University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 120: Introduction to Computing

Static Hazards*

* *

Circuit Timing Can Cause Problems with Functionality

For our class, you need understand only the basics of timing:

- **how to estimate delay** (as gate delays),
- and **how to check for stable states** (trace changes until nothing changes).

In later classes, you will need to understand timing more deeply.

So let’s take a look at how timing matters, just as a preview.

* *

Hazards, Glitches, and Errors

When a circuit **may have a problem** due to timing, we say that the circuit has a **hazard**.

If a combinational circuit's output is temporarily incorrect, we say that its output exhibits a **glitch**.

When a sequential circuit enters a state (a set of stored bits) that it should not enter by design, we say that the circuit has an **error**.

Typically, an error implies a glitch, which in turn implies a hazard, but not vice-versa.

* *

Static Hazards Allow for Change in Constant Output

The notes (Section 2.6.3* and following) discuss three types of hazards.

Static hazards

- allow a combinational circuit's output to change when moving between input combinations that should produce the same output.
- With a **static-1 hazard**, for example, both input combinations **should produce a constant output of 1**, but the **output may drop to 0** briefly because of timing.

* *

Dynamic Hazards Allow Bouncing in Changing Output

Dynamic hazards

- occur when an input combination changes from one that should produce an output of 0 to a combination that should produce an output of 1 (or vice-versa).
- In these cases, the **output should change exactly once**.
- If a dynamic hazard is present, the **output may bounce between 0 and 1** before settling to its final value.

* *

Essential Hazards Cannot be Eliminated

Essential hazards

- are **related to the function implemented** by the circuit.
- Unlike static and dynamic hazards, **they cannot be eliminated**.

In clocked synchronous sequential circuits (and, thus, in the designs in our class), all essential hazards are mapped to **clock skew**.

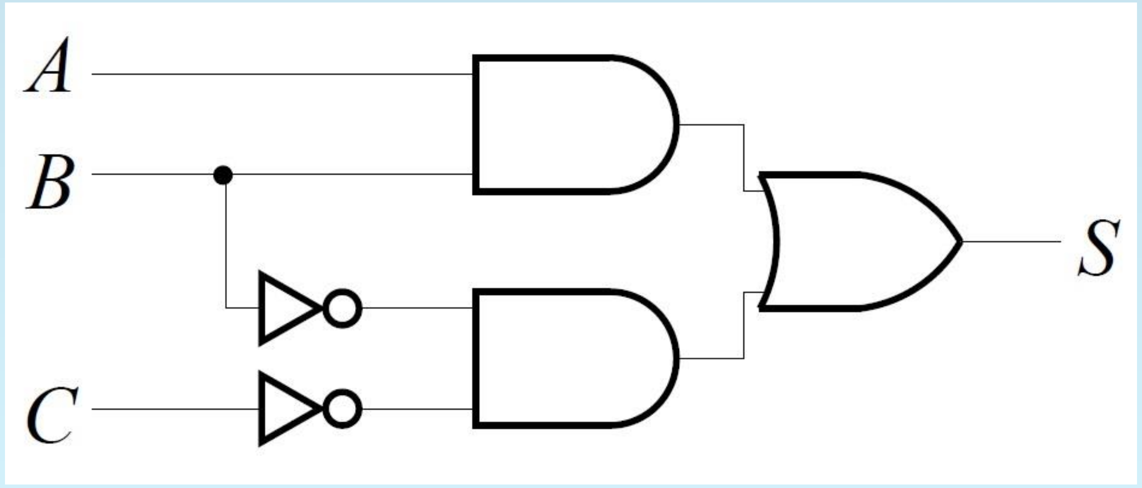
* *

Removing Static Hazards

If glitches in a circuit's output can cause problems, one can eliminate all static hazards.

Consider the circuit below. **What is S?**

S = AB + B'C'



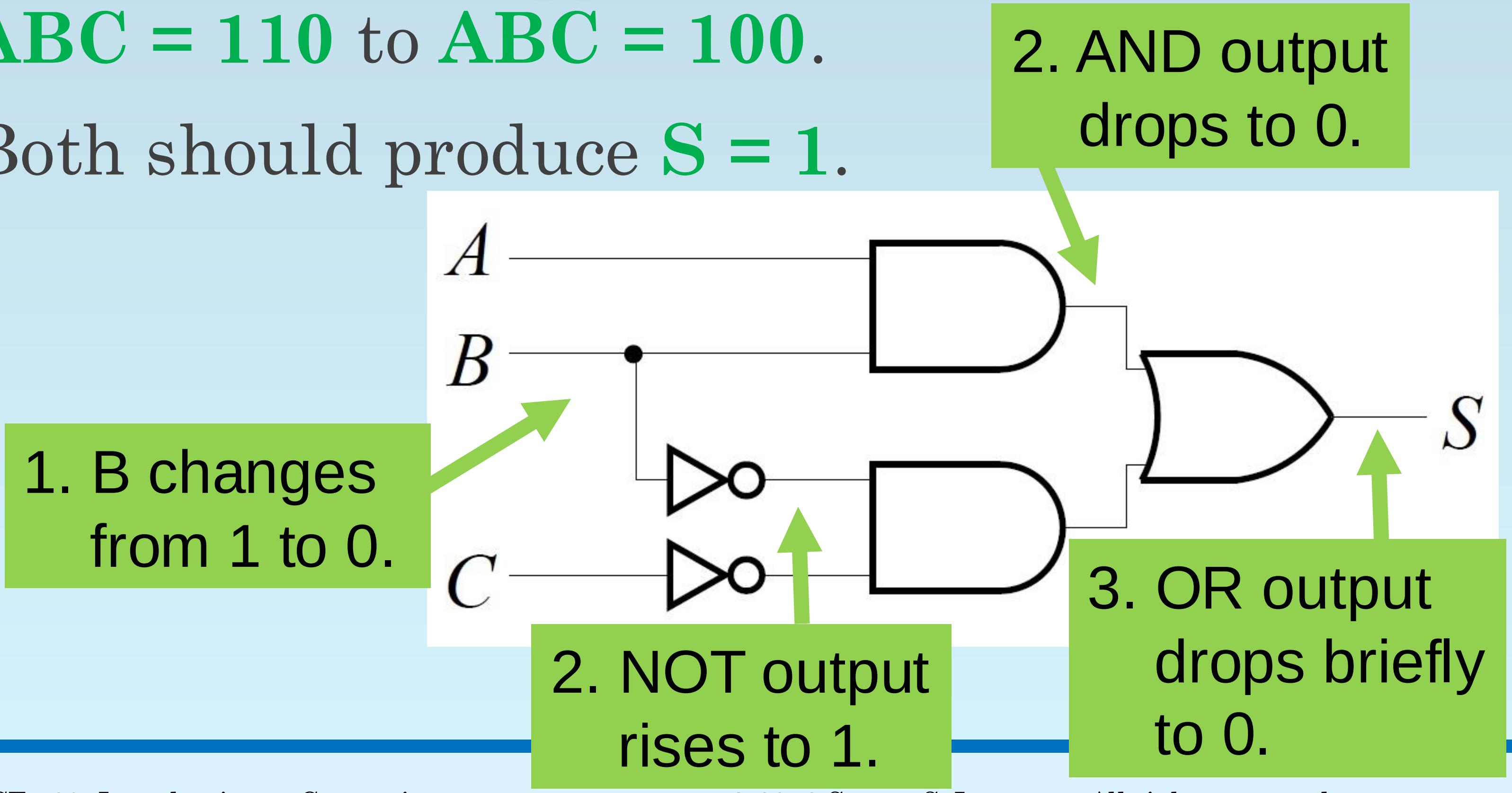
* *

The Output S Should Stay at 1

So **S = AB + B'C'**.

Let's see what happens when we move from **ABC = 110** to **ABC = 100**.

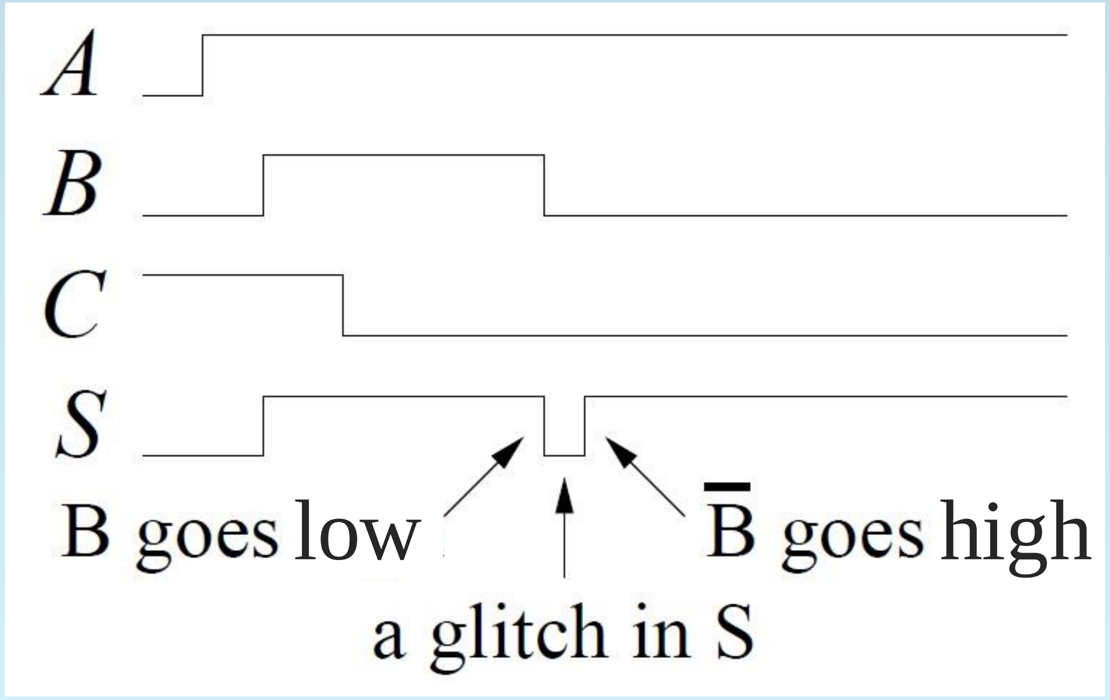
Both should produce **S = 1**.


2. AND output drops to 0.
A
B
C
S
1. B changes from 1 to 0.
2. NOT output rises to 1.
3. OR output drops briefly to 0.

* *

The Output S Drops to 0

The output glitches because the inverter for **B** delays the change in the lower AND gate.



* *

Fix Static Hazards by Adding More Gates

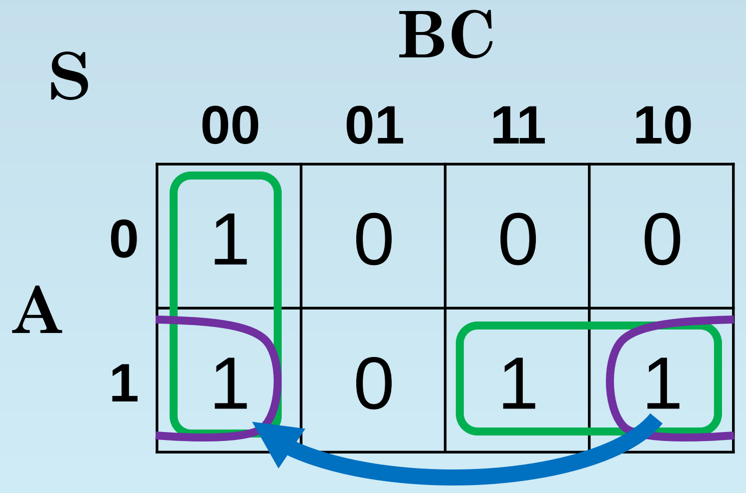
What can we do?

Take a look at the K-map.

The loops represent the AND gates.

ABC = 110 to **100** moves between loops.

Let's add a **new loop** (and a new AND gate).



The new AND gate will stay at 1.

* *

Read the Notes for More Information

See Notes Sections 2.6.3* through 2.6.6*
if you want to learn more.

University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 120: Introduction to Computing

Registers

A Register Stores a Set of Bits

Most of our representations use sets of bits: **unsigned**, **2's complement**, **floating-point**, **ASCII**.

Even messages between bit slices often require more than one bit to convey a given meaning.

A flip-flop stores a single bit.

A **register** is
- a storage element
- **composed from one or more flip-flops**
- **operating on a common clock**.

Add an Input to Control Changing a Register's Bits

A flip-flop stores a new bit every cycle.

With registers, we want to control
when the bits change value.

So we **add a LOAD** (or LD) **input**.

When **LOAD = 1 on a rising clock edge**, the
register stores a new set of bits.

When **LOAD = 0**, the
register retains its currently stored bits.

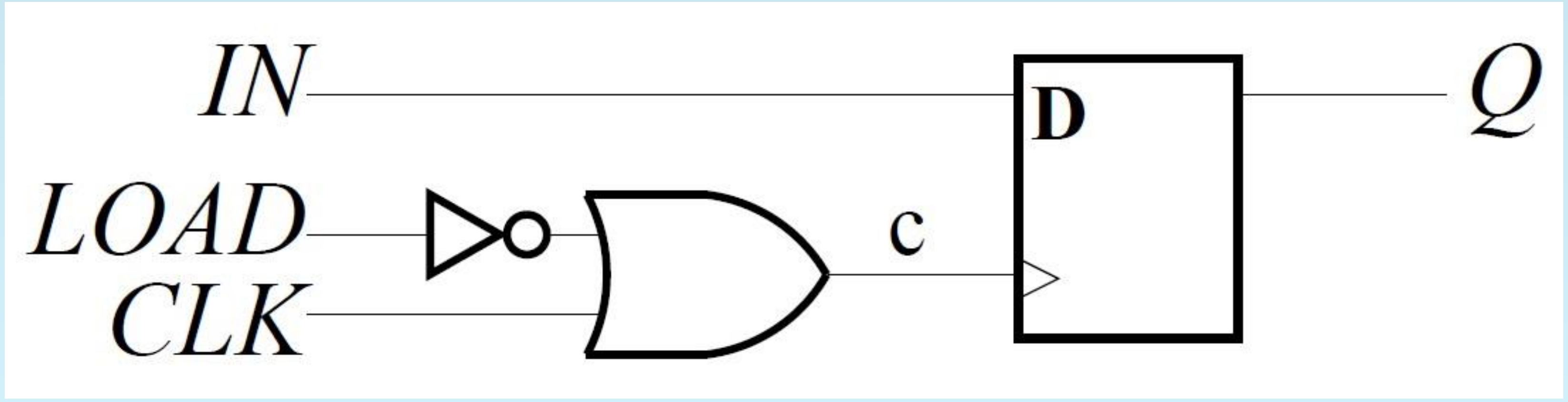
Clock Gating Uses Extra Gates to Hide the Clock Signal

How should we implement the **LOAD** input?

The approach below may seem attractive.

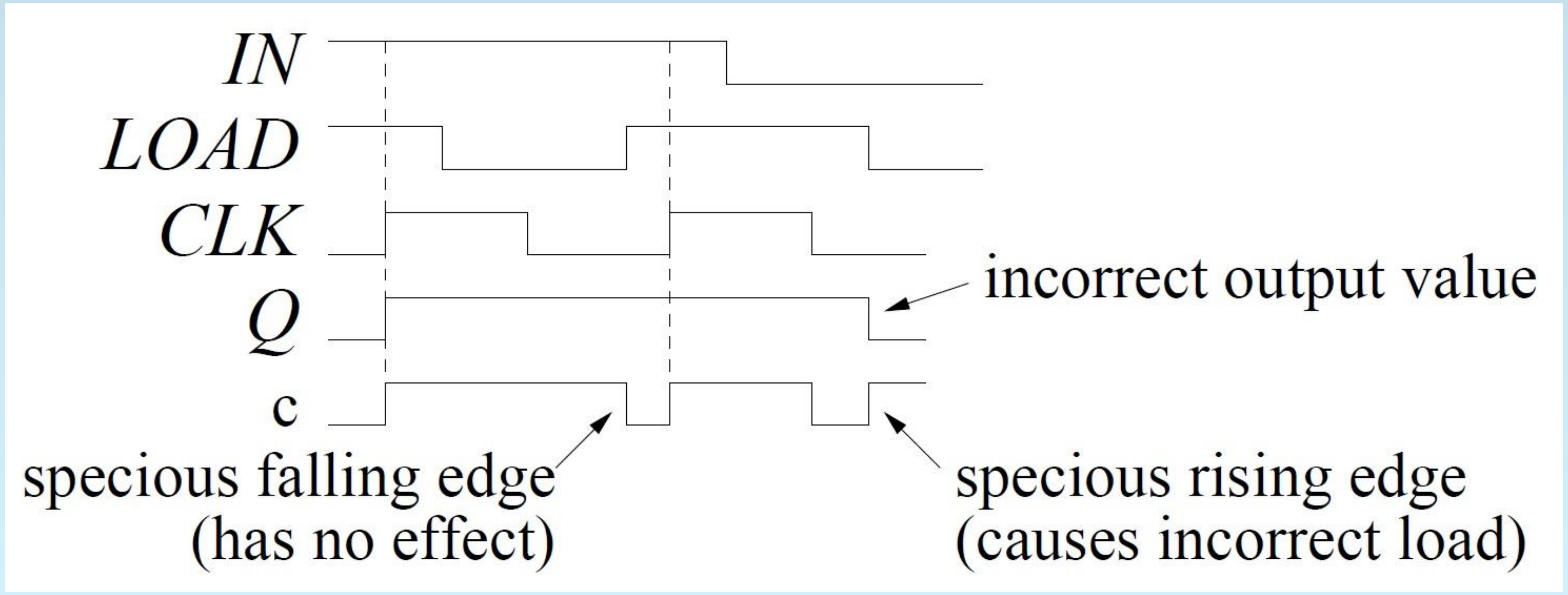
It's called **clock gating**.

Generally, you should avoid this technique.



Changes to LOAD Must be Timed Carefully

From previous figure, **c = LOAD' + CLK**.



IN
LOAD
CLK
Q
c
incorrect output value
specious falling edge (has no effect)
specious rising edge (causes incorrect load)

Clock Gating Contributes to Clock Skew

More importantly,

- the **extra gates** in front of CLK
- **contribute to clock skew**!

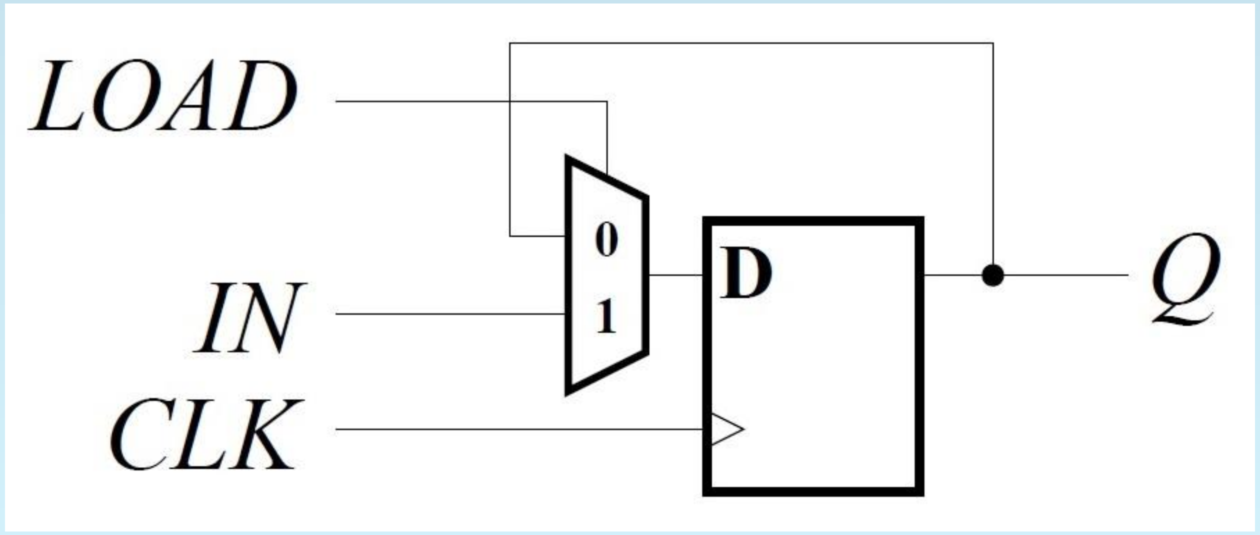
So clock gating adds further complexity to the problem of distributing the clock signal to all of the flip-flops.

Except for one application (ripple counters, in a couple of weeks), you should always use and assume a common clock signal in our class (no clock gating).

LOAD Controls Whether a Register Loads a New Value

So the question remains: **How should we implement the LOAD input?**

Use a mux!

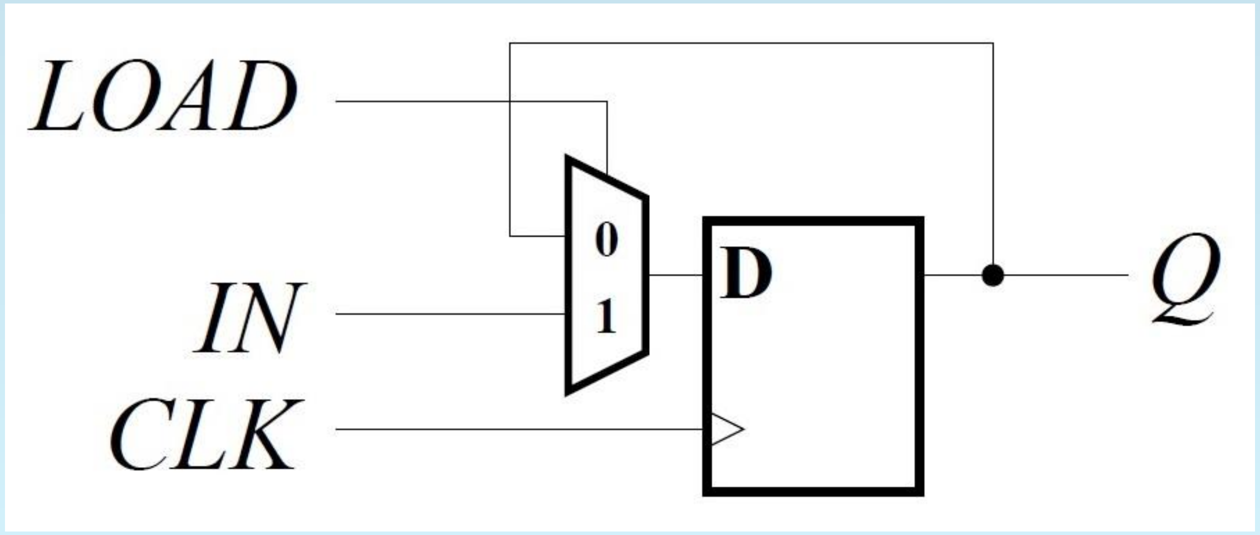


A 1-Bit Register with a LOAD Input

The design below is a **1-bit register**.

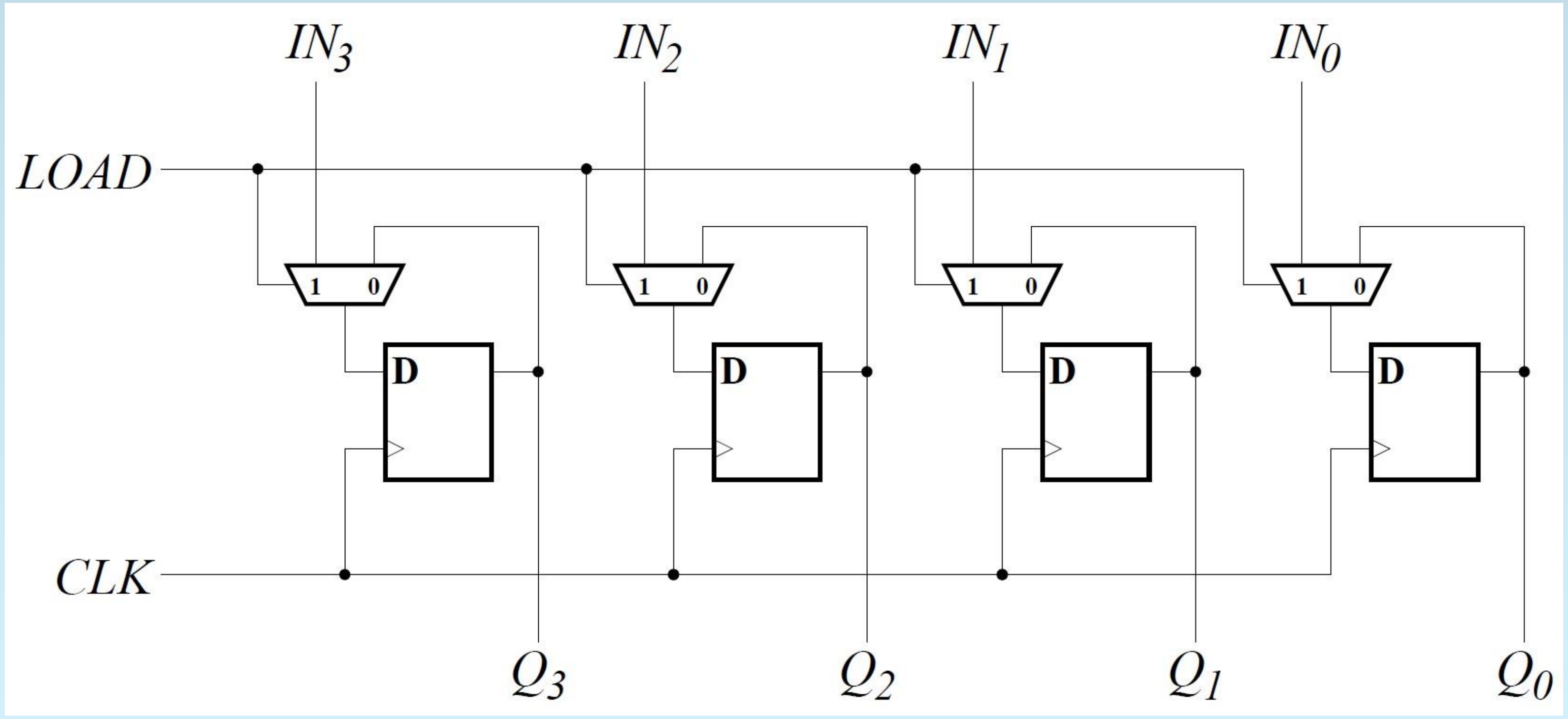
How can we create an N-bit register?

Use this design as a bit slice.



The LOAD Signal Controls All Bits of the Register

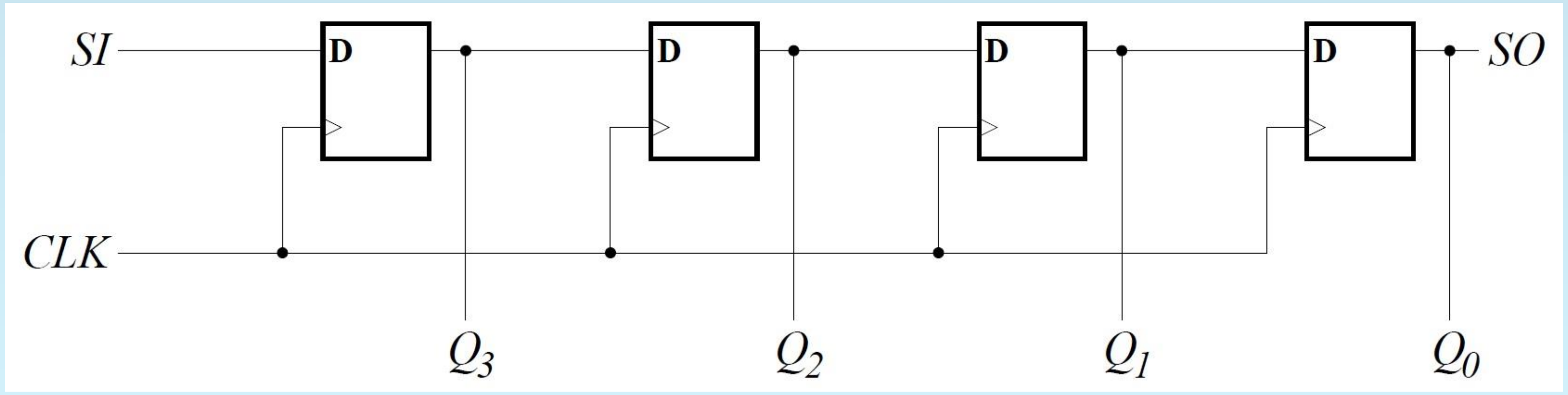
A 4-bit register with **parallel load**.



A Shift Register Shifts Bits from Flip-Flop to Flip-Flop

If we need to load registers one bit at a time, we can construct a **shift register**, as shown below (this one is a **right shift register**).

In every cycle, bits shift in from serial input **SI** and shift out through serial output **SO**.



Simple Shift Registers Have Many Applications

For example, optical networks can transmit bits at rates above **100×10^9 / second** (**100 Gbps**), but CMOS clock speeds rarely exceed **4-5 GHz**.

Deserialization (and **serialization**, **SERDES**) can be done with shift registers:

- shift into a **25-bit shift register** at **100 GHz**,
- then read **25 bits** out in parallel at **4 GHz**.

Shift Registers Provide Fixed Delay

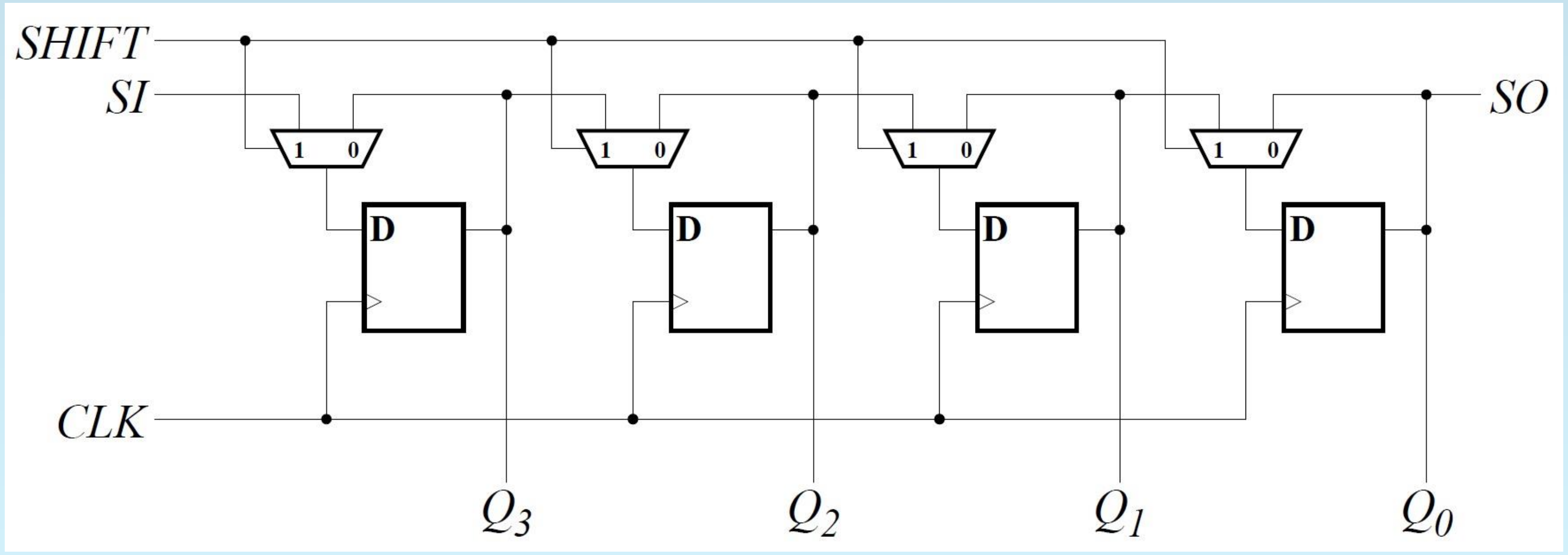
An example:

Data arrive from memory in a block (in a single cycle), but different parts of the data are needed in different cycles.

Solution? Use shift registers to deliver each part of the data to the computation elements in the correct cycle.

Shift Registers Can Also Be Designed to Stop Shifting

A shift register need not shift in every cycle. Below, we use the **SHIFT** input to make the register hold its current value (**SHIFT = 0**).



Shift Registers Also Require Fewer Wires

Serial load (shift registers) is also useful

- when wires are the limiting resource,
- as is usually the case with pins on a chip.
- Recall that parallel load of an **N-bit register** requires **N** input wires (not counting **LOAD**).

Examples of such applications include

- **configuration of reconfigurable hardware** such as Field-Programmable Gate Arrays (FPGAs, which you will use in ECE385),
- and **testing of digital systems** (shift bits in, run for a cycle, shift bits out for testing).

Many Options for the Design of Shift Registers

direction (meaningful for some representations):

- **right**: from most significant bit (MSB) to least significant bit (LSB)
- **left**: from LSB to MSB.

boundaries: how to manage serial input

- exposed: input signal for serial input **SI**
- **logical**: shift in 0s (serial input)
- **arithmetic**: shift based on 2's complement
- **cyclic**: connect **SO** back to **SI**, possibly through another register (allows building of bigger shifts from smaller ones).

We Can Combine Several Types

But we don't have to pick one design.

Let's build one register that performs one of four distinct operations based on control inputs **C_1C_0**.

For example…

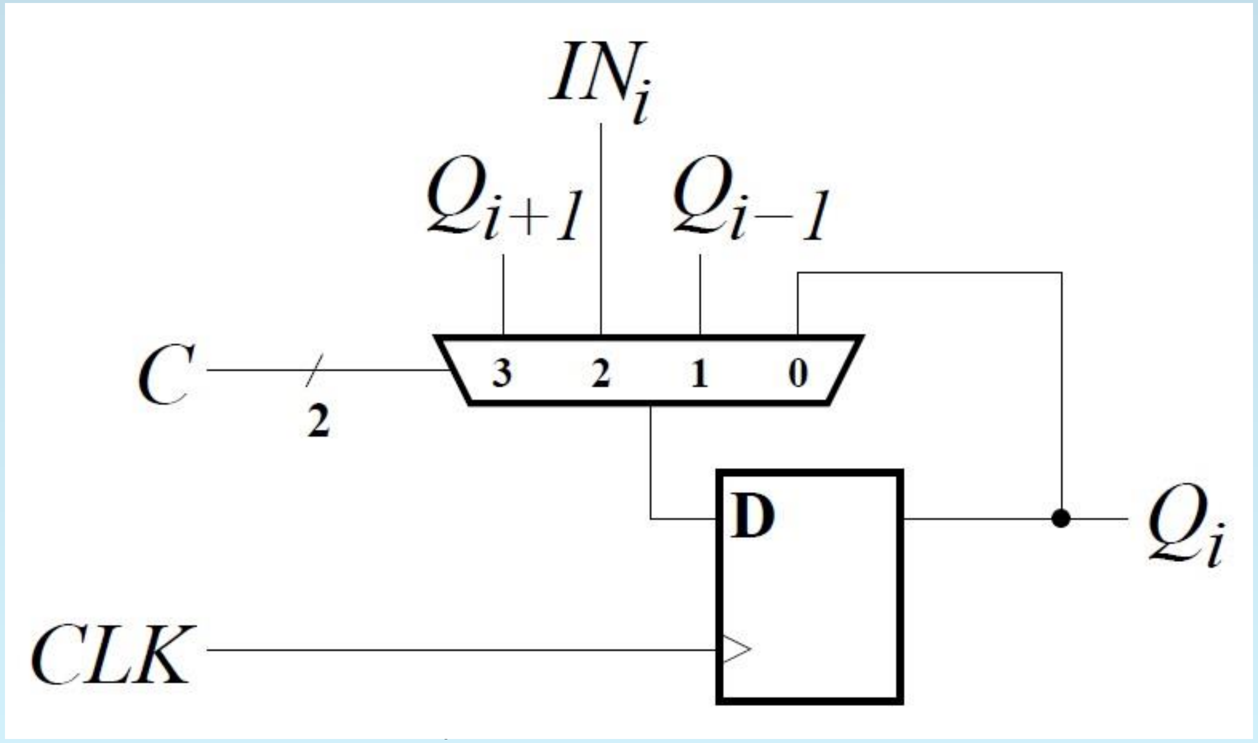
How can we build it?

With a mux!

C_1C_0	meaning
00	retain current value
01	shift left (low to high)
10	load new value (from **IN_i**)
11	shift right (high to low)

We Can Combine Several Types

Each bit of the register uses a **4-to-1 mux**.


IN_i
Q_{i+1}
Q_{i-1}
C
2
3
2
1
0
D
Q_i
CLK