flat assembler

Programming Tutorial

Chapter 1

Information encoding

This chapter discusses the elementary knowledge about the information encoding, whose understaing is essential for anyone wishing to learn programming. If you already know that all or you want to quickly get some practice instead of theory only, you can skip to the next chapters, and you can always come back here, when you don't understand something.

You will also find here some tables, which can become very handy reference for you at all the time when reading this tutorial.

1.1 Bits and binary values

Every computer operates on the elementary portions of information, which are called bits. One bit is piece of information equivalent to answer "Yes" or "No" for a single question. What does a particular bit mean, depends of course on the question for which that bit is an answer. The content of that answer we call the value of bit, and in the world of computers it is usually written as a single digit, 1 for positive answer and 0 for negative. If we have more bits, they can give us more information - as if we were asking more and more detailed questions about what interest us. And computers use some common schemmas for interpreting a bunch of bits as a some particular kind of information – a number, text, image or sound.

The most basic of such schemmas is the binary encoding of numbers. To understand it better, let's start with recalling the decimal encoding, which we all are using everyday. To write a number, we use a pack of digits, each of them can be from 0 to 9, the last of them is the count of ones, the one before last is the count of tens, the second before last is the count of hundreds, and so on. Actually it's good to call the last digit the first one – as the growing powers of ten we are adding on the left end of the number, and I'll use this convention here.

Now, what if we wanted to write a number using the binary digits (that is bits)? The first digit cannot be higher than 1, so the second digit needs to be the count of twos, and – applying the same scheme as we have with decimal encoding – third digit will be the count of fours, and each next will be count of amounts twice as large as previous one – namely will be the next power of two. If we write digits in the way that the corresponding amounts increase from right to left (exactly as we do it with decimal numbers), the number two becomes written as 10, three is 11 (as it's the two plus one), four is 100. The bit 1, as it's the maximal value of digit, plays role similar to digit 9 in the decimal numbers. When we have a number being the chain of ones, the number larger by one will be written as the same count of zeros following the single 1. As this second number is just the two to the power of zeros' count, the highest number we can write with the n bits is $2^n - 1$.

To distinguish between decimal and binary numbers, we will attach the letter "b" to the binary ones. So, with this convention, the 7 is the same as 111b and 10 is the same as 1010b. This is also how we will write such numbers when programming.

1.2 2–adic numbers and binary arithmetics

Now I'll introduce a bit more abstract scheme of encoding, which is not actually used in computers (as it is physically impossible, as we can store only finite amount of bits in the computer, and this scheme is using an abstract, infinite number of binary digits), but it can be helpful in deeper understanding the arithmetic operations on binary numbers, which computers are performing.

This time we will reverse the order of digits and write the first digit on the left end of number. Also, as we can add any amount of zeros after the last digit, we will put the infinite amount of zeros on the right end. For example, 101100000... is a decimal number 13 written this way, the three dots mean that infinite amount of zeros follows. This way we can encode any natural number with an infinite chain of binary digits.

If we want to add such numbers, we can do it just in the same way as when we add decimal numbers – begin with adding first digits, so you'll get the first digit of result and a carry, then add second digits with a carry, and you'll get second digit of result and some other carry. You can continue this scheme endlessly, but – as out numbers consist only of zeros starting from some place – the result will also have only zeros starting from some (possibly further) place. Look at this example, we are adding two such binary numbers, the small digit above each column is the carry:

$$\begin{array}{cccccccccccccccc}
 & & \scriptstyle 0 & \scriptstyle 0 & \scriptstyle 0 & \scriptstyle 1 & \scriptstyle 1 & \scriptstyle 1 & \scriptstyle 1 & \scriptstyle 0 & \scriptstyle 0 & \scriptstyle 1 & \scriptstyle 1 & \scriptstyle 0 & \scriptstyle 0 & \\
 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & \ldots \\
+ & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & \ldots \\
\hline
 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & \ldots
\end{array}$$

We end up adding only zeros from some place, but what if there were infinitely many ones in the chain? The scheme will still work, and we'll get some result, altough such chains are not numbers in our hitherto interpretation – as we won't get any finite result when adding infinitely many powers of two, that each one larger than previous.

Let's try adding such "infinite" number to a natural number, and see what happens. We will take number 1 as a first chain, and add to it chain consisting of infinitely many ones:

$$\begin{array}{cccccccccccccccc}
 & & \scriptstyle 1 & \scriptstyle 1 & \scriptstyle 1 & \scriptstyle 1 & \scriptstyle 1 & \scriptstyle 1 & \scriptstyle 1 & \scriptstyle 1 & \scriptstyle 1 & \scriptstyle 1 & \scriptstyle 1 & \scriptstyle 1 & \scriptstyle 1 & \\
 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots \\
+ & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \ldots \\
\hline
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots
\end{array}$$

The carry is always 1 and the result consists only of zeros. The result is actually the zero number! So we can interprete that chain consisting of infinitely many ones as the minus one. We could also find it out the other way, if we tried to use the substracting scheme with borrowing to substract one from zero. To see how does it work, we will now substract number 13 from zero, and we will see how does the -13 look like (the small digit below each column is the borrow):

$$\begin{array}{cccccccccccccccc}
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots \\
- & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots \\
 & & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & \\
\hline
 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \dots
\end{array}$$

We are always forced to borrow, as we've got only zeros in the upper chain, and therefore, when we have only zeros in lower chain left, we end up substracting 0 digit from lower chain and 1 borrow from the two (obtained by taking the next borrow), so we get only ones in the result. We can see, that in general, representation of any negative number contains only ones starting from some place, just in the same way as any positive number end with infinitely many zeros.

So we have both positive and negative numbers encoded this way, we know how to add and substract one from another. What about multiplication? Again we can try adapting the scheme we know for the decimal numbers. In the following example we multiply number 3 by 5:

$$\begin{array}{ccccccc}
 & 1 & 1 & 0 & 0 & 0 & \dots \\
\cdot & 1 & 0 & 1 & 0 & 0 & \dots \\
\hline
 & 1 & 1 & 0 & 0 & 0 & \dots \\
 & & 0 & 0 & 0 & 0 & \dots \\
 & & & 1 & 1 & 0 & \dots \\
 & & & & 0 & 0 & \dots \\
 & & & & & 0 & \dots \\
 & & & & & & \dots \\
\hline
 & 1 & 1 & 1 & 1 & 0 & \dots
\end{array}$$

For each digit in the second chain we take the first chain, multiply it by this digit, and shift to the right so its first digit will be at the same column as the digit, by which we were multiplying. Then we add all the rows to obtain the result – each column contains only finitely many digits, so it can be done. The binary multiplication is even easier compared to decimal one, as we have to multiply only by one or zero – in first case we just copy the chain, in second case we fill it with zeros. And this operation can be performed even on chains containing infinitely many ones, too – you can try multiplying -1 by itself, and see that you get 1 as result.

And what about chains that contain infinitely many ones mixed with infinitely many zeros? Could they correspond to fractions? For example, let's try to find a chain, that would give us 1 when multiplied by some natural number. Imagine we are multiplying some natural number by some chain. To get 1 as a first digit of the result, we need both first digits of our natural number and that second chain to be 1. So we can try it only with odd numbers. But the next digit we have to get is zero – we have the first row of our multiplying scheme fixed, and we can manipulate the second one to have zero or one as the first digit, so we can choose second digit of out chain such that we'll get zero as a second digit of result, and also in the same way we can choose all the next digits. So for any odd number we can find a chain, which is its reciprocal. For example $\frac{1}{3}$ is a chain that contains 1 followed by the sequence of 1 and 0 repeated infinitely, here's the multiplication scheme that comes with construction of it:

$$\begin{array}{ccccccccc}
 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & \ldots \\
\cdot & 1 & 1 & 0 & 1 & 0 & 1 & 0 & \ldots \\
\hline
 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & \ldots \\
 & & 1 & 1 & 0 & 0 & 0 & 0 & \ldots \\
 & & & 0 & 0 & 0 & 0 & 0 & \ldots \\
 & & & & 1 & 1 & 0 & 0 & \ldots \\
 & & & & & 0 & 0 & 0 & \ldots \\
 & & & & & & 1 & 1 & \ldots \\
 & & & & & & & 0 & \ldots \\
 & & & & & & & & \ldots \\
\hline
 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots
\end{array}$$

In fact, we could also get reciprocals of even numbers if we allowed chains that have some finite amount of digits also to the left from the first digit, these digits would correspond to the negative powers of two – such construction gives all rational numbers as the chains, which are periodic starting from some place, and all other chains correspond to irrational numbers. Such infinite chains are called by mathematicians the 2–adic numbers, but we don't need any more details about them here. We will now come back to simple binary numbers that are used in computers, and see what the just gained knowledge can be used for in that matter.

In computer we have some fixed number of bits for storing each number, so we can only encode positive numbers not greater than $2^n - 1$, where n is the number of bits we have. Two such numbers can be added using the same scheme, as the one we were using for infinite 2–adic numbers, and the result will fit in the $n+1$ bits, with the highest bit being equal to the last carry. Processor stores this additional bit in some special place, when it is one, it means that the result is too large to fit in out n bits.

Also substraction of two such numbers is performed using the same scheme as we were using above, and the last borrow is also stored in some special place by processor – if it's one, it means that we have substracted the larger positive number from the smaller one. But in such case the result is a negative number, and we get the first $n+1$ bits of the chain, which – as we remember – is filled with ones after that initial bits. And therefore there is a second interpretation of such numbers – we look at the highest bit, and if it's 0, we treat that number as followed by zeros (so it is a positive number), and if the highest bit is 1, we treat such number as followed by ones (so it is a negative number). The highest bit is in this case called a sign bit, as it indicates whether number is positive or negative (zero is treated as positive in this interpretation) and the largest positive number we can encode this way is $2^{n-1} - 1$, as we have one less bit for it (the sign bit must be zero), the smaller positive number is -2^{n-1} (it has the sign bit set to 1, and all lower bits set to 0), and they are just cut initial parts of their corresponding 2–adic chains (and the sign bit is the first of the repeated zeros or ones ending that chain).

When when use the first interpretation (without sign bit, only positive numbers), it called that we interpret the bit chain as unsigned value, otherwise we interpret it a signed value. A good point is that the addition and substraction can be performed the same way in both cases (but we must pay attention to the carry/borrow bit).

When we multiply two n–bit numbers with the same scheme as we used for the p-adic chains, in the last row we have the n–th bit of the first number shifted right by n columns, so it's obvious that the result will need at least $2n$ bits to fit in. For that reason processor usually gives us two n–bit numbers as the result of multiplication, one containing first n bits of the result (we will call them lower bits), and one containing the next n bits (we will call them higher bits). Also, as all rows but last need to be extended with bits higher than n during the multiplication, processor needs to have specified whether the numbers should be interpreted as unsigned (and so always extended with zeros) or as signed (and extended with the repeated sign bit). However, because those extension bits occur only in the columns after the n-th one, lower n bits of result are not a affected, so the lower half of result will be the same with both unsigned and singed multiplication schemes.

Processor is also capable of dividing binary numbers, but as it's usually much slower than multiplying, we will later utilize the schemma for getting 2-adic reciprocals for the tricks of division by known in advance number done with multiplication.

1.3 Logical operations

Operations that are performed on individual bits are called the logical ones, it is related to fact, that we can interprete one bits as a logical value of truth or false. The simplest such operation is the negation – it just switches the value of a bit to the opposite one, if it was 0, it becomes 1, and if it was 1, it becomes 0. If we perform negation on the value consisting of more bits, each of them is changed to its opposite[1], for example negation of the value 10110001b is 01001110b.

Other logical operations are performed on two bit values, giving the one bit as a result. The conjuction of two bits (which we will call the AND operation) has the value of 1 only if both bits have the value 1, otherwise it is 0. The alternative of two bits (which we will call the OR operation) has the value of 0 only when both bits have the value 0, otherwise it is 1. The disjunction of two bits (also known as the "exclusive or", we will call it the XOR operation) is 1 only when the bits have different values. The table 1.1 shows the results of those operations for the four possible pairs of bits.

		AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Table 1.1: Logical operations.

If we perform such operation on some n-bit value with a second n-bit value, each bit of the result will come from the corresponding bits of both values, for example result of conjunction of 11010101b with 00011111b is 00010101b.

1.4 Hexadecimal numbers

Binary numbers have one disadvantage when written as text – they are very long compared to decimal numbers. Not only they take more place, but are also harder to read. For example, number 128 written in binary is 10000000b – it is easy to make a mistake when quickly counting

[1] We can also do a logical negation arithmetically, by substracting the number from -1, as in such substraction scheme borrow is always zero, and each bit is changed to its opposite.

zeros. In decimal, to make such numbers more readable, whe usually split them into groups of three digits. As with bits we are already used to the powers of two, in case of binary numbers we'll make groups of four digits, so our number 128 will consist of two such packs[2]. But instead of just separating those groups with a comma or space, we'll do a better trick – replace each group with a single character.

How is it supposed to work? Four bits in a group can be interpreted as a number from 0 to 15, and each such pack is corresponding to this value multiplied by some power of 2^4, that is by some power of 16. So if we look at each such group as at the single digit, the whole number can be seen as written with encoding of base 16 (in the same way as binary has base of 2 and decimal has base of 10), such encoding is called hexadecimal. As we need 16 different digits, we will use letters A–F in addition to ten taken from the decimal system. The table 1.2 shows the correspondence between hexadecimal digits and the binary 4–bit values.

Hex	Bin	Hex	Bin	Hex	Bin	Hex	Bin
0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

Table 1.2: Hexadecimal digits.

We will usually add letter "h" at the end of hexadecimal number to distinguish it from others. For example, number 137, which in binary is 10001001b, in hexadecimal is 89h – you can check it by splitting the binary representation into two groups of four bits and looking up in the table the corresponding hexadecimal digits.

Hexadecimal encoding, while being as handy as decimal, allows us to keep the track of the bits – therefore this encoding we'll be using the most often when programming.

1.5 Bytes and ASCII encoding

The fundamental unit of information larger than bit is a byte. One byte consists of 8 bits, and it is the most elementary unit for all the operations in computer, every kind of information is stored as a bytes – even when some larger units are used, they are simply some multiplies of bytes.

As byte contains 8 bits, any value of it can be written with two digits in hexadecimal system, and it can be any unsigned number from 0 to 255 (because 2^8 is equal to 256). This is not much, but is enough for the purpose of ASCII encoding, for which individual bytes are the most commonly used. ASCII stands for the American Standard Code for Information Interchange, and it encodes a text as a chain of bytes. Each byte encodes one letter (or some other character), in table 1.3 you can see the correspondence between numerical values of bytes (given in both decimal and hexadecimal system) and the various characters. As you can see, table only lists the numbers smaller than 80h – that's because characters encoded by larger numbers (they are called the extended ASCII set) are not standarized, and can be different for the encoding of various human languages. Also numbers smaller than 20h have some special meanings assigned instead of characters, more details about them be introduced later.

[2] Such pack of four bits is often called a nibble

And for a simple example: the word “byte”, as it consists of four letters, needs four bytes to be encoded, and after looking up appropriate values in the table, we get that these bytes should be 62h, 79h, 74h and 65h. And this is how such text is seen by computer.

1.6 Larger information units

Because one byte is able to keep only small numbers, for the calculations usually some larger units are used. If we use the unit of two bytes, that is of 16 bits, we can encode unsigned numbers up to 65535 (as the 2^{16} is 65536), with the four hexadecimal digits. We can still think of it as of the two separate bytes, one being the lower part of number, and one being the higher. For example 16–bit value 1B73h have lower byte 73h and higher byte 1Bh. As bits in higher part correspond to the powers of two from the eight to fifteen, the value of 16–bit number is the value, which we obtain when we multiply the value of higher byte by 256 and then add the value of lower byte to it. We will also use 32–bit or even 64–bit units for calculations. They also can be easily splitted into smaller units.

If we want to convert some small number into number encoded with the larger amount of bits, we have to extend it with the additional high bits. And what should be values of those bits, it depends whether we interprete that number as a signed or unsigned one. When the number unsigned, we simply have to fill all additional bits with zero, and this is called the zero–extension. When it is signed, we have to fill the additional bits with the same value, as the sign bit has (so in particular the new sign bit will have the same value), this is called the sign-extension.

There are also some much larger units used to measure big amounts of information. One kilobyte is equal to 1024 bytes (as this is 2^{10}, and we are using powers of two everywhere, as they are the “round” numbers for computer), one megabyte is 1024 kilobytes, one gigabyte is 1024 megabytes and one terabyte is 1024 gigabytes.

Dec	Hex	ASCII	Dec	Hex	ASCII	Dec	Hex	ASCII	Dec	Hex	ASCII
0	00	Null	32	20	Space	64	40	&	96	60	‘
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmission	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Bell	39	27	’	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Negative acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End of block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	Delete

Table 1.3: ASCII codes.

Chapter 2

DOS Programming

The Disk Operating System is the oldest of systems, for which we will learn programming here, so we'll start with it. You don't have to worry, if you have only Windows system on your computer, as every Windows system contains a subsystem compatible with DOS. We will begin with learning some basics of this system usage, so if you are already familiar with DOS, and know how to use its commands, you can skip it and proceed to the section 2.2.

2.1 Basic DOS commands

This section has not been written yet...

2.2 Introduction to assembly

The flat assembler executable (`fasm.exe`) can be executed from command line, and requires two parameters – first should be the name of existing file, which will be taken as source file, second should be the name of file to which assembler will write its output.

The source file contains commands which tell assembler what data should it put into the output file (we will call these command the directives). Let's see it on example. Use the text editor to create the file `example.asm` and put the following text inside:

```
db 97
```

Now assemble it with command `fasm example.asm example.txt`. DB tells assembler to put data of byte size into the output file, the value following it is the value of byte, which should be put into output. So file `example.txt` should now contain a byte of value 97, that is the "a" letter. You can open this file in the text editor and see that it's true. DB allows you to put more than one byte at a time, and values of bytes can be given in many different ways. Open the file `example.asm` and modify it to get the following:

```
db 97,62h,'c'
```

This will put three bytes into output file – they are separated with commas. First byte is the same as in previous example, second byte has value given in hexadecimal – it's the code of letter "b" (98 in decimal), the third byte is given literally – in this case it should be enclosed in quotation marks. After assembling it you should have the text "abc" in the output file – check it. When quoting bytes literally, you can also put them more that one of them inside the quotation marks and all that bytes will be put into output, the same result as above can be obtained from the much simpler source:

```
db 'abc'
```

You also put more such instructions into the source file, but each one must reside in separate line, for example:

```
db 97
db 98
db 99
```

All right, this way we can create file with any data in it, but what is the main purpose of an assembler is writing programs. Program is a special kind of file that contains instructions for processor. These instructions are encoded as a chains of bytes, this encoding is called the machine code. The simplest kind of program is file with the `com` extension – such file is simply loaded in some place in memory and then executed. Let's make some simple example to see how does the machine code look like. Edit the file `example.asm` to contain the following source:

```
db 0CDh,20h
```

Now assemble it with command `fasm example.asm example.com`, you'll get the `example.com` file, which can be executed. You can try it, but nothing special will happen. This is because the two bytes we have put into that file form one instruction in the machine code, which in DOS causes the program to be terminated. If there was not such instruction, processor would keep going and execute everything what was in memory even after the loaded content of file, and that would be a bit dangerous.

So, different byte values encode different instructions, but you won't have to remember all such number to programm in assembly. That's because assembler allows you to use the so–called mnemonics, directives that encode the instructions for us. Using this feature, the above program can be rewritten like below:

```
int     20h
```

After assembling it, our program will contain the same two bytes as previously. So it contains the instruction called INT with parameter 20h. This instruction tells the processor to execute some special procedure, which is called an interrupt. There are 256 different interrupts, numbered from 0 to 0FFh, so we have to specify which one we want to have executed. In this case we execute the interrupt of number 20h, which is handled by DOS and causes the program to be terminated. Now we will learn one more useful interrupt function, and a few other things by the way, here's the more complex example of program:

```
mov     ah,2
mov     dl,97
int     21h
int     20h
```

Now we are executing few more instructions before the interrupt terminating the program – there is one more interrupt, with number 21h. It does its job and returns the control to our program, but what job is done, depends on some additional parameters. The parameters for interrupts have to be provided in registers - places in processor which can hold some values. For example, there is a register called AH which can hold a byte value, and interrupt 21h needs it contain a number identifying the job which it have to do.

We put the value 2 into AH register with the instruction, which has the mnemonic MOV. This instruction puts some value into the chosen place, it needs two parameters - first tells where we want to put some value (in this case this is the AH register), second contains the value we want to put there. The value 2 in AH register tells the interrupt 21h to display some character on the screen. The character, which will be displayed, is taken from the DL register. This may look a bit complicated, but you don't have to remember that everything - when you want to use some interrupt, you can look into appendix A and find the necessary information (you'll soon learn everything what is needed to understand all the details provided there).

We put the code of "a" letter into DL register and then execute the interrupt 21h. After it does what it has to do, it returns control to our program and we execute the interrupt 20h - program ends. You can check it, this program will display the letter "a" on the screen – it is first our program which actually does something.

In the above program we have used two registers - there are more of them, some hold 8 bits (one byte) of data, and some 16 bits (two bytes, we will call it a word). Each of the byte size registers is actually the lower or higher part of larger register. There are four 16–bit registers about which we should learn for the beginning, they are called AX, BX, CX and DX[1]. Each of them consists of two 8–bit registers, name of lower part begins with the same letter as the whole register and ends with the L letter, name of higher part ends with the H letter. So AX registers consists of AL and AH registers, BX contains BL and BH, CX contains CL and CH, and DX contains DL and DH.

Let's see on example how we can utilize the fact that 8–bit registers are contained withing 16–bit ones. There is a function of interrupt 21h which allows to terminate program and return some value to DOS - this function has number 4Ch, so this value has to be stored in AH register. Also, it needs the exit code to be provided in AL register. Usually zero is used as exit code to signalize that program has done its job succesfully, other value means that some error has occured. Let's say that we want to return zero. Then we have to put 4Ch into AH and 0 into AL, so we would do it with following instructions:

```
        mov     ah,4Ch
        mov     al,0
        int     21h
```

But we know that those two registers are actually the higher and lower part of the AX register. So we can put the values into them both with just one instruction:

```
        mov     ax,4C00h
        int     21h
```

[1] Altough those names seem to come just from consecutive letters of alphabet, they are also acronyms for the functions of registers: AX is often called accumulator, BX – base index, CX – counter, DX – data register. You'll see later what do they come from.

There are many operations we can perform with registers. Let's start with instructions doing addition and substraction, they have mnemonics ADD and SUB. Remember that always when you perform some operation, the first parameter following the mnemonic (those parameters are called operands) specifies what value you want to modify (it is called destination operand), and the second specifies the value with which the operation will be performed. So if you want to add 2 to the value in register DL, you should do it this way:

```
        add     dl,2
```

Similarly, you can substract some value from register:

```
        sub     dl,2
```

There are also instructions that increment and decrement the register by 1, those mnemonics are INC a DEC. First just adds 1 to the destination operand, second substracts 1 from it. In this case no second parameter is needed, because the mnemonic itself defines by what value the modification will be done, so you have to specify only the destination operand, like:

```
        inc     dl
```

We already know how to display the value of DL register as a character, so we can check the result of performing some operation on register. Here's a small sample:

```
        mov     dl,'0'
        add     dl,3
        mov     ah,2
        int     21h
        mov     ax,4C00h
        int     21h
```

First it loads the value of character "0" into DL and then adds 3 to it, so the value of character "3" is obtained. Then it displays that character on the screen and exits.

So far we were performing all operation with destination operand being a register, and source operand being some number (it is often called an immediate operand, as its value is obtained immediatelly from the instruction), but that's not all we can do. The source operand operand can also be a register, you can see the example in the following program, which is the modification of previous:

```
        mov     dl,'0'
        mov     ah,2
        add     dl,ah
        int     21h
        mov     ax,4C00h
        int     21h
```

We are adding the value of AH register to the DL register, so we obtain the value of "2" character.

Instead of registers, we can also use the memory operands – they give us access to computer's memory, which contains a large amount of bytes. Each byte in memory has its address – a number, which tells where the byte is placed. First byte has address 0, second has address 1, and so on. To access memory of given address, we need just to write that address in the square brackets, for example `mov al,[0]` will copy value of byte placed at address 0 into AL register. It's also possible to access larger units of information in memory, `mov ax,[0]` will copy 16–bit word from address 0 into AX register, that is it will copy the first byte of it (which has address 0) into AL, and the second byte (which has address 1) into AH register.

In the same way we can use memory as a destination of operation, but we cannot use more than one operand at one time. Therefore to copy a value from one place in memory to another, we need to first copy it to some register, and then from that register to the destination in memory.

But before we can start playing with the memory, we need some more information. We already know, that program is loaded into memory to be executed. Simple binary programs, like those few we have already written, are always loaded starting from address 100h, and from that place the instructions are executed. But let's say we have placed some bytes later in our program, after other instructions. How do we know what exactly address they have? To help us in such cases, assembler allows us to define labels anywhere in our program. We can use any word for a label[2], and it should be followed by the colon. This way we assign to that word the value of address for the place of program, where we put this label. If we then place that word in some instruction, it will be the same, as if we put there the address, that is assigned to it.

It should become more clear, when we see it on example, but there's one more detail we need to know about before we use labels – our program is loaded by DOS at address 100h, but the assembler by default assumes, that the instructions it generates will be placed at address 0. So we need to tell the assembler, that the address of program is 100h, otherwise it will assign the wrong values to our labels. We can do it with the ORG directive, which forces the assembler to assume, that the bytes of the following instructions will be placed at address given by its parameter. And here is our example program:

```
        org     100h

        mov     ah,2
        mov     dl,[exclamation]
        int     21h
        mov     ax,4C00h
        int     21h

    exclamation:
        db      '!'
```

It defines the label for byte containing the code of exclamation character, and uses that label to display it on the screen.

Now we are prepared to use another function of interrupt 21h, which allows to display more characters at one time. It has the number 9, and needs an address to be provided in DX register. It displays the characters from memory starting from that address, until it reaches the character of code 24h (the dollar sign) – it doesn't display such character, and stops on it. So we can use this function to display some message:

```
        org     100h
```

[2] Except for the words like instruction mnemonics or register names. If we try to define label of such name, assembler will give us an error message "reserved word used as symbol".

```
        mov     ah,9
        mov     dx,message
        int     21h
        mov     ax,4C00h
        int     21h

    message:
        db      'Hello!',24h
```

This time we don't use any instruction accessing the memory, we just load the address of our message as an immediate value into DX register, and the interrupt function displays it.

Note that it's not neccesary to start a new line after a label, like we have to do it after every instruction, last two lines of the above program can be combined into one line, we can also put more than one label at the beginning of a line.

When we try to make some operation on memory with an immediate value, the another problem emerges – we need to specify on how large unit of information we want the operation to be performed. The assembler won't accept instruction like `mov [100h],0`, because it's ambiguous. Instead, we shall write `mov byte [100h],0` to copy a byte (8 bits), or `mov word [100h],0` to copy two bytes (16 bits). The words BYTE and WORD in those instructions are called size operators, we will introduce more of them later. Every operand can be preceded by such operator, but we didn't need to use them with register operands, as all registers have predetermined sizes. If one of the operands has determined size, there is no need specify a size for the second one – that's why we didn't have to place size operators before immediate operands. And, of course, we cannot specify operands of different sizes for one operation.

But we can also make a memory operand to have the predetermined size as the registers have, and therefore avoid being forced to use size operator. We can achieve it by associating a label to the data of some size, it's done by putting the label just before the directive that defines data and omitting the colon character. This is how should it look like:

```
    message db 'Hello!',24h
```

With such definition `mov [message],0` is a valid instruction, and operates on the 8–bit data, but instruction `mov ax,[message]` is no longer valid, because the sizes don't match. Anyway, such label can still be used to perform 16–bit operations, by writing the instructions like `mov word [message],0` or `mov ax,word [message]` – such use of a size operator is called the size override.

Any immediate value, including the addresses inside the square brackets, can be represented by some expression, whose value is calculated at the assembly time. For example, we can write `mov al,[message+1]` to access the second byte of data following our label, as we are using the address larger by one.

In the next following sample program we will play a bit with instructions accessing memory and modify this way the message that will be displayed. The instructions that call the interrupt functions are the same as in out previous program, also the text of the message is the same, but it is not really the text that will be displayed when running this program – that's because of those three new instructions at the beginning of program:

```
        org     100h
        mov     [message+1],'i'
        mov     ax,word [message+5]
        mov     word [message+2],ax

        mov     ah,9
        mov     dx,message
        int     21h
        mov     ax,4C00h
        int     21h

    message db 'Hello!',24h
```

First of those inserted instructions modifies the second byte of our message to be the code of letter "i" – we don't have to use the size operator, because the label for our message declares the operation on data of byte size. Next instruction copies two bytes – the sixth and seventh byte of our message into AX register, and the following one copies those two bytes from the AX back to the memory – but this time into third and fourth byte of our message, this time we need to use the size override. This way we obtain the short message "Hi!", as later bytes won't get displayed – because the mark of the message's end has been placed at fourth byte of our data.

The other thing we can use the label for is the jump instruction. This one forces the processor to change the place from which it takes the instructions to execute. Our programs are always loaded at address 100h and from that point processor starts executing the instructions consecutively until it reaches the instruction that causes the program termination. But we can change this order with the jump instruction, which has the JMP mnemonic. For example `jmp 0` instruction would force the processor the begin executing the instructions from address 0. As we don't know what bytes reside at that address, it is better not to try it[3]. Instead we will define some label and jump to it. Look on this modification of the previous program:

```
        org     100h

        jmp     display_message
        mov     [message+1],'i'
        mov     ax,word [message+5]
        mov     word [message+2],ax

    display_message:
        mov     ah,9
        mov     dx,message
        int     21h
        mov     ax,4C00h
        int     21h

    message db 'Hello!',24h
```

This time program begins with jump instruction, and it moves the execution of program to the `display_message` label. So the next instruction executed after the jump is `mov ah,9` – the

[3] Altough in this case it wouldn't be harmful at all, as it just causes the program termination – we will later see why.

instructions that modify the message bytes are skipped and therefore the original message is displayed.

The JMP instruction we've been using above is often called an unconditional jump, because it performs the jump always, no matter where and when it's used. And there are also instructions, that perform jump only in some certain condition and do nothing otherwise – they are called conditional jumps. The conditions that can affect the behavior of such jumps are determined by the contents of some special register, containing separate bits, each having some special meaning. These bits are called flags and their values are set depending on the results of various operations performed by processor. For example, there's a bit called CF (carry flag), which is set to the value of last carry after the addition operation, or to the value of last borrow after the substraction.

Appendix A

Selected DOS interrupt functions