Introduction to Computer Architecture: The (almost) Comprehensive Notes

Josh Felmeden

2018
December

Contents

1 Integer representation and arithmetic 1

2 Transistors 3
2.1 MOSFETs . . . 4
2.1.1 N-MOSFET . . . 4
2.1.2 P-MOSFET . . . 4
2.2 Manufacture . . . 4

3 Logic Gates 5
3.1 NAND gate . . . 5
3.2 NOR gate . . . 6
3.3 Physical limitations . . . 7

4 Combinatorial Logic 8
4.1 Design patterns . . . 8
4.1.1 Decomposition . . . 8
4.1.2 Sharing . . . 8
4.1.3 Isolated Replication . . . 8
4.1.4 Cascaded Replication . . . 9
4.2 Mechanical Derivation . . . 9
4.3 Karnaugh Map . . . 10
4.4 Building blocks . . . 10
4.4.1 Multiplexer . . . 11
4.4.2 Demultiplexer . . . 11
4.5 Half adder . . . 12
4.6 Full adder . . . 12
4.7 Equality comparator . . . 13
4.8 Less-than comparator . . . 13

5 Sequential logic 15
5.1 Clocks . . . 16
5.2 Latches and flipflops . . . 16
5.2.1 SR component . . . 16
5.2.2 D component . . . 17
5.2.3 Updates . . . 18
5.2.4 Registers . . . 18
5.3 Returning to the original issue . . . 18
5.4 Memory Component . . . 19
5.4.1 History of memory . . . 19
5.4.2 Low-level implementation . . . 20
5.4.3 Higher level implementation . . . 21

6 Finite State Machines 21
6.1 Automata Theory . . . 21
6.2 FSMs in hardware . . . 23

7 Data, control, and instructions **25**
7.1 Instructions 25

8 Memory **26**
8.1 Addressing 27
8.1.1 Immediate addressing 27
8.1.2 Direct addressing 27
8.1.3 Memory-indirect addressing 27
8.1.4 Register-Indirect addressing 28
8.1.5 Indexed addressing 28

9 Compilers **28**
9.1 Instruction set 28

10 Hex Architecture **30**
10.1 The stack 30
10.2 The translation process 31

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:

$$\begin{aligned}\hat{x} &\mapsto x\\ \hat{y} &\mapsto y\\ \hat{r} &\mapsto r\end{aligned}$$

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:

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

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

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

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:

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

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 $+ve + +ve \rightarrow -ve$ 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 bottom 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 bottom 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:

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

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:

$$
\begin{aligned}
co &= x \wedge y \\
s &= x \oplus y
\end{aligned}
$$

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.

4.7 Equality comparator

The behaviour of the equality comparator is as follows:

$$r = \neg(x \oplus y)$$

And can be modelled by this table:

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

4.8 Less-than comparator

The behaviour of the less-than comparator can be modelled as follows:

$$r = \neg x \wedge y$$

And can be shown in this table:

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

In the same way that we had a ripple carry adder, we can have an n-bit comparison. For a 3-bit number, for them to be equal, every single bit must match. Therefore, if we first compare the first bits, and then and this with the second bits, and then and it again this with the third bits, this will give us the result we require.

For a cyclic less-than format, we have to do use a less than and an equality block to be able to go through and work out which is bigger. Let's say we take two numbers, x and y, where $x = 123$ and $y = 223$. We can see that y is bigger than x, because we started at the left most digit and compared these. This is quite an easy example, so let's look at another one.

In this example, $x = 121,\ y = 123$. Here, we start at the left most digit, which are the same. So, we move inward, which is the same again. Doing this one more time, we can see that this digit of y is bigger than that of x, therefore $y > x$. We have some rules here then:

1. If $x_i < y_i$, then $x < y$
2. If $x_i > y_i$, then $x \not< y$
3. if $x_i = y_i$, then $x < y$ IF the $rest(x) < rest(y)$

In the format of an equation:

$$x < y \text{if } (x_i < y_i) \vee (x_i = y_i \wedge \text{ rest } x < \text{ rest } y)$$

We have this joined up to each bit of the input.

The final set of components are the **encoder** and the **decoder**. They can also be viewed as translators, or, formally:

1. an n-to$m$ encoder translates an n-bit input into some m-bit code word, and
2. an m-to-n decoder translates an m-bit code word **back** into the same n-bit output.

Where if only one output is allowed to be 1 at a time, we call it a **one of many** encoder.

A *general* building block is impossible because it depends on the scheme for encoding or decoding. Let's look at an example:

1. To encode, take n inputs, for example x_i for $0 \leq i < n$ and produce an unsigned integer x' that determines which $x_i = 1$
2. To decode, take x' and set the right $x_i = 1$

Where for all $j \neq i, x'_j = 0$ (so both are **one-of-many**) and this means we recover the original x.

We can map the behaviour by creating a truth table:

x_3	x_2	x_1	x_0	x'_1	x'_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Which also correlates to:

$$x'_0 = x_1 \vee x_3$$
$$x'_1 = x_2 \vee x_3$$

The decoder should be exactly the opposite of this, and can be modelled by this equation:

$$\begin{aligned} x_0 &= \neg x'_0 \wedge \neg x'_1 \\ x_1 &= x'_0 \wedge \neg x'_1 \\ x_2 &= \neg x'_0 \wedge x'_1 \\ x_3 &= x'_0 \wedge x'_1 \end{aligned}$$

There is a problem with this because if we break the rules and set more than one bit of x to 1, then the encoder fails because it produces two 1s. The solution is to produce a priority encoder, where one input is given priority over another.

x_3	x_2	x_1	x_0	x'_1	x'_0
0	0	0	1	0	0
0	0	1	?	0	1
0	1	?	?	1	0
1	?	?	?	1	1

5 Sequential logic

Imagine that we need a cyclic n-bit counter, such as a component whose output steps through the values:

$$0, 1, \cdots, 2^n - 1, 0, 1, \cdots$$

But is otherwise free running.

We already know how to make an n bit adder, so we can just compute $r \leftarrow r + 1$ over and over. It sounds cool, but we can't initialise the value, and also we don't let the output of each full-adder settle before it's used again as an output.

The main issue that we have is that combinatorial logic has some limitations because we can't control *when* a design computes some output, nor remember the output when produced. A good solution is **sequential logic** which demands that there is some way to control components, one or more components that remember the state that they're in, and a mechanism to perform computation as a sequence of steps, rather than continuously.

5.1 Clocks

A clock is a signal that alternates between 1 and 0. We call the period of time where it is 1 as *positive level*, and when it has a value of 0, it has a *negative level*. Also, when it changes from positive to negative, we call this a **negative edge**, while the opposite of this is called a **positive edge**.

We want to clock to *trigger* events in order to synchronise components within a design, while the clock frequency is how many clock cycles happen in a unit of time. It's gotta be fast enough that the design goals are met, but slow enough that the critical path of a given step is not exceeded. The faster the clock ticks, the better things are, but we can't go too fast. The clock is therefore a conductor.

An n phase-clock is distributed as n separate signals along n separate wires. A 2-phase instance is really useful because it has the features of a 1-phase clock, such as the clock period etc, but there is a guarantee that positive levels of the two clocks don't overlap, and the behaviour is parameterisable by altering the clock length (δ_i).

5.2 Latches and flipflops

A bistable component can exist in two stable states, like 0 or 1 at a given point. It can:

- Retain some **current state** – Q –, which can also be read as output, and
- be updated to some **next state** – Q' – which is provided as an input.

Under control of an enable signal *en*. We say it is:

- **Level-triggered** and hence a latch if it's updated by a given level of *en*, or
- **Edge triggered** and hence a flipflop if it's updated by a given edge of *en*.

They both have different properties, but both are useful.

5.2.1 SR component

An "SR" latch/flip-flop component has two inputs: S (for set), and R (for reset). When enabled, the following behaviour can be observed:

- S = 0, R = 0, the component retains Q
- S = 1, R = 0, the components updates to Q = 1,
- S = 0, R = 1, the component updates to Q = 0,

- S = 1, R = 1, the component is metastable.

When it's not enabled, the component is in 'storage mode'

It can also be described by the truth table:

S	R	Q	$\neg Q$	Q'	$\neg\ Q'$
0	0	0	1	0	1
0	0	1	0	1	0
0	1	?	?	0	1
1	0	?	?	1	0
1	1	?	?	?	?

5.2.2 D component

A "D" latch/flip-flop component has a single input - D. When it's enabled, the following behaviour can be observed:

- $D = 1$, the component updates to $Q = 1$
- $D = 0$, the component updates to $Q = 0$

When it's not enabled, the component is in storage mode, and retains Q. The behaviour can be modelled by this truth table:

D	Q	$\neg Q$	Q'	$\neg Q'$
0	?	?	0	1
1	?	?	1	0

How do we design a simple SR latch? We use two *cross-coupled* NOR gates. It can be shown through these equations:

$$S \bar{\wedge} Q = \neg Q$$
$$R \bar{\wedge} \neg Q = Q$$

There's a loop here, because in order to know the output from the top NOR gate, we need to know the output from the bottom one, and vice versa. So what gives?? Well, it's alright if we look at the behaviour of a NOR gate.

We know that a NOR gate gives us a 1 only when both inputs are 0. If either input is 1, then NOR will produce a 0. This is really important for our gate to actually function.

If we set $S = 0$, then the output from the top NOR gate is forced to produce 0. Then, if R is 0, then the output is 1. Similarly, if R is 0, the output for THAT NOR gate is 1. Therefore, we get what we want. The reverse is true when R is 1 and S is 0.

There is some form of issue, though, because if both R and S are 0, then either the top NOR gate spits out 1, or the bottom NOR gate does. That's just the way it is. There's no logic error, it's just a bit weird.

There's just one case left to study. If S and R are 1. In the truth table, we stated that we didn't care what happened, but in the implementation, then both Q and $\neg Q$ are both 0, which is clearly a bloody issue because Q cannot equal $\neg Q$. What happens now? Well, the latch has to settle in some case, either $Q = 1$ or $Q = 0$. We have 0 control over what happens. We don't want this behaviour at all. This is what meta stability means. It's stable in some sense, but kind of not.

5.2.3 Updates

We want to be able to control when updates occur, because without this, it's not really a latch yet. To do this, we have an AND gate before we get to the criss-cross NOR gated latch, so we AND S and the enable signal, and R and the enable signal, and then pass the results from this into the old S and R respectively. This is a technique known as **gating**.

Ok, this just in. WE CAN GET RID OF THE ISSUE OF META-STABILITY. Yep, you heard that right. We force $R = \neg S$, so we get either $S = 0, R = 1$ or $S = 1, R = 0$. Dope, right?

The difference between a latch and a flip-flop is that is has a little triangle on the symbol, indicating that it's edge triggered.

For a latch, the point value of Q only changes when the enable signal is 1, it then matches the input D. Some people call this a 'transparent' latch, because it's like the latch isn't even there.

In a flip flop, the input only changes to the input D on the edges of the clock (when the enable switch is on).

5.2.4 Registers

We normally group these components into **registers**, which is an n bit register that can then store an n bit value. We take each latch as a single bit.

5.3 Returning to the original issue

So we have some data path of computation and/or storage components, and a control path, that tells the components in the data path what to do and when to do it.

What we have is some latch based register, that takes in ϕ_1, the first clock signal as an enable signal. This gives an input to some combinatorial logic, which then passes the result into another latch based register that takes in ϕ_2, the second clock signal. This register passes the information back to the first register. And so on. This creates a kind of buffer that gives us a counter that we wanted. The two registers mean that the loop is broken, because it is impossible for both registers to be enabled at the same time; breaking the infinite loop that we would have without them.

The combinatorial logic in the middle is just a big boy ripple carry adder. There is also a reset signal that is NOTED, and then ANDED with the individual bits of the ripple carry adder, meaning that when the RESET signal is 1, it becomes 0, thereby resetting all bits, because $x \wedge 0 = 0$.

5.4 Memory Component

An n-bit register based on latches or flip-flops has a couple of limitations:

1. Each latch or flip-flop in the register needs a fairly big number of transistors. This limits the viable capacity
2. The register is also not addressable, because an address or index allows dynamic rather than static reference to some stored data. That means that we have something like variables, when we want an array.

The solution to these limitations is to have a memory component. The memory is connected to a **user** which could be a CPU by control signals, a data bus, and an address bus. Sound familiar? We'll also say that the memory has a capacity of $n = 2^{n'}$ addressable words and each word is w bits, where $n \gg w$.

There are a number of ways to classify a given memory component:

1. Volatility
   - **Volatile** means that the content is lost when the component is powered off
   - **Non volatile** means that the content is not lost when the component is powered off
2. Interface type:
   - **Synchronous** where a clock or pre-determined timing information synchronises steps
   - **Asynchronous** where a protocol synchronises steps
3. Access type
   - **Random vs. Constrained** access to content.
   - **Random Access Memory (RAM)** which can be read from *and* written to.
   - **Read Only Memory (ROM)** which can only be read to

We're going to focus on a **volatile, synchronous RAM**.

5.4.1 History of memory

In the olden days, we had **delay line** memory, that contained mercury. There is a speaker at one end to store sound waves into the line and a microphone at the other end to read them out.

Values are stored in the sense that the corresponding waves take some time to propagate, when they get to one end, they are either replaced, or fed back into the other.

This is known as sequential access because you need to wait for the data you want to appear.

After delay line came **magnetic-core** memory, where the basic idea is that the memory is a matrix of small magnetic rings or cores which can be magnetically polarised to store values. Wires are threaded through the cores to control them (so to read or write values). The magnetic polarisation is retained, so this is non-volatile. Sometimes, main memory is termed **core-memory** (or **core dump**), which is in reference to this.

5.4.2 Low-level implementation

There are two kinds of RAM that we can use: **Static RAM** or (SRAM) and **Dynamic RAM** or (DRAM).

SRAM is:

- manufacturable in lower densities
- More expensive to manufacture
- Faster to access
- Easier to interface with
- Ideal for latency optimised contexts (like cache memory)

DRAM is:

- Manufacturable in higher densities.
- Less expensive to manufacture
- Slower to access
- Harder to interface with
- Ideal for capacity optimised contexts (like main memory)

Simply, an **SRAM cell** resembles two NOT gates on the inner loop. It basically carries around a signal that is reinforced by the NOT gates. On the outside, there are two transistors that are connected to some control signal wl. The transistors allow access to the values inside the loop. To read from this, we pre-charge the transistors to 1. This is logically inconsistent, but it means that we allow one of them to be overridden after we allow access to the inside of the loop (through wl). To write to this, we charge the left transistor (or bl) to the value, and the right one to the opposite (since it is called $\neg bl$).

An issue that we might find is different signal strengths. We're going to kind of brush it under the carpet and not think about it.

In total, we'd need 6 transistors: 2 for each of the 2 NOT gates, and one each for the two on the outside. Therefore, it's known as a **6t** SRAM cell. It's loads less than the 20 or so required by the latches and flip-flops. This therefore solves one of the limitations of the latches.

A **DRAM** cell is constructed only using 1 single transistor and a capacitor (kind of like a battery because it stores charge). The transistor again has an enable signal wl, and some input bl. If we want to read, then we set bl to 1. If the contents is 1, then there is a discharge through bl, else there will not be. To be able to write, we set bl to the required value, and then set wl to 1. After this, the capacitor is charged with the required value.

The issue with this cell is that the capacitor can only do limited charges. Also, the same issue as before, because the signal strength might not be enough for the required circuit. Additionally, the charge stored in the capacitor will decay over time. We need some refresh thing in the background to ensure that the value is not lost over time. Reading from a DRAM cell means that the value is destroyed, so we need to rectify that through the refresh mechanism. The latency is long from a DRAM cell because the capacitor has a long access time.

5.4.3 Higher level implementation

A **memory device** is constructed from (roughly) three components:

1. A **memory array** (or matrix) of replicated cells with r rows and c columns.
2. A **row decoder** which given an address (de)activates associated cells in that row, and
3. A **column decoder** which given an address (de)selects associated cells in that column

Along with additional logic to allow use:

1. **Bit line conditioning** to ensure that the bit lines are strong enough to be effective,
2. **Sense amplifiers** to ensure that the output from the array is usable.

To access these bad boys, we get an input to the row decode from the address bus, and an output from the column decode. It's kind of like a multiplexer and then a demultiplexer.

The difference between DRAM and SRAM is simply that the cells in the middle are DRAM rather than SRAM.

DRAM also has buffers (both for the row and the column).

6 Finite State Machines

6.1 Automata Theory

First things first, we need some definitions:

- **Alphabet**: a non-empty set of symbols
- **String**: with respect to some alphabet Σ is a sequence of finite length, whose elements are members of Σ.
- **Language**: a set of strings.

Finite state machines are a model of computation. What does this mean? It's a set of allowable (or possible) operations. It is basically an idealised computer that is in some finite set of states at a given point in time. It accepts an input string (with respect to some alphabet) one symbol at a time. Each symbol produces a change in state. Once it's run out of inputs, the computer stops. Depending on the state it stops in, it can either *accept* or *reject* the string. For a language of all possible strings that the computer could accept, the computer **accepts** the language.

Finite state machines do not have any memory. The more advanced machine that you have, the more memory you have. We're looking at a kind of mid-rate deal, that does not have any memory.

For a more formal definition of FSMs, it is a tuple:

$$C = (S, s, A, \Sigma, \Gamma, \delta, \omega)$$

With the following definitions:

- S is a finite set of states that includes a **start state**, $s \in S$
- $A \subseteq S$, a finite set of **accepting states**
- An **input alphabet** Σ and an **output alphabet** Γ
- A **transition function**:
$$\delta : S \times \Sigma \to S$$
- An **output function**:
$$\omega : S \to \Gamma$$
In the case of a **Moore FSM**, or
$$\delta : S \times \Sigma \to \Gamma$$
In the case of a **Mealy FSM**

An empty input is called ϵ

We need to design an FSM that decides whether a binary sequence X has an odd number of 1 elements in it. The *accepting state* is S_{odd}.

For example, say we have the input string $X = \langle 1, 0, 1, 1 \rangle$ the transition are:

$$\to S_{even} \xrightarrow{X_0=1} S_{odd} \xrightarrow{X_1=0} S_{odd} \xrightarrow{X_2=1} S_{even} \xrightarrow{X_3=1} S_{odd}$$

So the input is accepted because there are an odd number of 1 elements.

The real world example of this is to use a regular expression (or Regex).

6.2 FSMs in hardware

Using an FSM with a latch based implementation is that:

1. The state is retained in a register
2. δ and ω are simple combinatorial logic
3. Within the current clock cycle:
    a) ω computes the output from the current state and input
    b) δ computes the next state from the current state and input
4. The next state is latched by an appropriate feature in the clock.

This is a computer that we can actually build now!

We have two options as to the choice of encoding the state of the machine:

There is **binary encoding**:

$$\begin{aligned} S_0 &\mapsto \langle 0,0,0 \rangle \\ S_1 &\mapsto \langle 1,0,0 \rangle \\ S_2 &\mapsto \langle 0,1,0 \rangle \\ S_3 &\mapsto \langle 1,1,0 \rangle \\ S_4 &\mapsto \langle 0,0,1 \rangle \\ S_5 &\mapsto \langle 1,0,1 \rangle \end{aligned}$$

Or, there is **One- hot encoding**:

$$\begin{aligned} S_0 &\mapsto \langle 1,0,0,0,0,0 \rangle \\ S_1 &\mapsto \langle 0,1,0,0,0,0 \rangle \\ S_2 &\mapsto \langle 0,0,1,0,0,0 \rangle \\ S_3 &\mapsto \langle 0,0,0,1,0,0 \rangle \\ S_4 \mapsto \langle 0,0,0,0,1,0 \rangle S_5 &\mapsto \langle 0,0,0,0,0,1 \rangle \end{aligned}$$

There is a trade off here because there is more space required for the one-hot, but there is less energy required because we are only flipping two bits at a time.

Now, we want to design an FSM that acts like a cyclic counter modulo n (rather than 2^n as before). If $n = 6$ for example, we want a component whose output r steps through values:

$$0, 1, 2, 3, 4, 5, 0, 1, \cdots$$

The first couple of steps of our algorithm state that we need to enumerate each state, and then give them some abstract label. We can represent this using a table:

	δ	ω
Q	Q'	r
S_0	S_1	0
S_1	S_2	1
S_2	S_3	2
S_3	S_4	3
S_4	S_5	4
S_5	S_0	5

Next, we need to design the state assignment step:

$$S_i \mapsto i (0 \leq i \leq 5)$$

Since $2^3 = 8 > 6$, we can represent them using 6 concrete values, namely:

$$\begin{aligned} S_0 &\mapsto \langle 0,0,0 \rangle = 000_2 \\ S_1 &\mapsto \langle 1,0,0 \rangle = 001_2 \\ S_2 &\mapsto \langle 0,1,0 \rangle = 010_2 \\ S_3 &\mapsto \langle 1,1,0 \rangle = 011_2 \\ S_4 &\mapsto \langle 0,0,1 \rangle = 100_2 \\ S_5 &\mapsto \langle 1,0,1 \rangle = 101_2 \end{aligned}$$

And capture:

$$\begin{aligned} Q &= \langle Q_0, Q_1, Q_2 \equiv \text{the current state} \\ Q' &= \langle Q'_0, Q'_1, Q'_2 \rangle \equiv \text{the next state} \end{aligned}$$

in a 3-bit register (so like 3 latches or 3 flip-flops)

We can rewrite the abstract labels to give us some concrete truth table:

Q_2	Q_1	Q_0	Q'_2	Q'_1	Q'_0	r_2	r_1	r_0
0	0	0	0	0	1	0	0	0
0	0	1	0	1	0	0	0	1
0	1	0	0	1	1	0	1	0
0	1	1	1	0	0	0	1	1
1	0	0	1	0	1	1	0	0
1	0	1	0	0	0	1	0	1
1	1	0	?	?	?	?	?	?
1	1	1	?	?	?	?	?	?

The big block of 'don't care' means that we should never really reach those. We can also apply the technique of Karnaugh maps.

And that's it for the Dan Page bit of the course!

7 Data, control, and instructions

Abstraction is pretty great. A higher level only needs to know how to interface with the level directly below it. So, why should we care about hardware? Well, it's kind of important because if we write a program that's too slow, or it isn't running right, having a grasp of how a computer works lower down the levels of abstraction will really help us to be able to fix whatever problems might arise.

A processor is dictated by two separate functions:

- **Control**: the information and instructions
- **Data**: the information operated on to get the result.

These two influences form two paths into the processor logic.

The definition of **data** is simply some stored information. It can be stored or formatted in a number of ways. Data stores information and forms input to, and outputs from, calculations.

Data is stored in storage elements. There are many different storage elements in modern processing systems. During this course, we only need to focus on two: **memory** and **registers**. For this section, we can treat them both as *black boxes*, meaning that we don't really care what's going on inside. Processor instructions always operate on data in registers, and if they're lucky, they get to operate on data in memory, too.

Control is also information, but the usage differs slightly from data. It specifies what needs to be done and is applied to a system, rather than consumed by it.

7.1 Instructions

Instructions are really useful for telling the computer exactly what we want to be doing. For example, if an instruction "A" is called, then the processor should do "X". At a high level, we can treat them as abstract symbols, whose value causes different processor behaviour. Instructions need to be **decoded** before they can be used.

An instruction is *also* information. However, it has a defined purpose, which is to specify the exact amount of work that needs to be done by a processor. This leads it to have a specific form and formatting. Only a subset of all possible control values are actually valid instructions.

Instructions allow the encoding of control information that is necessary to control a computing system. The trick is to use a unique code (called the **op-code**) to signify a unique function.

There are many different possible encodings of instructions for the same meaning. Each system has its own tailored decode module to figure out the meaning of the op-codes to generate control signals.

There is more than one way to make a decoder. It could be combinatorial logic, demultiplexer based, or lookup based.

8 Memory

Memory, simply, is a place to store information. There are two basic operations that we can perform on memory: **read** and **write**. Each piece of information in memory is assigned to a unique address. In order to access or update information, we need to specify this address to a memory, and then our information can be returned or changed. Addresses are specified as indexes. Values can be op-codes.

A single memory location can combine both an instruction and some data. This can be useful for constant based operations. Consider ADD1 or something that loads a constant, such as MOVE2. We could express ADD as 1, therefore we could also express ADD1 as 11. There is a memory hierarchy that looks like this:

1. Register
    - 32 words
    - Access time of < 1 ns
2. L1 cache
    - 32KB
    - 1ns access time
    - Part of SRAM
3. L2 cache
    - 512KB
    - 5-10ns access time
    - Part of SRAM
4. L3 cache
    - 1-8MB
    - 10-100ns access time
    - Part of SRAM
5. Main memory (DRAM)
    - 1-16GB
    - 100ns access time
6. Hard disk/SSD
    - 100-1000GBs

- 10ms access time

8.1 Addressing

Addressing is when we want to access memory, we need to specify which memory address to use. For example: MEM[10] means to access memory address 10. Ideally, we could specify a memory address directly every time, but that's not always possible. Sometimes, we want to specify a sequence of addresses. Therefore, we have invented a load of different ways to specify a memory address.

8.1.1 Immediate addressing

Immediate addressing is when data is supplied **in an instruction**. There is no real memory address and all information is embedded in the instruction. Also, data is immediately available. It's really fast and simple (the simplest). An example looks like: rl $<-$ 42.

All of the information is embedded in instruction, so it's predictable. This makes it really fast. It's pretty easy to understand and it's good for optimisers to analyse. Unfortunately, in the words of the mighty Dan Page, there's no free lunch. There is a lack of flexibility and it's gotta be inserted statically. There's a limited range of instructions (seeing as it's limited by the permitted number of operand bits in the opcode).

8.1.2 Direct addressing

An instruction like: MEM[10] is pretty cool, but how is it formed? It's formed in the kind of format: Operation — Operand1 — Operand2. For example 6, 10, 42. The exact memory address used is embedded in the instruction. This is known as **direct addressing**. The exact memory address used is embedded in the instruction.

Direct addressing has the same pros and cons as Immediate addressing 8.1.1. But, it's a little slower in return for a larger range.

8.1.3 Memory-indirect addressing

Memory-indirect addressing solves the problem of limited range by storing the address to be accessed in memory itself.

An example would be: MEM[MEM[42]] which means to go and look at the memory address in 42 and fetch the value. That value is the address to write the value in r1 to.

It's good because it's got a larger range and the source memory location for the address may be dynamically changed.

Unfortunately, there are some bad points. The first memory address is still statically compiled. The range restriction is just changed to the initial memory range. It's also slower than direct addressing.

8.1.4 Register-Indirect addressing

This method provides even more flexibility. It uses the register's value as the memory address: MEM[r1] ¡- r1.

There are loads of advantages to register indirect addressing like the memory address can be dynamically computed and the value does not need to be stored in the instruction thereby reducing code size. The register is internal to the processor so it's faster and more energy efficient.

This also allows for native support of pointers. Accessing indirectly is equivalent to a dereferencing operation, like *p in C.

8.1.5 Indexed addressing

Sometimes, you just gotta define a base address and access memory based on this. It's pretty useful for stacks, arrays, and caches etc. Indexed addressing extends indirect addressing to support this. We have a **base** and an *offset*.

Normally, the base and the offset are both stored in the registers, but this doesn't have to be the case. We get instructions like: MEM[r1 + r2] ¡- r3. Here, r1 is the base and r2 is the offset. Base and offset can be varied independently.

Many implementations support the base and offset construct natively. Architectures often have a dedicated register to help, normally called something like the stack pointer or the base register.

The stack/base registers may or may not be general purpose depending on the architecture. The offset usually comes from an additional general purpose register. An example of indexed addressing based on an array.

9 Compilers

We're going to be looking at an 8 bit machine and how we'd go about building a machine. Anytime you start with producing a computer or processor, we need to make an instruction set. How do we design an instruction set while keeping it simple?

9.1 Instruction set

Firstly, in any machine, we need some *registers*. We need one or two things to hold the arithmetic logic. The registers we have are:

- **PC** - The program counter
- **AREG** - Holds the arithmetic stuff
- **BREG** - Same as AREG

- **OREG** - Holds the instruction and the operand that comes with the instruction

These two registers are about the minimum that we can get away with. Now, we have two parts: the **function** and the **operand**. These are both 4-bits long, meaning that each instruction is 8-bits long. In terms of the instructions, there are three or four different classes of instructions:

- Load
  - LDAM - load from A
  - LDBM - Load from B
  - STAM - Store in A
- Load (constants)
  - LDAC - Load A with a constant
  - LDBC - Load B with a constant
  - LDAP - Load address in program: This means that we can supply an address of the program as an operand, and store it in a register, and this means that we can use it as a callback.
- Indirect loads
  - LDAI - Load A indirectly
  - LDBI - Load B indirectly
  - STAI - Store A indirectly
- Arithmetic operations and branches
  - ADD - Adds
  - SUB - subtracts
  - BR - Branch always
  - BRZ - Branch if zero
  - BRN - Branch if negative
  - BRB - Branch to the contents of the B register

We now have 15 instructions, so we have one more instruction: PFIX, which is a prefix. This means that we are able to encode a 16-bit value by first supplying the PFIX with the first half, and then the function with the second half. It attaches the operand to the operand of the next instructions. Even this instruction set is able to do some quite complicated things, such as actually be a compiler.

The arithmetic units will be used for both doing the arithmetic and for computing the addresses. For example, it will have to add the PC to the operand etc.

Let's start with some registers. So, we have the registers that are attached to the arithmetic unit via some multiplexers. We also might need one of the values to address the memory. If we're reading data from the memory, then we need another multiplexer to be able to take values from either the memory or the ALU. We also need to be able to write to the memory, but because we've set up the instructions so that we can only write to memory from the A register, we only need to connect it to the AREG.

The rest of the machine can be seen online, so I won't be documenting it here because it doesn't really make sense to be described in words.

10 Hex Architecture

Having defined the instruction set above, we need to work out what we actually want to do with it. So what we have is some block of memory. In the case of our *simple machine*, we can split up the memory, working from the bottom, into:

- Jump (that jumps to the beginning of the program)
- The stack pointer. This initially points to the top of the stack. As we call the functions, we then *decrement* the stack pointer, so that we can get back to the place we came from. We can keep doing this with no problems.
- Global variables
- Constants
- Strings
- Program
- Stack

10.1 The stack

As we lay out the information in the stack, when we do a procedure call, we execute some instructions that load the return address into the AREG. This means that the stack now holds the address that is stored in the AREG, and then we decrement the stack pointer.

We might also want to pass some parameters to the function, and these are also stored in the stack (right above the return state value). We can still use the same instructions to access the incoming parameters because we know the offset.

Let's look at the following example:

```
LDBM 1
STAI 0 'bottom location of the stack
LDAC -5 'The offset required for the locals/variables etc.
OPR ADD
STAM 1
```

```
LDAM 1
LDAI 6
BRN L207
BR L206
```

So, we've jumped to this function. Now, we load the stack pointer that is in memory location 0. We then load -5, which is the space required for the local variables of the function. We then ADD the two together, and then place the result back into the stack at memory location 1. Now, we're loading up one of the incoming parameters, and then testing to see if the result is negative or not.

The actual code of the stack use is that we use:

```
LDAP
BR F
R:

F:
```

LDAP is used because it stores the program location into register A. Next, it branches to some function F, with the return value stored in the A register. After this, we just decrement the stack pointer in the same way discussed above. This can be done recursively, and then we can just read back to get the same order as before.

We have a couple of things that we need to be introduced to. There is something called a **lexical analyser**, which takes all of the incoming symbols and converts them into a signal, and something called a **syntax analyser** which converts the symbols into a kind of data structure. After it's made into a tree (the kind of data structure that the compiler prefers), it looks to simplify it first, and then it solves it. For example, in the code:

```
if c then p else q
```

We end up with a data structure that looks like:

```
| if | c | p | q |
```

10.2 The translation process

When we translate an if statement, we don't know how long the condition is going to be, so we kind of do a rough input for the first pass, and then an algorithm goes over the code afterwards to input the necessary offsets and such.

When a name (of a variable) is read in, it's looked up in a name table, which is essentially a hash table. The very first time it's read in, it's input into the name table, but every subsequent time, it is just pointed to the memory address associated with the name. We also have to look up all of the things like 'if', 'then', 'else' etc. in the same way, so we have to hard code those in.