COSC 204 LABWORK#1

Objectives:

 To introduce the students to:

1. The general structure of computer register
2. Using a debugger to view the functionality of register during an instruction execution
3. Creating simple .COM program using debugger

To begin working with Debug, type the following prompt in your computer: C:/>Debug [Enter]

On the next line a dash will appear, this is the indicator of Debug, at this moment the instructions of Debug can be introduced using the following command:

to visualize the values of the internal registers of the CPU using the Debug program type -r[Enter]

AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0D62 ES=0D62 SS=0D62 CS=0D62 IP=0100 NV EI PL NZ NA PO NC
0D62:0100 2E CS:
0D62:0101 803ED3DF00 CMP BYTE PTR [DFD3],00 CS:DFD3=03

To view a specific register the "r" command using as a parameter the name of the register whose value wants to be seen. For example:

-rbx
BX 0000
:

This instruction will only display the content of the BX register and the Debug indicator changes from "-" to ":" to change the value of the register type the new value and [Enter], or the old value can be left by pressing [Enter] without typing any other value.

**To create a program on the debug**

Type debug[Enter] on the operative system prompt.

To assemble a program on the Debug, the "a" (assemble) command is used; when this command is used, the address where you want the assembling to begin can be given as a parameter, if the parameter is omitted the assembling will be initiated at the locality specified by CS:IP, usually 0100h, which is the locality where programs with .COM extension must be initiated

C:\>debug
-a 100
0D62:0100 mov ax,0002
0D62:0103 mov bx,0004
0D62:0106 add ax,bx
0D62:0108 nop
0D62:0109

Type the command "t" (trace), to execute each instruction of this program,
example:

-t

AX=0002 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0D62 ES=0D62 SS=0D62 CS=0D62 IP=0103 NV EI PL NZ NA PO NC
0D62:0103 BB0400 MOV BX,0004

You see that the value 2 move to AX register. Type the command "t" (trace),
again, and you see the second instruction is executed.

-t

AX=0002 BX=0004 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0D62 ES=0D62 SS=0D62 CS=0D62 IP=0106 NV EI PL NZ NA PO NC
0D62:0106 01D8 ADD AX,BX

Type the command "t" (trace) to see the instruction add is executed, you will see the follow lines:

-t

AX=0006 BX=0004 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0D62 ES=0D62 SS=0D62 CS=0D62 IP=0108 NV EI PL NZ NA PE NC
0D62:0108 90 NOP

The possibility that the registers contain different values exists, but AX and BX must be the same, since they are the ones we just modified.

To exit Debug use the "q" (quit) command.

**Storing and loading the programs**

The steps to save a program that it is already stored on memory are:

Obtain the length of the program subtracting the final address from the initial address, naturally in hexadecimal system.
Give the program a name and extension. Put the length of the program on the CX register. Order Debug to write the program on the disk.

By using as an example the following program, we will have a clearer idea of how to take these steps:

When the program is finally assembled it would look like this:

0C1B:0100 mov ax,0002
0C1B:0103 mov bx,0004
0C1B:0106 add ax,bx
0C1B:0108 int 20
0C1B:010A

To obtain the length of a program the "h" command is used, since it will show us the addition and subtraction of two numbers in hexadecimal. To obtain the length of ours, we give it as parameters the value of our program's final address (10A), and the program's initial address (100). The first result the command shows us is the addition of the parameters and the second is the subtraction.

-h 10a 100
020a 000a

The "n" command allows us to name the program.

-n test.com

The "rcx" command allows us to change the content of the CX register to the value we obtained from the size of the file with "h", in this case 000a, since the result of the subtraction of the final address from the initial address.

-rcx
CX 0000
:000a

Lastly, the "w" command writes our program on the disk, indicating how many bytes it wrote.

-w
Writing 000A bytes

To load an already saved file two steps are necessary:

Give the name of the file to be loaded.
Load it using the "l" (load) command.

To obtain the correct result of the following steps, it is necessary that the above program be already created.

Inside Debug we write the following:

-n test.com
-l
-u 100 109
0C3D:0100 B80200 MOV AX,0002
0C3D:0103 BB0400 MOV BX,0004
0C3D:0106 01D8 ADD AX,BX
0C3D:0108 CD20 INT 20

The last "u" command is used to verify that the program was loaded on memory. What it does is that it disassembles the code and shows it disassembled. The parameters indicate to Debug from where and to where to disassemble.

Debug always loads the programs on memory on the address 100H, otherwise indicated.

COSC 204 LABWORK#2

Objectives:

 To introduce the students to:

1. The general structures of both exe-format and com-format programs.
2. Full and simplified segment directives.
3. Assembling and linking both exe-format and com-format programs using MASM version 5.00, TASM version 4.1, LINK version 5.6, TLINK version 7.1.30.1
4. THE GENERAL STRUCTURE OF AN EXE-FORMAT PROGRAM

 The general form of an exe-format program having a data segment is:

 STACK SEGMENT STACK

 DB 400H DUP(?)

 STACK ENDS

 DATA SEGMENT

 ; data definitions using DB, DW, DD, etc. come here

 . . .

 DATA ENDS

 CODE SEGMENT

 ASSUME CS : CODE , DS : DATA , SS : STACK

 ENTRY: MOV AX , DATA ; Initialize DS

 MOV DS , AX ;

 . . .

 ; Return to DOS

 MOV AX , 4C00H

 INT 21H

 CODE ENDS

 END ENTRY

Example: The following exe-format program displays the string COSC 204 on the screen:

STACK\_SEG SEGMENT STACK

 DB 200 DUP(?)

 STACK\_SEG ENDS

 DATA\_SEG SEGMENT

 MESSAGE DB “COSC 204$ “

 DATA\_SEG ENDS

 CODE\_SEG SEGMENT

 ASSUME CS : CODE\_SEG , DS : DATA\_SEG , SS : STACK\_SEG

 START: MOV AX , DATA\_SEG ; Initialize DS

 MOV DS , AX ;

 ; Display the string

 MOV AH , 09H

 MOV DX , OFFSET MESSAGE

 INT 21H

 ; Return to DOS

 MOV AX , 4C00H

 INT 21H

 CODE\_SEG ENDS

 END START

1. THE GENERAL STRUCTURE OF A COM-FORMAT PROGRAM

 CODE SEGMENT

 ASSUME CS : CODE , DS : CODE , SS : CODE

 ORG 100H

 ENTRY: JMP L1

 ; data definitions come here

 . . .

 L1:

 . . .

 ; Return to DOS

 MOV AX , 4C00H

 INT 21H

 CODE ENDS

 END ENTRY

alternatively, the following general structure may be used:

CODE SEGMENT

 ASSUME CS : CODE , DS : CODE , SS : CODE

 ORG 100H

 ENTRY:

 . . .

 ; Return to DOS

 MOV AX , 4C00H

 INT 21H

; data definitions, if any, come here

 . . .

 CODE ENDS

 END ENTRY

Example: The following com-format program displays the string COSC 204 on the screen:

 CODE SEGMENT

 ASSUME CS : CODE , DS : CODE , SS : CODE

 ORG 100H

 ENTRY: JMP START

 MESSAGE DB “COSC 204$ “

 START: ; Display the string

 MOV AH , 09H

 MOV DX , OFFSET MESSAGE

 INT 21H

 ; Return to DOS

 MOV AX , 4C00H

 INT 21H

 CODE ENDS

 END ENTRY

1. SIMPLIFIED SEGMENT DIRECTIVES

MASM version 5.0 and above, and TASM provide a simplified set of directives for declaring segments called simplified segment directives. To use these directives, you must initialize a memory model, using the .MODEL directive, before declaring any segment. The format of the .MODEL directive is:

.MODEL memory-model

The memory-model may be TINY, SMALL, MEDIUM, COMPACT, LARGE, or HUGE :

|  |  |
| --- | --- |
| memory-model | description |
| TINY | One segment. Thus code and data together may not be greater than 64K |
| SMALL | One code-segment. One data-segment. Thus neither code nor data may be greater than 64K |
| MEDIUM | More than one code-segment. One data-segment. Thus code may be greater than 64K |
| COMPACT | One code-segment. More than one data-segment. Thus data may be greater than 64K |
| LARGE | More than one code-segment. More than one data-segment. No array larger than 64K. Thus both code and data may be greater than 64K |
| HUGE | More than one code-segment. More than one data-segment. Arrays may be larger than 64K. Thus both code and data may be greater than 64K |

 All of the program models except TINY result in the creation of exe-format programs. The TINY model creates com-format programs. The .MODEL directive automatically generates the required ASSUME statement; thus programs with simplified segment directives do not have this directive.

The simplified segment directives are: .CODE , .DATA , .STACK . The .CODE directive may be followed by the name of the code segment. The .STACK directive may be followed by the size of the stack segment, by default the size is 1K i.e., 1,024 bytes. The definition of a segment extends from a simplified segment directive up to another simplified segment directive or up to the END directive if the defined segment is the last one.

 The DOSSEG directive, which is optional, tells the assembler to adopt the DOS segment-ordering conversion. For a SMALL model program, the order is code, data, stack. This directive should appear before any segment definitions.

 Note: The instructions which initialize the DS register for an exe-format program with simplified segment directives are:

 MOV AX , @DATA

 MOV DS , AX

 where AX may be replaced by any other 16-bit general purpose register.

 Example:

 DOSSEG

 .MODEL SMALL

.STACK 200

.DATA

 MESSAGE DB “COSC 204$ “

.CODE

 START: MOV AX , @DATA ; Initialize DS

 MOV DS , AX ;

 ; Display the string

 MOV AH , 09H

 MOV DX , OFFSET MESSAGE

 INT 21H

 ; Return to DOS

 MOV AX , 4C00H

 INT 21H

 END START

For a com-format program simplified segment directives can only be used if the data is not at the end of the code segment after the Return to DOS instructions. If the data is at the end then full segment declarations using SEGMENT and ENDS directives must be used. Example:

 .MODEL TINY

 .CODE

 ORG 100H

 ENTRY: JMP START

 MESSAGE DB “COSC 204$ “

 START: ; Display the string

 MOV AH , 09H

 MOV DX , OFFSET MESSAGE

 INT 21H

 ; Return to DOS

 MOV AX , 4C00H

 INT 21H

 END ENTRY

# ASSEMBLING, LINKING, AND EXECUTING 8086 ASSEMBLY LANGUAGE PROGRAMS

The assemblers and linkers have a number of options, which are of the form: /option. These options are not case-sensitive for MASM Version 5.00, TASM Version 4.1, LINK Version 5.3, and TLINK Version 7.1. However, they are case sensitive for ML.EXE (MASM Version 6.00 and above). In addition, the options for MASM (Version 5.00), TASM, LINK, TLINK may or may not be separated by blanks. The options for ML.EXE must be separated by blanks.

In what follows we only mention the options that we will use in this course. The elements within square brackets are optional:

 Masm Version 5.00

 Usage: MASM

 or

MASM [options] source[.asm] , [object[.obj]] , [list[.lst]] , [cref[.crf]] [;]

/c Generate cross-reference in listing

/l Generate normal listing

/la Generate expanded listing

/n Suppress symbol tables in listing

/z Display source line for each error message

/Zi Generate symbolic information for CodeView

/Zd Generate line-number information

--------------------------------------------------------------------------------------------------------------------------

 Turbo Assembler Version 4.1

Usage: TASM [options] source[.asm] , [object[.obj]] , [list[.lst]] , [cref[.crf]] [;]

/c Generate cross-reference in listing

/l Generate normal listing

/la Generate expanded listing

/m# Allow # multiple passes to resolve forward references

/n Suppress symbol tables in listing

/z Display source line with error message

/zi, /zd, /zn Debug info: zi=full, zd=line numbers only, zn=none

Microsoft (R) Segmented Executable Linker Version 5.31.009

Usage: LINK

 or

LINK [options] objfiles, runfile, mapfile, libfiles, deffile [;]

 /CODEVIEW Full symbolic debug information for CodeView debugger

 /CO Full symbolic debug information for CodeView debugger

/LINENUMBERS

/TINY Create COM file

 /T Create COM file

----------------------------------------------------------------------------------------------------------------

Turbo Link Version 7.1.26.1.

Syntax: TLINK [options] objfiles, runfile, mapfile, libfiles, deffile, resfiles [;]

/x No map

/v Full symbolic debug information

 /t Create COM file (same as /Tdc)

 /k Suppress "No stack" warning msg

---------------------------------------------------------------------------------------------------------------

 ML [options] filelist [/link linkOptions]

/AT Enable tiny model (.COM file)

/c Assemble without linking

/Sn Suppress symbol-table listing

/Fe<file> Name executable

/Fl[file] Generate listing

/Fo<file> Name object file

 /Zd Add line number debug info

 /Zi Add symbolic debug info

 /Zm Enable MASM 5.10 compatibility

/link <linker options and libraries>

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

LAB EXERCISE:

Assemble and link the example above using TASM in the directory C:\TASM. Try to assemble with several assembler and linker option.

Run the obtained executable file using the command prompt.

COSC 204 LABWORK#3

Objectives:

 An introduction to:

1. DOS functions 01H , 02H, 08H, 09H, 40H, and 0AH
2. INT, MOV, INC, DEC, ADD, and SUB instructions.
3. Write an 8086 COM-format assembly language program which prompts for a lowercase letter x1 in the range a to x:

PLEASE, ENTER A LOWERCASE LETTER IN THE RANGE a TO x:

 it then displays an output of the form:

X3 X2 X1

where X2 and X3 are the two uppercase characters which come after uppercase X1 in the alphabet.

For example, if the input is c then the output is:

E D C

Note: each two characters in the output are separated by one two blanks.

Hint: LowercaseLetter - 20H = UppercaseLetter

COSC 204 LABWORK#4

Objectives:

1. Learning how to link separately assembled modules from the command line.
2. Learning how to use object module libraries.

## LINKING SEPARATELY ASSEMBLED MODULES

Suppose PROG1.ASM, PROG2.ASM, PROG3.ASM are three modules to be executed as a single program. The steps are:

1. Assemble each of the files separately, in whatever order. Ignore any warning of a missing stack segment. This will create the object files:

PROG1.OBJ , PROG2.OBJ , and PROG3.OBJ

1. Link the three object files:

>LINK PROG1 + PROG2 + PROG3 , EXEFILE\_NAME (for exe-format modules)

 or

>LINK /T PROG1 + PROG2 + PROG3 , COMFILE\_NAME (for com-format modules)

PROG1, PROG2, and PROG3 in the above command lines may appear in any order. If the name for the executable file is missing, the first object file name in the list is taken as the name of the executable file.

The same format is used if the linker is TLINK.

Note: If any of the object files is in a different directory from the directory in which the linker is located, then the full path of that file must be given.

1. Execute the generated executable file by the command:

>EXECUTABLE\_FILE\_NAME

 A full path may be required.

Note: In an IDE such as The Programmers Work Bench, the concept of a project is used in assembling, linking, and executing multi-module programs.

-----------------------------------------------------------------------------------------------------------------------

Using the code DSPLYMDL.ASM, READMDL.ASM, CALLER1.ASM, AND CALLER2.ASM FROM below,

1. CREATE THE A SEPARATE ASSEMBLY FILE( .ASM) FILE FOR EACH OF THEM
2. ASSEMBLE CALLER1.ASM AND THEN DSPLYMDL.ASM, LINK CALLER1.OBJ AND DSPLYMDL.OBJ, AND FINALLY EXECUTE THE GENERATED EXECUTABLE FILE.
3. ASSEMBLE CALLER2.ASM AND THEN READMDL.ASM, LINK CALLER2.OBJ, DSPLYMDL.OBJ, AND READMDL.OBJ, AND FINALLY EXECUTE THE GENERATED EXECUTABLE FILE.

-----------------------------------------------------------------------------------------------------

DsplyMdl.asm:

PUBLIC DISPLAY\_CHAR , DISPLAY\_STRING\_WITH\_FNCTN\_09H , DISPLAY\_STRING\_WITH\_FNCTN\_40H

CODE SEGMENT

 ASSUME CS : CODE

DISPLAY\_CHAR PROC FAR

 ; Enter with DL the character to be displayed

 PUSH AX ; save AX

 MOV AH , 02H ; display character passed in DL

 INT 21H ;

 POP AX ; restore AX

 RET

DISPLAY\_CHAR ENDP

DISPLAY\_STRING\_WITH\_FNCTN\_09H PROC FAR

 ; Enter with DX the offset of the string to be displayed and DS the segment number of the string.

 PUSH AX ; save AX

 MOV AH , 09H ; display a $-terminated string

 INT 21H ;

 POP AX ; restore AX

 RET

DISPLAY\_STRING\_WITH\_FNCTN\_09H ENDP

DISPLAY\_STRING\_WITH\_FNCTN\_40H PROC FAR

; Enter with DX the offset of the string to be displayed, DS the segment number of the string, and CX the length of ; the string

 PUSH AX ; save AX

 PUSH BX ; save BX

 MOV AH , 40H ; display string

 MOV BX , 01H ; handle for the screen

 INT 21H ;

 POP BX ; restore BX

 POP AX ; restore AX

 RET

DISPLAY\_STRING\_WITH\_FNCTN\_40H ENDP

CODE ENDS

END

----------------------------------------------------------------------------------------------------

ReadMdl.asm:

PUBLIC READ\_CHAR\_WITH\_ECHO , READ\_CHAR\_WITHOUT\_ECHO , READ\_STRING

DATA SEGMENT

 BUFFER DB 81 , 82 DUP(?)

DATA ENDS

CODE SEGMENT

 ASSUME CS : CODE , DS : DATA

 READ\_CHAR\_WITH\_ECHO PROC FAR

 PUSH CX ; save CX

 MOV CH , AH ; save AH

 MOV AH , 01H ; read character with echo

 INT 21H ;

 MOV AH , CH ; restore AH

 POP CX ; restore CX

 RET

 READ\_CHAR\_WITH\_ECHO ENDP

 READ\_CHAR\_WITHOUT\_ECHO PROC FAR

 PUSH CX ; save CX

 MOV CH , AH ; save AH

 MOV AH , 08H ; read character without echo

 INT 21H ;

 MOV AH , CH ; restore AH

 POP CX ; restore CX

 RET

 READ\_CHAR\_WITHOUT\_ECHO ENDP

 READ\_STRING PROC FAR

 ; returns BP (this module's DS value), SI (length of input string) , DI (offset of input string)

 PUSH AX ; save AX

 PUSH BX ; save BX

 PUSH DX ; save DX

 PUSH DS ; save DS of caller

 MOV AX , DATA ; initialize DS

 MOV DS , AX ;

 MOV AH , 0AH ; read string into BUFFER

 LEA DX , BUFFER ;

 INT 21H ;

 MOV BP , DS ; return DS value of this module's DATA segment

 LEA SI , BUFFER[2] ; return the offset of the input string

 MOV BH , 0 ; return length of the input string

 MOV BL , BUFFER[1] ;

 MOV DI , BX ;

 POP DS ; restore the callers DS

 POP DX ; restore DX

 POP BX ; restore BX

 POP AX ; restore AX

 RET

 READ\_STRING ENDP

CODE ENDS

END

------------------------------------------------------------------------------------------------------------------------

Caller1.asm:

EXTRN DISPLAY\_STRING\_WITH\_FNCTN\_40H : FAR , DISPLAY\_STRING\_WITH\_FNCTN\_09H : FAR

STACKSG SEGMENT STACK

 DB 0400H DUP(?)

STACKSG ENDS

DATA SEGMENT

 MSG1 DB 'I LIKE ASSEMBLY PROGRAMMING VERY MUCH', 0DH , 0AH

 LEN\_MSG1 EQU $ - MSG1

 MSG2 DB 'THAT IS WHY I HEREBY RESOLVE TO WORK HARD', '$'

DATA ENDS

CODE SEGMENT

 ASSUME CS : CODE , DS : DATA , SS : STACKSG

ENTRY: MOV AX , DATA

 MOV DS , AX

 LEA DX , MSG1

 MOV CX , LEN\_MSG1

 CALL DISPLAY\_STRING\_WITH\_FNCTN\_40H

 LEA DX , MSG2

 CALL DISPLAY\_STRING\_WITH\_FNCTN\_09H

 MOV AX , 4C00H

 INT 21H

CODE ENDS

END ENTRY

-------------------------------------------------------------------------------------------------------

Caller2.asm:

EXTRN DISPLAY\_STRING\_WITH\_FNCTN\_40H : FAR , DISPLAY\_STRING\_WITH\_FNCTN\_09H : FAR

EXTRN READ\_STRING : FAR

STACKSG SEGMENT STACK

 DB 0400H DUP(?)

STACKSG ENDS

DATA SEGMENT

 MSG1 DB 'PLEASE ENTER YOUR NAME (MAX LENGTH 80 CHARACTERS):', 0DH , 0AH

 LEN\_MSG1 EQU $ - MSG1

 MSG2 DB 0DH , 0AH, 0DH, 0AH, 'WELCOME TO THE MINI-WORLD OF ASSEMBLY PROGRAMMING MR. ', '$'

DATA ENDS

CODE SEGMENT

 ASSUME CS : CODE , DS : DATA , SS : STACKSG

ENTRY: MOV AX , DATA

 MOV DS , AX

 LEA DX , MSG1

 MOV CX , LEN\_MSG1

 CALL DISPLAY\_STRING\_WITH\_FNCTN\_40H

 CALL READ\_STRING ; Returns the length of the string in DI,

 ; the offset address of the string in SI,

 ; and the segment address of the string in BP

 LEA DX , MSG2

 CALL DISPLAY\_STRING\_WITH\_FNCTN\_09H

 MOV CX , DI ; Assign the length of the string to CX

 MOV DX , SI ; Assign the offset address of the string to DX

 MOV DS , BP ; Assign the segment address of the string to DS

 CALL DISPLAY\_STRING\_WITH\_FNCTN\_40H

 MOV AX , 4C00H

 INT 21H

CODE ENDS

END ENTRY

# CREATING AND USING LIBRARIES OF OBJECT MODULES

The utility LIB.EXE (Microsoft Library Manager) or TLIB.EXE (Turbo Library Manager) can be used to create libraries of object modules. Such libraries form an important software tool for the assembly language programmer.

The usage of these two utilities is given below:

--------------------------------------------------------------------------------------------------------------

LIB.EXE (Microsoft Library Manager) Version 3.20.010

Usage: LIB library\_name [options] [commands] [,listfile [,newlibrary]] [;]

Where:

1. the items in square brackets are optional.
2. library\_name identifies the library to be created or updated. It may include the path.
3. newlibrary identifies the output library file. This entry is ignored if the library library\_name is being created; otherwise if this field is empty, and the library library\_name does exist, LIB.EXE creates a backup copy of the library called library\_name.BAK
4. listfile identifies the output file for the library listing. This is a cross-reference file with information about all FAR procedures and modules in the library.
5. Options:

 /? : display LIB options

 /HELP : display help on LIB

 /IGNORECASE : ignore case on names

 /NOEXTDICTIONARY : do not build extended dictionary

 /NOIGNORECASE : do not ignore case on names

 /NOLOGO : do not display signon banner

 /PAGESIZE:n : set library page size to n. The default is 16 bytes.

1. Commands:

A command is of the form:

command name

 where name is the name of a single object file or a library name. The name may be separated from the command by zero or more blanks. The command is performed on a single object file if name is that of a single object file; otherwise if name is a library name then the command is performed on all object files contained in the library. The commands and their meanings are given in the following table:

|  |  |
| --- | --- |
| Command symbol | Meaning |
| + | Add to an object file or object files to the library. |
| - | Delete an object file or object files from the library. |
| \* | Copy (extract) an object file or object files from the library to an object file or object files with the same names. |
| -+ | Replace an object file or object files from the library by an object file with the same name or names as in the command. |
| -\* | Move (delete and extract) an object file or object files from the library to an object file or object files with same names. |

Note:

1. If name is more than 8 characters in length it is truncated to 8 characters by LIB.EXE. (TLIB.EXE accepts names that are longer then 8 characters).
2. If name is the name of a library and no extension is supplied then .LIB is the default extension.
3. When one double-clicks on LIB.EXE it displays prompts in the following sequence:
	1. Library name:
	2. If the library supplied in response to the above prompt does not exist, the following prompt is displayed:

 Library does not exist. Create? (y/n)

* 1. If the library does exist or if the response to the above prompt was y, the following prompt is displayed:

 Operations:

* 1. List file:

Note: Double-clicking on TLIB.EXE only result in the display of the usage of TLIB.EXE.

1. Both LIB.EXE and TLIB.EXE will reject the addition of an object file to a library if that object file contains a name in its PUBLIC directive that already appears in the PUBLIC directive of one of the modules in the library. LIB.EXE will issue a warning of the form:

Warning U415: ‘xxxxxxx’: symbol defined in module yyyyyy, redefinition ignored.

TLIB.EXE will issue a warning of the form:

 Warning: ‘xxxxxxx’ already in LIB, not changed !

----------------------------------------------------------------------------------------------------------------------

TLIB (Turbo Library Manager) Version 4.00

Usage: TLIB library\_name [/C] [/E] [/P] [/0] commands [, listfile]

 library\_name library file (may include the path)

 commands sequence of operations to be performed (optional)

 listfile file name for listing file (optional)

The commands are the same as those of LIB.EXE except that TLIB.EXE also supports the commands:

 +- Replace an object file or object files from the library.

 \*- Move (delete and extract) an object file or object files from the library.

Options:

 /C case-sensitive library

 /E create extended dictionary

 /PSIZE set the library page size to SIZE. The default is 16 bytes.

 /0 purge comment records

Note: If the library library\_name is modified, TLIB.EXE creates a backup copy of the library called library\_name.BAK

-------------------------------------------------------------------------------------------------------------------------EXERCISE 5 (PROGRAMMING ASSIGNMENT)

(a) Write an exe-format, 8086 assembly language program, which has full segment directives, to prompt for and read a character (DON’T USE READ\_STRING). The program must then display an output of the form:

THE INPUT CHARACTER IS:

followed by the character that was read. Use DSPLY\_CHAR to display the character.

Your program must use the FAR procedures defined in ReadMdl.asm and DsplyMdl.asm for:

1. Displaying the prompt.
2. Reading the character.
3. Displaying the message: THE INPUT CHARACTER IS:
4. Displaying the character.

(b) Assemble the program, then link the object file to ReadMdl.obj and DsplyMdl.obj, and finally execute the generated executable file.

------------------------------------------------------------------------------------------------------------------------

EXERCISE 6 (PROGRAMMING ASSIGNMENT)

1. Write an 8086 EXE-format assembly language program that prompts for and reads a user’s name USERNAME (of maximum length 50 characters). It then prompts for and reads the name of the user’s town USERTOWN (of maximum length 30 characters). It then displays a message of the form:

Mr. USERNAME lives in

 USERTOWN town.

Your program must contain appropriate NEAR procedures to read a string and to display a string.

1. Assemble your program, adding debug information to the object file, by a command line of the form:

 TASM /zi prog , prog , prog

1. Link the program, adding debug information to the executable file, by a command line of the form:

 TLINK /v prog , prog

1. After executing your program, invoke Turbo debugger by a command line of the form:

 TD prog

1. Notice the difference in tracing the program by F7 (Trace Into) and F8 (Step over).
2. Observe the values pushed on top of the stack when your execute procedure calls.
3. Observe the values popped from the stack when you execute RET instructions.

 Note:

* 1. To reset a program after normal termination, use Ctrl-F2 or select the *Run* option and then select *Program reset*.
	2. To view the top of the stack, select the *View* option, select *Dump*, right click on the dump window, select *Goto . . .* , type *ss : sp* in the dialogue box which appears, and then click on the *OK* button.

COSC 204 LABWORK#5

Objectives:

Introducing students to:

1. MACROS

(a) macros calling other macros, (b) Local and global identifiers, (c) .LALL and .SALL directives, (d) macro include-files.

1. PROCUDURES WHICH PASS AND RETURN VALUES THROUGH THE STACK.
2. USING BATCH FILES IN ASSEMBLING AND LINKING ASSEMBLY LANGUAGE PROGRAMS.

The contents of macro1.asm and macro2.asm are given below:

macro1.asm:

----------------------------------------------------------------------------------------------------------------------------

DISPLAY\_CHAR MACRO CHAR

 PUSH AX

 PUSH DX

 MOV AH , 02H

 MOV DL , CHAR

 INT 21H

 POP DX

 POP AX

ENDM

CRLF MACRO

 DISPLAY\_CHAR 0DH

 DISPLAY\_CHAR 0AH

ENDM

STACK SEGMENT STACK

 DB 400 DUP(?)

STACK ENDS

CODE SEGMENT

 ASSUME CS : CODE , SS : STACK

ENTRY:

 DISPLAY\_CHAR 'A'

 CRLF

 DISPLAY\_CHAR 'B'

 MOV AX , 4C00H

 INT 21H

CODE ENDS

END ENTRY

----------------------------------------------------------------------------------------------------------------------

macro2.asm:

---------------------------------------------------------------------------------------------------------------------

TOWNS MACRO

 LOCAL TOWN, DONE

 PUSH AX

 PUSH DX

 PUSH DS

 MOV AX, CS

 MOV DS, AX

 MOV DX , OFFSET TOWN ; local TOWN

 CALL DISPLAY\_STRING ; DISPLAY\_STRING is a global identifier

 POP DS

 POP AX

 MOV DX , OFFSET COUNTRY ; COUNTRY is a global identifier

 CALL DISPLAY\_STRING

 POP DX

 JMP DONE

 TOWN DB 'KADUNA', 0DH , 0AH , '$'

 DONE:

ENDM

.MODEL SMALL

.STACK 0400H

.DATA

 TOWN DB 'ZARIA', 0DH , 0AH , '$'

 COUNTRY DB 'NIGERIA', 0DH , 0AH , '$'

.CODE

 MAIN PROC

 MOV AX, @DATA

 MOV DS , AX

 MOV DX , OFFSET TOWN ;global TOWN

 CALL DISPLAY\_STRING

 TOWNS ; a macro call

 MOV AX , 4C00H

 INT 21H

 MAIN ENDP

 DISPLAY\_STRING PROC

 PUSH AX

 MOV AH , 09H

 INT 21H

 POP AX

 RET

 DISPLAY\_STRING ENDP

END MAIN

--------------------------------------------------------------------------------------------------------------------

.SALL (Suppress ALL) and .LALL (List ALL) directives

The .SALL directive suppresses macro expansions in the listing file. The directive can be placed anywhere in the .asm file before the calls to the macro or macros whose expansions are to be suppressed. To re-enable macro expansions, use the .LALL directive.

---------------------------------------------------------------------------------------------------------------------------

TASK

Generate macro1.lst and macro2.lst by:

1. Not using .SALL in the .asm files, and
2. Using .SALL in the .asm files.

in both cases observe the listing files.

------------------------------------------------------------------------------------------------------------------------------

PROGRAMMING ASSIGNMENTS:

1. Write an 8086 Assembly language program that prompts for and reads a student’s name, it then prompts for and reads a student’s major. It finally displays an output of the form:

Mr. STUDENTNAME majors in STUDENTMAJOR

The reading must be done using a macro READ\_STRING that has one parameter: the name of the buffer to read into. All string displays, including the buffer displays, must be done in a macro STRING\_DISPLAY that has the header:

 STRING\_DISPLAY MACRO STRINGNAME , LENSTRING

Where the actual parameter corresponding to STRINGNAME is the name of the string to be displayed, and LENSTRING corresponds to a *16-bit* length of the string to be displayed. Your macro must use DOS function 40H.

1. Write an 8086 Assembly language module that contains the following definitions in its data segment:

MESSAGE DB “I LIKE ASSEMBLY LANGUAGE”

MESSAGELEN EQU $ - MESSAGE

The module calls a separately assembled 8086 Assembly language module, passing to it the segment address of the data segment in the DS register, the offset address of MESSAGE in the stack, and the length of MESSAGE in the stack. The called module then uses DOS function 40H to display the string MESSAGE.

COSC 204 LABWORK#6

Objectives: exercise on

1. JMP, Conditional jump, LOOP, conditional loop instructions.

2. One-dimensional array manipulation.

1. Write a com-format, 8086 Assembly language program that prompts for and reads a character. It then displays one of the massages: THE CHARACTER IS A HEXADECIMAL DIGIT or THE CHARACTER IS NOT A HEXADECIMAL DIGIT accordingly.

Note: The reading of the character, the display of the prompt, and the display of the appropriate message must be done using two appropriate macros defined in a macro include-file.

1. Write an exe-format, 8086 Assembly language program which:
	1. prompts for and reads a maximum of 10 ASCII digits into a one-dimensional array of size 10.
	2. prompts for and reads a search character.
	3. determines whether the search character is a member of the one-dimensional array or not.
	4. displays one of the messages: THE CHARACTER IS FOUND or THE CHARACTER IS NOT FOUND accordingly.

Your program must:

1. use appropriate procedures.

(ii) check for input validity. In case of invalid input, the program should beep (the bell character is 07H), return the cursor one column back (the backspace character is 08H), and give the user a chance to enter another value.

(iii) be able to process less than 10 elements for task (a), that is, if the user presses the enter key (0DH) before 10 values are read, the reading loop must terminate.

(iii) use an appropriate conditional loop instruction in performing task (c)

COSC 204 LABWORK#7

Objectives:

1. Command-line text processing.
2. Two-dimensional array manipulation (BasedIndexed addressing mode).
3. Structures, Nested structures, arrays of structures.

PROGRAMMING ASSIGNMENTS

1. Write an exe-format 8086 Assembly language program that accepts a command line of the form:

x+y

or

 x-y

Where x and y are digits such that if the operator is + then x + y ≤ 9, and if the operator is - , then x ≥y .

Your program should display the result of the operation on the two digits. Don’t test for input validity; but make sure your command-line satisfies the given conditions.

1. A two-dimensional array is defined as:

ARRAY DB 1 , 0 , 2 , 1

 DB 2 , 1 , 0 , 0

 DB 0 , 1 , 1 , 0

Write an 8086 Assembly language program to find the sum of the array. Your program must use BasedIndexed addressing mode. The program must be able to find the sum of any two-dimensional byte array of size 3 \* 4 and whose sum is a value in the range 0 to 9.

COSC 204 LABWORK#8

Objectives:

1. RECURSIVE PROCEDURES.

Terminating gracefully from a recursive procedure in which an error condition is detected.

Preventing the stack overflow condition in recursive procedures.

1. STAND ALONE PROCEDURES.

Using macros to write stand-alone procedures.

Using Nested Procedures to write stand -alone procedures.

1. HEXADECIMAL I/O ROUTINES.
2. DETECTING SIGNED AND UNSIGNED OVERFLOW.

DETECTING UNSIGNED OVERFLOW

* The Carry Flag (CF) is set if there is unsigned overflow. Unsigned overflow occurs if the result is too large to fit into the destination operand. In a program unsigned overflow can be detected by a construct of the form:

Flag modifying instruction

JC UNSIGNED\_OVERFLOW

.

. ; action if there is no unsigned overflow

.

JMP DONE

UNSIGNED\_OVERFLOW:

 .

 . ; action if there is unsigned overflow

 .

DONE:

DETECTING SIGNED OVERFLOW

* The Overflow Flag (OF) is set if there is signed overflow. The Carry Flag is irrelevant for signed operations. Signed overflow occurs if the result is too large to fit into the destination operand or if the destination sign bit changes. In a program unsigned overflow can be detected by a construct of the form:

Flag modifying instruction

JO SIGNED\_OVERFLOW

.

. ; action if there is no signed overflow

.

JMP DONE

SIGNED\_OVERFLOW:

 .

 . ; action if there is signed overflow

 .

DONE:

HEXADECIMAL I/O ROUTINES

1. Hexadecimal output.

Write a procedure to display the contents of BX in hexadecimal.

The algorithm uses the conversions:

DecimalNumber(0, 1, 2, . . . ,9) + 30H HexadecimalDigit('0', '1', '2', . . . ,'9')

DecimalNumber(10, 11, 12, 13, 14, 15) + 37H HexadecimalDigit('A', 'B', 'C', 'D', 'E', 'F')

There are 4 hexadecimal digits in a 16-bit operand. The algorithm displays one hexadecimal digit of BX at a time:

for 4 times do

 MOV DL , BH ; put the current two highest hex digits of BX in DL

 Shift DL 4 times to the right ; leave the current highest hex digit of BX in DL

 if((DL) ≤ 9)then

 (DL) := (DL) + 30H ; convert to hex digit in {'0', '1', '2', . . . , '9'}

 else

 (DL) := (DL) + 37H ; convert to hex digit in {'A', 'B', 'C', 'D', 'E', 'F'}

 endif

 output: (DL)

 Rotate BX left 4 times ; put the next hex digit in the highest 4 bits of BX

 endfor

 output: 'H'

HEX\_DISPLAY PROC

 ; Displays the contents of BX in hexadecimal

 PUSH AX

 PUSH CX

 PUSH DX

 MOV AH , 02H

 MOV CX , 4

 FOR: MOV DL , BH

 PUSH CX

 MOV CL , 4

 SHR DL , CL

 CMP DL , 9

 JA LETTER

 ADD DL , 30H

 JMP DISPLAY

 LETTER: ADD DL , 37H

 DISPLAY: INT 21H

 ROL BX , CL

 POP CX

 LOOP FOR

 MOV DL , 'H'

 INT 21H

 POP DX

 POP CX

 POP AX

 RET

 HEX\_DISPLAY ENDP

1. Hexadecimal input

Write a procedure to input a hexadecimal value in the BX register. The reading should stop if either 4 hexadecimal digits have been read or if the "Enter" key (ASCII code: 0DH) is pressed. For any invalid input, the procedure should beep and give the user a chance of entering another value. If only 0DH is entered the procedure should set the Carry Flag.

The algorithm uses the following conversions:

character in {'0' , '1' , '2' , . . . , '9'} -30H digit in {0 , 1 , 2 , . . . , 9}

character in {'A' , 'B' , 'C', . . . , 'F'} -37H digit in {10 , 11 , 12 , . . . , 15}

character in {'a' , 'b' , 'c', . . . , 'f'} -57H digit in {10 , 11 , 12 , . . . , 15}

The pseudo-code algorithm is:

 MOV BX , 0

 MOV CL , 4

 MOV CH , 0 ; counts the number of hexadecimal digits read

 do

 {

 Read: character

 if(character is a hexadecimal digit)then

 {

 convert character to binary

 INC CH

 SHL BX , CL ; shift zeroes in the lowest 4 bits of BX

 Insert character in the lowest 4 bits of BX

 }

 else if( character ≠ 0DH )then

 {

 Beep ;

 Move the cursor back ;

 }

 else

 break ;

 endif

 } while ( (CH) < 4 )

 if( (CH) = 0 )then

 STC

 else

 CLC

 endif

HEX\_INPUT PROC

 ; Inputs a 16-bit hexadecimal value in the BX register

 PUSH AX

 PUSH CX

 PUSH DX

 MOV BX , 0

 MOV CL , 4

 MOV CH , 0

 START: MOV AH , 01H

 INT 21H

 CMP AL , '0'

 JB INVALID?

 CMP AL , '9'

 JBE VALID1

 CMP AL , 'A'

 JB INVALID

 CMP AL , 'F'

 JBE VALID2

 CMP AL , 'a'

 JB INVALID

 CMP AL , 'f'

 JBE VALID3

 JMP INVALID

 VALID1: SUB AL , 30H

 JMP INSERT

 VALID2: SUB AL , 37H

 JMP INSERT

 VALID3: SUB AL , 57H

 INSERT: INC CH

 SHL BX , CL

 OR BL , AL

 JMP NEXT

 INVALID?: CMP AL , 0DH

 JE END\_DO\_WHILE

 INVALID: MOV AH , 02H ; beep

 MOV DL , 07H ;

 INT 21H ;

 MOV DL , 08H ; backspace

 INT 21H ;

 NEXT: CMP CH , 4

 JB START

 END\_DO\_WHILE:

 CMP CH , 0

 JE NO\_INPUT

 CLC

 JMP EXIT

 NO\_INPUT: STC

 EXIT: POP DX

 POP CX

 POP AX

 RET

 HEX\_INPUT ENDP

COSC 204 LABWORK#9

Objectives:

1. FILE I/O using File handles. (DOS functions: 3CH, 5BC, 3DH, 3EH, 41H, 42H, 3FH, 40H)
2. Searching for a text file in a directory (excluding its subdirectories). (DOS functions: 4EH, 1AH, 4FH)

# PROGRAMMING ASSIGNMENT#1

Write an 8086 Assembly language program that prompts for and reads the path, of maximum length 80, to a text file. If the file already exists the user should be prompted with a Y/N (Yes or No) question as to whether he wants to overwrite the existing file or not. If the answer is “N”, the program should terminate. If the answer is “Y” or if the text file does not exist, the program should prompt and then read from the user some text of maximum size 300 bytes. The text should then be written to the existing text file or to a new created text file respectively.

# PROGRAMMING ASSIGNMENT#2

Using a text editor create two text files. Write some text, not exceeding 500 characters, in each of the files. Write an 8086 Assembly language program that will append the text of one of the text files to the other text file. Your program must then display the contents of the resulting file on the screen.