Chapter 9

Counters and Shift Registers

計數器與移位記錄器

9.1 Basic Concept of Digital Counters

- **Counter**: A Sequential Circuit that counts pulses. Used for Event Counting, Frequency Division, Timing, and Control Operations.

- **Shift Register**: A Sequential Circuit that moves stored data bits in a specific direction. Used in Serial Data Transfers, SIPO/PISO Conversions, Arithmetic, and Delays.

Counter Terminology(術語) – 1

- A **Counter** is a digital circuit whose outputs progress in a predictable repeating pattern. It advances one state for each clock pulse.

- **State Diagram**: A graphical diagram showing the progression of states in a sequential circuit.

Counter Terminology – 2

- **Count Sequence**: The specific series of output states through which a counter progresses.

- **Modulus**: The number of states through which a counter sequences repeating (mod-n).

- **Counter directions**:
  - UP - count with an ascending sequence.
  - DOWN - count with an descending sequence.

Counter Modulus

- Modulus of a counter is the number of states through which the counter progresses.

- A Mod-12 UP Counter counts 12 states from 0000 to 1011 (0 to 11 in decimal), recycles to 0000, and continues, and repeats.

- A Mod-12 DOWN counter counts from 1011 to 0000 (11 to 0 in decimal), recycle to 1011, and repeats.

State Diagram

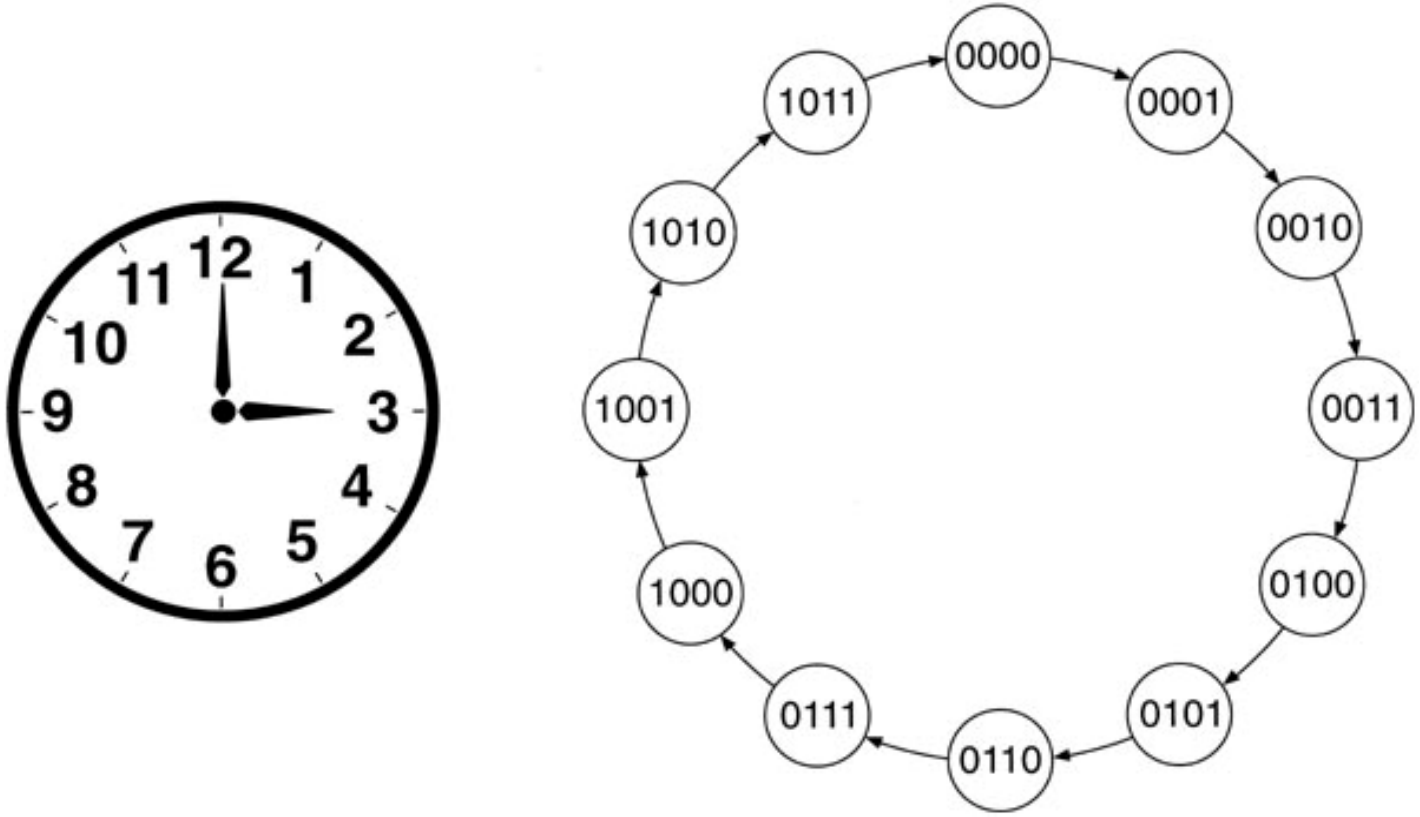
- A diagram that shows the progressive states of a sequential circuit.

- The progression from one state to the next state is shown by an arrow.

  $(0000 \Rightarrow 0001 \Rightarrow 0010)$.

- Each state progression is caused by a pulse on the clock to the sequential circuit.

MOD-12 Counter State Diagram -1



Mod-12 State Diagram and Analog Clock Face

MOD-12 Counter State Diagram -2

Both the clock face and the state diagram represent a closed system of counting. In each case, when we reach the end of the count sequence, we start over from the beginning of the cycle.

if we want to know the 8th state after 0111 in a mod-12 UP counter, we start at state 0111 and count 8 positions in the direction of the arrows. This brings us to state 0000 (the recycle point) in 5 counts and then on to state 0011 in another 3 counts. This closed system of counting and adding is known as **modulo arithmetic.**

Number of Bits and Maximum Modulus -1

KEY TERMS

Maximum Modulus (m_{max}) The largest number of counter states that can be represented by n bits ($m_{max} = 2^n$).

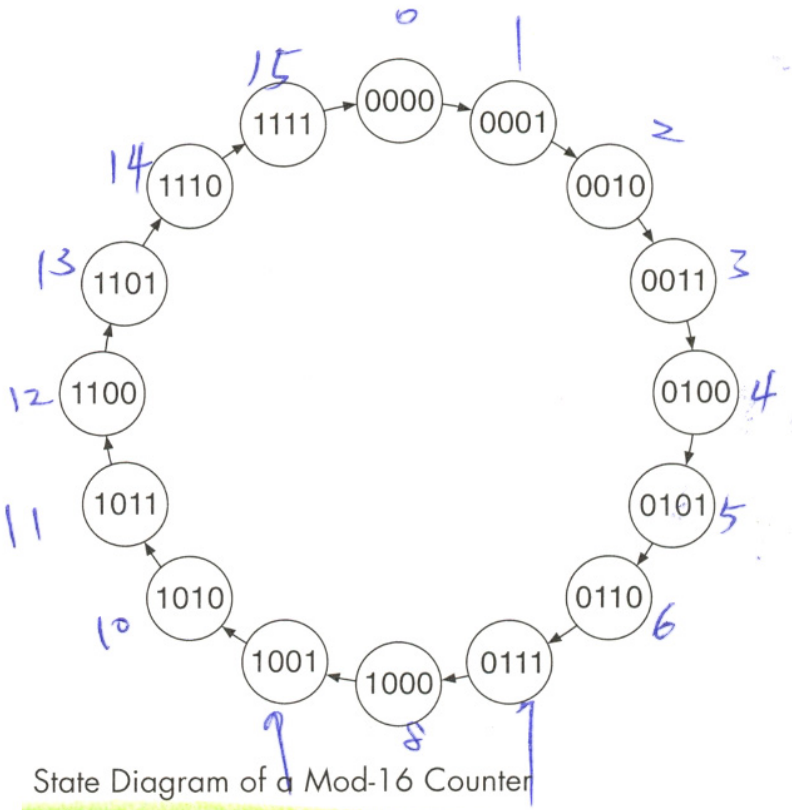
Full-Sequence Counter A counter whose modulus is the same as its maximum modulus ($m = 2^n$ for an n-bit counter).

Binary Counter A counter that generates a binary count sequence.

Truncated-Sequence Counter A counter whose modulus is less than its maximum modulus ($m < 2^n$ for an n-bit counter).

Number of Bits and Maximum Modulus -2

- Maximum Modulus



State Diagram of a Mod-16 Counter

The **maximum modulus** of a 4-bit counter is 16 (= 2^4). The count sequence of a mod-16 UP counter is from 0000 to 1111 (0 to 15 in decimal), as illustrated in the state diagram

Number of Bits and Maximum Modulus -3

- Full-sequence counter

In general, an n-bit counter has a maximum modulus of 2^n and a count sequence from 0 to $(2^n - 1)$ (i.e., all 0s to all 1s). Since a mod-16 counter has a modulus of 2^n $(= m_{max})$, we say that it is a **full-sequence counter.**

- Truncated sequence counter

A counter, such as a mod-12 counter, whose modulus is less than 2^n, is called a **truncated sequence counter.**

Count-Sequence Table and Timing Diagram

KEY TERM

Count-Sequence Table A list of counter states in the order of the count sequence.

Two ways to represent a count sequence other than a state diagram are by a **count-sequence table** and by a timing diagram. The count sequence table is simply a list of counter states in the same order as the count sequence. Table 9.1 and Table 9.2 show the count-sequence tables of a mod-16 UP counter and a mod-12 UP counter, respectively.

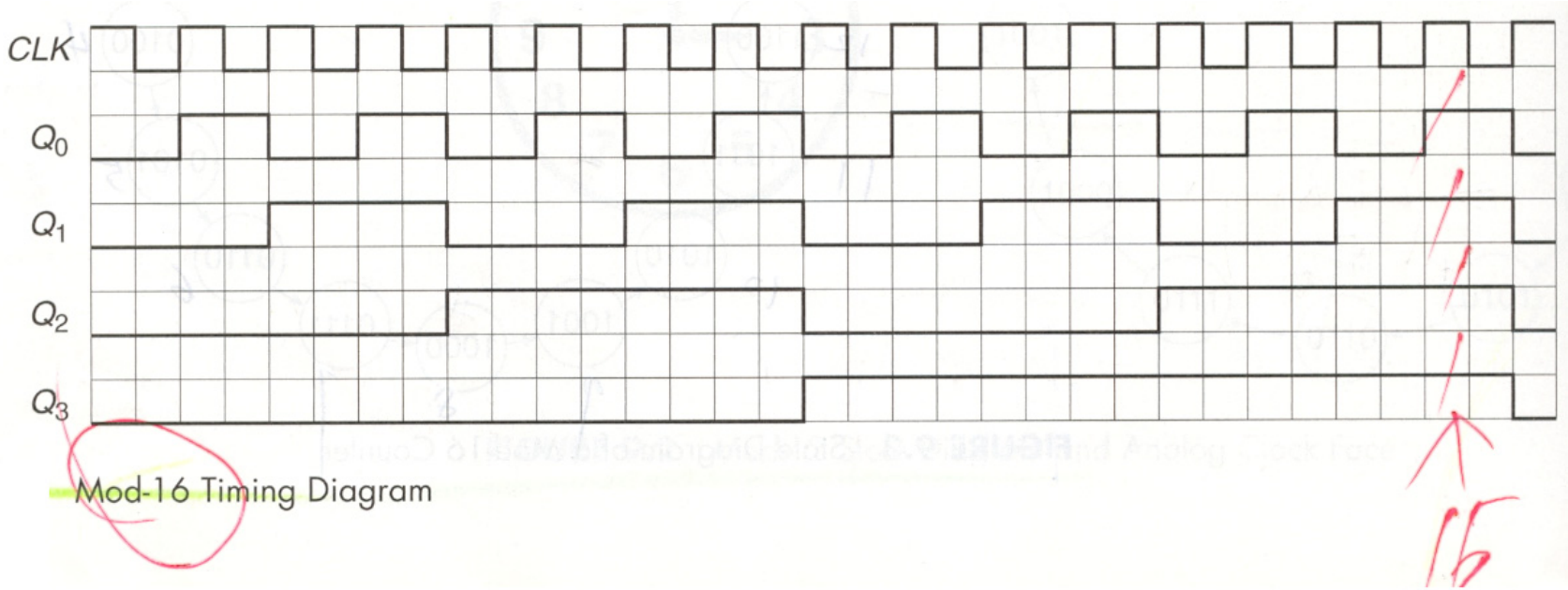
TABLE 9.1 Mod-16 Count-Sequence Table

Q_3	Q_2	Q_1	Q_0
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

TABLE 9.2 Mod-12 Count-Sequence Table

Q_3	Q_2	Q_1	Q_0
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1

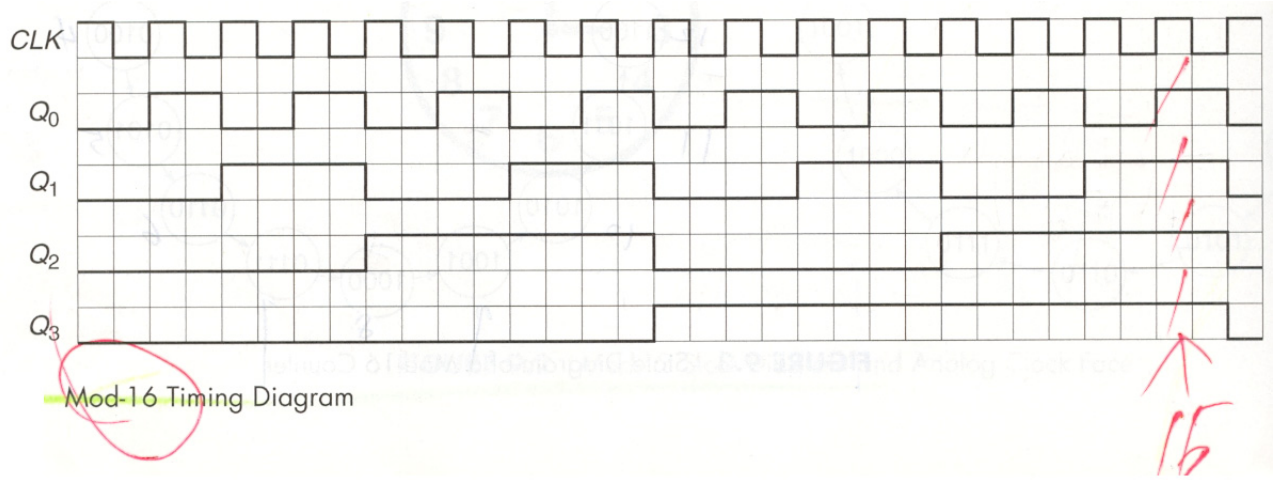
Counter Timing Diagrams – 1



- Shows the timing relationships between the input clock and the outputs Q_3, Q_2, Q_1, …Q_n of a counter.
- For a 4-bit mod 16 counter, the output Q_0 changes for every clock pulse, Q_1 changes on every two clock pulses, Q_2 on four, and Q_3 on 8 clocks.

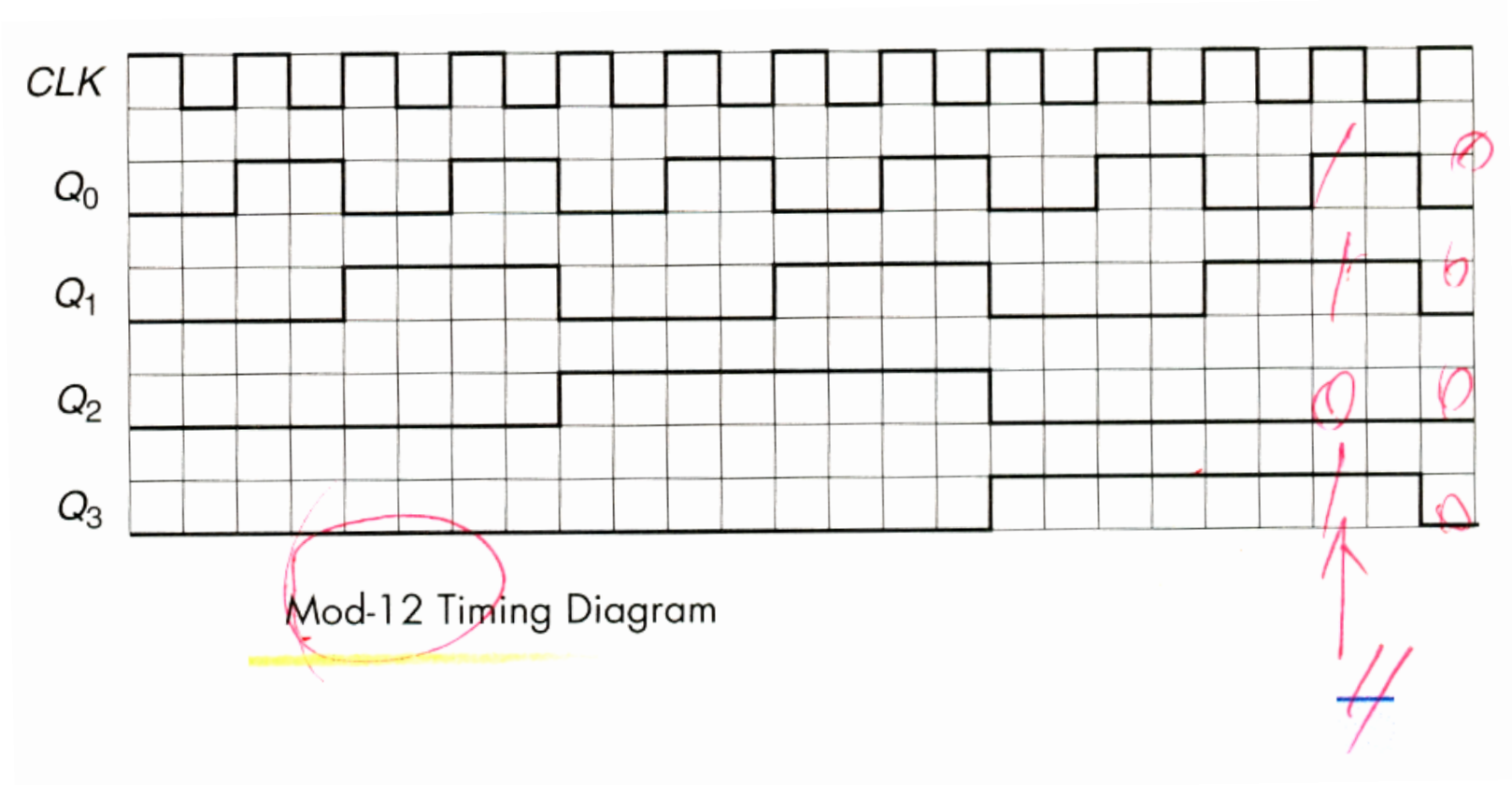
Counter Timing Diagrams – 2

- The outputs ($Q_0 \Rightarrow Q_3$) of the counter can be used as **frequency dividers** with Q_0 = clock ÷ 2, Q_1 = clock ÷ 4, Q_2 = clock ÷ 8, and Q_3 = clock ÷ 16.



A divide-by-two ratio relates the frequencies of adjacent outputs of a binary counter. For example, if the clock frequency is f_c = 16 MHz, the frequencies of the output waveforms are: 8 MHz ($f_0 = f_c/2$); 4 MHz ($f_1 = f_c/4$); 2 MHz ($f_2 = f_c/8$); 1 MHz ($f_3 = f_c/16$).

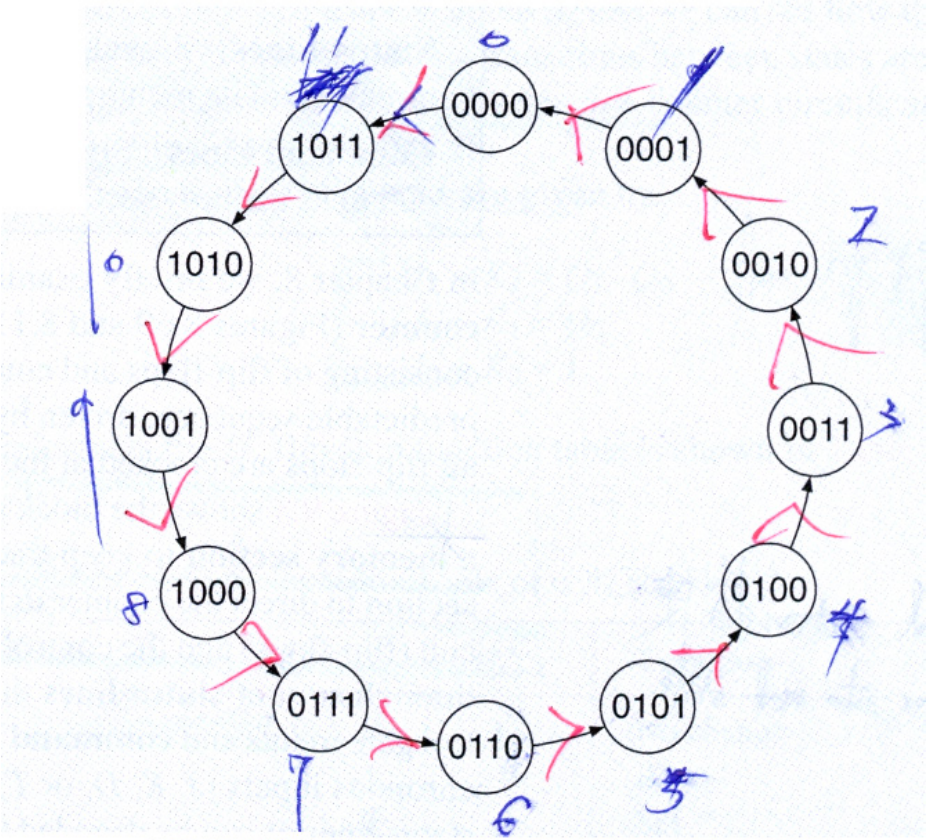
Counter Timing Diagrams – 3



12 counter the relationships between clock frequency, f_c, and output frequencies are: $f_0 = f_c/2$; $f_1 = f_c/4$; $f_2 = f_c/12$; $f_3 = f_c/12$. Note that both Q_2 and Q_3 have the same frequencies (f_2 and f_3), but are out of phase with one another.

Example 9.12 -1

Draw the state diagram, count-sequence table, and timing diagram for a mod-12 DOWN counter.

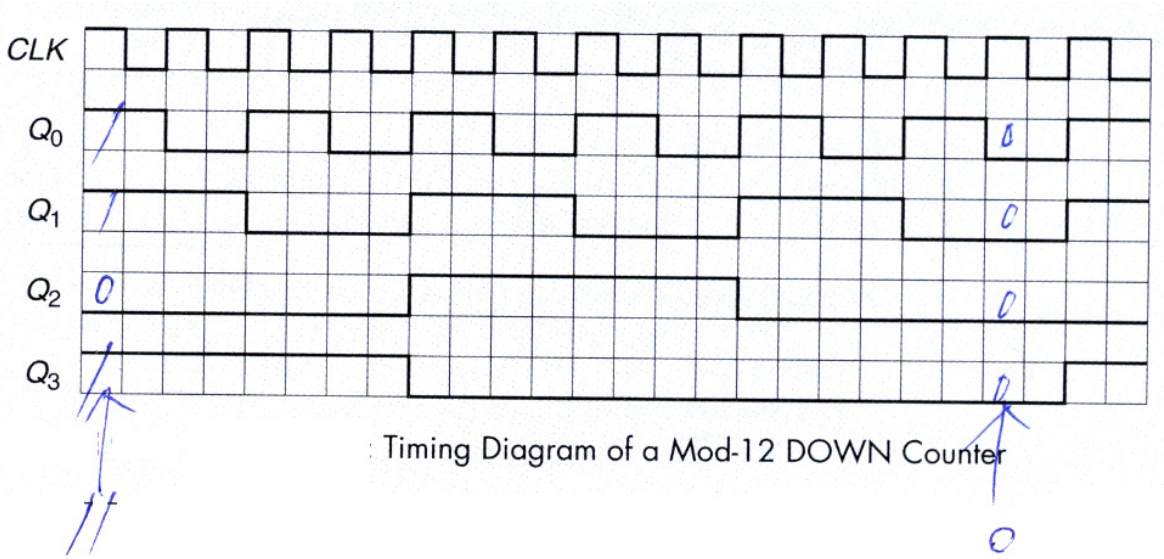


State Diagram of a Mod-12 DOWN Counter

Example 9.12 -2

Count-Sequence Table
for a Mod-12 DOWN Counter

Q_3	Q_2	Q_1	Q_0	
1	0	1	1	11
1	0	1	0	10
1	0	0	1	9
1	0	0	0	8
0	1	1	1	7
0	1	1	0	6
0	1	0	1	5
0	1	0	0	4
0	0	1	1	3
0	0	1	0	2
0	0	0	1	1
0	0	0	0	0



Timing Diagram of a Mod-12 DOWN Counter

9.2 Synchronous Counters -1

Synchronous Counter A counter whose flip-flops are all clocked by the same source and thus change in synchronization with each other.

Memory Section A set of flip-flops in a synchronous circuit that holds its present state.

Present State The current state of flip-flop outputs in a synchronous sequential circuit.

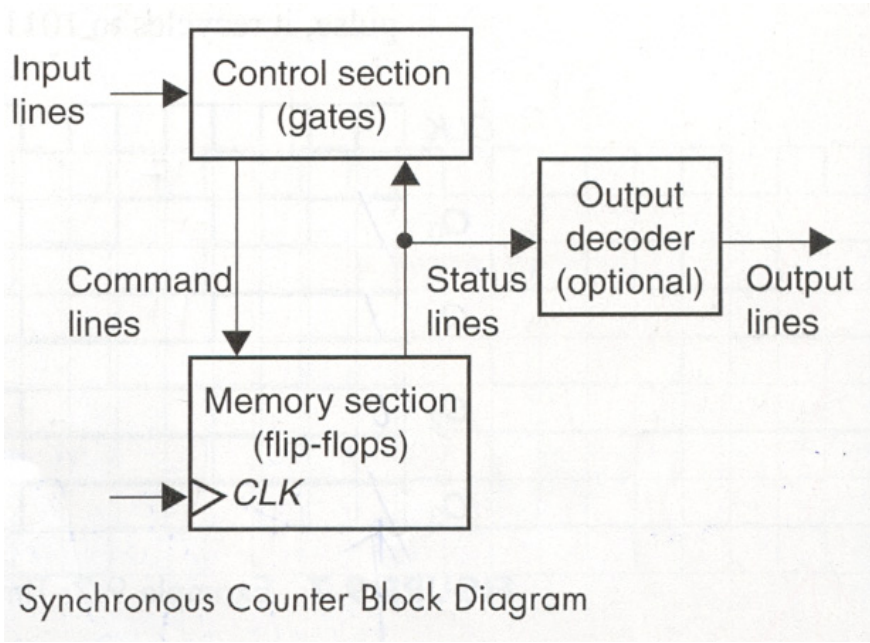
Control Section The combinational logic portion of a synchronous circuit that determines the next state of the circuit.

Next State The desired future state of flip-flop outputs in a synchronous sequential circuit after the next clock pulse is applied.

Status Lines Signals that communicate the present state of a synchronous circuit from its memory section to its control section.

Command Lines Signals that connect the control section of a synchronous circuit to its memory section and direct the circuit from its present to its next state.

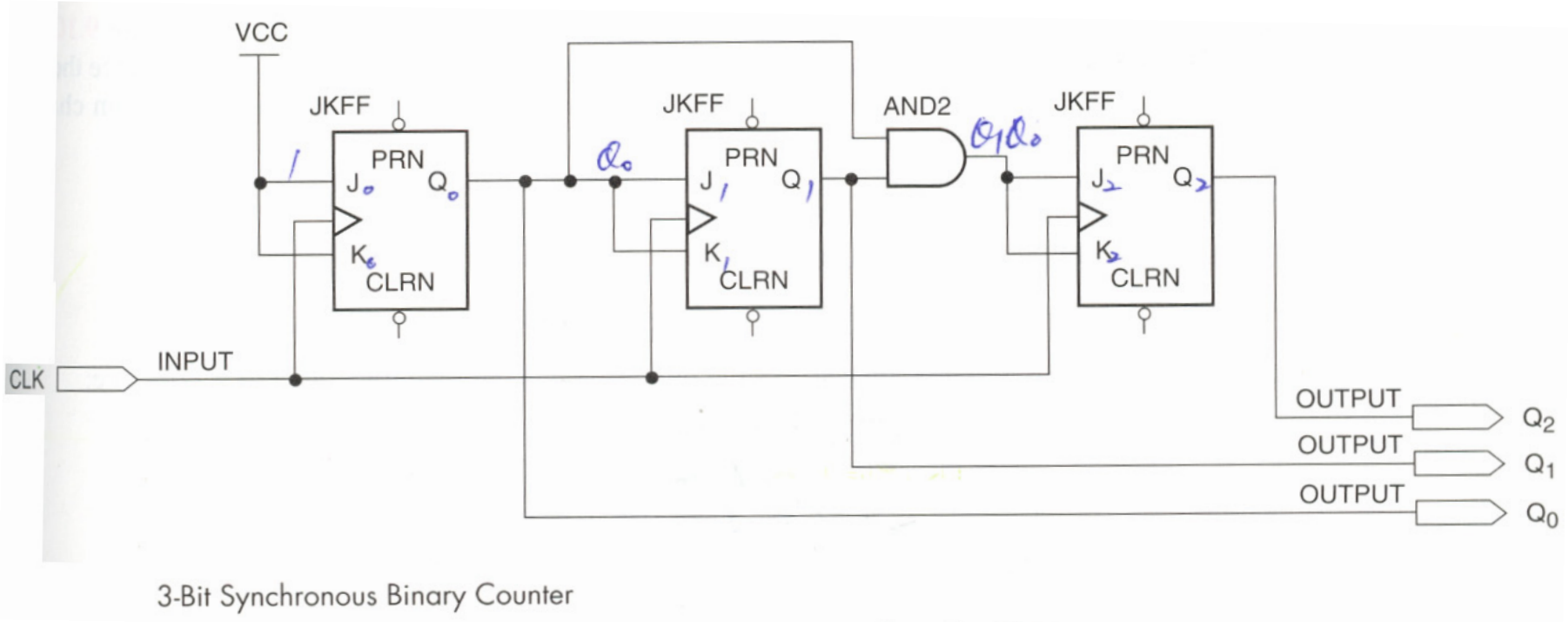
Synchronous Counters -2



Synchronous Counter Block Diagram

memory section to keep track of the **present state** of the counter and a **control section** to direct the counter to its **next state.** The memory section is a sequential circuit (flip-flops) and the control section is combinational (gates). They communicate through a set of **status lines** that go from the Q outputs of the flip-flops to the control gate inputs and **command lines** that connect the control gate outputs to the synchronous inputs (J, K, D, or T) of the flip-flops.

3-Bit Synchronous Binary Counter -1



3-Bit Synchronous Binary Counter

$$J_2 = K_2 = Q_1 \cdot Q_0$$
$$J_1 = K_1 = Q_0$$
$$J_0 = K_0 = 1$$

Function Table of a JK Flip-Flop

J	K	Q_{t+1}	Function
0	0	Q_t	No change
0	1	0	Reset
1	0	1	Set
1	1	$\bar{Q}_t$	Toggle

3-Bit Synchronous Binary Counter -2

Assume that the counter output is initially $Q_2Q_1Q_1 = 000$. Before any clock pulses are applied, the J and K inputs are at the following states:

$J_2 = K_2 = Q_1 \cdot Q_0 = 0 \cdot 0 = 0$ (No change)
$J_1 = K_1 = Q_0 = 0$ (No change)
$J_0 = K_0 = 1$ (Constant) (Toggle)

The transitions of the outputs after the clock pulse are:

Q_2: $0 \rightarrow 0$ (No change)
Q_1: $0 \rightarrow 0$ (No change)
Q_0: $0 \rightarrow 1$ (Toggle)

The new conditions of the J and K inputs are:

$J_2 = K_2 = Q_1 \cdot Q_0 = 0 \cdot 1 = 0$ (No change)
$J_1 = K_1 = Q_0 = 1$ (Toggle)
$J_0 = K_0 = 1$ (Constant) (Toggle)

The transitions of the outputs generated by the second clock pulse are:

Q_2: $0 \rightarrow 0$ (No change)
Q_1: $0 \rightarrow 1$ (Toggle)
Q_0: $1 \rightarrow 0$ (Toggle)

3-Bit Synchronous Binary Counter -3

The new output is $Q_2Q_1Q_0 = 010$, since both Q_0 and Q_1 change and Q_2 stays the same. The J and K conditions are now:

$J_2 = K_2 = Q_1 \cdot Q_0 = 1 \cdot 0 = 0$ (No change)
$J_1 = K_1 = Q_0 = 0$ (No change)
$J_0 = K_0 = 1$ (Constant) (Toggle)

The output transitions are:

Q_2: $0 \rightarrow 0$ (No change)
Q_1: $1 \rightarrow 1$ (No change)
Q_0: $0 \rightarrow 1$ (Toggle)

The output is now $Q_2Q_1Q_0 = 011$, which results in the JK conditions:

$J_2 = K_2 = Q_1 \cdot Q_0 = 1 \cdot 1 = 1$ (Toggle)
$J_1 = K_1 = Q_0 = 1$ (Toggle)
$J_0 = K_0 = 1$ (Constant) (Toggle)

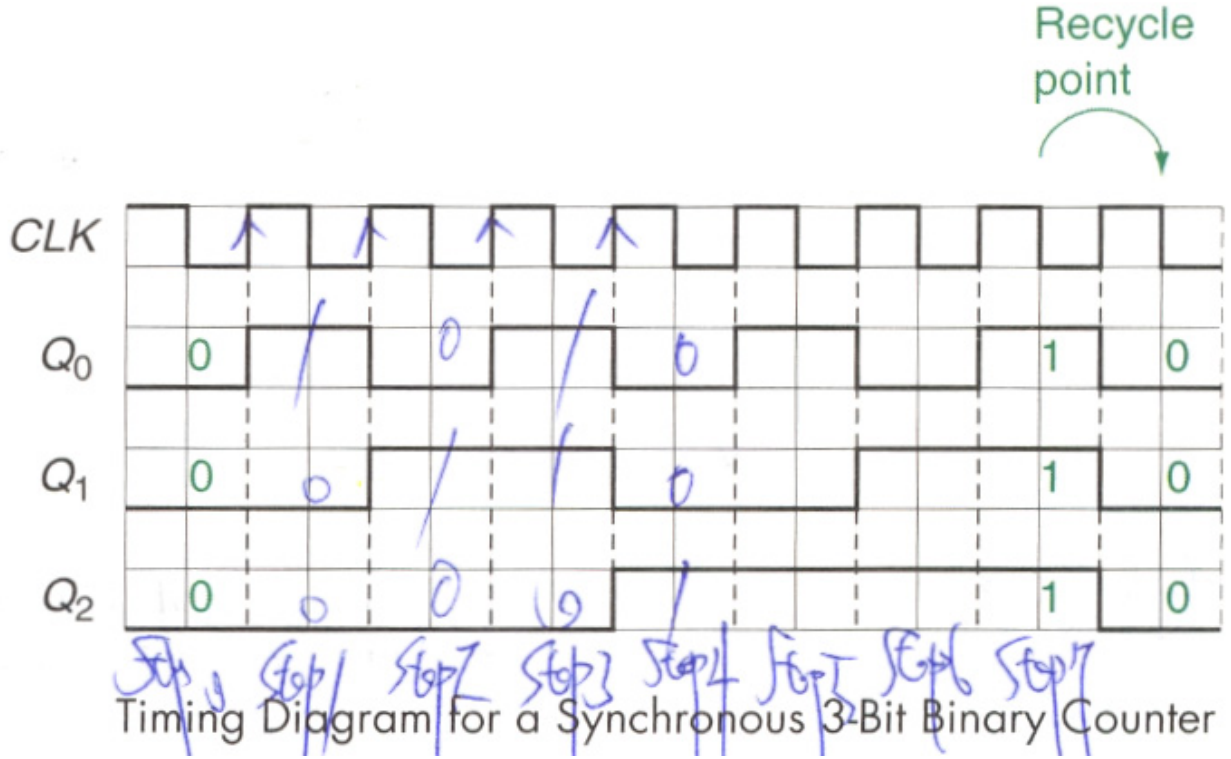
The above conditions result in output transitions:

Q_2: $0 \rightarrow 1$ (Toggle)
Q_1: $1 \rightarrow 0$ (Toggle)
Q_0: $1 \rightarrow 0$ (Toggle)

All of the outputs toggle and the new output state is $Q_2Q_1Q_0 = 100$.

3-Bit Synchronous Binary Counter -4

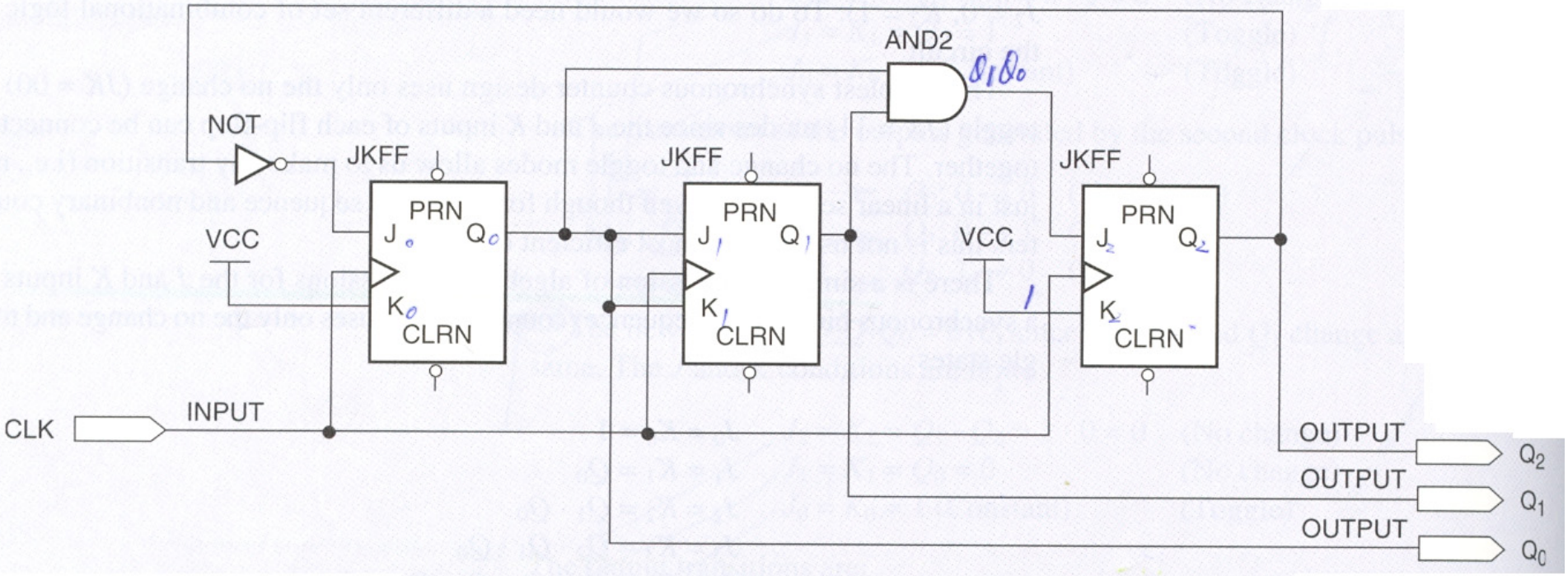
The J and K values repeat this pattern in the second half of the counter cycle (states 100 to 111). Go through the exercise of calculating the J, K, and Q values for the rest of the cycle. Compare the result with the timing diagram



Timing Diagram for a Synchronous 3-Bit Binary Counter

Example 9.3 -1

Find the count sequence of the synchronous counter shown in Figure 9.11 and, from the count sequence table, draw the timing diagram and state diagram. What is the modulus of the counter?



Synchronous Counter of Unknown Modulus

Example 9.3 -2

Solution

J and K equations are:

$J_2 = Q_1 \cdot Q_0$ $\quad J_1 = Q_0$ $\quad J_0 = \bar{Q}_2$

$K_2 = 1$ $\quad K_1 = Q_0$ $\quad K_0 = 1$

Present State			Synchronous Inputs									Next State		
Q_2	Q_1	Q_0	J_2	K_2		J_1	K_1		J_0	K_0		Q_2	Q_1	Q_0
0	0	0	0	1	(R)	0	0	(NC)	1	1	(T)	0	0	1
0	0	1	0	1	(R)	1	1	(T)	1	1	(T)	0	1	0
0	1	0	0	1	(R)	0	0	(NC)	1	1	(T)	0	1	1
0	1	1	1	1	(T)	1	1	(T)	1	1	(T)	1	0	0
1	0	0	0	1	(R)	0	0	(NC)	0	1	(R)	0	0	0

(R) = Reset

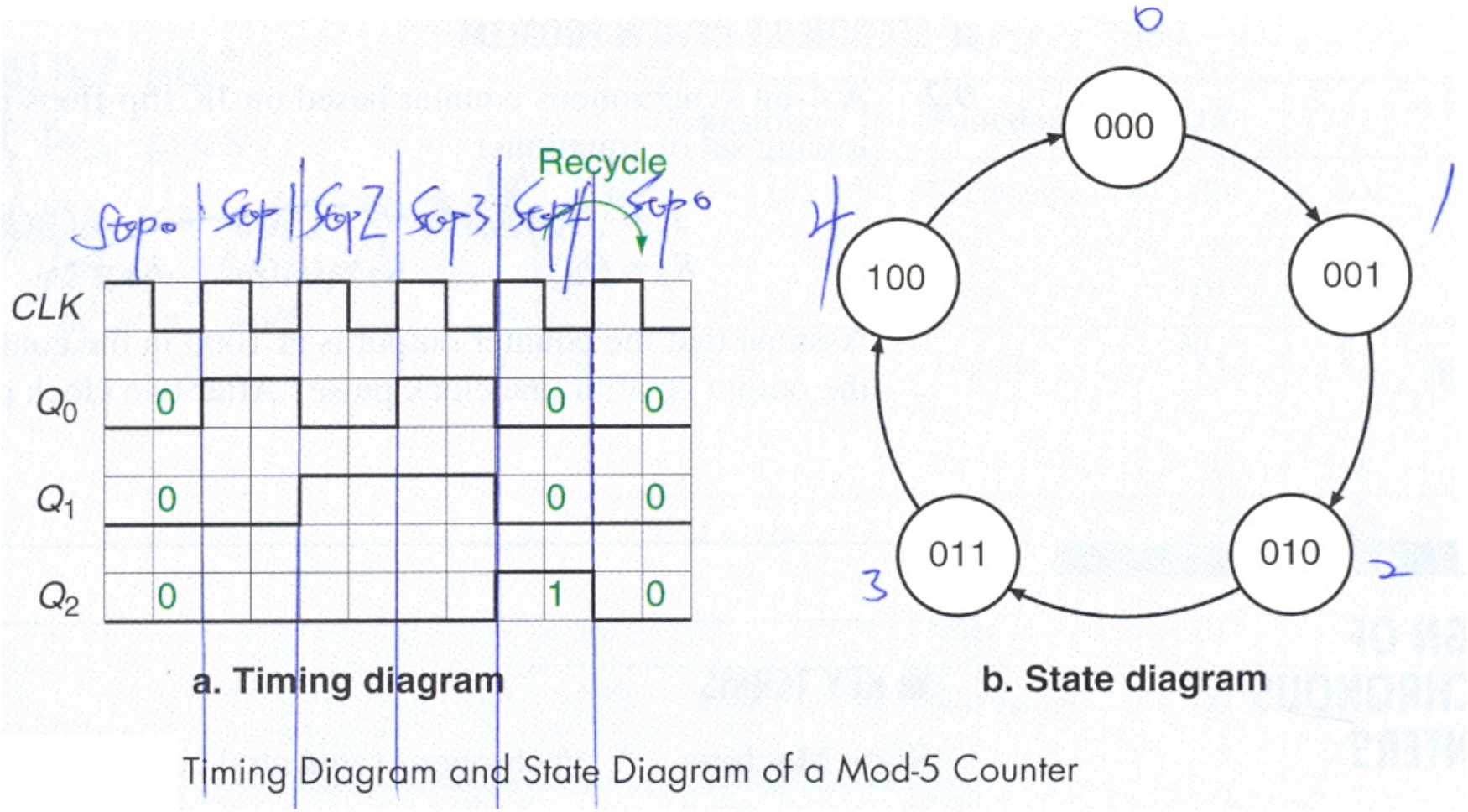
(T) = Toggle

(NC) = No change

repeat = 0 0 0

Since there are five unique output states, the counter's modulus is 5.

Example 9.3 -3



Timing Diagram and State Diagram of a Mod-5 Counter

Only 5 of a possible 8 output states: truncated sequence counter.

Example 9.4 -1

Extend the analysis of the counter in Example 9.3 to include its unused states. Redraw the counter's state diagram to show how these unused states enter the main sequence (if they do).

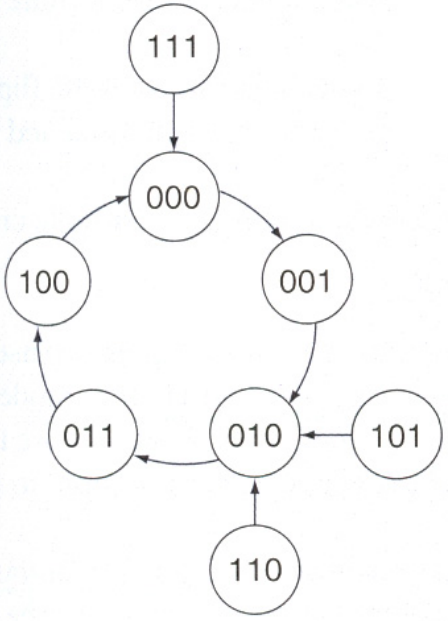
Solution

The synchronous input equations are:

$J_2 = Q_1 \cdot Q_0$ $\quad J_1 = Q_0$ $\quad J_0 = \bar{Q}_2$

$K_2 = 1$ $\quad K_1 = Q_0$ $\quad K_0 = 1$

The unused states are $Q_2Q_1Q_0$ = 101, 110, and 111. Table 9.6 shows the transitions made by the unused states. Figure 9.13 shows the completed state diagram.



Complete State Diagram

TABLE 9.6 State Table for Mod-5 Counter Including Unused States

Present State			Synchronous Inputs									Next State		
Q_2	Q_1	Q_0	J_2	K_2		J_1	K_1		J_0	K_0		Q_2	Q_1	Q_0
0	0	0	0	1	(R)	0	0	(NC)	1	1	(T)	0	0	1
0	0	1	0	1	(R)	1	1	(T)	1	1	(T)	0	1	0
0	1	0	0	1	(R)	0	0	(NC)	1	1	(T)	0	1	1
0	1	1	1	1	(T)	1	1	(T)	1	1	(T)	1	0	0
1	0	0	0	1	(R)	0	0	(NC)	0	1	(R)	0	0	0
1	0	1	0	1	(R)	1	1	(T)	0	1	(R)	0	1	0
1	1	0	0	1	(R)	0	0	(NC)	0	1	(R)	0	1	0
1	1	1	1	1	(T)	1	1	(T)	0	1	(R)	0	0	0

9.3 Design of Synchronous Counters

KEY TERMS

State Machine A synchronous sequential circuit.

Excitation Table A table showing the required input conditions for every possible transition of a flip-flop output.

Function Table of a JK Flip-Flop

J	K	Q_{t+1}	Function
0	0	Q_t	No change
0	1	0	Reset
1	0	1	Set
1	1	$\bar{Q}_t$	Toggle

TABLE 9.7 JK Flip-Flop Excitation Table

Transition	Function	JK	
$0 \rightarrow 0$	No change or reset	00 01	0X
$0 \rightarrow 1$	Toggle or set	11 10	1X
$1 \rightarrow 0$	Toggle or reset	11 01	X1
$1 \rightarrow 1$	No change or set	00 10	X0

TABLE 9.8 Condensed Excitation Table for a JK Flip-Flop

Transition	JK
$0 \rightarrow 0$	0X
$0 \rightarrow 1$	1X
$1 \rightarrow 0$	X1
$1 \rightarrow 1$	X0

A flip-flop excitation table shows all possible transitions of a flip-flop output and the synchronous input levels needed to effect these transitions.

Design of a Synchronous Mod-12 Counter -1

1. *Define the problem.* The circuit must count in binary sequence from 0000 to 1011 and repeat. The output progresses by 1 for each applied clock pulse. Since the outputs are 4-bit numbers, we require 4 flip-flops.
2. *Draw a state diagram.* The state diagram for this problem is shown in Figure 9.14.

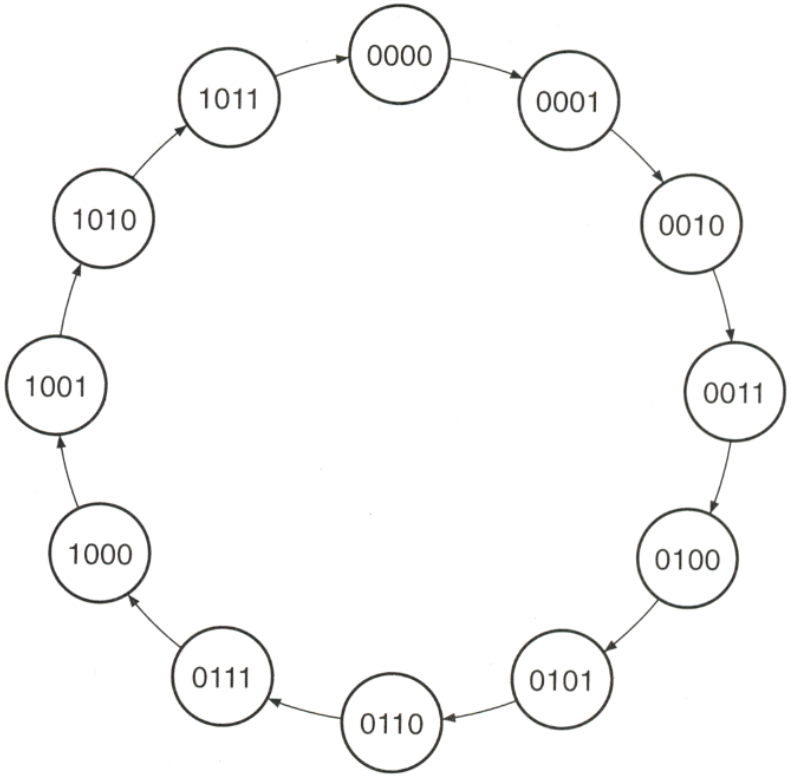


FIGURE 9.14 State Diagram for a Mod-12 Counter

Design of a Synchronous Mod-12 Counter -2

3. *Make a state table showing each present state and the corresponding next state.*
4. *Use flip-flop excitation tables to fill in the J and K entries in the state table.* Table 9.9 shows the combined result of steps 3 and 4. Note that all present states are in binary order.

TABLE 9.9 State Table for a Mod-12 Counter

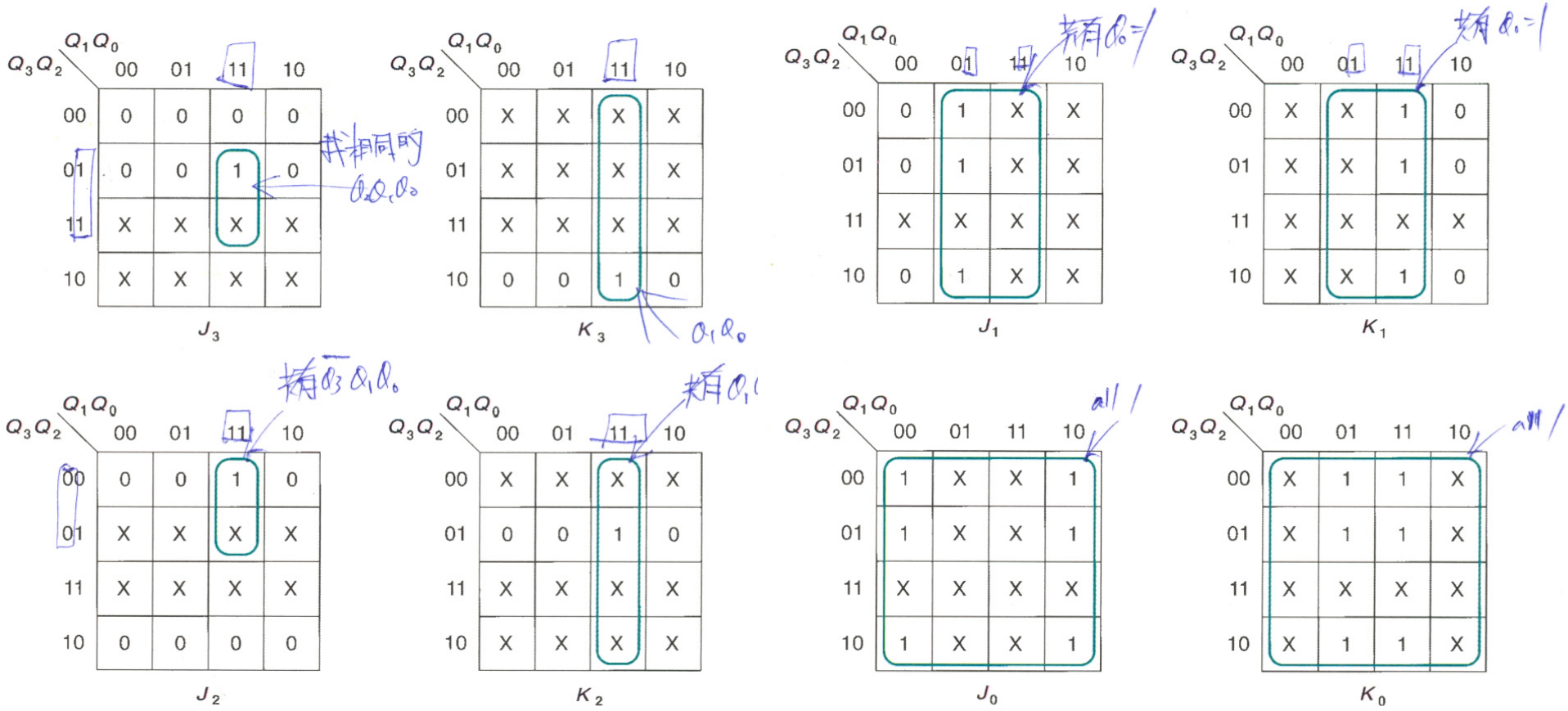
Present State				Next State				Synchronous Inputs							
Q_3	Q_2	Q_1	Q_0	Q_3	Q_2	Q_1	Q_0	J_3	K_3	J_2	K_2	J_1	K_1	J_0	K_0
0	0	0	0	0	0	0	1	0	X	0	X	0	X	1	X
0	0	0	1	0	0	1	0	0	X	0	X	1	X	X	1
0	0	1	0	0	0	1	1	0	X	0	X	X	0	1	X
0	0	1	1	0	1	0	0	0	X	1	X	X	1	X	1
0	1	0	0	0	1	0	1	0	X	X	0	0	X	1	X
0	1	0	1	0	1	1	0	0	X	X	0	1	X	X	1
0	1	1	0	0	1	1	1	0	X	X	0	X	0	1	X
0	1	1	1	1	0	0	0	1	X	X	1	X	1	X	1
1	0	0	0	1	0	0	1	X	0	0	X	0	X	1	X
1	0	0	1	1	0	1	0	X	0	0	X	1	X	X	1
1	0	1	0	1	0	1	1	X	0	0	X	X	0	1	X
1	0	1	1	0	0	0	0	X	1	0	X	X	1	X	1
1	1	0	0	X	X	X	X	X	X	X	X	X	X	X	X
1	1	0	1	X	X	X	X	X	X	X	X	X	X	X	X
1	1	1	0	X	X	X	X	X	X	X	X	X	X	X	X
1	1	1	1	X	X	X	X	X	X	X	X	X	X	X	X

TABLE 9.8 Condensed Excitation Table for a JK Flip-Flop

Transition	*JK*
$0 \rightarrow 0$	0X
$0 \rightarrow 1$	1X
$1 \rightarrow 0$	X1
$1 \rightarrow 1$	X0

Design of a Synchronous Mod-12 Counter -3

5. *Simplify the Boolean expression for each input.*

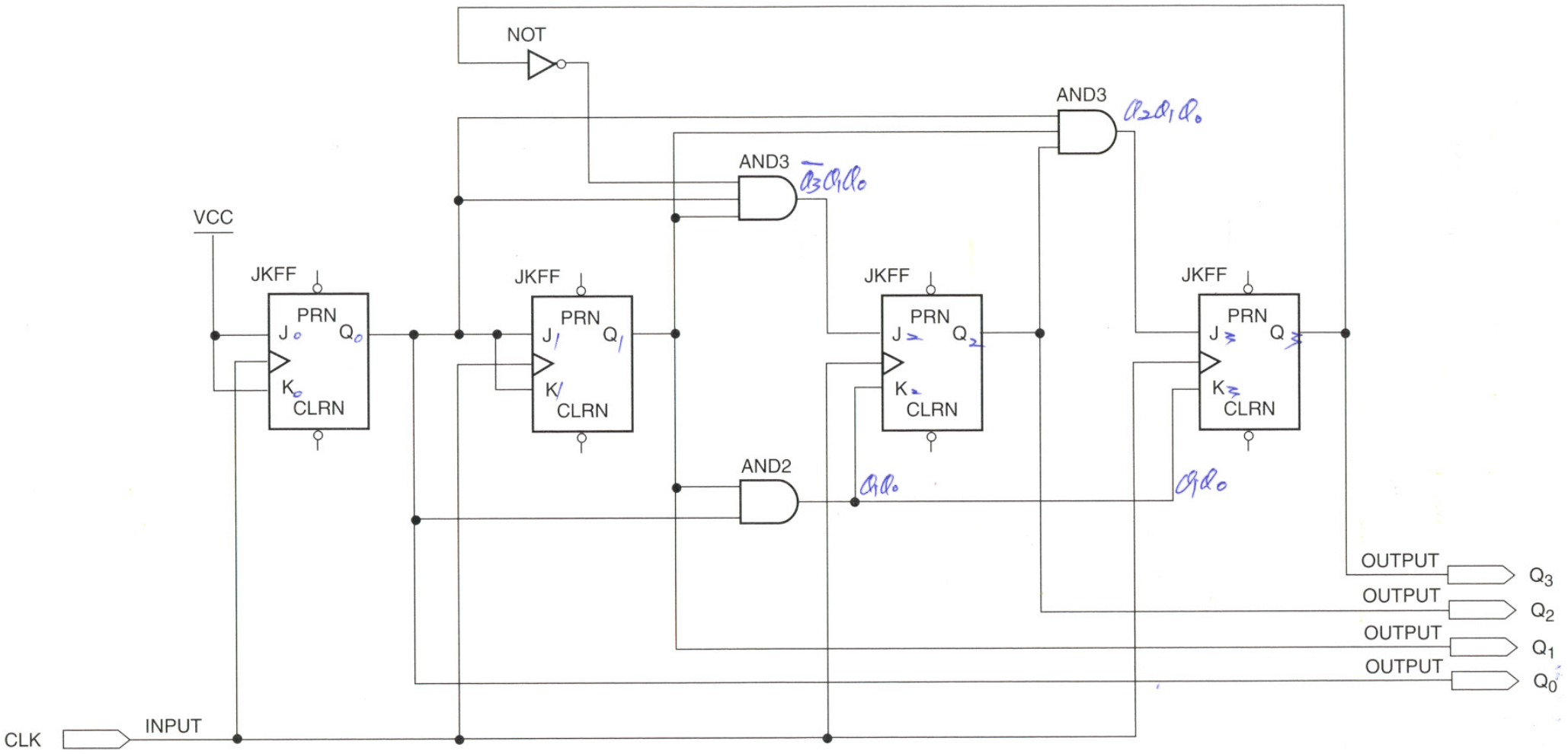


$J_0 = 1$
$K_0 = 1$
$J_1 = Q_0$
$K_1 = Q_0$
$J_2 = \bar{Q}_3 Q_1 Q_0$
$K_2 = Q_1 Q_0$
$J_3 = Q_2 Q_1 Q_0$
$K_3 = Q_1 Q_0$

K-Map Simplification of Table 9.9

Design of a Synchronous Mod-12 Counter -4

6. *Draw the required logic circuit.*



Synchronous Mod-12 Counter

Design of a Synchronous Mod-12 Counter -5

❑ Unused States

We have assumed that states 1100 to 1111 will never occur in the operation of the mod-12 counter. This is normally the case, but when the circuit is powered up, there is no guarantee that the flip-flops will be in any particular state.

If a counter powers up in an unused state, the circuit should enter the main sequence after one or more clock pulses.

$J_2 = \bar{Q}_3 Q_1 Q_0$ $\quad$ $J_0 = 1$
$K_2 = Q_1 Q_0$ $\quad$ $K_0 = 1$
$J_3 = Q_2 Q_1 Q_0$ $\quad$ $J_1 = Q_0$
$K_3 = Q_1 Q_0$ $\quad$ $K_1 = Q_0$

Unused States in a Mod-12 Counter

Present State				Synchronous Inputs								Next State			
Q_3	Q_2	Q_1	Q_0	J_3	K_3	J_2	K_2	J_1	K_1	J_0	K_0	Q_3	Q_2	Q_1	Q_0
1	1	0	0	0	0	0	0	0	0	1	1	1	1	0	1
1	1	0	1	0	0	0	0	1	1	1	1	1	1	1	0
1	1	1	0	0	0	0	0	0	0	1	1	1	1	1	1
1	1	1	1	1	1	0	1	1	1	1	1	0	0	0	0

Design of a Synchronous Mod-12 Counter -6

Figure 9.17 shows the complete state diagram for the designed mod-12 counter. If the counter powers up in an unused state, it will enter the main sequence in no more than four clock pulses.

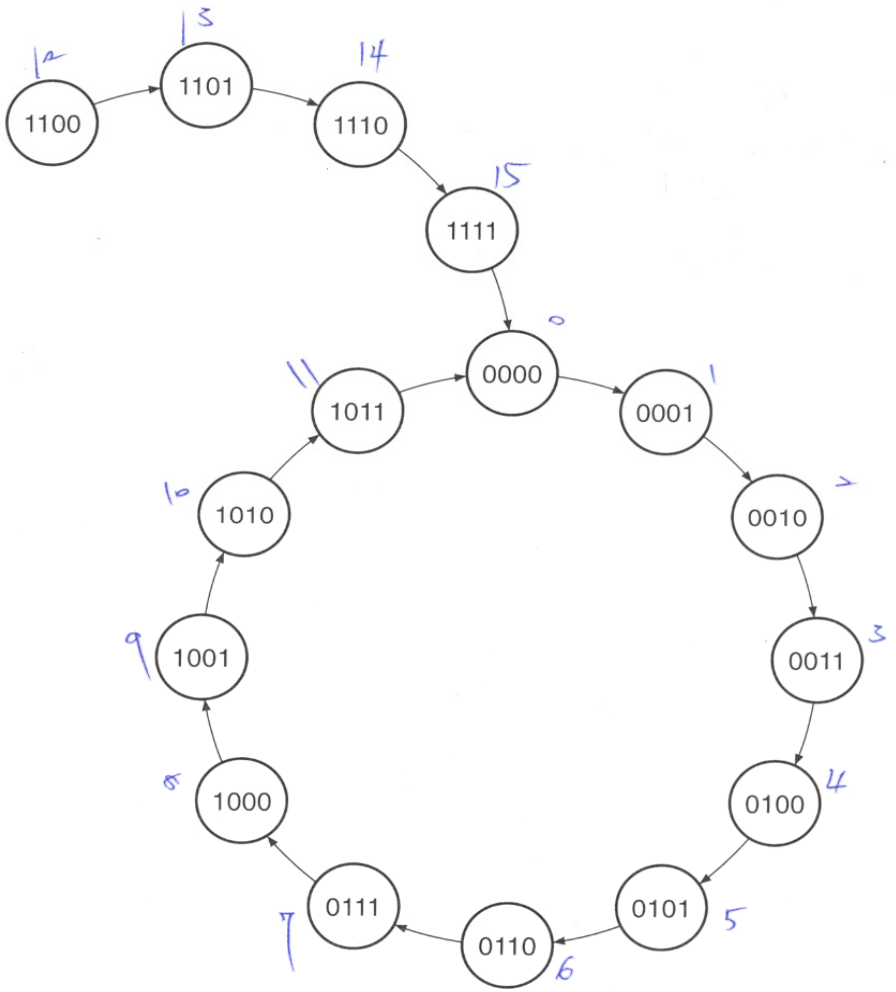


FIGURE 9.17 Complete State Diagram of Mod-12 Counter in Figure 9.16

Example 9.5 -1

Derive the synchronous input equations of a 4-bit synchronous binary counter based on D flip-flops. Draw the corresponding counter circuit.

■ Solution

The first step in the counter design is to derive the excitation table of a D flip-flop. Recall that Q follows D when the flip-flop is clocked. Therefore the next state of Q is the same as the input D for any transition. This is illustrated in Table 9.11.

TABLE 9.11 Excitation Table of a D Flip-Flop

Output Transition	Input D
$0 \rightarrow 0$	0
$0 \rightarrow 1$	1
$1 \rightarrow 0$	0
$1 \rightarrow 1$	1

Example 9.5 -2

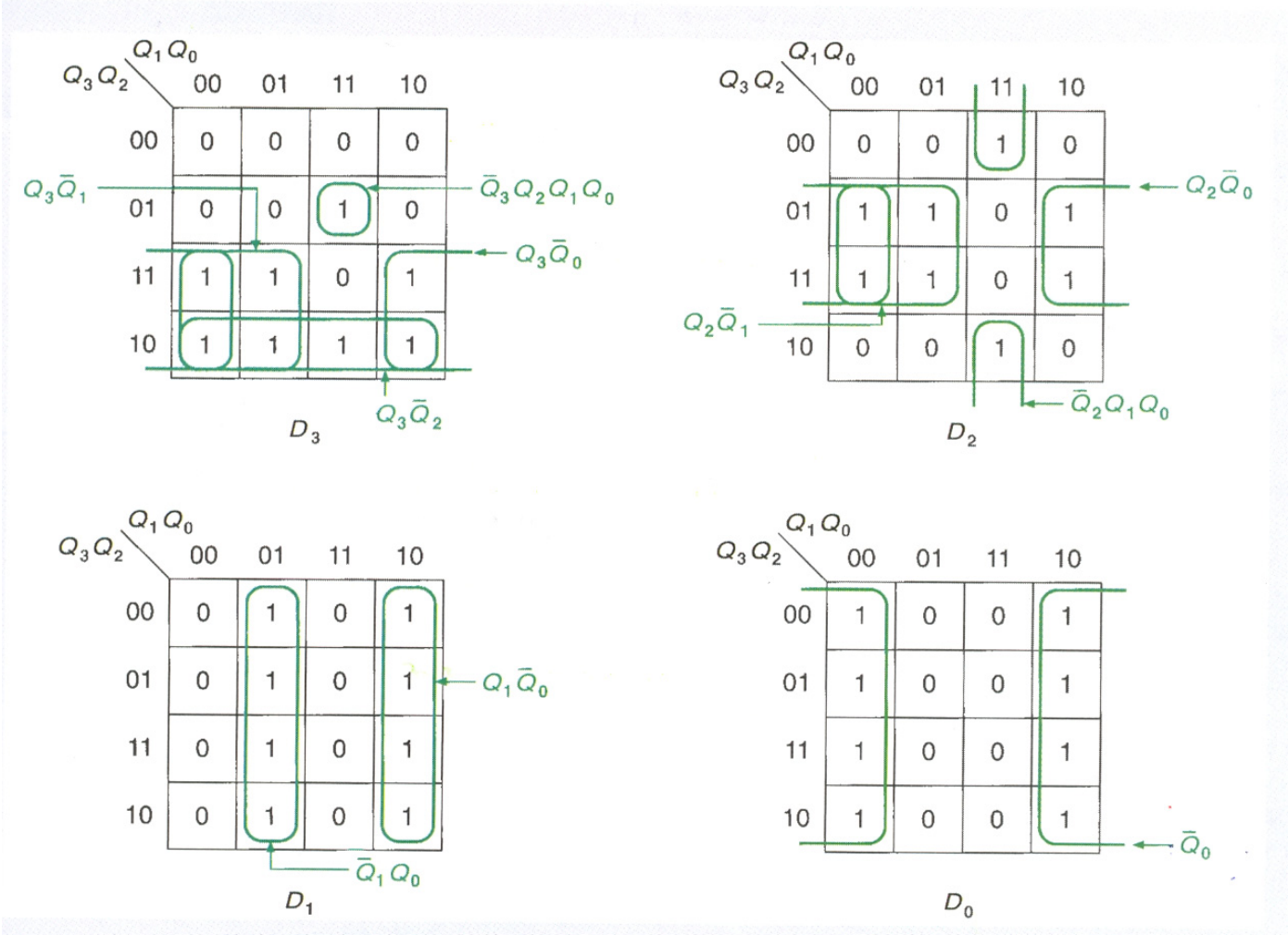
- Step2: State table

TABLE 9.12 State Table for a 4-Bit Binary Counter

Present State				Next State				Synchronous Inputs			
Q_3	Q_2	Q_1	Q_0	Q_3	Q_2	Q_1	Q_0	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	1	0	0	1	0	0	0	1	0
0	0	1	0	0	0	1	1	0	0	1	1
0	0	1	1	0	1	0	0	0	1	0	0
0	1	0	0	0	1	0	1	0	1	0	1
0	1	0	1	0	1	1	0	0	1	1	0
0	1	1	0	0	1	1	1	0	1	1	1
0	1	1	1	1	0	0	0	1	0	0	0
1	0	0	0	1	0	0	1	1	0	0	1
1	0	0	1	1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	1	1	0	1	1
1	0	1	1	1	1	0	0	1	1	0	0
1	1	0	0	1	1	0	1	1	1	0	1
1	1	0	1	1	1	1	0	1	1	1	0
1	1	1	0	1	1	1	1	1	1	1	1
1	1	1	1	0	0	0	0	0	0	0	0

Example 9.5 -3

- Step3: K-Map



K-Maps for a 4-Bit Counter Based on D Flip-Flops

Example 9.5 -4

The simplified equations are:

$$D_3 = \bar{Q}_3 Q_2 Q_1 Q_0 + Q_3 \bar{Q}_2 + Q_3 \bar{Q}_1 + Q_3 \bar{Q}_0$$
$$D_2 = \bar{Q}_2 Q_1 Q_0 + Q_2 \bar{Q}_1 + Q_1 \bar{Q}_0$$
$$D_1 = \bar{Q}_1 Q_0 + Q_1 \bar{Q}_0$$
$$D_0 = \bar{Q}_0$$

the equation for D_3 can be rewritten using DeMorgan's theorem ($\bar{x} + \bar{y} + \bar{z} = \overline{xyz}$) and our knowledge of Exclusive OR (XOR) functions ($\bar{x}y + x\bar{y} = x \oplus y$).

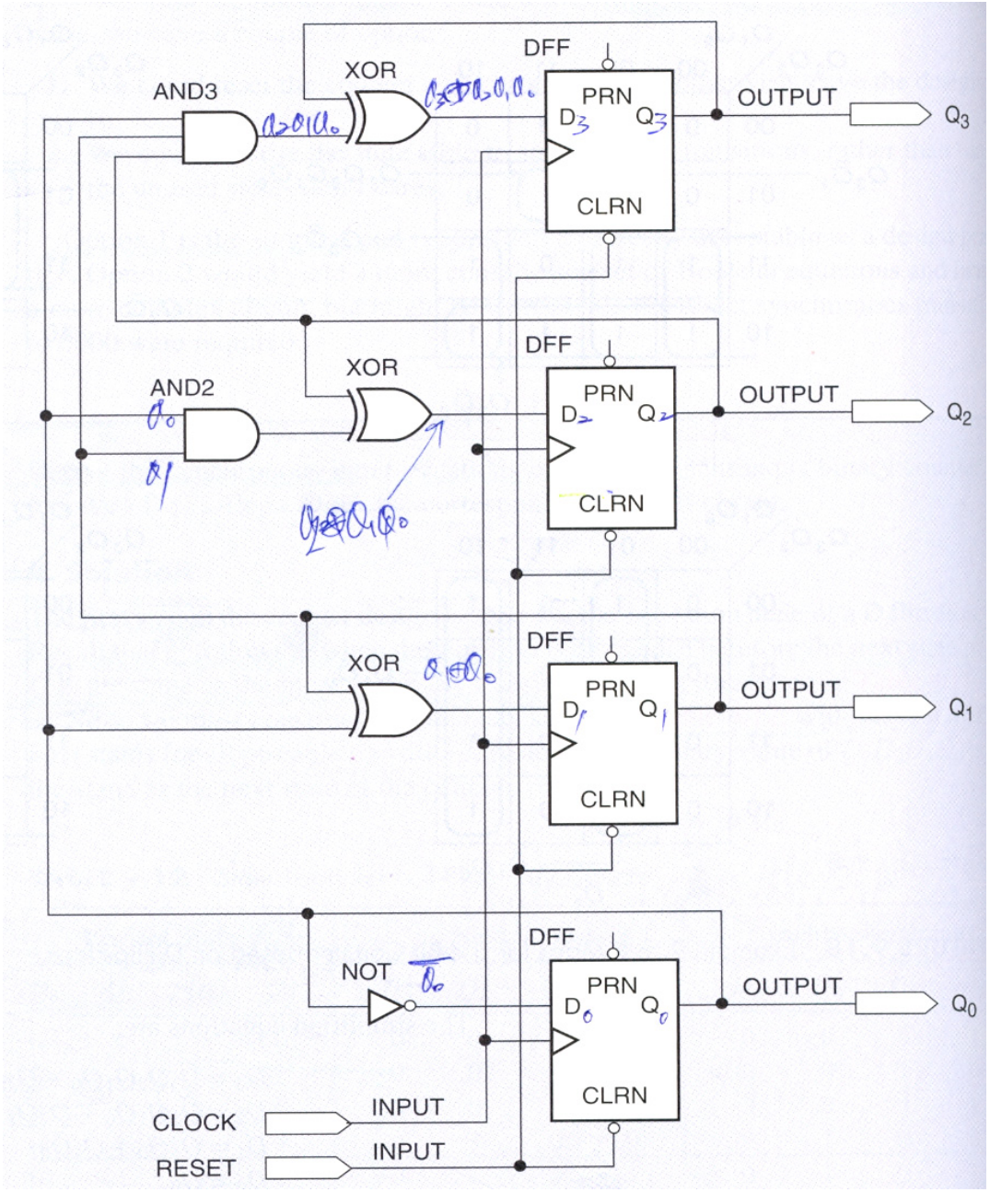
$$\begin{aligned} D_3 &= \bar{Q}_3 Q_2 Q_1 Q_0 + Q_3 \bar{Q}_2 + Q_3 \bar{Q}_1 + Q_3 \bar{Q}_0 \\ &= \bar{Q}_3 (Q_2 Q_1 Q_0) + Q_3 (\bar{Q}_2 + \bar{Q}_1 + \bar{Q}_0) \\ &= \bar{Q}_3 (Q_2 Q_1 Q_0) + Q_3 \, \overline{(Q_2 Q_1 Q_0)} \\ &= Q_3 \oplus Q_2 Q_1 Q_0 \end{aligned}$$

We can write similar equations for the other D inputs as follows:

$$D_2 = Q_2 \oplus Q_1 Q_0$$
$$D_1 = Q_1 \oplus Q_0$$
$$D_0 = Q_0 \oplus 1 = \bar{Q}_0$$

Example 9.5 -5

- Step 4: Circuit



4-bit Counter Using D Flip-Flops

9.4 Programming Binary Counters for CPLDs

Behavioral Description of Counters -1

The following VHDL code shows the behavioral description of a simple 8-bit counter **simple_int_counter.vhd** with asynchronous clear.

```
-- simple_int_counter.vhd
-- 8-bit synchronous counter with asynchronous clear.
-- Uses INTEGER type for counter output.
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
```

Behavioral Description of Counters -2

```
ENTITY simple_int_counter IS
    PORT(
        clock : IN  STD_LOGIC;
        reset : IN  STD_LOGIC;
        q     : OUT INTEGER RANGE 0 TO 255);
END simple_int_counter;

ARCHITECTURE counter OF simple_int_counter IS
    BEGIN
    PROCESS (clock, reset)
       VARIABLE count : INTEGER RANGE 0 TO 255;
    BEGIN
       IF (reset = '0') THEN
           count := 0;
       ELSE
          IF (clock'EVENT AND clock = '1') THEN
              count := count + 1;
          END IF;
       END IF;
       q <= count;
    END PROCESS;
END counter;
```

Behavioral Description of Counters -3

This VHDL code relies on three main constructs to perform the count function:

- a PROCESS statement to monitor the two inputs, **clock** and **reset,** which control the state of the counter.
- a variable, called **count,** to hold the present value of the counter.
- an IF statement to evaluate the **clock** and **reset** inputs to determine whether the counter should increment or clear.

Using STD_Logic Types for Counter -1

```
-- simple_std_counter.vhd
-- 8-bit synchronous counter with asynchronous clear.
-- Uses STD_LOGIC_VECTOR type for counter output.
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY simple_std_counter IS
   PORT(
       clock : IN     STD_LOGIC;
       reset : IN     STD_LOGIC;
       q     : BUFFER STD_LOGIC_VECTOR(7 downto 0));
END simple_std_counter;

  ARCHITECTURE counter OF simple_std_counter IS
  BEGIN
    PROCESS (clock, reset)
    BEGIN
       IF (reset = '0') THEN
           q  <= (others => '0');
       ELSE
           IF (clock'EVENT AND clock = '1') THEN
                  q <= q + 1;
           END IF;
       END IF;
    END PROCESS;
  END counter;
```

Using STD_Logic Types for Counter -2

- Output **q** is defined as type STD_LOGIC_VECTOR. The width of the vector can be defined explicitly or with a parameter in a GENERIC clause.
- Eliminate the variable, **count.** Operate on the port, **q,** directly.
- The statement `q <= q + 1;` indicates that **q** acts as both an output (left side of the assignment) and input (right side). To make this possible, **q** must be of mode BUFFER, rather than of mode OUT.
- To update the value of **q,** we add 1 to its present value, which means that we must be able to treat a port of type STD_LOGIC_VECTOR as an unsigned integer. You can't add anything to a STD_LOGIC_VECTOR, which is only a string of bits in a particular sequence. To allow it to be treated as an integer, you require the routines contained in the **std_logic_unsigned** package, which is part of the **ieee** library.
- For the reset function, we could write `q <= "00000000";` for an 8-bit counter. It is more convenient to use the notation `q <= (others => '0');` This statement can be applied to a counter of any width. It can be loosely translated as, "set each bit of **q** to '0', regardless of how many bits that actually is."

LPM Counters in VHDL -1

- The Altera LPM (Library of Parameterized Modules) counter can be used to create counter designs in VHDL.
- This is a **structured design approach** that uses the LPM-counter as a component in a hierarchy.
- The LPM counter is instantiated in the structured design.
- The basic parameters of the LPM counter, such as width, are defined with a generic map.
- The port map is used to connect LPM counter I/O to the actual VHDL design entity.

LPM Counters in VHDL -2

```
-- simple_lpm_counter.vhd
-- Eight-bit binary counter based on a component
--   from the Library of Parameterized Modules (LPM)
-- Counter has an active-LOW asynchronous clear,

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
LIBRARY lpm;
USE lpm.lpm_components.ALL;

ENTITY simple_lpm_counter IS
   PORT(
        clk, clear : IN  STD_LOGIC;
        q_out      : OUT STD_LOGIC_VECTOR (7 downto 0));
END simple_lpm_counter;
```

```
ARCHITECTURE count OF simple_lpm_counter IS
   SIGNAL clrn : STD_LOGIC;
BEGIN
-- Instantiate 8-bit counter
   count: lpm_counter
      GENERIC MAP (LPM_WIDTH => 8)
      PORT MAP ( clock       => clk,
                 aclr        => clrn,
                 q           => q_out(7 downto 0));

   clrn   <= not clear;
END count;
```

Internal signal for active-LOW clear

Internal clear mapped to asynch. clear port of counter

Input port inverted and mapped to internal clear signal

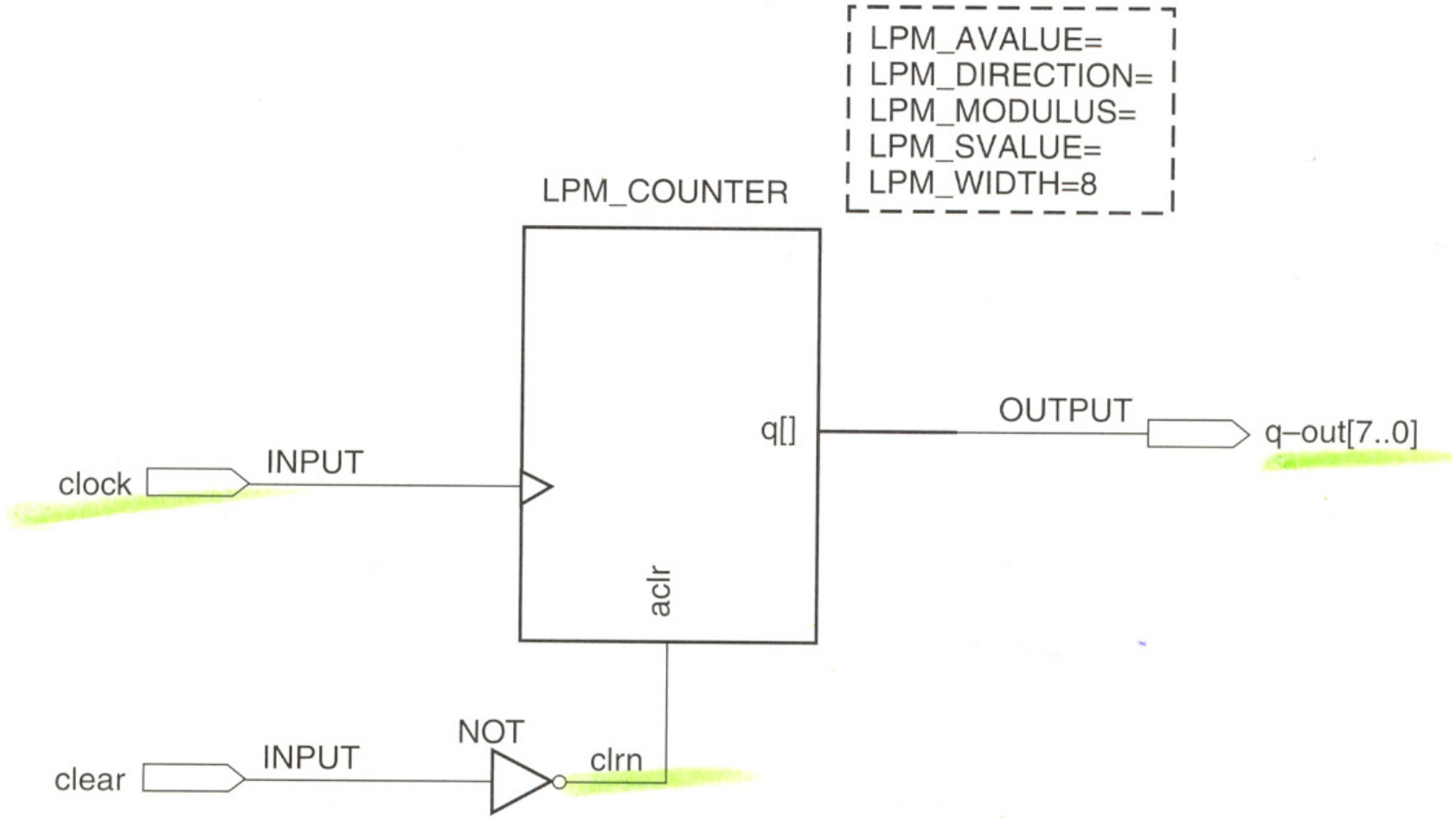
LPM Counters in VHDL -3

```
__instance_name: __component_name
   GENERIC MAP (__parameter_name => __parameter_value,
                __parameter_name => __parameter_value)
   PORT MAP (__component_port => __connect_port,
             __component_port => __connect_port);
```

The component name is the name of the LPM component. Parameter names are those defined in the LPM component, such as LPM_WIDTH. Parameter values are those values assigned in the instance of the component. Component ports are the LPM port names. Connect ports are the names of identifiers declared in the entity or as ports, signals, or variables.

If we want to invert the active level of an LPM input port, we must use a signal assignment statement (e.g., `clrn <= not clear;`).

LPM Counters in VHDL -4

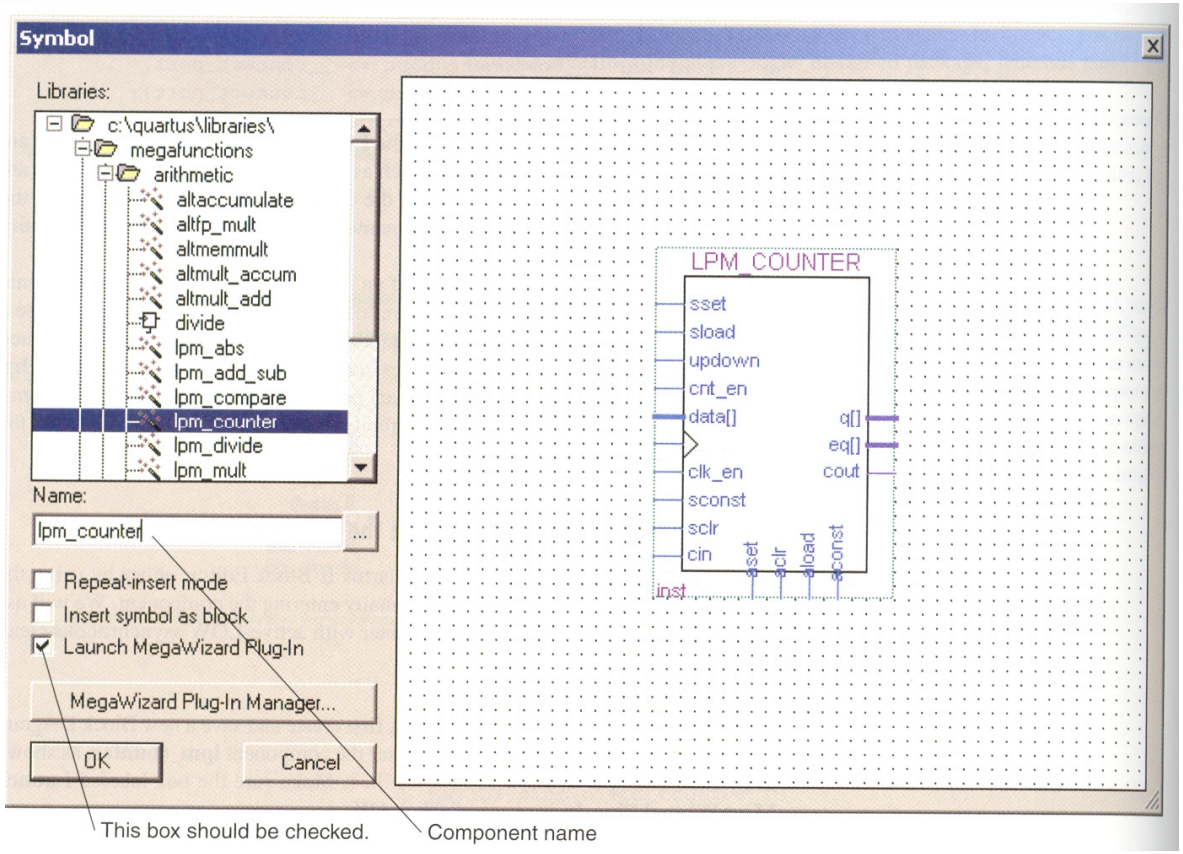


Graphic Equivalent of an LPM Counter with Active-LOW Clear

Entering Simple LPM Counters in Quartus II -1

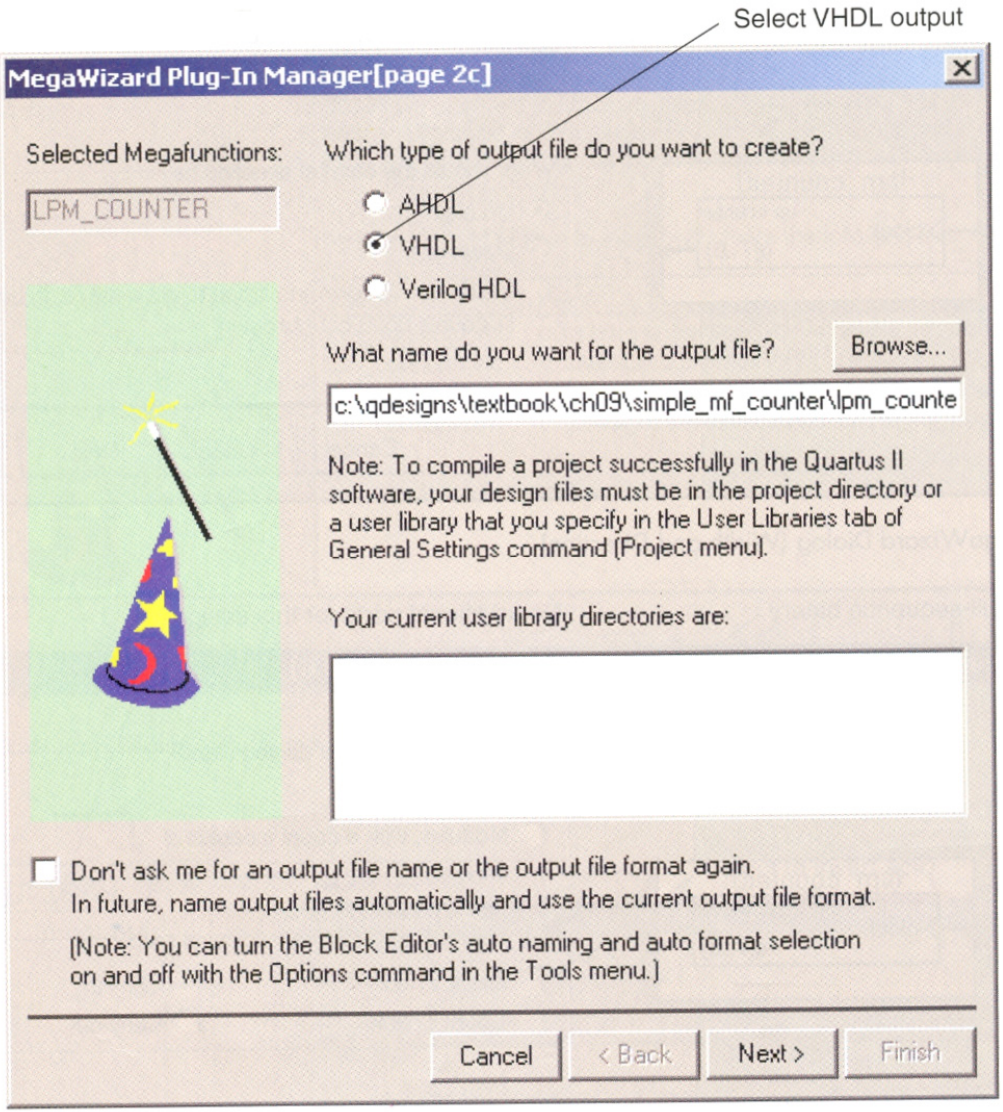
MegaWizard Plug-In Manager

first create and save a new Block Diagram File, and use it to make a new project. Enter the component **lpm_counter,** as shown in **Symbol** dialog box, shown in Figure 9.21. Make sure the box labeled **Launch MegaWizard Plug-In** is *checked.* Click **OK.**



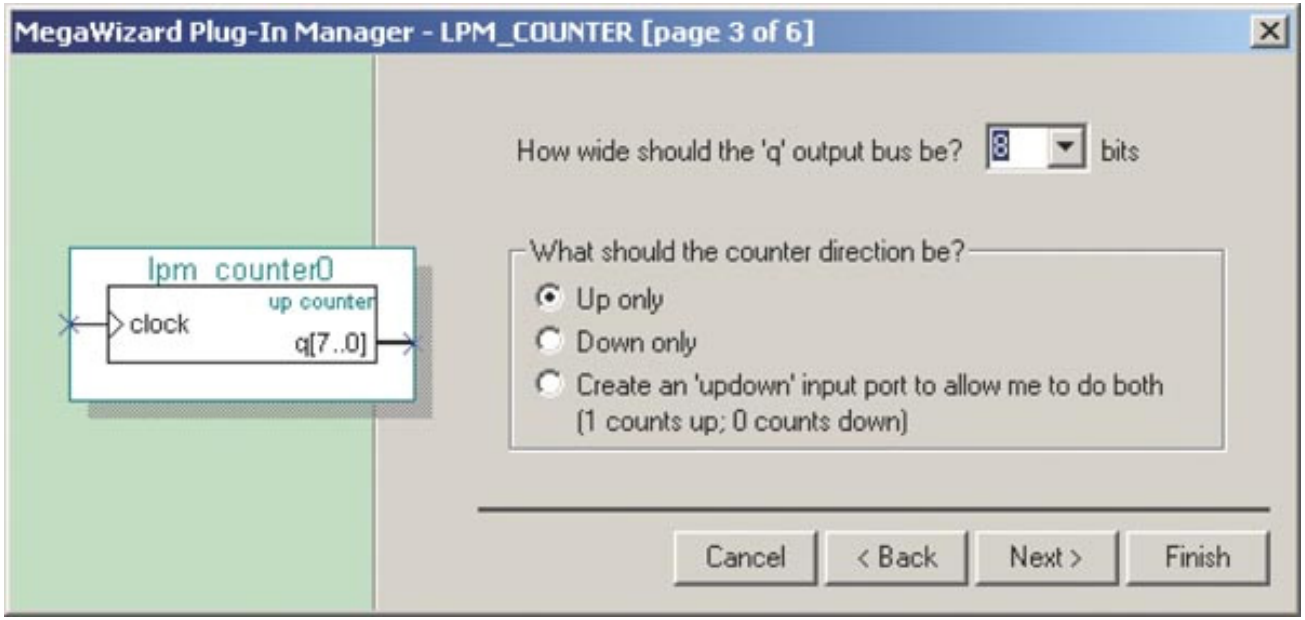
Entering Simple LPM Counters in Quartus II -2

screen shown in Figure 9.22, choose **VHDL** and click **Next.**



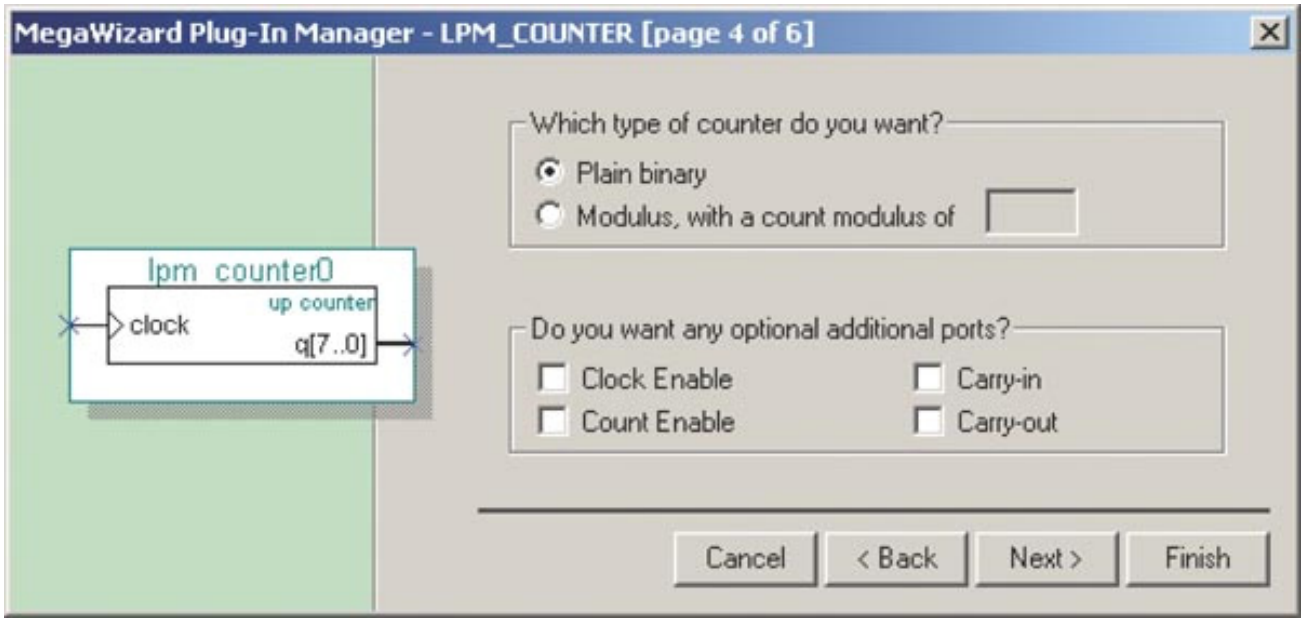
Entering Simple LPM Counters in Quartus II -3

The box shown in Figure 9.23 selects the width of the counter output and the count direction. Select **8 bits** and **Up only.**



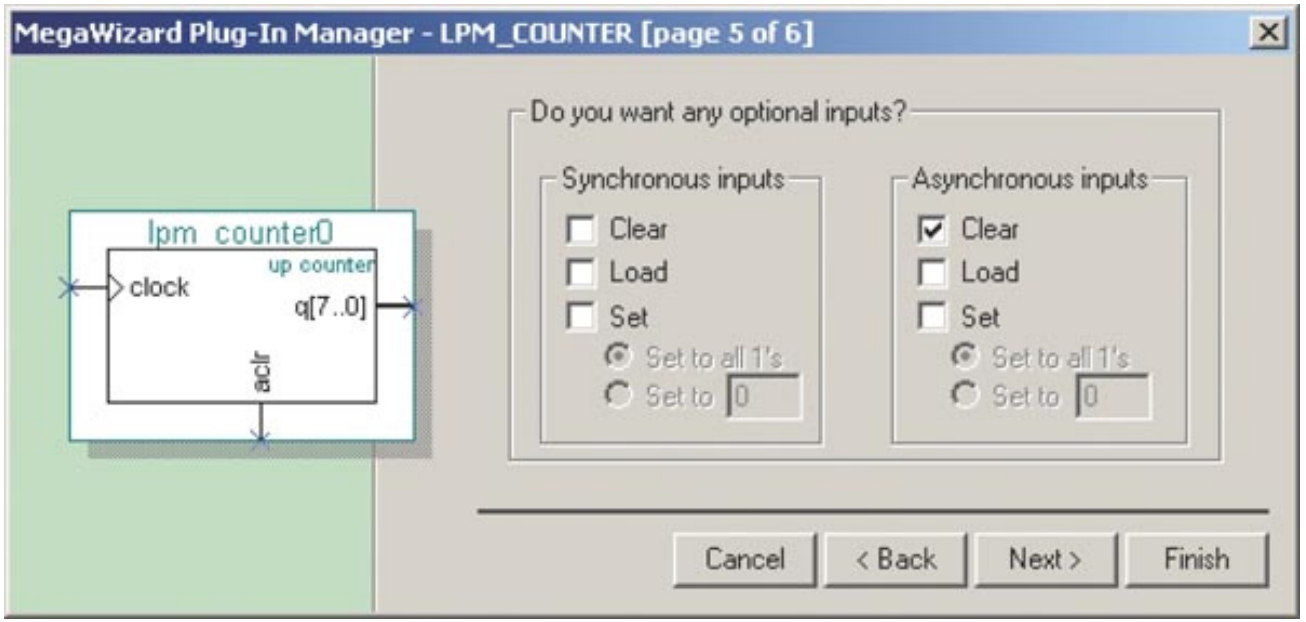
Entering Simple LPM Counters in Quartus II -4

In the screen shown in Figure 9.24, select the type of counter as **Plain binary** (i.e., full-sequence, not truncated-sequence). Do not select any optional ports for this example. They will be examined in a later section of the chapter.



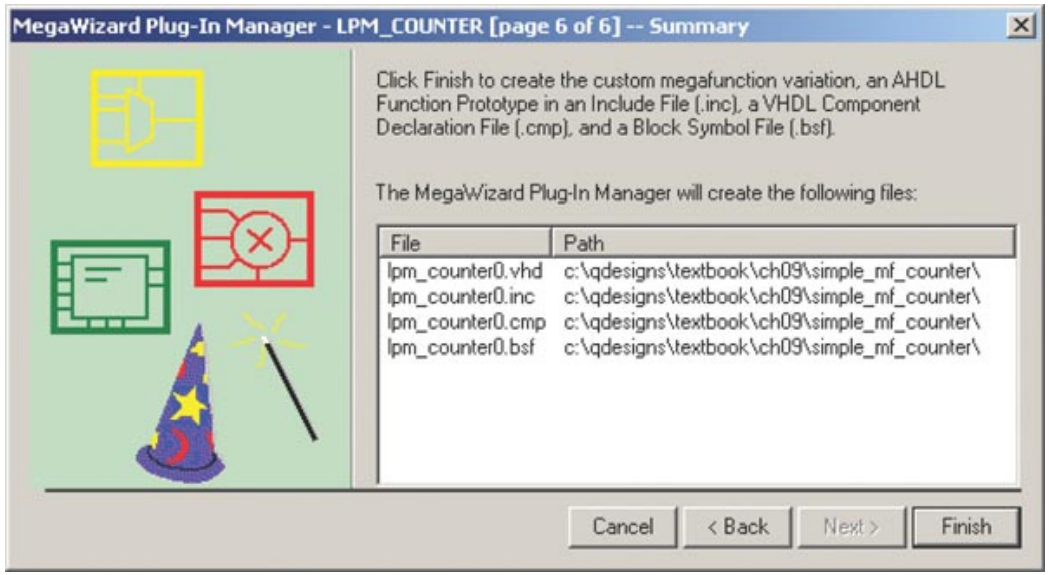
Entering Simple LPM Counters in Quartus II -5

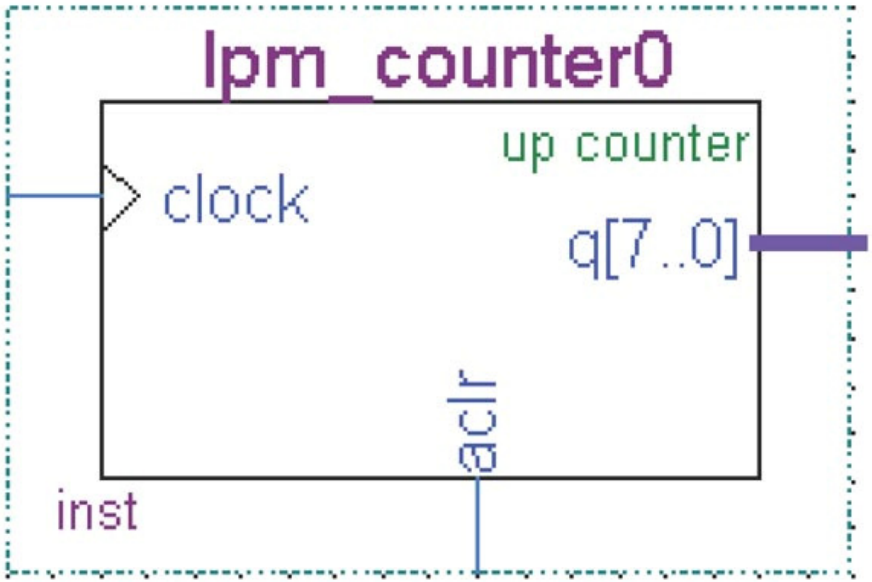
The box shown in Figure 9.25 allows us to choose one or more synchronous or asynchronous control inputs. Select **asynchronous clear.**



Entering Simple LPM Counters in Quartus II -6

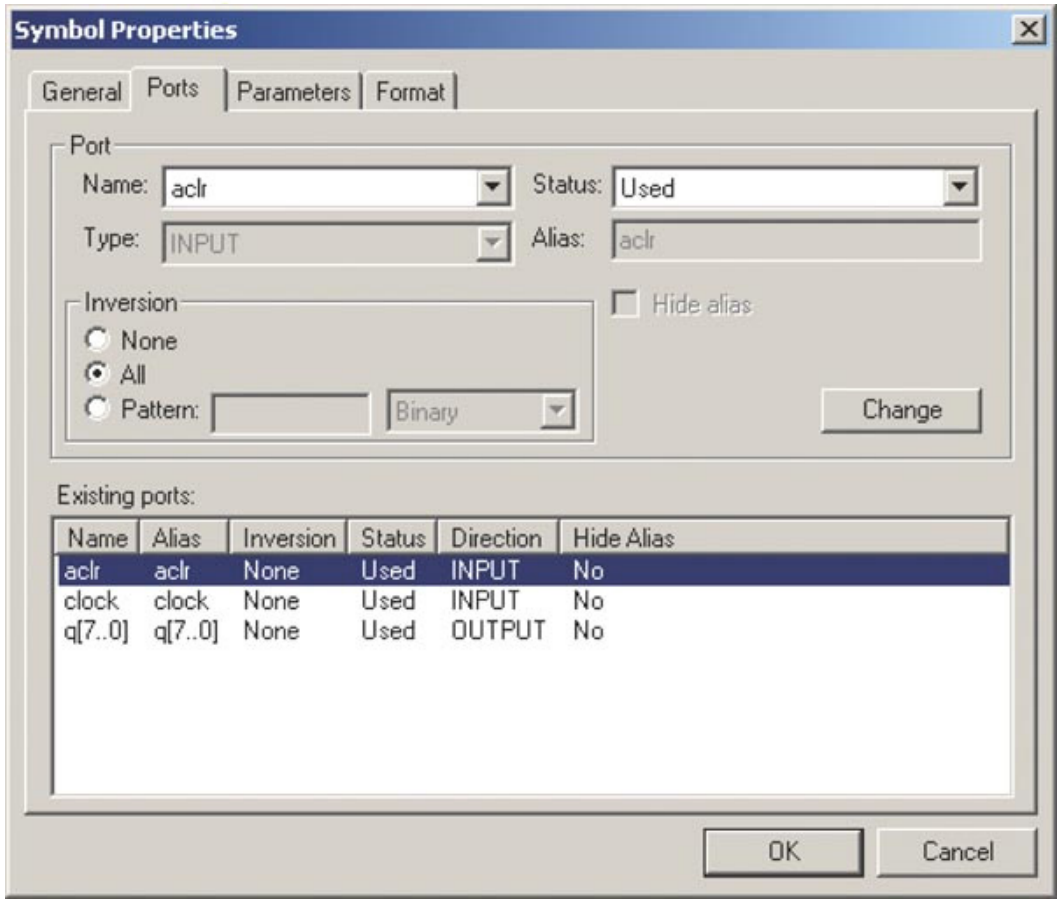
Figure 9.26 shows a list of files generated for the component by the MegaWizard Plug-In Manager. Click **Finish** to insert the component in the Block Diagram File. Figure 9.27 shows the MegaWizard LPM symbol.





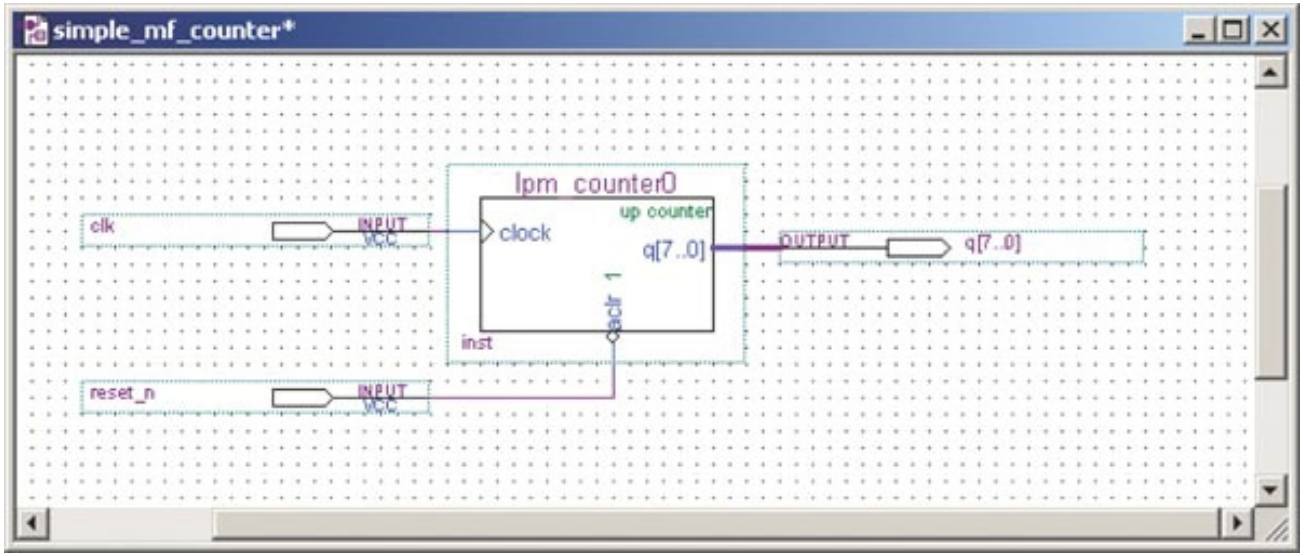
Entering Simple LPM Counters in Quartus II -7

To change the **aclr** (asynchronous clear) input to active-LOW, click on the symbol, then right-click and select **Properties** from the resultant pop-up menu. In the **Ports** tab of the **Symbol Properties** dialog, shown in Figure 9.28, select **aclr,** then click **All** in the box labeled **Inversion.** Click **OK** to accept the choice and close the box.



Entering Simple LPM Counters in Quartus II -8

Figure 9.29 shows the modified MegaWizard LPM symbol in a Quartus II Block Diagram File.

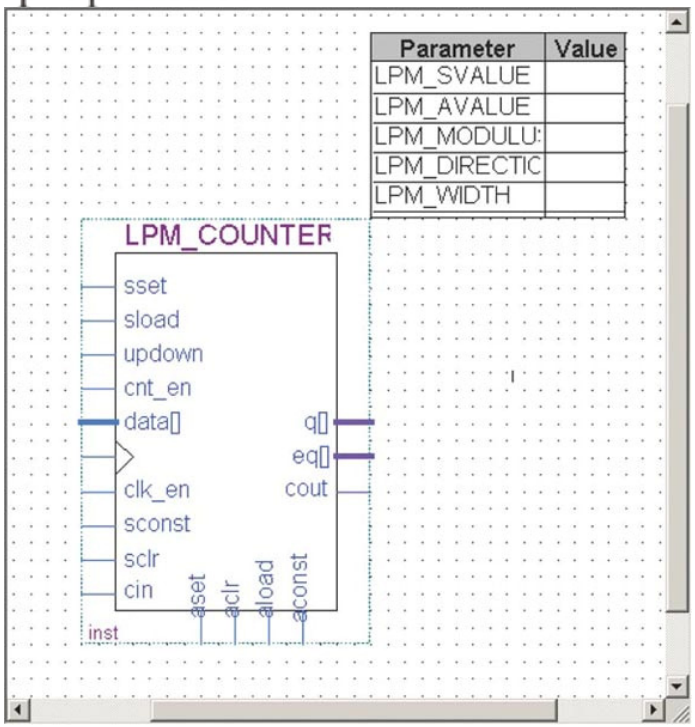


Manually-Entered LPM Component -1

Manually-Entered LPM Component

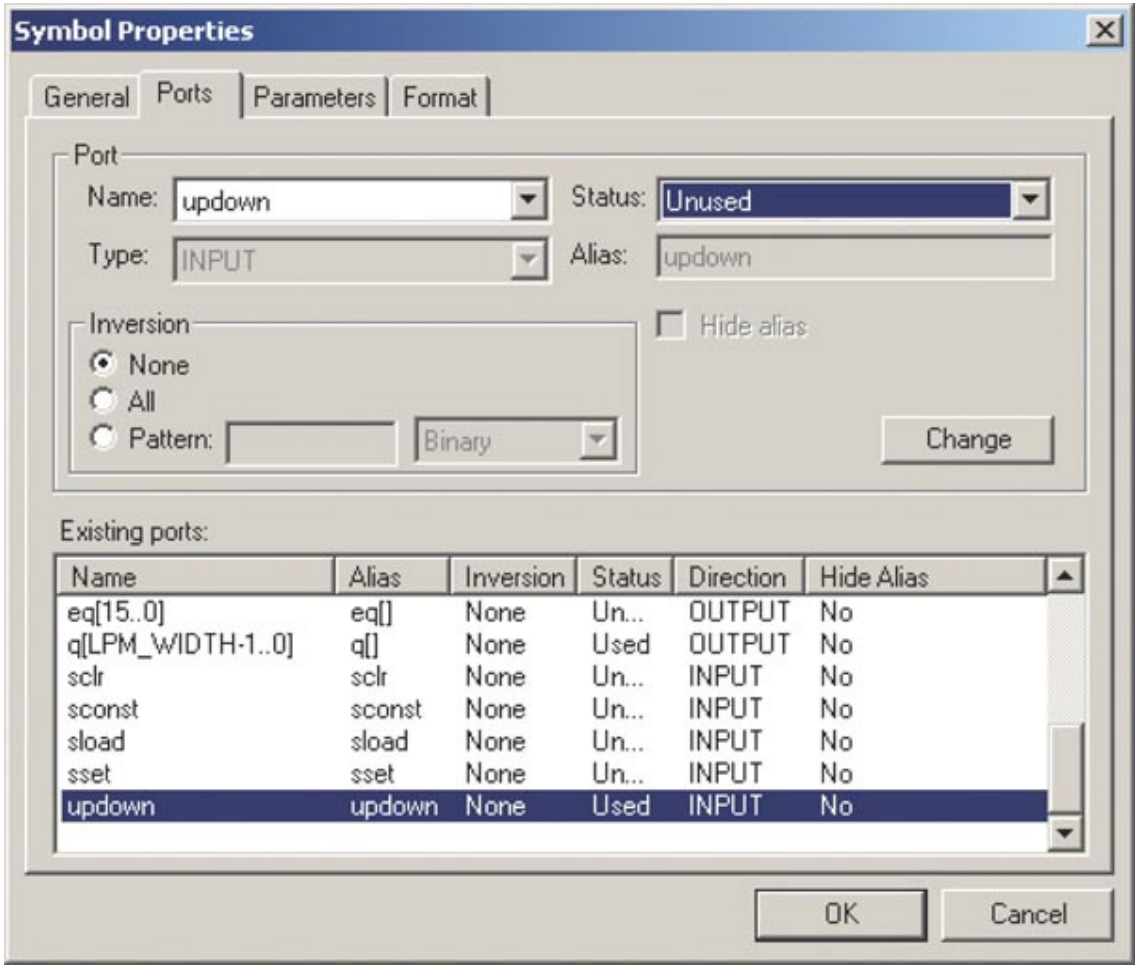
To manually enter an LPM counter in the Quartus II Block Editor, create and save a Block Diagram File and use it to create a new project. Enter the symbol called **lpm_counter,** but make sure the box labeled **Launch MegaWizard Plug-In** is *unchecked.* Click **OK.**

Figure 9.30 shows the default symbol for an LPM counter, with all ports and parameters initially selected. To choose only the required ports and parameters, open the **Symbol Properties** dialog by clicking the component, right-clicking, and selecting **Properties** from the pop-up menu.



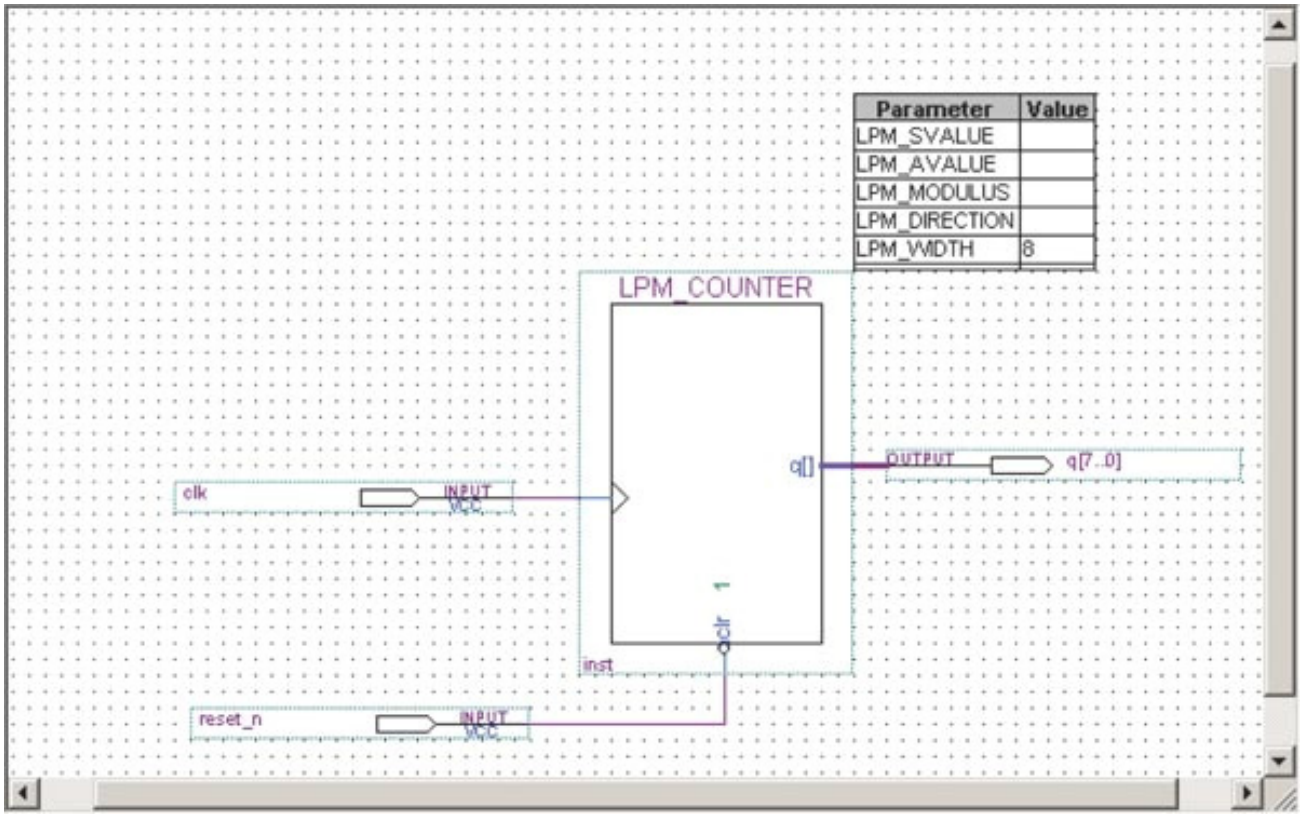
Manually-Entered LPM Component -2

Change the status of all ports to **Unused** except **aclr, clock,** and **q[LPM_WIDTH-1..0],** as shown in Figure 9.31. Select **aclr** and choose **All** in the **Inversion** box. In the **Parameters** tab (not shown), enter the value of **8** for the parameter **LPM_WIDTH.** Click **OK.**



Manually-Entered LPM Component -3

Figure 9.32 shows the modified LPM symbol as a component in a Block Diagram File.



9.5 Control Options For Synchronous Counters

KEY TERMS

Parallel Load A function that allows simultaneous loading of binary values into all flip-flops of a synchronous circuit. Parallel loading can be synchronous or asynchronous.

Clear Reset (synchronous or asynchronous).

Count Enable A control function that allows a counter to progress through its count sequence when active and disables the counter when inactive.

Bidirectional Counter A counter that can count up or down, depending on the state of a control input.

Output Decoding A feature in which one or more outputs activate when a particular counter state is detected.

Ripple Carry Out or Ripple Clock Out (RCO) An output that produces one pulse with the same period as the clock upon terminal count.

Terminal Count The last state in a count sequence before the sequence repeats (e.g., 1111 is the terminal count of a 4-bit binary UP counter; 0000 is the terminal count of a 4-bit binary DOWN counter).

Presettable Counter A counter with a parallel load function.

Parallel Loading -1

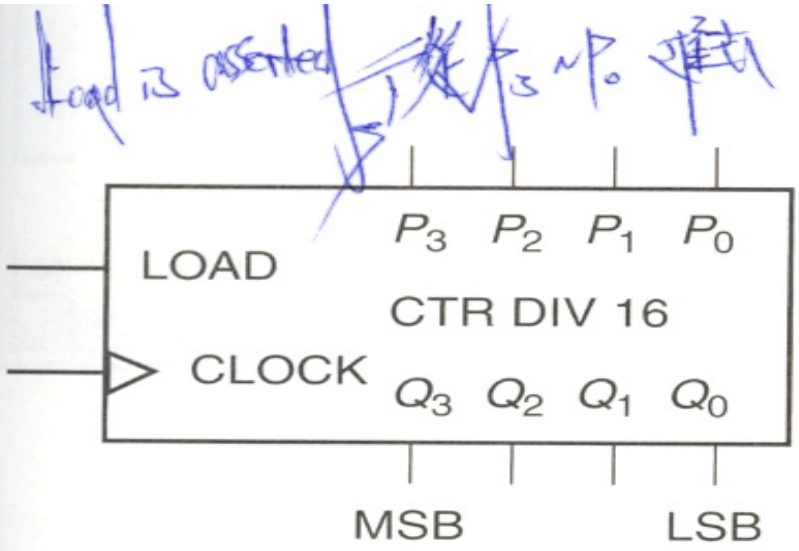
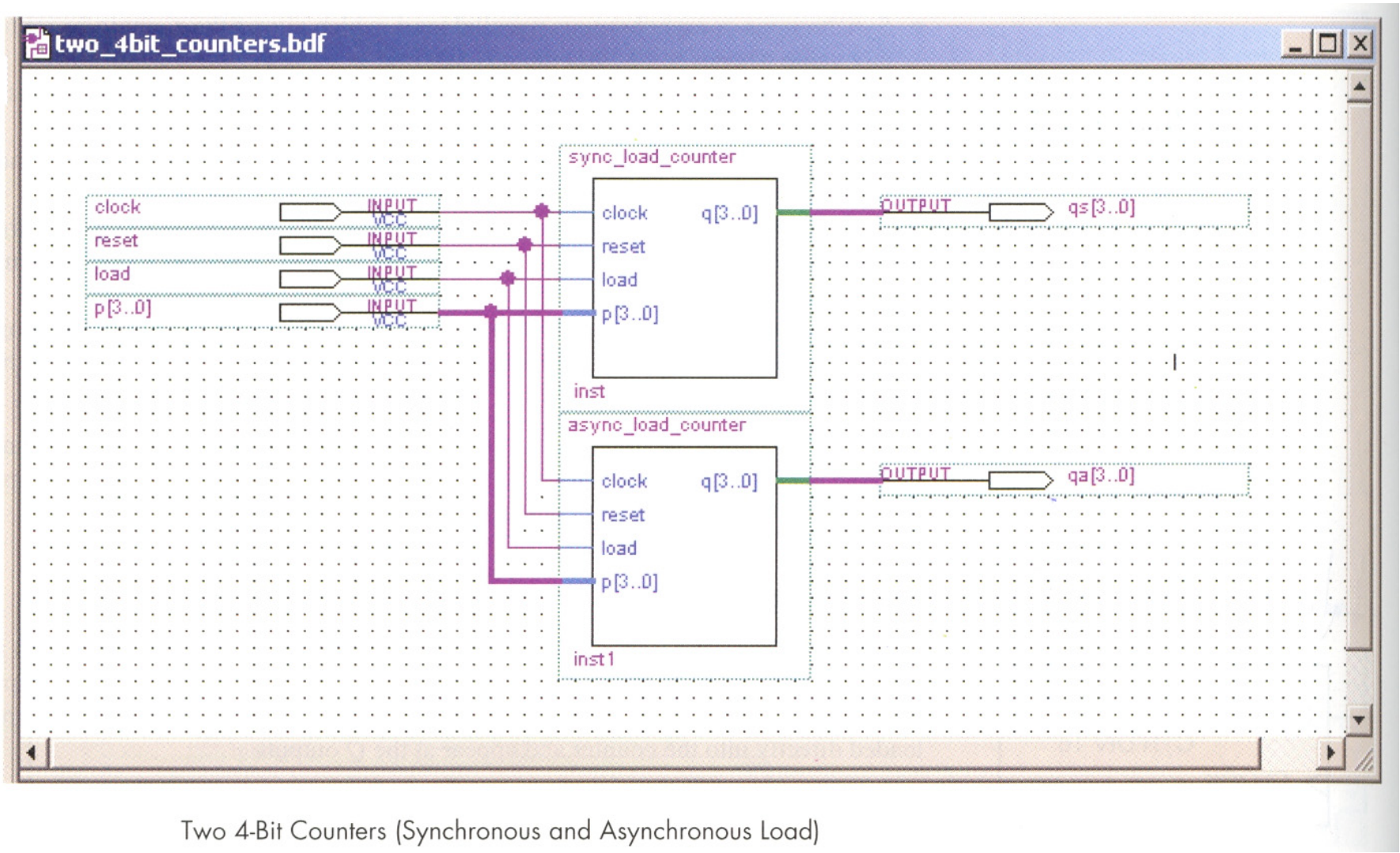


FIGURE 9.33 4-Bit Counter with Parallel Load

a 4-bit **presettable counter** (i.e., a counter with a parallel load function). The parallel inputs, P_3 to P_0, have direct access to the flip-flops of the counter. When the *LOAD* input is asserted, the values at the *P* inputs are loaded directly into the counter and appear at the *Q* outputs.

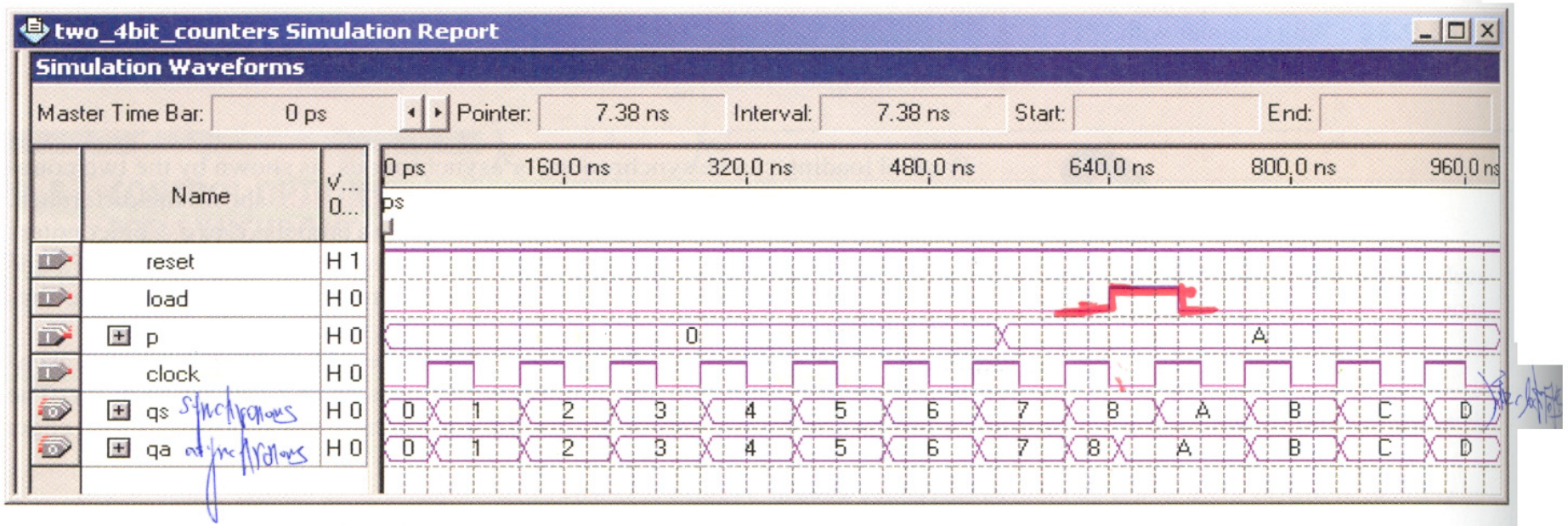
Parallel Loading -2

Parallel loading can be synchronous or asynchronous,



Two 4-Bit Counters (Synchronous and Asynchronous Load)

Parallel Loading -3

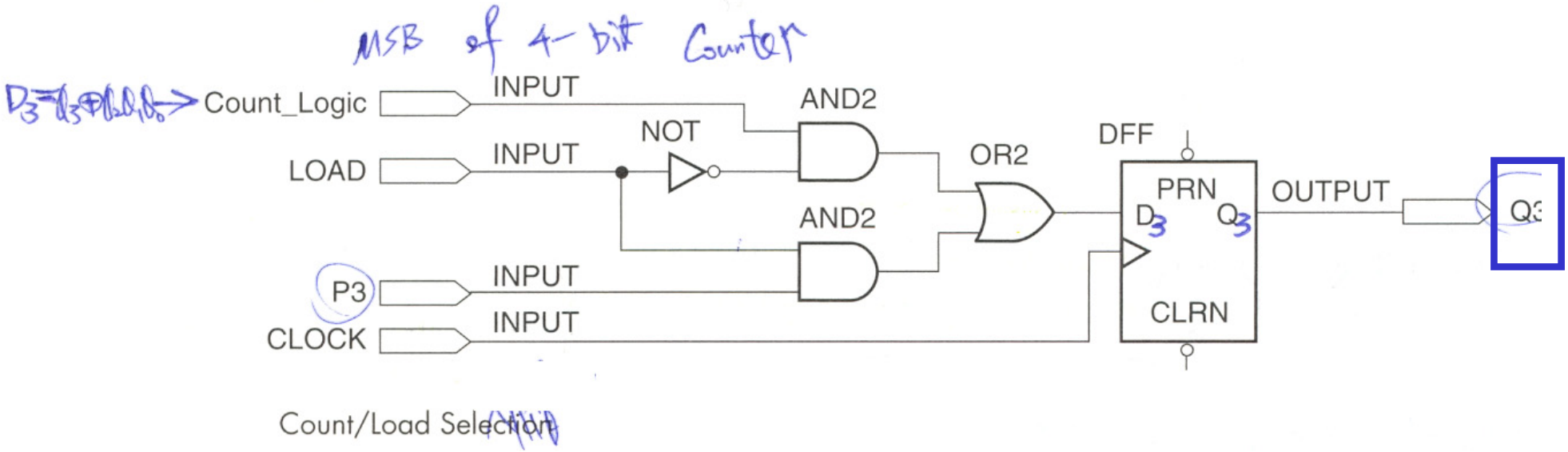


Simulation of Two 4-Bit Counters

When *LOAD* goes HIGH, the value of P[3..0] (= AH) is loaded into the asynchronously loading counter (**qa**) immediately after a short propagation delay. The counter with synchronous load (**qs**) is not loaded until the next positive clock edge.

Synchronous Load -1

Count/Load Selection



Count/Load Selection

The *LOAD* input selects whether the flip-flop synchronous input will be fed by the count logic or by the parallel input P_3. When *LOAD* = 0, the upper AND gate steers the count logic to the flip-flop,

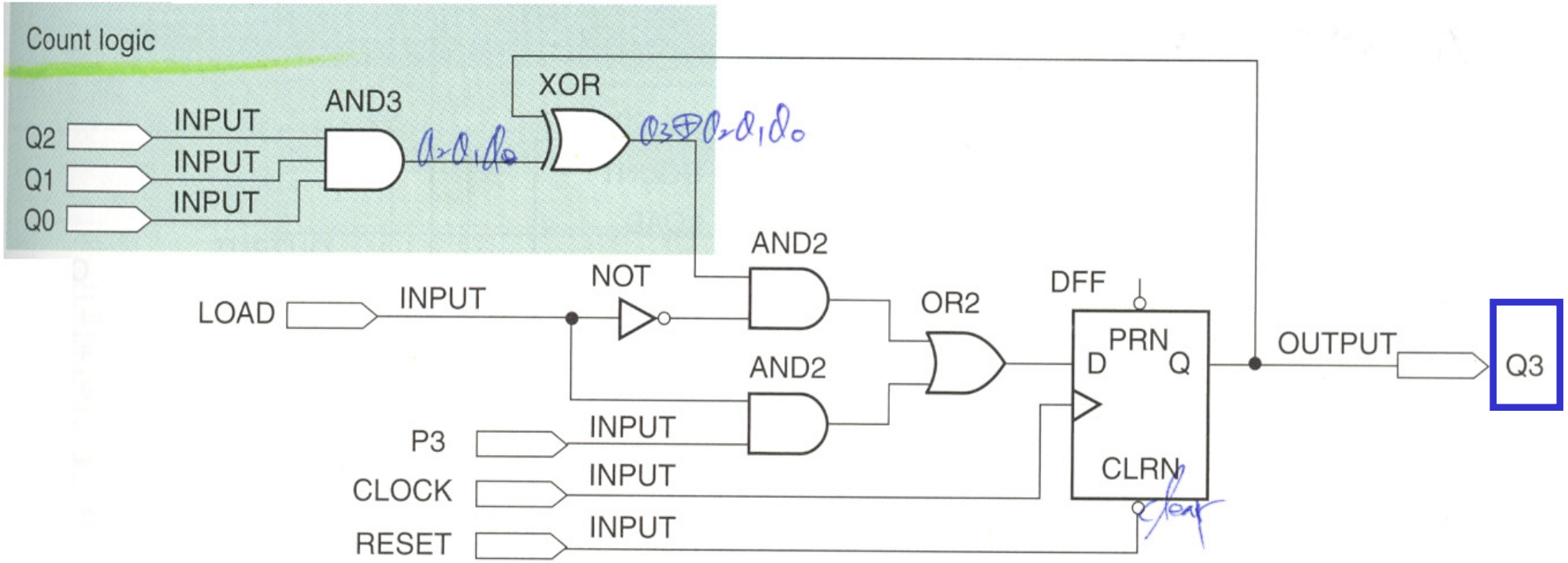
When *LOAD* = 1, the lower AND gate loads the logic level at P_3 directly into the flip-flop on the next clock pulse.

Load=0 做Count動作, $P_3 = Q_3 \oplus Q_2 Q_1 Q_0$ 給 D_3

Load=1, 做load動作, $P_3 \rightarrow D_3$

Synchronous Load -2

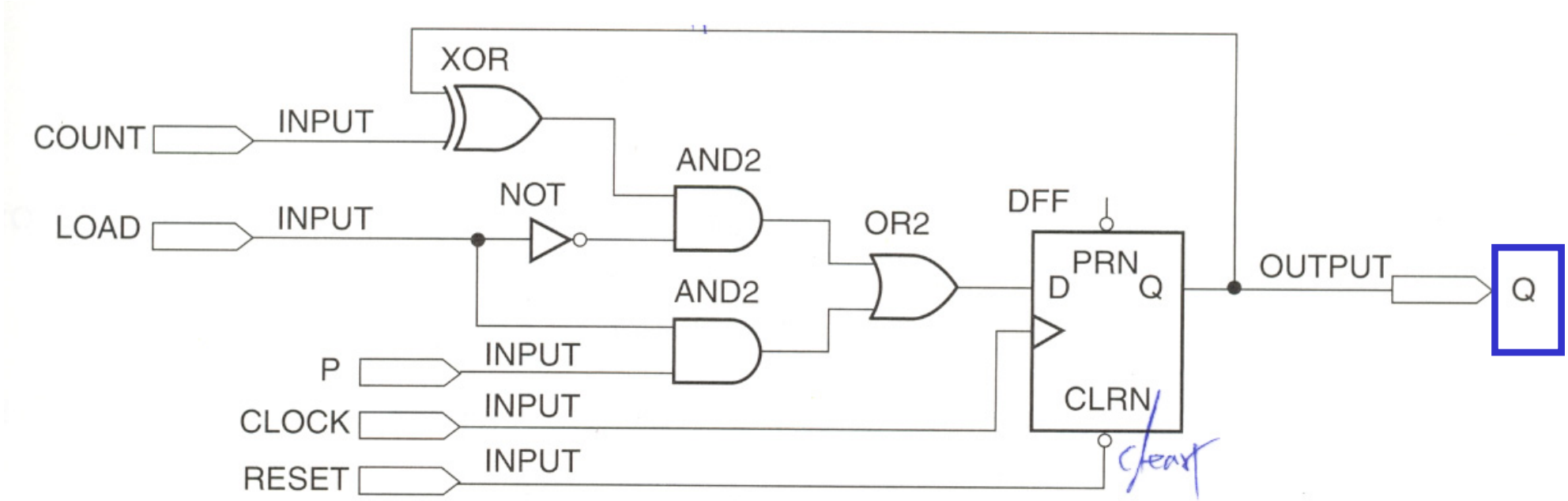
Count/Load Selection Including the Count Logic



Counter Element with Synchronous Load and Asynchronous Clear

shows the same circuit, but includes the count logic.

Synchronous Load -3
General Counter Element



Counter Element with Synchronous Load and Asychronous Reset (sl_count)

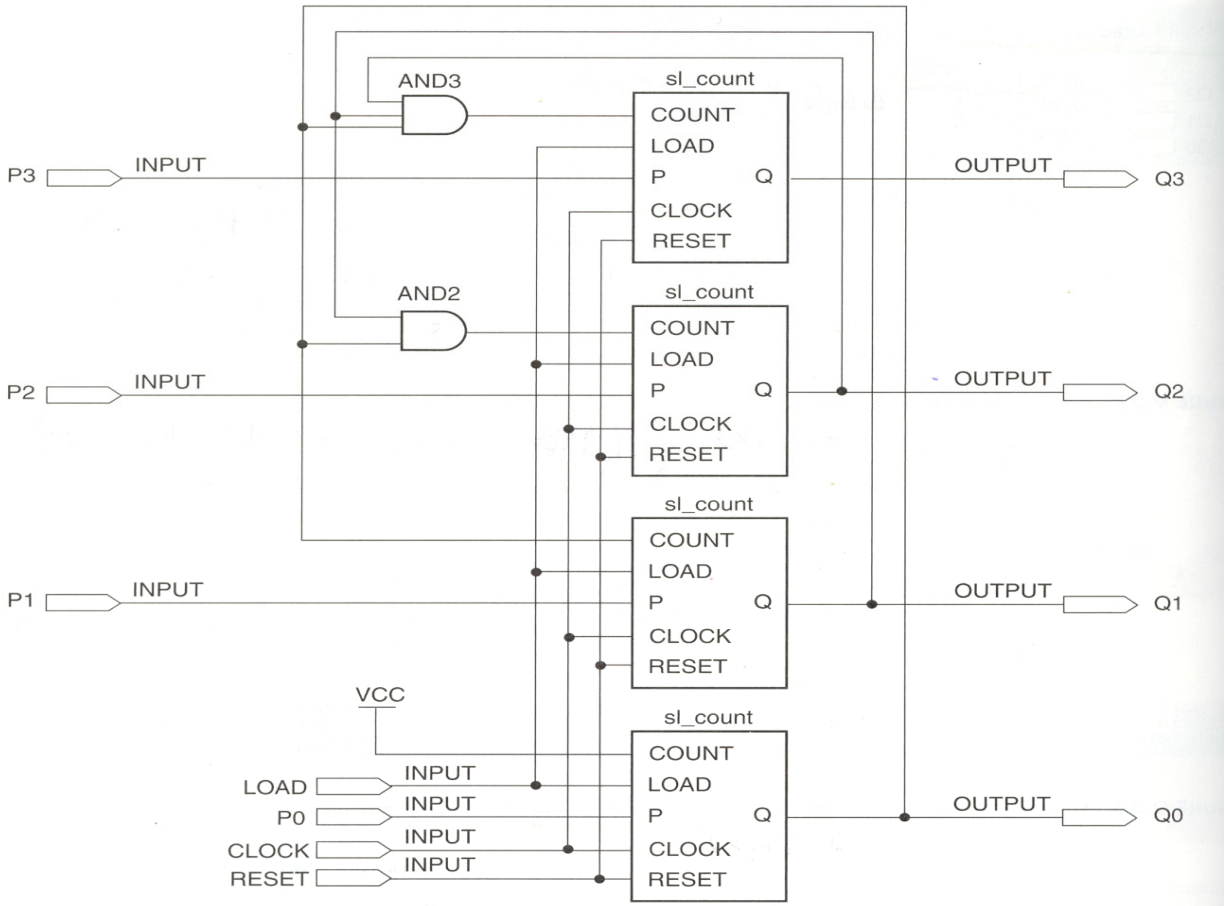
(和clk同步)

Synchronous load

If we leave out the 3-input AND gate, we have a circuit that can be used as a general element (called **sl_count**) in a synchronous presettable counter.

Synchronous Load -4

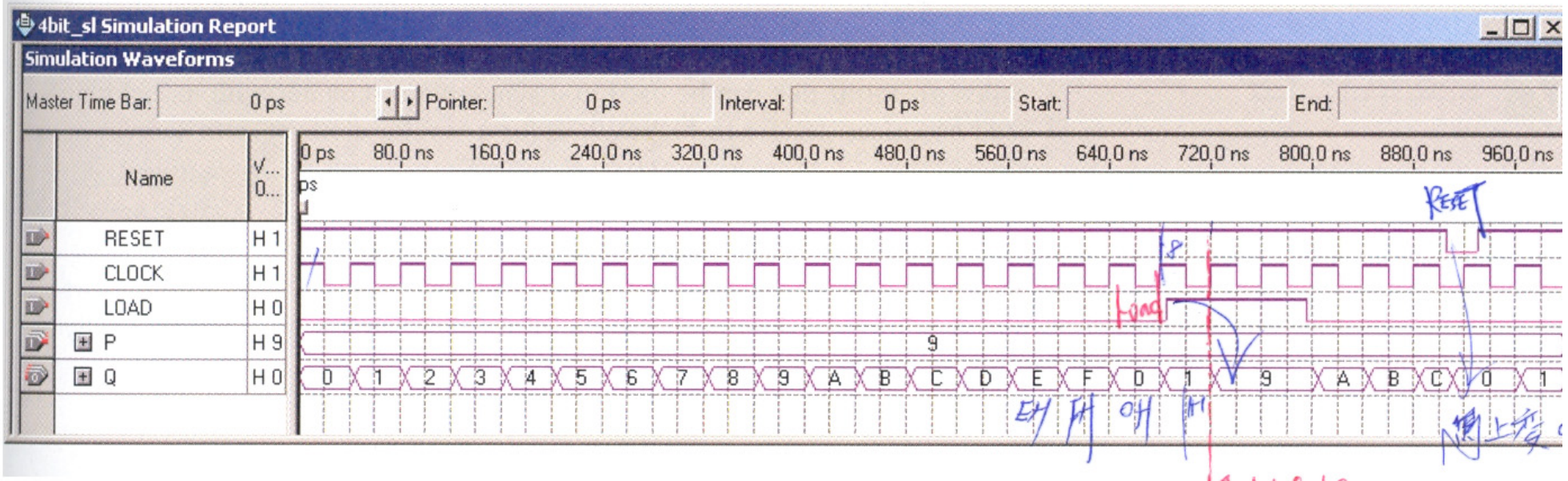
4-Bit Counter



4-Bit Counter with Synchronous Load and Asynchronous Reset

the logic diagram of a 4-bit synchronously presettable counter consisting of four instances of the counter element and appropriate AND gates for a synchronous counter.

Synchronous Load -5
Simulation

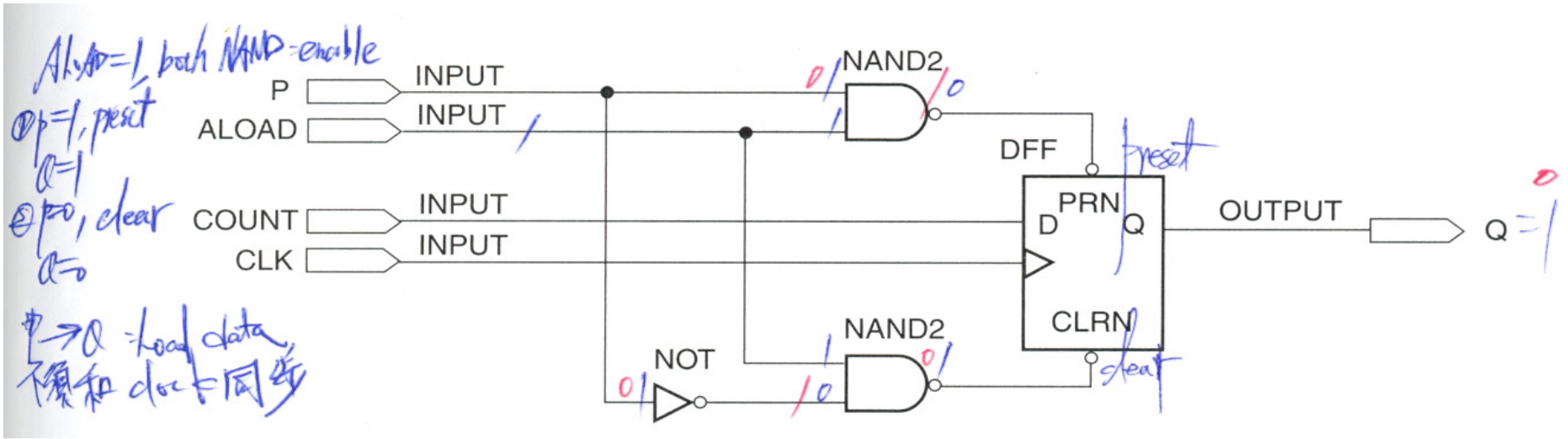


Simulation of a 4-Bit Counter with Synchronous Load and Asynchronous Reset

The first 18 clock pulses drive the counter through its normal 4-bit cycle from 0H to FH, then up to 1H. At this point, we set the *LOAD* input HIGH and the value at the *P* inputs (9H) is loaded into the counter on the rising edge of the next clock pulse. An asynchronous *RESET* pulse then drives the counter outputs to 0H, after which the count resumes.

Asynchronous Load -1

Asynchronous Load Element

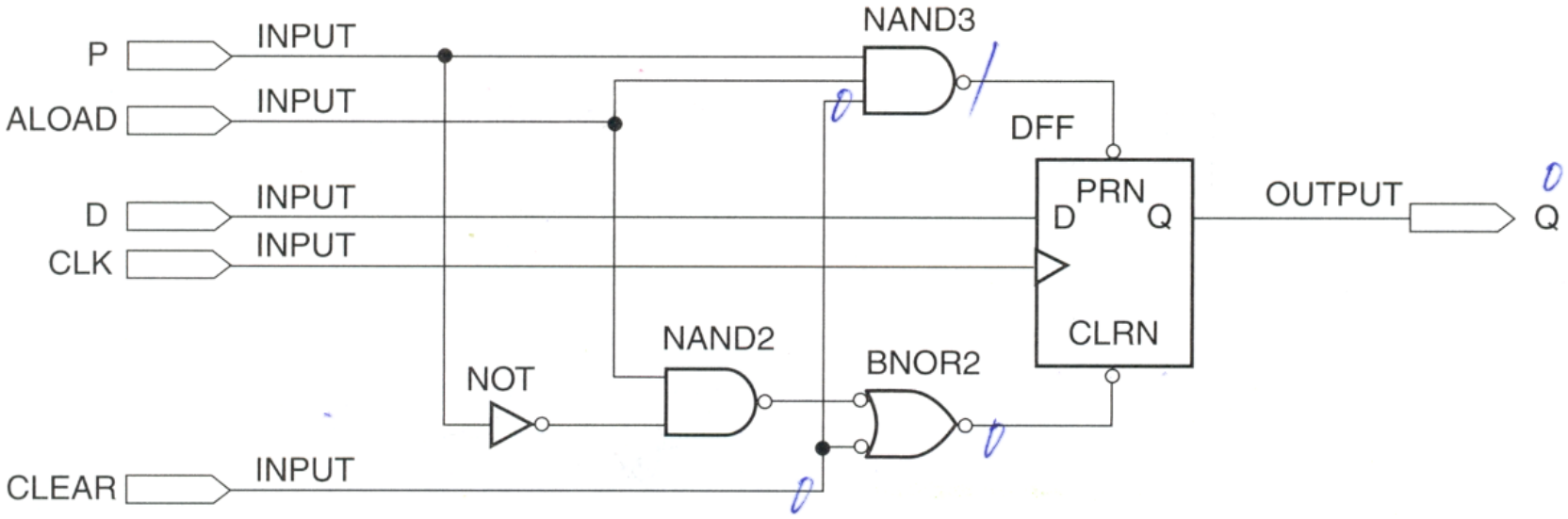


Asynchronous Load Element

The asynchronous load function of a counter makes use of the asynchronous preset and clear inputs of the counter's flip-flops.

Asynchronous Load -2

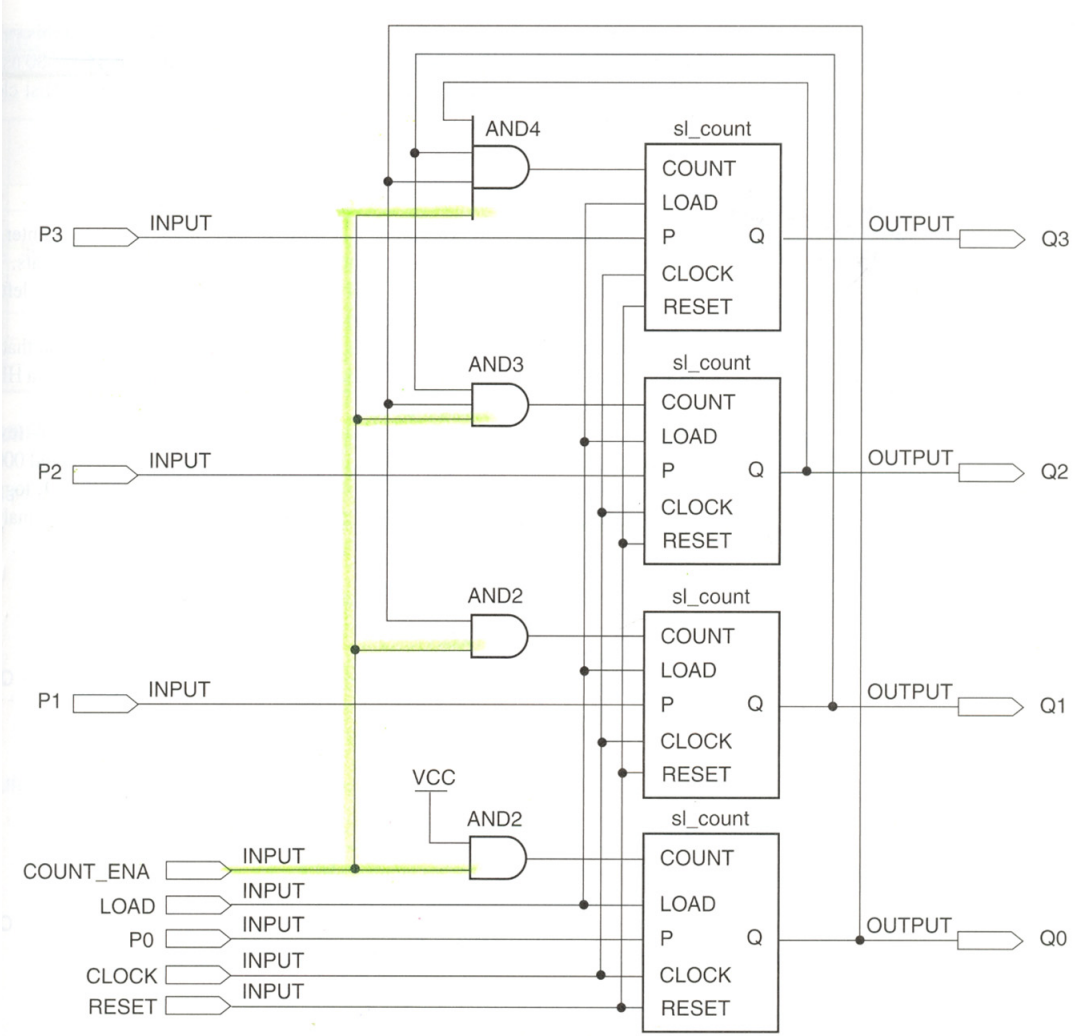
Asynchronous Load Element with Asynchronous Clear



Asynchronous Load Element with Asynchronous Clear

shows the asynchronous load circuit with an asynchronous clear (reset) function added.

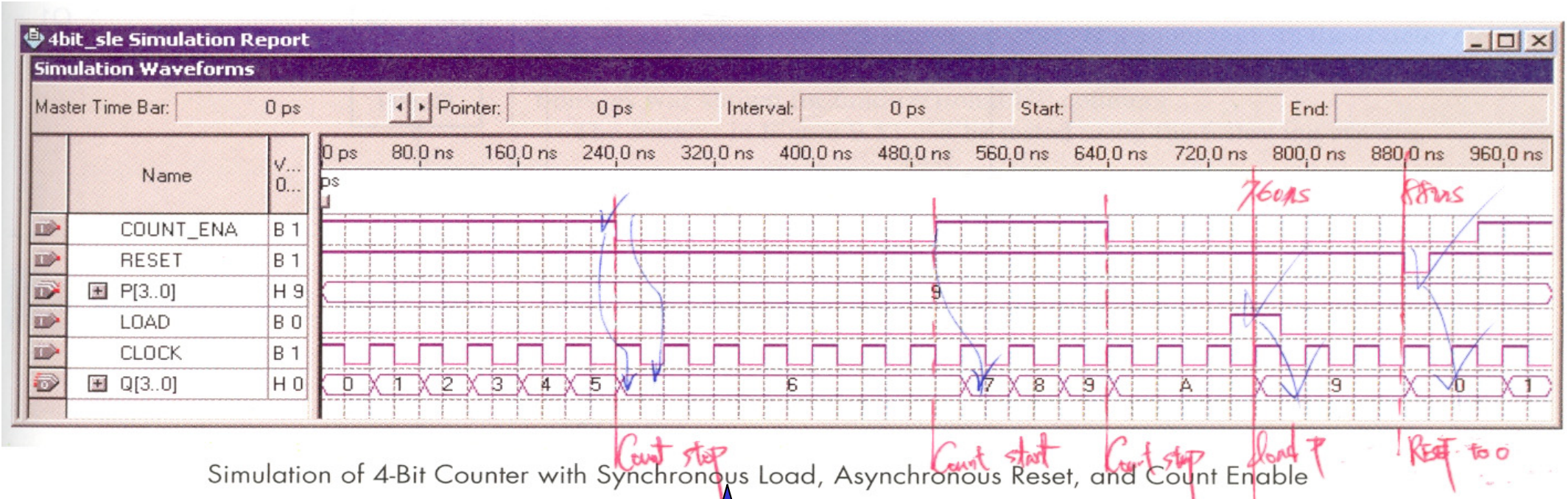
Count Enable -1



4-Bit Counter with Synchronous Load, Asynchronous Reset, and Count Enable

Each AND gate has an extra inpu
which is used to enable or inhibit the count logic function to each flip-flop.

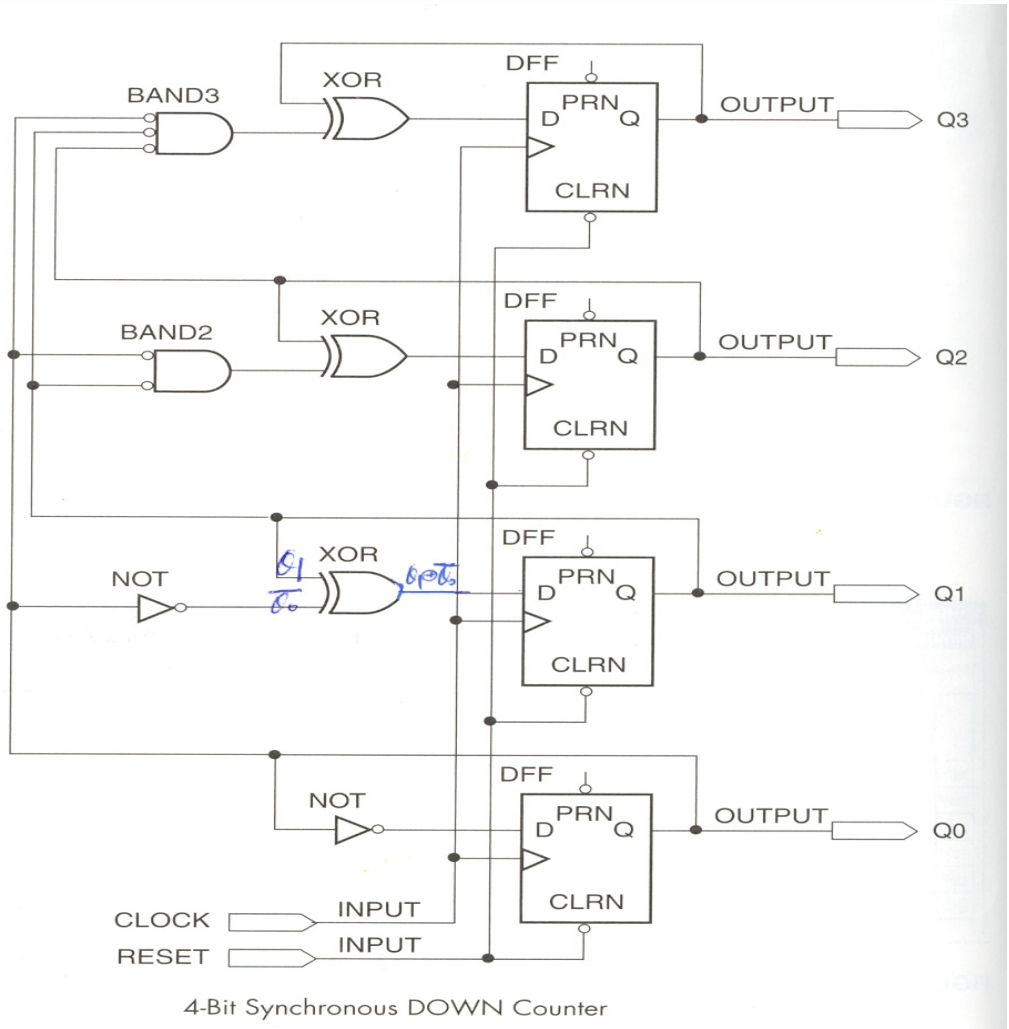
Count Enable -2



Simulation of 4-Bit Counter with Synchronous Load, Asynchronous Reset, and Count Enable

note that the count enable has no effect on the synchronous load and asynchronous reset functions. In the latter part of the simulation, the count stops at AH ($Q_3Q_2Q_1Q_0 = 1010_2$), when *COUNT_ENA* goes LOW. At 760 ns, the synchronous load function loads the value of 9H into the counter. The counter stays at this value, even after LOAD is no longer active, since the count is still disabled. At 880 ns, an asynchronous reset pulse clears the counter.

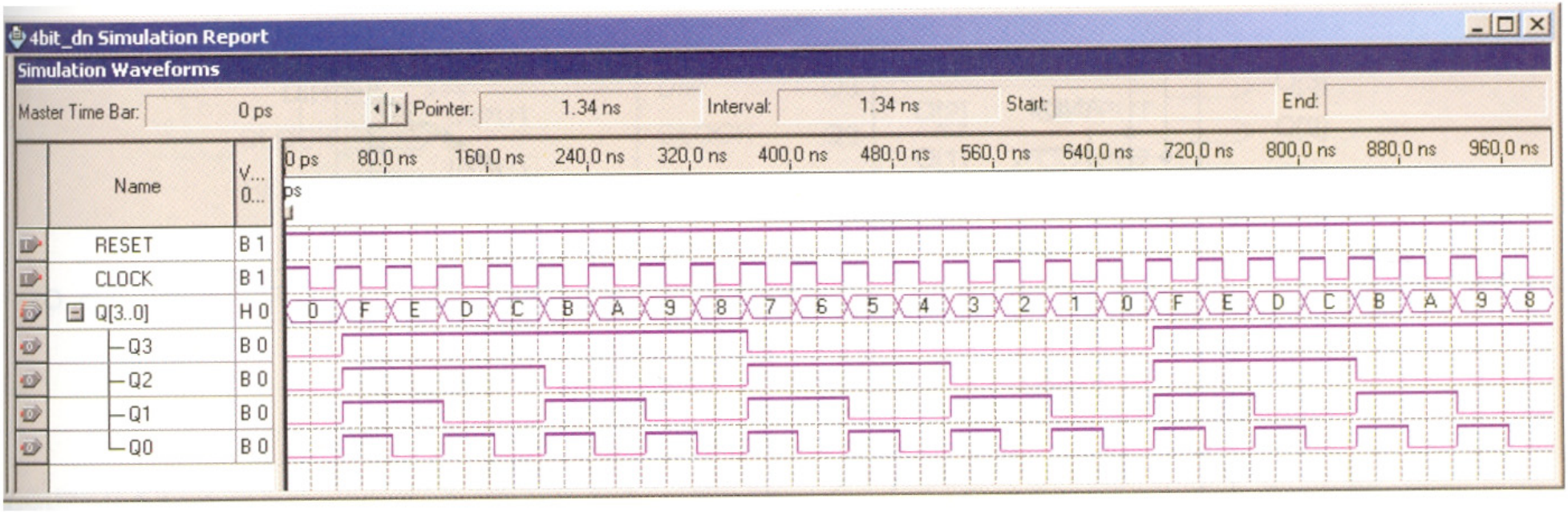
Down Counter -1



4-Bit Synchronous DOWN Counter

the logic diagram of a 4-bit synchronous DOWN counter. Its count sequence starts at 1111 and counts backwards to 0000, then repeats.

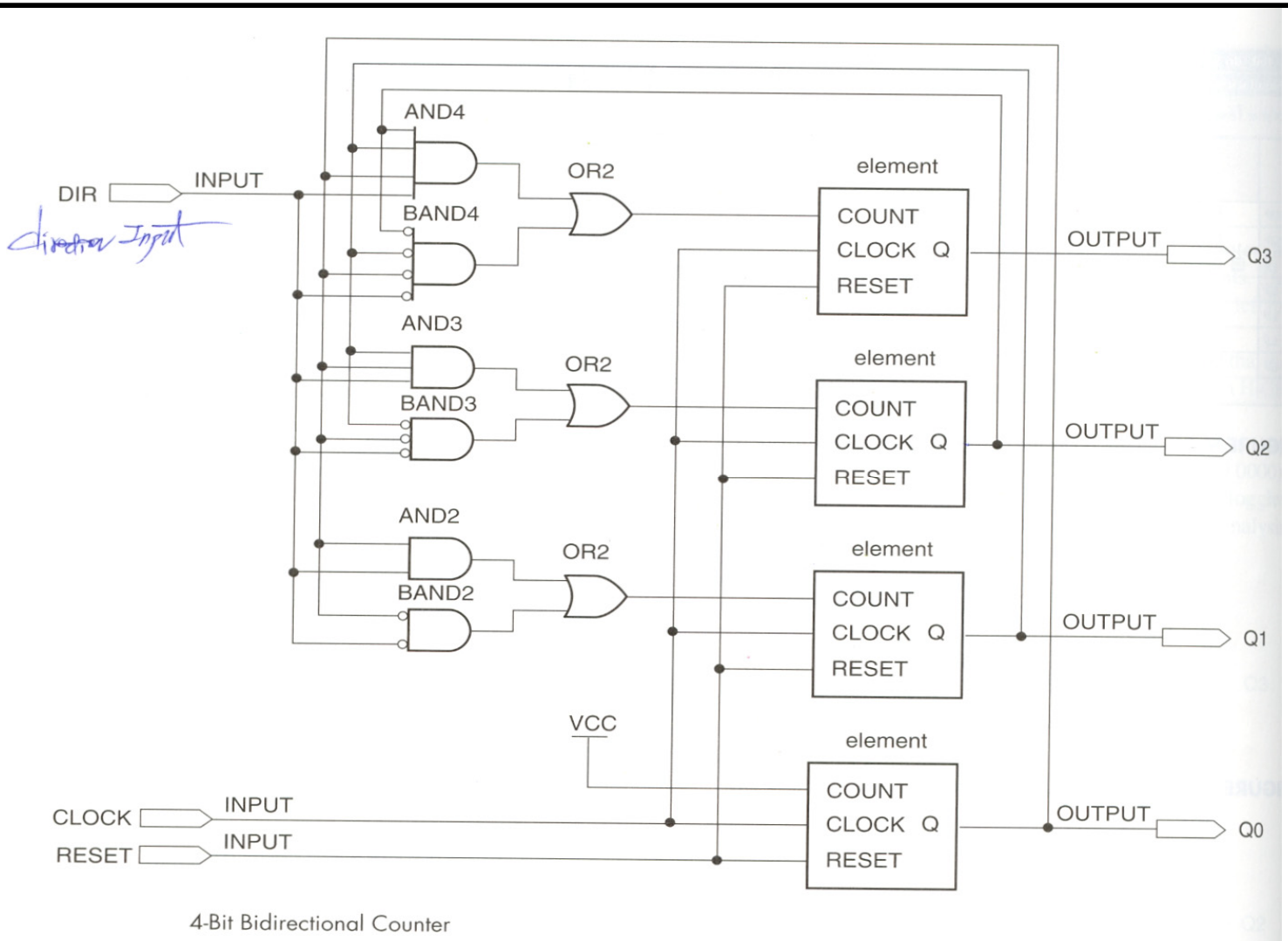
Down Counter -2



Synchronous DOWN Counter Simulation

the counter will count down from 1111 (FH) to 0000 (0H) and repeat.

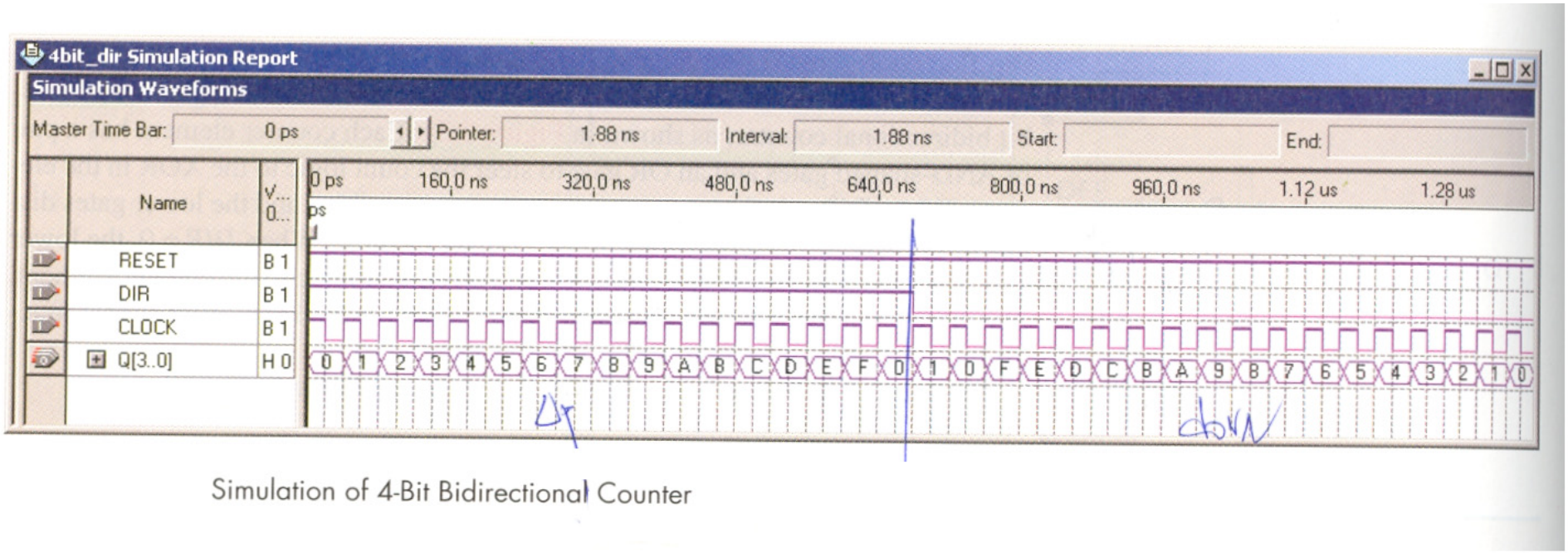
Bidirectional Counter -1



4-Bit Bidirectional Counter

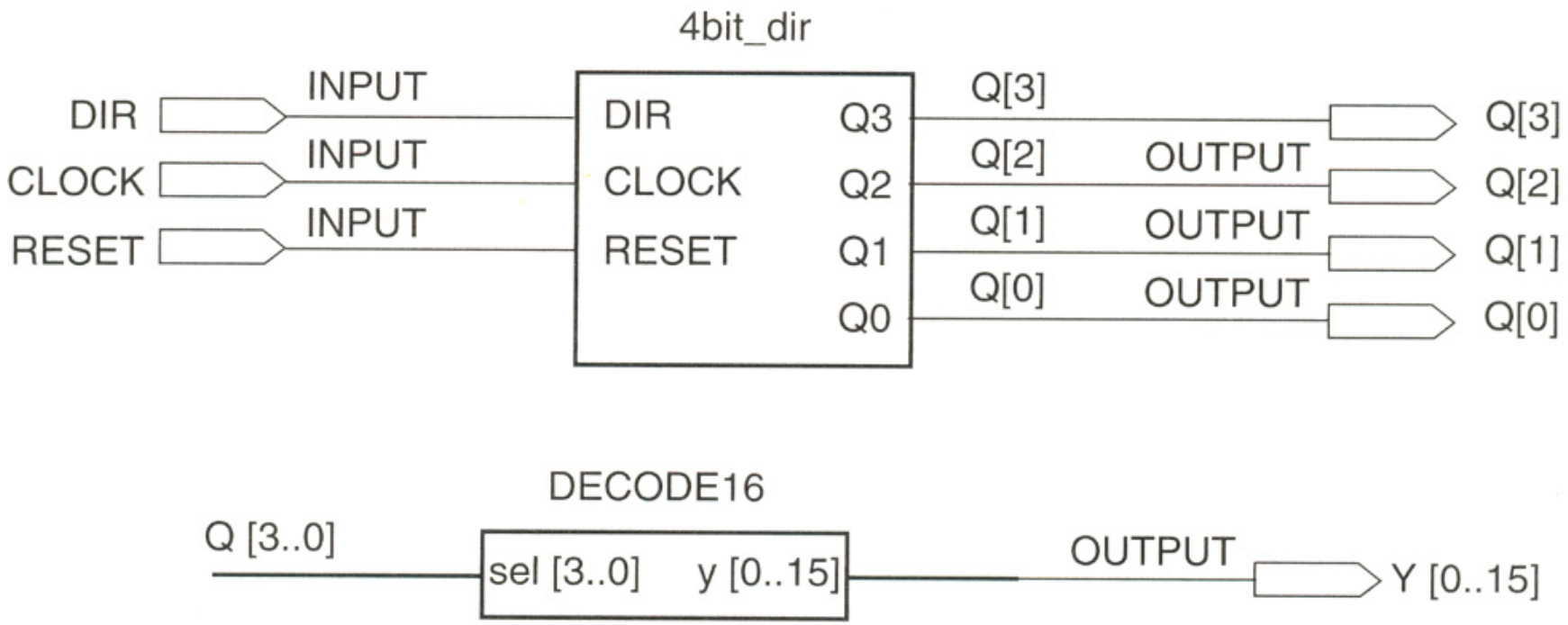
Observe the effect of the *DIR* input. When *DIR* = 1, the count should progress from 0H to FH. When *COUNT_ENA* = 0, the count should go from FH to 0H.

Bidirectional Counter -2



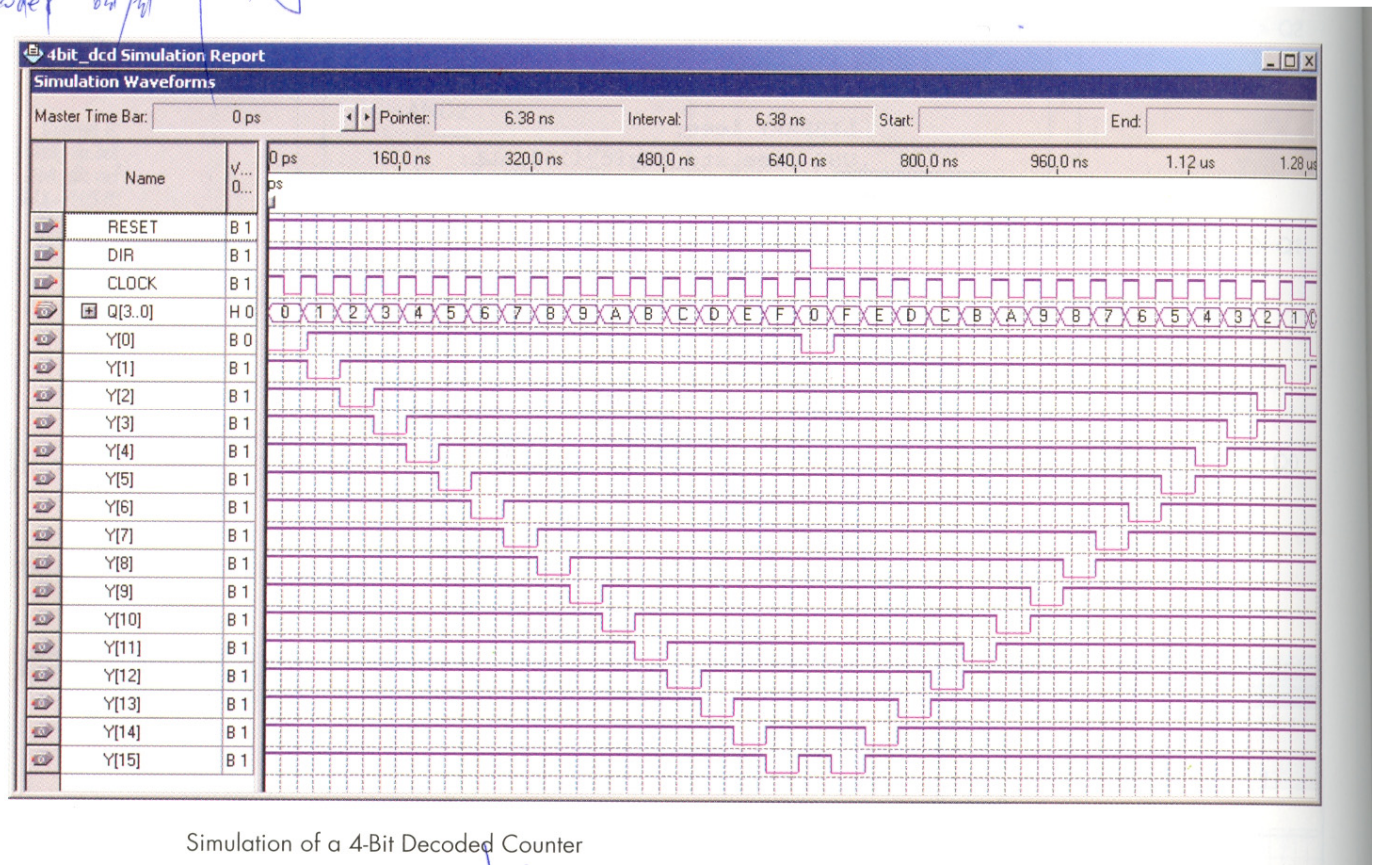
Simulation of 4-Bit Bidirectional Counter

Decoding the Output of a Counter -1



a Block Diagram File of a 4-bit bidirectional counter with an output decoder.

Decoding the Output of a Counter -2



Simulation of a 4-Bit Decoded Counter

- The counter should be clocked at least 32 times to observe a full count cycle from 0H to FH and from FH to 0H.
- Test the directional function. Count should be up when *DIR* = 1 and down when *DIR* = 0.
- The decoder outputs should activate in the same sequence as the count value.

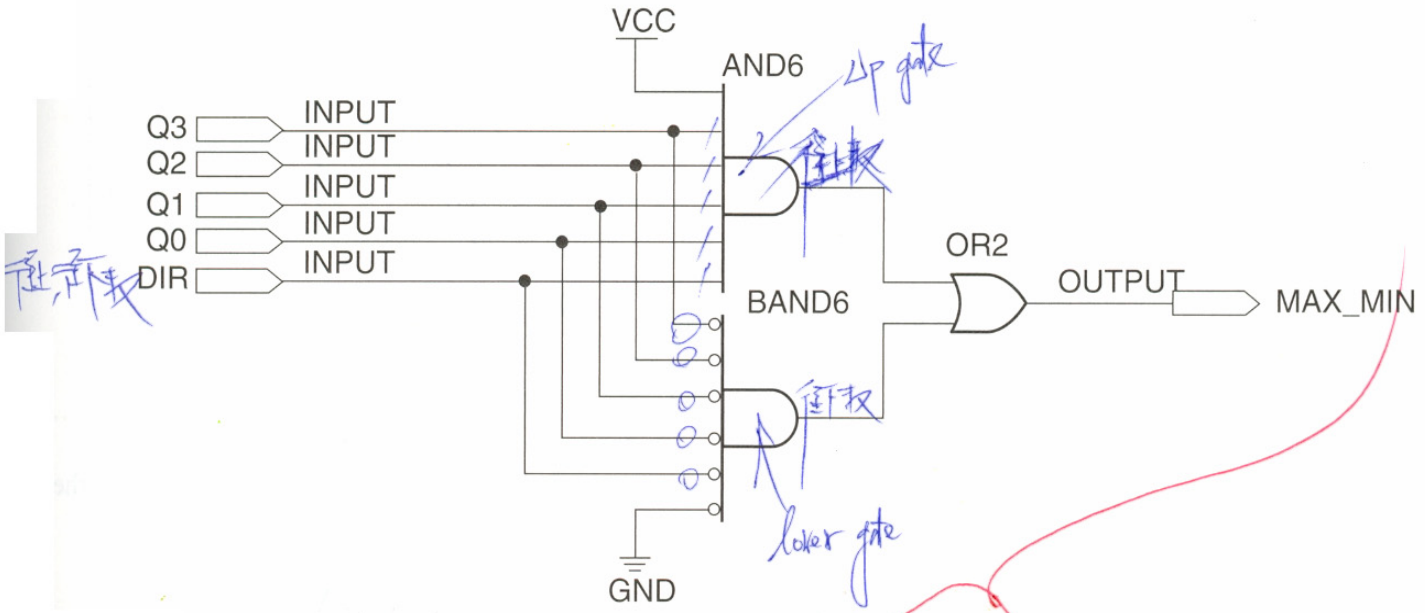
Terminal Count -1

- Uses a combinational decoder to detect when the last state of a counter is reached (terminal count).

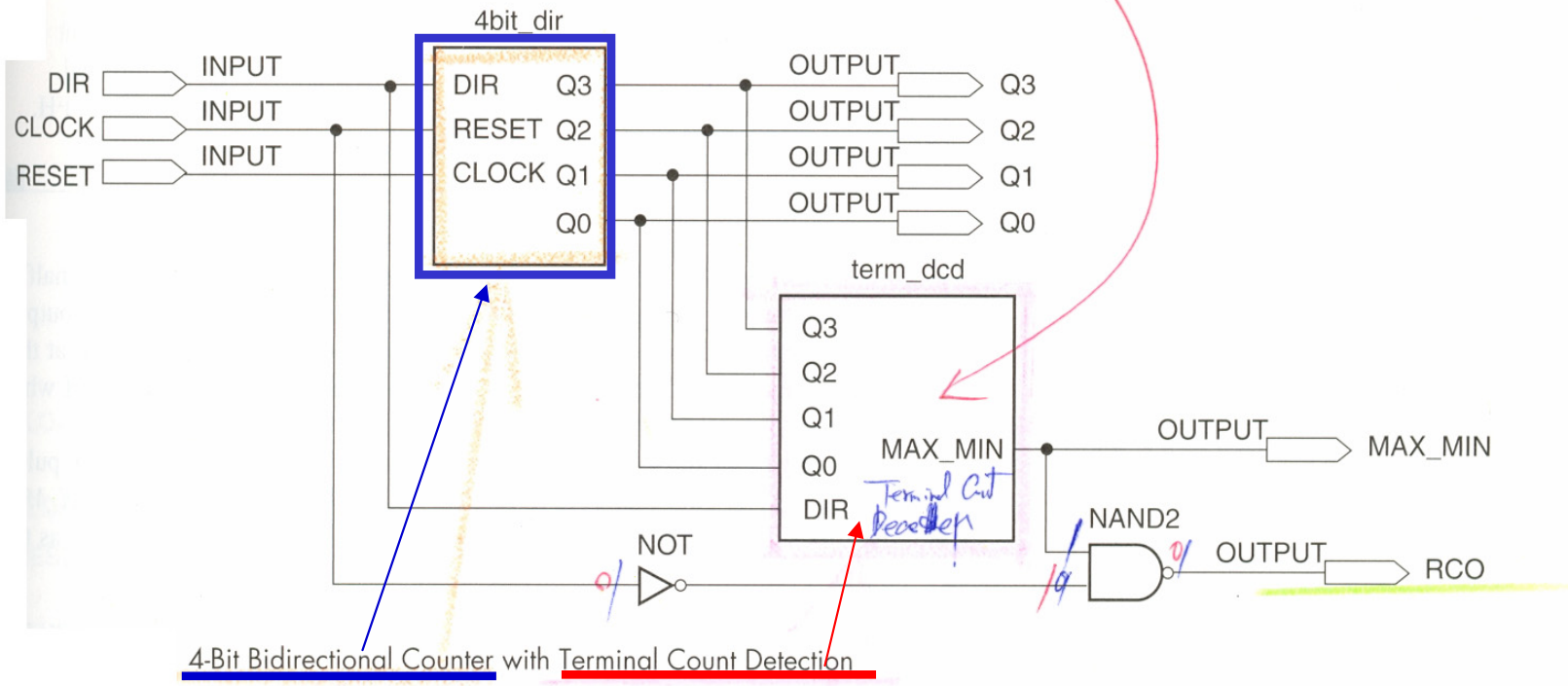
- Determines a maximum count out for an UP counter and a minimum for a DOWN counter.

A 4-bit binary UP counter has a terminal count of 1111; a 4-bit binary DOWN counter has a terminal count of 0000. A circuit to detect these conditions must detect the *maximum value* of an UP count and the *minimum value* of a DOWN count.

Terminal Count -2



Terminal Count Decoder for a 4-Bit Bidirectional Counter

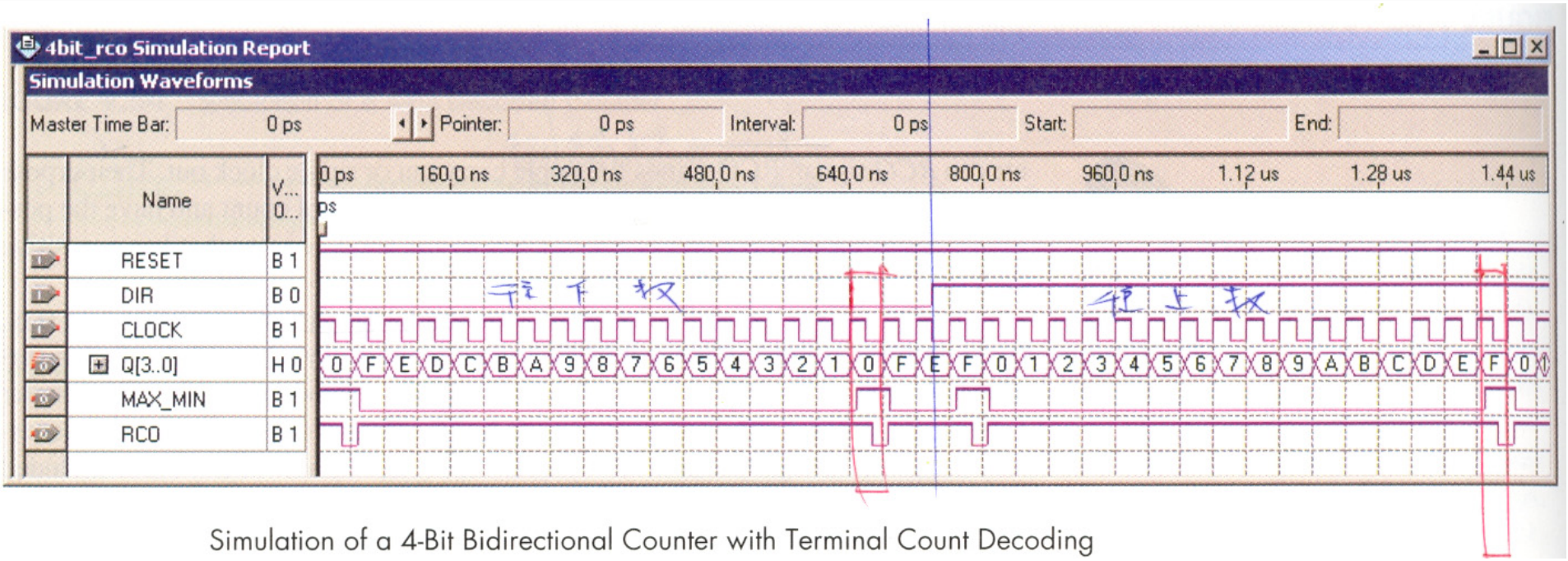


4-Bit Bidirectional Counter with Terminal Count Detection

Ripple Carry Out

- The terminal count decoder generates a RCO (ripple carry out) when the terminal count is reached (a high pulse for 1 clock period).

Simulation



Simulation of a 4-Bit Bidirectional Counter with Terminal Count Decoding

- The counter should be clocked for at least 32 clock cycles to observe a full up-count and a full down-count.
- When *DIR* = 0, the counter should count down from FH to 0H and recycle. When *DIR* = 1, the counter should count up from 0H to FH and recycle.
- When the count is DOWN, *MAX_MIN* should go HIGH when the count is 0H. *RCO* should reproduce one full cycle of the clock during this interval.
- When the count is UP, *MAX_MIN* should go HIGH when the count is FH. *RCO* should reproduce one full cycle of the clock during this interval.

9.6 Programming Presentable and Bidirectional Counters for CPLDs

The following lists the VHDL code for an 8-bit bidirectional counter with count enable, terminal count decoding, and asynchronous load and clear:

```
-- presettable_8bit_counter_async_load.vhd
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY presettable_8bit_counter_async_load IS
   PORT(
      clk, count_ena          : IN     STD_LOGIC;
      clear, load, direction  : IN     STD_LOGIC;
      p                       : IN     STD_LOGIC_VECTOR(7 downto 0);
      max_min                 : OUT    STD_LOGIC;
      q                       : BUFFER STD_LOGIC_VECTOR(7 downto 0));
END presettable_8bit_counter_async_load;
```

8-Bit Counter with Asynchronous Load -2

```
ARCHITECTURE a OF presettable_8bit_counter_async_load IS
   SIGNAL terminal_count : STD_LOGIC_VECTOR(8 downto 0);
BEGIN
   PROCESS (clk, clear, load, p)
   BEGIN
      IF (clear = '0') THEN                          -- Asynchronous clear
         q <= (others => '0');
      ELSIF (load = '1' and clear = '1') THEN -- Asynchronous load
         q <= p;
      ELSE
         IF (clk'EVENT AND clk = '1') THEN
            IF (count_ena = '1' and direction = '0') THEN
               -- count down, if enabled
               q <= q - 1;
            ELSIF (count_ena = '1' and direction = '1') THEN
               -- count up, if enabled
               q <= q + 1;
            END IF;
         END IF;
      END IF;
   END PROCESS;

   -- Terminal count decoder (combinational)
   terminal_count <= direction & q;

WITH terminal_count SELECT
   max_min <= '1' WHEN "000000000",
              '1' WHEN "111111111",
              '0' WHEN others;
END a;
```

Sensitivity list contains clk and all asynchronous controls

Asynch. functions checked before clk

Synchronous functions checked as a condition of clk

8-Bit Counter with Asynchronous Load -3

After the PROCESS statement executes, **q** has several possible values. They are:

- **q** = 0 (if **clear** = 0),
- **q** = **p** (if **clear** = 1 and **load** = 1),
- **q** increments (if there is a positive clock edge, **count_ena** = 1, and **direction** = 1),
- **q** decrements (if there is a positive clock edge, **count_ena** = 1, and **direction** = 0), or
- **q** stays the same (if none of the conditions are met).

8-Bit Counter with Synchronous Load -1

```
-- presettable_8bit_counter_sync_load
-- 8-bit presettable counter with synchronous clear and load
-- and terminal count decoding using STD_LOGIC types

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY presettable_8bit_counter_sync_load IS
   PORT(
        clk, count_ena           : IN      STD_LOGIC;
        clear, load, direction   : IN      STD_LOGIC;
        p                        : IN      STD_LOGIC_VECTOR(7 downto 0);
        max_min                  : OUT     STD_LOGIC;
        q                        : BUFFER  STD_LOGIC_VECTOR(7 downto 0));
END presettable_8bit_counter_sync_load;
```

8-Bit Counter with Synchronous Load -2

```
ARCHITECTURE a OF presettable_8bit_counter_sync_load IS
   SIGNAL terminal_count : STD_LOGIC_VECTOR(8 downto 0);
BEGIN
   PROCESS (clk) ——— Since all functions are synchronous, only clk is in sensitivity list
   BEGIN
      IF (clk'EVENT AND clk = '1') THEN
      IF (clear = '0') THEN -- Synchronous clear
         q <= (others => '0');
      ELSIF (load = '1') THEN -- Synchronous load
         q <= p;
      ELSIF (count_ena = '1' and direction = '0') THEN
         q <= q - 1;
          ELSIF (count_ena = '1' and direction = '1') THEN
             q <= q + 1;
          END IF;
       END IF;
   END PROCESS;

   -- Terminal count decoder (combinational)
   terminal_count <= direction & q;

   WITH terminal_count SELECT
      max_min <= '1' WHEN "000000000",
                 '1' WHEN "111111111",
                 '0' WHEN others;
   END a;
```

8-Bit Counter with Synchronous Load -3

The PROCESS statement in the synchronous counter has only one identifier in its sensitivity list—that of the clock input. Load and clear status are not evaluated until after the process checks for a positive clock edge.

Simulation Result -1

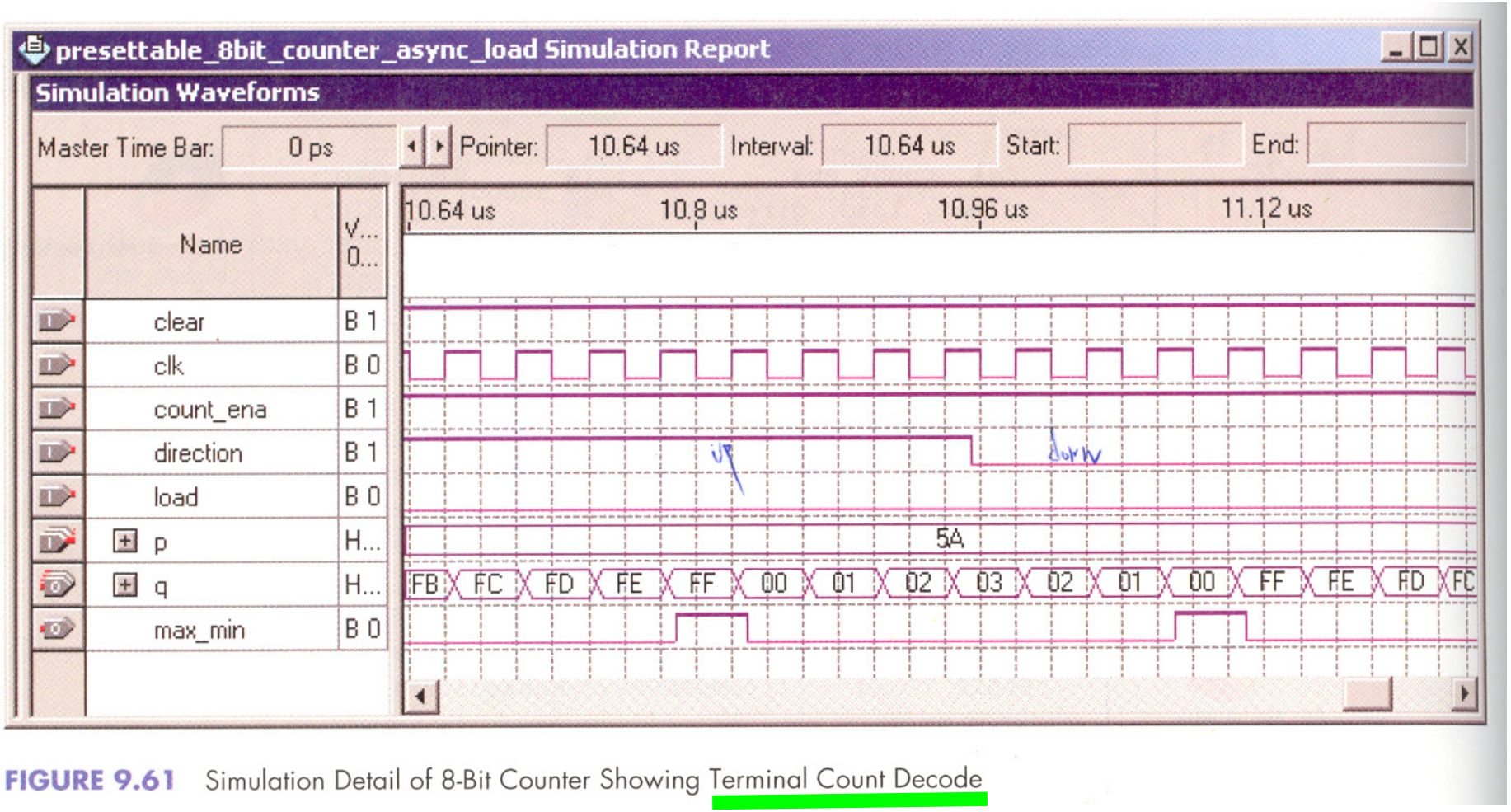


FIGURE 9.61 Simulation Detail of 8-Bit Counter Showing Terminal Count Decode

Simulation Result -2

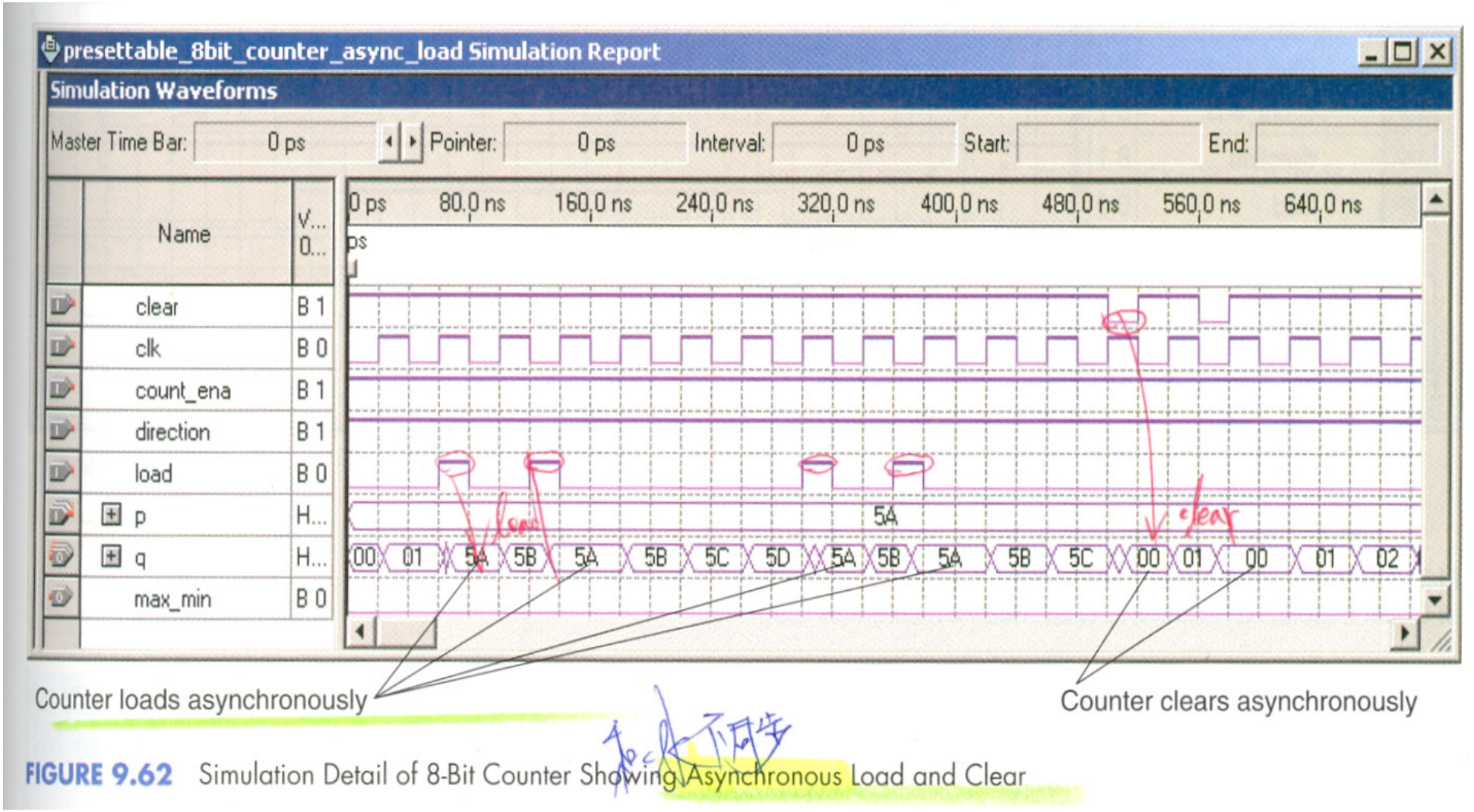


FIGURE 9.62 Simulation Detail of 8-Bit Counter Showing Asynchronous Load and Clear

Simulation Result -3

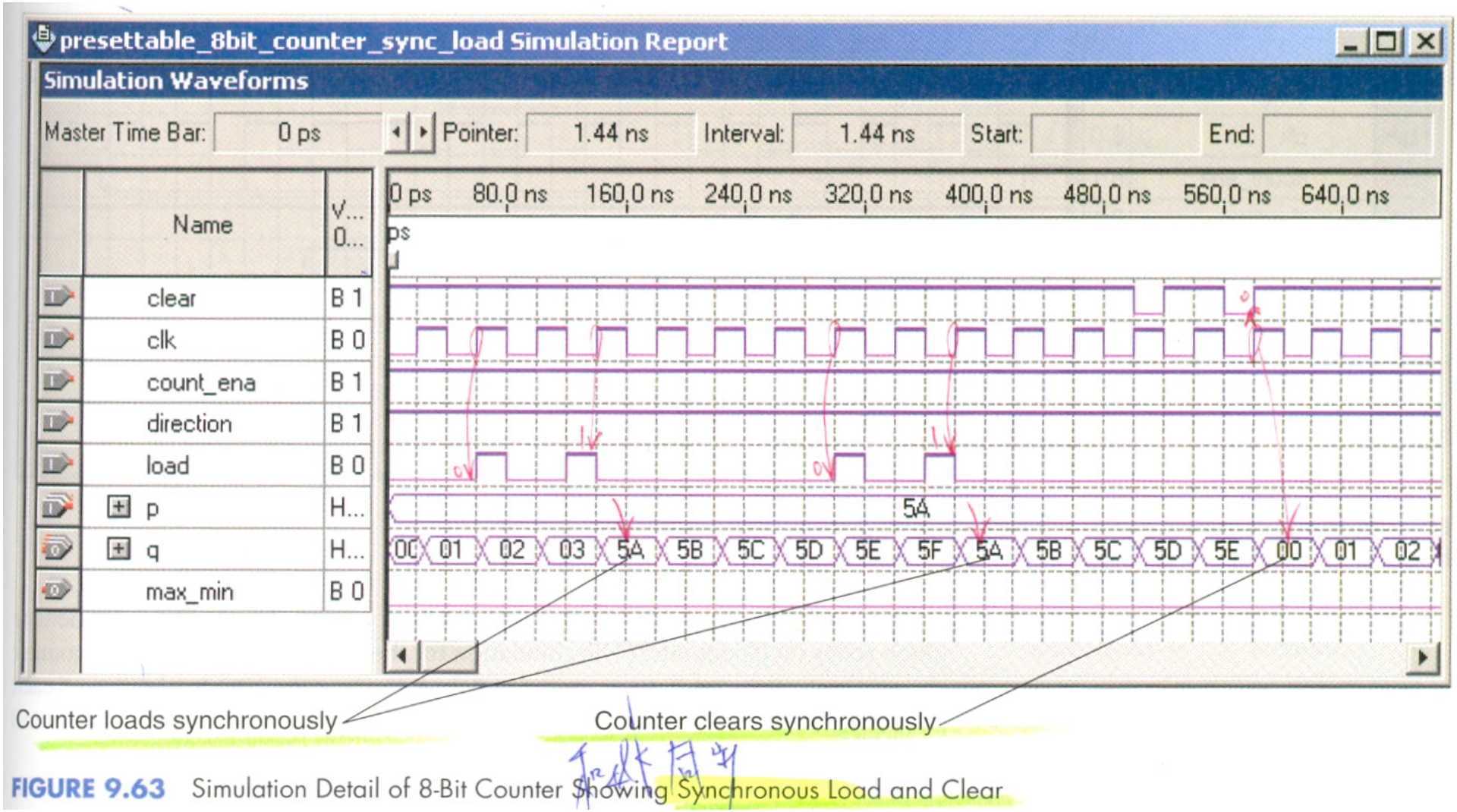


FIGURE 9.63 Simulation Detail of 8-Bit Counter Showing Synchronous Load and Clear

Simulation Result -4

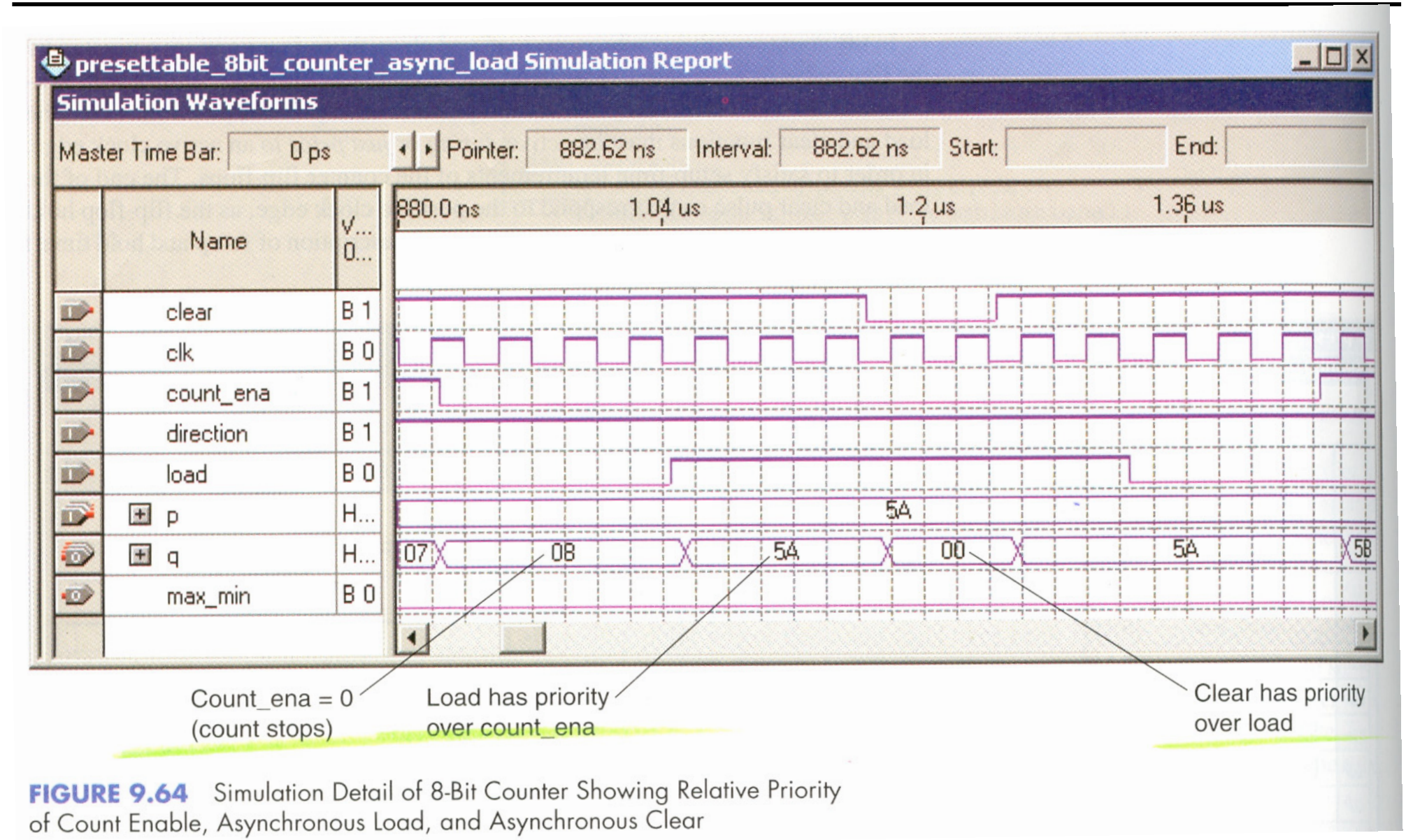


FIGURE 9.64 Simulation Detail of 8-Bit Counter Showing Relative Priority of Count Enable, Asynchronous Load, and Asynchronous Clear

LPM Counter Functions -1

- LPM counters can be used as a simple 8-bit counter.
- The component lpm_counter has a number of other functions that can be implemented using specific ports and parameters. These functions are indicated on Table 9.13.

TABLE 9.13 Available Functions of an LPM counter

Function	Ports	Parameters	Description
Basic count operation	clock, q []	LPM_WIDTH	Output **q[]** increases by one with each positive clock edge. **LPM_WIDTH** is the number of output bits.
Synchronous load	sload, data []	none	When **sload** = 1, output **q[]** goes to the value at input **data[]** on the next positive clock edge. **data[]** has the same width as **q[].**
Synchronous clear	sclr	none	When **sclr** = 1, output **q[]** goes to zero on positive clock edge.
Synchronous set	sset	LPM_SVALUE	When **sset** = 1, output goes to value of **LPM_SVALUE** on positive clock edge. If **LPM_SVALUE** is not specified, **q[]** goes to all 1s.
Asynchronous load	aload, data[]	none	Output goes to value at **data[]** when **aload** = 1.
Asynchronous clear	aclr	none	Output goes to zero when **aclr** = 1.

LPM Counter Functions -2

Asynchronous set	aset	LPM_AVALUE	Output goes to value of **LPM_AVALUE** when aset = 1. If **LPM_AVALUE** is not specified, outputs all go HIGH when **aset** = 1.
Directional control	updown	LPM_DIRECTION	Optional direction control. Default direction is UP. Only one of **updown** and **LPM_DIRECTION** can be used. **updown** = 1 for UP count, updown = 0 for DOWN count. **LPM_DIRECTION** = "UP", "DOWN", or "DEFAULT".
Count enable	cnt_en	none	When **cnt_en** = 1, count proceeds upon positive clock edges. No effect on other synchronous functions **(sload, sclr, sset).** Defaults to "enabled" when not specified.
Clock enable	clk_en	none	All synchronous functions are enabled when **clk_en** = 1. Defaults to "enabled" when not specified.
Modulus control	none	LPM_MODULUS	Modulus of counter is set to value of **LPM_MODULUS.**
Output decoding (BDF or AHDL only; not available in VHDL)	eq[15..0]	none	Sixteen active HIGH decoded outputs, one for each internal counter value from 0 to 15.

Example 9.8 -1

Write a VHDL file for an 8-bit LPM counter with ports for the following functions: asynchronous load, asynchronous clear, directional control, and count enable.

■ Solution

The required VHDL file is as follows. Note that no behavioral descriptions are required for the functions, only a mapping from the defined port names to the entity inputs and outputs.

```
-- pre_lpm8
-- 8-bit presettable counter with asynchronous clear and load,
-- count enable, and a directional control port

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
LIBRARY lpm;
USE lpm.lpm_components.ALL;

ENTITY pre_lpm8 IS
   PORT (
        clk, count_ena           : IN  STD_LOGIC;
        clear, load, direction   : IN  STD_LOGIC;
        p                        : IN  STD_LOGIC_VECTOR(7 downto 0);
        q_out                    : OUT STD_LOGIC_VECTOR(7 downto 0));
END pre_lpm8;
```

Example 9.8 -2

```
ARCHITECTURE a OF pre_lpm8 IS
BEGIN
   counter1: lpm_counter
      GENERIC MAP (LPM_WIDTH => 8)
      PORT MAP ( clock  => clk,
                 updown => direction
                 cnt_en => count_ena
                 data   => p,
                 aload  => load,
                 aclr   => clear,
                 q      => q_out);

END a;
```

Example 9.9 -1

Write a VHDL file that uses an LPM counter to generate a DOWN counter with a modulus of 500. Create a Quartus II simulation file to verify the counter's operation.

■ Solution

A mod-500 counter requires nine bits since $2^9 < 500 < 2^2$. Since the counter always counts DOWN, we can use the parameter LPM_DIRECTION to specify the DOWN counter rather than using an unnecessary port. The required VHDL code follows.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
LIBRARY lpm;
Use lpm.lpm_components.ALL;
```

Example 9.9 -2

```
ENTITY mod5c_lpm IS
   PORT(
      clk : IN  STD_LOGIC;
      q   : OUT STD_LOGIC_VECTOR (8 downto 0) );
END mod5c_lpm;

ARCHITECTURE a OF mod5c_lpm IS
BEGIN
   counter1 : lpm_counter
      GENERIC MAP(LPM_WIDTH     => 9,
                  LPM_DIRECTION => "DOWN",
                  LPM_MODULUS   => 500)
      PORT MAP ( clock => clk,
                 q     => q);
END a;
```

Example 9.9 -3

Figure 9.68 shows a partial simulation of the counter, indicating the point at which the output rolls over from 0 to 499 (decimal).

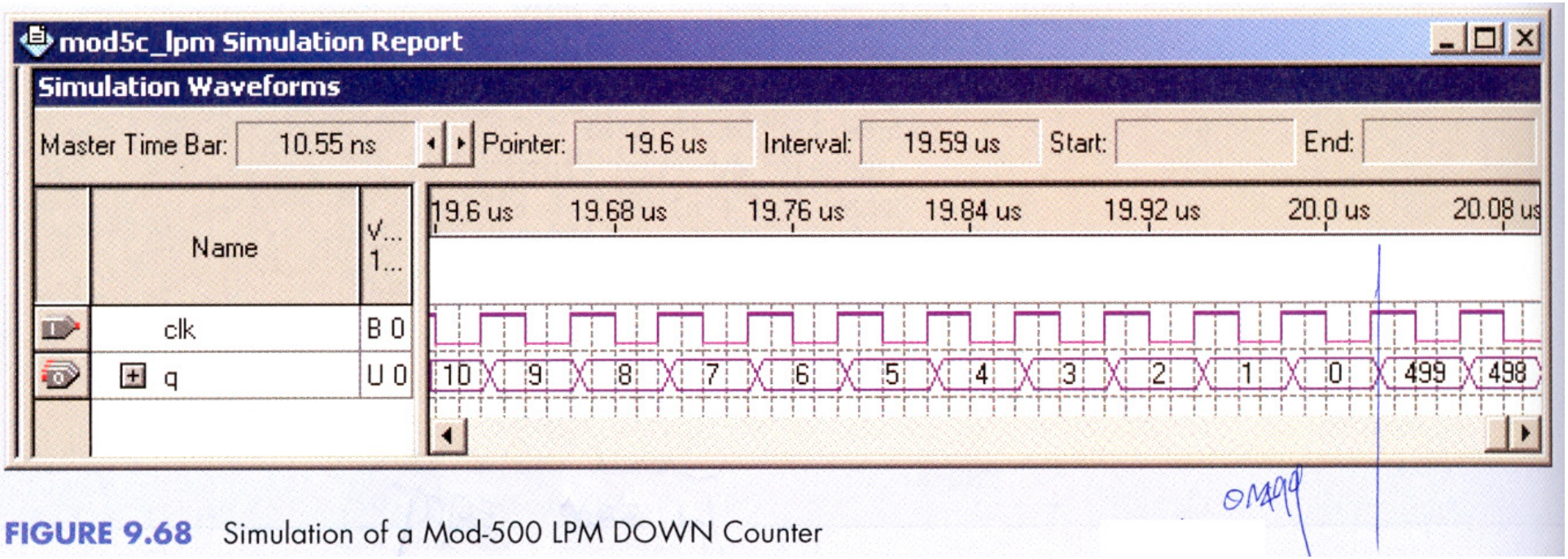


FIGURE 9.68 Simulation of a Mod-500 LPM DOWN Counter

Example 9.10 -1

Write a VHDL file that instantiates a 12-bit LPM counter with asynchronous clear and synchronous set functions. Design the counter to set to 2047 (decimal). Create a simulation to verify the counter operation.

$2^{11} = 2048$

■ **Solution**

The required VHDL file is:

```
-- sset_lpm.vhd
-- 12-bit LPM counter with sset and aclr

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
LIBRARY lpm;
USE lpm.lpm_components.ALL;

ENTITY sset_lpm IS
   PORT(
            clk          : IN  STD_LOGIC;
            clear, set   : IN  STD_LOGIC;
            q            : OUT STD_LOGIC_VECTOR (11 downto 0));
END sset_lpm;

ARCHITECTURE a OF sset_lpm IS
BEGIN
   counter1: lpm_counter
      GENERIC MAP       (LPM_WIDTH => 12,
                         LPM_SVALUE => "2047")
      PORT MAP ( clock => clk,
                 sset  => set,
                 aclr  => clear,
                 q     => q);
END a;
```

Example 9.10 -2

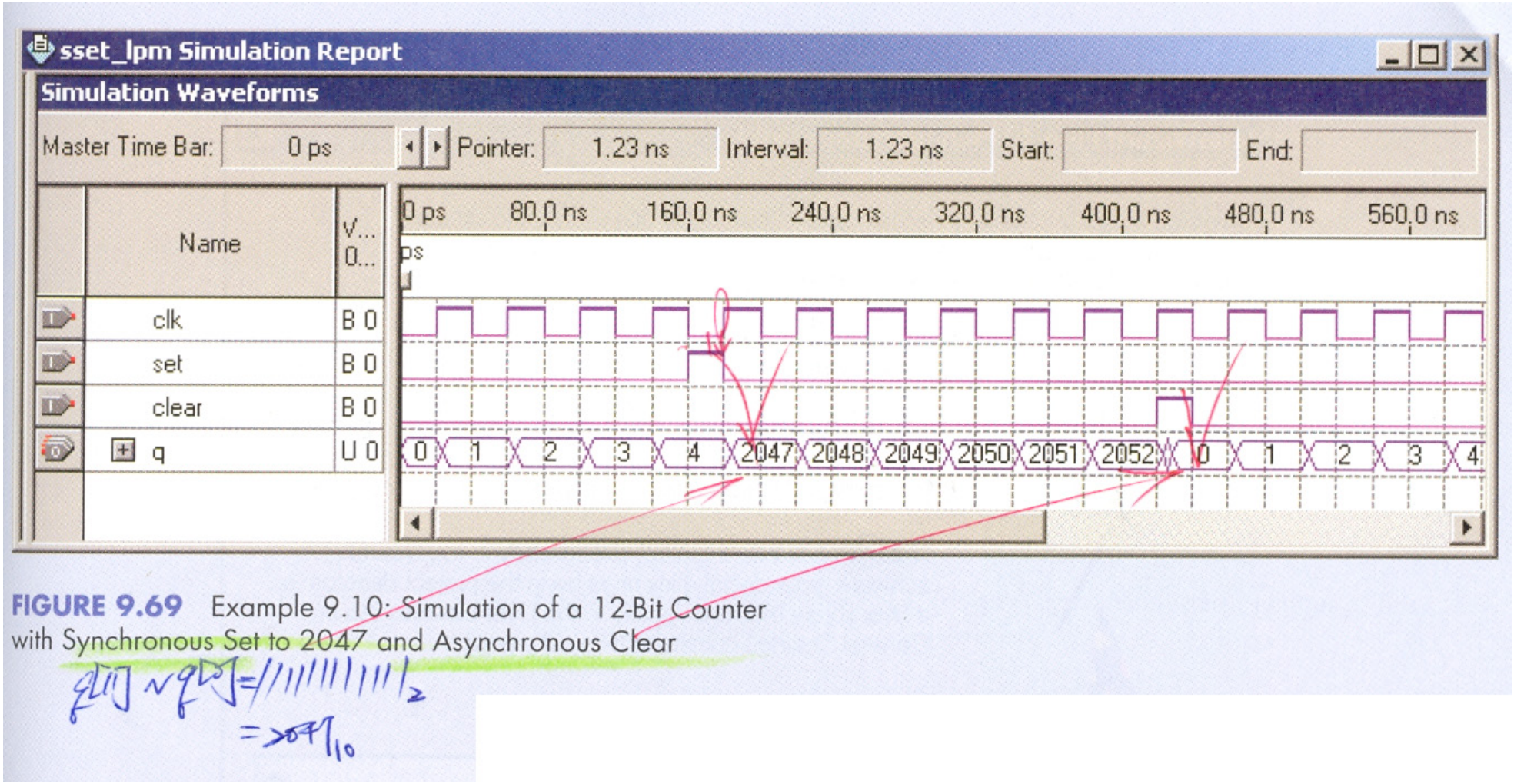


FIGURE 9.69 Example 9.10: Simulation of a 12-Bit Counter with Synchronous Set to 2047 and Asynchronous Clear

9.7 Shift Register
Key Terms -1

Shift Register A synchronous sequential circuit that will store and move *n*-bit data, either serially or in parallel, in *n* flip-flops.

SRG*n* Abbreviation for an *n*-bit shift register (e.g., SRG4 indicates a 4-bit shift register).

Serial Shifting Movement of data from one end of a shift register to the other at a rate of one bit per clock pulse.

Parallel Transfer Movement of data into all flip-flops of a shift register at the same time.

Rotation Serial shifting of data with the output(s) of the last flip-flop connected to the synchronous input(s) of the first flip-flop. The result is continuous circulation of the same data.

Key Terms -2

Right Shift A movement of data from the left to the right in a shift register. (Right is defined in Quartus II as toward the LSB.)

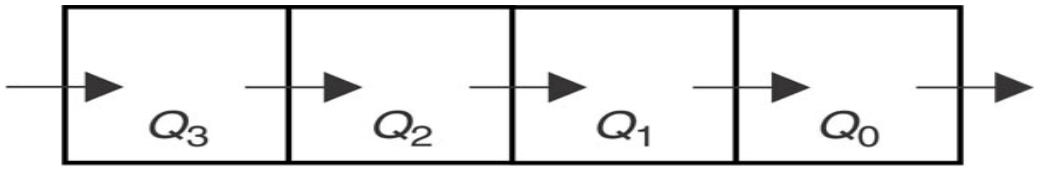
Left Shift A movement of data from the right to the left in a shift register. (Left is defined in Quartus II as toward the MSB.)

Bidirectional Shift Register A shift register that can serially shift bits left or right according to the state of a direction control input.

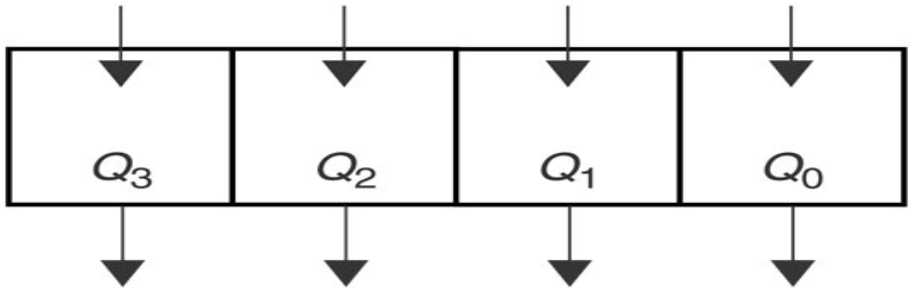
Parallel-Load Shift Register A shift register that can be preset to any value by directly loading a binary number into its internal flip-flops.

Universal Shift Register A shift register that can operate with any combination of serial and parallel inputs and outputs (i.e., serial in/serial out, serial in/parallel out, parallel in/serial out, parallel in/parallel out). A universal shift register is often bidirectional, as well.

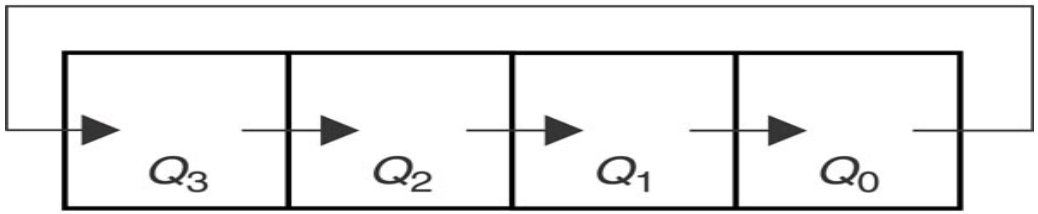
Data Movement in a 4-Bit Shift Register



a. Serial shifting



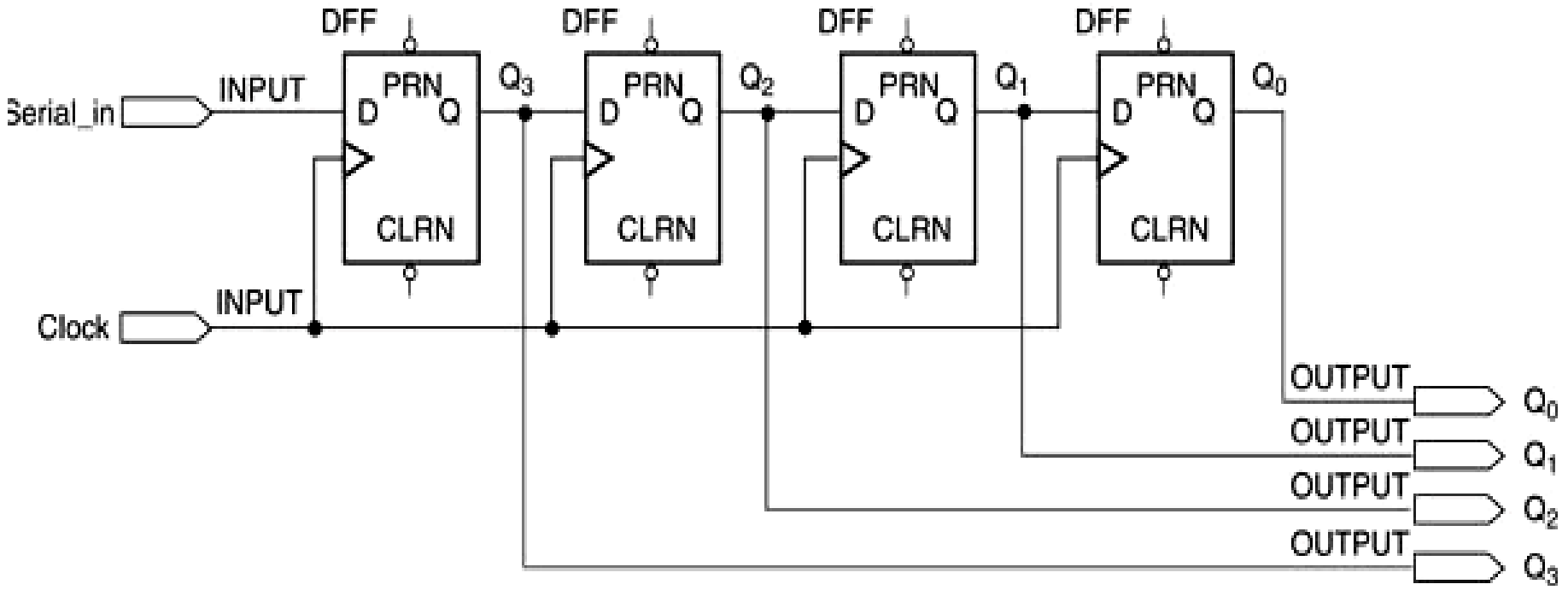
b. Parallel transfer



c. Rotation

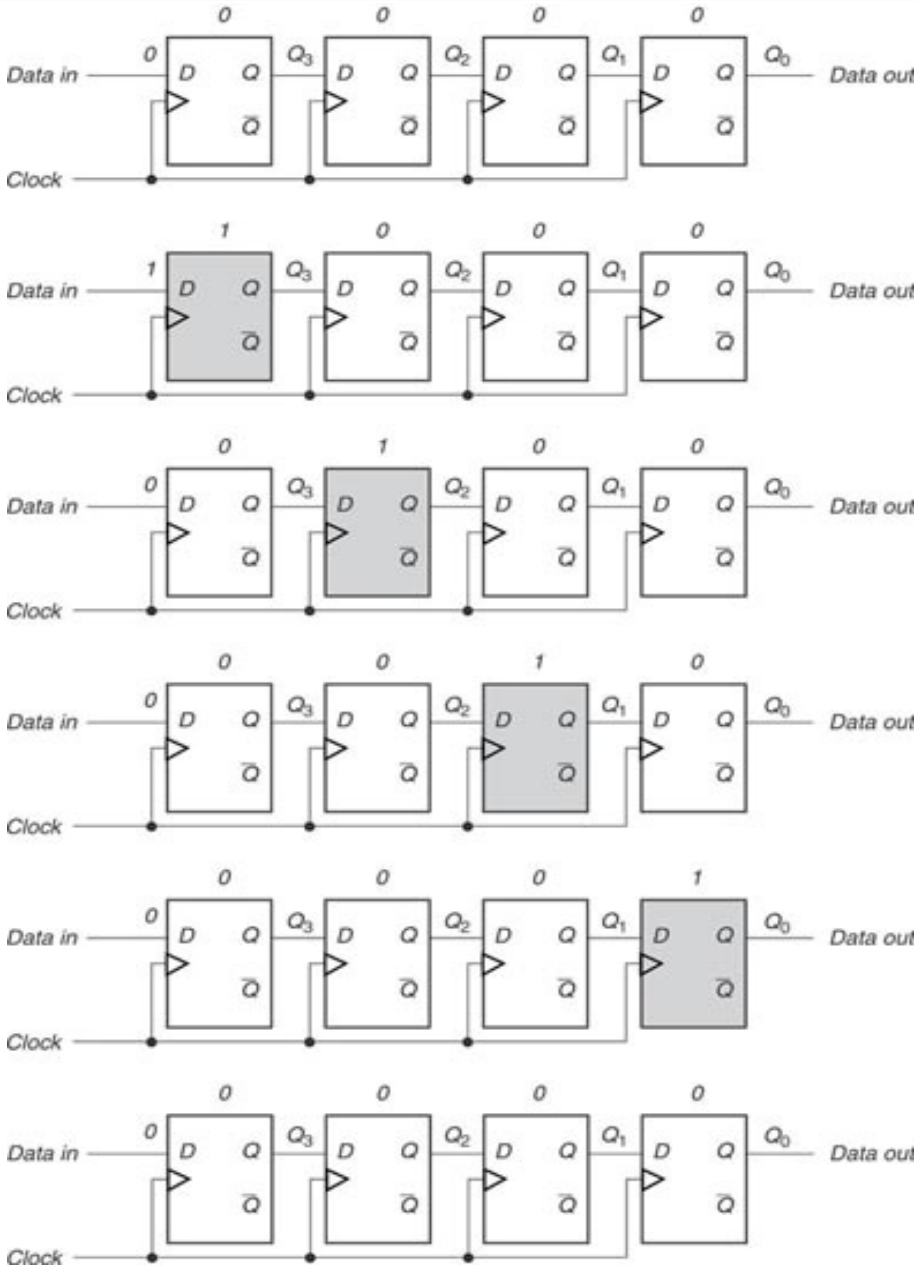
Serial Shift Registers -1

- 4-bit serial shift register configured to shift right



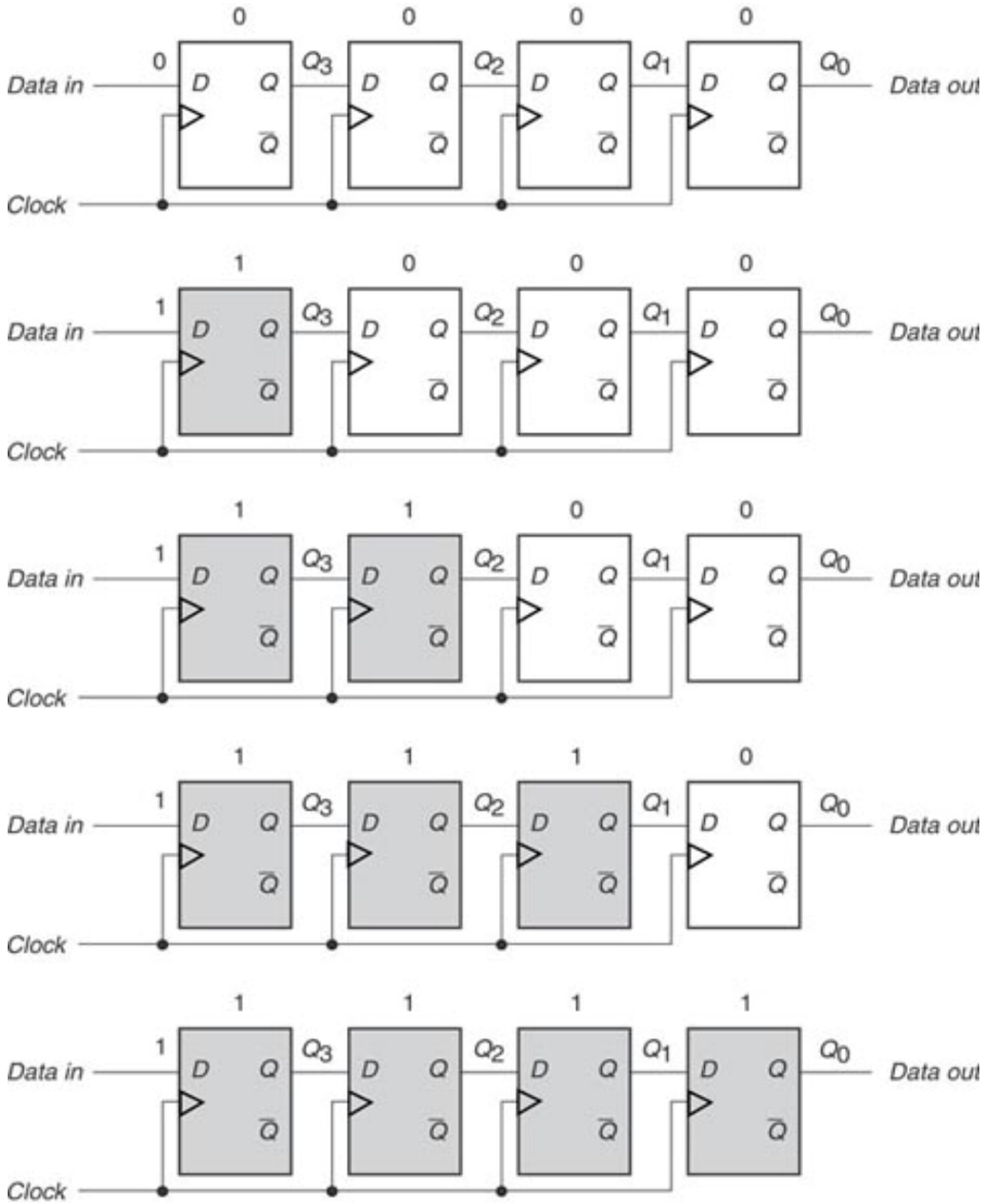
Serial Shift Registers -2

00001 →

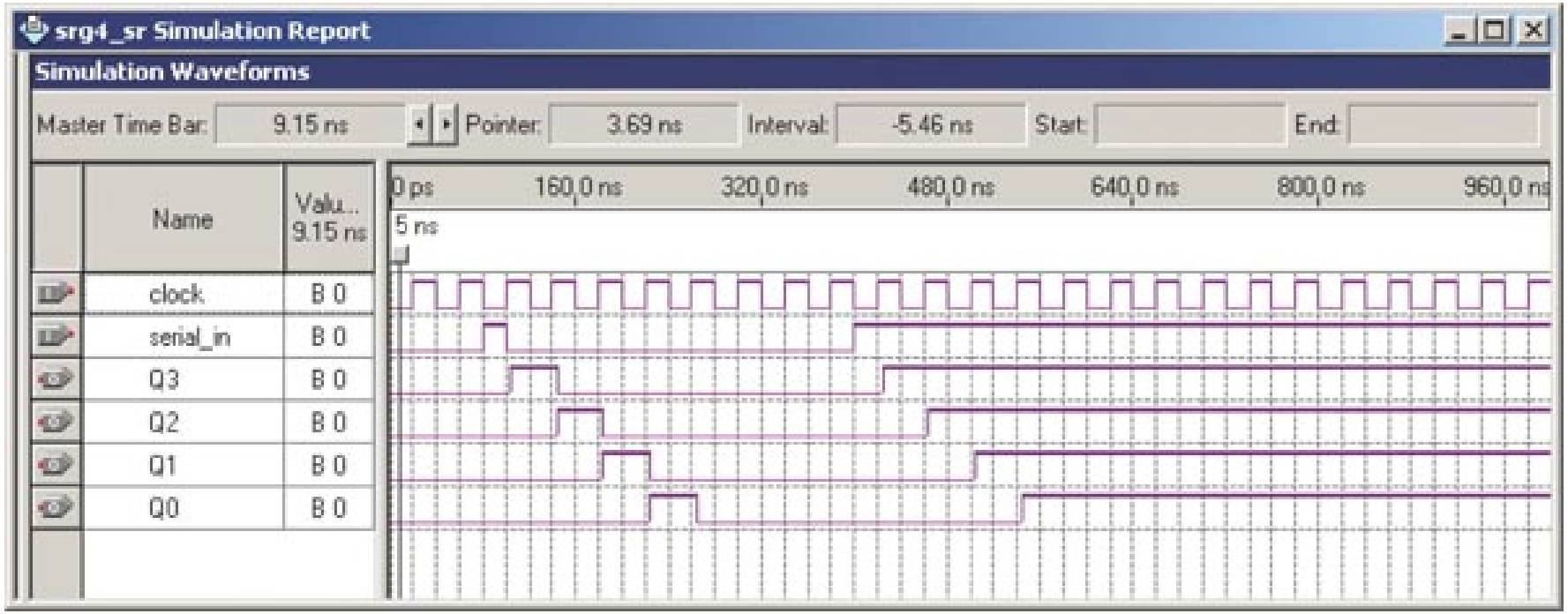


Serial Shift Registers -3

1111 ⟶



Serial Shift Registers -4



Bidirectional Shift Register – 1

- Uses a control input signal called direction to change circuit function from shift right to shift left.

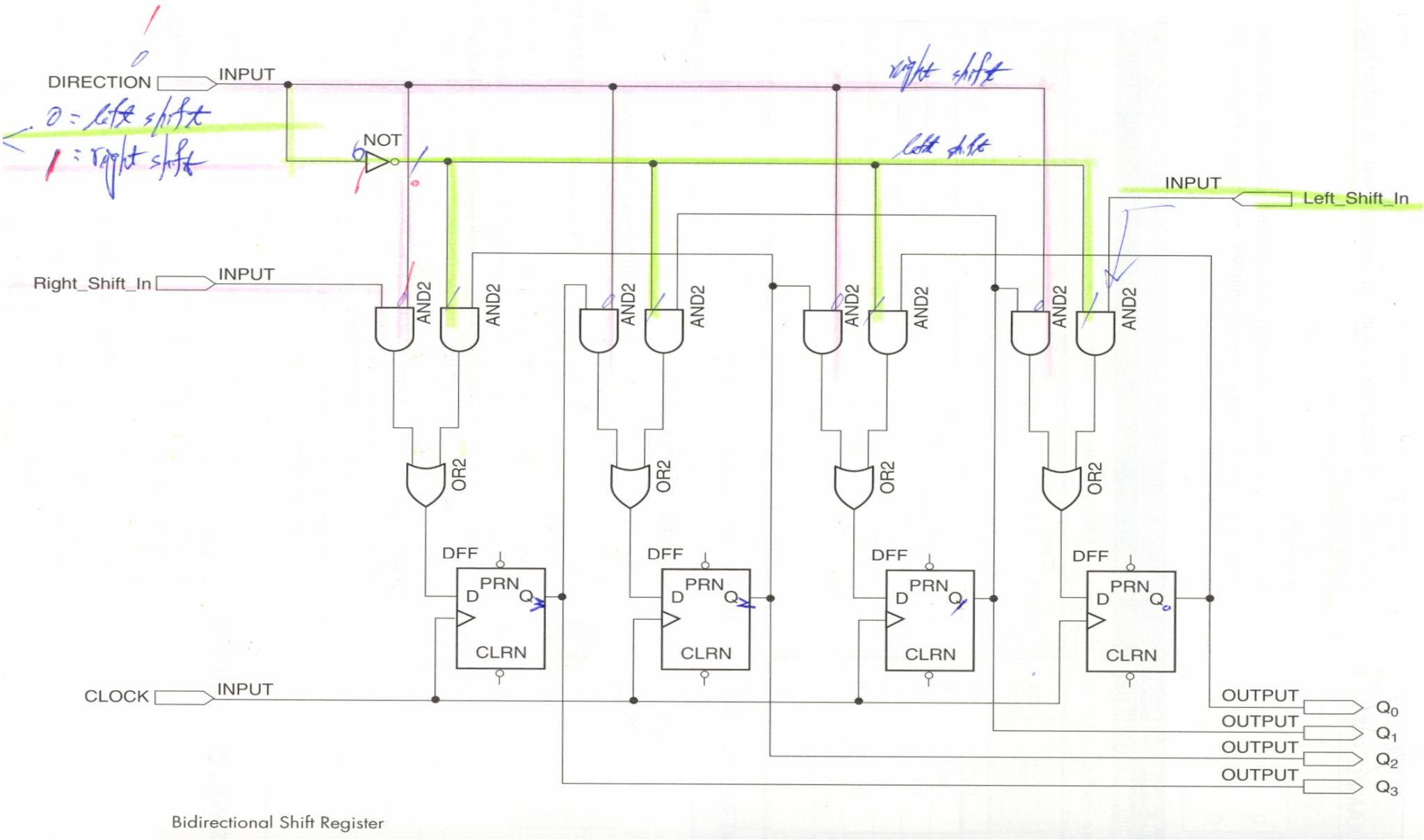
- When DIR = 0, the path of Left_Shift_In is selected.

$$Q_3 \Leftarrow Q_2 \Leftarrow Q_1 \Leftarrow Q_0 \Leftarrow$$

$$Q_3 \Rightarrow Q_2 \Rightarrow Q_1 \Rightarrow Q_0 \Rightarrow$$

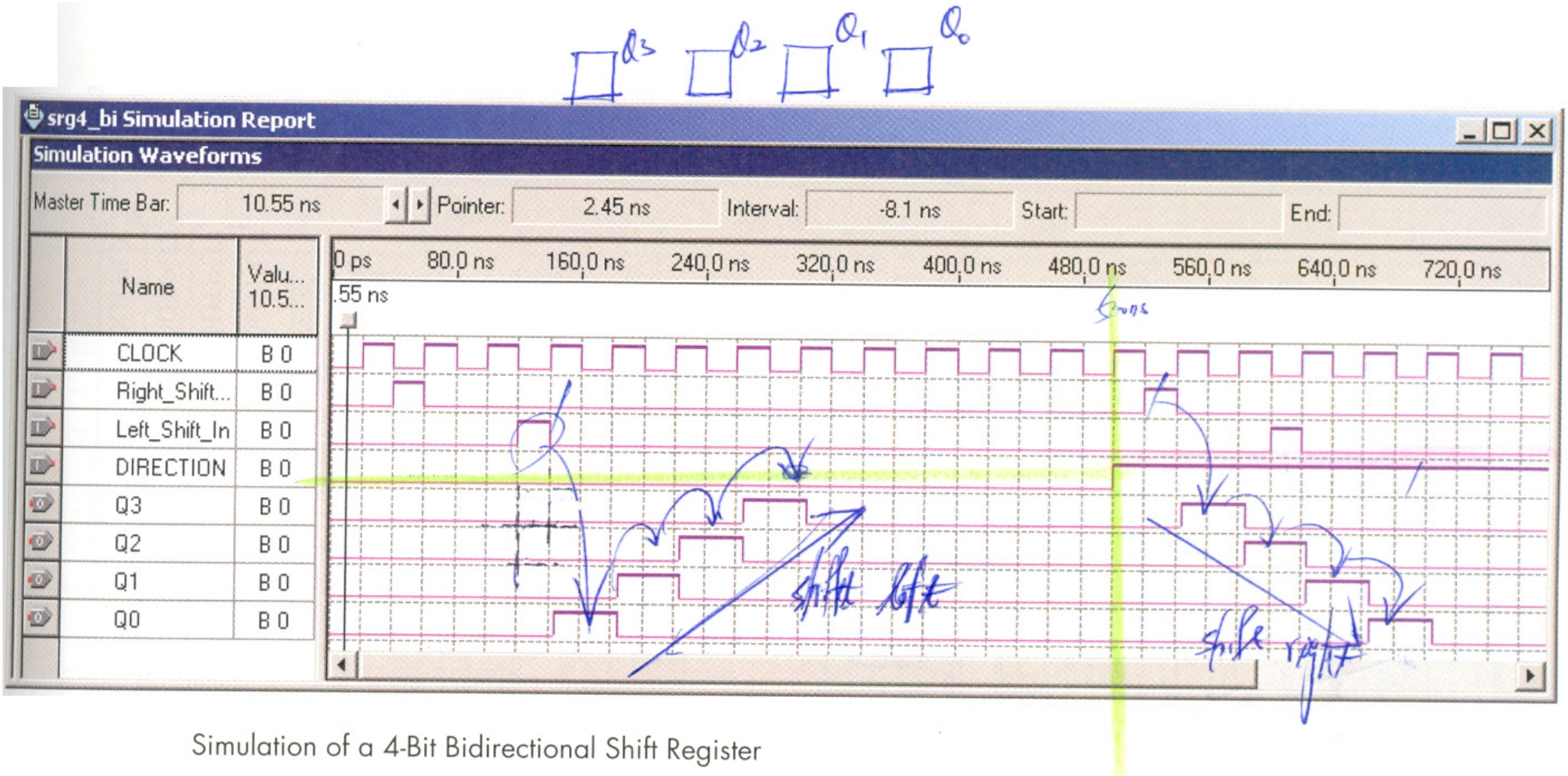
- When DIR = 1, it selects the Right Shift In Path.

Bidirectional Shift Register – 2



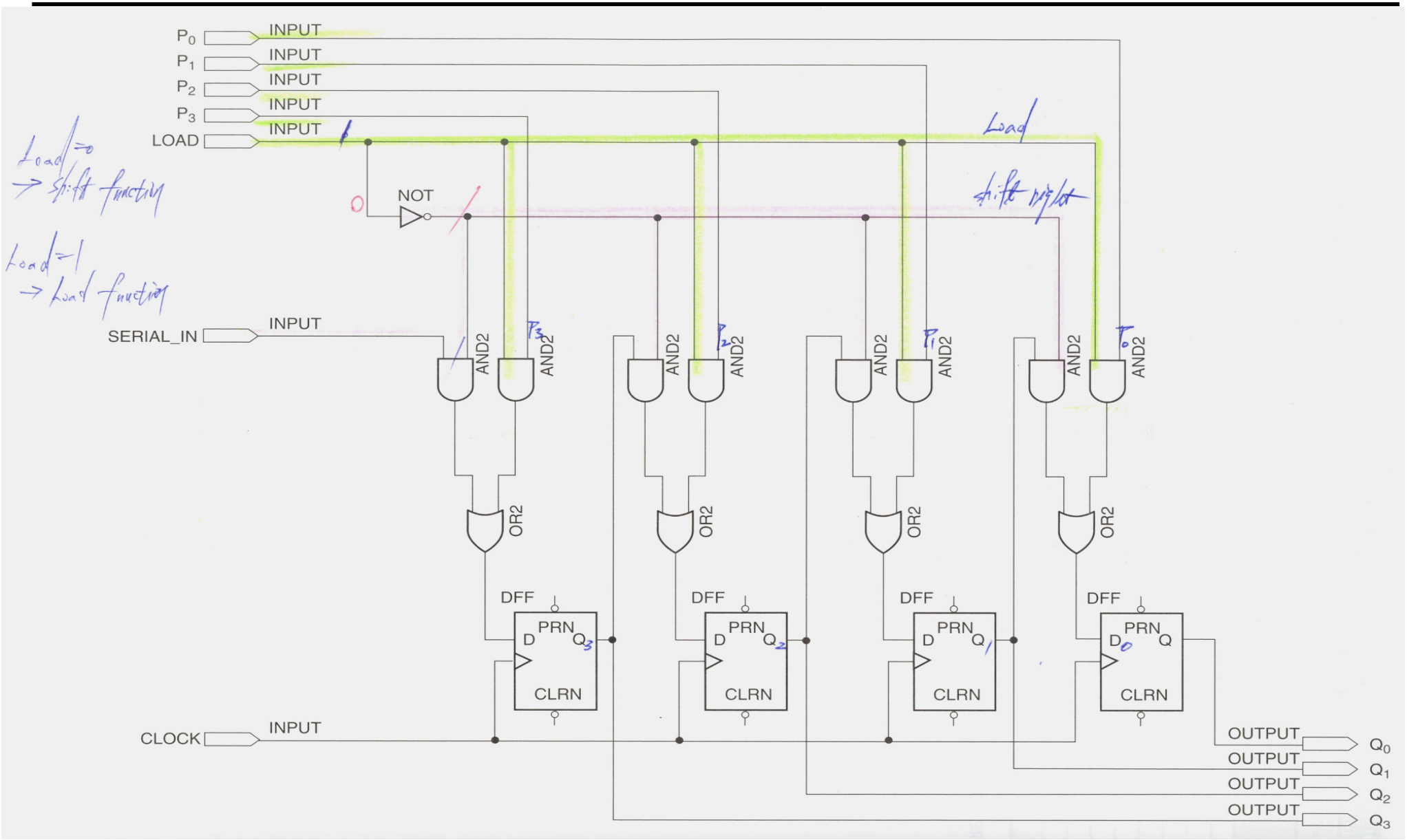
Bidirectional Shift Register

Bidirectional Shift Register – 3

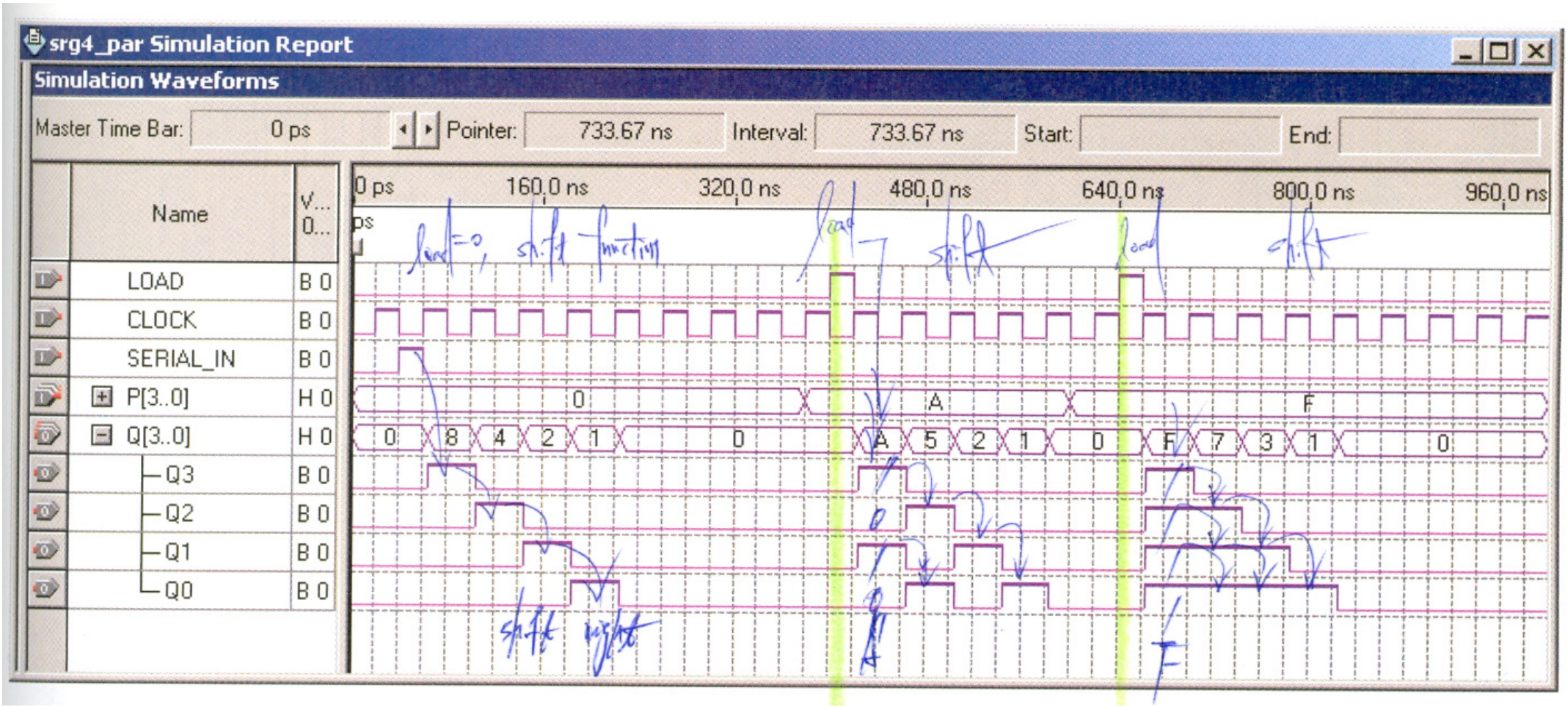


Simulation of a 4-Bit Bidirectional Shift Register

Shift Register with Parallel Load -1



Shift Register with Parallel Load -2



Universal Shift Register -1

- Combines the basic functions of a Parallel Load SR with a Bi-Directional SR.

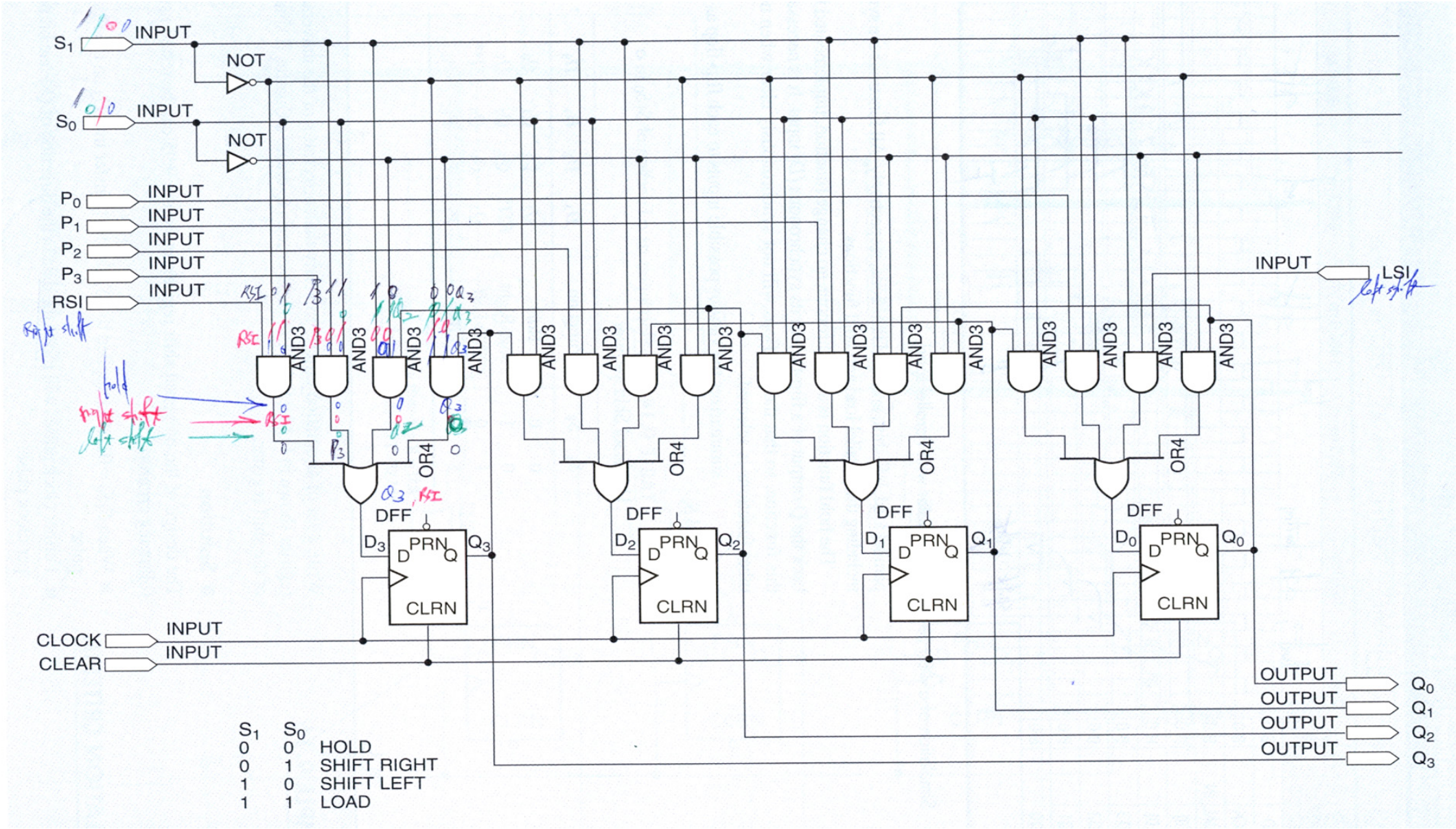
- Uses Two Control Inputs (S_1,S_0) to select the function

TABLE 9.14 Flip-Flop Inputs as a Function of S_1S_0 in a Universal Shift Register

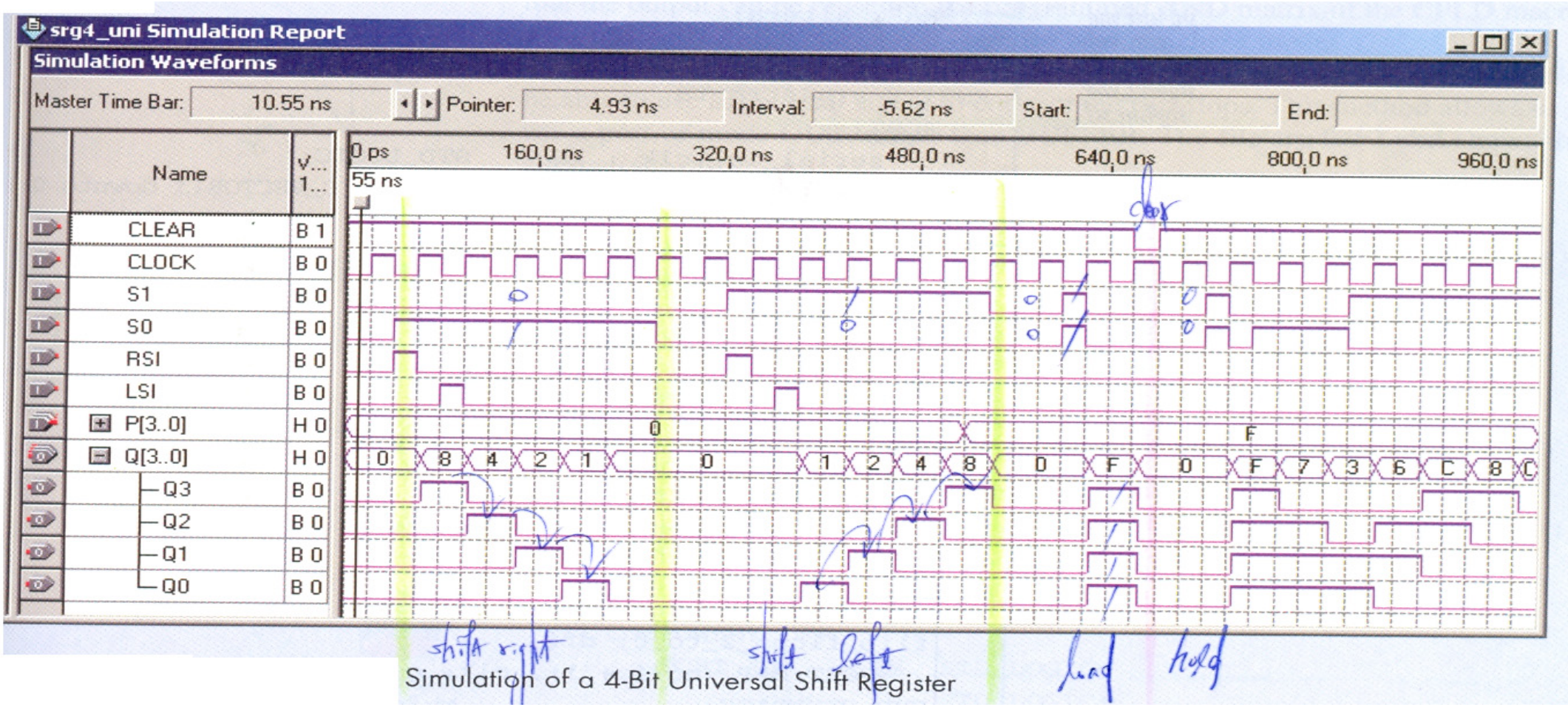
S_1	S_0	Function	D_3	D_2	D_1	D_0
0	0	Hold	Q_3	Q_2	Q_1	Q_0
0	1	Shift Right	*RSI**	Q_3	Q_2	Q_1
1	0	Shift Left	Q_2	Q_1	Q_0	*LSI***
1	1	Load	P_3	P_2	P_1	P_0

**RSI* = Right-shift input
***LSI* = Left-shift input

Universal Shift Register -2



Universal Shift Register -3



Simulation of a 4-Bit Universal Shift Register

9.8 Programming Shift Registers in VHDL

KEY TERMS

Structural Design A VHDL design technique that connects predesigned components using internal signals.

Dataflow Design A VHDL design technique that uses Boolean equations to define relationships between inputs and outputs.

Behavioral Design A VHDL design technique that uses descriptions of required behavior to create the design.

Structure Design -1

Structural design is like taking components out of a bin and connecting them together to make a circuit. We can use the **DFF** component from the Quartus II primitives library and instantiate enough components to make a shift register, with connections made by internal signals.

Method1:

```
-- srg4strc.vhd
-- Structural description of a 4-bit serial shift register
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
LIBRARY altera;
USE altera.maxplus2.ALL;

ENTITY srg4strc IS
    PORT(
        serial_in, clk : IN     STD_LOGIC;
        qo             : BUFFER STD_LOGIC_VECTOR(3 downto 0));
END srg4strc;

ARCHITECTURE right_shift of srg4strc IS
    COMPONENT DFF
        PORT (d : IN  STD_LOGIC;
              clk : IN  STD_LOGIC;
              q   : OUT STD_LOGIC);
    END COMPONENT;
BEGIN
    flip_flop_3: dff
        PORT MAP (serial_in, clk, qo(3) );

    dffs:
    FOR i IN 2 downto 0 GENERATE
    flip_flops_2_to_0: dff
        PORT MAP (qo(i + 1), clk, qo(i) );
    END GENERATE;
END right_shift;
```

Structure Design -2

Method2:

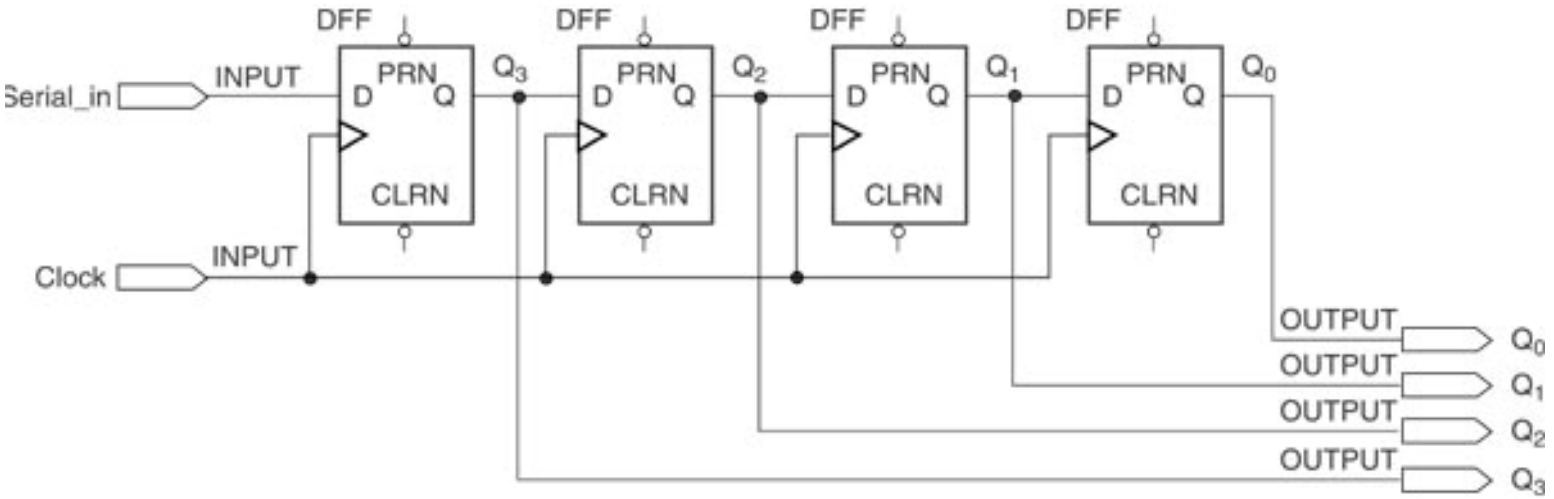
```
flip_flop_3: dff
   PORT MAP (serial_in, clk, qo(3) );
flip_flop_2: dff
   PORT MAP(qo(3), clk, qo(2) );
flip_flop_1: dff
   PORT MAP(qo(2), clk, qo(1) );
flip_flop_0: dff
   PORT MAP(qo(1), clk, qo(0) );
```

Dataflow Design -1

(用 Boolean relationships)

Dataflow design describes a design entity in terms of the Boolean relationships between different parts of the circuit. The Boolean relationships in a 4-bit shift register are defined by the expressions for the flip-flop synchronous inputs:

$$D3 = \text{serial\_in}$$
$$D2 = Q3$$
$$D1 = Q2$$
$$D0 = Q1$$



Dataflow Design -2

```
-- srg4dflw.vhd
-- Dataflow description of a 4-bit serial shift register
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY srg4dflw IS
   PORT (
      serial_in, clk : IN     STD_LOGIC;
      q              : BUFFER STD_LOGIC_VECTOR(3 downto 0) );
END srg4dflw;

ARCHITECTURE right_shift of srg4dflw IS
   SIGNAL d : STD_LOGIC_VECTOR(3 downto 0);
BEGIN
   PROCESS (clk)
   BEGIN
      -- Define a 4-bit D flip-flop
      IF clk'EVENT and clk = '1' THEN
         q <= d;
      END IF;
   END PROCESS;

   d <= serial_in & q(3 downto 1);
END right_shift;
```

Dataflow Design -3

A signal assignment statement implements the Boolean equations for the shift register. It is written as a single statement for efficiency, but could also be written as four separate assignment statements, as follows:

```
d(3) <= serial_in;
d(2) <= q(3);
d(1) <= q(2);
d(0) <= q(1);
```

We must define **q** as mode BUFFER, since we are using it as both input and output.

Behavior Design

Next States of Flip-Flops in a Serial Shift Register

Q_3	Q_2	Q_1	Q_0
serial_in	Q_3	Q_2	Q_1

```
-- srg4behv.vhd
-- Behavioral description of a 4-bit serial shift register
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY srg4behv IS
   PORT (
      serial_in, clk : IN     STD_LOGIC;
      q              : BUFFER STD_LOGIC_VECTOR (3 downto 0) );
END srg4behv;

ARCHITECTURE right_shift of srg4behv IS
BEGIN
   PROCESS (clk)
   BEGIN
      IF (clk'EVENT and clk = '1') THEN
         q <= serial_in & q(3 downto 1);
      END IF;
   END PROCESS;
END right_shift;
```

In the behavioral design, we are not concerned with the flip-flop inputs or other internal connections; the behavioral description is sufficient for the VHDL compiler to synthesize the required hardware. Compare this to the dataflow description, where we created a set of flip-flops, then assigned Boolean functions to the *D* inputs. In this case, the behavioral design method combines these two steps into one.

Shift Registers of Generic Width -1

- Uses a VHDL Generic Clause in the Entity to specify a Width Variable. General form is GENERIC
  - (Clause := *Value*)
- For a 4-Bit SR we use GENERIC.
  - (Width : Positive := 4).

Shift Registers of Generic Width -2

Ex1:

```
ENTITY srt_bhv IS
   GENERIC (width : POSITIVE := 4);
   PORT (
      serial_in, clk : IN     STD_LOGIC;
      q              : BUFFER STD_LOGIC_VECTOR (width-1 downto 0));
END srt_bhv;

ARCHITECTURE right_shift of srt_bhv IS
BEGIN
   PROCESS (clk)
   BEGIN
      IF (clk'EVENT and clk = '1') THEN
         q(width-1 downto 0) <= serial_in & q(width-1 downto 1);
   END IF;
END PROCESS;
END right_shift;
```

Shift Registers of Generic Width -3

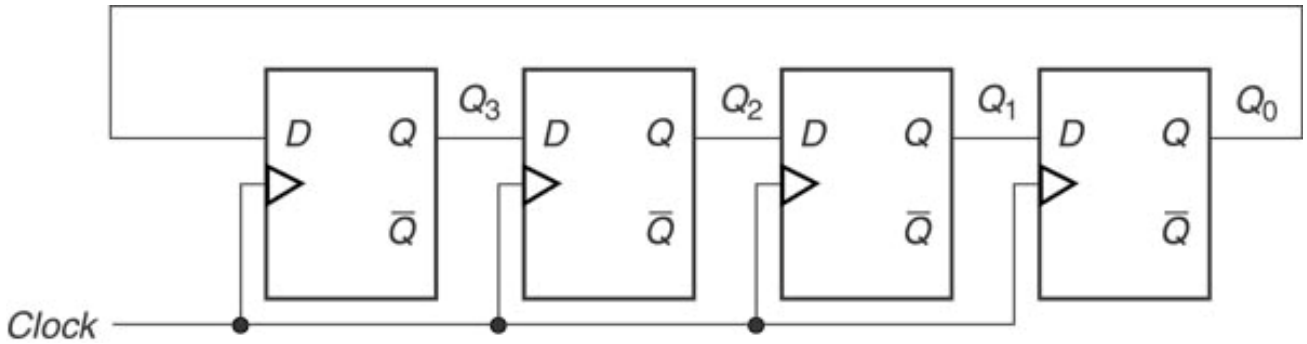
Ex2:

```
-- srt8_bhv.vhd
-- 8-bit shift register that instantiates srt_bhv
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY srt8_bhv IS
   PORT (
        data_in, clock : IN     STD_LOGIC;
        qo             : BUFFER STD_LOGIC_VECTOR(7 downto 0));
END srt8_bhv;

ARCHITECTURE right_shift of srt8_bhv IS
COMPONENT srt_bhv  ( instantiates)
   GENERIC (width : POSITIVE);
   PORT (
      serial_in, clk : IN  STD_LOGIC;
      q              : OUT STD_LOGIC_VECTOR(7 downto 0));
END COMPONENT;
BEGIN
   Shift_right_8: srt_bhv
      GENERIC MAP (width=> 8)
      PORT MAP (serial_in => data_in,
                clk       => clock,
                q         => qo);
END right_shift;
```

9.9 Shift Register Counters

- Two types: Ring and Johnson
- Ring Counter: A serial Shift Register with feedback from the output of the last FF to the input of the first FF.
- Johnson Counter: A serial Shift Register with complemented feedback from the output of the last FF to the input of the first FF.

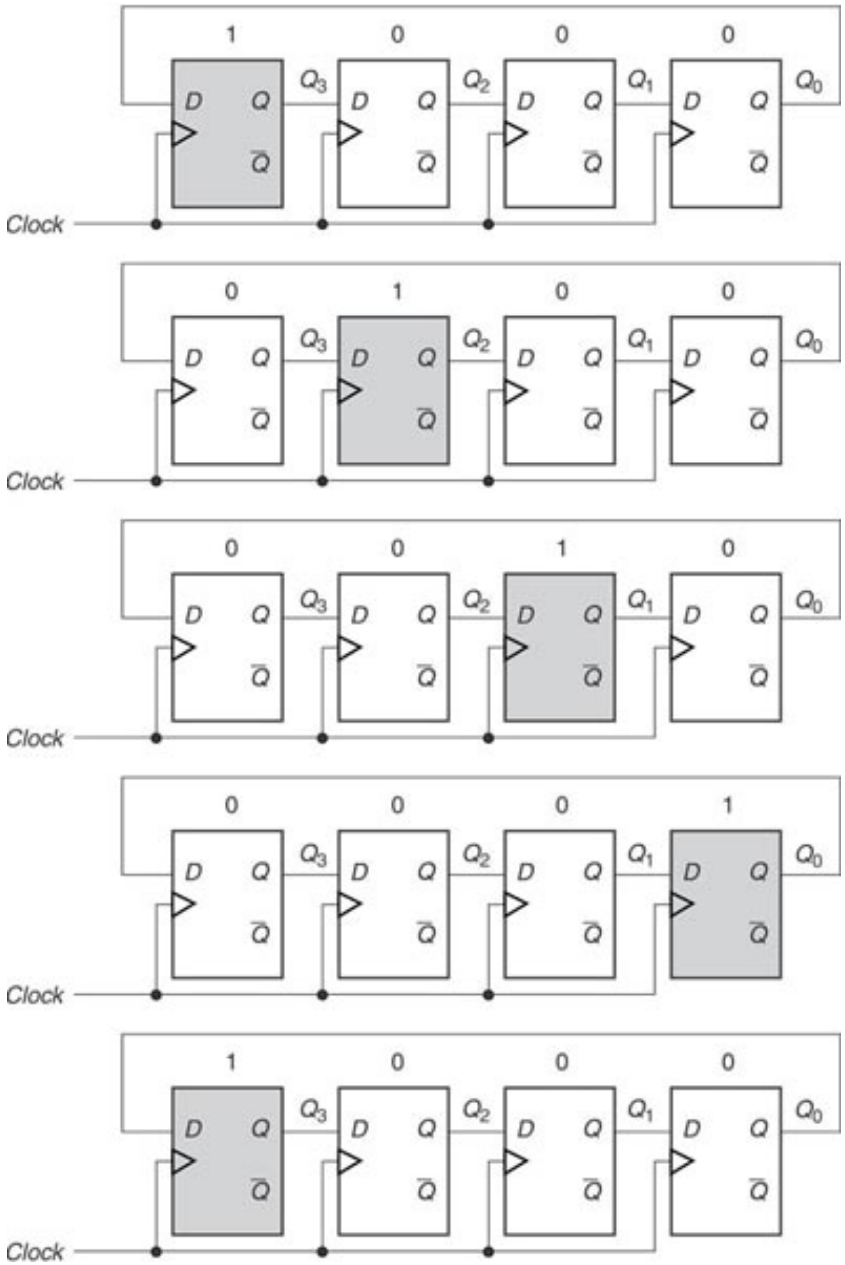
Ring Counters – 1



- A basic Ring Counter is constructed of D-FF with a Feedback Loop.
- Data is initially loaded into the SR by using either Resets or Presets.
- The counter can circulate a 0 or 1 by loading a 1000 or 0111.

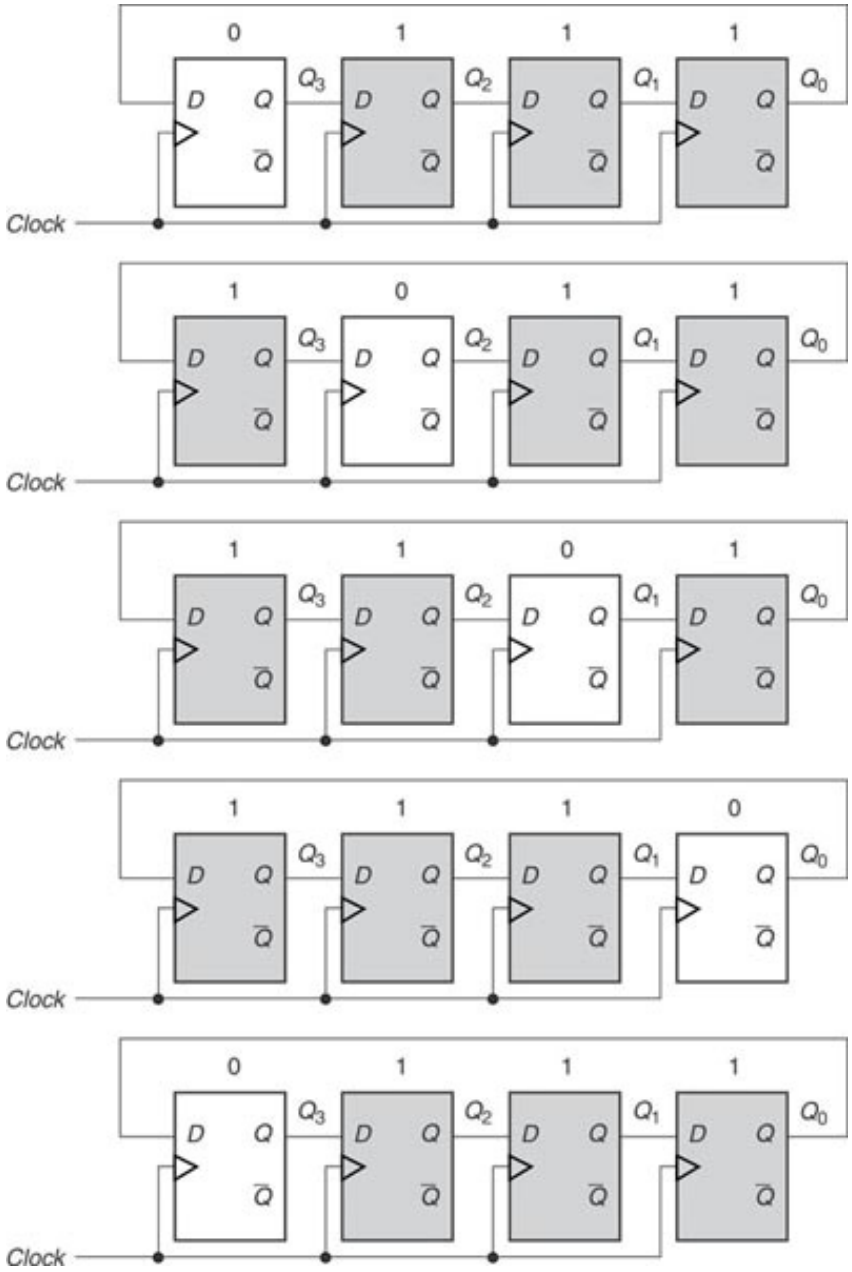
Ring Counters – 2

- Circulating a 1 in a ring counter:
  → loading a 1000



Ring Counters – 3

- Circulating a 0 in a ring counter:
  → loading a 0111



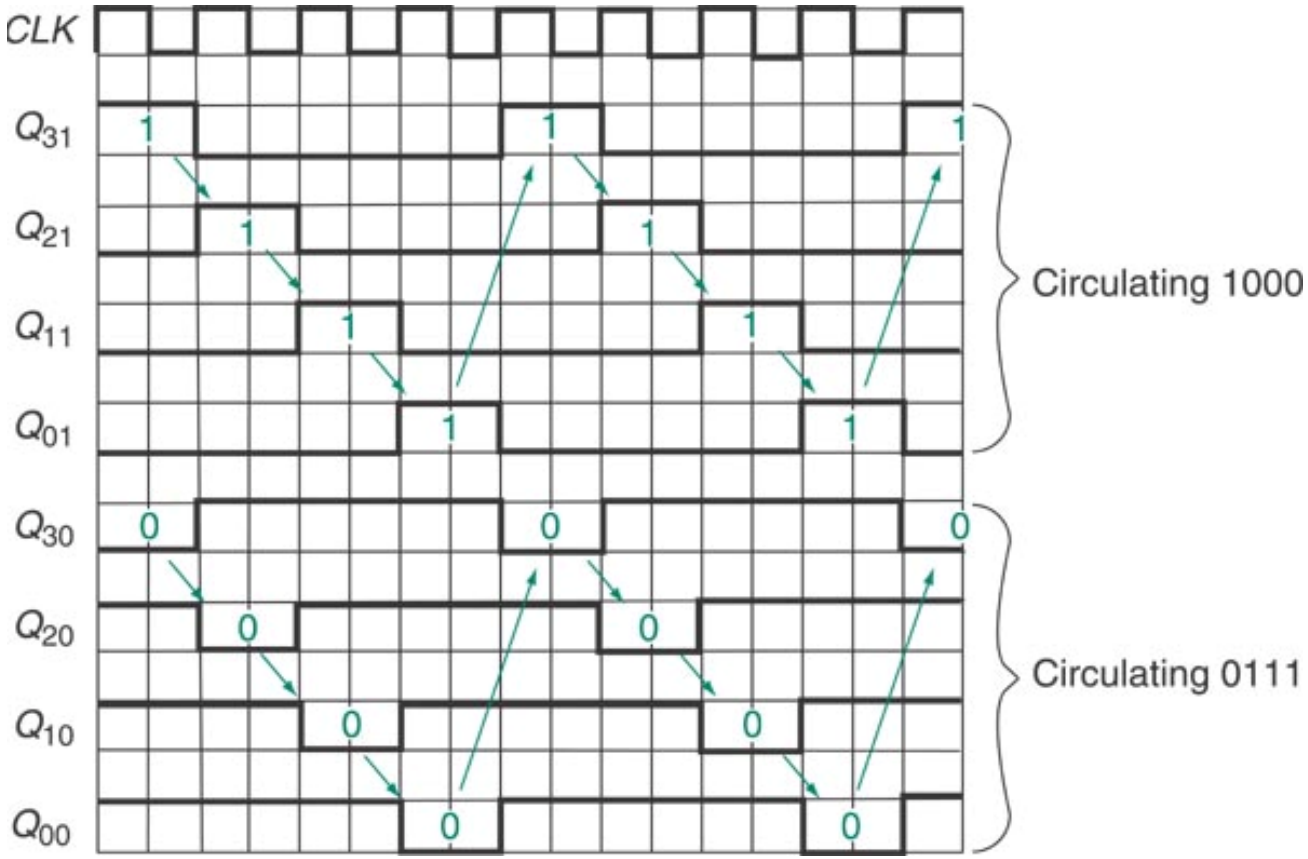
Ring Counters – 4

- The Modulus of a Ring Counter is defined as the maximum number of unique states.

- Modulus is dependent on the initial load value {1000, 0100, 0010, 0001} = Mod4 while {1010, 0101} = Mod2.

- Typically an N-FF Ring Counter has N-States, not 2^N like a binary counter.

Ring Counters – 5

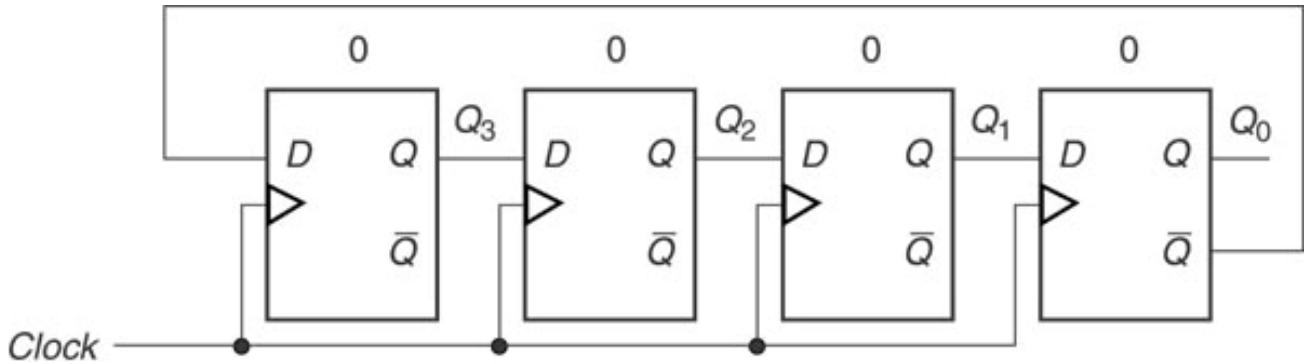


Johnson Counters – 1

- Johnson Counter: A serial shift register with the complemented feedback from the output of the last FF to the input of the first FF.

- Same as the Ring Counter sequences based on a continuous rotation of data through the SR.

Johnson Counters – 2

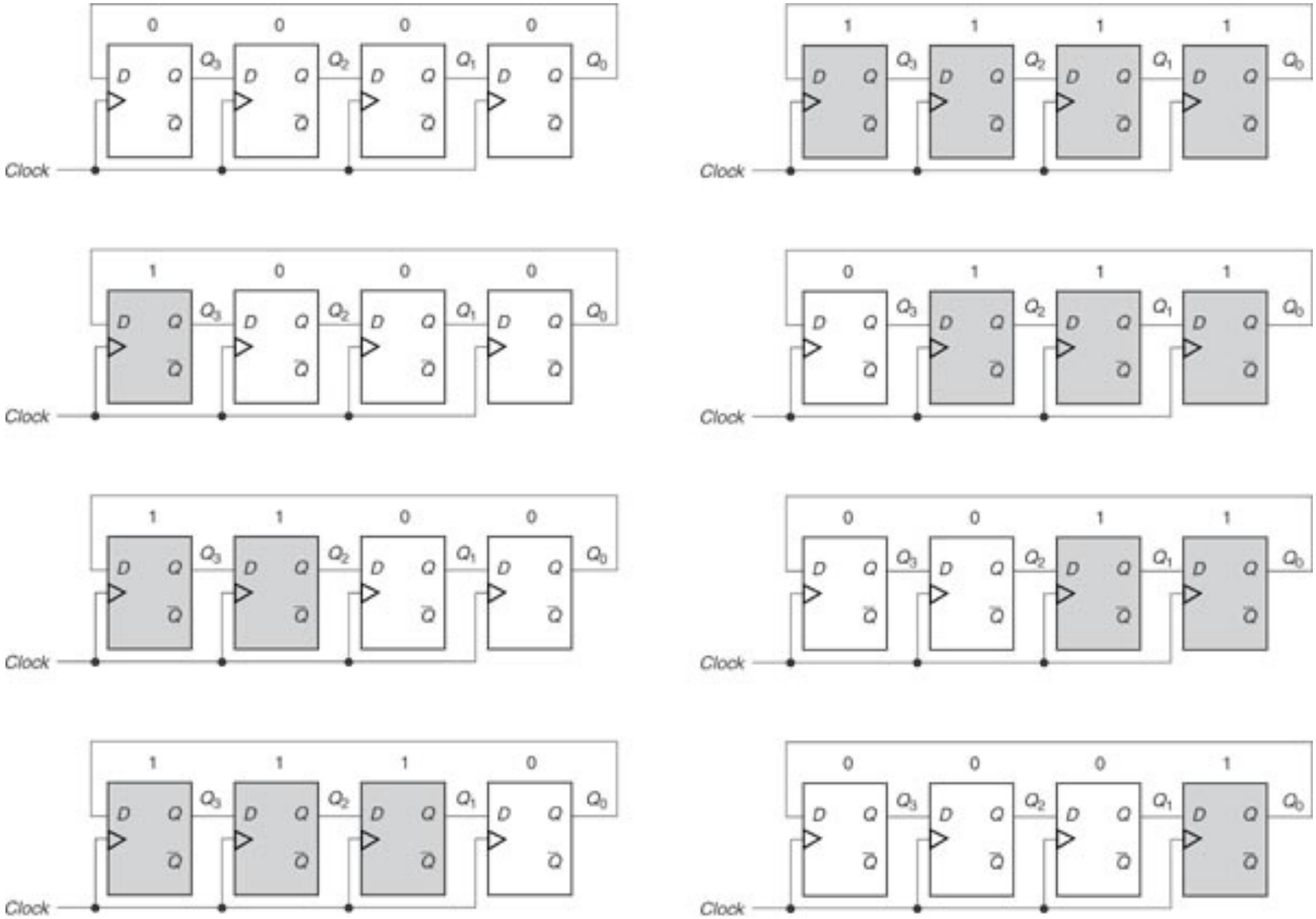


- Same as a Ring Counter except that (Complement) is fed back to D_3, not to Q_0.
- Adds a complement or "twist" to the data and is called a Twisted Ring Counter.
- Usually Initialized with 0000 by a Clear.

Johnson Counters – 3

- Typically has more states than a ring counter.
- Sequence of states = {0000, 1000, 1100, 1110, 1111, 0111, 0011, 0001}.
- Maximum Modulus is $2n$ for a circuit with n flip-flops.

Johnson Counters – 4



HW