 0;
end

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]} ;
```

```
                        State <= State + 2 ;
                    end
                end
            2, 4, 6, 8 :
                begin
                    ACC <= {1'b0, ACC[8:1]} ;
                    State <= State + 1 ;
                end
            9 :
                begin
                    State <= 0 ;
                end
        endcase
    end

    assign Done = (State == 9) ? 1'b1 : 1'b0 ;
    assign Result = (State == 9) ? ACC[7:0] : 8'b01010101 ;

endmodule
```

Array Multiplier:

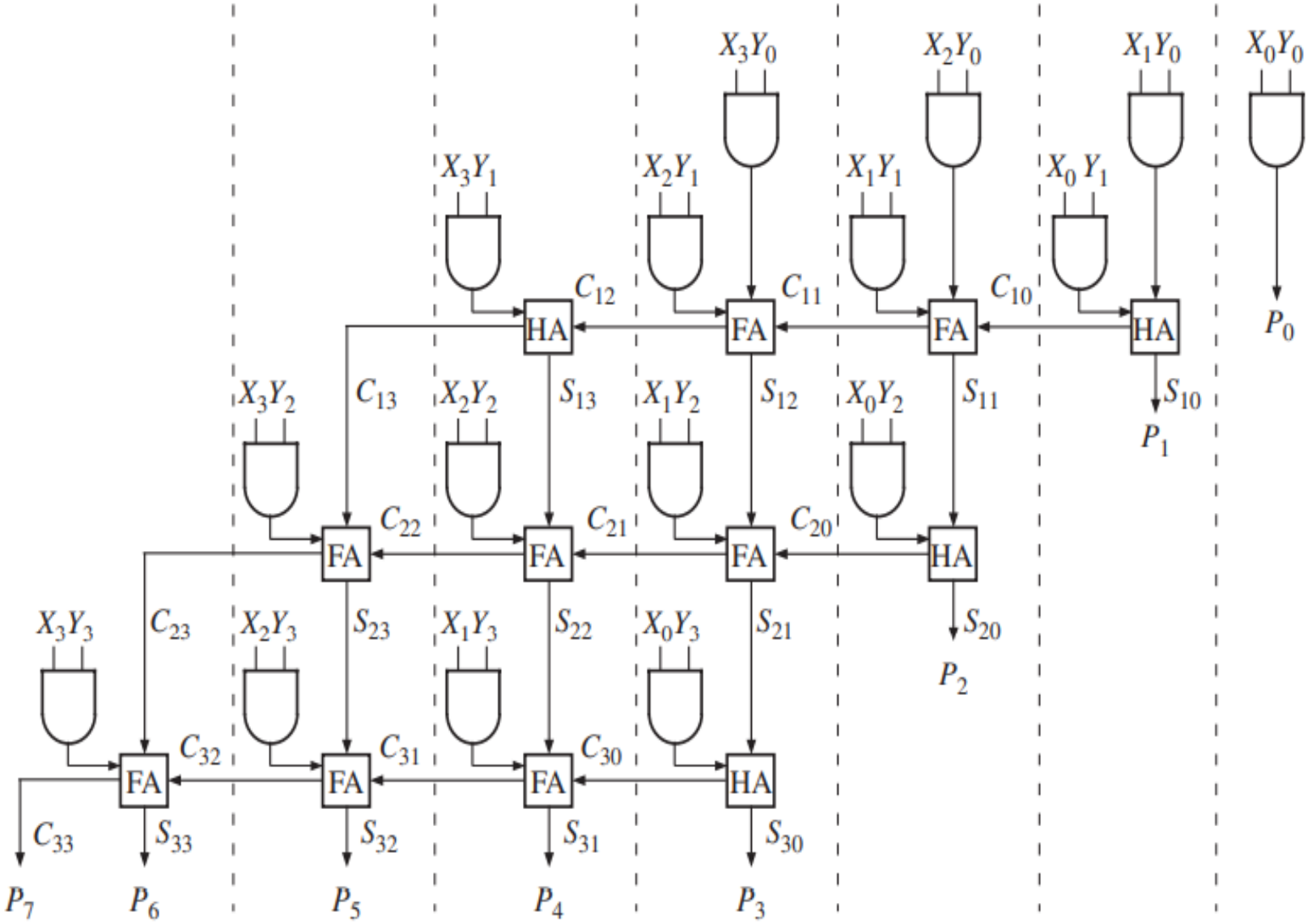
An array multiplier is a parallel multiplier that generates the partial products in a parallel fashion. Consider the process of multiplication as illustrated figure.

Two 4-bit unsigned numbers, X3 X2 X1 X0 and Y3 Y2 Y1 Y0 , are multiplied to generate a product that is possibly 8 bits. Each of the Xi Yj product bits can be generated by an AND gate

			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 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.
- This multiplier requires 16 AND gates, 8 full adders, and 4 half-adders.

- In general, an n-bit-by-n-bit array multiplier would require n^2 AND gates, n(n -2) full adders, and n half-adders.


X_3Y_0
X_2Y_0
X_1Y_0
X_0Y_0
X_3Y_1
X_2Y_1
X_1Y_1
X_0Y_1
C_{12}
C_{11}
C_{10}
HA
FA
FA
HA
P_0
X_3Y_2
C_{13}
X_2Y_2
S_{13}
X_1Y_2
S_{12}
X_0Y_2
S_{11}
S_{10}
P_1
C_{22}
C_{21}
C_{20}
FA
FA
FA
HA
X_3Y_3
C_{23}
X_2Y_3
S_{23}
X_1Y_3
S_{22}
X_0Y_3
S_{21}
S_{20}
P_2
C_{32}
C_{31}
C_{30}
FA
FA
FA
HA
C_{33}
S_{33}
S_{32}
S_{31}
S_{30}
P_7
P_6
P_5
P_4
P_3

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[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]);
FullAdder FA8 (C2[3], XY3[3], C3[2], C3[3], S3[3]);
HalfAdder HA1 (XY0[1], XY1[0], C1[0], S1[0]);
```

```
    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;
    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
```

A Signed Integer/Fraction Multiplier:

- Several algorithms are available for multiplication of signed binary numbers. The following procedure is a straightforward way to carry out the multiplication:
  1. **Complement the multiplier if negative.**
  2. **Complement the multiplicand if negative.**
  3. **Multiply the two positive binary numbers.**
  4. **Complement the product if it should be negative.**

- This method is simple, but requires more hardware and computation time than the other available methods.
- In other method, requires only to complement the multiplicand. Complementation of the multiplier or product is not necessary. Although the method works equally well with integers or fractions.
- Using 2's complement for negative numbers, we will represent signed binary fractions in the following form:

0.101 +5/8 1.011 -5/8

- When both the multiplicand and the multiplier are positive, standard binary multiplication is used. For example,

0.1 1 1	(+7/8)	←	Multiplicand
× 0.1 0 1	(+5/8)	←	Multiplier
(0. 0 0)0 1 1 1	(+7/64)	←	*Note*: The proper representation of the
(0.)0 1 1 1	(+7/16)	←	fractional partial products requires extension
0. 1 0 0 0 1 1	(+35/64)		of the sign bit past the binary point, as indicated in parentheses. (Such extension is not necessary in the hardware.)

- When the multiplicand is negative and the multiplier is positive, the procedure is the same as in the previous case, except that we must extend the sign bit of the multiplicand so that the partial products and final product will have the proper negative sign. For example,

1.1 0 1	(−3/8)		
× 0.1 0 1	(+5/8)		
(1. 1 1)1 1 0 1	(−3/64)	←	*Note*: The extension of the sign bit provides proper representation of the negative products.
(1.)1 1 0 1	(−3/16)	←	
1. 1 1 0 0 0 1	(−15/64)		

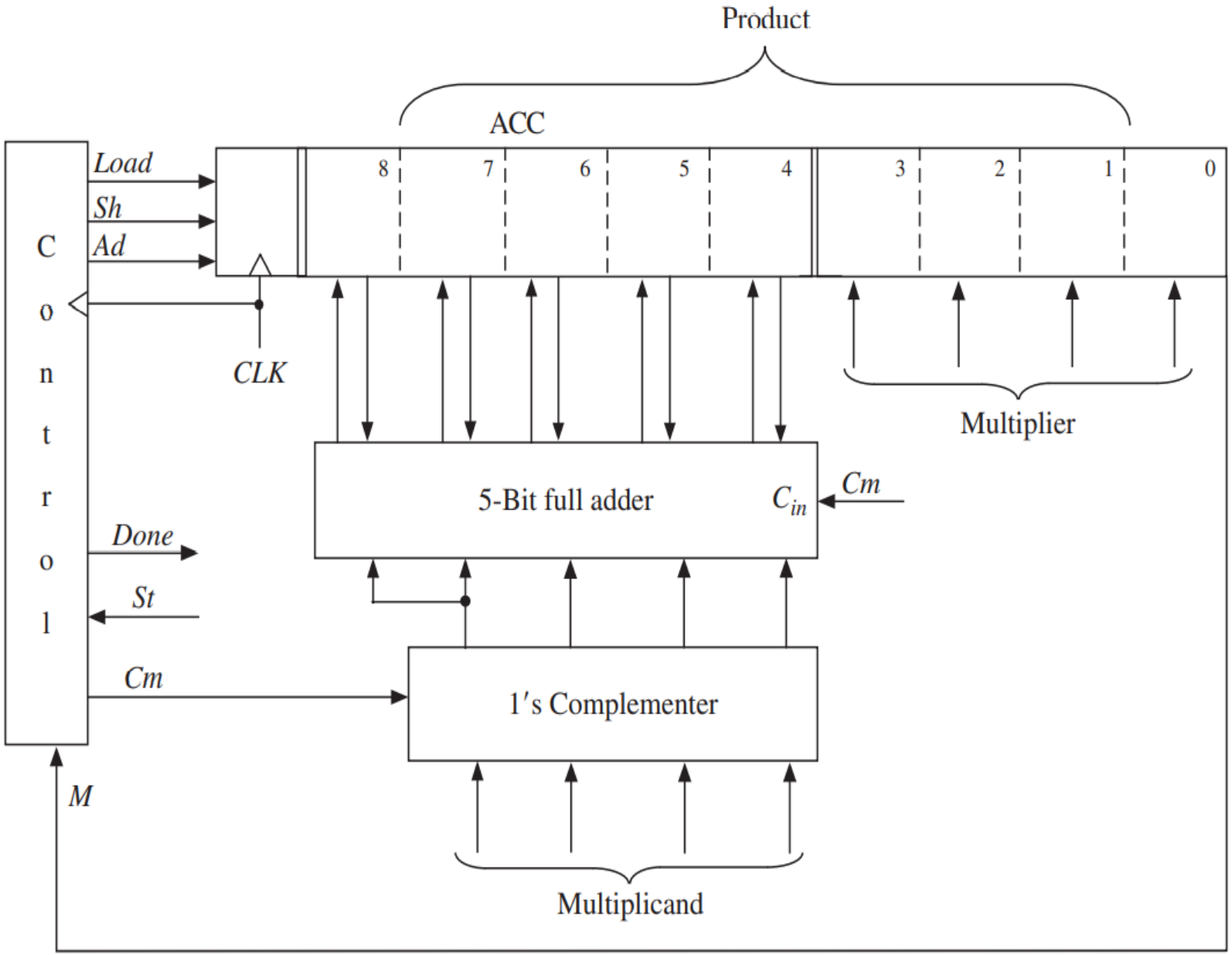
- When the multiplier is negative and the multiplicand is positive, we must make a slight change in the multiplication procedure.

0.1 0 1	(+5/8)		
× 1.1 0 1	(−3/8)		
(0.0 0)0 1 0 1	(+5/64)		
(0.)0 1 0 1	(+5/16)		
(0.)0 1 1 0 0 1			
1. 0 1 1	(−5/8)	←	*Note*: The 2's complement of the multiplicand is added at this point.
1. 1 1 0 0 0 1	(−15/64)		

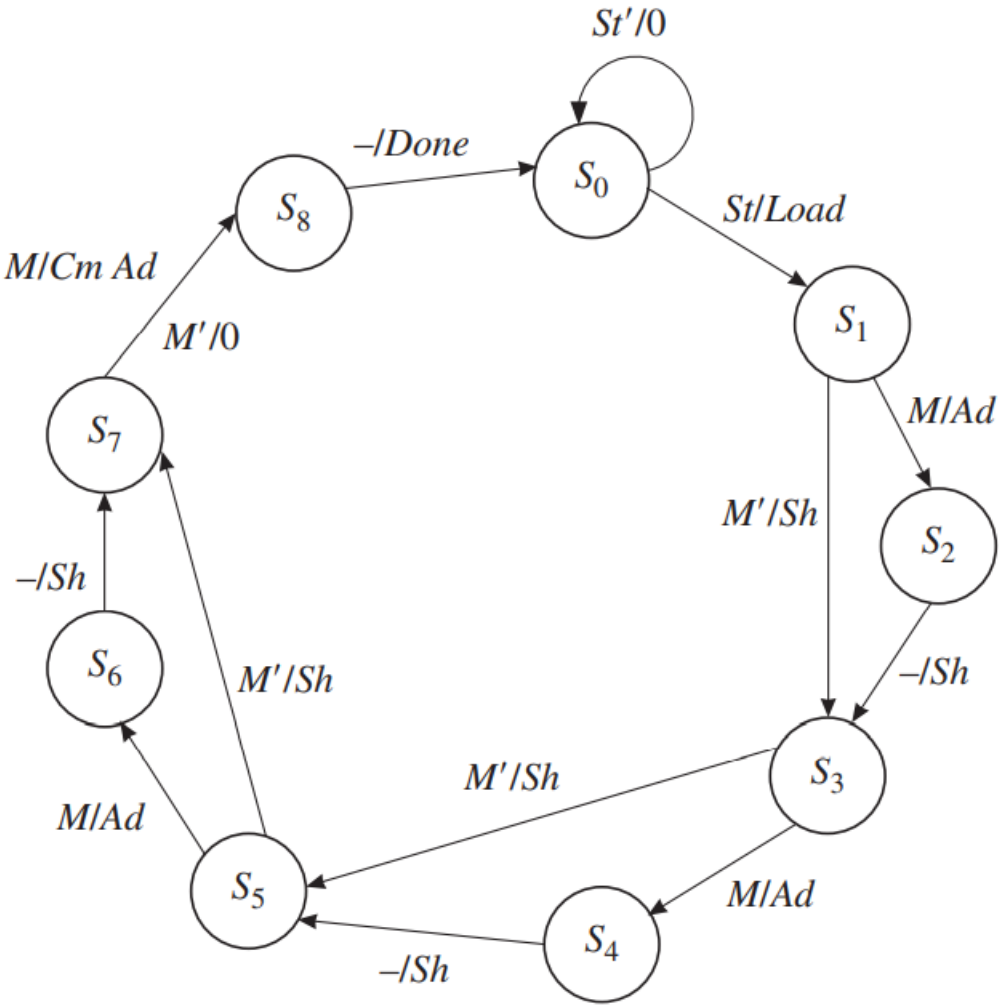
- When both the multiplicand and the multiplier are negative, the procedure is the same as before. At each step, we must be careful to extend the sign bit of the partial product to preserve the proper negative sign, and at the final step we must add in the 2's complement of the multiplicand, since the sign bit of the multiplier is negative. For example,

```
      1.1 0 1      (−3/8)
    × 1.1 0 1      (−3/8)
-------------------------
(1. 1 1) 1 1 0 1   (−3/64)   ← Note: Extend sign bit
(1.)1 1  0 1       (−3/16)
-------------------------
 1. 1 1  0 0 0 1
 0. 0 1 1          (+3/8)    ← Add the 2's complement of the
-------------------------       multiplicand.
 0. 0 0  1 0 0 1   (+9/64)
```

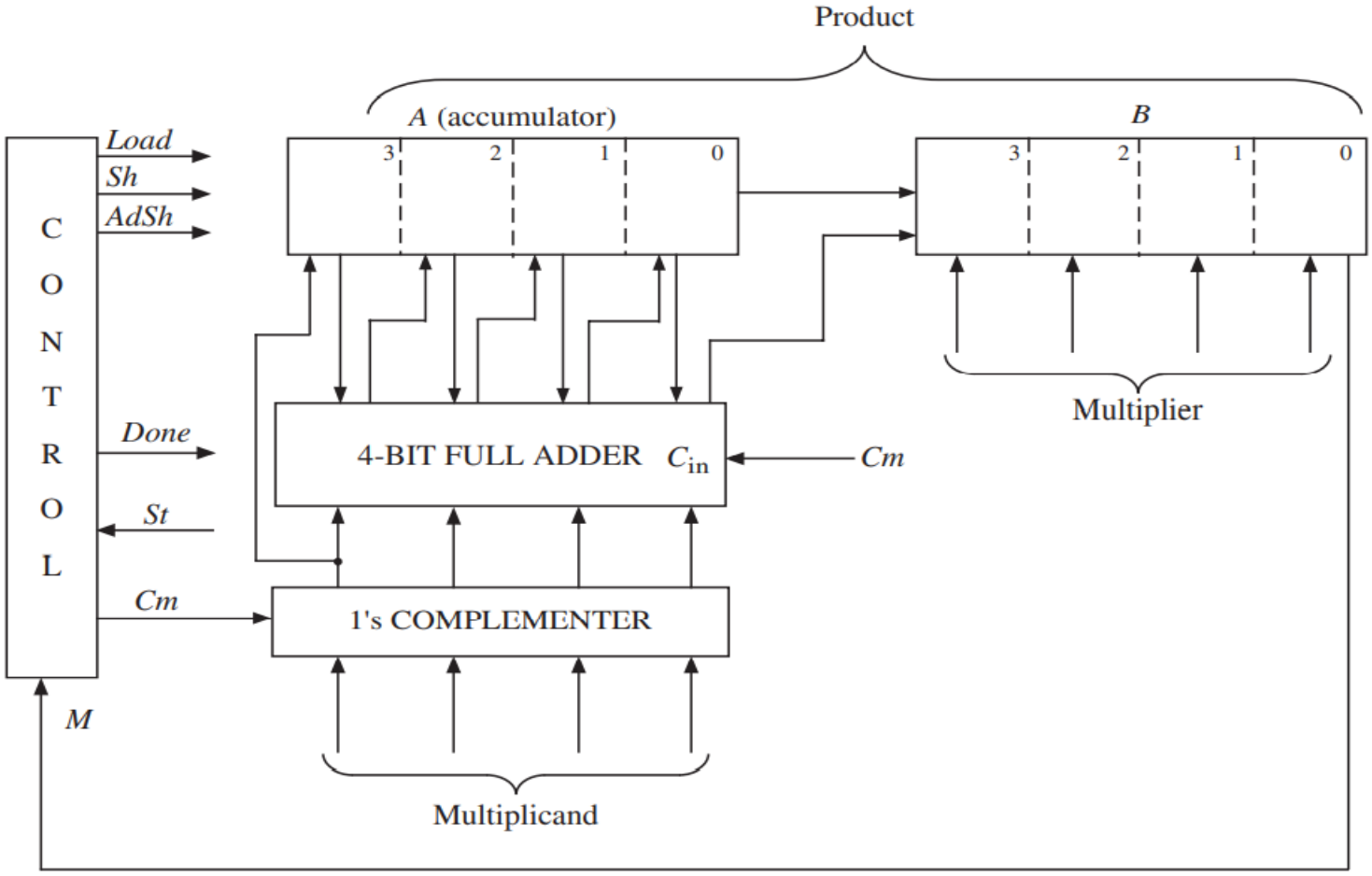
- In summary, the procedure for multiplying signed 2's complement binary fractions is the same as for multiplying positive binary fractions, except that we must be careful to preserve the sign of the partial product at each step, and if the sign of the multiplier is negative, we must complement the multiplicand before adding it in at the last step. The hardware is almost identical to that used for multiplication of positive numbers, except a complementer must be added for the multiplicand.
- Figure shows the hardware required to multiply two 4-bit fractions (including the sign bit).
- A 5-bit adder is used so the sign of the sum is not lost due to a carry into the sign bit position.
- The M input to the control circuit is the currently active bit of the multiplier. Control signal Sh causes the accumulator to shift right one place with sign extension. Ad causes the ADDER output to be loaded into the left 5 bits of the accumulator.
- The carry out from the last bit of the adder is discarded, since we are doing 2's complement addition. Cm causes the multiplicand (Mcand) to be complemented (1's complement) before it enters the adder inputs. Cm is also connected to the carry input of the adder so that when Cm = 1, the adder adds 1 plus the 1's complement of Mcand to the accumulator, which is equivalent to adding the 2's complement of Mcand.


Product
ACC
8
7
6
5
4
3
2
1
0
Load
Sh
Ad
C
o
n
t
r
o
l
CLK
Multiplier
5-Bit full adder
C_{in}
Cm
Done
St
Cm
1′s Complementer
M
Multiplicand

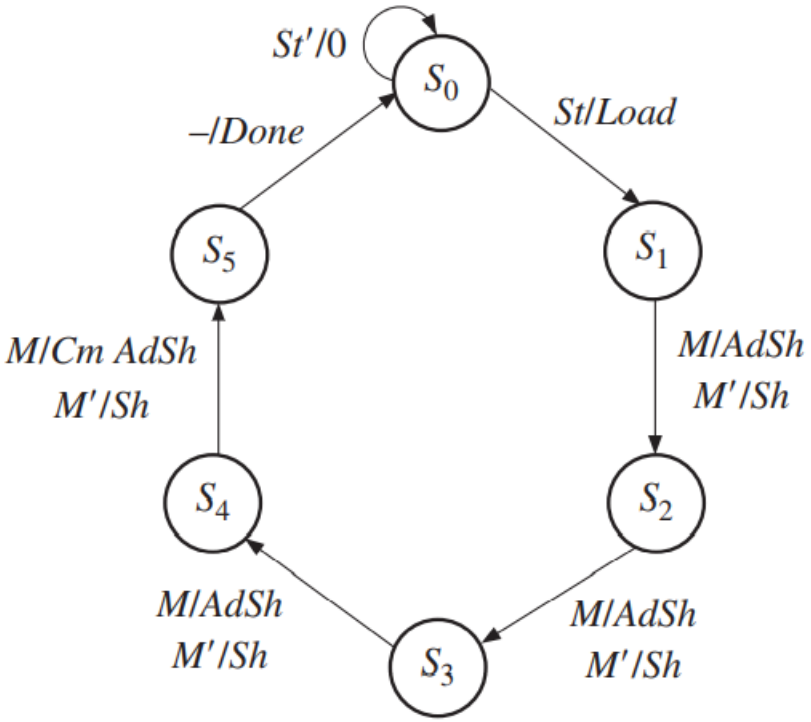
- Figure shows a state graph for the control circuit. Each multiplier bit (M) is tested to determine whether to add and shift or whether to just shift. In state S7 , M is the sign bit, and if M = 1, the complement of the multiplicand is added to the accumulator.



- Faster multiplier: When the hardware shown in the above Figure, the add and shift operations must be done at two separate clock times. We can speed up the operation of the multiplier by moving the wires from the adder output one position to the right so that the adder output is already shifted over one position when it is loaded into the accumulator.



State Graph for Faster Multiplier



A behavioral Verilog model for this multiplier is shown below. Shifting the A and B registers together is accomplished by the sequential statements

```
A <= {A[3], A[3:1]} ;
B <= {A[0], B[3:1]} ;
```

```
`define M B[0]

module mult2C (CLK, St, Mplier, Mcand, Product, Done);

   input CLK;
   input St;
   input[3:0] Mplier;
   input[3:0] Mcand;
   output[6:0] Product;
   output Done;

   reg[2:0] State;
   reg[3:0] A;
   reg[3:0] B;
   reg[3:0] addout;


   initial
   begin
       State = 0;
   end
```

```
always @(posedge CLK)
begin
        case (State)
            0 :
                        begin
                            if (St == 1'b1)
                            begin
                                A <= 4'b0000 ;
                                B <= Mplier ;

                    State <= 1 ;
                end
                else
                    State <= 0;
            end
1, 2, 3 :
            begin
                if (`M == 1'b1)
                begin
                    addout = A + Mcand;
                    A <= {Mcand[3], addout[3:1]} ;
                    B <= {addout[0], B[3:1]} ;
                end
```

```
            else
            begin
                A <= {A[3], A[3:1]} ;
                B <= {A[0], B[3:1]} ;
            end
            State <= State + 1 ;
        end
4 :
        begin
            if (`M == 1'b1)
            begin
                addout = A + ~Mcand + 1;
                A <= {~Mcand[3], addout[3:1]} ;
                B <= {addout[0], B[3:1]} ;
            end
            else
            begin
                A <= {A[3], A[3:1]} ;
                B <= {A[0], B[3:1]} ;
            end
            State <= 5 ;
        end
```

```
                    end
            5 :
                    begin
                      State <= 0 ;
                    end

        default :
                    begin
                      State <= 0 ;
                    end

            endcase
            end

    assign Done = (State == 5) ? 1'b1 : 1'b0 ;
    assign Product = {A[2:0], B} ;

endmodule
```