Microprocessor and Assembly Language CSC-321

Sheeza Zaheer

Lecturer

COMSATS UNIVERSITY ISLAMABAD
LAHORE CAMPUS

Introduction

OUTLINE

- **Introduction**
  - About this course
  - About Assembly Language
  - Syntax of Assembly Language
  - Basic Instructions
  - Variables
  - Translation
  - Program Structure
- **References**
  - **Chapter 4,** Ytha Yu and Charles Marut, “Assembly Language Programming and Organization of IBM PC

WHAT IS THIS COURSE ABOUT?

Course Objectives

- To understand organization of a computer system
  - To gain an insight knowledge about the internal architecture and working of microprocessors.
  - To understand working of memory devices, interrupt controllers and I/O devices.
- To learn Assembly Language
  - To understand how low-level logic is employed for problem solving by using assembly language as a tool.

Textbooks/Reference Books

- **TB-1**: "*Assembly Language Programming and Organization of the IBM PC*", Ytha Yu and Charles Marut, McGraw Hill
- **TB-2**: "*Computer Organization and Architecture*", 8th Edition, William Stallings, Prentice Hall 2002

- "*Assembly Language for Intel-based Computers*", 4th Edition, Irvine, Prentice Hall 2003
- "*Computer Organization*", Carl Hamacher & ZvonkoVranesic
- "*Computer Organization and Design: The Hardware Software Interface*", 2nd Edition, David A Patterson and john L Hennessy
- "*Assembly Language and Computer Architecture*", Anthony J. Dos Reis

*A*SSEMBLY *L*ANGUAGE

Computer Level Hierarchy

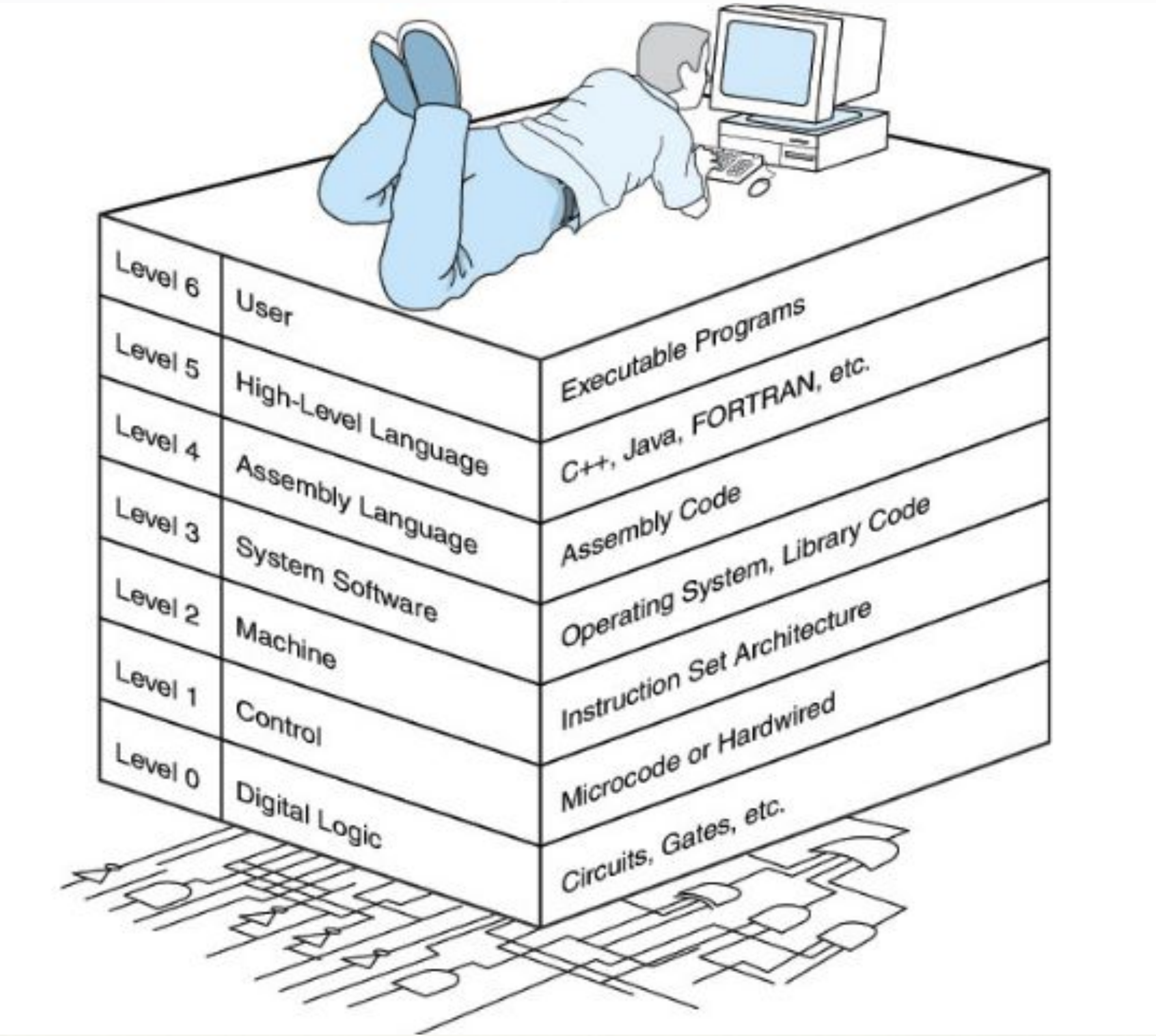


***Figure Reference**:*
http://users.dickinson.edu/~braught/courses/cs251f09/topics/slides/intro.pdf

Programming Languages

- High-Level Languages (HLL)
- Assembly Language
- Machine Language

High-Level Language

- Allow programmers to write programs that look more like natural language.
- Examples: C++, Java, C#.NET etc
- A program called **Compiler** is needed to translate a high-level language program into low-level language program

Machine Language

- The "native" language of the computer
- Numeric instructions and operands that can be stored in memory and are directly executed by computer system.
- Each ML instruction contains an *op code* (operation code) and zero or more operands.
- Examples:

```
OpcodeOperand      Meaning
-----------------------------------------------
40                 increment the AX register
05         0005          add 0005 to AX
```

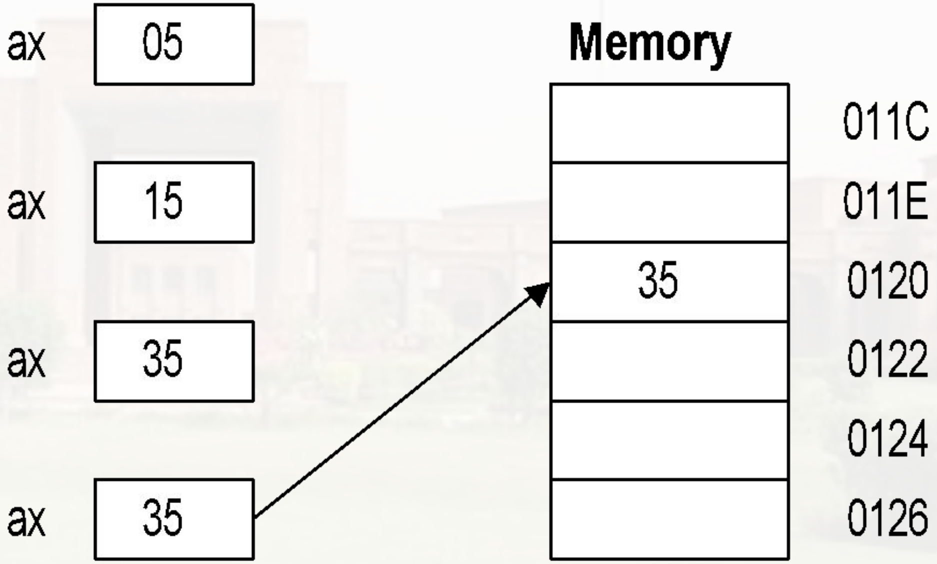
Assembly Language

- Use instruction mnemonics that have one-to-one correspondence with machine language.
- An ***instruction*** is a symbolic representation of a single machine instruction
- Consists of:

label	always optional
mnemonic	always required
operand(s)	required by some instructions
comment	always optional

Sample Program

1. mov ax, 5
2. add ax, 10
3. add ax, 20
4. mov [0120], ax
5. int 20



Essential Tools

- ***Assembler*** is a program that converts source-code programs into a machine language (*object file*).
- ***Linker*** joins two or more object files and produces a single executable file.
- ***Debugger*** loads an executable program, displays the source code, and lets the programmer step through the program one instruction at a time, and display and modify memory.
- ***Emulator*** allows you to load and run assembly language programs, examine and change contents of registers. Example: EMU8086

Why Learn Assembly Language?

- Learn how a processor works
  - Explore the internal representation of data and instructions
  - How to structure a program so it runs more efficiently. (High Level Language □ Low Level Language)

- Compilers/Device Drivers/ OS codes
- Games

BASIC ELEMENTS

Statements

- Syntax:

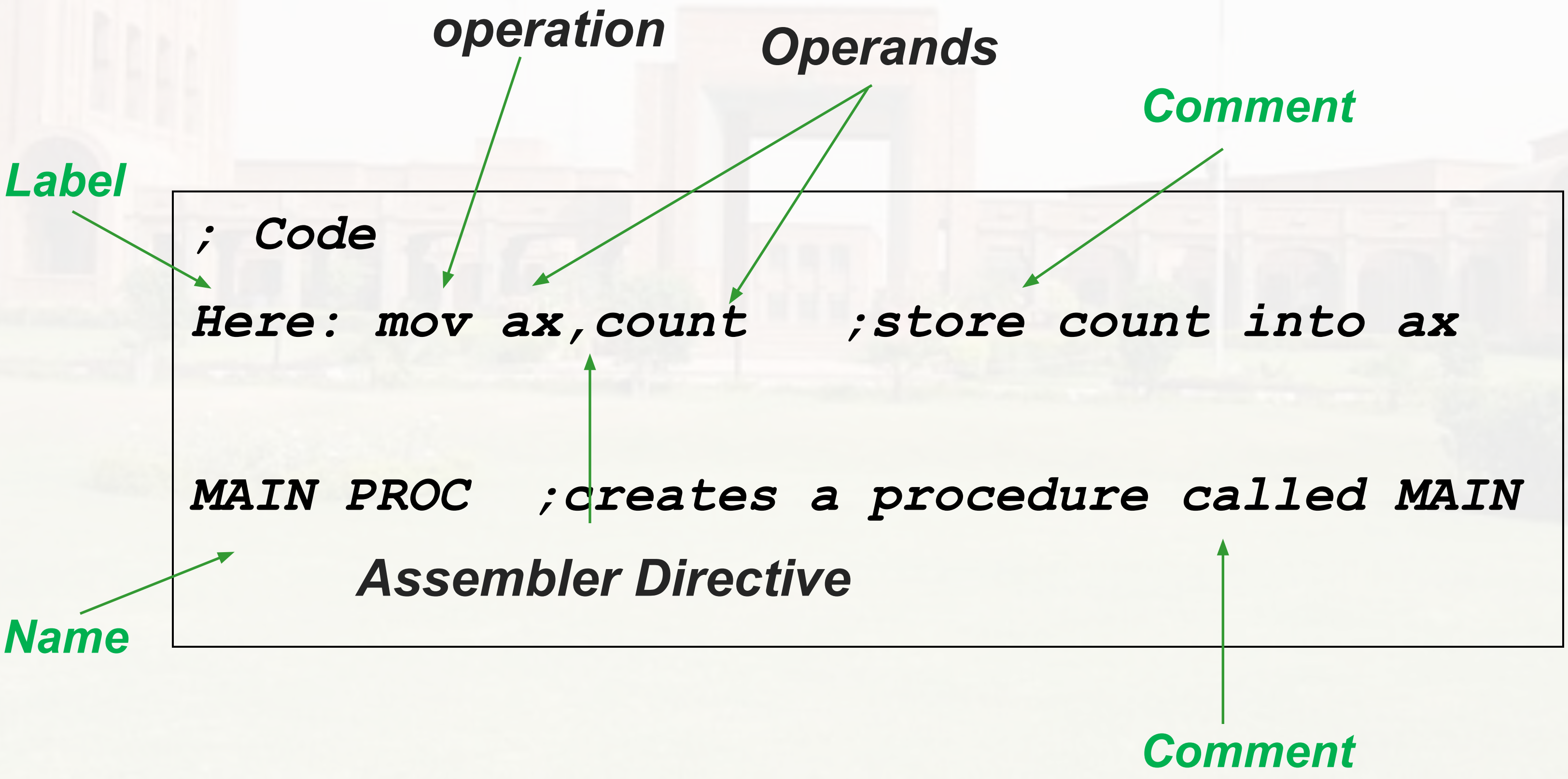
  name operation operand(s) comments

  - name and comment are optional
  - Number of operands depend on the instruction

- One statement per line
  - At least one blank or tab character must separate the field.

- Each statement is either:
  - Instruction (translated into machine code)
  - Assembler Directive (instructs the assembler to perform some specific task such as creating a procedure)

Statement Example



operation

Operands

Comment

Label

```
; Code
Here: mov ax,count    ;store count into ax

MAIN PROC  ;creates a procedure called MAIN
```

Assembler Directive

Name

Comment

Name/Label Field

- The assembler translates names into memory addresses.
- Names can be 1 to 31 character long and may consist of letter, digit or special characters ? . @ _ $ %. If period is used, it must be first character.
- Embedded blanks are not allowed.
- May not begin with a digit.
- Not case sensitive

Examples of legal names	**Examples of illegal names**
COUNTER_1	TWO WORDS
@character	2abc
.TEST	A45.28
DONE?	YOU&ME

Operation Field: Symbolic operation (Op code)

- Symbolic op code translated into Machine Language op code
- ***Examples***: ADD, MOV, SUB

- In an assembler directive, the operation field represents Pseudo-op code
- Pseudo-op is not translated into Machine Language op code, it only tells assembler to do something.
- ***Example***: **PROC** psuedo-op is used to create a procedure

Operand Field

- An instruction may have zero, one or more operands.
- In two-operand instruction, first operand is destination, second operand is source.
- ***Examples***

```
PUSHF                   ;no operand
INC AX                ;one operand, adds 1 to the contents of AX
 ADD AX, 2            ;two operands, adds value 2 to the contents of AX
```

Comments

- Optional
- Marked by semicolon in the beginning
- Ignored by assembler
- Good practice

```
;
;initialize registers
;
MOV AX, 0
MOV BX, 0
```

Program Data

- Processor operates only on binary data.
- In assembly language, you can express data in:
  - Binary
  - Decimal
  - Octal
  - Hexadecimal
  - Characters
- Numbers
  - For Hexadecimal, the number must begin with a decimal digit. E.g.: write 0ABCh not only ABCH.
  - Cannot contain any non-digit character. E.g.: 1,234 not allowed
- Characters enclosed in single or double quotes.
  - ASCII codes can be used
  - No difference in “A” and 41h

Contd..

- Use a radix symbol (suffix) to select binary, octal, decimal, or hexadecimal

```
6A15h        ; hexadecimal
0BAF1h       ; leading zero required
32o          ; octal
1011b        ; binary
35d          ; decimal (default)
```

Variables

- Each variable has a data type and is assigned a memory address by the program.
- Possible Values:
  - **8 Bit Number Range**: Signed (-128 to 127), Unsigned (0-255)
  - **16 Bit Number Range:** Signed (-32,678 to 32767), Unsigned (0-65,535)
  - **?** To leave variable uninitialized

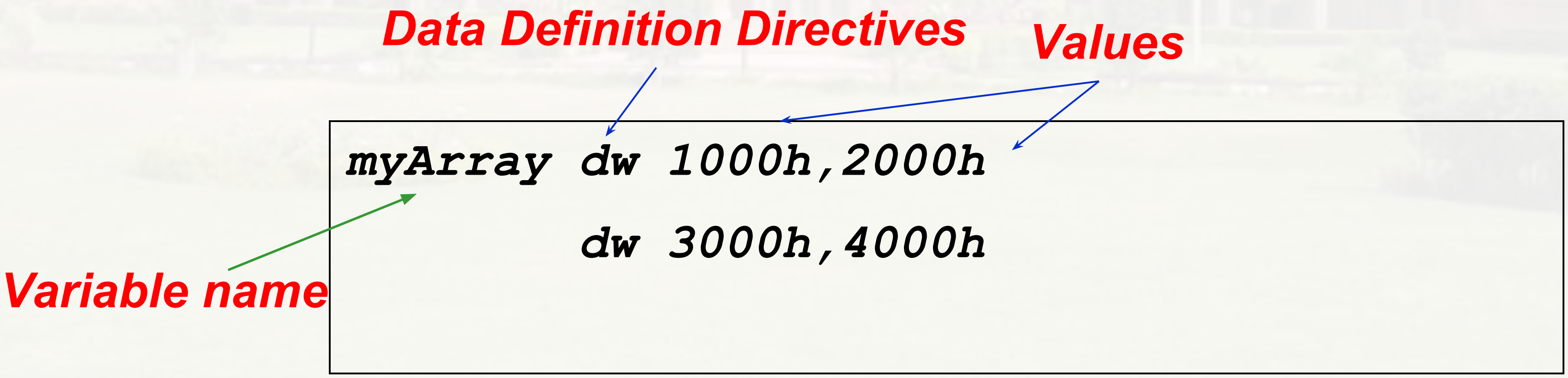
Contd..

- Syntax

  variable_name type initial_value

  variable_name type value1, value2, value3

- Data Definition Directives Or Data Defining Pseudo-ops
  - DB, DW, DD, DQ, DT



Data Definition Directives

Values

```
myArray dw 1000h,2000h
        dw 3000h,4000h
```

Variable name

***Remember**: you can skip variable name!*

Contd..

Examples	Bytes	Description	Pseudo-ops
var1 DB ‘A’ Var2 DB ? array1 DB 10, 20,30,40	1	Define Byte	**DB**
var2 DW ‘AB’ array2 DW 1000, 2000	2	Define Word	**DW**
Var3 DD -214743648	4	Define Double Word	**DD**

Note:

Consider

var2 DW 10h

Still in memory the value saved will be 0010h

Arrays

- Sequence of memory bytes or words
- **Example 1:**

  B_ARRAY DB 10h, 20h, 30h

Symbol	**Address**	**Contents**
B_ARRAY	0200h	10h
B_ARRAY+1	0201h	20h
B_ARRAY+2	0202h	30h

***If B_ARRAY is assigned offset address 0200h by assembler**

Example 2

- W_ ARRAY DW 1000, 40, 29887, 329

If W_ARRAY is assigned offset address 0300h by assembler

Symbol	Address	Contents
W_ARRAY	0300h	1000d
W_ARRAY+ 2	0302h	40d
W_ARRAY+ 4	0304h	29887d
W_ARRAY+ 6	0306h	329d

- ***High & Low Bytes of a Word***

  WORD1 DW 1234h

- *Low Byte = 34h, symbolic address is WORD1*
- *High Byte = 12h, symbolic address is WORD1+1*

Character String

LETTERS DB ‘ABC’

Is equivalent to

LETTERS DB 41h, 42h, 43h

- Assembler differentiates between upper case and lower case.
- Possible to combine characters and numbers.

MSG DB ‘HELLO’, 0Ah, 0Dh, ‘$’

Is equivalent to

MSG DB 48h, 45h, 4Ch, 4Ch, 4Fh, 0Ah, 0Dh, 24h

Example 3

- Show how character string “RG 2z” is stored in memory starting at address 0.
- Solution:

Address	Character	ASCII Code (HEX)	ASCII Code (Binary) [Memory Contents]
0	R	52	0101 0010
1	G	47	0100 0111
2	Space	20	0010 0000
3	2	32	0011 0010
4	z	7A	0111 1010

Named Constants

- Use symbolic name for a constant quantity
- **Syntax**:

  name **EQU** constant

- **Example**:

```
LF      EQU   0Ah
```

- No memory allocated

A FEW BASIC INSTRUCTIONS

MOV

- Transfer data
  - Between registers (mov ax, bx)
  - Between register and a memory location (mov ax, var1)
  - Move a no. directly to a register or a memory location (mov ax, 1234h)
- Syntax

  MOV *destination, source*
- Example

  MOV *AX, WORD1*

	Before	***After***
AX	*0006*	*0008*
WORD1	*0008*	*0008*

- **Difference?**
  - MOV AH, ‘A’
  - MOV AX, ‘A’

Legal Combinations of Operands for MOV

Destination Operand	Source Operand	Legal
General Register	General Register	YES
General Register	Memory Location	YES
General Register	Segment Register	YES
General Register	Constant	YES
Memory Location	General Register	YES
Memory Location	Memory Location	NO
Memory Location	Segment Register	YES
Memory Location	Constant	YES

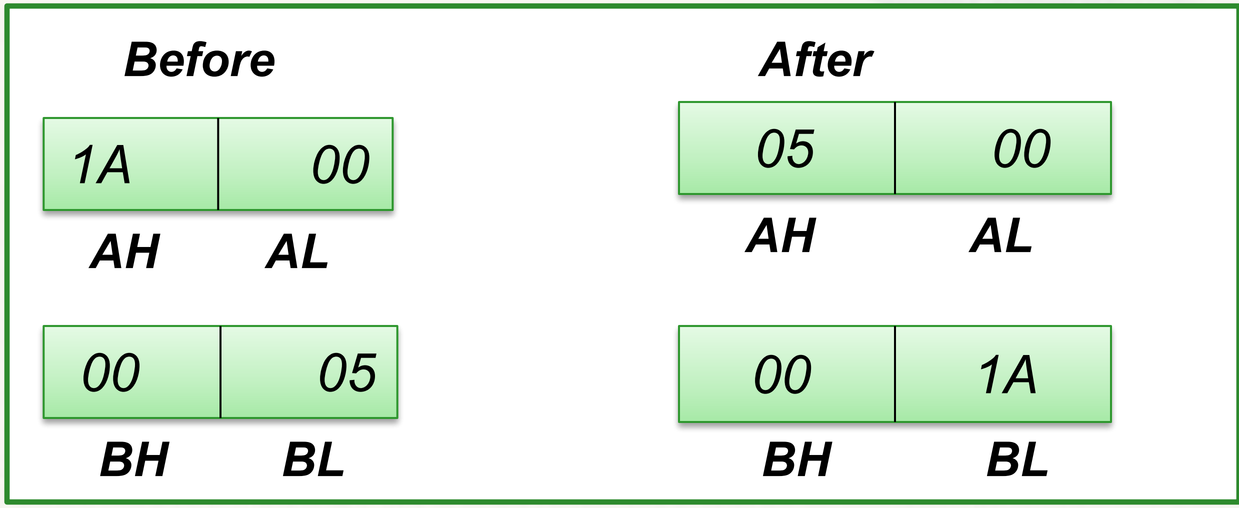
XCHG

- Exchange the contents of
  - Two registers (xchg ax, bx)
  - Register and a memory location (xchg ax, var1)
- Syntax

XCHG *destination, source*

- Example

XCHG *AH, BL*



Before

1A	00
AH	**AL**
00	05
BH	**BL**

After

05	00
AH	**AL**
00	1A
BH	**BL**

Legal Combinations of Operands for XCHG

Destination Operand	Source Operand	Legal
General Register	General Register	YES
General Register	Memory Location	YES
Memory Location	General Register	YES
Memory Location	Memory Location	NO

ADD Instruction

- To add contents of:
  - Two registers (Add ax, bx)
  - A register and a memory location (Add ax, var1)
  - A number to a register (Add ax, 1234h)
  - A number to a memory location (Add var1, 1234h)
- Syntax: ADD destination, source
- Example

  ADD WORD1, AX

	Before	***After***
AX	*01BC*	*01BC*
WORD1	*0523*	*06DF*

SUB Instruction

- To subtract the contents of:
  - Two registers (Sub ax, bx)
  - A register and a memory location (sub ax, var1)
  - A number from a register (sub ax, 1234h)
  - A number from a memory location (sub var1, 1234h)
- Syntax: SUB destination, source
- Example

  SUB AX, DX

	Before	After
AX	0000	FFFF
DX	0001	0001

Legal Combinations of Operands for ADD & SUB instructions

Destination Operand	Source Operand	Legal
General Register	General Register	YES
General Register	Memory Location	YES
General Register	Constant	YES
Memory Location	General Register	YES
Memory Location	Memory Location	NO
Memory Location	Constant	YES

Contd..

ADD BYTE1, BYTE2 ILLEGAL instruction

- Solution?

```
MOV AL, BYTE2
ADD BYTE1, AL
```

- **How can you add two word variables?**

INC & DEC

- **INC** (increment) instruction is used to add 1 to the contents of a register or memory location.
  - Syntax: INC *destination*
  - Example: INC WORD1

- **DEC** (decrement) instruction is used to subtract 1 from the contents of a register or memory location.
  - Syntax: DEC *destination*
  - Example: DEC BYTE1

- Destination can be 8-bit or 16-bits wide.
- Destination can be a register or a memory location.

Contd..

INC WORD1

	Before	After
WORD1	0002	0003

DEC BYTE1

	Before	After
BYTE1	FFFE	FFFD

NEG

- Used to negate the contents of destination.
- Replace the contents by its 2’s complement.
- Syntax

  NEG *destination*

- Example

  NEG BX

	Before	**After**
BX	0002	FFFE

How?

TRANSLATION

Examples

- Consider instructions: MOV, ADD, SUB, INC, DEC, NEG
- **A** and **B** are two-word variables
- Translate statements into assembly language:

Statement	**Translation**
B = A	MOV AX, A MOV B, AX
A = 5 - A	MOV AX, 5 SUB AX, A MOV AX, A **OR** NEG A ADD A, 5

Contd..

Statement	Translation
A = B – 2 x A	MOV AX, B SUB AX, A SUB AX, A MOV A, AX

- □ ***Remember:*** *Solution not unique!*
- □ ***Be careful!*** *Word variable or byte variable?*

PROGRAM STRUCTURE

Program Segments

- Machine Programs consists of
  - Code
  - Data
  - Stack
- Each part occupies a memory segment.
- Same organization is reflected in an assembly language program as **Program Segments**.

Memory Models

- Determines the size of data and code a program can have.
- Syntax:

  .MODEL memory_model

Model	Description
SMALL	code in one segment, data in one segment
MEDIUM	code in more than one segment, data in one segment
COMPACT	code in one segment, data in more than one segment
LARGE	Both code and data in more than one segments No array larger than 64KB
HUGE	Both code and data in more than one segments array may be larger than 64KB

Data Segment

- All variable definitions
- Use **.DATA** directive
- For Example:

```
 .DATA
 WORD1 DW 2
BYTE1 DB 10h
```

Stack Segment

- A block of memory to store stack
- Syntax

  .STACK size

  - Where size is optional and specifies the stack area size in bytes
  - If size is omitted, 1 KB set aside for stack area

- For example:

  .STACK 100h

Code Segment

- Contains a program’s instructions
- Syntax

.CODE name

  - Where name is optional
  - Do not write name when using SMALL as a memory model

Putting it Together!

```
ORG 0100h

.MODEL SMALL
.STACK 100h

.DATA
    ;data definition go here
.CODE
    ;instructions go here
```