Introduction to Machine- and Assembly-Language Programming

Prepared for Tynan
(and other crazy people)

by Prof. Dr. Brad Richards
University of Applied Sciences, NW Switzerland
Institute for Business Information Systems
November 2012

Table of Contents

Chapter 1 – Getting Started

All of the familiar computer languages are compiled or interpreted languages. The statements in these languages are “high level” statements that must be translated into the binary language of the machine. A single high-level statement may turn into dozens of machine-language commands (called “opcodes”).

We can program directly in binary, in “machine language”, which is fun for those of us who are slightly crazy. Some programs really are developed at this low level, but the programmers use “assembly language”, which lets them use names rather than numbers, and helps in other ways as well. We will start out with machine language, and then move on to assembly language.

A few important notes:

- This tutorial assumes that you are working under Windows.
- When discussing binary numbers, we always use hexadecimal. The hexadecimal number 13 is the decimal value nineteen! In assembly language, hexadecimal numbers are marked with an 'h', for example, “13h”..
- We will be writing 16-bit programs, as these are somewhat simpler that 32-bit or 64-bit programs. We will execute the commands in the command-line console.

  Important: if you are running a 64-bit version of windows, you will be unable to run 16-bit programs. You will need to either run Windows XP compatibility mode, or else download the free 16-bit emulator DOSBox. See the references at the end of this document for links.
- Since we are working “old fashioned”, it is important that all of your directories and files abide by the 8.3 naming convention. No names with more than 8 characters!

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del

Hello, World!

We will start by writing “Hello, World!”

In order to do this, you will need a hex editor. Under Windows, you can get a good, free program from HHD software: http://www.hhdsoftware.com/free-hex-editor

You should also have an ASCII table handy. There are lots of these online, for example, http://www.asciitable.com/. You'll also find one on the right. The letter 'H' is represented by the binary (hex) value 48.

Our programs will use the old-fashioned but simple “.com” format for executable files.

This format contains nothing but code and data. The executable file is loaded into memory, beginning at memory address 0100.

In order to write “Hello, World!”, we will carry out the steps listed below. Don't worry too much about the details yet; explanations of registers and instructions will be coming soon!

1. Load the location of the string into the CPU register “dx”.
2. Load the number of the DOS service that prints a string to the console into register “ah”. This is service number 9 “print string”
3. Interrupt: causes DOS to do something, in this case, to execute the “print string” command
4. Load the exit command 4c, with error code 00 into register “ax”
5. Interrupt: exits the program, returns to the command line

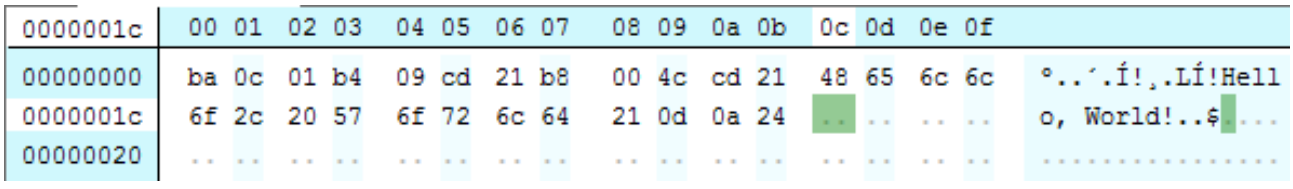
Here are the steps, shown both in assembly language and in machine code. The memory addresses are on the left. Note: The first memory address shown is 0100, because this is where the program will be loaded into memory. However, you will enter this code in the hex-editor beginning at address 0000 in the file.

Address	***Assembly***	***Machine code***	***Comment***
`0100`	`mov dx, 010ch`	`ba 0c 01`	Location of string into dx
`0103`	`mov ah, 09`	`b4 09`	DOS command 09 into ah
`0105`	`int 21h`	`cd 21`	Interrupt
`0107`	`mov ax, 4c00h`	`b8 00 4c`	Exit, error code 0
`010A`	`int 21h`	`cd 21`	Interrupt
`010C`	`db 'Hello, World!', '$'`	`48 65 6c 6c 6f 2c 20 57 6f 72 6c 64 21 0d 0a 24`	The string “Hello, World!”

Look at the second and third bytes in the program “0c01”. These represent the address of the string we want to print, which is 010c. For ancient historical reasons, addresses are entered with the bytes reversed, so that the least-significant byte is first[1]. Hence, this represents the memory address 010c. The same thing occurs when we load the register ax with the DOS command “4c” and the error code 00: the bytes are reversed in the actual machine code.

The string “Hello, World!” ends with a new-line. In Windows/DOS a new-line always consists of two characters: a “line-feed” and a “return”. Finally, the DOS-service to print text looks for a '$' to tell it when to stop printing; hence, this is the last character in the string.

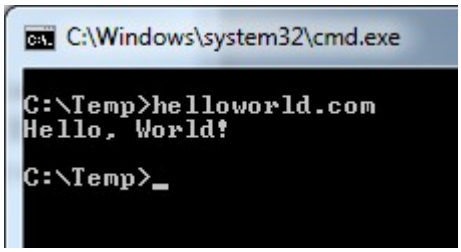
Once entered into the hex editor, you should see something like this:



1 This is called “little-endian”. If we were storing a 32-bit value, the pattern continues: The value a5b6c7d8 would be stored in reverse: d8-c7-b6-a5.

Save the file as “helloworld.com”.

When you run the program in a command-line window, you should see the result shown an the right:



That's your first program!

Understanding the x86 processor

Before we continue, you need to know something about the structure of the x86 processor. Processors read instructions from memory, and execute these instructions on data. The data that the instructions use is generally held in registers. So, before we can understand the instructions, we need to look at the registers, and then at the different ways of addressing data: immediate values, in registers or in memory.

The original x86 processors were 8-bit processors; later, they grew to 16-bit, then 32-bit, and today 64-bit. For this reason, even with today's 64-bit processors, we can still address 8-bit, 16-bit and 32-bit registers. The programs that we will write are essentially 16-bit programs.

Registers

Registers are locations where the processor can quickly access values, and where it can store results. We will be using the “general purpose” registers for our exercises. There are also several special registers, but we won't be using them.

Here is an overview of the general-purpose registers:

Register	Accumulator		Counter		Data		Base		Stack Pointer	Stack Base Pointer	Source	Destination
16-bit	AX	AL AH	CX	CL CH	DX	DL DH	BX	BL BH	SP	BP	SI	DI
32-bit	EAX		ECX		EDX		EBX		ESP	EBP	ESI	EDI

Look at the “Accumulator”. In 64-bit mode, the accumulator is called RAX. We can address just the lower 32-bits (compatible with 32-bit processors) as EAX. We can address just the lowest 16-bits as AX. Within the lowest 16-bits, we can address the least-significant byte as AL, and the most-significant byte as AH.

While these registers are “general purpose”, each of them is normally used for particular purposes, and this is reflected in the instruction that we will see in the next section. The registers that we will use:

- Accumulator: Arithmetic operations
- Counter: Counting and loops
- Data: Arithmetic operands, also input/output
- Base: Pointer to data

The registers are numbered; these numbers will be important for our machine code. The numbers we need are shown in the table below:

Number	*8-bit registers*	*16-bit registers*	*32-bit registers*
0	AL	AX	EAX
1	CL	CX	ECX
2	DL	DX	EDX
3	BL	BX	EBX
4	AH	SP	ESP
5	CH	BP	EBP
6	DH	SI	ESI
7	BH	DI	EDI

Note that these numbers are the same as the order (from left to right) in the first picture.

As an example, consider the very first instruction in our “Hello, World!” program. The instruction to move 16-bits of memory into a 16-bit register is specified as “B8 + rw” (see page 3-402 of the “Intel Architecture software Developer's Manual” in the references). This means that the actual instruction is B8 + register-number. In this case, we want to store the value in DX, which is register 2, so the actual instruction is B8+2 = BA.

Here's a short program to try these out: We have the string “abcd” in memory. We load the letter 'a' into the accumulator, add 5 to it, and then store the result back in the original location. When we print the string, we should see the result “fbcd”.

Address	*Assembly*	*Machine code*	*Comment*
`0100`	`mov al,[0114h]`	`a0 14 01`	First char of string into al
`0103`	`add al,5`	`04 05`	Add 5 to al
`0105`	`mov [0114h],al`	`a2 14 01`	Copy al to first char of string
`0108`	`mov dx,0114h`	`ba 14 01`	Location of string into dx
`010B`	`mov ah,09h`	`b4 09`	DOS command 09 into ah
`010D`	`int 21h`	`cd 21`	Interrupt
`010F`	`mov ax,4c00h`	`b8 00 4c`	Exit, error code 0
`0112`	`int 21h`	`cd 21`	Interrupt
`0114`	`db 'abcd',24h`	`61 62 63 64 0d 0a 24`	The string “abcd”

Addressing modes

When we want to access a particular value, this value may be stored in four possible ways.

- **Immediate addressing.** The value is part of the instruction, like “5” in the command “add al,5”.
- **Register addressing.** The value is in a register, like “al” in the command “add al,5”.
- **Memory addressing.** The value is stored at a particular memory address, as when we load the value stored at 0104 in the instruction “mov al,[0104h]

- **Register indirect addressing.** The value is at a particular memory address, and this memory address is in a register. We haven't seen this yet, but the assembly code would be something like “mov al,[dx]”.

The addressing mode generally modifies the opcode. Consider the “mov” instruction, as described in the Intel manual:

Opcode	Instruction	Description
88 /r	MOV r/m8,r8	Move r8 to r/m8
89 /r	MOV r/m16,r16	Move r16 to r/m16
89 /r	MOV r/m32,r32	Move r32 to r/m32
8A /r	MOV r8,r/m8	Move r/m8 to r8
8B /r	MOV r16,r/m16	Move r/m16 to r16
8B /r	MOV r32,r/m32	Move r/m32 to r32
8C /r	MOV r/m16,Sreg**	Move segment register to r/m16
8E /r	MOV Sreg,r/m16**	Move r/m16 to segment register
A0	MOV AL,moffs8*	Move byte at (seg:offset) to AL
A1	MOV AX,moffs16*	Move word at (seg:offset) to AX
A1	MOV EAX,moffs32*	Move doubleword at (seg:offset) to EAX
A2	MOV moffs8*,AL	Move AL to (seg:offset)
A3	MOV moffs16*,AX	Move AX to (seg:offset)
A3	MOV moffs32*,EAX	Move EAX to (seg:offset)
B0+ rb	MOV r8,imm8	Move imm8 to r8
B8+ rw	MOV r16,imm16	Move imm16 to r16
B8+ rd	MOV r32,imm32	Move imm32 to r32
C6 /0	MOV r/m8,imm8	Move imm8 to r/m8
C7 /0	MOV r/m16,imm16	Move imm16 to r/m16
C7 /0	MOV r/m32,imm32	Move imm32 to r/m32

In the previous example, we used the “mov” instruction five times. The op-code for the “mov”-instruction depends on the addressing mode and (if applicable) the register number (as described in the previous section).

Assembly	*Addressing modes*	*Opcode calculation*	*Opcode*
`mov al,[0114h]`	8-bit memory to register	A0 + register-number for AL	a0
`mov [0114h],al`	8-bit register to memory	A2 + register-number for AL	a2
`mov dx,0114h`	16-bit immediate to register	B8 + register-number for DX	ba
`mov ah,09h`	8-bit immediate to register	B0 + register-number for AH	b4
`mov ax,4c00h`	16-bit immediate to register	B8 + register-number for AX	b8

Instructions

The main reference for the instruction set is the Intel Architecture Software Developer's Manual; the link is given in the references. The initial chapters give a more detailed explanation of the architecture than the overview above. Chapter 3 contains an alphabetical list of the available instructions.

We will get to know some of these instructions in the next chapters. After you have an idea how to write some simple programs, take the time to browse through the Intel manual and see what else you can do.

A reminder: we are writing our programs in 16-bit mode. Programming in 32-bit or 64-bit mode is beyond the scope of this tutorial. However, all of the same principles apply, and moving to these more modern modes is easy.

Assembly language

This tutorial is now going to use assembly language rather than machine language. For those who want to continue programming in machine language, you can continue to do so, just by translating the instructions into binary yourself. For the slighly saner people in the world, assembly language makes life a bit easier.

While most programs today are developed in higher-level languages, assembly language is still used for drivers, or for software that has to be very fast. Compilers for high-level languages can produce optimized code, but a good assembly-language programmer will beat them every time. Also: assembly-language programs tend to be very small. Some developers also just prefer working “close to the machine”; one well-known example is Steve Gibson of http://www.grc.com/, who writes all of his software in assembly.

To begin, you need to download an “assembler”. An assember is a program that translates assembly language into machine language. For our purposes here, we will use the “flat assembler”. We will continue working on the command-line. This means that you should download the version for DOS, available from http://flatassembler.net/.

Reminder: when you install FASM, be sure that none of the directories have names longer than 8 characters. It will be easiest just to install it at the top level, for example, in C:\fasm

Getting started with assembly language

To get started, let's reimplement the last example, where we change the string “abcd” into “fbcd” and then print it to the console. To write the source code, it is probably simplest to use Notepad. Alternatively, within a command-line window, you can use the “edit” command.

When we wrote the example in machine language, we also showed the assembly language next to it. *This assembly language was not quite complete!*

Assembly language is basically written in four columns. It's easiest to use 'tab' to move to these columns:

- **First column: labels.** Labels are used instead of memory addresses. When your program is compiled, the labels are replaced with the actual memory address – the assembler calculates these for you.
- **Second column: instructions.** The symbolic instructions that the assemblre will translate into opcodes.
- **Third column: operands.** The data or registers that the instructions operate on.
- **Fourth column: comments.** You do comment your code, don't you?

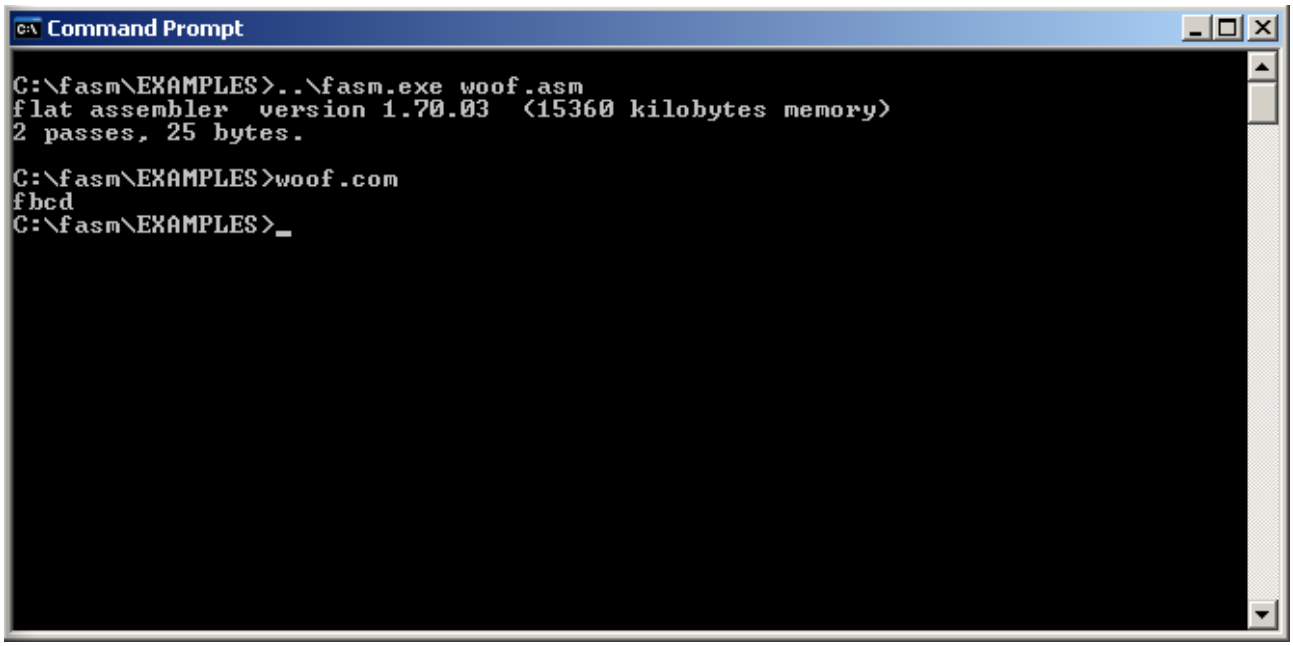
In the table below, the left column shows the example as shown before. The right column shows the program as we would actually write it in assembly language.

Previous example	*Complete assembly language*

	`; Alter the first character of the text "abcd",`
	`; then print the altered text to the console`
	
	`org 100h ; code starts at offset 100h`
	`use16 ; use 16-bit code`
	
	`; Program begins here`

`mov al,[0114h]`
`add al,5`
`mov [0114h],al`
`mov dx,0114h`
`mov ah,09h`
`int 21h`
`mov ax,4c00h`
`int 21h`
 	
`db 'abcd',24h`
	`mov al,[text] ; move first character to al`
	`add al,5 ; add 5 to first character`
	`mov [text],al ; put character back`
	`mov dx,text ; put address of string in dx`
	`mov ah,09h ; tell DOS to print string`
	`int 21h ; execute (interrupt)`
	`mov ax,4c00h ; tell DOS we want to exit`
	`int 21h ; execute (interrupt)`
	
	`text db 'abcd',24h ; the text`

Save the assembly language in a file with the extension ".asm". Run the assembler (fasm.exe) on this file; this will create a machine-code file with the extension ".com". You can then execute the ".com" file.



The flat-assembler comes with several nice programs in its "examples" directory. Have a look at these if you like. The next chapter will continue with some tips on implementing common programming structures.

Basic programming techniques

Before we get into programming, there are a few techniques that will help you, when you start writing machine-language programs.

Plan your program. Write out the process you will follow in pseudocode. One you are writing machine-language, it is easy to get lost in the details. You can use your plan to stay on track. For example:

- Read number from user
- Loop 5 times
- Double number
- End loop
- Print result

Place data after code. A memory address is a memory address. In principle, you can mix your code and data, but in practice this leads to unnecessary errors. Put all of your code together at the start of the file, and all of your data at the end.

Use subroutines. Once you start writing more complex programs, break the programs into small subroutines. Each subroutine should have a simple, clear task. We will see how to use subroutines in this section.

With that out of the way, let's look at some basic programming techniques...

The *if* statement

The equivalent of an “if” statement is a conditional-jump instruction. This instruction will either do nothing, allowing execution to continue with the next instruction, or else it will jump to a defined label. The conditional-jump decides what to do based on a previously-executing compare-instruction. Hence, we must do three things: (1) define a label, (2) compare two operands, and (3) perform a conditional-jump. The conditional jump instruction will jump to the label if the specified condition is met.

Note: the condition for the conditional-jump is the opposite of what we are used to in high-level languages. We are testing for the condition that will cause us to jump over the selected code

High level programming	***Assembly language programming***

`if (condition) then`
`  do this code`
`end if`
`program continues`
	`  make comparison`
	`  if (not condition) jump to` ***`continue`***
	`  do this code`
	
	`continue` `program continues`

In order to implement an if-then-else, we will require two labels, because we need to jump over two pieces of code.

High level programming	*Assembly language programming*

`if (condition is true) then`
`  do this code`
`else`
`  do other code`
`end if`
`program continues`
	`  make comparison`
	`  if (not condition) jump to else`
	`  do this code`
	`  jump to continue`
	`else  do other code`
	
	`continue  program continues`

For example, continuing our example from earlier, let's suppose that we have two strings and an integer constant. If the integer has the value 1, we print the first string, otherwise we print the second string. This requires the if-then-else pattern. Here's the program:

```
; Alter the first character of the string "abcd",
; then print the altered string to the console

        org     100h    ; code starts at offset 100h
        use16           ; use 16-bit code

which_text = 2

; Program begins here

        mov     al, 1           ; we will compare to 1
        cmp     al, which_text  ; compare to which_text
        jne     print2          ; if not-equal, jump to print2
        mov     dx,text1        ; put address of text1 in dx
        jmp     continue        ; jump to continue
print2:
        mov     dx,text2        ; put address of text2 in dx
continue:
        mov     ah,09h          ; tell DOS to print string
        int     21h             ; execute (interrupt)
        mov     ax,4c00h        ; tell DOS we want to exit
        int     21h             ; execute (interrupt)

text1   db 'first text',24h     ; text1
text2   db 'second text',24h    ; text2
```

Note that we can define constants in assembly language; the assembler puts the value into the machine code whenever we use the constant.

Look in the Intel manual to see what kinds of comparisons are possible (page 3-76), and what kinds of conditional jumps (page 3-329).

The *for* loop

Implementing a “for” loop is quite similar to implementing an “if”. The only difference is that you must have a counter somewhere. The C-registers are commonly used for this task.

The program below implements a loop using decrement, compare and jump instructions.

```
; Print a text five times
; Loop using compare, decrement and jump instructions

      org   100h  ; code starts at offset 100h
      use16       ; use 16-bit code

; Program begins here

      mov   dx,text          ; put address of text1 in dx
      mov   ah,09h           ; tell DOS to print string
      mov   cx,05h           ; set cx to 5
again:
      int   21h         ; execute print command
      dec   cx          ; decrement counter
      cmp   cx,0        ; compare to 0
      jg    again       ; loop if counter > 0

      mov   ax,4c00h    ; tell DOS we want to exit
      int   21h         ; execute (interrupt)

text  db 'repeat 5 times',0dh,0ah,24h
```

However, x86 assembly language also provide a loop-instruction, which saves us a bit of work. The loop instruction combines the decrement, compare and jump instructions of the program above. Here is what it looks like:

```
; Print a text five times
; Loop using compare, decrement and jump instructions

      org   100h  ; code starts at offset 100h
      use16       ; use 16-bit code

; Program begins here

      mov   dx,text     ; put address of text1 in dx
      mov   ah,09h      ; tell DOS to print string
      mov   cx,05h      ; set cx to 5
again:
      int   21h         ; execute print command
      loop  again       ; decrement, loop if cx > 0

      mov   ax,4c00h    ; tell DOS we want to exit
      int   21h         ; execute (interrupt)

text  db 'repeat 5 times',0dh,0ah,24h
```

Reading user input

We have used the “int” (interrupt) instruction without really knowing what it does. Our programs are running in DOS mode, and DOS provides a large set of interrupt-driven ser-

vices. A complete list can be found on Wikipedia (and Google is your friend); links in the references.

In order to read user input, we are going to use another of these services. Service 1 reads one character from the user and places it in register “al”.

The following program is an adaptation of the earlier example, where we print one of two differents texts. In this new program, we will decide which text to print, based on the first character that the user enters. If the character is '1', we print the first text, otherwise we print the second text.

Question: Why do we compare the character to the value 31h?

```
; Alter the first character of the string "abcd",
; then print the altered string to the console

      org   100h        ; code starts at offset 100h
      use16             ; use 16-bit code

read_char = 01h         ; read-character command
print_string = 09h      ; print-string command

; Program begins here

      mov   ah,read_char      ; tell DOS to read char
      int   21h               ; execute
      cmp   al, 31h           ; compare
      jne   print2            ; if not-equal, jump to print2
      mov   dx,text1          ; put address of text1 in dx
      jmp   continue          ; jump to continue
print2:
      mov   dx,text2          ; put address of text2 in dx
continue:
      mov   ah,print_string   ; tell DOS to print string
      int   21h               ; execute
      int   20h               ; terminate program

text1 db 'first text',24h
text2 db 'second text',24h
```

Look at the last instruction in this program. As it turns out, there are several ways to terminate a program. We have been using the “int 21” service 4c, which terminates with a return code. We can also use the “int 21” service 0, which terminates with no return code. Finally, we can also use the “int 20” service, which also terminate the program, without requiring any additional parameter.

Subroutines

As your program grows larger, you will want to define and use subroutines. A subroutine is what a Java programmer would call a method: a piece of code that is called from other places in the program, and that returns some result.

Not yet finished...

An example

Some larger example...

Suggested exercises

How to implement...

References

1. Hex editor from HHD Software
   http://www.hhdsoftware.com/free-hex-editor
2. ASCII Table online
   http://www.asciitable.com/
3. Hello, World in assembly language (english)
   http://digiassn.blogspot.ch/2006/02/asm-hello-world.html
4. Hello, World in machine language (german)
   http://de.wikibooks.org/wiki/Maschinensprache_i8086/_Hallo_Welt
5. Overview of the x86 architecture
   http://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture
6. Intel Architecture Software Developer's Manual (instruction set reference)
   http://download.intel.com/design/intarch/manuals/24319101.pdf
7. Flat Assembler: an open-source assembler
   http://flatassembler.net/
8. Windows XP compability mode
   http://www.microsoft.com/windows/virtual-pc/download.aspx
9. DOSBox 16-bit emulator
   http://www.dosbox.com/
10. DOS INT 21h services
    http://en.wikipedia.org/wiki/MS-DOS_API#DOS_INT_21h_services
    http://spike.scu.edu.au/~barry/interrupts.html