Assembly Programming

Chapter3: Assembly Language Fundamentals

CSE3030
Prof. Youngjae Kim
Distributed Computing and Operating Systems Laboratory
https://discos.sogang.ac.kr

Overview

- This chapter is the focus of learning the basic building blocks of the Microsoft MASM assembler.
  - You will see how constants and variables are defined, standard formats for numeric and string literals, and how to assemble and run your first programs.

- Chapter Elements
  - Basic Language Elements
    - Identifiers, Directives, Instruction etc
    - Example: Adding and Subtracting Integers
  - Assembling, Linking, and Running Programs
    - The Assemble-Link-Execute Cycle
  - Defining Data
    - Data Types, BYTE and SBYTE Data, WORD and SWORD Data, etc
  - Symbolic Constants

Assembly Programming

- Assembly programming might have a reputation for being obscure and tricky, but

  Think of another way – it is a language that gives you nearly total information.

  You get everything that is going on, even in the CPU's registers and flags!

- However, programmers have the responsibility to manage data representation details and instruction formats at a very detailed level.

First Assembly Language Program

```
/* AddTwo program */


1: main PROC
2:   move eax, 5                    ; move 5 to the eax register
3:   add  eax, 6                    ; add 6 to the eax register
4:
5:   INVOKE ExitProcess, 0          ; end the program
6: main ENDP
```

Main procedure, the entry point of the program

eax register

Comments with ;

Calling a Winders service (known as a function)
To halt the program and return control to the operating system

Line numbers

The ending maker of the main procedure

First Assembly Language Program

sum is a variable with a size of 32 bits, using the DRWORD keyword.

Directive

In Assembly there are a number of size keywords, but they only specify a size.

```
/* Adding a Variable to AddTwo program*/
1:  .data                          ; this is the data area
2:  sum DRWORD 0                   ; create a variable named sum
3:
4:  .code
5:  main PROC
6:    move eax, 5                  ; move 5 to the eax register
7:    add  eax, 6                  ; add 6 to the eax register
8:    mov sum, eax
9:
10:  INVOKE ExitProcess, 0         ; end the program
11: main ENDP
```

Data segment

Code segment

Call a Window service that halts the program and returns control to the OS

We are going to learn language details, how to declare literals (constants), identifiers, directives, and instructions.

Microsoft Syntax Notation

- We will use Microsoft syntax notation throughout the book.

[…] : optional.

{ … | … | … } : a choice of one of the enclosed elements separated by **|** character.

Elements in ***italics*** : items which have known definitions or descriptions.

Integer Literals / Integer Constant

- Formats
  - **[{+|-}] *digits* [radix]**
  - **26, -26, 26d, 11010011b, 42q, 42o, 1Ah, 0A3h**
  - h : hexadecimal, q/o : octal, d : decimal,
    b : binary, r : encoded real, t : decimal(alt), y : binary(alt)

- Example
  - Integer literals declared with various radixes

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

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

Constant Integer Expressions

- A *constant integer expression* is a mathematical expression involving integer literals and arithmetic operators.
- Arithmetic Operators and precedence levels:

Operator	Name	Precedence Level
( )	parentheses	1
+, -	unary plus, minus	2
*, /	multiply, divide	3
MOD	modulus	3
+, -	add, subtract	4

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

Real Number Literals

- Formats
  - **[*sign*] *integer*.[*integer*][*exponent*]**
  - ***sign*** : **{+|-}**, ***exponent***: **E[{+|-}]*integer***

  - Encoded Real : The binary representation of a number.
    - **+1.0 ⇔ 0011 1111 1000 0000 0000 0000 0000 0000**

      or

      3F800000r

The encoded real represents a real number in hexadecimal, using the IEEE floating-point format for short reals. (Chapter 12)

Character and String Literals

- Formats
  - **'A', "x"**
  - **'A Boy', "A Girl"**
  - **'10010010'** and **10010010b** are different data.

  - Embedded quotes
    - **"It isn't a book"**
    - **'Say "Goodnight," Kim'**

- Note
  - String literals are stored in memory as sequences of integer byte values.
  - For example, the string literal "ABCD" contains the four bytes 41h, 42h, 43h, and 44h. The memory contains binary values as the following:

    1000 0001 1000 0010 1000 0011 1000 1000

Reserved Words

- *Reserved words* have special meaning.
  - Instruction mnemonics, such as MOV, ADD, and MUL
  - Register names
  - Directives, which tell the assembler how to assemble programs
  - Attributes, which provide size and usage information for variables and operands such as BYTE and WORD
  - Operators, used in constant expressions
  - Predefined symbols, such as @data, which return constant integer values at assembly time

- The reserved words are not case-sensitive.
  - For example, MOV is the same as mov and Mov.

Identifiers

- Formats
  - Programmer-chosen names of variables, constants, procedures and labels. 1-247 characters, including digits.
  - Case insensitive (by default).
  - First character must be a letter, _, @, or $.

- Reserved words can't be used as identifiers
  - Instruction mnemonics, directives, type attributes, operators, predefined symbols.

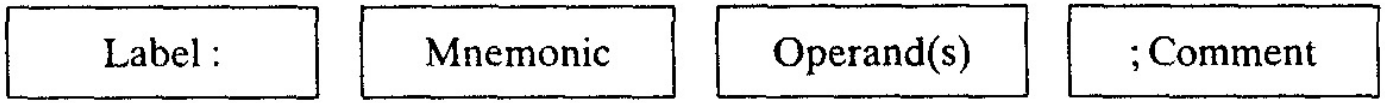
Directives

- A *directive* is a command understood by the assembler.
  - Case insensitive
    - Examples: .data, .DATA, and .Data are equivalent.
  - Not part of the Intel instruction set
  - Used to declare code, data areas, select memory model, declare procedures, etc.
    - Examples: `.data`, `.code`, *`name`* `PROC`, etc.
  - Different assembler may have different directives.

```
myVar    DWORD  26
Mov      eax, myVar

.data              ; .DATA directive to define variables

.code              ; .CODE directive to identify the area of a
                     program containing executable instructions

.stack 100h        ; .STACK directive to identify the area of a
                     program holding the runtime stack with size
```

Instructions

- An instruction is a statement that becomes executable when a program is assembled.
  - The instruction is translated by the assembler into machine language, which is loaded and executed by the CPU at runtime.
- Standard instruction format:



  - Labels
    - Marks the address (offset) of code and data
    - Code label : followed by colon
    - Data label : not followed by colon
  - Comments : begin with semicolon (;)
    - Good programs: programs with well written comments
    - For debugging, revision, documentation, etc

Instruction Format Examples

No operands

```
stc                     ; set Carry flag
```

A single operand

```
inc eax                 ; register
inc myByte              ; memory
```

Two operands

```
add ebx,ecx             ; register, register
sub myByte,25           ; memory, constant
add eax,36*25           ; register, constant-expression
```

Example: Adding and Subtracting Integers

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

add necessary definitions and setup inform.

beginning of the code segment

beginning of a procedure

calles predefined MS-Windows function that halt the proc.

the end of procedure ‘main’.

the last line of programs to be assembled.

Example: Adding and Subtracting Integers

- Example Output

```
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
```

Programming Suggestions

- Coding Styles : develop your own.
  - Some approaches to capitalization
    - Capitalize nothing, or everything.
    - Capitalize all reserved words, including instruction mnemonics and register names.
    - Capitalize only directives and operators
  - Other suggestions
    - Descriptive identifier names.
    - Blank lines between procedures.
  - Indentation and spacing
    - Code and data labels – no indentation.
    - Executable instructions – indent 4-5 spaces.
    - Comments: begin at column 40-45, aligned vertically.
    - 1-3 spaces between instruction and its operands.
      - ex: `mov  ax,bx`
    - 1-2 blank lines between procedures.

Alternative Version of Addsub

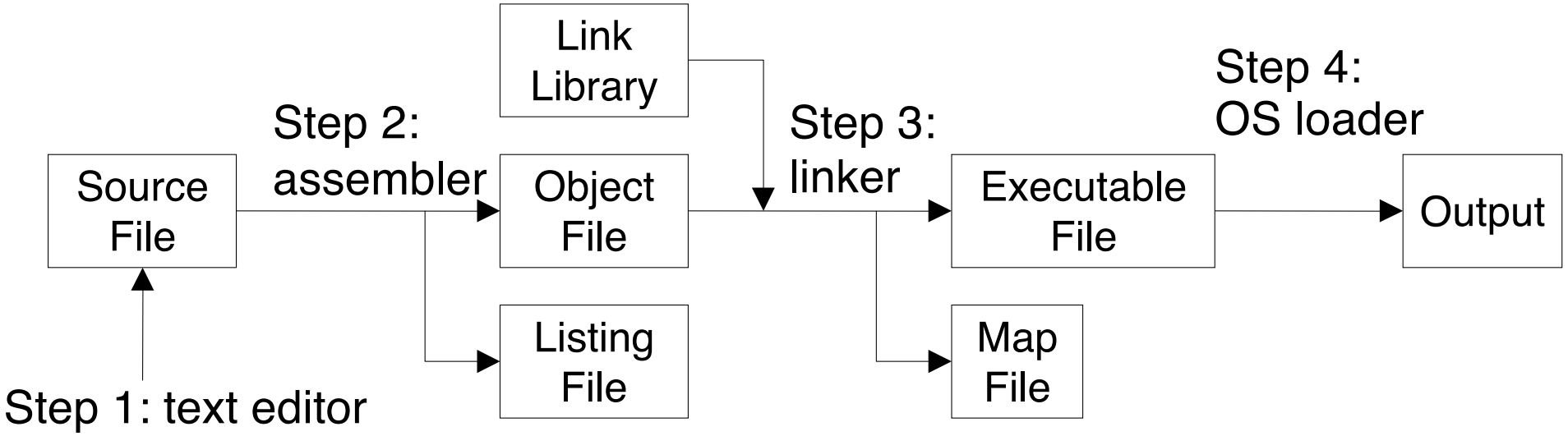
```
TITLE Add and Subtract          (AddSubAlt.asm)
; Adds and subtracts 32-bit integers.
.386                    minimum CPU required for this program
.MODEL flat,stdcall     flat : generate code for protected mode
.STACK 4096             stdcall : enables the calling of MS Windows
                                  functions.
ExitProcess PROTO, dwExitCode:DWORD
DumpRegs PROTO          procedure from Irvine32 link lib.
.code
main PROC
   mov eax,10000h          ; EAX = 10000h
   add eax,40000h          ; EAX = 50000h
   sub eax,20000h          ; EAX = 30000h
   call DumpRegs
   INVOKE ExitProcess,0
main ENDP               INVOKE : called a procedure or function
END main
```

- Program Template

```
TITLE Program Template                    (Template.asm)

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

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

Assembling, Linking, and Running Programs

- Assemble-Link Execute Cycle
  - The following diagram describes the steps from creating a source program through executing the compiled program.
  - If the source code is modified, Steps 2 through 4 must be repeated.



• Listing File

```
Microsoft (R) Macro Assembler Version 6.15.8803    10/26/01 13:50:21
Add and Subtract                (AddSub.asm)                Page 1 - 1

                        TITLE Add and Subtract                (AddSub.asm)

                        ; This program adds and subtracts 32-bit integers.

                        INCLUDE Irvine32.inc
                C       ; Include file for Irvine32.lib (Irvine32.inc)
                C       INCLUDE SmallWin.inc

00000000                .code
00000000                main PROC

00000000  B8 00010000       mov eax,10000h     ; EAX = 10000h
00000005  05 00040000       add eax,40000h     ; EAX = 50000h
0000000A  2D 00020000       sub eax,20000h     ; EAX = 30000h
0000000F  E8 00000000E      call DumpRegs

                          exit
0000001B                main ENDP
                        END main

Structures and Unions:  (omitted)

Segments and Groups:
N a m e                         Size     Length    Align  Combine Class
FLAT . . . . . . . . . . . . .  GROUP
STACK  . . . . . . . . . . . .  32 Bit  00001000  DWord  Stack   'STACK'
_DATA  . . . . . . . . . . . .  32 Bit  00000000  DWord  Public  'DATA'
_TEXT  . . . . . . . . . . . .  32 Bit  0000001B  DWord  Public  'CODE'

Procedures,  parameters and locals (list abbreviated):
N a m e                 Type   Value     Attr
CloseHandle . . . . . . P Near  00000000  FLAT  Length=00000000 External STDCALL
ClrScr . . . . . . . . . P Near  00000000  FLAT  Length=00000000 External STDCALL
...
main . . . . . . . . . . P Near  00000000 _TEXT  Length=0000001B Public STDCALL
Symbols (list abbreviated):
N a m e                         Type   Value      Attr
@CodeSize  . . . . . . . . . .  Number 00000000h
@DataSize  . . . . . . . . . .  Number 00000000h
@Interface . . . . . . . . . .  Number 00000003h
@Model . . . . . . . . . . . .  Number 00000007h
@code  . . . . . . . . . . . .  Text              _TEXT
@data  . . . . . . . . . . . .  Text              FLAT
@fardata?  . . . . . . . . . .  Text              FLAT
@fardata . . . . . . . . . . .  Text              FLAT
@stack . . . . . . . . . . . .  Text              FLAT
...
exit . . . . . . . . . . . . .  Text    INVOKE ExitProcess,0
              0 Warnings
              0 Errors
```

- Use it to see how your program is compiled
- Contains
  - source code
  - Addresses
  - object code (machine language)
  - segment names
  - symbols (variables, procedures, and constants)

- Project Property -> Configuration Properties -> Microsoft Macro Assembler -> Listing File -> Assembled Code Listing File -> $(InputName).lst

- Map File (by ***/MAP*** when LINK)
  - Information about each program segment:
    - starting address
    - ending address
    - size
    - segment type

Defining Data

- Data Definition Statement
  - A 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] ...
```

  - All initializers become binary data in memory

Defining Data

- Defining Bytes
  - Defining 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
```

  - Defining multiple bytes of storage with initialization:

```
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 Data

- Defining Strings
  - A string is implemented as an array of characters
    - For convenience, it is usually enclosed in quotation marks
    - It usually has a null byte at the end
  - 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 Demo"
         BYTE " program.", 0
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
```

0Dh = carriage return
0Ah = line feed

Define all the data in the same area of the data segment.

Defining Data

- DUP Operator
  - Use DUP to allocate (create space for) an array or string.
  - Counter and argument must be constants or constant expressions

```
var1 BYTE 20 DUP(0)     ; 20 bytes, all equal to zero
var2 BYTE 20 DUP(?)       ; 20 bytes, uninitialized
var3 BYTE 2 DUP("STACK") ; 10 bytes: "STACKSTACK"
Var4 BYTE 10, 3 DUP(0), 20
```

- Defining WORD and SWORD Data (16 bit)

```
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
```

- Defining DWORD and SDWORD Data (32 bit)

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

- Defining QWORD, TBYTE, Real Data

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

Defining Data

- 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.
  - Example:

```
val1 DWORD 12345678h
```

- Declaring Unitialized Data
  - Use the **.data?** directive to declare an unintialized data segment.
  - Within the segment, declare variables with "?" initializers:

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

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

Adding Variables to AddSub

```
TITLE Add and Subtract, Ver. 2          (AddSub2.asm)
; This program adds and subtracts 32-bit unsigned
; 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
```

Symbolic Constants

```
COUNT = 500
. . .
mov al,COUNT
```

- Equal-Sign Directive
  - ***name = expression*** (32bit int exp or const)
  - May be redefined and ***name*** is called a symbolic constant.
  - good programming style to use symbols.

- EQU Directive
  - Define a symbol as either an integer or text expression.
  - Cannot be redefined.

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

Symbolic Constants

- TEXTEQU Directive
  - Define a symbol as either an integer or text expression.
  - Called a text macro.
  - Can be redefined.

```
continueMsg TEXTEQU <"Do you wish to continue (Y/N)?">
rowSize = 5
.data
prompt1 BYTE continueMsg
count TEXTEQU %(rowSize * 2)  ; eval. the expression
move TEXTEQU <mov>
setupAL TEXTEQU <move al,count>
.code
setupAL                       ; generates: "mov al,10"
```

Symbolic Constants

- Current location counter: $
  - Subtract address of list.
  - Difference is the number of bytes.
  - Calculating the Size of a Byte Array

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

  - The Size of a Word Array

```
list WORD 1000h,2000h,3000h,4000h
ListSize = ($ - list) / 2
```

  - The Size of a Doubleword Array

```
list DWORD 1,2,3,4
ListSize = ($ - list) / 4
```

Real-Address Mode Programming

- Generate 16-bit MS-DOS Programs
- Advantages
  - Enables calling of MS-DOS and BIOS functions
  - No memory access restrictions
- Disadvantages
  - Must be aware of both segments and offsets
  - Cannot call Win32 functions (Windows 95 onward)
  - Limited to 640K program memory
- Requirements
  - INCLUDE Irvine16.inc
  - Initialize DS to the data segment:

```
mov ax,@data
mov ds,ax
```

Add and Subtract, 16-Bit Version

```
TITLE Add and Subtract, V. 2    (AddSub2.asm)
INCLUDE Irvine16.inc
.data
val1 DWORD 10000h
val2 DWORD 40000h
val3 DWORD 20000h
finalVal DWORD ?
.code
main PROC
   mov ax,@data   ; initialize DS
   mov ds,ax
   mov eax,val1   ; get first value
   add eax,val2   ; add second value
   sub eax,val3   ; subtract third value
   mov finalVal,eax ; store the result
   call DumpRegs     ; display registers
   exit
main ENDP
END main
```