Tutorial Emu86

1) Numbering systems tutorial

What is it?

There are many ways to represent the same numeric value. Long ago, humans used sticks to count, and later learned how to draw pictures of sticks in the ground and eventually on paper. So, the number 5 was first represented as: **| | | | |** (for five sticks).

Later on, the Romans began using different symbols for multiple numbers of sticks: **| | |** still meant three sticks, but a **V** now meant five sticks, and an **X** was used to represent ten of them!

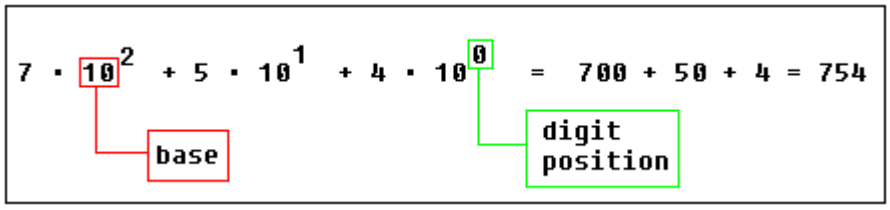
Using sticks to count was a great idea for its time. And using symbols instead of real sticks was much better.

Decimal System

Most people today use decimal representation to count. In the decimal system there are 10 digits:

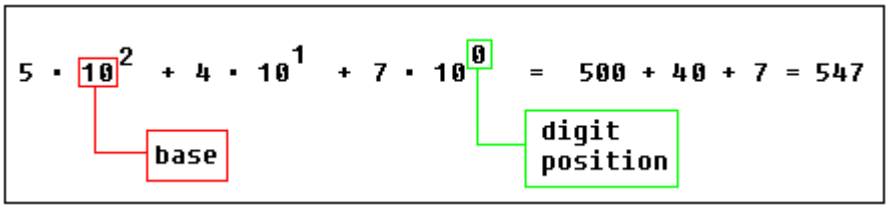
0, 1, 2, 3, 4, 5, 6, 7, 8, 9

These digits can represent any value, for example:
754.
The value is formed by the sum of each digit, multiplied by the **base** (in this case it is **10** because there are 10 digits in decimal system) in power of digit position (counting from zero):

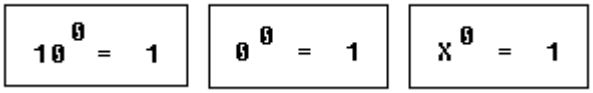


Position of each digit is very important! for example if you place "7" to the end:
547

it will be another value:



Important note: any number in power of zero is 1, even zero in power of zero is 1:



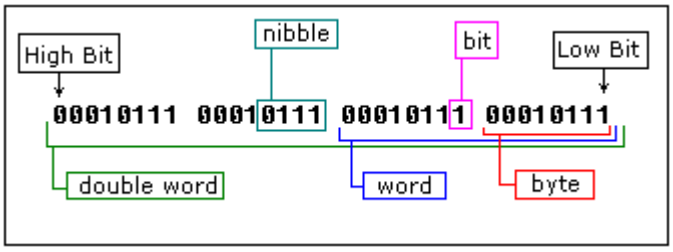
Binary System

Computers are not as smart as humans are (or not yet), it's easy to make an electronic machine with two states: **on** and **off**, or **1** and **0**. Computers use binary system, binary system uses 2 digits:

0, 1

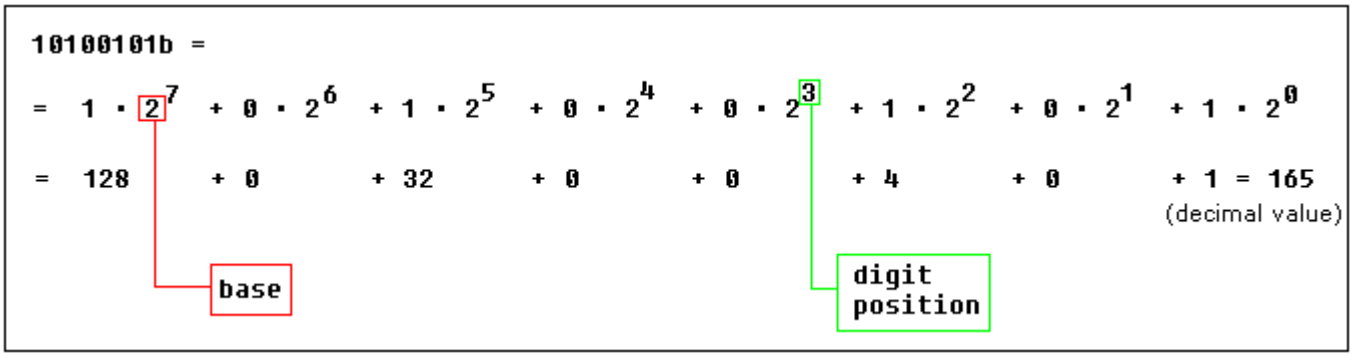
And thus the **base** is **2**.

Each digit in a binary number is called a **BIT**, 4 bits form a **NIBBLE**, 8 bits form a **BYTE**, two bytes form a **WORD**, two words form a **DOUBLE WORD** (rarely used):



There is a convention to add **"b"** in the end of a binary number, this way we can determine that 101b is a binary number with decimal value of 5.

The binary number **10100101b** equals to decimal value of 165:



Hexadecimal System

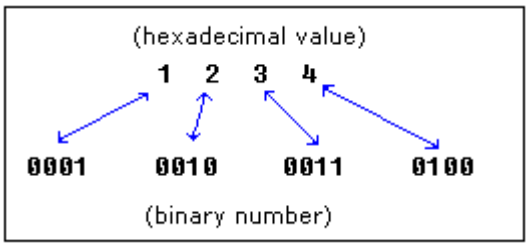
Hexadecimal System uses 16 digits:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

And thus the **base** is **16**.

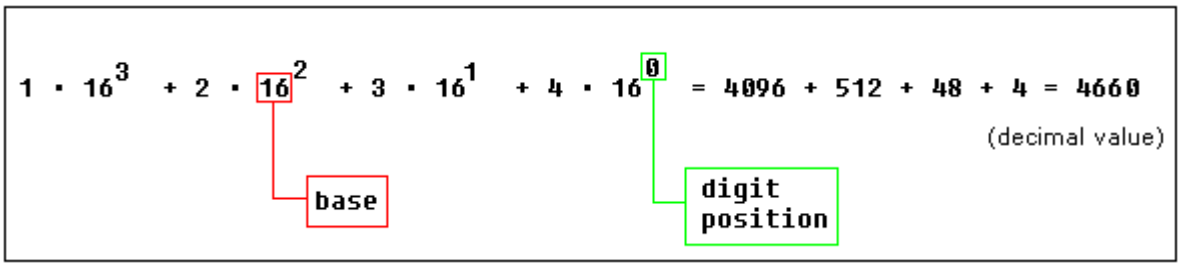
Hexadecimal numbers are compact and easy to read.
It is very easy to convert numbers from binary system to hexadecimal system and vice-versa, every nibble (4 bits) can be converted to a hexadecimal digit using this table:

Decimal (base 10)	Binary (base 2)	Hexadecimal (base 16)
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F



There is a convention to add **"h"** in the end of a hexadecimal number, this way we can determine that 5Fh is a hexadecimal number with decimal value of 95.
We also add **"0"** (zero) in the beginning of hexadecimal numbers that begin with a letter (A..F), for example **0E120h**.

The hexadecimal number **1234h** is equal to decimal value of 4660:

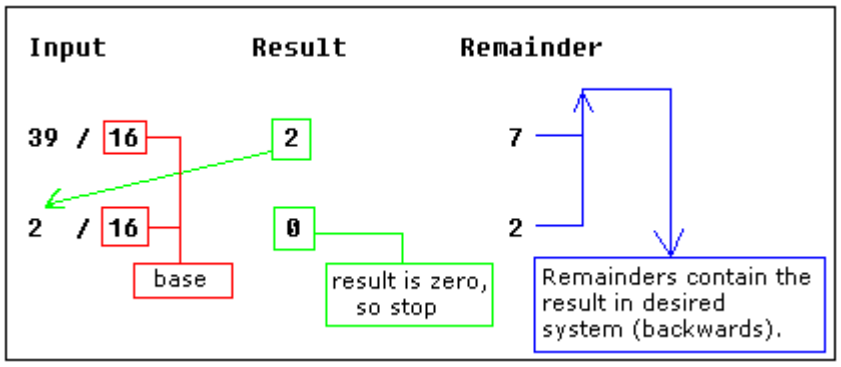


Converting from Decimal System to Any Other

In order to convert from decimal system, to any other system, it is required to divide the decimal value by the **base** of the desired system, each time you should remember the **result** and keep the **remainder**, the divide process continues until the **result** is zero.

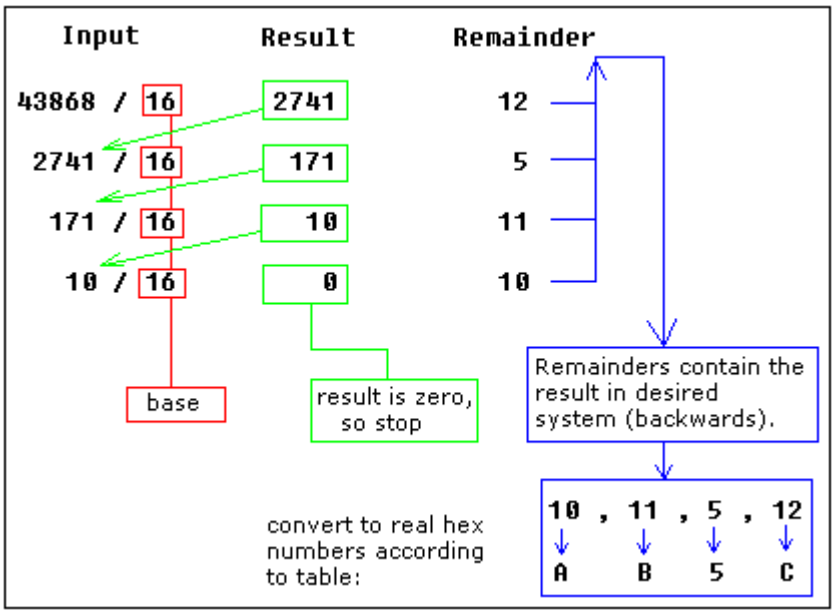
The **remainders** are then used to represent a value in that system.

Let's convert the value of **39** (base 10) to *Hexadecimal System* (base 16):



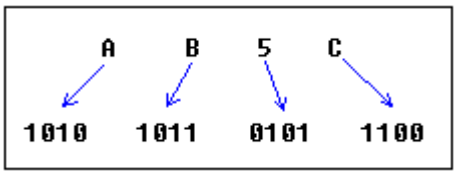
As you see we got this hexadecimal number: **27h**.
All remainders were below **10** in the above example, so we do not use any letters.

Here is another more complex example:
let's convert decimal number **43868** to hexadecimal form:



The result is **0AB5Ch**, we are using **the above table** to convert remainders over **9** to corresponding letters.

Using the same principle we can convert to binary form (using **2** as the divider), or convert to hexadecimal number, and then convert it to binary number using **the above table**:



As you see we got this binary number: **1010101101011100b**

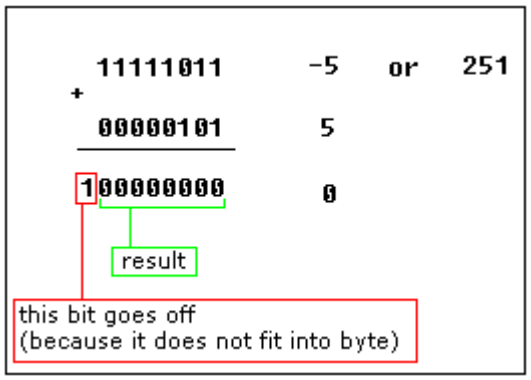
Signed Numbers

There is no way to say for sure whether the hexadecimal byte **0FFh** is positive or negative, it can represent both decimal value "**255**" and "**- 1**".

8 bits can be used to create **256** combinations (including zero), so we simply presume that first **128** combinations (**0..127**) will represent positive numbers and next **128** combinations (**128..256**) will represent negative numbers.

In order to get "**- 5**", we should subtract **5** from the number of combinations (**256**), so it we'll get: **256 - 5 = 251**.

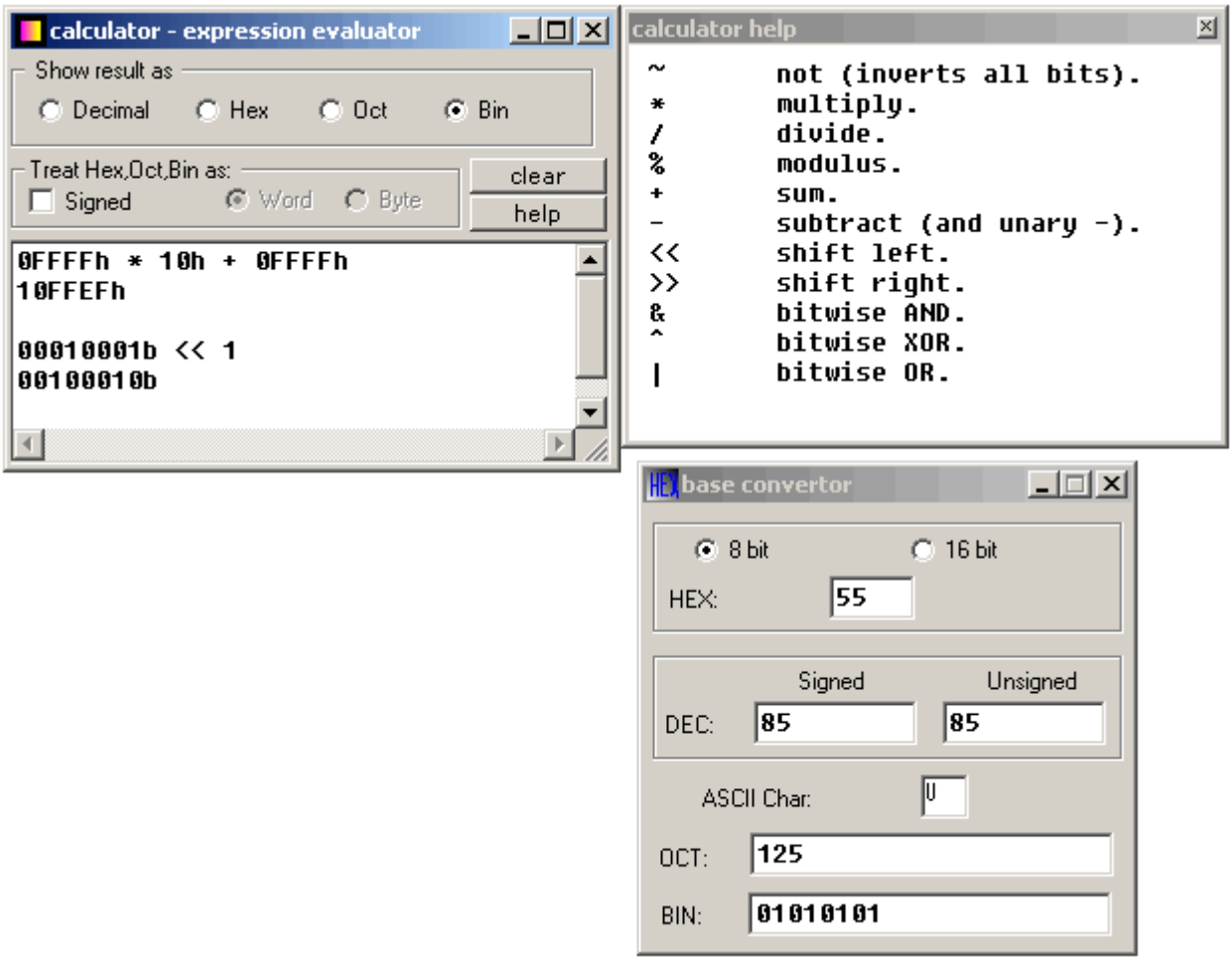
Using this complex way to represent negative numbers has some meaning, in math when you add "**- 5**" to "**5**" you should get zero. This is what happens when processor adds two bytes **5** and **251**, the result gets over **255**, because of the overflow processor gets zero!



When combinations **128..256** are used the high bit is always **1**, so this maybe used to determine the sign of a number.

The same principle is used for **words** (16 bit values), 16 bits create **65536** combinations, first 32768 combinations (**0..32767**) are used to represent positive numbers, and next 32768 combinations (**32767..65535**) represent negative numbers.

There are some handy tools in emu8086 to convert numbers, and make calculations of any numerical expressions, all you need is a click on **Math** menu:



Base converter allows you to convert numbers from any system and to any system. Just type a value in any text-box, and the value will be automatically converted to all other systems. You can work both with **8 bit** and **16 bit** values.

Multi base calculator can be used to make calculations between numbers in different systems and convert numbers from one system to another. Type an expression and press enter, result will appear in chosen numbering system. You can work with values up to **32 bits**. When **Signed** is checked evaluator assumes that all values (except decimal and double words) should be treated as **signed**. Double words are always treated as signed values, so **0FFFFFFFFh** is converted to **-1**.
For example you want to calculate: 0FFFFh * 10h + 0FFFFh (maximum memory location that can be accessed by 8086 CPU). If you check **Signed** and **Word** you will get -17 (because it is evaluated as (-1) * 16 + (-1) . To make calculation with unsigned values uncheck **Signed** so that the evaluation will be 65535 * 16 + 65535 and you should get 1114095.
You can also use the **base converter** to convert non-decimal digits

to signed decimal values, and do the calculation with decimal values (if it's easier for you).

These operation are supported:

~ not (inverts all bits).
* multiply.
/ divide.
% modulus.
+ sum.
- subtract (and unary -).
<< shift left.
>> shift right.
& bitwise AND.
^ bitwise XOR.
| bitwise OR.

Binary numbers must have "**b**" suffix, example:
00011011b

Hexadecimal numbers must have "**h**" suffix, and start with a zero when first digit is a letter (A..F), example:
0ABCDh

Octal (base 8) numbers must have "**o**" suffix, example:
77o

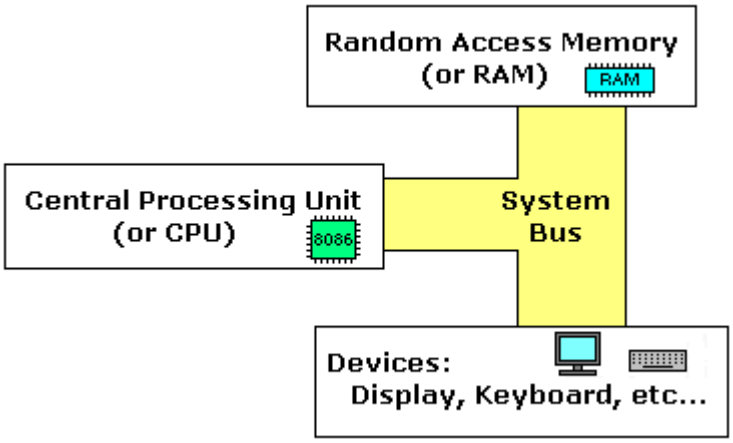
2) 8086 assembler tutorial for beginners (part 1)

this tutorial is intended for those who are not familiar with assembler at all, or have a very distant idea about it. of course if you have knowledge of some other programming language (basic, c/c++, pascal...) that may help you a lot.
but even if you are familiar with assembler, it is still a good idea to look through this document in order to study emu8086 syntax.

it is assumed that you have some knowledge about number representation (hex/bin), if not it is highly recommended to study **numbering systems tutorial** before you proceed.

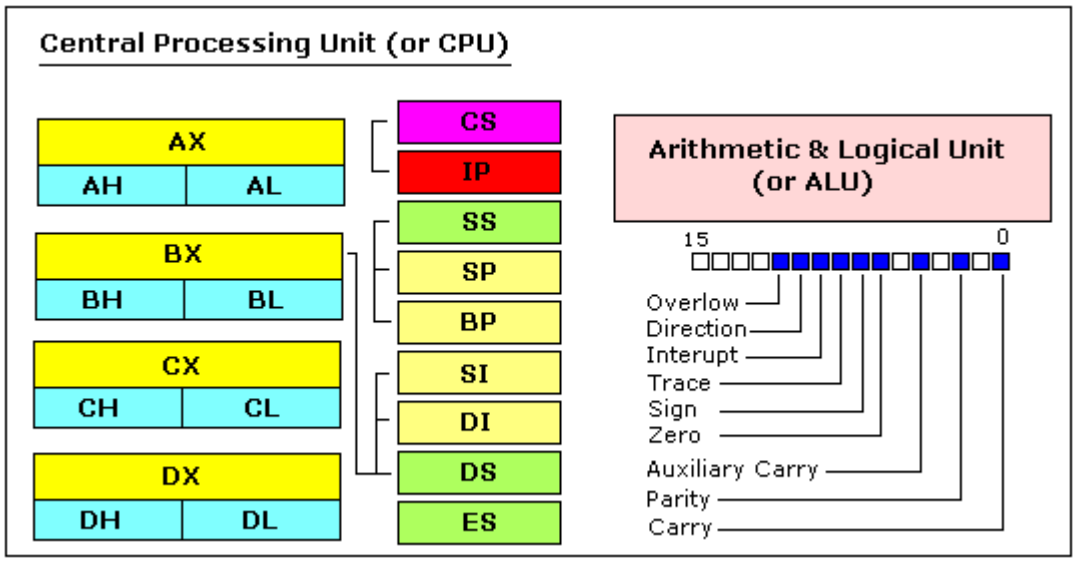
what is assembly language?

assembly language is a low level programming language. you need to get some knowledge about computer structure in order to understand anything. the simple computer model as i see it:



the **system bus** (shown in yellow) connects the various components of a computer.
the **CPU** is the heart of the computer, most of computations occur inside the **CPU**.
RAM is a place to where the programs are loaded in order to be executed.

inside the cpu



general purpose registers

8086 CPU has 8 general purpose registers, each register has its own name:

- **AX** - the accumulator register (divided into **AH / AL**).
- **BX** - the base address register (divided into **BH / BL**).
- **CX** - the count register (divided into **CH / CL**).
- **DX** - the data register (divided into **DH / DL**).
- **SI** - source index register.
- **DI** - destination index register.
- **BP** - base pointer.
- **SP** - stack pointer.

despite the name of a register, it's the programmer who determines the usage for each general purpose register. the main purpose of a register is to keep a number (variable). the size of the above registers is 16 bit, it's something like: **0011000000111001b** (in binary form), or **12345** in decimal (human) form.

4 general purpose registers (AX, BX, CX, DX) are made of two separate 8 bit registers, for example if AX= **0011000000111001b**, then AH=**00110000b** and AL=**00111001b**. therefore, when you modify any of the 8 bit registers 16 bit register is also updated, and vice-versa. the same is for other 3 registers, "H" is for high and "L" is for low part.

because registers are located inside the CPU, they are much faster than memory. Accessing a memory location requires the use of a system bus, so it takes much longer. Accessing data in a register usually takes no time. therefore, you should try to keep variables in the registers. register sets are very small and most registers have special purposes which limit their use as variables, but they are still an excellent place to store temporary data of calculations.

segment registers

- **CS** - points at the segment containing the current program.
- **DS** - generally points at segment where variables are defined.
- **ES** - extra segment register, it's up to a coder to define its usage.
- **SS** - points at the segment containing the stack.

although it is possible to store any data in the segment registers, this is never a good idea. the segment registers have a very special purpose - pointing at accessible blocks of memory.

segment registers work together with general purpose register to access any memory value. For example if we would like to access memory at the physical address **12345h** (hexadecimal), we should set the **DS = 1230h** and **SI = 0045h**. This is good, since this way we can access much more memory than with a single register that is limited to 16 bit values.
CPU makes a calculation of physical address by multiplying the segment register by 10h and adding general purpose register to it (1230h * 10h + 45h = 12345h):

```
 12300
+ 0045
-------
 12345
```

the address formed with 2 registers is called an **effective address**.
by default **BX, SI** and **DI** registers work with **DS** segment register; **BP** and **SP** work with **SS** segment register.
other general purpose registers cannot form an effective address!
also, although **BX** can form an effective address, **BH** and **BL** cannot.

special purpose registers

- **IP** - the instruction pointer.
- **flags register** - determines the current state of the microprocessor.

IP register always works together with **CS** segment register and it points to currently executing instruction.

flags register is modified automatically by CPU after mathematical operations, this allows to determine the type of the result, and to determine conditions to transfer control to other parts of the program.
generally you cannot access these registers directly, the way you can access AX and other general registers, but it is possible to change values of system registers using some tricks that you will learn a little bit later.

3) 8086 assembler tutorial for beginners (part 2)

Memory Access

---

to access memory we can use these four registers: **BX, SI, DI, BP**. combining these registers inside **[ ]** symbols, we can get different memory locations. these combinations are supported (addressing modes):

[BX + SI] [BX + DI] [BP + SI] [BP + DI]	[SI] [DI] d16 (variable offset only) [BX]	[BX + SI + d8] [BX + DI + d8] [BP + SI + d8] [BP + DI + d8]
[SI + d8] [DI + d8] [BP + d8] [BX + d8]	[BX + SI + d16] [BX + DI + d16] [BP + SI + d16] [BP + DI + d16]	[SI + d16] [DI + d16] [BP + d16] [BX + d16]

d8 - stays for 8 bit signed immediate displacement (for example: 22, 55h, -1, etc...)

d16 - stays for 16 bit signed immediate displacement (for example: 300, 5517h, -259, etc...).

displacement can be a immediate value or offset of a variable, or even both. if there are several values, assembler evaluates all values and calculates a single immediate value..

displacement can be inside or outside of the **[ ]** symbols, assembler generates the same machine code for both ways.

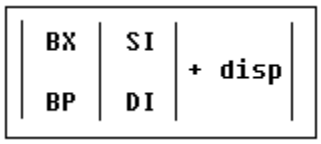
displacement is a **signed** value, so it can be both positive or negative.

generally the compiler takes care about difference between **d8** and **d16**, and generates the required machine code.

for example, let's assume that **DS = 100**, **BX = 30**, **SI = 70**.
The following addressing mode: **[BX + SI] + 25**
is calculated by processor to this physical address: **100 * 16 + 30 + 70 + 25 = 1725**.

by default **DS** segment register is used for all modes except those with **BP** register, for these **SS** segment register is used.

there is an easy way to remember all those possible combinations using this chart:



you can form all valid combinations by taking only one item from each column or skipping the column by not taking anything from it. as you see **BX** and **BP** never go together. **SI** and **DI** also don't go together. here are an examples of a valid addressing modes:
[BX+5] , **[BX+SI]** , **[DI+BX-4]**

---

the value in segment register (CS, DS, SS, ES) is called a **segment**, and the value in purpose register (BX, SI, DI, BP) is called an **offset**. When DS contains value **1234h** and SI contains the value **7890h** it can be also recorded as **1234:7890**. The physical address will be 1234h * 10h + 7890h = 19BD0h.

if zero is added to a decimal number it is multiplied by 10, however **10h = 16**, so if zero is added to a hexadecimal value, it is multiplied by 16, for example:

7h = 7
70h = 112

---

in order to say the compiler about data type,
these prefixes should be used:

byte ptr - for byte.
word ptr - for word (two bytes).

for example:

```
byte ptr [BX]     ; byte access.
   or
word ptr [BX]     ; word access.
```

assembler supports shorter prefixes as well:

b. - for **byte ptr**
w. - for **word ptr**

in certain cases the assembler can calculate the data type automatically.

MOV instruction

- copies the **second operand** (source) to the **first operand** (destination).
- the source operand can be an immediate value, general-purpose register or memory location.
- the destination register can be a general-purpose register, or memory location.
- both operands must be the same size, which can be a byte or a word.

these types of operands are supported:

MOV REG, memory
MOV memory, REG
MOV REG, REG
MOV memory, immediate
MOV REG, immediate
REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable, etc...

immediate: 5, -24, 3Fh, 10001101b, etc...

for segment registers only these types of **MOV** are supported:

MOV SREG, memory
MOV memory, SREG
MOV REG, SREG
MOV SREG, REG
SREG: DS, ES, SS, and only as second operand: CS.

REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable, etc...

The **MOV** instruction cannot be used to set the value of the **CS** and **IP** registers.

here is a short program that demonstrates the use of **MOV** instruction:

```
ORG 100h          ; this directive required for a simple 1 segment .com program.
MOV AX, 0B800h    ; set AX to hexadecimal value of B800h.
MOV DS, AX        ; copy value of AX to DS.
MOV CL, 'A'       ; set CL to ASCII code of 'A', it is 41h.
MOV CH, 1101_1111b ; set CH to binary value.
MOV BX, 15Eh      ; set BX to 15Eh.
MOV [BX], CX      ; copy contents of CX to memory at B800:015E
RET               ; returns to operating system.
```

you can **copy & paste** the above program to emu8086 code editor, and press [**Compile and Emulate**] button (or press **F5** key on your keyboard).

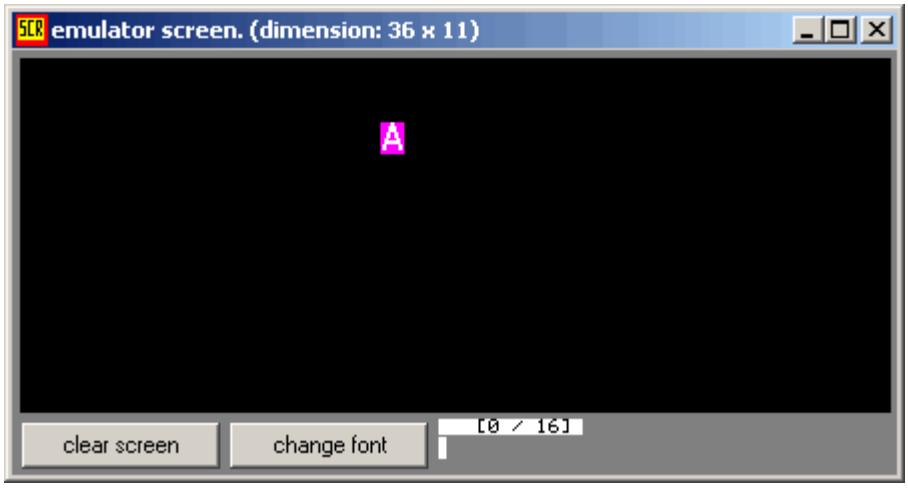
the emulator window should open with this program loaded, click [**Single Step**] button and watch the register values.

how to do **copy & paste**:

1. select the above text using mouse, click before the text and drag it down until everything is selected.
2. press **Ctrl + C** combination to copy.
3. go to emu8086 source editor and press **Ctrl + V** combination to paste.

as you may guess, "**;**" is used for comments, anything after "**;**" symbol is ignored by compiler.

you should see something like that when program finishes:



actually the above program writes directly to video memory, so you may see that **MOV** is a very powerful instruction

4) 8086 assembler tutorial for beginners (part 3)

Variables

Variable is a memory location. For a programmer it is much easier to have some value be kept in a variable named "**var1**" then at the address 5A73:235B, especially when you have 10 or more variables.

Our compiler supports two types of variables: **BYTE** and **WORD**.

Syntax for a variable declaration:

name **DB** *value*

name **DW** *value*

DB - stays for Define Byte.
DW - stays for Define Word.

name - can be any letter or digit combination, though it should start with a letter. It's possible to declare unnamed variables by not specifying the name (this variable will have an address but no name).

value - can be any numeric value in any supported numbering system (hexadecimal, binary, or decimal), or "**?**" symbol for variables that are not initialized.

As you probably know from *part 2* of this tutorial, **MOV** instruction is used to copy values from source to destination.
Let's see another example with **MOV** instruction:

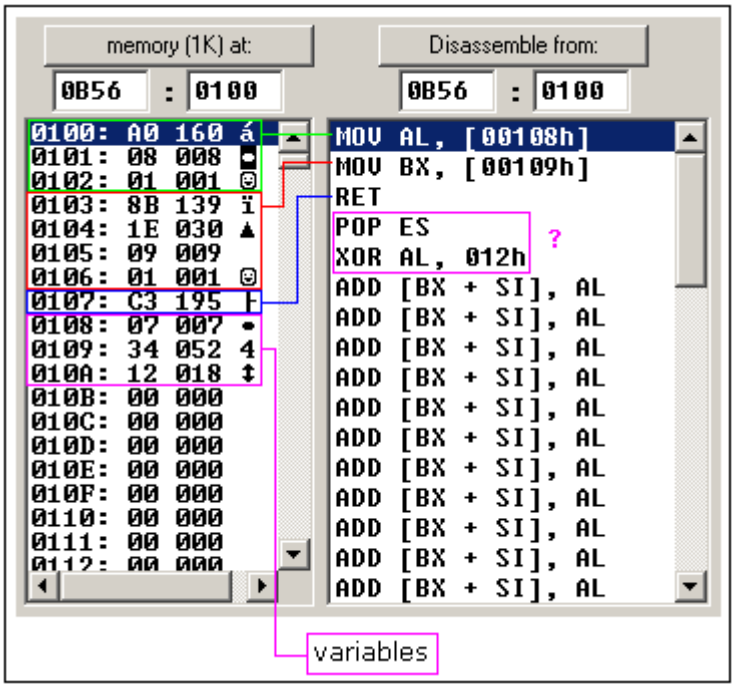
```
ORG 100h

MOV AL, var1
MOV BX, var2

RET    ; stops the program.

VAR1 DB 7
var2 DW 1234h
```

Copy the above code to emu8086 source editor, and press **F5** key to compile and load it in the emulator. You should get something like:



As you see this looks a lot like our example, except that variables are replaced with actual memory locations. When compiler makes machine code, it automatically replaces all variable names with their **offsets**. By default segment is loaded in **DS** register (when **COM** files is loaded the value of **DS** register is set to the same value as **CS** register - code segment).

In memory list first row is an **offset**, second row is a **hexadecimal value**, third row is **decimal value**, and last row is an **ASCII** character value.

Compiler is not case sensitive, so "**VAR1**" and "**var1**" refer to the same variable.

The offset of **VAR1** is **0108h**, and full address is **0B56:0108**.

The offset of **var2** is **0109h**, and full address is **0B56:0109**, this variable is a **WORD** so it occupies **2 BYTES**. It is assumed that low byte is stored at lower address, so **34h** is located before **12h**.

You can see that there are some other instructions after the **RET** instruction, this happens because disassembler has no idea about where the data starts, it just processes the values in memory and it understands them as valid 8086 instructions (we will learn them later).
You can even write the same program using **DB** directive only:

```
ORG 100h  ; just a directive to make a simple
.com file (expands into no code).

DB 0A0h
DB 08h
DB 01h

DB 8Bh
DB 1Eh
DB 09h
DB 01h

DB 0C3h

DB 7

DB 34h
DB 12h
```

Copy the above code to emu8086 source editor, and press **F5** key to compile and load it in the emulator. You should get the same disassembled code, and the same functionality!

As you may guess, the compiler just converts the program source to the set of bytes, this set is called **machine code**, processor understands the **machine code** and executes it.

ORG 100h is a compiler directive (it tells compiler how to handle the source code). This directive is very important when you work with variables. It tells compiler that the executable file will be loaded at the **offset** of 100h (256 bytes), so compiler should calculate the correct address for all variables when it replaces the variable names with their **offsets**. Directives are never converted to any real **machine code**.
Why executable file is loaded at **offset** of **100h**? Operating system keeps some data about the program in the first 256 bytes of the **CS** (code segment), such as command line parameters and etc.
Though this is true for **COM** files only, **EXE** files are loaded at offset of **0000**, and generally use special segment for variables. Maybe we'll talk more about **EXE** files later.

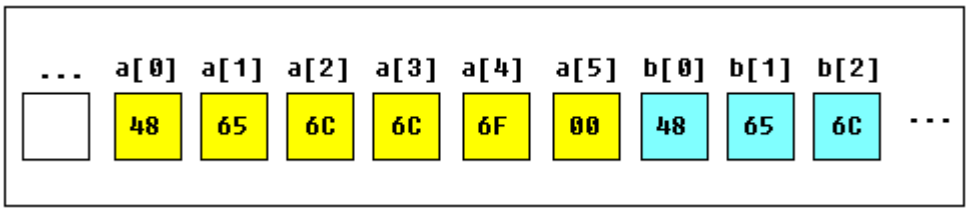
Arrays

Arrays can be seen as chains of variables. A text string is an example of a byte array, each character is presented as an ASCII code value (0..255).

Here are some array definition examples:

```
a DB 48h, 65h, 6Ch, 6Ch, 6Fh, 00h
b DB 'Hello', 0
```

b is an exact copy of the *a* array, when compiler sees a string inside quotes it automatically converts it to set of bytes. This chart shows a part of the memory where these arrays are declared:



You can access the value of any element in array using square brackets, for example:

```
MOV AL, a[3]
```

You can also use any of the memory index registers **BX, SI, DI, BP**, for example:

```
MOV SI, 3
MOV AL, a[SI]
```

If you need to declare a large array you can use **DUP** operator. The syntax for **DUP**:

number DUP (value(s))

number - number of duplicate to make (any constant value).
value - expression that DUP will duplicate.

for example:

```
c DB 5 DUP(9)
```

is an alternative way of declaring:

```
c DB 9, 9, 9, 9, 9
```

one more example:

```
d DB 5 DUP(1, 2)
```

is an alternative way of declaring:

```
d DB 1, 2, 1, 2, 1, 2, 1, 2, 1, 2
```

Of course, you can use **DW** instead of **DB** if it's required to keep values larger then 255, or smaller then -128. **DW** cannot be used to declare strings.

Getting the Address of a Variable

There is **LEA** (Load Effective Address) instruction and alternative **OFFSET** operator. Both **OFFSET** and **LEA** can be used to get the offset address of the variable.
LEA is more powerful because it also allows you to get the address of an indexed variables. Getting the address of the variable can be very useful in some situations, for example when you need to pass parameters to a procedure.

Reminder:

In order to tell the compiler about data type,
these prefixes should be used:

BYTE PTR - for byte.
WORD PTR - for word (two bytes).

For example:

```
BYTE PTR [BX]     ; byte access.
   or
WORD PTR [BX]     ; word access.
```

emu8086 supports shorter prefixes as well:

b. - for **BYTE PTR**
w. - for **WORD PTR**

in certain cases the assembler can calculate the data type automatically.

Here is first example:

```
ORG 100h

MOV    AL, VAR1            ; check value of
VAR1 by moving it to AL.

LEA    BX, VAR1            ; get address of VAR1
in BX.

MOV    BYTE PTR [BX], 44h    ; modify the
contents of VAR1.

MOV    AL, VAR1            ; check value of
VAR1 by moving it to AL.

RET

VAR1   DB  22h

END
```

Here is another example, that uses **OFFSET** instead of **LEA**:

```
ORG 100h

MOV    AL, VAR1            ; check value of
VAR1 by moving it to AL.

MOV    BX, OFFSET VAR1     ; get address of
VAR1 in BX.

MOV    BYTE PTR [BX], 44h  ; modify the
contents of VAR1.

MOV    AL, VAR1            ; check value of
VAR1 by moving it to AL.

RET

VAR1   DB  22h

END
```

Both examples have the same functionality.

These lines:
LEA BX, VAR1
MOV BX, OFFSET VAR1
are even compiled into the same machine code: MOV BX, num
num is a 16 bit value of the variable offset.

Please note that only these registers can be used inside square brackets (as memory pointers): **BX, SI, DI, BP**!
(see previous part of the tutorial).

Constants

Constants are just like variables, but they exist only until your program is compiled (assembled). After definition of a constant its value cannot be changed. To define constants **EQU** directive is used:

name EQU < any expression >

For example:

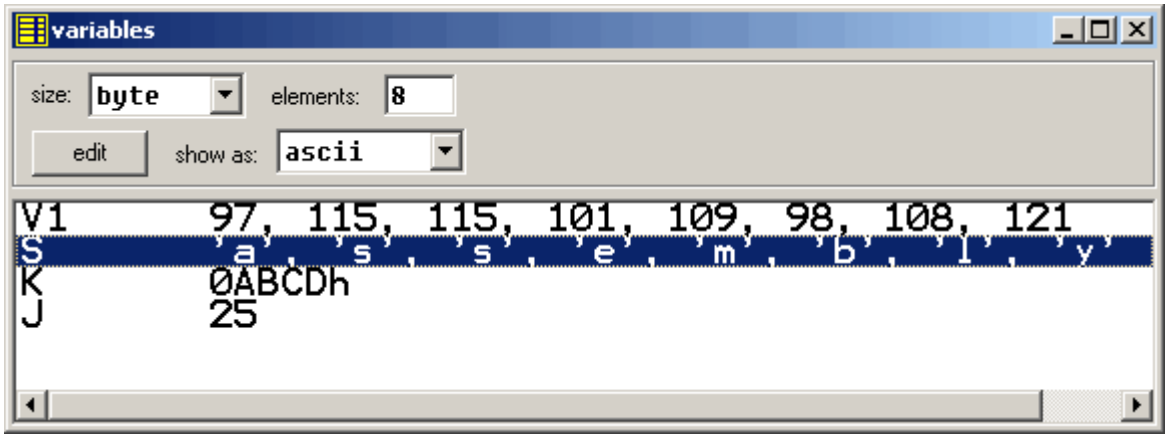
```
k EQU 5

MOV AX, k
```

The above example is functionally identical to code:

```
MOV AX, 5
```

You can view variables while your program executes by selecting "**Variables**" from the "**View**" menu of emulator.



To view arrays you should click on a variable and set **Elements** property to array size. In assembly language there are not strict data types, so any variable can be presented as an array.

Variable can be viewed in any numbering system:

- **HEX** - hexadecimal (base 16).
- **BIN** - binary (base 2).
- **OCT** - octal (base 8).
- **SIGNED** - signed decimal (base 10).
- **UNSIGNED** - unsigned decimal (base 10).
- **CHAR** - ASCII char code (there are 256 symbols, some symbols are invisible).

You can edit a variable's value when your program is running, simply double click it, or select it and click **Edit** button.

It is possible to enter numbers in any system, hexadecimal numbers should have "**h**" suffix, binary "**b**" suffix, octal "**o**" suffix, decimal numbers require no suffix. String can be entered this way:
'hello world', 0
(this string is zero terminated).

Arrays may be entered this way:
1, 2, 3, 4, 5
(the array can be array of bytes or words, it depends whether **BYTE** or **WORD** is selected for edited variable).

Expressions are automatically converted, for example:
when this expression is entered:
5 + 2
it will be converted to **7** etc...

5) 8086 assembler tutorial for beginners (part 4)

Interrupts

Interrupts can be seen as a number of functions. These functions make the programming much easier, instead of writing a code to print a character you can simply call the interrupt and it will do everything for you. There are also interrupt functions that work with disk drive and other hardware. We call such functions **software interrupts**.

Interrupts are also triggered by different hardware, these are called **hardware interrupts**. Currently we are interested in **software interrupts** only.

To make a **software interrupt** there is an **INT** instruction, it has very simple syntax:
INT value
Where **value** can be a number between 0 to 255 (or 0 to 0FFh), generally we will use hexadecimal numbers.
You may think that there are only 256 functions, but that is not correct. Each interrupt may have sub-functions.
To specify a sub-function **AH** register should be set before calling interrupt.
Each interrupt may have up to 256 sub-functions (so we get 256 * 256 = 65536 functions). In general **AH** register is used, but sometimes other registers maybe in use. Generally other registers are used to pass parameters and data to sub-function.

The following example uses **INT 10h** sub-function **0Eh** to type a "Hello!" message. This functions displays a character on the screen, advancing the cursor and scrolling the screen as necessary.

```
ORG    100h ; directive to make a simple .com
file.

; The sub-function that we are using
; does not modify the AH register on
; return, so we may set it only once.

MOV    AH, 0Eh    ; select sub-function.

; INT 10h / 0Eh sub-function
```

```
; receives an ASCII code of the
; character that will be printed
; in AL register.

MOV    AL, 'H'    ; ASCII code: 72
INT    10h        ; print it!

MOV    AL, 'e'    ; ASCII code: 101
INT    10h        ; print it!

MOV    AL, 'l'    ; ASCII code: 108
INT    10h        ; print it!

MOV    AL, 'l'    ; ASCII code: 108
INT    10h        ; print it!

MOV    AL, 'o'    ; ASCII code: 111
INT    10h        ; print it!

MOV    AL, '!'    ; ASCII code: 33
INT    10h        ; print it!

RET               ; returns to operating system.
```

Copy & paste the above program to emu8086 source code editor, and press [**Compile and Emulate**] button. Run it!

See **list of supported interrupts** for more information about interrupts.

6) 8086 assembler tutorial for beginners (part 5)

Library of common functions - emu8086.inc

To make programming easier there are some common functions that can be included in your program. To make your program use functions defined in other file you should use the **INCLUDE** directive followed by a file name. Compiler automatically searches for the file in the same folder where the source file is located, and if it cannot find the file there - it searches in **Inc** folder.

Currently you may not be able to fully understand the contents of the **emu8086.inc** (located in **Inc** folder), but it's OK, since you only need to understand what it can do.

To use any of the functions in **emu8086.inc** you should have the following line in the beginning of your source file:

```
include 'emu8086.inc'
```

---

emu8086.inc defines the following **macros**:

- **PUTC char** - macro with 1 parameter, prints out an ASCII char at current cursor position.
- **GOTOXY col, row** - macro with 2 parameters, sets cursor position.
- **PRINT string** - macro with 1 parameter, prints out a string.
- **PRINTN string** - macro with 1 parameter, prints out a string. The same as PRINT but automatically adds "carriage return" at the end of the string.
- **CURSOROFF** - turns off the text cursor.
- **CURSORON** - turns on the text cursor.

To use any of the above macros simply type its name somewhere in your code, and if required parameters, for example:

```
include emu8086.inc

ORG    100h

PRINT 'Hello World!'

GOTOXY 10, 5

PUTC 65          ; 65 - is an ASCII code for 'A'
PUTC 'B'

RET              ; return to operating system.
END              ; directive to stop the compiler.
```

When compiler process your source code it searches the **emu8086.inc** file for declarations of the macros and replaces the macro names with real code. Generally macros are relatively small parts of code, frequent use of a macro may make your executable too big (procedures are better for size optimization).

---

emu8086.inc also defines the following **procedures**:

- **PRINT_STRING** - procedure to print a null terminated string at current cursor position, receives address of string in **DS:SI** register. To use it declare: **DEFINE_PRINT_STRING** before **END** directive.
- **PTHIS** - procedure to print a null terminated string at current cursor position (just as PRINT_STRING), but receives address of string from Stack. The ZERO TERMINATED string should be defined just after the CALL instruction. For example:

  ```
  CALL PTHIS
  db 'Hello World!', 0
  ```

  To use it declare: **DEFINE_PTHIS** before **END** directive.
- **GET_STRING** - procedure to get a null terminated string from a user, the received string is written to buffer at **DS:DI**, buffer size should be in **DX**. Procedure stops the input when 'Enter' is pressed. To use it declare: **DEFINE_GET_STRING** before **END** directive.

- **CLEAR_SCREEN** - procedure to clear the screen, (done by scrolling entire screen window), and set cursor position to top of it. To use it declare: **DEFINE_CLEAR_SCREEN** before **END** directive.
- **SCAN_NUM** - procedure that gets the multi-digit SIGNED number from the keyboard, and stores the result in **CX** register. To use it declare: **DEFINE_SCAN_NUM** before **END** directive.
- **PRINT_NUM** - procedure that prints a signed number in **AX** register. To use it declare: **DEFINE_PRINT_NUM** and **DEFINE_PRINT_NUM_UNS** before **END** directive.
- **PRINT_NUM_UNS** - procedure that prints out an unsigned number in **AX** register. To use it declare: **DEFINE_PRINT_NUM_UNS** before **END** directive.

To use any of the above procedures you should first declare the function in the bottom of your file (but before the **END** directive), and then use **CALL** instruction followed by a procedure name. For example:

```
include 'emu8086.inc'

ORG    100h

LEA    SI, msg1       ; ask for the number
CALL   print_string   ;
CALL   scan_num       ; get number in CX.

MOV    AX, CX         ; copy the number to AX.

; print the following string:
CALL   pthis
DB  13, 10, 'You have entered: ', 0

CALL   print_num      ; print number in AX.

RET                   ; return to operating system.

msg1   DB  'Enter the number: ', 0

DEFINE_SCAN_NUM
DEFINE_PRINT_STRING
DEFINE_PRINT_NUM
DEFINE_PRINT_NUM_UNS  ; required for
print_num.
DEFINE_PTHIS

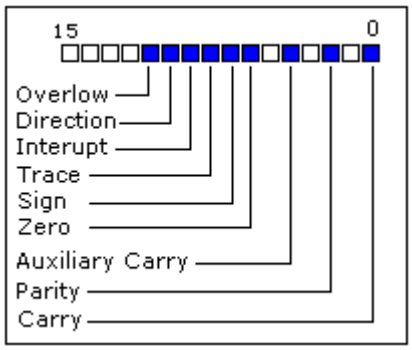
END                   ; directive to stop the compiler.
```

First compiler processes the declarations (these are just regular the macros that are expanded to procedures). When compiler gets to **CALL** instruction it replaces the procedure name with the address of the code where the procedure is declared. When **CALL** instruction is executed control is transferred to procedure. This is quite useful, since even if you call the same procedure 100 times in your code you will still have relatively small executable size. Seems complicated, isn't it? That's ok, with the time you will learn more, currently it's required that you understand the basic principle.

7) 8086 assembler tutorial for beginners (part 6)

Arithmetic and logic instructions

Most Arithmetic and Logic Instructions affect the processor status register (or **Flags**)



As you may see there are 16 bits in this register, each bit is called a **flag** and can take a value of **1** or **0**.

- **Carry Flag (CF)** - this flag is set to **1** when there is an **unsigned overflow**. For example when you add bytes **255 + 1** (result is not in range 0...255). When there is no overflow this flag is set to **0**.
- **Zero Flag (ZF)** - set to **1** when result is **zero**. For none zero result this flag is set to **0**.
- **Sign Flag (SF)** - set to **1** when result is **negative**. When result is **positive** it is set to **0**. Actually this flag take the value of the most significant bit.
- **Overflow Flag (OF)** - set to **1** when there is a **signed overflow**. For example, when you add bytes **100 + 50** (result is not in range -128...127).
- **Parity Flag (PF)** - this flag is set to **1** when there is even number of one bits in result, and to **0** when there is odd number of one bits. Even if result is a word only 8 low bits are analyzed!
- **Auxiliary Flag (AF)** - set to **1** when there is an **unsigned overflow** for low nibble (4 bits).

- **Interrupt enable Flag (IF)** - when this flag is set to **1** CPU reacts to interrupts from external devices.
- **Direction Flag (DF)** - this flag is used by some instructions to process data chains, when this flag is set to **0** - the processing is done forward, when this flag is set to **1** the processing is done backward.

There are 3 groups of instructions.

First group: **ADD**, **SUB**,**CMP**, **AND**, **TEST**, **OR**, **XOR**

These types of operands are supported:

REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate
REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable, etc...

immediate: 5, -24, 3Fh, 10001101b, etc...

After operation between operands, result is always stored in first operand. **CMP** and **TEST** instructions affect flags only and do not store a result (these instruction are used to make decisions during program execution).

These instructions affect these flags only:
CF, **ZF**, **SF**, **OF**, **PF**, **AF**.

- **ADD** - add second operand to first.
- **SUB** - Subtract second operand to first.
- **CMP** - Subtract second operand from first **for flags only**.
- **AND** - Logical AND between all bits of two operands. These rules apply:

  1 AND 1 = 1
  1 AND 0 = 0

0 AND 1 = 0
0 AND 0 = 0

As you see we get **1** only when both bits are **1**.

- **TEST** - The same as **AND** but **for flags only**.
- **OR** - Logical OR between all bits of two operands. These rules apply:

  1 OR 1 = 1
  1 OR 0 = 1
  0 OR 1 = 1
  0 OR 0 = 0

  As you see we get **1** every time when at least one of the bits is **1**.

- **XOR** - Logical XOR (exclusive OR) between all bits of two operands. These rules apply:

  1 XOR 1 = 0
  1 XOR 0 = 1
  0 XOR 1 = 1
  0 XOR 0 = 0

  As you see we get **1** every time when bits are different from each other.

Second group: **MUL**, **IMUL**, **DIV**, **IDIV**

These types of operands are supported:

REG
memory
REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable, etc...

MUL and **IMUL** instructions affect these flags only:
CF, **OF**
When result is over operand size these flags are set to **1**, when result fits in operand size these flags are set to **0**.

For **DIV** and **IDIV** flags are undefined.

- **MUL** - Unsigned multiply:

when operand is a **byte**:
AX = AL * operand.

when operand is a **word**:
(DX AX) = AX * operand.

- **IMUL** - Signed multiply:

  when operand is a **byte**:
  AX = AL * operand.

  when operand is a **word**:
  (DX AX) = AX * operand.

- **DIV** - Unsigned divide:

  when operand is a **byte**:
  AL = AX / operand
  AH = remainder (modulus). .

  when operand is a **word**:
  AX = (DX AX) / operand
  DX = remainder (modulus). .

- **IDIV** - Signed divide:

  when operand is a **byte**:
  AL = AX / operand
  AH = remainder (modulus). .

  when operand is a **word**:
  AX = (DX AX) / operand
  DX = remainder (modulus). .

Third group: **INC**, **DEC**, **NOT**, **NEG**

These types of operands are supported:

REG
memory
REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

memory: [BX], [BX+SI+7], variable, etc...

INC, **DEC** instructions affect these flags only:
ZF, **SF**, **OF**, **PF**, **AF**.

NOT instruction does not affect any flags!

NEG instruction affects these flags only:
CF, **ZF**, **SF**, **OF**, **PF**, **AF**.

- **NOT** - Reverse each bit of operand.
- **NEG** - Make operand negative (two's complement). Actually it reverses each bit of operand and then adds 1 to it. For example 5 will become -5, and -2 will become 2.

---

8) 8086 assembler tutorial for beginners (part 7)

program flow control

---

controlling the program flow is a very important thing, this is where your program can make decisions according to certain conditions.

- **unconditional jumps**

  The basic instruction that transfers control to another point in the program is **JMP**.

  The basic syntax of **JMP** instruction:

  JMP label

  To declare a *label* in your program, just type its name and add "**:**" to the end, label can be any character combination but it cannot start with a number, for example here are 3 legal label definitions:

  label1:
  label2:
  a:

  Label can be declared on a separate line or before any other instruction, for example:

  x1:
  MOV AX, 1

  x2: MOV AX, 2

  here's an example of **JMP** instruction:

```
org   100h

mov   ax, 5       ; set ax to 5.
mov   bx, 2       ; set bx to 2.

jmp   calc        ; go to 'calc'.

back: jmp stop    ; go to 'stop'.
```

```
calc:
add   ax, bx        ; add bx to ax.
jmp   back          ; go 'back'.

stop:

ret                 ; return to operating system.
```

Of course there is an easier way to calculate the some of two numbers, but it's still a good example of **JMP** instruction.
As you can see from this example **JMP** is able to transfer control both forward and backward. It can jump anywhere in current code segment (65,535 bytes).

- **Short Conditional Jumps**

  Unlike **JMP** instruction that does an unconditional jump, there are instructions that do a conditional jumps (jump only when some conditions are in act). These instructions are divided in three groups, first group just test single flag, second compares numbers as signed, and third compares numbers as unsigned.

  Jump instructions that test single flag

Instruction	Description	Condition	Opposite Instruction
JZ , JE	Jump if Zero (Equal).	ZF = 1	JNZ, JNE
JC , JB, JNAE	Jump if Carry (Below, Not Above Equal).	CF = 1	JNC, JNB, JAE
JS	Jump if Sign.	SF = 1	JNS
JO	Jump if Overflow.	OF = 1	JNO
JPE, JP	Jump if Parity Even.	PF = 1	JPO
JNZ , JNE	Jump if Not Zero (Not Equal).	ZF = 0	JZ, JE
JNC , JNB, JAE	Jump if Not Carry (Not Below, Above Equal).	CF = 0	JC, JB, JNAE
JNS	Jump if Not Sign.	SF = 0	JS
JNO	Jump if Not Overflow.	OF = 0	JO

JPO, JNP	Jump if Parity Odd (No Parity).	PF = 0	JPE, JP

- as you may already notice there are some instructions that do that same thing, that's correct, they even are assembled into the same machine code, so it's good to remember that when you compile **JE** instruction - you will get it disassembled as: **JZ**, **JC** is assembled the same as **JB** etc...
  different names are used to make programs easier to understand, to code and most importantly to remember. very offset dissembler has no clue what the original instruction was look like that's why it uses the most common name.

  if you emulate this code you will see that all instructions are assembled into **JNB**, the operational code (opcode) for this instruction is **73h** this instruction has fixed length of two bytes, the second byte is number of bytes to add to the **IP** register if the condition is true. because the instruction has only 1 byte to keep the offset it is limited to pass control to -128 bytes back or 127 bytes forward, this value is always signed.
-
- ```
    jnc a
    jnb a
    jae a

    mov ax, 4
  a: mov ax, 5
  ret
  ```
  ```

9) 8086 assembler tutorial for beginners (part 8)

Procedures

Procedure is a part of code that can be called from your program in order to make some specific task. Procedures make program more structural and easier to understand. Generally procedure returns to the same point from where it was called.

The syntax for procedure declaration:

```
name PROC

    ; here goes the code
    ; of the procedure ...

RET
name ENDP
```

name - is the procedure name, the same name should be in the top and the bottom, this is used to check correct closing of procedures.

Probably, you already know that **RET** instruction is used to return to operating system. The same instruction is used to return from procedure (actually operating system sees your program as a special procedure).

PROC and **ENDP** are compiler directives, so they are not assembled into any real machine code. Compiler just remembers the address of procedure.

CALL instruction is used to call a procedure.

Here is an example:

```
ORG   100h

CALL  m1

MOV   AX, 2

RET             ; return to operating system.

m1    PROC
MOV   BX, 5
RET             ; return to caller.
```

```
m1    ENDP

END
```

The above example calls procedure **m1**, does **MOV BX, 5**, and returns to the next instruction after **CALL**: **MOV AX, 2**.

There are several ways to pass parameters to procedure, the easiest way to pass parameters is by using registers, here is another example of a procedure that receives two parameters in **AL** and **BL** registers, multiplies these parameters and returns the result in **AX** register:

```
ORG   100h

MOV   AL, 1
MOV   BL, 2

CALL  m2
CALL  m2
CALL  m2
CALL  m2

RET             ; return to operating system.

m2    PROC
MUL   BL          ; AX = AL * BL.
RET             ; return to caller.
m2    ENDP

END
```

In the above example value of **AL** register is update every time the procedure is called, **BL** register stays unchanged, so this algorithm calculates **2** in power of **4**,
so final result in **AX** register is **16** (or 10h).

Here goes another example,
that uses a procedure to print a *Hello World!* message:

```
ORG   100h

LEA   SI, msg       ; load address of msg to SI.
```

```
CALL   print_me

RET                ; return to operating system.

;
============================================================
; this procedure prints a string, the string should be null
; terminated (have zero in the end),
; the string address should be in SI register:
print_me     PROC

next_char:
   CMP  b.[SI], 0    ; check for zero to stop
   JE   stop         ;

   MOV  AL, [SI]     ; next get ASCII char.

   MOV  AH, 0Eh      ; teletype function number.
   INT  10h          ; using interrupt to print a char in AL.

   ADD  SI, 1        ; advance index of string array.

   JMP  next_char    ; go back, and type another char.

stop:
RET                ; return to caller.
print_me     ENDP
;
============================================================

msg    DB  'Hello World!', 0   ; null terminated string.

END
```

"**b.**" - prefix before [SI] means that we need to compare bytes, not words. When you need to compare words add "**w.**" prefix instead. When one of the compared operands is a register it's not required because compiler knows the size of each register.

10) 8086 assembler tutorial for beginners (part 9)

The Stack

Stack is an area of memory for keeping temporary data. Stack is used by **CALL** instruction to keep return address for procedure, **RET** instruction gets this value from the stack and returns to that offset. Quite the same thing happens when **INT** instruction calls an interrupt, it stores in stack flag register, code segment and offset. **IRET** instruction is used to return from interrupt call.

We can also use the stack to keep any other data,
there are two instructions that work with the stack:

PUSH - stores 16 bit value in the stack.

POP - gets 16 bit value from the stack.

Syntax for **PUSH** instruction:

PUSH REG
PUSH SREG
PUSH memory
PUSH immediate
REG: AX, BX, CX, DX, DI, SI, BP, SP.

SREG: DS, ES, SS, CS.

memory: [BX], [BX+SI+7], 16 bit variable, etc...

immediate: 5, -24, 3Fh, 10001101b, etc...

Syntax for **POP** instruction:

POP REG
POP SREG
POP memory
REG: AX, BX, CX, DX, DI, SI, BP, SP.

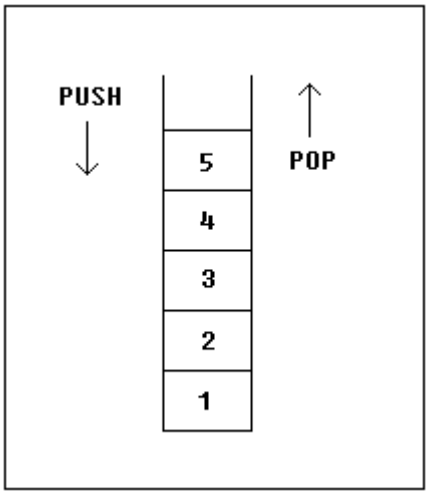
SREG: DS, ES, SS, (except CS).

memory: [BX], [BX+SI+7], 16 bit variable, etc...

Notes:

- **PUSH** and **POP** work with 16 bit values only!
- Note: **PUSH immediate** works only on 80186 CPU and later!

The stack uses **LIFO** (Last In First Out) algorithm,
this means that if we push these values one by one into the stack:
1, 2, 3, 4, 5
the first value that we will get on pop will be **5**, then **4**, **3**, **2**, and only then **1**.



It is very important to do equal number of **PUSH**s and **POP**s, otherwise the stack maybe corrupted and it will be impossible to return to operating system. As you already know we use **RET** instruction to return to operating system, so when program starts there is a return address in stack (generally it's 0000h).

PUSH and **POP** instruction are especially useful because we don't have too much registers to operate with, so here is a trick:

- Store original value of the register in stack (using **PUSH**).
- Use the register for any purpose.
- Restore the original value of the register from stack (using **POP**).

Here is an example:

```
ORG    100h

MOV    AX, 1234h
PUSH   AX          ; store value of AX in stack.

MOV    AX, 5678h   ; modify the AX value.

POP    AX          ; restore the original value of
AX.

RET

END
```

Another use of the stack is for exchanging the values, here is an example:

```
ORG    100h

MOV    AX, 1212h   ; store 1212h in AX.
MOV    BX, 3434h   ; store 3434h in BX


PUSH   AX          ; store value of AX in stack.
PUSH   BX          ; store value of BX in stack.

POP    AX          ; set AX to original value of BX.
POP    BX          ; set BX to original value of AX.

RET

END
```

The exchange happens because stack uses **LIFO** (Last In First Out) algorithm, so when we push **1212h** and then **3434h**, on pop we will first get **3434h** and only after it **1212h**.

The stack memory area is set by **SS** (Stack Segment) register, and **SP** (Stack Pointer) register. Generally operating system sets values of these registers on program start.

"**PUSH *source***" instruction does the following:

- Subtract **2** from **SP** register.
- Write the value of ***source*** to the address **SS:SP**.

"**POP *destination***" instruction does the following:

- Write the value at the address **SS:SP** to ***destination***.
- Add **2** to **SP** register.

The current address pointed by **SS:SP** is called **the top of the stack**.

For **COM** files stack segment is generally the code segment, and stack pointer is set to value of **0FFFEh**. At the address **SS:0FFFEh** stored a return address for **RET** instruction that is executed in the end of the program.

You can visually see the stack operation by clicking on [**Stack**] button on emulator window. The top of the stack is marked with "**<**" sign.

11) 8086 assembler tutorial for beginners (part 10)

Macros

Macros are just like procedures, but not really. Macros look like procedures, but they exist only until your code is compiled, after compilation all macros are replaced with real instructions. If you declared a macro and never used it in your code, compiler will simply ignore it. **emu8086.inc** is a good example of how macros can be used, this file contains several macros to make coding easier for you.

```
Macro definition:

name    MACRO  [parameters,...]

          <instructions>

ENDM
```

Unlike procedures, macros should be defined above the code that uses it, for example:

```
MyMacro    MACRO  p1, p2, p3

    MOV AX, p1
    MOV BX, p2
    MOV CX, p3

ENDM

ORG 100h

MyMacro 1, 2, 3

MyMacro 4, 5, DX

RET
```

The above code is expanded into:

```
MOV AX, 00001h
MOV BX, 00002h
MOV CX, 00003h
MOV AX, 00004h
MOV BX, 00005h
```

```
MOV CX, DX
```

Some important facts about **macros** and **procedures**:

- When you want to use a procedure you should use **CALL** instruction, for example:

  ```
  CALL MyProc
  ```

- When you want to use a macro, you can just type its name. For example:

  ```
  MyMacro
  ```

- Procedure is located at some specific address in memory, and if you use the same procedure 100 times, the CPU will transfer control to this part of the memory. The control will be returned back to the program by **RET** instruction. The **stack** is used to keep the return address. The **CALL** instruction takes about 3 bytes, so the size of the output executable file grows very insignificantly, no matter how many time the procedure is used.
- Macro is expanded directly in program's code. So if you use the same macro 100 times, the compiler expands the macro 100 times, making the output executable file larger and larger, each time all instructions of a macro are inserted.
- You should use **stack** or any general purpose registers to pass parameters to procedure.
- To pass parameters to macro, you can just type them after the macro name. For example:

  ```
  MyMacro 1, 2, 3
  ```

- To mark the end of the macro **ENDM** directive is enough.
- To mark the end of the procedure, you should type the name of the procedure before the **ENDP** directive.

Macros are expanded directly in code, therefore if there are labels inside the macro definition you may get "Duplicate declaration" error when macro is used for twice or more. To avoid such problem, use **LOCAL** directive followed by names of variables, labels or procedure names. For example:

```
MyMacro2    MACRO
            LOCAL label1, label2

            CMP  AX, 2
            JE label1
            CMP  AX, 3
            JE label2
            label1:
                        INC  AX
            label2:
                        ADD  AX, 2
ENDM


ORG 100h

MyMacro2

MyMacro2

RET
```

If you plan to use your macros in several programs, it may be a good idea to place all macros in a separate file. Place that file in **Inc** folder and use **INCLUDE *file-name*** directive to use macros. See **Library of common functions - emu8086.inc** for an example of such file.

12) 8086 assembler tutorial for beginners (part 11)

making your own operating system

Usually, when a computer starts it will try to load the first 512-byte sector (that's Cylinder **0**, Head **0**, Sector **1**) from any diskette in your **A:** drive to memory location **0000h:7C00h** and give it control. If this fails, the BIOS tries to use the MBR of the first hard drive instead.

This tutorial covers booting up from a floppy drive, the same principles are used to boot from a hard drive. But using a floppy drive has several advantages:

- you can keep your existing operating system intact (windows, dos, linux, unix, be-os...).
- it is easy and safe to modify the boot record of a floppy disk.

example of a simple floppy disk boot program:

```
; directive to create BOOT file:
#make_boot#

; Boot record is loaded at 0000:7C00,
; so inform compiler to make required
; corrections:
ORG 7C00h

PUSH    CS   ; make sure DS=CS
POP     DS

; load message address into SI register:
LEA SI, msg

; teletype function id:
MOV AH, 0Eh

print:   MOV AL, [SI]
         CMP AL, 0
         JZ done
         INT 10h   ; print using teletype.
         INC SI
         JMP print

; wait for 'any key':
done:      MOV AH, 0
```

```
        INT 16h


; store magic value at 0040h:0072h:
;   0000h - cold boot.
;   1234h - warm boot.
MOV     AX, 0040h
MOV     DS, AX
MOV     w.[0072h], 0000h ; cold boot.

JMP         0FFFFh:0000h            ; reboot!


new_line EQU 13, 10

msg DB  'Hello This is My First Boot Program!'
   DB  new_line, 'Press any key to reboot', 0
```

copy the above example to the source editor and press **emulate**. the emulator automatically loads **.bin** file to **0000h:7C00h** (it uses supplementary .binf file to know where to load).

you can run it just like a regular program, or you can use the **virtual drive** menu to **write 512 bytes at 7c00h to boot sector** of a virtual floppy drive (it's "**FLOPPY_0**" file in c:\emu8086). after your program is written to the virtual floppy drive, you can select **boot from floppy** from **virtual drive** menu.

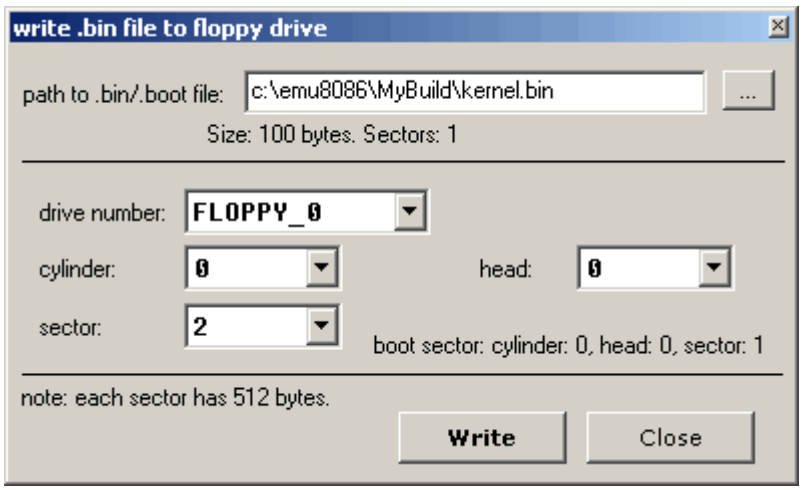
.bin files for boot records are limited to 512 bytes (sector size). if your new operating system is going to grow over this size, you will need to use a boot program to load data from other sectors (just like *micro-os_loader.asm* does). an example of a tiny operating system can be found in c:\emu8086\examples and "**online**":

micro-os_loader.asm
micro-os_kernel.asm

To create extensions for your Operating System (over 512 bytes), you can use additional sectors of a floppy disk. It's recommended to use "**.bin**" files for this purpose (to create "**.bin**" file select "**BIN Template**" from "**File**" -> "**New**" menu).

To write "**.bin**" file to virtual floppy, select **"Write .bin file to floppy..."** from **"Virtual drive"** menu of emulator, you should write it anywhere but the boot sector (which is Cylinder: **0**, Head: **0**,

Sector: **1**).



you can use this utility to write **.bin** files to virtual floppy disk ("**FLOPPY_0**" file), instead of "**write 512 bytes at 7c00h to boot sector**" menu. however, you should remember that **.bin** file that is designed to be a boot record should always be written to cylinder: **0**, head: **0**, sector: **1**

Boot Sector Location:
Cylinder: 0
Head: 0
Sector: 1

to write .bin files to real floppy disk use writebin.asm, just compile it to com file and run it from command prompt. to write a boot record type: **writebin loader.bin** ; to write kernel module type: **writebin kernel.bin /k**
/k - parameter tells the program to write the file at sector 2 instead of sector 1. it does not matter in what order you write the files onto floppy drive, but it does matter where you write them.

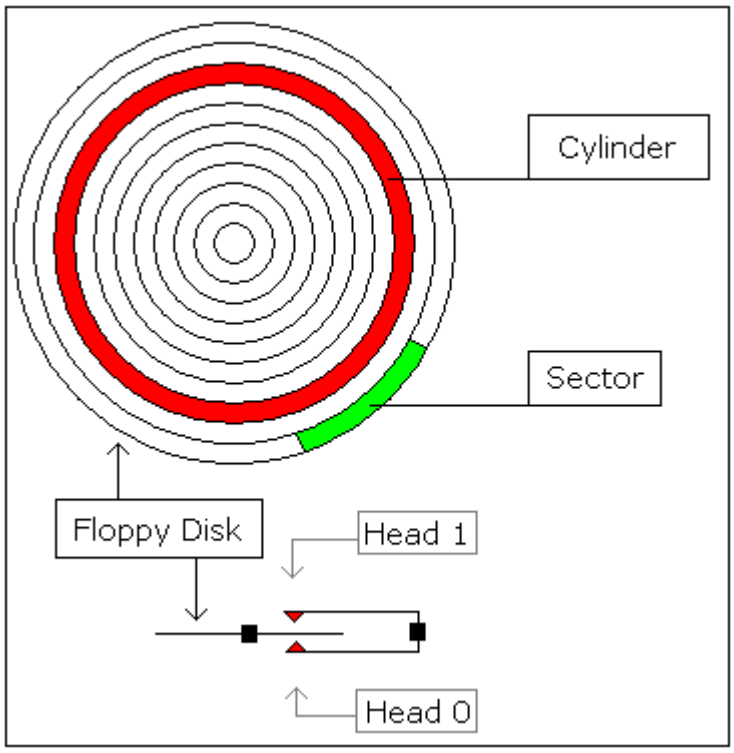
---

mote: this boot record is not MS-DOS/Windows compatible boot sector, it's not even Linux or Unix compatible, operating system may not allow you to read or write files on this diskette until you re-format it, therefore make sure the diskette you use doesn't contain any important information. however you can write and read anything to and from this disk using low level disk access interrupts, it's even possible to protect valuable information from the others this way; even if someone gets the disk he will probably think that it's empty and will reformat it because it's the default option in windows

operating system... such a good type of self destructing data carrier :)

---

idealized floppy drive and diskette structure:



for a **1440 kb** diskette:

- floppy disk has 2 sides, and there are 2 heads; one for each side (**0..1**), the drive heads move above the surface of the disk on each side.
- each side has 80 cylinders (numbered **0..79**).
- each cylinder has 18 sectors (**1..18**).
- each sector has **512** bytes.
- total size of floppy disk is: 2 x 80 x 18 x 512 = **1,474,560** bytes.

note: the MS-DOS (windows) formatted floppy disk has slightly less free space on it (by about 16,896 bytes) because the operating system needs place to store file names and directory structure (often called FAT or file system allocation table). more file names - less disk space. the most efficient way to store files is to write them directly to

sectors instead of using file system, and in some cases it is also the most reliable way, if you know how to use it.

to read sectors from floppy drive use **INT 13h / AH = 02h**.

13) 8086 assembler tutorial for beginners (part 12)

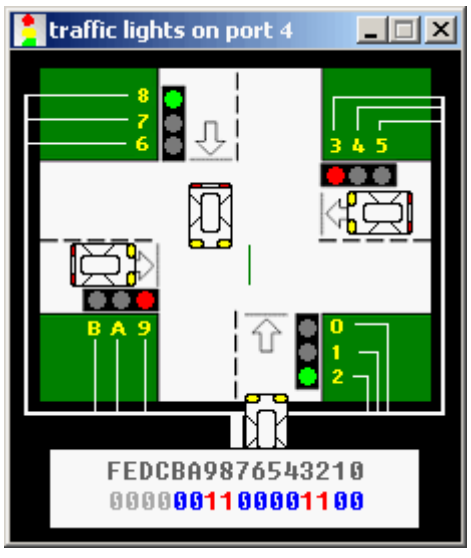
Controlling External Devices

There are 7 devices attached to the emulator: traffic lights, stepper-motor, LED display, thermometer, printer, robot and simple test device. You can view devices when you click "**Virtual Devices**" menu of the emulator.

For technical information refer to **I/O ports** section of emu8086 reference.

In general, it is possible to use any x86 family CPU to control all kind of devices, the difference maybe in base I/O port number, this can be altered using some tricky electronic equipment. Usually the "**.bin**" file is written into the Read Only Memory (ROM) chip, the system reads program from that chip, loads it in RAM module and runs the program. This principle is used for many modern devices such as micro-wave ovens and etc...

Traffic Lights



Usually to control the traffic lights an array (table) of values is used. In certain periods of time the value is read from the array and sent to a port. For example:

```
; controlling external device with 8086 microprocessor.
```

```
; realistic test for c:\emu8086\devices\Traffic_Lights.exe

#start=Traffic_Lights.exe#

name "traffic"


mov ax, all_red
out 4, ax


mov si, offset situation


next:
mov ax, [si]
out 4, ax

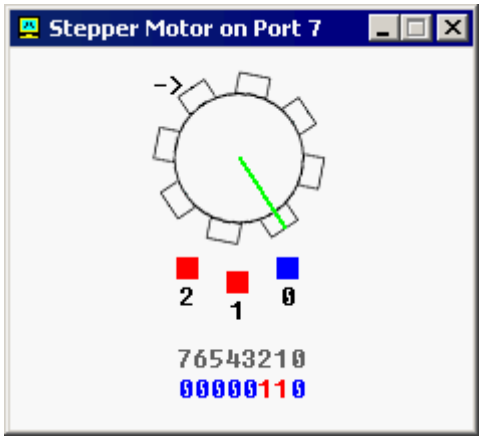
; wait 5 seconds (5 million microseconds)
mov     cx, 4Ch    ;    004C4B40h = 5,000,000
mov     dx, 4B40h
mov     ah, 86h
int     15h


add si, 2 ; next situation
cmp si, sit_end
jb  next
mov si, offset situation
jmp next


;                   FEDC_BA98_7654_3210
situation       dw      0000_0011_0000_1100b
s1              dw      0000_0110_1001_1010b
s2              dw      0000_1000_0110_0001b
s3              dw      0000_1000_0110_0001b
s4              dw      0000_0100_1101_0011b
sit_end = $


all_red         equ     0000_0010_0100_1001b
```

Stepper-Motor



The motor can be half stepped by turning on pair of magnets, followed by a single and so on.

The motor can be full stepped by turning on pair of magnets, followed by another pair of magnets and in the end followed by a single magnet and so on. The best way to make full step is to make two half steps.

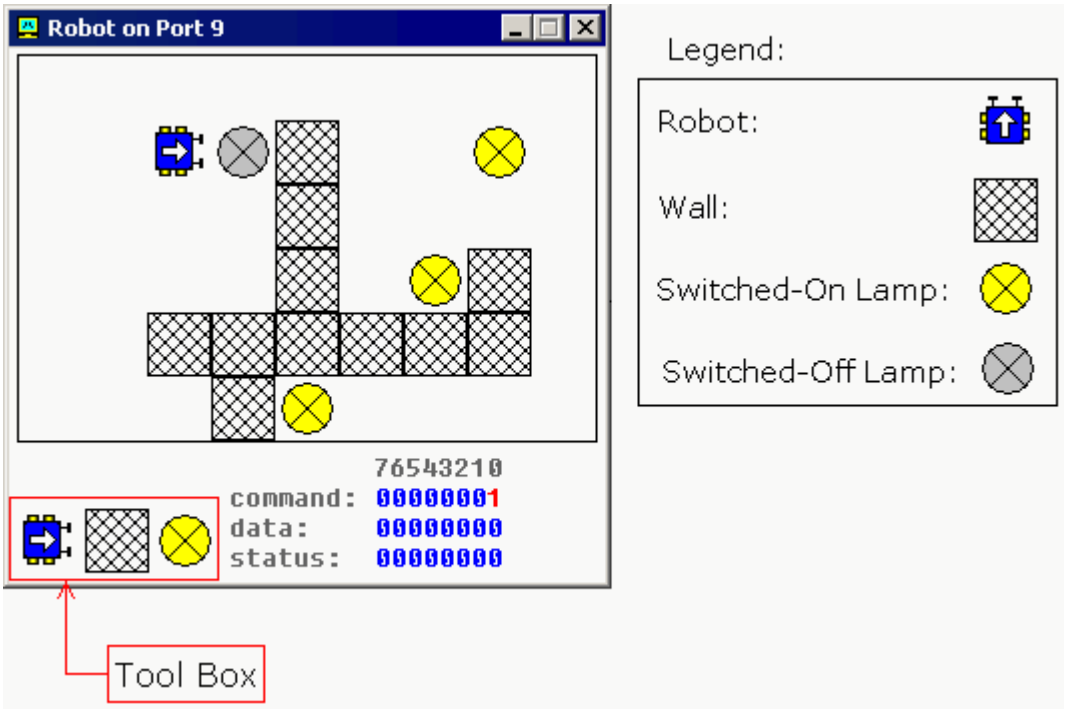
Half step is equal to **11.25** degrees.
Full step is equal to **22.5** degrees.

The motor can be turned both clock-wise and counter-clock-wise.

See **stepper_motor.asm** in c:\emu8086\examples\

See also **I/O ports** section of emu8086 reference.

Robot



Complete list of robot instruction set is given in **I/O ports** section of emu8086 reference.

To control the robot a complex algorithm should be used to achieve maximum efficiency. The simplest, yet very inefficient, is random moving algorithm, see **robot.asm** in c:\emu8086\examples\

It is also possible to use a data table (just like for Traffic Lights), this can be good if robot always works in the same surroundings.

14) Source Code Editor

Using the Mouse

Editor supports the following mouse actions:

Mouse Action	Result
L-Button click over text	Changes the caret position
R-Button click	Displays the right click menu
L-Button down over selection, and drag	Moves text
Ctrl + L-Button down over selection, and drag	Copies text
L-Button click over left margin	Selects line
L-Button click over left margin, and drag	Selects multiple lines
Alt + L-Button down, and drag	Select columns of text
L-Button double click over text	Select word under cursor
Spin IntelliMouse mouse wheel	Scroll the window vertically
Single click IntelliMouse mouse wheel	Select the word under the cursor
Double click IntelliMouse mouse wheel	Select the line under the cursor
Click and drag splitter bar	Split the window into multiple views or adjust the current splitter position
Double click splitter bar	Split the window in half into multiple views or unsplit the window if already split

Editor Hot Keys:

Command	Keystroke
Toggle Bookmark	Control + F2
Next Bookmark	F2
Prev Bookmark	Shift + F2
Copy	Control + C, Control + Insert
Cut	Control + X, Shift + Delete, Control + Alt + W
Cut Line	Control + Y
Cut Sentence	Control + Alt + K
Paste	Control + V, Shift + Insert
Undo	Control + Z, Alt + Backspace
Document End	Control + End
Document End Extend	Control + Shift + End
Document Start	Control + Home
Document Start Extend	Control + Shift + Home
Find	Control + F, Alt + F3
Find Next	F3
Find Next Word	Control + F3
Find Prev	Shift + F3
Find Prev Word	Control + Shift + F3
Find and Replace	Control + H, Control + Alt + F3
Go To Line	Control + G
Go To Match Brace	Control +]
Select All	Control + A
Select Line	Control + Alt + F8
Select Swap Anchor	Control + Shift + X
Insert New Line Above	Control + Shift + N
Indent Selection	Tab
Outdent Selection	Shift + Tab
Tabify Selection	Control + Shift + T
Untabify Selection	Control + Shift + Space
Lowercase Selection	Control + L
Uppercase Selection	Control + U, Control + Shift + U
Left Word	Control + Left
Right Word	Control + Right
Left Sentence	Control + Alt + Left
Right Sentence	Control + Alt + Right
Toggle Overtype	Insert
Display Whitespace	Control + Alt + T
Scroll Window Up	Control + Down
Scroll Window Down	Control + Up
Scroll Window Left	Control + PageUp
Scroll Window Right	Control + PageDown

Delete Word To End Control + Delete
Delete Word To Start Control + Backspace

Extend Char Left Shift + Left
Extend Char Right Shift + Right
Extend Left Word Control + Shift + Left
Extend Right Word Control + Shift + Right
Extend to Line Start Shift + Home
Extend to Line End Shift + End
Extend Line Up Shift + Up
Extend Line Down Shift + Down
Extend Page Up Shift + PgUp
Extend Page Down Shift + Next

Comment Block Ctrl + Q
Uncomment Block Ctrl + W

Regular Expression Syntax Rules for Search and Replace

Wildcards:
? (for any character),
+ (for one or more ot something),
* (for zero or more of something).

Sets of characters:
Characters enclosed in square brackets
will be treated as an option set.

Character ranges may be specified
with a - (e.g. [a-c]).

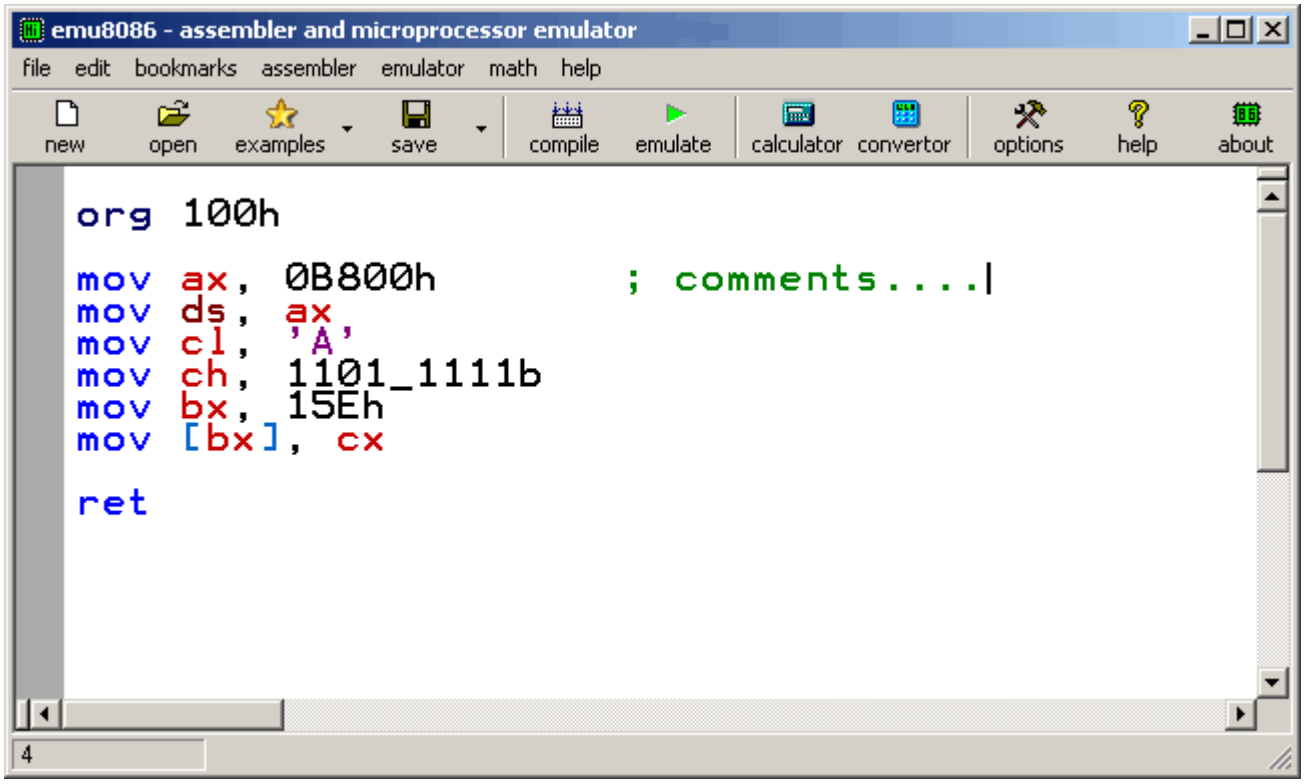
Logical OR:
Subexpressions may be ORed together
with the | pipe symbol.

Parenthesized subexpressions:
A regular expression may be enclosed
within parentheses and will be treated as a unit.

Escape characters:
Sequences such as:
\t - tab
etc.
will be substituted for an equivalent
single character. \\ represents the backslash.

If there are problems with the source editor you may need to manually copy "**cmax20.ocx**" from program's folder into **Windows\System** or **Windows\System32** replacing any existing version of that file (restart may be required before system allows to replace existing file).

15)compiling the assembly code



type your code inside the text area, and click **compile** button. you will be asked for a place where to save the compiled file.
after successful compilation you can click **emulate** button to load the compiled file in emulator.

---

the output file type directives:

```
#make_com#
#make_bin#
#make_boot#
#make_exe#
```

you can insert these directives in the source code to specify the required output type for the file. only if compiler cannot determine the output type automatically and it when it cannot find any of these directives it may ask you for *output type* before creating the file.

there is virtually no difference between how .com and .bin are assembled because these files are raw binary files, but .exe file has a special header in the beginning of the file that is used by the operating system to determine some properties of the executable file.

---

description of output file types:

- **#make_com#** - the oldest and the simplest format of an executable file, such files are loaded with 100h prefix (256 bytes). Select **Clean** from the **New** menu if you plan to compile a COM file. Compiler directive **ORG 100h** should be added before the code. Execution always starts from the first byte of the file.
  This file type is selected automatically if **org 100h** directive is found in the code.
  Supported by DOS and Windows Command Prompt.

- **#make_exe#** - more advanced format of an executable file. not limited by size and number of segments. stack segment should be defined in the program. you may select **exe template** from the **new** menu in to create a simple exe program with pre-defined data, stack, and code segments. the entry point (where execution starts) is defined by a programmer. this file type is selected automatically if **stack** segment is found.
  supported by dos and windows command prompt.

- **#make_bin#** - a simple executable file. You can define the values of all registers, segment and offset for memory area where this file will be loaded. When loading "**MY.BIN**" file to emulator it will look for a "**MY.BINF**" file, and load "**MY.BIN**" file to location specified in "**MY.BINF**" file, registers are also set using information in that file (open this file in a text editor to edit or investigate).
  in case the emulator is not able to find "**MY.BINF**" file, current register values are used and "**MY.BIN**" file is loaded at current **CS:IP**.
  the execution starts from values in **CS:IP**.
  bin file type is not unique to the emulator, however the directives are unique and will not work if .bin file is executed outside of the emulator because their output is stored in a separate file independently from pure binary code.

  .BINF file is created automatically if assembler finds any of the following directives.

  these directives can be inserted into any part of the source

code to preset registers or memory before starting the program's execution:

```
#make_bin#
#LOAD_SEGMENT=1234#
#LOAD_OFFSET=0000#
#AL=12#
#AH=34#
#BH=00#
#BL=00#
#CH=00#
#CL=00#
#DH=00#
#DL=00#
#DS=0000#
#ES=0000#
#SI=0000#
#DI=0000#
#BP=0000#
#CS=1234#
#IP=0000#
#SS=0000#
#SP=0000#
#MEM=0100:FFFE,00FF-0100:FF00,F4#
```

- all values must be in hexadecimal.

  when not specified these values are set by default:
  LOAD_SEGMENT = 0100
  LOAD_OFFSET = 0000
  CS = ES = SS = DS = 0100
  IP = 0000

  if **LOAD_SEGMENT** and **LOAD_OFFSET** are not defined, then **CS** and **IP** values are used and vice-versa.

  "#mem=..." directive can be used to write values to memory before program starts
  #MEM=nnnn,[bytestring]-nnnn:nnnn,[bytestring]#
  for example:
  #MEM=1000,01ABCDEF0122-0200,1233#
  all values are in hex, nnnn - for physical address, or (nnnn:nnnn) for logical address.
  - separates the entries. spaces are allowed inside.

  note: all values are in hex. hexadecimal suffix/prefix is not required. for each byte there must be exactly 2 characters, for example: 0A, 12 or 00.

if none of the above directives directives are preset in source code, binf file is not created.
when emulator loads .bin file without .binf file it will use **c:\emu8086\default.binf** instead.
this also applies to any other files with extensions that are unfamiliar to the emulator.

- ---
-

the format of a typical "**.BINF**" file:

```
8000        ; load to segment.
0000        ; load to offset.
55          ;  AL
66          ;  AH
77          ;  BL
88          ;  BH
99          ;  CL
AA          ;  CH
BB          ;  DL
CC          ;  DH
DDEE        ;  DS
ABCD        ;  ES
EF12        ;  SI
3456        ;  DI
7890        ;  BP
8000        ;  CS
0000        ;  IP
C123        ;  SS
D123        ;  SP
```

-

we can observe that first goes a number in hexadecimal form and then a comment.
Comments are added just to make some order, when emulator loads a **BINF** file it does not care about comments it just looks for a values on specific lines, so line order is very important.
NOTE: existing .binf file is automatically overwritten on re-compile.

- ---
-

In case **load to offset** value is not zero (0000), **ORG ????h** should be added to the source of a **.BIN** file where **????h** is the *loading offset*, this should be done to allow compiler calculate correct addresses.

- **#make_boot#** - this type is a copy of the first track of a floppy disk (boot sector). the only difference from #make_bin#

is that loading segment is predefined to 0000:7c00h (this value is written to accompanied .binf file). in fact you can use #make_bin# without any lack of performance, however to make correct test in emulator you will need to add these directives: #cs=0# and #ip=7c00# - assembler writes these values into .binf file.
You can write a boot sector of a virtual floppy (FLOPPY_0) via menu in emulator:
[virtual drive] -> [write 512 bytes at 7c00 to boot sector]
First you should compile a **.bin** file and load it in emulator (see "**micro-os_loader.asm**" and "**micro-os_kernel.asm**" in "c:\emu8086\examples\" for more information).

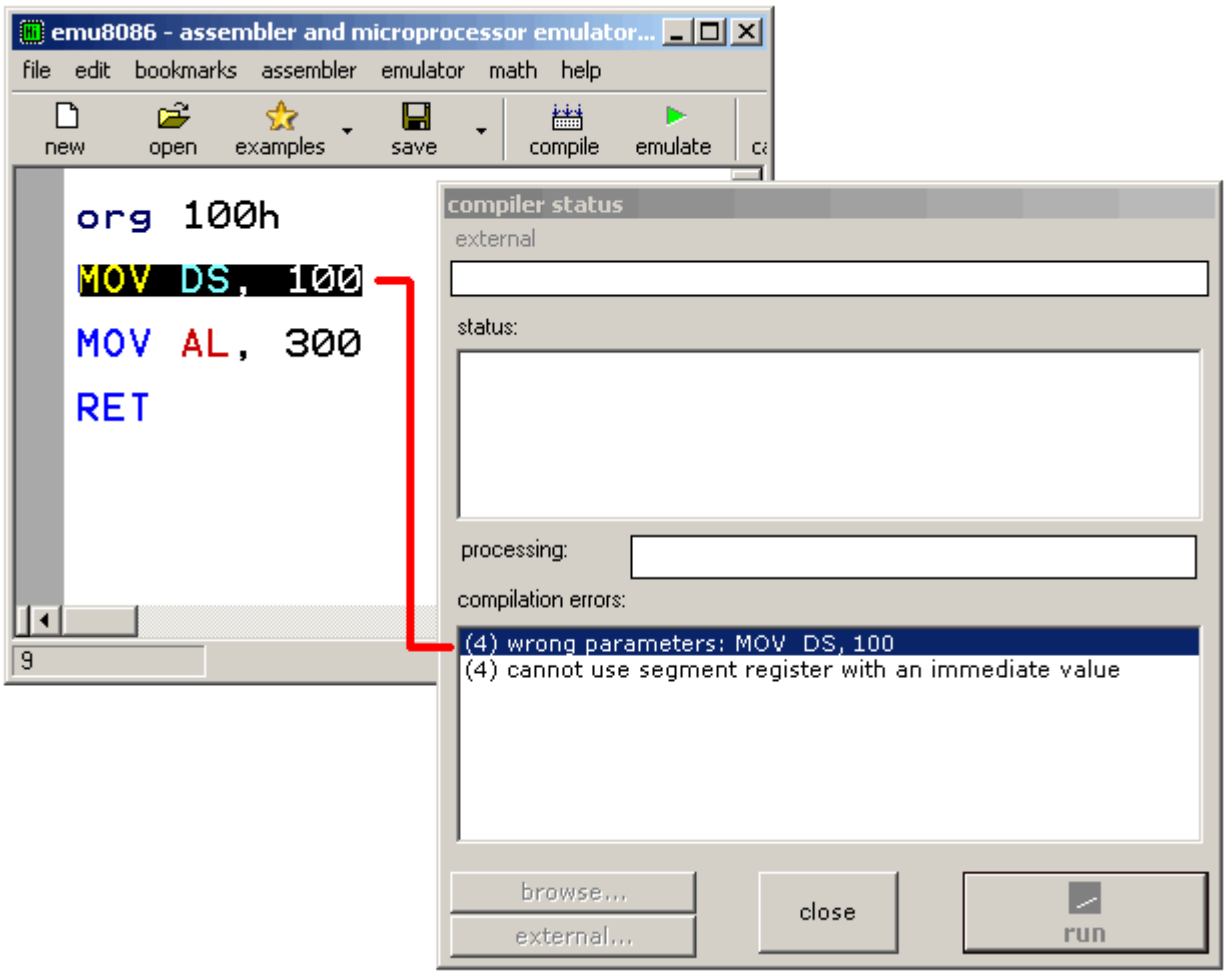
then select **[virtual drive] -> [boot from floppy]** menu to boot emulator from a virtual floppy.

then, if you are curious, you may write the same files to real floppy and boot your computer from it. you can use "writebin.asm" from c:\emu8086\examples\
micro-operating system does not have ms-dos/windows compatible boot sector, so it's better to use an empty floppy disk. refer to **tutorial 11** for more information.
compiler directive **org 7c00h** should be added before the code, when computer starts it loads first track of a floppy disk at the address 0000:7c00.
the size of a **boot record** file should be less then 512 bytes (limited by the size of a disk sector).
execution always starts from the first byte of the file.
this file type is unique to emu8086 emulator.

---

error processing

assembly language compiler (or assembler) reports about errors in a separate information window:



MOV DS, 100 - is illegal instruction because segment registers cannot be set directly, general purpose register should be used, for example
MOV AX, 100
MOV DS, AX

MOV AL, 300 - is illegal instruction because **AL** register has only 8 bits, and thus maximum value for it is 255 (or 11111111b), and the minimum is -128.

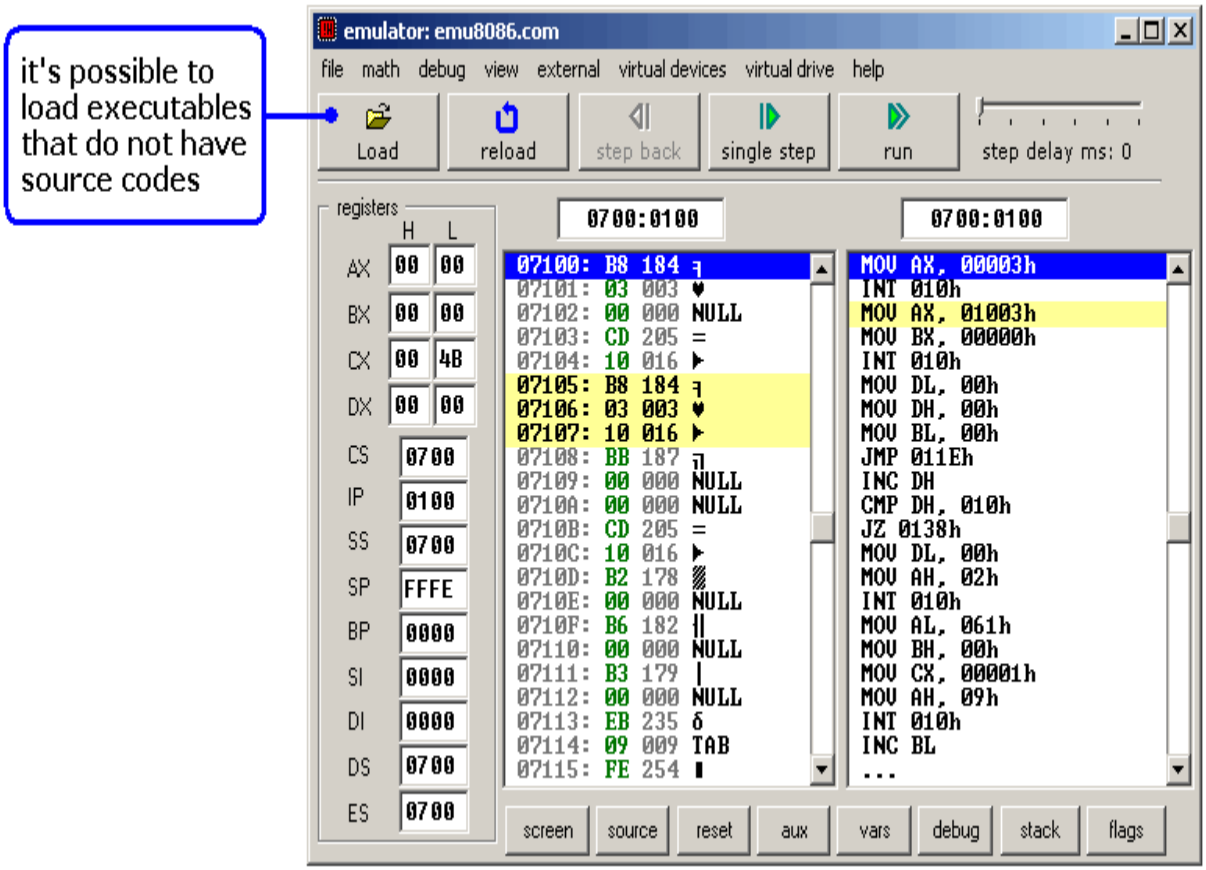
When saving an assembled file, compiler also saves 2 other files that are later used by the emulator to show original source code when you run the binary executable, and select corresponding lines. Very often the original code differs from the disabled code because there are no comments, no segment and no variable declarations. Compiler directives produce no binary code, but everything is converted to pure machine code. Sometimes a single original instruction is assembled into several machine code instructions, this is done mainly

for the compatibility with original 8086 microprocessor (for example **ROL AL, 5** is assembled into five sequential **ROL AL, 1** instructions).

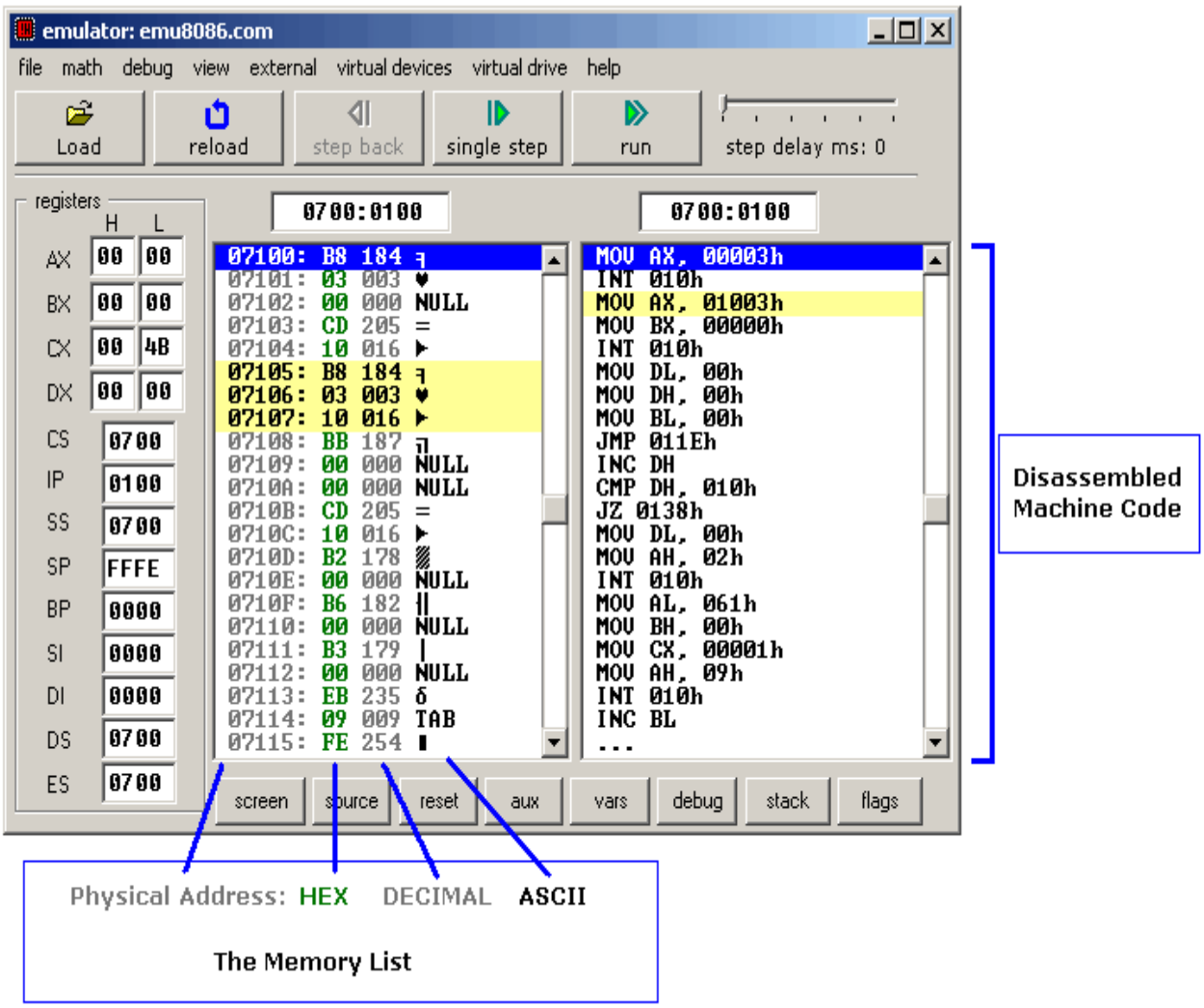
- ***.~asm** - this file contains the original source code that was used to make an executable file.
- ***.debug** - this file has information that enables the emulator select lines of original source code while running the machine code.
- ***.symbol** - symbol table, it contains information that enables to show the "variables" window. It is a plain text file, so you may view it in any text editor (including emu8086 source editor).
- ***.binf** - this ASCII file contains information that is used by emulator to load BIN file at specified location, and set register values prior execution; (created only if an executable is a BIN file).

16) Using the microprocessor emulator

If you want to load your code into the emulator, just click **Emulate** .
But you can also use emulator to load executables even if you don't have the original source code. Select **Show emulator** from the **Emulator** menu.



Try loading files from "**MyBuild**" folder. If there are no files in "**MyBuild**" folder return to source editor, select *Examples* from the *File* menu, load any example, compile it and then load into the emulator:



[**Single Step**] button executes instructions one by one stopping after each instruction.

[**Run**] button executes instructions one by one with delay set by **step delay** between instructions.

Double click on register text-boxes opens **Extended viewer** window with value of that register converted to all possible forms. You can modify the value of the register directly in this window. Double click on memory list item opens **Extended viewer** with WORD value loaded from memory list at selected location. Less significant byte is at lower address: LOW BYTE is loaded from selected position and HIGH BYTE from next memory address. You can modify the value of the memory word directly in the **Extended Viewer** window,

You can modify the values of registers on runtime by typing over the existing values.

[**Flags**] button allows you to view and modify flags on runtime.

17) Virtual drives

Emulator supports up to 4 virtual floppy drives. By default there is a **FLOPPY_0** file that is an image of a real floppy disk (the size of that file is exactly 1,474,560 bytes).

To add more floppy drives select **[Create new floppy drive]** from **[Virtual drive]** menu. Each time you add a floppy drive emulator creates a **FLOPPY_1**, **FLOPPY_2**, and **FLOPPY_3** files. Created floppy disks are images of empty IBM/MS-DOS formatted disk images. Only **4** floppy drives are supported (0..3)!
To **delete** a floppy drive you should close the emulator, delete the required file manually and restart the emulator.

You can determine the number of attached floppy drives using **INT 11h** this function returns **AX** register with BIOS equipment list. Bits 7 and 6 define the number of floppy disk drives (minus 1):

- `Bits 7-6 of AX:`
- `00 single floppy disk.`
- `01 two floppy disks.`
- `10 three floppy disks.`
- `11 four floppy disks.`
- Emulator starts counting attached floppy drives from starting from the first, in case file **FLOPPY_1** does not exist it stops the check and ignores **FLOPPY_2** and **FLOPPY_3** files.

  To write and read from floppy drive you can use **INT 13h** function, see **list of supported interrupts** for more information.

  emulator can emulate tiny operating system, check out **operating system tutorial**.

18) Global Memory Table

8086 CPU can access up to **1 MB** of random access memory (RAM).
This is more than enough for any kind of computations (if used wisely).
memory table of the emulator (and typical ibm pc memory table):

physical address of memory area in HEX	**short description**
00000 - 00400	Interrupt vectors. The emulator loads this file: **c:\emu8086\INT_VECT** at the physical address 000000.
00400 - 00500	System information area. We use a trick to set some parameters by loading a tiny last part (21 bytes) of **INT_VECT** in that area (the size of that file is 1,045 or 415h bytes, so when loaded it takes memory from 00000 to 00415h). this memory block is updated by the emulator when configuration changes, see **system information area** table.
00500 - A0000	A free memory area. A block of **654,080 bytes**. Here you can load your programs.
A0000 - B1000	Video memory for vga, monochrome, and other adapters. It is used by video mode 13h of INT 10h.
B1000 - B8000	Reserved. Not used by the emulator.
B8000 - C0000	32 kb video memory for color graphics adapter (cga). The emulator uses this memory area to keep 8 pages of video memory. The emulator screen can be resized, so less memory is required for each page, although the emulator always uses 1000h (4096 bytes) for each page (see INT 10h / AH=05h in **the list of supported interrupts**).
C0000 - F4000	Reserved.
F4000 - 10FFEF	ROM BIOS and extensions. the emulator loads **BIOS_ROM** file at the physical address 0F4000h. addresses of interrupt table points to this memory area to make emulation of the interrupt functions.

interrupt vector table (memory from 00000h to 00400h)

INT number in hex	address in interrupt vector	address of BIOS sub-program
00	00x4 = 00	F400:0170 - CPU-generated, divide error.
04	04x4 = 10	F400:0180 - CPU-generated, INTO detected overflow.
10	10x4 = 40	F400:0190 - video functions.
11	11x4 = 44	F400:01D0 - get BIOS equipment list.
12	12x4 = 48	F400:01A0 - get memory size.
13	13x4 = 4C	F400:01B0 - disk functions.
15	15x4 = 54	F400:01E0 - BIOS functions.
16	16x4 = 58	F400:01C0 - keyboard functions.
17	17x4 = 5C	F400:0400 - printer.
19	19x4 = 64	FFFF:0000 - reboot.
1A	1Ax4 = 68	F400:0160 - time functions.
1E	1Ex4 = 78	F400:AFC7 - vector of diskette controller parameters.
20	20x4 = 80	F400:0150 - DOS function: terminate program.
21	21x4 = 84	F400:0200 - DOS functions.
33	33x4 = CC	F400:0300 - mouse functions.
all the others	??x4 = ??	F400:0100 - default interrupt stub.

A call to BIOS sub-system is disassembled as **BIOS DI** (Basic Input/Output System - Do Interrupt). To encode this 4 byte instruction, **FFFF** opcode prefix is used. for example: **FFFFCD10** is used to make the emulator to execute interrupt number 10h.

At address **F400:0100** there is this machine code **FFFFCDFF** (it is decoded as INT 0FFh, it is used to generate a default error message, unless you make your own interrupt replacement for missing functions).

System information area (memory from 00400h to 00500h)

address (hex)	size	description
0040h:0010	WORD	BIOS equipment list. bit fields for BIOS-detected installed hardware: bit(s) Description 15-14 number of parallel devices. 13 reserved. 12 game port installed. 11-9 number of serial devices. 8 reserved. 7-6 number of floppy disk drives (minus 1): 00 single floppy disk; 01 two floppy disks; 10 three floppy disks; 11 four floppy disks. 5-4 initial video mode: 00 EGA,VGA,PGA, or other with on-board video BIOS; 01 40x25 CGA color. 10 80x25 CGA color (emulator default). 11 80x25 mono text. 3 reserved. 2 PS/2 mouse is installed. 1 math coprocessor installed. 0 set when booted from floppy.
0040h:0013	WORD	kilobytes of contiguous memory starting at absolute address 00000h. this word is also returned in **AX** by **INT 12h**. this value is set to: **0280h** (640KB).
0040h:004A	WORD	number of columns on screen. default value: **0032h** (50 columns).
0040h:004E	WORD	current video page start address in video memory (after 0B800:0000). default value: **0000h**.
0040h:0050	8 WORDs	contains row and column position for the cursors on each of eight video pages. default value: **0000h** (for all 8 WORDs).
0040h:0062	BYTE	current video page number. default value: **00h** (first page).
0040h:0084	BYTE	rows on screen minus one. default value: **13h** (19+1=20 columns).

19) I/O ports and Hardware Interrupts

The emulator does not reproduce any input/output devices of the original IBM PC ®, however theoretically it may be possible to create emulation of the original ibm pc devices. emu8086 supports user-created virtual devices that can be accessed from assembly language program using **in** and **out** instructions. devices that can be created by anyone with basic programming experience in any high or low level programming language. the simplest virtual device in assembly language can be found in examples: **simplest.asm**

Input / Output ports

emu8086 supports additional devices that can be created by anyone with basic programming experience in any language device can be written in any language, such as: java, visual basic, vc++, delphi, c#, .net or in any other programming language that allow to directly read and write files. for more information and sample source code look inside this folder: **c:\emu8086\DEVICES\DEVELOPER**

The latest version of the emulator has no reserved or fixed I/O ports, input / output addresses for custom devices are from **0000 to 0FFFFh** (0 to 65535), but it is important that two devices that use the same ports do not run simultaneously to avoid hardware conflict.

Port 100 corresponds to byte 100 in this file: **c:\emu8086.io** , port 0 to byte 0, port 101 to byte 101, etc...

Emulation of Hardware Interrupts

External hardware interrupts can be triggered by external peripheral devices and microcontrollers or by the 8087 mathematical coprocessor.

Hardware interrupts are disabled when interrupt flag (IF) is set to 0. when interrupt flag is set to 1, the emulator continually checks first 256 bytes of this file **c:\emu8086.hw** if any of the bytes is none-zero the microprocessor transfers control to an interrupt handler that matches the triggering byte offset in **emu8086.hw** file (0 to 255) according to the interrupt vector table (memory 0000-0400h) and resets the byte in **emu8086.hw** to 00.

These instructions can be used to disable and enable hardware interrupts:

cli - clear interrupt flag (disable hardware interrupts).
sti - set interrupt flag (enable hardware interrupts).

by default hardware interrupts are enabled and are disabled automatically when software or hardware interrupt is in the middle of the execution.

Examples of Custom I/O Devices

Ready devices are available from **virtual devices** menu of the emulator.

- **Traffic Lights** - port 4 (word)

  the traffic lights are controlled by sending data to i/o port 4. there are 12 lamps: 4 green, 4 yellow, and 4 red.

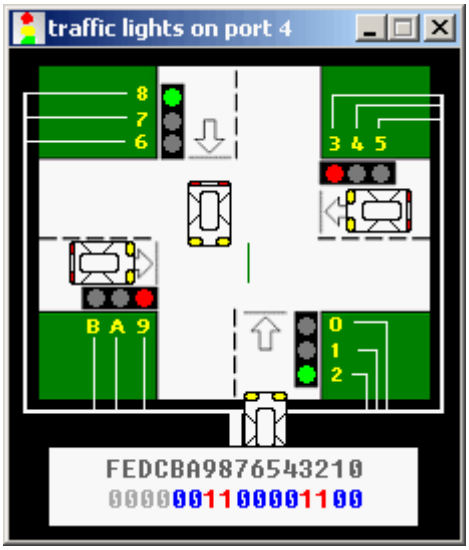
  you can set the state of each lamp by setting its bit:

  1 - the lamp is turned on.
  0 - the lamp is turned off.

  only 12 low bits of a word are used (0 to 11), last bits (12 to 15) are unused.

  for example:

```
MOV AX, 0000001011110100b
OUT 4, AX
```



we use yellow hexadecimal digits in caption (to achieve compact view), here's a conversion:

- Hex - Decimal
-
- A - 10
- B - 11
- C - 12 (unused)
- D - 13 (unused)
- E - 14 (unused)

F - 15 (unused)

first operand for **OUT** instruction is a port number (**4**), second operand is a word (**AX**) that is written to the port. first operand must be an immediate byte value (0..255) or **DX** register. second operand must be **AX** or **AL** only.

see also **traffic_lights.asm** in c:\emu8086\examples.

---

if required you can read the data from port using **IN** instruction, for example:

```
IN AX, 4
```

first operand of **IN** instruction (**AX**) receives the value from port, second operand (**4**) is a port number. first operand must be **AX** or **AL** only. second operand must be an immediate byte value (0..255) or **DX** register.

- **Stepper Motor** - port 7 (byte)

  the stepper motor is controlled by sending data to i/o port 7.

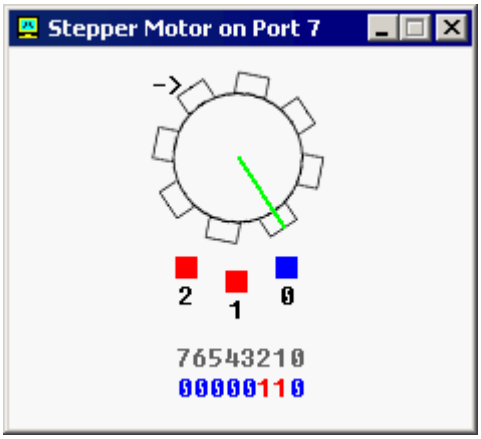
  stepper motor is electric motor that can be precisely controlled by signals from a computer.

  the motor turns through a precise angle each time it receives a signal.

  by varying the rate at which signal pulses are produced, the motor can be run at different speeds or turned through an exact angle and then stopped.

  This is a basic 3-phase stepper motor, it has 3 magnets controlled by bits **0, 1 and 2**. other bits (3..7) are unused.

  When magnet is working it becomes red. The arrow in the left upper corner shows the direction of the last motor move. Green line is here just to see that it is really rotating.



  For example, the code below will do three clock-wise half-steps:

```
MOV AL, 001b ; initialize.
OUT 7, AL

MOV AL, 011b ; half step 1.
OUT 7, AL

MOV AL, 010b ; half step 2.
OUT 7, AL

MOV AL, 110b ; half step 3.
OUT 7, AL
```

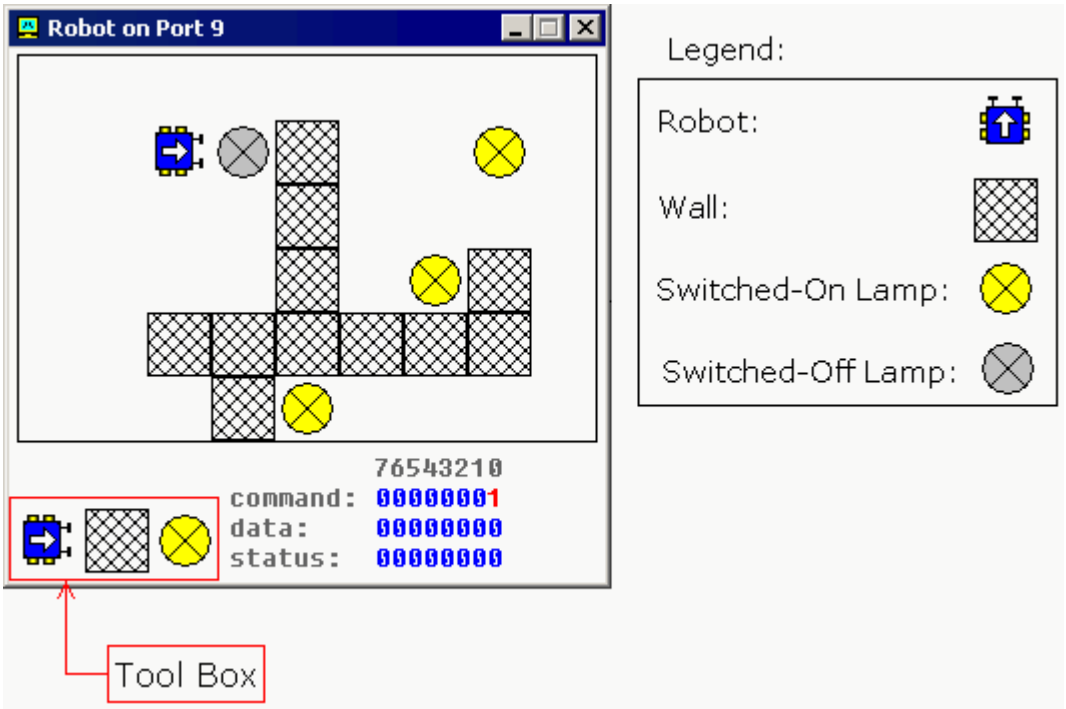
If you ever played with magnets you will understand how it works. try experimenting, or see **stepper_motor.asm** in c:\emu8086\examples.

If required you can read the data from port using **IN** instruction, for example:

```
IN AL, 7
```

Stepper motor sets topmost bit of byte value in port 7 when it's ready.

- **Robot** - port 9 (3 bytes)



The robot is controlled by sending data to i/o port 9.

-
-
-

- The first byte (port 9) is a **command register**. set values to this port to make robot do something. supported values:

decimal value	binary value	action
0	00000000	do nothing.
1	00000001	move forward.
2	00000010	turn left.
3	00000011	turn right.
4	00000100	examine. examines an object in front using sensor. when robot completes the task, result is set to **data register** and **bit #0** of **status register** is set to **1**.
5	00000101	switch on a lamp.
6	00000110	switch off a lamp.

- The second byte (port 10) is a **data register**. this register is set after robot completes the **examine** command:

decimal value	binary value	meaning
255	11111111	wall
0	00000000	nothing
7	00000111	switched-on lamp
8	00001000	switched-off lamp

- The third byte (port 11) is a **status register**. read values from this port to determine the state of the robot. each bit has a specific property:

bit number	description
bit #0	**zero** when there is no new data in **data register**, **one** when

	there is new data in **data register**.
bit #1	**zero** when robot is ready for next command, **one** when robot is busy doing some task.
bit #2	**zero** when there is no error on last command execution, **one** when there is an error on command execution (when robot cannot complete the task: move, turn, examine, switch on/off lamp).

- example:

```
MOV AL, 1  ; move forward.
OUT 9, AL  ;

MOV AL, 3  ; turn right.
OUT 9, AL  ;

MOV AL, 1  ; move forward.
OUT 9, AL  ;

MOV AL, 2  ; turn left.
OUT 9, AL  ;

MOV AL, 1  ; move forward.
OUT 9, AL  ;
```

keep in mind that robot is a mechanical creature and it takes some time for it to complete a task. you should always check bit#1 of **status register** before sending data to port 9, otherwise the robot will reject your command and "**busy!**" will be shown. see **robot.asm** in c:\emu8086\examples.

Creating Custom Robo-World Map

It is possible to change the default map for the robot using the tool box.

if you click the robot button and place robot over existing robot it will turn 90 degrees counter-clock-wise. to manually move the robot just place it anywhere else on the map.

If you click lamp button and click switched-on lamp the lamp will be switched-off, if lamp is already switched-off it will be deleted. click over empty space will create a new switched-on lamp.

Placing wall over existing wall deletes the wall.

Current version is limited to a single robot only. if you forget to place a robot on the map it will be placed in some random coordinates.

When robot device is closed the map is automatically saved inside this file: **c:\emu8086\devices\robot_map.dat**
It is possible to have several maps by renaming and coping this file before starting the robot device.

The right-click over the map brings up a popup menu that allows to switch-on or switch-off all the lamps at once.