Documentation for Emu8086

- Where to start?
- Tutorials
- *Emu8086* reference
- Complete 8086 instruction set

Emu8086 Overview

Everything for learning assembly language in one pack! Emu8086 combines an advanced source editor, assembler, disassembler, software emulator (Virtual PC) with debugger, and step by step tutorials.

This program is extremely helpful for those who just begin to study assembly language. It compiles the source code and executes it on emulator step by step.

Visual interface is very easy to work with. You can watch registers, flags and memory while your program executes.

Arithmetic & Logical Unit (ALU) shows the internal work of the central processor unit (CPU).

Emulator runs programs on a Virtual PC, this completely blocks your program from accessing real hardware, such as hard-drives and memory, since your assembly code runs on a virtual machine, this makes debugging much easier.

8086 machine code is fully compatible with all next generations of Intel's micro-processors, including Pentium II and Pentium 4, I'm sure Pentium 5 will support 8086 as well. This makes 8086 code very portable, since it runs both on ancient and on the modern computer systems. Another advantage of 8086 instruction set is that it is much smaller, and thus easier to learn.

Emu8086 has a much easier syntax than any of the major assemblers, but will still generate a program that can be executed on any computer that runs 8086 machine code; a great combination for beginners!

Note: If you don't use *Emu8086* to compile the code, you won't be able to step through your actual source code while running it.

Where to start?

1. Start *Emu8086* by selecting its icon from the start menu, or by running **Emu8086.exe**.
2. Select "**Samples**" from "**File**" menu.
3. Click **[Compile and Emulate]** button (or press **F5** hot key).
4. Click [**Single Step**] button (or press **F8** hot key), and watch how the code

is being executed.

5. Try opening other samples, all samples are heavily commented, so it's a great learning tool.

6. This is the right time to **see the tutorials**.

Tutorials

8086 Assembler Tutorials

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!

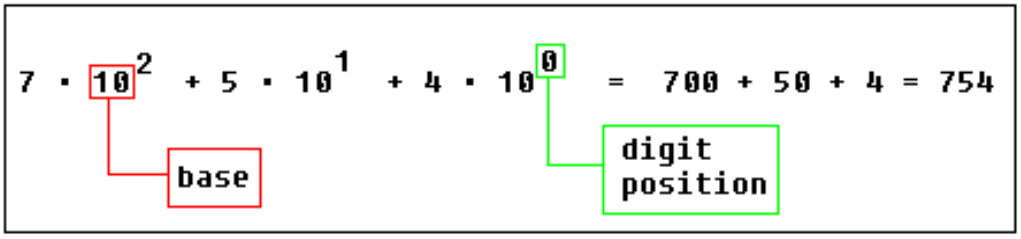
Using sticks to count was a great idea for its time. And using symbols instead of real sticks was much better. One of the best ways to represent a number today is by using the modern decimal system. Why? Because it includes the major breakthrough of using a symbol to represent the idea of counting ***nothing***. About 1500 years ago in India, **zero** (**0**) was first used as a number! It was later used in the Middle East as the Arabic, *sifr*. And was finally introduced to the West as the Latin, *zephiro*. Soon you'll see just how valuable an idea this is for all modern number systems.

Decimal System

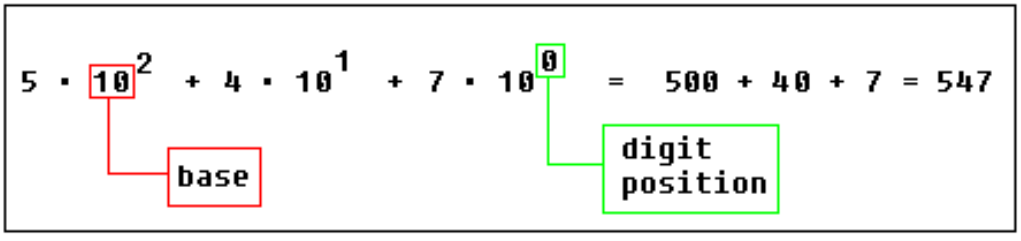
Most people today use decimal representation to count. In the decimal system there are 10 digits:

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:

$10^0 = 1$ $0^0 = 1$ $X^0 = 1$

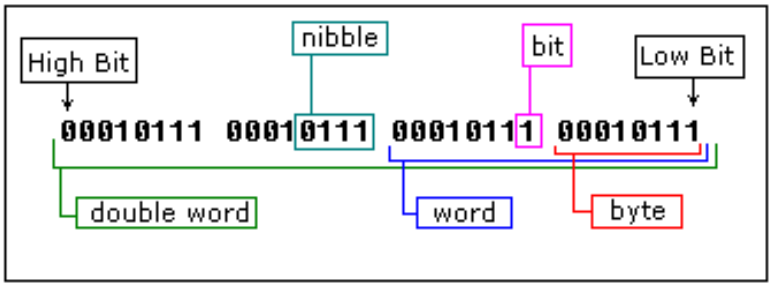
Binary System

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

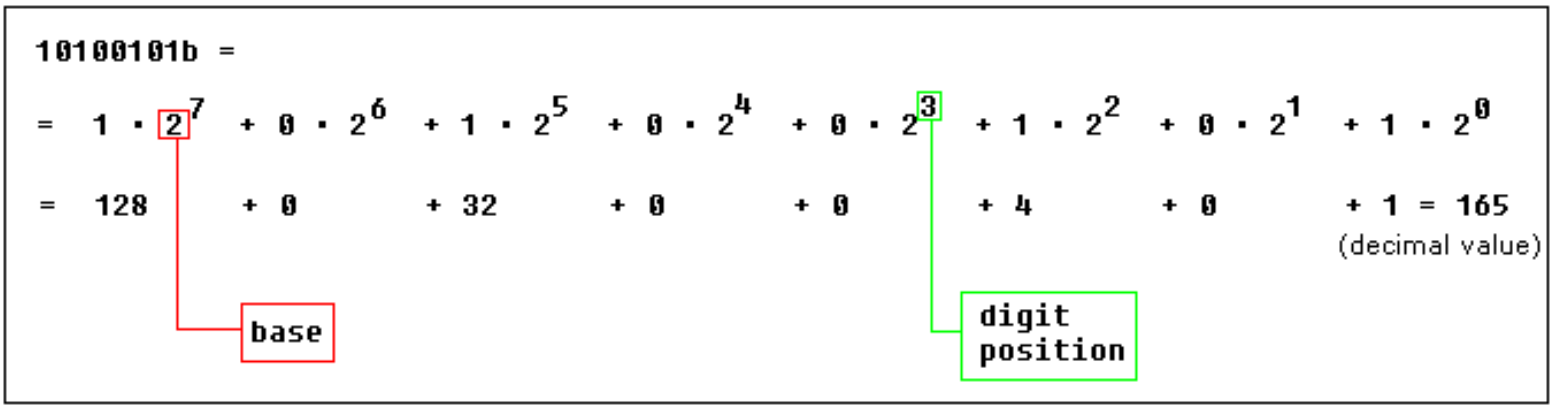
And thus the **base** is **2**.

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

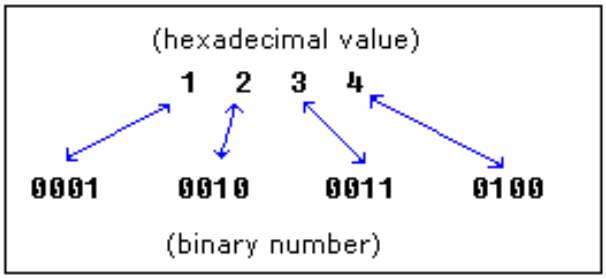
Hexadecimal System uses 16 digits:

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

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

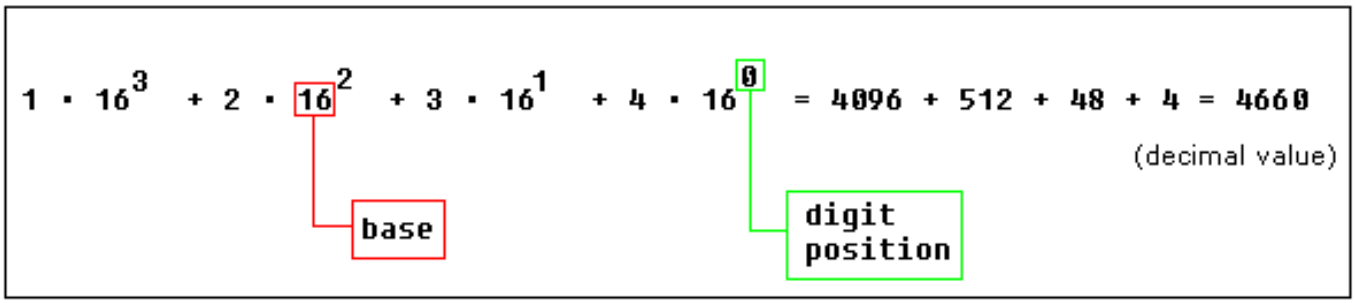
Hexadecimal numbers are compact and easy to read.
It is very easy to convert numbers from binary system to hexadecimal system and vice-versa, every nibble (4 bits) can be converted to a hexadecimal digit using this table:

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:

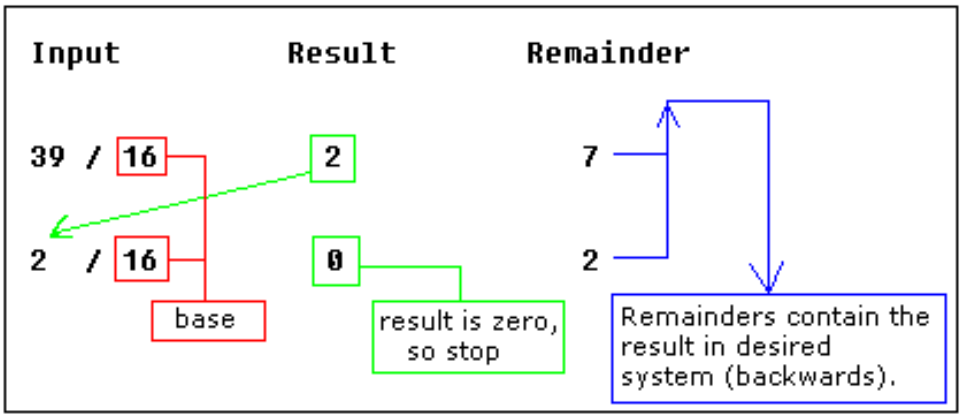


Converting from Decimal System to Any Other

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.

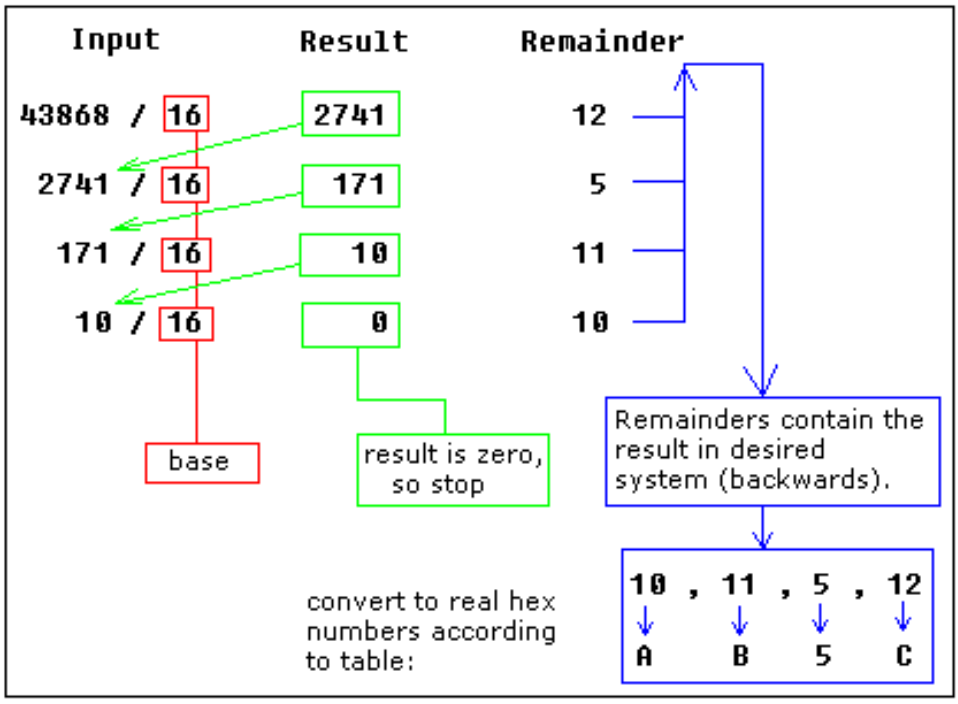
The **remainders** are then used to represent a value in that system.

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.

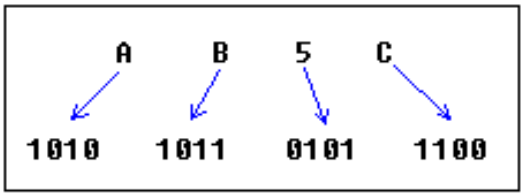
Here is another more complex example:
let's convert decimal number **43868** to hexadecimal form:



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**

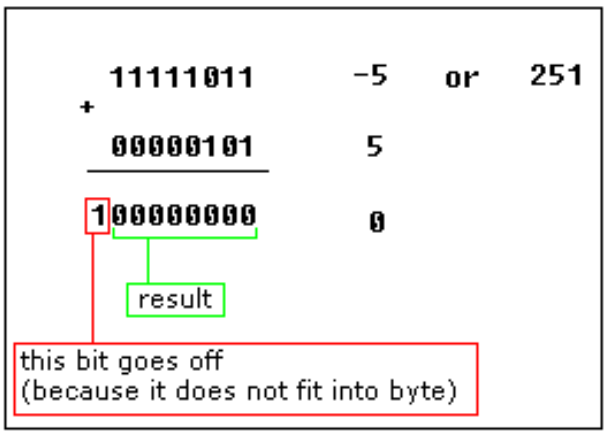
Signed Numbers

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**.

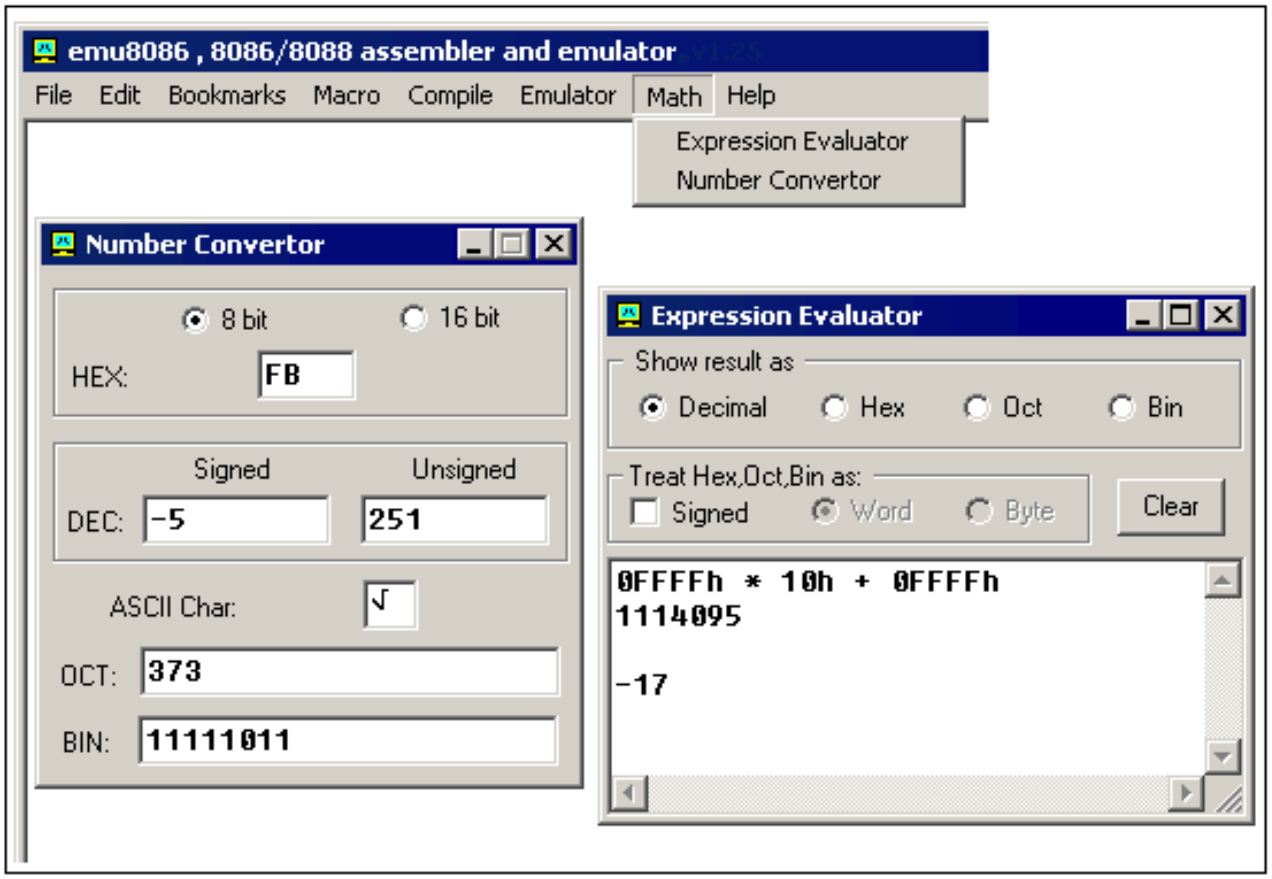
Using this complex way to represent negative numbers has some meaning, in math when you add "**- 5**" to "**5**" you should get zero.
This is what happens when processor adds two bytes **5** and **251**, the result gets over **255**, because of the overflow processor gets zero!



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:



Number Convertor 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.

Expression Evaluator 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 **Number Convertor** 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

>>> Next Tutorial >>>

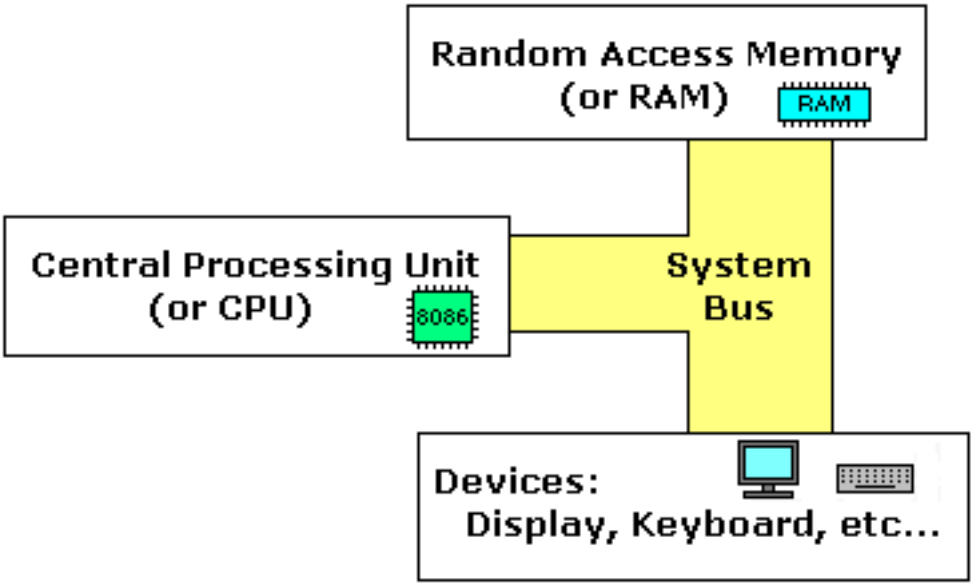
8086 Assembler Tutorial for Beginners (Part 1)

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.

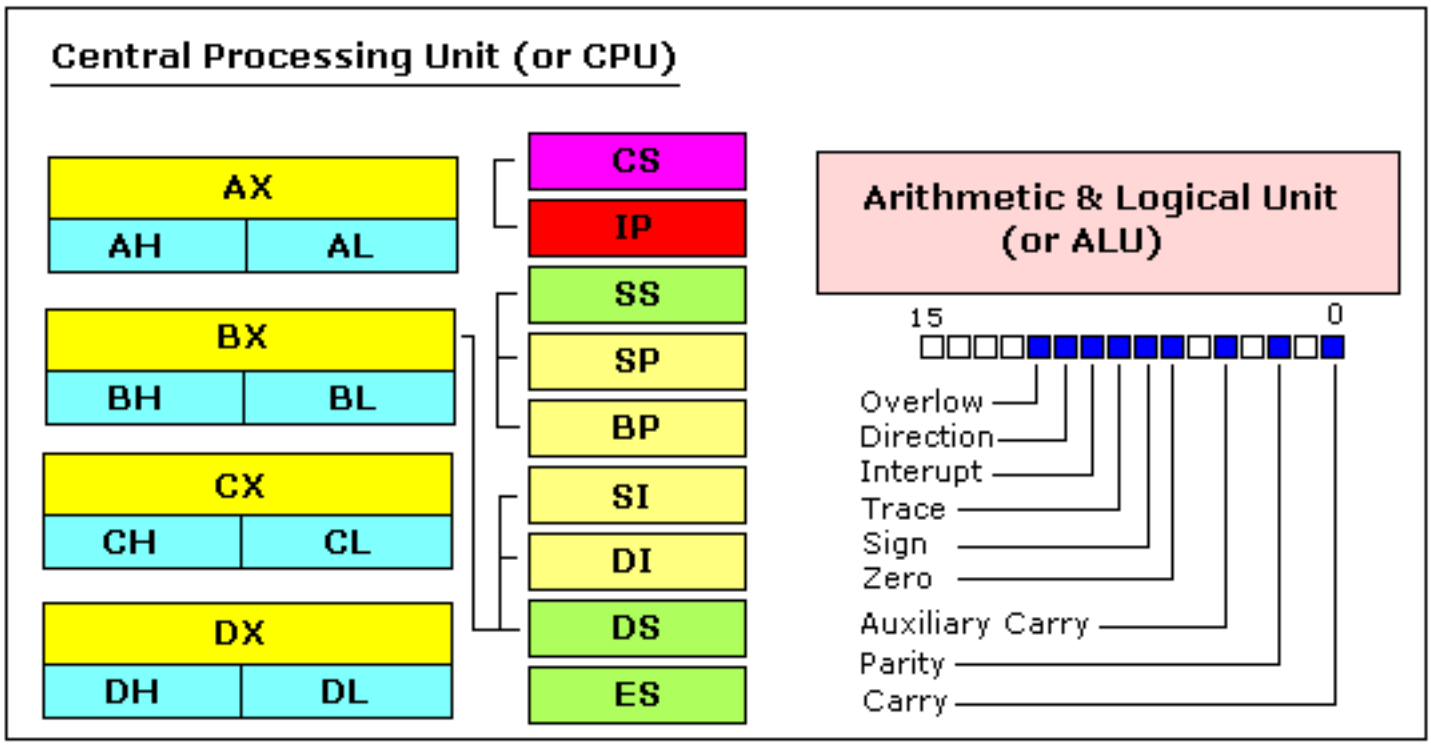
What is an assembly language?

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 processor.

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.

>>> Next Part >>>

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 displacement.

d16 - stays for 16 bit displacement.

Displacement can be a immediate value or offset of a variable, or even both. It's up to compiler to calculate a single immediate value.

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

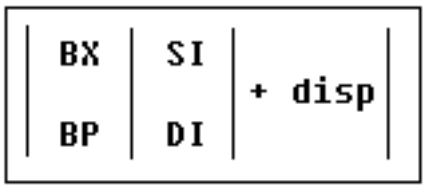
Displacement is a **signed** value, so it can be both positive or negative.

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 is an example of a valid addressing mode: **[BX+5]**.

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.

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.
```

Emu8086 supports shorter prefixes as well:

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

sometimes compiler can calculate the data type automatically, but you may not and should not rely on that when one of the operands is an immediate value.

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:

```
#MAKE_COM#       ; instruct compiler to make COM file.
ORG 100h         ; directive required for a 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, 01011111b ; 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).

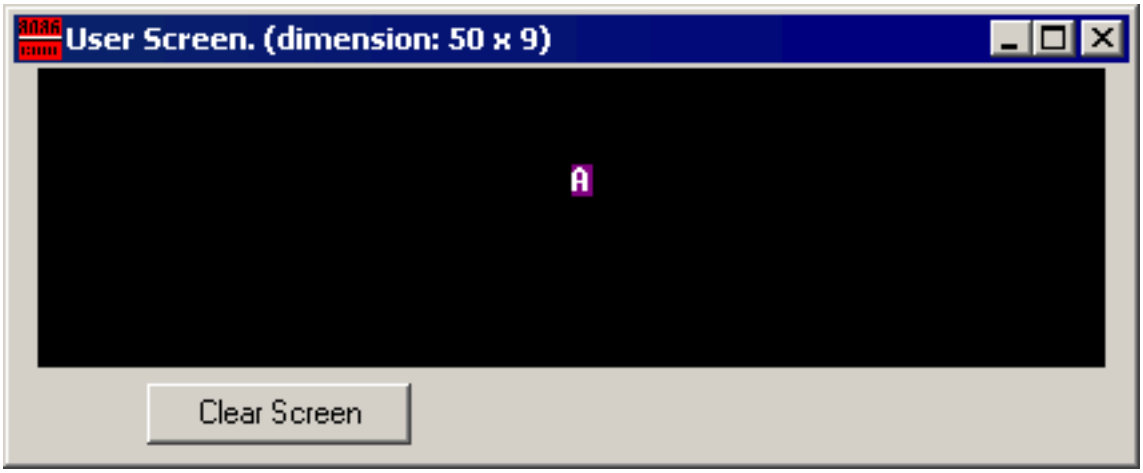
The Emulator window should open with this program loaded, click [**Single Step**] button and watch the register values.

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.

<<< Previous Part <<< **>>> Next Part >>>**

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:

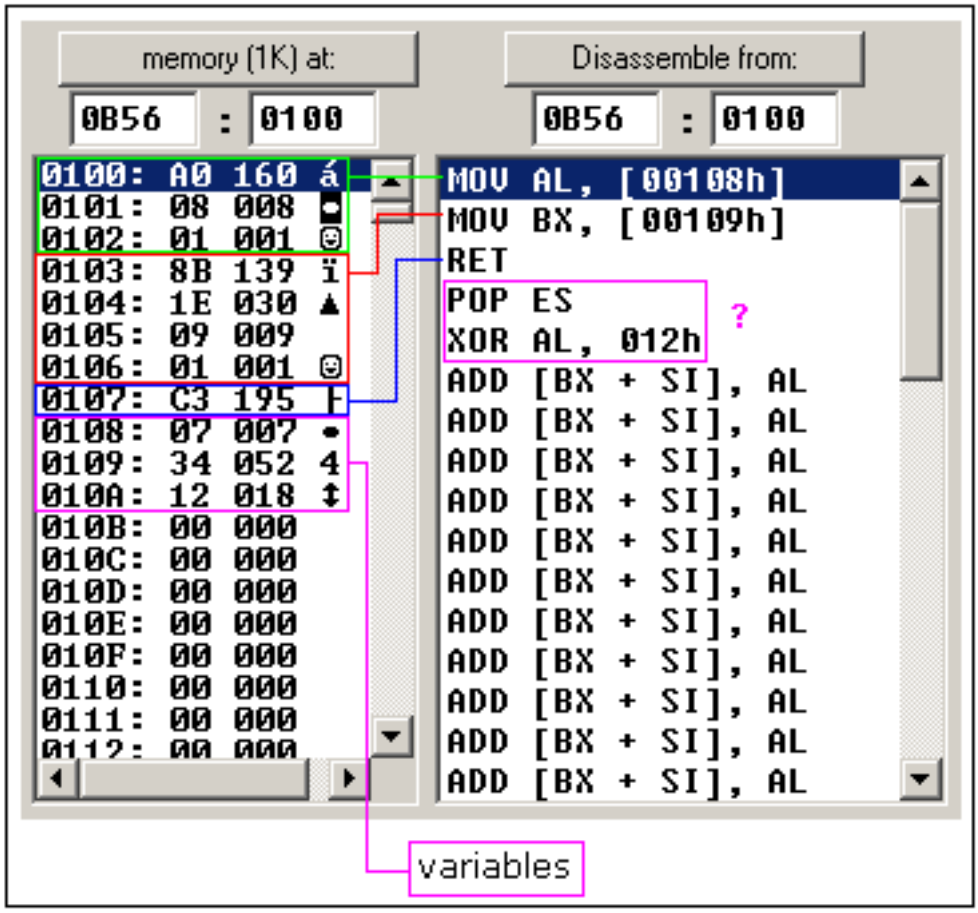
```
#MAKE_COM#
ORG 100h

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:

```
#MAKE_COM#
ORG 100h

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.

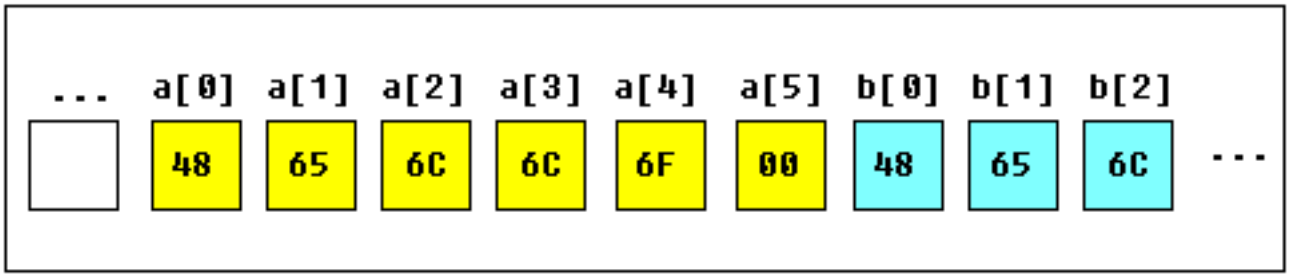
Arrays

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!

The expansion of **DUP** operand should not be over 1020 characters! (the expansion of last example is 13 chars), if you need to declare huge array divide declaration it in two lines (you will get a single huge array in the memory).

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**

sometimes compiler can calculate the data type automatically, but you may not and should not rely on that when one of the operands is an immediate value.

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:

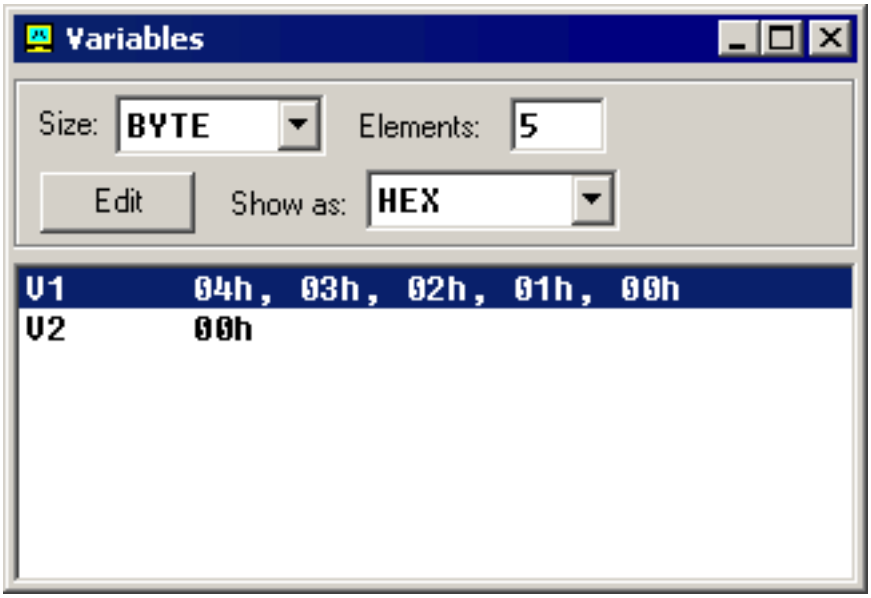
```
k EQU 5

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...

<<< Previous Part <<< **>>> Next Part >>>**

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.

```
#MAKE_COM#      ; instruct compiler to make COM file.
ORG   100h

; 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.

<<< Previous Part <<< **>>> Next Part >>>**

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 **END**!!), 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.
```

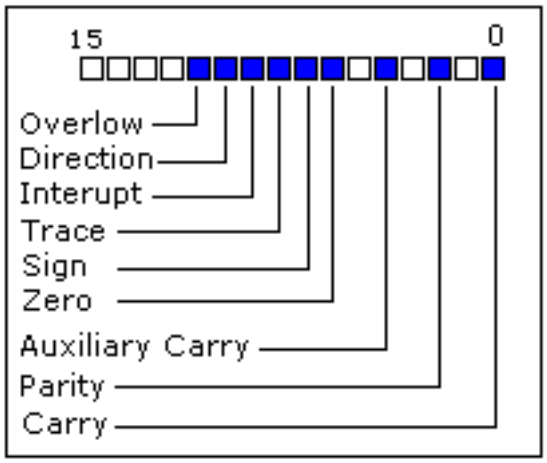
First compiler processes the declarations (these are just regular the macros that are expanded to procedures). When compiler gets to **CALL** instruction it replaces the procedure name with the address of the code where the procedure is declared. When **CALL** instruction is executed control is transferred 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.

<<< Previous Part <<< **>>> Next Part >>>**

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.

<<< Previous Part <<< **>>> Next Part >>>**

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 is 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.

END                   ; directive to stop the compiler.
```

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 can see 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**. Different names are used to make programs easier to understand and code.

Jump instructions for signed numbers

Instruction	Description	Condition	Opposite Instruction
JE , JZ	Jump if Equal (=). Jump if Zero.	ZF = 1	JNE, JNZ
JNE , JNZ	Jump if Not Equal (<>). Jump if Not Zero.	ZF = 0	JE, JZ
JG , JNLE	Jump if Greater (>). Jump if Not Less or Equal (not <=).	ZF = 0 and SF = OF	JNG, JLE
JL , JNGE	Jump if Less (<). Jump if Not Greater or Equal (not >=).	SF <> OF	JNL, JGE
JGE , JNL	Jump if Greater or Equal (>=). Jump if Not Less (not <).	SF = OF	JNGE, JL
JLE , JNG	Jump if Less or Equal (<=). Jump if Not Greater (not >).	ZF = 1 or SF <> OF	JNLE, JG

<> - sign means not equal.

Jump instructions for unsigned numbers

Instruction	Description	Condition	Opposite Instruction
JE , JZ	Jump if Equal (=). Jump if Zero.	ZF = 1	JNE, JNZ
JNE , JNZ	Jump if Not Equal (<>). Jump if Not Zero.	ZF = 0	JE, JZ

JA , JNBE	Jump if Above (>). Jump if Not Below or Equal (not <=).	CF = 0 and ZF = 0	JNA, JBE
JB , JNAE, JC	Jump if Below (<). Jump if Not Above or Equal (not >=). Jump if Carry.	CF = 1	JNB, JAE, JNC
JAE , JNB, JNC	Jump if Above or Equal (>=). Jump if Not Below (not <). Jump if Not Carry.	CF = 0	JNAE, JB
JBE , JNA	Jump if Below or Equal (<=). Jump if Not Above (not >).	CF = 1 or ZF = 1	JNBE, JA

Generally, when it is required to compare numeric values **CMP** instruction is used (it does the same as **SUB** (subtract) instruction, but does not keep the result, just affects the flags).

The logic is very simple, for example:
it's required to compare 5 and 2,
5 - 2 = 3
the result is not zero (Zero Flag is set to 0).

Another example:
it's required to compare 7 and 7,
7 - 7 = 0
the result is zero! (Zero Flag is set to 1 and **JZ** or **JE** will do the jump).

Here is an example of **CMP** instruction and conditional jump:

```
include emu8086.inc

ORG    100h

MOV    AL, 25     ; set AL to 25.
MOV    BL, 10     ; set BL to 10.

CMP    AL, BL     ; compare AL - BL.

JE     equal      ; jump if AL = BL (ZF = 1).

PUTC   'N'        ; if it gets here, then AL <> BL,
JMP    stop       ; so print 'N', and jump to stop.

equal:            ; if gets here,
PUTC   'Y'        ; then AL = BL, so print 'Y'.

stop:

RET               ; gets here no matter what.

END
```

Try the above example with different numbers for **AL** and **BL**, open flags by clicking on [**FLAGS**] button, use [**Single Step**] and see what happens, don't forget to recompile and reload after every change (use **F5** shortcut).

All conditional jumps have one big limitation, unlike **JMP** instruction they can only jump **127** bytes forward and **128** bytes backward (note that most instructions are assembled into 3 or more bytes).

We can easily avoid this limitation using a cute trick:

- Get a opposite conditional jump instruction from the table above, make it jump to *label_x*.
- Use **JMP** instruction to jump to desired location.

- Define *label_x:* just after the **JMP** instruction.

label_x: - can be any valid label name.

Here is an example:

```
include emu8086.inc

ORG    100h

MOV    AL, 25     ; set AL to 25.
MOV    BL, 10     ; set BL to 10.

CMP    AL, BL     ; compare AL - BL.


JNE    not_equal  ; jump if AL <> BL (ZF = 0).
JMP    equal
not_equal:


; let's assume that here we
; have a code that is assembled
; to more then 127 bytes...


PUTC   'N'        ; if it gets here, then AL <> BL,
JMP    stop       ; so print 'N', and jump to stop.

equal:           ; if gets here,
PUTC   'Y'        ; then AL = BL, so print 'Y'.

stop:

RET              ; gets here no matter what.

END
```

Another, yet rarely used method is providing an immediate value instead of a label. When immediate value starts with a '$' character relative jump is performed, otherwise compiler calculates instruction that jumps directly to given offset. For example:

```
ORG    100h

; unconditional jump forward:
; skip over next 2 bytes,
JMP $2
a DB 3    ; 1 byte.
b DB 4    ; 1 byte.

; JCC jump back 7 bytes:
; (JMP takes 2 bytes itself)
MOV BL,9
DEC BL      ; 2 bytes.
CMP BL, 0   ; 3 bytes.
JNE $-7

RET

END
```

<<< Previous Part <<< **>>> Next Part >>>**

Emu8086 reference

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.

<<< Previous Part <<< **>>> Next Part >>>**

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.

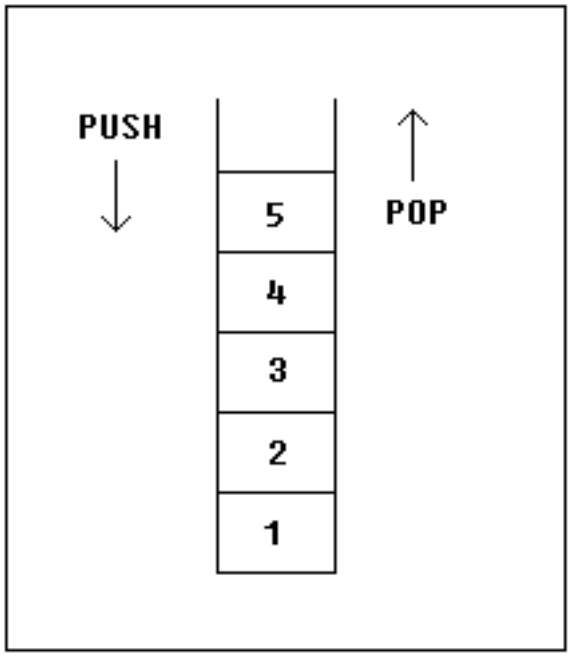
SREG: DS, ES, SS, (except CS).

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.

<<< Previous Part <<< **>>> Next Part >>>**

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.

<<< Previous Part <<< **>>> Next Part >>>**

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...).
- It is easy 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

; 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 **Emu8086** source editor and press [**Compile and Emulate**] button. The Emulator automatically loads ".boot" file to 0000h:7C00h.

You can run it just like a regular program, or you can use the **Virtual Drive** menu to **Write 512 bytes at 7C00h to** the **Boot Sector** of a virtual floppy drive (FLOPPY_0 file in Emulator's folder).
After writing your program to the Virtual Floppy Drive, you can select **Boot from Floppy** from **Virtual Drive** menu.

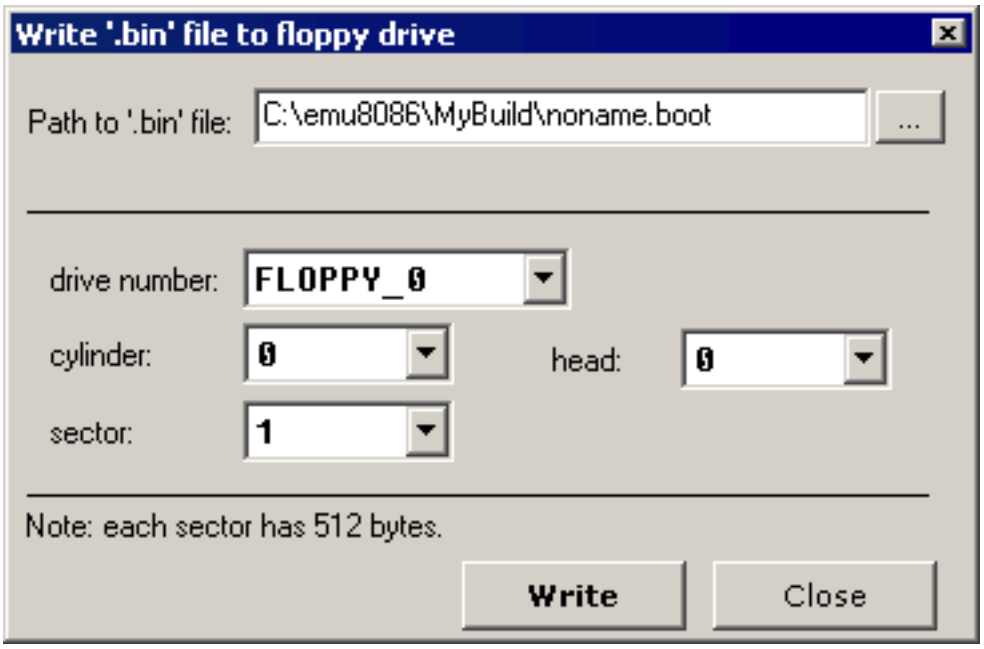
If you are curious, you may write the virtual floppy (**FLOPPY_0**) or "**.boot**" file to a real floppy disk and boot your computer from it, I recommend using "RawWrite for Windows" from: **http://uranus.it.swin.edu.au/~jn/linux/rawwrite.htm**
(recent builds now work under all versions of Windows!)

Note: however, that this **.boot** file is ***not*** an MS-DOS compatible boot sector (it will not allow you to read or write data on this diskette until you format it again), so don't bother writing only this sector to a diskette with data on it. As a matter of fact, if you use any 'raw-write' programs, such at the one listed above, they will erase all of the data anyway. So make sure the diskette you use doesn't contain any important data.

"**.boot**" files 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. A good example of a tiny Operating System can be found in "Samples" folder as:
micro-os_loader.asm
micro-os_kernel.asm

To create extensions for your Operating System (over 512 bytes), you can use "**.bin**" files (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:



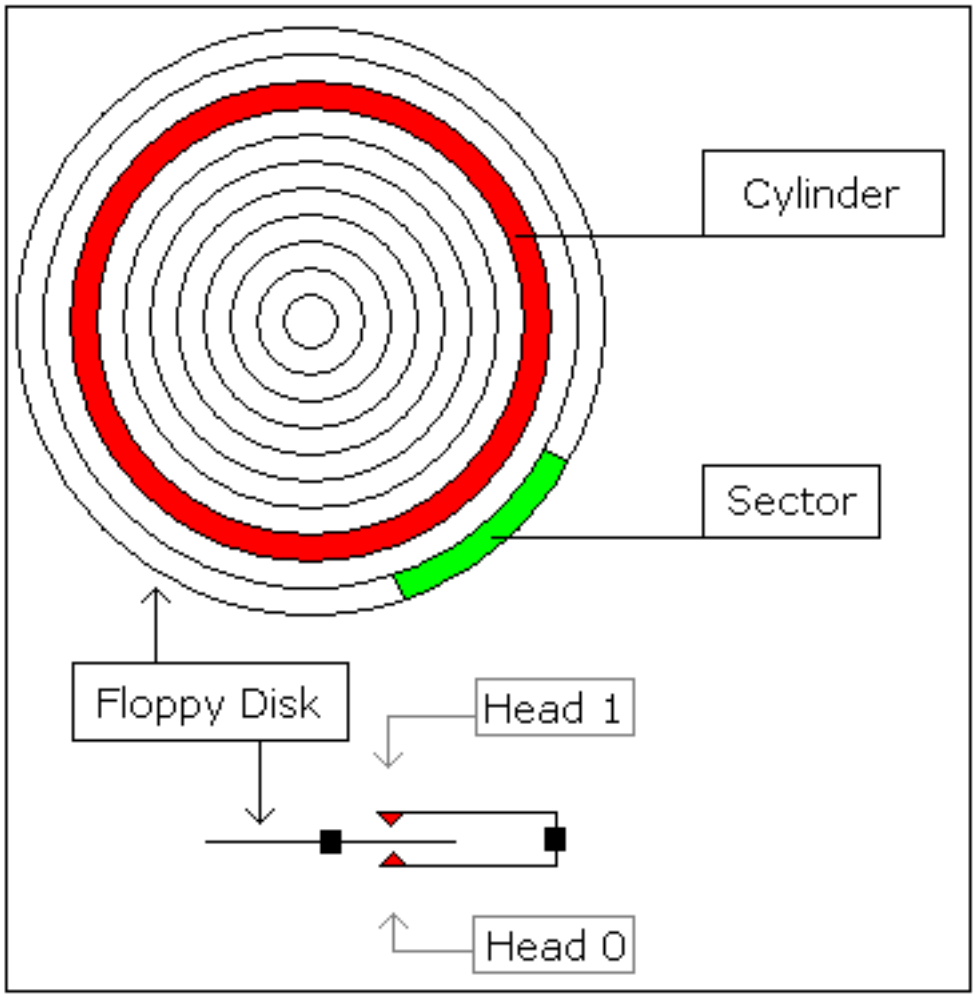
You can also use this to write "**.boot**" files.

Sector at:

Cylinder: 0
Head:0
Sector: 1

is the boot sector!

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.

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

<<< Previous Part <<< **>>> Next Part >>>**

8086 Assembler Tutorial for Beginners (Part 12)

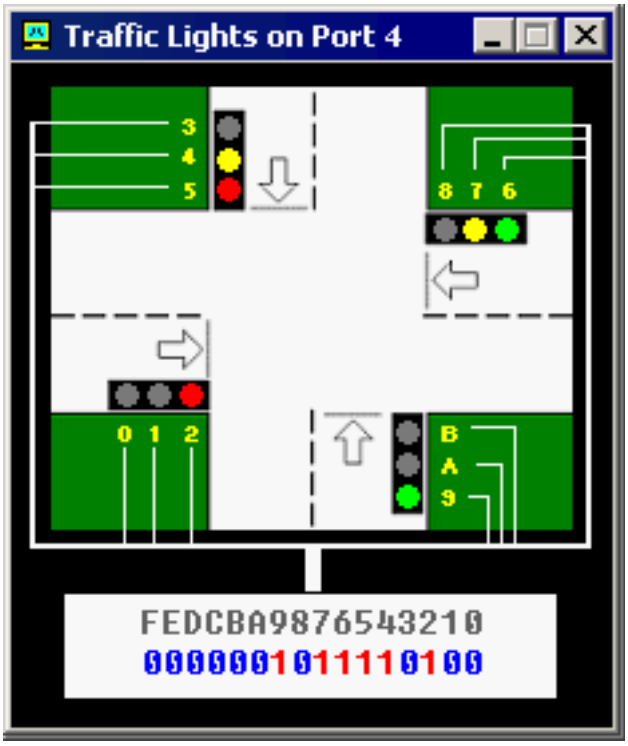
Controlling External Devices

There are 3 devices attached to the emulator: Traffic Lights, Stepper-Motor and Robot. You can view devices using "**Virtual Devices**" menu of the emulator.

For technical information see **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:

```
; directive to create BIN file:
#MAKE_BIN#
#CS=500#
#DS=500#
#SS=500#
#SP=FFFF#
#IP=0#

; skip the data table:
JMP start

table DW 1000011100001b
      DW 1100111110011b
      DW 0011000011100b
      DW 0111100111110b

start:

MOV SI, 0

; set loop counter to number
; of elements in table:
MOV CX, 4

next_value:

; get value from table:
MOV AX, table[SI]

; set value to I/O port
; of traffic lights:
OUT 4, AX

; next word:
ADD SI, 2

CALL PAUSE

LOOP next_value

; start from over from
```

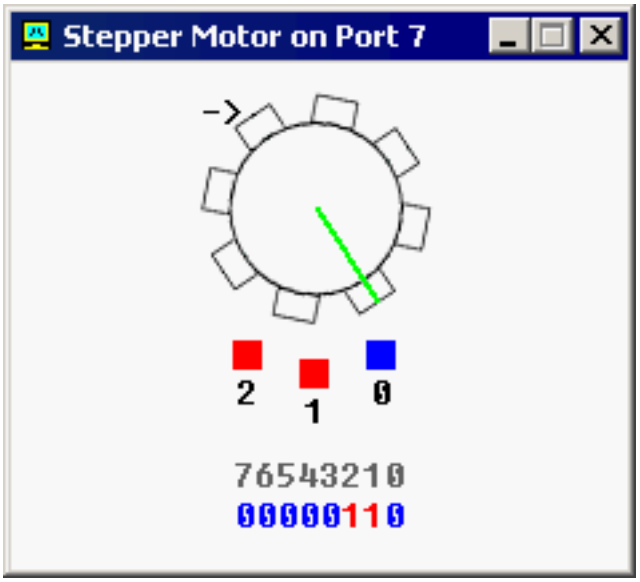
```
; the first value
JMP start

; ============================
PAUSE PROC
; store registers:
PUSH CX
PUSH DX
PUSH AX

; set interval (1 million
; microseconds - 1 second):
MOV     CX, 0Fh
MOV     DX, 4240h
MOV     AH, 86h
INT     15h

; restore registers:
POP AX
POP DX
POP CX
RET
PAUSE ENDP
; ============================
```

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.

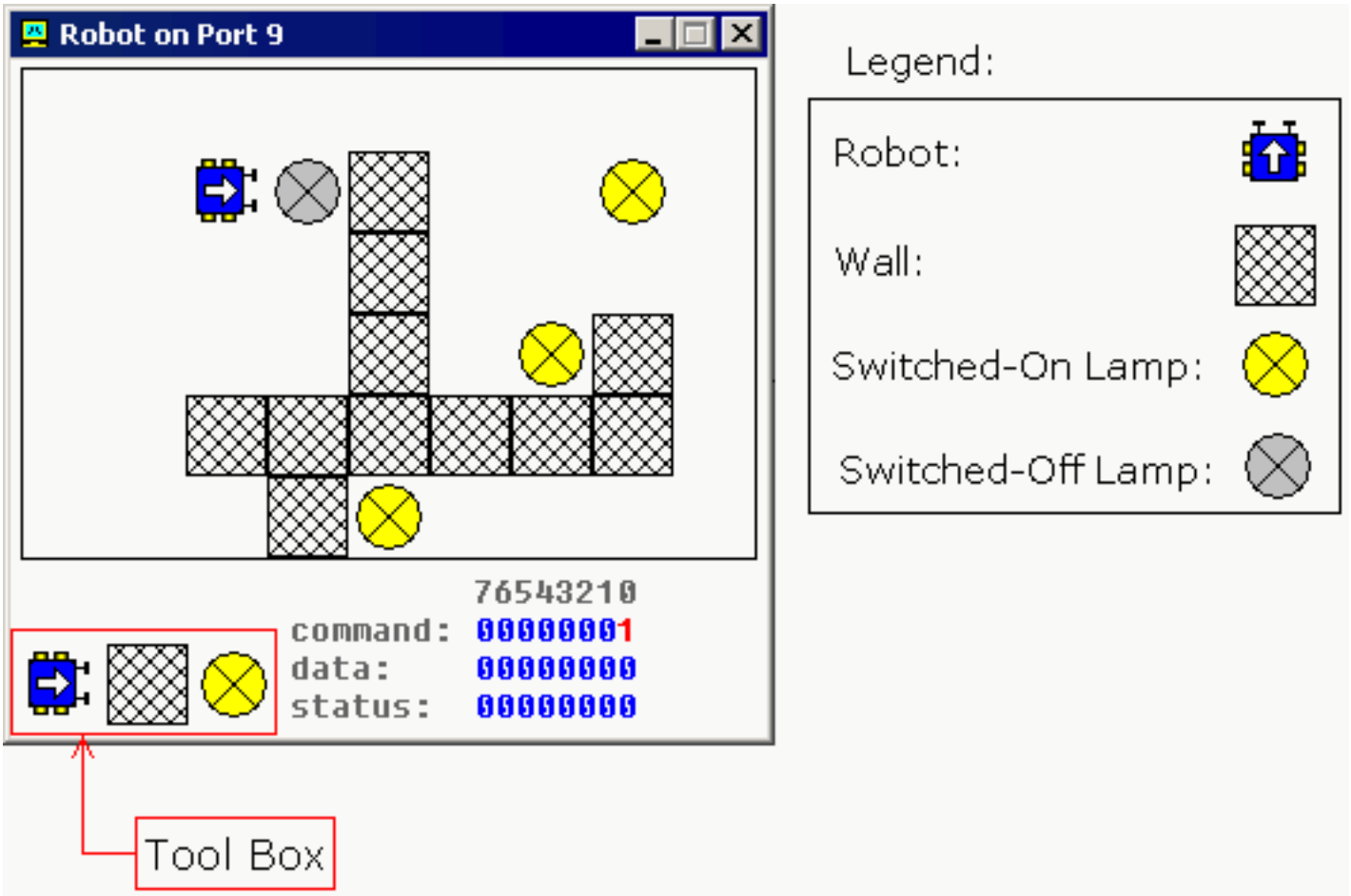
Half step is equal to **11.25** degrees.
Full step is equal to **22.5** degrees.

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

See **stepper_motor.asm** in Samples folder.

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 Samples folder.

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.

<<< Previous Part <<< **>>> Next Part >>>**

```
#MAKE_BIN#
#CS = 500#
#DS = 500#
#SS = 500#    ; stack is set
#SP = FFFF#   ; automatically.
#IP = 0#

; This is an example of controlling
; the robot attached to a computer.

; This code randomly moves the robot
; and makes it to switch off/on the
; lamps.

; Try improving it!

; Keep in mind that robot is a
; mechanical creature and it takes
; some time for it to complete
; a task.

; robot base I/O port:
R_PORT EQU 9

;=====================================

eternal_loop:
; wait until robot
; is ready:
CALL WAIT_ROBOT

; examine the area
; in front of the robot:
MOV AL, 4
OUT R_PORT, AL

CALL WAIT_EXAM

; get result from
; data register:
IN AL, R_PORT + 1
```

```
; nothing found?
CMP AL, 0
JE cont  ; - yes, so continue.

; wall?
CMP AL, 255
JE cont  ; - yes, so continue.

; switched-on lamp?
CMP AL, 7
JNE lamp_off  ; - no, so skip.
; - yes, so switch it off,
;   and turn:
CALL SWITCH_OFF_LAMP
JMP  cont  ; continue

lamp_off: NOP

; if gets here, then we have
; switched-off lamp, because
; all other situations checked
; already:
CALL SWITCH_ON_LAMP

cont:
CALL RANDOM_TURN

CALL WAIT_ROBOT

; try to step forward:
MOV AL, 1
OUT R_PORT, AL

CALL WAIT_ROBOT

; try to step forward again:
MOV AL, 1
OUT R_PORT, AL

JMP eternal_loop ; go again!

;=====================================
```

```
; This procedure does not
; return until robot is ready
; to receive next command:
WAIT_ROBOT PROC
; check if robot busy:
busy: IN AL, R_PORT+2
      TEST AL, 00000010b
      JNZ busy ; busy, so wait.
RET
WAIT_ROBOT ENDP

;=======================================

; This procedure does not
; return until robot completes
; the examination:
WAIT_EXAM PROC
; check if has new data:
busy2: IN AL, R_PORT+2
       TEST AL, 00000001b
       JZ busy2 ; no new data, so wait.
RET
WAIT_EXAM ENDP

;=======================================

; Switch Off the lamp:
SWITCH_OFF_LAMP PROC
MOV AL, 6
OUT R_PORT, AL
RET
SWITCH_OFF_LAMP ENDP

;=======================================

; Switch On the lamp:
SWITCH_ON_LAMP PROC
MOV AL, 5
OUT R_PORT, AL
RET
SWITCH_ON_LAMP ENDP

;=======================================
```

```
; Generates a random turn using
; system timer:
RANDOM_TURN PROC

; get number of clock
; ticks since midnight
; in CX:DX
MOV AH, 0
INT 1Ah

; randomize using XOR:
XOR DH, DL
XOR CH, CL
XOR CH, DH

TEST CH, 2
JZ no_turn

TEST CH, 1
JNZ turn_right

; turn left:
MOV AL, 2
OUT R_PORT, AL
; exit from procedure:
RET

turn_right:
MOV AL, 3
OUT R_PORT, AL

no_turn:
RET
RANDOM_TURN ENDP

;=======================================
```