Computer systems: from numbers to processor

Kris Demuynck

19th January 2021

Abstract

In this book the principles of the inner workings of a computer are discussed. Knowledge of these principles is crucial to be able to install computers and networks. But also in order to develop correct working software.

This document consists of two parts. In the first part representations of different kinds of information are explained. First, numeral systems are described with an emphasis on the binary and hexadecimal system. Next, this document explains how computers can calculate with natural numbers, integers, and decimal numbers. Also, an introduction is given how computers represent characters.

In the second part a simple but complete working processor is constructed. It starts by explaining the basic components that make up a computer (logic gates) and shows how to combine them to create more complicated logic circuits. At the end all this is combined to construct the processor and to introduce the reader to assembly language. This allows for a greater understanding of higher level programming languages and how they map to operations executed on the processor.

Copyright

Kris Demuynck

Computer systems: from numbers to processor

Contents

Chapter 1

Numeral systems

In this chapter we will explore how numbers are represented in the computer. In general the binary numeral system is used, but frequently the octal and hexadecimal numeral systems will also be used to enhance human readability.

Further we will explain how to do calculations with binary numbers. It is important to realise that computers can suffer from overflow and rounding errors. This can have an big impact of the accuracy of the calculations.

1.1 Representation of numbers

We are so used to read and write numbers that it is hard to imagine that there are other ways to do that. Nevertheless, our way of writing down numbers is not the only possible way. In other times and other cultures different systems are used. However, numbers are always drawn as a sequence of symbols.

The easiest way of representing numbers is by drawing lines. The number 7 would then be written as |||||||. Since this notation becomes a burden for bigger numbers, other symbols are used to group a number of lines. The Egyptians already had symbols for groups of 5, 10, and 100. This was used by more modern societies. The Romans used V for 5, X for 10, and so on. An important disadvantage of this system is that there is no symbol for zero.

Originally, the order of these symbols did not matter. VII was the same as IVI of IIV. Later, the Romans started to write 4 as IV and since then, the order of the symbols mattered.

1.1.1 The decimal numeral system

In Western culture, the decimal numeral system is used. It consists of 10 symbols (which we call 'digits'): 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. A number is a sequence of those digits. The position of a digit in a number is important. If we write the number 44, the first 4 has a different value than the second 4. The first 4 will count 10 times more than the second one.

If one wants to know the value of a decimal number, the easiest way is to look at the digits from right to left. The most right one is called 'units', the one to the left is called 'tens', then the 'hundreds' are written, and so on. For example, the value of the number '123' is given by $3 \cdot 1 + 2 \cdot 10 + 1 \cdot 100$.

The factors we use to multiply each digit are all powers of 10 which is exactly the number of digits in this number system. That is very important.

A more abstract way of looking at the decimal system is as follows. Given a number that consists of digits $a_4a_3a_2a_1a_0$, its value is given by: $a_4 \cdot 10000 + a_3 \cdot 1000 + a_2 \cdot 100 + a_1 \cdot 10 + a_0$. This can be writen shorter:

$$\sum_{i=0}^{4} a_i \cdot 10^i$$

So more in general, the value of a number with $n+1$ digits is given by:

$$\sum_{i=0}^{n} a_i \cdot 10^i$$

Another way to look at the decimal numeral system is from the perspective of a counter. When one wants to count in decimal, one starts with all zeros. Imagine, there are 3 digits, then the counter will start with 000. Every time we want to add 1 to this counter, the right-most digit is increased, until there are no more digits left. Thus, the counter will go to 001, 002, 003, all the way till 009. If one wants to add another 1 to this counter, the right-most digit will be set to 0 again and the digit left to it will be increased. This renders 010 and we can count like this until 099 like this. Then, the reasoning is repeated: both 9's are set to 0 and the third digit (from the right) is increased. And so on.

1.1.2 The binary numeral system

Computers work with electrical signals. In order to create a computer that can calculate in the decimal numeral system, one would need 10 different electrical signals. This is not impossible, but very hard to do. Furthermore, such a computer would easily make mistakes as it could misunderstand a signal for another. That is why the binary numeral system was chosen. In this system, only 2 digits are used: 0 and 1. In practice they respond to 2 different voltages.

If there are only 2 digits, a number consists solely of 0's and 1's. In order to know the value of a binary number, one can use powers of 2 in stead of powers of 10. For example, the number 1011 corresponds to (from right to left): $1 \cdot 1 + 1 \cdot 2 + 0 \cdot 2^2 + 1 \cdot 2^3 = 1 + 2 + 8 = 11$.

More generally the value of a binary number with digits a_i is now given by:

$$\sum_{i=0}^{n} a_i \cdot 2^i$$

In order to indicate that the number 1011 is binary, we shall note this as: $[1001]_2$. The decimal value can then be written as $[11]_{10}$.

As you can see, knowing the powers of 2 can be a great help converting binary numbers into decimal. Complete the following table:

i	2^i	i	2^i
0	1	9	
1	2	10	1024
2	4	11	
3		12	
4		13	
5		14	
6		15	
7		16	
8	256	17	131072

Counting in binary is similar to counting in decimal, but now there are only 2 digits. If a counter starts at 000, the next value is still 001. But then, the rightmost digit will go back to 0 and increase the digit next to it. This renders 010 (which is the number 2 in decimal). In the following table, all possible values for this counter are shown:

binary	decimal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

As you can see, one needs more digits in binary then in decimal. These digits are also called 'bits'.

1.1.3 The hexadecimal numeral system

Binary numbers can become very lengthy. For instance, the decimal number 1000 needs 10 bits in binary. For humans this can become a problem as it is easy to miss a digit while reading or writing a number it. For this reason the hexadecimal numeral system is used.

In the hexadecimal numeral system there are 16 digits. These are: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. The characters A till F get values 10 till 15 respectively.

In order to find the value of a hexadecimal number, we shall now use powers of 16. For example, the number $[\text{AF4}]_{16}$ is equal to $4 \cdot 1 + 15 \cdot 16^1 + 10 \cdot 16^2 = 4 + 240 + 2560$ which is $[2804]_{10}$.

In general the value of a hexadecimal number with digits a_i is equal to:

$$\sum_{i=0}^{n} a_i \cdot 16^i$$

The value $[\text{AF4}]_{16}$ will also be written as '0xAF4'. In many programming languages, this notation is used.

It might seem a bit awkward to use 16 digits to write down a number. But in 1.2.3 at page 13 we will explain why this is so useful.

1.1.4 The octal numeral system

In the octal numeral system 8 digits are used. The reasoning is the same as before, which means that the value of an octal number is given by:

$$\sum_{i=0}^{n} a_i \cdot 8^i$$

This system is not used very often, but you will encounter it from time to time. One example is file permissions in Unix. They can also be expressed in the octal numeral system.

1.1.5 Other numeral systems

On could make a numeral system with any number of digits. You just need a finite set of symbols. If there are G symbols, each symbol gets a value $0, 1, ..., G-1$ respectively.

The value of a number with $n+1$ digits is now given by:

$$\sum_{i=0}^{n} a_i \cdot G^i$$

For example, take the number 8DH30 in the 18 numeral system. It's value (in decimal) can be determined as follows:

$0 \cdot 1 + 3 \cdot 18 + 17 \cdot 18^2 + 13 \cdot 18^3 + 8 \cdot 18^4 = 54 + 17 \cdot 324 + 13 \cdot 5832 + 8 \cdot 104976 = 921186$

The decimal value of number $[124]_5$ is given by:

$4 \cdot 1 + 2 \cdot 5^1 + 1 \cdot 5^2 = 4 + 2 \cdot 5 + 1 \cdot 25 = 39$

1.2 Conversion

Since we will work with many different numeral systems, it is important to be able to convert numbers to another system.

1.2.1 From any numeral system to decimal

This is very simple and was already discussed before (1.1.5) so we will not repeat that.

1.2.2 From decimal to any other numeral system

What if you start with a decimal number and you want to know it's value in another numeral system (say, binary or hexadecimal)?

In order to do this, we will use an 'algorithm'. This is a step-by-step plan or procedure that one can execute and it will render the right result.

In this case, the algorithm works as follows (we convert a number N to a G-numeral system):

1. Divide the number N by G. This results in a quotient (Q), but also a remainder (R). The remainder is what is left if you subtract $Q \cdot G$ from the N.
2. Write down the value of R. This is a digit of the number we are looking for.
3. If Q is greater than 0, go back to step 1, but use the value of Q (in stead of N).
4. If you reverse all the digits found, that will result in the value we are looking for.

An example is given to clarify this. Imagine you want to convert the number $[1234]_{10}$ to hexadecimal notation (N=1234 and G=16).

Starting in step 1, we divide 1234 by 16. This gives 77 and the remainder is 2 ($= 1234 - 16 \cdot 77$). Since 77 is greater than 16, we divide it again by 16. Now the result is 4 and the remainder is 13. If we divide the last result by 16, we get 0 and a remainder of 4. Now the algorithm stops because the quotient was 0. We found the remainders 2, 13, and 4 consecutively. If we reverse

this, we get 4, 13, and 2. The number 13 is written by the digit 'D' in hexadecimal. So the result is: $[4D2]_{16}$.

We can verify if this is correct by converting this number back to decimal: $2+13\cdot16+4\cdot16^2 = 1234$.

In order to make this conversion a bit more easy on paper, one can use the following scheme:

1234	2
77	13
4	4
0	

We start on the top left. The quotient is always written under the previous result. And the remainder is written on the right. The digits are all on the right from bottom to top.

This technique can be used to convert from decimal to any other numeral system. We could also convert the number 1234 to the 5 numeral system. This gives:

1234	4
246	1
49	4
9	4
1	1
0	

Hence, the result is: $[14414]_5$.

And if the number is converted to binary, we get:

1234	0
617	1
308	0
154	0
77	1
38	0
19	1
9	1
4	0
2	0
1	1
0	

And the result is: $[100\ 1101\ 0010]_2$.

Notice that all digits are always smaller than G. There can never be digits higher or equal to G.

1.2.3 Binary and hexadecimal

The hexadecimal numeral system has a very interesting property. It allows easy conversion to binary and vice versa. Let's explore this with an example.

In this example we start from the decimal number $[40102]_{10}$. If we convert this to hexadecimal and to binary, we get $[9CA6]_{16}$ and $[1001\ 1100\ 1010\ 0110]_2$ respectively.

Notice that we grouped the bits in groups of 4. Each group of 4 bits can be converted to hexadecimal separately and it will give the same result:

- $[1001]_2 = [9]_{10} = [9]_{16}$
- $[1100]_2 = [12]_{10} = [C]_{16}$
- $[1010]_2 = [10]_{10} = [A]_{16}$
- $[0110]_2 = [6]_{10} = [6]_{16}$

This means it is very simple to convert between binary and hexadecimal numeral systems. Just group the bits per four and convert them separately. There is no need to go through the decimal system if you know all 16 possibilities of 4 bits by heart. Complete the next table which gives all possible combinations of 4 bits:

0	0000	4		8	1000	C	
1	0001	5	0101	9		D	1101
2		6		A		E	
3		7		B		F	1111

Hexadecimal numbers are used a lot. In fact they are a short, more readable version of binary numbers. You will see them being used for memory addresses, characters, mac addresses, machine code, ...

In the same way, it is also easy to convert between the octal and the binary system. In this case, you group the bits by 3 (because $2^3 = 8$).

1.3 Negative numbers

The memory of a computer consists of a very long sequence of bits. A memory of 16GB consists of 137 438 953 472 bits! Each of those bits can be 0 or 1. These bits are grouped per 'byte' (8 bits) of per multiple of bytes (1, 2, 4 or 8 bytes). Each group can then be used to store a number. For instance, if you write the following code in java:

```
int a = 100;
```

the computer will use 4 bytes (32 bits) in memory to store the value of a. The contents of those bits will be 0000 0000 0000 0000 0000 0000 0110 0100. Notice that by using a fixed number of bytes, some numbers will not fit into this space. That is why every type in java has a certain 'range'. More information can be found in 1.4.1.

A positive number is thus easily stored in a group of bytes. But what if we want to store negative numbers? On paper we can just write a '-' sign in front, but in the memory of the computer there is only 0 and 1. That means we have to use some system to indicate that the number is negative. There are different ways to do this and they will be discussed in this section.

It is important to see that, given a certain amount of bits, one cannot know if this represents a positive or a negative value if you don't know which system was used. We will demonstrate this below, but it is already important to notice this. In programming languages, one defines the used system by specifying the 'type' of a variable (int, char, byte, ...). More information can also be found in 1.4 on page 19.

There are different ways to encode a negative number into bits. For every possibility, the following has to be taken into account:

- How many numbers can we represent with the given number of bits?

- How many negative and how many positive numbers can we represent?
- Are there multiple ways to represent the same number (preferably not)?
- Does the order of the numbers change when we regard the numbers as natural numbers?
- Is it easy to calculate when the numbers are represented a certain way?

We discuss four ways to represent negative numbers:

- sign bit: this will be used in chapter 2 on page 36
- 1-complement: this is not used a lot, but a great introduction to 2-complement
- 2-complement: this is the most used format to represent integers
- offset: this will be used in chapter 2 on page 36

1.3.1 Sign bit

Sign bit is the simplest way to format integers. It just uses 1 bit for the sign. For instance, if a value needs to be stored in 4 bytes, the first bit is reserved to hold the sign. A zero means that the number is positive, a 1 means it is negative. The remainder 31 bits can then be used for the value.

So in order to be able to store negative numbers, one bit is sacrificed. One can still store positive numbers, but the range is smaller (it is halved). In exchange negative numbers can also be stored.

The advantage of this system is that it comes very close to how humans write numbers. But there are also a number of disadvantages. For instance, one can write 0 in 2 ways: +0 and -0. Also, doing calculations is more difficult because one always has to check the sign. If you want to add 2 numbers, but one is negative, you should subtract them.

A simple example will clarify this system a bit further. Imagine one wants to store the number $[-100]_{10}$ to sign bit notation with 8 bits. In order to do this, we convert $[100]_{10}$ to binary, yielding $[110\ 0100]_2$. After this, the first bit is set to 1. The final result is then: 1110 0100.

Remark that one could also interpret this number as a positive value (when you don't know it was constructed with sign bit). In this case the value can be read as $[228]_{10}$. It is therefore very important to make sure you know the format that was used! In some programming languages this is achieved by specifying the type of a variable (like 'int' and 'unsigned' in the C programming language).

By the way, remark that the positive value $[100]_{10}$ is also representable with sign bit. The sign bit is in this case equal to 0. This results in: 0110 0100.

As a last example, all possible values of 3 bits are shown, together with their 'unsigned' value and the value using sign bit:

bit pattern	unsigned	sign bit
000	0	+0
001	1	1
010	2	2
011	3	3
100	4	-0
101	5	-1
110	6	-2
111	7	-3

As you can see, there are 2 possible ways to represent 0. Also, the order of the numbers interrupts between 3 and -0 and then reverses after that. There are, in fact, 2 jumps in the numbers. The first one is when the value goes from 3 to -0 and the second one happens when you add 1 to the last number. Since there are only 3 bits, all bits will flip back to 0. But the sign bit value jumps more than one (it goes from -3 to +0). We call these jumps also 'discontinuities'. Discontinuities make it difficult to do calculations with the numbers (more information will follow).

Generally, we get the following results using sign bit with n bits:

- there are $2^n - 1$ different values possible
- the range goes from $-2^{n-1} + 1$ (all ones) till $2^{n-1} - 1$ (a zero followed by all ones)
- the number zero has two representations: all zeros or a one followed by all zeros
- the order of the numbers changes when you look at the numbers as being unsigned (all negative numbers are in reverse order).
- in order to do arithmetic operations on the numbers, one needs to take into account the sign

1.3.2 1-complement

Another system to encode negative numbers is 1-complement. In this representation a negative number is stored as the inverted version of its positive value. The inverted is found by flipping all bits (all ones become zero and vice versa). The first bit is not used to store the number. As a consequence one can still use this bit to determine if the value is negative or positive.

For example, the number $[-100]_{10}$ will be represented with 8-bits 1-complement as follows. First, 100 is converted to binary (8 bits), resulting in: $[0110\ 0100]_2$. Then all bits are inverted. This yields: 1001 1011 which is stored in memory.

With this technique one can still determine the sign of the number looking at the first bit. For example, if 1111 0101 is stored in memory, the first bit indicates that this is a negative number. In order to find its positive value, all bits need to be inverted. This results in 0000 1010 which is the binary representation of the number decimal number 10. Therefore, the stored number is equal to $[-10]_{10}$.

Positive numbers are still stored as before, but the first bit cannot be used. For example the stored value 0110 1000 is just $[104]_{10}$.

If we put all possible values of 3 bits in a table, the following result is found:

bit pattern	unsigned	1-complement
000	0	+0
001	1	1
010	2	2
011	3	3
100	4	-3
101	5	-2
110	6	-1
111	7	-0

There are still 2 discontinuities (between 3 and -3, and between -0 and +0). The following properties hold:

- there are $2^n - 1$ different values possible
- the range goes from $-2^{n-1} + 1$ (a one followed by all zeros) till $2^{n-1} - 1$ (a zero followed by all ones)
- the number zero still has two representations: all zeros or all ones
- the order of the numbers is better than with sign bit (all negative numbers are now in the right order)
- in order to do arithmetic operations on the numbers, one still needs to take into account the sign

1.3.3 2-complement

One of the problems with the previous formats is the presence of two representations for 0. If you look closely at the 3-bit example in 1-complement, these two 0's are next to each other (if you add 1 to all ones, you end up with all zeros). Therefore, it would be interesting to shift all negative numbers one row down (111 will become -3 in this case). This has two advantages: the value of -0 disappears and it creates room for another number. The bit-pattern 100 can now be 4 or -4. Although one might be tempted to assign the number 4 to that pattern, in reality it is regarded as -4. In this way negative numbers can still be recognised by the first bit.

The resulting table with 3-bit patterns now becomes:

bit pattern	unsigned	2-complement
000	0	0
001	1	1
010	2	2
011	3	3
100	4	-4
101	5	-3
110	6	-2
111	7	-1

This is the 2-complement notation. So in order to represent a negative number, the positive value is still inverted, but also increased with 1. Positive numbers are still stored as before. For example, the number $[-100]_{10}$ will be constructed as follows (again 8 bits as in the previous sections). The binary value of $[100]_{10}$ was $[0110\ 0100]_2$. In order to change this into -100, all bits are inverted and 1 is added to it. This results in: 1001 1100.

This technique (taking the inverted and adding one) is also called calculating 'the 2-complement of a value'. It has an interesting property: if you apply it twice, you end up with the original number. For example, if you start with 1001 1100 ($[-100]_{10}$) and take its 2-complement, one finds 0110 0100 which is $[100]_{10}$ again.

Another interesting property of 2-complement notation is that one can do calculations with it without taking into account negative numbers. This is discussed more in detail in section 1.6 on page 26. A small example will already give some insight. Imagine one wants to calculate $[-1 + 2]_{10}$ using the 3 bits representation as in the table. In memory these numbers are stored as $[111]_2$ and $[010]_2$. If we regard these numbers as positive integers, they have values 7 and 2. Adding those to yields 9, which is $[1001]_2$ in binary. Since there are only 3 bits allowed, we delete the first bit and end up with $[001]_2$. This result is correct! This means we can do calculations with

negative numbers without having to take that into account at all (more information in 1.6). This is why 2-complement is the most used representation in computer science.

In short we find the following properties for 2-complement:

- there are 2^n different values possible
- the range goes from -2^{n-1} (a one followed by all zeros) till $2^{n-1} - 1$ (a zero followed by all ones)
- the number zero only has one representation: all zeros
- the order of the numbers is always preserved, except when the value goes from the highest value to the lowest
- in order to do arithmetic operations on the numbers, one can do those as if they were just positive integers

1.3.4 Offset

A fourth way to represent negative numbers is completely different. In order to be able to store negative numbers, a constant value (called the 'offset') is added to each number (also the positive ones). In other words, all numbers are shifted to a higher value. That value is stored in binary. In order to decode a stored value, one has to subtract the offset to obtain the original value.

The value of the offset is fixed and must be given. Otherwise it is impossible to know how to store values or to know which value was stored in memory.

Let's take the example we used before. We want to store the value $[-100]_{10}$ with 8 bits and an offset of 127. In this case, we add -100 and 127, resulting in 27. This is converted to binary with eight bits, yielding: 0001 1011. So the idea is to make all numbers positive so they can be stored like that. Remark that the first bit does not give any information about the sign any more!

If you want to represent +5 in the same format (8 bits, offset=127), then these two values are also added. This gives 132 and yields 1000 0100 when converted to binary.

When decoding the offset needs to be subtracted. For instance if the pattern 0011 1111 was encoded with an offset of 127, one converts the number first to decimal: $[63]_{10}$. Then the offset is subtracted to obtain the value: $63 - 127 = -64$.

The advantage of this technique is that one can choose how many negative and positive numbers are representable. The disadvantage is that the number 0 is not encoded as all zeros.

With 3 bits and an offset of 3, one gets the following table:

bit pattern	unsigned	offset
000	0	-3
001	1	-2
010	2	-1
011	3	0
100	4	1
101	5	2
110	6	3
111	7	4

The offset method has the following properties:

- there are 2^n different values possible
- the range goes from $-$offset (all zeros) till $2^n - 1 -$ offset (all ones)
- the number zero only has one representation: the binary representation of the offset
- the order of the numbers is preserved
- in order to do arithmetic operations on the numbers, one has to take into account the offset

1.4 The binary system in IT

As discussed before binary numbers are extremely important in the world of IT. All information is stored in memory using the binary numeral system. Every digit is called a 'bit' which is an abbreviation of 'binary digit'.

The memory of a computer (internal RAM, disk space, USB drives, ...) consists of billions of bits. Those bits are grouped per 8. Each group is called a 'byte'. One can also make groups of 16, 32 or 64 bits. In this case, the word 'word' is used to indicate such a group.

Since groups of 4 bits also make sense (because there is a link with hexadecimal), they also received a name. A group of 4 bits is called a 'nibble'.

When storing numbers in memory, the computer will always reserve a number of bytes for it. This means that the range of the number is always limited.

1.4.1 Data types in java

Java defines different types for variables. Depending on the type, a different amount of bytes is reserved in memory. For integer types java always uses 2-complement notation. The following table shows these types:

name	number of bytes	range
byte	1	-128 till 127
short	2	-32 768 till 32 767
int	4	-2 147 483 648 till 2 147 483 647
long	8	-9 223 372 036 854 775 808 till 9 223 372 036 854 775 807

If one wants to store a number that is outside the range of a given type, the number is truncated. This situation is called 'overflow'. The programmer will not always get an error!

In java there is no 'unsigned' version of the data types. Programming languages such as C do have that.

1.4.2 Converting numbers using java

In java one can easily convert numbers to another numeral system using the static method `Integer.toString(i, r)`. The number that needs to be converted is i and r is the number of digits in the other numeral system.

The following static methods can also be useful: `Integer.toBinaryString(i)`, `Integer.toHexString(i)`, `Integer.toOctalString(i)`. The class Long also contains these methods.

1.4.3 Literals in Java

In java one can define constant values in different ways. Normally the numbers are always in the decimal numeral system. However, it is also possible to give a value (also called a 'literal') in binary, hexadecimal or octal. This is done as follows:

```
int a = 123;         // decimal
int b = 0b10010010; // binary
int c = 0xFF;        // hexadecimal
int d = 012;         // octal
```

Note that octal number start with a 0. So never use decimal values with leading zeros in java. They will be interpreted as octal values!

1.4.4 Units

Because computers use the binary system, powers of 2 are also important.

This was already clear because bits are grouped in groups of 4, 8, 16, 32 or 64 bits. For larger amount of bits (or bytes), powers of two are also used.

For instance, 1024 bytes (2^{10}) is called a 'kilobyte'. The word 'kilo' does not mean 1000 in this case! In the following table you can see how many bytes are in each case:

name	abbreviation	amount
kilo	K	1 024
mega	M	1 048 576
giga	G	1 073 741 824
terra	T	1 099 511 627 776
peta	P	1 125 899 906 842 624

Due to commercial reasons, one kilo will sometimes be used to indicate 1000, which is even more confusing. A hard disk with a capacity of 1 TB should normally contain 1,099,511,627,776 bytes, but in practice there are only 1,000,000,000,000. A difference of almost 100 GB!

Network speeds are also given in powers of 10. A network with 1 Mbps will send 1 000 000 bits per second.

In order to overcome the confusion between powers of 2 and powers of 10, new names were introduced. They are used more and more frequently. For example, 1024 bytes is called a 'kibibyte'. If you see these names, you know it is expressed in powers of 2.

The following names are used for powers of 2:

name	abbreviation	amount
kibi	Ki	1 024
mebi	Mi	1 048 576
gibi	Gi	1 073 741 824
tebi	Ti	1 099 511 627 776
pebi	Pi	1 125 899 906 842 624

1.4.5 MSB en LSB

The abbreviations MSB and LSB are very important if you want to name certain parts of a binary number. MSB stands for 'Most Significant Bits' or 'Most Significant Bytes' (depending on the context). LSB stands for 'Least Significant Bits' or 'Least Significant Bytes'.

When sign bit was discussed, we talked about the 'first' bit of the number. But this is very ambiguous. We look at these numbers frequently from right to left, so what is then the first bit? We could also call it the 'left most' bit, but then bits are stored on 2 dimensional chips. There is no left and right there. In order to be clear, this bit is called the MSB because the weight (significance) of that bit in the number is the highest.

So if one considers the number $[1110\ 0100\ 1001\ 1011]_2$, the MSB is equal to one. The 4 MSB are equal to 1110. You can also say that the MSB is equal to 1110 0100. In the latter case the 'B' stands for 'byte'.

The LSB of the previous number is equal to 1, the 4 LSB are equal to 1011 and the LSB is equal to 1001 1011.

In the remainder of this course text MSB and LSB will always be used.

1.4.6 Little endian and big endian

As explained before numbers are stored in a number of consecutive bytes in memory. Every byte has an 'address' in memory. This means every byte has a location, which can also be expressed as a number.

For example a 32 bit integer will be stored in 4 bytes. The first byte is e.g. stored at location 0xC000, the second at location 0xC001, the third at location 0xC002 and the last at location 0xC003 (remark that we use hexadecimal numbers which is custom for memory locations).

But this can still be done in 2 ways: one can start with the MSB and then go to the LSB. Or one could start with the LSB. The latter is probably counter intuitive, but it is the most used way to store (and also send) numbers.

If a number is stored starting with the MSB, this is called 'big endian'. If one starts with the LSB, this is called 'little endian'.

This means that the 32-bit integer with value 0x12345678 is stored as follows:

- big endian: 0x12, 0x34, 0x56, 0x78
- little endian: 0x78, 0x56, 0x34, 0x12

Every processor has his own way to store numbers in memory. So it becomes very important to know this when computers are connected through a network. If they send a number with little endian of big endian can make a big difference.

Most computers these days use little endian to store numbers in memory. But to send them over a network, big endian is used more frequently.

1.5 Characters

Computers not only work with numbers. They also have to store texts which consist of characters. Characters must be converted to bits somehow. This is usually done with a table. The table maps all characters to a number. That number can then be stored in memory.

Through history many different tables have been used to associate characters with numbers. Therefore, one needs to make sure which table was used in order to interpret a text that was stored in memory. For example, the character 'A' can be represented by the number 65 in one table. But in another table it might get the number 193.

Notice that, next to having a table, one also needs to determine how to 'encode' the number into bits. If the numbers are always between certain borders, a fixed number of bits can be used. But numbers of characters can also be encoded in a variable amount of bits. This is discussed more in detail in 1.5.5.

1.5.1 Legacy systems

In the early days, the most used character set was called EBCDIC (Extended Binary Coded Decimal Interchange Code). This system is not used a lot any more. But it can still be encountered on some mainframes that are programmed in COBOL.

In this system, every character gets a value between 0 en 255. Therefore, each character can be encoded into 1 byte of memory.

The big disadvantage of this system is that characters are not put alphabetically in the table. This makes it difficult to sort words for instance.

1.5.2 ASCII

In the sixties a new character set emerged: the ASCII set (American Standard Code for Information Interchange). This character set contains 128 characters, so 7 bits are needed to represent a character. The table only contains characters that are used in the English language. Therefore, it was unsuitable for other languages.

The ASCII table is shown here:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0									\b	\t	\n			\r		
1																
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

The row number is octal and the column number is hexadecimal. The row specifies the 3 MSB of the number and the column specifies the 4 LSB of the number. Therefore, the character 'A' is associated with the binary number 100 0001 ($[41]_{16}$).

The first 32 entries in the table are special characters like backspace, tab, newline, and carriage return. There are other characters that were used to indicate certain events when characters need to be sent over a network. We will not discuss these special characters here as they are barely used any more. Character 32 ($[20]_{16}$) is a space.

There are a few interesting advantages about the ASCII set. First of all, the characters are alphabetically ordered. This means that the associated number can be used to determine if a character should come before of after another one (alphabetically). Capital characters are first

in the table. This is the reason why a lot of applications tend to always sort capital characters first.

A second advantage is that it is very easy to convert capital characters to lower case and vice versa. You just need to set one bit to 1 or 0 in order to do this. This can also be accomplished by adding or subtracting 32 from the number.

1.5.3 ISO 8859

ASCII characters are normally encoded in 8 bits. This means that the MSB is always 0. This bit is used to add other characters to the set. For instance, in order to write Latin languages such as French or Spanish, the set was extended with the following set:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8																
9																
A		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬		®	¯
B	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

This extension enables to write in a lot of languages such as Afrikaans, Albanian, Icelandic, Indonesian, Italian, Norwegian, Portuguese, Scottish, Spanish, Swahili, Swedish, and others. This set is called 'Latin-1' or 'ISO 8859-1' and is widely adopted.

However, there are a lot of characters not present in this set. If one wants to write a Russian text, for instance, another extension is needed. The Cyrillic characters have been put in an extension called 'Latin/Cyrillic' or 'ISO 8859-5'.

This means that you need to know the language in which a text was written, so one can use the right table to decode the characters. Another disadvantage is that it is impossible to mix certain languages in one document. For these reasons, the ISO 8859 standard was slowly abandoned and replaced by Unicode. However, some operating systems like Windows still support the ISO 8859 standard and can create a lot of confusion in an international context.

1.5.4 Unicode: code points

Exchanging information in an international context is key in our modern society. The ASCII data set and its extensions are not adequate. This is why another standard emerged in the nineties: 'Unicode' or 'ISO 10646'.

Unicode is also a table of characters, but there is no limit on how far it may grow. Every number associated with a character is called a 'code point'. In 2018 the largest code point had a value of 0x1FA6D, but new characters have been added since then. Unicode has characters for almost any language.

The first 128 code points correspond to the ASCII set. Code points are always noted in the same way: a capital 'U' followed by a 'plus'-sign followed by the hexadecimal value of the code point with at least 4 digits. For example, the word 'Hello' has the following code points:

U+0048 U+0065 U+006C U+006C U+006F

Since code points don't have a maximum value, storing them in a fixed number of bytes will result in loss of range. Therefore, more complicated encodings are used (such as UTF-8). They are described below.

1.5.5 Unicode: encodings

Code points need to be stored in memory and in files, which consist of bytes. Code points are therefore 'encoded' in those bytes. The UTF-8 encoding is probably the most used, but there are other ways as well.

1.5.5.1 UCS

A simple method to encode code points is to use a fixed number of bytes. In the early days of Unicode, people still believed that code points would not easily exceed the range of a 16 bit number. That is why people started to store each character in exactly 2 bytes. Java uses this encoding internally for the 'char' type.

This encoding is also known as 'UCS-2' (Universal Character Set). For example, the word 'Hello' could be encoded as the following consecutive bytes in a file (in hexadecimal):

00 48 00 65 00 6C 00 6C 00 6F

This uses big-endian notation (the MSB is written first). But one could also use little endian. In that case, one obtains the following sequence:

48 00 65 00 6C 00 6C 00 6F 00

As one can notice, a file consisting of only ASCII characters contains a lot of zeros and the file size is doubled. There is also a problem when these files need to be processed in older programming languages such as C. For example, in the C programming language the end of a string is indicated with a byte equal to 0. This means that it is not always possible to use existing software to process UCS-2 files.

If one wants to store code points greater than U+FFFF one could increase the number of bytes to 4. This is called 'UCS-4' but it is not frequently used as the disadvantages of UCS-2 are even more expressed in this encoding.

1.5.5.2 UTF

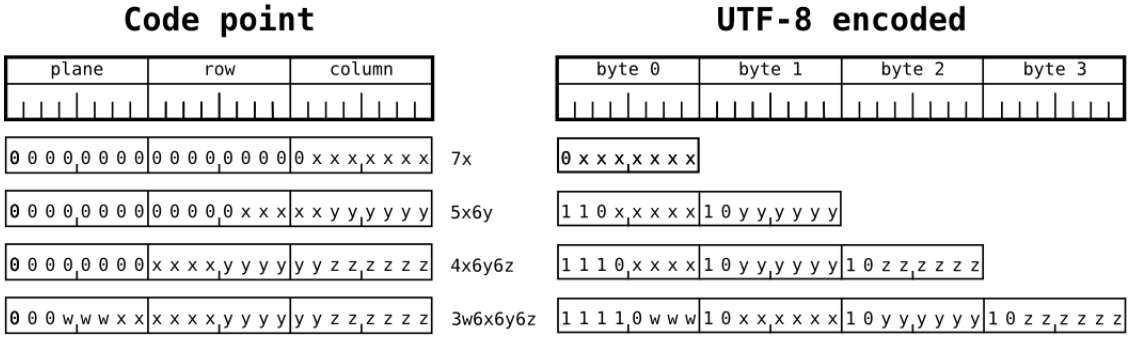
Backward compatibility with ASCII is very important when it comes to encoding code points. UTF (Unicode Transformation Format) is an answer to that. There are two variants: UTF-8 and UTF-16. We will only discuss UTF-8 because it is the most frequently used format.

UTF-8 works as follows:

- if the code point has a value between U+0000 and U+007F (max 7 bits and thus within the ASCII range), the code point is encoded in 1 byte. The most significant bit of this byte will be set to 0. This means that all ASCII characters fit in one byte and old software will have no problem reading these from a file. So for example, the character 'A' with code point U+0041 will be encoded as one byte: 0100 0001 (or 41 hexadecimal).

- if the code point lies between U+0080 and U+07FF (max 11 bits) 2 bytes are used. The first byte will begin the the bit sequence '110' and the second byte will begin with '10'. The other bits of the two bytes are used to store the code point (the 6 least significant bits will go in the second byte and the other bits in the first byte). An example might help to understand this. The Greek character 'alpha' (α) has code point U+03B1. This is 1110110001 binary. It will be encoded in two bytes. It is easier to start with the second byte. That will start with the bits 10 and continue with the 6 LSB of the code point (110001). The second byte will thus be 10110001 (B1 hexadecimal). The first byte will begin with 110 and the other (up till 5) bits of the code point. In this case, there are only 4 bits left: 1110. So we add a zero to become 5 bits: 01110. The first byte will therefore be 11001110 (CE). Concluding, the character alpha will be encoded as (hexadecimal): CE B1.
- if the code point lies between U+0800 and U+FFFF (max 16 bits) 3 bytes are used. The first byte will begin with '1110' and the next two bytes will begin with '10'. The six least significant bits of the code point will be put in the last byte, the next 6 bits in the second byte and the most significant bits will be encoded in the first byte. For example, the character with code point U+0B87 will be encoded as E0 AE 87.
- if the code point lies between U+10000 and U+10FFFF (max 21 bits) 4 bytes are used. The first byte will begin with '11110' and the next three bytes will begin with '10'. The bits of the code point are encoded in the four bytes similar to the previous reasonings. For example, the character with code point U+10330 will be encoded as F0 90 8C B0.

The following scheme shows all discussed possibilities:



The left shows the four cases for the code points. Bits that are not 0 are marked with x, y, z, and w. On the right one can see where the indicated bits go in the encoded version.

The disadvantage of UTF-8 is that one cannot know the number of characters in a file by counting the bytes. The only way to determine this is to go through the file. Fortunately, the first byte of each character is easily recognisable:

- if the most significant bit is 0, only one byte was used
- if the three most significant bits are '110', two bytes were used
- if the four most significant bits are '1110', three bytes were used
- if the five most significant bits are '11110', four bytes were used
- if the two most significant bits are '10', this cannot be the start of a character

Next to UTF-8 there is also UTF-16 and even UTF-32 but they are less frequently used.

In order to indicate the encoding in a file, one can (optionally) let it start with a special character, called 'BOM' (Byte Order Mark). This character corresponds to code point U+FEFF. It is encoded and put at the start of the file. A program that reads the file can recognise the BOM and determine the encoding from that. The file can start with the following bytes:

- EF BB BF: this is the BOM encoded with UTF-8
- FF FE: this is the BOM encoded with UCS-2 little endian
- FE FF: this is the BOM encoded with UCS-2 big endian

1.6 Binary arithmetic

Doing calculations with binary numbers is very simple. One only needs the table of multiplication for 0 and 1! Below we will show how to do the basic operations in binary.

1.6.1 Additions and subtractions

In order to see how 2 binary numbers are added, we can first take a look at how this is done in decimal. If one needs to add 2 decimal numbers one starts by writing those numbers underneath each other. Then one starts by adding the least significant digits. If the result is greater than 9, a one is carried over to the next significant digits. After this the next two digits are added until all digits have been added. Here you see a small example:

	1		1	
		6	2	8
+		5	3	4
	1	1	6	2

When adding two binary numbers, the principle stays the same. One starts with adding the least significant bits. In this case, if the result is greater than 1, a one is carried over to the next bits. This means that (for the second and next significant bits) one has to add 3 bits: the two bits that need to be added and the bit that was carried over. The latter is called 'carry bit'. An example is given here: adding $[48]_{10}$ and $[28]_{10}$ in binary (with 8 bits) results in the following:

carry		0	0	1	1	0	0	0	0
48	=		0	0	1	1	0	0	0	0
+ 28	=		0	0	0	1	1	1	0	0
76	=		0	1	0	0	1	1	0	0

The last carry bit was 0 is this case. However, it can also be 1. The computer will drop this bit, as the result needs to fit in 8 bits again. If the numbers were unsigned, the result will be wrong because we exceeded the range of the data type.

But an interesting thing happens when we use 2-complement numbers. Let's add the numbers -4 and 28 in 8 bit 2-complement:

carry		1	1	1	1	1	1	0	0
-4	=		1	1	1	1	1	1	0	0
+ 28	=		0	0	0	1	1	1	0	0
24	=		0	0	0	1	1	0	0	0

In this case, the result is correct when we drop the most significant carry bit! Because of this property 2-complement numbers are so popular in programming languages. It means we can do a subtraction by just taking the 2-complement and perform an addition. In terms of hardware this means there only needs to be a circuit that can add numbers. Subtraction is automatically available.

But it does not always work. Let's look at this example:

carry		1	0	0	0	0	0	0	0
-116	=		1	0	0	0	1	1	0	0
-126	=		1	0	0	0	0	0	1	0
-242	$\neq$		0	0	0	0	1	1	1	0

The result is not correct. This is because it is impossible to represent -242 with 8 bit 2-complement. The range of the data type was exceeded. This is called 'overflow'. The problem is that most programming languages will not give you an error message! They will just give you the wrong result. This is extremely important when you program with integer types!!! Always be aware of the range of the numbers that you are using!

Overflow cannot be detected by looking at the most significant carry bit. The following example illustrates this:

carry		0	1	1	1	1	1	1	0
127	=		0	1	1	1	1	1	1	1
+ 10	=		0	0	0	0	1	0	1	0
137	$\neq$		1	0	0	0	1	0	0	1

In this case, no bit was dropped but it is still an overflow because the result is -119 (in 2-complement). So how can one know if an overflow occurred or not? Well, take a look at the two most significant carry bits in each of the examples. If they are equal, there was no overflow. When they are different, overflow occurred. One can prove this but it is left to the reader to do this.

1.6.2 Multiplication

There are different approaches possible to multiply binary numbers. Three ways will be discussed. The first one cannot be implemented by a computer, but it will give more insight on how the next two algorithms work.

1.6.2.1 The long multiplication method

In order to understand binary multiplication, it is good to look at decimal multiplication first. Most people have learned to do this in school. Imagine we want to multiply the numbers 456 and 142. We start by writing these number underneath each other. As a first step 456 is multiplied by 2 (the least significant digit of 142). Then, 456 is multiplied by 4 and 'shifted' one digit to the left. Finally 456 is multiplied by 1 and shifted 2 places to the left. The three found numbers are then added together to get the result. It looks like this:

			4	5	6
×			1	4	2
			9	1	2
	1	8	2	4	
+	4	5	6		
	6	4	7	5	2

Binary numbers can be multiplied in the same way, but things get a lot easier as only 1 and 0 can occur. This means that one only has to be able to multiply by 0 and by 1. For example, multiplying 6 by 42 gives:

6	=									0	0	0	0	0	1	1	0
× 42	=									0	0	1	0	1	0	1	0
										0	0	0	0	0	0	0	0
									0	0	0	0	0	1	1	0	
								0	0	0	0	0	0	0	0		
							0	0	0	0	0	1	1	0			
						0	0	0	0	0	0	0	0				
					0	0	0	0	0	1	1	0					
				0	0	0	0	0	0	0	0						
+			0	0	0	0	0	0	0	0							
252	=	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0	0

Notice that the number of bits in the result is doubled. In this text, we will always put the result in double the amount of bits to avoid overflow. But in practice, the most significant bits will be dropped to fit the result back into memory. This means that overflow is an even bigger problem when multiplying binary numbers!

The described technique only works with positive integers. But it is easy to adapt it to 2-complement. This is done as follows. We start with 2 numbers in 2-complement notation. Each number takes n bits.

The next step is to take the 'sign extend' of these numbers. This is obtained by putting n bits in front of the number. The bits all have the value of the sign bit of the original number. For example '0011' will become '00000011' and '1010' will become '11111010'. The idea is to double the amount of bits without changing the 2-complement value of the number.

Now you can multiply the new numbers as before. Any bits that appear beyond the 2n bits can be dropped in this process.

As an example, we multiply -4 by 3 with 4 bits (i.e. '1100' and '0011' in 2-complement). The sign extend of these numbers is '11111100' and '00000011' respectively. Now we can multiply as before (dropping any bits that appear beyond the 8th position):

-4	=	1	1	1	1	1	1	0	0
× 3	=	0	0	0	0	0	0	1	1
		1	1	1	1	1	1	0	0
		1	1	1	1	1	0	0	
		0	0	0	0	0	0		
		0	0	0	0	0			
		0	0	0	0				
		0	0	0					
		0	0						
+		0							
-12	=	1	1	1	1	0	1	0	0

As you can see, the result is correct.

1.6.2.2 The long multiplication algorithm

In order for a computer to be able to multiply two numbers the technique described above has to be converted into an 'algorithm'. This is a plan of action containing different steps to be taken.

If the steps are followed rigorously, the result will be correct. Computers are extremely good at executing algorithms.

The 'long multiplication algorithm' has the following steps (multiplying two n-bit numbers a and b):

1. create a number 'A' of $2n$ bits and store the sign extend of a in it.
2. create a number 'B' of $2n$ bits and store the sign extend of b in it.
3. create a number 'P' of $2n$ bits en store the value 0 in it.
4. repeat the following steps $2n$ times:
   (a) inspect the least significant bit of B
   (b) if this bit is equal to 1 perform: P = P + A
   (c) shift A on digit to the left (called a 'shift left')
   (d) shift B one digit to the right (called a 'shift right')
5. the result is now in P

If you look closely to this algorithm, it follows the same steps as in the previous section. By shifting B to the right with each step, all bits of B are examined one by one. In this way, there is no need to keep a counter to know with which bit of B we have to multiply A.

Let's demonstrate this with an example. Take $a = 5$ en $b = -8$. We use 8 bits per number ($n = 8$). The result will be 16 bits long.

The number 5 is '0000 0101' and the number -8 is '1111 1000' in 2-complement 8-bit notation.

We start with steps 1-3 and find the following results:

1. A = 0000 0000 0000 0101
2. B = 1111 1111 1111 1000
3. P = 0000 0000 0000 0000

Now we execute the loop 16 times (2 times n). After each step, we note the value of P, A an B (in reality the values are overwritten):

P	A	B
P_0=0000 0000 0000 0000	0000 0000 0000 0101	1111 1111 1111 1000
P_1=0000 0000 0000 0000	0000 0000 0000 1010	0111 1111 1111 1100
P_2=0000 0000 0000 0000	0000 0000 0001 0100	0011 1111 1111 1110
P_3=0000 0000 0000 0000	0000 0000 0010 1000	0001 1111 1111 1111
P_4=0000 0000 0010 1000	0000 0000 0101 0000	0000 1111 1111 1111
P_5=0000 0000 0111 1000	0000 0000 1010 0000	0000 0111 1111 1111
P_6=0000 0001 0001 1000	0000 0001 0100 0000	0000 0011 1111 1111
P_7=0000 0010 0101 1000	0000 0010 1000 0000	0000 0001 1111 1111
P_8=0000 0100 1101 1000	0000 0101 0000 0000	0000 0000 1111 1111
P_9=0000 1001 1101 1000	0000 1010 0000 0000	0000 0000 0111 1111
P_{10}=0001 0011 1101 1000	0001 0100 0000 0000	0000 0000 0011 1111
P_{11}=0010 0111 1101 1000	0010 1000 0000 0000	0000 0000 0001 1111
P_{12}=0100 1111 1101 1000	0101 0000 0000 0000	0000 0000 0000 1111
P_{13}=1001 1111 1101 1000	1010 0000 0000 0000	0000 0000 0000 0111
P_{14}=0011 1111 1101 1000	0100 0000 0000 0000	0000 0000 0000 0011
P_{15}=0111 1111 1101 1000	1000 0000 0000 0000	0000 0000 0000 0001
P_{16}=1111 1111 1101 1000	0000 0000 0000 0000	0000 0000 0000 0000

At the end the numbers A and B should always be 0. The result is in P_{16} and is equal to -40. This is the correct result.

1.6.2.3 Booth's algorithm

Executing a multiplication involves a lot of steps and takes a lot of time to perform. In the example above 16 additions and 32 shift operations of 16-bit numbers needed to be performed. If one wants to multiply two 64-bit numbers, 128 additions and 256 shift operations need to be done. Additions are very costly: a 128 bit addition takes at least 128 times more time than a 128 bit shift operation! The reason is that the carry bits need to ripple through. So before one can add the last bits, one has to have added the previous bits first. In a shift operation all bits can be calculated independently. In hardware this can be done in parallel.

Andrew Booth noticed in 1951 that binary numbers are mostly made up of groups of ones and zeros. Positive numbers very often start with a lot of zeros and negative numbers mostly contain a big chunk of ones at the start. He developed an algorithm that, on average, needs less additions, due to this property.

In order to grasp this, let's look at a decimal multiplication. Say that we want to multiply $[134{,}357{,}646{,}785]_{10}$ with $[999{,}999]_{10}$. Using a long multiplication, this would take a lot of time. But if you look closely, we can rewrite 999,999 as 1,000,000-1. This means we need to multiply the multiplicand first by one million and then subtract it 1 time. Multiplying by a million is simple: one shifts the number six times to the left. So the whole multiplication can be done by doing on shift left operation and one subtraction.

We can generalise this to bigger numbers. If we want to multiply by 990,990, this can be replaced by (1,000,000 - 10,000 + 1,000 - 10). So we only need 4 shift left operations and 3 additions (we regard a subtraction as an addition since we work with 2-complement).

In the binary numeral system this also works. For example, $[1111\ 1100]_2$ can be rewritten as (1 0000 0000 - 100) and $[11\ 1000\ 1100\ 0111]_2$ is equal to (100 0000 0000 0000 - 1000 0000 0000 + 1 0000 0000 - 100 0000 + 1 000 - 1). Notice that, going through the bits from LSB to MSB, one needs an addition every time there is a change from 1 to 0 and a subtraction every time there is a change from 0 to 1.

This is what Booth's algorithm does: it scans the multiplicand from LSB to MSB. Every time the bit goes from 1 to 0 an addition is performed. When the bit goes from 0 to 1 a subtraction is done. If the bits stay the same, no operation is done. The algorithm is given here (we want to multiply the n-bit 2-complement numbers a en b):

1. Create 3 numbers A, A', and P. Each number can hold up to $2n + 1$ bits.
2. Set the n most significant bits of A to the value of a. The $n + 1$ least significant bits are set to zero.
3. Set the n most significant bits of A' to the 2-complement value of a. The $n + 1$ least significant bits are set to zero.
4. Set the n most significant bits of P to zero. Then set the next n bits to the value of b. The last (least significant) bit is set to zero.
5. Perform the next steps n times:
   (a) examine the 2 least significant bits of P
   (b) if they are equal to '01', calculate the following: $P = P + A$

(c) if they are equal to '10', calculate: $P = P + A'$

(d) Shift the number P 1 bit to the right. When doing this, the most significant bit of P should be set to the sign bit of the original value of P i.e. the sign of P should remain the same, even after shifting. This is called an 'arithmetic shift right'. A normal shift right will always introduce a zero on the left, but an arithmetic shift right retains the sign of the number. In fact, an arithmetic shift right is exactly the same as dividing the number P by 2.

6. Delete the least significant bit of P. The result is now in P.

Let's illustrate this using the same example as we used with the long multiplication algorithm. Imagine we want to multiply $[5]_{10}$ en $[-8]_{10}$ using 8 bit 2-complement binary notation (i.e. 0000 0101 and 1111 1000).

First, we create A, A' en P:

1. A = 0000 0101 0000 0000 0
2. A' = 1111 1011 0000 0000 0
3. P = 0000 0000 1111 1000 0

After this, we perform the steps (a) till (d) 8 times. The following table shows how the value of P evolves with each step (the values of A and A' never change):

value of P	operation
P_0 =0000 0000 1111 1000 0	$P_1 = P_0 >> 1$
P_1 =0000 0000 0111 1100 0	$P_2 = P_1 >> 1$
P_2 =0000 0000 0011 1110 0	$P_3 = P_2 >> 1$
P_3 =0000 0000 0001 1111 0	$P_4 = (P_3 + A') >> 1$
P_4 =1111 1101 1000 1111 1	$P_5 = P_4 >> 1$
P_5 =1111 1110 1100 0111 1	$P_6 = P_5 >> 1$
P_6 =1111 1111 0110 0011 1	$P_7 = P_6 >> 1$
P_7 =1111 1111 1011 0001 1	$P_8 = P_7 >> 1$
P_8 =1111 1111 1101 1000 1	

If one deletes the least significant bit of P_8, the result is 1111 1111 1101 1000. Converting this back to decimal yields $[-40]_{10}$. This is correct.

Notice that only one addition was needed and the loop was only performed 8 times. This algorithm is therefore much faster than the long multiplication algorithm.

1.6.3 Division

One can also divide 2 binary integers. The technique used is very similar to the long division. It can be performed by doing shift operations and subtractions. When dividing a dividend by a divisor, the result consists of a quotient and a remainder. For example, when dividing 100 (dividend) by 3 (divisor), the result is 33 (quotient) and the remainder is 1.

A possible algorithm for a division is as follows (two n-bit numbers a and b are divided):

1. create a number Q which is n bits long and set its value to zero
2. create a number R which is n bits long and set its value to zero

3. create a number A which is n bits long and set its value to a
4. repeat n times:
   (a) shift R one bit to the left
   (b) if the MSB of A is equal to 1 then $R = R + 1$
   (c) shift A one bit to the left
   (d) shift Q one bit to the left
   (e) if $R >= b$ then: $R = R - b$ and $Q = Q + 1$

The result of the division will be in Q and R. Q is the quotient and R is the remainder.

This algorithm only works with positive numbers. If you want to adapt this to 2-complement, you need to make the numbers positive first, and then convert the result back to negative if necessary.

1.7 Exercises

1.7.1 Converting between numeral systems

Convert the following numbers to the given numeral system. Verify your result by doing the inverse conversion:

1. $[123]_4 = [\ldots]_{10}$
2. $[123]_5 = [\ldots]_{10}$
3. $[123]_{16} = [\ldots]_{10}$
4. $[123]_3 = [\ldots]_{10}$
5. $[1100\ 0111]_2 = [\ldots]_{10}$
6. $[\text{A1BC}]_{16} = [\ldots]_{10}$
7. $[1100\ 1010\ 1111\ 1110]_2 = [\ldots]_{16}$
8. $[\text{BABE}]_{16} = [\ldots]_2$
9. $[12345]_{10} = [\ldots]_{20}$
10. $[34567]_{10} = [\ldots]_{14}$
11. $[891]_{10} = [\ldots]_3$
12. $[465]_{10} = [\ldots]_7$
13. $[750]_8 = [\ldots]_2$
14. $[110\ 100\ 000]_2 = [\ldots]_8$

1.7.2 Negative numbers

1.7.2.1 Representations

Make a table with all four-bit binary numbers in one column. Now add new columns in which you write the decimal value of each bit pattern, using the following representations respectively: unsigned, sign bit, 1-complement, 2-complement, and offset (take offset=7).

1.7.2.2 From decimal to binary

Convert the following numbers to 8 bits representations:

decimal	sign bit	1-complement	2-complement	offset (240)
-10				
-15				
42				
192				
-230				
-128				

1.7.2.3 From binary to decimal

Convert the following numbers to the decimal numeral system for each of the systems.

binair	sign bit	1-complement	2-complement	offset (240)
1100 0101				
0001 1001				
1111 1111 1101 0100				
1111 1001 0100 0101				
0000 0011 0101 0110				
1000 0001				

1.7.2.4 Range

What is the range of the following numbers?

1. 8 bits 2-complement
2. 8 bits 1-complement
3. 4 bits sign bit
4. 4 bits 2-complement
5. 16 bits offset 32 767
6. 16 bits offset 127
7. 32 bits 2-complement

1.7.3 Unicode and UTF-8 encoding

Given the following words:

- the English word ‘Computer’
- the French word ‘élève’ (which means ‘student’)
- the Greek word ‘καλημύχτα’ (which means ‘good night’)
- the Czech word ‘čtyři’ (which means ‘four’)

- the Russian word 'работа' (which means 'to work')
- the Turkish word 'çay' (which means 'tea')

First find all the code points for each character. You can use the the website http://unicode.org/charts for this (there are many other sites that can also be useful).

Then convert all code-points to UTF-8.

1.7.4 Unicode and UTF-8 decoding

In a file, the following bytes are stored in UTF-8 (hexadecimal):

EF BB BF D0 BA D0 BE D0 BC D0 BF D1 8E D1 82 D1 8A D1 80 20 69 73 20 63 6F 6D 70 75 74 65 72 0A E9 9B BB E8 85 A6 20 6F 6F 6B 20 E2 98 BA

Decode these bytes and determine which characters were stored.

1.7.5 Adding and subtracting

Perform the following calculations using 8 bits 2-complement notation. Convert the result to decimal to verify its value. Also, determine if overflow occurred or not.

1. $50 + 50$
2. $50 - 8$
3. $58 - 100$
4. $15 - 32$
5. $-10 + 52$
6. $-10 - 120$
7. $123 + 80$

1.7.6 Multiplying

Perform the following multiplications (the number of bits per number is given). Try both the long multiplication algorithm and Booth's algorithm. Compare both algorithms. Convert the result to decimal to verify its value.

1. 6×7 (4 bits per number)
2. $6 \times (-7)$ (4 bits per number)
3. $20 \times (-4)$ (6 bits per number)
4. $(-40) \times 42$ (8 bits per number)

Try to do the third assignment with 32 bit numbers. Which method is the most efficient?

1.7.7 Calculations in java

Given the following program in Java:

```
int a = 46400;
int b = a*a;
System.out.println("b = " + b);
```

What is the value of b? Why is it not correct? Can you explain the result? How can you rectify this?

1.7.8 Divisions in java

Given the following program in Java:

```
int a = 32700;
int b = (a / 1000) * 500;
System.out.println("b = " + b);
int c = (a * 500) / 1000;
System.out.println("c = " + c);
```

What result do you expect for b and c? What are the results? Which one is correct? Can you explain why this happens?

1.7.9 Dataypes in java

What datatype in java is the most appropriate for:

a) the resolution of a display in pixels

b) the size of a hard disk in bytes

c) the size of a RAM memory in bytes

d) the distance between the planets and the Sun in meter

e) the total distance driven with a car in km

Chapter 2

Real numbers

In this chapter we will introduce a way to represent real numbers in the computer. Special attention needs to be payed to rounding numbers and errors that are made while performing operations.

2.1 Real numbers in binary

Writing real numbers in binary is very similar to writing them in the decimal numeral system. In the decimal system, digits behind the 'decimal point' are given a weight. This weight is a power of 10 (just as the digits before the decimal point), but now the powers are negative. For instance, the first digit behind the decimal point has a weight of 10^{-1} (one tenth), the second has a weight of 10^{-2} (one hundredth), and so on.

In binary every bit behind the 'decimal point' is given a weight as a power of 2 in stead of 10. For instance, the number $[0.111]_2$ is equal to $2^{-1} + 2^{-2} + 2^{-3} = 1/2 + 1/4 + 1/8 = 0.875$.

More generally, a binary number with digits $a_n a_{n-1} \ldots a_1 a_0 . a_{-1} a_{-2} \ldots a_{-m+1} a_{-m}$ will have a value of:

$$\sum_{i=-m}^{n} a_i \cdot 2^i$$

The a_i are always 0 or 1.

Another example illustrates this: the number $[101.101]_2$ is equal to $2^2 + 2^0 + 2^{-1} + 2^{-3} = 4 + 1 + 0.5 + 0.125 = 5.625$.

Note that $[101.101]_2$ is NOT equal to 5.5! This is a mistake that many people make: they think all the digits behind the decimal point can be interpreted as a number and convert that into decimal. That is not correct!

2.2 From decimal to binary

In this section two ways of converting decimal real numbers to binary are discussed: the iterative and the direct method.

2.2.1 Iterative method

In order to convert a decimal number to the binary numeral system it is first split in 2 parts: the 'integer part' and the 'fractional part'. The integer part can be converted to binary as discussed in 1.2.2 on page 12.

The fractional part is converted as follows:

1. Multiply the number by 2. This will render a new number that lies between 0 and 2 (the original fractional part lies between 0 and 1)
2. Write down the integer part of the new number. This can only be 0 or 1 and is a digit of the number we are looking for
3. Now delete the integer part so we end up again with only a fractional part
4. If the number is not 0 and one has not found enough digits, go back to step 1

All found digits can be written behind the decimal point in the order they were found.

For example, we will convert $[10.625]_{10}$ to binary. First the number is split into 10 and 0.625. The first number can be converted as before and renders $[1010]_2$. The fractional part (0.625) is multiplied by 2. This is 1.25. This means that the first digit behind the decimal point will be 1 (the integer part of 1.25). Now we take the fractional part of the last result (0.25) and multiply this again by 2. This gives 0.5, meaning that the next digit is 0. Multiplying this another time by 2, we get 1. This is the last digit of the number. The fractional part of the result will thus be 0.101. Combining this with the integer part of the original number, we can conclude that $[10.625]_{10}$ is equal to $[1010.101]_2$.

This method can also be put into a scheme:

0.625	1.25
0.25	0.5
0.5	1
0	

The result on the right is always the number on the left multiplied by 2. The numbers on the left are the fractional part of the number on the right, but one row higher.

Note that at some point we found all digits behind the decimal point. However this will not always be the case. Sometimes there will be an infinite number of digits. This happens i.e. converting the number $[0.2]_{10}$ to binary:

0.2	0.4
0.4	0.8
0.8	1.6
0.6	1.2
0.2	0.4
0.4	0.8
0.8	1.6
0.6	1.2
0.2	0.4
0.4	0.8
0.8	1.6
...	...

The result is: $[0.001100110011\ldots]_2$ which is infinitely long! In practice the computer will have a limited number of bits to store the number. So the number will be broken off at some point. This means that 0.2 will not be stored exactly in the computer which will result in calculation errors (as we shall see later).

2.2.2 Direct method

In this section another method is presented to convert a decimal real number to binary. It is called the 'direct method' because the number will not be split into two parts. This method requires that you know how many binary digits behind the decimal point you want in the result.

The number is multiplied by 2^n where n is the number of binary digits behind the decimal point. Then the integer part of this number is converted to binary. Now the decimal point is put in the right place (after the nth digit starting from the LSB).

Imagine you want to convert $[10.2]_{10}$ to binary with 20 digits behind the decimal point. With this method we multiply the number by 2^{20} (1048576). This yields 10695475.2. We convert the integer part of this result to binary: $[1010\ 0011\ 0011\ 0011\ 0011\ 0011]_2$. If we put the decimal point after the 20th bit (starting from LSB) we get: $[1010.00110011001100110011]_2$. This result is correct!

2.3 Fixed point representation

In order to store real numbers in memory a certain amount of bits will be allocated for it. Since there are only ones and zeros in memory there is no way to indicate where the decimal point should go. One way to solve this problem is to put the decimal point at a fixed predefined place. This representation is called 'fixed point'.

For example we could allocate 8 bits for a real number and agree that there are 3 digits behind the decimal point. The decimal point is in this case fixed after 5 digits. In order to store the number $[101.00101]_2$ in this representation it is converted to 5 bits before the decimal point and 3 behind yielding $[00101.001]_2$ (and effectively losing one bit precision since we can only store 3 bits behind the decimal point). This is stored, without the point in memory: '00101001'. Remark that one needs to know the representation in order to determine the value of these bits. If it is interpreted as an unsigned number, it can be read as 41.

Also remark that we can only represent positive real numbers like this. In order to store negative real numbers, one has to agree on a system to do that. Sign bit is very popular in combination with fixed point.

Fixed point numbers have some very nice properties. One is that doing calculations is very easy.

If one wants to add two fixed point numbers the numbers can be added as if there was no decimal point. For instance, adding 5.25 and 3.5 can be done as follows:

carry		0	0	1	1	1	0		0	0			
5.25	=		0	0	1	0	1	.	0	1	0	=	42
+ 3.5	=		0	0	0	1	1	.	1	0	0	=	28
8.75	=		0	1	0	0	0	.	1	1	0	=	70

On the right side one can see what happens if the numbers are regarded as integers. The first value is 42 and the second is 28. If one adds these two, the result is 70. If one introduces the decimal point in this number, the result is 8.75. So in order to add two fixed point numbers, one can use the integer addition as discussed previously.

Multiplying is also extremely easy. One can do the multiplication ignoring the decimal point. But now the result will contain 2 times too many digits behind the decimal point (multiplying 2 numbers with 3 fractional digits will result in a number with 6 fractional digits). This means that we have to drop bits in order to get the same number of fractional bits as we started of with. This is done by a 'shift right' operation. So if there are n bits behind the decimal point, one has to shift the result n bits to the right. Remark that we can loose bits like this. A multiplication with fixed point numbers will therefore not always be exact!

For example we can multiply 5.25 by 3.5 (of the previous example) by just multiplying 42 by 28. This will give 1176, or $[100\ 1001\ 1000]_2$ binary. This result is shifted 3 digits to the right yielding: 10010011. If we interpret this result as a fixed point number with 3 fractional digits the value is $[18.375]_{10}$. This is correct (in this case no bits were lost since the 3 LSB were equal to 0):

5.25	=				0	0	1	0	1	.	0	1	0	=	42
× 3.5	=				0	0	0	1	1	.	1	0	0	=	28
18.375	=	1	0	0	1	0	.	0	1	1	0	0	0	=	1176

Dividing fixed point numbers is also easy. In this case the divisor is first shifted to the left n bits and then the division is done ignoring the decimal point.

So if we want to divide 5.25 by 3.5 we calculate:

5.25	=	1	0	1	.	0	1	0	0	0	0	=	336
/ 3.5	=		0	0	0	1	1	.	1	0	0	=	28
1.5	=		0	0	0	0	1	.	1	0	0	=	12

The computer will calculate 336/28 and store the result (12) in 8 bits. If this value is interpreted as a fixed point number, the result is correct.

There are also drawbacks on using fixed point. In the above representation one cannot store integers greater than 5 bits long even though integers don't have fractional digits. Also, if one wants to store a number with only fractional bits, one looses all bits before the decimal point. It would be more interesting to move the decimal point to the most optimal point, according to the number stored. This is done in floating-point representation.

2.4 Floating point

Putting the decimal point at a fixed position is easy but is not desirable. Some numbers (very big or very small ones) will suffer from this. It would be better to put the decimal point at the most optimal place depending on the number stored.

This can be achieved by storing two numbers: the number without the decimal point and a number indicating where the point must be placed. In fact, the first number can be regarded as a fixed point number with the decimal point after the first digit. The second number will indicate if that point needs to be shifted to the right (positive number) or to the left (negative number). This is called 'floating point'.

This format is also used on calculators. It is called 'scientific notation'. The number 123456 will be displayed as '1.23456E05'. As you can see it consists of two numbers (1.23456 and 05). The first number is called the 'mantissa' and the second is called 'exponent'. Remark that the exponent can be positive of negative. When the exponent is negative, the whole number can still be positive. If one wants to be able to store negative numbers an extra sign bit needs to be added.

A negative exponent always gives numbers that are smaller than one. For instance the number 1.23456E-03 is equal to 0.00123456. When the exponent is zero, the mantissa does not have to be changed (1.23456E00 is equal to 1.23456).

In order to store floating-point numbers in memory one has to agree on a lot of things:

- how many bits are used for the mantissa?
- how is a negative number represented?
- how many bits are used for the exponent?
- how is a negative exponent represented?

In the seventies and eighties every computer manufacturer had their own implementation of floating-point numbers. This resulted in a lot of problems when those computers needed to communicate. Therefore a standard was created. These days almost all computers adhere to this standard. It is discussed in the next section.

2.5 IEEE 754

In 1985 a standard was developed to represent floating-point numbers in memory. This standard is known as 'IEEE 754' and is used in almost any computer system these days.

The basic format is as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sign	exponent					mantissa									

the MSB is used as sign bit for the whole number. The next bits are used to store the exponent and the last bits hold the mantissa. The exponent is stored with offset notation. There are different versions depending on the total number of bits:

name	precision	sign bit	exponent bits	mantissa bits	offset	total
binary16	half precision	1	5	10(+1)	15	16
binary32	single precision	1	8	23(+1)	127	32
binary64	double precision	1	11	52(+1)	1023	64
binary128	quadruple precision	1	15	112(+1)	16383	128
binary256	octuple precision	1	19	236(+1)	262143	256

The standard also includes representations for decimal numbers, but this is not discussed in this document as they are not used frequently.

Remark that the mantissa will always begin with a 1. This is because one always puts the decimal point behind the first significant digit in the number to become the mantissa. In binary there is only one significant digit (not equal not zero) possible: 1. This means that the mantissa always starts with a one. Therefore one has chosen to not store this bit. This explains the (+1) in the table. For instance binary16 has 10 bits for the mantissa but one has to add '1.' before it to become the real value of the mantissa. So in fact there are 11 bits in the mantissa.

2.5.1 The value of a IEEE 754 number

In this section we give an example to clarify how numbers are stored. Imagine the number [1100 0000 1011 0000 0000 0000 0000 0000] is found in memory and one knows that it holds a IEEE 754 number. Since there are 32 bits in this number it uses the binary32 format.

In order to know its value, we split the number into sign, exponent, and mantissa:

- The sign bit is 1 so the number is negative
- The exponent is 10000001. This is 129 in decimal. As the exponent was stored using offset notation we need to subtract the offset (127 in this case) from this number. The real exponent is thus 2.
- In order to know the mantissa, we put '1.' before the mantissa-bits from the given number, yielding 1.011000...

Now we can calculate the value stored. The decimal point in the mantissa needs to be shifted 2 times to the right (positive exponent). This gives: 101.1. Now we can convert this back to decimal, yielding 5.5. The number stored in the above bit is thus equal to -5.5.

2.5.2 From decimal to IEEE 754

It is also possible to convert a decimal number into IEEE 754. Say we want to convert $[15.28]_{10}$ into binary16 format.

The first thing to do is convert the number to binary. This yields: 1111.01000111101011100001... The mantissa of binary16 consists of the first 11 bits of this number with the decimal point behind the first digit: 1.1110100011. We can leave out the MSB of this number since it is always equal to 1. The mantissa will be stored as '1110100011'.

The exponent will be 3 in this case because the decimal point in the mantissa needs to be shifted 3 places to the right in order to become the original number. The number 3 will be stored with an offset of 15 yielding 10010.

The number is positive, so the sign bit will be equal to 0.

Finally the number $[15.28]_{10}$ will be stored as [0 10010 1110100011] in binary16 format.

Converting a number from decimal to floating point requires a lot of computing power. Reading a text document containing 15 million real numbers will take a lot of time. If the numbers are stored in binary this will go a lot faster.

Remark that the number $[15.28]_{10}$ was not stored exact. Rounding errors were made. If we convert the number [0 10010 1110100011] back to decimal we obtain the value $[15.2734375]_{10}$ (verify this as an exercise). This is close, but not equal to the original number!

2.5.3 Denormalisation

There is only one big problem with the previously described format: how to represent 0? As you always put '1.' before the stored mantissa, there is no way to end up with only zeros.

In order to address this problem the standard allows for special cases. These cases are indicated with the maximum and the minimum value for the exponent. If the exponent is minimal (only zeros) then one must put '0.' in front of the mantissa in order to get the right value. This allows for even other numbers to be represented. This is called a 'denormalised' floating point.

When the exponent is maximal (all ones) three other 'numbers' can be represented depending on the mantissa:

- if the mantissa is 0 the value of the floating-point number is equal to + or - infinity
- if the mantissa is not 0 the value is 'NaN' which means 'Not a Number'. This is used when the result of a calculation does not render a valid result (e.g. the square root of -1).

In binary16 this gives the following results:

- 0 is represented by 0 00000 0000000000 (all zeros)
- + infinity is represented by 0 11111 0000000000
- - infinity is represented by 1 11111 0000000000
- NaN is represented by x 11111 xxxxxxxxxx (the x's form a number that is not equal to 0)

This table gives an overview:

exponent	mantissa	meaning
1 till max-1	whatever	normalised: put '1.' in front of the mantissa
0	whatever	denormalised: put '0.' in front of the mantissa
max	0	+ or - infinity (depends on the sign bit)
max	not 0	NaN

2.6 Floating point in java

Java uses the IEEE 754 floating-point numbers for the types 'float' and 'double'. A float is always a binary32 and a double is a binary64.

Java also supports the denormalised versions as well as infinity and NaN. The following predefined functions can help:

```
boolean isNaN = Double.isNaN(doubleValue);
boolean isNaN = Float.isNaN(floatValue);
boolean isInfinity = Double.isInfinity(doubleValue);
boolean isInfinity = Float.isInfinity(floatValue);
```

Java also defines some constants:

```
Float.NaN
Float.POSITIVE_INFINITY
Float.NEGATIVE_INFINITY
Double.NaN
Double.POSITIVE_INFINITY
Double.NEGATIVE_INFINITY
```

Using the following functions, one can copy the bits of a float or double into an int or long:

```
long bits = Double.doubleToRawLongBits(doubleValue);
int bits = Float.floatToRawIntBits(floatValue);
```

Strings can be converted to float or double using:

```
float result = Float.parseFloat(“3.14”);
double result = Double.parseDouble(“3.14”);
```

And floats and doubles can also be converted back to strings:

```
String result = Float.toString(floatValue);
String result = Double.toString(doubleValue);
```

Because floating-point numbers can cause a lot of errors, java foresees another datatype that is more exact. It is called ‘BigDecimal’. BigDecimals are special data structures in which a decimal number is stored. They are always exact. You can use them as follows:

```
BigDecimal a = new BigDecimal(“3.141592”);
BigDecimal b = new BigDecimal(“0.010324”);
BigDecimal c = a.multiply(b);
System.out.println(“c = ” + c);
```

Remark that using BigDecimal calculations will need a lot more time. Use them only when precision is more important than speed!

2.7 Exercises

2.7.1 Decimal - Binary

convert the following decimal numbers to the binary numeral system:

1. 5.5
2. 100.625
3. 3.1415926535 (the result only needs to have 5 fractional bits)
4. 6.283185307 (the result only needs to have 5 fractional bits)
5. 0.03125
6. 3.25
7. 8.414 (the result only needs to have 10 fractional bits)

2.7.2 Binary - Decimal

Convert the following binary numbers to decimal:

1. 1011.10011
2. 11.0101
3. 1001.1001101

2.7.3 Convert to fixed point

Convert the following decimal values to fixed point with 6 integer bits and 10 fractional bits:

1. 5.5
2. 100.625
3. 3.1415926535
4. 6.283185307

2.7.4 Calculations with fixed point

Calculate the following in fixed point with 4 integer bits and 4 fractional bits:

1. $[100.1011 + 10.11]_2$
2. $[100.1011 - 10.11]_2$
3. $[10.10 \cdot 1.1]_2$
4. $[11.1 \cdot 10.0]_2$

2.7.5 Convert to floating point

Convert the following decimal numbers to binary16 and to binary32:

1. 42
2. 0.03125
3. -3.25
4. 100.625
5. 3.1415926535
6. 8.414
7. -10.5
8. -32768
9. -32770
10. -1
11. 0

2.7.6 Convert from fixed point and floating point

Here are 5 16-bit numbers. They can be interpreted in different ways. Which value to they have when you interpret them as 2-complement, fixed point (with 5 fractional bits), and binary16 respectively?

1. 1100 0010 0000 0000
2. 1011 1101 0000 0000
3. 0100 0100 0000 0000
4. 1000 0101 1101 1001
5. 0010 0110 0100 1110
6. 0100 0100 0101 0110
7. 0000 0000 0000 0000
8. 1111 1111 1111 1111

2.7.7 Floating point in java

Write a java program that verifies the following (try it with float and with double):

1. $0.1 + 0.2 = 0.3$?
2. $0.1 + 0.3 = 0.4$?
3. $0.1 + 0.1 + 0.1 = 0.3$?
4. $0.1 + 0.1 + 0.1 + 0.1 + 0.1 = 0.5$?
5. $(0.1 + 0.2) + 0.3 = 0.1 + (0.2 + 0.3)$?
6. $(2.0 \cdot 0.1)/3.0 = 2.0 \cdot (0.1/3.0)$?
7. $1/0 =$?
8. $-1/0 =$?
9. $\sqrt{-1} =$?
10. $\cos\left(\frac{\pi}{2}\right) = 0$?

Which properties hold and which do not? Is double always better than float?

2.7.8 Errors in java code

What is wrong in the following code?

1. The following code prints 'no' while the result of $\cos(\pi/2)$ should be 0. How come? And how to resolve this issue?

```
double d = Math.cos(Math.PI/2);
if (d==0.0) System.out.println("yes"); else System.out.println("no");
```

2. What result is printed by the following code? What did you expect? What went wrong? Why?

```
float f = 1000000F;
for(int i=0; i<100000000; i++) {
    f += 0.01F;
}
System.out.println("f = " + f);
```

Chapter 3

Logic circuits

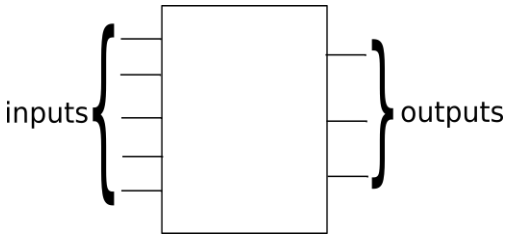
In this chapter a new important concept in introduced. Logic circuits are the building blocks that make up a computer. First the basic notion is introduced. The next chapters build further on these concepts and build a complete functional processor from these circuits.

3.1 Definition

A 'logic circuit' is an apparatus (or machine) with a number of inputs and outputs.

Every input and output can only have a value of 0 or 1. So in fact all inputs form a binary number and so do all outputs. Logic circuits will convert the input number to an output number. In theory this happens momentarily (if the input changes, so does the output). In practice however some time is needed for the output to become ready. This time is called 'propagation delay'. It will determine the maximum clock speed that is attainable by the computer. In this text however we will neglect propagation delay in most cases.

Logic circuits can be seen as a kind of black box:

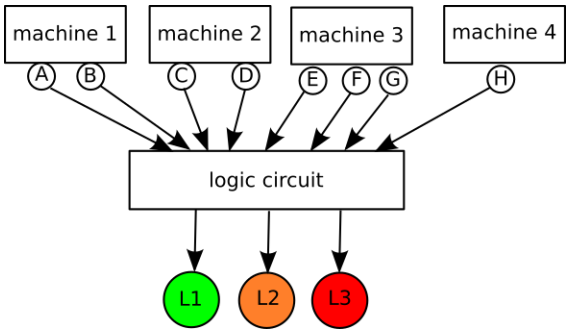


The inner workings of a logic circuit van be described by a 'truth table'. This is a table that lists all possible inputs and their corresponding output values. As an example imagine an alarm system in a plant. There are a number of machines in the plant with sensors. The sensors measure pressure, temperature, forces and other things that are important for the machines to work properly. The sensor will output 0 if the values are within normal parameters. And it will output a 1 when the values are outside those parameters. In order to know the status of the machines a logic circuit is used. Its inputs come from all the sensors. Its output consists of 3 bits:

- the first output is one when everything is working normally.
- the second output becomes 1 when some sensors indicate a problem, but the problem is not severe. It needs to be looked into.

- the third output becomes 1 when a critical error has occurred and everything needs to be shut down.

If there are 8 sensors, one can draw the system like this:



The output is determined by the 8 inputs. The truth table could look like the following:

A	B	C	D	E	F	G	H	L1	L2	L3
0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	1	0	1	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	0	1	1	0	0	1
...	...	...	...	...	...	...	...	...	...	...

It contains all possible values for the inputs A, ..., H and the corresponding output values L1, L2, and L3. As one can see, the number of lines in this table can be very large. In this case 256 lines must be put in the table to cover all possible inputs. In general one needs 2^n lines where n is the number of input. If there are 32 inputs the number of lines increases to 4 billion! Therefore logic circuits are mostly build up by smaller ones. Combining them can lead to very complex systems.

3.2 Logic gates

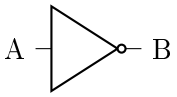
Logic circuits will always be built out of smaller ones. So it is interesting to take a look at the most simple logic circuits that can be made. These are called 'logic gates'. Most of them have 2 inputs and 1 output. Except the first one, which only has 1 input and 1 output.

We shall see that all logic circuits can be built using these logic gates.

3.2.1 The NOT gate

The 'NOT gate' is the most simple meaningful logic circuit possible. It only has 1 input and 1 output. The behaviour is simple: it inverts its input. If the input is 0, the output will be 1 and vice versa.

A NOT gate is drawn as follows:



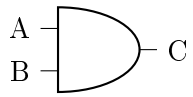
The truth table of a NOT gate is given by:

A	B
0	1
1	0

3.2.2 The AND gate

An 'AND gate' is a logic circuit with 2 inputs and 1 output. The output will only be one if both the inputs are one. Otherwise the output will be zero.

The AND gate is drawn as follows:



One can also state that this gate is a kind of operation. It 'AND's the bits together'. We could write this as C=A AND B but a better notation will be introduced later.

The truth table of an AND gate contains 4 rows and is given by:

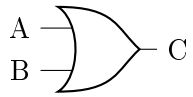
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

One can also make an AND gate with more than 2 inputs. In that case the output will be one if all inputs are one simultaneously.

3.2.3 The OR gate

The 'OR gate' has two inputs and one output. The output is one as soon as at least one of the inputs is one.

The OR gate is draw as follows:



The truth table of an OR gate is given by:

A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

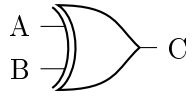
If an OR gate has more than 2 inputs, the output will only be zero when all inputs are zero.

3.2.4 The XOR gate

The 'XOR gate' has two inputs and one output. The output will be one if either of the inputs are one, but not both at the same time.

One can also describe the behaviour as follows: the output is one if the inputs are different.

The XOR gate is drawn as follows:



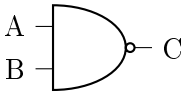
The truth table of a XOR gate is:

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

3.2.5 The NAND gate

The 'NAND gate' does the inverse of an AND gate. Its output will be zero if both inputs are one. Otherwise it outputs a one.

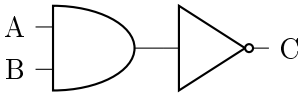
In a circuit the following symbol is used:



The truth table is given by:

A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

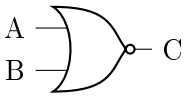
In fact a NAND gate can be built using an AND gate and a NOT gate as follows:



3.2.6 The NOR gate

Similar to the NAND gate, one can also construct the inverse of an OR gate: the 'NOR gate'. It will emit a one only when both inputs are zero.

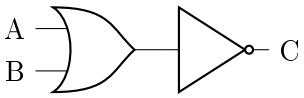
The NOR gate is drawn as:



The truth table of a NOR gate is given by:

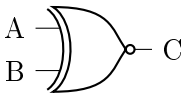
A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

If one wants to construct a NOR gate using an OR gate and a NOT gate, one gets:



3.2.7 The XNOR gate

The 'XNOR gate' is the inverse of an XOR gate. It will output a one when both inputs are equal:

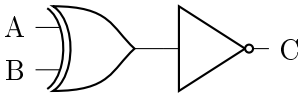


The truth table of an XNOR gate is given by:

A	B	C
0	0	1
0	1	0
1	0	0
1	1	1

One can also regard the XNOR gate as an 'XAND gate', but it is always referred to as an XNOR gate.

Constructing an XNOR gate using an XOR gate and a NOT gate is straightforward:



3.3 Implementing a logic gate

Logic gates will be the building blocks to build larger logic circuits. This means if one can make the logic gates, one can create any logic circuit (such as a computer).

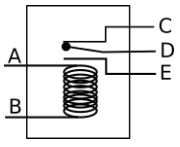
There are many different ways to implement logic gates. One can use transistors (which is the most used component these days), but one can also use completely different approaches. There are mechanical versions, game-of-life implementations, domino implementations, optical ones, ones that use vacuum tubes, relays, and so on.

In this section some popular implementations are discussed.

3.3.1 The relay implementation

Relays were used in the early 20th century (around 1930) to make simple computers. Although they are slow and consume a lot of electricity, they are easy to understand and ideal to explain the inner workings of a logic gate without requiring too much knowledge of electronics.

A relay is an electric device that has the following schematic:



It contains a solenoid (also called 'coil' or 'inductor') and a switch (on the right-upper side of the schematic).

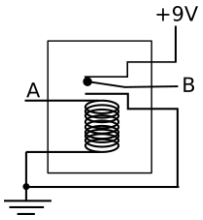
The contacts A and B are connected to the coil. If no electric current is flowing through it, the coil doesn't do anything. The switch stays in the position that is drawn. This means that electric current can flow from C to D or from D to C.

If electric current flows through the coil, it becomes a magnet. The magnet will pull the switch so the middle connection (D) flips down and makes contact with E. So now there is no connection between C and D anymore. But current can flow from D to E or from E to D.

Relays are not only used to make computers. They are also used in thermostats to turn on or off the heater. In cars they are used to turn on or off the petrol pump.

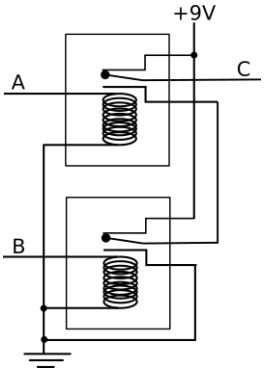
In this section two gates are constructed with relays so the reader can see its inner workings and that it is really possible to make functional gates.

The first example is a NOT gate, using one relay. The relay in this example needs a voltage of 9 volts to operate. The minus of the battery represents a logical 0 and the plus represents a logical 1. The minus is sometimes also called 'ground' or '0 volt' and is represented by the horizontal bars on the bottom of the following circuit:



If the connection A is connected to the ground (the input is in this case 0) no current flows through the coil. This means that the output B is connected through the relay with the +9 Volts, resulting in an output equal to 1. If A is connected to +9V (the input becomes 1) the switch in the relay will flip connecting the output B with the ground (representing a 0).

A NAND gate can be constructed using two relays:

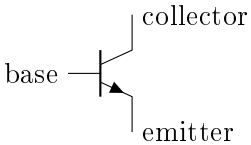


We leave it up to the reader to verify the workings of this circuit (just try all possible inputs and see what happens with the relays).

3.3.2 Transistors

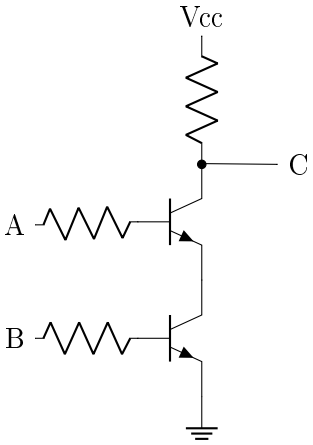
In most cases logic circuits are made with transistors. This text does not focus on electronics but for completeness the principle is shown here.

A transistor has 3 connections: emitter, base and collector. Its symbol is shown here:



A transistor can be used as a switch. Normally no current can flow between emitter and collector. But is there is a small current flowing from base to emitter, the transistor 'opens' and current can flow from collector to emitter.

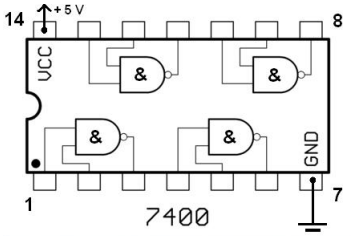
As an illustration a circuit is shown of a NAND gate using transistors (this is called 'TTL logic'):

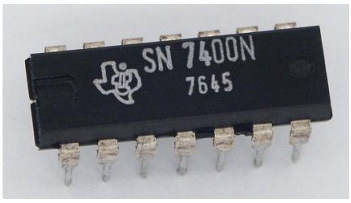


3.3.3 TTL integrated circuits

One can also buy logic circuits in integrated circuits. An example of this is the 7400 chip. It contains 4 NAND gates. The chips needs 5 Volts to operate. A voltage between 0 and 0.8 Volts is interpreted as a logical 0 and a voltage between 2.8 and 5 Volts is interpreted as a logical 1.

The pinout of this chip is as follows:





These chips can be bought at very low prices. Modern processors are nothing more than a combination of billions of these chips.

3.4 Exercises

3.4.1 Gates with relays

Try to make a circuit that implements the following gates using relays:

- OR gate
- AND gate
- XOR gate

3.4.2 All possible gates

The following table shows all possible truth tables with two inputs. It represents all possible gates with 2 inputs and 1 output.

A	B	X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}	X_{11}	X_{12}	X_{13}	X_{14}	X_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Can you recognise the gates described before? How would you name the other gates? Try to find a suitable name for them and put them in the following table. Some gates are already given as an example.

poort	betekenis
X_0	ZERO
X_1	
X_2	
X_3	$\overline{A}$
X_4	
X_5	
X_6	
X_7	
X_8	AND
X_9	
X_{10}	
X_{11}	
X_{12}	
X_{13}	$A \geq B$
X_{14}	
X_{15}	

Chapter 4

Boolean algebra

4.1 Definition

Logic gates are combined in order to create more complicated logic circuits. When designing a logic circuit one wants to find the most optimal implementation, using the least number of gates possible. This can become very complicated. One way to address this problem is to use 'Boolean algebra'. A 'Boolean expression' is a formula representing a logic circuit.

In order to understand this, a notation is introduced in which each gate is regarded as a mathematical operation. For instance, an AND gate can be seen as an operation done on its inputs. In fact this operation is exactly the same as a multiplication. Therefore an AND operation is written as a multiplication.

An OR operation is almost the same as an addition except for the case in which both inputs are 1. Because of its similarity with the addition, an OR operation is written down as an addition (with a + sign). It is important to know that it is not really an addition. It is just a notation.

An XOR operation is also similar to an addition but + is already taken. So the symbol $\oplus$ is used.

Finally a NOT gate, which only has one input, is represented as a bar over the input.

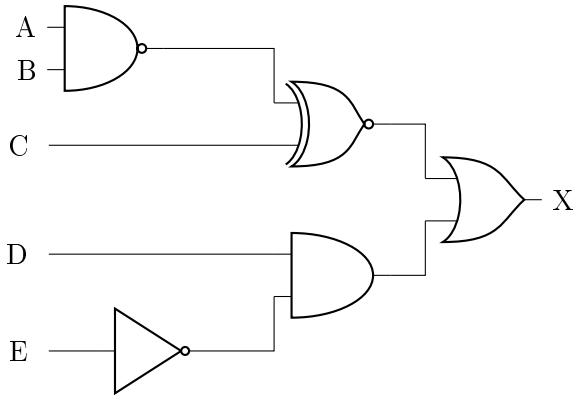
In short the following notation is used:

- $\overline{A}$ =NOT A
- $A + B$= A OR B
- $A \cdot B$= A AND B
- $A \oplus B$= A XOR B

A and B are this inputs of the gates, so they can only be 0 or 1.

Using this notation one can also write down other gates. A NAND gate will be written as: $\overline{A \cdot B}$ (the inverse of A AND B).

By combining all operators one can create complicated circuits in a very compact way. For example, the formula $X = \overline{\overline{A \cdot B} \oplus C + D \cdot \overline{E}}$ corresponds to the following circuit:



The operations are evaluated in the same order as in normal algebra: the AND operation has a higher precedence than the OR operation. XOR and OR are evaluated from left to right. Using brackets this order of evaluation can be altered or made more clear.

The following equations demonstrate the order of precedence of the operators:

- $A + B \cdot C = A + (B \cdot C)$
- $A + B \oplus C = (A + B) \oplus C$
- $A \oplus B + C = (A \oplus B) + C$

4.2 De Morgan's laws

There are many interesting properties in Boolean algebra. Two of them are called 'De Morgan's laws':

- $\overline{A + B} = \overline{A} \cdot \overline{B}$
- $\overline{A \cdot B} = \overline{A} + \overline{B}$

They show a connection between AND and OR operations. One can convert on into the other by 'cutting' or 'pasting' the NOT on top of the variables. This can be very handy if one only has a certain type of gates available.

Verify these properties by completing the following truth tables:

A	B	$\overline{A + B}$	$\overline{A} \cdot \overline{B}$	$\overline{A \cdot B}$	$\overline{A} + \overline{B}$
0	0				
0	1				
1	0				
1	1				

These laws can also be applied to programming. As an example the following two lines of code in Java are equivalent:

- `if (!(a == 0 && b == 1)) {}`
- `if (a != 0 || b != 1) {}`

4.3 Simplifying Boolean expressions

Using Boolean algebra it is possible to simplify a Boolean expression. In order to do so, one can use De Morgan's laws but in addition the following properties hold:

$A + A = A$	$A \cdot A = A$
$A \cdot (B + C) = A \cdot B + A \cdot C$	$A + (B \cdot C) = (A + B) \cdot (A + C)$
$A + \overline{A} = 1$	$A \cdot \overline{A} = 0$
$A + 1 = 1$	$A \cdot 0 = 0$
$A + 0 = A$	$A \cdot 1 = A$
$A + A \cdot B = A$	$A \cdot (A + B) = A$
$A + \overline{A} \cdot B = A + B$	$A \cdot (\overline{A} + B) = A \cdot B$

Remark that the left and right column are very similar. The only difference between the two is that OR is replaced by AND (and vice versa) and 1 is replaced by 0 (and vice versa). Because of this peculiarity AND and OR are called 'dual operations'.

In addition to the previous properties, the following are also true:

- $\overline{\overline{A}} = A$
- $A \oplus B = \overline{A} \oplus \overline{B}$
- $A \oplus B = \overline{A} \cdot B + A \cdot \overline{B}$
- $\overline{A \oplus B} = A \cdot B + \overline{A} \cdot \overline{B}$

These properties can be used to simplify a Boolean expression. An example is given here:

$$\begin{aligned} X &= \overline{\overline{A} \cdot B} \cdot C + A \cdot \overline{C} \\ &= (\overline{\overline{A}} + \overline{B}) \cdot C + A \cdot \overline{C} \\ &= (A + \overline{B}) \cdot C + A \cdot \overline{C} \\ &= A \cdot C + \overline{B} \cdot C + A \cdot \overline{C} \\ &= A \cdot (C + \overline{C}) + \overline{B} \cdot C \\ &= A \cdot 1 + \overline{B} \cdot C \\ &= A + \overline{B} \cdot C \end{aligned}$$

There are many different ways to simplify a Boolean expression, but taking the following steps works most of the time:

1. convert any occurrence of '$\oplus$" to a combination of '$+$' and '$\cdot$' using $A \oplus B = \overline{A} \cdot B + A \cdot \overline{B}$
2. use De Morgan's laws to get rid of long NOT bars until the bars only stretch one variable. If there are multiple bars on top of each other, start with the highest one. Applying these laws, one must always keep the original order of the operations in tact. Introduce brackets when this is needed!
3. use the distributive property ($A \cdot (B + C) = A \cdot B + A \cdot C$) to remove all brackets. The result will be a sum of products.
4. now find terms with a common factor. Introduce brackets again by applying the distributive property in reverse. This is the hardest step: depending on which terms you take together the result between the brackets will be simple or complicated. In the example above, this resulted in $(C + \overline{C})$, which can be replaced by 1.
5. at the end one can use the last 2 rows of the property-table to simplify even further

4.4 Convert to NAND gates

Imagine you have to design a circuit for $X = \overline{\overline{A \cdot B} \oplus C} + D \cdot \overline{E}$, but you only have NAND gates. Is this possible?

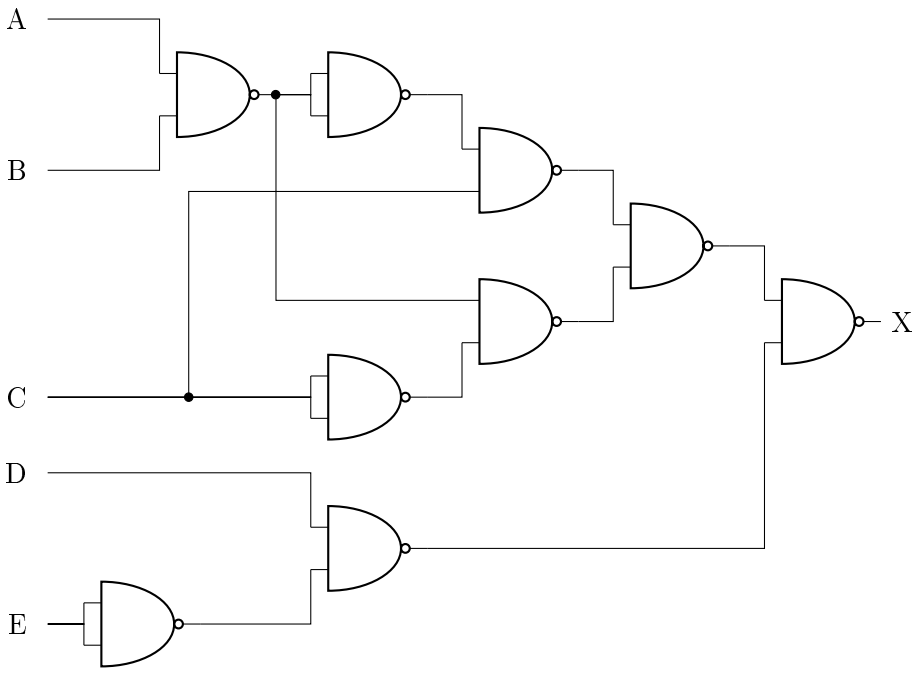
So the question is: can you convert any Boolean expression so it only contains elements of the form $\overline{A \cdot B}$? The answer is: yes!

The idea is as follows: use De Morgan to convert any OR operation to AND. This can only be done if there is a NOT over the OR operation. If there is none, one can introduce 2 NOT gates on top of each other. The lowest one can then be split.

Here is an example:

$$
\begin{array}{rcll}
X & = & \overline{\overline{A \cdot B} \oplus C} + D \cdot \overline{E} & \\
 & = & \left((A \cdot B \cdot C) + (\overline{A \cdot B} \cdot \overline{C}) \right) + (D \cdot \overline{E}) & \text{get rid of } \oplus \\
 & = & \left(\overline{A \cdot B \cdot C} \cdot \overline{\overline{A \cdot B} \cdot \overline{C}} \right) + (D \cdot \overline{E}) & \text{De Morgan} \\
 & = & \overline{\left(\overline{A \cdot B \cdot C} \cdot \overline{\overline{A \cdot B} \cdot \overline{C}} \right) + (D \cdot \overline{E})} & \text{double negation} \\
 & = & \overline{\left(\overline{A \cdot B \cdot C} \cdot \overline{\overline{A \cdot B} \cdot \overline{C}} \right)} \cdot \overline{(D \cdot \overline{E})} & \text{De Morgan} \\
 & = & \overline{\left(\overline{\overline{A \cdot B} \cdot C} \cdot \overline{\overline{A \cdot B} \cdot \overline{C}} \right)} \cdot \overline{(D \cdot \overline{E})} & \text{double negation}
\end{array}
$$

The end result looks very complicated but it only contains NAND and NOT gates. A NOT gate can also be made from a NAND gate. This is done by connecting both inputs to each other. This means that the above result can be drawn as:



This is very important: one can build any logic circuit using only NAND gates! Implementing a NAND gate in any technology enables to create a complete computer.

Similarly one can prove that you can also create any logic circuit using only NOR gates.

4.5 Converting truth tables

In many cases the most easy way to describe the behaviour of a logic circuit is to write down a truth table. It would therefore be interesting to be able to convert a truth table into a Boolean expression. There are a number of ways to do that of which we discuss two.

4.5.1 The conjunctive normal form

The simplest way to convert a truth table to a Boolean expression is to write it down in the 'conjunctive normal form'. This is a formula that contains a sum of products and that describes when the output is equal to 1. In mathematics the word 'conjunction' is also used for an AND and 'disjunction' is used for an OR.

An example should clarify this. We start from the following truth table:

A	B	C	X
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

In words, the output is 1 when (A is zero AND B is zero AND C is zero) OR (A is zero AND B is one AND C is one) OR (A is one AND B is zero AND C is one) OR (A is one AND B is one AND C is one). This can be written down in a Boolean expression as follows:

$$X = \overline{A} \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot B \cdot C + A \cdot \overline{B} \cdot \overline{C} + A \cdot \overline{B} \cdot C + A \cdot B \cdot C$$

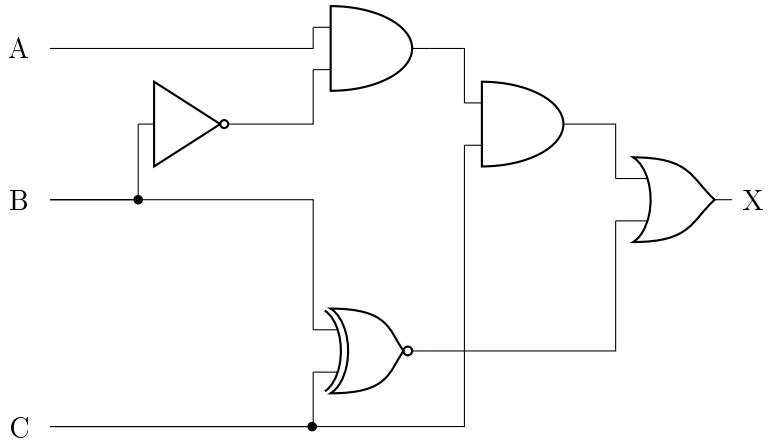
So there is a term for each 1 in the truth table.

This formula is called the (conjunctive) 'normal form'.

Of course this is not the most optimal way to make a circuit for the truth table. On solution is to use Boolean algebra to simplify it. In this example one might do the following:

$$\begin{aligned} X &= \overline{A} \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot B \cdot C + A \cdot \overline{B} \cdot \overline{C} + A \cdot \overline{B} \cdot C + A \cdot B \cdot C \\ &= \overline{B} \cdot \overline{C} \cdot (\overline{A} + A) + B \cdot C \cdot (\overline{A} + A) + A \cdot \overline{B} \cdot C \\ &= \overline{B} \cdot \overline{C} + B \cdot C + A \cdot \overline{B} \cdot C \\ &= \overline{B \oplus C} + A \cdot \overline{B} \cdot C \end{aligned}$$

The resulting circuit becomes:



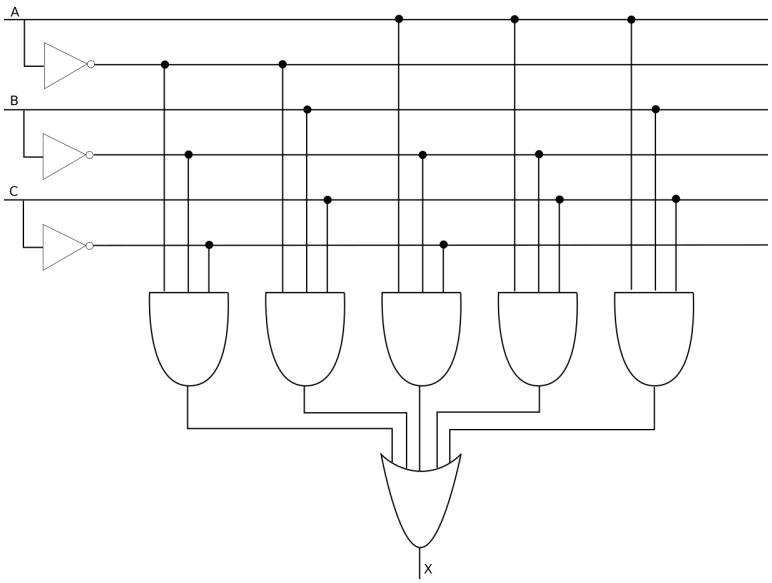
Notice that the last step in the derivation is often skipped. This is because XOR gates are not very commonly used and are mostly implemented using NAND or NOR gates in practice.

4.5.2 Programmable logic arrays (PLA)

In stead of simplifying the normal form one can also draw the circuit as is. In the previous example the following formula was found:

$$X = \overline{A} \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot B \cdot C + A \cdot \overline{B} \cdot \overline{C} + A \cdot \overline{B} \cdot C + A \cdot B \cdot C$$

This can be directly drawn as follows:



This is an interesting design because it can also be used to represent other Boolean expressions as long as they are in the normal form. The only thing that changes is the connections in the upper part of the circuit.

This was used in the seventies to make 'programmable' circuits. The above circuit was made as an integrated circuit but all connections are made. Then, using a laser, the chip is programmed by destroying the unnecessary connections. It allowed to make more general purpose integrated circuits which greatly reduced the cost of the chips. These chips were called 'Programmable Logic Arrays' (PLA).

The resulting circuit is not the simplest but is very fast.

These days other techniques are used to create programmable hardware such as FPGA (Field Programmable Gate Array) and CPLD (Complex Programmable Logic Device). One can create a complete functional microcontroller using one of those chips.

4.5.3 Karnaugh maps

The normal form is not the most economic way to create a new circuit (if you count the number of gates needed). Of course, one can reduce the number of gates by simplifying the Boolean expression, but there is a technique that allows to write down the most optimal solution in one step. This technique uses 'Karnaugh maps'.

A Karnaugh map is in fact a truth table that is drawn in a different way so one can easily see the most optimal expression by grouping the ones. In order to understand this, let's take a look at a normal truth table which consists of multiple columns. The information is only contained in the last column (the output). If one would only get the values of the last column, one can reconstruct the whole truth table because that only depends on the number of values in the output column. This means that the position of each output determines exactly to which input it corresponds.

In a Karnaugh map the last column (output) of a truth table is written as a 2-dimensional structure. The position of each output also determines the input it corresponds to. Because it is 2-dimensional one can see the structures that can be found in the output more easily.

There are different Karnaugh maps for 2, 3 and 4 inputs. If one wants to use more than 4 inputs, other techniques are more appropriate (like Quine-McCluskey).

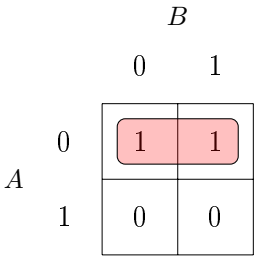
In order to clarify the technique we shall present a few examples.

4.5.3.1 Two inputs

The first example has two inputs and has the following truth table:

A	B	X
0	0	1
0	1	1
1	0	0
1	1	0

The Karnaugh map is drawn as follows:



The input A is on the left side and the input B is on the top. They can both be 0 or 1 resulting in 4 squares in which the corresponding output is written. So the upper left corner corresponds to input 00, the upper right to 01, the lower left to 10 and the lower right to 11.

The next step is to find groups of 8, 4, 2 or 1 cells containing only ones. The groups must be as big as possible and are allowed to overlap. In this case, there is a group of 2 ones on the top row (indicated on the map).

The final step is now to write down a formula for each group and OR them together. One does this by looking at all the inputs that have the same value for all the elements in the group.

In our example one can see that $A = 0$ for all elements in the group. Therefore the formula for that group is $X = \overline{A}$. Since there are no other groups, this is the final formula.

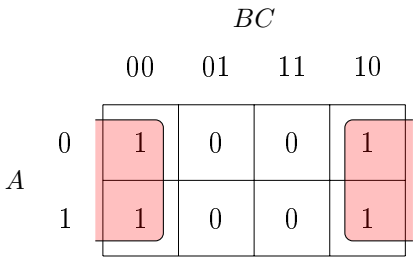
4.5.3.2 Three inputs

Of course, for only 2 inputs, one could have also found the formula directly from the truth table. Karnaugh maps are more interesting if there are 3 or 4 inputs.

We shall illustrate a Karnaugh map with 3 inputs starting from the following truth table:

A	B	C	X
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

The Karnaugh map for 3 inputs is drawn as follows:



Remark that the top row contains all possibilities for the inputs B and C. These possibilities are not numbered sequentially. They are numbered using the 'Gray code' scheme. The idea is that only one bit is allowed to change going from one value to the next. So in this case this results in 00, 01, 11, 10. Remark that there is also only one bit of difference between the last and the first value. Therefore the cells on the right are considered neighbours of the cells on the left. And the cells on top are considered neighbours of the ones on the bottom.

This is an important feature. In the above Karnaugh diagram one can identify 2 groups of 2. But since the right cells are neighbours of the left cells, there is in fact one group of 4! One can easily see that C is zero for all four cells, thus the formula becomes: $X = \overline{C}$.

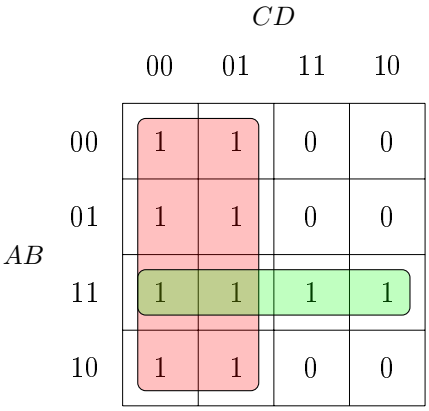
If we would have taken the 2 groups separately, the formula $X = \overline{B} \cdot \overline{C} + B \cdot \overline{C}$ would have been found. This is much more complicated and it shows that identifying large groups is very important to find the most optimal solution.

4.5.3.3 Four inputs

For four inputs we start with an example with the following truth table:

A	B	C	D	X
0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

This is put in a Karnaugh map as follows:



Now Gray codes are used for both the columns and the rows.

In this map we can identify 2 overlapping groups (as indicated): one with eight cells and one with four cells. The group of 8 can be written as $X = \overline{C}$ and the group of 4 can be written as $A \cdot B$ resulting in the formula: $X = A \cdot B + \overline{C}$.

Remark that it is important to know that groups are allowed to overlap. If not, one has to take the 2 cells next to the first group together. But then the end result becomes: $X = A \cdot B \cdot C + \overline{C}$ which is more complicated than the previous formula.

4.5.4 Don't care states

Sometimes one needs to design a circuit where the output is not specified for certain input combinations. These outputs are called 'don't care'. It allows to make the circuit more simple and Karnaugh maps can be of great help designing these circuits.

As an example we shall create a circuit for a 'BCD decoder'. In 1.2.2 we discussed how decimal numbers can be stored as binary numbers in memory. But sometimes another technique is used. One could also encode every decimal digit into 4 bits and store it as a nibble. For instance the number 2568 could be stored as 0010 0101 0100 1000. As every nibble contains one decimal digit, only values 0 till 9 are allowed. Values 10-15 are not allowed to be stored. This is called 'BCD' (Binary Coded Decimal).

Now we want to design a circuit with 10 outputs. It takes one nibble as an input and shows the value by setting the corresponding output to 1. So all outputs will be zero except for the one that is selected by the input. As there are 10 outputs we need to create 10 circuits. Every circuit will work in parallel with the others and will decide if its output should be zero or one.

When the input of this circuit is a number between 10 and 15, we don't care about the output. That is because this should also never happen. These are called 'don't care states'.

The truth table for all 10 circuits looks like this:

A	B	C	D	0	1	2	3	4	5	6	7	8	9
0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	0	0	0	0	0	0
0	0	1	1	0	0	0	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0	1	0	0	0	0	0
0	1	0	1	0	0	0	0	0	1	0	0	0	0
0	1	1	0	0	0	0	0	0	0	1	0	0	0
0	1	1	1	0	0	0	0	0	0	0	1	0	0
1	0	0	0	0	0	0	0	0	0	0	0	1	0
1	0	0	1	0	0	0	0	0	0	0	0	0	1
1	0	1	0	X	X	X	X	X	X	X	X	X	X
1	0	1	1	X	X	X	X	X	X	X	X	X	X
1	1	0	0	X	X	X	X	X	X	X	X	X	X
1	1	0	1	X	X	X	X	X	X	X	X	X	X
1	1	1	0	X	X	X	X	X	X	X	X	X	X
1	1	1	1	X	X	X	X	X	X	X	X	X	X

An X is used when the output doesn't matter: it can be zero or one.

Let's look at the circuit for output X_8. The Karnaugh map looks like this:

AB \ CD	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	X	X	X	X
10	1	0	X	X

Now we can choose the values of the X's in order to create large groups of cells. On a Karnaugh map this is very visible. In the above map one group of four can be created (as indicated). The resulting formula is: $X_8 = A \cdot \overline{D}$.

When we would have set all X-values to 0, the resulting formula would have been $X_8 = A \cdot \overline{B} \cdot \overline{C} \cdot \overline{D}$ which is much more complicated!

4.6 Exercises

4.6.1 Converting formulas, circuits, and truth tables

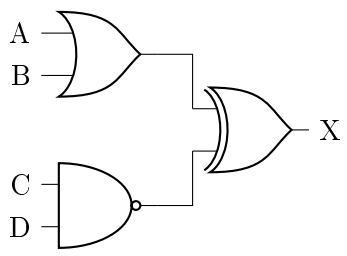
Draw the circuit and the truth table for the following Boolean expressions:

1. $X = \overline{A + B}$
2. $X = \overline{A \oplus B}$
3. $X = A \cdot (B + C)$
4. $X = A \cdot B + C$
5. $X = A \cdot \overline{B} \oplus C$
6. $X = \left(A + \overline{B}\right) \cdot \left(C \oplus \overline{D}\right)$
7. $X = \overline{(\overline{A} + B) \cdot \overline{C}}$

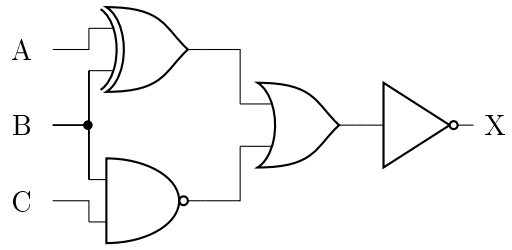
4.6.2 From circuit to formula and truth table

Give the Boolean expression and the truth table for the following circuits:

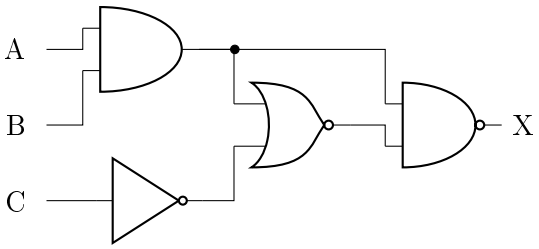
a)



b)



c)



4.6.3 Simplifying Boolean expressions

Simplify the following Boolean expressions. Use the properties described in 4.3 and De Morgan's laws.

1. $X = \overline{A} \cdot B + A \cdot \overline{B} + \overline{A + B}$
2. $X = \overline{\overline{A} + B} + A \cdot \overline{C} + \overline{\overline{B} + \overline{A + \overline{B}}} + \overline{C + A}$
3. $X = \overline{A \cdot \overline{\left(\overline{A} + C\right)}}$
4. $X = \overline{A \cdot B \cdot C + \overline{A \cdot B} \cdot C}$
5. $X = \overline{A \cdot (B + C) + \overline{B} \cdot \overline{C}}$
6. $X = \overline{\left(A + \overline{B} \cdot C\right)} \cdot \left(A \cdot \overline{B \cdot \overline{C}}\right)$
7. $X = \overline{D} \cdot \overline{(A + B) + C} + \overline{A} \cdot \overline{B + C}$
8. $X = \left(A \cdot \overline{B} + A \cdot B + \overline{A} \cdot B\right) \cdot \left(A + \overline{A} \cdot B\right)$
9. $X = (A \cdot \overline{B} \cdot C) \cdot \overline{\left(\overline{A \cdot C} + \overline{D \cdot A}\right)} + \left(E \cdot \overline{A} + F\right) \cdot \overline{G}$
10. $X = A \cdot B \cdot (\overline{C + \overline{D \cdot A}}) \cdot \overline{\left(\overline{D \cdot A} \cdot (\overline{A \cdot B} + C)\right)}$
11. $X = \overline{\left(\overline{P + Q \cdot R}\right) \cdot T + \overline{P} + \overline{Q} + \overline{T}}$

4.6.4 Convert to NAND and NOR gates

Verify you can create all basic gates (AND, OR, XOR, NAND, NOR) using only NAND gates. Draw the circuit for every case.

Now do the same with only NOR gates.

How can one make an AND gate with 3 inputs using only NAND/NOR gates?

4.6.5 Convert to NAND and NOR gates

Convert the following circuits using only NAND gates and draw the resulting circuit:

1. $X = A + B \cdot C$
2. $X = \overline{A \cdot B} + C + \overline{D}$
3. $X = \overline{A \cdot B} \cdot \left(M + \overline{K} + L\right)$
4. $X = \overline{A} \cdot \overline{\overline{C} + D} + C \cdot \overline{B}$
5. $X = \overline{D} \cdot \overline{\overline{A \cdot B} + C} + \overline{A} \cdot \overline{B + C}$
6. $X = \overline{A \cdot \overline{B + C} + \overline{\overline{E} + \overline{B} + C}}$

Convert the following circuits using only NOR gates and draw the resulting circuit:

1. $X = A + B \cdot C$
2. $X = \left(\overline{F} + C\right) \cdot \overline{A} \cdot \left(D + \overline{B}\right)$
3. $X = \left(N + \overline{\overline{K} \cdot L}\right) \cdot \overline{B \cdot \overline{A}}$

4.6.6 The normal form

Determine the normal form for the following truth tables:

a)

A	B	C	X
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

b)

A	B	C	X
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

c)

A	B	C	X
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

d)

A	B	C	D	X
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

4.6.7 PLA

Convert the normal forms from exercise 4.6.6 to PLA representation.

4.6.8 Karnaugh maps

Draw the Karnaugh map and write the optimal formula for X for the following truth tables:

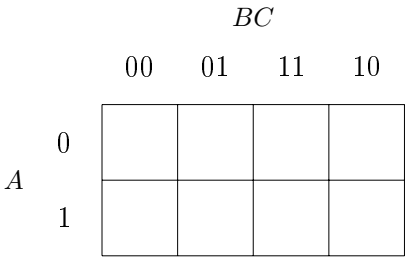
a)

A	B	C	X
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

		BC			
		00	01	11	10
A	0				
	1				

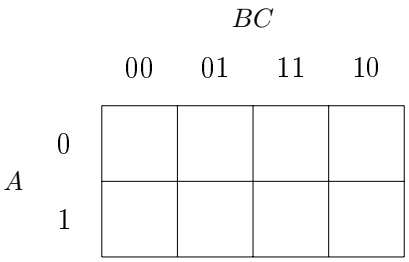
b)

A	B	C	X
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1



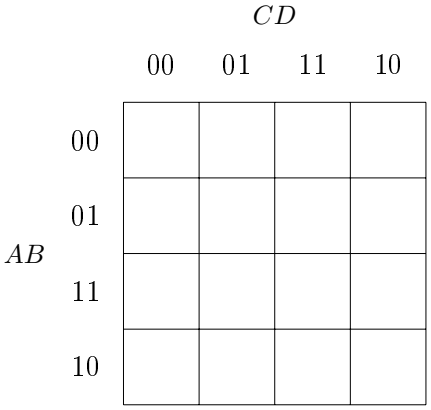
c)

A	B	C	X
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1



d)

A	B	C	D	X
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0



4.6.9 Karnaugh maps

Determine the most optimal formula from the following Karnaugh maps:

1.

A \ BC	00	01	11	10
0	1	1	1	0
1	0	1	1	1

2.

A \ BC	00	01	11	10
0	1	0	1	1
1	1	0	0	1

3.

A \ BC	00	01	11	10
0	1	0	1	1
1	1	1	1	0

4.

AB \ CD	00	01	11	10
00	1	0	1	0
01	1	0	1	0
11	1	0	1	0
10	1	0	1	0

5.

AB \ CD	00	01	11	10
00	1	0	0	1
01	0	1	1	0
11	0	1	1	0
10	1	0	0	1

6.

AB \ CD	00	01	11	10
00	1	0	1	1
01	0	0	1	0
11	1	0	1	0
10	0	0	0	0

7.

AB \ CD	00	01	11	10
00	0	0	1	0
01	0	1	1	1
11	0	0	1	1
10	0	0	1	0

8.

AB \ CD	00	01	11	10
00	0	1	1	1
01	0	1	1	1
11	0	1	1	1
10	0	1	1	1

9.

AB \ CD	00	01	11	10
00	1	0	0	0
01	1	0	0	1
11	1	0	0	1
10	1	0	0	0

10.

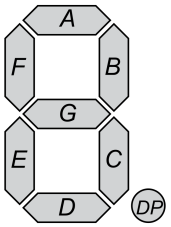
AB \ CD	00	01	11	10
00	1	0	0	1
01	0	0	0	1
11	0	0	0	1
10	1	0	0	1

4.6.10 Don't care states

In the discussion above only X_8 was determined. Now find the circuits for X_0 till X_9, using the same technique. Some of these will not be very different from the normal form, but others will be considerably simpler.

4.6.11 Display driver

Create a circuit with 4 inputs and 7 outputs. The 4 inputs determine a number from 0-9. The outputs are connected to a seven segment display:



Every output will indicate if its corresponding segment should be lit or not. Use Karnaugh maps to determine the circuit for every output.

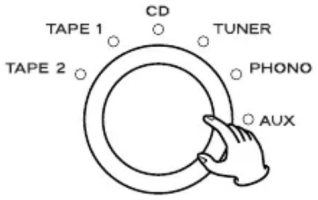
Chapter 5

Common logic circuits

In this part some very important logic circuits are introduced. In the previous discussion and exercises a BCD decoder and a display driver were already introduced. The next sections will describe other common circuits.

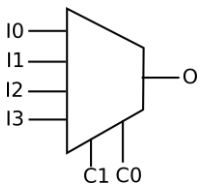
5.1 Multiplexer (MUX)

A 'multiplexer' (also named 'MUX' or 'data selector') is a circuit that allows to select a certain signal and route it to an output. One can compare this to the selector knob on an amplifier with which one can choose the music device one wants to listen to:

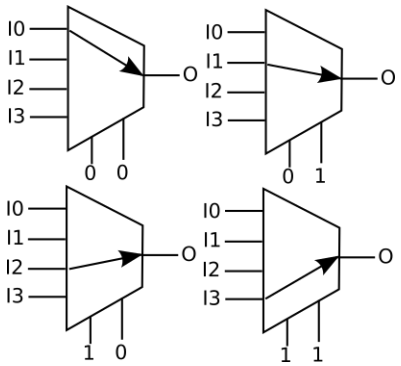


A MUX has a number of data inputs and one output. One can choose which input is routed to the output. In a MUX the input is not chosen by turning a physical knob, but rather by a binary number that is fed to the MUX through extra inputs (called 'control inputs'). A MUX with n control inputs has 2^n data inputs. The number of data inputs is therefore always a power of 2.

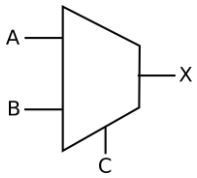
Schematically a MUX is represented as:



In this case there are 4 data inputs (I0, I1, I2, and I3). The inputs C0 and C1 are the control inputs that determine which input is selected. The output will be connected to the selected input. The following image shows all possible values of the control inputs and to which input the output is connected:



There are different MUX's possible with a different amount of inputs and control inputs. The most simple MUX only has one control input and two data-inputs. Its symbol would be:



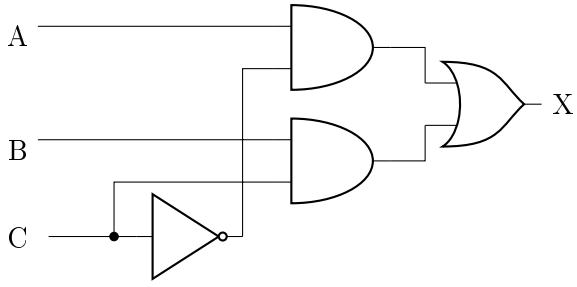
If $C = 0$ then $X = A$. If $C = 1$ then $X = B$. The truth table will thus become:

A	B	C	X
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

This can be converted into a Karnaugh map:

A \ BC	00	01	11	10
0	0	0	1	0
1	1	0	1	1

This results in the formula $X = A \cdot \overline{C} + B \cdot C$ which can be plotted as a circuit:



Remark that one can also intuitively understand the above circuit. The AND gates are used as switches. An AND gate can only output a one if both inputs are one. This means that if an input is zero, the AND gate cannot output a one. The AND gate is 'closed'. In the above circuit one input of each AND gate is connected to an input. The input can only travel through the AND gate if the other input is one. That is controlled by the control input.

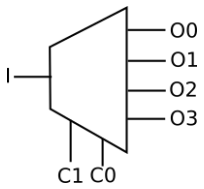
One can easily extend this circuit to create a MUX with more inputs. If one wants to create a MUX with 2 control inputs, one needs to use 4 AND gates. Each AND gate will be connected to a data input. The AND gates will need to have 3 inputs in this case. The two extra inputs are used to determine if the AND gate needs to be 'open' or 'closed'. This is accomplished by routing all possibilities of the two control inputs to the AND gates. The following section will introduce another circuit that illustrates that same idea.

MUX's are used a lot in computers. One application is to select a bit from memory. The inputs of a MUX are connected to all bits in memory. The control inputs will select a certain bit from memory and route it to the output of the MUX. In this case, a very large MUX is needed with billions of inputs and typically 32 or 64 control inputs! There are some tricks to use smaller MUX's but that is out of scope for this document.

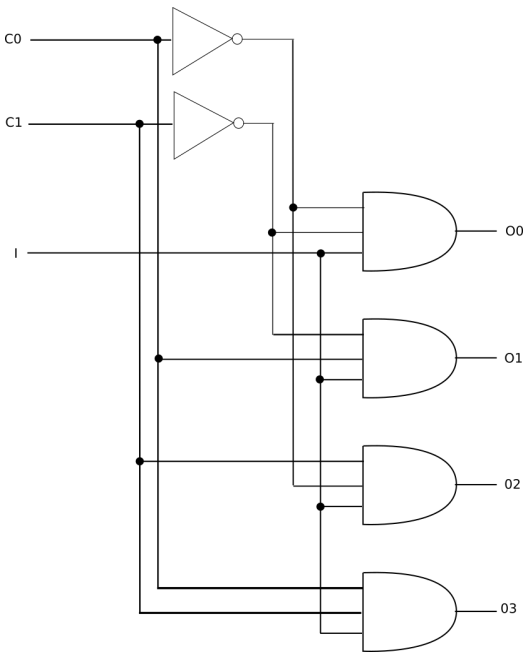
5.2 Demultiplexer (DEMUX)

A 'demultiplexer' (also called 'DEMUX' or 'decoder') does the inverse of a MUX. A DEMUX has one input that can be routed to one of its outputs. The output is determined by the control inputs.

For instance, a DEMUX with 2 control inputs can be represented by the following symbol:



The circuit of this DEMUX is shown here:



The AND gates are used as switches again. In this case, an AND gate can only be 'open' when a specific combination of bits is presented to the control inputs (00, 01, 10, or 11). In order to do this, each control input is also inverted.

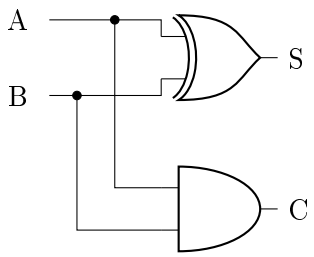
The top AND gate will only be open if both control inputs are 0. Indeed, when this is the case, the inverted values will be 1 and will be routed to the top AND gate. That gate will route the input to its output. The other AND gates will have at least one 0 in their inputs and are therefore 'closed'. The reader can verify the workings of the circuit by going through all possible values of the control inputs.

5.3 Half adder

A 'half adder' is a logical circuit that is able to add 2 bits. It has 2 inputs (one for each bit) and 2 outputs (the sum can go up to $[1 + 1 = 10]_2$). The truth table is as follows:

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Looking at the outputs one can easily recognise two logical gates. The output C is only 1 if both inputs are one and the output S is one if both inputs are not equal. This is an AND gate and an XOR gate. This means that: $S = A \oplus B$ and $C = A \cdot B$. The circuit therefore becomes:



5.4 Full adder

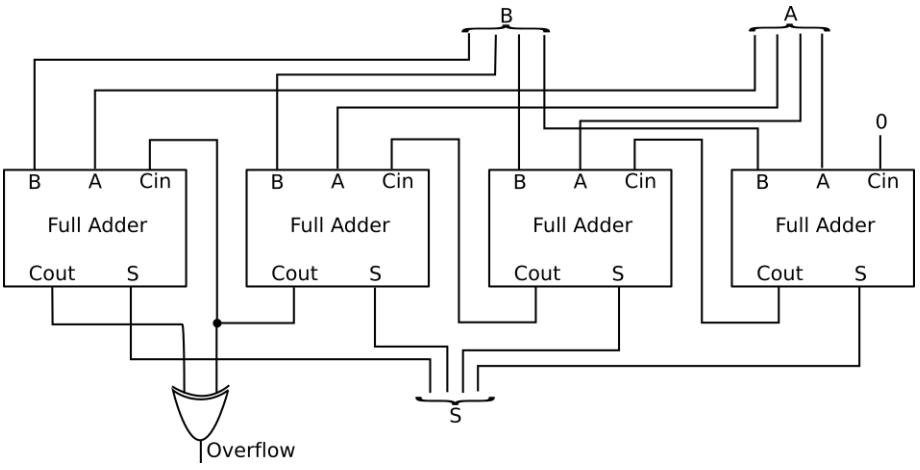
The previous circuit can be used to add the least significant bits of two numbers. But in order to add the next bits, one needs to take into account the carry bit of the previous calculation (the output C). This means we need a circuit that can add three bits: 2 bits and a carry. Using a circuit like that, we can create a circuit that adds two binary numbers by putting them is a cascade.

A circuit that can add 2 bits and a carry is called a 'full adder'. It has three inputs and two outputs. Its truth table looks like:

A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

There are many ways to create a circuit for a full adder: one can make Karnaugh maps, use the normal form, or combine two half adders. It is left as an exercise to the reader to find a circuit for the full adder.

Full adders can be put in a 'cascade' to add number with any number of bits. As an example one can add two 4-bit numbers as follows:



Four full adders are put in a cascade. The right most full adder adds the 2 least significant bits of A and B. Its carry-in is hard coded to 0. One could also use a half adder but being able to add 1 to the result by just altering that first carry-in has a lot of advantages (see e.g. exercise 5.6.5 at page 81).

The carry-out of the first full adder is fed into the carry-in of the next full adder. In this way the carry bits 'ripple through' the circuit until the last full adder uses it to do its addition. This circuit is therefore called a 'ripple-carry adder'. There are other ways to implement an addition but this is beyond the scope of this document. The fact that the carry needs to ripple through induces a time delay before the final result S stabilises. The electric signals need time to travel through the circuit. As such, this time puts a constraint on the maximal clock speed a processor can achieve.

The sum of the addition is shown on the bottom (S). In addition to this result other information can be extracted from this calculation. This is shown with the XOR gate on the bottom. It compares that 2 most significant carry bits. If they are different, the 'overflow' signal is put to one. So the 'overflow' signal indicates if there was a 2-complement overflow while adding A and B. Other signals could be added to this circuit, giving information about the calculation. Examples include: is the result negative? is the result equal to zero? These signals are called 'flags' and will be used later to construct a processor.

The above circuit (excluding the overflow flag) was implemented as an integrated circuit with number 74283.

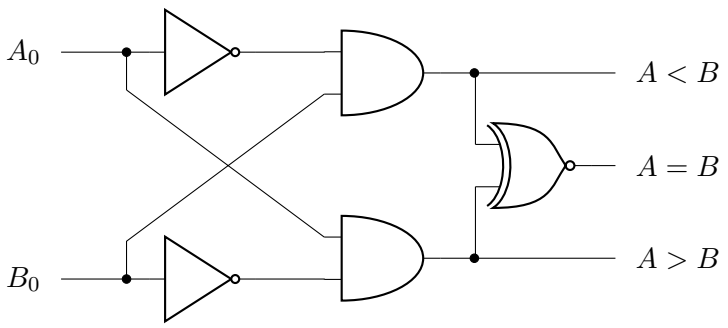
5.5 Comparator

A 'comparator' is a circuit that can compare 2 binary numbers. Its output indicates if one number is equal, greater than, or smaller than the other number. The circuit therefore has $2n$ inputs and 3 outputs.

The most simple comparator is one that compares 2 numbers of 1 bit. The truth table is as follows:

A_0	B_0	A<B	A=B	A>B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

The circuit can be drawn as follows:

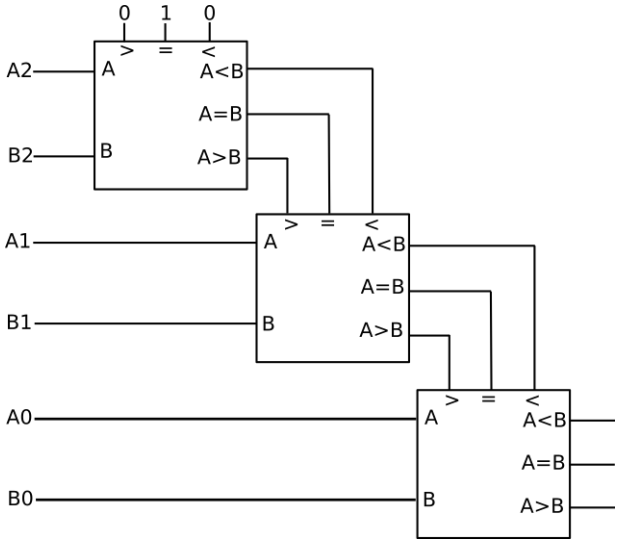


Remark that the output A=B is not calculated from the inputs. This is mostly done for aesthetic purposes (the lines do not cross), but it also illustrates a very important feature of the comparator: any output can be calculated from the two other outputs. This is because exactly one output is 1 at any time. This means the XNOR gate could also be replaced by a NOR gate which will output a 1 when the two other outputs are 0.

One can also design a comparator that compares numbers with more bits. The number of inputs doubles each time one adds a bit. Therefore, the truth table can become very large. Another approach to build a comparator for larger numbers is to create a cascade of 1-bit comparators (analogous to the full adder).

Let's see how this could work. When comparing binary numbers, one starts with the most significant bit. If these are different, one can immediately see that one number if greater than the other. If they are equal, one needs to compare the next bits. If one reaches the least significant bit, the numbers are equal.

So in order to create a cascade of 1-bit comparators, the first comparator will compare the 2 most significant bits. The output of this comparator is given to the next, together with the next 2 bits to compare. This is shown for two 3-bit numbers in the following diagram:



Each block is a comparator that compares 2 bits, but also takes into account the information of the previous comparison. There are now 5 inputs per comparator. But one can notice that one only needs 4 inputs because the fifth is determined by the other 4 (e.g. one can predict the = on the basis of < and >).

The truth table is as follows (here the = input is not used):

A	B	A<B	A>B	A<B	A=B	A>B
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	1	0	0
0	0	1	1	X	X	X
0	1	0	0	1	0	0
0	1	0	1	0	0	1
0	1	1	0	1	0	0
0	1	1	1	X	X	X
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	0	0
1	0	1	1	X	X	X
1	1	0	0	0	1	0
1	1	0	1	0	0	1
1	1	1	0	1	0	0
1	1	1	1	X	X	X

Notice that there are don't care states because the A<B and A>B input signals cannot be 1 at the same time. One can create a circuit from this table using Karnaugh maps. This is left as an exercise.

5.6 Exercises

5.6.1 MUX

1. Draw the circuit of a MUX with 4 inputs. Use 4 AND gates, each with 3 inputs. How many control inputs are there?

2. Simulate this circuit in Logisim or Digital.

3. One can also create a larger MUX using smaller ones. Try to make the previous MUX, but this time assemble it using 3 small MUX's with 2 inputs.

5.6.2 DEMUX

1. Draw the circuit of a DEMUX with 2 outputs and 1 control signal (the minimal DEMUX possible)

2. Simulate this circuit in Logisim or Digital.

3. Build a DEMUX with 8 outputs (3 control signals) using only DEMUX's with 2 outputs. You will need 7 of them.

5.6.3 Full adder

1. Determine the formulas and the circuit for a full adder and simulate it. There are different ways to do this: Karnaugh maps, PLA, or a cascade of 2 half adders.

2. Simulate a cascade of full adders like in 5.4 and verify its workings.

5.6.4 Comparator

Draw the circuit of the 1 bit comparator that can be cascaded. You can use the truth table from above. Simulate the circuit.

Build a cascade of those comparators and see if you can compare 4 bit numbers with this.

5.6.5 ALU

Try to find a circuit that can subtract 2 numbers. Hint: you have to add the 2-complement of the second number to the first. This means that you have to take the inverse of the second number and add one that. Adding one can be accomplished by putting the first carry-in bit to one!

This illustrates again why 2-complement is so interesting. With a small modification one can turn an adder into a subtractor.

Now add one more input to the circuit. If the input is 0 the circuit should calculate $A + B$. If the input is 1 the circuit should calculate $A - B$. You can use MUX's to choose values.

The resulting circuit is a simple version of an 'ALU' (Arithmetic Logic Unit). A real ALU can do multiple mathematical and logic calculations with its input numbers. Which calculation is determined by extra inputs. Such an ALU will be introduced in chapter 9 on page 98.

Chapter 6

Sequential circuits

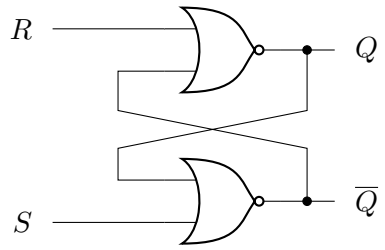
Until now the output of a circuit is determined directly by its input. Data flows from the input towards the output. These circuits are also called 'combinational circuits'.

But what if (some of) the output were rerouted to the input? In many cases, this will result in erratic behaviour because the output will change the input which will in turn change the output. So it might seem like a weird idea.

But in some cases the output of such a circuit can be stable. These circuits can be used as memory since they can retain a certain value until one of the inputs is changed. These kind of circuits are also called 'sequential circuits'. Some of the most interesting ones are discussed in this chapter.

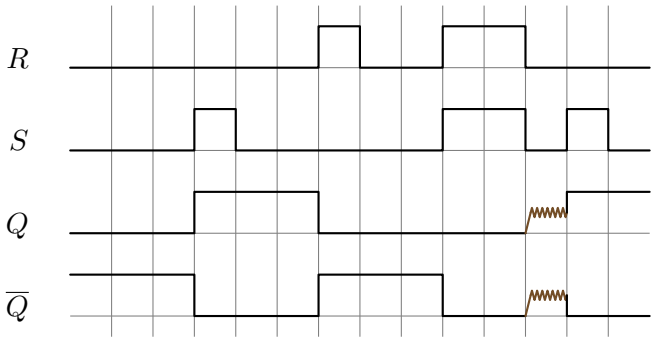
6.1 The RS-latch

A very simple sequential circuit is built like this:



In order to determine the behaviour of this circuit one could try to make a truth table. But this does not describe the behaviour very well since a truth table assumes that the input determines the output. In this case however, the output is fed back causing it to change again, until it (hopefully) stabilises. Therefore, a kind of graph is used to describe how the output evolves as a function of the input: a 'timing diagram'. In this diagram all input and output signals are shown as a function of time. The input signals are given as example values over time and the output shows the state to which the output evolves. Only the stable version of the output is shown. So if an output alternates between 1 and 0 a few times before settling at 0, only the last 0 is shown. If the output is unstable, a zigzag line is drawn.

For the RS-latch a timing diagram could be as follows:

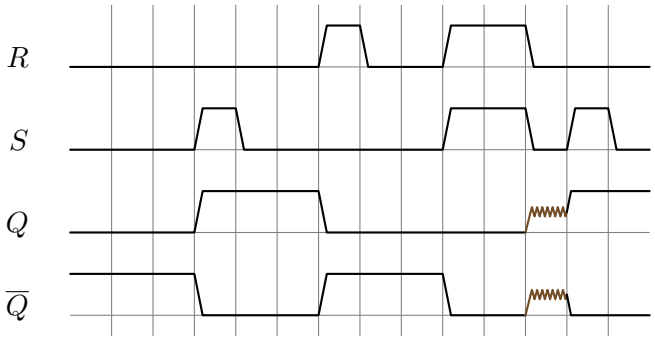


The top 2 rows show how the input is changed over time: R and S start at value 0, then S is put to 1 and back to 0. After this R is put to 1 and back to 0. At the end both R and S are put to 1 and released at the same time.

Let's now see what what happens to the output. The output is already shown on the previous diagram, but it can be found by following the signals in the circuit. There are 2 outputs Q and $\overline{Q}$. As the name suggests, $\overline{Q}$ is supposed to be the inverse of Q. This is almost always the case, except when R and S are both equal to 1. In timing diagrams one always need to know the initial state of the outputs since they also function as an input.

Imagine the inputs R and S as switches. They stand for 'set' and 'reset'. Pressing the S button, will put Q to 1 (it will 'set' Q). Pressing the R button, will 'reset' Q (its value is set to 0). When trying to set and reset the circuit at the same time (S and R are put to 1), both outputs become 0. This is not a desirable input. Moreover when releasing both buttons at the same instant will result in an unknown state of the system. In theory the output will keep on alternating between 1 and 0, which is known as a 'race condition'. In practice however Q and $\overline{Q}$ will get a value but it is impossible to predict which one. This is why the timing diagram shows a scrambled line. Since the S button is pressed at the end, the circuit becomes stable again.

In this text it is assumed that the logic gates respond immediately and infinitely fast. Therefore the states of the inputs and outputs change instantly. Of course, this is not the case in reality. In order to indicate the fact that time is needed to change and propagate a signal, the vertical lines are sometimes drawn slanted. For the above timing diagram, this would result in:



Summarising, the following observations can be made:

- Q and $\overline{Q}$ always have opposite values, except when R and S are both equal to 1.
- As long as R and S are both 0, Q en $\overline{Q}$ retain their value. In this case the circuit 'remembers' it state. It is functioning as one bit of memory.
- When S becomes 1 Q also turns to 1 and $\overline{Q}$ becomes 0. Even if S returns to 0, Q remains at 1.

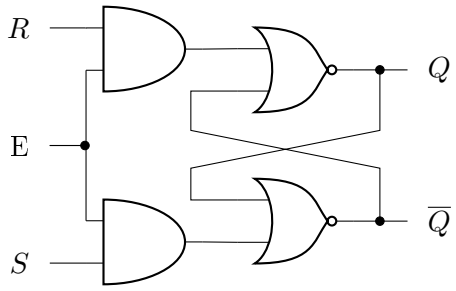
- Reversed: when R becomes 1, Q turns to 0. Even after R is put back to 0.
- When R and S are both equal to 1 both Q and $\overline{Q}$ turn to 0.

This circuit is called an 'RS-latch' and is used to store 1 bit of data. The value of the memory is given by the output Q.

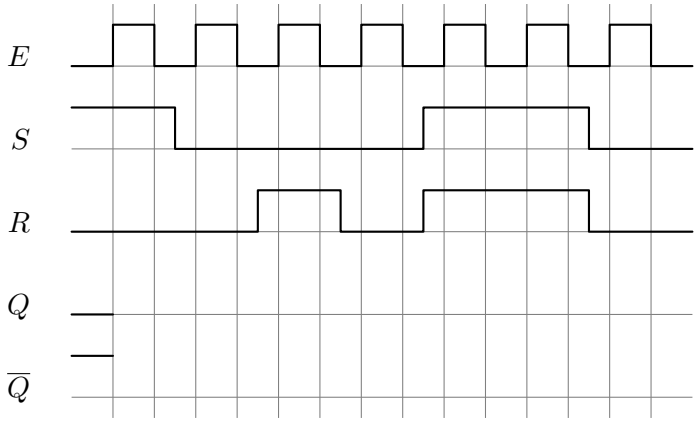
6.2 The synchronous RS-latch

In a large memory chip, billions of RS-latches are used to store information. If data needs to be stored at a specific location, that latch needs to be selected. A selected RS-latch will store the data while the non-selected RS-latches will ignore the data. In order to accomplish this another input signal is added to the latch. This signal is called 'enable'.

When the enable signal is equal to 0 it should block the input to the latch. If it is 1, the input should be presented to the latch. When the MUX was introduced, it became clear that an AND gate can be used as a switch. This can now also be used. Each input signal is routed through an AND gate. The second input of the AND gate is controlled by the enable signal. The circuit is shown here:



Try to complete the following timing diagram to understand its workings:

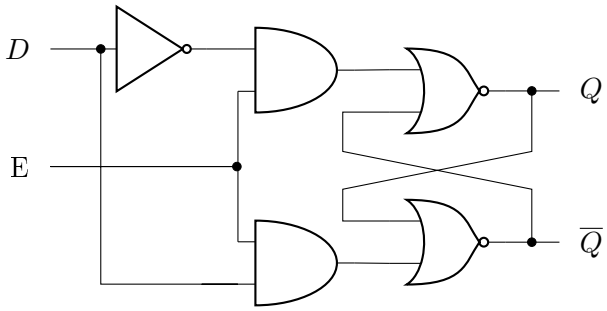


Remark that Q and $\overline{Q}$ can only change while (and as long as) $E = 1$. When $E = 0$ the output will remain at the value that it had when E dropped to 0. The enable signal will let the input through, effectively storing the bit at the input.

At the end of the previous timing diagram the race condition occurs again. This is because the enable signal goes to zero while both inputs are 1. This had the same effect as releasing both inputs in the RS-latch. When E becomes 1 again, the output is determined again.

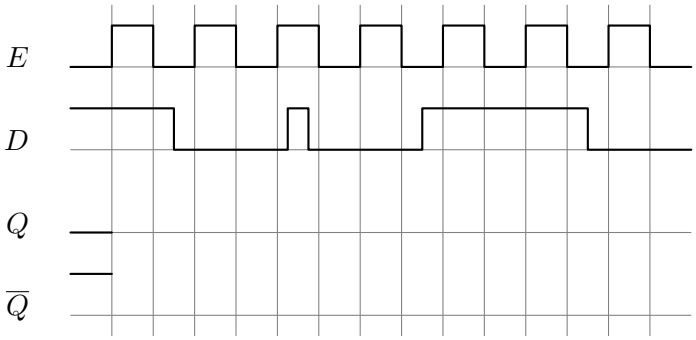
6.3 The synchronous D-latch

The synchronous RS-latch is very useful but it still has one drawback: race conditions can occur and are not wanted. There is a simple way to get rid of these though: by making sure that R and S are always the inverse of each other. This can be easily achieved by adding a NOT gate to the circuit as follows:

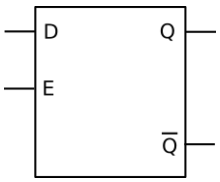


The input is now called 'D' (data) and presents a bit to be stored in the latch. The bit is only stored when E is equal to 1. While E is 1 the output will follow the value of the input D.

Completing the following time diagram is left to the reader:



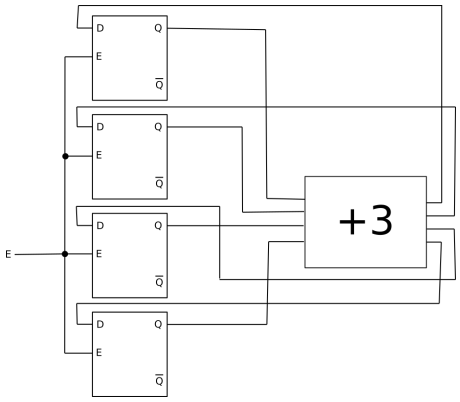
D-latches are used a lot. It is often drawn with the following symbol:



D-latches are used to create 'static RAM' (SRAM). More information can be found in 8.3 on page 94.

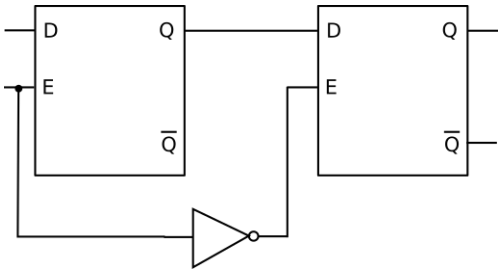
6.4 The master-slave D flip-flop

Imagine the following setup: we put 4 synchronous D-latches next to each other. Together they represent a 4-bit number that is stored within these latches. The output of the latches is fed into a circuit that will add 3 to its input. This adder circuit can be easily constructed with the methods described before. Now the newly created value needs to be stored in the same location it came from. In essence we want to create a circuit that repeatedly adds 3 to a number and stores the value again. This circuit could look like this:



Clearly this cannot work. When E becomes 1 the latches will read and store the new number, but the output will immediatly be sent to the adder, therefore changing the input of the latches again. This results in erratic behaviour. There is no clock to synchronise the circuit.

The biggest problem is that that state of the latches changes as long as E=1. We would like to change this so that the state can only change at a very specific moment in time. This can be accomplished by replacing the latches with a combination of two latches as follows:



This combination is known as a 'master-slave D flip-flop' or simply a 'D flip-flop'. The second latch (the 'slave') will see the inverse of E. This means that when the first latch (the 'master') is 'closed', the second latch is 'open' and vice versa.

Let's see what happens when a bit is presented at the D-input and E=0. In this case, the master will not store the new data. The slave will store and also output whatever value is stored in the left latch. But since the value of the master doesn't change, the output of the master also stays fixed. It shows the data that was previously stored in the master.

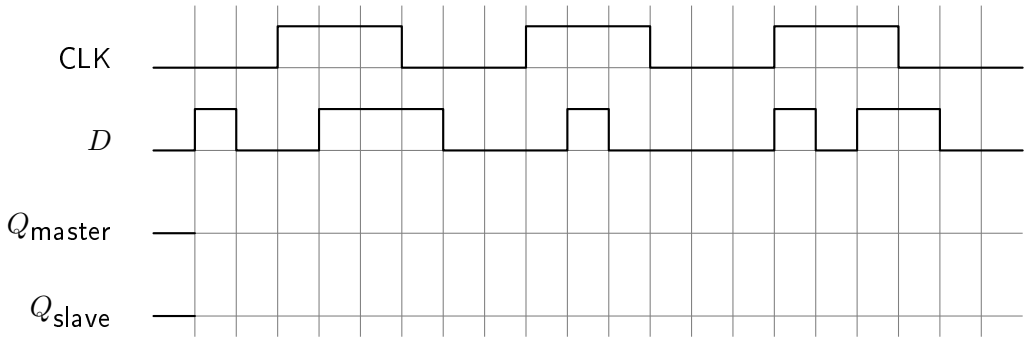
When E becomes one the master will read and store the new data. But this data will not flow to the slave as that is blocked now. So the output of the circuit is still the old value.

Now when E turns to zero, the slave will store the new value of D. At the same time the master closes so any changes to D will not be reflected any more. Now the output of the circuit shows the new value of D.

This means that data is stored at a very specific moment in time: when E goes from 1 to 0. This is called the 'falling edge'. The signal E is now called 'clock' as it can be used to synchronise a circuit. If one would replace the latches in the adder circuit with flip-flops, it would actually work. The new value would be stored every time the clock goes from one to zero.

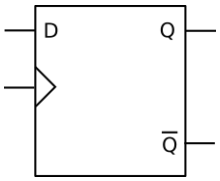
One can also construct a flip-flop that responds to the 'rising edge' of the clock. In this case the NOT gate is put on the first latch. In this text however, we will only use falling edge flip-flops.

Complete the following timing diagram to understand the inner workings better:



Remark that Qslave only changes on the falling edges of the clock.

A D flip-flop is used a lot in microprocessors and it is represented by this symbol:

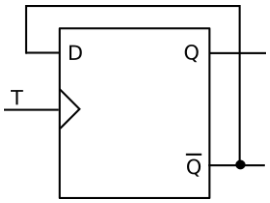


The clock (CLK) is indicated by a triangle.

By decoupling the input and the output we are now able to use the stored value in a flip-flop, do a calculation with it, and feed the result back. This will be used in chapter 7 on page 90.

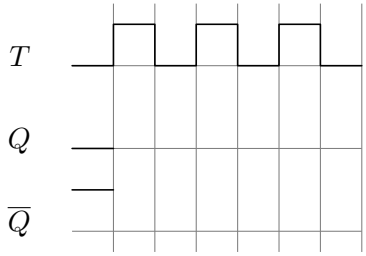
6.5 The T flip-flop

A T flip-flop demonstrates how a D flip-flop can be used. It is constructed as follows:



The output $\overline{Q}$ is fed back to the input of the flip-flop. This means that the flip-flop will store the inverse of the stored value, each time the clock goes from one to zero (falling edge). In other words, the value of the flip-flop toggles each time the clock has a falling edge. The T in T flip-flop stands for toggle.

A T flip-flop only has one input, which is the clock. Complete the following timing diagram and see what happens when the clock goes up and down repeatedly:

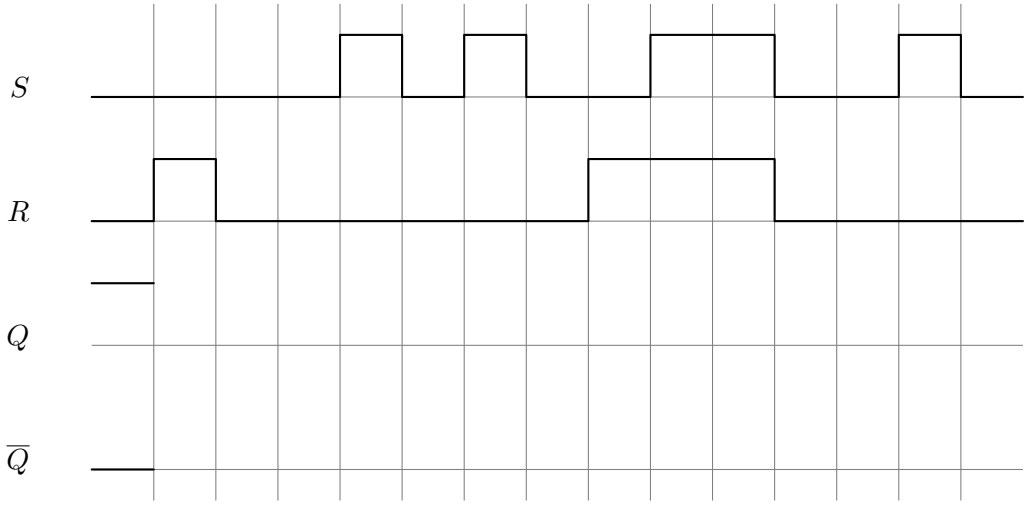


As you can see, the output of the T flip-flop is again a clock signal, but its frequency is divided by 2. This is how T flip-flops are often used: to divide a clock frequency by a power of 2 (one can put multiple T flip-flops after each other).

6.6 Exercises

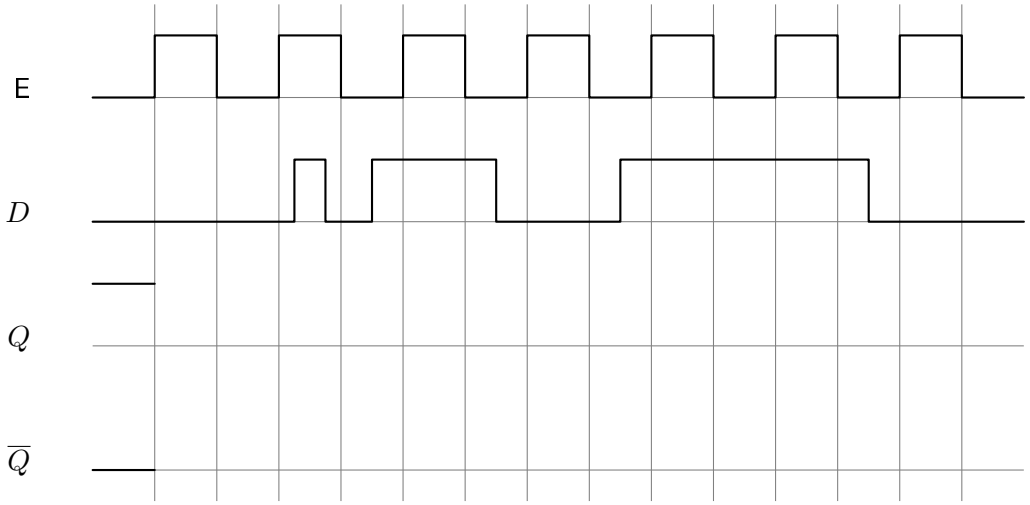
6.6.1 The RS-latch

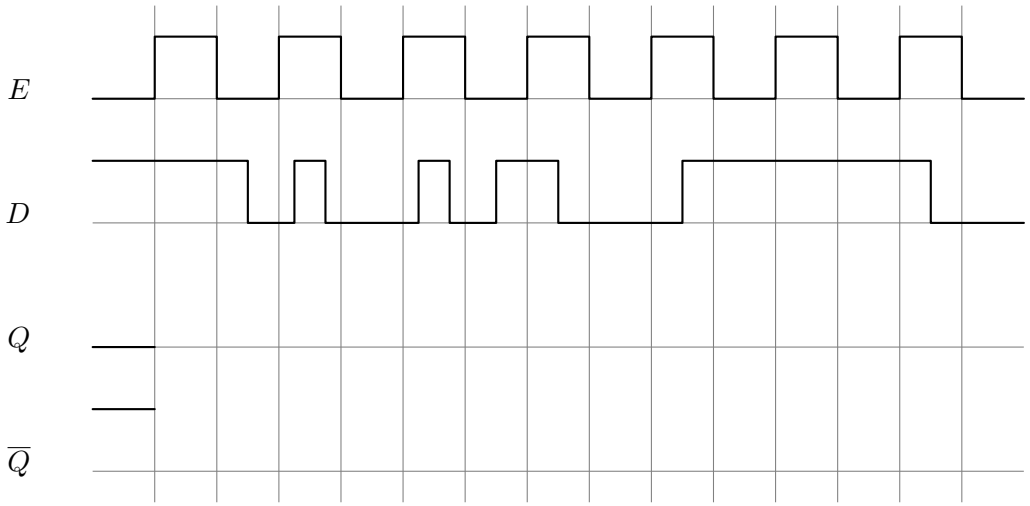
Complete the following timing diagram if you know that the circuit is a RS-latch:



6.6.2 Synchronous D-latch

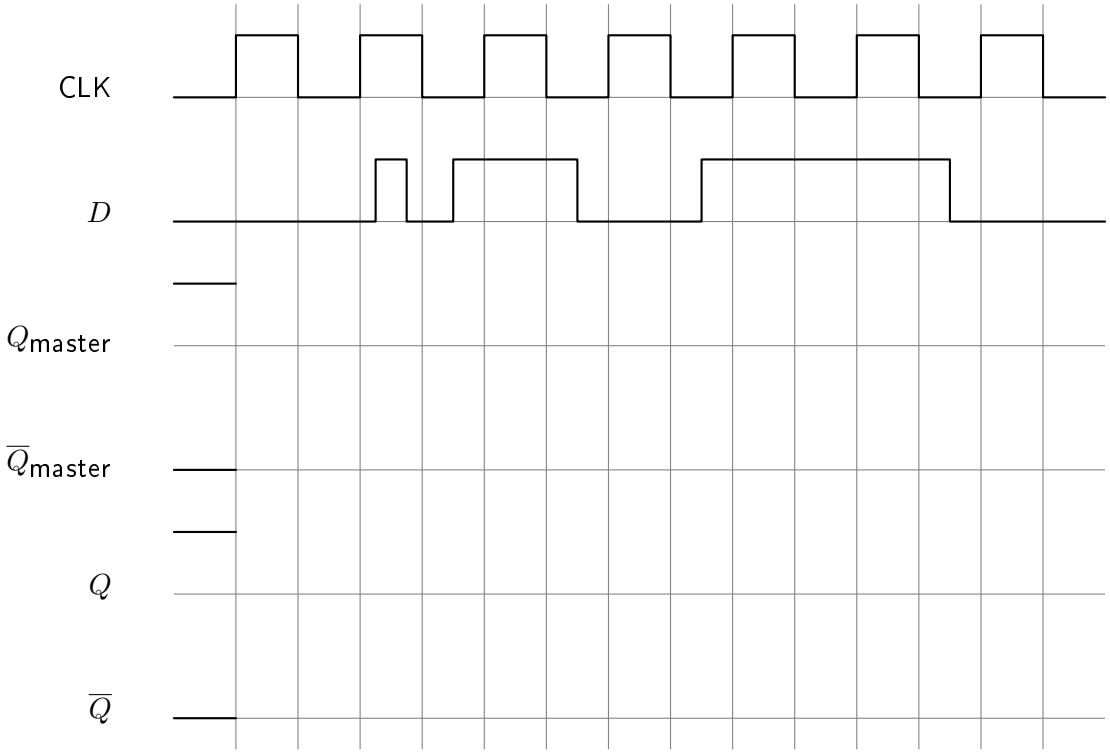
Complete the following timing diagrams if you know that the circuit is a synchronous D-latch:





6.6.3 Master-slave D flip-flop

Complete the following timing diagram if you know that the circuit is a D flip-flop:



Chapter 7

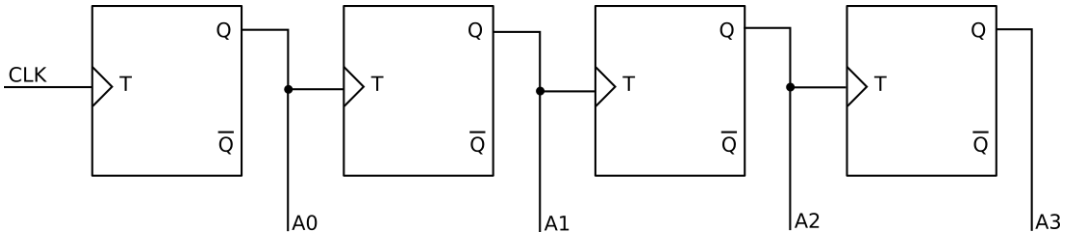
Common sequential circuits

In this chapter some very commonly used sequential circuits are discussed.

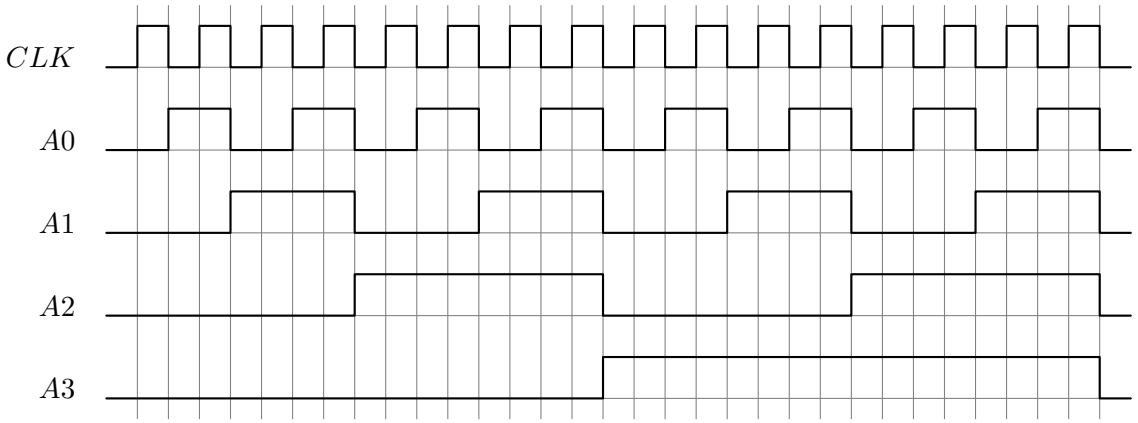
7.1 Counter

A counter is a circuit that can count in binary. At every falling edge of the clock, the counter is incremented by 1.

A counter can very easily be constructed with T flip-flops. A counter with 4 bits looks like this:



The timing diagram becomes:

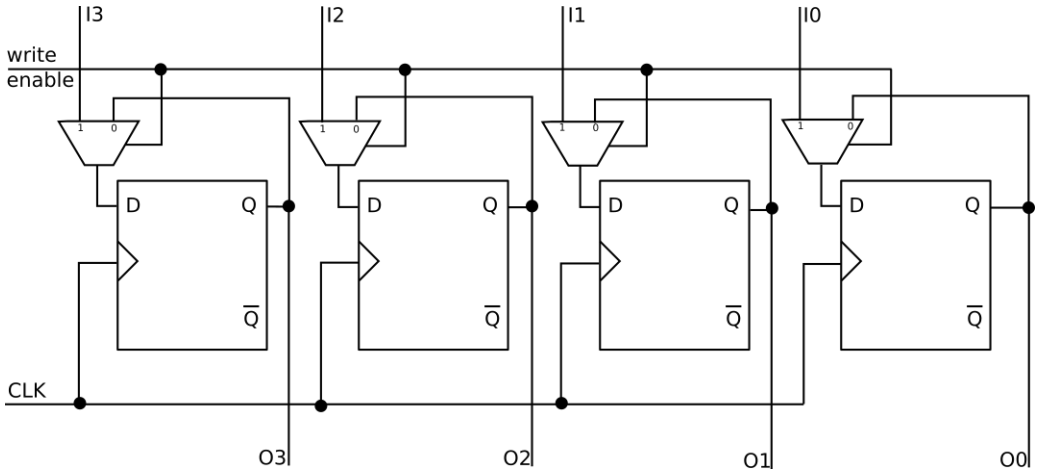


Every T flip-flop will divide its incoming frequency by 2 and give it to the next. If one looks at the output as one number (where A0 is the LSB and A3 is the MSB), the output evolves as follows at each falling edge: 0000, 0001, 0010, 0011, 0100, ..., 1111. After this the output will go back to 0000.

7.2 Registers

Registers are groups of D flip-flops that can hold a binary number. They all share the same clock, so they can read a new value simultaneously (in 'parallel'). As we shall see in chapter 9 on page 98, a processor contains many of these registers, each holding a binary number. All registers share the same clock signal. This means that if one wants to store a new value in a certain register, it needs to be 'selected' somehow. This is comparable to the synchronous D-latch in which an enable signal was added.

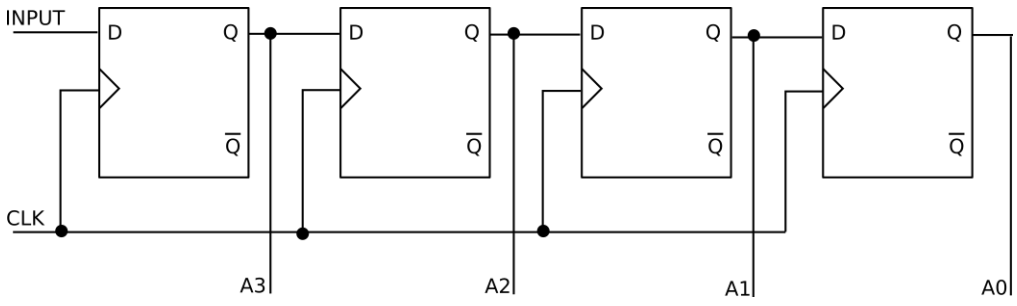
Here we will also add an enable signal that will determine if a new value needs to be written in the register. If the enable signal is equal to zero, the register will retain the value already stored. This is done using MUX's that will select the new or the old value to store. The circuit looks like this (for 4 bits):



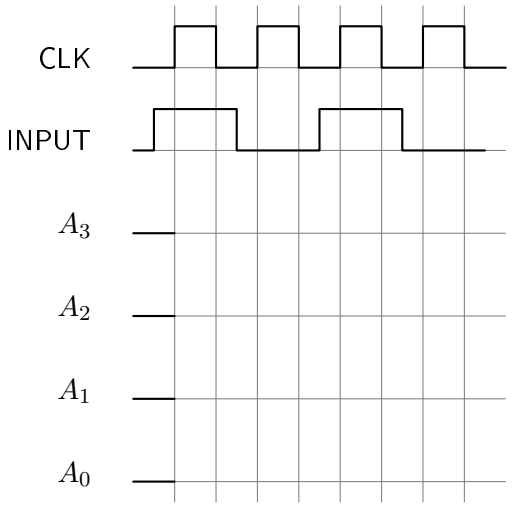
7.3 Shift registers

Transferring data from one device to the other requires a lot of wires. For instance, a mouse has to send 8 bits of data to the computer with every little step it is moved. This means that the mouse should have at least 8 wires to connect to the computer. In order to reduce the number of wires needed the data is not sent in parallel, but 'serial'. In a serial connection data is sent bit by bit. So one can use only one wire to send information. A USB connection is a good example of this. It only has 4 wires: 2 are used for power and the other 2 are used to send and retrieve data.

In order to convert serial data to parallel data and vice versa, a shift register is used. Here is the circuit for a shift register of 4 bits. It converts incoming serial data to parallel output:



Every clock cycle (at the falling edge) all data in the register is shifted to the right and new data is read from the left. The following timing diagram shows the behaviour (complete it as an exercise):



The INPUT signal is the serial data. The LSB is sent first and shifted all the way to the last flip-flop. After 4 clock cycles the data can be read in parallel from the outputs of the flip-flops.

7.4 Exercises

7.4.1 Simulation

Simulate the given circuits in Logisim or Digital. Play with them so you can learn their behaviour.

7.4.2 Shift register

Create a shift register and add an output that becomes one when the next four bits were received. In other words, add a circuit that outputs a 1 every 4 clock cycles.

7.4.3 Serial transmitter and receiver

Create 2 shift registers (4 bits) that are connected to one another. The first shift register is the transmitter. It can be loaded with a new value in parallel and send the data bit by bit. This circuit will be a combination of the register and shift register circuits from above.

The second circuit is a receiver (as shown above).

There can only be 2 connections between the transmitter and receiver: the data-line and the clock.

Simulate these circuits and see if you can send 4 bits from the transmitter to the receiver.

7.4.4 Bidirectional shift register

Create a shift register that can shift its data to the left or to the right. The direction is given by an extra input.

7.4.5 Counter

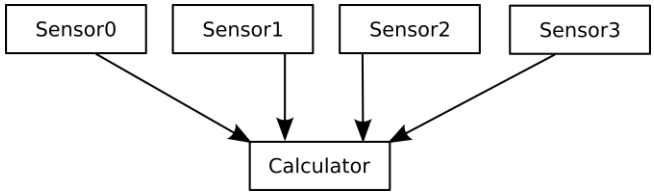
Create a circuit that contains 6 bits. Every clock cycle the circuit adds 3 to its output and stores the result. This circuit was already discussed in 6.4 on page 85. Simulate it and verify its workings.

Chapter 8

Bus structures

8.1 The problem

Imagine one wants to calculate the average temperature of 4 different rooms. In every room there is a sensor that measures the temperature as an eight bit 2-complement number. The four numbers need to be sent to a main circuit (the calculator) that will calculate the average. This situation is shown below:



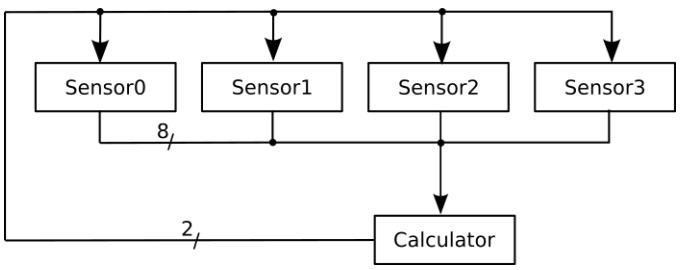
Since every sensor produces eight bits, eight cables need to be drawn from each sensor to the calculator. This means 32 wires are used to connect the sensors with the calculator. If one wants to add sensors, the number of wires continues to increase.

One solution would be to send all the data bit by bit (serial), reducing the number of wires dramatically. But this chapter will introduce another solution that can be used: busses.

8.2 The solution

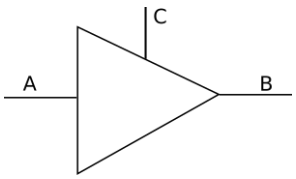
Since the Calculator only needs to calculate the average from time to time, it would be perfectly acceptable to read the sensors one by one and then calculate the average. If only one sensor is sending its value, there is no need to have 8 wires per sensor. The sensors could share 8 wires, resulting in a 'bus' structure. A bus is a bunch of wires that is shared between different circuits. Of course, each sensor now needs to know if it needs to send its value or not. The calculator will decide this using 2 wires that indicate the number of the selected sensor.

The result looks like this:



The lines in the above circuit are in fact multiple lines as indicated with the given number. The calculator will now first select a sensor using the 2 lines on its left. These lines are called the 'control bus'. They are shared with all sensors. If a sensor detects its number on the control bus, it will put its output on the 8 lines towards the calculator. These lines are called the 'data bus'. The calculator will select each sensor one by one (using a counter) and accumulate the result at each step. The calculator will use registers to store the values that were received from the sensors. Since the calculator decides who can put data on the data bus, it is called the 'master' of the bus.

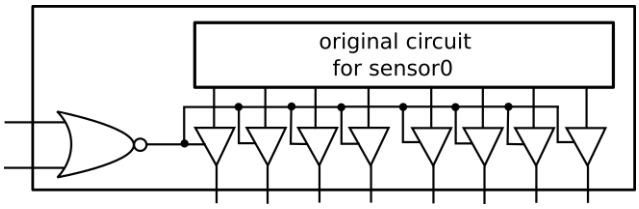
This might seem as a good solution, but there is still a problem. The output of the sensor is a binary number. If it is not selected, it will output 00000000. This is still an electric signal. This means that those signals will interfere with the data coming from the selected sensor, resulting in a short circuit. Somehow we need to be able to completely cut the connection between a sensor and the bus. This is done with 'tri-state buffers'. A tri-state buffer is like a switch. Its symbol is the following:



When C is equal to 1, the output B will be equal to the input A (so the switch is closed). When C is equal to 0, the output B will be disconnected. In this text we will indicate this with the letter Z (sometimes called 'high impedance'). So this circuit can output three values: 0, 1, and Z. The truth table is as follows:

A	C	B
0	0	Z
0	1	0
1	0	Z
1	1	1

The circuits of the sensors need to be adapted. They need to know if they are selected and use tri-state buffers to connect the output to the data bus. For sensor0 this could look like:



The NOR gate will detect if the number 00 was put on the control bus. It will drive all 8 tri-state buffers so the output of the sensor will appear on the bus if and only if it was selected.

8.3 (Static) RAM memory

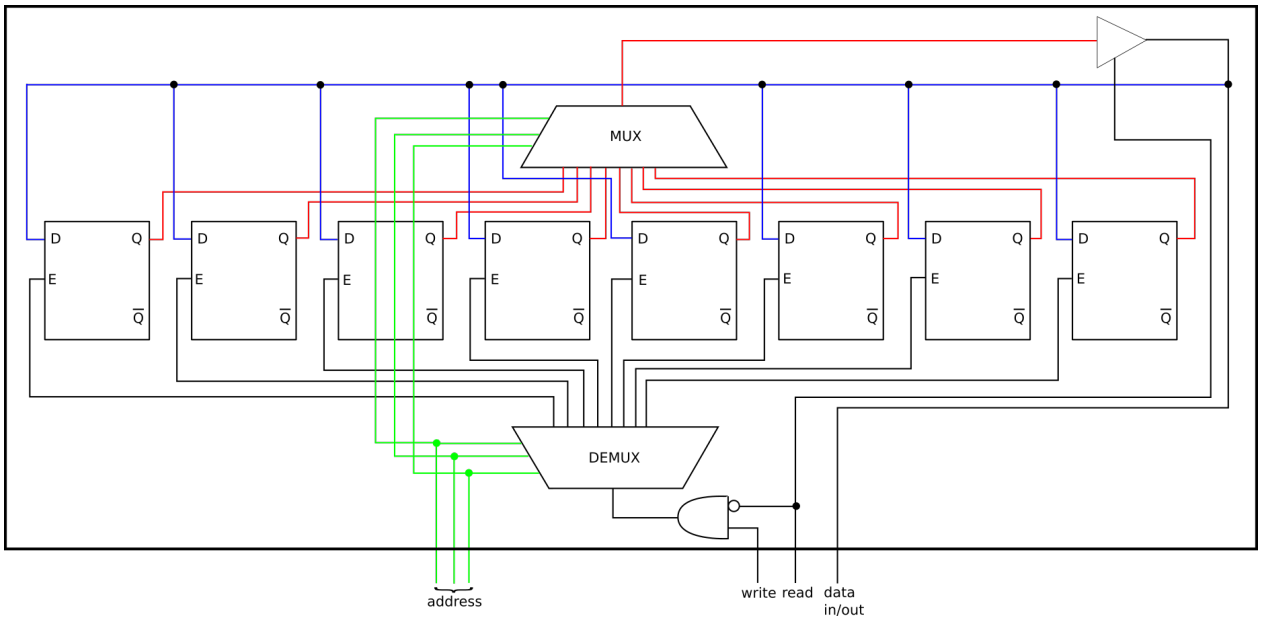
RAM (Random Access Memory) is a very important component in a computer. It is used to hold to code to be executed and the data that it processes. RAM can be interpreted as a gigantic table with 2 columns: addresses and values. In a memory of 16 GB the addresses go from 0 to

17179869183 (0x3FFFFFFFF). Addresses are mostly given in hexadecimal. The values consist of bytes (8 bits). The processor is able to read values from RAM and write values to it. In order to do this, the following connections are present on a RAM:

- *'address bus'*: the processor uses this to indicate an address in memory where data needs to be read or written
- *'control bus'*: these are signals that indicate whether the processor wants to read, write, or do nothing
- *'data bus'*: these pins are used in 2 ways: the processor can read data from it or put new data on it. Tri-state buffers are used so no short circuits can occur.

A RAM can be constructed out of D-latches (in which case we speak of 'Static RAM' or 'SRAM'). Each D-latch holds 1 bit of information. The address bus is connected to a gigantic MUX and DEMUX. The MUX is used to select a latch for reading. The DEMUX is used to select a latch to write data to.

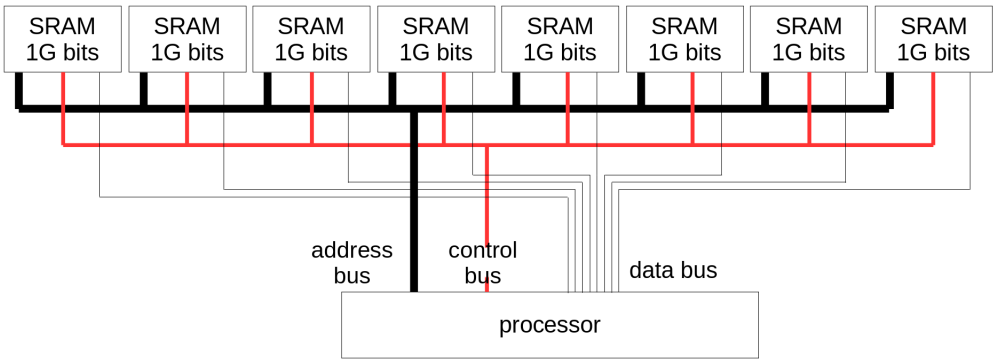
A simple SRAM holding only 8 bits is shown here:



If the processor wants to write a bit in a certain location, the bit is presented at the 'data in/out' pin. The address is put on the address bus. The processor will then put the 'write' signal to 1 and the 'read' signal to 0. Now the enable signal of the selected D-latch is activated and allows the bit to be stored.

If the processor wants to read a bit from a certain location, the address is put again on the address bus. The processor will put the 'write' signal to 0 and the 'read' signal to 1. This will enable the selected bit to flow through the tri-state buffer to the 'data in/out' pin.

In order to make a larger memory, the MUX and DEMUX will be made bigger. But it will still only allow one bit to be set or read. In order to be able to store a byte 8 of these circuits are put in parallel as follows:



Each of the 'SRAM 1G bits' blocks is the same circuit as above, but with 1073741824 D-latches! The address bus is 30 bits wide in this case. The left SRAM will store all the MSB's of each location and the right SRAM will store the LSB's of each byte. On older computers one can see exactly this setup as there were 8 different chips on the memory modules. These days even more SRAMs are combined to the computer can read or write 64 bits in one go (the data bus is 64 bits wide is this case).

SRAM is incredibly fast and is used in video cards but it also needs a lot of space on a chip. This is because a lot of gates are needed per bit. In order to make cheaper RAM, one can also replace the D-latches with capacitors. A capacitor is a small electronic component that can hold an electrical charge for a short time. Since it depletes very fast, it needs to be refreshed all the time. This type of RAM is therefore called 'dynamic RAM' or 'DRAM'. In this text we will always assume SRAM.

8.4 Exercises

8.4.1 Bus architecture

Simulate the circuit with the sensors given above. You can replace each sensor with 8 inputs so you have control over the value they send to the calculator.

You can also replace the calculator by inputs and outputs, so you can be the calculator yourself. Try to select a sensor and verify that its value is sent to the calculator.

Expert level exercise: if everything works as expected, try to create the circuit for the calculator. It needs a counter that will count from 0 to 3 and back to 0. It will also need a clock to trigger the counter. Next, it needs 4 registers that can retrieve the value of the sensor. A DEMUX can be used to trigger the right register when a value arrives. Finally, the calculator needs a circuit that computes the average of the four numbers in the registers (you can add them with full adders and divide by performing a 'arithmetic shift right'.

8.4.2 Alarm

Imagine you want to create an alarm system. When turned on, the alarm will go off as soon as a door or window is opened. Imagine 10 windows and 3 doors that need to be monitored. Each window or door has a sensor that outputs 1 bit.

Make the alarm first without a bus: each of the outputs is gathered together to the central alarm system. In this system a simple OR gate can decide if one of the alarms went off.

But 13 cables is not practical. So recreate the system with 1 data-wire that will pass by every sensor. Every sensor is connected to this wire. Also introduce a control bus with 4 wires so you

can select a sensor. Modify the sensors so that, if selected, they will put their output value on the data-wire with a tri-state buffer.

You can emulate the sensors with inputs and the alarm system with inputs and outputs.

Expert level: make the circuit for the central alarm system. It will have a clock, a counter and a circuit that will sound the alarm if a 1 is sent by any of the sensors.

8.4.3 SRAM

Create a circuit that can hold 4 bytes. You can use 8 sub-circuits that can each hold 4 bits. The address bus will be 2 bits wide and the data bus will be 8 bits.

Chapter 9

The processor

In this chapter a complete working processor (the 'Moncky-1') is built only using circuits discussed before. It is a fictive processor that is especially designed for educational purposes. Although real processors are much more complicated it is capable of performing the same tasks, except for tasks that involve interrupts (this complicates the architecture of the processor significantly). It is perfectly feasible to create a physical chip for the Moncky-1, using ASIC technology or FPGA chips. A simulation in logisim exists and can be retrieved through the author.

The Moncky-1 processor has the following properties:

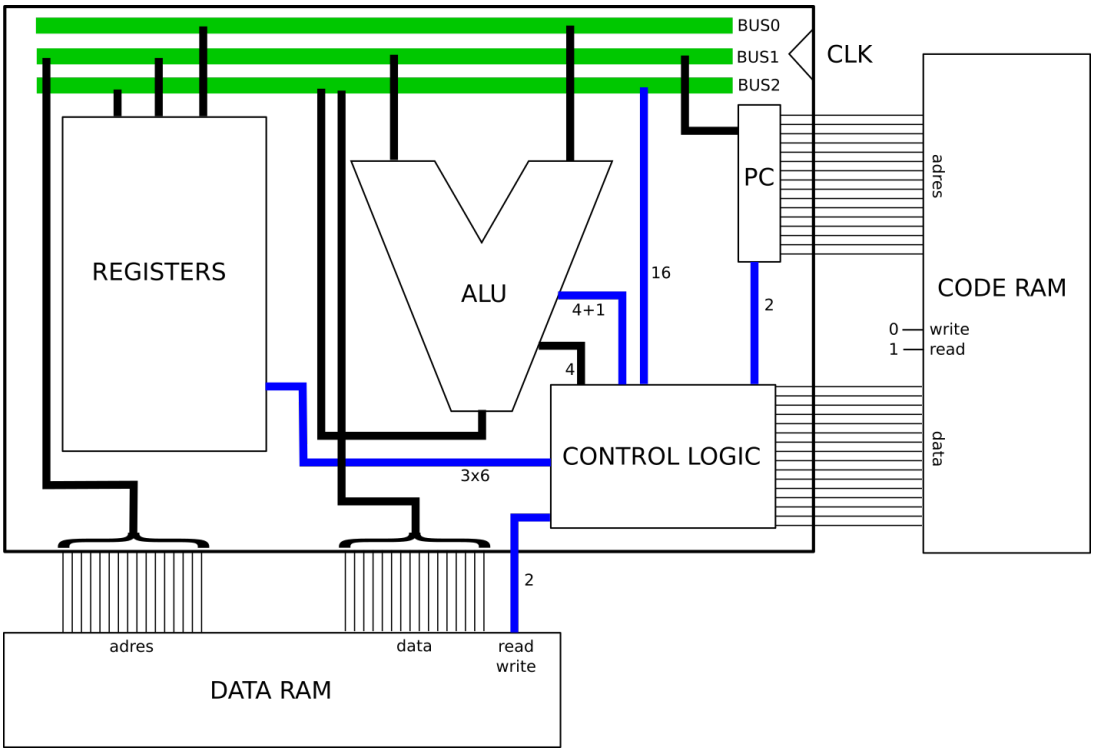
- there are only 7 instructions
- each instruction can be performed in one clock cycle
- memory is limited to 2x128 KiB
- there are no interrupts (see also 10.7 on page 126)

9.1 Architecture

A processor always contains at least the following components:

- the 'registers': these hold temporary data
- the 'ALU': this will perform arithmetic and logic operations on the data
- the 'instruction decoder' or 'control logic': this interprets the instructions and controls the other components in the processor
- internal busses: these connect all the components together

The Moncky-1 processor has the following architecture:



Two RAMs are connected to the processor. In this case both RAMs have an address bus that is 16 bits wide. The data bus is also 16 bits wide. A RAM can thus be regarded as an array of 65536 values. Each value is a 16 bit number.

The 'code RAM' contains instructions that need to be executed. Each instruction is 16 bits wide. The 'data RAM' is a memory that will hold the values of variables. This text will only concentrate on values that are 16 bits long, but this processor can easily process larger numbers. Using two RAMs makes the design of the processor simple and keeps performance very high.

There are 16 registers in the Moncky-1. Each register can hold a number of 16 bits. The registers are numbered: r0, r1, ..., r15. They are used to hold temporary data i.e. data that is used to perform a calculation. If data needs to be stored, it is always saved to the data RAM.

There is one special register called 'PC' (Program Counter). This register is connected to the address bus of the code RAM. It is in fact a counter that starts at 0 and increments each clock cycle. It holds the address of the instruction that needs to be executed. The code RAM is hardwired for reading. It will put the value at the given address on its data bus. The register PC is sometimes also called 'instruction pointer' (IP) as it points to the current instruction.

It is important to understand that any 16-bit number can be interpreted in many different ways, depending on how it is used. So a register containing a value could be any of these:

- an unsigned number ranging from 0 to 65535
- a 2-complement number ranging from -32768 to 32767
- an IEEE 754 floating-point number ranging from -65504.0 to 65504.0
- an instruction (see below for the interpretation)
- an address in the data RAM ranging from 0x0000 to 0xFFFF
- an address in the code RAM ranging from 0x0000 to 0xFFFF

The ALU (Arithmetic Logic Unit) is capable of doing calculations. It reads two numbers, performs the calculation, and puts the result on its output. It also generates 'flags' that give information about the result. The inputs are on top and the output is on the bottom.

There are 3 busses in the Moncky-1. They allow the components to send information to each other. For example, the ALU will read the two numbers from busses 0 and 1. The value on bus 0 and bus 1 will come from a chosen register. The control logic will determine which registers are put on the busses. The result of the ALU is put on bus 2 and can be read into a register.

The control logic is connected to the data bus of the code RAM. It will read the instruction and will translate this into control signals to the other components.

The processor has a clock input. This signal is assumed to alternate between 0 and 1 at a regular rhythm. In modern processors this happens about 2 to 3 billion times per second. For the Moncky-1 processor a clock speed of about 1Ghz should be attainable (1 billion clock cycles per second). All registers, including PC, share are connected to this clock.

9.2 The instruction set

In the code RAM all values are interpreted as instructions. In this section we shall give an overview of all instructions.

In the Moncky-1 processor all instructions are exactly 16 bits long but there are processors that have instructions of 8 bits or 32 bits long. There are also processors with variable length instructions.

The Moncky-1 processor knows the following 7 instructions:

- stop executing instructions (this is used to stop the processor after executing a program)
- load a number (a constant) into a register
- perform a calculation with the registers and put the result back into a register
- load a number from data RAM (at a given location) into a register
- store a number from a register into data RAM at a given location
- jump to a specific address in the code RAM
- jump to a specific address in the code RAM if the result of the previous calculation was zero, negative, positive, ...

Every instruction is embedded in 16 bits. These bit patterns constitute the 'machine language' of the processor. Because the are very hard to read, a more human readable version is also provided. It consists of a short command name and parameters. This language is called 'assembly language'.

In the Moncky-1 processor the 7 instructions are encoded in 16 bits as follows:

machine language	assembly	meaning	example
`0000 xxxx xxxx xxxx`	halt	the processor stops	halt
`00x1 iiii iiii rrrr`	li r, i	reg[r] = i	li r3, 42
`01xx AAAA rrrr ssss`	ALU r, s	reg[r] = reg[r] ALU reg[s]	add r3, r4
`100x xxxx rrrr ssss`	ld r, (s)	reg[r] = DATARAM[reg[s]]	ld r2, (r15)
`101x xxxx rrrr ssss`	st r, (s)	DATARAM[reg[s]] = reg[r]	st r1, (r15)
`110x xxxx xxxx rrrr`	jp r	PC = reg[r]	jp r12
`1111 xxxx xccc rrrr`	jpc r	if (c) then PC=reg[r]	jpz r1

The following notation is used:

- 'iiii iiii' stands for an 8-bit value that is embedded into the instruction. This is also called an 'immediate' value as the value is immediately available from the instruction.
- "rrrr" and "ssss" stand for register numbers (0-15)
- "AAAA" stands for an ALU operation. The following operations are supported:
  - 0000 = NOP (No OPeration: the result is equal to the second input of the ALU)
  - 0001 = OR (a bitwise OR)
  - 0010 = AND (a bitwise AND)
  - 0011 = XOR (a bitwise XOR)
  - 0100 = ADD (an addition)
  - 0101 = SUB (a subtraction)
  - 0110 = SHL (shift left: the first number is shifted to the left. The number of bits to shift is indicated by the second number)
  - 0111 = SHR (shift right)
  - 1000 = ASHR (arithmetic shift right)
  - 1001 = NOT (logic inverse: the result is the 1-complement of the second input)
  - 1010 = NEG (negative: the result is the 2-complement of the second input)
- "ccc" indicates a certain condition. Every time a calculation is done by the ALU, 4 flags are set according to the result: the carry flag, the zero flag, the sign flag, and the overflow flag). These flags can be used to decide to jump to a certain location in the code RAM. The following conditions can be checked:
  - 000 = 'c' = if the carry flag is equal to 1 (the calculation resulted in 1 extra bit being discarded)
  - 001 = 'nc' = if the carry flag is equal to 0 (the calculation did not result in an extra bit)
  - 010 = 'z' = if the zero flag is equal to 1 (the result of the calculation was 0)
  - 011 = 'nz' = if the zero flag is equal to 0 (the result of the calculation was not equal to 0)
  - 100 = 's' = if the sign flag is equal to 1 (the result of the calculation was negative)
  - 101 = 'ns' = if the sign flag is equal to 0 (the result of the calculation was not negative)
  - 110 = 'o' = if the overflow flag is equal to 1 (the calculation resulted in an overflow)
  - 111 = 'no' = if the overflow flag is equal to 0 (the calculation did not result in an overflow)
- "x" stands for don't care (the values of these bits can be anything; we will always put them to zero in the examples below)

The 4 most significant bits indicate which instruction is to be executed. They are called the 'opcode'. The other bits hold the parameters for that instruction. These are called the 'operands'.

A small program for the Moncky-1 could look like the following:

```
0001 0000 0101 0000   li r0, 5    ; load the number 5 in register 0
0001 0000 0011 0001   li r1, 3    ; load the number 3 in register 1
0100 0100 0000 0001   add r0, r1  ; add register 0 and 1 together
                                    and put the result in register 0
0001 0000 0010 0001   li r1, 2    ; load the number 2 in register 1
1010 0000 0000 0001   st r0, (r1) ; store the value van r0 (8 in this case)
                                  ; at location given by r1 (2 in this case)
                                  ; in the data RAM
0000 0000 0000 0000   halt        ; end the program
```

The left column shows the binary form of the machine code. These are the bits that are stored in the code RAM. The middle column shows the assembly version of the instructions. The third column contains comments (here a comment starts with a semicolon).

This program is stored in the code RAM. This means the code RAM will contain the following values (in hexadecimal): $[1050, 1031, 4401, 1021, A001, 0000]_{16}$.

When the processor is started, PC is set to 0. This means that the first instruction will be executed. After one clock cycle PC is incremented and the next instruction can be executed. This continues until PC=5. Since the instruction is 'halt', PC will not be incremented any more. This halts the processor.

9.3 The Compiler

Machine code and assembly language is very dependent on the processor architecture. This means that every processor has its own language. This poses a problem when one wants to run the same application on different processors. In order to overcome this, a program is written in a higher programming language like C, C++, java, C#, ... These languages are independent of the processor. The program is then translated to the target processor. This translation is done by a 'compiler'.

For each variable the compiler will assign an address in the data RAM. The value of a variable is always kept in the data RAM at its own location. If the value needs to be changed, the processor will read it into a register, perform calculations, and then write the result back to the data RAM. It is important to see that the registers do not hold information long. This is particularly important when a system consists of multiple processors (or cores) which all have their own registers but share the same data RAM.

The compiler translates code statement per statement, independently from each other. The registers can and will be reused whenever possible. The compiler will not keep track of the values in registers after translating a statement.

In order to illustrate this, three examples are given: calculating with variables, programming a condition, and programming an iteration.

9.3.1 Variables and calculations

Imagine the following java code (notice we only use the 'short' datatype as it contains 16 bits; other types can also be used, but make things much more complicated):

```
short a;
short b;
```

```
a = 4;
b = 38;
a = a + b;
```

The compiler will translate this code as follows:

- the first statement only declares a variable. No value is stored in it, so it is only a declaration. This means no code must be generated by the compiler. The compiler will just determine where the value of the variable will be stored in the data RAM. The compiler keeps a table with all the variables and their location (this is called the 'symbol table'). In this example we will assume that address 33 is available and it is chosen to hold the value of the variable 'a'.
- the second statement declares another variable 'b'. So the compiler will select another address in the data RAM to store its value. We assume the compiler chooses address 34.
- the third statement stores the value 4 into the variable 'a'. This means that the processor should store the value 4 in location 33 in the data RAM. This cannot be done in one assembly instruction. We need 3 instructions to be able to do this:

```
0001 0000 0100 0000    li r0, 4     ; load the number 4 in register 0
0001 0010 0001 0001    li r1, 33    ; load the address of 'a' in register 1
1010 0000 0000 0001    st r0, (r1)  ; write the value of r0 (4) to the
                                      data RAM at the location to which r1
                                      'points' (33)
```

- the fourth statement does almost the same as the third. Now the value 38 needs to be stored in the data RAM at location 34. Since all registers can be reused, the compiler will copy the previous code and just enter the new values:

```
0001 0010 0110 0000    li r0, 38    ; load the number 38 in register 0
0001 0010 0010 0001    li r1, 34    ; load the address of 'b' in register 1
1010 0000 0000 0001    st r0, (r1)  ; write the value of r0 (38) to the
                                      data RAM at the location to which r1
                                      'points' (34)
```

- the fifth statement adds the value of 'a' and 'b' together and stores the result back into 'a'. The processor can only calculate with registers. So the first thing it needs to do is load the values of both variables into registers. Then it can add the two numbers and finally it can store the result back into the data RAM. The compiler will generate the following code to accomplish this:

```
0001 0010 0001 0000    li r0, 33    ; load the address of 'a' in register 0
1000 0000 0001 0000    ld r1, (r0)  ; load the value of 'a' into register 1
0001 0010 0010 0000    li r2, 34    ; load the address of 'b' in register 2
1000 0000 0010 0000    ld r3, (r0)  ; load the value of 'b' into register 3
0100 0100 0001 0010    add r1, r3   ; r1 = r1 + r3
1010 0000 0001 0000    st r1, (r0)  ; store the result back in the data RAM
                                      at the location of 'a'. This value is
                                      still in r0
```

Notice that the compiler never reuses the values of variables over statements. But within a statement registers can be reused (like the value of r0 in the last statement).

This code is put in the code RAM. So the contents of the code RAM will look like (all values are hexadecimal):

address	value
0	1040
1	1211
2	A001
3	1260
4	1221
5	A001
6	1210
7	8010
8	1220
9	8020
A	4412
B	A010
C	0000

The processor can now execute this code. How that works is discussed in9.4 on page 111.

9.3.2 Selections

Programs do not only consist of sequential operations. In many cases code needs to be executed only when some conditions are met. In java and many other languages this is achieved using an 'if statement'. In assembly language there is no 'if' instruction available. But using the jump and conditional jump instructions, the same effect can be realised. The following example illustrates this. We want to translate the following code:

```
short a = 3;
if (a == 4) {
  a++;
} else {
  a--;
}
```

In order to translate this, the code is transformed into another format:

```
short a = 3;
if ((a-4) != 0) goto else;
  a++;
  goto end;
else:
  a--;
end:
```

This code is not valid java any more, but in C this is still allowed (although strongly discouraged). One can see now that there are 2 goto instructions: the first one is only executed if the result of

the calculation returns true and the second is unconditional. The Boolean expression 'a==4' is here replaced by a subtraction. In assembly language this is often done. If one needs to compare two numbers, just subtract them and look at the result (using the flags).

This code can be translated in to assembly like this (the variable 'a' is put at address 42 in this example):

```
; a = 3
 0: li r0, 42    ; load the address of 'a' in r0
 1: li r1, 3     ; load the number 3 in r1
 2: st r1, (r0)  ; store r1 (3) at location r0 (42) in the data RAM
; if ((a-4)!=0) goto else;
 3: li r0, 42    ; load the address of 'a' in r0
 4: ld r1, (r0)  ; load the value of 'a' into r1
 5: li r2, 4     ; load r2 with the value 4
 6: sub r1, r2   ; subtract r1 from r2 (a-4)
 7: li r3, 0x10  ; load the address of 'else' in r3
 8: jpnz r3      ; if the result of the last calculation
                   was not zero, goto r3 (line number 0x10)
; a++
 9: li r0, 42    ; load the address of 'a' in r0
 A: ld r1, (r0)  ; load the value of 'a' in r1
 B: li r2, 1     ; set the value of r2 to 1
 C: add r1, r2   ; add r1 and r2 (a+1)
 D: st r1, (r0)  ; put the result (r1) back into the data RAM
; goto end
 E: li r0, 0x15  ; load the address of 'end' in r0 (0x15)
 F: jp r0        ; jump to 'end'
:else
; a--
10: li r0, 42    ; load the address of 'a' in r0
11: ld r1, (r0)  ; load the value of 'a' in r1
12: li r2, 1     ; set the value of r2 to 1
13: sub r1, r2   ; subtract r1 and r2 (a-1)
14: st r1, (r0)  ; put the result (r1) back into the data RAM
:end
15: halt         ; end of program
```

So if one needs to compare two numbers they are always subtracted. Doing the subtraction makes the ALU set its flags. Using these flags, one can jump to another location if necessary. The following flags are most frequently used:

- zero flag = 1: the two numbers were equal
- zero flag = 0: the two numbers were different
- sign flag = 1: the second number was greater than the first
- sign flag = 0: the second number was smaller than or equal to the first

The following conditional jumps can be used: jpz, jpnz, jps, and jpns.

Notice that one always needs to put the address to jump to in a register. These values can be hard to predict. E.g. at line 7 we need to put the value 0x10 in the register. This is the line

number to jump to. But when writing the code, one cannot know already where this is going to be. Therefore, many assembly languages allow to use labels. In the Moncky-1 assembly language a label always starts with a colon. This means that we can rewrite line number 7 as:

```
6: li r3, :else
```

The expression ':else' stands in this case for the number 0x10. In the next examples we will always use labels as it makes the code more readable.

9.3.3 Iterations

It is also possible to implement iterations in assembly language. These are loops that need to be executed a number of times until a certain condition is met. Here is an example:

```
short sum = 0;
for(short i=0; i<100; i++) {
    sum += i;
}
```

In order to translate this into assembly language, the code is rewritten to this equivalent code:

```
short sum = 0;
short i = 0;
while (i<100) {
    sum += i;
    i++;
}
```

Now, the code can be translated using goto statements as follows:

```
short sum = 0;
short i = 0;
loop:
if (i >= 100) goto end;
  sum += i;
  i++;
goto loop;
end:
```

Now the same structures are used as before. The code can be translated to assembly as follows ('sum' is stored at location 25 and 'i' is stored at location 30):

```
; sum = 0
 0: li r0, 25
 1: li r1, 0
 2: st r1, (r0)
; i = 0
 3: li r0, 30
 4: li r1, 0
```

```
 5: st r1, (r0)
; if (i >= 100) goto :end
:loop
 6: li r0, 30
 7: ld r1, (r0)
 8: li r0, 100
 9: sub r1, r0
 A: li r2, :end
 B: jpns r2
; sum += i
 C: li r0, 25
 D: ld r1, (r0)
 E: li r0, 30
 F: ld r2, (r0)
10: add r1, r2
11: li r0, 25
12: st r1, (r0)
; i++
13: li r0, 30
14: ld r1, (r0)
15: li r2, 1
16: add r1, r2
17: st r1, (r0)
; goto :loop
18: li r0, :loop
19: jp r0
:end
1A: halt
```

The labels 'loop' and 'end' will be equivalent to the numbers 0x06 and 0x1A respectively.

9.3.4 Special constructions

As there are only seven instructions in the processor, one has to be creative to translate a higher programming language into assembly.

In this section a few tricks are shown that occur very frequently in assembly languages.

9.3.4.1 Loading big numbers

Imagine one wants to translate the following statement:

```
short a = 0x1D5F;
```

The variable 'a' is stored at location number 5 in the data RAM. Following the previously discussed reasoning, this could translate into:

```
li r0, 5
li r1, 0x1D5F
st r1, (r0)
```

But this is impossible. You can see why by translating the second line into machine code. The 'li' instruction only has 8 bits for the number, but now we want to load a 16-bit number into r1. In order to do this, the following trick is used: load the MSB in a register, shift it to the left and add the LSB to it. In assembly, this looks like the following:

```
li r0, 5
li r1, 0x1D
li r2, 8
shl r1, r2
li r2, 0x5F
add r1, r2
st r1, (r0)
```

The 'add' instruction can also be replaced by an 'or' instruction. See if you can understand why. In the Moncky-1 processor there is no reason to replace the 'add' by an 'or', but on most processors the 'or' is much faster than an 'add'.

9.3.4.2 Loading negative numbers

The 'li' instruction will always add zero's in the MSB. This means one can only load positive numbers from 0 to 255 with this instruction.In order to load negative numbers in a register, there are two possibilities: one can load the positive value and use the 'neg' instruction to make it negative, or one can load the 2-complement value as a 16-bit value as described in the previous section. These two strategies are demonstrated below:

In order to store -7 into r0:

```
li r0, 7
neg r0, r0
```

In order to store -700 in r0 (this is 0xFD44 in 2-complement):

```
li r0, 0xFD
li r1, 8
shl r0, r1
li r1, 0x44
add r0, r1
```

9.3.4.3 Clearing a register

In many occasions one wants to clear the contents of a register (i.e. putting its value to zero). There are different ways to do this. The following three statement will all put the value of r0 to zero:

```
li r0, 0    ; load 0 into r0
sub r0, r0  ; subtract r0 from itself
xor r0, r0  ; do a bitwise xor of r0 with itself
```

In the Moncky-1 processor there is no reason to prefer one instruction over the other because all instructions take 1 clock cycle to execute. But in other processors there can be a big difference in execution time. In that case, one has to choose wisely which instruction is used.

9.3.4.4 Copying a register to another

If one wants to copy the contents of a register to another, there are different ways to achieve this. Imagine that one wants to copy the contents of r5 to r6. This can be done as follows:

```
li r6, 0
add r6, r5
```

Or it can be done using only logic operators:

```
xor r6, r6
or r6, r5
```

But on the Moncky-1 processor this can also be done in 1 instruction:

```
nop r6, r5
```

When the ALU executes a 'nop' instruction, it will not do anything and will output the value of r5 as its result.

9.3.4.5 Manipulating bits

In some processors (like the Atmel processor on an Arduino), some registers are connected to output pins. External hardware will be connected to these pins. If one wants to put one of these pins to 1 or 0, one has to manipulate a certain bit in a register. Manipulating bits is always done using a 'mask'. This is a bit pattern that indicates which bits should be altered or which bits should not be altered.

The following examples illustrate this technique.

Imagine r1 contains a number and we want to erase bits number 2 and 4 in this number. The other bits need to remain at their original value. This is done using the mask: 1111 1111 1110 1011. The mask has a 1 for each bit we want to keep. Using a bitwise AND makes will erase the bits. The following code demonstrates this:

```
li r0, 0xFF
li r2, 8
shl r0, r2
li r2, 0xEB
add r0, r2
and r1, r0
```

This code loads the number 0xFFEB (the mask) in register r2 and AND's this number with the number in r1. This erases the bits where the mask is 0.

One can also set certain bits in a register. In this case, the mask contains a 1 for all the bits that need to be set and a 0 for all the bits that need to remain at their current value. Next, a logical OR is used to set the bits. The following code will set bits 2 and 4 in register r1 (the mask is now 0000 0000 0001 0100):

```
li r0, 0x14
or r1, r0
```

One can also toggle certain bits in a number, using an XOR. The following instructions will toggle bits 2 and 4 in register r1:

```
li r0, 0x14
xor r1, r0
```

9.3.4.6 Multiplication and division

There are no multiplication or division instructions on the Moncky-1 processor. This is because the circuits to implement that functionality need a lot of time do the calculations and they would slow down the whole processor since every instruction needs to fit in one clock cycle. In modern processors these instructions are implemented but they will also need a lot more clock cycles per instruction.

In order to implement a multiplication, one can use the Booth's algorithm. This is left as an exercise.

But in some cases, multiplications are very easy. More in particularly when one needs to multiply with a constant value.

If one wants to multiply the register r1 by 2, then the following statement will do exactly that:

```
add r1, r1
```

Another way to do this is:

```
li r0, 1
shl r1, r0
```

A shift left will multiply the value by 2. This solution takes more clock cycles but it is much more flexible. One can use the same code to multiply by any power of 2. For instance, multiplying r1 by 8 is done as follows:

```
li r0, 3
shl r1, r0
```

Dividing a number by a power of 2 is done in the same way. Now a shift right is used. If the number is 2-complement, an arithmetic shift right is used instead. For instance, dividing r1 by 16 is done like:

```
li r0, 4
ashr r1, r0
```

If the multiplier is not a power of two, but still a constant, one can sometimes do the multiplication very quickly by emulating a long multiplication. For example, imagine we want to multiply the value in register r1 by 80. Converting 80 into the binary numeral system yields: 0101 0000 (80=16+64). So this means the following:

$$r1 \cdot 80 = r1 \cdot (16 + 64) = r1 \cdot 16 + r1 \cdot 64$$

So we need to multiply r1 by 16 and by 64 and add the results together. Multiplying by 16 and 64 is easy: they are powers of two. So the resulting code is like this:

```
nop r2, r1  ; make a copy of r1 in r2
li r0, 4    ; 16=2^4
shl r1, r0  ; multiply r1 by 16
li r0, 6    ; 64=2^6
shl r2, r0  ; multiply r2 by 64
add r1, r2  ; add the two together
```

9.4 The internal components

In this section the different components of the Moncky-1 processor are discussed in detail. It explains their working and allows the enthusiast reader to build a working implementation. There is also a simulation in logisim available. Please contact the author if you would like to get that file.

9.4.1 The busses

The Moncky-1 contains 3 busses that connect the data inputs and outputs of the different components. They are numbered 0, 1, and 2. Every bus consists of 16 lines. The registers, the ALU, the PC register, the data RAM and the control logic are connected to them using tri-state buffers. The control logic is the master of the busses and decides who can write to which bus.

9.4.2 The registers

The Moncky-1 processor contains 16 registers that each contain a 16-bit number. All registers are built with master-slave D flip-flops. They all share the same clock. Every clock cycle a register can read a new value from a bus of write its value to a bus. MUX's and DEMUX's are used to select the right register and to indicate if it needs to read, write, or just retain its value.

Looking at the registers as one big sequential circuit, it has the following connections:

- a clock input that is forwarded to all 256 flip-flops.
- per bus:
  - 16 connections that can be configured as input or output. These are used to send or retrieve data
  - 4 inputs that determine which register should be connected to the bus (they lead to the MUX's and DEMUX's). These come from the control logic.
  - 1 input (rw) that determines if the the selected register should read or write. Its value is determined by the control logic.
  - 1 input (enable) that connects the selected register to the bus, or disconnects it. This will drive the tri-state buffers. Its value is determined by the control logic.

For example, if the control logic wants to send the information on bus 0 to register r5, the 4 inputs for bus 0 will be put on 0101. The rw input will be 0 (read) and the enable input will be 1. The enable inputs of the other busses will be set to 0 and their rw will be put at 1 (write). When the clock now goes from 1 to 0, register r5 will read the value from bus 0.

The PC register is also a register, but it can also increase its value by 1 every clock cycle. The PC register can put its value on bus 1 or read a new value from it.

9.4.3 The ALU

The ALU is in fact a very big combinational circuit. It has 37 inputs and 20 outputs. The output is always directly determined by the input. The inputs are:

- 16 inputs for the first number
- 16 inputs for the second number
- 1 input that determines if the output of the ALU should be connected to bus 2
- 4 inputs to select the operation the ALU needs to perform.

The outputs are:

- 16 outputs for the result
- 4 outputs that give information about the result (the flags):
  - zero flag: is the result equal to 0?
  - carry flag: is the most significant carry bit set?
  - sign flag: is the result negative?
  - overflow flag: was there a 2-complement overflow?

9.4.4 The control logic

The control logic is a main component in the processor. It interprets an instruction and drives all other components. It is a big combinational circuit with 20 inputs and 42 outputs.

The inputs are:

- 16 bits for the instruction. These bits come from the code RAM.
- 4 bits coming from the ALU (the flags)

The outputs are:

- 2 signals go to the control bus of the data RAM and indicate if the RAM needs to read or write or do nothing
- 4 signals go to the ALU to indicate the operation it needs to perform
- 1 signal goes to the ALU to indicate if it has to put its output on bus 2
- 2 signals to the PC register to determine if the register should increment, read its value from bus 1, or write its value to bus 1
- 3x6 signals to the registers determining which register should be connected to each bus and what action it should take
- 16 data signals to bus 2 to send a number directly to a register

The Moncky-1 processor is designed so that every instruction in executed in one clock cycle. When the clock is low, the control logic interprets the instruction and puts all the control signals to the right values. When the clock goes high nothing changes. The ALU and the data RAM get time to perform their operation and will put the result on the right bus. When the clock goes from one to zero (falling edge), the registers store the new value and the program counter is increased. When PC changes, a new instruction is presented to the control logic. The next subsections will discuss a few instructions and explain in detail how the control logic drives all components in order to execute the instruction.

9.4.4.1 Load instruction (ld)

The first instruction that will be discussed is 'ld r2, (r1)'. In this instruction r1 points to a location in the data RAM. The processor needs to load the value at that location and store it in register r2. Let's assume that register r1 contains the value 1024 and that the instruction itself is stored at location 2040 in the code RAM.

At the falling edge, when the clock goes from 1 to 0, PC reaches the value 2040 (`0000 0111 1111 1000`). This value is put on the address bus of the code RAM which drives the MUX within it, selecting the right value at this location. The RAM will respond by putting the instruction (`1000 0000 0010 0001`) on its data bus. The control logic notices this instruction begins with '`100`', so it knows it is a 'ld' instruction. The following signals are then sent to all components:

- the registers get the following signals per bus:
  - bus0: is decoupled (enable signal is put to 0)
  - bus1: bits 0-3 of the instruction (`0001`) are forwarded so that register r1 is selected. The rw signal is put to 0 (read) and the enable signal is put to 1. As a result, register r1 will put its value on bus1
  - bus2: bits 4-7 of the instruction (`0010`) are forwarded so that register r2 is selected. The rw signal is put to 1 (write) and the enable signal is put to 1. As a result, register 2 will read its value from bus2
- the data RAM is configured for reading (read=1, write=0). It will put the data on bus2.
- register PC is asked to increment itself on the next clock cycle

As a consequence of these signals, the value of register r1 (1024) flows through bus1 to the address bus of the data RAM. The data RAM will put the value stored at this location on bus2. The value is read by register r2. This register contains flip-flops so it does not yet store the new value.

When the clock goes from 1 to 0, register 2 will store the new value and PC will increment. The instruction is now executed and, since PC is incremented, the next instruction will appear in the control logic.

9.4.4.2 A calculation (add)

The second instruction is 'add r2, r3'. This instruction adds the values of r2 and r3 and puts the result back into r2. The instruction looks like: `0100 0100 0010 0011`.

When the control logic gets this instruction, it notices it begins with '01'. The following signals are now sent:

- bits 8-11 of the instruction contain the operation that needs to be performed ('add' in this case). They are sent to the ALU.
- the ALU is instructed to put its value on bus2 (using tri-state buffers)
- the registers get the following signals:
  - bus0: register r2 puts its value on this bus
  - bus1: register r3 puts its value on this bus
  - bus2: register r2 reads its value on this bus

- register PC is instructed to increment

Notice that register r2 puts its value on bus0 and reads a new value from bus2. This is possible due to the master-slave architecture of the flip-flops.

The values of r2 and r3 flow through the busses to the ALU. The ALU performs the right operation (it adds them together) and puts the result on bus2. This result is read by register r2.

When the clock now goes to 0, the sum of r2 and r3 is stored in r2 and PC is incremented.

9.4.4.3 A conditional jump (jpz)

The last instruction is 'jpz r15'. This instructs the processor to jump to another location in the code RAM (given by the value of r15), but only if the 'zero flag' is set (meaning that the result of the last calculation was zero). The instruction looks like: `1111 0000 0010 1111`.

The control logic will notice that the instruction begins with 1111. It will therefore inspect bits 4-6 from the instruction. As these bits are equal to 010, the status of the zero flag is inspected. If the zero flag is equal to 1, the following signals are sent:

- register r15 is asked to put its value on bus1
- register PC is asked to read its value from bus1
- the ALU is disconnected
- bus0 and bus2 are not connected to any registers

However, when the zero flag was equal to 0, the following signals are sent instead:

- register PC is asked to increment
- the ALU is disconnected
- bus0, bus1, and bus2 are not connected to any registers

When the clock goes to 0, register PC will contain the right value, depending on the state of the zero flag.

9.5 Exercises

9.5.1 Machine language

Convert the assembly language programs from 9.3.2 and 9.3.3 to machine language. Simulate the program in logisim.

9.5.2 Executing assembly language

a) What value will be stored at location 3 in the data RAM after executing the following code?

What are the values of registers r3 till r7 at the end?

```
0: li r3, 100
1: li r4, 10
2: li r5, 3
3: li r6, 10
4: li r7, 9
5: sub r3, r4
6: jpz r7
7: st r3, (r5)
8: jp r6
9: st r4, (r5)
A: halt
```

b) Execute the following code. What value is stored at location 10 in the data RAM when the program finishes?

Hint: there is a loop in this program. Try to figure out what it does and how many times it is executed.

```
0: 0001 0000 0101 0000 ; li r0, 5
1: 0001 0000 0011 0001 ; li r1, 3
2: 0001 0000 0000 0010 ; li r2, 0
3: 0001 0000 1101 1110 ; li r14, 13
4: 0001 0000 1010 0011 ; li r3, 10
5: 0100 0101 0010 0011 ; sub r2, r3
6: 1111 0000 0010 1110 ; jpz r14
7: 0100 0100 0010 0011 ; add r2, r3
8: 0100 0101 0000 0001 ; sub r0, r1
9: 0001 0000 0001 0011 ; li r3, 1
A: 0100 0100 0010 0011 ; add r2, r3
B: 0001 0000 0011 1110 ; li r14, 3
C: 1100 0000 0000 1110 ; jp r14
D: 0001 0000 1010 0001 ; li r1, 10
E: 1010 0000 0000 0001 ; st r0, (r1)
F: 0000 0000 0000 0000 ; halt
```

c) Execute the following code. What values are stored at locations 0 till 9 in the data RAM at the end?

```
li r0, 10
li r1, 1
li r2, 1
li r3, 0
:loop
li r4, 1
li r5, :end
sub r0, r4
jpz r5
nop r4, r1
add r4, r2
nop r1, r2
nop r2, r4
```

```
st r2, (r3)
li r4, 1
add r3, r4
li r4, :loop
jp r4
:end
halt
```

9.5.3 Write assembly language

Translate the following java code to assembly language and to machine code. Simulate the result in logisim and verify that it works.

Use the following memory locations for the variables: a: 50, b: 60, result: 70, i: 80, n: 90, q:100, r:110

You can only use 4 registers (r0 till r3).

a) calculating the average of a and b

```
short a = 6;
short b = -7;
short result = (a + b)/2;
```

b) set a bit at location a if a>1

```
short a = 1;
short result = 1;
if (a > 1) {
  result = result << a;
}
```

c) add all numbers between 0 and i

```
short i = 10;
short a = 0;
while(i != 0) {
  a = a + i;
  i--;
}
```

d) calculate the square root of a

```
short a = 125;
short b = 0;
while(a >= 0) {
  a = a - b;
  b = b + 1;
  a = a - b;
}
short result = b - 1;
```

e) calculate $a \cdot b$ very slowly

```
short a = 3;
short b = 4;
short result = 0;
for(short i=0; i<a; i++) {
  result = result + b;
}
```

f) divide n by 10. In this exercise you are (exceptionally) allowed to retain the value of q in a register until its value is final.

```
short n = -42;
n = n + ((n >> 15) & 9);
short q = (n >> 1) + (n >> 2);
q = q + (q >> 4);
q = q + (q >> 8);
q = q >> 3;
short r = n - (((q << 2) + q) << 1);
short result = q + ((r + 6) >> 4);
```

9.5.4 Booth's algorithm

The following code multiplies 2 numbers with Booth's algorithm. Translate it into assembly language.

```
short a = 142;
short b = -42;
short P_MSB = 0;
short P_LSB = b;
short Pextra_LSB = 0;
for(int i=0; i<16; i++) {
    short lowestBit = P_LSB & 1;
    if ((lowestBit==0) && (Pextra_LSB==1)) {
        P_MSB += a;
    }
    if ((lowestBit==1) && (Pextra_LSB==0)) {
        P_MSB += -a;
    }
    Pextra_LSB = lowestBit;
    lowestBit = P_MSB & 1;
    P_MSB = P_MSB >> 1;
    P_LSB = P_LSB >>> 1;
    if (lowestBit == 1) {
        P_LSB = P_LSB | 0x8000;
    }
}
```

Put the variables at the following locations in the data RAM:

name	location
a	23
b	24
P_MSB	25
P_LSB	26
Pextra_LSB	27
i	28
lowestBit	29

9.5.5 Arrays in assembly

One can also use arrays in assembly language. Arrays are stored in consecutive locations in memory. To access a certain value, one has to calculate the address where it is stored by adding the address of the first element with the index.

Translate the following program to assembly (put variable i on address 7 and let the array 'fib' start at location 10).

```
short[] fib = new short[100];
fib[0] = 1;
fib[1] = 1;
for(short i = 2; i<100; i++) {
    fib[i] = fib[i-1]+fib[i-2];
}
```

9.5.6 Multiplication

Write a program in assembly that multiplies the values in r0 and r1 using the long multiplication algorithm. The result should be stored in r3. You can use other registers for intermediate results.

You can assume that the numbers in r0 and r1 are maximum 8 bits long. The registers will then be large enough to hold the values of A, B, and P.

Advanced:

- Can you change the code so r0 and r1 can hold bigger numbers? Store the result in two registers: r3 (MSB) and r4 (LSB).
- Can you also implement Booth's algorithm directly in assembly?

9.5.7 Video output

In a computer video output is generated by a separate chip that shares a part of the data RAM with the processor.

Imagine this chip can only produce text on the screen. It is able to show 80 columns and 25 rows on the screen. Each cell can contain a UCS-2 character. Imagine now that the data to produce this image is stored in the data RAM from location 0 till 1999. The first 80 locations are the first row, then the next 80 locations make up the second row, and so on.

If the number 65 is stored at location 85, this means that the character 'A' will appear on the second line, sixth column.

Now write the following programs directly in assembly language:

1. a program that clears the screen (fills it up with spaces)
2. a program that shows 'Hello, world!' on the top left of the screen
3. a program that can put a character (stored in r0) at a location given by r1 (column number from 0 till 79) and r2 (row from 0 till 24)

9.5.8 Subroutines

Imagine you want to be able to reuse the code of the third step of the last question as a subroutine that can be called multiple times.

The code is stored at a specific location. It should be possible to jump to this code and then jump back to where it came from. How could you write code like this?

Try to make a program that puts you own name on the screen starting at location (10, 10). Reuse as much code as possible.

9.5.9 Inner workings of the Moncky-1

Try to figure out how the following instructions will be processed by the control logic and which signals it will send to all components:

- `0001 0111 1000 0101` ; li r5, 120
- `1010 0000 0100 0101` ; st r4, (r5)
- `1100 0000 0000 1110` ; jp r14

9.5.10 Add an instruction

Take a close look at the Moncky-1 simulation in logisim. can you add a 'inc' instruction with machine code:

0010 xxxx xxxx rrrr

This instruction will add 1 to register rrrr.

Chapter 10

Practical processors

Although the Moncky-1 processor could be used in a real computer (after adding interrupts), it has some drawbacks. The emphasis of this processor is more educational then practical use.

In this chapter some techniques are discussed that are used in real modern processors. There will be many references to processors in which those techniques were introduced. This chapter will not focus on logic circuits any more, but rather on design choices on a higher level.

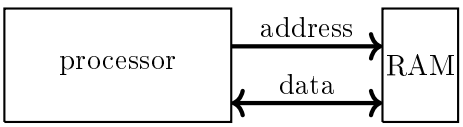
10.1 The Von Neumann architecture

One of the problems using the Moncky-1 processor in a personal computer is that its code needs to be already loaded in the code RAM. The processor cannot write to the code RAM. So when a new program must be loaded and started, the processor needs to be shut down, the code needs to be loaded into the code RAM, and then the processor needs to be restarted. This is okay in embedded systems like Arduino (using an ATmega328 processor) or routers, camera's, ... where there is no need to load programs on the fly. But for a personal computer this can be a problem.

The main problem is that there are two memories: data RAM and code RAM. This architecture is called 'Harvard architecture'. If there would only be one memory, the processor could write data into memory and then execute it as code. Code and data are in fact not so different, as we already discussed before (in the case of the Moncky-1 data and code are all 16 bit integers).

Being able to load data and execute it, enables a lot of features. For instance, the 'operating system' (like Linux, Mac OSX or Windows) can be loaded when the computer boots up. After this, programs like word processors, browsers, ... can be started and stopped at run time.

John Von Neumann also noticed that code and data are very alike. He proposed to make a processor that is connected to one RAM. This RAM contains both data and code. The architecture looks like this:



This architecture also enables so called 'self-modifying code'. This is a program that overwrites itself in memory. This is not used a lot any more, but in the 80's it was a very popular technique to obfuscate the workings of code. It made it very hard for hackers to know what the code was doing.

Another advantage of this architecture is that the processor van execute java code. The 'Java Virtual Machine' (JVM) is a program that can execute bytecode. The bytecode is stored in the RAM. The JVM will translate the bytecode to machine code and store that in the RAM. Then the code is executed.

This all seems to be good to be true and the reader might wonder why there are still processors using the Harvard architecture. The truth is that the Von Neumann architecture also has its drawbacks. An example might clarify this. Imagine the processor needs to load data from memory into a register. The instruction to do this, is also stored in memory. This means that the processor needs to read the instruction first and then read the value from memory. These two steps cannot be executed in the same clock cycle as there is only one connection to memory. One needs more then one clock cycle to execute certain instructions. This means that processors using a Von Neumann architecture are slower.

Most Von Neumann processors need at least 3 clock cycles to process any instruction. During the first clock cycle the instruction is read from memory (this is called 'fetch' stage). A next clock cycle will trigger the control logic to decode the instruction and decide what to do (this is called 'decode' stage). During the next clock cycles the instruction is executed ('execute' stage). These stages are present in almost all processors using the Von Neumann architecture.

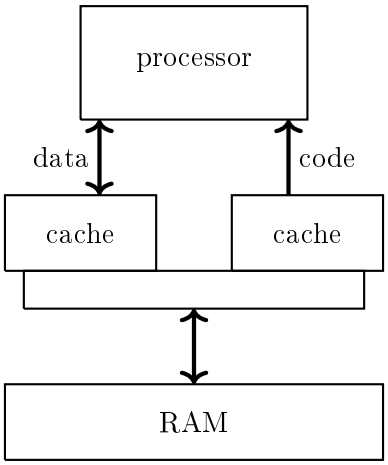
This has enormous implications on the design of the processor. It will need to have circuitry to know at which stage it is during any given clock cycle. This means that the processor is executing a kind of program for each instruction. This program is implemented in hardware and is sometimes referred to as 'micro code'. A lot of extra registers are needed in order to hold the current 'state' of the processor.

Processors using the Von Neumann architecture are more complex, but in many cases they can be made using less transistors. This is because one can execute instructions in more steps. A 16-bit addition for instance, can be executed using a 4-bit adder. The adder will first add the two least significant nibbles, then the next, and so on. It will need 4 clock cycles to do the addition, but the hardware itself is smaller.

10.2 The Modified Harvard architecture

The Von Neumann architecture has the advantage of having only one memory, but it slows down the processor because it can no longer read code and read/write data at the same time. In order to overcome these problems hybrid architecture was developed: the 'modified Harvard architecture'. It was first introduced in the Pentium II processor.

The idea is to create a processor using the Harvard architecture, but to connect both RAM connections to the same RAM, using cache memory. Schematically this looks as follows:



The cache memories are very small and are sometimes called ‘level-2 cache’ or ‘L2 cache’. They contain a copy of a small portion of the RAM and they will refresh automatically when needed. The processor does not know this. It thinks it is connected to 2 RAM’s. When the caches have to synchronise the processor is halted until the operation is done. This will still slow down the processor, but on average it will be faster then the Von Neumann alternative.

In the Pentium II the caches where separate chips. The processor was sold on a small board that also contained the caches. This can be seen in the picture below:

The two chips on the left and right are the caches for data and code. In modern processors these caches are integrated with the processor.

10.3 Registers

The Moncky-1 processor has 16 registers of 16 bits. This varies enormously amongst processors. Some processors have very few registers. An example is the MOS 6502 processor which was one of the first processors to be used for personal computers (1975). This processor only contains 6 registers. Therefore, less transistors were needed to make this processor, making it much smaller and cheaper.

The registers of the Moncky-1 are general purpose, meaning that one can use any register for any operation. In other processors this is not always the case. In the MOS 6502 for instance, every register is used for a specific goal. If one does a calculation, the result is always stored in the same register. Addresses can only be stored in certain registers.

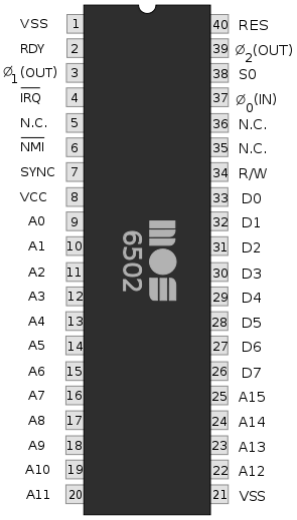
Sometimes registers get a name in stead of a number. In the Zilog Z80 processor registers are called ‘a’, ‘bc’, ‘de’, ‘hl’, ... This can make the assembly language more readable.

If there are fewer registers, it becomes more clear that they cannot be used to hold data for a long time. Variables will therefore always be stored in RAM. The registers will only be used temporarily.

Registers can hold a wide variety of bits. The ATmega328 for instance, contains 32 registers of only 8 bits long. If one needs to do calculations with more bits, multiple steps are needed in order to do that.

10.4 Busses

The processor is connected to the RAM. These connections can be seen in the the pinout of a processor. The MOS 6502 for instance, has the following connections:

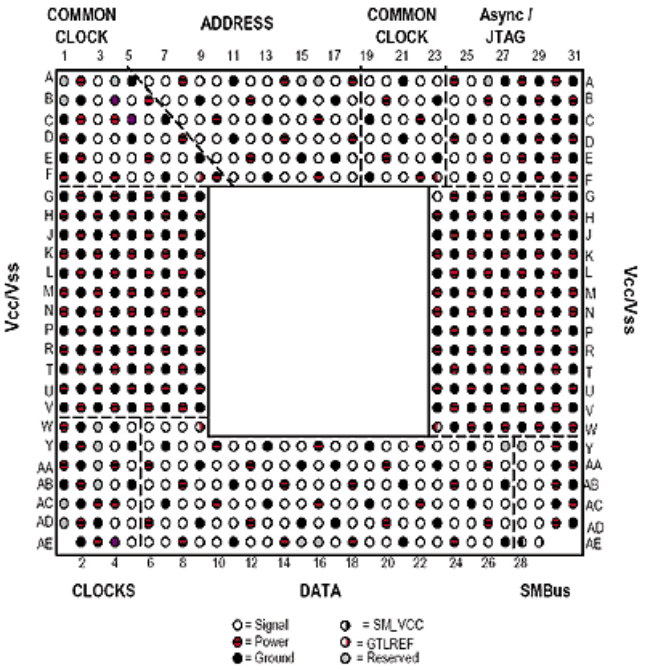


The connections A0 till A15 are connected to the 'address bus' of the RAM. They determine the address (or location) in memory where the processor wants to read or write a value. If the processor needs to read a value, the RAM will put the value on pins D0 till D7. This is called the 'data bus'. When the processor needs to write a value to the RAM, it uses the same data bus. Tri-state buffers are used so there is no conflict on the data bus. The value of pin 34 (r/w) will determine if the processor wants to read and write and will drive the tri-state buffers. This pin is part of the 'control bus'.

This processor thus has an address bus of 16 bits wide and a data bus of 8 bits. Since the address bus has 16 bits, a maximum of 65536 address can be accessed. The RAM memory can be no larger than 64 KiB.

The size of the data bus determines the number of bits that can be read or written in one cycle. A larger data bus will therefore make the processor faster.

The pins for addresses and data can always be seen on any processor. For instance, the pinout of an Intel Xeon processor looks like this (it has much more pins):



Here you can clearly see the pins for 'ADDRESS' and 'DATA'. In this case the address bus is 36 bits wide and the data bus is 64 bits wide. The processes can therefore read and write 8 bytes in one cycle. Can you determine the maximum RAM memory that can be accessed like this?

Modern processors even have more pins (more than 1000), but address and data pins are always identifiable.

10.5 Little and Big endian

During a long time period data busses were 8 bits wide. Even if the internal registers were larger, the processor could only write one byte at a time to memory. For example, a 32 bit register could contain the value 0xF007BA11. If it needs to be written to memory, it will need four consecutive bytes. Imagine the addresses to store the value are 123, 124, 125, and 126 in memory. Which byte is then written to which position?

One could choose to write the MSB (0xF0) to location 123, then 0x07 to location 124, 0xBA to location 125, and finally the LSB (0x11) to location 126. This strategy is called 'big endian'. Big endian processors will always write the MSB at the first memory location.

But one could also start with the LSB. In this case 0x11 is stored at 123, 0xBA is stored at 124, 0x07 is stored at 125, and 0xF0 is stored at 126. Processors that do this are called 'little endian'.

In principle it does not matter whether a processor uses little endian or big endian notation. But when computers are connected to each other, they need to send information to each other. Then it becomes vital to know how the information is stored in memory and which byte needs to be sent first.

Most processors these days are little endian, but old processors like the Motorola 68000 use big endian. This caused incompatibilities between Apple and IBM computers in the eighties.

10.6 The stack pointer

The stack pointer is a special purpose register that is present in most processors. It is given the name 'sp' and contains an address in memory. It is used to enable jumping to a subroutine, having local variables and returning back to the original code.

A 'subroutine' (e.g. a method in java or a function in C) is a piece of code in the code RAM. When the processor wants to execute a subroutine (we say the processor 'calls' the subroutine), it needs to jump to the location where the subroutine starts. But after executing that code, it needs to return to the 'caller'. Therefore, a 'return address' needs to be stored somewhere. One could use a register to hold the value of the return address, but the subroutine might also call another subroutine, therefore overwriting the value.

In order to overcome this problem, a data structure known as a 'stack' is used. The return address is stored in RAM at the location where 'sp' is pointing to. Then 'sp' is decremented. If another subroutine needs to be called, the next return address is written in RAM and 'sp' is decreased again. One says that the return address is 'pushed' onto the stack.

When returning from a subroutine, 'sp' is first increased and the return address is read at that location. The return address is 'popped' from the stack.

The stack can also be used to save the contents of registers or to hold local variables. A stack is a data structure that can also be used for other purposes, but if it is used in this context, it is also referred to as 'call stack'.

In order to clarify the procedure of calling a subroutine, using a call stack, the following example is presented. Imagine that the stack pointer has value 123 and the program counter is currently at 42. The processor will read the instruction at location 42. This instruction is 'call 12' (not implemented in the Moncky-1, but it can be programmed using existing instructions). This instruction indicates that the subroutine at location 12 needs to be executed. The processor will do the following:

1. the value 43 (pc+1) is saved in RAM at location 123 (sp)
2. the value of sp is decremented (sp- -), so it now contains the value 122
3. the processor jumps to the subroutine at location 12 (pc=12)
4. the processor executes all instructions from address 12 until a return is needed
5. at a certain location a 'return' instruction is encountered
6. the processor will increment sp (sp++), so it is back to 123
7. the processor reads the value at location sp (123) and puts it in pc (pc=43)
8. as a consequence, the processor has jumped back to the original code

When calling a method in java or any other programming language will always initiate this kind of procedure.

Most processors have the following instructions available:

- 'call' address: store pc+1 on the stack and jump to the address
- 'ret': get the return address from the stack and jump to it
- 'push' register: put the contents of a register to the stack
- 'pop' register: get the value of a register from the stack

If we would add these instructions to the Moncky-1 processor, it would be possible to write code as follows:

```
104A  li r0, 5     ; r0=5
104B  li r1, 6     ; r1=6
104C  li r2, 0x20  ; address of procedure
104D  call r2      ; calculate r0=r0*r1
104E  li r1, 4     ; r1=4
104F  call r2      ; calculate r0=r0*r1
1050  li r1, 100   ; address in memory
1051  st r0, (r1)  ; store the result at place 100
1052  halt
```

This code puts values 5 and 6 in registers r0 and r1. Next, it calls a subroutine at location 0x20. This subroutine will multiply both registers and puts the result in r0. The code of the subroutine looks like:

```
; r0 = r0 * r1 (slow algorithm)
20  push r2      ; save r2
21  push r3      ; save r3
22  push r4      ; save r4
23  push r5      ; save r5
24  xor r2, r2   ; r2=0
25  li r3, 0x28  ; address of loop
26  li r4, 1     ; constant 1 in r4
27  li r5, 0x2D  ; exit loop
28  nop r1, r1   ; compare r1 to itself
29  jpz r5       ; if r1=0, exit loop
2A  add r2, r0   ; add r0 to r2
2B  sub r1, r4   ; decrease r1 by 1
2C  jp r3        ; go back to loop
2D  nop r0, r2   ; copy result to r0
2E  pop r5       ; restore r5
2F  pop r4       ; restore r4
30  pop r3       ; restore r3
31  pop r2       ; restore r2
32  ret          ; return to caller
```

As you can see the values of r2 till r5 are saved on the stack in the beginning and restored at the end.

Although the Moncky-1 processor does not have these instructions, they can easily be programmed using the existing set of instructions (this is left to the reader as an exercise).

10.7 Interrupts

Almost all processors have at least one input pin for 'interrupts'. They are very frequently names 'IRQ' (interrupt request). Interrupts are very important when the processor needs to interact with other devices like a keyboard, mouse, hard disk, network card, ... For example, when a user presses a key on a keyboard, the processor is probably busy doing other tasks. In order for the processor to register the key press, it needs to be interrupted. When the processor receives an interrupt, it will finish the current instruction and then call a specific subroutine (the 'interrupt handler'). In this code the key press is processed. After that, the processor jumps back to the code it was executing (much like discussed before).

Interrupts are used to get input from all devices within the computer. Every time a key is pressed, the mouse is moved, a network package arrives, ... an interrupt is generated. This slows the computer down, so the interrupt handler must be a piece of code that executes very fast.

10.8 CISC and RISC processors

Most processors need multiple clock cycles to execute an instruction. This can vary from instruction to instruction, but on average this is about 2 to 7 clock cycles. This means a processor can also do more complicated operations in one instruction. Processors that have a lot of complicated instructions are called 'CISC' processors (Complex Instruction Set Computer). For instance, a processor might have an instruction that increments a value in RAM. The processor will load the value, increment it, and write the result back. In the Moncky-1 processor this will need at

least 3 instructions (and hence also 3 clock cycles). In a CISC processor this could be done in one instruction, but the number of clock cycles will depend on the processor).

As the Moncky-1 processor only has very few and simple instructions, it is called a 'RISC' (Reduced Instruction Set Computer). A RISC might seem slower than a CISC because most operations need multiple instructions, but a CISC processor might need a lot of clock cycles to do one instruction. In practice RISC processors are considered faster than CISC, but this also depends on the internal architecture.

CISC processors have instructions that can typically do common tasks as:

- increase or decrease a variable in RAM
- call or return from a subroutine (using the stack pointer)
- copying a block of memory to another location
- multiply two numbers and add the result to a third number (multiply-accumulate)
- perform a floating-point calculation
- ...

The Zilog Z80 is an example of a CISC processor. A nice example is the LDIR instruction. This can be used to copy a block of code from one location to another. The following code will copy 50 bytes from location 100 to location 200 ('hl', 'de', and 'bc' are names of registers):

```
ld hl, 100  ; 10 clock cycles
ld de, 200  ; 10 clock cycles
ld bc, 50   ; 10 clock cycles
ldir        ; 1066 clock cycles
```

The last instruction does all the work and needs 1066 clock cycles to copy all the data. Remark that even the simple instructions need a lot of clock cycles on this processor.

If one would write the same functionality without this instruction, it would look like this:

```
      ld hl, 100
      ld de, 200
      ld bc, 50
loop: ld a, (hl)  ; 7 clock cycles
      ld (de), a  ; 7 clock cycles
      inc hl      ; 6 clock cycles
      inc de      ; 6 clock cycles
      dec bc      ; 6 clock cycles
      jpnz loop   ; 10 clock cycles
```

This code will need 2100 clock cycles to complete the loop. So the benefit of using the ldir instruction is great.

On the Moncky-1 processor however, this would look like this:

```
        li r0, 100
        li r1, 200
        li r2, 50
        li r3, 1
        li r4, :loop
:loop   ld r5, (r0)
        st r5, (r1)
        add r0, r3
        add r1, r3
        sub r2, r3
        jpnz r4
```

The loop will now only take 300 clock cycles to execute!

So why did they ever make CISC processors? The answer is: because they were cheaper to produce. CISC processors generally have less transistors which makes them smaller and thus cheaper to make. When transistors got smaller and it became apparent that RISC processors were better, it was not possible to abandon CISC architecture because the new processors needed to be compatible with the previous ones. Especially Intel had this problem as their processors were ubiquitous. ARM (Advanced RISC machine) and MIPS processors were designed as RISC processors from the start, but their incompatibility with Intel made them unpopular in the beginning. Intel introduced RISC in the Pentium 4 but they put a lot of logic around it so it could also behave as a CISC. This logic translates the complex instructions to the more simple instructions of the core.

10.9 Instruction length

In the Monkcy-1 processor all instructions have equal length: 16 bits. In other processors this is not always the case. Instructions can vary from 1 byte to dozens of bytes. A processor with an 8 bit data bus can only read 1 byte per cycle so even reading an instruction can take some time.

An instruction consists of an 'opcode' and 'operands'. The opcode tells the processor what to do and the operands are the parameters for that instruction. In the Moncky-1 the opcode is always contained in the 4 most significant bits of the instruction. The operands are stored in the other bits.

In the Zilog Z80 this is different. For example, the instruction 'ld hl, (0x0100)' loads the values at location 0x0100 and 0x0101 in RAM to the 16 bit register 'hl'. This instruction is 3 bytes long: 0x2A, 0x00, 0x01 . The first byte is the opcode and the next two bytes are the operand (in this case the number 0x0100 in little endian). The processor needs at least 3 clock cycles to read this instruction. In total it needs 16 clock cycles to execute the whole instruction.

10.10 Coprocessors

When processors started to be made transistors were still very big. Since the price of a processor is directly proportional to its area, it was vital to limit the number of transistors. For this reason floating-point operations could not be implemented in hardware. All floating-point calculations had to be done in software.

In order to add floating-point arithmetic to a processor a coprocessor was developed. This is a chip that is designed to only do floating-point calculations. It can be driven by another processor.

An example of this is the Intel 8087 coprocessor which was designed to work with the Intel 8086 processor. When present, it could be used to do the calculations which enhanced the speed significantly.

When transistors became small enough, the coprocessor was integrated with the processor. The Intel 80486 is an example of this.

This happened a lot in the past: first multiple chips are placed on a motherboard to work together and later they are integrated on 1 chip. Current chips can contain almost a whole computer: multiple processors, memory, graphics, interrupt controller, memory controller, ...

10.11 Multiprocessing

The Moncky-1 processor can execute a program, but sometimes one needs to execute multiple programs at the same time (a browser, a word processor, a collaboration tool, ...). All these programs are loaded into memory.

As the processor can only do one task at the same time, a technique called 'time slicing' is used to seamlessly execute multiple programs at the same time. The idea is that the processor is allowed to execute part of a task after which it is interrupted. The interrupt handler will then choose the next program to run. Like this it seams that programs run simultaneously. This strategy is also called 'round robin scheduling'.

In order to allow this, the processor must have interrupts. Most processors also contain special instructions that make the switches to other programs faster.

10.11.1 Different modes

When several programs are executed, they are not allowed to influence each other. As they all share the same memory, programs could easily access each others data or even code. This could allow hackers to make programs that will change other programs. Even without hackers, the system could become very unstable or even crash when there are bugs in programs.

In order to overcome this problem, the processor can be put in different modes. In some modes the processor can execute all instructions. In other modes the processor can only execute certain instructions.

Also, the processor is restricted to access certain parts of memory when it operates in certain modes. Most processors have at least two modes: 'kernel' mode and 'user' mode.

10.11.2 Hyperthreading

The time slicing technique introduced before has an important disadvantage. Every time the processor needs to switch between tasks, the processor needs to save its state (i.e. the contents of all registers) to memory and load the state of another task. This takes a lot of time.

A processor with hyperthreading capability has multiple sets of registers. In this way it just needs to switch the registers when it needs to execute another task. The processor behaves like it has multiple cores although it just constantly switches between programs.

Intel introduced hyperthreading in the Pentium 4.

10.11.3 Virtualisation

Sometimes it is necessary to simulate complete computers. These are called 'virtual machines' and they behave as separate computers. This enables to run a complete operating system next to the 'host operating system'. One can also simulate a computer with another processor enabling to run Intel programs on an ARM processor for instance.

Simulating a complete computer is a very computational intensive task. The computer must read the machine code, alter it so it works on the host machine, and then execute it. This makes virtual machines very slow when they are implemented in software.

To accelerate this, extra instructions are added to the processor. Intel calls these 'VT-x' instructions and AMD calls them AMD-V. With these instructions enabled, the processor can run a virtual machine almost at the same speed as a native machine.

10.12 SIMD instructions

In the early nineties, graphics and audio became more and more important. Computers started to be used to process images, to watch video's and to do audio compression. These operations require an incredible amount of calculations. However, these calculations can frequently be done in parallel. For instance, if one wants to increase the brightness of a picture by 50%, all values of the pixels need to be multiplied by 1.5. The Moncky-1 processor would have to read the value of every pixel, multiply it and store it back to memory. It would be more interesting if there would be multiple ALU's in the processor that can perform the same operation on different pixels at the same time.

Modern processors do have more ALU's and special instructions to use them. These instructions are called 'SIMD' instructions (Single Instruction, Multiple data). They operate on a specific set of registers. The data is first loaded in the registers and then the SIMD instruction is used to perform an operation on all registers.

The first Intel processor containing these instructions was the Pentium (the instructions were called MMX (MultiMedia eXtention)). In the Pentium the registers could only contain integers, which made them unsuitable to do calculations for 3D graphics. In the Pentium III the SIMD instructions could also handle floating-point numbers.

Today, specific processors are made for graphics. They contain massive parallel ALU's that can do thousands of operations in parallel. They can be integrated with the processor, or be put on a separate graphics card that communicates with the processor through a high-speed bus.

10.13 Bugs in hardware

Bugs are little errors that occur a lot in software. They cause programs to crash or to deliver wrong results. In software this is not catastrophic as they can be corrected easily. By updating the software the bugs can be removed.

But with hardware this is a very serious problem. Any little mistake in the circuitry of a processor (containing billions of transistors!) will render it useless. Intel has experienced this when they developed the Pentium processor. There was a bug in the floating-point unit when two numbers were divided. In some cases the result was not correct. But the production of the Pentium had already begun and millions of units were already sold before the bug became apparent. In order to correct the error, Intel would have had to design a new chip and start all over. This was not feasible because the cost was too high. Fortunately they found a solution: every time the

software needed to do a division, it needed to check and correct the error in the software. This means that the performance of floating-point divisions was reduced but the cost to replace the processor was avoided.

Developing a processor containing billions of transistors therefore takes a tremendous effort on testing. Millions of tests are run on a simulation before production starts. These tests are even made before the final design is done. When high quality is needed in software, this is also done: tests are written before the code. This is called 'test-driven development' (TDD).

10.14 Caching

The most frequently used memory is not SRAM but DRAM. DRAM does not work with latches (as seen before), but with capacitors. A capacitor can hold a little bit of electric energy for a short time. They are very small and very cheap to make. But since they loose their electric charge, they need to be refreshed all the time. This is done by reading all values of the memory and storing them again on a very regular base. This makes DRAM much slower as the processor cannot access the memory while it is refreshing. Also, accessing DRAM is not done in one clock cycle, making them even slower.

In order to overcome the slow DRAM chips the processor speed and the RAM speed are disconnected. The processor will run at a much higher speed than the RAM. This means however, that the processor will have to wait from time to time for data to be ready at the data bus.

A solution for this is to use a 'level-1 cache' or 'L1-cache'. This is a bit of very fast memory that is put between the DRAM and the processor (around 8 to 64 KiB). The cache is in fact SRAM which responds quicker and it will contain a copy of certain parts of the DRAM. The caches can update their contents and read ahead so the processor can run at full speed most of the time. Of course, the processor will have to wait from time to time, but the overall performance is much better.

It is important to notice that the clock speed of the processor is no longer a good measure of the total performance of a computer. The speed of the memory (connected to the 'front-side bus') and the speed of the motherboards buses are equally important.

10.15 Power management

Increasing the speed of a processor has a big disadvantage: the processor becomes very hot and needs to be cooled. In laptops this is a huge problem. There is need for good cooling but the power dissipation of the processor should also be regulated.

Modern processors have circuits to limit power dissipation. In the Moncky-1 processor one can see that the ALU will do a calculation at each clock cycle, even if the instruction doesn't need it. It could be shut down for those instructions, resulting in lower power requirements. In modern processors all components that are not used are shut down for a brief period of time.

Another trick that is used a lot is to reduce the clock speed when there is no need for a lot of activity. A personal computer has to wait most of the time until an event happens. This can easily be 98% of the time (this is called 'idle time'). During the idle time the clock speed is reduced, resulting in less power needed.

10.16 Pipelining

Pipelining is a technique to increase the clock speed of a processor. The clock speed is determined by the slowest operation. In the Moncky-1 for instance, the clock speed is determined by the speed of a 'sub' instruction. In order to perform it, two registers need to be selected, the ALU must do its calculation (the carry has to ripple through) and the flip-flops of the receiving registers must stabilise. All the other operations need less time, so they can all do their work in one clock cycle.

This could be made faster by dividing the instructions is many small parts that can be executed faster. For instance, one could put a 4 bit ALU in the processor and use it 4 times to do a 16 bit calculation. Of course, the total time to execute the instruction will still be the same. But the trick is that while the calculation is going on, one could start reading the next instruction. This is called 'pipelining'.

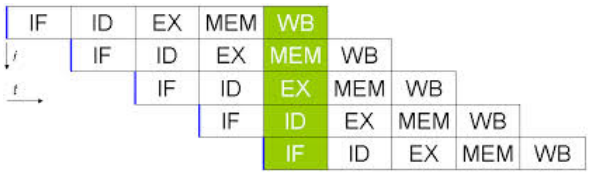
Pipelined processors perform instructions in at least 5 steps:

1. Instruction Fetch (IF): read the instruction from RAM
2. Instruction Decode (ID): convert the instruction to internal signals in the processor
3. Execution (EX): execute the instruction (e.g. do a calculation)
4. Memory access (MEM): read the necessary data from RAM (if needed)
5. Write Back (WB): write the result to RAM or to a register

The processor will always go through these steps for each instruction. With pipelining the steps can be done in parallel as follows:

1. read the first instruction (IF)
2. decode the first instruction (ID) and read the second (IF)
3. execute the first instruction (EX), decode the second (ID) and read the third (IF)
4. ...

Schematically the following happens (http://www.wikiwand.com/simple/Instruction_pipelining):



It will still take 5 clock cycles to perform an instruction, but other instructions are executed almost simultaneously. The clock speed can be increased because every step is fast enough.

Pipelining doesn't come without any drawbacks. At first sight, 5 instructions are executed in parallel as soon as the pipeline is at full speeds. But this is not always possible. Let's take a look at the following code (we use Moncky-1 assembly):

```
        li r0, 123
        ld r1, (r0)
        li r2, 7
        or r1, r1
        jpz :next
        li r1, 5
:next   sub r1, r2
        ...
```

At the first clock cycle the first instruction will be read. At the second clock cycle the second instruction is read. But the second instruction depends on the result of the first. This means the processor must wait until the first instruction is completely executed (lasting 3 more clock cycles). This can be accomplished in two ways.

The first way to solve this, is to change the compiler so it rearranges the instructions and introduces 'NOP' (No OPeration) instructions. These instructions don't do anything but allow the pipeline to advance a step. The program could be adjusted to:

```
        li r0, 123
        li r2, 7
        nop
        nop
        ld r1, (r0)
        nop
        nop
        nop
        or r1, r1
        nop
        nop
        nop
        jpz :next
        li r1, 5
        nop
        nop
        nop
:next   sub r1, r2
        ...
```

A second possibility is to change the processor so it can detect dependencies between instructions. It will automatically wait until the previous instruction is done. The consequence is that the hardware of the processor becomes much more complicated.

Another optimisation can be done for the 'jpz' instruction. In the above example the jump only needs to be performed if the result of the 'or r1, r1' instruction is zero. The processor could guess that the result is not zero and start to execute the following instructions. When the result is zero, it can backtrack and empty the pipeline again. This is called 'branch prediction'. Although it makes the processor very complex this technique is used a lot in many processors.

Pipelines can become very long. In the Pentium 4 the pipeline contains 20 stages!

10.17 Multi-core processors

The clock speed of processors was increased a lot during the years, but then it started to stagnate. This is because the physical limits are almost reached. The signals cannot flow through the processor at speeds greater than the speed of light. This means that a clock pulse of a 3 GHz clock has a length of 10 cm. If a path within the processor is longer than 10 cm, the clock pulse will have another value at both ends of the path. Because transistors also need time to change their state, this length is reduced even more. This puts a constraint on the maximum size of a chip.

Increasing the clock speed also results in more heat produced by the processor. This is because the gates in the processor will use more energy when they change state.

Because clock speeds cannot be increased any longer, another technique is used to make them faster: parallelism. Four, eight or even 16 processors (or 'cores') are put on one chip. These processors will run together to theoretically they can do 4, 8 or 16 times the work of one processor. Of course, this is only theory; there is a lot of overhead because processors need to share memory and sometimes need to wait for results of another one. The operating system must be adapted to make use of the cores in an efficient way.

Multi-core processors will have extra cache memory between the RAM and shared with all the cores. This is called 'level-3 cache' or 'L3-cache'.

10.18 Evolution

In the eighties processors evolved enormously. The number of transistors on a chip doubled about each 1.5 years. This is called 'Moore's law'.

This doubling speed is slowing down a bit. Since 2009 it is not possible any more to do a lot of architectural changes for every processor generation. The difference between two generations of processors is therefore not that big any more.

This is why the generation of a processor is not always part of the computer description any more. Nevertheless this is also important information. The table below shows the generations of the Intel Core processors:

generation	name
1	Nehalem
2	Sandy Bridge
3	Ivy Bridge
4	Haswell
5	Broadwell
6	Skylake
7	Kaby Lake
8	Coffee Lake
...	...

Each of these generations has at least the following variants: i3, i5, and i7. The difference between these is the number of cores and the ability to do hyperthreading.

10.19 Exercises

10.19.1 Practical processors

Complete the following table (you can find the information on the Internet):

processor	year	#transistors	# registers	#bits/register	address bus	data bus	max RAM
MOS 6502							
Z80							
8086							
80386							
80486							
Pentium							
Pentium II							
ARM7							
MIPS							

10.19.2 Moore's law

Make a plot of the number of transistors as a function of the year, using the numbers of the previous exercise. Draw the theoretical line over it using Moore's law. Do they match? What does this tell us about the future?

10.19.3 De 80x86

All Intel processors are still compatible with the first 8086 that was developed in the previous century. When you turn on your computer, the processor behaves like a 8086. When the operating system is loaded, the other features are enabled.

So it is interesting to take a look at the properties of this ancient processor. Find the following information on the Internet:

1. How many registers are there and what are their names?
2. What is the register 'IP' used for?
3. What is the function of registers 'SP', 'SI', and 'DI'?
4. What is the function of registers 'CS', 'DS', 'ES', and 'SS'?
5. What do the instructions 'push' and 'pop' do?
6. Take a look at the example code on http://en.wikipedia.org/wiki/Intel_8086. Can you understand what it does?

10.19.4 Calculating with 8 bits

Imaging a processor has the following instructions:

- li r1, i: load register r1 with value i
- add r1, r2: add the values of r1 and r2 and put the result in r1

- adc r1, r2: add the values of r1 and r2. If the carry bit is equal to 1, then add another 1 to the result. Put the result in r1

Imagine there are 16 registers but they are only 8 bits long. Registers r0 and r1 contain a 16-bit number (r0 is the LSB and r1 is the MSB).

Registers r2 and r3 also contain a 16-bit number.

Can you write code to load the numbers 300 and 400 and add both 16-bit numbers using the previous instructions?

10.19.5 Assembly language

Every processor has a different assembly and machine language. This is because those languages are very closely linked with the hardware which is very specific to a processor. Here some assembly programs are shown for different processors. Use the Internet to find information on the instruction sets and try to figure out what the following code does:

MOS 6502:

```
LDA 0x0100
LDX 0x0101
INX
ADC 5, X
STA 0x0102
```

Z80:

```
     ld hl, 2045
     ld a, (hl)
     inc a
     ld (hl), a
     call foo
     halt
foo: ld bc, 2046
     ld a, (bc)
     dec a
     ld (bc), a
     ret
```

Index