Introduction to Computer Architecture: The (almost) Comprehensive Notes

Josh Felmeden

May 24, 2019

Contents

Part I.
Maths and Physical Parts

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:

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

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

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

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.

---

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

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 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:

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

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:

$$r = \;\; x_0 \wedge x_1 \;\; \wedge \;\; x_2 \wedge x_3$$
$$= (x_0 \wedge x_1) \wedge \; (x_2 \wedge x_3)$$

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 $0 \le j < n$, 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:

$$co = (x \wedge y) \vee (x \wedge ci) \vee (y \wedge ci)$$
$$= (x \wedge y) \vee ((x \oplus y) \wedge ci)$$
$$s = x \oplus y \oplus ci$$

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:

$$x_0 = \neg x_0' \wedge \neg x_1'$$
$$x_1 = x_0' \wedge \neg x_1'$$
$$x_2 = \neg x_0' \wedge x_1'$$
$$x_3 = x_0' \wedge x_1'$$

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**:

$$S_0 \mapsto \langle 0, 0, 0 \rangle$$
$$S_1 \mapsto \langle 1, 0, 0 \rangle$$
$$S_2 \mapsto \langle 0, 1, 0 \rangle$$
$$S_3 \mapsto \langle 1, 1, 0 \rangle$$
$$S_4 \mapsto \langle 0, 0, 1 \rangle$$
$$S_5 \mapsto \langle 1, 0, 1 \rangle$$

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

$$S_0 \mapsto \langle 1, 0, 0, 0, 0, 0 \rangle$$
$$S_1 \mapsto \langle 0, 1, 0, 0, 0, 0 \rangle$$
$$S_2 \mapsto \langle 0, 0, 1, 0, 0, 0 \rangle$$
$$S_3 \mapsto \langle 0, 0, 0, 1, 0, 0 \rangle$$
$$S_4 \mapsto \langle 0, 0, 0, 0, 1, 0 \rangle$$
$$S_5 \mapsto \langle 0, 0, 0, 0, 0, 1 \rangle$$

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:

$$S_0 \mapsto \langle 0, 0, 0 \rangle = 000_2$$
$$S_1 \mapsto \langle 1, 0, 0 \rangle = 001_2$$
$$S_2 \mapsto \langle 0, 1, 0 \rangle = 010_2$$
$$S_3 \mapsto \langle 1, 1, 0 \rangle = 011_2$$
$$S_4 \mapsto \langle 0, 0, 1 \rangle = 100_2$$
$$S_5 \mapsto \langle 1, 0, 1 \rangle = 101_2$$

And capture:

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

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!

Part II.
Digital Logic and Computer Processors

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. Memory Hierarchy

The ideal computer and the ideal memory have slightly different definitions. The idea computer will have:

- Zero-cycle Latency instruction execution and data supply

- Infinite capacity instruction execution and data supply

- Zero cost instruction execution and data supply

- Perfect Control flow

- Infinite bandwidth data supply

The ideal memory will have:

- Zero access time

- Zero cost

- Infinite capacity

- Infinite bandwidth

Unfortunately, bigger memory means that it takes longer to determine the location. Faster is also more expensive because of the memory technology. Also, higher bandwidth is more expensive because of the increased need for ports, banks, and higher frequency.

9.1. DRAM

One bit of data is stored as a charge on the capacitor. It also has one transistor for reading or writing. It loses charge when read and it also loses charge over time

9.2. SRAM

This consists of two cross-coupled inverters that store one bit. Six transistors are needed for this (two for each inverter and two for the read/write). It retains the memory if it's powered so there's no need for a refresh.

9.3. Memory bank (generic)

A memory bank is a 2D storage array of memory cells (made up of either SRAM or DRAM cells). To access a cell in the bank, we decode the row from the instruction, and activate the lines. Next, we decode the column address and select a subset of the row via the bit lines. Finally, we amplify the row data and send this as output.

9.4. Comparisons

Here is a table showing the differences between SRAM and DRAM:

SRAM	DRAM
Volatile	Volatile
Faster access (no capacitor required)	Slower access (capacitor required)
Lower density	Higher density
Higher cost	Lower cost
No need for refreshing to keep the data	Refreshing is mandatory
Can be used for logic	Capacitors are bad for logic operations

Here is a comparison between hard disk against flash drives

Hard disk	Flash drive
Non-volatile	Non-volatile
Data stored as magnetisation patterns on circular disks	Data stored in floating gate transistors
Mechanical (has moving parts)	No moving parts
Slower	Faster
Cheaper	More expensive

9.5. Locality

Locality just means that things that are nearby are related. For example, things that just happened give a pretty good estimate for the future. Additionally, things that are close together are normally related (such as people you sit close to are normally your friends).

This is the same in memory:

- **Temporal locality** is a typical program that accesses the same memory location repeatedly for a short time.

- **Spatial locality** is a program that references a cluster of nearby addresses. The instructions tend to be in sequence and arrays or data structures are stored in nearby memory locations.

9.5.1. Caching

Caching exploits both temporal and spatial locality. Recently accessed data is automatically stored in a fast memory because the typical access time is much faster (since the memory is physically closer). Similarly, information that is stored in memory addresses that are *nearby* recently accessed addresses mean that typical access time decreases because a program is likely to want to access the nearby memory.

10. Instruction Set Architectures (ISAs)

No, an ISA is not one of those bank account. Actually, it's a contract between the software and the hardware. More specifically, it's the definition of the instructions that are supported by a particular processor. The ISA specifies the exact behaviour of instructions fed to the processor. The ISA specifies the exact behaviour of instructions fed to the processor. The ISA should be designed sympathetically for the programming language that will write code for it.

An ISA includes (but is not limited to):

- Op-codes

- Data types

- Registers

- Memory addressing modes

- Memory architecture

- Interrupt and exception handling

- I/O

An instruction is a definition of a unique operation code as well as some contextual information. For example, the instruction '1 01' could be decoded with:

- 1 As the op-code (move in this case)

- 01 as the argument. Here, we have the constant immediate value 1.

10.1. Decoding

All instructions need to be uniquely decodable. It'd be outrageous if some strings decoded to two different things. Some ISAs don't defined all of the possible bit-strings, which is allowed, but can cause some problems if an undefined instruction is stumbled upon.

10.2. Creating Instructions

So, to create an ISA, we need to have a list of instructions and their possible arguments. The actual purpose of the ISA will influence which instructions make it to the final set of instructions.

Most instructions naturally fall into one of these classes:

- Arithmetic

- Comparison

- Memory

- Control flow

Simple processors only need those 4 classes, but some high performance ones might employ:

- Vector processing

- Digital signal processing

When we select the instructions to be a part of the ISA, we need to think about the tasks in hand. There are two categories: RISC and CISC:

10.3. Categories

10.3.1. RISC

RISC is the reduced instruction set computer. The aim is to produce a minimal set of instructions that are fast and lightweight to execute. Each instruction should hold a similar runtime, and complex instructions should be made up of a few simple ones. Normally, the instructions in the RISC model have a fixed length. RISC machines run fast in general, but they need a lot of instructions to do a complex job.

10.3.2. CISC

CISC is the complex instruction set computer. They are pretty much the opposite of a RISC. If there is a specialised operation to be done, then add a dedicated instruction. Because there are a lot of instructions that are complicated, the programs are smaller. The processor is normally quite complex and slow unfortunately, along with mixed length instructions which leads to unpredictability and compiler nightmares.

10.4. Options

There are a few options for ISAs to take into account:

10.4.1. Instruction choices

For many operations, you may have a set of choices (such as `r0 <- r0 + r1`). How many ways can we do this?

We could have multiple accumulators ACC1 and ACC2 such that:

1. `ACC1 <- r0`

2. `ACC2 <- r1`

3. `ACC1 <- ACC1 + ACC2`

4. `r0 <- ACC1`

This only takes four very simple instructions. Another option would be to use a stack:

1. `PUSH [r0]`

2. `PUSH [r1]`

3. `ADD`

4. `POP [r0]`

Where r0 is assumed to be a memory address. We could even simplify it down to three instructions:

1. `ACC <- r0`   : `MOV ACC, r0`

2. `ACC <- ACC + r1` : `ADD r1`

3. `r0 <- ACC`   : `MOV r0, ACC`

Or even two:

1. `MOV ACC, r1`

2. `ADD r0, r0, ACC`

How can we possible choose between all of these possibilities? Well, we kind of just need to take into account all of the tradeoffs.

Here is a comparison of the tradeoffs:

Tradeoff 1: More instructions → more bits	Tradeoff 2: More operands → more bits
More instructions, greater information	More bits, more work
More space, more energy	More space, more energy

The choices of instruction specificity really impacts how a machine might be programmed. It can imply a choice of paradigm, and here are some examples:

- Implicit source and destination → stack machine

- Implicit destination → Accumulator machine

- Explicit source and destinations → register machine.

Now, we need to cover the useful space of execution needs. This means that we need no repetition, no coverage where it's not needed, Minimum size to get the job done, check if the instruction inclusion is harmful (Amdahl's law).

10.5. Amdahl's law

If a new instruction is added to speed up an operation by s times, this law gives the maximum overall speedup:

$$\text{speedup} = \frac{s}{1 + f(s - 1)}$$

Where $0 \leq f \leq 1$ is the fraction of time spent NOT running the operation in question in the original design.

10.6. Encoding instructions

Now we've selected our set of instructions, we need to encode them *unambiguously*. A common way to separate them is to do it into **op-codes** and **operand** areas.

Space is a major factor in deciding the instruction encodings. Ultimately, anything can be improved at the cost of it taking more space. The instruction lengths can vary, too. Some instructions need to provide more information than others and therefore it'll be a longer instruction. If an instruction can be shorter, then that's good. Variable length instructions can save space too.

Unfortunately, it's not all plain sailing to use variable length instructions:

Pros	Cons
Size efficient	Almost infinite amount of possibilities causing complex decoding
Can be extended to support more operands	Pipeline slowdown
Optimal in some sense	

In comparison, fixed length instructions might be sub-optimal. They also might limit the length of immediates. The major advantage is that it has a simplified decode stage. It also tends to be a winner in most modern designs.

Part III.
8-bit computer and Software Applications

11. The simple computer

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?

11.1. 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.

12. 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

12.1. 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 |
```

13. The general compiler actions

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.

On a reading of an 'if' keyword, we have to skip over the symbol so that we're clear of the keyword, and then we start reading the condition. On top of that, we look for a 'then' (recursively). If we don't find that, there's an error. At the end of the whole thing, we return a data structure as discussed before of the 'if'.

If we have a global variable AND a local variable defined as the same name, we obviously want the compiler to only mess with the local one if it's in the local function. This is part of the **scope rule**. After this, we need to try and *optimise* the code, such as if we see '0 + x', we could just make it 'x'. Similarly, if we see '3+5', we can just convert this into '8'.

13.1. The scope rule

When we declare a name, we define a point in the stack that has both the name, and the address in memory. This amount of memory is known as the **offset**. The part of the compiler that deals with the declaration of local variables is fairly self explanatory for the parts discussed previously. When we encounter a name in the middle of the expression, we go down the stack, and either find the name and return the offset of the stack with the information of the name, or we don't find it and return an undefined error.

13.2. Optimisation

The optimisation process does more than just optimise. It also converts all of the relational operators into simply an equals to or a less than comparator (because the computer is only able to compute less than or equals operations). In the case of finding a return node, it will replace it with an optimised

return node. If we encounter a name, we see if we can find a value, and then just return the value as the result. Additionally, we see if it was a constant, or a computed value, so then we can convert the tree node into a value. What we're doing here is *pruning* the tree node.

If we find an '=' symbol, then we can do a few things:

- If we have '!x = !y', then we can just replace this with 'x = y'

- If we have a '!=', then not only look ahead to see if they are both notted (as before), we convert the node into an equals node, and then insert another node to not that, so that we can just deal with equals signs.

If we find a 'greater than' symbol, we switch around the code, so that we always get the 'less than' symbol. There are a lot more things that we can do, such as using DeMorgan's law to simplify things, remove the 0 operands in addition or ORs etc.

Having all these sorted out, we're now in a position where we can start to translate the code into machine code that is actually executable.

13.3. Translation into executable

13.3.1. Control structure

The steps required to translate the code into executable operates much in the same way as that of the optimiser; often recursively calling itself and so on. It has parameters that keeps track of what to do when the statement is finished. For example, if we have a piece of code that says:

```
if (c1) then
    while c2 do p
else
    q
end if
```

This would look like this in machine code:

```
c1
BRZ ELSE
START:
c2
BRZ END
P
BR START

ELSE:
    q
END:
```

There's a function called 'generate statement', that has a few flags:

- One that has the incoming part of the tree

- A flag that tells you whether once the statement is compiled, if it flows straight through, or output some control

- A flat that tells you where the control needs to go (if there is one)

It compiles itself recursively if it is given a semi-colon list. The interesting part of this is that there could be something where the else part is empty, in which case the compilation is simplified. We generate some labels for the 'if' and the 'else' parts, and then compile the parts of code.

For the 'while' code, we set the label for the beginning of the loop, set up the condition, and then the code, with a branch back to the start. This is how control structures are generated.

Let's look at another example:

```
if (x AND y) then
    p
else
    q
end if
```

We'd get:

```
x
BRZ ELSE
Y
BRZ ELSE
p
BR END

ELSE:
q
END:
```

We don't even need to do any actual logic, because we just check each operand in turn, and then branch to the else if either of them are 0.

For the less than function, we just check if one of the parameters is zero. If not, then subtract one from the other and branch if negative.

The function calls are a little complicated. If we have the code:

```
P(x,y,z)
```

The problem is that the parameters might be quite complicated. For example, there could be procedure calls embedded in the arguments, so we have to go digging around trying to find the code. The compiler has some counters to this, such as to look inside the parameter list to find any embedded functions and call those first, before generating the rest of the instructions.

13.3.2. Assigning variables

When we do a simpler assignment (such as name = othername), we have a left hand side and a right hand side, we just check the value of one, and then assign it to the other one.

If we have a more complicated one, such as a subscripted (a[i] for example) assignation, then we have a **base pointer** and an **offset**. We check to see whether the offset is a value, in which case we just generate an expression for that. However, we might be using an index to iterate through an array. In this case, we need to generate some code to form the address, and then use the address to store the value. Because we have so few registers in the machine, we need some code that will get the base, and the offset, and then store the data in the stack. After this, we have some code to store it in the appropriate place.

If we ever run out of registers, the compiler just slaps them on the stack.

13.3.3. Translating expressions

This final step just handles the logical operations, the function calls and other such things like that.

13.3.4. Code buffer

As seen before, when we have an 'if' statement, we're going to have a 'branch zero' to some else tag, and then do the statement. But, because we don't know how many lines of code we need to process before placing the 'else' tag, we need to put the label into the buffer. The buffer then fixes up the offset. The offset might be longer than the 4 bits that we get for the buffer, so we might also need a prefix. Jesus, who does this?

14. Bootstrapping

Bootstrapping is where you write a language in its own language. It originated in something like 1960. When we move a language from one instruction set to another, we need something called *ocode*.

When we write a compiler, we write them in a suitable source. So, we have a compiler that is written in language x, that compiles in language y. Now, we will also need something that compiles language y and so on. But, with bootstrapping, the source and the executable are the same language, so things are good and alright.

At some point, we might want to stop writing our source in a language, so we write a new compiler, and then we get a new language, and eventually, we can push out the old language entirely, and this is how you make a new language and it's compiler in bootstrapping form. If we didn't have C, this would be really hard, because we'd have to go right down to machine code, or a macro generator (one that kind of brute forces it). Notice that because we're going to throw away the old language, it doesn't matter how inefficient it is, as long as we get to the right destination.

The process of adding new features is pretty much the same: we create a 'new' language, which is just the old language with added bells and whistles, and then apply that to the compiler, and then away we go.

Let's explore an example of this. Say we wanted to add another 'special character' (the likes of '\n') then we make a compiler that recognises the new one, and then use that in recognising the new code

itself.

If we wanted to add a new key word (for example 'until'), then we need to declare the new keyword, and then add a line in the list of ones that recognises the key word, and then pretty much just copy the while code, but have a NOT at the end of the condition.

14.1. Optimising a language

If we want to add some optimisations to the language using bootstrapping, then we need to do the following.

- We have the source code S_i and an executable E_i. We compile the source with the executable to produce the same thing: $E_i = E_i(S_i)$

- We want to produce the optimised source (S_{i+1}), and this will be compiled with the same executable: $E_i(S_{i+1})$, this produces something else that won't be the same as either, so we'll call it O_i because it can optimise. However, it hasn't had the optimisations applied to it.

- We now need to apply the optimisations to the compiler itself: $O_i(S_{i+1}) = E_{i+1}$.

- Now, we're left with $E_{i+1}(S_{i+1}) = E_{i+1}$

Sometimes, the object code is smaller, even though the source code is bigger because of the optimisations. Conversely, it might just make the code a bit worse, and you just won't know until you've tried it.

15. Garbage collection

Garbage collection (and memory allocation) can get pretty hard to understand, and therefore we have some techniques to automate this. The techniques aren't new; they've been around for ages. It was originally made for some random language, but now has been adopted for a few other ones.

We have something called a *heap* with some globals underneath it, then the stack, and then the program and some constants. What we want to do is automatically allocate some memory in the heap when the program needs it, and then deallocate it when the program is done with it.

We might write something like `x = [1,2,5]`, and so the garbage algorithm will now have some objects:

1. Size

2. Flag (used to mark the elements in the heap that are needed)

3. Address

The next step is to identify which parts of the heap are needed. What we want to be able to do is to go through the region and mark all the objects in the heap that are pointed to by the items in the

program (by things in the stack really). In other words, we want to know which items in the stack are pointing to elements in the heap.

There are a few options for doing this. The compiler might leave a map for the variables that are used, but sometimes, the language doesn't do this. Another method is just to guess. Finally, we could just explicitly mark the data. Hardware designers would be really cross with you for this because they'd need 65-bit designs now.

What we're going to do is steal a bit from the words in memory. Unfortunately, you lose some functionality. Sometimes, the bottom bits aren't even used in memory addressing, which sounds kind of weird. When we do this, it means that every time we add up, we need to subtract 1 from the result because the final bit is used for the flag, but this means that we need to change the compilation.

Now, we step through the stack and see which items are pointing to the heap. When we find one, we set the flag in the address. Not only this, but we need to set those items that the heap memory points to (multiple pointers and whatnot).

To make space available, we have a pointer in the heap, that goes from bottom to top. To avoid fragmenting it, we move the objects down so that they're **compacted**. BUT, if we just do this willy nilly, then we'll run into errors because all of the pointers will be messed up and there's no easy way to find out what items are pointing to a given entity. So, there's another step that we need to consider.

15.1. Compacting

To move the items down, we need to consider the items that are pointing to a given entity. To do this, we start at the bottom of the stack. If an item is not *marked* (i.e. doesn't has a flag), then we skip over it. We know how much to skip by because the size is at the bottom of the entity. If it is marked, then we store the address that it will be moved to at the bottom. We're allowed to do this because there's a spare word in each data item. When we find another item, we do the same; recording the final position of each item, until each marked item has a destination recorded

Having done this, we can return to scanning the stack. When we step through the stack and identify the pointers to the heap, we can now update the pointers in the stack to the 'final address' information that we recorded before. We have to carry on this process through into the globals, and also the heap itself.

Finally, we can go back once more to the stack, and perform the copies. The items that are not marked, are replaced, and the old marked locations are unmarked, ready to be overwritten.

This is a fairly simple collector, but there are simpler ones because this one can deal with arbitrary sized data.

15.2. Marking

We're not done with this yet, however. When we mark an object in the heap, we then also need to mark the items that the new marked item is pointing to (else it will be erased). The most obvious one is to recursively mark the items. This is not a good solution to the problem, unfortunately. This is

because to do something recursively, we need a stack, which uses some memory. The issue with this is that we need to allocate a relatively large amount of space for this stack.

An alternative position is to iteratively mark the items (go to the items that the thing points to and mark it. Rinse and repeat). This would take so many iterations, which means that the performance take a hit, rather than the size (as with the recursive method).

Thankfully, there is a solution that isn't huge or slow. The way that we're going to do it is to implement some more pointers: one to **current** and one to **next** and one to **previous** . When we want to move down a level, we update the *current* pointer to the new item, the previous to the current, and the next to the following item.

Most of the time, we do a combination of the above (such as using iteration to a certain depth, and then using the third one). This third one is called the 'stop the world' algorithm, because when it runs out of memory, it stops.

Garbage collection is really hard because there are lots of points in which things can go wrong, such as the program trying to access the data as the collector is moving the data. This can be solved using locks and so on, but you get the point: garbage collection is real hard.

16. Communication, Interrupts, and Protection

We need some means by which two independent devices can communicate. So far, we've looked at the two phase clock. However, when we start communicating between computers, they each have their own clocks, so we need some kind of encoding standard so that the data can be read. The traditional method is the 'handshake'. This means that computer A sends out a request, computer B confirms the request, computer A confirms the confirmation, and then data transfer can continue.

In layman's terms, the sender puts the request on the data wire. We assert the request to confirm that the data is actually valid. Now, the sender waits until the receiver puts the acknowledge wire on. The sender can then take off the request from the data wire, and then the wire can change state.

There are a few optimisations that can be done to this, but this is called the 'return to 0' scheme. A faster way of doing this is to do it on the edges of the changes, to make it faster. It almost halves the data transfer rate, but it's still not perfect because it's impossible to guarantee the transmission speed of the wire, because the receiver might see the request at a time when the data becomes invalid.

An alternative to this is that we have three signals: REQ0, REQ1, and ACK. If we want to request a 0, we make REQ0 1, and wait for the ACK to be 1. Next, we might want to request a 1, so we make REQ1 1, and wait for ACK to change state (remember that it was 1 so now we wait for it to become 0). We might then want another 0, so we change the REQ0 to 0 once more, and wait for the ACK to change state. This repeats. This is a good method because it doesn't matter how long it takes for the signal to reach because we are only waiting for the changes in the signals.

Did you want another method? No?? Tough. This one has four wires for transmission. Now we have 2 bits of transmission. You could EVEN have 6 wires, and once 3 change state, the recipient responds. This means we can encode 20 values. These two aren't hugely used, but some psychopaths do use it.

16.1. Transferring data

When we communicate between devices, we need a way to transfer the data. When a device receives some information, it has to take it into some flip flop to be able to understand the data. Unfortunately, the flip flop COULD change state at the exact same time as the clock signal, which would make it enter the meta-stable state. As long as you wait a while, the flip flop will stabilise, but if the circuit isn't designed appropriately, then the computer will fail (a mistake that has actually been made in the past).

Now we know how it works physically, we need to know how to code these. We could of course have another register (or a special memory location) that behaves just like any other memory location, but the memory is solely reserved for data transfer. Normally, we have a system to allow the program to transmit data (or wait to communicate) while the program is still going (asynchronously), and this is usually achieved through interrupts.

16.2. Interrupts

The basic idea is that the program tries to communicate some data, and then the interrupt occurs once the transfer has completed, meaning that the program is able to continue supplying data.

In order to make an interrupt system work, we need to transfer control on some specific address. This is kind of like a subroutine call, but it can happen at any point in time. We also need to restore the position we were at. So, unlike a subroutine call, we don't always have the location of the program, so we need to save the state automatically.

Because of this, we have another register (where the PC is transferred) before the control is transferred to the desired device. On top of this, the address for the control to be handled to is stored at the bottom of memory and they are called '**interrupt vectors**'.

We have another thing called an 'in kernel' flag. The *kernel* is the innermost part of the operating system is called the **kernel**, and the flag denotes whether or not we are in the kernel (or handling an interrupt in other words).

If we are trying to transfer some data to the buffer, and then we get another interrupt to get some data from the buffer, we might lose some data. We therefore need some safeguard to stop the interrupts getting data from the same buffer once the interrupt is received.

An optimisation of this can be used and this is quite important. If we have some memory system, and some processor able to access the memory (as well as some I/O), then when we transfer data from the I/O, the process is very similar to what we discussed above. Somewhere in memory space, we want to create a situation where the user cannot corrupt the memory. We have some system that protects the program from overwriting the kernel (so that we have something left if a program crashes). The kernel also does some scheduling (like the OS does).

16.2.1. Entering the Kernel

One of the earliest protection protocols to protect the kernel was to just split the memory in half and then the programs can only access the bits in memory that won't corrupt the whole operating system. If we are operating in the kernel, then the 'inK' flag is set, otherwise we need to set the region of memory that programs are able to access. These limits need to be stored within the kernel so that the program is unable to edit the limit themselves and therefore access the kernel. There will be some code that kinda looks like:

```
if inK then
    mem[oreg] <- areg
else
    if oreg < as then
        mem[ab + oreg]
            <- areg
    else error 'Outside the region (inside the kernel) and therefore error
```

If the program was to fail in some way (such as dividing by 0), then we might have some entry into the kernel, and the kernel will probably have some idea of why the kernel is being entered. The error reasons might also be encoded in such a way that we know what the error is. In the same way that the interrupt request uses a vector, we get this in the same way.

In addition to device drivers, a kernel normally allocates memory and I/O devices to application programs. It also shares the processors time between several programs. This is known as scheduling.

16.2.2. Types of scheduling

Some of the interrupts that might occur are:

- **Timer** - When this occurs, we update the current time, and see if there's any consequence because of this (does a program overrun its timeslice etc). Also, some programs request to be woken up at a certain time.

- **Input interrupt** - This means that a program might want to store some data, so it puts the data in the buffer, and this is what this is for.

- **Kernel case input** This means that if the buffer is not empty, it will hand over the data to the program, or if it's empty, it will deschedule the program, and reschedule the next one.

- **Output Interrupt** - This will try to take something out of the buffer.

- **Kernel case output** - Almost the same as above.

- **Page Fault** - We have pages as the unit of transfer between the memory, and if the program tries to reach a page of memory that doesn't belong to it, then we get a page fault.

- **Kernel calls** - Loads of these

- **Errors** - Divide by 0, other such issues.

Due to the inK flag, one of these interrupts cannot happen at the same time as the in kernel processing. Because we have interrupts, it doesn't hold up the program for long and it can continue to be processing other things

17. Memory systems

In our simple machine, pretty much all of the instructions executed at the same time, but as the transistors get smaller, they get faster, but unfortunately, the wires do not hold this same trait. Therefore, we need to try to mitigate this effect. Accessing memory is all about communicating data.

A *memory array* is a large collection of cells that are connected to a buffer. If you did nothing to combat the latency, then the instruction execution time would be 30% waiting for the data transfer. There are a few methods to combat this, and one of them is minimisation hiding. This means that when the computer is waiting for something to happen, then it enables other things to happen in the background.

17.1. Latency hiding

Latency hiding is achieved by a hierarchy of memories. When we look at the CPU, we have some registers (that can be accessed very fast). We also have a cache, which can hold a moderate amount of data. This data can be accessed within a cycle or two, and it's organised so that it can take data from memory when asked, and write it back when it's done. It's also pretty close to the memory. This is the level one cache. After this, there's a level 2 cache, and there might be some more caches. A level 3 cache is shared by all of the CPUs, whereas the level 1 and 2 are individual to the CPUs. After this, we leave the chip, and then we have the larger memory, but we won't worry about this.

When the data is fetched into the respective caches, it has to get passed down through the caches. It may turn out that the caches fill up, and there's a method to migrate data from the high latency memory to the lower level caches. Ideally, we want the majority of these to be doing the read and write to be using stuff from the L1 cache.

17.1.1. Direct map

What happens is that we have some lines in the cache, and lots of lines in the memory. We have 2^m lines in memory, and there are 2^c lines in cache. The cache line has words of data (maybe 4) corresponding to the memory address, a tag (which is used to determine which line it's currently holding), one bit called *valid* (which sees if anything is in the line) and one called *dirty* (which means something has been written to it and therefore it needs to be written back into main memory). This method is known as a direct map cache. The issues we have is that it's possible that the cache can just be continually overwritten. Programs also access a lot of different locations. It might turn out that the way the memory is distributed, the data is really far apart so then we suffer performance drops. This scheme is pretty cheap and cheerful, but it's rarely used.

17.1.2. Associative

Here, we are allowed to put any line of the memory in any line in the cache. Now, the tag part has to be bigger. The disadvantage of this is that we don't know what to get rid of when we want to get a new line into the cache. We can use the 'least recently used line' to get rid of that, but it's really hard to know which one is used last because it's so low level (it's really expensive). The strategy normally used is that there's a random line deleted. Again, this isn't really used.

17.1.3. Set associative

This is the most common technique used. There is a set of lines in the cache, each has a number of lines and then we have a mapping scheme. It's like a direct map cache, because by looking at the line, we know what set it's going to go into. The issue of deleting is the same as the direct map, but we delete a whole set. Because of the 4 or 8 lines, we lose the risk of having a collision.

17.2. Paging

A page has a header that is split into two halves, the Page address and the Offset. There is also a page table, and the page address corresponds to a place in the page table. The page table gives a mapping into physical memory and then the offset corresponds to a word. It's possible that we address something that doesn't have an address because it looks as though the memory is contiguous (but it isn't), and then there could be no physical address. When this happens, we get a page fault (which causes an entry to the kernel). What the kernel then does is that it finds where the page actually is, allocate it as a new page, update the translation, and then return control.

This is used so that programs that are too big to fit in memory can still be run. This can possibly slow everything down. The page table is pretty big you know. Also, there's two accesses to memory which therefore slows it down. Because of this, we implement a new buffer for the page table under the assumption that when we access something, we'll probably want the surrounding data too. This buffer is called the **translation location buffer**. This was discussed above in locality (9.5).

18. Parallelism

Parallelism is the answer to the slowing down of sequential machines. We are reaching a limit of the performance of the machines that work sequentially, and we are discovering new techniques for parallelism.

Pipeline is another technique that is possible. If we have a process that is able to be broken up into steps, we take data into one end, and then process them at each operation; passing the result onto each subsequent process. Pipelines turn up in a number of places. If pipelining isn't enough, we can do something like parallel pipelining, or stream processing.

In movie rendering (or any graphics rendering) we can use large clusters with a main node in the middle, so that it can distribute the tasks to one of the nodes in the cluster, and then they return as

and when they finish their task, only to be handed another one later.

When we are trying to model something physical, we can normally break it up into a load of pieces. Let's take the example of graphics processing. We can break up all of the processes, and make each processor (if there are multiple) do a little bit each. This is known as **domain decomposition**. Of course, you can use any of these ideas in combination.

It is important to note that each of the parallel techniques will then be running sequentially. The most straightforward method to speed this up is to pipeline, because each instruction happens one after the other (the fetch execute cycle), and so we reuse each result into the next instruction. It could be that the instructions are not sequential, and therefore there is no need for a sequential model, and we can parallelise this.

18.1. Super scalar processing

Normally, there are four stages, *fetch, decode, execute*, and *write back*. You can do these one at a time, or we can use a pipelining method to do a lot at once. If we look at the processing of 4 instructions (where each space is a timeslice):

```
1: F D E W
2:   F D E W
3:     F D E W
4:       F D E W
```

This sounds nice, but there are some snags. If we have the simple pipeline, then the operand of the first instruction would not be able to be accessed by the second instruction because it hasn't been written yet, so the pipeline is just waiting for a while so it's a waste of time. A technique to avoid this is called **forwarding**, where it becomes possible to feed the results forward into the next instruction.

Branches are the issues we face because if we start executing instructions before we know whether they will be executed is stupid. Unconditional branches are easy to deal with because we know that they'll We also have things called **branch predictors**. If we have a branch if true branch, then it's probably a loop, and it almost always loops back, so the branch predictor predicts the branch to go back up to the top of the loop. There are some quite complicated branching predictors nowadays.

Another method of parallelism is that we fetch four instructions at the same time. Each of these instructions is then decoded and sent to execution units (stored in reservation stations before hand). Somewhere, the register file is accessed to learn what operands are needed. The instruction is checked to see whether the operand exists yet, and if it does, then it continues on to the execution unit. They may come out in a completely different order, so we need something called a **reorder buffer**, so that they are sent back to the registers in the right order. Sometimes, the reservation stations might be incomplete, because one of the sources haven't been computed yet. What happens is that once the items come out of the bottom, we see if any of the destinations match any of the waiting operands, and if they are, they are fed in there straight away.

18.2. Concurrency

A common arrangement of the caches are that they have a shared L3 cache (while each processor has their own L1 and L2 cache), so that we can share all of the updates among all of the processors (known as concurrency).

18.3. Single instruction, multiple data

Single instruction, multiple data is a technique where we have a collection of execution units, and they all have some memory they can access, but they have a single piece of control logic that distributes the instruction through all of them, so they all perform the same action. The same kind of idea can appear in other forms. In the 1990s, they built this into the heart of the processor. If you have a 64-bit word, you can split this up into 4 16-bit words. Now, we just need some instructions to do some operations on the 16-bit words that would yield the same result as if we were to do it on a single 64-bit word.

If we are building a cluster, we can just have a big switch that sends stuff to the different parts, but it can be slow and not good when used in the context of high performance computers. The switch or network connects a bunch of processors (e.g. eight). This will produce $n \times (n-1)$ connections. If n is really big, it means that there becomes a huge amount of wires. This means that we really can't fully connect it (even though that would be really nice). Instead, we put some form of network. The network would be a collection of switches (some of the switches actually do connect the processors) and then some *edge* switches. It kinda looks like a CNN. It's pretty good cause it has $\log n$ efficiency.

18.4. Hyper cube

A hyper cube consists of the following:

- First order hyper cube is a dot

- Second order is a line

- Third order is a square

- Fourth order is one with eight nodes

- Fifth order is one with 16

And so on. Clearly