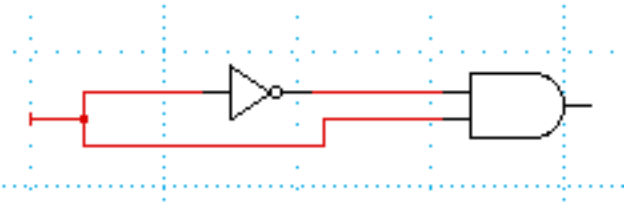
Computer Science 324
Computer Architecture
Mount Holyoke College
Fall 2009

Topic Notes: Sequential Circuits

Let's think about how life can be bad for a circuit.

Edge Detection

Consider this one:



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

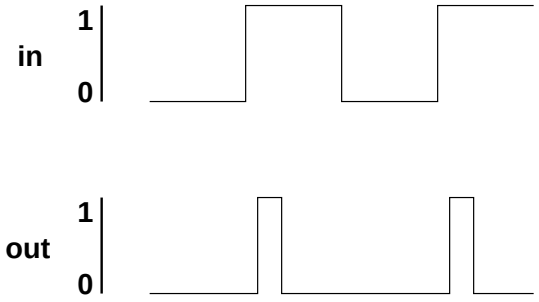
But think about what happens when the input changes from 0 to 1.

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 right away, it's just a wire, but it takes some time for the inverter to react to its new input (gate delay), so the AND is seeing 2 1's for a brief time, and produces a 1.

But soon after, the inverter does its thing and the AND gate gets a 0 from the inverter and things go back to the expected output of 0.

If the input is switched 0 to 1 and back over time, here's what happens:

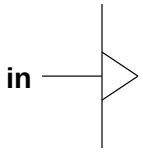


The length of time that the output is 1 after the switch of the input from 0 to 1 depends on the gate delay of the inverter.

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

We have built a *leading edge detector* (LED).

On a chip, an LED is denoted:



So this gate delay that seemed mostly like an annoyance when we were computing something with a circuit might actually be beneficial.

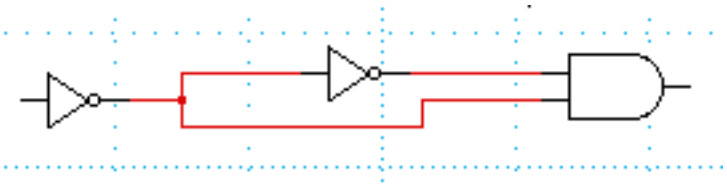
However, if we are building a circuit and this kind of behavior comes up unintentionally, it can be seen as a *glitch* in our circuit.

Why might we want this?

If we want a one-time pulse out of a signal that is going to be high for a while. We'll see applications soon.

How about *trailing edge detection* (TED)?

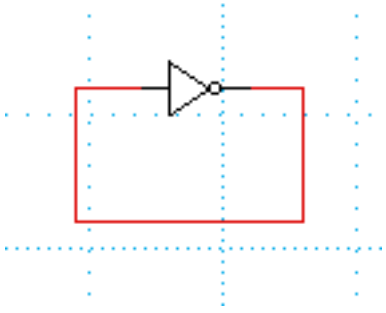
Can we just invert the input to get one?



This seems kind of weird. Do we really want to depend on this kind of thing?

Clocks

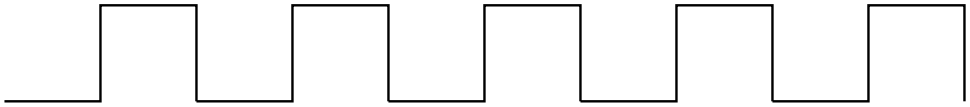
Now consider this circuit:



The electrons can go around the wire from the output back to the input very quickly – something around half the speed of light, maybe more.

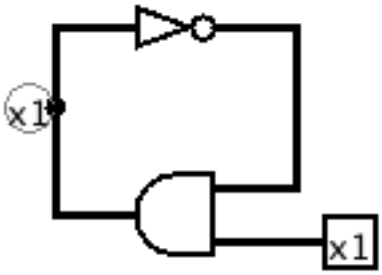
However, the gate delay of the inverter will take some time.

So what happens?



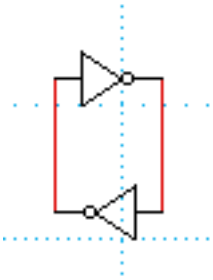
A *clock*! An oscillating circuit that cycles back and forth between 0 and 1 at a fixed rate.

We can hook this up in Logisim:



Logisim Circuit:
`/home/jteresco/shared/cs324/examples/logisim/simpleclock.circ`

How about this one?



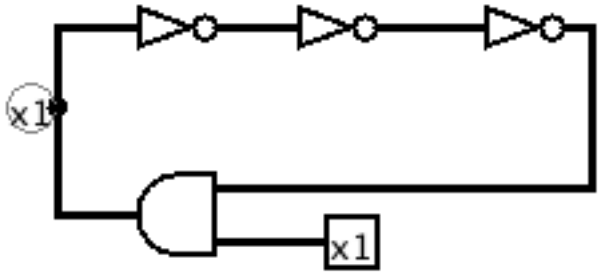
This has two possible states. 0 on the left, 1 on the right, or 1 on the left, 0 on the right. This is a *bistable* circuit.

If we put a 1 on one side for a bit, that side gets 1, the other gets 0. If we ground one side for a bit, that side gets 0, the other gets 1.

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

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

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:
`/home/jteresco/shared/cs324/examples/logisim/slowerclock.circ`

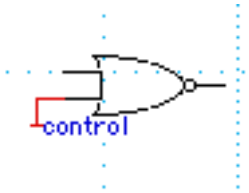
This is a cheap, quick way to build a clock.

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.

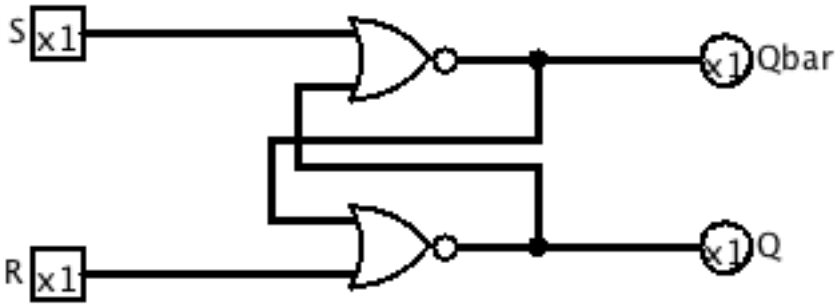
First notice that a NOR gate can be set up as a "controllable inverter":



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:
`/home/jteresco/shared/cs324/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.

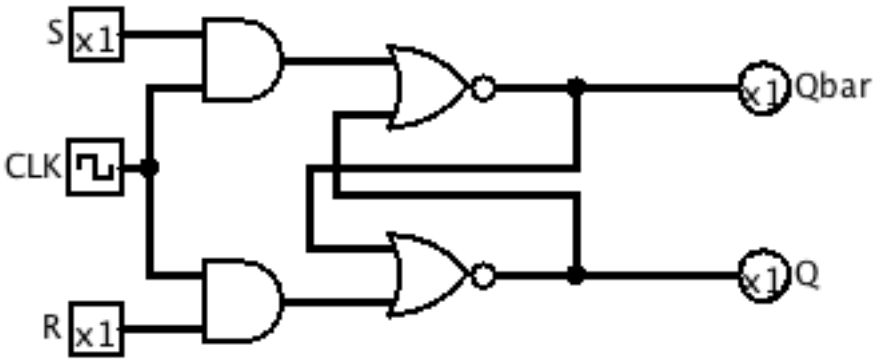
When S is 0 again, the value remains.

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 can save an inverter.

It is also possible to build an S-R Latch from NAND gates (think about how – you'll 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):



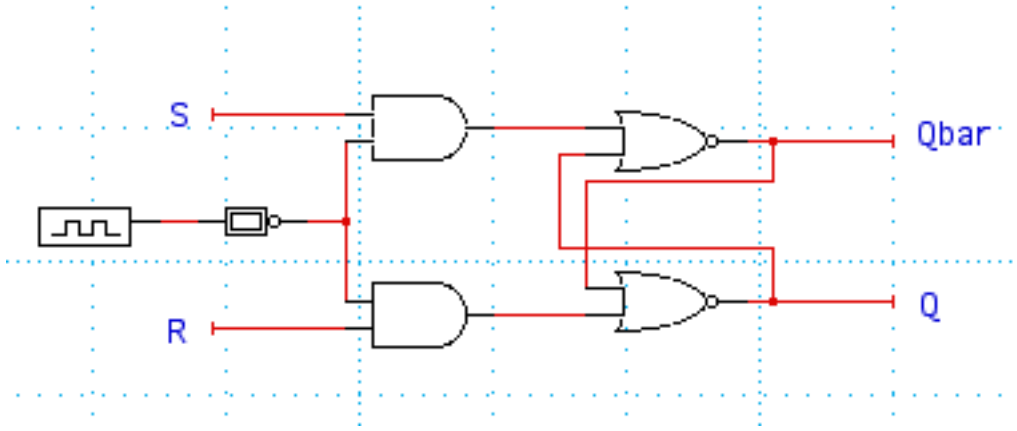
Logisim Circuit:
`/home/jteresco/shared/cs324/examples/logisim/clockedsrlatch.circ`

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.



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

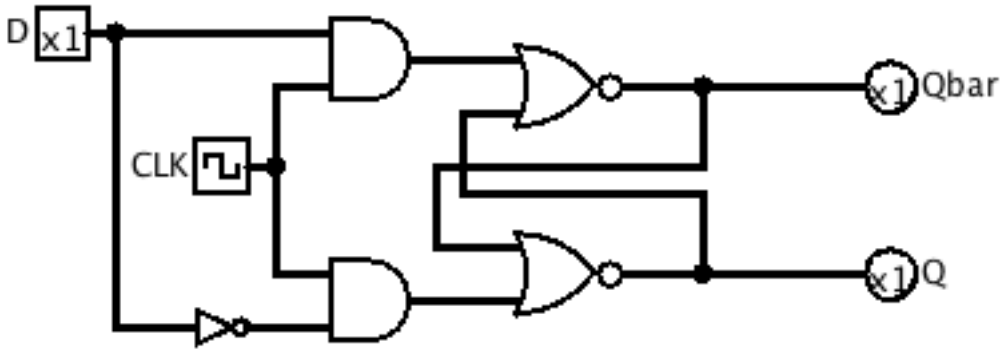
It is a provided building block in Logisim.

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:

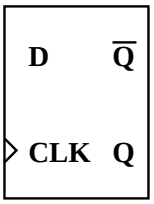


Logisim Circuit:
`/home/jteresco/shared/cs324/examples/logisim/dtypeflipflop.circ`

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:



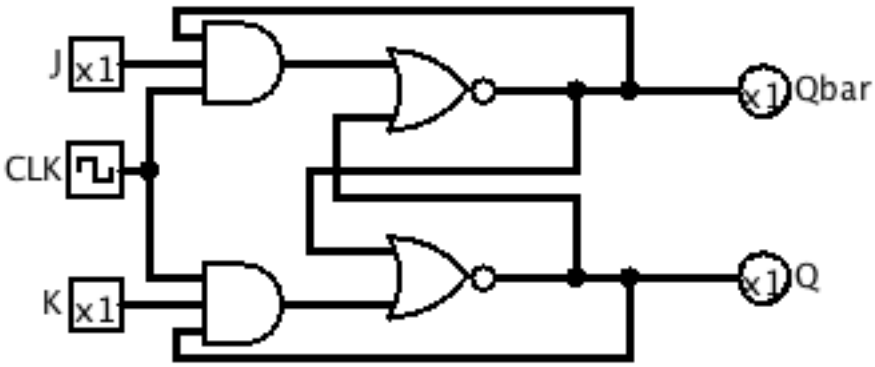
These are the main component of SRAM (typically used in L2 cache).

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:
`/home/jteresco/shared/cs324/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.

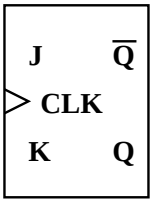
So what happens if both J and K are set when the clock goes high?

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:



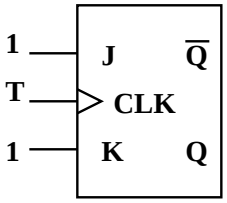
That `>CLK` means an leading edge-detected input named CLK.

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

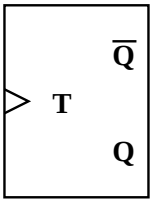
So suppose we connect it up like this:



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

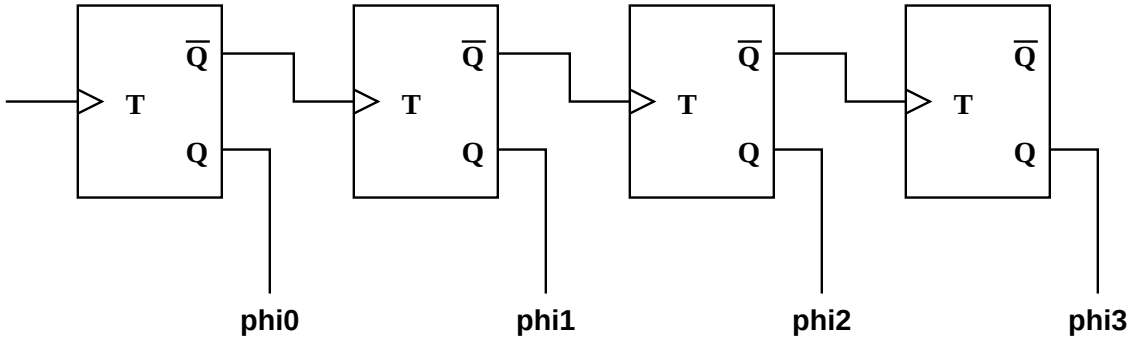
This is a T-type flip-flop.

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



Counters

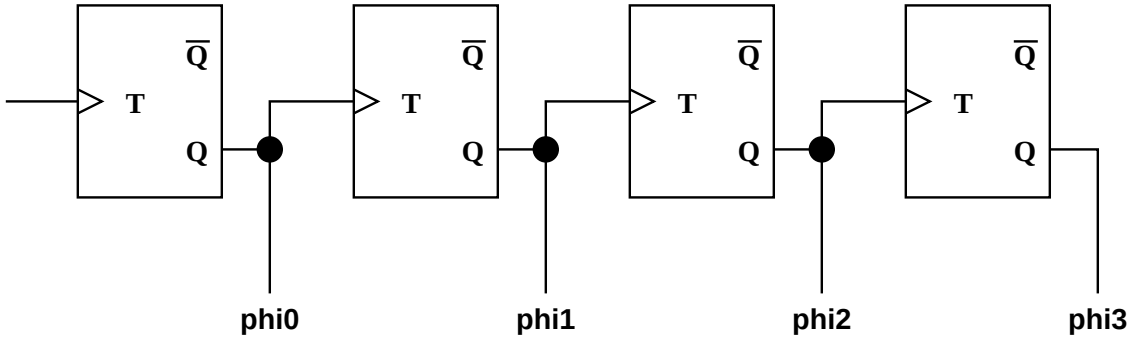
Here's one way to use T-type flip-flops:



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.

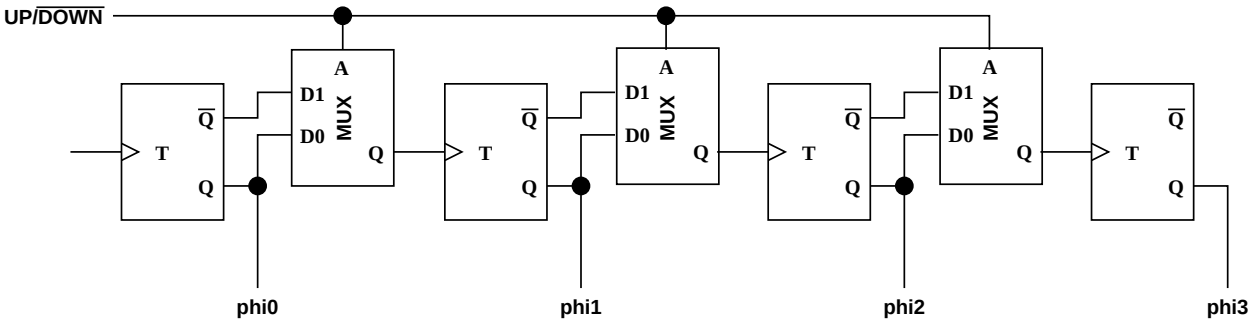
So we have a countdown device!

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.

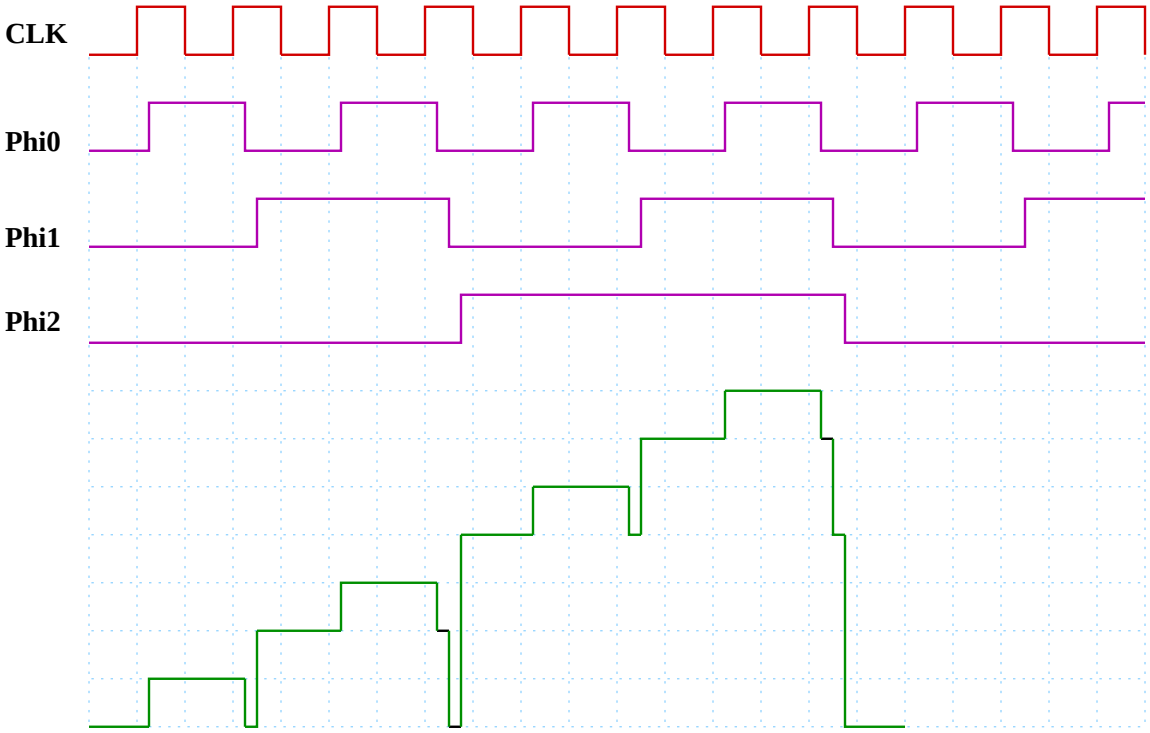
Insert a 2-way MUX:

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



Synchronous Counters

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



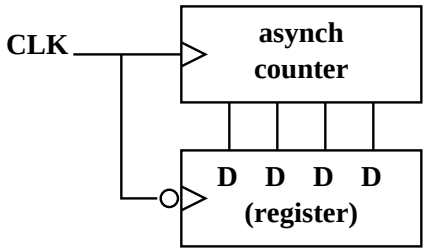
There is really a short gate delay period before each output "digit" updates in response to a rising edge.

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?