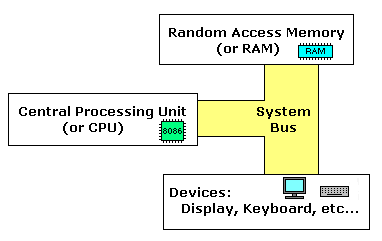
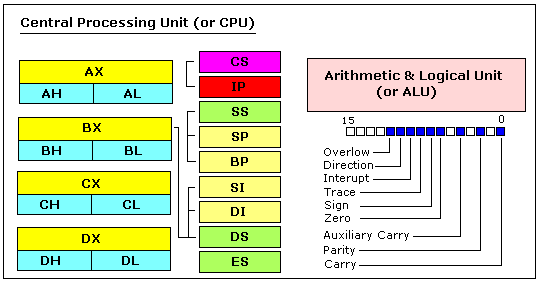
**8086 Assembler**   
  
  
This tutorial is intended for those who are not familiar with assembler at all, or have a very distant idea about it. Of course if you have knowledge of some other programming language (Basic, C/C++, Pascal...) that may help you a lot.   
But even if you are familiar with assembler, it is still a good idea to look through this document in order to study *Emu8086* syntax.   
  
It is assumed that you have some knowledge about number representation (HEX/BIN), if not it is highly recommended to study [**Numbering Systems Tutorial**](http://ce.kashanu.ac.ir/sabaghian/assembly/8086%20tutorial/Numbering%20Systems%20Tutorial.htm) before you proceed.   
  
  
**What is an assembly language?**   
  
Assembly language is a low level programming language. You need to get some knowledge about computer structure in order to understand anything. The simple computer model as I see it:  
[](http://ce.kashanu.ac.ir/sabaghian/assembly/8086%20tutorial/fp1/MODEL.GIF)   
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.

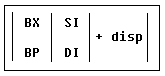
Although it is possible to store any data in the segment registers, this is never a good idea. The segment registers have a very special purpose - pointing at accessible blocks of memory.   
  
Segment registers work together with general purpose register to access any memory value. For example if we 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):  
http://ce.kashanu.ac.ir/sabaghian/assembly/8086%20tutorial/fp1/effective_address.gif   
The address formed with 2 registers is called an **effective address**.   
By default **BX, SI** and **DI** registers work with **DS** segment register;  
**BP** and **SP** work with **SS** segment register.  
Other general purpose registers cannot form an effective address!   
Also, although **BX** can form an effective address, **BH** and **BL** cannot!   
  
**SPECIAL PURPOSE REGISTERS**

* **IP** - the instruction pointer.
* **Flags Register** - determines the current state of the processor.

**IP** register always works together with **CS** segment register and it points to currently executing instruction.  
**Flags Register** is modified automatically by CPU after mathematical operations, this allows to determine the type of the result, and to determine conditions to transfer control to other parts of the program.  
Generally you cannot access these registers directly.

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

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

**d8** - stays for 8 bit displacement.  
  
**d16** - stays for 16 bit displacement.  
  
Displacement can be a immediate value or offset of a variable, or even both. It's up to compiler to calculate a single immediate value.  
  
Displacement can be inside or outside of **[ ]** symbols, compiler generates the same machine code for both ways.   
  
Displacement is a **signed** value, so it can be both positive or negative.   
  
Generally the compiler takes care about difference between **d8** and **d16**, and generates the required machine code.   
  
  
For example, let's assume that **DS = 100**, **BX = 30**, **SI = 70**.  
The following addressing mode: **[BX + SI] + 25**   
is calculated by processor to this physical address: **100 \* 16 + 30 + 70 + 25 = 1725**.   
  
By default **DS** segment register is used for all modes except those with **BP** register, for these **SS** segment register is used.   
  
There is an easy way to remember all those possible combinations using this chart:   
  
   
You can form all valid combinations by taking only one item from each column or skipping the column by not taking anything from it. As you see **BX** and **BP** never go together. **SI** and **DI** also don't go together. Here is an example of a valid addressing mode:**[BX+5]**. 

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

In order to say the compiler about data type,  
these prefixes should be used:  
  
**BYTE PTR** - for byte.  
**WORD PTR** - for word (two bytes).  
  
For example:

BYTE PTR [BX] ; byte access.

or

WORD PTR [BX] ; word access.

*Emu8086* supports shorter prefixes as well:  
  
**b.** - for **BYTE PTR**  
**w.** - for **WORD PTR**  
  
sometimes compiler can calculate the data type automatically, but you may not and should not rely on that when one of the operands is an immediate value. 

**MOV instruction** 

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

|  |
| --- |
| These types of operands are supported:  MOV REG, memory MOV memory, REG MOV REG, REG MOV memory, immediate MOV REG, immediate  **REG**: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.  **memory**: [BX], [BX+SI+7], variable, etc...  **immediate**: 5, -24, 3Fh, 10001101b, etc... |

|  |
| --- |
| For segment registers only these types of **MOV** are supported:  MOV SREG, memory MOV memory, SREG MOV REG, SREG MOV SREG, REG  **SREG**: DS, ES, SS, and only as second operand: CS.  **REG**: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.  **memory**: [BX], [BX+SI+7], variable, etc... |

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

|  |
| --- |
| Here is a short program that demonstrates the use of **MOV** instruction:  #MAKE\_COM# ; instruct compiler to make COM file.  ORG 100h ; directive required for a COM program.  MOV AX, 0B800h ; set AX to hexadecimal value of B800h.  MOV DS, AX ; copy value of AX to DS.  MOV CL, 'A' ; set CL to ASCII code of 'A', it is 41h.  MOV CH, 01011111b ; set CH to binary value.  MOV BX, 15Eh ; set BX to 15Eh.  MOV [BX], CX ; copy contents of CX to memory at B800:015E  RET ; returns to operating system. |

**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.   
  
Label supports two types of variables: **BYTE** and **WORD**. 

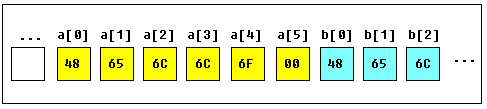
|  |
| --- |
| Syntax for a variable declaration:  *name* **DB** *value*  *name* **DW** *value*  **DB** - stays for Define Byte. **DW** - stays for Define Word.  *name* - can be any letter or digit combination, though it should start with a letter. It's possible to declare unnamed variables by not specifying 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: 

|  |
| --- |
| MOV AL, var1  MOV BX, var2  VAR1 DB 7  var2 DW 1234h |

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 the debugger 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: 

|  |
| --- |
| DB 0A0h  DB 08h  DB 01h  DB 8Bh  DB 1Eh  DB 09h  DB 01h  DB 0C3h  DB 7  DB 34h  DB 12h |

**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 assembler sees a string inside quotes it automatically converts it to set of bytes. This chart shows a part of the memory where these arrays are declared:  
  
   
  
You can access the value of any element in array using square brackets, for example:  
MOV AL, a[3]   
  
You can also use any of the memory index registers **BX, SI, DI, BP**, for example:  
MOV SI, 3  
MOV AL, a[SI]  
  
  
If you need to declare a large array you can use **DUP** operator.  
The syntax for **DUP**:  
  
number DUP ( value(s) )   
number - number of duplicate to make (any constant value).  
value - expression that DUP will duplicate.  
  
for example:  
c DB 5 DUP(9)   
is an alternative way of declaring:  
c DB 9, 9, 9, 9, 9   
  
one more example:  
d DB 5 DUP(1, 2)   
is an alternative way of declaring:  
d DB 1, 2, 1, 2, 1, 2, 1, 2, 1, 2   
  
Of course, you can use **DW** instead of **DB** if it's required to keep values larger then 255, or smaller then -128. **DW** cannot be used to declare strings!   
  
The expansion of **DUP** operand should not be over 1020 characters! (the expansion of last example is 13 chars), if you need to declare huge array divide declaration it in two lines (you will get a single huge array in the memory). 

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

**Reminder:**  
In order to tell the compiler about data type,  
these prefixes should be used:  
  
**BYTE PTR** - for byte.  
**WORD PTR** - for word (two bytes).  
  
For example:

BYTE PTR [BX] ; byte access.

or

WORD PTR [BX] ; word access.

Here is first example: 

|  |
| --- |
| 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 |

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

|  |
| --- |
| 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**!

**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 |

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).

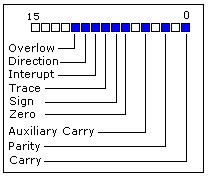
You can edit a variable's value when your program is running, simply double click it, or select it and click **Edit** button.   
  
It is possible to enter numbers in any system, hexadecimal numbers should have "**h**" suffix, binary "**b**" suffix, octal "**o**" suffix, decimal numbers require no suffix. 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...

**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. 

|  |
| --- |
| ; 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. |

**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.

**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:

|  |
| --- |
| 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: |

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

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

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

* As you can see there are some instructions that do that same thing, that's correct, they even are assembled into the same machine code, so it's good to remember that when you compile **JE** instruction - you will get it disassembled as: **JZ**.  
  Different names are used to make programs easier to understand and code.   
    
    
  **Jump instructions for signed numbers**

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

* <> - sign means not equal.   
    
    
  **Jump instructions for unsigned numbers**

|  |  |  |  |
| --- | --- | --- | --- |
| Instruction | Description | Condition | Opposite Instruction |
| JE , JZ | Jump if Equal (=). Jump if Zero. | ZF = 1 | JNE, JNZ |
| JNE , JNZ | Jump if Not Equal (<>). Jump if Not Zero. | ZF = 0 | JE, JZ |
| JA , JNBE | Jump if Above (>). Jump if Not Below or Equal (not <=). | CF = 0 and ZF = 0 | JNA, JBE |
| JB , JNAE, JC | Jump if Below (<). Jump if Not Above or Equal (not >=). Jump if Carry. | CF = 1 | JNB, JAE, JNC |
| JAE , JNB, JNC | Jump if Above or Equal (>=). Jump if Not Below (not <). Jump if Not Carry. | CF = 0 | JNAE, JB |
| JBE , JNA | Jump if Below or Equal (<=). Jump if Not Above (not >). | CF = 1 or ZF = 1 | JNBE, JA |

* Generally, when it is required to compare numeric values **CMP** instruction is used (it does the same as **SUB** (subtract) instruction, but does not keep the result, just affects the flags).  
    
  The logic is very simple, for example:  
  it's required to compare 5 and 2,  
  5 - 2 = 3  
  the result is not zero (Zero Flag is set to 0).   
    
  Another example:  
  it's required to compare 7 and 7,  
  7 - 7 = 0  
  the result is zero! (Zero Flag is set to 1 and **JZ** or **JE** will do the jump).   
    
  Here is an example of **CMP** instruction and conditional jump:

|  |
| --- |
| 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 al,'N' ; if it gets here, then AL <> BL,  JMP stop ; so print 'N', and jump to stop.  equal: ; if gets here,  MOV al,'Y' ; then AL = BL, so print 'Y'.  stop: |

* Try the above example with different numbers for **AL** and **BL**, open flags by clicking on [**FLAGS**] button, use [**Single Step**] and see what happens, don't forget to recompile and reload after every change (use **F5** shortcut).
* All conditional jumps have one big limitation, unlike **JMP** instruction they can only jump **127** bytes forward and **128** bytes backward (note that most instructions are assembled into 3 or more bytes).   
    
  We can easily avoid this limitation using a cute trick:
  + Get a opposite conditional jump instruction from the table above, make it jump to *label\_x*.
  + Use **JMP** instruction to jump to desired location.
  + Define *label\_x:* just after the **JMP** instruction.

*label\_x:* - can be any valid label name.   
  
Here is an example:

|  |
| --- |
| MOV AL, 25 ; set AL to 25.  MOV BL, 10 ; set BL to 10.  CMP AL, BL ; compare AL - BL.  JNE not\_equal ; jump if AL <> BL (ZF = 0).  JMP equal  not\_equal:  ; let's assume that here we  ; have a code that is assembled  ; to more then 127 bytes...  MOV al,'N' ; if it gets here, then AL <> BL,  JMP stop ; so print 'N', and jump to stop.  equal: ; if gets here,  MOV al,'Y' ; then AL = BL, so print 'Y'. |

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

|  |
| --- |
| ; unconditional jump forward:  ; skip over next 2 bytes,  JMP $2  a DB 3 ; 1 byte.  b DB 4 ; 1 byte.  ; JCC jump back 7 bytes:  ; (JMP takes 2 bytes itself)  MOV BL,9  DEC BL ; 2 bytes.  CMP BL, 0 ; 3 bytes.  JNE $-7 |

**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: 

|  |
| --- |
| CALL m1  MOV AX, 2  RET ; return to operating system.  m1 PROC  MOV BX, 5  RET ; return to caller.  m1 ENDP |

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: 

|  |
| --- |
| 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 |

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: 

|  |
| --- |
| 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 byte ptr [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 |

**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: 

|  |
| --- |
| 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. |

Another use of the stack is for exchanging the values,  
here is an example:

|  |
| --- |
| 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. |

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.

**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.

|  |
| --- |
| 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 |

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  MyMacro2  MyMacro2 |

Operand types:  
  
**REG**: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.  
  
**SREG**: DS, ES, SS, and only as second operand: CS.  
  
**memory**: [BX], [BX+SI+7], variable, etc...(see [**Memory Access**](http://ce.kashanu.ac.ir/sabaghian/assembly/8086%20tutorial/8086%20Assembler%20Tutorial%20for%20Beginners%20(Part%202).htm)).  
  
**immediate**: 5, -24, 3Fh, 10001101b, etc...

Notes:

* When two operands are required for an instruction they are separated by comma. For example:  
    
  REG, memory
* When there are two operands, both operands must have the same size (except shift and rotate instructions). For example:  
    
  AL, DL  
  DX, AX  
  m1 DB ?  
  AL, m1  
  m2 DW ?  
  AX, m2
* Some instructions allow several operand combinations. For example:  
    
  memory, immediate  
  REG, immediate  
    
  memory, REG  
  REG, SREG
* Some examples contain macros, so it is advisable to use **Shift + F8** hot key to *Step Over* (to make macro code execute at maximum speed set **step delay** to zero), otherwise emulator will step through each instruction of a macro. Here is an example that uses PRINTN macro:
* #make\_COM#
* include 'emu8086.inc'
* ORG 100h
* MOV AL, 1
* MOV BL, 2
* PRINTN 'Hello World!' ; macro.
* MOV CL, 3
* PRINTN 'Welcome!' ; macro.

RET

These marks are used to show the state of the flags:  
  
**1** - instruction sets this flag to **1**.  
**0** - instruction sets this flag to **0**.  
**r** - flag value depends on result of the instruction.  
**?** - flag value is undefined (maybe **1** or **0**).

**Some instructions generate exactly the same machine code, so disassembler may have a problem decoding to your original code. This is especially important for Conditional Jump instructions (see "**[**Program Flow Control**](http://ce.kashanu.ac.ir/sabaghian/assembly/8086%20tutorial/8086%20Assembler%20Tutorial%20for%20Beginners%20(Part%207).htm)**" in Tutorials for more information).** 

Instructions in alphabetical order:

|  |  |  |
| --- | --- | --- |
| Instruction | Operands | Description |
| AAA | No operands | ASCII Adjust after Addition. Corrects result in AH and AL after addition when working with BCD values.   It works according to the following Algorithm:   if low nibble of AL > 9 or AF = 1 then:   * AL = AL + 6 * AH = AH + 1 * AF = 1 * CF = 1   else   * AF = 0 * CF = 0   in both cases: clear the high nibble of AL.   Example:  MOV AX, 15 ; AH = 00, AL = 0Fh  AAA ; AH = 01, AL = 05  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | r | ? | ? | ? | ? | r | |
| AAD | No operands | ASCII Adjust before Division. Prepares two BCD values for division.   Algorithm:   * AL = (AH \* 10) + AL * AH = 0   Example:  MOV AX, 0105h ; AH = 01, AL = 05  AAD ; AH = 00, AL = 0Fh (15)  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | ? | r | r | ? | r | ? | |
| AAM | No operands | ASCII Adjust after Multiplication. Corrects the result of multiplication of two BCD values.   Algorithm:   * AH = AL / 10 * AL = remainder   Example:  MOV AL, 15 ; AL = 0Fh  AAM ; AH = 01, AL = 05  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | ? | r | r | ? | r | ? | |
| AAS | No operands | ASCII Adjust after Subtraction. Corrects result in AH and AL after subtraction when working with BCD values.   Algorithm:  if low nibble of AL > 9 or AF = 1 then:   * AL = AL - 6 * AH = AH - 1 * AF = 1 * CF = 1   else   * AF = 0 * CF = 0   in both cases: clear the high nibble of AL.   Example:  MOV AX, 02FFh ; AH = 02, AL = 0FFh  AAS ; AH = 01, AL = 09  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | r | ? | ? | ? | ? | r | |
| ADC | REG, memory memory, REG REG, REG memory, immediate REG, immediate | Add with Carry.   Algorithm:  operand1 = operand1 + operand2 + CF   Example:  STC ; set CF = 1  MOV AL, 5 ; AL = 5  ADC AL, 1 ; AL = 7  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | r | r | r | r | r | r | |
| ADD | REG, memory memory, REG REG, REG memory, immediate REG, immediate | Add.   Algorithm:  operand1 = operand1 + operand2   Example:  MOV AL, 5 ; AL = 5  ADD AL, -3 ; AL = 2  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | r | r | r | r | r | r | |
| AND | REG, memory memory, REG REG, REG memory, immediate REG, immediate | Logical AND between all bits of two operands. Result is stored in operand1.  These rules apply:  1 AND 1 = 1 1 AND 0 = 0 0 AND 1 = 0 0 AND 0 = 0   Example:  MOV AL, 'a' ; AL = 01100001b  AND AL, 11011111b ; AL = 01000001b ('A')  RET   |  |  |  |  |  | | --- | --- | --- | --- | --- | | C | Z | S | O | P | | 0 | r | r | 0 | r | |
| CALL | procedure name label 4-byte address | Transfers control to procedure, return address is (IP) is pushed to stack. *4-byte address* may be entered in this form: 1234h:5678h, first value is a segment second value is an offset (this is a far call, so CS is also pushed to stack).   Example:  #make\_COM#  ORG 100h ; for COM file.  CALL p1  ADD AX, 1  RET ; return to OS.  p1 PROC ; procedure declaration.  MOV AX, 1234h  RET ; return to caller.  p1 ENDP   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| CBW | No operands | Convert byte into word.   Algorithm:   if high bit of AL = 1 then:   * AH = 255 (0FFh)   else   * AH = 0   Example:  MOV AX, 0 ; AH = 0, AL = 0  MOV AL, -5 ; AX = 000FBh (251)  CBW ; AX = 0FFFBh (-5)  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| CLC | No operands | Clear Carry flag.   Algorithm:   CF = 0   |  | | --- | | C | | 0 | |
| CLD | No operands | Clear Direction flag. SI and DI will be incremented by chain instructions: CMPSB, CMPSW, LODSB, LODSW, MOVSB, MOVSW, STOSB, STOSW.   Algorithm:   DF = 0   |  | | --- | | D | | 0 | |
| CLI | No operands | Clear Interrupt enable flag. This disables hardware interrupts.   Algorithm:   IF = 0   |  | | --- | | I | | 0 | |
| CMC | No operands | Complement Carry flag. Inverts value of CF.   Algorithm:   if CF = 1 then CF = 0 if CF = 0 then CF = 1   |  | | --- | | C | | r | |
| CMP | REG, memory memory, REG REG, REG memory, immediate REG, immediate | Compare.   Algorithm:  operand1 - operand2   result is not stored anywhere, flags are set (OF, SF, ZF, AF, PF, CF) according to result.   Example:  MOV AL, 5  MOV BL, 5  CMP AL, BL ; AL = 5, ZF = 1 (so equal!)  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | r | r | r | r | r | r | |
| CMPSB | No operands | Compare bytes: ES:[DI] from DS:[SI].   Algorithm:   * DS:[SI] - ES:[DI] * set flags according to result: OF, SF, ZF, AF, PF, CF * if DF = 0 then   + SI = SI + 1   + DI = DI + 1   else   * + SI = SI - 1   + DI = DI - 1   Example: see **cmpsb.asm** in Samples.   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | r | r | r | r | r | r | |
| CMPSW | No operands | Compare words: ES:[DI] from DS:[SI].   Algorithm:   * DS:[SI] - ES:[DI] * set flags according to result: OF, SF, ZF, AF, PF, CF * if DF = 0 then   + SI = SI + 2   + DI = DI + 2   else   * + SI = SI - 2   + DI = DI - 2   Example: see **cmpsw.asm** in Samples.   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | r | r | r | r | r | r | |
| CWD | No operands | Convert Word to Double word.   Algorithm:   if high bit of AX = 1 then:   * DX = 65535 (0FFFFh)   else   * DX = 0   Example:  MOV DX, 0 ; DX = 0  MOV AX, 0 ; AX = 0  MOV AX, -5 ; DX AX = 00000h:0FFFBh  CWD ; DX AX = 0FFFFh:0FFFBh  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| DAA | No operands | Decimal adjust After Addition. Corrects the result of addition of two packed BCD values.   Algorithm:   if low nibble of AL > 9 or AF = 1 then:   * AL = AL + 6 * AF = 1   if AL > 9Fh or CF = 1 then:   * AL = AL + 60h * CF = 1   Example:  MOV AL, 0Fh ; AL = 0Fh (15)  DAA ; AL = 15h  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | r | r | r | r | r | r | |
| DAS | No operands | Decimal adjust After Subtraction. Corrects the result of subtraction of two packed BCD values.   Algorithm:   if low nibble of AL > 9 or AF = 1 then:   * AL = AL - 6 * AF = 1   if AL > 9Fh or CF = 1 then:   * AL = AL - 60h * CF = 1   Example:  MOV AL, 0FFh ; AL = 0FFh (-1)  DAS ; AL = 99h, CF = 1  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | r | r | r | r | r | r | |
| DEC | REG memory | Decrement.   Algorithm:  operand = operand - 1    Example:  MOV AL, 255 ; AL = 0FFh (255 or -1)  DEC AL ; AL = 0FEh (254 or -2)  RET   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Z | S | O | P | A | | r | r | r | r | r |   CF - unchanged! |
| DIV | REG memory | Unsigned divide.   Algorithm:  when operand is a **byte**: AL = AX / operand AH = remainder (modulus)  when operand is a **word**: AX = (DX AX) / operand DX = remainder (modulus)  Example:  MOV AX, 203 ; AX = 00CBh  MOV BL, 4  DIV BL ; AL = 50 (32h), AH = 3  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | ? | ? | ? | ? | ? | ? | |
| HLT | No operands | Halt the System.  Example:  MOV AX, 5  HLT   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| IDIV | REG memory | Signed divide.   Algorithm:  when operand is a **byte**: AL = AX / operand AH = remainder (modulus)  when operand is a **word**: AX = (DX AX) / operand DX = remainder (modulus)  Example:  MOV AX, -203 ; AX = 0FF35h  MOV BL, 4  IDIV BL ; AL = -50 (0CEh), AH = -3 (0FDh)  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | ? | ? | ? | ? | ? | ? | |
| IMUL | REG memory | Signed multiply.   Algorithm:  when operand is a **byte**: AX = AL \* operand.  when operand is a **word**: (DX AX) = AX \* operand.  Example:  MOV AL, -2  MOV BL, -4  IMUL BL ; AX = 8  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | r | ? | ? | r | ? | ? |   CF=OF=0 when result fits into operand of IMUL. |
| IN | AL, im.byte AL, DX AX, im.byte AX, DX | Input from port into **AL** or **AX**. Second operand is a port number. If required to access port number over 255 - **DX** register should be used.  Example:  IN AX, 4 ; get status of traffic lights.  IN AL, 7 ; get status of stepper-motor.   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| INC | REG memory | Increment.   Algorithm:  operand = operand + 1   Example:  MOV AL, 4  INC AL ; AL = 5  RET   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Z | S | O | P | A | | r | r | r | r | r |   CF - unchanged! |
| INT | immediate byte | Interrupt numbered by immediate byte (0..255).   Algorithm:  Push to stack:   * + flags register   + CS   + IP * IF = 0 * Transfer control to interrupt procedure   Example:  MOV AH, 0Eh ; teletype.  MOV AL, 'A'  INT 10h ; BIOS interrupt.  RET   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | I | | unchanged | | | | | | 0 | |
| INTO | No operands | Interrupt 4 if Overflow flag is 1.   Algorithm:  if OF = 1 then INT 4   Example:  ; -5 - 127 = -132 (not in -128..127)  ; the result of SUB is wrong (124),  ; so OF = 1 is set:  MOV AL, -5  SUB AL, 127 ; AL = 7Ch (124)  INTO ; process error.  RET |
| IRET | No operands | Interrupt Return.   Algorithm:  Pop from stack:   * + IP   + CS   + flags register  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | popped | | | | | | |
| JA | label | Short Jump if first operand is Above second operand (as set by CMP instruction). Unsigned.   Algorithm:  if (CF = 0) and (ZF = 0) then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 250  CMP AL, 5  JA label1  PRINT 'AL is not above 5'  JMP exit  label1:  PRINT 'AL is above 5'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JAE | label | Short Jump if first operand is Above or Equal to second operand (as set by CMP instruction). Unsigned.   Algorithm:  if CF = 0 then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 5  CMP AL, 5  JAE label1  PRINT 'AL is not above or equal to 5'  JMP exit  label1:  PRINT 'AL is above or equal to 5'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JB | label | Short Jump if first operand is Below second operand (as set by CMP instruction). Unsigned.   Algorithm:  if CF = 1 then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 1  CMP AL, 5  JB label1  PRINT 'AL is not below 5'  JMP exit  label1:  PRINT 'AL is below 5'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JBE | label | Short Jump if first operand is Below or Equal to second operand (as set by CMP instruction). Unsigned.   Algorithm:  if CF = 1 or ZF = 1 then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 5  CMP AL, 5  JBE label1  PRINT 'AL is not below or equal to 5'  JMP exit  label1:  PRINT 'AL is below or equal to 5'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JC | label | Short Jump if Carry flag is set to 1.   Algorithm:  if CF = 1 then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 255  ADD AL, 1  JC label1  PRINT 'no carry.'  JMP exit  label1:  PRINT 'has carry.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JCXZ | label | Short Jump if CX register is 0.   Algorithm:  if CX = 0 then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV CX, 0  JCXZ label1  PRINT 'CX is not zero.'  JMP exit  label1:  PRINT 'CX is zero.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JE | label | Short Jump if first operand is Equal to second operand (as set by CMP instruction). Signed/Unsigned.   Algorithm:  if ZF = 1 then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 5  CMP AL, 5  JE label1  PRINT 'AL is not equal to 5.'  JMP exit  label1:  PRINT 'AL is equal to 5.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JG | label | Short Jump if first operand is Greater then second operand (as set by CMP instruction). Signed.   Algorithm:  if (ZF = 0) and (SF = OF) then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 5  CMP AL, -5  JG label1  PRINT 'AL is not greater -5.'  JMP exit  label1:  PRINT 'AL is greater -5.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JGE | label | Short Jump if first operand is Greater or Equal to second operand (as set by CMP instruction). Signed.   Algorithm:  if SF = OF then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 2  CMP AL, -5  JGE label1  PRINT 'AL < -5'  JMP exit  label1:  PRINT 'AL >= -5'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JL | label | Short Jump if first operand is Less then second operand (as set by CMP instruction). Signed.   Algorithm:  if SF <> OF then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, -2  CMP AL, 5  JL label1  PRINT 'AL >= 5.'  JMP exit  label1:  PRINT 'AL < 5.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JLE | label | Short Jump if first operand is Less or Equal to second operand (as set by CMP instruction). Signed.   Algorithm:  if SF <> OF or ZF = 1 then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, -2  CMP AL, 5  JLE label1  PRINT 'AL > 5.'  JMP exit  label1:  PRINT 'AL <= 5.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JMP | label 4-byte address | Unconditional Jump. Transfers control to another part of the program. *4-byte address* may be entered in this form: 1234h:5678h, first value is a segment second value is an offset.   Algorithm:  always jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 5  JMP label1 ; jump over 2 lines!  PRINT 'Not Jumped!'  MOV AL, 0  label1:  PRINT 'Got Here!'  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JNA | label | Short Jump if first operand is Not Above second operand (as set by CMP instruction). Unsigned.   Algorithm:  if CF = 1 or ZF = 1 then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 2  CMP AL, 5  JNA label1  PRINT 'AL is above 5.'  JMP exit  label1:  PRINT 'AL is not above 5.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JNAE | label | Short Jump if first operand is Not Above and Not Equal to second operand (as set by CMP instruction). Unsigned.   Algorithm:  if CF = 1 then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 2  CMP AL, 5  JNAE label1  PRINT 'AL >= 5.'  JMP exit  label1:  PRINT 'AL < 5.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JNB | label | Short Jump if first operand is Not Below second operand (as set by CMP instruction). Unsigned.   Algorithm:  if CF = 0 then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 7  CMP AL, 5  JNB label1  PRINT 'AL < 5.'  JMP exit  label1:  PRINT 'AL >= 5.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JNBE | label | Short Jump if first operand is Not Below and Not Equal to second operand (as set by CMP instruction). Unsigned.   Algorithm:  if (CF = 0) and (ZF = 0) then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 7  CMP AL, 5  JNBE label1  PRINT 'AL <= 5.'  JMP exit  label1:  PRINT 'AL > 5.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JNC | label | Short Jump if Carry flag is set to 0.   Algorithm:  if CF = 0 then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 2  ADD AL, 3  JNC label1  PRINT 'has carry.'  JMP exit  label1:  PRINT 'no carry.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JNE | label | Short Jump if first operand is Not Equal to second operand (as set by CMP instruction). Signed/Unsigned.   Algorithm:  if ZF = 0 then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 2  CMP AL, 3  JNE label1  PRINT 'AL = 3.'  JMP exit  label1:  PRINT 'Al <> 3.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JNG | label | Short Jump if first operand is Not Greater then second operand (as set by CMP instruction). Signed.   Algorithm:  if (ZF = 1) and (SF <> OF) then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 2  CMP AL, 3  JNG label1  PRINT 'AL > 3.'  JMP exit  label1:  PRINT 'Al <= 3.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JNGE | label | Short Jump if first operand is Not Greater and Not Equal to second operand (as set by CMP instruction). Signed.   Algorithm:  if SF <> OF then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 2  CMP AL, 3  JNGE label1  PRINT 'AL >= 3.'  JMP exit  label1:  PRINT 'Al < 3.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JNL | label | Short Jump if first operand is Not Less then second operand (as set by CMP instruction). Signed.   Algorithm:  if SF = OF then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 2  CMP AL, -3  JNL label1  PRINT 'AL < -3.'  JMP exit  label1:  PRINT 'Al >= -3.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JNLE | label | Short Jump if first operand is Not Less and Not Equal to second operand (as set by CMP instruction). Signed.   Algorithm:  if (SF = OF) and (ZF = 0) then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 2  CMP AL, -3  JNLE label1  PRINT 'AL <= -3.'  JMP exit  label1:  PRINT 'Al > -3.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JNO | label | Short Jump if Not Overflow.   Algorithm:  if OF = 0 then jump  Example:  ; -5 - 2 = -7 (inside -128..127)  ; the result of SUB is correct,  ; so OF = 0:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, -5  SUB AL, 2 ; AL = 0F9h (-7)  JNO label1  PRINT 'overflow!'  JMP exit  label1:  PRINT 'no overflow.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JNP | label | Short Jump if No Parity (odd). Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.   Algorithm:  if PF = 0 then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 00000111b ; AL = 7  OR AL, 0 ; just set flags.  JNP label1  PRINT 'parity even.'  JMP exit  label1:  PRINT 'parity odd.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JNS | label | Short Jump if Not Signed (if positive). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.   Algorithm:  if SF = 0 then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 00000111b ; AL = 7  OR AL, 0 ; just set flags.  JNS label1  PRINT 'signed.'  JMP exit  label1:  PRINT 'not signed.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JNZ | label | Short Jump if Not Zero (not equal). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.   Algorithm:  if ZF = 0 then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 00000111b ; AL = 7  OR AL, 0 ; just set flags.  JNZ label1  PRINT 'zero.'  JMP exit  label1:  PRINT 'not zero.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JO | label | Short Jump if Overflow.   Algorithm:  if OF = 1 then jump  Example:  ; -5 - 127 = -132 (not in -128..127)  ; the result of SUB is wrong (124),  ; so OF = 1 is set:  include 'emu8086.inc'  #make\_COM#  org 100h  MOV AL, -5  SUB AL, 127 ; AL = 7Ch (124)  JO label1  PRINT 'no overflow.'  JMP exit  label1:  PRINT 'overflow!'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JP | label | Short Jump if Parity (even). Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.   Algorithm:  if PF = 1 then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 00000101b ; AL = 5  OR AL, 0 ; just set flags.  JP label1  PRINT 'parity odd.'  JMP exit  label1:  PRINT 'parity even.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JPE | label | Short Jump if Parity Even. Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.   Algorithm:  if PF = 1 then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 00000101b ; AL = 5  OR AL, 0 ; just set flags.  JPE label1  PRINT 'parity odd.'  JMP exit  label1:  PRINT 'parity even.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JPO | label | Short Jump if Parity Odd. Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.   Algorithm:  if PF = 0 then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 00000111b ; AL = 7  OR AL, 0 ; just set flags.  JPO label1  PRINT 'parity even.'  JMP exit  label1:  PRINT 'parity odd.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JS | label | Short Jump if Signed (if negative). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.   Algorithm:  if SF = 1 then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 10000000b ; AL = -128  OR AL, 0 ; just set flags.  JS label1  PRINT 'not signed.'  JMP exit  label1:  PRINT 'signed.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| JZ | label | Short Jump if Zero (equal). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.   Algorithm:  if ZF = 1 then jump  Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AL, 5  CMP AL, 5  JZ label1  PRINT 'AL is not equal to 5.'  JMP exit  label1:  PRINT 'AL is equal to 5.'  exit:  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| LAHF | No operands | Load AH from 8 low bits of Flags register.   Algorithm:  AH = flags register  AH bit: 7 6 5 4 3 2 1 0  [SF] [ZF] [0] [AF] [0] [PF] [1] [CF]  bits 1, 3, 5 are reserved.   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| LDS | REG, memory | Load memory double word into word register and DS.   Algorithm:   * REG = first word * DS = second word   Example:  #make\_COM#  ORG 100h  LDS AX, m  RET  m DW 1234h  DW 5678h  END  AX is set to 1234h, DS is set to 5678h.   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| LEA | REG, memory | Load Effective Address.   Algorithm:   * REG = address of memory (offset)   Generally this instruction is replaced by MOV when assembling when possible.   Example:  #make\_COM#  ORG 100h  LEA AX, m  RET  m DW 1234h  END  AX is set to: 0104h. LEA instruction takes 3 bytes, RET takes 1 byte, we start at 100h, so the address of 'm' is 104h.   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| LES | REG, memory | Load memory double word into word register and ES.   Algorithm:   * REG = first word * ES = second word   Example:  #make\_COM#  ORG 100h  LES AX, m  RET  m DW 1234h  DW 5678h  END  AX is set to 1234h, ES is set to 5678h.   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| LODSB | No operands | Load byte at DS:[SI] into AL. Update SI.  Algorithm:   * AL = DS:[SI] * if DF = 0 then   + SI = SI + 1   else   * + SI = SI - 1   Example:  #make\_COM#  ORG 100h  LEA SI, a1  MOV CX, 5  MOV AH, 0Eh  m: LODSB  INT 10h  LOOP m  RET  a1 DB 'H', 'e', 'l', 'l', 'o'   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| LODSW | No operands | Load word at DS:[SI] into AX. Update SI.  Algorithm:   * AX = DS:[SI] * if DF = 0 then   + SI = SI + 2   else   * + SI = SI - 2   Example:  #make\_COM#  ORG 100h  LEA SI, a1  MOV CX, 5  REP LODSW ; finally there will be 555h in AX.  RET  a1 dw 111h, 222h, 333h, 444h, 555h   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| LOOP | label | Decrease CX, jump to label if CX not zero.   Algorithm:   * CX = CX - 1 * if CX <> 0 then   + jump   else   * + no jump, continue   Example:  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV CX, 5  label1:  PRINTN 'loop!'  LOOP label1  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| LOOPE | label | Decrease CX, jump to label if CX not zero and Equal (ZF = 1).   Algorithm:   * CX = CX - 1 * if (CX <> 0) and (ZF = 1) then   + jump   else   * + no jump, continue   Example:  ; Loop until result fits into AL alone,  ; or 5 times. The result will be over 255  ; on third loop (100+100+100),  ; so loop will exit.  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AX, 0  MOV CX, 5  label1:  PUTC '\*'  ADD AX, 100  CMP AH, 0  LOOPE label1  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| LOOPNE | label | Decrease CX, jump to label if CX not zero and Not Equal (ZF = 0).   Algorithm:   * CX = CX - 1 * if (CX <> 0) and (ZF = 0) then   + jump   else   * + no jump, continue   Example:  ; Loop until '7' is found,  ; or 5 times.  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV SI, 0  MOV CX, 5  label1:  PUTC '\*'  MOV AL, v1[SI]  INC SI ; next byte (SI=SI+1).  CMP AL, 7  LOOPNE label1  RET  v1 db 9, 8, 7, 6, 5   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| LOOPNZ | label | Decrease CX, jump to label if CX not zero and ZF = 0.   Algorithm:   * CX = CX - 1 * if (CX <> 0) and (ZF = 0) then   + jump   else   * + no jump, continue   Example:  ; Loop until '7' is found,  ; or 5 times.  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV SI, 0  MOV CX, 5  label1:  PUTC '\*'  MOV AL, v1[SI]  INC SI ; next byte (SI=SI+1).  CMP AL, 7  LOOPNZ label1  RET  v1 db 9, 8, 7, 6, 5   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| LOOPZ | label | Decrease CX, jump to label if CX not zero and ZF = 1.   Algorithm:   * CX = CX - 1 * if (CX <> 0) and (ZF = 1) then   + jump   else   * + no jump, continue   Example:  ; Loop until result fits into AL alone,  ; or 5 times. The result will be over 255  ; on third loop (100+100+100),  ; so loop will exit.  include 'emu8086.inc'  #make\_COM#  ORG 100h  MOV AX, 0  MOV CX, 5  label1:  PUTC '\*'  ADD AX, 100  CMP AH, 0  LOOPZ label1  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| MOV | REG, memory memory, REG REG, REG memory, immediate REG, immediate  SREG, memory memory, SREG REG, SREG SREG, REG | Copy operand2 to operand1.  The MOV instruction cannot:   * set the value of the CS and IP registers. * copy value of one segment register to another segment register (should copy to general register first). * copy immediate value to segment register (should copy to general register first).   Algorithm:  operand1 = operand2  Example:  #make\_COM#  ORG 100h  MOV AX, 0B800h ; set AX = B800h (VGA memory).  MOV DS, AX ; copy value of AX to DS.  MOV CL, 'A' ; CL = 41h (ASCII code).  MOV CH, 01011111b ; CL = color attribute.  MOV BX, 15Eh ; BX = position on screen.  MOV [BX], CX ; w.[0B800h:015Eh] = CX.  RET ; returns to operating system.   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| MOVSB | No operands | Copy byte at DS:[SI] to ES:[DI]. Update SI and DI.  Algorithm:   * ES:[DI] = DS:[SI] * if DF = 0 then   + SI = SI + 1   + DI = DI + 1   else   * + SI = SI - 1   + DI = DI - 1   Example:  #make\_COM#  ORG 100h  LEA SI, a1  LEA DI, a2  MOV CX, 5  REP MOVSB  RET  a1 DB 1,2,3,4,5  a2 DB 5 DUP(0)   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| MOVSW | No operands | Copy **word** at DS:[SI] to ES:[DI]. Update SI and DI.  Algorithm:   * ES:[DI] = DS:[SI] * if DF = 0 then   + SI = SI + 2   + DI = DI + 2   else   * + SI = SI - 2   + DI = DI - 2   Example:  #make\_COM#  ORG 100h  LEA SI, a1  LEA DI, a2  MOV CX, 5  REP MOVSW  RET  a1 DW 1,2,3,4,5  a2 DW 5 DUP(0)   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| MUL | REG memory | Unsigned multiply.   Algorithm:  when operand is a **byte**: AX = AL \* operand.  when operand is a **word**: (DX AX) = AX \* operand.  Example:  MOV AL, 200 ; AL = 0C8h  MOV BL, 4  MUL BL ; AX = 0320h (800)  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | r | ? | ? | r | ? | ? |   CF=OF=0 when high section of the result is zero. |
| NEG | REG memory | Negate. Makes operand negative (two's complement).   Algorithm:   * Invert all bits of the operand * Add 1 to inverted operand   Example:  MOV AL, 5 ; AL = 05h  NEG AL ; AL = 0FBh (-5)  NEG AL ; AL = 05h (5)  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | r | r | r | r | r | r | |
| NOP | No operands | No Operation.  Algorithm:   * Do nothing   Example:  ; do nothing, 3 times:  NOP  NOP  NOP  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| NOT | REG memory | Invert each bit of the operand.  Algorithm:   * if bit is 1 turn it to 0. * if bit is 0 turn it to 1.   Example:  MOV AL, 00011011b  NOT AL ; AL = 11100100b  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| OR | REG, memory memory, REG REG, REG memory, immediate REG, immediate | Logical OR between all bits of two operands. Result is stored in first operand.  These rules apply:  1 OR 1 = 1 1 OR 0 = 1 0 OR 1 = 1 0 OR 0 = 0   Example:  MOV AL, 'A' ; AL = 01000001b  OR AL, 00100000b ; AL = 01100001b ('a')  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | 0 | r | r | 0 | r | ? | |
| OUT | im.byte, AL im.byte, AX DX, AL DX, AX | Output from **AL** or **AX** to port. First operand is a port number. If required to access port number over 255 - **DX** register should be used.   Example:  MOV AX, 0FFFh ; Turn on all  OUT 4, AX ; traffic lights.  MOV AL, 100b ; Turn on the third  OUT 7, AL ; magnet of the stepper-motor.   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| POP | REG SREG memory | Get 16 bit value from the stack.   Algorithm:   * operand = SS:[SP] (top of the stack) * SP = SP + 2   Example:  MOV AX, 1234h  PUSH AX  POP DX ; DX = 1234h  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| POPA | No operands | Pop all general purpose registers DI, SI, BP, SP, BX, DX, CX, AX from the stack. SP value is ignored, it is Popped but not set to SP register).  Note: this instruction works only on **80186** CPU and later!   Algorithm:   * POP DI * POP SI * POP BP * POP xx (SP value ignored) * POP BX * POP DX * POP CX * POP AX  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| POPF | No operands | Get flags register from the stack.   Algorithm:   * flags = SS:[SP] (top of the stack) * SP = SP + 2  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | popped | | | | | | |
| PUSH | REG SREG memory immediate | Store 16 bit value in the stack.  Note: **PUSH immediate** works only on 80186 CPU and later!   Algorithm:   * SP = SP - 2 * SS:[SP] (top of the stack) = operand   Example:  MOV AX, 1234h  PUSH AX  POP DX ; DX = 1234h  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| PUSHA | No operands | Push all general purpose registers AX, CX, DX, BX, SP, BP, SI, DI in the stack. Original value of SP register (before PUSHA) is used.  Note: this instruction works only on **80186** CPU and later!   Algorithm:   * PUSH AX * PUSH CX * PUSH DX * PUSH BX * PUSH SP * PUSH BP * PUSH SI * PUSH DI  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| PUSHF | No operands | Store flags register in the stack.   Algorithm:   * SP = SP - 2 * SS:[SP] (top of the stack) = flags  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| RCL | memory, immediate REG, immediate  memory, CL REG, CL | Rotate operand1 left through Carry Flag. The number of rotates is set by operand2.  When **immediate** is greater then 1, assembler generates several **RCL xx, 1** instructions because 8086 has machine code only for this instruction (the same principle works for all other shift/rotate instructions).   Algorithm:  shift all bits left, the bit that goes off is set to CF and previous value of CF is inserted to the right-most position.  Example:  STC ; set carry (CF=1).  MOV AL, 1Ch ; AL = 00011100b  RCL AL, 1 ; AL = 00111001b, CF=0.  RET   |  |  | | --- | --- | | C | O | | r | r |   OF=0 if first operand keeps original sign. |
| RCR | memory, immediate REG, immediate  memory, CL REG, CL | Rotate operand1 right through Carry Flag. The number of rotates is set by operand2.   Algorithm:  shift all bits right, the bit that goes off is set to CF and previous value of CF is inserted to the left-most position.  Example:  STC ; set carry (CF=1).  MOV AL, 1Ch ; AL = 00011100b  RCR AL, 1 ; AL = 10001110b, CF=0.  RET   |  |  | | --- | --- | | C | O | | r | r |   OF=0 if first operand keeps original sign. |
| REP | chain instruction | Repeat following MOVSB, MOVSW, LODSB, LODSW, STOSB, STOSW instructions CX times.   Algorithm:  check\_cx:  if CX <> 0 then   * do following chain instruction * CX = CX - 1 * go back to check\_cx   else   * exit from REP cycle  |  | | --- | | Z | | r | |
| REPE | chain instruction | Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 1 (result is Equal), maximum CX times.   Algorithm:  check\_cx:  if CX <> 0 then   * do following chain instruction * CX = CX - 1 * if ZF = 1 then:   + go back to check\_cx   else   * + exit from REPE cycle   else   * exit from REPE cycle   Example: see **cmpsb.asm** in Samples.   |  | | --- | | Z | | r | |
| REPNE | chain instruction | Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 0 (result is Not Equal), maximum CX times.   Algorithm:  check\_cx:  if CX <> 0 then   * do following chain instruction * CX = CX - 1 * if ZF = 0 then:   + go back to check\_cx   else   * + exit from REPNE cycle   else   * exit from REPNE cycle  |  | | --- | | Z | | r | |
| REPNZ | chain instruction | Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 0 (result is Not Zero), maximum CX times.   Algorithm:  check\_cx:  if CX <> 0 then   * do following chain instruction * CX = CX - 1 * if ZF = 0 then:   + go back to check\_cx   else   * + exit from REPNZ cycle   else   * exit from REPNZ cycle  |  | | --- | | Z | | r | |
| REPZ | chain instruction | Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 1 (result is Zero), maximum CX times.   Algorithm:  check\_cx:  if CX <> 0 then   * do following chain instruction * CX = CX - 1 * if ZF = 1 then:   + go back to check\_cx   else   * + exit from REPZ cycle   else   * exit from REPZ cycle  |  | | --- | | Z | | r | |
| RET | No operands or even immediate | Return from near procedure.   Algorithm:   * Pop from stack:   + IP * if immediate operand is present: SP = SP + operand   Example:  #make\_COM#  ORG 100h ; for COM file.  CALL p1  ADD AX, 1  RET ; return to OS.  p1 PROC ; procedure declaration.  MOV AX, 1234h  RET ; return to caller.  p1 ENDP   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| RETF | No operands or even immediate | Return from Far procedure.   Algorithm:   * Pop from stack:   + IP   + CS * if immediate operand is present: SP = SP + operand  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| ROL | memory, immediate REG, immediate  memory, CL REG, CL | Rotate operand1 left. The number of rotates is set by operand2.   Algorithm:  shift all bits left, the bit that goes off is set to CF and the same bit is inserted to the right-most position.  Example:  MOV AL, 1Ch ; AL = 00011100b  ROL AL, 1 ; AL = 00111000b, CF=0.  RET   |  |  | | --- | --- | | C | O | | r | r |   OF=0 if first operand keeps original sign. |
| ROR | memory, immediate REG, immediate  memory, CL REG, CL | Rotate operand1 right. The number of rotates is set by operand2.   Algorithm:  shift all bits right, the bit that goes off is set to CF and the same bit is inserted to the left-most position.  Example:  MOV AL, 1Ch ; AL = 00011100b  ROR AL, 1 ; AL = 00001110b, CF=0.  RET   |  |  | | --- | --- | | C | O | | r | r |   OF=0 if first operand keeps original sign. |
| SAHF | No operands | Store AH register into low 8 bits of Flags register.   Algorithm:  flags register = AH  AH bit: 7 6 5 4 3 2 1 0  [SF] [ZF] [0] [AF] [0] [PF] [1] [CF]  bits 1, 3, 5 are reserved.   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | r | r | r | r | r | r | |
| SAL | memory, immediate REG, immediate  memory, CL REG, CL | Shift Arithmetic operand1 Left. The number of shifts is set by operand2.   Algorithm:   * Shift all bits left, the bit that goes off is set to CF. * Zero bit is inserted to the right-most position.   Example:  MOV AL, 0E0h ; AL = 11100000b  SAL AL, 1 ; AL = 11000000b, CF=1.  RET   |  |  | | --- | --- | | C | O | | r | r |   OF=0 if first operand keeps original sign. |
| SAR | memory, immediate REG, immediate  memory, CL REG, CL | Shift Arithmetic operand1 Right. The number of shifts is set by operand2.   Algorithm:   * Shift all bits right, the bit that goes off is set to CF. * The sign bit that is inserted to the left-most position has the same value as before shift.   Example:  MOV AL, 0E0h ; AL = 11100000b  SAR AL, 1 ; AL = 11110000b, CF=0.  MOV BL, 4Ch ; BL = 01001100b  SAR BL, 1 ; BL = 00100110b, CF=0.  RET   |  |  | | --- | --- | | C | O | | r | r |   OF=0 if first operand keeps original sign. |
| SBB | REG, memory memory, REG REG, REG memory, immediate REG, immediate | Subtract with Borrow.   Algorithm:  operand1 = operand1 - operand2 - CF   Example:  STC  MOV AL, 5  SBB AL, 3 ; AL = 5 - 3 - 1 = 1  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | r | r | r | r | r | r | |
| SCASB | No operands | Compare bytes: AL from ES:[DI].   Algorithm:   * ES:[DI] - AL * set flags according to result: OF, SF, ZF, AF, PF, CF * if DF = 0 then   + DI = DI + 1   else   * + DI = DI - 1  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | r | r | r | r | r | r | |
| SCASW | No operands | Compare words: AX from ES:[DI].   Algorithm:   * ES:[DI] - AX * set flags according to result: OF, SF, ZF, AF, PF, CF * if DF = 0 then   + DI = DI + 2   else   * + DI = DI - 2  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | r | r | r | r | r | r | |
| SHL | memory, immediate REG, immediate  memory, CL REG, CL | Shift operand1 Left. The number of shifts is set by operand2.   Algorithm:   * Shift all bits left, the bit that goes off is set to CF. * Zero bit is inserted to the right-most position.   Example:  MOV AL, 11100000b  SHL AL, 1 ; AL = 11000000b, CF=1.  RET   |  |  | | --- | --- | | C | O | | r | r |   OF=0 if first operand keeps original sign. |
| SHR | memory, immediate REG, immediate  memory, CL REG, CL | Shift operand1 Right. The number of shifts is set by operand2.   Algorithm:   * Shift all bits right, the bit that goes off is set to CF. * Zero bit is inserted to the left-most position.   Example:  MOV AL, 00000111b  SHR AL, 1 ; AL = 00000011b, CF=1.  RET   |  |  | | --- | --- | | C | O | | r | r |   OF=0 if first operand keeps original sign. |
| STC | No operands | Set Carry flag.   Algorithm:   CF = 1   |  | | --- | | C | | 1 | |
| STD | No operands | Set Direction flag. SI and DI will be decremented by chain instructions: CMPSB, CMPSW, LODSB, LODSW, MOVSB, MOVSW, STOSB, STOSW.   Algorithm:   DF = 1   |  | | --- | | D | | 1 | |
| STI | No operands | Set Interrupt enable flag. This enables hardware interrupts.   Algorithm:   IF = 1   |  | | --- | | I | | 1 | |
| STOSB | No operands | Store byte in AL into ES:[DI]. Update DI.  Algorithm:   * ES:[DI] = AL * if DF = 0 then   + DI = DI + 1   else   * + DI = DI - 1   Example:  #make\_COM#  ORG 100h  LEA DI, a1  MOV AL, 12h  MOV CX, 5  REP STOSB  RET  a1 DB 5 dup(0)   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| STOSW | No operands | Store word in AX into ES:[DI]. Update DI.  Algorithm:   * ES:[DI] = AX * if DF = 0 then   + DI = DI + 2   else   * + DI = DI - 2   Example:  #make\_COM#  ORG 100h  LEA DI, a1  MOV AX, 1234h  MOV CX, 5  REP STOSW  RET  a1 DW 5 dup(0)   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| SUB | REG, memory memory, REG REG, REG memory, immediate REG, immediate | Subtract.   Algorithm:  operand1 = operand1 - operand2   Example:  MOV AL, 5  SUB AL, 1 ; AL = 4  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | r | r | r | r | r | r | |
| TEST | REG, memory memory, REG REG, REG memory, immediate REG, immediate | Logical AND between all bits of two operands for flags only. These flags are effected: **ZF, SF, PF.** Result is not stored anywhere.  These rules apply:  1 AND 1 = 1 1 AND 0 = 0 0 AND 1 = 0 0 AND 0 = 0   Example:  MOV AL, 00000101b  TEST AL, 1 ; ZF = 0.  TEST AL, 10b ; ZF = 1.  RET   |  |  |  |  |  | | --- | --- | --- | --- | --- | | C | Z | S | O | P | | 0 | r | r | 0 | r | |
| XCHG | REG, memory memory, REG REG, REG | Exchange values of two operands.   Algorithm:  operand1 < - > operand2   Example:  MOV AL, 5  MOV AH, 2  XCHG AL, AH ; AL = 2, AH = 5  XCHG AL, AH ; AL = 5, AH = 2  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| XLATB | No operands | Translate byte from table. Copy value of memory byte at DS:[BX + unsigned AL] to AL register.   Algorithm:  AL = DS:[BX + unsigned AL]   Example:  #make\_COM#  ORG 100h  LEA BX, dat  MOV AL, 2  XLATB ; AL = 33h  RET  dat DB 11h, 22h, 33h, 44h, 55h   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | unchanged | | | | | | |
| XOR | REG, memory memory, REG REG, REG memory, immediate REG, immediate | Logical XOR (Exclusive OR) between all bits of two operands. Result is stored in first operand.  These rules apply:  1 XOR 1 = 0 1 XOR 0 = 1 0 XOR 1 = 1 0 XOR 0 = 0   Example:  MOV AL, 00000111b  XOR AL, 00000010b ; AL = 00000101b  RET   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | C | Z | S | O | P | A | | 0 | r | r | 0 | r | ? | |