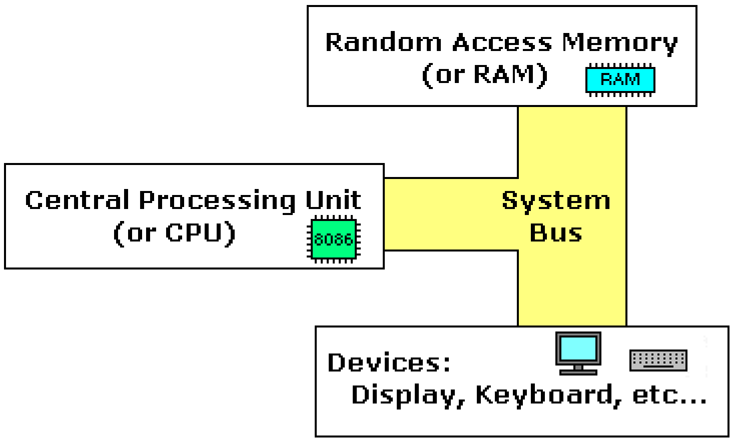
8086 Assembler

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 to understand anything. The simple computer model as I see it:

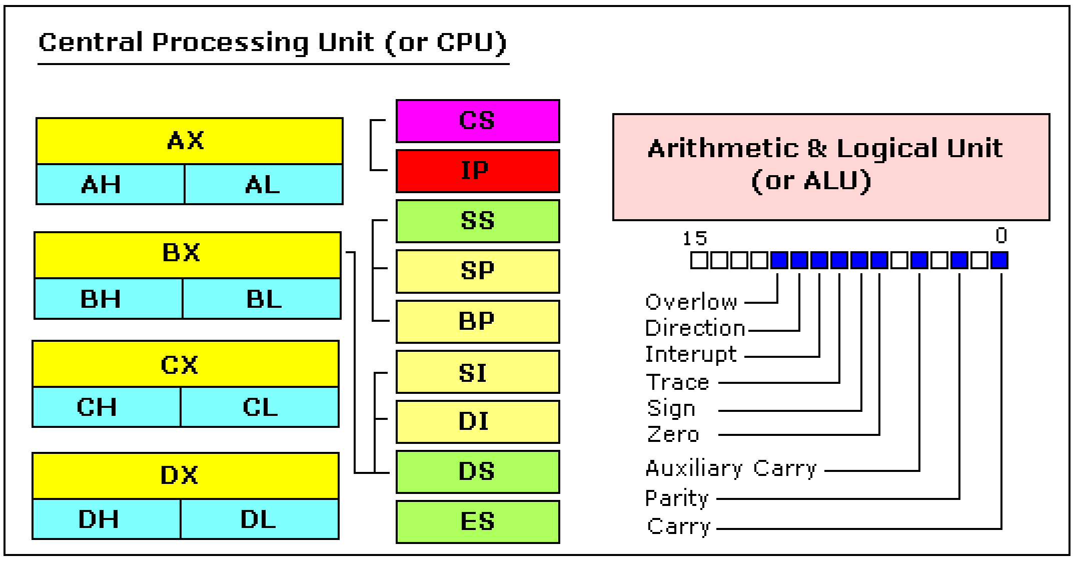


The **system bus** (shown in yellow) connects the various components of a computer.

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 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.

4 general purpose registers (AX, BX, CX, DX) are made of two separates 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 the 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 storetemporary 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 toaccess any memory value. For example, if we would like to access memoryat the physical address **12345h** (hexadecimal), we should set the **DS = 1230h** and **SI = 0045h**. This is good, since this way we can access muchmore 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 current 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.

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. For example, [BX], [BX+SI+7], variable, etc...

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 willbe 1234h * 10h + 7890h = 19BD0h.

<u>MOV</u> instruction

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

These types of operands are supported:

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

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

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

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

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

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

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

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

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

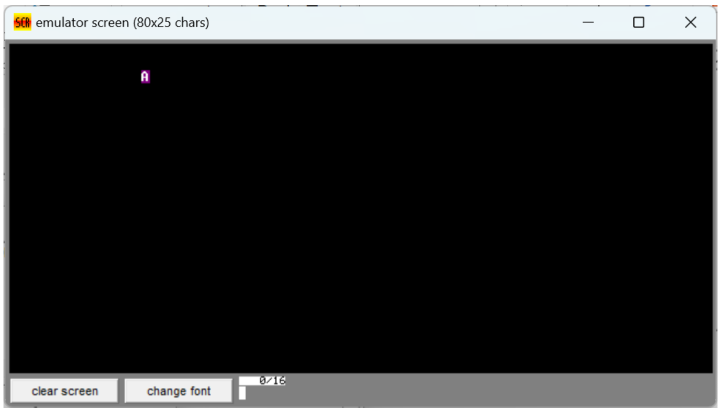
The **MOV** instruction cannot be used to set the value of the **CS** and **IP** registers.

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

```
ORG 100h              ; 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 editor, and press [**RUN**] button.

As you may guess, "**;**" is used for comments, anything after "**;**" symbol is ignored by compiler. You should see something like that when program finishes:



(this is how it looks in **emu8086** microprosessor emulator).

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

Variables

Variable is a memory location. For a programmer it is much easier to have some value be kept in a variable named "**var1**" than 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 this section 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
```

ORG 100h is a compiler directive (it says to compiler how to handle the source code). This directive is very important when you work with variables. It says to 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**? The 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.

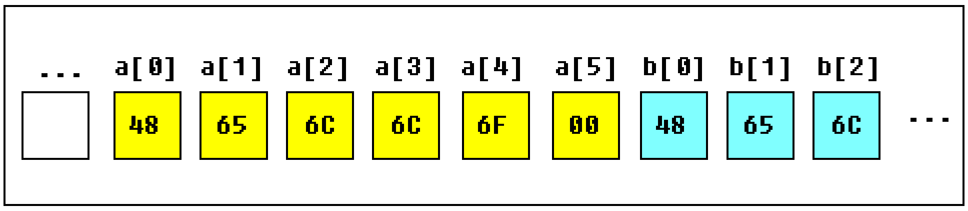
Arrays

Arrays can be seen as chains of variables. A text string is anexample 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**:

<u>number</u> DUP (<u>value(s)</u>)

<u>number</u> - number of duplicate to make (any constant value).

<u>value</u> - 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 youto get the address of an indexed variable. Getting the address of the variable can be very useful in some situations, for example when you need to pass parameters to a procedure.

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

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

RET

VAR1 DB 22h
```

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

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

RET

VAR1 DB 22h
```

Both examples have the same functionality. These lines:

```
LEA BX, VAR1
MOV BX, OFFSET VAR1
```

are even compiled into the same machine code:

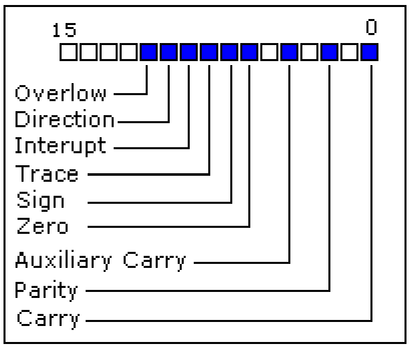
```
MOV BX, num
```

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**!

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**

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** instruction affects flags only and do not store a result (these instruction are used to make decisions during programexecution).
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**.

Second group: **MUL, DIV**

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 instruction affect these flags only:
CF, **OF**

When the result is over operand size these flags are set to **1**, when resultfits in operand size these flags are set to **0**.

- **MUL** :

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

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

- **DIV** :
  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**

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**.

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.
```

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 jump (jump only when some conditions are in act).

Jump instructions

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.

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:

```
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).

MOV CX,   0   ; if it gets here, then AL <> BL,
JMP    stop       ; so MOV 0, and jump to stop.

equal:            ; if gets here,
MOV CX, 1      ; then AL = BL, so MOV 1.

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.

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.

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
```

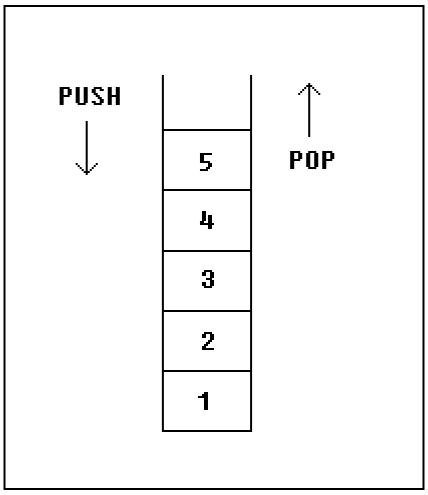
REG: AX, BX, CX, DX, DI, SI, BP, SP.
SREG: DS, ES, SS, CS.
memory: [BX], [BX+SI+7], 16 bit variable, etc...
immediate: 5, -24, 3Fh, 10001101b, etc...

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...

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 may be corrupted, and it will be impossible to return to operating system. As you already know we use **RET** instruction to return to the operating system, so when a 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 many 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
```

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
```

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.

In **emu8086** Microprocessor Emulator you can visually see thestack operation by clicking on [**Stack**] button on emulator window. Thetop of the stack is marked with "**<**" sign.

Interrupts

Interrupts can be seen as several 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 foryou. 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 interruptions** 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 notcorrect. 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 may be in use. Generally other registers are used to pass parameters and data to sub-functions. The following example uses **INT 10h** sub-function **0Eh** to type a "Hello!" message. This functiondisplays a character on the screen, advancing the cursor and scrolling the screen as necessary.

```
#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.
```