3

Assembly Language Fundamentals

OUTLINES

- Basic Language Elements
- Assembling, Linking, and Running Programs
- Defining Data
- Symbolic Constants

3.1 BASIC LANGUAGE ELEMENTS

```
main PROC

 mov eax, 5          ; move 5 to the EAX register
 add eax, 6          ; add 6 to the EAX register
 INVOKE ExitProcess,0 ; end the program

main ENDP
```

***AddTwo* program**

```
.data
  sum DWORD 0

.code
  main PROC
   mov eax,5
   add eax,6
   mov sum,eax
   INVOKE ExitProcess,0
  main ENDP
```

Integer Literals

- 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:

```
[{+ | - }] digits [ radix ]
```

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

Table 3-1 Arithmetic Operators.

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

- **Real Number Literals**

```
[sign] integer.[integer][exponent]
```

Examples

↑2., +3.0, -44.2E+05, 26.E5

- A **character literal** is a single character enclosed in single or double quotes. The assembler stores the value in memory as the character's binary ASCII code.
  - E.g. 'A', "x".

- A **string literal** is a sequence of characters (including spaces) enclosed in single or double quotes.
  - E.g. 'ABC', 'X', "Good night, Gracie"

- **Reserved words** have special meaning and can only be used in their correct context (not case sensitive).
  - Instruction mnemonics, Register names, Directives
  - Attributes, which provide size and usage information for variables and operands. Examples are BYTE and WORD
  - Operators, used in constant expressions
  - Predefined symbols, such as @data, which return constant integer values at assembly time.

- An **identifier** is a programmer-chosen name. It might identify a variable, a constant, a procedure, or a code label.
- Rules:
  - They may contain between 1 and 247 characters.
  - They are not case sensitive.
  - The first character must be a letter (A..Z, a..z), underscore (_), @ , ?, or $. Subsequent characters may also be digits.
  - An identifier cannot be the same as an assembler reserved word.

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

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

- **Labels** act as place markers
  - marks the address (offset) of code and data

Data label

- `count DWORD 100`

Code label

- target of `jump` and `loop` instructions

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

- Instruction Mnemonics:

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: a value that is used for input or output for an instruction:

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

- 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

3.2 EXAMPLE

```
 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
```

- `.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.

- `.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 :

1. to hold passed parameters
2. 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.
3. Stack can hold local variables.

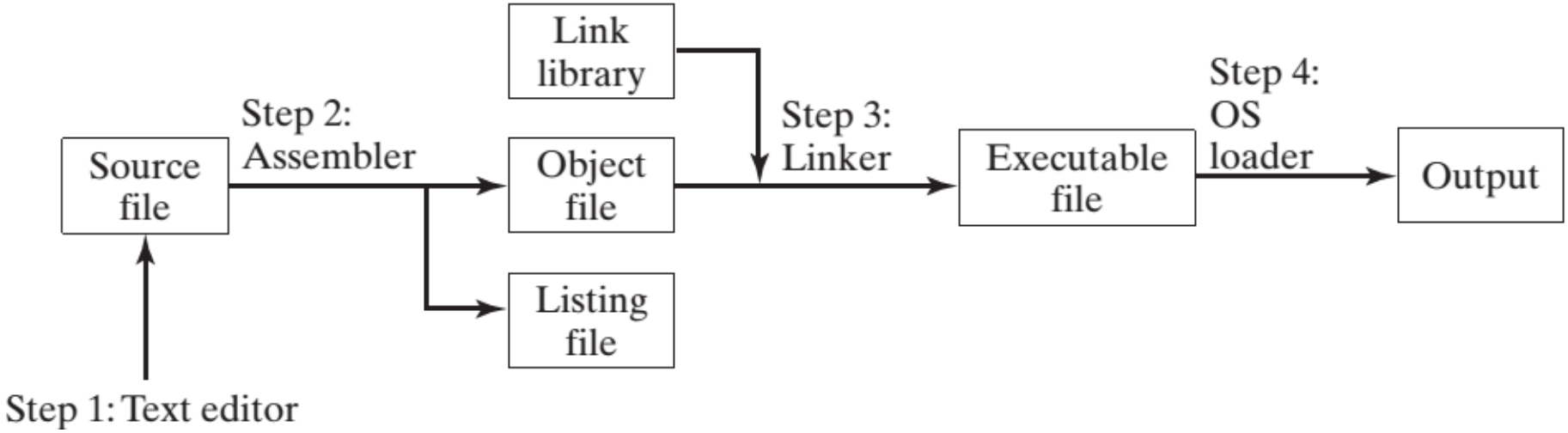
- `.code` marks the beginning of the code area of a program.

- 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.

ASSEMBLING, LINKING, AND RUNNING PROGRAMS



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 ?
```

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

Table 3-3 Legacy Data Directives.

Directive	Usage
DB	8-bit integer
DW	16-bit integer
DD	32-bit integer or real
DQ	64-bit integer or real
DT	define 80-bit (10-byte) integer

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
```

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
```

list

Offset	Value
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.

- 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
```

var4	10
	0
	0
	0
	20

- **Defining WORD and SWORD Data**
- The WORD (define word) and SWORD (define signed word) directives create storage for one or more 16-bit integers:

```
word1   WORD   65535              ; largest unsigned value
word2   SWORD -32768              ; smallest signed value
word3   WORD   ?                  ; uninitialized, unsigned
word4   WORD   "AB"               ; double characters
myList  WORD   1,2,3,4,5          ; array of words
array   WORD   5 DUP(?)           ; uninitialized array
```

- The legacy DW directive can also be used:

```
val1 DW 65535 ; unsigned
val2 DW -32768 ; signed
```

```
myList WORD 1,2,3,4,5
```

Offset	Value
0000:	1
0002:	2
0004:	3
0006:	4
0008:	5

Defining DWORD and SDWORD Data

- Storage definitions for signed and unsigned 32-bit integers:

```
val1 DWORD  12345678h      ; unsigned
val2 SDWORD -2147483648    ; signed
val3 DWORD  20 DUP(?)      ; unsigned array
val4 SDWORD -3,-2,-1,0,1   ; signed array
```

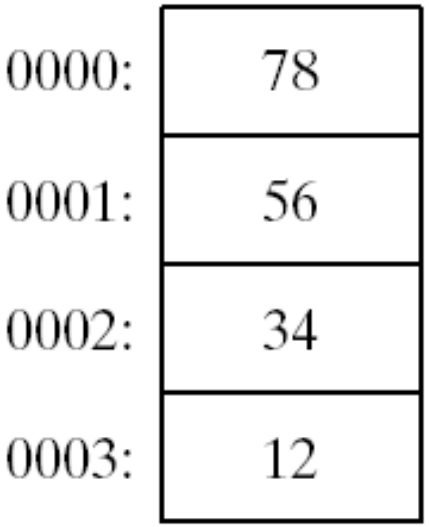
```
myList DWORD 1,2,3,4,5
```

Offset	Value
0000:	1
0004:	2
0008:	3
000C:	4
0010:	5

LITTLE ENDIAN ORDER

- All data types larger than a byte store their individual bytes in reverse order. The least significant byte occurs at the first (lowest) memory address.

```
val1 DWORD 12345678h
```



DECLARING UNITIALIZED DATA

Declaring Uninitialized Data

- Use the `.data?` directive to declare an uninitialized data segment:

```
.data?
```

- When defining a large block of uninitialized data, the `.DATA?` directive reduces the size of a compiled program
- Within the segment, declare variables with "?" initializers:

```
smallArray DWORD 10 DUP(?)
```

Advantage: the program's EXE file size is reduced.

```
.data
smallArray DWORD 10 DUP(0) ; 40 bytes
.data?
bigArray DWORD 5000 DUP(?) ; 20,000 bytes, not initialized
```

The following code, on the other hand, produces a compiled program 20,000 bytes larger:

```
.data
smallArray DWORD 10 DUP(0) ; 40 bytes
bigArray DWORD 5000 DUP(?) ; 20,000 bytes
```

3.5 SYMBOLIC CONSTANTS

- A *symbolic constant* (or *symbol definition*) is created by associating an identifier (a symbol) with an integer expression or some text.
- Symbols do not reserve storage.

	Symbol	Variable
Uses storage?	No	Yes
Value changes at runtime?	No	Yes

- **Equal-Sign Directive**

```
name = expression
```

- ↑ expression is a 32-bit integer (expression or constant)
- ↑ may be redefined
- ↑ *name* is called a *symbolic constant*

- *Current Location Counter :* the symbol **$**, is the current location counter, that returns offset of current location:

```
selfPtr DWORD $
```

  - This statement declares a variable named **selfPtr** and initializes it with the variable's offset value.

- **Calculating the Sizes of Arrays and Strings**

- As $ operator returns the offset associated with the current program statement, it can be used to calculate the size of array/string:

```
list BYTE 10,20,30,40
ListSize = ($ - list)
```

ListSize must follow immediately after **list.**

SUMMARY

- Integer expression, character constant
- Directive – interpreted by the assembler; Instruction – executes at runtime
- Code, Data, and Stack Segments
- Source, Object, Executable files
- Data definition directives:
  - BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, QWORD, TBYTE
- DUP operator, location counter ($)
- Symbolic constant