Documentation for Emu8086

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 microprocessors, 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

Emu8086 reference

Complete 8086 instruction set

Quick reference:

AAA
AAD
AAM
AAS
ADC
ADD
AND
CALL
CBW
CLC
CLD
CLI
CMC
CMP

CMPSB
CMPSW
CWD
DAA
DAS
DEC
DIV
HLT
IDIV
IMUL
IN
INC
INT
INTO
IRET
JA

JAE
JB
JBE
JC
JCXZ
JE
JG
JGE
JL
JLE
JMP
JNA
JNAE
JNB

JNBE
JNC
JNE
JNG
JNGE
JNL
JNLE
JNO
JNP
JNS
JNZ
JO
JP
JPE

JPO
JS
JZ
LAHF
LDS
LEA
LES
LODSB
LODSW
LOOP
LOOPE
LOOPNE
LOOPNZ
LOOPZ

MOV
MOVSB
MOVSW
MUL
NEG
NOP
NOT
OR
OUT
POP
POPA
POPF
PUSH
PUSHA
PUSHF
RCL

RCR
REP
REPE
REPNE
REPNZ
REPZ
RET
RETF
ROL
ROR
SAHF
SAL
SAR
SBB

SCASB
SCASW
SHL
SHR
STC
STD
STI
STOSB
STOSW
SUB
TEST
XCHG
XLATB
XOR

Operand types:

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

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

memory: [BX], [BX+SI+7], variable, etc...(see **Memory Access**).

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

Notes:

- When two operands are required for an instruction they are separated by comma. For example:

  REG, memory

- When there are two operands, both operands must have the same size (except shift and rotate instructions). For example:

  AL, DL
  DX, AX
  m1 DB ?
  AL, m1
  m2 DW ?
  AX, m2

- Some instructions allow several operand combinations. For example:

  memory, immediate
  REG, immediate

  memory, REG
  REG, SREG

- Some examples contain macros, so it is advisable to use **Shift + F8** hot key to *Step Over* (to make macro code execute at maximum speed set **step delay** to zero), otherwise emulator will step through each instruction of a macro. Here is an example that uses PRINTN macro:

```
#make_COM#
include 'emu8086.inc'
ORG 100h
MOV AL, 1
MOV BL, 2
PRINTN 'Hello World!'   ; macro.
MOV CL, 3
PRINTN 'Welcome!'       ; macro.
RET
```

These marks are used to show the state of the flags:

1 - instruction sets this flag to **1**.
0 - instruction sets this flag to **0**.
r - flag value depends on result of the instruction.
? - flag value is undefined (maybe **1** or **0**).

Some instructions generate exactly the same machine code, so disassembler may have a problem decoding to your original code. This is especially important for Conditional Jump instructions (see "Program Flow Control" in Tutorials for more information).

Instructions in alphabetical order:

Instruction	Operands	Description
AAA	No operands	ASCII Adjust after Addition. Corrects result in AH and AL after addition when working with BCD values. It works according to the following Algorithm: if low nibble of AL > 9 or AF = 1 then: • AL = AL + 6 • AH = AH + 1 • AF = 1 • CF = 1 else • AF = 0 • CF = 0 in both cases: clear the high nibble of AL. Example: MOV AX, 15 ; AH = 00, AL = 0Fh AAA ; AH = 01, AL = 05 RET

<table>
<tr><td></td><td></td><td><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>?</td><td>?</td><td>?</td><td>?</td><td>r</td></tr></table></td></tr>
<tr><td>AAD</td><td>No operands</td><td>ASCII Adjust before Division.
Prepares two BCD values for division.

Algorithm:<ul><li>AL = (AH * 10) + AL</li><li>AH = 0</li></ul>Example:

MOV AX, 0105h   ; AH = 01, AL = 05
AAD   ; AH = 00, AL = 0Fh (15)
RET

<table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>?</td><td>r</td><td>r</td><td>?</td><td>r</td><td>?</td></tr></table></td></tr>
<tr><td>AAM</td><td>No operands</td><td>ASCII Adjust after Multiplication.
Corrects the result of multiplication of two BCD values.

Algorithm:<ul><li>AH = AL / 10</li><li>AL = remainder</li></ul>Example:

MOV AL, 15   ; AL = 0Fh
AAM   ; AH = 01, AL = 05
RET

<table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr></table></td></tr>
</table>

		? r r ? r ?
AAS	No operands	ASCII Adjust after Subtraction. Corrects result in AH and AL after subtraction when working with BCD values. Algorithm: if low nibble of AL > 9 or AF = 1 then: - AL = AL - 6 - AH = AH - 1 - AF = 1 - CF = 1 else - AF = 0 - CF = 0 in both cases: clear the high nibble of AL. Example: MOV AX, 02FFh ; AH = 02, AL = 0FFh AAS ; AH = 01, AL = 09 RET C Z S O P A r ? ? ? ? r

<table>
<tr>
<td>ADC</td>
<td>REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate</td>
<td>Add with Carry.

Algorithm:

operand1 = operand1 + operand2 + CF

Example:

STC   ; set CF = 1
MOV AL, 5 ; AL = 5
ADC AL, 1 ; AL = 7
RET

<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
<tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr>
</table>
</td>
</tr>
<tr>
<td>ADD</td>
<td>REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate</td>
<td>Add.

Algorithm:

operand1 = operand1 + operand2

Example:

MOV AL, 5   ; AL = 5
ADD AL, -3 ; AL = 2
RET

<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
<tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr>
</table>
</td>
</tr>
</table>

<table>
<tr>
<td>AND</td>
<td>REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate</td>
<td>Logical AND between all bits of two operands. Result is stored in operand1.

These rules apply:

1 AND 1 = 1
1 AND 0 = 0
0 AND 1 = 0
0 AND 0 = 0

Example:

MOV AL, 'a'   ; AL = 01100001b
AND AL, 11011111b ; AL = 01000001b ('A')
RET

<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td></tr>
<tr><td>0</td><td>r</td><td>r</td><td>0</td><td>r</td></tr>
</table>
</td>
</tr>
<tr>
<td>CALL</td>
<td>procedure name
label
4-byte address</td>
<td>Transfers control to procedure, return address is (IP) is pushed to stack. <i>4-byte address</i> may be entered in this form: 1234h:5678h, first value is a segment second value is an offset (this is a far call, so CS is also pushed to stack).

Example:

#make_COM#
ORG 100h ; for COM file.

CALL p1

ADD AX, 1

RET   ; return to OS.

p1 PROC   ; procedure declaration.
    MOV AX, 1234h
    RET   ; return to caller.
p1 ENDP</td>
</tr>
</table>

		C Z S O P A unchanged
CBW	No operands	Convert byte into word. Algorithm: if high bit of AL = 1 then: • AH = 255 (0FFh) else • AH = 0 Example: MOV AX, 0 ; AH = 0, AL = 0 MOV AL, -5 ; AX = 000FBh (251) CBW ; AX = 0FFFBh (-5) RET C Z S O P A unchanged
CLC	No operands	Clear Carry flag. Algorithm: CF = 0 C 0

CLD	No operands	Clear Direction flag. SI and DI will be incremented by chain instructions: CMPSB, CMPSW, LODSB, LODSW, MOVSB, MOVSW, STOSB, STOSW. Algorithm: DF = 0 D 0
CLI	No operands	Clear Interrupt enable flag. This disables hardware interrupts. Algorithm: IF = 0 I 0
CMC	No operands	Complement Carry flag. Inverts value of CF. Algorithm: if CF = 1 then CF = 0 if CF = 0 then CF = 1 C r

<table>
<tr>
<td>CMP</td>
<td>REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate</td>
<td>
<p>Compare.</p>
<p>Algorithm:</p>
<p>operand1 - operand2</p>
<p>result is not stored anywhere, flags are set (OF, SF, ZF, AF, PF, CF) according to result.</p>
<p>Example:</p>
<pre>
MOV AL, 5
MOV BL, 5
CMP AL, BL ; AL = 5, ZF = 1 (so equal!)
RET
</pre>
<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
<tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr>
</table>
</td>
</tr>
<tr>
<td>CMPSB</td>
<td>No operands</td>
<td>
<p>Compare bytes: ES:[DI] from DS:[SI].</p>
<p>Algorithm:</p>
<ul>
<li>DS:[SI] - ES:[DI]</li>
<li>set flags according to result:
OF, SF, ZF, AF, PF, CF</li>
<li>if DF = 0 then
<ul><li>SI = SI + 1</li><li>DI = DI + 1</li></ul>
else
<ul><li>SI = SI - 1</li><li>DI = DI - 1</li></ul>
</li>
</ul>
<p>Example:
see <b>cmpsb.asm</b> in Samples.</p>
<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
<tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr>
</table>
</td>
</tr>
</table>

<table>
<tr>
<td>CMPSW</td>
<td>No operands</td>
<td>Compare words: ES:[DI] from DS:[SI].

Algorithm:

• DS:[SI] - ES:[DI]
• set flags according to result:
OF, SF, ZF, AF, PF, CF
• if DF = 0 then
○ SI = SI + 2
○ DI = DI + 2
else
○ SI = SI - 2
○ DI = DI - 2

Example:
see <b>cmpsw.asm</b> in Samples.

<table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table></td>
</tr>
<tr>
<td>CWD</td>
<td>No operands</td>
<td>Convert Word to Double word.

Algorithm:

if high bit of AX = 1 then:

• DX = 65535 (0FFFFh)

else

• DX = 0

Example:

MOV DX, 0   ; DX = 0
MOV AX, 0   ; AX = 0
MOV AX, -5 ; DX AX = 00000h:0FFFBh
CWD   ; DX AX = 0FFFFh:0FFFBh</td>
</tr>
</table>

		RET C Z S O P A unchanged

DAA	No operands	Decimal adjust After Addition.
		Corrects the result of addition of two packed BCD values.
		Algorithm:
		if low nibble of AL > 9 or AF = 1 then:
		• AL = AL + 6
		• AF = 1
		if AL > 9Fh or CF = 1 then:
		• AL = AL + 60h
		• CF = 1
		Example:
		MOV AL, 0Fh ; AL = 0Fh (15)
		DAA ; AL = 15h
		RET
		C Z S O P A
		r r r r r r

<table>
<tr>
<td>DAS</td>
<td>No operands</td>
<td>Decimal adjust After Subtraction.
Corrects the result of subtraction of two packed BCD values.

Algorithm:

if low nibble of AL > 9 or AF = 1 then:

• AL = AL - 6
• AF = 1

if AL > 9Fh or CF = 1 then:

• AL = AL - 60h
• CF = 1

Example:

MOV AL, 0FFh ; AL = 0FFh (-1)
DAS ; AL = 99h, CF = 1
RET

<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
<tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr>
</table>
</td>
</tr>
<tr>
<td>DEC</td>
<td>REG
memory</td>
<td>Decrement.

Algorithm:

operand = operand - 1

Example:

MOV AL, 255 ; AL = 0FFh (255 or -1)
DEC AL ; AL = 0FEh (254 or -2)
RET

<table>
<tr><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
</table>
</td>
</tr>
</table>

		r r r r r CF - unchanged!
DIV	REG memory	Unsigned divide. Algorithm: when operand is a **byte**: AL = AX / operand AH = remainder (modulus) when operand is a **word**: AX = (DX AX) / operand DX = remainder (modulus) Example: MOV AX, 203 ; AX = 00CBh MOV BL, 4 DIV BL ; AL = 50 (32h), AH = 3 RET C Z S O P A ? ? ? ? ? ?
HLT	No operands	Halt the System. Example: MOV AX, 5 HLT C Z S O P A unchanged

IDIV	REG memory	Signed divide. Algorithm: when operand is a **byte**: AL = AX / operand AH = remainder (modulus) when operand is a **word**: AX = (DX AX) / operand DX = remainder (modulus) Example: MOV AX, -203 ; AX = 0FF35h MOV BL, 4 IDIV BL ; AL = -50 (0CEh), AH = -3 (0FDh) RET C Z S O P A ? ? ? ? ? ?
IMUL	REG memory	Signed multiply. Algorithm: when operand is a **byte**: AX = AL * operand. when operand is a **word**: (DX AX) = AX * operand. Example: MOV AL, -2 MOV BL, -4 IMUL BL ; AX = 8 RET

		C Z S O P A r ? ? r ? ? CF=OF=0 when result fits into operand of IMUL.
IN	AL, im.byte AL, DX AX, im.byte AX, DX	Input from port into **AL** or **AX**. Second operand is a port number. If required to access port number over 255 - **DX** register should be used. Example: IN AX, 4 ; get status of traffic lights. IN AL, 7 ; get status of stepper-motor. C Z S O P A unchanged
INC	REG memory	Increment. Algorithm: operand = operand + 1 Example: MOV AL, 4 INC AL ; AL = 5 RET Z S O P A r r r r r CF - unchanged!

<table>
<tr>
<td>INT</td>
<td>immediate byte</td>
<td>Interrupt numbered by immediate byte (0..255).

Algorithm:

Push to stack:
○ flags register
○ CS
○ IP
• IF = 0
• Transfer control to interrupt procedure

Example:

MOV AH, 0Eh ; teletype.
MOV AL, 'A'
INT 10h   ; BIOS interrupt.
RET

<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td><td>I</td></tr>
<tr><td colspan="6">unchanged</td><td>0</td></tr>
</table>
</td>
</tr>
<tr>
<td>INTO</td>
<td>No operands</td>
<td>Interrupt 4 if Overflow flag is 1.

Algorithm:

if OF = 1 then INT 4

Example:

; -5 - 127 = -132 (not in -128..127)
; the result of SUB is wrong (124),
; so OF = 1 is set:
MOV AL, -5
SUB AL, 127   ; AL = 7Ch (124)
INTO   ; process error.
RET</td>
</tr>
</table>

IRET	No operands	Interrupt Return. Algorithm: Pop from stack: ○ IP ○ CS ○ flags register C Z S O P A popped
JA	label	Short Jump if first operand is Above second operand (as set by CMP instruction). Unsigned. Algorithm: if (CF = 0) and (ZF = 0) then jump Example: include 'emu8086.inc' #make_COM# ORG 100h MOV AL, 250 CMP AL, 5 JA label1 PRINT 'AL is not above 5' JMP exit label1: PRINT 'AL is above 5' exit: RET C Z S O P A unchanged

JAE	label	Short Jump if first operand is Above or Equal to second operand (as set by CMP instruction). Unsigned. Algorithm: if CF = 0 then jump Example: include 'emu8086.inc' #make_COM# ORG 100h MOV AL, 5 CMP AL, 5 JAE label1 PRINT 'AL is not above or equal to 5' JMP exit label1: PRINT 'AL is above or equal to 5' exit: RET C Z S O P A unchanged
JB	label	Short Jump if first operand is Below second operand (as set by CMP instruction). Unsigned. Algorithm: if CF = 1 then jump Example: include 'emu8086.inc' #make_COM# ORG 100h MOV AL, 1 CMP AL, 5

<table>
<tr>
<td></td>
<td></td>
<td>JB label1
PRINT 'AL is not below 5'
JMP exit
label1:
PRINT 'AL is below 5'
exit:
RET

<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
<tr><td colspan="6">unchanged</td></tr>
</table>
</td>
</tr>
<tr>
<td>JBE</td>
<td>label</td>
<td>Short Jump if first operand is Below or Equal to second operand (as set by CMP instruction). Unsigned.

Algorithm:

if CF = 1 or ZF = 1 then jump

Example:

include 'emu8086.inc'
#make_COM#
ORG 100h
MOV AL, 5
CMP AL, 5
JBE label1
PRINT 'AL is not below or equal to 5'
JMP exit
label1:
PRINT 'AL is below or equal to 5'
exit:
RET

<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
<tr><td colspan="6">unchanged</td></tr>
</table>
</td>
</tr>
</table>

JC	label	Short Jump if Carry flag is set to 1. Algorithm: if CF = 1 then jump Example: include 'emu8086.inc' #make_COM# ORG 100h MOV AL, 255 ADD AL, 1 JC label1 PRINT 'no carry.' JMP exit label1: PRINT 'has carry.' exit: RET C Z S O P A unchanged
JCXZ	label	Short Jump if CX register is 0. Algorithm: if CX = 0 then jump Example: include 'emu8086.inc' #make_COM# ORG 100h MOV CX, 0 JCXZ label1 PRINT 'CX is not zero.' JMP exit

		label1: PRINT 'CX is zero.' exit: RET C Z S O P A unchanged

JE	label	Short Jump if first operand is Equal to second operand (as set by CMP instruction). Signed/Unsigned.
		
		Algorithm:
		
		      if ZF = 1 then jump
		
		Example:
		
		   include 'emu8086.inc'
		   #make_COM#
		   ORG 100h
		   MOV AL, 5
		   CMP AL, 5
		   JE label1
		   PRINT 'AL is not equal to 5.'
		   JMP exit
		label1:
		   PRINT 'AL is equal to 5.'
		exit:
		   RET
		
		C Z S O P A
		unchanged

JG	label	Short Jump if first operand is Greater then second operand (as set by CMP instruction). Signed. Algorithm: if (ZF = 0) and (SF = OF) then jump Example: include 'emu8086.inc' #make_COM# ORG 100h MOV AL, 5 CMP AL, -5 JG label1 PRINT 'AL is not greater -5.' JMP exit label1: PRINT 'AL is greater -5.' exit: RET C Z S O P A unchanged
JGE	label	Short Jump if first operand is Greater or Equal to second operand (as set by CMP instruction). Signed. Algorithm: if SF = OF then jump Example: include 'emu8086.inc' #make_COM# ORG 100h MOV AL, 2 CMP AL, -5

		JGE label1 PRINT 'AL < -5' JMP exit label1: PRINT 'AL >= -5' exit: RET C Z S O P A unchanged
JL	label	Short Jump if first operand is Less then second operand (as set by CMP instruction). Signed. Algorithm: if SF <> OF then jump Example: include 'emu8086.inc' #make_COM# ORG 100h MOV AL, -2 CMP AL, 5 JL label1 PRINT 'AL >= 5.' JMP exit label1: PRINT 'AL < 5.' exit: RET C Z S O P A unchanged

JLE	label	Short Jump if first operand is Less or Equal to second operand (as set by CMP instruction). Signed.
		Algorithm:
		    if SF <> OF or ZF = 1 then jump
		Example:
		   include 'emu8086.inc'
		   #make_COM#
		   ORG 100h
		   MOV AL, -2
		   CMP AL, 5
		   JLE label1
		   PRINT 'AL > 5.'
		   JMP exit
		label1:
		   PRINT 'AL <= 5.'
		exit:
		   RET
		C Z S O P A
		unchanged

JMP	label
	4-byte address
		Unconditional Jump. Transfers control to another part of the program. *4-byte address* may be entered in this form: 1234h:5678h, first value is a segment second value is an offset.
		Algorithm:
		    always jump
		Example:
		   include 'emu8086.inc'
		   #make_COM#
		   ORG 100h

<table>
<tr><td></td><td></td><td>MOV AL, 5
JMP label1   ; jump over 2 lines!
PRINT 'Not Jumped!'
MOV AL, 0
label1:
PRINT 'Got Here!'
RET
<table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table></td></tr>
<tr><td>JNA</td><td>label</td><td>Short Jump if first operand is Not Above second operand (as set by CMP instruction). Unsigned.
Algorithm:
if CF = 1 or ZF = 1 then jump
Example:
include 'emu8086.inc'
#make_COM#
ORG 100h
MOV AL, 2
CMP AL, 5
JNA label1
PRINT 'AL is above 5.'
JMP exit
label1:
PRINT 'AL is not above 5.'
exit:
RET
<table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table></td></tr>
</table>

JNAE	label	Short Jump if first operand is Not Above and Not Equal to second operand (as set by CMP instruction). Unsigned. Algorithm: if CF = 1 then jump Example: include 'emu8086.inc' #make_COM# ORG 100h MOV AL, 2 CMP AL, 5 JNAE label1 PRINT 'AL >= 5.' JMP exit label1: PRINT 'AL < 5.' exit: RET C Z S O P A unchanged
JNB	label	Short Jump if first operand is Not Below second operand (as set by CMP instruction). Unsigned. Algorithm: if CF = 0 then jump Example: include 'emu8086.inc' #make_COM# ORG 100h MOV AL, 7 CMP AL, 5

		JNB label1 PRINT 'AL < 5.' JMP exit label1: PRINT 'AL >= 5.' exit: RET C Z S O P A unchanged
JNBE	label	Short Jump if first operand is Not Below and Not Equal to second operand (as set by CMP instruction). Unsigned. Algorithm: if (CF = 0) and (ZF = 0) then jump Example: include 'emu8086.inc' #make_COM# ORG 100h MOV AL, 7 CMP AL, 5 JNBE label1 PRINT 'AL <= 5.' JMP exit label1: PRINT 'AL > 5.' exit: RET C Z S O P A unchanged

JNC	label	Short Jump if Carry flag is set to 0. Algorithm: if CF = 0 then jump Example: include 'emu8086.inc' #make_COM# ORG 100h MOV AL, 2 ADD AL, 3 JNC label1 PRINT 'has carry.' JMP exit label1: PRINT 'no carry.' exit: RET C Z S O P A unchanged
JNE	label	Short Jump if first operand is Not Equal to second operand (as set by CMP instruction). Signed/Unsigned. Algorithm: if ZF = 0 then jump Example: include 'emu8086.inc' #make_COM# ORG 100h MOV AL, 2 CMP AL, 3 JNE label1

<table>
<tr>
<td></td>
<td></td>
<td>

```
   PRINT 'AL = 3.'
   JMP exit
label1:
   PRINT 'Al <> 3.'
exit:
   RET
```

<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
<tr><td colspan="6">unchanged</td></tr>
</table>

</td>
</tr>
<tr>
<td>JNG</td>
<td>label</td>
<td>

Short Jump if first operand is Not Greater then second operand (as set by CMP instruction). Signed.

Algorithm:

```
        if (ZF = 1) and (SF <> OF) then jump
```

Example:

```
   include 'emu8086.inc'
   #make_COM#
   ORG 100h
   MOV AL, 2
   CMP AL, 3
   JNG  label1
   PRINT 'AL > 3.'
   JMP exit
label1:
   PRINT 'Al <= 3.'
exit:
   RET
```

<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
<tr><td colspan="6">unchanged</td></tr>
</table>

</td>
</tr>
</table>

JNGE

label

Short Jump if first operand is Not Greater and Not Equal to second operand (as set by CMP instruction). Signed.

Algorithm:

```
        if SF <> OF then jump
```

Example:

```
   include 'emu8086.inc'
   #make_COM#
   ORG 100h
   MOV AL, 2
   CMP AL, 3
   JNGE  label1
   PRINT 'AL >= 3.'
   JMP exit
label1:
   PRINT 'Al < 3.'
exit:
   RET
```

C	Z	S	O	P	A
unchanged					

JNL

label

Short Jump if first operand is Not Less then second operand (as set by CMP instruction). Signed.

Algorithm:

```
        if SF = OF then jump
```

Example:

```
   include 'emu8086.inc'
   #make_COM#
   ORG 100h
   MOV AL, 2
   CMP AL, -3
   JNL label1
   PRINT 'AL < -3.'
   JMP exit
label1:
   PRINT 'Al >= -3.'
exit:
   RET
```

C	Z	S	O	P	A
unchanged					

JNLE

label

Short Jump if first operand is Not Less and Not Equal to second operand (as set by CMP instruction). Signed.

Algorithm:

```
        if (SF = OF) and (ZF = 0) then jump
```

Example:

```
   include 'emu8086.inc'
   #make_COM#
   ORG 100h
   MOV AL, 2
   CMP AL, -3
   JNLE label1
   PRINT 'AL <= -3.'
   JMP exit
label1:
   PRINT 'Al > -3.'
exit:
   RET
```

C	Z	S	O	P	A
unchanged					

JNO	label	Short Jump if Not Overflow. Algorithm: if OF = 0 then jump Example: ; -5 - 2 = -7 (inside -128..127) ; the result of SUB is correct, ; so OF = 0: include 'emu8086.inc' #make_COM# ORG 100h MOV AL, -5 SUB AL, 2 ; AL = 0F9h (-7) JNO label1 PRINT 'overflow!' JMP exit label1: PRINT 'no overflow.' exit: RET C Z S O P A unchanged
		Short Jump if No Parity (odd). Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. Algorithm: if PF = 0 then jump Example: include 'emu8086.inc'

JNP	label	#make_COM# ORG 100h MOV AL, 00000111b ; AL = 7 OR AL, 0 ; just set flags. JNP label1 PRINT 'parity even.' JMP exit label1: PRINT 'parity odd.' exit: RET C Z S O P A unchanged
JNS	label	Short Jump if Not Signed (if positive). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. Algorithm: if SF = 0 then jump Example: include 'emu8086.inc' #make_COM# ORG 100h MOV AL, 00000111b ; AL = 7 OR AL, 0 ; just set flags. JNS label1 PRINT 'signed.' JMP exit label1: PRINT 'not signed.' exit: RET C Z S O P A unchanged

JNZ	label	Short Jump if Not Zero (not equal). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. Algorithm: if ZF = 0 then jump Example: include 'emu8086.inc' #make_COM# ORG 100h MOV AL, 00000111b ; AL = 7 OR AL, 0 ; just set flags. JNZ label1 PRINT 'zero.' JMP exit label1: PRINT 'not zero.' exit: RET C Z S O P A unchanged
		Short Jump if Overflow. Algorithm: if OF = 1 then jump Example: ; -5 - 127 = -132 (not in -128..127) ; the result of SUB is wrong (124), ; so OF = 1 is set: include 'emu8086.inc'

JO	label	#make_COM# org 100h MOV AL, -5 SUB AL, 127 ; AL = 7Ch (124) JO label1 PRINT 'no overflow.' JMP exit label1: PRINT 'overflow!' exit: RET C Z S O P A unchanged

JP	label	Short Jump if Parity (even). Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.
		
		Algorithm:
		
		        if PF = 1 then jump
		
		Example:
		
		   include 'emu8086.inc'
		   #make_COM#
		   ORG 100h
		   MOV AL, 00000101b ; AL = 5
		   OR AL, 0 ; just set flags.
		   JP label1
		   PRINT 'parity odd.'
		   JMP exit
		label1:
		   PRINT 'parity even.'
		exit:
		   RET
		
		C Z S O P A

		unchanged
JPE	label	Short Jump if Parity Even. Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions. Algorithm: if PF = 1 then jump Example: include 'emu8086.inc' #make_COM# ORG 100h MOV AL, 00000101b ; AL = 5 OR AL, 0 ; just set flags. JPE label1 PRINT 'parity odd.' JMP exit label1: PRINT 'parity even.' exit: RET C Z S O P A unchanged

<table>
<tr>
<td>JPO</td>
<td>label</td>
<td>
Short Jump if Parity Odd. Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.

Algorithm:
<pre>
        if PF = 0 then jump
</pre>
Example:
<pre>
   include 'emu8086.inc'
   #make_COM#
   ORG 100h
   MOV AL, 00000111b   ; AL = 7
   OR AL, 0   ; just set flags.
   JPO label1
   PRINT 'parity even.'
   JMP exit
label1:
   PRINT 'parity odd.'
exit:
   RET
</pre>
<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
<tr><td colspan="6">unchanged</td></tr>
</table>
</td>
</tr>
<tr>
<td>JS</td>
<td>label</td>
<td>
Short Jump if Signed (if negative). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.

Algorithm:
<pre>
        if SF = 1 then jump
</pre>
Example:
<pre>
   include 'emu8086.inc'
   #make_COM#
   ORG 100h
   MOV AL, 10000000b   ; AL = -128
</pre>
</td>
</tr>
</table>

<table>
<tr>
<td></td>
<td></td>
<td>OR AL, 0   ; just set flags.
JS label1
PRINT 'not signed.'
JMP exit
label1:
PRINT 'signed.'
exit:
RET
<table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table></td>
</tr>
<tr>
<td>JZ</td>
<td>label</td>
<td>Short Jump if Zero (equal). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.
Algorithm:
if ZF = 1 then jump
Example:
include 'emu8086.inc'
#make_COM#
ORG 100h
MOV AL, 5
CMP AL, 5
JZ label1
PRINT 'AL is not equal to 5.'
JMP exit
label1:
PRINT 'AL is equal to 5.'
exit:
RET
<table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table></td>
</tr>
</table>

LAHF	No operands	Load AH from 8 low bits of Flags register. Algorithm: AH = flags register AH bit: 7 6 5 4 3 2 1 0 [SF] [ZF] [0] [AF] [0] [PF] [1] [CF] bits 1, 3, 5 are reserved. C Z S O P A unchanged
LDS	REG, memory	Load memory double word into word register and DS. Algorithm: • REG = first word • DS = second word Example: #make_COM# ORG 100h LDS AX, m RET m DW 1234h DW 5678h END AX is set to 1234h, DS is set to 5678h.

		C Z S O P A unchanged
LEA	REG, memory	Load Effective Address. Algorithm: • REG = address of memory (offset) Generally this instruction is replaced by MOV when assembling when possible. Example: #make_COM# ORG 100h LEA AX, m RET m DW 1234h END AX is set to: 0104h. LEA instruction takes 3 bytes, RET takes 1 byte, we start at 100h, so the address of 'm' is 104h. C Z S O P A unchanged

<table>
<tr>
<td>LES</td>
<td>REG, memory</td>
<td>Load memory double word into word register and ES.

Algorithm:

• REG = first word
• ES = second word

Example:

#make_COM#
ORG 100h

LES AX, m

RET

m DW 1234h
   DW 5678h

END

AX is set to 1234h, ES is set to 5678h.

<table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Load byte at DS:[SI] into AL. Update SI.

Algorithm:

• AL = DS:[SI]
• if DF = 0 then
    ○ SI = SI + 1
  else
    ○ SI = SI - 1

Example:</td>
</tr>
</table>

LODSB	No operands	#make_COM# ORG 100h LEA SI, a1 MOV CX, 5 MOV AH, 0Eh m: LODSB INT 10h LOOP m RET a1 DB 'H', 'e', 'l', 'l', 'o' C Z S O P A unchanged
LODSW	No operands	Load word at DS:[SI] into AX. Update SI. Algorithm: • AX = DS:[SI] • if DF = 0 then ○ SI = SI + 2 else ○ SI = SI - 2 Example: #make_COM# ORG 100h LEA SI, a1 MOV CX, 5 REP LODSW ; finally there will be 555h in AX. RET

		a1 dw 111h, 222h, 333h, 444h, 555h C Z S O P A unchanged
LOOP	label	Decrease CX, jump to label if CX not zero. Algorithm: - CX = CX - 1 - if CX <> 0 then ○ jump else ○ no jump, continue Example: include 'emu8086.inc' #make_COM# ORG 100h MOV CX, 5 label1: PRINTN 'loop!' LOOP label1 RET C Z S O P A unchanged

LOOPE	label	Decrease CX, jump to label if CX not zero and Equal (ZF = 1). Algorithm: - CX = CX - 1 - if (CX <> 0) and (ZF = 1) then ○ jump else ○ no jump, continue Example: ; Loop until result fits into AL alone, ; or 5 times. The result will be over 255 ; on third loop (100+100+100), ; so loop will exit. include 'emu8086.inc' #make_COM# ORG 100h MOV AX, 0 MOV CX, 5 label1: PUTC '*' ADD AX, 100 CMP AH, 0 LOOPE label1 RET C Z S O P A unchanged

LOOPNE	label	Decrease CX, jump to label if CX not zero and Not Equal (ZF = 0). Algorithm: - CX = CX - 1 - if (CX <> 0) and (ZF = 0) then ○ jump else ○ no jump, continue Example: ; Loop until '7' is found, ; or 5 times. include 'emu8086.inc' #make_COM# ORG 100h MOV SI, 0 MOV CX, 5 label1: PUTC '*' MOV AL, v1[SI] INC SI ; next byte (SI=SI+1). CMP AL, 7 LOOPNE label1 RET v1 db 9, 8, 7, 6, 5 C Z S O P A unchanged

<table>
<tr>
<td>LOOPNZ</td>
<td>label</td>
<td>

Decrease CX, jump to label if CX not zero and ZF = 0.

Algorithm:

```
• CX = CX - 1
• if (CX <> 0) and (ZF = 0) then
      ○ jump
  else
      ○ no jump, continue
```

Example:

```
; Loop until '7' is found,
; or 5 times.

   include 'emu8086.inc'
   #make_COM#
   ORG 100h
   MOV SI, 0
   MOV CX, 5
label1:
   PUTC '*'
   MOV AL, v1[SI]
   INC SI         ; next byte (SI=SI+1).
   CMP AL, 7
   LOOPNZ label1
   RET
   v1 db 9, 8, 7, 6, 5
```

<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
<tr><td colspan="6">unchanged</td></tr>
</table>

</td>
</tr>
</table>

LOOPZ	label	Decrease CX, jump to label if CX not zero and ZF = 1. Algorithm: • CX = CX - 1 • if (CX <> 0) and (ZF = 1) then ○ jump else ○ no jump, continue Example: ; Loop until result fits into AL alone, ; or 5 times. The result will be over 255 ; on third loop (100+100+100), ; so loop will exit. include 'emu8086.inc' #make_COM# ORG 100h MOV AX, 0 MOV CX, 5 label1: PUTC '*' ADD AX, 100 CMP AH, 0 LOOPZ label1 RET C Z S O P A unchanged

MOV	REG, memory memory, REG REG, REG memory, immediate REG, immediate SREG, memory memory, SREG REG, SREG SREG, REG	Copy operand2 to operand1. The MOV instruction <u>cannot</u>: - set the value of the CS and IP registers. - copy value of one segment register to another segment register (should copy to general register first). - copy immediate value to segment register (should copy to general register first). Algorithm: operand1 = operand2 Example: #make_COM# ORG 100h MOV AX, 0B800h ; set AX = B800h (VGA memory). MOV DS, AX ; copy value of AX to DS. MOV CL, 'A' ; CL = 41h (ASCII code). MOV CH, 01011111b ; CL = color attribute. MOV BX, 15Eh ; BX = position on screen. MOV [BX], CX ; w.[0B800h:015Eh] = CX. RET ; returns to operating system. C Z S O P A unchanged

MOVSB	No operands	Copy byte at DS:[SI] to ES:[DI]. Update SI and DI. Algorithm: - ES:[DI] = DS:[SI] - if DF = 0 then ○ SI = SI + 1 ○ DI = DI + 1 else ○ SI = SI - 1 ○ DI = DI - 1 Example: #make_COM# ORG 100h LEA SI, a1 LEA DI, a2 MOV CX, 5 REP MOVSB RET a1 DB 1,2,3,4,5 a2 DB 5 DUP(0) C Z S O P A unchanged

MOVSW	No operands	Copy **word** at DS:[SI] to ES:[DI]. Update SI and DI. Algorithm: - ES:[DI] = DS:[SI] - if DF = 0 then ○ SI = SI + 2 ○ DI = DI + 2 else ○ SI = SI - 2 ○ DI = DI - 2 Example: #make_COM# ORG 100h LEA SI, a1 LEA DI, a2 MOV CX, 5 REP MOVSW RET a1 DW 1,2,3,4,5 a2 DW 5 DUP(0) C Z S O P A unchanged

MUL	REG memory	Unsigned multiply. Algorithm: when operand is a **byte**: AX = AL * operand. when operand is a **word**: (DX AX) = AX * operand. Example: MOV AL, 200 ; AL = 0C8h MOV BL, 4 MUL BL ; AX = 0320h (800) RET C Z S O P A r ? ? r ? ? CF=OF=0 when high section of the result is zero.
NEG	REG memory	Negate. Makes operand negative (two's complement). Algorithm: • Invert all bits of the operand • Add 1 to inverted operand Example: MOV AL, 5 ; AL = 05h NEG AL ; AL = 0FBh (-5) NEG AL ; AL = 05h (5) RET C Z S O P A r r r r r r

NOP	No operands	No Operation. Algorithm: • Do nothing Example: ; do nothing, 3 times: NOP NOP NOP RET C | Z | S | O | P | A unchanged
NOT	REG memory	Invert each bit of the operand. Algorithm: • if bit is 1 turn it to 0. • if bit is 0 turn it to 1. Example: MOV AL, 00011011b NOT AL ; AL = 11100100b RET C | Z | S | O | P | A unchanged

<table>
<tr>
<td>OR</td>
<td>REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate</td>
<td>Logical OR between all bits of two operands. Result is stored in first operand.

These rules apply:

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

Example:

MOV AL, 'A'   ; AL = 01000001b
OR AL, 00100000b ; AL = 01100001b ('a')
RET

<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
<tr><td>0</td><td>r</td><td>r</td><td>0</td><td>r</td><td>?</td></tr>
</table>
</td>
</tr>
<tr>
<td>OUT</td>
<td>im.byte, AL
im.byte, AX
DX, AL
DX, AX</td>
<td>Output from <b>AL</b> or <b>AX</b> to port.
First operand is a port number. If required to access port number over 255 - <b>DX</b> register should be used.

Example:

MOV AX, 0FFFh ; Turn on all
OUT 4, AX   ; traffic lights.

MOV AL, 100b ; Turn on the third
OUT 7, AL   ; magnet of the stepper-motor.

<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
<tr><td colspan="6">unchanged</td></tr>
</table>
</td>
</tr>
</table>

<table>
<tr>
<td>POP</td>
<td>REG
SREG
memory</td>
<td>
Get 16 bit value from the stack.

Algorithm:

• operand = SS:[SP] (top of the stack)
• SP = SP + 2

Example:

MOV AX, 1234h
PUSH AX
POP DX   ; DX = 1234h
RET

<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
<tr><td colspan="6">unchanged</td></tr>
</table>
</td>
</tr>
<tr>
<td>POPA</td>
<td>No operands</td>
<td>
Pop all general purpose registers DI, SI, BP, SP, BX, DX, CX, AX from the stack.
SP value is ignored, it is Popped but not set to SP register).

Note: this instruction works only on <b>80186</b> CPU and later!

Algorithm:

• POP DI
• POP SI
• POP BP
• POP xx (SP value ignored)
• POP BX
• POP DX
• POP CX
• POP AX

<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
<tr><td colspan="6">unchanged</td></tr>
</table>
</td>
</tr>
</table>

<table>
<tr>
<td>POPF</td>
<td>No operands</td>
<td>
Get flags register from the stack.

Algorithm:
<ul>
<li>flags = SS:[SP] (top of the stack)</li>
<li>SP = SP + 2</li>
</ul>
<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
<tr><td colspan="6">popped</td></tr>
</table>
</td>
</tr>
<tr>
<td>PUSH</td>
<td>REG
SREG
memory
immediate</td>
<td>
Store 16 bit value in the stack.

Note: <b>PUSH immediate</b> works only on 80186 CPU and later!

Algorithm:
<ul>
<li>SP = SP - 2</li>
<li>SS:[SP] (top of the stack) = operand</li>
</ul>
Example:

MOV AX, 1234h
PUSH AX
POP DX   ; DX = 1234h
RET

<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
<tr><td colspan="6">unchanged</td></tr>
</table>
</td>
</tr>
</table>

PUSHA	No operands	Push all general purpose registers AX, CX, DX, BX, SP, BP, SI, DI in the stack. Original value of SP register (before PUSHA) is used. Note: this instruction works only on **80186** CPU and later! Algorithm: • PUSH AX • PUSH CX • PUSH DX • PUSH BX • PUSH SP • PUSH BP • PUSH SI • PUSH DI C Z S O P A unchanged
PUSHF	No operands	Store flags register in the stack. Algorithm: • SP = SP - 2 • SS:[SP] (top of the stack) = flags C Z S O P A unchanged

RCL	memory, immediate REG, immediate memory, CL REG, CL	Rotate operand1 left through Carry Flag. The number of rotates is set by operand2. When **immediate** is greater then 1, assembler generates several **RCL xx, 1** instructions because 8086 has machine code only for this instruction (the same principle works for all other shift/rotate instructions). Algorithm: shift all bits left, the bit that goes off is set to CF and previous value of CF is inserted to the right-most position. Example: STC ; set carry (CF=1). MOV AL, 1Ch ; AL = 00011100b RCL AL, 1 ; AL = 00111001b, CF=0. RET C O r r OF=0 if first operand keeps original sign.
RCR	memory, immediate REG, immediate memory, CL REG, CL	Rotate operand1 right through Carry Flag. The number of rotates is set by operand2. Algorithm: shift all bits right, the bit that goes off is set to CF and previous value of CF is inserted to the left-most position. Example: STC ; set carry (CF=1). MOV AL, 1Ch ; AL = 00011100b RCR AL, 1 ; AL = 10001110b, CF=0. RET

<table>
<tr><td></td><td></td><td><table><tr><td>C</td><td>O</td></tr><tr><td>r</td><td>r</td></tr></table>OF=0 if first operand keeps original sign.</td></tr>
<tr><td>REP</td><td>chain instruction</td><td>Repeat following MOVSB, MOVSW, LODSB, LODSW, STOSB, STOSW instructions CX times.
Algorithm:
check_cx:
if CX <> 0 then<ul><li>do following chain instruction</li><li>CX = CX - 1</li><li>go back to check_cx</li></ul>else<ul><li>exit from REP cycle</li></ul><table><tr><td>Z</td></tr><tr><td>r</td></tr></table></td></tr>
<tr><td>REPE</td><td>chain instruction</td><td>Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 1 (result is Equal), maximum CX times.
Algorithm:
check_cx:
if CX <> 0 then<ul><li>do following chain instruction</li><li>CX = CX - 1</li><li>if ZF = 1 then:<ul><li>go back to check_cx</li></ul>else</li></ul></td></tr>
</table>

		○ exit from REPE cycle else • exit from REPE cycle Example: see **cmpsb.asm** in Samples. Z r
REPNE	chain instruction	Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 0 (result is Not Equal), maximum CX times. Algorithm: check_cx: if CX <> 0 then • do following chain instruction • CX = CX - 1 • if ZF = 0 then: ○ go back to check_cx else ○ exit from REPNE cycle else • exit from REPNE cycle Z r

REPNZ	chain instruction	Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 0 (result is Not Zero), maximum CX times. Algorithm: check_cx: if CX <> 0 then • do following chain instruction • CX = CX - 1 • if ZF = 0 then: ○ go back to check_cx else ○ exit from REPNZ cycle else • exit from REPNZ cycle Z r
REPZ	chain instruction	Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 1 (result is Zero), maximum CX times. Algorithm: check_cx: if CX <> 0 then • do following chain instruction • CX = CX - 1 • if ZF = 1 then: ○ go back to check_cx else ○ exit from REPZ cycle

		else • exit from REPZ cycle Z r
RET	No operands or even immediate	Return from near procedure. Algorithm: • Pop from stack: ○ IP • if immediate operand is present: SP = SP + operand Example: #make_COM# ORG 100h ; for COM file. CALL p1 ADD AX, 1 RET ; return to OS. p1 PROC ; procedure declaration. MOV AX, 1234h RET ; return to caller. p1 ENDP C Z S O P A unchanged

<table>
<tr>
<td>RETF</td>
<td>No operands
or even immediate</td>
<td>Return from Far procedure.

Algorithm:

• Pop from stack:
○ IP
○ CS
• if immediate operand is present: SP = SP + operand

<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
<tr><td colspan="6">unchanged</td></tr>
</table>
</td>
</tr>
<tr>
<td>ROL</td>
<td>memory, immediate
REG, immediate

memory, CL
REG, CL</td>
<td>Rotate operand1 left. The number of rotates is set by operand2.

Algorithm:

shift all bits left, the bit that goes off is set to CF and the same bit is inserted to the right-most position.

Example:

MOV AL, 1Ch   ; AL = 00011100b
ROL AL, 1   ; AL = 00111000b, CF=0.
RET

<table>
<tr><td>C</td><td>O</td></tr>
<tr><td>r</td><td>r</td></tr>
</table>
OF=0 if first operand keeps original sign.</td>
</tr>
</table>

<table>
<tr>
<td>ROR</td>
<td>memory, immediate
REG, immediate

memory, CL
REG, CL</td>
<td>Rotate operand1 right. The number of rotates is set by operand2.

Algorithm:

shift all bits right, the bit that goes off is set to CF and the same bit is inserted to the left-most position.

Example:

MOV AL, 1Ch   ; AL = 00011100b
ROR AL, 1   ; AL = 00001110b, CF=0.
RET

<table><tr><td>C</td><td>O</td></tr><tr><td>r</td><td>r</td></tr></table>OF=0 if first operand keeps original sign.</td>
</tr>
<tr>
<td>SAHF</td>
<td>No operands</td>
<td>Store AH register into low 8 bits of Flags register.

Algorithm:

flags register = AH

AH bit: 7   6   5   4   3   2   1   0
[SF] [ZF] [0] [AF] [0] [PF] [1] [CF]

bits 1, 3, 5 are reserved.

<table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table></td>
</tr>
</table>

SAL	memory, immediate REG, immediate memory, CL REG, CL	Shift Arithmetic operand1 Left. The number of shifts is set by operand2. Algorithm: • Shift all bits left, the bit that goes off is set to CF. • Zero bit is inserted to the right-most position. Example: MOV AL, 0E0h ; AL = 11100000b SAL AL, 1 ; AL = 11000000b, CF=1. RET C O r r OF=0 if first operand keeps original sign.
SAR	memory, immediate REG, immediate memory, CL REG, CL	Shift Arithmetic operand1 Right. The number of shifts is set by operand2. Algorithm: • Shift all bits right, the bit that goes off is set to CF. • The sign bit that is inserted to the left-most position has the same value as before shift. Example: MOV AL, 0E0h ; AL = 11100000b SAR AL, 1 ; AL = 11110000b, CF=0. MOV BL, 4Ch ; BL = 01001100b SAR BL, 1 ; BL = 00100110b, CF=0. RET C O r r

		OF=0 if first operand keeps original sign.
SBB	REG, memory memory, REG REG, REG memory, immediate REG, immediate	Subtract with Borrow. Algorithm: operand1 = operand1 - operand2 - CF Example: STC MOV AL, 5 SBB AL, 3 ; AL = 5 - 3 - 1 = 1 RET C Z S O P A r r r r r r
SCASB	No operands	Compare bytes: AL from ES:[DI]. Algorithm: • ES:[DI] - AL • set flags according to result: OF, SF, ZF, AF, PF, CF • if DF = 0 then ○ DI = DI + 1 else ○ DI = DI - 1 C Z S O P A r r r r r r

<table>
<tr>
<td>SCASW</td>
<td>No operands</td>
<td>Compare words: AX from ES:[DI].

Algorithm:
<ul><li>ES:[DI] - AX</li><li>set flags according to result:
OF, SF, ZF, AF, PF, CF</li><li>if DF = 0 then<ul><li>DI = DI + 2</li></ul>else<ul><li>DI = DI - 2</li></ul></li></ul><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table></td>
</tr>
<tr>
<td>SHL</td>
<td>memory, immediate
REG, immediate

memory, CL
REG, CL</td>
<td>Shift operand1 Left. The number of shifts is set by operand2.

Algorithm:
<ul><li>Shift all bits left, the bit that goes off is set to CF.</li><li>Zero bit is inserted to the right-most position.</li></ul>Example:

MOV AL, 11100000b
SHL AL, 1   ; AL = 11000000b, CF=1.

RET

<table><tr><td>C</td><td>O</td></tr><tr><td>r</td><td>r</td></tr></table>OF=0 if first operand keeps original sign.</td>
</tr>
</table>

SHR	memory, immediate REG, immediate memory, CL REG, CL	Shift operand1 Right. The number of shifts is set by operand2. Algorithm: • Shift all bits right, the bit that goes off is set to CF. • Zero bit is inserted to the left-most position. Example: MOV AL, 00000111b SHR AL, 1 ; AL = 00000011b, CF=1. RET C O r r OF=0 if first operand keeps original sign.
STC	No operands	Set Carry flag. Algorithm: CF = 1 C 1

STD	No operands	Set Direction flag. SI and DI will be decremented by chain instructions: CMPSB, CMPSW, LODSB, LODSW, MOVSB, MOVSW, STOSB, STOSW. Algorithm: DF = 1 D 1
STI	No operands	Set Interrupt enable flag. This enables hardware interrupts. Algorithm: IF = 1 I 1
STOSB	No operands	Store byte in AL into ES:[DI]. Update DI. Algorithm: • ES:[DI] = AL • if DF = 0 then ○ DI = DI + 1 else ○ DI = DI - 1 Example: #make_COM# ORG 100h LEA DI, a1 MOV AL, 12h MOV CX, 5

		REP STOSB RET a1 DB 5 dup(0) C | Z | S | O | P | A unchanged
STOSW	No operands	Store word in AX into ES:[DI]. Update DI. Algorithm: • ES:[DI] = AX • if DF = 0 then ○ DI = DI + 2 else ○ DI = DI - 2 Example: #make_COM# ORG 100h LEA DI, a1 MOV AX, 1234h MOV CX, 5 REP STOSW RET a1 DW 5 dup(0) C | Z | S | O | P | A unchanged

<table>
<tr>
<td>SUB</td>
<td>REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate</td>
<td>Subtract.

Algorithm:

operand1 = operand1 - operand2

Example:

MOV AL, 5
SUB AL, 1   ; AL = 4

RET

<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
<tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr>
</table>
</td>
</tr>
<tr>
<td>TEST</td>
<td>REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate</td>
<td>Logical AND between all bits of two operands for flags only. These flags are effected: <b>ZF, SF, PF.</b> Result is not stored anywhere.

These rules apply:

1 AND 1 = 1
1 AND 0 = 0
0 AND 1 = 0
0 AND 0 = 0

Example:

MOV AL, 00000101b
TEST AL, 1   ; ZF = 0.
TEST AL, 10b   ; ZF = 1.
RET

<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td></tr>
<tr><td>0</td><td>r</td><td>r</td><td>0</td><td>r</td></tr>
</table>
</td>
</tr>
</table>

<table>
<tr>
<td>XCHG</td>
<td>REG, memory
memory, REG
REG, REG</td>
<td>Exchange values of two operands.

Algorithm:

operand1 < - > operand2

Example:

MOV AL, 5
MOV AH, 2
XCHG AL, AH ; AL = 2, AH = 5
XCHG AL, AH ; AL = 5, AH = 2
RET

<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
<tr><td colspan="6">unchanged</td></tr>
</table>
</td>
</tr>
<tr>
<td>XLATB</td>
<td>No operands</td>
<td>Translate byte from table.
Copy value of memory byte at DS:[BX + unsigned AL] to AL register.

Algorithm:

AL = DS:[BX + unsigned AL]

Example:

#make_COM#
ORG 100h
LEA BX, dat
MOV AL, 2
XLATB ; AL = 33h

RET

dat DB 11h, 22h, 33h, 44h, 55h

<table>
<tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr>
<tr><td colspan="6">unchanged</td></tr>
</table>
</td>
</tr>
</table>

<table>
<tr><td></td><td></td><td></td></tr>
<tr><td>XOR</td><td>REG, memory
memory, REG
REG, REG
memory, immediate
REG, immediate</td><td>Logical XOR (Exclusive OR) between all bits of two operands. Result is stored in first operand.

These rules apply:

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

Example:

MOV AL, 00000111b
XOR AL, 00000010b   ; AL = 00000101b
RET

<table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>0</td><td>r</td><td>r</td><td>0</td><td>r</td><td>?</td></tr></table></td></tr>
</table>

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!

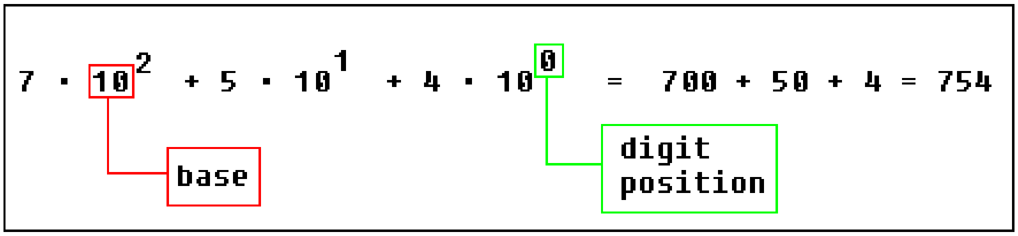
Using sticks to count was a great idea for its time. And using symbols instead of real sticks was much better. One of the best ways to represent a number today is by using the modern decimal system. Why? Because it includes the major breakthrough of using a symbol to represent the idea of counting ***nothing***. 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

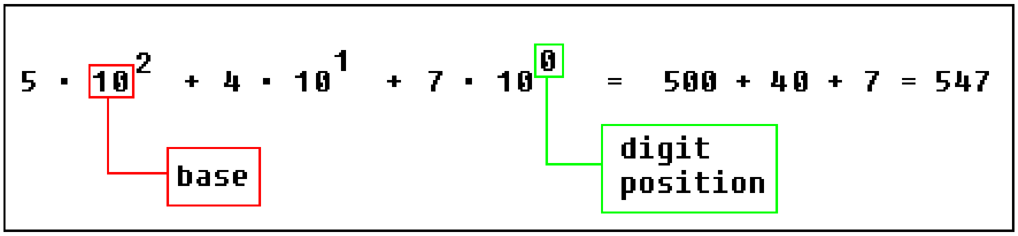
Most people today use decimal representation to count. In the decimal system there are 10 digits:

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:

$10^0 = 1$ $0^0 = 1$ $x^0 = 1$

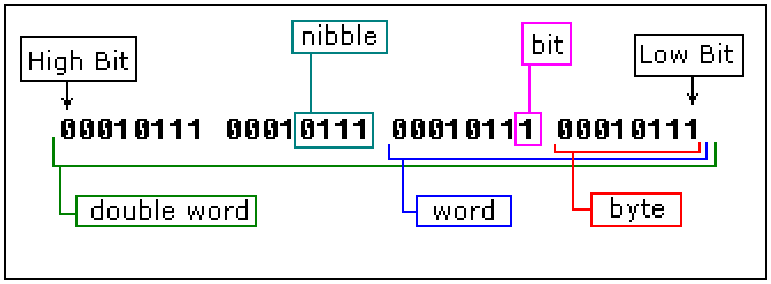
Binary System

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

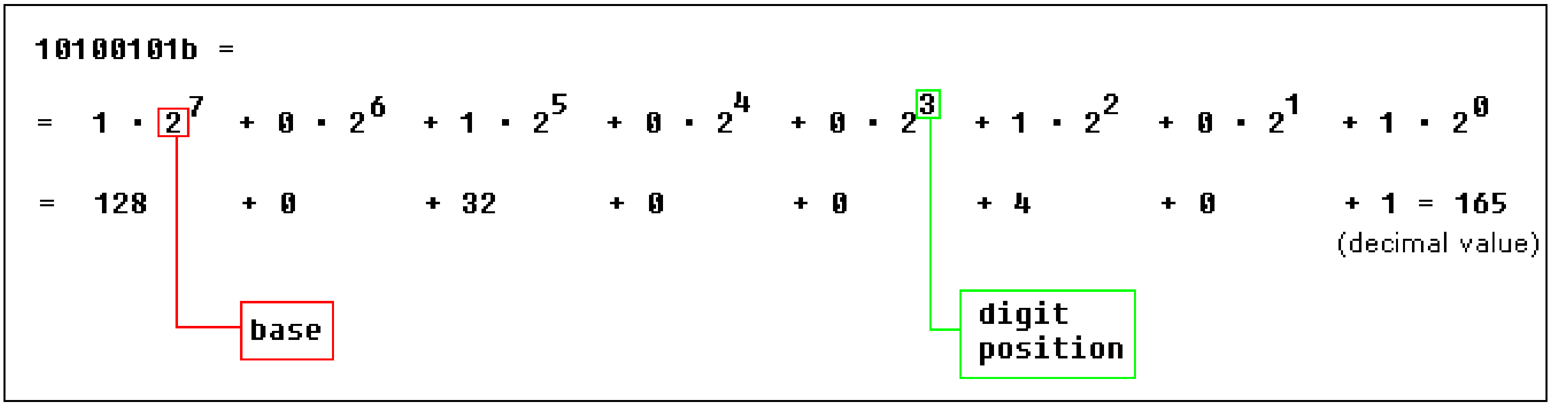
And thus the **base** is **2**.

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:



$$10100101b =$$

$$= 1 \cdot \boxed{2}^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^{\boxed{3}} + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

$$= 128 + 0 + 32 + 0 + 0 + 4 + 0 + 1 = 165$$

Hexadecimal System

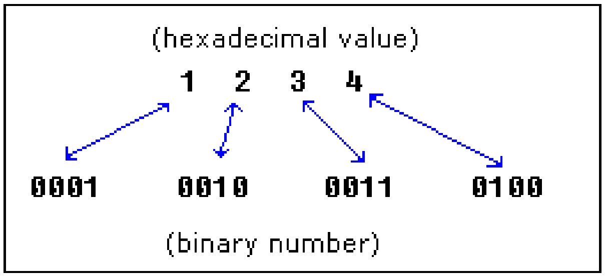
Hexadecimal System uses 16 digits:

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

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

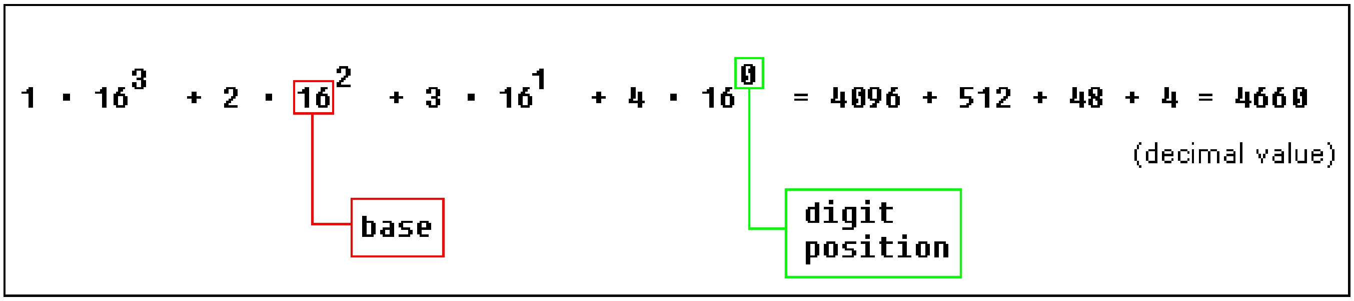
Hexadecimal numbers are compact and easy to read.
It is very easy to convert numbers from binary system to hexadecimal system and vice-versa, every nibble (4 bits) can be converted to a hexadecimal digit using this table:

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:

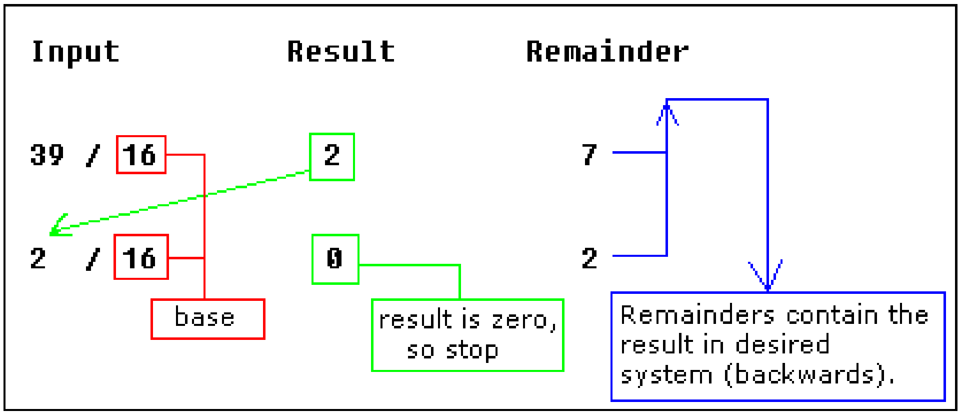


Converting from Decimal System to Any Other

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.

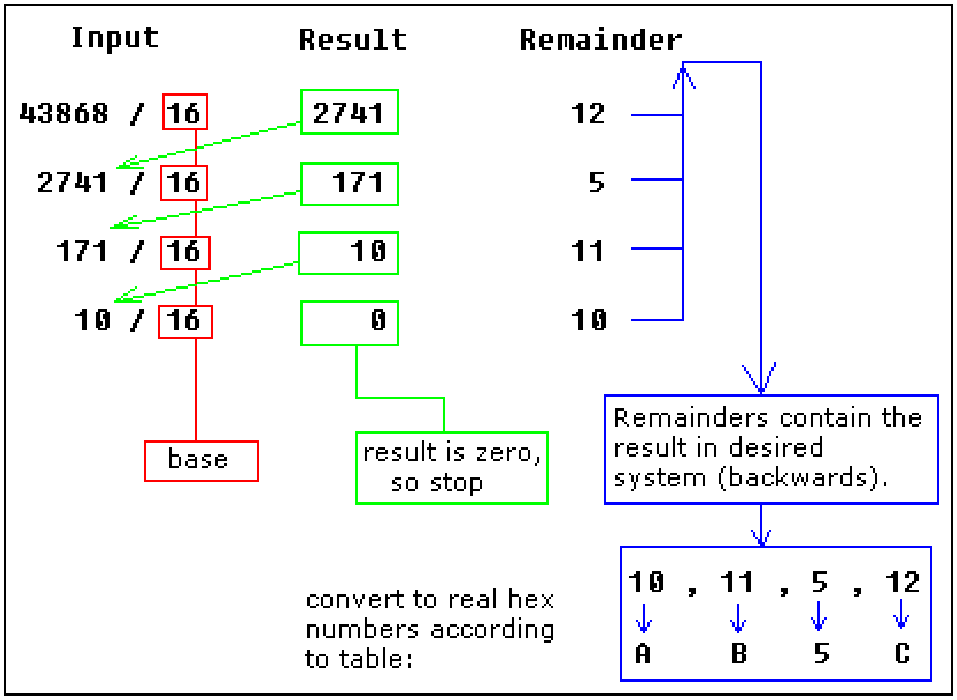
The **remainders** are then used to represent a value in that system.

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



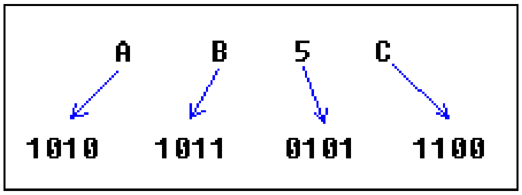
As you see we got this hexadecimal number: **27h**.
All remainders were below **10** in the above example, so we do not use any letters.

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**

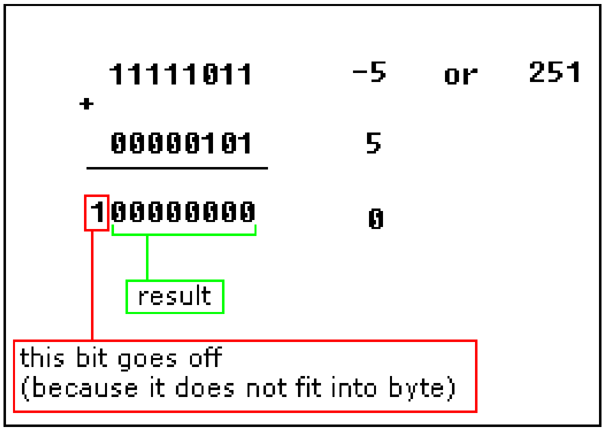
Signed Numbers

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**.

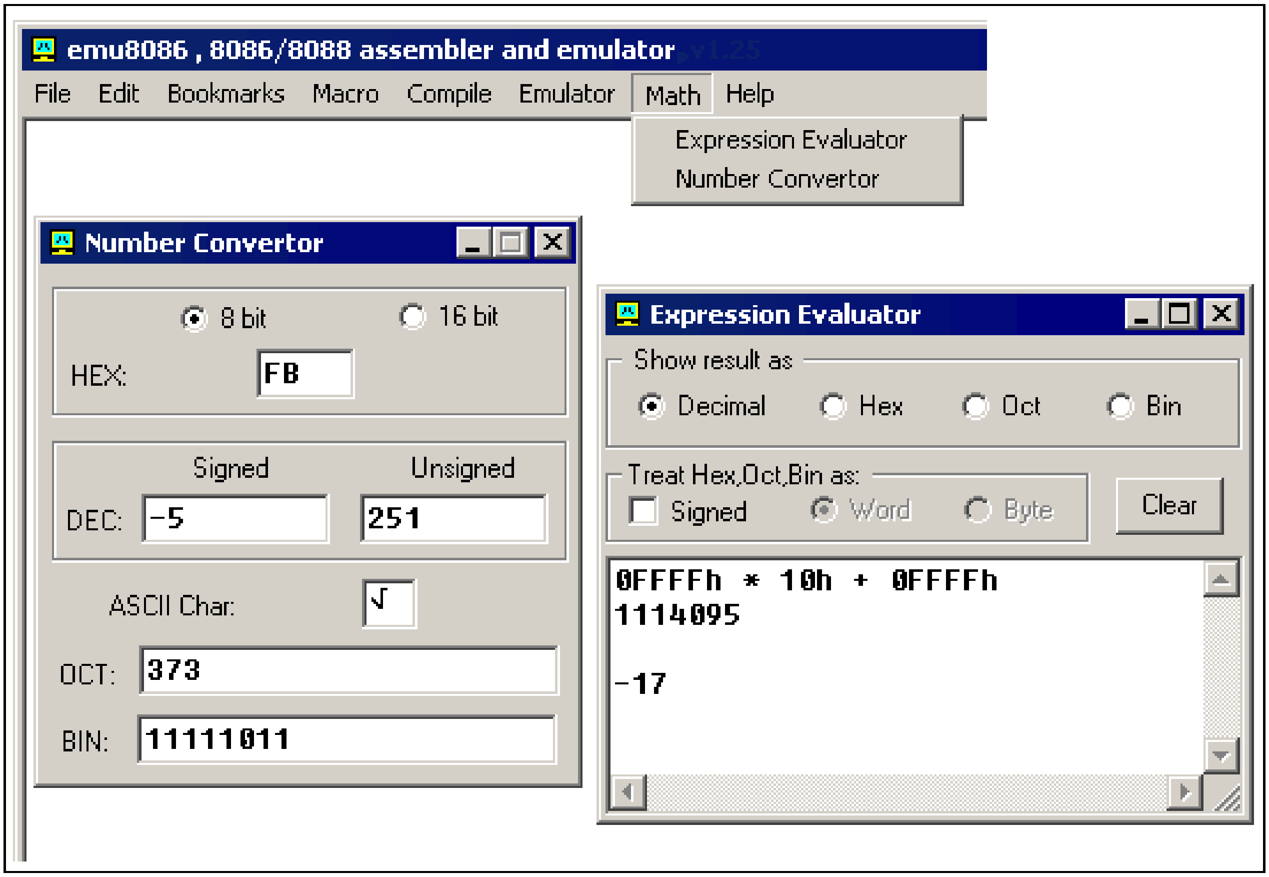
Using this complex way to represent negative numbers has some meaning, in math when you add "**- 5**" to "**5**" you should get zero.
This is what happens when processor adds two bytes **5** and **251**, the result gets over **255**, because of the overflow processor gets zero!



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

● 8 bit ○ 16 bit

HEX: FB

	Signed	Unsigned
DEC:	-5	251

ASCII Char: √

OCT: 373

BIN: 11111011

Expression Evaluator

Show result as: ● Decimal ○ Hex ○ Oct ○ Bin

Treat Hex,Oct,Bin as: □ Signed ● Word ○ Byte

Clear

```
0FFFFh * 10h + 0FFFFh
1114095

-17
```

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 >>>

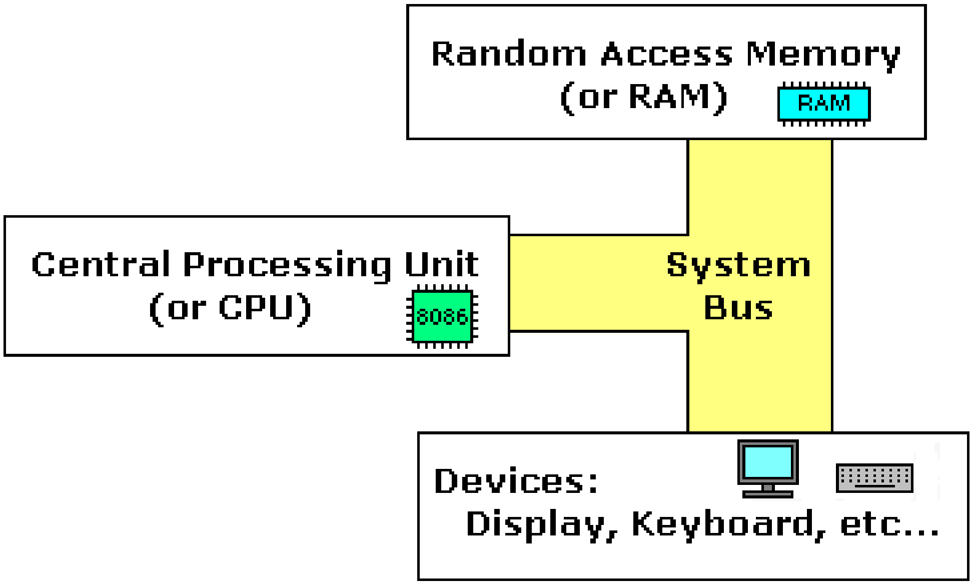
8086 Assembler Tutorial for Beginners (Part 1)

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.

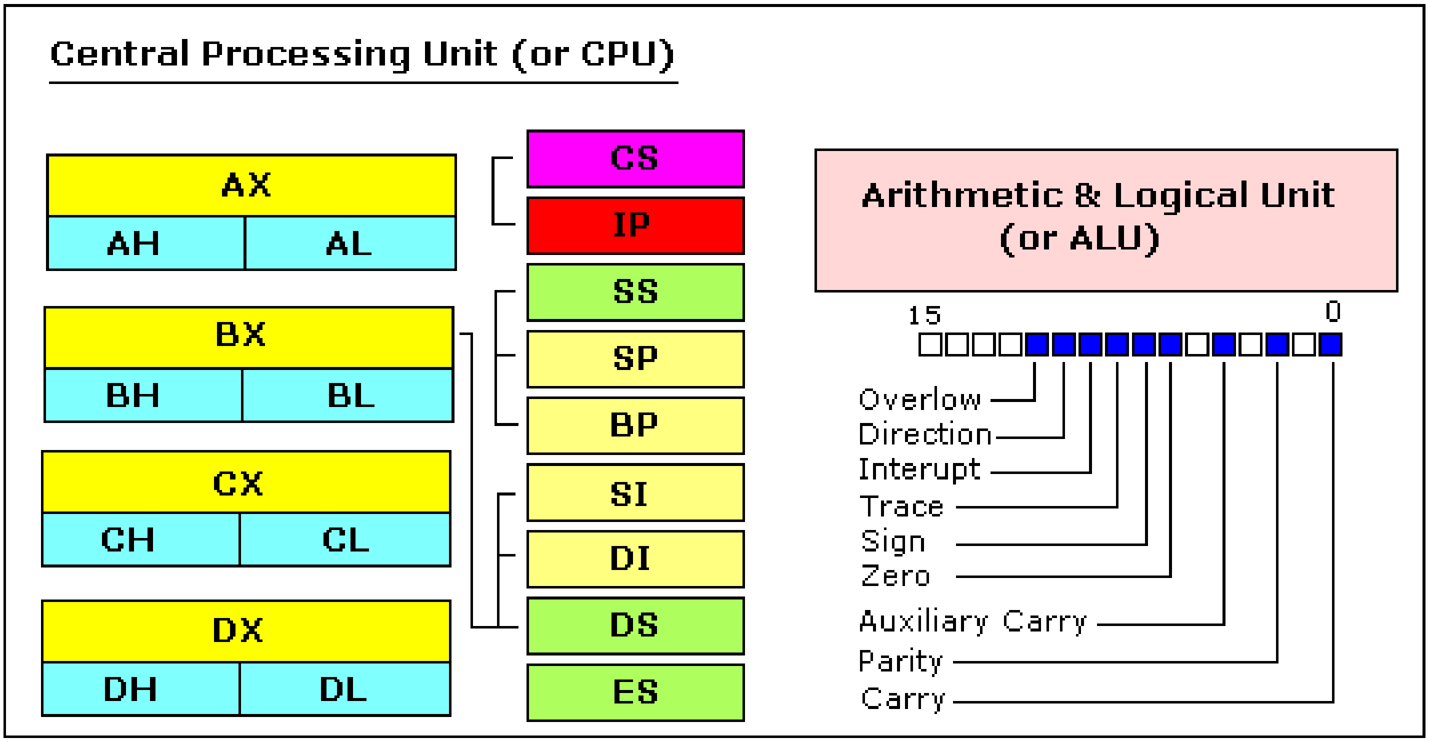
What is an assembly language?

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.

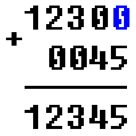
Because registers are located inside the CPU, they are much faster than memory. Accessing a memory location requires the use of a system bus, so it takes much longer. Accessing data in a register usually takes no time. Therefore, you should try to keep variables in the registers. Register sets are very 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.

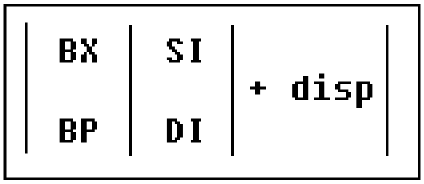
Displacement is a **signed** value, so it can be both positive or negative.

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:



BX	SI	+ disp
BP	DI	

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**:

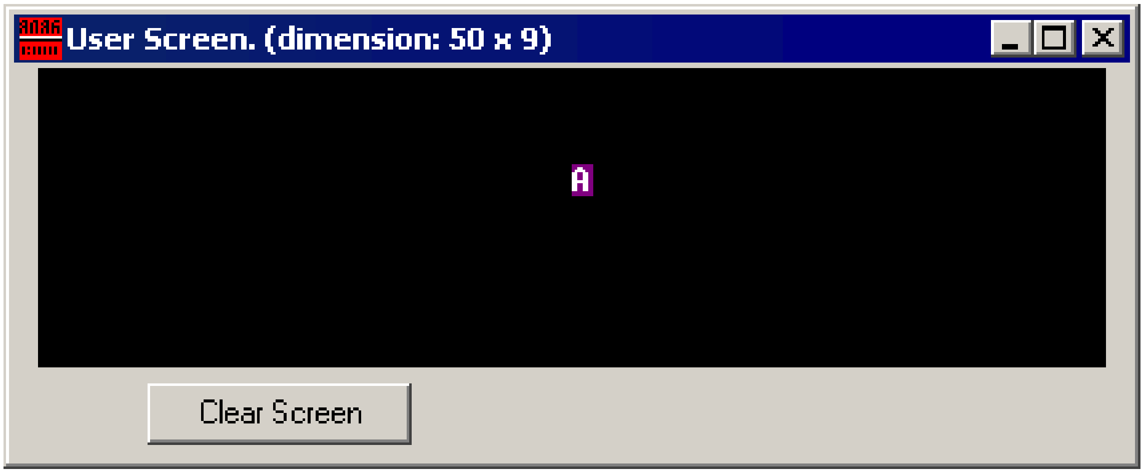
1. Select the above text using mouse, click before the text and drag it down until everything is selected.

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.

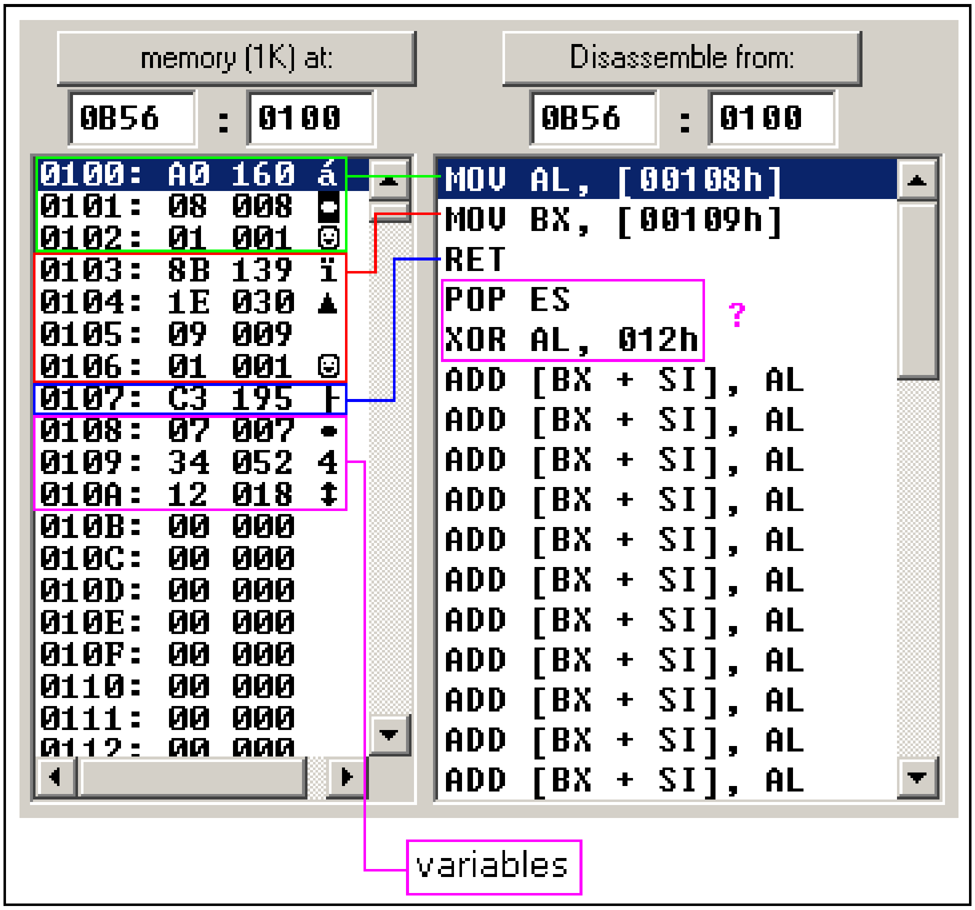
name - can be any letter or digit combination, though it should start with a letter. It's possible to declare unnamed variables by not specifying the name (this variable will have an address but no name).

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:



```
0B56 : 0100                 0B56 : 0100

0100: A0 160 á              MOV AL, [00108h]
0101: 08 008 ◘              MOV BX, [00109h]
0102: 01 001 ☺              RET
0103: 8B 139 ï              POP ES
0104: 1E 030 ▲              XOR AL, 012h
0105: 09 009                ADD [BX + SI], AL
0106: 01 001 ☺              ADD [BX + SI], AL
0107: C3 195 ├              ADD [BX + SI], AL
0108: 07 007 •              ADD [BX + SI], AL
0109: 34 052 4              ADD [BX + SI], AL
010A: 12 018 ↕              ADD [BX + SI], AL
010B: 00 000                ADD [BX + SI], AL
010C: 00 000                ADD [BX + SI], AL
010D: 00 000                ADD [BX + SI], AL
010E: 00 000                ADD [BX + SI], AL
010F: 00 000                ADD [BX + SI], AL
0110: 00 000
0111: 00 000
0112: 00 000
```

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.

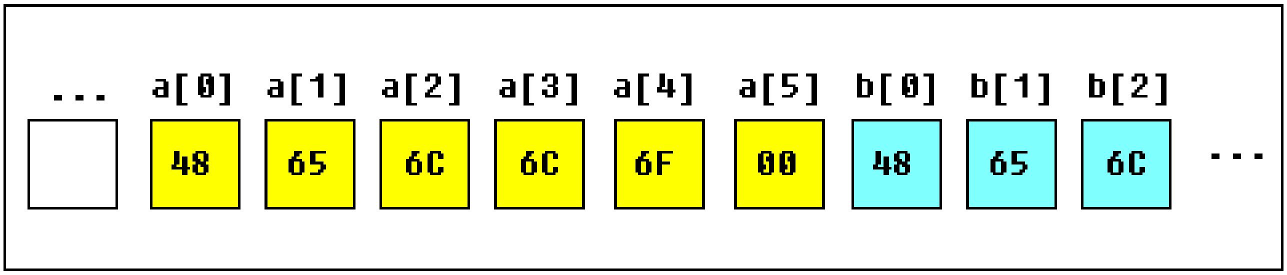
Arrays

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 >
```

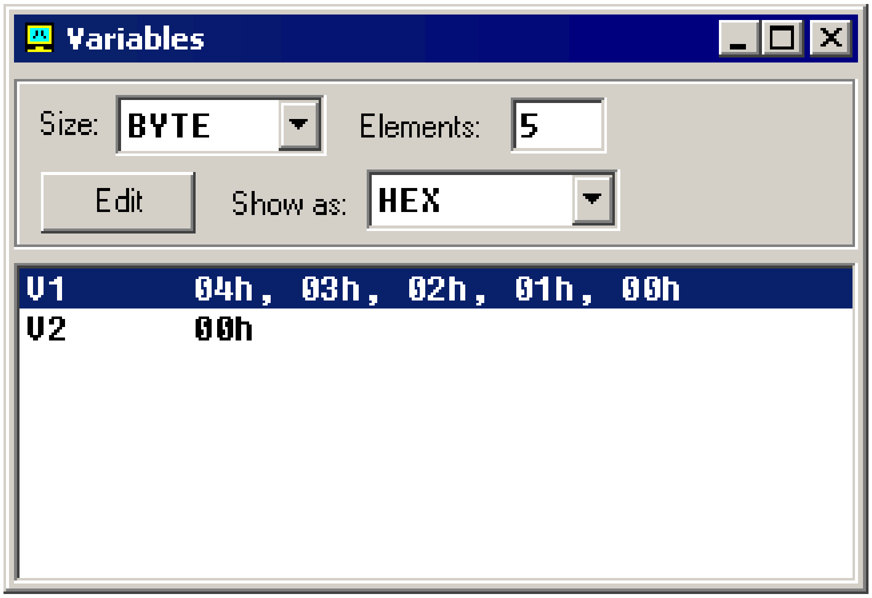
For example:

```
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.



```
V1      04h, 03h, 02h, 01h, 00h
V2      00h
```

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.
```

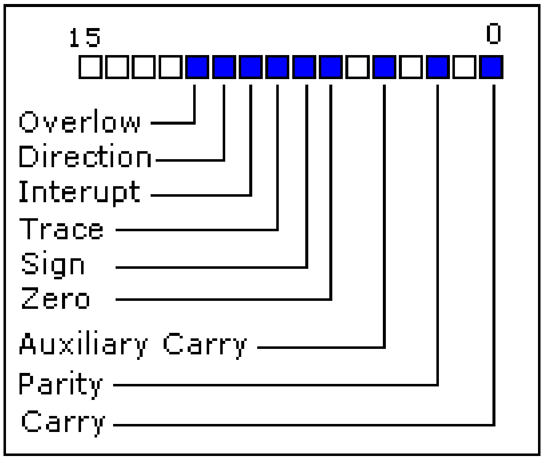
First compiler processes the declarations (these are just regular the macros that are expanded to procedures). When compiler gets to **CALL** instruction it replaces the procedure name with the address of the code where the procedure is declared. When **CALL** instruction is executed control is transferred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

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 ...

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.

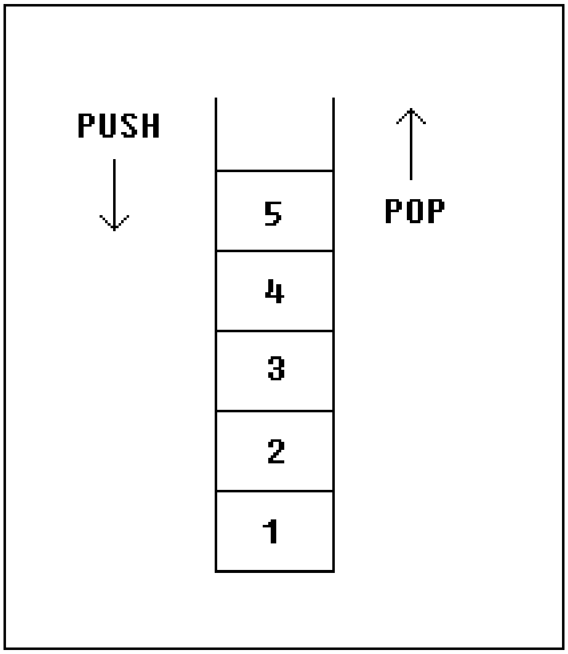
SREG: DS, ES, SS, (except CS).

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.

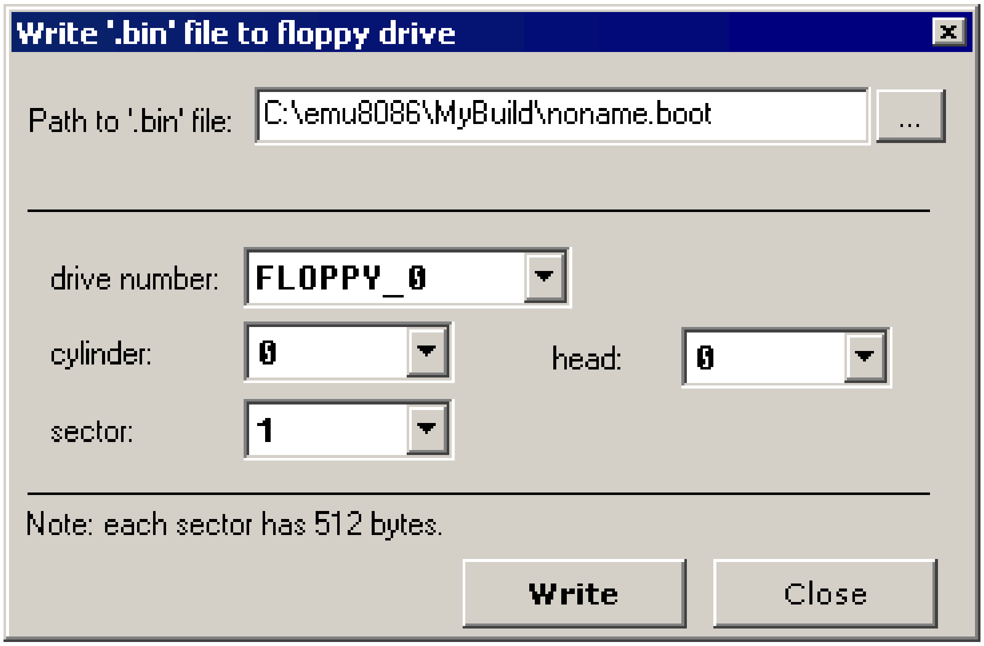
If you are curious, you may write the virtual floppy (**FLOPPY_0**) or "**.boot**" file to a real floppy disk and boot your computer from it, I recommend using "RawWrite for Windows" from: **http://uranus.it.swin.edu.au/~jn/linux/rawwrite.htm**
(recent builds now work under all versions of Windows!)

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:



Path to '.bin' file: C:\emu8086\MyBuild\noname.boot

drive number: FLOPPY_0
cylinder: 0
head: 0
sector: 1

Note: each sector has 512 bytes.

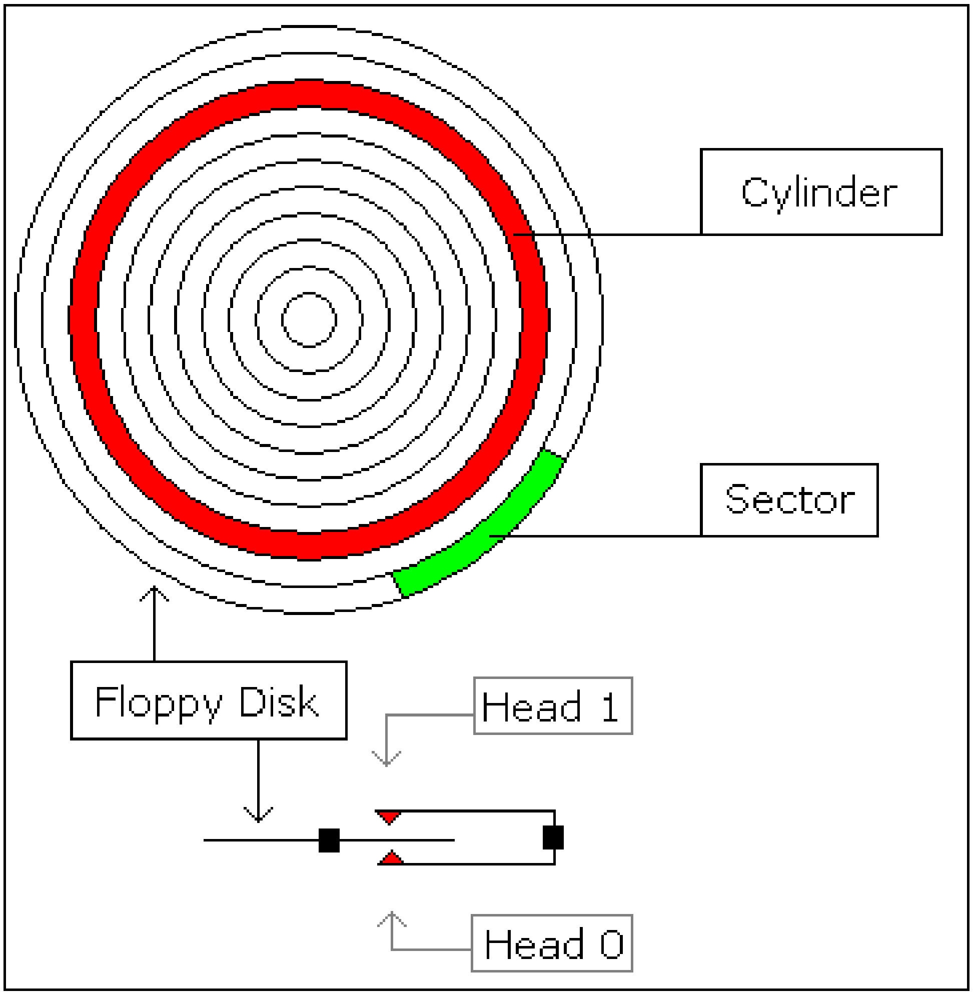
You can also use this to write "**.boot**" files.

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)

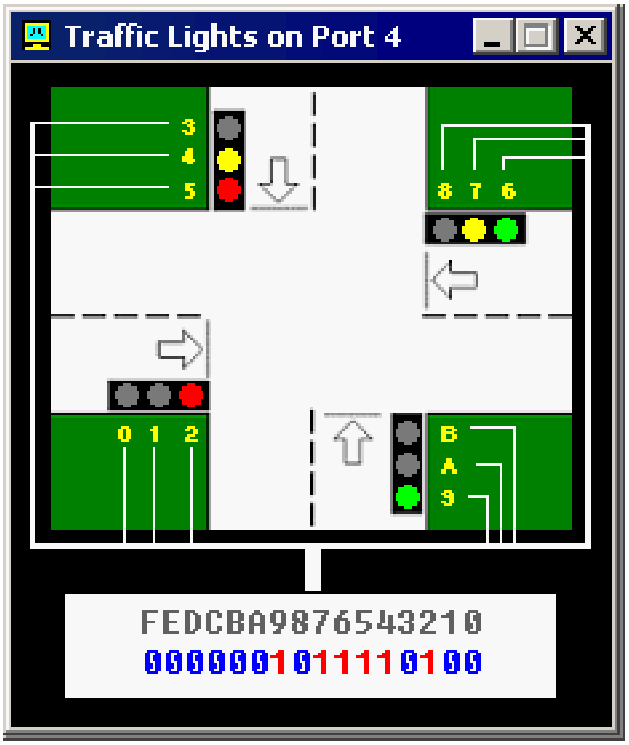
Controlling External Devices

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...

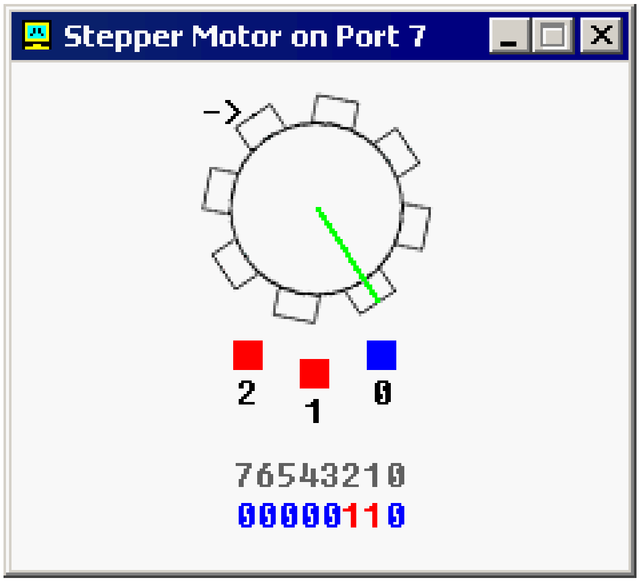
Traffic Lights



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 100001100001b
      DW 110011110011b
      DW 001100001100b
      DW 011110011110b

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.

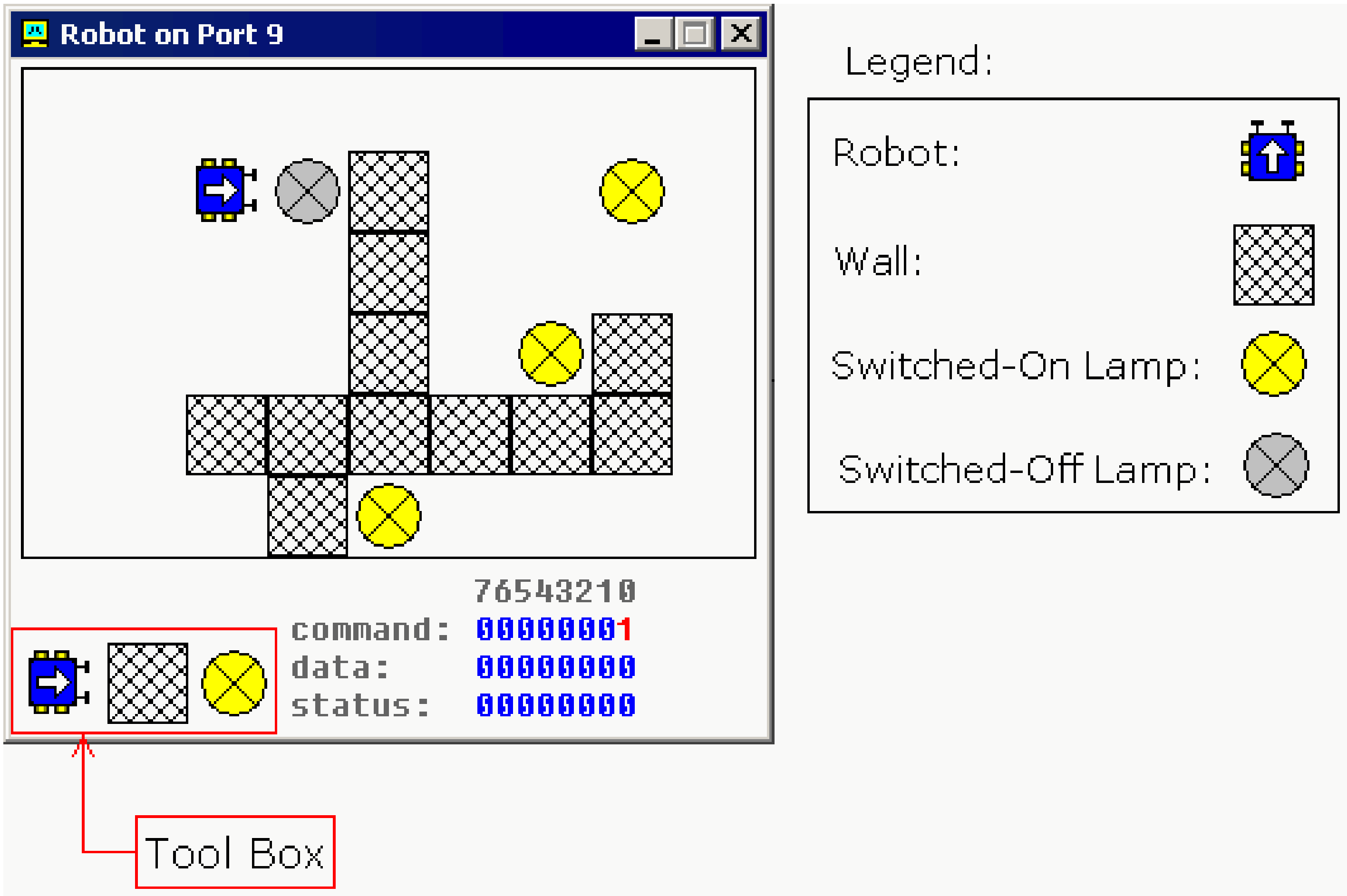
Half step is equal to **11.25** degrees.
Full step is equal to **22.5** degrees.

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



```
          76543210
command:  00000001
data:     00000000
status:   00000000
```

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 >>>**

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

Record Macro	Control + Shift + R
Set Repeat Count	Control + R

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]).

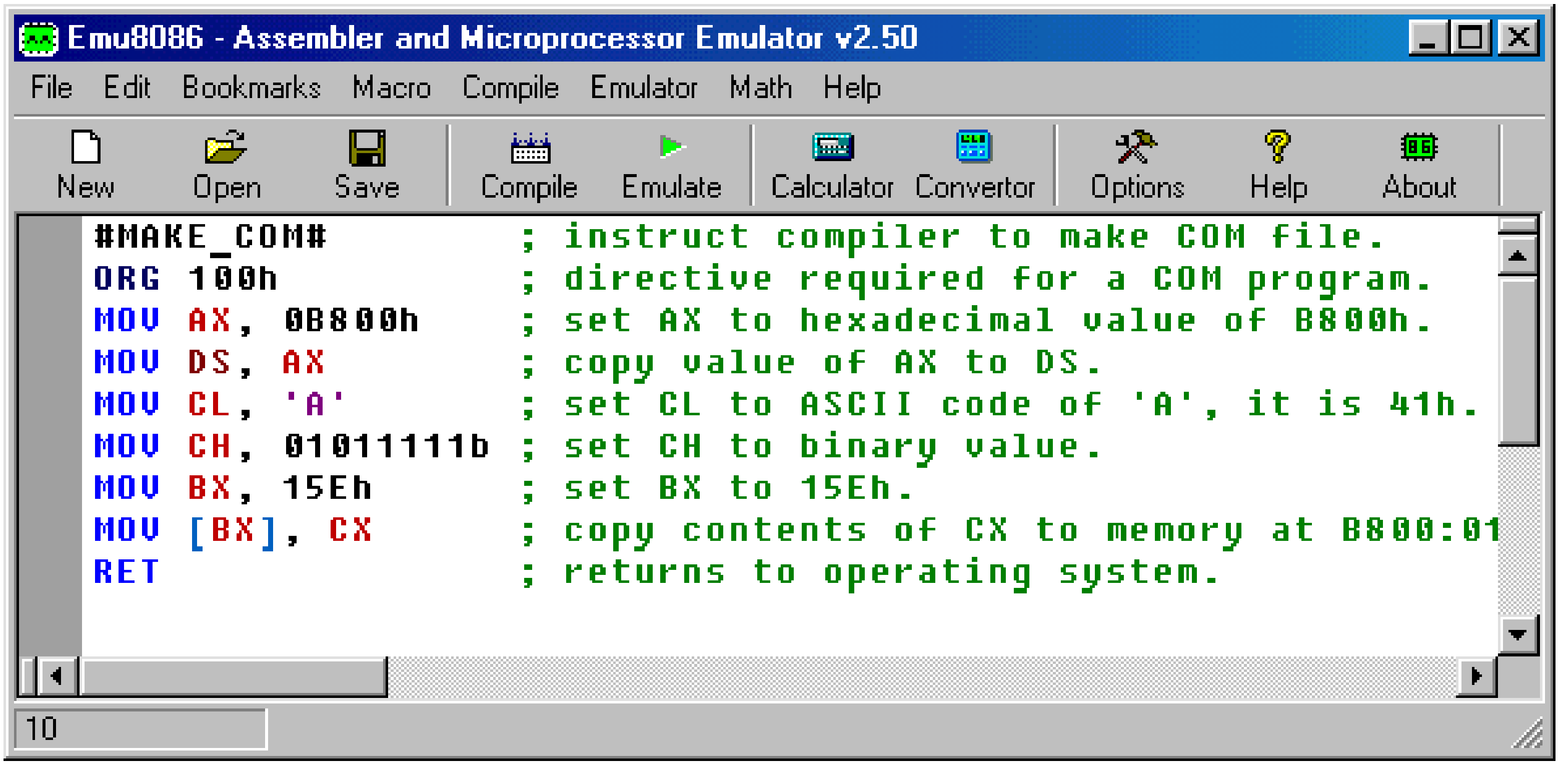
Logical OR:
Subexpressions may be ORed together
with the | pipe symbol.

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).

Compiling Assembly Code



```
#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 B800:01
RET                     ; returns to operating system.
```

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 find any of these directives it will ask you for *output type* before creating the 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.
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 defined Data, Stack, and Code segments.
  Entry point (where execution starts) is defined by a programmer.
  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 emulator is not able to find "**MY.BINF**" file, current register values are used and "**MY.BIN**" file is loaded at current **CS:IP**.
  Execution starts from values in **CS:IP**.
  This file type is unique to *Emu8086* emulator.

  ".BINF file is created automatically by compiler if it finds **#MAKE_BIN#** directive.
  WARNING! any existing ".binf" file is overwritten!

```
#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#
```

Values must be in HEX!

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.

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).
  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 "**.boot**" file and load it in emulator (see "**micro-os_loader.asm**" and "**micro-os_kernel.asm**" in "Samples" for more info).

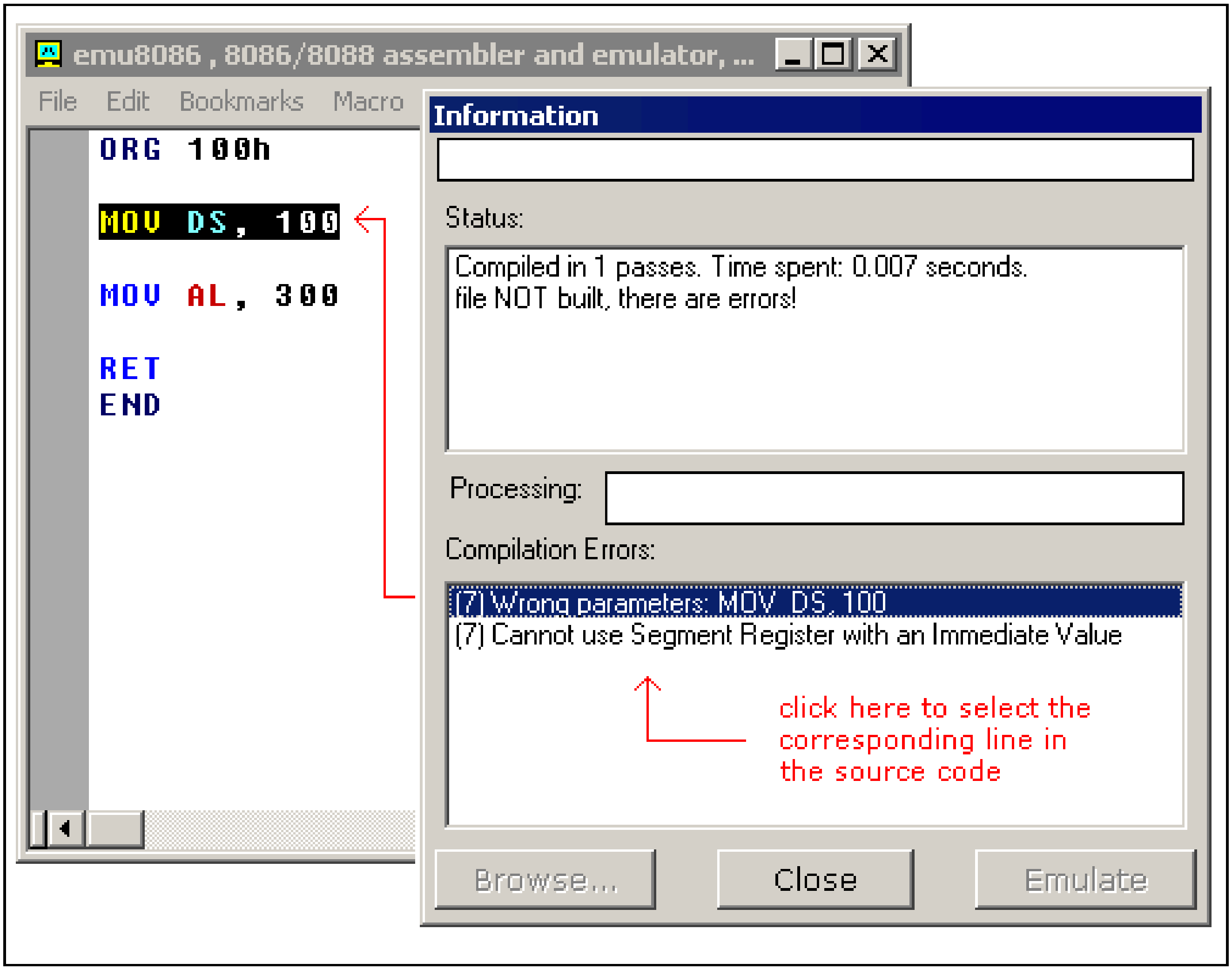
  Then select **[Virtual Drive] -> [Boot from Floppy]** menu to boot emulator from a virtual floppy.

  Then, if you are curious, you may write the virtual floppy to real floppy and boot your computer from it, I recommend using "RawWrite for Windows" from: http://uranus.it.swin.edu.au/~jn/linux/rawwrite.htm
  (note that "**micro-os_loader.asm**" is not using MS-DOS compatible boot sector, so it's better to use and empty floppy, although it should be IBM (MS-DOS) formatted).
  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** 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

Compiler reports about errors in a separate information window:



```
ORG 100h

MOV DS, 100

MOV AL, 300

RET
END
```

Status:

Compiled in 1 passes. Time spent: 0.007 seconds.
file NOT built, there are errors!

Processing:

Compilation Errors:

(7) Wrong parameters: MOV DS, 100
(7) Cannot use Segment Register with an Immediate Value

click here to select the corresponding line in the source code

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

Compiler makes several passes before generating the correct machine code, if it finds an error and does not complete the required number of passes it may show incorrect error messages. For example:

```
#make_COM#
ORG 100h

MOV AX, 0
MOV CX, 5
m1: INC AX
LOOP m1              ; not a real error!

MOV AL, 0FFFFh       ; error is here.

RET
```

```
List of generated errors:
(7) Condition Jump out of range!: LOOP m1
(9) Wrong parameters: MOV AL, 0FFFFh
(9) Operands do not match: Second operand is over 8 bits!
```

First error message (7) is incorrect, compiler did not finish calculating the offsets for labels, so it presumes that the offset of **m1** label is **0000**, that address is out of the range because we start at offset **100h**.

Make correction to this line: **MOV AL, 0FFFFh** (AL cannot hold **0FFFFh** value). This fixes both errors! For example:

```
#make_COM#
ORG 100h

MOV AX, 0
MOV CX, 5
m1: INC AX
LOOP m1              ; same code no error!

MOV AL, 0FFh         ; fixed!

RET
```

When saving a compiled file, compiler also saves 2 other files that are used for

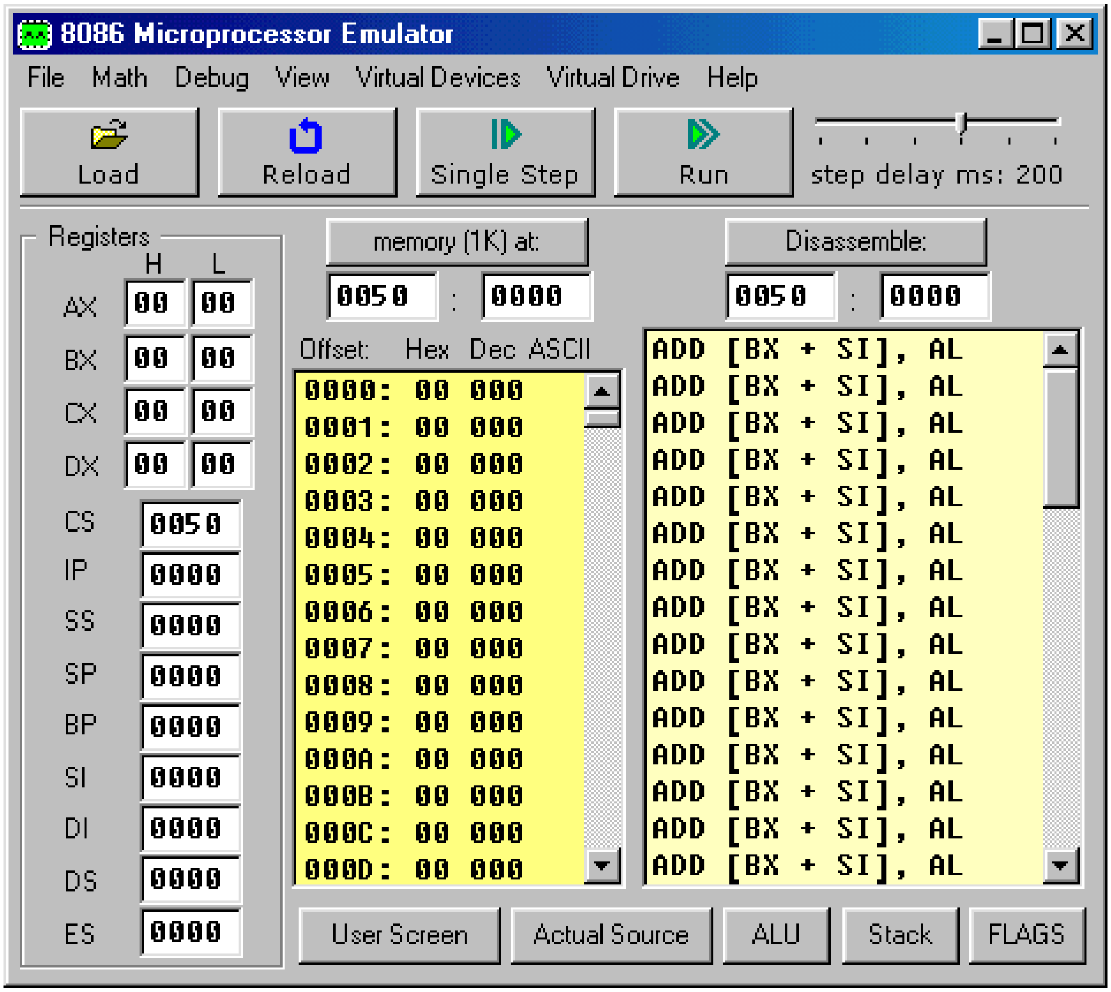
Emulator to show actual source when you run it, and select corresponding lines.

- ***.~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 text file, so you may view it in any text editor.
- ***.binf** - this 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).

Using Emulator

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

Click this button to load an executable file



	H	L
AX	00	00
BX	00	00
CX	00	00
DX	00	00

CS	0050
IP	0000
SS	0000
SP	0000
BP	0000
SI	0000
DI	0000
DS	0000
ES	0000

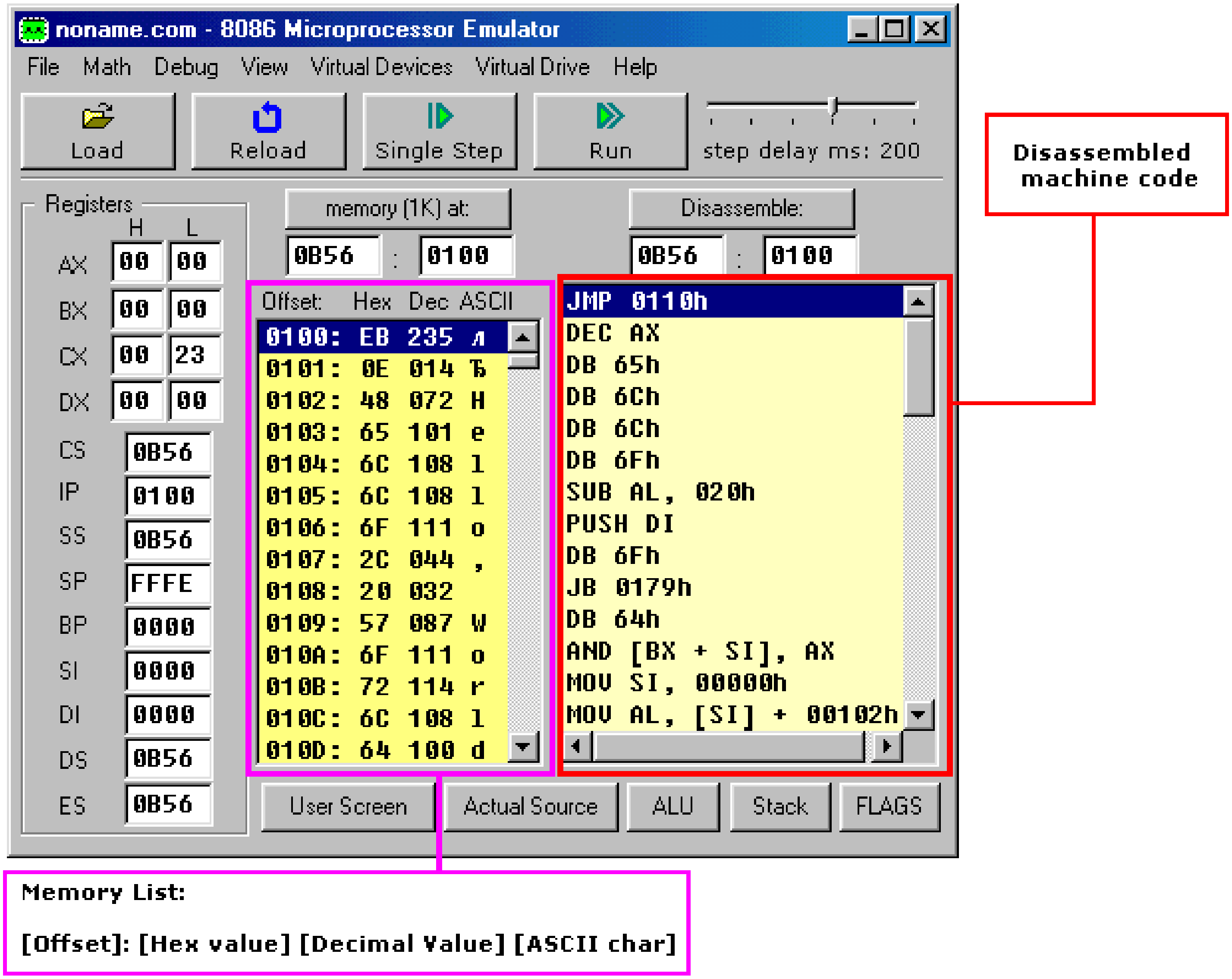
memory (1K) at: 0050 : 0000

```
Offset:  Hex  Dec ASCII
0000:  00 000
0001:  00 000
0002:  00 000
0003:  00 000
0004:  00 000
0005:  00 000
0006:  00 000
0007:  00 000
0008:  00 000
0009:  00 000
000A:  00 000
000B:  00 000
000C:  00 000
000D:  00 000
```

Disassemble: 0050 : 0000

```
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
ADD [BX + SI], AL
```

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



```
Registers
     H   L
AX   00  00
BX   00  00
CX   00  23
DX   00  00
CS   0B56
IP   0100
SS   0B56
SP   FFFE
BP   0000
SI   0000
DI   0000
DS   0B56
ES   0B56

memory (1K) at: 0B56 : 0100
Disassemble:    0B56 : 0100
```

```
Offset:  Hex Dec ASCII
0100: EB 235 л
0101: 0E 014 Ђ
0102: 48 072 H
0103: 65 101 e
0104: 6C 108 l
0105: 6C 108 l
0106: 6F 111 o
0107: 2C 044 ,
0108: 20 032
0109: 57 087 W
010A: 6F 111 o
010B: 72 114 r
010C: 6C 108 l
010D: 64 100 d
```

```
JMP 0110h
DEC AX
DB 65h
DB 6Ch
DB 6Ch
DB 6Fh
SUB AL, 020h
PUSH DI
DB 6Fh
JB 0179h
DB 64h
AND [BX + SI], AX
MOV SI, 00000h
MOV AL, [SI] + 00102h
```

Disassembled machine code

Memory List:

[Offset]: [Hex value] [Decimal Value] [ASCII char]

[**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.

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.

Ever wanted to write your own operating system?

> You can write a boot sector of a virtual floppy via menu in emulator:
> **[Virtual Drive] -> [Write 512 bytes at 7C00 to Boot Sector]**
> First you should compile a "**.boot**" file and load it in emulator (see "**micro-os_loader.asm**" and "**micro-os_kernel.asm**" in "Samples" for more info).
>
> Then select **[Virtual Drive] -> [Boot from Floppy]** menu to boot emulator from a virtual floppy.
>
> Then, if you are curious, you may write the virtual floppy to real floppy and boot your computer from it, I recommend using "RawWrite for Windows" from: **http://uranus.it.swin.edu.au/~jn/linux/rawwrite.htm** (note that "**micro-os_loader.asm**" is not using MS-DOS compatible boot sector, so it's better to use and empty floppy, although it should be IBM (MS-DOS) formatted).
>
> 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** file should be less then 512 bytes (limited by the size of a disk sector).

Interrupts currently supported by emulator

Quick reference:

INT 10h/00h
INT 10h/01h
INT 10h/02h
INT 10h/03h
INT 10h/05h
INT 10h/06h
INT 10h/07h

INT 10h/08h
INT 10h/09h
INT 10h/0Ah
INT 10h/0Eh
INT 10h/13h
INT 10h/1003h
INT 11h

INT 12h
INT 13h/00h
INT 13h/02h
INT 13h/03h
INT 15h/86h
INT 16h/00h
INT 16h/01h

INT 19h
INT 1Ah/00h
INT 21h

A list of supported interrupts with descriptions:

INT 10h / **AH = 00h** - set video mode.

> *input:*
> **AL** = desired video mode.
>
> These video modes are supported:
>
> **00h** - Text mode 40x25, 16 colors, 8 pages.
>
> **03h** - Text mode 80x25, 16 colors, 8 pages.

INT 10h / **AH = 01h** - set text-mode cursor shape.

> *input:*
> **CH** = cursor start line (bits 0-4) and options (bits 5-7).
> **CL** = bottom cursor line (bits 0-4).
>
> When bits 6-5 of CH are set to **00**, the cursor is visible, to hide a cursor set these bits to **01** (this CH value will hide a cursor: 28h - 00101000b). Bit 7 should always be zero.

INT 10h / **AH = 02h** - set cursor position.

input:
DH = row.
DL = column.
BH = page number (0..7).

INT 10h / **AH = 03h** - get cursor position and size.

input:
BH = page number.
return:
DH = row.
DL = column.
CH = cursor start line.
CL = cursor bottom line.

INT 10h / **AH = 05h** - select active video page.

input:
AL = new page number (0..7).
the activated page is displayed.

INT 10h / **AH = 06h** - scroll up window.
INT 10h / **AH = 07h** - scroll down window.

input:
AL = number of lines by which to scroll (00h = clear entire window).
BH = attribute used to write blank lines at bottom of window.
CH, CL = row, column of window's upper left corner.
DH, DL = row, column of window's lower right corner.

INT 10h / AH = 08h - read character and attribute at cursor position.

input:
BH = page number.
return:
AH = attribute.
AL = character.

INT 10h / AH = 09h - write character and attribute at cursor position.

input:
AL = character to display.
BH = page number.
BL = attribute.
CX = number of times to write character.

INT 10h / AH = 0Ah - write character only at cursor position.

input:
AL = character to display.
BH = page number.
CX = number of times to write character.

INT 10h / AH = 0Eh - teletype output.

input:
AL = character to write.

This functions displays a character on the screen, advancing the cursor and scrolling the screen as necessary. The printing is always done to current active page.

INT 10h / AH = 13h - write string.

input:
AL = write mode:
bit 0: update cursor after writing;
bit 1: string contains attributes.
BH = page number.
BL = attribute if string contains only characters (bit 1 of AL is zero).
CX = number of characters in string (attributes are not counted).
DL,DH = column, row at which to start writing.
ES:BP points to string to be printed.

INT 10h / AX = 1003h - toggle intensity/blinking.

input:
BL = write mode:
0: enable intensive colors.
1: enable blinking (not supported by emulator!).
BH = 0 (to avoid problems on some adapters).

Bit color table:
Character attribute is 8 bit value, low 4 bits set foreground color, high 4 bits set background color. Background blinking not supported.

HEX	BIN	COLOR
0	0000	black
1	0001	blue
2	0010	green
3	0011	cyan
4	0100	red
5	0101	magenta
6	0110	brown
7	0111	light gray
8	1000	dark gray
9	1001	light blue
A	1010	light green
B	1011	light cyan

C	1100	light red
D	1101	light magenta
E	1110	yellow
F	1111	white

INT 11h - get BIOS equipment list.

return:
AX = BIOS equipment list word, actually this call returns the contents of the word at 0040h:0010h.

Currently this function can be used to determine the number of installed number of floppy disk drives.

Bit fields for BIOS-detected installed hardware:
Bit(s) Description
15-14 number of parallel devices.
13 not supported.
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 not supported.
2 not supported.
1 math coprocessor installed.
0 set when booted from floppy (always set by emulator).

INT 12h - get memory size.

return:
AX = kilobytes of contiguous memory starting at absolute address 00000h, this call returns the contents of the word at 0040h:0013h.

Floppy drives are emulated using *FLOPPY_0(..3)* **files.**

INT 13h / **AH = 00h -** reset disk system, (currently this call doesn't do anything).

INT 13h / **AH = 02h -** read disk sectors into memory.
INT 13h / **AH = 03h -** write disk sectors.

input:

AL = number of sectors to read/write (must be nonzero)
CH = cylinder number (0..79).
CL = sector number (1..18).
DH = head number (0..1).
DL = drive number (0..3 , depends on quantity of FLOPPY_? files).
ES:BX points to data buffer.

return:

CF set on error.
CF clear if successful.
AH = status (0 - if successful).
AL = number of sectors transferred.

Note: each sector has **512** bytes.

INT 15h / **AH = 86h -** BIOS wait function.

input:

CX:DX = interval in microseconds

return:

CF clear if successful (wait interval elapsed),
CF set on error or when wait function is already in progress.

Note:

the resolution of the wait period is 977 microseconds on many systems, Emu8086 uses 1000 microseconds period.

INT 16h / **AH = 00h** - get keystroke from keyboard (no echo).

return:

AH = BIOS scan code.
AL = ASCII character.
(if a keystroke is present, it is removed from the keyboard buffer).

INT 16h / **AH = 01h** - check for keystroke in keyboard buffer.

return:

ZF = 1 if keystroke is not available.
ZF = 0 if keystroke available.
AH = BIOS scan code.
AL = ASCII character.

(if a keystroke is present, it is not removed from the keyboard buffer).

INT 19h - system reboot.

Usually, the BIOS will try to read sector 1, head 0, track 0 from drive A: to 0000h:7C00h. Emulator just stops the execution, to boot from floppy drive select from the menu: **'Virtual Drive' -> 'Boot from Floppy'**

INT 1Ah / **AH = 00h** - get system time.

return:

CX:DX = number of clock ticks since midnight.
AL = midnight counter, advanced each time midnight passes.

Notes:
There are approximately **18.20648** clock ticks per second,
and **1800B0h** per 24 hours.
AL is not set by emulator yet!

MS-DOS can not be loaded completely in emulator yet, so I made an emulation for some basic DOS interrupts also:

INT 20h - exit to operating system.

INT 21h / **AH=09h** - output of a string at DS:DX.

INT 21h / **AH=0Ah -** input of a string to DS:DX, fist byte is buffer size, second byte is number of chars actually read.

INT 21h / **AH=4Ch** - exit to operating system.

INT 21h / **AH=01h -** read character from standard input, with echo, result is stored in AL.

INT 21h / AH=02h - write character to standard output, DL = character to write, after execution AL = DL.

Global Memory Table

8086 CPU can access up to **1 MB** of random access memory (RAM), it is limited by segment/offset construction. Since segment registers (**CS, SS, ES, DS**) can hold maximum value of **0FFFFh** and offset registers (**IP, BX, SI, DI, BP, SP**) can also hold maximum value of **0FFFFh**, the largest logical memory location that we can access is **FFFF:FFFF** or physical address: 0FFFFh * 10h + 0FFFFh = 10FFEFh = 65535 * 16 + 65535 = 1,114,095 bytes
Modern processors have a larger registers so they have much larger memory area that can be accessed, but the idea is still the same.

Memory Table of Emulator (and typical IBM PC):

Physical address of memory area in HEX	**Short Description**
00000 - 00400	Interrupt vectors. Emulator loads "**INT_VECT**" file at the physical address 00000h.
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 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. Not used by emulator!
B1000 - B8000	Reserved. Not used by emulator!

B8000 - C0000	32 KB video memory for Color Graphics Adapter (CGA). 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 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. Emulator loads "**BIOS_ROM**" file at the physical address 0F4000h. Interrupt table points to this memory area to get emulation of interrupt functions.

Interrupt Vector (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.
19	19x4 = 64	FFFF:0000 - Reboot.

1A	1Ax4 = 68	F400:0160 - Time functions.
1E	1Ex4 = 78	F400:AFC7 - Vector of Diskette Controller Params.
20	20x4 = 80	F400:0150 - DOS function: terminate program.
21	21x4 = 84	F400:0200 - DOS functions.
all others	??x4 = ??	F400:0100 - The default interupt catcher. Prints out "Interupt not supported yet" message.

A call to BIOS sub-system is disassembled by "BIOS DI" (it doesn't use DI register in any way, it's just because of the way the encoding is done: we are using "FF /7" for such encoding, "FFFFCD10" is used to make emulator to emulate interrupt number 10h).

F400:0100 has this code FFFFCDFF (decoded as INT 255, and error message is generated).

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

Address (hex)	**Size**	**Description**
		BIOS equipment list. Bit fields for BIOS-detected installed hardware: Bit(s) Description 15-14 number of parallel devices. 13 not supported. 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;

0040h:0010	WORD	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 not supported. 2 not supported. 1 math coprocessor installed. 0 set when booted from floppy (always set by emulator). This word is also returned in **AX** by **INT 11h**. Default value: **0021h** or **0000 0000 0010 0001b**
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).

See also: **Custom Memory Map**

Custom Memory Map

You can define your own memory map (different from IBM-PC). It is required to create "**CUSTOM_MEMORY_MAP.inf**" file in the same folder where **Emu8086.exe** is located. Using the following format add information into that configuration file:

address - filename
...

For example:

```
0000:0000 - System.bin
F000:0000 - Rom.bin
12AC - Data.dat
```

Address can be both physical (without ":") or logical, value must be in hexadecimal form. Emulator will look for the file name after the "-" and load it into the memory at the specified address.

Emulator will not update **System information area (memory from 00400h to 00500h)** if your configuration file has "**NO_SYS_INFO**" directive (on a separate line). For example:

```
NO_SYS_INFO
0000:0000 - System.bin
F000:0000 - Rom.bin
12AC - Data.dat
```

Emulator will allow you to load "**.bin**" files to any memory address (be careful not to load them over your custom system/ data area).

Warning! standard interrupts will not work when you change the memory map, unless you provide your own replacement for them. To disable changes just delete or rename "**CUSTOM_MEMORY_MAP.inf**" file, and restart the program.

See also: **Global Memory Table**

MASM / TASM compatibility

Syntax of *Emu8086* is fully compatible with all major assemblers including *MASM* and *TASM*; though some directives are unique to *Emu8086*. If required to compile using any other assembler you may need to comment out these directives, and any other directives that start with a '**#**' sign:

```
#MAKE_COM#
#MAKE_EXE#
#MAKE_BIN#
#MAKE_BOOT#
```

Emu8086 does not support the **ASSUME** directive, actually most programmers agree that this directive just causes some mess in your code. Manual attachment of **CS:**, **DS:**, **ES:** or **SS:** segment prefixes is preferred, and required by *Emu8086* when data is in segment other then **DS**. For example:

```
MOV AX, [BX]       ; same as MOV AX, DS:[BX]
MOV AX, ES:[BX]
```

Emu8086 does not require to define segment when you compile a **COM** file, though *MASM* and *TASM* may require this, for example:

```
CSEG    SEGMENT     ; code segment starts here.

; #MAKE_COM#        ; uncomment for Emu8086.

ORG 100h

start:  MOV AL, 5   ; some sample code...
        MOV BL, 2
        XOR AL, BL
        XOR BL, AL
        XOR AL, BL

        RET

CSEG    ENDS        ; code segment ends here.

END     start       ; stop compiler, and set entry point.
```

Entry point for **COM** file should always be at **0100h** (first instruction after **ORG 100h** directive), though in *MASM* and *TASM* you may need to manually set an entry point using **END** directive. *Emu8086* works just fine, with or without it.

In order to test the above code, save it into **test.asm** file (or any other) and run these commands from command prompt:

For MASM 6.0:

```
MASM test.asm
LINK test.obj, test.com,,, /TINY
```

For TASM 4.1:

```
TASM test.asm
TLINK test.obj /t
```

We should get **test.com** file (11 bytes), right click it and select **Send To** and **emu8086**. You can see that the disassembled code doesn't contain any directives and it is identical to code that *Emu8086* produces even without all those tricky directives.

A template used by *Emu8086* to create **EXE** files is fully compatible with *MASM* and *TASM*, just comment out **#MAKE_EXE#** directive to avoid **Unknown character** error at line **11**.

EXE files produced by *MASM* are identical to those produced by *emu8086*. *TASM* does not calculate the checksum, and has slightly different EXE file structure, but it produces quite the same machine code.

Note: there are several ways to encode the same machine instructions for the 8086 CPU, so generated machine code may vary when compiled on different compilers.

Emu8086 assembler supports shorter versions of **BYTE PTR** and **WORD PTR**, these are: **B.** and **W.**

For *MASM* and *TASM* you have to replace **B.** and **W.** with **BYTE PTR** and **WORD PTR** accordingly.

For example:

```
LEA BX, var1
MOV WORD PTR [BX], 1234h ; works everywhere.
MOV w.[BX], 1234h        ; same instruction, but works in Emu8086 only.
HLT

var1  DB  0
var2  DB  0
```

I/O ports

Emulator does not reproduce any original I/O ports of IBM PC, instead it has virtual devices that can be accessed by IN/OUT instructions.

Custom I/O Devices

"*Emu8086*" supports additional devices that can be created by 3rd party vendors. Device can be written in any language, such as: Visual Basic, VC++, Delphi. For more information and sample source code look inside **DEVICES** folder.

Reserved input / output addresses for custom devices are: from 00000Fh to 0FFFFh (15 to 65535). Port 100 corresponds to byte 100 in "**EmuPort.io**" file, port 101 to the byte 101, etc...
(we count from zero).
"**EmuPort.io**" file is located in Windows "**Temp**" directory (can be accessed by *GetTempPath()* function of the API).

I'll be glad to include devices made by you in the next release of "*Emu8086*". If you decide to share your device with other developers around the world - please send us the source code of the device!

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.

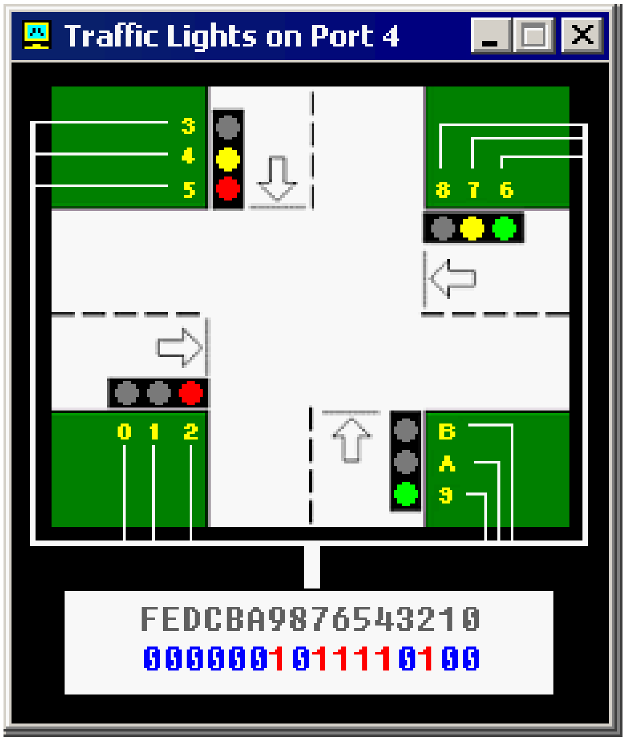
  You can set the state of each lamp by setting its bit:

  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
```



```
FEDCBA9876543210
0000001011110100
```

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 Samples.

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.

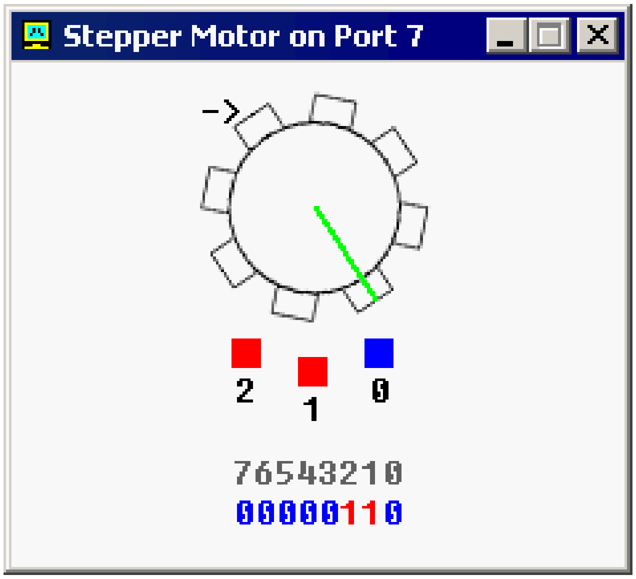
Stepper Motor is electric motor that can be precisely controlled by signals from a computer.

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
```

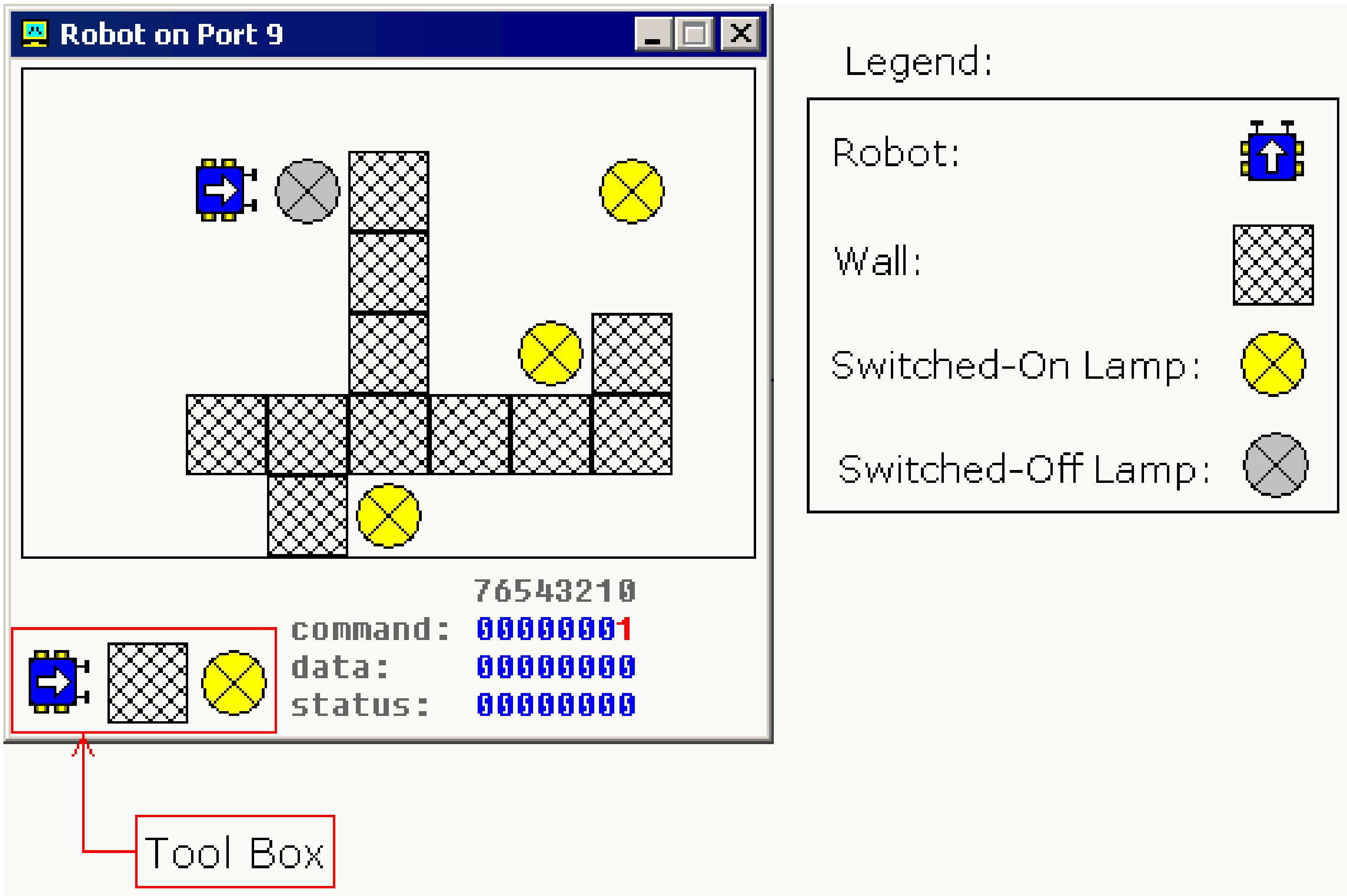
```
MOV AL, 011b ; half step 1.
OUT 7, AL

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 Samples.

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

```
IN AL, 7
```

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



Robot on Port 9

```
          76543210
command:  00000001
data:     00000000
status:   00000000
```

The Robot is controlled by sending data to I/O Port 9.

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.

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

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 Samples.

Creating Custom *Robot World* Map

You can create any map for the robot using the tool box.

If you choose **robot** and place it 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 choose **lamp** and place it over **switched-on lamp** the **lamp** will become **switched-off**, if **lamp** is already **switched-off** it will be **deleted**.

Placing **wall** over existing **wall** will **delete** it.

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.

The map is automatically saved on exit.

Right-click the map to get a popup menu that allows to Switch-Off/On all Lamps.

Documentation for Emu8086

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

```
; This sample shows how
; to use CMPSW instruction
; to compare strings.

#make_COM#

; COM file is loaded at 100h
; prefix:
        ORG     100h

; set forward direction:
        CLD

; load source into DS:SI,
; load target into ES:DI:
        MOV     AX, CS
        MOV     DS, AX
        MOV     ES, AX
        LEA     si, dat1
        LEA     di, dat2

; set counter to data length:
        MOV     CX, 4

; compare until equal:
        REPE    CMPSW
        JNZ     not_equal

; "Yes" - equal!
        MOV     AL, 'Y'
        MOV     AH, 0Eh
        INT     10h

        JMP     exit_here

not_equal:

; "No" - not equal!
        MOV     AL, 'N'
        MOV     AH, 0Eh
        INT     10h
```

```
exit_here:

        RET

; data:
dat1 DW 1234h, 5678h, 9012h, 3456h
dat2 DW 1234h, 5678h, 9012h, 3456h

END
```

```
; This sample shows how
; to use CMPSB instruction
; to compare strings.

#make_COM#

; COM file is loaded at 100h
; prefix:
        ORG     100h

; set forward direction:
        CLD

; load source into DS:SI,
; load target into ES:DI:
        MOV     AX, CS
        MOV     DS, AX
        MOV     ES, AX
        LEA     si, str1
        LEA     di, str2

; set counter to string length:
        MOV     CX, 11

; compare until equal:
        REPE    CMPSB
        JNZ     not_equal

; "Yes" - equal!
        MOV     AL, 'Y'
        MOV     AH, 0Eh
        INT     10h

        JMP     exit_here

not_equal:

; "No" - not equal!
        MOV     AL, 'N'
        MOV     AH, 0Eh
        INT     10h
```

```
exit_here:

        RET

; data:
str1 db 'Test string'
str2 db 'Test string'

END
```