

# 2 Number Systems

## Student Group

First Name	Surname	Matrikel Nr.

## Table of Contents

- 2. Number Systems** ..... 3
- 2.1 Types of Number Systems** ..... 3
- 2.1.1 Addition Systems ..... 3
- 2.1.2 Decimal and Binary ..... 4
- 2.1.3 Other bases ..... 6
- Hexadecimal ..... 6
- Octal ..... 8
- 2.1.4 From decimal to other bases ..... 8
- Fast Approach: Try and Subtract ..... 8
- Works everytime: Repeated Division / Multiplication ..... 8
- 1. handling the integer decimal ..... 8
- 2. handling the decimal places ..... 9
- 2.1.5 Binary Coded Decimals ..... 11
- 2.1.6 Marking the base of a numeral ..... 11
- When to use which base? ..... 11
- 2.2 basic arithmetic operations in binary and hexadecimal** ..... 12
- 2.2.1 Addition ..... 12
- In Binary ..... 12
- In Hexadecimal ..... 13
- 2.2.2 Subtraction ..... 14
- In Binary ..... 14
- In Hexadecimal ..... 14
- Note! ..... 15
- 2.2.3 Multiplication and Division ..... 15
- related Links** ..... 16
- Exercises ..... 16
- Exercise 2.3.1. power tables ..... 16
- Exercise 2.3.2. Conversion to Binary ..... 19
- Exercise 2.3.3. Conversion to Hexadecimal and Decimal ..... 20

---

Exercise 2.3.4. Conversion to Hexadecimal and Binary .....	20
Exercise 2.3.5. Addition and Subtraction .....	20
Exercise 2.3.6. Multiplication and Division .....	20
Exercise 2.3.7. Operation in Hex .....	21
Exercise 2.3.8. Further Questions .....	21
Exercise 2.3.9. One's Complement .....	21

# 2. Number Systems

## 2.1 Types of Number Systems

In the previous chapter we had a look onto the way a processor (and a computer) can deal with the digital values '\$0\$' and '\$1\$'. However, we haven't seen how the processor can handle larger numbers. In order to approach this, first a short historical outline is shown.

### 2.1.1 Addition Systems

The first used number system were **addition systems**. These are also still in use: when enjoying a German beer in the beer garden the waiter is counting the 'progress' by dropping dashes onto the coaster (see [figure 1](#))

Fig. 1: number of drinks on a German Bierdeckel (coaster)



In ancient Rome these systems were deeper elaborated. There were different symbols which represent numbers of various sizes:

- \$I = 1\$
- \$V = 5\$
- \$X = 10\$
- \$L = 50\$
- \$C = 100\$
- \$D = 500\$
- \$M = 1000\$

Besides this representation for quantities also the position of the symbol in the **numeral** were important:

- In general: the letters have to be arranged decreasing from left to right. For example  $\text{MDCI} = 1601$
- There are deviations of this rule: When up to three of the a lower symbol is written to the left, these have to be subtracted. Sounds complicated?  
A kind of.... for example  $\text{MCCD} = 1354$ .  
Luckily, we do not have to learn this, but by this trick the length of the numeral could be shortened,

It becomes even more complicated, when trying to calculate with the numbers: what is the result of the multiplication  $\text{CCMXXXVII} \cdot \text{DDIIX}$ ?

## 2.1.2 Decimal and Binary

Luckily, we have nowadays a better system for writing numbers: the [positional system](#).

We 'just know' what a number like  $23$  means. However, for understanding how the computer works we have to investigate this gut feeling and put some technical terms onto it.

1. We are accustomed to count with our fingers from  $1$  to  $10$ . For this we have 10 symbols to count:  $0,1,2,3,4,5,6,7,8,9$ . These group of distinguishable symbols are called **digits**.
2. The amount of the digits is called **base**  $B$ . We are used to the decimal base  $B=10$ , in logic we used binary (also called dual)  $B=2$ .
3. When we count beyond the maximum number we are used to 'enlarge the number to the left': after the  $9$ , we count  $10$ . But the digit  $1$  in  $10$  is more worth than the  $1$  in  $31$  (of course). It is on a different **position**, on the position of the tens.
4. Each position gets numbered: the ones count  $0$ , the tens  $1$ , the hundreds  $2$ , the thousands  $3$  and so on. This 'position number' is called **index**  $i$ .
5. Knowing the index, also the 'worth of the position' can be derived: the **place factor**  $p$  (like one, ten, hundred) can be calculated with the base and the index:  $p=B^i$ .
6. A **numeral** as a group of digits represents what is commonly known as a number.
7. A **code** or **encoding** means a way to translate one way to display information into another. E.g. A decimal numeral into a Binary, or an idea of an algorithm into a computer language.

In order to recapitulate this for  $B=10$  we will calculate the amount of a decimal numeral here once in detail (click on the arrow to the right " $>$ " to see the next step, alternatively see [here](#)):

Ok, that was simple. But what about a binary number? Let's calculate the amount of a binary numeral:

So, what did we find out?

- The shown process is a relatively simple way to convert binary numerals to decimal.
- A 8 digit binary numeral is equal an up to 3 digit decimal numeral.  $\rightarrow$  Numerals in binary become lengthy

Therefore, a it would be better to have a more structured way in presenting the numerals in the which are used in the processor. Internally, the processor just knows 0's and 1's. But investigating a huge bunch of these (e.g. when analyzing the internal memory or a file) is not really catchy in order to understand anything.

The first step is to group the bits:

- 4 bits are called a **nibble** (the name derives from 'to bit' and 'to nibble')
- 8 bits are called a **byte**
- 16 bits are (usually) called a **word**. In details, this depends on the processor.
- 32 bits are (usually) called a **double word**. Like the word this also depends on the processor.

By this, one can separate parts of information (e.g. in a file) better. It is also important to mark the order of the bits. For decimal numerals like \$42\$ the rightmost digit has always the lowest value. The technical term for the 'lowest value' in a binary numeral is called **lowest significant bit** or **LSB**. When the LSB is on position 0 (= rightmost) this order is called **LSB 0**. This was used in the calculation above and is commonly used. In some cases (some memory setup and communication protocols) the order is just the other way around. In this case the **most significant bit** is on position 0. This order is therefore called **MSB 0**. Example:  $3_{10} = 0000_2; 0011_2$  (LSB 0),  $3_{10} = 1100_2; 0000_2$  (MSB 0)

What is still missing are expressions for large amounts of data. We can describe these using prefixes and the powers of two. You may already know the prefixes such as kilo and mega, but it is worth brushing up on the powers of two. The easiest way to illustrate this is with a chessboard. Maybe you know the legend of the inventor of the chessboard, who was granted a wish by the king. His wish was that the king would give him one grain of rice on the first chessboard square and every time twice as much on each subsequent squares. We'll play through this briefly, here to write down the powers of two:

- in the first square we enter two to the power of 0, which results in 1.
- In the second square, two to the power of 1, resulting in two.
- Then 4, 8, 16, 32, 64, 128, 256, 512 and then in the next line two to the power of ten, resulting in 1024.

You should definitely remember these sequence of the values for the power of 2! They are not only important for the exam, but also for the following semesters and computer science.

Fig. 2: chessboard with doubling numerals



1024 bit is also called **kbit** or **kilobit** in the semiconductor industry. You will notice here that the kilo is slightly more than 1000. To make it easier to distinguish, according to the ISO or ECE standard you should say **kibi** instead of "kilo" and write kibibit. The same applies to  $2^{20}$ , i.e. 1024 to the power of two: **megabit** has become common there. However, **mebibit**, should be used to differentiate. The nomenclature continues in the same way for  $2^{30}$  and  $2^{40}$ : gibibit and tebibit.

### 2.1.3 Other bases

When looking onto multiple bases, it is important to clearly **mark the base** of the numerals. Already in the chapter before a numeral like \$110\$ could either be \$110\_{10}\$ in decimal or \$110\_2\$ in binary, which is \$6\_{10}\$ in decimal.

In the following the base will be written as a subscript: \$110\_2 = 6\_{10}\$. In the end of this subchapter we will also see other ways to mark the base.

#### Hexadecimal

With the ideas of the previous subchapter we already can structure the bits. In order not to use the lengthy binary presentation it is common to use other bases. One important base is \$16\_{10}\$ for hexadecimal numerals. There, we need 16 distinguishable symbols: \$0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\$. With this digits it is possible to encode (=rewrite by other means) exactly 4 bit or one nibble. The encoding will be as follows:

Dual	Decimal	Hexadecimal
\$0000_2\$	\$0_{10}\$	\$0_{16}\$
\$0001_2\$	\$1_{10}\$	\$1_{16}\$
\$0010_2\$	\$2_{10}\$	\$2_{16}\$
\$0011_2\$	\$3_{10}\$	\$3_{16}\$
\$0100_2\$	\$4_{10}\$	\$4_{16}\$
\$0101_2\$	\$5_{10}\$	\$5_{16}\$
\$0110_2\$	\$6_{10}\$	\$6_{16}\$
\$0111_2\$	\$7_{10}\$	\$7_{16}\$
\$1000_2\$	\$8_{10}\$	\$8_{16}\$
\$1001_2\$	\$9_{10}\$	\$9_{16}\$
\$1010_2\$	\$10_{10}\$	\$A_{16}\$
\$1011_2\$	\$11_{10}\$	\$B_{16}\$
\$1100_2\$	\$12_{10}\$	\$C_{16}\$
\$1101_2\$	\$13_{10}\$	\$D_{16}\$
\$1110_2\$	\$14_{10}\$	\$E_{16}\$
\$1111_2\$	\$15_{10}\$	\$F_{16}\$

Tab. 1: hex encoding

This directly reduces the necessary amount of digits to show data. The hexadecimal representation is for example used in the file type \*.hex. This is the output file of an embedded c compiler and contains code in an machine-readable representation. An example of a c code and its hex-file is shown in [figure 3](#). In the hex-file the bytes (= 2 nibbles = 2 digits) are visibly grouped.

```

41 // Deklarationen -----
42 // Festlegung der Quartzfrequenz
43
44
45 #ifndef F_CPU
46 #define F_CPU 1843200UL // Minimaler mit 18,432 MHz Quartz
47 #endif
48
49 // Include von Header-Dateien
50
51 #include <avr/io.h> // I/O Konfiguration (intern weitere Dateien)
52 #include <util/delay.h> // Definition von Delays (Wartezeiten)
53 #include "lcd_lib_da.h" // Funktionsbibliothek zum LCD-Display
54
55
56 // Konstanten
57
58 #define MIN_PER 100 // minimale Periodendauer in "Timerticks"
59 #define MAX_PER 255 // maximale Periodendauer in "Timerticks"
60 #define WAIT_TIME 2000 // Wartezeit zwischen Taktungen in ms
61
62 // Funktionsprototypen
63
64 void initDisplay(void); // Initialisierung Display und Startanzeige
65 void initPorts(void); // Initialisierung der I/O-Ports
66 void initTimer(void); // Timer 0 initialisieren (Soundgenerierung)
67 void init(void); // generelle Initialisierungsfunktion
68
69
70 // Hauptprogramm -----
71
72
73 int main()
74 {
75     init(); // Ports und Timer 0 initialisieren
76     initDisplay(); // Display aktivieren
77
78     while(1) // Start der unendlichen Schleife
79     {
80         for (OCR0A=MAX_PER; OCR0A>MIN_PER; OCR0A--) // Frequenz erhöhen
81         {
82             _delay_ms(10); // in Schritten von 10ms
83         }
84         _delay_ms(WAIT_TIME); // Wartezeit hohe Frequenz
85
86         for (OCR0A=MIN_PER; OCR0A<MAX_PER; OCR0A++) // Frequenz absenken
87         {
88             _delay_ms(10); // in Schritten von 10 ms
89         }
90         _delay_ms(WAIT_TIME); // Wartezeit niedrige Frequenz
91
92     }
93
94 // Ende der unendlichen Schleife
95 }

```

Fig. 3: example of c code and hex-file

```

1 :1000000019C02BC02AC029C028C027C026C025C0BF
2 :1000100024C023C022C021C020C01FC01EC01DC0DC
3 :100020001CC01BC01AC019C018C017C016C015C00C
4 :1000300014C013C01241FBECFEFD4E0DEBFCDBF6C
5 :1000400011E0A0E0B1E0EAE1F2E002C005900D921B
6 :10005000A433B107D9F7B6D0DEC0D2CF5F9A5F988C
7 :100060000895CF93C82F88B1807F88B988B19C2F1D
8 :1000700092959F70892B88B9F1DF88B1807F88B90C
9 :1000800088B1CF70C82BC8B9E9DFCF910895F89433
10 :100090009FE097B98AB180698AB998B988B18F7696
11 :1000A0008BB92FEF8FE191E0215080409040E1F734
12 :1000B0000C0000088B1807F88B988B1836088B9AA
13 :1000C000CDDF8FEF93EB0197F1F700C0000C6DPA3
14 :1000D0008FEF93EB0197F1F700C0000BFDF88B10D
15 :1000E000817088B98FEF93E20197F1F700C0000AB
16 :1000F000B5DF86EE90E00197F1F7000088E2B1DF0E
17 :1001000096EF9A95F1F78CE0ACDF86EE90E00197E0
18 :10011000F1F7000082E0A5DF86EE90E00197F1F7AD
19 :10012000000081E09EDF8FEF95E30197F1F700C0BB
20 :10013000000081E096DF8FEF95E30197F1F700C0B3
21 :100140000000789408955C9A8CDF8DEF90E0019721
22 :10015000F1F70000895CF93DF93EC0188818823A5
23 :1001600029F02196F0DF89918111FCCDF91CF91A9
24 :1001700008955C9890E4899F600D1124862F805823
25 :1001800070DF86EE90E00197F1F70000895229A63
26 :10019000559A089582E184BD84E085BD85E487BDDC
27 :1001A000089575DF60E080E0E4DF81E191E0D3DF76
28 :1001B00060E081E0DEDF82E291E0CDDF0895E7DFED
29 :1001C000E9DF0895FCDFEDDF35E423E383E090E031
30 :1001D00037BD2A984FEF5FEF6CE2415050406040CE
31 :1001E000E1F700C0000027BD2A9A4FEF5FEF6CE2F5
32 :1001F000415050406040E1F700C0000019741F7D6
33 :1002000017BC8FEF9FE34CEE815090404040E1F7EB
34 :0A0210000C00000DBCF894FFCF20
35 :10021A00202020202020202020202020202020D4
36 :10022A000020204578706572696D656E74203261A3
37 :10023A002D0020506F6C697A65692020202020CB
38 :04024A002020000070
39 :00000001FF

```

The bytes in the first line are:

Byte0	Byte1	Byte2	Byte3	Byte4	Byte5	...
\$10_{16}\$	\$00_{16}\$	\$00_{16}\$	\$00_{16}\$	\$19_{16}\$	\$C0_{16}\$	...





For each of the shown steps, the integer of the division of the step before is used (\$28\$, \$1\$). The last steps (and any following) result in a remainder of \$0\$. For hexadecimal numeral we have to focus on the remainder from below to top and convert the value into hexadecimal digits

each position in decimal	$\color{red}{1}$	$\color{green}{12}$	$\color{blue}{4}$
each position in hexadecimal	$\color{red}{1}$	$\color{green}{C}$	$\color{blue}{4}$

The result is  $\color{red}{1}\color{green}{C}\color{blue}{4}_{16}$

## 2. handling the decimal places

The decimal places \$0.12\$ can be converted by repeatedly multiplying by the base (here \$B=16\$) and using the integer:



For each of the shown steps, the decimal places of the multiplication of the step before is used (\$0.92 \rightarrow .72 \rightarrow .52 \rightarrow .32 \rightarrow .12 \rightarrow .92\$).

Decimal place in the second last line is equal to the first one. Therefore also any further places will also be equal. That leads to repeating decimals.

For hexadecimal numeral we have to focus on the integer from top to below and convert the value into hexadecimal digits.

each position in decimal	$\text{\color{blue}\{1\}}$	$\text{\color{green}\{14\}}$	$\text{\color{brown}\{11\}}$	$\text{\color{red}\{8\}}$	$\text{\color{grey}\{5\}}$	$\text{\color{blue}\{1\}}$	$\text{\color{green}\{14\}}$	$\dots$
each position in hexadecimal	$\text{\color{blue}\{1\}}$	$\text{\color{green}\{E\}}$	$\text{\color{brown}\{B\}}$	$\text{\color{red}\{8\}}$	$\text{\color{grey}\{5\}}$	$\text{\color{blue}\{1\}}$	$\text{\color{green}\{E\}}$	

The result is  $0.\overline{\text{\color{blue}\{1\} \text{\color{green}\{E\} \text{\color{brown}\{B\} \text{\color{red}\{8\} \text{\color{grey}\{5\}}}}$

Results:

1. A decimal numeral has to be separated into integer decimal and decimal places.
2. By dividing / multiplying with the base the integer decimal / decimal places can be converted to another base. This works for all other bases like \$2\$ or \$8\$.
3. The results have to be converted (as least for base \$B > 10\$, like hexadecimal).
4. A small decimal places can lead to a longer (or even infinite) numerals in another base.

Especially the last result has a mayor impact for calulations on microcontrollers and computer: The internal logic is only based on binary, which also show this problem. However, the internal memory for a numeral is limited.

Even when stored in 32bit - it is not possible to exactly convert the \$0.12\_{10}\$ to binary. In the following table the \$n\$-bit equivalent of \$0.12\_{10}\$ in binary and hexadecimal system is shown. Additionally, this value is also re-converted to a decimal numeral.

number of bits \$n\$	number system	numeral
\$8\$	binary	\$0.0001\; 1111_2\$
	hex	\$0.1F_{16}\$
	equiv. dec	\$0.12109375_{10}\$
\$16\$	binary	\$0.0001\; 1110\; 1011\; 1000_2\$
	hex	\$0.1EB8_{16}\$
	equiv. dec	\$0.11999511718_{10}\$
\$24\$	binary	\$0.0001\; 1110\; 1011\; 1000\; 0101\; 0010_2\$
	hex	\$0.1EB851_{16}\$
	equiv. dec	\$0.11999994516..._{10}\$

number of bits \$n\$	number system	numeral
\$32\$	binary	\$0.0001\; 1110\; 1011\; 1000\; 0101\; 0001\; 1110\; 1100\_2\$
	hex	\$0.1EB851EC_{16}\$
	equiv. dec	\$0.12000000011..._{10}\$

This might seem as a little issue. But there are a lot of areas, where exact decimal places are mandatory, like banking, or simulations.

This situation even arises, when using [floating point representation](#) - the principal problem is also nicely explained in [this clip](#).

### 2.1.5 Binary Coded Decimals

The first approach to this was the development of **Binary Coded Decimals**.

The encoding algorithm from a decimal numeral into BCD is simple: *“don't use the division/multiplication method mentioned before - just take the same hexadecimal digit on each decimal position like the decimal one”* .

This means the decimal numeral \$391.21\_{10}\$ is encoded to \$391.21\_{BCD}\$. Inside the processor each digit is handled as hexadecimal number: \$3\;9\;1\;.\;2\;1\_{BCD}\$ equals \$0011\;1001\;0001\;.\;0010\;0001\_{2}\$.

The main disadvantage is the ineffective storage management and more complex algorithms for addition, subtraction and so on.

### 2.1.6 Marking the base of a numeral

Up to here, the marking was done by the subscript. Postfixes cannot be used in the software development environment, and sometimes are also not used in datasheets. Alternative ways for the marking are the following:

base	subscripted (mathematically)	prefixed (in code)	postfixed	further marking
2 (binary)	\$0010 1010_2\$	0b00101010	00101010B	\$\%00101010\$, 00101010b
10 (decimal)	\$1027_{10}\$	1027	1027D	-
8 (octal)	\$1027_{8}\$	01027	1027O	-
8 (hexadecimal)	\$27D_{16}\$	0x27D	27H	\$27, 27h

Be aware, that in the code 01027 is not equal to 1027!

#### When to use which base?

When programming code for an embedded system, the system will always see 0's and 1's after compiling. So, the microprocessor does not have to be considered.

In some cases on the other hand, the binary or hexadecimal numeral is much more convenient to read:

Here an example, where the binary numeral shows a smiley (e.g. for writing this on the screen), but the hexadecimal (or decimal) value does not give a clue what the output will be:



Fig. 4: Simulation of a half adder

The next step shall be a bit more complicated - or better: some bit more complicated. We want to do the calculation  $A + B = 0011_2 + 0111_2$ .

```
\begin{align*} \color{white}{+}\,00\boldsymbol{11}_2 \ \ +01\boldsymbol{11}_2 \\ \hline \color{white}{+}\overset{\color{red}{1}}{1}\overset{\color{red}{1}}{0}\overset{\color{red}{1}}{1} \\ \color{red}{\boldsymbol{1}}\boldsymbol{1}\overset{\color{red}{1}}{0}_2 \end{align*}
```

The rightmost, first step is easy, since we already had this in the examples before. The next step (marked in bold) is differing: not only do we have to consider the digits from  $A$  and  $B$ , but also the carry from the calculation before. The carry from before has to be added for all the next steps similarly.

For the calculation by hand, we did four individual additions from right to the left. The processor has to do the same. But first we have to expand the half adder. For a complete addition step we need a logic with the inputs  $A$ ,  $B$  and carry from the step before  $C_{in}$ . The output will be still  $S$  and  $C$ .

Fig. 5: Simulation of a full adder

In [figure 5](#) the full adder is shown. It is an alternative representation from what we have done for calculation by hand:

1. first add one bit from  $A$  to one bit from  $B$  (half adder). The result is in  $S'$  and  $C'$ .
2. Then, take the result  $S'$  and add it with  $C_{in}$  (second half adder). The result is in  $S$  and  $C''$ .  $C''$  is already the wanted output.
3. For the carry output we need to consider both carries  $C'$  and  $C''$ . When two or all of the inputs  $A$ ,  $B$ ,  $C_{in}$  are high, then the carry has to be set to high. This can be implemented with  $C' \text{ OR } C''$ .

This full adder can now be stacked together in order to add multiple bits  $A_0, A_1, \dots$  to  $B_0, B_1; \dots$ .

Fig. 5: Simulation of a 4-bit adder

**In Hexadecimal**

The calculation in hexadecimal is conceptually the same. Some examples, which we will discuss below:

```
\begin{align*} \begin{array}{|l} { a) } \\ { b) } \\ { c) } \\ { d) } \end{array} \\ \color{white}{+}5_{16} \ \ +3_{16} \\ \hline \color{white}{+}\overset{\color{red}{8}}{8}_{16} \\ \color{white}{+}\,1_{16} \ \ +D_{16} \\ \hline \color{white}{+}\overset{\color{red}{1}}{1}\overset{\color{red}{1}}{8}_{16} \\ \color{white}{+}E_{16} \ \ +A_{16} \\ \hline \color{red}{\boldsymbol{1}}\boldsymbol{8}_{16} \end{array} \end{align*}
```

- a) This one is simple: looks like a decimal formula..
- b) Here, the summands look like decimal numerals, but the result  $7_{10} + 3_{10} = 10_{10}$  is still within the range of the base. The correct symbol would be  $10_{10} = A_{16}$
- c) Now, the summands are a 'bit more hexadecimal'ly. The easiest way is: "convert the single digit

from hex to decimal, do the operation, re-convert to hex". For the given example:  $1_{10} + 13_{10} = 14_{10} = D_{16}$

d) Also for this calculation the described way is beneficial:  $E_{16} + A_{16} = 14_{10} + 10_{10} = 24_{10}$ . The result is larger than the base, and therefore the value has to be separated in more digits:  $24_{10} = 16_{10} + 8_{10} = 10_{16} + 8_{16} = 18_{16}$

For a hexadecimal value with more digits the carry of the calculation before has to be added - otherwise every step remains the same.

### 2.2.2 Subtraction

#### In Binary

In the following some examples shall show the concept for binary:

```
\begin{align*} \begin{array}{|l|} \hline a) \color{white}{0}_2 \ - \ 0_2 \\ \hline b) \color{white}{1}_2 \ - \ 1_2 \\ \hline c) \color{white}{1}_2 \ - \ 0_2 \\ \hline d) \color{white}{0}_2 \ - \ 1_2 \\ \hline \color{red}{1} \ \color{white}{0} \ \color{white}{1}_2 \\ \hline \end{array} \end{align*}
```

The calculation for d) shows the carry, which here has to borrow a bit from the next upper digit. This is similar to the calculation:  $\begin{array}{|l|} \hline \color{red}{1} \ 1 \ 9_{10} \\ \hline \end{array}$

With more digits the calculation in the binary system will look like the following:

```
\begin{align*} \color{white}{0}_2 \ - \ 01_2 \\ \hline \color{red}{+} \ \color{red}{0} \ \color{red}{0} \ \color{red}{1}_2 \\ \hline \end{align*}
```

In this example the bold column will be explained shortly:

```
\begin{align*} \color{white}{1}_2 \ - \ 1_2 \\ \hline \color{red}{1} \ \color{white}{0} \ \color{red}{1}_2 \\ \hline \end{align*}
```

The calculation has to be executed as follows:  $1_2 - (1_2 + \text{color{red}{small{1}}}) = 1_2$ . Additionally, another carry has to be taken from the next digit.

#### In Hexadecimal

The calculation in hexadecimal is conceptually again the same. Some examples, which we will discuss below:

```
\begin{align*} \begin{array}{|l|} \hline a) \color{white}{15}_{16} \ - \ \color{white}{3}_{16} \\ \hline b) \color{white}{23}_{16} \ - \ \color{white}{B}_6 \\ \hline c) \color{white}{3F}_{16} \ - \ 1A_{16} \\ \hline d) \color{white}{+} \ 38_{16} \ + \ 1A_{16} \\ \hline \color{red}{1} \ \color{red}{1} \ \color{red}{B}_{16} \\ \hline \end{array} \end{align*}
```

`\end{array} \end{align*}`

- a) This one is (again) simple: looks (again) like a decimal formula..
- b) Here, the both hexadecimal numerals look like decimal numerals, but the result  $3_{10} - 6_{10} = 7_{10} + C$  lead to an underflow, where we have to take a carry of  $C = -10_{10}$  for a decimal calculation. In hexadecimal the carry differs. A better way to solve such a subtraction in hexadecimal is to add the carry before:  $3_{16} - 6_{16} + \text{color{red}}\{10_{16}\}$ . Then the calculation can be converted to decimal:  $3_{10} - 6_{10} + \text{color{red}}\{16_{10}\} = 19_{10} - 6_{10} = 13_{10}$ . This result has to be converted back to hexadecimal:  $13_{10} = D_{16}$ . For the next column the carry has to be considered.
- c) For this formula the idea from b) helps, too: for each digit by digit subtraction, we will have a look, whether a transformation to decimal is beneficial. For the rightmost it is:  $F_{16} - A_{16} = 15_{10} - 10_{10} = 5_{10}$ . The rightmost result is now:  $5_{10} = 5_{16}$
- d) We can do the same thing here, for  $8_{16} - A_{16}$ : consider the carry, convert to decimal, calculate, re-convert to hexadecimal.  $8_{10} - 10_{10} + \text{color{red}}\{16_{10}\} = 24_{10} - 10_{10} = 14_{10}$ . The result is  $E_{16}$ .

**Note!**

The subtraction within the processor is done a bit differently, with the **two's complement**. In short: the range of numerals will be divided in such a way that negative numbers can also be represented. By this the subtraction can be transformed into an addition. It is roughly like transforming  $10_{10} - 4_{10}$  into  $10_{10} + (-4)_{10}$ .

**2.2.3 Multiplication and Division**

The multiplication table for binary is simple:

`\begin{align*} 0_2 \cdot 0_2 = 0_2 \quad 0_2 \cdot 1_2 = 0_2 \quad 1_2 \cdot 0_2 = 0_2 \quad 1_2 \cdot 1_2 = 1_2 \end{align*}`

Multiplication of longer binary numerals have to be executed similar to decimal numerals:

```

\begin{align*} \begin{array}{l} 1011_2 \cdot \\ \color{blue}\{1\}\color{brown}\{1\}0\color{violet}\{1\}_2 \quad \hline \color{blue} \\ \color{1011}\!\!\color{brown}\{\downarrow\};\color{violet}\{\downarrow\} \quad +;\color{brown} \\ \color{1011}\!\!\color{violet}\{\downarrow\} \quad +;\color{violet}\{1011\} \quad \hline \color{red}\{1\}\{1\}\overset{\color{red}\{1\}}{\{0\}}\overset{\color{red}\{1\}}{\{0\}}\overset{\color{red}\{1\}}{\{0\}}\overset{\color{red}\{1\}}{\{0\}} \\ \color{red}\{1\}\{0\}\overset{\{1\}}{\overset{\{1\}}{\overset{\{1\}}{\overset{\{1\}}{2}}}} \end{array} \end{align*}

```

The division is also relatively simple:

`\begin{align*} 0_2 : 1_2 = 0_2 \quad 1_2 : 1_2 = 1_2 \end{align*}`

With longer numerals it looks like the following:

Fig. 7: binary division  $10101_2 : 11_2 = 111_2$

## related Links

- [Conversion tool from decimal to hexadecimal](#), this tool shows also the steps and can be used vice versa
- A short video on the “[Everything Formula](#)” which prints all possible output and is directly related to binary encoding.

## Exercises

### Exercise 2.3.1. power tables

fill out the tables and remember the numerals!

Power...	Power...	Decimal numeral			
$2^{13}$	—				
$2^{12}$	$16^3$	4	0	9	6
$2^{11}$	—				
$2^{10}$	—				
$2^9$	—				
$2^8$	$16^2$				
$2^7$	—				
$2^6$	—				
$2^5$	—				
$2^4$	$16^1$				
$2^3$	—				
$2^2$	—				
$2^1$	—				
$2^0$	$16^0$				1

Fig. 8: Important numerals 1

Fig. 9: Important numerals 2

Power...	Power...	Decimal numeral							
$2^{-1}$	—	0.	5						
$2^{-2}$	—	0.							
$2^{-3}$	—	0.							
$2^{-4}$	$16^{-1}$	0.							
$2^{-5}$	—	0.							
$2^{-6}$	—	0.							
$2^{-7}$	—	0.	0	0	7	8	1	2	5

Text is not SVG - cannot display

Binary				Decimal	Hex	BCD
0	0	0	0			
1	1	1	1			

Fig. 10: Important numerals 3

Text is not SVG - cannot display

**Exercise 2.3.2. Conversion to Binary**

Convert the following hexadecimal numerals to the binary system.  
 Tip: Convert nibble by nibble to binary.

1. \$4321H\$
2. \$0xCAFE\$
3. \$ \\$ \$4be79 \$
4. \$ \\$ \$398B4C \$
5. \$0x3a4f.4ecd \$
6. \$ \\$1.3DAF \$

### Exercise 2.3.3. Conversion to Hexadecimal and Decimal

Convert the following numerals from dual to hexadecimal and decimal numerals.

Tip: Use the numerals from Exercise 2.3.1.

1.  $0b10110110_2$
2.  $10100101_2$
3.  $1111_2, 0111_2, 0001_2, 0011_2$
4.  $0110111.0101_2$
5.  $0.10101_2$
6.  $0.1001_2, 1001_2, \dots_2$

### Exercise 2.3.4. Conversion to Hexadecimal and Binary

Convert the following numerals from decimal to hexadecimal and dual numerals.

Tip: Use the numerals from Exercise 2.3.1.

1.  $123_{10}$
2.  $637_{10}$
3.  $1777_{10}$
4.  $8999_{10}$
5.  $41.4_{10}$
6.  $0.9_{10}$
7.  $11.3_{10}$

### Exercise 2.3.5. Addition and Subtraction

Execute the following operations manually with the given dual numerals.

1.  $01110111_2 + 11010101_2$
2.  $01011001_2 + 10011111_2$
3.  $01100111_2 - 01001100_2$
4.  $10101110_2 - 10000111_2$
5.  $0011.111_2 - 0011.100_2$
6.  $001111.011_2 + 1.01001_2$
7.  $01101.100_2 - 01000.111_2$
8.  $100101.11_2 - 110001.10_2$

### Exercise 2.3.6. Multiplication and Division

Execute the following operations manually with the given dual numerals.

1.  $1011_2 \cdot 0101_2$
2.  $01100101_2 \cdot 110_2$
3.  $10101111_2 : 101_2$
4.  $11110000_2 : 1000_2$

### Exercise 2.3.7. Operation in Hex

Execute the following operations manually with the given dual numerals.

1.  $\$1334H + 07ABH\$$
2.  $\$0DC43H - 0BD19H\$$
3.  $\$1F23H + 90E8H\$$
4.  $\$98C5H - 84CAH\$$
5.  $\$234AH + 7EE6H\$$
6.  $\$0F10CH - 0ED43H\$$

### Exercise 2.3.8. Further Questions

1. Write down the decimal numeral for the first 4 hexadecimal places
2. Which possible value range can be covered with an 8-bit variable?

### Exercise 2.3.9. One's Complement

In the simulation in [figure 11](#) the one's complement of a 4bit value is shown. On initialization the value are the following:

- The value for  $\$A\$$  is  $\$1010_2 = 10_{\{10\}\$}$ . The decimal value 10 is also shown in the first display from the left in the circuit.
- The value for  $\$B\$$  is  $\$1110_2 = 14_{\{10\}\$}$ . The decimal value 14 is also shown in the second display from the left in the circuit.
- The addition  $\$A+B\$$  leads to  $\$S = 1000_2 = 8_{\{10\}\$}$ . The decimal value 10 is also shown in the rightmost display in the circuit.
- There is another vaule in the second display from the right. This is called  $\$B'\$$  and it is equal to  $\$0010_2 = 2_{\{10\}\$}$ .

Fig. 11: One's Complement

1. Why is  $10 + 14 = 8$ ? What happens for other values?  
The input values A and B can be changed by clicking on the bit values.
2. Try to analyze how B' (shown in the brackets) is derived from B.
3. Have a look onto the wiki page of the [Ones' complement](#) in order to understand how negative numbers are represented in a microcontroller.
4. Imagine that you have to rescue data from an old storage device. The interesting bits are given by the boxed area in [figure 12](#). The bytes are LSB 0 oriented.
  1. What are these values in unsigned integer?
  2. What are these values in signed integer based on the ones' complement?

Fig. 12: example for stored data in an EEPROM



From: Courbon F., Skorobogatov S., Woods C. (2017) Reverse Engineering Flash EEPROM Memories Using Scanning Electron Microscopy. In: Lemke-Rust K., Tunstall M. (eds) Smart Card Research and Advanced Applications. CARDIS 2016. Lecture Notes in Computer Science, vol 10146. Springer, Cham. [https://doi.org/10.1007/978-3-319-54669-8\\_4](https://doi.org/10.1007/978-3-319-54669-8_4)

From:  
<https://wiki.mexle.org/> - **MEXLE Wiki**

Permanent link:  
[https://wiki.mexle.org/introduction\\_to\\_digital\\_systems/number\\_systems?rev=1666739586](https://wiki.mexle.org/introduction_to_digital_systems/number_systems?rev=1666739586)

Last update: **2022/10/26 01:13**

