**Unit VI**

**Assembly Programming**

Assembly language is a low-level programming language for a computer or other programmable device specific to a particular computer architecture in contrast to most high-level programming languages, which are generally portable across multiple systems. Assembly language is converted into executable machine code by a utility program referred to as an assembler like NASM, MASM, etc.

## What is Assembly Language?

Each personal computer has a microprocessor that manages the computer's arithmetical, logical, and control activities.

Each family of processors has its own set of instructions for handling various operations such as getting input from keyboard, displaying information on screen and performing various other jobs. These set of instructions are called 'machine language instructions'.

A processor understands only machine language instructions, which are strings of 1's and 0's. However, machine language is too obscure and complex for using in software development. So, the low-level assembly language is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form.

### Advantages Of Assembly Language

1. Programs written in machine language are replaceable by mnemonics which are easier to remember.
2. Memory Efficient.
3. It is not required to keep track of memory locations.
4. Faster in speed.
5. Easy to make insertions and deletions.
6. Hardware Oriented.
7. Requires fewer instructions to accomplish the same result.

### Disadvantages Of Assembly Language

1. Long programs written in such languages cannot be executed on small sized computers.
2. It takes lot of time to code or write the program, as it is more complex in nature.
3. Difficult to remember the syntax.
4. Lack of portability of program between computers of different makes.
5. No SDKs (System Development Kit).

**An assembly program can be divided into three sections −**

* The **data** section,
* The **bss** section, and
* The **text** section.

## The *data* Section

The **data** section is used for declaring initialized data or constants. This data does not change at runtime. You can declare various constant values, file names, or buffer size, etc., in this section.

The syntax for declaring data section is −

section.data

## The *bss* Section

The **bss** section is used for declaring variables. The syntax for declaring bss section is −

section.bss

## The *text* section

The **text** section is used for keeping the actual code. This section must begin with the declaration **global \_start**, which tells the kernel where the program execution begins.

The syntax for declaring text section is −

section.text

global \_start

\_start:

## Comments

Assembly language comment begins with a semicolon (;). It may contain any printable character including blank. It can appear on a line by itself, like −

; This program displays a message on screen

or, on the same line along with an instruction, like −

add eax, ebx ; adds ebx to eax

## Assembly Language Statements

Assembly language programs consist of three types of statements −

* Executable instructions or instructions,
* Assembler directives or pseudo-ops, and
* Macros.

The **executable instructions** or simply **instructions** tell the processor what to do. Each instruction consists of an **operation code** (opcode). Each executable instruction generates one machine language instruction.

The **assembler directives** or **pseudo-ops** tell the assembler about the various aspects of the assembly process. These are non-executable and do not generate machine language instructions.

**Macros** are basically a text substitution mechanism.

## Syntax of Assembly Language Statements

Assembly language statements are entered one statement per line. Each statement follows the following format −

[label] mnemonic [operands] [;comment]

The fields in the square brackets are optional. A basic instruction has two parts, the first one is the name of the instruction (or the mnemonic), which is to be executed, and the second are the operands or the parameters of the command.

Following are some examples of typical assembly language statements −

INC COUNT ; Increment the memory variable COUNT

MOV TOTAL, 48 ; Transfer the value 48 in the

; memory variable TOTAL

ADD AH, BH ; Add the content of the

; BH register into the AH register

AND MASK1, 128 ; Perform AND operation on the

; variable MASK1 and 128

ADD MARKS, 10 ; Add 10 to the variable MARKS

MOV AL, 10 ; Transfer the value 10 to the AL register

## The Hello World Program in Assembly

The following assembly language code displays the string 'Hello World' on the screen −

section .text

global \_start ;must be declared for linker (ld)

\_start: ;tells linker entry point

mov edx,len ;message length

mov ecx,msg ;message to write

mov ebx,1 ;file descriptor (stdout)

mov eax,4 ;system call number (sys\_write)

int 0x80 ;call kernel

mov eax,1 ;system call number (sys\_exit)

int 0x80 ;call kernel

section .data

msg db 'Hello, world!', 0xa ;string to be printed

len equ $ - msg ;length of the string

When the above code is compiled and executed, it produces the following result −

Hello, world!

## Compiling and Linking an Assembly Program in NASM

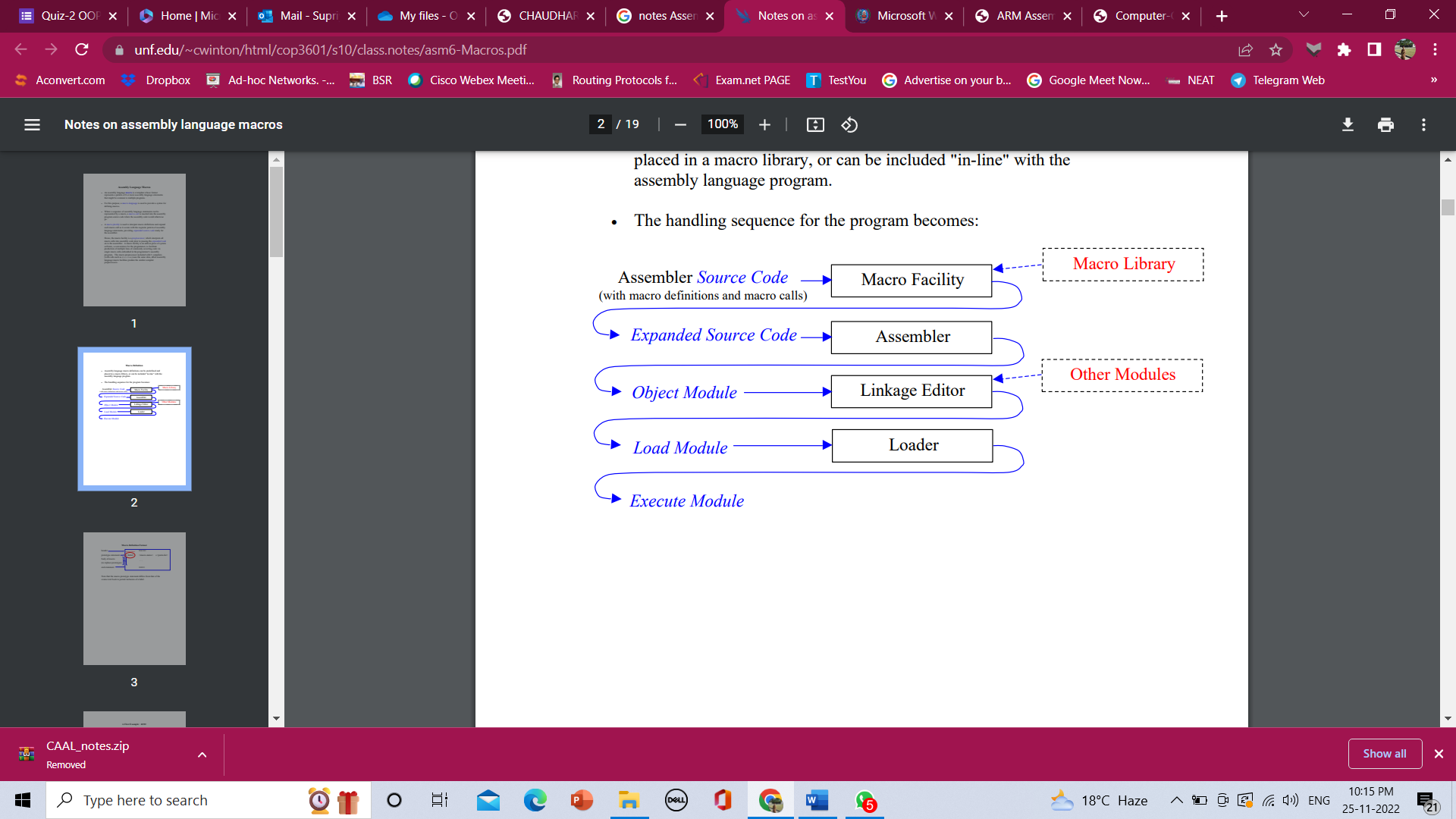
Make sure you have set the path of **nasm** and **ld** binaries in your PATH environment variable. Now, take the following steps for compiling and linking the above program −

* Type the above code using a text editor and save it as hello.asm.
* Make sure that you are in the same directory as where you saved **hello.asm**.
* To assemble the program, type **nasm -f elf hello.asm**
* If there is any error, you will be prompted about that at this stage. Otherwise, an object file of your program named **hello.o** will be created.
* To link the object file and create an executable file named hello, type **ld -m elf\_i386 -s -o hello hello.o**
* Execute the program by typing **./hello**

If you have done everything correctly, it will display 'Hello, world!' on the screen.

# Macros

A macro is an extension to the basic ASSEMBLER language. They provide a means for generating a commonly used sequence of assembler instructions/statements. The sequence of instructions/statements will be coded ONE time within the macro definition. Whenever the sequence is needed within a program, the macro will be "called".



A macro definition precedes all CSECTs and DSECTs. It consists of four parts:

1. The macro instruction MACRO starting in column 10
2. The prototype statement (this line specifies the macro name and the arguments that it takes)
3. The macro body
4. The macro instruction MEND starting in column 10

MACRO signals the beginning of a macro definition.

### Prototype format:

label macro-name arguments (0 to 200 possible)

Column: 1 10 16(usually)

The label in the prototype is optional. It can be used to put a label at the beginning of one of the lines of the loop body.

The macro body contains the macro instructions to be executed and the assembly instructions to be copied into the code.

MEND signals the end of a macro definition.

For example:

MACRO

EXMPL1

LA 1,PARMLIST

L 15,=V(BUILD)

BALR 14,15

MEND

To call the EXMPL1 macro:

EXMPL1

In the assembly, you'll see:

+ LA 1,PARMLIST

+ L 15,=V(BUILD)

+ BALR 14,15

The + in column 1 indicates that the lines of code were in a macro.

### Variable Symbols

**Variable symbols** are symbols that can be assigned values by either the programmer or the assembler. There are three types:

1. Symbolic Parameters
2. System Variables
3. SET Variables

Variable symbol names are:

1. Two to eight characters in length
2. The first character is ALWAYS an ampersand (&)
3. The second character is ALWAYS a letter
4. The rest of the characters are either letters or digits

**Symbolic Parameters** are used in a macro definition and are assigned a value by the programmer. When the macro is called, these parameters are replaced by the values that are assigned to them. There are two types of symbolic parameters:

1. Positional Parameters
2. Keyword Parameters

**Positional parameters** are symbolic parameters that must be specified in a specific order every time the macro is called. The parameter will be replaced within the macro body by the value specified when the macro is called.

MACRO

&LABEL EXMPL1 &SUBRTN,&PARMS

&LABEL LA 1,&PARMS

L 15,=V(&SUBRTN)

BALR 14,15

MEND

The new call of the EXMPL1 macro:

CALL1 EXMPL1 BUILD,PARM1

In the assembly:

+CALL1 LA 1,PARM1

+ L 15,=V(BUILD)

+ BALR 14,15

**Keyword parameters** are symbolic parameters that can be specified in any order when the macro is called. The parameter will be replaced within the macro body by the value specified when the macro is called. These parameters can be given a default value. If no default value is specified and if the parameter is not given a value when the macro is called, then the parameter will be replaced by a null string.

Each keyword parameter will have an equal sign (=) as the last character of the parameter name.

MACRO

&LABEL EXMPL1 &SUBRTN=BUILD,&PARMS=

&LABEL LA 1,&PARMS

L 15,=V(&SUBRTN)

BALR 14,15

MEND

The new call of the EXMPL1 macro:

EXMPL1 PARMS=PARM2,SUBRTN=PRINT

In the assembly:

+ LA 1,PARM2

+ L 15,=V(PRINT)

+ BALR 14,15

EXMPL1 PARMS=PARM3

In the assembly:

+ LA 1,PARM3

+ L 15,=V(BUILD)

+ BALR 14,15

EXMPL1

In the assembly:

+ LA 1,

+ L 15,=V(BUILD)

+ BALR 14,15

If a combination of positional and keyword parameters is used, all of the positional parameters must be coded BEFORE the keyword parameters.

|  |  |
| --- | --- |
| **Macro** | **Subroutine** |
| Macro can be called only in the program it is defined. | Subroutine can be called from other programs also. |
| macro doesn't have any return statement | .but a subroutine can have . |
| Macro can be called only after its definition. | This is not true for Subroutine. |
| Macro is used when same thing is to be done in a program a number of times. | Subroutine is used for modularization. |