**`Notes on Assembly Language Fundamentals (Based on Irvine's Book)  
Chapter 3 Complete**

**1.1 Basic Language Elements**

**1.1.1 First Assembly Language Program**

* Assembly language provides near-total control over CPU operations.
* Requires explicit management of data representation and instruction formats.
* A simple program, **AddTwo**, adds two numbers and stores the result in a register.

**Example Code:**

main PROC

mov eax, 5 ; Move 5 into EAX register

add eax, 6 ; Add 6 to the value in EAX (result: 11)

INVOKE ExitProcess, 0 ; End the program

main ENDP

* **Explanation of Code:**
  + PROC and ENDP define the procedure.
  + mov instruction loads a value into a register.
  + add instruction performs addition.
  + INVOKE ExitProcess, 0 calls a Windows service to terminate the program.
  + Comments begin with **(;)**
* The program can be executed using a **debugger** to inspect register values.

# **1.2 Adding a Variable**

* The result of an operation can be stored in a **variable**.
* Requires **data and code sections**.
* Those code and data areas we mentioned, which were marked by the .code and .data directives, are called segments.

**Integer Literals in Assembly Language (MASM)**

**Definition**

An **integer literal** (or **integer constant**) consists of:

* An **optional** + or - sign
* One or more **digits**
* An **optional radix character** to specify the base

**Syntax:**  
[{+ | - }] digits [ radix ]

**Radix Characters (Number Bases)**

Numbers in assembly can be written in different bases using **radix characters**:

| **Radix** | **Character** | **Example** |
| --- | --- | --- |
| **Decimal** | d (optional) | 26 or 26d |
| **Hexadecimal** | h | 1Ah, 0A3h (Leading 0 is required if the first digit is a letter) |
| **Octal** | q or o | 42q or 42o |
| **Binary** | b or y | 1101b or 1101y |

**Examples**

26 ; Decimal (default)

26d ; Decimal

11010011b ; Binary

42q ; Octal

42o ; Octal

1Ah ; Hexadecimal

0A3h ; Hexadecimal (Leading 0 needed)

**Key Points**

✅ If no radix is specified, the number is **assumed to be decimal**.  
✅ **Hexadecimal values starting with a letter must have a leading zero** (e.g., 0A3h).  
✅ Assembly language follows **Microsoft syntax notation**, where:

* **Square brackets** [ ] indicate optional elements.
* **Curly braces** { } indicate a required choice.
* **Italics** indicate defined items.

This ensures that integer literals are **correctly interpreted** by the assembler! 🚀

### ****Constant Integer Expressions in Assembly Language****

#### ****Definition****

A **constant integer expression** is a **mathematical expression** that:  
✅ Uses **integer literals** and **arithmetic operators**  
✅ Must **evaluate to a 32-bit integer** (0 to FFFFFFFFh)  
✅ Is **evaluated at assembly time**

#### ****Operator Precedence (Order of Operations)****

Operators are applied in the following order (highest to lowest precedence):

| **Precedence Level** | **Operator** | **Description** |
| --- | --- | --- |
| **1 (Highest)** | ( ) | **Parentheses** (force priority) |
| **2** | +, - (unary) | **Unary plus, minus** |
| **3** | \*, /, MOD | **Multiply, divide, modulus** |
| **4 (Lowest)** | +, - (binary) | **Add, subtract** |

#### ****Example Expressions and Their Evaluations****

**4 + 5 \* 2 ; 5 \* 2 = 10, then 4 + 10 = 14**

**12 - 1 MOD 5 ; 1 MOD 5 = 1, then 12 - 1 = 11**

**-5 + 2 ; -5 + 2 = -3**

**(4 + 2) \* 6 ; (4 + 2) = 6, then 6 \* 6 = 36**

**16 / 5 ; 16 / 5 = 3 (integer division)**

**(3 \* 4) \* (6 - 1) ; 3 \* 4 = 12, 6 - 1 = 5, then 12 \* 5 = 60**

**25 MOD 3 ; 25 MOD 3 = 1**

#### ****Key Points****

✅ **Expressions are evaluated during assembly time** (not runtime).  
✅ **Use parentheses ( )** to **clarify** the order of operations.  
✅ **Modulus (MOD) returns the remainder** of division.  
✅ **Integer division (/) drops the decimal part** (e.g., 16 / 5 = 3).

🔹 **Tip:** Always use **parentheses** to avoid mistakes in complex expressions! 🚀

### ****Real Number Literals in Assembly Language****

#### ****Definition****

A **real number literal** (floating-point literal) can be represented in:  
✅ **Decimal format**  
✅ **Encoded (hexadecimal) format** (IEEE floating-point representation)

#### ****Decimal Real Format****

A decimal real number consists of:

* **Optional sign** (+ or -)
* **Integer part**
* **Decimal point** (.)
* **Optional fractional part**
* **Optional exponent** (E followed by an integer)

**Syntax:**  
[sign] integer . [integer] [E[{+,-}] integer]

#### ****Examples of Decimal Reals****

**2. ; Valid real number**

**+3.0 ; Valid with explicit sign**

**-44.2E+05 ; Exponent notation**

**26.E5 ; Integer with exponent**

✅ **At least one digit and a decimal point are required.**

#### ****Encoded (Hexadecimal) Real Format****

* Real numbers can be stored in **IEEE floating-point format**.
* Example: **+1.0 in binary**

0011 1111 1000 0000 0000 0000 0000 0000

* In assembly language, this is written as:

3F800000r

#### ****Key Points****

✅ **Decimal real numbers** are more common in early learning.  
✅ **Hexadecimal (encoded) real numbers** follow IEEE floating-point format.  
✅ **Most x86 instructions focus on integer processing**, but real-number operations are covered in **Chapter 12**.  
✅ **Floating-point arithmetic is complex and technical**, but very important for advanced computations!

### ****Character and String Literals in Assembly Language:****

#### ****3.1.5 Character Literals****

* A **character literal** is a single character enclosed in **single (') or double (") quotes**.
* The assembler stores the **ASCII code** of the character in memory.

**Examples:**

'A' ; ASCII code 65 (41h)

"d" ; ASCII code 100 (64h)

✅ **Character literals are stored as integers** using **ASCII encoding**.  
✅ Example: 'A' is stored as **65 (41h in hexadecimal)**.

#### ****3.1.6 String Literals****

* A **string literal** is a sequence of characters (including spaces) enclosed in **single or double quotes**.

**Examples:**

'ABC' ; Stored as 41h, 42h, 43h

'X' ; Stored as 58h

"Good night, Gracie" ; Stored as ASCII bytes

'4096' ; Stored as ASCII bytes, not a number

✅ **Strings are stored as sequences of ASCII byte values**.  
✅ Example: "ABCD" is stored as:

41h, 42h, 43h, 44h

#### ****Using Embedded Quotes****

* You can **include quotes inside a string** by alternating between **single (') and double (") quotes**.

**Examples:**

"This isn't a test" ; Using double quotes around a single quote

'Say "Good night," Gracie' ; Using single quotes around double quotes

📌 **Key Takeaways:**  
✅ **Character literals → Stored as ASCII integer values**  
✅ **String literals → Stored as a sequence of ASCII byte values**  
✅ **Use embedded quotes properly** to avoid syntax errors. 🚀

### ****Reserved Words & Identifiers in Assembly Language****

#### ****3.1.7 Reserved Words****

* **Reserved words** have special meanings and can **only be used in their correct context**.
* **Not case-sensitive** by default (MOV, mov, and Mov are the same).

📌 **Types of Reserved Words:**

1. **Instruction mnemonics** → MOV, ADD, MUL
2. **Register names** → AX, BX, CX
3. **Directives** → Tell the assembler how to compile the program
4. **Attributes** → Define size & usage (BYTE, WORD)
5. **Operators** → Used in constant expressions
6. **Predefined symbols** → Example: @data (returns a constant at assembly time)

✅ A full list of reserved words is available in **Appendix A**.

#### ****3.1.8 Identifiers****

* **Identifiers** are **user-defined names** (for variables, constants, procedures, or labels).
* **Rules for identifiers:**  
  ✅ **Length:** 1 to **247** characters  
  ✅ **Not case-sensitive** (e.g., MyLabel and mylabel are the same)  
  ✅ **First character** must be **a letter, underscore \_, @, ?, or $**  
  ✅ **Can’t use reserved words** as identifiers

**Example of Valid Identifiers:**

myVar ; Valid

\_count ; Valid

Label1 ; Valid

@data ; Invalid (reserved word)

MOV ; Invalid (reserved word)

💡 **Tip:** To make keywords and identifiers **case-sensitive**, use the -Cp command-line switch. 🚀

## **3.1.7 Reserved Words**

* Reserved words have special meanings and can only be used in the correct context.
* They are **not case-sensitive** (e.g., MOV, mov, and Mov are equivalent).
* Types of reserved words:
  + **Instruction mnemonics**: MOV, ADD, MUL, etc.
  + **Register names**
  + **Directives**: Guide the assembler in assembling the program.
  + **Attributes**: Define size and usage of variables (e.g., BYTE, WORD).
  + **Operators**: Used in expressions.
  + **Predefined symbols**: Constants available during assembly time (e.g., @data).
* A list of reserved words is available in Appendix A.

## **3.1.8 Identifiers**

* **Programmer-chosen names** for variables, constants, procedures, or labels.
* **Rules for identifiers**:
  + Can have **1 to 247 characters**.
  + **Not case-sensitive**.
  + Must **start with a letter (\_ , @, ?, or $ allowed)**; subsequent characters can include digits.
  + Cannot be a **reserved word**.
* **Tip**: To make identifiers case-sensitive, use the -Cp command-line switch.

## **3.1.9 Directives**

* **Directives are commands** embedded in source code that instruct the assembler.
* **Do not execute at runtime** but help define variables, macros, and segments.
* **Not case-sensitive** (e.g., .data, .DATA, .Data are the same).
* **Example**:

myVar DWORD 26

mov eax, myVar

* **Segment Directives**:
  + .DATA: Defines a data segment (stores variables).
  + .CODE: Defines a code segment (stores instructions).
  + .STACK: Defines stack segment and sets size.

.stack 100h

* Different assemblers may have unique directives.

## **3.1.10 Instructions**

* **Executable statements** that the assembler converts into machine language.
* **Parts of an instruction**:
  + **Label (optional)**
  + **Instruction mnemonic (required)**
  + **Operands (usually required)**
  + **Comment (optional)**
* **Syntax**:

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

### ****Labels****

* **Identifiers for data or code locations**.
* **Two types**:
  + **Data Labels**: Define variable locations.

count DWORD 100

* + **Code Labels**: Mark instruction locations, used in jumps/loops.

target:

mov ax, bx

jmp target

### ****Instruction Mnemonics****

* **Short words representing CPU instructions**.
* **Common mnemonics**:

| **Mnemonic** | **Description** |
| --- | --- |
| MOV | Move (assign) one value to another |
| ADD | Add two values |
| SUB | Subtract one value from another |
| MUL | Multiply two values |
| JMP | Jump to a new location |
| CALL | Call a procedure |

### ****Operands****

* **Values used in instructions** (registers, memory locations, constants).
* **Example operand types**:

| **Example** | **Operand Type** |
| --- | --- |
| 96 | Integer literal |
| 2 + 4 | Integer expression |
| eax | Register |
| count | Memory |

* **Examples of instructions with different operands**:

stc ; No operands (sets carry flag)

inc eax ; One operand (increments EAX)

mov count, ebx ; Two operands (copies EBX to count)

imul eax, ebx, 5 ; Three operands (EBX \* 5 → EAX)

### ****Comments****

* **Used to describe code for better understanding**.
* **Two types**:
  + **Single-line comments**: Begin with ;

**mov EAX, 10; Load 10 into EAX**

* + **Block comments**: Use COMMENT directive with a user-defined symbol.

**COMMENT!**

**This is a block comment.**

**It can span multiple lines.**

**!**

### ****The NOP (No Operation) Instruction****

* **NOP (No Operation)** does nothing but occupies **1 byte** in memory.
* **Usage of NOP:**
  + Helps align code to **doubleword boundaries** for efficient execution.
  + Used by compilers and assemblers for padding.
* **Example:**

mov ax, bx

nop ; Align next instruction

mov edx, ecx

* x86 processors load instructions faster from **aligned memory addresses**.

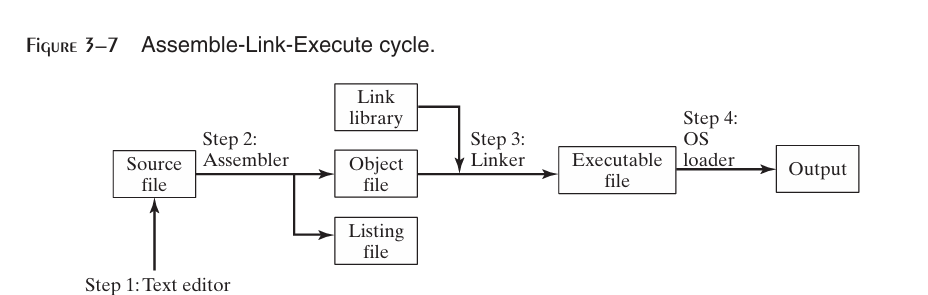
### Notes on Assembly Process and Listing File(Imp LQ)

**3.3.1 The Assemble-Link-Execute Cycle:**

1. **Step 1 (Editing)**: A programmer writes the source code using a text editor, creating an ASCII text file (source file).
2. **Step 2 (Assembling)**: The assembler reads the source file and produces an object file (machine code). If there are errors, the programmer fixes them and returns to Step 1.
3. **Step 3 (Linking)**: The linker reads the object file, checks for required external procedures, and combines them to create the executable file.
4. **Step 4 (Executing)**: The operating system loader loads the executable file into memory, and the program starts execution.

**3.3.2 Listing File:**

* **What is a listing file?**: A listing file contains:
  + Source code with line numbers.
  + Machine code in hexadecimal.
  + The numeric address of each instruction.
  + A symbol table that contains information about program identifiers and segments.
* **Example from AddTwo program**:
  + The listing file shows each instruction’s address and corresponding machine code.
  + It also includes an opcode (e.g., B8 for the MOV instruction) that represents the machine instruction.
  + In the example, the INVOKE ExitProcess line generates the PUSH and CALL instructions.



**Additional Notes:**

* **Listing File Usage**: Helps programmers and advanced users understand the machine code generated by the assembler. It is useful for debugging and learning how assembly instructions are translated into machine code.

This process is essential for converting high-level assembly code into executable machine code that the operating system can run. The listing file serves as a detailed breakdown of this translation, helping to understand how each part of the code is transformed.

### Notes on Data Definition in Assembly Language

#### 3.4.2 Data Definition Statement

* **Purpose**: A data definition statement allocates storage in memory for a variable, optionally assigning it a name.
* **Syntax**: [name] directive initializer [, initializer]...
* **Intrinsic Data Types**:
  + **BYTE**: 8-bit unsigned integer
  + **SBYTE**: 8-bit signed integer
  + **WORD**: 16-bit unsigned integer
  + **SWORD**: 16-bit signed integer
  + **DWORD**: 32-bit unsigned integer
  + **SDWORD**: 32-bit signed integer
  + **FWORD**: 48-bit integer
  + **QWORD**: 64-bit integer
  + **TBYTE**: 80-bit (10-byte) integer
  + **REAL4**: 32-bit IEEE short real
  + **REAL8**: 64-bit IEEE long real
  + **REAL10**: 80-bit IEEE extended real

#### Legacy Data Definition Directives

* **DB**: 8-bit integer
* **DW**: 16-bit integer
* **DD**: 32-bit integer or real
* **DQ**: 64-bit integer or real
* **DT**: 80-bit (10-byte) integer

#### Data Definition Rules

* **Initializer**: Every data definition requires at least one initializer, which can be a literal or expression.
* **Special Initializer**: The ? symbol can be used to leave a variable uninitialized.
* **Multiple Initializers**: Can be separated by commas. For example:
  + list BYTE 10, 20, 30, 40 defines multiple bytes.

#### 3.4.3 Example: Adding a Variable to the AddTwo Program

* **Example Program (AddTwoSum.asm)**:
  + A variable sum is defined in the .data section: sum DWORD 0.
  + The program adds 5 and 6, stores the result in sum, and calls ExitProcess.

#### 3.4.4 Defining BYTE and SBYTE Data

* **BYTE**: Defines an unsigned 8-bit value.
* **SBYTE**: Defines a signed 8-bit value.
* **Examples**:
  + value1 BYTE 'A' (character literal)
  + value2 BYTE 0 (smallest unsigned byte)
  + value3 BYTE 255 (largest unsigned byte)
  + value4 SBYTE -128 (smallest signed byte)
  + value5 SBYTE +127 (largest signed byte)

#### Defining Arrays and Multiple Initializers

* Data definitions can be used to define arrays or sequences of bytes:
  + Example: list BYTE 10, 20, 30, 40 assigns these values in consecutive memory slots.

#### Defining Strings

* **Null-Terminated Strings**: Enclose characters in single or double quotes to define strings.
  + **Example**: greeting1 BYTE "Good afternoon", 0
* **String Continuation**: Use the line continuation character to break strings over multiple lines.

#### Memory Layout and Offsets

* In an array, each value is stored sequentially at different memory offsets:
  + **Example**: list BYTE 10, 20, 30, 40:
    - Offset 0000: 10
    - Offset 0001: 20
    - Offset 0002: 30
    - Offset 0003: 40

**43.5 Symbolic Constants**

**Definition:**

* A **symbolic constant** (or **symbol definition**) is an identifier (symbol) associated with an integer expression or text.
* **Symbols do not reserve storage**; they are used by the assembler during the preprocessing stage and cannot change at runtime.

**Differences Between Symbols and Variables:**

* **Symbols:**
  + Do not use storage.
  + Values do not change at runtime.
* **Variables:**
  + Use storage.
  + Values can change at runtime.

**3.5.1 Equal-Sign Directive**

**Syntax:**

name = expression

* **expression** is typically a 32-bit integer value.
* During assembly, all occurrences of name are replaced by expression.

**Example:**

COUNT = 500

mov eax, COUNT

* After assembly, the instruction becomes:

mov eax, 500

**Advantages of Using Symbols:**

* **Readability:** Programs are easier to read and maintain.
* **Maintainability:** Easily change the value of a symbol used multiple times in a program.

COUNT = 600

* + All instances of COUNT are automatically updated to 600 upon reassembly.

**Current Location Counter ($):**

* The $ symbol represents the **current location counter**.
* Example:

selfPtr DWORD $

* + selfPtr is initialized with the current offset value.

**Keyboard Definitions:**

* Symbols can represent keyboard codes for better readability.

Esc\_key = 27

mov al, Esc\_key ; Good style

mov al, 27 ; Poor style

**Using the DUP Operator:**

* Symbolic constants can be used with the DUP operator for array and string storage.

array dword COUNT DUP(0)

**Redefinitions:**

* Symbols defined with = can be redefined within the same program.

COUNT = 5

mov al, COUNT ; AL = 5

COUNT = 10

mov al, COUNT ; AL = 10

COUNT = 100

mov al, COUNT ; AL = 100

* The value of COUNT changes based on the assembler's sequential processing, not runtime execution.

**3.5.2 Calculating the Sizes of Arrays and Strings**

**Array Size Calculation:**

* Use the $ operator to calculate the size of an array or string.

list BYTE 10,20,30,40

ListSize = ($ - list)

* ListSize must follow immediately after list to avoid incorrect calculations.

**Example of Incorrect Calculation:**

list BYTE 10,20,30,40

var2 BYTE 20 DUP(?)

ListSize = ($ - list) ; Incorrect: ListSize = 24

**String Length Calculation:**

myString BYTE "This is a long string, containing"

BYTE "any number of characters"

myString\_len = ($ - myString)

**Arrays of Words and Doublewords:**

* For arrays of words (16-bit) or doublewords (32-bit), divide the total size by the size of each element.

list WORD 1000h,2000h,3000h,4000h

ListSize = ($ - list) / 2 ; Each word is 2 bytes

list DWORD 10000000h,20000000h,30000000h,40000000h

ListSize = ($ - list) / 4 ; Each doubleword is 4 bytes

**3.5.3 EQU Directive**

**Syntax:**

name EQU expression

name EQU symbol

name EQU <text>

* **expression:** A valid integer expression.
* **symbol:** An existing symbol name.
* **text:** Any text within <...>.

**Example:**

PI EQU <3.1416>

pressKey EQU <"Press any key to continue...",0>

.data

prompt BYTE pressKey

**Matrix Example:**

matrix1 EQU 10 \* 10

matrix2 EQU <10 \* 10>

.data

M1 WORD matrix1 ; M1 = 100

M2 WORD matrix2 ; M2 = 10 \* 10

**No Redefinition:**

* Symbols defined with EQU cannot be redefined in the same source file, preventing accidental value changes.

**3.5.4 TEXTEQU Directive**

**Syntax:**

name TEXTEQU <text>

name TEXTEQU textmacro

name TEXTEQU %constExpr

* **text:** Assigns text.
* **textmacro:** Assigns the contents of an existing text macro.
* **constExpr:** Assigns a constant integer expression.

**Example:**

continueMsg TEXTEQU <"Do you wish to continue (Y/N)?">

.data

prompt1 BYTE continueMsg

**Building Text Macros:**

rowSize = 5

count TEXTEQU %(rowSize \* 2)

move TEXTEQU <mov>

setupAL TEXTEQU <move al,count>

* The statement setupAL assembles to:

mov al, 10

**Redefinition:**

* Symbols defined by TEXTEQU can be redefined at any time.

**Summary**

* **Symbolic Constants:** Enhance readability and maintainability by associating identifiers with integer expressions or text.
* **Directives:**
  + **=:** Defines and redefines symbols with integer expressions.
  + **EQU:** Defines symbols with integer expressions, existing symbols, or text; cannot be redefined.
  + **TEXTEQU:** Defines text macros that can be redefined.
* **Current Location Counter ($):** Used to calculate offsets and sizes of arrays and strings.
* **Array and String Size Calculation:** Use $ to dynamically calculate sizes, ensuring accurate and maintainable code.

Review Ex

**Integer literal for -35 in MASM syntax:**

**Decimal:** -35

**Hexadecimal:** -23h

**Octal:** -43o

**Binary:** -100011b

**Is A5h a valid hexadecimal literal?** → **Yes**

**Does (\*) have a higher precedence than (/) in integer expressions?** → **No** (They have the same precedence and are evaluated left to right.)

**Expression using all operators from Section 3.1.2:**  
Example:

assembly

CopyEdit

mov eax, (10 + 5) \* 4 - (20 / 2) + (3 AND 1) OR (6 XOR 2)

**Write -6.2 × 10⁴ as a real number literal in MASM syntax:**

-6.2E4

**Must string literals be enclosed in single quotes?** → **No but also wth “ ”**

**Reserved words can be instruction mnemonics, attributes, operators, predefined symbols, and...** → **Directives**

**Maximum length of an identifier in MASM?** → **247 characters**

**Review**

1. **Types of files produced by the assembler:**
   * **Object files** (.obj or .o)
   * **Listing files** (.lst, optional)
   * **Symbol files** (for debugging, optional)
2. **The linker extracts assembled procedures from the link library and inserts them in the executable program.** → **True**
3. **When a program’s source code is modified, it must be assembled and linked again before it can be executed with the changes.** → **True**
4. **Which operating system component reads and executes programs?** → **The Loader (part of the OS kernel)**
5. **Types of files produced by the linker:**
   * **Executable files** (.exe or .out)
   * **Map files** (optional, for debugging)
   * **Dynamic link libraries (DLLs)** (.dll) in Windows or .so in Linux

4o