Week 1 - Supplementary Reading

Site:	Monash Moodle1	Printed by:	M Utomo
Unit:	FIT9137 Introduction to computer architecture and networks - S1 2025	Date:	Wednesday, 5 March 2025, 11:52 AM
Book:	Week 1 - Supplementary Reading		

Description

The **Supplementary Reading Moodle Book** contains additional reading materials to complement the lectures. These materials are **not a substitute** for the lectures; however, they can help deepen your understanding of the concepts covered.

It is recommended that you **watch the lectures first** and then review the supplementary reading materials before attending this week's **Workshop and Applied Session**.

Table of contents

1. Introduction

The chapters in this textbook are organized 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/ UEFI* 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*.

1.1. How to succeed

The basic recipe for success here is to watch the video lectures and work through the supplementary reading material and to complete the other pre-class tasks before you attend the workshops and applied sessions. The lessons in your LMS ("Learning Management System", e.g. Moodle) contain quiz questions and other exercises that will help you deepen your understanding. Some sections in the Moodle books are optional, and some have pointers to additional material. It will always be clearly marked what is optional and what is part of the core content of the unit (i.e., what we expect you to read because it may be part of the assessment).

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 teaching staff 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!

1.2. Let’s 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 organizing a meet up to study for FIT9137). 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 (Wi-Fi, 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 chapters in this book, 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!

2. Number Systems

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 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

999

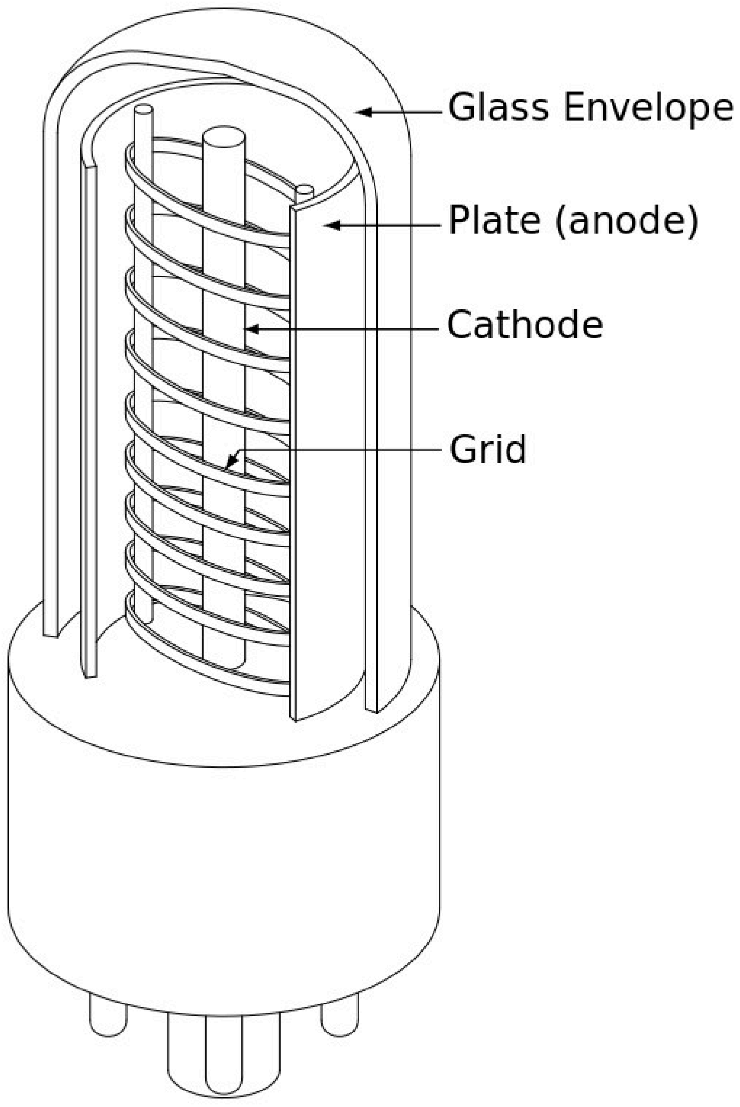
In Australia, **000** is the national emergency phone number, while **999** has the same meaning in other countries such as Malaysia or the United Kingdom. Both are also just sequences of digits, but they don't have any meaning as a number (e.g., you can't just dial **0** or **00** although, as a number, they would mean the exact same thing as **000**!).

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).

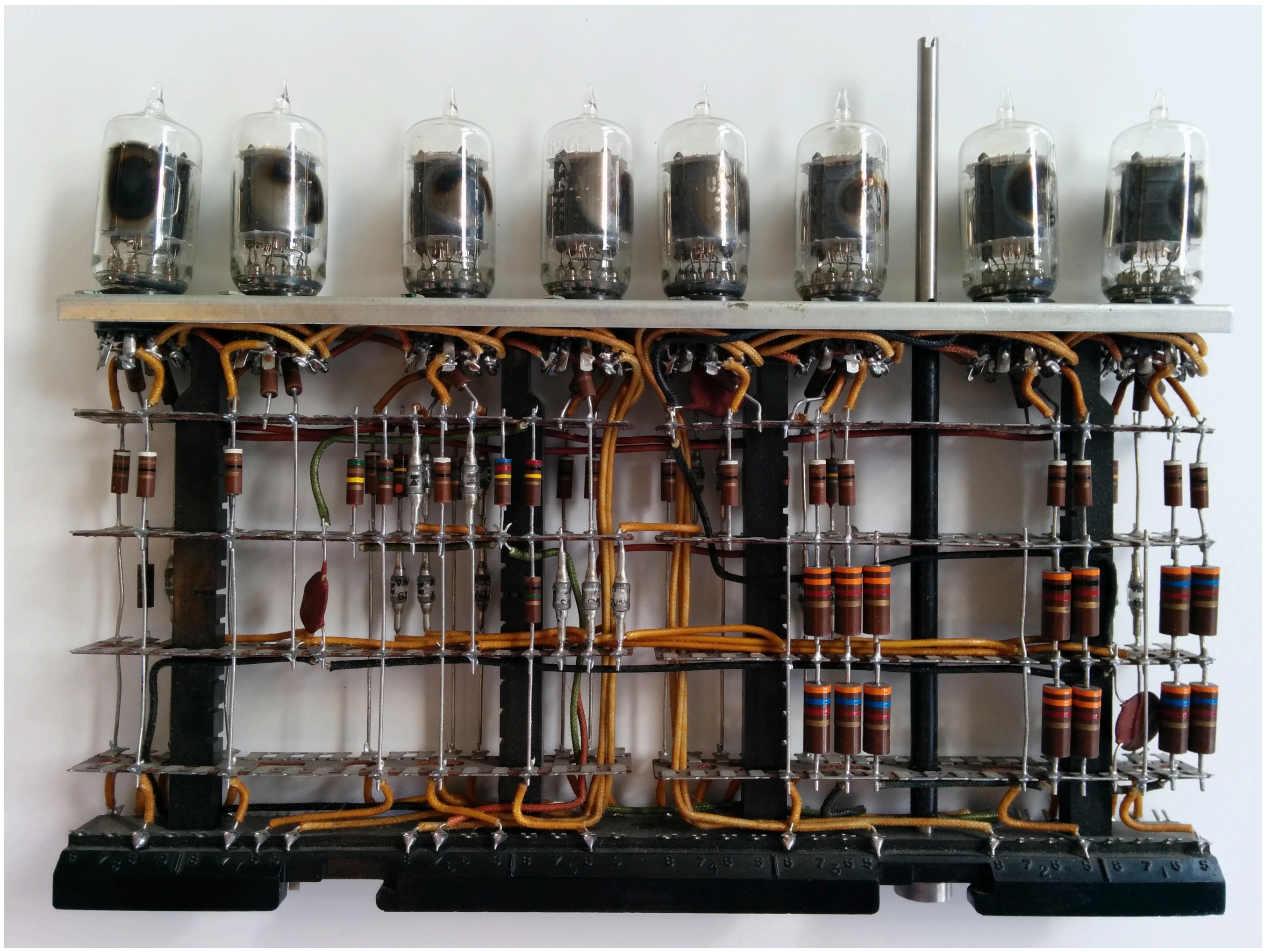
Modern, digital computers work with only **two different symbols**. That's because two symbols are enough to represent anything else, any number and any other piece of data. And it's reasonably simple to design electronic circuits that work with two symbols, by representing them as "high voltage" and "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.

2.1. 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 be imagined as switches that can be electronically turned on and off. The first computers used relays (electromagnetically operated switches) or vacuum 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 an amplifier or a 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 had 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 built came with the development of *transistors*. The word transistor stands short for "transfer resistor". 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 don't 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. In the history of the development of computers, there were experiments with switches that had 5, 8, or 10 different states. A 10-state switch could represent a decimal digit, so this seemed like a good thing to try. However, it turned out that switches with more than two states (just ON and OFF) were less practical.

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.

Note that the technical details of how vacuum tubes and transistors work is not part of the assessable content of this unit.

This video provides some information about the history behind the development towards the binary representation in computers:

2.2. Binary numbers

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).

1. 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:

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

2. 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:

1xxxxxxxx

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 insert a 0:

10xxxxxxx

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

101xxxxxx

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:

1011xxxxx

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

10110xxxx

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

101101xxx

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

1011011xx

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

2.2.3. 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 *arbitrarily long* binary numbers: all operations are limited to a **fixed number of digits**. In a later section, 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 binary digits we compute with:

- A **bit** is a single binary digit, i.e., a single 0 or 1. The term "bit" is in fact a portmanteau (i.e., a blend) of the two words "binary" and "digit".
- A collection of eight bits is called a **byte**. Many early computers could only perform 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 $\lceil \log_2 x \rceil$ bits.

2.3. Values/Sizes & Measurements

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 table shows prefixes based on powers of 10 for large numbers.

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

And this table shows the power-of-10 prefixes for small numbers.

Prefix	Symbol	Factor
deci-	d	10^{-1} = 0.1
centi-	c	10^{-2} = 0.01
milli-	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? 2^n

Very often, there is confusion wheter 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

Prefix	Symbol	Factor
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

2.4. Hexadecimal numbers

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

1. Slow method for conversion

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

2. Faster method

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

Division		**Result**	**Remainder (decimal)**	**Remainder (Hex)**	**Base 16 number**
93530 / 16	=	5845	10	A	xxxxA
5845 / 16	=	365	5	5	xxx5A
365 / 16	=	22	13	D	xxD5A
22 / 16	=	1	6	6	x6D5A
1 / 16	=	0	1	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.

2.5. Negative binary numbers

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.

1. 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.

2. One's 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 **one's complement**, and here is the table for 3-bit numbers:

Decimal	3-bit one's complement
0	`000`

Decimal	3-bit one’s complement
1	`001`
2	`010`
3	`011`
-0	`111`
-1	110
-2	101
-3	100

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

2-1 = 2 + (-1) = 010_2 + 110_2 + 1 = 1001

Now we simply ignore the leading 1 (it’s just an artefact of the one’s 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.

3. Two’s complement

In the two’s 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 all 3-bit two’s complement numbers:

Decimal	3-bit two’s 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.

Two’s 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:

Decimal	**Two's complement**
$2_{10}+1_{10} = 3_{10}$	$010_2 + 001_2 = 011_2$
$3_{10}-1_{10} = 3_{10} + (-1_{10}) = 2_{10}$	$011_2 + 111_2 = 010_2$ (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 two's complement. The result, 5, or 101_2, is a negative number if interpreted as two's 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.

2.6. Encoding text into binary

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

Bit pattern	Character	Decimal value	Bit pattern	Character	Decimal value
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
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? $2^{32} = 4,294,967,296$!

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.

2.7. Floating Point Numbers (OPTIONAL)

Floating Point Numbers

Note: This entire section is optional. It is not part of the assessed material of FIT9137. However, if you're interested in computer architecture, this will give you an overview of how floating-point numbers are represented in binary.

The numbers we've looked at so far were all just different representations of the *integers,* i.e., whole numbers that can be positive or negative (or zero). Many real-world applications of course require the use of non-integer numbers. A simple example could be a banking application, where it would be unacceptable to round every amount to a full dollar value. On the other hand, since the smallest unit of our currency is a cent, we could simply represent all values as cents, so that we can use integers again. This is called a **fixed-point representation**, because it only supports a fixed number of digits after the decimal point.

For many applications, fixed-point arithmetic is not sufficient. Let's look at an example. The speed of light in a vacuum is roughly 300,000 km per second. We want to compute the distance travelled in 0.000015 seconds at the speed of light. In order to use fixed-point arithmetic, we would have to multiply our time scale by one million, so that instead of 0.000015 seconds we would be calculating with 15 microseconds. Accordingly, the 300,000 km per second would have to be expressed as 0.3 km per microsecond - but there we have a decimal point again! So we have to change units for the distances as well, going to 300 m per microsecond instead. Now we can finally do the calculation, and get the result of 4,500 meters. As you can see, we had to change both our units of measurement in order to fit the calculation into an integer range.

A common way of expressing large and small numbers is to use **scientific notation**. For example, we could express 300,000 km per second as 3×10^5 km per second, and 0.000015 seconds as 1.5×10^{-5} seconds. Then the calculation is much easier: $3 \times 10^5 \times 1.5 \times 10^{-5} = 3 \times 1.5 \times 10^{5-5} = 4.5$, so the result is 4.5 kilometres. The advantage is that we can do the main multiplication with small numbers, and do simple additions of the exponents.

1. Representation

Computers use a very similar representation to enable them to work with non-integer numbers - essentially, they are using scientific notation in base 2 rather than base 10. This is called **floating point representation**, due to the fact that the "decimal" point (it's not really decimal because we're using binary) can be shifted left and right by changing the exponent.

A floating point number typically would be represented as $a \times 2^b$, where a and b are binary numbers, each with a fixed number of bits. We call a the **significand** and b the **exponent**. The exponent can be both positive and negative, so that we can shift the "decimal" point in both directions. In addition, we usually have one more bit that tells us whether the floating point number is positive or negative (much like the signed magnitude representation).

2. Precision

As mentioned above, floating point numbers are represented with a fixed number of bits. Their range is therefore limited. This is not surprising, and in fact we often would use rounded representations for fractional decimal numbers as well. For example, 1/3 could be represented with a precision of 8 digits (including the leading zero) as 0.3333333. Similarly, 2/3 would be 0.6666666. But now what happens when you add the two numbers: 1/3+2/3=1, but 0.3333333+0.6666666=0.9999999! We have introduced a **rounding error**.

Binary floating point numbers have exactly the same problem. Let's convert the decimal number 0.1 into this format. Unfortunately, one tenth cannot be represented as a sum of powers of two, so we end up with the infinite binary number 0.00011001100110011001… Convince yourself that this is correct:

$$0.1 = 0 \times 2^0 + 0 \times \frac{1}{2} + 0 \times \frac{1}{4} + 0 \times \frac{1}{8} + 1 \times \frac{1}{16} + 1 \times \frac{1}{32} + \ \ldots$$

The IEEE 754 standard for 64 bit floating point numbers uses 52 bits for the significand and 11 bits for the exponent, plus one bit for the sign. The closest binary value to 0.1 we can represent is therefore

$$1.1001100110011001100110011001100110011001100110011001101 \times 2^{-4}$$

This uses the full 52 bits for the significand, and we can easily represent -4 using the 11 bits for the exponent. But when we convert this back to a decimal number, we get

$$0.1000000000000000055511151231257827021181583404541015625$$

This error looks quite small, but small errors can add up!

2.8. Error detection (OPTIONAL)

Error detection

Note: This entire section is optional. It is not part of the assessed material of FIT9137. However, if you're interested in computer architecture, this will give you an overview of how computers can detect errors in binary data.

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:

2.8.1. 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 final digit 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 in position 6 has changed from 1 to 0:

01011000

As we have agreed on even parity, we can now check if this is still correct. We find three 1s, which is an odd number. Thus, there must have been an error. As mentioned above, the goal is to *detect* that the binary data has changed, not to *fix* that error. We only know that *one of the bits* has changed, but not which particular one.

Now what happens if there are two errors:

01111000

In this situation, there are again four 1s, but in the wrong positions. Since we have even number of 1s and the parity bit is 0, we cannot detect this error at all. Parity bits therefore only work for short sequences of bits (e.g., one parity bit for every 8 data bits), and when the likelyhood of multiple bits being corrupted at the same time is low.

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.

2.8.2. 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 31 305

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

43 52 43 29 31 305

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 31 305

However, what about these two errors?

43 54 43 28 31 305

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.

Note that 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 Logic

Computers are less than a hundred years old. But some of the fundamental principles are a lot older.

In the previous section, we discussed *numbers*, which are of course fundamental to computing and have been used for thousands of years. In this section, we will see how we can use 0s and 1s to implement *logic* operations. This kind of logic, based on just a couple of very simple operations, is called *Boolean* logic, named after **George Boole**, a 19th century English mathematician and philosopher. For more details on Boole, check out his Wikipedia page.

Boole's work was influential in creating a *formal* logic, i.e., a mathematical system that formalizes what it means to *reason*. This was mostly a philosophical and mathematical topic for the next hundred years or so.

But then it turned out that Boolean logic is the perfect theoretical foundation on which to build electronic computers. That's why in this section, we will introduce the basics of Boolean reasoning, in a system called *Boolean Algebra*. We will also see how this translates into very simple electronic circuits, which we will then use in the next chapter to construct a computer.

Don't be intimidated by the words "theory", "logic", "formal", "algebra" etc. The basic ideas are very simple, and we will introduce them step by step.

3.1. Basic Concepts

1. Basic Concepts

Boolean algebra, or Boolean logic, defines formal, mathematical rules that describe how humans can reason about statements that are either **TRUE** or **FALSE**.

For example, the statement *"I am enrolled in FIT9137"* is either true or false. Similarly, the question *"Are you enrolled in FIT9137?"* has a yes/no (or true/false) answer. In addition to simple statements of (true or false) facts, we can also come up with more complex statements that combine several facts. How about this one:

"Anyone who is enrolled in FIT9137 and whose sum of assignment marks is greater than 50 will pass FIT9137 ."

This statement combines three facts:

1. whether you're enrolled,
2. whether your sum of assignment marks is greater than 50, and
3. whether you pass.

It says that if the first two statements are true (enrolled and marks greater than 50), then the third statement is true (pass).

So far, we have just stated facts or rules. But logic allows us to go one step further and *reason* about those facts. For example, if I tell you that I didn't pass FIT9137, then you can use *logical reasoning* (also called *inference* or *deduction*) to find out additional facts. In this case, the rule above would tell you that I was either not enrolled in FIT9137, or I did not get enough marks.

In Boolean algebra, an expression can only have one of two possible values: **TRUE** or **FALSE**. No other values exist. This maps nicely to digital electronic circuits, which also only have two states for each input or output. The value **TRUE** is usually represented by **1**, while **FALSE** is represented by **0**.

The logical statements we've seen so far are still quite informal, just written in natural language English text. In order to formalise and automate the reasoning process, we will need mathematical notation to express statements and rules. Instead of writing complex statements such as "Enrolled in FIT9137", we typically just use mathematical variables such as x and y or A and B. We also need to express the way we connect different statements more formally. In the introductory examples above, we've already used the most fundamental Boolean *operators*: **AND**, **OR**, and **NOT** (in addition to "if-then", which, as we will see later, can be expressed in terms of **NOT** and **OR**).

1. 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.

Note: This section also defines the acceptable notation for your assessment. You may receive a deduction or 0 marks for submissions that don't use the notation defined here.

AND

A statement **A AND B** is only **TRUE** if both individual statements are **TRUE**, and **FALSE** otherwise.

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

A AND B can be written as

$A \land B$ or AB

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

OR

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

A OR B can be written as

$A \vee B$ or $A + B$

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:

A	B	$A \vee B$
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.

NOT A can be written as

$\bar{A}$ or $\neg A$

The truth table of NOT looks like this:

A	$\bar{A}$
0	1
1	0

Notation

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 distinguish 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)

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.

A	B	$A \bigoplus B$
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.

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

- $A \rightarrow B$, sometimes also written $A \implies B$, can be read as A implies B. This expresses that if A is TRUE, then B follows (is also TRUE). It is logically the same as $\overline{A} \vee B$ and therefore has the following truth table:

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

3.2. From Boolean algebra to circuits

From Boolean algebra to electrical (digital) circuits

This section introduces logic gates, the basic building blocks of digital circuits, and then explains how to construct a circuit from a given Boolean function.

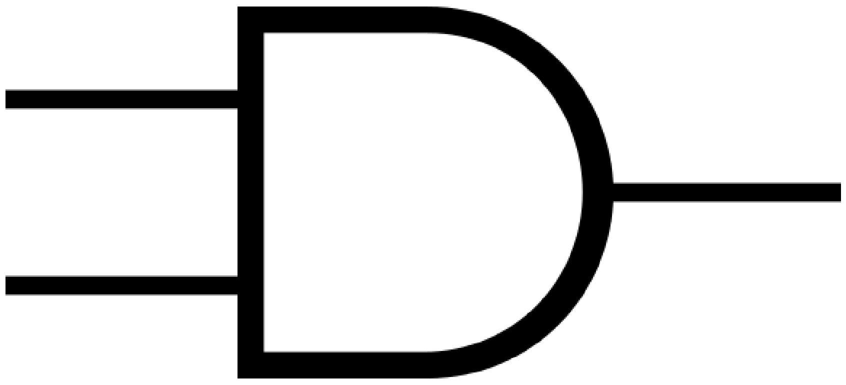
1. Logic gates

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

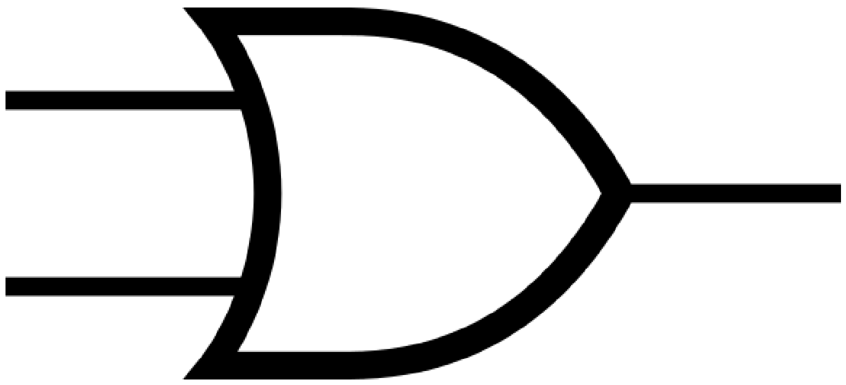
A gate performs one (or several) logical operations on some logical input (i.e. bits) and produces a single logical output. We look at these electrical circuits on an abstract level as "logical circuits". These logical circuits do not behave like electrical circuits. This can be a bit confusing, as a 0 input can result in a 1 output (e.g. when using NOT). This is the correct behavior 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 idealized 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.

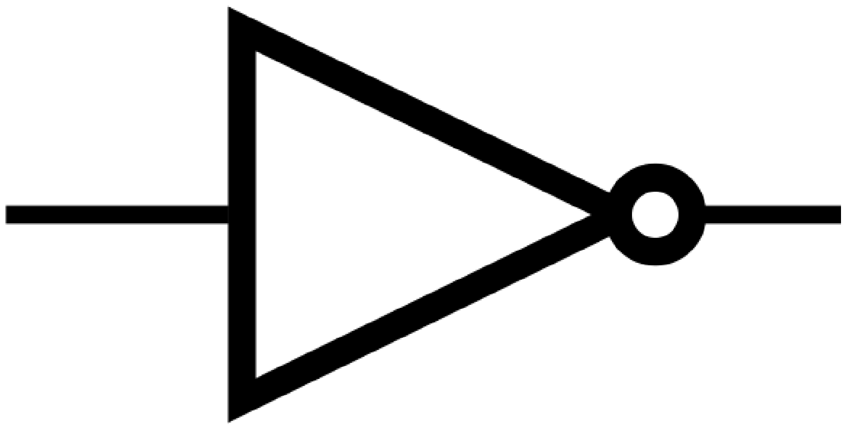
AND Gate



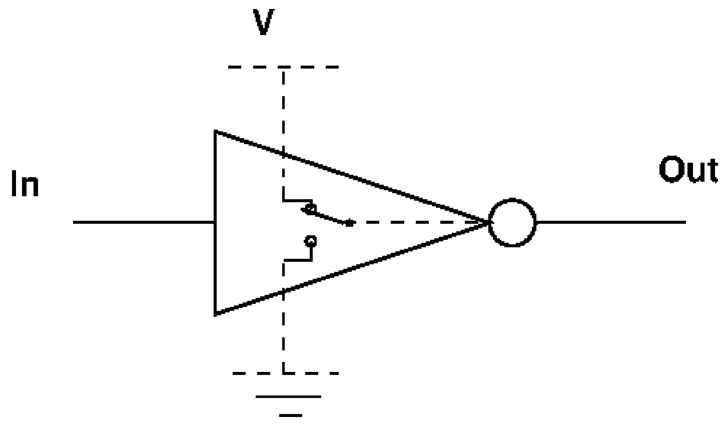
OR Gate



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.

2. From truth table to circuit

Truth tables uniquely define Boolean functions: each Boolean function has exactly one corresponding truth table, and each truth table corresponds to exactly one function.

Of course, the same function can be defined by several different *expressions*. For example, the expressions $X + Y$, $Y + X$ and *:math:X+Y(Z+overline{Z})* all define the same function. We will come back to this later, where we will see how to *simplify* Boolean functions.

Let's assume we have been given the following truth table:

X	Y	Z	$F(X, Y, Z)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

An easy way to turn this into a Boolean expression is to think about what each *row* means: The function is *true* exactly if the inputs match either row 3, row 4, row 5, row 7 or row 8.

Let's focus on row 3. When do the inputs match that row? Exactly if X is false (0), Y is true (1), and Z is false (0). We can write this down as a Boolean expression: $\overline{X}Y\overline{Z}$.

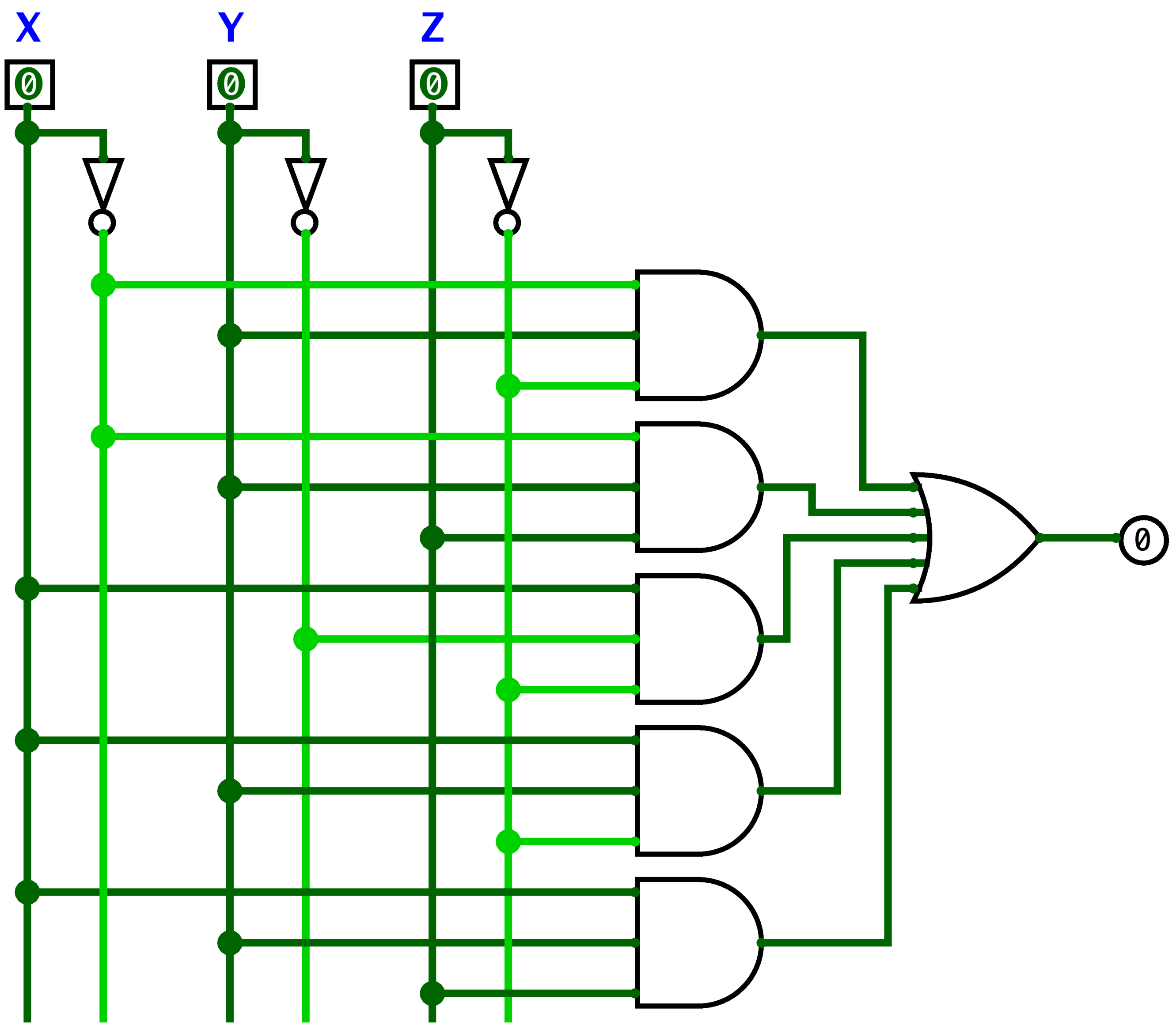
What about row 4? The expression would be $\overline{X}YZ$. If we continue this for the other rows that result in a true output, we get $X\overline{Y}\,\overline{Z}$, $XY\overline{Z}$ and XYZ.

Now let's go back to the beginning: We said that the inputs need to match row 3 or row 4 or row 5 or row 7 or row 8. So all we have to do is to combine the individual expressions for the rows in a big *or* expression:

$$\overline{X}Y\overline{Z} + \overline{X}YZ + X\overline{Y}\,\overline{Z} + XY\overline{Z} + XYZ$$

This is called a *sum-of-products* form, and it is one of the *canonical* forms of Boolean expressions.

Since it follows a nice, uniform structure, it is quite easy to design a circuit that implements the function:



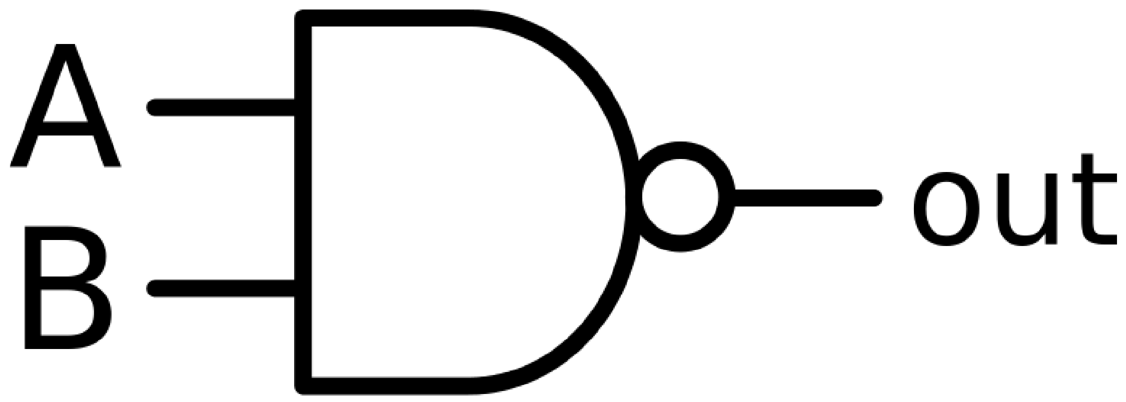
3. Universal gates

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.

3.3. Boolean algebra rules

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:

$$\begin{array}{cc} AND & OR \\ 1 \land A = A & 0 \lor A = A \end{array}$$

Null Law (or Dominance Law):

$$\begin{array}{cc} AND & OR \\ 0 \land A = 0 & 1 \lor A = 1 \end{array}$$

Idempotent Law:

$$\begin{array}{cc} AND & OR \\ A \land A = A & A \lor A = A \end{array}$$

Complement Law:

$$\begin{array}{cc} AND & OR \\ A \land \overline{A} = 0 & A \lor \overline{A} = 1 \end{array}$$

Commutative Law:

$$\begin{array}{cc} AND & OR \\ A \land B = B \land A & A \lor B = B \lor A \end{array}$$

Associative Law:

$$\begin{array}{cc} AND & OR \\ (A \land B) \land C = A \land (B \land C) & (A \lor B) \lor C = A \lor (B \lor C) \end{array}$$

Distributive Law:

$$\begin{array}{cc} AND & OR \\ A \lor (B \land C) = (A \lor B) \land (A \lor C) & A \land (B \lor C) = (A \land B) \lor (A \land C) \end{array}$$

Absorption Law:

$$\begin{array}{cc} AND & OR \\ A \land (A \lor B) = A & A \lor (A \land B) = A \end{array}$$

DeMorgans Law:

$$\begin{array}{cc} AND & OR \\ A \overline{\land} B = \overline{A} \lor \overline{B} & A \overline{\lor} B = \overline{A} \land \overline{B} \end{array}$$

Double Complement Law:

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