Chapter 4.11-12
Design Examples

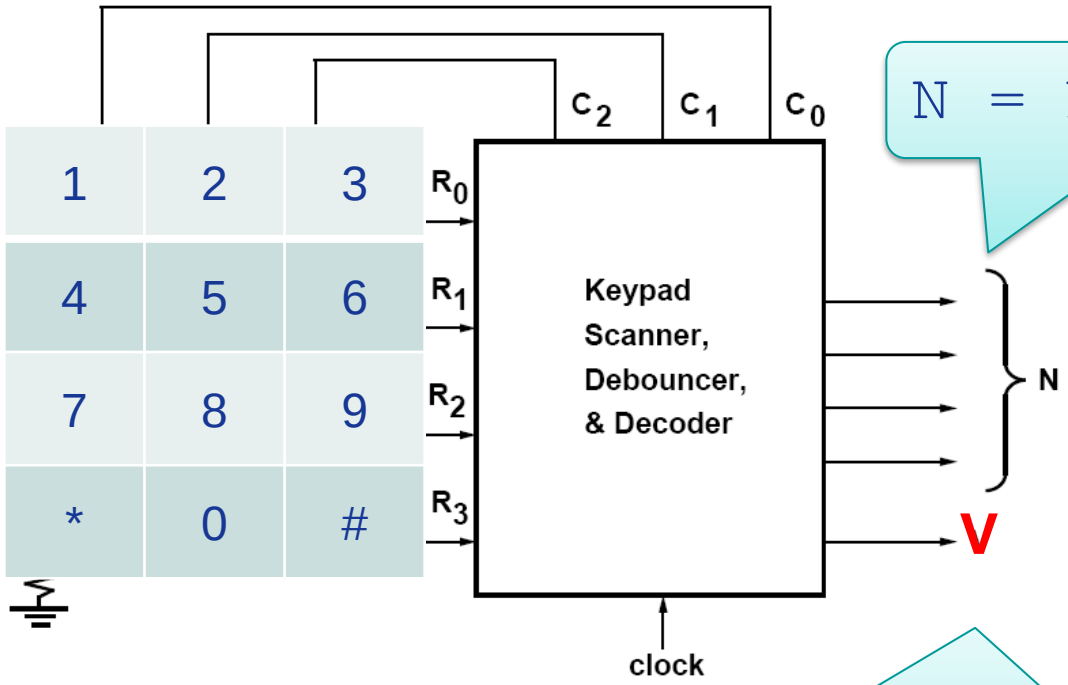
Version: 2023/12/12

Chapter 4 Design Examples

	Contents
1	BCD to Seven-Segment Display Decoder
2	A BCD Adder
3	32-Bit Adders
4	Traffic Light Controller
5	State Graphs for Control Circuits
6	Scoreboard and Controller
7	Synchronization and Debouncing
8	Add-and-Shift Multiplier
9	Array Multiplier
10	A Signed Integer/Fraction Multiplier
11	**Keypad Scanner**
12	Binary Dividers

11 Keypad Scanner

Keypad with Three Columns and Four Rows

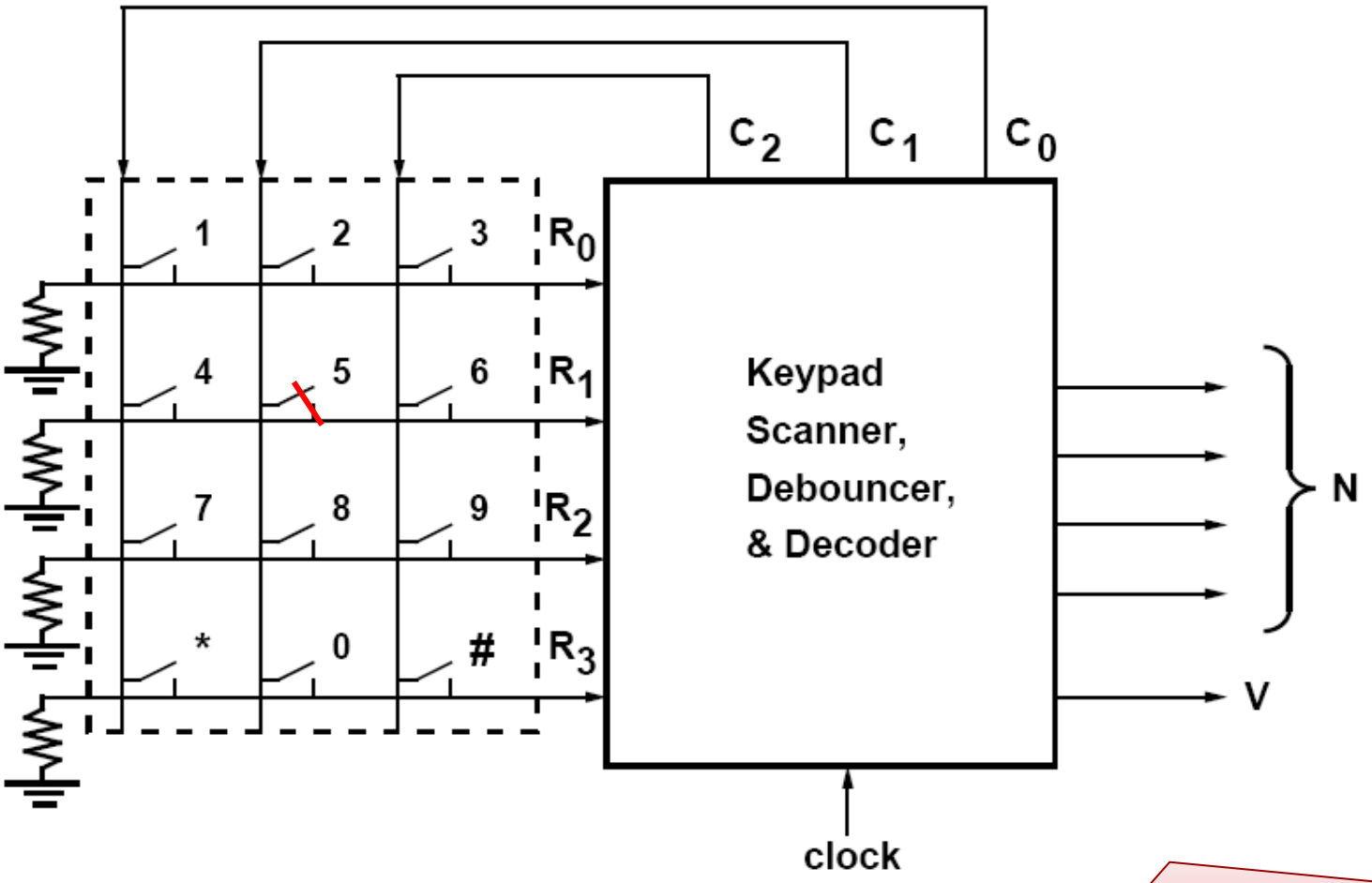


$N = N_3N_2N_1N_0$

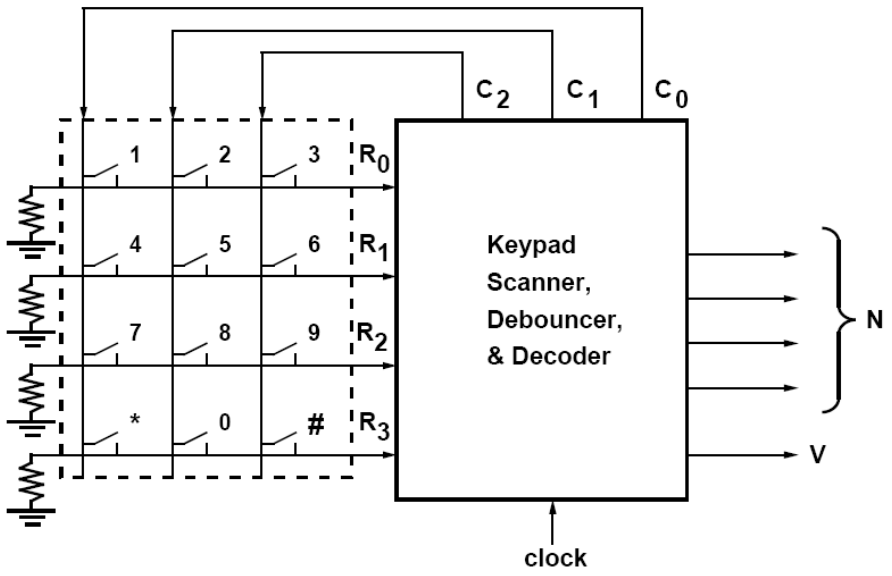
Examples:
5 : 0101
* : 1010
# : 1011

When a valid key has been detected, SCANNER outputs **V** for one clock time

11 Keypad Scanner

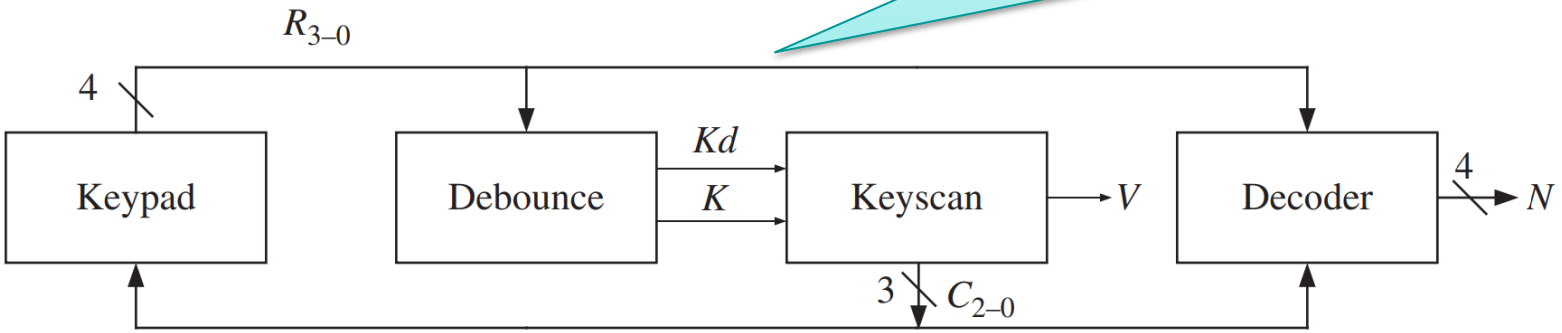


Assumption: only one key is pressed at a time

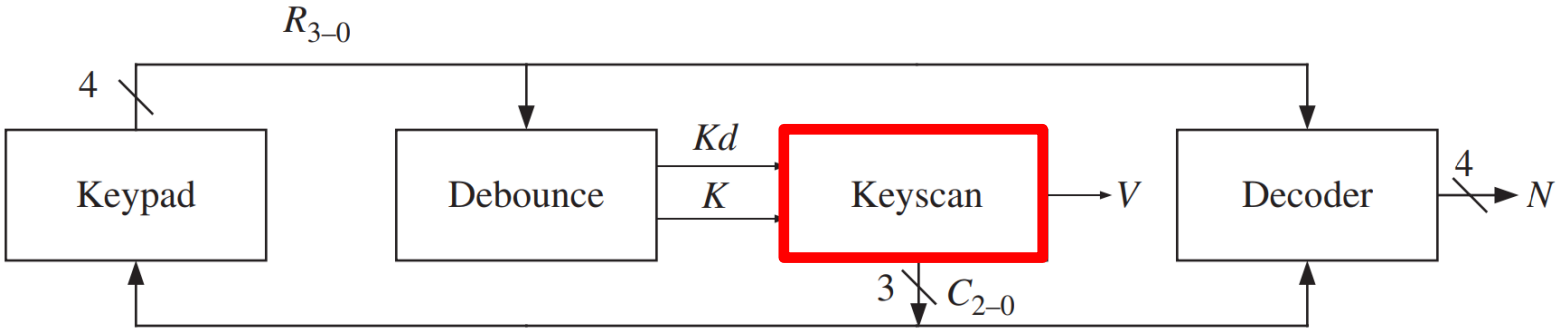


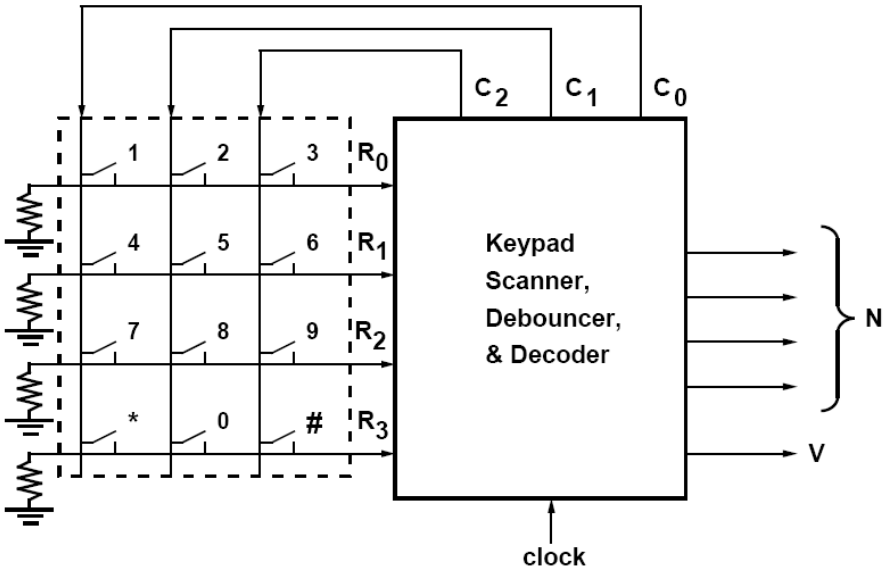
Scanner modules

K : key press
Kd : debounded key press



4.11.1 Scanner

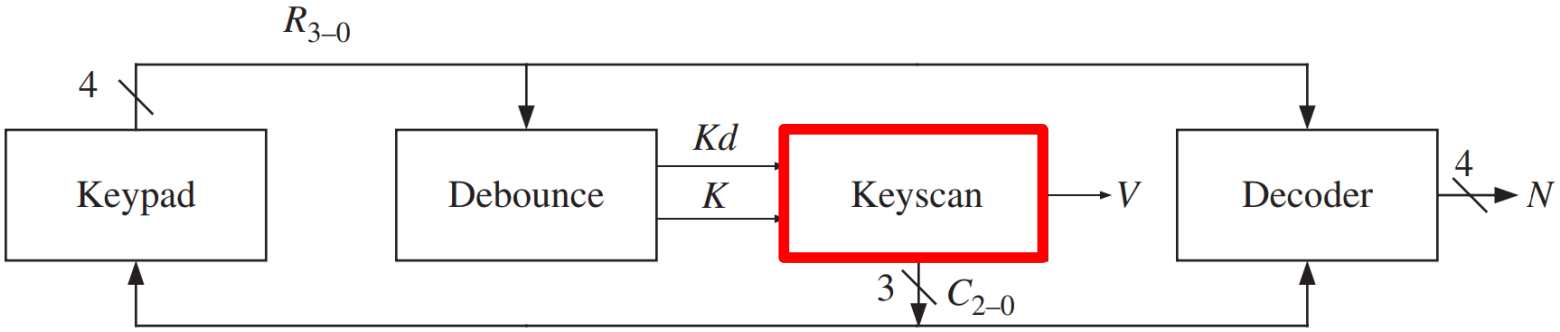


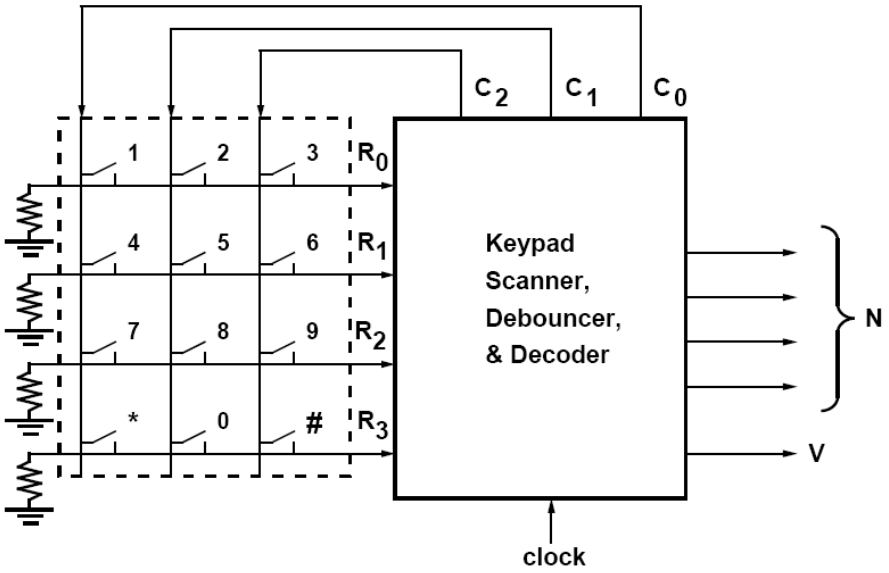


Step 1: Apply logic 1's to C0, C1, C2 and wait

Step 2: If any key is pressed, apply a 1 to C0->C1->C2

4.11.1 Scanner

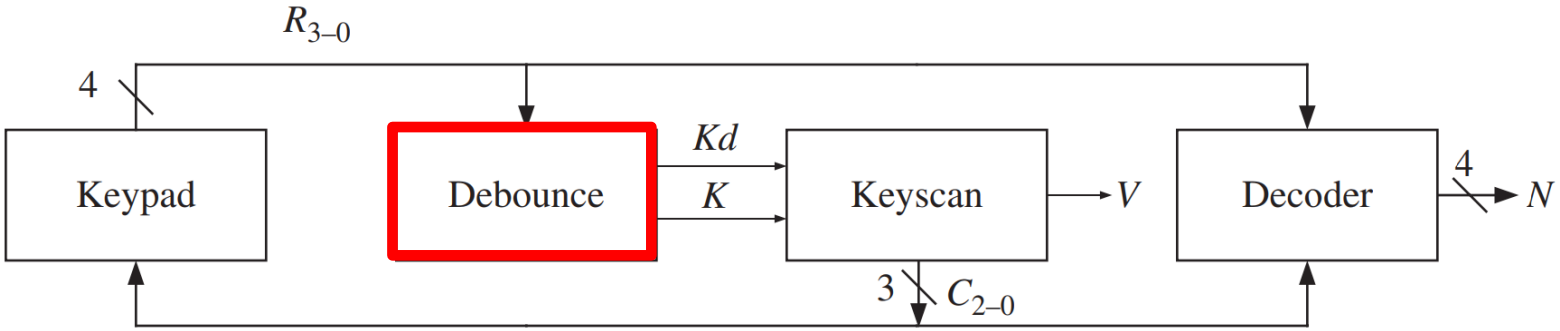




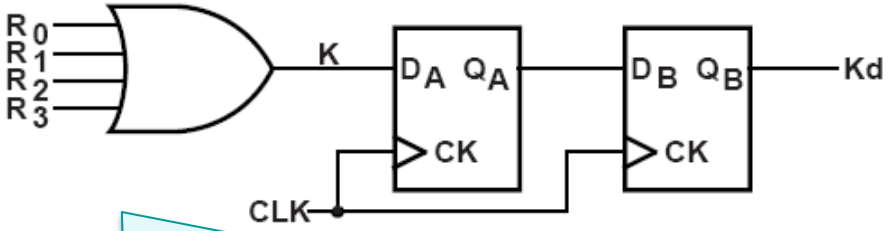
Step 3: If Ri is detected, set V = 1

Step 4: Apply 1's to C0, C1, C2 and wait until no key is pressed

4.11.2 Debouncer



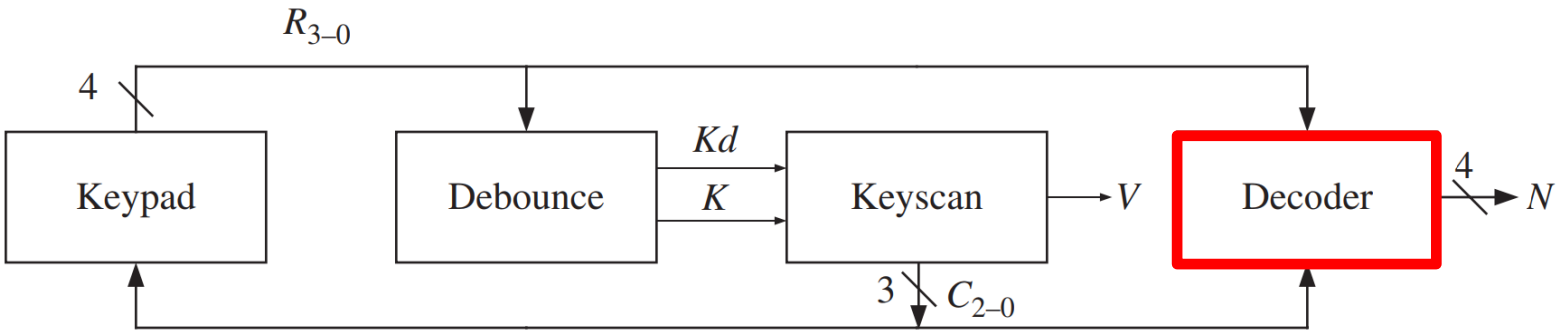
Debouncing and Synchronizing Circuit



- Kd is fed to sequential circuit

- R0~3 are connected to an OR gate to form K
- K turns on when a key is pressed and a column signal is applied

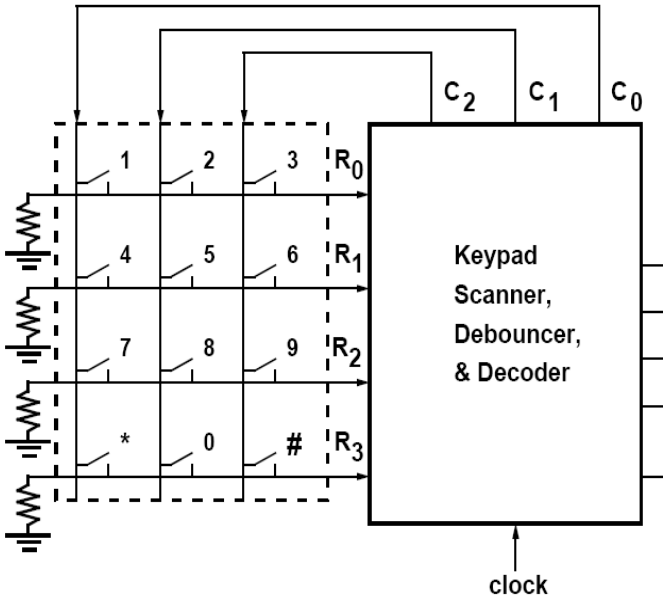
4.11.3 Decoder



- Decoder is combinational circuit, it output will change as the keypad is scanned

- At the a valid key is detected (K=1 and V=1), decoder will have the correct output and this value can be saved in a register

4.11.3 Decoder



Truth Table for Decoder

R_3	R_2	R_1	R_0	C_0	C_1	C_2	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

N3 =
N2 =
N1 =
N0 =

?

4.11.3 Decoder

R_3	R_2	R_1	R_0	C_0	C_1	C_2	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	(#)

\R3R2 R1R0\	00	01	11	10
00	X		X	
01		X	X	X
11	X	X	X	X
10		X	X	X

4.11.3 Decoder

$R_3R_2R_1R_0C_0C_1C_2$	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	(#)

\R3R2 R1R0\	00	01	11	10
00	X		X	
01		X	X	X
11	X	X	X	X
10		X	X	X

4.11.3 Decoder

$R_3R_2R_1R_0C_0C_1C_2$	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	(#)

\R3R2 R1R0\	00	01	11	10
00	X	C0'	X	C1'
01	0	X	X	X
11	X	X	X	X
10	0	X	X	X

4.11.3 Decoder

$R_3R_2R_1R_0C_0C_1C_2$	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	(#)

\R3R2 R1R0\	00	01	11	10
00	X	C0'	X	C1'
01	0	X	X	X
11	X	X	X	X
10	0	X	X	X

N3 = R2C0' + R3C1'

4.11.3 Decoder

$R_3R_2R_1R_0C_0C_1C_2$	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	(#)

\R3R2 R1R0\	00	01	11	10
00	X		X	
01		X	X	X
11	X	X	X	X
10		X	X	X

4.11.3 Decoder

$R_3R_2R_1R_0C_0C_1C_2$	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	(#)

\R3R2 R1R0\	00	01	11	10
00	X	C0	X	0
01	0	X	X	X
11	X	X	X	X
10	1	X	X	X

4.11.3 Decoder

$R_3R_2R_1R_0C_0C_1C_2$	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	(#)

\R3R2 R1R0\	00	01	11	10
00	X	C0	X	0
01	0	X	X	X
11	X	X	X	X
10	1	X	X	X

$N2 = R2C0 + R1$

4.11.3 Decoder

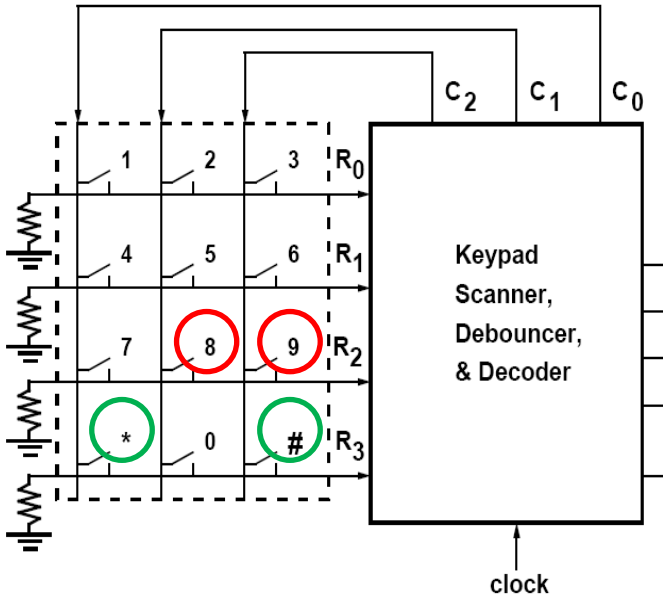
$R_3R_2R_1R_0C_0C_1C_2$	N_3	N_2	N_1	N_0	
0 0 0 1 1 0 0	0	0	0	1	
0 0 0 1 0 1 0	0	0	1	0	
0 0 0 1 0 0 1	0	0	1	1	
0 0 1 0 1 0 0	0	1	0	0	
0 0 1 0 0 1 0	0	1	0	1	
0 0 1 0 0 0 1	0	1	1	0	
0 1 0 0 1 0 0	0	1	1	1	
0 1 0 0 0 1 0	1	0	0	0	
0 1 0 0 0 0 1	1	0	0	1	
1 0 0 0 1 0 0	1	0	1	0	(*)
1 0 0 0 0 1 0	0	0	0	0	
1 0 0 0 0 0 1	1	0	1	1	(#)

\R3R2 R1R0\	00	01	11	10
00	X	C1'	X	C2
01	C1'	X	X	X
11	X	X	X	X
10	C1	X	X	X

$$N0 = R1C1 + R3C2 + R3'R1'C1'$$

4.11.3 Decoder

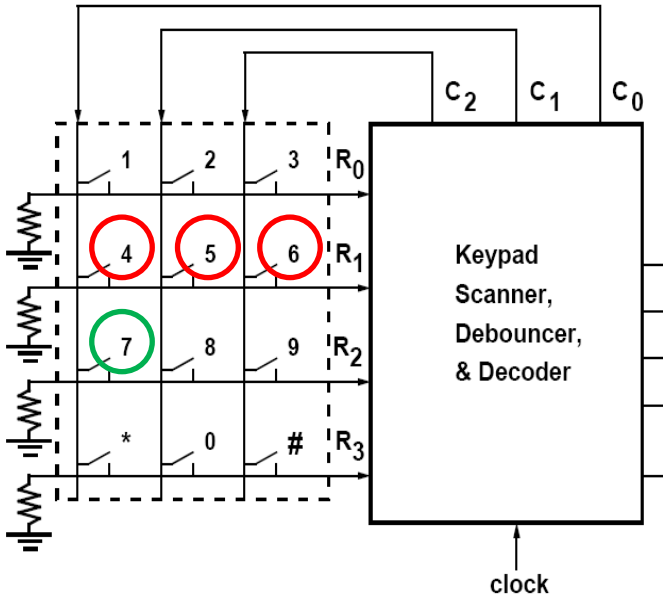
$R_3R_2R_1R_0C_0C_1C_2$	N_3	N_2	N_1	N_0	
0 0 0 1 1 0 0	0	0	0	1	
0 0 0 1 0 1 0	0	0	1	0	
0 0 0 1 0 0 1	0	0	1	1	
0 0 1 0 1 0 0	0	1	0	0	
0 0 1 0 0 1 0	0	1	0	1	
0 0 1 0 0 0 1	0	1	1	0	
0 1 0 0 1 0 0	0	1	1	1	
0 1 0 0 0 1 0	1	0	0	0	
0 1 0 0 0 0 1	1	0	0	1	
1 0 0 0 1 0 0	1	0	1	0	(*)
1 0 0 0 0 1 0	0	0	0	0	
1 0 0 0 0 0 1	1	0	1	1	(#)



N3 = R2C0' + R3C1'

4.11.3 Decoder

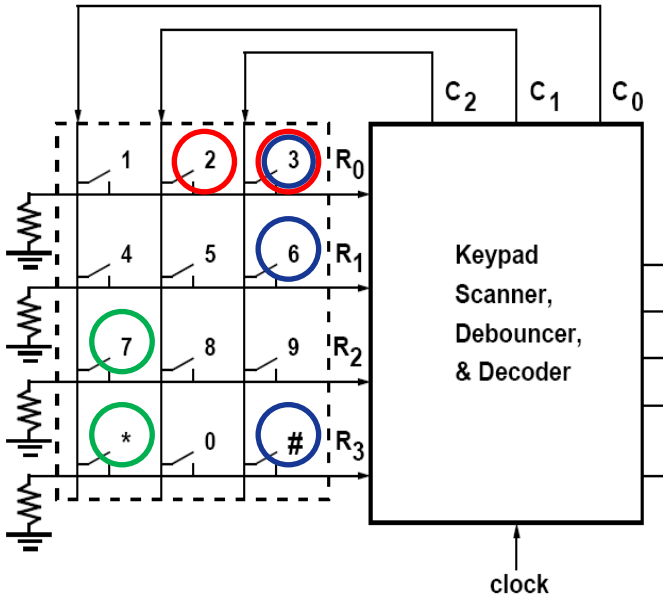
$R_3R_2R_1R_0C_0C_1C_2$	N_3	N_2	N_1	N_0	
0 0 0 1 1 0 0	0	0	0	1	
0 0 0 1 0 1 0	0	0	1	0	
0 0 0 1 0 0 1	0	0	1	1	
0 0 1 0 1 0 0	0	1	0	0	
0 0 1 0 0 1 0	0	1	0	1	
0 0 1 0 0 0 1	0	1	1	0	
0 1 0 0 1 0 0	0	1	1	1	
0 1 0 0 0 1 0	1	0	0	0	
0 1 0 0 0 0 1	1	0	0	1	
1 0 0 0 1 0 0	1	0	1	0	(*)
1 0 0 0 0 1 0	0	0	0	0	
1 0 0 0 0 0 1	1	0	1	1	(#)



$N2 = R1 + R2C0$

4.11.3 Decoder

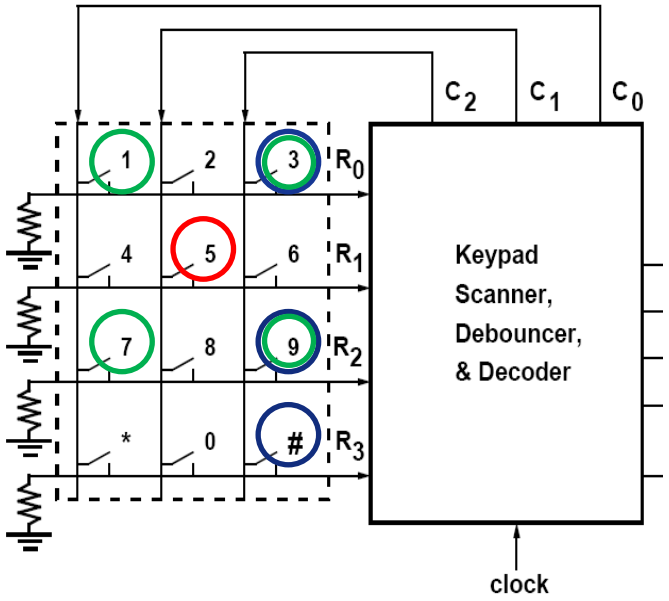
$R_3R_2R_1R_0C_0C_1C_2$	N_3	N_2	N_1	N_0	
0 0 0 1 1 0 0	0	0	0	1	
0 0 0 1 0 1 0	0	0	1	0	
0 0 0 1 0 0 1	0	0	1	1	
0 0 1 0 1 0 0	0	1	0	0	
0 0 1 0 0 1 0	0	1	0	1	
0 0 1 0 0 0 1	0	1	1	0	
0 1 0 0 1 0 0	0	1	1	1	
0 1 0 0 0 1 0	1	0	0	0	
0 1 0 0 0 0 1	1	0	0	1	
1 0 0 0 1 0 0	1	0	1	0	(*)
1 0 0 0 0 1 0	0	0	0	0	
1 0 0 0 0 0 1	1	0	1	1	(#)



N1 = R0C0' + R2'C2 + R0'R1'C0

4.11.3 Decoder

$R_3R_2R_1R_0C_0C_1C_2$	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	(#)



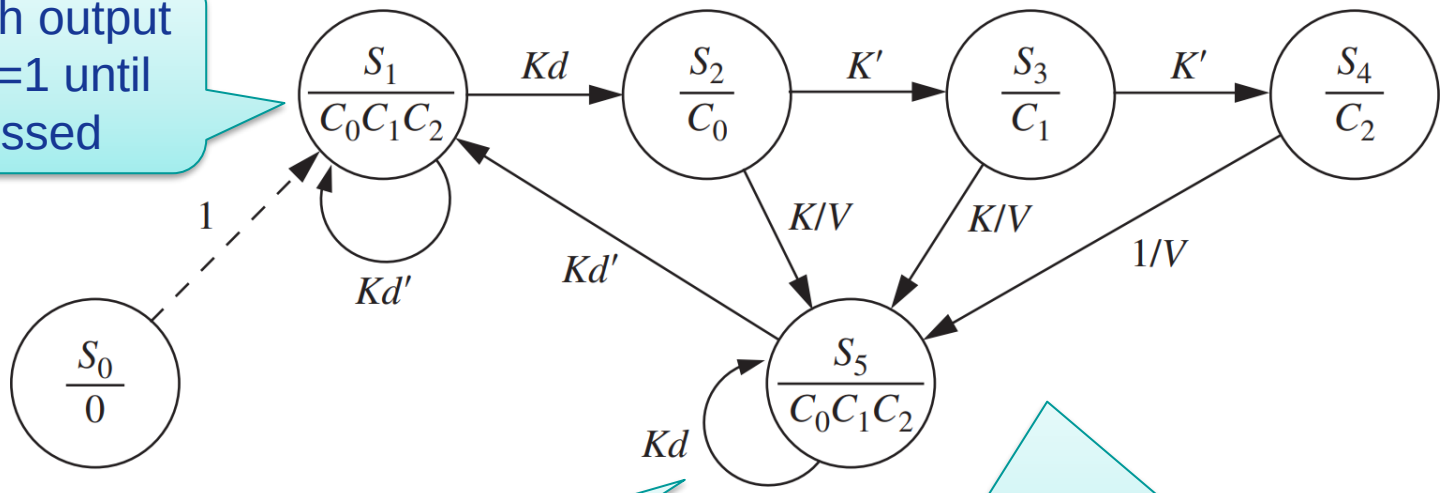
N0 = R1C1 + R1'C2 + R3'R1'C1'

4.11.4 Controller

State graph for Keypad Scanner_V1

S2: C0=1, so if the pressed key is in column 0, K = 1, and the circuit outpus V=1 and goes to S5

S1: wait with output C0=C1=C2=1 until a key is pressed

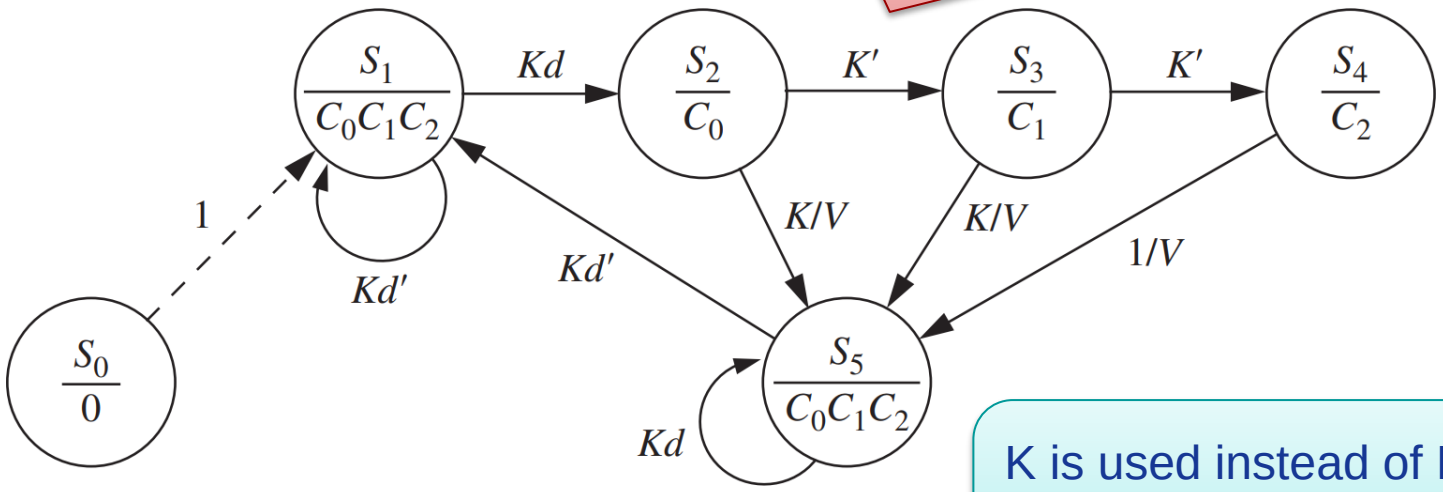


S5: The circuit waits until all key are released and Kd goes to 0 before resetting

S3, S4: If no key is found in column 0, column 1 is checked, and if necessary, column 2 is checked

4.11.4 Controller

State graph for Keypad Scanner_V1

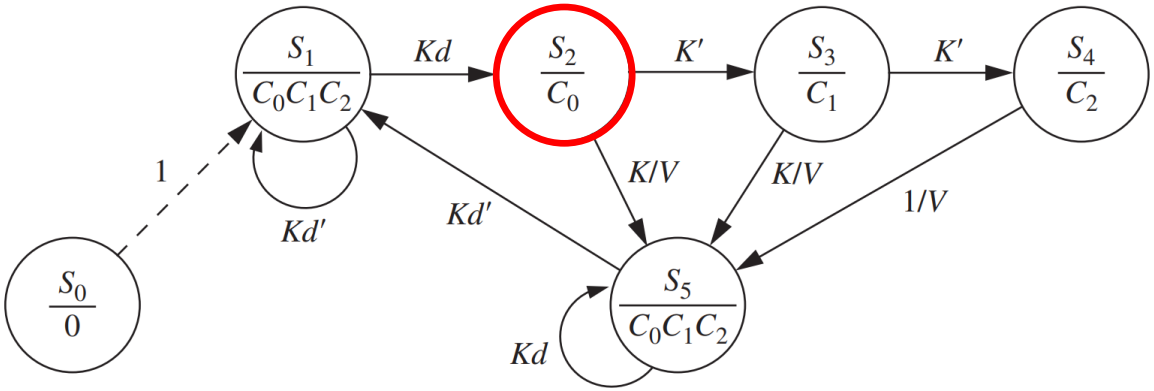


Why we use K here instead of Kd?

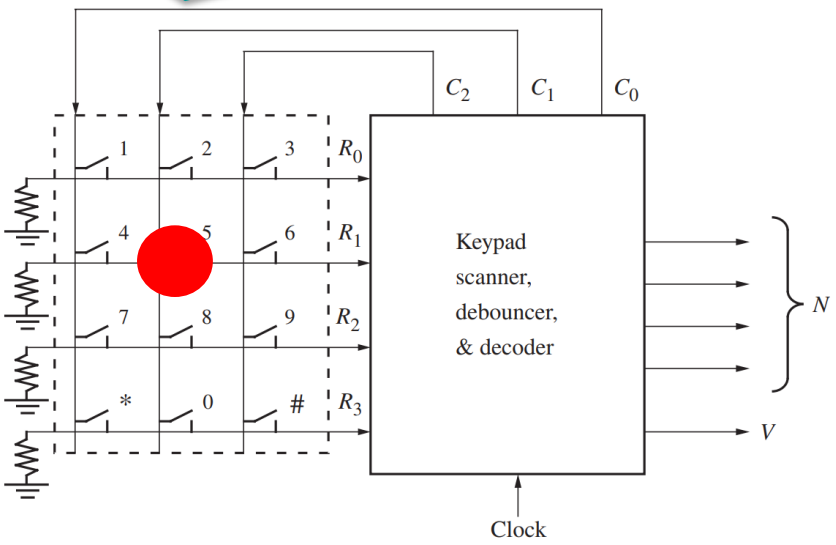
K is used instead of Kd, since the key press is already debounced

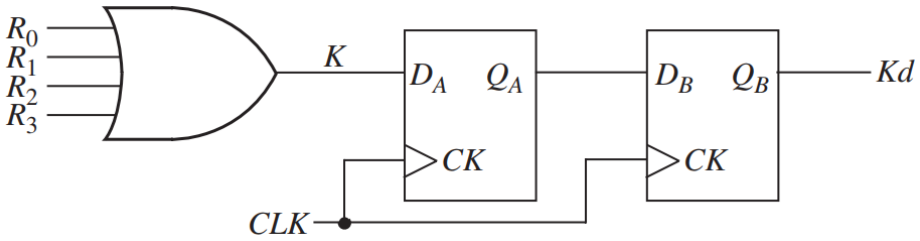
For example ...

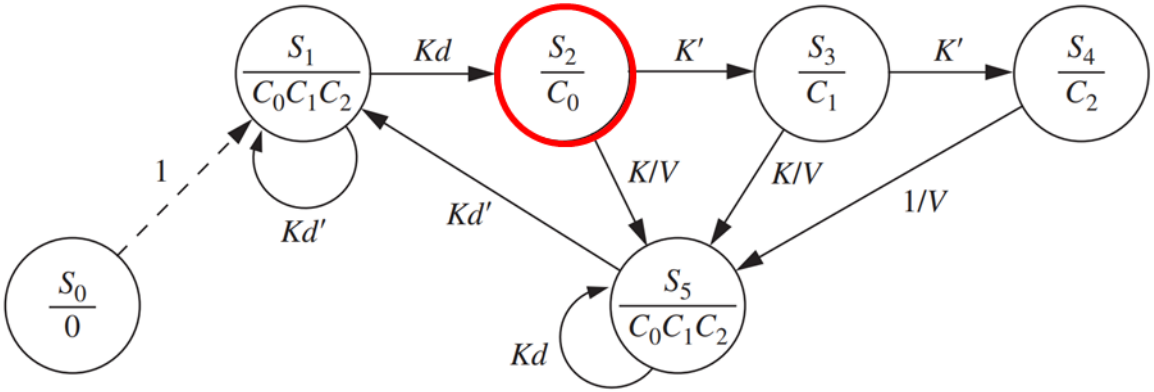
4.11.4 Controller

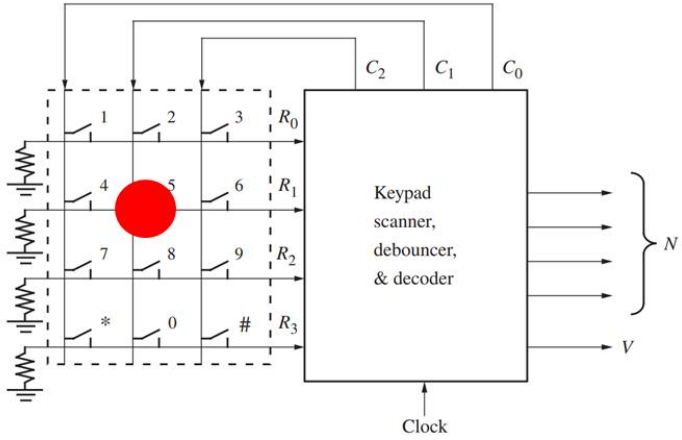


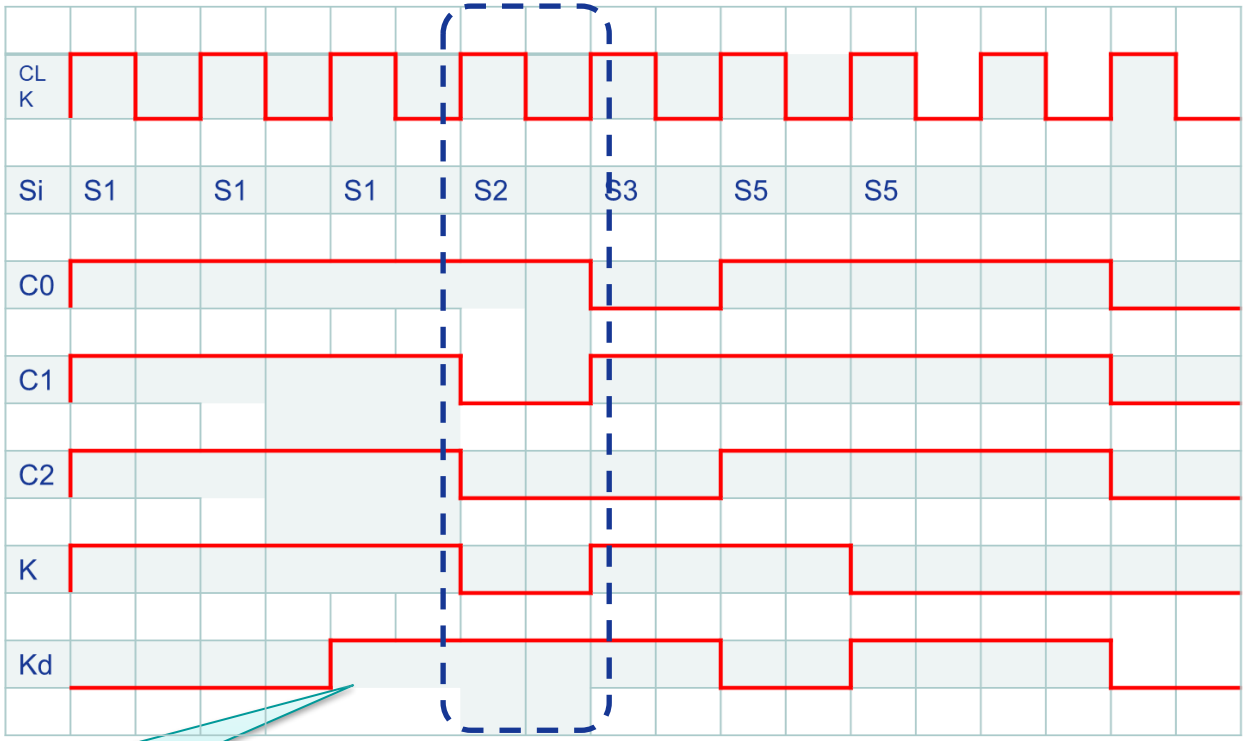
Example: assume 5 is pressed









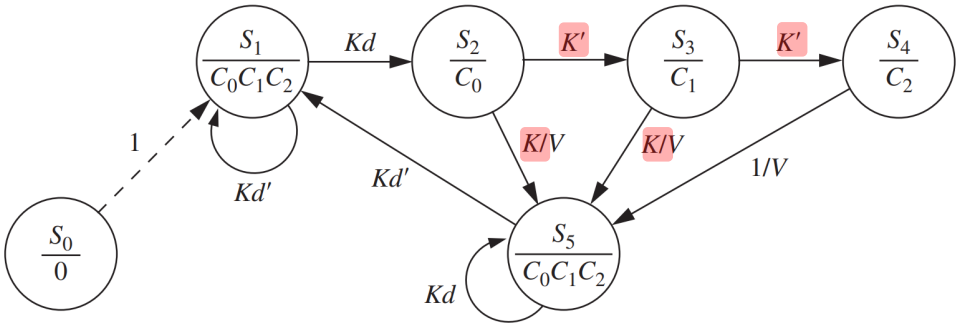


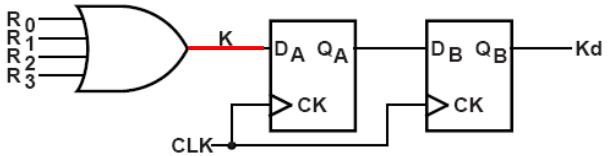
When S2, C1 = 0, K = 0

However, due to the two flip-flops, Kd will keep 1 for 2 clock periods

4.11.4 Controller

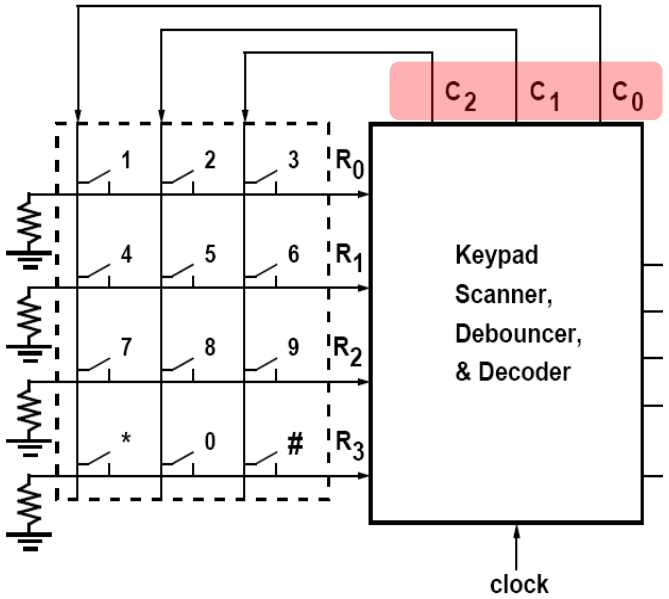
Timing problem with State graph_V1





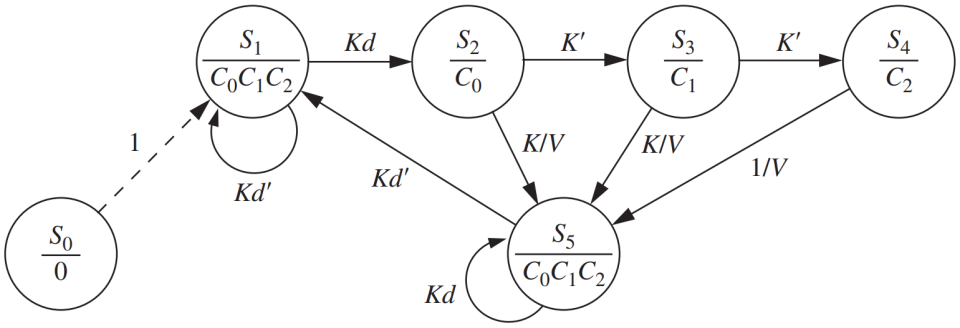
Problem 1: Is K=1 whenever a button is pressed?

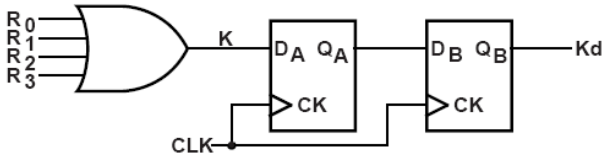
- **No**. Although *K* is true if any one of the row signals R_1, R_2, R_3, or R_4 is true
- if the column scan signals are not active, none of R_1–R_4 can be true, although the button is pressed



4.11.4 Controller

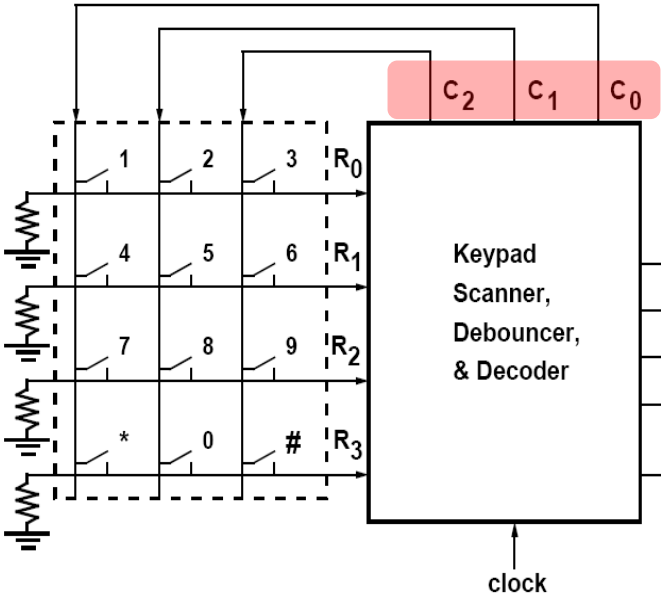
Timing problem with State graph_V1





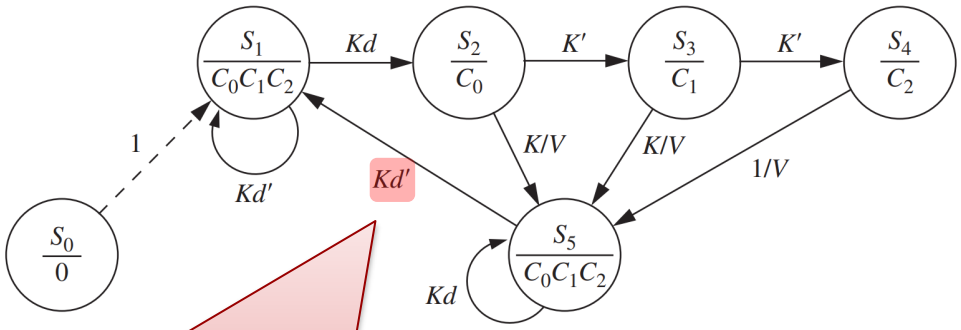
A button is pressed ⇏ K = 1

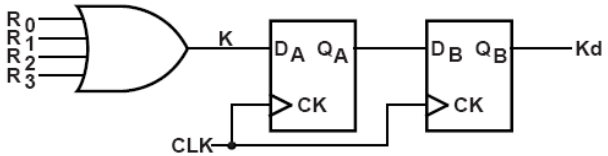
A button is pressed & corresponding Ci is active ⇒ K = 1



4.11.4 Controller

Timing problem with State graph_V1





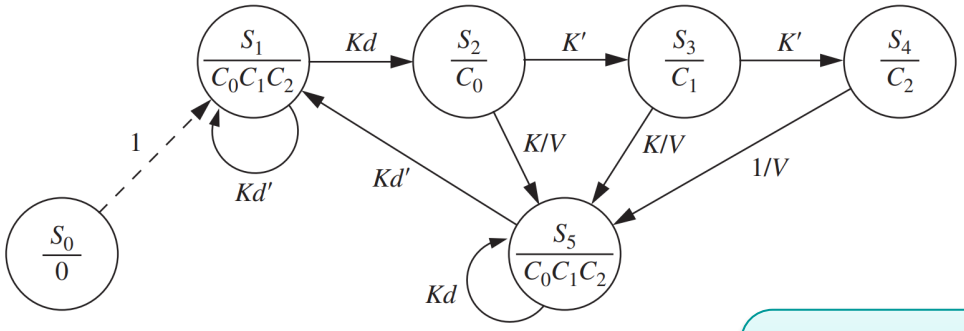
Problem 2: Can Kd=0 when a button is continuing to be pressed?

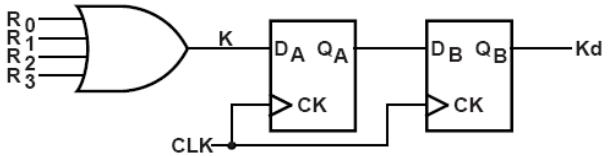
- **Yes**. Signal *Kd* is nothing but *K* delayed by two clock cycles
- *K* can go to 0 during the scan process even when the button is being pressed

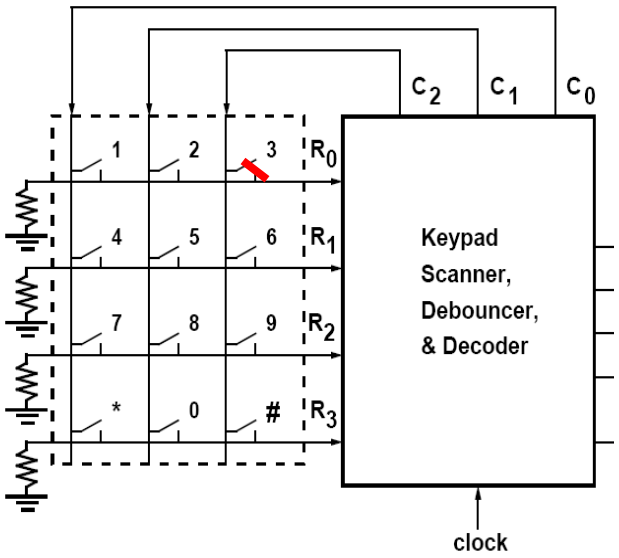
- For example...

4.11.4 Controller

Timing problem with State graph_V1







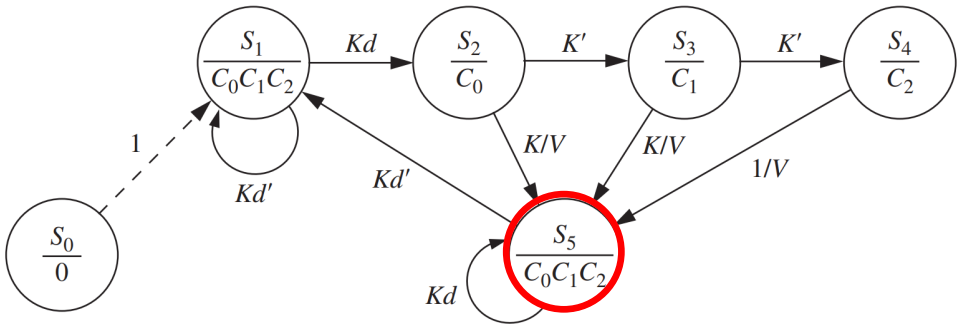
Example: consider the case when a key in the rightmost column is pressed

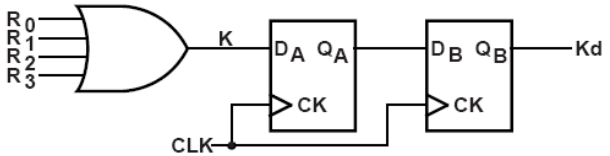
- During scan of the first two columns, *K* goes to 0
- If *K* goes to 0 at any time, *Kd* will go to zero two cycles later

Neither *K* nor *Kd* is the same as pressing the button

4.11.4 Controller

Timing problem with State graph_V1





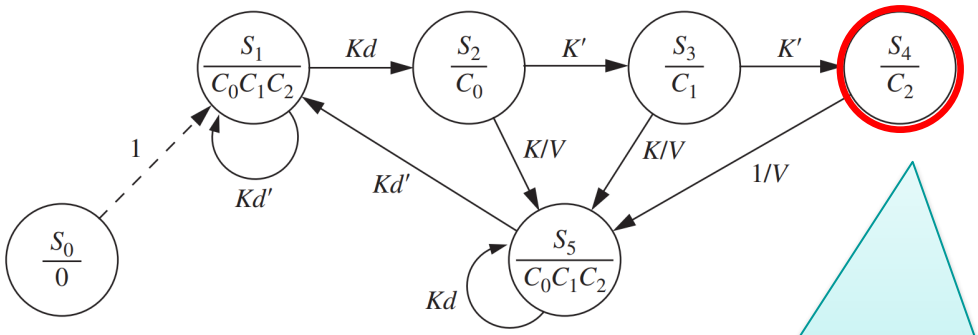
Problem 3: Can S5 go to S1 when a button is still pressed?

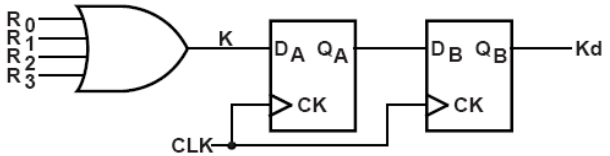
- **Yes**. *S*4-to-*S*5 transition could happen when *Kd* = 0. *Kd* might have become false while scanning *C*0 and *C*1
- Hence, it is possible that one reaches back to *S*1 when the key is still being pressed

- For example...

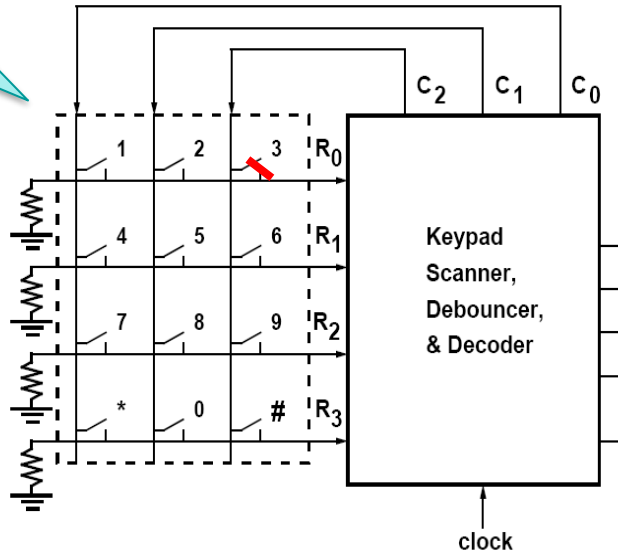
4.11.4 Controller

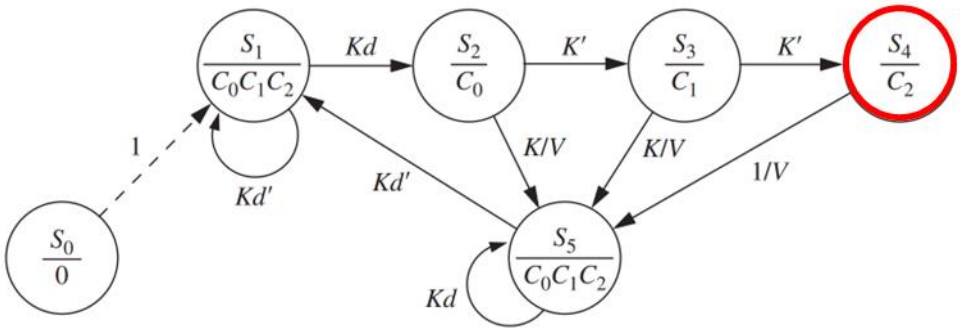
Timing problem with State graph_V1

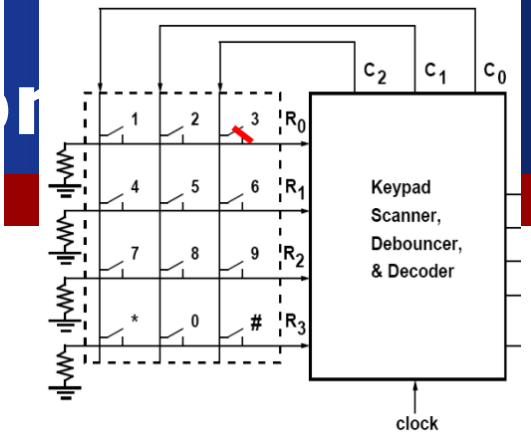


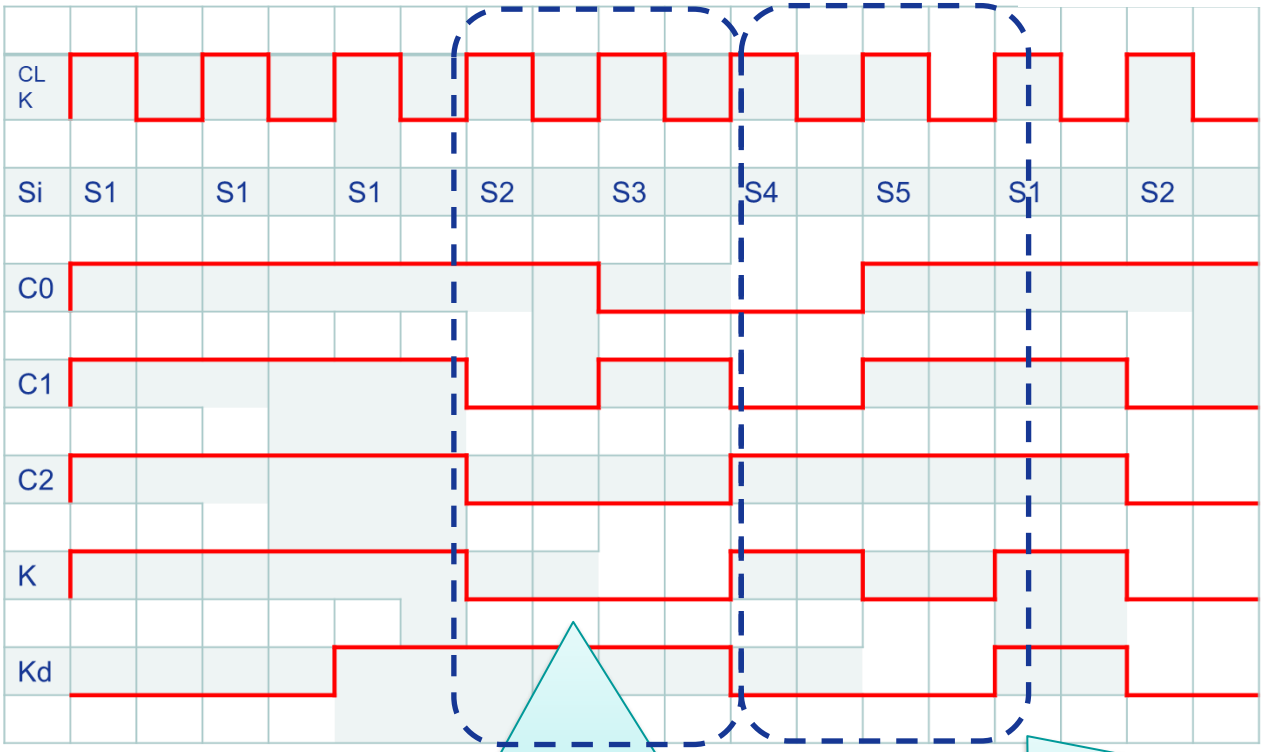


- Let us assume that a button is pressed in column C2
- This is to be detected in S4



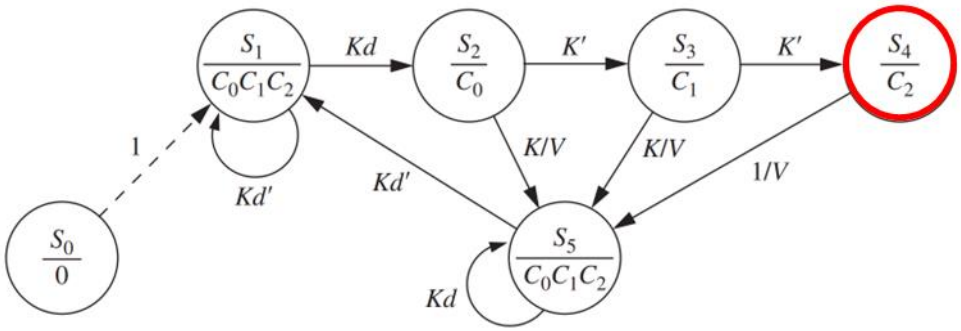


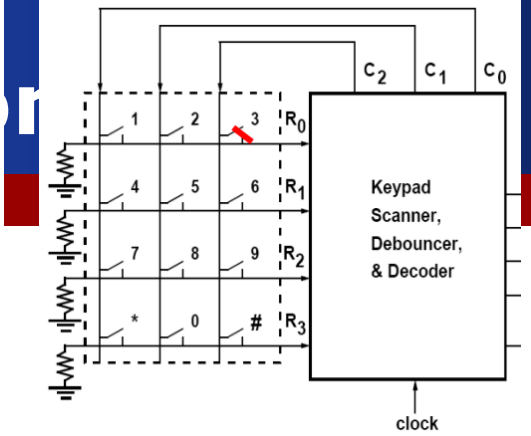


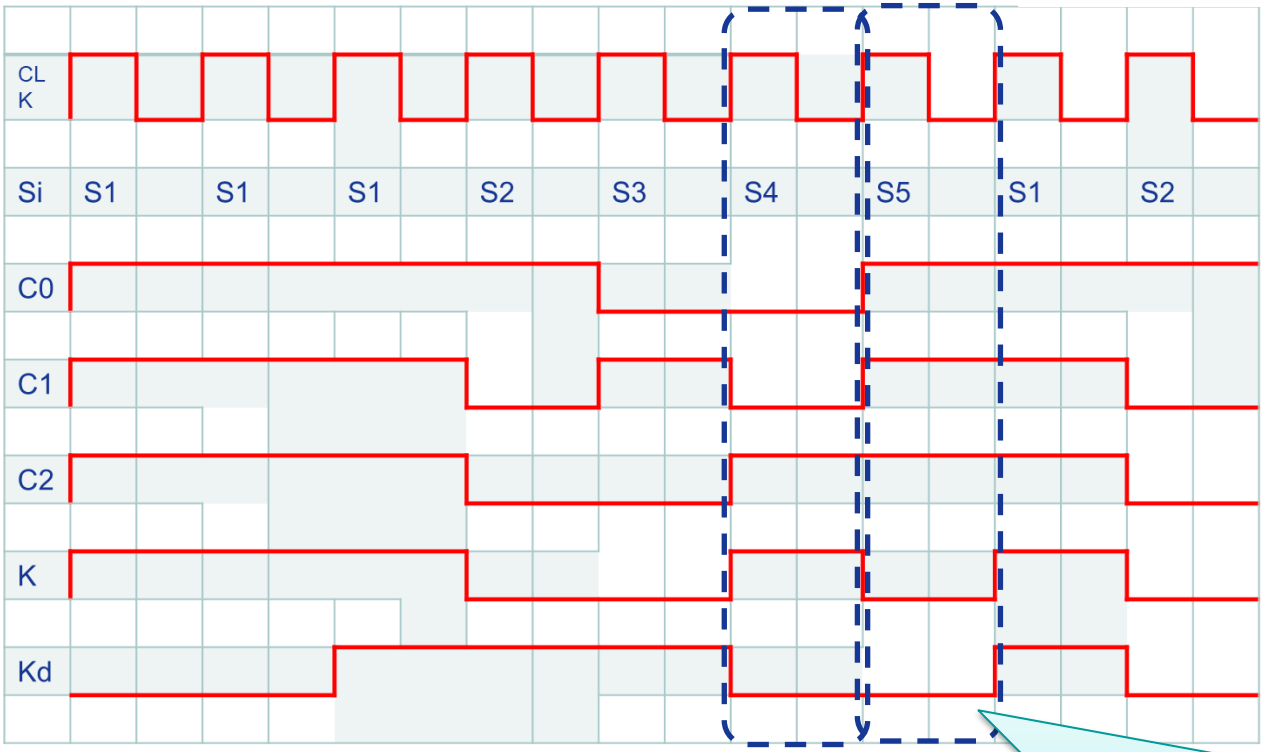


- However, during the scanning process in S2 and S3, K is 0

- Hence, two cycles later Kd will be 0 even if the button stays pressed





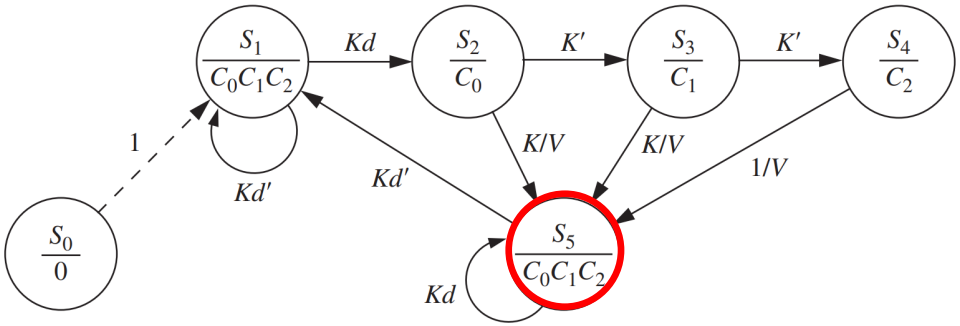


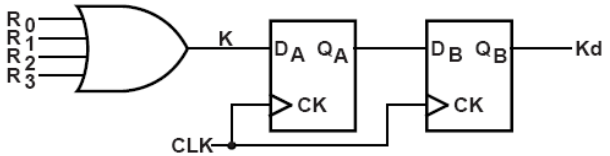
- During the scan in S4, the correct key can be found

- However, the system can reach S5 when Kd is still 0 and a malfunction can happen

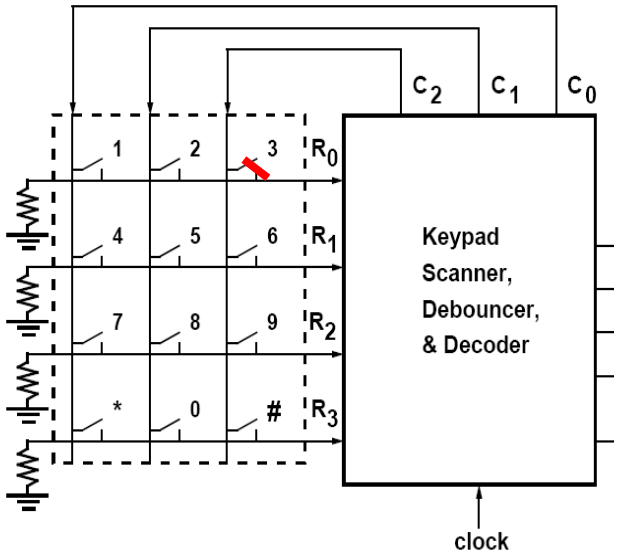
4.11.4 Controller

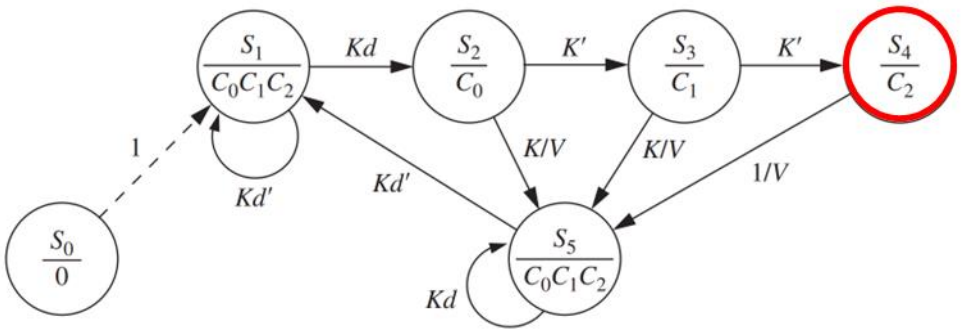
Timing problem with State graph_V1

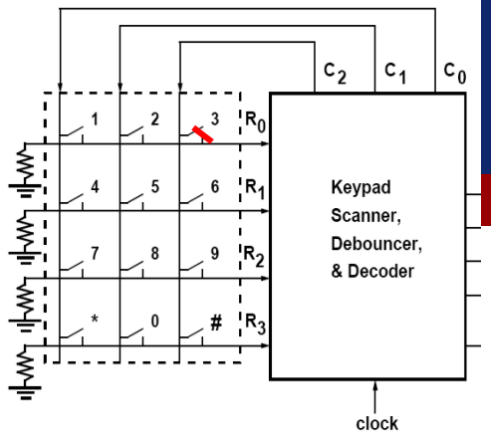


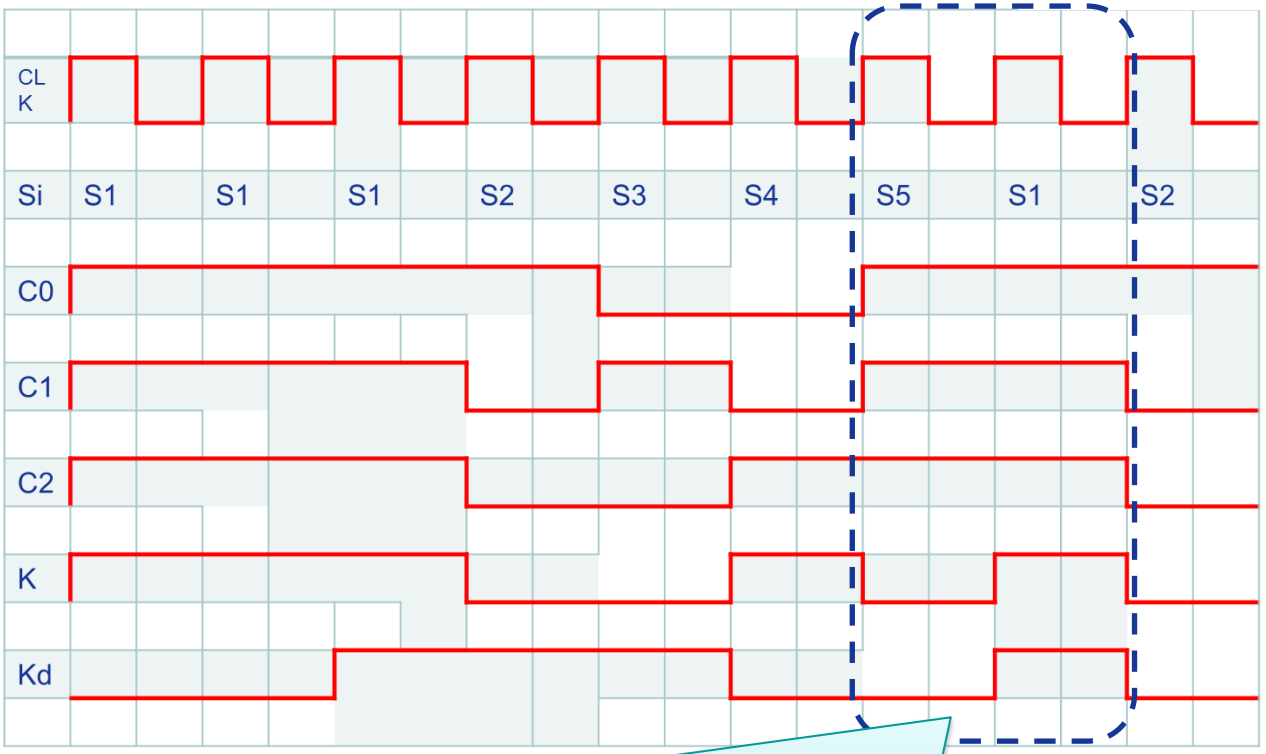


- S5 is intended to sense the release of the key
- However, Kd is not synonymous to pressing the button and Kd' does not truly indicate that the button got released





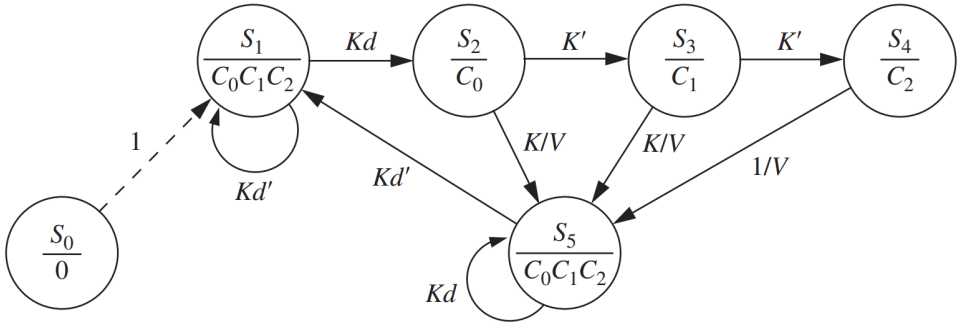




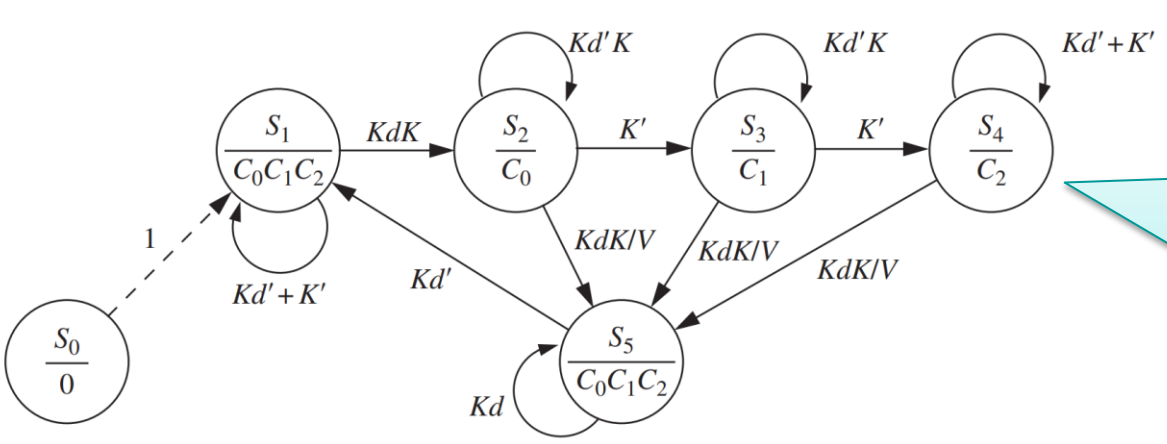
- In such a case, the same key may be read multiple times

- Since Kd' can appear when the button is still pressed, if S5 is reached when Kd' is true due to scanning activity in a previous state, the system can go from S5 to S1 without a key release

State graph for Keypad Scanner_V1

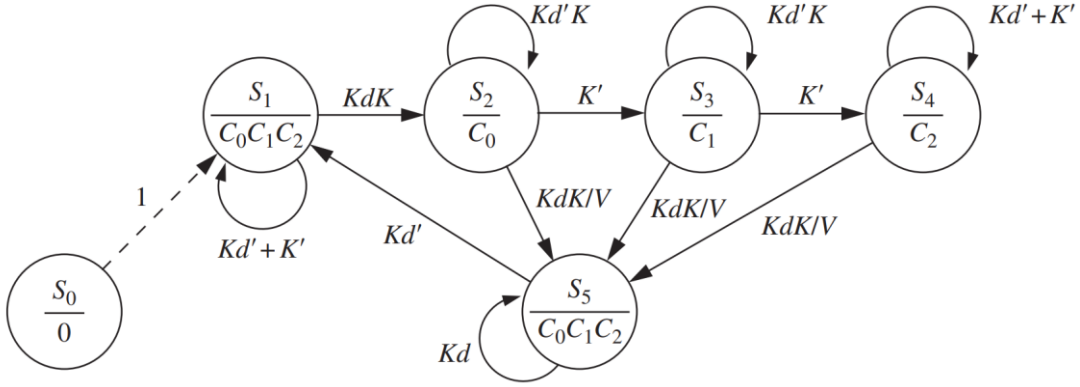


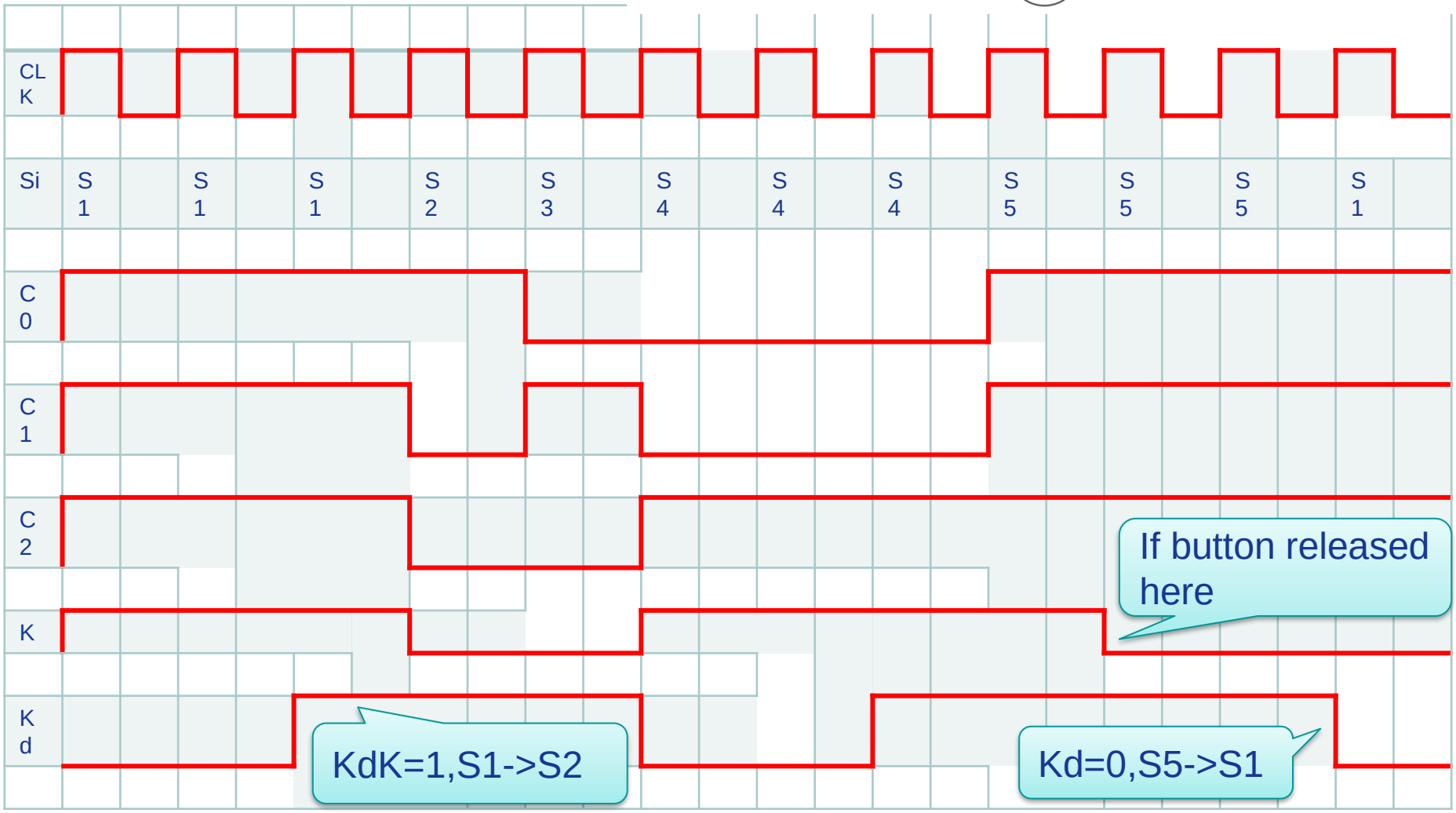
State graph for Keypad Scanner_V2



The above problems can be fixed by assuring that one can reach S5 only if Kd is true

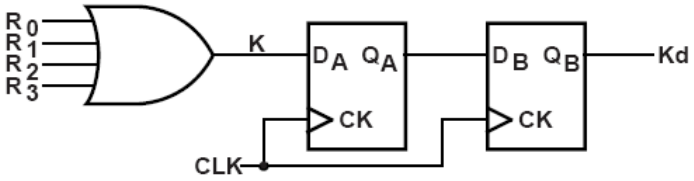
State graph_V2





4.11.5 VHDL Code

```
entity scanner is
  port(R0, R1, R2, R3, CLK: in bit;
       C0, C1, C2: inout bit;
       N0, N1, N2, N3, V: out bit);
end scanner;
```




```
N3 = R2 C0' + R3 C1'
N2 = R1 + R2 C0
N1 = R0 C0' + R2'C2 + R1'R0'C0
N0 = R1 C1 + R1'C2 + R3'R1'C1'
```

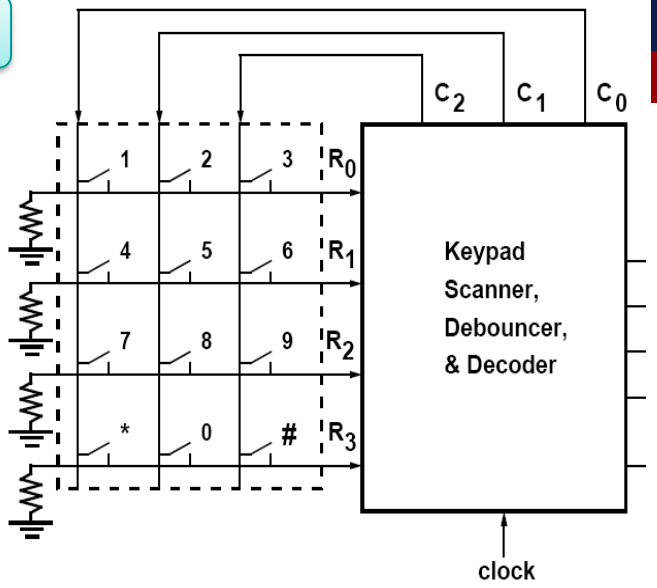
```
architecture behavior of scanner is
signal QA, K ,Kd: bit;
signal state, nextstate: integer range 0 to 5;
begin
```

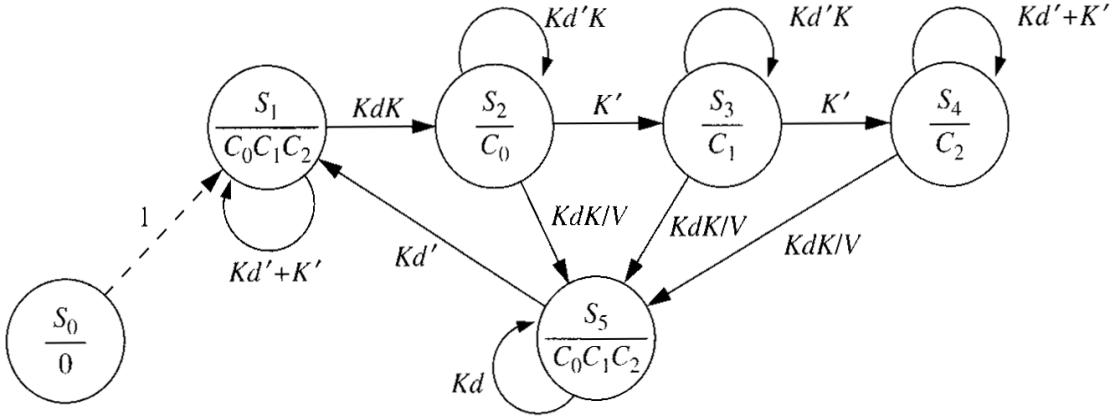
```
K <= R0 or R1 or R2 or R3;
N3 <= (R2 and not C0) or (R3 and not C1);
N2 <= R1 or (R2 and C0);
N1 <= (R0 and not C0) or (not R2 and C2) or (not R1 and not R0 and C0);
N0 <= (R1 and C1) or (not R1 and C2) or (not R3 and not R1 and not C1);
```

Decoder

```
process(state, R0, R1, R2, R3, C0, C1, C2, K, Kd, QA)
 begin
  C0 <= '0'; C1 <= '0'; C2 <= '0'; V <= '0';
  case state is
    when 0 => nextstate <= 1;
    when 1 => C0 <= '1'; C1 <= '1'; C2 <= '1';
      if (Kd and K) = '1' then nextstate <= 2;
      else nextstate <= 1;
      end if;
    when 2 => C0 <= '1';
      if (Kd and K) = '1' then V <= '1'; nextstate <= 5;
      elsif K = '0' then nextstate <= 3;
      else nextstate <= 2;
      end if;
    when 3 => C1 <= '1';
      if (Kd and K) = '1' then V <= '1'; nextstate <= 5;
      elsif K = '0' then nextstate <= 4;
      else nextstate <= 3;
      end if;
    when 4 => C2 <= '1';
      if (Kd and K) = '1' then V <= '1'; nextstate <= 5;
      else nextstate <= 4;
      end if;
    when 5 => C0 <= '1'; C1 <= '1'; C2 <= '1';
      if Kd = '0' then nextstate <= 1;
      else nextstate <= 5;
      end if;
  end case;
end process;
```

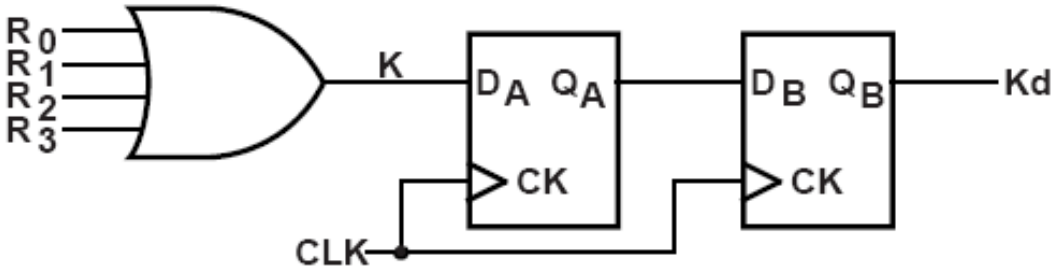
Scanner



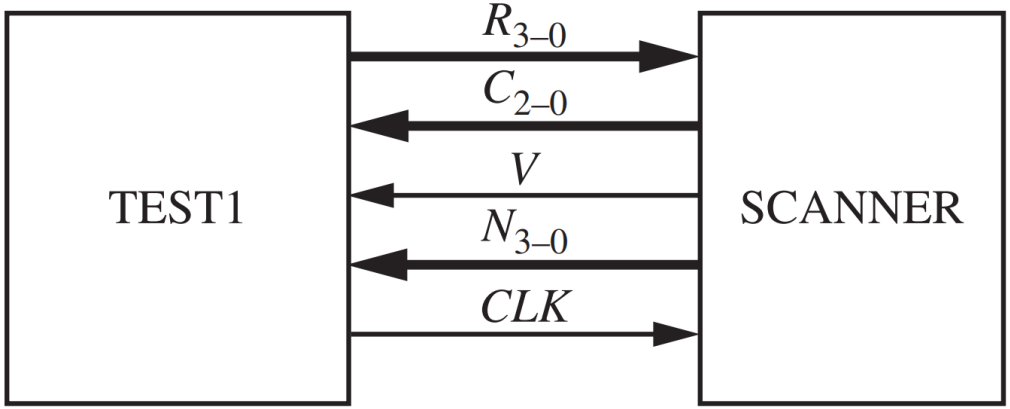


```
process(CLK)
 begin
   if CLK = '1' and CLK'EVENT then
     state  <= nextstate;
     QA <= K;
     Kd <= QA;
   end if;
 end process;
end behavior;
```

Debouncer



4.11.6 Test Bench for Keypad Scanner

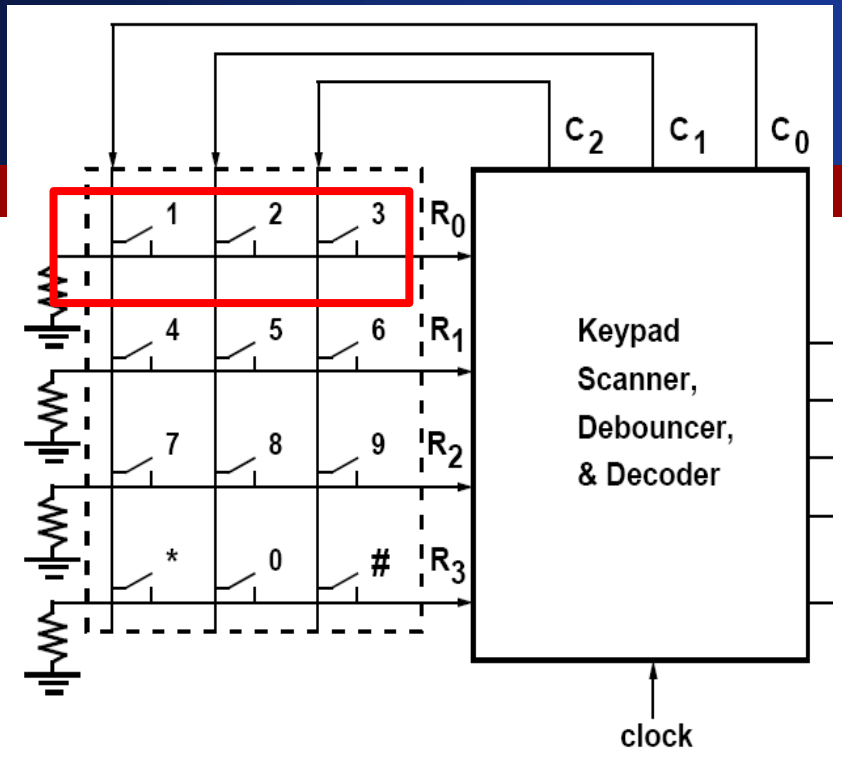


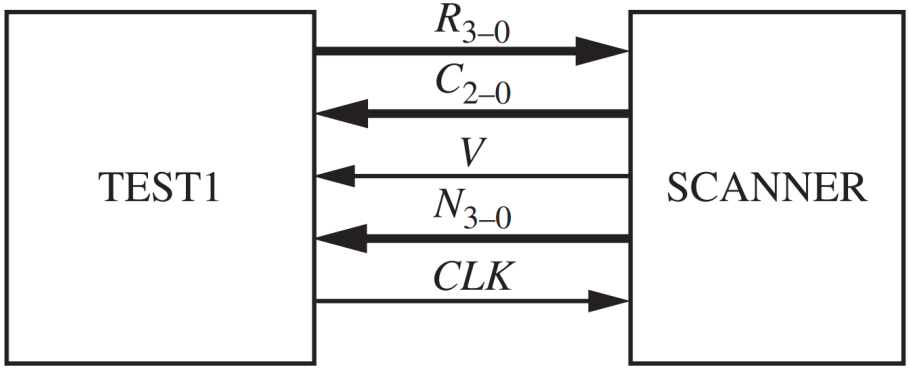

```
library IEEE;
use IEEE.numeric_bit.all;

entity scantest is
end scantest;

architecture test1 of scantest is
component scanner
  port(R0, R1, R2, R3, CLK: in bit;
       C0, C1, C2: inout bit;
       N0, N1, N2, N3, V: out bit);
end component;
type arr is array (0 to 23) of integer;          -- array of keys to test
constant KARRAY: arr := (2,5,8,0,3,6,9,11,1,4,7,10,1,2,3,4,5,6,7,8,9,10,11,0);
signal C0, C1, C2, V, CLK, R0, R1, R2, R3: bit;   -- interface signals
signal N: unsigned(3 downto 0);
signal KN: integer;                              -- key number to test
```

Scanner is treated as a component and embedded in test bench



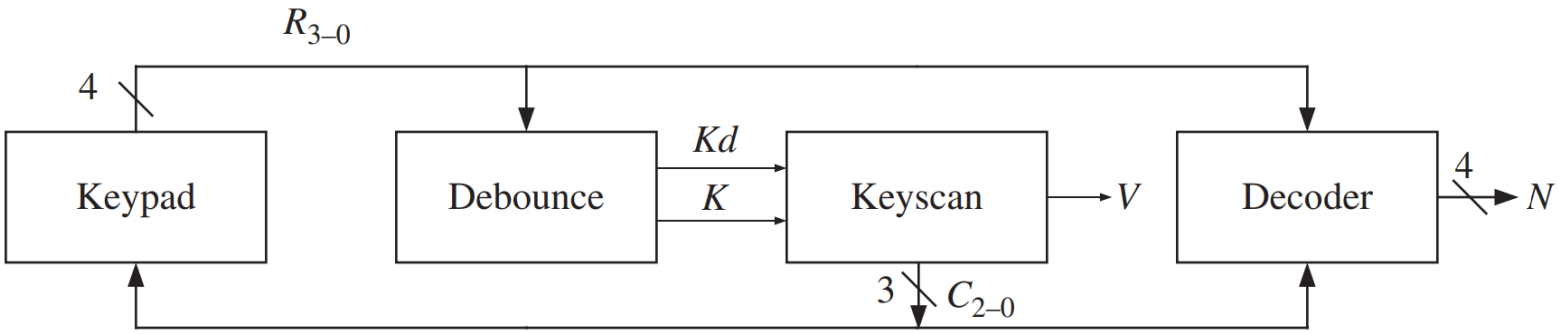


Emulate the keypad

```
begin
  CLK <= not CLK after 20 ns;                    -- generate clock signal
  R0 <= '1' when (C0='1' and KN=1) or (C1='1' and KN=2) or (C2='1' and KN=3)
        else '0';
  R1 <= '1' when (C0='1' and KN=4) or (C1='1' and KN=5) or (C2='1' and KN=6)
        else '0';
  R2 <= '1' when (C0='1' and KN=7) or (C1='1' and KN=8) or (C2='1' and KN=9)
        else '0';
  R3 <= '1' when (C0='1' and KN=10) or (C1='1' and KN=0) or (C2='1' and KN=11)
        else '0';
```

```
process                          -- this section tests scanner
  begin
    for i in 0 to 23 loop              -- test every number in key array
      KN <= KARRAY(i);                  -- simulates keypress
      wait until (V = '1' and rising_edge(CLK));
      assert (to_integer(N) = KN)       -- check if output matches
        report "Numbers don't match"
        severity error;
      KN <= 15;                        -- equivalent to no key pressed
      wait until rising_edge(CLK);      -- wait for scanner to reset
      wait until rising_edge(CLK);
      wait until rising_edge(CLK);
    end loop;
    report "Test Complete.";
  end process;
  scanner1: scanner port map(R0,R1,R2,R3,CLK,C0,C1,C2,N(0),N(1),N(2),N(3),V);
                                  -- connect test1 to scanner
end test1;
```

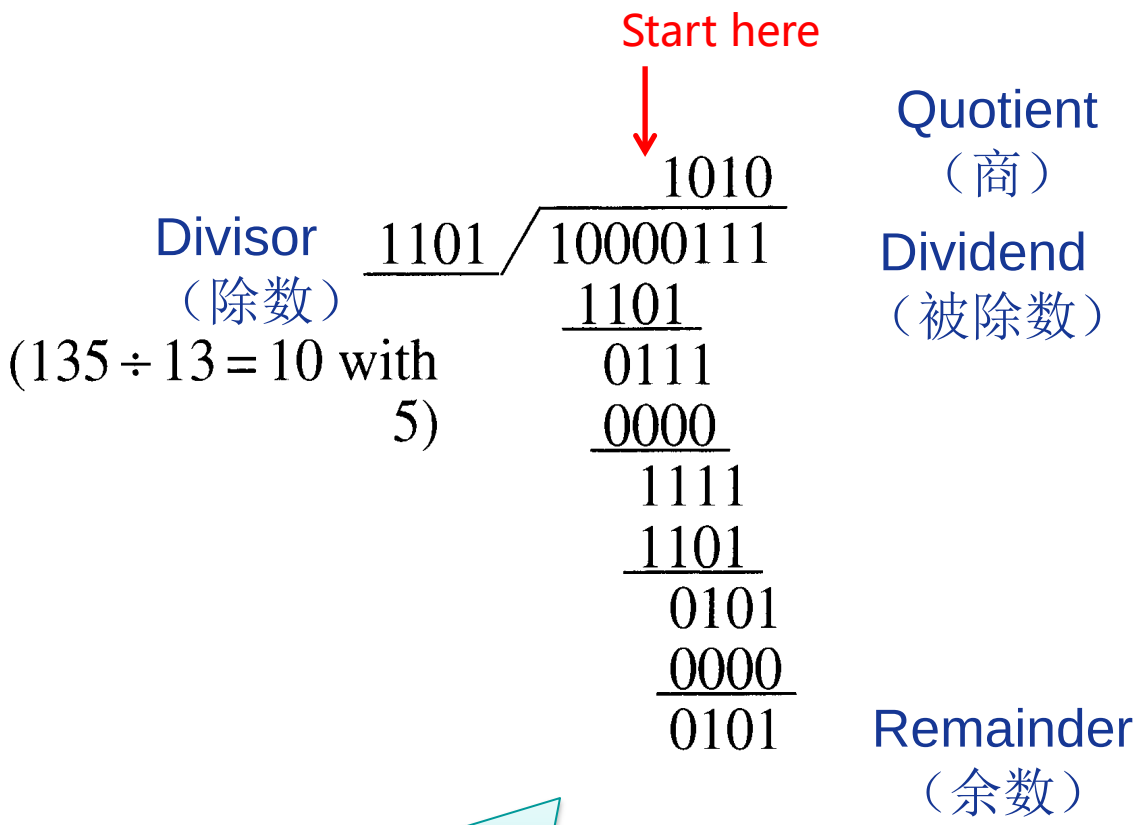
CLK = '1' and CLK'EVENT



Chapter 4 Design Examples

	Contents
1	BCD to Seven-Segment Display Decoder
2	A BCD Adder
3	32-Bit Adders
4	Traffic Light Controller
5	State Graphs for Control Circuits
6	Scoreboard and Controller
7	Synchronization and Debouncing
8	Add-and-Shift Multiplier
9	Array Multiplier
10	A Signed Integer/Fraction Multiplier
11	Keypad Scanner
12	**Binary Dividers**

4.12.1 Unsigned Divider(无符号数除法器)



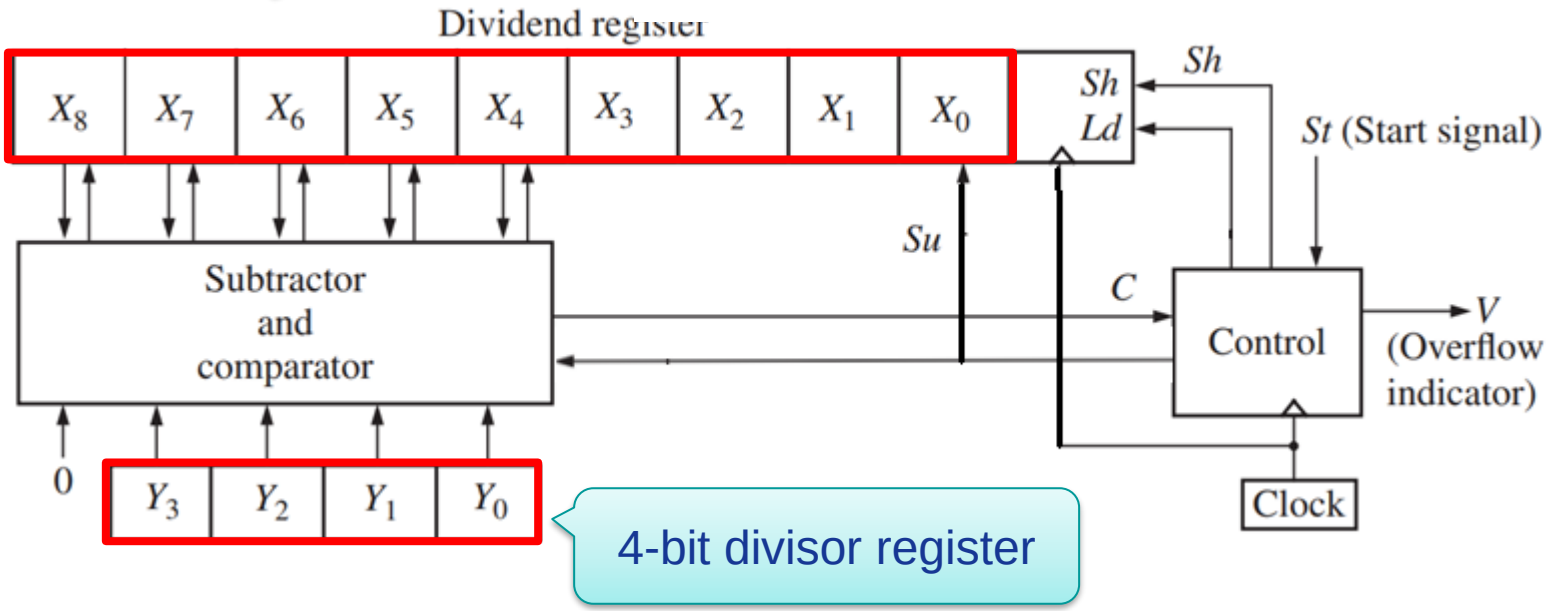
Division can be carried out by a series of **subtract** & **shift** operations

4.12.1 Unsigned Dividers

$$
\begin{array}{r|l}
1101 & 10000111 \\
 & \underline{1101} \\
 & 0111 \\
 & \underline{0000} \\
 & 1111 \\
 & \underline{1101} \\
 & 0101 \\
 & \underline{0000} \\
 & 0101
\end{array}
\qquad 1010
$$

$(135 \div 13 = 10 \text{ with } 5)$

Dividend register (9 bits?)



4-bit divisor register

? Quotient and remainder register

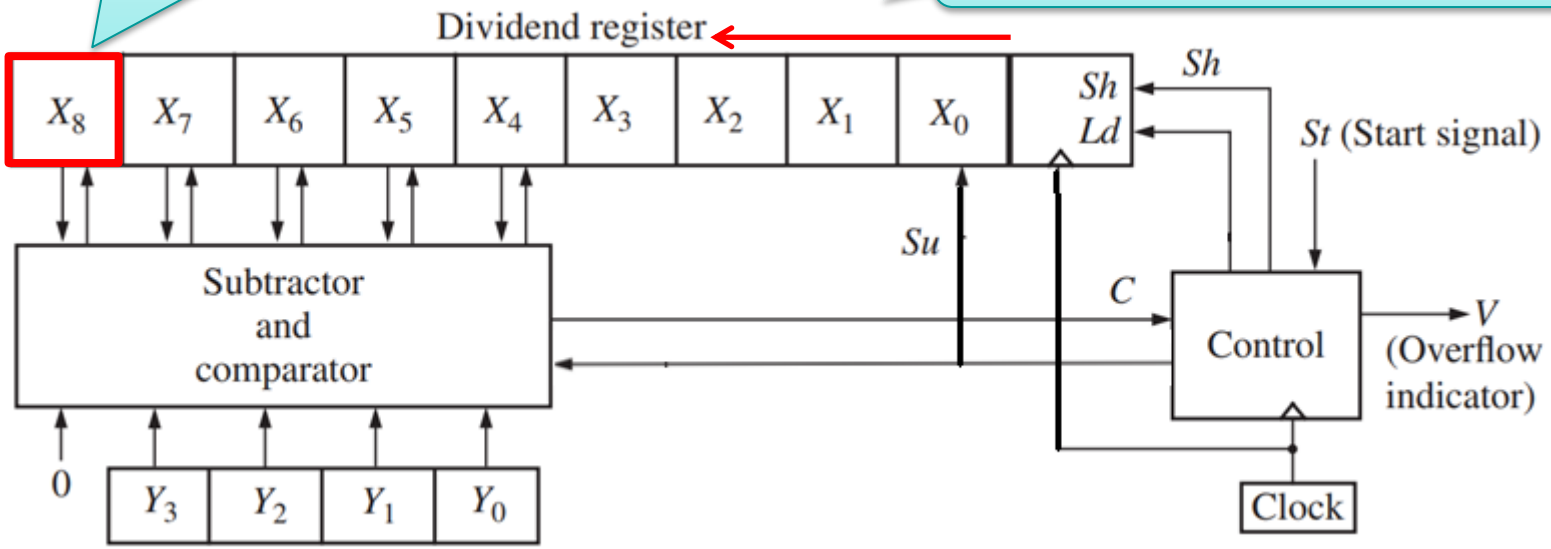
4.12.1 Unsigned Dividers

```
        1010
1101 / 10000111
       1101 →
       0111
       0000
        1111
        1101 →
        0101
        0000
        0101
```

Divisor is shifted right →

X8 is used to store the bit when the dividend is shifted left

Instead, we shift the dividend left ←

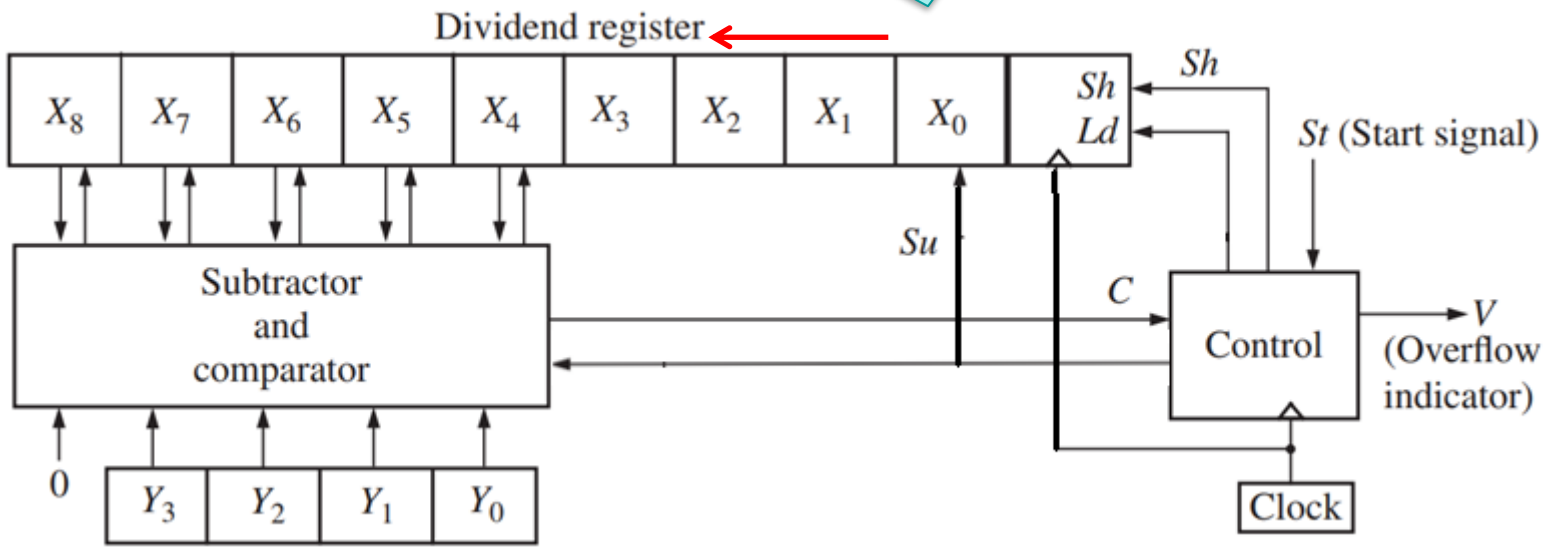


4.12.1 Unsigned Dividers

(135 ÷ 13 = 10 with 5)

```
          1010
1101 ) 10000111
       1101
       0111
       0000
        1111
        1101
         0101
         0000
         0101
```

Quotient is stored bit-by-bit into the right end of X



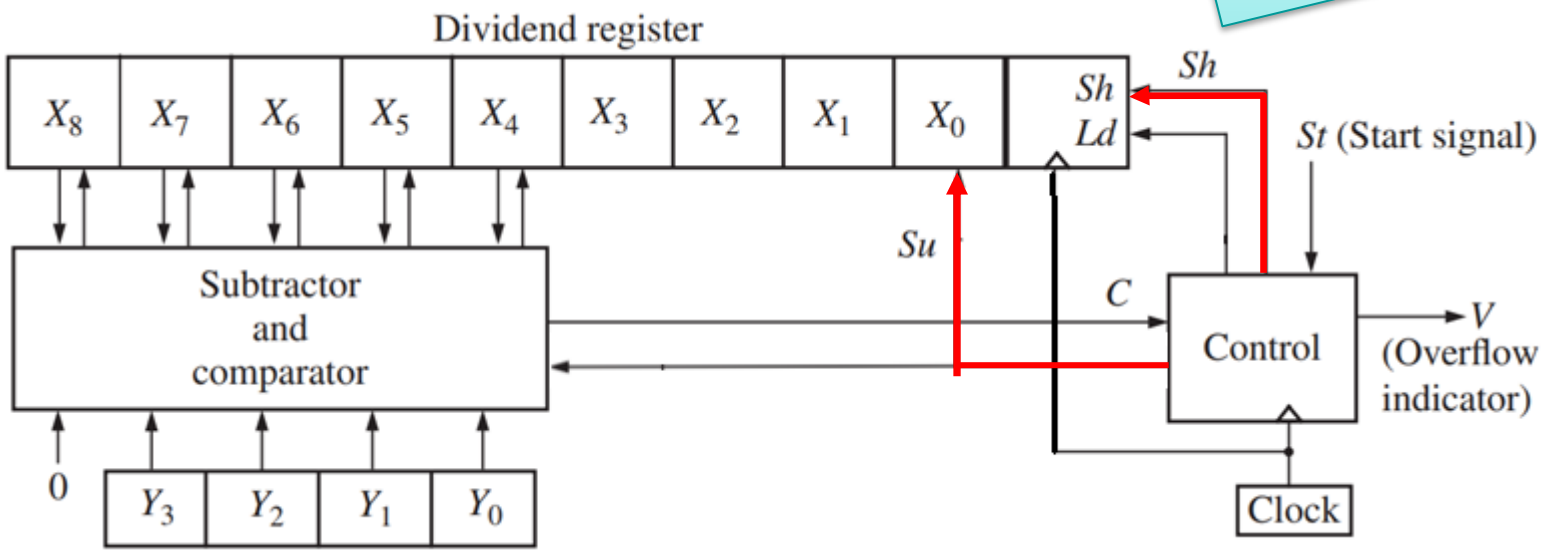
4.12.1 Unsigned Dividers

$$
\begin{array}{r}
1010 \\
1101 \overline{)\,10000111} \\
\underline{1101} \\
0111
\end{array}
$$

$(135 \div 13 = 10 \text{ with } 7)$

Cotroller outputs:

- Sh: shift dividend one bit to the left
- Su: subtract divisor Y(3:0) from X(8:4) and set quotient bit X(0) to 1



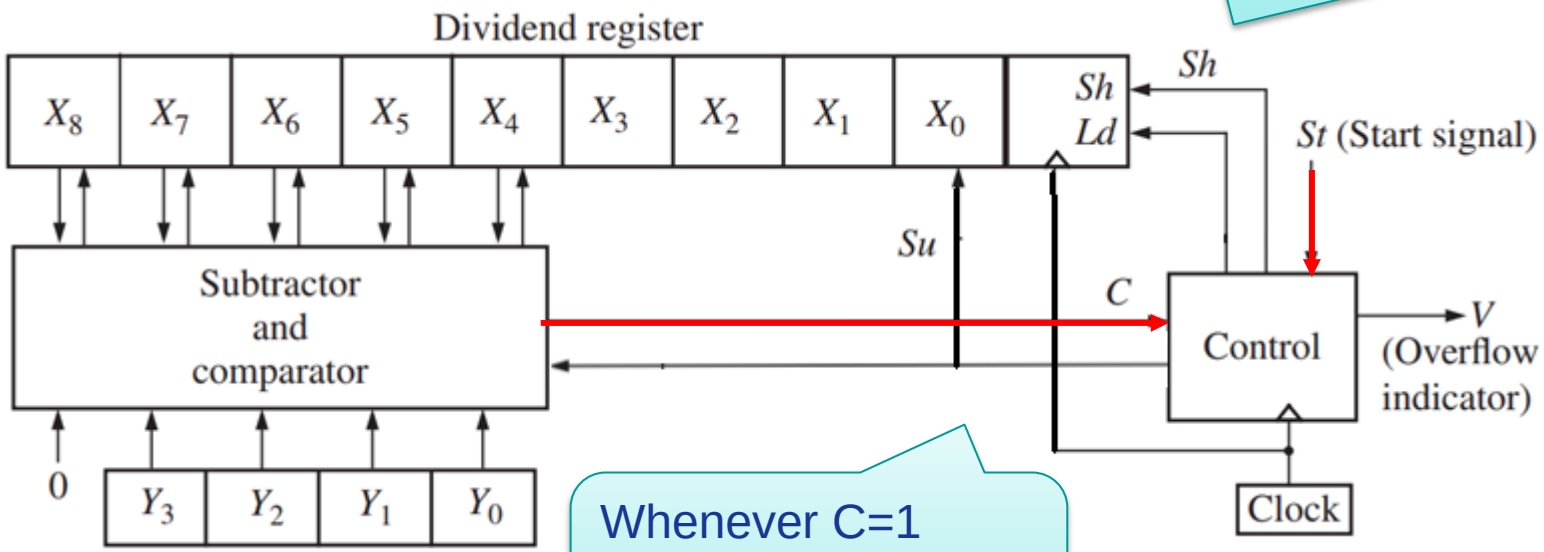
4.12.1 Unsigned Dividers

$$
\begin{array}{r|l}
 & 1010 \\
1101 & 10000111 \\
 & 1101 \\
 & 0111
\end{array}
$$

$(135 \div 13 = 10 \text{ with } 5)$

Controller inputs:

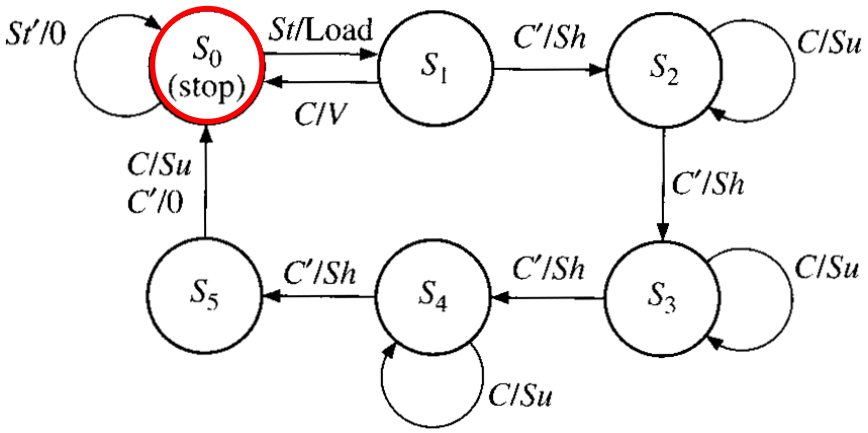
- St: Start signal
- C: C=0 when divisor > X(8:4); otherwise, C=1



Whenever C=1

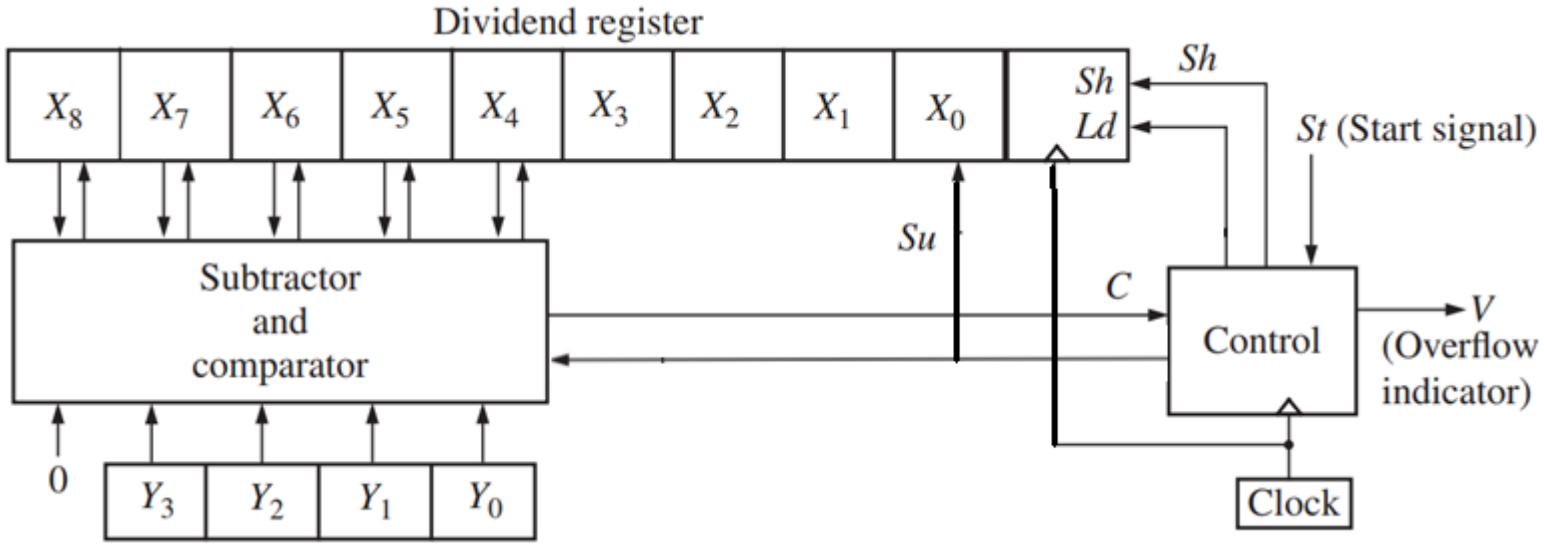
- Su is generated
- X(0) is set to 1

4.12.1 Unsigned Dividers

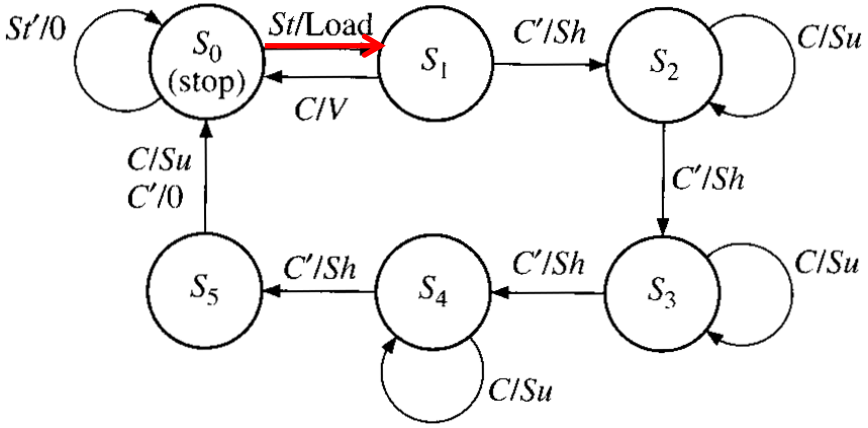


(135 ÷ 13 = 10 with 5)

```
      1010
1101 ) 10000111
       1101
       0111
       0000
        1111
        1101
         0101
         0000
         0101
```

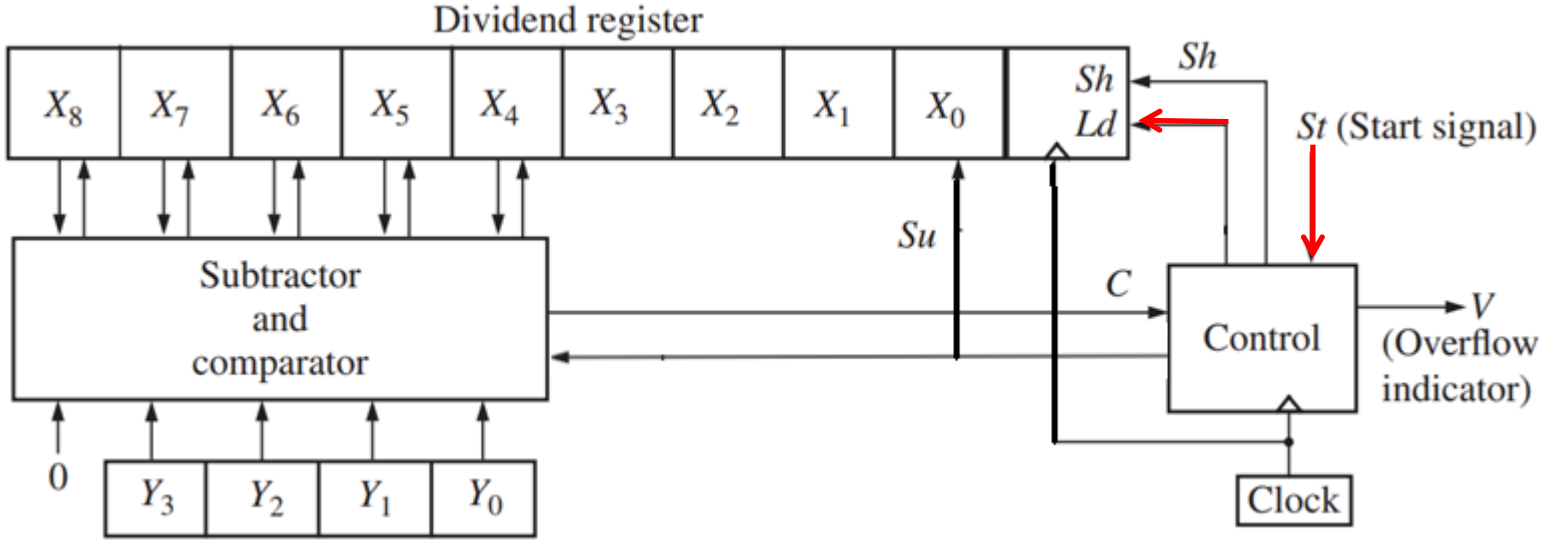


4.12.1 Unsigned Dividers

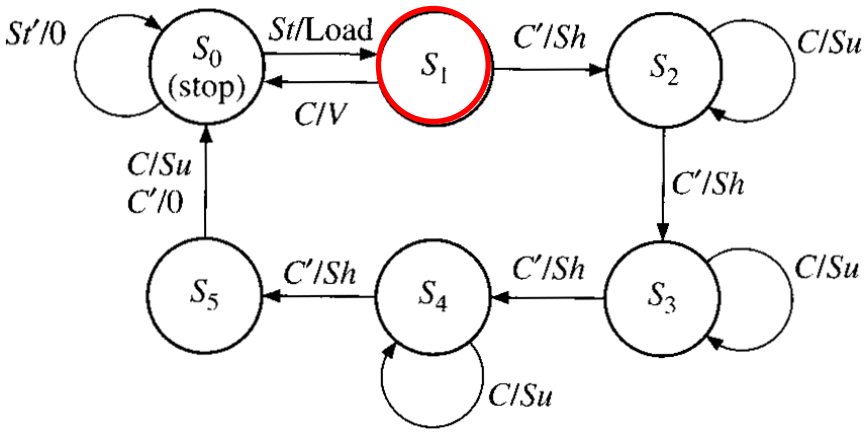


$$
\begin{array}{r}
1010 \\
1101 \overline{)\,10000111} \\
1101 \\
\hline
0111 \\
0000 \\
\hline
1111 \\
1101 \\
\hline
0101 \\
0000 \\
\hline
0101
\end{array}
$$

(135 ÷ 13 = 10 with 5)

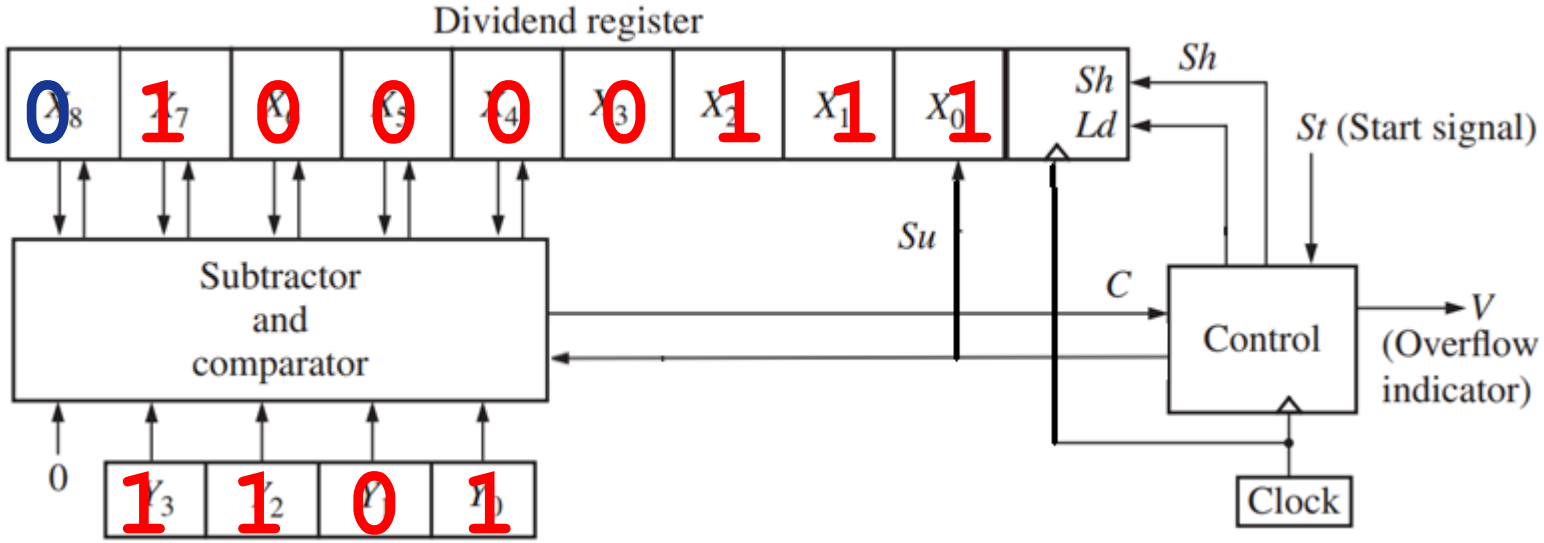


4.12.1 Unsigned Dividers

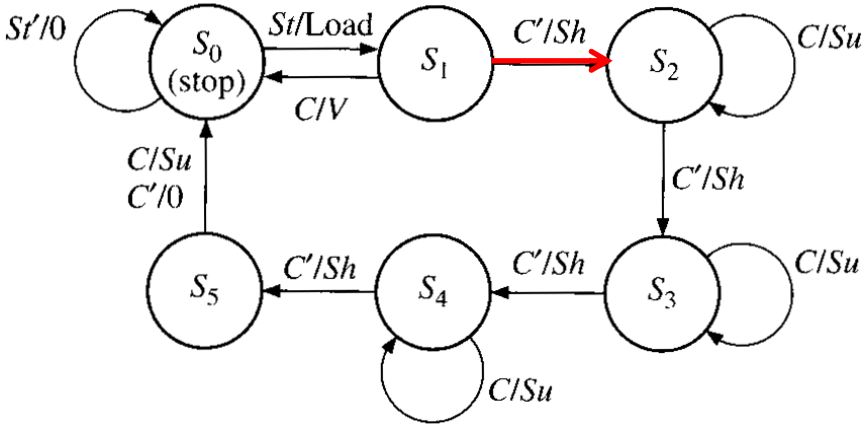


$$
\begin{array}{r}
1010 \\
1101 \overline{)\,10000111} \\
1101 \\
0111 \\
0000 \\
1111 \\
1101 \\
0101 \\
0000 \\
0101
\end{array}
$$

(135 ÷ 13 = 10 with 5)

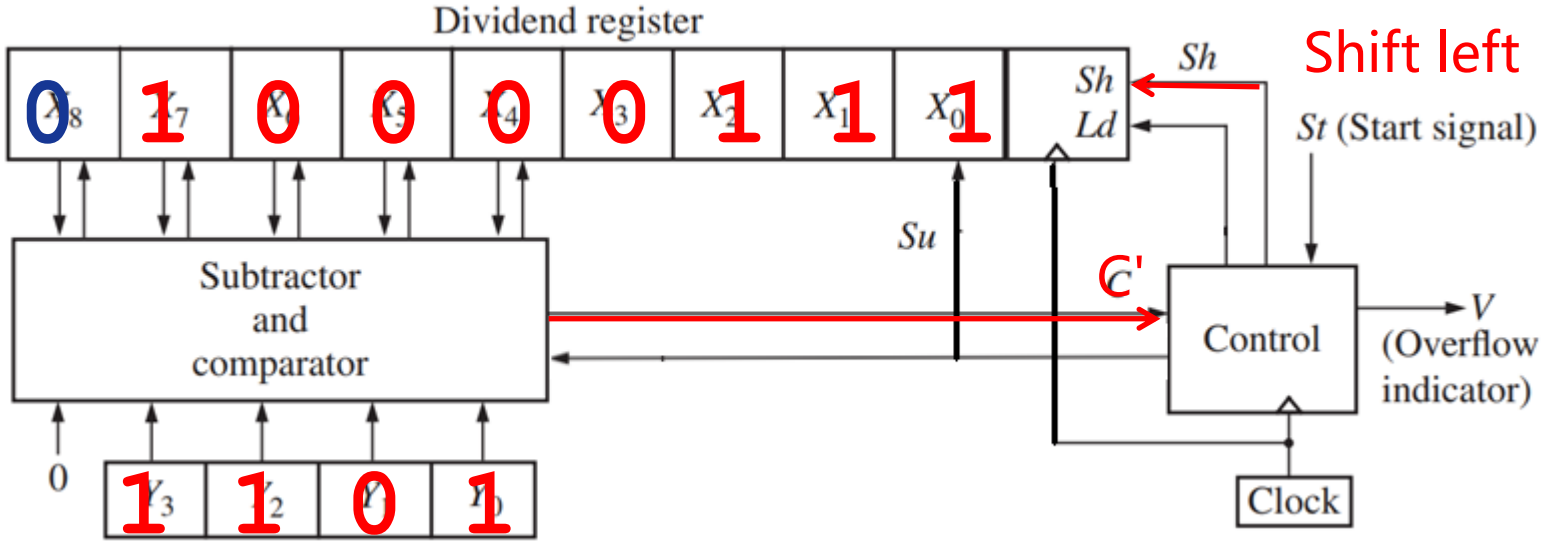


4.12.1 Unsigned Dividers

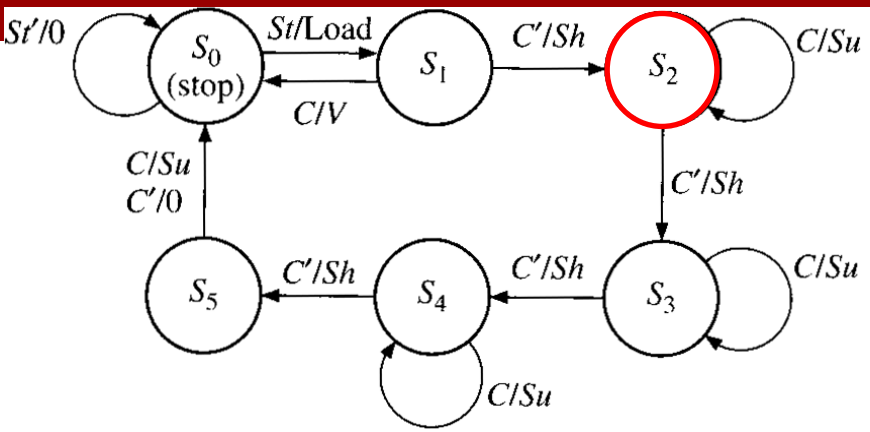


$$
\begin{array}{r}
1010 \\
1101 \overline{)\,10000111} \\
1101 \\
\hline
0111 \\
0000 \\
\hline
1111 \\
1101 \\
\hline
0101 \\
0000 \\
\hline
0101
\end{array}
$$

(135 ÷ 13 = 10 with 5)

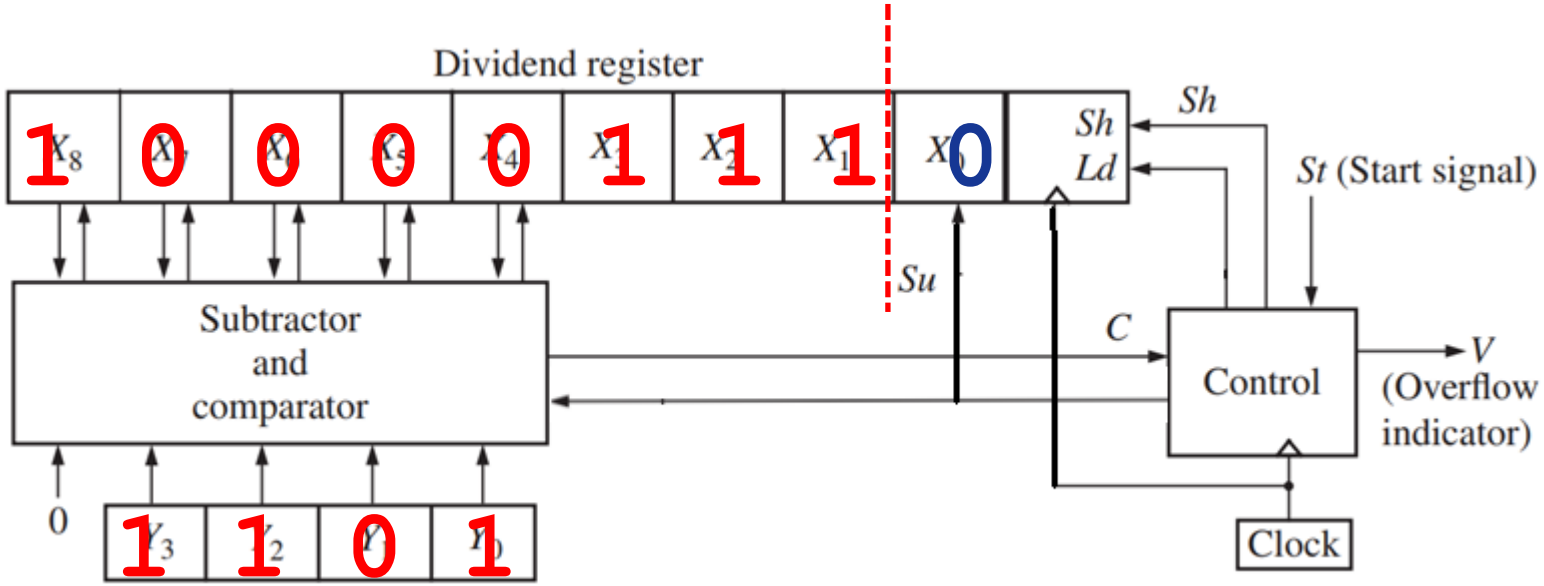


4.12.1 Unsigned Dividers

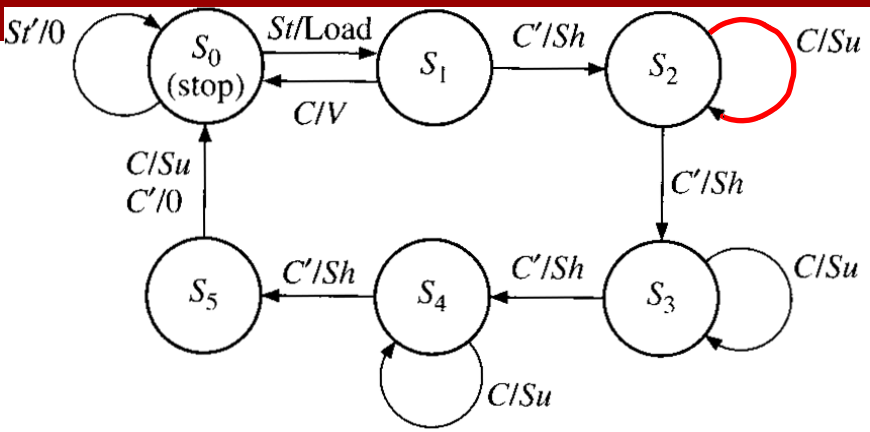


(135 ÷ 13 = 10 with 5)

$$
\begin{array}{r|l}
 & 1010 \\
\hline
1101 & 10000111 \\
 & \underline{1101} \\
 & 0111 \\
 & \underline{0000} \\
 & 1111 \\
 & \underline{1101} \\
 & 0101 \\
 & \underline{0000} \\
 & 0101
\end{array}
$$

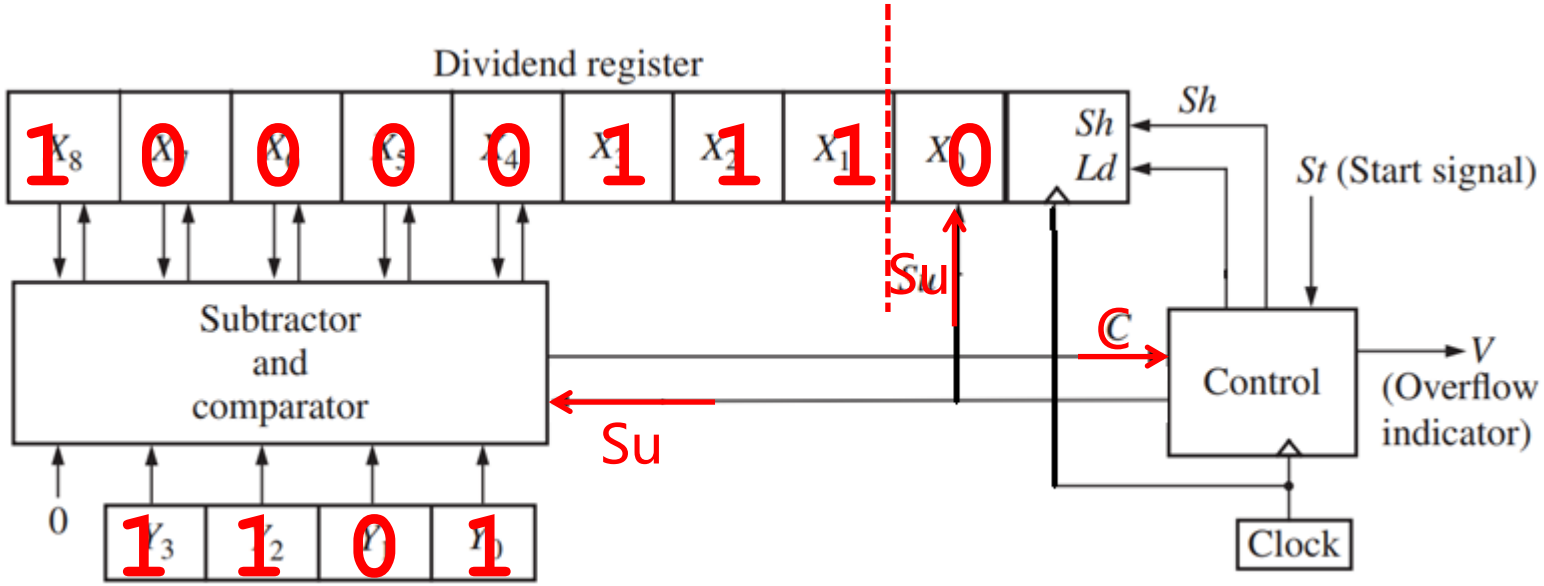


4.12.1 Unsigned Dividers

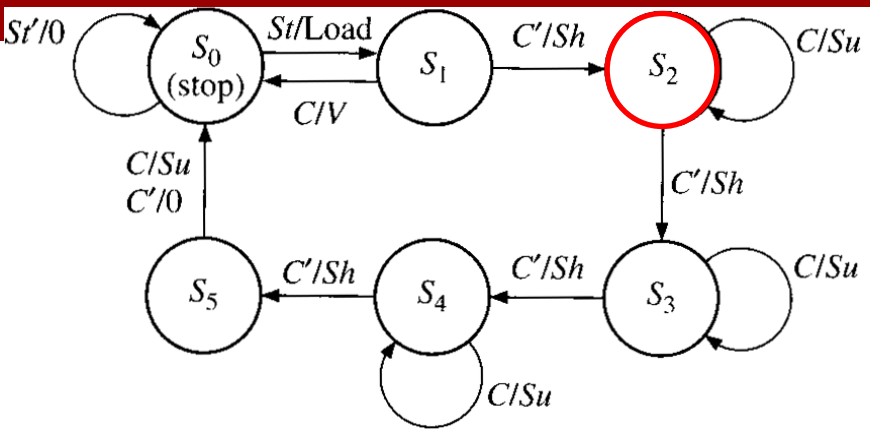


(135 ÷ 13 = 10 with 5)

$$
\begin{array}{r|l}
 & 1010 \\
\hline
1101 & 10000111 \\
 & \underline{1101} \\
 & 0111 \\
 & \underline{0000} \\
 & 1111 \\
 & \underline{1101} \\
 & 0101 \\
 & \underline{0000} \\
 & 0101
\end{array}
$$

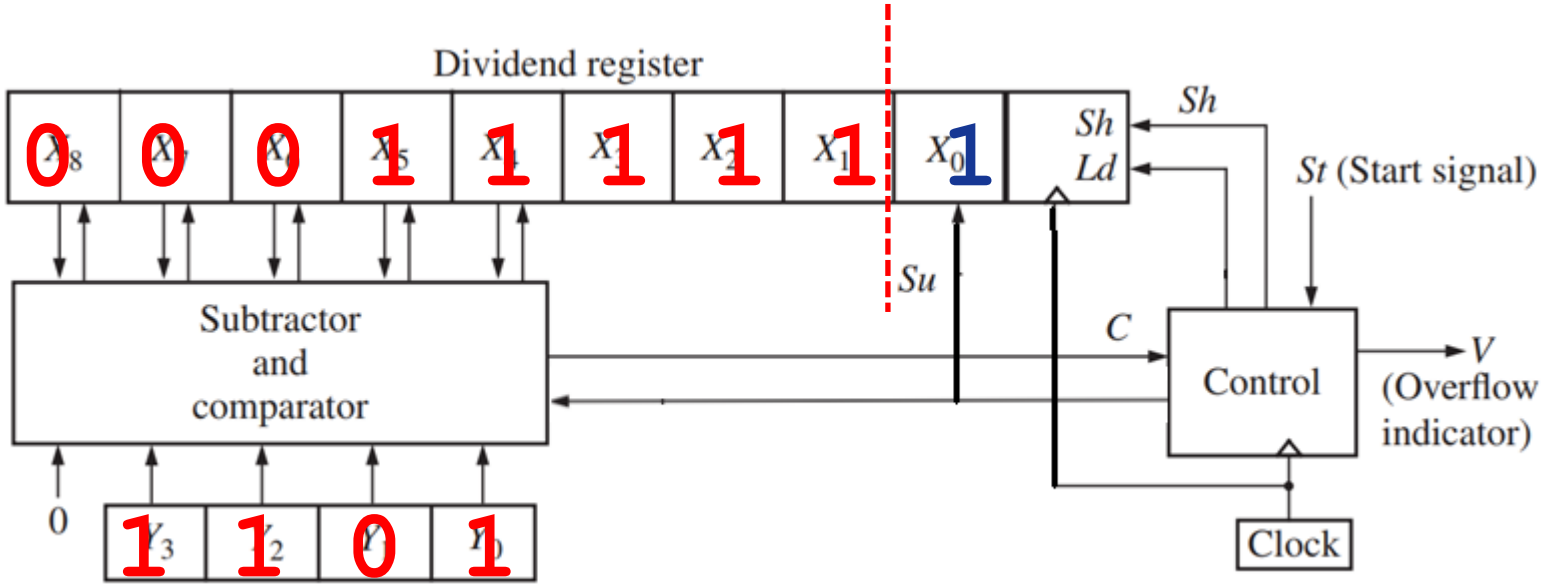


4.12.1 Unsigned Dividers

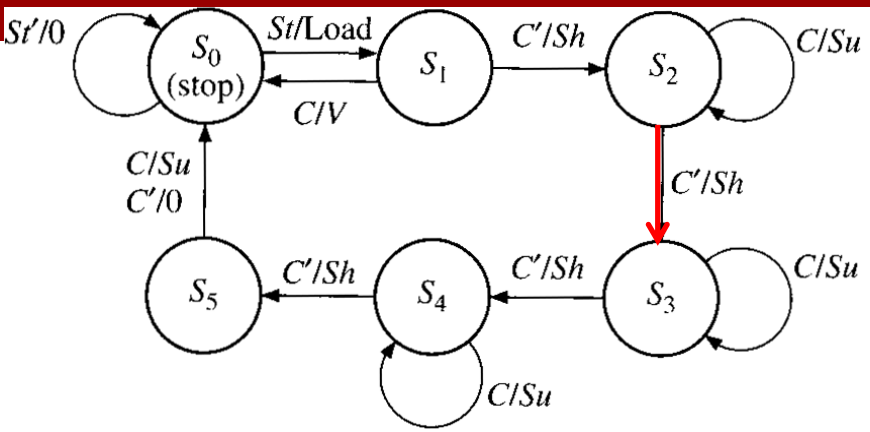


(135 ÷ 13 = 10 with 5)

$$
\begin{array}{r|l}
 & 1010 \\
\hline
1101 & 10000111 \\
 & \underline{1101} \\
 & 0111 \\
 & \underline{0000} \\
 & 1111 \\
 & \underline{1101} \\
 & 0101 \\
 & \underline{0000} \\
 & 0101
\end{array}
$$

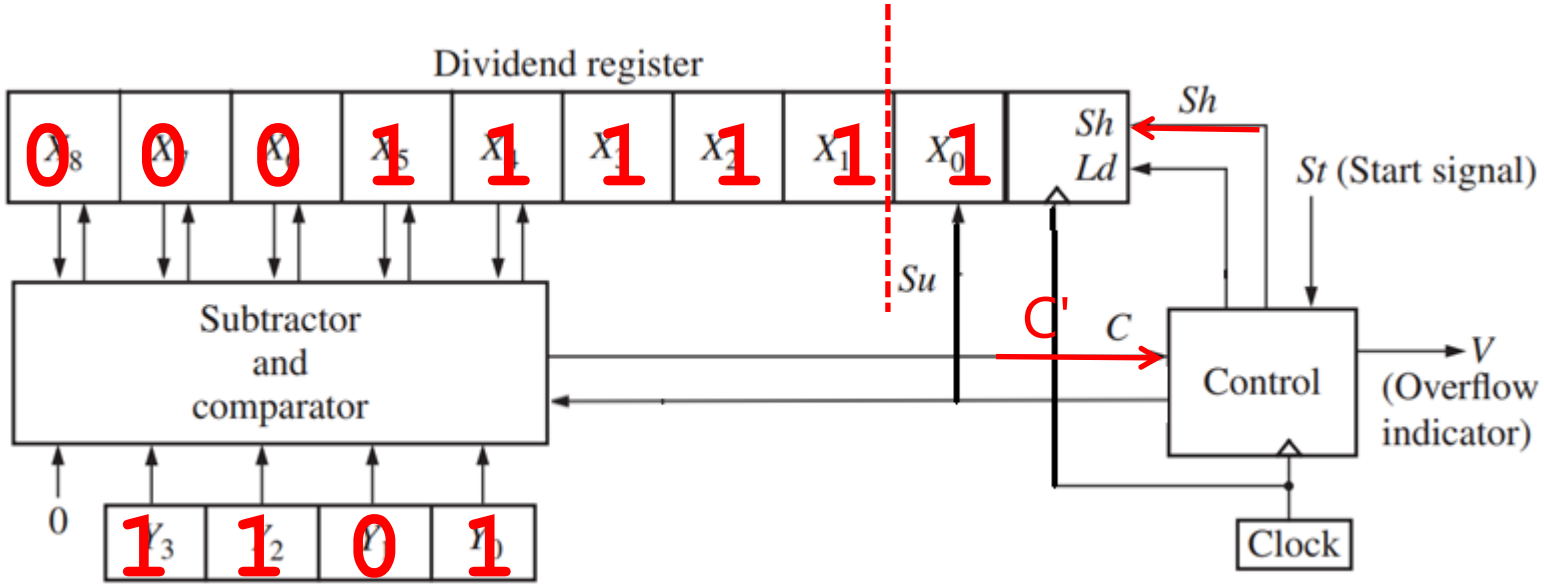


4.12.1 Unsigned Dividers

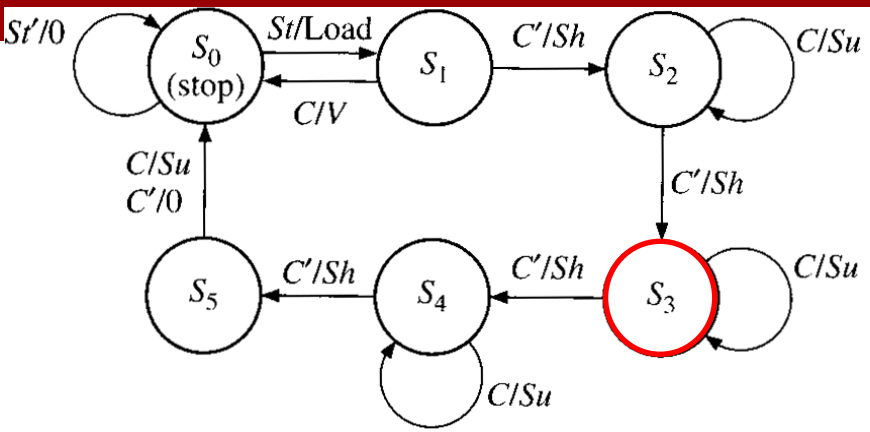


(135 ÷ 13 = 10 with 5)

$$
\begin{array}{r|l}
 & 1010 \\
\hline
1101 & 10000111 \\
 & \underline{1101} \\
 & 0111 \\
 & \underline{0000} \\
 & 1111 \\
 & \underline{1101} \\
 & 0101 \\
 & \underline{0000} \\
 & 0101
\end{array}
$$

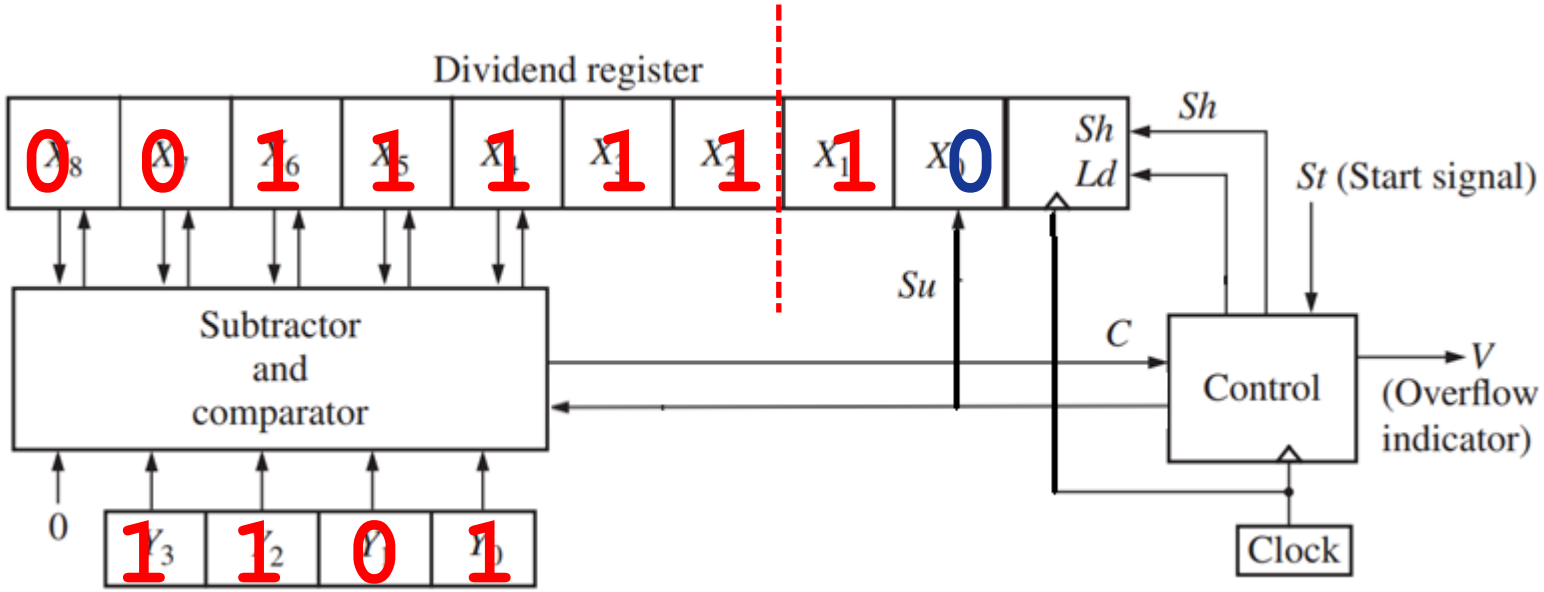


4.12.1 Unsigned Dividers

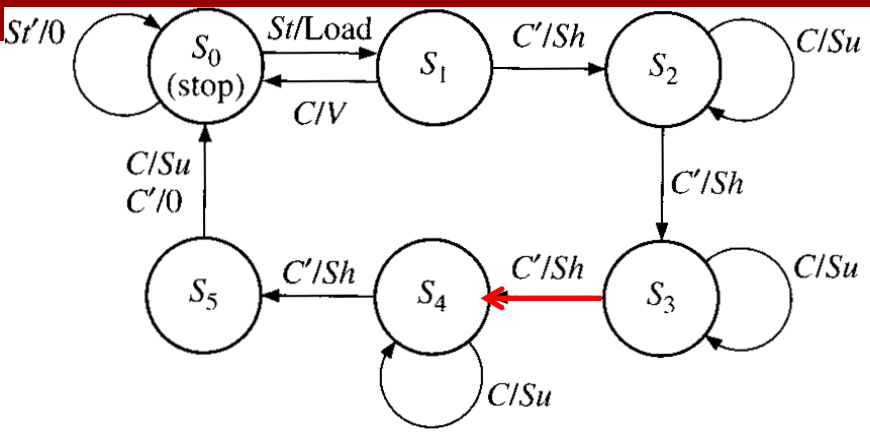


(135 ÷ 13 = 10 with 5)

```
        1010
1101 ) 10000111
       1101
       0111
       0000
        1111
        1101
         0101
         0000
         0101
```

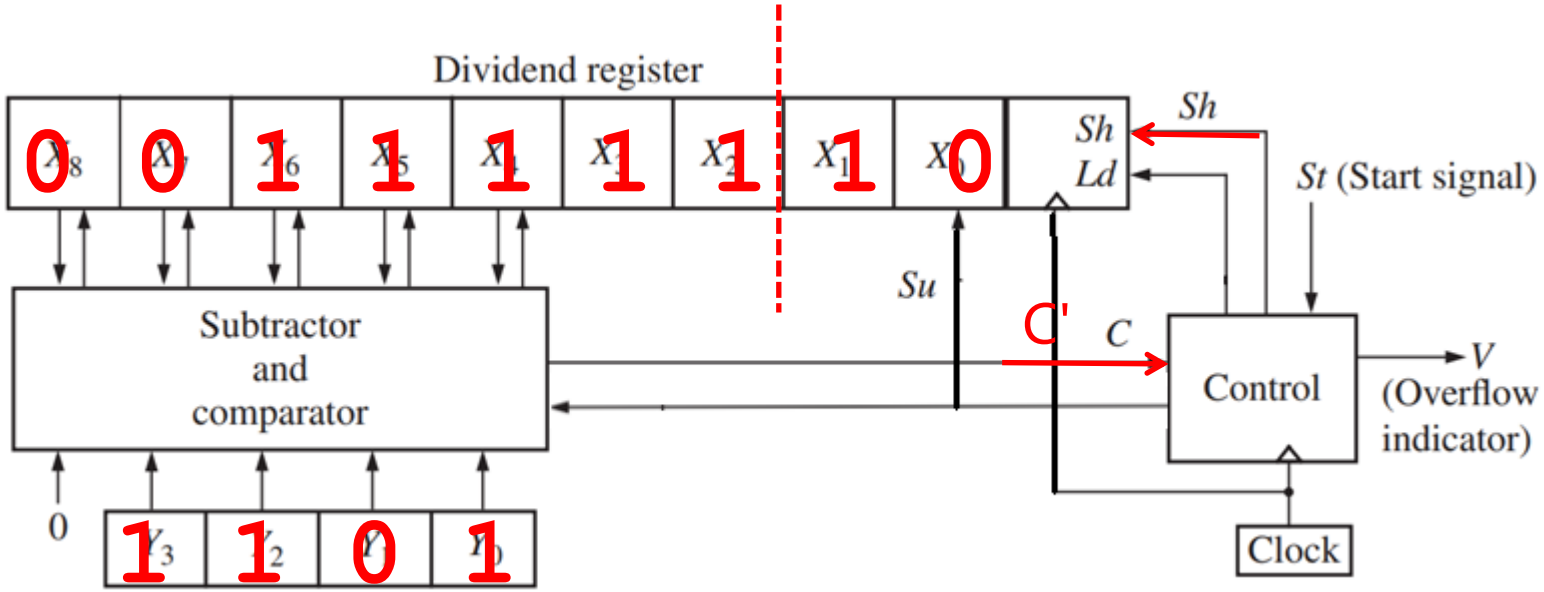


4.12.1 Unsigned Dividers

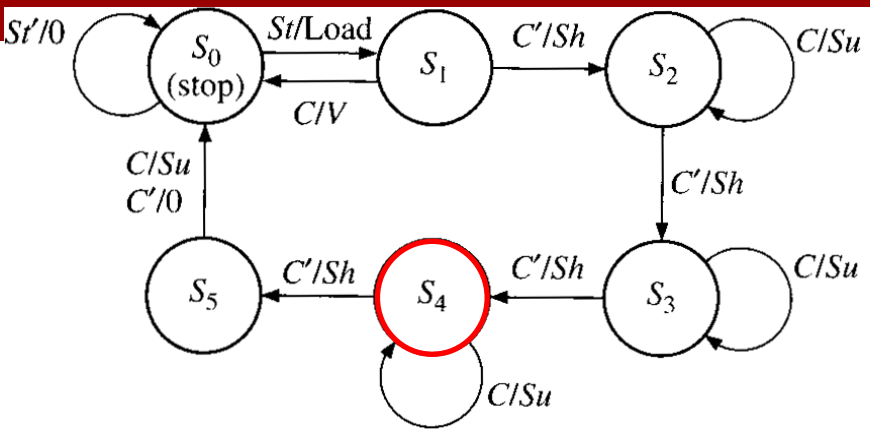


(135 ÷ 13 = 10 with 5)

$$
\begin{array}{r|l}
 & 1010 \\
\hline
1101 & 10000111 \\
 & \underline{1101} \\
 & 0111 \\
 & \underline{0000} \\
 & 1111 \\
 & \underline{1101} \\
 & 0101 \\
 & \underline{0000} \\
 & 0101
\end{array}
$$

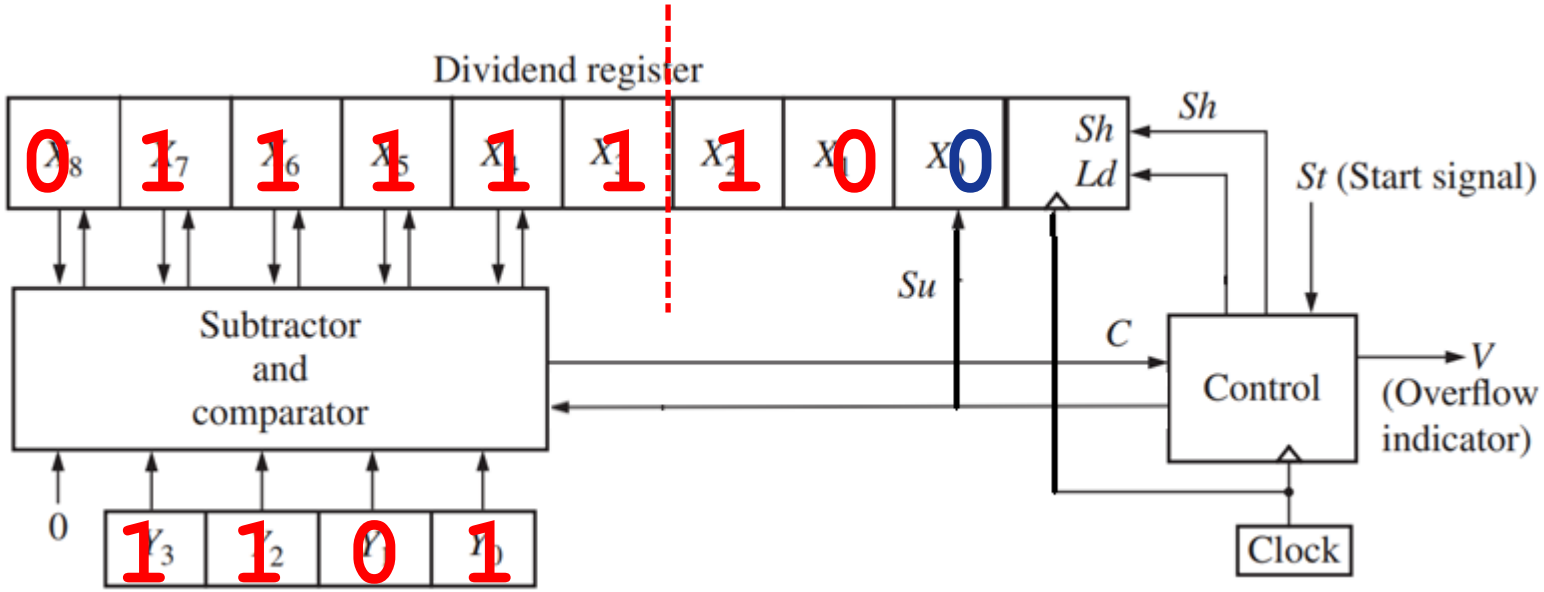


4.12.1 Unsigned Dividers

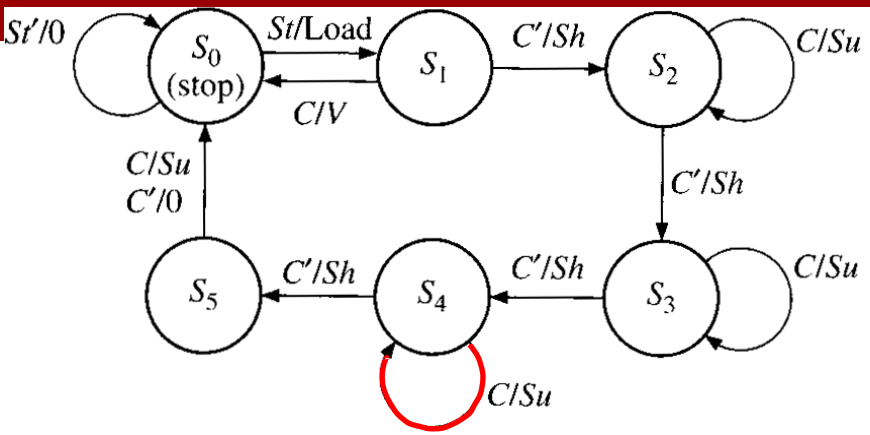


(135 ÷ 13 = 10 with 5)

$$
\begin{array}{r|l}
 & 1010 \\
\hline
1101 & 10000111 \\
 & \underline{1101} \\
 & 0111 \\
 & \underline{0000} \\
 & 1111 \\
 & \underline{1101} \\
 & 0101 \\
 & \underline{0000} \\
 & 0101
\end{array}
$$

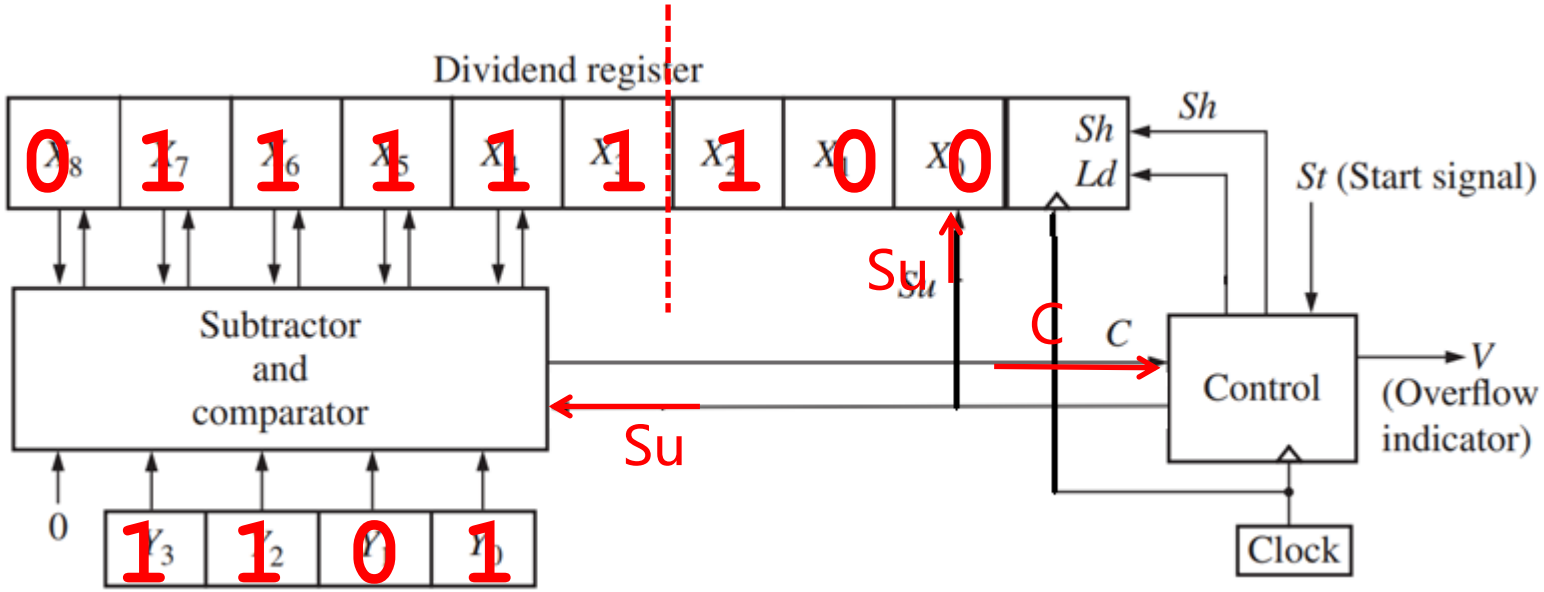


4.12.1 Unsigned Dividers

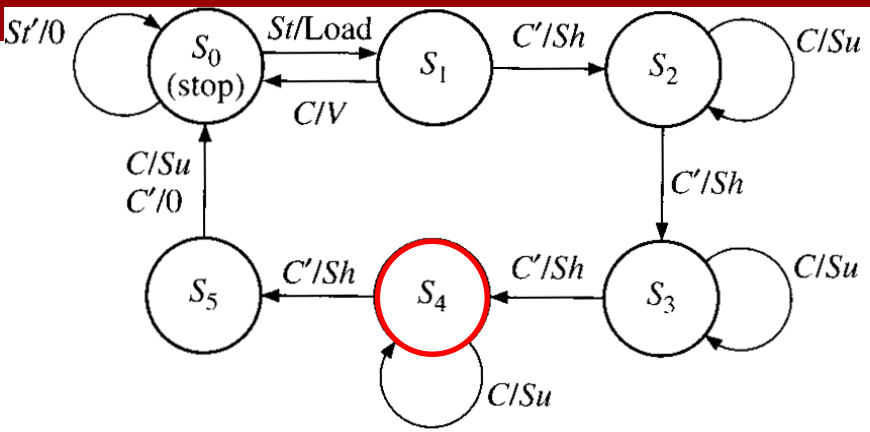


(135 ÷ 13 = 10 with 5)

$$
\begin{array}{r|l}
 & 1010 \\
\hline
1101 & 10000111 \\
 & \underline{1101} \\
 & 0111 \\
 & \underline{0000} \\
 & 1111 \\
 & \underline{1101} \\
 & 0101 \\
 & \underline{0000} \\
 & 0101
\end{array}
$$

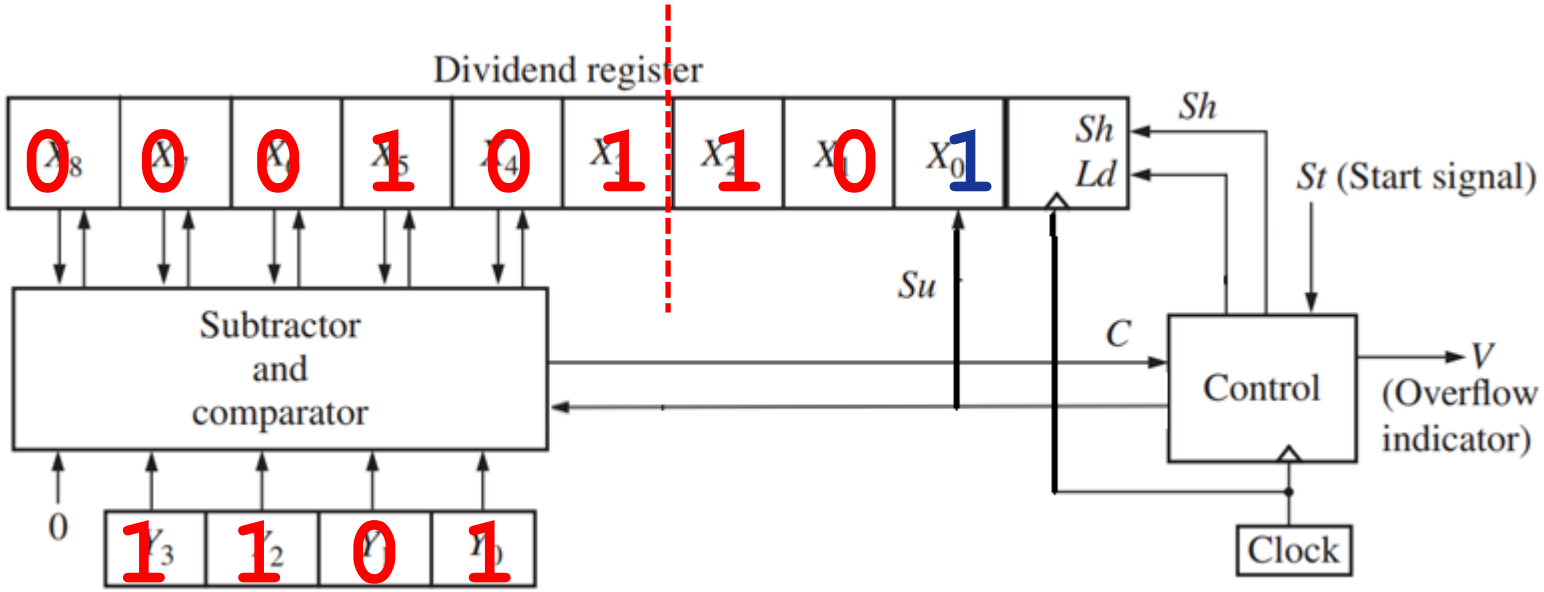


4.12.1 Unsigned Dividers

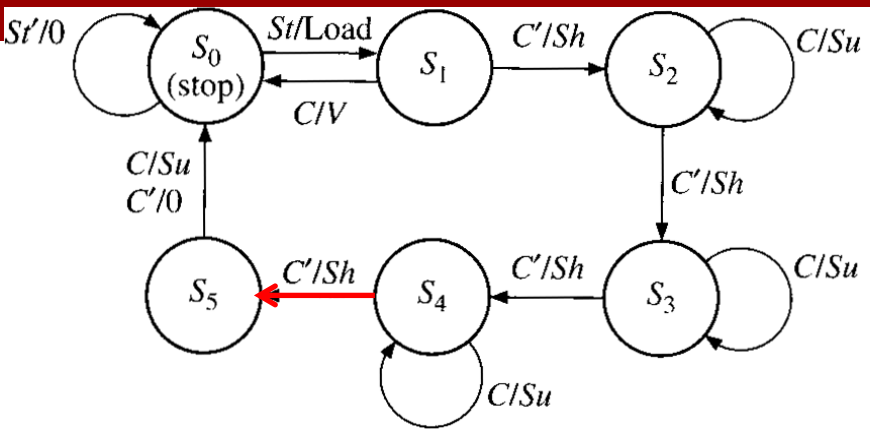


(135 ÷ 13 = 10 with 5)

```
        1010
1101 ) 10000111
       1101
       0111
       0000
        1111
        1101
         0101
         0000
         0101
```

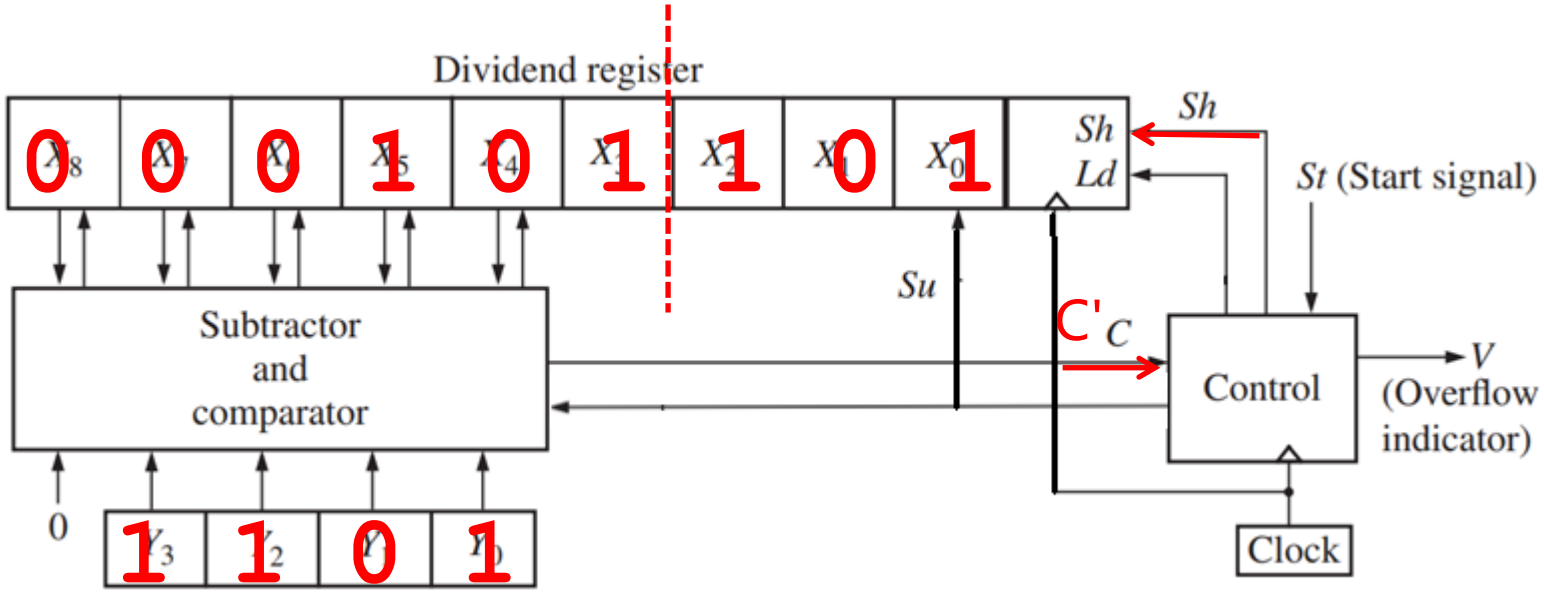


4.12.1 Unsigned Dividers

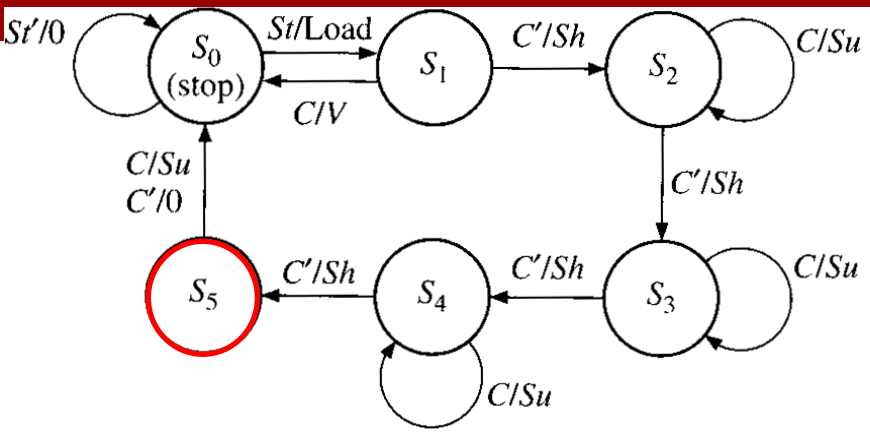


(135 ÷ 13 = 10 with 5)

```
        1010
1101 / 10000111
       1101
       0111
       0000
        1111
        1101
         0101
         0000
         0101
```

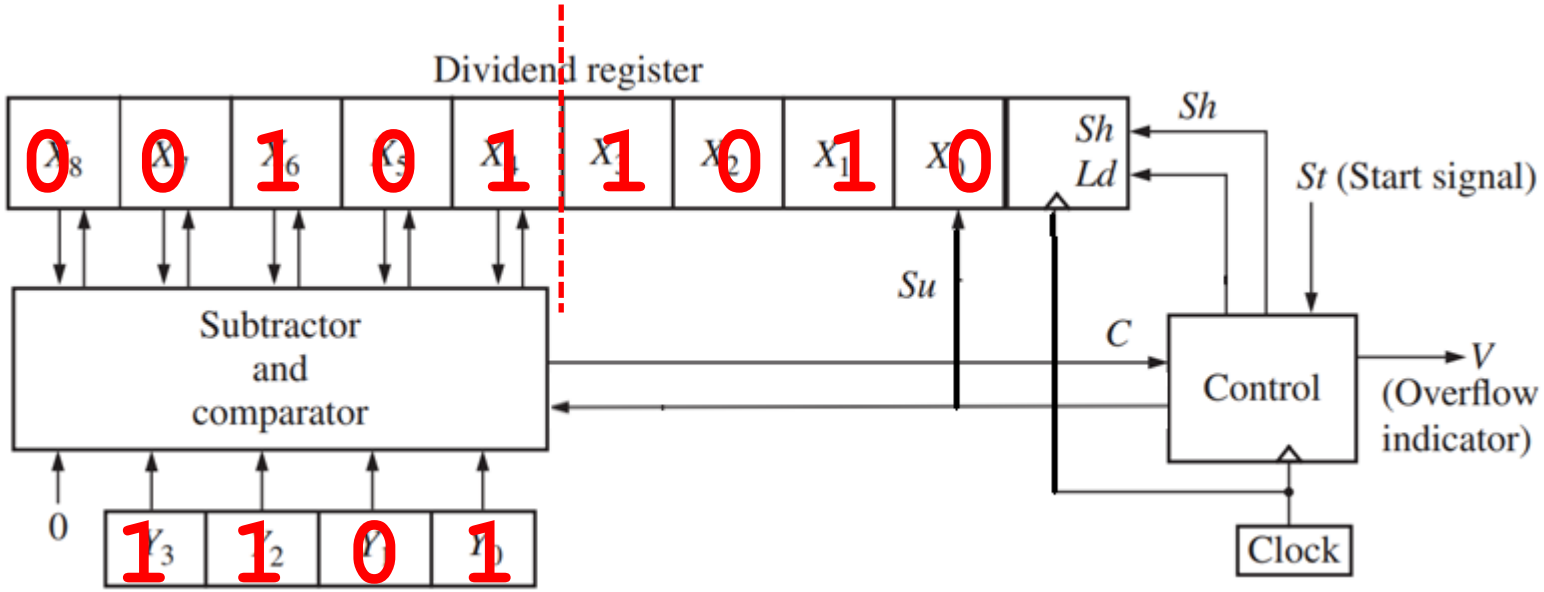


4.12.1 Unsigned Dividers

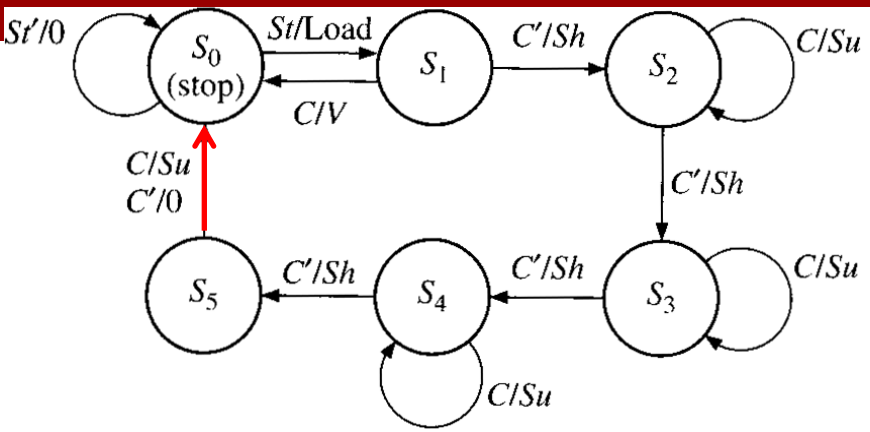


(135 ÷ 13 = 10 with 5)

$$
\begin{array}{r|l}
 & 1010 \\
\hline
1101 & 10000111 \\
 & \underline{1101} \\
 & 0111 \\
 & \underline{0000} \\
 & 1111 \\
 & \underline{1101} \\
 & 0101 \\
 & \underline{0000} \\
 & 0101
\end{array}
$$

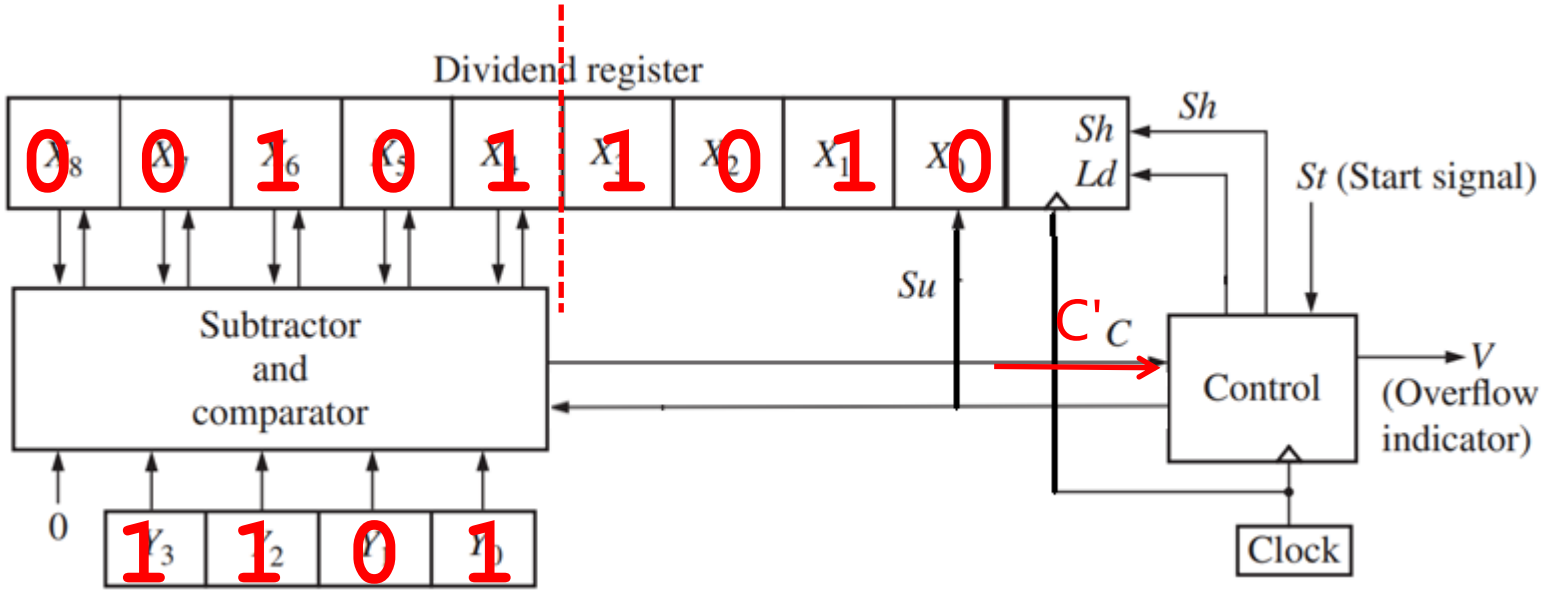


4.12.1 Unsigned Dividers

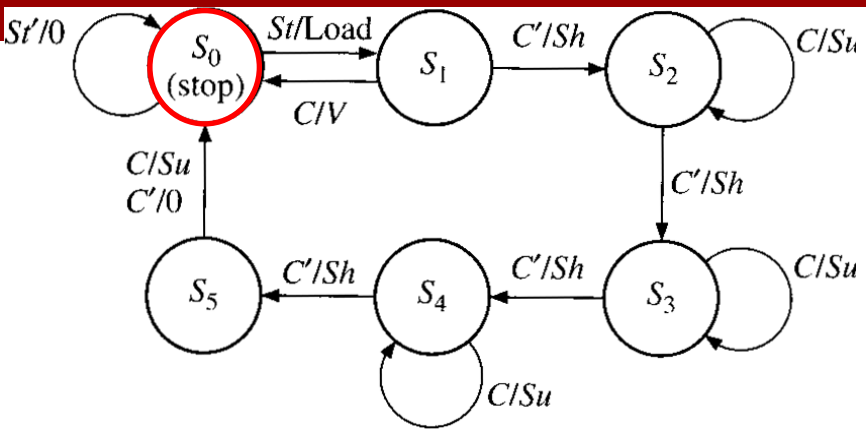


$$
\begin{array}{r}
1010 \\
1101 \overline{)\,10000111} \\
1101 \\
\hline
0111 \\
0000 \\
\hline
1111 \\
1101 \\
\hline
0101 \\
0000 \\
\hline
0101
\end{array}
$$

(135 ÷ 13 = 10 with 5)

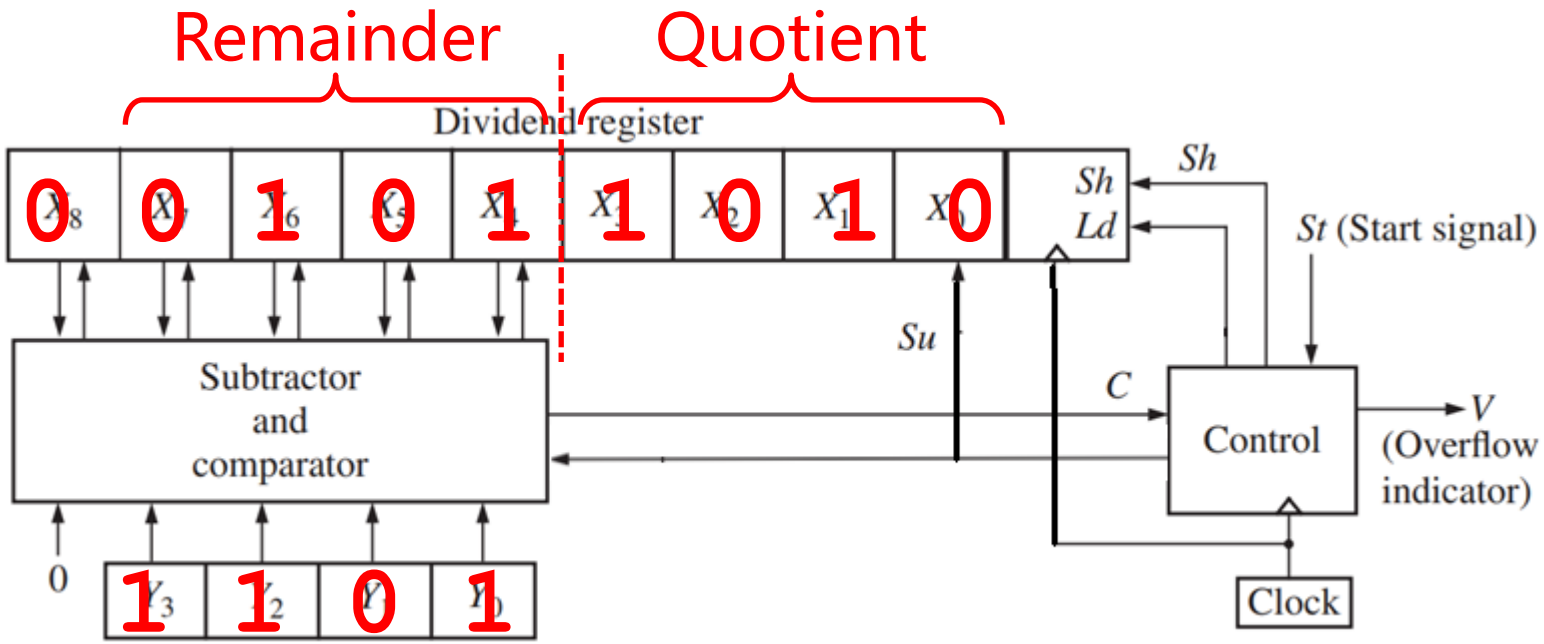


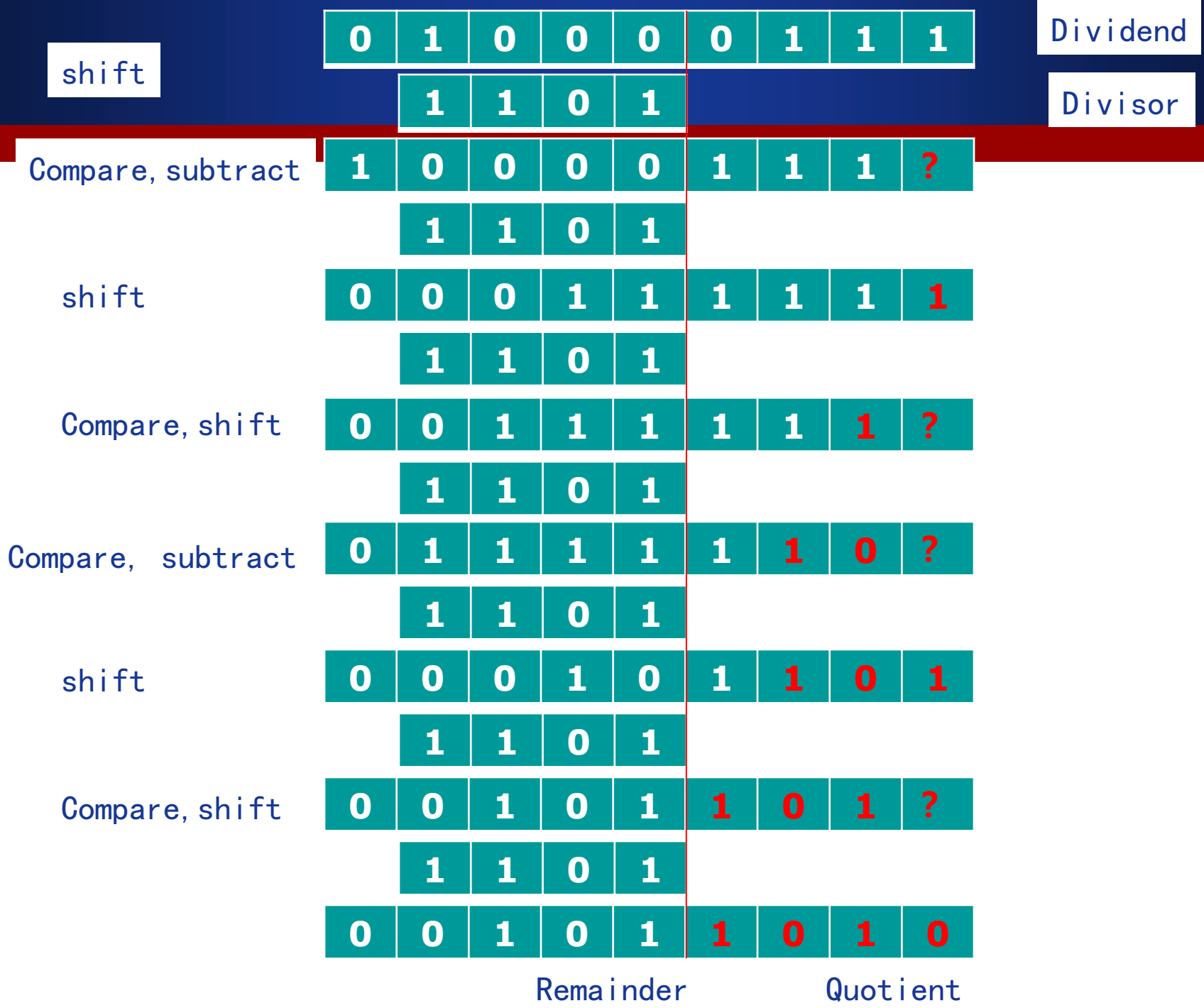
12 Binary Dividers



$(135 \div 13 = 10 \text{ with } 5)$

$$
\begin{array}{r|l}
 & 1010 \\
\hline
1101 & 10000111 \\
 & \underline{1101} \\
 & 0111 \\
 & \underline{0000} \\
 & 1111 \\
 & \underline{1101} \\
 & 0101 \\
 & \underline{0000} \\
 & 0101
\end{array}
$$

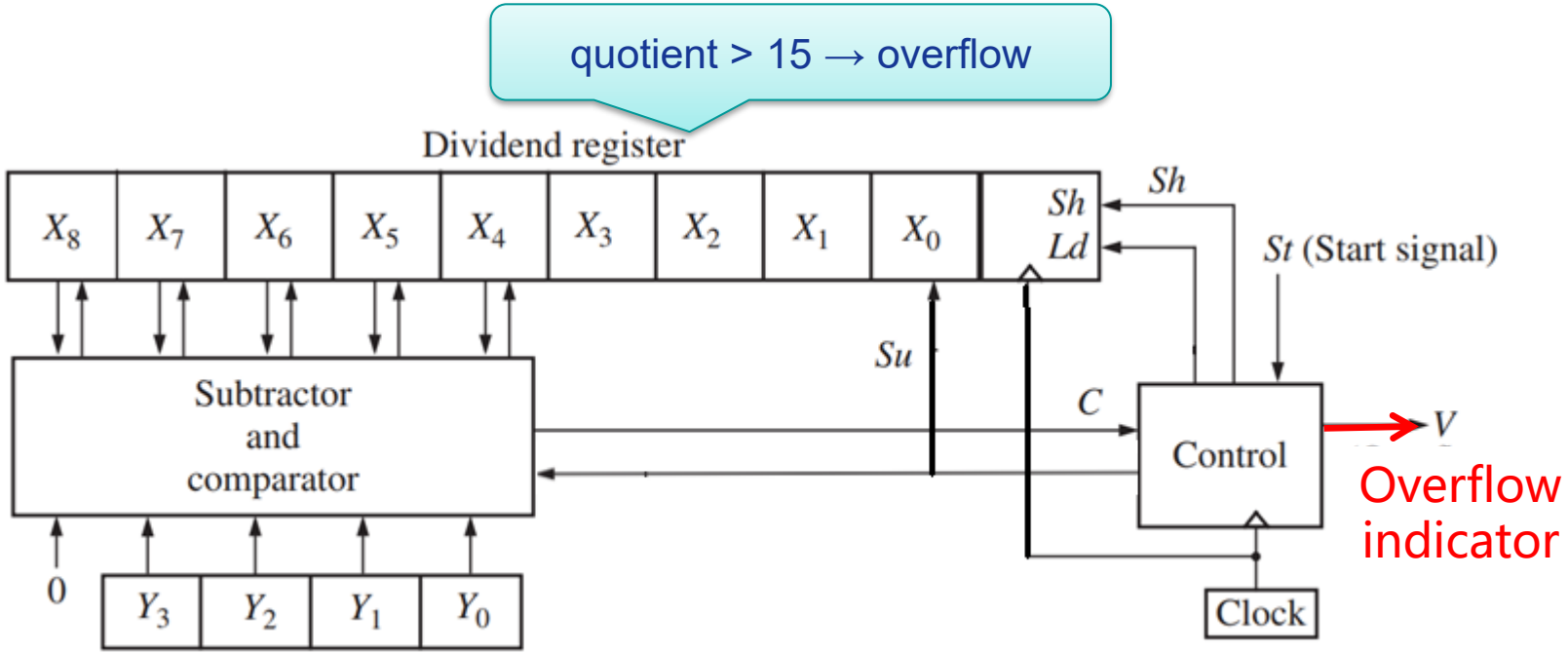



Dividend
Divisor
shift
0 1 0 0 0 0 1 1 1
1 1 0 1
Compare, subtract
1 0 0 0 0 1 1 1 ?
1 1 0 1
shift
0 0 0 1 1 1 1 1 1
1 1 0 1
Compare, shift
0 0 1 1 1 1 1 1 ?
1 1 0 1
Compare, subtract
0 1 1 1 1 1 1 0 ?
1 1 0 1
shift
0 0 0 1 0 1 1 0 1
1 1 0 1
Compare, shift
0 0 1 0 1 1 0 1 ?
1 1 0 1
0 0 1 0 1 1 0 1 0
Remainder
Quotient

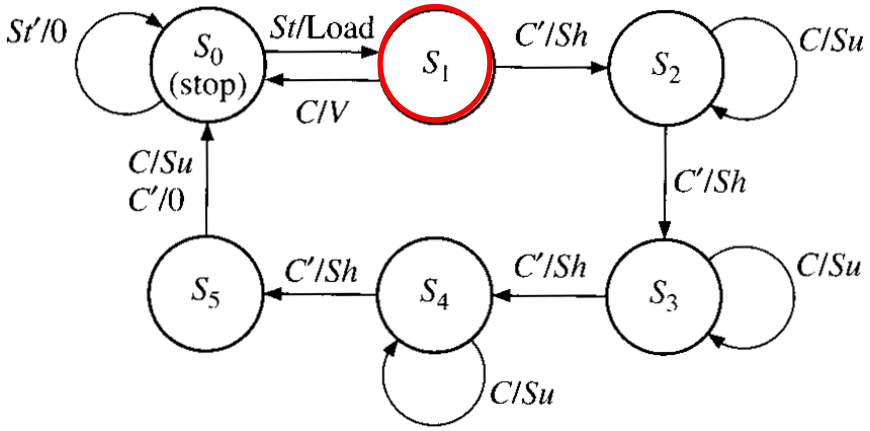
4.12.1 Unsigned Dividers

Overflow（溢出）

Overflow occurs when the quotient contains more bits than are available for storing the quotient

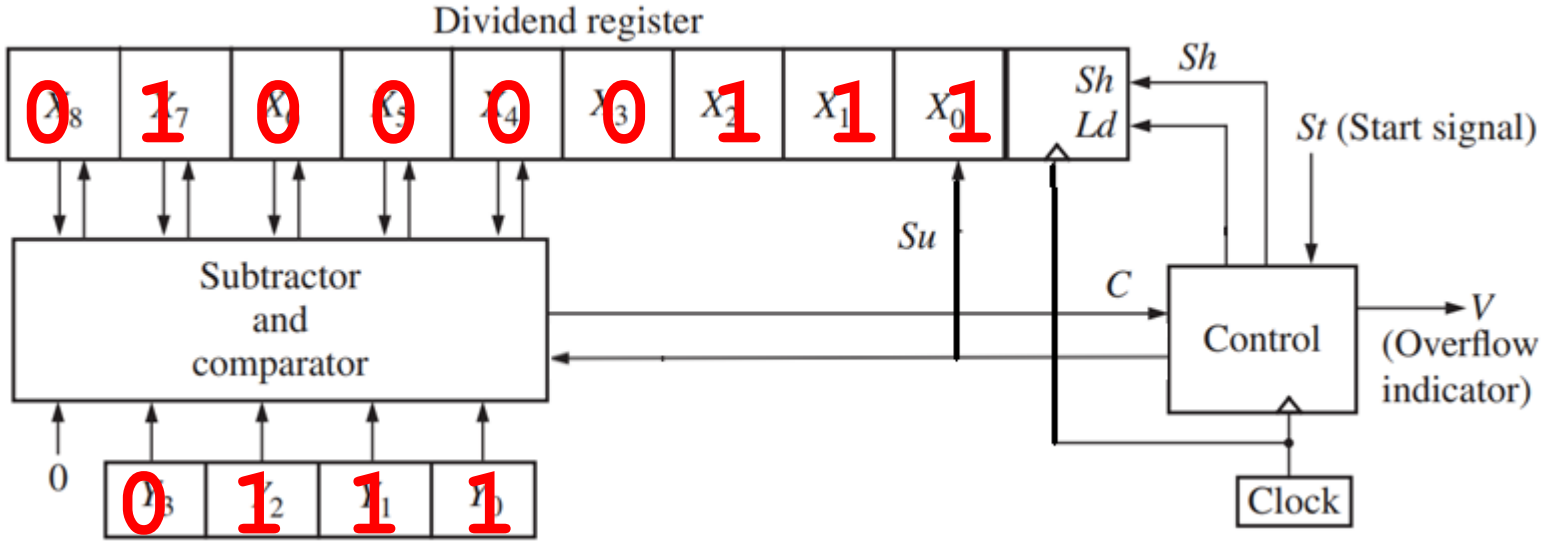


12 Binary Dividers

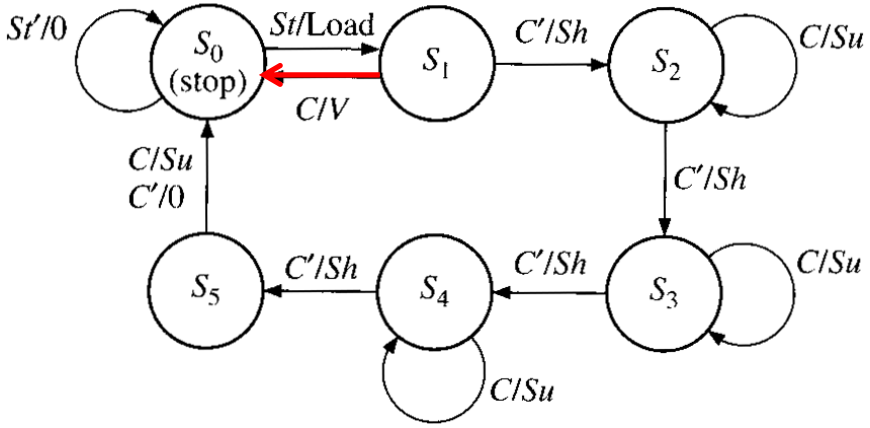


Example:

135 / 7 = 19•••2

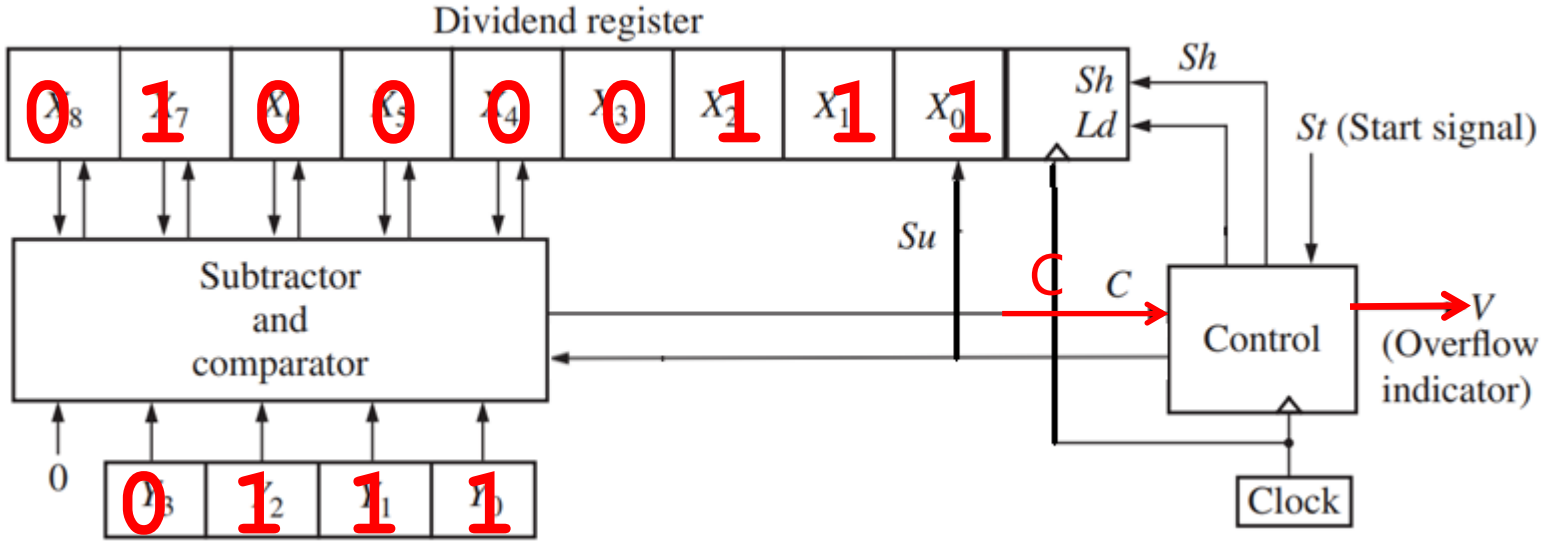


12 Binary Dividers



Example:

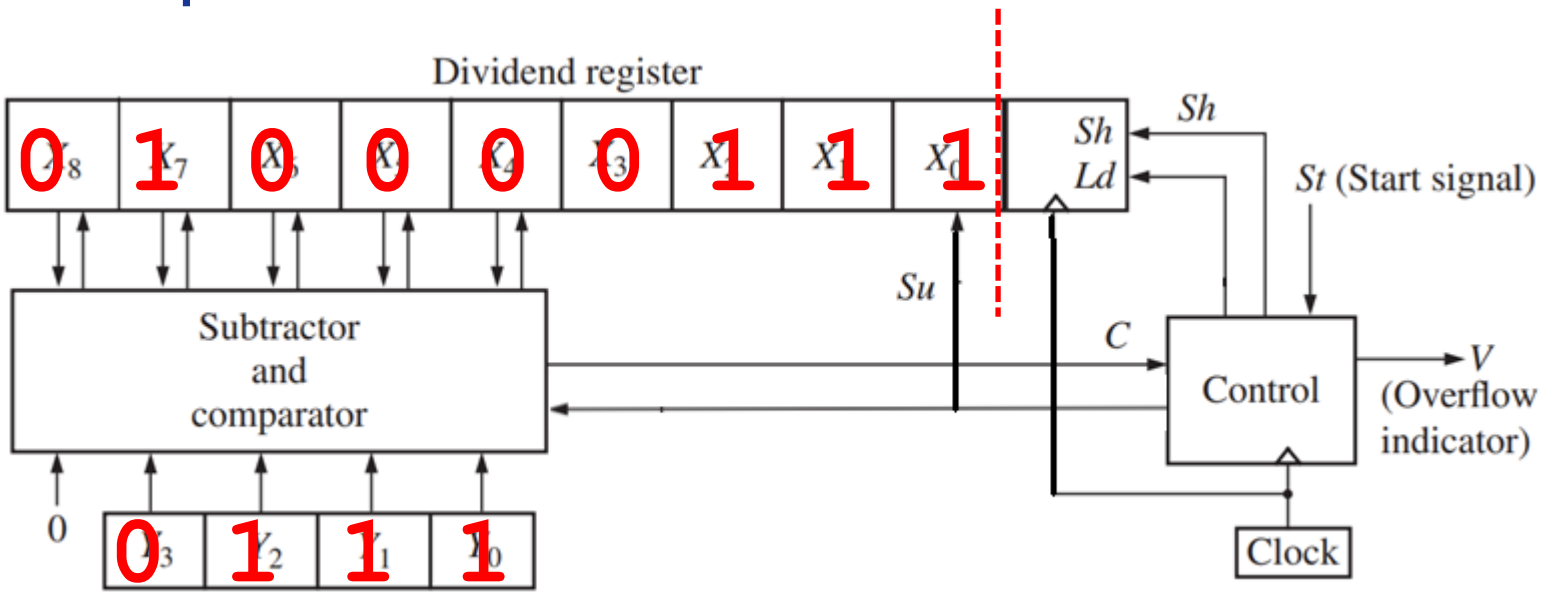
135 / 7 = 19•••2



12 Binary Dividers

Overflow

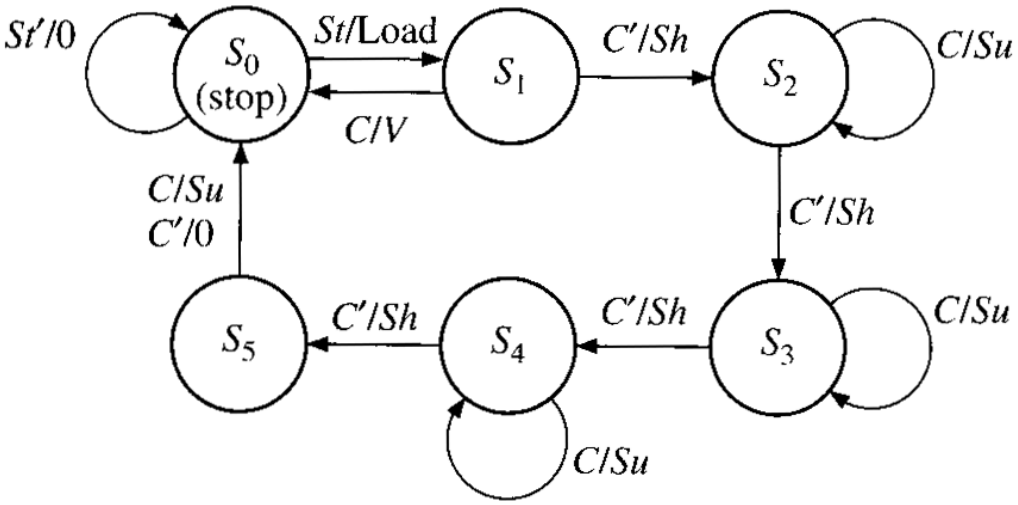
Example: 135 / 7



$$X_8X_7X_6X_5X_4 \geq Y_3Y_2Y_1Y_0$$

$$\frac{X_8X_7X_6X_5X_4X_3X_2X_1X_0}{Y_3Y_2Y_1Y_0} \geq \frac{X_8X_7X_6X_5X_40000}{Y_3Y_2Y_1Y_0} = \frac{X_8X_7X_6X_5X_4 \times 16}{Y_3Y_2Y_1Y_0} \geq 16$$

State Diagram for Divider Control Circuit

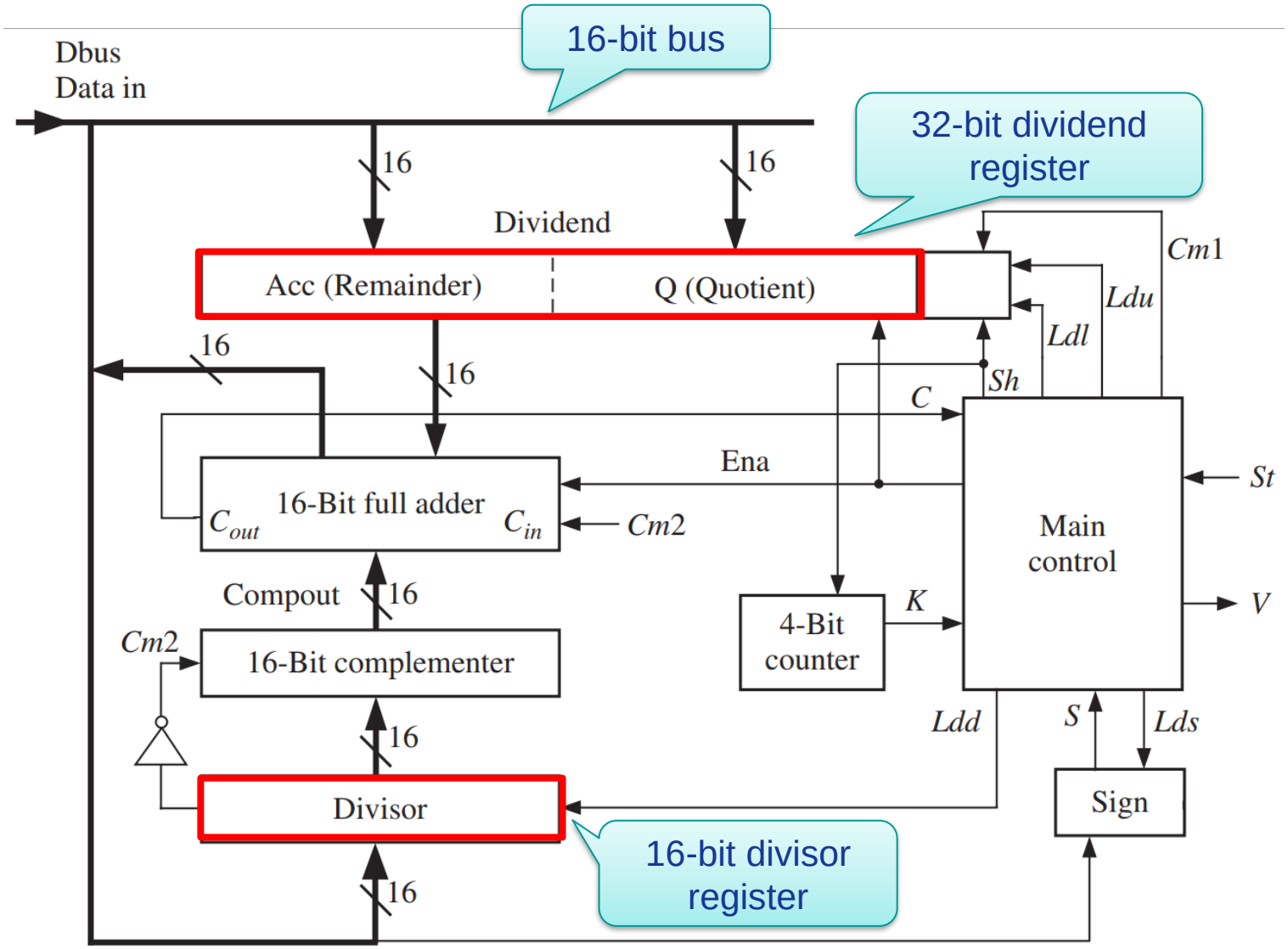


State Table for Divider Control Circuit

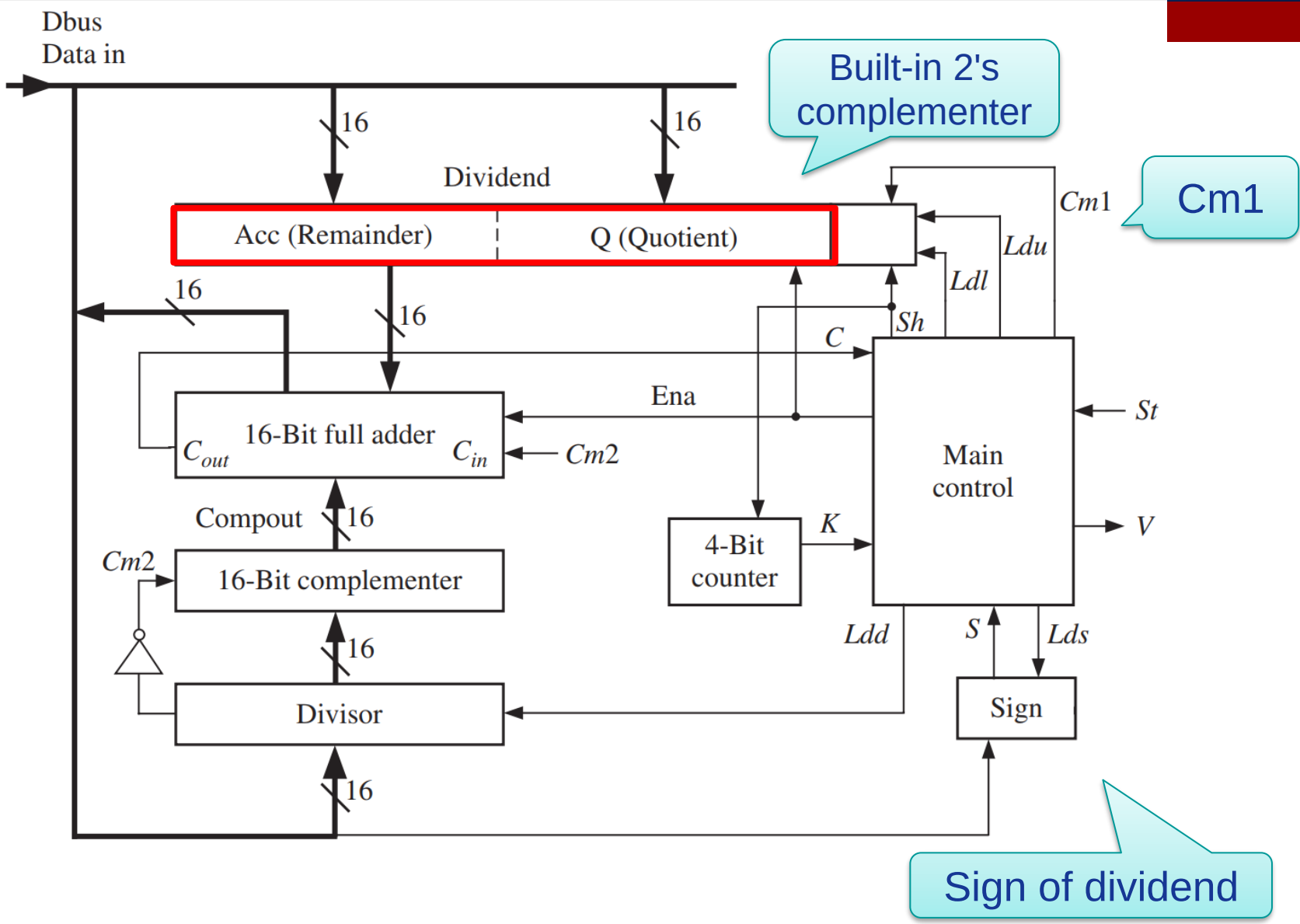
State	StC 00	01	11	10	StC 00	01	11	10
S_0	S_0	S_0	S_1	S_1	0	0	Load	Load
S_1	S_2	S_0	—	—	*Sh*	V	—	—
S_2	S_3	S_2	—	—	*Sh*	*Su*	—	—
S_3	S_4	S_3	—	—	*Sh*	*Su*	—	—
S_4	S_5	S_4	—	—	*Sh*	*Su*	—	—
S_5	S_0	S_0	—	—	0	*Su*	—	—

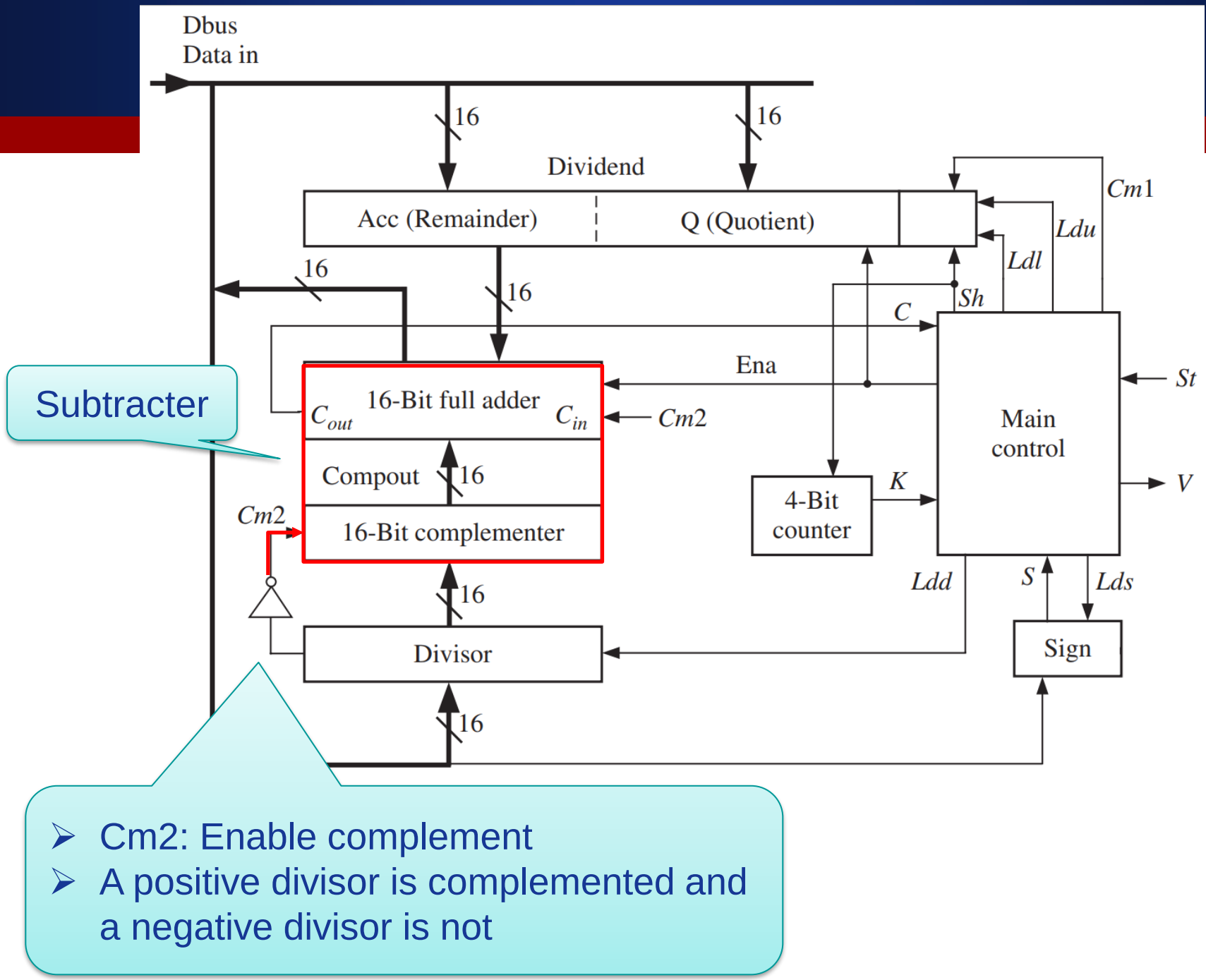
-- : “don't care” because St = 0 in S1~S5

4.12.2 Signed Divider



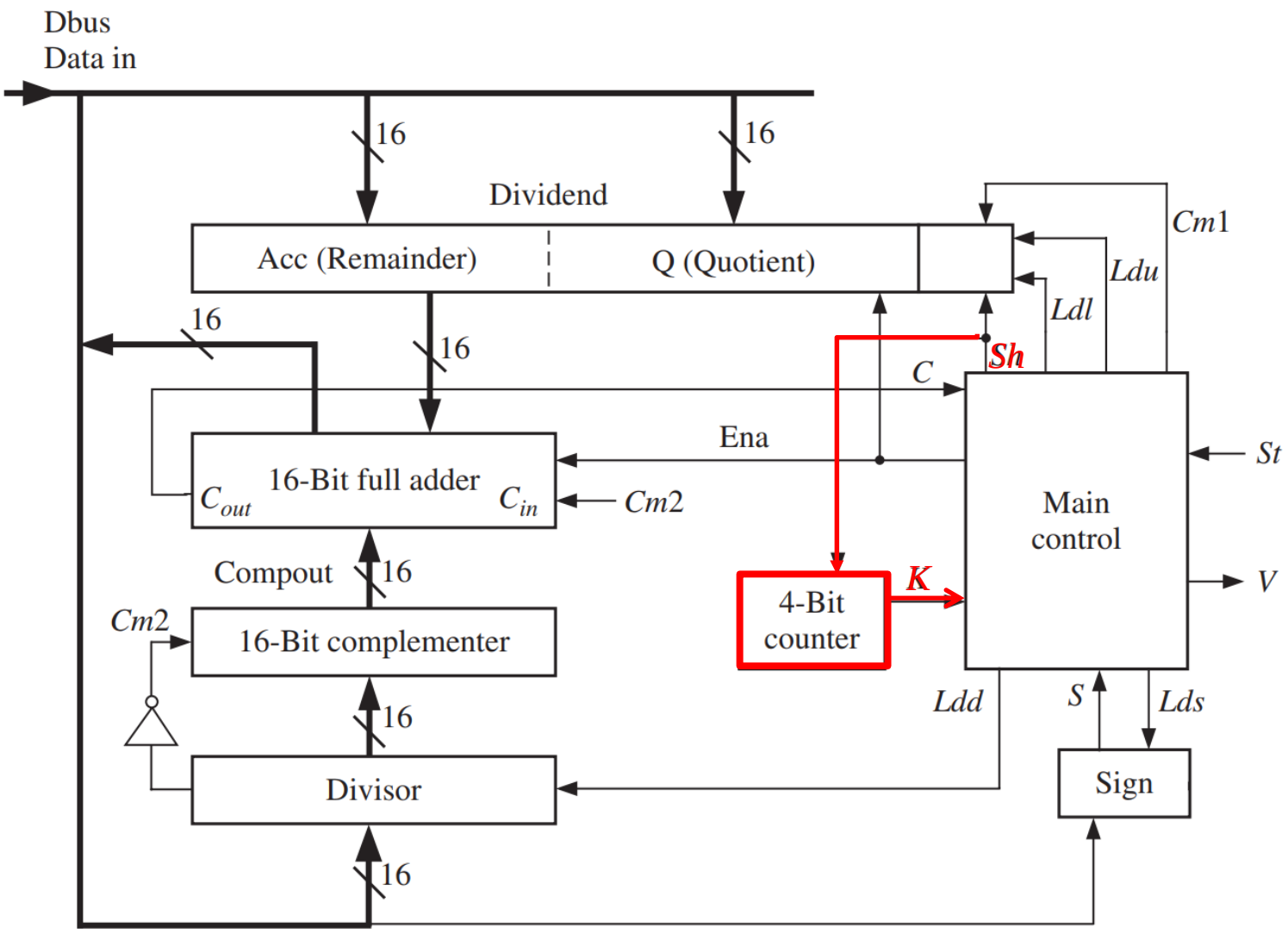
4.12.2 Signed Divider



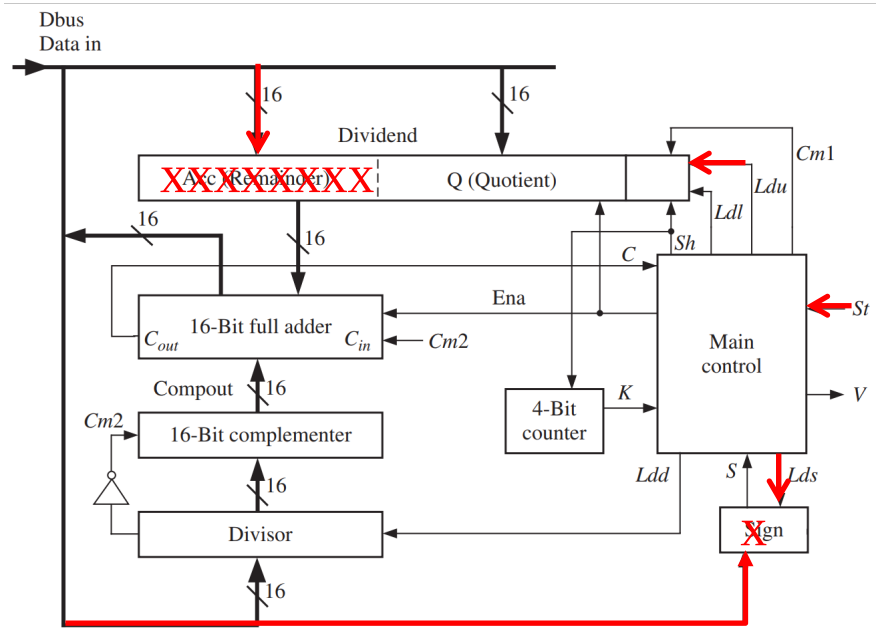


- Cm2: Enable complement
- A positive divisor is complemented and a negative divisor is not

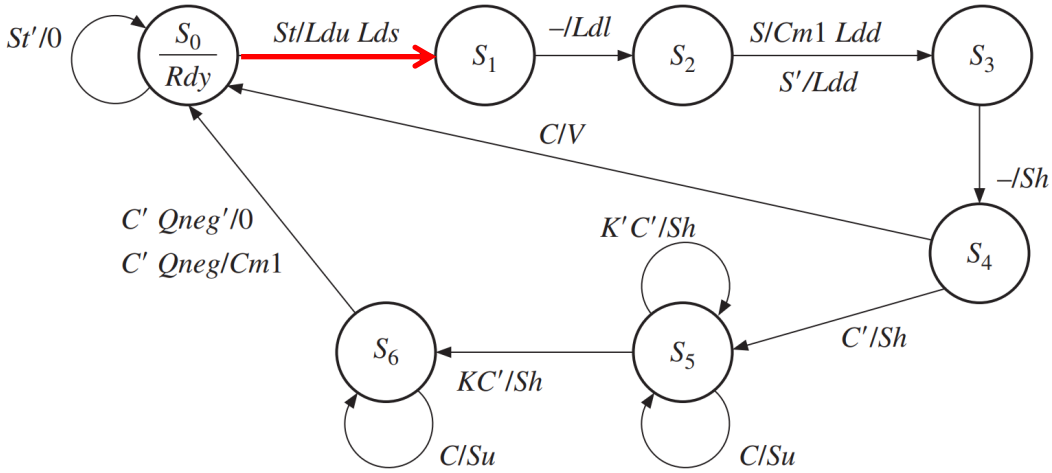
4.12.2 Signed Divider



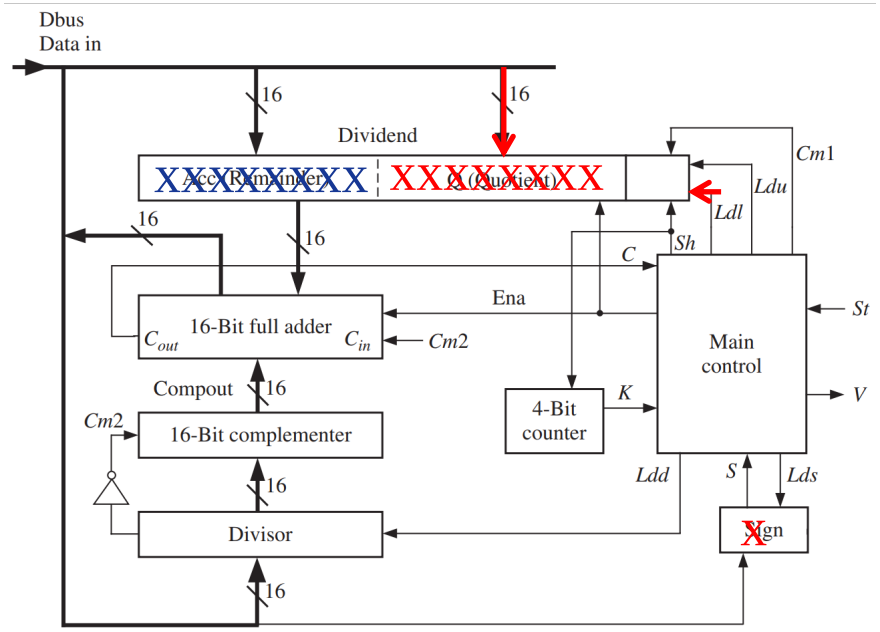
Procedure for carrying out the signed division



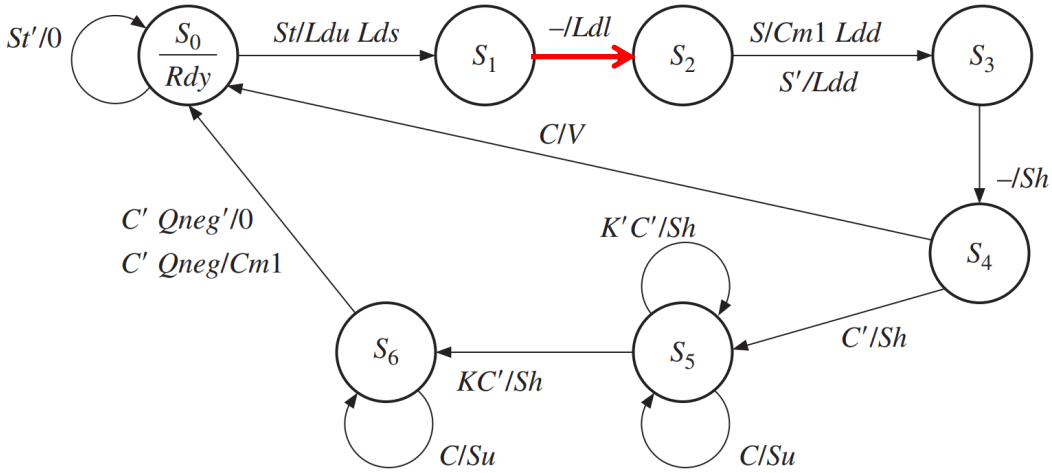
1. Load dividend(31:16) from the bus
   Copy dividend(31) into sign flip-flop **S**



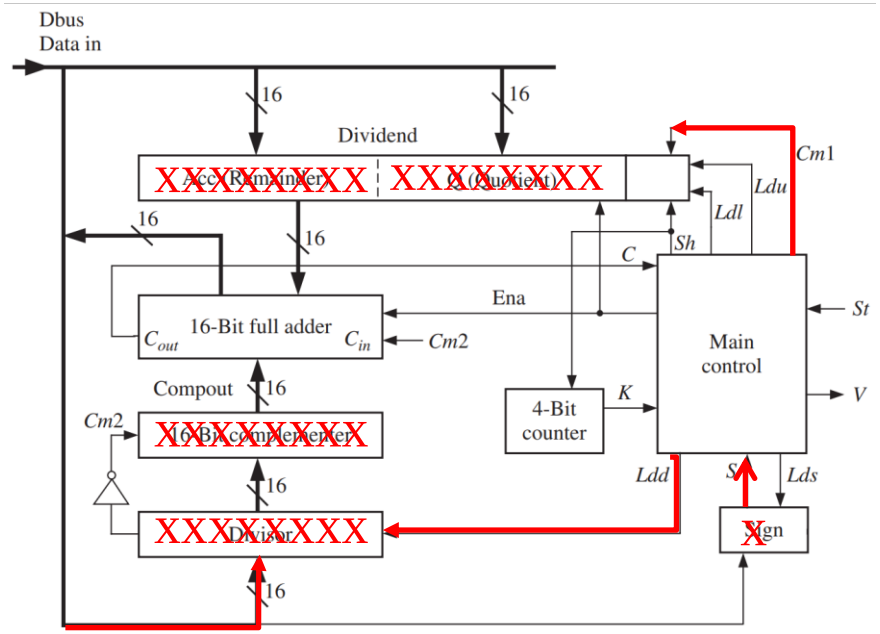
Procedure for carrying out the signed division



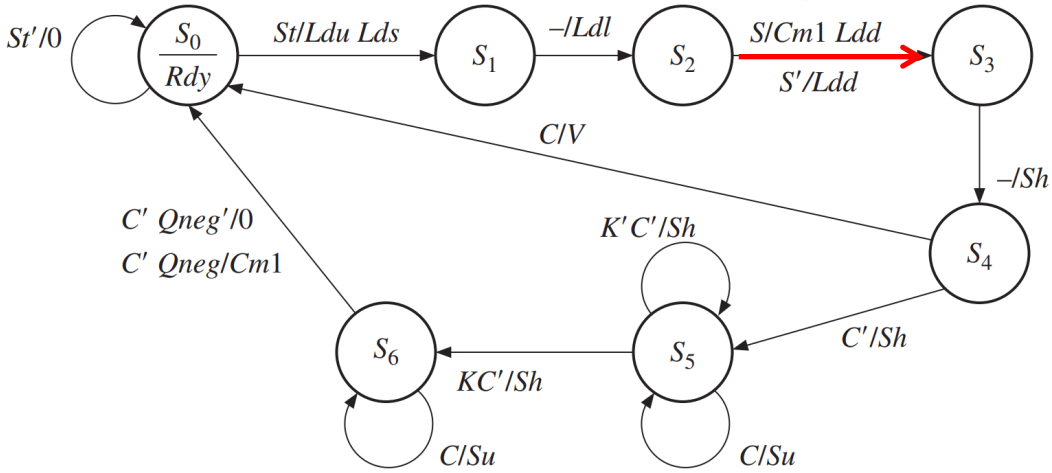
2. Load dividend(15:0) from the bus



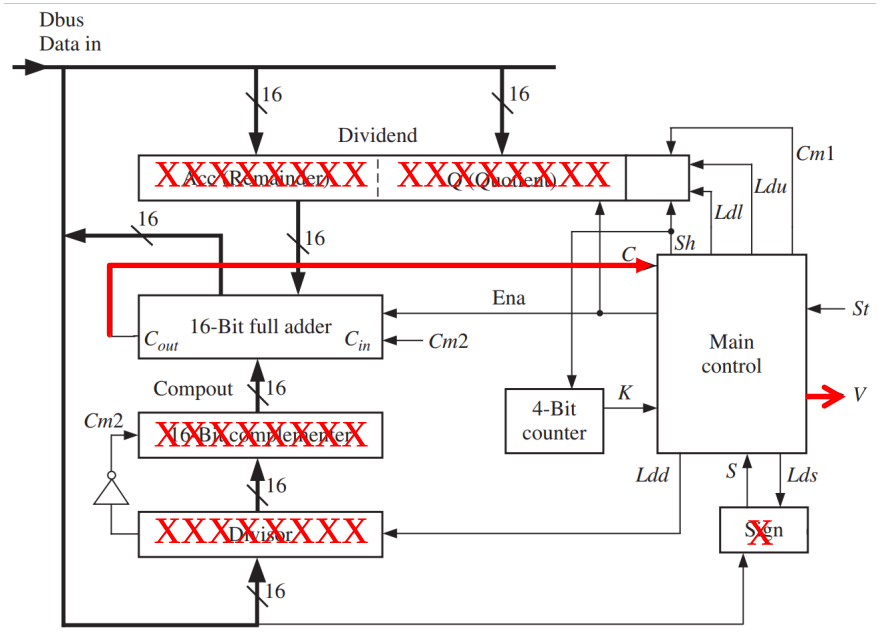
Procedure for carrying out the signed division



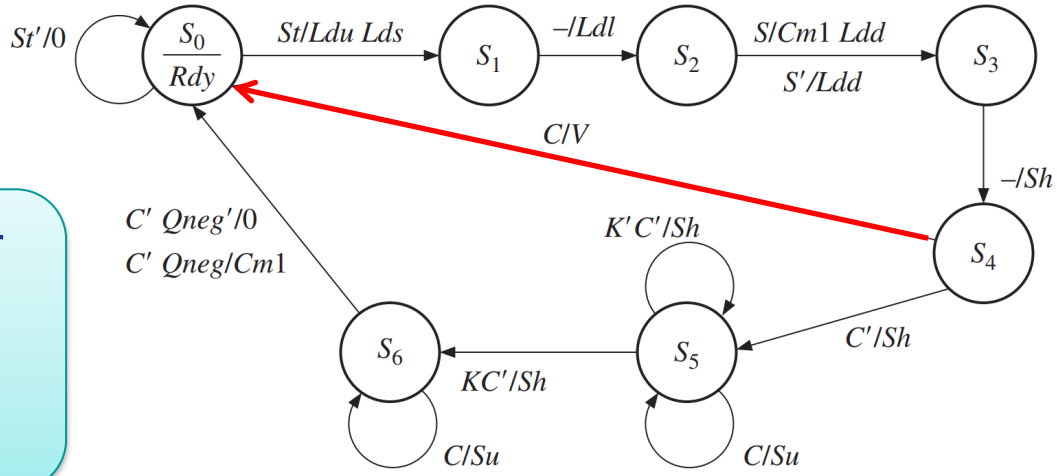
3. Load the divisor from the bus
4. Complement the dividend if it is negative



Procedure for carrying out the signed division

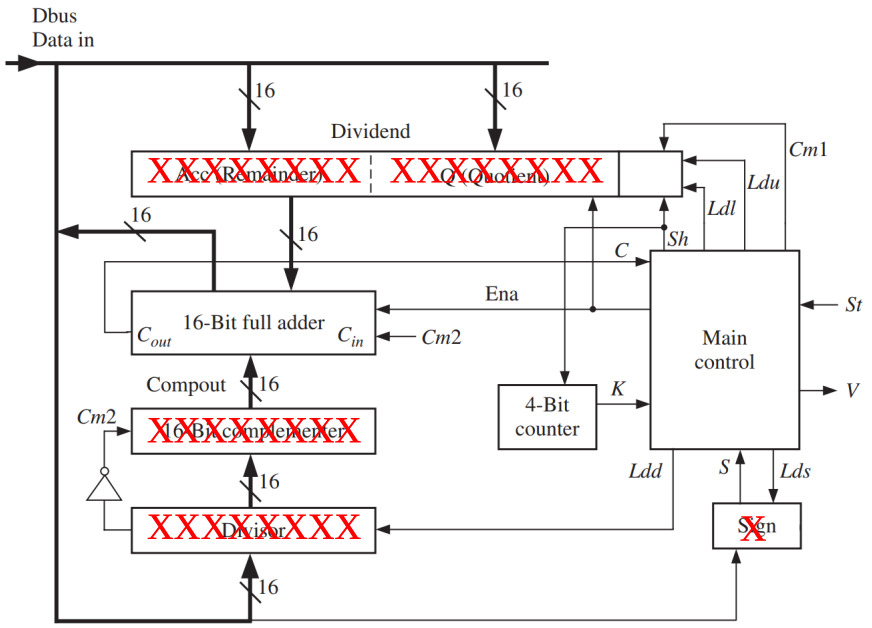


5. If an overflow condition is present, go to the done state

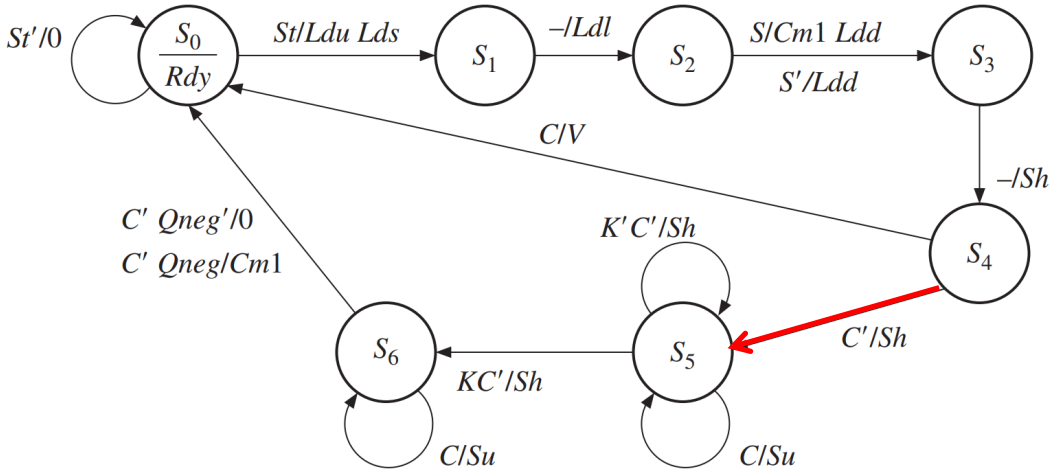


- C: Carry output from adder
- If C = 1, the divisor can be subtracted from the upper dividend

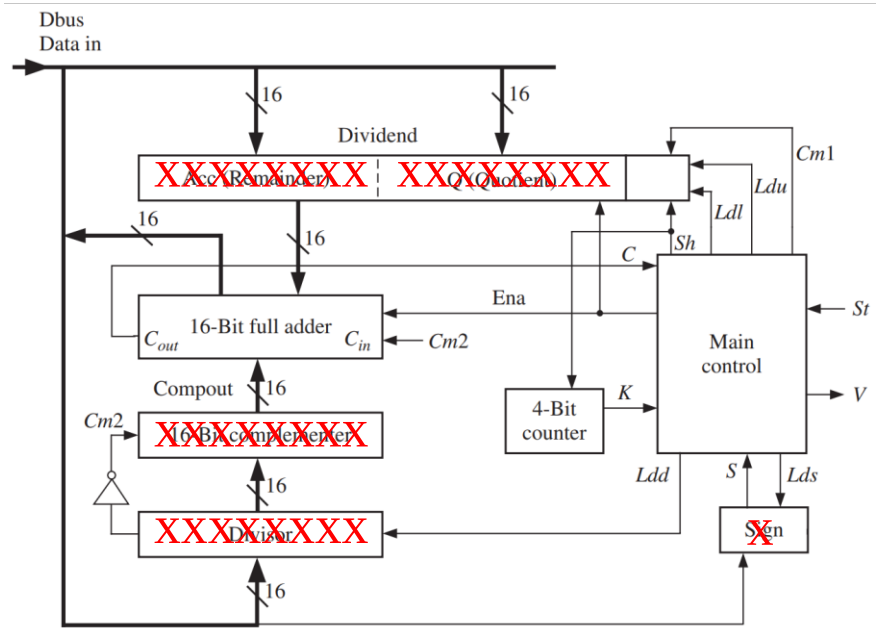
Procedure for carrying out the signed division



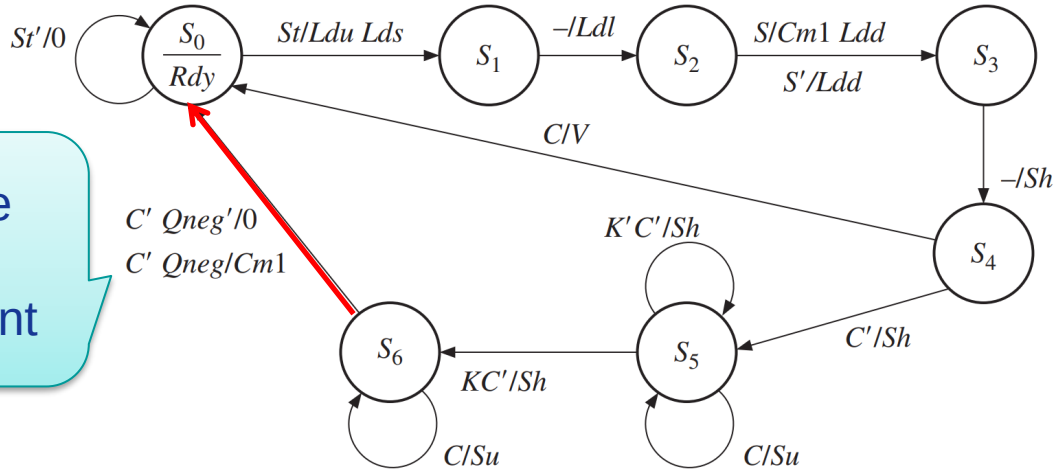
6. Else carry out the division by a series of shifts and subtracts



Procedure for carrying out the signed division

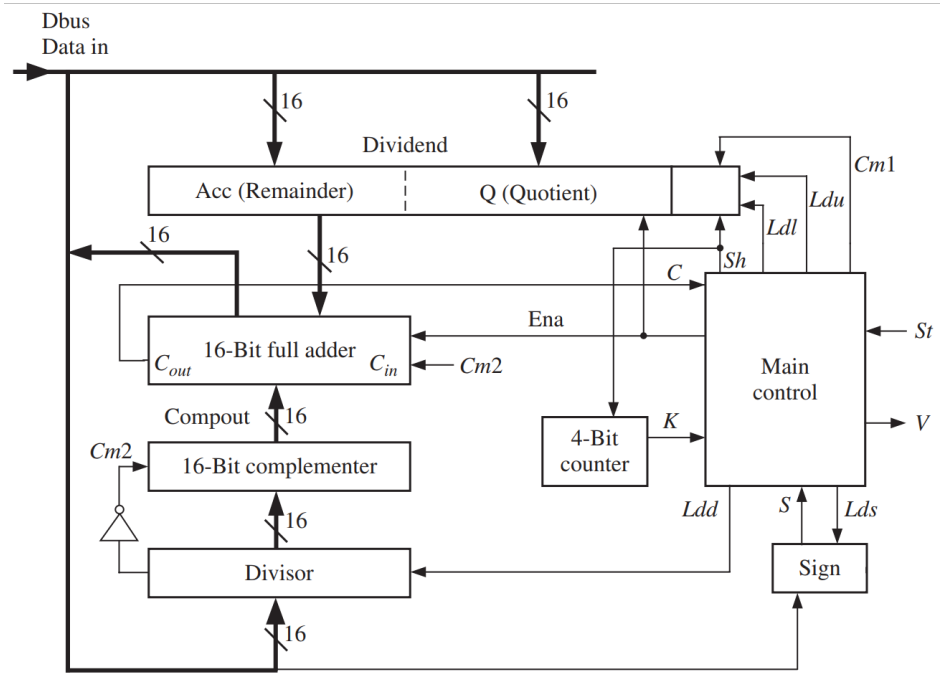


7. When division is complete, complement the quotient if necessary, and go to the done state



- Qneg: Quotient will be negative
- Qneg = 1 when the sign of the dividend and divisor are different

VHDL Model of 32-Bit Signed Divider

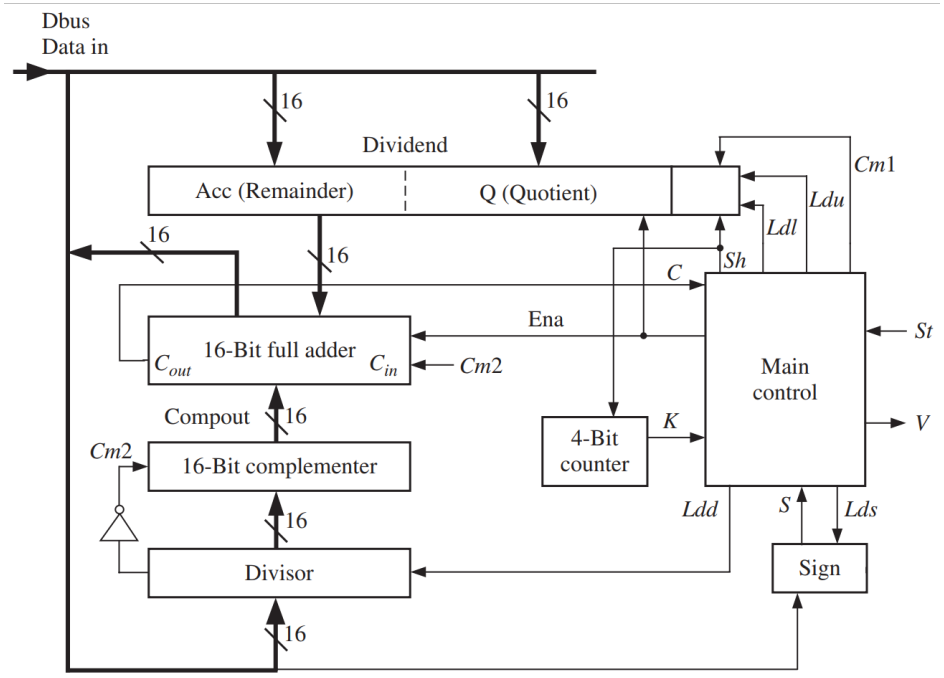



```
library IEEE;
use IEEE.numeric_bit.all;

entity sdiv is
   port(CLK, St: in bit;
        Dbus: in unsigned(15 downto 0);
        Quotient: out unsigned(15 downto 0);
        V, Rdy: out bit);
end sdiv;
```

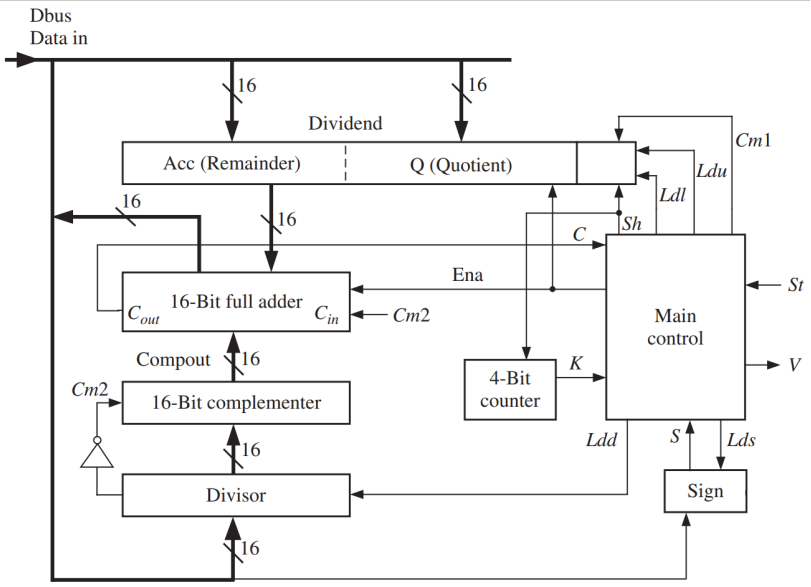
Remainder is not an output here

VHDL Model of 32-Bit Signed Divider




```
architecture Signdiv of sdiv is
signal State: integer range 0 to 6;
signal Count: unsigned(3 downto 0); -- integer range 0 to 15
signal Sign, C, Cm2: bit;
signal Divisor, Sum, Compout: unsigned(15 downto 0);
signal Dividend: unsigned(31 downto 0);
alias Acc: unsigned(15 downto 0) is Dividend(31 downto 16);
```

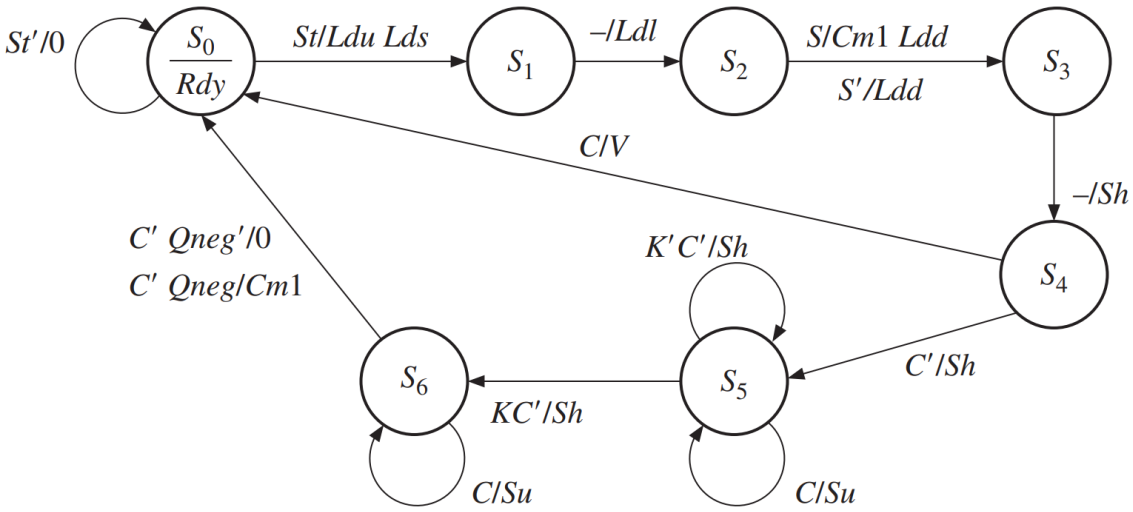
VHDL Model of 32-Bit Signed Divider




```
begin
    Cm2 <= not divisor(15);
    compout <= divisor when Cm2 = '0'
               else not divisor;
    Sum <= Acc + compout + unsigned'(0=>Cm2);
    C <= not Sum(15);
    Quotient <= Dividend(15 downto 0);
    Rdy <= '1' when State = 0 else '0';
    process(CLK)
    begin
```

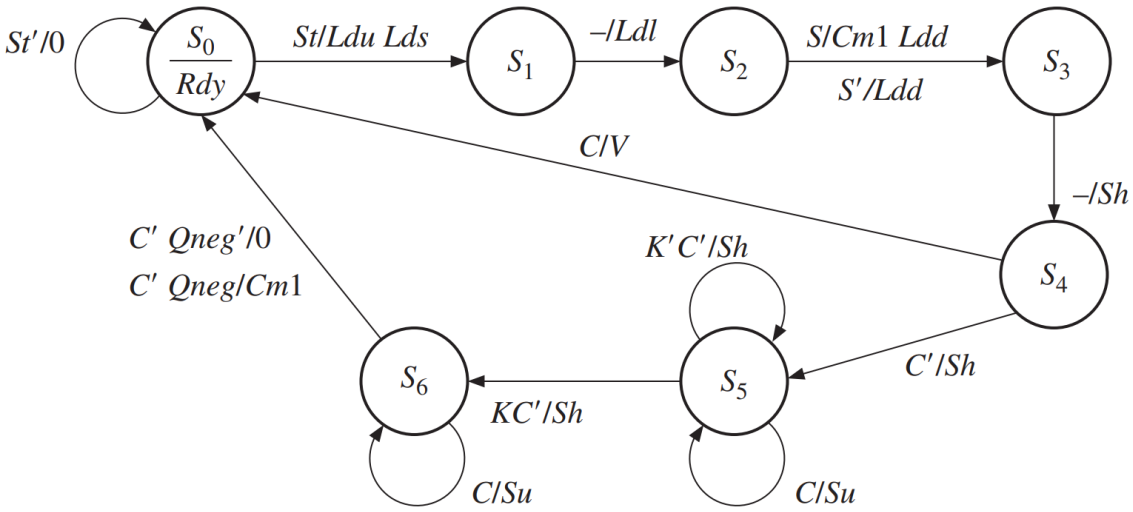
VHDL Model of 32-Bit Signed Divider

```
process(CLK)
begin
   if CLK'event and CLK = '1' then
      case State is
         when 0 =>
            if St = '1' then
               Acc <= Dbus;          -- load upper dividend
               Sign <= Dbus(15);
               State <= 1;
               V <= '0';             -- initialize overflow
               Count <= "0000";      -- initialize counter
            end if;
```



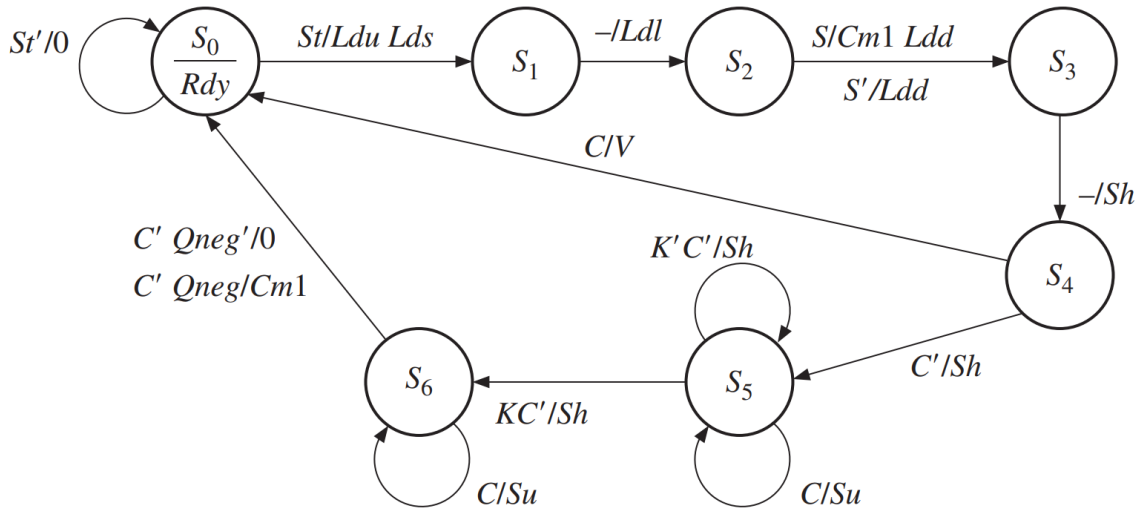
VHDL Model of 32-Bit Signed Divider

```
when 1 =>
    Dividend(15 downto 0) <= Dbus; -- load lower dividend
    State <= 2;
when 2 =>
    Divisor <= Dbus;
    if Sign = '1' then    -- two's complement Dividend if necessary
       Dividend <= not Dividend + 1;
    end if;
    State <= 3;
when 3 =>
    Dividend <= Dividend(30 downto 0) & '0';  -- left shift
    Count <= Count + 1;
    State <= 4;
```



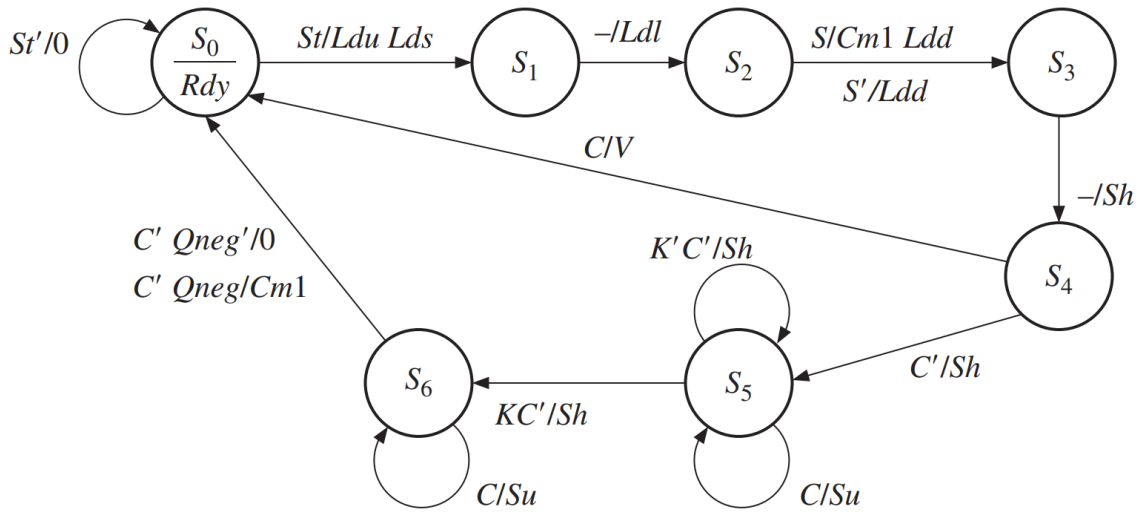
VHDL Model of 32-Bit Signed Divider

```
when 4 =>
   if C = '1' then
      V <= '1';
      State <= 0;
   else
      Dividend <= Dividend(30 downto 0) & '0';
      Count <= Count + 1;
      State <= 5;
   end if;
when 5 =>
   if C = '1' then
      Acc <= Sum;     -- subtract
      Dividend(0) <= '1';
   else
      Dividend <= Dividend(30 downto 0) & '0';   -- left shift
      if Count = 15 then State <= 6; end if;     -- KC'
      Count <= Count + 1;
   end if;
```



VHDL Model of 32-Bit Signed Divider

```
        when 6 =>
           State <= 0;
           if C = '1' then
              Acc <= Sum;      -- subtract
              Dividend(0) <= '1';
              State <= 6;
           elsif (Sign xor Divisor(15)) ='1' then      -- C'Qneq
              Dividend <= not Dividend + 1;            -- 2's complement Dividend
           end if;
        end case;
     end if;
  end process;
end Signdiv;
```



Test bench for signed divider

```
library IEEE;
use IEEE.numeric_bit.all;

entity testsdiv is
end testsdiv;

architecture test1 of testsdiv is
component sdiv
  port(CLK, St: in bit;
       Dbus: in unsigned(15 downto 0);
       Quotient: out unsigned(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 unsigned(31 downto 0);
type arr2 is array(1 to N) of unsigned(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: unsigned(15 downto 0);
signal Dividend: unsigned(31 downto 0);
signal Count: integer range 0 to N;
```

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 (CLK'event and CLK = '1');
      Dbus <= dividendarr(i) (15 downto 0);
      wait until (CLK'event and CLK='1');
      Dbus <= divisorarr(i);
      St <= '0';
      dividend <= dividendarr(i) (31 downto 0); -- save dividend for listing
      divisor <= divisorarr(i);      -- save divisor for listing
      wait until (Rdy = '1');      -- save index for triggering
      count <= i;
   end loop;
  end process;
  sdiv1: sdiv port map(CLK, St, Dbus, Quotient, V, Rdy);
end test1;
```

Test bench for signed divider

```
VSIM 70> add list -notrigger dividend divisor quotient V -trigger count
VSIM 71> run 5300 ns
```

List - Default

ns / delta		/testsdiv/Dividend	/testsdiv/divisor	/testsdiv/Quotient	/testsdiv/V	/testsdiv/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