Introduction to Assembly Language

COE 205

Computer Organization and Assembly Language

Dr. Aiman El-Maleh

College of Computer Sciences and Engineering

King Fahd University of Petroleum and Minerals

[Adapted from slides of Dr. Kip Irvine: Assembly Language for Intel-Based Computers]

Presentation Outline

- Basic Elements of Assembly Language
- Flat Memory Program Template
- Example: Adding and Subtracting Integers
- Assembling, Linking, and Debugging Programs
- Defining Data
- Defining Symbolic Constants
- Data-Related Operators and Directives

Constants

- Integer Constants
  - Examples: –10, 42d, 10001101b, 0FF3Ah, 777o
  - Radix: b = binary, d = decimal, h = hexadecimal, and o = octal
  - If no radix is given, the integer constant is decimal
  - A hexadecimal beginning with a letter must have a leading 0
- Character and String Constants
  - Enclose character or string in single or double quotes
  - Examples: 'A', "d", 'ABC', "ABC", '4096'
  - Embedded quotes: "single quote ' inside", 'double quote " inside'
  - Each ASCII character occupies a single byte

Assembly Language Statements

- Three types of statements in assembly language
  - Typically, one statement should appear on a line

1. Executable Instructions
   - Generate machine code for the processor to execute at runtime
   - Instructions tell the processor what to do
2. Assembler Directives
   - Provide information to the assembler while translating a program
   - Used to define data, select memory model, etc.
   - Non-executable: directives are not part of instruction set
3. Macros
   - Shorthand notation for a group of statements
   - Sequence of instructions, directives, or other macros

Instructions

- Assembly language instructions have the format:

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

- Instruction Label (optional)
  - Marks the address of an instruction, must have a colon **:**
  - Used to transfer program execution to a labeled instruction
- Mnemonic
  - Identifies the operation (e.g. MOV, ADD, SUB, JMP, CALL)
- Operands
  - Specify the data required by the operation
  - Executable instructions can have zero to three operands
  - Operands can be registers, memory variables, or constants

Instruction Examples

- No operands

```
stc                 ; set carry flag
```

- One operand

```
inc  eax            ; increment register eax
call Clrscr         ; call procedure Clrscr
jmp  L1             ; jump to instruction with label L1
```

- Two operands

```
add  ebx, ecx       ; register ebx = ebx + ecx
sub  var1, 25       ; memory variable var1 = var1 - 25
```

- Three operands

```
imul eax,ebx,5      ; register eax = ebx * 5
```

Identifiers

- Identifier is a programmer chosen name
- Identifies variable, constant, procedure, code label
- May contain between 1 and 247 characters
- Not case sensitive
- First character must be a letter (A..Z, a..z), underscore(_), @, ?, or $.
- Subsequent characters may also be digits.
- Cannot be same as assembler reserved word.

Comments

- Comments are very important!
  - Explain the program's purpose
  - When it was written, revised, and by whom
  - Explain data used in the program
  - Explain instruction sequences and algorithms used
  - Application-specific explanations
- Single-line comments
  - Begin with a semicolon **;** and terminate at end of line
- Multi-line comments
  - Begin with **COMMENT** directive and a chosen character
  - End with the same chosen character

Next . . .

- Basic Elements of Assembly Language
- Flat Memory Program Template
- Example: Adding and Subtracting Integers
- Assembling, Linking, and Debugging Programs
- Defining Data
- Defining Symbolic Constants
- Data-Related Operators and Directives

Flat Memory Program Template

```
TITLE Flat Memory Program Template   (Template.asm)

; Program Description:
; Author:                        Creation Date:
; Modified by:                   Modification Date:

.686
.MODEL FLAT, STDCALL
.STACK

INCLUDE Irvine32.inc
.DATA
    ; (insert variables here)
.CODE
main PROC
    ; (insert executable instructions here)
    exit
main ENDP
    ; (insert additional procedures here)
END main
```

TITLE and .MODEL Directives

- **TITLE** line (optional)
  - Contains a brief heading of the program and the disk file name
- **.MODEL** directive
  - Specifies the memory configuration
  - For our purposes, the **FLAT** memory model will be used
    - Linear 32-bit address space (no segmentation)
  - **STDCALL** directive tells the assembler to use …
    - Standard conventions for names and procedure calls
- **.686** processor directive
  - Used **before** the **.MODEL** directive
  - Program can use instructions of Pentium P6 architecture
  - At least the **.386** directive should be used with the **FLAT** model

.STACK, .DATA, & .CODE Directives

- **.STACK** directive
  - Tells the assembler to define a runtime stack for the program
  - The size of the stack can be optionally specified by this directive
  - The runtime stack is required for procedure calls
- **.DATA** directive
  - Defines an area in memory for the program data
  - The program's variables should be defined under this directive
  - Assembler will allocate and initialize the storage of variables
- **.CODE** directive
  - Defines the code section of a program containing instructions
  - Assembler will place the instructions in the code area in memory

INCLUDE, PROC, ENDP, and END

- **INCLUDE** directive
  - Causes the assembler to include code from another file
  - We will include **Irvine32.inc** provided by the author Kip Irvine
    - Declares procedures implemented in the **Irvine32.lib** library
    - To use this library, you should link **Irvine32.lib** to your programs
- **PROC** and **ENDP** directives
  - Used to define procedures
  - As a convention, we will define ***main*** as the first procedure
  - Additional procedures can be defined after ***main***
- **END** directive
  - Marks the end of a program
  - Identifies the name (***main***) of the program's startup procedure

Next . . .

- Basic Elements of Assembly Language
- Flat Memory Program Template
- Example: Adding and Subtracting Integers
- Assembling, Linking, and Debugging Programs
- Defining Data
- Defining Symbolic Constants
- Data-Related Operators and Directives

Adding and Subtracting Integers

```
TITLE Add and Subtract                (AddSub.asm)
; This program adds and subtracts 32-bit integers.
.686
.MODEL FLAT, STDCALL
.STACK
INCLUDE Irvine32.inc

.CODE
main PROC
   mov eax,10000h              ; EAX = 10000h
   add eax,40000h              ; EAX = 50000h
   sub eax,20000h              ; EAX = 30000h
   call DumpRegs               ; display registers
   exit
main ENDP
END main
```

Example of Console Output

Procedure **DumpRegs** is defined in **Irvine32.lib** library

It produces the following console output,

showing registers and flags:

```
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
```

Suggested Coding Standards

- Some approaches to capitalization
  - Capitalize nothing
  - Capitalize everything
  - Capitalize all reserved words, mnemonics and register names
  - Capitalize only directives and operators
  - MASM is NOT case sensitive: does not matter what case is used
- Other suggestions
  - Use meaningful identifier names
  - Use blank lines between procedures
  - Use indentation and spacing to align instructions and comments
    - Use tabs to indent instructions, but do not indent labels
    - Align the comments that appear after the instructions

Understanding Program Termination

- The **exit** at the end of main procedure is a **macro**
  - Defined in **Irvine32.inc**
  - Expanded into a call to **ExitProcess** that terminates the program
  - **ExitProcess** function is defined in the **kernel32** library
  - We can replace **exit** with the following:

```
push 0                  ; push parameter 0 on stack
call ExitProcess        ; to terminate program
```

  - You can also replace **exit** with: **`INVOKE ExitProcess, 0`**
- **PROTO** directive (Prototypes)
  - Declares a procedure used by a program and defined elsewhere

```
ExitProcess PROTO, ExitCode:DWORD
```

  - Specifies the parameters and types of a given procedure

Modified Program

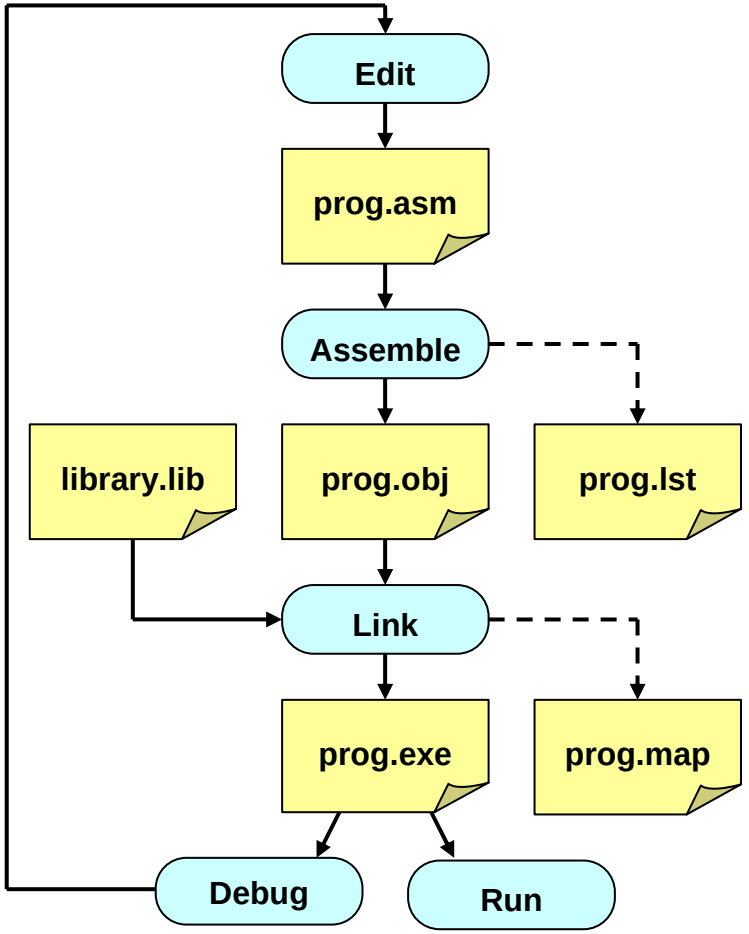
```
TITLE Add and Subtract                  (AddSubAlt.asm)
; This program adds and subtracts 32-bit integers

.686
.MODEL flat,stdcall
.STACK 4096

; No need to include Irvine32.inc
ExitProcess PROTO, dwExitCode:DWORD

.code
main PROC
   mov  eax,10000h              ; EAX = 10000h
   add  eax,40000h              ; EAX = 50000h
   sub  eax,20000h              ; EAX = 30000h

   push 0
   call ExitProcess             ; to terminate program
main ENDP
END main
```

Next . . .

- Basic Elements of Assembly Language
- Flat Memory Program Template
- Example: Adding and Subtracting Integers
- Assembling, Linking, and Debugging Programs
- Defining Data
- Defining Symbolic Constants
- Data-Related Operators and Directives

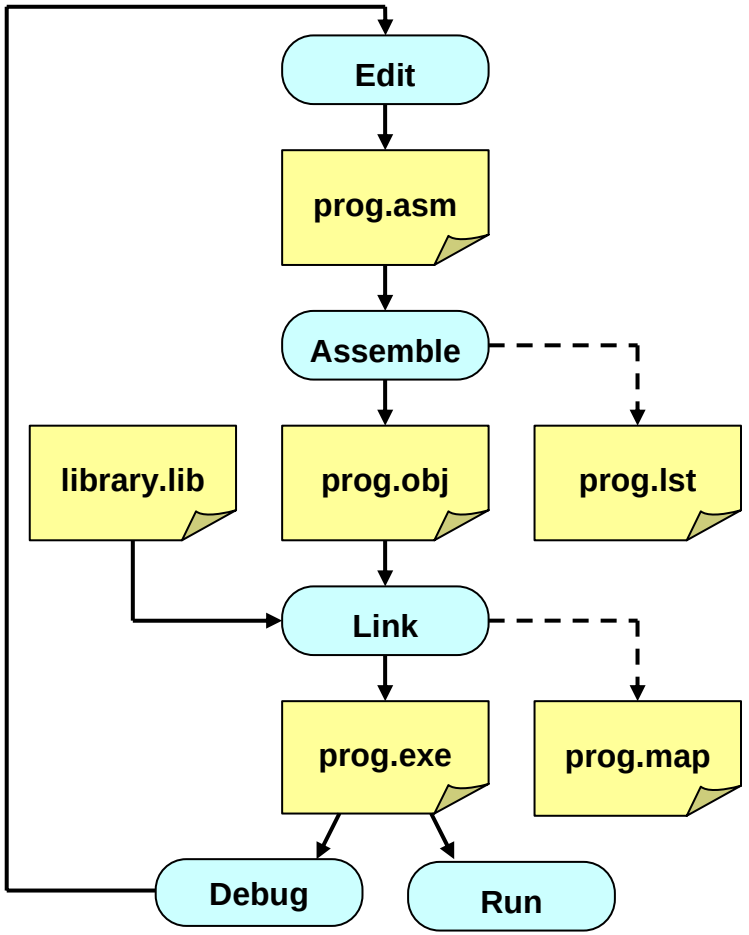
Assemble-Link-Debug Cycle

- Editor
  - Write new (**.asm**) programs
  - Make changes to existing ones
- Assembler: **ML.exe** program
  - Translate (**.asm**) file into object (**.obj**) file in machine language
  - Can produce a listing (**.lst**) file that shows the work of assembler
- Linker: **LINK32.exe** program
  - Combine object (**.obj**) files with link library (**.lib**) files
  - Produce executable (**.exe**) file
  - Can produce optional (**.map**) file



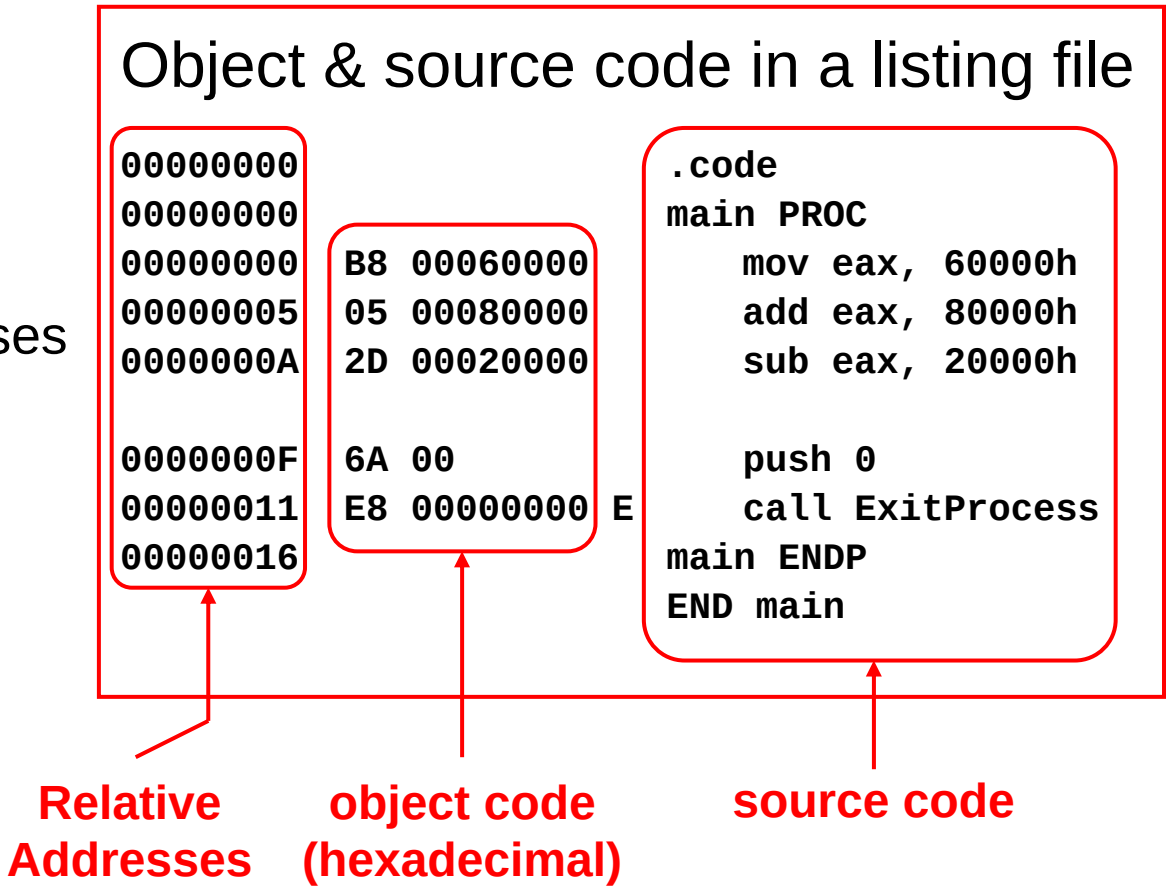
Assemble-Link-Debug Cycle – cont'd

- **MAKE32.bat**
  - Batch command file
  - Assemble and link in one step
- Debugger: **WINDBG.exe**
  - Trace program execution
    - Either step-by-step, or
    - Use breakpoints
  - View
    - Source (**.asm**) code
    - Registers
    - Memory by name & by address
    - Modify register & memory content
  - Discover errors and go back to the editor to fix the program bugs



Listing File

- Use it to see how your program is assembled
- Contains
  - Source code
  - Object code
  - Relative addresses
  - Segment names
  - Symbols
    - Variables
    - Procedures
    - Constants



Next . . .

- Basic Elements of Assembly Language
- Flat Memory Program Template
- Example: Adding and Subtracting Integers
- Assembling, Linking, and Debugging Programs
- Defining Data
- Defining Symbolic Constants
- Data-Related Operators and Directives

Intrinsic Data Types

- BYTE, SBYTE
  - 8-bit unsigned integer
  - 8-bit signed integer
- WORD, SWORD
  - 16-bit unsigned integer
  - 16-bit signed integer
- DWORD, SDWORD
  - 32-bit unsigned integer
  - 32-bit signed integer
- QWORD, TBYTE
  - 64-bit integer
  - 80-bit integer

- REAL4
  - IEEE single-precision float
  - Occupies 4 bytes
- REAL8
  - IEEE double-precision
  - Occupies 8 bytes
- REAL10
  - IEEE extended-precision
  - Occupies 10 bytes

Data Definition Statement

- Sets aside storage in memory for a variable
- May optionally assign a name (label) to the data
- Syntax:

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

```
val1   BYTE    10
```

- All initializers become binary data in memory

Defining BYTE and SBYTE Data

Each of the following defines a single byte 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
```

- MASM does not prevent you from initializing a BYTE with a negative value, but it's considered poor style.
- If you declare a SBYTE variable, the Microsoft debugger will automatically display its value in decimal with a leading sign.

Defining Byte Arrays

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
```

Defining Strings

- A string is implemented as an array of characters
  - For convenience, it is usually enclosed in quotation marks
  - It is often terminated with a NULL char (byte value = 0)
- Examples:

```
str1 BYTE "Enter your name", 0
str2 BYTE 'Error: halting program', 0
str3 BYTE 'A','E','I','O','U'
greeting  BYTE "Welcome to the Encryption "
          BYTE "Demo Program", 0
```

Defining Strings – cont'd

- To continue a single string across multiple lines, end each line with a comma

```
menu BYTE "Checking Account",0dh,0ah,0dh,0ah,
    "1. Create a new account",0dh,0ah,
    "2. Open an existing account",0dh,0ah,
    "3. Credit the account",0dh,0ah,
    "4. Debit the account",0dh,0ah,
    "5. Exit",0ah,0ah,
    "Choice> ",0
```

- End-of-line character sequence:
  - 0Dh = 13 = carriage return
  - 0Ah = 10 = line feed

Idea: Define all strings used by your program in the same area of the data segment

Using the DUP Operator

- Use DUP to allocate space for an array or string
  - Advantage: more compact than using a list of initializers
- Syntax

  counter **DUP (*argument*)**

  Counter and *argument* must be constants expressions
- The DUP operator may also be nested

```
var1 BYTE 20 DUP(0)           ; 20 bytes, all equal to zero
var2 BYTE 20 DUP(?)           ; 20 bytes, all uninitialized
var3 BYTE 4 DUP("STACK")      ; 20 bytes: "STACKSTACKSTACKSTACK"
var4 BYTE 10,3 DUP(0),20      ; 5 bytes: 10, 0, 0, 0, 20
var5 BYTE 2 DUP(5 DUP('*'), 5 DUP('!')) ; '*****!!!!!*****!!!!!'
```

Defining 16-bit and 32-bit Data

- Define storage for 16-bit and 32-bit integers
  - Signed and Unsigned
  - Single or multiple initial values

```
word1  WORD   65535         ; largest unsigned 16-bit value
word2  SWORD  –32768        ; smallest signed 16-bit value
word3  WORD   "AB"          ; two characters fit in a WORD
array1 WORD   1,2,3,4,5     ; array of 5 unsigned words
array2 SWORD  5 DUP(?)      ; array of 5 signed words
dword1 DWORD  0ffffffffh    ; largest unsigned 32-bit value
dword2 SDWORD –2147483648   ; smallest signed 32-bit value
array3 DWORD  20 DUP(?)   ; 20 unsigned double words
array4 SDWORD –3,–2,–1,0,1 ; 5 signed double words
```

QWORD, TBYTE, and REAL Data

- QWORD and TBYTE
  - Define storage for 64-bit and 80-bit integers
  - Signed and Unsigned
- REAL4, REAL8, and REAL10
  - Defining storage for 32-bit, 64-bit, and 80-bit floating-point data

```
quad1 QWORD  1234567812345678h
val1  TBYTE  100000000123456789Ah
rVal1 REAL4  -2.1
rVal2 REAL8  3.2E-260
rVal3 REAL10 4.6E+4096
array REAL4  20 DUP(0.0)
```

Symbol Table

- Assembler builds a symbol table
  - So we can refer to the allocated storage space by name
  - Assembler keeps track of each name and its offset
  - Offset of a variable is relative to the address of the first variable

- Example

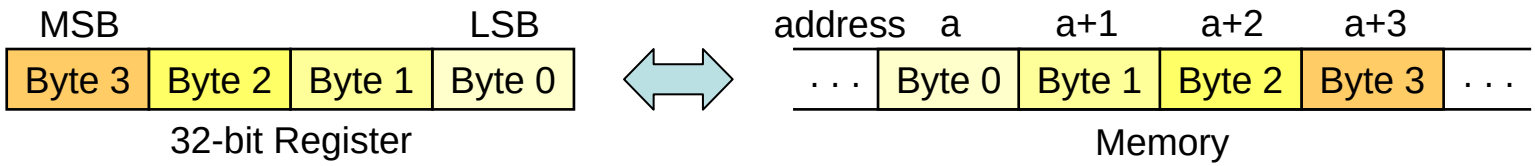
```
.DATA
value   WORD    0
sum     DWORD   0
marks   WORD   10 DUP (?)
msg     BYTE   'The grade is:',0
char1   BYTE   ?
```

Symbol Table

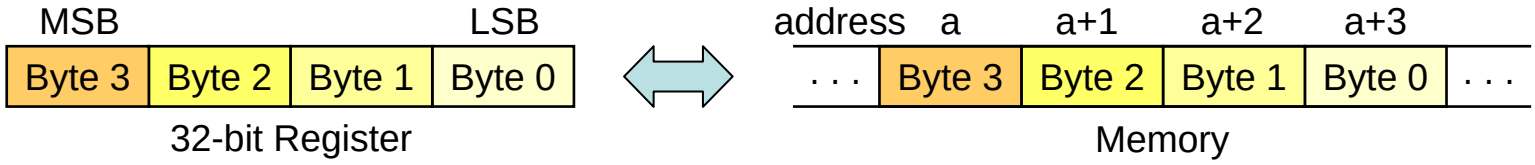
Name	Offset
value	0
sum	2
marks	6
msg	26
char1	40

Byte Ordering and Endianness

- Processors can order bytes within a word in two ways
- Little Endian Byte Ordering
  - Memory address = Address of **least significant byte**
  - Examples: Intel 80x86



- Big Endian Byte Ordering
  - Memory address = Address of **most significant byte**
  - Examples: MIPS, Motorola 68k, SPARC



Adding Variables to AddSub

```
TITLE Add and Subtract, Version 2                (AddSub2.asm)
.686
.MODEL FLAT, STDCALL
.STACK
INCLUDE Irvine32.inc
.DATA
val1   DWORD 10000h
val2   DWORD 40000h
val3   DWORD 20000h
result DWORD ?
.CODE
main PROC
    mov  eax,val1              ; start with 10000h
    add  eax,val2              ; add 40000h
    sub  eax,val3              ; subtract 20000h
    mov  result,eax            ; store the result (30000h)
    call DumpRegs              ; display the registers
    exit
main ENDP
END main
```

Next . . .

- Basic Elements of Assembly Language
- Flat Memory Program Template
- Example: Adding and Subtracting Integers
- Assembling, Linking, and Debugging Programs
- Defining Data
- Defining Symbolic Constants
- Data-Related Operators and Directives

Defining Symbolic Constants

- Symbolic Constant
  - Just a name used in the assembly language program
  - Processed by the assembler ⇒ pure text substitution
  - Assembler does NOT allocate memory for symbolic constants
- Assembler provides three directives:
  - = directive
  - EQU directive
  - TEXTEQU directive
- Defining constants has two advantages:
  - Improves program readability
  - Helps in software maintenance: changes are done in one place

Equal-Sign Directive

- *Name* = *Expression*
  - *Name* is called a symbolic constant
  - *Expression* is an integer constant expression
- Good programming style to use symbols

```
COUNT = 500      ; NOT a variable (NO memory allocation)
. . .
mov eax, COUNT   ; mov eax, 500
. . .
COUNT = 600      ; Processed by the assembler
. . .
mov ebx, COUNT   ; mov ebx, 600
```

- *Name* can be redefined in the program

EQU Directive

- Three Formats:

Name EQU *Expression*	Integer constant expression
Name EQU *Symbol*	Existing symbol name
Name EQU <*text*>	Any text may appear within < …>

```
SIZE     EQU 10*10          ; Integer constant expression
PI       EQU <3.1416>       ; Real symbolic constant
PressKey EQU <"Press any key to continue...",0>
.DATA
prompt BYTE PressKey
```

- No Redefinition: *Name* cannot be redefined with EQU

TEXTEQU Directive

- TEXTEQU creates a text macro. Three Formats:

Name TEXTEQU <*text*>	assign any text to *name*
Name TEXTEQU *textmacro*	assign existing text macro
Name TEXTEQU %*constExpr*	constant integer expression

- *Name* can be redefined at any time (unlike EQU)

```
ROWSIZE = 5
COUNT   TEXTEQU  %(ROWSIZE * 2)    ; evaluates to 10
MOVAL   TEXTEQU  <mov al,COUNT>
ContMsg TEXTEQU  <"Do you wish to continue (Y/N)?">
.DATA
prompt  BYTE     ContMsg
.CODE
MOVAL                              ; generates: mov al,10
```

Next . . .

- Basic Elements of Assembly Language
- Flat Memory Program Template
- Example: Adding and Subtracting Integers
- Assembling, Linking, and Debugging Programs
- Defining Data
- Defining Symbolic Constants
- Data-Related Operators and Directives

OFFSET Operator

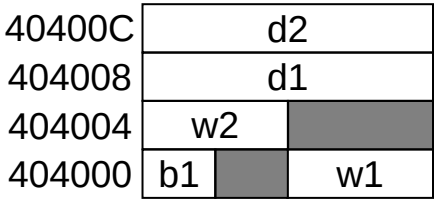
- OFFSET = address of a variable within its segment
  - In FLAT memory, one address space is used for code and data
  - OFFSET = linear address of a variable (32-bit number)

```
.DATA
bVal  BYTE  ?           ; Assume bVal is at 00404000h
wVal  WORD  ?
dVal  DWORD ?
dVal2 DWORD ?

.CODE
mov esi, OFFSET bVal    ; ESI = 00404000h
mov esi, OFFSET wVal    ; ESI = 00404001h
mov esi, OFFSET dVal    ; ESI = 00404003h
mov esi, OFFSET dVal2   ; ESI = 00404007h
```

ALIGN Directive

- ALIGN directive aligns a variable in memory
- Syntax: ALIGN *bound*
  - Where *bound* can be 1, 2, 4, or 16
- Address of a variable should be a multiple of *bound*
- Assembler inserts empty bytes to enforce alignment

```
.DATA          ; Assume that
b1 BYTE  ?     ; Address of b1 = 00404000h
ALIGN 2        ; Skip one byte
w1 WORD  ?     ; Address of w1 = 00404002h
w2 WORD  ?     ; Address of w2 = 00404004h
ALIGN 4        ; Skip two bytes
d1 DWORD ?     ; Address of d1 = 00404008h
d2 DWORD ?     ; Address of d2 = 0040400Ch
```



TYPE Operator

- TYPE operator
  - Size, in bytes, of a single element of a data declaration

```
.DATA
var1 BYTE ?
var2 WORD ?
var3 DWORD ?
var4 QWORD ?

.CODE
mov eax, TYPE var1   ; eax = 1
mov eax, TYPE var2   ; eax = 2
mov eax, TYPE var3   ; eax = 4
mov eax, TYPE var4   ; eax = 8
```

LENGTHOF Operator

- LENGTHOF operator
  - Counts the number of elements in a single data declaration

```
.DATA
array1    WORD    30 DUP(?),0,0
array2    WORD    5 DUP(3 DUP(?))
array3    DWORD   1,2,3,4
digitStr  BYTE   "12345678",0

.code
mov ecx, LENGTHOF array1          ; ecx = 32
mov ecx, LENGTHOF array2          ; ecx = 15
mov ecx, LENGTHOF array3          ; ecx = 4
mov ecx, LENGTHOF digitStr        ; ecx = 9
```

SIZEOF Operator

- SIZEOF operator
  - Counts the number of bytes in a data declaration
  - Equivalent to multiplying LENGTHOF by TYPE

```
.DATA
array1      WORD    30 DUP(?),0,0
array2      WORD    5 DUP(3 DUP(?))
array3      DWORD   1,2,3,4
digitStr    BYTE   "12345678",0

.CODE
mov ecx, SIZEOF array1              ; ecx = 64
mov ecx, SIZEOF array2              ; ecx = 30
mov ecx, SIZEOF array3              ; ecx = 16
mov ecx, SIZEOF digitStr            ; ecx = 9
```

Multiple Line Declarations

A data declaration spans multiple lines if each line (except the last) ends with a comma

The LENGTHOF and SIZEOF operators include all lines belonging to the declaration

In the following example, array identifies the first line WORD declaration only

Compare the values returned by LENGTHOF and SIZEOF here to those on the left

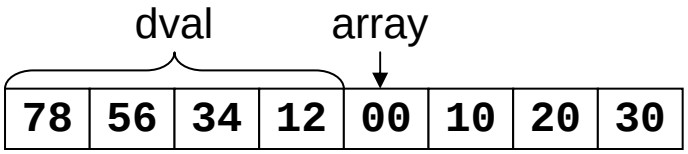
```
.DATA
array WORD 10,20,
           30,40,
           50,60

.CODE
mov eax, LENGTHOF array ; 6
mov ebx, SIZEOF array   ; 12
```

```
.DATA
array  WORD 10,20
       WORD 30,40
       WORD 50,60

.CODE
mov eax, LENGTHOF array ; 2
mov ebx, SIZEOF array   ; 4
```

PTR Operator

- PTR Provides the flexibility to access part of a variable
- Can also be used to combine elements of a smaller type
- Syntax: *Type* PTR (Overrides default type of a variable)

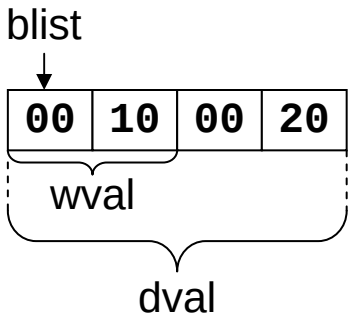
```
.DATA
dval  DWORD 12345678h
array BYTE  00h,10h,20h,30h

.CODE
mov al,  dval              ; error – why?
mov al,  BYTE PTR dval     ; al = 78h
mov ax,  dval              ; error – why?
mov ax,  WORD PTR dval     ; ax = 5678h
mov eax, array             ; error – why?
mov eax, DWORD PTR array   ; eax = 30201000h
```



LABEL Directive

- Assigns an alternate name and type to a memory location
- LABEL does not allocate any storage of its own
- Removes the need for the PTR operator
- Format: *Name* LABEL *Type*

```
.DATA
dval   LABEL DWORD
wval   LABEL WORD
blist  BYTE 00h,10h,00h,20h
.CODE
mov eax, dval      ; eax = 20001000h
mov cx,  wval      ; cx  = 1000h
mov dl,  blist     ; dl  = 00h
```



Summary

- Instruction ⇒ executed at runtime
- Directive ⇒ interpreted by the assembler
- .STACK, .DATA, and .CODE
  - Define the code, data, and stack sections of a program
- Edit-Assemble-Link-Debug Cycle
- Data Definition
  - BYTE, WORD, DWORD, QWORD, etc.
  - DUP operator
- Symbolic Constant
  - =, EQU, and TEXTEQU directives
- Data-Related Operators
  - OFFSET, ALIGN, TYPE, LENGTHOF, SIZEOF, PTR, and LABEL