Hitting Some Bits (Optional)

Lab 03A

6.S188 IAP 2026

This lab is somehow even more optional than the other labs. We're including it just in case you want to be a cool person. Do you want to be a cool person? It's fine if not...

The Game

Those of you who are still with us and didn't immediately bail out at the mention of the word "optional" are going to build a hardware only version of the famous game "Kill the Bit" which we'll call the more 2026-appropriate-title "hit the bit". This game was one of the little programs included with the documentation for the Altair 8800, one of the first commercial-successful personal computers that helped kick off the home computer revolution in the 1970's, which is pretty sweet.

Shift Registers

As was discussed in lecture 02b, you can hook up your D flops in a shift-register configuration. The term "register" is generally used interchangeably with flip flop in most digital design btw. In a shift register situation, all the flops share a common clock, and each flop feeds its output into the next like a Human Centipede type situation. The passing of "data" from the output of one flop to the input of the other flop happens on each rising clock edge. This is the hardware-based implementation of the `>>` (right-shift) operation in programming languages. In fact... when you do `>>` in a language, you are almost 100% using shift registers behind the scenes.

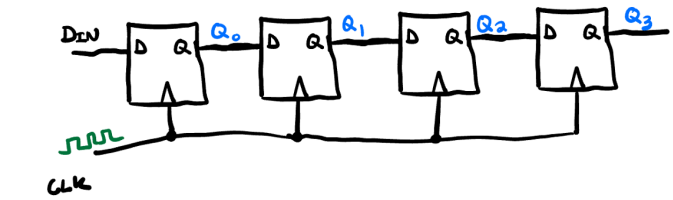

Figure 1: Shift Registers. Data flows from D_{IN} rightwards, moving through one flop on each rising clock edge.

In this case you introduce bits into the shift register on the left and they take four clock cycles to get to the end at which point they fall off the edge like lemmings off a cliff.

A code equivalent of this behavior would be the following:

```
int x = 16;
while(1){
    x = x>>1;
}
```

We can also feed the last bit back around again so the data stays in the system, so doing something like the following piece of code:

```
int x = 8;
while(1){
    x = (x>>1)|((x&0x1)<<3);
}
```

which can be achieved with just a tiny feedback path from the right-most flop's output to the input of the left-most.

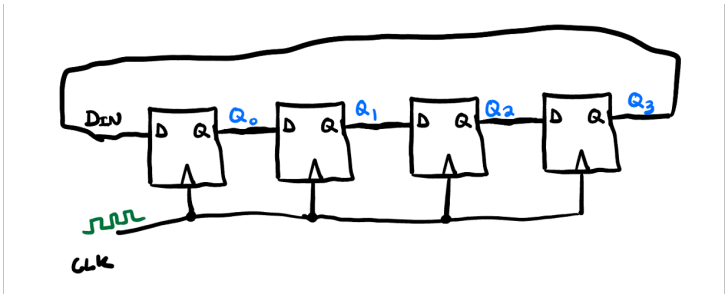

Figure 2: Shift Registers with Feedback

OK, that's fine, but like right now what we have will just start up in some random configuration of 1's and 0's, and then that random sequence of bits will cycle through the flops. That's nice, but we're ultimately going to need to have a way to adjust those values once the thing is running. Here's one thing we can do, so that we can use, say a button, to affect the bits as they flow through. Here we're XOR'ing Q_3 to produce our left-most bit:

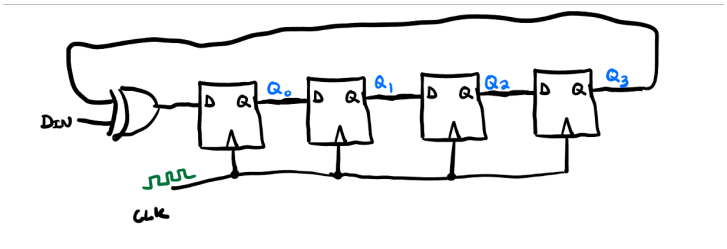

Figure 3: Shift Registers with an XOR to introduce new bits

. . . which is like doing this in code:

```
int x = 8;
while(1){
    x = (x>>1)|(d_in^(x&0x1)<<3);
}
```

So nice. Shift registers are super useful and form the basis of many useful things like cyclic redundancy checks, cryptographic things, many data structures, etc. . .

But can you use them to make something fun? Perhaps a game at the same level of *Silksong*, *The Elder Scrolls*, *Super Mario Galaxy*, *Burger Time*, or *Big Rigs: Over the Road Racing*? No we can't. But we can make *a* game: the *Hit the Bit* game we talked about earlier.

The idea here is going to be to put one of these little XOR button combos on each of the data inputs, so that pushing the button when a bit is on flips it to off, and vice versa. So then ultimately, the goal of the game will be to have things start with some number of the bits on, and then use the little buttons as your input to turn them all off (with the caveat that accidentally clicking on one of the ones that's currently off will turn it on again, which then makes your job even harder). But we'll talk more about that on the next page.

Making the Game

The schematic for the game is shown below (mostly):

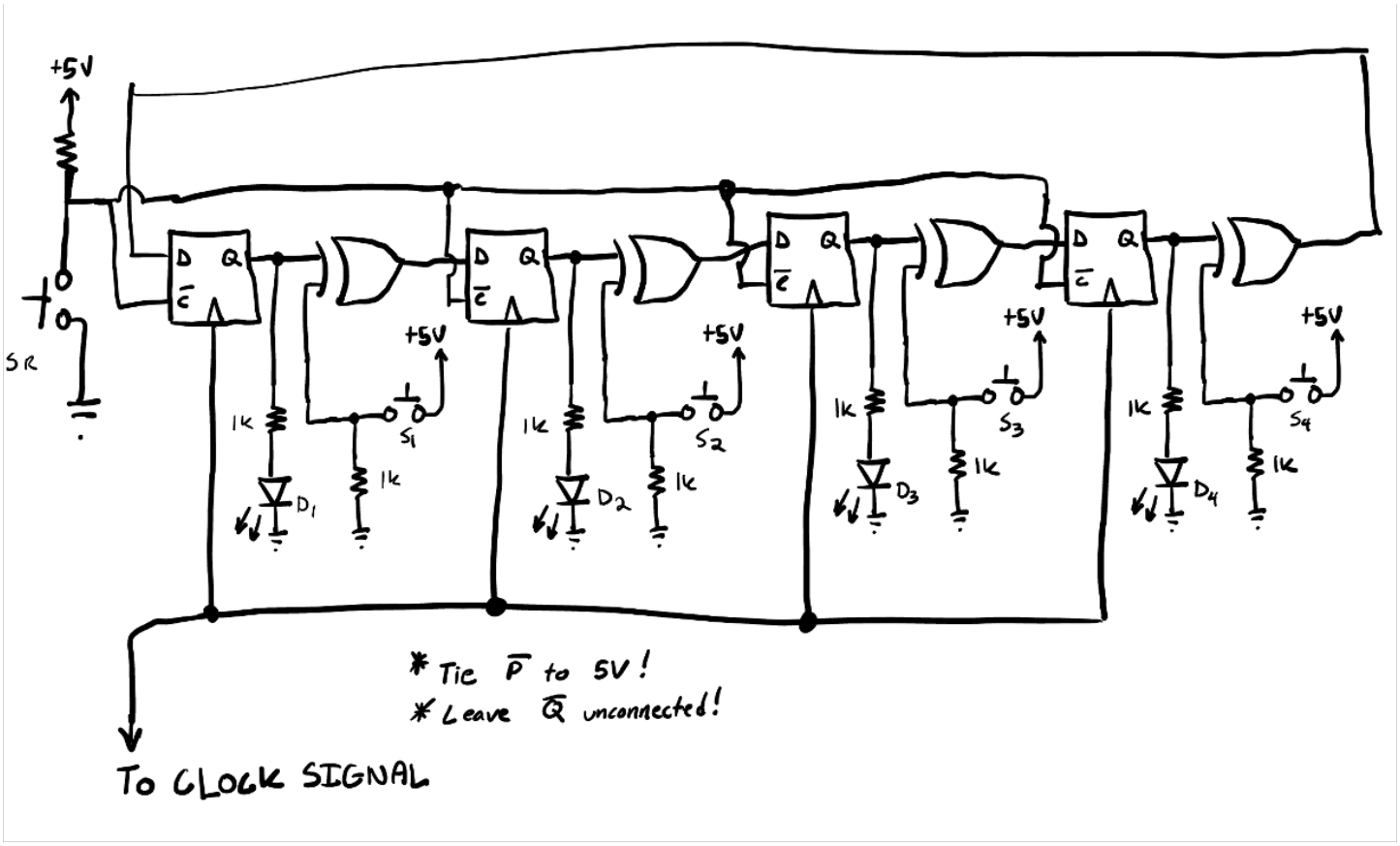

Figure 4: Hit the Bit Schematic

We'll use the same quad XOR as last week (74LS86). But to save on board space, we'll use a different D-flip-flop chip than we used last time. Today we'll use the 74LS175, which has not two but *four* flip-flops per chip. Here's the pinout:

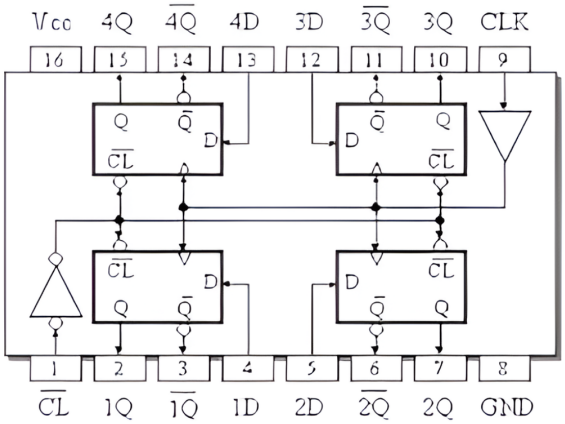

Figure 5: 74LS175 Pinout

The core of our circuit is a four-bit looped shift register as discussed above. Then you have five buttons. One button (active low) is a global reset/clear for all four flops. The other four buttons are game inputs that are set up as active high. These four values are XOR-ed with the value being handed off. Remembering the Boolean expression for XOR is $Y + \overline{A}B + A\overline{B}$, we can see that XOR is high when two signals *are different*. In the context of our cycling shift registers what happens then is at every clock edge, the current bit being handed off is XOR'ed against a button value. If they are the same, a 0 is fed into the next flip flop... if they are different, a 1 is fed into the next flip flop.

You can use this to make a game in conjunction with a clock signal that is outside the player's control. If a set of 1's and 0's currently exist on the four flops, such as 0110, without any user input (buttons all unpushed and therefore inputs 0) they will evolve as follows: $0110 \rightarrow 0011 \rightarrow 1001 \rightarrow 1100 \rightarrow 0110 \rightarrow ....$ If instead you press the button connected to the final flop's input, you'd have: $0110 \rightarrow 0010 \rightarrow 0000$ because you XOR'ed a 1 with a 1 on both clock cycles. You have therefore successfully "hit" the bit(s). If you kept holding that same input on though on the next clock cycle you'd re-introduce a new 1 into the system: $0000 \rightarrow 0001...$ and so on.

For some definition of the word "fun," this is a very "fun" game. Yeah, the seventies were a simpler time.

Make a Clock

OK, so one thing we left off of the last schematic is that the game requires we have a periodic clock signal to drive our system. You could make this with a sixth button that you the user pushes, but that would take away some of the challenge of having to time your button presses with a event controlled by a third party. So let's make a circuit that repeats itself every so often automatically. We'll do that using a very very simple clock circuit using a fedback RC-charging circuit. We talked a bit about this in lecture on Tuesday:

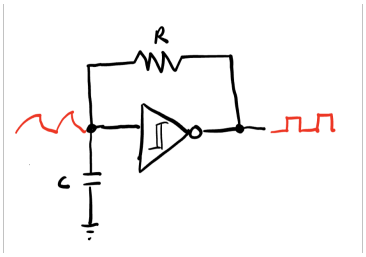

Figure 6: Clock Circuit (simple and crappy)

The frequency is approximately related to $\approx \frac{0.2}{RC}$. That means you can pick your frequency using whatever R and C you can find. Changing these values changes how quickly the little bits march along their line, which is kind of like a way to change the difficulty of the game. So just go ahead and pick/calculate/try some and try it out; once you have that, feel free to adjust the speed by changing R and C until everything is moving at a comfy pace. Maybe try for a couple Hz maximum since it gets kinda hard after that (pro gamers only).

While people who play *Call of Duty* or *Battlefield* games are always bragging about 60 or 120 frames per second (FPS), *Hit the Bit* implemented in 74-series logic could, in principle, have FPS up towards the millions. While that would make the game impossibly hard (not only can we not push buttons that fast, the LEDs wouldn't even light up) and so we shouldn't do it, it's still kind of cool to think about I guess.

Play the Game

You should be all set. You an play the game by:

1. Resetting the system by pushing the reset button and then releasing.
2. At some point, pressing a few buttons to "set" the "puzzle"
3. Have your friend (or you) then try to press the right set of buttons at the right times to clear all the bits. If they fail to do that, they clearly aren't worthy of your friendship.

Release Version 2.0

If you find four bits isn't challenging enough, this game can be extended arbitrarily upwards in size. The fun of the game scales like $\Theta(\log(n))$ where n is the number of flip flops. So hey... if you feel like it... go ahead and add some more bits to the chain.