Chapter Overview

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

CODE:

**MOV AX, (value)** ; Moves (value) into AX

(value) can be ...

- **Immediate Value**
  - Integer Constants ( Ex. MOV AL, 10h )
  - Integer Expressions ( Ex. MOV AL, 10 * 20 )
  - Characters ( Ex. MOV AL, "A" )
  - Some Reserved Words ( Ex. MOV AX, @data )

- **Register**
  - (Ex. MOV AX, BX)

- **Memory Location**
  - (Ex. MOV AX, Word1)

DATA:

**Word1 WORD (value)** ; Declares a word in memory

(value) can be most immediate values
Integer Constants

- Optional leading + or – sign
- binary, decimal, hexadecimal, or octal digits
- Common radix characters:
  - h – hexadecimal
  - d – decimal
  - b – binary
  - r – encoded real

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

Note: Large numbers should never have commas in them for formatting since MASM will interpret the comma as starting a new value.
Integer Expressions

- Operators and precedence levels:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Precedence Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>parentheses</td>
<td>1</td>
</tr>
<tr>
<td>+, -</td>
<td>unary plus, minus</td>
<td>2</td>
</tr>
<tr>
<td>*, /</td>
<td>multiply, divide</td>
<td>3</td>
</tr>
<tr>
<td>MOD</td>
<td>modulus</td>
<td>3</td>
</tr>
<tr>
<td>+, -</td>
<td>add, subtract</td>
<td>4</td>
</tr>
</tbody>
</table>

- Examples:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 / 5</td>
<td>3</td>
</tr>
<tr>
<td>-(3 + 4) * (6 - 1)</td>
<td>-35</td>
</tr>
<tr>
<td>-3 + 4 * 6 - 1</td>
<td>20</td>
</tr>
<tr>
<td>25 mod 3</td>
<td>1</td>
</tr>
</tbody>
</table>

Character and String Constants

- Enclose characters 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 and Identifiers

- **Reserved words cannot be used as identifiers**
  - Instruction mnemonics, directives, type attributes, operators, predefined symbols
  - See MASM reference in Appendix A for complete list

- **Identifiers**
  - Can not be a reserved word
  - 1-247 characters, including digits
  - not case sensitive
  - first character must be a letter, _, @, ?, or $

*Note:* @ is used extensively for predefined symbols, avoid using @ as the first symbol of an identifier.
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, TASM, and MASM32 are not the same as MASM

- **Example Directives:**
  
  .data   .code   .stack
Instructions

- Assembled into machine code by assembler
- Executed at runtime by the CPU
- An instruction contains:
  - Label: (optional)
  - Mnemonic: (required)
  - Operand: (depends on the instruction)
  - Comment: (optional)
- Syntax
  
  \[
  [\text{Label:}] \ \text{Mnemonic} \ \text{[Operand(s)]} \ \{; \ \text{Comment}\}
  \]
Labels

- **Act as place markers**
  - marks the address (offset) of code and data
- **Same rules for use as identifiers**
- **Data label**
  - must be unique
  - example: `myArray BYTE 00h` (not followed by colon)
- **Code label**
  - must be unique within the same procedure
  - target of jump and loop instructions
  - example: `L1: MOV AX,0000h` (followed by colon)
Mnemonics and Operands

- **Instruction Mnemonics**
  - examples: MOV, ADD, SUB, MUL, INC, DEC

- **Operands**
  - Instructions may have no operands, or up to three operands
  - Types of Operands include:
    - Constant, constant expression, register, memory (data label), code label, etc...

Constants and constant expressions are often called immediate values
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

*Note*: When there are two operands the first one is the destination
Comments

• **Comments are good!**
  • explain the program's purpose
  • when it was written, and by whom
  • revision information
  • explains tricky coding techniques
  • help you remember what you were thinking

• **Single-line comments**
  • begin with semicolon (;)

• **Multi-line comments**
  • begin with COMMENT directive and a programmer-chosen character, and end with the same programmer-chosen character

```plaintext
COMMENT
 !
This line is a comment
This line is also a comment
!
```
Suggested Coding Standards

- **Some approaches to capitalization**
  - Lowercase for keywords, mixed for identifiers, and capitalize constants (like C, C++, Java)
  - Capitalize everything (old school)
  - Capitalize reserved words, including instructions and registers
  - Capitalize only directives and operators, mixed for identifiers (book)

- **Other suggestions**
  - Descriptive identifier names
  - Pick one format and use consistently
  - Spaces surrounding arithmetic operators

- **Indentation and spacing**
  - Executable instructions – indent at least 4 spaces (room for labels)
  - Comments: begin at column 40-45, aligned vertically
  - 1-3 spaces between instruction and its operands
  - 1-2 blank lines or comment lines between procedures
What's Next

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

```
Source File
   ↓
Step 1: text editor

Step 2: assembler

Link Library

Object File
   ↓

Step 3: linker

Listing File

Executable File
   ↓

Map File

Step 4: OS loader

Output
```

Web site  Examples
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)
• Example: addSub.lst (C:\Irvine\Examples\Ch03\addSub.lst)
Map File

- **Information about each program segment:**
  - starting address
  - ending address
  - size
  - segment type
- **Example:** `addSub.map` (16-bit version)

<table>
<thead>
<tr>
<th>Start</th>
<th>Stop</th>
<th>Length</th>
<th>Name</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000H</td>
<td>006E2H</td>
<td>006E3H</td>
<td>TEXT</td>
<td>CODE</td>
</tr>
<tr>
<td>006E4H</td>
<td>008FDH</td>
<td>0021AH</td>
<td>DATA</td>
<td>DATA</td>
</tr>
<tr>
<td>00900H</td>
<td>028FFH</td>
<td>02000H</td>
<td>STACK</td>
<td>STACK</td>
</tr>
<tr>
<td>02900H</td>
<td>02AFFH</td>
<td>00200H</td>
<td>BSS</td>
<td>BSS</td>
</tr>
</tbody>
</table>

Origin Group

005E:0 DGROUP

Program entry point at 0000:0000
What's Next

- Basic Elements of Assembly Language
- Assembling, Linking, and Running Programs
- Defining Data
- Symbolic Constants
- Example: Adding and Subtracting Integers
- 64-bit Programming
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:
  \[ \text{name} \ \text{directive} \ \text{initializer} \ \text{initializer} \ldots \]
  \[
  \text{value1} \ \text{BYTE} \ 10
  \]
- All initializers become binary data in memory
- Additional initializers can be added by separating with commas.
Intrinsic Data Types

- **BYTE, SBYTE, DB**
  - 8-bit unsigned integer; 8-bit signed integer
- **WORD, SWORD, DW**
  - 16-bit unsigned & signed integer
- **DWORD, SDWORD, DD**
  - 32-bit unsigned & signed integer
- **QWORD, DQ**
  - 64-bit integer
- **TBYTE, DT**
  - 80-bit integer
- **REAL4**
  - 4-byte IEEE short real number
- **REAL8**
  - 8-byte IEEE long real number
- **REAL10**
  - 10-byte IEEE extended real number
### Defining BYTE and SBYTE Data

Each of the following defines a single byte of storage:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value1</td>
<td>BYTE 'A'</td>
</tr>
<tr>
<td>value2</td>
<td>BYTE 0</td>
</tr>
<tr>
<td>value3</td>
<td>BYTE 255</td>
</tr>
<tr>
<td>value4</td>
<td>SBYTE -128</td>
</tr>
<tr>
<td>value5</td>
<td>SBYTE +127</td>
</tr>
<tr>
<td>value6</td>
<td>BYTE ?</td>
</tr>
</tbody>
</table>

- MASM does not prevent you from initializing a BYTE with a negative value, but it's considered poor style.

---


Web site  Examples
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
```

**Notice:** Hexadecimal values beginning with a letter must have a proceeding 0.

```
Memory holds:
10-20-30-40-10-20-30-40-50-60-70-80-81-82-83-84-
    list1    list2
    ↑         ↑
    ↑         ↑
    ↑         ↑
    ↑         ↑
list3
```

Defining Strings

- A string is implemented as an array of characters
  - For convenience, it is usually enclosed in quotation marks
  - It often will be null-terminated

```assembly
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 ",0
    BYTE "created by Kip Irvine.",0
```

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

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

0Dh = Carriage Return
0Ah = Line Feed
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 ; 8 bytes
```
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
  ```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000:</td>
<td>78</td>
</tr>
<tr>
<td>0001:</td>
<td>56</td>
</tr>
<tr>
<td>0002:</td>
<td>34</td>
</tr>
<tr>
<td>0003:</td>
<td>12</td>
</tr>
</tbody>
</table>

  ```
  val1 BYTE 01h,02h,03h ; bytes
  val2 WORD 1234h,5678h ; words
  val3 DWORD 12345678h ; doublewords
  ```

  Memory holds: 01-02-03-34-12-78-56-78-56-34-12
Defining WORD and SWORD Data

- Define storage for 16-bit integers
  - The size of two bytes or characters
  - Signed (SWORD) or Unsigned (WORD)
  - single value or multiple values

```
word1   WORD   65535 ; largest unsigned value
word2   SWORD  -32768 ; smallest signed value
word3   WORD    ? ; uninitialized, unsigned
myList  WORD  1,2,3,4,5 ; array of words
array   WORD   5 DUP(?) ; uninitialized array
```
Defining DWORD and SDWORD Data

- Define storage for 32-bit integers
  - The size of four bytes or two words
  - Signed (SDWORD) or unsigned (DWORD)
  - Single value or multiple values may be declared

```assembly
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

- Define storage for larger numbers
  - Quadword (QWORD) (64-bit: 8 bytes or 4 words)
  - Tenbyte (TBYTE) (80-bit: 10 bytes or 5 words)
  - Real (REAL4: 4 byte | REAL8: 8 byte | REAL10: 10 byte)
  - Single value or multiple values may be declared

```
quad1 QWORD 1234567812345678h
val1 TBYTE 10000000000123456789Ah
rVal1 REAL4 -2.1
rVal2 REAL8 3.2E-260
rVal3 REAL10 4.6E+4096
ShortArray REAL4 20 DUP(0.0)
```
Mixing Data and Code

- The assembler lets you switch back and forth between data and code:

```
    .code
    mov eax, ebx
    .data
    temp DWORD ?
    .code
    mov temp, eax
```

**Note:** Avoid doing this in your programs. It often makes your code harder to read, and opens up more opportunities for mistakes.
What's Next

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

- Used to associate a common name (identifier) with an integer expression or text.
- Reserves no space
- When compiled all occurrences are replaced by the number or text.

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 = 50
.
.
mov al,COUNT
```
Calculating the Size of a Byte Array
$ - \text{Current location counter}$
Subtract address of list: difference is the number of bytes

```
list BYTE 1,2,3,4
ListSize = 4
```

```
list BYTE 10,20,30,40
ListSize = ($ - \text{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 = ($ - \text{list}) / 2
```

Calculating the Size of a Doubleword Array
Divide total number of bytes by 4 (the size of a doubleword)
EQU Directive

- Define a symbol as either an integer or text expression.
- When compiled all occurrences are replaced by the number or text.
- Cannot be redefined

```
COUNTER EQU 25
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

```assembly
rowSize = 5

.data
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
• Assembling, Linking, and Running Programs
• Defining Data
• Symbolic Constants
• **Example: Adding and Subtracting Integers**
• 64-bit Programming
Program Template

TITLE Program Template (Template.asm)

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

INCLUDE Irvine32.inc
.data
    ; (insert variables here)
.codes
main PROC
    ; (insert executable instructions here)
    exit
main ENDP
    ; (insert additional procedures here)
END main
Example: Adding and Subtracting Integers

TITLE Add and Subtract (AddSub.asm)
; This program 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

Program output, showing registers and flags:

| EAX=00030000 | EBX=7FFDF000 | ECX=00000101 | EDX=FFFFFFFF |
| ESI=00000000 | EDI=00000000 | EBP=0012FF00 | ESP=0012FFC4 |
| EIP=00401024 | EFL=00000206 | CF=0 | SF=0 | ZF=0 | OF=0 |

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 DW 10000h
val2 DW 40000h
val3 DW 20000h
finalVal DW ?

.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
What's Next

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

- Requires 64-bit version of MASM (MASM 11.0 in VS12)
- Statements Using PROTO do not have parameters in 64-bit.
- 64-bit version of MASM does not support INVOKE

```assembly
ExitProcess PROTO

.data
val1 QWORD 10000h
val2 QWORD 40000h
val3 QWORD 20000h
finalVal QWORD ?

.code
main PROC
    mov rax,val1 ; start with 10000h
    add rax,val2 ; add 40000h
    sub rax,val3 ; subtract 20000h
    mov finalVal,rax ; store the result (30000h)
    call ExitProcess
main ENDP
END main
```