EE-2003
Computer Organization & Assembly Language

INSTRUCTOR

Engr. Aashir Mahboob
Lecturer, Department of Computer Science
FAST NUCES (Karachi)
aashir.mahboob@nu.edu.pk

Chapter No: 03

ASSEMBLY LANGUAGE FUNDAMENTALS

INSTRUCTIONS

- An instruction is a statement that becomes executable when a program is assembled.

- Instructions are translated by the assembler into machine language bytes, which are loaded and executed by the CPU at runtime.

- An instruction contains four basic parts:

- Label (optional)
- Instruction mnemonic (required)
- Operand(s) (usually required)
- Comment (optional)

INSTRUCTIONS

- This is how the different parts are arranged:

- [label:] mnemonic [operands] [;comment]

- A **label** is an identifier that acts as a place marker for instructions and data.
- It implies the address of instruction or variable.
- A **data label** identifies the location of a variable, providing a convenient way to reference the variable in code.
- Example
  - count DWORD 100
  - array DWORD 1024, 2048

INSTRUCTIONS

- A label in the code area of a program (where instructions are located) must end with a colon (:) character.
- . Code labels are used as targets of jumping and looping instructions.
- Example

```
target:
    mov   ax,bx
    ...
    jmp   target
```

- A code label can share the same line with an instruction, or it can be on a line by itself:

```
L1: mov   ax,bx
L2:
```

Instruction Mnemonic

- An instruction mnemonic is a short word that identifies an instruction.
- It provide hints about the type of operation they perform.

Mnemonic	Description
MOV	Move (assign) one value to another
ADD	Add two values
SUB	Subtract one value from another
MUL	Multiply two values
JMP	Jump to a new location
CALL	Call a procedure

Operands

- An operand is a value that is used for input or output for an instruction.
- Assembly language instructions can have between zero and three operands.
- Each of which can be a register, memory operand, integer expression, or input–output port.

Example	Operand Type
96	*Integer literal*
2 + 4	Integer expression
eax	Register
count	Memory

Instruction Format Examples

- No operands
  - stc ; set Carry flag
- One operand
  - inc eax ; register
  - inc myByte ; memory
- Two operands
  - add ebx,ecx ; register, register
  - sub myByte,25 ; memory, constant
  - add eax,36 * 25 ; register, constant-expression

Instruction Format Examples

- Three operands
  - imul eax, ebx, 5

- There is a natural ordering of operands.
- When instructions have multiple operands, the first one is typically called the destination operand.
- The second operand is usually called the source operand.
- In general, the contents of the destination operand are modified by the instruction.

The NOP (No Operation) Instruction

- The safest (and the most useless) instruction is NOP (no operation).
- It takes up 1 byte of program storage and doesn't do any work.
- It is sometimes used by compilers and assemblers to align code to efficient address boundaries.
- In the following example, the first MOV instruction generates three machine code bytes.
- The NOP instruction aligns the address of the third instruction to a doubleword boundary (even multiple of 4):

```
00000000 66 8B C3       mov ax,bx
00000003 90             nop                 ; align next
instruction
00000004 8B D1          mov edx,ecx
```

- x86 processors are de ord addresses.

Integer Constants

- An integer literal (also known as an integer constant) is made up of an optional leading sign, one or more digits, and an optional radix character that indicates the number's base:

h	hexadecimal	r	encoded real
q/o	octal	t	decimal (alternate)
d	decimal	y	binary (alternate)
b	binary		

- . If Constant doesn't have a radix, so we assume it's in decimal format.

Integer Constants

- Here are some integer literals declared with various radixes. Each line contains a comment:

```
26                      ; decimal
26d                     ; decimal
11010011b               ; binary
42q                     ; octal
42o                     ; octal
1Ah                     ; hexadecimal
0A3h                    ; hexadecimal
```

- A hexadecimal literal beginning with a letter must have a leading zero to prevent the assembler from interpreting it as an identifier.
- Such is the case with the hexadecimal value A3h in the foregoing list, which must be written as 0A3h.

CLASS ACTIVITY

- 1. MOV AL, 255
- 2. MOV BL, 10
- 3. MOV CL, 20
- 4. ADD AL, 1
- 5. SUB BL,CL
- 6. SUB AL,1
- 7. INC BL
- 8. INC CL
- 9. SUB BL,CL

Write down Contents Of AL,BL & CL After execution of each instruction

Integer Expressions

- An integer expression is a mathematical expression involving integer values and arithmetic operators
- The integer expression must evaluate to an integer,

Operator	Name	Precedence Level
( )	Parentheses	1
+, –	Unary plus, minus	2
*, /	Multiply, divide	3
MOD	Modulus	3
+, –	Add, subtract	4

- Precedence refers to the implied order of operations when an expression contains two or more operators

```
4 + 5 * 2                    Multiply, add
12 - 1 MOD 5                 Modulus, subtract
-5 + 2                       Unary minus, add
(4 + 2) * 6                  Add, multiply
```

Integer Expressions

- Example:

Expression	Value
16 / 5	3
−(3 + 4) * (6 − 1)	−35
−3 + 4 * 6 − 1	20
25 mod 3	1

DEFINING DATA

- The assembler recognizes a basic set of intrinsic data types, which describe types in terms of their size.

[name] directive initializer [, initializer]...

- Initializer: At least one initializer is required in a data definition, even if it is zero.

```
.data
sum DWORD 0
```

- If you prefer to leave the variable uninitialized (assigned a random value), the ? symbol can be used as the initializer

```
sum DWORD ?
```

DEFINING DATA

Type	Usage
BYTE	8-bit unsigned integer. B stands for byte
SBYTE	8-bit signed integer. S stands for signed
WORD	16-bit unsigned integer
SWORD	16-bit signed integer
DWORD	32-bit unsigned integer. D stands for double
SDWORD	32-bit signed integer. SD stands for signed double
FWORD	48-bit integer (Far pointer in protected mode)
QWORD	64-bit integer. Q stands for quad
TBYTE	80-bit (10-byte) integer. T stands for Ten-byte
REAL4	32-bit (4-byte) IEEE short real
REAL8	64-bit (8-byte) IEEE long real
REAL10	80-bit (10-byte) IEEE extended real

DEFINING DATA

- Defining BYTE and SBYTE Data
  - Each initializer must fit into 8 bits of storage

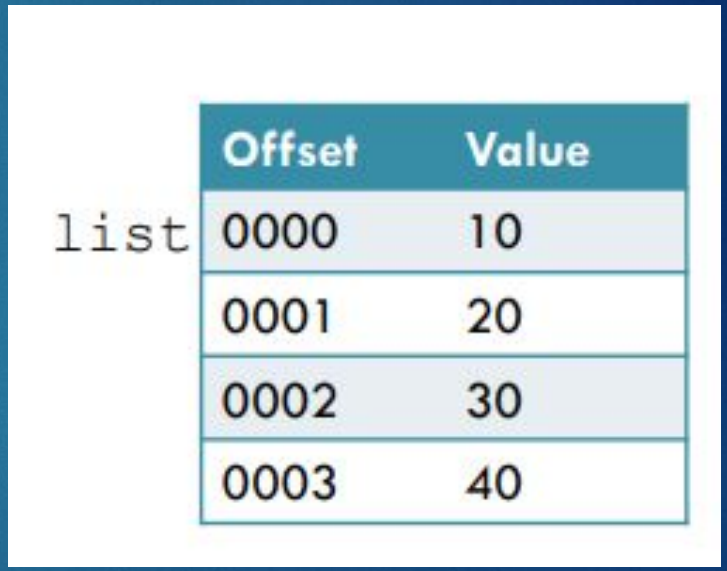
```
value1 BYTE 'A'          ; character constant
value2 BYTE 0            ; smallest unsigned byte
value3 BYTE 255          ; largest unsigned byte
value4 SBYTE -128        ; smallest signed byte
value5 SBYTE +127        ; largest signed byte
value6 BYTE ?            ; uninitialized byte
```

DEFINING DATA

- **Multiple Initializers**

•If multiple initializers are used in the same data definition, its label refers only to the offset of the first initializer.

list BYTE 10,20,30,40



	Offset	Value
list	0000	10
	0001	20
	0002	30
	0003	40

Within a single data definition, its initializers can use different radixes. Character and string literals can be freely mixed. In the following example, list1 and list2 have the same contents:

list1 BYTE 10, 32, 41h, 00100010b
list2 BYTE 0Ah, 20h, 'A', 22h

Examples that use multiple initializers:

- list1 BYTE 10,20,30,40
- list2 BYTE 10,20,30,40
  BYTE 50,60,70,80
  BYTE 81,82,83,84
  list3 BYTE ?,32,41h,00100010b
  list4 BYTE 0Ah,20h,‘A’,22h

	Offset	Value
list1	0000	10
	0001	20
	0002	30
	0003	40
list2	0004	10
	0005	20
	0006	30
	0007	40
	0008	50
	0009	60
	000A	70
	000B	80
	000C	81
	000D	82
	000E	83
	000F	84
list3	0010	

Defining Strings:

- The most common type of string ends with a null byte (containing 0), called a null-terminated string.

greeting1 BYTE "Good afternoon",0

greeting2 BYTE 'Good night',0

•Each character uses a byte of storage.

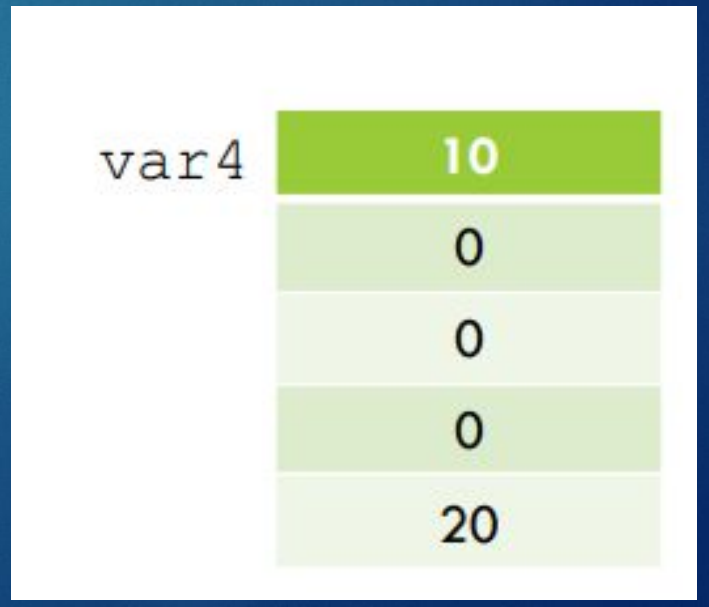
•The rule that byte values must be separated by commas does not apply on strings.

DUP OPERATOR

- The DUP operator allocates storage for multiple data items, using a integer expression as a counter.

```
BYTE 20 DUP(0)                ;20 bytes, all equal to zero
BYTE 20 DUP(?)                ;20 bytes, uninitialized
BYTE 4 DUP("STACK")           ;20 bytes: "STACKSTACKSTACKSTACK"
```

var4 BYTE 10,3 DUP(0), 20



EXERCISE

- Define

  i. Largest unsigned Value (16-bits)

  ii. Smallest Signed Value (16-bits)

  iii. Initialized Array of 5 Words.

  iv. Un-initialized Array of 5 Words.

```
word1   WORD   65535
word2   SWORD  -32768
```

```
myList  WORD   1,2,3,4,5
array   WORD   5 DUP(?)
```

- Define

  i. Largest unsigned Value (32-bits)

  ii. Smallest Signed Value (32-bits)

  iii. Unsigned Array. (32- bits)

  iv. Signed Array . (32-bits)

```
val1  DWORD   12345678h
val2  SDWORD  -2147483648
```

```
val3  DWORD   20 DUP(?)
val4  SDWORD  -3,-2,-1,0,1
```

Real Number

- Real number constants are represented as decimal reals or encoded (hexadecimal) reals.
- A decimal real contains are optional sign followed by an integer, a decimal point an optional integer that expresses a fraction, and an optional exponent:

[sign] integer. [integer] [exponent]

- Following are the syntax for the sign and exponent:

sign	**{ + , - }**
exponent	**E [{ + , - }] integer**

- Following are examples of valid number constants:

2.

+3.0

-44.2E+05

26.E5

- At least one digit and a decimal point are required.

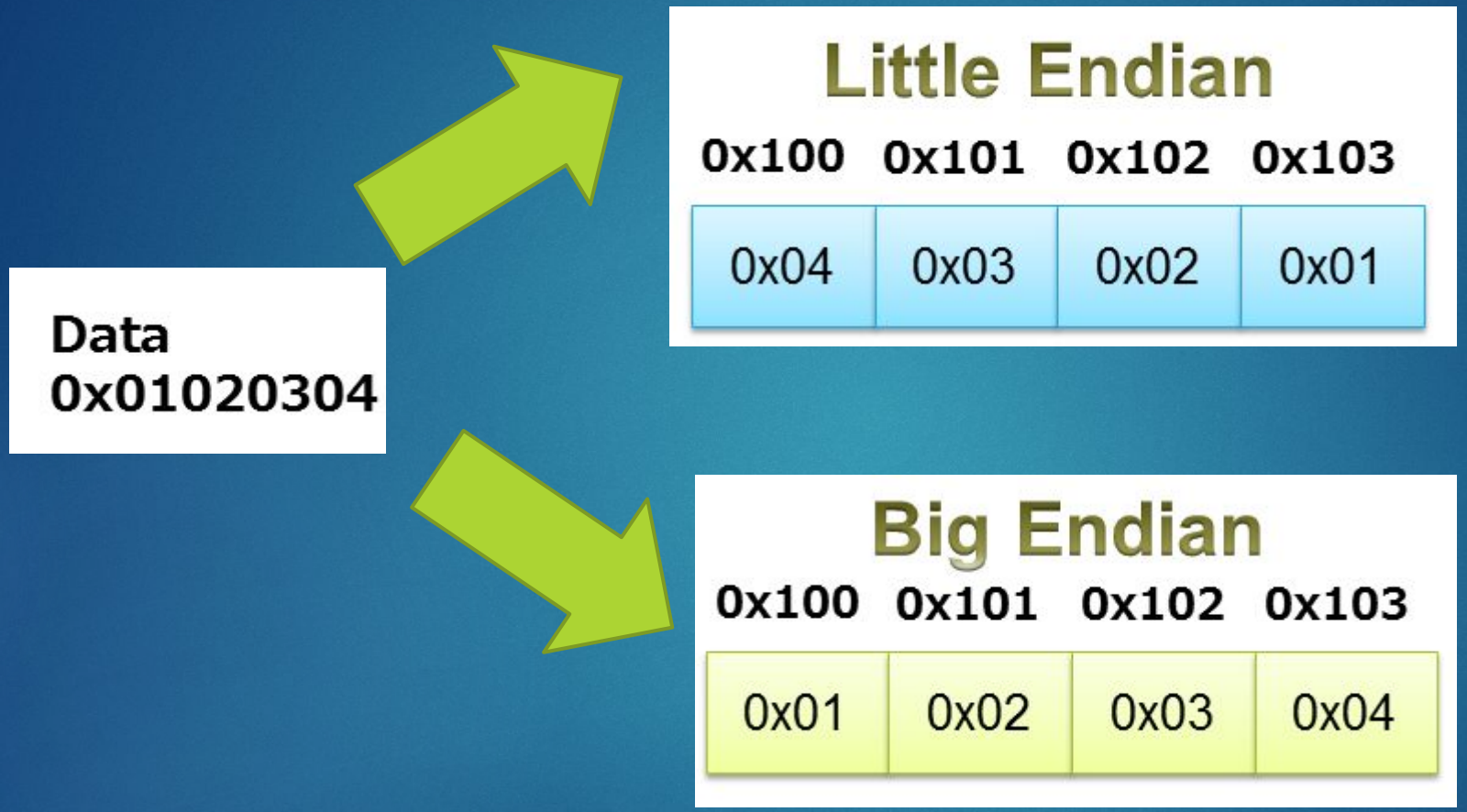
Real Number

```
rVal1       REAL4 -1.2
rVal2       REAL8  3.2E-260
rVal3       REAL10 4.6E+4096
ShortArray  REAL4  20 DUP(0.0)
```

REAL4 defines a 4-byte single-precision floating-point variable. REAL8 defines an 8-byte double-precision value, and REAL10 defines a 10-byte extended-precision value. Each requires one or more real constant initializers:

The MASM assembler includes data types such as REAL4 and REAL8, suggesting the values they represent are real numbers. More correctly, the values are floating-point numbers, which have a limited amount of precision and range.

Little and Big Endian



Directive

- •A directive is a command embedded in the source code that is recognized and acted upon by the assembler.
- • .data
- • .code
- • .stack
- • DWORD

DIRECTIVE VS INSTRUCTION

- **myVar DWORD 26**
- DWORD directive tells the assembler to reserve space in the program for a doubleword variable.

- **mov eax,myVar**
- The MOV instruction, on the other hand, executes at runtime, copying the contents of myVar to the EAX register.

EXAMPLE CODE

```
 1: ; AddTwo.asm - adds two 32-bit integers
 2: ; Chapter 3 example
 3:
 4: .386
 5: .model flat,stdcall
 6: .stack 4096
 7: ExitProcess PROTO, dwExitCode:DWORD
 8:
 9: .code
10: main PROC
11: mov eax,5 ; move 5 to the eax register
12: add eax,6 ; add 6 to the eax register
13:
14: INVOKE ExitProcess,0
15: main ENDP
16: END main
```

EXAMPLE CODE (EXPLANATION)

- .386 directive identifies the program as a 32-bit program that can access 32-bit registers and addresses.
- .model flat, stdcall selects the programs memory model, and identifies the calling convention.
- The stdcall keyword tells the assembler how to manage the runtime stack when procedures are called.
- It is a calling convention, that is a scheme for how subroutines receive parameters from their caller and how they return a result.

EXAMPLE CODE (EXPLANATION)

- .stack 4096 sets aside 4096 bytes of storage for the runtime stack.

- It tells the assembler how many bytes of memory to reserve for the program's runtime stack.
- Stack are used :
- to hold passed parameters
- to hold the address of the code that called the function. The CPU uses this address to return when the function call finishes, back to the spot where the function was called.
- Stack can hold local variables.

EXAMPLE CODE (EXPLANATION)

- When your program is ready to finish, it calls ExitProcess that returns an integer that tells the operating system that your program worked just fine.

- The ENDP directive marks the end of a procedure.
- Our program had a procedure named main, so the endp must use the same name.

- Line 16 uses the END directive to mark the last line to be assembled (end of program), and it identifies the program entry point (main).

- If you add any more lines to a program after the END directive, they will be ignored by the assembler.

REVIEW QUESTIONS

- What is the Purpose of **TITLE** Directive?
- It marks entire line as comment.
- Example TITLE Add and Subtract (AddSub.asm)

- What is the Purpose of **INCLUDE** Directive?
- Copies necessary definitions and setup information from text file.
- Example **INCLUDE irvine32.inc**