

UNIVERSITY OF PUERTO RICO
MAYAGÜEZ CAMPUS

Department of Electrical and Computer Engineering

INEL 4206
8086 Microprocessors
Introduction to Assembly Programming

Prof. Ramón Vázquez
Prof. Vidya Manian
Assistant: Janice Morales

Edition 1999

Table of Contents

INTRODUCTION.....	1
(I) BASIC DOS COMMANDS.....	1
(II) USING THE ASSEMBLER AVAILABLE IN CRAI LABORATORIES.....	1
(III) SEGMENTS OF A PROGRAM.....	3
(IV) USING CODE VIEW.....	3
(V) AN EXAMPLE EXERCISE.....	4
(VI) USEFUL TOOLS.....	6
(VII) EXIT TO DOS.....	8
(VIII) INTERRUPTS.....	9
(IX) MODELS.....	11
(X) MACROS.....	11
(XI) USING DISK FILES.....	12
(XII) ASSEMBLER DATA DIRECTIVES.....	13
(XIII) EXERCISES.....	15
(XIV) REFERENCES.....	16
(XV) APPENDIX.....	17

Introduction

An Assembly Language program consists of a series of lines that are assembly language instructions. These instructions consist of a mnemonic, which is a command, and an operand, which is the data to be manipulated. The programs usually include comments which are written at the end of a line or in a separate line beginning with a ';' and are ignore by the assembler. This manual is created with the intention of guiding the INEL 4206 students on learning how to program using assembly language.

(I) Basic DOS Commands

To be able to work in a DOS environment some basic commands must be known. Some of these commands are:

cd..	to go up one directory level
cd directory_name	to access that specific directory
dir/w	to view content of directory
E:	changes to E directory
X:	to access your account
A:	to access floppy disc

(II) Using the assembler available in CRAI laboratories

In order to use the assembler available in CRAI laboratories follow these steps:

- 1st Write your program using notepad
- 2nd Save it as "name.asm", it is important to use the ""

After the program is properly written and saved the next step is to run it. For this purpose you need to access Command Prompt available in Programs. At this point you are accessing a DOS environment.

C:\> This is the prompt that appears

C:\>E: Change from C to E directory to access the assembler, type *E:*
E:\> This will now appear
E:\>dir/w To view content of directory E type *dir\w*
E:\>cd assem To access the assembler
E:\>ASSEM This will now appear

Once you have access the assembler you need to create an object and an executable file using MASM and LINK. These directives examine and run the program. If an error occurs the errors will be displayed along with the line number they are at. For program running follow these steps:

E:\ASSEM>cd masm Access MASM to create the object file
E:\ASSEM\MASM>masm Type *masm*

How to create the object file:

Source filename[.asm]:X:*.asm Type *X: and the name of your program.asm*
if the program is saved in your account
Object filename[.obj]:X:*.obj Type *X: the name of your program.obj*
Source Listing[NUL.LST]:↵ Type *enter*
Cross-Reference[NUL.CRF]: ↵ Type *enter*

You have just created the object file, at this point the errors and warnings are shown along with the line number. If there is no error the next step is to create the executable file, if there are errors you need to go to your program and fix it then create the object file again. To create the executable:

E:\ASSEM\MASM>link Type *link*
Object Modules[.obj]:X:*.obj Type *your program's name.obj*
Run File[E:*.exe]:X:*.exe Type *your program's name.exe*
List File[NUL>MAP]: ↵ *Enter*
Libraries[.lib]: ↵ *Enter*

Now the executable file of your program has been created. To run it:

E:\ASSEM\MASM>X:*.exe Type *X:your program's name.obj* and your program will be executed

(III) Segments of a Program

Usually a program consists of at least three segments: the stack, code and data segments. The code segment contains the assembly language instructions that perform the tasks. In the data segment the data to be manipulated is stored. And the stack segment is used to store data temporarily. In a program a segment must be started and ended. An example:

```
data segment
...           ; data belonging to this segment must be written here
data ends
```

Defining the stack segment is reserving a space of memory for it. An example of segments definition when starting a program is:

```
data segment
var1 db ?           ;define variable named var1
mes1 dw "hello$"   ;define message
data ends
stack segment stack
db 64 dup(?)       ;reserve 64 bytes of memory for the stack
stack ends
code segment
assume ds:data,cs:code
start:             ;start of program
mov ax,data
mov ds,ax
...                ;program continues
code ends
end start          ;at end of program
```

(IV) Using Code View

The programs can also be run using a debugger. The debugger lets you run the program step by step (line by line) and view what is happening with certain important files. In this assembler there is a similar approach called the code view. To access the code view:


```

MOV AH,09H                ;to output a message to screen
MOV DX,OFFSET MES1
INT 21H

MOV AH,02H                ;making a return
MOV DL,0DH
INT 21H
MOV DL,0AH
INT 21H

MOV AH,4CH                ;exit to DOS
INT 21H
CODE ENDS                 ;the code segment ends
END START                 ;the program ends

```

After properly typing and saving the program go to program prompt and access the assembler as mentioned above. Do the following steps (assuming the program is saved in account X):

- 1) Type *masm* to create the object file and type the following:
 - Source filename[.asm]:X:ex1.asm
 - Object filename[E:ex1.obj]:X:ex1.obj
 - Source listing[Nul.lst]:..enter
 - Cross-reference[Nul.crf]:..enter
- 2) If there are errors it will show now, go to Notepad and correct. If corrections are made you have to save it again and create a new object file otherwise continue.
- 3) Type *link* to create executable:
 - Object Modules[.obj]:X:ex1.obj
 - Run file[E:ex1.exe]:X:ex1.exe
 - List file[Nul.map]:..enter
 - Libraries[.lib]:..enter
- 4) If executable is created type the name of the program:
 - E:\ASSEM\MASM>X:ex1.exe
 - What happens?
- 5) Open code view:
 - E:\ASSEM\MASM>CV X:ex1.exe
 - Now press F2 to view segment content. Then type F8 to go line by line. Try to understand how the registers are changing while

the program is being performed. Analyze what is happening, this will help you when you are to program by yourself. When it finishes an error message appears, pay no attention to it and type F4 to see output.

(VI) Useful Tools

A) Code Conversion using XLAT

In many microprocessor-based systems (like ours) the keyboard is not an ASCII type one. One can use XLAT to translate the hex keys of such keyboards to ASCII. Here is a program example that changes the hex digits of 0-F to their ASCII equivalent:

```
DATA SEGMENT
ASC_TABLE  DB '0','1','3','4','5','6','7','8'
           DB '9','A','B','C','C','D','E','F'
HEX_VALUE  DB ?
ASC_VALUE  DB ?
```

B) Using Procedures (Subroutines)

It is common when programming to break down the program into small modules. A procedure is a group of assembly language instructions that are combined so they can be called by another module. The PROC and ENDP directives are used to indicate the beginning and end of a procedure. A procedure can be NEAR or FAR. For NEAR procedure only IP is saved since CS of the called procedure is the same as the calling program. For FAR procedure both IP and CS are saved. The name assigned to PROC needs to be the same as that assigned to PROC. An example of a far procedure named "MULT":

```
MULT PROC FAR
....
MULT ENDP
```

C) Jumps and Calls

It is often necessary to transfer program control to a different location. Two instructions that perform this are jumps and calls. If control is transferred to a memory location within the current code segment it is NEAR, otherwise is FAR.

Jumps

There are conditional and unconditional jumps. For conditional jumps control is transferred to a new location if a certain condition is met indicated by the flag register. For example *JNZ red*, the processor looks at the zero flag to see if it is raised. If not the CPU starts fetching and executing instruction from the address of the label *red*. If ZF=1 it will not jump and execute the next instruction normally. "*JMP red*" is an unconditional jump in which control is transferred unconditionally to location *red*.

Call

It is used to call a procedure that performs a task that needs to be performed frequently. To call procedure MULT: CALL MULT.

D) Using .IF .ELSE

Sometimes it is easier to use statements .IF, .ELSE, .ELSEIF and .ENDIF than using conditional jumps. There are also other statements like DO-WHILE and REPEAT-UNTIL that can be used for the same purpose. These only works for versions 6.X and up. Here is a table of relational operators used with the .IF statement:

Operator	Function
==	Equal or the same as
!=	Not equal
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
&	Bit test
!	Logical inversion
&&	Logical AND
	Logical OR

E) Loops

The LOOP instruction replaces:

```
DEC CX  
JNZ XXXX
```

When the LOOP XXXX is executed, CX is automatically decremented and, if CX≠0, the microprocessor jumps to target address XXXX. If CX=0 the next instruction is executed.

The DO-WHILE loop is also provided for version 6.X which is used with the .WHILE and .ENDW statements. There is also available the REPEAT-UNTIL loop which is repeated until some condition occurs. The .REPEAT statements defines the start and the end is defined by .UNTIL statement which is followed by a condition.

F) Adjust After Addition (AAA)

If addition results in a value of more than 9, AAA will correct it and pass the extra bit to carry and add 1 to AH. An example:

```
SUB AH,AH ;AH=00  
MOV AL,'7' ;AL=37H  
MOV BL,'S' ;BL=35H  
ADD AL,BL ;37H+35H=6CH=AL  
AAA ;changes 6CH to 02 in AL, AH=CF=1  
OR AX,3030h ;AX=3132 which is the ASCII for 12H  
(you can also subtract 3030H)  
(Consult class text for other decimal and BCD adjusts)
```

(VII) Exit to DOS

Every program has to have a part at the end that lets it return control to the operating system (DOS). For this purpose the two last instructions of a program are:

```
MOV AH,4CH  
INT 21H
```

(VIII) Interrupts

The most widely used interrupts are the INT21H and the INT10H. Each one can perform many functions. Before the service of either of these interrupts is requested, certain registers must have specific values in them depending on the function being requested.

INT 10H

Some of its functions are changing the color of characters or background, clearing the screen or changing the location of the cursor. These options are chosen changing the value of register AH. (Please refer text Appendix A for more details and functions)

- 1) Clearing the screen: AH=06, AL=00, BH=07, CX=0000, DH=24, DL=79. The code will be like this:

```
MOV AH,06
MOV AL,00
MOV BH,07
MOV CH,00
MOV CL,00
MOV DH,24
MOV DL,79
INT 10H
```

- 2) Setting cursor to specific location: AH=02, DH=row value, DL=column value.

Example//Code to set the cursor position to row=15=0FH and column=25=19H.

```
MOV AH,02
MOV BH,00 ;page 0
MOV DL,25
MOV DH,15
INT 10H
```

Exercise problem: Write a program that clears the screen and sets the cursor at the center of the screen.

- 3) Get current cursor position: AH=03. When executing the following code registers DH and DL will have the current row and column values and CX provides information about the cursor:

```
MOV AH,03
MOV BH,00
INT 10H
```

INT 21H

This interrupt is provided by DOS and performs extremely useful functions.(Refer to text, Appendix A for more details and functions)

- 1) Outputting a string of data to the monitor:AH=09, DX=the offset address of the ASCII data to be displayed. Example:

```
DATA DB "HELLO THERE!!$"
MOV AH,09
MOV DX,OFFSET DATA
INT 21H
```

- 2) Outputting a single character to the monitor:AH=02,DL=loaded with the character to be displayed. The following displays the letter 'J':

```
MOV AH,02
MOV DL,'J'
INT 21H
```

- 3) Inputting a single character with echo:AH=01. This function waits until a character is input from the keyboard and then echoes it to the monitor.

- 4) Inputting a string of data from the keyboard:AH=0AH, DX=offset address at which the string of data is stored.

```
ORG 0010H
DATA1 DB 6,?,6 DUP (FF)
MOV AH,0AH
MOV DX,OFFSET DATA1
INT 21H
```

(IX) **Models**

Programs are developed either using full segment or using models. Models are used for short programs or for beginners. Full segments are much more powerful and advanced. If models are to be used in a program the .MODEL statement should appear followed by the size of the model chosen. The sizes are:

TINY: uses only one segment, the code segment

SMALL: all data fits into a single 64K data segment and code into a single 64K code segment

MEDIUM: all data fits into a single 64K data segment and code fits into more than one code segment

There is also COMPACT, LARGE, HUGE and FLAT.

(X) **Macros**

Macros allow the programmer to write a task only once and to invoke it whenever it is needed. Every macro definition must have three parts as follows:

```
name MACRO dummy1,dummy2...dummyN
```

```
...
```

```
ENDM
```

The MACRO directive indicates the start and ENDM its end. The dummies are names, parameters or registers that are mentioned in the body of the macro. After it is written it can be called by its name. The following is a macro for displaying a string:

```
STRING MACRO DATA1          ;DATA1 is a dummy
      MOV AH,09
      MOV DX,OFFSET DATA1
      INT 21H
      ENDM
```

If I want to display the message 'Hello there':

```
MES1 DB "HELLO THERE$"
```

```
...
```

```
STRING MES1          ;invoke the macro
```

(XI) Using Disk Files

It is useful to know how to create, read, write, append and close disk files. For this purpose DOS INT 21H is used. For this purpose you must save the following program that contains four macros that are useful in developing programs. Two of these macros display data on the video display, one reads a key and another exits to DOS.

```
DISP MACRO P1
    MOV AH,2
    MOV DL,P1
    INT 21H
KEY MACRO
    MOV AH,1
    INT 21H
    ENDM
STRING MACRO WHERE
    MOV AH,9
    MOV DX,OFFSET WHERE
    INT 21H
    ENDM
EXIT MACRO
    MOV AX,4C00H
    INT 21H
    ENDM
```

Save these macros in a file called MACS,INC on a floppy disc for further use. Now an example that creates a file and saves the file HANDLE(file until it is closed) in a word size memory location:

```
CREATE MACRO HANDLE,FNAME ;create FNAME, saves HANDLE
    MOV AH,3CH ;create function number
    XOR CX,CX
    MOV DX, OFFSET FNAME
    INT 21H
    MOV HANDLE, AX ;save HANDLE
    ENDM
```

Add this macro to the macro file MACS.INC. Now, the next example shows how to write data to a file. It also shows a macro that closes that file. A single segment should be used for this purpose.

```
WRITE MACRO HANDLE,BUFFER,COUNT
    MOV AH,40H
    MOV BX,HANDLE
    MOV CX,COUNT
    MOV DX,OFFSET BUFFER
    INT 21H
    ENDM
CLOSE MACRO HANDLE
    MOV AH,3EH
    MOV BX,HANDLE
    INT 21H
    ENDM
```

Now follows an example that creates a file on floppy disk A and then closes it using the macros in MACS.INC:

```
CODE SEGMENT CODE
    ASSUME CS:CODE
    INCLUDE MACS.INC           ;include macro file
    HAN1 DW ?
    NAME1 'A:\MYFILE.TST',0
    MAIN PROC FAR
        MOV AX,CS
        MOV DS,AX
        CREATE HAN1,NAME1
        CLOSE HAN1
        EXIT
    MAIN ENDP
CODE ENDS
END MAIN
```

(XII) Assembler Data Directives

The 8086 Microprocessor has standardized the directives for data representation. The following are some of the data directives:

ORG (origin)

It is used to indicate the beginning of the offset address.

DB (define byte)

It allows allocation of memory in byte sized chunks which is the smallest allocation permitted.

DUP (duplicate)

It is used to duplicate a given number of characters avoiding a lot of typing.

DW (define word)

It allocates memory two bytes at a time.

EQU (equate)

It is used to define a constant without occupying a memory location.

It associates a constant value with a data label so that when the label appears in the program the constant value substitutes it.

DD (define doubleword)

Allocates memory locations that are 4 bytes in size.

DQ (define quadwords)

Allocates memory 8 bytes in size.

DT (define 10 bytes)

It allocates packed BCD numbers.

Arrays

The DUP directive creates an array as shown here:

```
LIST DB 10 DUP (?)
```

This reserves 10 locations of memory but stores no value on them. If a *10 DUP (2)* had been written instead, 10 bytes of memory would have been reserved with each location of LIST array initialized with a 02H.

Note: If a "?" is placed as an operand for DW, DB or any other, the assembler sets aside a location and does not initialize it to any specific value (reserving memory).

(XIII) Exercises

- 1) Implement a program that reads one line of data (your name) from the keyboard and stores it into an area of memory. When enter key is pressed the line of data has finished. Use INT21H with AH=0AH and the "enter" key is stored in memory as 0DH.
- 2) Modify exercise #1 so that it displays the following message before typing name: "Please type your name and then press the enter key". After this is done the user's name should be displayed in the screen.
- 3) Implement a program that adds two 1-digit numbers entered by keyboard and displays the result in the screen.
- 4) Modify exercise #3 for it to add two 2-digit numbers. Then two 4-digit numbers.
- 5) Implement a program, using arrays, that when a five-digit number is entered the output is the opposite. For example if number: 12345 is entered the output displayed should be 54321.
- 6) Create a program that displays the binary number of the powers of 2 entered by user. For example if user enters 3: $2^3=8=1000$ is displayed.
- 7) Create a program to create a file called FROG.JMP(using the macros available in your floppy disc) and store five bytes of data:48H, 65H, 6CH and 6FH in this file and then close it.
- 8) Create two macros, one that reads bytes of data from a file and another one that opens an existing file for a read or write operation.

(XIV) References

Brey, Barry B., *The Intel Microprocessor 8086/88*, Fourth Edition, New Jersey, Prentice Hall, Inc., 1997

Mazidi, Muhammad A., Janice Gillispie Mazidi, *The 8086 IBM PC and Compatible Computers*, Vol I & II. New Jersey, Prentice Hall, Inc., 1995

(XV) Appendix

ASCII Codes Table

TABLE 1-7 ASCII code

		Second															
First		X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	XA	XB	XC	XD	XE	XF
0X		NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1X		DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2X		SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3X		0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4X		@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5X		P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6X		`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7X		p	q	r	s	t	u	v	w	x	y	z	{		}	~	⋮

TABLE 1-8 Extended ASCII code as printed by the IBM ProPrinter

First		Second															
		X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	XA	XB	XC	XD	XE	XF
0X		☺	☹	♥	♦	♣	♠	●	■	○	◐	♂	♀	♪	♫	⚙	
1X		▶	◀	!	!!	¶	§	■	‡	‡	‡	‡	‡	‡	‡	‡	‡
8X		Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å
9X		É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	€	£	¥	℞	f
AX		á	í	ó	ú	ñ	Ñ	ª	º	¿	¡	½	¼	¡	«	»	
BX		⋮	⋮	⋮													
CX		⋮	⋮	⋮													
DX		⋮	⋮	⋮													
EX		α	β	Γ	π	Σ	σ	μ	γ	Φ	Θ	Ω	δ	∞	φ	ε	∅
FX		≡	±	≥	≤	∫	∫	÷	≈	°	·	·	√	n	2	■	