Chapter 05

# TASM Assembler

## Table of content

## ■ Introduction

This chapter is an introduction to assembly language programming with 80x86. First the basic form of a program is explained, followed by the steps required to edit, assemble, link, and run a program.

## ■ Directives and a sample program

- A given assembly language program is a series of statements, or lines, which are either **assembly language instructions** such as ADD and MOV, or statements called **directives**.
- Directives (also called pseudo-instructions) give directions to assembler about how it should translate the assembly language instructions into machine code.
- An assembly language instruction consists of four fields:

  [**Label**] **mnemonic** [**operands**] [; **comment**]

  Brackets indicate that the field is optional

**Label**

- ✍ The **label** field allows the program to refer to a line of code by name. The label field cannot exceed 31 characters.
- ✍ Labels for directives do not need to end with a colon.
- ✍ A label must end with a colon when it refers to an opcode generating instruction

**Mnemonics**

- ✍ The assembly language **mnemonics** (**instruction**) and operand(s) fields together perform the real work of the program such as

```
ADD AL,BL
MOV AX,6764
```

- ADD and MOV are the mnemonics opcodes and AL, BL and AX, 6764 are the operands. Instead of a mnemonics and operand, these two fields could contain assembler pseudo-instructions, or directives. They are used by the assembler to organize the program as well as other output files.
- Directives do not generate any machine code.
- The commands DB, END, and ENDP are examples of directives

- The comment field begins with a ";". Comments may be at the end of a line or on a line by themselves

**Model definition**

- Model is a directive that selects the size of the memory model. Among the options for the memory model are **SMALL**, **MEDIUM**, **COMPACT**, and **LARGE**

```
.MODEL SMALL          ; define the model as small
```

- SMALL is one of the most widely used memory models for assembly language programs and is sufficient for the programs here in this book.
- The **small** model uses a maximum of 64K bytes of memory for code and another 64K bytes for data.

**Segment definition**

- The 80x86 CPU has four segment registers: CS, DS, SS, and ES.

- Every line of an assembly language program must correspond to one of these segments. The simplified segment definition format uses three simple directives:

  **.CODE** correspond to CS

  **.DATA** (DS)

  **.STACK** (SS)

**Segments of a program**

- Although one can write an assembly language program that uses only one segment, normally a program consists of at least three segments: the **stack segment**, the **data segment**, and the **code segment**.

  **.STACK** ; mark the beginning of the stack segment

  **.DATA** ; mark the beginning of the data segment

  **.CODE** ; mark the beginning of the code segment

- Assembly language statements are grouped into segments in order to be recognized by the assembler and consequently by the CPU.
- The **stack segment** defines storage for the stack, the **data segment** defines the data that the program will use, and the **code segment** contains the assembly language instructions.
- The following directive reserves 64 bytes of memory for the stack

```
.STACK 64
```

- Data segment

```
 .DATA
DATA1     DB    52H
DATA2     DB    29H
```

```
SUM         DB      ?
```

- The list above defines three data items: DATA1, DATA2, and SUM. Each is defined as DB (define byte).
- The **DB** directive is used by the assembler to allocate memory in byte-sized chunks.
- Memory can be allocated in different sizes, such as 2 bytes (**DW**). The data items defined in the data segment will be accessed in the code segment by their labels. DATA1 and DATA2 are given initial values in the data section. SUM is not given an initial value, but storage is set aside for it.

## Code Segment definition

## PROC directive

- A **procedure** is a group of instructions designed to accomplish a specific function.
- A code segment may consist of only one procedure, but usually is organized into several small procedures in order to make the program more structured.
- Every procedure must have a name defined by the **PROC** directive, followed by the assembly language instructions and closed by the **ENDP** directive.
- The **PROC** and the **ENDP** statements must have the same label.
- The **PROC** directive may have the option **FAR** or **NEAR**. The operating system that controls the computer must be directed to the beginning of the program in order to execute it.
- DOS requires that the entry point to the user program be a FAR procedure. From then on, either FAR or NEAR can be used.

- The DOS operating system must pass control to the program so that it may execute, but before it does that it assigns values for the segment registers.
- The operating system must do this because it knows how much memory is installed in the computer, how much of it is used by the system, and how much is available.
- Various DOS versions require different amounts of memory, and since the user program must be able to run across different versions, one cannot tell DOS to give the program a specific area of memory, say from 25FFF to 289E2. Therefore, it is the job of DOS to assign exact values for the segment registers. When the program begins executing, of three segment registers, only CS and SS have the proper values. The DS value (and ES if used) must be initialized by the program. This is done as follows:

```
MOV AX,@DATA    ; DATA refers to the start of the data segment
MOV DS,AX
```

After these two lines you can write your program, it may be like

```
MOV AL,DATA1
MOV BL,DATA2
ADD AL,BL
MOV SUM,AL
```

- **NOTE:** you may need in the last of your program to return control to the operating system, so you will write

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

## TASM layout

```
                .MODEL SMALL
                .STACK 64
                .DATA
                ;
                ; place data definition here
                ;
                .CODE
MAIN            PROC FAR
                MOV AX,@DATA
                MOV DS,AX
                ;
                ; place code here
                ;
                MOV AH,4CH
                INT    21H
MAIN            ENDP
                END    MAIN
```

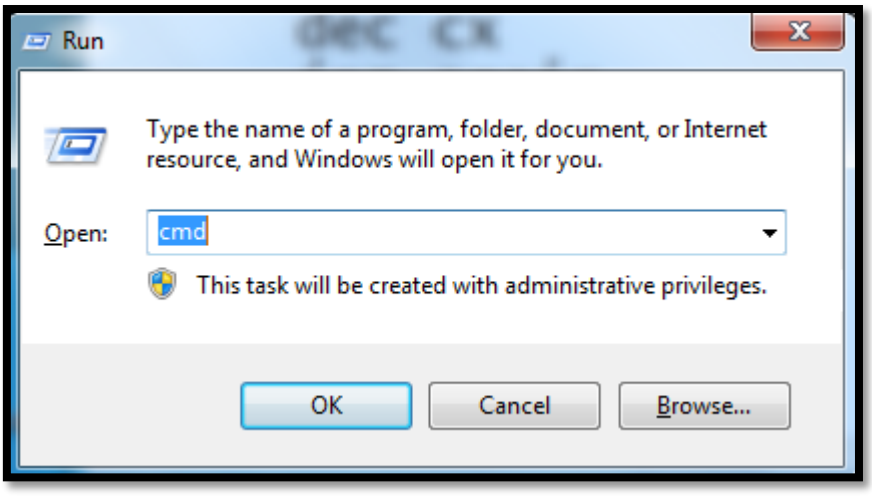
## Example

Write, run, and analyze a program that adds 5 bytes of data and saves the result. The data should be the following hex numbers: 25H, 12H, 15H, 1FH, and 2BH.

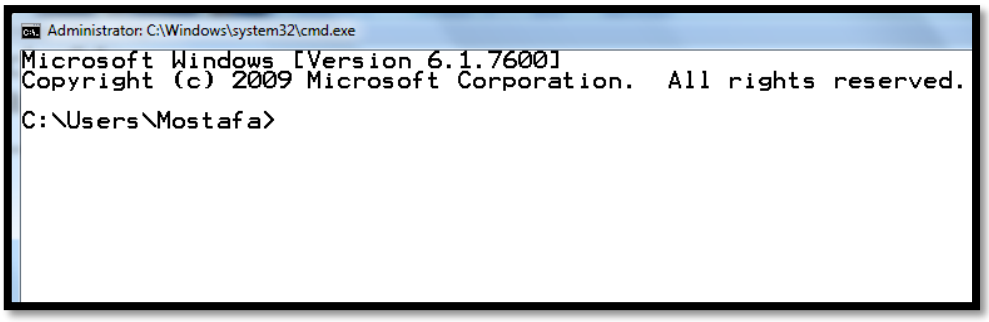
```
                .MODEL SMALL
                .STACK 64
                .DATA
DATA_IN         DB 25H, 12H, 15H, 1FH,2BH
SUM             DB ?
                .CODE
MAIN            PROC FAR
```

```
                MOV AX,@DATA
                MOV DS,AX
                MOV CX,05
                MOV BX,OFFSET DATA_IN
                MOV AL,0
AGAIN:          ADD AL,[BX]
                INC BX
                DEC CX
                JNZ AGAIN
                MOV SUM,AL
                MOV AH,4CH
                INT 21H
MAIN            ENDP
                END MAIN
```

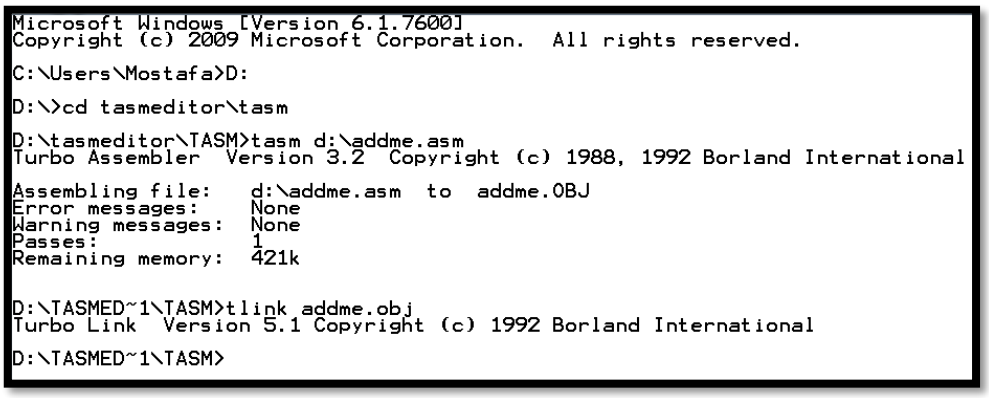
Open your DOS session by typing **cmd** in the run window




The session will start as shown below




Go to the folder in which you installed TASM




Execute your .exe file using the DEBUG utility found in your DOS shell by

```
D:\TASMED~1\TASM>debug addme.exe
-
```

You can look at the code section including the instructions in the code segment using the **u** (Unassemble the code) by

```
D:\TASMED~1\TASM>debug addme.exe
-u cs:0 19
142A:0000 B82C14          MOV     AX,142C
142A:0003 8ED8            MOV     DS,AX
142A:0005 BB0000          MOV     BX,0000
142A:0008 B90500          MOV     CX,0005
142A:000B B000            MOV     AL,00
142A:000D 0207            ADD     AL,[BX]
142A:000F 43              INC     BX
142A:0010 49              DEC     CX
142A:0011 75FA            JNZ     000D
142A:0013 A20500          MOV     [0005],AL
142A:0016 B44C            MOV     AH,4C
142A:0018 CD21            INT     21
-
```

Then you can run the exe file using **g** for "go"

```
-g

Program terminated normally
-
```

There are many ways in which any program may be written. The method that was used defined one field of data and used pointer **[BX]** to access data elements. In the method used below, a name is assigned to each data item

```
                .MODEL SMALL
                .STACK 64
                .DATA
DATA1           DB 25H
DATA2           DB 12H
DATA3           DB 15H
DATA4           DB 1FH
DATA5           DB 2BH
SUM             DB ?
                .CODE
MAIN            PROC FAR
```

```
        MOV AX,@DATA
        MOV DS,AX
        MOV AL,DATA1
        ADD AL,DATA2
        ADD AL,DATA3
        ADD AL,DATA4
        ADD AL,DATA5
        MOV SUM,AL
        MOV AH,4CH
        INT 21H
MAIN    ENDP
        END MAIN
```

There is a quite difference between these two methods of writing the same program. While in the first one the register indirect addressing mode was used to access data, in the second method the direct addressing mode was used

**Example**

Write and run a program that adds four words of data and saves the result. The values will be 234DH, 1DE6H, 3BC7H, and 566AH. Use debug to verify the sum is D364

**Solution**

```
        .MODEL SMALL
        .STACK 64
        .DATA
DATA_IN DW 234DH, 1DE6H, 3BC7H, 566AH
```

```
        ORG 10H
        SUM   DW ?
        .CODE
MAIN    PROC FAR
        MOV AX,@DATA
        MOV DS,AX
        MOV DI,OFFSET DATA_IN
        MOV CX,04
        MOV BX,0
AGAIN:  ADD BX,[DI]
        INC DI
        INC DI
        DEC CX
        JNZ AGAIN
        MOV SI,OFFSET SUM
        MOV [SI],BX
        MOV AH,4CH
        INT 21H
MAIN    ENDP
        END MAIN
```

- In previous programs where **ORG** was not used, the assembler would start at offset 0000 and use memory for each data item.

- The **ORG** directive can be used to set the offset addresses for data items. Although the programmer cannot assign exact physical addresses, one is allowed to assign offset addresses. The ORG directive here caused SUM to be stored at DS:0010, as can be seen by looking at the debug display of the data segment.

```
D:\tasmeditor\TASM>debug addmew.exe
-u cs: 0 19
142A:0000 B82C14          MOV     AX,142C
142A:0003 8ED8            MOV     DS,AX
142A:0005 BF0000          MOV     DI,0000
142A:0008 B90400          MOV     CX,0004
142A:000B BB0000          MOV     BX,0000
142A:000E 031D            ADD     BX,[DI]
142A:0010 47              INC     DI
142A:0011 47              INC     DI
142A:0012 49              DEC     CX
142A:0013 75F9            JNZ     000E
142A:0015 BE1000          MOV     SI,0010
142A:0018 891C            MOV     [SI],BX
```

```
-d 142c:0000
142C:0000  4D 23 E6 1D C7 3B 6A 56-00
142C:0010  00 00 00 00 00 00 00 00-00
```

```
-g

Program terminated normally
-d 142c:0010
142C:0010  64 D3 00 00 00 00 00 00-00
142C:0020  00 00 00 00 00 00 00 00-00
142C:0030  00 00 00 00 00 00 00 00-00
```

**Control Transfer Instructions**

- In the sequence of instructions to be executed, it is often necessary to transfer program control to a different location. There are many instructions in the 80x86 to achieve this. Before discussing that, it is necessary to explain the concept of FAR and NEAR as it applies to jump and call instructions

## FAR and NEAR

- If control is transferred to a memory location within the current code segment, it is NEAR. This is sometimes called intrasegment (within segment).
- If control is transferred outside the current code segment, it is a FAR or intersegment (between segments) jump. Since the CS:IP registers always point to the address of the next instruction to be executed, they must be updated when a control transfer instruction is executed.
- In a near jump, the IP is updated and CS remains the same, since control is still inside the current code segment. In a FAR jump, because control is passing outside the current code segment, both CS and IP have to be updated to the new values

  **NOTE:** in any control transfer instruction, such as jump or call, the IP must be changed, but only in the FAR case the CS is changed, too

## Conditional jumps

- Conditional jumps have mnemonics such as **JNZ** (jump not zero) and **JC** (jump if carry). In the conditional jump, control is transferred to a new location if a certain condition is met. The flag register is the one that indicates the current condition.
- For example, with "**JNZ label**", the processor looks at the zero flag to see if it raised. If not, the CPU starts to fetch and execute instructions from the address of label. If ZF = 1, it will not jump but will execute the next instruction below the JNZ

| Mnemonic | Condition tested | Jump if |
|---|---|---|
| **JA/JNBE** | CF = 0 and ZF = 0 | Above/ not below nor zero |
| **JAE/JNB** | CF=0 | Above or equal / not below |

| | | |
|---|---|---|
| **JB/JNAE** | CF=1 | Below/ not above nor equal |
| **JBE/JNA** | (CF or ZF) = 1 | Below or equal / not above |
| **JC** | CF = 1 | Carry |
| **JE/JZ** | ZF = 1 | Equal / zero |
| **JG/JNLE** | ((SF xor OF) or ZF) = 0 | Greater / not less nor equal |
| **JGE/JNL** | (SF xor OF) = 0 | Greater or equal / not less |
| **JL/JNGE** | (SF xor OF) =1 | Less/not greater nor equal |
| **JLE/JNG** | ((SF xor OF) or ZF)= 1 | Less or equal / not greater |
| **JNC** | CF = 0 | Not carry |
| **JNE/JNZ** | ZF = 0 | Not equal / not zero |
| **JNO** | OF = 0 | Not overflow |
| **JNP/JPO** | PF = 0 | Not parity / parity odd |
| **JNS** | SF = 0 | Not sign |
| **JO** | OF = 1 | Overflow |
| **JP/JPE** | PF = 1 | Parity / parity equal |
| **JS** | SF = 1 | Sign |

**Note:** above and below refer to the relationship of two unsigned values; greater and less refer to the relationship of two signed values

**Short jumps**

- All conditional jumps are short jumps. In a short jump, the address of the target must be within -128 to +127 bytes of the IP. In other words, the conditional jump is a two-byte instruction: one byte is the opcode of the J condition and the second byte is a value between 00 and FF. an offset range of 00 and FF gives 256 possible address; splitting between backward jumps (to -128) and forward jumps (to +127)

## Unconditional jumps

- "**JMP label**" is an unconditional jump in which control is transferred unconditionally to the target location label. The unconditional jump can take the following forms:
- **SHORT JUMP**, which is specified by the format "JMP SHORT label". This is a jump in which the address of the target location is within -128 to +127 bytes of memory relative to the address of the current IP. In this case, the opcode is EB and the operand is 1 byte in the range 00 to FF. coding the directive "short" makes the jump more efficient in that it will be assembled into a 2-byte instruction instead of a 3-byte instruction
- **NEAR JUMP**, which is the default, has the format "JMP label". This is a near jump (within the current code segment) and has the opcode E9. The target address can be any of the addressing modes of direct, register, register indirect, or memory indirect.
- **FAR JUMP** which has the format "JMP FAR PTR label". This is a jump out of the current code segment, meaning that not only the IP but also the CS is replaced with new values

## CALL statements

- Another control transfer instruction is the CALL instruction, which is used to call a procedure. Calls to procedures are used to perform tasks that need to be performed frequently. This makes a program more structured.
- The last instruction of the called subroutine must be a RET instruction which directs the CPU to POP the top 2 bytes of the stack into the IP and resume executing at offset address

## Assembly language subroutines

- In assembly language programming, it is common to have one main program and many subroutines to be called from the main program. This allows you to make each subroutine into a separate module. Each module can be tested separately and then brought together. The main program is the entry point from DOS and is a FAR as explained earlier, but subroutines called within the main program can be FAR or NEAR