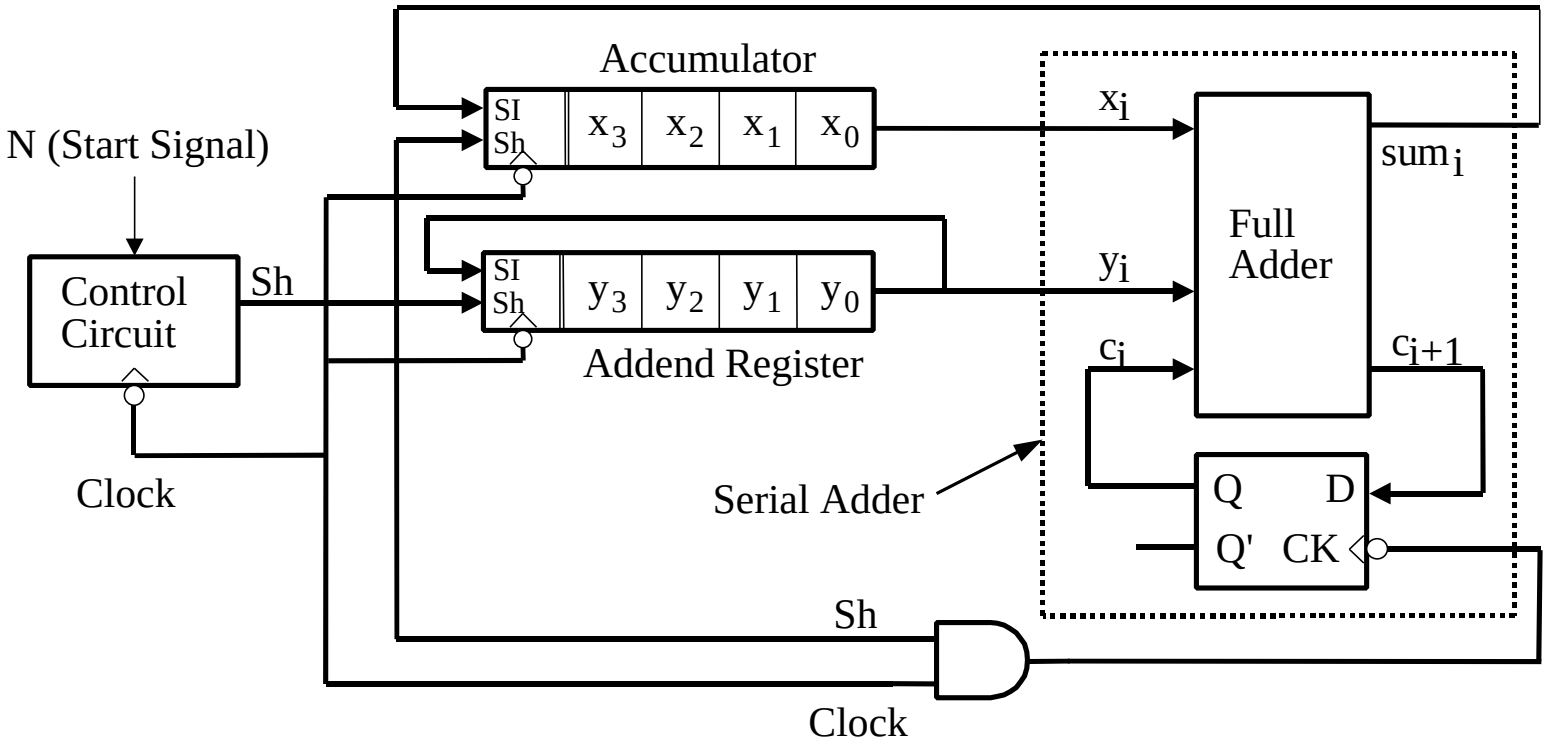
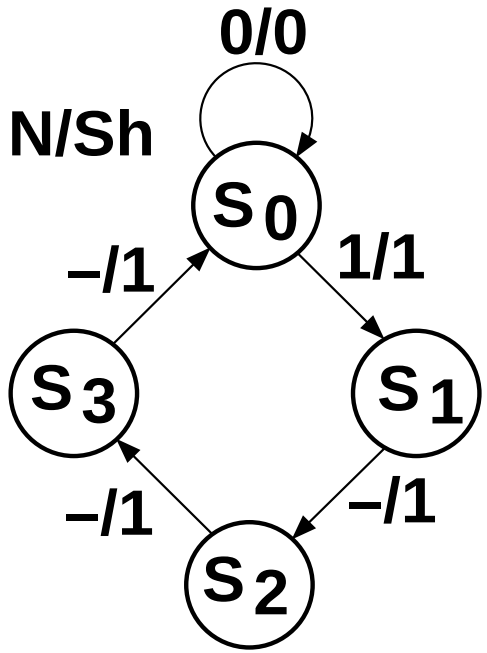
## Figure 4-1 Serial Adder with Accumulator




| | X | Y | $c_i$ | $sum_i$ | $c_{i+1}$ |
|---|---|---|---|---|---|
| $t_0$ | 0101 | 0111 | 0 | 0 | 1 |
| $t_1$ | 0010 | 1011 | 1 | 0 | 1 |
| $t_2$ | 0001 | 1101 | 1 | 1 | 1 |
| $t_3$ | 1000 | 1110 | 1 | 1 | 0 |
| $t_4$ | 1100 | 0111 | 0 | (1) | (0) |

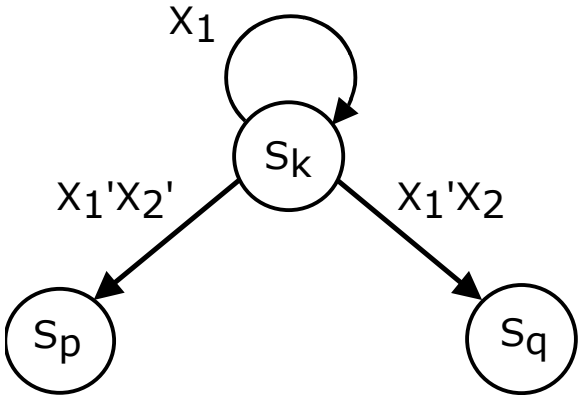
## Figure 4-2 Control State Graph and Table for Serial Adder



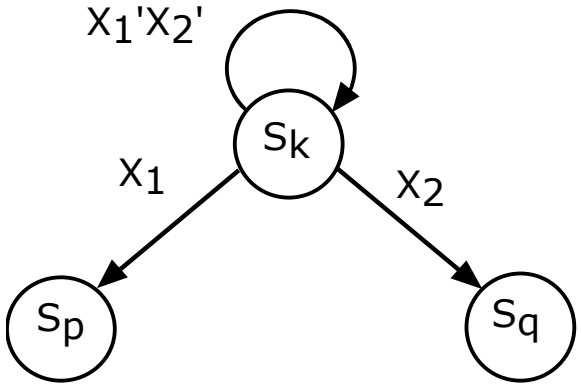

| Present State | Next State N=0 | Next State N=1 | Present Output (Sh) N=0 | Present Output (Sh) N=1 |
|---|---|---|---|---|
| $S_0$ | $S_0$ | $S_1$ | 0 | 1 |
| $S_1$ | $S_2$ | $S_2$ | 1 | 1 |
| $S_2$ | $S_3$ | $S_3$ | 1 | 1 |
| $S_3$ | $S_0$ | $S_0$ | 1 | 1 |

## Constraints on Input Labels for Every State $S_k$ *(From Page 123-124)*

1. If $I_i$ and $I_j$ are any pair of input labels on arcs exiting state $S_k$, then $I_iI_j = 0$ if $i \neq j$.
2. If n arcs exit state $S_k$ and the n arcs have input labels $I_1, I_2, ..., I_n$, respectively, then $I_1 + I_2 + ... + I_n = 1$.



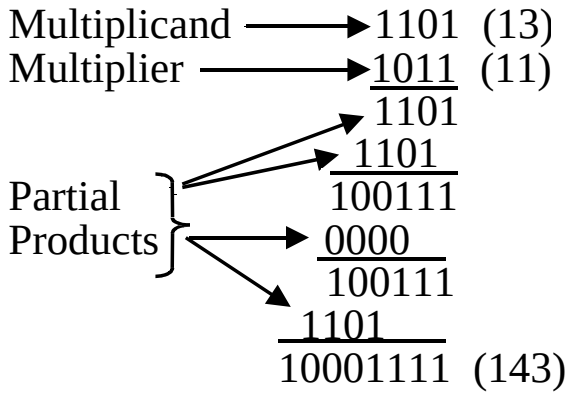

$(X_1)(X_1'X_2') = 0$
$(X_1)(X_1'X_2) = 0$
$(X_1'X_2')(X_1'X_2) = 0$
$X_1 + X_1'X_2' + X_1'X_2 = 1$




Inputs are $X_1$ $X_2$ $X_3$
($X_1 = X_2 = 1$ not allowed)

| | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| $S_k$ | $S_k$ | $S_k$ | $S_q$ | $S_q$ | $S_p$ | $S_p$ | – | – |

## Multiplication of $13_{10}$ by $11_{10}$ In Binary – *From Page 124*



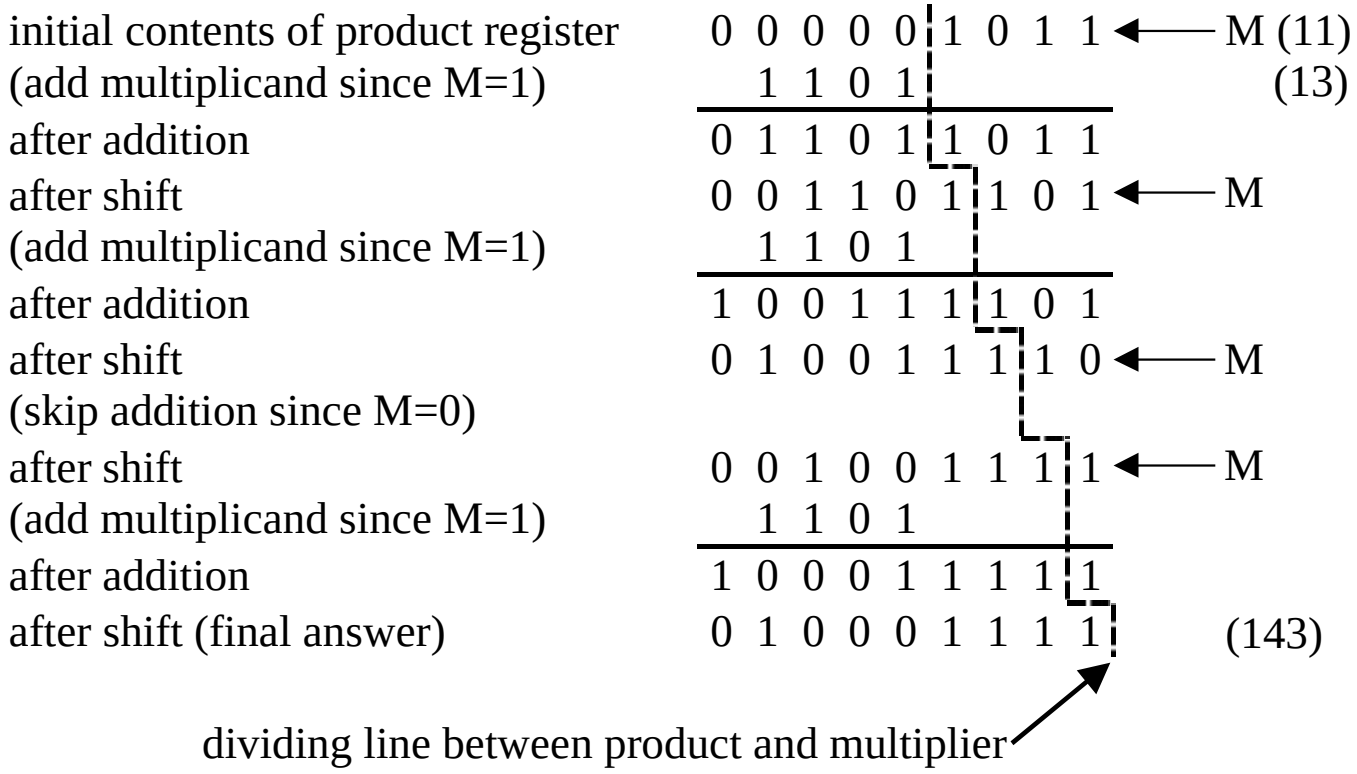
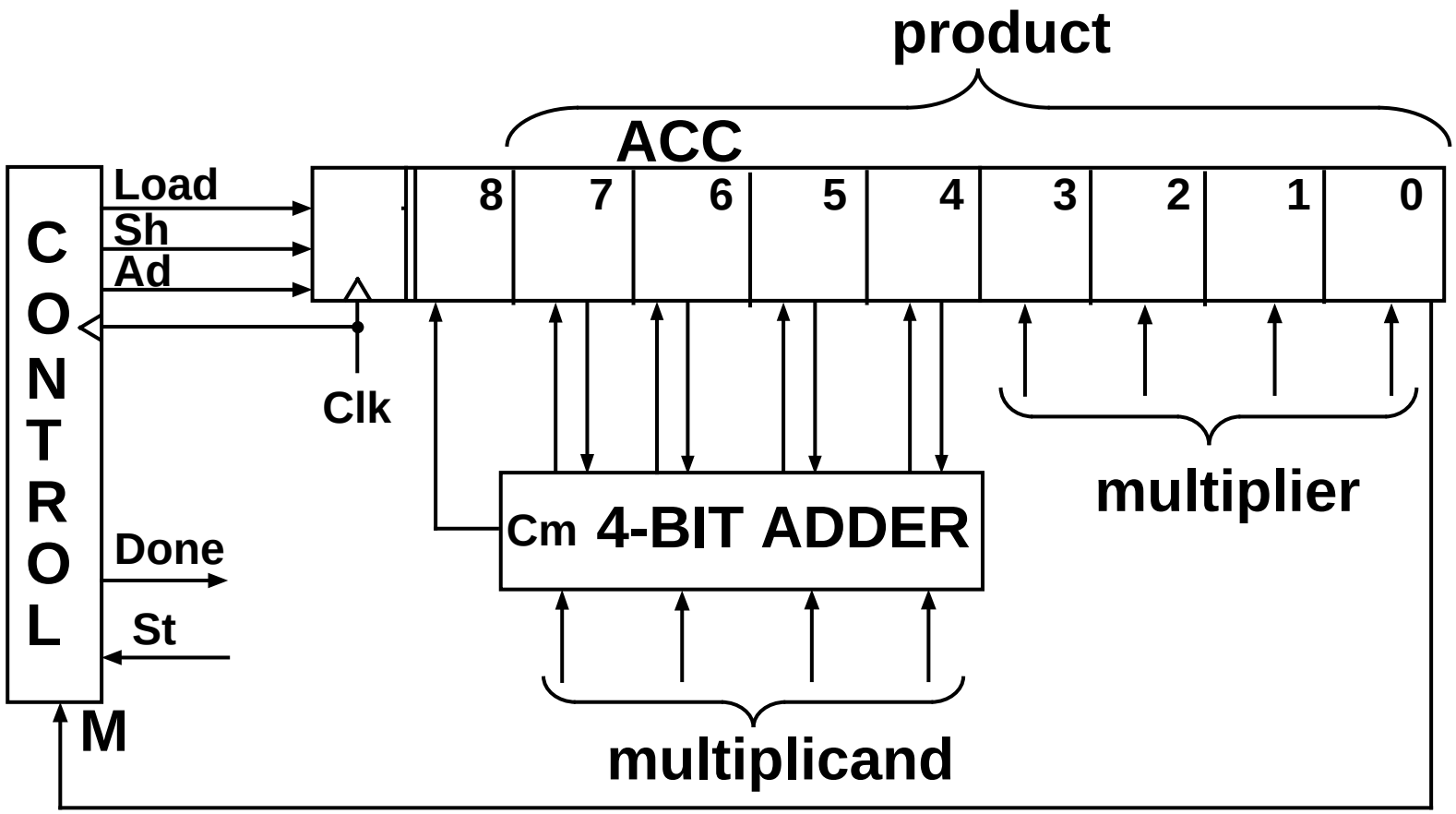
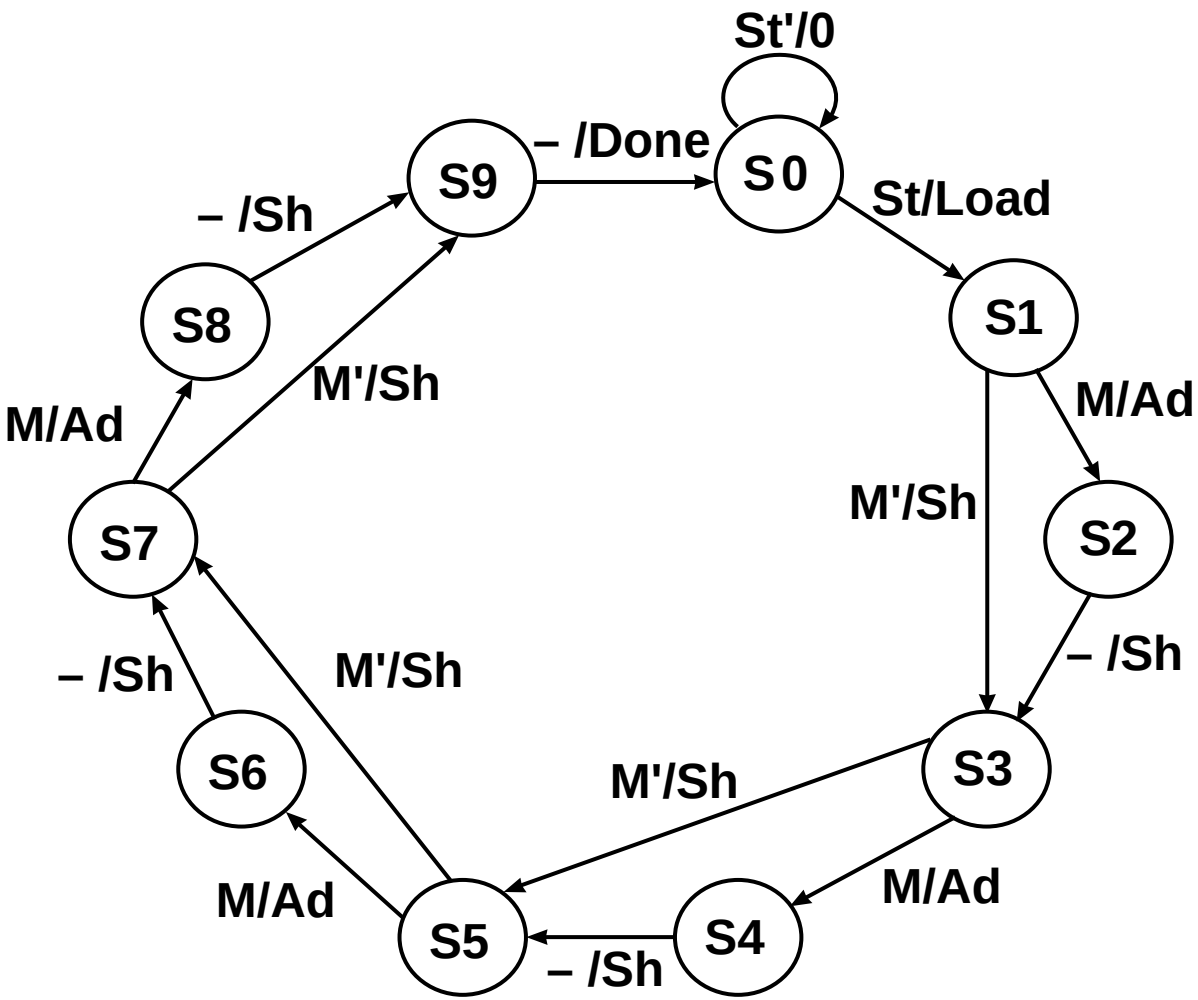



**Figure 4-3 Block Diagram for Binary Multiplier**



# Figure 4-4 State graph for Binary Multiplier Control



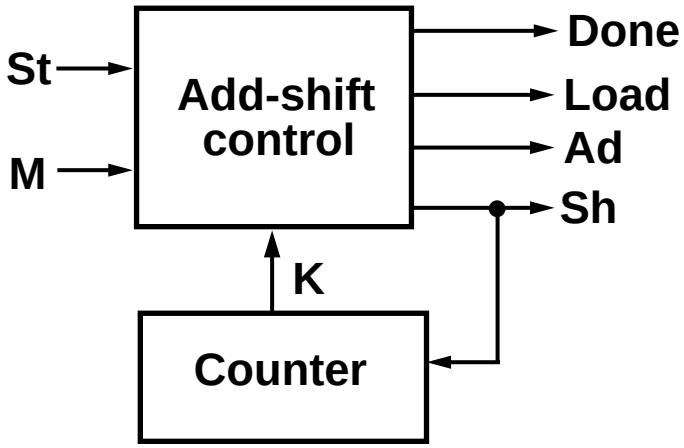
## Figure 4-5(a) Behavioral Model for 4 x 4 Binary Multiplier

```
-- 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.

library BITLIB;
use BITLIB.bit_pack.all;
entity mult4X4 is
      port (Clk, St: in bit;
            Mplier,Mcand : in bit_vector(3 downto 0);
            Done: out bit);
end mult4X4;
architecture behave1 of mult4X4 is
      signal State: integer range 0 to 9;
      signal ACC: bit_vector(8 downto 0);          -- accumulator
      alias M: bit is ACC(0);                      -- M is bit 0 of ACC
begin
      process
      begin
            wait until Clk = '1';                  -- executes on rising edge of clock
            case State is
                  when 0=>                                    -- initial State
                        if St='1' then
                              ACC(8 downto 4) <= "00000";     -- Begin cycle
                              ACC(3 downto 0) <= Mplier;      -- load the multiplier
                              State <= 1;
                        end if;
```

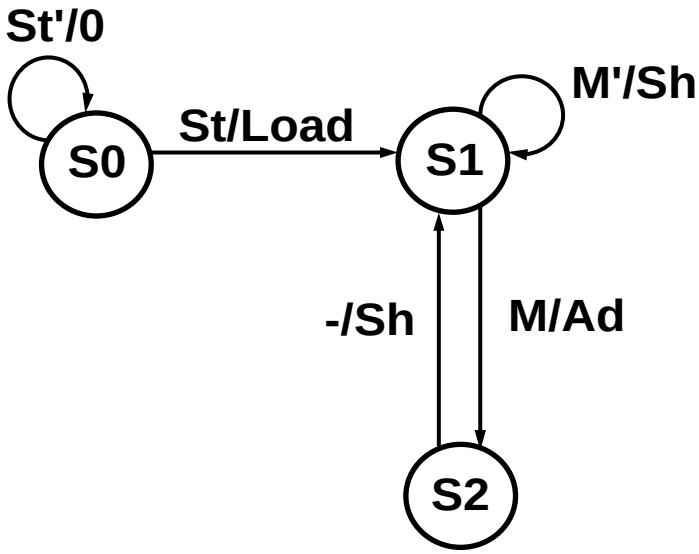
## Figure 4-5(b) Behavioral Model for 4 x 4 Binary Multiplier

```
            when 1 | 3 | 5 | 7  =>                          -- "add/shift" State
                if M = '1' then                             -- Add multiplicand
                    ACC(8 downto 4) <=add4(ACC(7 downto 4),Mcand,'0');
                    State <= State + 1;
                else
                    ACC <= '0' & ACC(8 downto 1);           --Shift accumulator right
                    State <= State + 2;
                end if;
            when 2 | 4 | 6 | 8 =>                           -- "shift" State
                ACC <= '0' & ACC(8 downto 1);               -- Right shift
                State <= State + 1;
            when 9 =>                                       -- End of cycle
                State <= 0;
        end case;
    end process;
    Done <= '1' when State = 9 else '0';
end behave1;
```

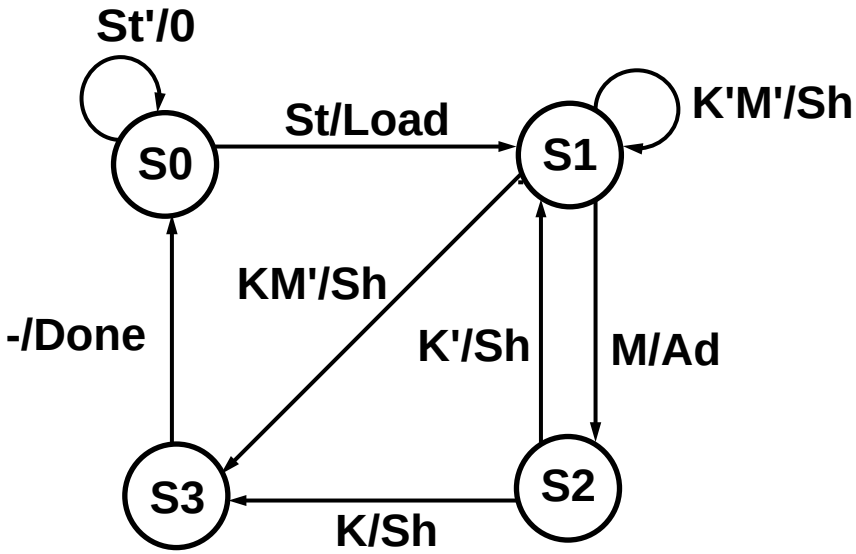
# Figure 4-6 Multiplier Control with Counter




(a) Multipler control




(b) State graph for add-shift control




(c) Final state graph for add-shift control

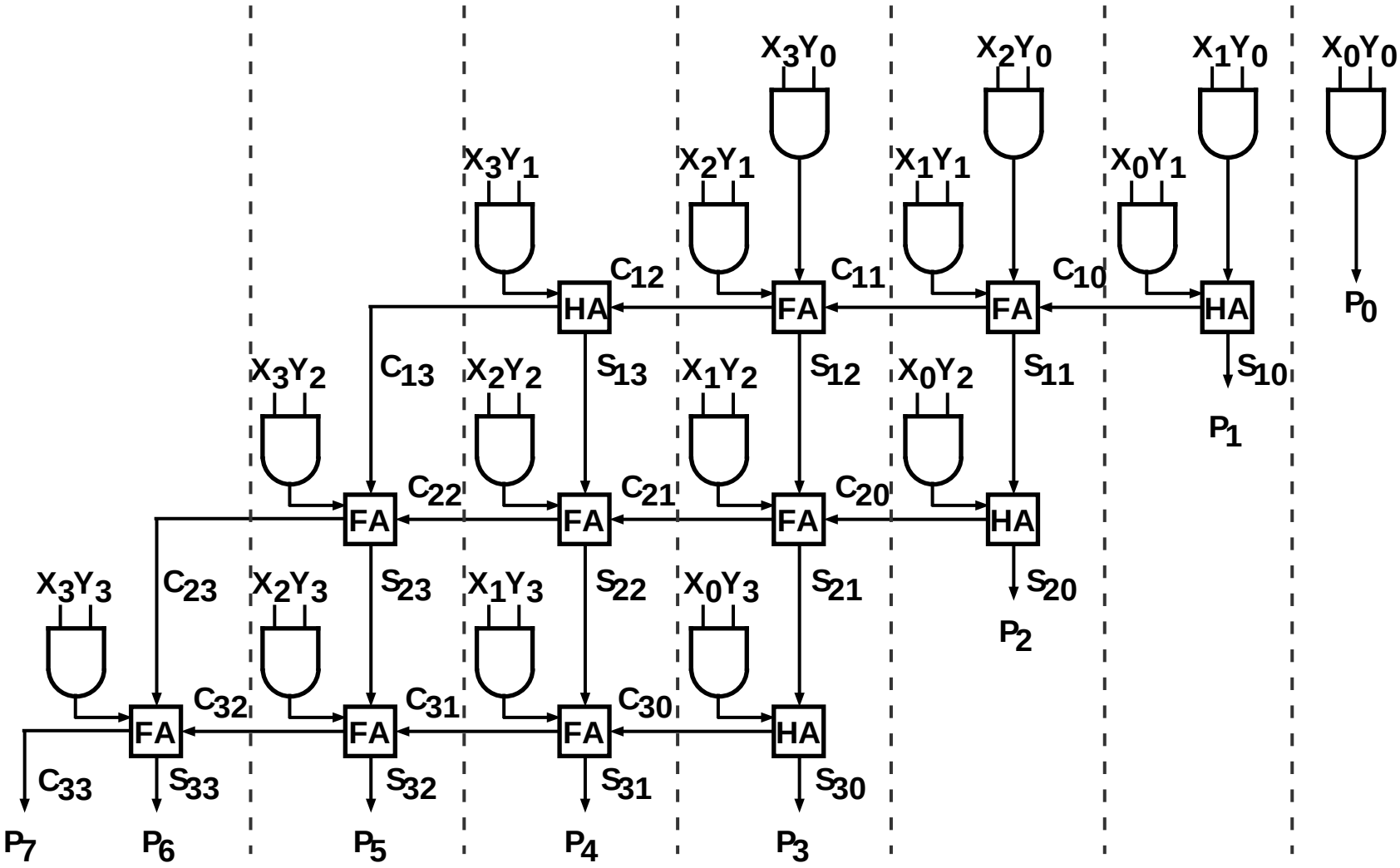
**Table 4-2 Operation of Multiplier Using a Counter**

| Time | State | Counter | Product Register | St | M | K | Load | Ad | Sh | Done |
|---|---|---|---|---|---|---|---|---|---|---|
| t0 | S0 | 00 | 000000000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| t1 | S0 | 00 | 000000000 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| t2 | S1 | 00 | 000001011 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| t3 | S2 | 00 | 011011011 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| t4 | S1 | 01 | 001101101 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| t5 | S2 | 01 | 100111101 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| t6 | S1 | 10 | 010011110 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| t7 | S1 | 11 | 001001111 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| t8 | S2 | 11 | 100011111 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| t9 | S3 | 00 | 010001111 | 0 | 1 | 0 | 0 | 0 | 0 | 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 4-7 Block Diagram of 4 x 4 Array Multiplier



***From Page 133***

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

```
          1. 1  0  1      (–3/8)
        X 0. 1  0  1      (+5/8)
-------------------------------------
(1. 1  1) 1  1  0  1      (–3/64)   <-- Note: The extension of the sign bit provides
(1.)1  1  0  1            (–3/16)   <-- proper representation of the negative products.
-------------------------------------
 1. 1  1  0  0  0  1      (–15/64)
```

***From Pages 133 – 134***

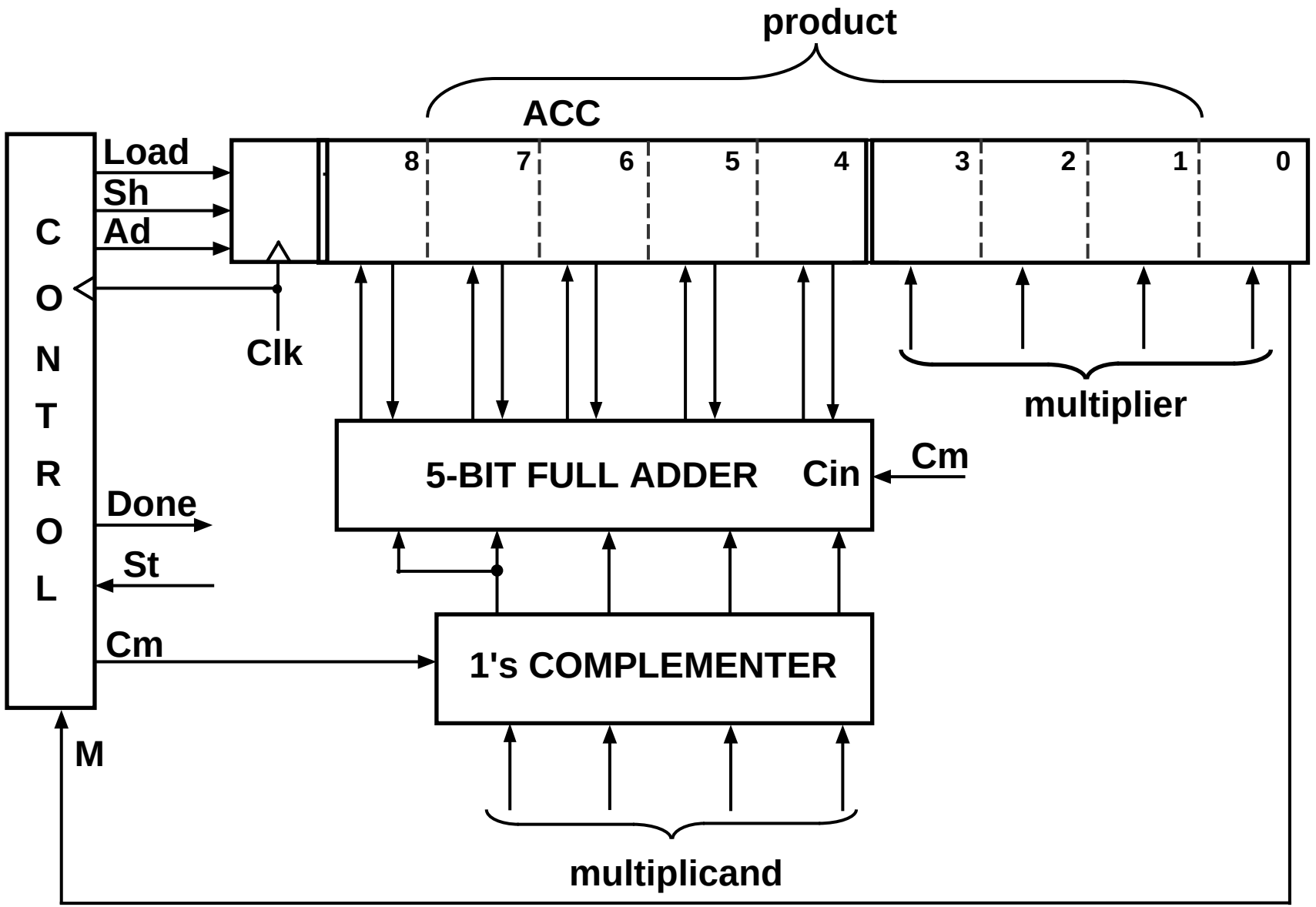
```
          0. 1  0  1      (+5/8)
        X 1. 1  0  1      (–3/8)
-------------------------------------
(0. 0  0) 0  1  0  1      (+5/64)
(0.)0  1  0  1            (+5/16)
-------------------------------------
(0.)0  1  1  0  0  1
 1. 0  1  1               (–5/8)
-------------------------------------
 1. 1  1  0  0  0  1      (–15/64)
```

← *Note*: The two's complement of the multiplicand is added at this point.

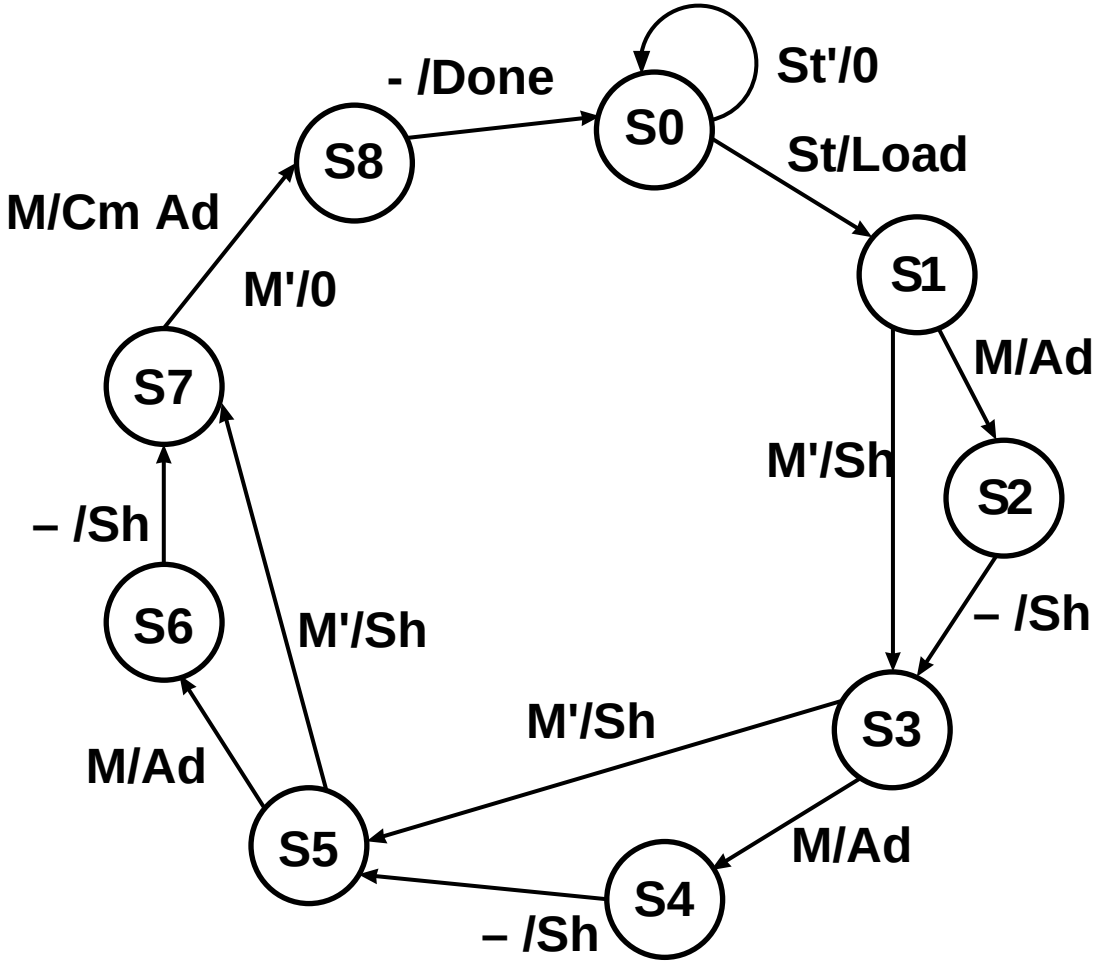
```
          1. 1  0  1      (–3/8)
        X 1. 1  0  1      (–3/8)
-------------------------------------
(1. 1  1) 1  1  0  1      (–3/64)
(1.)1  1  0  1            (–3/16)
-------------------------------------
(1.)1  1  0  0  0  1
 0. 0  1  1               (+3/8)
-------------------------------------
 0. 0  0  1  0  0  1      (+9/64)
```

← Add the two's complement of the multiplicand

# Figure 4-8 Block Diagram for 2's Complement Multiplier



# Figure 4-9 State Graph for 2's Complement Multiplier



# Figure 4-10 Block Diagram for Faster Multiplier

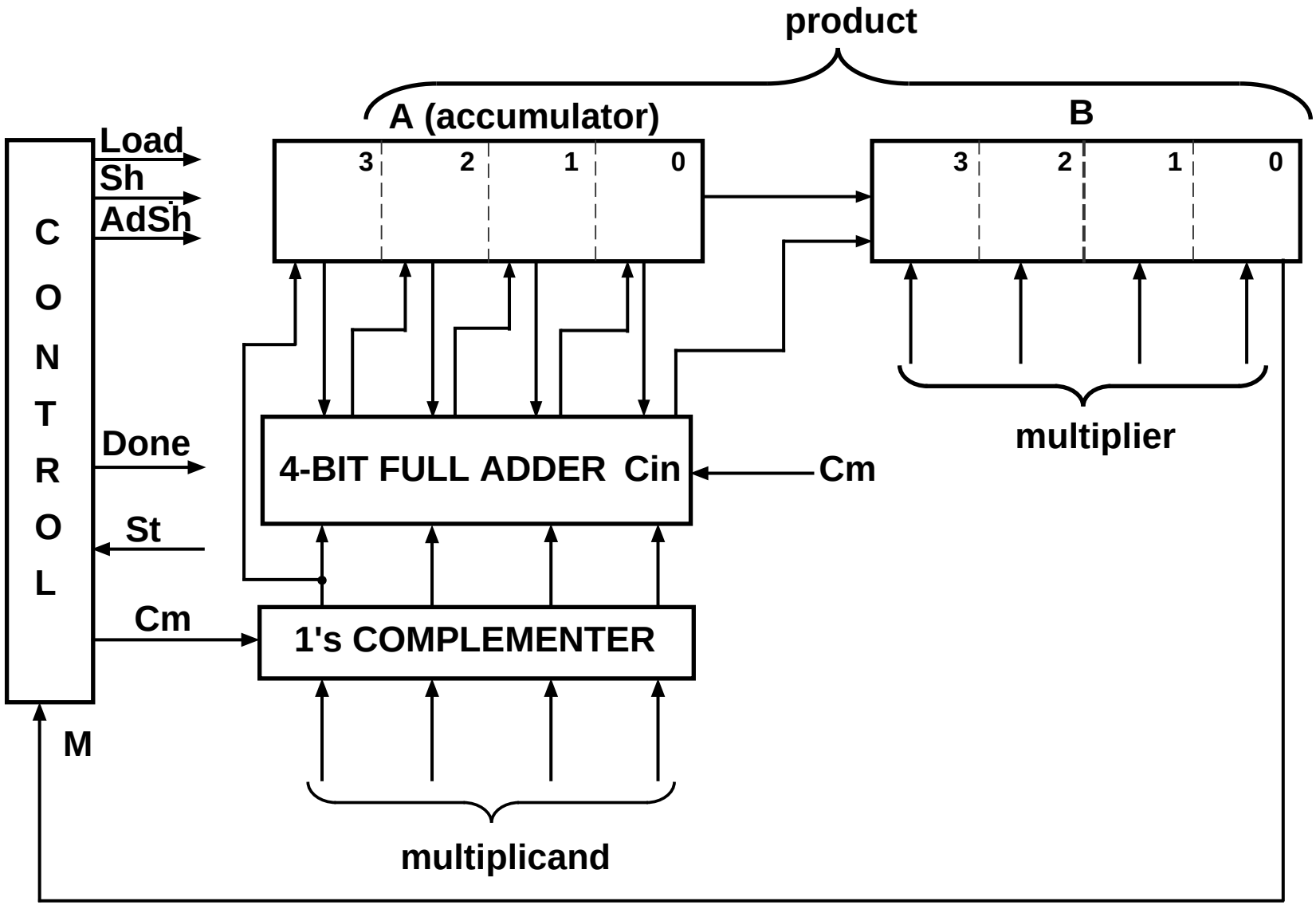
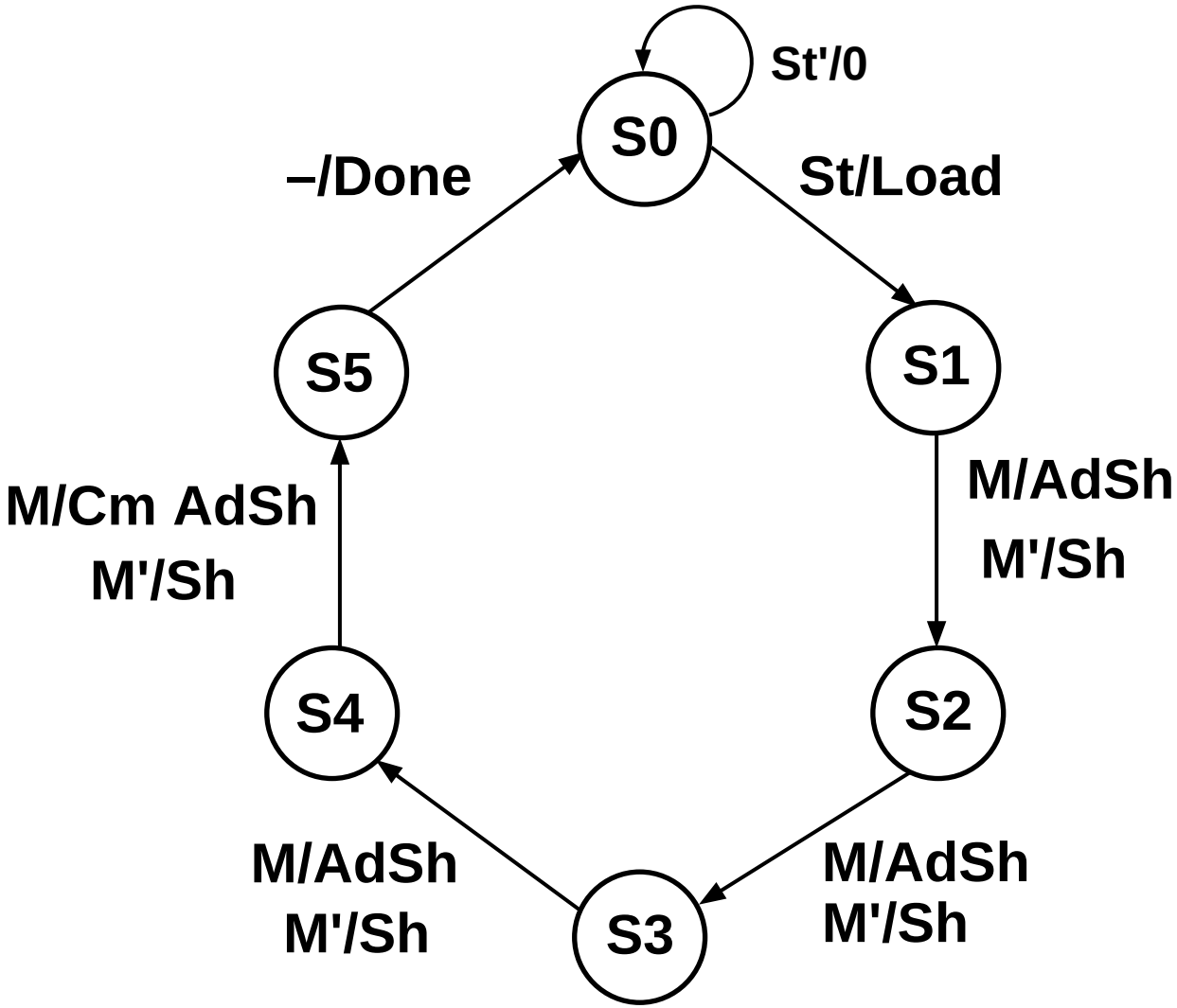


**Figure 4-11 State Graph for Faster Multiplier**



## Figure 4-12(a) Behavioral Model for 2's Complement Multiplier

```
library BITLIB;
use BITLIB.bit_pack.all;

entity mult2C is
      port (CLK, St: in bit;
            Mplier,Mcand : in bit_vector(3 downto 0);
            Product: out bit_vector (6 downto 0);
            Done: out bit);
end mult2C;

architecture behave1 of mult2C is
      signal State : integer range 0 to 5;
      signal A, B: bit_vector(3 downto 0);
      alias M: bit is B(0);
begin
      process
      variable addout: bit_vector(4 downto 0);
      begin
            wait until CLK = '1';
            case State is
                  when 0=>                          -- initial State
                        if St='1' then
                              A <= "0000";          -- Begin cycle
                              B <= Mplier;          -- load the multiplier
                              State <= 1;
                        end if;
```

## Figure 4-12(b) Behavioral Model for 2's Complement Multiplier

```
            when 1 | 2 | 3  =>                          -- "add/shift" State
                if M = '1' then
                    addout := add4(A,Mcand,'0'); -- Add multiplicand to A and shift
                    A <= Mcand(3) & addout(3 downto 1);
                    B <= addout(0) & B(3 downto 1);
                else
                    A <= A(3) & A(3 downto 1);  -- Arithmetic right shift
                    B <= A(0) & B(3 downto 1);
                end if;
                State <= State + 1;
            when 4  =>                                  -- add complement if sign bit
                if M = '1' then                         -- of multiplier is 1
                    addout := add4(A, not Mcand,'1');
                    A <= not Mcand(3) & addout(3 downto 1);
                    B <= addout(0) & B(3 downto 1);
                else
                    A <= A(3) & A(3 downto 1);  -- Arithmetic right shift
                    B <= A(0) & B(3 downto 1);
                end if;
                State <= 5;  wait for 0 ns;
                    Done <= '1';  Product <= A(2 downto 0) & B;
            when 5 =>                                   -- output product
                State <= 0;
                Done <= '0';
        end case;
    end process;
end behave1;
```

## Figure 4-13 Command File and Simulation Results for (+5/8 by -3/8)

```
-- command file to test signed multiplier
list CLK St State A B Done Product
force st 1 2, 0 22
force clk 1 0, 0 10 - repeat 20
-- (5/8 * -3/8)
force Mcand 0101
force Mplier 1101
run 120
```

| 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 |

## Figure 4-14 Test Bench for Signed Multiplier

```
library BITLIB;
use BITLIB.bit_pack.all;
entity testmult is  end testmult;

architecture test1 of testmult is
component mult2C
      port(CLK, St: in bit;
            Mplier,Mcand : in bit_vector(3 downto 0);
            Product: out bit_vector (6 downto 0);
            Done: out bit);
end component;
      constant N: integer := 11;  type arr is array(1 to N) of bit_vector(3 downto 0);
      constant Mcandarr: arr := ("0111", "1101", "0101", "1101", "0111", "1000", "0111",
            "1000", "0000", "1111", "1011");
      constant Mplierarr: arr := ("0101", "0101", "1101", "1101", "0111", "0111", "1000",
            "1000", "1101", "1111", "0000");
      signal CLK, St, Done: bit;  signal Mplier, Mcand: bit_vector(3 downto 0);
      signal Product: bit_vector(6 downto 0);
begin
      CLK <= not CLK after 10 ns;
      process
      begin
            for i in 1 to N loop
                  Mcand <= Mcandarr(i);  Mplier <= Mplierarr(i);  St <= '1';
                  wait until rising_edge(CLK);  St <= '0';  wait until falling_edge(Done);
            end loop;
      end process;
      mult1: mult2c port map(Clk, St, Mplier, Mcand, Product, Done);
end test1;
```

## Figure 4-15 Command File and Simulation of Signed Multiplier

-- Command file to test results of signed multiplier

list -NOtrigger Mplier Mcand product -Trigger done
run 1320

```
  ns  delta mplier mcand product done
   0     +1   0101  0111 0000000    0
  90     +2   0101  0111 0100011    1   5/8 * 7/8 = 35/64
 110     +2   0101  1101 0100011    0
 210     +2   0101  1101 1110001    1   5/8 * -3/8 = -15/64
 230     +2   1101  0101 1110001    0
 330     +2   1101  0101 1110001    1   -3/8 * 5/8 = -15/64
 350     +2   1101  1101 1110001    0
 450     +2   1101  1101 0001001    1   -3/8 * -3/8 = 9/64
 470     +2   0111  0111 0001001    0
 570     +2   0111  0111 0110001    1   7/8 * 7/8 = 49/64
 590     +2   0111  1000 0110001    0
 690     +2   0111  1000 1001000    1   7/8 * -1 = -7/8
 710     +2   1000  0111 1001000    0
 810     +2   1000  0111 1001000    1   -1 * 7/8 = -7/8
 830     +2   1000  1000 1001000    0
 930     +2   1000  1000 1000000    1   -1 * -1 = -1 (error)
 950     +2   1101  0000 1000000    0
1050     +2   1101  0000 0000000    1   -3/8 * 0 = 0
1070     +2   1111  1111 0000000    0
1170     +2   1111  1111 0000001    1   -1/8 * -1/8 = 1/64
1190     +2   0000  1011 0000001    0
1290     +2   0000  1011 0000000    1   0 * -3/8 = 0
1310     +2   0101  0111 0000000    0
```

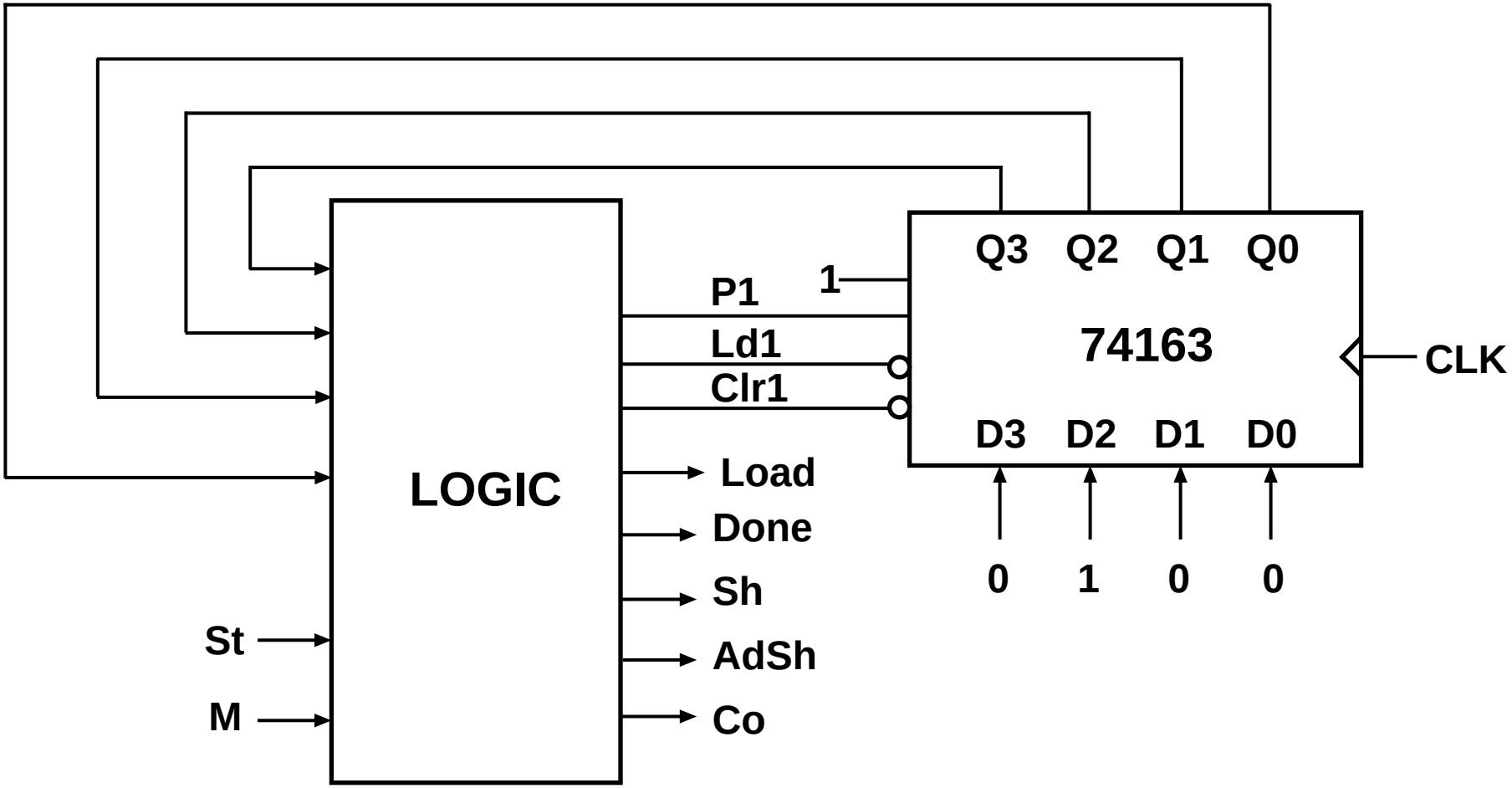
**Figure 4-16(a) [revised] Model for 2's Complement Multiplier with Control Signals**

```
library BITLIB;
use BITLIB.bit_pack.all;

entity mult2Cs is
port (CLK, St: in bit;
      Mplier,Mcand : in bit_vector(3 downto 0);
      Product: out bit_vector (6 downto 0);  Done: out bit);
end mult2Cs;

architecture behave2 of mult2Cs is
      signal State, Nextstate: integer range 0 to 5;  signal A, B: bit_vector(3 downto 0);
      signal AdSh, Sh, Load, Cm: bit;  signal addout: bit_vector(4 downto 0);
      alias M: bit is B(0);
begin
      process (state, st, M)
      begin
            Load <= '0'; AdSh <= '0'; Sh <= '0'; Cm <= '0'; Done <= '0';
            case State is
                  when 0=>                                          -- initial State
                        if St='1' then Load <= '1'; Nextstate <= 1;  end if;
                  when 1 | 2 | 3  =>                                -- "add/shift" State
                        if M = '1' then AdSh <= '1';  else Sh <= '1';  end if;
                        Nextstate <= State + 1;
                  when 4  =>                                        -- add complement if sign
                        if M = '1' then Cm <= '1'; AdSh <= '1';-- bit of multiplier is 1
                        else Sh <= '1';  end if;
                        nextstate <= 5;
```

**Figure 4-16(b) [revised] Model for 2's Complement Multiplier with Control Signals**

```
            when 5 =>                                   -- Output product
                done <= '1';
                nextstate <= 0;
        end case;
    end process;

    addout <= add4(A, Mcand, '0') when Cm = '0'  else add4(A, not Mcand, '1');

    process
    begin
        wait until CLK = '1';                           -- executes on rising edge
        if Load = '1' then                              -- Load the multiplier
            A <= "0000";
            B <= Mplier;
        end if;
        if AdSh = '1' then                              -- Add multiplicand to A and Shift
            A <= (Mcand(3) xor Cm) & addout(3 downto 1);
            B <= addout(0) & B(3 downto 1);
        end if;
        if Sh = '1' then
            A <= A(3) & A(3 downto 1);  B <= A(0) & B(3 downto 1);
        end if;
        State <= Nextstate;
    end process;

    Product <= A(2 downto 0) & B;
end behave2;
```

# Figure 4-17 Realization of Multiplier Control Network



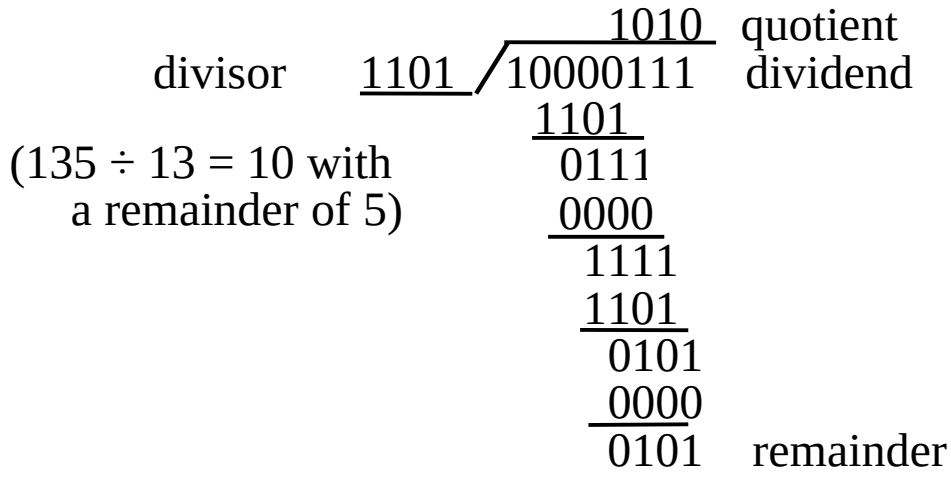
## Figure 4-18(a) Model for 2's Complement Multiplier Using Control Equations

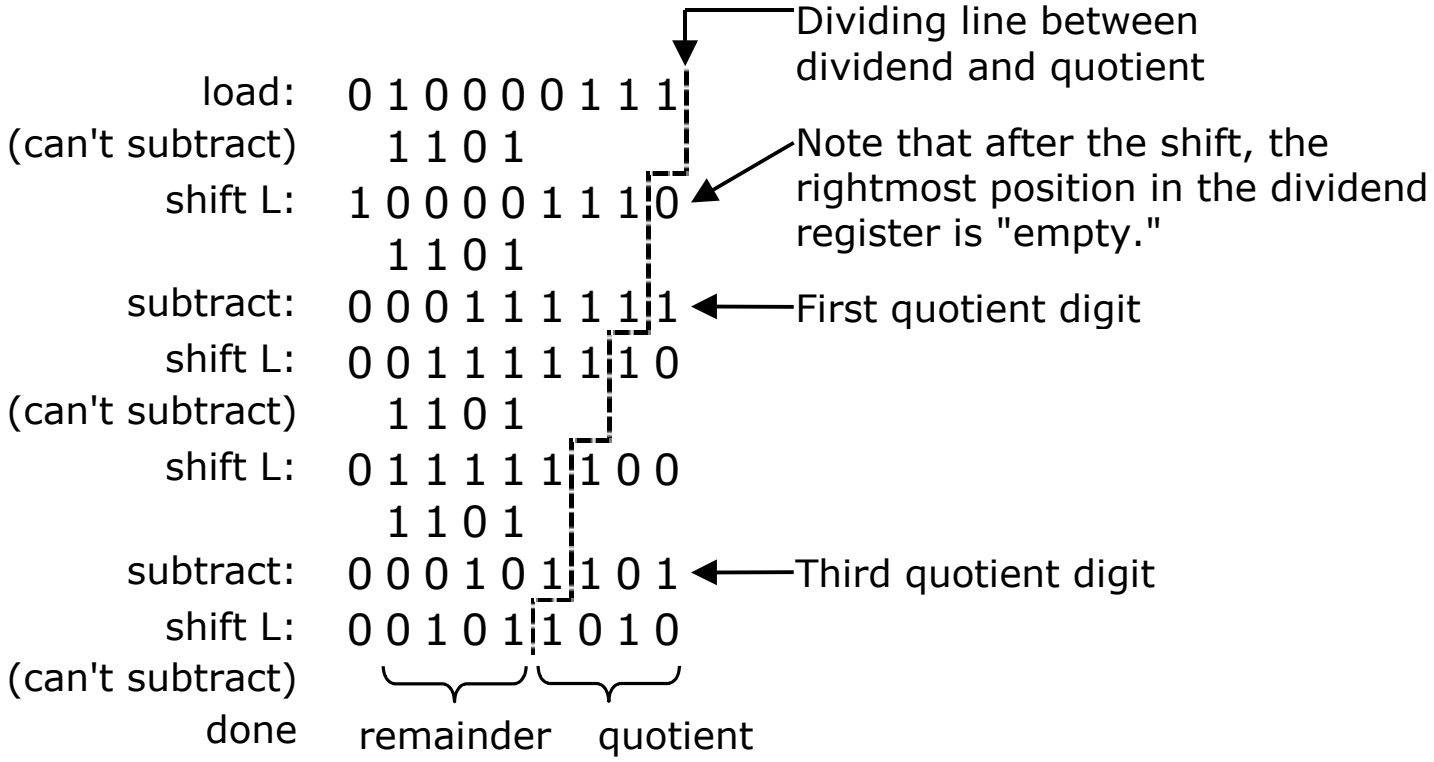
```
-- This model of a 4-bit multiplier for 2's complement numbers
-- implements the controller using a counter and logic equations.
library BITLIB;
use BITLIB.bit_pack.all;

entity mult2CEQ is
      port(CLK, St: in bit;
           Mplier,Mcand: in bit_vector(3 downto 0);
           Product: out bit_vector(6 downto 0));
end mult2CEQ;

architecture m2ceq of mult2CEQ is
      signal A, B, Q, Comp: bit_vector(3 downto 0);
      signal addout: bit_vector(4 downto 0);
      signal AdSh, Sh, Load, Cm, Done, Ld1, CLR1, P1: bit;
      Signal One: bit:='1';
      Signal Din: bit_vector(3 downto 0) := "0100";
      alias M: bit is B(0);
begin
      Count1: C74163 port map (Ld1, CLR1, P1, One, CLK, Din, open, Q);
      P1 <= Q(2);  CLR1 <= not Q(3);  Done <= Q(3);  Sh <= not M and Q(2);
      AdSh <= M and Q(2);  Cm <= Q(1) and Q(0) and M;
      Load <= not Q(3) and not Q(2) and St; Ld1 <= not Load;
      Comp <= Mcand xor (Cm & Cm & Cm & Cm);      -- complement Mcand if Cm='1'
      addout <= add4(A,Comp,Cm);                 -- add complementer output to A
```

## Figure 4-18(b) Model for 2's Complement Multiplier Using Control Equations

```
    process
    begin
        wait until CLK = '1';                    -- executes on rising edge
        if Load = '1' then                       -- load the multiplier
            A <= "0000";
            B <= Mplier;
        end if;
        if AdSh = '1' then                       -- Add multiplicand to A and shift
            A <= (Mcand(3) xor Cm) & addout(3 downto 1);
            B <= addout(0) & B(3 downto 1);
        end if;
        if Sh = '1' then                         -- Right shift with sign extend
            A <= A(3) & A(3 downto 1);
            B <= A(0) & B(3 downto 1);
        end if;
        if Done = '1' then
            Product <= A(2 downto 0) & B;
        end if;
    end process;
end m2ceq;
```

## Parallel Divider for Positive Binary Numbers – *From Page 144*



# Figure 4-19 Block Diagram for Parallel Binary Divider

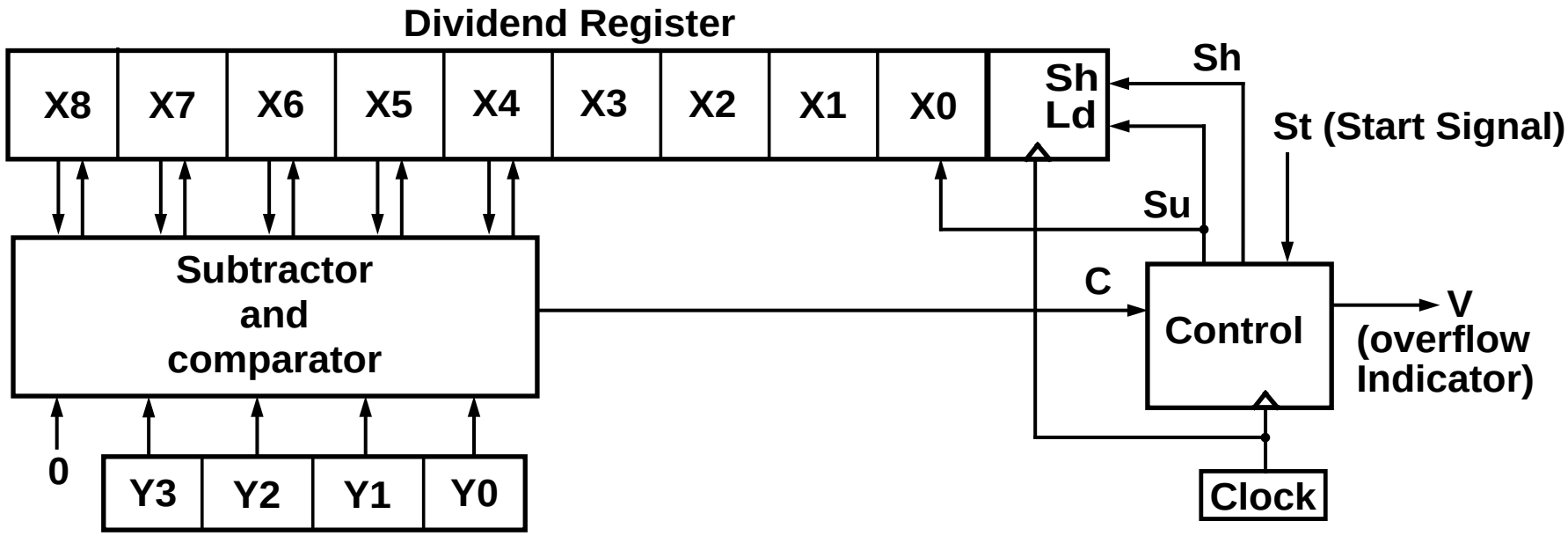
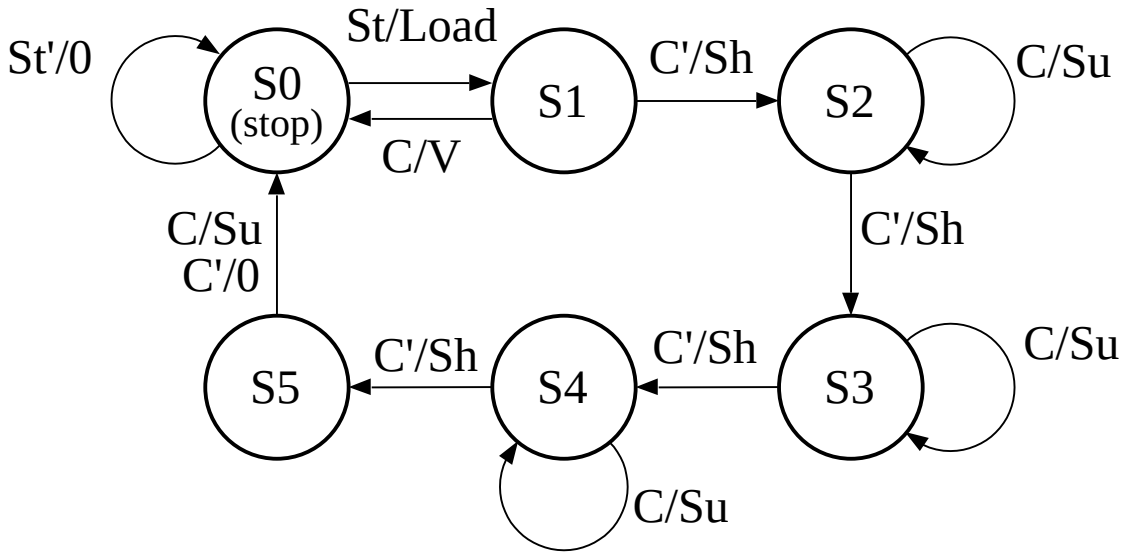


**Figure 4-20 State Diagram for Divider Control Circuit**




| State | StC 00 | 01 | 11 | 10 | StC 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* | – | – |

## Control Signals for Signed Divider

| | |
|---|---|
| *LdU* | Load upper half of dividend from bus |
| *LdL* | Load lower half of dividend from bus |
| *Lds* | Load sign of dividend into sign flip-flop |
| *S* | Sign of dividend |
| *Cm1* | Complement dividend register (2's complement) |
| *Ldd* | Load divisor from bus |
| *Su* | Enable adder output onto bus (Ena) and load upper half of dividend from bus |
| *Cm2* | Enable complementer (Cm2 equals the complement of the sign bit of the divisor, so that a positive divisor is complemented and a negative divisor is not) |
| *Sh* | Shift the dividend register left one place and increment the counter |
| *C* | Carry output from adder (If C = 1, the divisor can be subtracted from the upper dividend.) |
| *St* | Start |
| *V* | Overflow |
| *Qneg* | Quotient will be negative (Qneg = 1 when sign of dividend and divisor are different) |

# Figure 4-21 Block Diagram for Signed Divider

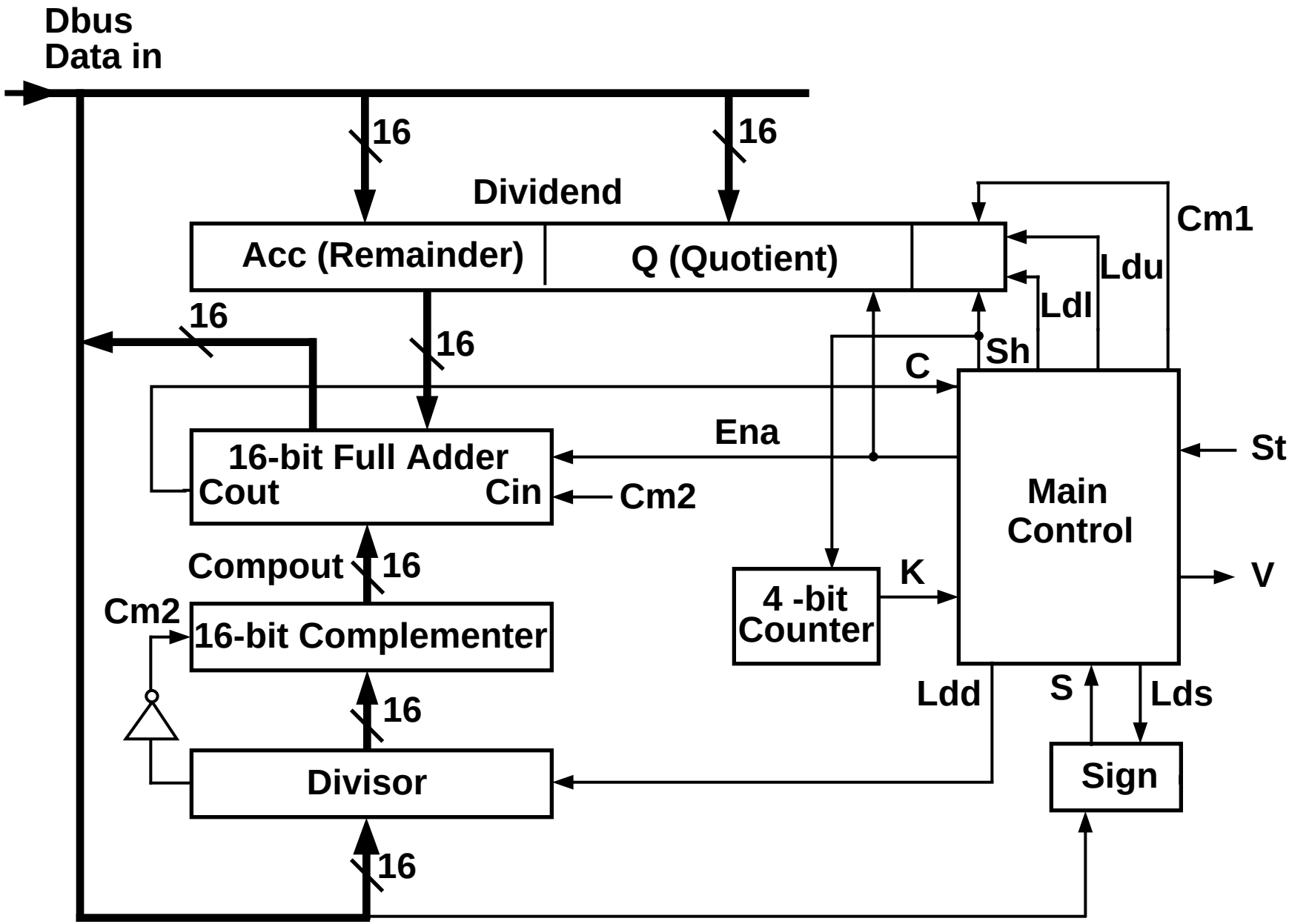
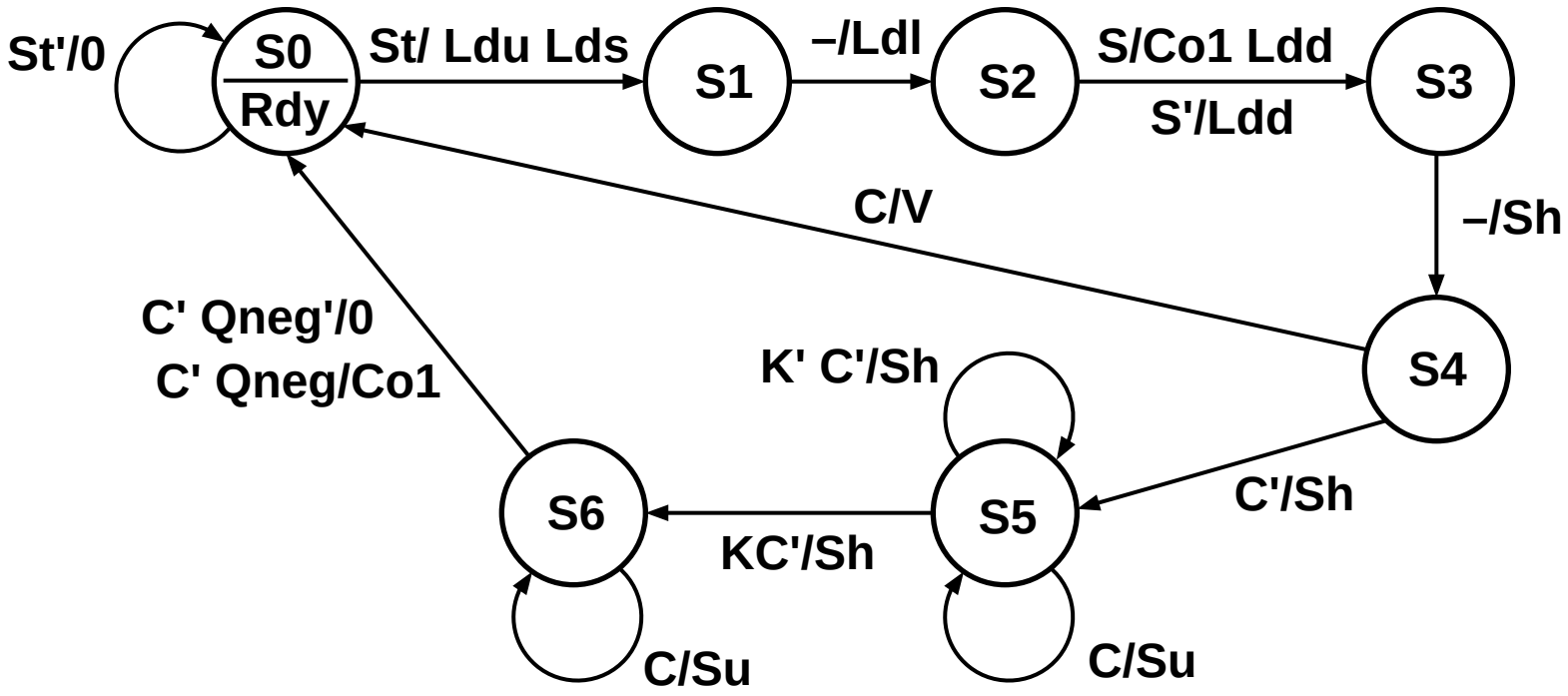


**Figure 4-22 State Graph for Signed Divider Control Network**



## Figure 4-23(a) VHDL Model of 32-bit Signed Divider

```
library BITLIB;
use BITLIB.bit_pack.all;
entity sdiv is
      port(Clk,St: in bit;
            Dbus: in bit_vector(15 downto 0); Quotient: out bit_vector(15 downto 0);
            V, Rdy: out bit);
end sdiv;
architecture Signdiv of Sdiv is
      constant zero_vector: bit_vector(31 downto 0):=(others=>'0');
      signal State: integer range 0 to 6;  signal Count : integer range 0 to 15;
      signal Sign,C,NC: bit;  signal Divisor,Sum,Compout: bit_vector(15 downto 0);
      signal Dividend: bit_vector(31 downto 0);
      alias Q: bit_vector(15 downto 0) is Dividend(15 downto 0);
      alias Acc: bit_vector(15 downto 0) is Dividend(31 downto 16);
begin                                                    -- concurrent statements
      compout <= divisor when divisor(15) = '1'          -- 1's complementer
            else not divisor;
      Addvec(Acc,compout,not divisor(15),Sum,C,16);  -- 16-bit adder
      Quotient <= Q; Rdy <= '1' when State=0 else '0';
```

## Figure 4-23(b) VHDL Model of 32-bit Signed Divider

```
process
begin
    wait until Clk = '1';                          -- wait for rising edge of clock
    case State is
        when 0=>
            if St = '1' then
                Acc <= Dbus;                       -- load upper dividend
                Sign <= Dbus(15);  State <= 1;
                V <= '0';  Count <= 0;             -- initialize overflow// initialize counter
            end if;
        when 1=>
            Q <= Dbus;  State <= 2;                -- load lower dividend
        when 2=>
            Divisor <= Dbus;
            if Sign ='1'then                       -- two's complement Dividend if necessary
                addvec(not Dividend,zero_vector,'1',Dividend,NC,32);
            end if;  State <= 3;
        when 3=>
            Dividend <= Dividend(30 downto 0) & '0';        -- left shift
            Count <= Count+1;  State <= 4;
        when 4 =>
            if C ='1' then                                  -- C
                v <= '1';  State <= 0;
            else                                            -- C'
                Dividend <= Dividend(30 downto 0) & '0';    -- left shift
                Count <= Count+1;  State <= 5;
            end if;
```

## Figure 4-23(c) VHDL Model of 32-bit Signed Divider

```
            when 5 =>
                if C = '1' then                                          -- C
                    ACC <= Sum;                                          -- subtract
                    Q(0)<= '1';
                else
                    Dividend <= Dividend(30 downto 0) & '0';             -- left shift
                    if Count = 15 then                                   -- KC'
                        count<= 0; State <= 6;
                    else Count <= Count+1;
                    end if;
                end if;
        when 6=>
            if C = '1' then                                              -- C
                Acc <= Sum;                                              -- subtract
                Q(0) <= '1';
                else if (Sign xor Divisor(15))='1' then                  -- C'Qneg
                    addvec(not Dividend,zero_vector,'1',Dividend,NC,32);
                end if;                                    -- 2's complement Dividend
                state <= 0;
            end if;
        end case;
    end process;
end signdiv;
```

## Figure 4-24(a) Test Bench for Signed Divider

```
library BITLIB;
use BITLIB.bit_pack.all;
entity testsdiv is
end testsdiv;
architecture test1 of testsdiv is
component sdiv
      port(Clk,St: in bit;
            Dbus: in bit_vector(15 downto 0);  Quotient: out bit_vector(15 downto 0);
            V, Rdy: out bit);
end component;
constant N: integer := 12;                                    -- test sdiv1 N times
type arr1 is array(1 to N) of bit_vector(31 downto 0);
type arr2 is array(1 to N) of bit_vector(15 downto 0);
constant dividendarr: arr1 := (X"0000006F",X"07FF00BB",X"FFFFFE08",
      X"FF80030A",X"3FFF8000",X"3FFF7FFF",X"C0008000",X"C0008000",
      X"C0008001",X"00000000",X"FFFFFFFF",X"FFFFFFFF");
constant divisorarr: arr2 := (X"0007", X"E005", X"001E", X"EFFA", X"7FFF", X"7FFF", X"7FFF",
      X"8000", X"7FFF", X"0001", X"7FFF", X"0000");
signal CLK, St, V, Rdy: bit;  signal Dbus, Quotient, divisor: bit_vector(15 downto 0);
signal Dividend: bit_vector(31 downto 0);  signal count: integer range 0 to N;
```

## Figure 4-24(b) Test Bench for Signed Divider

```
begin
      CLK <= not CLK after 10 ns;
      process
      begin
            for i in 1 to N loop
                  St <= '1';
                  Dbus <= dividendarr(i) (31 downto 16);
                  wait until rising_edge(CLK);
                  Dbus <= dividendarr(i) (15 downto 0);
                  wait until rising_edge(CLK);
                  Dbus <= divisorarr(i);
                  St <= '0';
                  dividend <= dividendarr(i);          -- save dividend for listing
                  divisor <= divisorarr(i);            -- save divisor for listing
                  wait until (Rdy = '1');
                  count <= i;                          -- save index for triggering
      end loop;
      end process;
      sdiv1: sdiv port map(Clk, St, Dbus, Quotient, V, Rdy);
end test1;
```

## Figure 4-25 Simulation Test Results for Signed Divider

-- Command file to test results of signed divider
list -hex -Notrigger dividend divisor Quotient V -Trigger count
run 5300

| ns | delta | dividend | divisor | quotient | v | count |
|---|---|---|---|---|---|---|
| 0 | +0 | 00000000 | 0000 | 0000 | 0 | 0 |
| 470 | +3 | 0000006F | 0007 | 000F | 0 | 1 |
| 910 | +3 | 07FF00BB | E005 | BFFE | 0 | 2 |
| 1330 | +3 | FFFFFE08 | 001E | FFF0 | 0 | 3 |
| 1910 | +3 | FF80030A | EFFA | 07FC | 0 | 4 |
| 2010 | +3 | 3FFF8000 | 7FFF | 0000 | 1 | 5 |
| 2710 | +3 | 3FFF7FFF | 7FFF | 7FFF | 0 | 6 |
| 2810 | +3 | C0008000 | 7FFF | 0000 | 1 | 7 |
| 3510 | +3 | C0008000 | 8000 | 7FFF | 0 | 8 |
| 4210 | +3 | C0008001 | 7FFF | 8001 | 0 | 9 |
| 4610 | +3 | 00000000 | 0001 | 0000 | 0 | A |
| 5010 | +3 | FFFFFFFF | 7FFF | 0000 | 0 | B |
| 5110 | +3 | FFFFFFFF | 0000 | 0002 | 1 | C |