DESIGN EXAMPLES

Objectives

Several Verilog design examples is presented to illustrate the design of small digital systems.

- Introduction
- BCD to 7-Segment Display Decoder
- A BCD Adder
- 32-Bit Adders
- State Graphs for Control Circuits
- Synchronization and Debouncing
- A Shift-and-Add Multiplier
- Array Multiplier
- A Signed Integer/Fraction Multiplier
- Keypad Scanner
- Binary Dividers
- Summary

Introduction

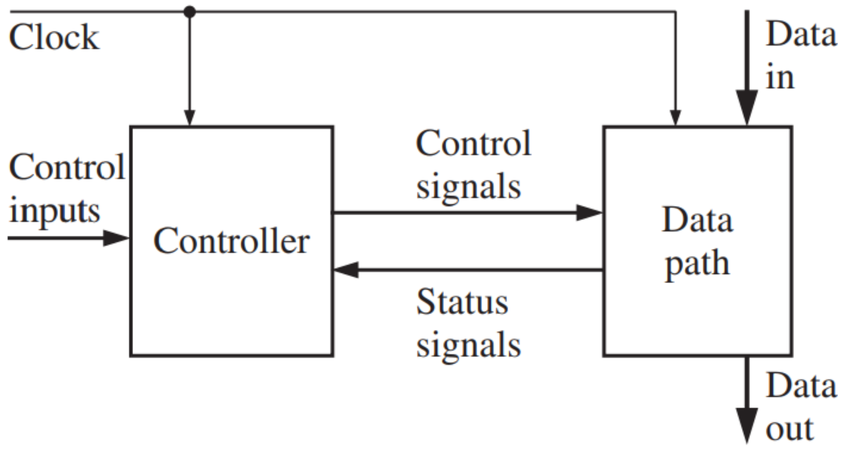
- In any design:
  - State the problem.
  - Obtain clear design specifications.
  - Define basic building blocks necessary to accomplish what is specified, e.g. adders, shift registers, counters, and so on.

Introduction (continued)

- Traditional design methodology splits a design into :
  - Data path: hardware that actually performs the data processing.
    - For microprocessor: the arithmetic and logic unit (ALU)
  - Controller: sends control signals or commands to the data path and can obtain feedback in the form of status signals from the data path.
    - Many modifications can be made here.

Introduction (continued)

Separation of a Design into Data Path and Controller

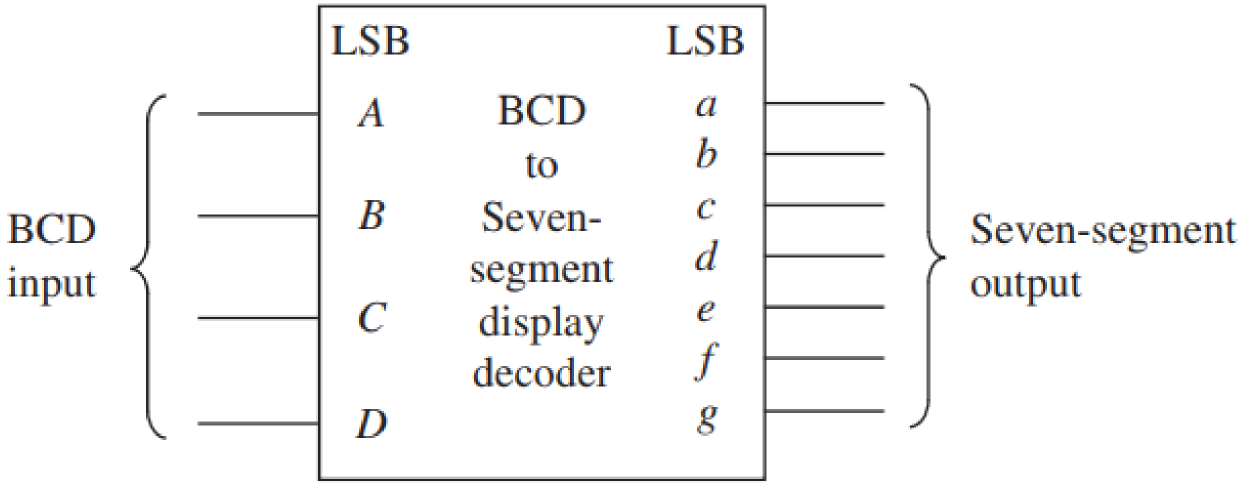


In the context of a microprocessor, the data path is the ALU (Arithmetic and Logic Unit) that performs the core of the processing. The controller is the control logic that sends appropriate control signals to the data path, instructing it to perform addition, multiplication, shifting, or whatever action is called for by the instruction. Many users have a tendency to mistakenly consider the term “data path” to be synonymous with the data bus, but “data path” in traditional design terminology refers to the actual data processing unit.

- Various design examples such as <u>arithmetic</u> and <u>non-arithmetic</u> circuits will be discussed.

BCD to 7-Segment Display Decoder

- Binary coded decimal (BCD): each decimal digit is encoded into binary.
- Seven segment displays are often used to display digits in digital counters, watches, and clocks.
- Block diagram of a BCD to 7-segment display decoder:

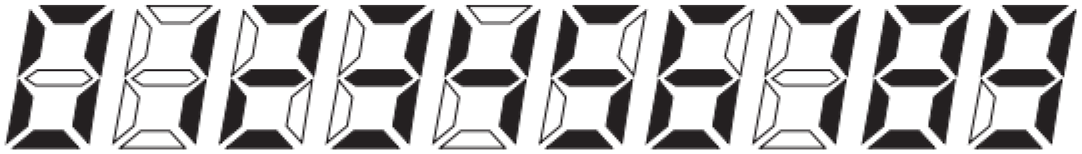


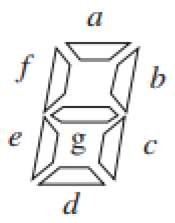
BCD to 7-Segment Display Decoder (continued)

- In this format, each digit of a decimal number is encoded into 4-bit binary representation. This decoder is a purely combinational circuit; hence, no state machine is involved here.
- One can create a behavioral Verilog architectural description by using a single process with a case statement to model this combinational circuit.

BCD to 7-Segment Display Decoder (continued)

```
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
```





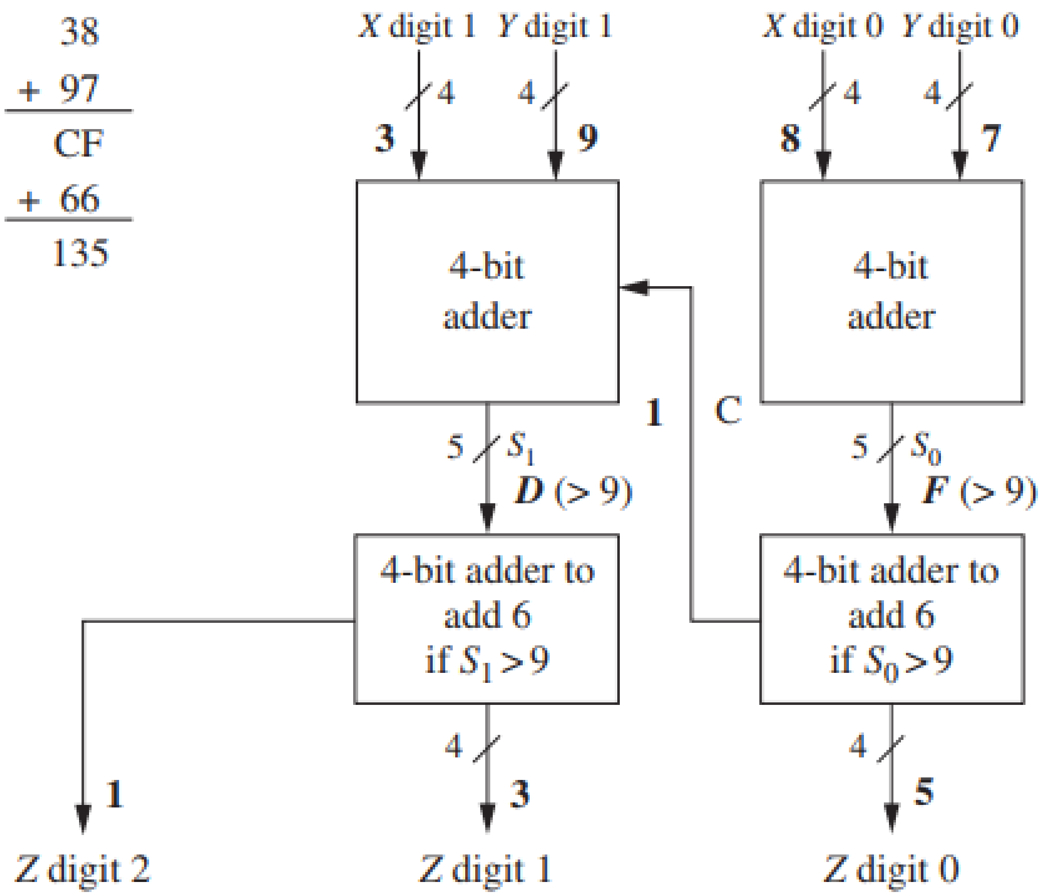
BCD Adder

- In BCD representation:
  - A to F are not used. As 6 out of 16 representations possible with 4 binary bits are skipped, a BCD number will take more bits than the corresponding binary representation.
- When BCD numbers are added:
  - Each sum digit should be adjusted to skip the six unused codes.
    - 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.

BCD Adder (continued)

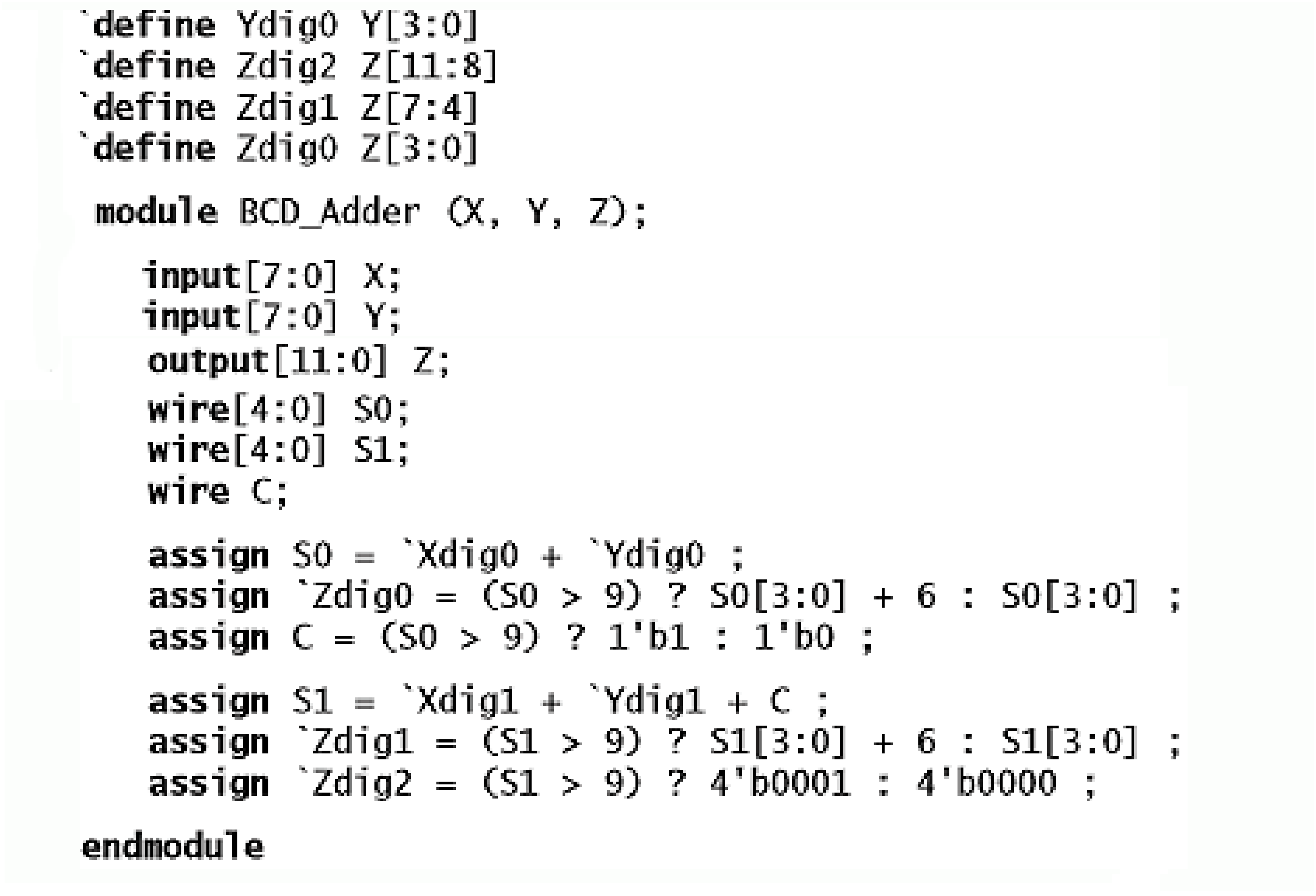
- Addition of 2 BCD numbers:

```
   38
 + 97
 ----
   CF
 + 66
 ----
  135
```



BCD Adder (continued)

- Verilog code for BCD adder:



```
`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

- 32-Bit Adders:
  - Ripple-carry adder:
    - 32 copies of one-bit adder are connected in succession.
    - The carry "ripples" from the least significant bit to the most significant bit.
    - If gate delays are t_g , a one bit adder delay is $2*t_g$
    - A 32 bit ripple-adder will take 64 gate delays and if gate delays are 1 ns, then max frequency of operation is 16 MHz. Hence, many use faster adders.

32-Bit Adders (continued)

- Carry look-ahead adder:
  - Carry signals are calculated in advance, based on the input signals.
  - For any stage *i*, the carry-out is: $C_{i+1} = A_i B_i + (A_i \oplus B_i) \bullet C_i$
  - Carry generate (G_i): $G_i = A_i B_i$
  - Carry propogate (P_i): $P_i = A_i \oplus B_i$ or $P_i = A_i + B_i$
  - Sum signal: $S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i$
  - Carry-out equation can be rewritten by substituting the carry generate (G_i) and carry propagate (P_i) equations. Thus:
    - In a 4-bit adder, the C_i's can be generated by repeating the equation above.

$$C_{i+1} = G_i + P_i C_i$$

32-Bit Adders (continued)

– 4-bit carry look-ahead adder:

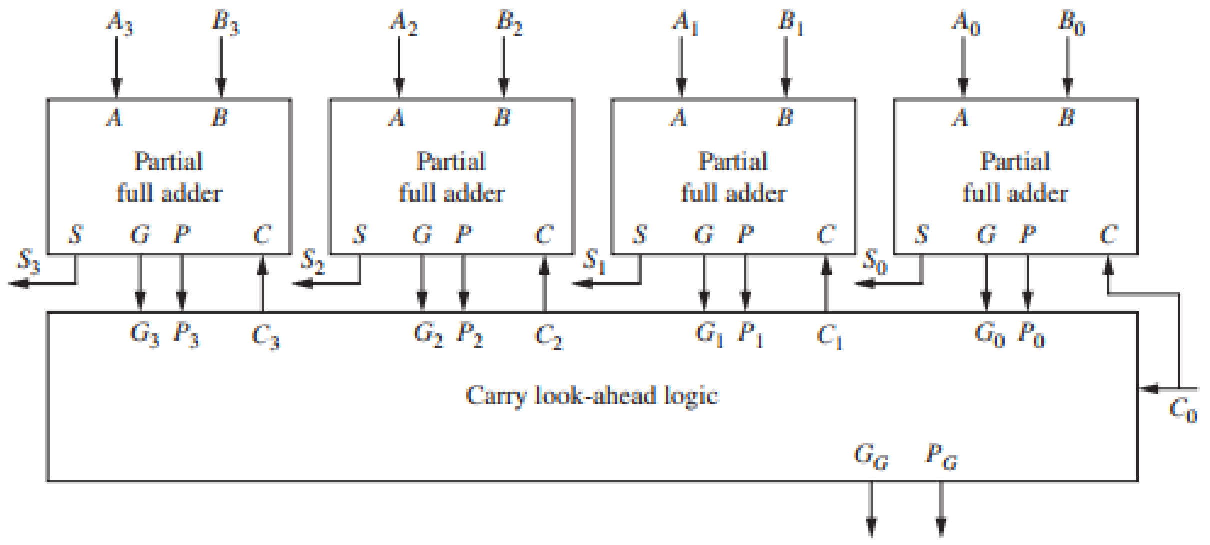
$$C_1 = G_0 + P_0C_0$$

$$C_2 = G_1 + P_1C_1 = G_1 + P_1G_0 + P_1P_0C_0$$

$$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$$

$$C_4 = G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$$

$C_4 = G_G + P_GC_0$ $\quad P_G = P_3P_2P_1P_0$ $\quad G_G = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0$



gets quite complicated for more than 4 bits. For that reason, carry look-ahead adders are usually implemented as 4-bit modules and are used in a hierarchical structure to realize adders that have multiples of 4 bits.

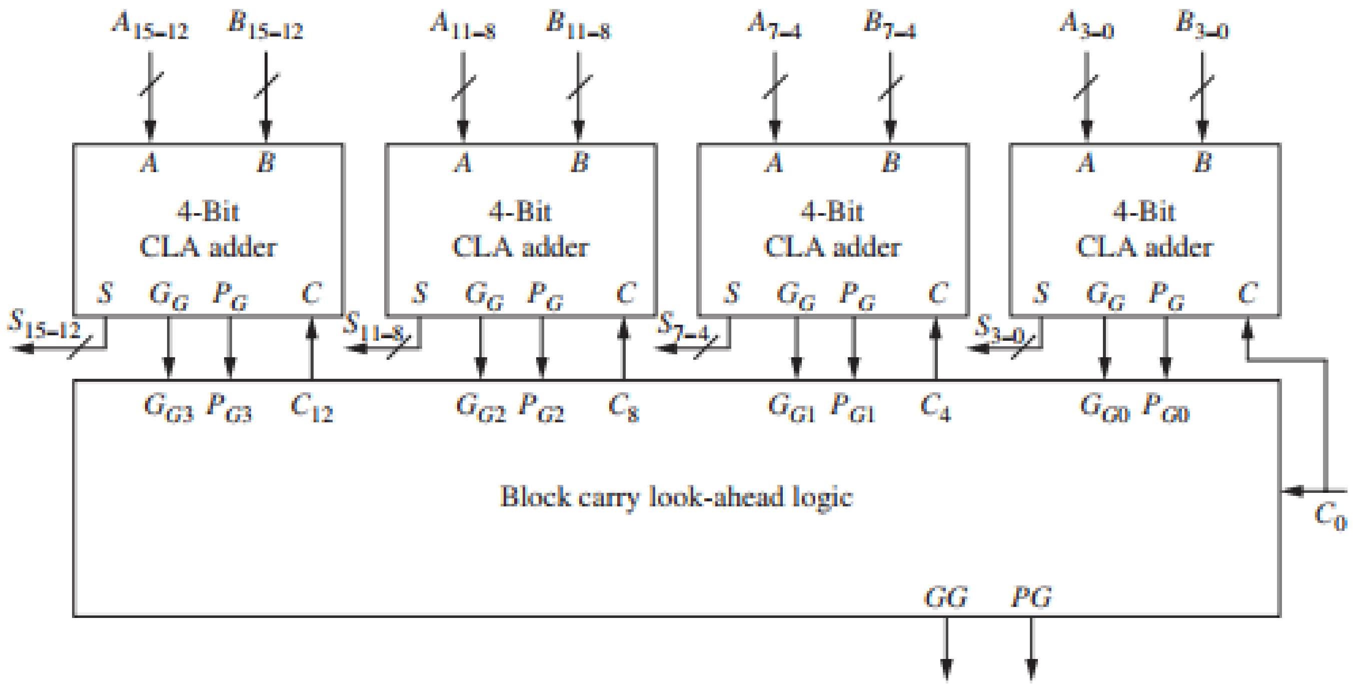
32-Bit Adders (continued)

– Disadvantage of carry look-ahead adder:
  - Logic gets quite complicated for more than 4 bits. For that reason, carry look-ahead adders are usually implemented as 4-bit modules and are used in a hierarchical structure to realize adders that have multiples of 4 bits.

– Block carry look-ahead logic: generates input carry bits to be fed to each 4-bit adder using a group propagate (P_G) and group generate (G_G) signal, which is produced by each 4-bit adder. Next level uses P_G and G_G and generates the required carry bits in parallel.

32-Bit Adders (continued)

- The group propagate P_G and generate G_G will be available after 3 and 4 gate delays (1 or 2 additional delays than the P_i and G_i signals).



32-Bit Adders (continued)

- Equations for block carry look-ahead:

$$C_4 = G_{G0} + P_{G0}C_0$$
$$C_8 = G_{G1} + P_{G1}G_{G0} + P_{G1}P_{G0}C_0$$
$$C_{12} = G_{G2} + P_{G2}G_{G1} + P_{G2}P_{G1}G_{G0} + P_{G2}P_{G1}P_{G0}C_0$$

- C_{16}, which is a final carry of 16-bit CLA is:

$$C_{16} = GG + PGC_0$$

- Verilog code for a 16-bit carry look-ahead adder can be done by instantiating 4 copies of the 4-bit carry look ahead adder and 1 more copy of the carry look-ahead logic.

```
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
```

32-Bit Adders (continued)

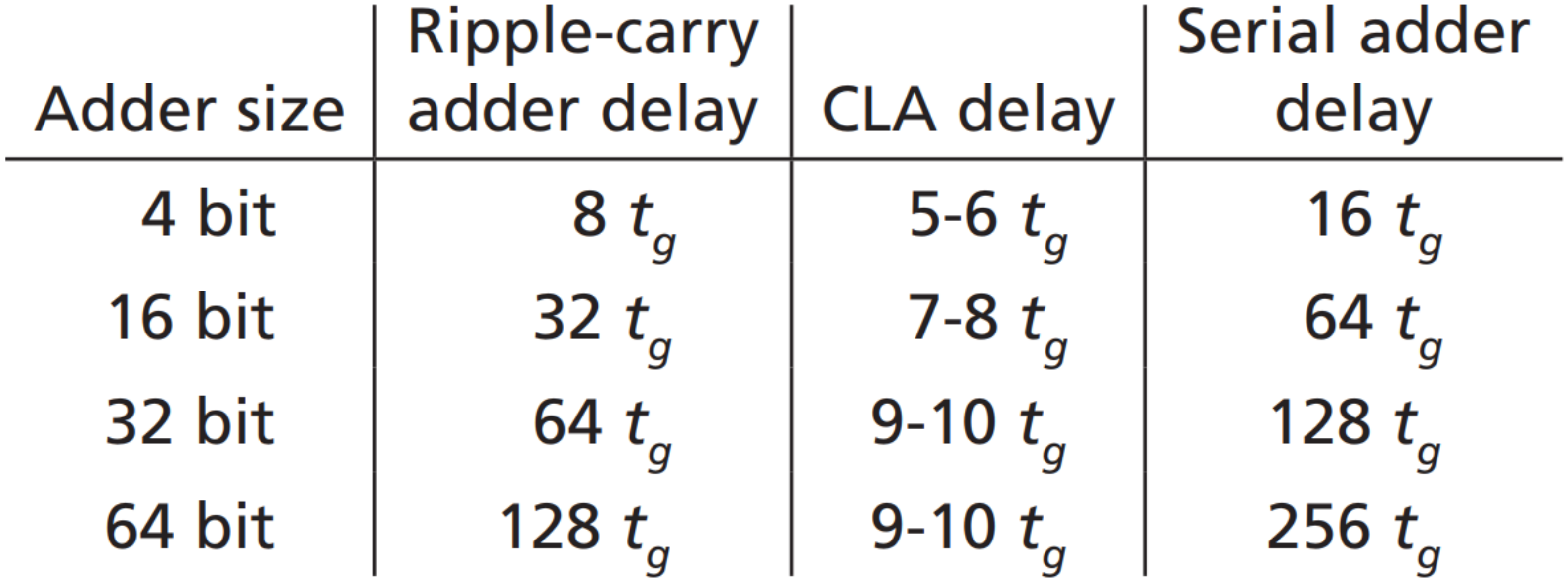
- Example: If gate delays are t_g, what is the delay of the fastest 32-bit adder? Assume that the amount of hardware consumed is not a constraint. Only speed is important.

32-Bit Adders (continued)

- Answer: Although it is not very practical, theoretically, the delay of the fastest adder will be 2 t_g if gate delays are t_g .

32-Bit Adders (continued)

- Delays of a ripple-carry, carry look-ahead, and serial adder for a gate-based implementation:



Adder size	Ripple-carry adder delay	CLA delay	Serial adder delay
4 bit	8 t_g	5-6 t_g	16 t_g
16 bit	32 t_g	7-8 t_g	64 t_g
32 bit	64 t_g	9-10 t_g	128 t_g
64 bit	128 t_g	9-10 t_g	256 t_g

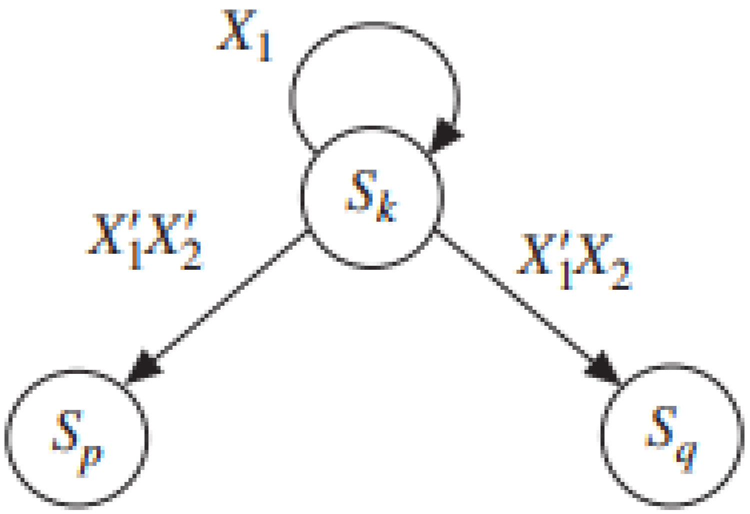
State Graphs for Control Circuits

- Notations and conditions used on control state graphs:
  - Variable names are used, not 0's and 1's
  - Label of arc on a Mealy state graph $X_i\ X_j\ Z_p\ Z_q$: means if inputs $X_i\ X_j$ are 1, the outputs $Z_p\ Z_q$ are 1 (other outputs are 0). Then traverse arc to next state.
  - for a circuit with four inputs ($X1$, $X2$, $X3$, $X4$) and four outputs ($Z1$, $Z2$, $Z3$, $Z4$), the label $X1X4'/Z2Z3$ is equivalent to 1–0/0110.

State Graphs for Control Circuits (continued)

- To ensure next states are uniquely defined for inputs, must have constraints on the input labels for every state S_k:
  - 1. If I_i and I_j are any pair of input labels on arcs exiting state S_k, then $I_i I_j = 0$ if $i \neq j$.
    - At most, one input label can be 1 at any given time.
  - 2. If *n* arcs exit state S_k and the *n* arcs have input labels $I_1, I_2, \ldots, I_n$, respectively, then $I_1+I_2+\ldots+I_n = 1$.
    - At least one input label will be 1 at any given time.

State Graphs for Control Circuits (continued)

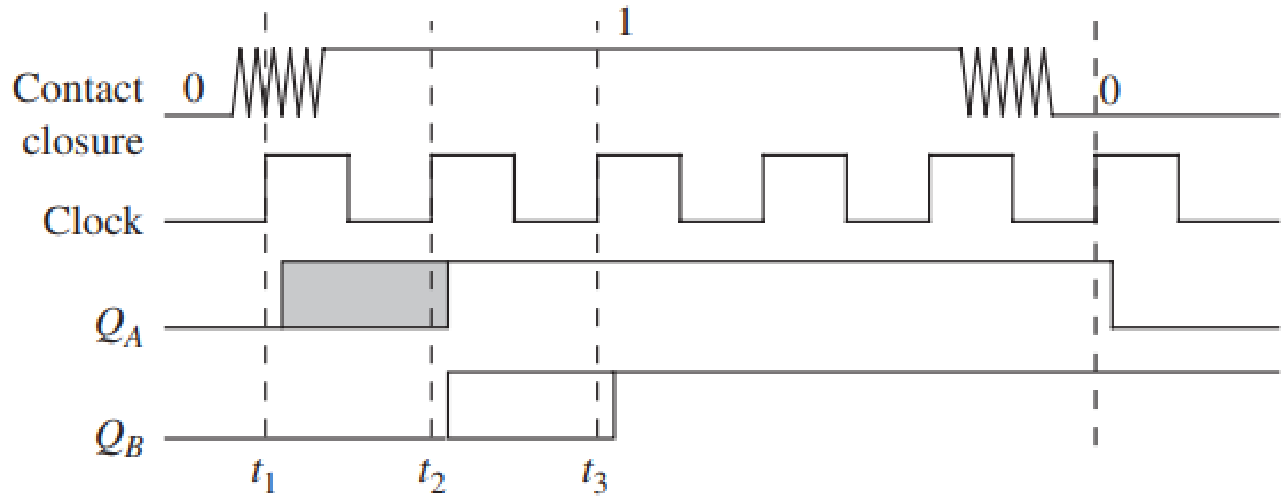


$$(X_1)\ (X_1'X_2') = 0$$
$$(X_1)\ (X_1'X_2) = 0$$
$$(X_1'X_2')\ (X_1'X_2) = 0$$
$$X_1 + X_1'X_2' + X_1'X_2 = 1$$

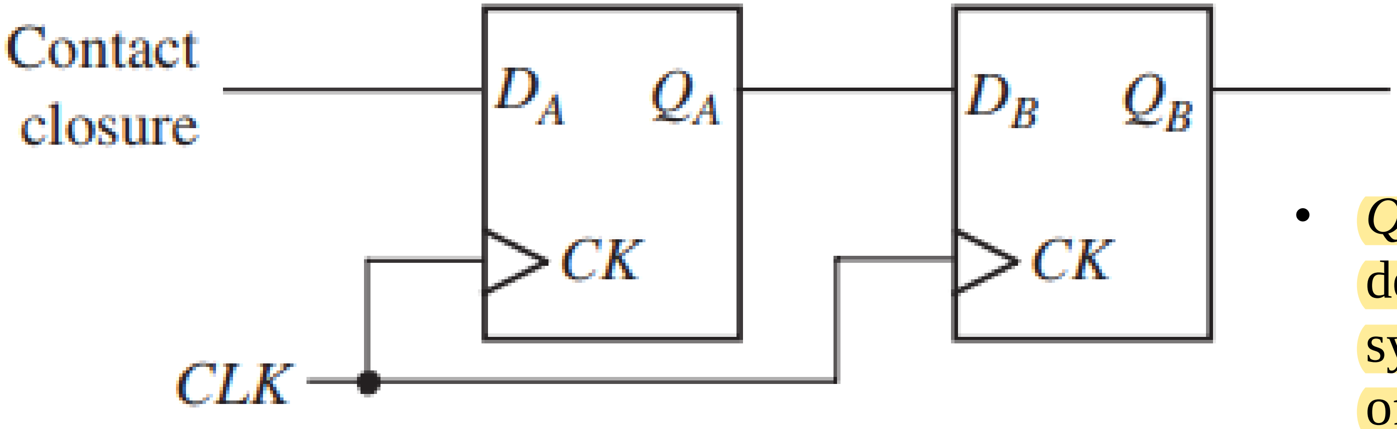
Synchronization and debouncing

- Issues in systems with external inputs:
  - Synchronization: outputs from a keypad or push button switches are not synchronous to the system clock signal.
  - Switch bounce: when a mechanical switch is closed or opened, the switch contact will bounce, causing noise in the switch output. After switch closure, must wait for the bounce to settle before reading the key (several milliseconds).
- Flip-flops are very useful devices when contacts must be synchronized and debounced.

Synchronization and debouncing



Debouncing and synchronizing circuit



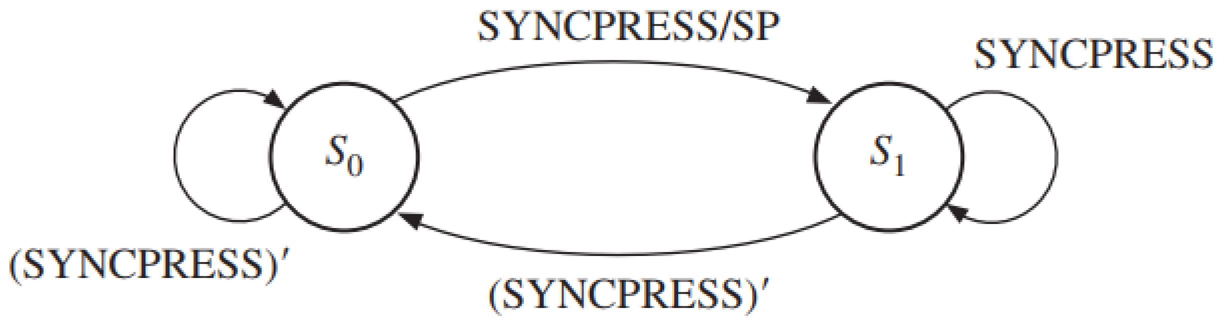
- *QA* will be a debounced and synchronized version of the contact closure.

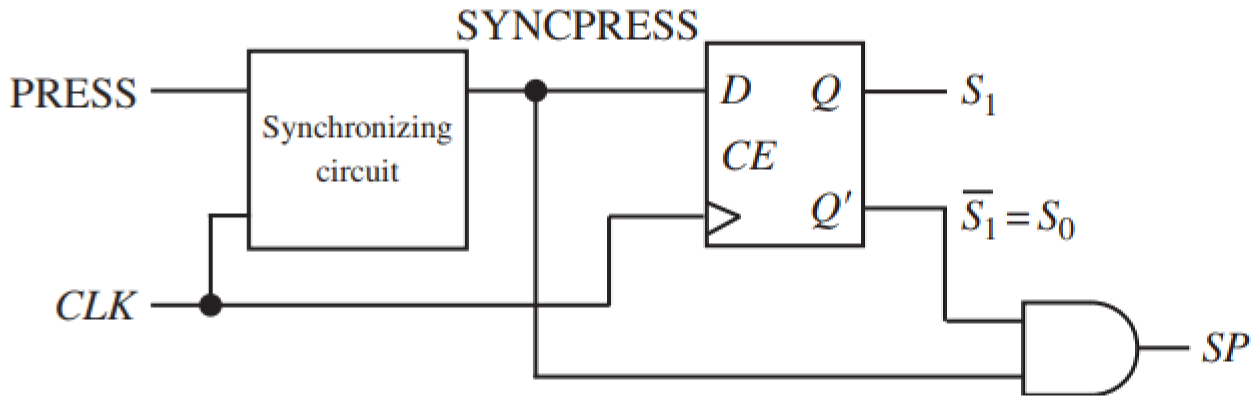
However, a possibility of failure exists if *the switch* changes very close to the clock edge such that the setup or hold time is violated. In this case the flip-flop output *QA* may oscillate or otherwise malfunction.

Synchronization and debouncing (continued)

- Single pulser:
  - Digital systems generally run at speeds higher than actions by humans and practically, It is difficult for humans to produce a signal that lasts only for a clock pulse.
  - Solution: develop a circuit that generates a single pulse for a human action of pressing a button or switch. Can be used in many applications.
  - State diagram with 2 states:

State Diagram of Single Pulse





Synchronization and debouncing (continued)

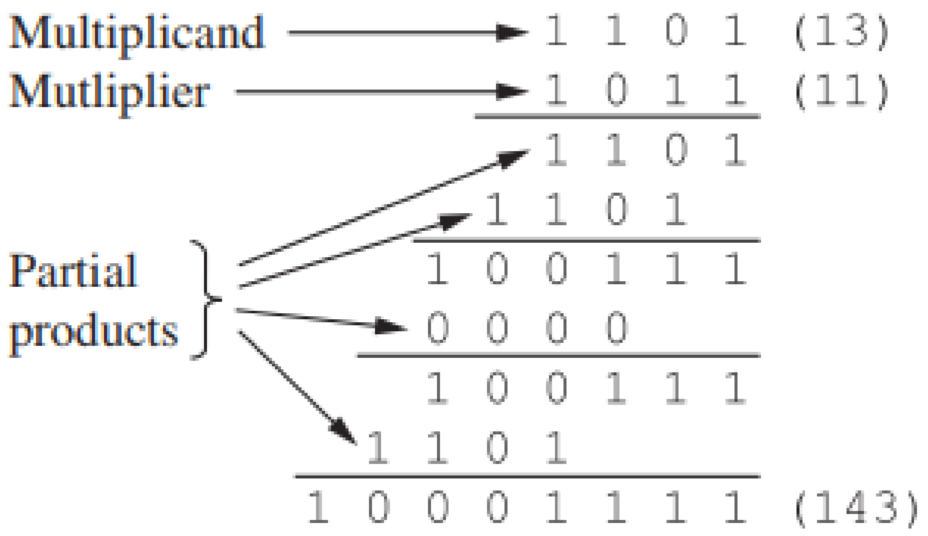
- Since there are only two states for this circuit, it can be implemented using one flip-flop. Flip-flop implements the two states of the state machine.
- Equation for single pulse:
  $SP = S0 \cdot SYNCPRESS$

A Shift-and-Add Multiplier

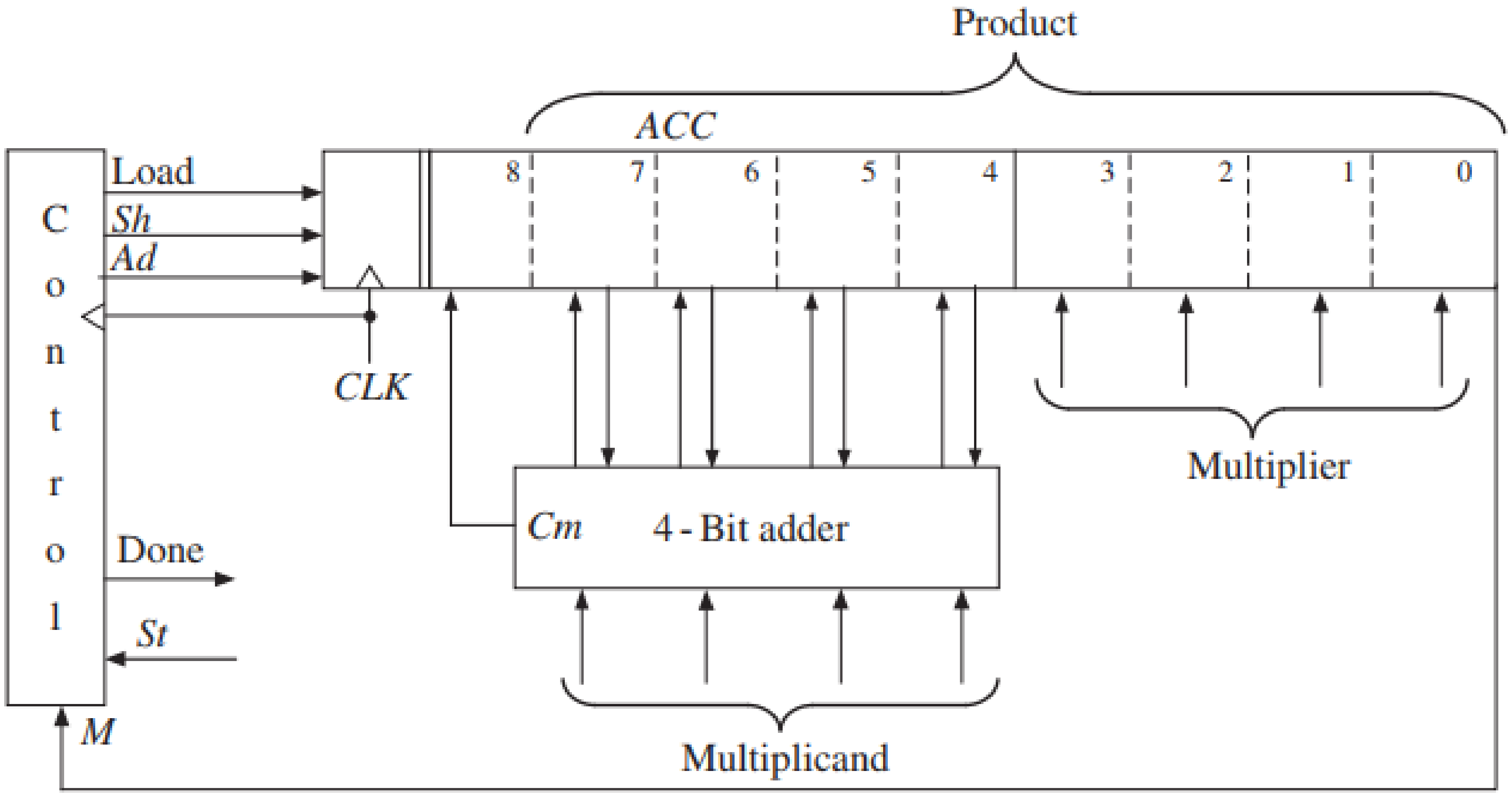
- we will design a multiplier for unsigned binary numbers.
- In A × B, the first operand (A) is called the multiplicand and the second operand (B) is called the multiplier.
- Binary multiplication process:
  - Multiplicand is shifted.
  - Next bit of multiplier is examined (also a shifting step).
  - If this bit is 1, shifted multiplicand is added to the product.

A Shift-and-Add Multiplier (continued)

- Example: multiply 13_{10} by 11_{10} in binary:



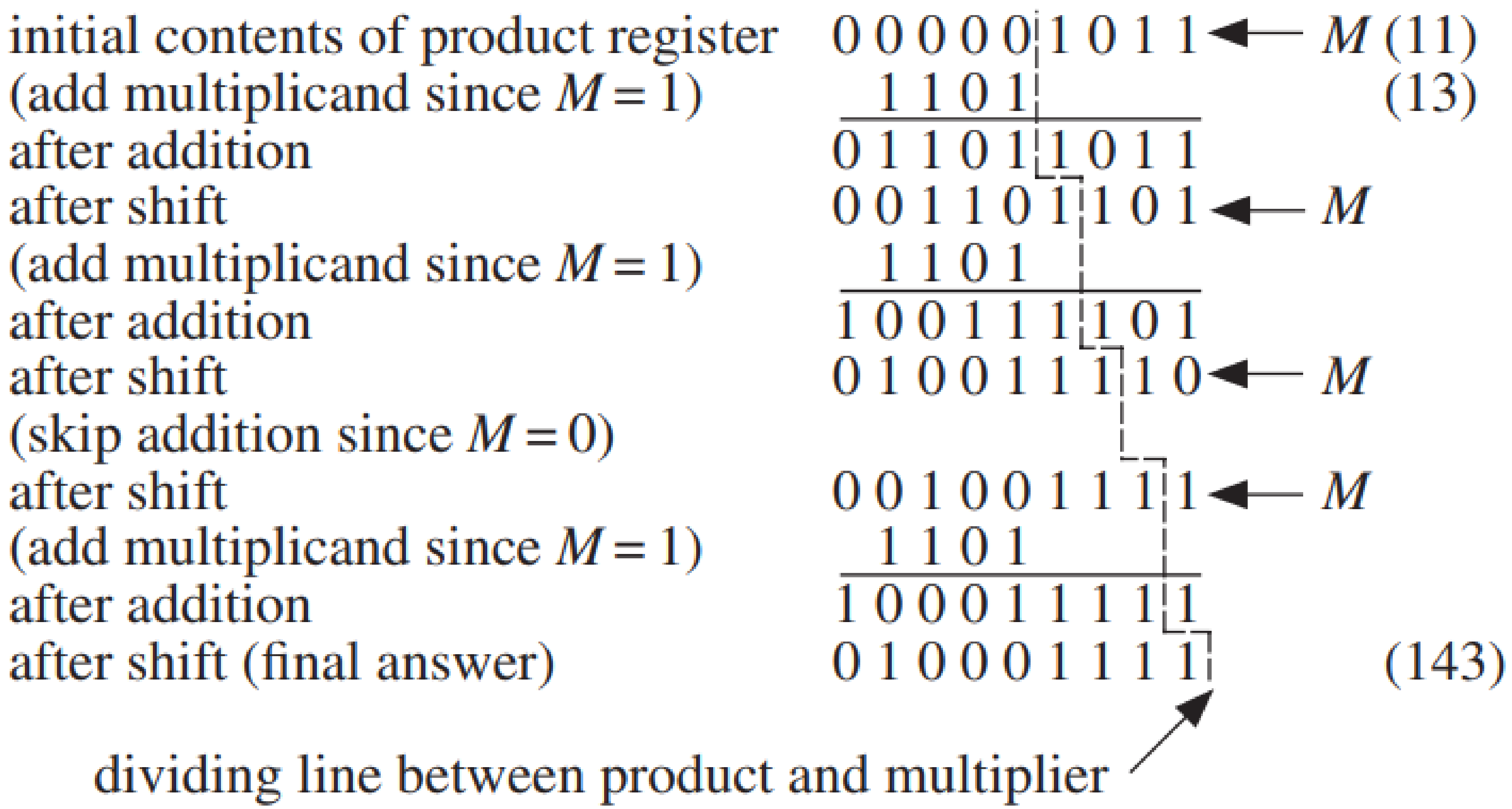
  – Sometimes called a serial-parallel multiplier as the multiplier bits are processed serially, but addition takes place in parallel.



Block Diagram for Binary Multiplier

A Shift-and-Add Multiplier (continued)

- Example reworked to show the location of the bits in the registers at clock time:

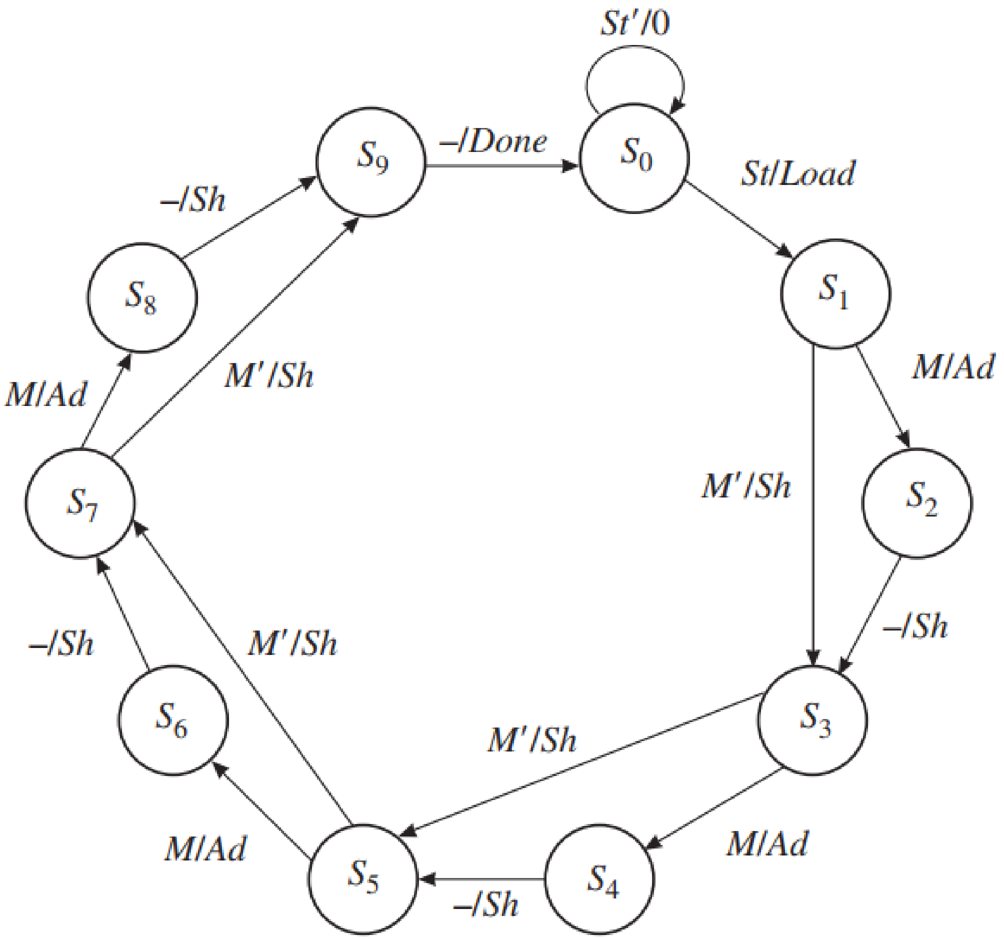


```
initial contents of product register   0 0 0 0 0|1 0 1 1 ←— M (11)
(add multiplicand since M = 1)           1 1 0 1|                (13)
                                       ___________________
after addition                         0 1 1 0 1|1 0 1 1
after shift                            0 0 1 1 0 1|1 0 1 ←— M
(add multiplicand since M = 1)           1 1 0 1
                                       ___________________
after addition                         1 0 0 1 1 1|1 0 1
after shift                            0 1 0 0 1 1 1|1 0 ←— M
(skip addition since M = 0)
after shift                            0 0 1 0 0 1 1 1|1 ←— M
(add multiplicand since M = 1)           1 1 0 1
                                       ___________________
after addition                         1 0 0 0 1 1 1 1|1
after shift (final answer)             0 1 0 0 0 1 1 1 1|        (143)
```

dividing line between product and multiplier

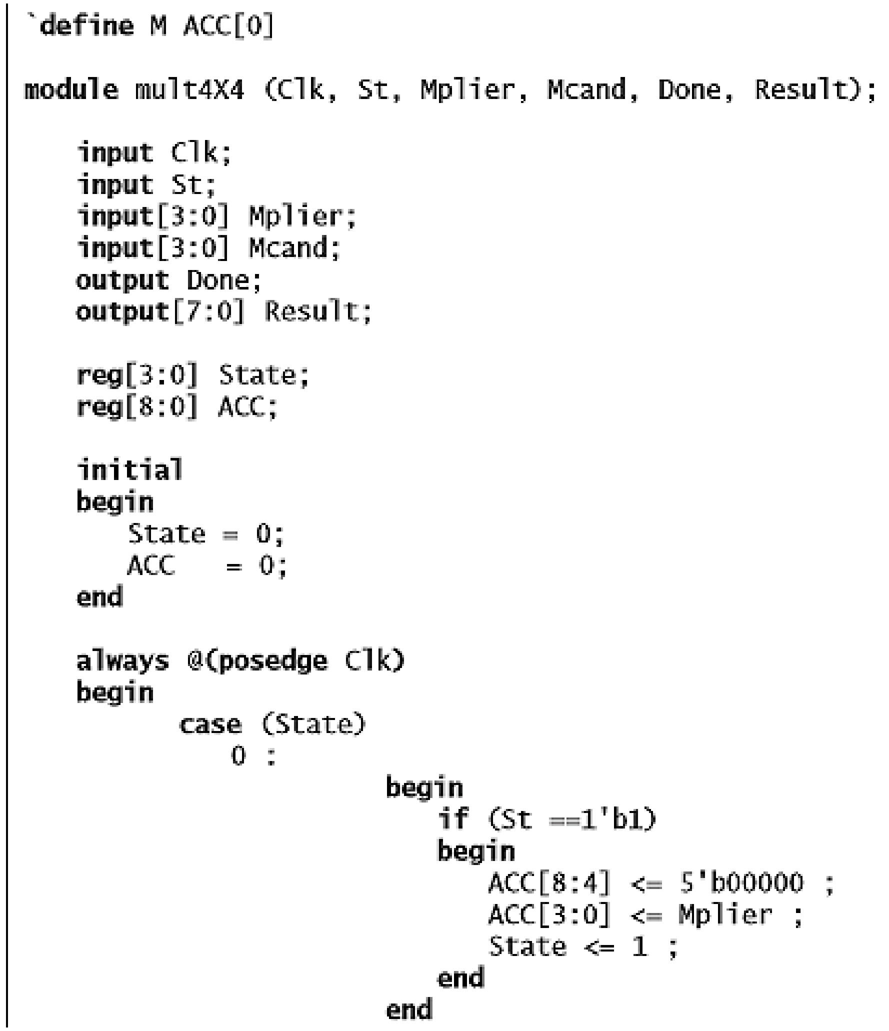
A Shift-and-Add Multiplier (continued)

- State graph for control circuit (designed to output the proper sequence of add and shift signals):



A Shift-and-Add Multiplier (continued)

- Behavioral model for a 4 x 4 binary multiplier:



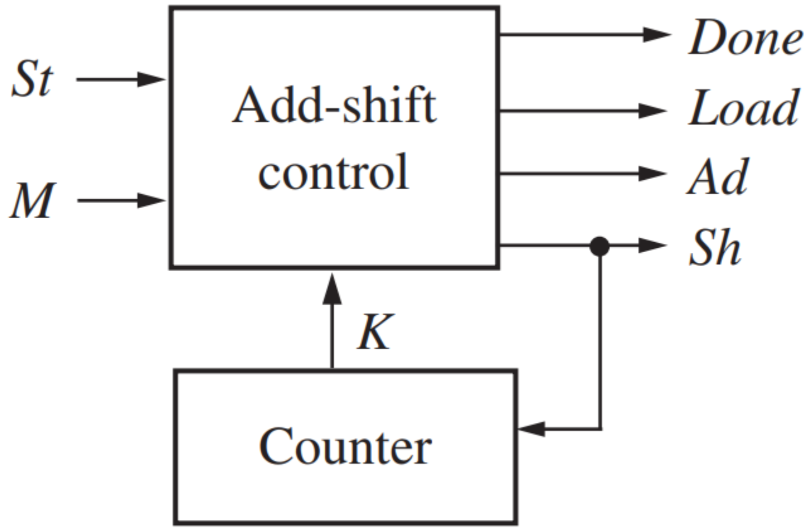
```
`define M ACC[0]

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
```

A Shift-and-Add Multiplier (continued)

```
          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
```

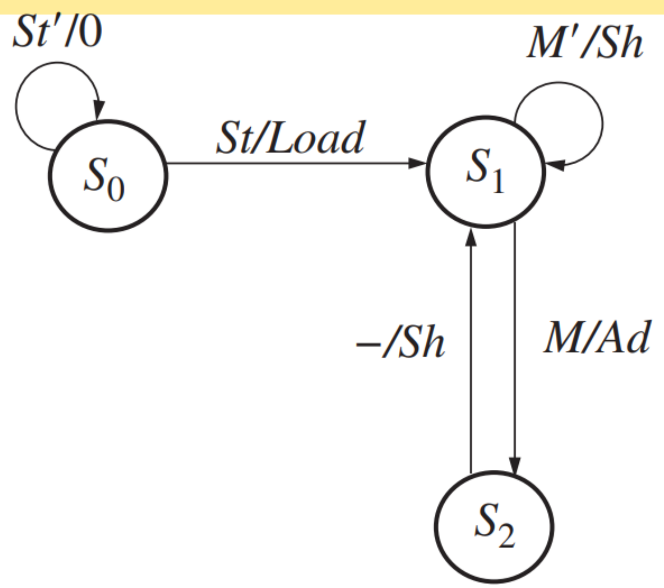
A Shift-and-Add Multiplier (continued)

- As the state graph for the multiplier indicates, the control performs 2 functions—generating add or shift signals as needed and counting the number of shifts. If the number of bits is large, it is convenient to divide the control circuit into a counter and an add-shift control:



A Shift-and-Add Multiplier (continued)

- First, derive a state graph for the add-shift control that tests St and M and outputs the proper sequence of add and shift signals.



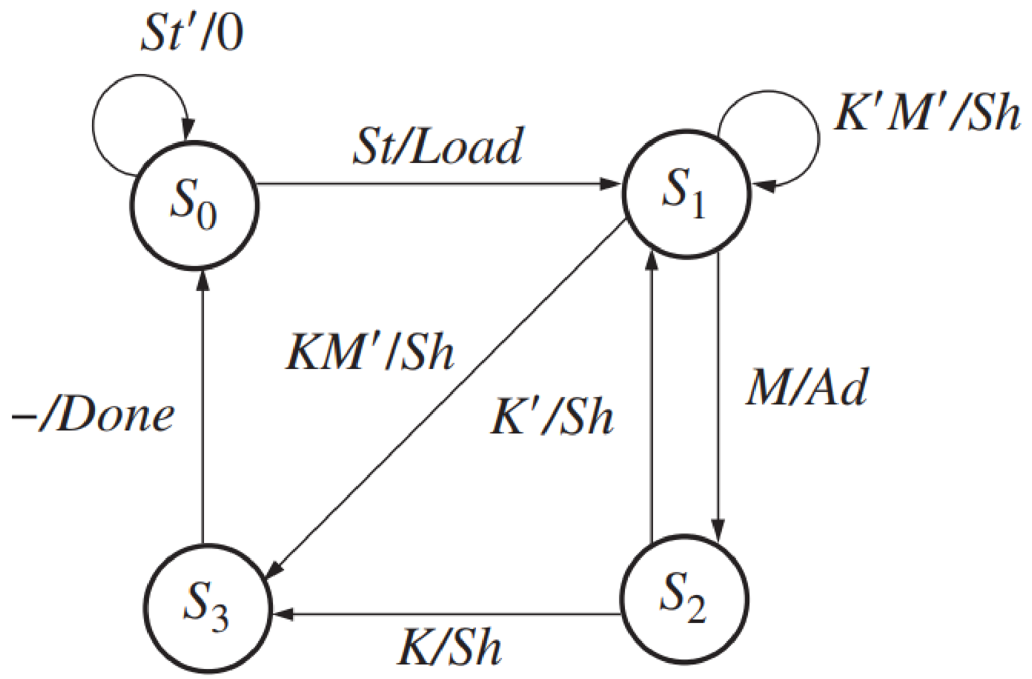
- Then we will add a completion signal (K) from the counter that stops the multiplier after the proper number of shifts have been completed.

A Shift-and-Add Multiplier (continued)

- To determine when the multiplication is completed, the counter is incremented each time a shift signal is generated. If the multiplier is n bits, n shifts are required. Design the counter so that a completion signal (K) is generated after n - 1 shifts have occurred.

A Shift-and-Add Multiplier (continued)

- When K = 1, the circuit should perform one more addition, if necessary, and then do the final shift.
- Final state graph for add-shift control:



Operation of Multiplier Using a Counter

Time	State	Counter	Product Register	St	M	K	Load	Ad	Sh	Done
t_0	S_0	00	000000000	0	0	0	0	0	0	0
t_1	S_0	00	000000000	1	0	0	1	0	0	0
t_2	S_1	00	000001011	0	1	0	0	1	0	0
t_3	S_2	00	011011011	0	1	0	0	0	1	0
t_4	S_1	01	001101101	0	1	0	0	1	0	0
t_5	S_2	01	100111101	0	1	0	0	0	1	0
t_6	S_1	10	010011110	0	0	0	0	0	1	0
t_7	S_1	11	001001111	0	1	1	0	1	0	0
t_8	S_2	11	100011111	0	1	1	0	0	1	0
t_9	S_3	00	010001111	0	1	0	0	0	0	1

Array Multiplier

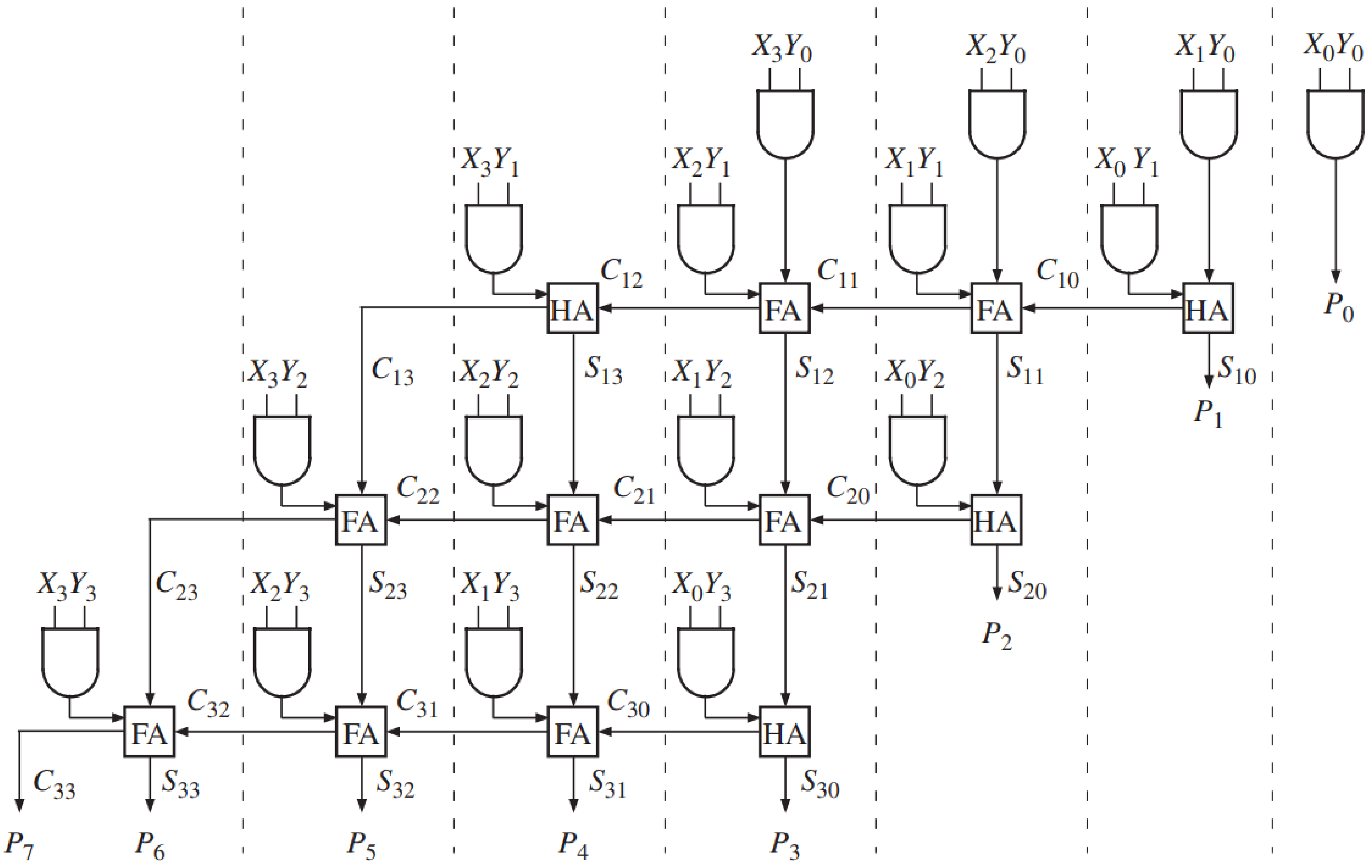
- Array multiplier: a parallel multiplier that generates the partial products in a parallel fashion. Partial products are added as soon as they are available.

Array Multiplier (continued)

- 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

Array Multiplier (continued)



Array Multiplier (continued)

- This multiplier requires 16 AND gates, 8 full adders, and 4 half-adders.
- The longest path (from input to output): 8 adders. If t_{ad} is the worst-case delay through an adder, and t_g is the longest AND gate delay, then the worst-case time to complete the multiplication is $8t_{ad} + t_g$.
- n-bit-by-n-bit array multiplier requires n^2 AND gates, n(n - 2) full adders, and n half-adders.

Array Multiplier (continued)

- For an $n \times n$ array multiplier, the longest path from input to output goes through n adders in the top row, n-1 adders in the bottom row and n-3 adders in the middle rows.
  - Worst-case multiply time: $(3n - 4)t_{ad} + t_g$.
  - The longest delay in a circuit: critical path.
  - Worse-case can be improved to $2nt_{ad} + t_g$ by forwarding carry from each adder to the diagonally lower adder.

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

- Binary fixed point:
  - 1. Split number into integer and fraction parts of pre-defined widths(fixed-point).
  - 2. Define number of significant digits along with an indicator— the exponent —of where to set the point(floating-point).

A Signed Integer/Fraction Multiplier (continued)

- Binary fixed point example:
  - Need to have at least 5 bits for the integer for the sign. So in 9-bit format:

Value	Fixed	1’s complement	2’s complement
-13.45	01101.0111	10010.1000	10010.1001

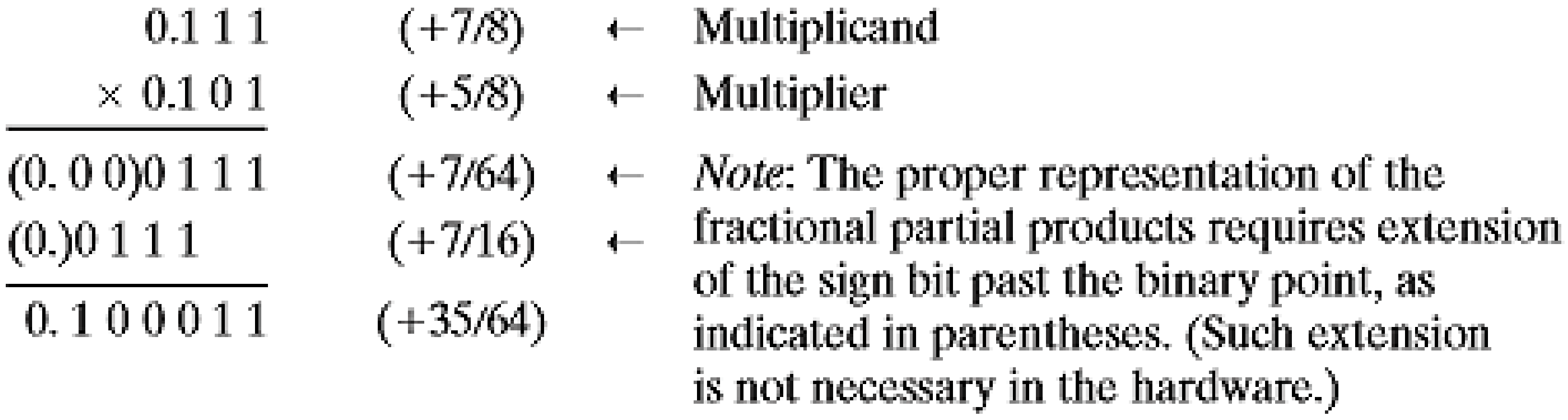
A Signed Integer/Fraction Multiplier (continued)

- Four cases to consider:

Multiplicand	Multiplier
+	+
-	+
+	-
-	-

A Signed Integer/Fraction Multiplier (continued)

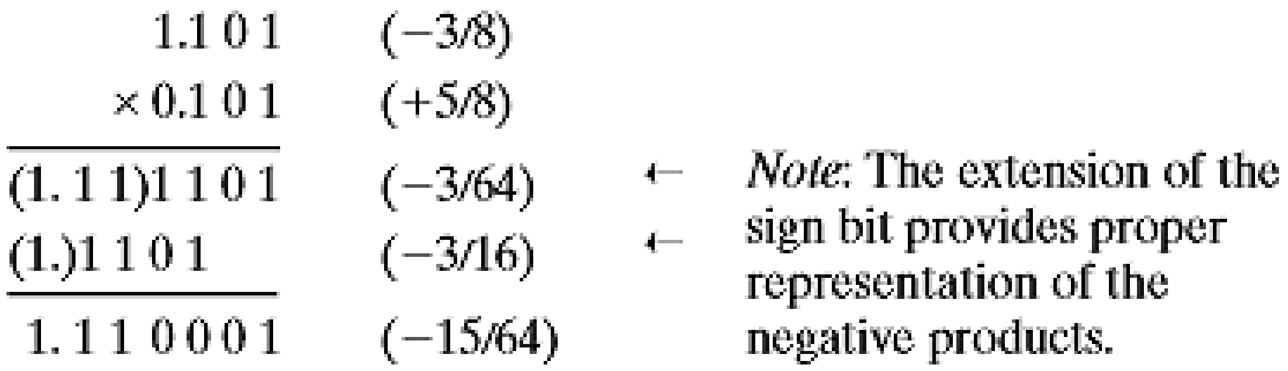
- When both the multiplicand and the multiplier are “+”, standard binary multiplication is used:



```
      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
   -----------                   of the sign bit past the binary point, as
  0. 1 0 0 0 1 1   (+35/64)      indicated in parentheses. (Such extension
                                 is not necessary in the hardware.)
```

A Signed Integer/Fraction Multiplier (continued)

- When the multiplicand is “-” and the multiplier is “+”, extend the sign bit of the multiplicand so that the partial products and final product will have the proper “-” sign:



```
         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
   (1.)1 1 0 1       (−3/16)    ←    sign bit provides proper
   ---------------                   representation of the
    1. 1 1 0 0 0 1   (−15/64)        negative products.
```

A Signed Integer/Fraction Multiplier (continued)

- When the multiplier is “-” and the multiplicand is “+”, make a slight change in the multiplication procedure. A negative fraction of the form 1.g has a numeric value -1 + 0.g:
  - So, for 1.g: treat .g as a positive fraction, but the sign bit is treated as -1.
- When we reach the negative sign bit, we must add in the 2’s complement of the multiplicand instead of the multiplicand itself.

A Signed Integer/Fraction Multiplier (continued)

- When the multiplier is “-” and the multiplicand is “+”:

```
         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
   1. 1 1 0 0 0 1       (−15/64)       this point.
```

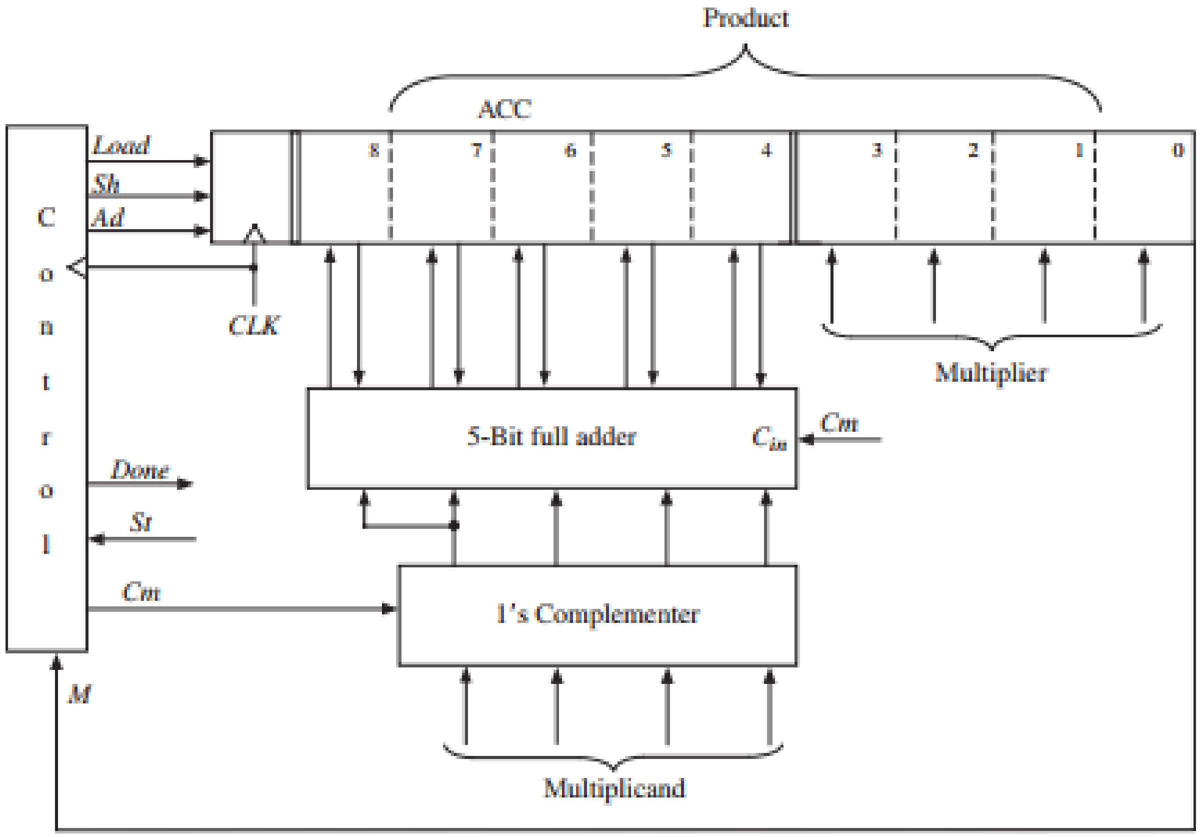
A Signed Integer/Fraction Multiplier (continued)

- When both the multiplicand and the multiplier are “-”, the procedure is the same as before. Preserve the proper negative sign, and at the final step, add in the 2’s complement of the multiplicand:

```
          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)
```

A Signed Integer/Fraction Multiplier (continued)

- Hardware required to multiply two 4-bit fractions (includes sign bit):

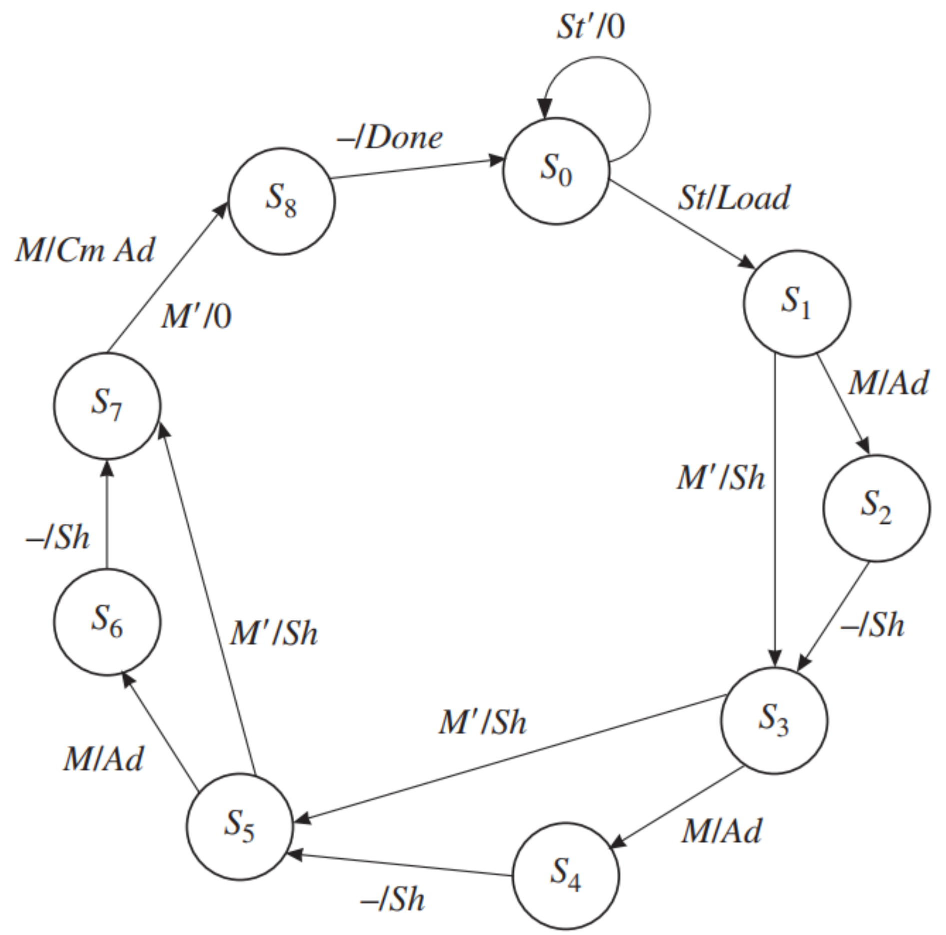


Block Diagram for 2's Complement Multiplier

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.

A Signed Integer/Fraction Multiplier (continued)

- State diagram for the control circuit:



A Signed Integer/Fraction Multiplier (continued)

- Behavioral model for 2's complement multiplier:

```
`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 ;
```

A Signed Integer/Fraction Multiplier (continued)

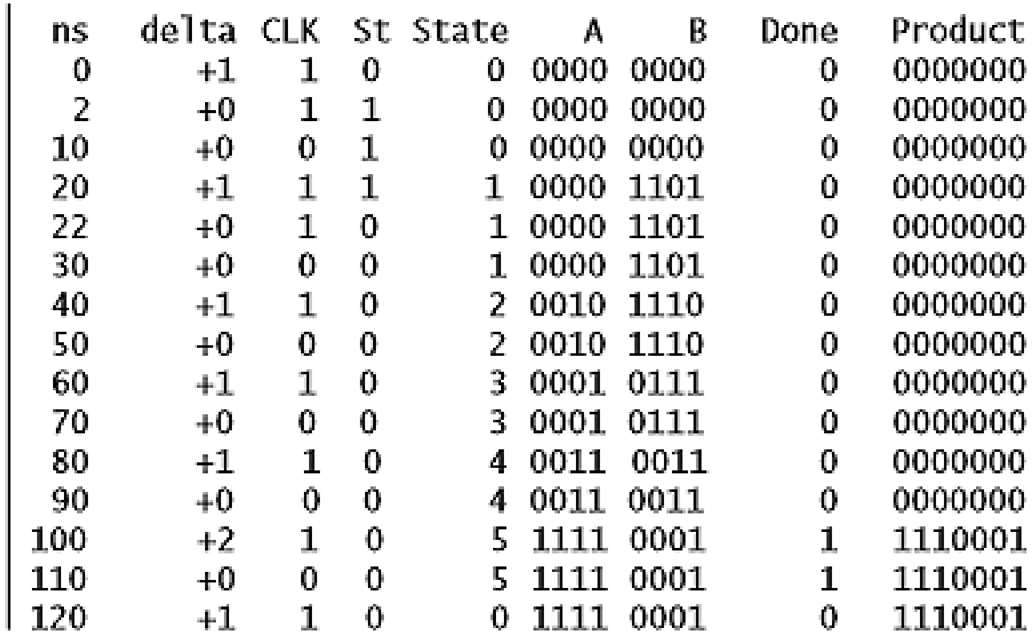
```
            4 :
                     begin
                        if (`M == 1'b1)
                        begin
                           addout = A + ~Mcand + 1;
                           A <= {~Mcand[3], addout[3:1]} ;
                           B <= {addout[0], B[3:1]} ;
                  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

A Signed Integer/Fraction Multiplier (continued)

```
            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
```

A Signed Integer/Fraction Multiplier (continued)

- Command file and simulation results for +5/8 by -3/8:



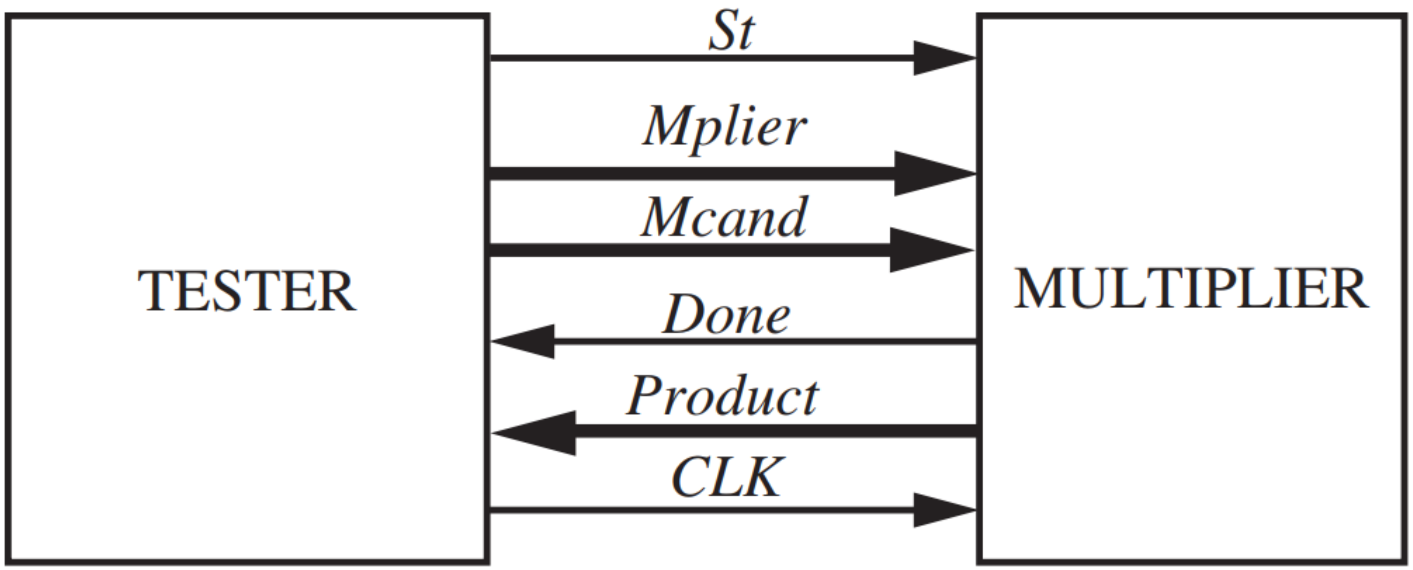
```
  ns    delta  CLK  St  State     A     B    Done    Product
   0      +1     1   0      0  0000  0000       0    0000000
   2      +0     1   1      0  0000  0000       0    0000000
  10      +0     0   1      0  0000  0000       0    0000000
  20      +1     1   1      1  0000  1101       0    0000000
  22      +0     1   0      1  0000  1101       0    0000000
  30      +0     0   0      1  0000  1101       0    0000000
  40      +1     1   0      2  0010  1110       0    0000000
  50      +0     0   0      2  0010  1110       0    0000000
  60      +1     1   0      3  0001  0111       0    0000000
  70      +0     0   0      3  0001  0111       0    0000000
  80      +1     1   0      4  0011  0011       0    0000000
  90      +0     0   0      4  0011  0011       0    0000000
 100      +2     1   0      5  1111  0001       1    1110001
 110      +0     0   0      5  1111  0001       1    1110001
 120      +1     1   0      0  1111  0001       0    1110001
```

A Signed Integer/Fraction Multiplier (continued)

- To test the multiplier, test not only the four standard cases (++,+-,-+, and --) but also special cases and limiting cases.
- Test values for the multiplicand and multiplier should include 0, the largest positive fraction, the most negative fraction, and all 1s.
- Test bench will provide a sequence of values for the multiplicand and the multiplier. It can also check for the correctness of the multiplier output.

A Signed Integer/Fraction Multiplier (continued)

- Interface between multiplier and its test bench:

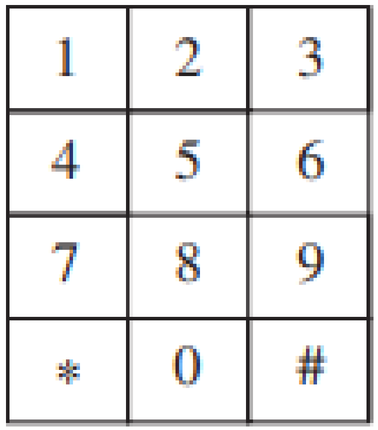


Keypad Scanner Example

- Design for scanner for keypad will include:
  - Three columns and four rows. Keypad wired in matrix form with a switch at intersections.
  - Purpose of scanner: to determine which key has been pressed and to output a binary number ($N = N_3N_2N_1N_0$) that corresponds to the key number. When a valid key has been detected, the scanner should output a signal V for one clock time.
  - Hardware to protect the circuitry from malfunction due to *keypad bounces*.

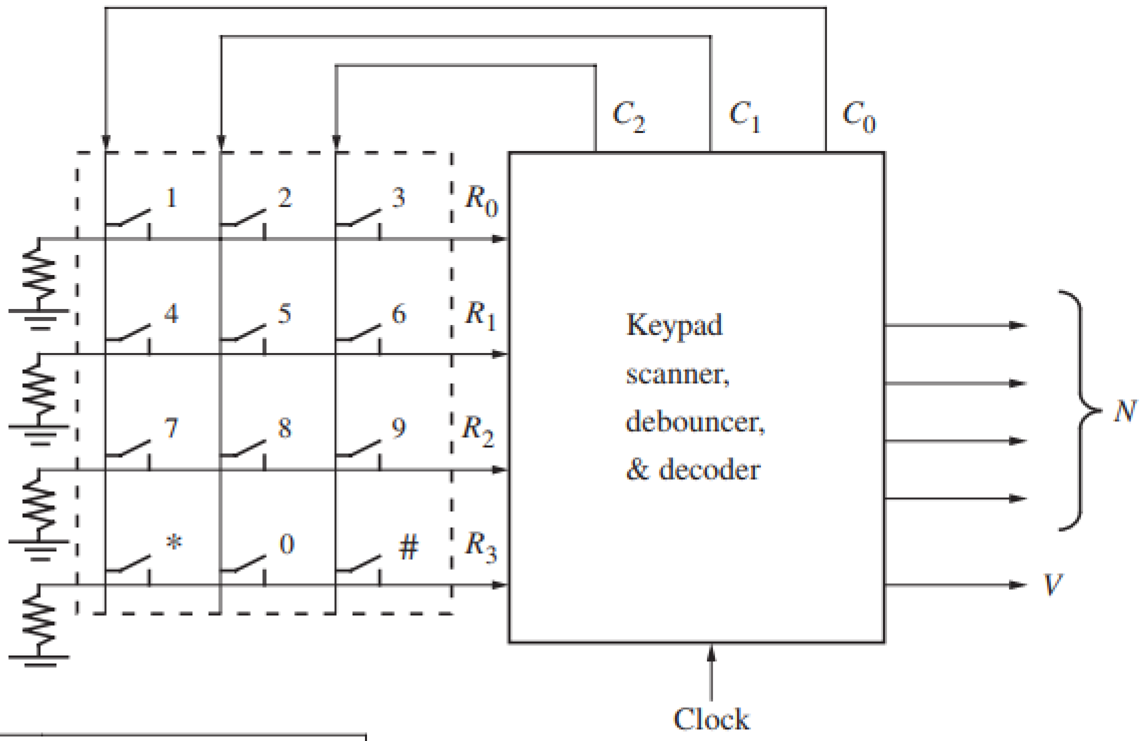
Keypad Scanner Example (continued)

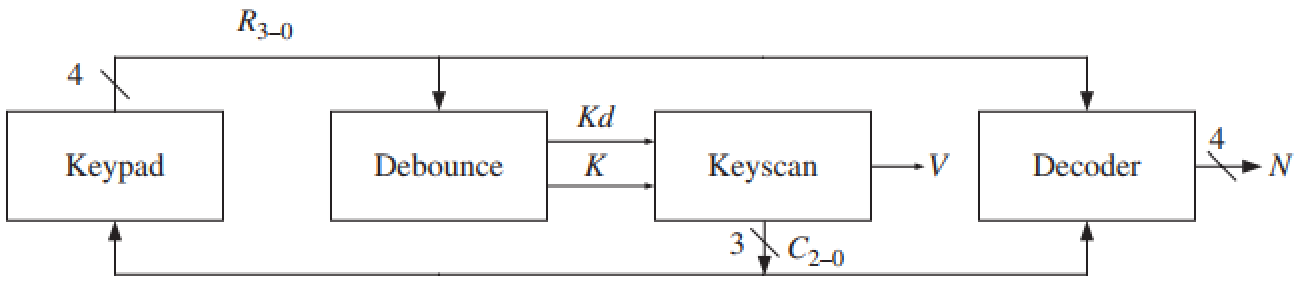
Keypad with Three Columns
And Four Rows



1	2	3
4	5	6
7	8	9
*	0	#

- Block diagram:





Scanner Modules

Keypad Scanner Example (continued)

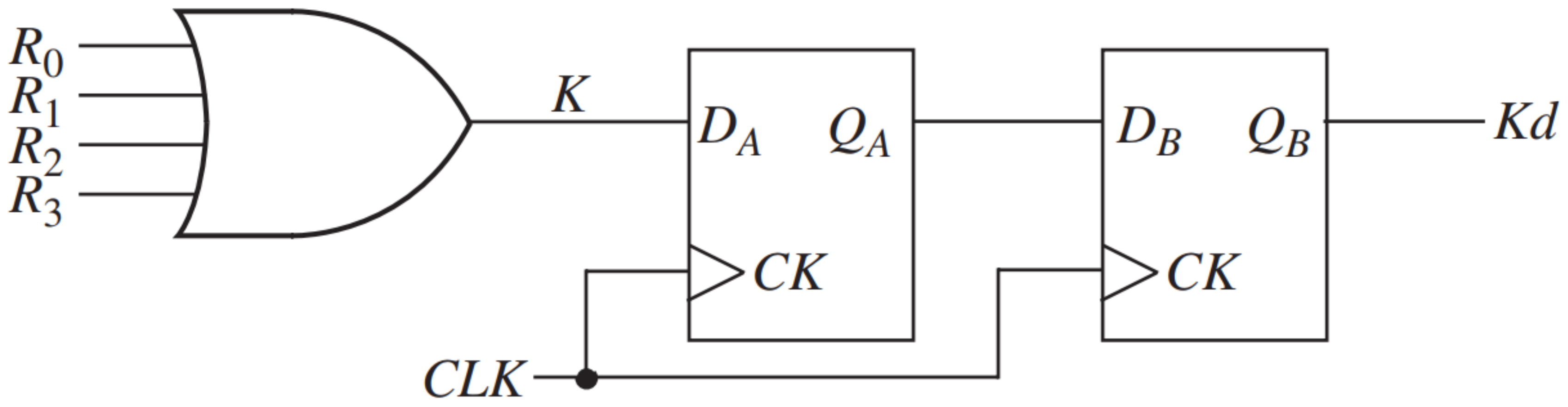
- Divide the design into modules:
  - Scanner: scans rows and columns of keypad.
  - Keyscan module: creates the column signals to scan the keyboard.
  - Debounce module: creates a signal K when a key has been pressed and a signal Kd after it's been debounced.
  - Decoder: determines the key number from the row and column numbers upon selection of a valid key.

Keypad Scanner Example (continued)

- Scanner procedure:
  - Apply logic 1s to columns C_0, C_1, and C_2 and wait. If any key is pressed, a 1 will appear on R_0, R_1, R_2, or R_3. Apply a 1 to column C_0 only. If any of the R_is is 1, a valid key is detected. If R_0 is received, one knows that switch 1 was pressed.
  - If R_1, R_2, or R_3 is received, it indicates switch 4, 7, or * was pressed. If so, set $V = 1$ and output the corresponding N.
  - If no key is detected in the first column, apply a 1 to C_1 and repeat.
  - If no key is detected in the second column, repeat for C_2.
  - When a valid key is detected, apply 1s to C_0, C_1, and C_2 and wait until no key is pressed.

Keypad Scanner Example (continued)

- Debounce and synchronize the circuit to avoid malfunctions. The four row signals are connected to an OR gate to form signal K, which turns on when a key is pressed and a column scan signal is applied. The debounced signal Kd will be fed to the sequential circuit.



Keypad Scanner Example (continued)

- Decoder: a combinational circuit; determines the key number from the row and column numbers using a truth table that has one row for each of the 12 keys. Remaining rows have don’t care outputs as it is assumed that only one key is pressed at a time.

Keypad Scanner Example (continued)

- Truth table and logic equations for decoder:

R_3	R_2	R_1	R_0	C0	C1	C2	N_3	N_2	N_1	N_0	
0	0	0	1	1	0	0	0	0	0	1	
0	0	0	1	0	1	0	0	0	1	0	
0	0	0	1	0	0	1	0	0	1	1	
0	0	1	0	1	0	0	0	1	0	0	
0	0	1	0	0	1	0	0	1	0	1	
0	0	1	0	0	0	1	0	1	1	0	
0	1	0	0	1	0	0	0	1	1	1	
0	1	0	0	0	1	0	1	0	0	0	
0	1	0	0	0	0	1	1	0	0	1	
1	0	0	0	1	0	0	1	0	1	0	(*)
1	0	0	0	0	1	0	0	0	0	0	
1	0	0	0	0	0	1	1	0	1	1	(#)

Logic Equations for Decoder

$$N_3 = R_2C_0' + R_3C_1'$$

$$N_2 = R_1 + R_2C_0$$

$$N_1 = R_0C_0' + R_2'C_2 + R_1'R_0'C_0$$

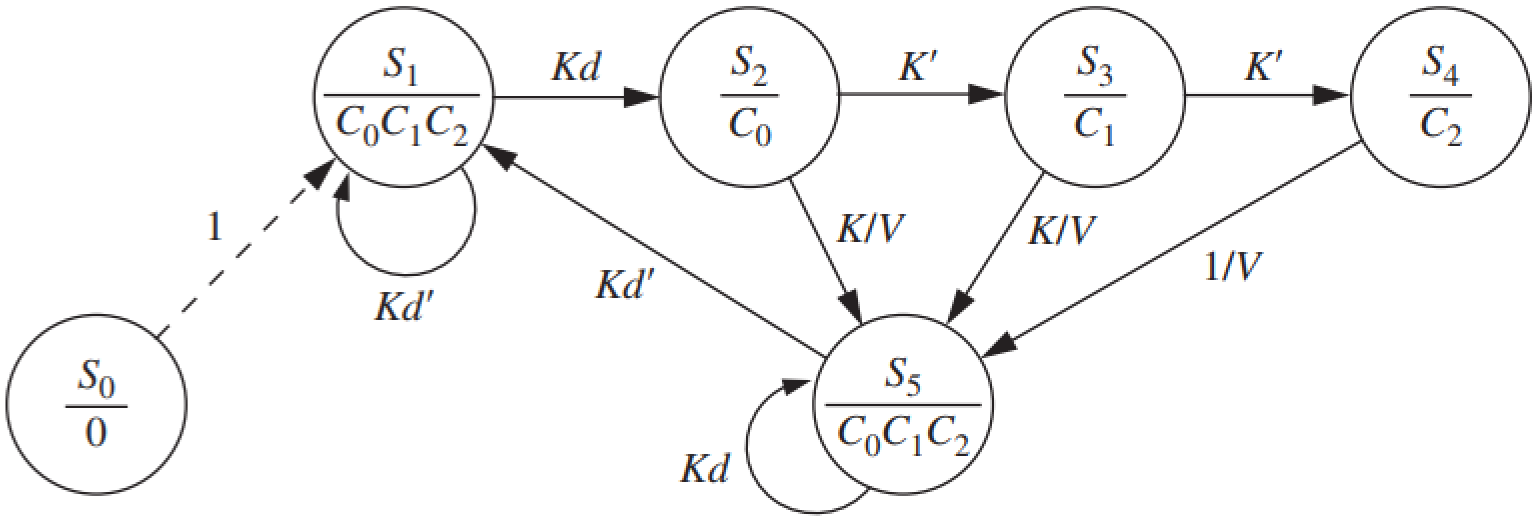
$$N_0 = R_1C_1 + R_1'C_2 + R_3'R_1'C_1'$$

At the time a valid key is detected (K = 1 and V = 1), its output will have the correct value and this value can be saved in a register at the same time the circuit goes to S5.

Keypad Scanner Example (continued)

- Controller: waits in S_1 with outputs

$C_0=C_1=C_2=1$ until a key is pressed.



State Graph for Keypad Scanner

Keypad Scanner Example (continued)

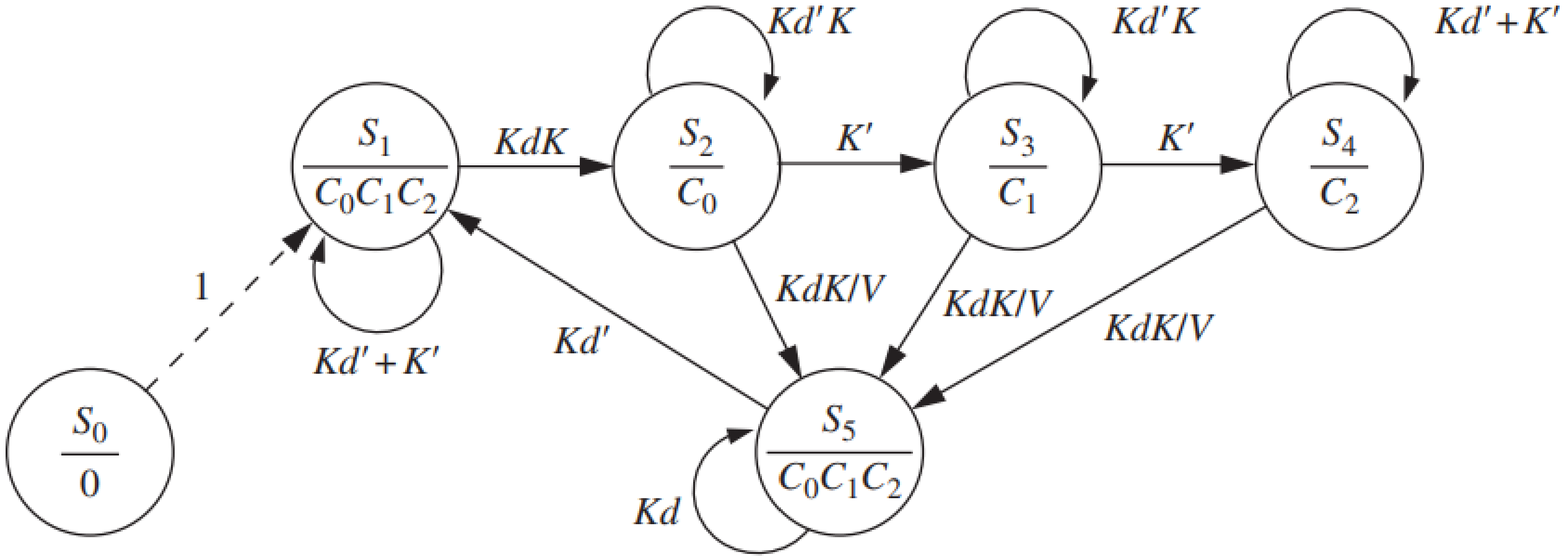
- Timing issues with preceding state diagram:
  - 1. Is K true whenever a button is pressed? No.
  - 2. Can Kd be false when a button is continuing to be pressed? Yes.
  - 3. Can you go from S_5 to S_1 when a button is still pressed? S_4-to-S_5 transition could happen when Kd is false. Kd might have become false while scanning C_0 and C_1. Hence, it is possible that one reaches back to S_1 when the key is still being pressed.

Keypad Scanner Example (continued)

- Timing issues (continued):
  - 4. What if a key is pressed for only one or two clock cycles? If the key is pressed and released very quickly, there would be problems especially if the key is in the third column. By the time the scanner reaches state S_4, the key might have been released already.

Keypad Scanner Example (continued)

- Issues can be fixed by assuring that one can reach S_5 only if Kd is true. Modified state diagram:



Keypad Scanner Example (continued)

- Verilog Code:
  - Equations for decoder, K and V are implemented by concurrent statements. The process implements the next state equations for the keyscan and debounce flip-flops.

Keypad Scanner Example (continued)

```
module scanner (R0, R1, R2, R3, CLK, C0, C1, C2, N0, N1, N2, N3, V);

    input R0;
    input R1;
    input R2;
    input R3;
    input CLK;
    inout C0;
    inout C1;
    inout C2;
    output N0;
    output N1;
    output N2;
    output N3;
    output V;

    reg V;
    reg C0_tmp, C1_tmp, C2_tmp;
```

Keypad Scanner Example (continued)

```
reg QA;
wire K;
reg Kd;
reg[2:0] state;
reg[2:0] nextstate;

assign C0 = C0_tmp;
assign C1 = C1_tmp;
assign C2 = C2_tmp;

assign K = R0 | R1 | R2 | R3 ;
assign N3 = (R2 & ~C0) | (R3 & ~C1) ;
assign N2 = R1 | (R2 & C0) ;
assign N1 = (R0 & ~C0) | (~R2 & C2) | (~R1 & ~R0 & C0) ;
assign N0 = (R1 & C1) | (~R1 & C2) | (~R3 & ~R1 & ~C1) ;

initial
begin
    state = 0;
    nextstate = 0;
end

always @(state or R0 or R1 or R2 or R3 or C0 or C1 or C2 or K or Kd or QA)
begin
  C0_tmp = 1'b0 ;
  C1_tmp = 1'b0 ;
  C2_tmp = 1'b0 ;
  V = 1'b0 ;
  case (state)
      0 :
```

Keypad Scanner Example (continued)

```
case (state)
    0 :
              begin
                 nextstate = 1 ;
              end
    1 :
              begin
                CO_tmp = 1'b1 ;
                C1_tmp = 1'b1 ;
                C2_tmp = 1'b1 ;
                 if ((Kd & K) == 1'b1)
                 begin
                    nextstate = 2 ;
                 end
                 else
                 begin
                    nextstate = 1 ;
                 end
              end
   2 :
              begin
                CO_tmp = 1'b1 ;
                 if ((Kd & K) == 1'b1)
                begin
                    V = 1'b1 ;
```

Keypad Scanner Example (continued)

```
                  nextstate = 5 ;
               end
               else if (K == 1'b0)
               begin
                  nextstate = 3 ;
               end
               else
               begin
                 nextstate = 2 ;
               end
            end
3 :
            begin
               C1_tmp = 1'b1 ;
               if ((Kd & K) == 1'b1)
               begin
                  V = 1'b1 ;
                  nextstate = 5 ;
               end
               else if (K == 1'b0)
               begin
                  nextstate = 4 ;
               end
               else
               begin
                  nextstate = 3 ;
               end
            end
```

Keypad Scanner Example (continued)

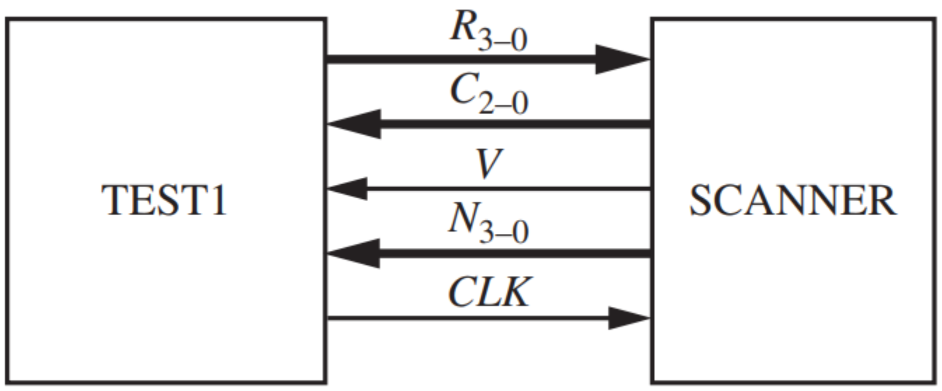
```
4 :
          begin
             C2_tmp <= 1'b1 ;
             if ((Kd & K) == 1'b1)
             begin
                V <= 1'b1 ;
                nextstate = 5 ;
             end
             else
             begin
                nextstate = 4 ;
             end
          end
5 :
          begin
             C0_tmp = 1'b1 ;
             C1_tmp = 1'b1 ;
             C2_tmp = 1'b1 ;
             if (Kd == 1'b0)
             begin
                nextstate = 1 ;
             end
             else
```

Keypad Scanner Example (continued)

```
                              begin
                                 nextstate = 5 ;
                              end
                           end
      endcase
    end
    always @(posedge CLK)
    begin
          state <= nextstate ;
          QA <= K ;
          Kd <= QA ;
    end
  endmodule
```

Keypad Scanner Example (continued)

- Testbench:
  - Simulates a key press by supplying the appropriate R signals in response to the C signals from the scanner. When test bench receives V = 1 from the scanner, it checks to see whether the value of N corresponds to the key that was pressed.



Keypad Scanner Example (continued)

- Test process:
  - 1. Read a key number from the array to simulate pressing a key.
  - 2. Wait until V=1 and the rising edge of the clock occurs.
  - 3. Verify that the N output from the scanner matches the key number.
  - 4. Set KN=15 to simulate no key pressed. (Since 15 is not a valid key number, all Rs will go to 0.)
  - 5. Wait until Kd=0 before selecting a new key.

Binary Dividers

Consider the design of a parallel divider for positive binary numbers.

- Binary division process:
  - Shift dividend left.
  - Enter the quotient bit by bit into the right end of the dividend register as dividend is shifted left.
  - If divisor is going to be negative, shift dividend one place to the left before subtraction.
  - Subtract to get new dividend.
  - Shift dividend left. If result will be negative, repeat shift then subtract.
  - Perform final shift.
  - If the quotient contains more bits that are available for storing the quotient, an *overflow* has occurred.

Binary Dividers (continued)

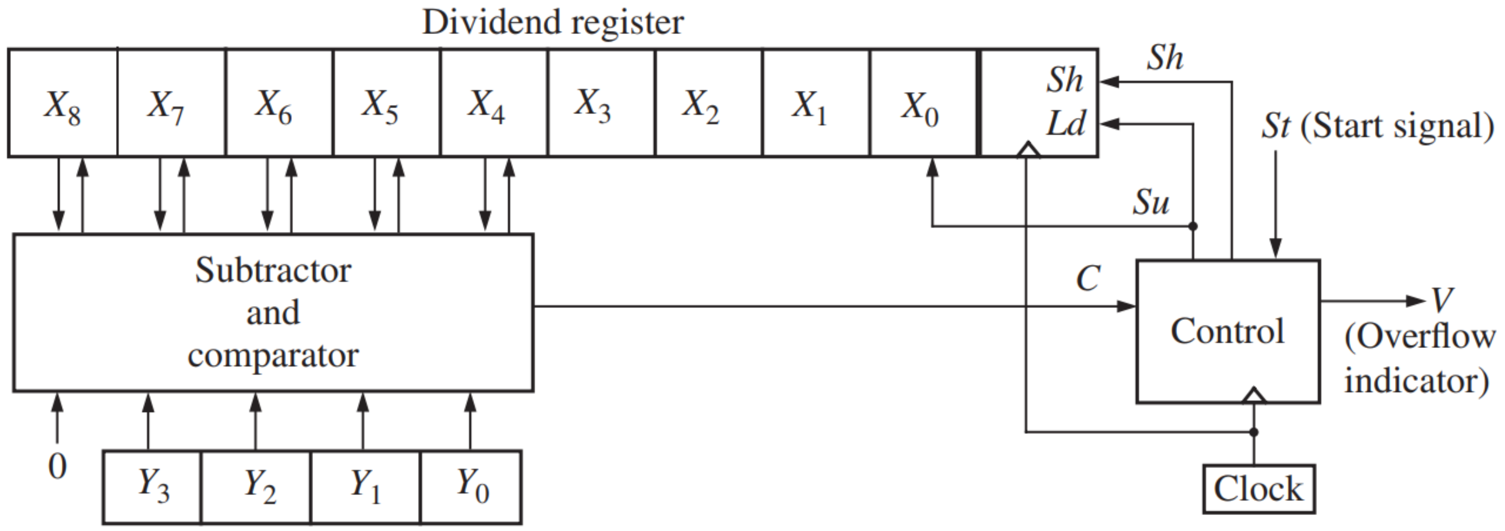
- Unsigned divider:
  - Example: divide an 8-bit dividend by a 4-bit divisor to obtain a 4-bit quotient: $135 \div 13 = 10$

```
                              1010       quotient
Divisor   1101 / 10000111               dividend
                 1101
(135 ÷ 13 = 10 with              0111
 a remainder of 5)             0000
                    1111
                    1101
                     0101
                     0000
                     0101       Remainder
```

Binary Dividers (continued)

Just as binary multiplication can be carried out as a series of add and shift operations, division can be carried out by a series of subtract and shift operations.

- 9-bit dividend register and a 4-bit divisor register:



- Initially, dividend and divisor are entered as:

$135 \div 13 = 10$

0	1	0	0	0	0	1	1	1

1	1	0	1

Binary Dividers (continued)

– Shift dividend left: $135 \div 13 = 10$

```
1  0  0  0  0  1  1  1 ┆   ← Dividing line between dividend and quotient
   1  1  0  1          ┆ _ ← Note that after the shift, the right most position
                             in the dividend register is "empty"
```

– Subtract. First quotient digit of 1 is stored in the unused position of the dividend register:

```
0  0  0  1  1  1  1  1 ┆ 1 ← first quotient digit
```

– Shift dividend one place left:

```
0  0  1  1  1  1  1 ┆ 1  _
   1  1  0  1       ┆
```

Binary Dividers (continued)

- Subtraction would yield a negative result, so shift dividend left again (2nd quotient remains empty):

```
0  1  1  1  1  1 ¦ 1  0  _
   1  1  0  1    ¦
```

- Subtract. Third quotient digit of 1 is stored in the unused position of the dividend register:

$$0\ \ 0\ \ 0\ \ 1\ \ 0\ \ 1 \ \vdots\ 1\ \ 0\ \ 1 \longleftarrow \text{third quotient digit}$$

- Final shift. Fourth quotient bit is set to 0.

$$\underbrace{0\ \ 0\ \ 1\ \ 0\ \ 1}_{\text{remainder}} \ \vdots\ \underbrace{1\ \ 0\ \ 1\ \ 0}_{\text{quotient}}$$

Binary Dividers (continued)

A shift signal (S_h) will shift the dividend one place to the left. A subtract signal (S_u) will subtract the divisor from the 5 leftmost bits in the dividend register and set the quotient bit (the rightmost bit in the dividend register) to 1.
If the divisor is greater than the 5 leftmost dividend bits, the comparator output is $C = 0$; otherwise, $C = 1$. Whenever $C = 0$, subtraction cannot occur without a negative result, so a shift signal is generated. Whenever $C = 1$, a subtract signal is generated, and the quotient bit is set to 1. The control circuit generates the required sequence of shift and subtract signals.

Binary Dividers (continued)

- If, as a result of a division operation, the quotient contains more bits than are available for storing the quotient, we say that an *overflow* has occurred. The quotient would be too large to store in the 4 bits we have allocated for it, and we have detected an overflow condition.

- If initially $X_8X_7X_6X_5X_4 \geq Y_3Y_2Y_1Y_0$ (i.e., if the left 5 bits of the dividend register exceed or equal the divisor), the quotient will be greater than 15 and an *overflow* occurs.

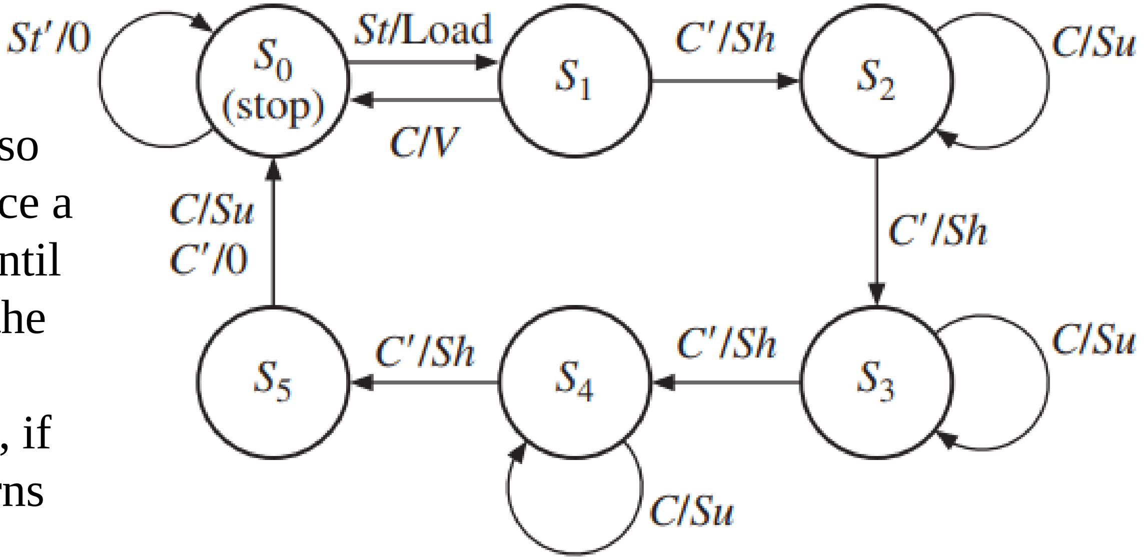
Binary Dividers (continued)

- Test for overflow condition:
  - Shift the dividend left one place and then comparing the upper half of the dividend (divu) with the divisor.
  - If $divu \geq divisor$, the quotient would be greater than the maximum value, which is an overflow condition.

Binary Dividers (continued)

- State diagram and table for unsigned divider :

- When a start signal (*St*) occurs, the 8-bit dividend and 4-bit divisor are loaded into the appropriate registers.
- If C is 1, the quotient would require 5 or more bits. Since space is provided only for a 4-bit quotient, this condition constitutes an overflow, so the divider is stopped and the overflow indicator is set by the V output.
- Normally, the initial value of C is 0, so a shift will occur first and the control circuit will go to state S2.
- If C = 1, subtraction occurs.
- After the subtraction is completed, C will always be 0,so the next clock pulse will produce a shift. This process continues until four shifts have occurred and the control is in state S5.
- Then a final subtraction occurs, if necessary, and the control returns to the stop state.



Design of a Binary Divider

One flip-flop is used for each state with

$$Q_0 = 1 \text{ in } S_0,\ Q_1 = 1 \text{ in } S_1,\ Q_2 = 1 \text{ in } S_2,\ \text{etc}$$

by inspection, The next-state and output equation

$$Q_0^{+} = St'Q_0 + CQ_1 + Q_5 \qquad Q_1^{+} = StQ_0$$

$$Q_2^{+} = C'Q_1 + CQ_2 \qquad Q_3^{+} = C'Q_2 + CQ_3$$

$$Q_4^{+} = C'Q_3 + CQ_4 \qquad Q_5^{+} = C'Q_4$$

$$Load = StQ_0 \qquad \mathrm{V} = \mathrm{C}\mathrm{Q}_1$$

$$Sh = C'(Q_1 + Q_2 + Q_3 + Q_4) = C'(Q_0 + Q_5)'$$

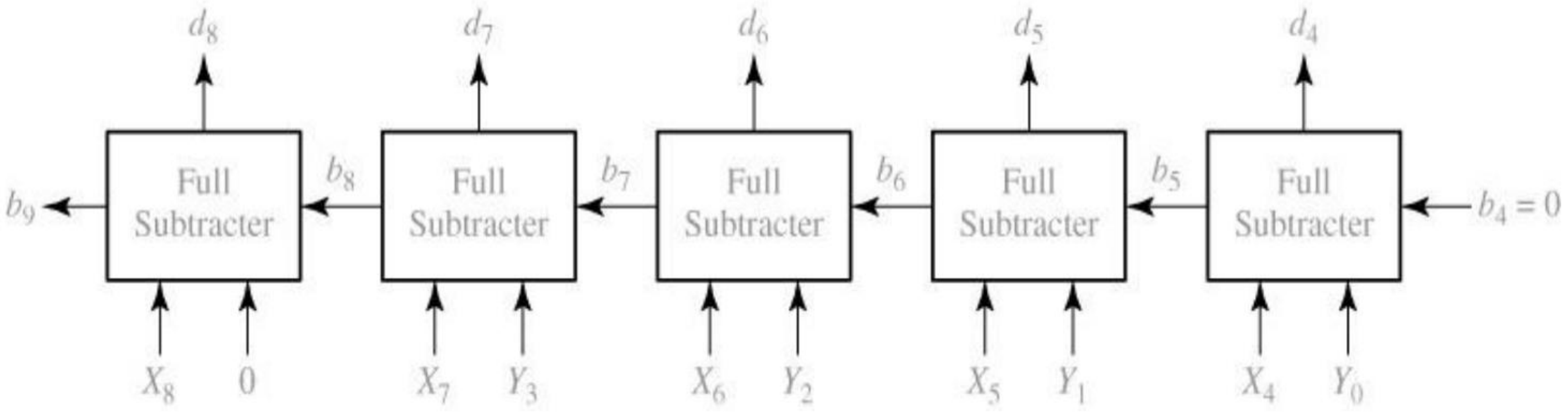
$$Su = C(Q_2 + Q_3 + Q_4 + Q_5) = C(Q_0 + Q_1)'$$

Binary Dividers (continued)

	StC				StC			
State	00	01	11	10	00	01	11	10
S_0	S_0	S_0	S_1	S_1	0	0	Load	Load
S_1	S_2	S_0	–	–	*Sh*	*V*	–	–
S_2	S_3	S_2	–	–	*Sh*	*Su*	–	–
S_3	S_4	S_3	–	–	*Sh*	*Su*	–	–
S_4	S_5	S_4	–	–	*Sh*	*Su*	–	–
S_5	S_0	S_0	–	–	0	*Su*	–	–

Since we assumed that St = 0 in states S1, S2, S3, and S4, the next states and outputs are “don’t cares” for these states when St = 1. The entries in the output table indicate which outputs are 1.

Design of a Binary Divider



Summary

- Traditional design methodology splits a design into a “data path” (the ALU) and a “controller.”
- Non-arithmetic circuit examples:
  - BCD (binary code display) to 7-segment display
  - Keypad scanner
- Arithmetic circuit examples:
  - Shift-and-Add Multiplier (and serial-parallel multiplier)
  - BCD adder
  - Array multiplier

Summary (continued)

- Issues in systems with external inputs:
  - Synchronization
  - Debouncing
  - Single pulser
- Many design examples included a block diagram, state graph, Verilog code, and test bench.

Summary (continued)

- Algorithms described:
  - Addition (CLA, RCA).
  - Multiplication (Signed and unsigned)
  - Division of unsigned binary numbers.
  - Discussion on overflow.

Thank You