Chapter Two

Assembly Language Programming

Assembly Language Programming

- Assembly language programming is writing machine instructions in mnemonic form, using an assembler to convert these mnemonics into actual processor instructions and associated data.
- An assembly language is a low-level programming language for microprocessors and other programmable devices.

Features of assembly language programming

- Assembly language is the most basic programming language available for any processor.
- Assembly languages generally lack high-level conveniences such as variables and functions.
- It has the same structures and set of commands as machine language, but it allows a programmer to use names instead of numbers.
- This language is still useful for programmers when speed is necessary or when they need to carry out an operation that is not possible in high-level languages.

Some important features of assembly language programming are:-

- Allows the programmer to use mnemonics when writing source code programs, like 'ADD' (addition), 'SUB' (subtraction), JMP (jump) etc.
- Variables are represented by symbolic names, not as memory locations, like MOV A, here 'A' is the variable.
- Symbolic code, and error checking,
- Changes can be quickly and easily incorporated with a re-assembly,
- Programming aids are included for relocation and expression evaluation.

Advantages of assembly language programming

- Easy to understand and use
- Easy to locate and correct errors
- Easy to modify
- No worry to address.
- Efficient than machine language programming

Disadvantages of assembly language programming

- The programmer requires knowledge of the processor architecture and instruction set.
- Machine language coding
- Many instructions are required to achieve small tasks
- Source programs tend to be large and difficult to follow

- In general, programming of a microprocessor usually takes several iterations before the right sequence of machine code instructions is written.
- The process, however, is facilitated using a special program called an "**Assembler**".
- The Assembler allows the user to write alphanumeric instructions, or mnemonics, called Assembly Language instructions.
- The Assembler, in turn, generates the desired machine instructions from the Assembly Language instructions.

ASSEMBLER DIRECTIVES

- Assembler directives are the commands to the assembler that direct the assembly process.
- They indicate how an operand is treated by the assembler and how assembler handles the program.
- They also direct the assembler how program and data should arrange in the memory.

ALP's are composed of two type of statements.

(i) The instructions which are translated to machine codes by assembler.

(ii) The directives that direct the assembler during assembly process, for which no machine code is generated.

1. ASSUME: Assume logical segment name.

The ASSUME directive is used to inform the assembler the names of the logical segments to be assumed for different segments used in the program .In the ALP each segment is given name.

Syntax: ASSUME segreg: segname,… segreg: segname

Ex: ASSUME CS:CODE

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

UNIT-2 8086 ASSEMBLY LANGUAGE PROGRAMMING ECE DEPARTMENT

2. DB: Define Byte

The DB directive is used to reserve byte or bytes of memory locations in the available memory.

Syntax: Name of variable DB initialization value.

Ex: MARKS DB 35H,30H,35H,40H

NAME DB “VARDHAMAN”

3. **DW: Define Word**

The DW directive serves the same puposes as the DB directive,but it now makes the assembler reserve the number of memory words(16-bit) instead of bytes.

Syntax: variable name DW initialization values.

Ex: WORDS DW 1234H,4567H,2367H

WDATA DW 5 Dup(522h) (or) Dup(?)

4. DD: Define Double:

The directive DD is used to define a double word (4bytes) variable.

Syntax: variablename DD 12345678H

Ex: Data1 DD 12345678H

5. DQ: Define Quad Word

- This directive is used to direct the assembler to reserve 4 words (8 bytes) of memory for the specified variable and may initialize it with the specified values.

Syntax: Name of variable DQ initialize values.

Ex: Data1 DQ 123456789ABCDEF2H

6. DT: Define Ten Bytes

The DT directive directs the assembler to define the specified variable requiring 10 bytes for its storage and initialize the 10-bytes with the specified values.

Syntax: Name of variable DT initialize values.

Ex: Data1 DT 123456789ABCDEF34567H

7. END: End of Program

The END directive marks the end of an ALP. The statement after the directive END will be ignored by the assembler.

8. ENDP: End of Procedure

The ENDP directive is used to indicate the end of procedure. In the AL programming the subroutines are called procedures.

Ex: Procedure Start

- :
- Start ENDP

UNIT-2 8086 ASSEMBLY LANGUAGE PROGRAMMING ECE DEPARTMENT

9. ENDS: End of segment

The ENDS directive is used to indicate the end of segment.

Ex: DATA SEGMENT

:

DATA ENDS

10. OFFSET: offset of a label

- When the assembler comes across the OFFSET operator along with a label, it first computing the 16-bit offset address
- of a particular label and replace the string 'OFFSET LABEL' by the computed offset address.
- Ex : MOV SI, offset list

PROC: Procedure

- The PROC directive marks the start of a named procedure in the statement.

Ex: RESULT PROC NEAR

ROUTINE PROC FAR

Laboratory requirements

- For the entire lab session, we will use the following software
  - Linux OS(Linux Mint)
  - Netwide Assembler NASM (8086_Assembler)

Steps to execute an assembly program

1. Write the code on any text editor
2. Save the file as **file_name.asm** and set the type to All files.
   - Make sure that you are in the same directory as where you saved your program
3. Assemble by typing: **nasm -f elf file_name.asm**
   - If there is any error, you will be prompted about that at this stage. Otherwise, an object file of your program named **file_name.o** will be created.
4. To link the object file and create an executable file:
   - **ld -m elf_i386 -s -o file_name file_name.o**
5. Execute by typing: **./file_name**

Assembling the program:

- The assembler is used to convert the assembly language instructions to machine code.
- It is used immediately after writing the Assembly Language program.
- It starts by checking the syntax, or validity of the structure, of each instruction in the source file.
- If any errors are found, the assembler displays a report on these errors along with a brief explanation of their nature.
- However, if the program does not contain any errors, the assembler produces an object file that has the same name as the original file but with the “o” extension

linking

- The linker is used to convert the object file to an executable file.
- The executable file is the final set of machine code instructions that can directly be executed by the microprocessor.
- An object file may represent one segment of a long program.
  - This segment cannot operate by itself, and must be integrated with other object files representing the rest of the program, in order to produce the final self-contained executable file.

Executing the program

- The executable file contains the machine language code.
- It can be loaded in the RAM and be executed by the microprocessor simply by typing, from the DOS prompt, the name of the file followed by the Carriage Return Key (Enter Key).
- If the program produces an output on the screen, or a sequence of control signals to control a piece of hardware, the effect should be noticed almost immediately.
- However, if the program manipulates data in memory, nothing would seem to have happened as a result of executing the program.

Structure of an Assembly Language Program

- An assembly language program is written according the following structure and includes the following assembler directives:
  - The data section
  - The bss section
  - 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**

Program Structure

- **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 main or global _start which tells the kernel where the program execution begins.
  - The syntax for declaring text section is:
    - **section .text**
    - **global main/ _start**
    - **main:/_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**
  - **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

Symbols

- A symbolic name consists of a sequence of letters, digits, and special characters, with the following restrictions:
- A symbol cannot begin with a numeric digit.
- A name can have any combination of upper and lower case alphabetic characters.
  - The assembler treats upper and lower case equivalently.
- A symbol may contain any number of characters, however only the first 31 are used.
  - The assembler ignores all characters beyond the 31st.
- The _, $, ?, and @ symbols may appear anywhere within a symbol.
  - However, $ and ? are special symbols; you cannot create a symbol made up solely of these two characters.
- A symbol cannot match any name that is a reserved symbol

Symbols

- Some examples of valid symbols include:
  - L1 Bletch RightHere
  - Right_Here Item1 __Special $1234
  - @Home $_@1 Dollar$
  - WhereAmI? @1234
- Some examples of illegal symbols include:
  - 1TooMany - Begins with a digit.
  - Hello.There - Contains a period in the middle of the symbol.
  - $ - Cannot have $ or ? by itself.
  - LABEL - Assembler reserved word.
  - Right Here - Symbols cannot contain spaces.
  - Hi,There - or other special symbols besides _, ?, $, and @.

Variables

- Here we will simply use the 8086 registers as the variables in our programs.
- Registers have predefined names and do not need to be declared.

Variable Declaration

- NASM provides various define directives for reserving storage space for variables. The define assembler directive is used for allocation of storage space.
- It can be used to reserve as well as initialize one or more bytes.

Allocating Storage Space for Initialized Data

- The syntax for storage allocation statement for initialized data is:

[variable-name] define-directive initial-value [, initial-value] ...

- Where, variable-name is the identifier for each storage space. The assembler associates an offset value for each variable name defined in the data segment.
- There are five basic forms of the define directive:

Directive	Purpose	Storage Space
DB	Define Byte	allocates 1 byte
DW	Define Word	allocates 2 bytes
DD	Define Doubleword	allocates 4 bytes
DQ	Define Quadword	allocates 8 bytes
DT	Define Ten Bytes	allocates 10 bytes

VARIABLE DECLARATION

- Each variable has a type and assigned a memory address.
- **Example**
- **Byte Variables**
  - Assembler directive format assigning a byte variable
    - ***Syntax: Name DB initial value***
  - A question mark ("?") place in initial value leaves variable uninitialized
    - **L DB 4** ;define variable L with initial value 4
    - **J DB ?** ;Define variable J with uninitialized value
    - **Name DB "Course"** ;allocate 6 bytes for name
    - **K DB 5, 3,-1** ;allocate 3 bytes

VARIABLE DECLARATION

- L1 db 0 ; byte labeled L1 with initial value 0
- L2 dw 1000 ; word labeled L2 with initial value 1000
- L3 db 110101b ; byte initialized to binary 110101
- L4 db 12h ; byte initialized to hex 12 (18 in decimal)
- L5 db 17o ; byte initialized to octal 17 (15 in decimal)
- L6 dd 1A92h ; double word initialized to hex 1A92
- L7 resb 1 ; 1 uninitialized byte
- L8 db "A" ; byte initialized to ASCII code for A (65)
- Double quotes and single quotes are treated the same. Consecutive data definitions are stored sequentially in memory. That is, the word L2 is stored immediately after L1 in memory. Sequences of memory may also be defined.
- L9 db 0, 1, 2, 3 ; defines 4 bytes
- L10 db "w", "o", "r", 'd', 0 ; defines a C string = "word"
- L11 db 'word', 0 ; same as L10

Allocating Storage Space for Uninitialized Data

- The reserve directives are used for reserving space for uninitialized data.
- The reserve directives take a single operand that specifies the number of units of space to be reserved.
- Each define directive has a related reserve directive.
- There are five basic forms of the reserve directive:

Directive	Purpose
RESB	Reserve a Byte
RESW	Reserve a Word
RESD	Reserve a Doubleword
RESQ	Reserve a Quadword
REST	Reserve a Ten Bytes

Allocating Storage Space for Uninitialized Data

- **Multiple Definitions**
  - You can have multiple data definition statements in a program. For example:
    - choice DB 'Y' ;ASCII of y = 79H
    - number1 DW 12345 ;12345D = 3039H
    - number2 DD 12345679 ;123456789D = 75BCD15H
- The assembler allocates contiguous memory for multiple variable definitions.
- **Multiple Initializations**
- The **TIMES** directive allows multiple initializations to the same value.
- For example, **an array named marks of size 9** can be defined and initialized to zero using the following statement:
  - marks TIMES 9 DW 0
- The TIMES directive is useful in defining arrays and tables.

CONSTANTS

- There are several directives provided by NASM that define constants. Some of them are :
  - EQU
  - %assign
  - %define
- **The EQU Directive**
  - The EQU directive is used for defining constants.
  - The syntax of the EQU directive is as follows:
    - ***CONSTANT_NAME EQU expression***
  - For example, ***TOTAL_STUDENTS equ 50***
  - You can then use this constant value in your code, like:
    - mov ecx, TOTAL_STUDENTS
    - cmp eax, TOTAL_STUDENTS
- The operand of an EQU statement can be an expression:
    - **LENGTH equ 20**
    - **WIDTH equ 10**
    - **AREA equ length * width** Above code segment would define AREA as 200.

CONSTANTS

- **The %assign Directive**
  - The %assign directive can be used to define numeric constants like the EQU directive. This directive allows redefinition.
  - For example, you may define the **constant TOTAL as:**
    - **%assign TOTAL 10**
    - **Later in the code, you can redefine it as:**
    - **%assign TOTAL 20**
  - This directive is case-sensitive.
- **The %define Directive**
  - The %define directive allows defining both numeric and string constants. This directive is similar to the #define in C.
  - For example, you may define the constant PTR as:
    - %define PTR [EBP+4]
  - The above code replaces PTR by [EBP+4].
  - This directive also allows redefinition and it is case-sensitive.

Linux System Calls

- System calls are APIs for the interface between the user space and the kernel space.
- You can make use of Linux system calls in your assembly programs. You need to take the following steps for using Linux system calls in your program:
  - **Put the system call number in the EAX register.**
  - **Store the arguments to the system call in the registers EBX, ECX, etc.**
  - **Call the relevant interrupt (80h).**
  - **The result** is usually returned in the **EAX register**.
- There are six registers that store the arguments of the system call used. These are the EBX, ECX, EDX, ESI, EDI, and EBP. These registers take the consecutive arguments, starting with the EBX register.
- The following code snippet shows the use of the system call **sys_exit**:
  - **mov eax,1; system call number (sys_exit)**
  - **int 0x80 ; call kernel**

Linux System Calls

here are some of system calls commonly used

No	Func Name	Description
1	exit	terminate the current process
2	fork	create a child process
3	read	read from a file descriptor
4	write	write to a file descriptor
5	open	open a file or device
6	close	close a file descriptor
7	waitpid	wait for process termination

- The following code snippet shows the use of the system call sys_write:

```
mov edx,4 ; 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(interrupt)
```

Some practical Examples

Important!!!

1. To display a message/value
   1. place the address of the message/value to be displayed to ecx (e.g. mov ecx,msg)
   2. place the length of the message/value to be displayed to edx register (e.g. mov edx,length)
   3. Place the Sys_write call number to eax (e.g. mov eax,4)
   4. Place the standard output device to ebx(e.g. mov ebx,1)
   5. Call for interrupt (int 0x80)

2. To read data from keyboard
   1. Place the sys_read call number to eax (e.g. mov eax, 3)
   2. Place the standard input device no to ebx (e.g.mov ebx, 2)
   3. Place the address of the data to ecx (e.g. mov ecx, num)
   4. Place the size of the data to edx (e.g. mov edx, 5)
   5. Call for interrupt (int 80h)

Some practical Examples

1. A program to Display a message "Hello World"

```
section.data
msg db 'Hello World!', 0Ah; assign msg variable with your message string
SECTION .text
global  _start
_start:
mov edx, 13 ; number of bytes to write - one for each letter plus 0Ah (line feed character)
mov  ecx, msg   ; move the memory address of our message string into ecx
mov  ebx, 1  ; write to the STDOUT file
mov  eax, 4  ; invoke SYS_WRITE (kernel opcode 4)
int  80h      ; intrrupt
```

Some practical Examples

2. A program to Read and Display a message

```
section .data
SYS_EXIT equ 1
SYS_READ equ 3
SYS_WRITE equ 4
STDIN equ 0
STDOUT equ 1
msg1 db "Enter Your Name", 0xA,0xD
len1 equ $- msg1
msg2 db "Hello! "
len2 equ $- msg2
segment .bss
name resb 20;reserving a memory for name
```

Some practical Examples

```
section .text
global _start ;must be declared for using gcc
 _start: ;tell linker entry point
;prompting for message 1
mov eax, SYS_WRITE
mov ebx, STDOUT
mov ecx, msg1
mov edx, len1
int 0x80
;reading name
mov eax, SYS_READ
mov ebx, STDIN
mov ecx, name
mov edx, 20
int 0x80
```

Some practical Examples

```
;displaying the message
mov eax, SYS_WRITE
mov ebx, STDOUT
mov ecx, msg2
mov edx, len2
int 0x80
; print the Message
mov eax, SYS_WRITE
mov ebx, STDOUT
mov ecx, name
mov edx, 20
int 0x80
;exit:
mov eax, SYS_EXIT
int 0x80
```

Some practical Examples

3. A program to Read and Display a number

```
section .data ;Data segment
;Ask the user to enter a number
msg1 db 'Please enter a number: '
len1 equ $-msg1 ;The length of the message
 msg2 db 'The Entered number is: '
 len2 equ $-msg2
 section .bss
num resb 5 ;Uninitialized data
```

Some practical Examples

```
section .text ;Code Segment
 global _start
_start:
 ;User prompt
 mov eax, 4 ;sys_write
 mov ebx, 1
 mov ecx, msg1
 mov edx, len1
 int 80h
;Read and store the user input
 mov eax, 3 ;sys_read
 mov ebx, 2 ;STDIN
 mov ecx, num
 mov edx, 5 ;5 bytes  of that information
 int 80h
```

Some practical Examples

```
;Output the message 'The entered number is: '
 mov eax, 4
 mov ebx, 1
 mov ecx, msg2
 mov edx, len2
 int 80h
 ;Output the number entered
 mov eax, 4
 mov ebx, 1
 mov ecx, num
 mov edx, 5
 int 80h
; Exit code
 mov eax, 1
 mov ebx, 0
 int 80h
```

Some practical Examples

4. Aprogram to add two integers

```
section .text
global _start
_start:
mov eax,'2'
sub eax,'0' ;to connvert to ASCII
mov ebx,'4'
sub ebx,'0';to convert to ASCII
add eax,ebx
add eax,'0' ;to convert to Decimal
mov [sum],eax ;copying the result to sum
mov ecx,msg ; placing the address of the message to ecx
mov edx,len  ;placing the length of the message to edx
mov ebx,1 ;stdout
mov eax,4 ;sys_write
int 0x80 ;call for interrupt
```

Some practical Examples

4. Aprogram to add two integers

```
mov ecx,sum ;
mov edx,10   ; size of the sum
mov ebx,1 ; stdout
mov eax,4 ;sys_write
int 0x80

section .data
msg db "The sum is:",0xA,0xD
len equ $-msg
section .bss
sum resb 10
```