Emu8086 Tutorials

8086 assembler tutorials

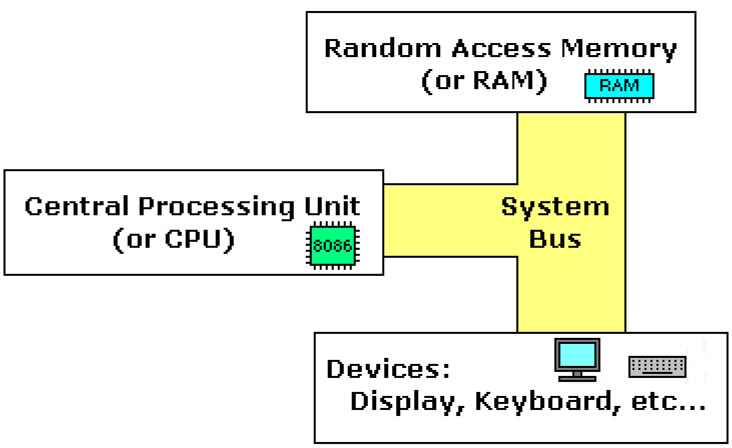
8086 assembler tutorial for beginners (part 1)

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 high level programming language (java, basic, c/c++, pascal...) that may help you a lot.
but even if you are familiar with assembler, it is still a good idea to look through this document in order to study emu8086 syntax.

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

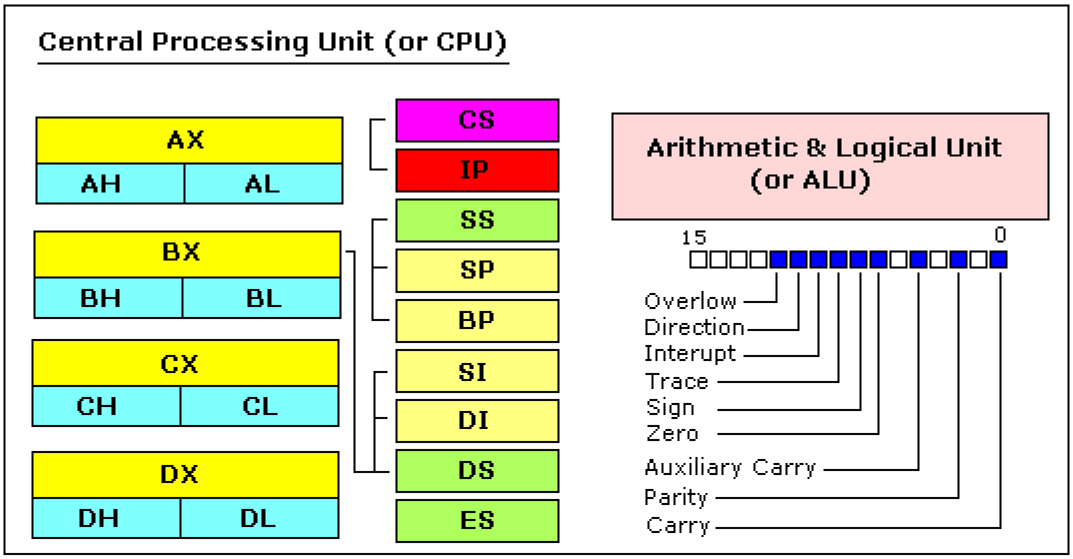
what is assembly language?

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



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

inside the CPU



general purpose registers

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

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

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

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

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

segment registers

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

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

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

(1230h * 10h + 45h = 12345h):

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

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

special purpose registers

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

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

8086 assembler tutorial for beginners (part 2)

Memory Access

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

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

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

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

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

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

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

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

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

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

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

```
| 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 are an examples of a valid addressing modes: **[BX+5]** , **[BX+SI]** , **[DI+BX-4]**

---

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

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

7h = 7

70h = 112

---

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

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

for example:

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

assembler supports shorter prefixes as well:

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

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

MOV instruction

- copies the **second operand** (source) to the **first operand** (destination).
- the source operand can be an immediate value, general-purpose register or memory location.
- the destination register can be a general-purpose register, or memory location.

- both operands must be the same size, which can be a byte or a word.

these types of operands are supported:

MOV REG, memory
MOV memory, REG
MOV REG, REG
MOV memory, immediate
MOV REG, immediate

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

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

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

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

MOV SREG, memory
MOV memory, SREG
MOV REG, SREG
MOV SREG, REG

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

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

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

the **MOV** instruction <u>cannot</u> be used to set the value of the **CS** and **IP** registers.

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

```
ORG 100h           ; this directive required for a simple 1 segment .com program.
MOV AX, 0B800h     ; set AX to hexadecimal value of B800h.
MOV DS, AX         ; copy value of AX to DS.
MOV CL, 'A'        ; set CL to ASCII code of 'A', it is 41h.
MOV CH, 1101_1111b ; set CH to binary value.
MOV BX, 15Eh       ; set BX to 15Eh.
```

```
MOV [BX], CX       ; copy contents of CX to memory at B800:015E
RET                ; returns to operating system.
```

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

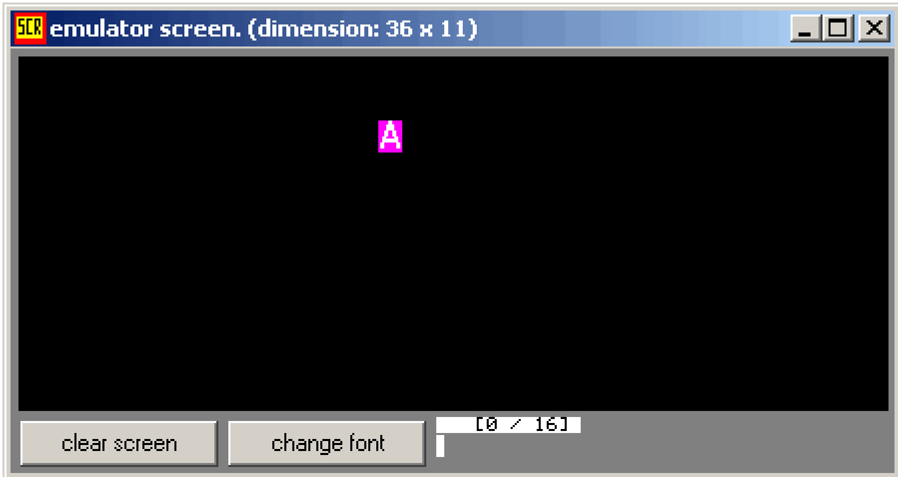
the emulator window should open with this program loaded, click [**Single Step**] button and watch the register values.

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 the 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.

8086 assembler tutorial for beginners (part 3)

Variables

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

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

Syntax for a variable declaration:

name **DB** *value*

name **DW** *value*

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

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

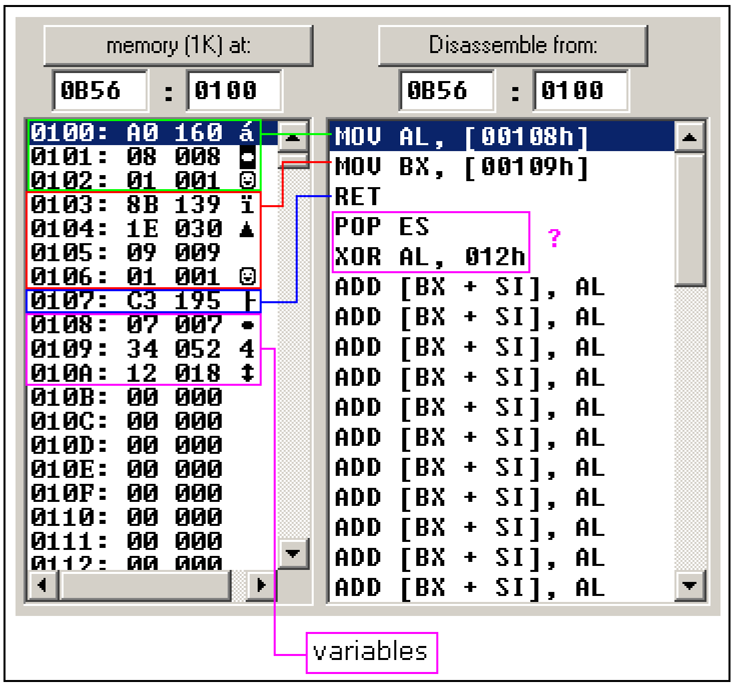
value - can be any numeric value in any supported numbering system (hexadecimal, binary, or decimal), or "**?**" symbol for variables that are not initialized.

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

```
ORG 100h

MOV AL, var1
MOV BX, var2

RET    ; stops the program.
```

```
VAR1 DB 7
var2 DW 1234h
```

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



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

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

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

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

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

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

```
ORG 100h

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

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

ORG 100h is a compiler directive (it tells compiler how to handle the source code). This directive is very important when you work

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

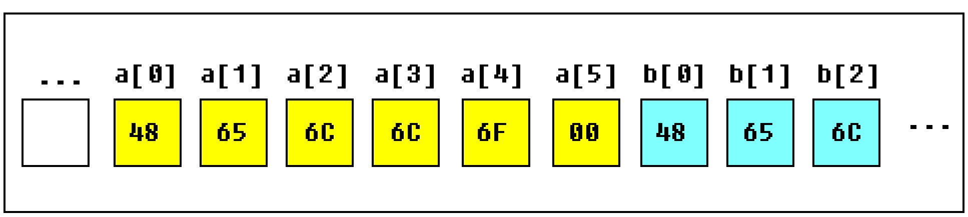
Arrays

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

Here are some array definition examples:

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

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



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

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

You can also use any of the memory index registers **BX, SI, DI, BP**, for example:
MOV SI, 3
MOV AL, a[SI]

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

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

for example:
c DB 5 DUP(9)
is an alternative way of declaring:
c DB 9, 9, 9, 9, 9

one more example:
d DB 5 DUP(1, 2)
is an alternative way of declaring:
d DB 1, 2, 1, 2, 1, 2, 1, 2, 1, 2

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

Getting the Address of a Variable

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

Reminder:

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

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

For example:

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

assembler supports shorter prefixes as well:

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

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

Here is first example:

```
ORG 100h

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

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

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

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

RET

VAR1   DB  22h

END
```

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

```
ORG 100h

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

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

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

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

RET

VAR1   DB  22h

END
```

Both examples have the same functionality.

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

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

Constants

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

name EQU < any expression >

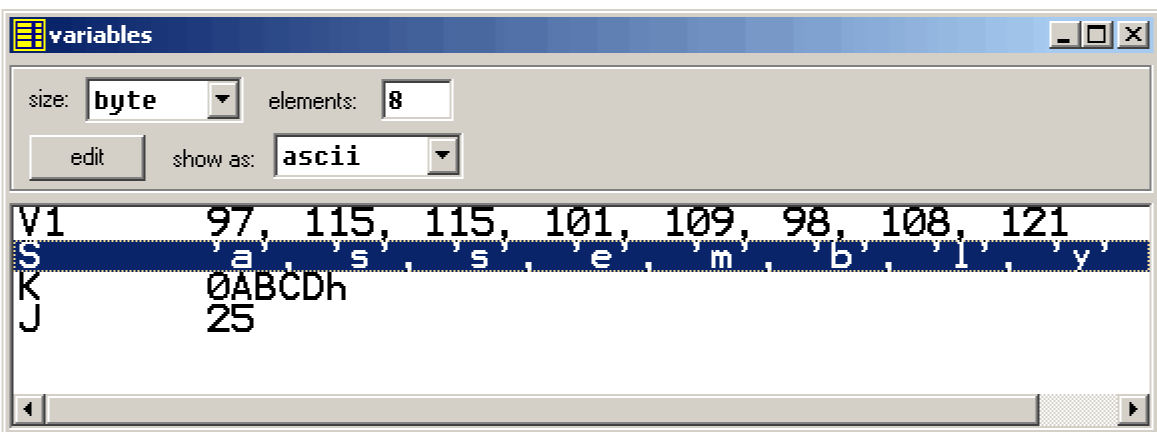
For example:

```
k EQU 5

MOV AX, k
```

The above example is functionally identical to code:

```
MOV AX, 5
```

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



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

Variable can be viewed in any numbering system:

- **HEX** - hexadecimal (base 16).

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

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

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

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

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

8086 assembler tutorial for beginners (part 4)

Interrupts

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

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

To make a **software interrupt** there is an **INT** instruction, it has very simple syntax:
INT value

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

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

```
ORG    100h              ; instruct compiler to make
simple single segment .com file.
```

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

MOV    AH, 0Eh    ; select sub-function.

; INT 10h / 0Eh sub-function
; receives an ASCII code of the
; character that will be printed
; in AL register.

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

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

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

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

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

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

RET               ; returns to operating system.
```

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

8086 assembler tutorial for beginners (part 5)

Library of common functions - emu8086.inc

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

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

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

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

---

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

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

- **CURSOROFF** - turns off the text cursor.
- **CURSORON** - turns on the text cursor.

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

```
include emu8086.inc

ORG    100h

PRINT 'Hello World!'

GOTOXY 10, 5

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

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

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

---

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

- **PRINT_STRING** - procedure to print a null terminated string at current cursor position, receives address of string in **DS:SI** register. To use it declare: **DEFINE_PRINT_STRING** before **END** directive.
- **PTHIS** - procedure to print a null terminated string at current cursor position (just as PRINT_STRING), but receives address

of string from Stack. The ZERO TERMINATED string should be defined just after the CALL instruction. For example:

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

To use it declare: **DEFINE_PTHIS** before **END** directive.

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

- **CLEAR_SCREEN** - procedure to clear the screen, (done by scrolling entire screen window), and set cursor position to top of it. To use it declare: **DEFINE_CLEAR_SCREEN** before **END** directive.

- **SCAN_NUM** - procedure that gets the multi-digit SIGNED number from the keyboard, and stores the result in **CX** register. To use it declare: **DEFINE_SCAN_NUM** before **END** directive.

- **PRINT_NUM** - procedure that prints a signed number in **AX** register. To use it declare: **DEFINE_PRINT_NUM** and **DEFINE_PRINT_NUM_UNS** before **END**directive.

- **PRINT_NUM_UNS** - procedure that prints out an unsigned number in **AX** register. To use it declare: **DEFINE_PRINT_NUM_UNS** before **END** directive.

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

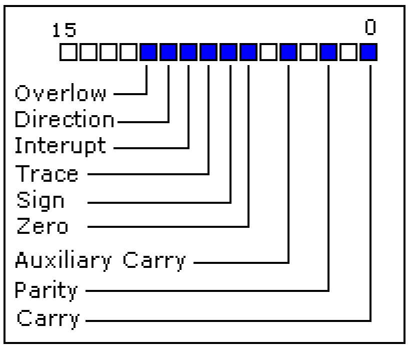
```
include 'emu8086.inc'

ORG    100h

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

MOV    AX, CX         ; copy the number to AX.

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

CALL   print_num      ; print number in AX.

RET                   ; return to operating system.

msg1   DB  'Enter the number: ', 0

DEFINE_SCAN_NUM
DEFINE_PRINT_STRING
DEFINE_PRINT_NUM
DEFINE_PRINT_NUM_UNS  ; required for
print_num.
DEFINE_PTHIS

END                   ; directive to stop the compiler.
```

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

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.

8086 assembler tutorial for beginners (part 7)

program flow control

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

- **unconditional jumps**

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

  The basic syntax of **JMP** instruction:

  ```
  JMP label
  ```

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

  ```
  label1:
  label2:
  a:
  ```

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

  ```
  x1:
  MOV AX, 1

  x2: MOV AX, 2
  ```

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

```
org    100h

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

```
jmp    calc          ; go to 'calc'.

back:  jmp stop      ; go to 'stop'.

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

stop:

ret                  ; return to operating system.
```

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

- **Short Conditional Jumps**

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

Jump instructions that test single flag

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

JPE, JP	Jump if Parity Even.	PF = 1	JPO
JNZ , JNE	Jump if Not Zero (Not Equal).	ZF = 0	JZ, JE
JNC , JNB, JAE	Jump if Not Carry (Not Below, Above Equal).	CF = 0	JC, JB, JNAE
JNS	Jump if Not Sign.	SF = 0	JS
JNO	Jump if Not Overflow.	OF = 0	JO
JPO, JNP	Jump if Parity Odd (No Parity).	PF = 0	JPE, JP

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

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

```
   jnc a
   jnb a
   jae a

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

-

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's 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.
```

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. you can use **F5**hotkey to recompile and reload the program into the emulator.

---

loops

instruction	operation and jump condition	opposite instruction
LOOP	decrease cx, jump to label if cx not zero.	DEC CX and JCXZ
LOOPE	decrease cx, jump to label if cx not zero and equal (zf = 1).	LOOPNE
LOOPNE	decrease cx, jump to label if cx not zero and not equal (zf = 0).	LOOPE
LOOPNZ	decrease cx, jump to label if cx not zero and zf = 0.	LOOPZ

LOOPZ	decrease cx, jump to label if cx not zero and zf = 1.	LOOPNZ
JCXZ	jump to label if cx is zero.	OR CX, CX and JNZ

- loops are basically the same jumps, it is possible to code loops without using the loop instruction, by just using conditional jumps and compare, and this is just what loop does. all loop instructions use **CX** register to count steps, as you know CX register has 16 bits and the maximum value it can hold is 65535 or FFFF, however with some agility it is possible to put one loop into another, and another into another two, and three and etc... and receive a nice value of 65535 * 65535 * 65535 ....till infinity.... or the end of ram or stack memory. it is possible store original value of cx register using **push cx** instruction and return it to original when the internal loop ends with **pop cx**, for example:

```
org 100h

mov bx, 0  ; total step counter.

mov cx, 5
k1: add bx, 1
    mov al, '1'
    mov ah, 0eh
    int 10h
    push cx
    mov cx, 5
      k2: add bx, 1
      mov al, '2'
      mov ah, 0eh
      int 10h
      push cx
         mov cx, 5
         k3: add bx, 1
         mov al, '3'
         mov ah, 0eh
         int 10h
         loop k3    ; internal in internal loop.
      pop  cx
      loop  k2      ; internal loop.
    pop cx
```

- loop k1 ; external loop.
-
- ret
-
- bx counts total number of steps, by default emulator shows values in hexadecimal, you can double click the register to see the value in all available bases.

  just like all other conditional jumps loops have an opposite companion that can help to create workarounds, when the address of desired location is too far assemble automatically assembles reverse and long jump instruction, making total of 5 bytes instead of just 2, it can be seen in disassembler as well.

  for more detailed description and examples refer to **complete 8086 instruction set**

-

-

  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 an 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, but there must not be two or more labels with the same name.

  here's an example:

```
include "emu8086.inc"
```

```
org    100h

mov    al, 5
mov    bl, 5

cmp    al, bl     ; compare al - bl.


; je equal        ; there is only 1 byte



jne    not_equal  ; jump if al <> bl (zf = 0).
jmp    equal
not_equal:


add    bl, al
sub    al, 10
xor    al, bl

jmp skip_data
db 256 dup(0)     ; 256 bytes
skip_data:


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
```

Note: the latest version of the integrated 8086 assembler automatically creates a workaround by replacing the conditional jump with the opposite, and adding big unconditional jump. To check if you have the latest version of emu8086 click **help-> check for an update** from the menu.

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

```
org    100h

; unconditional jump forward:
; skip over next 3 bytes + itself
; the machine code of short jmp instruction takes 2
bytes.
jmp $3+2
a db 3    ; 1 byte.
b db 4    ; 1 byte.
c db 4    ; 1 byte.

; conditional jump back 5 bytes:
mov bl,9
dec bl      ; 2 bytes.
cmp bl, 0   ; 3 bytes.
jne $-5     ; jump 5 bytes back

ret
```

8086 assembler tutorial for beginners (part 8)

Procedures

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

The syntax for procedure declaration:

```
name PROC

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

RET
name ENDP
```

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

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

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

CALL instruction is used to call a procedure.

Here is an example:

```
ORG    100h
```

```
CALL   m1

MOV    AX, 2

RET                  ; return to operating system.

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

END
```

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

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

```
ORG    100h

MOV    AL, 1
MOV    BL, 2

CALL   m2
CALL   m2
CALL   m2
CALL   m2

RET                  ; return to operating system.

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

END
```

In the above example value of **AL** register is update every time the procedure is called, **BL** register stays unchanged, so this algorithm

calculates **2** in power of **4**,
so final result in **AX** register is **16** (or 10h).

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

```
ORG    100h

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

CALL   print_me

RET                  ; return to operating system.

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

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

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

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

    ADD  SI, 1        ; advance index of string array.

    JMP  next_char    ; go back, and type another char.

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

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

END
```

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

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.

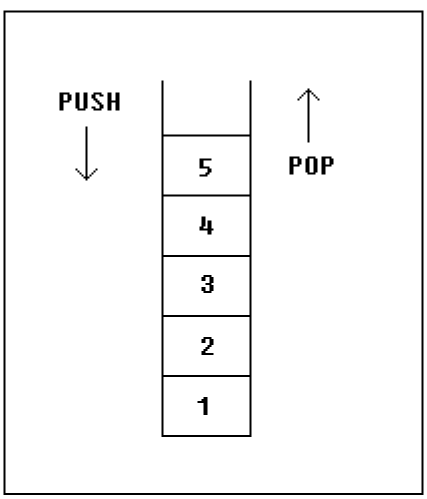
SREG: DS, ES, SS, (except CS).

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.

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.