Introduction to computer systems, networks and security

Guido Tack, Carsten Rudolph

Introduction to computer systems, networks and security

Guido Tack, Carsten Rudolph

Contents

1 Introduction

This syllabus contains a series of modules that will introduce you to how modern computer systems work. We will cover a wide range of topics, such as basic electronics, the hardware components of a computer, how a CPU (the main processor) works, how the basic software manages the computer's resources, how computers communicate over networks, and what it means for a system to be *secure*. Clearly, we can only touch on some aspects of each of these topics, without going into too much depth. But at the end of this unit, you will have a good overview, and hopefully know which topics you want to study in more depth!

Have a chat

Let's take a common, everyday activity as an example: you pick up your phone and reply to a group chat (probably organising a meet up to study for FIT1047). There's so much going on under the hood to make this happen!

- The phone needs to turn your interactions with the touch screen into electrical, digital signals.
- These signals, which are just different levels of voltage representing zeroes and ones, are processed by digital logic circuits. Somehow, your message (and all other data) must be *represented* as sequences of zeroes and ones.
- These digital logic circuits are the basic building blocks for things like the main processor (CPU), graphics processor (GPU), memory (RAM), secondary memory (Flash storage), network interfaces (Wifi, 4G), sound processor, and other components.
- The CPU runs all the programs that are required to process your messages (and implement all the other functionality of your phone). This includes the *Operating System* (e.g. Android, iOS or Windows Phone), as well as the instant messaging app you're using. These programs are also represented as zeroes and ones (but of course nobody wrote them as zeroes and ones, we used programming languages like C, Java or Python, so how does that work?).
- The instant messaging app, together with the Operating System, communicates with the networking hardware to send and receive messages. How is it possible to exchange messages between, e.g., an Android and an iOS phone? Your phone can't directly communicate with the destination phone, it uses "*The Internet*", but what does that actually mean?
- Many instant messaging apps now offer *end-to-end encryption* to make sure your conversations are secure. How does that work? How can you make sure a message is actually from the person it claims to be from? How do you know your message wasn't tampered with or intercepted?

Don't worry if you don't yet understand all the steps and all the jargon above. Through the sequence of modules in this syllabus, we will explain them step by step. We don't expect any prerequisite knowledge, except some familiarity with using computers, and some curiosity for finding out how they work!

Syllabus structure

The modules in this syllabus are organised from the bottom up. We first cover the basics of the binary number system, which is essential for understanding how computers work. Binary numbers are closely related to Boolean logic and so-called *gates*, the fundamental building blocks of digital circuits. We will see how we can build complex logic circuits that can do arithmetic, store data, and execute programs, from these very simple gates. In order to understand how computers execute programs, we'll look at a

very simple programming language called *assembly code*. Now that we've covered the hardware and know how to program a CPU, we'll move on to the system software such as the *BIOS* and the *Operating System,* which provide the interface between the hardware and the applications we want to run. This concludes the overview of a *single* computer, and we can start thinking about how to make *multiple* computers communicate over a network. Finally, we'll discuss what it means to make such a system of computers *secure*.

How to succeed

The basic recipe for success here is to work through the reading material and answer the quiz questions before you attend the lectures. Each module has questions embedded that will help you deepen your understanding, before you take the graded Moodle quizzes. Some modules have pointers to additional material. It will always be clearly marked whether that is "prescribed reading" (i.e., we expect you to read it and it may be part of the assessment) or optional, just in case you want to learn more.

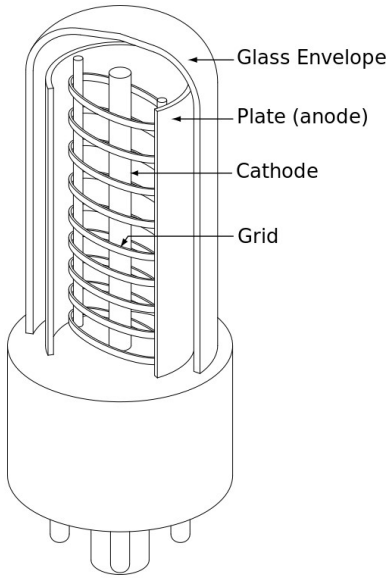
There are lots of practical exercises, which will help you reach a deeper understanding of the topics. It is absolutely crucial that you work through these exercises yourself, since many of the topics covered here can only be learned by doing! Your tutors will help you when you get stuck, and we encourage you to use the online forums and consultation hours to get additional help.

The learning curve may seem steep, but trust us - at least a basic understanding of these topics is fundamental to your IT degree!

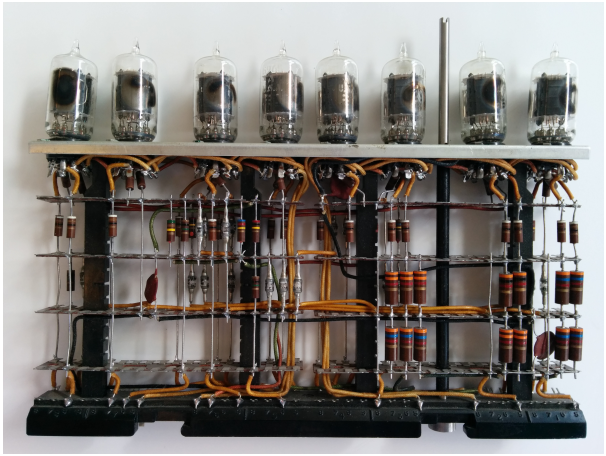
2
Representing numbers (and other things)

Why are the numbers 0 and 1 so important in computers?

The large majority of computers are based on the same internal concept. Everything is ultimately expressed using only two different states: **0** and **1** expressed as high/low current/voltage within electronic circuits. (Note that analogue computers and the idea of quantum computers is not covered by the explanations here). The smallest elements in current computers can basically imagined as switches that can be electronically turned on and off. First computers used relays (electromagnetically operated switches) or tubes as the core components. One example of such a vacuum tube is the *triode* as shown below in schematics and within a module of a IBM 700 series computer from the 1950s.



Structure of a Triode Vacuum Tube. © User:Ojibberish / Wikimedia Commons / CC-BY-SA-2.5



A module from a IBM 700 series computer, with eight 5965A/12AV7 dual triode vacuum tubes. © Wikipedia

User:Autopilot / Wikimedia Commons / CC-BY-SA-3.0

A triode can be used as amplifier or switch:

- Very small changes to the control Grid cause much larger changes in the electron flow between Anode and Cathode. A weak signal on the Grid is amplified. (Example: guitar amplifier)
- A large negative charge on the Grid stops the electron flow between Anode and Cathode. This type of vacuum tubes was used for computation.

Computers with vacuum tubes worked, but the approach did have a number of problems: modules with vacuum tubes are very large and they generate a lot of heat. Furthermore, tubes are not dependable. They have a tendency to burn out.

A revolutionary change to the way computers are build came with the development of *transistors*. The word transistor stands short for "transresistance". In principle, a transistor is a "solid-state" version of the triode. The solid medium is usually silicon or germanium. Both are semiconductors, which means, that they dont conduct electricity particularly well. However, by introduction of impurities into a semiconductor crystal (so-called *doping*) the actual conductivity can be fine-tuned. Transistors use layers of differently doped semiconductors in a way that it behaves similar to a vacuum tube, except that it is much smaller, more reliable, and generates less heat.

Thus, the smallest elements in the computer are basically switches and in the history of the development of computers it turned out that switches with more states than just ON and OFF were less practical. There were experiments with 5, 8, or 10 different states.

Thus, we have basically just 0 and 1 to express everything that we want to store and compute on a computer. The following sections look at the representation of numbers.

This video provides some information about the history behind the development towards binary in computers: https://www.youtube.com/watch?v=thrx3SBEpL8

Numbers

What do you think of when you see the following:

365

Probably the *number* of days in a year that isn't a leap year? If we analyse it more carefully, we see a *sequence* of three *symbols*, each representing a *digit.* The first symbol represents "three", the second "six", and the third "five". The *position* of the digit in the sequence also has a well-defined meaning, with each position being "worth" ten times more than the position to its right.

So 365 means three times one hundred plus six times ten plus five times one.

But we have made several assumptions:

- First, that the symbols mean what we think they mean (the numbers "three", "six", "five"). If you're interested, have a look at this Wikipedia article (https://en.wikipedia.org/wiki/Numerical_digit) to find out how

numbers are written in other writing systems.
- Second, that each position is worth ten times more than the next ("times one hundred", "times ten", "times one").
- And third, that the rightmost digit is worth the least ("times one").

This is what we call the **decimal system**. Generally, when you see a number written anywhere, you can assume that the decimal system is used.

What do you think of when you see the following:

000

In Australia, this is the national emergency phone number. It's also a sequence of digits, but it doesn't have any meaning as a number (e.g., you can't just dial **0** or **00** although, as a number, they would mean the exact same thing!).

The point we're trying to make here is that a sequence of symbols can be *interpreted* as a number (if you know the rules), or as something else (e.g. a phone number).

Modern, digital computers work with only **two different symbols**. That's because two symbols are enough to represent anything else, any number and any other piece of data. And it's reasonably simple to design electronic circuits that work with two symbols, by representing them as "high voltage" and "low voltage" (power on - power off). In the rest of this module, we'll see how to represent numbers and text using only two symbols.

Binary numbers

The binary system works just like the decimal system, with two differences: we only use symbols **0** and **1**, and each position in the sequence is worth *twice* as much as the position to its right (instead of ten times).

So, what could the following sequence mean if we *interpret* it as a binary number:

101101101

Just like for decimal numbers, we start from the rightmost position and add up the digits, multiplied by what their positions are worth: one times one plus zero times two plus one times four plus one times eight plus zero times sixteen plus one times thirty-two plus one times sixty-four plus zero times one-hundred-and-twenty-eight plus one times two-hundred-and-fifty-six. We've written the numbers as English words here to make it clear that we're dealing with the "actual" numbers, not the usual, decimal *notation* of the numbers. But of course it is much more convenient to write it like this:

$$1 \times 256 + 0 \times 128 + 1 \times 64 + 1 \times 32 + 0 \times 16 + 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1$$

In fact, the structure of the number system becomes even more apparent if we write what each position is worth using a *basis* and an *exponent:*

$$1 \times 2^8 + 0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

If we add everything up, of course we get three-hundred-and-sixty-five (365). In order to avoid any confusion between different number systems, we often write the base as a subscript, e.g. 365_{10} or

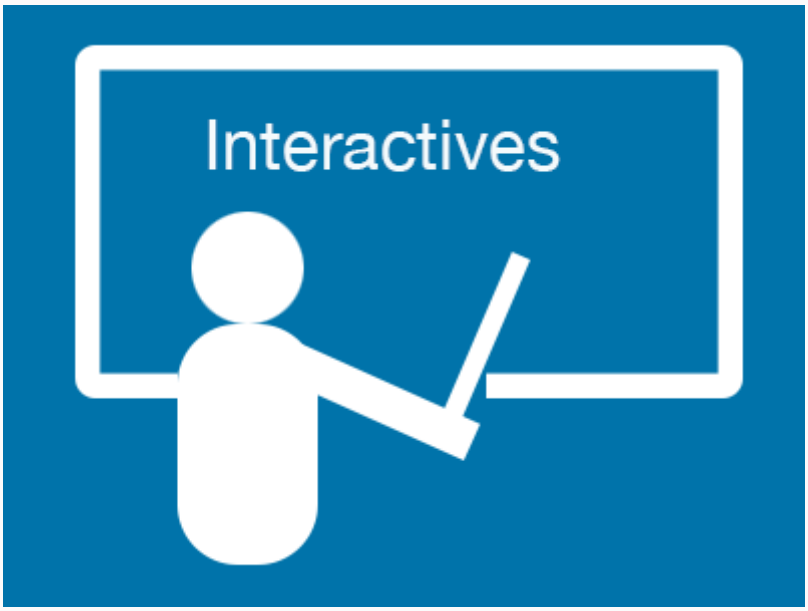
101101101_2 (so that we can distinguish it from 101101101_{10}, or one-hundred-and-one billion one-hundred-and-one thousand one-hundred-and-one).

Arithmetic on binary numbers

Adding two binary numbers works just like adding two decimal numbers: Start from the rightmost digit, add the matching digits, and if the result doesn't fit in one digit, add a *carry over* to the next digit. Here's an example:

$$\begin{array}{cccccc} & 1 & 1 & 0 & 1 \\ + & & & 1 & 1 \\ \hline 1 & 0 & 0 & 0 & 0 \end{array}$$

You can use the little test below to practice addition of binary numbers. It will generate a new pair of numbers to add every time you get the answer right.



Click to open

Conversions

Converting from any number system into decimal numbers is relatively easy. We know how much each position is worth, so all we have to do is add up the digits times what each position is worth (as shown above). But how can we convert from decimal numbers into binary?

Let's use 365 as an example again.

The basic idea is to find the *largest* power of two that is *smaller* than the number we want to convert. In our case, that would be 2^8=256. This tells us that the 9th position (since we will need digits for 2^8 all the way down to 2^0) is 1:

1xxxxxxx

We write x's here for the positions we don't know yet. The next steps are to check for each digit in turn, from left to right, whether it can be used to represent part of the remainder we still need to convert. With the first 1 we've "used up" 256, so the rest of the number must be 365-256=109. The next power of 2 is 2^7=128, which is larger than 109, so we can't use it and have to add a 0:

10xxxxxx

Now 2^6=64, smaller than 109, so we add a 1:

101xxxxx

We've used up another 64, and 109-64=45. The next digit is worth 2^5=32, which is smaller than 45, so it's a 1:

1011xxxx

Again, 45-32=13, next digit is 2^4=16, larger than 13, so it's a 0:

10110xxx

Next digit: 2^3=8, smaller than 13, so we add a 1 and have 13-8=5 still left:

101101xx

2^2=4, smaller than 5, so we add a 1 and have 5-4=1 left:

1011011x

2^1=2 which is larger than 1 (so that's a 0), and finally 2^0=1, which gives us the final two digits:

101101101

Bits, bytes and words

Computers use binary numbers for all their computations.

An important restriction of all modern computers is that they can't compute with *arbitrary* binary numbers, but all operations are limited to a **fixed number of digits**. In a later module, we will see how these computations are implemented in hardware, and then it will become clear why this restriction to a fixed "width" is necessary in order to make the hardware simple and fast.

The following terminology is used to describe these fixed numbers of digits we compute with:

- A **bit** is a single binary digit, i.e., a single 0 or 1.
- A collection of eight bits is called a **byte**. Many early computers could only do computations at a byte level (e.g., add up two 8-bit numbers). What is the largest number you can represent in a byte?

- Any fixed-width collection of bits can be called a **word**. Typical word sizes in modern computers are 16, 32 or 64. Usually, all operations in a CPU use the same word size.
- In order to talk about larger collections of data, we use the prefixes **kilo**=1000, **mega**=1000^2, **giga**=1000^3**, tera**=1000^4, **peta**=1000^5 (and exa, zetta, yotta... for really large numbers). So one kilobyte (written 1 kB) is 1000 bytes, one terabyte (1 TB) is 1000 gigabytes or one trillion bytes or eight trillion bits. Since it's sometimes easier to compute with powers of two, we can also use the prefixes **kibi**=1024, **mebi**=1024^2, **gibi**=1024^3 and so on (you may sometimes see these, but we will only use the decimal units here).

Given that computers use a fixed word width, we can reason about the largest numbers they can deal with. A word of size *n* (i.e., *n* bits), when interpreted as binary numbers, can represent the numbers from 0 (all bits are 0) to 2^n-1 (all bits are 1). Convince yourself that that is true by converting the numbers 1111_2 and 11111111_2 into decimal! Conversely, in order to represent a decimal number x, we will need $\lfloor \log_2(x) \rfloor$ bits.

Hexadecimal numbers

Binary numbers are important because that's what computers operate on. But they are a bit awkward to work with as a human being because they are long. For example, as programmers, we often need to understand exactly what is stored in the computer's memory, and it would be easy to get lost in endless sequences of zeroes and ones.

That's why you will often see another number system besides binary and decimal: the system with base 16, called **hexadecimal numbers** (or *hex* for short). Again, the principle is exactly the same as for binary and decimal, but now each digit can be one of *sixteen* different symbols, and each position is worth *sixteen* times more than the position to its right. The symbols used for hexadecimal numbers are the digits 0-9 and the letters A-F. The table below shows the hexadecimal numbers 0-F with their decimal and binary equivalents.

Hex	Decimal	4-bit binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Note how each hex digit corresponds to exactly four bits. That makes it extremely easy to convert between hex and binary! For example, the binary number 101101101 is easily converted into hexadecimal by splitting it up into blocks of four bits (starting from the right): 1 0110 1101, and then

looking up each block in the table above: 16D. We can verify that 1 times 16^2 plus 6 times 16^1 plus 13 times 16^0 = 256+96 + 13 = 365.

To convert from hexadecimal into decimal, we could use a similar procedure as above for binary-to-decimal conversion. Let's call this the **slow method** for converting numbers between bases.

Slow method for conversion

The following table shows the base 10 values for different powers of 16.

Base 16
16^4 65536_{10}
16^3 4096_{10}
16^2 256_{10}
16^1 16_{10}
16^0 1_{10}

To convert from base 10 to base 16, we divide step by step by the different powers of 16, and continue with the remainder. Let's convert 93530_{10} into hex:

Remainder	Step	Step value	Calculation base 10	Value in base 16 for step
93530_{10}	16^4	65536	93530 / 65536	1
27994_{10}	16^3	4096	27994 / 4096	6
3418_{10}	16^2	256	3418 / 256	D
90_{10}	16^1	16	90 / 16	5
10_{10}	16^0	1	10 / 1	A

This method works for all bases, but always requires to calculate with rather large intermediate results using the values for each place.

Faster method

A faster method successively divides by the base, building up the decimal number from the right:

Division	Result	Remainder	Base 16 number
93530 / 16	=	5845	10 xxxxA
5845 / 16	=	365	5 xxx5A
365 / 16	=	22	13 xxD5A
22 / 16	=	1	6 x6D5A
1 / 16	=	0	1 16D5A

Similarly, the conversion from base 16 into base 10 can be done by successive multiplication by 16, and the system also works for any other base.

Encoding text into binary

Of course computers do more than just add up numbers. But fundamentally, they still need to represent any piece of data as a sequence of bits, i.e., zeroes and ones. We will now see how a common type of

data, namely text, can be represented that way.

The fundamental idea is to assign a fixed *bit pattern* to each possible character. We call this an **encoding** of characters into binary (i.e., into bits). The most common encoding used by almost all computers is called **ASCII** (for *American Standard Code for Information Interchange*). Each character in the English alphabet is mapped to a specific bit 7-pattern (and some symbols, too):

Bit pattern	Character	Decimal value	Bit pattern	Character	Decimal value
0100000	SPACE	32	0100001	!	33
0100010	"	34	0100011	#	35
0100100	$	36	0100101	%	37
0100110	&	38	0100111	'	39
0101000	(	40	0101001	)	41
0101010	*	42	0101011	+	43
0101100	,	44	0101101	-	45
0101110	.	46	0101111	/	47
0110000	0	48	0110001	1	49
0110010	2	50	0110011	3	51
0110100	4	52	0110101	5	53
0110110	6	54	0110111	7	55
0111000	8	56	0111001	9	57
0111010	:	58	0111011	;	59
0111100	<	60	0111101	=	61
0111110	>	62	0111111	?	63
1000000	@	64	1000001	A	65
1000010	B	66	1000011	C	67
1000100	D	68	1000101	E	69
1000110	F	70	1000111	G	71
1001000	H	72	1001001	I	73
1001010	J	74	1001011	K	75
1001100	L	76	1001101	M	77
1001110	N	78	1001111	O	79
1010000	P	80	1010001	Q	81
1010010	R	82	1010011	S	83
1010100	T	84	1010101	U	85
1010110	V	86	1010111	W	87
1011000	X	88	1011001	Y	89
1011010	Z	90	1011011	[	91
1011100	\	92	1011101	]	93
1011110	^	94	1011111	_	95
1100000	`	96	1100001	a	97
1100010	b	98	1100011	c	99
1100100	d	100	1100101	e	101
1100110	f	102	1100111	g	103
1101000	h	104	1101001	i	105
1101010	j	106	1101011	k	107
1101100	l	108	1101101	m	109

Bit pattern	Character	Decimal value	Bit pattern	Character	Decimal value
1101110	n	110	1101111	o	111
1110000	p	112	1110001	q	113
1110010	r	114	1110011	s	115
1110100	t	116	1110101	u	117
1110110	v	118	1110111	w	119
1111000	x	120	1111001	y	121
1111010	z	122	1111011	{	123
1111100	|	124	1111101	}	125
1111110	~	126			

As you can see, we've added the decimal interpretation of each bit pattern into the table as well. So you could say that the pattern 1100011 represents the character c, or at the same time, since $1100011_2=99_{10}$, we could also say that the character c is encoded as the number 99, represented in binary. The first 31 characters are not "printable", they represent things like the end of a line or a tabulator.

For the earliest computers, 7-bit ASCII was sufficient. Of course nowadays computers need to be able to deal with all kinds of character sets, from simple accented latin characters (e.g. ç or ä) and non-latin writing systems, to an ever growing number of emoji ☺. In order to accommodate these requirements, the **Unicode** character set has been defined. It uses up to 32 bits to represent each character, and defines individual codes (i.e., bit patterns), for characters in 135 scripts! A small exercise for you: how many different characters could you theoretically encode using 32 bits? Reveal [1]

Given that each character is represented by a fixed number of bits, we can now easily compute how much memory (or disk space) is required to store a text of a certain size. A plain text file with 1000 characters, encoded as ASCII, will require 1000 times 7 bits. In practice, most computers use an 8-bit extended ASCII set, so each character requires one byte, which means a text with 1000 characters requires exactly one kilobyte. Now let's assume the same text is represented as Unicode: each character needs 32 bits or 4 bytes, so the same text now requires 4 kilobytes to be stored! The designers of Unicode therefore came up with a system that uses a *variable* number of bytes per character - the original ASCII characters still only need a single byte, common characters require two bytes, and less common characters three or four bytes. The main standard for this variable-size encoding is called **UTF-8**, where UTF stands for *Unicode Transformation Format* and the 8 stands for the smallest character width of 8 bits. Encoding ASCII texts into UTF-8 is therefore quite efficient (a 1 kB ASCII text will also use 1 kB after encoding into UTF-8). There's also **UTF-16**, which uses two bytes (16 bits) per character.

Start Quiz (https://www.alexandriarepository.org/app/WpProQuiz/236)

Negative binary numbers

So far we have only seen positive numbers encoded into binary. In order to extend this system to also support negative numbers, let's think about how negative numbers are represented in decimal notation: we add a "-" sign in front of the number to signify that it is negative. We can also use a "+" sign to stress that a number is positive (as in, "the temperature was +5 degrees"). Let's see how that idea looks in binary.

Sign and magnitude

The intuitive approach would be to simply use one bit as the *sign bit.* Let's assume that for a given, fixed-with binary number, the leftmost bit represents its sign, and that 0 means a positive number and 1 means

negative. We call this the *sign-and-magnitude* representation of negative numbers. For example, let's take the 8-bit sign-and-magnitude number 11010110. The leftmost bit is 1, so the sign is negative. The rest of the number, 1010110, is simply interpreted as an (unsigned) binary number, the magnitude, and $1010110_2=86_{10}$. So together, we can say that 11010110, interpreted as an 8-bit sign-and-magnitude number, represents -86_{10}.

Here is a table with all 3-bit sign-and-magnitude numbers and their decimal equivalents!

Decimal	3-bit sign-magnitude binary
0	000
1	001
2	010
3	011
-0	100
-1	101
-2	110
-3	111

The main advantage of sign-and-magnitude is that it is very easy to negate a number: just "flip" the leftmost bit (i.e., turn 0 into 1 or 1 into 0). But the representation has two important drawbacks:

- It contains two zeroes, one of them negative.
- It is difficult to implement basic arithmetic operations. E.g., think about how to add signed-magnitude numbers. It basically requires lots of special case handling.

So this intuitive approach doesn't work well for computers - let's look at an alternative.

Ones' complement

The next idea would be that instead of just "flipping" the leftmost bit to negate a number, let's flip *all* the bits. As before, if the leftmost bit is 0, we consider the number to be positive. This is called **ones' complement**, and here is the table for 3-bit numbers:

Decimal	3-bit ones' complement
0	000
1	001
2	010
3	011
-0	111
-1	110
-2	101
-3	100

The great news is that ones' complement makes it easy to do subtraction, because it just means adding the ones' complement plus an additional carry bit:

$$2\text{-}1 = 2 + (\text{-}1) = 010_2 + 110_2 + 1 = 1001$$

Now we simply ignore the leading 1 (it's just an artefact of the ones' complement notation) and get the result 001. So the good news is that we can reuse normal addition as long as we keep track of the carry

bit and overflow. But since we always have to add the carry bit anyway, we can incorporate that into the representation, which leads us to the main representation of negative numbers used in modern computers.

Twos' complement

In the twos' complement representation, negation of a number is achieved by flipping all bits and then adding 1 (and discarding any potential carry bit that overflows). Here's a table with both ones' and twos' complement:

Decimal	3-bit twos' complement
0	000
1	001
2	010
3	011
-1	111
-2	110
-3	101
-4	100

Note that with 3 bits, we cannot represent the number 4, because in binary it would be 100_2, and since the leftmost bit is 1, this is interpreted as a negative number.

Twos' complement has nice properties:

- There is a single representation of 0 (no -0), with all bits being 0.
- Negative 1 is always 111...1 (i.e., all bits are 1).
- The smallest negative number is 1000...0.
- The largest positive number is 01111...1.

Addition and subtraction work as expected, all we need to do is to ignore any carry bit that overflows. Here are a few examples:

- $2_{10}+1_{10} = 3_{10}$
  $010_2 + 001_2 = 011_2$
- $3_{10}-1_{10} = 3_{10} + (-1_{10}) = 2_{10}$
  $011_2 + 111_2 = 010_2$ (ignoring carry bit)
- $2_{10}-4_{10} = 2_{10} + (-4_{10}) = -2_{10}$
  $010_2 + 100_2 = 110_2$

However, we still need to be careful with overflows. Let's see what happens when we try to add 3+2 in 3-bit twos' complement. The result, 5, or 101_2, is a negative number if interpreted as twos' complement (you can see in the table above that it stands for -3). Similarly, -4-3 can be computed as $100_2+101_2=1001_2$, and if we ignore the carry bit (as usual) we would get 001_2, or decimal 1. So in both cases, the computation has *overflown*, since the result is not representable with the limited number of bits we have chosen. There are two simple rules to detect an overflow:

- If the sum of two positive numbers results in a negative number
- If the sum of two negative numbers results in a positive number

In any other case, the result is correct.

Error detection

Binary data is basically just strings of bits (i.e. strings of 0 and 1). If there is an error that causes one of these bits to be changed, the result can cause lots of problems if it is not detected. This section introduces three basic methods for error detection, namely **parity bits**, **checksums**, and **cyclic redundancy checks CRCs**.

The main approach is to agree on a particular way to detect errors in order to be able to react on this error. Similar to asking "beg your pardon" in human communication if some of the words where not properly understood. The goal of these simple methods is not to correct the error.

A brief introduction to *error correcting* codes can be found here:

https://www.youtube.com/watch?v=5sskbSvha9M

Parity bits

Parity is just another fancy word for equality. The term *Par* in golf means *equal to the expected number of strokes.*

The goal of adding a *parity bit* to binary data (e.g. a number or one ASCII encoded character) is to detect that one single bit has changed.

- A parity bit requires one additional bit to be added.
- Before using the parity, one needs to decide if the parity bit shall be *even* or *odd.*
- Then, one sets the additional parity bit to 0 or 1 so that in the complete sequence (including the parity bit) the number of 1s is even (for even parity), or odd (for odd parity)

Example for even parity (the red number is the parity bit):

01011100

Note that there is 4 1s, thus there is already an even number of 1s. Thus, the parity bit need to be 0 for the complete number of 1s to stay even. For odd parity, the parity bit would consequently need to be 1.

Now lets look what happens when one error occurs. We assume that the bit on position 6 is changed (showed in green):

01011000

As we have agreed on even parity, we can now check if this is still correct. We find 3 1s, which is an odd number. Thus, there must have been an error.

But what happens if there are two errors:

01111000

In this situation, there are again four 1s, but at the wrong positions. However, it is an even number of 1s

and therefore, this error cannot be detected.

Summary parity bits:

- Put one additional parity bit either on the left or on the right-hand side.
- Add all the ones and make the additional bit 0 or 1 so that the result is **even** for even parity or **odd** for odd parity.
- A parity bit can only detect a single bit error.

Checksums

In parity bits, only the number of bits is counted. This is an extremely efficient way to check for single-bit errors. To detect more errors, a more complicated mechanism is needed.

For a checksum, a message needs to be interpreted as numbers. Instead of agreeing on odd or even, one now needs to agree on a particular number. Then, the numbers of the message are added up and then divided by the number. The remainder is added to the message as checksum. The same process is executed for checking the message. If the remainder mismatches, the message was changed.

Example for a checksum (16 is the agreed number):

43 52 43 30 31 30

Add up all numbers 43+52+43+30+31+30=229 and divide by the agreed number: 229/16=14 with a remainder 5. Thus, the message including the checksum looks like this:

43 52 43 30 31 30 5

One error is easily detected. Assume the following message is received:

43 52 43 29 31 30 5

The check results in 228/16=14 with a remainder of 4. Thus, the checksum 5 does not match and the error is found.

Also two errors can be detected as the remainder is now only 2:

43 50 43 29 31 30 5

However, what about these two errors?

43 54 43 28 31 30 5

In this case, the sum is again 229 and the remainder is 5. The two errors cancel each other out. Thus, a checksum can in principle detect multiple errors, but only if they dont cancel each other out or are bigger than the divisor agreed on.

Cyclic Redundancy Check CRC

CRCs are widely used, for example in QR codes. Instead of adding up the numbers, now the numbers are concatenated to build one large number that is then divided by the agreed number. Again, the remainder is used for checking.

For the example if we agree again on 16 as the divisor

43 52 43 30 31 30

the CRC is calculated by taking the remainder of 435243303130 / 16, which is 10. Errors can, of course, still result in the same CRC, but it is much more stable than just taking a checksum. Many different standardised CRCs do exist and are used in practical applications. Note that standardisation is one way to "agree" on a particular divisor.

When looking at binary data, CRCs are usually seen as division in the ring of polynomials over the finite field GF(2). The actual type and number of errors that can be found depends on the actual CRC and on the amount of redundancy in a message. QR codes, for example, can include up to 30% of redundancy.

Do you think CRC codes provide security? [2]

---

[1] 2^{32}=4,294,967,296

[2] CRC codes do not provide security! An attacker could just manipulate the message and compute a new CRC code. CRC is not a security measure. CRCs are about errors (safety) not malicious attacks (security).

3
Boolean Algebra

History

George Boole, born November 2, 1815, Lincoln, Lincolnshire, England, died December 8, 1864, Ballintemple, County Cork, Ireland

He was an English mathematician who helped establish modern symbolic logic and whose *algebra of logic*, now called *Boolean algebra*, is basic to the design of digital computer circuits.

(Enciclopaedia Britannica)

Basic Concepts

In Boolean algebra, an expression can only have one of two possible values: **TRUE** or **FALSE**. No other values exist. Therefore, in addition to many other uses, Boolean Algebra is used to describe digital circuits that also only have two states for each input or output.

The value **TRUE** is often represented by **1**, while **FALSE** is represented by **0**.

The core operators for Boolean Algebra are just three: **AND**, **OR**, and **NOT**.

Notation

Various different notations are used to express AND, OR, and NOT. The following variants are used in FIT1047 and are accepted for exam and assignments:

A AND B can be written as

A ∧ B or **AB**

A OR B can be written as

A ∨ B or **A+B**

NOT A can be written as

$\overline{\mathbf{A}}$ or **¬ A**

Although looking quite similar, the 0 and 1 in Boolean Algebra should not be confused with 0 and 1 in binary numbers. In particular as the short notation for AND and OR looks like mathematical operations for addition and multiplication, the behaviour in Boolean Algebra is different. To help distinguishing Boolean

algebra terms from other mathematical terms, we use capital letters (A.B,C,...).

	Binary	Boolean
0+0	0	0 (because it is FALSE OR FALSE)
1+1	10 (equals 2)	1 (because it is TRUE OR TRUE)

Boolean Operators

The following paragraphs provide explanations of the concepts of AND, OR, and NOT. In principle, they are all very similar to the meaning of the words in natural language. However, there are some subtle differences that needs to be considered when using Boolean Algebra.

AND

A statement **A AND B** is only **TRUE** if both individual statements are TRUE.

AND in Boolean algebra matches our intuitive understanding of the word "and".

Using 1 and 0, we can get a very compact representation as a **truth table**:

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

When you look at this truth table, what would be the value of TRUE AND FALSE? Reveal [1]

OR

A OR B means that either A or B or both are TRUE

Note that OR in Boolean algebra is slightly different from our usual understanding of "or". Very often, the word "or" in natural language use means that either one or the other is true, but not both. For example, the sentence "I would like a toast with jam or honey." probably does mean that either a toast with jam or a toast with honey is okay, but not both. In Boolean algebra, the expression would mean that both together are okay as well.

Truth table for OR:

$\mathbf{A}$	$\mathbf{B}$	$\mathbf{A} \vee \mathbf{B}$
0	0	0
0	1	1
1	0	1
1	1	1

NOT

By just adding negation, Boolean algebra becomes a quite powerful tool that can be used to express complex logical statements, policies, or large digital circuits.

If a statement **A** is **TRUE**, then, the negation **NOT A** is **FALSE**. If a statement A is FALSE, then the negation NOT A is TRUE. Thus, negation just flips the value of a Boolean algebra statement from TRUE to FALSE or from FALSE to TRUE.

The truth table of NOT looks like this:

$\mathbf{A}$	$\overline{\mathbf{A}}$
0	1
1	0

Other operators and gates

In principle, AND, OR, and NOT are sufficient to express everything we want. However, a few other operators are also frequently used:

- **XOR** is the exclusive or that more resembles our intuitive understanding of "or". A XOR B is TRUE if either A is TRUE or B is TRUE, but not both.

$\mathbf{A}$	$\mathbf{B}$	$\mathbf{A} \oplus \mathbf{B}$
0	0	0
0	1	1
1	0	1
1	1	0

- **NAND** is the negated AND. A NAND B is TRUE only if at least one of A or B is FALSE.

$\mathbf{A}$	$\mathbf{B}$	$\mathbf{A} \wedge \mathbf{B}$	$\overline{\mathbf{A} \wedge \mathbf{B}}$
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

- **NOR** is the negated OR. A NOR B is TRUE only if both, A and B, are FALSE.

A	B	$A \vee B$	$\overline{A \vee B}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

Can you write the truth table for NOT(A) AND NOT(B)? Which of the operators above has the same truth table? Reveal [2]

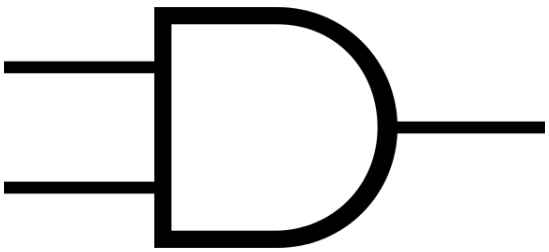
From Boolean algebra to electrical (digital) circuits

In electrical circuits, AND, OR, NOT and additional operators (e.g. XOR, NAND, NOR) are realised as so-called logic gates.

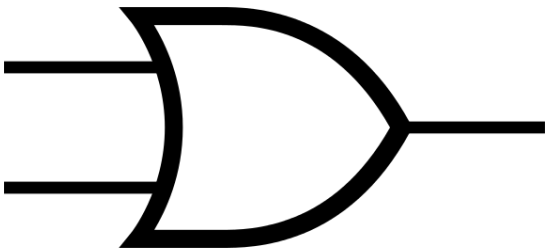
A gate performs one (or several) logical operations on some logical input (i.e. bits) and produces a single logical output. We look at these electrical circuits on an abstract level as "logical circuits". These logical circuits do not behave like electrical circuits. This can be a bit confusing, as a 0 input can result in a 1 output (e.g. when using NOT). This is the correct behaviour of a logical circuit, but it can contradict the intuitive understanding that in an electric circuit there can be no power at the output if there is no power coming in at the input side. Indeed, a logical gate for NOT only has one input and the negated output and a 0 input gets "magically" converted to a 1 output. The reason for this is that in an actual implementation, the NOT gate will have another input that provides power. Then, the NOT gate is actually a switch that connects this additional power input to the output if the actual input is 0. In the idealised notion of logical circuits, these additional inputs are not shown, as they do not change state.

In logical circuits, the following symbols are used for the different types of gates.

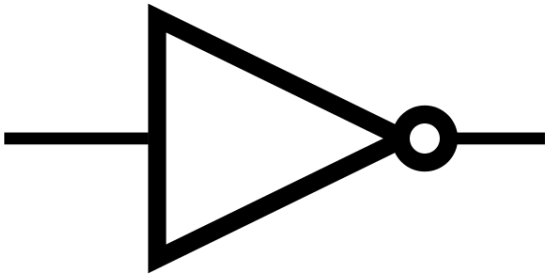
AND Gate



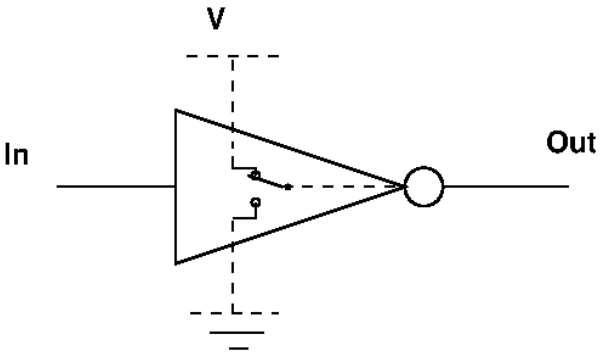
OR Gate



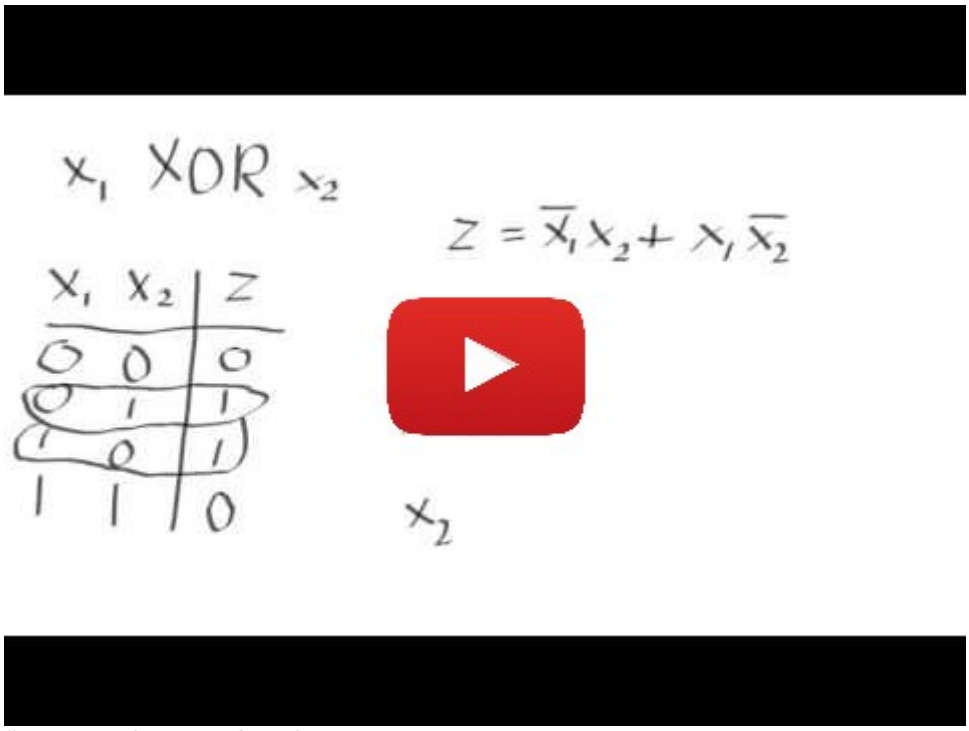
NOT Gate



A slightly less idealised version of the NOT gate would also show the additional connections that provide power to the gate:



These gates can be combined to create logical circuits for Boolean functions. The following video explains how to get from a truth table for a Boolean function to a logical circuit using the three basic gates for AND, OR, and NOT.



(https://youtu.be/3pwDogfWuRg)

Boolean algebra rules

The term *Boolean algebra* implies that we might be able to do arithmetic on symbols. Indeed, there are a number of rules we can use to manipulate Boolean expressions in symbolic form, quite analogous to those rules of arithmetic. Some of these *laws* for Boolean algebra exist in versions for AND and OR.

Identity Law

AND	OR
$1 \wedge A = A$	$0 \vee A = A$

Null Law (or Dominance Law)

AND	OR
$0 \wedge A = 0$	$1 \vee A = 1$

Idempotent Law

AND	OR
$A \wedge A = A$	$A \vee A = A$

Complement Law

AND	OR
$A \wedge \overline{A} = 0$	$A \vee \overline{A} = 1$

Commutative Law

AND	OR
$A \wedge B = B \wedge A$	$A \vee B = B \vee A$

Associative Law

AND	OR
$(A \wedge B) \wedge C = A \wedge (B \wedge C)$	$(A \vee B) \vee C = A \vee (B \vee C)$

Distributive Law

AND	OR
$A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$	$A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$

Absorption Law

AND	OR
$A \wedge (A \vee B) = A$	$A \vee (A \wedge B) = A$

DeMorgans Law

AND	OR
$\overline{A \wedge B} = \overline{A} \vee \overline{B}$	$\overline{A \vee B} = \overline{A} \wedge \overline{B}$

Double Complement Law

$\overline{\overline{A}} = A$

Optimization of Boolean functions / K-maps

When realizing a function as circuit, one would like to minimize the number of gates used for the circuit. Obviously, the different Boolean laws can be used to derive smaller representations for Boolean functions. However, determining the correct order of applying the laws is sometimes difficult and trying different laws is not very efficient. In addition to having a smaller number of gates, one would also like to minimize the use of different types of gates. Therefore, normalized forms for Boolean functions can be useful.

One generic approach for minimizing (smaller) Boolean functions are **Karnaugh maps** or **K-maps**. The idea behind K-maps is to use a graphical representation to find cases where different terms in a Boolean formular can be combined to one simpler term. The simplification used in K-maps is based on the following observation. In both cases, the function is independent from the value of B. Thus, B can be omitted.

$$(A \wedge B) \vee (A \wedge \overline{B}) = A \wedge (B \vee \overline{B}) = A \wedge 1 = A$$
$$(A \wedge B \wedge C) \vee (A \wedge \overline{B} \wedge C) = A \wedge C$$

K-maps are best understood by looking at a few examples.

A K-map with 2 variables

As described above, the following function is obviously independent from B. The example illustrates how this can be derived by using the graphical method of a K-map. The example uses the short notation (AB for $A \wedge B$ and $A + B$ for $A \vee B$).

$$F(A, B) = AB + A\overline{B}$$

In principle, a K-map is another type of truth table. The values of the two variables are noted on the top and the side of a table, while the function's value for a particular combination of inputs is noted within the table.

A \ B	0	1
0	0	0
1	1	1

Now, the goal is to find groups of 1s in the map. Each group of ones that is not diagonal represents a group of terms that can be combined into one term. In the small example, there is only one group:

A \ B	0	1
0	0	0
1	1	1

This group means that for $A = 1$ the value of the function is 1 whatever the value of *B* is. Thus, the function can be minimized as follows.

$$F(A, B) = AB + A\overline{B} = A$$

A K-map with 3 variables

While the example with just 2 variables is quite obvious, it gets a bit less obvious with three variables. Now, the values of two of the variables are noted at one side and the other variable at the second side. Note that the order and choice of variables does not matter. The only important rule is that between two columns (and also two rows) there can only be one variable that is different. Thus, the order for two variables needs to be 00, 01, 11, 10. This is different to the order usually used in truth tables.

This example illustrates the use of K-maps for 3 variables.

$$F(A,B,C) = AB\overline{C} + A\overline{B}\overline{C} + A\overline{B}C + ABC + \overline{A}\overline{B}C$$

We have 5 terms in this function. Thus, we have to place 5 1s into the K-map:

C \ A B	00	01	11	10
0	0	0	1	1
1	1	0	1	1

There is one large group with four 1s that is covering the complete space for A=1.

C \ A B	00	01	11	10
0	0	0	1	1
1	1	0	1	1

The remaining 1 seems to stand alone. Nevertheless, we can find a group by wrapping round to the other side of the table. This group covers two 1s for the area that is valid for C=1 and B=0. Thus, this group is independent from A.

C \ A B	00	01	11	10
0	0	0	1	1
1	1	0	1	1

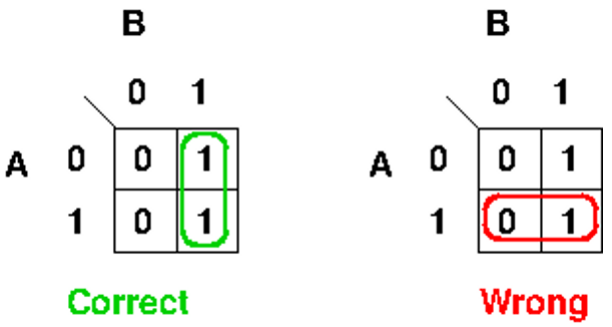
wrap around

The minimized function now has only two terms representing the two groups in the K-map.

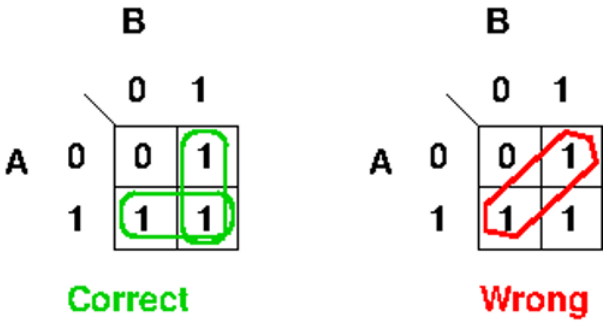
$$F(A,B,C) = A + \overline{B}C$$

Rules for K-maps

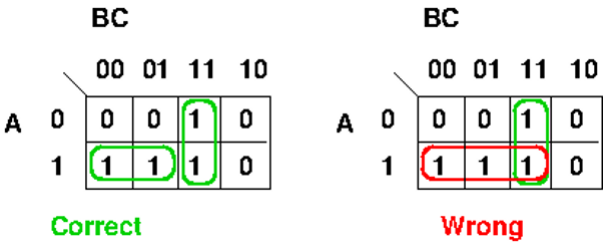
Rule 1: No group can contain a **zero**.



Rule 2: Groups may be horizontal/vertical/square, but **never diagonal**.



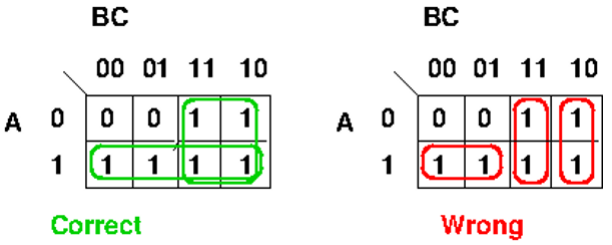
Rule 3: Groups must contain 1,2,4,8,16,32,... (**powers of 2**).



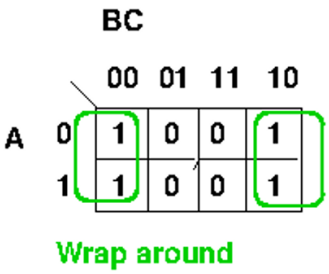
Rule 4: Each group must be as **large** as possible.

Rule 5: Groups can **overlap**.

Rule 6: Each **1** must be **part of at least one group**.



Rule 7: Groups may **wrap around** the map.



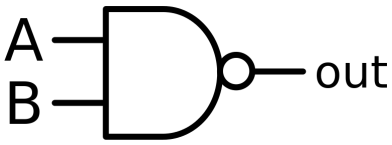
Universal gates

NAND has special properties:

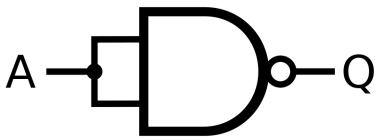
- NAND can be physically implemented very efficiently.
- All other gates can be build only using NAND gates.

Thus, all logical circuits can be implemented using hardware with just a single type of gates. The same holds for NOR. Therefore, NAND and NOR are also called *universal gates.*

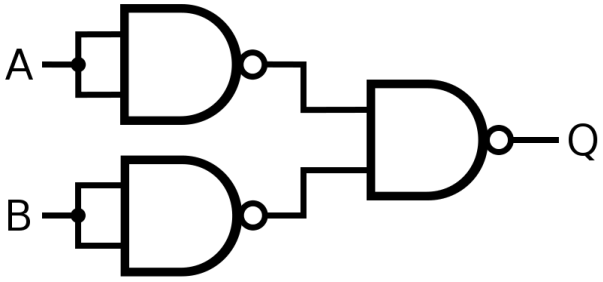
The symbol for a NAND gate is this:



If NAND is negated, obviously the result is just AND. Thus, if we can realize NOT and OR with NAND, all three basic gates can be implemented just using NAND gates. The following circuits show how NOT and OR can be implemented just using NAND gates. Coorectness of these circuits is easily checked using truth tables.



Imlpementation of a NOT gate using NAND



Implementation of an OR gate using NAND

[↩1] FALSE, of course. Nothing can be true and false at the same time. Boolean logic is really very "black and white".

[↩2] Indeed, it is exactly the same truth table as NOR, the negated OR. The Boolean law for this is de Morgans law, which we will see a bit later.

4
Computer Architecture

This module introduces the *architecture* that is used for all modern computer systems. Architecture, in this context, means how the different tasks that a computer needs to do are broken down in small, manageable components, and how these components fit together.

4.1 Overview

The Von Neumann architecture

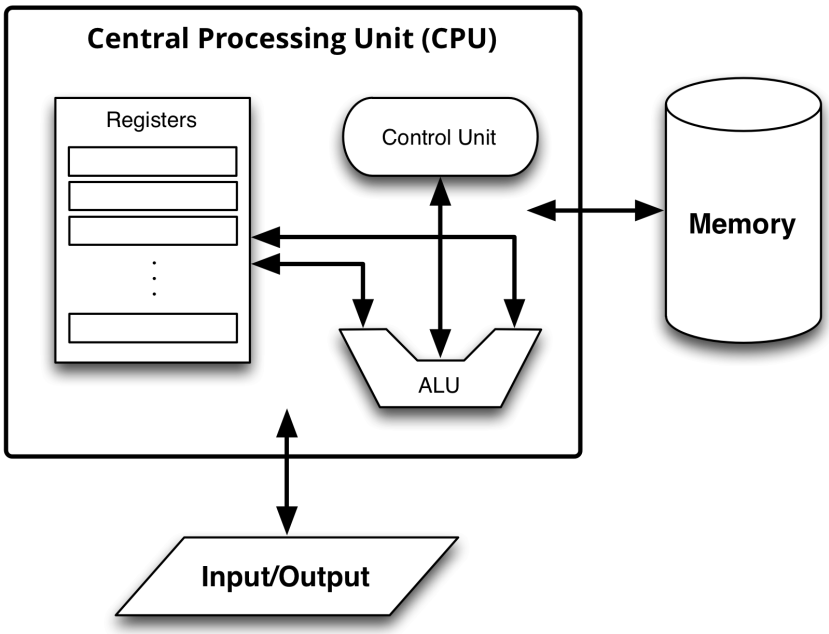
Modern computers come in many different forms. Let's take a look at three very different computers: a desktop PC, a smartphone, and a washing machine.

Clearly, these three computers come in quite different form factors, we communicate with them in different ways, and they have different ways of interacting with the rest of the world. For example, we use a keyboard and mouse to work with a PC, a touch screen for the smartphone, and dials and buttons for the washing machine. Both the PC and smartphone have a screen and a speaker as output devices, the washing machine probably just has a little display for the remaining time and the currently selected program, but it controls the actual hardware (valves, motors etc). The PC may be connected to the Internet using a cable, the smartphone uses a 4G or WiFi wireless network, and the washing machine is (most likely) not connected at all.

The surprising thing is that all these computers are based on the same fundamental architecture!

The Von Neumann Architecture

The main architecture in use today is called the *Von Neumann* architecture, after John von Neumann, a mathematician and early computer scientist who described it in a 1945 report on the EDVAC computer. If you're interested, you can read more about him in his article on Wikipedia (https://en.wikipedia.org/wiki/John_von_Neumann).



The Von Neumann architecture consists of a *Central Processing Unit* (CPU), the *memory*, and the *input/output devices*. Furthermore, the CPU can be subdivided into the *Arithmetic/Logic Unit* (ALU), a number of *registers,* and the *Control Unit* (CU).

The CPU is the "brain" of the computer, it executes programs, which are made up of instructions that manipulate data. Both the program instructions and the data are stored together in the same memory. This is one of the distinguishing features of the Von Neumann architecture - other architectures had

separate memory for instructions and data. The registers inside the CPU are used to store temporary results and move instructions and data around (from the memory or I/O device into the CPU, and then into the ALU, and back to memory or I/O). The ALU performs the actual calculations (e.g. addition, multiplication, logical operations).

The connections between memory, I/O devices and CPU are called *buses*. Most modern computers have a number of different buses for different functions (e.g. separating the data bus that carries data and instructions between memory and CPU, from the address bus that tells the memory which data item to load or store).

Syllabus structure

We will now look into the different components of the Von Neumann architecture in turn, using a very simple architecture as an example. The first submodule explains how CPUs work, followed by submodules on memory and input/output devices.

4.2
Central Processing Units (CPUs)

This module is about the "brains" of digital computers, the *Central Processing Units*. We will introduce the *language* that they understand (called *machine code*), and construct an entire CPU from basic logic gates.

Before you start working through this module, you should understand the following concepts:

- The Von Neumann architecture
- Binary numbers and twos' complement
- Boolean logic (AND, OR, NOT, NAND)

The first part of this module describes what CPUs do, and discusses how they have changed from the early days of computing in the 1960s to modern, highly integrated circuits. The second part introduces a simple machine code and assembly language, and you'll learn how to program a CPU at the lowest level. The third part develops the basic circuits required to build a CPU, such as adders, multiplexers, decoders and flip-flops. The fourth part, finally, combines all these circuits to construct an *Arithmetic/Logic Unit (ALU),* a *Register File*, and a *Control Unit* - so after completing this module, you will have a very good overview of how most aspects of a modern CPU work.

4.2.1
CPU basics and History

Introduction

A Central Processing Unit, or CPU, is the component of a computer that does most of the actual "computing". Almost all modern computers are based on the same *architecture*, called the *Von Neumann architecture*, where the instructions that make up the programs we want to run, as well as the data for those programs, are stored in the *memory*, and the CPU is connected to the memory by a set of wires called the *bus*. The bus also connects the CPU to external devices (such as the screen, a network interface, a printer, or input devices like touch screens or keyboards).

A program, at least for the purposes of the CPU, is a sequence of very simple *instructions*. You may have seen programs in languages like Java, Python, JavaScript, or C, but these languages are far too complex for a CPU. We have to be able to construct a CPU from simple logic gates (AND, OR, NOT), so we have to keep the language it understands simple, too. Typically, CPU instructions are of the form:

- *Take one or two words of data, do a simple operation on them, and store the result somewhere.*
- *If a certain condition is true (e.g., if one word of data is equal to zero), continue execution with a different instruction.*
- *Transfer one word of data to or from memory, or to or from an input/output device.*

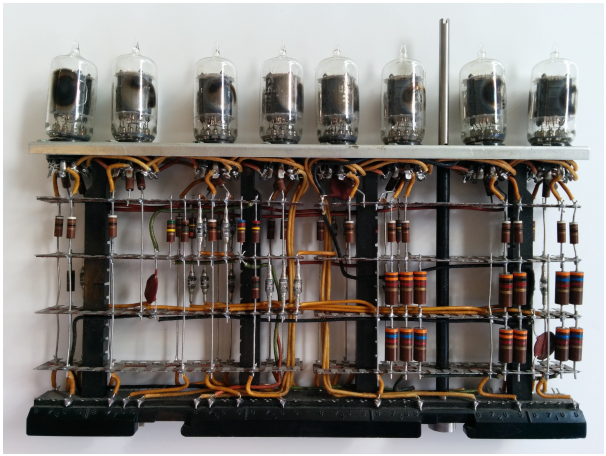
As you can see, *words* of data play an important role. CPUs use a fixed word-width, typically 16, 32 or 64 bits, and all instructions and all data has to be broken down into words of this fixed width. Again, the reason for this is to simplify the circuits.

History

The historic facts in this part of the module **are not assessable.** You should however understand the *concepts*, e.g., why computers are no longer built out of tubes.

Some early CPUs

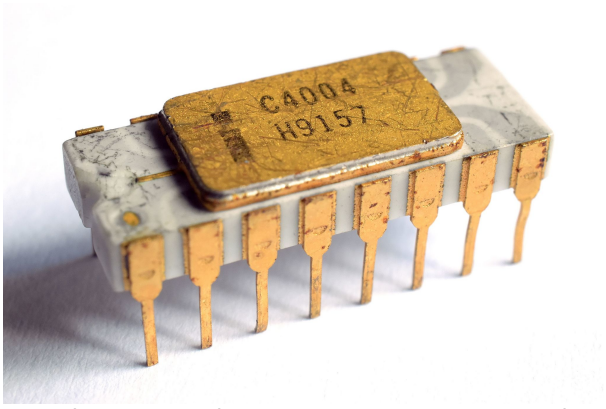
The first CPUs were built out of vacuum tubes and resistors, which can be combined to form simple logic gates such as NOR and NAND, from which all other logic circuits can then be built. Here is a picture of part of an early IBM 700 series CPU (tubes at the top, resistors below):



A module from a IBM 700 series computer, with eight 5965A/12AV7 dual triode vacuum tubes. © Wikipedia User:Autopilot / Wikimedia Commons / CC-BY-SA-3.0

The computer revolution really started with successive miniaturisation. First, *transistors* replaced tubes. They have the same function (and are also used to form logic gates), but transistors are much smaller and at the same time much less prone to failure. Then, the introduction of *integrated circuits*, combining many transistors on a single chip, meant that computers became at the same time smaller, faster, and much, much cheaper.

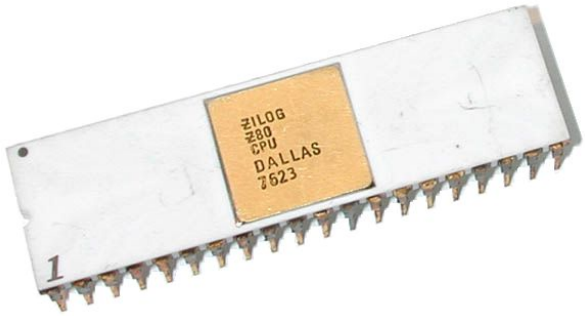
Intel's first commercially available microprocessor, originally designed for building calculators, was the Intel 4004. It packed 2300 transistors on a single chip, and worked on 4-bit words. So what is the largest number it can compute with? Reveal [1] This doesn't sound useful for a desktop calculator, does it? Well, in fact calculators based on this chip did not work with long binary numbers, but rather encoded each decimal digit into a binary number - and how many bits are needed to encode the digits 0-9? Reveal [2]



Intel C4004, By Thomas Nguyen, CC BY-SA 4.0, via Wikimedia Commons

The 1980s was the decade of *home computers,* and many of those were based on either the *MOS Technology 6502* (3,510 transistors) or the *Zilog Z80* (8,500 transistors). These microprocessors used 8-bit words.

MOS 6502 Processor, by Dirk Oppelt, CC BY-SA, via Wikimedia Commons



Zilog Z80 processor, by Gennadiy Shvets, CC BY-SA 2.5, via Wikimedia Commons

Current designs

Modern CPUs contain millions of transistors. The two most widely used families are *ARM* and *Intel x86.* ARM processors can be found in most smart phones and tablet computers, but also in all kinds of devices from televisions to washing machines, and small computers like the Raspberry Pi (you can buy a simple ARM-based computer for under $10!) . Intel x86 family processors are the most common CPUs for PCs (the current models are typically models from the i3, i5 or i7 series).

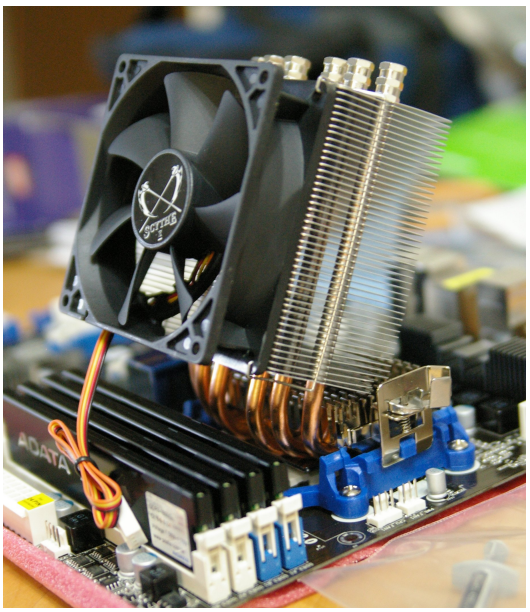
Let us take these two examples to explain the differences between different CPU architectures.

The main difference between different families or types of CPU is the *Instruction Set Architecture (ISA)*, which is the "language" of instructions they understand. For example, an Intel Core i5 processor does not understand a program written for ARM CPUs. But different processor types within the same family may use the same ISA, such as the Intel Core i5 and Core i7 ranges: both can run the same code, but it will (usually) run faster on an i7.

Moore's Law

As we have seen above, the basic building blocks of CPUs haven't changed much: from the earliest computers to the smart phone in your pocket, they contain components that form logic gates, and those are connected into circuits that implement all the required functionality. What has changed is the *density* of those circuits: The IBM 700 series filled an entire room with its vacuum tubes, while a modern Core i7 packs 2.6 *billion* transistors onto a chip with a surface area of just 355 mm^2.

In 1965, Gordon Moore observed that the number of components per integrated circuit seemed to double every year. Surprisingly, this development continued in many areas of electronics for many decades after he first observed it. We therefore say that the development of electronic circuits has followed **Moore's Law** (even though there is of course no physical law that would prescribe a doubling every year or so).



A heatsink with fan, by Wikipedia user Hustvedt, CC BY-SA 3.0, via Wikimedia Commons

In recent years, the trend has somewhat slowed down. The individual structures on a processor have become very small (they are typically around 22 nm wide, with structures down to 10 nm already in planning). Making them even smaller (and thus increasing the possible density of transistors packed on a chip) may require developing completely new materials and processes.

In the past, the miniaturisation had a direct impact on the performance of a CPU - the smaller, the quicker. That trend has basically stopped, because the increase in raw speed meant, at the same time, an increase in energy consumption and, more importantly, waste heat generated by operating the CPU. This waste heat needs to be removed from the CPU (because it can only operate in a certain temperature range), so extensive cooling is required, using heatsinks, fans, or even liquid cooling. When the cooling and power requirements hit a limit, CPU designers instead put multiple *cores* onto a single chip, each of which is (basically) a full CPU. Programs will only run faster on these new CPUs if they are designed to perform multiple tasks in parallel (so that all cores have something to do).

Programs

Let's first think about what a *program* actually is. Here is simple a program (the old classic "*Hello World*") in the Java programming language:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

Or how about a Python program that greets you with your name:

```
import sys
name = sys.argv[1]
print 'Hello, ' + name + '!'
```

At Monash, we develop a special-purpose programming language called *MiniZinc*, this is what a simple logic puzzle looks like in that language:

```
int: n;
array[0..n-1] of var 0..n: s;
constraint forall(i in 0..n-1) (
  s[i] = sum(j in 0..n-1)(s[j]=i)
);
```

```
solve satisfy;
```

So how are these (very different) programs executed by a CPU? Well, it turns out that, strictly speaking, *none of them* can be executed by a CPU! That's because CPUs can only execute **machine code**. Any other program needs to be either **compiled** into machine code or **interpreted.**

Compilers

A compiler takes a program in a language like Java or C++ and translates it into a lower level language. E.g. in the case of C++, it translates it directly into machine code that can be executed by the CPU. In other cases, such as Java, it translates it into so called *byte code*, which in turn is interpreted.

Compilers that generate machine code need to *target* a particular architecture. For example, a program written in C++ can be compiled for the ARM architecture or for the Intel x86 architecture (because the machine code is different!).

Interpreters

An interpreter executes, or interprets, the instructions written in an interpreted programming language such as Python, without first translating them into machine code. The interpreter is itself a program, usually written in a compiled programming language (so it can run directly on the CPU).

An advantage of interpreted languages is that only the interpreter program itself needs to be available as machine code for the target platform. For example, as long as we have a Python interpreter for our computer, we can run any Python code without first compiling it all for our architecture.

A disadvantage of interpreted languages is that they are typically slower to execute than compiled languages. The reason for this is that an interpreter always needs to first "*understand*" (i.e., analyse and, well, interpret) each statement in our program before it can be executed. In a program that executes the same code a million times in a loop, the interpreter needs to look at the code in the loop a million times. A compiler only looks at it once!

Machine Code

Machine code is a very low-level programming language. A program in machine code is a *sequence* of individual *instructions*, each of which is just a sequence of bits. Usually, an instruction consists of one or more words. A CPU can only execute a machine code program that is stored in the main memory of the computer. Every type of CPU has its own machine code, so a program that can run on an ARM processor will not work on an Intel x86 processor.

Here is a *memory dump*, it shows part of the memory contents of a computer at a particular point in time:

```
0001000000000100
0011000000000101
0010000000000110
0111000000000000
0000000010001110
0000110110000000
0000000000000000
```

Each line is one word in this architecture (the *word size* is therefore 16). Just from looking at it, we cannot know whether these bit patterns represent a machine code program or are in fact just data. We will see later that the first four words are in fact instructions, while the last three are data (numbers, in this case). **This is a really important point:** the memory contains both instructions and data, and there is no way of distinguishing the two.

A machine code program is executed **step by step**: The CPU starts with the first instruction, executes it, then moves on to the instruction immediately following the first one, and so on.

Instruction Set Architectures

The set of machine code instructions that a particular type of CPU understands is called the *Instruction Set Architecture*, or *ISA.* As mentioned above, a machine code program for one ISA, such as the *ARM* architecture, will not run on an incompatible one like Intel x86.

Even though many different, incompatible ISAs exist, they all need to achieve roughly similar tasks. Recall the *Von Neumann* architecture, in which a CPU contains a *Control Unit* and an *Arithmetic/Logic Unit,* connected to the memory and the I/O devices. An ISA needs to reflect the capabilities of these components, and therefore must be able to

- do some maths (add, subtract, multiply, compare)
- move data between memory, CPU and I/O devices
- execute *conditionals* and *loops*

Why the first kind of instructions (arithmetics etc) is needed should be self-evident. The second kind is required because CPUs, as we will see soon, typically do not "operate" directly on data in memory. Instead, they copy small pieces of data (usually individual words) into the CPU, then perform operations, and then copy them back into memory. Similarly for input and output devices, the CPU must be able to transfer data e.g. from the keyboard or the hard disk into memory, or send data to the network interface.

The third kind of instruction allows us to code things like "*if the user entered 'A' then do something, otherwise do something else*". That's what we call a *conditional* instruction. A *loop* is a piece of code that is executed multiple times, for instance, "*for each character in the text, do the following: change the character from lower case to upper case*".

CPU components

In order to understand how a CPU can run a machine code program, we first need to get an overview of the different components that typical CPUs consist of.

Memory

Before we start describing the CPU, we first need to understand roughly how *memory* works, even though it is not part of the CPU. As a computer user, you're probably aware that e.g. your laptop has a certain amount of main memory (RAM), such as 4GB. Perhaps you have opened a tool like the Windows Task Manager, the Mac OS Activity Monitor, or the Gnome System Monitor on Linux, which show you how much memory each application is currently using. But from a user's point of view, memory stays quite opaque. Let's take a look inside.

The main memory of a computer can be thought of as a sequence of *locations*, each of which can store

one *value*. Each value has a fixed *width,* i.e., a fixed number of bits. A program can *read* the value stored in a memory location, and also *change* it. In order to do that, programs need to be able to say *which* memory location they want to read or change. That's why each location gets an *address*, by consecutively labelling the locations, starting from 0.

Let's take another look at the memory dump from above:

```
0001000000000100
0011000000000101
0010000000000110
0111000000000000
0000000010001110
0000110110000000
0000000000000000
```

If we assume that the first line is also the first memory location in the computer, it will get address 0. Furthermore, in this case we assume that each location can store a 16-bit word. The value stored at location 3 is therefore what? Reveal [3]

At the risk of repeating ourselves, keep in mind that the memory here simply stores 16-bit values. Each value can be an instruction *or* data (or in fact both at the same time, as we'll see later!).

Registers

A register is a very fast memory location *inside* the CPU. Each register can typically only store a single word, but it is many orders of magnitude faster to read from or change the value in a register compared to accessing the main memory. We distinguish between *general purpose* registers, which can be used by the programmer to store intermediate results of computations, and *special purpose* registers, which are used internally by the CPU.

There are two special purpose registers that can be found in almost any CPU architecture: the *Program Counter* (PC) and the *Instruction Register* (IR). The Program Counter PC always stores the address of the *next* instruction that the CPU should execute, while the IR stores the *current* instruction that the CPU is executing.

ALU, Control Unit and the Bus

The **arithmetic logic unit** (ALU) is responsible for performing basic computations such as addition and multiplication, as well as Boolean logic operations, for example comparisons or AND and OR operations.

The **control unit** (CU) is the actual "machine" inside the CPU. It coordinates the other components. For example, it can switch the memory into "read" or "write" mode, select a certain register for reading or writing, and tell the ALU what kind of operation to perform. All this is based on the current instruction in the IR (instruction register).

The **buses** are the connections between the components inside the CPU, as well as to external components such as the memory or the input/output devices.

Fetch, Decode, Execute

Now that we have seen all the individual components, we can explain how they work together to execute a program. The control unit in the CPU performs the so-called *fetch-decode-execute* cycle.

Fetch: The PC register contains the memory address where the next instruction to be executed is stored. In the fetch cycle, the CU transfers the instruction from memory into the IR (instruction register). It then increments the PC by one, so that it points to the *next* instruction again. This concludes the fetch cycle.

Decode: In the decode cycle, the CU looks at the instruction in the IR and decodes what it "means". For example, it will get any data ready that is required for executing the instruction. This could mean setting up the memory to read from a certain address, or enabling one of the general purpose registers for reading or writing.

Execute: In the execute cycle, the actual operation encoded by the instruction needs to be performed. For example, the CU may load a word of data from memory into a register, switch the ALU into "addition mode", and store the result of the addition back into a register.

When the *execute* cycle concludes, the control unit starts again with the next *fetch*. Since the PC was incremented in the previous fetch cycle, the next one will now go on with the next instruction of the program.

↩1 15

↩2 Ok, that was kind of obvious: 4 bits.

↩3 0111000000000000

4.2.2
MARIE - A simple CPU model

This module introduces a simplified CPU architecture called *MARIE.* It is very basic (e.g., it only has 16 different instructions), but it is complex enough so we can use it to explain the most important features of modern CPUs.

Why should I learn how to program MARIE?

This is a question we get asked a lot. Obviously, "Proficient MARIE programmer" is not something you see as a requirement in any job ad. Even "Proficient ARM assembly programmer" is not a highly sought-after skill.

Learning to program the MARIE machine is about learning *how a CPU works.* You will understand much better how memory is organised in a computer, what the fundamental operations are that a computer needs to execute, and how a high-level program (written e.g. in Java or Python) needs to be mapped down to machine code for a particular architecture in order to be executed.

The MARIE architecture

Let us now make things much more concrete, and introduce a particular machine architecture called *MARIE.* Compared to real architectures, it is very, very simple:

- Words are 16 bits wide
- There are only 16 different instructions
- Each instruction is one word (16 bits) wide, composed of a 4-bit *opcode* and a 12-bit *address*
- There is a single general-purpose register

Registers

The MARIE architecture contains the following registers:

- **AC** (accumulator): This is the only general-purpose register.
- **MAR** (Memory Address Register): Holds a memory address of a word that needs to be read from or written to memory.
- **MBR** (Memory Buffer Register): Holds the data read from or written to memory.
- **IR** (Instruction Register): Contains the instruction that is currently being executed.
- **PC** (Program Counter): Contains the address of the next instruction.

For the moment, only the AC and PC registers are important. We will see how the other registers work in the module From Instructions to Circuits.

Instructions

As already mentioned above, each instruction in MARIE is a 16-bit word. Since each location in MARIE memory can hold a 16-bit value, one instruction fits exactly into a memory location.

Now we could simply make a list of all the instructions we need, and assign a 16-bit pattern to each individual instruction. But Instruction Set Architectures are typically constructed in a much more structured way, to make it easy to implement the Control Unit hardware (which is responsible for actually *decoding* the instructions). In the case of MARIE, the leftmost 4 bits in each instruction represent the *opcode*, which tells us what kind of instruction it is. The remaining 12 bits contain an *address* of a memory location that the instruction should work with.

For example, the opcode `0001` means "*Load the value stored at the address mentioned in the remaining 12 bits into the AC register*". With that information, we can now try to understand the first value in the memory dump: the `0001000000000100` begins with the opcode `0001`, so it is an instruction to load data from memory, and the address to load from is `000000000100`. This is of course the binary number for decimal 4. When the CPU executes this instruction, it will therefore load the value currently stored at memory address 4, and put it into the AC register inside the CPU, so that further instructions can use it. So what will be the value of AC after executing this instruction? Reveal [1]

Assembly language

Machine code is obviously hard to write and read. Instead of dealing directly with the 4-bit opcodes, we introduce a *mnemonic* for each opcode that can be easily remembered and recognised. We also call these mnemonic opcodes *assembly code*, and an *assembler* is a tool that translates an assembly code program into real machine code (it is basically a very simple compiler!).

Here's an overview of part of the MARIE instruction set. The X in the instructions stands for the address part.

Opcode	Mnemonic	Explanation
`0001`	`Load X`	Load value from location X into AC
`0010`	`Store X`	Store value from AC into location X
`0011`	`Add X`	Add value stored at location X to current value in AC
`0100`	`Subt X`	Subtract value stored at location X from current value in AC
`0101`	`Input`	Read user input into AC
`0110`	`Output`	Output current value of AC
`0111`	`Halt`	Stop execution
`1010`	`Clear`	Set AC to `0`

Notice how the instructions use the AC register as temporary storage.

Using these mnemonics, we can now write a simple program that will add up two numbers from memory. Let's first write it down in *pseudocode.* A pseudocode is a program written in a "programming language" that doesn't exist. It is useful when you plan a program, since you can just write down the general structure without having to use the exact syntax of a particular programming language.

In this case, the program consists of four steps that are supposed to be executed in this order:

```
Load number from memory address 4 into AC register
Add number from memory address 5 to AC register
Store result from AC register into memory address 6
Stop execution
```

Using the table of mnemonics above, it is now easy to translate this into MARIE assembly code:

```
Load 4
Add 5
Store 6
Halt
```

Now we can reveal the secret of the mysterious memory dump from above: it's of course the binary representation of exactly this program, and it already contains some data at addresses 4 and 5. Can you guess what value will be stored in memory location 6 after executing this program? Hint [2] Reveal solution [3]

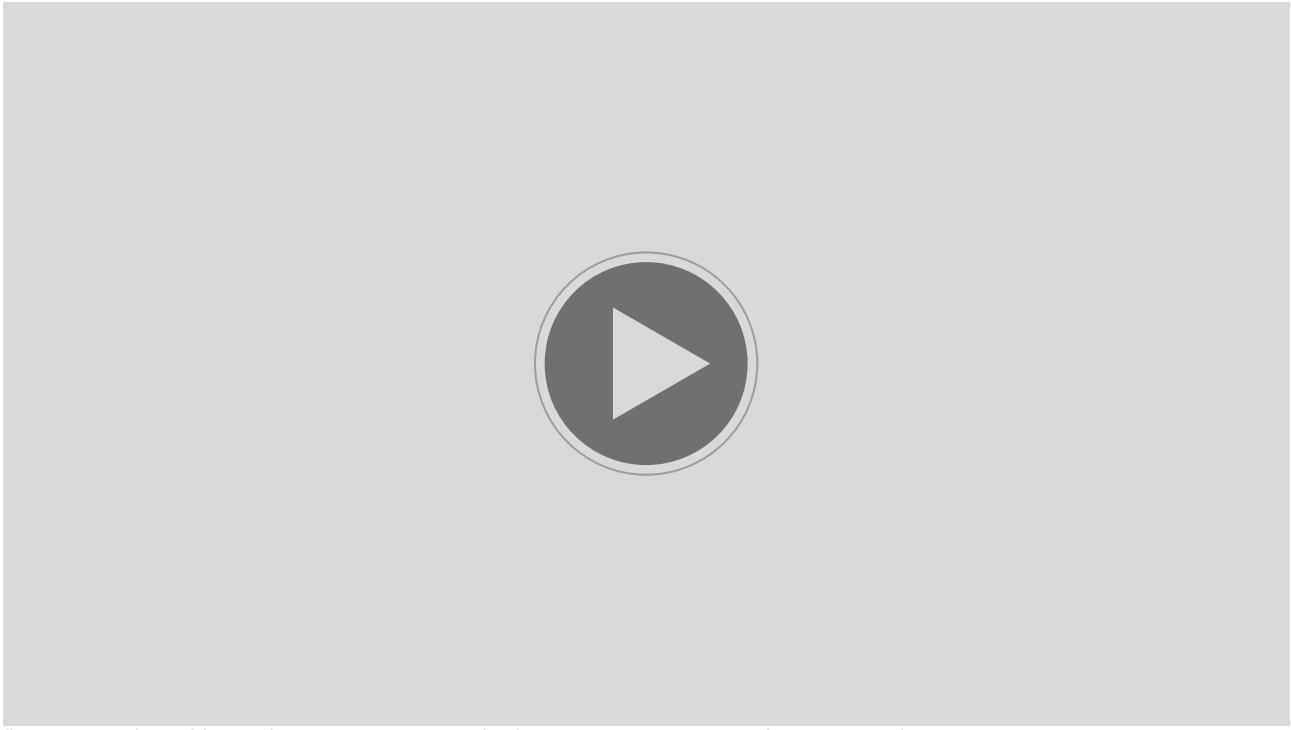
The `Input` and `Output` instructions can be used to read input from a user, and to output values to the screen, respectively. Again, they both use the AC register. So in order to e.g. output a value that is stored in memory, we first have to `Load` it into the AC, and then use the `Output` instruction. In a real computer, similar instructions would be used to read data from a hard disk or the network, or to send data to a screen, a sound device or a motor of a robot.

The MARIE simulator

Of course since MARIE is not a real architecture, you can't just buy a little piece of hardware to try out your MARIE programs. And that would be quite inconvenient anyway, because you'd have to write the program e.g. on your laptop, then somehow transfer it to the MARIE hardware, and somehow observe its behaviour in order to find out if you made any mistakes.

So instead of using real hardware, we are providing a *MARIE simulator* (https://cyderize.github.io/MARIE.js/), so you can experiment with MARIE code directly in the web browser.

The following video again explains the basic concepts such as memory and instructions, and then shows you how to use the MARIE simulator.



(https://www.alexandriarepository.org/wp-content/uploads/20170120134911/MARIE-intro.mp4.mp4)

Jumps, loops and tests

With the instructions we've seen so far, we cannot construct any interesting programs. The CPU starts executing with the first instruction, and then just follows the list of instructions until it reaches a HALT. But most interesting computations cannot be expressed with a fixed number of steps. For example, how can a program react to user input if it always has to continue with the next instruction, no matter what the user typed?

We therefore need instructions that can *jump* to different parts of the program, depending on certain *conditions*. The MARIE instruction set contains two instructions for this purpose:

Opcode	Mnemonic	Explanation
1000	SkipCond X	Skip next instruction under certain condition (depends on X)
1001	Jump X	Continue execution at location X

Let's start with the second instruction, `Jump X`, which is quite straightforward. It causes the CPU to get the next instruction from location X, and then continue from there. What the CPU actually does, internally, is to set the value of the PC register to X. Remember that the PC always points to the *next* instruction to be executed, so this has exactly the right behaviour!

The `SkipCond` instruction is a bit more complicated. In fact, many students struggle a bit with how `SkipCond` works, so it's worth spending some time with the simulator to try it out in detail. The `SkipCond` instruction is a *conditional* instruction, because it behaves differently depending on the current value in the AC and the value of X. Note that X is not used as an address of a memory location here, but instead, it is used to distinguish between three different versions of `SkipCond`:

- `SkipCond 000`: If the value in AC is smaller than 0, then skip the next instruction.
- `SkipCond 400`: If the value in AC is equal to 0, then skip the next instruction.
- `SkipCond 800`: If the value in AC is greater than 0, then skip the next instruction.

Any other value for X should not be used with `SkipCond`.

In most cases, we want to use a *combination* of `SkipCond` and `Jump` instructions to implement conditional code. For example, let's consider the following pseudocode:

```
Get input from user into AC
if AC&gt;0 then output AC and go back to line 1, else halt
```

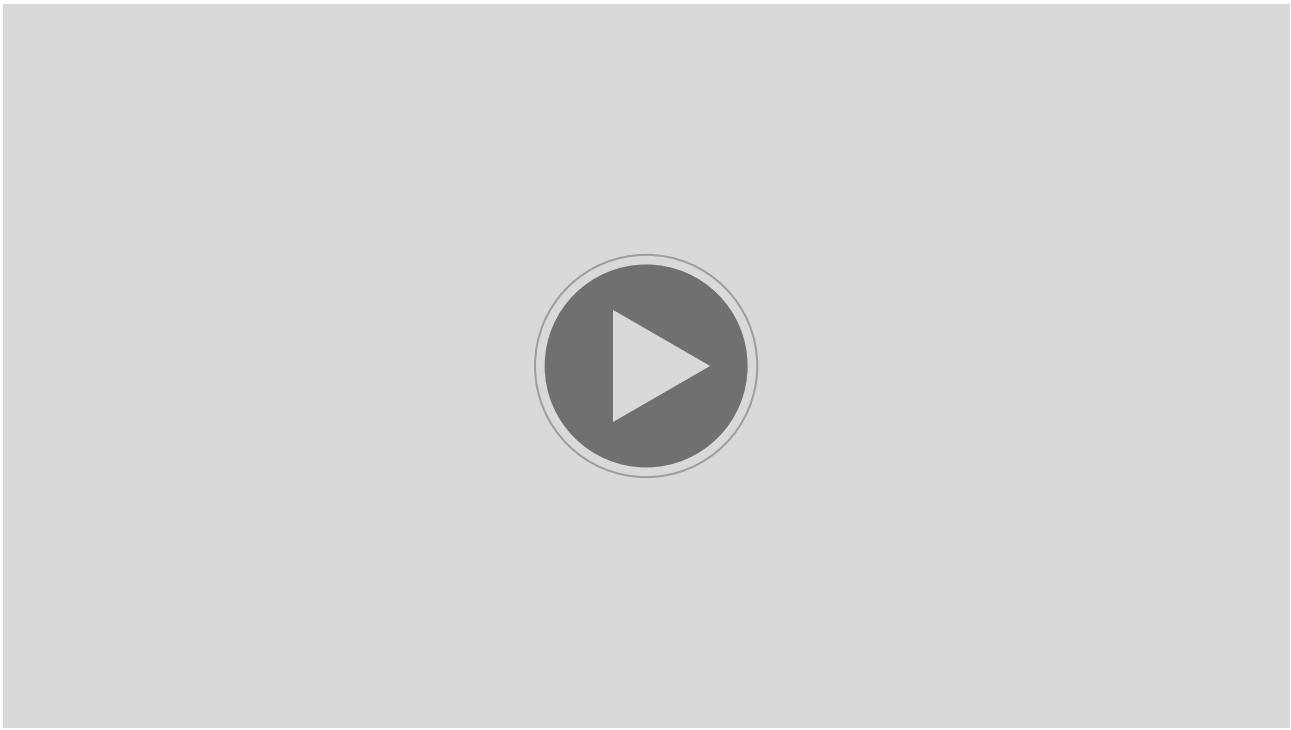
How can we turn this *if-then-else* construct into a sequence of MARIE instructions? The good news is that `SkipCond` allows us to test if AC is greater than 0 (by using condition code `800`). Now the confusing thing about `SkipCond` is that it *skips* the next instruction if the condition is true. So the part in the *else* has to go immediately after the `SkipCond`, and the *then* part comes after that:

```
Input          / Get user input into AC
SkipCond 800   / Skip next instruction if AC&gt;0
Halt           / Halt (if AC not greater than 0!)
Output         / Output AC
Jump 0         / Jump back to beginning of the program
```

Luckily we only wanted to do a single thing here if the condition is false: halt the machine. If there's more to do in the *else* part of a conditional, we need to use a `Jump` to continue execution at a different address.

We leave that as an exercise!

The following video explains jumps and conditionals again.



(https://www.alexandriarepository.org/wp-content/uploads/20170120140056/loops_new.mp4.mp4)

Indirect addressing

We've already covered nine out of the sixteen possible MARIE instructions (remember we only use four bits for the opcode). Most of the remaining instructions are just variants of the ones we've already seen, but they are different in an important way.

Let's look again at the `Load X` instruction. It directly loads the value stored at address X into the AC register. What this means is that we can only use fixed, precomputed addresses that are hard-coded into the instructions. But a typical coding pattern is to store data in an *array*, i.e., a sequence of consecutive locations in memory, often without a fixed length. For example, for a Twitter application it may be enough to say that the text for a tweet is stored at memory locations 100-239 (since tweets are limited to 140 characters), but what about an email application? With the fixed-address instructions we've seen so far, there is no way to loop through all the characters in the email text (e.g. in order to print them onto the screen).

The solution to this problem is to use *indirect* addressing. Instead of accessing the value stored at location X, we can use the value stored at X as the address at which the actual value we want to use is stored. That sounds complicated so let's look at an example. Here's what the current contents of our memory could look like starting from location `100`:

Address	Value
100	3
101	2
102	100
103	101

Now a `Load 102` instruction would look into memory location `102`, find the value `100` there, and load that value into the AC. But the instruction `LoadI 102`, which is the indirect addressing version of `Load`, would look into location `102`, and use the value `100` stored there as an address, or a *pointer*, to where the real value to load can be found: so it looks into location `100`, and loads the value 3 into the AC.

The big difference is that we can now use other instructions to change what gets loaded into AC! For example, what would the following program output?

```
LoadI 102
Output
Load 103
Store 102
LoadI 102
Output
```

Let's go through it step by step. The first indirectly loads from the address pointed to by address `102`. That's what we just discussed, so it will load the value 3 into AC, and the next line will output it. Now the interesting thing happens: line 3 loads the value stored at `103` into AC, so AC will become `101`. Then line 4 stores that value into location `104`. So at this point, the memory will look like this:

Address	Value
100	3
101	2
102	101
103	101

And now we have the same two instructions again as at the beginning of the program: `LoadI 102` and `Output`. But this time, `LoadI 102` finds that location `102` points to location `101` instead of `100`! So the output will be 2.

There are four indirect addressing instructions in the MARIE instruction set:

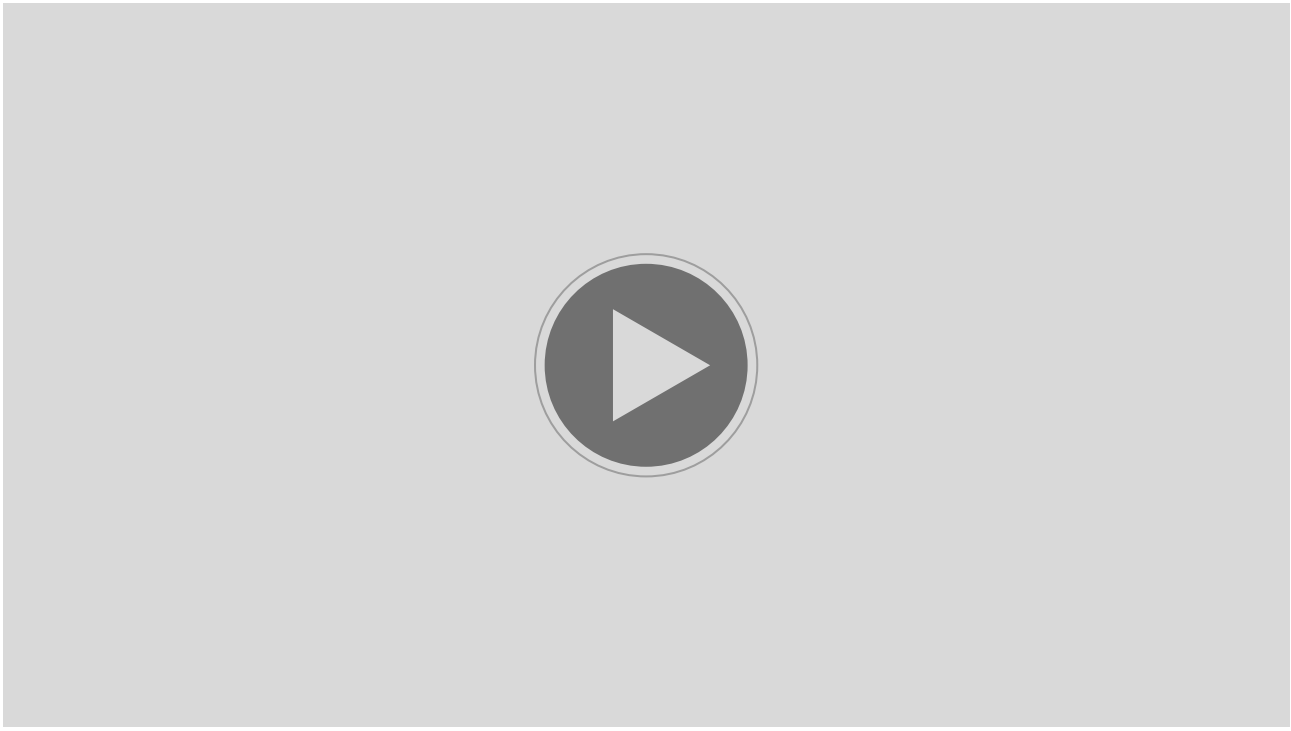
Opcode	Mnemonic	Explanation
1011	`AddI X`	Add value pointed to by X to AC
1100	`JumpI X`	Continue execution at location pointed to by X
1101	`LoadI X`	Load from address pointed to by X into AC
1110	`StoreI X`	Store AC into address pointed to by X

Here is an example program that loops through an array of numbers (starting at address 00F) and outputs each of them, until it finds a zero. Switch the Output mode in the MARIE simulator to ASCII to see the message! You can add arbitrarily many numbers to the array as long as you finalise the sequence with a zero. This is the usual representation of *strings* (i.e., sequences of characters) in many programming languages.

```
Loop, LoadI Addr
      SkipCond 800
      Jump End
      Output
      Load Addr
      Add One
      Store Addr
```

```
      Jump Loop
End,  Halt
One,  DEC 1
Sum,  DEC 0
Addr, HEX 00C
      DEC 70
      DEC 73
      DEC 84
      DEC 0
```

Here's a video on indirect addressing in MARIE.



(https://www.alexandriarepository.org/wp-content/uploads/20170120152601/indirect.mp4.mp4)

Subroutines

There's only one MARIE instruction we haven't covered yet. It is called `JnS X`, which stands for "Jump and Store", and it's main purpose is to enable writing *subroutines.*

A subroutine, also known as a procedure, function or method in other programming languages, is a piece of code that

- has a well-defined purpose or function
- needs to be executed often
- we can *call* from our code, passing *arguments* to it
- *returns* to where it was called from after it has finished, possibly with a *return value* (or *result)*

Examples for common subroutines in high-level programming languages are `System.out.println` in Java, which takes a string as its argument and prints it to the console, or `math.log` in Python, which computes the logarithm of its argument and returns it.

Subroutines are probably the most important concept in programming: they allow us to *structure* a program, breaking it up into small parts. Of course, since all high-level languages are, in the end,

executed by machine code instructions, most ISAs have instructions that make it easy to implement subroutines directly in machine code.

In MARIE, `JnS X` stores the address of the next instruction (i.e., the one immediately after the `JnS`) into the memory location X. It then continues execution at address X+1. The actual subroutine code is then stored at X+1, and it concludes with the instruction `JumpI X` (an *indirect jump*) that jumps back to the address stored at X.

Here is an example MARIE program that uses a subroutine to print some user input.

```
            Input
            Store Print_Arg
            JnS   Print
            Input
            Store Print_Arg
            JnS   Print
            Halt
            / Subroutine that prints one number
Print_Arg, DEC 0              / put argument here
Print,     HEX 0              / placeholder for return address
           Load Print_Arg
           Output
           JumpI Print        / return to caller
```

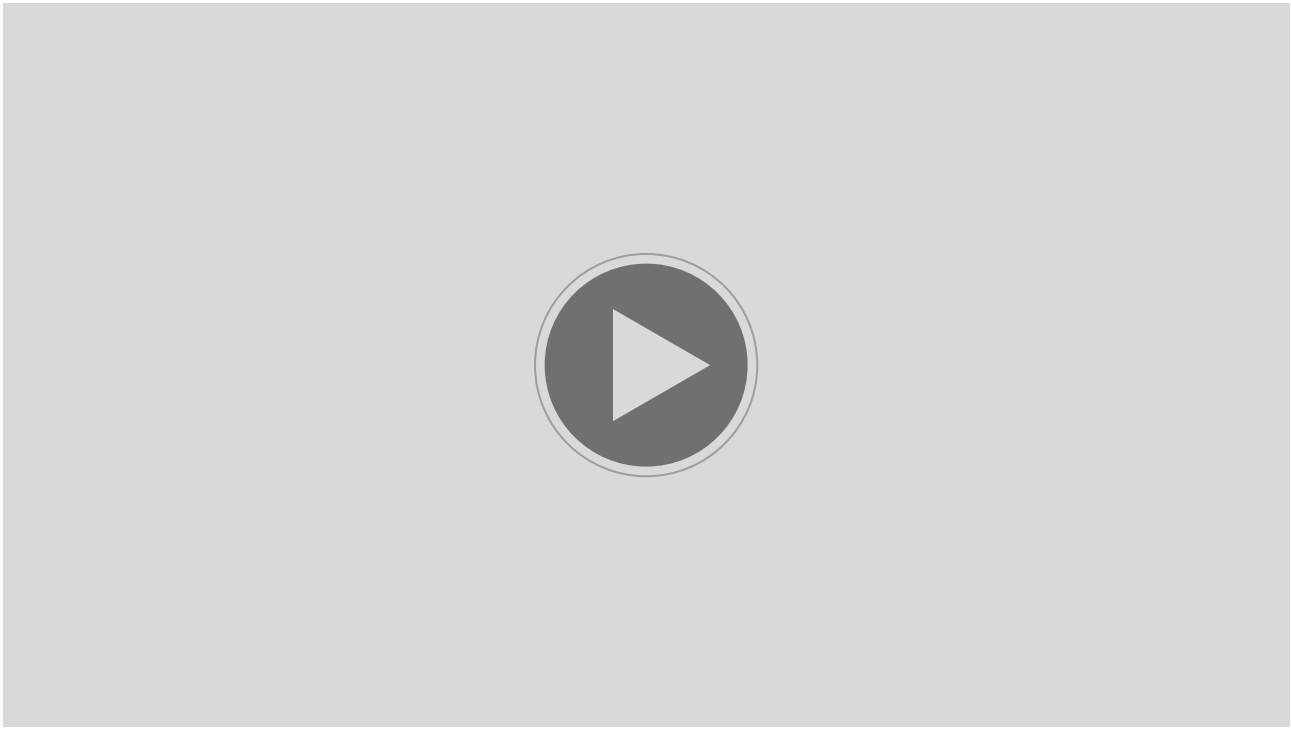
This code gets a user input and stores it in the location where the subroutine expects its argument. The `JnS Print` instruction then saves the *return address,* i.e., where the program should continue after the subroutine is finished, into address `Print` (where we put a placeholder `HEX 0`), and jumps to address `Print+1`, where the actual subroutine starts. In this case the `JnS` on line 3 will store the address `0003` in `Print` (because that's the address of the *next* instruction, the `Input` on line 4).

The subroutine in this case just loads the argument from memory and outputs it. It then uses an indirect jump, `JumpI Print`, to jump back to the address pointed to by `Print`: remember that's where `JnS` saved the address `0003`. Now we can do the same thing again on lines 4-6. This time, the `JnS` will store `0006` as the return address (the address of the `Halt` instruction).

Run this code in the MARIE simulator to get a better intuition for how it works.

We can explain in a little more detail how `JnS X` works. It first stores the current value of the PC register (which points to the instruction after the `JnS`!) into X. It then sets PC to X+1, causing the CPU to continue execution there. The `JumpI X` instruction then sets the PC to the value stored at location X. This causes the execution to resume at the instruction right after the subroutine call.

The following video explains subroutines again step by step.



(https://www.alexandriarepository.org/wp-content/uploads/20170120155041/subroutines.mp4.mp4)

Wheeler Jumps

This is **additional** material that is **not going to be assessed**. It's just provided in case you want to learn more about an alternative implementation of subroutines.

Subroutines were invented very early in the history of computers. One of the people credited with inventing the concept is *David Wheeler*, who worked on the EDSAC machine at the University of Cambridge, UK, in the early 1950s.

But EDSAC didn't have instructions that use indirect addressing - so how was it possible to implement subroutines in EDSAC? The trick that Wheeler developed is to use *self-modifying code*, which means that the code overwrites instructions in memory to change its own behaviour.

The so-called *Wheeler Jump* consists of three steps:

1. The calling program loads the return address into the register (let's call it AC as in MARIE).
2. The subroutine overwrites its own final instruction with a jump instruction to the address stored in AC.
3. When the subroutine finishes, the new jump instruction continues execution where the calling program left off.

We can use the Wheeler Jump technique in MARIE to implement subroutines without `JnS`:

```
JumpFrom,   Load JumpFrom      / Load instruction "Load JumpFrom"
            Jump Sub           / Execute subroutine
            Halt

Sub,        Add Wheeler        / Add "magic number"
            Store SubReturn    / Overwrite final subroutine instruction
```

```
            Load FortyTwo      / Do something
            Output             / ... that's all
SubReturn,  HEX 0              / This is where the return jump goes

FortyTwo,   DEC 42             / Just some data
Wheeler,    HEX 8002           / Magic number that turns Load X
                               / into Jump X+2
```

So how does it work? The code uses two neat tricks. The first is in the first line: The instruction loads itself! So the value of the accumulator after executing the instruction is the bitpattern that represents `Load JumpFrom`.

The second trick is that the subroutine adds a "magic" number to the accumulator when it starts. The magic number is hexadecimal `8002`. A `Load` instruction is always of the form 1XXX, where 1 is the opcode for `Load`, and XXX is the address to load from. So adding `8002` to 1XXX results in 9YYY, where YYY is two greater than XXX. This encodes a `Jump` instruction (opcode 9) to the address two greater than the original `Load JumpFrom` - right behind the jump to the subroutine, and exactly where execution must continue upon return.

Try it out in the MARIE simulator!

↩1 0000000010001110, or 142_{10}

↩2 It's the sum of the values stored in locations 4 and 5.

↩3 3598 (or, in binary, `0000111000001110`)

4.2.3
From Instructions to Circuits

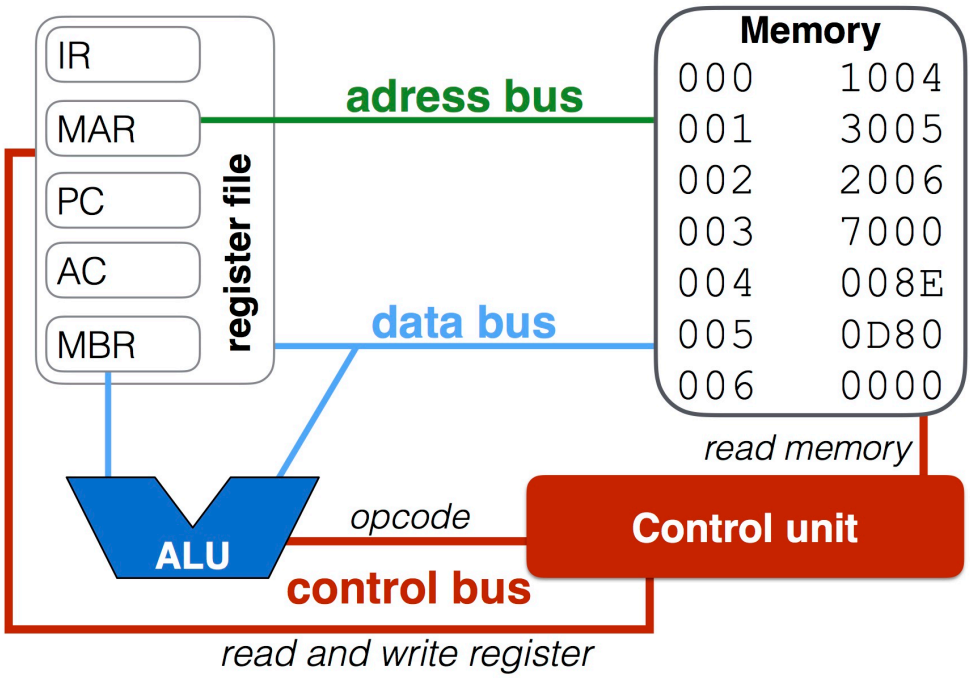
The MARIE instruction set is very simple, but it contains enough instructions to explain the basic functions of most modern CPUs, such as memory access, arithmetic, jumps, conditionals, and input/output. In the previous module, we have described what each instruction does quite informally. For example, the assembly instruction Add X "adds the value stored at memory address X to the current value stored in the AC register". That description is not very detailed. How does the CPU find out that this is an Add instruction? How does the it tell the memory that it should use address X? How does it tell the arithmetic/logic unit (ALU) to perform an addition? Answering these questions will get us closer to the actual hardware, and enable us to design correct circuits for these tasks in the following modules.

The three concepts introduced in this module are **data paths**, **register transfer language (RTL)**, and **control signals**. The data paths describe how information flows through the CPU. RTL is a more formal way of describing the individual steps that the CPU must take to fully execute an instruction. Control signals can be thought of as "wires" in the CPU that the Control Unit can use to switch certain components on and off.

Data Paths

The data paths in a CPU describe how the different functional units, in particular the registers and the ALU, are connected. The hardware implementation of the data paths is the **system bus** (or simply **bus**), the set of "wires" in the CPU that connects all components. We will use the MARIE architecture as an example here, but most modern architectures are quite similar.

The module on MARIE mentioned that the architecture has five registers. We have seen how the AC register is used as a temporary storage location for almost all instructions, so let's now explain the function of the remaining registers. The following figure illustrates the data paths in the MARIE architecture.



The blue *data bus* can transport individual words of data between the memory, the registers and the ALU. What is not shown in the picture is that the *only* way to get a data word from memory into a register, or from the registers into memory, is via the *memory buffer register* MBR. So even for an instruction like `Load 005` the CPU can't directly transfer the value at address `005` into the AC register, it first has to load it from memory into MBR, and then from there into AC.

The green *address bus* connects the memory with the MAR, the *memory address register.* It is responsible for selecting the memory address that the CPU reads from or writes to. Let's take the example of `Load 005` again: the CPU has to put the value `005` into MAR, which tells the memory to "activate" address `005`.

The red *control bus* doesn't transport addresses or data words. It is used by the Control Unit (CU) to select different modes of operation on the other components. For example, it is not enough for the memory to know that the address it should use is `005`, it also needs to know whether it should transfer the current contents of `005` into the MBR (as for a `Load` instruction), or rather go the opposite direction and transfer the current data word from MBR into address `005`. The control unit can therefore switch the memory from "read mode" into "write mode", and that switch is one of the signals on the control bus. The CU also needs to be able to tell the ALU which operation to perform, e.g., whether to add or subtract. This is also done via the control bus. Finally, the CU needs to select which register to read from and write to. E.g., in a `Jump 102` instruction it would have to write the value `102` into the PC register, but in many other instructions, it has to read from or write to the AC register.

The PC and `IR` registers are used by the CU to keep track of which part of our program is currently executing. The `IR` contains the currently executing instruction, while the PC contains the *address* of the *next* instruction to be executed after the current one has finished.

The data paths are just the general *architecture*, i.e., they show us how the different components are connected. The next section defines, for each individual instruction, in what order data and addresses need to be transferred between registers and memory.

Register Transfer Language

Assembly language (and machine code) already look quite low-level. You may have the impression that they are the "atomic" operations that CPUs work with, i.e., that each instruction is executed by the CPU in a single step. That, however is not the case, and we already got a glimpse of this idea in the module on CPU basics and History: The *fetch-decode-execute* cycle defines a sequence of even lower-level steps that each instruction can be broken down into. We will now introduce *register transfer language* (RTL), which helps us define these lower-level steps.

Let us start with the ***fetch***. Here is a quick recap of what it does. The PC register contains the address of the next instruction that needs to be executed. The control unit needs to load the next instruction from the memory address stored in PC into the `IR` register, and then increment the PC register, so that it points to the next instruction again. Recall from the discussion above that loading from memory means that the address must be put into the MAR, and the result will then be transferred into the MBR - but we need the result in the `IR`, so we need another transfer from MBR into `IR`. That means that even for the *fetch* stage of each instruction, the CPU already does four small steps. Note that each of these steps involves the transfer of some data between registers or between registers and the memory. In register transfer language (RTL) we would write down these steps as follows:

```
1. MAR ← PC
2. MBR ← M[MAR]
3. IR ← MBR
```

```
4. PC ← PC+1
```

Each line corresponds to one transfer from the register (or memory address) on the right into the register on the left of the arrow.

The first step copies the address stored in the PC into MAR. The next step reads the memory at location MAR - we write this `M[MAR]` - into MBR, before the third step copies the contents of MBR into the IR. Now the instruction that is stored at the address in PC has been fetched into the instruction register IR. The final step increments the PC. Note how we can write simple arithmetic expressions on the right hand side, which will have to be performed by the ALU.

The next phase in the cycle is ***decode***, where the control unit looks at the instruction in the IR and figures out what needs to happen. Most instructions contain an address X, and since we have a special register for handling addresses (the MAR), the first step in the decode phase is to put X into MAR.

```
5. MAR ← X
```

If the instruction needs to read any data from memory, then that is also done in the decode phase. In the case of MARIE, the instructions that read from memory are `Load X`, `Add X`, `Subt X`, and all indirect addressing instructions. For these instructions, the final step of the decode phase is therefore to read the data word from memory into the memory buffer register MBR:

```
6. MBR ← M[MAR]
```

Now we are ready for the final phase, ***execute***. This phase, naturally, depends on the actual instruction being executed. Here is a table with the most common instructions:

Instruction	RTL
`Load X`	`7. AC ← MBR`
`Store X`	`6. MBR ← AC` `7. M[MAR] ← MBR`
`Add X`	`7. AC ← AC + MBR`
`Subt X`	`7. AC ← AC - MBR`

`SkipCond X`	`6. If MAR=0x800 and AC>0 then PC ← PC + 1`
	`If MAR=0x400 and AC=0 then PC ← PC + 1`
	`If MAR=0x000 and AC<0 then PC ← PC + 1`

`Jump X`	`6. PC ← MAR`
`Clear`	`5. AC ← 0`

`AddI X`	`7. MAR ← MBR`
	`8. MBR ← M[MAR]`
	`9. AC ← AC + MBR`

`JumpI X`	`7. PC ← MBR`

`LoadI X`	`7. MAR ← MBR`
	`8. MBR ← M[MAR]`
	`9. AC ← MBR`

`StoreI X`	`7. MAR ← MBR`
	`8. MBR ← AC`
	`9. M[MAR] ← MBR`

`JnS X`	`6. MBR ← PC`
	`7. M[MAR] ← MBR`
	`8. AC ← MAR`
	`9. PC ← AC + 1`

Note how the numbering of the steps starts at 5, 6 or 7, depending on the decode phase for the

instruction. For example, the `Clear` instruction doesn't have an X parameter, so its execute phase starts at step 5. The `Store X` instruction's decode phase copies X into MAR, but doesn't read the memory from that address, so the execute phase for `Store X` starts with step 6. The `Load X` instruction, on the other hand, does need to read the memory contents, so its execution starts with step 7.

Let's take a look at a couple of instructions that are a bit more complicated.

The `SkipCond X` instruction just implements a simple case distinction, looking at the contents of MAR (which now contains X after the decode step 5) and AC.

In the `JumpI X` instruction, after the decode phase we have the contents of memory location X in the MBR register. Since we want to do an *indirect* jump, this is in fact the address we want to jump to, so we simply transfer it into the PC register so that the next fetch cycle continues from there.

For the `LoadI X` instruction, the decode phase has also delivered the actual address we want to load from into MBR. So we transfer this address into MAR in step 7, then read the memory at that address in step 8, and finally transfer the result into the AC register.

The `JnS X` instruction needs to first store the current PC at memory location X and then jump to X+1. The address X has already been transferred into MAR by the decode phase, so the next step is to copy the current PC into MBR in step 6, so that we can store it into memory in step 7. Step 8 copies MAR (which still holds the address X) into AC, so that we can add 1 to it and transfer the result to PC, implementing the jump to X+1. An interesting fact to note here is that we use the AC register because it is connected to the ALU, so that we can perform the addition of 1. This means that a `JnS` overwrites whatever is stored in AC before. This is important to know, because it means that we can't use the AC register to pass arguments to a subroutine! A different implementation would have been to use the following sequence of steps: `8. PC ← MAR`, `9. PC ← PC+1`. This would avoid the so-called "clobbering" of the AC register, but would have required slightly more complicated hardware to implement, which is why the designers of MARIE opted for the approach using AC.

The only instructions missing from this table are `Halt`, `Input` and `Output`. The `Halt` instruction simply halts execution and therefore doesn't do any register transfers. The `Input` instruction transfers the input value into AC and `Output` transfers the value in AC to the output device - we'll look into I/O in a later module.

Control Signals

Before we move on to implementing this architecture using actual circuits, we will add one extra level of information to the RTL, the *control signals* that the control unit needs to generate in order to implement each RTL step.

Each control signal is a particular set of "wires" of the control bus that can switch a component on or off or select its mode of operation (for example addition vs. subtraction in the ALU). The following table explains the different control signals:

Signal	Signal wires	Number of bits	Possible values

Register read	$P_2P_1P_0$	3	000 (None) 001 (MAR) 010 (PC) 011 (MBR) 100 (AC) 111 (IR)
Register write	$P_5P_4P_3$	3	000 (None) 001 (MAR) 010 (PC) 011 (MBR) 100 (AC) 111 (IR)
Memory read	M_r	1	0 or 1
Memory write	M_w	1	0 or 1
ALU operation	$A_2A_1A_0$	3	000 (nothing) 010 (add) 001 (subtract) 011 (clear) 100 (increment by 1)

Now we can annotate each RTL step of an instruction with the control signals that need to be activated for that step. We will only list the signals that need to be 1, all other signals are assumed to be 0. For the Add X instruction, the individual steps would look like this:

```
1. MAR ← PC                    P3    P1
2. MBR ← M[MAR]             P4 P3          Mr
3. IR ← MBR              P5 P4 P3    P1 P0
4. PC ← PC+1                P4       P1          A2
5. MAR ← X                     P3 P2 P1 P0
6. MBR ← M[MAR]             P4 P3          Mr
7. AC ← AC + MBR         P5       P2             A1
```

Let's go through the steps one at a time.

1. We need to enable MAR for writing which means the signals P5P4P3 need to be set to 001. Since only P3 is active (it's the only 1 in the pattern), we write down only that one signal. Similarly, we activate PC for reading, which means P2P1P0 need to be set to 010, so only P1 is active. All other signals are inactive.
2. Activate MBR for writing, which means P5P4P3 needs to be 011, so only P4 and P3 are active. We read from memory, which means activating the Mr signal.
3. In order to transfer MBR into IR, we activate IR for writing (111 or P5P4P3) and MBR for reading (011 or P1P0).
4. We have to use the ALU to increment the PC. This is done by activating the PC for both reading and writing (010, or P4 for writing and P1 for reading), and by switching the ALU into increment mode (100 for the signals A2A1A0, just represented by A2 here).
5. This part of the decode phase is interesting: we want to transfer the address X into the memory address register MAR. Remember that the X is part of the instruction (the 12 lowest bits), which is currently stored in the IR register. That means that we activate MAR for writing and IR for reading. Since this is the only time that we ever activate IR for reading (no other RTL step uses IR!), we can implement this as a special case in the hardware: When we activate IR for reading, the topmost four bits (which contain the opcode) are always read as 0. That way, only the X part of the instruction will be transferred.
6. This is exactly as step 2.
7. Finally, we execute the Add instruction by activating AC for reading and writing (100, or P5 and P2), and activating the addition mode of the ALU (A2A1A0 set to 010, which means only A1 is activate).

> You may wonder how we select MBR for the second argument to the addition. The ALU is simply hard-wired to always read MBR as the second argument. If you look carefully through all the possible RTL steps, you will see that no instruction requires a different second argument!

Each RTL step for each instruction can now be expressed as a set of control signals that need to be activated for that step. Clearly, we're getting much closer to the hardware, and indeed this is the level of detail that we will need in order to implement the Control Unit in the module on Constructing a CPU.

4.2.4
Basic circuits

We will now introduce some very fundamental digital circuits, which we can then combine into an almost realistic model of a complete CPU. Although modern CPUs are very complex (they consist of billions of transistors), we only need to understand a few basic functions to get a really good idea of how a CPU works:

1. The Arithmetic/Logic Unit (ALU) in a CPU needs to be able to perform simple math: addition, subtraction, some logic operations on word-size data.
2. The Registers in a CPU need to be able to store instructions and data words.
3. The Control Unit (CU) in a CPU needs to run the fetch-decode-execute cycle.

This may be a good time to quickly revisit the topics mentioned above, as well as the basic Boolean gates: AND, OR, NOT, and NAND.

Circuits

A *circuit,* for our purposes, is a collection of Boolean gates that are connected by "wires". Circuits typically include a number of *inputs* and *outputs*. For example, a circuit that can add two 8-bit binary numbers may have 16 input wires for the two numbers, the gates implementing the actual logic, and 9 output wires (since the result of an addition could require one more bit to be represented than the inputs).

The remainder of this module is split up into two parts: *combinational* circuits, which perform simple functional computations, and *sequential* circuits, which can be used to implement memory.

4.2.4.1 Combinational circuits

In a combinational circuit, the outputs only depend on the inputs. We therefore say that the circuit computes a simple function of the inputs. The most basic case would be an individual gate. For example, consider a circuit consisting of a single AND gate with two inputs and one output. The value at the output only depends on the two inputs, and we can say that the circuit computes the function defined by the truth table of the Boolean AND.

We will now look at how we can combine simple gates to compute more interesting functions.

Adders

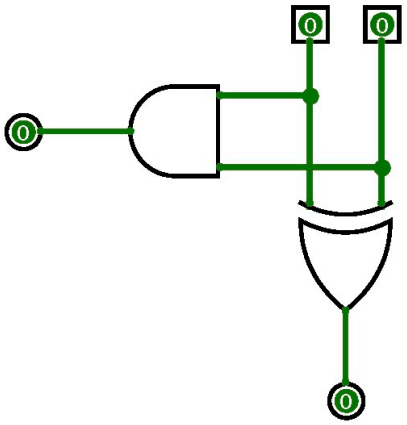
Let's start with something very simple: adding two bits, let's call them **A** and **B**. We can easily list all possible outcomes for any combination of inputs: 0+0=0, 0+1=1, 1+0=1, 1+1=2. Of course the output needs to be in binary, so in fact we should write $1+1=10_2$. We can see that we will need two input wires (one for each input bit) and *two* output wires: one for the "result" of the addition, and one for the "carry bit", i.e., the bit that "overflows" into the next position. Let's put this in table form to make it more readable:

A	B	Carry	Result
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Half adders

Now that we have a truth table, it's quite easy to construct a circuit that *implements* this function, by looking at the outputs. First, the **Carry** output. It is 1 if and only if *both* inputs are 1, and otherwise it is 0. This is easy: it's exactly what an AND gate does. So we can use an AND gate to compute the value of the carry. Now for the **Result** output. We can see that it is 1 if and only if *one* of the inputs is 1, but not if both inputs are 0 or both are 1. Does that ring a bell? Reveal [1]

So we can put the two together and construct a so-called *half-adder circuit:*



You should try this out using the Logisim tool, and observe what happens to the outputs when you change the inputs.

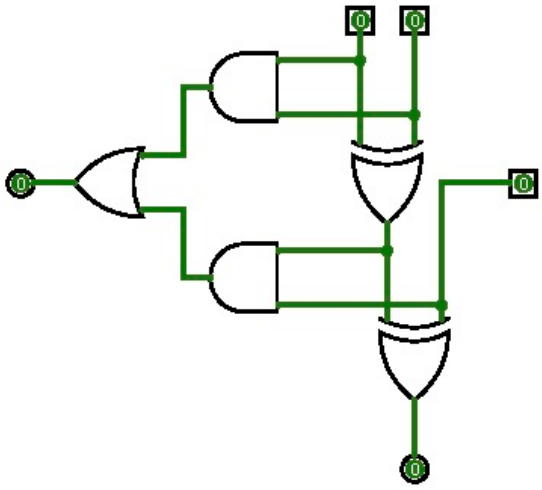
Full adders

Unfortunately, a half adder is only half useful. All it can do is add up two bits, but it produces two new bits as output, so we can't use it to construct adders for larger numbers. To do that, we need a circuit that can add *three* bits (two "real" inputs and a carry-in). The result can still be represented in two bits (remember that $1+1+1=11_2$).

Here's the truth table for a three-bit addition:

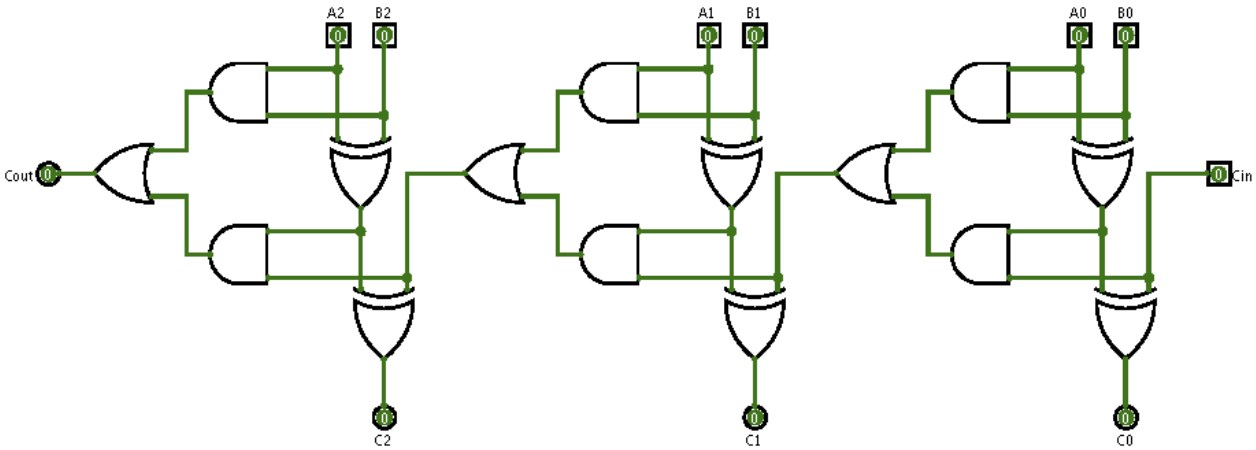
A	B	Carry-in	Carry-out	Result
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

The corresponding circuit is called a *full adder.* We can go through each output again to deduce what kind of gates we can put together to compute it from the inputs (we leave this to you as an exercise), but in effect we're simply *combining two half adders!* The output of the first one, together with the carry-in, is fed into the two inputs of the second one. The two carry-out outputs of the two half adders are then simply combined using an OR gate. This is correct because only one of the two can generate a carry-out. Here's the circuit diagram for a full adder as you would construct it in Logisim:



Ripple-carry adders

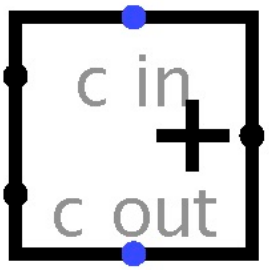
The full adder circuit may seem very primitive (when do we ever want to add up individual bits?). But because it can deal with a carry-in and produces a carry-out,we can put several full adders together to create a circuit that can add longer binary numbers. For example, if we want to add two 3-bit numbers, we can construct a chain of three full adders like this:



The output is a 3-bit result plus a carry-out. Note that the first full adder (the rightmost one) doesn't require a carry-in, so we'll just set that input to 0 (or we could use a half-adder instead). Also note how the carry-out of each adder is fed into the carry-in of the next adder. That's why we call this type of adder a *ripple-carry-adder*. You should try this out in Logisim to get a feel for how the data flows through the circuit when you update individual input bits.

Standard circuit symbols

When we construct more complex circuits, we often want to hide the details of parts that are well understood. So e.g. if we need an adder in a larger circuit, we don't want to see all the individual gates - we just want to know that it is an adder. That's why there is a standard symbol for adders:



It has two inputs on the left (each input represents multiple bits at once, and the adder can be configured to any bit-width you need), a carry-in (at the top) and a carry-out (at the bottom), as well as of course an output of the result on the right (again representing multiple bits in a single output).

Delay and circuit efficiency

The circuits we have constructed so far seem to operate instantaneously: when you change an input (e.g. in the Logisim simulator), the outputs react immediately. This is not true for real circuits! Any change to an input of a gate requires a tiny amount of time to *propagate* to the output, i.e., the output only changes to the correct result after this so called *propagation delay.* For an entire circuit, the overall propagation delay is the time it takes from *any* input being changed to *all* outputs being correct. In general, for combinational circuits, this will be the sum of all the individual gate delays *on the longest path through the circuit,* i.e., the path that goes through the largest number of gates.

Have a look at the 3-bit ripple-carry adder again. The longest path through the circuit starts at the **B0** input, goes through the XOR gate, the AND gate and the OR gate of the first full adder, then through the AND and OR gates of the second and third full adders. So in total, if the **B0** input changes, seven gates need to update their output before the overall output is correct.

Since this path is always the longest path no matter how many full adders we chain together, we can compute the propagation delay for an n-bit ripple carry adder as $2n+1$.

The propagation delay of basic circuits such as adders is crucial for the performance of a CPU (or any complex digital circuit for that matter). For example, an addition in a CPU should normally not take longer than a single clock cycle. Let's assume a clock frequency of 1 GHz, i.e., one billion cycles per second. A single clock cycle then takes 1 ns (nanosecond). If the CPU contained a 32-bit ripple carry adder in its ALU, the propagation delay would be 65, meaning that within 1 ns, 65 gates have to fully propagate their value. The delay of a single gate must therefore not be longer than 1/65 of a nanosecond, or 15 picoseconds. (This is a very rough calculation just to give you the basic idea.) So the propagation delay of all the circuits in a CPU and the CPU's clock frequency are quite closely linked!

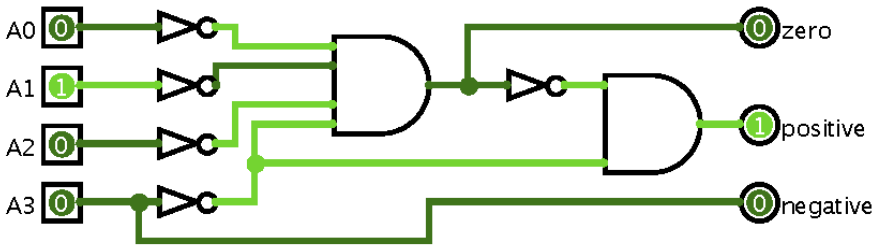
Since the efficiency of these basic circuits is so important for the raw speed of a CPU, circuits such as adders have been highly optimised. The trick to make adders more efficient is to avoid having the carry ripple through all bits. Instead, we add more gates to pre-compute whether entire groups of bits will generate a carry. We're not going to look into efficient adders in any more detail in this unit. You just need to understand how a ripple-carry adder works, and why it may be inefficient.

Comparators

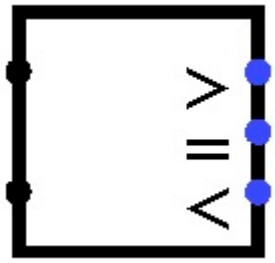
Another important operation that can be implemented using combinational circuits is the *comparison* of two numbers. In the MARIE instruction set, the `SkipCond` instruction compares the current contents of the AC register to the value zero. That's in fact all we need, since any comparison between two numbers *A* and *B* can be expressed as the comparison of the difference *A*-*B* to zero.

So let's construct a circuit that can compare an *n*-bit twos' complement number *A* with zero. We have to distinguish three cases. If *A* is equal to zero, all its bits must be 0. We can test that using *n* NOT gates and an *n*-bit AND gate. If *A* is less than zero, then its leftmost bit must be 1 (because *A* is represented in twos' complement!). If *A* is neither equal to nor less than zero, it's greater than zero.

Here is a circuit that implements this idea for a 4-bit number *A*:



The standard symbol for a comparator actually works with two inputs *A* and *B*, so it computes the difference *A*-*B* internally before employing the logic discussed above:

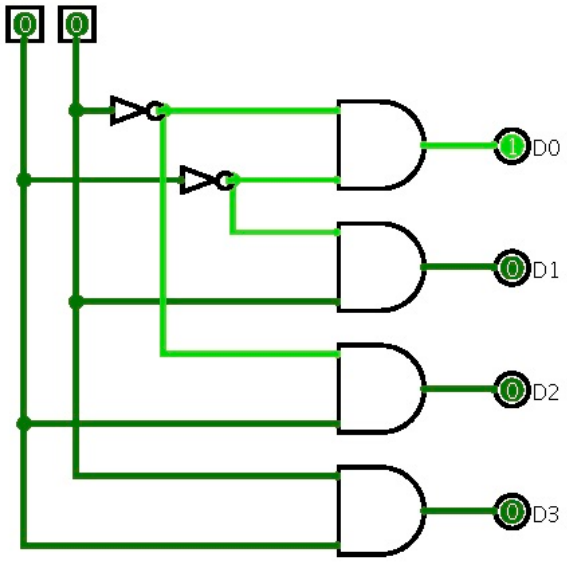


Decoders

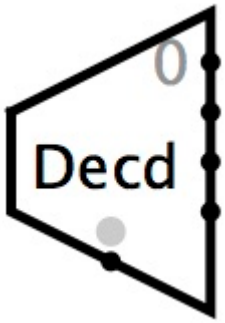
Combinational circuits can also be used to do things other than arithmetic. One circuit that is used quite often is a so-called *decoder*, which basically turns a binary number into a *unary* representation. It has *n* inputs and 2^n outputs, and it activates the output that corresponds to the binary number that is present at the input. So, e.g., a 2-bit decoder would have 2 inputs, and if the first input is 0 and the second one is 1, this corresponds to the binary number $10_2=2_{10}$, so the third output (we start counting at 0 here!) would be activated to 1, while all other outputs stay at 0. Here is the truth table:

Input 0	Input 1	Output 0	Output 1	Output 2	Output 3
0	0	1	0	0	0
1	0	0	1	0	0
0	1	0	0	1	0
1	1	1	0	0	1

A 2-bit decoder can be implemented with just a few NOT and AND gates:



Here is the standard symbol for a decoder:



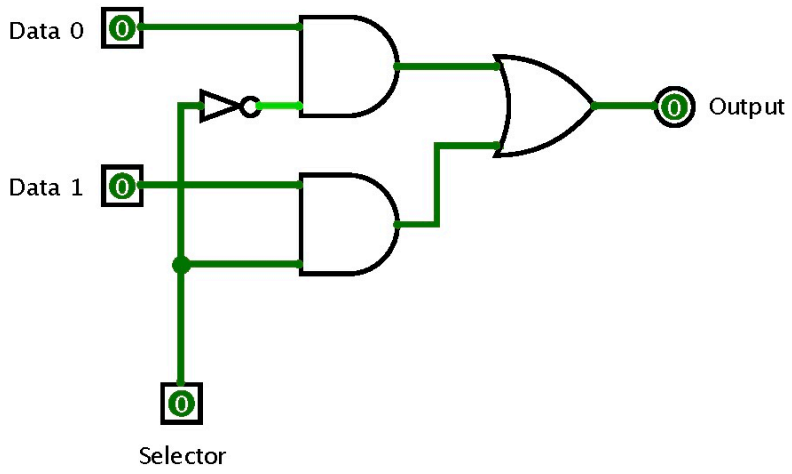
This one has a 2-bit input at the bottom, and four outputs (In Logisim, you can combine several wires into a multi-bit wire like the 2-bit one here, in order to make the diagram look cleaner).

Multiplexers

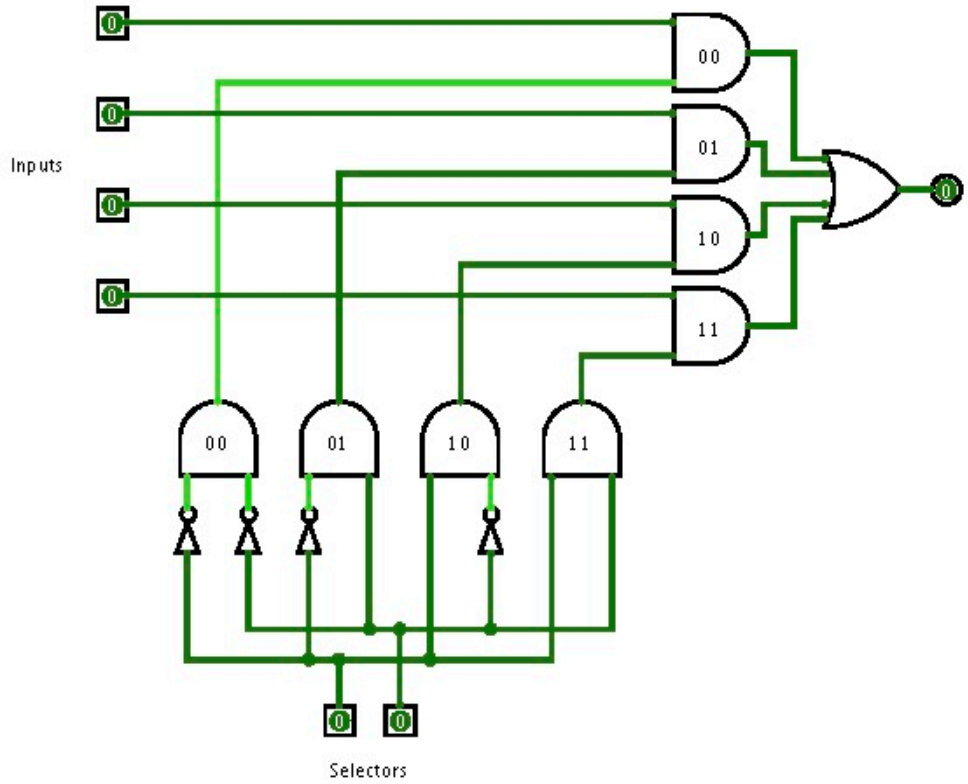
The next combinational circuit we're going to look at is called a *multiplexer*. Its function is to *select* one out of several data inputs. In order to do this, it has *n selection inputs*, which determine which one of 2^n the data inputs to pick. The simplest form of multiplexer would have two data inputs, one selector, and one output. The truth table would look like this:

Data 0	Data 1	Selector	Output
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	0
1	1	1	1

As you can see, if the selector is 0, then the output is equal to data input 0. If the selector is 1, the output is equal to data input 1. We can turn this into a simple circuit:



In general, multiplexers have 2^n data inputs, n selectors, and one output. The basic scheme looks very much like the one above, except that we need to *decode* the selector first. Here's an example of a 4-bit multiplexer. Note how the bottom row is essentially a 2-bit decoder (we could have used just two NOT gates instead of the four):



Just like for decoders, we also introduce a standard symbol for multiplexers:

The four inputs are on the left hand side, the 2-bit selector is on the bottom, and the single output is shown on the right hand side.

↩1 This is exactly the XOR function.

4.2.4.2
Sequential circuits

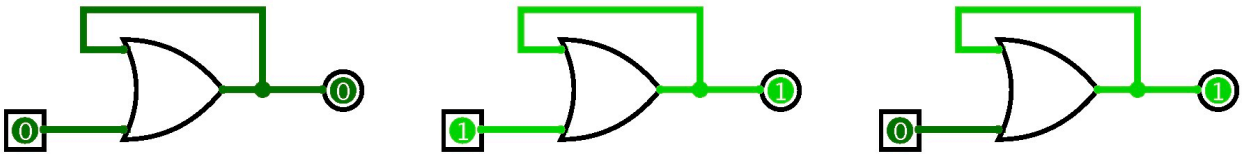
The combinational circuits in the previous module all had in common that they act like mathematical functions: the outputs are determined only by the inputs. In particular, those circuits could not *store* any information, because storage means that the behaviour of the circuit depends on what has happened in the past.

This module introduces *sequential* circuits, who can do just that: their outputs depend on their inputs *and* on the *sequence* of events (i.e., inputs and outputs) that has happened in the past.

Feedback

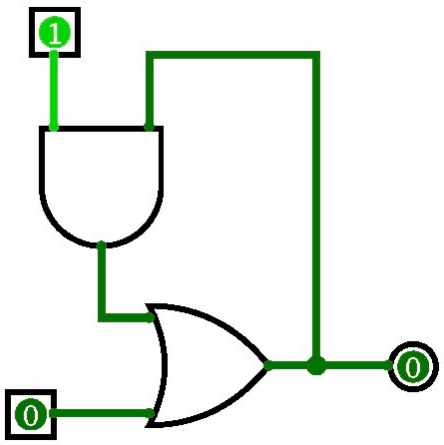
The main mechanism that allows a circuit to remember the past is to **pass its output back into its input.** That way, we can establish a feedback loop.

In its simplest form, a feedback loop could feed the output of an OR gate back into its input:



The diagram shows a sequence of three steps. In the first step, both input and output are 0. If we now set the input to 1 (the second step), the output of course also becomes 1. But if we now reset the input to 0, the 1 from the output is still feeding into the other input of the OR gate, which means that the output stays 1. This circuit therefore remembers **whether the input has ever been set to 1.**

This is a very simple form of memory, but arguably not a very useful one. So let's add a switch to control the state.

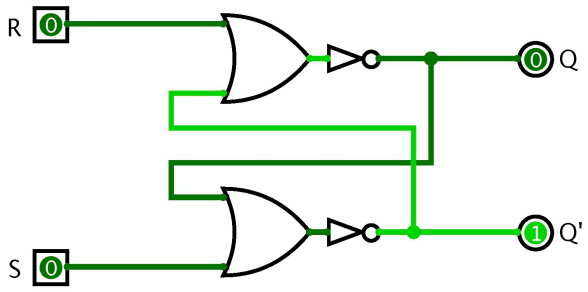


The switch (the input at the top of the circuit) now controls the memory. If it is set to 1 (as in the picture),

the circuit behaves as before, and as soon as we set the other input to 1, that 1 will be "stored". However, think about what happens if the output is 1 and we set the switch to 0: the AND gate now will output 0, so we can reset the stored state. Try this out in Logisim!

Flip Flops

By extending the idea of a storage cell with a switch a bit further, we can construct a famous basic memory component, the **S/R latch** (set/reset latch). Its circuit diagram looks like this:



You can see that it uses the idea of feedback, but in a more complicated way than above. The latch consists of two halves, containing an OR gate and a NOT gate each. When an output is 1, like the bottom output in the picture, it forces the other output to be 0, by feeding the 1 back into the other OR and NOT gate. Further, the other output, the 0, is fed into its own input OR gate.

Now let's think about what happens when the bottom input is switched to 1. The bottom OR gate outputs 1, the NOT gate negates it, and the bottom output becomes 0. At the same time, the top OR gate now gets fed the 0, its output becomes 0, and the top NOT gate negates it so that the top output becomes 1. We have switched the latch from 0/1 to 1/0! Even if we now switch the bottom input back to 0, the top output is fed into the bottom OR gate, so the stored state remains the same. The circuit is completely symmetric, so we can switch it back to 0/1 by toggling the top input from 0 to 1 and back to 0.

As for all circuits, it's best to try this out in the simulator to really understand it!

For combinational circuits, we have always used truth tables to describe their function. It turns out that we can extend truth tables to also work for sequential circuits, if we list the outputs at a previous point in time as inputs in the table. The truth table for an S/R latch looks as follows:

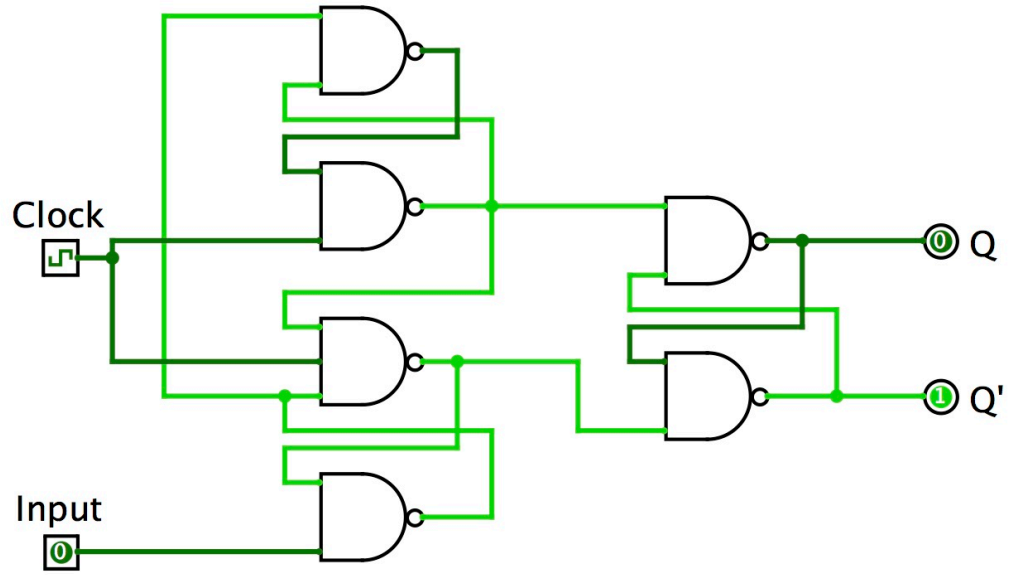
R	**S**	**Q(t)**	**Q(t + 1)**
0	0	0	0
0	1	0	1
1	0	0	0
1	1	0	forbidden
0	0	1	1
0	1	1	1
1	0	1	0
1	1	1	forbidden

The **S** and **R** inputs stand for "Set" and "Reset", they are the two actual inputs of the circuit. **Q(t)** stands

for the top output *at the previous point in time (t)*, and **Q(t+1)** is the new output at time t+1. The table tells us that if the output is currently 0 (i.e., **Q(t)**=0), then the only way to make it 1 is by setting **R** to 1 (resetting the latch). If the output is currently 1, then the only way to get it back to 0 is by setting **S** to 1 (setting the latch). Note that the combination **R**=1, **S**=1 is not allowed.

This simple S/R latch can be used to store a single bit of data. However, in order to use it e.g. for implementing a register in a CPU, it is missing an important function. In a CPU, rather than setting and resetting a latch using two different signal wires, we would like to store a bit that is represented on one individual wire. For instance, we would like to combine 4 storage cells that can each hold one bit, and connect that to the 4-bit output of our simple ALU to store whatever result it produces. This functionality is provided by an extension of the S/R latch called the *D-Flip-Flop.* A flip-flop can also store a single bit of information, but it reads the bit from a single input line, and it has an additional control line (called the "clock") to select whether the stored bit should change or stay the same.

A D-Flip-Flop circuit looks like this:



The input is the data bit that is supposed to be stored, and the output **Q** is the data bit that is currently stored in the flip-flop. The output **Q'** is not really required, it simply outputs the negation of the stored bit. The interesting input is the **clock**: The state of the flip-flop can only change on the "positive edge" of the clock, i.e., when the clock input changes from 0 to 1.

If you look at the circuit closely, you can see that it consists of three separate latches (in this case they are constructed using negated AND gates instead of the negated OR we used above). Without going into too much detail, the left two latches make sure that the input to the latch on the right is never the forbidden state, even if the clock and the input bit do not change exactly at the same time.

4.2.5 Constructing a CPU

We have now seen all of the basic circuits that we need to construct a (very simple) CPU. So let's do that!

Recall the basic Von Neumann architecture. We need an arithmetic/logic unit, a number of registers, and a control unit.

The Arithmetic-Logic-Unit (ALU)

The ALU is a combinational circuit (i.e., it computes a result as a function of its inputs). The ALU we want to construct here will be able to perform all the operations that are available in the MARIE instruction set:

1. Addition of two numbers (`Add` and `AddI` instructions)
2. Subtraction of two numbers (`Subt` and `SubtI` instructions)
3. Incrementing a number by 1 (To increment the PC register during the fetch-decode-execute execute)
4. Testing if a number is less, equal to, or greater than zero (`SkipCond` instructions). The result of the comparison will be 0 if the input number is equal to 0, 1 if it is greater than 0, and 2 (i.e., 10_2) if it is less than 0.

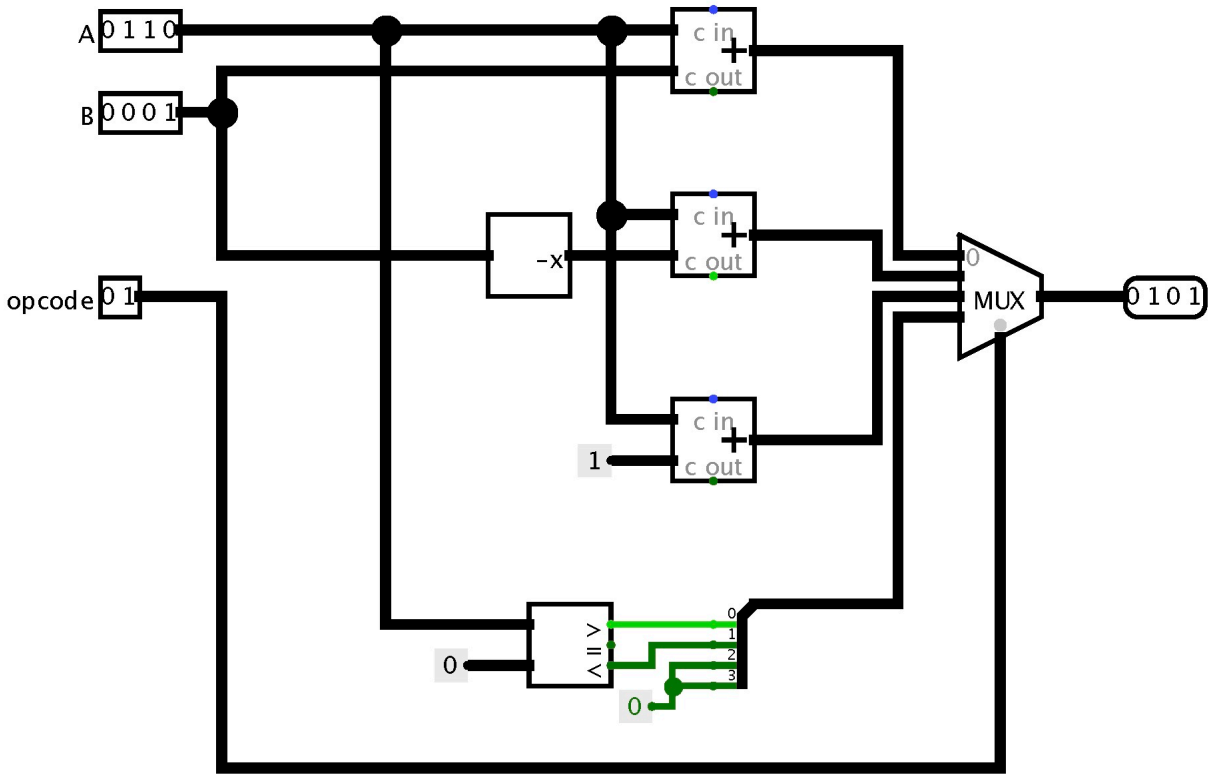
Such an ALU needs two *n*-bit data inputs (both are used in addition and subtraction operations, but only one is used in comparison and increment), and one 2-bit input for the so-called *op-code*, which selects the kind of operation (since there are four different operations, we can select the one we want using just two bits). The ALU has one *n*-bit data output for the result.

All four operations are easily implemented using adders and some additional logic:

1. Addition can be done using a ripple-carry adder (or any other, more efficient adder circuit).
2. Subtraction *A*-*B* can be implemented as *A*+*(-B)*, i.e., we have to first negate *B*. Our ALU (like most computers) will use twos' complement to represent negative numbers, so we need a circuit that can negate *B*. This is easy: flip all bits (using NOT gates) and then add 1 (using an *n*-bit adder).
3. Incrementing by 1 can be done using an adder circuit (like a ripple-carry adder) where we set one of the inputs to a constant 1.
4. To compare *A* with zero, we will use a comparator circuit. We just need to connect the "greater than" output to bit 0 of the result, and the "less than" output to bit 1, while connecting all other output bits to a constant 0.

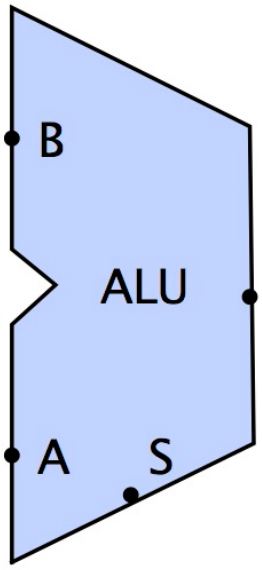
But how do we select which operation should be performed? The answer is really easy: just always perform all four, and then select the desired result (the one prescribed by the op-code) using a multiplexer!

Here is a complete 4-bit ALU that can perform the four operations needed for MARIE:



It currently shows opcode `01` selected, which is subtraction, so the output shows the result of 0110_2-0001_2=0101_2.

An ALU is typically represented with the following standard symbol, where the two data inputs are on the left, the opcode is connected to the *S* input, and the result is output on the right:

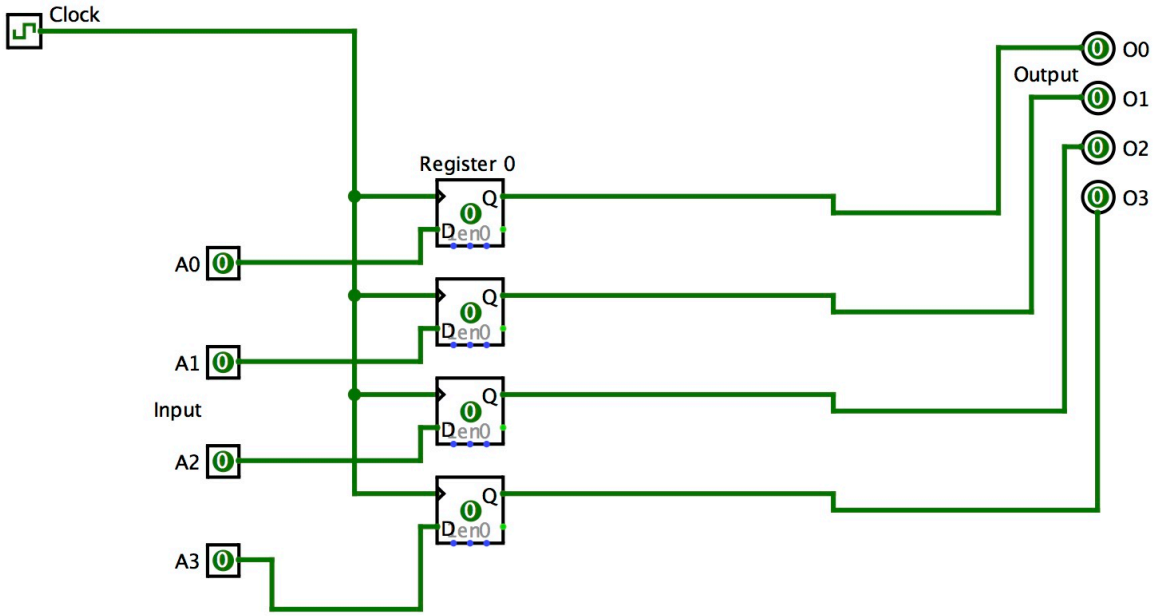


The Register File

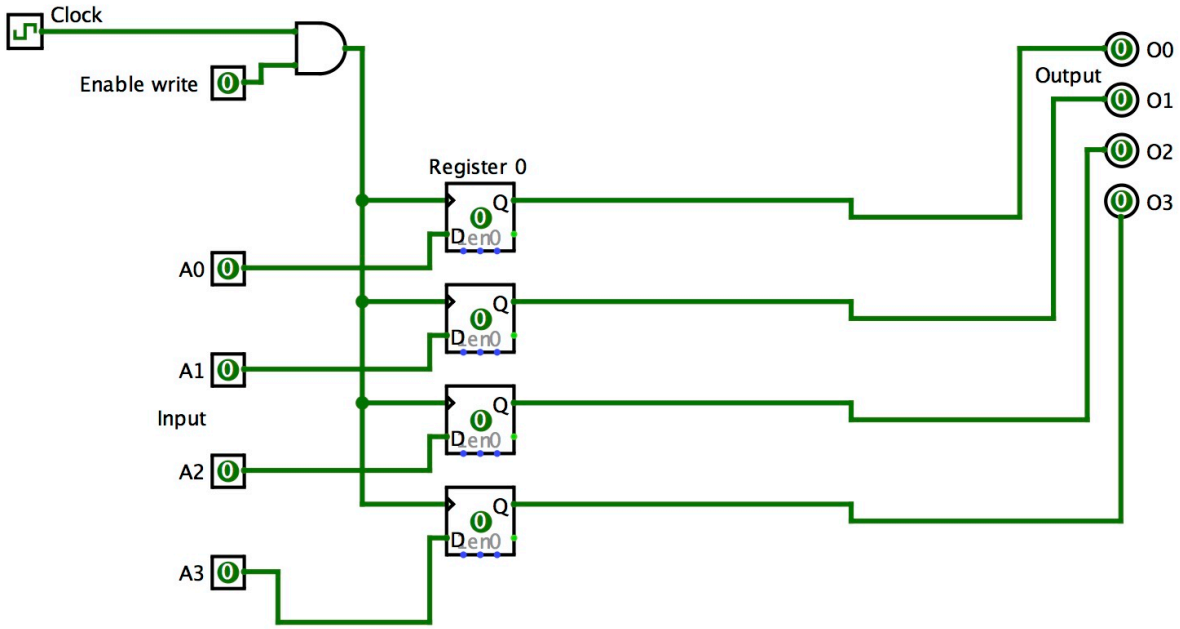
We have seen that registers are the very fast storage cells inside the CPU that are used for storing single-word temporary results. The *register file* collects all the registers into a single circuit, and it enables the

control unit to select one of the registers for reading and another one for writing at any one time. This simplifies the overall design: The register file has one *n*-bit data input (for writing an *n*-bit word into the selected register), one input for selecting the register to read from and another one for the register to write to, and one *n*-bit output that always outputs the word that is stored in the register that's been selected for reading.

In order to construct a register file, let's first construct a single *n*-bit register. The idea is simple: store each of the *n* bits in its own D-flip-flop. For our examples here we will assume that words have 4 bits, but you will see that the construction is very regular and can be extended easily to any number of bits.

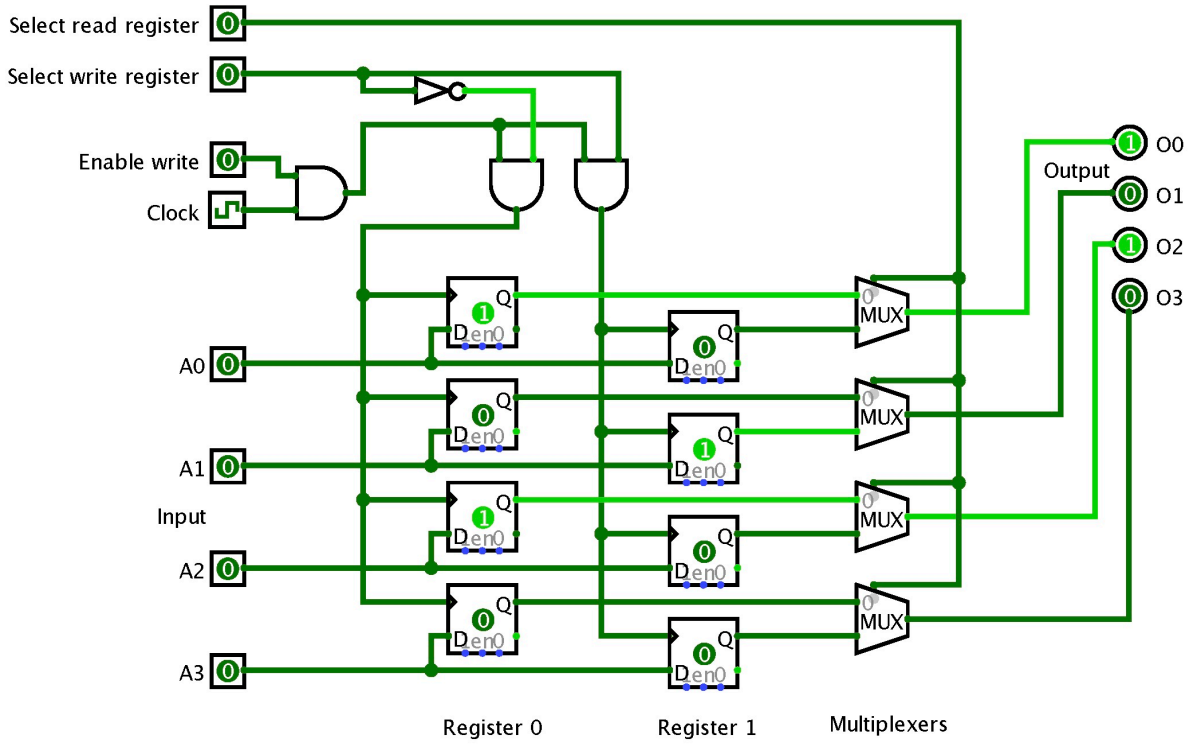


Above, you can see four D-flip-flops connected to four inputs and four outputs. Every clock cycle transfers the current input into the register, which stores it and outputs it. However, there is only once central clock for the entire CPU (in order to keep everything synchronised). So that would mean that the currently selected register is always overwritten during each clock cycle. But what if we don't want to write a new value at every clock cycle? To avoid that, we can introduce a "write-signal", another input that is AND-ed with the clock signal. This is required so that the clock is only passed to the flip-flops when an operation actually needs to change the register contents:

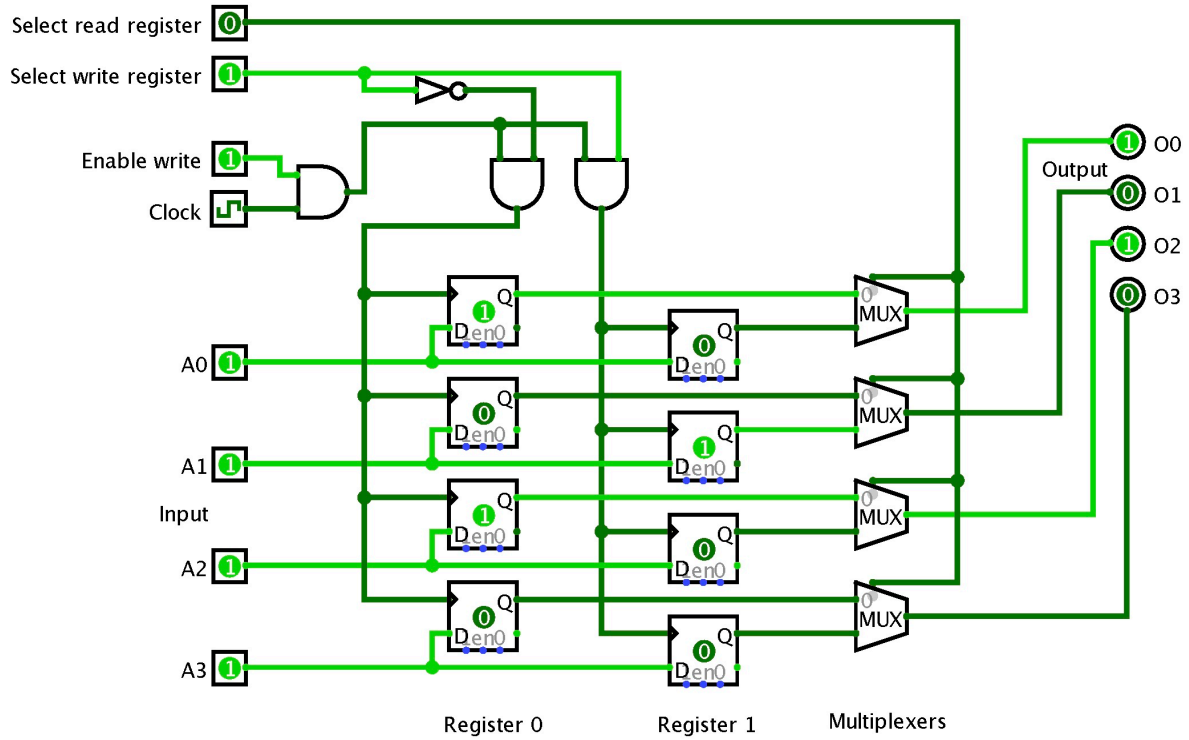


Now the "Enable write" signal is something the control unit can trigger whenever the current instruction requires writing to a register. The AND gate makes sure that only when "Enable write" is 1 *and* the clock signal is rising (see the discussion for D-flip-flops), the register contents actually change.

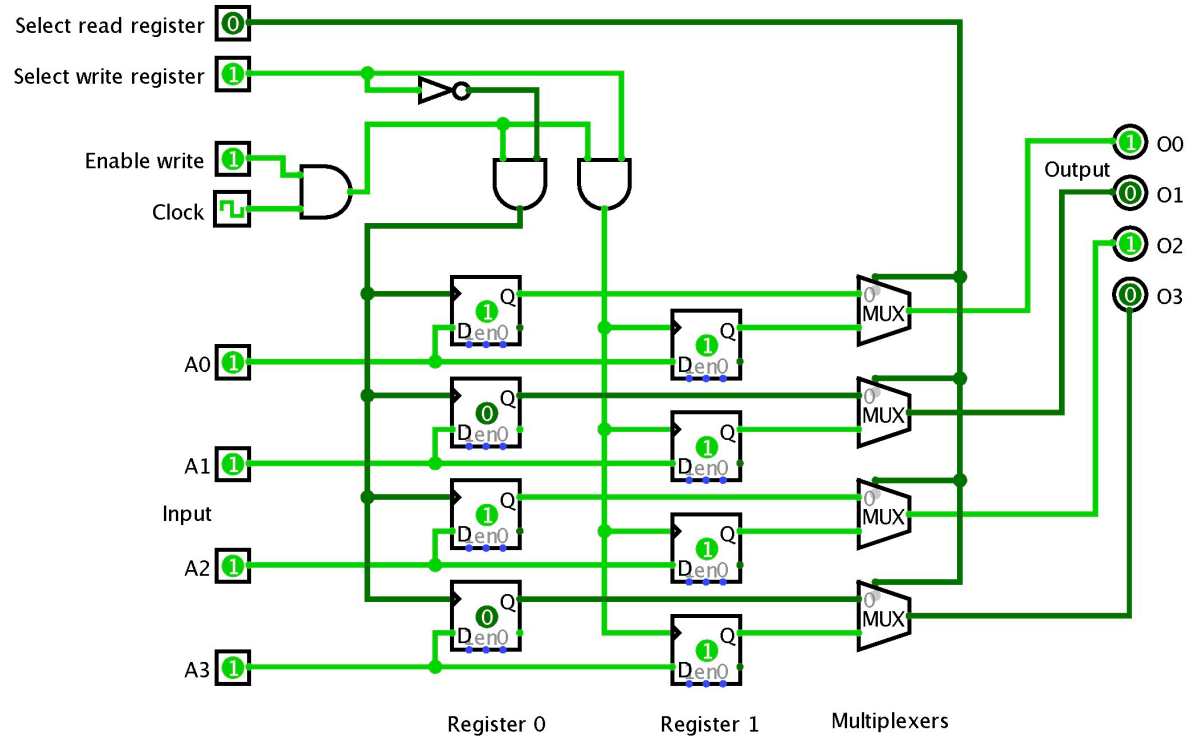
In order to extend this *n*-bit register into a register file, we must add four more flip-flops per register, as well as inputs that can select the registers to read from and to write to. Selecting the register to write to is easy, since we can use the same idea as for the "Enable write" signal above. We simply pass the clock signal into only the D-flip-flops of the selected register. Selecting the register to read from is also straightforward, since we can just pass all the individual register outputs through a multiplexer. Here is the complete register file with two registers:



The current state shows that Register 0 on the left stores the value 0101, while Register 1 stores 0010. The read selector is 0, which means that the output shows the current contents of Register 0. In order to now replace the value stored in Register 1 by, say, 1111, we need to set the inputs to 1111, set the write selector to 1, and enable writing:

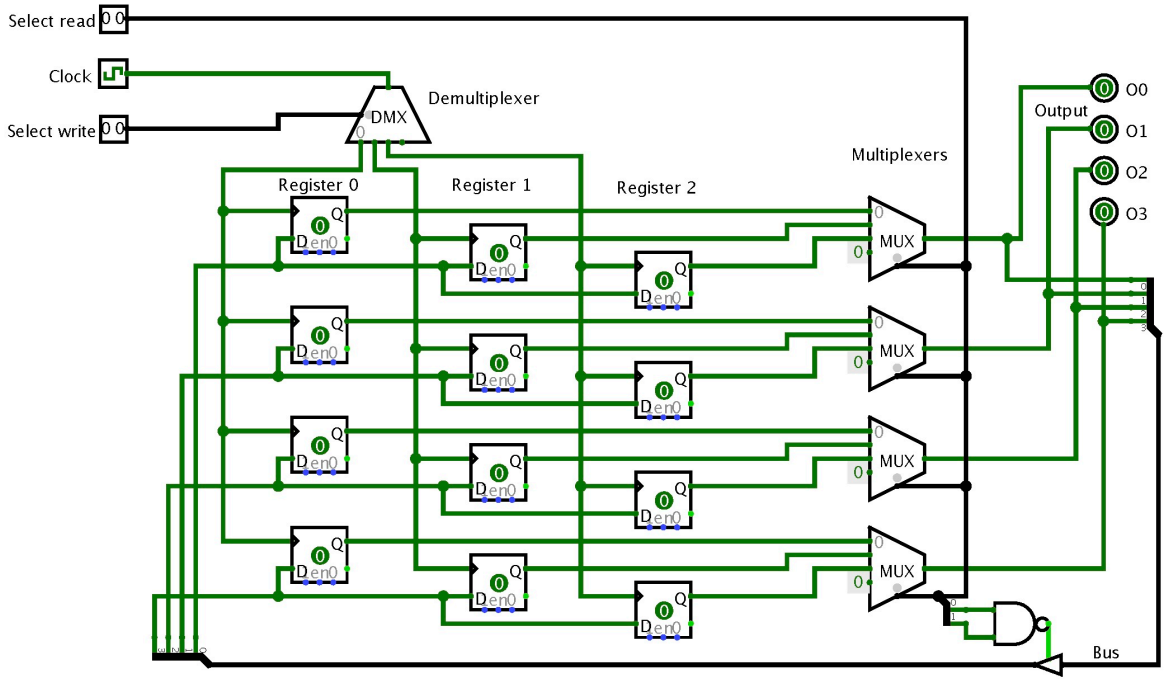


Nothing has changed yet, but as soon as we toggle the clock to 1, the contents of Register 1 will be overwritten:



In reality, the register file of course doesn't have inputs and outputs that we can poke with a simulation

tool. Both inputs and outputs are connected to the *data bus*, which also connects the register file to other components such as the memory and the ALU, as in the example below.



This setup also allows us to copy the contents of one register into another one: if we set the read and write selectors to different registers, the data stored in the read register is put on the bus, and the write register stores the data from the bus. Remember that several operations in the MARIE instruction set require this behaviour (e.g. transferring the PC into MAR, or transferring the MBR into AC).

A real register file doesn't actually use individual D flip-flops but is implemented using more efficient technology (similar to very fast RAM).

The Control Unit

The only piece of the Van Neumann architecture that we haven't discussed yet is the Control Unit (CU). Its main task is to perform the *fetch-decode-execute* cycle, by enabling control signals in the correct order at the right time. Recall from the module From Instructions to Circuits that each instruction can be broken down into a sequence of steps in RTL (register transfer language), and each of those steps can be represented by the control signals that are active at that time.

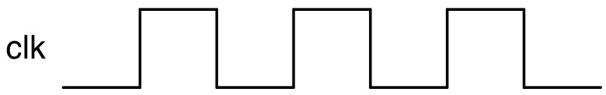
We know which control signals we need, and we know what should happen when they get activated. For example, the P5, P4 and P3 signals introduced previously enable a register for writing. So we just need to connect those wires to the "Select write" input of the register file. The A2, A1, A0 signals will be connected to the ALU opcode, and so on for all the other control signals.

Can you guess what is missing? We mentioned above that the CU activates the control signals *in the correct order* and *at the right time.* So we need a notion of *time.*

Clocks and cycle counters

The usual measure of time in a CPU is a *clock cycle*, as determined by the *system clock*, which generates an oscillating signal. This means that a special "wire" in the CPU continuously changes from 0 to 1 and back to 0 with a fixed frequency. For example, the computer that this is being written on makes 1.6 billion transitions from 0 to 1 and back to 0 per second. We say that it has a *clock frequency* of 1.6 GHz (Giga-Hertz).

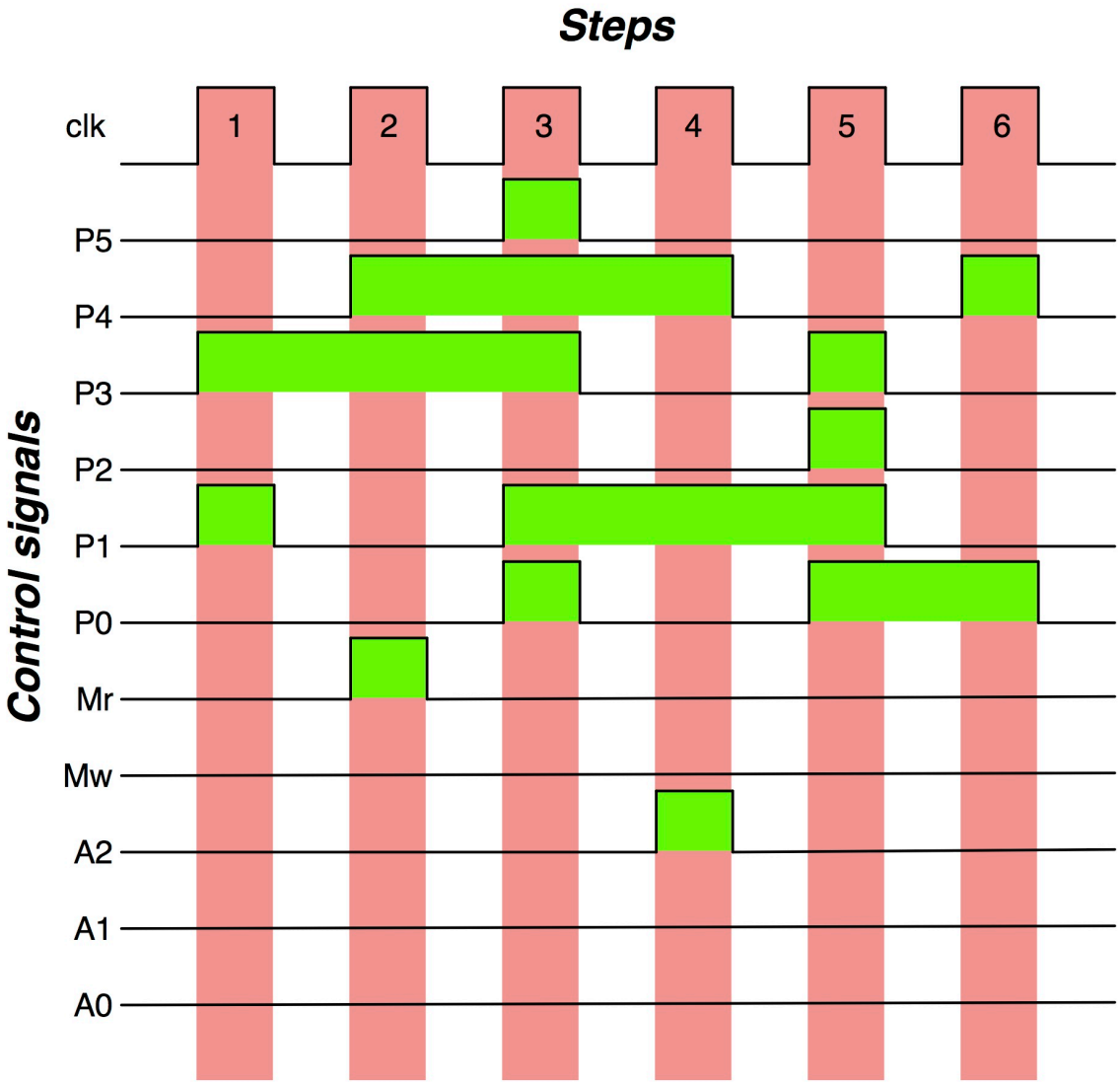
We can visualise a clock signal in a *timing diagram* like this:



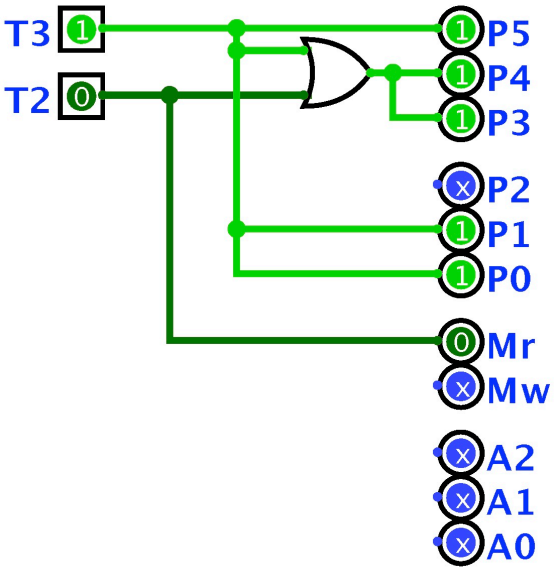
The *x*-axis represents time, and the *y*-axis represents whether the signal is active (1) or not (0) at a particular time. Given a system clock, we can now think about what the control unit needs to do in order to execute a given instruction. Let's take a simple instruction like `Jump X`. Its RTL and corresponding control signals look like this:

```
1. MAR ← PC                    P3    P1
2. MBR ← M[MAR]             P4 P3          Mr
3. IR ← MBR              P5 P4 P3    P1 P0
4. PC ← PC+1                P4       P1          A2
5. MAR ← X                     P3 P2 P1 P0
6. PC ← MAR                 P4          P0
```

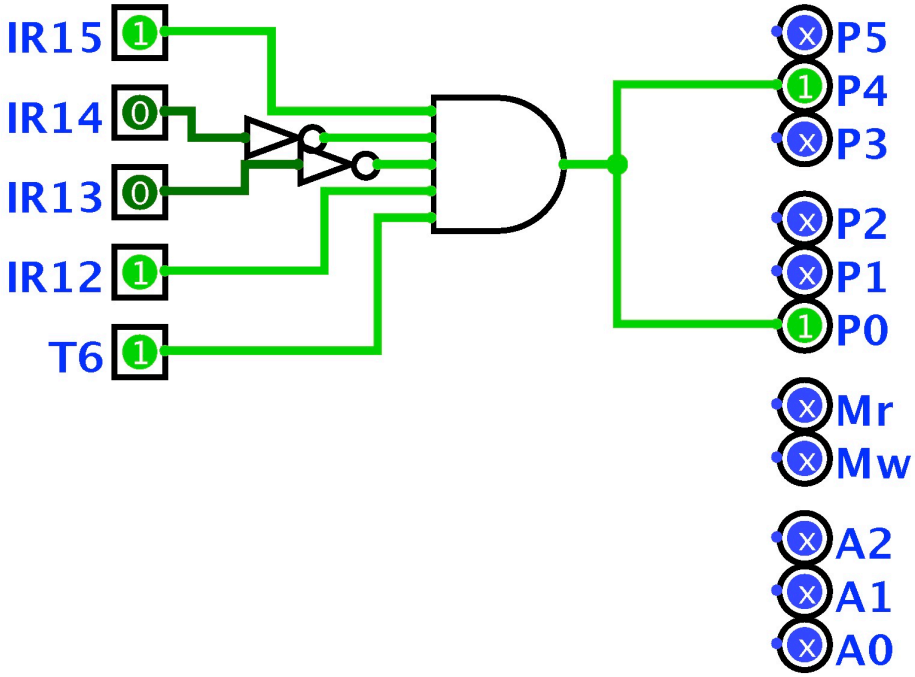
We can transform this into a timing diagram where each clock cycle corresponds to one of the six steps, and we show the activity of the control signals at each step:



This timing diagram exactly shows *what* the Control Unit needs to do at each clock cycle, for this particular instruction. For instance, during cycle number 2, it needs to activate the P4, P3 and Mr control signals. In cycle 3 it's P5, P4, P3, P1 and P0. Let's now assume that we had additional signals that tell us which cycle we're currently in - let's call them T1, T2, T3 and so on (T for timing). Then we can implement a very simple piece of logic that activates the correct control signals. Here's the circuit just for cycle 2 and 3:



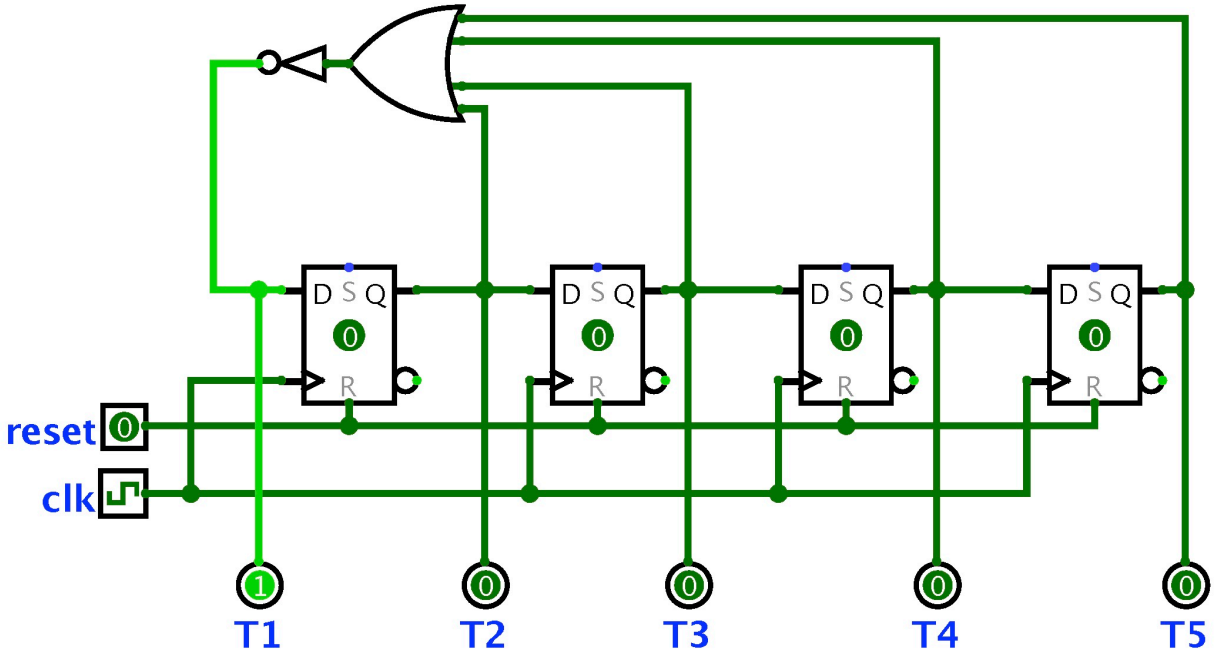
For the first four cycles, the control signals don't depend on the instruction (the first four cycles *fetch* the instruction, after all). So let's have a look at cycle 6 of the above example. If we're in cycle 6, *and* the instruction is `Jump X`, the CU needs to activate P4 and P0. So how does the CU find out that the instruction is `Jump X`? It simply looks into the four highest bits of the instruction register `IR`, which we represent as IR15, IR14, IR13 and IR12. The `Jump` instruction has opcode 1001, and we can use a decoder circuit to detect that:



Now if the `IR` contains a `Jump` instruction (as shown in the diagram above), *and* the timing signal T6 is active, then P4 and P0 get activated.

The only missing piece now is the circuit that *generates* the timing signals T1, T2, T3 etc. This type of circuit is called a *cycle counter.* We are not going to explain it in detail, all you need to understand is that it has *n* outputs T1 to T*n*, and at each clock cycle, a different output gets activated. After T*n* has been activated, the cycle starts again at T1.

The following circuit diagram shows a cycle counter with five timing signals.

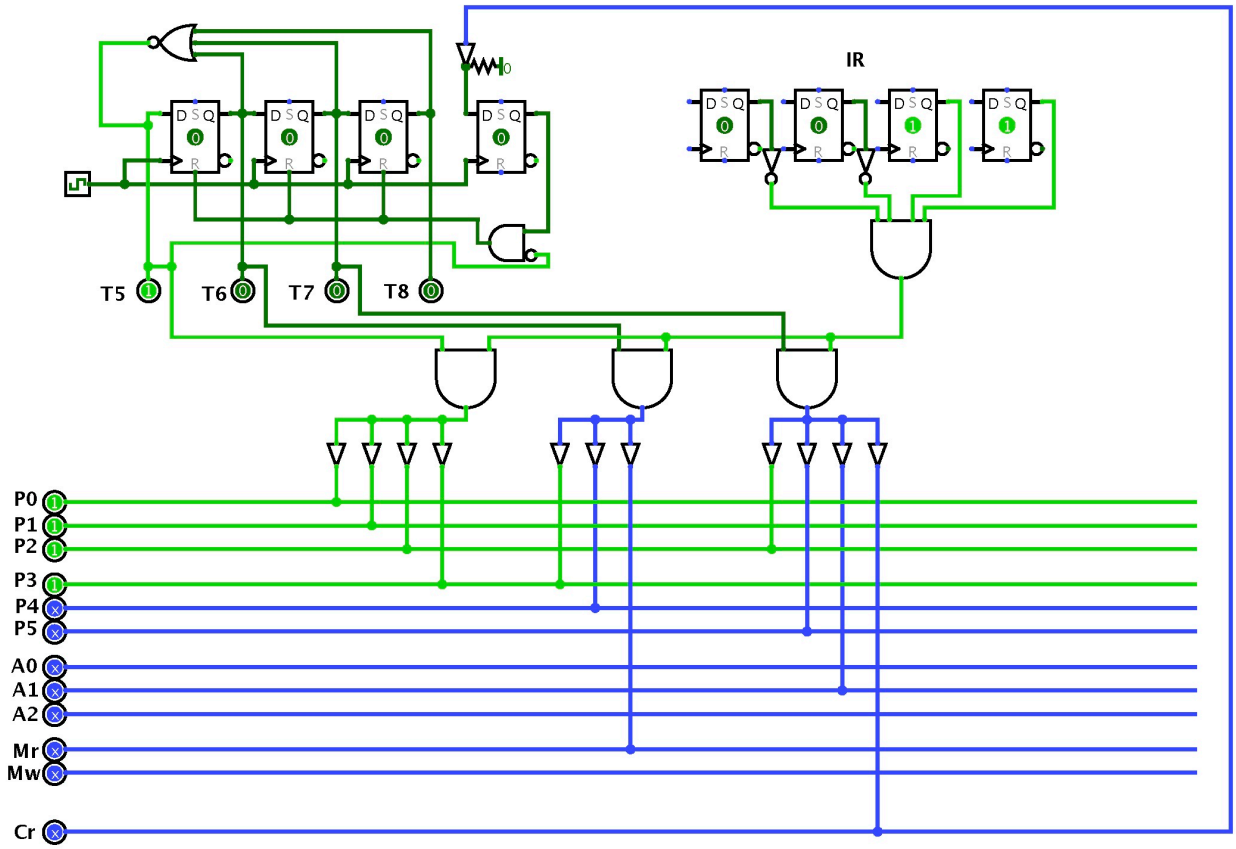


For the MARIE instruction set, we will need a cycle counter with nine timing signals, since all instructions use at most 9 RTL steps. For instructions that use fewer than 9 steps (like `Jump X` discussed above), we simply have to trigger the reset signal of the cycle counter after the final step, which will re-activate T1 and therefore start the next fetch cycle.

Putting it all together

We could now combine all the circuits we have seen into a complete CPU. At the core of the Control Unit, the cycle counter triggers the fetch-decode-execute phases by activating the T1 - T9 timing signals. For each RTL step of each instruction, we have a little decoder circuit that enables the correct control signals for that step. The control signals are connected to the register file (selecting the register for reading and writing), to the ALU (to select the operation to perform), and to the memory (to enable reading from or writing to the memory location stored in MAR).

Here is the circuit diagram for steps 5-7 of an `Add X` instruction.



The top right part of the circuit is a cycle counter. For simplicity, counting here starts at T5, but of course a real one would start at T1 and go all the way to T9.

Note how the IR register (top right corner) is currently set to 0011, which corresponds to an Add instruction. The NOT gates and the AND gate together *decode* the instruction, so that the output of the AND gate is 1 exactly if the instruction is in fact an Add. In that case, for timing step 5, we need to activate P3, P2, P1 and P0. So if the timing signal T5 is 1 *and* the instruction is an Add, we activate those control signals (that's the left-most AND gate above the control wires). Similarly, for timing step T6, signals P4, P3 and Mr are activated. For the final step (T7), we not only activate the control signals (P2, P5 and A1), but also the reset signal Cr for the cycle counter, so that it goes back to the next fetch cycle.

This type of Control Unit is called **hard-wired control**, because each step of each instruction is implemented directly using logic gates. Hard-wired control has the big advantage that is results in a very fast circuit. However, it has three major disadvantages: it is complicated to design, the design is difficult to extend if we want to add new instructions, and it is impossible to fix an error in the design when the CPU has already been produced and shipped.

Designing a hardwired control circuit for a simple architecture like MARIE with only 16 instructions is already quite complicated (you can try it out!). Now imagine the control logic required for a CPU such as an Intel Core i7, which has hundreds of instructions!

Microprogrammed control

Implementing a hard-wired control logic is clearly unrealistic for large CPU projects. In order to deal with the large complexity of a realistic instruction set, we can use the fact that many instructions have *similar steps in their RTL definitions*.

The solution is therefore to break up instructions into **microoperations**, one for each unique RTL step. For example, many MARIE instructions contain the step `MBR ← M[MAR]`, which reads a word from memory (step 6 for all instructions that read from memory, and step 8 in `AddI`, `LoadI` and `StoreI`). Now we just implement hard-wired control logic for each microoperation, and add a small memory to the CPU that contains the list of microoperations for each machine code instruction. After fetching an instruction from memory, the control logic then executes the small **microprogram** for the instruction. We could say that the CPU contains a hardware-based *interpreter* for the machine code that is based on these microprograms.

The big advantage of *microprogrammed control* is that we need to implement far fewer hard-wired control circuits, and each of them is much simpler than the circuits required for a large and complex instruction set. And the memory inside the CPU that stores the microprograms can be updated! So if there is a bug in one of the microprograms, the CPU vendor can fix it *even after the CPU has been shipped* to millions of customers, via a simple software update. Modern PC processors (from the Intel x86 family) are micro-programmable, and Intel has indeed delivered software updates for its CPUs.

4.3 Memory

The module on CPU basics and History already briefly explained the concept of memory: it's a sequence of *locations*, each of which has an *address* (consecutive integers, usually starting from 0), and each location can store one data value of a fixed *width* (i.e., a fixed number of bits). The CPU can read the value currently stored at a location, and overwrite it with a different value.

Addressing memory locations

An address is simply an unsigned integer that references one unique memory location. Addresses are usually consecutive numbers starting at 0 and ranging up to the number of distinct memory locations (minus one). In most architectures, **one memory location stores one byte**. Consequentially, each location, each byte, needs its own address. This is called **byte-addressable memory.**

In some architectures, such as MARIE, **one memory location stores one word**. Each address therefore references a whole word, and we call this **word-addressable memory**. Recall that words in MARIE are 16 bits (or 2 bytes) wide, but other systems may use word sizes such as 32 bits or 64 bits.

Let's now compute the *number of different addresses* we need in order to address each location in a certain size memory. For *n* locations, we clearly need *n* different, consecutive integer numbers starting from 0. For example, in order to address 1MB of memory, which is 1,000,000 bytes, in byte-addressing mode, we need 1,000,000 different addresses, from 0 to 999,999.

The next question is how many *bits* we need to *represent* any of these addresses as a binary number. For our example of 1MB, the highest address is 999,999, or written in binary: 1111 0100 0010 0011 1111. We therefore need 20 bits to represent any address between 0 and 999,999. More generally, to represent *n* different addresses, we need $\lceil \log_2(n) \rceil$ bits.

In most cases we're dealing with memory whose size is a power of 2, e.g. 2 gibibyte is 2×2^{30} bytes. In a byte-addressable architecture, we therefore need $2 \times 2^{30} = 2^{31}$ different addresses, ranging from $0 \ldots 2^{31} - 1$. Using powers of two makes it really easy to compute the number of bits needed: $\log_2 2^{31} = 31$.

Let's now assume a word-addressable architecture where the word size is 16 bits. We still have 2×2^{30} bytes of memory, but each memory location now holds *two* bytes (16 bits). So the number of memory locations is only \frac{2\times 2^{30}}{2} = 2^{30}[\latex]. Accordingly, we only need 30 bits to address each location.

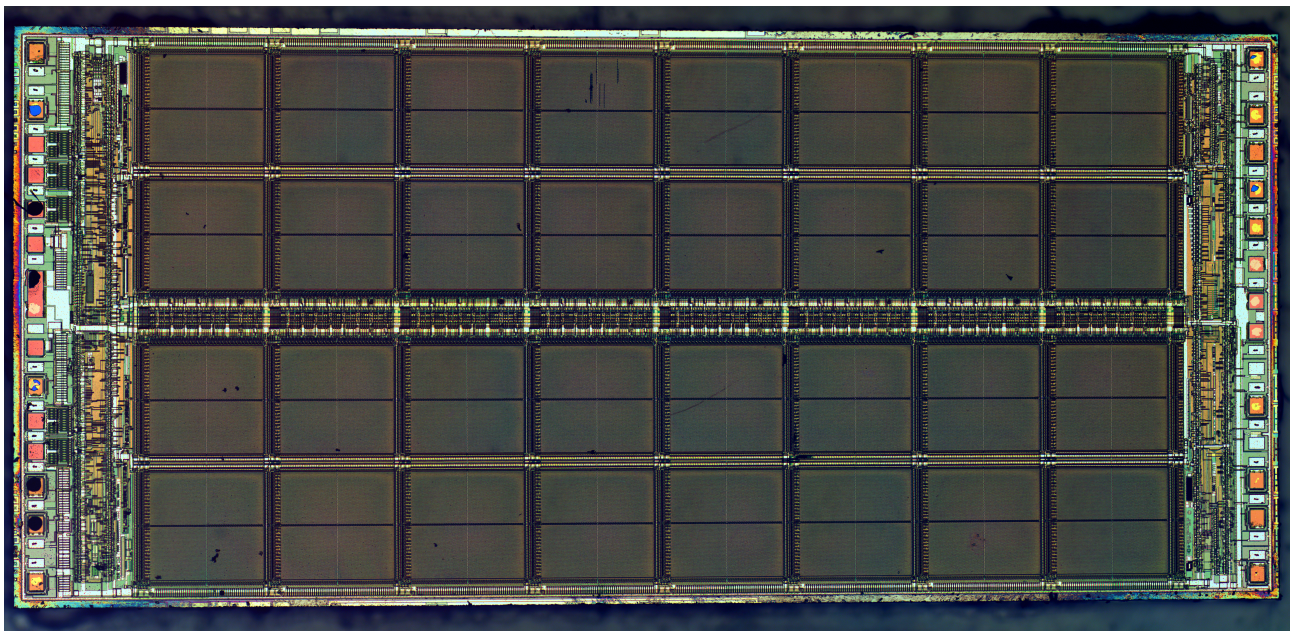
So to summarise, **in order to address 2^n memory locations, we always need *n* bits for the addresses**. In byte-addressed systems, the number of locations is the number of bytes. In word-addressed systems, it's the number of words, which is half the number of bytes if the word size is 16, a quarter if the word size is 32, or an eighth if the word size is 64.

No. of bytes	No. of addresses	No. of bits needed	
Byte-addressable	2^n	2^n	*n*
16-bit word addressable	2^n	$2^n/2=2^{n-1}$	*n-1*

No. of bytes	No. of addresses	No. of bits needed	
32-bit word addressable	2^n	$2^n/4=2^{n-2}$	n-2

RAM

RAM stands for Random Access Memory, which emphasises that the CPU can access any memory location in RAM (i.e., read from them or write into them) in basically the same amount of time. This is different from, e.g., a hard disk, where it may be very fast to read the data that is currently under the read/write head, in a sequential fashion, but it can be very slow to read arbitrary pieces of data that are stored in different physical locations on the disk.

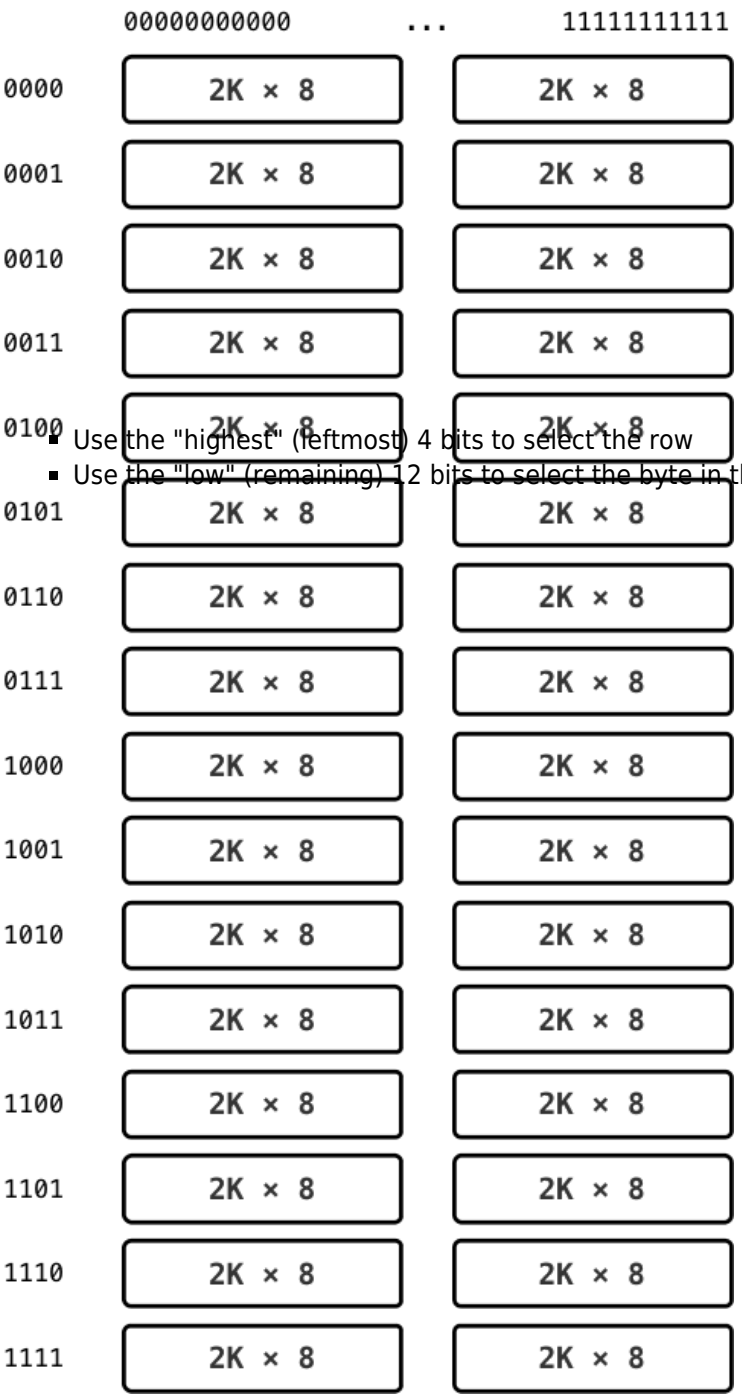


Micron MT4C1024. 1 mebibit (220 bit) dynamic ram. By Zeptobars, CC-BY 3.0 (https://zeptobars.com/en)

This image shows a RAM module that has been treated with concentrated acid, so that we can take a look inside. What we can see immediately is that a RAM module has a lot of structure. It seems to consist of a top and a bottom section, with each containing 16 square blocks in two rows, and each block in turn seems to have two parts. This structure is no coincidence.

Memory organisation

RAM modules are made up of multiple chips. Each chip has a fixed size $L \times W$, where L is the number of locations, and W is the number of bits per location. For example, 2K × 8 means 2×2^{10} locations of 8 bits each. RAM chips are combined in rows and columns to construct larger modules. For example, we can use thirty-two 2K × 8 chips to build a 64K × 8 memory module made up of 16 rows and two chips in each row:



A RAM module organised into rows and columns

Let's now assume that we want to use this RAM in a computer that uses byte-addressable memory. Since we have 64 × 2^{10} = 2^{16} locations, we will need addresses that are 16 bits long.

So how does the hardware know which RAM chip it should use when a given address is requested? We split up the addresses into two parts:

- Use the "highest" (leftmost) 4 bits to select the row
- Use the "low" (remaining) 12 bits to select the byte in the row

This is called *memory interleaving* (and in particular, high-order interleaving if the highest bits are used to select the row, and low-order interleaving if the lowest bits are used). In modern architectures, interleaving can significantly improve memory performance, because it can allow the CPU to address several different memory chips at the same time (e.g. one for reading and another one for writing).

In a word-addressable architecture, we can of course use the same technique. Assuming that our example 64K × 8 RAM module is accessed by an architecture that uses 16 bit words per location, we only need 15 bits for the addresses (since each address now represents twice the amount of memory!), and the leftmost 4 bits of an address still indicate the row while the remaining 11 bits indicate the *word* inside the row.

4.4 Input/Output devices

Computers are completely useless without some form of input and output. We need input devices to get data and programs into the machine, and output devices to communicate the results of the computation back to us. In this module, we'll first talk about the different types of input and output devices, and then discuss how these devices communicate with the CPU.

Early I/O

Teletype Corporation ASR-33 teleprinter. Image by ArnoldReinhold (via Wikimedia Commons). CC BY-SA 3.0.

Five hole and eight hole perforated paper tape as used in telegraph and computer applications. By TedColes (via Wikimedia Commons), Public Domain.

In the earliest computers, input often consisted of *hard-wiring* the programs and data (if you're interested what that looked like, you can read more about it in the Wikipedia article on Core Rope Memory (https://en.wikipedia.org/wiki/Core_rope_memory)), or of simple switches, and soon punched paper tape or cards. In fact, punched tape and cards predate digital computers by more than two centuries! They were used to control automatic looms (the first records seem to be from around 1725, and the first fully automatic machine was the *Jaquard Loom* from 1801), and later to tabulate data such as census records.

teleprinters were then adapted as output devices for early computers as well. Their typewriter keyboards could also be used to create punched paper tape for input.

Modern I/O devices

A modern computer (and this includes things like smartphones and washing machines) features many different I/O devices. Things that come to mind as typical input devices include keyboard, mouse, touch pad, touch screen, voice control, gestures, cameras, fingerprint sensors, iris scanners, accelerometers, barometers, or GPS. Output devices are things like screens, printers, audio, and robotic actuators. But other components such as external storage (hard disks) and network devices (WiFi, 4G, Bluetooth) are also classified as I/O.

Today, most I/O devices communicate with the CPU via standardised **interfaces**. The most common ones are USB (for which you probably know quite a few applications), SATA (for connecting hard disks), DisplayPort and HDMI (for displays), or PCI Express (for expansion cards). What characterises a standard

interface is

- a standardised set of connectors (i.e., the physical dimensions of plugs and sockets)
- a standardised electrical behaviour (defining the "meaning" of the wires in the plugs)
- standardised software protocols (so that you can use the same device with any computer)

I/O devices can be connected to the CPU *internally* (i.e., in the same case and possibly on the same printed circuit board), or *externally* (e.g. using a plug and cable). In either case, the device would use a standard interface (for example, the popular Raspberry Pi "single-board computer" ships with an Ethernet network interface that is soldered onto the same circuit board and connected via an internal USB interface, without using any cables and plugs).

I/O and the CPU

Let's now look at how I/O devices can communicate with the CPU. Clearly, the communication needs to be bi-directional: the CPU can send data to a device (e.g. pixels to the screen), and a device can send data to the CPU (e.g. the key that the user just pressed). In fact, almost all devices are *both* input *and* output devices. For instance, we may need to set parameters (such as sensitivity) of a touch screen, which means writing to an input device. Or we may want to check the status of a printer, which means reading from an output device. And, obviously, devices such as network interfaces or mass storage need to do both input and output anyway.

The next step is to look at how data are transferred between the CPU and the I/O devices. Conceptually, we can think of each I/O device as having its own set of **registers**, i.e., small pieces of memory that hold the data that needs to be transferred to and from the CPU. For example, a keyboard could have a register that holds an integer value that represents the key that's currently pressed. A network interface could have a register that holds the next byte to be transmitted, or the last byte that's been received.

The I/O device is connected to the CPU via the bus, so that data can flow between the I/O registers (located inside the I/O device) and the CPU registers. However, the data cannot be simply transmitted between I/O device and CPU all the time! The currently running program should be in control of **when to read from and write to** an I/O device. This is realised through one of two mechanisms, depending on the concrete architecture.

In the **memory-mapped I/O architecture**, the I/O registers are "mapped" into the "address space" of the CPU. In the **instruction-based I/O architecture**, the CPU has special instructions to read from or write to particular I/O devices.

Let's explain memory-mapped I/O with an example. In the MARIE architecture, we access the memory using `Load X` and `Store X` instructions, where X is a 12-bit address. That means we can use the addresses from `0x000` to `0xFFF`, which gives us access to 4096 different locations. Memory-mapped I/O would now use some of these addresses to access I/O devices instead of real memory. For example, we could use the highest address `0xFFF` to read from the keyboard, and the one below (`0xFFE`) to write to the screen. The instruction `Load 0xFFF` would then load the value that identifies the currently pressed key into the AC register in the CPU, while `Store 0XFFE` would print the current value of AC to the screen. This is convenient: we can do I/O using the already existing instructions, we just need to modify the hardware of the Control Unit a bit to recognise these special addresses when it decodes a `Load` or `Store` instruction.

A disadvantage of memory-mapped I/O is that each I/O device now takes up *address space*, i.e., the special I/O addresses cannot be used to access RAM any longer, so the total amount of RAM that can be

used by a program is reduced. Depending on how it is implemented in hardware, and how big the address space is to begin with, this can be a severe limitation.

The alternative - instruction-based I/O - should now be clear. Instead of re-using `Load` and `Store`, the CPU has additional special-purpose instructions just to perform I/O. In MARIE, the `Input` and `Output` instructions play that role. But in more realistic architectures, we can't have special instructions for each potential I/O device we might want to plug into our computer! So instead of (very specific) instructions like `Input` and `Output`, we may have instructions like `Read X` and `Write X`, where X identifies the device that we want to read from or write to. In effect, `Read` and `Write` play the same role as `Load` and `Store`, but for I/O! The X in `Read X` and `Write X` identifies a concrete register in a particular I/O device, so we can regard it as an *I/O address* (as opposed to a memory address in `Load X` and `Store X`). Sometimes I/O addresses are also called *I/O ports*, and instruction-based I/O is then referred to as *port-mapped I/O*.

An advantage of instruction-based I/O is that, since we typically require far fewer I/O addresses than memory addresses, we can use a reduced address width for I/O. E.g., this approach is used in the x86 processor family, where memory addresses are 32 bits wide, but only 16 bits are used for I/O ports. This simplifies the hardware implementation of the logic that decodes I/O port addresses.

When to do I/O

Now we have seen *how* to communicate with I/O devices - using special addresses (memory-mapped) or special instructions (instruction-based). For output operations, that is all we need to do, because the program determines when to produce output. For input operations, things are slightly different: how does the program get notified that "new" input is available? For instance, how does the program find out that the user has pressed a key, or that a new message has arrived on the network interface? So the big question is *when* to perform an input operation.

This questions is further complicated by the fact that I/O devices are *much, much slower* than the CPU. Think about a keyboard: the very fastest typists can do something around 17 characters per second. But a modern CPU can execute *billions* of instructions per second. Even a fast network interface is nowhere close to the speed of the system bus, i.e., data cannot be fed into the CPU at the same speed that the CPU can operate.

We therefore need a mechanism for finding out whether new data is available from an I/O device. There are two main approaches to this, *programmed* I/O and *interrupt-based* I/O.

Programmed I/O

In programmed I/O, the program itself is responsible for checking periodically whether new data has arrived from an I/O device. This means that the programmer must be careful to insert checks into the program that will be executed at regular intervals. The following pseudocode illustrates this:

```
while (true) {
  if (IORegister1.canRead()) {
    processRegister1();
  } else if (IORegister2.canRead()) {
```

```
    processRegister2();
  } else if (IORegister3.canRead()) {
    processRegister3();
  } else {
    // do something productive
    doSomeComputation();
  }
}
```

The programmer wrote an infinite loop (`while (true)`), which in each iteration checks whether one of three I/O registers has new data to read, and if it does, it calls a subroutine to process the new data. If there is no new data, it calls a subroutine that performs some other computation that the program needs to do. We also call this ***polling I/O***, because the program "polls" the I/O devices.

The key characteristic of programmed I/O is that it is the responsibility of the programmer to get the timing right. If the program checks I/O too often, it wastes time doing the checks that could be spent on actually computing something useful. If it doesn't check often enough (e.g. because `doSomeComputation` takes too long to finish), the system will feel "laggy", e.g. it will take a while for a character to appear on screen after the user pressed the key.

The main advantage of programmed I/O is that the programmer has full control, e.g., they can prioritise certain I/O devices over others (by querying them more often). However, this comes at the cost of additional complexity for the programmer, who now has to worry about getting these priorities right, and needs to structure their code around the main I/O loop. We say that the programs become *I/O driven*. Another big disadvantage is that the CPU is constantly busy: it must run at 100% capacity all the time in order to execute the main loop. Modern CPUs have very sophisticated mechanisms for saving energy when they are not currently used. With programmed I/O, your smartphone's battery would probably only last minutes rather than hours!

Interrupts

The alternative to programmed I/O, and the approach implemented in most modern CPUs, is to make the hardware *notify* the CPU when new data is available. The CPU then interrupts the program it is currently executing and jumps into a special subroutine to process the I/O request. Afterwards, it continues with the normal program where it left off. This is called *interrupt-based I/O.*

With interrupt-based I/O, the programmer does not have to be aware of I/O at all. The main program will be interrupted when necessary, the CPU jumps into an *interrupt handler* subroutine, and when the handler is finished, continues executing the main program. We have shifted the responsibility for handling I/O from the software into the hardware.

Interrupt signals and handlers

Before we look at interrupt handlers in more detail, we need to understand how interrupts are implemented inside the CPU.

The CPU has control signals that I/O devices can use to signal that they want to interrupt the CPU. When an I/O devices activates its interrupt control signal, this sets a bit in a special CPU register. The CPU checks the status of this register in each iteration of the fetch-decode-execute cycle, before the fetch phase. If an interrupt bit is set, it processes the interrupt. Otherwise, it continues with the normal fetch operation.

We can express this additional step before the *fetch* in terms of RTL. Since it happens before *fetch,* we will call it steps 0.1, 0.2 etc.

```
0. If InterruptBit is set:
   0.1 Clear InterruptBit
   0.2 MAR ← SavePC
   0.3 MBR ← PC
   0.4 M[MAR] ← MBR
   0.5 PC ← InterruptHandler
1. MAR ← PC
2. MBR ← M[MAR]
3. IR ← MBR
4. PC ← PC+1
```

Let's explain these steps one by one. If the interrupt bit is not set, we simply continue with the normal first step of the *fetch* phase. Otherwise, we first reset the interrupt bit (step 0.1) so that the device can signal a new interrupt in the next cycle. The next three steps (0.2-0.4) store the current PC into a special memory location SavePC. This is required so that the CPU can jump back to the current point in the program after executing the interrupt handler (this is very similar to what the JnS instruction does - go back and check its RTL!). Finally, we set the PC to the fixed address `InterruptHandler` (step 0.5) and then continue with the normal *fetch*, which will now fetch the first instruction of the interrupt handler.

The interrupt handler is a piece of code written by a programmer, and would typically be part of the operating system. It has a few special requirements:

- It should be reasonably quick to execute (otherwise it will slow down the rest of the system).
- It must jump back to the address stored in SavePC when it has finished, just like a subroutine. Which instruction would it use for that jump? Reveal [1]
- It must leave the CPU in exactly the same state as when it was called.

Let's consider the third requirement in more detail. The main user program will get interrupted at an arbitrary point in time. For example, let's assume that this is part of the program that's currently running when an interrupt happens:

```
...
Load X
Add Y
Store Z
...
```

Let's assume the interrupt is detected before the *fetch* phase begins for the Add instruction. At this point, the CPU already loaded the value from address X into the AC register. Now our interrupt handler is executed. In order to do anything useful, it will probably have to use the AC register, which will overwrite its current value. That's bad news for our program: the Add instruction will expect to find the value stored at X in the AC! We therefore have to make sure that the value of AC after executing the interrupt handler is exactly the same as before.

This can be achieved by either having a second register file, called the *shadow registers*, which the CPU uses whenever it's executing interrupt handler code. At the end of an interrupt handler, it simply switches back to the primary register file, which is unchanged. Alternatively, the interrupt handler can save all registers into memory before using them, and then restoring them before it finishes (similar to how we used SavePC above).

Interrupt vectors

In the pseudocode example for programmed I/O above, the code tested different I/O devices and then *dispatched* into different subroutines depending on which device has new data to read. We can do the same with interrupts. Our simple example above just checked a single bit, but in a realistic scenario, there are multiple I/O devices that could signal interrupts at any given time. So in order to distinguish the interrupts from different devices, each device is assigned an *identification number*, and when raising an interrupt, it stores that number in a special register (rather than just a single bit). The interrupt handler can then jump into an *interrupt vector*, a list of addresses of device-specific handlers.

Here is a bit of MARIE pseudocode that illustrates how an interrupt handler with an interrupt vector could look like:

```
InterruptHandler,  Store SaveAC
                   Load IVec
                   Add   DeviceID
                   Store Dest
                   JumpI Dest

SaveAC,            HEX 0    / Temporary storage for AC
Dest,              HEX 0    / Destination of jump
IVec,              HEX C03

C03                HEX DF0  / Address of first handler
C04                HEX DFA  / Address of second handler
C05                HEX DB0  / Address of third handler

DFA                ... / do some work
                   Load SaveAC     / Restore AC register
                   JumpI SavePC    / Return to interrupted code
```

The first step is to save the AC register so it can be restored when the handler is finished. Then it loads the base address of the interrupt vector (`C03`) from `IVec`. It adds the device identifier to the base address, and then jumps to that computed address. So, for example, if device number 1 caused the interrupt, we would compute `C03+1=C04`, then do an indirect jump to that address, which means jumping to DFA. The code stored at address DFA would now handle the I/O for device number 1. It would first do some device-specific work (e.g. handling a key press), and then restore the AC register by loading it from `SaveAC`, and finally jump back to the main program (`JumpI SavePC`).

In all these cases we are simplifying the system a bit by ignoring the fact that another interrupt could happen while the interrupt handler is currently being executed. Typically, the system would be designed so that further interrupts are disabled during this time.

Interrupts in real hardware

The original 8086-based PCs had 15 interrupt request (IRQ) signals. Each piece of hardware needed to be configured to use a different IRQ, e.g. by setting a jumper or a little switch on the actual hardware expansion card. Since some of the 15 IRQs were already used by internal devices, only very few were available for external hardware. The user had to make sure that two devices were not set to the same IRQ, otherwise the devices (or even the whole system) would simply not work. Some devices were able to *share* IRQs in order to make more efficient use of this scarce resource.

Modern systems use *Advanced Programmable Interrupt Controllers (APICs),* which are often integrated directly into the CPUs. These allow for more IRQs (and therefore lead to fewer potential conflicts between devices), and they can be configured automatically in a *plug&play* fashion. Imagine how inconvenient it would be if you had to set a little hardware switch to select a free IRQ every time you want to plug in a USB drive!

APICs also contain high-resolution *timers*, which are another really important source of interrupts. A timer simply creates a *software interrupt* at regular intervals (i.e., the interrupt is not triggered by an actual I/O device). This enables the CPU to execute a special subroutine of the operating system every few milliseconds, e.g. to switch between different programs, or to update the time displayed on screen. We will come back to this feature in the module on Operating Systems.

DMA

Interrupt-based I/O is the standard way of implementing I/O in modern computer architectures. However, on its own it still has a major disadvantage that causes a significant efficiency problem: All memory transfers need to be done by the CPU. For example, in order to read a file from disk into memory, the CPU needs to copy every single byte from the I/O device (the hard disk controller) into a register, and then from there into the memory. Another example is displaying something on screen: the CPU must transfer an image from memory into the graphics card, byte by byte. Since I/O devices and memory are both much slower than the CPU, these operations can become serious performance bottlenecks, where the CPU is essentially doing nothing but memory transfers most of the time.

In order to relieve the CPU of the task of just copying data to and from memory, modern architectures implement a feature called *Direct Memory Access* (DMA). The CPU can *delegate* memory transfer operations to a dedicated *controller* (a special piece of hardware). For example, the hard disk controller can copy a file directly from disk into RAM. The graphics card can fetch an image directly from RAM. This frees up the CPU to do more productive work.

The CPU and the DMA controllers however *share the data bus*. This means that only one can perform memory transfers at a time.

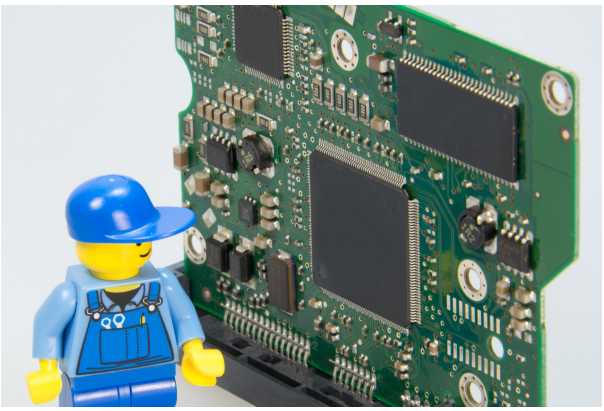
[↩1] Jumpl SavePC

5
Booting the system

Introduction

In order to understand what a computer needs to do to actually start and get to a status where it can be used, we first need to have a look at the different components within a computer. We use a typical PC to identify these components. All of them also exist in other computers (such as mobile phones, embedded devices, cloud servers, etc.), but they might look different or have different names.

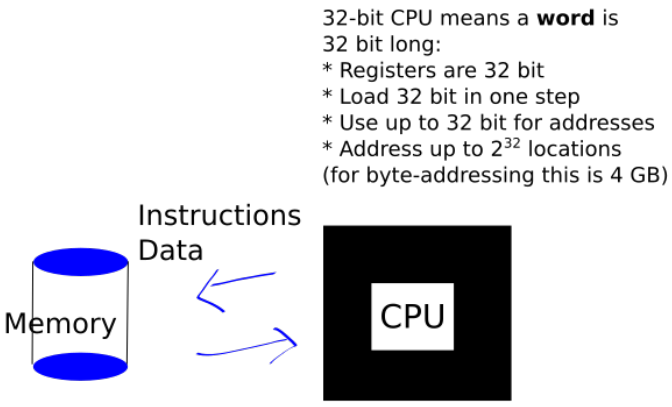
You will see a lot of terminology and there are quite a number of abbreviations. Most of them are considered "common knowledge" in IT or Computer Science. Therefore, you should at least know what they mean and which role the particular component plays in a computer.



Components of a computer

We start with the core of the computer, the Central Processing Unit CPU. If you look at the *motherboard* of a computer (the largest circuit board with lots of elements), the biggest piece of silicon encapsulated in black plastics should be the CPU or processor. If you buy a motherboard for a PC, usually the CPU is not installed yet, but there is a particular socket for the processor to be placed on.

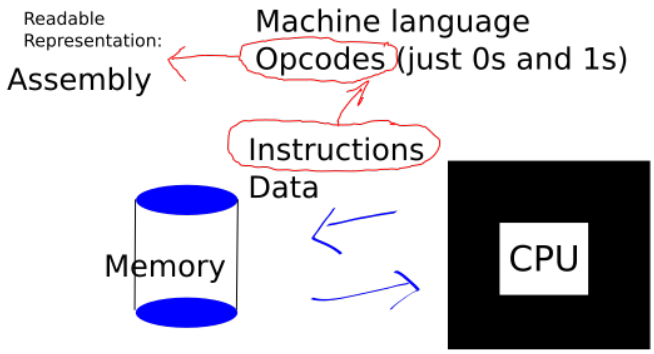
Thus, the CPU does the processing/computations. But in order to do this, it needs a program and data. Both are stored in memory. Thus, the main components are CPU and memory.



The picture shows some numbers for a 32-bit CPU.

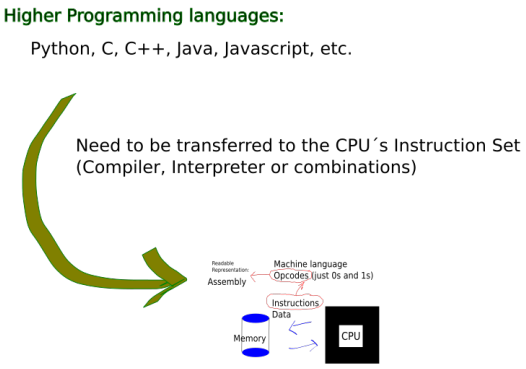
How many bytes can the toy CPU MARIE store with 12 bits for addressing and 16 bits per word?
Reveal [1]

As we have seen with MARIE, the actual CPU can only deal with instructions in the form of so-called *opcodes. Assembly* is the name of a programming language that directly translates into opcodes. In principle, Assembly provides a readable representation of the machine language for a particular processor. Note that assembly can be quite different for various architectures, depending on the available instruction set.



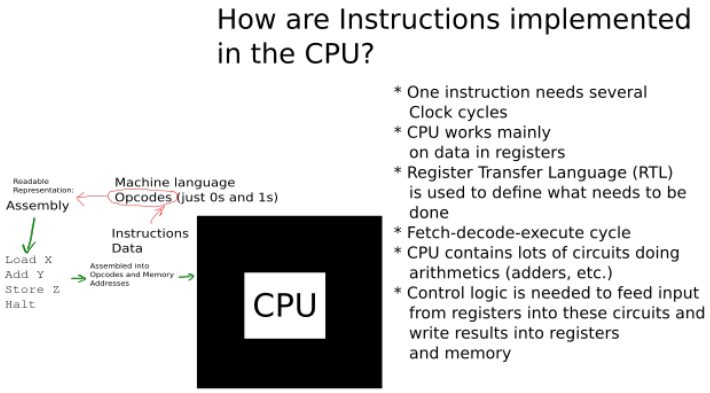
As you have seen with MARIE, explicitly defining which data needs to be loaded into which register etc. can be quite tedious. Furthermore, Assembly programs are not transferable to other architectures. Programs would need to be totally re-written for each architecture they are supposed to run on. Thus, people came up with the idea of creating higher level programming languages, such as Python, C, Java, or Lisp. The first programming languages were FORTRAN, B0/COBOL and LISP in the early 1950s. Now, there

is exist lots of different more-or-less specialised programming languages. What they have in common is, that all of them need to be somehow translated into the particular CPUs instruction set in order to be executed on a computer. Thus, compiler or interpreter transfer code from higher-level languages into actually executable code. Obviously, this executable code cannot be just executed on other architectures, while the code in a higher level programming language in principle just needs a different compiler. Note that this is not always as easy as it sounds, as different architectures might have different constraints on what can be done.

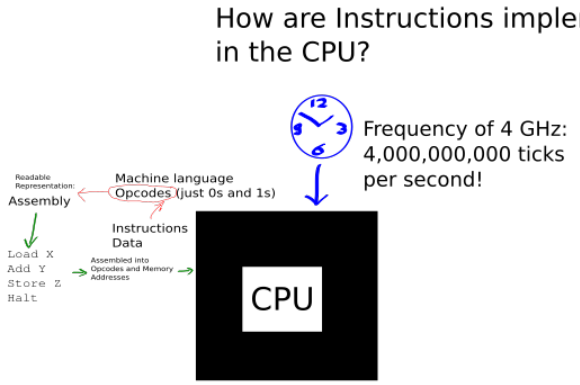


Some programming languages can be compiled into a so-called Bytecode that needs another software layer (i.e. a virtual machine) in order to run.

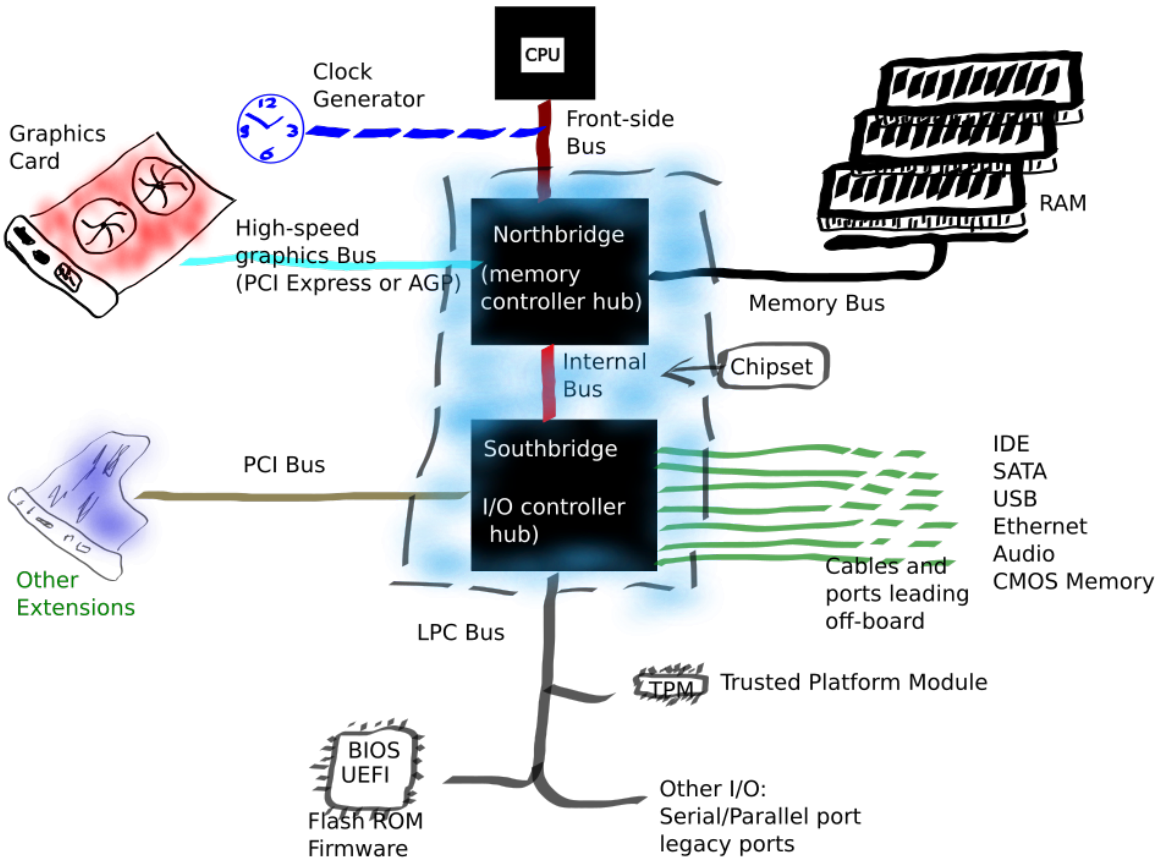
Now, lets briefly revise how instructions actually run in the CPU. We know from MARIE that one single instruction needs several *clock cycles* to go through the fetch-decode-execute cycle. Register Transfer Language is used to define what should be done by the CPU in the different steps. Inside the CPU there are many circuits that actually do the computation and arithmetic. Just to give some idea of the scale. A NAND gate can be implemented with as few as two transistors and current processors for PCs come with up to 5 Billion transistors. Clock cycles are produced by a very fast ticking clock. These clocks do not really tick. Instead, they use oscillations of quartz crystals. If you read a frequency of 4 GHz, this actually means 4 Billion cycles per second. Now, if we assume an (artificial) average of 10 cycles for one



instructions, how many instructions can the processor do per second? Reveal [2]



Obviously, a computer needs a few more than just CPU and memory. The following sketch shows some of the elements that you will find in most PCs.



Lets go through it from top to bottom:

- We already know the CPU and clock. It should be noted that modern CPUs contain a part of memory known as *Cache*. This memory takes part of the main memory and copies it as some kind of working copy into this cache memory. The advantage is, that cache is much faster and therefore, data can be delivered to the CPU in a faster way.
- The actual memory of the system is the RAM (random access memory). It is usually writable and volatile (i.e. stored data disappears if power is switched off). RAM is much faster than non-volatile storage on hard-disks, USB drives etc. Furthermore, access to data happens at almost the same speed for all locations.
- RAM is connected to the system via the Memory Bus and a set of electrical elements / chips, called the Northbridge (or memory controller hub).
- Also connected to the northbridge we find other components that require higher speed. E.g. graphics cards are connected via high speed busses such as PCI Express or AGP.
- I/O and slower components are connected via the southbridge. North- and southbridge together are what is commonly called the *chipset.*
- Southbridge is also called I/O controller hub. All wires leading off the board are connected to the southbridge.
- Examples include busses to connect harddisks to the board (IDE, SATA), networking (Ethernet), audio, universal serial bus (USB), Firewire, etc.
- The PCI bus can be used to connect other extensions that do not need the high speed of the PCI Express bus.
- Finally, the rather outdated and slow low-pincount bus (LPC) is used to connect the security chip rusted Platform Module TPM and the ROM (read-only memory) containing the first code to be executed after switching on the computer. In PCs, this code was called the BIOS (Basic Input/Output system), that has now been largely replaced by UEFI, the Unified Extensible Firmware Interface.

Boot process

Step 1: Turn on power

Power supply starts and provides energy to the motherboard and other components in the computer. Components should only really start to work after a stable power level is established.
A **power good** signal can be sent to the motherboard which triggers the timer chip to reset the processor and start clock ticks.

- First the main fan starts (or other cooling) and motherboards clock-cycle starts.
- CPU gets power and starts working.

Now, there is power available and the CPU, we are ready to use the computer, right?

RevealBoot1 [3]
Booting means to load software step-by-step and activate all components one step after the other.

But why it is called *Booting*?

RevealBoot2 [4]

Step 2: Initial software

BIOS (Basic Input Output System or first steps of UEFI Unified Extensible Firmware Interface in modern PCs) is stored in non-volatile memory (ROM - read only memory) on the motherboard.
It controls the start-up steps, provides initial system configuration (power saving, security, etc.), and

initially configures accessible hardware.

- The reset command in the CPU triggers the execution of an instruction at a specific location in the BIOS chip.
- Location contains a Jump instruction that points to the actual BIOS start-up program in the chip.
- Booting really starts with the execution of this start-up program.

Boot process: POST

BIOS starts with a **power-on-self-test POST**:

- System memory is OK
- System clock / timer is running
- Processor is OK
- Keyboard is present
- Screen display memory is working
- BIOS is not corrupted

Results of POST usually was communicated through system *beep* on devices starting with a BIOS.

Boot process: Video card

The first thing after a successful POST is to initialise the video card and show some initial message on the screen.
Note that the BIOS can only do a rudimentary initialisation. Use of 3D, fancy graphics, etc. needs additional software, the so-called *driver*.

Boot process: Other hardware

Then, the BIOS goes through all available hardware and initialises as far as possible without more complex driver software (UEFI has more options).
Examples are type and size of hard-disk, DVD drive, timing of RAM (random access memory) chips, networking, sound, etc.

Boot process: Find Operating System

Only devices with very restricted resources have all the software needed to run (the Operating System) stored in the firmware. All other computers need to load the Operating System (e.g. Windows, Linux, Android, iOS) from some non-volatile storage. This storage must be configured to support booting from it and it must be enabled for booting in the BIOS configuration.

Thus, BIOS needs to look for a *bootable drive*. This can be on a hard-disk, SD-Card, USB Stick, DVD, floppy disk, ...
The order that BIOS checks for anything bootable is defined in the BIOS configuration (usually accessible by holding a particular key while start-up screen is shown). In UEFI systems that enable faster booting, this option is not always available, it needs first to be activated in the system configuration on a running system or becomes automatically available of the Operating System fails to load.

Boot process: Boot sector

On a bootable drive, there needs to be a *boot sector* with code that can be executed (called the *boot loader*). On a hard disk, this information is in the *Master Boot Record (MBR)*. The boot loader first loads the core part of the operating system, the *kernel*. Then it loads various modules, device drivers, etc.

Once all drivers are loaded, the *Graphical User interface GUI* is started and personal settings are loaded. The computer is ready to use.

BIOS versus UEFI

BIOS is outdated. It was intended to be an abstraction layer to access I/O, but it is no longer used for this and never really was.
Restrictions include only 1,024 kilobytes of space for BIOS code, only works with hard-drives up to 2.2 terabyte hard-disks, cannot work with various other current technology (and future technology).
UEFI or Unified Extensible Firmware Interface is a programmable interface. It is software that is sits in some non-volatile memory, but not necessarily all in a separate chip, such as BIOS. It rather works on top of the computer's firmware, and can actually run on top of a BIOS).

- Can address hard-disks up to 9.4 zettabytes (1 zettabyte is about a billion terabytes).
- Provides access to all hardware. Faster hardware initialization.
- Security and authentication features before the OS has started.
- Network access before the OS has started.

There was quite bit of criticism when UEFI was introduced:

- Boot restrictions (i.e. *secure boot*) can prevent users from installing the operating system of their choice.
- Additional complexity provides additional possibilities for errors and new attack vectors.

Most new PCs now use UEFI.

A little bit on values/sizes/measurements

In the context of computers, storage, networks, etc. some prefixes are used to describe very large or very small numbers. Examples of such prefixes are Giga-, Kilo-, Mega-, Nano-. They denote that a number needs to be multiplied by a power of 10 to get the correct value. The following two tables show prefixes based on powers of 10 for large numbers and small numbers.

Prefix	Symbol	Factor
	Base	**$10^0 = 1$**
deka-	da	$10^1 = 10$
hecto-	h	$10^2 = 100$
kilo-	**k**	**$10^3 = 1{,}000$**
mega-	**M**	**$10^6 = 1{,}000{,}000$**
giga-	**G**	**$10^9 = 1{,}000{,}000{,}000$**

Prefix	Symbol	Factor
tera-	**T**	**10^{12} = 1,000,000,000,000**
peta-	**P**	**10^{15} = 1,000,000,000,000,000**

Prefix	Symbol	Factor
	Base	**10^{0} = 1**
deci-	**d**	**10^{-1} = 0.1**
centi-	**c**	**10^{-2} = 0.01**
bitsmilli-	**m**	**10^{-3} = 0.001**
micro-	**µ**	**10^{-6} = 0.000,001**
nano-	**n**	**10^{-9} = 0.000,000,001**
pico-	**p**	**10^{-12} = 0.000,000,000,001**
femto-	**f**	**10^{-15} = 0,000,000,000,000,001**

In computers, we often have numbers that are powers of 2 rather than powers of 10. One example is the capacity of memory.

How many memory addresses can we get from *n* bits? Revealn [5]

Very often, there is confusion if 100 Gigabyte (GB) storage actually mean 10^9 bytes or 2^{30} bytes. The difference is not that small. Therefore, a standard was developed as part of the ISO/IEC 80000 standard (International System of Quantities). The following table shows prefixes for a basis 2.

Prefix	Symbol	Factor
Bytes (Bits)	B (bit)	**2^{0} = 1**
Kibi-	**KiB (Kibit)**	**2^{10} = 1024**
Mebi-	**MiB (Mibit)**	**2^{20} = 1,048,576**
Gibi-	**GiB (Gibit)**	**2^{30} = 1,073,741,824**
Tebi-	**TiB (Tibit)**	**2^{40} = 1,099,511,627,776**
Pebi-	**PiB (Pibit)**	**2^{50} = 1,125,899,906,842,624**

↩1 Our toy CPU MARIE has 16-bit words. However, it only uses 12 bits for addressing. It can address \(2^{12}\) locations with 16 bits stored in each of these locations. Thus, it can address \(2^{12}\times 2 \) bytes.

↩2 This was probably too easy: It is just a tenth of the clock cycles per second: 400 Million instructions per second. But if you cont the number of instructions in your biggest MARIE program, you can do the math and get some idea of what a current CPU can compute within one second.

↩3 No, the CPU cannot do much without software. During this phase of the boot process, the computer is really not able to do much. The CPU does not know about a hard-disk an operating system might sit on or any peripherals connected to the computer (mouse, keyboard, monitor, hardware, DVD-drive, printer, etc.).

↩4 Wikipedia says: "*Boot* is short for *bootstrap* or *bootstrap load* and derives from the phrase *to pull oneself up by one's bootstraps* *(https://en.wikipedia.org/wiki/Bootstrapping#Etymology)*. The usage calls attention to the requirement that, if most software is loaded onto a computer by other software already running on the computer, some mechanism must exist to load the initial software onto the computer."

↩5 \(2^n\)

6
Operating Systems

An *Operating System* (OS) is a piece of software (a program), or a collection of different pieces of software, that manages the *resources* in a computer and provides a convenient interface between application programs and the hardware.

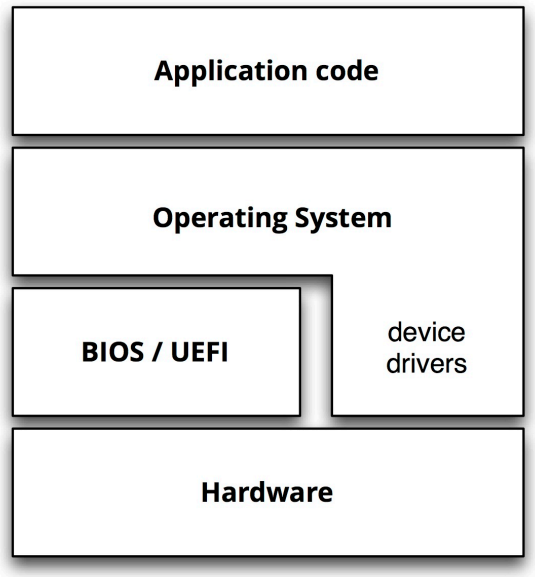
Most people would think of concepts like the Windows menu and task bar, some configuration dialogues, maybe the web browser, or the file system explorer, as essential parts of the operating system. Maybe if you have some programming experience, you would know that the operating system provides the *graphical user interface (GUI)* that applications can use to draw windows, buttons and so on.

But the core functionality of an OS is actually much more fundamental.

6.1 Introduction

What does an OS do?

An operating system provides a *level of abstraction* between hardware and software. This is an important concept in IT: We want to hide complicated, diverse, low-level concepts behind a simple *interface*. The following figure shows how an OS fits into our overall view of a computer:



Any higher-level module only "talks" directly to the modules with which it has a boundary. In this example, an application program would never talk to the hardware directly - rather, it uses well-defined interfaces in the OS to access things like the network or the graphics hardware. The big advantage is that, as an application programmer, you don't need to know how the hardware works. For example, you don't need to know whether your computer is connected to the Internet via cable-based Ethernet or wireless 4G networking (which are *very* different technologies) - you can simply call a function in the OS (like a subroutine) to send a message to another computer via the Internet!

More broadly, Operating Systems have the following core tasks:

- Managing *multiple processes running in parallel.* A process is a program that is currently being executed.
- Managing the *memory* that processes use.
- Provide access to file systems, the network and other I/O resources.

Modern OSs provide many more functions (such as the graphical user interface or a number of system tools mentioned above), but we will focus on these core tasks here. The core functionality of an OS is provided by the **Operating System Kernel**, or kernel, for short. Almost all other functions are in fact provided as application code that is simply pre-installed when you install an OS.

A bit of history

The very first computers did not have operating systems. The operator would simply load a program into memory (from punched cards), the computer would execute it, and then the operator would load another program. It soon became clear that many programs would require very similar tasks to be performed, such as producing output or computing certain mathematical functions. These would be bundled into *libraries*, which are simply pieces of code that are loaded at well-known memory addresses, so that programmers could simply jump into these pieces of code rather than writing their own subroutines. We can consider these libraries to be very rudimentary operating systems, since they would allow a programmer to use the output functionality of a computer without knowing exactly how to talk to the hardware - the library provides that important level of abstraction.

Operating Systems only became really important when the first computers arrived that had support for *multiprogramming*, the ability to run multiple programs at the same time. This was important since computers were expensive, so this scarce resource needed to be utilised as efficiently as possible. We have seen that CPUs execute an individual instruction at a time (in the module on Central Processing Units (CPUs)) - and while modern CPUs *can* in fact process multiple instructions in parallel, they are still not able to run multiple *programs* in parallel. The innovation of the first multiprogramming OSs was therefore to implement *task switching*, where each program is allowed to run for a certain amount of time, before the CPU switches to the next program. This is still one of the most important pieces of functionality in any modern OS kernel.

With the advent of multiprogramming, another problem appeared, and it also needed to be solved by the OS. When two programs are running at the same time, we have to make sure that they cannot read or write each other's memory. Otherwise, an incorrect (or malicious) program can cause havoc and affect the stability, correctness and security of the entire system. Similarly, if different programs belong to different *users*, the OS must make sure that their files are protected from unauthorised access.

The UNIX line of operating systems was born in this environment. It was deployed on mainframe computers that could be used by hundreds of users, running many processes simultaneously, with reasonable protection against malicious and buggy code.

Many early operating systems for home computers and personal computers did not come with any networking support built-in. Users had to pay for special versions of the OS if they wanted to join a network! Modern Operating Systems include a lot of the features of the original UNIX systems, and many of them are in fact "descendants" of UNIX in one way or another. Mac OS and Linux are both quite closely related to UNIX (in terms of their internal structure), while Windows is less closely related. All modern OSs support running hundreds of processes simultaneously, provide access to a diverse range of hardware, and have full networking support built-in.

Abstraction

The main goal of an OS is to make computers easier to use, both for end users and for programmers.

For end users, an OS typically provides a consistent *user interface*, and it manages *multiple applications* running simultaneously. Most OSs also provide some level of protection against *malicious or buggy code.*

For programmers, the OS provides a *programming interface* that enables easy access to the hardware and input/output devices. The OS also manages system resources such as memory, storage and network.

We can summarise these functions in one word: *abstraction*. The OS hides some of the complexity between consistent, well-documented interfaces - both for the end user, and for the programmer. Abstraction is probably the most important concept in IT! Without many levels of abstraction and interfaces and layers that are built on top of each other, it would be impossible for anyone to master the complexity of a modern computer system.

Processes and Programs

Before we start discussing how an OS achieves this abstraction, we need to introduce the notion of a *process,* and discuss how a process relates to a *program*. A short definition is the following:

A process is a running instance of a program.

So a program is the code that you write, or the result of compiling the code into machine code. In most cases, a program is a file that's stored on disk (or maybe printed in a textbook). A process, on the other hand, is created by loading the program code into the computer and then executing it.

Let's further clarify this distinction. Assume that you work for a software development company. You write some code for the company that gets distributed as a mobile app. This is a *program.* Hopefully your company is really successful, and your app is used by millions of people around the world. On each of your users' devices, an *instance* of your program is running as a process.

OS abstractions through virtualisation

Operating Systems achieve abstraction through *virtualisation*. This means that they provide a *virtual* form of each *physical resource* to each processes. Physical resources include the CPU, the memory, the external storage (hard disks etc) and the network and other I/O devices. Instead of using these physical resources directly, the process uses functionality in the OS to get access, and the OS can provide the "illusion" that each process

- has the CPU completely to itself (i.e., there are no other processes running)
- has a large, contiguous memory just for itself
- has exclusive access to system resources (i.e., it doesn't have to worry about sharing these resources with other processes)

Let's look at an every-day example of virtualisation.

On a normal day, let's say you leave the house at 7:30am, drive to work, park your car at 8:10am, leave work at 5pm, pick up your car from the car park and drive home, arriving at 5:45pm, where the car sits in the garage until the next morning.

In a *car sharing* scenario, your car could be somewhere else while you're not using it, e.g. someone could use it during the day to visit clients, and someone else working night shifts could have it from 6pm till 7am.

Let's assume we can make sure that whenever you and the other two car users need the car, it is right where you and they expect it to be. Then we have turned the one *physical* car into three *virtual* cars.

Operating systems do the same, sharing the scarce physical resources among multiple processes. For each process, the OS makes it look as if it had all the resources to itself.

But it goes further: Imagine everyone could leave some stuff in the glove compartment, but they always only see their own stuff! And the fuel gauge is always at the level where you left the car, no matter who drove it in the meantime. Operating systems isolate processes against each other, preventing them from accessing each others' resources, such as memory and files on disk.

6.2 Virtualising the CPU

The goal of *virtualising* the CPU is to run multiple processes "simultaneously", but in a way that individual processes don't need to know that they are running in parallel with any other process. The Operating System creates this illusion of many processes running in parallel by switching between the processes several times per second. This poses two main challenges:

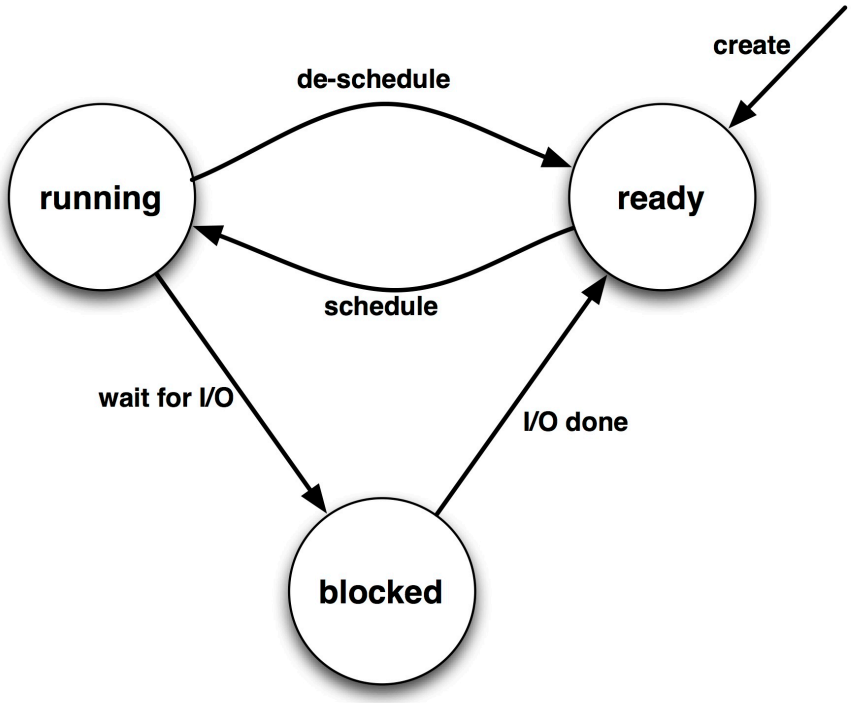
- A process shouldn't notice that it's not running continuously. For example, let's assume that the Operating System executes process *A* for 100 milliseconds, then switches to *B* for another 100 milliseconds, and then switches back to *A*. The programmer who writes the code for *A* shouldn't have to take into account which other processes might be running at the same time as *A*.
- Each process should get a fair share of the overall CPU time. Otherwise, the illusion of concurrency would break down. For example, if you want to play some music in the background while scrolling through a web page, the music player process and the web browser process both need to get enough time every second to do their work. If for one second, only the music player had the CPU to itself, the scrolling would start to lag behind, but if only the web browser was "hogging" the CPU, the music would stop.

The following sections develop solutions to these two challenges. The first section deals with *how* to virtualise the CPU, i.e., the *mechanisms* that the OS uses to switch between processes. The second section explains *when* to switch between processes, i.e., the *policies* that the OS uses to determine which process gets how much time, and which process to switch to next.

Virtualisation mechanisms

The mechanisms for virtualising a CPU classify each process as being in one of three states: *ready*, *running*, or *blocked*. When the process is created by loading program code into memory, it is put into the *ready* state. *Ready* means ready for execution, but not currently being executed by the CPU. When the OS decides that the process should now be executed, it puts it into the *running* state. We say that the OS *schedules* the process. In the *running* state, the code for the process is actually executed on the CPU. Now one of two things can happen. Either the OS decides that the time for this process is up. In this case, the process is *de-scheduled*, putting it back into the *ready* state, from which it will be scheduled again in the future. Or, the process requests some I/O to happen, e.g. opening a file from an external storage. Since I/O can take a long time, and the process will have to wait until the file has been opened, the OS takes advantage of this opportunity and puts the process into the *blocked* state. As soon as the I/O is finished, the process will be put back into *ready*, from where it will be scheduled again.

Here's a picture (a so-called *state transition diagram*) that summarises these concepts:



Using the example of a music player and a web browser running concurrently, here is a table of different process states they could be in over a number of time steps.

Time	Media Player (MP)	Web Browser (WB)	Description
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	MP initiates I/O
4	Blocked	Running	OS has switched to WB
5	Blocked	Running	
6	Ready	Running	I/O has finished
7	Ready	Running	
8	Running	Ready	OS has switched back to MP
9	Running	Ready	
10	Ready	Running	OS has switched back to WB

Limited Direct Execution

CPU virtualisation via process switching as discussed above has a number of challenges. The first one is *performance*. Clearly, virtualisation should not create a huge overhead, and most of the CPU time should be spent on actually running processes, not managing them. The second challenge is *control,* which means that the OS should enable fair scheduling of processes and offer a certain level of protection against malicious or buggy code.

The solution to these challenges is to get some support from the hardware, in a mechanism called *limited direct execution (LDE).* In order to achieve good performance, each process is allowed to run directly on the CPU. That is what *direct* means in LDE.

Now remember that the OS is nothing but a piece of software, and while the code for a process is executed on the CPU, clearly the code for the OS is not executed at the same time. So how can the OS switch processes or protect against buggy code while it is not running? That's where the *limited* in LDE becomes important. CPUs have different *modes* of operation that enable the OS to restrict what an application program can do.

In **user mode**, only a subset of all instructions are allowed to be executed. Typically, instructions that perform I/O would not be allowed in user mode, and we will see in the module on Virtual Memory that certain memory operations are also blocked. Normal applications are executed in user mode.

In **kernel mode**, code is run without any restrictions, i.e., it has full access to I/O devices and memory. The OS runs in kernel mode. Whenever an interrupt happens (as discussed in Input/Output devices), the CPU switches into kernel mode to execute the interrupt handler (which is part of the OS, since it deals with I/O). Kernel mode is sometimes also called *supervisor* mode.

Clearly, user mode places quite severe restrictions on application programs. Without access to I/O instructions, a process cannot read any files, send or receive messages over the network, or draw characters on the screen. But applications need to do all those things! So in order to give user mode processes access to these "privileged" operations, the OS provides a number of "hooks" that a user mode process can use, much like a subroutine, to perform functions like reading from disk or sending a message over the network. These hooks are called ***system calls***.

System calls

A system call is, at its core, a special CPU instruction that switches the CPU from user mode into kernel mode and then jumps to a special subroutine of the OS. Many CPU architectures simply provide an instruction that causes an interrupt - as mentioned above, any interrupt will cause the CPU to switch to kernel mode. We call this a *software interrupt*, in order to distinguish it from interrupts caused by I/O devices. Some architectures (such as Intel x86) provide special instructions for triggering a system call, which can be executed faster than an interrupt.

The application code that runs in user mode of course needs to let the OS know *what* kind of privileged operation it wants to perform (open a file or send a message?). To enable this, the OS sets up a **table of system call handlers**. This table is just a contiguous block of memory, and each location contains an address of a subroutine that performs one particular function. A user mode application can then put the number of the OS subroutine it wants to call into a register before triggering an interrupt or calling the special system call instruction. The following steps are then going to happen:

1. The CPU is switched into kernel mode
2. It jumps to the interrupt handler for software interrupts
3. The interrupt handler saves the *process context* into memory (i.e., the current state of registers etc)
4. The interrupt handler makes an indirect jump to entry *i* of the system call table, if *i* was the number that the user mode application stored in the register before triggering the interrupt.
5. The code for the system call handler is executed and returns to the interrupt handler.
6. The interrupt handler restores the process context from memory.
7. The CPU is switched back to user mode.
8. The interrupt handler makes a jump to return to the user space application that called it.

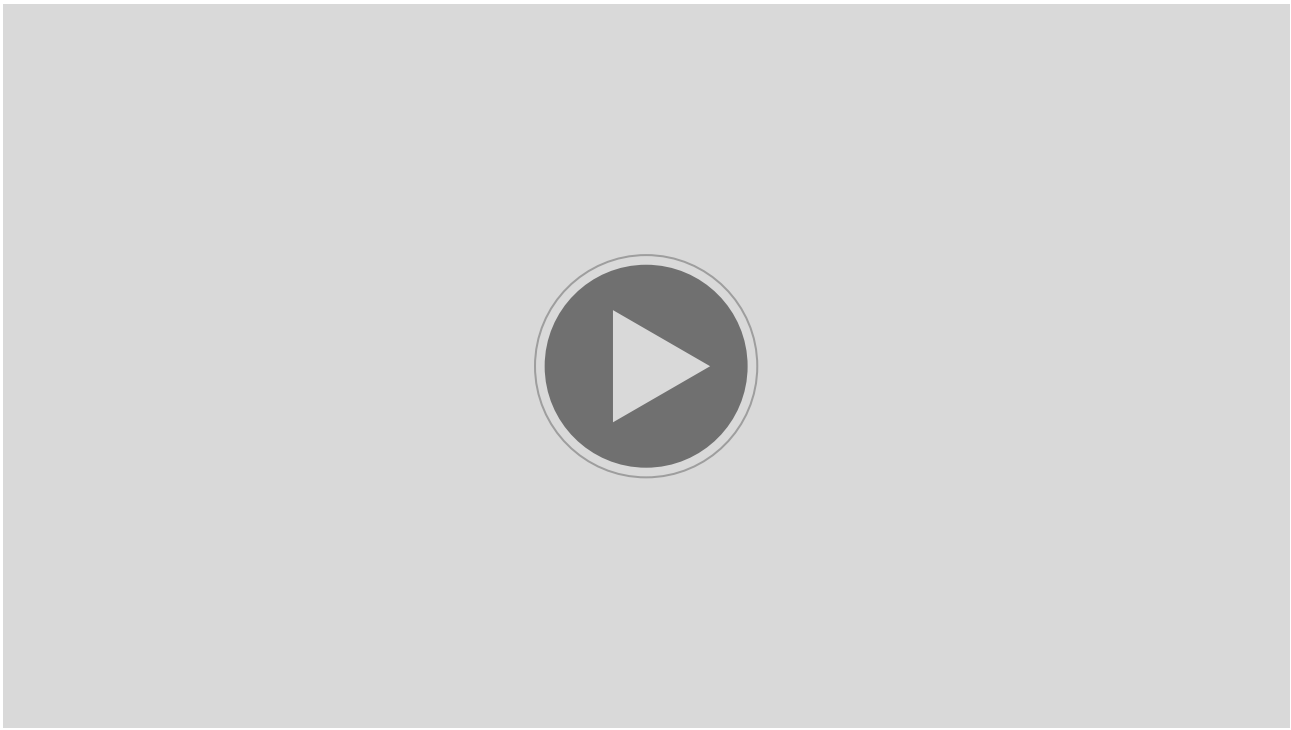
All in all, this is the same mechanism as an interrupt vector, which we discussed in the module on Input/Output devices. An important step for both interrupt vectors and system call tables is the saving and restoring of the process context. That's why transitions from user mode to kernel mode (and back)

are also called **context switches**.

Process switching

In the discussion so far, one step was still missing: How does an OS switch from one process to another one? Just look again at step 6 above: *The interrupt handler restores the process context from memory.* Since this interrupt handler is part of the OS, it can simply decide to restore a *different* process context. In fact, the OS keeps process contexts stored in memory for all processes that are currently in the *ready* or *blocked* state. Process switching therefore means to pick the process context of a process that is currently *ready*, and then jump into that process's code in step 8 above.

The video below shows how a CPU might switch between processes when an interrupt is triggered.



(https://www.alexandriarepository.org/wp-content/uploads/20170405140200/process_switch.mp4.mp4)

Cooperative and Preemptive Timesharing

As we have seen above, the mechanism that triggers a switch from user mode applications into the OS kernel is an interrupt. This includes "real" I/O interrupts, but also software interrupts (system calls) that the application makes.

If we only have these two sources of interrupts, it is possible that the OS will not get an opportunity to switch to a different process for a long time. Imagine a process that just performs a long running computation, without making any system calls, and without any interrupts happening from I/O devices. In that case, the process would block the entire system, since the OS never gets a chance to switch to another process. We call this **cooperative** timesharing, because all processes must cooperate with the OS and make system calls in regular intervals, otherwise some other processes can "starve" (i.e., not get scheduled at all).

The advantage of cooperative timesharing is that it is relatively easy to implement, but the downside is that buggy or malicious processes can make a system unusable.

In order to address these disadvantages, modern computer architectures introduce a third type of interrupt (in addition to I/O and software interrupts): *timer* interrupts. These are hardware circuits that generate an interrupt in regular intervals, usually after a programmable number of clock ticks. This gives the OS full control: it sets up a timer interrupt before executing the context switch to a process, so it can be guaranteed that the process will be preempted at the latest when the timer "fires", if it doesn't make any system calls before that. Consequently, we call this **preemptive** timesharing.

In preemptive timesharing systems, the OS (or the user) can always kill buggy or malicious processes (e.g. through the task manager in Windows or the `kill` command in Linux and Mac OS), since the OS will regain control of the system several times per second.

Summary

Let's quickly summarise the mechanism for process switching. The CPU has a user mode and a kernel mode, in order to control the I/O and memory access for applications, which use system calls into the OS to access the privileged operations. Interrupts cause the CPU to switch into kernel mode. These can be I/O interrupts, software interrupts (system calls), or timer interrupts. The latter enable preemptive timesharing, where the OS always regains control of the system several times per second.

Now that we have seen *how* to switch between processes, let's look at how the OS makes the decision *when* to switch.

Process scheduling

This section deals with the *policies* that the Operating System uses in order to switch between processes, the so-called *process scheduling.* The OS needs to decide how long each process gets to use the CPU before it switches to a different process, and which process to switch to next.

There are multiple conflicting goals that an OS could try to achieve with its scheduling policy. One goal could be to finish each process as quickly as possible, i.e., try to achieve an optimal *turnaround time* (the time between the process becoming *ready* for the first time and its completion). A different goal may be to allocate the same amount of CPU time to each process in a given time interval, e.g. if 10 processes need to be executed concurrently, make sure each of them gets roughly 1/10 of the CPU time on average. This would achieve a level of *fairness* between the processes.

In the following we will look at three simple scheduling policies: *first-come first-served, shortest job first,* and *round-robin* scheduling.

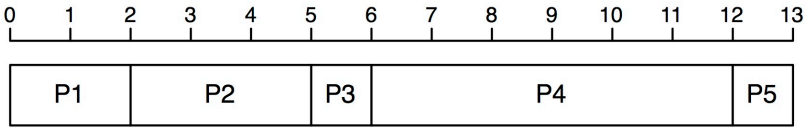
First-come first-served

This is a simple policy that you are familiar with from the supermarket checkout. Let's assume a single checkout is open, and five people are queuing. The first person buys two items, the next one three, the third has a single item, the fourth buys six items, and you are the final customer with just a single item in your trolley. Let's make a simple assumption that the time each customer takes at the checkout just depends on the number of items they buy. So customer 1 required two time units, customer 4 requires six time units, and you take a single time unit.

We can now define a *metric* for how good the schedule is. A usual metric would be the *turnaround time*, which is the time between arriving at the end of the queue and leaving the supermarket. Let's assume

that all customer arrive roughly at the same time, then your turnaround time would be 2+3+1+6+1=13 time units (if each item takes one minute to process, you'd be waiting 13 minutes at the checkout before you can leave the supermarket).

Now for you as an individual that's bad news: you only have a single item to buy, and you're waiting for 13 minutes! But even overall this schedule isn't great. We can compute the *average* turnaround time over all customers, which in this example would be $\frac{2+(2+3)+(2+3+1)+(2+3+1+6)+(2+3+1+6+1)}{5} = 7.6$.

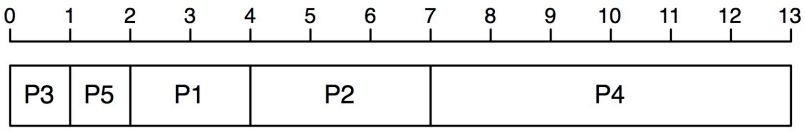


$$\frac{2+5+6+12+13}{5} = 7.6$$

So on average, these five customers wait 7.6 time units before they can leave. Let's look at a policy that can improve this average.

Shortest job first

If we sort the customers by the number of items they want to buy, in increasing order, we get the following schedule:



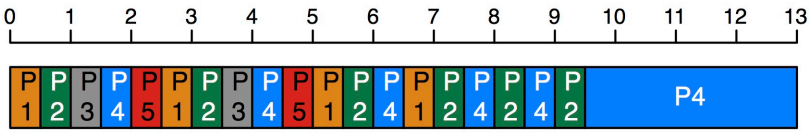
$$\frac{1+2+4+7+13}{5} = 5.4$$

As you can see, the average turnaround time has been reduced to 5.4, and in fact it's possible to show that this policy always results in an optimal schedule with respect to turnaround time. We call this the *shortest job first* policy. Of course it wouldn't be that easy to implement this strategy in a supermarket with a single checkout, because customers would get angry if we start allowing people with few items in their trolleys to jump the queue. But adding "express checkouts" for customers with few items has a similar effect.

Both *first-come first-served* and *shortest job first* assume perfect knowledge about the time that a "job" takes from start to completion. In some types of scheduling problems, this may be a valid assumption (for example when scheduling the production lines in a factory). In other situations, we may be able to make good guesses that approximate the actual job duration. But in the case of scheduling processes in an Operating System, we typically have no idea how long a single process will take to finish. Some processes are in fact not designed to ever finish. Can you think of a process for which that would be true? Reveal [1]
Furthermore, the OS needs to not only schedule a process and then wait until it's finished, but also *preempt* processes and switch to other ones in order to create the illusion of concurrency. The next policy can deal with these requirements.

Round-robin scheduling

Compared to the previous two policies, the next one will split up each process into short *time slices.* The OS can then cycle through all the processes one by one.



In the figure, you can see how P1 (which takes two time units in total) has been split into four short time slices, and P2 has been split into six. The schedule first cycles through all five processes twice. After that, P3 and P5 have already finished, which are repeated twice. Then P1 finishes, and we get P2, P4, P2, and then the rest of P4.

This type of scheduling produces a *fair* schedule, which means that during a certain time interval, all processes get roughly equal access to the CPU. The shorter we make each time slice, the fairer the schedule is going to be. But on the other hand, each time the OS switches between different processes, it has to fire a timer interrupt, execute the interrupt handler, and perform the context switch to the new process. This switching takes time, so if we make the time slices too short, we will create a noticeable overhead. The OS needs to make the right compromise between fairness and efficiency here.

Another problem with simple round-robin scheduling is that some processes may be more important than others. For example, if a video player cannot update the picture at least 25 times per second, the video will begin to flicker. A process that just performs a lengthy computation in the background, on the other hand, won't be affected as much if it gets less access to the CPU for a short while. Modern Operating Systems therefore implement a variant of round-robin scheduling that can give certain processes (such as the video player) **higher priority** than others.

↩1 E.g. a web server process, or the process that handles the task bar

6.3 Virtualising the memory

Limited Direct Execution, i.e., virtualising the CPU, takes care of limiting access to the CPU as well as to the I/O devices. However, it does not limit access to the *memory*. Virtualising the memory has three main goals:

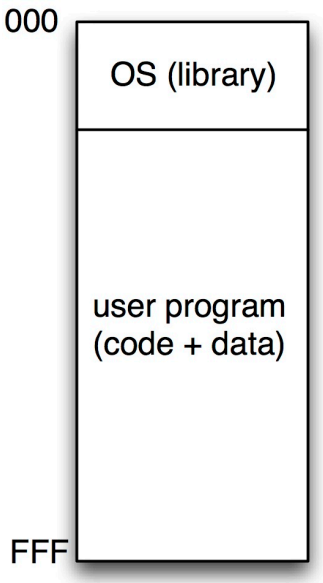
1. To enable *protection* of a process's memory against access from other (malicious or buggy) processes.
2. To make *programming easier* because a programmer does not need to know exactly how the memory on the target computer is organised (e.g. how much RAM is installed, and at which address the program will be loaded into RAM).
3. To enable processes to use more memory than is physically installed as RAM in the computer, by using external storage (e.g. hard disks) as temporary memory.

Virtualising the CPU meant that for each process, it looks as if it had exclusive access to the CPU. The same holds for virtual memory: a process does not need to "know" that there are other processes with which it shares the memory.

Address space

We call the addresses that can be used by a process its *address space*. For example, as a MARIE programmer, you can use the entire RAM available in MARIE, so the address space of a program running on a MARIE CPU is `000` to FFF (since addresses in MARIE are 12 bits wide).

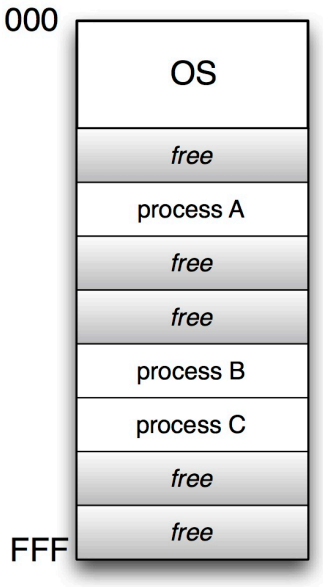
This is the same situation as in early computers, which only ran a single program at a time. The OS was typically loaded as a library, starting at address `0`, and the user program and data used the addresses behind the OS, as illustrated in the figure below.



Since there is only one program (we can't really speak of processes since there is no scheduling or concurrency) running at a time, there is no need for memory protection or creating the illusion that the program has the memory to itself.

Multiprogramming

The first multiprogramming systems divided up the memory, and each process was allocated a fixed region that it was allowed to use. This figure shows three processes sharing the memory, and a few regions of memory still being free for new processes.



This setup poses two challenges. First, you don't know where a program will start in memory when you write it. Remember how instructions in MARIE require you to write down concrete addresses, such as `Add 100` (meaning add the value stored in address `100` to the `AC` register). Even if you use symbolic labels, as soon as you assemble your program into machine code, those labels get replaced by fixed, concrete addresses. You can imagine that programming like this is pretty much impossible if you don't know at which address your code will start when it gets loaded by the OS! So the first challenge is to enable programmers to program *as if* their code was loaded at address `0`, and then have a mechanism that makes sure that those *virtual* addresses are translated to *physical* addresses when the code is loaded or executed.

The second challenge is that, in the picture above, nothing prevents process B from reading or writing the memory area that's been set up for process A or C. If the code for B contains bugs, it may accidentally overwrite the code for process A in memory, so that when the OS continues executing A, it crashes or behaves in an undesired way. Even worse, if B contains *malicious* code, it could not only make A crash, it could also access sensitive data (imagine A uses your credit card details) or modify A to do something the user doesn't want. That's essentially how viruses work.

Virtual memory therefore means that the operating system, together with the hardware, creates the illusion that each process has its own address space, and it prevents any process from accessing any other process's address space.

Virtual memory

In a virtual memory system, the instructions operate on *virtual addresses*, which the OS, together with the hardware, translates into *physical addresses* (i.e., the actually addresses of the RAM).

Virtualising the addresses

A simple approach for implementing virtual addresses uses an additional register, the **base register** B. At any point in time, it simply has to contain the address at which the code for the currently running process starts in memory.

Any instruction that accesses the memory now always adds the value stored in B to the address that's being used. For example, `Load 100` now really means `Load B+100`, `Add A300` means `Add B+A300` and so on. Note that this doesn't change the instruction set, i.e., a programmer still codes as if the addresses start at `0` (like in MARIE), but the CPU takes into account the base register B when executing the instructions.

Let's assume process A, in the figure above, consists of the following code:

```
Load  200
Add   300
Store 200
Halt
```

Let's further assume that the code was loaded into RAM starting at address `A000`. When the OS schedules process A, it needs to set the base register B to `A000` before jumping to the first instruction. Each instruction uses the base register for any memory access, so the first instruction will load the value stored at address `A000+200`, the second one will add to that the value stored at `A000+300`, and the third instruction will store the result back into `A000+200`. Of course, if the code gets loaded into RAM at a different starting address, say `E200`, all the OS needs to do is change the value of the B register to `E200` before executing the code.

Since the behaviour of all instructions is now different (compared to a system without a base register), we should be able to write the modified RTL for each instruction (note: the discussion about RTL here will not be assessed). The change is very minimal: every RTL step that involves writing to the MAR now needs to add the value of B to the address. For example, here are the steps required for a modified `Load X` instruction:

```
1. MAR ← B + PC
2. MBR ← M[MAR]
3. IR ← MBR
4. PC ← PC+1
5. MAR ← B + X
6. MBR ← M[MAR]
7. AC ← MBR
```

Note that both step 1 and step 5 now add the value stored in the base register B to the address being loaded. Step 1 in particular is interesting: the PC now is also *relative* to the process's base address. Here is the RTL code for another instruction, `JnS X`:

```
1. MAR ← B + PC
2. MBR ← M[MAR]
3. IR ← MBR
4. PC ← PC+1
5. MAR ← B + X
6. MBR ← PC
7. M[MAR] ← MBR
```

```
8. AC ← X
9. PC ← AC + 1
```

You may notice a change in step 8 compared to the RTL explained in From Instructions to Circuits, which contained the line `AC ← MAR`. The change is due to the fact that we've already added the value of B in line 5, but we need the "raw" value X as the address to jump to.

Memory protection

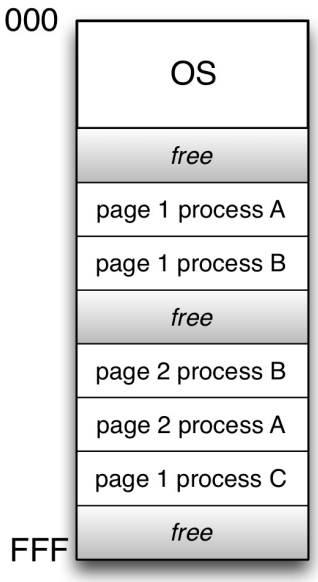
Now that we know how the OS can give every process its own address space, the second task is to make sure it can only read and write inside its own address space, too. Just using the simple base register B is not enough: If a process has been allocated the physical addresses from, say, `A00` to `B00`, it could still execute an instruction `Load 105`. In that case, since the base register would be set to `A00`, it would actually load from address `A00+105=B05`, which is outside of its address space.

Staying with our simple model, we can extend the system by one more register, the **bounds register**, which contains the highest address that the current process is allowed to access. The CPU will then check for each memory access whether it is an address between the base register and the bounds register. If the process tries to access memory outside of its address space, this generates an interrupt, which causes a context switch and gives the operating system the chance to kill the process.

Real virtual memory systems

The simple approach explained above is a bit unrealistic, and wouldn't work for modern computers and operating systems. The main drawback is that each process needs to be allocated a fixed block of physical RAM. Often, the OS can't know yet how much RAM a particular process will need when it loads it. E.g., you can't know exactly how long a document you're going to write when you open a word processor application.

Realistic virtual memory systems therefore implement a more complex approach, where memory is allocated in smaller chunks called *pages*. The OS keeps a list of pages for each process (rather than just a single block). A process can request a block of memory to use from the OS, which will either return an address from an existing page associated with the process, or add a new page for the process if the existing ones are already full. The advantage is that the pages for a single process don't all have to form a contiguous region of physical memory, as illustrated below.



Clearly, this creates more work for the OS, since it needs to keep track of the mapping from virtual addresses (inside the process) to physical memory pages. A simple base register isn't enough any more. Modern CPUs in fact contain complex *Memory Management Units,* which are hardware circuits that perform much of the "paging" for the OS.

A paged virtual memory system has another big advantage. A page of virtual memory doesn't need to exist at all in physical RAM! For example, if a process hasn't used a certain page for a while, the OS could decide to store the contents of that page to the external storage (hard disk), and reuse the memory for other processes. As soon as the original process tries to access that page again, the Memory Management Unit will cause an interrupt (because the process tries to read from a page that doesn't exist), which gives the OS an opportunity to load the page back from disk (probably in exchange for another page that hasn't been used for a while).

This works very well as long as the "swapping" of pages from memory to the hard disk and back doesn't become too frequent. You can sometimes observe a computer becoming unresponsive if an individual process requests so much memory that the OS is busy doing nothing but taking care of page swaps.

7
Networks

Computer networks have become so important that it is almost unthinkable to have a computer that is not somehow "connected". We can access the Internet from our laptops, phones, and desktop computers, and most of the applications we use all the time require a network connection to be useful. But even things like light bulbs, thermostats and door locks are becoming "smart" and form part of the "Internet of Things" (IoT). In this module, you will discover the basic technology that has driven this innovation or rather, this revolution, for the last three to four decades.

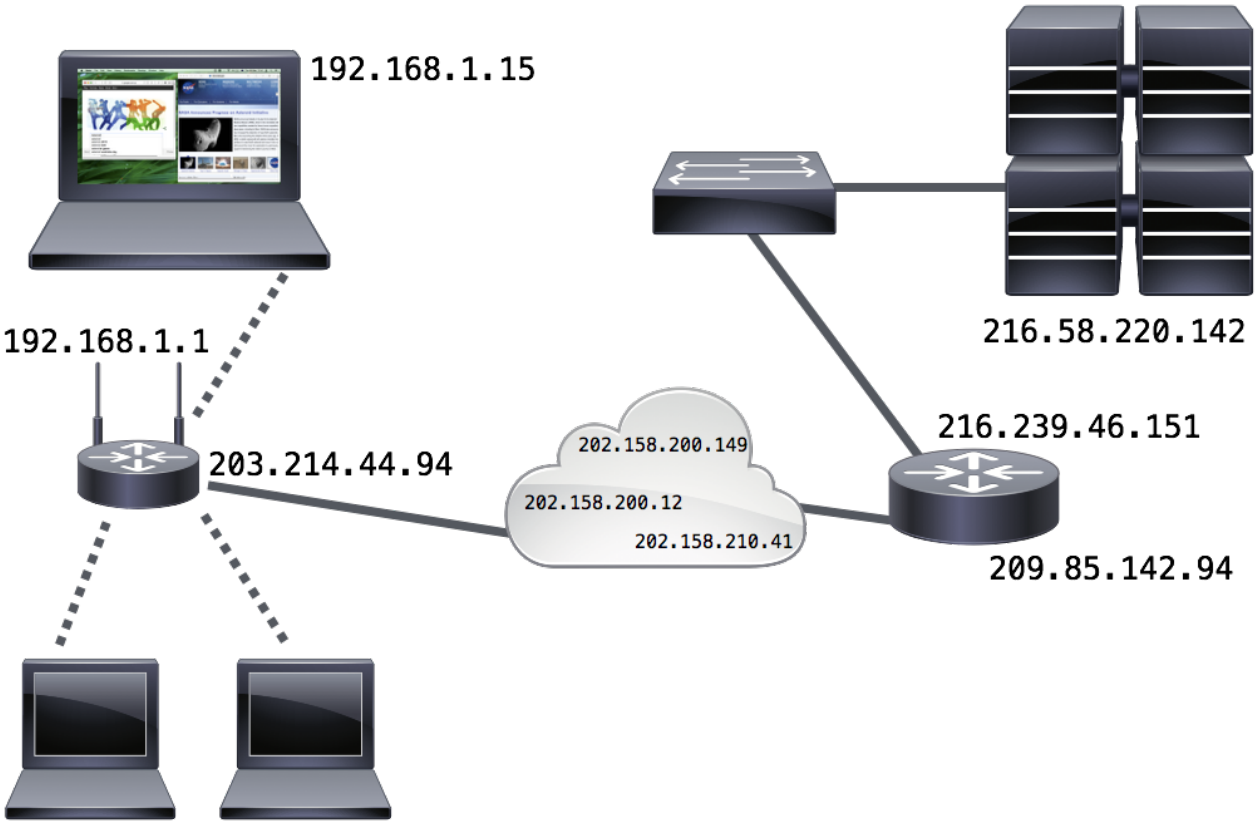
7.1 Introduction

The primary goal of connecting different computers with each other in a network is to enable them to *communicate*, i.e., to exchange information. Based on this simple idea of being able to transmit data from one computer to another, we can build all the applications that we have become so familiar with: the World Wide Web, instant messaging, video conferencing, online multiplayer games, ride-sharing apps, or video streaming services, to name just a few.

In this module, we will introduce the components that networks are comprised of and the different types of network architectures, and define some of the basic notions used in networking such as transmission rates, latency, and addresses.

Network structure

Let's start with the following diagram, which shows a typical network, consisting of a number of different devices.



The left half shows a wireless local area network (WLAN) with three laptops, connected to a WLAN access point and router in the middle. The right half shows a network with a server (the large box) that's connected to a router via a switch. The two networks are both connected to the Internet, which we represent as a cloud. This simple network contains the four main hardware components that are used to build all modern computer networks.

A **client** is a device (e.g. a computer or a smart phone) that enables users to *access* the network, such as the laptops in the picture above. Clients can also be very specialised devices, for instance IP phones, which are telephones that are directly connected to the computer network rather than to a dedicated telephone network (e.g. all the telephones used at Monash are Cisco IP phones).

A **server** is a device (usually a dedicated computer) that *provides services* to clients. For example, when you open the Monash homepage in a web browser, your client establishes a connection to the Monash web server computer, which sends the requested information (text, images etc.) back to you. In addition to sending information back to you, a server can also provide other types of services. A print server, for example, lets multiple users share a printer. A "smart" light bulb lets you turn it on and off via the network. An email server forwards messages to the specified recipients.

Clients and servers are connected with each other via the *circuit*, which we call all cables, radio links, and devices in between that enable the communication. Two types of devices in particular are essential in modern networks.

A **switch** connects multiple devices to form a *Local Area Network* (LAN). All devices in the *same* LAN can directly communicate with each other. A **router** connects *different* networks. If a device wants to communicate with another device that is outside of its own network, it has to send the messages via a router. In the diagram above, the WLAN access point on the left acts as both a switch (establishing a local area network) and a router (connecting it to the Internet). The LAN with the server is constructed using a separate switch and router.

Although we represent the Internet as an opaque cloud in the diagram, there's actually nothing special about it: it's just a collection of routers, switches, servers and clients. We say that the Internet is a *network of networks*, connecting millions of individual networks, which contain billions of devices. We will discuss the structure of the Internet in more detail in the module about The Internet.

Addresses

In the diagram above, the clients, the routers and the server are all annotated with their *IP addresses*. An address, in general, is a unique identifier. Remember how we use addresses to identify individual locations in memory (see Memory). In the same way, each device in a network needs a unique address to identify it as the receiver of a particular message. On the Internet, each device that needs to either send and receive messages (clients and servers), or forward them to other networks (routers), needs one IP address for each network that it is connected to. Routers therefore have at least two IP addresses. We will learn more about IP addresses in the module about the Network and Transport layers.

Something that isn't shown in the diagram is that each device in fact also has another address, the so-called *MAC address*. While the IP address is used to send messages from one network to another network, the MAC address is used within the same local area network. This will be explained in the module about the Data Link Layer.

Types of networks

As mentioned above, the Internet simply consists of millions of networks that are constructed using switches, routers, servers and clients. Of course it is not just an unstructured mess of cables and devices - that would have become impossible to manage as soon as the network grew beyond a few dozen computers. The Internet, and networks in general, are highly hierarchical in structure. The following types of networks are typically found *within a single organisation*, such as a single company or a university.
A **Local Area Network** (LAN) is a group of clients and/or servers that share a local circuit, i.e., they are

directly connected to each other using just switches and cables or radio waves. All devices in a LAN can communicate directly with each other (without going through a router). LANs are typically limited to a single building, or a single floor of a building, potentially even a single room. One example of a LAN would probably be the wireless network you use at home.

A **Backbone Network** (BN) connects multiple LANs using routers. Often BNs don't contain any clients or servers, but simply serve as the circuit that connects the different LANs. Backbone networks usually provide very high speed, because they need to be able to handle all the network traffic between the LANs. Backbone networks are usually still quite local. For example, at Monash, a BN would connect the different floors of a building, or the different building on a campus.

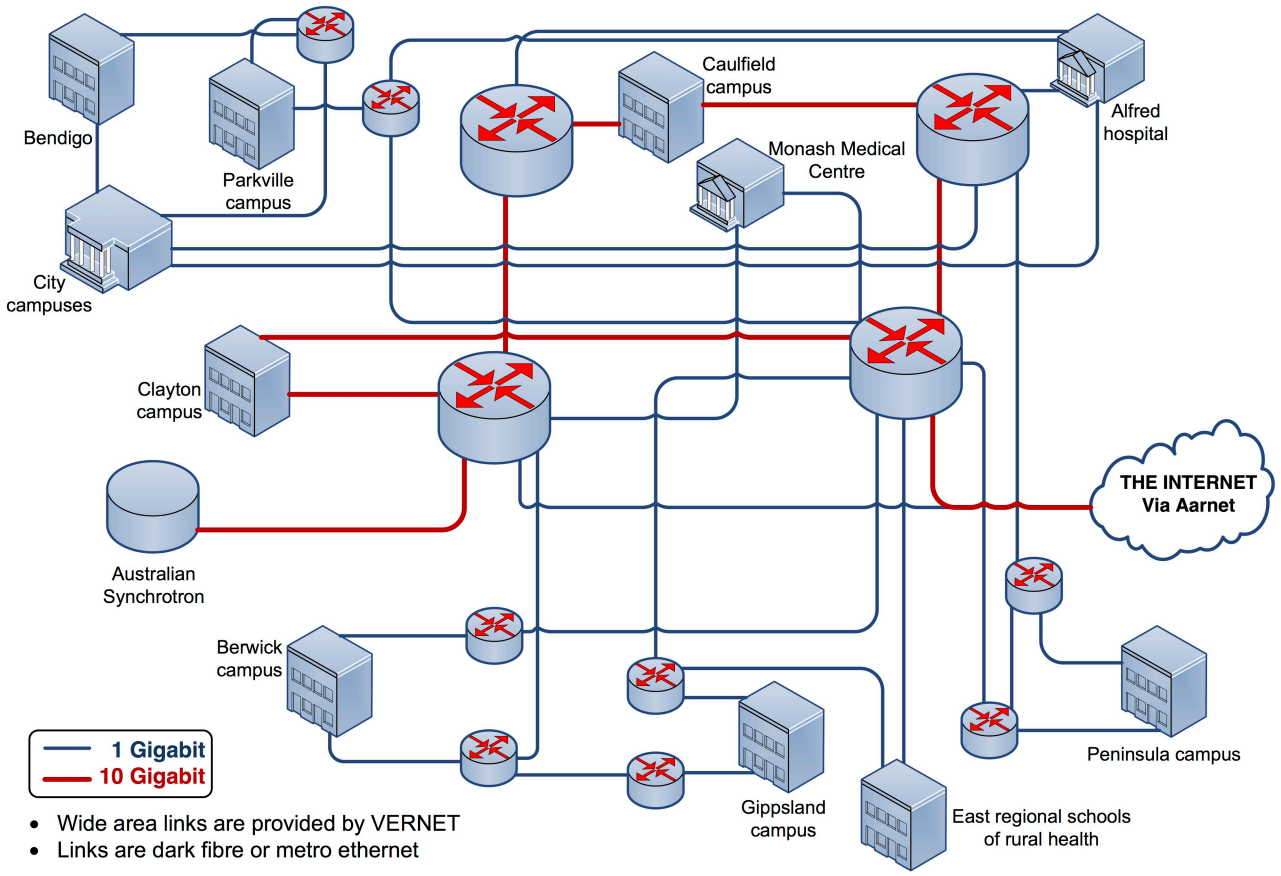
Both LANs and BNs are usually owned and operated by the organisation that uses them (e.g. Monash owns all the hardware including all the cables and devices for its LANs and BNs), and they don't require any licenses or permits to be built and operated. When we go to larger scales, this typically changes.

A **Metropolitan Area Network** (MAN) is the next larger scale network. It can span several kilometres, and connects LANs and BNs across different locations. For instance, the Monash Caulfield and Clayton campuses are connected via a MAN. A MAN is usually not built and owned by the organisation that uses it, but *leased* from a telecommunications company. This makes sense, because it would be prohibitively expensive to dig trenches across the city every time two offices want to get connected. A telecommunications company can operate its own network and then lease capacity to their clients.

Finally, a **Wide Area Network** (WAN) is very similar to a MAN except that it would connect networks over large distances. For example, if Monash had a direct connection between its Australian and Malaysian campuses, that would be considered a WAN. Just as with MANs, the actual circuits used for WANs are usually owned and operated by third-party companies who sell access to their networks.

An organisation would typically use multiple of these network types depending on its requirements. A small company may only operate a single LAN or maybe a couple of LANs and a BN, while larger organisations connect their LANs and BNs using MANs and/or WANs. Any such large network could be operated completely autonomously, i.e., without any connection to other networks outside of the organisation. But of course most companies would also connect their networks to the wider Internet (which will be covered in the module on The Internet).

Here's a diagram showing the structure of (part of) the Monash network.



You can see that the different campuses are connected using routers (since each campus operates its own LANs and BNs), and there is a connection to the Internet. The diagram also says that "*Wide area links are provided by VERNET*", which points to the fact that Monash doesn't own the cables that link different campuses (in our terminology this would include both MANs and WANs).

Transmission rates and latency

In the Monash network diagram above, different links are drawn in different colours and line thickness, and the legend states that those links are "1 Gigabit" (blue) and "10 Gigabit" (red). These numbers mean, in fact, gigabits per second, which is a common measure of the speed of a connection. We also call it the *transmission rate*, and it's one of the fundamental characteristics of a network.

Typical transmission rates are

- 1 Mbps (megabit per second, or million bits per second) from your home to your Internet Service Provider if you connect to the Internet using ADSL.
- 10-20 Mbps from your Internet service provider to your home is you connect using ADSL. Notice how these rates are *asymmetric*, your downloads are 10-20 times faster than your uploads. In fact the "A" in "ADSL" stands for asymmetric for exactly this reason.
- 50-500 Mbps within a wireless local area network (WLAN, sometimes also called WiFi). This means that if you e.g. use your home WiFi to copy a file from your laptop to your phone, the file will be transmitted at between 50 and 500 Mbps, depending on the concrete WiFi technology that you're using.
- 1 Gbps (gigabit per second, or billion bits per second) within a typical cable-based LAN, for example within a Monash computer lab.
- 10 Gbps in many backbone networks.

- Tbps (terabits per second) in the fastest networks that are currently in development.

Assume that you want to send a very high-definition movie (with 4k resolution) to a friend. The movie is 50 GByte of data. How long will it take using an ADSL connection? Reveal [1] How long if the destination is in the same WLAN with a 50 Mbps speed? Reveal [2] How long from one Monash lab computer to another one in the same room? Reveal [3]

Another fundamental characteristic of a network is *latency*, which describes how long it takes for one bit of data to travel from a sender to a receiver. A number of factors affect the latency of a network. The most fundamental one is the signal speed, meaning for example how fast a signal can travel in a copper cable or as a radio wave. This is limited by the speed of light (roughly 300,000 km/s), no message can travel faster than that, but usually messages in cables are slower (we can assume roughly 200,000 km/s in a copper cable, for example). Then there are all the devices that messages pass through on their way from the sender to the receiver. Each switch and router takes a short amount of time to process the message and decide what to do with it (we'll see this in detail later). All of this adds up to a measurable delay, especially across long distances such as between continents.

A good example of latency versus transmission rate is communication with a spacecraft. Let's assume that humanity has sent a few astronauts on a trip to Mars. They have entered orbit around the red planet after six months of interplanetary travel. Of course on such a long mission, we have to keep them entertained, so we provide them with the fastest interplanetary Internet connection we can afford: a 400 Mbps radio link! That's a super fast connection, and our astronauts will be happily streaming the newest episodes of their favourite shows in very high definition, because for that we only need to send around 3-5 Mbps. However, let's now assume they want to switch to a different show. When they click on the "next" icon, their web browser sends a tiny message to their streaming video provider with the command to start streaming the new episode. This message will take between 4 minutes and 24 minutes to reach the server (depending on the current distance between Mars and Earth), and if the server immediately starts sending the data for the new show, it will again take between 4 and 24 minutes for the data to arrive at the spacecraft! Similarly, a video conference with their families can transmit beautiful high-definition video (because the transmission rates are high enough), but it would only arrive with a delay of at least 4 minutes (due to latency), making any normal conversation impossible.

This is an extreme example of latency, but even on our own planet, typical delays between Australia and Europe are around 0.15 seconds, or 0.07 seconds between Australia and North America. Assuming a distance of 15,000 km between Australia and Europe, how much of the latency is (approximately) due to the speed of light? Reveal [4] The rest is due to processing times in switches and routers.

Network application architectures

As mentioned in the introduction to this module, networks enable computers to communicate. In most cases, a client will communicate with a server, and together they provide an *application* to the user. For example, you can consider Moodle to be an application that is provided by the combination of a web server and a web browser, or a video chat application that is provided by an app on your smartphone, a server on the Internet, and an app on the phone of the person you're calling.

Each application has different tasks to fulfil, and when implementing an application, we have to decide who performs which task: the server or the client(s)? A standard way of describing the different tasks is the following.

The **presentation logic** is the part of the application that provides the *user interface*, i.e., the elements such as menus, windows, buttons etc. that the end user interacts with.
The **application logic** or **business logic** defines how the application *behaves*, i.e., it implements what

should happen when the user clicks a button or receives a new message or types a word.

The **data access logic** defines how the application *manages its data*, it is responsible e.g. for updating text documents when the user makes changes, or retrieving a piece of information when the user performs a search.

The **data storage** is where the data is kept, e.g. in the form of files on a disk.

In a "traditional" application that doesn't use any networking, all four tasks would be performed on the same computer. But in a networked application, we can split up the work between the client and the server. Depending on how we split the work, we can define different **application architectures**.

In a **server-based** architecture, all four tasks are performed by the server. The client is just a "dumb terminal", which sends individual keystrokes back to the server and displays text-based output from the server. This was a popular architecture in the 1960s and 1970s, because it enabled multiple users to access a single large, expensive "mainframe" computer.

DEC VT100 terminal as used in server-based architectures. Image by Jason Scott (Flickr: IMG_9976) [CC BY 2.0], via Wikimedia Commons

Server-based architectures have the advantage that any update to the software on the server is immediately available to all users (since no software is actually running on the clients). However, upgrading the hardware means replacing a large, expensive computer rather than inexpensive terminals. The second architecture we're going to look at places only the data storage on the server, while presentation, application and data access logic are performed by the client. This is called a **client-based** architecture, and it was developed in the late 1980s. The typical example for this type of architecture is a *file server*. For example, when you log into any lab computer at Monash, your own files are available over the network. This means that they are not stored on the local hard disk of that computer, but rather on a Monash file server. Now let's say you open a text document in Microsoft Word to make some changes. The entire document file will be transferred from the server to your client. Microsoft Word implements all the presentation, application and data access logic. When you save the file, it is sent back over the network to the file server.

Client-based architectures became popular because they enabled companies to have a central file storage facility, enabling multiple users to work on the same files together, and providing a central back-up mechanism. However, client-based architectures also have some disadvantages. Imagine searching for

the phone number of your lecturer in the Monash directory. In a client-based architecture, you would have to download the entire directory over the network to your client, just to search for a single entry. Even worse, if you were accessing some database to make small changes, for every change you would need to transmit the whole database. This would put enormous stress on the network, but also cause big problems if multiple users want to access the same database simultaneously.

These issues can be mitigated by using **client-server** architectures. Here, the client only performs the presentation and application logic, while the server implements both data access and storage. The typical example are database servers, which can process *queries*, such as searches or requests for modification of individual records, and send only the relevant results back to the clients. We can consider typical email systems as client-server, where emails are stored on a server and clients access only those that the user is currently interested in.

If you've kept track, you will have noticed that there is one combination left that we haven't discussed yet. If the client performs only the presentation logic, while the server implements application and data access logic and data storage, we are looking at a **thin-client** architecture. You have all used this kind of architecture, in fact you are probably using it while reading this text: most web applications can be considered thin-client. The web browser only "renders" the page on the user's screen, but any interaction (e.g. clicking a button) is sent back to the web server, which implements the application logic and sends back the results for the browser to display. Similarly, many smartphone apps only work while you are connected to the Internet, because parts of their application logic are performed by remote servers rather than on your phone. A good example is also "digital assistants", like Microsoft Cortana, Google Now, Amazon Echo, or Apple's Siri. When you ask them a question, that question is sent over the network to a server, which does all the complex processing to analyse your request and come up with an answer. The result is sent back to your phone, which implements the presentation logic by turning the result into a synthesised voice, or displaying it on its screen.

In many client-server and thin-client architectures, more than one server is required to handle the demands of potentially millions of users. Usually, this means that the tasks are further split, with dedicated servers for the application logic, and dedicated database servers handling the data access and storage. We call this a **multi-tier** architecture.

Finally, there is an architecture that doesn't use servers at all. Some applications directly connect multiple clients with each other, with each client implementing all aspects of the application. This is called a **peer-to-peer** architecture. Prominent use cases for peer-to-peer systems are distributed file sharing as well as some audio and video conferencing services.

↩1 50 GByte = 400 Gbit = 400,000 Mbit, at 1 Mbps that's roughly 4 days, 15 hours

↩2 2 hours 13 minutes

↩3 400 Gbit / 1 Gbit / s = 6 minutes 40 seconds

↩4 15,000 km / 300,000 km/s = 0.05 s

7.2
Layers and Protocols

The point we want to bring across in this module is **how to transfer data** in a network that is made up of millions of devices, made by thousands of different hardware manufacturers, running very different software.

Of course, this can only work in the presence of international, open standards that prescribe how devices "talk to each other". But standardisation is not enough. The web browser on your phone may know how to "talk to" any web server in the world. But does it also need to know how to access both the 4G mobile network and WiFi, depending on what you're using at any one time? These two technologies are very different and require quite different software to run, but of course the software engineers that develop your browser have enough on their hands just dealing with modern web technology, and shouldn't have to worry about every possible piece of network hardware that their browsers may have to use.

We've seen in the module on Operating Systems that the usual answer to these problems is *virtualisation*. The OS provides a layer of abstraction above the hardware, and the application software just accesses the OS (through systems calls), which contains drivers that are tailored to the concrete hardware that's present in your computer.

The same principle is used for implementing complex computer networks. We define a hierarchy of layers of abstraction, each with a specific set of tasks and a well-defined interface to the other layers. In addition, each layer defines a "language", which we call a **protocol**, that prescribes how the software implementing the same layer on different devices can interact.

The Internet Model

In the remainder of this module, we will describe what is known as the **Internet Model** of networking. The first thing to note about the Internet Model is that it describes a **packet switching** network. That means that any communication between devices over the network happens in the form of (relatively) short packets, which are sent ("switched") across multiple intermediate points, which on the Internet are called routers. Since each packet can only contain a limited amount of data (typically in the order of 1500 bytes), larger data needs to be split up by the sender into individual packets, and reassembled by the receiver.

Based on this idea of packet-switching, the Internet Model defines five different layers of abstraction. From the bottom up, they are the hardware, data link, network, transport and application layers.

The **hardware layer** (layer 1, also known as the **physical layer**) is concerned with the actual hardware, such as cables, plugs and sockets, antennas, etc. It also specifies the *signals* that are transmitted over cables or radio waves, i.e., how the sequence of bits that make up a packet is converted into electrical or optical wave forms.

The **data link layer** (layer 2) defines the interface between hardware and software. It specifies how devices within one local area network, e.g. those connected directly via cables or radio waves to a switch, can exchange packets.

The **network layer** (layer 3) is responsible for *routing*, i.e., deciding which path a packet takes through

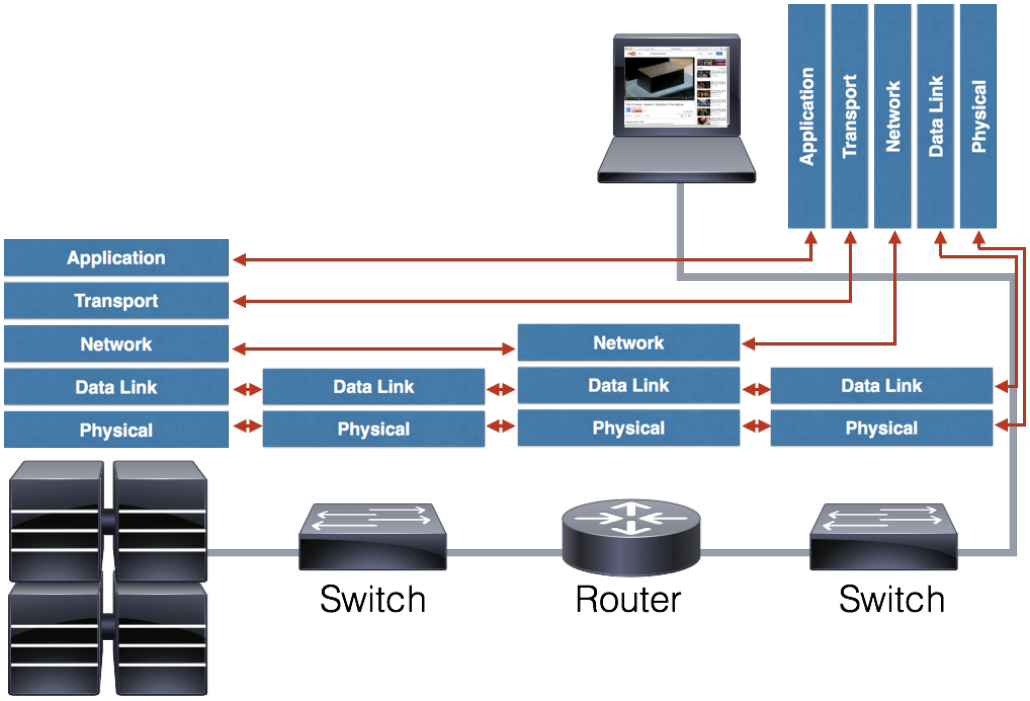
the network. There are often multiple possible paths, and each individual packet may take a different path from source to destination. The network layer is of course most important for routers, which are the network devices whose primary task it is to perform routing. But every client and server also has network layer software and performs routing, because it may be connected to more than one network (e.g. 4G and WiFi).

The **transport layer** (layer 4) establishes a *logical connection* between an application sending a message and the receiving application. It takes care of breaking up a large message into individual packets and reassembles them at the receiving side. It also makes sure that messages are received correctly, re-sending packets if they were received with errors (or not received at all). The transport layer therefore performs the main form of virtualisation that makes implementing networked applications much easier: From an application point of view, we can open a connection to a remote server, send a long message, and receive a reply, without worrying about anything that goes on at the lower-level layers.

The **application layer** (layer 5), finally, is the actual application software that a user interacts with. For example, your web browser or your instant messaging app are implemented at the application layer.

Different devices on the network implement a different subset of these layers. A switch only needs to implement the hardware and data link layers, because it is only responsible for communication inside a local network. It ignores the higher level layers. Similarly, a router requires an implementation of the network layer in order to perform its function, but it can ignore the transport and application layer. Servers and clients implement all five layers.

The diagram below illustrates how different devices communicate at the different layers.



The server in the bottom left corner is connected to a switch (at the physical or hardware layer), and its data link layer software communicates with the data link layer software in the switch. There's also a router connected to the switch, which means that the router and the server are part of the same LAN. The network layer software on the server can communicate with the router's network layer. This happens, e.g., when an application on the server sends a message to the client in the top right corner (which is also connected to the same router via another switch). Since the client is not part of the same LAN, the

message must be sent via the router. But as you can see, the network layer software handled the delivery via the router, while the transport and application layer software can virtually communicate with the client directly.

The big advantage of this approach is that applications are completely independent of the underlying network hardware. For example, the server could be connected to the switch via a copper cable, the switch to the router via an optical fibre, and the client could be using a wireless technology such as 4G or WiFi.

Now that we've established how to split up the work by defining layers of abstraction, the next thing to look at is how we can make sure that the software for each layer on one device can talk to its counterpart on a different device.

Protocols and Message Encapsulation

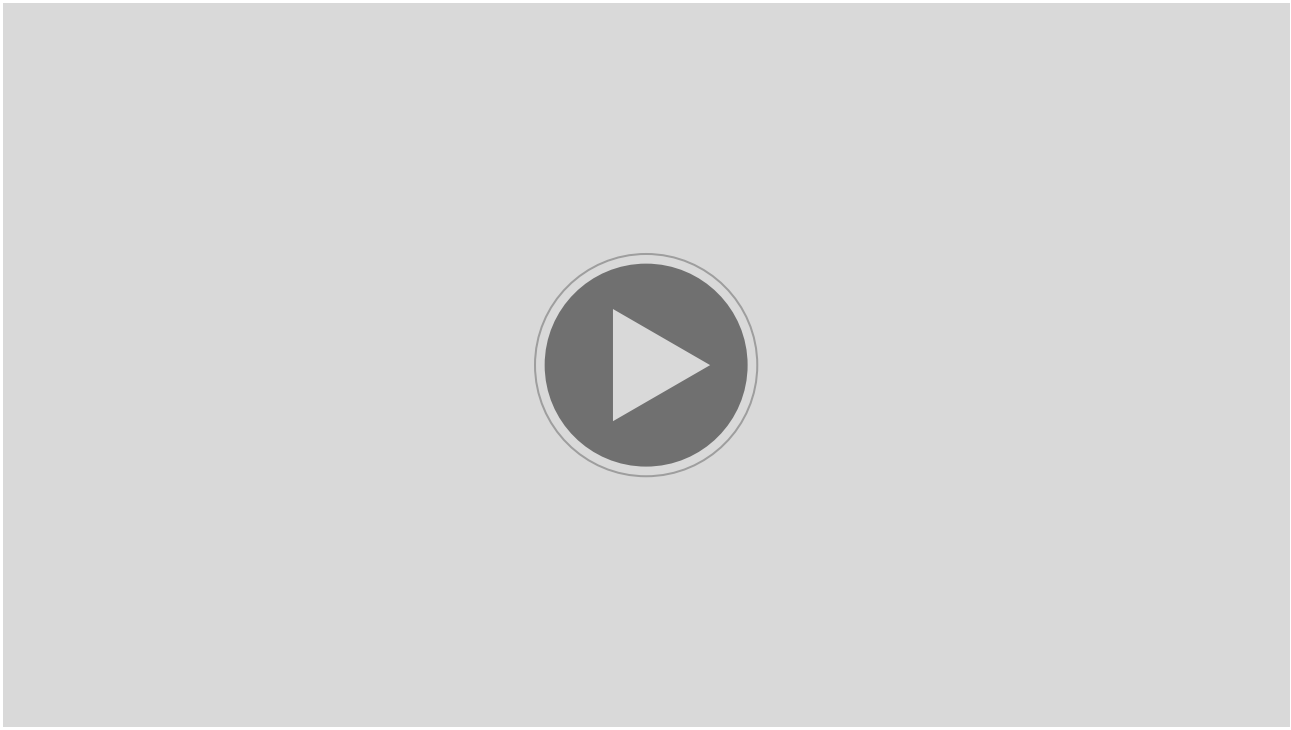
A **protocol** is a formal language that defines how two applications talk to each other. In the case of the "middle layers" of the Internet Model (i.e., data link, network and transport), the protocols take the form of well-defined *headers* of data that are added to the actual message data by the sender, containing information such as sender and receiver addresses, the type of content of the message, and error detection codes like a CRC (see Representing numbers).

The easiest analogy is that of an *envelope*. The actual message is put inside an envelope, and the sender and receiver addresses are written on the outside of the envelope. In addition, each layer may add some additional information. For example, if the transport layer had to split up a large message into multiple packets, it would write something like "packet 12 of 134" on the envelope as well, so that the receiver can puzzle them back together in the right order.

Compared to the regular postal service, each layer adds its own envelope, and puts the envelope for the layers above inside. So the actual packet that is being transmitted by the hardware would be a data link layer envelope that contains a network layer envelope that contains a transport layer envelope that contains (part of) the application layer message! This is called **message encapsulation**.

Now you can begin to understand how e.g. a switch works: it receives a packet, and it only needs to look at the outer-most envelope (which has the data link layer address of the destination) to deliver the message. A router already needs to take a look inside that envelope, in order to find out the network layer destination address. It will then create a new envelope, with the next data link layer destination address that the packet should be sent to (which may be another router!). But neither a switch nor a router will ever look into the network layer envelope (and they don't need to understand its contents). The transport layer software on a client or server will use the information on the transport layer envelope to reassemble the entire message, before handing it over to the application layer software.

Here's a video that explains encapsulation (and the basics of routing).



(https://www.alexandriarepository.org/wp-content/uploads/20170421124558/routing_extended_720.mp4.mp4)

These "envelopes" are called **protocol data units** (PDU), and the PDU for each layer has its own name. At the hardware layer, the PDU is simply a bit. At the data link layer, the PDU is called a *frame*. The network layer PDU is called a *packet*, and the transport layer PDU is a *segment* or a *datagram* (depending on the concrete protocol used). At the application layer, we generally talk about *messages*.

7.3 Application Layer

Software at the application layer implements the functionality of a networked application that users are actually interested in, such as web browsing, email, video chats etc. The different application architectures (client-server, thin client etc.) are implemented at this layer. The application layer builds on top of all the other layers, which means that it can take full advantage of the virtualisation of the network. Applications can simply open a connection to an application on another computer, and then start sending and receiving data. That's why we'll look at the application layer first, before explaining how the lower-level layers work in later modules.

This module covers two concrete applications, which are the most widely used on the Internet: the World Wide Web, and electronic mail.

The World Wide Web

The World Wide Web, or WWW, is what you access through a web browser. Its defining features are *web pages* that can be accessed at particular *URLs*, and that can *link* to other web pages.

The WWW was invented by Tim Berners-Lee at CERN, the *European Organization for Nuclear Research.* Faced with the problem of organising the immense amount of documents and data generated at CERN, Berners-Lee proposed to develop a system based on *HyperText,* which is the now very familiar idea of links embedded in pages, which lead to other pages. Here is how he motivated the project in 1989 (the full proposal can, of course, be found on the web (http://info.cern.ch/hypertext/WWW/Proposal.html)):

At CERN, a variety of data is already available: reports, experiment data, personnel data, electronic mail address lists, computer documentation, experiment documentation, and many other sets of data are spinning around on computer discs continuously. It is however impossible to "jump" from one set to another in an automatic way [...]

The first web server at CERN. Image by Coolcaesar (CC-BY-SA-3.0), via Wikimedia Commons

Berners-Lee also developed the first web server and browser. The concepts he developed have evolved into the protocols and languages that still define the web today. In order to understand how the web works, we need to look at (at least) three technologies: URLs, HTTP and HTML.

A *Uniform Resource Locator* (URL) is the address of a *document* on the world wide web. If you're reading this in a web browser, the address bar at the top of the page will contain the URL for this document. A document can be a web page, but it can also be an image, a video, a program written in JavaScript, or an app that you want to download to your computer. The *Hypertext Transfer Protocol* (HTTP) is a standard set of commands that is understood by all web browsers and web servers. A browser can send an HTTP command to a web server to request a certain document, based on the document's URL, and the server will use HTTP to

send the document back. The *Hypertext Markup Language* HTML, finally, is the document format for web pages, i.e., it is a way of describing the contents and layout that browsers should display.

URLs, HTML and HTTP

Uniform Resource Locators

A URL is a textual address that uniquely identifies where to find a particular document on the Internet, and how to retrieve it. For example, the following URL identifies this document:

https://www.alexandriarepository.org/module/application-layer/

The URL above has three components:

- The *scheme* describes which protocol must be used to retrieve the document. In this case, the scheme is `https`, which is a secure version of HTTP (discussed in more detail in Security).
- The *host* identifies the server. In the URL above, the host is `www.alexandriarepository.org`.
- The *path* identifies a particular document on the server. The name of the document here is `/module/application-layer/`

So the above URL, when typed into a web browser, instructs the browser to use the HTTPS protocol, contact the server `www.alexandriarepository.org`, and request the document called `module/application-layer/`. You have probably noticed that the path of a URL is often empty (e.g. `http://www.monash.edu/`), and sometimes it can include more information, such as a search query:

https://www.google.com.au/search?q=FIT1047

HTML documents

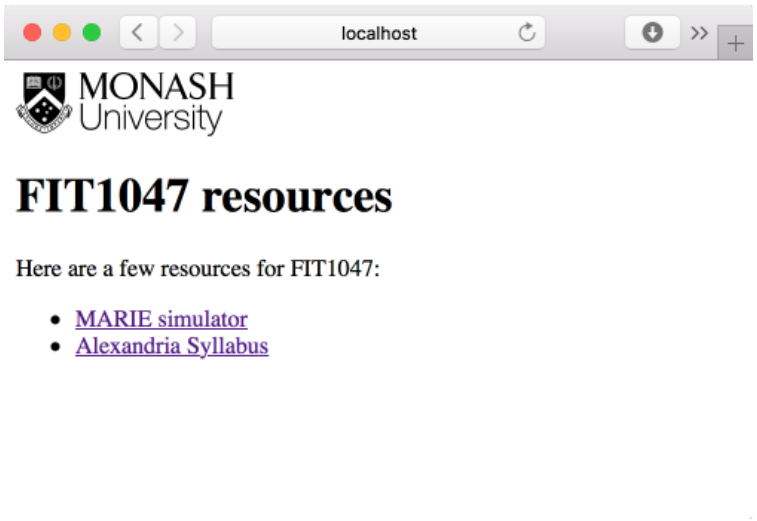
The earliest web pages contained just plain text and links to other pages. Modern web pages are of course often results of very careful design, using many graphical elements, different fonts, and media such as audio and video. But at its core, a web page is still represented by a document written in the Hypertext Markup Language HTML.

Such a document is a plain text file that has been annotated with *tags* that describe the purpose of different elements in the file. Here's an example:

```
<html>
  <body>
  <img
src="http://www.monash.edu/__data/assets/git_bridge/0006/509343/deploy/mysource_
files/monash-logo-mono.svg">
    <h1>FIT1047 resources</h1>
    <p>Here are a few resources for FIT1047:</p>
    <ul>
      <li><a href="https://mariejs.xyz">MARIE simulator</a></li>
      <li><a
href="https://www.alexandriarepository.org/syllabus/introduction-to-computer-sys
```

```
tems-networks-and-security/">Alexandria Syllabus</a></li>
    </ul>
  </body>
</html>
```

The resulting document may look like this when rendered by a web browser:



The text between < and > brackets is called a *tag,* it is not displayed but rather tells the browser something about the structure of the document. For example, anything between the <h1> and </h1> tags is a level 1 heading, while ... denotes a paragraph. The <a> tag is particularly relevant: it introduces an "anchor", meaning that it links to another document, which is specified as its `href` (hypertext reference) attribute. The text between the <a> and is used as the actual link text displayed by the browser. The `<img...>` tag inserts an image into the page. Note how the document does not *contain* the image, it merely *refers* to it via a URL.

The HTTP request-response cycle

The final piece of the puzzle is the Hypertext Transfer Protocol. If defines how a web browser can request a document from a web server, and how the web server responds (delivering the document, or perhaps sending an error message if something went wrong).

HTTP operates in a so-called *request-response cycle*. Let's use the simple web page above as an example for what that means. The user enters the URL of the page into their web browser (or clicks on a link that points to that page). For simplicity, let's assume the URL is `http://www.monash.edu/fit1047.html` (this one doesn't really work).

The browser notices that the URL starts with `http`, and that the host is `www.monash.edu`, so it starts the process by opening a connection to the HTTP server `www.monash.edu`, and then sending a request for the document `fit1047.html`. In HTTP, this request is written as follows:

```
GET /fit1047.html HTTP/1.1
Host: www.monash.edu
```

So our web browser will send a message containing these two lines to the web server. The server will then process the request and return its *response*, which may look like this:

```
HTTP/1.0 200 OK
Date: Fri, 21 Apr 2017 05:30:10 GMT
Content-type: text/html
Content-Length: 463
Last-Modified: Fri, 21 Apr 2017 04:39:52 GMT

<html>
 <body>
 <img
src="http://www.monash.edu/__data/assets/git_bridge/0006/509343/deploy/mysource_
files/monash-logo-mono.svg">
 <h1>FIT1047 resources</h1>
 <p>Here are a few resources for FIT1047:</p>
 <ul>
 <li><a href="https://mariejs.xyz">MARIE simulator</a></li>
 <li><a
href="https://www.alexandriarepository.org/syllabus/introduction-to-computer-sys
tems-networks-and-security/">Alexandria Syllabus</a></li>
 </ul>
 </body>
</html>
```

You will recognise the actual HTML document in the second half of the response, but the server sends a few additional lines. We will discuss them below, but let's first see what happens next. The web browser analyses the content of the HTML document to find out if it needs any other documents in order to render it. It will discover the <img> tag, so it has to make another request to the server, this time for the image:

```
GET /__data/assets/git_bridge/0006/509343/deploy/mysource_files/monash-logo-
mono.svg HTTP/1.1
Host: www.monash.edu
```

The server will now respond by sending back the image file. This cycle continues until all *assets*, i.e., all documents that are required to render the page, have been loaded. Note that the documents linked to from <a> tags are *not* requested immediately, because they are only required if the user actually clicks on that link.

HTTP methods, requests and responses

In addition to the GET command introduced above, HTTP supports a few additional commands. Commands are called *methods* in HTTP. The most important method, apart from GET, is POST, which allows the browser to *send* a document to the server (rather than requesting a document from the server). The use case is for example posting a message to a news forum, or uploading a file. Other methods are used less often.

Let's now look at the full structure of HTTP request and response messages.

A **request** consists of the *request line*, which includes the method, a path, and the protocol version that is to be used (`HTTP/1.1` in our example). The request line is followed by the *request header*, which must always include the `Host: ...` line, indicating the name of the host, but it can include other lines specifying additional information. For example, the web browser can request that the browser only return certain types of files, or only files newer than a certain date, or only documents written in a particular language. The request header needs to be followed by a blank line. If the request requires sending a document to the server (as in the case of a POST request), then the document is sent as the so-called

request body after the header.

A **response** has a similar structure. The first line is always the *response status*. In the example above, the server replied with `HTTP/1.0 200 OK`. This identifies the protocol version that the server uses, and two status codes, `200` and `OK`. They both mean the same thing: the numerical version is easy to interpret for the browser, the text version is easy to read for humans. Other potential status codes include `404 Not found` if the server can't find the requested document, or `403 Forbidden` if the browser doesn't have the right permissions to access the document (e.g. you need to log in first).

The response status is followed by the *response header*. In the header, the web server can provide some *metadata,* i.e., additional information about the document that could be useful for the browser. In our example above, the server lets the browser know that the document is in fact an HTML document (using the `Content-type` header), and when it was last modified. Just as for the request header, the response header ends with a blank line, followed by the *response body* that contains the document.

State in HTTP

One important thing to understand about HTTP is that it is *stateless*. This means that every request is sent independently, and the server doesn't have any idea whether two consecutive requests are from the same user or from different users.

Why is that a problem? The prime example is an online shop. You browse the web pages with lists of products, and when you find something you like you click the "Add to my shopping cart" button. Then you continue browsing. Of course, when you later click on the shopping cart, or add another item, you want the cart to still contain the first item you put in. We call the continuing interactions of a user with the same web site a *session*, and all the data that needs to be kept consistent within a session (such as a shopping cart) the *state* of the session.

But how does the server know that you've put an item in the shopping cart, if it doesn't even know that a request a minute later is in fact from you? How can it keep track of the different session states of thousands of people using the web site at the same time?

Since HTTP is a stateless protocol, we have to keep information about the session state somewhere else, and transmit it as part of the requests and responses. This is usually implemented using a *session ID*, a long, unique number that the web server generates when a user makes their first request. The user's browser is then instructed to send that session ID as part of every single future request. That way, the server can identify the user, and send the correct shopping cart data (from its database) back in the response.

There are two tricks that servers and browsers use to exchange session IDs. The first one is quite simple: The web server returns the document with the web page, but every link in that document (inside every <a> tag) is extended by the session ID. For example, if the session ID was `NKH5WJWQB4rr8wfxtaDIHQ`, then the server might return an HTML document that looks like this:

```
<html>
 <body>
<h1>FIT1047 resources</h1>
 <p>Here are a few resources for FIT1047:</p>
 <ul>
 <li><a href="https://mariejs.xyz/?id=NKH5WJWQB4rr8wfxtaDIHQ">MARIE
simulator</a></li>
```

```
 <li><a
href="https://www.alexandriarepository.org/syllabus/introduction-to-computer-sys
tems-networks-and-security/?id=NKH5WJWQB4rr8wfxtaDIHQ">Alexandria
Syllabus</a></li>
 </ul>
 </body>
</html>
```

As you can see, the links now include the session ID, so whenever the user click on a link, the ID will be sent as part of the request! The disadvantage of this approach is that it doesn't work if the user closes the browser window and opens a new window for the same web site (because then the server will generate a new session ID).

The alternative to embedding session IDs in URLs is to use **cookies**. A cookie is a a small piece of data that is returned as part of the HTTP response header. For example, the server may return the following header:

```
HTTP/1.0 200 OK
Date: Fri, 21 Apr 2017 05:30:10 GMT
Content-type: text/html
Set-cookie: sessionid=NKH5WJWQB4rr8wfxtaDIHQ
Content-Length: 463
Last-Modified: Fri, 21 Apr 2017 04:39:52 GMT
```

The web browser stores the cookie, and every time it makes a request to that web server, it will send the cookie back to the server as part of the request header:

```
GET /fit1047.html HTTP/1.1
Host: www.monash.edu
Cookie: sessionid=NKH5WJWQB4rr8wfxtaDIHQ
```

Cookies are extremely useful, and most web sites nowadays rely on cookies to implement session management. But at the same time, cookies can be used to track user behaviour, even across multiple web sites.

For example, let's assume you are a member of a popular online community where people share important information on where they had coffee etc. When you first logged in, you entered your personal data, and the web site sent a cookie to your browser, so that you don't have to enter your password every time you visit the site. Another service offered by this site is that a lot of other web sites, such as news and shopping sites, add a "Great!" button to every article, so that users can share the fact that they find the content "Great!" with their friends.

Obviously, if you click the "Great!" button next to a picture of a particularly smooth flat white, you contact your community web site, and it will know that you find that particular picture great. So it can learn more about you, and potentially use that to place ads, or even sell your profile information to other companies.

But using cookies, the community web site can even track your behaviour *without* you clicking on the button! Let's assume the button is an image like this:  . The actual image file is stored on the community web site, so the news site you're browsing just contains an `<img>` tag that points to that image. As soon as your browser loads the image, it sends the cookie that was stored before, and the community web site knows you've been looking at a particular news article. That way, they can build a complete profile of your web browsing behaviour.

Web browsers allow you to restrict what kind of cookies they store and send back, so perhaps you may want to explore the privacy settings of the browser you're using.

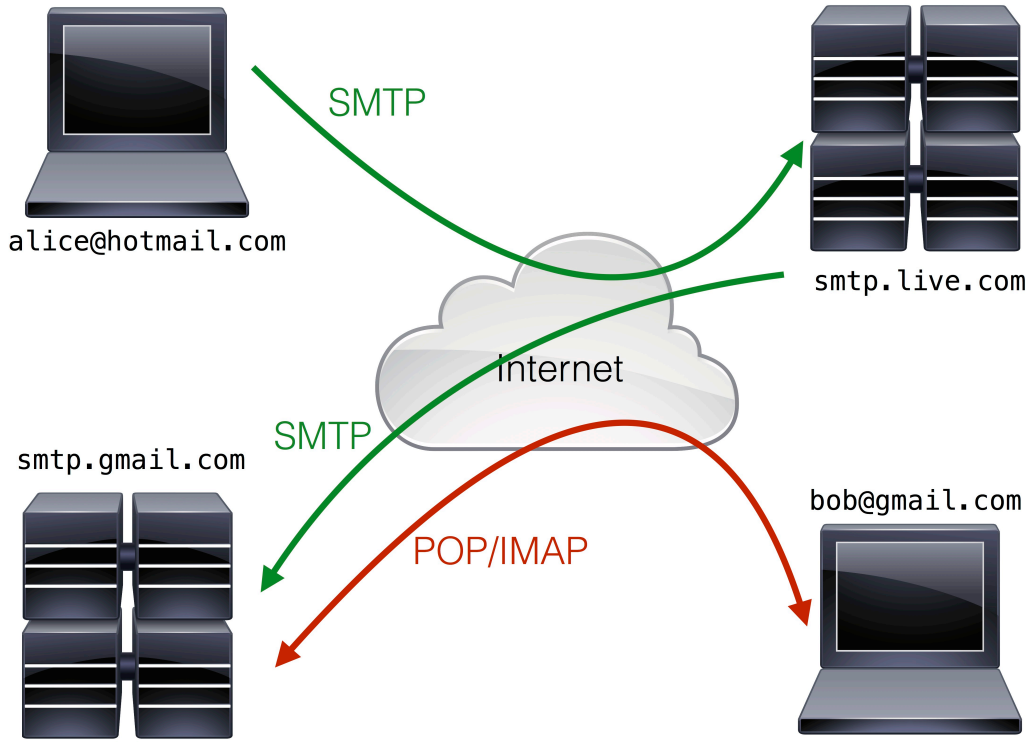
Electronic Mail

Email is still one of the most widely used applications on the Internet. It is estimated that over 200 billion emails are sent every day (with a significant fraction, estimated between 50% and 70%, being spam). The basic protocols for email were defined in the early 1980s, and they are still in use (more or less unchanged).

The client-server approach to email

The traditional email setup is a classic two-tier client-server application. A sender uses the *Simple Mail Transfer Protocol* (SMTP) to send a message to a mail server, which forwards the message to the recipient's mail server, also using SMTP. The recipient uses either the *Post Office Protocol* (POP) or the *Internet Message Access Protocol* (IMAP) to access their emails on the server.

Here's a diagram showing an email exchange from Alice to Bob.



Why are we using different protocols for sending and receiving email? The main reason is that we assume that servers are available 24/7, while clients such as Bob's laptop are not online all the time, and Bob may want to access his emails from different clients such as his laptop and his phone. So we can always send an email to the server using SMTP. But rather than `smtp.gmail.com` "pushing" the new emails to Bob's client, Bob "pulls" them from the server on demand.

The difference between POP and IMAP is mainly that POP downloads messages onto the client, and usually deletes them from the server in the process. With IMAP, messages remain on the server, and

multiple clients can access the same email account at the same time, marking messages as read or unread, and synchronising the state of the "inbox" across all devices.

SMTP

An SMTP session is very similar to the HTTP session we have seen above. The following messages might be exchanged between a client and an SMTP server:

```
220 MyMailServer ESMTP Exim 4.82 Ubuntu Sat, 07 Mar 2015 20:37:24 +1100
HELO my.laptop
250 MyMailServer Hello laptop [192.168.1.5]
MAIL FROM:<alice@mymail.com>
250 OK
RCPT TO:<bob.builder@monash.edu>
250 Accepted
DATA
354 Enter message, ending with "." on a line by itself
From: "Alice" <alice@mymail.com>
To: "Bob T. Builder" <bob.builder@monash.edu>
Date: Fri, 21 Apr 2017 19:24:00 +1000
Subject: test message

Hi Bob!
This is just a test.

Cheers,
Alice

.
250 OK id=1YUBBf-0003Ch-M1
QUIT
221 MyMailServer closing connection
```

The highlighted red lines are sent by the client, the black lines are sent by the server. The protocol starts by the server greeting the client (line 1), and the client greeting back using its name (`my.laptop`, line 2). Then the client specifies the sender (line 4) and recipient (line 6) addresses. It could specify more than one recipient. To tell the server that the client is now ready to send the actual text of the email, it sends the message DATA (line 8). The server instructs the client to enter the message, ending it with a "." on a line by itself, which the client does (lines 10-21). The server acknowledges the message (line 22) before the client tells the server that it's finished (line 23) and the server closes the connection (line 24).

The actual message sent by the client is structured a bit like an HTTP response, with a message header (lines 9-13) and a message body (lines 15-20). The header repeats the information about sender and receiver addresses, and this is the information that the recipient's mail software is actually going to display. It also contains metadata like the date and time the message was sent, and the subject line.

Beyond plain text

As you have seen above, both HTTP and SMTP are plain text protocols, i.e., they were designed for sending textual data (perhaps "enriched" a bit using tags in HTML). But we already saw above that we can request an image over HTTP, and you've certainly seen emails that contain images or even videos, and multiple attached documents.

One way to solve this problem would have been to redesign the protocols so that they can deal with arbitrary binary data (such as images, videos and PDF documents) rather than just ASCII text. But that would have meant changing a lot of existing infrastructure. So another solution was found: *encode* arbitrary binary data into ASCII text.

The standard way of doing that is the MIME format (Multi-purpose Internet Mail Extensions). It can encode any binary data into plain text (and decode it back), and it specifies how to send emails that contain multiple documents. Here is the beginning of a MIME-encoded image:

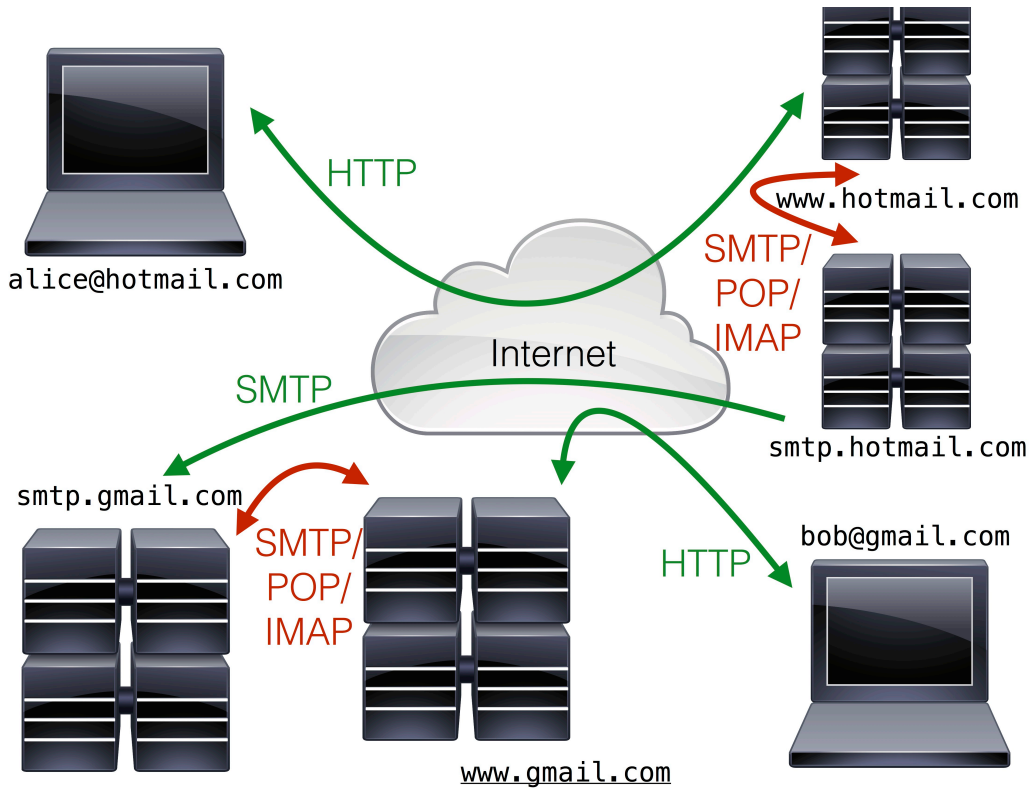
```
Content-Transfer-Encoding: base64
Content-Disposition: inline;
 filename=image.jpg
Content-Type: image/jpeg;
 name="image.jpg"


/9j/4AAQSkZJRgABAQEASABIAAD/4gVASUNDX1BST0ZJTEUAAQEAAAUwYXBwbAIgAABtbnRyUkdC
IFhZWiAH2QACABkACwAaAAthY3NwQVBQTAAAAABhcHBsAAAAAAAAAAAAAAAAAAAAAAAA9tYAAQAA
AADTLWFwcGwAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAtk
c2NtAAABCAAAAvJkZXNjAAAD/AAAAG9nWFlaAAAEbAAAABR3dHB0AAAEgAAAABRyWFlaAAAElAAA
ABRiWFlaAAAEqAAAABRyVFJDAAAEvAAAAA5jcHJ0AAAEzAAAADhjaGFkAAAFBAAAACxnVFJDAAAE
vAAAAA5iVFJDAAAEvAAAAA5tbHVjAAAAAAAAABEAAAAMZW5VUwAAACYAAAJ+ZXNFUwAAACYAAAGC
ZGFESwAAAC4AAAHqZGVERQAAACwAAAGoZmlGSQAAACgAAADcZnJGVQAAACgAAAEqaXRJVAAAACgA
AAJWbmxOTAAAACgAAAIYbmJOTwAAACYAAAEEcHRCUgAAACYAAAGCc3ZTRQAAACYAAAEEamFKUAAA
```

The key point is the first header line. It specifies "base64" as the encoding type. Here's how it works in principle. We are going to pick 64 ASCII characters that are easy to print (so SMTP and HTTP can transmit them without problems): A-Z, a-z, 0-9, + and /. Since we can represent the numbers from 0 to 63 using six bits, each of these characters now stands for a six-bit binary number between 000000 and 111111, with A being 000000 and / being 111111. Now all we have to do is break up our binary data into six-bit blocks, and encode each of those blocks into one of our characters. In the end, each sequence of three bytes (24 bits) in the original binary data will be encoded into four characters.

Web mail

The traditional email setup as a client-server architecture, where users access their email using dedicated applications, has been augmented and in many cases replaced by *web mail services*, where users access their email through a web page. One example you are probably familiar with is of course the Google mail service used by Monash University.

A web mail service can be seen as a classic example of a multi-tier thin client architecture. Conceptually, web mail providers simply add a web server that implements the email web application, but internally, it then uses SMTP and IMAP or POP to deliver and receive emails using the existing mail servers. Here's an overview picture of how Alice and Bob could exchange emails using a web mail architecture:



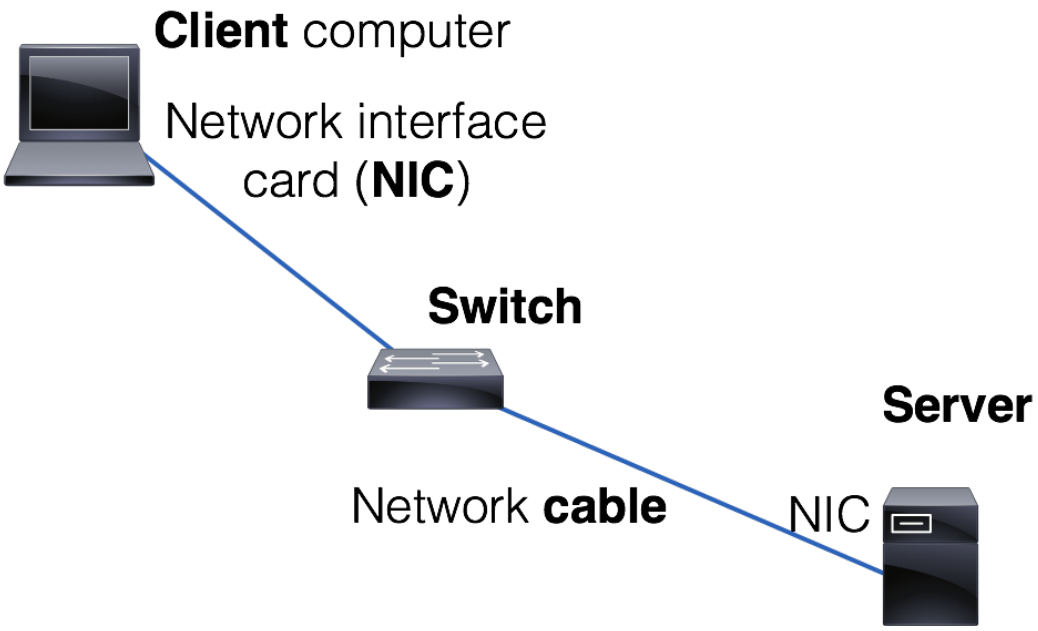
Alice would use the web interface to compose an email. When she clicks the "Send" button in her browser, the email is transferred to the web server using an HTTP POST request. The web server then starts an SMTP session with the mail server to deliver the email into the traditional email service.

Note that the core email service is still implemented using SMTP, in particular the email exchange between mail servers of different providers. That way, the system still uses the same open standard protocols and is fully compatible with traditional client-server email. Bob will receive the email no matter whether he prefers to use web mail or IMAP to access his emails, and in fact, he may be using both (e.g. a dedicated app using IMAP on his phone, but a web application on his laptop).

7.4 Physical Layer

The physical (or hardware) layer is the lowest-level layer of the Internet Model. It contains the actual network hardware, including cables, antennas, and network interfaces. For a particular type of network, the physical layer is standardised so that devices from different manufacturers can interoperate. Two devices that are directly connected by a cable or a radio link exchange messages at the physical layer.

Let's take a look at a typical Local Area Network (LAN):



A message that is sent, e.g., from the client to the server follows a number of steps:

1. In the client software, the message is represented digitally as a sequence of bits.
2. The message is passed to the Data Link Layer software in the client, together with the *address* of the sender and the receiver. The Data Link Layer software on the client now controls the *network interface card* (NIC), the actual piece of hardware that establishes the connection to the network. It instructs the hardware to send the message.
3. The hardware translates the bits of the message into *signals*. In our example, let's assume we're using standard Ethernet, which means that the signals are in fact electrical signals that are transmitted through a copper cable.
4. In the switch, the physical layer hardware receives the signal, decodes it back into bits, which are interpreted by the switch's Data Link Layer software. The switch checks the destination address and forwards the message to the destination, by sending it out on the physical port that the server is connected to. Again, the physical layer in the switch converts the bits into a signal that is transmitted through the cable to the server.
5. In the NIC inside the server, the physical layer receives the signals and translates them back into bits. The Data Link Layer reconstructs the message, checks that it has been received correctly, and then passes it on to the higher-level layers.

Network Hardware

Let's first look at the actual hardware, i.e., the components we can buy and hold in our hands. After that, we'll see how messages (sequences of bits) can be encoded into signals.

Network Interface Cards

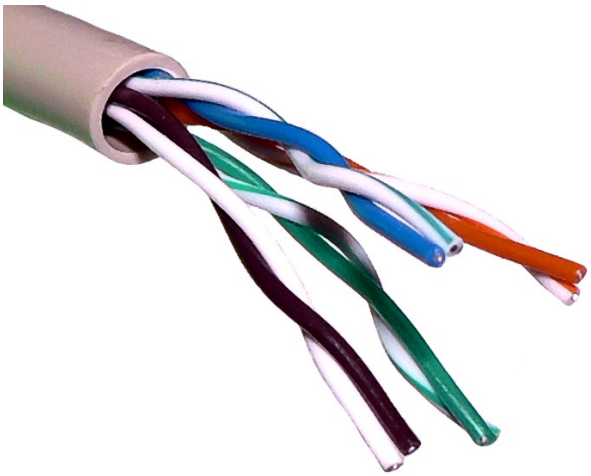
The NICs are the hardware components that connect devices to the network. In the case of wired networks, the physical connection is provided in the form of a socket into which you can plug the network cable. For wireless networks, a NIC is connected to an antenna in order to send and receive radio signals.

NICs are called "cards" because in the early days of networking, you would buy an expansion card that you could slot into your PC in order to connect to a network. Nowadays, most computers even have multiple NICs built into their motherboards. For instance, a smartphone can usually connect to the mobile phone (3G or 4G) network, to WiFi networks, and to other devices via Bluetooth. That's already three different NICs that are all integrated into the phone's motherboard. Most Laptops don't come with wired NICs any longer but only contain WiFi and Bluetooth (and sometimes 4G) interfaces. PCs still often have Ethernet built-in, and sometimes WiFi.

Network Cables

Networks can be built using a variety of different cables. The first distinction can be made by the main material of the cable. If the network uses electrical signals, the cable will contain copper wires. Examples for this type of cable are

- UTP (Unshielded Twisted Pair), which contain several pairs of copper cables, and each pair is twisted together in order to reduce interference. Each individual copper cable is surrounded by a plastic insulation layer, and there's another layer of insulation surrounding all the pairs. This is the most common type of LAN cable. E.g., Monash computer labs and most phones on Campus use UTP cables to connect to the network. UTP cables come in different *categories*. The higher the number, the better the quality of the cable, and the higher the transmission rate they can be used for.
- STP (Shielded Twisted Pair) is similar to UTP, but adds metal shielding to provide better protection from electromagnetic interference. STP cables are required for very high speed Ethernet (beyond 10 Gb/s), or in environments with strong electromagnetic interference.
- Coaxial cables. These look like TV cables, with an inner wire surrounded by an insulation layer and a wire mesh. The original Ethernet used coaxial cables, but they are not common in network installations any more.



Unshielded Twisted Pair cable. By Baran Ivo (Own work, Public domain), via Wikimedia Commons

If the network uses light signals, the cable contains an optical fibre. Laser light is then sent through the fibre, which conducts the light with very little losses. Networks that use optical fibres can achieve higher data transfer rates than networks based on copper cables, and are not affected by electromagnetic interference at all. The submarine cables that provide network communications between the continents use optical fibres to maximise the amount of data that can be sent through a single cable. Optical fibres are also commonly used for backbone networks, e.g. as high-speed connections between switches.

Signals

In order to get a rough understanding how we can physically send a message from one computer to another one, we first need to understand that we transmit messages using *physical signals*. A signal can be defined as *energy* travelling through a *medium*. Take an everyday signal as an example: someone bumps into a parked car and the car alarm goes off. The alarm is implemented as an "audio signal", a sound that is generated by a siren in the car, transmitted through the air, and received by everyone within a few hundred meters. Physically, a *sound wave* travels through the air, which means that the air is "rhythmically" compressed and decompressed in a certain pattern. That way, the pressure differences transmit energy from the alarm to your ear, which then uses the energy contained in the air pressure waves to create electrochemical signals in your brain that you interpret as hearing a noise.

What's important to learn from the example is that the signal is a *wave* that travels through a *medium,* in this case, air. In computer networks, we have *electrical signals* that travel through copper cables, or *radio waves* that travel through space, or *light waves* that travel through optical fibres (or, indeed space, since we can even implement networks using laser light, but that's rather experimental).

Digital Versus Analog

In particular when talking about signals in cables, we can distinguish between digital and analog signals. Digital always means that we have *discrete* states, whereas analog refers to something *continuous*, typically varying over time. E.g. you could consider share prices to be digital: the price can only increase or decrease by a certain amount at a time (e.g., one cent). On the other hand, temperature or air pressure are analog: if the temperature increases from 19 degrees to 20 degrees, it has to go through all possible values in between, there are no discrete steps.

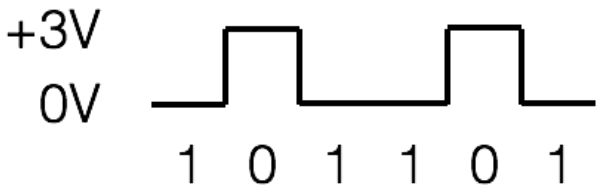
The same is true for signals. A digital signal is a waveform with a limited number of discrete states. An analog signal is a continuous wave, which can often be described mathematically in terms of sine waves. We can use both digital and analog signals for transmitting data.

Digital Signals

You should already be familiar with digital signals after working through the module on Boolean Algebra: it's simply a sequence of 0s and 1s that are sent through a wire, much like the inputs and outputs of logic gates. What wasn't covered in earlier modules was what exactly it means for a wire to transport a 0 or a 1. The obvious, intuitive answer would be that since a wire carries electricity, 0 mean "no electricity" and 1 means "electricity", much like switching a light off and on. In reality, this is a bit more complex.

A digital signal encodes 0s and 1s into different *voltage levels* on a copper cable. In the most basic form, this could indeed be 0V to encode a 0, and some positive voltage (typically +3V or +5V) to encode a 1.

But nothing prevents us from choosing the opposite encoding, where 0V represents a 1, and +3V represents a 0, as in the picture below.

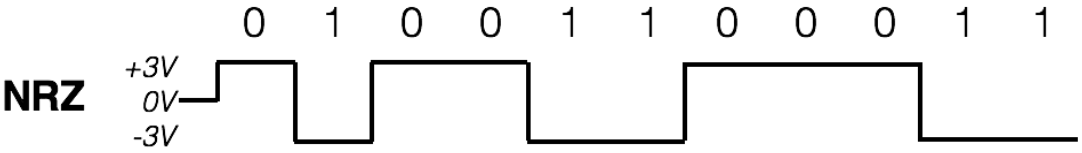


This is what we call a *square wave,* because the wave form rises and falls sharply. The horizontal axis represents time. As time progresses, the sender switches the voltage on the cable from 0V to +3V, which the receiver can detect, and it now knows that the sender is sending a 0 (in this case). When the sender switches back to 0V, the receiver interprets that as a 1.

Look at the two 1s in the middle of the picture. How does the receiver know it's two 1s in a row rather than a single 1? The trick is to define a fixed *time window* for each bit that's being transmitted. For example, let's assume we want to achieve a transmission rate of 1,000 bits per second (which is really slow for modern networks!). In that case, each bit will be transmitted for 1 millisecond (so that we can fit 1,000 bits into one second). For the message above, the sender would first leave the voltage at 0V for 1 millisecond, then switch to +3V for 1 millisecond, and then switch back to 0V for 2 milliseconds to send the two 1s. The receiver needs to measure the time in order to decode the message, and it needs to know quite precisely how long each bit is being transmitted.

The technique we just saw is called *unipolar encoding*, because we only use one polarity (positive voltage). This is very simple, but it has the disadvantage that for the sender it can sometimes be difficult to distinguish between 0V and a small positive voltage, e.g. if there is noise in the signal. A more robust strategy is to use both positive and negative voltage, to achieve a bigger difference in the signal. This is called *bipolar encoding.*

The simplest bipolar encoding is called "Non-Return to Zero", and it looks like this:

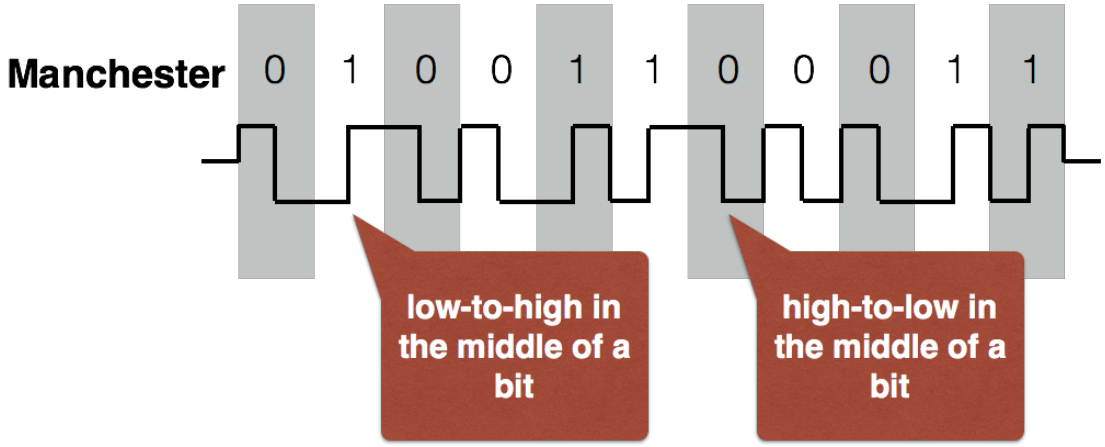


Here, a 0 is represented by positive voltage, while a 1 is represented by negative voltage. As for unipolar encoding, the choice of 0-positive, 1-negative is completely arbitrary, but sender and receiver have to agree to use the same encoding.

Both unipolar and NRZ require the receiver to somehow synchronise with the sender, in order to find out exactly how long the transmission of each bit is supposed to take. The easiest way to achieve synchronisation is to send a sequence of 0s and 1s, so that the signal switches between the two states regularly, and the receiver can measure the length of the time window. But if the receiver's clock runs just a little bit faster or slower than the sender's, then the synchronisation will *drift*: imagine the message is just a long sequence of 10,000 zeroes, at 1,000 bits per second. The receiver simply sees the +3V for ten seconds. But it now has to measure the ten seconds with a precision of one millisecond, because if it measures +3V for what it thinks is 9.999 seconds, it only receives 9,999 zeroes! So if we get the timing wrong just by a little bit, and without an additional way of synchronising the signals, this can easily

introduce errors.

Manchester Encoding was developed to avoid this kind of synchronisation problem. The idea is to *embed* a synchronisation signal into the data signal, so that the receiver can always re-synchronise itself with the sender. We say that Manchester Encoding provides a *self-clocking* signal. Here's how it works:

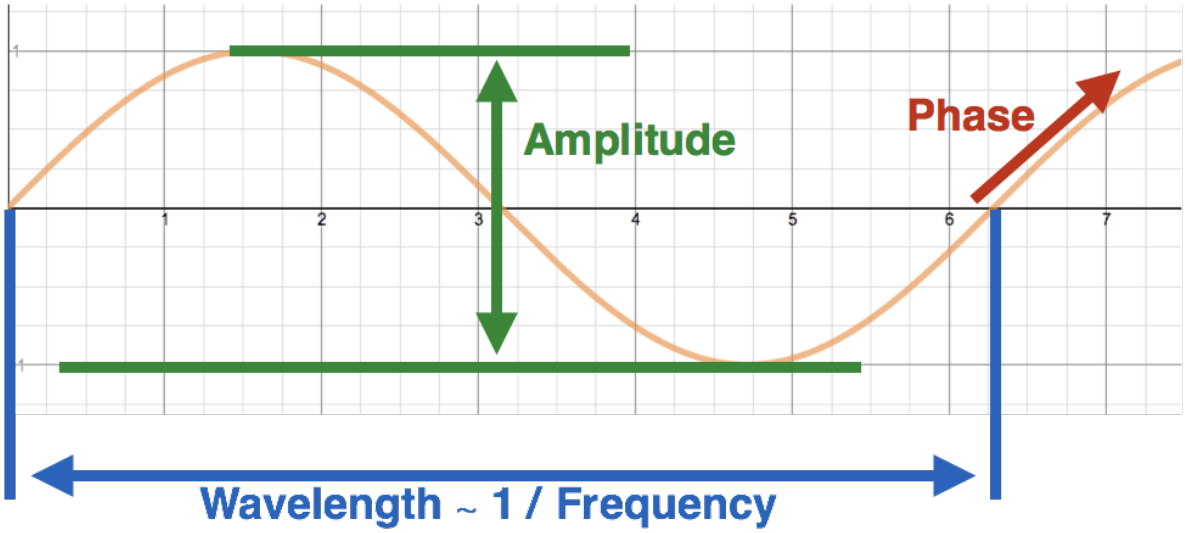


The actual data is transmitted in the middle of the time window for each bit. A 1 is transmitted as a transition from negative to positive voltage, and a 0 as a transition from positive to negative voltage. In order to transmit two 1s or two 0s in a row, we have to "reset" the signal in between the two bits. For example, in the picture above, the third and fourth bit are both 0, which requires a high-to-low transition. In between the two bits, we therefore have to bring the signal back from low to high. Since each bit is represented by a *transition* (i.e., low-to-high or high-to-low) rather than a *state* (high or low), we are guaranteed to have at least one change of voltage in each time unit. The receiver can extract the timing from this information and continuously re-synchronise itself with the sender.

The early Ethernet standards (with 10 Mbps transmission speeds, using coaxial or UTP cables) were based on Manchester Encoding.

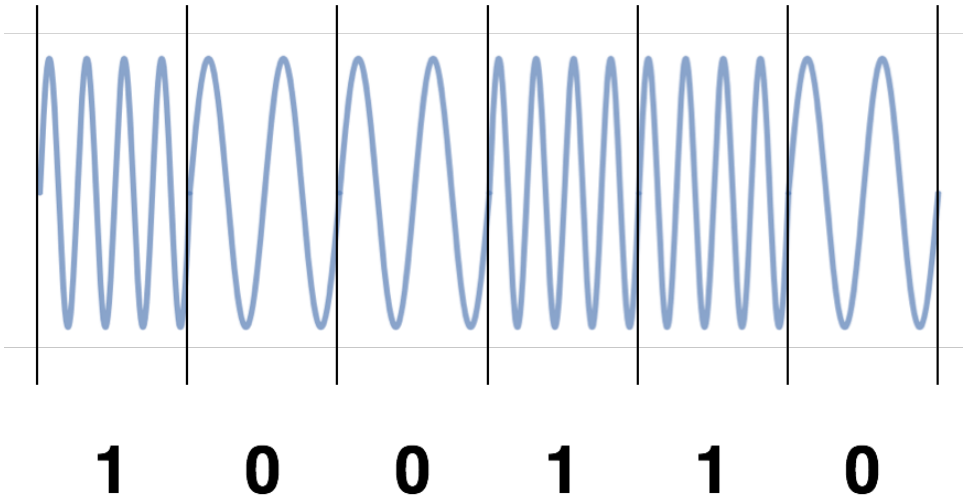
Analog Signals

The square waves of digital signals are easy to connect to digital hardware, because they more or less correspond directly to the two states of logic gates. However, digital signal don't make very efficient use of the full transmission capacity of the network cable, its so-called *bandwidth*. That's because they only encode information into one aspect of the square wave, and that is whether it is at the positive or negative end of the voltage at any point in time. Analog signals can contain much richer information. Let's assume that our signal is a sine wave, looking like this:



Like the square waves in digital encoding, the wave goes from zero to some upper limit, then down into the negative, and then back up. But rather than the steep "switch" of a square wave, an analog (sine) wave *gradually* changes between the different states. We can describe the wave using three parameters. The *amplitude* is the height of the wave. In a sound wave, the amplitude determines the volume, i.e., higher amplitude means a louder noise. The *frequency* determines how many oscillations per second the wave goes through. In sound waves, a higher frequency results in a higher-pitch tone. For light waves, we often talk about the *wavelength*, which is the distance a wave travels during one full oscillation, but wavelength and frequency characterise exactly the same aspect of a wave. Finally, the *phase* tells us in which direction the wave is going at a particular time point. E.g., at time point 0, the wave above starts moving in an upwards direction.

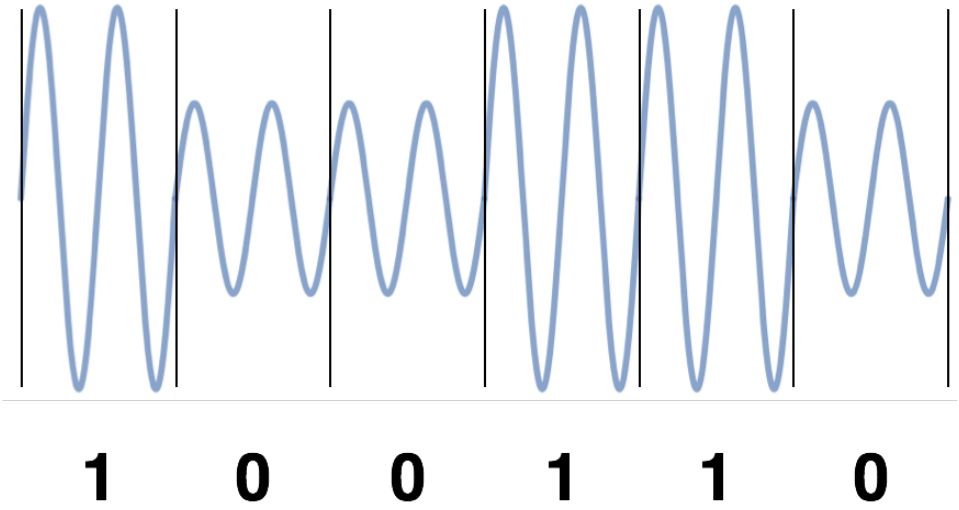
In order to transmit data using analog waves, we can now simply modify each of these three parameters. For example, if we modify the frequency depending on whether we want to send a 0 or a 1, it could look like this:



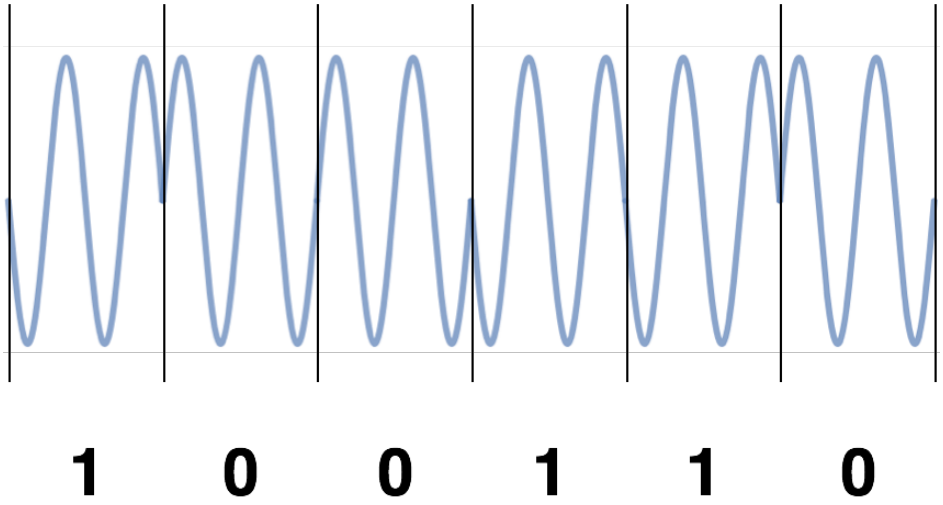
In the first time unit, the wave has a high frequency (many oscillations per second), which we here interpret as a 1. In the next time unit, the frequency is low, which we can interpret as a 0. This technique is called *frequency modulation* (FM).

Alternatively, we can encode our data into the amplitude of a wave, while leaving the frequency constant

over the whole transmission:

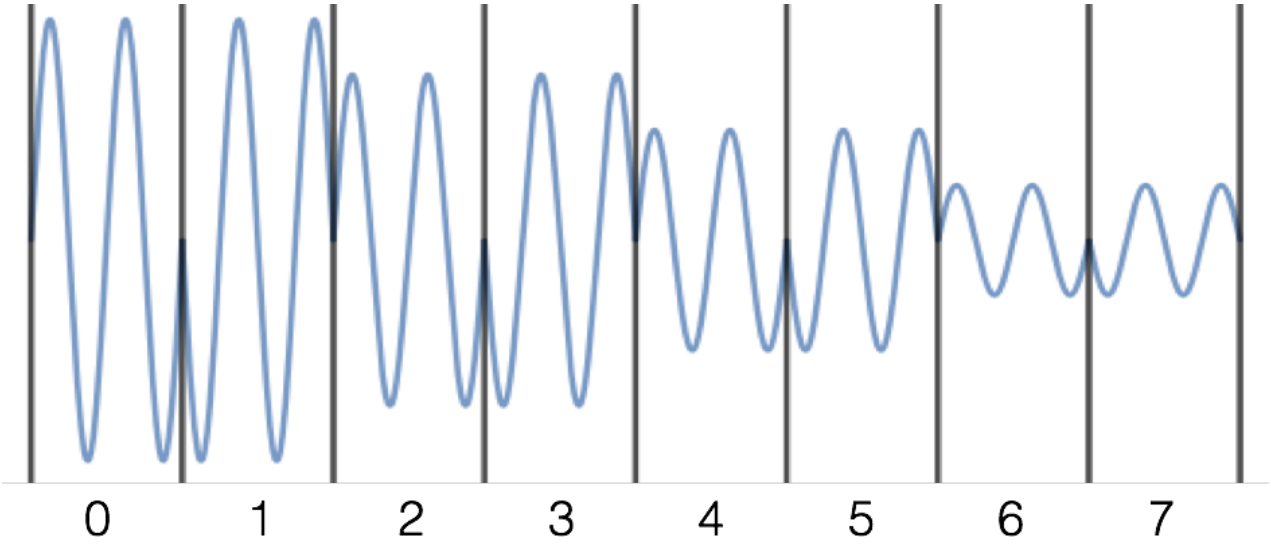


Here, again we chose a high amplitude (a "loud" signal) to represent 1, and a lower amplitude to represent 0. This is called *amplitude modulation* (AM). Finally, of course, we can play with the phase:



In the first time unit, the wave is starting in a downward direction, which we interpret as a 1. In the second and third time unit, the wave is starting in an upward direction, interpreted as a 0. Note how the direction changes when going from 1 to 0 or 0 to 1. This method is called *phase modulation* (PM) or *phase-shift keying* (PSK).

Up until now it may not be obvious why analog modulation techniques should allow us to pack more data into the same time unit, i.e., increase the transmission rates. The key to that lies in using *multiple different* amplitudes and phases instead of just two, and in *combining* amplitude and phase modulation at the same time. Here's an example:



By using four different amplitudes and two different phases, in each time unit we can now encode a number between 0 and 7 rather than just a 0 or a 1. E.g., the second-highest amplitude combined with the downwards phase represents the number 3, while the lowest of the four amplitudes combined with the upwards phase encodes 6. That means we can transmit *four times the amount of data* in the same time unit compared to simple AM or PM, and also compared to digital encoding.

Attenuation

An important effect that applies to any kind of signal is *attenuation*, which is the weakening of the signal with increasing distance from the transmitter. We all know the effect from sound waves, where the closer we get to a speaker, the louder the sound appears to be. The same happens with all signals. This is particularly pronounced with radio waves, such as the ones used for wireless communication. For example, WiFi networks use frequencies in the range of 2.4 GHz or 5 GHz (depending on the standard used, see the Data Link Layer module). Any obstacle, such as a wooden door or a light wall, will cut signal strength in half, while concrete walls can reduce signal strength by a factor of more than 10!

Modems

The process of turning digital data into analog signals is called *modulation*, and the reverse process is *demodulation*. A device that does both these things is called a *modulator/demodulator*, or **modem**, for short.

Early modems did not just turn digital data into analog electrical signals, they actually used a speaker to turn them into acoustic signals (sound waves), which were then transmitted using an ordinary telephone. On the other end, a microphone would record the sound signals, turn them back into (analog) electrical signals, and the receiving modem demodulated them into digital signals. The second generation of modems was then directly connected to a telephone line, avoiding the (noisy) conversion to sound and back, which improved data rates quite a bit. Still, modems were limited to the *bandwidth* of audio signals, i.e., the frequencies that telephone lines were designed to carry.

ADSL, one of the most popular technologies for accessing the Internet now, also uses modems connected to telephone lines. The difference is that the receiving modem is at most a few kilometres away, in the local telephone exchange. That way, ADSL can use a much broader range of frequencies (because they don't have to be carried through the legacy phone network), and achieve much higher data rates.

Acoustic coupler modem. Image by Flickr user Ryan Finnie, CC BY-SA 2.0.

Modern networking technologies such as 4G, or communication over optical fibres, all use variants of the modem idea.

7.5
Data Link Layer

Sitting right above the physical layer, the data link layer is responsible for **controlling** the hardware and for **error detection**. The main function we will look at here is the so-called **Media Access Control** (MAC), which controls when a device is allowed to transmit.

Media Access Control

The main problem that MAC tries to solve is that in most kinds of network, only one device is allowed to transmit at the same time. Compare this to a room full of people - if everyone is talking at the same time, you won't be able to communicate with someone on the other side of the room. This is because everyone in the room *shares the same medium* (in this case, the air used to transmit sound waves). The same is true for many networks.

There are two approaches to MAC. The first one is *controlled access*, where only one device has permission to send at any point in time, and we either have a central authority assigning permission to send, or the permission gets passed from device to device. The analogy would be a moderator in a discussion or a teacher in a classroom. The second approach to MAC is called *contention-based access.* Here, access is provided on a first-come first-served basis, i.e., any device can start transmitting at any time. Usually, devices would avoid starting a transmission if they can "sense" that some other device is currently transmitting. However, it is possible (and unavoidable) that two devices start transmitting at exactly the same time. In that case, we say that the frames they are sending **collide,** which means that the two signals are superimposed and can't be decoded by the receivers any more. The frames are damaged. Contention-based MAC therefore needs a mechanism for detecting when a collision has happened, in order to cause a re-transmission of the damaged frames. To use an analogy again, think of a conversation among friends. You would avoid talking over one another, but rather wait until whoever is speaking finishes. Then two people might start speaking at the same time, which they will notice, and through some "social protocol", one of them will wait while the other will start again. The advantage of contention-based MAC is that no "moderator" is required, the network is essentially self-organising. That's why this was the natural choice for the type of LAN technology that is most popular today: Ethernet.

Ethernet

The original Ethernet technology was developed in 1973 and standardised in 1980, in a series of standards in the IEEE 802.3 family. It was a huge success, because it allowed companies to set up networks at a fraction of the cost of competing technologies. Today, almost all Local Area Networks are based on Ethernet (which has of course evolved a lot in the last 40 years).

At the physical layer, the original Ethernet used a single coaxial cable that was shared by all devices on the same LAN. A new device could be connected by "tapping" the cable, i.e., essentially drilling a tap wire through the insulation into the copper core. This technology could deliver data at 10 Mbit/s - not bad considering that even today a lot of people are still connecting to the Internet using ADSL at speeds around 10-20 Mbit/s! Nowadays, Ethernet mostly uses UTP cables connected to switches (rather than a single long cable), and typical speeds are 1 Gbit/s (e.g. in the Monash labs) or 10 Gbit/s (e.g. in the Monash backbone networks), with speeds beyond 100 Gbit/s under development.

Media Access Control in Ethernet is based on the CSMA/CD method:

- CS means **carrier sense**: A device "listens" to the network, and only starts a transmission when no other device is currently transmitting.
- MA means **multiple access**, which just describes the fact that multiple devices share the same medium (the same cable).
- CD means **collision detection**: While a device is sending, it will monitor the network, and if it detects any other signal than the one it is sending, it knows that a collision has happened. In that case, it immediately stops the transmission of the actual frame and transmits a *jam signal* instead, which makes it unmistakably clear to all other connected devices that a collision has happened. It then starts re-transmitting the frame.

Can you spot the problem in the CD phase? For a collision to happen, it always takes two devices, transmitting at the same time. Now both devices detect a collision, both start a jam signal, and then both re-transmit the frame they were trying to send. So this would just lead to a guaranteed collision again. To solve this problem, any device simply waits for *a random, short amount of time* before trying to re-transmit. If after that time another device has already started transmitting, the carrier sense (CS) part of the method will make sure that our device keeps quiet. The randomisation has the effect that it is likely that only one device picks the shortest waiting time and starts transmitting. All other devices wait longer, detect the carrier from the first device, and wait for it to finish.

Note that each device only sends a *single frame* at a time, giving other devices a chance to jump in before it starts transmitting the next frame.

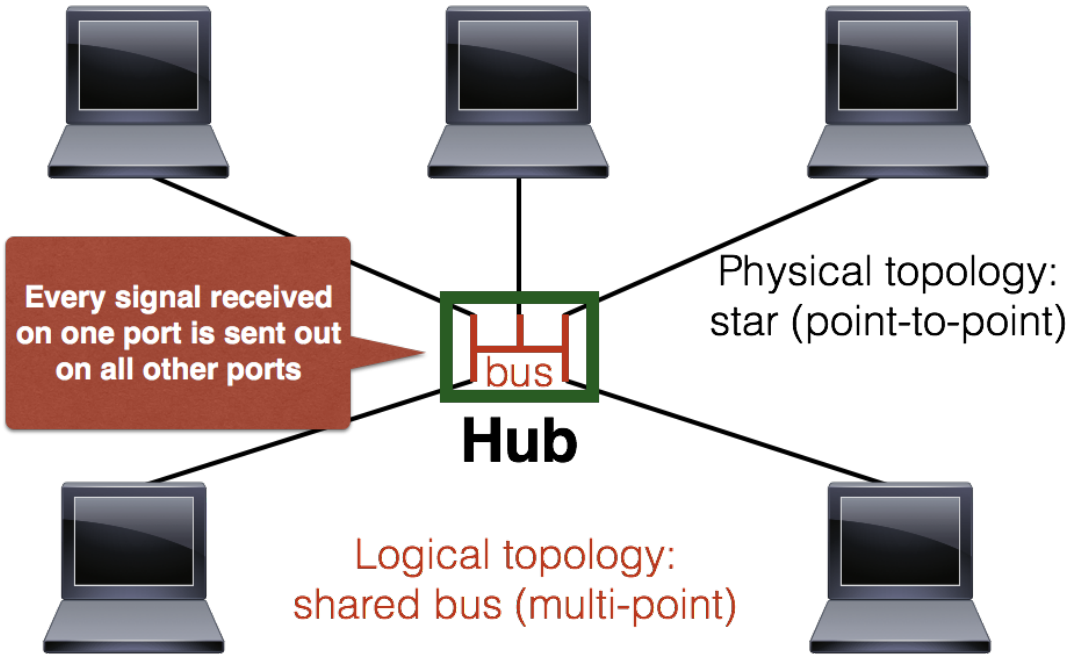
Ethernet as a Shared Bus

As mentioned above, an original Ethernet LAN was based on a single long cable that all devices were connected to. We call this a *shared bus* topology, because all devices share a single bus (like the bus in a CPU), or, alternatively, a *multi-point* topology (each "point" is where a device is connected to the cable).

One consequence of a shared bus approach is that all devices receive all messages that are sent into the network, even the ones that were not meant for them. Each message must therefore include a *destination address*. All devices receive the message, each device checks whether the destination address in the message is equal to its own address, and only the intended recipient will process the message. All other devices simply discard it.

For this to work, each device in an Ethernet LAN needs a unique address. To make things really simple to set up, the designers of Ethernet decided that the manufacturers of Ethernet hardware would have to hard-wire a unique address into each Ethernet NIC they sell. That way, we can simply plug a new device into an existing network, without first configuring its address (and it also means we can't make mistakes in the configuration and set up two devices with the same address). We call these addresses *MAC addresses*, and they consist of six bytes, usually written as six hexadecimal numbers separated by colons. For example, the computer that this module was written on has the MAC address ac:87:a3:14:9e:59.

Using one long cable for a LAN had the advantage of being comparatively cheap, but it made maintenance difficult. If any of the physical connections wasn't done properly, or any of the attached devices wasn't behaving as it should, it would affect the entire network, up to the point where it would simply stop working altogether. The problem here is the bus topology, so technology was developed to change it into a *star topology:*



The central component in a star topology was a *hub*, a small device with a number of sockets, so that each computer on the network could be connected to the hub using an individual cable. However, the trick with hubs was that they made the network *behave* as if all computers were still connected to a shared cable. We say that while the *physical* topology was now a star, the *logical* topology was still a bus. The way this works is that a hub simply repeats any signal it receives via one socket on all sockets except for the one from which the signal was received. That means that all computers can still "hear" all other computers on the network (for carrier sensing), but it also means that when computers start transmitting simultaneously, the frames are damaged in collisions.

Let's summarise the disadvantages of shared-bus Ethernet:

- It is *half-duplex*, meaning that only one device can send at a time.
- The network *broadcasts* all messages, which means that messages get delivered to all devices rather than just the actual destinations.
- The reliance on collision detection limits the size of the network (we won't explain the details here, but if the network gets too large, a device may not be able to detect that a collision is happening while it is still sending a message).

In the next section, we will see how modern Ethernet networks work around these limitations by replacing hubs with switches.

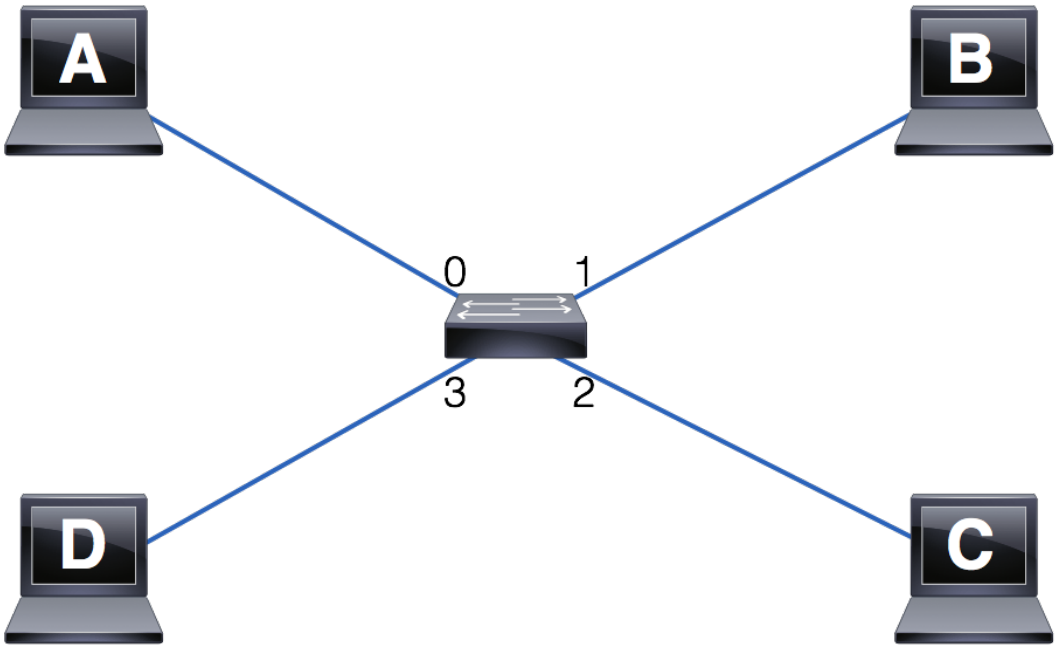
Switched Ethernet

The solution to the problems of hub-based, shared medium Ethernet is to move from a logical bus topology to a logical star topology. That would mean that the circuit is no longer shared, and messages are sent directly from one device to another, rather than broadcasting them to the entire network. The device that enables this kind of networking is called a *switch*. It looks just like a hub, but it behaves quite differently.

A switch is a true data link layer device. It reads an incoming frame, checks its destination MAC address, and then sends the frame to the correct port that is connected to the device with that address. So how does the switch know where to send the frame? Again, it would seem that the simplest idea would be that

a network admin has to configure a switch, telling it which computer (or rather, which MAC address) is connected to which socket. But that has two problems: it's not very flexible (we can't just plug in a new computer), and it would make the switch much more complex since you suddenly need a way to set up that configuration.

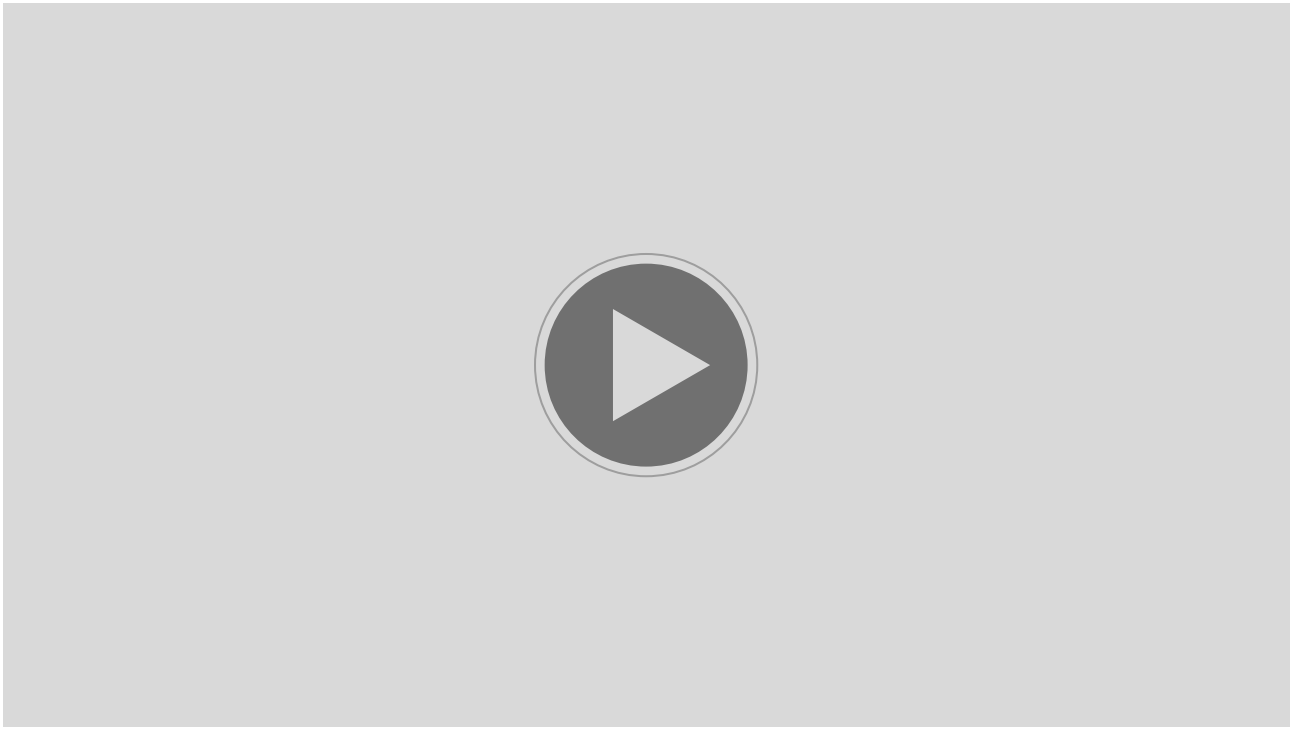
Switches work with a much simpler approach. Let's assume we've just turned the power on in the network below, and the switch starts operating. It doesn't know anything about the devices it is connected to.



Now let's assume laptop A wants to send a message that is addressed to laptop B. The switch receives the message, but it has no idea whether B is connected to port 1, 2 or 3 (it knows it's not connected to port 0). So it simply behaves like a hub and broadcasts the message to all ports! That's always a valid approach, because Ethernet devices need to be able to deal with frames that are not addressed to them (after all the devices don't know whether they are connected to a switch or a hub).

There's one more thing the switch does before broadcasting the frame everywhere. In addition to the destination MAC address, an Ethernet frame also contains the source MAC address. That way, the switch can learn that A is connected to port 0. It stores that information in a **forwarding table**. Let's now assume that B replies to A's message. The switch receives a frame addressed to A, and now it finds A in the forwarding table, so that it can deliver the frame directly, without broadcasting. In addition, it learns that B is connected to port 1, so that any future messages to B can also be delivered directly.

The following animation explains again how a switch learns its forwarding table.

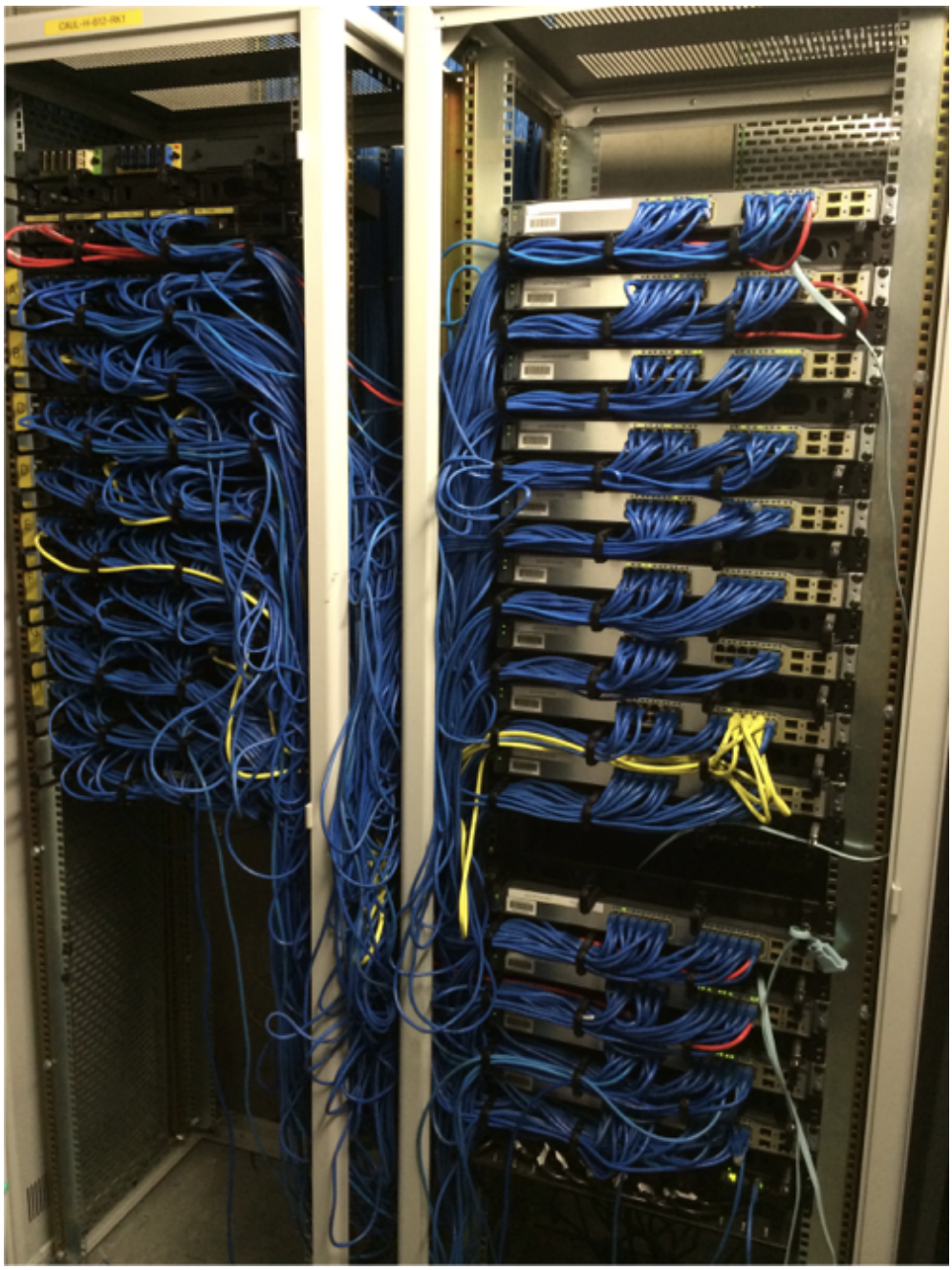


(https://www.alexandriarepository.org/wp-content/uploads/20170511073749/switch_forwarding.mp4.mp4)

That way, after just a single frame from each connected device, the switch has learnt all the MAC addresses and doesn't have to use broadcasting any more. Every message from now on will be delivered point-to-point. Switches can do even more if they are built with a little bit of buffer memory. Assume that both A and B want to send a frame to D at the same time. The switch can receive both frames simultaneously, store one of them in its memory, forward the other one, and when it's finished, forward the stored one. This means that switches can eliminate almost all collisions.

In effect, switched Ethernet can use up to 95% of the theoretical capacity of an Ethernet network, whereas hub-based shared Ethernet cannot achieve more than 50%.

Here's what a typical installation of switches looks like at Monash University:



The switches are located in the rack on the right. Each switch has 24 ports, which are connected using the blue UTP cables to patch boards on the left. The red cables connect the switches to the rest of the Monash network. The patch boards are in turn connected to the cables leading into all the offices on one floor of the building. That way, we can change which switch an office is connected to by simply changing the patch cable. What you can't see easily here, is that there are many empty sockets on the patch board. The idea is that you fit out each office and meeting room with several network sockets, in case you need them later (because it is expensive to put in new cables later). But you only connect those that are actually in use to a switch.

Wireless Local Area Networks

Wireless networks are probably the standard way you connect to the Internet. On your phone, you may be using a 4G network while you're outside, and a WLAN (also known as WiFi) when you're at home or on campus. If you have a laptop, it probably uses WLAN as well. The advantages are obvious:

- We don't need any cables. This can be particularly important if there are no network cables

installed in a building and we're not allowed to change that (e.g. because we're renting or because it's a heritage building).
- Network access is much more flexible because we are not limited to stay close to a socket.
- Wireless technology can facilitate completely new ways of using the network, by enabling people to be mobile in their work environment (e.g. nurses and doctors in a hospital).

In this module we will only look at Wireless LAN as defined in the IEEE 802.11 family of standards. These standards are usually known as WLAN or WiFi. They were developed between 1997 and 1999, and are now widely used. Each new version of the standard has brought faster transmission rates, and different versions are marked by adding a letter to the standard, such as 802.11a, 802.11b, 802.11g, 802.11n, 802.11ac, with the n and ac versions being the ones that are supported by most modern hardware.

There are other wireless technologies that we won't cover in any detail here. But you may come across the term WiMAX, which has been standardised as IEEE 802.16, and which was designed as a successor to both WiFi and 3G mobile phone technology. Another wireless networking technology you've probably heard of is Bluetooth (IEEE 802.15), which is also called Wireless Personal Area Network (WPAN), since it is mostly used to create small networks of personal devices (e.g. computer-to-smartphone or smartphone-to-fitness-tracker).

WLAN Frequencies

Since WLAN is a wireless standard, it uses radio waves to communicate. The *spectrum* of radio waves, i.e., the range of frequencies different devices are allowed to use, is strictly regulated. For example, mobile phone companies typically pay billions of dollars to the state for the permission to use a certain range of frequencies exclusively. WLAN is different, because you don't need to ask anyone for permission when you install a new wireless network. That's because WLAN uses a part of the spectrum that has been allocated to be used freely, as long as devices don't send with too much power (so that their range is essentially local, and you can't interfere with too many other devices around you).

There are two main *bands* that WLAN devices can use, the original one around the frequency of 2.4 GHz, and a newer one around 5 GHz. The important thing to understand about radio communication is that higher frequencies mean higher transmission rates: Remember that we pack information into an individual wave cycle, and the higher the frequency, the shorter the wave cycles, the more cycles per second. So why don't we just pick the highest possible frequency we can generate? Unfortunately, high frequencies also have one significant disadvantage: They have much stronger *attenuation*, meaning that they become weaker with distance much more quickly than lower frequencies. So we have to find the right compromise between transmission speed and maximum distance between sender and receiver.

As mentioned above, a WLAN device may use a frequency *around* the 2.4 GHz mark. In fact, the IEEE standard defines a number of *channels*, which are well-defined frequency ranges that WLAN devices are allowed to use. That way, you can use a different channel than your neighbours, which means that your wireless network doesn't interfere with theirs. It works a bit like radio stations, each sending on a different frequency, and you tune into a particular station.

Here's a table with the channels that are available in the 2.4 GHz frequency band. The frequency ranges are only approximate and depend on the exact protocols and standards used.

Channel number	Frequency range (GHz)
1	2.401 - 2.423
2	2.406 - 2.428
3	2.411 - 2.433

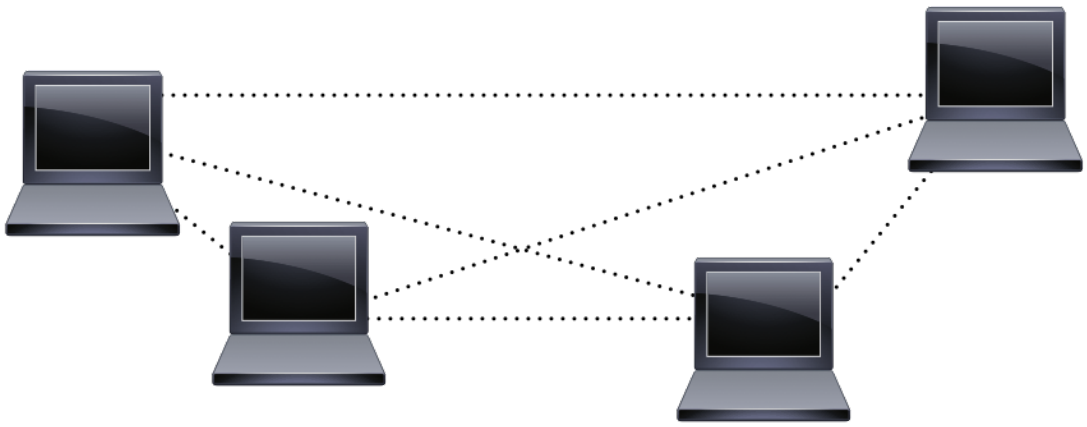
Channel number	Frequency range (GHz)
4	2.416 - 2.438
5	2.421 - 2.443
6	2.426 - 2.448
7	2.431 - 2.453
8	2.436 - 2.458
9	2.441 - 2.463
10	2.446 - 2.468
11	2.451 - 2.473
12	2.456 - 2.478
13	2.461 - 2.483

As you can see, the standard defines 13 channels, each of which is 22 MHz wide, but most of them overlap. In fact, only three channels don't overlap: channels 1, 6 and 11. What that means in practice is that we would usually set up WLAN networks that are located close to each other on different channels, so that they don't interfere. What happens if we don't do that? There are two scenarios:

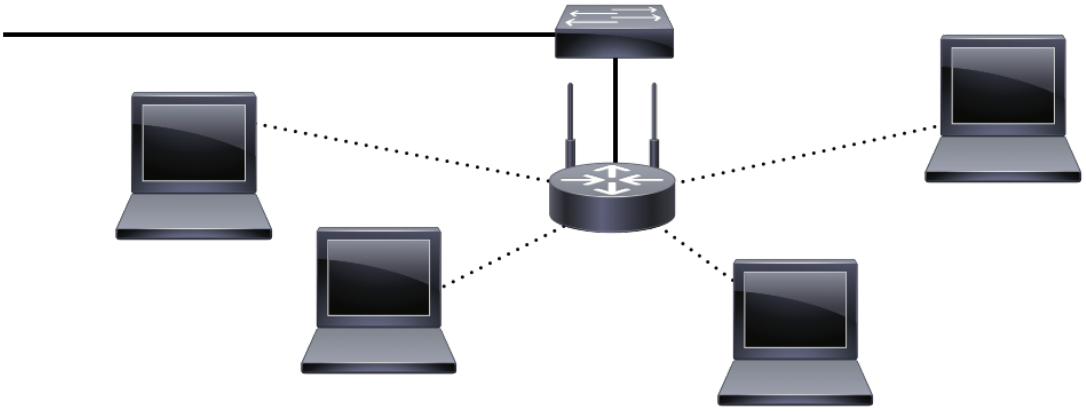
- Let's assume we set up two neighbouring networks on channels 1 and 3, which overlap in their frequencies. When the channel 1 network is transmitting, the channel 3 network can't really "hear" channel 1 (because it's mostly on different frequencies), but it hears a lot of noise in the frequencies that overlap. This can lead to frames being received with errors, which will result in a lower data rate.
- Alternatively, we could have set up two neighbouring networks both on the same channel, say channel 1. Now both can perfectly hear each other, and because WLAN also uses carrier sensing, they will try not to transmit at the same time. While the error rate doesn't necessarily increase (as in the case above), both networks now need to share the same frequencies, essentially resulting in only half the transmission rates that would be possible if each had their own channel.

WLAN Topology

The simplest possible setup for a wireless network is just a number of devices that can talk to each other. We call this an *ad-hoc* or *independent* network, and the technical term is an *independent Basic Service Set* (BSS), as in this diagram:



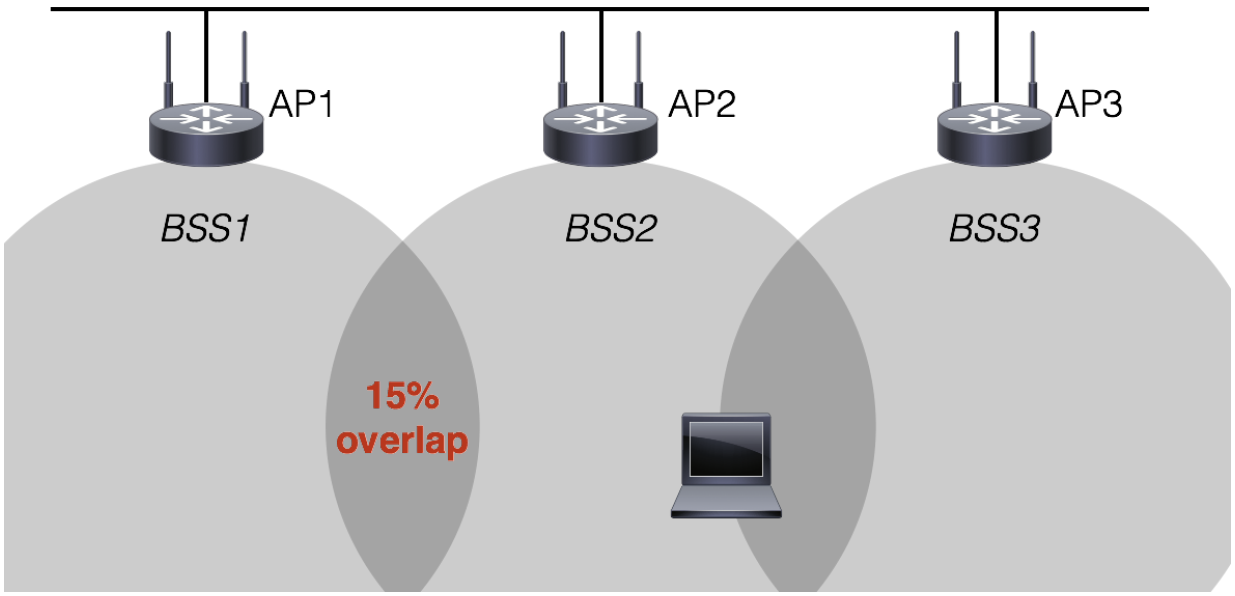
An independent BSS behaves much like an original, shared Ethernet, in that there is no central hub, a device can just send a frame into the network, and the receiver identifies that it should handle the frame by checking the frame's destination MAC address. This is the type of network you can create e.g. between a number of laptops.
However, most wireless networks use a central *Access Point* (AP):



We call this an *Infrastructure BSS*. The access point is connected to the rest of the network using cable-based Ethernet. At home, you may be using an access point that is integrated into your ADSL modem (or some other integrated device supplied by your Internet Service Provider). At Monash, you can see multiple APs attached to the ceiling of each lecture theatre and in many rooms and hallways. They are usually square plastic boxes, sometimes with antennas sticking out.

In an infrastructure BSS, all communication goes through the AP. That means that if the laptop on the left wants to communicate with the laptop on the right, it sends its frame to the AP, which then relays it to the other laptop, much like a hub would do. Still, all devices *can* hear all messages (because we're using a shared medium after all), but they will only react to messages from the AP.

Now that we know what a *basic* service set is, we can connect multiple of them to form an *Extended Service Set* (ESS). Here, multiple access points work together. They are all connected to the same cable-based network, and they have been installed so that the areas that they cover overlap. Furthermore, they all transmit the same *identifier* of the network (such as "eduroam" or "Monash Free Wifi"). That way, a mobile device can *roam* between different BSSs, by moving out of the coverage of one AP and into the coverage of another one, without ever losing connection to the network. Here's an illustration of an ESS:

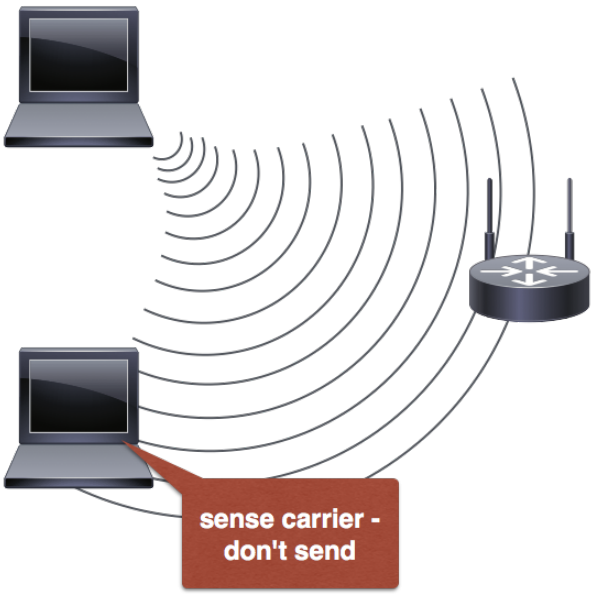


Each access point can cover an area of maybe 50×50 meters (depending on the technology and frequencies used, the materials of walls, and other factors). The laptop in the picture is already in a zone covered by both access points, which it can find out by measuring their respective signal strengths. As

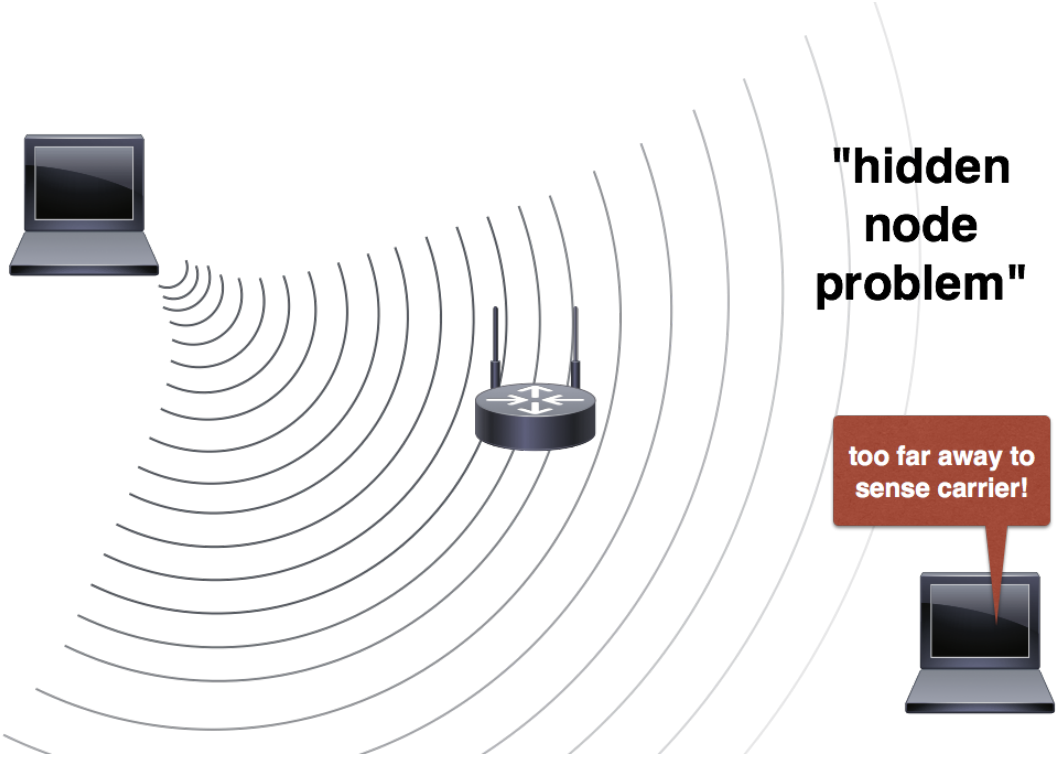
soon as the signal strength of AP2 is a little too weak and AP3 seems strong enough, the laptop will switch its connection to AP3. The technology is designed so that this happens entirely at the data link layer, which means that higher layers don't even notice that anything has changed. For example, if you're currently using in a video conference on your phone while walking around, you will stay connected as long as there is still an AP for the same network in your vicinity.

WLAN MAC

Media Access Control in wireless LANs is similar to MAC in a shared Ethernet. After all, we're also sharing a common medium, it's just that in this case it's radio frequencies rather than a shared cable. This means that collisions are a problem for WLAN just as they are for shared Ethernet. This diagram shows how two devices in a wireless network use carrier sensing:



The laptop in the top left sends a frame to the access point in the middle, and the laptop in the bottom right detects the carrier, so it waits with its own transmission until the other transmission has finished. But unfortunately the situation can become a little more complex in a wireless network. Have a look at the next diagram.



Now the two laptops are on opposite sides of the access point in the middle. When the left laptop sends, the access point can receive the frame without any problem, but the right laptop is so far away that the signal has become too weak. The laptop can't sense the carrier any longer because the transmitting laptop is "hidden". This "hidden node problem" could lead to many collisions, just due to the fact that radio signals get weaker over distance much more quickly than signals travelling in a copper cable.

WLAN therefore needs to be a bit more proactive than Ethernet. Instead of CSMA/CD, it uses CSMA/CA, where the CA stands for **collision avoidance**. There are two different CA mechanisms:

- Automatic Repeat Request (ARQ): After sending a frame to the access point, a WLAN device will wait for an *acknowledgement* from the access point that the frame was received correctly. That way, if the access point doesn't acknowledge a frame, the device knows that something has gone wrong, even if it didn't detect the collision because it was too far away from the other device. It can then re-send the frame. Of course, just like with regular Ethernet, re-sending immediately may be a bad idea because the other device involved in the collision would do the same. So, as for Ethernet, both devices will wait a random time before starting a new transmission. With each unsuccessful transmission, they will wait a little longer, so that eventually the other device can successfully deliver its frame.
- In addition, WLAN devices may also use *controlled access*. In that case, the device can send a short "Request to Send" (RTS) message to the access point, and it will only start transmitting a frame when the access point replies with a "Clear to Send" (CTS). This method is not used in all wireless networks (it's typically only required when many devices use the same network, such as on campus).

ARQ is in fact a very common technique for error correction, and we'll see it again in the Network Layer and Transport Layer modules.

7.6
Network Layer

The main responsibility of the network layer is **routing**, which is the mechanism for exchanging packets between different local area networks, and delivering them across multiple intermediate networks to their correct destinations.

Of course all layers are required for a network to function, but the network layer provides the core functionality for building *large* networks, by enabling networks to be split up into independent components that each have a manageable size.

On the Internet, the network layer is implemented by the **Internet Protocol**, the "IP" part of "TCP/IP".

In this module, we will first discuss *addresses* at the network layer - you may have heard of these addresses before, they are called IP addresses. After that, we will see how routers work.

IP addresses

Each device on the Internet that needs to send, receive or route messages requires at least one IP address for each of its NICs (Network Interface Cards). In principle, these addresses must be unique, so that a given address exactly identifies one interface.

The original Internet Protocol that was used for the first few decades of building the Internet has version number 4. It is still the most widely used protocol, so let's start by looking at IPv4 addresses.

IPv4 addresses

An IP address in version 4 of the protocol is 32 bits long. To make addresses more human-readable, they are typically written in "dotted-decimal-notation", as four decimal numbers representing the four bytes, separated by dots. For example, 130.194.66.43 would be a typical IP address. We can convert each of the four decimal numbers into binary to get the sequence of 32 bits:

10000010111000010010000101000101011

Now IP addresses are not just random 32 bit numbers, because that would make it very difficult to decide how to *route* a packet to its destination. Instead, IP addresses are *hierarchical*, much like postal addresses.

For the concrete address above, the first two bytes (130.194) in fact identify Monash University. That means that any IP address starting with 130.194... belongs to a computer inside the Monash network. That's already some progress compared to completely random addresses: If a device outside of the Monash network sends a packet to an address starting with 130.194, the routers on the Internet can look up this address and see that they need to send the packet to the Monash network (which can then deal with the details). To use the postal system as an analogy, if you post a letter from Melbourne to Sydney, the post office in Melbourne doesn't care about the street address, it just forwards the letter to the distribution centre in Sydney.

The Monash network comprises tens of thousands of devices (servers, routers, PCs, laptops, etc.), so we would also like to create some hierarchies internally that make it easier to manage the system. The 130.194 network is therefore divided into *subnets*, each of which is identified by a number of extra bits of the IP address. In our example address, 130.194.66.43, the first 16 bits identified the network (Monash University), so let's assume now that the next 8 bits identify the subnet. That would mean that any device with an IP address starting with 130.194.66 is a member of the same subnet within Monash University. The remaining 8 bits identify the concrete device within the subnet, which for historical reasons is often called the *host*.

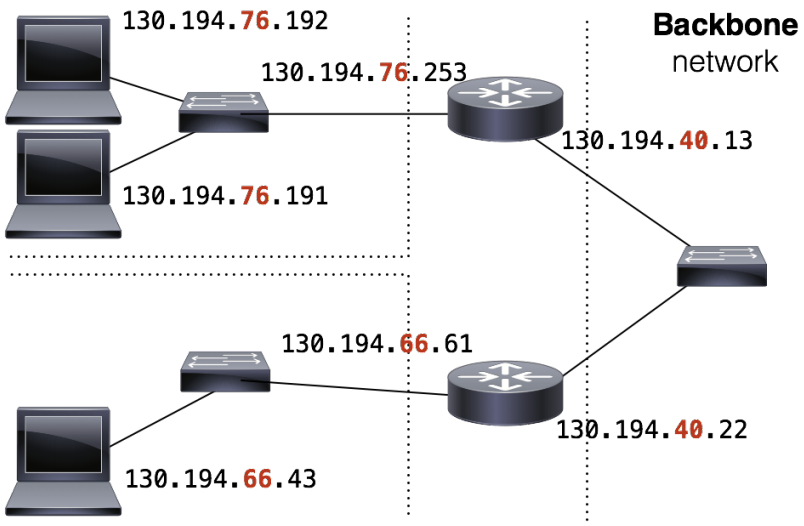
The example above divided the network, subnet and host part neatly into 8-bit blocks. That makes it easy to explain an address in dotted-decimal notation, but in realistic networks, it would be a bit limiting. If we always had to use 16 bits to identify the network, 8 bits for the subnet, and 8 bits for the host, each subnet would contain 256 different hosts (in fact, for technical reasons, only 254). We wouldn't have any flexibility to create larger or smaller subnets. For example Monash may want to put only 126 different computers in each subnet, but have 512 different subnets, instead of the 256 it could run using 8 bits to identify the subnet.

The solution to this problem is the *subnet mask*, which tells us how many bits of an IP address are used for the network plus subnet identifier. In the example above, the first 24 bits identify the network and subnet, so we would write the IP address including its subnet mask as 130.194.66/24. An alternative way of writing subnet masks is again dotted-decimal notation, where all the network and subnet bits are 1 and the other bits are 0. A subnet mask /24 can then be written as 255.255.255.0 in dotted-decimal notation (24 ones and 8 zeroes). Above we mentioned the idea to only have 126 hosts in each subnet, but more subnets. This could be achieved easily by using 25 bits for network plus subnet part of an address. If we need more hosts per subnet, and fewer subnets, then maybe 23 bits works better.

Subnets and LANs

So the subnet mask can tell us which part of an IP address identifies the network and subnet, and which part identifies the device inside the subnet. But how is this information used in real networks? There's an important link between subnet masks and the lower layers of the Internet model. In general, **each subnet corresponds to a single LAN**. That means that all devices *inside* a LAN, i.e., typically those devices connected to the same switch, belong to the *same* subnet, while all devices *outside* of that LAN belong to a *different* subnet.

The combination of IP address and subnet mask therefore enables devices to make decisions about where to send a packet. Consider the following network, which consists of three LANs that are connected using two routers.



Let's assume that all the subnet masks are /24, i.e., the parts of the IP addresses that are highlighted in red identify the subnets. If the client 130.194.76.192 (top left) wants to send a packet to 130.194.76.191, it can compare the destination address to its own IP address using the subnet mask. The first 24 bits are identical (130.194.76), which means that the destination is in the same subnet. In that case, the client **sends the packet directly to the destination**.

A different case would be if 130.194.76.192 wanted to send a packet to 130.194.66.42 (bottom left). Now the first 24 bits are different! Since the destination is outside its own subnet, 130.194.76.192 must now **send the packet to a router**. Luckily, there is a router in the .76 subnet, so we can send it there, and it becomes that router's job to deliver it.

In the network shown above, the router has addresses in two different subnets, .76 and .40. Neither of these are the subnets of the destination (130.194.66.42), but the router knows that it can pass the packet on to the second router. The second router has an address in the .40 subnet, so it can receive packets from the first router, but it also has an address in the .66 subnet, so it can deliver our packet.

This kind of network organisation is quite common, where different LANs are connected using routers to a **backbone network**. The backbone is only used for communication between routers. It is typically implemented using very fast hardware (e.g., the LANs could be running 1 Gbps Ethernet, while the routers communicate via 10 Gbps Ethernet).

IP version 6

As mentioned above, IP addresses are 32 bits long. In theory, that would be enough to address 2^{32}, or 4,294,967,296, different devices. However, in practice quite a number of these addresses are reserved for special purposes and therefore not available for use on the Internet, and some addresses have been allocated but are not used. For example, we've seen above that a company may have bought a /16 network, which means that it could potentially host 2^{16}=65,536 different devices. If it "only" operates, say, 40,000 different devices, then 25,536 IP addresses remain unused.

In total, it is estimated that only half of the theoretically possible IP addresses are actually usable in practice. Now given that the global population is over 7 billion people, we don't even have close to one address per person. This is a serious problem, because it essentially means that we have run out of IP addresses. If you, as a company, want to get an allocation of, say, a /16 network (like Monash has), someone else would essentially have to give up their allocation (this is facilitated through online registries

that handle allocation and deallocation of blocks of addresses).

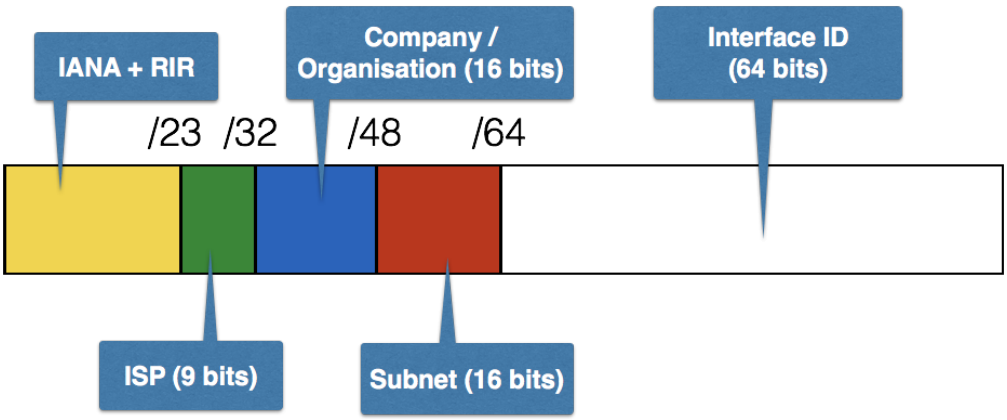
The big problem with the limited number of IPv4 addresses was recognised early, and in 1998 a new version of the Internet Protocol, version 6, was introduced. It increased the address size from 32 bits to 128 bits. So in theory, instead of having a bit over four billion different addresses, we now have this many with IPv6:

340,282,366,920,938,463,463,374,607,431,768,211,456

That's more than 340 undecillion. To put this number into perspective, it is at least 7 addresses for every atom inside every living person on earth, or 665,570,793,348,866,943,898,599 addresses per square meter of the surface of the earth.

Does that sound a bit excessive to you? Why didn't the designers of IPv6 go with 64 bits instead of 128? The rationale here is to make it very easy to allocate addresses and whole subnets, without having to worry about running out of address blocks.

Here's what a typical break down of the allocation of an IPv6 address would look like:



The first 23 bits would identify a particular *Regional Internet Registry* (RIR), which may be responsible for allocating addresses in a certain region such as North America or Europe. The next 9 bits could then identify an *Internet Service Provider* (ISP). When an ISP requires a new block of addresses, it asks the RIR for a new sequence of 9 bits. The ISP has to use the first 32 bits (23+9) it was given by the RIR, but it can then freely choose the next 16 bits to identify an individual customer, for example your company that wants to operate its own network. Note that this would allow an ISP to have over 65,000 customers (using 16 bits), and if that is not enough, it could simply ask for another prefix of 23+9 bits from the RIR. Each customer now has a fixed prefix of 48 bits (23+9+16), and can use the next 16 bits to identify subnets within its organisation. That means every customer is allowed to run over 65,000 different subnets!

Finally, the whole second half of the address, the remaining 64 bits, is reserved to identify a device inside its subnet (the so-called Interface ID). Having so much space for the Interface ID is useful because in IPv6, every device can use many different addresses for the same interface. This is useful for setting up the device in the first place (you can simply guess a random Interface ID and it's unlikely that anybody else already guessed the same one). But it can also help with anonymity, because a device can use different Interface IDs for different connections, making it harder to track a user.

Given the current rapid growth of the Internet, IPv6 is clearly required. There are some workarounds for reusing IPv4 addresses, e.g., there are tricks that make it look as if all of your devices at home to use the

same IPv4 address, so that your ISP only needs to allocate you one address when you connect. However, in the medium term, IPv4 will have to be replaced by IPv6. We won't go into the details of how that can be achieved, but you should at least note that IPv6 is a completely different network layer protocol than IPv4. This means that all devices from the sender to the receiver (including all routers in between) will have to understand the new protocol. If the server you want to connect to only has an IPv6 address but you don't, you won't be able to contact that server. And conversely, if you only have an IPv6 address but a server only runs IPv4, again you can't connect. That's why it takes a while for the entire Internet to switch over. In fact, many servers, routers and clients nowadays run both versions simultaneously.

Address Resolution

You have now seen three different types of addresses. At the Application Layer, we use human-readable addresses such as URLs (for web sites) or email addresses. At the Network Layer, IP addresses (version 4 or 6) identify a device (or rather, one of its interfaces). And at the Data Link Layer, MAC addresses are used to send messages within a Local Area Network.

The question is now how we can map from a higher-layer address to a lower-layer address. E.g., when your web browser needs to connect to `www.google.com`, how does it find out the corresponding IP address? Or when a computer needs to send a packet to its gateway router (in the same LAN), how does it find out the router's MAC address? The step that maps from a higher-layer address to a lower-layer address is called *address resolution.*

The Domain Name System

The Domain Name System, or DNS, is responsible for mapping human-readable addresses to IP addresses. It is implemented as an application-layer protocol, where clients send a *DNS request* to a special DNS server, which would basically contain the question "What is the IP address for x.y.z?", and then the DNS server would reply with one of the following options:

- An error message if no IP address was found for the given human-readable address
- The IP address of *another* DNS server that would be able to handle the request
- The IP address that was registered for the human-readable address in the query

The DNS system is implemented as a large, distributed database. This means that we don't have a single, central DNS server that everybody would use to make these request. Rather, there are a number of *root servers*, which can delegate requests to servers for each *top-level domain* (TLD, such as .edu, .au, .com and so on), which in turn delegate requests to servers for individual networks.

Let's assume a client wants to find out the IP address for `www.unimelb.edu.au`. It could ask one of the root servers, which may reply something like "Ask 37.209.198.5" instead. The client would then send a new request to 37.209.198.5, which may reply "Ask 203.21.131.1". And when the client then sends again a new request to 203.21.131.1, it may reply "The IP address for `www.unimelb.edu.au` is 202.9.95.188″. We call this method *iterative* DNS, because we iteratively ask servers from the root down until one of them knows the answer or can tell us that the name doesn't exist.

Alternatively, in a *recursive* DNS lookup, the server that a client asks will perform the lookup through all the hierarchies on behalf of the client. This may have the advantage that certain common addresses can be cached by the DNS server, which would reduce unnecessary network traffic.

Note that a DNS lookup can return both an IPv4 and an IPv6 address for the same name. If the client only supports IPv4, it would then connect to the IPv4 address, otherwise it may connect to the IPv6 address

instead. This is one of the mechanisms that was introduced to make sure that the IPv4 to v6 transition can happen without most users even noticing (because they still use the same human-readable application layer addresses).

Mapping IP addresses to MAC addresses

The second address resolution protocol we are going to cover here can map an IP addresses the corresponding MAC address. This is only required within a LAN. In fact, it is not possible to find out the MAC address of a device outside of one's own LAN!

Consider the following scenario for IP-to-MAC mapping. You have just connected your laptop to a WiFi network. Your laptop now has its own IP address and subnet mask (let's assume 192.168.1.10/24), and it knows the IP address of its gateway router (let's say 192.168.1.1). Your email app is open, and it now tries to connect to the mail server using POP in order to check for new mail. The IP address of the mail server is outside of your LAN, so your laptop has to send the POP packet to the router, which will then make sure it gets delivered to the correct destination LAN.

Recall that in order to send the POP packet to the router, your laptop needs to know the router's MAC address. The MAC address is required whenever we want to send a packet to a device inside our LAN. But the laptop only knows the router's IP address. That's where the *Address Resolution Protocol* (ARP) comes in. The laptop will send an ARP request packet, which asks "Who has IP address 192.168.1.1?" But there's no central ARP server (as was the case for DNS). Instead, the ARP packet is sent as a *broadcast*, which means that it gets sent to all devices in the LAN. The router will receive the packet and send a reply with its MAC address.

The Address Resolution Protocol is in fact only used for IPv4. In IPv6, a different protocol called *neighbour solicitation* is used, but the main idea is very similar.

Routing

Let's now come to the main function of the network layer. A router is a device that is connected to multiple networks, and routing means to forward a packet from one network into another network. Routers determine how packets flow through the Internet, so that when your laptop sends a request to a web site in North America or Europe, the packets for the request (and the response) will find their way to the correct destination.

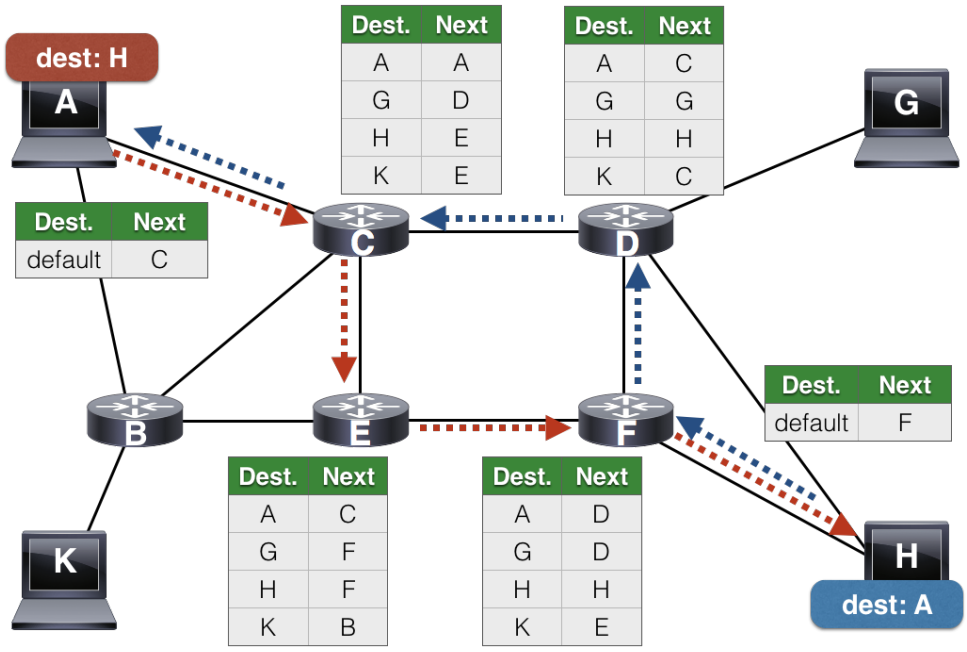
Since the Internet is nothing but a *network of networks*, routers are probably the most important device on the Internet, because without them, we would just have a huge collection of individual networks that cannot communicate with each other.

A router is a true Network Layer device. Each of its interfaces has an IP address, usually one per subnet that the router is connected to. A client sends a packet to a router if the destination is outside of the client's own subnet. A router then looks at the destination IP address of the packet and decides where the packet is sent next. That could be another router, or it could be the actual destination if it is located in one of the subnets that the router is connected to directly.

Routing Tables

Routers use *routing tables* to make the decisions where to send packets. A routing table contains entries for different networks, and for each network it would tell the router which other router can handle that

network. Here's an example of a network with five routers and their corresponding routing tables:



Take a look at the red packet that is sent from laptop A (top left) to laptop H (bottom right). The laptop itself has a routing table that just contains a *default gateway*, i.e., the router it needs to use for any packet whose destination is outside its own LAN. So A sends the packet to router C. The router can now look up the destination (H) in its routing table, and which tells it to send the packet to router E. This process continues, E sends it to F, and F can deliver it directly to H. If H now replies (the blue packet addressed to A), the reply may take a different path through the network.

Real routing tables of course can't contain a list of all potential destination addresses (otherwise routers would have to know about every single computer on the Internet!). Instead, they map entire *networks* to destination routers. For example, assume that laptop H has the IP address 130.194.66.43. We saw above that 130.194.X.Y identifies a Monash computer. So router C may just have an entry in its routing table that says "Any packet with the prefix 130.194 should be sent to router E". Router E may then look at the next 8 bits and say "Any packet with prefix 130.194.66 should be sent to router F, but packets with prefix 130.194.65 should go to router B". That way, routers use the hierarchies inside the IP addresses to make decisions. This is an important reason for building these hierarchies, because they help keep the routing tables short and the routers fast!

Static routing

Of course routing tables don't just appear in the routers, we need a mechanism to set them up. The simplest possible mechanism, called *static routing*, just means that a router has a fixed routing table that was set up either by a human operator, or by some remote configuration protocol. For example, the router you may have in your ADSL modem at home has a very simple routing table: forward everything to the Internet Service Provider. In these very simple cases, a static routing table is sufficient.

But consider the case of a router in a complex network like the one at Monash. The routing tables would have to contain lots of entries, and every time the network structure changes, e.g. when a new computer lab or a new server room is added, or when a particular part of the network becomes temporarily unavailable, they would have to be updated. This would be almost impossible to manage by hand. Therefore, routers can configure their routing tables automatically, using so-called *dynamic routing*

protocols.

Dynamic routing

In dynamic routing, routers "talk" to each other, they exchange information so that they can build accurate routing tables automatically, and change the tables dynamically when the network changes.

There are essentially two different types of dynamic routing protocols.

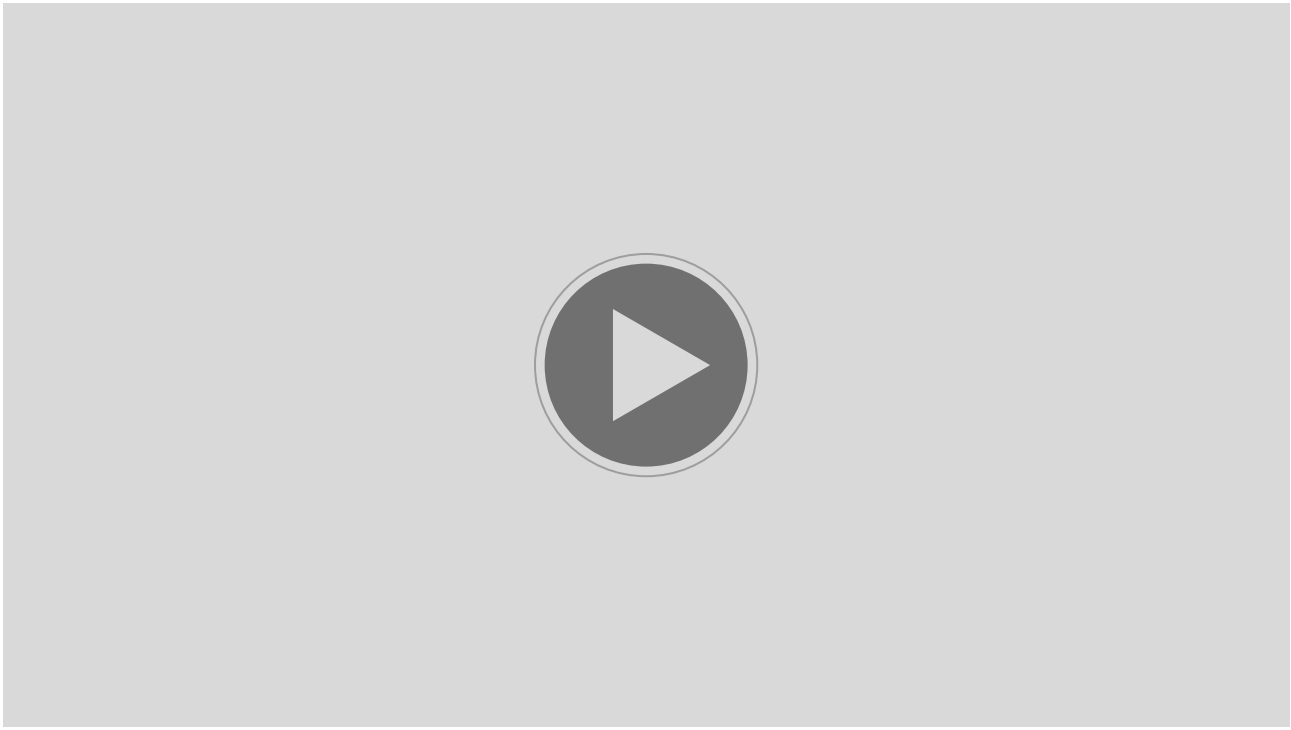
Routers that use **distance vector routing** protocols exchange information about the *distance* to a destination network and the *target router* for that network. Distance is typically measured just by counting the number of routers that a packet would have to go through until it reaches its destination. A router would then choose the path with the fewest such "hops" through other routers. Examples of distance vector protocols are EIGRP (Enhanced Interior Gateway Routing Protocol), RIP (Routing Information Protocol), and the very important BGP (Border Gateway Protocol), which we will look at again a little later.

The alternative to distance vector routing is **link state routing**. Here, instead of just exchanging information about the distance, routers also include information about the *quality* (or state) of a link. This has the advantage that a router could choose a path that requires more hops but uses a faster or more reliable network. The most common link state routing protocol is OSPF (Open Shortest Path First). We will not go into any more detail on link state routing.

Distance vector routing example

A typical implementation of distance vector routing, as used by e.g. RIP, is for routers to simply send their routing tables to all their neighbours. Initially, the routing tables would just contain the networks that a router is immediately connected to. This is easy to set up by a network admin. When a router *A* then sends this table to a neighbouring router *B*, the neighbour learns about which networks *A* can reach, and if any of those networks isn't present in its own routing table, it can simply add it. In addition, routers also keep the distance (in number of hops) to each network. So if *B* already knows a route to a destination network, but *A* knows a shorter one, *B* can update its routing table. After a little while, all routers have *converged*, which means that they have exchanged all the available information about the network.

The animation below shows how routing tables are exchanged.



(https://www.alexandriarepository.org/wp-content/uploads/20170516151501/RIP.mp4.mp4)

7.7 Transport Layer

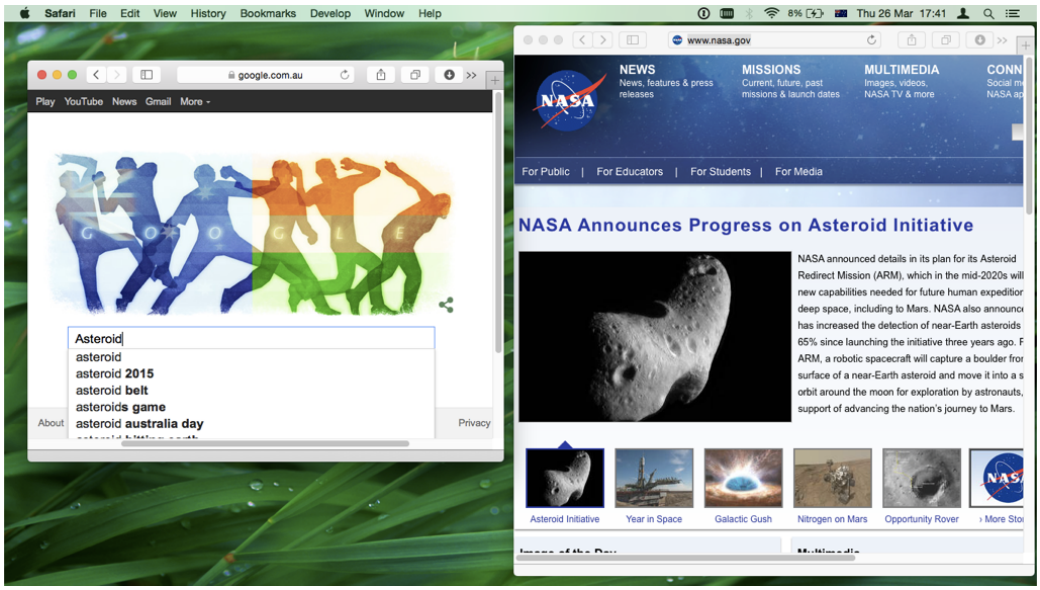
When we look at the data link and network layer, and compare the messages that those layers can handle with the typical requirements of an application, there's a big gap. The data link and network layer handle individual, short frames or packets, each typically around 1,500 bytes long. Furthermore, there is no error correction at those layers, so if a frame gets "damaged" (i.e., some bits in the frame are transmitted incorrectly), or simply doesn't arrive at all, the information it carried is lost. Now compare that with want we require at the application layer. An application, e.g. a web server, should be able to send a very long message (e.g. containing a video) to a client, without having to worry about how to transfer it using short (1,500 byte) packets, and without having to deal with packets not arriving.

This gap is filled by the transport layer. In particular, the *Transmission Control Protocol* (TCP) is responsible for providing a **virtual circuit**, i.e., to create the illusion of a reliable, point-to-point connection between two applications. To achieve this, TCP automatically splits up application layer messages into short *segments*, makes sure that the segments arrive correctly, and reassembles them in the correct order into the original message at the destination.

TCP is used by many major application layer protocols, including HTTP, SMTP, IMAP and SSH.

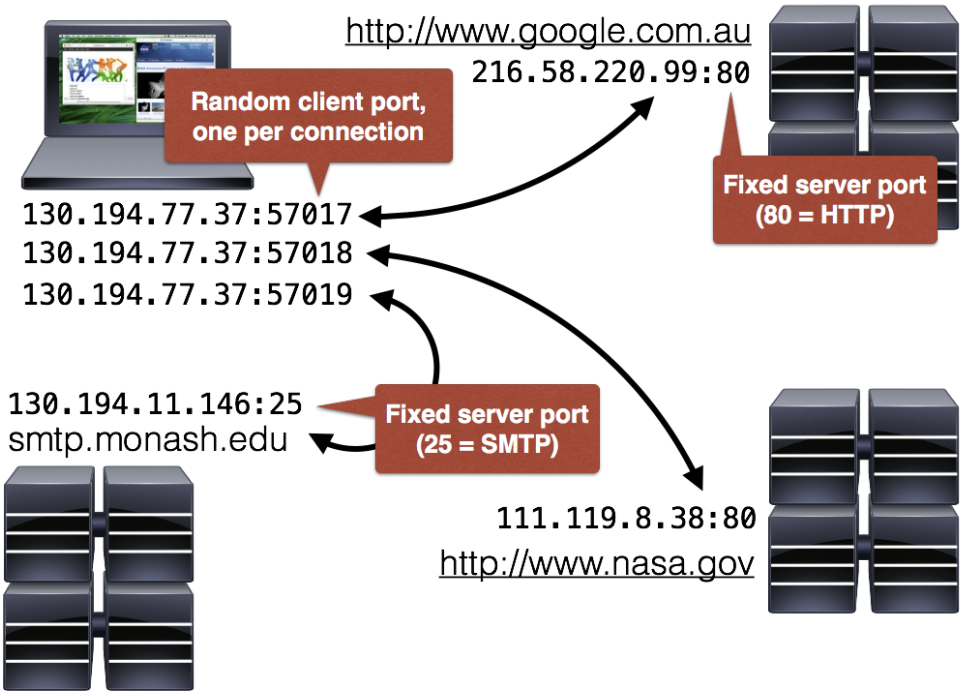
Addressing applications

One thing we only mentioned briefly above is that TCP provides a virtual circuit between *applications*. So far, we have only seen that the data link and network layer connect different *devices*. The screen shot below illustrates the difference:



The screen shot shows two web browser windows, one with the Google home page, the other with a web page from the NASA web server. But how does the computer know that an incoming packet from the NASA server belongs to the page in the browser window on the right and not the other one? From what we've seen so far, the packet just contains a destination IP address, but the IP address only identifies the client, not which application it is for.

So we need an address at the transport layer, too. Each application has a *port number*, which together with the IP address, lets us uniquely identify a connection between a server and a client. The diagram below illustrates this in a bit more detail.



Have a look at the connection between the client (top left) and the Google server (top right) first. The client's IP address is 130.194.77.37, and the server's address is 216.58.220.99. In order to identify the left browser window (the one that shows the Google page), the client picks a random port number (and remembers that it belongs to the left browser window), in this case 57017. On the other side, we use the "well-known" port number 80, which identifies a web server application. This tuple of four numbers (130.194.77.37, 57017, 216.58.220.99, 80) now uniquely identifies the *connection* between the browser window and the web server. If we open another Google window on the same client, it would get a different port number. If we open a browser window for a different web site (e.g. the NASA site), it would have a different destination IP address and local port number.

Similarly, on the server, the port number identifies the service we want to use. For instance, we could have both a web server application and a mail server (SMTP) application running on the same computer. Both would be reachable via the same IP address, but the web server uses port 80, while SMTP uses port 25.

TCP error control and session management

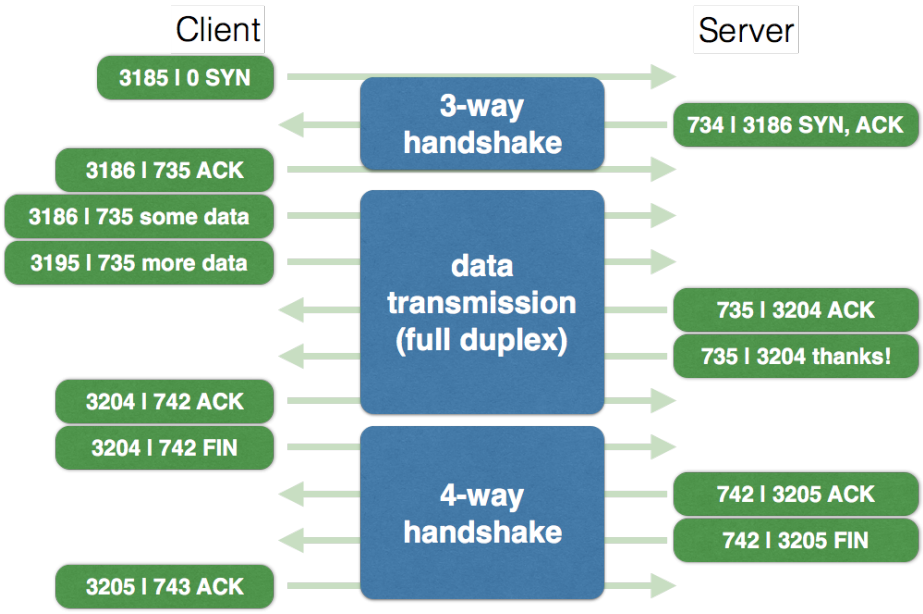
The missing piece in the puzzle is now how TCP sets up a *reliable channel,* by splitting up large application layer messages into short chunks that can be sent over the network layer, and by making sure these packets arrive correctly.

We have actually seen the basic mechanism for this before in the module on the Data Link Layer. It is called Automatic Repeat Request (ARQ), and it simply means that every packet must be acknowledged by the receiver, and if it isn't acknowledged within a certain time-out, the sender will send it again.

TCP implements ARQ using two numbers that are sent with every packet. Let's say *A* is sending a TCP

packet to *B*. The *sequence number* of the packet is the number of bytes that *A* has already transmitted to *B* (not including the packet that is now being sent). The *acknowledgement number* is the number of bytes that *A* has received from *B* so far. Using these two numbers, both *A* and *B* can continuously check whether everything they have sent has been received correctly.

A typical TCP session between a client and a server consists of three phases. First, in the "three-way handshake", client and server exchange sequence numbers to set up the connection. In the second phase, client and server transmit the actual data (such as the web page or an email). Finally, the third phase is the "four-way handshake" which cleanly closes down the connection. We will use the following sample session to explain this in a bit more detail.



The three-way handshake starts with the client sending a special "SYN" (synchronise) packet to the server, with a completely random sequence number (here we picked 3185) and an acknowledgement number of 0. The server replies with a special "SYN, ACK" (synchronise and acknowledge) packet, choosing its own random sequence number (here 734), and using the client's sequence number plus 1 as its new acknowledgement number (here 3185+1=3186). Finally, the three-way handshake ends with the client sending an "ACK" (acknowledge) packet back to the server to indicate that it received the "SYN, ACK" correctly, again swapping the sequence and acknowledgement numbers around and adding 1 to the server's number (734+1=735). Now we're set up for the actual data transfer: both client and server know each other's sequence numbers.

During the actual transmission, both client and server can send data at any time. In this example, the client sends a packet containing the data "some data". Immediately afterwards, it sends another packet with the data "more data". Note how in the second packet, it incremented the sequence number by 9. That's exactly the number of bytes it sent in the first packet ("some data" is 9 ASCII characters). As the next step, the server decided to acknowledge that it received the two packets correctly. It swaps the sequence and acknowledgement numbers, and adds the number of bytes it has received. Beginning from the first packet (sequence number 3186), it has received 9+9=18 bytes, so it uses 3186+18=3204 as the acknowledgement number. That way, the client knows that all the data has been received correctly. The server now sends a message to the client ("thanks!", 7 bytes), which the client acknowledges by replying with the acknowledgement number of 735+7=742.

At this point, both server and client are finished and don't want to send any more data. One of the two, in this example it's the client, sends a "FIN" packet (finalise) to the server, which the server acknowledges.

The server also sends a "FIN" to the client, which again is acknowledged, at which point the connection is considered to be closed.

On the Internet, it is possible that packets don't arrive in the order they were sent (e.g. if two packets happen to be sent via different routes, or if a packet had to be sent again). The sequence numbers allow a receiver to puzzle them back into the correct order, because they identify exactly the position in the stream of data where a packet fits in.

TCP parameters

Since TCP is splitting application layer messages into short segments, there are two important parameters that can have a big effect on the efficiency of the transmission:

- How large should a segment be?
- How fast should we be sending segments?

The maximum size of a segment should ideally be determined by the *shortest* frame length on the entire path between sender and receiver. Imagine we have a sender and receiver that are connected via a range of different technologies. Let's say the sender is connected to its LAN via Ethernet, to the Internet via optical fibre, then via ADSL to a home network, and from there via WiFi to the receiver. Each of these technologies may have a different maximum size of frame that it can deliver. For example, Ethernet typically limits frames to around 1500 bytes, while frames on WiFi may be longer. We call this the *Maximum Transfer Unit* (MTU) of the path from the sender to the receiver.

So how can the sender know the MTU along the entire path? There are two approaches to solve this problem. The first one is to simply pick a "reasonable" size, e.g. assume that the whole path can accept 1500 bytes per frame. If any of the networks on the path has a lower MTU, the network layer will in fact "fragment" the packet on that link (we didn't cover that in the Network Layer module, and it is not an ideal solution). The second, better solution is to use a mechanism called *Path MTU Discovery*. It requires routers on the path to send error messages back to the sender if a packet is too long. The sender can therefore increase the packet length until it receives error messages, and then decrease the size again to the level that didn't cause errors. Path MTU Discovery therefore lets the sender find out the ideal packet size over the entire path.

The second parameter, how fast to send frames, can also be important. Imagine you are downloading a video from a Monash server using a slow ADSL connection at home. If the TCP software on the server was sending the segments at the fastest possible rate that the Monash network supports, the frames would leave the Monash network at around 1 Gbps. But now they arrive at your local telephone exchange where they have to be sent through your 10 Mbps ADSL connection. The frames can't be sent fast enough, so the ADSL modem in the telephone exchange will start dropping frames at some point (because it usually doesn't have enough memory to store them temporarily). The server would have to send a lot of frames again (because it didn't get an acknowledgement from your TCP software!), causing unnecessary network traffic. To solve this problem, the sender will start sending frames at a slow rate, waiting for an acknowledgement after every frame before sending the next. Then it will send two, then four, then eight frames before waiting for an acknowledgement. At some point, it will not receive an acknowledgement (because it was sending too fast), so it will drop down to a slower rate again.

Both methods together allow the TCP software in the sender to adjust to the network characteristics dynamically.

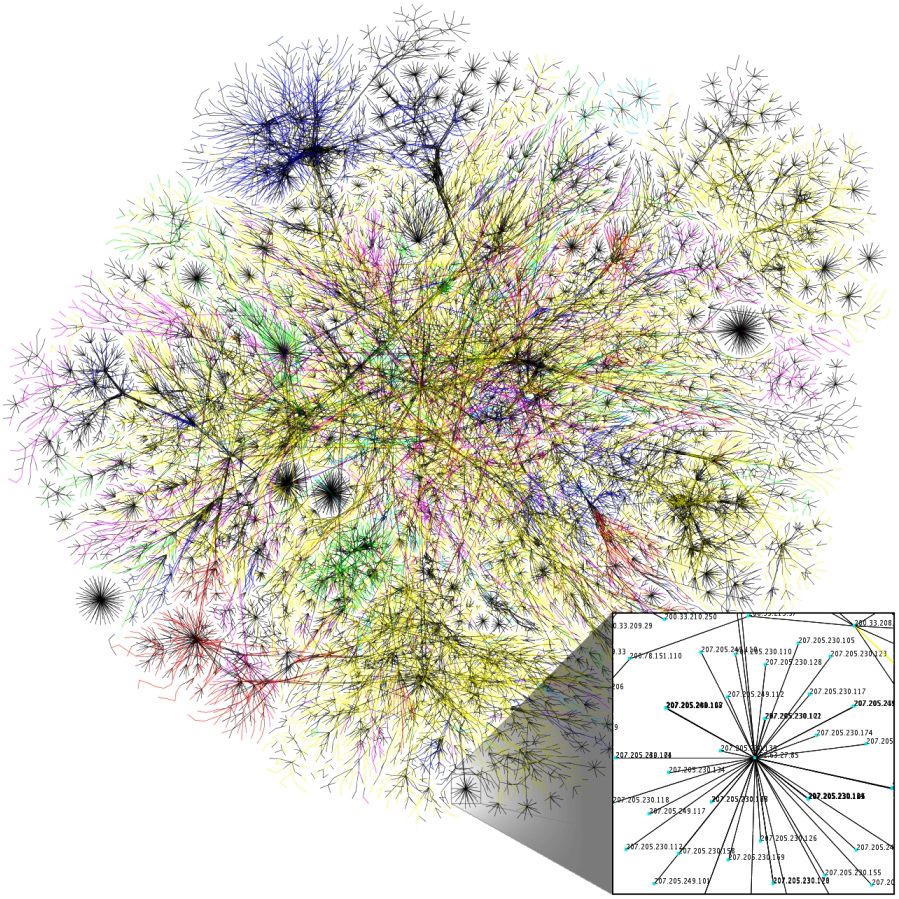
7.8
The Internet

We've now seen all the basic technology, so let's put everything together to understand how we can use switches, routers, servers and clients to construct the Internet.

A Network of Networks

The Internet is simply the collection of all the devices that are running the TCP/IP protocol suite and that are connected via routers. It does not exist as something separate from e.g. the Monash University network, but the Monash University network is simply *part of* the Internet. In the same way, your network at home is not really connected to the Internet, but it becomes part of it.

The scale of the Internet is truly impressive. It consists of millions of networks and billions of devices. It only works because all of these devices use the same underlying set of protocols and standards.



A (partial) map of the Internet in 2005. By The Opte Project (CC BY 2.5), via Wikimedia Commons

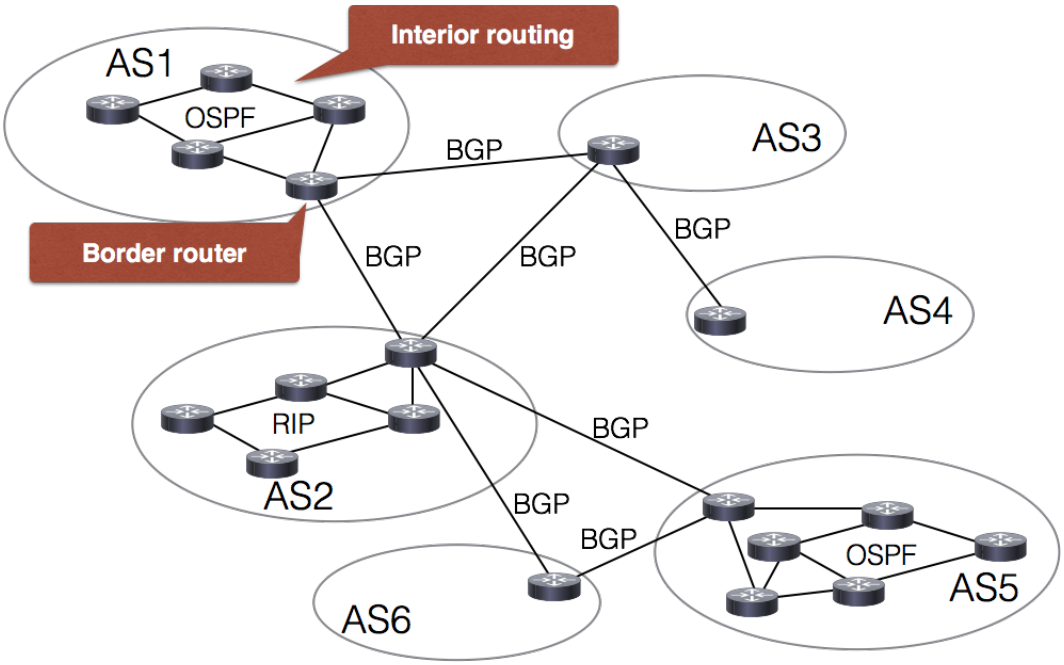
Autonomous Systems

We've already seen in the Network Layer module that routing is based on *hierarchical* addresses, in order to cope with the size of the network. At the highest level, the Internet is made up of *Autonomous Systems*

(AS), which are networks that are operated by a single organisation. For example, Monash University is operating an AS, and each Internet Service Provider is operating an AS.

Splitting the Internet into Autonomous Systems is important for routing. Whenever a packet is sent from a device *A* to another device *B* that is part of the same AS, it is routed *internally,* i.e., the packet doesn't ever leave the AS. Only when a packet destination is part of a different AS will the packet be routed *externally*, via a so-called *border router,* which connects an AS to one or more other ASs.

Here is a diagram that illustrates this concept:



The routers within each AS can use whatever routing protocol the system administrators prefer. E.g., the AS in the top left corner uses OSPF as its interior routing protocol, while the AS below it uses RIP. But importantly, all border routers, which connect an AS to its neighbours, must run the *Border Gateway Protocol* (BGP) for *exterior routing*. Apart from TCP/IP, BGP is therefore a cornerstone of the Internet architecture.

Internet Structure and Peering

As you can see in the diagram above, different Autonomous Systems need to be connected to each other. Most users access the Internet through an Internet Service Provider (ISP), a commercial company that provides access to its AS via technology such as ADSL, 4G (mobile phone networks), cable modems, or the NBN. Now of course the ISP's network must be connected to other Autonomous Systems, so that you as a customer can send packets to any other device on the entire Internet.

Some ISPs connect to the rest of the world through other, larger ISPs. For example, AS4 in the diagram above could be a small, regional ISP, and all its traffic is routed through the network of the larger ISP that operates AS3. In that case, the ISP that operates AS3 would charge the smaller ISP a fee for forwarding the traffic. We would say that AS3 is a *Tier 1* ISP, while AS4 is a *Tier 2*.

An alternative to this approach is for several ISPs to connect to each other at a so-called *Internet Exchange Point* (IXP). For example, we could assume that AS2, AS5 and AS6 are all connected to each

other, and simply agree to carry each other's traffic for free (assuming a similar amount of traffic flows in all directions). This is called *peering*.

An IXP simply provides the hardware for multiple ISPs to connect. This hardware essentially consists of a very fast switch! Each ISP is operating their own border router, and e.g. all three border routers of AS2, AS5 and AS6 in the example above would be connected to the same switch, so that they can freely route packets to each other's networks.

Scaling up the Internet

From its inception in the early 1970s, the Internet has grown to a size that must have been completely beyond the imagination for the inventors of the TCP/IP, BGP, or HTTP protocols. Not only the number of devices connected to the Internet has grown, but also their distance, both geographically and in terms of the number of routers on the path between them. At the same time, network technology and computing power has increased immensely, and with it the demand for fast network access by billions of users. Many applications in fact rely on low latency, such as real-time video and audio conferencing, while others benefit greatly from low latency. For example, fast page load times have been shown to improve the "conversion rate" of web users, meaning people buy more products online if the web page loads quickly!

The original Internet protocols, in particular HTTP, TCP, IP and BGP, were not designed for a network of this size and very low latency requirements. In fact, they perform quite poorly. So why does the Internet still work, how has it been able to cope with the massive increase in size and demand? We will only look at three techniques here that have helped mitigate the problems with the original set of protocols: *load balancing, content caching* and *content delivery networks.*

Load Balancing

Many services on the Internet are used by so many people at the same time that it would be impossible to implement the service using a single server computer. For example, one estimate for the bandwidth used by Netflix was that its users stream on average 10 terabits per second. That is 1000 times faster than the usual 10 Gbps fast Ethernet networks, so even if we could build a server that was capable of producing 10 Tbps streams, we couldn't deliver the data using any existing networking technology. The same problem arises for basically any large Internet company (e.g. Google processes more than 40,000 queries per second, Twitter manages thousands of tweets per second, etc.).

The obvious answer to this problem is of course to use multiple servers to split up the load. These servers will be located in multiple data centres to also achieve a better geographic balance. But who makes the decision which server will handle a request when a user types `netflix.com` into their web browser?

The easiest solution is to use DNS-based load balancing. The idea is simply that different people will get a different answer to the question what the IP address is for the name `netflix.com`. Here is a little experiment that shows the different IP addresses that were returned for the address `www.google.com` based on where the request was made:

Request	Response
Inside the Monash network	216.58.220.132
From Optus network	74.125.237.209
From Germany	173.194.112.176
From France	74.125.21.105

As you can see, different users are simply pointed to different servers, which will achieve an automatic

load balancing across the globe. Additionally, the DNS servers can make sure that you access an IP address that is geographically close to you, to reduce latency.

Load balancing can also be implemented by special devices inside the network of the company that operates the servers. In that case, all requests enter through a single server (all through the same IP address), which then distributes the requests to other servers. This is used when the amount of network traffic is not too high to be handled by a single machine, but the processing time required for each request would be prohibitive.

Content Caching

In addition to pointing a user to a geographically close server via DNS, a router on the path between a client and a server can also store a copy of frequently requested documents (such as web pages or images). This is called *content caching*. It is completely transparent to the client, but it can improve page load times significantly.

Content caching is in fact explicitly supported by the HTTP protocol. The HTTP standard defines that a GET request cannot make any modifications on the server and may therefore be cached. When a client makes a GET request, the server may provide in its response an additional header field `Expires:` that states how long the document could be cached (e.g., if it changes frequently, the server could say that it expires in one minute, but if it is just a company logo that appears on every page, maybe it could be set to expire in one month). The caching router on the path can then store the document, and if it sees a future request for the same document before it has expired, it simply returns its local copy rather than forwarding the request to the actual server.

This works particularly well for static web content such as CSS style files, JavaScript programs, or images.

Content Delivery Networks

Simple load balancing, where all requests end up in a single data centre, doesn't solve the latency problem, because all requests and responses still need to travel through the Internet. Some companies identified this as a business opportunity and started building lots of data centres close to major urban centres. The data centres are connected via a very high speed dedicated network, so that data between the data centres doesn't need to be routed through the Internet. The resulting system is called a *Content Delivery Network* (CDN), because it can deliver content such as web pages and movie streams very efficiently to the end users. Companies that operate CDNs then sell access to their systems to content providers, for which it is a great advantage that they don't have to build up their own worldwide system of servers close to their customers.

CDNs typically work very closely with IXPs, and allow any ISP to peer with them. For end users, this has the advantage that when they access a web page located on a CDN, there will only be a single router between them and the content, resulting in very low latency and very fast delivery. For ISPs, it has the advantage that they don't have to pay for the traffic being routed through their upstream Internet provider (in the case of a Tier 2 ISP), and that they can offer their users a better experience. In fact, some ISPs can offer "free" or "unmetered" streaming of certain services (which means that downloading e.g. certain media content does not count towards the monthly download allowance), because they don't incur any additional costs due to the peering arrangements.

It is estimated that over 50% of all traffic on the Internet is handled by CDNs.

8
Security

8.1
Introduction to Cryptography

What is *Cryptography* (a definition)

The Merriam Webster Dictionary defines cryptography as

> *1*: secret writing
> *2*: the enciphering and deciphering of messages in secret code or cipher; *also* : the computerized encoding and decoding of information

a more restricted view at cryptography would define it as "the coding and decoding of secret messages" (Merriam Webster Student Dictionary).

In IT, we now actually have a much broader definition for cryptography. Lets look at how one well-known cryptography expert defines it:

"Historically, cryptography arose as a means to enable parties to maintain privacy of the information they send to each other, even in the presence of an adversary with access to the communication channel. While providing privacy remains a central goal, the field has expanded to encompass many others, including not just other goals of communication security, such as guaranteeing integrity and authenticity of communications, but many more sophisticated and fascinating goals. (Mihir Bellare)"

Indeed, cryptography is much more than hiding messages from eavesdroppers. However, keeping messages or data secret is still one of the main uses of cryptography.

A little bit of history

The need to keep messages or written text secret has probably been around as long as languages exist and early mechanisms to hide messages have been developed just a short time after writing was invented. There is documented use of cryptography for at least 3000 years, but early (mostly mechanical) ways of hiding and recovering messages have been around much earlier.

Cryptographic methods used before 1900 are fundamentally different from what we use now. With our current knowledge and computing power they are really easy to break. However, they are easy to understand and can provide a good basis to learn about cryptography. A very simple example of a code is ROT13, which just replaces each letter of the alphabet with the letter 13 places after it, wrapping around and starting from the beginning for letters in the second half of the alphabet. This transformation just hides a message from a casual observer, but will obviously not protect from any careful analysis.

Let's try ROT13. What is the cleartext (the actual decrypted message) for **qba'g cnavp** ?

Reveal [1]
ROT13 is an example of an *alphabetic cipher*. Each letter in the alphabet is just replaced by one other letter. The *key* is just this relation between the letters. This type of cipher (with different types of alphabetic replacements, of course) is called *Caesar cipher*, after the Roman leader Gaius Julius Caesar (100BC - 44BC) who apparently used this really simple for of encryption to protect written military

messages.

If you want to learn more about classical ciphers and try them out, you can use the online version of the Cryptool:

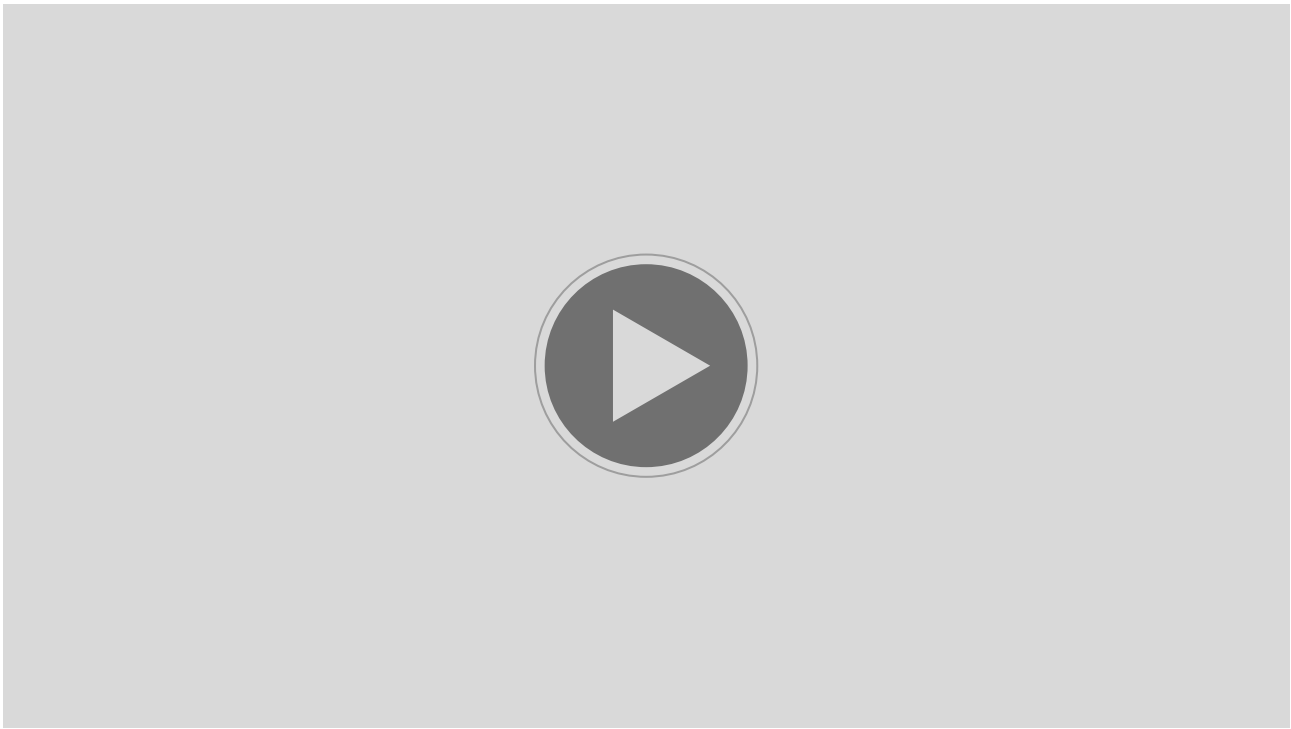
http://www.cryptool-online.org/

Some of them are much more complex and more difficult to break for humans. However, using computers, none of them can be considered secure.

Now lets have a look at the different basic concepts we use in cryptography today.

Symmetric Encryption

The main idea of a symmetric algorithm is to use the same key for encryption and decryption. All the classical ciphers are basically symmetric encryption algorithms. The following video shows an animation where the same key is used on both sides to first encrypt the message and then to decrypt it in order to get the original text.

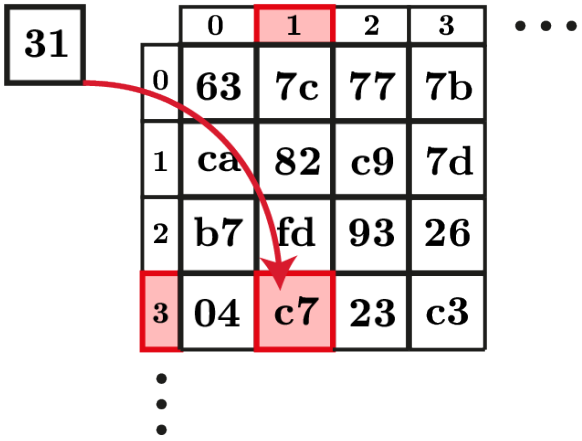


(https://www.alexandriarepository.org/wp-content/uploads/20170410115311/SymmetricCryptography.mp4)

In contrast to classical ciphers, modern symmetric cryptography will not individually encrypt single characters, but take blocks of a message (e.g. 128 bits or 256 bits) and scrambles these blocks in a way that each change in the original block will have an effect to the complete encrypted block and that there are no detectable dependencies between individual bits in the block and specific bits in the encrypted block. In principle, most symmetric algorithms work with a mix of substitutions (similar to a classical cipher) and permutations. Substitutions are defined by so-called S-boxes.

S-boxes and permutations

Lets look at a small piece of a block consisting of 8 bits, i.e. a two digit hexadecimal number. Now, the S-box is like a look-up table that has the first hex digit on the side and the second digit on the top. These two digits are replaced by the value found in the table where the row and the column intersect. The following picture shows a small part of such an S-box. The hex number 31 would be replaced by c7.



Then, the resulting substituted message is permuted, i.e. the order of the bits is changed. Then, another round of substitutions using the next part of the key is applied etc.

AES - The Advanced Encryption Standard

Towards the end of the last century NIST (National Institute for Standards and Technology, U.S.) had started an open selection process for the next generation of secure symmetric encryption. NIST asked experts, researchers, consortia, worldwide to submit proposals for a secure and efficient new symmetric encryption algorithm. In the final round, 15 designs were openly evaluated and in October 2000 the Rijndal algorithm by Jon Daemen and Vincent Rijmen was chosen to become the new Advanced Encryption Standard AES. The symmetric key for AES is currently recommended for 128 bits. For the future, recommendations go to 256 bits keylength.

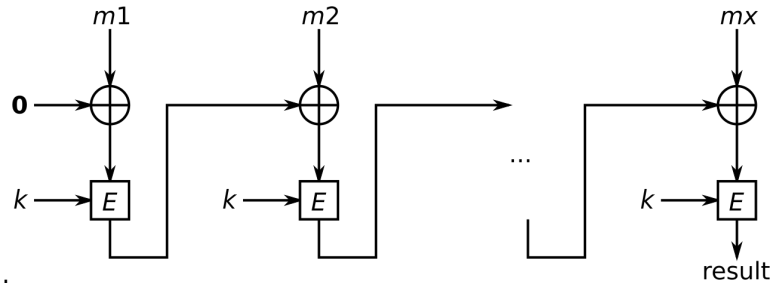
AES is the state-of-the-art algorithm that is now used in most applications and protocols.

As described above, AES works on message blocks. In its basic form (each block is individually ad independently encrypted) it provides confidentiality for the encrypted data. Only someone with the correct key can read the message.

Do you think AES also provides *integrity* of the message? Integrity means that the recipient can tell if the message was changed or if it is still the message that was originally encrypted by the sender..

Reveal Integrity [2]
In addition to the straightforward problem with the blocks, also changes of single bits are difficult detect if there is not enough redundancy in the message. Changes in a few bits will totally change the complete block. However, if the data is just numbers, or any other unstructured information, such a change might go unnoticed. Therefore, an additional mechanism is used to provide integrity for AES encrypted data. This so-called Cipher-Block-Chaining creates a chain of blocks by using bit-wise XOR of the encrypted precious block with the next block. The following figure shows blocks m1,...,mx. First, m1 is encrypted. Then, bit-wise XOR is used to combine the result with m2. The result of XOR is the encrypted using key k

and the result goes into XOR with m3, etc. The final result will be send in addition to the actual AES encryption and can then be used to check if any changes to the sequence of blocks has been made.



.

This Cipher-Block-Chaining is an example of a so-called Message Authentication Code MAC. I basically takes a longer message (blocks m1,2,...,mx in the example) plus a secret key as input and outputs a much smaller (just the length of one encrypted block) digest that can be used to check integrity of the message. If you have enabled WPA2 encryption on your Wifi access point, you will probably use this kind of integrity protection.

Note that in addition to protecting messages there are many other uses for symmetric cryptography. One example is the encryption of file systems, complete hard disks or USB sticks.

Symmetric encryption is very efficient, does not increase the message size by too much, and has rather short keys. Nevertheless, there are a few problems with symmetric encryption. There are basically 3 disadvantages of symmetric encryption:

1. **Key distribution**: Somehow, one needs to establish a shared secret. An alternative secure channel for key distribution is necessary.
2. **Scalability:** Each pair of sender and receiver needs a unique secret key. The number of keys grows exponentially with the number of participants (12 participants need 66 keys, 1000 need 499,500 keys and a million participants need an unrealistic 499,999,500,000 keys if everybody wants to talk to all others)
3. **Non-repudiation is not possible:** For non-repudiation there needs to be some evidence who has actually generated a message. With symmetric keys, sender and receiver need to have the same key. Thus, all messages can in principle have been generated by both of them and cannot be attributed to on of the key owners.

Most of this was not a big problem when cryptography was used for individual messages on a small scale (e.g. for military communication during the second world war). However, with billions of devices on the Internet, electronic commerce, banking, digital contracts, etc. the situation is quite different. Luckily, some really smart minds came up with the idea of public key cryptography.

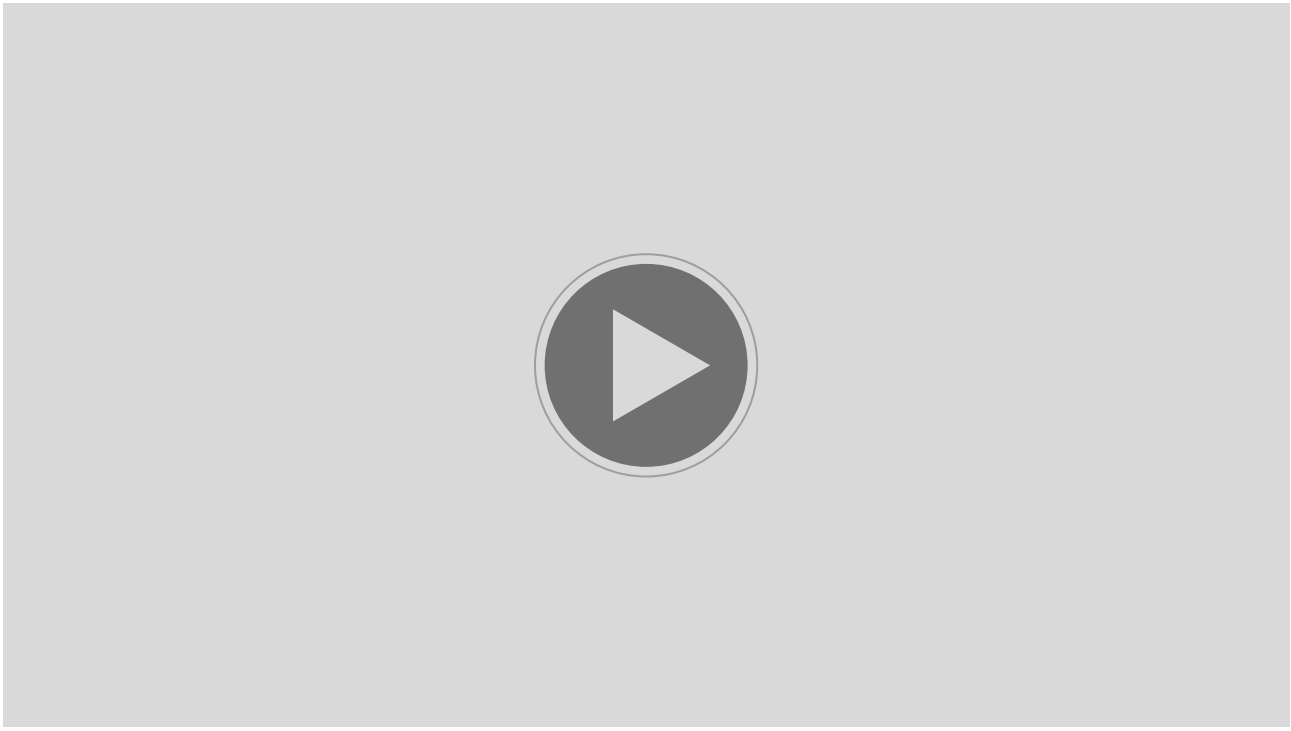
Public key cryptography

The main difference to symmetric cryptography is that instead of a single symmetric key there is now a pair of keys (the private/secret key and the pubic) used for encryption or for digital signatures.

In the early 1970s cryptographers developed the idea of "non-secret encryption". First (publicly known) practically usable schemes were developed in 1976 by Diffie and Hellman (influenced by Merkle) (known as Diffie-Hellman Key Exchange) and in 1978 by Rivest, Shamir and Adleman (known as RSA).

The general idea is to rely on a "hard" mathematical problem. Then, a large more-or-less random numbers with particular properties, a key-pair is generated, such that the private key cannot be derived from the public key without solving the underlying mathematical problem. Every principal owns a unique pair of keys.

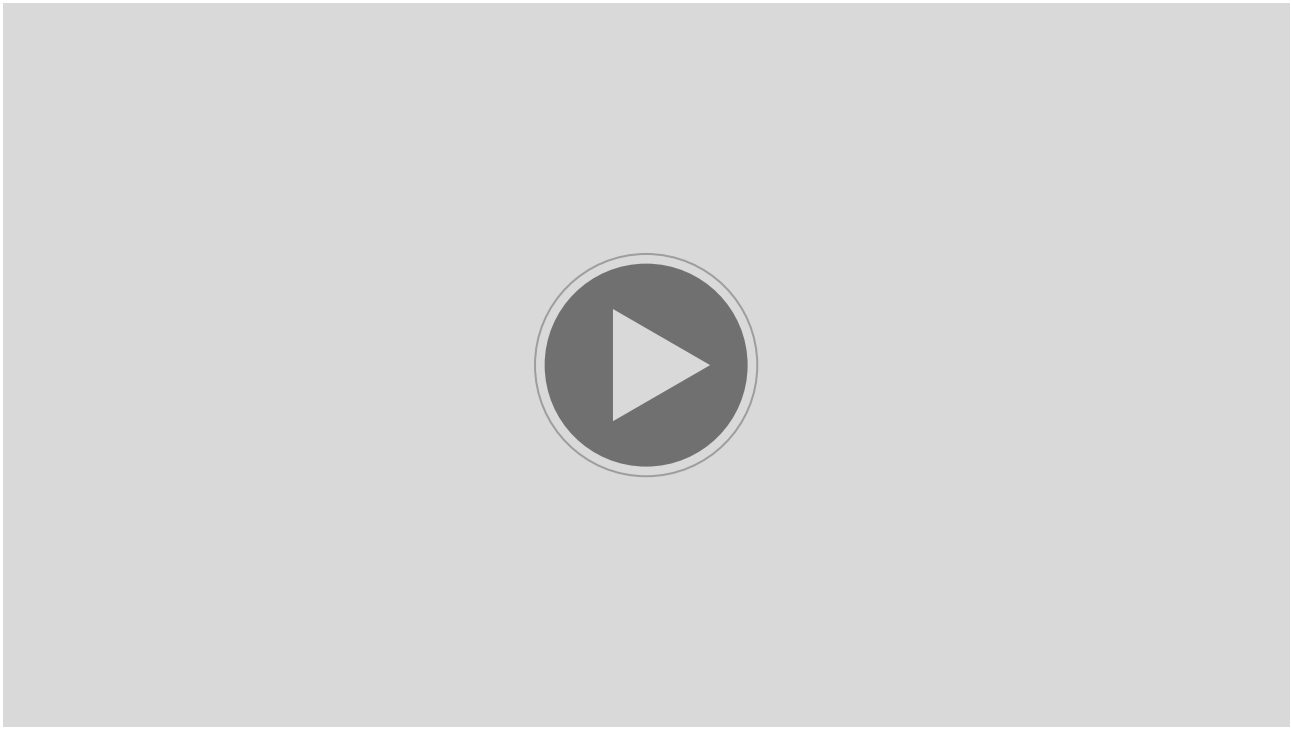
The following animations shows that the public key is given to the sender of the message (the green key) while the private key is kept by the receiver (the red key). In fact, the public key can be literally published. Then, the sender encrypts the message using the public key and the recipient can decrypt it using his private key.



(https://www.alexandriarepository.org/wp-content/uploads/20170410192551/PublicKeyCryptography.mp4)

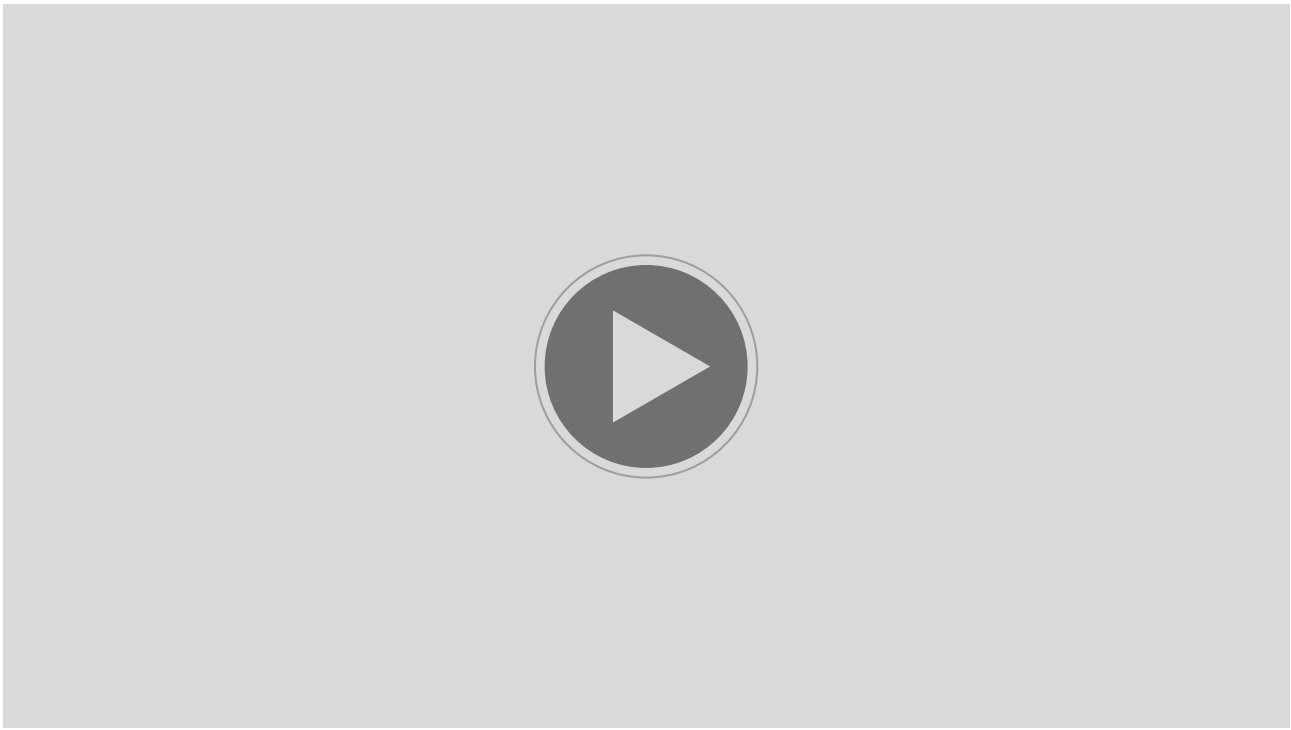
This way of encrypting messages solves the key distribution problem and the scalability problem. Only the public keys need to be distributed, but these do not need to remain secret. Further, each principal only needs a single key pair plus the public keys of all other principals. Thus, for 1000 principals we only need 1000 key pairs (2000 keys) instead of nearly 500,000.

The non-repudiation problem can be solved by the reverse application of the key pair. Now, the private key is used to produce a *digital signature* of the message. This digital signature is now send along with the original message. Everybody can now use the matching public key to check who has signed the message. This procedure is shown in the following animation.



(https://www.alexandriarepository.org/wp-content/uploads/20170410193646/DigitalSignatures.mp4)

Public key cryptography sounds really great. The only drawback is that key sizes are much bigger and encryption/decryption is much less efficient compared to symmetric key cryptography. Therefore, hybrid schemes are often used. In such a hybrid scheme, a key derivation function is used to generate a shared secret using two key pairs. The following animation depicts this procedure.



(https://www.alexandriarepository.org/wp-content/uploads/20170410194336/KeyEstablishment.mp4)

A very popular is public key scheme is RSA. It was developed by Ron Rivest, Adi Shamir and Leonard Adleman and first published in 1977. The mathematical problem is the factorisation of large numbers, i.e. finding prime factors for a large number. The following briefly explains key generation, encryption and decryption using RSA. □ □□□ □ means the remainder of □ divided by □

RSA key generation:
1. Generate two large prime numbers □ and □.
2. Let □=□∗□ and □= □(□)=(□−1)(□−1)
3. Chose a small number □ co-prime to □, i.e. □□□(□,□)=1 and 1<□<□
4. Find □ such that □∗□ □□□ □ =1
5. Publish □ and □ as public key and keep □ and □ as secret key.

RSA encryption:
For a message M calculate $enc(M) = M^e mod\, n$
RSA decryption:
$decr(enc(M)) = enc(M)^d mod n = M$

RSA key length:
Currently, 2048 bits is considered secure for RSA. Some agencies/government bodies recommend 3072 bits after 2020, others after 2030.

In addition to the actual cryptographic algorithms (e.g. the symmetric algorithm AES or the public-key algorithm RSA) there are a number of other components that are relevant for the secure realisation of cryptography. These are random numbers and cryptographic hash functions.

Random numbers

All types of cryptography need random numbers for example for

- Key generation: Nobody should be able to guess or predict a new key. Thus, it needs to be random.
- Use in protocols to mark messages or sessions as new.
- Initialisation vectors

Many attacks on cryptography have been based on badly generated random numbers.

Cryptographic hash functions

A hash function maps input of arbitrary length to a fixed length output. These mechanisms are used in many places in computing, e.g. for indexing in data-bases.
Cryptographic hash functions have additional properties. In particular, they are infeasible to invert. They are used in digital signatures, for storing and comparing passwords, in message authentication codes, etc.

Ideal cryptographic hash functions need to have the following properties:

1. Computing a hash value for a message needs to be fast and use low resources.
2. Given just a hash, it is infeasible to find the original message (except by trying all possible messages)
3. Hashes for similar messages should not be correlated (small change in message -> large change in hash)
4. Infeasible to find collisions (i.e. two messages with the same hash).

Examples for cryptographic hash functions are:
MD5 was widely used, but is not secure. Sometimes it is still used for integrity protection.
SHA1 is better, but attacking SHA1 is much easier than brute-force and attacks get more efficient and

more realistic. Therefore, it is no longer recommended for digital signatures and some agencies recommend not to use SHA1 any longer.
Current recommendations are **SHA-256, SHA-384** and **SHA-512.**

---

[1] Don't panic (Words on the cover of "The Hitchhiker's Guide to the Galaxy")

[2] One problem with the block-by-block encryption is that an attacker might just exchange some blocks in the message. This cannot be noticed by the recipient, because the message can still be decrypted. Thus, integrity is not satisfied!