Chapter 3: Assembly Language Fundamentals

Kip Irvine

Chapter Overview

- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- Assembling, Linking, and Running Programs
- Defining Data
- Symbolic Constants
- 64-Bit Programming

Basic Elements of Assembly Language

- Integer constants
- Integer expressions
- Character and string constants
- Reserved words and identifiers
- Directives and instructions
- Labels
- Mnemonics and Operands
- Comments
- Examples

Example Program

```
main PROC
mov eax, 5      ; move 5 to the EAX register
add eax, 6      ; add 6 to the EAX register
call WriteInt   ; display value in EAX
exit            ; quit
main ENDP
```

Add two numbers and displays the result

Integer Constants

- [{+ | -}] digits [radix]
- Optional leading + or – sign
- binary, decimal, hexadecimal, or octal digits
- Common radix characters:
  - h – hexadecimal
  - q | o – octal
  - d – decimal
  - b – binary
  - r – encoded real
- If no radix given, assumed to be decimal

Examples: 30d, 6Ah, 42, 1101b
Hexadecimal beginning with letter: 0A5h

Integer Expressions

integer values and arithmetic operators

- Operators and precedence levels:

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

Must evaluate to an integer that can be stored in 32 bits

These can be evaluated at assembly time – they are not runtime expressions

- Examples:

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

Precedence Examples:

4 + 5 * 2	Multiply, add
12 – 1 MOD 5	Modulus, subtract
-5 + 2	Unary minus, add
(4 + 2) * 6	Add, multiply

Real Number Constants

- Represented as decimal reals or encoded (hexadecimal) reals
- Decimal real contains optional sign followed by integer, decimal point, and optional integer that expresses a fractional and an optional exponent
  - [sign] integer.[integer] [exponent]
  - Sign {+, -}
  - Exponent E[{+, -}] integer
- Examples
  - 2.
  - +3.0
  - -44.2E+05
  - 26.E5

Character and String Constants

- Enclose character in single or double quotes
  - 'A', "x"
  - ASCII character = 1 byte
- Enclose strings in single or double quotes
  - "ABC"
  - 'xyz'
  - Each character occupies a single byte
- Embedded quotes:
  - 'Say "Goodnight," Gracie'

Reserved Words

- Reserved words cannot be used as identifiers
  - Instruction mnemonics
    - MOV, ADD, MUL, …
  - Register names
  - Directives – tells MASM how to assemble programs
  - type attributes – provides size and usage information
    - BYTE, WORD
  - Operators – used in constant expressions
  - predefined symbols – @data
  - See MASM reference for further details
    - https://msdn.microsoft.com/en-us/library/afzk3475.aspx?f=255&MSPPError=-2147217396

Identifiers

- Identifiers
  - Programmer-chosen name to identify a variable, constant, procedure, or code label
  - 1-247 characters, including digits
  - not case sensitive
  - first character must be a letter, _, @, ?, or $
    - Subsequent characters may also be digits
  - Cannot be the same as a reserved word
  - @ is used by assembler as a prefix for predefined symbols, so avoid it identifiers
- Examples
  - Var1, Count, $first, _main, MAX, open_file, myFile, xVal, _12345

Directives

- Commands that are recognized and acted upon by the assembler
  - Not part of the Intel instruction set
  - Used to declare code, data areas, select memory model, declare procedures, etc.
  - not case sensitive
- Different assemblers have different directives
  - NASM not the same as MASM, for example

```
myVar  DWORD 26          ; DWORD directive, set aside
                         ; enough space for double word
Mov    eax, myVar        ; MOV instruction
```


Instructions

- An instruction is a statement that becomes executable when a program is assembled.
- Assembled into machine code by assembler
- Executed at runtime by the CPU
- We use the Intel IA-32 instruction set
- An instruction contains:
  - Label (optional)
  - Mnemonic (required)
  - Operand (depends on the instruction)
  - Comment (optional)
- Basic syntax
  - [label:] mnemonic [operands] [; comment]

Labels

- Act as place markers
  - marks the address (offset) of code and data
- Follow identifer rules
- Data label
  - must be unique
  - example: **myArray** (not followed by colon)
  - `count DWORD 100`
- Code label
  - target of jump and loop instructions
  - example: **L1:** (followed by colon)

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


Mnemonics and Operands

- Instruction Mnemonics
  - memory aid
  - examples: MOV, ADD, SUB, MUL, INC, DEC
- Operands
  - constant 96
  - constant expression 2 + 4
  - register eax
  - memory (data label) count

Constants and constant expressions are often called immediate values

Mnemonics and Operands

Examples

```
STC instruction
        stc                     ; set Carry flag

INC instruction
        inc     eax             ; add 1 to EAX

MOV instruction
        mov     count, ebx      ; move EBX to count
                                ; first operation is destination
                                ; second is the source
IMUL instruction (three operands)
        imul    eax, ebx, 5     ; ebx multiplied by 5, product in EAX
```


Comments

- Comments are good!
  - explain the program's purpose
  - when it was written, and by whom
  - revision information
  - tricky coding techniques
  - application-specific explanations
- Single-line comments
  - begin with semicolon (;)
- Multi-line comments
  - begin with COMMENT directive and a programmer-chosen character
  - end with the same programmer-chosen character

Comments

- Single line comment
  - inc eax ; single line at end of instruction
  - ; single line at beginning of line
- Multiline comment

```
COMMENT !
    This line is a comment
    This line is also a comment
!
COMMENT &
    This is a comment
    This is also a comment
&
```


Instruction Format Examples

- No operands
  - stc ; set Carry flag
- One 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

NOP instruction

- Doesn't do anything
- Takes up one byte
- Sometimes used by compilers and assemblers to align code to even-address boundaries.
- The following MOV generates three machine code bytes. The NOP aligns the address of the third instruction to a doubleword boundary (even multiple of 4)

```
00000000    66    8B    C3    mov ax, bx
00000003    90                nop     ;  align next instruction
00000004    8B    D1          mov edx, ecx
```


What's Next

- Basic Elements of Assembly Language
- **Example: Adding and Subtracting Integers**
- Assembling, Linking, and Running Programs
- Defining Data
- Symbolic Constants
- 64-Bit Programming

Example: Adding and Subtracting Integers

```
; AddTwo.asm - adds two 32-bit integers

.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD
.code
main PROC
   mov  eax,5     ; move 5 to the EAX register
   add  eax,6     ; add  6 to the EAX register

   INVOKE ExitProcess,0
main ENDP
END main
```


Example Output

Showing registers and flags in the debugger:

```
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 (1 of 2)

- Some approaches to capitalization
  - capitalize nothing
  - capitalize everything
  - capitalize all reserved words, including instruction mnemonics and register names
  - capitalize only directives and operators
- Other suggestions
  - descriptive identifier names
  - spaces surrounding arithmetic operators
  - blank lines between procedures

Suggested Coding Standards (2 of 2)

- Indentation and spacing
  - code and data labels – no indentation
  - executable instructions – indent 4-5 spaces
  - comments: right side of page, aligned vertically
  - 1-3 spaces between instruction and its operands
    - ex: mov ax,bx
  - 1-2 blank lines between procedures

Required Coding Standards

- **Program Header**

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

- **Labels and Identifiers**
  - Labels use a mixture of uppercase and lowercase characters; and underscore characters should be avoided
  - In our coding standard, we strongly recommend the selective use of uppercase characters instead of underscores to indicate breaks in multi-word labels. For example:
    - use waitRDRF instead of wait_RDRF or
    - Use veryLongLabel instead of very_long_label

Required Coding Standards

- **File and Subroutine Headers**
  - Files and subroutines use title blocks to describe their purpose and to document other important information. These title blocks are sometimes called file headers and subroutine headers. Every file and every subroutine should include a header.

```
;*************************************************************************
;
;* RoutineName - expanded name or phrase describing purpose
;* Brief description, typically a few lines explaining the
;* purpose of the program.
;*
;* I/O: Explain what is expected and what is produced
;*
;* Calling Convention: How is the routine called?
;*
;* Stack Usage: (when needed) When a routine has several variables
;* on the stack, this section describes the structure of the
;* information.
;*
;* Information about routines that are called and any registers
;* that get destroyed. In general, if some registers are pushed at
;* the beginning and pulled at the end, it is not necessary to
;* describe this in the routine header.
;*************************************************************************
```


Alternative Version of AddSub

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

; This program adds and subtracts 32-bit integers.
.386
.MODEL flat,stdcall
.STACK 4096

ExitProcess PROTO, dwExitCode:DWORD
DumpRegs PROTO

.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
END main
```


Program Template

```
; Program Template            (Template.asm)

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

.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD
.data
; declare variables here
.code
main PROC
    ; write your code here
    INVOKE ExitProcess,0
main ENDP
; (insert additional procedures here)
END main
```


What's Next

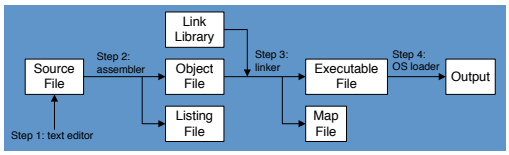
- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- **Assembling, Linking, and Running Programs**
- Defining Data
- Symbolic Constants
- 64-Bit Programming

Assembling, Linking, and Running Programs

- Assemble-Link-Execute Cycle
- Listing File
- Map File

Assemble-Link Execute Cycle

- Assembly language program must be translated to machine language for the target processor.
- 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.



Assemble-Link-Execute

Step 1: A programmer uses a text editor to create an ASCII text file named the source file.
Step 2: The assembler reads the source file and produces an object file, a machine-language translation of the program. Optionally, it produces a listing file. If any errors occur, the programmer must return to Step 1 and fix the program.
Step 3: The linker reads the object file and checks to see if the program contains any calls to procedures in a link library. The linker copies any required procedures from the link library, combines them with the object file, and produces the executable file.
Step 4: The operating system loader utility reads the executable file into memory and branches the CPU to the program's starting address, and the program begins to execute.

Listing File

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

Map File

- Information about each program segment:
  - starting address
  - ending address
  - size
  - segment type

What's Next

- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- Assembling, Linking, and Running Programs
- **Defining Data**
- Symbolic Constants
- 64-Bit Programming

Defining Data

- Intrinsic Data Types
- Data Definition Statement
- Defining BYTE and SBYTE Data
- Defining WORD and SWORD Data
- Defining DWORD and SDWORD Data
- Defining QWORD Data
- Defining TBYTE Data
- Defining Real Number Data
- Little Endian Order
- Adding Variables to the AddSub Program
- Declaring Uninitialized Data

Intrinsic Data Types (1 of 2)

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

Intrinsic Data Types (2 of 2)

- REAL4
  - 4-byte IEEE short real
- REAL8
  - 8-byte IEEE long real
- REAL10
  - 10-byte IEEE extended real

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*] . . .

```
value1 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
```

	Offset	Value
list1	0000	10
	0001	20
	0002	30
	0003	40
list2	0004	10
	0005	20
	0006	30
	0007	40
	0008	50
	0009	60
	000A	70
	000B	80
	000C	81
	000D	82
	000E	83
	000F	84
list3	0010	

Defining Strings (1 of 3)

- A string is implemented as an array of characters
  - For convenience, it is usually enclosed in quotation marks
  - It often will be null-terminated (ending with ,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 Demo program "
          BYTE "created by Kip Irvine.",0
```

Defining Strings (2 of 3)

- 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
```


Defining Strings (3 of 3)

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

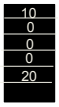
```
str1 BYTE "Enter your name:    ",0Dh,0Ah
     BYTE "Enter your address: ",0

newLine BYTE 0Dh,0Ah,0
```

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

Using the DUP Operator

- Use DUP to allocate (create space for) an array or string. Syntax: *counter* DUP (*argument*)
- *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 4 DUP("STACK")     ; 20 bytes: "STACKSTACKSTACKSTACK"
var4 BYTE 10,3 DUP(0),20     ; 5 bytes
```



Defining WORD and SWORD Data

- Define storage for 16-bit integers
  - or double characters
  - single value or multiple values

```
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

Storage definitions for signed and unsigned 32-bit integers:

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


Defining QWORD, TBYTE, Real Data

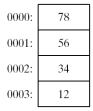
Storage definitions for quadwords, tenbyte values, and real numbers:

```
quad1 QWORD  1234567812345678h
val1  TBYTE  1000000000123456789Ah
rVal1 REAL4  -2.1
rVal2 REAL8  3.2E-260
rVal3 REAL10 4.6E+4096
ShortArray REAL4 20 DUP(0.0)
```

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
```



Adding Variables to AddSub

```
TITLE Add and Subtract, Version 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
```


Declaring Unitialized Data

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

```
.data?
```

- Within the segment, declare variables with "?" initializers:

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

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

What's Next

- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- Assembling, Linking, and Running Programs
- Defining Data
- **Symbolic Constants**
- 64-Bit Programming

Symbolic Constants

- Equal-Sign Directive
- Calculating the Sizes of Arrays and Strings
- EQU Directive
- TEXTEQU Directive

Equal-Sign Directive

- *name* = *expression*
  - expression is a 32-bit integer (expression or constant)
  - may be redefined
  - *name* is called a symbolic constant
- good programming style to use symbols

```
COUNT = 500
.
.
mov ax,COUNT
```

Calculating the Size of a Byte Array

- current location counter: $
  - subtract address of list
  - difference is the number of bytes

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


Calculating the Size of a Word Array

Divide total number of bytes by 2 (the size of a word)

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


Calculating the Size of a Doubleword Array

Divide total number of bytes by 4 (the size of a doubleword)

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


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
```


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)      ; evaluates the expression
setupAL TEXTEQU <mov al,count>

.code
setupAL                           ; generates: "mov al,10"
```


What's Next

- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- Assembling, Linking, and Running Programs
- Defining Data
- Symbolic Constants
- **64-Bit Programming**

64-Bit Programming

- MASM supports 64-bit programming, although the following directives are not permitted:
  - INVOKE, ADDR, .model, .386, .stack
  - (Other non-permitted directives will be introduced in later chapters)

64-Bit Version of AddTwoSum

```
1: ; AddTwoSum_64.asm - Chapter 3 example.
3: ExitProcess PROTO
5: .data
6: sum DWORD 0
8: .code
9: main  PROC
10:   mov  eax,5
11:   add  eax,6
12:   mov  sum,eax
13:
14:   mov  ecx,0
15:   call ExitProcess
16: main ENDP
17: END
```


Things to Notice About the Previous Slide

- The following lines are not needed:

```
.386
.model flat,stdcall
.stack 4096
```

- INVOKE is not supported.
- CALL instruction cannot receive arguments
- Use 64-bit registers when possible

Summary

- Integer expression, character constant
- directive – interpreted by the assembler
- instruction – executes at runtime
- code, data, and stack segments
- source, listing, object, map, executable files
- Data definition directives:
  - BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, QWORD, TBYTE, REAL4, REAL8, and REAL10
  - DUP operator, location counter ($)
- Symbolic constant
  - EQU and TEXTEQU

4C 61 46 69 6E