Introduction to Computer Architecture: The (almost) Comprehensive Notes

Josh Felmeden

2018
December

Contents

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:

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

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

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

Now, we're going to create a really simple algorithm (it's going to be called ADD), and is going to look like 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, 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

*These tables took so damn long please appreciate them

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:

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

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.

So, while MOSFETs are pretty great, there are down sides. Designing complex functionality using transistors alone is really hard because transistors are simply *too* low level. We can address this problem by repackaging groups of transistors into logic gates, since logic gates are ordered logic gates such that we get certain functionality. Pretty cool right? It's like nerdy Lego.

3 Logic Gates

If we form a **pull-up network** of P-MOSFET transistors, connected to V_{dd} (which is the high voltage rail), and a **pull-down network** of N-MOSFETs, connected to V_{ss} (the low voltage rail), and assume that the power rails are everywhere:

$$V_{ss} = 0V \approx 0$$
$$V_{dd} = 5V \approx 1$$

We can then describe the operation of each logic gate using a truth table.

3.1 NAND gate

If both x and y are 0 (connected to V_{ss}), then:

1. Both top P-MOSFETs will be connected.
2. Both botton N-MOSFETs will be disconnected.

3. r (the output) will be connected to V_{dd}

If x is 1 and y is 0, then:

1. The right most P-MOSFET will be connected.
2. The upper-most N-MOSFET will be disconnected.
3. r will be connected to V_{dd}

If x is 0 and y is 1, then:

1. The left most P-MOSFET will be connected.
2. The lower-most N-MOSFET will be disconnected.
3. r will be connected to V_{dd}

Finally, if both x and y are 1, then:

1. Both top P-MOSFETs will be disconnected.
2. Both bottom N-MOSFETs will be connected.
3. r will be connected to V_{ss}

3.2 NOR gate

If both x and y are 0, then:

1. Both top P-MOSFETs will be connected.
2. Both bottom N-MOSFETs will be disconnected.
3. r will be connected to V_{dd}

If x is 1 and y is 0, then:

1. The upper-most P-MOSFET will be disconnected
2. the left-most N-MOSFET will be connected.
3. r will be connected to V_{ss}

If x is 0 and y is 1, then:

1. The lower-most P-MOSFET will be disconnected.
2. The right-most N-MOSFET will be connected.
3. r will be connected to V_{ss}.

If both x and y are 1, then:

1. Both top P-MOSFETs will be disconnected.
2. Both botton N-MOSFETs will be connected.
3. r will be connected to V_{ss}.

3.3 Physical limitations

There are, of course, some physical limitations that we haven't discussed yet. There are two classes of delay (often described as **propagation delay**), that will dictate the time between change to some input and corresponding change in an output. These are:

- **Wire delay:** This relates to the time taken for the current to move through the conductive wire from one point to another.
- **Gate delay:** This relates to the time taken for the transistors in each gate to switch between the connected and disconnected states

Normally, the gate delay is more than wire delay, and both relate to the implementations. Gate delay is the fault of the properties of the transistors used, and wire delay to the properties of the wire.

Critical path is the longest sequential delay possible in some combinatorial logic. Basically, the worst case scenario in a given logic circuit.

Ideally, we'd get a perfect digital on/off graph of the response, while in reality, we get a more curved graph. It would also make sense to have 1 and 0 as a threshold, rather than a definitive voltage. This fuzzy representation allows for some inaccuracies that are unavoidable with this physical implementation.

Including gate delay gives us a *dynamic* view computation. We're gonna kind of ignore wire delay for the moment and arbitrarily choose some delays for the gates:

1. NOT: 10ns
2. AND: 20ns
3. OR: 20ns

If we then did some computation, we could see the output changing after we switch x from 0 to 1 and keep y as 1. If we're using an XOR gate, then it would take 50ns to completely change to 0 (since x and y are both 1, hence x XOR y = 0).

It is cool to use *3-stage logic*, using an extra value:

- 0 is false
- 1 is true

- Z is **high impedance**

High impedance is the null value, so we can allow a wire to be disconnected.

If we have two inputs connected to an output, you can have some weird results where the two inputs are in conflict with the other. The way to fix this is to have some enable switch on the two inputs to the output, so we don't get any inconsistencies.

Here are some more definitions:

- **fan-in** is used to describe the number of inputs to a given gate.
- **fan-out** is used to describe the number of inputs (or *other* gates) the output of a given gate is connected to.

4 Combinatorial Logic

The logic gates that we've discussed already are higher level than the transistors that we started off with. Armed with these new logic gates, we can move onto *actual* high level components. We want components that are closer to what we can actually do things with.

Thus far, we've just looked at various ways of writing the same things; having moved from the completely abstract pencil and paper, to the implementation of logic gates and NAND-boards.

4.1 Design patterns

There are a number of design patterns that we can take:

4.1.1 Decomposition

Divide and conquer, take some complicated design, and break it down into simpler and more manageable parts.

4.1.2 Sharing

We can replace two AND gates that exist in some design with a single AND gate, using the same gate from the usage points. It makes sense because the output will always be the same.

4.1.3 Isolated Replication

Say we have a 2 input AND gate, and we want a 2-input m-bit AND gate, which is simply a replication of 2-input, 1-bit AND gates, then we are going to have m-bits of the AND gate. We're also saying that they are operating in parallel, and that each bit does not affect the neighbouring bits.

4.1.4 Cascaded Replication

Using the same example as before, instead of using m AND gates, we want something like this:

$$\begin{aligned} r = \ & x_0 \wedge x_1 \ \wedge \ x_2 \wedge x_3 \\ = \ & (x_0 \wedge x_1) \wedge \ (x_2 \wedge x_3) \end{aligned}$$

You can also write that like this:

$$r = \bigwedge_{i=0}^{n-1} x_i$$

It's different from isolated replication, because the and gates are entirely isolated from one another.

4.2 Mechanical Derivation

In every case, the process is the same. We want to be able to process some truth table and get something that will work (with a boolean expression). So, let's let T_i denote the jth input for $0 \leq j < n$, and let O denote the single output.

1. Find a set T such that $i \in T$ iff. $O = 1$ in the ith row of the truth table.
2. For each $i \in T$, form a term t_i by ANDing together all the variables while following two rules:
   a) if $T_j = 1$ in the ith row, then we use T_j as is, but
   b) If $T_j = 0$ in the ith row, then we use $-T_j$
3. An expression implementing this function is then formed by ORing all of the terms together:

$$e = \bigvee_{i \in T} t_i$$

Let's consider the example of deriving an expression for XOR:

$$r = f(x, y) = x \oplus y$$

This is a function described by the following table:

x	y	r
0	1	1
1	0	1
1	1	0

4.3 Karnaugh Map

The simple algorithm for creating a Karnaugh map is as follows:

1. Draw a rectangular $(p \times 1)$ element grid:
   a) $p = q = 0(mod 2)$, and
   b) $p \cdot q = 2^n$
2. Fill the grid elements with the output that corresponds to inputs for that row and column.
3. Cover rectangular groups of adjacent 1 elements which are of total size 2^n for some group m, groups can 'wrap around' if you like, so they can go over edges of the grid and overlap.
4. Translate each group into a single term in some SoP form Boolean expression, where
   a) bigger groups
   b) less groups

   mean a simpler expression.

The basic idea is that we don't count up as per normal in binary, but we actually count up by changing one binary digit at a time. You then group them, and the groups have to be adjacent (but they can't form a kind of L shape). After this, it's easy to group them into some format.

4.4 Building blocks

There are two more blocks that we're going to look at now:

1. **Multiplexer**
   - It has m inputs,
   - 1 output and
   - uses a $(\log_2(m))$-bit control signal input to choose which input is connected to the output.
2. **Demultiplexer**
   - It has 1 input,
   - m outputs and
   - uses a $(\log_2(m))$-bit control signal to choose which output is connected to the input

Remember that the inputs and outputs are n bits, but they obviously have to match up. The connection that is made is continuous, because both components are *combinatorial*.

4.4.1 Multiplexer

The behaviour of a 2-input, 1-bit multiplexer is as follows:

$$r = (\neg\, c \wedge x) \vee (c \wedge y)$$

It can be more clearly represented by this table:

c	x	y	r
0	0	?	0
0	1	?	1
1	?	0	0
1	?	1	1

4.4.2 Demultiplexer

The behaviour of a 2-output, 1-bit demultiplexer is as follows:

$$r_0 = \neg\, c \wedge x$$
$$r_1 - c \wedge x$$

It can be more clearly represented in this table:

c	x	r_1	r_0
0	0	?	0
0	1	?	1
1	0	0	?
1	1	1	?

Multiplexers can be used to for isolated replication. If we line up 4 multiplexers, we get a control signal in to choose between two choices, and we get one of those out, meaning that we end up with a 4-bit output. We can scale this in the same way as before and use the *cascaded* pattern, meaning that they interact with one another. We now need a 2-bit control signal however, because in the other design, each multiplexer uses the same control signal.

Building on this idea, we can look at two more components:

1. Half adder
    - Has 2 inputs, x and y,
    - Computes the 2-bit result $x + y$
    - Has 2 outputs: a sum s and a carry out co (which are the least significant bit and the most significant bit of the result)

2. Full adder

   - Has 3 inputs: x, y, and a carry-in ci
   - Computes the 2-bit result $x + y + ci$
   - Has 2 outputs: a sum s and a carry out co (which are the least significant bit and the most significant bit of the result)

4.5 Half adder

The behaviour of the half adder looks like this:

x	y	co	s

And can be displayed by the following equations:

$$co = x \wedge y$$
$$s = x \oplus y$$

4.6 Full adder

The behaviour of a full adder looks like this:

ci	x	y	co	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

And can be displayed by the following equations:

$$\begin{aligned} co &= (x \wedge y) \vee (x \wedge ci) \vee (y \wedge ci) \\ &= (x \wedge y) \vee ((x \oplus y) \wedge ci) \\ s &= x \oplus y \oplus ci \end{aligned}$$

A full adder is kind of like two half adders put together in a nice way.

There is a circuit for the full adder that allows us to implement the cascading design choice to replicate the n-bit addition; cascading because of the carries that we have are *cascaded* into one another. It

might not be immediately clear that it's cascaded when we initially looked at it, but now as we look at it from a higher value, then we can clearly see that it is indeed cascaded.

Moving onto equality comparators, we have 2 final building blocks:

1. An equality comparator
    - Has 2 inputs: x and y,
    - computes the 1 output as:
    $$r = \begin{cases} 1 & \text{if x = y} \\ 0 & \text{otherwise} \end{cases}$$
2. A less-than comparator
    - Has 2 inputs: x and y,
    - computes the 1 output as:
    $$r = \begin{cases} 1 & \text{if } x < y \\ 0 & \text{otherwise} \end{cases}$$

All of the inputs and outputs are only 1 bit.

Even though we only have these two, we can do a lot of other comparators, for example, we can get $x \neq y$ from passing the result of an equality gate through a not gate.