Combinational Logic

Lab 02A

6.S188 IAP 2026

Higher Purpose

In the previous few labs, we were mostly concerned with making simple logic behaviors out of even simpler components (diodes or transistors or what have you). We told you that if you can make a NAND or a NOR or a latch or whatever that you then have all of the pieces you need to make whatever your heart desires. You believed us and that was that.

Today we'll get some practice doing that, teaching our circuits to add and subtract numbers. It's kind of magical when you think about it, taking those little simple pieces and using them to make a circuit that does something "smart."

Binary Numbers

Most early mechanical adding machines actually did their math in base-10, since the internal gears could exist at more than two positions and everybody's already used to counting/thinking in base 10 (though maybe you could make an argument that we should have stuck with base-60).

At first glance, it seems like digital signals with just high and lows could not represent numbers... I mean you can do 0 and 1, and that's it....but it turns out it is just as suited as any other number system.

"How would one do that?" you might ask... I mean... numbers go 0 to 10, but we can only go 0 to 1 after all. Well what we need you to do is take a step back and think about the **numbers** that we're representing, as separate from their representation. The symbol set 0 to 9 is just one of many ways to represent a deeper underlying concept of numbers...

Representing Numbers

Base 10

In Base 10, each digit position represents a power of 10, starting from the right at 10^0:

```
  3      4      2      7
10^3   10^2   10^1   10^0
1000   100    10     1

Value = (3 x 1000) + (4 x 100) + (2 x 10) + (7 x 1)
       = 3000 + 400 + 20 + 7
      = 3427
```

Each position can hold digits 0-9 (ten possible values). When you count past 9, you carry to the next position.

Base 2

In Base 2, each digit position represents a power of 2 instead:

```
  1     0     1     1
2^3   2^2   2^1   2^0
8     4     2     1

Value = (1 x 8) + (0 x 4) + (1 x 2) + (1 x 1)
      = 8 + 0 + 2 + 1
      = 11 (in decimal)
```

Each position can only hold 0 or 1 (two possible values - hence "binary"). When you count past 1, you carry to the next position.

Brief Aside About Numerical Representation

Numerical respresentation is actually a really intereseting concept to think about. Having a low base, you have to handle fewer symbols per position, but you need more positions to represent a given number. Having a higher base, you have to handle the interpretation of more symbols per position, but you need fewer positions to represent a given number. You may ask what is the "best" base to have since there's pros and cons to both. We generally as humans have settles on base 10 natively due to what has largely been ascribed to the number of digits we have on our two hands, but historically there was also a lot of traction with base 12 number systems (why you'll see weird currency demoninations like one British shilling being 1/20th of a 12-pence) that arose from counting segments on fingers (ignore thumb and you have four three-segmented things on each hand).

Base-2 was really only chosen for digital systems since that was the most conducive to making robust circuits… I mean we have all these sweet/useful building blocks for working with binary inputs, after all. If you do out some equations and make some assumptions about the "cost" of having to represent more and more symbols vs having to write down fewer positions, you can end up with a numerical base of e (yes… the number ≈ 2.718) as being the actual optimal numerical base of representation, though it turns out this is practically unrealizable. After base-e, it actually turns out that base-3 is the most efficient and then base-2 and base-4. Very interestingly, some people noticed this early on in the fifties and made ternary logic computers to take advantage of this fact. The Soviet Union in particular was somewhat into this until the mid-1960s actually.

Ternary logic, composed of "trits" rather than "bits" is really interesting to think about. For many decades it was an academic curiosity, but surprisingly in the last few years after Moore's Law died and the hardware field has broken out, there has been more traction to resort back to ternary logic. Huawei has filed some patents on it in 2023 (only just coming out publicly with them in Sept of 2025). Who knows what the future will hold.

For now though, let us turn back to digital logic of the 1960s or so. Let's stick with base 2.

Adding

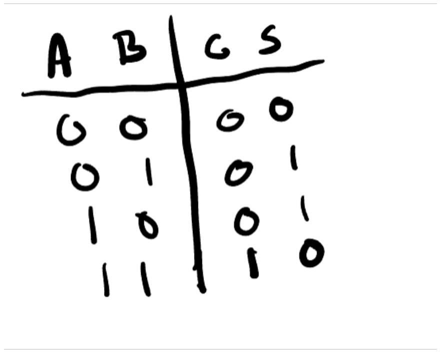
OK numbers can be represented digitally using just base 2. Fantastic. How can we add those two numbers. Well… let's think what this would look like… just consider what one "place" would have.

- $0 + 0 = 0$
- $0 + 1 = 1$
- $1 + 0 = 1$
- $1 + 1 = ...?$ … well, that would be 2. But we don't have a symbol for 2, so we'll need to carry over to the next bit and get 10.

So this output is actually two bits long in a general sense:

- $0 + 0 = 00$
- $0 + 1 = 01$
- $1 + 0 = 01$
- $1 + 1 = 10$

That means really we could refactor our system into two truth tables (like we did previously) describing operation… a truth table for the "Sum" value `S` and a truth table for the "Carry" value `C`.



A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Figure 1: Truth Tables for Sum and Carry

This little circuit can be taken together collectively and called a "Half Adder". Analyzing each truth table we'll come up with two Boolean equations:

$$S = \bar{A}B + A\bar{B} = A \oplus B$$

and

$$C = X \cdot Y$$

Circuit-wise, what logic gates could give us this behavior? Well it turns out an XOR and an AND gate. Hooking up the signals like below will give us a circuit called a "half-adder."

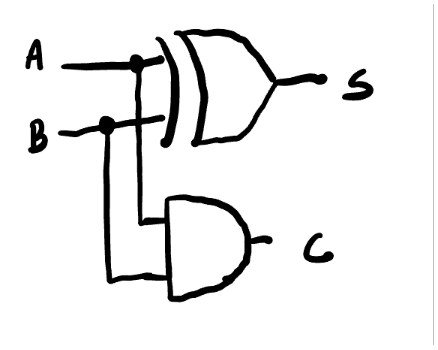


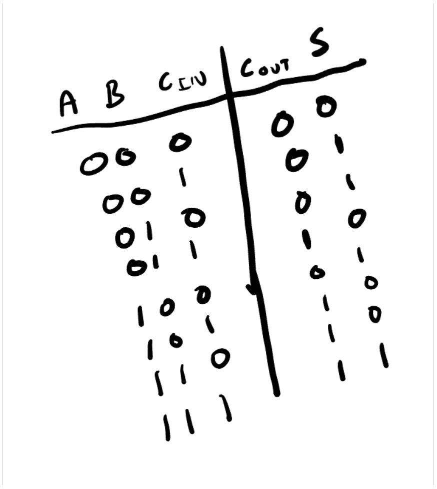
Figure 2: Half Adder

Making It More Flexible

No let's get a little bit crazy. What if… hear me out… what if instead of adding two single-bit numbers together, we added **three** single-bit numbers together?

That would have something like this as its sets of inputs/outputs:

- $0 + 0 + 0 \rightarrow 00$
- $0 + 0 + 1 \rightarrow 01$
- $0 + 1 + 0 \rightarrow 01$
- $0 + 1 + 1 \rightarrow 10$
- $1 + 0 + 0 \rightarrow 01$
- $1 + 0 + 1 \rightarrow 10$
- $1 + 1 + 0 \rightarrow 10$
- $1 + 1 + 1 \rightarrow 11$



A	B	C_{IN}	C_{OUT}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figure 3: Three Input Truth Table

It maybe takes a little bit of work to think about how we can realize this in circuit theory, but you were all at lecture where we went through that, so it hopefully won't surprise you to see the following "full adder" circuit again, which does exactly this because we designed it to do exactly this:

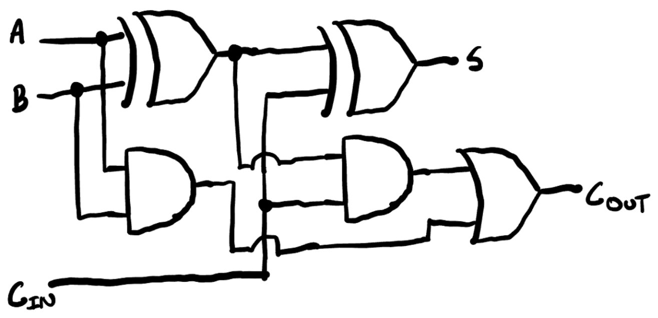


Figure 4: A Full Adder… takes in two bits and a potential carry bit, and outputs their sum (as two bits, one of which is a carry-out)

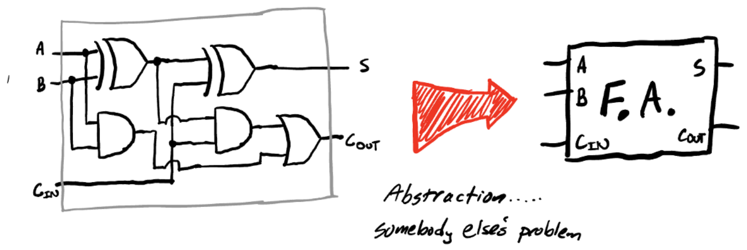


Figure 5: Ok we got a full adder... let's just draw it now as this little box

Scaling

What we've just implemented goes a long way toward accomplishing our goal of adding numbers together. When we're adding by hand, we go one column at a time, adding three small numbers together each time. What a second, hey, that sounds like what our full adder does!

So how can we scale this circuit up to bigger numbers? Well, we can just chain things together in stages. At each stage, we add together two of the corresponding bits with the carry-over bit from the previous stage, like so:

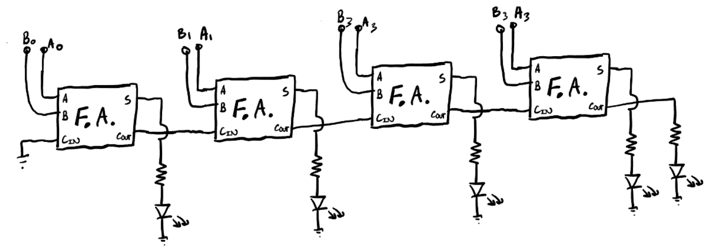


Figure 6: stacking full adders

Or, probably more meaningfully, lay them out like this so higher-order bits are further to the "left", like we'd want with our numbers.

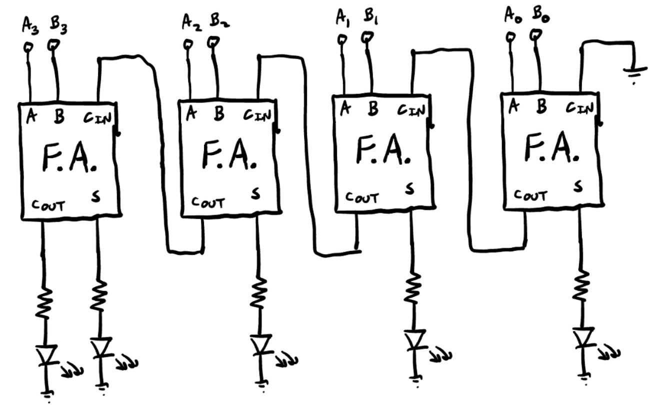


Figure 7: stacking full adders

Let's Build

Alright, our ultimate goal for this lab is to build a 4-bit adder circuit like above.

That said, there's also an optional part on the end if you want also to make a subtractor, which is really cool and totally worth doing. /gen

Logic Chips

Now the task of building this four bit adder might seem daunting. . . after all each one of those symbols representing AND or XOR or whatever is actually hiding the fact that we need to build a multi-transistor/resistor/diode/relay/something mini-circuit. The four-bit full adder would probably have about ~100 transistors or so in it if we built it this way.

Engineers from sixty years ago also felt this pain and realized that if one could package up pre-made logic gates there'd be value in it. You as the logic design could then just take these pre-packaged devices and do stuff with them without worry for the underlying transistors.

The 7400 Series

In the early 1960s Texas Instruments did just what we were daydreaming about and started to release pre-made devices containing already-tested logic gates. At first they started small, making packages that had a few NOR gates or something, but soon expanded to have a whole catalog of chips.

Along the way, engineers developed the Dual-Inline-Package (DIP) which proved to be a nice way to package and scale these modules. . . this then lead to lots of other standardizations. . . such as the modern common breadboard that we know and love (patented in 1971).

Four Bit Full Adder

We can add four bits using just three types of chips:

- The 7408: A Quad-AND gate
- The 7432: A Quad-OR gate
- The 7486: A Quad-XOR gate

All three of these chips have an identical pattern

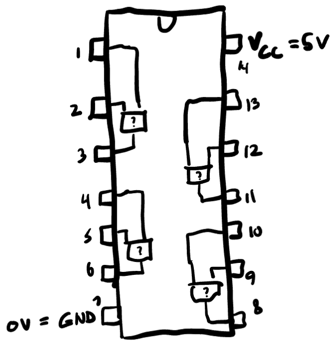


Figure 8: Quad-Two-Input Gate Packages in the 74-series. There are four "instances" of the two-input gate of interest. In the case of 7408, it is AND, in case of 7432, it is OR (inclusive). . . in case of 7486 it is XOR

At the front we have collections of these chips. Build the four-bit adder using them.

Remember the chips need to be powered (5V onto their highest pin and Ground/0V onto their "lower-right" pin is the most common for digital logic in this family).

Build it!

Adding is Great, Can we Subtract?

It certainly would be cool to subtract. How can we do that?

Two's Complement (extra)

Two's complement is a method for representing signed integers in binary that allows both positive and negative numbers to be stored and manipulated using the same circuitry. In an n-bit two's complement system, positive numbers are represented normally in binary from 0 to 2^(n-1) - 1, with the most significant bit (MSB) being 0. Negative numbers are represented by taking the positive binary representation of the absolute value, inverting all the bits (changing 0s to 1s and 1s to 0s), and then adding 1 to the result. For example, to represent -5 in 8-bit two's complement: start with +5 (00000101), invert all bits (11111010), and add 1 to get 11111011. The MSB serves as the sign bit, where 0 indicates positive and 1 indicates negative.

The beauty of two's complement is that addition and subtraction use identical hardware regardless of sign: you simply add the binary representations, and overflow/underflow is handled naturally by discarding carries beyond the fixed bit width. To convert a negative two's complement number back to decimal, you can either repeat the process (invert all bits and add 1) or recognize that the MSB has a negative weight: for an 8-bit number, interpret it as -128×b_7 + 64×b_6 + 32×b_5 + … + 1×b_0. This system elegantly represents the range from -2^(n-1) to 2^(n-1) - 1, which is why an 8-bit two's complement system can represent -128 to +127. The fact that there's one more negative number than positive (because zero takes up one positive slot) is a quirky but harmless property of the system (and has to happen since we have an even number of slots but numbers are "odd" in their shape with the single 0).

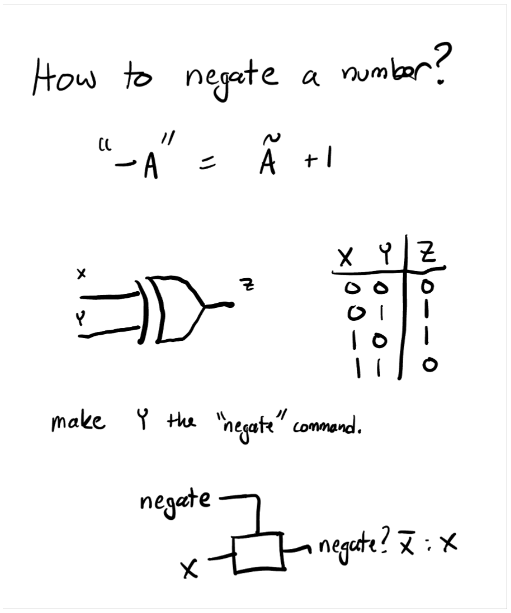


Figure 9: You can "negate" a number in two's complement by XOR-ing its bits and adding 1. That XOR can be applied to each bit of B separately

Then to take care of the +1 part:

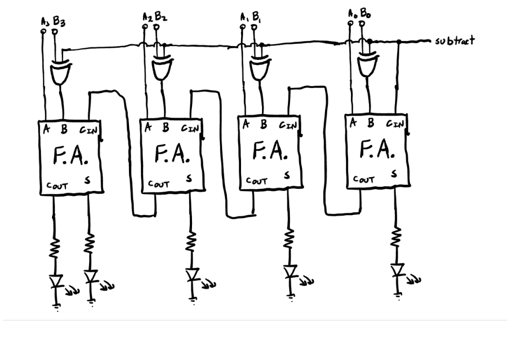


Figure 10: Make the oringal carry-in be the subtract or not bit…that way it adds that extra 1 to the whole number.

Subtractor

OK, so go ahead and do this on the board. Set up your circuit so that there is another switch that changes between doing addition and subtraction (so that when the switch is set one way you get $A + B$ on the output, and when it's set the other way you get $A - B$).