

8086 Assembler

Pdsilva 2410.

Index

Introduction to 8086's assembler.....	3
Inside the CPU.....	4
Memory Access.....	7
Variables.....	13
Arrays.....	17
Getting the Address of a Variable.....	19
Constants.....	22
Interrupts.....	24
Library of common functions - emu8086.inc.....	27
Arithmetic and Logic Instructions.....	31
Program flow control.....	36
Procedures.....	47
The Stack.....	52
Macros.....	57
Controlling External Devices.....	61
Appendix A.....	66
Appendix B.....	139

Introduction to 8086's assembler

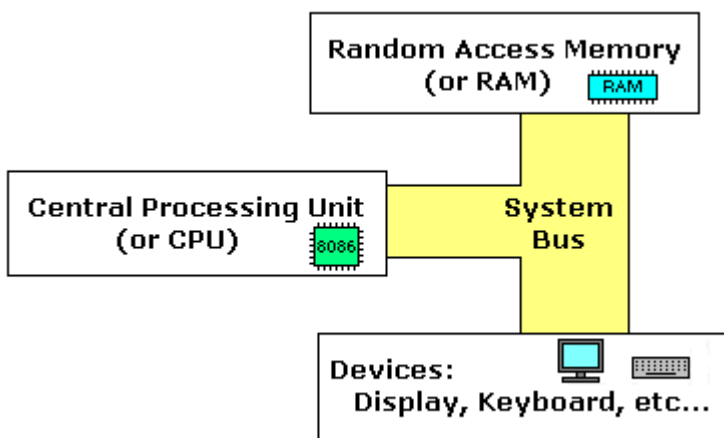
This tutorial is intended for those who are not familiar with assembler at all, or have a very distant idea about it. Of course if you have knowledge of some high level programming language (java, basic, c/c++, pascal...) that may help you a lot.

But even if you are familiar with assembler, it is still a good idea to look through this document in order to study emu8086 syntax.

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

what is assembly language?

Assembly language is a low level programming language. You need to get some knowledge about computer structure in order to understand anything. The simple computer model as I see it:

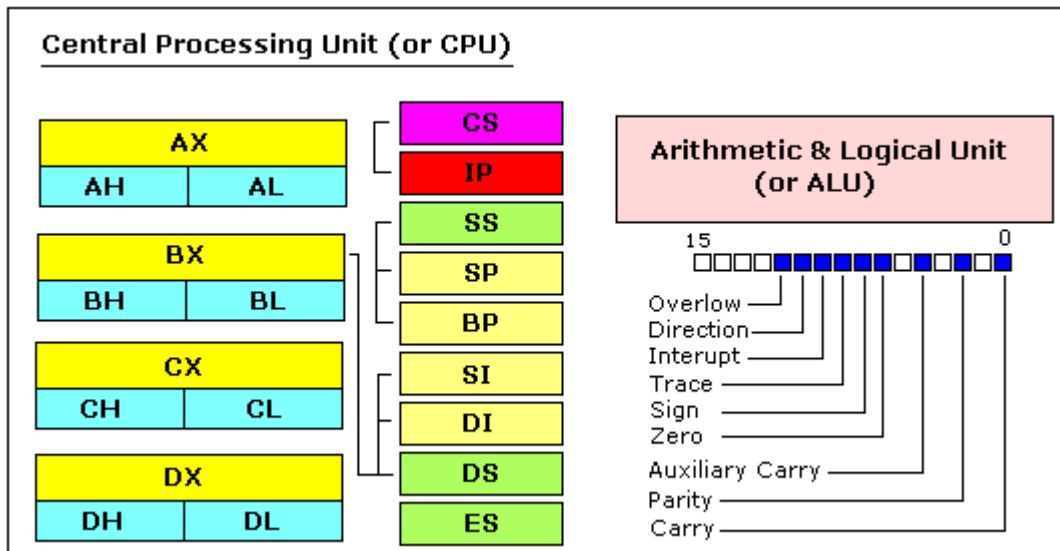


The system bus (shown in yellow) connects the various components of a computer. The CPU is the heart of the computer, most of computations occur inside the CPU. RAM is a place to where the programs are loaded in

8086 assembler tutorial for beginners

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

8086 assembler tutorial for beginners

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 =

8086 assembler tutorial for beginners

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

$$\begin{array}{r} 12300 \\ + 0045 \\ \hline 12345 \end{array}$$

The address formed with 2 registers is called an effective address.

By default BX, SI and DI registers work with DS segment register;

BP and SP work with SS segment register.

Other general purpose registers cannot form an effective address!

also, although BX can form an effective address, BH and BL cannot.

special purpose registers

- **IP** - the instruction pointer.

- **flags register** - determines the current state of the microprocessor.

IP register always works together with CS segment register and it points to currently executing instruction.

Flags register is modified automatically by CPU after mathematical

operations, this allows to determine the type of the result, and to determine conditions to transfer control to other parts of the program.

Generally you cannot access these registers directly, the way you can access AX and other general registers, but it is possible to change values of system registers using some tricks that you will learn a little bit later.

Memory Access

To access memory we can use these four registers: BX, SI, DI, BP.

Combining these registers inside [] symbols, we can get different memory locations.

These combinations are supported (addressing modes):

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

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

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

Displacement can be a immediate value or offset of a variable, or even

8086 assembler tutorial for beginners

both. If there are several values, assembler evaluates all values and calculates a single immediate value.

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

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

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

For example, let's assume that DS = 100, BX = 30, SI = 70.

The following addressing mode: [BX + SI] + 25

Is calculated by processor to this physical address: $100 * 16 + 30 + 70 + 25 = 1725$.

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

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

BX	SI	+ disp
BP	DI	

All valid combinations can be formed by taking only one item from each column or skipping the column by not taking anything from it. BX and BP never go together. Neither SI and DI do.

Examples of valid addressing modes:

[BX+5]

8086 assembler tutorial for beginners

[BX+SI]

[DI+BX-4]

The value in segment register (CS, DS, SS, ES) is called a segment, and the value in general purpose register (BX, SI, DI, BP) is called an offset.

When DS contains value 1234h and SI contains the value 7890h it can be also recorded as 1234:7890. The physical address will be $1234h * 10h + 7890h = 19BD0h$.

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

7h = 7

70h = 112

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

byte ptr - for byte.

word ptr - for word (two bytes).

for example:

byte ptr [BX] ; byte access.

8086 assembler tutorial for beginners

or

word ptr [BX] ; word access.

Emu Assembler supports shorter prefixes as well:

b. - for byte ptr

w. - for word ptr

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

MOV instruction

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

these types of operands are supported:

MOV REG, memory

MOV memory, REG

MOV REG, REG

MOV memory, immediate

MOV REG, immediate

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

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

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

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

MOV SREG, memory

8086 assembler tutorial for beginners

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

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

A short program that demonstrates the use of **MOV** instruction:

ORG 100h ; this directive required for a simple 1 segment .com program.

MOV AX, 0B800h ; set AX to hexadecimal value of B800h.

MOV DS, AX ; copy value of AX to DS.

MOV CL, 'A' ; set CL to ASCII code of 'A', it is 41h.

MOV CH, 1101_1111b ; set CH to binary value.

MOV BX, 15Eh ; set BX to 15Eh.

MOV [BX], CX ; copy contents of CX to memory at B800:015E

RET ; returns to operating system.

Once the above program is typed into the code editor, and [Compile and Emulate] button is pressed (F5 key)

The emulator window should open with this program loaded, clicking [Single Step] button should change the register values.

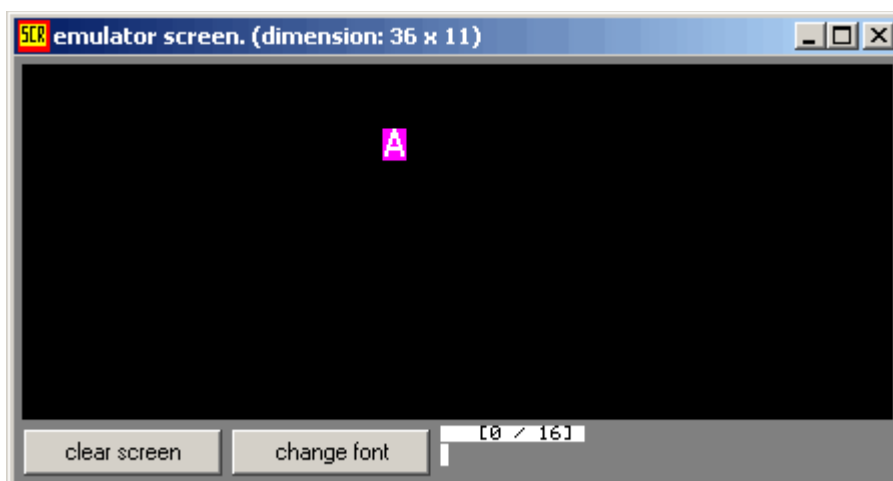
8086 assembler tutorial for beginners

How to do copy & paste:

1. Select the above text using mouse, click before the text and drag it down until everything is selected.
2. Press **Ctrl + C** combination to copy.
3. Click inside the source code editor and press **Ctrl + V** combination to paste.

";" is used for comments, anything after ";" symbol is ignored.

The result of a working program:



Actually the above program writes directly to video memory, now we can 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" then at the address 5A73:235B, especially when you have 10 or more variables.

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

Syntax for a variable declaration:

name **DB** value

name **DW** value

DB - stays for Define Byte.

DW - stays for Define Word.

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

8086 assembler tutorial for beginners

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

As you probably know from *part 2* of this tutorial, MOV instruction is used to copy values from source to destination.

Let's see another example with MOV instruction:

```
ORG 100h

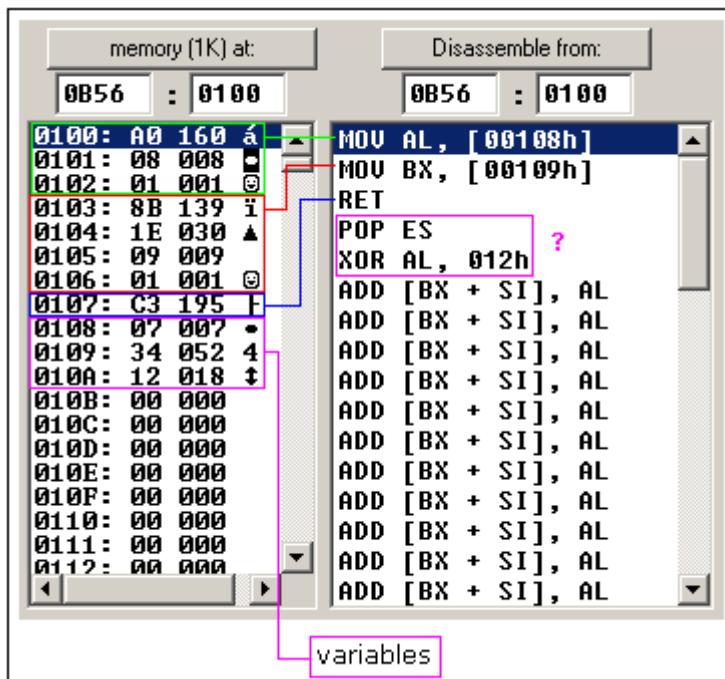
MOV AL, var1
MOV BX, var2

RET ; stops the program.

VAR1 DB 7
var2 DW 1234h
```

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

8086 assembler tutorial for beginners



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.

8086 assembler tutorial for beginners

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

You can even write the same program using DB directive only:

```
ORG 100h
```

```
DB 0A0h
```

```
DB 08h
```

```
DB 01h
```

```
DB 8Bh
```

```
DB 1Eh
```

```
DB 09h
```

```
DB 01h
```

```
DB 0C3h
```

```
DB 7
```

```
DB 34h
```

```
DB 12h
```

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

As you may guess, the compiler just converts the program source to the

8086 assembler tutorial for beginners

set of bytes, this set is called machine code, processor understands the machine code and executes it.

ORG 100h is a compiler directive (it tells compiler how to handle the source code). This directive is very important when you work with variables. It tells compiler that the executable file will be loaded at the offset of 100h (256 bytes), so compiler should calculate the correct address for all variables when it replaces the variable names with their offsets. Directives are never converted to any real machine code.

Why executable file is loaded at offset of 100h? Operating system keeps some data about the program in the first 256 bytes of the CS (code segment), such as command line parameters and etc.

Though this is true for COM files only, EXE files are loaded at offset of 0000, and generally use special segment for variables. Maybe we'll talk more about EXE files later.

Arrays

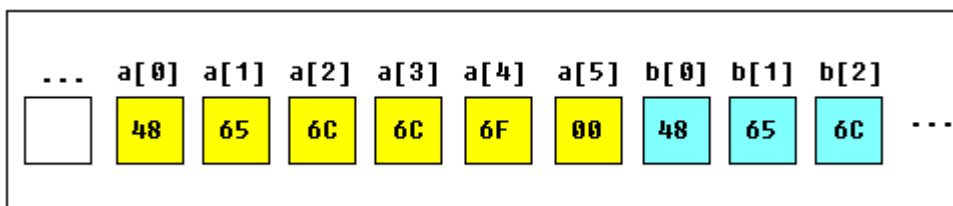
Arrays can be seen as chains of variables. A text string is an example of a byte array, each character is presented as an ASCII code value (0..255).

Here are some array definition examples:

8086 assembler tutorial for beginners

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

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



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

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

You can also use any of the memory index registers BX, SI, DI, BP, for example:

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

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

number DUP (value(s))

number - number of duplicate to make (any constant value).

value - expression that DUP will duplicate.

for example:

```
c DB 5 DUP(9)
```

is an alternative way of declaring:

```
c DB 9, 9, 9, 9, 9
```

one more example:

```
d DB 5 DUP(1, 2)
```

is an alternative way of declaring:

```
d DB 1, 2, 1, 2, 1, 2, 1, 2, 1, 2
```

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

Getting the Address of a Variable

There is LEA (Load Effective Address) instruction and

8086 assembler tutorial for beginners

alternative OFFSET operator. Both OFFSET and LEA can be used to get the offset address of the variable.

LEA is more powerful because it also allows you to get the address of an indexed variables. Getting the address of the variable can be very useful in some situations, for example when you need to pass parameters to a procedure.

Reminder:

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

BYTE PTR - for byte.

WORD PTR - for word (two bytes).

For example:

BYTE PTR [BX] ; byte access.

or

WORD PTR [BX] ; word access.

assembler supports shorter prefixes as well:

b. - for **BYTE PTR**

w. - for **WORD PTR**

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

Here is first example:

```
ORG 100h
```

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

8086 assembler tutorial for beginners

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

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

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

RET

VAR1 DB 22h

END
```

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

```
ORG 100h

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

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

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

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

RET

VAR1 DB 22h
```

8086 assembler tutorial for beginners

```
END
```

Both examples have the same functionality.

These lines:

```
LEA BX, VAR1
```

```
MOV BX, OFFSET VAR1
```

are even compiled into the same machine code: `MOV BX, num`
num is a 16 bit value of the variable offset.

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

Constants

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

name EQU < any expression >

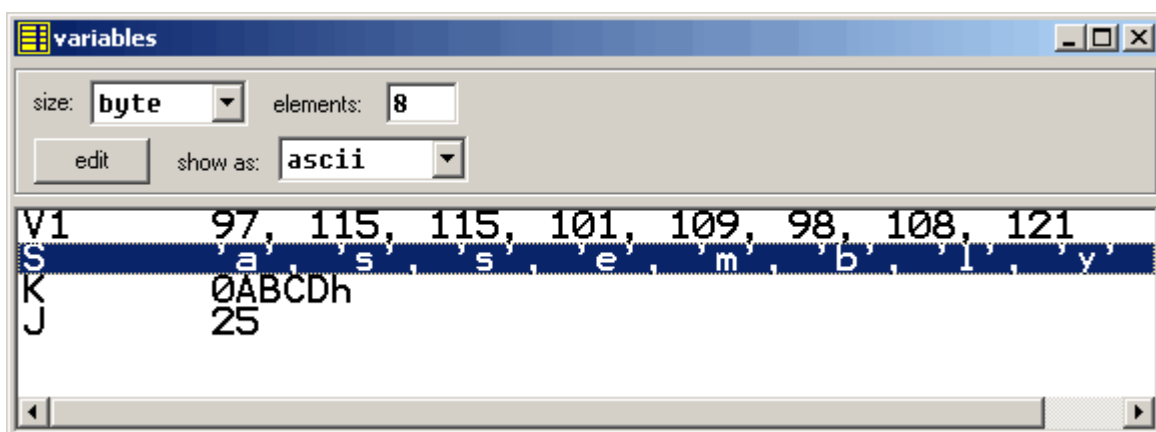
For example:

```
k EQU 5  
  
MOV AX, k
```

The above example is functionally identical to code:

```
MOV AX, 5
```

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



8086 assembler tutorial for beginners

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.

8086 assembler tutorial for beginners

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

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

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

```
MOV AH, 0Eh ; select sub-function.
```

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

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

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

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

```
MOV AL, 'l' ; ASCII code: 108
```

8086 assembler tutorial for beginners

```
INT 10h    ; print it!

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

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

RET        ; returns to operating system.
```

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

See [list of supported interrupts](#) for more information about interrupts.

Library of common functions - emu8086.inc

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

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

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

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

emu8086.inc defines the following macros:

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

8086 assembler tutorial for beginners

- CURSORON** - turns on the text cursor.

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

```
include emu8086.inc

ORG 100h

PRINT 'Hello World!'

GOTOXY 10, 5

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

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

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

emu8086.inc also defines the following procedures:

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

8086 assembler tutorial for beginners

•**PTHIS** - procedure to print a null terminated string at current cursor position (just as PRINT_STRING), but receives address of string from Stack. The ZERO TERMINATED string should be defined just after the CALL instruction. For example:

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

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

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

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

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

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

•**PRINT_NUM_UN**s - procedure that prints out an unsigned number in **AX** register. To use it declare: **DEFINE_PRINT_NUM_UN**s before **END** directive.

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

```
include 'emu8086.inc'

ORG 100h

LEA SI, msg1 ; ask for the number
CALL print_string ;
```

8086 assembler tutorial for beginners

```
CALL  scan_num    ; get number in CX.

MOV   AX, CX      ; copy the number to AX.

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

CALL  print_num    ; print number in AX.

RET                ; return to operating system.

msg1  DB 'Enter the number: ', 0

DEFINE_SCAN_NUM
DEFINE_PRINT_STRING
DEFINE_PRINT_NUM
DEFINE_PRINT_NUM_UNSP ; required for print_num.
DEFINE_PTHIS

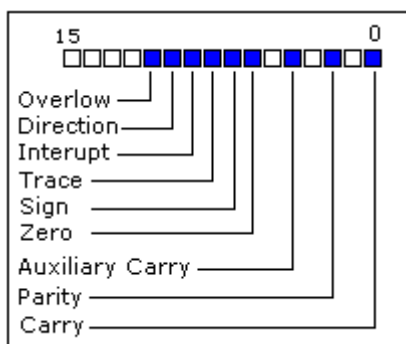
END                ; directive to stop the compiler.
```

First compiler processes the declarations (these are just regular the macros that are expanded to procedures). When compiler gets to CALL instruction it replaces the procedure name with the address of the code where the procedure is declared. When CALL instruction is executed control is transferred to procedure. This is quite useful, since even if you call the same procedure 100 times in your code you will still

have relatively small executable size. Seems complicated, isn't it? That's ok, with the time you will learn more, currently it's required that you understand the basic principle.

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).

8086 assembler tutorial for beginners

- **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 instructions are used to make decisions during program execution).

8086 assembler tutorial for beginners

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.

8086 assembler tutorial for beginners

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**:

$AX = AL * \text{operand}$.

when operand is a **word**:

$(DX\ AX) = AX * \text{operand}$.

•**IMUL** - Signed multiply:

when operand is a **byte**:

$AX = AL * \text{operand}$.

when operand is a **word**:

$(DX\ AX) = AX * \text{operand}$.

•**DIV** - Unsigned divide:

when operand is a **byte**:

$AL = AX / \text{operand}$

$AH = \text{remainder (modulus)}..$

when operand is a **word**:

$AX = (DX\ AX) / \text{operand}$

$DX = \text{remainder (modulus)}..$

•**IDIV** - Signed divide:

when operand is a **byte**:

$AL = AX / \text{operand}$

$AH = \text{remainder (modulus)}..$

when operand is a **word**:

$AX = (DX\ AX) / \text{operand}$

$DX = \text{remainder (modulus)}..$

8086 assembler tutorial for beginners

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

8086 assembler tutorial for beginners

x2: MOV AX, 2

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

org 100h

mov ax, 5 ; set ax to 5.

mov bx, 2 ; set bx to 2.

jmp calc ; go to 'calc'.

back: jmp stop ; go to 'stop'.

calc:

add ax, bx ; add bx to ax.

jmp back ; go 'back'.

stop:

ret ; return to operating system.

Of course there is an easier way to calculate the some of two numbers, but it's still a good example of **JMP** instruction.

As you can see from this example **JMP** is able to transfer control both forward and backward. It can jump anywhere in current code segment (65,535 bytes).

•Short Conditional Jumps

Unlike **JMP** instruction that does an unconditional jump, there are instructions that do a conditional jumps (jump only when some conditions are in act). These instructions are

8086 assembler tutorial for beginners

divided in three groups, first group just test single flag, second compares numbers as signed, and third compares numbers as unsigned.

Jump instructions that test single flag

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

as you may already notice there are some instructions that do that same thing, that's correct, they even are assembled into the same machine code, so it's good to remember that when you compile **JE** instruction - you will get it disassembled as: **JZ**, **JC** is assembled the same as **JB** etc...

different names are used to make programs easier to understand, to code and most importantly to remember. very offset dissembler has no clue what the original instruction was look like that's why it uses the most common name.

if you emulate this code you will see that all instructions are assembled into **JNB**, the

8086 assembler tutorial for beginners

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

`jnc a`

`jnb a`

`jae a`

`mov ax, 4`

a: `mov ax, 5`

`ret`

Jump instructions for signed numbers

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

8086 assembler tutorial for beginners

	Jump if Not Greater (not >).	or SF <> OF	
--	--------------------------------------	----------------	--

<> - 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:

8086 assembler tutorial for beginners

it's required to compare 5 and 2,
 $5 - 2 = 3$
the result is not zero (Zero Flag is set to 0).

Another example:
it's required to compare 7 and 7,
 $7 - 7 = 0$
the result is zero! (Zero Flag is set to 1 and **JZ** or **JE** will do the jump).

here's an example of **CMP** instruction and conditional jump:

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

```
org 100h
```

```
mov al, 25 ; set al to 25.
```

```
mov bl, 10 ; set bl to 10.
```

```
cmp al, bl ; compare al - bl.
```

```
je equal ; jump if al = bl (zf = 1).
```

```
putc 'n' ; if it gets here, then al <> bl,
```

```
jmp stop ; so print 'n', and jump to stop.
```

```
equal: ; if gets here,
```

```
putc 'y' ; then al = bl, so print 'y'.
```

```
stop:
```

```
ret ; gets here no matter what.
```

8086 assembler tutorial for beginners

try the above example with different numbers for **AL** and **BL**, open flags by clicking on flags button, use single step and see what happens. you can use **F5** hotkey to recompile and reload the program into the emulator.

loops

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

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

```
org 100h
```

```
mov bx, 0 ; total step counter.
```

```
mov cx, 5
```

8086 assembler tutorial for beginners

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

ret
```

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

just like all other conditional jumps loops have an opposite companion that can help to create workarounds, when the address

8086 assembler tutorial for beginners

of desired location is too far assemble automatically assembles reverse and long jump instruction, making total of 5 bytes instead of just 2, it can be seen in disassembler as well.

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

All conditional jumps have one big limitation, unlike **JMP** instruction they can only jump **127** bytes forward and **128** bytes backward (note that most instructions are assembled into 3 or more bytes).

We can easily avoid this limitation using a cute trick:

- Get an opposite conditional jump instruction from the table above, make it jump to *label_x*.
- Use **JMP** instruction to jump to desired location.
- Define *label_x*: just after the **JMP** instruction.

label_x: - can be any valid label name, but there must not be two or more labels with the same name.

here's an example:

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

```
org 100h
```

```
mov al, 5
```

```
mov bl, 5
```

```
cmp al, bl ; compare al - bl.
```

8086 assembler tutorial for beginners

; je equal ; there is only 1 byte

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

jmp equal

not_equal:

add bl, al

sub al, 10

xor al, bl

jmp skip_data

db 256 dup(0) ; 256 bytes

skip_data:

putc 'n' ; if it gets here, then al <> bl,

jmp stop ; so print 'n', and jump to stop.

equal: ; if gets here,

putc 'y' ; then al = bl, so print 'y'.

stop:

ret

8086 assembler tutorial for beginners

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

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

```
org 100h

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

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

```
ret
```

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.

8086 assembler tutorial for beginners

The syntax for procedure declaration:

```
name PROC  
  
    ; here goes the code  
    ; of the procedure ...
```

```
RET  
name ENDP
```

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

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

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

CALL instruction is used to call a procedure.

Here is an example:

```
ORG 100h  
  
CALL m1  
  
MOV AX, 2  
  
RET ; return to operating system.
```

8086 assembler tutorial for beginners

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

END
```

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

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

```
ORG  100h

MOV  AL, 1
MOV  BL, 2

CALL m2
CALL m2
CALL m2
CALL m2

RET      ; return to operating system.
```

8086 assembler tutorial for beginners

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

END
```

In the above example value of AL register is update every time the procedure is called, BL register stays unchanged, so this algorithm calculates 2 in power of 4, so final result in AX register is 16 (or 10h).

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

```
ORG  100h

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

CALL print_me

RET      ; return to operating system.

;
=====
===
```

8086 assembler tutorial for beginners

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

```
print_me  PROC
```

```
next_char:
```

```
    CMP  b.[SI], 0    ; check for zero to stop
```

```
    JE   stop        ;
```

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

```
    MOV  AH, 0Eh      ; teletype function number.
```

```
    INT  10h          ; using interrupt to print a char in AL.
```

```
    ADD  SI, 1        ; advance index of string array.
```

```
    JMP  next_char    ; go back, and type another char.
```

```
stop:
```

```
    RET              ; return to caller.
```

```
print_me  ENDP
```

```
;
```

```
=====
===
```

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

```
END
```

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

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

8086 assembler tutorial for beginners

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...

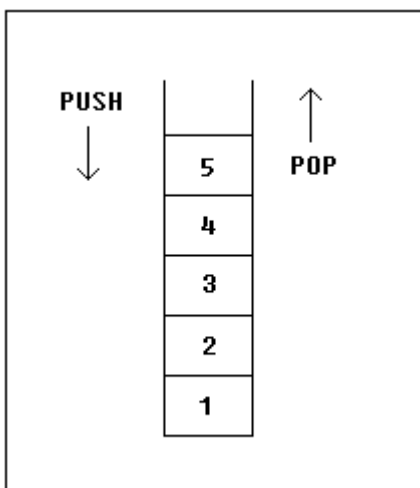
Notes:

- **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 PUSHs and POPs, 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

8086 assembler tutorial for beginners

to operating system, so when program starts there is a return address in stack (generally it's 0000h).

PUSH and POP instruction are especially useful because we don't have too much registers to operate with, so here is a trick:

- Store original value of the register in stack (using **PUSH**).
- Use the register for any purpose.
- Restore the original value of the register from stack (using **POP**).

Here is an example:

```
ORG 100h

MOV AX, 1234h
PUSH AX      ; store value of AX in stack.

MOV AX, 5678h ; modify the AX value.

POP AX       ; restore the original value of AX.

RET

END
```

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



8086 assembler tutorial for beginners

```
ORG 100h

MOV AX, 1212h ; store 1212h in AX.
MOV BX, 3434h ; store 3434h in BX

PUSH AX      ; store value of AX in stack.
PUSH BX      ; store value of BX in stack.

POP AX       ; set AX to original value of BX.
POP BX       ; set BX to original value of AX.

RET

END
```

The exchange happens because stack uses LIFO (Last In First Out) algorithm, so when we push 1212h and then 3434h, on pop we will first get 3434h and only after it 1212h.

The stack memory area is set by SS (Stack Segment) register, and SP (Stack Pointer) register. Generally operating system sets values of these registers on program start.

"PUSH *source*" instruction does the following:

- Subtract 2 from **SP** register.
- Write the value of *source* to the address **SS:SP**.

8086 assembler tutorial for beginners

"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 0FFFFEh. At the address SS:0FFFFEh 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. [emu8086.inc](#) is a good example of how macros can be used, this file contains several macros to make coding easier for you.

Macro definition:

```
name  MACRO [parameters,...]
```

```
    <instructions>
```

```
ENDM
```

Unlike procedures, macros should be defined above the code that uses it, for example:

```
MyMacro  MACRO p1, p2, p3
```

```
    MOV AX, p1
```

8086 assembler tutorial for beginners

```
MOV BX, p2
MOV CX, p3

ENDM

ORG 100h

MyMacro 1, 2, 3

MyMacro 4, 5, DX

RET
```

The above code is expanded into:

```
MOV AX, 00001h
MOV BX, 00002h
MOV CX, 00003h
MOV AX, 00004h
MOV BX, 00005h
MOV CX, DX
```

Some important facts about **macros** and **procedures**:

- When you want to use a procedure you should use **CALL** instruction, for example:
CALL MyProc
- When you want to use a macro, you can just type its name. For example:
MyMacro

8086 assembler tutorial for beginners

- 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
```

8086 assembler tutorial for beginners

```
label1:
    INC AX
label2:
    ADD AX, 2
ENDM

ORG 100h

MyMacro2

MyMacro2

RET
```

If you plan to use your macros in several programs, it may be a good idea to place all macros in a separate file. Place that file in Inc folder and use `INCLUDE file-name` directive to use macros. See [Library of common functions - emu8086.inc](#) for an example of such file.

Controlling External Devices

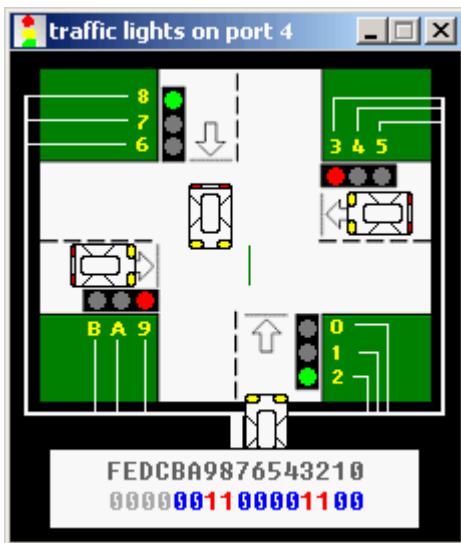
There are 7 devices attached to the emulator: traffic lights, stepper-motor, LED display, thermometer, printer, robot and simple test device. You can view devices when you click "Virtual Devices" menu of the emulator.

For technical information see [I/O ports](#) section of emu8086 reference.

In general, it is possible to use any x86 family CPU to control all kind of devices, the difference maybe in base I/O port number, this can be altered using some tricky electronic equipment. Usually the ".bin" file is written into the Read Only Memory (ROM) chip, the system reads program from that chip, loads it in RAM module and runs the program. This principle is used for many modern devices such as micro-wave ovens and etc...

Traffic Lights

8086 assembler tutorial for beginners



Usually to control the traffic lights an array (table) of values is used. In certain periods of time the value is read from the array and sent to a port. For example:

```
; controlling external device with 8086 microprocessor.  
; realistic test for c:\emu8086\devices\Traffic_Lights.exe
```

```
#start=Traffic_Lights.exe#
```

```
name "traffic"
```

```
mov ax, all_red
```

```
out 4, ax
```

```
mov si, offset situation
```


8086 assembler tutorial for beginners

next:

mov ax, [si]

out 4, ax

; wait 5 seconds (5 million microseconds)

mov cx, 4Ch ; 004C4B40h = 5,000,000

mov dx, 4B40h

mov ah, 86h

int 15h

add si, 2 ; next situation

cmp si, sit_end

jb next

mov si, offset situation

jmp next

; FEDC_BA98_7654_3210

situation dw 0000_0011_0000_1100b

s1 dw 0000_0110_1001_1010b

s2 dw 0000_1000_0110_0001b

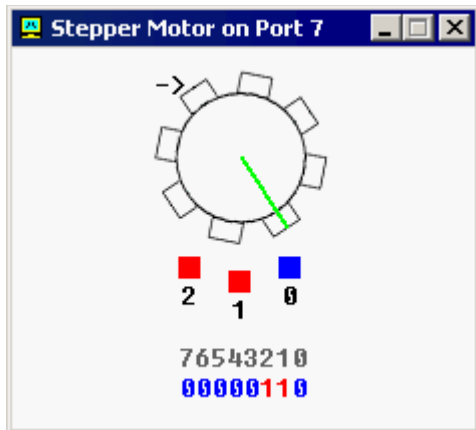
s3 dw 0000_1000_0110_0001b

s4 dw 0000_0100_1101_0011b

sit_end = \$

all_red equ 0000_0010_0100_1001b

Stepper-Motor



The motor can be half stepped by turning on pair of magnets, followed by a single and so on.

The motor can be full stepped by turning on pair of magnets, followed by another pair of magnets and in the end followed by a single magnet and so on. The best way to make full step is to make two half steps.

Half step is equal to 11.25 degrees.

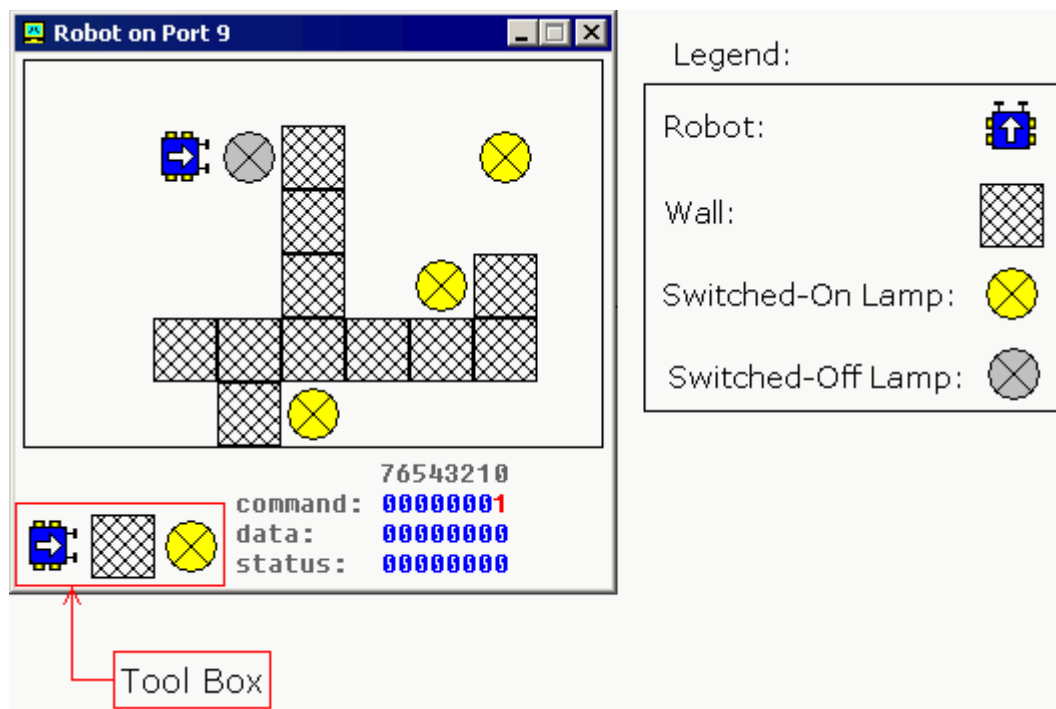
Full step is equal to 22.5 degrees.

The motor can be turned both clock-wise and counter-clock-wise.

open stepper_motor.asm from c:\emu8086\examples

See also [I/O ports](#) section of emu8086 reference.

Robot



Complete list of robot instruction set is given in [I/O ports](#) section of emu8086 reference.

To control the robot a complex algorithm should be used to achieve maximum efficiency. The simplest, yet very inefficient, is random moving algorithm, open robot.asm from c:\emu8086\examples

It is also possible to use a data table (just like for Traffic Lights), this can be good if robot always works in the same surroundings.

Appendix A

Complete 8086 instruction set

Quick reference:

<u>AAA</u>	<u>CMPSB</u>	<u>JAE</u>	<u>JNBE</u>	<u>JPO</u>	<u>MOV</u>	<u>RCR</u>	<u>SCASB</u>
<u>AAD</u>	<u>CMPSW</u>	<u>JB</u>	<u>JNC</u>	<u>JS</u>	<u>MOVSb</u>	<u>REP</u>	<u>SCASW</u>
<u>AAM</u>	<u>CWD</u>	<u>JBE</u>	<u>JNE</u>	<u>JZ</u>	<u>MOVSW</u>	<u>REPE</u>	<u>SHL</u>
<u>AAS</u>	<u>DAA</u>	<u>JC</u>	<u>JNG</u>	<u>LAHF</u>	<u>MUL</u>	<u>REPNE</u>	<u>SHR</u>
<u>ADC</u>	<u>DAS</u>	<u>JCXZ</u>	<u>JNGE</u>	<u>LDS</u>	<u>NEG</u>	<u>REPZ</u>	<u>STC</u>
<u>ADD</u>	<u>DEC</u>	<u>JE</u>	<u>JNL</u>	<u>LEA</u>	<u>NOP</u>	<u>REPZ</u>	<u>STD</u>
<u>AND</u>	<u>DIV</u>	<u>JG</u>	<u>JNLE</u>	<u>LES</u>	<u>NOT</u>	<u>RET</u>	<u>STI</u>
<u>CALL</u>	<u>HLT</u>	<u>JGE</u>	<u>JNO</u>	<u>LODSB</u>	<u>OR</u>	<u>RETF</u>	<u>STOSB</u>
<u>CBW</u>	<u>IDIV</u>	<u>JL</u>	<u>JNP</u>	<u>LODSW</u>	<u>OUT</u>	<u>ROL</u>	<u>STOSW</u>

8086 assembler tutorial for beginners

	<u>IMUL</u>				<u>POP</u>		
<u>CLC</u>	<u>IN</u>	<u>JLE</u>	<u>JNS</u>	<u>LOOP</u>	<u>POPA</u>	<u>ROR</u>	<u>SUB</u>
<u>CLD</u>	<u>INC</u>	<u>JMP</u>	<u>JNZ</u>	<u>LOOPE</u>	<u>POPF</u>	<u>SAHF</u>	<u>TEST</u>
<u>CLI</u>	<u>INT</u>	<u>JNA</u>	<u>JO</u>	<u>LOOPNE</u>	<u>PUSH</u>	<u>SAL</u>	<u>XCHG</u>
<u>CMC</u>	<u>INTO</u>	<u>JNAE</u>	<u>JP</u>	<u>LOOPNZ</u>	<u>PUSHA</u>	<u>SAR</u>	<u>XLATB</u>
<u>CMP</u>	<u>IRET</u>	<u>JNB</u>	<u>JPE</u>	<u>LOOPZ</u>	<u>PUSHF</u>	<u>SBB</u>	<u>XOR</u>
	<u>JA</u>				<u>RCL</u>		

Operand types:

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

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

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

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

Notes:

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

REG, memory

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

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

8086 assembler tutorial for beginners

- 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:

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

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

1 - instruction sets this flag to **1**.

0 - instruction sets this flag to **0**.

r - flag value depends on result of the instruction.

? - flag value is undefined (maybe **1** or **0**).

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

Instructions in alphabetical order:

Instruction	Operands	Description
AAA	No operands	<p>ASCII Adjust after Addition. Corrects result in AH and AL after addition when working with BCD values.</p> <p>It works according to the following Algorithm:</p> <p>if low nibble of AL > 9 or AF = 1 then:</p> <ul style="list-style-type: none">• AL = AL + 6• AH = AH + 1• AF = 1• CF = 1 <p>else</p> <ul style="list-style-type: none">• AF = 0• CF = 0 <p>in both cases: clear the high nibble of AL.</p> <p>Example:</p> <p>MOV AX, 15 ; AH = 00, AL = 0Fh</p> <p>AAA ; AH = 01, AL = 05</p> <p>RET</p>


		<table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>?</td><td>?</td><td>?</td><td>?</td><td>r</td></tr></table> <div></div>	C	Z	S	O	P	A	r	?	?	?	?	r
C	Z	S	O	P	A									
r	?	?	?	?	r									
AAD	No operands	<p>ASCII Adjust before Division. Prepares two BCD values for division.</p> <p>Algorithm:</p> <ul style="list-style-type: none">AL = (AH * 10) + ALAH = 0 <p>Example:</p> <p>MOV AX, 0105h ; AH = 01, AL = 05</p> <p>AAD ; AH = 00, AL = 0Fh (15)</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>?</td><td>r</td><td>r</td><td>?</td><td>r</td><td>?</td></tr></table> <div></div>	C	Z	S	O	P	A	?	r	r	?	r	?
C	Z	S	O	P	A									
?	r	r	?	r	?									
AAM	No operands	<p>ASCII Adjust after Multiplication. Corrects the result of multiplication of two BCD values.</p> <p>Algorithm:</p> <ul style="list-style-type: none">AH = AL / 10AL = remainder <p>Example:</p> <p>MOV AL, 15 ; AL = 0Fh</p> <p>AAM ; AH = 01, AL = 05</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr></table>	C	Z	S	O	P	A						
C	Z	S	O	P	A									

		<table><tr><td>?</td><td>r</td><td>r</td><td>?</td><td>r</td><td>?</td></tr></table> <div></div>	?	r	r	?	r	?						
?	r	r	?	r	?									
AAS	No operands	<p>ASCII Adjust after Subtraction. Corrects result in AH and AL after subtraction when working with BCD values.</p> <p>Algorithm:</p> <p>if low nibble of AL > 9 or AF = 1 then:</p> <ul style="list-style-type: none">• AL = AL - 6• AH = AH - 1• AF = 1• CF = 1 <p>else</p> <ul style="list-style-type: none">• AF = 0• CF = 0 <p>in both cases: clear the high nibble of AL.</p> <p>Example:</p> <p>MOV AX, 02FFh ; AH = 02, AL = 0FFh</p> <p>AAS ; AH = 01, AL = 09</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>?</td><td>?</td><td>?</td><td>?</td><td>r</td></tr></table> <div></div>	C	Z	S	O	P	A	r	?	?	?	?	r
C	Z	S	O	P	A									
r	?	?	?	?	r									
ADC	REG, memory memory, REG REG, REG memory, immediate	<p>Add with Carry.</p> <p>Algorithm:</p>												

8086 assembler tutorial for beginners

	REG, immediate	<p>operand1 = operand1 + operand2 + CF</p> <p>Example:</p> <p>STC ; set CF = 1</p> <p>MOV AL, 5 ; AL = 5</p> <p>ADC AL, 1 ; AL = 7</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table> 	C	Z	S	O	P	A	r	r	r	r	r	r
C	Z	S	O	P	A									
r	r	r	r	r	r									
ADD	REG, memory memory, REG REG, REG memory, immediate REG, immediate	<p>Add.</p> <p>Algorithm:</p> <p>operand1 = operand1 + operand2</p> <p>Example:</p> <p>MOV AL, 5 ; AL = 5</p> <p>ADD AL, -3 ; AL = 2</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table> 	C	Z	S	O	P	A	r	r	r	r	r	r
C	Z	S	O	P	A									
r	r	r	r	r	r									
AND	REG, memory memory, REG REG, REG memory, immediate REG, immediate	<p>Logical AND between all bits of two operands. Result is stored in operand1.</p> <p>These rules apply:</p> <p>1 AND 1 = 1</p> <p>1 AND 0 = 0</p>												

8086 assembler tutorial for beginners

		<p>0 AND 1 = 0 0 AND 0 = 0</p> <p>Example:</p> <p>MOV AL, 'a' ; AL = 01100001b</p> <p>AND AL, 11011111b ; AL = 01000001b ('A')</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td></tr><tr><td>0</td><td>r</td><td>r</td><td>0</td><td>r</td></tr></table> 	C	Z	S	O	P	0	r	r	0	r
C	Z	S	O	P								
0	r	r	0	r								
CALL	procedure name label 4-byte address	<p>Transfers control to procedure. Return address (IP) is pushed to stack. <i>4-byte address</i> may be entered in this form: 1234h:5678h, first value is a segment second value is an offset. If it's a far call, then code segment is pushed to stack as well.</p> <p>Example:</p> <p>ORG 100h ; for COM file.</p> <p>CALL p1</p> <p>ADD AX, 1</p> <p>RET ; return to OS.</p> <p>p1 PROC ; procedure declaration.</p> <p>MOV AX, 1234h</p> <p>RET ; return to caller.</p>										



8086 assembler tutorial for beginners


		<p>p1 ENDP</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
CBW	No operands	<p>Convert byte into word.</p> <p>Algorithm:</p> <p>if high bit of AL = 1 then:</p> <ul style="list-style-type: none">AH = 255 (0FFh) <p>else</p> <ul style="list-style-type: none">AH = 0 <p>Example:</p> <p>MOV AX, 0 ; AH = 0, AL = 0</p> <p>MOV AL, -5 ; AX = 000FBh (251)</p> <p>CBW ; AX = 0FFFBh (-5)</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
CLC	No operands	<p>Clear Carry flag.</p> <p>Algorithm:</p> <p>CF = 0</p> <table><tr><td>C</td></tr><tr><td>0</td></tr></table>	C	0										
C														
0														



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

CMP	REG, memory memory, REG REG, REG memory, immediate REG, immediate	<p>Compare.</p> <p>Algorithm:</p> <p>operand1 - operand2</p> <p>result is not stored anywhere, flags are set (OF, SF, ZF, AF, PF, CF) according to result.</p> <p>Example:</p> <p>MOV AL, 5</p> <p>MOV BL, 5</p> <p>CMP AL, BL ; AL = 5, ZF = 1 (so equal!)</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table> <div></div>	C	Z	S	O	P	A	r	r	r	r	r	r
C	Z	S	O	P	A									
r	r	r	r	r	r									
CMPSB	No operands	<p>Compare bytes: ES:[DI] from DS:[SI].</p> <p>Algorithm:</p> <ul style="list-style-type: none">DS:[SI] - ES:[DI]set flags according to result: OF, SF, ZF, AF, PF, CFif DF = 0 then<ul style="list-style-type: none">SI = SI + 1DI = DI + 1else<ul style="list-style-type: none">SI = SI - 1DI = DI - 1 <p>Example:</p>												




8086 assembler tutorial for beginners


		<p>open cmpsb.asm from c:\emu8086\examples</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table> <div></div>	C	Z	S	O	P	A	r	r	r	r	r	r
C	Z	S	O	P	A									
r	r	r	r	r	r									
CMPSW	No operands	<p>Compare words: ES:[DI] from DS:[SI].</p> <p>Algorithm:</p> <ul style="list-style-type: none">• DS:[SI] - ES:[DI]• set flags according to result: OF, SF, ZF, AF, PF, CF• if DF = 0 then<ul style="list-style-type: none">• SI = SI + 2• DI = DI + 2else<ul style="list-style-type: none">• SI = SI - 2• DI = DI - 2 <p>example:</p> <p>open cmpsw.asm from c:\emu8086\examples</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table> <div></div>	C	Z	S	O	P	A	r	r	r	r	r	r
C	Z	S	O	P	A									
r	r	r	r	r	r									
CWD	No operands	<p>Convert Word to Double word.</p> <p>Algorithm:</p> <p>if high bit of AX = 1 then:</p> <ul style="list-style-type: none">• DX = 65535 (0FFFFh) <p>else</p>												

		<ul style="list-style-type: none">• DX = 0 <p>Example:</p> <p>MOV DX, 0 ; DX = 0</p> <p>MOV AX, 0 ; AX = 0</p> <p>MOV AX, -5 ; DX AX = 00000h:0FFFFBh</p> <p>CWD ; DX AX = 0FFFFh:0FFFFBh</p> <p>RET</p> <table border="1"><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
DAA	No operands	<p>Decimal adjust After Addition.</p> <p>Corrects the result of addition of two packed BCD values.</p> <p>Algorithm:</p> <p>if low nibble of AL > 9 or AF = 1 then:</p> <ul style="list-style-type: none">• AL = AL + 6• AF = 1 <p>if AL > 9Fh or CF = 1 then:</p> <ul style="list-style-type: none">• AL = AL + 60h• CF = 1 <p>Example:</p> <p>MOV AL, 0Fh ; AL = 0Fh (15)</p> <p>DAA ; AL = 15h</p> <p>RET</p> <table border="1"><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table>	C	Z	S	O	P	A	r	r	r	r	r	r
C	Z	S	O	P	A									
r	r	r	r	r	r									

														
DAS	No operands	<p>Decimal adjust After Subtraction. Corrects the result of subtraction of two packed BCD values.</p> <p>Algorithm:</p> <p>if low nibble of AL > 9 or AF = 1 then:</p> <ul style="list-style-type: none"> • AL = AL - 6 • AF = 1 <p>if AL > 9Fh or CF = 1 then:</p> <ul style="list-style-type: none"> • AL = AL - 60h • CF = 1 <p>Example:</p> <p>MOV AL, 0FFh ; AL = 0FFh (-1)</p> <p>DAS ; AL = 99h, CF = 1</p> <p>RET</p> <table border="1"> <tr> <td>C</td> <td>Z</td> <td>S</td> <td>O</td> <td>P</td> <td>A</td> </tr> <tr> <td>r</td> <td>r</td> <td>r</td> <td>r</td> <td>r</td> <td>r</td> </tr> </table> 	C	Z	S	O	P	A	r	r	r	r	r	r
C	Z	S	O	P	A									
r	r	r	r	r	r									
DEC	REG memory	<p>Decrement.</p> <p>Algorithm:</p> <p>operand = operand - 1</p> <p>Example:</p> <p>MOV AL, 255 ; AL = 0FFh (255 or -1)</p> <p>DEC AL ; AL = 0FEh (254 or -2)</p> <p>RET</p>												




8086 assembler tutorial for beginners



		<table><tr><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table> <p>CF - unchanged!</p> <div></div>	Z	S	O	P	A	r	r	r	r	r		
Z	S	O	P	A										
r	r	r	r	r										
DIV	REG memory	<p>Unsigned divide.</p> <p>Algorithm:</p> <p>when operand is a byte:</p> <p>AL = AX / operand AH = remainder (modulus)</p> <p>when operand is a word:</p> <p>AX = (DX AX) / operand DX = remainder (modulus)</p> <p>Example:</p> <p>MOV AX, 203 ; AX = 00CBh</p> <p>MOV BL, 4</p> <p>DIV BL ; AL = 50 (32h), AH = 3</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td></tr></table> <div></div>	C	Z	S	O	P	A	?	?	?	?	?	?
C	Z	S	O	P	A									
?	?	?	?	?	?									
HLT	No operands	<p>Halt the System.</p> <p>Example:</p> <p>MOV AX, 5</p> <p>HLT</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														

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




8086 assembler tutorial for beginners

		<table><tr><td>r</td><td>?</td><td>?</td><td>r</td><td>?</td><td>?</td></tr></table> <p>CF=OF=0 when result fits into operand of IMUL.</p> <div></div>	r	?	?	r	?	?						
r	?	?	r	?	?									
IN	AL, im.byte AL, DX AX, im.byte AX, DX	<p>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.</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
INC	REG memory	<p>Increment.</p> <p>Algorithm:</p> <p>operand = operand + 1</p> <p>Example: MOV AL, 4 INC AL ; AL = 5 RET</p> <table><tr><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table> <p>CF - unchanged!</p> <div></div>	Z	S	O	P	A	r	r	r	r	r		
Z	S	O	P	A										
r	r	r	r	r										
INT	immediate byte	<p>Interrupt numbered by immediate byte (0..255).</p> <p>Algorithm:</p> <p>Push to stack:</p>												


		<ul style="list-style-type: none">• flags register• CS• IP• IF = 0• Transfer control to interrupt procedure <p>Example:</p> <p>MOV AH, 0Eh ; teletype.</p> <p>MOV AL, 'A'</p> <p>INT 10h ; BIOS interrupt.</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td><td>I</td></tr><tr><td colspan="6">unchanged</td><td>0</td></tr></table> 	C	Z	S	O	P	A	I	unchanged						0
C	Z	S	O	P	A	I										
unchanged						0										
INTO	No operands	<p>Interrupt 4 if Overflow flag is 1.</p> <p>Algorithm:</p> <p>if OF = 1 then INT 4</p> <p>Example:</p> <p>; -5 - 127 = -132 (not in -128..127)</p> <p>; the result of SUB is wrong (124),</p> <p>; so OF = 1 is set:</p> <p>MOV AL, -5</p> <p>SUB AL, 127 ; AL = 7Ch (124)</p> <p>INTO ; process error.</p> <p>RET</p> 														

8086 assembler tutorial for beginners



IRET	No operands	<p>Interrupt Return.</p> <p>Algorithm:</p> <p>Pop from stack:</p> <ul style="list-style-type: none">• IP• CS• flags register <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">popped</td></tr></table> <div></div>	C	Z	S	O	P	A	popped					
C	Z	S	O	P	A									
popped														
JA	label	<p>Short Jump if first operand is Above second operand (as set by CMP instruction). Unsigned.</p> <p>Algorithm:</p> <p>if (CF = 0) and (ZF = 0) then jump</p> <p>Example:</p> <pre>include 'emu8086.inc' 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</pre> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														

		<div></div> <div></div>												
JAE	label	<p>Short Jump if first operand is Above or Equal to second operand (as set by CMP instruction). Unsigned.</p> <p>Algorithm:</p> <p>if CF = 0 then jump</p> <p>Example:</p> <pre>include 'emu8086.inc' 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</pre> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
JB	label	<p>Short Jump if first operand is Below second operand (as set by CMP instruction). Unsigned.</p> <p>Algorithm:</p> <p>if CF = 1 then jump</p> <p>Example:</p>												


8086 assembler tutorial for beginners

		<pre>include 'emu8086.inc' 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</pre> <table border="1"><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> 	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
JBE	label	<p>Short Jump if first operand is Below or Equal to second operand (as set by CMP instruction). Unsigned.</p> <p>Algorithm:</p> <p>if CF = 1 or ZF = 1 then jump</p> <p>Example:</p> <pre>include 'emu8086.inc' ORG 100h MOV AL, 5 CMP AL, 5 JBE label1 PRINT 'AL is not below or equal to 5' JMP exit label1:</pre>												



8086 assembler tutorial for beginners


		<p>PRINT 'AL is below or equal to 5'</p> <p>exit:</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table>	C	Z	S	O	P	A	unchanged						
C	Z	S	O	P	A										
unchanged															
JC	label	<p>Short Jump if Carry flag is set to 1.</p> <p>Algorithm:</p> <p>if CF = 1 then jump</p> <p>Example:</p> <p>include 'emu8086.inc'</p> <p>ORG 100h</p> <p>MOV AL, 255</p> <p>ADD AL, 1</p> <p>JC label1</p> <p>PRINT 'no carry.'</p> <p>JMP exit</p> <p>label1:</p> <p>PRINT 'has carry.'</p> <p>exit:</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table>	C	Z	S	O	P	A	unchanged						
C	Z	S	O	P	A										
unchanged															
JCXZ	label	Short Jump if CX register is 0.													

8086 assembler tutorial for beginners


		<p>Algorithm:</p> <p>if CX = 0 then jump</p> <p>Example:</p> <pre>include 'emu8086.inc' ORG 100h MOV CX, 0 JCXZ label1 PRINT 'CX is not zero.' JMP exit label1: PRINT 'CX is zero.' exit: RET</pre> <table border="1"><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
JE	label	<p>Short Jump if first operand is Equal to second operand (as set by CMP instruction). Signed/Unsigned.</p> <p>Algorithm:</p> <p>if ZF = 1 then jump</p> <p>Example:</p> <pre>include 'emu8086.inc' ORG 100h MOV AL, 5 CMP AL, 5 JE label1 PRINT 'AL is not equal to 5.'</pre>												



		<p>JMP exit</p> <p>label1:</p> <p>PRINT 'AL is equal to 5.'</p> <p>exit:</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> 	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
JG	label	<p>Short Jump if first operand is Greater then second operand (as set by CMP instruction). Signed.</p> <p>Algorithm:</p> <p>if (ZF = 0) and (SF = OF) then jump</p> <p>Example:</p> <p>include 'emu8086.inc'</p> <p>ORG 100h</p> <p>MOV AL, 5</p> <p>CMP AL, -5</p> <p>JG label1</p> <p>PRINT 'AL is not greater -5.'</p> <p>JMP exit</p> <p>label1:</p> <p>PRINT 'AL is greater -5.'</p> <p>exit:</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> 	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														

JGE	label	<p>Short Jump if first operand is Greater or Equal to second operand (as set by CMP instruction). Signed.</p> <p>Algorithm:</p> <p style="padding-left: 40px;">if SF = OF then jump</p> <p>Example:</p> <pre>include 'emu8086.inc' ORG 100h MOV AL, 2 CMP AL, -5 JGE label1 PRINT 'AL < -5' JMP exit</pre> <p>label1:</p> <pre>PRINT 'AL >= -5'</pre> <p>exit:</p> <pre>RET</pre> <table border="1"><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
JL	label	<p>Short Jump if first operand is Less then second operand (as set by CMP instruction). Signed.</p> <p>Algorithm:</p> <p style="padding-left: 40px;">if SF <> OF then jump</p> <p>Example:</p> <pre>include 'emu8086.inc' ORG 100h MOV AL, -2</pre>												




		<div>CMP AL, 5</div> <div>JL label1</div> <div>PRINT 'AL >= 5.'</div> <div>JMP exit</div> <div>label1:</div> <div>PRINT 'AL < 5.'</div> <div>exit:</div> <div>RET</div> <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table></div> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
JLE	label	<div>Short Jump if first operand is Less or Equal to second operand (as set by CMP instruction). Signed.</div> <div>Algorithm:</div> <div>if SF <> OF or ZF = 1 then jump</div> <div>Example:</div> <div>include 'emu8086.inc'</div> <div>ORG 100h</div> <div>MOV AL, -2</div> <div>CMP AL, 5</div> <div>JLE label1</div> <div>PRINT 'AL > 5.'</div> <div>JMP exit</div> <div>label1:</div> <div>PRINT 'AL <= 5.'</div> <div>exit:</div> <div>RET</div>												




8086 assembler tutorial for beginners


		<table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div>↑</div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
JMP	label 4-byte address	<p>Unconditional Jump. Transfers control to another part of the program. <i>4-byte address</i> may be entered in this form: 1234h:5678h, first value is a segment second value is an offset.</p> <p>Algorithm:</p> <p>always jump</p> <p>Example:</p> <pre>include 'emu8086.inc' ORG 100h MOV AL, 5 JMP label1 ; jump over 2 lines! PRINT 'Not Jumped!' MOV AL, 0 label1: PRINT 'Got Here!' RET</pre> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div>↑</div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
JNA	label	<p>Short Jump if first operand is Not Above second operand (as set by CMP instruction). Unsigned.</p> <p>Algorithm:</p> <p>if CF = 1 or ZF = 1 then jump</p>												

8086 assembler tutorial for beginners

		<p>Example:</p> <pre>include 'emu8086.inc'</pre> <p>ORG 100h</p> <p>MOV AL, 2</p> <p>CMP AL, 5</p> <p>JNA label1</p> <p>PRINT 'AL is above 5.'</p> <p>JMP exit</p> <p>label1:</p> <p>PRINT 'AL is not above 5.'</p> <p>exit:</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> 	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
JNAE	label	<p>Short Jump if first operand is Not Above and Not Equal to second operand (as set by CMP instruction). Unsigned.</p> <p>Algorithm:</p> <p>if CF = 1 then jump</p> <p>Example:</p> <pre>include 'emu8086.inc'</pre> <p>ORG 100h</p> <p>MOV AL, 2</p> <p>CMP AL, 5</p> <p>JNAE label1</p>												

		<div>PRINT 'AL >= 5.'</div> <div>JMP exit</div> <div>label1:</div> <div>PRINT 'AL < 5.'</div> <div>exit:</div> <div>RET</div> <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table></div> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
JNB	label	<div>Short Jump if first operand is Not Below second operand (as set by CMP instruction). Unsigned.</div> <div>Algorithm:</div> <div>if CF = 0 then jump</div> <div>Example:</div> <div>include 'emu8086.inc'</div> <div>ORG 100h</div> <div>MOV AL, 7</div> <div>CMP AL, 5</div> <div>JNB label1</div> <div>PRINT 'AL < 5.'</div> <div>JMP exit</div> <div>label1:</div> <div>PRINT 'AL >= 5.'</div> <div>exit:</div> <div>RET</div> <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr></table></div>	C	Z	S	O	P	A						
C	Z	S	O	P	A									



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

		<pre>include 'emu8086.inc' ORG 100h MOV AL, 2 ADD AL, 3 JNC label1 PRINT 'has carry.' JMP exit label1: PRINT 'no carry.' exit: RET</pre> <table border="1"><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
JNE	label	<p>Short Jump if first operand is Not Equal to second operand (as set by CMP instruction). Signed/Unsigned.</p> <p>Algorithm:</p> <p>if ZF = 0 then jump</p> <p>Example:</p> <pre>include 'emu8086.inc' ORG 100h MOV AL, 2 CMP AL, 3 JNE label1 PRINT 'AL = 3.'</pre>												

8086 assembler tutorial for beginners


		<p>JMP exit</p> <p>label1:</p> <p>PRINT 'Al <> 3.'</p> <p>exit:</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
JNG	label	<p>Short Jump if first operand is Not Greater then second operand (as set by CMP instruction). Signed.</p> <p>Algorithm:</p> <p>if (ZF = 1) and (SF <> OF) then jump</p> <p>Example:</p> <p>include 'emu8086.inc'</p> <p>ORG 100h</p> <p>MOV AL, 2</p> <p>CMP AL, 3</p> <p>JNG label1</p> <p>PRINT 'AL > 3.'</p> <p>JMP exit</p> <p>label1:</p> <p>PRINT 'Al <= 3.'</p> <p>exit:</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														




		<div></div>												
JNGE	label	<p>Short Jump if first operand is Not Greater and Not Equal to second operand (as set by CMP instruction). Signed.</p> <p>Algorithm:</p> <p style="padding-left: 40px;">if SF <> OF then jump</p> <p>Example:</p> <pre>include 'emu8086.inc' ORG 100h MOV AL, 2 CMP AL, 3 JNGE label1 PRINT 'AL >= 3.' JMP exit label1: PRINT 'Al < 3.' exit: RET</pre> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
		<div></div>												
JNL	label	<p>Short Jump if first operand is Not Less then second operand (as set by CMP instruction). Signed.</p> <p>Algorithm:</p> <p style="padding-left: 40px;">if SF = OF then jump</p> <p>Example:</p>												



8086 assembler tutorial for beginners



		<pre>include 'emu8086.inc' ORG 100h MOV AL, 2 CMP AL, -3 JNL label1 PRINT 'AL < -3.' JMP exit label1: PRINT 'Al >= -3.' exit: RET</pre> <table border="1"><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
JNLE	label	<p>Short Jump if first operand is Not Less and Not Equal to second operand (as set by CMP instruction). Signed.</p> <p>Algorithm:</p> <p>if (SF = OF) and (ZF = 0) then jump</p> <p>Example:</p> <pre>include 'emu8086.inc' ORG 100h MOV AL, 2 CMP AL, -3 JNLE label1 PRINT 'AL <= -3.'</pre>												


8086 assembler tutorial for beginners

		<div>JMP exit</div> <div>label1:</div> <div>PRINT 'Al > -3.'</div> <div>exit:</div> <div>RET</div> <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table></div> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
JNO	label	<div>Short Jump if Not Overflow.</div> <div>Algorithm:</div> <div>if OF = 0 then jump</div> <div>Example:</div> <div>; -5 - 2 = -7 (inside -128..127)</div> <div>; the result of SUB is correct,</div> <div>; so OF = 0:</div> <div>include 'emu8086.inc'</div> <div>ORG 100h</div> <div>MOV AL, -5</div> <div>SUB AL, 2 ; AL = 0F9h (-7)</div> <div>JNO label1</div> <div>PRINT 'overflow!'</div> <div>JMP exit</div> <div>label1:</div> <div>PRINT 'no overflow.'</div> <div>exit:</div>												




8086 assembler tutorial for beginners

		<div>RET</div> <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table></div> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
JNP	label	<p>Short Jump if No Parity (odd). Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.</p> <p>Algorithm:</p> <p>if PF = 0 then jump</p> <p>Example:</p> <p>include 'emu8086.inc'</p> <p>ORG 100h</p> <p>MOV AL, 00000111b ; AL = 7</p> <p>OR AL, 0 ; just set flags.</p> <p>JNP label1</p> <p>PRINT 'parity even.'</p> <p>JMP exit</p> <p>label1:</p> <p>PRINT 'parity odd.'</p> <p>exit:</p> <p>RET</p> <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table></div> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
JNS	label	<p>Short Jump if Not Signed (if positive). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.</p>												

		<p>Algorithm:</p> <p>if SF = 0 then jump</p> <p>Example:</p> <p>include 'emu8086.inc'</p> <p>ORG 100h</p> <p>MOV AL, 00000111b ; AL = 7</p> <p>OR AL, 0 ; just set flags.</p> <p>JNS label1</p> <p>PRINT 'signed.'</p> <p>JMP exit</p> <p>label1:</p> <p>PRINT 'not signed.'</p> <p>exit:</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
JNZ	label	<p>Short Jump if Not Zero (not equal). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.</p> <p>Algorithm:</p> <p>if ZF = 0 then jump</p> <p>Example:</p> <p>include 'emu8086.inc'</p> <p>ORG 100h</p>												





8086 assembler tutorial for beginners

		<div>MOV AL, 00000111b ; AL = 7</div> <div>OR AL, 0 ; just set flags.</div> <div>JNZ label1</div> <div>PRINT 'zero.'</div> <div>JMP exit</div> <div>label1:</div> <div>PRINT 'not zero.'</div> <div>exit:</div> <div>RET</div> <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table></div> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
JO	label	<div>Short Jump if Overflow.</div> <div>Algorithm:</div> <div>if OF = 1 then jump</div> <div>Example:</div> <div>; -5 - 127 = -132 (not in -128..127)</div> <div>; the result of SUB is wrong (124),</div> <div>; so OF = 1 is set:</div> <div>include 'emu8086.inc'</div> <div>org 100h</div> <div>MOV AL, -5</div> <div>SUB AL, 127 ; AL = 7Ch (124)</div> <div>JO label1</div> <div>PRINT 'no overflow.'</div>												





8086 assembler tutorial for beginners

		<p>JMP exit</p> <p>label1:</p> <p>PRINT 'overflow!'</p> <p>exit:</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
JP	label	<p>Short Jump if Parity (even). Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.</p> <p>Algorithm:</p> <p>if PF = 1 then jump</p> <p>Example:</p> <p>include 'emu8086.inc'</p> <p>ORG 100h</p> <p>MOV AL, 00000101b ; AL = 5</p> <p>OR AL, 0 ; just set flags.</p> <p>JP label1</p> <p>PRINT 'parity odd.'</p> <p>JMP exit</p> <p>label1:</p> <p>PRINT 'parity even.'</p> <p>exit:</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														

		<div></div> <div></div>												
JPE	label	<p>Short Jump if Parity Even. Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.</p> <p>Algorithm:</p> <p>if PF = 1 then jump</p> <p>Example:</p> <p>include 'emu8086.inc'</p> <p>ORG 100h</p> <p>MOV AL, 00000101b ; AL = 5</p> <p>OR AL, 0 ; just set flags.</p> <p>JPE label1</p> <p>PRINT 'parity odd.'</p> <p>JMP exit</p> <p>label1:</p> <p>PRINT 'parity even.'</p> <p>exit:</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
JPO	label	<p>Short Jump if Parity Odd. Only 8 low bits of result are checked. Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.</p> <p>Algorithm:</p>												

8086 assembler tutorial for beginners


		<div>if PF = 0 then jump</div> <div>Example:</div> <div>include 'emu8086.inc'</div> <div>ORG 100h</div> <div>MOV AL, 00000111b ; AL = 7</div> <div>OR AL, 0 ; just set flags.</div> <div>JPO label1</div> <div>PRINT 'parity even.'</div> <div>JMP exit</div> <div>label1:</div> <div>PRINT 'parity odd.'</div> <div>exit:</div> <div>RET</div> <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table></div> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
JS	label	<div>Short Jump if Signed (if negative). Set by CMP, SUB, ADD, TEST, AND, OR, XOR instructions.</div> <div>Algorithm:</div> <div>if SF = 1 then jump</div> <div>Example:</div> <div>include 'emu8086.inc'</div> <div>ORG 100h</div> <div>MOV AL, 10000000b ; AL = -128</div> <div>OR AL, 0 ; just set flags.</div>												

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


8086 assembler tutorial for beginners

		<table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
LAHF	No operands	<p>Load AH from 8 low bits of Flags register.</p> <p>Algorithm:</p> <p>AH = flags register</p> <p>AH bit: 7 6 5 4 3 2 1 0</p> <p>[SF] [ZF] [0] [AF] [0] [PF] [1] [CF]</p> <p>bits 1, 3, 5 are reserved.</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
LDS	REG, memory	<p>Load memory double word into word register and DS.</p> <p>Algorithm:</p> <ul style="list-style-type: none">REG = first wordDS = second word <p>Example:</p> <p>ORG 100h</p> <p>LDS AX, m</p> <p>RET</p>												

8086 assembler tutorial for beginners


		<p>m DW 1234h</p> <p>DW 5678h</p> <p>END</p> <p>AX is set to 1234h, DS is set to 5678h.</p> <table border="1"><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
LEA	REG, memory	<p>Load Effective Address.</p> <p>Algorithm:</p> <ul style="list-style-type: none">REG = address of memory (offset) <p>Example:</p> <p>MOV BX, 35h</p> <p>MOV DI, 12h</p> <p>LEA SI, [BX+DI] ; SI = 35h + 12h = 47h</p> <p>Note: The integrated 8086 assembler automatically replaces LEA with a more efficient MOV where possible. For example:</p> <p>org 100h</p> <p>LEA AX, m ; AX = offset of m</p> <p>RET</p> <p>m dw 1234h</p>												

8086 assembler tutorial for beginners


		<div>END</div> <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table></div> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
LES	REG, memory	<div>Load memory double word into word register and ES.</div> <div>Algorithm:<ul style="list-style-type: none">REG = first wordES = second word</div> <div>Example:</div> <div>ORG 100h</div> <div>LES AX, m</div> <div>RET</div> <div>m DW 1234h</div> <div>DW 5678h</div> <div>END</div> <div>AX is set to 1234h, ES is set to 5678h.</div> <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														

		<div>⬆</div>												
LODSB	No operands	<p>Load byte at DS:[SI] into AL. Update SI.</p> <p>Algorithm:</p> <ul style="list-style-type: none">• AL = DS:[SI]• if DF = 0 then<ul style="list-style-type: none">• SI = SI + 1else<ul style="list-style-type: none">• SI = SI - 1 <p>Example:</p> <p>ORG 100h</p> <p>LEA SI, a1</p> <p>MOV CX, 5</p> <p>MOV AH, 0Eh</p> <p>m: LODSB</p> <p>INT 10h</p> <p>LOOP m</p> <p>RET</p> <p>a1 DB 'H', 'e', 'l', 'l', 'o'</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
		<div>⬆</div>												


8086 assembler tutorial for beginners

LODSW	No operands	<p>Load word at DS:[SI] into AX. Update SI.</p> <p>Algorithm:</p> <ul style="list-style-type: none">• AX = DS:[SI]• if DF = 0 then<ul style="list-style-type: none">• SI = SI + 2else<ul style="list-style-type: none">• SI = SI - 2 <p>Example:</p> <p>ORG 100h</p> <p>LEA SI, a1</p> <p>MOV CX, 5</p> <p>REP LODSW ; finally there will be 555h in AX.</p> <p>RET</p> <p>a1 dw 111h, 222h, 333h, 444h, 555h</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> 	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
LOOP	label	<p>Decrease CX, jump to label if CX not zero.</p> <p>Algorithm:</p> <ul style="list-style-type: none">• CX = CX - 1• if CX <> 0 then												

8086 assembler tutorial for beginners


		<ul style="list-style-type: none">• jump <p>else</p> <ul style="list-style-type: none">• no jump, continue <p>Example:</p> <pre>include 'emu8086.inc'</pre> <p>ORG 100h</p> <p>MOV CX, 5</p> <p>label1:</p> <pre>PRINTN 'loop!' LOOP label1 RET</pre> <table border="1"><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> 	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
LOOPE	label	<p>Decrease CX, jump to label if CX not zero and Equal (ZF = 1).</p> <p>Algorithm:</p> <ul style="list-style-type: none">• CX = CX - 1• if (CX <> 0) and (ZF = 1) then<ul style="list-style-type: none">• jump <p>else</p> <ul style="list-style-type: none">• no jump, continue <p>Example:</p> <p>; Loop until result fits into AL alone,</p> <p>; or 5 times. The result will be over 255</p> <p>; on third loop (100+100+100),</p>												

8086 assembler tutorial for beginners


		<p>; so loop will exit.</p> <p>include 'emu8086.inc'</p> <p>ORG 100h</p> <p>MOV AX, 0</p> <p>MOV CX, 5</p> <p>label1:</p> <p>PUTC '*'</p> <p>ADD AX, 100</p> <p>CMP AH, 0</p> <p>LOOPE label1</p> <p>RET</p> <table border="1"><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
LOOPNE	label	<p>Decrease CX, jump to label if CX not zero and Not Equal (ZF = 0).</p> <p>Algorithm:</p> <ul style="list-style-type: none">• CX = CX - 1• if (CX <> 0) and (ZF = 0) then<ul style="list-style-type: none">• jumpelse<ul style="list-style-type: none">• no jump, continue <p>Example:</p> <p>; Loop until '7' is found,</p> <p>; or 5 times.</p>												




8086 assembler tutorial for beginners

		<pre>include 'emu8086.inc' 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</pre> <table border="1"><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> 	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
LOOPNZ	label	<p>Decrease CX, jump to label if CX not zero and ZF = 0.</p> <p>Algorithm:</p> <ul style="list-style-type: none">• CX = CX - 1• if (CX <> 0) and (ZF = 0) then<ul style="list-style-type: none">• jumpelse<ul style="list-style-type: none">• no jump, continue <p>Example:</p> <p>; Loop until '7' is found,</p> <p>; or 5 times.</p>												


8086 assembler tutorial for beginners

		<pre>include 'emu8086.inc' 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</pre> <table border="1"><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> 	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
LOOPZ	label	<p>Decrease CX, jump to label if CX not zero and ZF = 1.</p> <p>Algorithm:</p> <ul style="list-style-type: none">• CX = CX - 1• if (CX <> 0) and (ZF = 1) then<ul style="list-style-type: none">• jumpelse<ul style="list-style-type: none">• no jump, continue <p>Example:</p> <p>; Loop until result fits into AL alone, ; or 5 times. The result will be over 255</p>												


8086 assembler tutorial for beginners

		<pre>; on third loop (100+100+100), ; so loop will exit. include 'emu8086.inc' ORG 100h MOV AX, 0 MOV CX, 5 label1: PUTC '*' ADD AX, 100 CMP AH, 0 LOOPZ label1 RET</pre> <table border="1"><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> 	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
MOV	<p>REG, memory memory, REG REG, REG memory, immediate REG, immediate</p> <p>SREG, memory memory, SREG REG, SREG SREG, REG</p>	<p>Copy operand2 to operand1.</p> <p>The MOV instruction <u>cannot</u>:</p> <ul style="list-style-type: none">• 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). <p>Algorithm:</p> <p>operand1 = operand2</p>												

8086 assembler tutorial for beginners



		<p>Example:</p> <p>ORG 100h</p> <p>MOV AX, 0B800h ; set AX = B800h (VGA memory).</p> <p>MOV DS, AX ; copy value of AX to DS.</p> <p>MOV CL, 'A' ; CL = 41h (ASCII code).</p> <p>MOV CH, 01011111b ; CL = color attribute.</p> <p>MOV BX, 15Eh ; BX = position on screen.</p> <p>MOV [BX], CX ; w.[0B800h:015Eh] = CX.</p> <p>RET ; returns to operating system.</p> <table border="1"><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
MOVSB	No operands	<p>Copy byte at DS:[SI] to ES:[DI]. Update SI and DI.</p> <p>Algorithm:</p> <ul style="list-style-type: none">• ES:[DI] = DS:[SI]• if DF = 0 then<ul style="list-style-type: none">• SI = SI + 1• DI = DI + 1else<ul style="list-style-type: none">• SI = SI - 1• DI = DI - 1 <p>Example:</p> <p>ORG 100h</p> <p>CLD</p>												

8086 assembler tutorial for beginners



		<div>LEA SI, a1</div> <div>LEA DI, a2</div> <div>MOV CX, 5</div> <div>REP MOVSB</div> <div>RET</div> <div>a1 DB 1,2,3,4,5</div> <div>a2 DB 5 DUP(0)</div> <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table></div> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
MOVSW	No operands	<div>Copy word at DS:[SI] to ES:[DI]. Update SI and DI.</div> <div>Algorithm:</div> <div><ul style="list-style-type: none">• ES:[DI] = DS:[SI]• if DF = 0 then<ul style="list-style-type: none">• SI = SI + 2• DI = DI + 2else<ul style="list-style-type: none">• SI = SI - 2• DI = DI - 2</div> <div>Example:</div> <div>ORG 100h</div> <div>CLD</div> <div>LEA SI, a1</div>												





8086 assembler tutorial for beginners

		<p>LEA DI, a2</p> <p>MOV CX, 5</p> <p>REP MOVSW</p> <p>RET</p> <p>a1 DW 1,2,3,4,5</p> <p>a2 DW 5 DUP(0)</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table>	C	Z	S	O	P	A	unchanged						
C	Z	S	O	P	A										
unchanged															
MUL	REG memory	<p>Unsigned multiply.</p> <p>Algorithm:</p> <p>when operand is a byte:</p> <p>AX = AL * operand.</p> <p>when operand is a word:</p> <p>(DX AX) = AX * operand.</p> <p>Example:</p> <p>MOV AL, 200 ; AL = 0C8h</p> <p>MOV BL, 4</p> <p>MUL BL ; AX = 0320h (800)</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>?</td><td>?</td><td>r</td><td>?</td><td>?</td></tr></table> <p>CF=OF=0 when high section of the result is zero.</p>	C	Z	S	O	P	A	r	?	?	r	?	?	
C	Z	S	O	P	A										
r	?	?	r	?	?										
NEG	REG	Negate. Makes operand negative (two's complement).													

8086 assembler tutorial for beginners

	memory	<p>Algorithm:</p> <ul style="list-style-type: none">• Invert all bits of the operand• Add 1 to inverted operand <p>Example:</p> <p>MOV AL, 5 ; AL = 05h</p> <p>NEG AL ; AL = 0FBh (-5)</p> <p>NEG AL ; AL = 05h (5)</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table> 	C	Z	S	O	P	A	r	r	r	r	r	r
C	Z	S	O	P	A									
r	r	r	r	r	r									
NOP	No operands	<p>No Operation.</p> <p>Algorithm:</p> <ul style="list-style-type: none">• Do nothing <p>Example:</p> <p>; do nothing, 3 times:</p> <p>NOP</p> <p>NOP</p> <p>NOP</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> 	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
NOT	REG memory	<p>Invert each bit of the operand.</p> <p>Algorithm:</p>												

8086 assembler tutorial for beginners



		<ul style="list-style-type: none">if bit is 1 turn it to 0.if bit is 0 turn it to 1. <p>Example:</p> <p>MOV AL, 00011011b</p> <p>NOT AL ; AL = 11100100b</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> 	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
OR	REG, memory memory, REG REG, REG memory, immediate REG, immediate	<p>Logical OR between all bits of two operands. Result is stored in first operand.</p> <p>These rules apply:</p> <p>1 OR 1 = 1 1 OR 0 = 1 0 OR 1 = 1 0 OR 0 = 0</p> <p>Example:</p> <p>MOV AL, 'A' ; AL = 01000001b</p> <p>OR AL, 00100000b ; AL = 01100001b ('a')</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>0</td><td>r</td><td>r</td><td>0</td><td>r</td><td>?</td></tr></table> 	C	Z	S	O	P	A	0	r	r	0	r	?
C	Z	S	O	P	A									
0	r	r	0	r	?									
OUT	im.byte, AL im.byte, AX DX, AL	<p>Output from AL or AX to port.</p> <p>First operand is a port number. If required to access port number over 255 - DX register should be used.</p>												

8086 assembler tutorial for beginners



	DX, AX	<p>Example:</p> <p>MOV AX, 0FFFh ; Turn on all</p> <p>OUT 4, AX ; traffic lights.</p> <p>MOV AL, 100b ; Turn on the third</p> <p>OUT 7, AL ; magnet of the stepper-motor.</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div>↑</div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
POP	REG SREG memory	<p>Get 16 bit value from the stack.</p> <p>Algorithm:</p> <ul style="list-style-type: none">operand = SS:[SP] (top of the stack)SP = SP + 2 <p>Example:</p> <p>MOV AX, 1234h</p> <p>PUSH AX</p> <p>POP DX ; DX = 1234h</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div>↑</div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
POPA	No operands	<p>Pop all general purpose registers DI, SI, BP, SP, BX, DX, CX, AX from the stack.</p> <p>SP value is ignored, it is Popped but not set to SP register).</p> <p>Note: this instruction works only on 80186 CPU and later!</p>												





8086 assembler tutorial for beginners

		<p>Algorithm:</p> <ul style="list-style-type: none">• POP DI• POP SI• POP BP• POP xx (SP value ignored)• POP BX• POP DX• POP CX• POP AX <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> 	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
POPF	No operands	<p>Get flags register from the stack.</p> <p>Algorithm:</p> <ul style="list-style-type: none">• flags = SS:[SP] (top of the stack)• SP = SP + 2 <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">popped</td></tr></table> 	C	Z	S	O	P	A	popped					
C	Z	S	O	P	A									
popped														
PUSH	REG SREG memory immediate	<p>Store 16 bit value in the stack.</p> <p>Note: PUSH immediate works only on 80186 CPU and later!</p> <p>Algorithm:</p> <ul style="list-style-type: none">• SP = SP - 2• SS:[SP] (top of the stack) = operand												



8086 assembler tutorial for beginners


		<p>Example:</p> <p>MOV AX, 1234h</p> <p>PUSH AX</p> <p>POP DX ; DX = 1234h</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> 	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
PUSHA	No operands	<p>Push all general purpose registers AX, CX, DX, BX, SP, BP, SI, DI in the stack.</p> <p>Original value of SP register (before PUSHA) is used.</p> <p>Note: this instruction works only on 80186 CPU and later!</p> <p>Algorithm:</p> <ul style="list-style-type: none">• PUSH AX• PUSH CX• PUSH DX• PUSH BX• PUSH SP• PUSH BP• PUSH SI• PUSH DI <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> 	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
PUSHF	No operands	<p>Store flags register in the stack.</p> <p>Algorithm:</p>												

8086 assembler tutorial for beginners



		<ul style="list-style-type: none">• SP = SP - 2• SS:[SP] (top of the stack) = flags <table border="1"><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
RCL	<div>memory, immediate REG, immediate</div> <div>memory, CL REG, CL</div>	<p>Rotate operand1 left through Carry Flag. The number of rotates is set by operand2.</p> <p>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).</p> <p>Algorithm:</p> <div>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.</div> <p>Example:</p> <p>STC ; set carry (CF=1).</p> <p>MOV AL, 1Ch ; AL = 00011100b</p> <p>RCL AL, 1 ; AL = 00111001b, CF=0.</p> <p>RET</p> <table border="1"><tr><td>C</td><td>O</td></tr><tr><td>r</td><td>r</td></tr></table> <p>OF=0 if first operand keeps original sign.</p> <div></div>	C	O	r	r								
C	O													
r	r													
RCR	<div>memory, immediate REG, immediate</div> <div>memory, CL REG, CL</div>	<p>Rotate operand1 right through Carry Flag. The number of rotates is set by operand2.</p> <p>Algorithm:</p> <div>shift all bits right, the bit that goes off is set to CF</div>												

8086 assembler tutorial for beginners

		<p>and previous value of CF is inserted to the left-most position.</p> <p>Example:</p> <p>STC ; set carry (CF=1).</p> <p>MOV AL, 1Ch ; AL = 00011100b</p> <p>RCR AL, 1 ; AL = 10001110b, CF=0.</p> <p>RET</p> <table><tr><td>C</td><td>O</td></tr><tr><td>r</td><td>r</td></tr></table> <p>OF=0 if first operand keeps original sign.</p> <div></div>	C	O	r	r
C	O					
r	r					
REP	chain instruction	<p>Repeat following MOVSB, MOVSW, LODSB, LODSW, STOSB, STOSW instructions CX times.</p> <p>Algorithm:</p> <p>check_cx:</p> <p>if CX <> 0 then</p> <ul style="list-style-type: none">do following <u>chain instruction</u>CX = CX - 1go back to check_cx <p>else</p> <ul style="list-style-type: none">exit from REP cycle <table><tr><td>Z</td></tr><tr><td>r</td></tr></table> <div></div>	Z	r		
Z						
r						
REPE	chain instruction	<p>Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 1 (result is Equal), maximum CX</p>				

		<p>times.</p> <p>Algorithm:</p> <p>check_cx:</p> <p>if CX <> 0 then</p> <ul style="list-style-type: none">do following <u>chain instruction</u>CX = CX - 1if ZF = 1 then:<ul style="list-style-type: none">go back to check_cx <p>else</p> <ul style="list-style-type: none">exit from REPE cycle <p>else</p> <ul style="list-style-type: none">exit from REPE cycle <p>example:</p> <p>open cmpsb.asm from c:\emu8086\examples</p> <table border="1"><tr><td>Z</td></tr><tr><td>r</td></tr></table> <div></div>	Z	r
Z				
r				
REPNE	chain instruction	<p>Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 0 (result is Not Equal), maximum CX times.</p> <p>Algorithm:</p> <p>check_cx:</p> <p>if CX <> 0 then</p> <ul style="list-style-type: none">do following <u>chain instruction</u>CX = CX - 1if ZF = 0 then:		

8086 assembler tutorial for beginners



		<ul style="list-style-type: none"> • go back to check_cx <p>else</p> <ul style="list-style-type: none"> • exit from REPNE cycle <p>else</p> <ul style="list-style-type: none"> • exit from REPNE cycle <div style="border: 1px solid black; padding: 2px; display: inline-block;">Z</div> <div style="border: 1px solid black; padding: 2px; display: inline-block;">r</div> 
REPZ	chain instruction	<p>Repeat following CMPSB, CMPSW, SCASB, SCASW instructions while ZF = 0 (result is Not Zero), maximum CX times.</p> <p>Algorithm:</p> <p>check_cx:</p> <p>if CX <> 0 then</p> <ul style="list-style-type: none"> • do following <u>chain instruction</u> • CX = CX - 1 • if ZF = 0 then: <ul style="list-style-type: none"> • go back to check_cx <p>else</p> <ul style="list-style-type: none"> • exit from REPZ cycle <p>else</p> <ul style="list-style-type: none"> • exit from REPZ cycle <div style="border: 1px solid black; padding: 2px; display: inline-block;">Z</div> <div style="border: 1px solid black; padding: 2px; display: inline-block;">r</div> 
REPZ	chain instruction	Repeat following CMPSB, CMPSW, SCASB, SCASW

8086 assembler tutorial for beginners




		<p>instructions while ZF = 1 (result is Zero), maximum CX times.</p> <p>Algorithm:</p> <p>check_cx:</p> <p>if CX <> 0 then</p> <ul style="list-style-type: none">• do following <u>chain instruction</u>• CX = CX - 1• if ZF = 1 then:<ul style="list-style-type: none">• go back to check_cxelse<ul style="list-style-type: none">• exit from REPZ cycle <p>else</p> <ul style="list-style-type: none">• exit from REPZ cycle <div><div>Z</div><div>r</div></div>
RET	No operands or even immediate	<p>Return from near procedure.</p> <p>Algorithm:</p> <ul style="list-style-type: none">• Pop from stack:<ul style="list-style-type: none">• IP• if <u>immediate</u> operand is present: SP = SP + operand <p>Example:</p> <p>ORG 100h ; for COM file.</p> <p>CALL p1</p>



8086 assembler tutorial for beginners



		<p>ADD AX, 1</p> <p>RET ; return to OS.</p> <p>p1 PROC ; procedure declaration.</p> <p>MOV AX, 1234h</p> <p>RET ; return to caller.</p> <p>p1 ENDP</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> 	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
RETF	No operands or even immediate	<p>Return from Far procedure.</p> <p>Algorithm:</p> <ul style="list-style-type: none">• Pop from stack:<ul style="list-style-type: none">• IP• CS• if <u>immediate</u> operand is present: SP = SP + operand <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> 	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
ROL	memory, immediate REG, immediate memory, CL REG, CL	<p>Rotate operand1 left. The number of rotates is set by operand2.</p> <p>Algorithm:</p> <p>shift all bits left, the bit that goes off is set to CF and the same bit is inserted to the right-most position.</p>												

8086 assembler tutorial for beginners

		<p>Example:</p> <p>MOV AL, 1Ch ; AL = 00011100b</p> <p>ROL AL, 1 ; AL = 00111000b, CF=0.</p> <p>RET</p> <table border="1"><tr><td>C</td><td>O</td></tr><tr><td>r</td><td>r</td></tr></table> <p>OF=0 if first operand keeps original sign.</p>	C	O	r	r	
C	O						
r	r						
ROR	<p>memory, immediate REG, immediate</p> <p>memory, CL REG, CL</p>	<p>Rotate operand1 right. The number of rotates is set by operand2.</p> <p>Algorithm:</p> <p>shift all bits right, the bit that goes off is set to CF and the same bit is inserted to the left-most position.</p> <p>Example:</p> <p>MOV AL, 1Ch ; AL = 00011100b</p> <p>ROR AL, 1 ; AL = 00001110b, CF=0.</p> <p>RET</p> <table border="1"><tr><td>C</td><td>O</td></tr><tr><td>r</td><td>r</td></tr></table> <p>OF=0 if first operand keeps original sign.</p>	C	O	r	r	
C	O						
r	r						
SAHF	No operands	<p>Store AH register into low 8 bits of Flags register.</p> <p>Algorithm:</p> <p>flags register = AH</p> <p>AH bit: 7 6 5 4 3 2 1 0</p> <p>[SF] [ZF] [0] [AF] [0] [PF] [1] [CF]</p>					



		<p>bits 1, 3, 5 are reserved.</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table> <div></div>	C	Z	S	O	P	A	r	r	r	r	r	r
C	Z	S	O	P	A									
r	r	r	r	r	r									
SAL	<p>memory, immediate REG, immediate</p> <p>memory, CL REG, CL</p>	<p>Shift Arithmetic operand1 Left. The number of shifts is set by operand2.</p> <p>Algorithm:</p> <ul style="list-style-type: none">Shift all bits left, the bit that goes off is set to CF.Zero bit is inserted to the right-most position. <p>Example:</p> <p>MOV AL, 0E0h ; AL = 11100000b</p> <p>SAL AL, 1 ; AL = 11000000b, CF=1.</p> <p>RET</p> <table><tr><td>C</td><td>O</td></tr><tr><td>r</td><td>r</td></tr></table> <p>OF=0 if first operand keeps original sign.</p> <div></div>	C	O	r	r								
C	O													
r	r													
SAR	<p>memory, immediate REG, immediate</p> <p>memory, CL REG, CL</p>	<p>Shift Arithmetic operand1 Right. The number of shifts is set by operand2.</p> <p>Algorithm:</p> <ul style="list-style-type: none">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. <p>Example:</p> <p>MOV AL, 0E0h ; AL = 11100000b</p> <p>SAR AL, 1 ; AL = 11110000b, CF=0.</p>												



8086 assembler tutorial for beginners




		<div>MOV BL, 4Ch ; BL = 01001100b</div> <div>SAR BL, 1 ; BL = 00100110b, CF=0.</div> <div>RET</div> <div><table><tr><td>C</td><td>O</td></tr><tr><td>r</td><td>r</td></tr></table></div> <div>OF=0 if first operand keeps original sign.</div> <div></div>	C	O	r	r								
C	O													
r	r													
SBB	<div>REG, memory</div> <div>memory, REG</div> <div>REG, REG</div> <div>memory, immediate</div> <div>REG, immediate</div>	<div>Subtract with Borrow.</div> <div>Algorithm:</div> <div>operand1 = operand1 - operand2 - CF</div> <div>Example:</div> <div>STC</div> <div>MOV AL, 5</div> <div>SBB AL, 3 ; AL = 5 - 3 - 1 = 1</div> <div>RET</div> <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table></div> <div></div>	C	Z	S	O	P	A	r	r	r	r	r	r
C	Z	S	O	P	A									
r	r	r	r	r	r									
SCASB	No operands	<div>Compare bytes: AL from ES:[DI].</div> <div>Algorithm:</div> <div><ul style="list-style-type: none">AL - ES:[DI]set flags according to result: OF, SF, ZF, AF, PF, CF</div>												




8086 assembler tutorial for beginners

		<ul style="list-style-type: none">if $DF = 0$ then<ul style="list-style-type: none">$DI = DI + 1$else<ul style="list-style-type: none">$DI = DI - 1$ <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table> 	C	Z	S	O	P	A	r	r	r	r	r	r
C	Z	S	O	P	A									
r	r	r	r	r	r									
SCASW	No operands	<p>Compare words: AX from ES:[DI].</p> <p>Algorithm:</p> <ul style="list-style-type: none">$AX - ES:[DI]$set flags according to result: OF, SF, ZF, AF, PF, CFif $DF = 0$ then<ul style="list-style-type: none">$DI = DI + 2$else<ul style="list-style-type: none">$DI = DI - 2$ <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table> 	C	Z	S	O	P	A	r	r	r	r	r	r
C	Z	S	O	P	A									
r	r	r	r	r	r									
SHL	memory, immediate REG, immediate memory, CL REG, CL	<p>Shift operand1 Left. The number of shifts is set by operand2.</p> <p>Algorithm:</p> <ul style="list-style-type: none">Shift all bits left, the bit that goes off is set to CF.Zero bit is inserted to the right-most position. <p>Example:</p> <p>MOV AL, 11100000b</p>												

		<p>SHL AL, 1 ; AL = 11000000b, CF=1.</p> <p>RET</p> <table><tr><td>C</td><td>O</td></tr><tr><td>r</td><td>r</td></tr></table> <p>OF=0 if first operand keeps original sign.</p> <div></div>	C	O	r	r
C	O					
r	r					
SHR	<p>memory, immediate REG, immediate</p> <p>memory, CL REG, CL</p>	<p>Shift operand1 Right. The number of shifts is set by operand2.</p> <p>Algorithm:</p> <ul style="list-style-type: none">Shift all bits right, the bit that goes off is set to CF.Zero bit is inserted to the left-most position. <p>Example:</p> <p>MOV AL, 00000111b</p> <p>SHR AL, 1 ; AL = 00000011b, CF=1.</p> <p>RET</p> <table><tr><td>C</td><td>O</td></tr><tr><td>r</td><td>r</td></tr></table> <p>OF=0 if first operand keeps original sign.</p> <div></div>	C	O	r	r
C	O					
r	r					
STC	No operands	<p>Set Carry flag.</p> <p>Algorithm:</p> <p>CF = 1</p> <table><tr><td>C</td></tr><tr><td>1</td></tr></table>	C	1		
C						
1						



				
STD	No operands	<p>Set Direction flag. SI and DI will be decremented by chain instructions: CMPSB, CMPSW, LODSB, LODSW, MOVSb, MOVSW, STOSB, STOSW.</p> <p>Algorithm:</p> <p>DF = 1</p> <table border="1"><tr><td>D</td></tr><tr><td>1</td></tr></table> 	D	1
D				
1				
STI	No operands	<p>Set Interrupt enable flag. This enables hardware interrupts.</p> <p>Algorithm:</p> <p>IF = 1</p> <table border="1"><tr><td>I</td></tr><tr><td>1</td></tr></table> 	I	1
I				
1				
STOSB	No operands	<p>Store byte in AL into ES:[DI]. Update DI.</p> <p>Algorithm:</p> <ul style="list-style-type: none">• ES:[DI] = AL• if DF = 0 then<ul style="list-style-type: none">• DI = DI + 1else<ul style="list-style-type: none">• DI = DI - 1 <p>Example:</p>		

8086 assembler tutorial for beginners



		<p>ORG 100h</p> <p>LEA DI, a1</p> <p>MOV AL, 12h</p> <p>MOV CX, 5</p> <p>REP STOSB</p> <p>RET</p> <p>a1 DB 5 dup(0)</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
STOSW	No operands	<p>Store word in AX into ES:[DI]. Update DI.</p> <p>Algorithm:</p> <ul style="list-style-type: none">• ES:[DI] = AX• if DF = 0 then<ul style="list-style-type: none">• DI = DI + 2else<ul style="list-style-type: none">• DI = DI - 2 <p>Example:</p> <p>ORG 100h</p> <p>LEA DI, a1</p> <p>MOV AX, 1234h</p>												




8086 assembler tutorial for beginners

		<p>MOV CX, 5</p> <p>REP STOSW</p> <p>RET</p> <p>a1 DW 5 dup(0)</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table>	C	Z	S	O	P	A	unchanged						
C	Z	S	O	P	A										
unchanged															
SUB	<p>REG, memory</p> <p>memory, REG</p> <p>REG, REG</p> <p>memory, immediate</p> <p>REG, immediate</p>	<p>Subtract.</p> <p>Algorithm:</p> <p>operand1 = operand1 - operand2</p> <p>Example:</p> <p>MOV AL, 5</p> <p>SUB AL, 1 ; AL = 4</p> <p>RET</p> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td><td>r</td></tr></table>	C	Z	S	O	P	A	r	r	r	r	r	r	
C	Z	S	O	P	A										
r	r	r	r	r	r										
TEST	<p>REG, memory</p> <p>memory, REG</p> <p>REG, REG</p> <p>memory, immediate</p> <p>REG, immediate</p>	<p>Logical AND between all bits of two operands for flags only. These flags are effected: ZF, SF, PF. Result is not stored anywhere.</p> <p>These rules apply:</p>													


8086 assembler tutorial for beginners

		<div>1 AND 1 = 1 1 AND 0 = 0 0 AND 1 = 0 0 AND 0 = 0</div> <div>Example: MOV AL, 00000101b TEST AL, 1 ; ZF = 0. TEST AL, 10b ; ZF = 1. RET</div> <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td></tr><tr><td>0</td><td>r</td><td>r</td><td>0</td><td>r</td></tr></table></div> <div></div>	C	Z	S	O	P	0	r	r	0	r		
C	Z	S	O	P										
0	r	r	0	r										
XCHG	REG, memory memory, REG REG, REG	<div>Exchange values of two operands.</div> <div>Algorithm: operand1 < - > operand2</div> <div>Example: MOV AL, 5 MOV AH, 2 XCHG AL, AH ; AL = 2, AH = 5 XCHG AL, AH ; AL = 5, AH = 2 RET</div> <div><table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table></div> <div></div>	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
XLATB	No operands	<div>Translate byte from table. Copy value of memory byte at DS:[BX + unsigned AL] to</div>												

		<p>AL register.</p> <p>Algorithm:</p> <p>AL = DS:[BX + unsigned AL]</p> <p>Example:</p> <p>ORG 100h</p> <p>LEA BX, dat</p> <p>MOV AL, 2</p> <p>XLATB ; AL = 33h</p> <p>RET</p> <p>dat DB 11h, 22h, 33h, 44h, 55h</p> <table border="1"><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td colspan="6">unchanged</td></tr></table> 	C	Z	S	O	P	A	unchanged					
C	Z	S	O	P	A									
unchanged														
XOR	REG, memory memory, REG REG, REG memory, immediate REG, immediate	<p>Logical XOR (Exclusive OR) between all bits of two operands. Result is stored in first operand.</p> <p>These rules apply:</p> <p>1 XOR 1 = 0 1 XOR 0 = 1 0 XOR 1 = 1 0 XOR 0 = 0</p> <p>Example:</p> <p>MOV AL, 00000111b</p>												



8086 assembler tutorial for beginners

		<div>XOR AL, 00000010b ; AL = 00000101b</div> <div>RET</div> <table><tr><td>C</td><td>Z</td><td>S</td><td>O</td><td>P</td><td>A</td></tr><tr><td>0</td><td>r</td><td>r</td><td>0</td><td>r</td><td>?</td></tr></table> <div></div>	C	Z	S	O	P	A	0	r	r	0	r	?
C	Z	S	O	P	A									
0	r	r	0	r	?									



Appendix B

Numbering Systems Tutorial

What is it?

There are many ways to represent the same numeric value. Long ago, humans used sticks to count, and later learned how to draw pictures of sticks in the ground and eventually on paper. So, the number 5 was first represented as: | | | | | (for five sticks).

Later on, the Romans began using different symbols for multiple numbers of sticks: | | | still meant three sticks, but a V now meant five sticks, and an X was used to represent ten of them.

Using sticks to count was a great idea for its time. And using symbols instead of real sticks was much better.

Decimal System

8086 assembler tutorial for beginners

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

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

These digits can represent any value, for example:
754.

The value is formed by the sum of each digit, multiplied by the base (in this case it is 10 because there are 10 digits in decimal system) in power of digit position (counting from zero):

$$7 \cdot 10^2 + 5 \cdot 10^1 + 4 \cdot 10^0 = 700 + 50 + 4 = 754$$

Position of each digit is very important! for example if you place "7" to the end:

547

it will be another value:

$$5 \cdot 10^2 + 4 \cdot 10^1 + 7 \cdot 10^0 = 500 + 40 + 7 = 547$$

Important note: any number in power of zero is 1, even zero in power of zero is 1:

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

Binary System

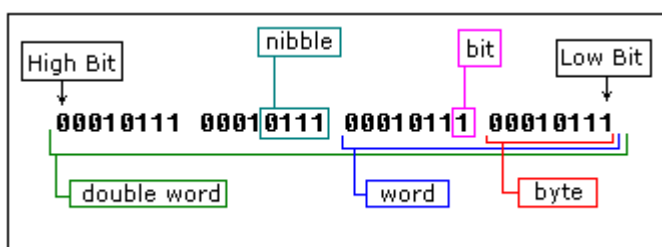
Computers are not as smart as humans are (or not yet), it's easy to make an electronic machine with two states: on and off, or 1 and 0.

Computers use binary system, binary system uses 2 digits:

0, 1

And thus the base is 2.

Each digit in a binary number is called a BIT, 4 bits form a NIBBLE, 8 bits form a BYTE, two bytes form a WORD, two words form a DOUBLE WORD (rarely used):



There is a convention to add "b" in the end of a binary number, this way we can determine that 101b is a binary number with decimal value of 5.

The binary number 10100101b equals to decimal value of 165:

$$\begin{aligned} 10100101b &= \\ &= 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 128 + 0 + 32 + 0 + 0 + 4 + 0 + 1 = 165 \end{aligned}$$

(decimal value)

The diagram includes two annotations: a red box labeled 'base' with a line pointing to the '2' in the first term of the equation, and a green box labeled 'digit position' with a line pointing to the '3' in the fifth term of the equation.

Hexadecimal System

Hexadecimal System uses 16 digits:

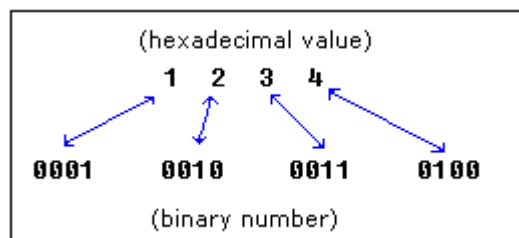
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

And thus the base is 16.

Hexadecimal numbers are compact and easy to read.

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

Decimal (base 10)	Binary (base 2)	Hexadecimal (base 16)
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B



8086 assembler tutorial for beginners

12	1100	C
13	1101	D
14	1110	E
15	1111	F

There is a convention to add "h" in the end of a hexadecimal number, this way we can determine that 5Fh is a hexadecimal number with decimal value of 95.

We also add "0" (zero) in the beginning of hexadecimal numbers that begin with a letter (A..F), for example 0E120h.

The hexadecimal number 1234h is equal to decimal value of 4660:

$$1 \cdot 16^3 + 2 \cdot 16^2 + 3 \cdot 16^1 + 4 \cdot 16^0 = 4096 + 512 + 48 + 4 = 4660$$

(decimal value)

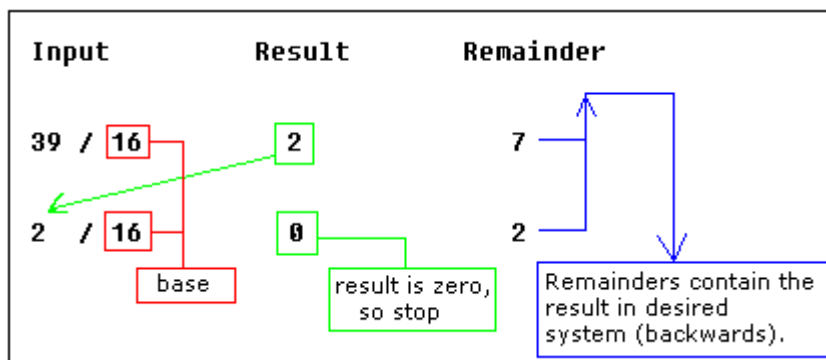
Converting from Decimal System to Any Other

In order to convert from decimal system, to any other system, it is required to divide the decimal value by the base of the desired system, each time you should remember the result and keep the remainder, the divide process continues until the result is zero.

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

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

8086 assembler tutorial for beginners

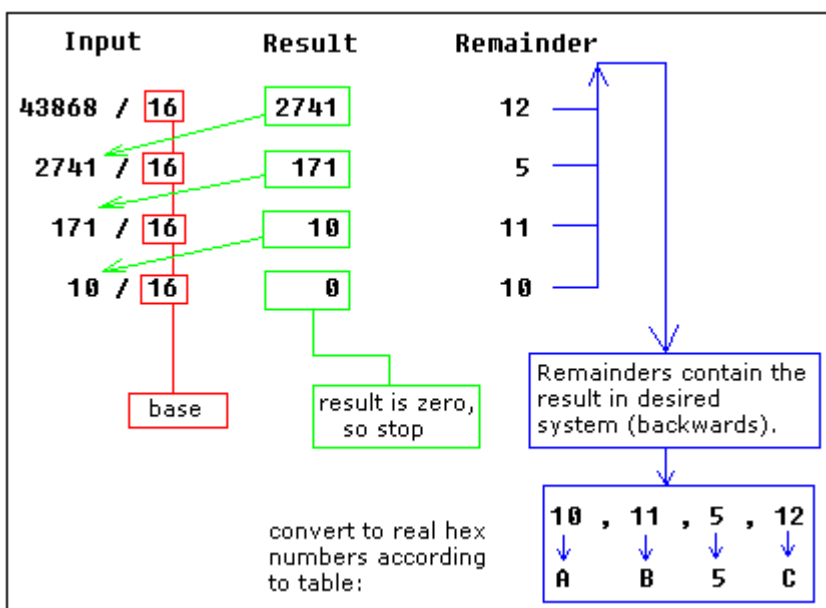


As you see we got this hexadecimal number: 27h.

All remainders were below 10 in the above example, so we do not use any letters.

Here is another more complex example:

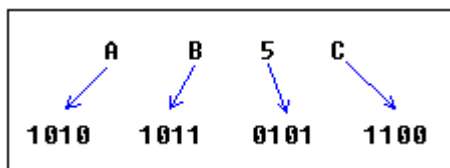
let's convert decimal number 43868 to hexadecimal form:



The result is 0AB5Ch, we are using [the above table](#) to convert remainders over 9 to corresponding letters.

Using the same principle we can convert to binary form (using 2 as the divider), or convert to hexadecimal number, and then convert it to binary

number using [the above table](#):



As you see we got this binary number: 1010101101011100b

Signed Numbers

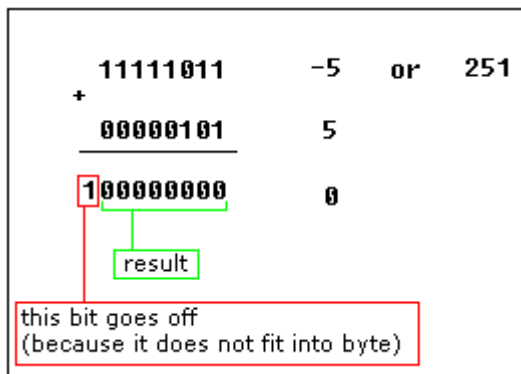
There is no way to say for sure whether the hexadecimal byte 0FFh is positive or negative, it can represent both decimal value "255" and "- 1".

8 bits can be used to create 256 combinations (including zero), so we simply presume that first 128 combinations (0..127) will represent positive numbers and next 128 combinations (128..256) will represent negative numbers.

In order to get "- 5", we should subtract 5 from the number of combinations (256), so it we'll get: $256 - 5 = 251$.

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

8086 assembler tutorial for beginners

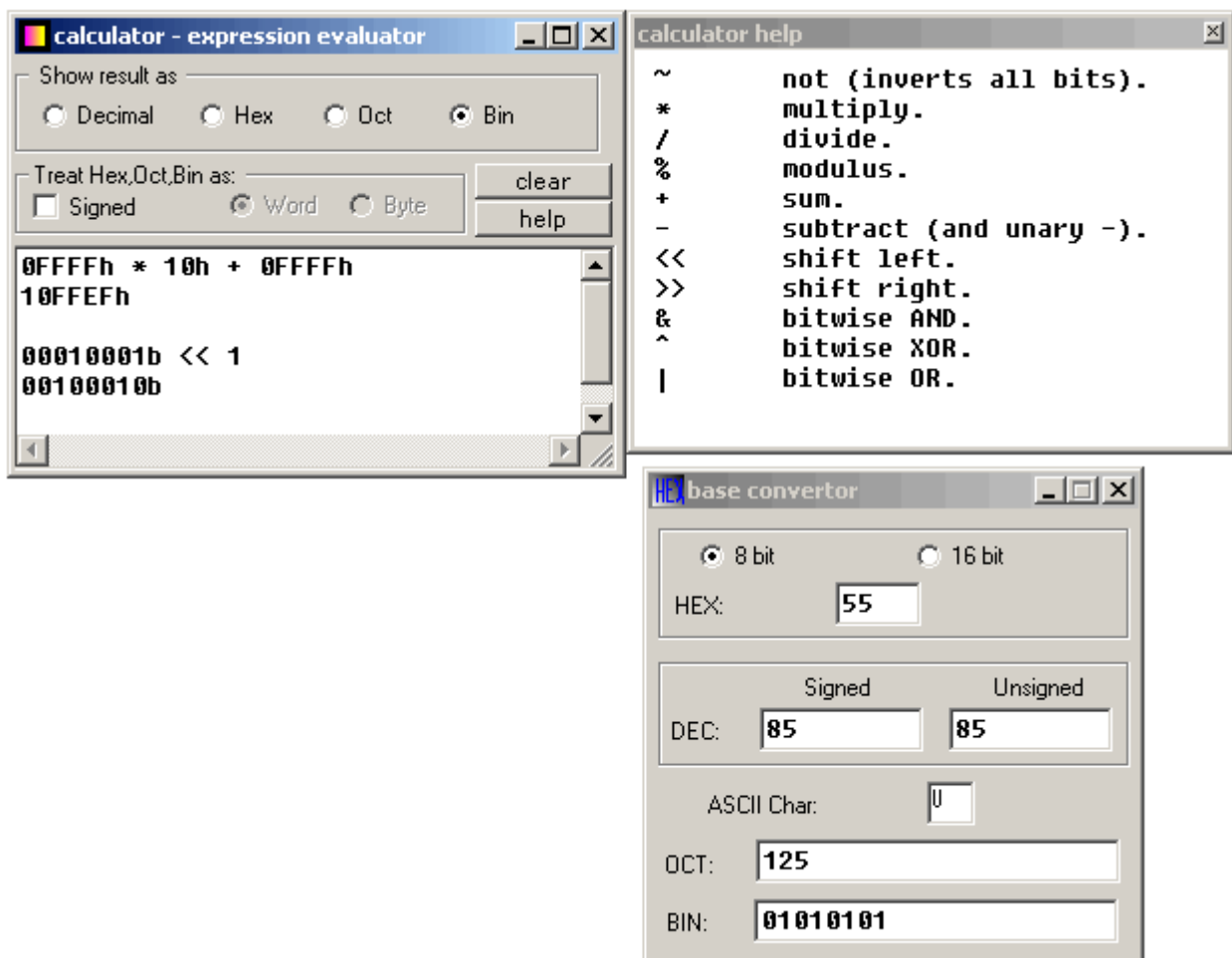


When combinations 128..256 are used the high bit is always 1, so this maybe used to determine the sign of a number.

The same principle is used for words (16 bit values), 16 bits create 65536 combinations, first 32768 combinations (0..32767) are used to represent positive numbers, and next 32768 combinations (32767..65535) represent negative numbers.

There are some handy tools in emu8086 to convert numbers, and make calculations of any numerical expressions, all you need is a click on Math menu:

8086 assembler tutorial for beginners



Base converter allows you to convert numbers from any system and to any system. Just type a value in any text-box, and the value will be automatically converted to all other systems. You can work both with 8 bit and 16 bit values.

Multi base calculator can be used to make calculations between numbers in different systems and convert numbers from one system to another. Type an expression and press enter, result will appear in chosen numbering system. You can work with values up to 32 bits.

When Signed is checked evaluator assumes that all values (except decimal and double words) should be treated as signed. Double words are always treated as signed values, so 0FFFFFFFFh is converted to -1. For example you want to calculate: 0FFFFh * 10h +

8086 assembler tutorial for beginners

0FFFFh (maximum memory location that can be accessed by 8086 CPU). If you check Signed and Word you will get -17 (because it is evaluated as $(-1) * 16 + (-1)$). To make calculation with unsigned values uncheck Signed so that the evaluation will be $65535 * 16 + 65535$ and you should get 1114095.

You can also use the base converter to convert non-decimal digits to signed decimal values, and do the calculation with decimal values (if it's easier for you).

These operation are supported:

- ~ not (inverts all bits).
- * multiply.
- / divide.
- % modulus.
- + sum.
- subtract (and unary -).
- << shift left.
- >> shift right.
- & bitwise AND.
- ^ bitwise XOR.
- | bitwise OR.

Binary numbers must have "b" suffix, example:
00011011b

Hexadecimal numbers must have "h" suffix, and start with a zero when first digit is a letter (A..F), example:
0ABCDh

Octal (base 8) numbers must have "o" suffix, example:77o

