6 Registers and Counters

6-1 REGISTERS

- **Register-** a group of binary cells suitable for holding binary information.

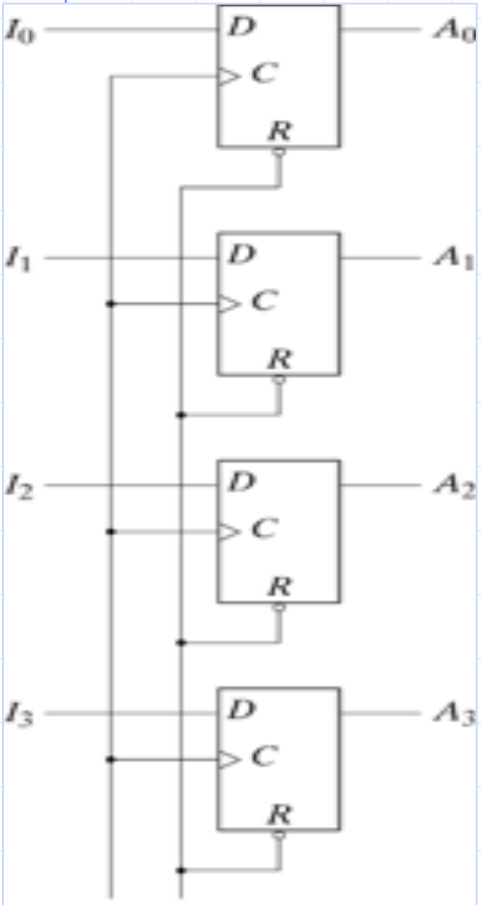


Fig. 6-1 4-Bit Register

❑Clock=0 -> 1 : input information transferred

❑Clock=0 : unchanged

❑Clear=0 : clearing the register to all 0's prior to its clocked operation.

Register with Parallel Load

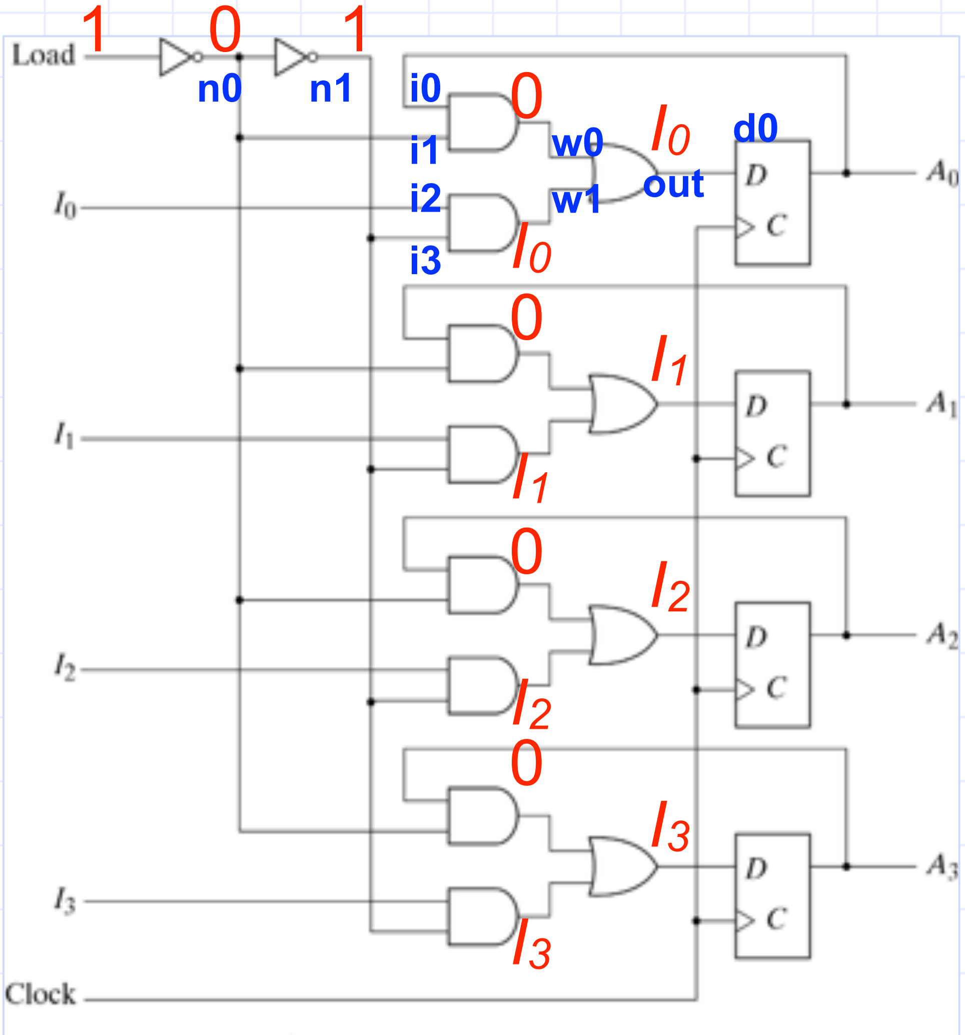


Fig. 6-2 4-Bit Register with Parallel Load

- Clock=0->1 : input information

  -> loading
- Clock=0 or 0->1 or 1: the content of the register -> unchanged
- Load input=1 : the I inputs are transferred into the register

Register with Parallel Load

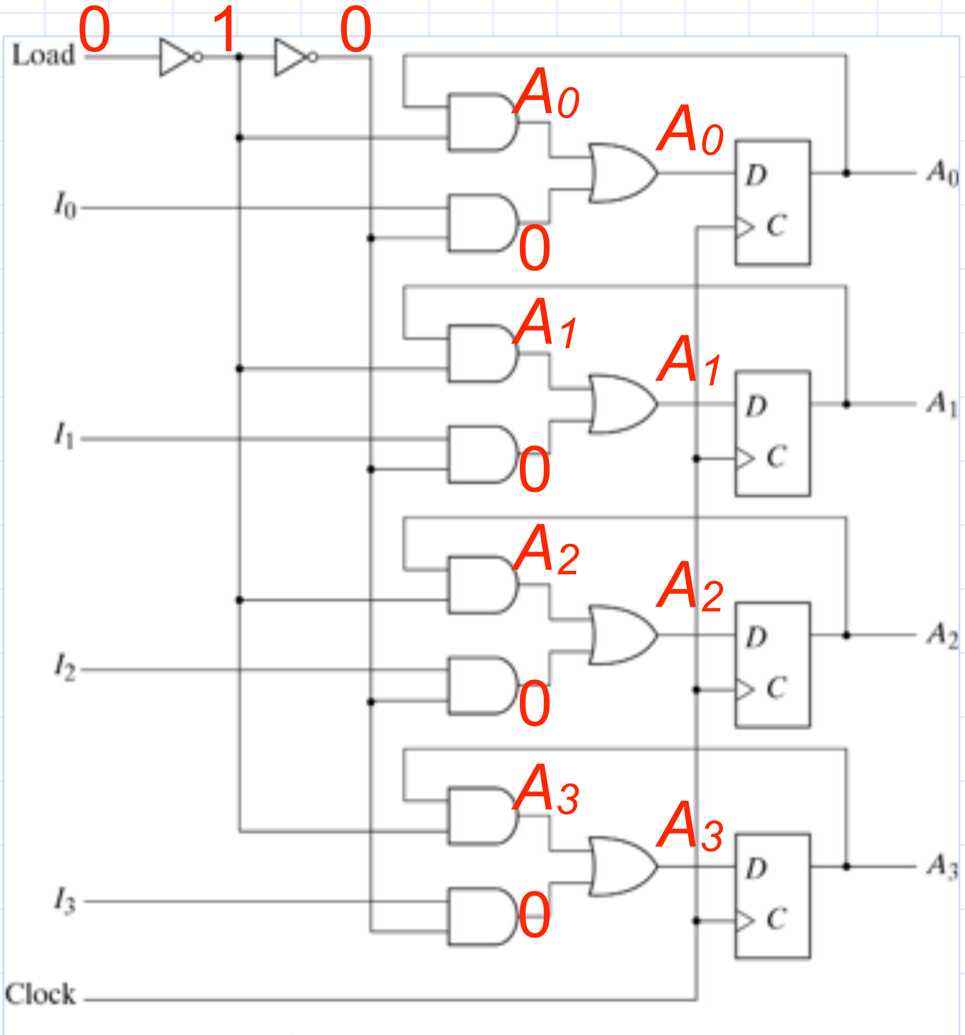


Fig. 6-2 4-Bit Register with Parallel Load

- Load input=0 ; maintain the content of the register

6-2 SHIFT REGISTERS

Shift register-capable of shifting its binary information in one or both directions

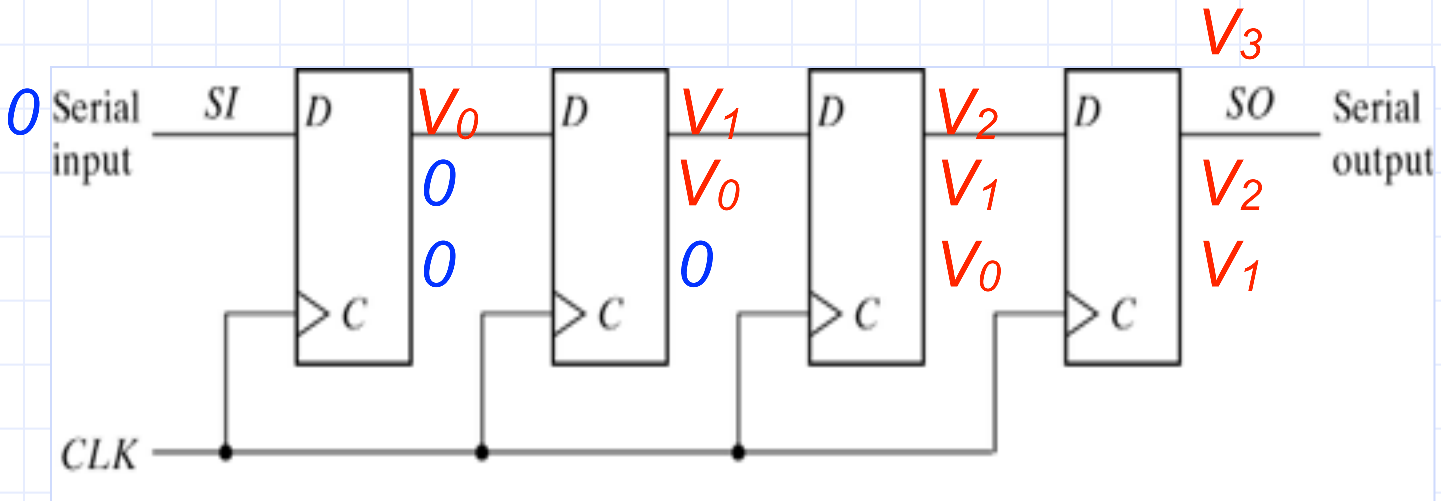
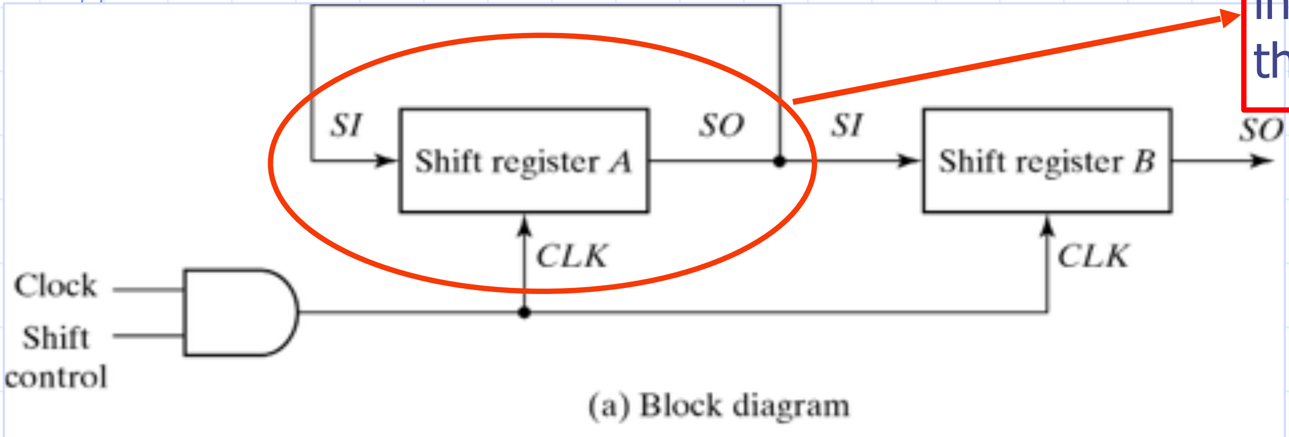


Fig. 6-3 4-Bit Shift Register

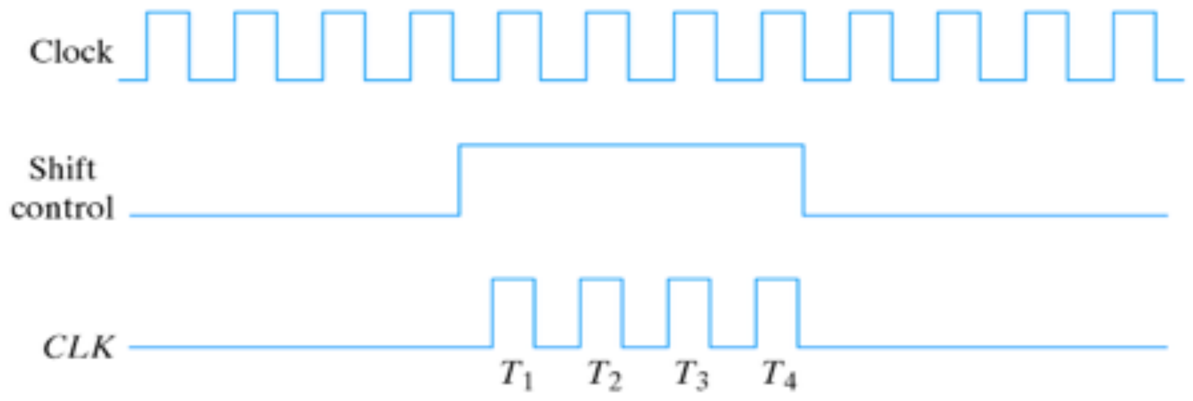
The simplest shift register

Serial Transfer

To prevent the loss of information stored in the source register



(a) Block diagram



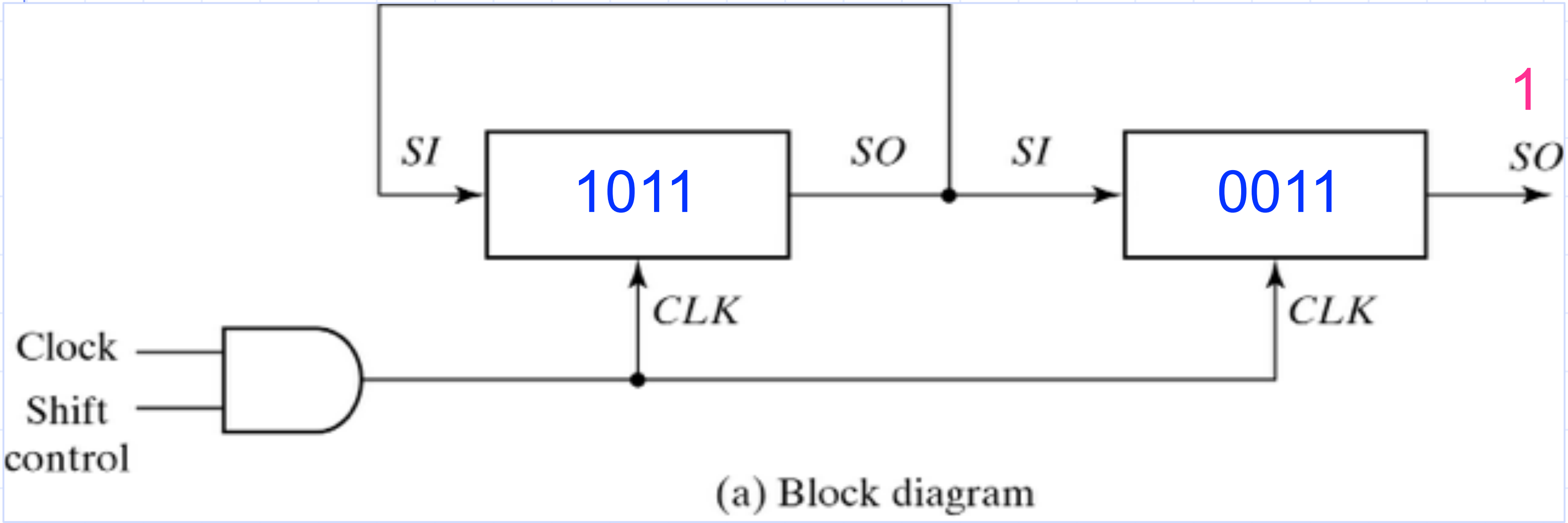
(b) Timing diagram

Fig. 6-4 Serial Transfer from Register *A* to register *B*

Serial-Transfer Example

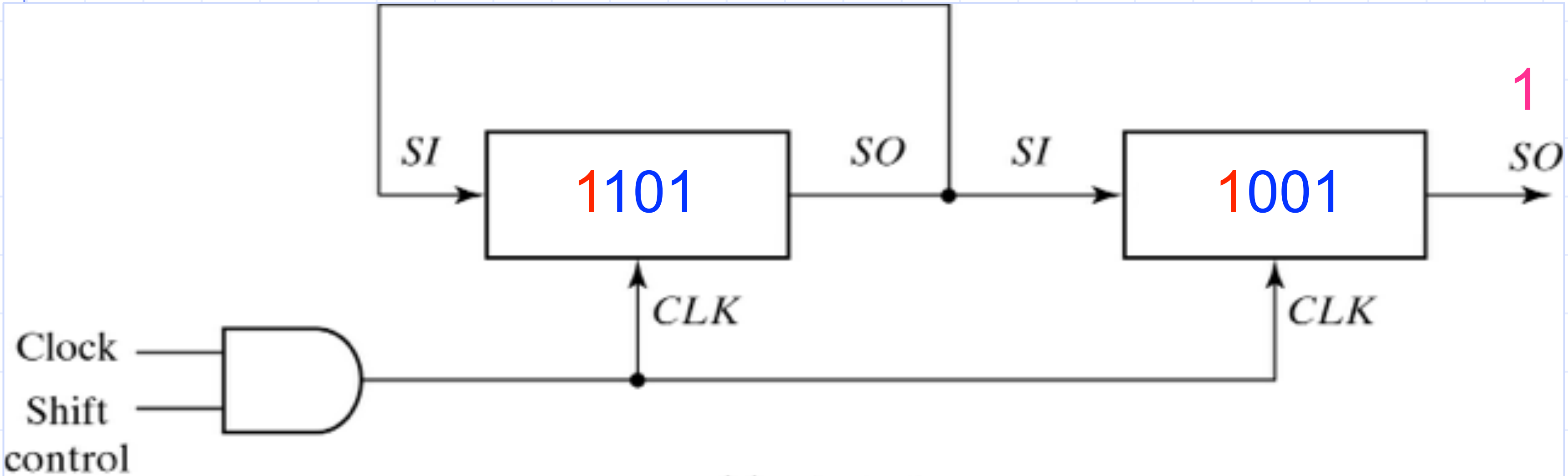
Timing pulse	Shift register A	Shift register B	Serial output of B
Initial value	1 0 1 1	0 0 1 0	0
After T_1	1 1 0 1	1 0 0 1	1
After T_2	1 1 1 0	1 1 0 0	0
After T_3	0 1 1 1	0 1 1 0	0
After T_4	1 0 1 1	1 0 1 1	1

Serial Transfer



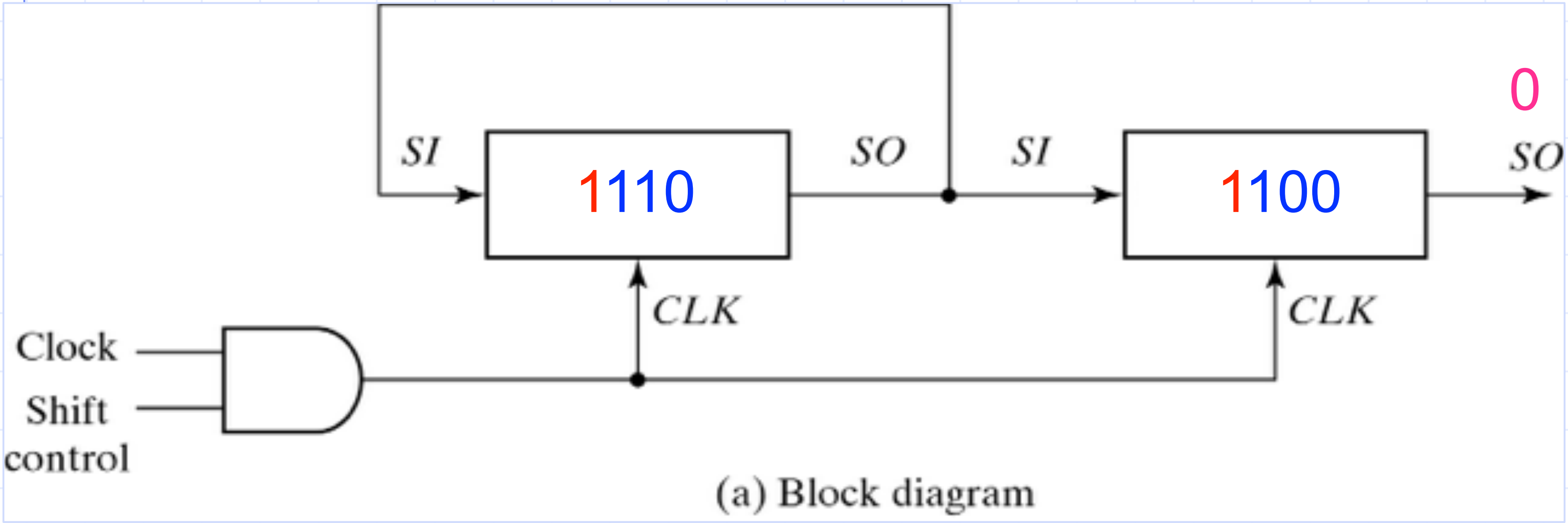
(a) Block diagram

Serial Transfer



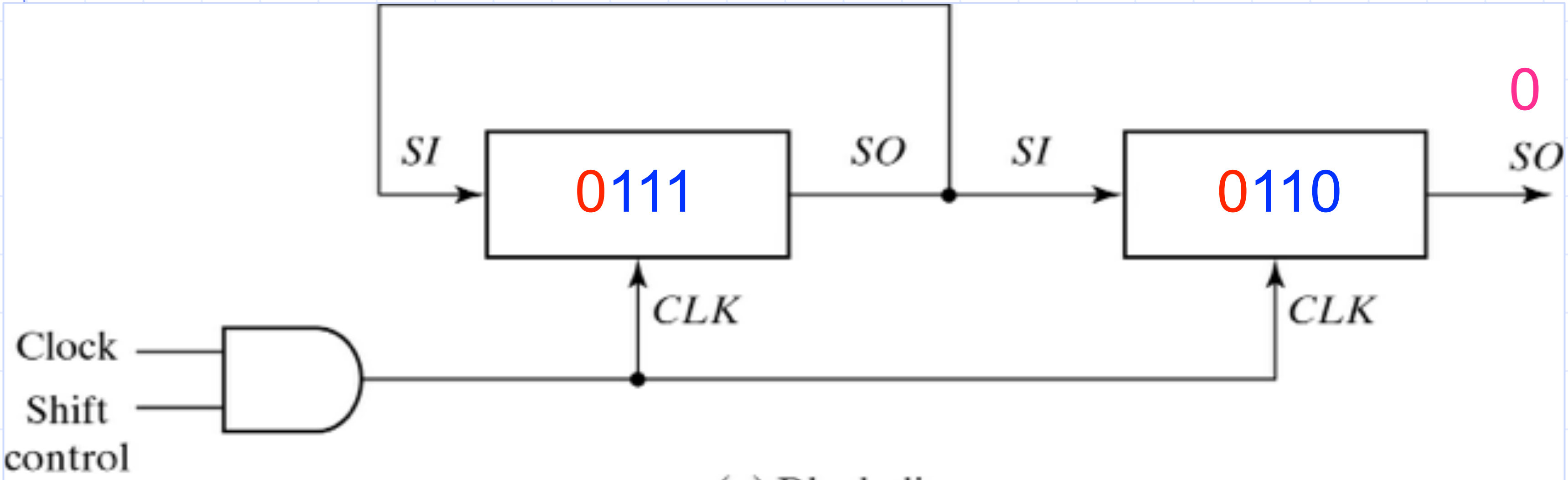
(a) Block diagram

Serial Transfer



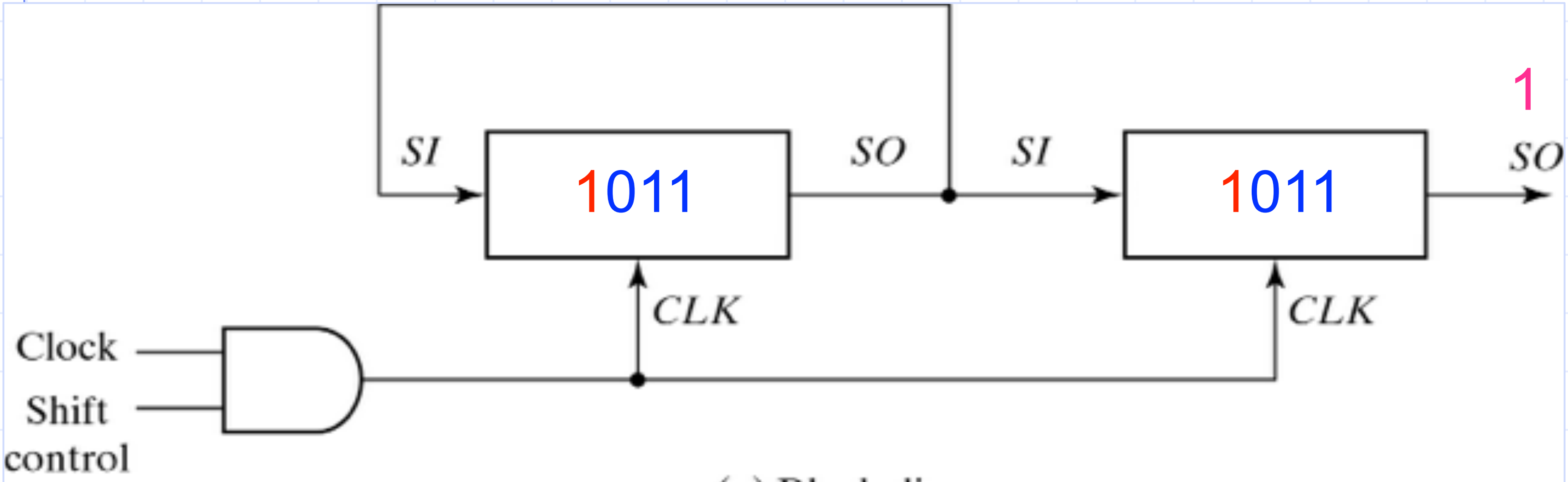
(a) Block diagram

Serial Transfer



(a) Block diagram

Serial Transfer



(a) Block diagram

Serial Addition

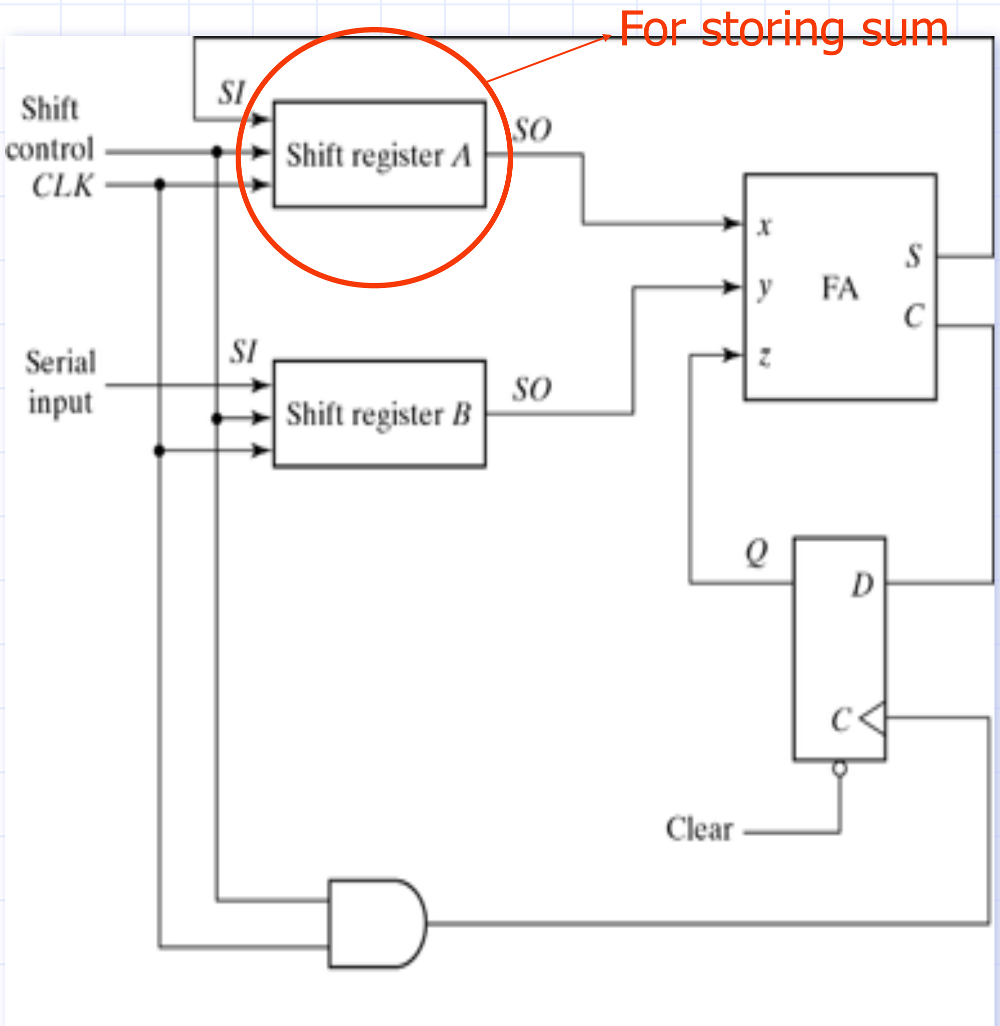


Fig. 6-5 Serial Adder

Operation

- calculate **A+B**
- The **SO** of **A** and **B** provide a pair of significant bits for the **FA**
- Output **Q** gives the input carry at **z**
- The shift-right control enables both registers and the carry flip-flop.
- The sum bit from **S** enters the leftmost flip-flop of **A**

Serial Addition

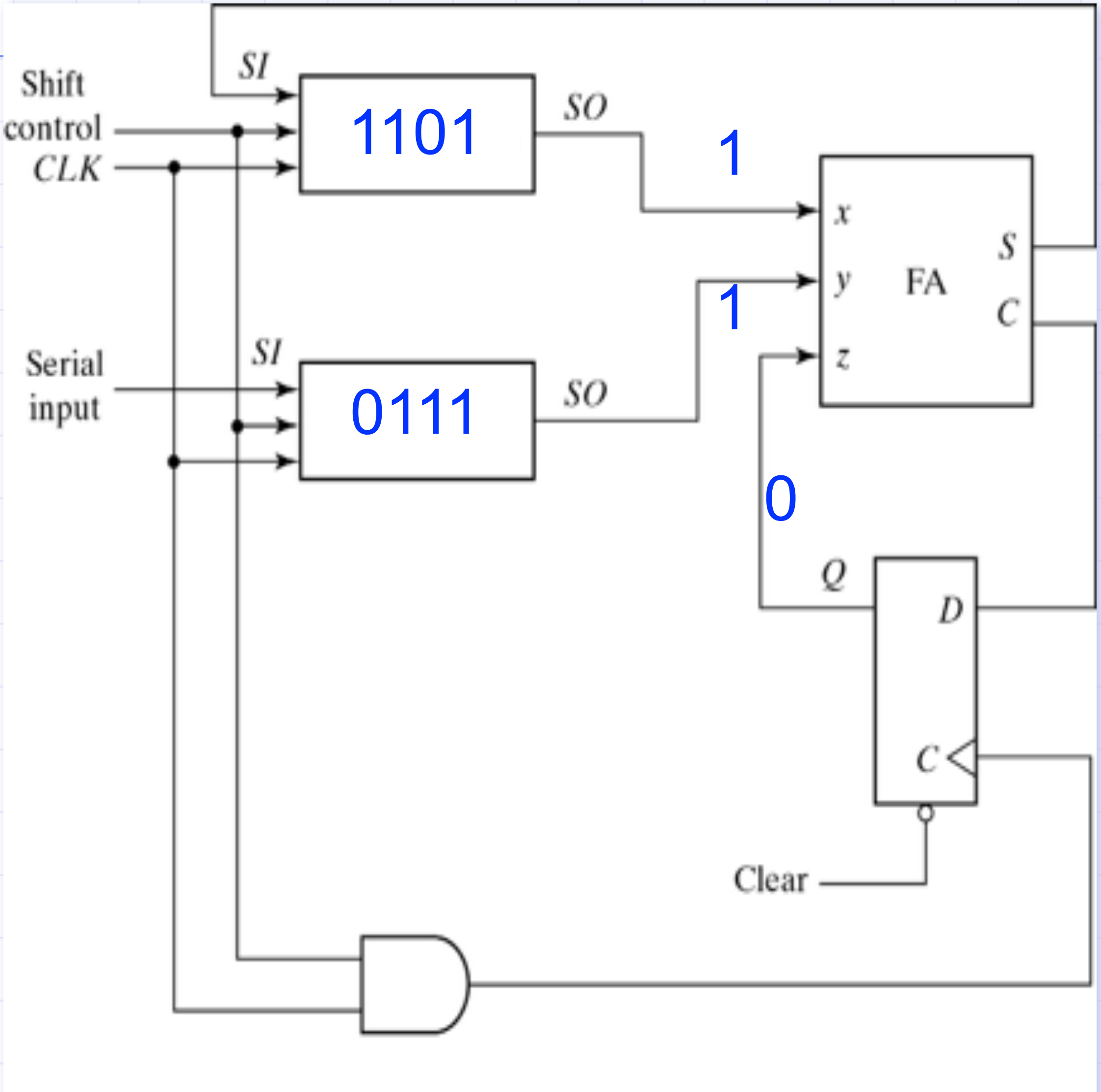


Fig. 6-5 Serial Adder

Serial Addition

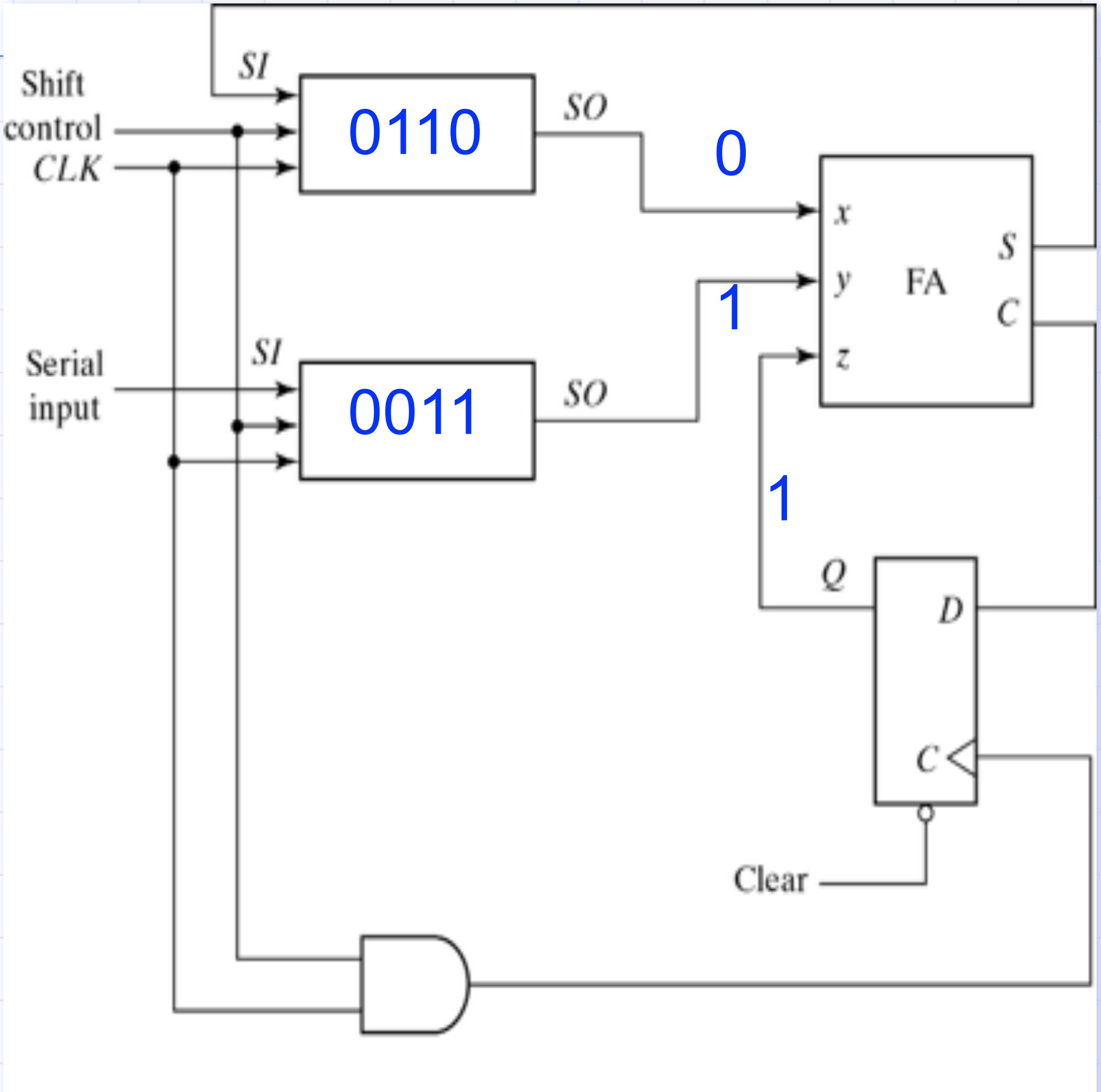


Fig. 6-5 Serial Adder

Serial Addition

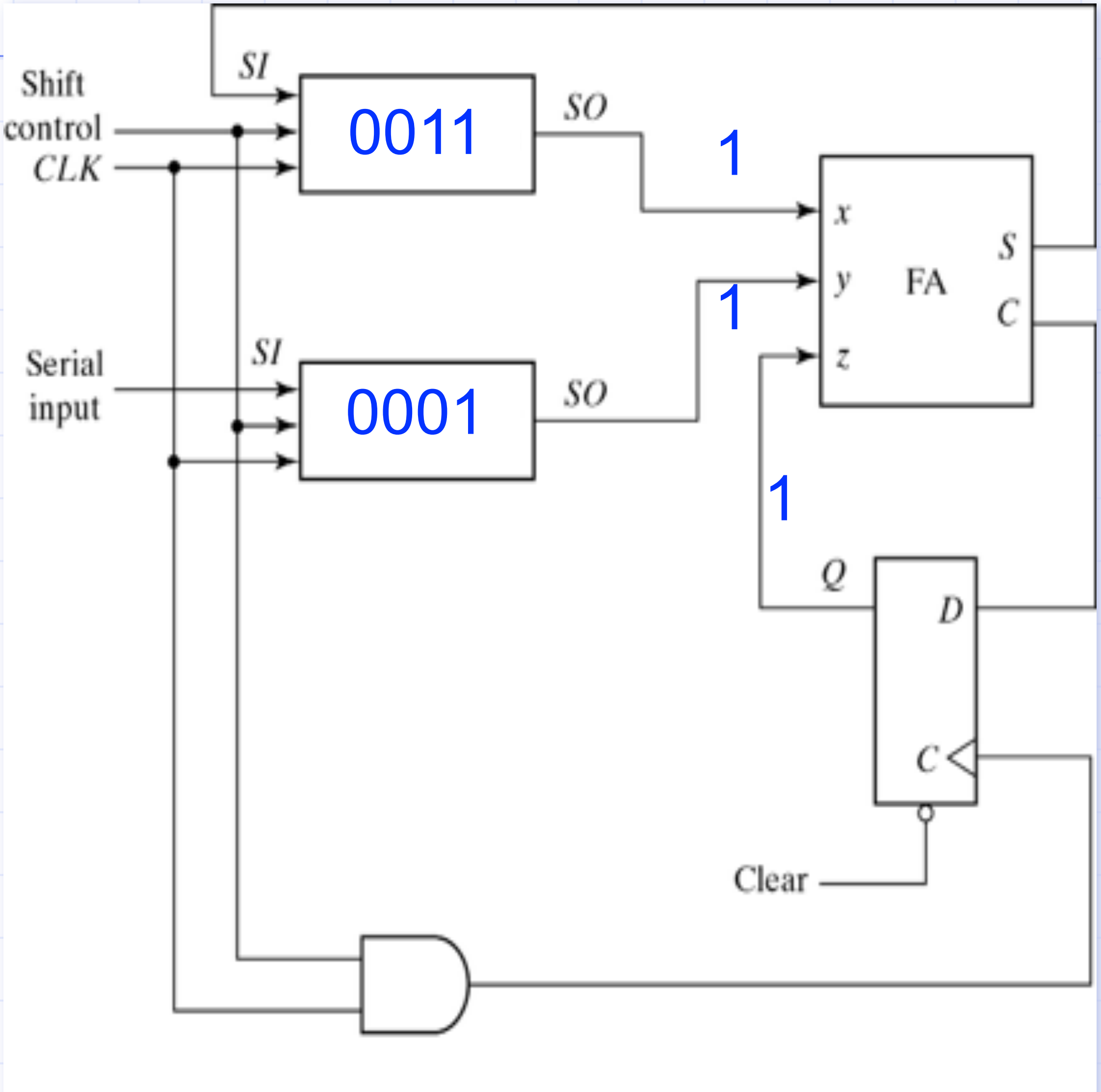


Fig. 6-5 Serial Adder

Serial Addition

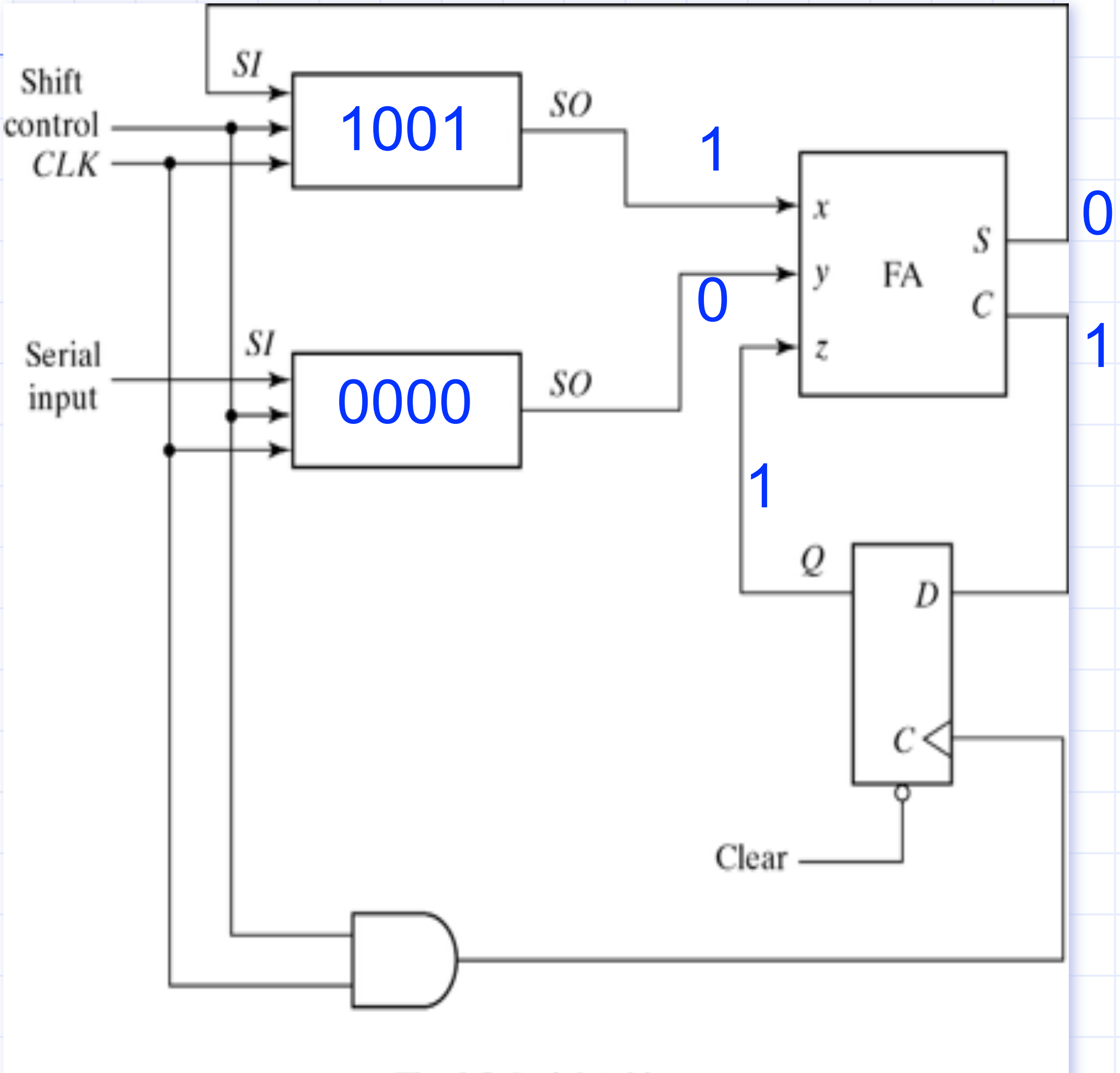


Fig. 6-5 Serial Adder

Serial Addition

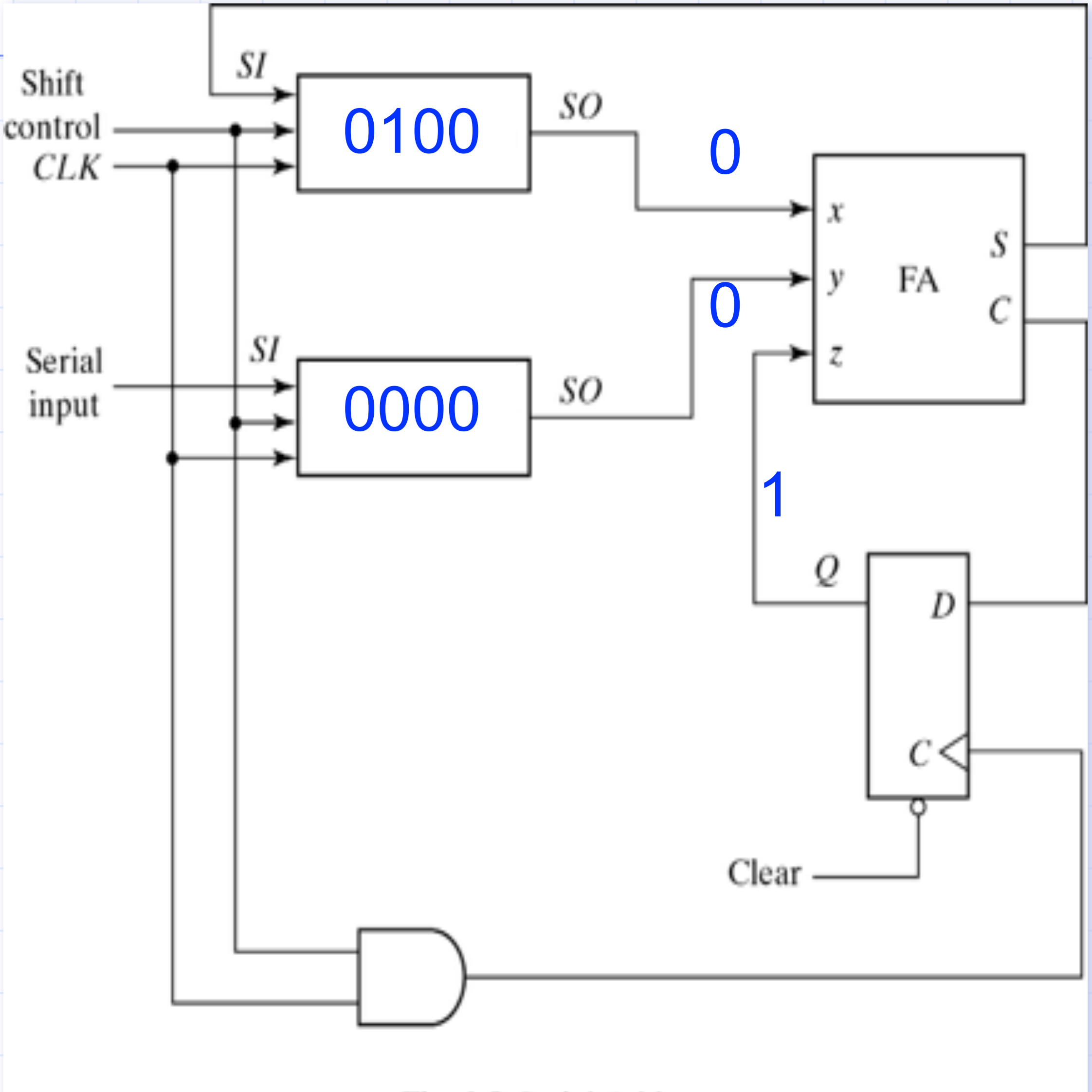


Fig. 6-5 Serial Adder

State Table for Serial Adder

Present value of carry

Output carry

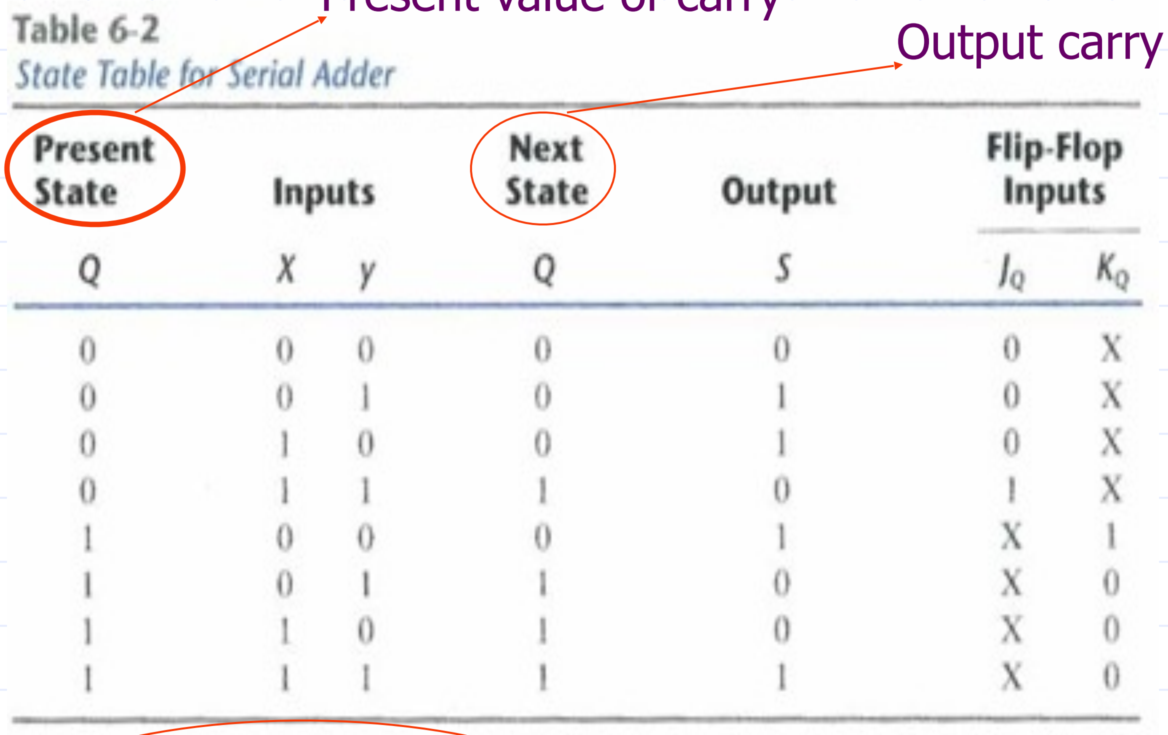


Table 6-2
State Table for Serial Adder

Present State	Inputs		Next State	Output	Flip-Flop Inputs	
Q	x	y	Q	S	J_Q	K_Q
0	0	0	0	0	0	X
0	0	1	0	1	0	X
0	1	0	0	1	0	X
0	1	1	1	0	1	X
1	0	0	0	1	X	1
1	0	1	1	0	X	0
1	1	0	1	0	X	0
1	1	1	1	1	X	0

$$J_Q = xy$$
$$K_Q = x'y' = (x+y)'$$
$$S = x \oplus y \oplus Q$$

By k-map

Second form of Serial Adder

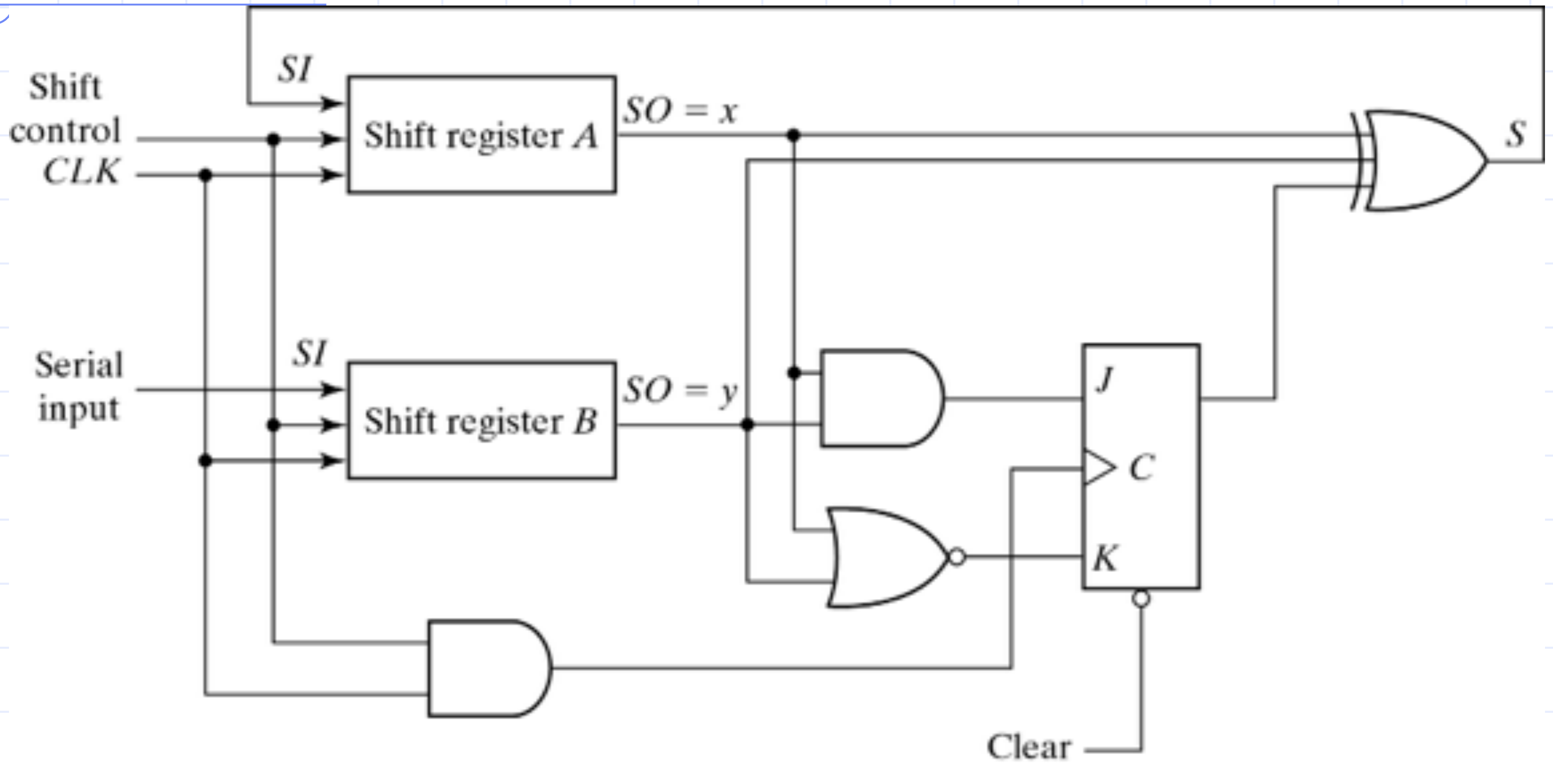


Fig. 6-6 Second form of Serial Adder

Universal Shift Register

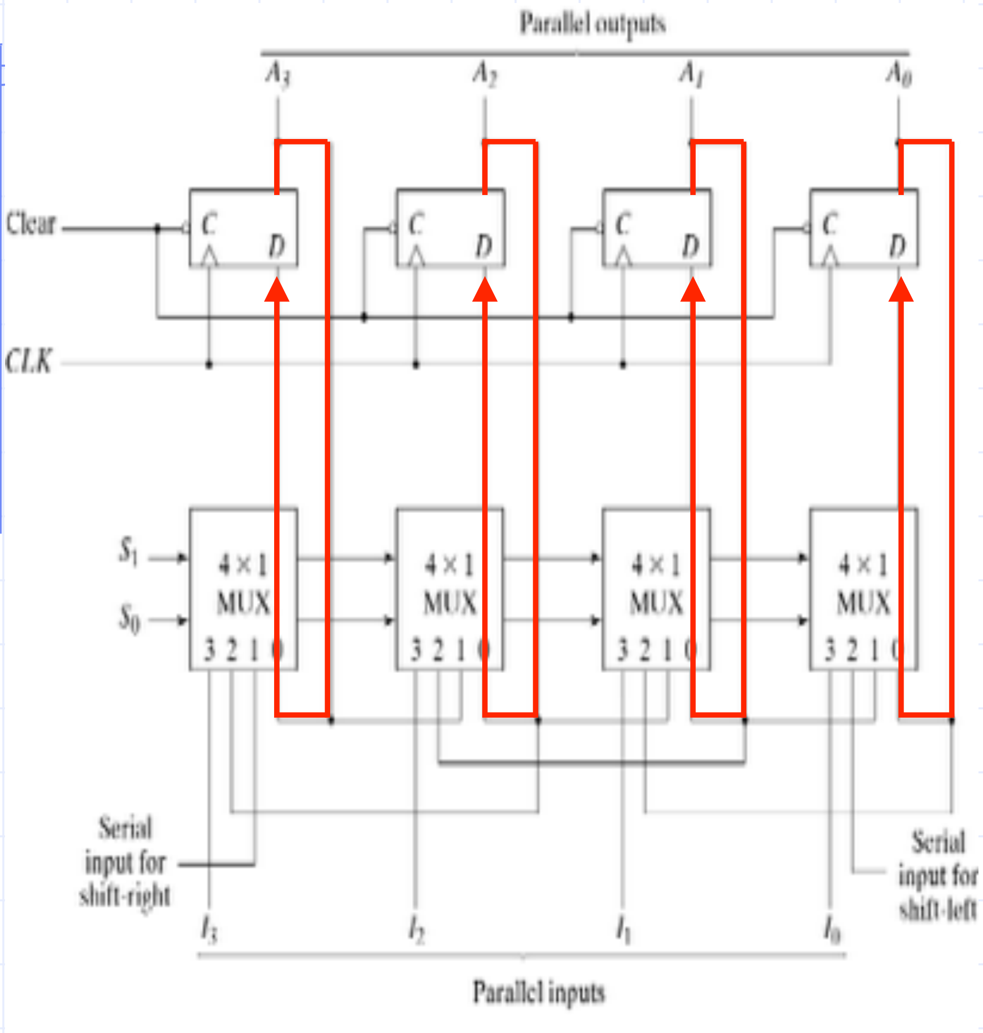


Fig. 6-7 4-Bit Universal Shift Register

- **S_1, S_0** -> 0, 0 **;No change**

Universal Shift Register

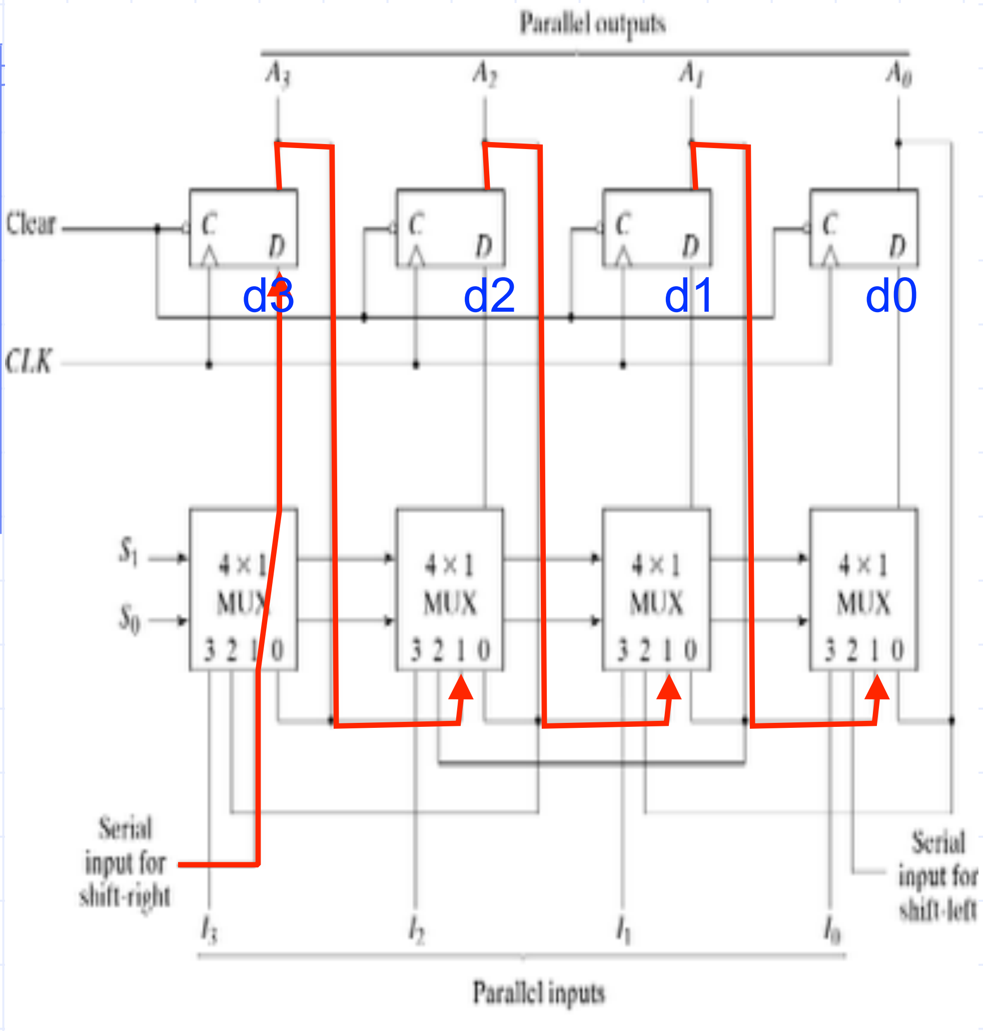


Fig. 6-7 4-Bit Universal Shift Register

- **S_1, S_0** -> 0, 1 **;Shift right**

Universal Shift Register

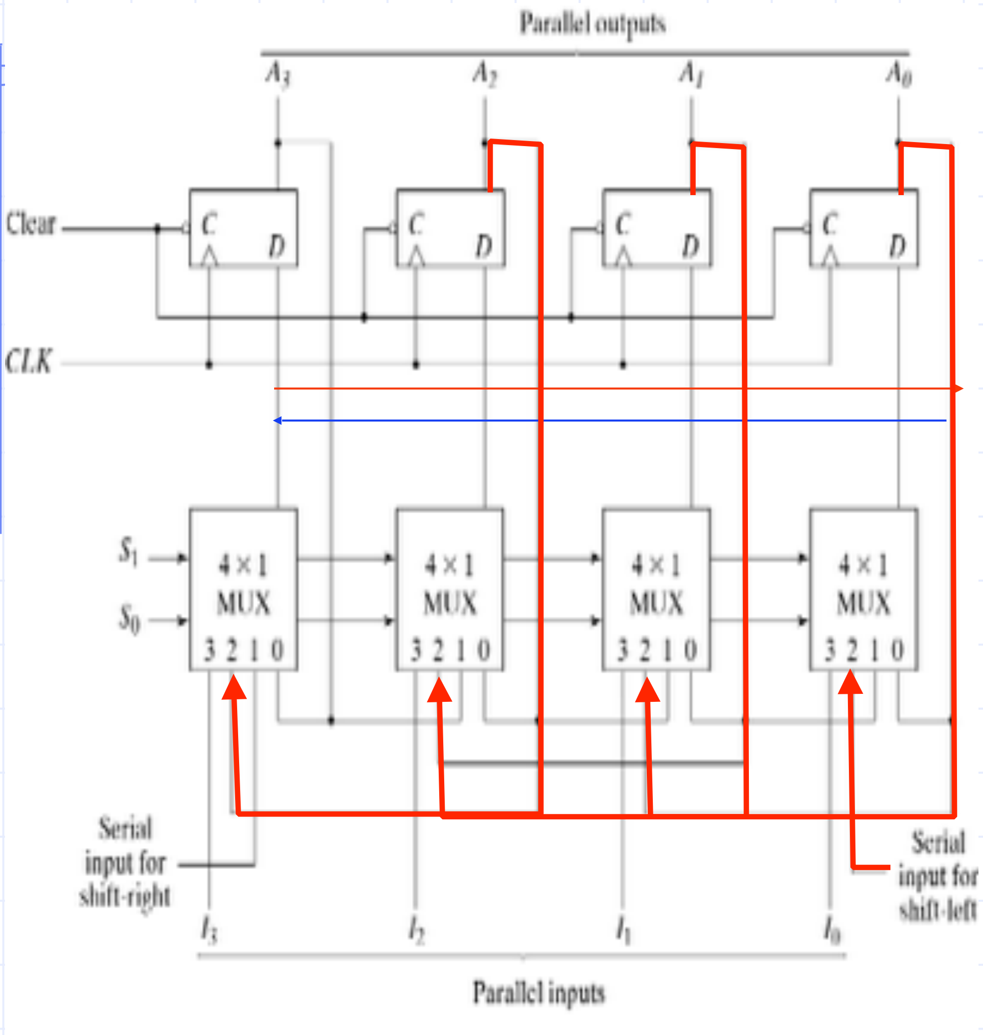


Fig. 6-7 4-Bit Universal Shift Register

- S_1, S_0 -> 1, 0 ;**Shift left**

Universal Shift Register

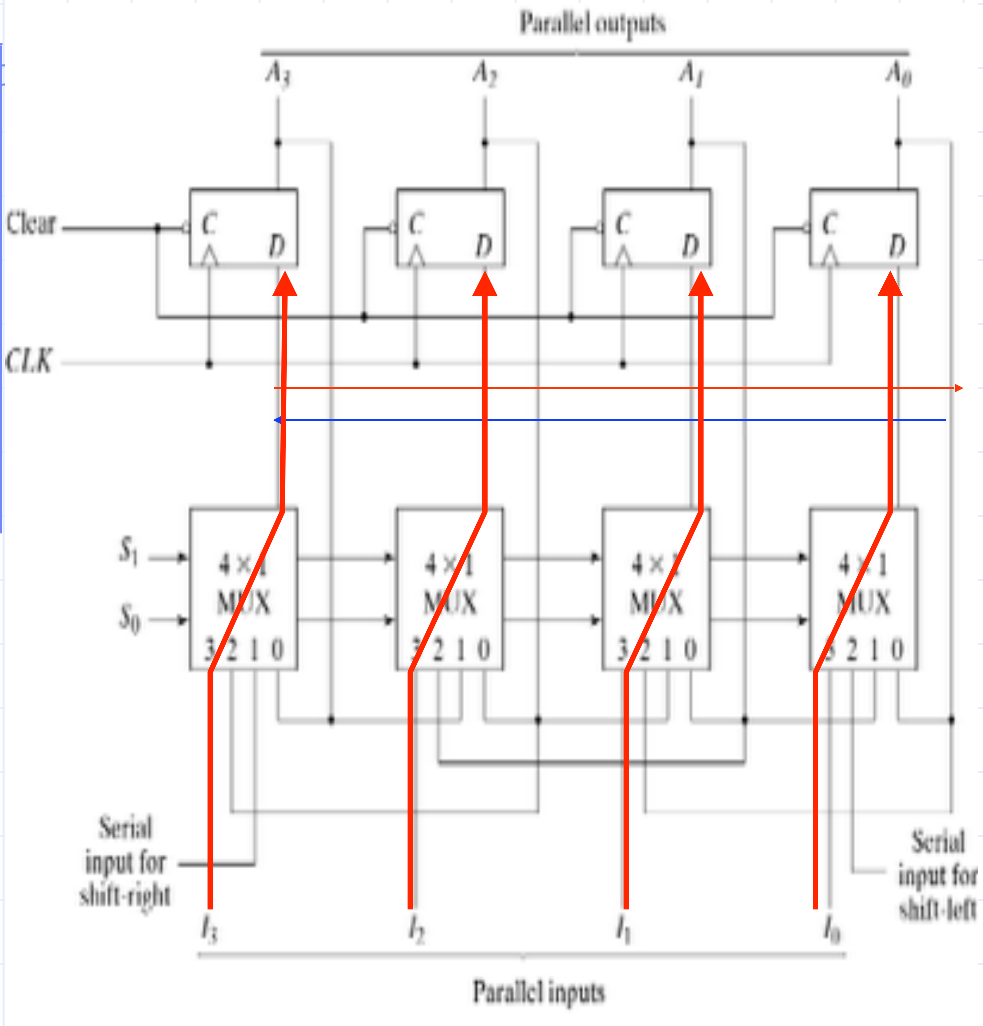
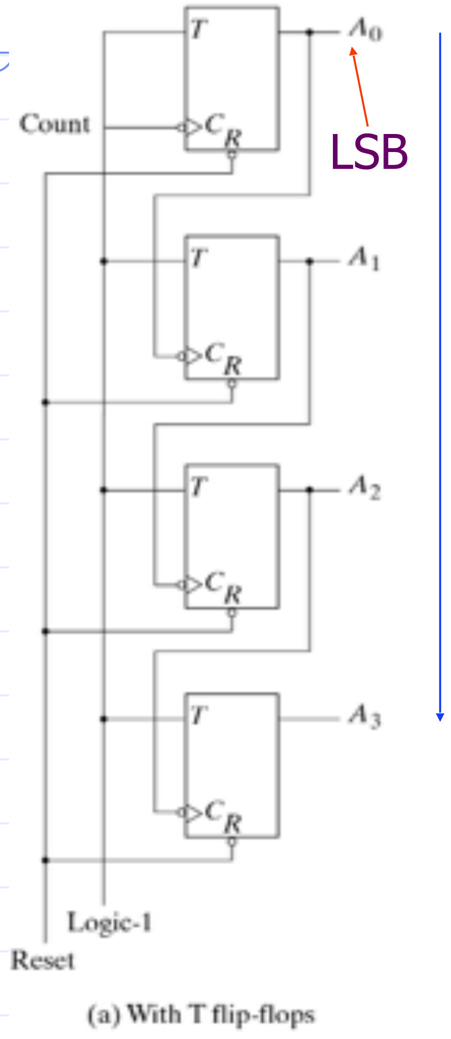


Fig. 6-7 4-Bit Universal Shift Register

- S_1, S_0 -> 1, 1 **;Parallel load**

6-3 RIPPLE COUNTERS



(a) With T flip-flops

0000

Count: 1 -> 0

0001

Count: 1 -> 0, A_0: 1->0

0010

Count: 1 -> 0

0011

Count: 1 -> 0, A_0: 1->0, A_1: 1->0

0100

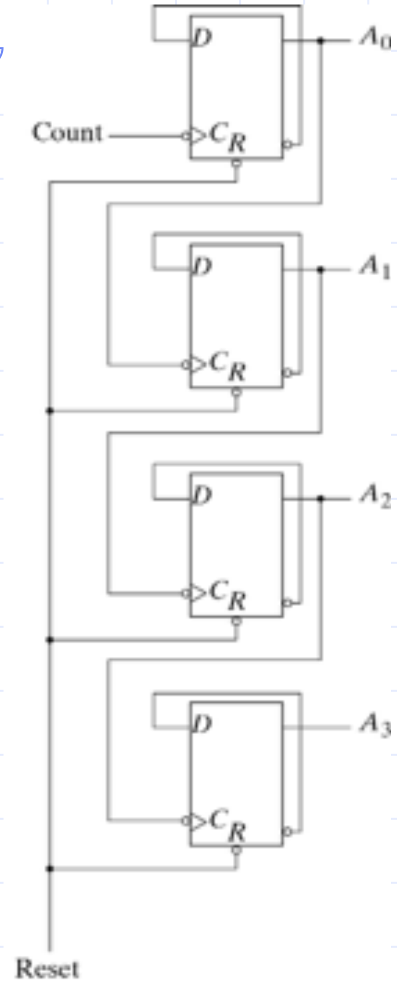
…

1111

Count: 1 -> 0,
A_0: 1->0, A_1: 1->0,
A_2: 1->0, A_3: 1->0

0000

6-3 RIPPLE COUNTERS



(b) With D flip-flops

0000

Count: 1 -> 0

0001

Count: 1 -> 0, A_0: 1->0

0010

Count: 1 -> 0

0011

Count: 1 -> 0, A_0: 1->0, A_1: 1->0

0100

…

1111

Count: 1 -> 0,
A_0: 1->0, A_1: 1->0,
A_2: 1->0, A_3: 1->0

0000

Count sequence for a binary Counter

Count sequence A_3 A_2 A_1 A_0		Conditions for Complementing
0 0 0 0	Complement A_0	
0 0 0 1	Complement A_0	A_0 will go from 1 to 0 and complement A_1
0 0 1 0	Complement A_0	
0 0 1 1	Complement A_0	A_0 will go from 1 to 0 and complement A_1 ; A_1 will go from 1 to 0 and complement A_2
0 1 0 0	Complement A_0	
0 1 0 1	Complement A_0	A_0 will go from 1 to 0 and complement A_1
0 1 1 0	Complement A_0	
0 1 1 1	Complement A_0	
………………		
1 0 0 0	and so on…	

BCD Ripple Counter

Count: 1 -> 0

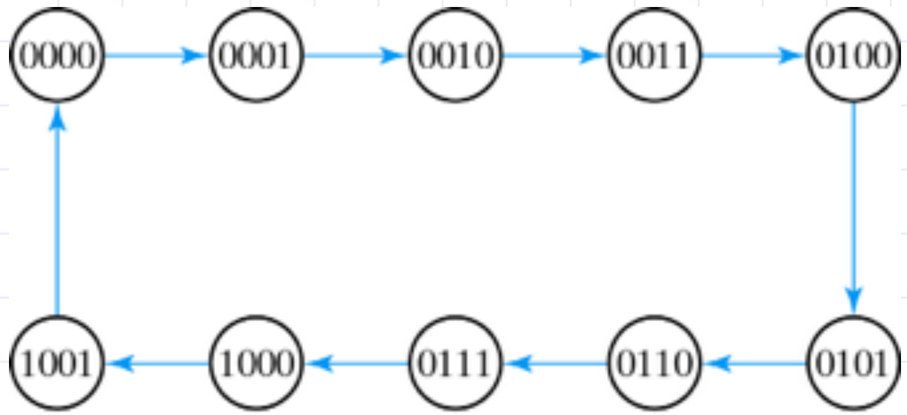


Fig. 6-9 State Diagram of a Decimal BCD-Counter

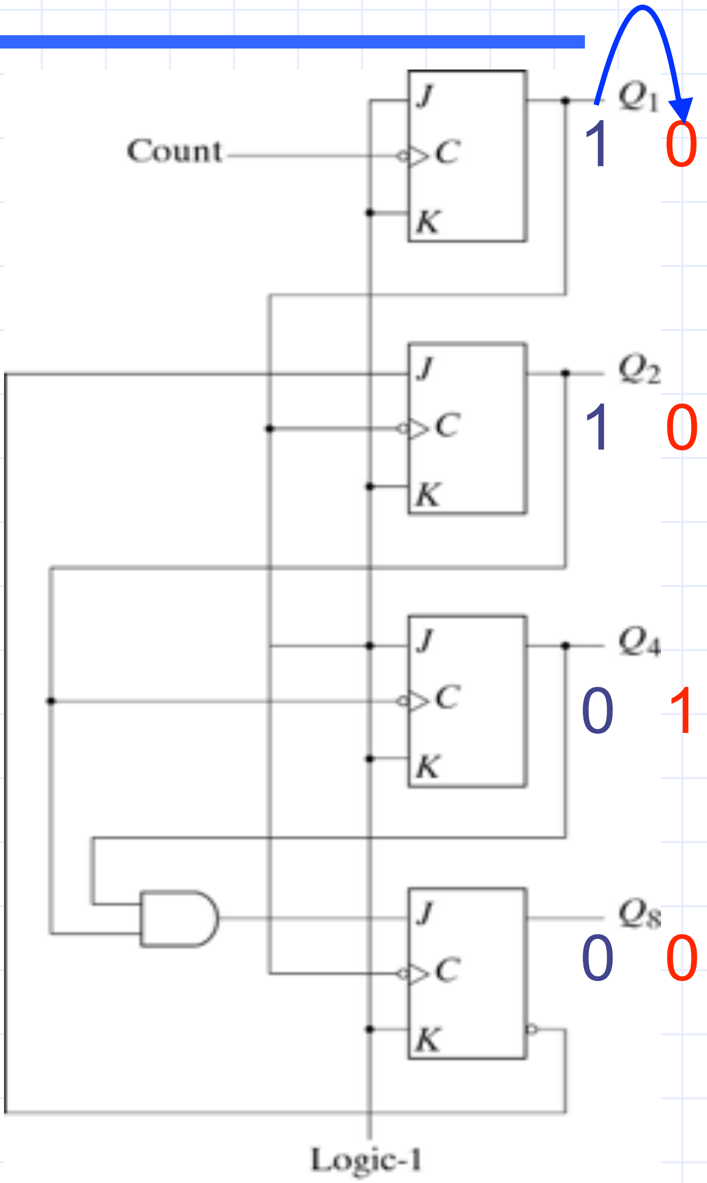


Fig. 6-10 BCD Ripple Counter

BCD Ripple Counter

Count: 1 -> 0

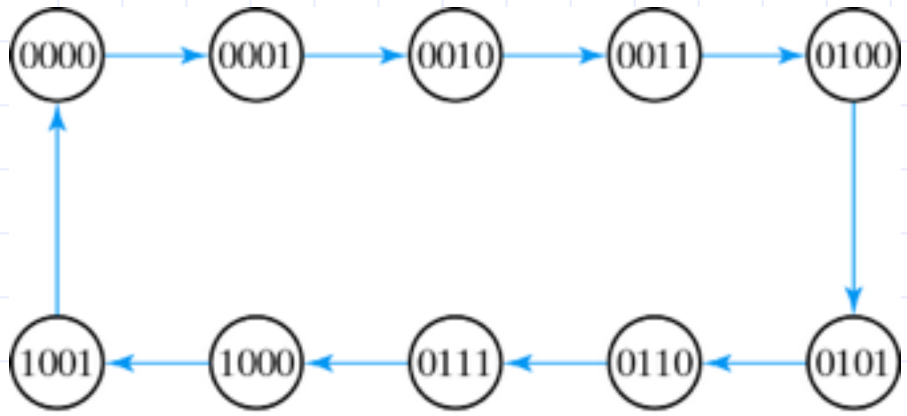


Fig. 6-9 State Diagram of a Decimal BCD-Counter

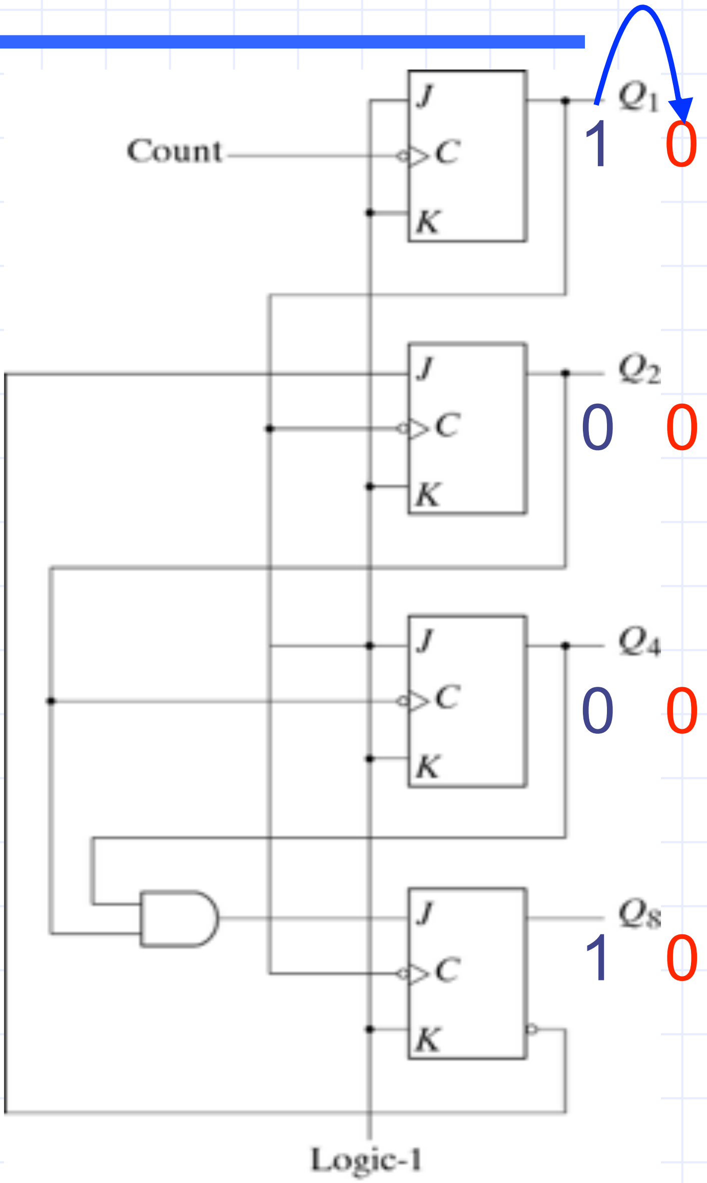


Fig. 6-10 BCD Ripple Counter

BCD Ripple Counter

operation

1. Q_1 is complemented on the negative edge of every count pulse.

2. Q_2 is complemented if $Q_8=0$ and Q_1 goes from 1 to 0. Q_2 is cleared if $Q_8=1$ and Q_1 goes from 1 to 0.

3. Q_4 is complemented when Q_2 goes from 1 to 0.

4 .Q_8 is complemented when $Q_4Q_2=11$ and Q_1 goes from 1 to 0. Q_8 is cleared if either Q_4 or Q_2 is 0 and Q_1 goes from 1 to 0

Three-Decade Decimal BCD Counter

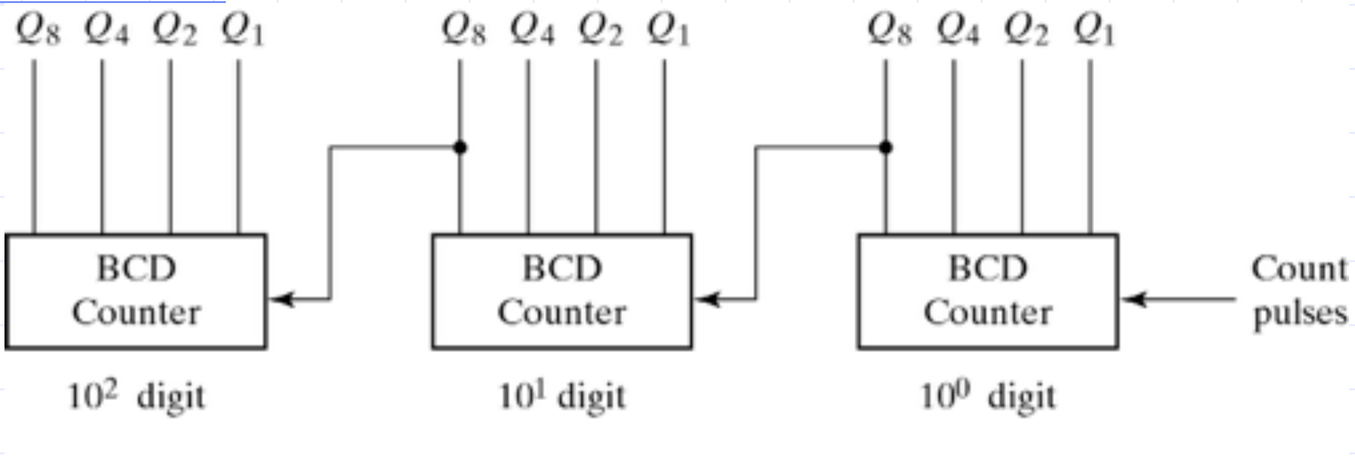


Fig. 6-11 Block Diagram of a Three-Decade Decimal BCD Counter

❑To count from **0** to **999**, We need a three-decade counter.

6-4 SYNCHRONOUS COUNTERS

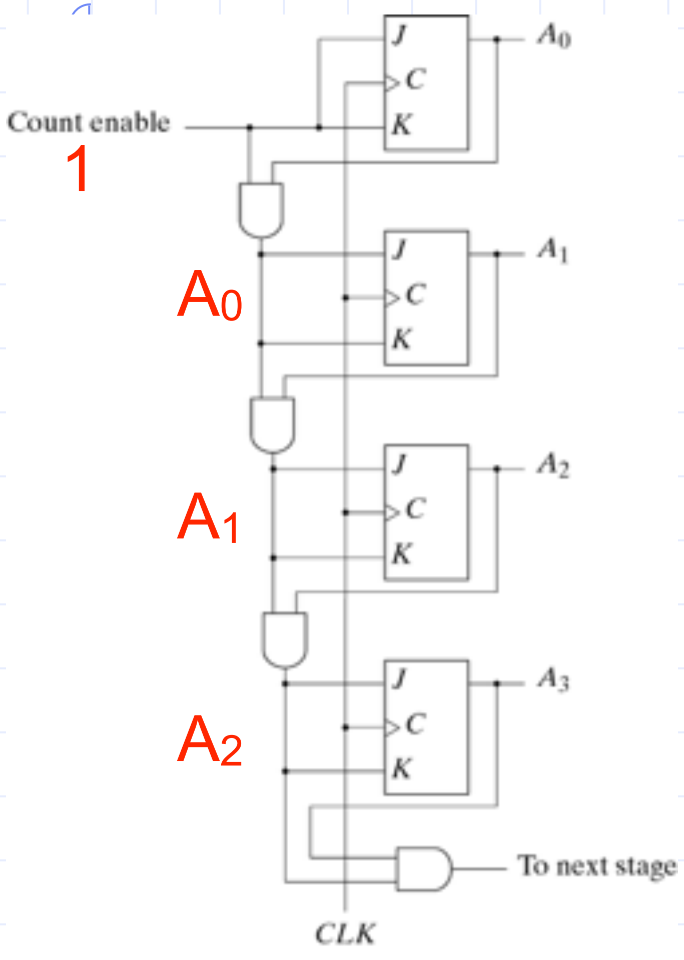


Fig. 6-12 4-Bit Synchronous Binary Counter

- ❑ The first stage **A_0** has its **J** and **K** equal to **1** if the counter is enabled .
- ❑The other **J** and **K** inputs are equal to **1** if all previous low-order bits are equal to **1** and the count is enabled.
- ❑ **A_0** = 1이면, 클럭의 상승에지가 발생할 때 마다, **A_1** 의 값은 반전된다.

Up-Down Binary Counter

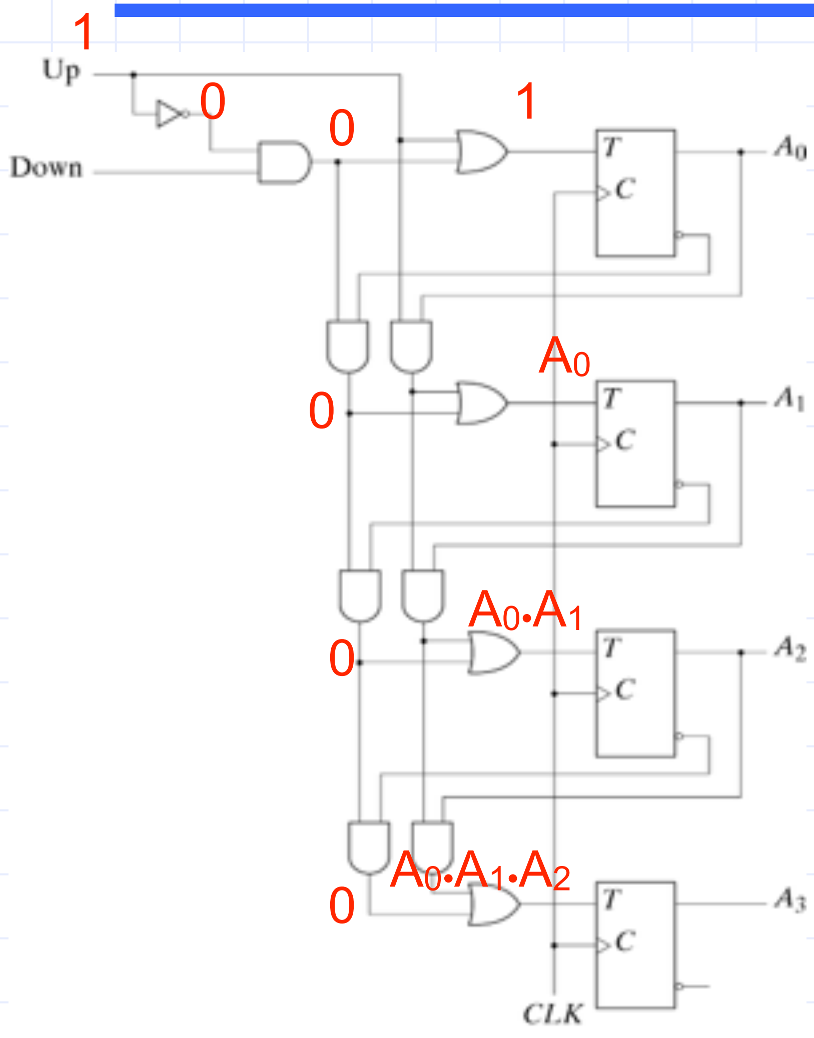


Fig. 6-13 4-Bit Up-Down Binary Counter

- **Up** input control=**1** ;count up (the **T** inputs receive their signals from the values of the previous normal outputs of the flip-flops.)

$$\begin{tikzcd}
0000 \arrow[d, bend left, "\text{CLK: 0->1}"] \\
0001 \arrow[d, bend left, "\text{CLK: 0->1}"] \\
0010 \arrow[d, bend left, "\text{CLK: 0->1}"] \\
0011 \arrow[d, bend left, "\text{CLK: 0->1}"] \\
0100
\end{tikzcd}$$

Up-Down Binary Counter

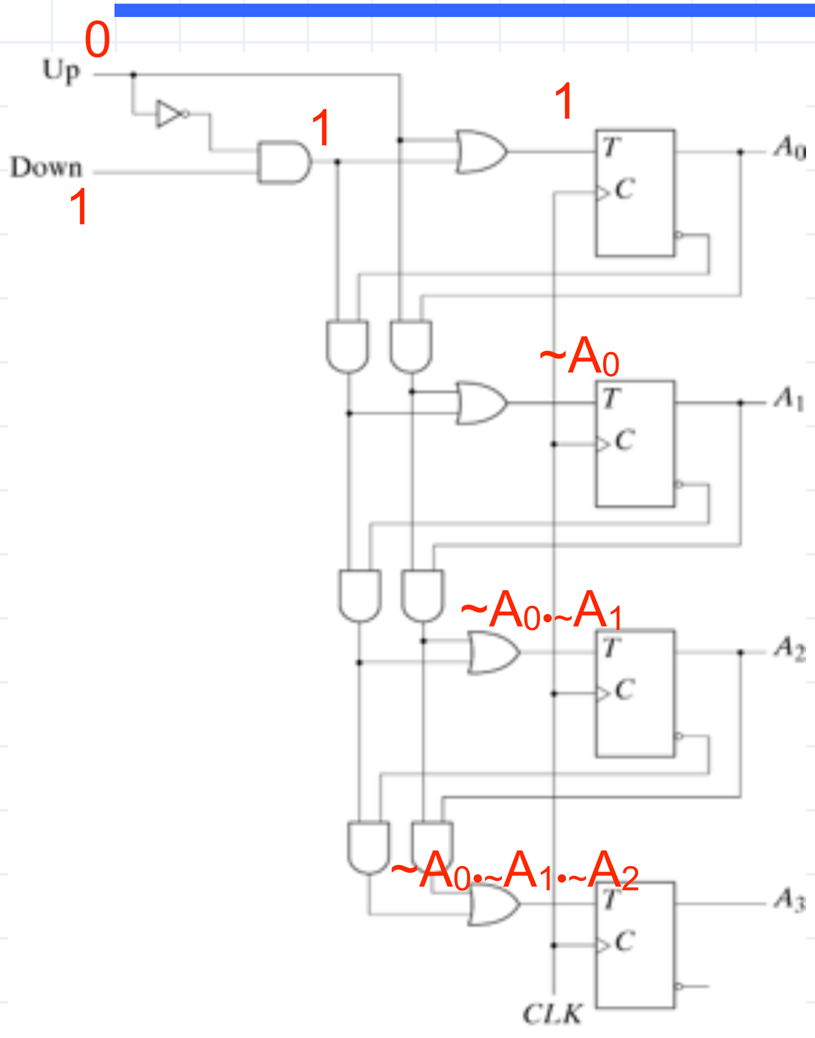


Fig. 6-13 4-Bit Up-Down Binary Counter

- **Down** input control=**1**, **up** input control=**0** ; count down

$$\begin{tikzcd}
1\,1\,1\,1 \arrow[d, bend left, "\text{CLK: 0->1}"] \\
1\,1\,1\,0 \arrow[d, bend left, "\text{CLK: 0->1}"] \\
1\,1\,0\,1 \arrow[d, bend left, "\text{CLK: 0->1}"] \\
1\,1\,0\,0 \arrow[d, bend left, "\text{CLK: 0->1}"] \\
1\,0\,1\,1
\end{tikzcd}$$

Up-Down Binary Counter

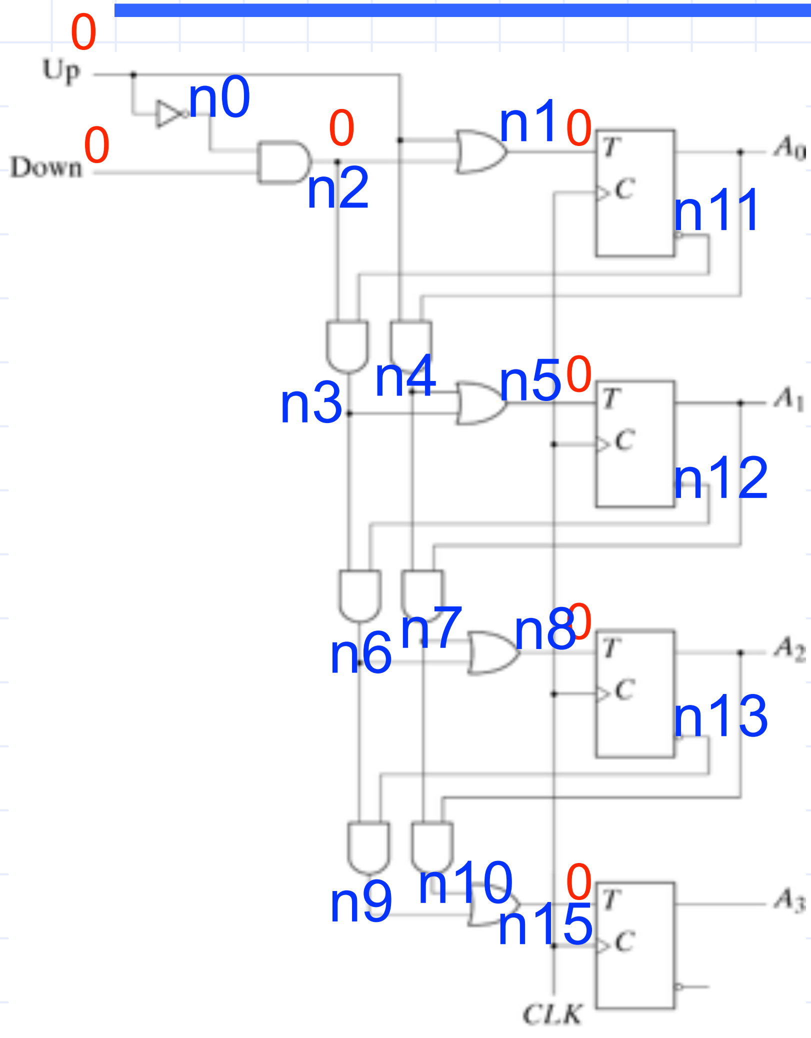


Fig. 6-13 4-Bit Up-Down Binary Counter

- **Up**=**down**=0 ;unchanged state

Up-Down Binary Counter

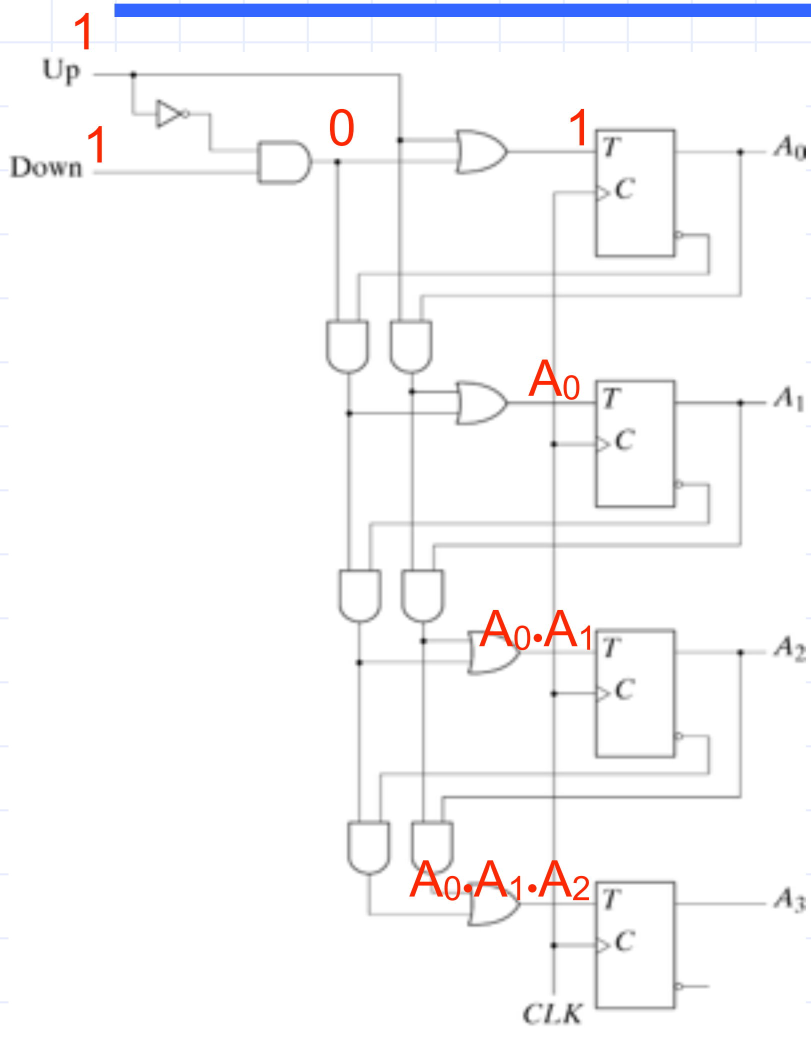


Fig. 6-13 4-Bit Up-Down Binary Counter

- **Up**=**down**=1 ;count up

BCD Counter

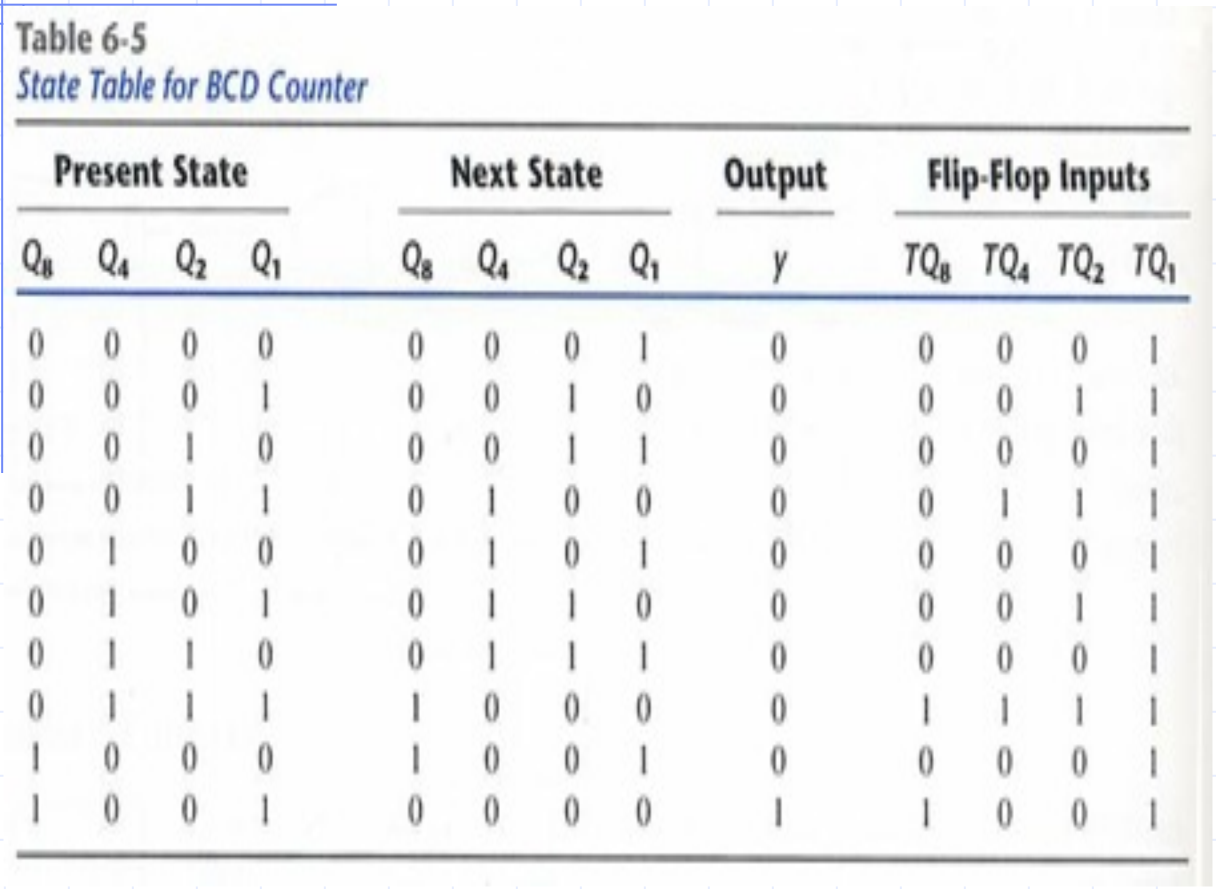


Table 6-5
State Table for BCD Counter

Present State				Next State				Output	Flip-Flop Inputs			
Q_8	Q_4	Q_2	Q_1	Q_8	Q_4	Q_2	Q_1	y	TQ_8	TQ_4	TQ_2	TQ_1
0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	1	0	0	0	0	1	1
0	0	1	0	0	0	1	1	0	0	0	0	1
0	0	1	1	0	1	0	0	0	0	1	1	1
0	1	0	0	0	1	0	1	0	0	0	0	1
0	1	0	1	0	1	1	0	0	0	0	1	1
0	1	1	0	0	1	1	1	0	0	0	0	1
0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	0	0	1	0	0	0	0	1
1	0	0	1	0	0	0	0	1	1	0	0	1

$$T_{Q1} = 1$$
$$T_{Q2} = Q'_8 Q_1$$
$$T_{Q4} = Q_2 Q_1$$
$$T_{Q8} = Q_8 Q_1 + Q_4 Q_2 Q_1$$
$$y = Q_8 Q_1$$

Binary Counter with Parallel Load

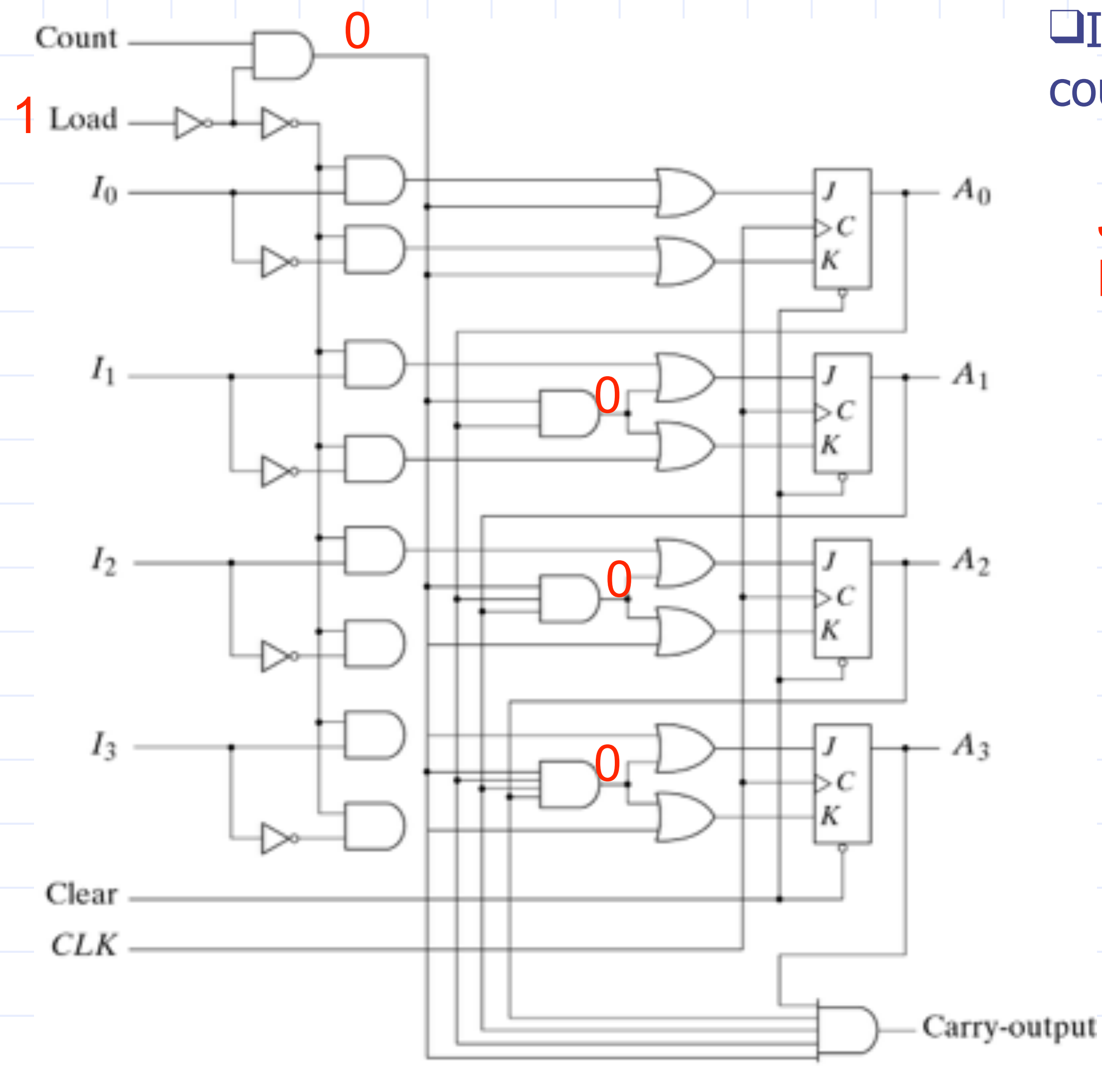


Fig. 6-14 4-Bit Binary Counter with Parallel Load

- Input **load** control=1 ; disables the count sequence ,data transfer

$J_i = I_i$

$K_i = \sim I_i$

$A_i = I_i$

Binary Counter with Parallel Load

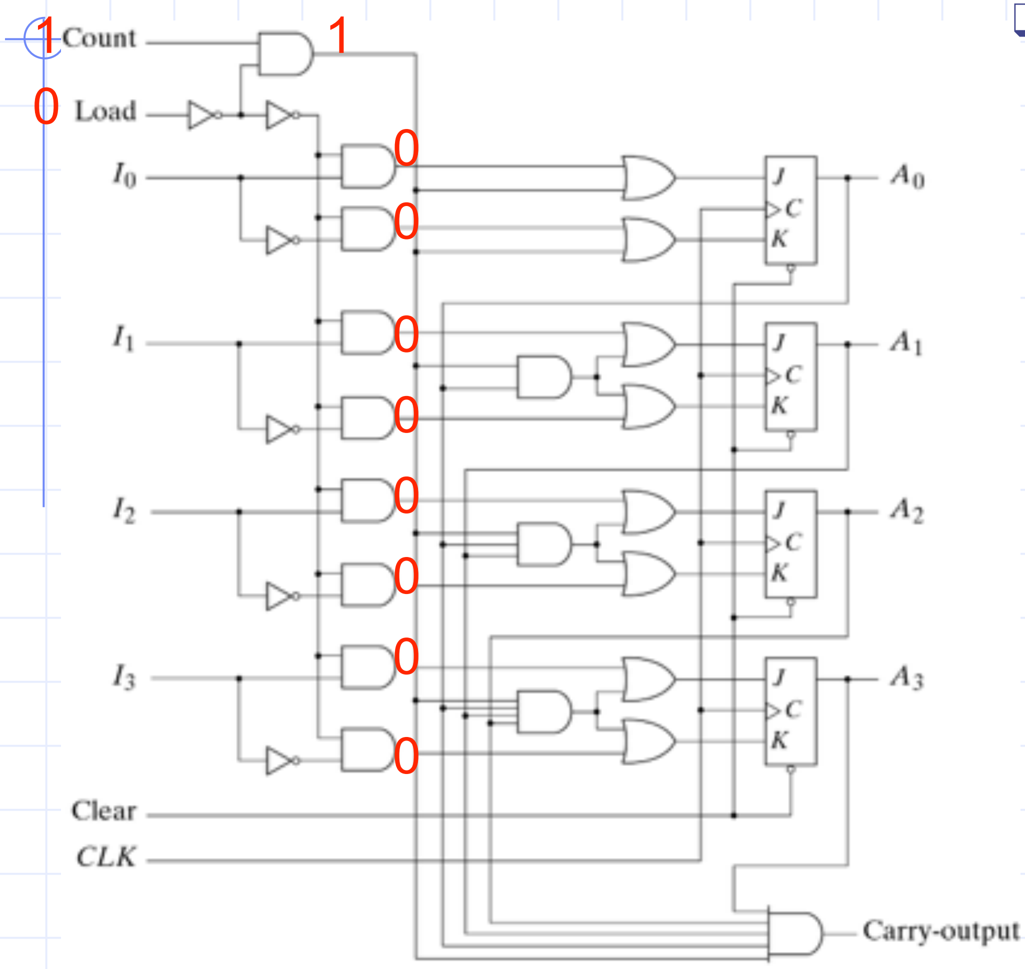


Fig. 6-14 4-Bit Binary Counter with Parallel Load

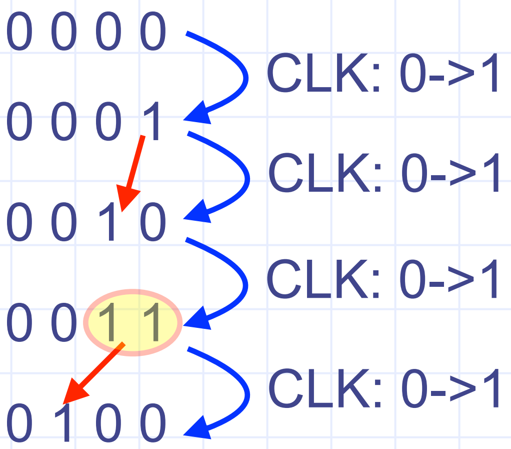
- **Load** =0 and **count**=1 ;count

$J_0=1$
$K_0 = 1$

$J_1=A_0$
$K_1 = A_0$

$J_2=A_0 \bullet A_1$
$K_2 = A_0 \bullet A_1$

$J_3=A_0 \bullet A_1 \bullet A_2$
$K_3 = A_0 \bullet A_1 \bullet A_2$



Binary Counter with Parallel Load

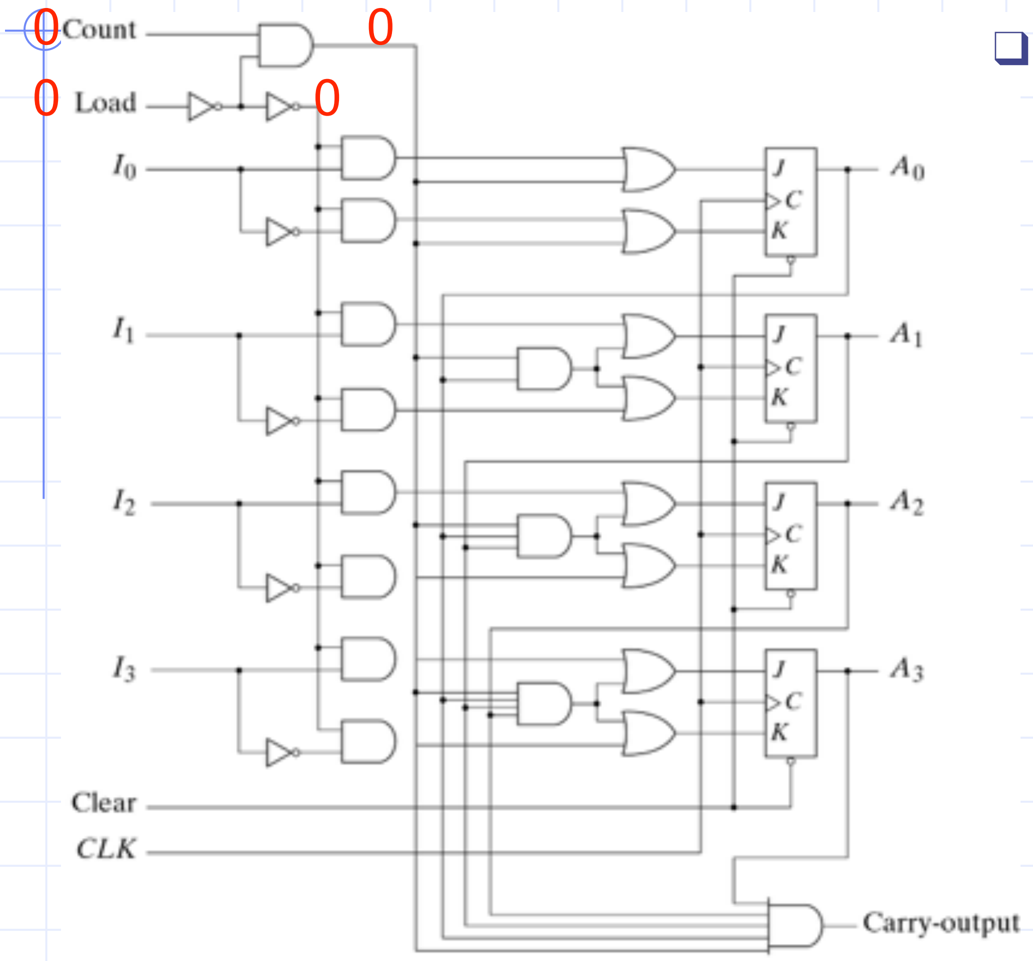


Fig. 6-14 4-Bit Binary Counter with Parallel Load

- **Load**=0 and **count**=0 ;unchanged

$J_i = 0$
$K_i = 0$

Binary Counter with Parallel Load

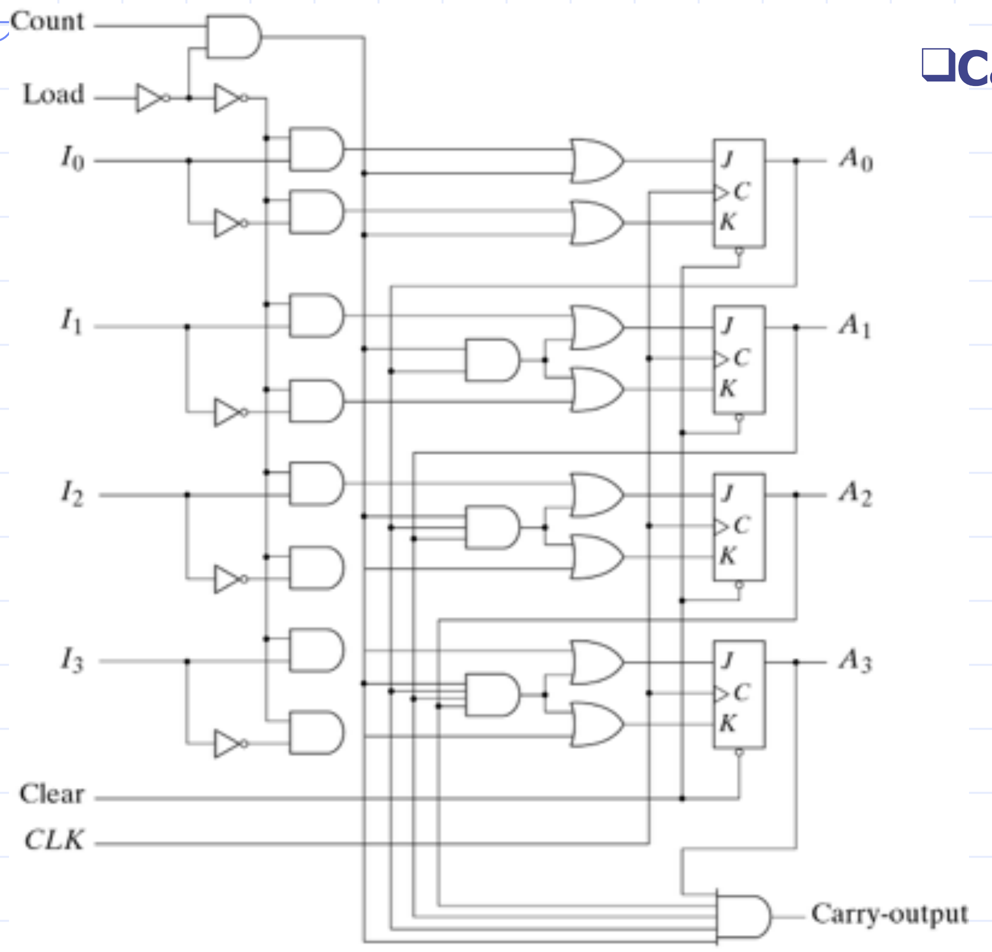


Fig. 6-14 4-Bit Binary Counter with Parallel Load

- **Carry out**=1(all flip-flop=1)

BCD COUNTER using Binary Counter with Parallel Load

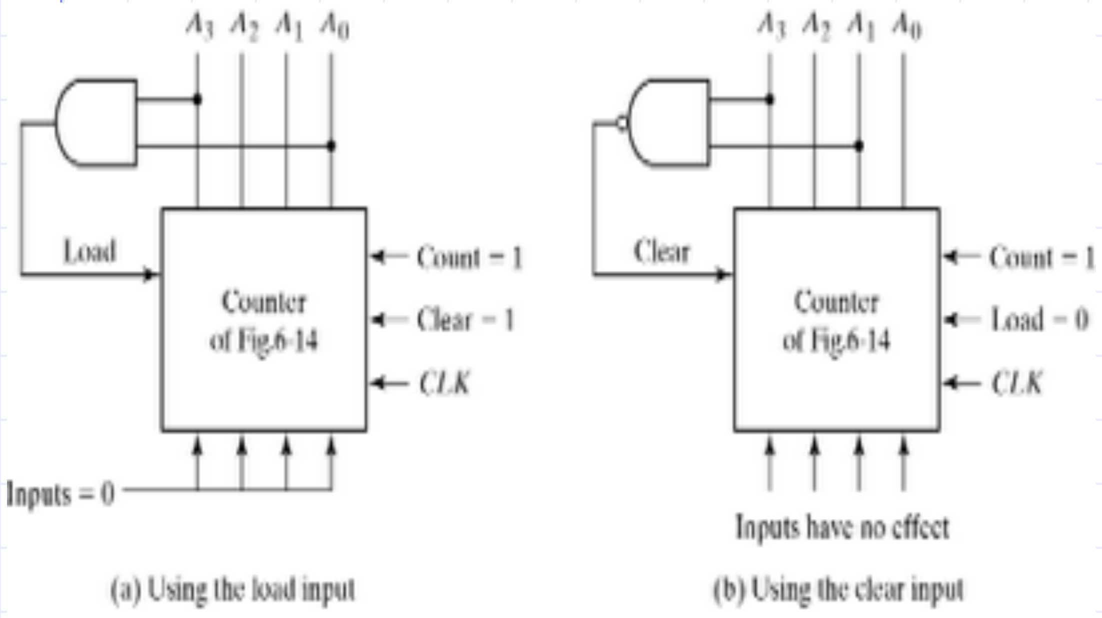


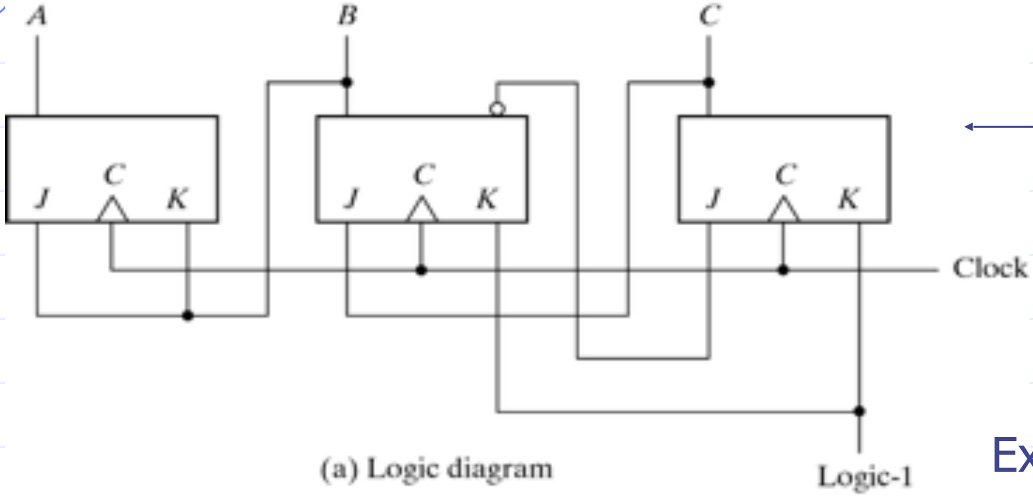
Fig. 6-15 Two ways to Achieve a BCD Counter Using a Counter with Parallel Load

❑The **AND** gate detects the occurrence of state **1001(9)** in the output. In this state, the load input is enabled and all-**0**'s input is loaded into register.

❑The **NAND** gate detects the count of **1010(10)** , as soon as this count occurs the register is **cleared.**

❑ A momentary **spike** occurs in output A2 as the count goes from **1001** to **1010** and immediately to **0000**

6-5 OTHER COUNTERS



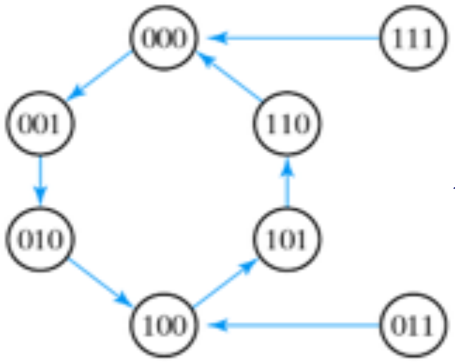
(a) Logic diagram

$J_A = B \qquad K_A = B$

$J_B = C \qquad K_B = 1$

$J_C = B' \qquad K_C = 1$

Except **011 ,111**



(b) State diagram

Fig. 6-16 Counter with Unused States

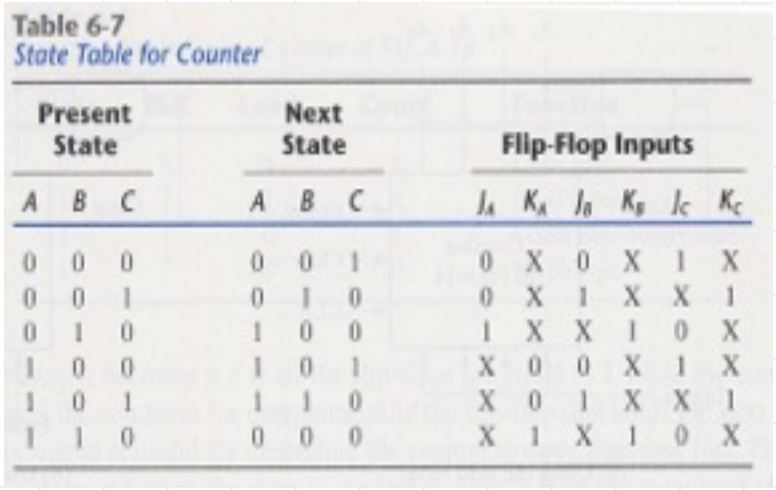


Table 6-7
State Table for Counter

Present State			Next State			Flip-Flop Inputs					
A	*B*	*C*	*A*	*B*	*C*	J_A	K_A	J_B	K_B	J_C	K_C
0	0	0	0	0	1	0	X	0	X	1	X
0	0	1	0	1	0	0	X	1	X	X	1
0	1	0	1	0	0	1	X	X	1	0	X
1	0	0	1	0	1	X	0	0	X	1	X
1	0	1	1	1	0	X	0	1	X	X	1
1	1	0	0	0	0	X	1	X	1	0	X

Ring Counter

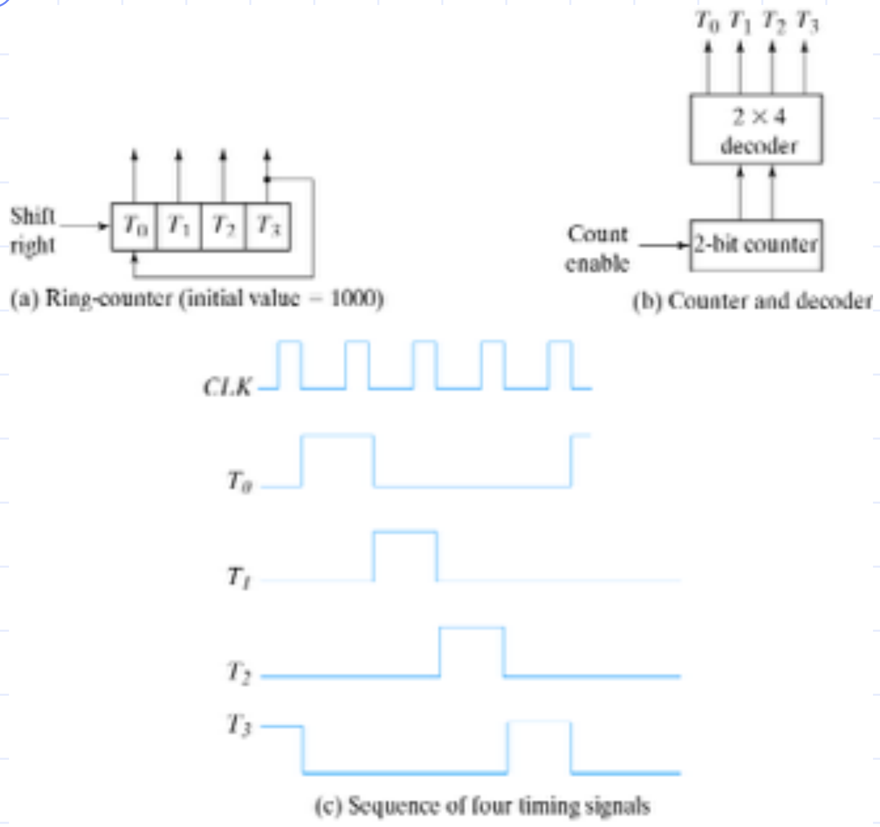
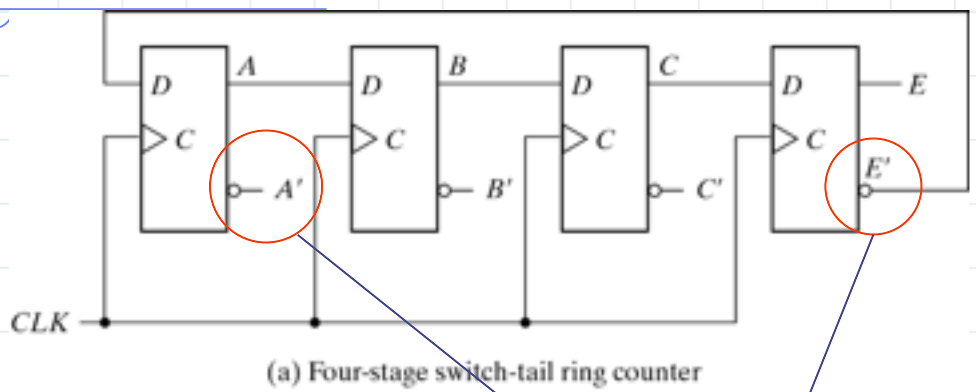


Fig. 6-17 Generation of Timing Signals

- A circular sift register with only one flip-flop being set at any **particular time**. ; all others are cleared.
- **The single bit** is shifted from one flip-flop to the other.

Johnson Counter



(a) Four-stage switch-tail ring counter

Sequence number	Flip-flop outputs A	B	C	E	AND gate required for output
1	0	0	0	0	$A'E'$
2	1	0	0	0	AB'
3	1	1	0	0	BC'
4	1	1	1	0	CE'
5	1	1	1	1	AE
6	0	1	1	1	$A'B$
7	0	0	1	1	$B'C$
8	0	0	0	1	$C'E$

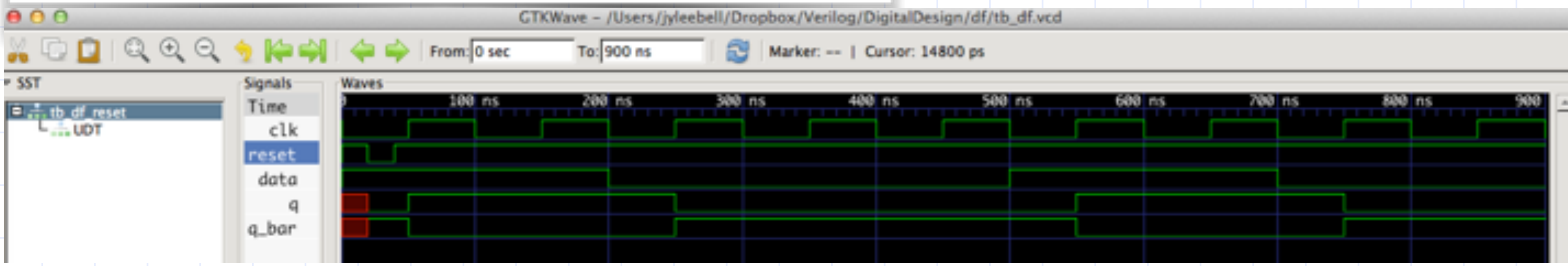
(b) Count sequence and required decoding

Fig. 6-18 Construction of a Johnson Counter

- A circular shift register with **the complement output of the last flip-flop** connected to the input of the first flip-flop.

D Flip-Flop

```
module df_reset (q, q_bar, data, reset, clk);
  output q, q_bar;
  input data, reset;
  input clk;

  reg q;

  assign q_bar = ~q;

  always @(posedge clk or negedge reset) begin
    if (reset == 0) q <= 0;
    else q <= data;
  end
endmodule
```


GTKWave – /Users/jyleebell/Dropbox/Verilog/DigitalDesign/df/tb_df.vcd

From: 0 sec To: 900 ns Marker: -- | Cursor: 14800 ps

6-1 REGISTERS

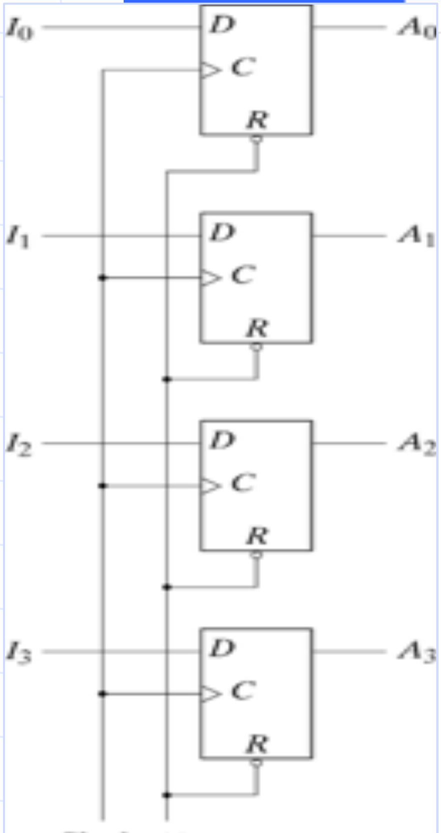
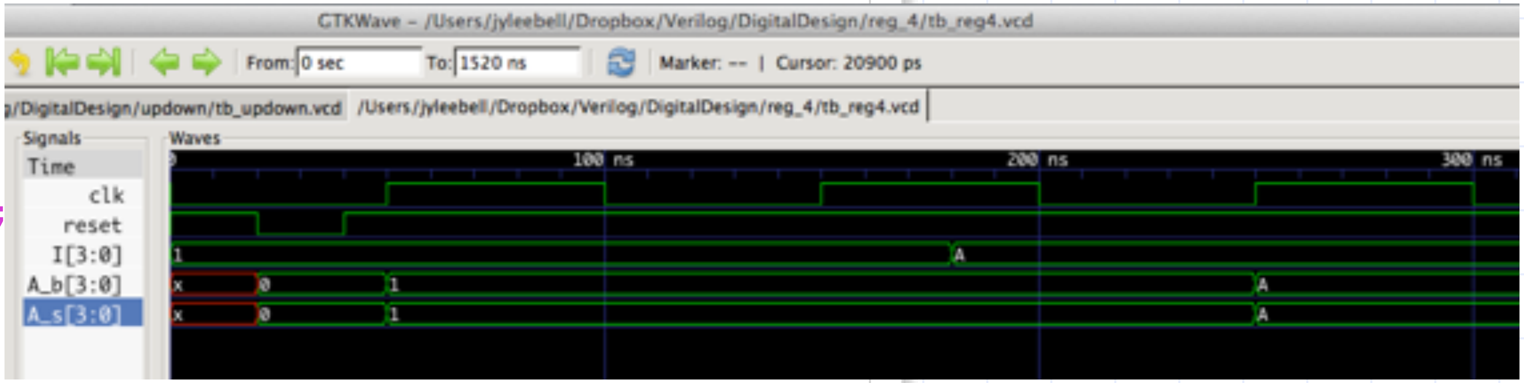


Fig. 6-1 4-Bit Register

```
module reg4_behav (A_b, I, Clear, Clock);
  output     [3:0] A_b;
  input      [3:0] I;
  input      Clear, Clock;

  reg        [3:0] A_b;

  always @(posedge Clock or negedge Clear) begin
    if (Clear == 0) A_b <= 4'b0000;
    else A_b <= I;
  end
endmodule

module reg4_str (A_s, I, Clear, Clock);
  output     [3:0] A_s;
  input      [3:0] I;
  input      Clear, Clock;

  df_reset FF0 (.q(A_s[0]), .data(I[0]), .reset(Clear), .clk(Clock));
  df_reset FF1 (.q(A_s[1]), .data(I[1]), .reset(Clear), .clk(Clock));
  df_reset FF2 (.q(A_s[2]), .data(I[2]), .reset(Clear), .clk(Clock));
  df_reset FF3 (.q(A_s[3]), .data(I[3]), .reset(Clear), .clk(Clock));
endmodule
```

6-1 REGISTERS

```
module tb_reg4;
  reg clk, reset;
  reg [3:0] I;

  wire [3:0] A_b, A_s;

  reg4_behav UDT0 (.A_b(A_b), .I(I), .Clear(reset), .Clock(clk));
  reg4_str UDT1 (.A_s(A_s), .I(I), .Clear(reset), .Clock(clk));

  initial begin
    clk = 0;
    I = 0;
    reset = 1;
    #20 reset = 0;
    #20 reset = 1;
  end

  always #50 clk = ~clk;

  initial begin
  I = 1;
  #180 I = 10;
  #180 I = 11;
  #200 I = 8;
  #20  reset = 0;
  #200  reset = 1;
  #180 I = 2;
  #180 I = 3;
  #200 I = 4;
  #180 $finish;
  end
endmodule
```


GTKWave – /Users/jyleebell/Dropbox/Verilog/DigitalDesign/reg_4/tb_reg4.vcd
From: 0 sec
To: 1520 ns
Marker: -- | Cursor: 20900 ps
/DigitalDesign/updown/tb_updown.vcd
/Users/jyleebell/Dropbox/Verilog/DigitalDesign/reg_4/tb_reg4.vcd
Signals
Waves
Time
clk
reset
I[3:0]
A_b[3:0]
A_s[3:0]
100 ns
200 ns
300 ns
1
A
x
0
1
A
x
0
1
A

Register with Parallel Load

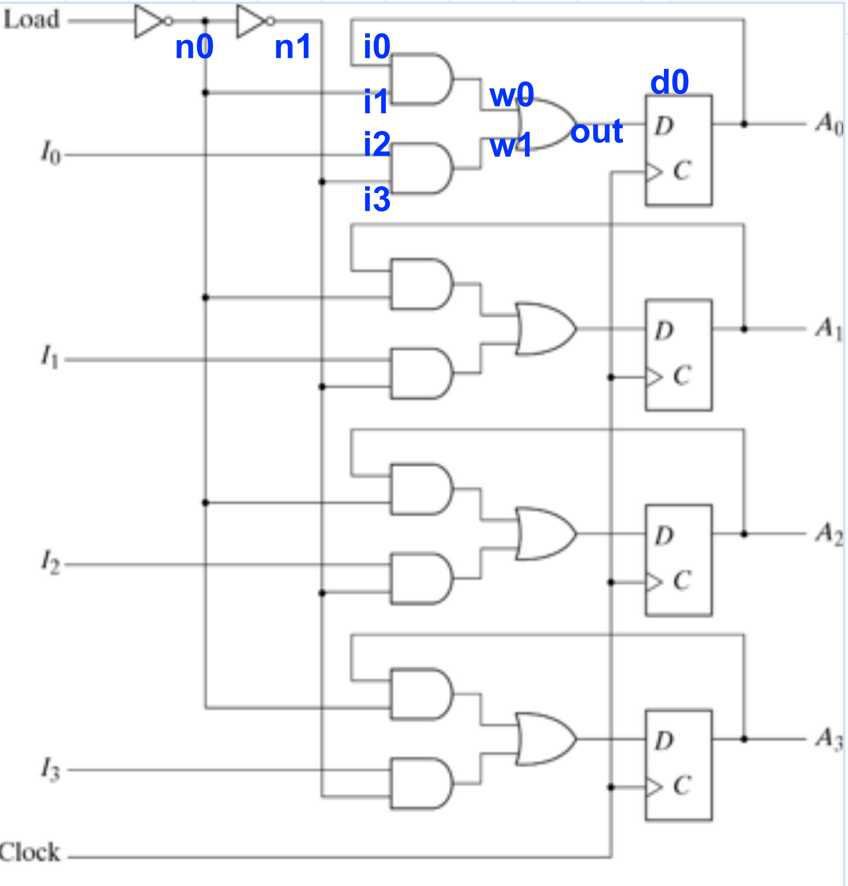


Fig. 6-2 4-Bit Register with Parallel Load

```
module and_or (i0, i1, i2, i3, out);
input i0, i1, i2, i3;
output out;

wire w0, w1;

and AND0 (w0, i0, i1);
and AND1 (w1, i2, i3);
or OR0 (out, w0, w1);
endmodule

module reg4_pl_str (A_s, I, Load, Clock);
output     [3:0] A_s;
input      [3:0] I;
input      Clock, Load;

wire d0, d1, d2, d3;
wire n0, n1;

not U0 (n0, Load);
not U1 (n1, n0);
and_or U2 (A_s[0], n0, I[0], n1, d0);
and_or U3 (A_s[1], n0, I[1], n1, d1);
and_or U4 (A_s[2], n0, I[2], n1, d2);
and_or U5 (A_s[3], n0, I[3], n1, d3);

  df_reset FF0 (.q(A_s[0]), .data(d0), .reset(1'b1), .clk(Clock));
  df_reset FF1 (.q(A_s[1]), .data(d1), .reset(1'b1), .clk(Clock));
  df_reset FF2 (.q(A_s[2]), .data(d2), .reset(1'b1), .clk(Clock));
  df_reset FF3 (.q(A_s[3]), .data(d3), .reset(1'b1), .clk(Clock));
endmodule
```

Register with Parallel Load

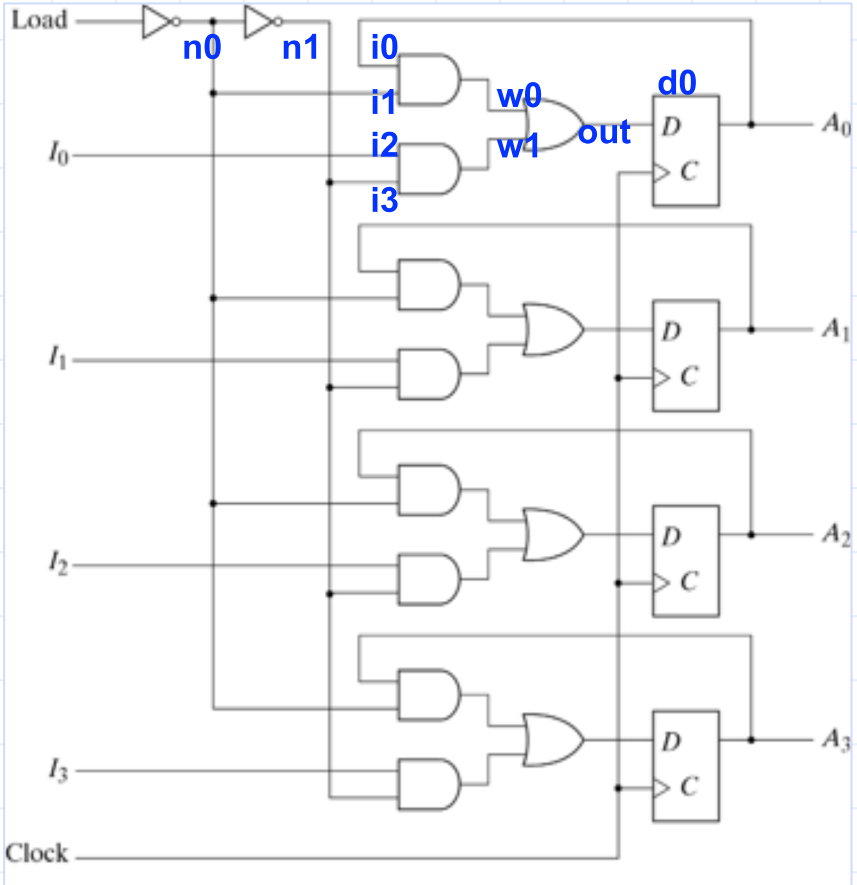
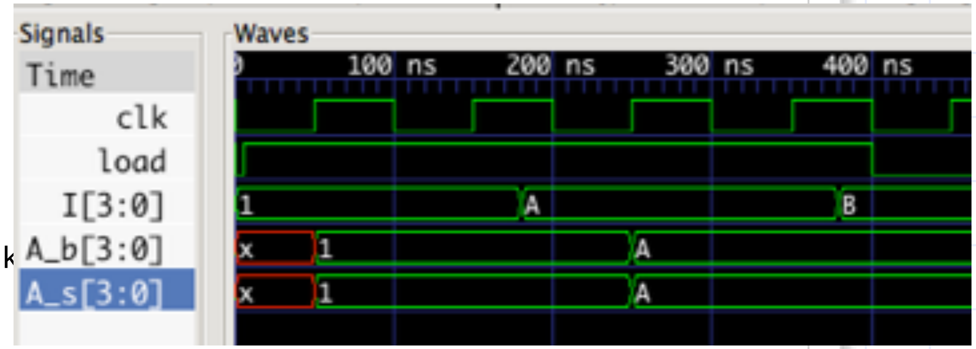


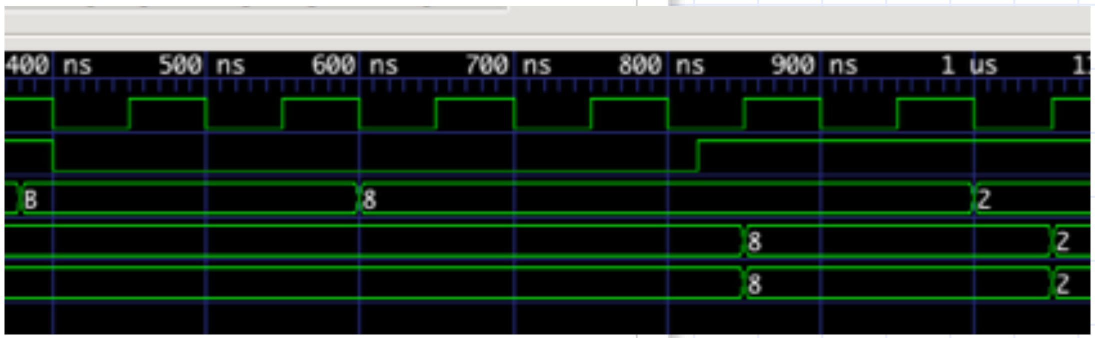
Fig. 6-2 4-Bit Register with Parallel Load

```
module reg4_pl_behav (A_b, I, Load, Clock);
  output    [3:0] A_b;
  input     [3:0] I;
  input     Load, Clock;

  reg       [3:0] A_b;

  always @(posedge Clock) begin
    if (Load == 1) begin
      A_b <= I;
    end
    else begin
      A_b <= A_b;
    end
  end
endmodule
```

Register with Parallel Load

```
module tb_reg4_pl;
  reg clk, load;
  reg [3:0] I;

  wire [3:0] A_b, A_s;

  reg4_pl_behav UDT0 (.A_b(A_b), .I(I), .Load(load), .Clock(clk));
  reg4_pl_str UDT1 (.A_s(A_s), .I(I), .Load(load), .Clock(clk));

  initial begin
    clk = 0;
    I = 0;
    load = 0;
    #6 load = 1;
  end

  always #50 clk = ~clk

  initial begin
    I = 1;
    #180 I = 10;
    #20  load = 1;
    #180 I = 11;
    #20  load = 0;
    #200 I = 8;
    #20  load = 0;
    #200 load = 1;
    #180 I = 2;
    #180 I = 3;
    #200 I = 4;
    #180 $finish;
  end
endmodule
```





6-2 SHIFT REGISTERS

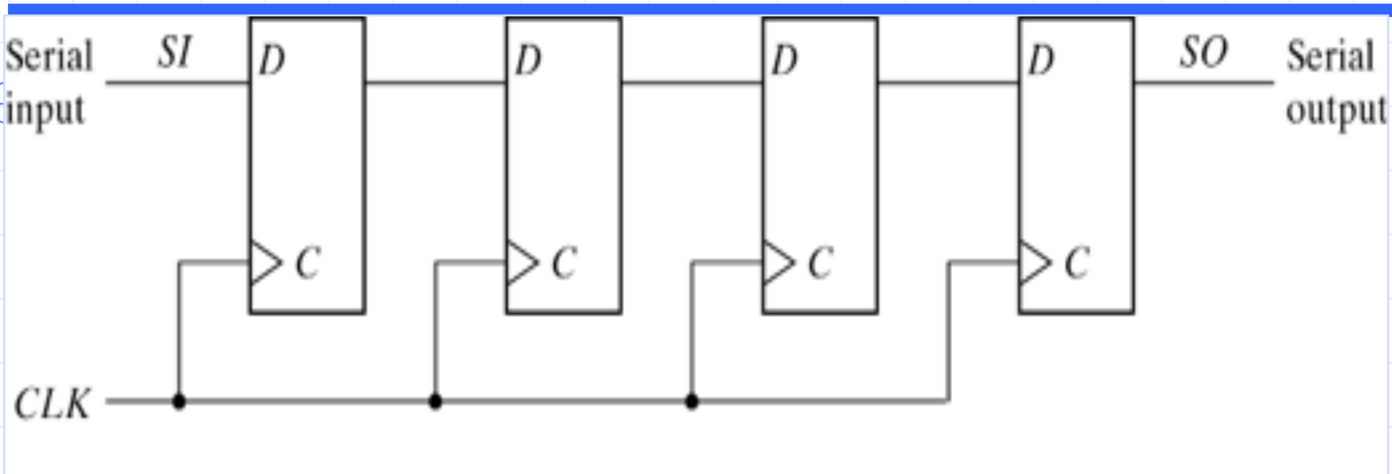


Fig. 6-3 4-Bit Shift Register

```
module shift_reg4_str (si, so, clock, r);
  output     so;
  output     [0:3] r;
  input      si;
  input      clock;

  wire d1, d2, d3;

  df_reset FF0 (.q(d1), .data(si), .reset(1'b1), .clk(clock));
  df_reset FF1 (.q(d2), .data(d1), .reset(1'b1), .clk(clock));
  df_reset FF2 (.q(d3), .data(d2), .reset(1'b1), .clk(clock));
  df_reset FF3 (.q(so), .data(d3), .reset(1'b1), .clk(clock));

  assign r = {d1, d2, d3, so};

endmodule
```

6-2 SHIFT REGISTERS

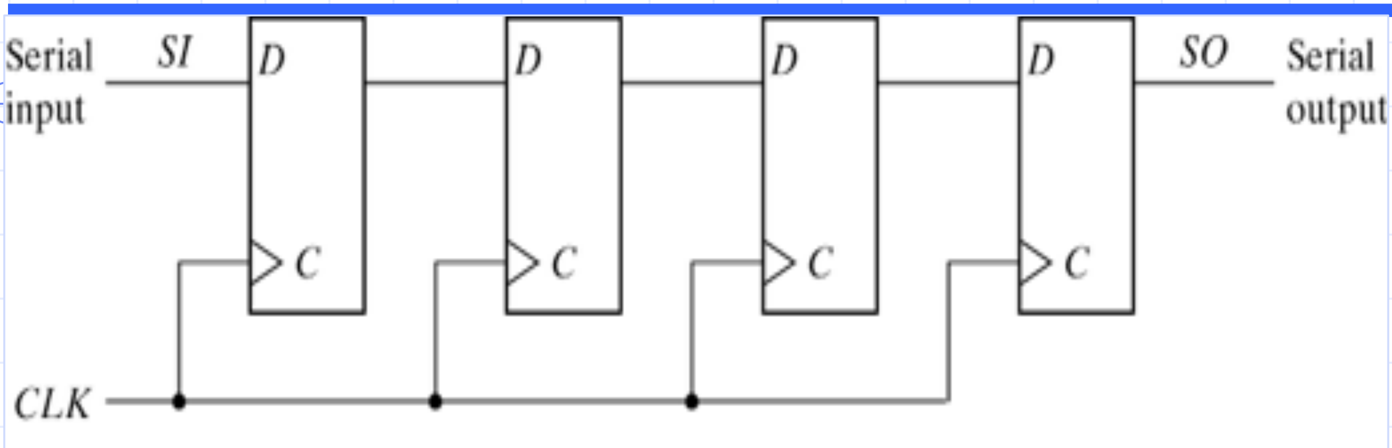
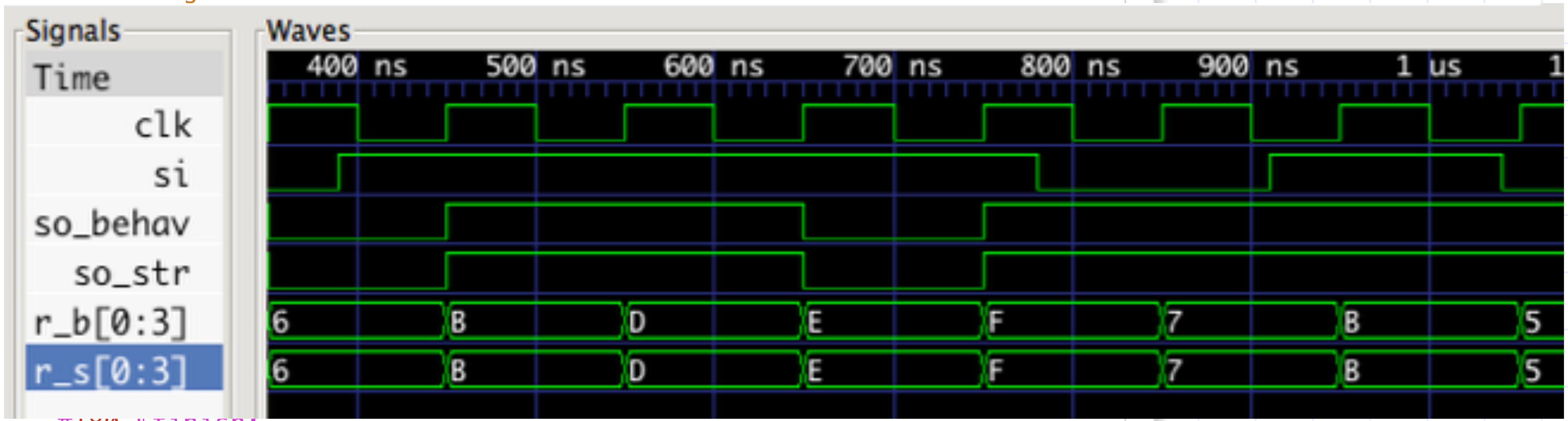


Fig. 6-3 4-Bit Shift Register

```
module shift_reg4_behav (si, so, clock, r);
  output    so;
  output    [0:3] r;
  input     si;
  input     clock;

  reg [0:3] R;
  wire      so;

  always @(posedge clock) begin
    R <= {si, R[0:2]} ;
  end

  assign so = R[3];
  assign r = R;
endmodule
```

6-2 SHIFT REGISTERS

```
module tb_shift_reg4;
  wire so_behav, so_str;
  wire [0:3] r_b, r_s;
  reg clk, si;

  shift_reg4_behav UDT0 (.si(si), .so(so_behav), .clock(clk),.r(r_b));
  shift_reg4_str UDT1 (.si(si), .so(so_str), .clock(clk),.r(r_s));
  initial begin
    clk = 0;
    si = 0;
  end

  always #50 clk = ~clk;

  initial begin
```



```
    #180 $finish;
  end
endmodule
```

Universal Shift Register

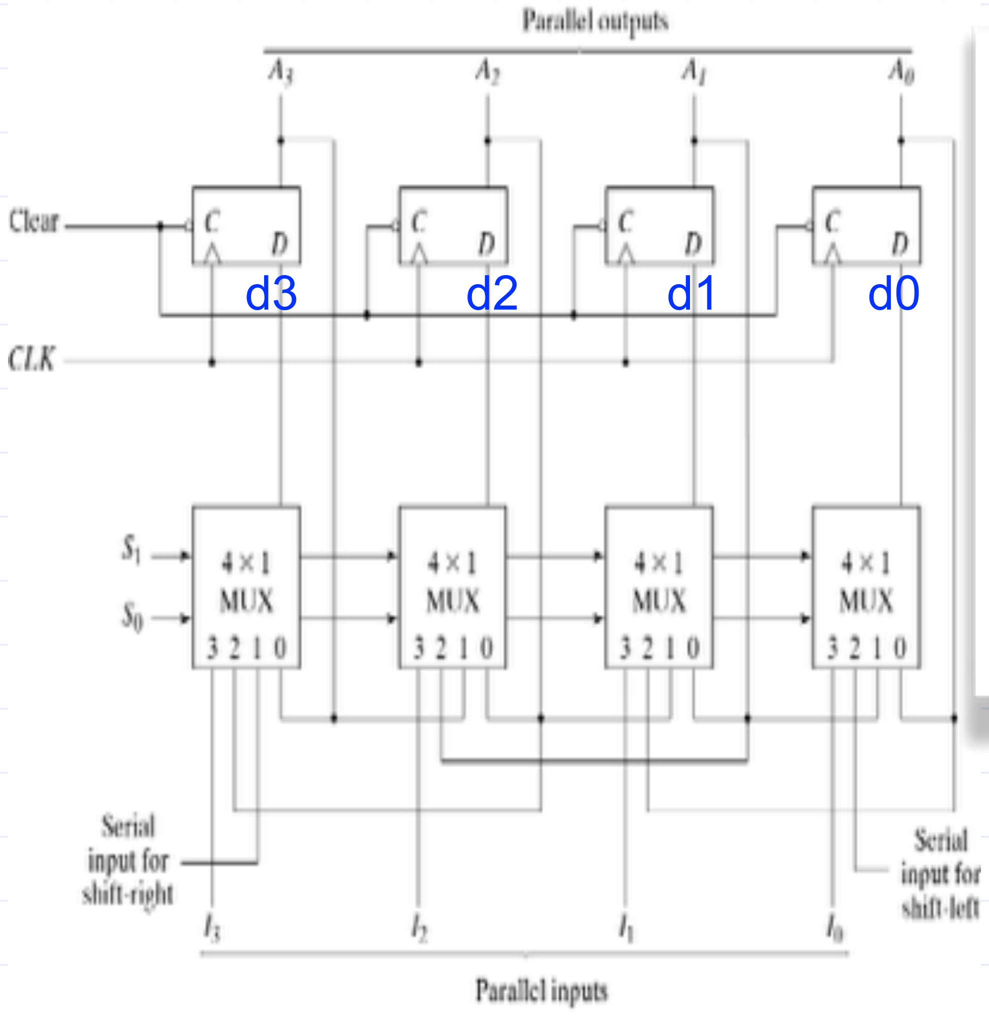


Fig. 6-7 4-Bit Universal Shift Register

```
module mux4x1(out, d3, d2, d1, d0, s1, s0);
  output out;
  input d0, d1, d2, d3;
  input s1, s0;

  reg out;

  always @* begin
    case ({s1, s0})
      0: out <= d0;
      1: out <= d1;
      2: out <= d2;
      3: out <= d3;
      default: out <= 1'bx;
    endcase
  end
endmodule
```

Universal Shift Register

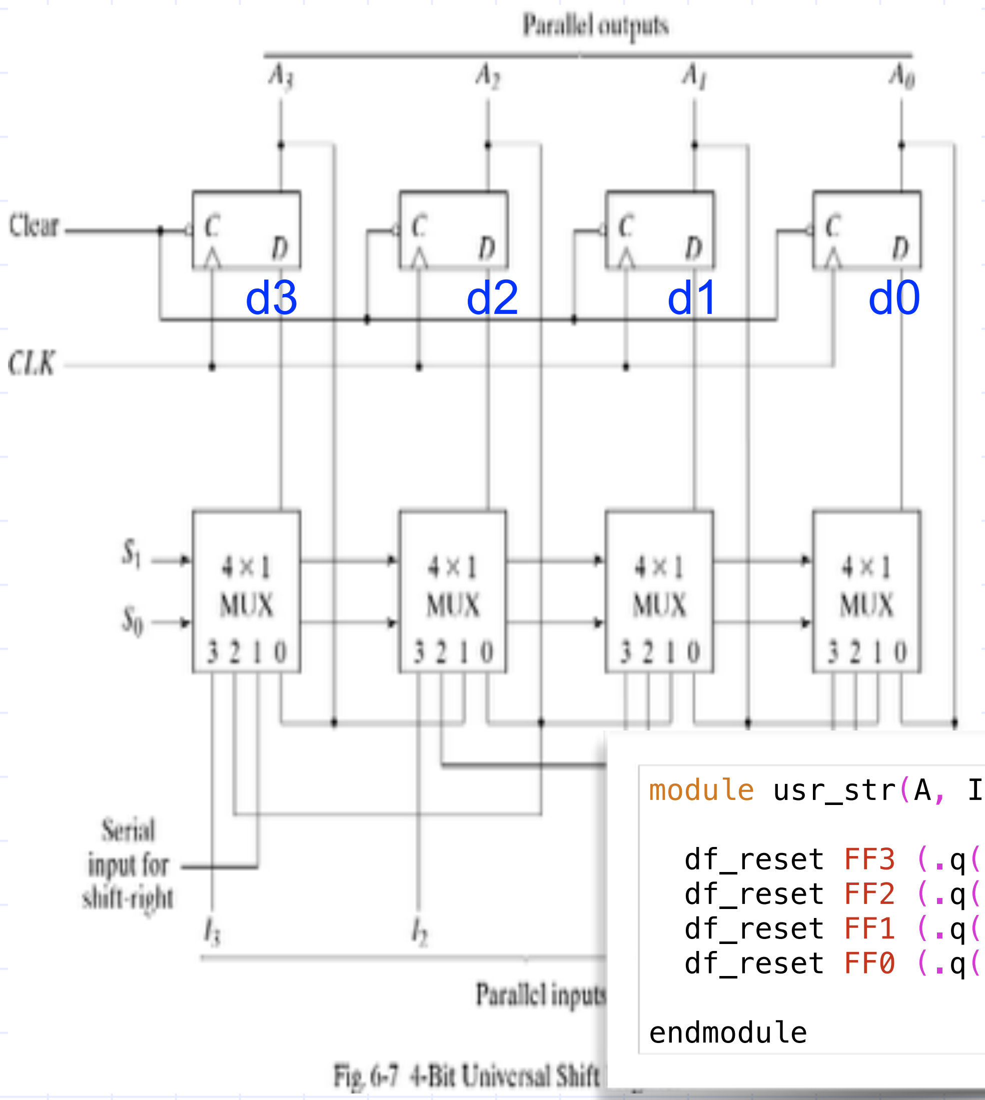
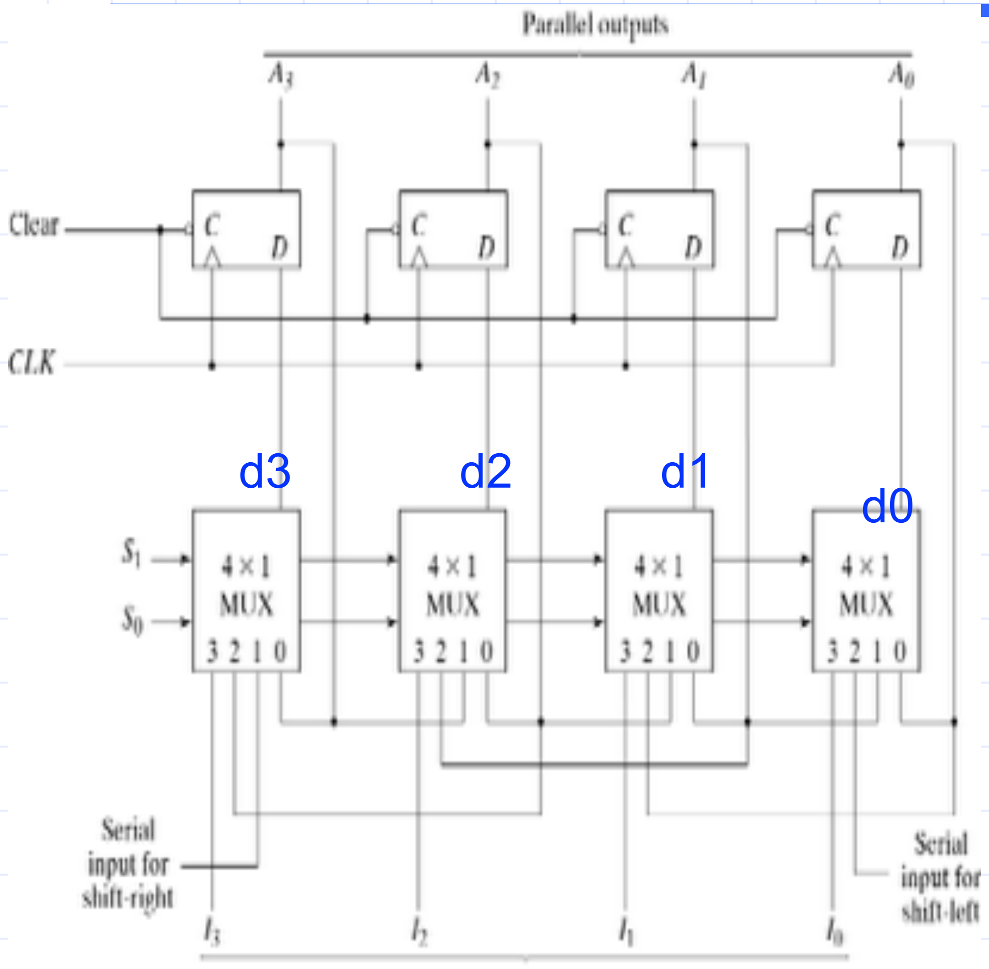


Fig. 6-7 4-Bit Universal Shift

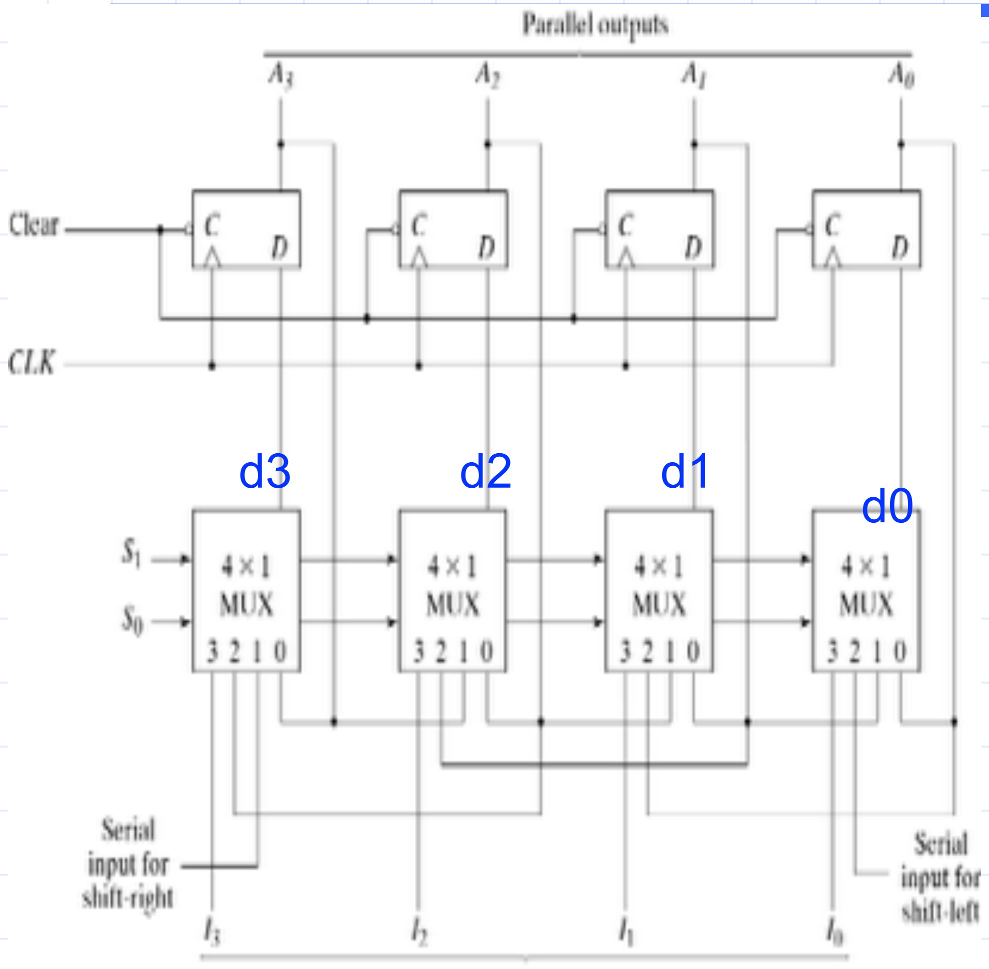
```
module usr_str(A, I, sir, sil, control, clear, clock);

  df_reset FF3 (.q(A[3]), .data(d3), .reset(clear), .clk(clock));
  df_reset FF2 (.q(A[2]), .data(d2), .reset(clear), .clk(clock));
  df_reset FF1 (.q(A[1]), .data(d1), .reset(clear), .clk(clock));
  df_reset FF0 (.q(A[0]), .data(d0), .reset(clear), .clk(clock));

endmodule
```

Universal Shift Register



```
mux4x1   MUX3 (.out(d3), .d3(I[3]), .d2(A[2]),    .d1(sir),   .d0(A[3]), .s1(control[1]), .s0(control[0]));
mux4x1   MUX2 (.out(d2), .d3(I[2]), .d2(A[1]),    .d1(A[3]), .d0(A[2]), .s1(control[1]), .s0(control[0]));
mux4x1   MUX1 (.out(d1), .d3(I[1]), .d2(A[0]),    .d1(A[2]), .d0(A[1]), .s1(control[1]), .s0(control[0]));
mux4x1   MUX0 (.out(d0), .d3(I[0]), .d2(A[sil]), .d1(A[1]), .d0(A[0]), .s1(control[1]), .s0(control[0]));
```

Universal Shift Register



```
mux4x1   MUX3 (.out(d3), .d3(I[3]), .d2(A[2]),   .d1(sir),  .d0(A[3]), .s1(control[1]), .s0(control[0]));
mux4x1   MUX2 (.out(d2), .d3(I[2]), .d2(A[1]),   .d1(A[3]), .d0(A[2]), .s1(control[1]), .s0(control[0]));
mux4x1   MUX1 (.out(d1), .d3(I[1]), .d2(A[0]),   .d1(A[2]), .d0(A[1]), .s1(control[1]), .s0(control[0]));
mux4x1   MUX0 (.out(d0), .d3(I[0]), .d2(A[sil]), .d1(A[1]), .d0(A[0]), .s1(control[1]), .s0(control[0]));
```

Universal Shift Register

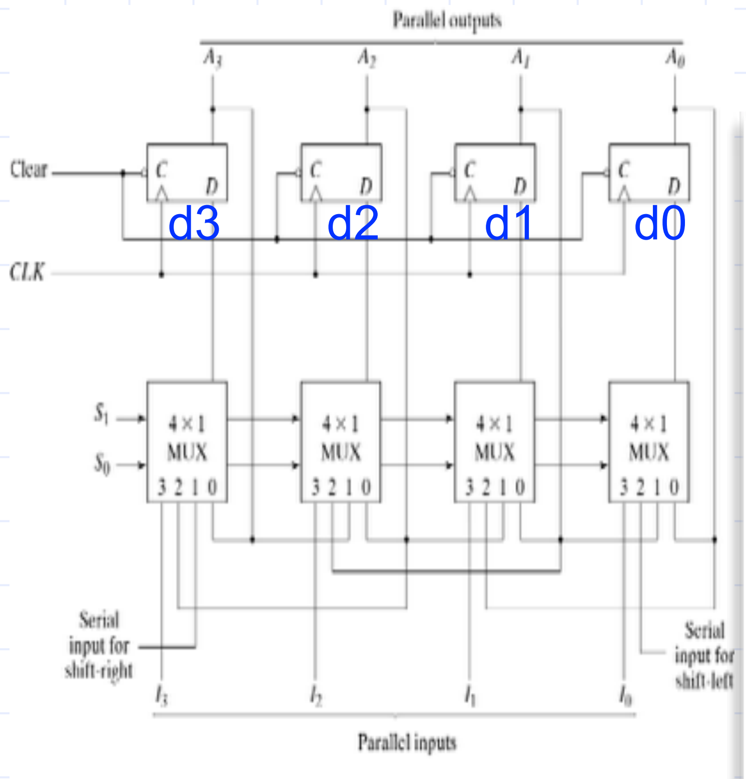
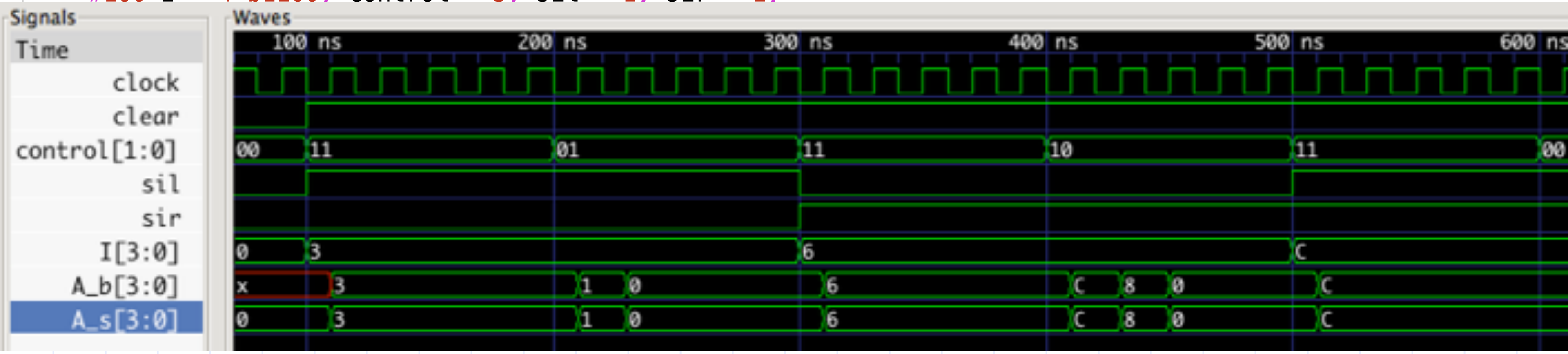


Fig. 6-7 4-Bit Universal Shift Register

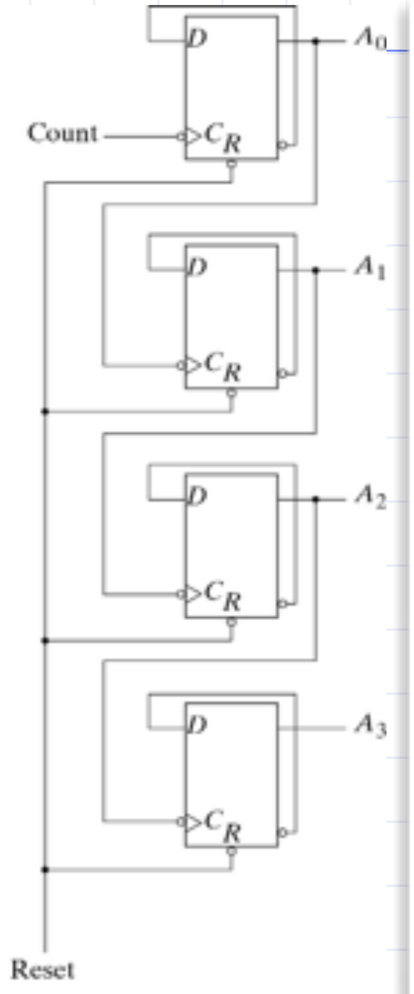
```
module usr_behav(A, I, sir, sil, control, clear, clock);
  output    [3:0] A;
  input     [3:0] I;
  input     [1:0] control;
  input     sir, sil, clear, clock;

  wire      d3, d2, d1, d0;
  reg       [3:0] A;
  wire      [3:0] I;

  always @(posedge clock or negedge clear) begin
    if (clear == 0) begin
      A <= 4'b0000;
    end begin
      case (control)
        0: A <= A;
        1: A <= {sir, A[3:1]};
        2: A <= {A[2:0], sil};
        3: A <= I;
      endcase
    end
  end

endmodule
```

Universal Shift Register

```
module tb_usr;
  reg   [3:0]   I;
  reg   sil, sir, clear, clock;
  reg   [1:0] control;
  wire  [3:0] A_s, A_b;

  usr_str DUT0 (.A(A_s), .I(I), .sir(sir), .sil(sil), .control(control), .clear(clear), .clock(clock));
  usr_behav DUT1 (.A(A_b), .I(I), .sir(sir), .sil(sil), .control(control), .clear(clear), .clock(clock));

  initial begin
    I = 0; sil = 0; sir = 0; clear = 0; clock = 0; control = 0;

    #100 I = 4'b0011; control = 3; sil = 1; sir = 0; clear = 1;
    #100 control = 1;

    #100 I = 4'b0110; control = 3; sil = 0; sir = 1;
    #100 control = 2;

    #100 I = 4'b1100; control = 3; sil = 1; sir = 1;
```



6-3 RIPPLE COUNTERS

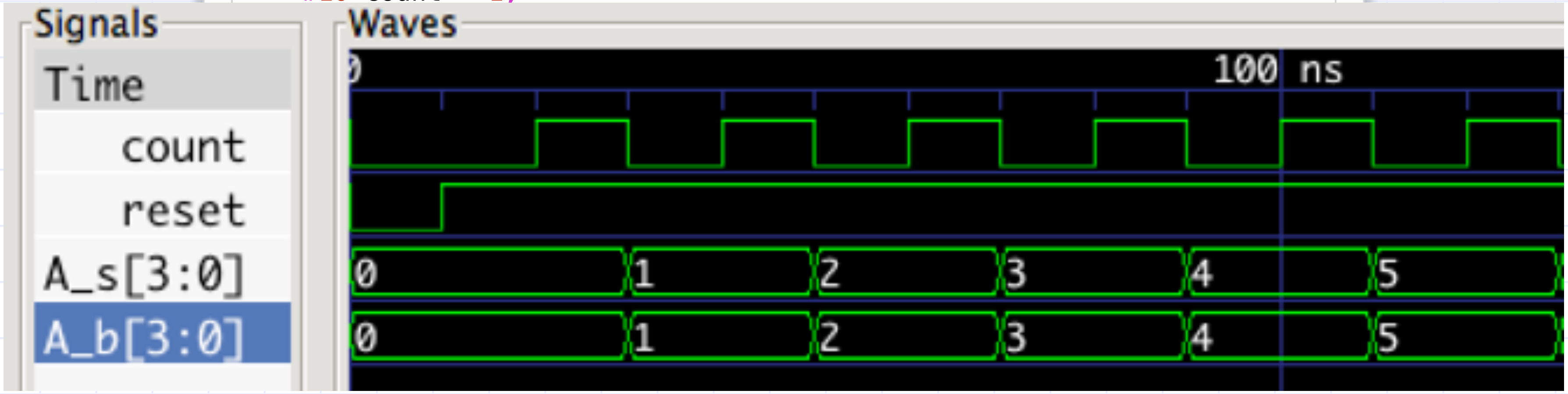


(b) With D flip-flops

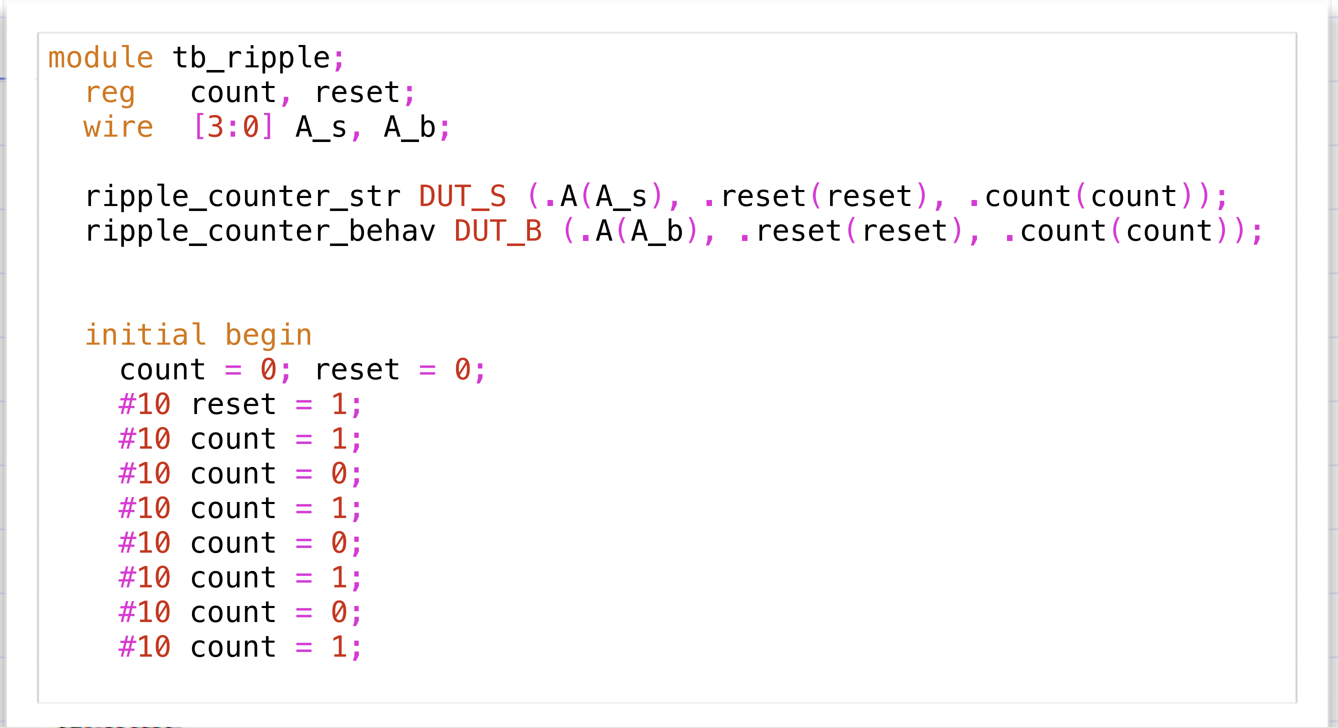
```
module df_reset (q, q_bar, data, reset, clk);
  output q, q_bar;
  input data, reset;
  input clk;

  reg q;

  assign q_bar = ~q;

  always @(negedge clk or negedge reset) begin
    if (reset == 0) q <= 0;
    else q <= data;
  end
endmodule

module ripple_counter_str (A, reset, count);
  output     [3:0] A;
  input      reset, count;

  wire       d3, d2, d1, d0;
  wire       [3:0] A;

  df_reset FF0 (.q(A[0]), .q_bar(d0), .data(d0), .reset(reset), .clk(count));
  df_reset FF1 (.q(A[1]), .q_bar(d1), .data(d1), .reset(reset), .clk(A[0]));
  df_reset FF2 (.q(A[2]), .q_bar(d2), .data(d2), .reset(reset), .clk(A[1]));
  df_reset FF3 (.q(A[3]), .q_bar(d3), .data(d3), .reset(reset), .clk(A[2]));
endmodule
```

6-3 RIPPLE COUNTERS

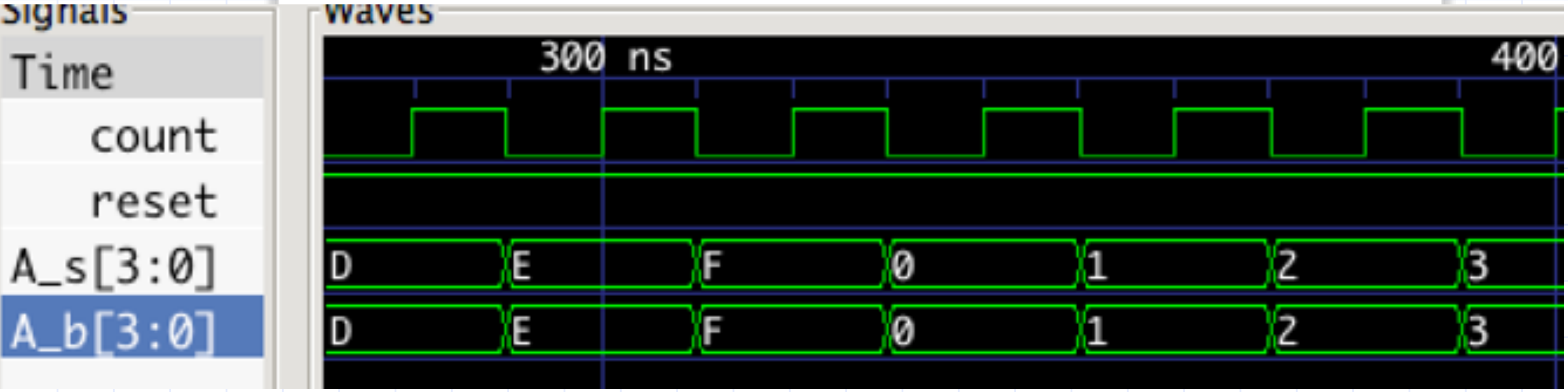
```
module tb_ripple;
  reg   count, reset;
  wire  [3:0] A_s, A_b;

  ripple_counter_str DUT_S (.A(A_s), .reset(reset), .count(count));
  ripple_counter_behav DUT_B (.A(A_b), .reset(reset), .count(count));


  initial begin
    count = 0; reset = 0;
    #10 reset = 1;
    #10 count = 1;
    #10 count = 0;
    #10 count = 1;
    #10 count = 0;
    #10 count = 1;
    #10 count = 0;
    #10 count = 1;
```



6-3 RIPPLE COUNTERS



```
module tb_ripple;
  reg   count, reset;
  wire  [3:0] A_s, A_b;

  ripple_counter_str DUT_S (.A(A_s), .reset(reset), .count(count));
  ripple_counter_behav DUT_B (.A(A_b), .reset(reset), .count(count));


  initial begin
    count = 0; reset = 0;
    #10 reset = 1;
    #10 count = 1;
    #10 count = 0;
    #10 count = 1;
    #10 count = 0;
    #10 count = 1;
    #10 count = 0;
    #10 count = 1;
```


Signals
Waves
Time
count
reset
A_s[3:0]
A_b[3:0]
300 ns
400
D
E
F
0
1
2
3
D
E
F
0
1
2
3