SIENAcollege
Computer Science

Computer Science 220
Assembly Language & Comp. Architecture
Siena College
Fall 2011

Topic Notes: Sequential Circuits

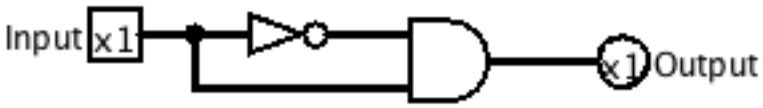
We have seen many examples of what we can do with *combinational logic* – taking a set of inputs, sending them through some circuit, producing a set of outputs. Once the circuit has computed the outputs for a given input, the outputs will not change until the inputs change.

It doesn't matter what the previous inputs were, the current inputs completely determine the current outputs.

Next, we are going to consider circuits whose outputs not only depend on the current inputs, but possibly on previous inputs. These are *sequential circuits*.

Edge Detection

We begin with a simple-looking circuit:



Logisim Circuit:
`~jteresco/shared/cs220/examples/logisim/leadingedgedetector.circ`

What is the output here? It should be always 0, right?

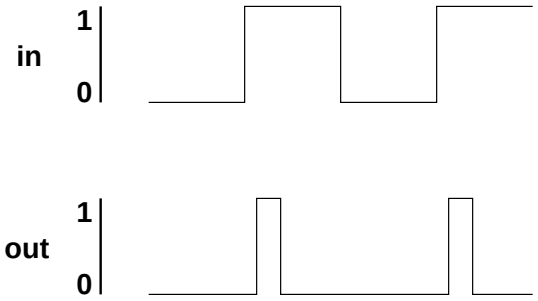
But think about what happens when the input changes from 0 to 1.

When it's 0, the inverter is feeding a 1 and the direct input is feeding 0 to the AND.

Switch the input to 1, and the 1 gets to the AND almost immediately (it's just a wire) but it takes some time for the inverter to react to its new input (there is some gate delay), so the AND is seeing 2 1's, which causes the output to produce a 1.

But soon (after an inverter's gate delay), the inverter has reacted to its new input and the AND gate gets a 0 from the inverter and the circuit again produces the expected output of 0.

If the input is switched back and forth between 0 to 1 over time, we see this behavior:



The length of time that the output is 1 after the switch of the input from 0 to 1 depends on the gate delay of the inverter.

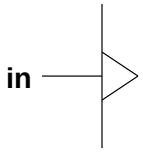
If we wanted it to stay 1 a little longer, we could do that by putting a few (odd number of) inverters in series.

This is a useful device called a *leading edge detector* (LED).

Go back and try the Logisim circuit by stepping the simulation as the input changes.

Why might we want this? It will allow us to send out a one-time pulse right when a signal goes from low to high, even if the signal is going to be high for a while. We'll see applications soon.

On a chip, an LED is denoted:

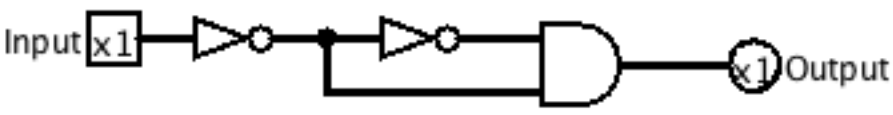


So the gate delay that annoyingly slows down combinational circuits is actually beneficial in this case.

However, if we are building a circuit and this kind of behavior comes up unintentionally, it can be seen as a *glitch* in our circuit.

How about *trailing edge detection* (TED)?

We just invert the input to get one.



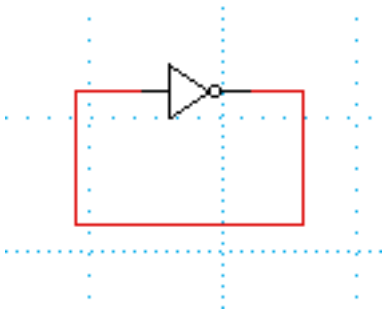
Logisim Circuit:
`~jteresco/shared/cs220/examples/logisim/trailingedgedetector.circ`

Another useful circuit.

These circuits are often a bit weird. We have to consider a circuit's time-dependent behavior to understand it fully. You might wonder if we really want to depend on this kind of thing, but as we go forward you will see that we certainly do.

Clocks

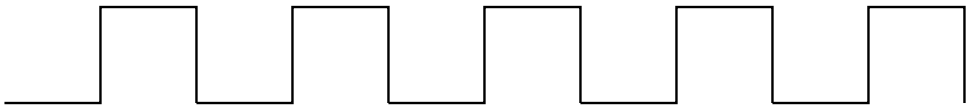
Now consider this circuit:



The electrons can go around the wire from the output back to the input very quickly – at some significant fraction of the speed of light.

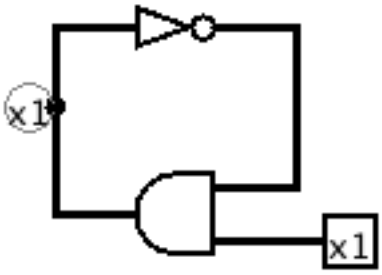
However, the gate delay of the inverter will take a relatively long time.

So what happens? If we look at the logic level on the wire, we will see:



We have a *clock* – an oscillating circuit that cycles back and forth between 0 and 1 at a fixed rate.

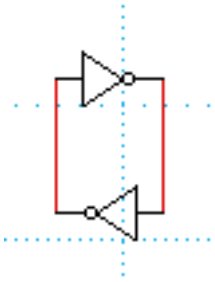
We can try this in Logisim:



Logisim Circuit:
`~jteresco/shared/cs220/examples/logisim/simpleclock.circ`

Note that you will need to step the simulation, and change the input of the AND gate that was added here to get things started.

How about this one?



This has two possible states. 0 on the left, 1 on the right, or 1 on the left, 0 on the right. But once in one of those states, it will stay there. This is a *bistable* circuit.

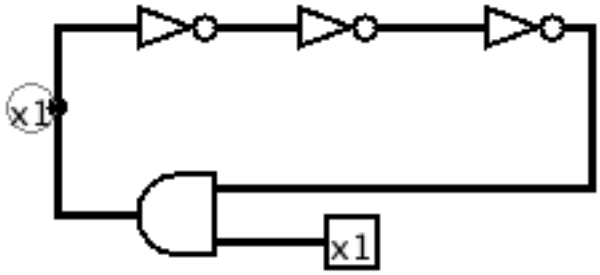
How do we get it into one of these states? If we could force a 1 on one side for just an instant, that side gets 1, the other gets 0. Or, if we force a 0 on (ground) one side for a bit, that side gets 0, the other gets 1.

So after a short delay, it remembers the value we "put on."

This is the simplest *latch* device. More on this in a minute.

For now, let's think about what happens if we add a third inverter.

We get a clock with a period three times longer than the original, since there are three gate delays instead of one.



Logisim Circuit:
`~jteresco/shared/cs220/examples/logisim/slowerclock.circ`

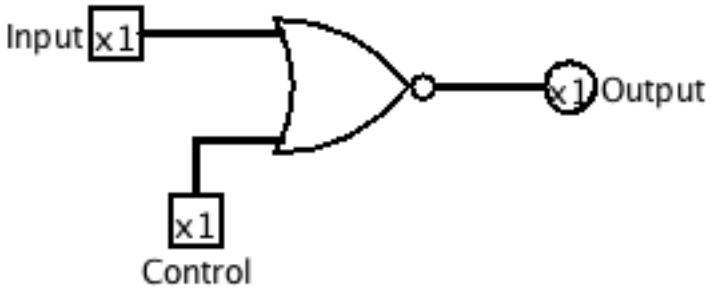
This is a cheap, quick way to build a clock.

Latches and Flip-Flops

S-R Latches and Flip-Flops

We will build on the idea of the bistable circuit to construct circuits that can remember values – the building blocks for circuits that can be used as registers and memory.

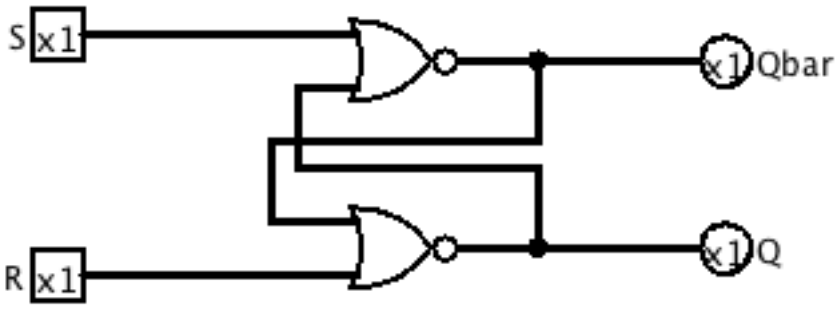
First notice that a NOR gate can be set up as a "controllable inverter":



Logisim Circuit:
`~jteresco/shared/cs220/examples/logisim/norcontrol.circ`

- If the control is 0, this behaves like an inverter.
- When control is 1, the output is always 0.

With this, we can build an *S-R Latch*:



Logisim Circuit: `~jteresco/shared/cs220/examples/logisim/srlatch.circ`

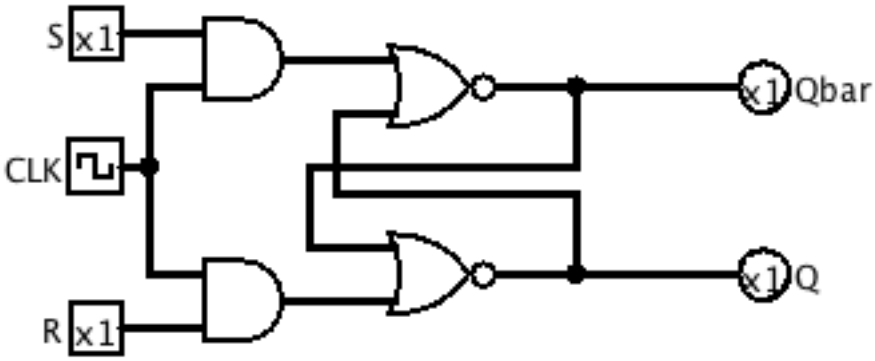
- At any given time, it has a stable value when S and R are 0.
- If R is presented a 1, it will make $Q = 0$, $\overline{Q} = 1$. When R is returned to 0, it maintains that value.
- S is the mirror image, and setting S to 1, Q becomes 1 and $\overline{Q}$ becomes 0. When S is 0 again, the value remains.

They are so named because S is a "set" and R is a "reset".

There is no extra cost to getting both Q and $\overline{Q}$ out of the latch, which is convenient when this is feeding a circuit that may want an input and its inverse both available. We may be able to save an inverter elsewhere.

It is also possible to build an S-R Latch from NAND gates (think about how – you'll have to do it in lab).

We can augment our S-R Latch to change input only when a clock signal is high (to make sure we only set or reset when we really mean to do so):



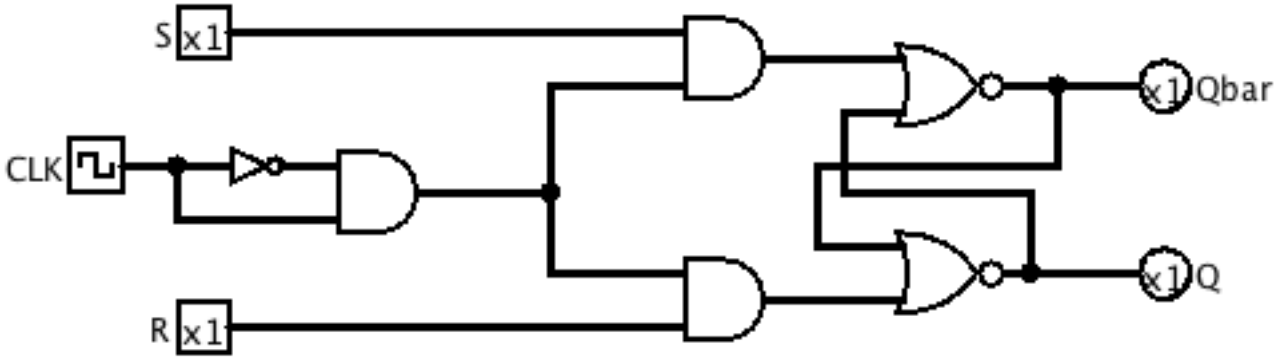
Logisim Circuit:
`~jteresco/shared/cs220/examples/logisim/clockedsrlatch.circ`

This is a *Clocked S-R Latch.*

It changes value only when the clock is high.

But we may want to be even more restrictive and have the S and R lines interpreted only on the rising edge of the clock (the brief moment when it switches from 0 to 1).

We do this by inserting a leading edge detector after the clock.



Logisim Circuit:
`~jteresco/shared/cs220/examples/logisim/clockedsrflipflop.circ`

This is a *Clocked S-R Flip-Flop*.

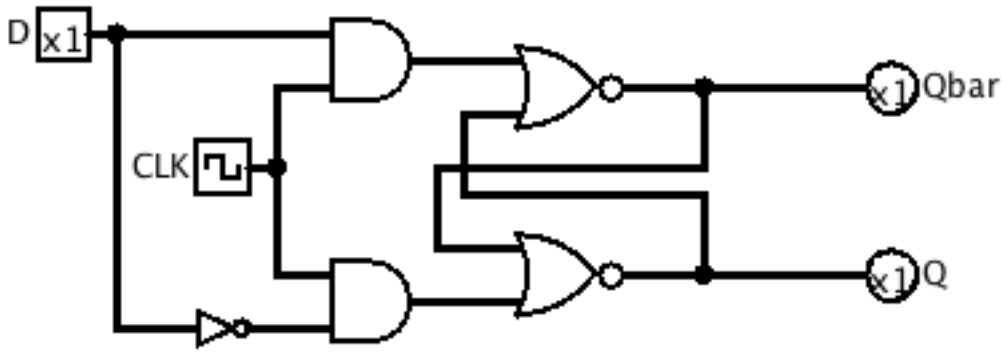
It is a provided building block in Logisim.

In all of these S-R latches and flip-flops, what happens if we have both S and R high at the same time?

Both Q and $\overline{Q}$ will be 0 when those inputs are being presented. When the inputs both go back to 0, a race condition will occur and the circuit will fall into one state or the other.

D-Type Flip-Flops

The race condition is not a desirable feature. An alternative:

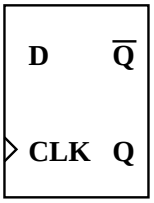


Logisim Circuit:
`~jteresco/shared/cs220/examples/logisim/dtypeflipflop.circ`

This is the *Clocked D-type Flip-Flop*.

We present the desired output onto D and it makes sure we feed in appropriate values to the S and R parts of our circuit.

The symbol for the D-type Flip-Flop:



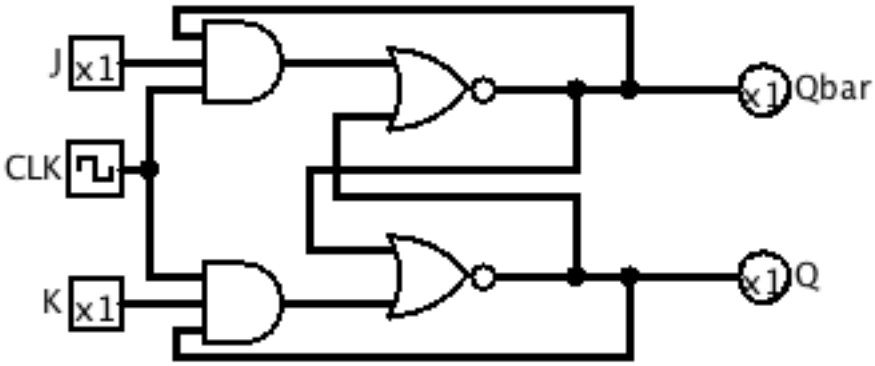
These are the main component of SRAM (typically used in L2 cache).

Since one of D and $\overline{D}$ must be 1, every leading edge attempts to set the state of the circuit.

This means that the input D must be present for every clock cycle. This is OK, as long as you only send a clock pulse when there is an appropriate value on D.

J-K Flip-Flops

Consider this approach, which augments the S-R Latch:



Logisim Circuit: `~jteresco/shared/cs220/examples/logisim/jklatch.circ`

This is a J-K Latch.

The feedback from Q and $\overline{Q}$:

1. Allows input J to pass through to S if Q is low and the clock goes high, and
2. allows input K to pass through to R if Q is high and the clock goes high.

So if Q is set, it allows a reset. If $\overline{Q}$ is set, it allows a set.

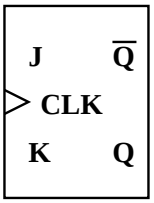
So what happens if both J and K are set when the clock goes high?

Toggle!

So this makes more sense as a flip-flop, where the CLK input is being provided by a LED. The inputs J and K are sampled briefly on the leading edge of the clock.

Otherwise, the J and K both 1 case will lead to continued toggling.

Here's our symbol for the J-K Flip-Flop:



That `>CLK` means an leading edge-detected input named CLK.

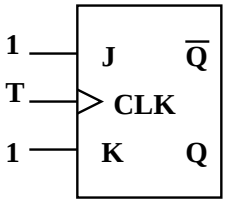
We have a very flexible device that allows 4 possibilities when the CLK input goes high:

1. $J = 0$, $K = 0$: Q remains unchanged
2. $J = 1$, $K = 0$: set Q to 1

3. $J = 0$, $K = 1$: set Q to 0
4. $J = 1$, $K = 1$: toggle Q

T-type Flip-Flops

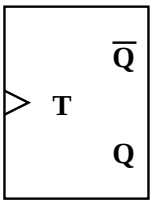
So suppose we connect it up like this:



The input T is so-named because it will toggle the outputs.

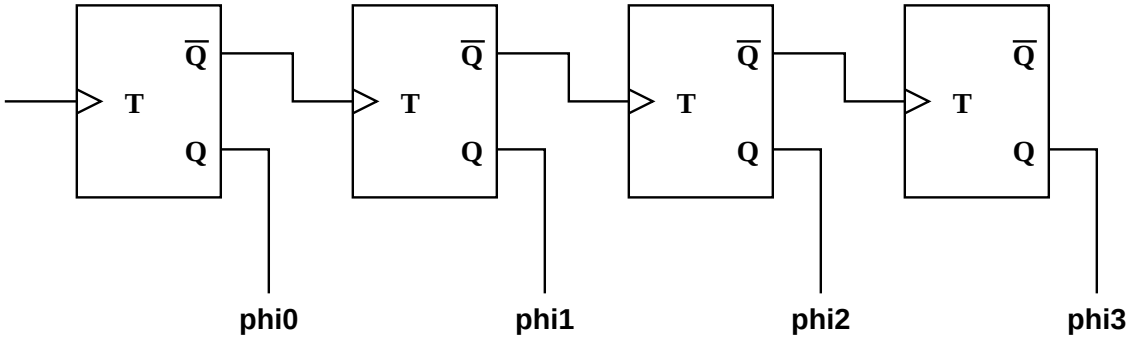
This is a T-type flip-flop.

We could build these out of J-K flip-flops, or reduce the number of inputs to the AND gates.



Counters

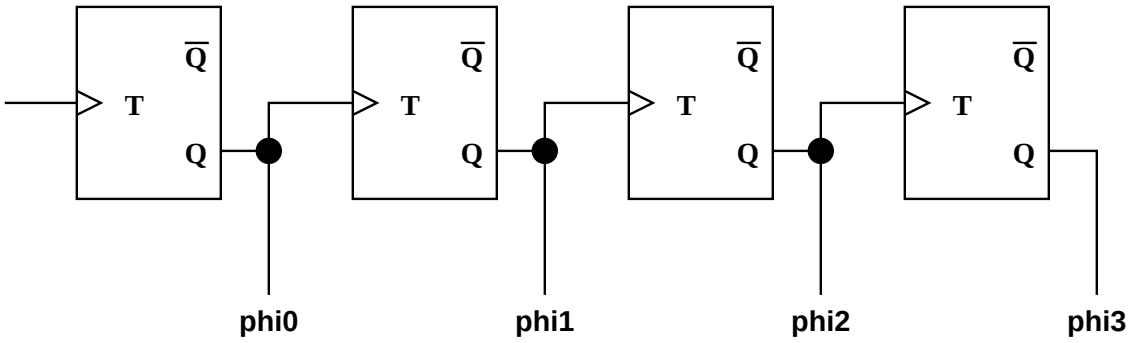
Here's one way to use T-type flip-flops:



The output is a 4-bit counter!

Up/Down Counters

What if we connect up output Q instead of $\overline{Q}$ to the subsequent clocks?



Everything changes on the leading edge.

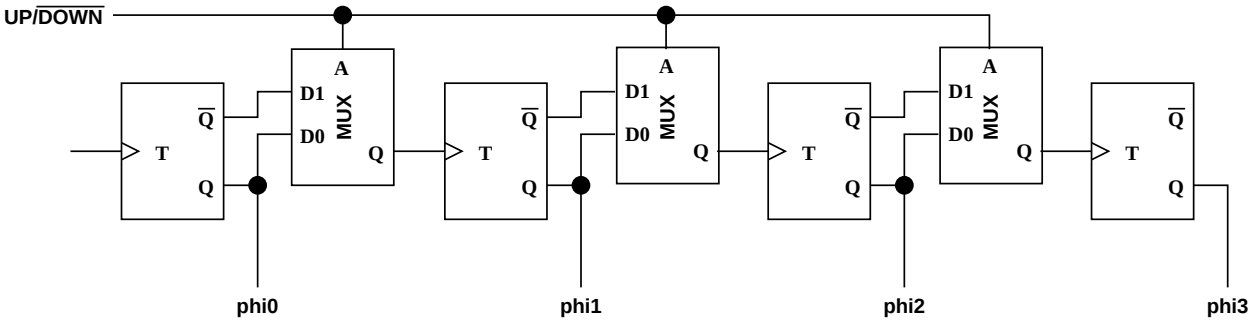
So we have a countdown device!

We can actually take our counter and make it a count up/down device by adding another input line called $UP/\overline{DOWN}$.

We pass along $\overline{Q}$ if we have a 1 on this line, pass along Q if we have a 0.

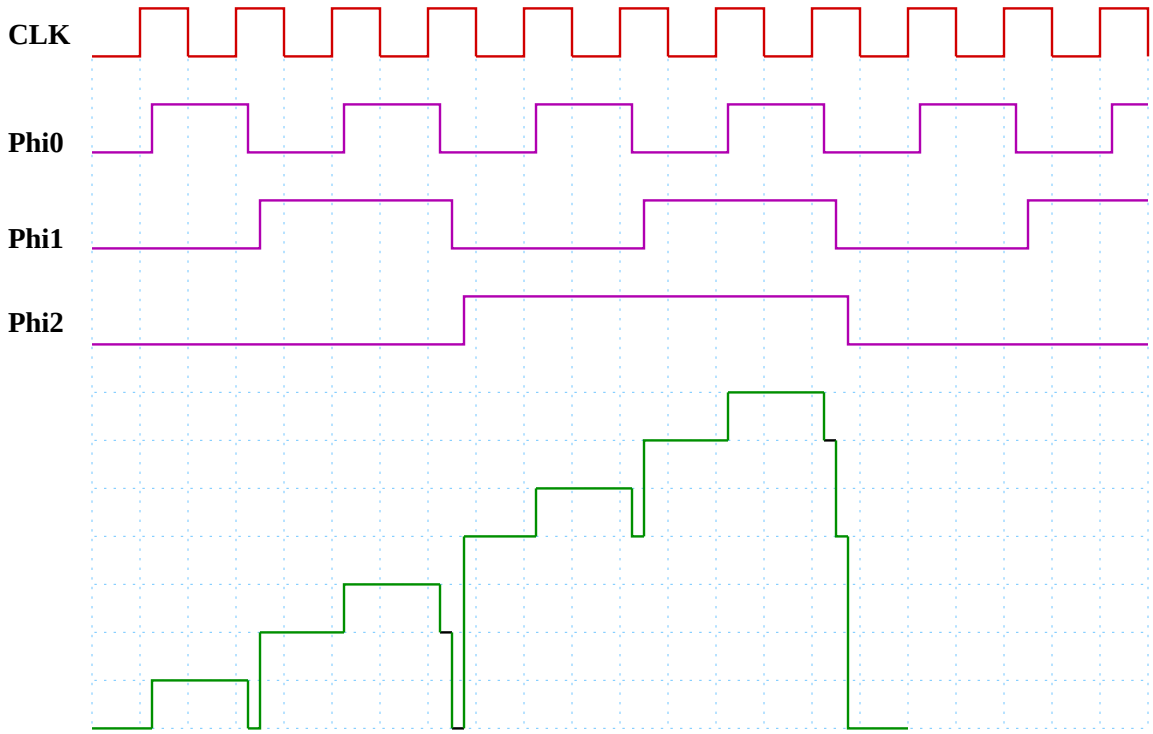
Insert a 2-way MUX:

(($\overline{Q}$ AND $UP/\overline{DOWN}$) OR (Q AND $UP/\overline{DOWN}$))



Synchronous Counters

But let's look carefully at the timing of this.



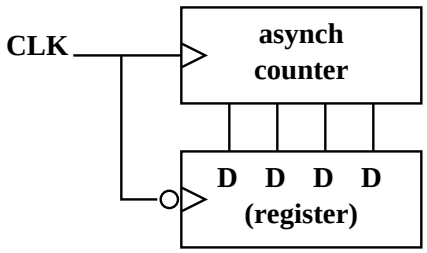
There is really a short gate delay period before each output “digit” updates in response to a rising edge.

This could be very bad if we’re waiting for a particular value (maybe 0) to come up, and we see it too soon.

This “skew” grows as the number of bits in the counter grows.

So this is called an *asynchronous counter*.

To fix this, we can feed our output of the aynchronous counter into a register (a bunch of D flip-flops):



The values can come out of the top counter asynchronously, but we don’t put them into our register until the clock goes back down.

The asynchronous counter is triggered on the leading edge, while the register is triggered on the trailing edge.

This whole thing is a *synchronous counter*.

Something to think about: we can easily count up to powers of 2, but what if we want to count in base 10?