



Special Sequential Circuits: Registers



Overview

- Basic Register
- Register with Parallel Load
 - Multiplexer-based Loading
- Shift Register
- Shift Register with Parallel Load
- Bidirectional Shift Register
- Application of Registers
 - Serial-to-Parallel or Parallel-to-Serial Data Conversion
 - Implementing Microoperations in Modern Processors
- Summary



Registers

- The most commonly used sequential devices
 - They are a good example of sequential analysis and design.
 - They are also frequently used in building larger sequential circuits.
- **Registers** hold larger quantities of data than individual flip-flops.
 - Registers are central to the design of modern processors.
 - There are many different kinds of registers.
 - We will show some applications of these special registers.

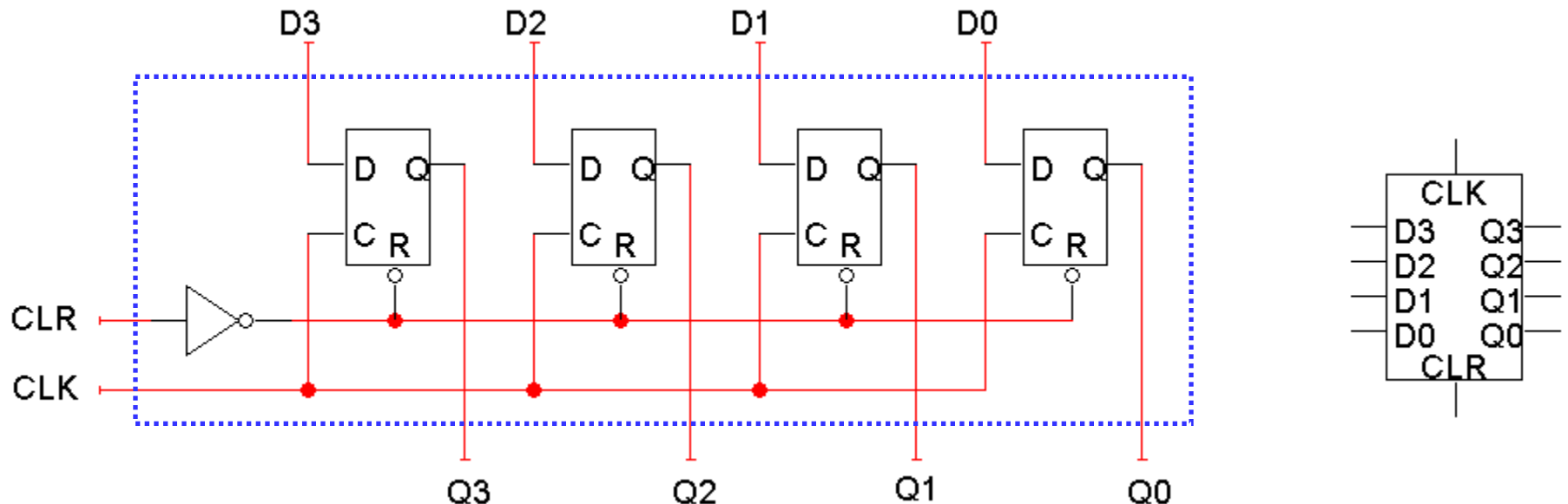


What are registers good for?

- Flip-flops are limited, they can store only one bit
 - Computers work with integers and floating-point numbers
 - These numbers are 32-bits or 64-bit long
- A **register** is an extension of a flip-flop that can store multiple bits.
 - *n*-bit *register* is a set of *n* flip-flops
 - capable of storing *n* bits of binary information
- With added combinational gates, the register can perform data-processing tasks
- Registers are commonly used as temporary storage in a processor
 - Faster and more convenient than main RAM memory
 - More registers can help speed up complex calculations

A Basic Register

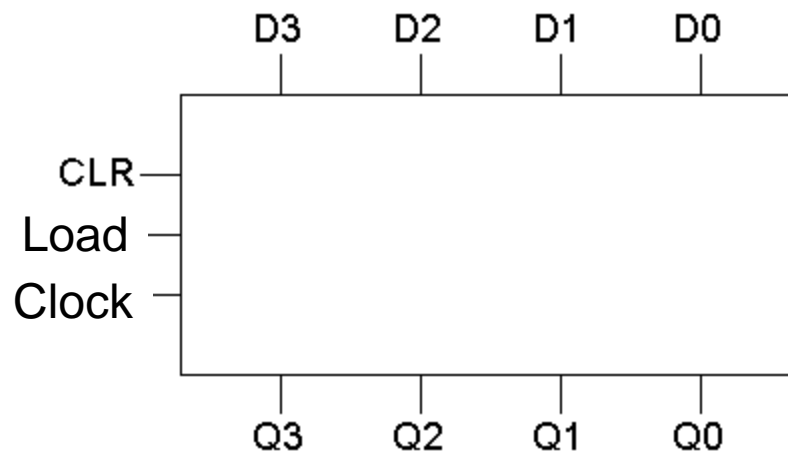
- We store multiple bits by putting a bunch of flip-flops together!
 - D flip-flops are used; store data without worrying about flip-flop input equations
 - All flip-flops share a common **CLK** and **CLR** signal
 - **CLK** input triggers all flip-flops on the rising edge and the data available at the four D inputs is stored in the register



Register with Parallel Load

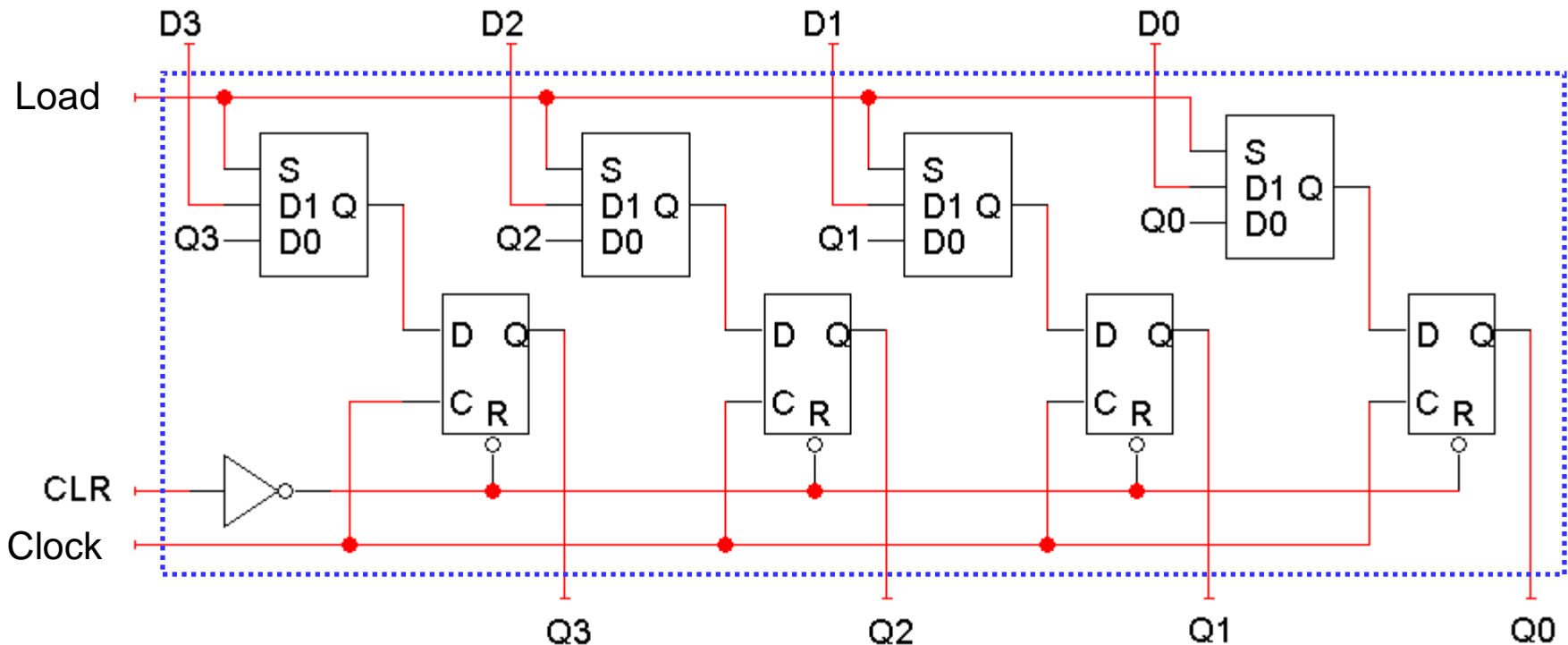
- The input D_3-D_0 is copied to the output Q_3-Q_0 on *every* clock cycle.
- How can we store the current value for more than one cycle?
- Let us add a load input signal **Load** to the register.
 - If **Load** = 0, the register keeps its current contents.
 - If **Load** = 1, the register stores a new value, taken from inputs D_3-D_0 .

LD	$Q(t+1)$
0	$Q(t)$
1	D_3-D_0



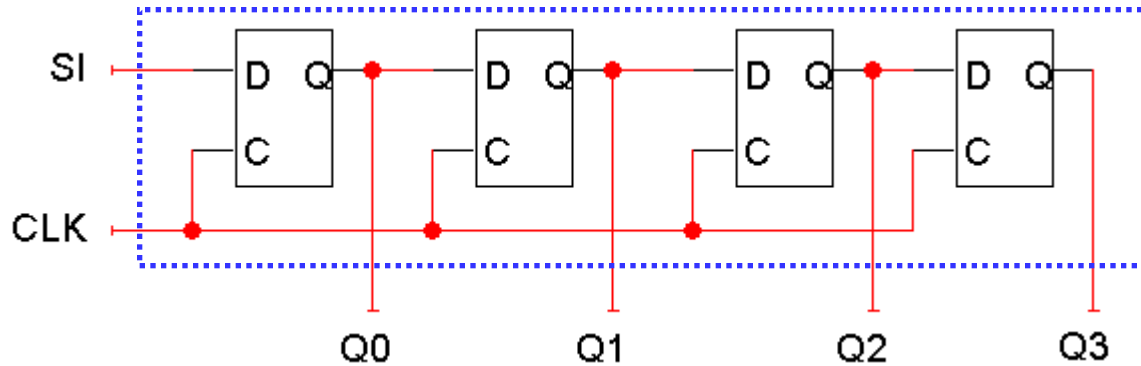
Parallel Load

- The idea is to modify the flip-flop D inputs. We add 2-to-1 MUX to each input D.
 - When **Load** = 0, the flip-flop inputs are Q_3 - Q_0 , so each flip-flop just keeps its current value.
 - When **Load** = 1, the flip-flop inputs are D_3 - D_0 , and this new value is “loaded” into the register.



Shift Register

- A **shift register** “shifts” its output once every clock cycle.



$$\begin{aligned} Q_0(t+1) &= SI \\ Q_1(t+1) &= Q_0(t) \\ Q_2(t+1) &= Q_1(t) \\ Q_3(t+1) &= Q_2(t) \end{aligned}$$

- **SI** is an input that supplies a new bit to shift “into” the register.
- For example, if on some positive clock edge we have:

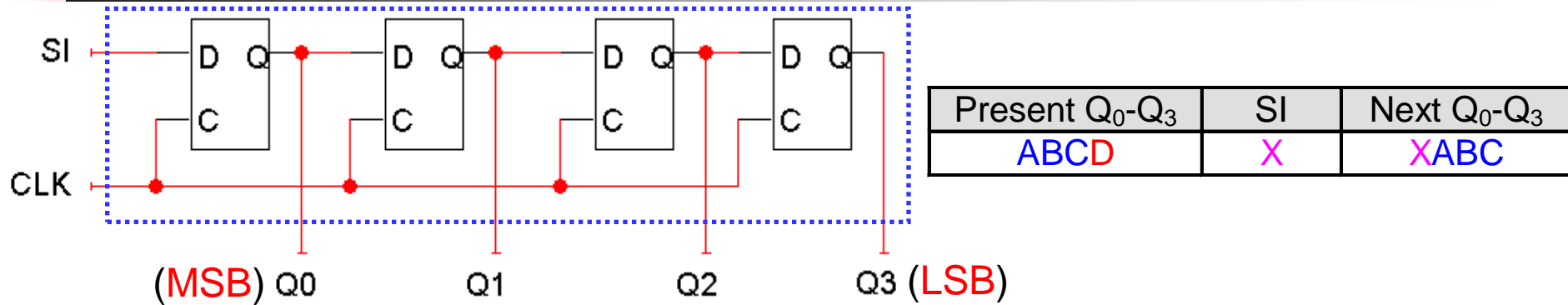
$$\begin{aligned} SI &= 1 \\ Q_0-Q_3 &= 0110 \end{aligned}$$

then the next state will be:

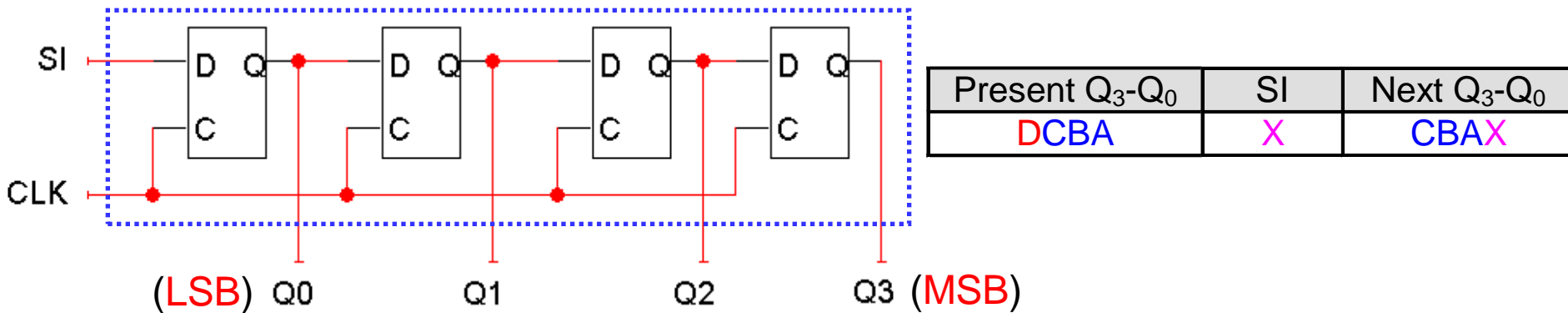
$$Q_0-Q_3 = 1011$$

- The current Q_3 (0 in this example) will be lost on the next cycle.

Shift Direction



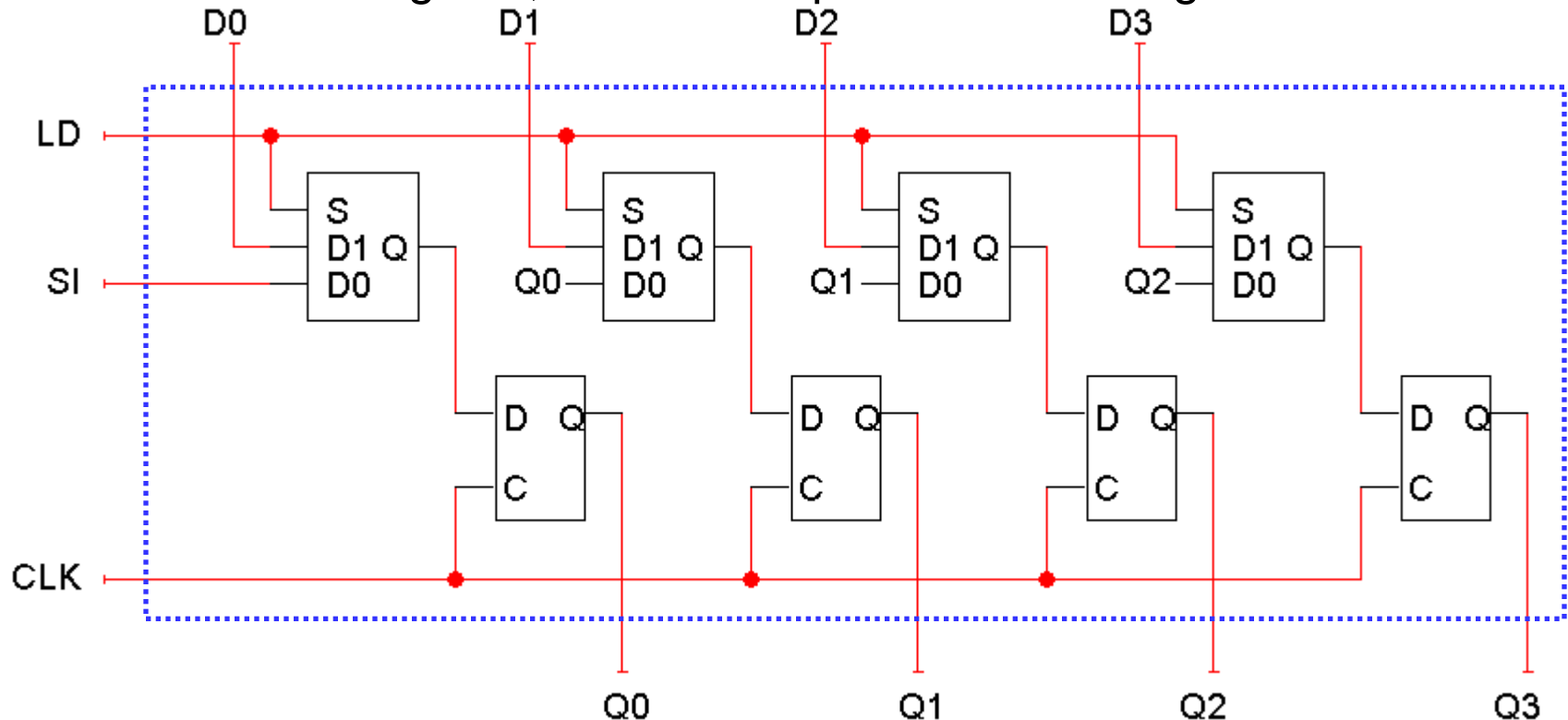
- The circuit and example **above** make it look like the register shifts **“right”**.



- But it really depends on your interpretation of the bits. If you consider Q_3 to be the most significant bit instead, then the register is shifting in the **opposite** direction (**“left”**)!

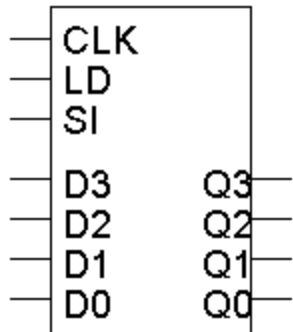
Shift Register with Parallel Load

- We can add a parallel load, just like we did for regular registers.
 - When $LD = 0$, the flip-flop inputs will be $SIQ_0Q_1Q_2$, so the register shifts on the next positive clock edge.
 - When $LD = 1$, the flip-flop inputs are D_0-D_3 , and a new value is loaded into the shift register, on the next positive clock edge.



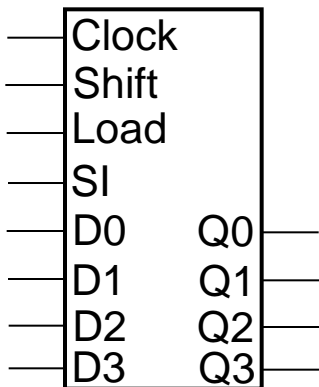
Shift Register with Parallel Load (cont.)

- Block symbol of the 4-bit shift register with load signal (LD)
 - The **Load** and **Shift** operations are controlled by one signal **LD**



LD	Operation
0	Shift $Q_0 \rightarrow Q_1, Q_1 \rightarrow Q_2 \dots$
1	Load parallel

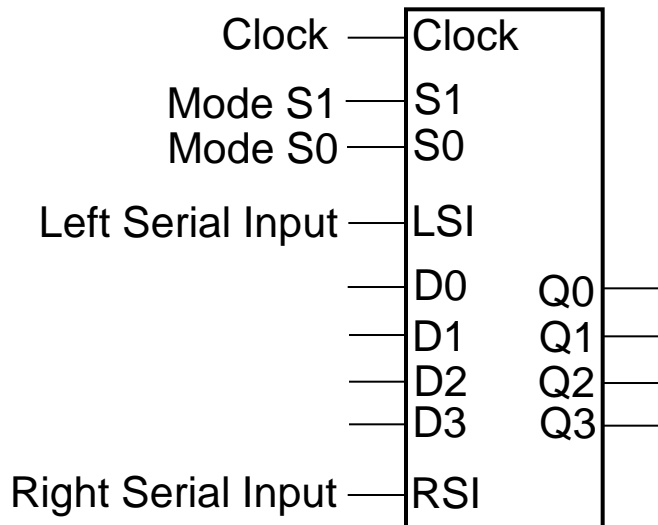
- In many cases it is useful to have separate signals that control the **Load** and **Shift** operations



Shift	Load	Operation
0	0	Nothing
0	1	Parallel Load
1	X	Shift $Q_0 \rightarrow Q_1, Q_1 \rightarrow Q_2 \dots$

Bidirectional Shift Register

- So far we have seen registers capable of shifting in only one direction, i.e., *unidirectional shift registers*.
- Registers that can shift in both directions are called *bidirectional shift registers*.



$S_1 S_0$	Operation
00	Nothing -- $Q_i(t+1) = Q_i(t)$
01	Shift down -- $Q_i \rightarrow Q_{i+1}$
10	Shift up -- $Q_i \rightarrow Q_{i-1}$
11	Parallel Load -- $Q_i = D_i$

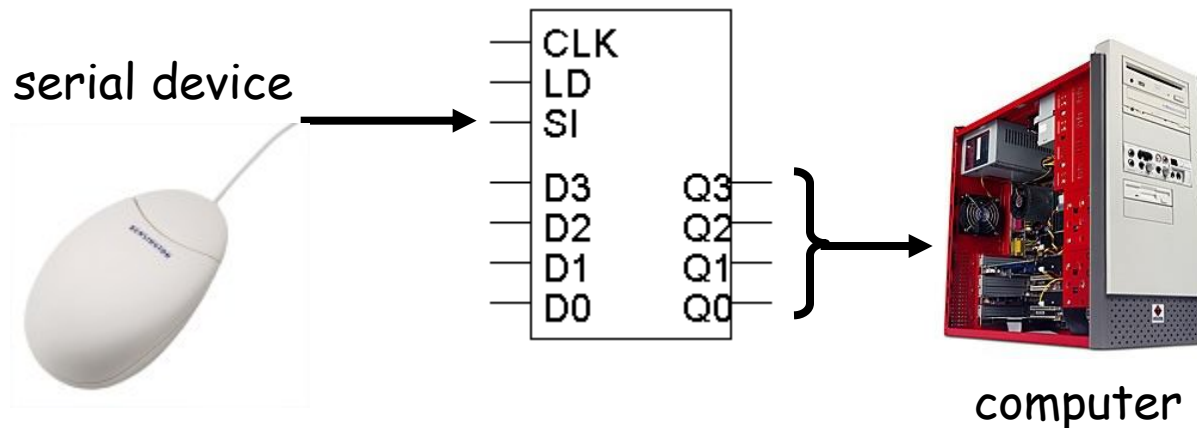
Serial Data Transfer

- One application of shift registers is converting between “serial data” and “parallel data.”
- Computers typically work with multiple-bit quantities.
 - ASCII text characters are 8 bits long.
 - Integers, single-precision floating-point numbers, and screen pixels are up to 64 bits long.
- But sometimes it is necessary to send or receive data **serially**, or one bit at a time. Why?
- Some examples include:
 - Input devices such as keyboards and mice.
 - Output devices like printers.
 - Any serial port, USB or Firewire device transfers data serially.
 - Serial ATA in hard drives.



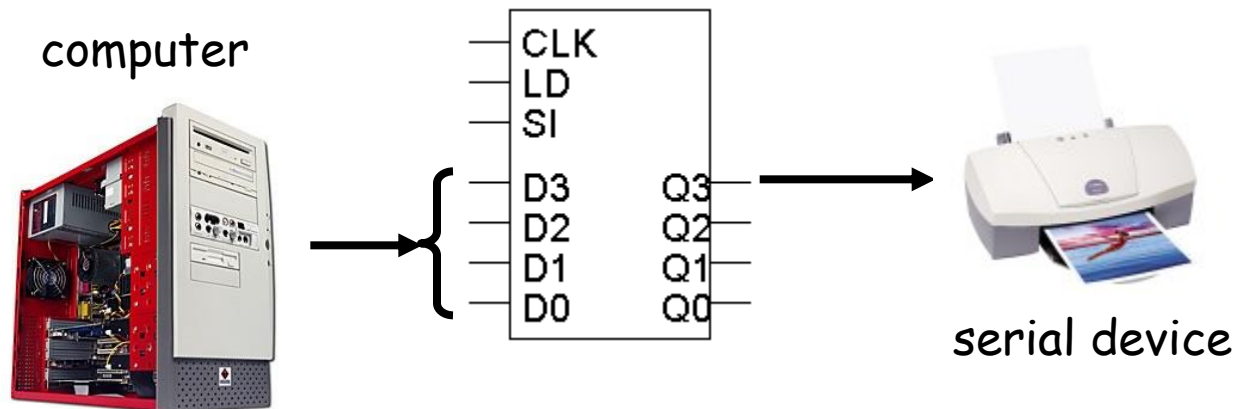
Receiving Serial Data

- To *receive* serial data using a shift register:
 - The serial device is connected to the register's SI input.
 - The shift register outputs Q3-Q0 are connected to the CPU.
- The serial device transmits one bit of data per clock cycle.
 - These bits go into the SI input of the shift register.
 - After four clock cycles, the shift register will hold a four-bit word.
- The computer then reads all four bits at once from the Q3-Q0 outputs.



Sending Data Serially

- To *send* data serially with a shift register, you do the opposite:
 - The CPU is connected to the register's D inputs.
 - The shift output (Q3 in this case) is connected to the serial device.
- The computer first stores a four-bit word in the register, in one cycle.
- The serial device can then read the shift output.
 - One bit appears on Q3 on each clock cycle.
 - After four cycles, the entire four-bit word will have been sent.

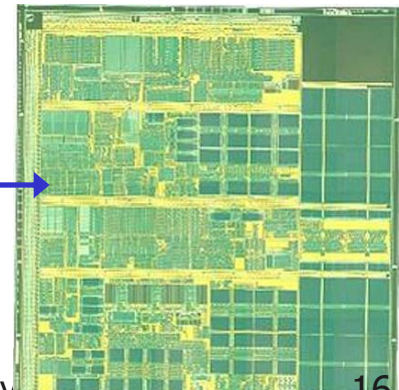


Registers in Modern Processors

- Registers store data in the CPU
 - Used to supply values to the ALU
 - Used to store the results

CPU	GPR's	Size	L1 Cache	L2 Cache
Pentium i7	16	64 bits	130 KB	1024KB
ARM Cortex-A53	16	64 bits	64 KB	2048 KB
Athlon 64	16	64 bits	64 KB	1024 KB
PowerPC 970 (G5)	32	64 bits	64 KB	512 KB
Itanium 2	128	64 bits	16 KB	256 KB
MIPS R14000	32	64 bits	32 KB	16 MB

- If we can use registers, why bother with RAM?
- Answer: Registers are expensive!
 - Registers occupy the most expensive space on a chip – the core.
 - L1 and L2 are very fast RAM – but not as fast as registers.



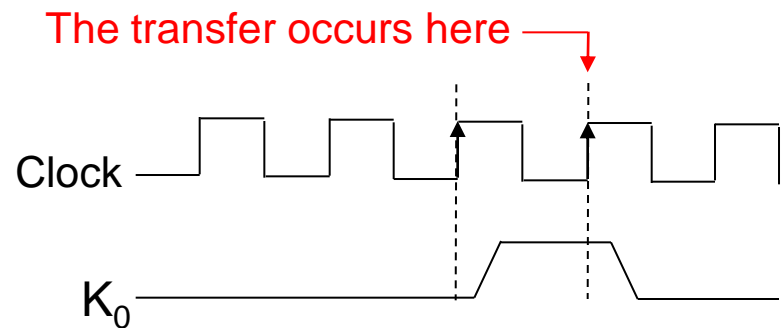
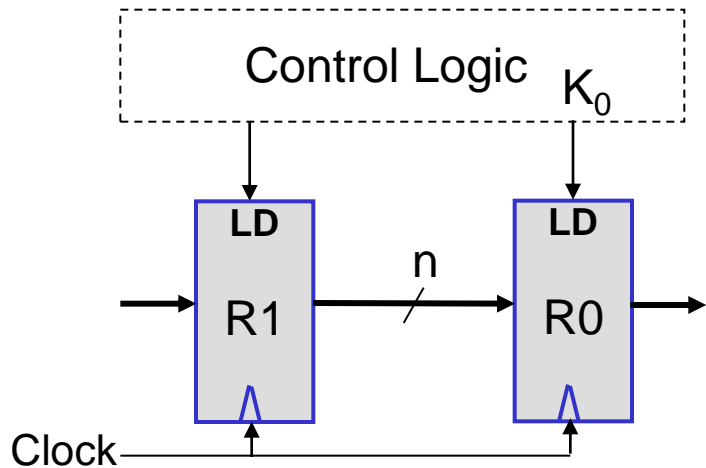


Registers and Microoperations

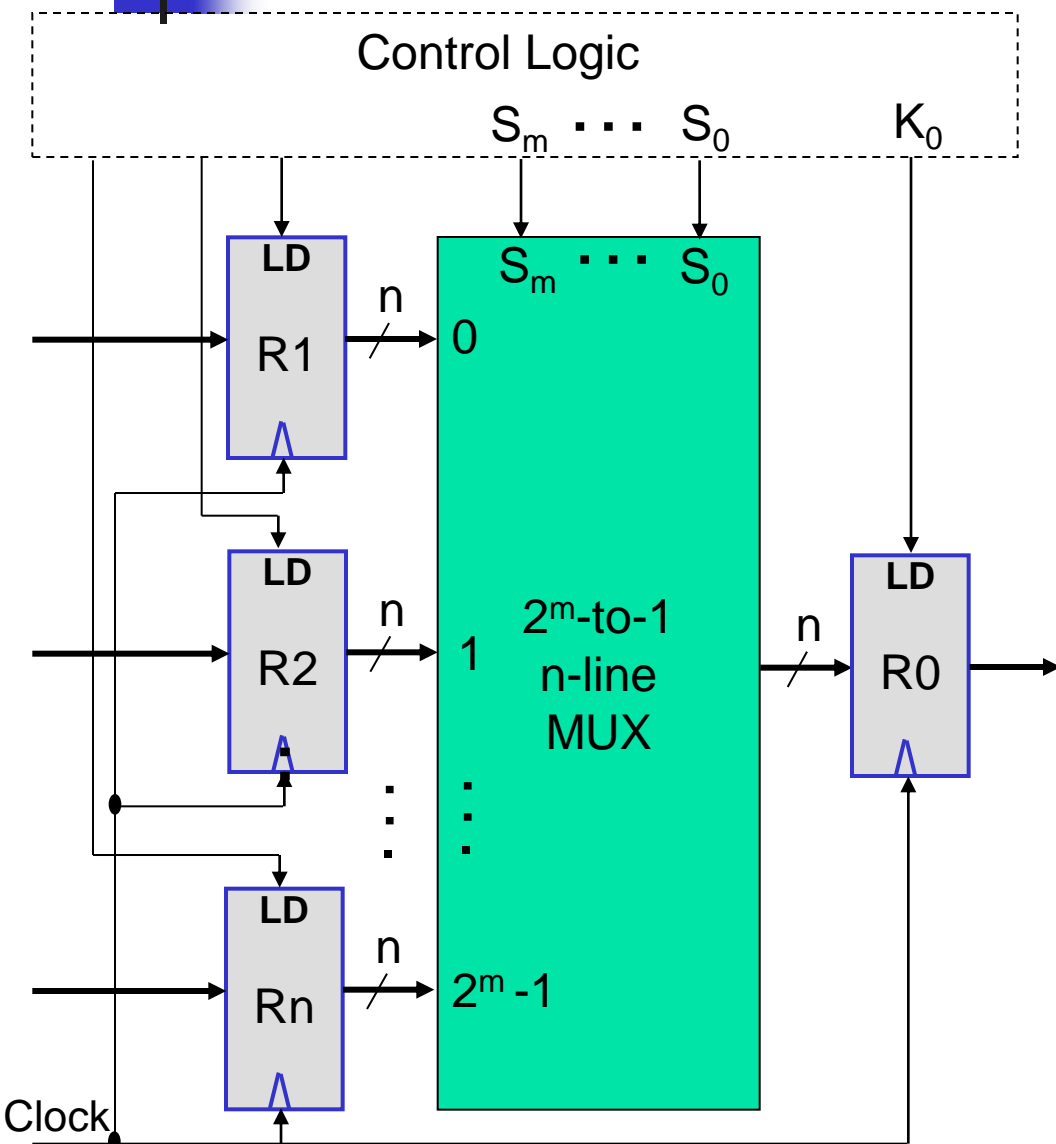
- In modern digital systems registers and combinational logic circuits are used to implement *Microoperations*.
- A microoperation is an elementary operation performed on data stored in registers or in memory.
- The microoperations most often encountered in digital systems are of four types:
 - **Transfer** microoperations, which only move binary data from one register to another.
 - **Arithmetic** microoperations, which perform arithmetic on data in registers.
 - **Logic** microoperations, which perform bit manipulation on data in registers.
 - **Shift** microoperations, which shift data in registers.

Basic *Transfer* Microoperation

- We will use the notation $K_0: R0 \leftarrow R1$
 - It reads:
“if $K_0 = 1$ then the content of $R1$ is transferred into $R0$ ”
 - $R1$ is called **source** register
 - $R0$ is called **destination** register
 - K_0 is a control signal (**condition**) generated by the control logic. K_0 can be any Boolean function.
- Implementation

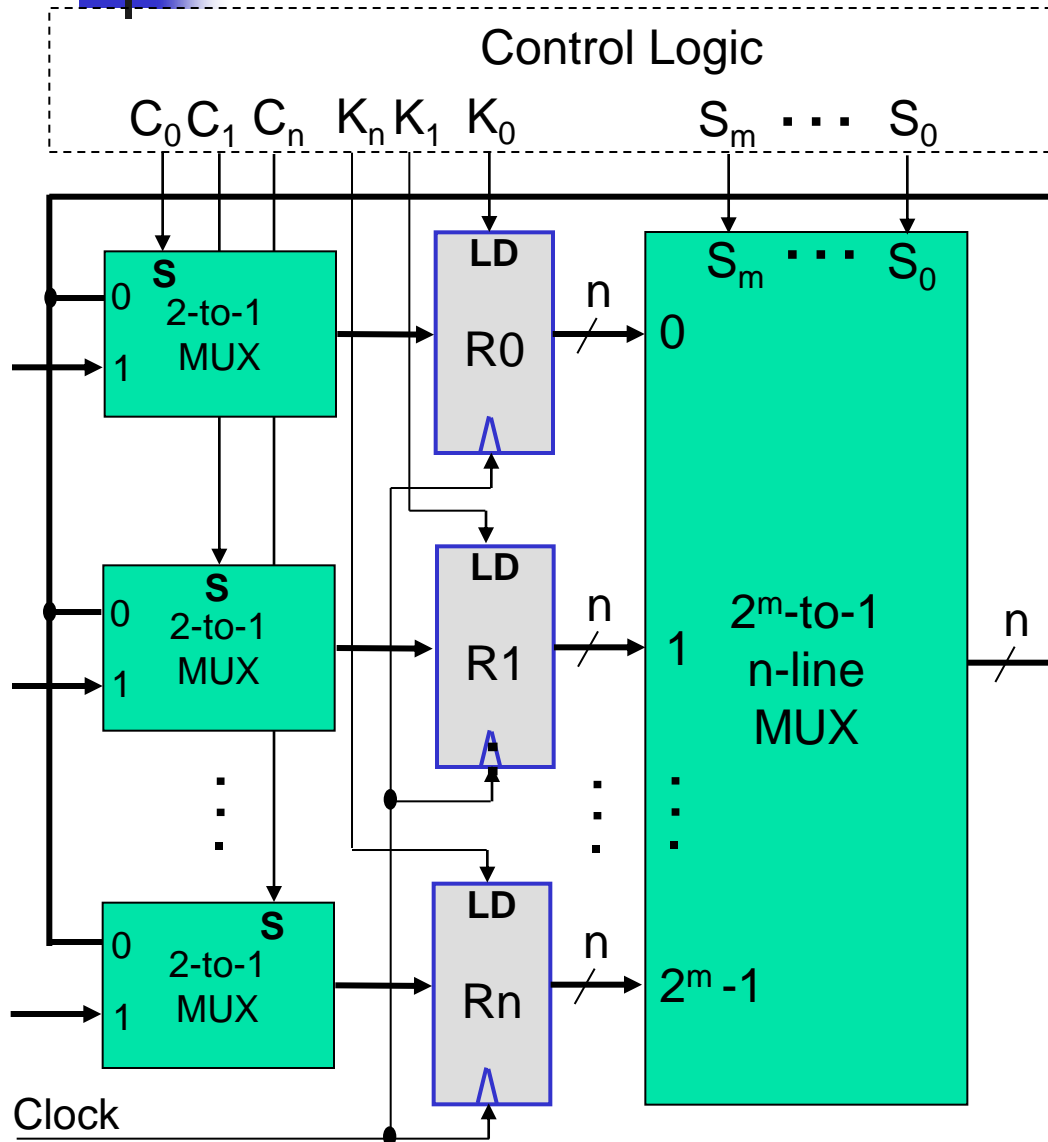


Single Register Multiplexer-Based *Transfer* Microoperations



- A single register can receive data from different sources at different times:
- $K_0 \bar{S}_m \dots \bar{S}_0: R_0 \leftarrow R_1$
- $K_0 \bar{S}_m \dots S_0: R_0 \leftarrow R_2$
- ...
- $K_0 S_m \dots S_0: R_0 \leftarrow R_n$
- How do you read the above notations?

Multiple Registers Multiplexer-Based *Transfer* Microoperations

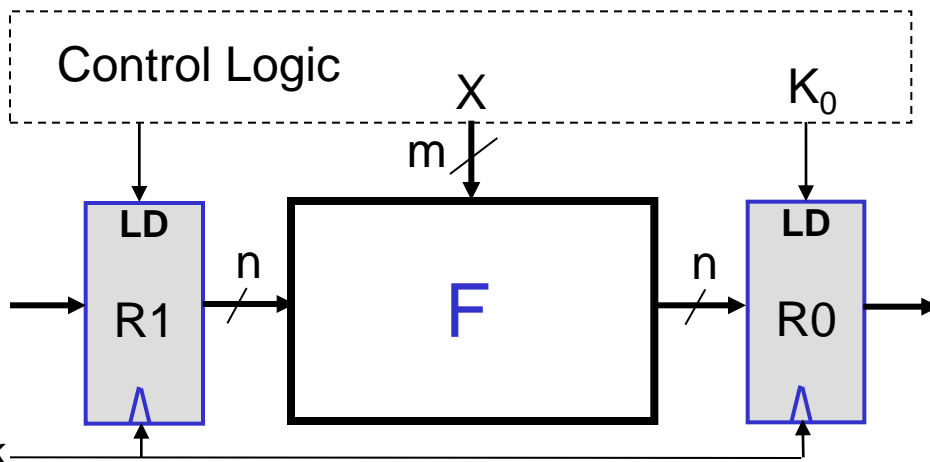


- Any register can receive data from different sources at different times.
 - $R_i \leftarrow R_j$
- Multiple registers can receive data from one source at the same time.
 - $R_0 \leftarrow R_j, \dots, R_n \leftarrow R_j$
- How?
 - By assigning proper values to signals C_0 to C_n, K_0 to K_n , and S_0 to S_m

Basic Single Operand Arithmetic/Logic/Shift Microoperations

- We will use the notation $K_0: R0 \leftarrow F(R1)$
 - It reads:
 - “if $K_0 = 1$ then register $R0$ stores the result of operation F applied on the content of register $R1$ ”
 - $R1$ is called **source** register, where the **single operand** is stored.
 - $R0$ is called **destination** register, where the result is stored.
 - K_0 is a control signal (**condition**) generated by the control logic. K_0 can be any Boolean function.
 - F is an arithmetic or logic or shift operation - see examples in the table below.

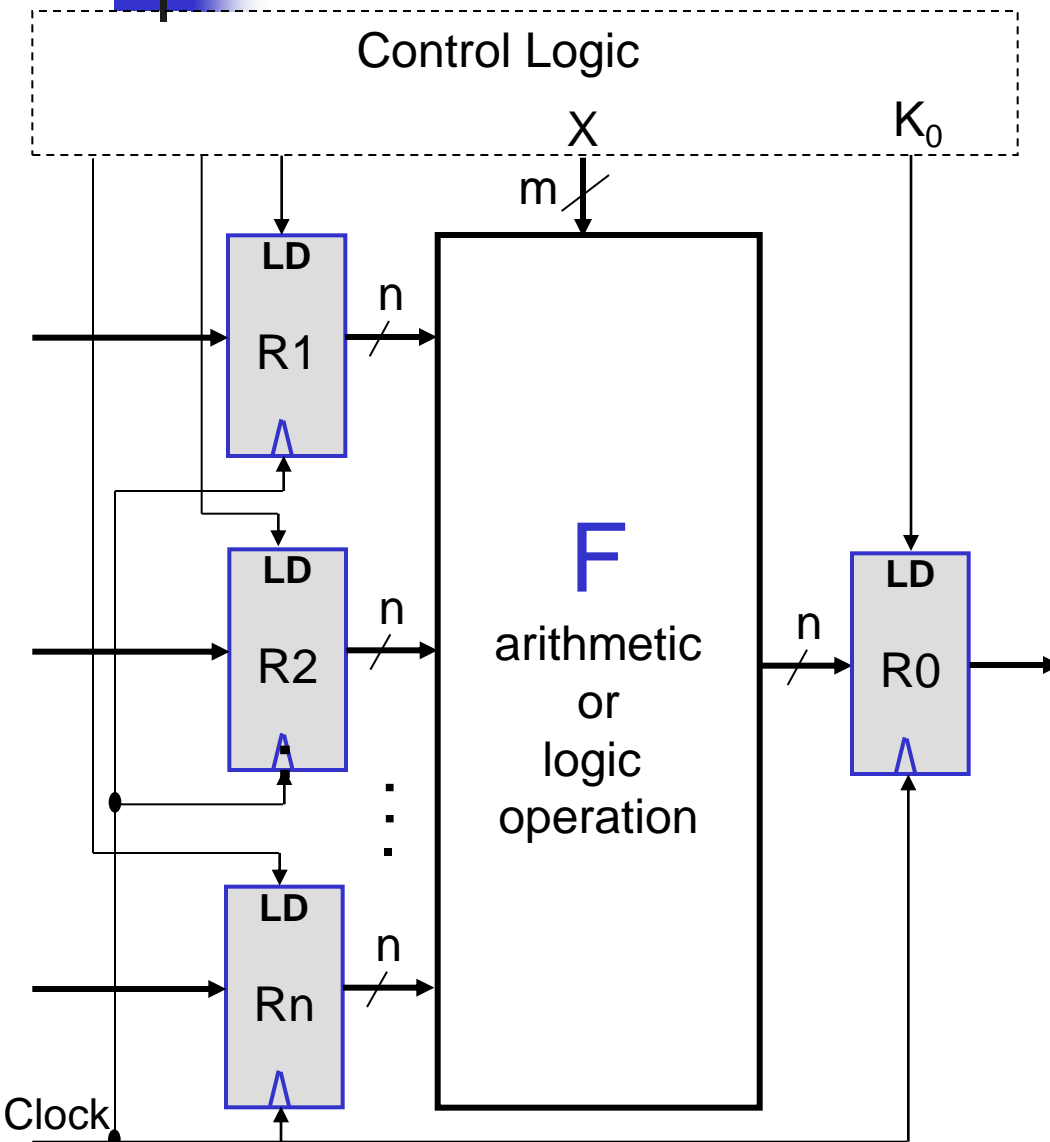
Implementation



Operation	Description
$R0 \leftarrow (R1)'$	1's complement of $R1$
$R0 \leftarrow (R1)' + 1$	2's complement of $R1$
$R0 \leftarrow R1 + 1$	Increment $R1$
$R0 \leftarrow R1 - 1$	Decrement $R1$
$R0 \leftarrow shl(R1)$	Shift-Left $R1$
$R0 \leftarrow shr(R1)$	Shift-Right $R1$

Note: In most cases F is a combinational logic circuit that implements one or several operations. X selects the operation.

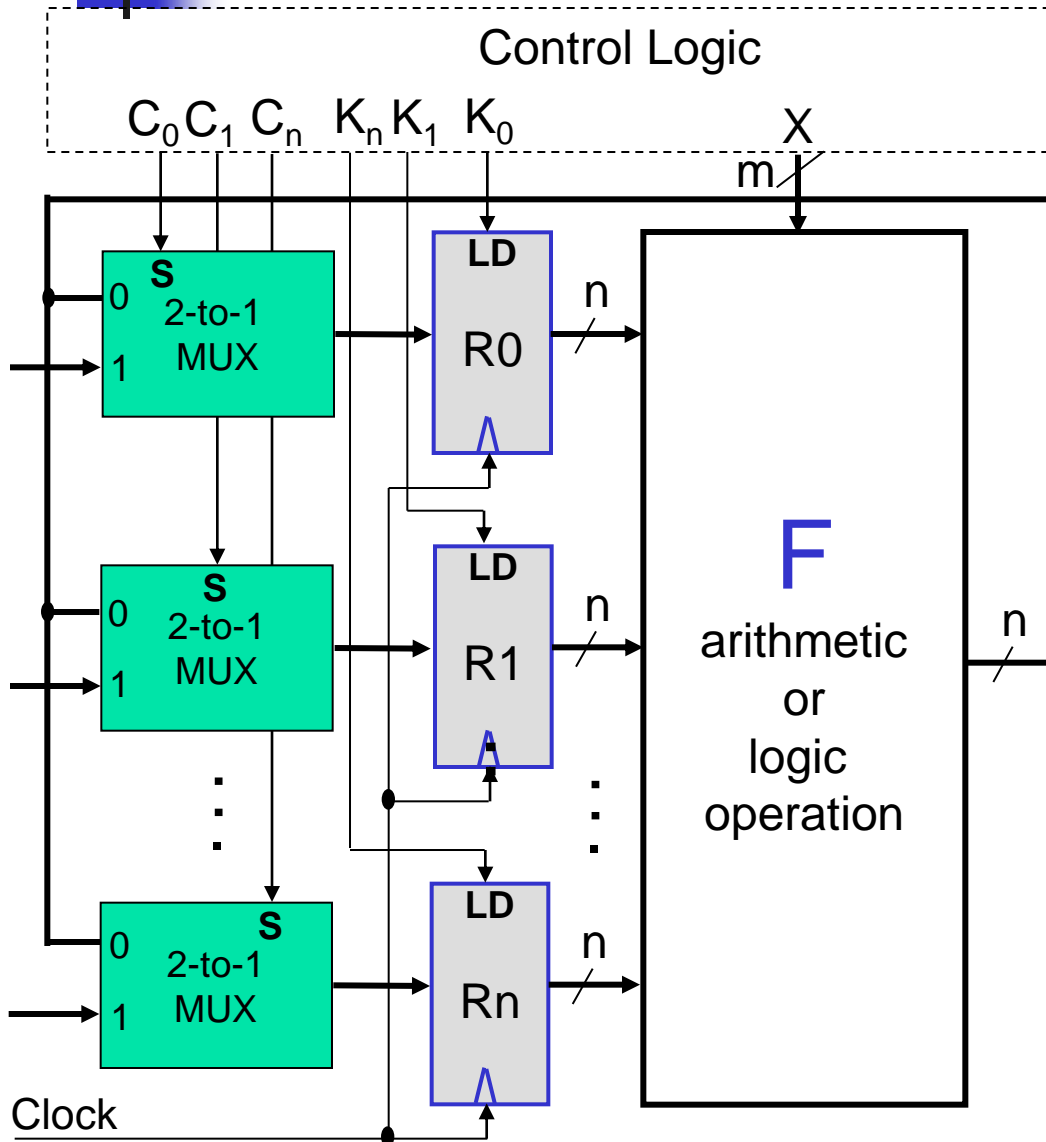
Basic Multiple Operand *Arithmetic/Logic* Microoperations



- $K_0: R_0 \leftarrow F(R_1, \dots, R_n)$
- Single destination
- Multiple operands
 - In most cases only two
- Example of basic two operand microoperations:

Operation	Description
$R_0 \leftarrow R_1 + R_2$	Add R1 and R2
$R_0 \leftarrow R_1 - R_2$	Subtract R1 from R2
$R_0 \leftarrow R_1 \& R_2$	Logic bitwise AND
$R_0 \leftarrow R_1 R_2$	Logic bitwise OR
$R_0 \leftarrow R_1 * R_2$	Multiply R1 by R2
$R_0 \leftarrow R_1 / R_2$	Divide R1 by R2

Multiple Operand Arithmetic/Logic Microoperations



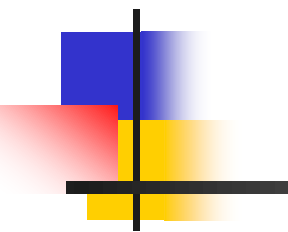
- $R0 \leftarrow F(R0, \dots, Rn)$
- ...
- $Rn \leftarrow F(R0, \dots, Rn)$
- Multiple destinations
- Multiple operands
 - In most cases only two
- Example of basic two operand microoperations:

Operation	Description
$R0 \leftarrow R0 + R1$	Add R0 and R1
$R0 \leftarrow R0 - R1$	Subtract R1 from R0
$R2 \leftarrow R0 \& R1$	Logic bitwise AND
$R2 \leftarrow R0 R1$	Logic bitwise OR
$R1 \leftarrow R0 * R1$	Multiply R0 by R1
$R1 \leftarrow R0 / R1$	Divide R0 by R1



Registers Summary

- A register is a special ***sequential circuit*** (state machine) that stores multiple bits of data.
- Several variations are possible:
 - Parallel loading to store data into the register.
 - Shifting the register contents either left or right.
- One application of shift registers is converting between serial and parallel data.
- Registers are a central part of modern processors.
 - Used to implement Microoperations.
 - You will see more during your design project.
- Most programs need more storage space than registers provide.
 - We will introduce RAM to address this problem.



Special Sequential Circuits: Counters



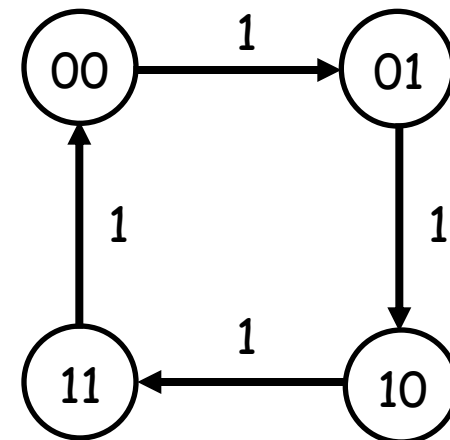
Overview

- Counters
 - Basic Idea
 - Binary Upward and Downward Counters
- Asynchronous (Ripple) Counters
 - Binary Upward Counter implemented with T flip-flops
 - Binary Downward Counter implemented with T, JK, and D flip-flops
 - Pros and Cons
- Synchronous Binary Counters
 - Parallel Binary Counters
 - Up-Down Binary Counter
 - Other Counters (BCD Counter, Arbitrary Sequence Counters, ...)
- Applications of Counters
- Summary

Introducing Counters: Basic Idea

- Counters are a specific type of sequential circuit.
- Like registers, the state, or the flip-flop values themselves, serves as the “output.”
- The output value increases by one on each clock cycle.
- After the largest value, the output “wraps around” back to 0.
- Using two bits, we would get something like this:

Present State		Next State	
A	B	A	B
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0





What are Counters good for?

- Counters can act as **simple clocks** to keep track of “time.”
- You may need to record how many times something has happened.
 - How many bits have been sent or received?
 - How many steps have been performed in some computation?
- Counters can be used as **clock frequency dividers**:
 - Given a clock signal with frequency f_{clk}
 - Given an n-bit counter
 - We can generate clock signals with frequencies $f_{\text{clk}}/2, f_{\text{clk}}/4, \dots, f_{\text{clk}}/2^n$
- All processors contain a **program counter**, or **PC**.
 - Programs consist of a list of instructions that are to be executed one after another (for the most part).
 - The PC keeps track of the instruction currently being executed.
 - The PC increments once on each clock cycle, and the next program instruction is then executed.

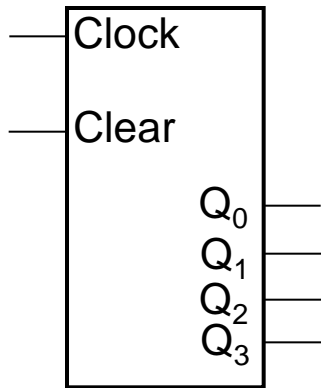


Counters

- A *counter* is a register that goes through a predetermined sequence of distinct states upon the application of a sequence of input pulses.
- The input pulses may be:
 - Clock pulses.
 - Pulses originating from other sources.
 - The pulses may occur at regular or irregular intervals of time.
- In our discussion of counters we assume clock pulses.
- The sequence of states may follow:
 - The binary number sequence -- *binary counter*
 - Any other predetermined sequence of states
- An n -bit *binary counter* consists of n flip-flops and can count from 0 through $2^n - 1$

A Basic Binary Upward Counter

Symbol



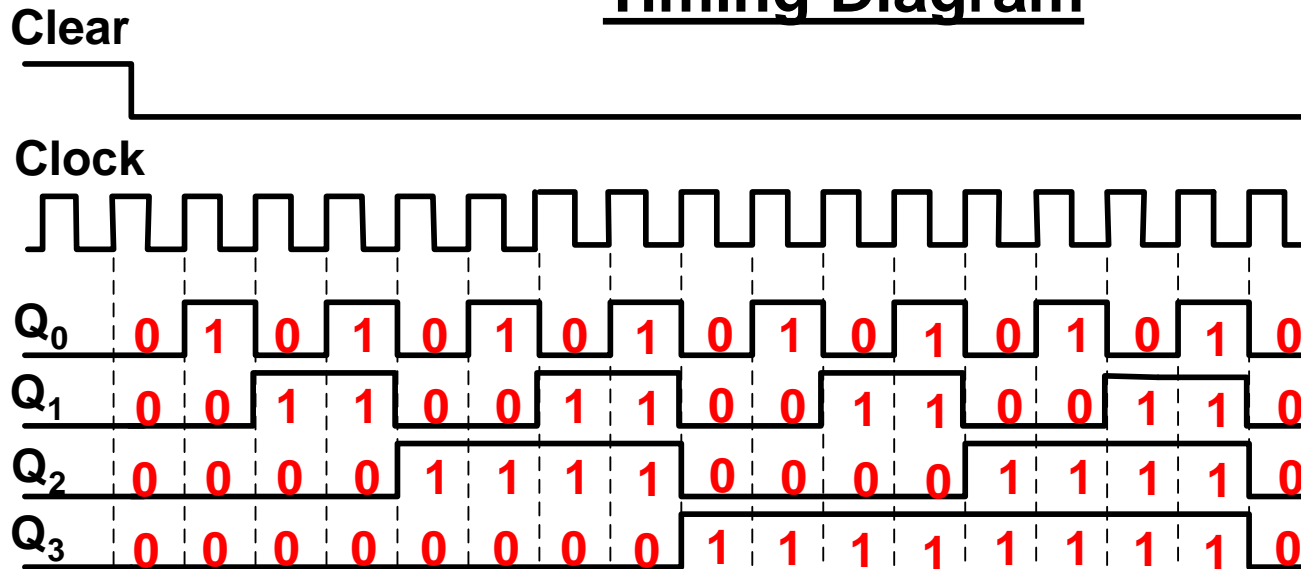
Function Table

Clear	Operation
0	Count Up
1	$Q_i = 0$

Upward Counting Sequence

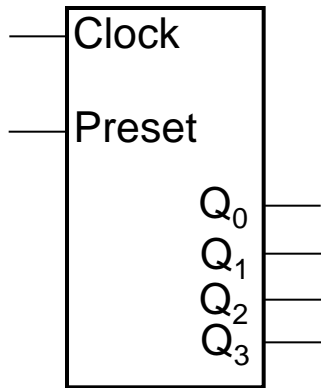
Q_3	Q_2	Q_1	Q_0
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

Timing Diagram



A Basic Binary Downward Counter

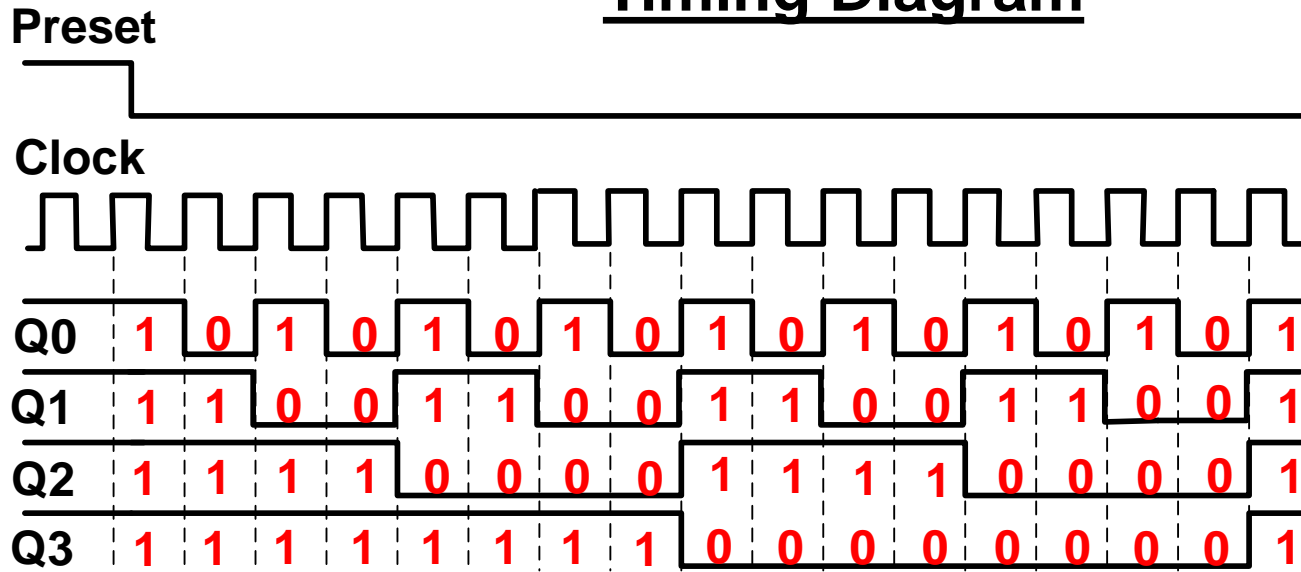
Symbol



Function Table

Preset	Operation
0	Count Down
1	$Q_i = 1$

Timing Diagram



Downward Counting Sequence

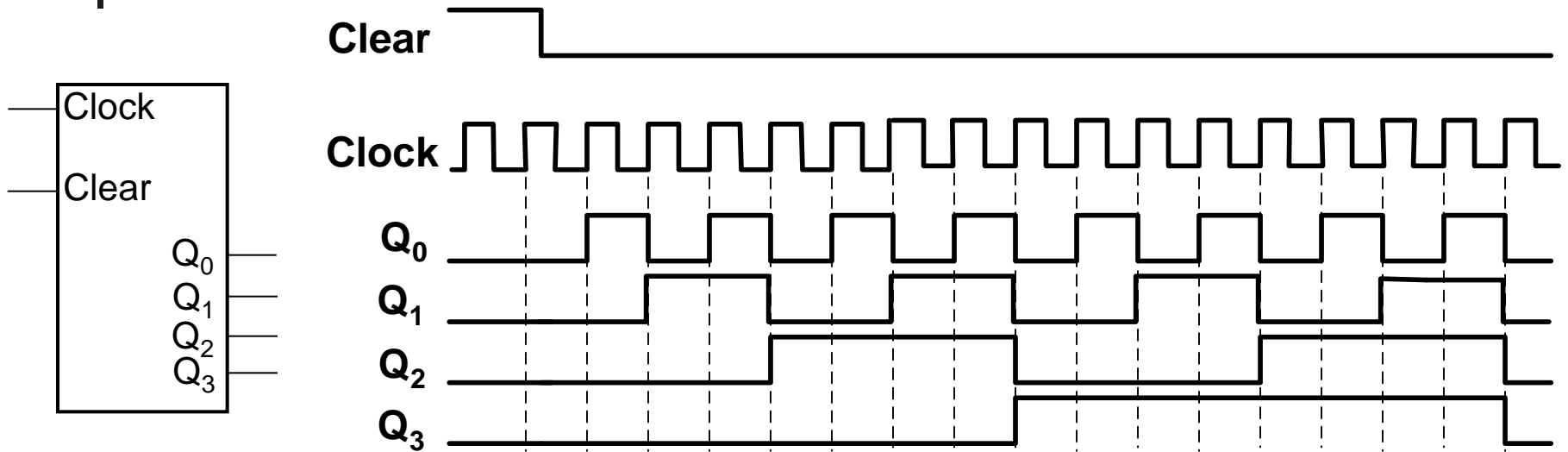
Q ₃	Q ₂	Q ₁	Q ₀
1	1	1	1
1	1	1	0
1	1	0	1
1	1	0	0
1	0	1	1
1	0	1	0
1	0	0	1
1	0	0	0
0	1	1	1
0	1	1	0
0	1	0	1
0	1	0	0
0	0	1	1
0	0	1	0
0	0	0	1
0	0	0	0



Types of Counters

- **Asynchronous (Ripple) Counters:**
The flip-flop output transition serves as a source for triggering other flip-flops. **No common clock.**
- **Synchronous Counter:**
All flip-flops receive the **common clock pulse**, and the change of state is determined from the present state.
 - Serial and Parallel Binary Counters
 - Binary Counter with Parallel Load
 - Up-Down Binary Counter
 - Other Counters
 - BCD Counter
 - Arbitrary sequence Counters

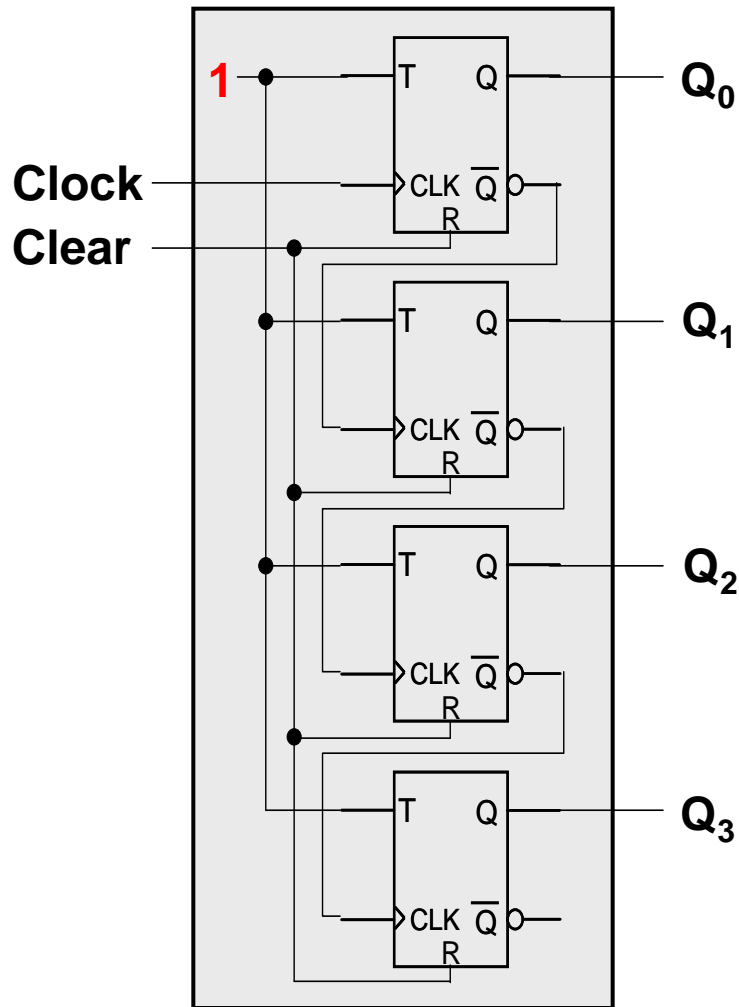
A 4-bit Binary *Upward Ripple* Counter



■ Observation:

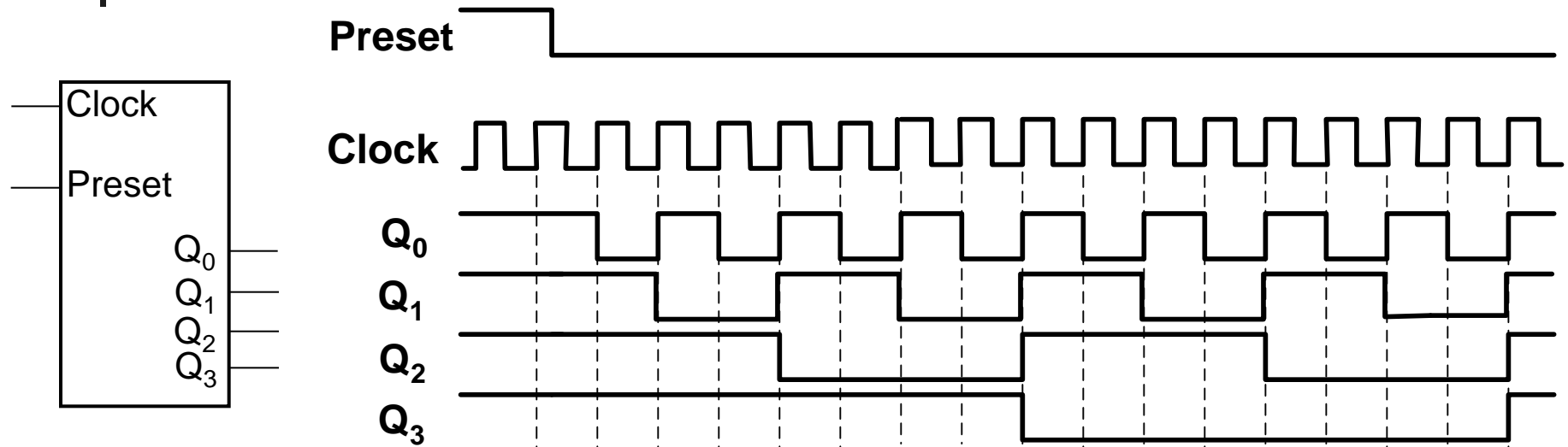
- The least significant bit (Q_0) is complemented with each rising-edge of the clock pulse input.
- Every time that Q_0 goes from 1 to 0, Q_1 is complemented.
- Every time that Q_1 goes from 1 to 0, Q_2 is complemented.
- Every time that Q_2 goes from 1 to 0, Q_3 is complemented.

A 4-bit Binary Upward Ripple Counter: Implementation using T Flip-Flops



- The output (Q') of each flip-flop is connected to the CLK input of the next flip-flop in the sequence. Why?
- The flip-flop holding the least significant bit Q_0 receives the incoming clock pulses. Why?
- The T inputs of all flip-flops are connected to a permanent logic 1. Why?
- All flip-flops respond to the 0-to-1 transition of the input CLK (rising-edge).

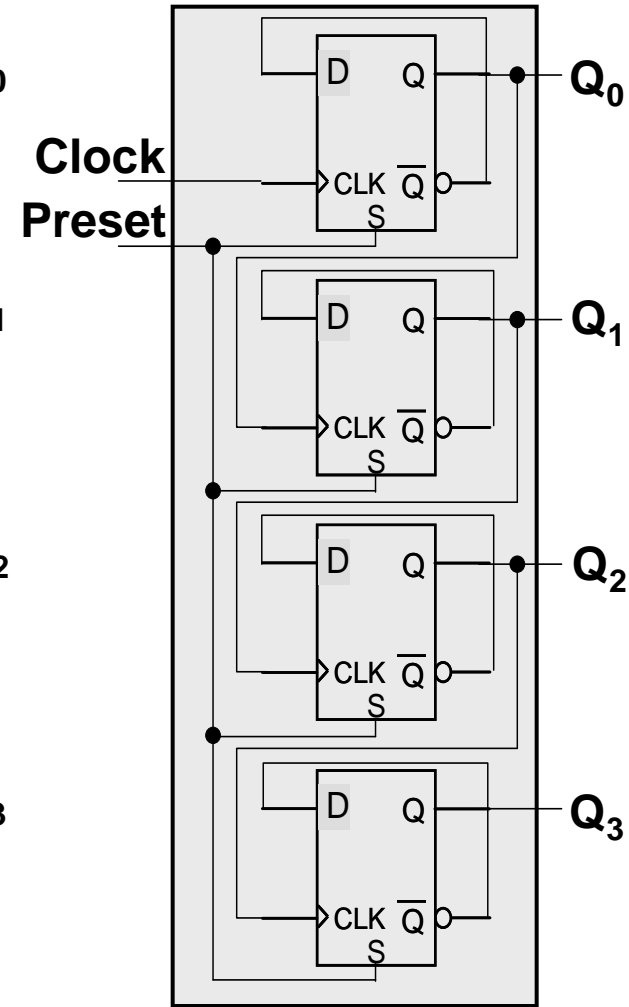
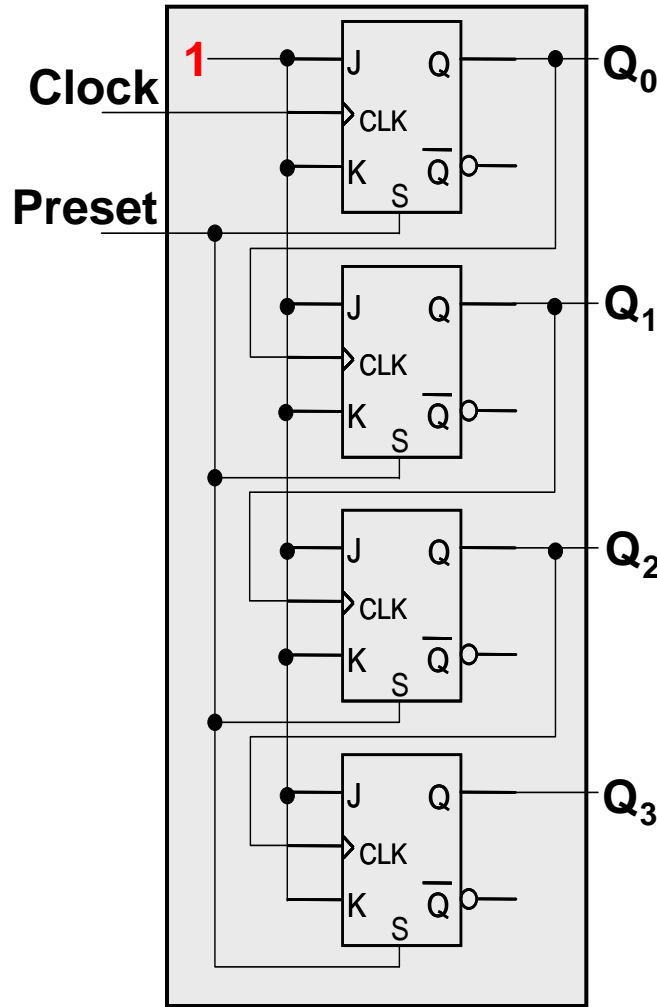
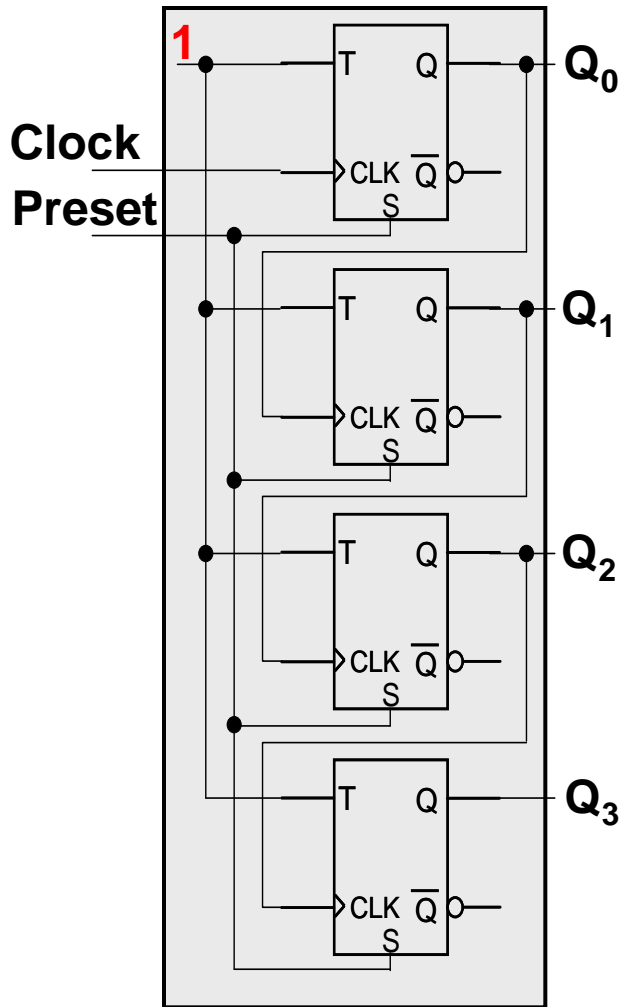
A 4-bit Binary *Downward Ripple* Counter



■ Observation:

- The least significant bit (Q_0) is complemented with each rising-edge of the clock pulse input.
- Every time that Q_0 goes from 0 to 1, Q_1 is complemented.
- Every time that Q_1 goes from 0 to 1, Q_2 is complemented.
- Every time that Q_2 goes from 0 to 1, Q_3 is complemented.

A 4-bit Binary Downward Ripple Counter: Implementation with *T*, *JK*, and *D* Flip-Flops



T Flip-Flops with *JK*!

T Flip-Flops with *D*!

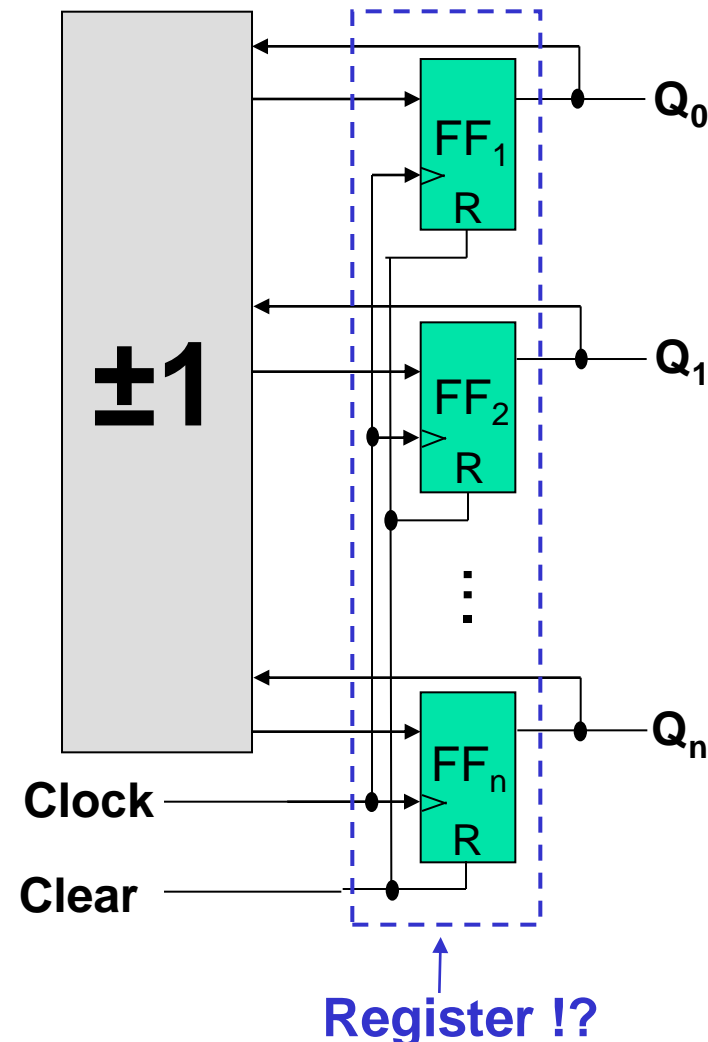


Binary Ripple Counters: Pros and Cons

- Recall that Binary Ripple Counters are **asynchronous** sequential circuits because the flip-flops are not connected to a common clock signal.
- Advantages:
 - Ripple Counters have **simple hardware** structure.
 - Ripple Counters are suitable for **low-power design**.
- Disadvantages:
 - Ripple Counters are asynchronous circuits and, with added logic, can become circuits with **delay dependent and unreliable behavior**.
 - Large Ripple Counters can be **slow circuits** due to the length of time required for the ripple to finish.
- Because of the above disadvantages, **synchronous binary counters are favored nowadays** in most of the designs.

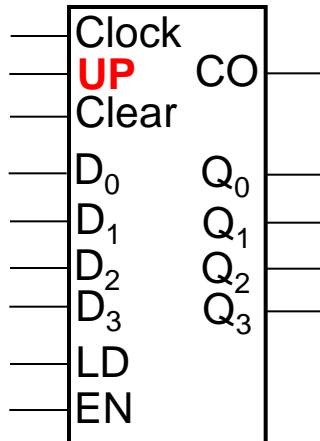
Synchronous Binary Counters

- Synchronous Counter is a register with increment /decrement circuit
- Synchronous Counters, in contrast to ripple counters, have
 - **common clock signal** applied to all flip-flops
 - clock pulse **triggers all flip-flops simultaneously** rather than one at a time, as in a ripple counter
 - **polarity of the clock is not essential here**, i.e., either rising or falling edge of the clock signal can be used
- The procedure in Lecture 10 can be used to design synchronous binary counters.
 - **You know how to do it – see Tutorial 3**



A 4-bit Binary *Up-Down* Counter

Symbol



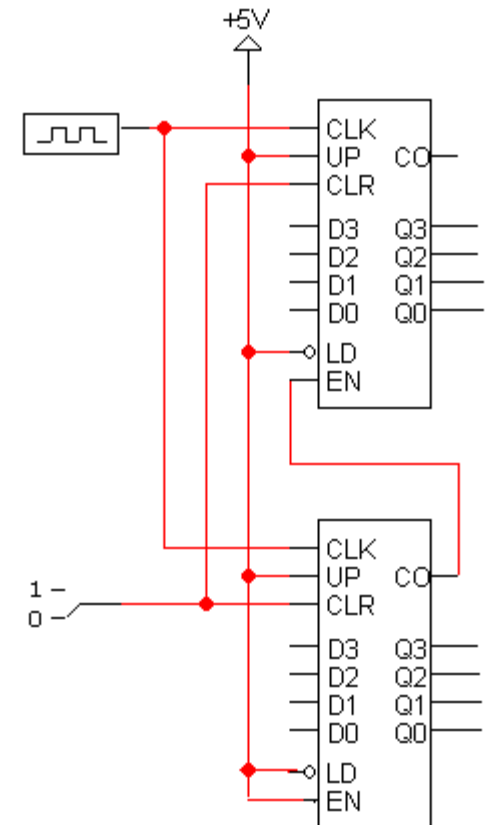
Function Table

Clear	LD	EN	UP	Operation
0	0	0	x	No Change
0	0	1	0	Count DOWN
0	0	1	1	Count UP
0	1	x	x	Load
1	x	x	x	$Q_i = 0$

- This is a full-featured counter.
 - You can immediately (asynchronously) clear the counter to 0000 by setting **Clear = 1**.
 - You can load the counter by setting D₃-D₀ to any four-bit value and **LD = 1**.
 - The active-high EN input enables or disables the counter.
 - When the counter is disabled (**EN = 0**), it continues to output the same value.
 - When the counter is enabled (**EN = 1**), it can increment or decrement, by setting the **UP** input to **1** or **0**.
 - The CO output is normally 0, but becomes 1 when the counter reaches its maximum value, 1111.

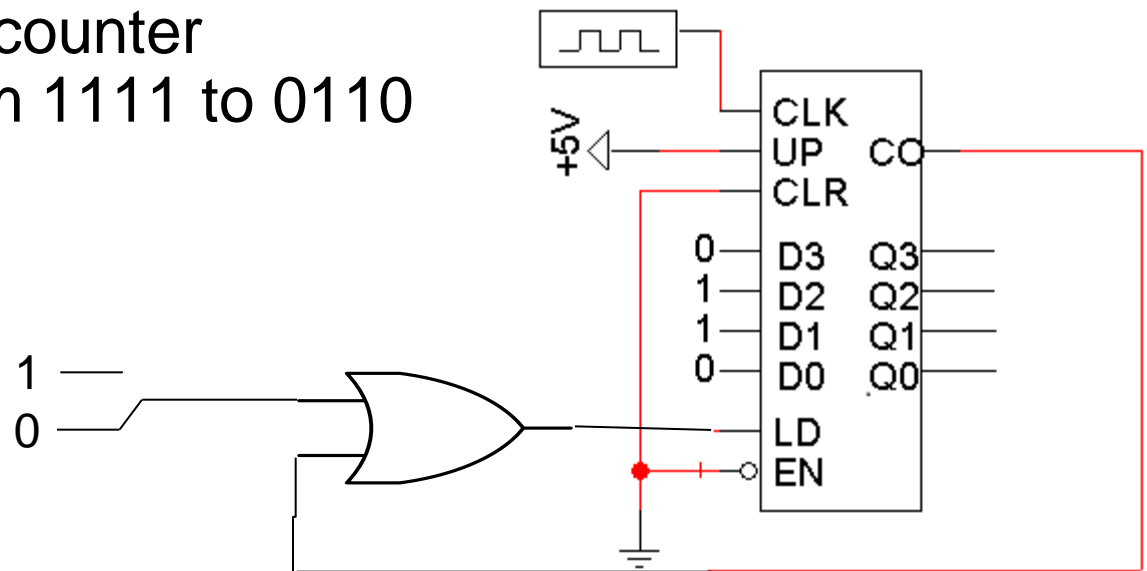
An 8-bit Counter using 4-bit Counters

- As you might expect by now, we can use these general counters to build other counters.
- Here is an 8-bit counter made from two 4-bit counters.
 - The bottom device represents the least significant four bits, while the top counter represents the most significant four bits.
 - When the bottom counter reaches 1111 (i.e., when $CO = 1$), it enables the top counter for one cycle.
- Other implementation notes:
 - The counters share clock and clear signals.
 - They always count UP.
 - The counters are never loaded.



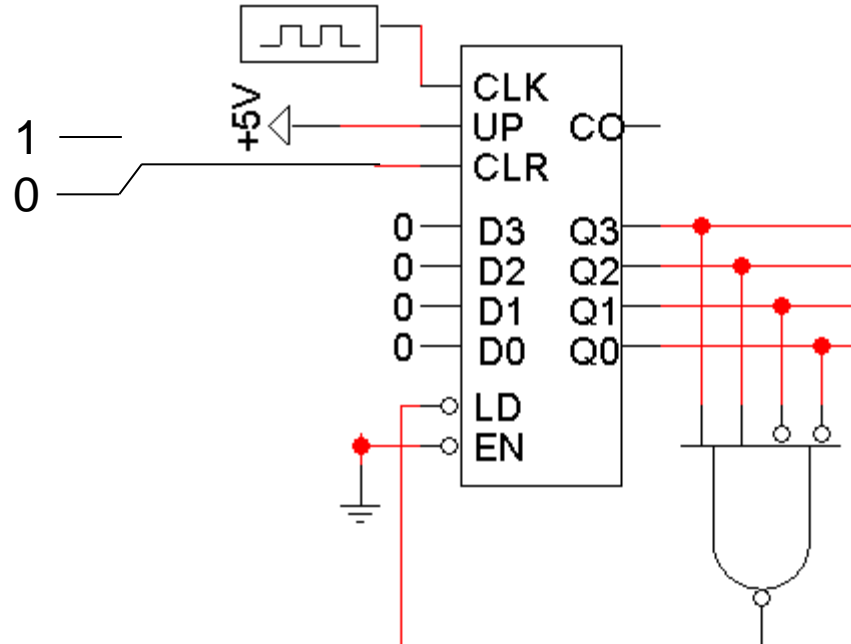
Restricted 4-bit Counter

- A counter that “starts” at some value different than 0000 and count upwards
 - Switch LD to 1, thereby loading the counter with 0110
 - Switch LD to 0, thereby starting the counter
 - When CO = 1, then LD signal forces the next state to be loaded from D₃-D₀
 - The result is this counter which wraps from 1111 to 0110 (instead of 0000)



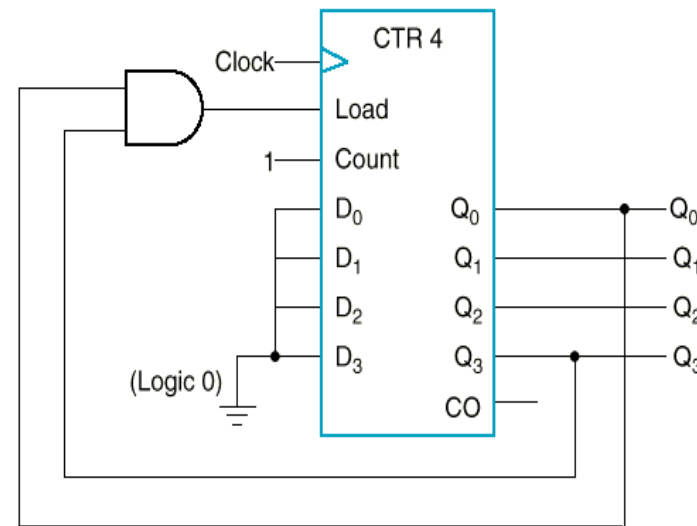
Another Restricted Counter

- A circuit that counts up to only 1100, instead of 1111
 - Switch CLR to 1, thereby resetting the counter
 - Switch CLR to 0, thereby starting the counter
 - When the counter value reaches 1100, the NAND gate forces the counter to load, so the next state becomes 0000



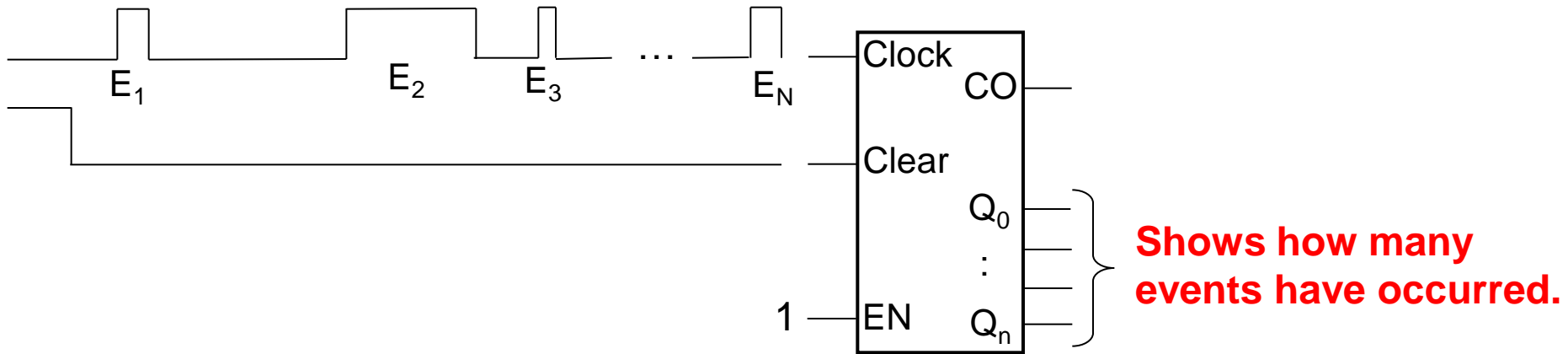
BCD Counter

- The binary counter with parallel load can be converted into a synchronous BCD counter by connecting an external AND gate to it.
- The counter starts with an all-zero output.
- As long as the output of the AND gate is 0, each positive clock pulse transition increments the counter by one.
- When the output reaches the count of 1001, both Q_0 and Q_3 become 1, making the output of the AND gate equal to 1. This condition makes Load active, so on the next clock transition, the counter does not count, but is loaded from its four inputs.
- The value loaded then is 0000.

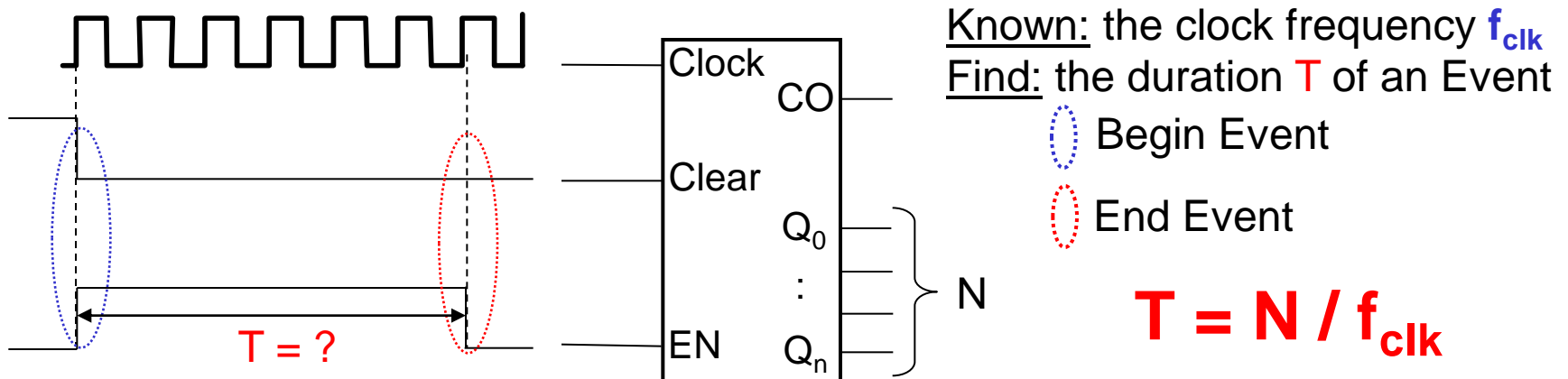


Applications of Counters

■ Counting events (E):

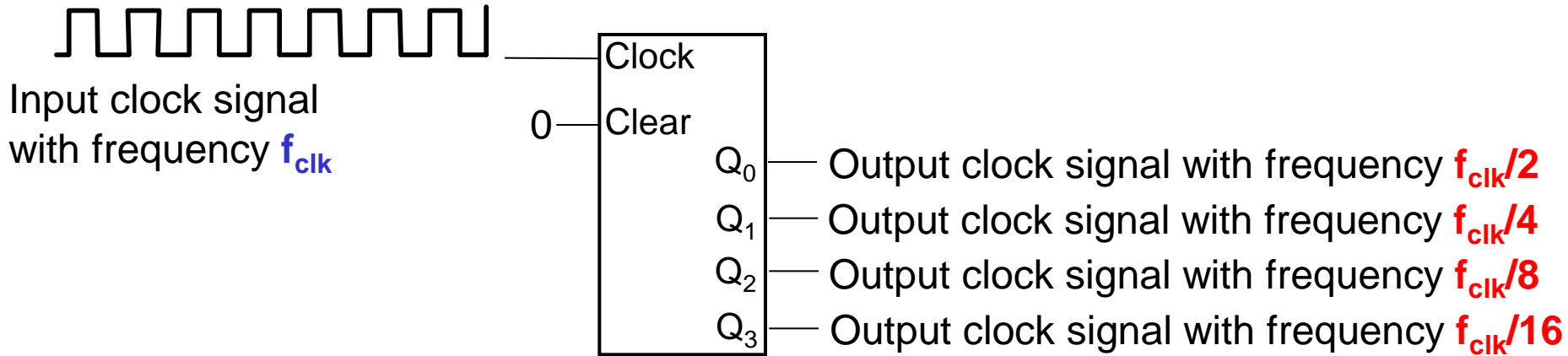


■ Simple clock to keep track of time:

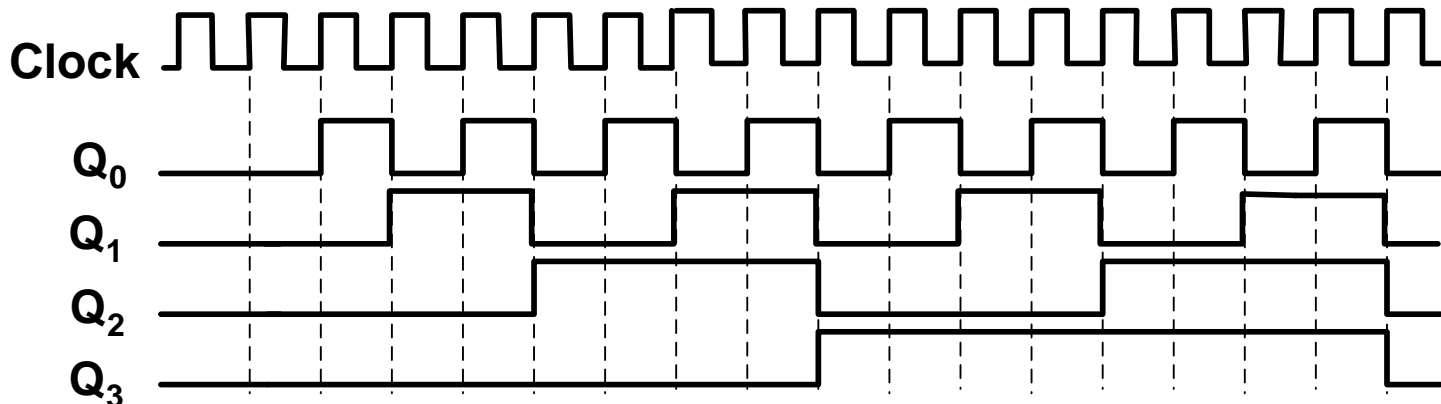


Applications of Counters (cont.)

■ Clock Frequency Divider:



To understand this recall the **Timing Diagram** of a counter!





Summary

- Counters serve many purposes in sequential logic design.
 - Counting events.
 - Simple clocks to keep track of time.
 - Clock frequency dividers.
 - Program Counters (**you will see this later or see your design project**).
- There are lots of variations on the basic counter.
 - Some can increment or decrement.
 - An enable signal can be added.
 - The counter's value may be explicitly set.
- There are also several ways to make counters.
 - You can follow the sequential design principles from Lecture 10 to build counters from scratch.
 - You could also modify or combine existing counter devices.