Microprocessor Based Systems

Presented by
Dr. Md. Abir Hossain
Associate Professor
Dept. Of ICT
MBSTU

Outline

- From Book: Chapter 4
- Assembly language syntax
  - Name Field
  - Operation Field
  - Operand Field
  - Comment Field
- Program Data
- Variables

Assembly language Syntax

- Assembly language programs are translated into machine language instructions by an assembler, so they must be written to confirm to the assembler's specifications.
- One of the most popular assembler is Microsoft Macro Assembler (MASM).
- Programs consist of statements, one per line
- Each statement is either an:
  - Instruction, which the assembler translates into machine code.
  - Assembler directive, which instructs the assembler to perform some specific task (ex. Allocating memory space for a variable or creating a procedure).

Syntax (Cont.)

- Both instructions and directives have up to four fields:

  [name] operation [operand(s)] [comment]

  - [Fields are optional]
  - At least one blank or tab character must separate the fields
  - The fields do not have to be aligned in a particular column, but they must appear in the above order.

- An example of an instruction:

  START: MOV CX,5 ; initialize counter

- An example of an assembler directive:

  MAIN PROC

Name Field

- The name field is used for:
  - Instruction labels.
  - Procedure names.
  - Variable names Ex. Table look-up instruction XLAT (used for translation)
- The assembler translates names into memory addresses.
- Names:
  - Can be from 1 to 31 characters long (not case sensitive).
  - May consist of letters, digits, and the special characters
  - ? . @ _ $ % (Thus, embedded blanks are not allowed).
- Names may not begin with a digit
- Uppercase and lowercase considered as same.

Name Field Examples

COUNTER1		
2abc		Begins with a digit
@CHARACTER		
A45. 28		. Not first character
TWO WORDS		Contains a blank
STD_NUM		
.TEST		
YOU&ME		Contains an illegal character

Operation Field

- For an instruction, the operation field contains a symbolic **op**eration **code (opcode)**
- The assembler translates a symbolic opcode into a machine language opcode
- Opcode symbols often describe the operation's function (ex. MOV, ADD, SUB)

Operation Field

- In an assembler directive, the operation field contains a **pseudo-op**eration code **(pseudo-op).**

- Pseudo-ops are not translated into machine code, rather, they simply tell the assembler to do something (ex. The PROC pseudo-op is used to create a procedure).

Operand Field

- For an instruction, the operand field specifies the data that are needed to be acted on by the operation.

- An instruction may have zero, one, or two operands

- Examples:
- NOP ; no operands... does nothing
- INC AX ; one operand... adds 1 to the contents of AX
- ADD WORD1,2 ; two operands... adds 2 to the contents of memory word WORD1

Destination operand
register or memory location where the result is stored (note:some instructions don't store the result)

Source operand
usually not modified by the instruction

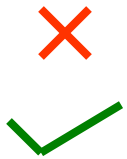
Comment Field

- The comment field of a statement is used by the programmer to say something about what the statement does
- A semicolon marks the beginning of this field, and the assembler ignores anything typed after the semicolon.
- It is almost impossible to understand an assembly language program without comments.
- Good programming practice dictates a comment on almost every line.

Comment Field (Cont.)

- Examples:

```
MOV CX, 0     ; move 0 to CX
MOV CX, 0     ; CX counts terms, initially 0
```



- Thus, comments are used to put the instruction into the context of the program
- It is permissible to make an entire line a comment, and to use them to create space in a program.

Program Data

- In an assembly language program we may express data as:
  - **Binary**: bit string followed by ‘B’ or ‘b’
  - **Decimal:** string of decimal digits followed by an optional ‘D’ or ‘d’
  - **Hex:** begins with a decimal digit and ends with ‘H’ or ‘h’
  - **Characters & Character strings**: enclosed in a single or double quotes or by there ASCII codes.

- Any number may have an optional sign.

Program Data (Cont.)

Number	Type
11011	
1101B	
64223	
-21843D	
1,234	
1BADH	
1B4D	
FFFFH	
0FFFFH	

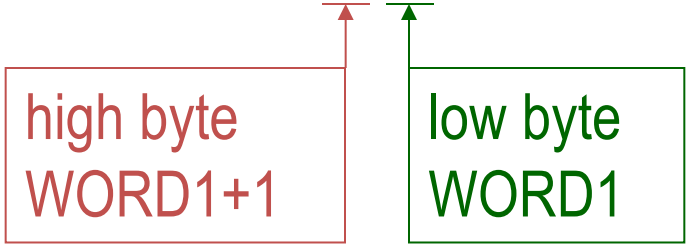
Variables

- Variables play the same role in assembly language that they do in HLL
- Each variable has a data type and is assigned a memory address by the program
- The table below shows the assembler directives that are used to define the variables

Pseudo-op	Stands for
DB	Define Byte
DW	Define Word
DD	Define Double Word
DQ	Define Quote
DT	Define Ten bytes

Bytes of the Words

- WORD1 DW 1234H



Variables – Byte & Word

- Syntax:

  name DB initial_value

- Example:
  - ALPHA DB 4 a memory byte is associated with the name ALPHA, and initialized to 4.
  - BYT DB ? a memory byte is associated with the name BYT, and uninitialized.
  - WRD DW -2 a memory word is associated with the name WRD, and initialized to -2.
- The decimal range is:
  - Unsigned representation: 0 to 255
  - Signed representation: -128 to 127

Variables - Arrays

- an array is a sequence of memory bytes or words.

- Example:

```
B_ARRAY        DB   10H,20H,30H
```

Symbol	Address	Contents
B_ARRAY	200H	10H
B_ARRAY+1	201H	20H
B_ARRAY+2	202H	30H

Variables – Array (words)

- Example:

```
W_ARRAY   DW      1000,40,29887,329
```

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

The DUP Operator

- It is possible to define arrays whose elements share a common initial value by using the DUP (duplicate) operator.

- Syntax:
  repeat_count DUP (value)

- Example:
  DELTA DB 212 DUP (?) creates an array of 212 uninitialized bytes.

  GAMMA DW 100 DUP (0) set up an array of 100 words, with each entry initialized to 0.

Character String

- Any array of ASCII codes can be initialized with a string of characters.
- Example:
  - LETTERS DB 'ABC'

    =

    LETTERS DB 41H,42H,43H

- Inside a string, the assembler differentiates between upper and lowercase.
- It is possible to combine characters and numbers in one definition:
- Example: MSG DB 'HELLO',0AH,0DH, '$'

Character Strings

- LETTERS DB ‘ABC’
- LETTER DB 41H, 42H, 43H
- MSG DB ‘HELLO’, 0AH, 0DH, ‘$’
- MSG DB 48H, 45H, 4CH, 4CH, 4FH, 0AH, 0DH, 24H

EQU

- The EQU (equates) pseudo-op is used to assign a name to a constant.

name **EQU** **constant**

- LF EQU 0AH
  - MOV DL, 0AH
  - MOV DL, LF
- PROMPT EQU ‘TYPE YOUR NAME’
  - MSG DB ‘TYPE YOUR NAME’
  - MSG DB PROMPT

No memory is allocated for EQU Names

Few Basic Instructions

117 instructions

- Over a hundred Instructions for 8086
- Some specially designed instructions for advanced processors
- We discuss six of most useful instructions

MOV

- The MOV instruction is used to transfer data between registers, between a register and a memory location, or to move a number directly into a register or memory location.
- **MOV destination, source**

```
MOV     AX, WORD1
MOV     AX, BX
MOV     AH, 'A'
```

XCHG

- The XCHG operation is used to exchange the contents of two registers, or a register, and a memory location.
- **XCHG destination, source**

```
XCHG   AH, BL
XCHG   AX, WORD1
```

Legal Combinations of Operands for MOV and XCHG

Source Operand	Destination Operand			
	General Register	Segment Register	Memory Location	Constant
General Register	Yes	Yes	Yes	**No**
Segment Register	Yes	**No**	Yes	**No**
Memory Location	Yes	Yes	**No**	**No**
Constant	Yes	**No**	Yes	**No**

Source Operand	Destination Operand	
	General Register	Memory Location
General Register	Yes	Yes
Memory Location	Yes	**No**

Restrictions on MOV and XCHG

```
ILLEGAL:    MOV WORD1, WORD2
```

But we can overcome this problem in the following lines

```
MOV    AX, WORD2
MOV    WORD1, AX
```

ADD and SUB

- The ADD and SUB instructions are used to add or subtract the contents of two registers, a register and a memory location, or to add (subtract) a number to (from) a register or memory location.

ADD destination, source

SUB destination, source

```
ADD     WORD1, AX
SUB     AX, DX
ADD     BL, 5
```

Restrictions on ADD and SUB

```
ILLEGAL:  ADD  BYTE1, BYTE2
We can solve using the following lines
MOV  AL, BYTE2          ; AX gets BYTE2
ADD  BYTE1, AL          ; add it to BYTE1
```

Legal Combinations of Operands for ADD and SUB

Source Operand	Destination Operand	
	General Register	Memory Location
General Register	Yes	Yes
Memory Location	Yes	**No**

INC and DEC

- INC (increment) is used to add 1 to the contents of a register or memory location and DEC (decrement) subtracts 1 form a register or memory location.

INC destination

DEC destination

INC WORD1

DEC BYTE1

NEG

- NEG is used to negate the contents of the destination.
- NEG does this by replacing the contents by its two's complement.

NEG destination

NEG BX

Type Agreement of Operands

```
MOV AX, BYTE1          ; illegal
MOV AH, 'A'
MOV AX, 'A'        ; move 0041h into AX
```

Translation of high-Level Language to Assembly Language

Statement	**Translation**
B = A	

```
MOV  AX, A; move A into AX
MOV  B, AX; and then into B
```

Translation of high-Level Language to Assembly Language

Statement	**Translation**
A = 5 – A	

	`MOV  AX, 5; put 5 in AX`
	`SUB  AX, A; AX contains 5 - A`
	`MOV  A, AX; put it in A`

A = 5 – A	
	`NEG  A    ; A = -A` `ADD  A, 5 ; A = 5 - A`

Translation of high-Level Language to Assembly Language

Statement **Translation**

A = B – 2 x A

```
MOV  AX, B; AX has B
SUB  AX, A; AX has B – A
SUB  AX, A; AX has B – 2 x A
MOV  A, AX; move result to A
```

PROGRAM STRUCTURE

- Assembly language program occupies code, data and stack segment in memory
- Same organization reflected in assembly language programs as well
- Code data and stack are structured as program segments
- Program segments are translated to memory segments by assembler

MEMORY MODELS

Size of code and data, a program can have is determined by specifying a memory model using .MODEL directive

```
.MODEL      memory_model
```

vvi

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	code in more than one segment data in more than one segment no array larger than 64k bytes
HUGE	code in more than one segment data in more than one segment arrays may be larger than 64k bytes

DATA SEGMENT

- A program's data segment contains all the variable definitions.
- Constant definitions are often made here as well, but they may be placed elsewhere in the program since no memory allocation is involved.

`.data` directive to declare a data segment

```
.DATA
WORD1       DW 2
WORD2       DW 5
MSG         DB 'THIS IS A MESSAGE'
MASK        EQU 10010111B
```

STACK SEGMENT

- The purpose of the stack segment declaration is to set aside a block of memory (the stack area) to store the stack.
- The stack area should be big enough to contain the stack at its maximum size.

```
.STACK      100H
```

- If size is omitted, by default 1kB is set aside

CODE SEGMENT

- The code segment contains a program's instructions.

```
.CODE name
```

- Inside a code segment, instructions are organized as procedures.

```
name PROC
; body of the procedure
name ENDP
```

- The last line in the program should be the END directive, followed by name of the main procedure.

```
MAIN        PROC
; instructions go here
MAIN        ENDP
; other procedures go here
```

PUTTING IT TOGETHER

vvi

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

INPUT AND OUTPUT INSTRUCTIONS

- CPU communicates with the peripherals through IO ports
  - IN and OUT instructions to access the ports directly
    - Used when fast IO is essential
    - Seldom used as
      - Port address varies among compluter models
      - Easier to program IO with service routine

IO SERVICE ROUTINES

IO Service routines

INT

- I/O service routines
  - The **B**asic **I**nput/**O**utput **S**ystem (BIOS) routines
  - The DOS routines
- The **INT (interrupt)** instruction is used to invoke a DOS or BIOS routine.
- `INT 21h`
  - invokes a BIOS routine that performs keyboard input.

INT 21H

- INT 21h may be used to invoke a large number of DOS functions.
- A particular function is requested by placing a function number in the AH register and invoking INT 21h.

FUNCTION 1: SINGLE-KEY INPUT

Input:

AH = 1

Output:

AL = ASCII code if character key is pressed

= 0 if non-character key is pressed

FUNCTION 1: SINGLE-KEY INPUT

```
MOV   AH, 1          ; input key function
INT   21h            ; ASCII code in AL
```

FUNCTION 2: DISPLAY A CHARACTER OR EXECUTE A CONTROL FUNCTION

Input:

AH = 2

DL = ASCII code of the display character or control character

Output:

AL = ASCII code of the display character or control character

FUNCTION 2: DISPLAY A CHARACTER OR EXECUTE A CONTROL FUNCTION

```
MOV     AH, 2       ; display character function
MOV     DL, '?'     ; character is '?'
INT     21h         ; display character
```

PRINCIPAL CONTROL CAHARCTERS

ASCII Code HEX	Symbol	Function
7	BEL	beep
8	BS	backspace
9	HT	tab
A	LF	line feed (new line)
D	CR	carriage return (start of current line)

A FIRST PROGRAM

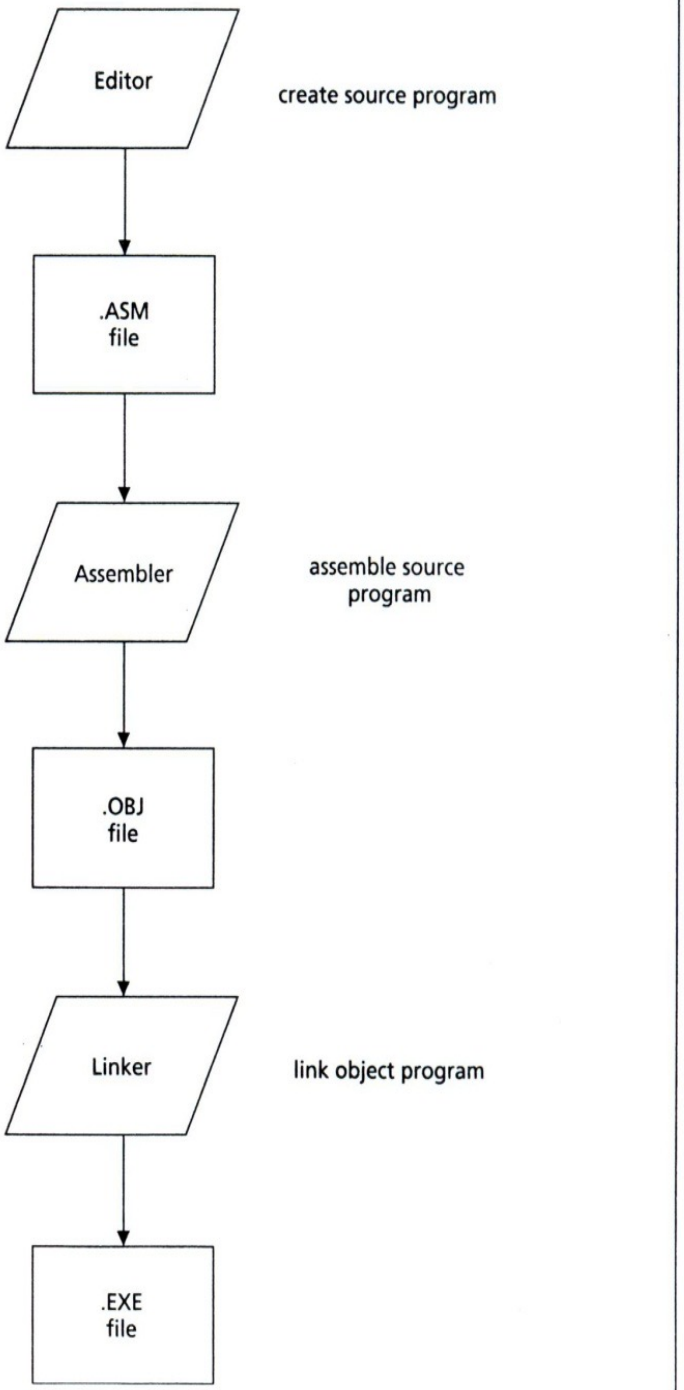
- ECH.ASM will read a character from the keyboard and display it at the beginning of the next line.
- The data segment was omitted because no variables were used.
- When a program terminates, it should return control to DOS.
- This can be accomplished by executing INT 21h, function 4Ch.


```
TITLEECHO PROGRAM
.MODEL    SMALL
.STACK    100H
.CODE
MAIN PROC
; display prompt
  MOV AH, 2     ; display character function
  MOV DL, '?'         ; character is '?'
  INT21H        ; display it
; input a character
  MOV AH, 1     ; read character function
  INT     21H         ; character in AL
  MOV BL, AL          ; save it in BL
```

```
; go to a new line
   MOV AH, 2          ; display character function
   MOV DL, 0DH             ; carriage return
   INT 21H            ; execute carriage return
   MOV DL, 0AH             ; line feed
   INT 21H            ; execute line feed
; display character
   MOV DL, BL              ; retrieve character
   INT 21H            ; and display it
; return to DOS
   MOV AH, 4CH             ; DOS exit function
   INT 21H            ; exit to DOS
MAIN  ENDP
   END MAIN
```

ASSEMBLY CODE

PROGRAMMING STEPS



STEP 1. CREATE THE SOURCE PROGRAM FILE

- An editor is used to create the preceding program.
- The .ASM is the conventional extension used to identify an assembly language source file.

STEP 2. ASSEMBLE THE PROGRAM

- The Microsoft Macro Assembler (MASM) is used to translate the source file (.ASM file) into a machine language object file (.OBJ file).
- MASM checks the source file for syntax errors.
- If it finds any, it will display the line number of each error and a short description.
- C:\>MASM *File_Name*;

STEP 3. LINK THE PROGRAM

- The Link program takes one or more object files, fills in any missing addresses, and combines the object files into a single executable file (.EXE file)
- This file can be loaded into memory and run.
- C:\>LINK *File_Name*;

STEP 4. RUN THE PROGRAM

- To run it, just type the run file name.
- C:\>*File_Name*

INT 21H, FUNCTION 9: DISPLAY A STRING

Input:

DX = offset address of string.

The string must end with a '$' character.

LEA

load effective address

- LEA is used to load effective address of a character string.
- **LEA destination, source**
- MSG DB ‘HELLO!$’

```
LEA     DX, MSG     ; get message
MOV     AH, 9       ; display string function
INT     21h         ; display string
```

PROGRAM SEGMENT PREFIX

- When a program is loaded into memory, DOS prefaces it 256 byte PSP which contains information about the program
- DOS places segment no of PSP in DS and ES before executing the program
- To correct this, a program containing a data segment must start with these instructions;

```
MOV AX, @DATA
MOV DS, AX
```

Print String Program

```
.MODEL      SMALL
.STACK      100H
.DATA
MSG   DB    'HELLO!$'
.CODE
MAIN  PROC
; initialize DS
   MOV AX, @DATA
   MOV DS, AX              ; intialize DS
; display message
   LEA DX, MSG             ; get message
   MOV AH, 9         ; display string function
   INT 21H           ; display message
; return to DOS
   MOV AH, 4CH
   INT 21H           ; DOS exit
MAIN  ENDP
   END MAIN
```

A CASE CONVERSION PROGRAM

- CASE.ASM begins by prompting the user to enter a lowercase letter, and on the next line displays another message with the letter in uppercase.
- The lowercase letters begin at 61h and the uppercase letters start at 41h, so subtraction of 20h from the contents of AL does the conversion.

CASE CONVERSION PROGRAM

```
.MODEL      SMALL
.STACK 100H
.DATA
CREQU0DH
LF EQU0AH
MSG1  DB    'ENTER A LOWER CASE LETTER: $'
MSG2  DB    CR, LF, 'IN UPPER CASE IT IS: '
CHAR  DB    ?, '$'
.CODE
MAIN  PROC
; intialize DS
    MOV     AX, @DATA     ; get data segment
    MOV     DS, AX        ; intialize DS
; print user prompt
    LEA  DX, MSG1     ; get first message
    MOV     AH, 9         ; display string function
    INT  21H          ; display first message
```

```
; input a character and convert to upper case
    MOV     AH, 1           ; read character function
    INT 21H                 ; read a small letter into AL
    SUB     AL, 20H                 ; convert it to upper case
    MOV     CHAR, AL        ; and store it
; display on the next line
    LEA DX, MSG2    ; get second message
    MOV     AH, 9           ; display string function
    INT 21H                 ; display message and upper case
    letter in front
; DOS exit
    MOV     AH, 4CH
    INT 21H                 ; DOS exit
MAIN ENDP
    END MAIN
```

CASE CONVERSION PROGRAM