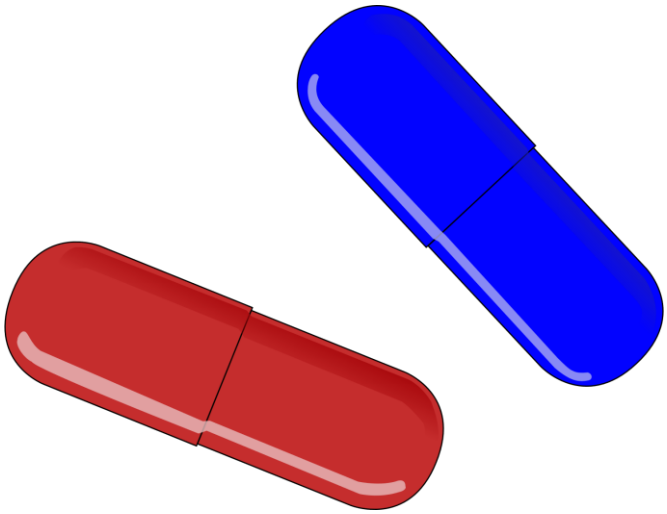
Practical Assembly Programming

From A to Z

Alexey Lyashko



Segmentation

1 Bootstrap

There is a widespread misbelief that one should by all means avoid using x86 Assembly (we will further use the term “Assembly” for simplicity) language as it is complicated, error prone, high level language compilers make better optimizations than one could make by hand and because programming in Assembly requires much more effort. While the accuracy of these statements is sometimes arguable and sometimes equals to 0, writing a book in order to simply refute them would be an inexcusable waste of time and effort, just like arguing whether chicken preceded the egg or vice versa. Instead, the intention behind this work is to show why it is important to know what lies beneath all the technology, even in this era of virtually unlimited resources and wide variety of software building instruments for every taste, and demonstrate how such knowledge can be applied in daily work.

1.1 Why another book on Assembly?..

Many books have been written on Assembly programming before, however, most of them successfully walk you through 16 bit DOS programming and stop somewhere in the beginning of 32 bit world and only a few dare to touch the “long mode” (64 bit universe). Others teach you how to write programs for specific platform (e.g. Windows, Linux). A variety of interesting and very useful tutorials and informative articles may be found on the internet; however, such resources tend to focus on author’s field of interest at the moment of writing or target very specific audiences. While all the aforementioned sources are extremely valuable, it is often too difficult to find needed information.

With this book, I would like to invite you to join me on a marvelous journey along the blooming tree of software development technology. Starting with the underlying infrastructure and processor internals, through different modes of processor operation and up to advanced programming techniques and approaches in software development on modern operating systems, while discovering some undocumented features thereof.

1.2 Prerequisites

An attempt has been made not to target specific audience with this book, but rather provide information and reference for anyone, starting with complete neophytes and up to experienced developers. Some programming experience or otherwise achieved understanding of certain basic terms (e.g. bit, byte, hexadecimal) is highly recommended.

What we would need, however, is a PC emulator of your choice. It may be Oracle VM VirtualBox (https://www.virtuabox.org) or VMware Workstation (http://www.vmware.com/products/workstation.html); more advanced readers may decide to use QEMU (http://wiki.qemu.org/Main_Page). My personal suggestion would be VirtualBox, not only because it is free, but because we hardly need a bulldozer for playing in a sandbox.

Another product that would be needed along our journey is an assembler (a program that translates Assembly code into machine code). You are free to choose any of many available on the internet, however, I would suggest Flat Assembler (https://flatassembler.net) as it is the one used for all the examples in this book, and it is free and very powerful. However, since the

idea is not to teach the reader specific assembler syntax or features, but rather introduce the power of Assembly Language, the choice is free.

2 Introduction to binary system

This chapter is a very brief introduction to binary arithmetic – the fundamental aspect of digital electronics and the foundation stone of modern computing systems. You are welcome to proceed to the next chapter if you feel confident enough in all related to basic arithmetic operations in binary world.

2.1 Bits, bytes and friends

In the beginning there was a *bit* – the minimal representation of information, which could only tell "yes" or "no", "true" or "false", "on" or "off". Bit, as an information representation means, may be considered one of the most significant inventions along with discovery of penicillin and alike. It is hard to imagine modern world without bits, as almost everything we see around us is working or simply exists in virtue of bits. However, how much information can one represent with a single bit? Well, just "true" or "false". Grouping bits in sequences is what makes the huge difference. For example, this book was initially a sequence of bits too, albeit, this would be a rather complex example to consider. Let us take a look at something simpler – Morse code representation of the "SOS" signal ". . . _ _ _ . . .". And now, let us convert it to a sequence of bits, where signal is represented by '1' and spaces between signals are represented by '0', in which case we get '101010111011010101", with '1' and '11' being short and long signals respectively. Such encoding is called "binary" (prefix "bi" means two). Interestingly enough, "SOS" signal in such form takes exactly 20 bits – the bit width of Intel 8086 processor's data bus. Such form, however, is far from being ideal – just imagine what a mess we could have if we had to come up with specific binary representation for everything. Instead, some sort of unification was needed and that is why bits were grouped into *bytes*.

It is a wide-spread misperception, that byte contains exactly 8 bits. In fact 8 bits are called *octet* [1]while byte may be of any reasonable size. This is how Werner Buchholz explains it in "Planning a computer system":

> *"Byte denotes a group of bits used to encode a character, or the number of bits transmitted in parallel to and from input-output units. A term other than character is used here because a given character may be represented in different applications by more than one code, and different codes may use different numbers of bits (i.e., different byte sizes)." (Buchholz, 1962)*

As we see, byte is rather a minimum addressing unit (although, in case of "Project Stretch", such claim is incorrect), which is exactly 8 bits on x86 systems. There are certain instructions

[1] Octet means a group of eight. Since there is no standard definition of size of a byte, the term "octet" is used in protocol specifications or in any document that refers to exactly eight bits.

operating on *nibbles*[1], but the minimum addressing unit is still 8 bits. ASCII character code, where each byte corresponds to a single symbol or control code, is a good example of how useful such organization of bits may be. However, no matter how good the idea is, it needs a better way of representation than bits. You would agree, that denoting ASCII code for letter 'A' as 01000001b is not quite comfortable, therefore, any value that occupies more than a single bit is usually represented in *hexadecimal*[2] or decimal notation. Hexadecimal notation, however, is much preferred as it allows identification of individual bytes.

Hexadecimal notation represents groups of four bits as a single character using numbers from `0` to `9` and letters from 'a' to 'f', where `0` is, well, `0` (binary `0000b`) and 'f' is `15` (binary `1111b`). Numbers in hexadecimal notation are either preceded by '`0x`' or followed by 'h' as `0x1234` or `1234h`. Important detail to remember when using the 'h' notation is that numerical values cannot start with a letter – compiler won't know that you intended to write a hexadecimal number in such case. In order to solve this problem, prepend '`0`' to a number like in `0CAFEh`.

Decimal	Binary	Hexadecimal	Decimal	Binary	Hexadecimal
0	0000	0x0	8	1000	0x8
1	0001	0x1	9	1001	0x9
2	0010	0x2	10	1010	0xA
3	0011	0x3	11	1011	0xB
4	0100	0x4	12	1100	0xC
5	0101	0x5	13	1101	0xD
6	0110	0x6	14	1110	0xE
7	0111	0x7	15	1111	0xF

Table 1. Correlation between decimal, binary and hexadecimal notations.

Let us use the "SOS" sequence mentioned above to demonstrate the transformation between binary and hexadecimal notations:

```
10101011011011010101b → 1010 1011 0110 1101 0101 → 0xAB6D5
                         A    B    6    D    5
```

Eight bits, however, are only capable of representing values in ranges `10000000b` to `01111111b` for signed and `00000000b` to `11111111b` for unsigned numbers, which are `-128` to `127` or `0` to `255` respectively. Following the bit to byte conversion method, we have to start grouping bytes if we need to use larger ranges of possible values (and we do). A group of two bytes on Intel/AMD based systems is called *word*. Furthermore, a group of two words (4 bytes) forms *double word* or *dword*, a group of two double words (8 bytes) forms *quad word* or *qword*. As each byte added to a group enlarges the range by 2^8, it is possible to host very large numbers with

[1] Nibble is a four bit half of a byte.

[2] The term *hexadecimal* means base 16. Hexadecimal representation only allows digits 0 to 9 and letters 'a' to 'f'.

virtually any amount of bits, the problem would only be in processing thereof and we will soon see why.

2.2 MSB & LSB[1]

Another problem yet to be solved is the order of bits in a byte and order of bytes in larger sequences. Everything is simple in case of bytes (regardless of byte size) as binary representation of a byte value is written from left, starting with the most significant bit, to right, ending with the least significant bit. Let us take value 0x5D as an example:

Bit rank	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Binary	0	1	0	1	1	1	0	1
Hexadecimal	5				D			

Table 2. Order of bits in a byte.

As we may see, the most significant bit of 0x5D, the one that stands for 2^7, is 0 and the least significant bit, the one representing 2^0, is 1.

We have seen how easy the conversion between binary and hexadecimal notations is; now let us use Table 1 as an illustration of binary to decimal conversion. In binary notation, each bit represents a $2^{(\text{bit position}2)}$ value and the final value of a byte is the sum of values represented by bits that are set to '1'. Taking the binary representation of the byte used in Table 1 we get the following equation:

$$2^0 + 2^2 + 2^3 + 2^4 + 2^6 = 1 + 4 + 8 + 16 + 64 = 93 = 0x5D$$

The same rule applies to values that occupy more than one byte. For example decimal value 953 (hexadecimal 0x3B9) – this value needs two bytes (8 bits each) and its binary form is 0000001110111001b, in which case the equation would be:

$$2^0 + 2^3 + 2^4 + 2^5 + 2^7 + 2^8 + 2^9 = 1 + 8 + 16 + 32 + 128 + 256 + 512 = 953 = 0x3B9$$

While everything is clear with the order of bits in a byte, the order of bytes in a word (dword, qword), while being written from left to right with most significant byte (the one with the most significant bit) on the left and the least significant byte on the right, may differ when stored in memory depending on the underlying architecture. There are two types of architectures:

1. *Little-endian* – the least significant byte is stored first, meaning the order of bytes in memory is the opposite to that of a written hexadecimal number;
2. *Big endian* – bytes are stored in memory in the same order as when written in hexadecimal notation.

Table 2 shows the difference between the two.

---

[1] 'MSB' and 'LSB' used in the title of this section stand for "Most significant byte" and "Least significant byte" respectively.

[2] Bit positions are counted starting with 0, not 1.

Platform type	Little-endian		Big-endian	
Written form	0x3B9		0x3B9	
Addresses	addr	addr + 1	addr	addr + 1
Bytes	0xB9	0x03	0x03	0xB9

Table 3. The difference between Little-endian and Big-endian byte orders.

As we are going to work on either Intel or AMD processors, we may temporarily forget about Big-endian platforms.

2.3 Binary arithmetic

Now we have a general understanding of how data is organized in computer's memory, however, such knowledge is completely useless, unless we know how to handle data in binary form. In other words, there is no good in numbers if you do not know how to add/subtract/etc.

This is where *Boolean*[1] *algebra* enters the scene. Boolean algebra naturally operates on values 1 and 0 (true and false), which perfectly fits our needs, and provides all the tools needed for arithmetic operations on binary numbers (that is why it is fundamental in computing and digital electronics). If we attempt to emulate Boolean algebra with regular arithmetic, we would perform all computations on 1's and 0's only, divide the resulting value by 2 and take the quotient as the final result. Such arithmetic is called integer arithmetic modulo 2 (or *GF(2)*[2]). However, it is important to mention that there is no such thing as addition/subtraction/etc. in Boolean algebra, at least not as we know it, instead, it uses Boolean operators (or *gates* in digital electronics).

There is a limited set of four basic[3] operators:

- *NOT* – inverts the given value (NOT 1 = 0).
- *OR* – "returns" the maximum of two values (0 OR 0 = 0, 1 OR 0 = 1).
- *AND* – "returns" the minimum of two values (1 AND 1 = 1, 1 AND 0 = 0).
- *XOR* – exclusive OR, "returns" 1 only if two values are different (1 XOR 1 = 0, 1 XOR 0 = 1).

The word "returns" is quoted because operators are not functions, they do not return values, rather operate on them. The following four truth tables show the exact behavior of each of the basic operators depending on the input values.

NOT	OR

[1] Named after English mathematician, philosopher and logician George Boole, Boolean algebra, unlike elementary algebra, operates on truth values "true" and "false", usually denoted as 1 and 0.

[2] Galois Field (named after Evariste Galois) of x elements or GF (x) is a field that contains a finite number of elements.

[3] XOR is not really a basic operator as it may be represented by a set of several AND and NOT operators.

Values	
0	*1*
1	*0*

Values	0	1
0	*0*	*1*
1	*1*	*1*

AND		
Values	0	1
0	*0*	*0*
1	*0*	*1*

XOR		
Values	0	1
0	*0*	*1*
1	*1*	*0*

Table 4. Truth tables for basic Boolean operators

2.3.1 Binary addition

These truth tables, however, do not provide any explanation on how binary arithmetic works. All we know is that when we add two bits like this (1 + 0) mod 2 the result would be 1, just like in normal arithmetic, but, if we add 1 to 1, shouldn't the resulting value be 2? The answer is no, at least not in case of addition of two one-bit values. Instead, we need some mechanism for detection of "lost" bit and that is when Boolean AND comes to our aid.

Suppose we have to values `A` and `B` that we want to add. Each value is exactly one bit and each of them equals to `1`. Before we proceed, it is the perfect time to introduce *`carry`*. *Carry is a digit that is transferred from one column of digits to another column of more significant digits* (Wikipedia) and, to put it in simple words, we ran out of digits. For now, as we have no column of more significant bits, we would be satisfied by simply knowing whether carry occurs or not.

As in binary arithmetic addition is performed by XOR operator we simply use it to add A to B (remember A = 1 and B = 1):

```
1 XOR 1 = 0
```

Then we use AND operator in order to check whether carry occurs:

```
1 AND 1 = 1
```

The result equals to 1, meaning that carry occurred.

Now it looks like we are mature enough to try a few more bits, for example 4. How about we add 9 to 7? This time we will be using a table to make the example more illustrative.

Bit rank		2^3	2^2	2^1	2^0
Carry					0
A		1	0	0	1
B		0	1	1	1
Result					

As we see, first bits of A and B are equal to 1, therefore all is simple and we just perform the same procedure as in previous example: `Result = 1 XOR 1 = 0` and `Carry = 1 AND 1 = 1`. Do not forget to move the carry to the next column (the 2^1 one).

Bit rank		2^3	2^2	2^1	2^0
Carry				1	0
A		1	0	0	1
B		0	1	1	1
Result					0

The situation is a bit different for the second and the rest of bits as we do not simply add second bit from A to second bit from B, but add the carry bit too: Result = (0 XOR 1) XOR 1 = 0 and carry would be (0 XOR 1) AND 1 = 1. The table will look like this:

Bit rank		2^3	2^2	2^1	2^0
Carry			1	1	0
A		1	0	0	1
B		0	1	1	1
Result				0	0

We proceed and perform the same procedure for the third bit and get:

Bit rank		2^3	2^2	2^1	2^0
Carry		1	1	1	0
A		1	0	0	1
B		0	1	1	1
Result			0	0	0

And the last time:

Bit rank		2^3	2^2	2^1	2^0
Carry	1	1	1	1	0
A		1	0	0	1
B		0	1	1	1
Result		0	0	0	0

The result equals to 0 despite the fact that `9 + 7 = 16` or `1001b + 0111b = 10000b`. As we see the result would be five bits long, but as we only have four bits for each variable in this example, the would-be most significant bit remains as carry (we will see how such situation is handled in the next chapter).

2.3.2 Binary subtraction

Before we proceed to binary subtraction, we have to understand how signed and unsigned numbers are implemented in binary arithmetic. In binary notation, the sign of a signed value is indicated by the most significant bit, where 1 denotes a negative value and 0 denotes a positive one. However, setting the most significant bit to 1 is not enough to convert a positive value to negative. For example, if we want to convert a byte value of 1, which would be represented as `00000001b`, to -1 and simply set the most significant bit to 1, we would get `10000001b` as the result which is -127, while negative 1 would be represented by `11111111b`. To understand how this happens, let us inspect the process of negation of a binary number, which is called *two's complement.* In order to change the sign of a number we have to invert its bits first, which in case of `00000001b` would result in `11111110b`, and then add 1. The final result would be `1`1111111b (the most significant bit set to 1 denoting a negative value).

Let's see how binary subtraction works by example. Suppose we have two bytes A and B, where A = `01110010b` (`0x72`) and B = `01010011b` (`0x53`) and we want to subtract B from A. We, of course, may perform the subtraction by means of regular arithmetic, but our intention is to see how it works with Boolean operators on a bit per bit level. Just remember to add bits and calculate carry bit with the previously mentioned formulae using XOR and AND operators.

1. The first step would be inversion of all bits of byte B:
   `NOT 01010011b = 10101100b;`
2. Complete the two's complement by adding 1 to the result of step one (remember to perform the calculations from right to left):

Bit rank		2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Carry	0	0	0	0	0	0	0	0	
A		1	0	1	0	1	1	0	0
B		0	0	0	0	0	0	0	1
Result		**1**	**0**	**1**	**0**	**1**	**1**	**0**	**1**

3. Add the resulting `10101101b` to byte A:

Bit rank		2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Carry	1	1	1	0	0	0	0	0	
A		0	1	1	1	0	0	1	0

B		1	0	1	0	1	1	0	1
Result		**0**	**0**	**0**	**1**	**1**	**1**	**1**	**1**

The result is `00011111b` (`0x1F`). Notice the carry bit – its presence means that we were subtracting smaller integer from bigger one and no *borrow*[1] has occurred. Let's try to subtract A from B to make this example even more illustrative.

1. The first step would be inversion of all bits of byte A:
   `NOT 01110010b = 10001101b;`
2. Complete the two's complement by adding 1 to the result of step one (remember to perform the calculations from right to left):

Bit rank		2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Carry								1	
A	0	1	0	0	0	1	1	0	1
B		0	0	0	0	0	0	0	1
Result		**1**	**0**	**0**	**0**	**1**	**1**	**1**	**0**

3. Add the resulting `10001110b` to byte B:

Bit rank		2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Carry	0	0	0	1	1	1	1	0	
A		1	0	0	0	1	1	1	0
B		0	1	0	1	0	0	1	1
Result		**1**	**1**	**1**	**0**	**0**	**0**	**0**	**1**

The result is `11100001b`, which, if treated as a signed value, is a negative number and the fact that carry bit equals to 0 denotes an occurred borrow.

It is important to note the fact that a number may be signed or unsigned within a certain range of bits, but that is only an abstraction which is important for us and the compiler, while the processor is unaware of sign as a concept at all, although, it provides us with certain indications of sign, sign change, etc.

[1] The most basic definition of "borrow" – it is the "carry" of subtraction.

2.3.3 Binary multiplication

Binary multiplication, just as any other multiplication, is simply an iterative addition. While the idea behind multiplication is the same, there are several ways of implementation. However, since our goal is to obtain general understanding of how multiplication is performed on binary level, we will not dive into technological aspects or into other, more complex, but more efficient algorithms.

What we do in decimal worlds is we multiply the multiplicand by each digit of the multiplier writing down products in column, while shifting each next product to the position of the digit we multiplied by. For example, 127 * 32 would be (decimal notation):

$$
\begin{array}{rrrr}
 & 1 & 2 & 7 \\
 & & 3 & 2 \\
\hline
 & 2 & 5 & 4 \\
3 & 8 & 1 & 0 \\
\hline
4 & 0 & 6 & 4
\end{array}
$$

Here we first multiplied 127 by 32 getting 254, then multiplied 127 by 3 getting 381, but, since it is a multiplication by 30 rather than by 3, we shifted 381 one digit to the left or multiplied it by 10. The final step was addition of the two products.

The same multiplication rule applies to binary multiplication with only one additional aspect to keep in mind – we should use Boolean operators in order to better understand how the process flows in a processor. As usual, we will be operating on eight bit bytes.

Suppose we have two bytes - A=`01110010b` and B=`00000101b` (`0x72` and `0x05` respectively), and we want to multiply A by B:

$$
\begin{array}{cccccccc}
0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
\hline
0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0
\end{array}
$$

Table 5. Long multiplication of binary numbers (Gray zeros represent shift amount).

As we see, we have to first multiply 01110010b by the first bit of B, which in Boolean representation is applying AND operator to each bit of A and the first bit of B. In the second step we do the same for the second bit of B and so on, up to the last bit of B. This example also introduces the fact that the *product is always twice the size of multiplicand* (and the size of multiplier is the size of multiplicand), which, in case of 8-bit multiplicand is 16 bit – 0000001000111010b.

2.3.4 Binary division

Binary division follows the same algorithm of long division as decimal numbers and it all comes down to iterative subtraction of divisor from portions of dividend. The following example fully illustrates the process.

Suppose we have two bytes A and B, where A=11010011b and B=00001100b, and we want to divide A by B. The first step we should take is to align the two values by their most significant bits and the alignment is performed by shifting left the divisor as needed. In our case, we shift the divisor 4 bits to the left B << 4 = 11000000b (grayed out bits represent the shift amount and do not participate in the further calculation process) so that we have:

A=11010011b
B=11000000b

Then we subtract B from the corresponding bits of A, meaning 1101b – 1100b, which results in 0001b. As this intermediate result fits the condition of being above or equal to 0, we write 1 to quotient while the intermediate result becomes new dividend.

Now we have to align the most significant bits of the dividend and the devisor, but this time, instead of shifting the devisor, we start borrowing bits from the original dividend and append bit at position 3 in the original dividend to our intermediate result getting 10b. However, since subtracting 1100b from 10b would result in a negative number, we append 0 to the quotient and borrow the next bit at position 2. Having 100b we would still get a negative number after subtraction, so we append 0 to the quotient and borrow the next bit at position 1. We will return on these steps until we either can subtract our divisor from the intermediate result or get a positive value or 0, or we run out of bits in the original dividend (in which case, the intermediate result is the remainder).

In this particular example, we will have to borrow all the remaining bits from the original dividend in order to be able to subtract 1100b and get a positive result. At the end, we would have 10001b as the quotient and 111b as the remainder.

```
11010011|1100 ← divisor
1100    |10001 ← quotient
  10011
   1100
    111 ← remainder
```

Grayed out bits in the intermediate result represent bits “borrowed” from the original dividend.

3 Central Processing Unit

CPU- the Central Processing Unit is the heart of any modern digital system, starting with a disc drive and up to a most sophisticated server. It is, however, important to remember, that CPU alone is just a piece of hardware and is completely useless without other devices (e.g. mother board, memory, etc.) and especially without a program to run.

3.1 Pre-micro era

The first Turing-complete general-purpose computer was presented to the public in autumn 1945. It may be safe enough to say, that the Electronic Numerical Integrator and Computer (*ENIAC*) was in fact the first CPU, rather than computer and it occupied 1800 ft^2 (167 m^2) – twice the size of an average apartment. Being such a tremendous thing, it was tremendously slow (in comparison to today's CPUs, of course) with a basic cycle of 20 cycles of 100 kHz clock. Neither did it have any internal memory to store data - punch cards were used for that purpose up until 1953, when a magnetic-core memory was added. Punch cards were also used as the only means of input/output system. There was a lot of whining about the need to use more than one diskette for Windows installation in 1990's, now try to imagine **thousands** of punch cards…

Although, computers were only becoming more sophisticated and significantly reduced size since 1945, it took the technology more than 20 years to reduce the size of a computer to something that would fit on a desk.

3.2 Intel 4004

In 1971 Intel Corporation released its Intel 4004 4-bit microprocessor – the first commercially available microprocessor on a single chip. Something that we take for granted these days was a huge technological breakthrough at that time, despite the fact that it was a 4-bit, *BCD*[1]-oriented processor. Just imagine that before Intel 4004, CPUs were literally not only built out of several integrated circuits, but it was a very common thing to build a CPU from several circuit boards.

3.3 8086 and so on

Intel delivered its first 16-bit processor in 1978. Just imagine that up until 1982 you could not access more than 1MB of RAM, while nowadays we can theoretically access terabytes or memory with 64 bit processors. Well, in principle, we can access much more, but there are physical limitations too.

As we advance through this book, we will see how the technology was changing since late 70's and early 80's in the previous century, throughout 90's and 00's. Starting with a trivial bootloader, which could nicely run on 8086 processor, we will finish our journey with a tiny and very basic 64-bit operating system. However, try not to forget, that even the most sophisticated processor is just a set of logic gates, which adheres the rules and principles developed by George Boole, who was born 202 years ago…

---

[1] BCD stands for Binary Coded Decimal – decimal numbers are encoded with 4 bit binary values from 0 to 9.

Are you ready to go?