Introduction to Computer Architecture: The (almost) Comprehensive Notes

Josh Felmeden

2018
December

1 Integer representation and arithmetic

First things first, we need to look at the ways that numbers are added together (I chose to skip over how numbers are stored in the computer because I think this is boring and if you need help with this then you really are beyond all hope.). Ultimately, we've started with addition to start with, so we're going to look at simple circuits for addition in a computer. By the way, if we have a letter with a hat on (namely: $\hat{x}$), this means that it's a bit sequence representing some integer x. This leaves us with this:

$$\hat{x} \mapsto x$$
$$\hat{y} \mapsto y$$
$$\hat{r} \mapsto r$$

Alongside this, we have the following relationship:

$$r = x + y \tag{1}$$

The question is, how do we take this and represent it using boolean algebra? Well, we're going to try:

$$\hat{r} = F(\hat{x}, \hat{y}) \tag{2}$$

Where F is some boolean expression. What this means is that the $+$ operator has a similar result than F. What the bloody hell is this function? Let's have a look.

It's actually not that bad. If we look at how humans do addition, we have:

$$
\begin{array}{cccc}
 & \hat{x} = 1 & 0 & 7 \\
+ & \hat{y} = 0 & 1 & 4 \\
\hline
 & c = 0 & 1 & 0 \\
 & \hat{r} = 1 & 2 & 1 \\
\end{array}
$$

Where c is the carry (we also have 0 as a carry out here). The same thing can be represented in binary:

$$
\begin{array}{lccccccccc}
107_{10} = & \hat{x} = 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\
+14_{10} = & \hat{y} = 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
\hline
 & c = 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
 & \hat{r} = 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\
\end{array}
$$

Now, we're going to create a really simple algorithm (it's going to be called ADD), and is going to look like $\text{ADD}(\hat{x}, \hat{y}, n, b, ci)$, where b is the base, ci is the carry in and n is the length of x and y. We would then have the algorithm as follows:

```
for i = 0 to (n-1)
    r(i) += (x(i) + y(i) + c(i)) mod b
    if(x(i) + y(i) + c(i) < b)
        c(i+1) = 0
    else
        c(i+1) = 1
    end if
next
co = c(n)
return r, co
```

Let's step through this algorithm:

$$\hat{x} = \langle 7, 0, 1 \rangle \mapsto 107_{10}$$
$$\hat{y} = \langle 4, 1, 0 \rangle \mapsto 14_{10}$$
$$n = 3, \ b = 10, \ ci = 0,$$
$$\text{ADD}(\hat{x}, \hat{,}y, 3, 10, 0)$$

i	$\hat{x}_i, \ \hat{y}_i, \ c_i$	$\hat{x}_i + \hat{y}_i + c_i$	$c_{i+1}, \hat{r}_i$
0	7, 4, 0	11	1, 1
1	0, 1, 1	2	0, 2
2	1, 0, 0	1	0, 1

Where the at the end, $\hat{r} = \langle 121 \rangle$, as stated by the last column.

In the algorithm above, the bit inside the for loop can be represented by F_i, where it has the inputs $\hat{x}_i, \hat{y}_i, \hat{c}_i$ and has outputs $\hat{r}_i, c_{xi+1}$. We don't have to know what the function is, but we can write down their behaviour. Because we know what should happen[*].

c_i	$\hat{x}_i$	$\hat{y}_i$	c_{i+1}	$\hat{r}_i$
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

We know that $c_{i+1} = (\hat{x}_i \wedge \hat{y}_i) \vee (\hat{x}_i \wedge c_i) \vee (\hat{y}_i \wedge c_i)$ and we ALSO know that $\hat{r}_i = \hat{x}_i \oplus \hat{y}_i \oplus c_i$. Thus ends the middle bit of the algorithm, so now let's look at the whole algorithm.

Basically, we just string together a load of the components that we had before. That is to say, if we have n bits, we need n of those adders. Each adder takes 3 inputs: the nth digit of x and y,

---

[*]These tables took so damn long please appreciate them

and a carry, which comes from the $n-1$th adder. At the very end of the block, we get a carry out. Additionally, at each adder, we get the nth digit of the result ($\hat{r}$). This could kinda be obvious, but if you think that, shut up. It's interesting to know that there is not computation other than that in the adders. This is called a **ripple carry adder** and it relates to the loop within the algorithm. Each one of the 'components' that I was talking about is called a **full adder**, but we might replace this with a half adder later (we definitely will). We can write this whole thing in forms of boolean expressions now, so there you go.

Before we finish with these bad boys, we need to explore some examples. Let's look at when $\hat{x} = 1111$ and $\hat{y} = 0001$. If we want to add those together, we get:

$$\begin{array}{rccccl}
\hat{x} = 1 & 1 & 1 & 1 & \mapsto 15_{10} \\
+ \quad \hat{y} = 0 & 0 & 0 & 1 & \mapsto 1_{10} \\
\hline
c = 1 & 1 & 1 & 0 & (c_o = 1) \\
\hat{r} = 0 & 0 & 0 & 0 &
\end{array}$$

This is an issue because $15 + 1$ is NOT 0, and this is an error. We use the carry out to determine whether there has been an error, because if there is a carry out of 1, then there is clearly an error.

If we look at the case when we use all 1s with 2's complement (i.e. -1), and we add 1 to it, happily we get 0. We have the same behaviour, but the result is right. Unfortunately, we lost the functionality of the 1 as a carry out error marker, because there are still the possibility for errors. Take, for example, $x = 0111_2 \mapsto 7_{10}$ and $y = 0001_2 \mapsto 1_{10}$. When we add these in binary, we end up with $1000_2 \mapsto -8_{10}$, with a carry out of 0. This is not the right answer. There is kind of a way around this, where if there are a mismatch between the first bit of x and y and the first (most significant) bit of c and r. Formally then, the sign of $\hat{x}, \hat{y}, \hat{r}$ should not end up with a $+ve + +ve \to -ve$ and vice versa. This is known as a *carry error*.

With the errors that we have detected, we should be responsible people and tell the programmers that an error has occurred by medium of a flag, or even CORRECT the error, (but we can't do that because we have a fixed number of bits).

2 Transistors

An electrical current is a *flow* of electrons. A capacitor (such as a battery) works by having **free electrons** move from high to low potential. A *conductivity* rating says how easily electrons can move.

- A conductor has *high conductivity* and allows electrons to move easily.

- An insulator has low conductivity and does **not** allow electrons to move easily.

Silicon is a DOPE material, because there's lots of it, and it's also pretty cheap. It's also inert (which means boring aka doesn't react in weird ways) because it's stable enough to not react in weird ways with normal things *and* it can be doped with a donor material, which will allow us to construct the materials with the precise sub atomic properties that we want.

The result of this is a semi conductor. 'What is this?' I hear you ask. Well, it's kind of a conductor and kind of not. If this isn't any clearer, here's a little more info:

- A **P-type** semi-conductor has extra holes, while **N-type** has extra electrons.

- if we sandwich together the P and N type layers together, the result is that the electrons can only move in one way. For example, from N to P, but not vice versa.

Back in the olden days, we used to have a vacuum tube, because when the filament heats up, the electrons are produced into the vacuum, which are then attracted by the plate. They're pretty reliable generally, but they fail a fail bit during power on and off. It's also where we get the term *bug* from, since a literal bug could cause failures in this thing.

2.1 MOSFETs

We're now going to look at MOSFETS gang. MOSFET stands for Metal Oxide Semi-conductor Field Effect Transistor. Yeah, really. That's why there is an abbreviation for it. A MOSFET has 3 parts, a **source**, **drain**, and a **gate**. The source and the drain are terminals, and the gate is what controls the flow of electrons between the source and the drain. That, on a simple level, is that, because any further description is pretty freaking complicated and is not necessary for this course.

2.1.1 N-MOSFET

An N-MOSFET (or negative MOSFET) is constructed from **n-type** semiconductor terminals inside a p-type body. This means that applying a potential difference to the gate *widens* the conductive channel, meaning that the source and drain are connected, and the transistor is activated. Removing the potential difference *narrows* the conductive channel and the source and the drain are disconnected. Simply, **p.d. = current flows through, else block**.

2.1.2 P-MOSFET

A P-MOSFET (or positive MOSFET) is constructed from **p-type** semiconductor terminals inside an n-type body. Applying a potential difference to the gate *narrows* the conductive channel, meaning that no current can flow, and removing the potential difference allows the current to flow. Simply, **p.d. = no current flowing, else there is current flowing**. Also, p-types have a funny looking bobble hat in a diagram.

These MOSFETS aren't normally used in isolation, and they are used in CMOS cells, which stands for complimentary metal oxide semiconductor. We combine 2 of one type and one of the other into one body, namely, the CMOS cell. It's pretty useful because they work in complimentary ways, but there is also little leakage, or **static** power consumption. It only consumes power during the switching action (**dynamic** consumption).

2.2 Manufacture

It's necessary to be able to construct these bad boys in batch, because otherwise we wouldn't be able to make big machines out of them because we need so many of them. What we do is:

1. Start with a clean, prepared **wafer**.

2. Apply a layer of **substrate** material, such as a metal or a semi conductor.

3. Apply a layer of photoresist. This material reacts differently when it is exposed to light.

4. To do this, we expose a precise negative (or *mask*) of design that hardens the exposed photoresist.

5. Wash away the unhardened photoresist.

6. Etch away the uncovered substrate.

7. Strip away the hardened photoresist.

Remember that this algorithm repeats over and over in order to make the result 3 dimensions, rather than 2. Regularity is a huge advantage because we can manufacture a great number of similar components in a layer using a single process. The feature size (it's 90nm big) relates to the resolution of the process.

These components are USELESS in this form, so they're packaged before use, which protects against damage, including heat sinks and an interface between the component and the outside world using pins bonded to internal inputs and outputs.