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.

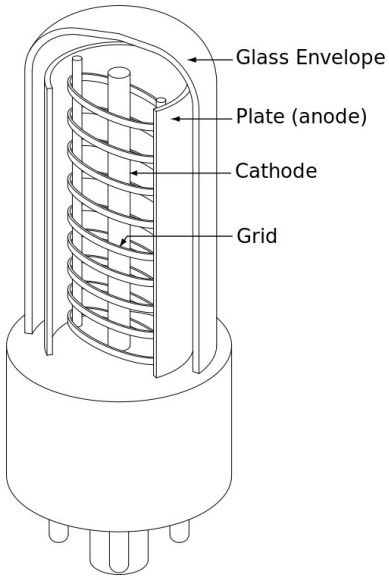
There are lots of practical exercises, which will help you reach a deeper understanding of the topics. It is absolutely crucial that you work through these exercises yourself, since many of the topics covered here can only be learned by doing! Your tutors will help you when you get stuck, and we encourage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

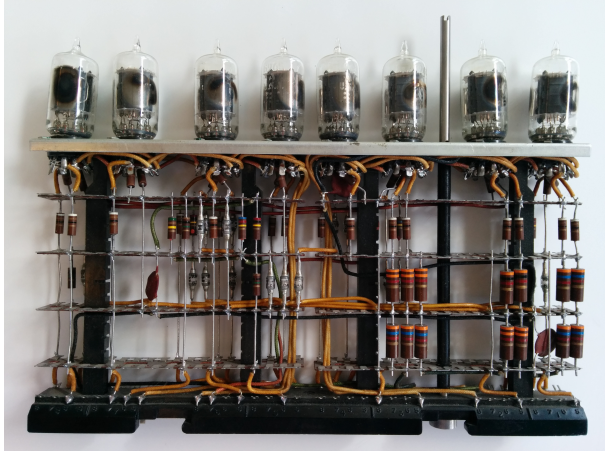
2
Representing numbers (and other things)

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:*

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

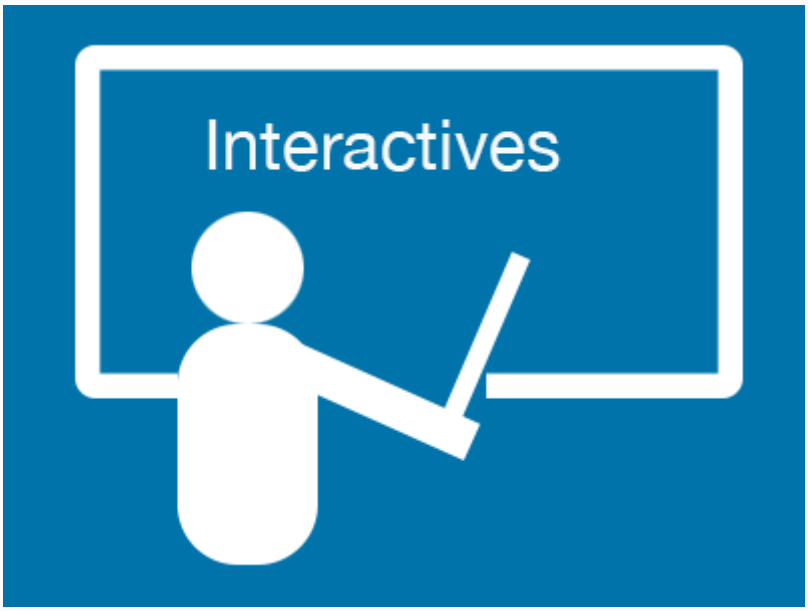
If we add everything up, of course we get three-hundred-and-sixty-five (365). In order to avoid any confusion between different number systems, we often write the base as a subscript, e.g. 365_{10} or 101101101_2 (so that we can distinguish it from 101101101_{10}, or one-hundred-and-one billion one-hundred-and-one 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:

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 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
100000	SPACE	32	100001	!	33
100010	"	34	100011	#	35
100100	$	36	100101	%	37
100110	&	38	100111	'	39
101000	(	40	101001	)	41
101010	*	42	101011	+	43
101100	,	44	101101	-	45
101110	.	46	101111	/	47
110000	0	48	110001	1	49
110010	2	50	110011	3	51
110100	4	52	110101	5	53
110110	6	54	110111	7	55
111000	8	56	111001	9	57
111010	:	58	111011	;	59
111100	<	60	111101	=	61
111110	>	62	111111	?	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
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

Bit pattern	Character	Decimal value	Bit pattern	Character	Decimal value
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 ride-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]

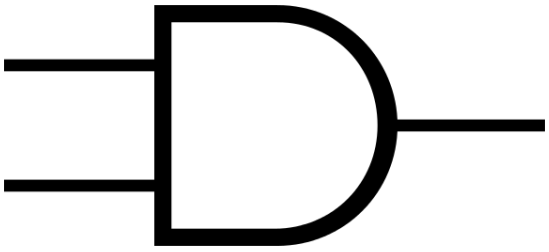
From Boolean algebra to electrical (digital) circuits

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

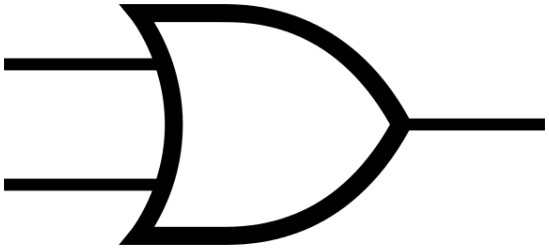
A gate performs one (or several) logical operations on some logical input (i.e. bits) and produces a single logical output. We look at these electrical circuits on an abstract level as "logical circuits". These logical circuits do not behave like electrical circuits. This can be a bit confusing, as a 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.

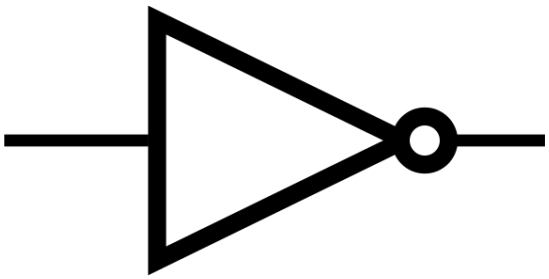
AND Gate



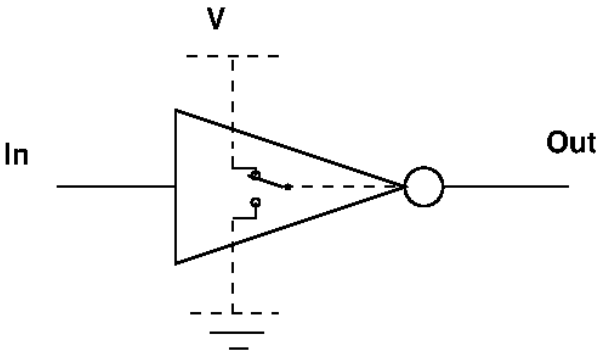
OR Gate



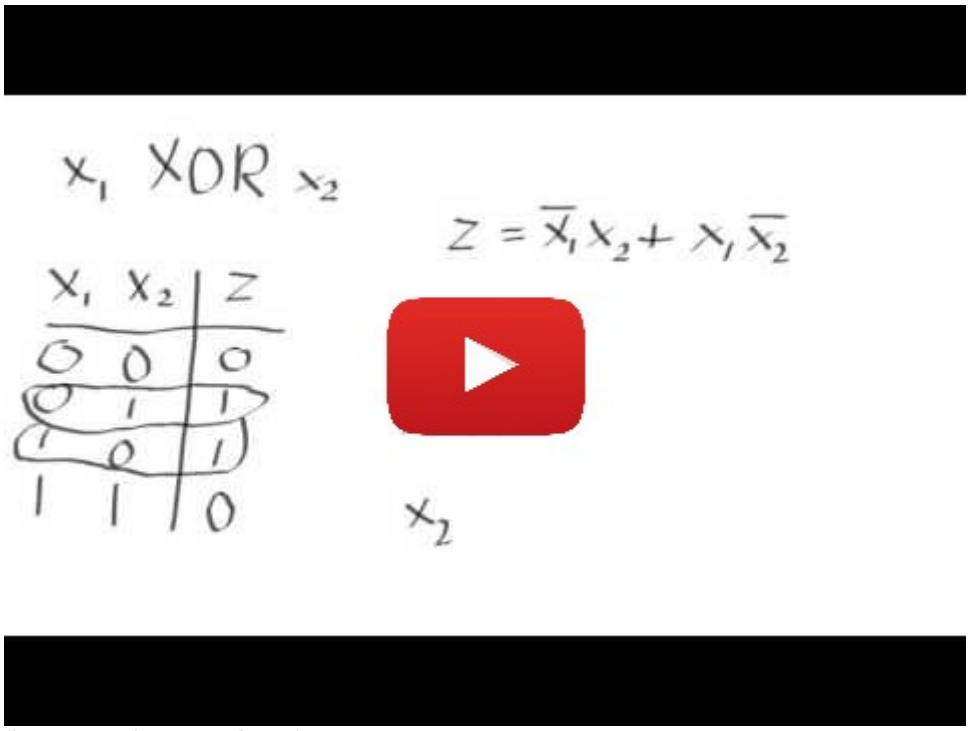
NOT Gate



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

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.

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

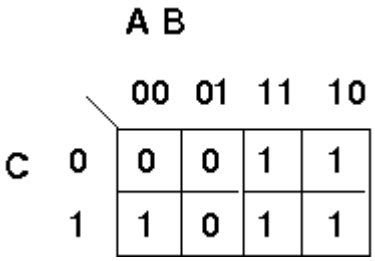
A K-map with 3 variables

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.

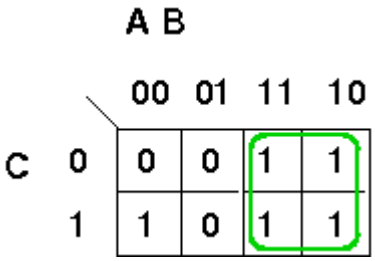
This example illustrates the use of K-maps for 3 variables.

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

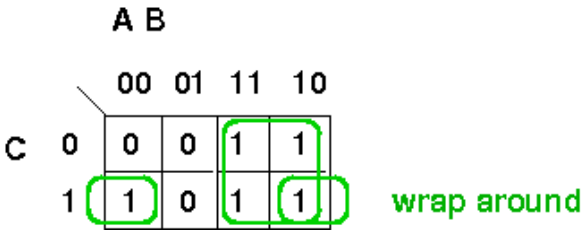
We have 5 terms in this function. Thus, we have to place 5 1s into the K-map:



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

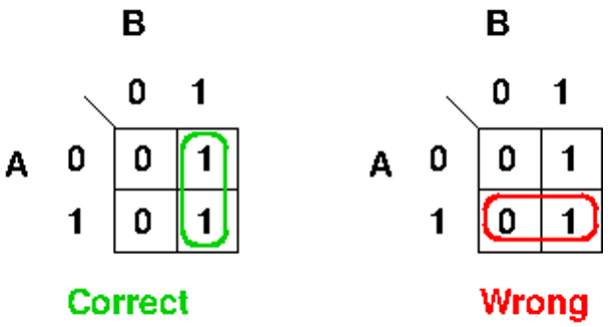


The minimized function now has only two terms representing the two groups in the K-map.

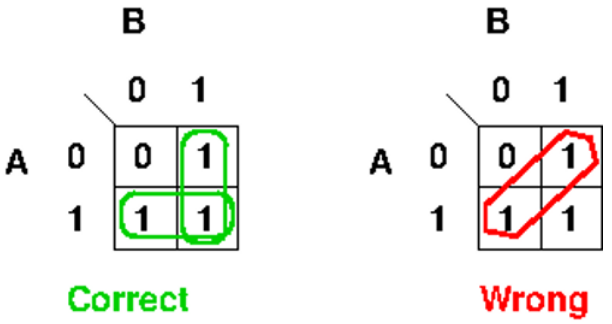
$$F(A,B,C) = A + \overline{B}C$$

Rules for K-maps

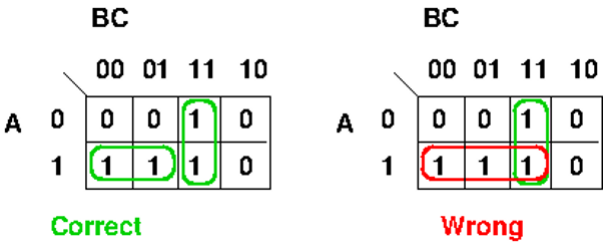
Rule 1: No group can contain a **zero**.



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



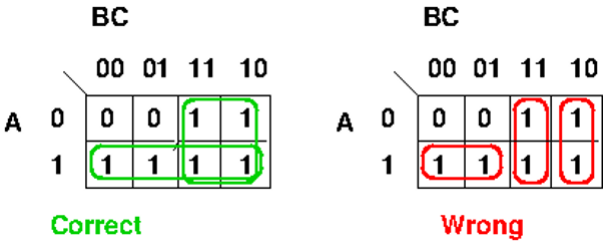
Rule 3: Groups must contain 1,2,4,8,16,32,... (**powers of 2**).



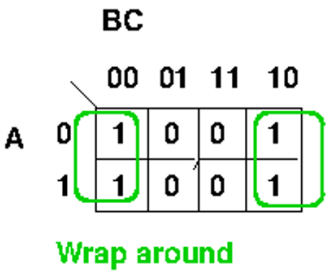
Rule 4: Each group must be as **large** as possible.

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.



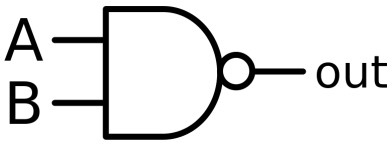
Universal gates

NAND has special properties:

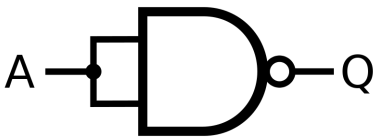
- NAND can be physically implemented very efficiently.
- All other gates can be build only using NAND gates.

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.*

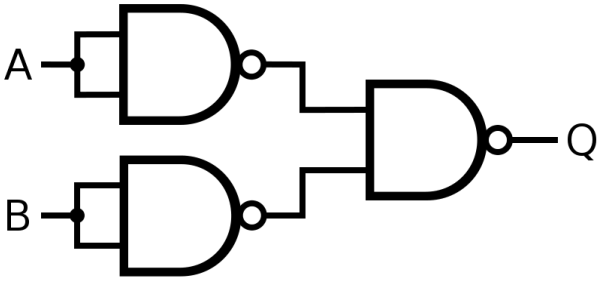
The symbol for a NAND gate is this:



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
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.

4.1
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.*

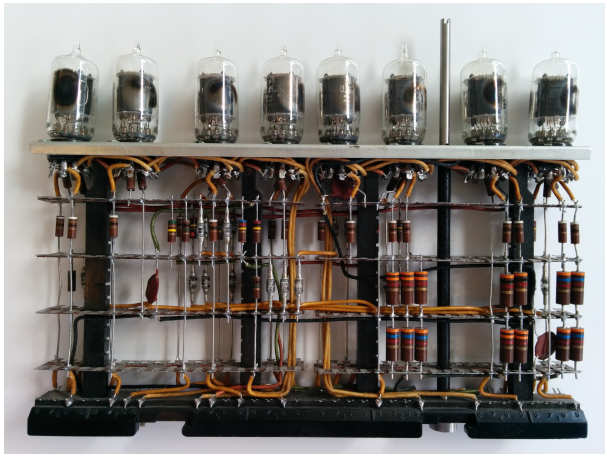
As you can see, *words* of data play an important role. CPUs use a fixed word-width, typically 16, 32 or 64 bits, and all instructions and all data has to be broken down into words of this fixed width. Again, the reason for this is to simplify the circuits.

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.

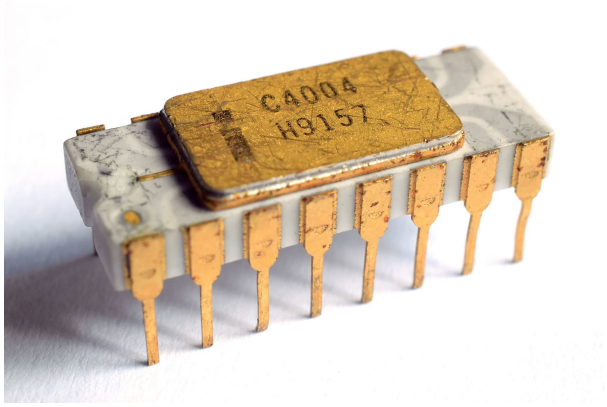
Some early CPUs

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.

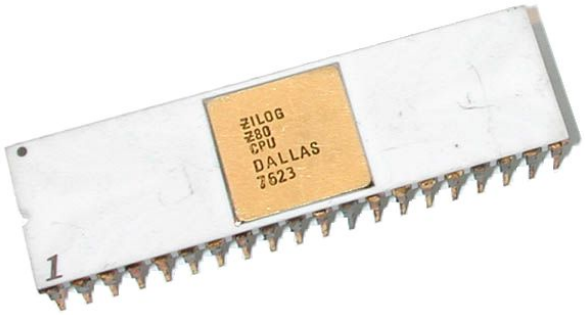


Intel C4004, By Thomas Nguyen, CC BY-SA 4.0, via Wikimedia Commons

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]

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).

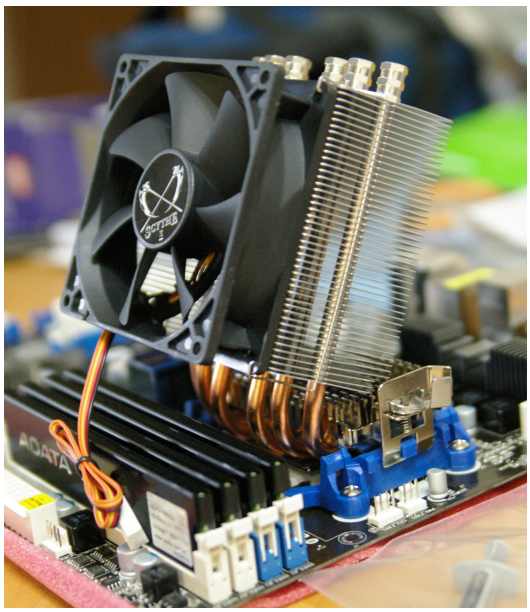
Let us take these two examples to explain the differences between different CPU architectures.

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
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 on implementing the MARIE CPU using logic 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]

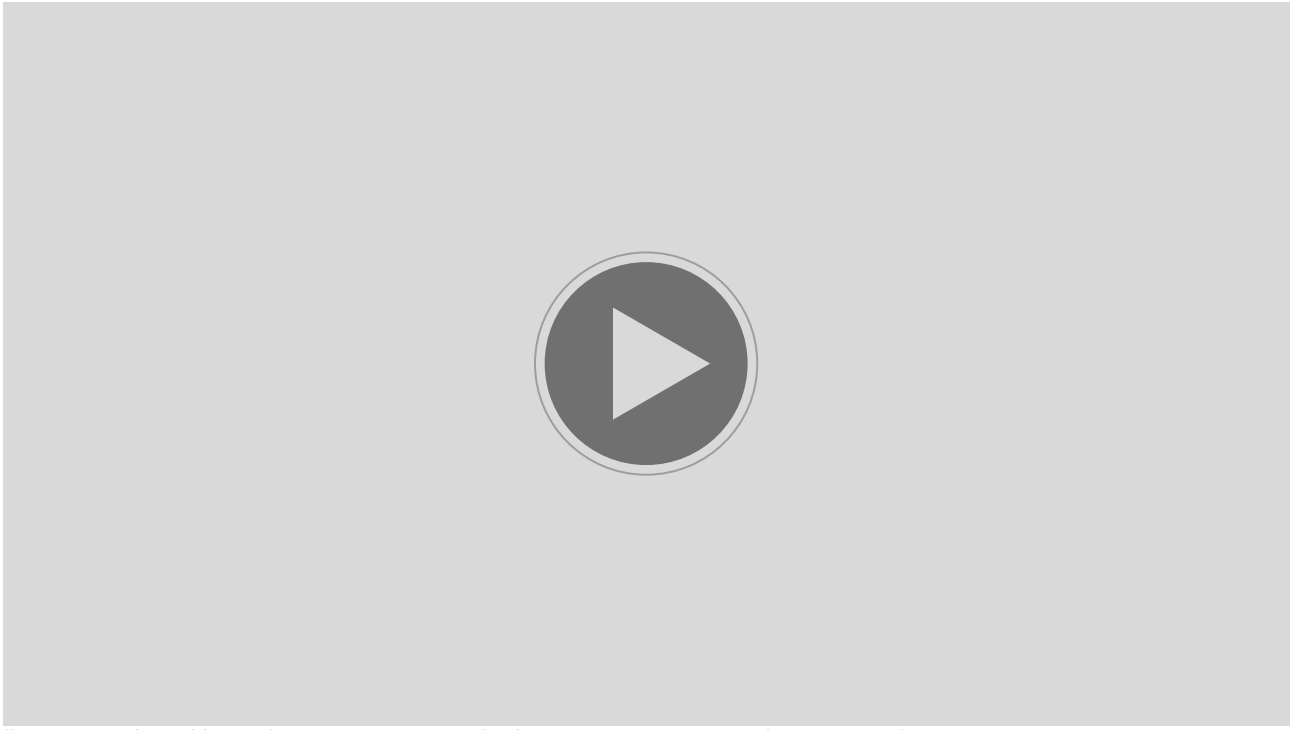
The `Input` and `Output` instructions can be used to read input from a user, and to output values to the screen, respectively. Again, they both use the AC register. So in order to e.g. output a value that is stored in memory, we first have to `Load` it into the AC, and then use the `Output` instruction. 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>0 then output AC and go back to line 1, else halt
```

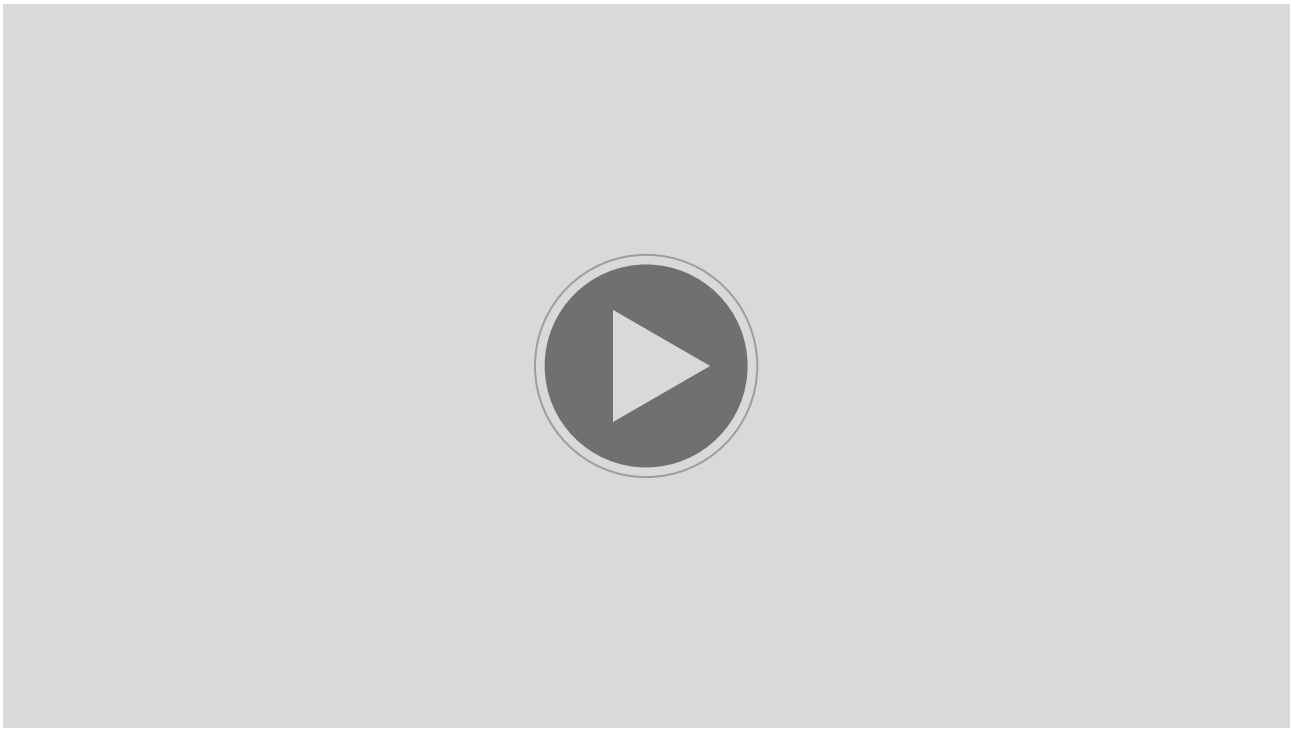
How can we turn this *if-then-else* construct into a sequence of MARIE instructions? The good news is that `SkipCond` allows us to test if AC is greater than 0 (by using condition code `800`). Now the confusing thing about `SkipCond` is that it *skips* the next instruction if the condition is true. So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

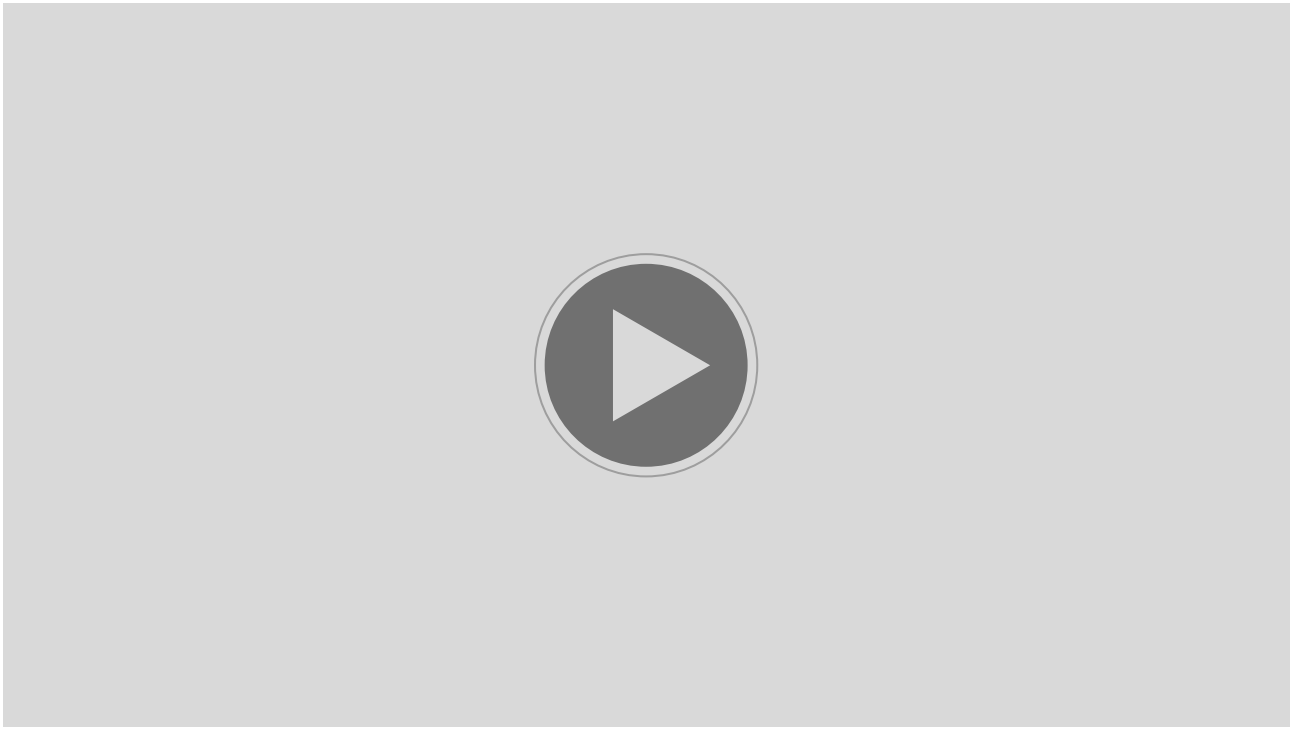
Opcode	Mnemonic	Explanation
1011	`AddI X`	Add value pointed to by X to AC
1100	`JumpI X`	Continue execution at location pointed to by X
1101	`LoadI X`	Load from address pointed to by X into AC
1110	`StoreI X`	Store AC into address pointed to by X

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
```

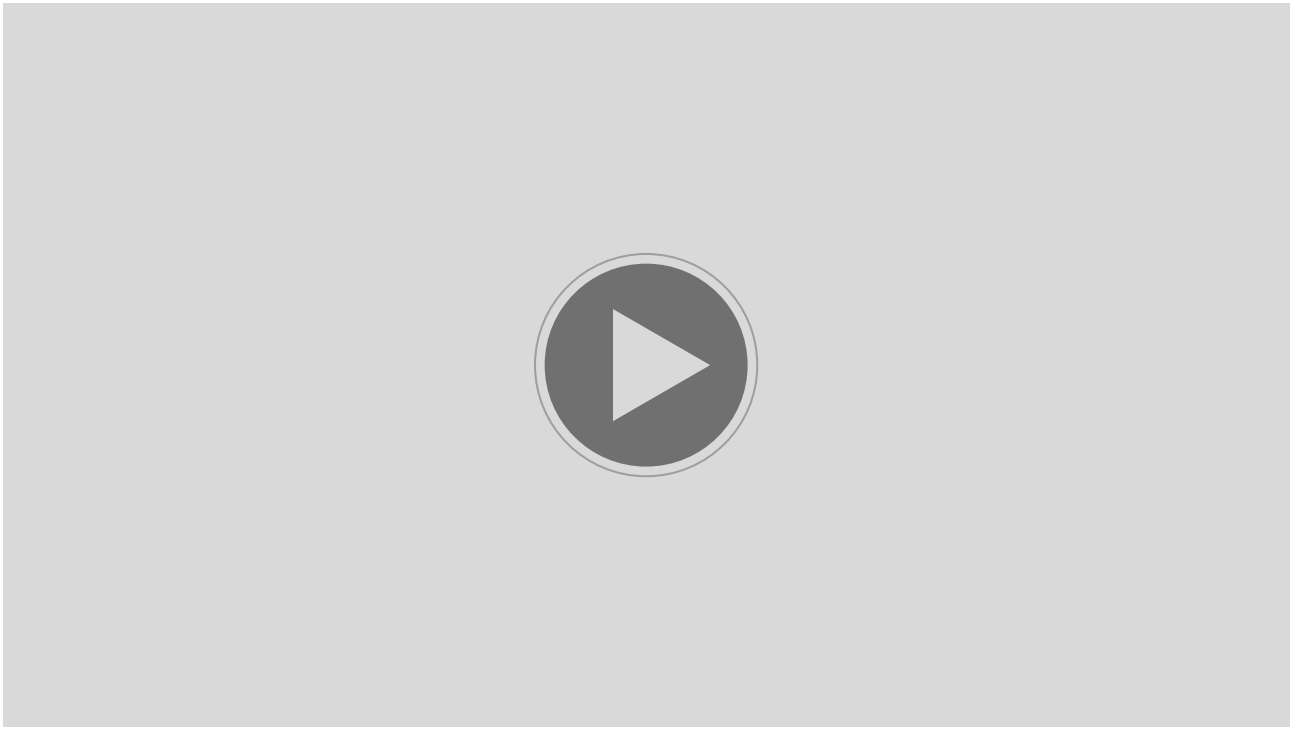
This code gets a user input and stores it in the location where the subroutine expects its argument. The `JnS Print` instruction then saves the *return address,* i.e., where the program should continue after the subroutine is finished, into address `Print` (where we put a placeholder `HEX 0`), and jumps 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`)

4.3 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.

4.3.1 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

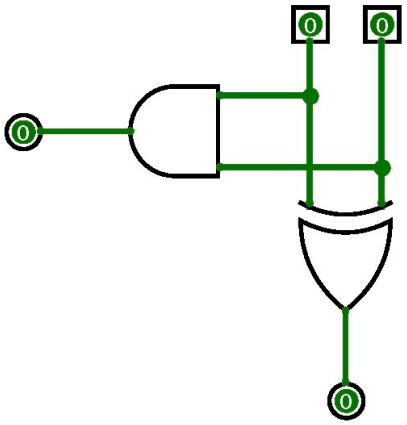
Let's start with something very simple: adding two bits, let's call them **A** and **B**. We can easily list all possible outcomes for any combination of inputs: 0+0=0, 0+1=1, 1+0=1, 1+1=2. Of course the output needs to be in binary, so in fact we should write $1+1=10_2$. We can see that we will need 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.

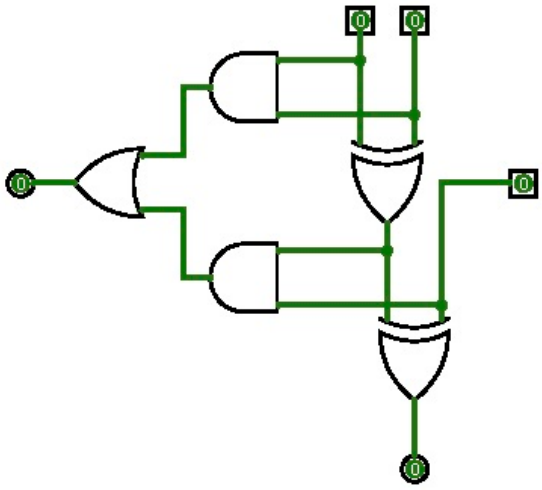
Full adders

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:

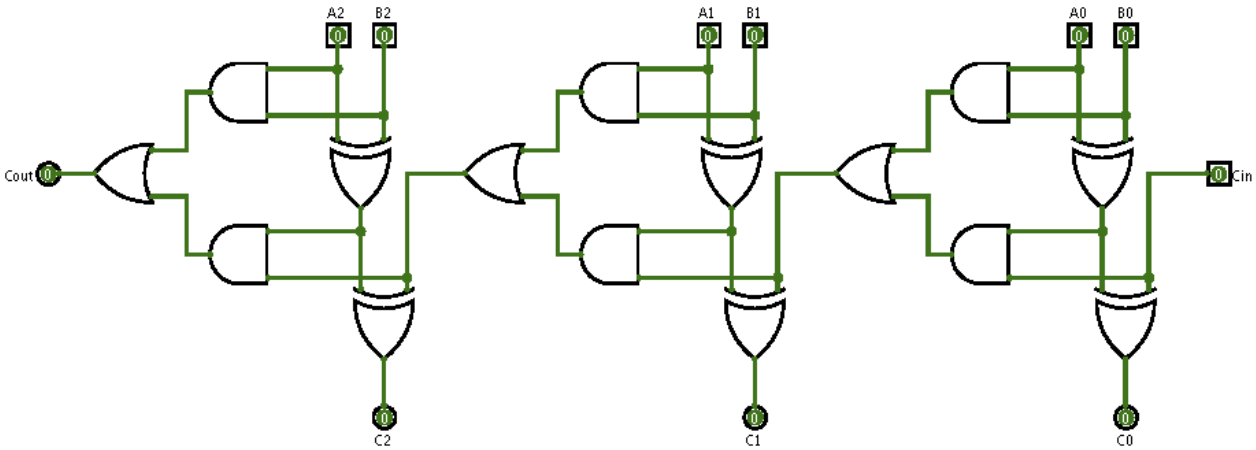
A	B	Carry-in	Carry-out	Result
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

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:

TODO: add graphics

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 and a carry-out, 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!

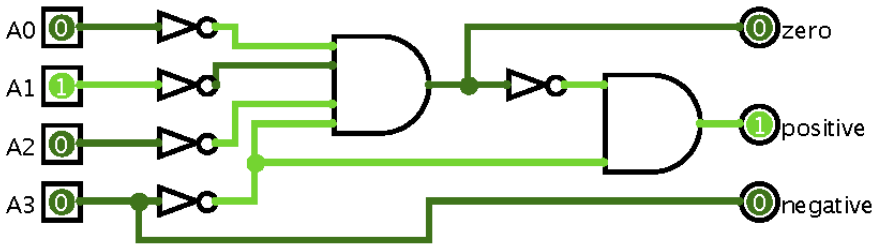
Since the efficiency of these basic circuits is so important for the raw speed of a CPU, circuits such as adders have been highly optimised. The trick to make adders more efficient is to avoid having the carry ripple through all bits. Instead, we add more gates to pre-compute whether entire groups of 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

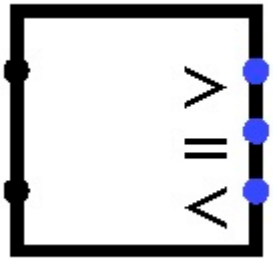
Another important operation that can be implemented using combinational circuits is the *comparison* of two numbers. In the MARIE instruction set, the `SkipCond` instruction compares the current contents of the AC register to the value zero. That's in fact all we need, since any comparison between two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

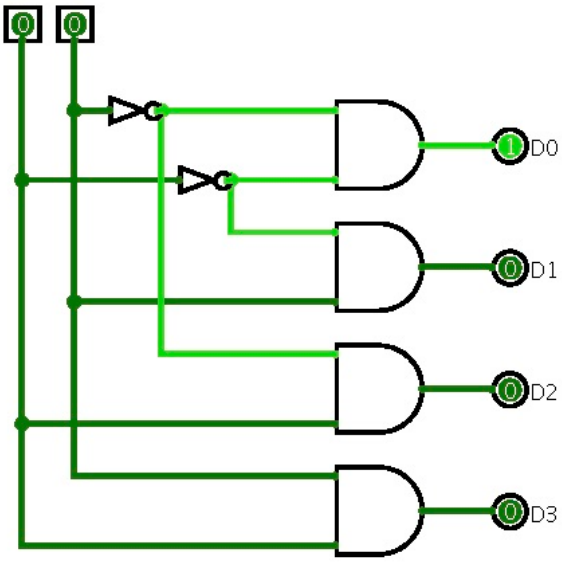


Decoders

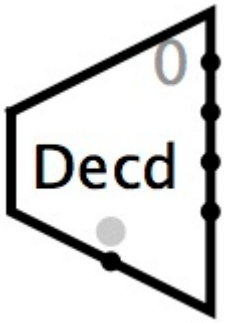
Combinational circuits can also be used to do things other than arithmetic. One circuit that is used quite often is a so-called *decoder*, which basically turns a binary number into a *unary* representation. It has *n* inputs and 2^n outputs, and it activates the output that corresponds to the binary 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:



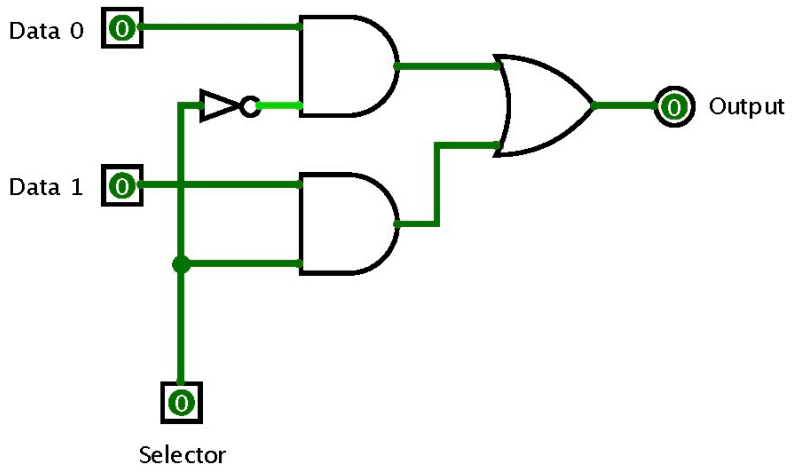
This one has a 2-bit input at the bottom, and four outputs (In Logisim, you can combine several wires into a multi-bit wire like the 2-bit one here, in order to make the diagram look cleaner).

Multiplexers

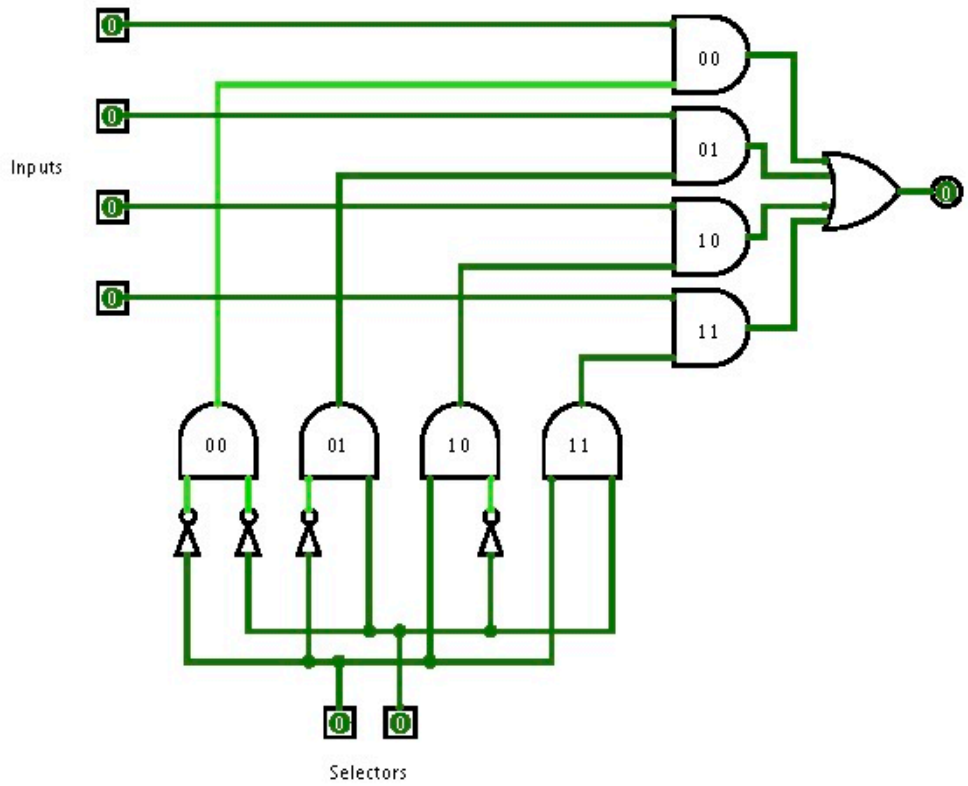
The next combinational circuit we're going to look at is called a *multiplexer*. Its function is to *select* one out of several data inputs. In order to do this, it has *n selection inputs*, which determine which one of 2^n the data inputs to pick. The simplest form of multiplexer would have two data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

Arithmetic-Logic-Unit

We have now introduced all the basic combinational circuits that are required to put together a simple arithmetic-logic-unit (ALU). Our ALU 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.

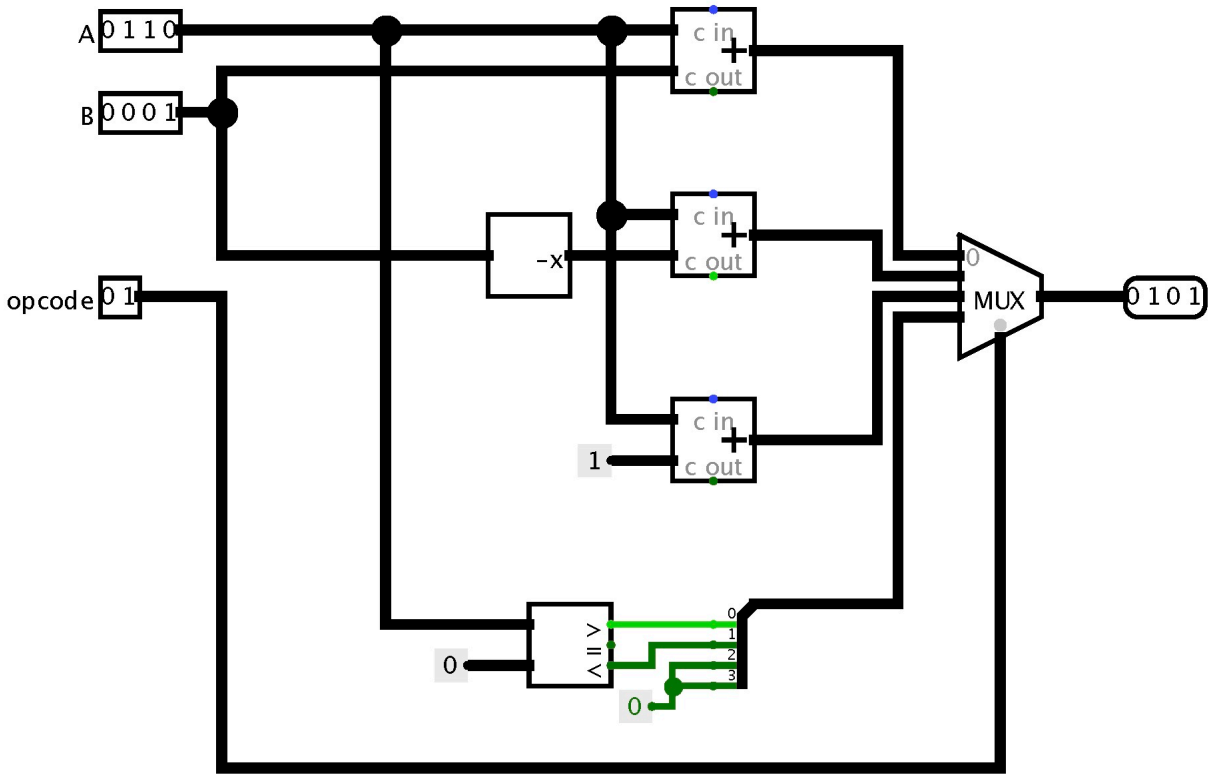
Such an ALU needs two *n*-bit data inputs (both are used in addition and subtraction operations, but only one is used in comparison and increment), and one 2-bit input for the so-called *op-code*, which selects the kind of operation (since there are four different operations, we can select the one we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

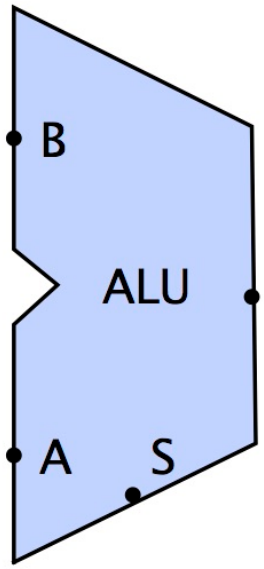
Here is a complete 4-bit ALU that can perform the four operations needed for MARIE:



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:



[1] This is exactly the XOR function.

4.3.2 Sequential circuits

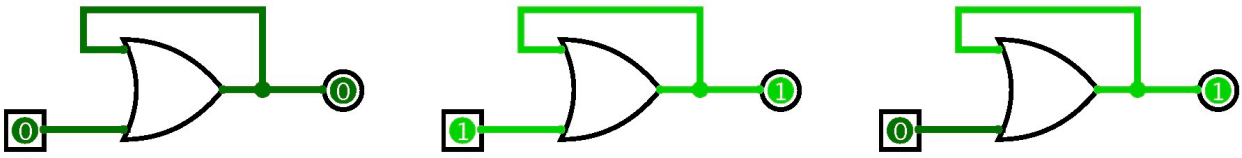
The combinational circuits in the previous module all had in common that they act like mathematical functions: the outputs are determined only by the inputs. In particular, those circuits could not *store* any information, because storage means that the behaviour of the circuit depends on what has happened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

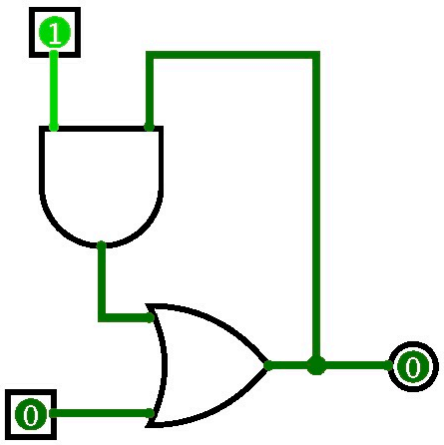
The main mechanism that allows a circuit to remember the past is to **pass its output back into its input.** That way, we can establish a feedback loop.

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.

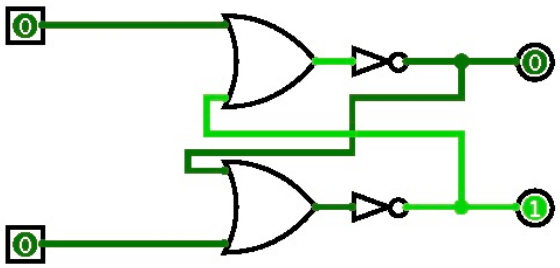


The switch (the input at the top of the circuit) now controls the memory. If it is set to 1 (as in the picture),

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!

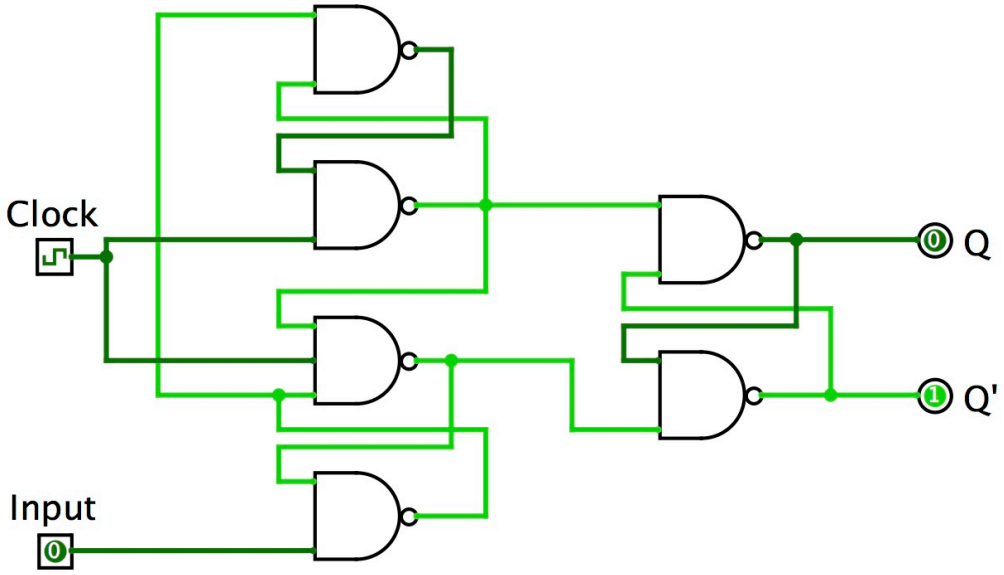
For combinational circuits, we have always used truth tables to describe their function. It turns out that we can extend truth tables to also work for sequential circuits, if we list the outputs at a previous point in time as inputs in the table. The truth table for an S/R latch looks as follows:

$\mathbf{S}$	$\mathbf{R}$	$\mathbf{Q(t)}$	$\mathbf{Q(t+1)}$
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.

4.4
Constructing a CPU

5
Memory and I/O

A CPU on its own is essentially useless. It needs **memory** to access data and instructions, and **input/output devices** to communicate with its environment (i.e., users or other computers). In this module, we'll first look at how memory is organised and implemented, and after that we'll see how external devices can communicate with the CPU.

5.1 Memory

The module on CPU basics 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.

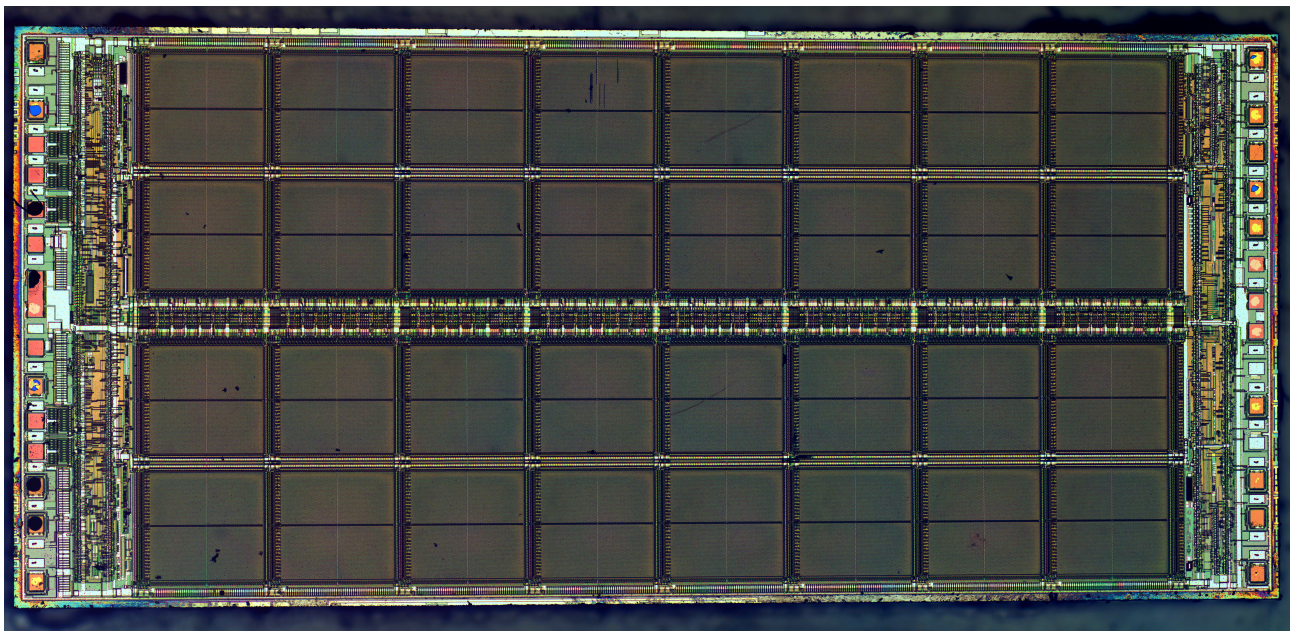
So to summarise, **in order to address 2^n memory locations, we always need *n* bits for the addresses**. In byte-addressed systems, the number of locations is the number of bytes. In word-addressed systems, it's the number of words, which is half the number of bytes if the word size is 16, a quarter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

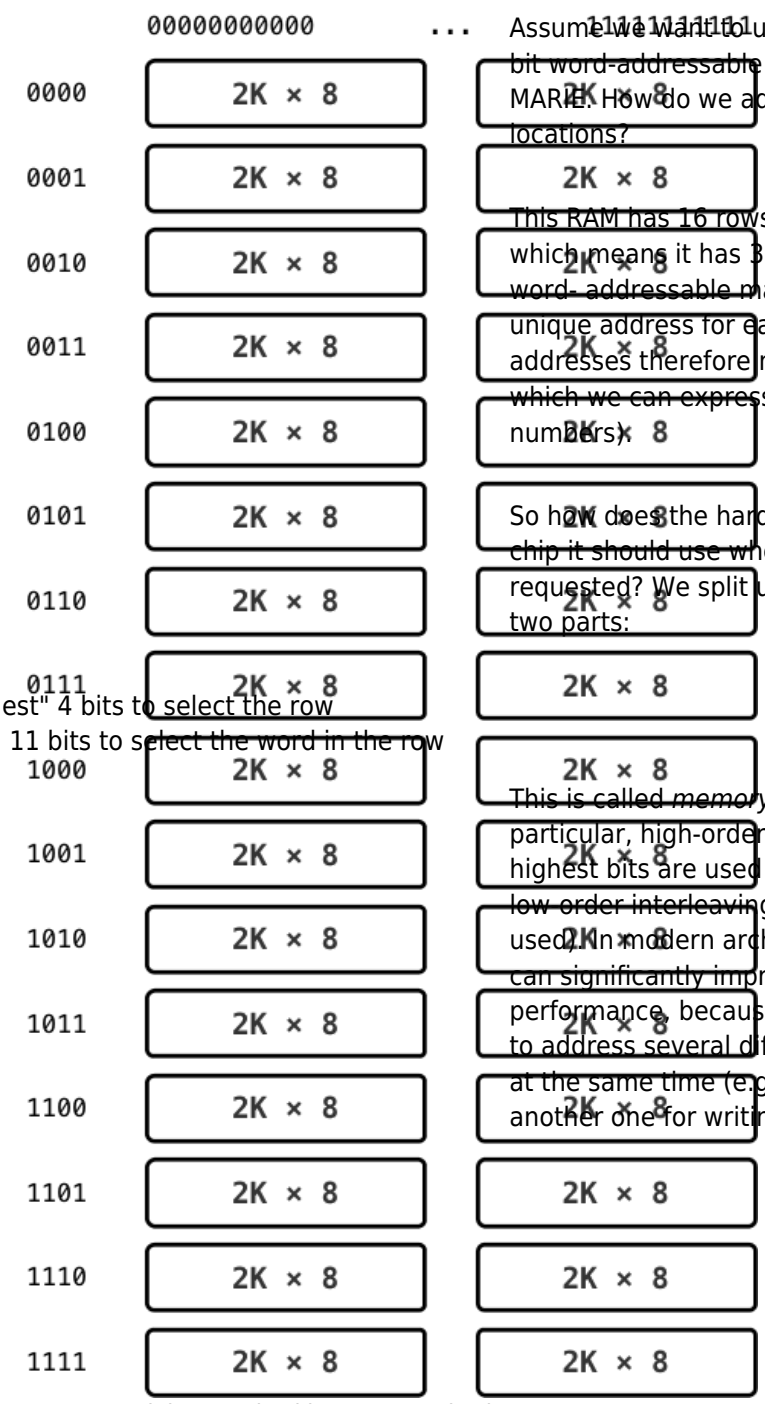


Micron MT4C1024. 1 mebibit (220 bit) dynamic ram. By Zeptobars, CC-BY 3.0 (https://zeptobars.com/en)

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 sixteen 2K × 8 chips to build a 20K × 16 memory module:



A RAM module organised into rows and columns

Assume we want to use this RAM with a 16-bit word-addressable architecture like MARIE. How do we address individual locations?

This RAM has 16 rows of 2K × 16 each, which means it has 32K = 2^{15} words. In a word- addressable machine, we need a unique address for each word. Our addresses therefore must have 15 bits (with which we can express 2^{15} different numbers).

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" 4 bits to select the row
- Use the "low" 11 bits to select the word 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).

5.2 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

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

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

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.

Interrupts

6
Booting the system

7
Operating Systems

8
Networks

9
Security