Introduction to 8086 Assembly Language Programming

Prepared By
Dr. D. Jayakumar, M.Tech., Ph.D.,
Associate Professor, Dept Of ECE,
Kuppam Engineering College, Kuppam

What Is Assembly Language

- Machine-Specific Programming Language
  - one-one correspondence between statements and native machine language
  - matches machine instruction set and architecture
- IBM-PC Assembly Language
  - refers to 8086, 8088, 80186, 80286, 80386, 80486, and Pentium Processors

What Is An Assembler?

- Systems Level Program
  - translates assembly language source code to machine language
    - object file - contains machine instructions, initial data, and information used when loading the program
    - listing file - contains a record of the translation process, line numbers, addresses, generated code and data, and a symbol table

Why Learn Assembly Language?

- Learn how a processor works
- Understand basic computer architecture
- Explore the internal representation of data and instructions
- Gain insight into hardware concepts
- Allows creation of small and efficient programs
- Allows programmers to bypass high-level language restrictions
- Might be necessary to accomplish certain operations

Data Representation

- Binary 0-1
  - represents the state of electronic components used in computer systems
- Bit - Binary digit
- Byte - 8 Bits
  - smallest addressable memory location (on the IBM-PC)
- Word - 16 Bits
  - Each architecture may define its own “wordsize”
- Doubleword - 32 Bits
- Quadword - 64 Bits
- Nybble - 4 Bits

Numbering Systems

- Binary - Base 2
  - 0, 1
- Octal - Base 8
  - 0, 1, 2, … 7
- Decimal - Base 10
  - 0, 1, 2, …, 9
- Hexadecimal (Hex)
  - 0, 1, …, 9, A, B, …, F

- Raw Binary format
  - All information is coded for internal storage
  - Externally, we may choose to express the information in any numeration system, or in a decoded form using other symbols

Decoding a Byte

- Raw
  - 01010000b
- Hex
  - 50h
- Octal
  - 120_8
- Decimal
  - 80d
- Machine Instruction
  - Push AX
- ASCII Character code
  - ‘P’
- Integer
  - 80 (eighty)
- BCD
  - 50 (fifty)
- Custom code ???

Machine Language

- A language of numbers, called the Processor's Instruction Set
  - The set of basic operations a processor can perform
- Each instruction is coded as a number
- Instructions may be one or more bytes
- Every number corresponds to an instruction

Assembly Language vs Machine Language Programming

- Machine Language Programming
  - Writing a list of numbers representing the bytes of machine instructions to be executed and data constants to be used by the program
- Assembly Language Programming
  - Using symbolic instructions to represent the raw data that will form the machine language program and initial data constants

Assembly Language Instructions

- Mnemonics represent Machine Instructions
  - Each mnemonic used represents a single machine instruction
  - The assembler performs the translation
- Some mnemonics require operands
  - Operands provide additional information
    - register, constant, address, or variable
- Assembler Directives

8086 Instruction - Basic Structure

Label *Operator* *Operand[s]* *;Comment*

Label - optional alphanumeric string
1st character must be **`a-z`**,**`A-Z`**,**`?`**,**`@`**,**`_`**,**`$`**
Last character must be **`:`**

Operator - assembly language instruction
mnemonic: an instruction format for humans
Assembler translates mnemonic into hexadecimal *opcode*
example: `mov` is f8h

Operand[s] - 0 to 3 pieces of data required by instruction
Can be several different forms
Delineated by commas
immediate, register name, memory data, memory address

Comment - Extremely useful in assembler language

These fields are separated by White Space (tab, blank, \n, etc.)

8086 Instruction - Example

Label *Operator* *Operand[s]* *;Comment*

```
INIT: mov   ax, bx        ; Copy contents of bx into ax
```

Label - **`INIT:`**
Operator - **`mov`**
Operands - **`ax`** and **`bx`**
Comment - alphanumeric string between **`;`** and **`\n`**

- Not case sensitive
- Unlike other assemblers, destination operand is first
- **`mov`** is the *mnemonic* that the assembler translates into an *opcode*

Assembler Language Segment Types

- Stack
  - For dynamic data storage
  - Source file defines size
  - Must have exactly 1
- Data
  - For static data Storage
  - Source file defines size
  - Source file defines content (optional)
  - Can have 0 or more
- Code
  - For machine Instructions
  - Must have 1 or more

Using MASM Assembler

- to get help:

```
C:\> masm /h
```

- Can just invoke MASM with no arguments:

```
C:\> masm
Source Filename            [.ASM]:    hello
Object Filename        [HELLO.OBJ]:
Source Listing           [NUL.LST]:
Cross Reference          [NUL.CRF]:
```

- `.ASM` - Assembler source file prepared by programmer
- `.OBJ` - Translated source file by assembler
- `.LST` - Listing file, documents “Translation” process
  - » Errors, Addresses, Symbols, etc
- `.CRF` – Cross reference file

x86 Instruction Set Summary
(Data Transfer)

```
CBW     ;Convert Byte to Word AL → AX
CWD     ;Convert Word to Double in AX →DX,AX
IN      ;Input
LAHF    ;Load AH from Flags
LDS     ;Load pointer to DS
LEA     ;Load EA to register
LES     ;Load pointer to ES
LODS    ;Load memory at SI into AX
MOV     ;Move
MOVS    ;Move memory at SI to DI
OUT     ;Output
POP     ;Pop
POPF    ;Pop Flags
PUSH    ;Push
PUSHF   ;Push Flags
SAHF    ;Store AH into Flags
STOS    ;Store AX into memory at DI
XCHG    ;Exchange
XLAT    ;Translate byte to AL
```

x86 Instruction Set Summary
(Arithmetic/Logical)

```
AAA                     ;ASCII Adjust for Add in AX
AAD                     ;ASCII Adjust for Divide in AX
AAM                     ;ASCII Adjust for Multiply in AX
AAS                     ;ASCII Adjust for Subtract in AX
ADC                     ;Add with Carry
ADD                     ;Add
AND                     ;Logical AND
CMC                     ;Complement Carry
CMP                     ;Compare
CMPS                    ;Compare memory at SI and DI
DAA                     ;Decimal Adjust for Add in AX
DAS                     ;Decimal Adjust for Subtract in AX
DEC                     ;Decrement
DIV                     ;Divide (unsigned) in AX(,DX)
IDIV                    ;Divide (signed) in AX(,DX)
MUL                     ;Multiply (unsigned) in AX(,DX)
IMUL                    ;Multiply (signed) in AX(,DX)
INC                     ;Increment
```

x86 Instruction Set Summary
(Arithmetic/Logical Cont.)

```
NEG                 ;Negate
NOT                 ;Logical NOT
OR                  ;Logical inclusive OR
RCL                 ;Rotate through Carry Left
RCR                 ;Rotate through Carry Right
ROL                 ;Rotate Left
ROR                 ;Rotate Right
SAR                 ;Shift Arithmetic Right
SBB                 ;Subtract with Borrow
SCAS                ;Scan memory at DI compared to AX
SHL/SAL             ;Shift logical/Arithmetic Left
SHR                 ;Shift logical Right
SUB                 ;Subtract
TEST                ;AND function to flags
XLAT                ;Translate byte to AL
XOR                 ;Logical Exclusive OR
```

x86 Instruction Set Summary
(Control/Branch Cont.)

```
CALL                    ;Call
CLC                     ;Clear Carry
CLD                     ;Clear Direction
CLI                     ;Clear Interrupt
ESC                     ;Escape (to external device)
HLT                     ;Halt
INT                     ;Interrupt
INTO                    ;Interrupt on Overflow
IRET                    ;Interrupt Return
JB/JNAE                 ;Jump on Below/Not Above or Equal
JBE/JNA                 ;Jump on Below or Equal/Not Above
JCXZ                    ;Jump on CX Zero
JE/JZ                   ;Jump on Equal/Zero
JL/JNGE                 ;Jump on Less/Not Greater or Equal
JLE/JNG                 ;Jump on Less or Equal/Not Greater
JMP                     ;Unconditional Jump
JNB/JAE                 ;Jump on Not Below/Above or Equal
JNBE/JA                 ;Jump on Not Below or Equal/Above
JNE/JNZ                 ;Jump on Not Equal/Not Zero
JNL/JGE                 ;Jump on Not Less/Greater or Equal
```

x86 Instruction Set Summary

(Control/Branch)

```
JNLE/JG                 ;Jump on Not Less or Equal/Greater
JNO                     ;Jump on Not Overflow
JNP/JPO                 ;Jump on Not Parity/Parity Odd
JNS                     ;Jump on Not Sign
JO                      ;Jump on Overflow
JP/JPE                  ;Jump on Parity/Parity Even
JS                      ;Jump on Sign
LOCK                    ;Bus Lock prefix
LOOP                    ;Loop CX times
LOOPNZ/LOOPNE           ;Loop while Not Zero/Not Equal
LOOPZ/LOOPE             ;Loop while Zero/Equal
NOP                     ;No Operation (= XCHG AX,AX)
REP/REPNE/REPNZ         ;Repeat/Repeat Not Equal/Not Zero
REPE/REPZ               ;Repeat Equal/Zero
RET                     ;Return from call
SEG                     ;Segment register
STC                     ;Set Carry
STD                     ;Set Direction
STI                     ;Set Interrupt
TEST                    ;AND function to flags
WAIT                    ;Wait
```

Assembler Directives

end *label*	end of program, label is entry point
proc far|near	begin a procedure; far, near keywords specify if procedure in different code segment (far), or same code segment (near)
endp	end of procedure
page	set a page format for the listing file
title	title of the listing file
.code	mark start of code segment
.data	mark start of data segment
.stack	set size of stack segment

Assembler Directives

db	define byte
dw	define word (2 bytes)
dd	define double word (4 bytes)
dq	define quadword (8 bytes)
dt	define tenbytes
equ	equate, assign numeric expression to a name

Examples:

db 100 dup (?)	define 100 bytes, with no initial values for bytes
db “Hello”	define 5 bytes, ASCII equivalent of “Hello”.
maxint equ	32767
count equ	10 * 20 ; calculate a value (200)

Program Example

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                    ;
;   This is an example program.  It prints the                       ;
;   character string "Hello World" to the DOS standard output        ;
;   using the DOS service interrupt, function 9.                      ;
;                                                                    ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
hellostk  SEGMENT BYTE STACK 'STACK'    ;Define the stack segment
          DB 100h DUP(?)    ;Set maximum stack size to 256 bytes (100h)
hellostk  ENDS

hellodat  SEGMENT BYTE 'DATA' ;Define the data segment
dos_print EQU 9                         ;define a constant via EQU
strng     DB  'Hello World',13,10,'$'   ;Define the character string
hellodat  ENDS

hellocod  SEGMENT BYTE 'CODE' ;Define the Code segment
START:    mov ax, SEG hellodat         ;ax <-- data segment start address
          mov ds, ax                   ;ds <-- initialize data segment register
          mov ah, dos_print            ;ah <-- 9 DOS 21h string function
          mov dx,OFFSET strng          ;dx <-- beginning of string
          int 21h                      ;DOS service interrupt
          mov ax, 4c00h                ;ax <-- 4c DOS 21h program halt function
          int 21h                      ;DOS service interrupt
hellocod  ENDS
          END       START              ; ‘END label’ defines program entry
```

Another Way to define Segments

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Use 'assume' directive to define segment types                  ;
;                                                                   ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
hellostk  SEGMENT         ;Define a segment
          DB 100h DUP(?)
hellostk  ENDS

hellodat  SEGMENT           ;define a segment
dos_print EQU 9                        ;define a constant
strng     DB  'Hello World',13,10,'$'  ;Define the character string
hellodat  ENDS

hellocod  SEGMENT           ;define a segment
          assume cs:hellocod, ds:hellodat, ss: hellostk
START:    mov ax, hellodat            ;ax <-- data segment start address
          mov ds, ax                  ;ds <-- initialize data segment register
          mov ah, dos_print           ;ah <-- 9 DOS 21h string function
          mov dx,OFFSET strng         ;dx <-- beginning of string
          int 21h                     ;DOS service interrupt
          mov ax, 4c00h               ;ax <-- 4c DOS 21h program halt function
          int 21h                     ;DOS service interrupt
hellocod  ENDS
END       START
```

Yet another way to define Segs

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;   Use .stack,.data,.code directives to define segment types       ;
;                                                                   ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
          .stack 100h        ; reserve 256 bytes of stack space

          .data
dos_print EQU 9                         ;define a constant
strng     DB  'Hello World',13,10,'$'   ;Define the character string


          .code

START:    mov ax, SEG strng            ;ax <-- data segment start address
          mov ds, ax                   ;ds <-- initialize data segment register
          mov ah, dos_print            ;ah <-- 9 DOS 21h string function
          mov dx,OFFSET strng          ;dx <-- beginning of string
          int 21h                      ;DOS service interrupt
          mov ax, 4c00h                ;ax <-- 4c DOS 21h program halt function
          int 21h                      ;DOS service interrupt

END       START
```

Program Statements

`name operation operand(s) comment`

- Operation is a predefined or reserved word
  - mnemonic - symbolic operation code
  - directive - pseudo-operation code
- Space or tab separates initial fields
- Comments begin with semicolon
- Most assemblers are not case sensitive

Program Data and Storage

- Pseudo-ops to define data or reserve storage
  - DB - byte(s)
  - DW - word(s)
  - DD - doubleword(s)
  - DQ - quadword(s)
  - DT - tenbyte(s)
- These directives require one or more operands
  - define memory contents
  - specify amount of storage to reserve for run-time data

Defining Data

- Numeric data values
  - 100 - decimal
  - 100B - binary
  - 100H - hexadecimal
  - '100' - ASCII
  - "100" - ASCII
- Use the appropriate DEFINE directive (byte, word, etc.)
- A list of values may be used - the following creates 4 consecutive words

```
DW 40CH,10B,-13,0
```

- A ? represents an uninitialized storage location

```
DB 255,?,-128,'X'
```

Naming Storage Locations

- Names can be associated with storage locations

```
ANum DB -4
 DW 17
ONE
UNO DW 1
X DD ?
```

- These names are called variables
- ANum refers to a byte storage location, initialized to FCh
- The next word has no associated name
- ONE and UNO refer to the same word
- X is an unitialized doubleword

Arrays

- Any consecutive storage locations of the same size can be called an array

```
X DW 40CH,10B,-13,0
Y DB 'This is an array'
Z DD -109236, FFFFFFFFH, -1, 100B
```

- Components of X are at X, X+2, X+4, X+8
- Components of Y are at Y, Y+1, …, Y+15
- Components of Z are at Z, Z+4, Z+8, Z+12

DUP

- Allows a sequence of storage locations to be defined or reserved
- Only used as an operand of a define directive

```
DB 40 DUP (?)
DW 10h DUP (0)
DB 3 dup ("ABC")
```

Word Storage

- Word, doubleword, and quadword data are stored in reverse byte order (in memory)

```
Directive      Bytes in Storage
DW 256         00 01
DD 1234567H    67 45 23 01
DQ 10          0A 00 00 00 00 00 00 00
X DW 35DAh     DA 35
```

Low byte of X is at X, high byte of X is at X+1

Named Constants

- Symbolic names associated with storage locations represent addresses
- Named constants are symbols created to represent specific values determined by an expression
- Named constants can be numeric or string
- Some named constants can be redefined
- No storage is allocated for these values

Equal Sign Directive

- name = expression
  - expression must be numeric
  - these symbols may be redefined at any time

```
maxint = 7FFFh
count = 1
DW count
count = count * 2
DW count
```

EQU Directive

- name EQU expression
  - expression can be string or numeric
  - Use < and > to specify a string EQU
  - these symbols cannot be redefined later in the program

```
sample EQU 7Fh
aString EQU <1.234>
message EQU <This is a message>
```

Data Transfer Instructions

- MOV *target, source*
  - reg, reg
  - mem, reg
  - reg, mem
  - mem, immed
  - reg, immed
- Sizes of both operands must be the same
- reg can be any non-segment register except IP cannot be the target register
- MOV's between a segment register and memory or a 16-bit register are possible

Sample MOV Instructions

```
b db 4Fh
w dw 2048

mov bl,dh
mov ax,w
mov ch,b
mov al,255
mov w,-100
mov b,0
```

- When a variable is created with a define directive, it is assigned a default size attribute (byte, word, etc)
- You can assign a size attribute using LABEL

```
LoByte LABEL BYTE
aWord DW 97F2h
```

Addresses with Displacements

```
b db 4Fh, 20h, 3Ch
w dw 2048, -100, 0

mov bx, w+2
mov b+1, ah
mov ah, b+5
mov dx, w-3
```

- Type checking is still in effect

- The assembler computes an address based on the expression
- *NOTE: These are address computations done at assembly time*

  MOV ax, b-1

  will not subtract 1 from the value stored at b

eXCHanGe

- XCHG *target, source*
  - reg, reg
  - reg, mem
  - mem, reg
- MOV and XCHG cannot perform memory to memory moves
- This provides an efficient means to swap the operands
  - No temporary storage is needed
  - Sorting often requires this type of operation
  - This works only with the general registers

Arithmetic Instructions

ADD *dest, source*

SUB *dest, source*

INC *dest*

DEC *dest*

NEG *dest*

- *Operands must be of the same size*
- *source* can be a general register, memory location, or constant
- *dest* can be a register or memory location
  - except operands cannot both be memory

Program Segment Structure

- Data Segments
  - Storage for variables
  - Variable addresses are computed as offsets from start of this segment
- Code Segment
  - contains executable instructions
- Stack Segment
  - used to set aside storage for the stack
  - Stack addresses are computed as offsets into this segment
- Segment directives

  `.data`

  `.code`

  `.stack` ***`size`***

Memory Models

- .Model *memory_model*
  - tiny: code+data <= 64K (.com program)
  - small: code<=64K, data<=64K, one of each
  - medium: data<=64K, one data segment
  - compact: code<=64K, one code segment
  - large: multiple code and data segments
  - huge: allows individual arrays to exceed 64K
  - flat: no segments, 32-bit addresses, protected mode only (80386 and higher)

Program Skeleton

```
.model small
.stack 100H
.data
   ;declarations
.code
main proc
   ;code
main endp
   ;other procs
end main
```

- Select a memory model
- Define the stack size
- Declare variables
- Write code
  - organize into procedures
- Mark the end of the source file
  - optionally, define the entry point