SECR2033
Computer Organization and Architecture

Lecture slides prepared by "Computer Organization and Architecture", 9/e, by William Stallings, 2013.

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.

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.5.

Module 3

Introduction to Assembly Language Programming

3.1 Introduction

3

- *Assembly language* is the oldest programming language, and of all languages, bears the closest resemblance to native *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.

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.2. 4

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.

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.59. 6

3 (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 _____) 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 *(alternate)*
d	Decimal	y	Binary *(alternate)*
b	Binary		

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.59. 7

3

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

```
26          42o
26d         1Ah
11010011b   0A3h
42q
```

_____ *since no radix*

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

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.59. 8

3 (2) Real Number Constants

- Represented as decimal reals or encoded (_____) 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
```

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.61. 9

3 (3) Character & String Constants

- A _____ is a single character enclosed in single or double quotes.
- **Examples**:

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

- ASCII character = 1 byte.

- A _____ 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.

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.61. 10

3

- 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"'
```

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.61. 11

3

(4) Reserved Words

- Reserved words cannot be used as identifiers:
  - Instruction mnemonics (`MOVE`, `ADD`, `MUL`)
  - Directives (`INCLUDE`)
  - _____ (`BYTE`, `WORD`)
  - Operators (`+`, `-`)
  - Predefined symbols (`@data`)

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.62. 12

3

(5) Identifiers

- An identifier is a programmer-chosen name.
- It might identify a _____, 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.

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.62. 13

3

- 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
```

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.62. 14

3

(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 _____ have different directives.

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.62. 15

3

- Example to show the difference between directives and instructions:

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

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

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

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.62. 16

3 (7) Instructions

- An instruction is an executable statement when a program is assembled.
- *Instructions* are translated by the _____ 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)

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.63. 17

(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.

Identifies the location of a variable.	Used as targets of jumping and looping instructions that end with a colon (:) character.
■ **Examples**: Defines a variable named count. `count DWORD 100`	■ **Examples**: `jmp` instruction transfers control to the label target, creating a loop .

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.64. 18

3

(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 two values
        Subtract one value from another
        Multiply two values
        Jump to a new location
call    Call a procedure
```

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.64. 19

3

(8) Operands

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

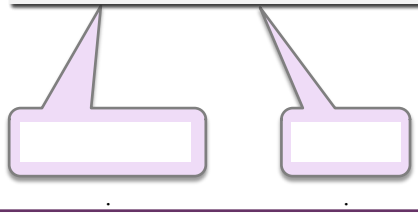
Example	Operand Type
96, 2019h, 1011b	Constant (*immediate value*)
2 + 4	Constant expression
eax , ebx, ax, ah	Register
count	Memory (Data Label)

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.64. 20

3

- 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
```



Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.64. 21

3

(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.

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.64. 22

3

Type of Comments

Single-line

- begin with semicolon (;)

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

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

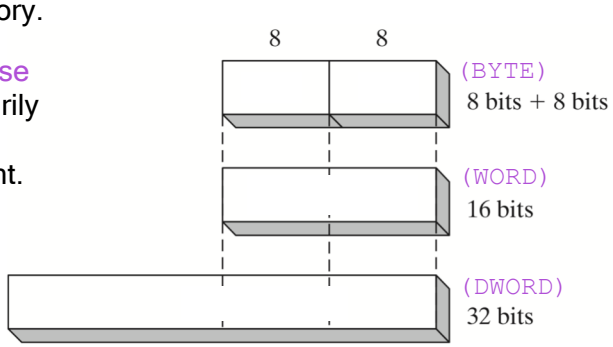
```
COMMENT !
  This line is comment 1.
  This line is comment 2.
!
```

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.65. 23

3

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.



Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.37. 24

3

- 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
		BH	BL
ECX	CX		
EDX			

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.38. 25

3

26

Module 3

Introduction to Assembly Language Programming

3 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.

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.70. 28

3

```
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
```

29

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
```

30

3

Example: Adding & Subtracting

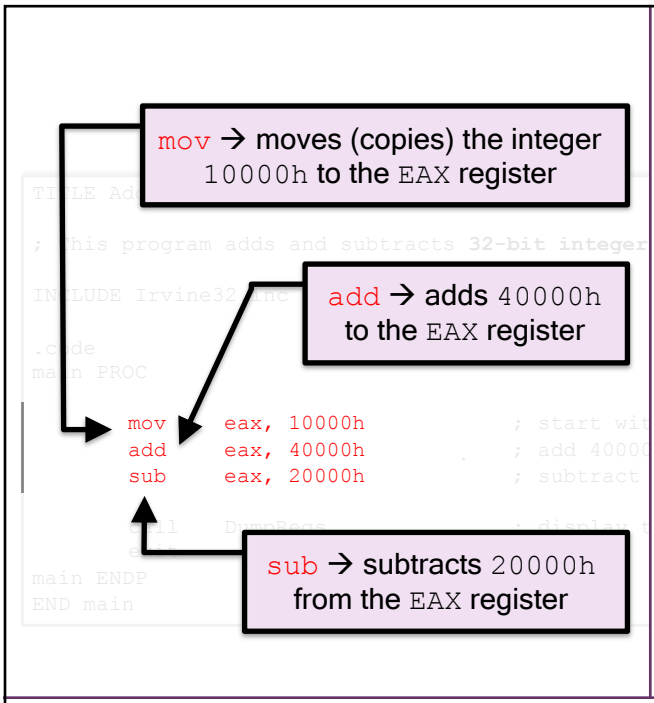
- *Registers* are used to hold the intermediate data, and we call a *library subroutine* to display the contents of the registers on the screen.
- Next is the program source code.
- Let's go through the program line indicated by the red font.

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.67. 31

3

```
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
```

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.67. 32



Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.67. 33

3

- 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
```

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.68. 34

Program 3.2:

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

```
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
```

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.64. 35

36

Module 3

Introduction to Assembly Language Programming

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.

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.77. 38

3

Table: Intrinsic Data Types.

Type	Usage
	8-bit unsigned integer. B stands for byte
SBYTE	8-bit signed integer. S stands for signed
	16-bit unsigned integer (can also be a Near pointer in real-address mode)
SWORD	16-bit signed integer
	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

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.78. 39

3

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]...
```

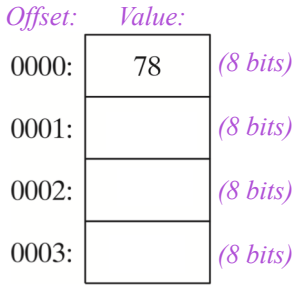
- **Examples**:

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

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.77. 40

3 Little Endian Order

- x86 Processors store and retrieve data from memory using little endian order (low to high).
- The _____ (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



Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.83. 41

3 Defining BYTE and SBYTE Data

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

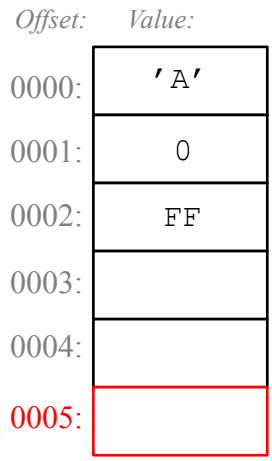
```
value1   BYTE      'A'   ; character constant
value2   BYTE       0    ; smallest unsigned byte
value3   BYTE     255    ; largest unsigned byte
value4   SBYTE   -128    ; smallest signed byte
value5   SBYTE   +127    ; largest signed byte
value6   BYTE       ?    ; uninitialized byte
```

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.78-79. 42

3

- Sequences of bytes in memory:

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



43

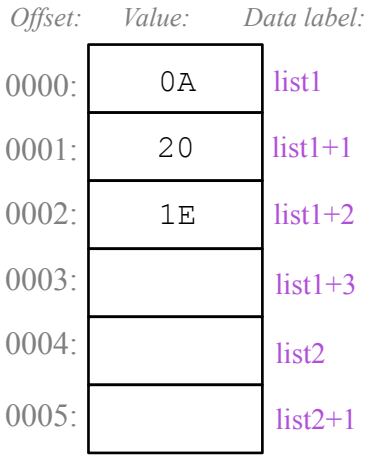
3

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   10,20h,30d
        BYTE   00100010b
list2   BYTE   45h,'A',
```

Initializers can use different radixes in a single data definition



Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.79.

44

3

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 concatenates 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".
```

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.80. 45

3

- 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.",_____,_____,0
```

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.80. 46

DUP Operator

3

- 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 _____ or _____.
- **Examples**:

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

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.80. 47

Defining WORD and SWORD Data

3

- The WORD (define word) and SWORD (define signed word) directives create storage for one or more 16-bit (___-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
```

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.80. 48

3 Defining DWORD and SDWORD Data

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

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

→ -2147483648 is in decimal → 80000000h

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

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.81. 49

3 Defining QWORD , TBYTE, Real Data

- The QWORD (define quadword) directive allocates storage for ____-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  80000000000001234h ;   valid
intVal  TBYTE  -1234              ; invalid
```

Constant initializers must be in hexadecimal

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.81-82. 50

3

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)
```

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.83. 51

3

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}

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.83. 52

3 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.

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.84. 53

```
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  ?

.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
```

54

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
```

55

3

56

Module 3
Introduction to Assembly Language Programming

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

3.5 Symbolic Constants

3

Overview

- A symbolic constant (or _____) 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

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.86

58

3 Equal-Sign (=) Directive

- Use the equal-sign directive (=) to create symbols representing _____ 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
```

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.86. 59

3 EQU Directive

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

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

For

```
PI EQU <3.1416>
pressKey EQU <"Press any key to continue...",0>
.
.
.data
prompt  BYTE   pressKey
```

For text expression

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.88. 60

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

61

Output:

3

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

62

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.

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.91. 63

3

- 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.

Irvine, K.R. (2011). *Assembly Language for x86 Processors* (6th Edition). New Jersey: Pearson Education Limited, p.91. 64