# Introduction to IBM PC Assembly Language

# Assembly Language Syntax

- Assembler
  - Assembly language programs are translated into machine language instructions by assembler
  - Assembly language code is not case sensitive
- Statements
  - Program consists of statements one per line
  - Two types: instruction and assembler directive
  - Statements have four fields
    - name operation operand(s) comment
    - Instruction- START: MOV CX,5 ;initialize counter
    - Assembler directive- MAIN PROC

# Name Field

- Used for instruction labels, procedure names and variable names
- Assembler translates name into memory address
- Names can be 1 to 31 characters long and may consist of letters, digits 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

# Name Field

| Legal names | Illegal Names |
|---|---|
| COUNTER_1 | TWO WORDS |
| @character | 2abc |
| .TEST | A45.28 |
| DONE? | YOU&ME |

# Operation Field

- Contains symbolic (Operation code)
- Assembler translates 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 code is not translated into machine code, it only tells assembler to do something.
- Example: PROC psuedo-op is used to create a procedure

# Operand Field

- Specifies the data that are to be acted on by the operation
- An instruction may have zero, one or more operands.
- In two-operand instruction, first operand is destination, second operand is source.
- For an assembler directive, operand field represents more information about the directive
- Examples

```
NOP             ;no operand, does nothing
INC AX          ;one operand, adds 1 to the contents of AX
ADD WORD1, 2; two operands, adds value 2 to the contents
                   of memory location WORD1
```

# Comment Field

- Say something about what the statement does
- Marked by semicolon in the beginning
- Assembler ignores anything after semicolon
- Optional
- Good practice

# Program Data

- Processor operates only on binary data.
- In assembly language, you can express data in:
  - Binary
  - Decimal
  - 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

# 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
- Mainly three types
  - Byte variables
  - Word variables
  - Arrays

# Data Defining Pseudo-Ops

| Pseudo-ops | Description | Bytes | Examples |
|---|---|---|---|
| **DB** | Define Byte | 1 | var1 DB ‘A’<br>Var2 DB ?<br>array1 DB 10, 20,30,40 |
| **DW** | Define Word | 2 | var2 DW ‘AB’<br>array2 DW 1000, 2000 |
| **DD** | Define Double Word | 4 | Var3 DD -214743648 |
| **DQ** | Define Quad Word | 8 | Var DQ ? |
| **DT** | Define Ten Bytes | 10 | Var DT ? |

# Example

- 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
- XCHG
- ADD
- SUB
- INC
- DEC
- NEG

# MOV

- Transfer data
  - Between registers
  - Between register and a memory location
  - Move a number directly to a register or a memory location
- Syntax
  - MOV destination, source
- Example
  - MOV AX, WORD1

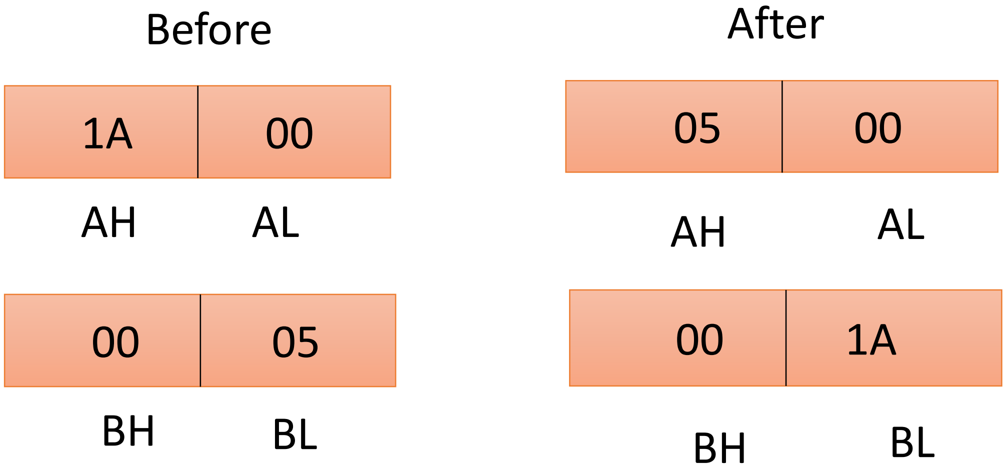
|  | Before | After |
|---|---|---|
| AX | 0006 | 0008 |
| WORD1 | 0008 | 0008 |

# 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
  - Register and a memory location
- 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

- To add contents of
  - Two registers
  - A register and a memory location
  - A number to a register
  - A number  to a memory location
- Syntax
  - ADD destination, source
- Example
  - ADD WORD1, AX

| | Before | After |
|---|---|---|
| AX | 01BC | 01BC |
| WORD1 | 0523 | 06DF |

# SUB

- To subtract the contents of:
  - Two registers
  - A register and a memory location
  - A number from a register
  - A number  from a memory location
- Syntax
  - SUB destination, source
- Example
  - SUB  AX, DX

|  | Before | After |
|---|---|---|
| AX | 0000 | FFFF |
| DX | 0001 | 0001 |

# Legal Combinations of Operands for ADD and SUB

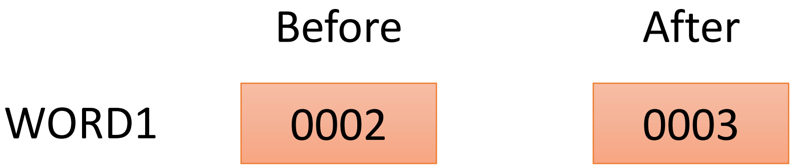
| Destination Operand | Source Operand | Legal |
|---|---|---|
| General Register | General Register | YES |
| General Register | Memory Location | YES |
| General Register | Constant | YES |
| Memory Location | General Register | YES |
| Memory Location | Memory Location | NO |
| Memory Location | Constant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

# Memory to Memory Instruction

- ADD BYTE1, BYTE2   Illegal instruction
- Solution?
  - MOV AL, BYTE2
    ADD BYTE1, AL
- What can be other possible solutions?
- How can you add two word variables?

# INC

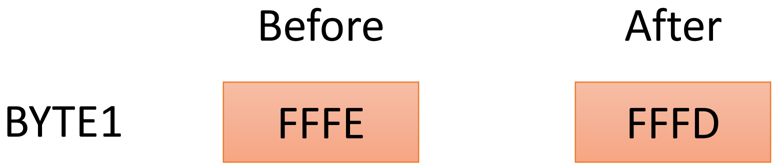
- INC (increment) instruction is used to add 1 to the contents of
  - a register
  - memory location.
- Syntax:
  - INC destination
- Example:
  - INC WORD1



|  | Before | After |
|---|---|---|
| WORD1 | 0002 | 0003 |

# DEC

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

# Translation of High-level Language to Assembly Language

- 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<br>MOV B, AX |
| **A = 5 - A** | MOV AX, 5<br>SUB AX, A<br>MOV A, AX<br>**OR**<br>NEG A<br>ADD A, 5 |
| **A = B – 2 x A** | MOV AX, B<br>SUB AX, A<br>SUB AX, A<br>MOV A, AX |

# Program Structure

- 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.
- Each program segment is translated into a memory segment by the assembler.

# 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
- Constant definitions are often made here
- 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
- Inside a code segment instructions are organized as procedures

# The Format of a Code

```
.MODEL SMALL
.STACK 100h
.DATA
    ;data definition go here
.CODE
MAIN PROC
    ;instructions go here
MAIN ENDP
    ;other procedures go here
END MAIN
```

# Input and Output Instructions

| Function Number | Routine | Function | Code |
|---|---|---|---|
| 1 | Single key input | Input: AH=1<br>Output: AL=ASCII Code if character is pressed<br>=0 if non-character key is pressed | MOV AH,1 ;input key function<br>INT 21H ;ASCII code in AL |
| 2 | Single character output | Input: AH=2<br>DL=ASCII code of the display character or control character<br>Output: AL=ASCII code of the display character or control character | MOV AH,2 ;display character function<br>MOV DL,'? ';character is ?<br>INT 21H; display character |
| 9 | Character string output | Input: DX=offset address of string. The string must end with a $ character | MSG DB 'HELLO$' |

# LEA Instruction

- It means Load Effective Address
- To get the offset address of the character string in DX we use LEA instruction
- Syntax
  - LEA destination, source
- Destination is  a general register and source is a memory location
- Example
  - LEA DX, MSG