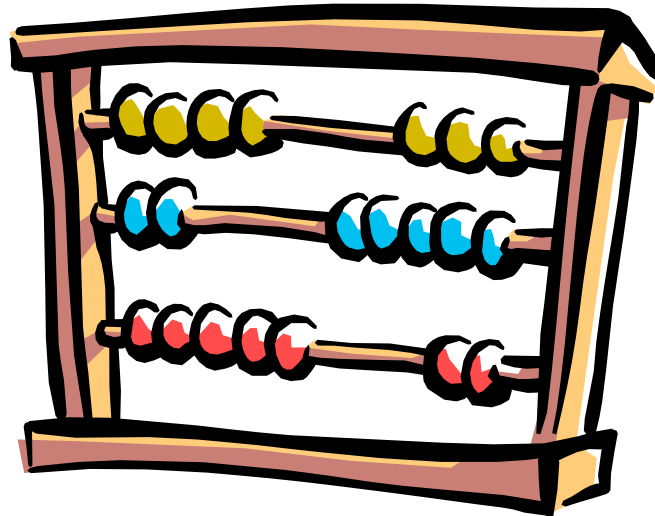


Registers and counters



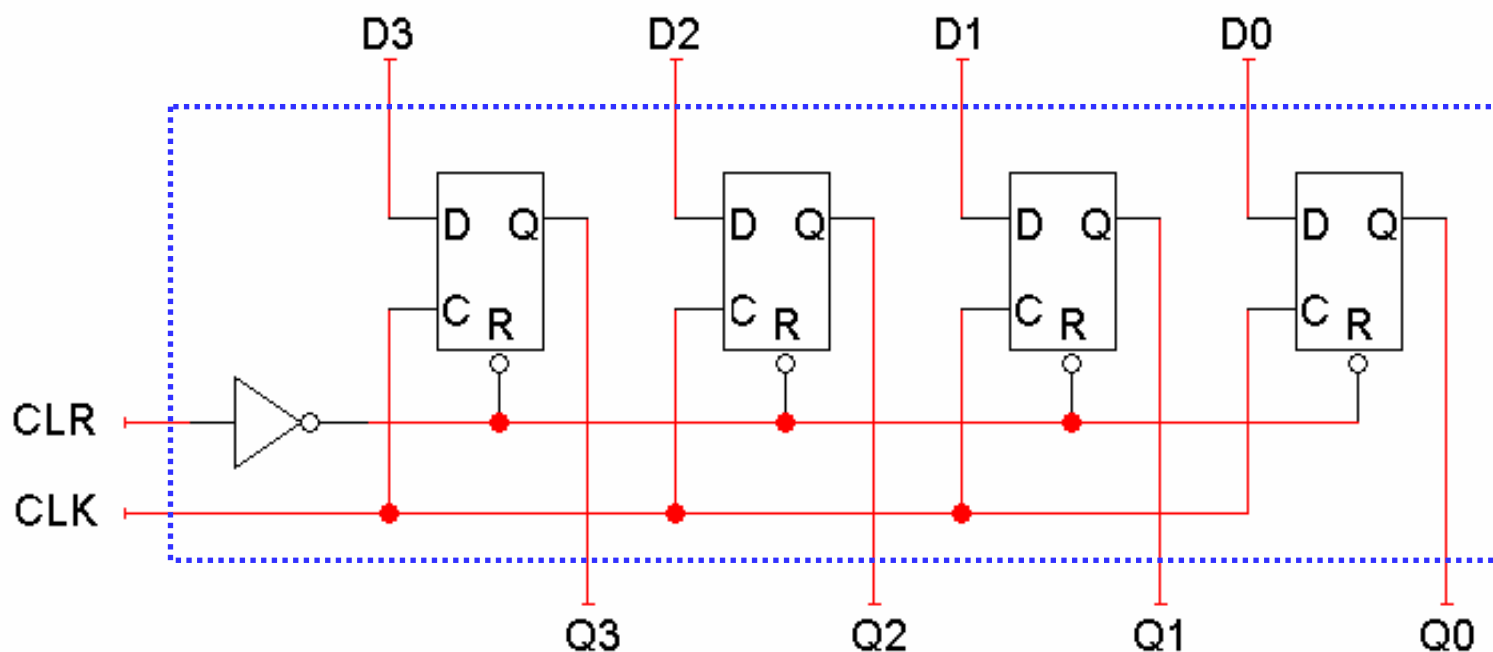
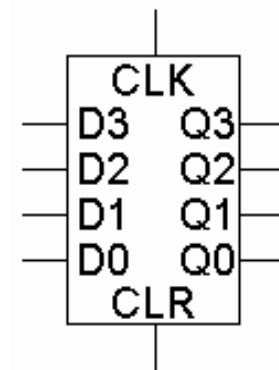
- Today we'll see two common sequential devices, registers and counters.
 - First we'll study some different kinds of **registers** and discuss how to build them. Several example circuits are also shown.
 - Then we'll talk about **counters** in more detail, looking at both some implementations and applications.
- These are not only examples of sequential analysis and design, but also real devices used in larger circuits, as we'll see in the coming weeks.

Registers

- Flip-flops are limited because they can store only one bit.
 - We had to use two flip-flops for most of our examples so far.
 - Most computers work with integers and single-precision floating-point numbers that are 32-bits long.
- A **register** is an extension of a flip-flop that can store multiple bits.
- Registers are commonly used as temporary storage in a processor.
 - They are faster and more convenient than main memory.
 - More registers can help speed up complex calculations.
- Later we'll learn more about how registers are used in processors, and some of the differences between registers and random-access memories or **RAM**.

A basic register

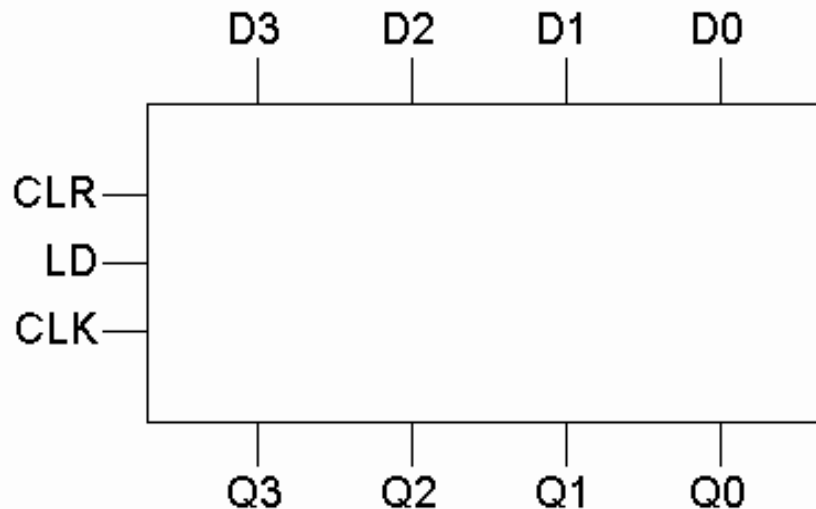
- Basic registers are easy to build. We can store multiple bits just by putting a bunch of flip-flops together!
- A 4-bit register from LogicWorks, **Reg-4**, is on the right, and its internal implementation is below.
 - This register uses D flip-flops, so it's easy to store data without worrying about flip-flop input equations.
 - All the flip-flops share a common **CLK** and **CLR** signal.



Adding another operation

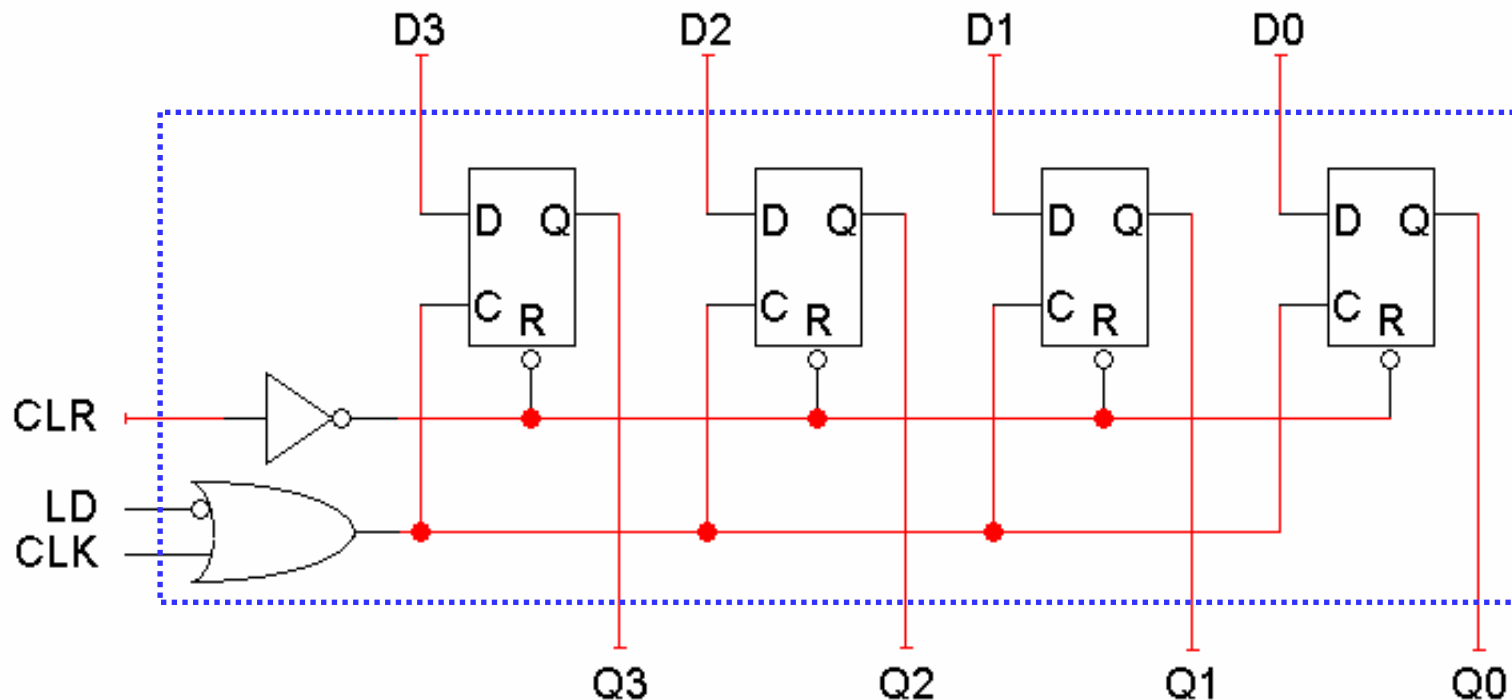
- The input **D3-D0** is copied to the output **Q3-Q0** on *every* clock cycle.
- How can we store the current value for more than one cycle?
- Let's try to add a load input signal **LD** to the register.
 - If **LD = 0**, the register keeps its current contents.
 - If **LD = 1**, the register stores a new value, taken from inputs D3-D0.

LD	Q(t+1)
0	Q(t)
1	D ₃ -D ₀



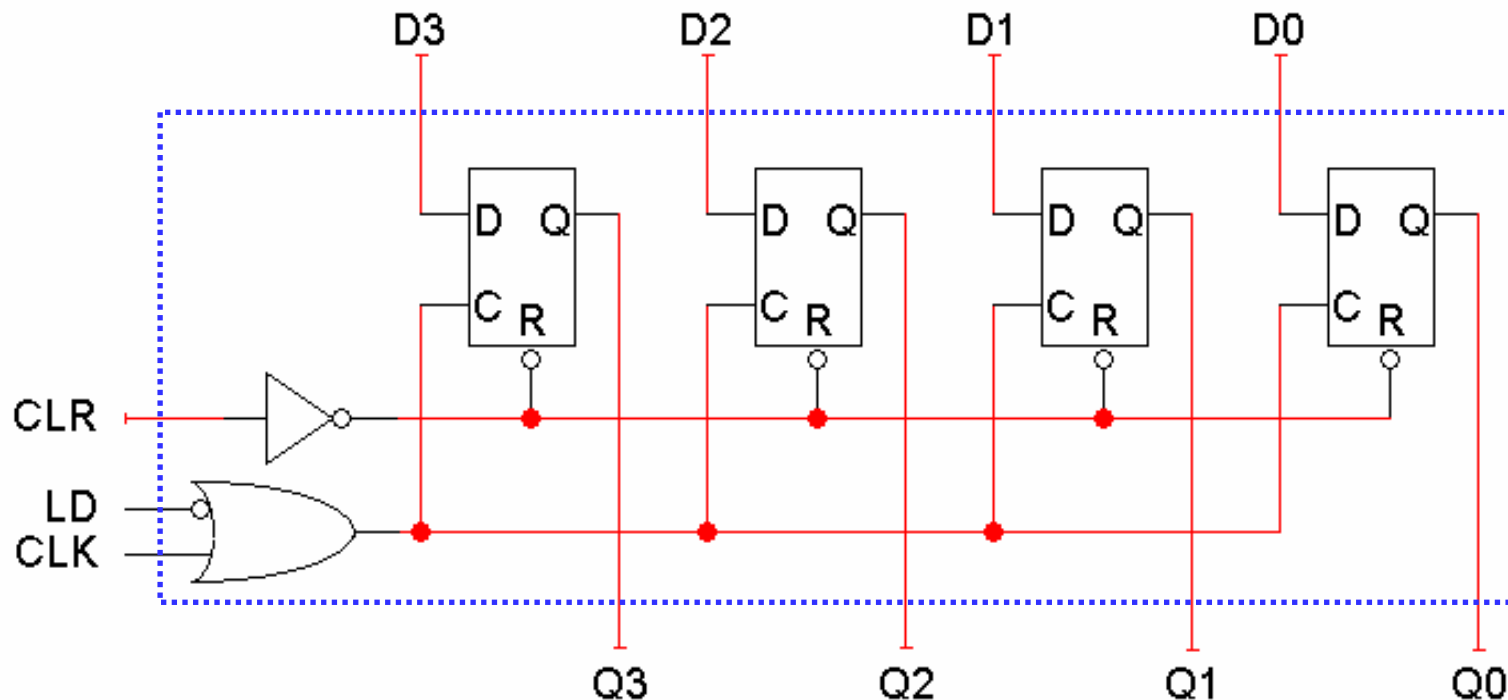
Clock gating

- We could implement the load ability by manipulating the **CLK** input, as shown below.
 - When **LD = 0**, the flip-flop **C** inputs are held at 1. There is no positive clock edge, so the flip-flops keep their current values.
 - When **LD = 1**, the **CLK** input passes through the OR gate, so all of the flip-flops will receive a positive clock edge and can load a new value from the **D3-D0** inputs.



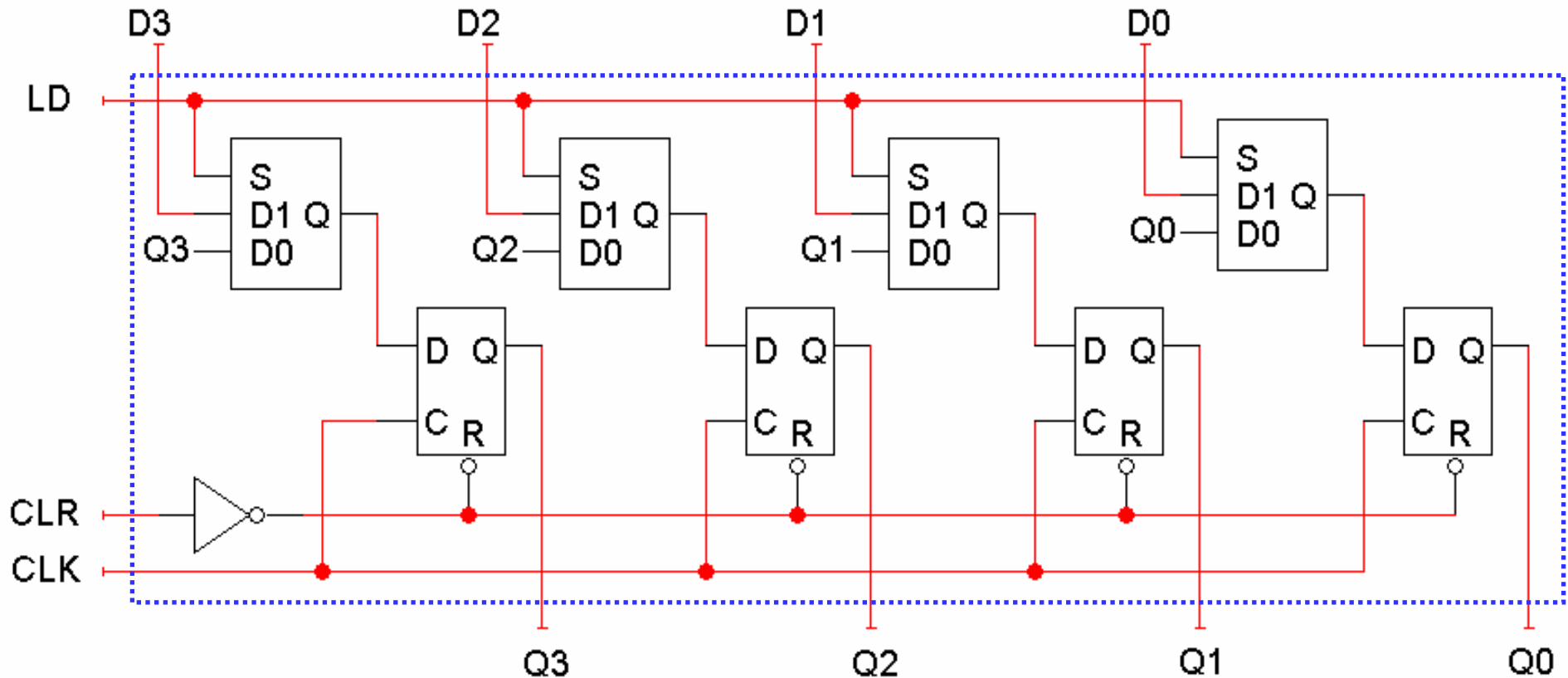
Clock gating is bad

- This is called **clock gating**, since gates are added to the clock signal.
- There can be timing problems similar to those of latches. Here, LD must be kept at 1 for the right length of time (one clock cycle) and no longer.
- The actual clock signal is delayed a little bit by the OR gate.
 - In more complex circuits, different flip-flops might receive the clock signal at slightly different times.
 - This **clock skew** can lead to synchronization problems.



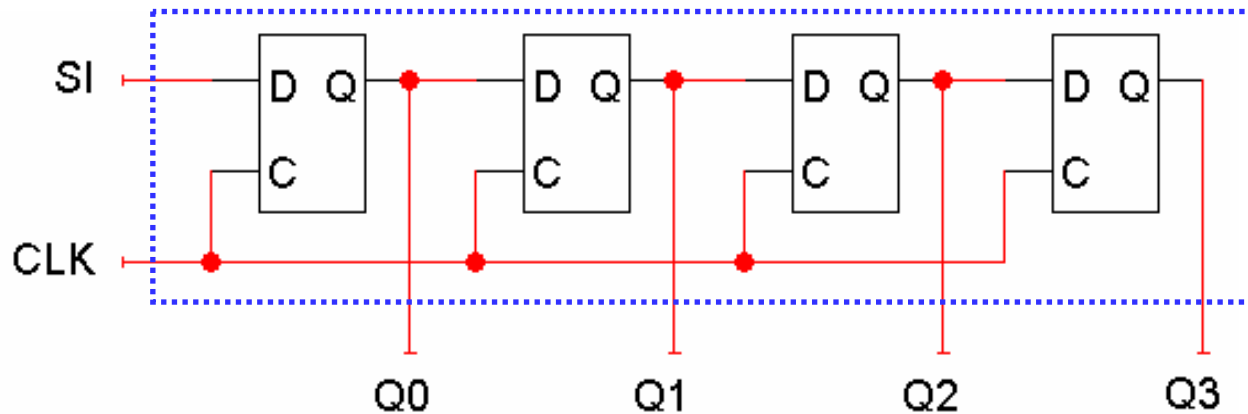
A better parallel load

- Another idea is to modify the flip-flop D inputs and not the clock signal.
 - When $LD = 0$ the flip-flop inputs are Q_3-Q_0 , so each flip-flop keeps its current value.
 - When $LD = 1$ the flip-flop inputs are D_3-D_0 , so this new value is loaded into the register.



Shift registers

- A **shift register** “shifts” its output once every clock cycle. **SI** is an input that supplies a new bit to shift “into” the register.



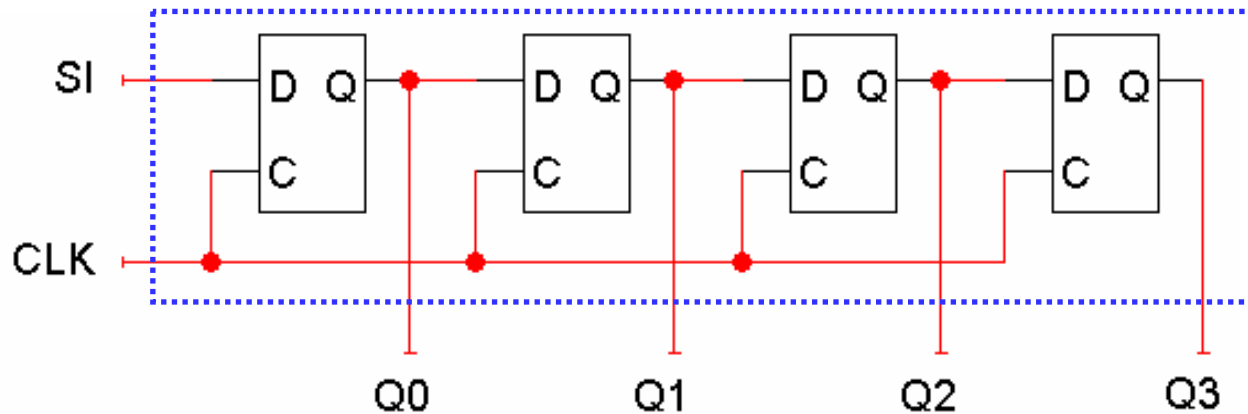
$$\begin{aligned} Q0(t+1) &= SI \\ Q1(t+1) &= Q0(t) \\ Q2(t+1) &= Q1(t) \\ Q3(t+1) &= Q2(t) \end{aligned}$$

- Here is one example transition.

Present State Q0-Q3	Input SI	Next State Q0-Q3
0110	1	1011

- The current Q3 (0 in this example) will be lost on the next cycle.

Shift direction



$$\begin{aligned}
 Q0(t+1) &= SI \\
 Q1(t+1) &= Q0(t) \\
 Q2(t+1) &= Q1(t) \\
 Q3(t+1) &= Q2(t)
 \end{aligned}$$

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

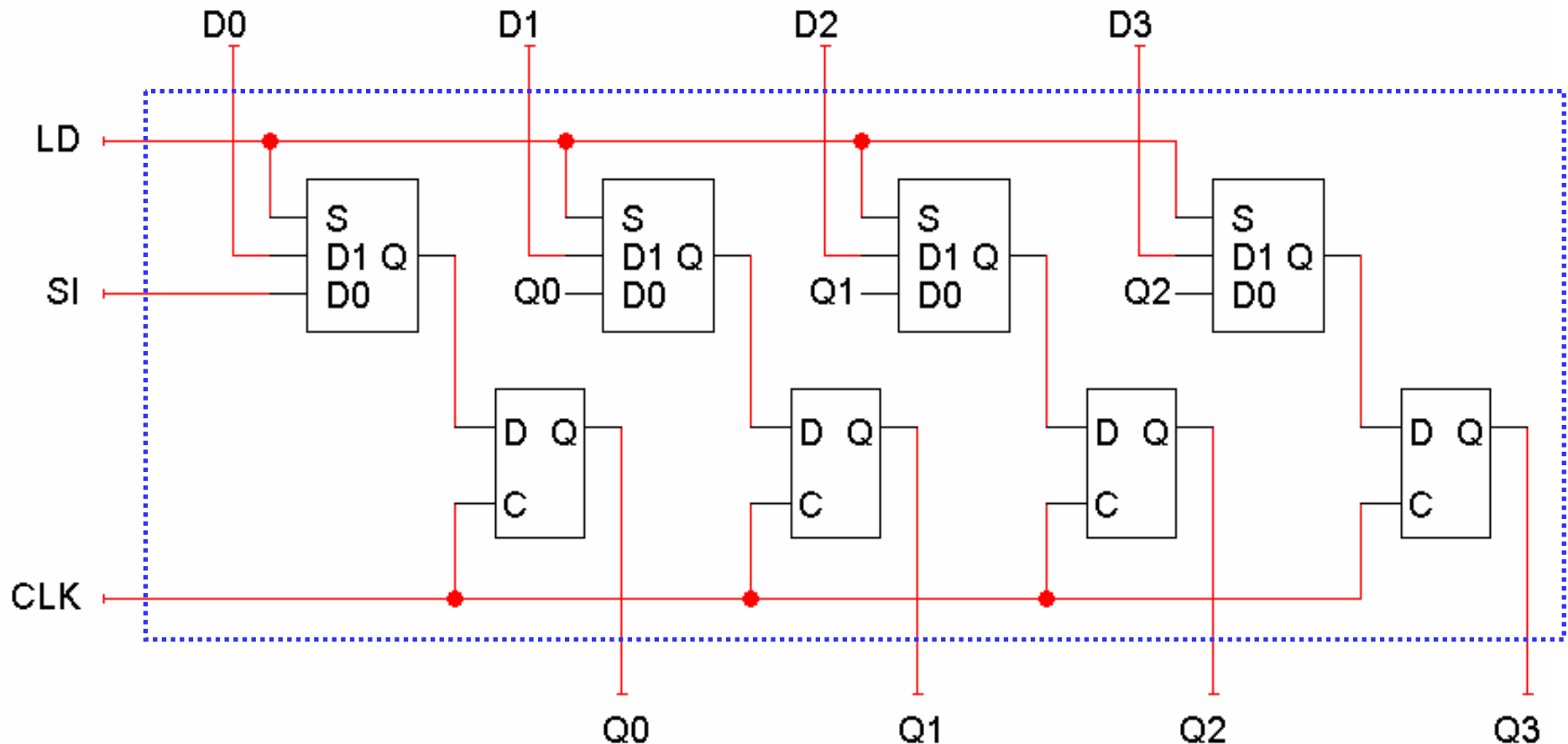
Present Q0-Q3	SI	Next Q0-Q3
ABCD	X	XABC

- But it all depends on your interpretation of the bits. If you regard Q3 as the most significant bit, then the register appears to shift in the *opposite* direction!

Present Q3-Q0	SI	Next Q3-Q0
DCBA	X	CBAX

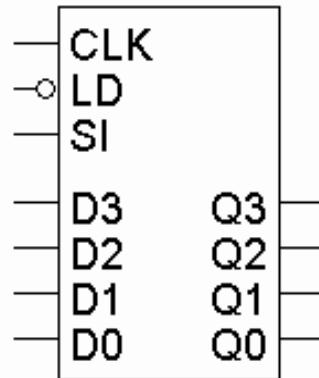
Shift registers with parallel load

- We can add a parallel load operation, just as we did for regular registers.
 - When $LD = 0$ the flip-flop inputs will be $SIQ_0Q_1Q_2$, so the register will shift 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 register on the next positive clock edge.



Shift registers in LogicWorks

- Here is a block symbol for the **Shift Reg-4** from LogicWorks.
- Its internal implementation is shown on the previous page, except the **LD** input here is active-low instead.



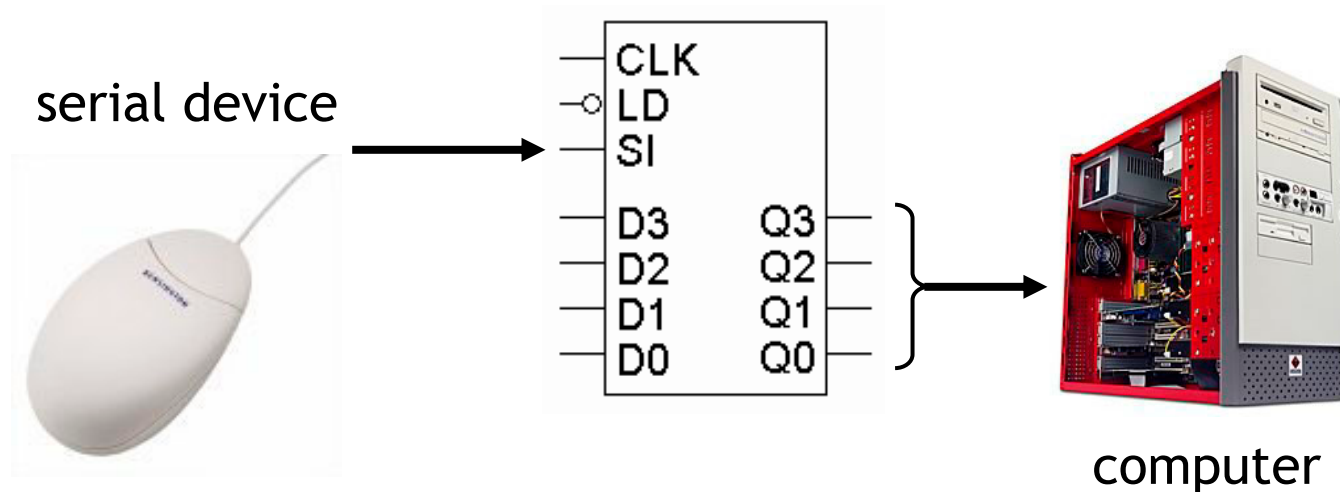
Serial data transfer

- One application of shift registers is converting between serial data and parallel data.
- Computers typically work with multi-bit quantities.
 - ASCII text characters are 8 bits long.
 - Integers, single-precision floating-point numbers, and screen pixels are up to 32 bits long.
- But sometimes it's necessary to send or receive data **serially**, one bit at a time. For example, USB and Firewire devices such as keyboards, mice and printers all transfer data serially.



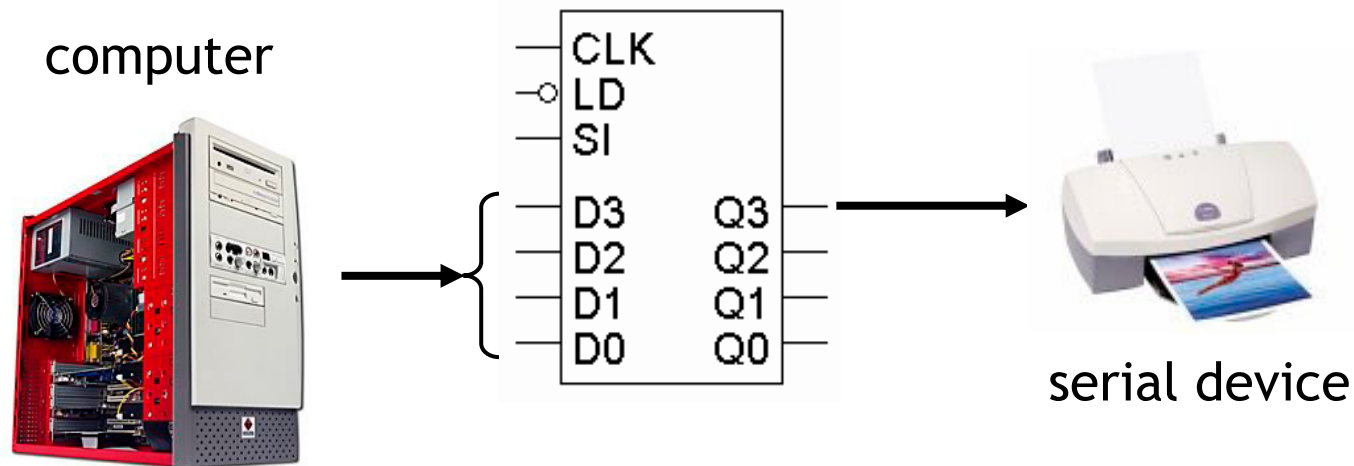
Receiving serial data

- You can convert serial data to parallel data using a shift register.
 - The serial device is connected to the register's **SI** input.
 - Shift register outputs **Q3-Q0** are connected to the parallel device.
- 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

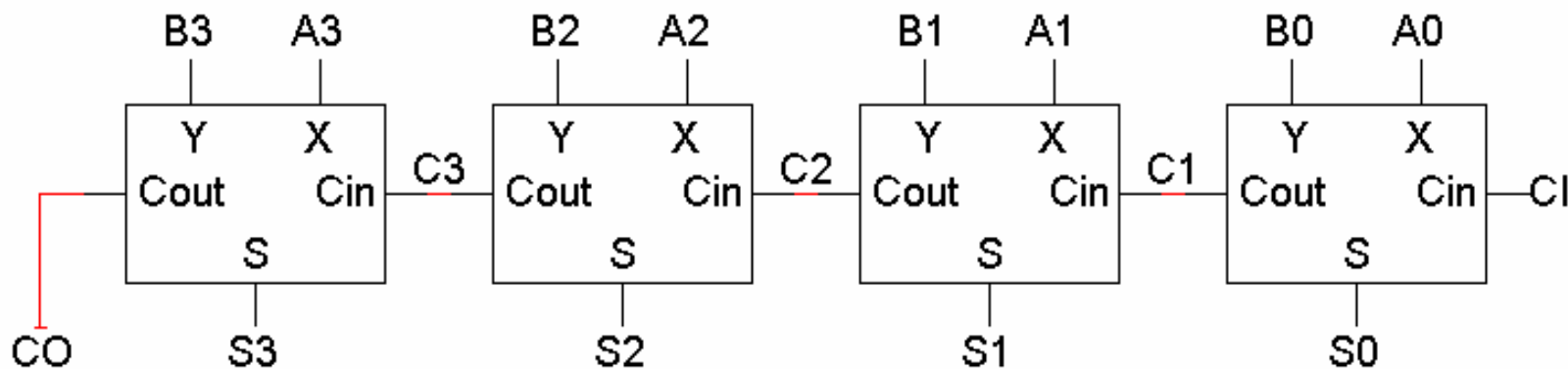
- To send data serially with a shift register, you can do the opposite.
 - The parallel device is connected to the register's **D3-D0** inputs.
 - The shift register output **Q3** is connected to the serial device.
- The computer first stores a four-bit word in the register, in one cycle.
- The serial 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.



Serial addition

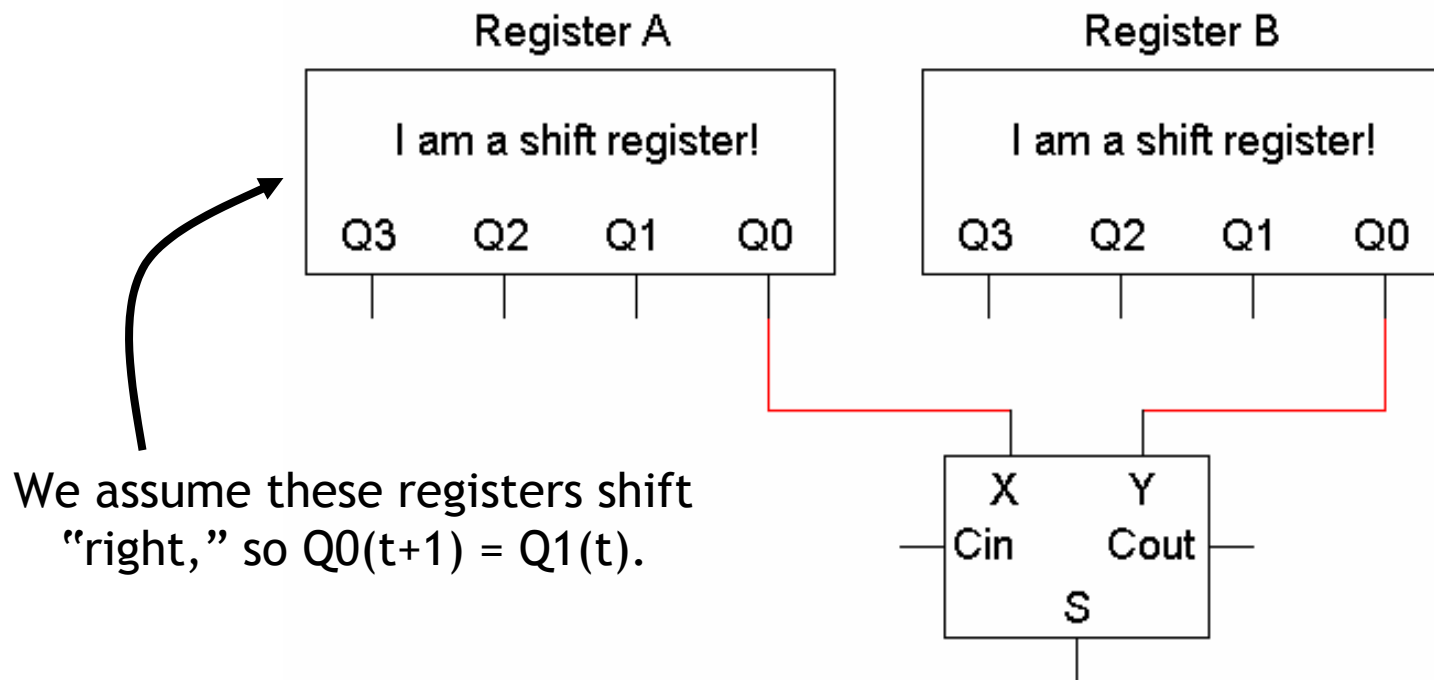
- A second example using shift registers is adding two n -bit numbers with significantly less hardware than a standard adder.
- A four-bit ripple-carry adder contains four full adders, but note that the addition really happens serially, one step at a time.

1. Add $A_0 + B_0 + C_1$ to get S_0 and C_1 .
2. Add $A_1 + B_1 + C_1$ to get S_1 and C_2 .
3. Add $A_2 + B_2 + C_2$ to get S_2 and C_3 .
4. Add $A_3 + B_3 + C_3$ to get S_3 and C_0 .



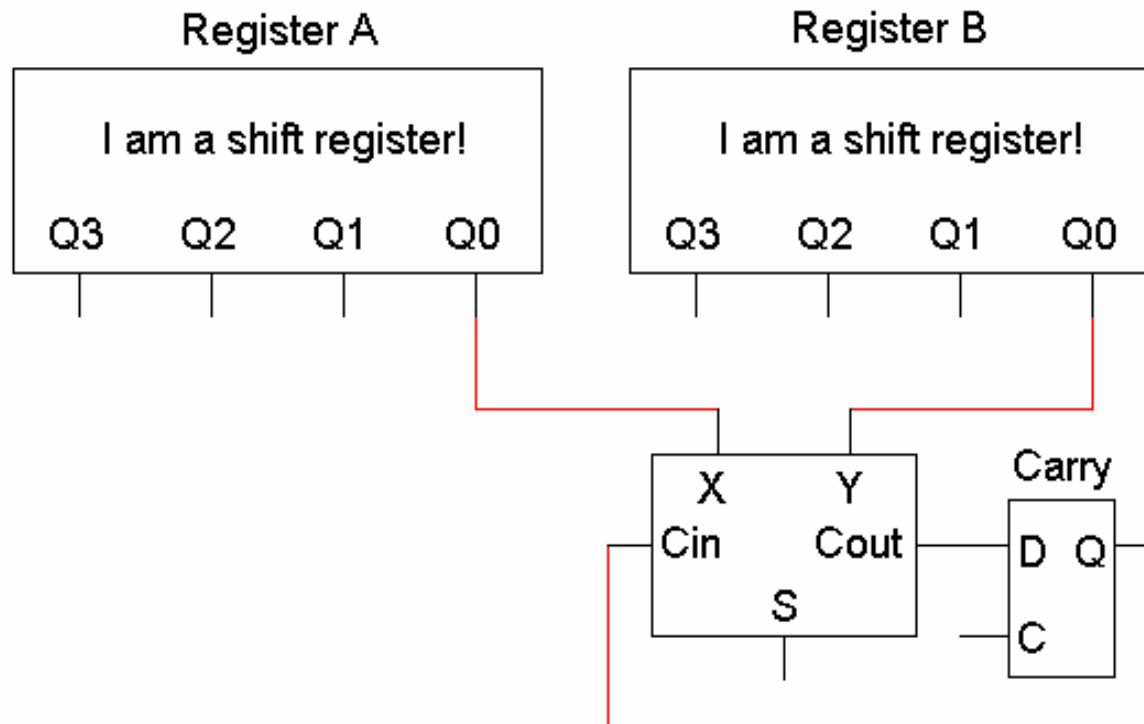
The basic setup for serial addition

- With shift registers, we can build an n -bit adder using only *one* full adder.
 - Inputs **A** and **B** are contained in shift registers.
 - Initially, the full adder computes $A_0 + B_0$.
 - On successive clock cycles, the values **A** and **B** are shifted to the right, so the adder computes $A_1 + B_1$, $A_2 + B_2$, etc.
 - The output **S** appears serially, one bit (S_0, S_1, S_2, S_3) per cycle.



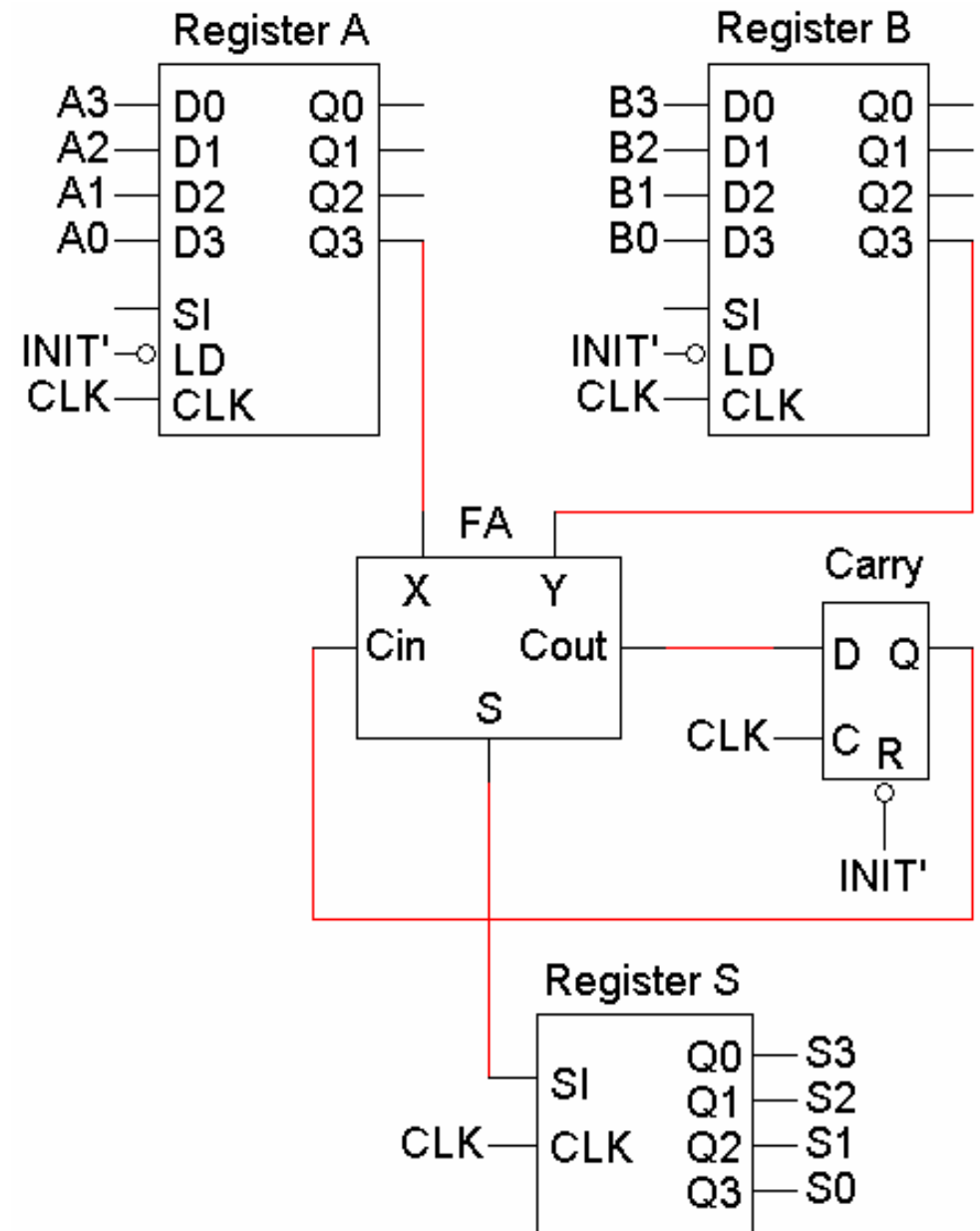
What about the carry?

- The carry out from one stage has to be added in the next stage.
- We need to add a D flip-flop as shown, so the carry out from one clock cycle is saved and used as the carry in for the next cycle.



The big unit

- First, set **INIT = 1** for one clock cycle. This loads the initial values of A and B into the shift registers on top, and sets the D flip-flop to 0 (the initial carry in).
- When **INIT = 0**, the registers will begin shifting, and the full adder results will be written to register S one bit at a time.
- The addition is completed after four clock cycles. The sum is stored in S, and the carry out is in the D flip-flop.
- Notice how we get the **Shift Reg-4** in LogicWorks to shift in the proper direction.

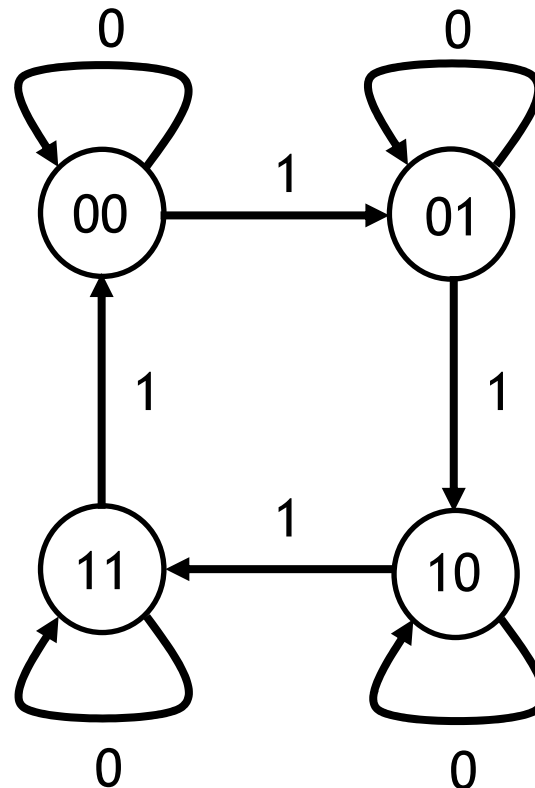


Serial addition: the good, the bad and the ugly

- There are several good things about these **serial adders**.
 - Only one full adder is needed, regardless of the length of the numbers being added. This can save a lot of circuitry.
 - Similar ideas can be applied to make serial multipliers, but with even more hardware savings.
- But there are some bad things too.
 - Adding two n -bit numbers takes n cycles, but in real processors we'd normally want the addition to be done in just one cycle.
 - Combinational circuits can use more efficient carry-lookahead adders instead of doing things purely sequentially.

A simple two-bit counter

- On Monday we saw a circuit for the two-bit **counter** shown here.
 - When $X = 0$, the next state is the same as the present state.
 - When $X = 1$, the next state is one more than the present state.
- Counters are really just another type of register—they store a multi-bit value (two bits in this case), and they support an increment operation.



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?
- 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, so the next instruction can then be executed.



Another simple counter example

- Let's try to design a slightly more advanced two-bit counter.
 - Again, the counter outputs will be 00, 01, 10 and 11, and there is a single input, X .
 - When $X = 0$, the counter value should *increment* on each clock cycle. But the value should *decrement* on successive cycles when $X = 1$.
- We'll need two flip-flops again. Here are the four possible states.

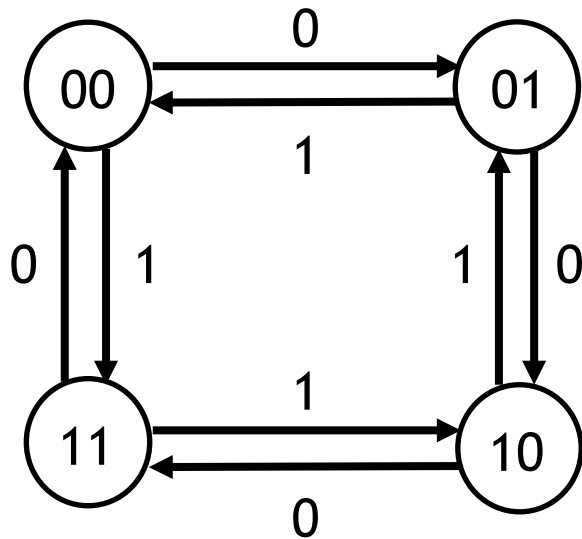
00

01

11

10

The complete state diagram and table



Present State		Inputs X	Next State	
Q1	Q0		Q1	Q0
0	0	0	0	1
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	0

D flip-flop inputs

- If we use D flip-flops, then the D inputs will just be the same as the desired next states.
- Equations for the D flip-flop inputs are shown at the right.
- Notice the second equation. Why does $D0 = Q0'$?

Present State		Inputs X	Next State	
Q1	Q0		Q1	Q0
0	0	0	0	1
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	0

		Q0	
		0	1
Q1	0	0	1
	1	1	0

X

$$D1 = Q1 \oplus Q0 \oplus X$$

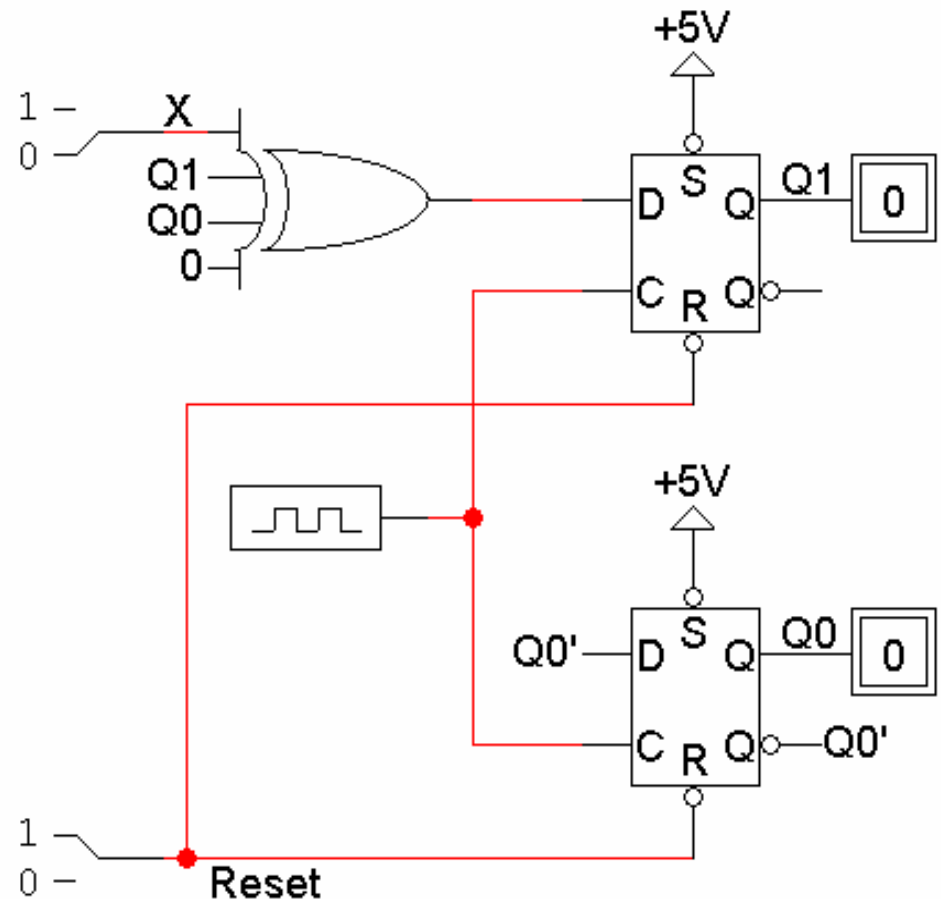
		Q0	
		0	1
Q1	0	1	1
	1	0	0

X

$$D0 = Q0'$$

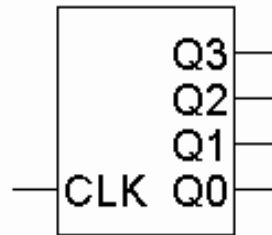
The counter in LogicWorks

- Here are some **D Flip Flop** devices from LogicWorks.
- They have both normal and complemented outputs, so we can access **Q0'** directly without using an inverter. (Q1' is not needed in this example.)
- This circuit counts normally when **Reset = 1**. But when Reset is 0, the flip-flop outputs are cleared to 00 immediately.
- There is no three-input XOR gate in LogicWorks so we've used a four-input version instead, with one of the inputs connected to 0.



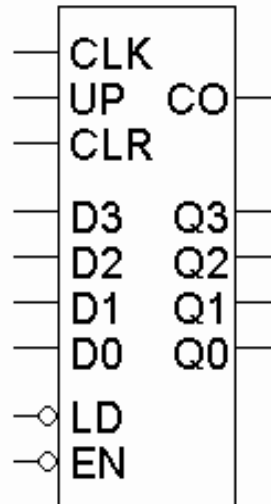
LogicWorks counters

- There are a couple of different counters available in LogicWorks.
- The simplest one, the **Counter-4 Min**, just increments once on each clock cycle.
 - This is a four-bit counter, with values ranging from 0000 to 1111.
 - The only “input” is the clock signal.



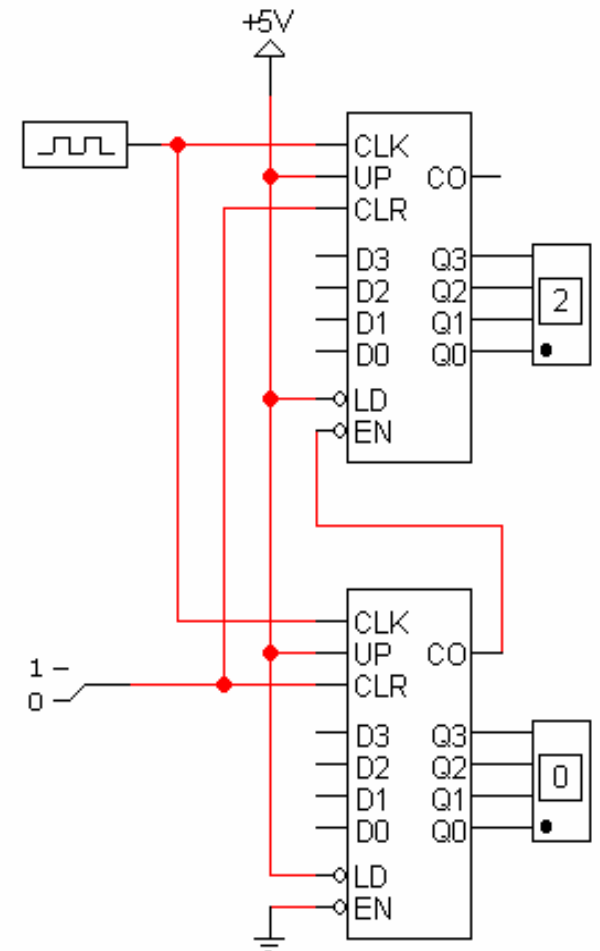
More complex counters

- More complex counters are also possible. The full-featured LogicWorks Counter-4 device below has several functions.
 - It can count up or down, depending on whether the UP input is 1 or 0.
 - You can clear the counter to 0000 asynchronously by setting CLR = 1.
 - You can perform a parallel load of D3-D0 when LD = 0.
 - The active-low EN input enables or disables the counter.
 - The “counter out” CO is normally 1, but becomes 0 when the counter reaches its maximum value of 1111 (if UP = 1) or 0000 (if UP = 0).



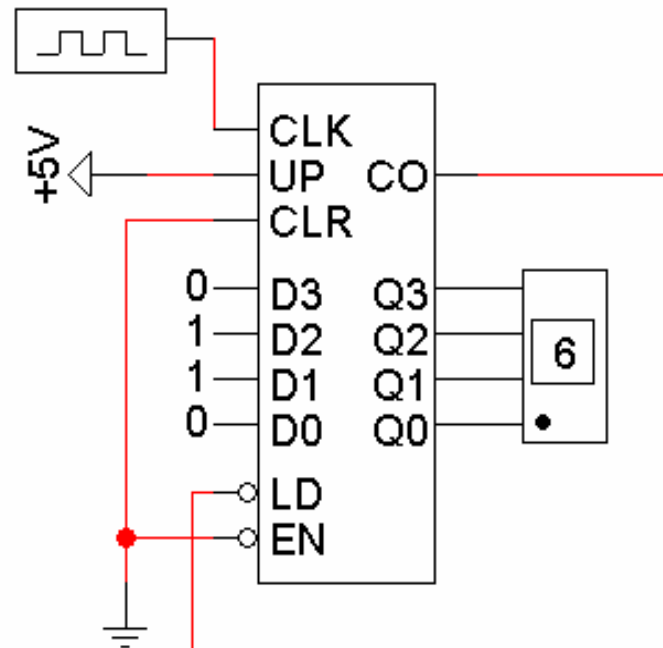
An 8-bit counter

- 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 $CO = 0$), it enables the top counter for one cycle.
- The two four-bit counters share clock and clear inputs. Sharing the clock is important to ensure that the two counters are synchronized with respect to each other.
- We've used [Hex Display](#) units here to view the four-bit output as a single hexadecimal digit.



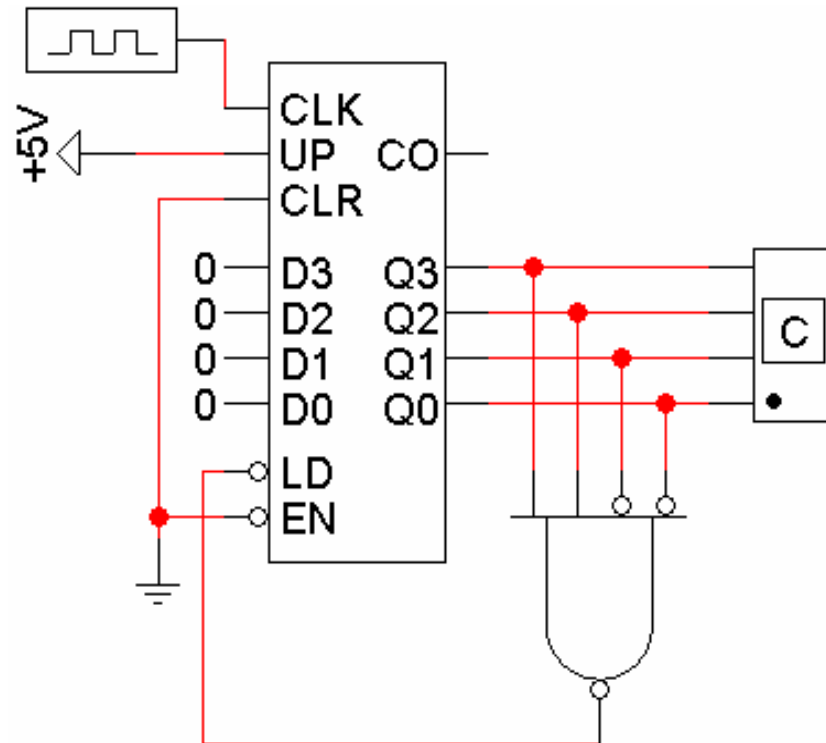
A restricted 4-bit counter

- We can also make a counter that “starts” at some value besides 0000.
- In the diagram below, when $CO = 0$ the LD signal forces the next state to be loaded from D3-D0.
- The result is this counter wraps from 1111 to 0110 (instead of 0000).



Another restricted counter

- We can also make a circuit that counts up to only 1100, instead of 1111.
- Here, when the counter output reaches 1100, the NAND gate forces the counter to load, so the next state becomes 0000.



Summary

- A **register** is made from several flip-flops, so it can store multi-bit values.
- There are many possible operations you can add to a basic register.
 - A **parallel load** register can load multi-bit values in one clock cycle.
 - **Shift registers** can shift their contents left or right on every cycle.
 - **Counters** are register that can increment or decrement each cycle.
- Shift register applications include handling serial data transfers and doing arithmetic operations like addition or multiplication.
- Counters are frequently used as simple clocks to keep track of time or the number of occurrences of some event.
- As usual, larger registers and counters can be built from smaller ones.

