Flipping Some Flops and Counting

Lab 02B

6.S188 IAP 2026

We previously built some logic circuits that only depended on current inputs. They were that one "cool" friend who just lived in the moment, no consideration of the past... "don't worry, they said. $Y = A + B$! Let's go party." There's a freedom in that. No baggage. Ugh what it must be like to be 21 again... I'm telling you.

Remembering Stuff

Unfortunately that friend usually ends up having a lot of issues and borrowing money from you which you lend to them knowing damn well in your heart you'll never see again. That's no way to go through life. You can't *just* live in the moment... You need to make decisions in the larger context of history and most productive things actually end up using both current inputs and *past* information. In order to *use* past information, we need to have access to it... which means we need something to take its output and feed it back to its input so that we can hold on to it until we're done with it (whenever that may be).

SR Latch

Believe it or not, this was happening in lab 1a when you were making that sweet Set-Reset latch with just relays. You can do the same thing with logic gates like so using just two NAND gates and two inverters (note we're going to build up to stateful logic in a slightly different way than was done in lecture to make it easier to build, if you choose to do that).

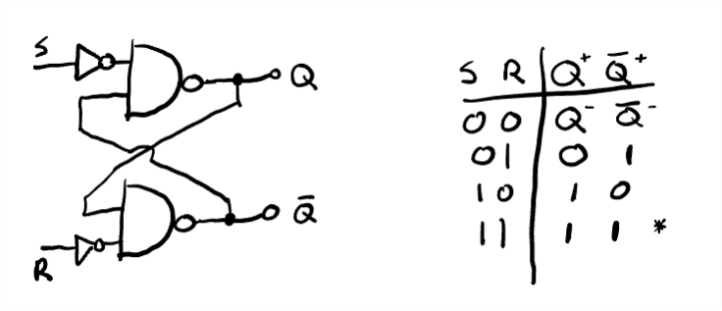

Figure 1: SR Latch out of two NAND gates. *The $1, 1$ input is usually considered invalid and/or "not allowed" since the resulting output $1, 1$ is no longer a complementary binary pair that Q and $\overline{Q}$ should have... consequently this input pair is usually ignored.

This Truth Table has a new type of notation in it, a superscript $+$ or $-$, indicating "future" and "past/current", respectively. We need to do this because the SR-latch is a *stateful* system. The output of the truth table is Q^+ and $\overline{Q}^+$ which are the future values of the two outputs of the SR-latch. In the case of this latch, input S, R having a value of $1, 0$, will make the outputs go to $1, 0$ or $Q^+ \to 1$ and $\overline{Q}^+ \to 0$, respectively (the "setting" of the latch.). If input S, R has value $0, 1$, the latch goes to $0, 1$ on its output or $Q^+ \to 0$ and $\overline{Q}^+ \to 1$ (the "resetting" of the latch). If the input S, R is $0, 0$, the latch stays in whatever state it was previously, or put another way: $Q^+ \to Q^-$ and $\overline{Q}^+ \to \overline{Q}^-$.

(Yes there is a fourth input pair for the SR-latch with S, R being $1, 1$. In this case, the outputs go to $1, 1$ which is not really useful and unstable and messes with future state, so it is avoided).

Maybe go ahead and build this on the board and see it working when push the buttons associated with S and R; you can hook up an LED (with a current-limiting resistor) to Q to see the output. You can use the 74LS00 chips up in the lab as your NANDs. If you do build it use 1K resistors for your pulldown resistors! The 74LS logic has BJTs at the inputs which pull a non-neglible amount of current (unlike a 74HC series gate) so too high of a pull-down (or pull-up) can cause the high and low voltages to not actually be high and low.

D Latch

The SR latch on its own is not super flexible in its behavior... yes it can "remember", but only in a very particular way. We can make it better by adding another circuit that sets the S/R values for us based on a single input value (D for "Data") and a E signal (for "Enable").

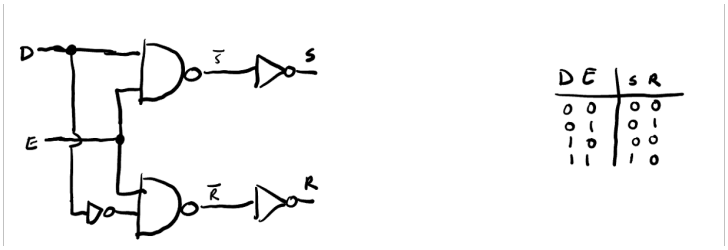

Figure 2: This circuit takes in two inputs, D, and E, and maps those to a set of S, and R values.

Note the two output signals, can be determined to be: $S = \overline{\overline{DE}} = DE$ and $R = \overline{\overline{D}E} = \overline{D}E$. Keep these in mind below.

This logic can actually be simpliifed like shown. The math for this is the following:

From the circuit, $\overline{S} = \overline{DE}$, which can be simplified to $S = DE$ which is the same as before.

For R, we can say: $\overline{R} = \overline{E\left(\overline{DE}\right)}$ since one input is the $\overline{S}$ signal. Using De Morgan's Law we can then say:

$$\overline{R} = \overline{E\left(\overline{D} + \overline{E}\right)}$$

which can distribute out to get:

$$\overline{R} = \overline{E\overline{D} + E\overline{E}}$$

Since $A\overline{A} = 0$ (one of the identities I can't remember... look them up, but it makes intuitive sense when you think about it), this simplifies to:

$$\overline{R} = \overline{E\overline{D}}$$

Which then implies $R = E\overline{D} = \overline{D}E$ which is the same as our original circuit.

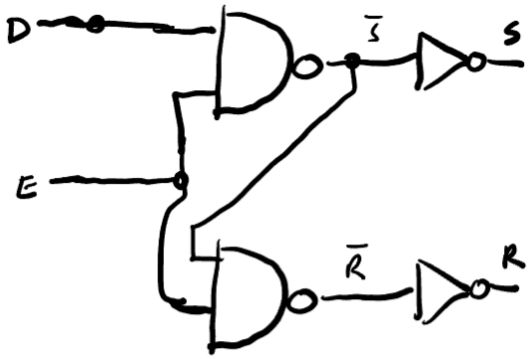

Figure 3: D, E input logic (simplified)

If we then recognize that connecting this circuit to the SR latch from before, we will have four redundant inverters on the S and R signals. That's pretty nicccce, so the circuit simplifies to the following, with its truth table shown on the right.

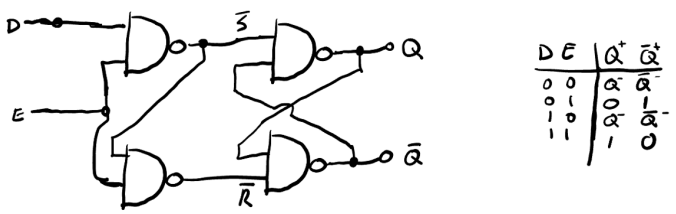

Figure 4: D latch

The D Latch now has memory, but it is more "usable" by most things... taken together, the D latch does the following: If E is high, set Q to be whatever D is (and $\overline{Q}$. If E is low, maintain Q (and $\overline{Q}$) at whatever they were previously.

Like with all things we can abstract it to a box since drawing all the gates repeatedly is tiring.:

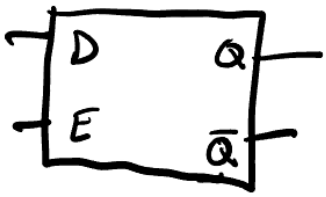

Figure 5: D latch abstracted into its own shape block.

And there you go. Boom Shaka Laka. A D Latch.

The D Latch shown above is very useful since it allows us to store the value of a signal... not a 1 or a 0. I realize this kinda sounds like I'm saying the same thing, but it is a subtle and important difference. In the case of the SR latch, the user specifies directly whether to store a 1 or a 0. In a D-Latch, the user just specifies whether or not to store a *bit* of data. It doesn't matter if the data is a 1 or a 0, you store it all the same... it defers the value that it is storing to some external entity specifying that bit. This is much more useful and much closer to what we want as human freaking beings when designing systems.

If you feel so inclined, go ahead and also build yourself a D latch from NAND gates, and see it behaving as expected. As usual, we're here to help if you want/need.

Flip Flop

But... it turns out that latches still quite aren't the "bee's knees" on their own when it comes to memory... they just don't get used a lot in digital design (on their own). Instead what we usually end up doing is coupling them together into pairs like shown below into something called a "Flip Flop" which also stores data but does so at a particular *instant* in time rather (the rising edge of a control signal). We can make a D-type flip flop by stringing two together with one feeding the other, and using the same signal for controlling both, with the exception that the first one uses the inversion of that signal and the second one using the inversion of the inversion of that signal (or the original enable signal).

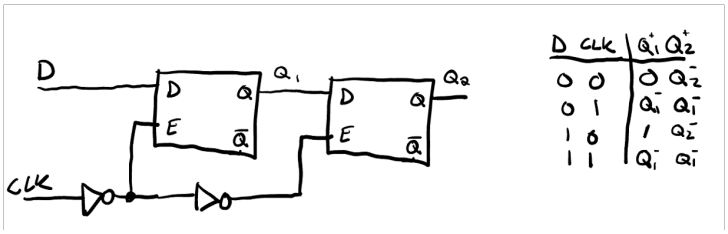

Figure 6: D flip-flop

The truth table for a D Flip Flop is shown on the right side of the figure above and you should try to think through the derivations. Since each latch is itself a stateful entity we need to use the "next" and "current/previous" notation for the

outputs of each latch, which we'll call Q_1 and Q_2. Note the latches still have $\overline{Q_1}$ and $\overline{Q_2}$, I'm just leaving them out here for clarity.

The D Flip Flop passes through values handed in at the input D, but instead of doing so based on the level of an enable signal, it does so using the derivative of that signal (and in particular only the positive derivative when that enable signal goes from 0 to 1). The time when that transition happens is extremely small, arguably instantaneous. When we use signals like this, we usually end up calling them a "clock" signal.

You can do some simplifications in the D Flip Flop truth table to show that basically when CLK is 0, the output of the first latch, Q_1 is D, but the output of the second latch stays at whatever it was previously. If you then change CLK to 1, the first latch "locks" and the second one opens, and uses the output of the first latch as its input which it passes through. In the process the value of D right at the time of the $0 \rightarrow 0$ transition of the CLK signal is all that is passed through and held at the output....it will hold until the next CLK transition from 0 to 1!

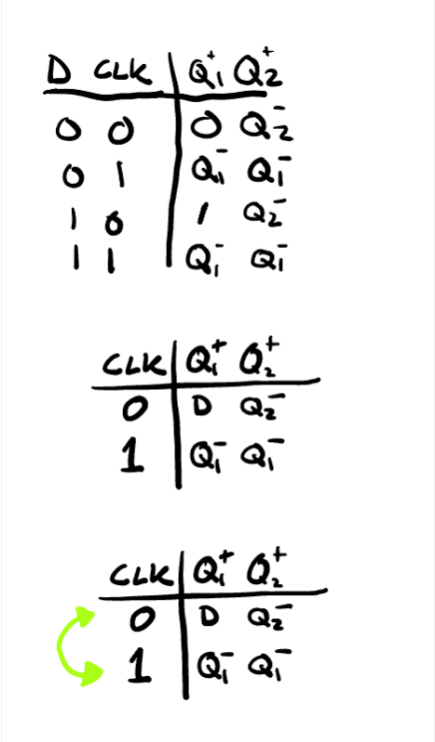

Figure 7: D flop truth table

We can, of course box this up like we've done with all circuits so far. Here I'm drawing the $\overline{Q}$ signal that you get "for free" with latches as well.

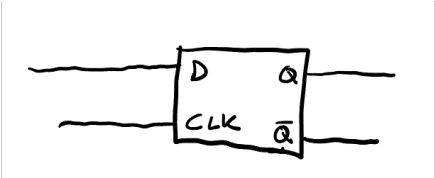

Figure 8: One way to draw a DFF

And you'll often see these drawn with out a explicitely labeled "CLK" signal and instead a triangle symbol:

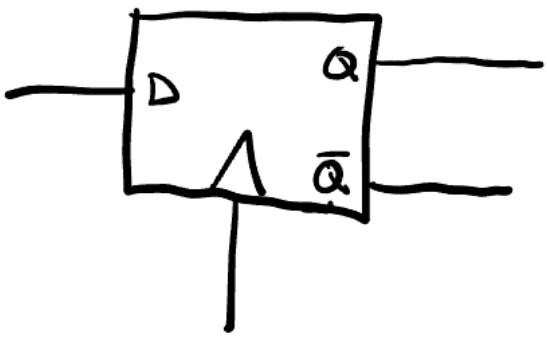

Figure 9: Another way to draw with a more conventional clock input clearly shown

And there you go. D Flip Flops form the core of all modern sequential design. These devices are used by the million or billion in any state-of-the-art chip. It is how things get remembered in many cases (with some exceptions where more dense memory architectures are needed)....just made of two latches and an inverter... which means eight NAND gates and a inverter (which could be a ninth NAND gate if you want). We're not asking you to build this one as part of the lab, but definitely go for it if you want to.

Toggling

One thing you can do with a D Flip Flop is feed its inverted output ($\overline{Q}$) back into its input D like shown below. When you do this, the circuit becomes a "toggler" which will change its output value every time a clock edge comes in. This behavior (and example signal) is shown below:

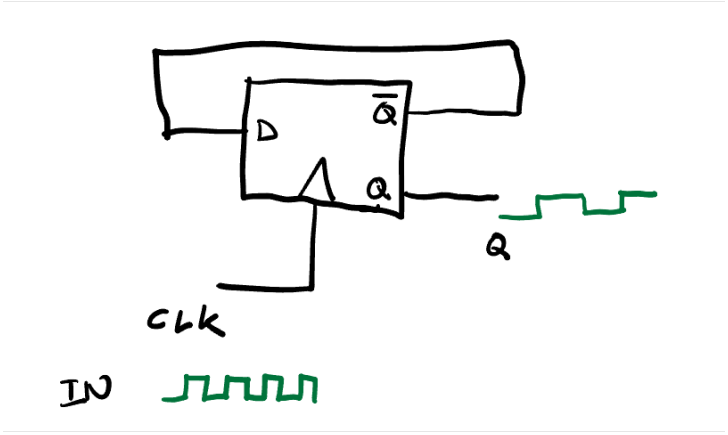

Figure 10: A D FF set up as a toggler

If you have a thing that "toggles" every other time an input signal goes from 0 to 1, in one sense we can say that this is a circuit that is dividing the frequency of an input signal by two. This is a very useful thing to have and you'll sometimes hear it called a "frequency divider" or a "divide-by-two" circuit. Another name for it is a "push-on-push-off" circuit which gets its name from the fact that the first "push" or clock edge sets the output value to some value X and the next identical "push" or clock edge sets the output value to $\overline{X}$ where X is 0 or 1.

When put in the context of numbers, this becomes even more monumental. To see this, let's chain two of these togglers together, using the $\overline{Q}$ output to drive the input of the next toggler like shown.

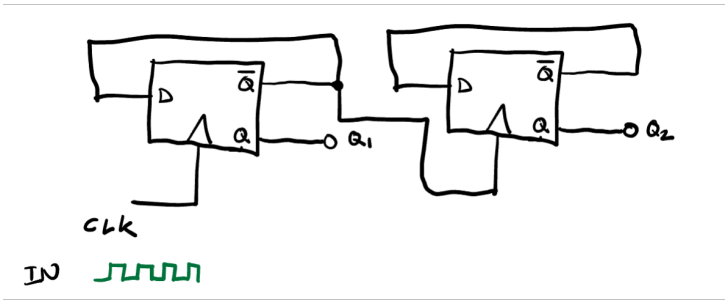

Figure 11: Chaining two toggler's/divide-by-two-counters together

Let's look at the resulting signals that arise from this circuit. Note at the bottom I list out the values of Q_2Q_1 side by side. Analyze them together as one single, multi-bit signal....

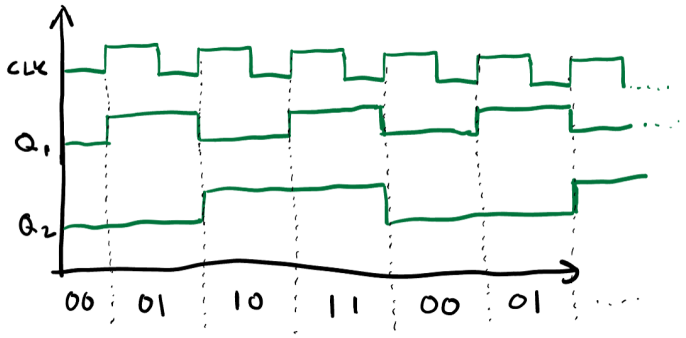

Figure 12: Emergent behavior

Anything interesting?... Nothing?... look closer... Wait a second... Is... is that... oh my god... is that completely unsentient pile of sand and abstract concepts with no soul or moral compass... is that... is that thing counting in base 2?

It is.

The circuit we just made is a ripple counter, a circuit which is capable of counting. Each stage, in the process of toggling at half the frequency of the prior feeding stage, ends up doing exactly what a base-2 counter needs to do! Pretty cool.

Count Up to 16

Let's build our own four-bit counter. Build the circuit below. It'll count the button pressues. When it reaches 15, it wraps back to 0 (overflows).

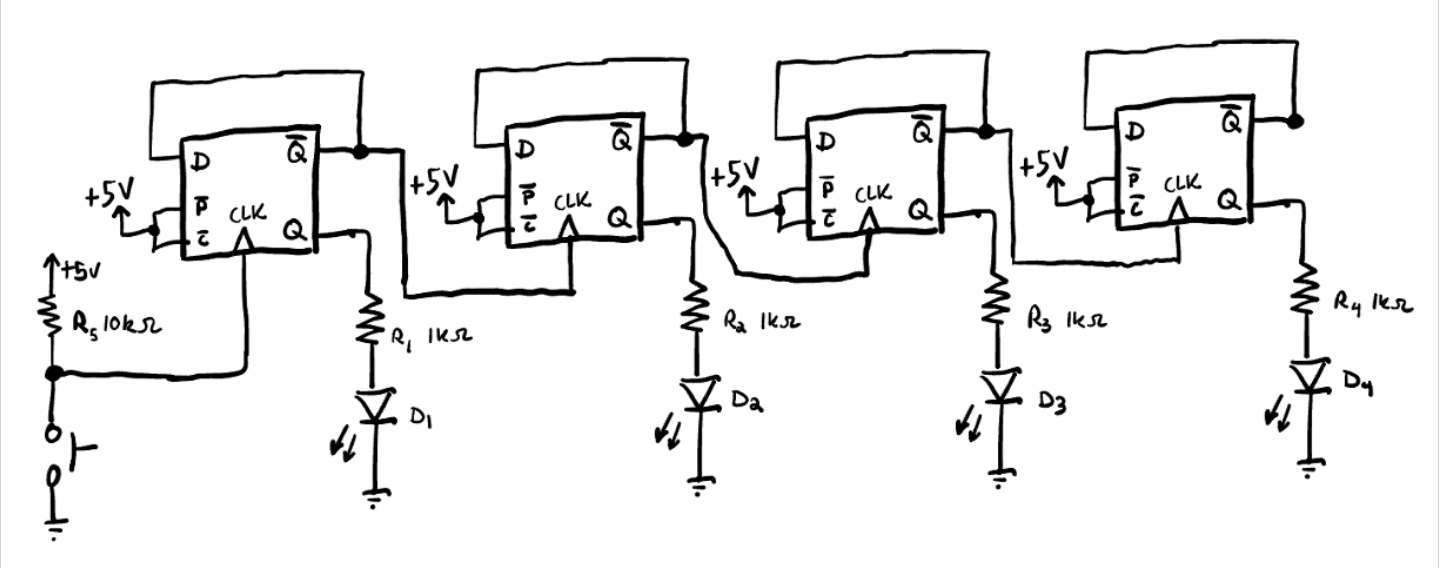

Figure 13: Full counter

How are you going to build this thing? We could make you build all the flip flops yourself. I was in a bad mood this morning so that was the plan, but then I had my coffee and saw some pigeons just living their little lives and had what some might refer to as 'a moment of clarity' realizing in the process that we should use the 74LS74 Dual D-flip flop chip which "does" a bunch of the wiring for you (not all, but a decent chunk). The pinout of the 74LS74 chip is shown below... note these flip flops have a preset and clear signal (both active low) that are like a start at 1 or 0, respectively command.

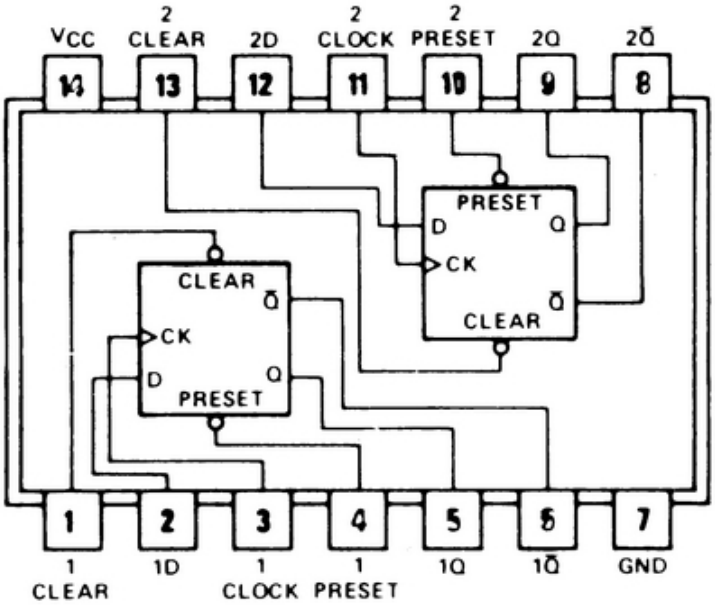

Figure 14: 74LS74 Dual D Flip Flop

So we need four of these flops... which means two of the chips. Grab two chips and a button and wires and resistors and LEDs and build the counter!

When done, you should have a counting device... it should hopefully count, but you may notice it is doing so unreliably.

Bouncy

The circuit is likely counting unreliably because we're dealing with something that hasn't mattered so far... "bouncy" electrical signals.

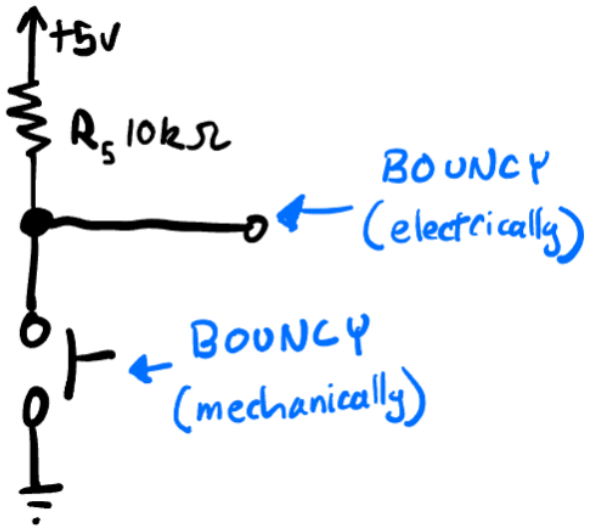

Figure 15: Bouncy

When a switch closes (or opens) there is violent mechanical action happening where metal is hitting metal or metal (in the case of a closing) is hitting plastic (in the case of an opening). These collisions are just like you might see at the human scale... things hit and rebound like a ball on the floor. What this means is that when a switch opens or closes, it may not actually do so cleanly but instead do something like open-close-open-close-open-close. The frequency at which it does this is usually much faster than a human can perceive, but our digital electronics are not humans... they can react to these accidental transitions since their response time is very fast. In some circuits, this might not matter, but in circuits who react to signal derivatives (cough... cough... our D Flip Flops who utilize clock edges), this means for a given button push or release, it might actually see multiple "edges". This is no good. We need to "debounce" the circuit.

This can be done in two steps. The first is to add in an additional resistor and capacitor to produce what is called a low-pass filter circuit. This circuit adds some electrical inertia to signal making the final output far less likely to swing back and forth in response to the bouncing (which happens very fast). Only the low-frequency, long-term state of the switch makes it through and the high-frequency bouncing gets blocked out:

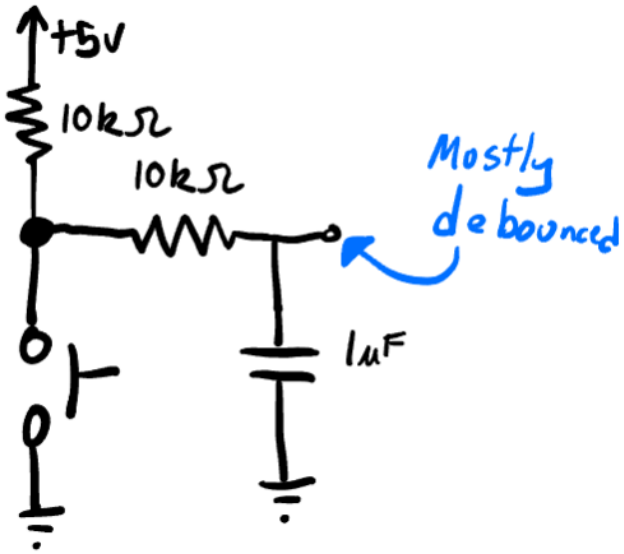

Figure 16: Less Bouncy

This solves part of the problem, but another tricky thing that may happen is that this RC circuit that we have may end up making voltages that aren't very 1-ish or 0-ish. Remember we are using our electronics digitally, so ideally signals are either almost the max voltage (5V) which we call "1" or almost the min voltage (0V) which we call "0". There are some ranges of what is permitted to be a "1" and a "0" with each logic family having different sets. One example range for TTL is shown below:

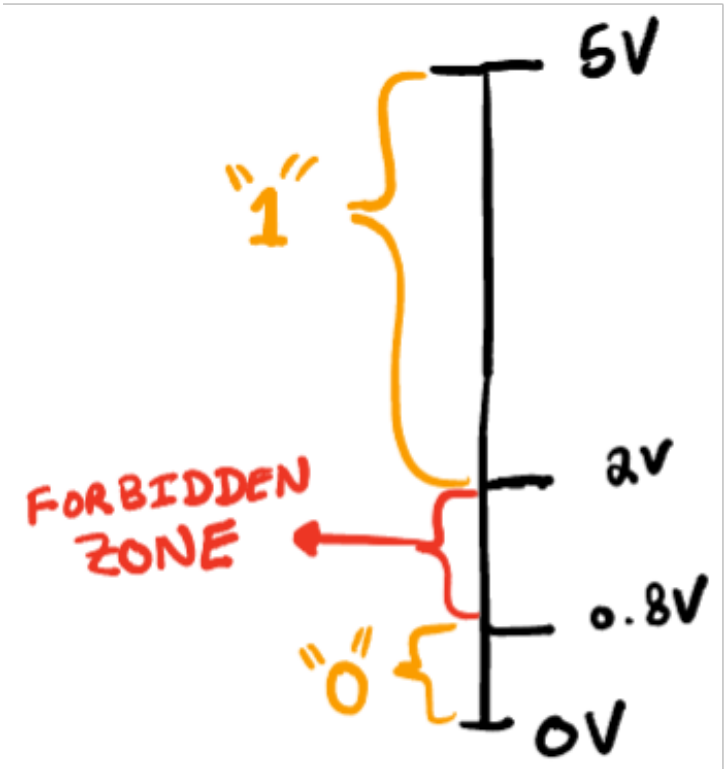

Figure 17: An example set of voltage levels for TTL logic

That's great, but what happens if a signal is in the "Forbidden Zone"? Are you physically not able to be in the Foribdden Zone? Or are you just banned from the Forbidden Zone on quasi-religious grounds like in the original 1968 film *Planet of the Apes* movie?

Well, it turns out you can be in that region... I mean what is going to stop you from doing that?... and if you are in that region, your digital abstraction can break down. If you feed a forbidden voltage into any digital circuit, its behavior becomes unpredictable and that's never good.

The way to protect against this is use a device that is more tolerant of forbidden voltages and generates only allowed voltages. One type of circuit that can do this is a Schmitt circuit, which you'll usually in the shape of a logical inverter..

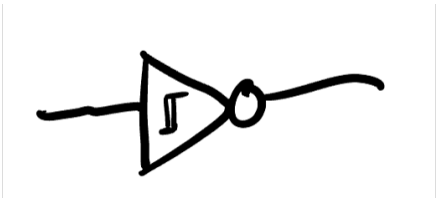

Figure 18: The Schmitt Inverter. It is an Inverter, but has some extra behavior in it

The Schmitt inverter is a logical inverter like any other logical inverter. The big difference is that its internal circuit has some extra stuff added in to make it very, very quickly "escape" the forbidden zone if it ever finds its input in that range. In addition, it has something known as "hysteresis" which means that when its output is transitioning, it cannot quickly go back the way it came. If it is at 0 and it starts to go towards 1, the circuit is physically incapable of stopping half-way (perhaps from a quick change in input) and going back to 0... it has to *fully transition* all the way to 1 before it could ever go back. Similarly. If its output starts at 1 and it starts to transition to 0, but gets input information that says to stop the transition it

can't... it has to go all the way to 0. The analogy for this behavior is that being in the forbidden zone is a very, very unstable state for this circuit to be like the cartoon below shows (ball rolling down a hill).

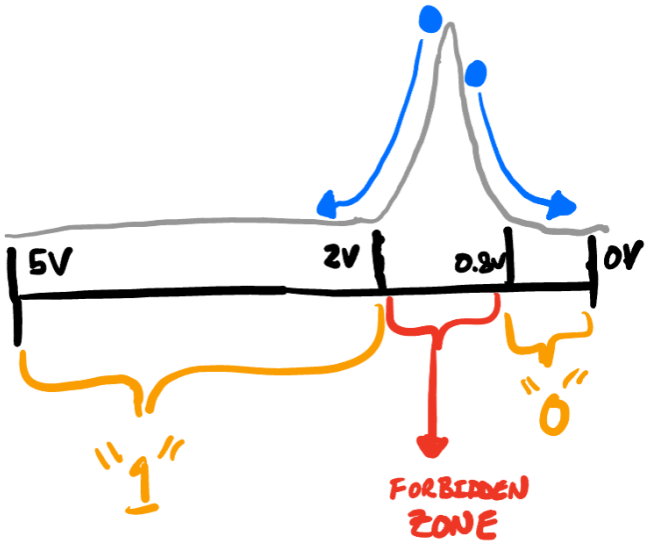

Figure 19: The Schmitt behavior. If you find yourself in the forbidden zone, very very quickly resolve towards a valid output voltage using internal positive feedback and hysteresis

the net result of a circuit like this is that if its input is exposed to a forbidden voltage, it makes sure that anything connects to its output is basically never exposed to that, thus protecting all the other digital electronics downstream. This has the added benefit of giving some extra delay to protect against switch bouncing... since bounces happen very quickly, the Schmitt inverter can add a little immunity to quick back-and-forth signals that might come from bouncing.

There are chips of these Schmitt inverters that we can use... the 74HCT14 is one that we have:

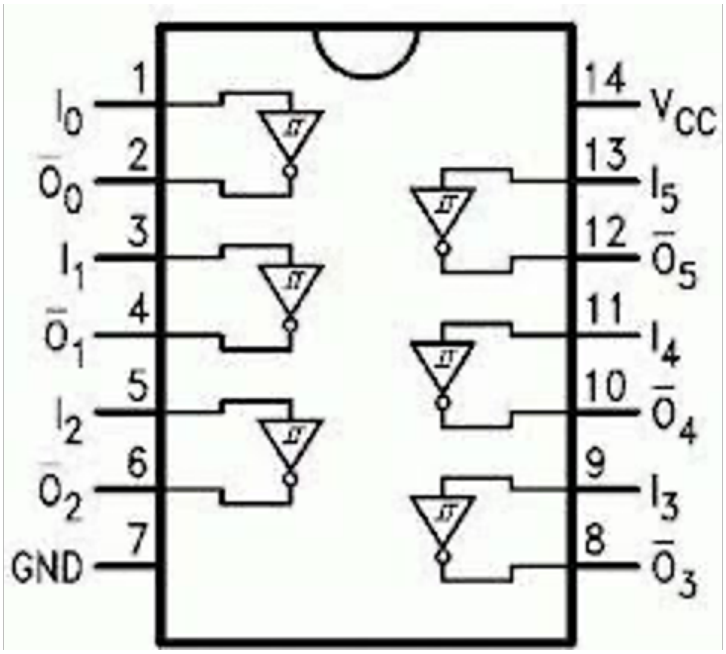

Figure 20: The 74HCT14 has six hex inverting Schmitt triggers in it

Grab one, and incorporate it into your circuit to make a fully debounced button input like shown below:

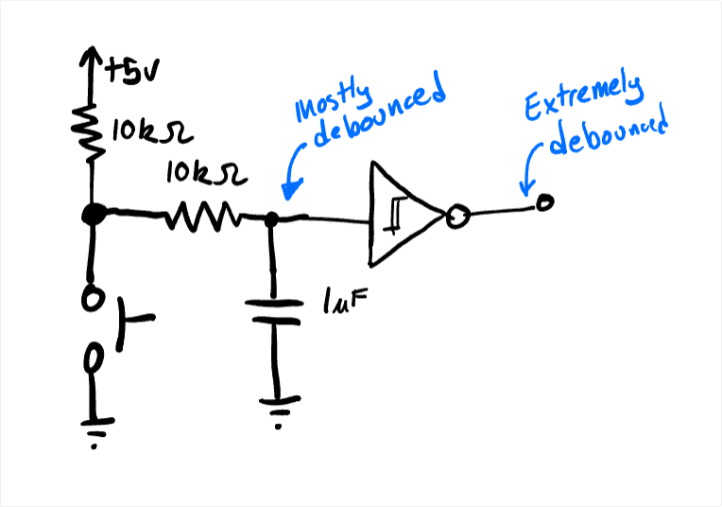

Figure 21: Least Bouncy

Once done, you should now be counting from 0 to 15 reliably.

Count to Your Favorite Number

Counting up to 15 is kinda cool, but it would be much, much cooler if you could count to a different number before rolling over... like maybe you want to count 0 to 5? Or 6? Or a crazy number like 7? We could do that by taking advantage of the reset signal in our flip flops.

Using the combinational logic we have our disposal... use the state of the counter to generate a reset signal (note it is *active low* sothe output will reset to zero when the `CLR` pin is set to 0)... if you want to count "5" you'll actually want your reset signal to be on "6" fyi. (keep the preset signal tied to 1). Make sure to tie all four of the reset signals to this generated output signal. We have a mixture of multi-input gates to make your logic (74x32 OR Gates, 74x08 AND Gates, 74x86 XOR Gates, 74x00 NAND Gates, 74x02 NOR Gates, ...) so feel free to use what you want (but note that the NOR gates have a different pinout than the other ones).

Make sure to pick a non-power-of-2 number to reset at so that you actually have to do some work for it.

When you're done, admire your counter!