emu8086

Where to start?

1. Click **code examples** and select **Hello, world**. A code example with many comments should open. All comments are green and they take up about **90%** of all text, so don't be scared by this tiny *"Hello Word"* code. The compiled executable is only about **100** bytes long, because it uses no interrupts and has only one loop for color *highlighting* the text. All other code is straight-forward and writes directly to video memory.

2. To run this example in the emulator, click **emulate** (or press **F5**). The program then attmepts to assemble and save the executable to **c:\emu8086\MyBuild**. If the assembler succeeds in creating the file, the emulator will also automatically load it into memory.

3. You can then click **single step** (or press **F8**) to step through the code one instruction at a time, observing changes in registers and the emulator screen. You can also click **step back** (or press **F6**) to see what happens when reversing those changes.

4. There are many ways to print *"Hello,World"* in assembly language, and this certainly isn't the shortest way. If you click **examples** and browse **c:\emu8086\examples**, you'll find **HelloWorld.asm** which assembles into only a **30**-byte executable. Unlike the previous example which carries out each step by itself, this one is much smaller because it uses a built-in interrupt function of the operating system to write to the display.

The integrated 8086 assembler can generate console programs that can be executed on any computer that runs x86 machine code (Intel/AMD architecture)

The architecture of the 8086 Intel microprocessor is called "Von Neumann architecture" after the mathematician who conceived of the design.

NOTE: A CPU can interpret the contents of memory as either

instructions or data; there's no difference in the individual bytes of memory, only the way in which they're arranged. Because of this, it's even possible for programs to re-write their own instructions, then execute the instructions they've changed.

Source Code Editor

Using the Mouse

Editor supports the following mouse actions:

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

	the current splitter position
Double click splitter bar	Split the window in half into multiple views or unsplit the window if already split

---

Editor Hot Keys:

Command Keystroke
==========
Toggle Bookmark Control + F2
Next Bookmark F2
Prev Bookmark Shift + F2

Copy Control + C, Control + Insert
Cut Control + X, Shift + Delete, Control + Alt + W
Cut Line Control + Y
Cut Sentence Control + Alt + K
Paste Control + V, Shift + Insert

Undo Control + Z, Alt + Backspace

Document End Control + End
Document End Extend Control + Shift + End
Document Start Control + Home
Document Start Extend Control + Shift + Home

Find Control + F, Alt + F3
Find Next F3
Find Next Word Control + F3
Find Prev Shift + F3
Find Prev Word Control + Shift + F3
Find and Replace Control + H, Control + Alt + F3
Go To Line Control + G
Go To Match Brace Control +]

Select All Control + A
Select Line Control + Alt + F8
Select Swap Anchor Control + Shift + X

Insert New Line Above Control + Shift + N

Indent Selection Tab
Outdent Selection Shift + Tab

Tabify Selection Control + Shift + T

Untabify Selection Control + Shift + Space

Lowercase Selection Control + L
Uppercase Selection Control + U, Control + Shift + U

Left Word Control + Left
Right Word Control + Right
Left Sentence Control + Alt + Left
Right Sentence Control + Alt + Right

Toggle Overtype Insert
Display Whitespace Control + Alt + T

Scroll Window Up Control + Down
Scroll Window Down Control + Up
Scroll Window Left Control + PageUp
Scroll Window Right Control + PageDown

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

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

Comment Block Ctrl + Q
Uncomment Block Ctrl + W

---

regular expression syntax rules for search and replace

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

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

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

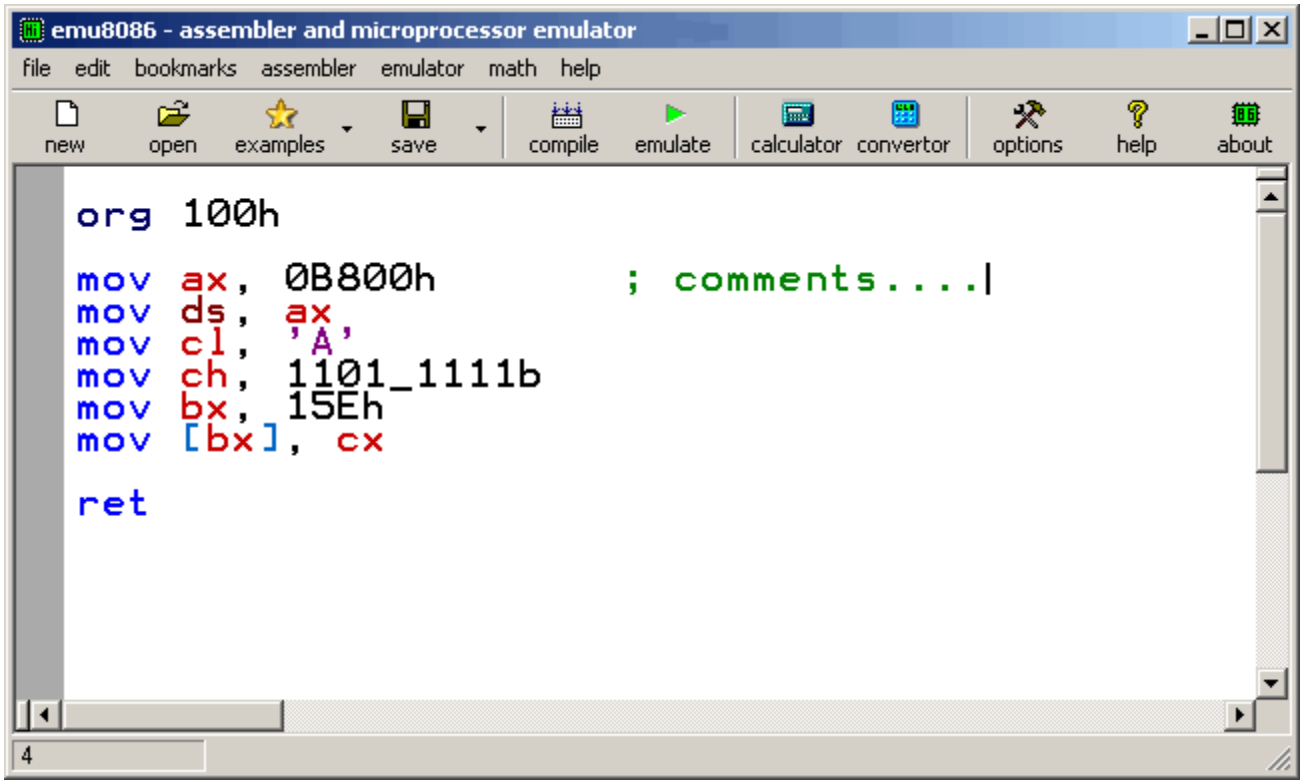
logical OR:
subexpressions may be ORed together
with the | pipe symbol.

rarenthesized 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 the assembly code



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

---

the output file type directives:

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

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

there is virtually no difference between how .com and .bin are assembled because these files are raw binary files, but .exe file has a

special header in the beginning of the file that is used by the operating system to determine some properties of the executable file.

---

description of output file types:

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

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

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

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

these directives can be inserted into any part of the source code to preset registers or memory before starting the program's execution:

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

- all values must be in hexadecimal.

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

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

  "#mem=..." directive can be used to write values to memory before program starts
  #MEM=nnnn,[bytestring]-nnnn:nnnn,[bytestring]#
  for example:

#MEM=1000,01ABCDEF0122-0200,1233#
all values are in hex, nnnn - for physical address, or (nnnn:nnnn) for logical address.
- separates the entries. spaces are allowed inside.

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

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

- ---
-

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

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

-

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

NOTE: existing .binf file is automatically overwritten on re-compile.

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

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

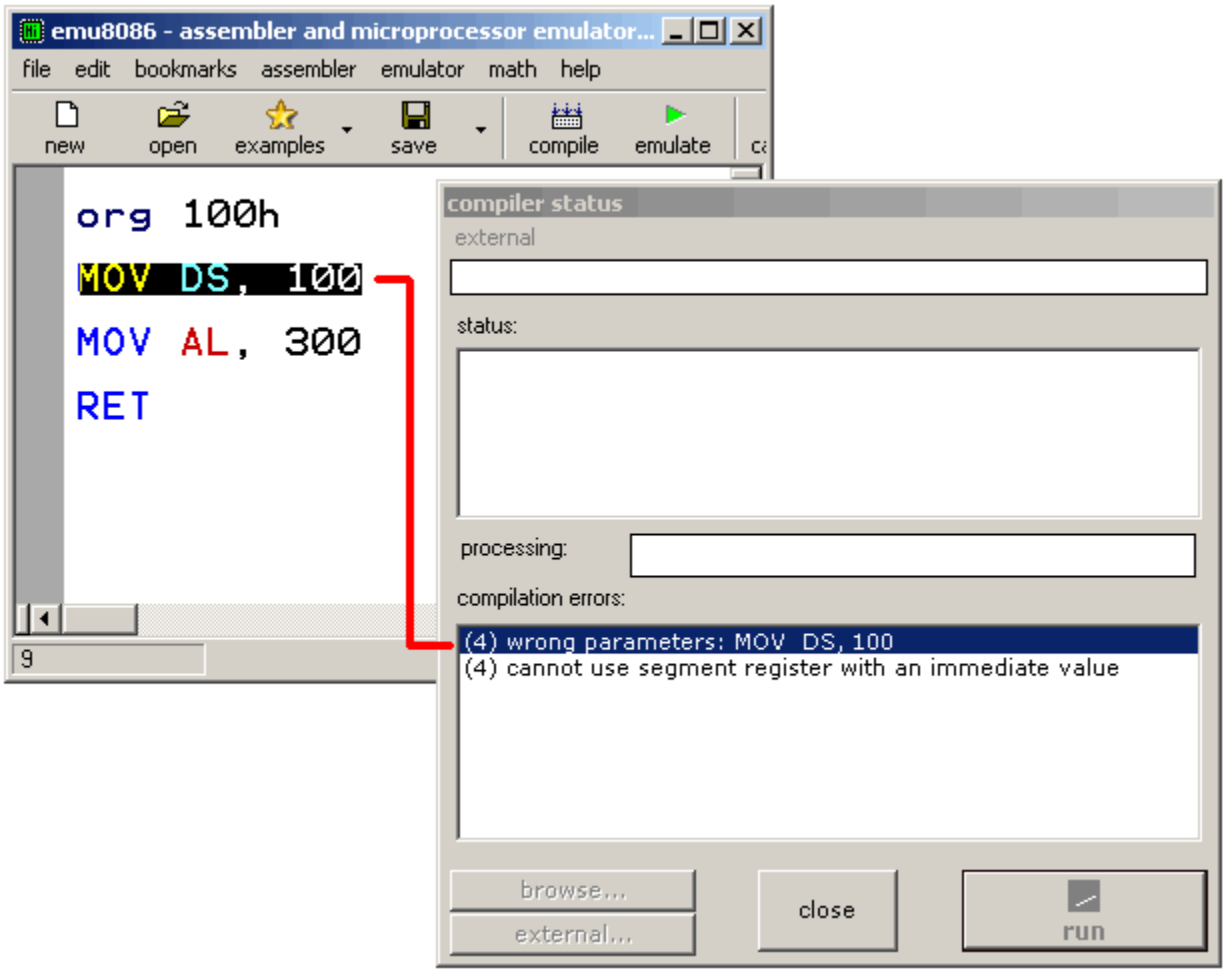
  then select **[virtual drive] -> [boot from floppy]** menu to boot emulator from a virtual floppy.

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

---

error processing

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



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

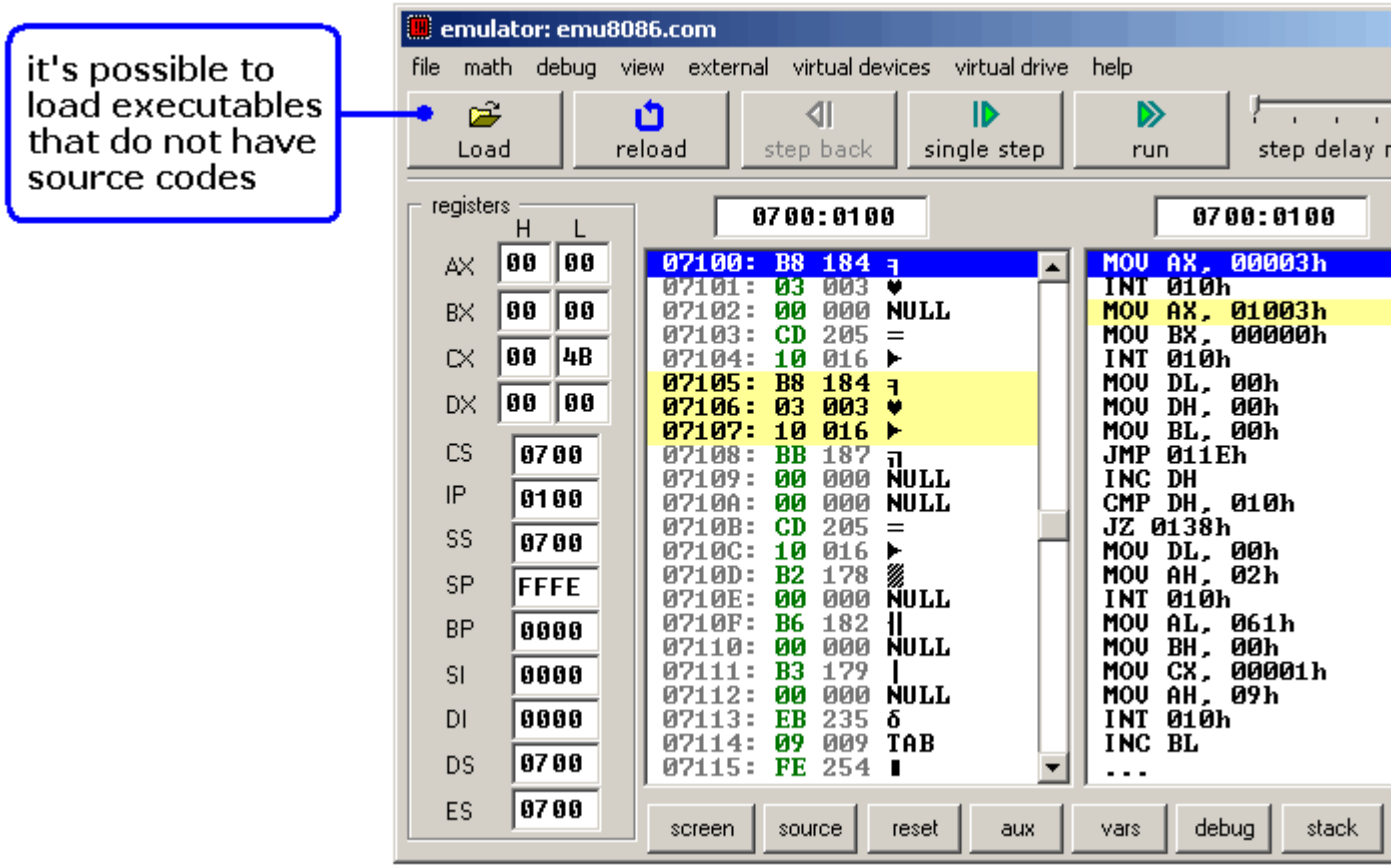
MOV AL, 300 - is illegal instruction because **AL** register has only 8 bits, and thus maximum value for it is 255 (or 11111111b), and the minimum is -128.

When saving an assembled file, compiler also saves 2 other files that are later used by the emulator to show original source code when you run the binary executable, and select corresponding lines. Very often the original code differs from the disabled code because there are no comments, no segment and no variable declarations. Compiler directives produce no binary code, but everything is converted to pure machine code. Sometimes a single original instruction is assembled into several machine code instructions, this is done mainly for the compatibility with original 8086 microprocessor (for example **ROL AL, 5** is assembled into five sequential **ROL AL, 1** instructions).

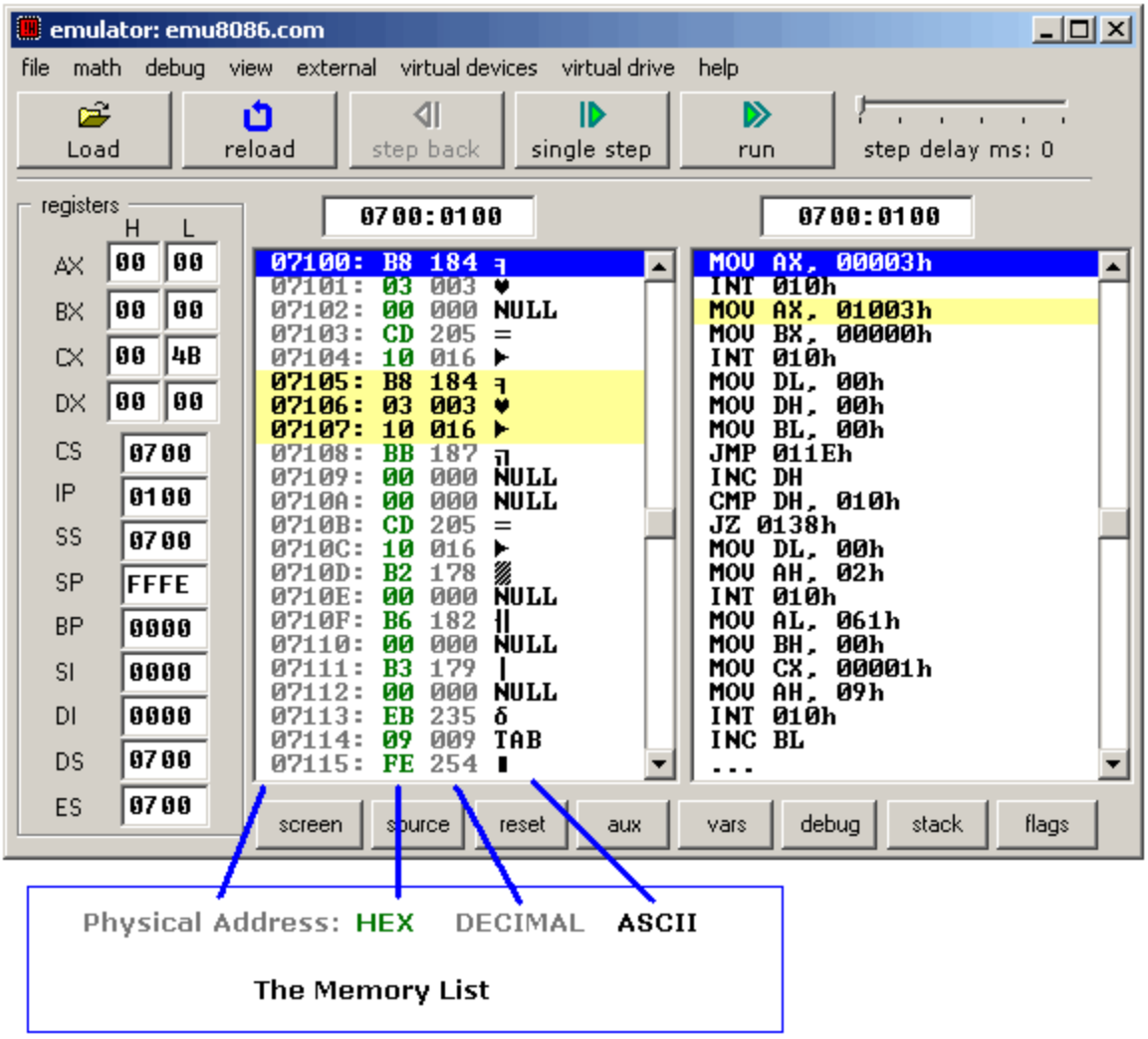
- ***.~asm** - this file contains the original source code that was used to make an executable file.
- ***.debug** - this file has information that enables the emulator select lines of original source code while running the machine code.
- ***.symbol** - symbol table, it contains information that enables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

Using the 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 the **Emulator** menu.



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



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

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

Double click on register text-boxes opens "**Extended Viewer**" window with value of that register converted to all possible forms. You can modify the value of the register directly in this window.

Double click on memory list item opens "**Extended Viewer**" with WORD value loaded from memory list at selected location. Less significant byte is at lower address: LOW BYTE is loaded from selected position and HIGH BYTE from next memory address. You can modify the value of the memory word directly in the "**Extended Viewer**"

window,

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

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

Virtual drives

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

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

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

```
Bits 7-6 of AX:
          00 single floppy disk.
          01 two floppy disks.
          10 three floppy disks.
          11 four floppy disks.
```

Emulator starts counting attached floppy drives from starting from the first, in case file **FLOPPY_1** does not exist it stops the check and ignores **FLOPPY_2** and **FLOPPY_3** files.

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

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

Global Memory Table

8086 CPU can access up to **1 MB** of random access memory (RAM). This is more than enough for any kind of computations (if used wisely).

---

memory table of the emulator (and typical ibm pc memory table):

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

F4000 - 10FFEF	ROM BIOS and extensions. the emulator loads **BIOS_ROM** file at the physical address 0F4000h. addresses of interrupt table points to this memory area to make emulation of the interrupt functions.

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

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

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

FFFFCD10 is used to make the emulator to execute interrupt number 10h.

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

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

address (hex)	size	description
0040h:0010	WORD	BIOS equipment list. bit fields for BIOS-detected installed hardware: bit(s) Description 15-14 number of parallel devices. 13 reserved. 12 game port installed. 11-9 number of serial devices. 8 reserved. 7-6 number of floppy disk drives (minus 1): 00 single floppy disk; 01 two floppy disks; 10 three floppy disks; 11 four floppy disks. 5-4 initial video mode: 00 EGA,VGA,PGA, or other with on-board video BIOS; 01 40x25 CGA color. 10 80x25 CGA color (emulator default). 11 80x25 mono text. 3 reserved. 2 PS/2 mouse is installed. 1 math coprocessor installed. 0 set when booted from floppy.
0040h:0013	WORD	kilobytes of contiguous memory starting at absolute address 00000h. this word is also returned in **AX** by **INT 12h**. this value is set to: **0280h** (640KB).
0040h:004A	WORD	number of columns on screen. default value: **0032h** (50 columns).

0040h:004E	WORD	current video page start address in video memory (after 0B800:0000). default value: **0000h**.
0040h:0050	8 WORDs	contains row and column position for the cursors on each of eight video pages. default value: **0000h** (for all 8 WORDs).
0040h:0062	BYTE	current video page number. default value: **00h** (first page).
0040h:0084	BYTE	rows on screen minus one. default value: **13h** (19+1=20 columns).

see also: **custom memory map**

Custom Memory Map

You can define your own memory map (different from IBM-PC). It is required to create this file: **c:\emu8086\custom_memory_map.inf** then you can use the following format to add settings 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 emu8086.

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 this assembler. 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_bin#
#make_boot#
#cs=...#
      etc...
```

---

emu8086 ignores the **ASSUME** directive. 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 ah, [bx]       ; read byte from DS:BX
mov ah, es:[bx]    ; read byte from ES:BX
```

---

emu8086 does not require to define segment when you compile segmentless **COM** file, however *MASM* and *TASM* may require this, for example:

```
name test

CSEG    SEGMENT     ; code segment starts here.

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**, however in *MASM* and *TASM* you may need to manually set an entry point using **END** directive even if there is no way to set it to some other location. emu8086 works just fine, with or without it; however error message is generated if entry point is set but it is not 100h (the starting offset for com executable). the entry point of com files is always the first byte.

if you compile this code with Microsoft Assembler or with Borland Turbo Assembler, you 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.

emu8086 has almost 100% compatibility with other similar 16 bit assemblers. the code that is assembled by emu8086 can easily be assembled with other assemblers such as TASM or MASM, however not every code that assembles by TASM or MASM can be assembled by emu8086.

---

a template used by emu8086 to create **EXE** files is fully compatible with *MASM* and *TASM*.

the majority of **EXE** files produced by *MASM* are identical to those produced by *emu8086*. However, it may not be exactly the same as TASM's executables because *TASM* does not calculate the checksum, and has slightly different EXE file structure, but in general 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 integrated assembler supports shorter versions of **byte ptr** and **word ptr**, these are: **b.** and **w.**

for *MASM* and *TASM* you have to replace **w.** 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 / shorter emu8086 syntax.
```

```
hlt

var1  db  0
var2  db  0
```

LABEL directive may not be supported by all assemblers, for example:

```
TEST1 LABEL BYTE
; ...
LEA DX,TEST1
```

the above code should be replaced with this alternative construction:

```
TEST1:
; ...
MOV DX, TEST1
```

the offset of **TEST1** is loaded into **DX** register. this solutions works for the majority of leading assemblers.