

SECR2033

Computer Organization and Architecture

Module 3 & Lab 1

Module 3

Introduction to Assembly Language Programming

Objectives:

- ❑ To understand the interaction between computer hardware, operating systems, and application programs.
- ❑ To apply and test theoretical information given in computer architecture and operating systems courses.
- ❑ To write the basic of assembly language and execute the program.

Module 3

Introduction to Assembly Language Programming

3.1 Introduction

3.2 Basic Elements of Assembly Language

3.3 Example: Adding and Subtracting Integers

3.4 Defining Data

3.5 Symbolic Constants

3.6 Summary

3.1 Introduction

3

- *Assembly language* is the oldest programming language, and of all languages, bears the closest resemblance to native *machine language*.
 - *Machine language* is a numeric language specifically understood by a computer's processor (the CPU).
 - *Assembly language* provides direct access to computer hardware, requiring us to understand much about your *computer's architecture* and *operating system*.
- Each *assembly language instruction* corresponds to a single *machine-language instruction*.

Module 3

Introduction to Assembly Language Programming

3.1 Introduction

3.2 Basic Elements of Assembly Language

3.3 Example: Adding and Subtracting Integers

3.4 Defining Data

3.5 Symbolic Constants

3.6 Summary

- ❑ Integer Constants
- ❑ Real Number Constants
- ❑ Character & String Constants
- ❑ Reserved Words
- ❑ Identifiers
- ❑ Directives
- ❑ Instructions
- ❑ Operands
- ❑ Comments
- ❑ Registers

3.2 Basic Elements of Assembly Language

3

- Generally, assembly language is not hard to learn if you're happy writing short programs that do practically nothing.
- Describing the basic elements will help us to write our first programs in assembly language.



(1) Integer Constants

- An **integer constant** (or integer literal) is made up of an optional leading **sign**, one or more **digits**, and an optional suffix character (called a **radix**) indicating the number's base:

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

- **Radix** may be one of the following (uppercase or lowercase):

h	Hexadecimal	r	Encoded real
q/o	Octal	t	Decimal (<i>alternate</i>)
d	Decimal	y	Binary (<i>alternate</i>)
b	Binary		

- If no radix is given, the integer constant is assumed to be decimal.

- Examples:

26

26d

11010011b

42q

42o

1Ah

0A3h

*Decimal
since no radix*

A **hexadecimal** constant beginning with a letter must leading with **zero** to prevent the assembler from interpreting it as an identifier.

(2) Real Number Constants

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

[sign] integer . [integer] [exponent]

- The syntax for the sign and exponent:

sign { + , - }

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

- **Examples**: Valid real number constants:

2 .

+3 . 0

-44 . 2E+05

26 . E5

(3) Character & String Constants

- A **character constant** is a single character enclosed in single or double quotes.

- Examples:

```
'A'  
"d"
```

- ASCII character = 1 byte.

- A **string constant** is a sequence of characters (including spaces) enclosed in single or double quotes.

- Examples:

```
'ABC '  
'X '  
"Good night, Gracie"  
'4096 '
```

- Each character occupies a single byte.

- **Embedded quotes** are permitted when used in the manner shown by the following examples:

```
"This isn't a test"  
'Say "Good night," Gracie'
```

```
' Say "COA is easy" '
```



(4) Reserved Words

■ Reserved words cannot be used as identifiers:

- Instruction mnemonics (`MOVE`, `ADD`, `MUL`)
- Directives (`INCLUDE`)
- Type attributes (`BYTE`, `WORD`)
- Operators (+, -)
- Predefined symbols (`@data`)

(5) Identifiers

- An **identifier** is a programmer-chosen name.
- It might identify a *variable*, a *constant*, a *procedure*, or a *code label*.
 - May contain 1– 247 characters, including digits.
 - Not case-sensitive (by default).
 - First character must be a letter (A . . Z, a . . z), `_`, `@`, `?` or `$`
 - Subsequent character may also be a digit.
 - Identifier cannot be the same as assembler reserved word.

- The @ symbol is used extensively by the assembler as a prefix for **predefined symbols**, so avoid using as identifiers.
- Make identifier names descriptive and easy to understand.
- **Examples of some valid identifiers:**

`var1``Count``$first``_main``MAX``open_file``myFile``xVal``_12345`

(6) Directives

- A **directive** is a command embedded in the source code that is recognized and acted upon by the assembler:

- Not execute at runtime.
- Can define variables, macros, and procedures.
- Not case sensitive (it recognizes “.data”, “.DATA”, and “.Data” as equivalent)
- Different assemblers have different directives.

- Example to show the difference between **directives** and **instructions**:

Tells the assembler to reserve space in the program for a doubleword variable.

```
myVar DWORD 26      ;DWORD directive  
mov  eax,myVar      ;MOV instruction
```

*Executes at runtime,
copying the contents of
myVar to the EAX register.*

(7) Instructions

- An **instruction** is an executable statement 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.
- Four basic parts:
 - ☐ Label (optional)
 - ☐ Instruction mnemonic (required)
 - ☐ Operand(s) (usually required)
 - ☐ Comment (optional)

(a) Labels

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

- A **label** is an identifier that acts as a place marker for instructions and data.
- Follow **identifier** rules.

Data Label

Identifies the location of a **variable**.

- **Examples:** Defines a variable named count.

```
count    DWORD    100
```

Code Label

Used as **targets** of jumping and **looping** instructions that end with a colon (:) character.

- **Examples:** `jmp` instruction transfers control to the label **target**, creating a **loop**.

(b) Instructions Mnemonic

- An **instruction mnemonic** is a short word that identifies the operation carried out by an instruction.

- **Examples:**

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

(8) Operands

- Assembly language instructions can have between zero and three **operands** (depending on the type of instruction).
- The following table contains several sample operands:

Example	Operand Type
96, 2019h, 1011b	Constant (<i>immediate value</i>)
2 + 4	Constant expression
eax , ebx, ax, ah	Register
count	Memory (Data Label)

- Examples of assembly language instructions having varying numbers of operands:

```
stc                ; (no operand) set Carry flag
inc eax            ; (1 operand) add 1 to EAX
mov count, ebx     ; (2 operand) move EBX to count
```

Destination

Source

(9) Comments

- **Comments** are an important way for the writer of a program to describe/tell about the program's design to a person reading the source code.
- Typical comments:
 - Description of the program's purpose.
 - Names of persons who created and/or revised the program.
 - Program creation and revision dates.
 - Technical notes about the program's implementation.

Type of Comments

Single-line

- begin with **semicolon (;)**

```
; This is line 1.  
; This is line 2.  
; This is line 3.
```

Block / Multi-line

- begin with **COMMENT directive** & any a programmer-chosen **character**.
- end with the same programmer-chosen **character**.

```
COMMENT !  
    This line is comment 1.  
    This line is comment 2.  
!
```

Review Questions

3

1. Identify valid suffix characters used in integer constants.
2. (*Yes/No*): Is A5h a valid hexadecimal constant?
3. (*Yes/No*): Does the multiplication operator (*) have a higher precedence than the division operator (/) in integer expressions?
4. Write a constant expression that divides 10 by 3 and returns the integer remainder.
5. Show an example of a valid real number constant with an exponent.
6. (*Yes/No*): Must string constants be enclosed in single quotes?
7. Reserved words can be instruction mnemonics, attributes, operators, predefined symbols, and _____.
8. What is the maximum length of an identifier?
9. (*True/False*): An identifier cannot begin with a numeric digit.

Review Questions

3

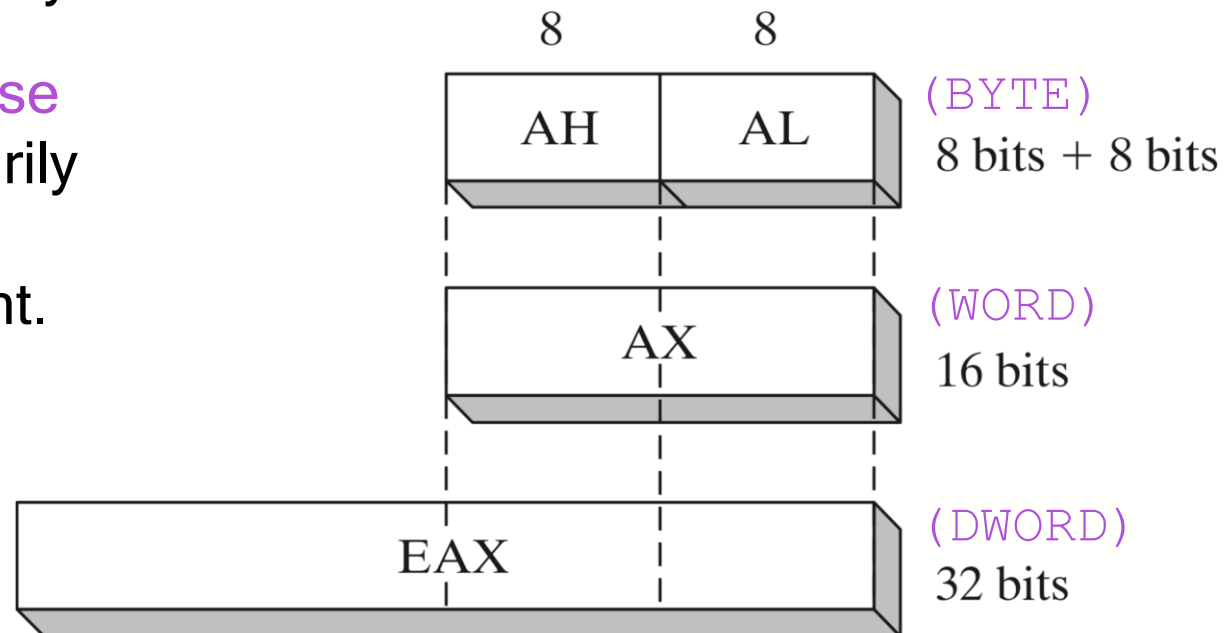


11. (*True/False*): Assembler directives execute at runtime.
12. (*True/False*): Assembler directives can be written in any combination of uppercase and lowercase letters.
13. Name the four basic parts of an assembly language instruction.
14. (*True/False*): MOV is an example of an instruction mnemonic.
15. (*True/False*): A code label is followed by a colon (:), but a data label does not have a colon.
16. Show an example of a block comment.
17. Why would it not be a good idea to use numeric addresses when writing instructions that access variables?

Registers

- *Registers* are high-speed storage locations directly inside the CPU, designed to be accessed at much higher speed than conventional memory.

- The *general-purpose registers* are primarily used for arithmetic and data movement.



- The same overlapping relationship exists for the EAX, EBX, ECX, and EDX registers:

32-Bit	16-Bit	8-Bit (High)	8-Bit (Low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

Module 3

Introduction to Assembly Language Programming

3.1 Introduction

3.2 Basic Elements of Assembly Language

3.3 Example: Adding and Subtracting Integers

□ Program Template

3.4 Defining Data

3.5 Symbolic Constants

3.6 Summary

Program Template

- Assembly language programs have a simple structure, with small variations.
- When begin a new program, it helps to start with an empty shell program with all basic elements in place.
- Redundant typing can be avoided by filling in the missing parts and saving the file under a new name.
- The next protected-mode program (`Template.asm`) can easily be customized.

```
TITLE Program Template                                (Template.asm)

; Program Description:
; Author:
; Creation Date:
; Revisions:
; Date:

INCLUDE Irvine32.inc

.data
    ; (insert variables here)

.code
main PROC
    ; (insert executable instructions here)

    exit
main ENDP

    ; (insert additional procedures here)
END main
```

Write the following program in
textpad or notepad and save as
Template.asm.

Program 3.1:

```
TITLE Program Template          (Template.asm)

; This is a template program

INCLUDE Irvine32.inc

.data
    ; (insert all variables here)

.code
main PROC

    ; (insert all executable instructions here)
    call    DumpRegs             ; display the registers
    exit

main ENDP

    ; (insert all additional procedures here)

END main
```

Example:



Adding & Subtracting

- *Registers* are used to hold the intermediate data, and we call a *library subroutine* to display the contents of the registers on the screen.
- Next is the program source code.
- Let's go through the program line indicated by the red font.


```
TITLE Add and Subtract          (AddSub.asm)

; This program adds and subtracts 32-bit integers

INCLUDE Irvine32.inc

.code
main PROC

    mov     eax, 10000h          ; start with 10000h
    add     eax, 40000h          ; add 40000h
    sub     eax, 20000h          ; subtract 20000h

    call    DumpRegs            ; display the registers
    exit

main ENDP
END main
```

mov → moves (copies) the integer
10000h to the EAX register

EAX = 0FC40000h

EAX = 00010000h

add → adds 40000h
to the EAX register

EAX = 00050000h

```
mov    eax, 10000h
add    eax, 40000h
sub    eax, 20000h
```

sub → subtracts 20000h
from the EAX register

EAX = 00030000h

TITLE Add

; This program adds and subtracts 32-bit integer

INCLUDE Irvine32.inc

.code

main PROC

; start with EAX = 0FC40000h

; add 40000h to EAX

; subtract 20000h from EAX

call DumpRegs ; display the contents of EAX

exit

main ENDP

END main

- The following is a snapshot of the program's output, generated by the call to `DumpRegs`:

```
EAX=00030000  EBX=7FFDF000  ECX=00000101  EDX=FFFFFFFF
ESI=00000000  EDI=00000000  EBP=0012FFF0  ESP=0012FFC4
EIP=00401024  EFL=00000206  CF=0   SF=0   ZF=0   OF=0
```

Write the following program in
textpad or notepad and save
as AddSub.asm.

Program 3.2:

```
TITLE Add and Subtract          (AddSub.asm)

; This program adds and subtracts 32-bit integers

INCLUDE Irvine32.inc

.code
main PROC

    mov     eax, 10000h          ; start with 10000h
    add     eax, 40000h          ; add 40000h
    sub     eax, 20000h          ; subtract 20000h

    call    DumpRegs            ; display the registers
    exit

main ENDP
END main
```

Module 3

Introduction to Assembly Language Programming

3.1 Introduction

- ❑ Intrinsic Data Types

3.2 Basic Elements of Assembly Language

- ❑ Data Definition Statement

- ❑ Little Endian Order

3.3 Example: Adding and Subtracting Integer

- ❑ Defining BYTE and SBYTE Data

- ❑ Defining WORD and SWORD Data

3.4 Defining Data

- ❑ Defining DWORD and SDWORD Data

3.5 Symbolic Constants

- ❑ Defining QWORD, TBYTE, Real Data

3.6 Summary

- ❑ Defining Real Number Data

- ❑ Adding Variables to AddSub Program

3.4 Defining Data

3

Intrinsic Data Types

- Each **intrinsic data types** describes a set of values that can be assigned to variables and expressions of the given type.
- The essential characteristic of each type is its size in bits:
→ 8, 16, 32, 48, 64, and 80.
- The assembler is not case sensitive, so a directive such as `DWORD` can be written as `dword`, `Dword`, `dWord`, and so on.

Table: Intrinsic Data Types.

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 (can also be a Near pointer in real-address mode)
SWORD	16-bit signed integer
DWORD	32-bit unsigned integer (can also be a Near pointer in protected mode). 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

Data Definition Statement

- A **data definition statement** sets aside storage in memory for a variable, with an optional name.
- It creates variables based on intrinsic data types (see previous table).
- A data definition has the following syntax:

*At least one
initializer*

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

- **Examples:**

count	DWORD	12345
nombor	BYTE	63
huruf	BYTE	'M'
nilai	WORD	25h, 7Ah, 99h

Little Endian Order

- x86 Processors store and retrieve data from memory using little endian order (low to high).
- The Least Significant Byte (LSB) is stored at the first memory address allocated for the data.
- The remaining bytes are stored in the next consecutive memory positions.
- **Example:** Double word 12345678h

Offset:	Value:	
0000:	78	(8 bits)
0001:	56	(8 bits)
0002:	34	(8 bits)
0003:	12	(8 bits)

Defining BYTE and SBYTE Data

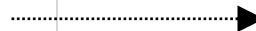
- The `BYTE` (define byte) and `SBYTE` (define signed byte) directives allocate storage for one or more unsigned or signed values.
- Each **initializer** must fit into **8 bits (1 Byte)** of storage.
- **Examples:**

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

- Sequences of bytes in memory:

value1	BYTE	'A'
value2	BYTE	0
value3	BYTE	255
value4	SBYTE	-128
value5	SBYTE	+127
value6	BYTE	?

Offset:	Value:
0000:	'A'
0001:	0
0002:	FF
0003:	80
0004:	7F
0005:	



Multiple Initializer

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

- **Examples:**

```
list1  BYTE  10, 20, 30
        BYTE  00100010b
list2  BYTE  41h, 'A',
```

Initializers can use different radices in a single data definition

Offset:	Value:	Data label:
0000:	10	list1
0001:	20	list1+1
0002:	30	list1+2
0003:	00100010	list1+3
0004:	41h	list2
0005:	'A'	list2+1

Defining Strings

- To define a string of **characters**, enclose them in single or double **quotation marks**.
- The most common type of string ends with a **null byte** (containing 0).
- Examples:

Character (\) will concatenate these two source code lines into a single statement

```
greeting1  BYTE "Good afternoon",0
greeting2  BYTE 'Good night',0
greeting3  \
            BYTE "Welcome to the "
            BYTE "Computer Organization "
            BYTE "and Architecture".
```

- The hexadecimal codes 0Dh and 0Ah are alternately called CR/LF (Carriage-Return / Line-Feed) or **end-of-line characters**.
- **Examples:**

```
greeting1 BYTE "Welcome to the COA class "  
            BYTE "with me.",0dh,0ah  
            BYTE "If you wish to pass, please "  
            BYTE "study hard.",0dh,0ah,0
```

The **DUP** directive tells the **assembler** to duplicate an expression a given number of times

3

DUP Operator

- The **DUP operator** allocates storage for multiple data items, using a constant expression as a counter.
- It is particularly useful when allocating space for a **string** or **array**.
- Examples:

```
BYTE 20 DUP(0)           ; 20 bytes, all equal to 0
BYTE 20 DUP(?)           ; 20 bytes, uninitialized
BYTE 3 DUP("COA")        ; 9 bytes: "COACOACO"
BYTE 10,2 DUP(0),20      ; 4 bytes: 0Ah,00,00,14h
```

Defining WORD and SWORD Data

- The WORD (define word) and SWORD (define signed word) directives create storage for one or more 16-bit (2-Byte) integers.
- Examples:

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


Defining DWORD and SDWORD Data

- The `DWORD` (define doubleword) and `SDWORD` (define signed doubleword) directives allocate storage for one or more **32-bit (4-byte)** integers:

```
val1 DWORD    12345678h    ; unsigned
val2 SDWORD   -2147483648   ; signed
```

→ -2147483648 is in decimal → **80000000h**

Remember little endian?

Offset:	Value (4 Bytes):				Data label:
0000:	78	56	34	12	var1
0004:	00	00	00	80	var2

Defining QWORD , TBYTE, Real Data

- The QWORD (define quadword) directive allocates storage for 64-bit (8-byte) values.
- The TBYTE directive to declare packed BCD (*Binary Coded Decimal*) variables in a 10-byte package
- Examples:

```
quad1    QWORD    1234567812345678h
intVal    TBYTE    80000000000000001234h ;    valid
intVal    TBYTE    -1234                    ;    invalid
```

*Constant initializers
must be in hexadecimal*

Defining Real Number Data

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

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

Table: Standard Real Number Types.

Data Type	Significant Digits	Approximate Range
Short real	6	1.18×10^{-38} to 3.40×10^{38}
Long real	15	2.23×10^{-308} to 1.79×10^{308}
Extended-precision real	19	3.37×10^{-4932} to 1.18×10^{4932}

Adding Variables to AddSub Program

- Using the `AddSub.asm` program from previous section, we can add a **data segment** containing several **doubleword variables**.
- The program in next slide.

```
TITLE Add and Subtract          (AddSub.asm)

; This program adds and subtracts 32-bit integers

INCLUDE Irvine32.inc

.code
main PROC

    mov     eax, 10000h          ; start with 10000h
    add     eax, 40000h          ; add 40000h
    sub     eax, 20000h          ; subtract 20000h

    call    DumpRegs            ; display the registers
    exit

main ENDP
END main
```

```
TITLE Add and Subtract, Version 2                (AddSub2.asm)
```

```
; This program adds and subtracts 32-bit integers  
; and stores the sum in a variable.
```

```
INCLUDE Irvine32.inc
```

```
.data
```

```
val1      dword  10000h
```

```
val2      dword  40000h
```

```
val3      dword  20000h
```

```
finalVal  dword  ?
```



Data Segment

```
.code
```

```
main PROC
```

```
        mov     eax, val1           ; start with 10000h  
        add     eax, val2           ; add 40000h  
        sub     eax, val3           ; subtract 20000h  
        mov     finalVal, eax       ; store the result (30000h)  
        call    DumpRegs            ; display the registers
```

```
        exit
```

```
main ENDP
```

```
END main
```

Program 3.3:

Write the following program in textpad or notepad and save as AddSub2.asm.

```
TITLE Add and Subtract          (AddSub2.asm)

; This program adds and subtracts 32-bit integers
; and stores the sum in a variable.

INCLUDE Irvine32.inc

.data
finalVal DWORD ?

.code
main PROC

    mov     eax, 10000h          ; start with 10000h
    add     eax, 40000h          ; add 40000h
    sub     eax, 20000h          ; subtract 20000h
    mov     finalVal, eax        ; store the result (30000h)

    call    DumpRegs            ; display the registers
    exit

main ENDP
END main
```


Review Questions

3

1. Create an uninitialized data declaration for a 16-bit signed integer.
2. Create an uninitialized data declaration for an 8-bit unsigned integer.
3. Create an uninitialized data declaration for an 8-bit signed integer.
4. Create an uninitialized data declaration for a 64-bit integer.
5. Which data type can hold a 32-bit signed integer?
6. Declare a 32-bit signed integer variable and initialize it with the smallest possible negative decimal value. (*Hint*: Refer to integer ranges in Chapter 1.)
7. Declare an unsigned 16-bit integer variable named **wArray** that uses three initializers.
8. Declare a string variable containing the name of your favorite color. Initialize it as a null-terminated string.
9. Declare an uninitialized array of 50 unsigned doublewords named **dArray**.
10. Declare a string variable containing the word “TEST” repeated 500 times.
11. Declare an array of 20 unsigned bytes named **bArray** and initialize all elements to zero.
12. Show the order of individual bytes in memory (lowest to highest) for the following double-word variable:

```
val1 DWORD 87654321h
```

Module 3

Introduction to Assembly Language Programming

3.1 Introduction

3.2 Basic Elements of
Assembly Language

3.3 Example: Adding and
Subtracting Integers

3.4 Defining Data

3.5 Symbolic Constants

3.6 Summary

- Overview
- Equal-Sign (=) Directive
- EQU Directive

3.5 Symbolic Constants

3

Overview

- 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

- Use the **equal-sign directive** (=) to create symbols representing **integer** expressions.

- The syntax:

`name = expression`

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

- **Example:**

```
COUNT = 5
mov al,COUNT      ; AL = 5
```

```
name EQU expression  
name EQU symbol  
name EQU <text>
```

EQU Directive

- The **EQU directive** associates a symbolic name with an **integer** expression or some arbitrary **text**.

- **Examples:**

For integer

```
PI EQU <3.1416>
```

```
pressKey EQU <"Press any key to continue...",0>
```

```
.
```

```
.
```

```
.data
```

```
prompt BYTE pressKey
```

*For text
expression*

Program 3.4:

Write the following program in textpad or notepad and save as `main.asm`.

```
TITLE TEST                                (main.asm)

; This is an executable program
; with EQU directive

INCLUDE Irvine32.inc


myName EQU <"Write your name here", 0dh, 0ah, 0>

.data
myMessage BYTE myName

.code
main PROC

    call    Clrscr                        ; to clear screen
    mov     edx, offset myMessage        ; start with the address
    call    WriteString                  ; display the message
    exit

main ENDP
END main
```



*For adding
new line*

Output:

```
Write your name here  
Press any key to continue . . .
```

1. Use this as the format for your lab report
2. Answer all activities and submit your lab report (pdf only) on e-learning by 8 March 5pm



Lab 1 - COA

Group Members:

Member 1

Member 2

Member 3

Activity 1

3

Exercise 3.1:

What are the sequences of bytes (in hexadecimal) in storage.

```
MyData    DWORD    3740625, 2 DUP (2ABCD0Eh)
```

MyData DWORD 3740625, 2 DUP (2ABCD0Eh)

Solution 3.1:

→ 3740625 is in decimal → 003913D1h

<i>Offset:</i>	<i>Value:</i>
0000:	
0001:	
0002:	
0003:	
0004:	
0005:	

<i>Offset:</i>	<i>Value:</i>
0006:	
0007:	
0008:	
0009:	
000A:	
000B:	

Activity 2

3

Exercise 3.2:

What are the sequences of bytes (in hexadecimal) in storage.

```
MyData2    WORD    2 DUP (4Fh) , ?, 'B' , 25, 500Dh
```

MyData2 WORD 2 DUP (4Fh) , ? , 'B' , 25 , 500Dh

Solution 3.2:

→ B = 42h

→ 25₁₀ = 19h

<i>Offset:</i>	<i>Value:</i>
0000:	
0001:	
0002:	
0003:	
0004:	
0005:	

<i>Offset:</i>	<i>Value:</i>
0006:	
0007:	
0008:	
0009:	
000A:	
000B:	

Activity 3

Exercise 3.3:

You are given the following code snippets. Answer these questions (in Hex).

1. Duplicate template-lab as Lab1Ex3
2. Update main.asm with the code in the box and build project
3. Debug the code and display the 4 windows (registers, memory, watch, autos) to get the values below
4. Screenshot your VS2019 with the 4 windows while running debugging to put in lab report

- AH =
- AX =
- BX =

```
.data
cth    BYTE  34H
cth2   BYTE  55H

val1   WORD  6677H
val2   WORD  8899H

.code
main PROC

MOV  AX, 0
MOV  AH, cth
MOV  BX, 0
MOV  BX, val1+1
exit
main ENDP
```

+ Example screenshot for lab

70

The screenshot displays the Visual Studio IDE with the following components:

- Menu Bar:** File, Edit, View, Git, Project, Build, Debug, Test, Analyze, Tools, Extensions, Window, Help.
- Toolbar:** Includes icons for search, run, and other debugging functions.
- Process:** [10228] Project.exe
- Thread:** [12188] Main Thread
- Stack Frame:** main

Solution Explorer: Shows the project structure with 'main.asm' selected.

main.asm: Assembly code for a program. It includes Irvine32.inc, defines variables val1, val2, val3, and finalVal, and contains a main PROC with instructions to move, add, subtract, and call DumpRegs.

Registers: A window showing the current state of registers. EAX is 00030000, EBX is 00249000, ECX is 004010AA, EDX is 004010AA, ESI is 004010AA, EDI is 004010AA, EIP is 00403676, ESP is 0019FF84, EBP is 0019FF94, and EFL is 00000206.

Memory: A window showing memory addresses and their corresponding values. The address 0x00405FFE is highlighted.

Diagnostic Tools: A window showing the diagnostics session (0 seconds, 77 ms selected) and process memory (KB).

Watch 1: A window showing the values of variables. The table below lists the variables and their values:

Name	Value	Type
val1	65536	unsigned long
val2	262144	unsigned long
val3	131072	unsigned long
finalVal	196608	unsigned long
eax	196608	unsigned int
val1	65536	unsigned long

Output: A window showing the output of the program, including the thread 0x2ac exiting with code 0 (0x0).

3.7 Summary

3

- An **integer expression** is a mathematical expression involving integer constants, symbolic constants, and arithmetic operators.
- Assembly language has a set of **reserved words** with special meanings that may only be used in the correct context.
- **Operands** are values passed to instructions. An assembly language instruction can have between zero and three operands, each of which can be a register, memory operand, or constant expression.

- Each assembler recognizes **intrinsic data types**, each of which describes a set of values that can be assigned to variables and expressions of the given type: BYTE, WORD, DWORD, REAL8
- x86 processors store and retrieve **data** from memory using **little endian** order.

Programming Exercise

3

- 3.1 Using the `AddSub` program as a reference, write a program that subtracts three integers using only 16-bit registers. Insert a `call DumpRegs` statement to display the register values.