

- To provide an understanding of assembly language.
- To provide an understanding of the process of assembling and linking a computer program.
- To familiarise the student with RISC processor architecture.
- To provide an introduction to multiprocessor based systems.
- To introduce mechanisms for improving system performance.

Advanced Computer Architecture

- Know the *process* of assembling and linking a computer program written in assembly language.
- Write computer programs using assembly language.
- Understand the concept of Reduced Instruction Set Computers (RISC).
- Understand the concept of cache memories and pipeline processing.
- Understand the architecture of multiprocessor systems.

- Assembly Language Programming
- RISC/CISC Architecture
- Performance Enhancements (Cache/Pipelines)
- Multiprocessor Systems



Reading List

Structural Computer Organisation
By Tanenbaum, A. S.
Prentice-hall, 1990

IBM PC Assembly Language & Programming
By Peter Abel
Prentice-hall, 1991

Assembly Language Programming and Organisation
By Ytha Yu & Charles Marut
McGraw-Hill

Advanced Computer Architectures
By Sima, Fountain, Kacsuk
Addison-Wesley, 1997

CS-01-ASM

March 04, 2001

5



Reading List

Advanced Assembly Language
By Steven Holzner
Simon & Shuster

Computer Organisation and Assembly Language Prog.
By Michael Thorne
Benjamin Cummings

and other related materials in the library.....

CS-01-ASM

March 04, 2001

6



Assembly Language Topics

- Introduction to Assembly Language
- Instruction Format/syntax
- Services (Pre-defined Routines)
- Data Defining Commands
- Mathematical Commands
- Bit Manipulation Commands
- Flow Control
- Procedures
- Macros
- Arrays

CS-01-ASM

March 04, 2001

7



Generation Of Languages

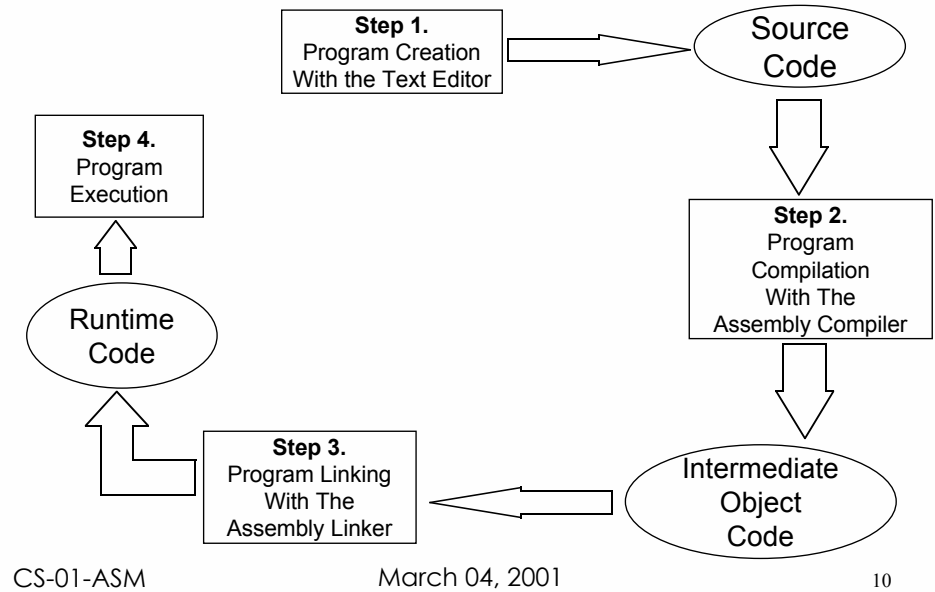
- 1) Machine Language
- 2) Assembly Language (symbolic language)
- 3) Procedure Oriented Languages
- 4) Problem Oriented Languages
- 5) Artificial Intelligence

CS-01-ASM

March 04, 2001

8

Compilation Stages



Reasons For Assembly Language

- Develop programs which perform very quickly (fast execution speed)
- Develop programs which are very small in its size (small object codes)
- Develop powerful programs (as they enable direct links with hardware devices)
- Allow us to learn more about the operations of the computer, since the program is operating at the lowest level.

Instruction Syntax/Format

An assembly language statement can be divided into *four* general section, as shown below:-

name/label	operation	operand	comment
------------	-----------	---------	---------

Name/Label

This part of the assembly language statement/instruction allows you to specify a *user-defined identifier* which is used to uniquely identify an object (or element) within the program.

Objects or elements within the program can include *variables* or even *procedures*.

Operand

Most instructions are incomplete if operands are not specified.

Operands refers to data sources which the instruction is obtaining its data from, and may include data from the computer system's *memory* or the computer system's internal *registers*.

Operation

This item, within an assembly language instruction, refers to the *task* to be carried out.

Types of operations would include such tasks as *moving* data between different areas of the computer system.

Comments

Being a language which operates closest to the lowest language level of a machine, comments should be easily included in the program to improve readability.

The last item of the assembly language instruction syntax refers to the *comments*.

Comments within an assembly language program begins with a *semi-colon*.



Example Program

```

.model small
.stack 100h
.code
MAIN PROC
;***** display a prompt
    mov  ah,2    ; display a character function
    mov  dl,'?'  ; character ? is displayed
    int  21h     ; call DOS to display

; ***** end program
    mov  ah,4Ch  ; DOS exit function
    int  21h     ; exit to DOS
MAIN endp
end MAIN

```



Example Program

```

.model small
.stack 100h
.code
MAIN PROC
;***** display a prompt
    mov  ah,2    ; display a character function
    mov  dl,'?'  ; character ? is displayed
    int  21h     ; call DOS to display

; ***** end program
    mov  ah,4Ch  ; DOS exit function
    int  21h     ; exit to DOS
MAIN endp
end MAIN

```

The size (model) of the computer program in memory



Memory Model

SMALL	code in 1 segment data in 1 segment
MEDIUM	code in more than 1 segment data in 1 segment
COMPACT	code in 1 segment data in more than 1 segment
LARGE	code in more than 1 segment data in more than 1 segment arrays no larger than 64K
HUGE	code in more than 1 segment data in more than 1 segment array can be larger than 64K



Example Program

```

.model small
.stack 100h
.code
MAIN PROC
;***** display a prompt
    mov  ah,2    ; display a character function
    mov  dl,'?'  ; character ? is displayed
    int  21h     ; call DOS to display

; ***** end program
    mov  ah,4Ch  ; DOS exit function
    int  21h     ; exit to DOS
MAIN endp
end MAIN

```

Size of the program's stack

.STACK size

This parameter specifies the size of the stack which the program is to use.

By default (if size is left out) the stack size will be set to 1Kbyte.

```
.model small
.stack 100h
.code
MAIN PROC
;***** display a prompt
    mov  ah,2    ; display a character function
    mov  dl,'?'  ; character ? is displayed
    int  21h     ; call DOS to display

; ***** end program
    mov  ah,4Ch  ; DOS exit function
    int  21h     ; exit to DOS
MAIN endp
end MAIN
```

Name of the user defined routine

```
.model small
.stack 100h
.code
MAIN PROC
;***** display a prompt
    mov  ah,2    ; display a character function
    mov  dl,'?'  ; character ? is displayed
    int  21h     ; call DOS to display

; ***** end program
    mov  ah,4Ch  ; DOS exit function
    int  21h     ; exit to DOS
MAIN endp
end MAIN
```

End of the user's main (first) routine

```
.model small
.stack 100h
.code
MAIN PROC
;***** display a prompt
    mov  ah,2    ; display a character function
    mov  dl,'?'  ; character ? is displayed
    int  21h     ; call DOS to display

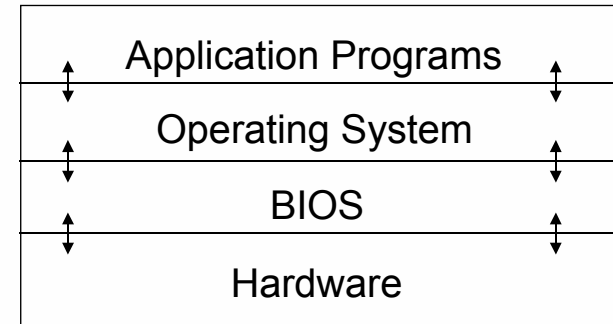
; ***** end program
    mov  ah,4Ch  ; DOS exit function
    int  21h     ; exit to DOS
MAIN endp
end MAIN
```

End of the whole program

Services

What are services?

Predefined functions which have been created by the computer manufacturer or OS to ease the programming of a computer

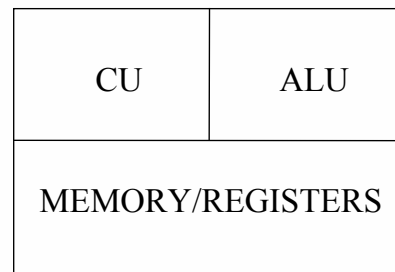
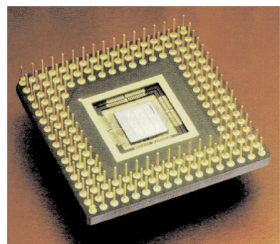


CS-01-ASM

March 04, 2001

26

Understanding Registers



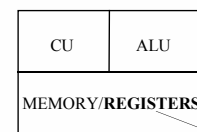
CS-01-ASM

March 04, 2001

27

Registers - General Purpose

	31	16	15	8	7	0
EAX				AH	AL	
EBX				BH	BL	
ECX				CH	CL	
EDX				DH	DL	
ESI				SI		
EDI				DI		
EBP				BP		

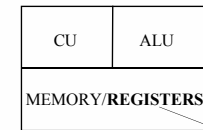
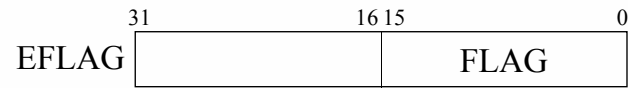


CS-01-ASM

March 04, 2001

28

Registers - Flag Register

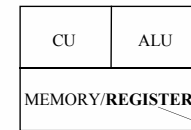


CS-01-ASM

March 04, 2001

29

Registers - Segment Register

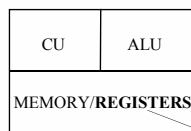
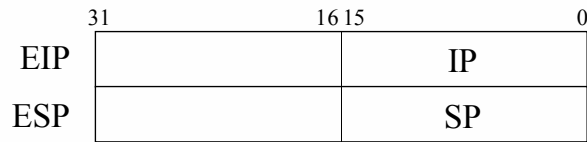


CS-01-ASM

March 04, 2001

30

Registers - Instruction/Stack Pointer

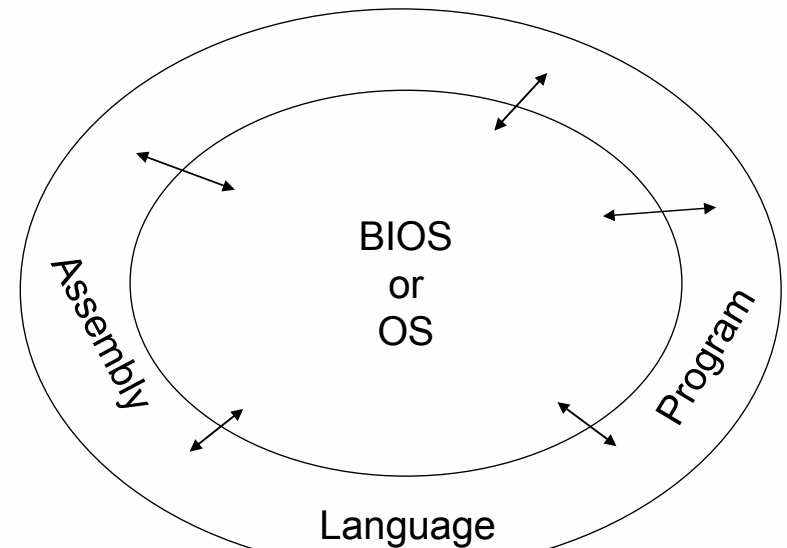


CS-01-ASM

March 04, 2001

31

Services



CS-01-ASM

March 04, 2001

32



Services - To Display a Character on the Screen

Interrupt 10h/Function 09h (BIOS)

<u>Input</u>	<u>Call Value</u>
AH	09h
AL	ASCII Character Code
BH	Screen page
BL	Character attribute/foreground colour
CX	Number of repetitions

<u>Output</u>	<u>Return Value</u>
None	



Services - To Display a Character on the Screen

Interrupt 21h/Function 02h (DOS)

<u>Input</u>	<u>Call Value</u>
AH	02h
DL	ASCII Character Code

<u>Output</u>	<u>Return Value</u>
None	



Services - To Display a String on the Screen

Interrupt 10h/Function 13h (BIOS)

<u>Input</u>	<u>Call Value/Return Value</u>
AH	13h
BH	Screen Page
BL	Character Attribute
CX	Length of string without attributes
DH	Start Cursor Row
DL	Start Cursor Column
BP	String Offset
ES	String Segment

<u>Output</u>	<u>Return Value</u>
None	



Services - To Display a String on the Screen

Interrupt 21h/Function 09h (DOS)

<u>Input</u>	<u>Call Value/Return Value</u>
AH	09h
DX	String Offset address
DS	String Segment address

<u>Output</u>	<u>Return Value</u>
None	

Comment

The string to be displayed must be terminated with a \$ character.



Services - To Read a Character From the Keyboard

Interrupt 16h/Function 00h (BIOS)

<u>Input</u>	<u>Call Value</u>
AH	00h
<u>Output</u>	<u>Return Value</u>
AH	Scan code
AL	ASCII code



Services - To Read a Character From the Keyboard

Interrupt 21h/Function 01h (DOS) - With Echo

<u>Input</u>	<u>Call Value</u>
AH	01h
<u>Output</u>	<u>Return Value</u>
AL	ASCII code



Services - To Read a String From the Keyboard

Interrupt 21h/Function 0Ah (DOS)

<u>Input</u>	<u>Call Value</u>
AH	0Ah
DS	Segment address of buffer/array
DX	Offset address of buffer/array
<u>Output</u>	<u>Return Value</u>
DS:DX	Buffer/array contains the characters read in from the keyboard (including the carriage return)

Comment

Before calling the service, offset 0 in the buffer must be set with a number indicating the total characters to be read.

After service returns, offset 1 in the buffer contains the actual number of characters read in (excluding the carriage return).



Example Program

```
.model small
.stack 100h
.code
MAIN PROC
;***** display a prompt
mov ah,2 ; display a character function
mov dl,'?' ; character ? is displayed
int 21h ; call DOS to display

;***** end program
mov ah,4Ch ; DOS exit function
int 21h ; exit to DOS
MAIN endp
end MAIN
```

How registers are
setup before the
service is called.

```
.model small
.stack 100h
.code
MAIN PROC
;***** display a prompt
mov     ah,2      ; display a character function
mov     dl,'?'    ; character ? is displayed
int     21h       ; call DOS to display
; ***** input character
mov     ah,1      ; call DOS to read a key
int     21h       ; from keyboard, store in AL
mov     bl,al     ; save character in BL
; ***** go to a new line
mov     ah,2      ; call DOS to display character
mov     dl,0Dh    ; carriage return character
```

Data Defining

The Assembly Language provides us with Data defining **Pseudo-Ops**. They enable us to *define* data.

They are as follows:-

- EQU - Enables us to create a constant.
- DB - Allows us to create a variable which contains a byte data (8 bits).
- DW - Allows us to create a variable which contains a word data (16 bits).
- DD - Allows us to create a variable which contains two words (32 bits).

.DATA

```
age      db    30
message  db    'Hello, This is Me!'
cr       EQU   0Dh
lf       EQU   0Ah
oldage   EQU   80
```

```
.model small
.stack 100h
.data
msg DB 'Hello This Is A Test!!$'

.code
MAIN PROC
; ***** display a message
mov ax,@Data ; get the data segment
; address
mov ds,ax ; point DS to data segment

; ***** call DOS to display a message
mov ah,9h ; function to display string
mov dx,offset msg ; get address of string
```

Questions

Question - Assembly Language

Write a complete assembly language program that can display on the computer's screen the following message:-

“I Love Assembly Language”

Question - Assembly Language

Write a program to display a menu (as shown below) on the screen and wait for a key to be pressed.

MAIN MENU

- 1) Student Information
- 2) Subject Information
- 3) Marks Information
- 4) Quit Program

Enter Choice :

Once a key is pressed (regardless of the input), end the program.

Question - Assembly Language

Write a complete assembly language program to display a message on the screen ("Going To Print....") and then print the letter/character "a" on the printer.

Mathematics

Being a computer, the most basic type of operation is the *mathematical* operation.

Assembly Language provides us with commands that can be used to perform the *four* basic mathematical operations,

- Addition
- Subtraction
- Multiplication
- Division

Mathematical Instructions

Addition

Addition in assembly language can be performed by using the **ADD** instruction or command.

For example,

```
ADD AX,CX
```

The first register is known as the *implied register* (as the results are automatically stored in it)



Subtraction

Subtraction in assembly language can be performed by using the **SUB** instruction or command.

For example,

SUB AX,CX

The first register is known as the *implied register* (as the results are automatically stored in it)



Byte Multiplication

MUL BL

This is interpreted as:-

$AL = AL * BL$

whereby AL is the *implied register*, and AH contains the *overflow*.



Multiplication

In assembly language there are generally *two* kinds of multiplication:-

- byte multiplication
- word multiplication

Byte Multiplication

A multiplication which involves *byte* registers and producing a result which is *word* (16bits) in size.

Word Multiplication

A multiplication which involves *word* registers and producing a result which is *double word* (32bits) in size.



Byte Multiplication

MUL CL

This is interpreted as:-

$AL = AL * CL$

whereby AL is the *implied register*, and AH contains the *overflow*.

MUL BX

This is interpreted as:-

$AX = AX * BX$

whereby *AX* is the *implied register*, and *DX* contains the *overflow*.

MUL CX

This is interpreted as:-

$AX = AX * CX$

whereby *AX* is the *implied register*, and *DX* contains the *overflow*.

In assembly language there are generally *two* kinds of division:-

- byte division
- word division

Byte Division

A division which involves dividing a *word* register (16bits) with a *byte* register (8bits) and producing a result which is *byte* in size.

Word Division

A division which involves dividing a *double word* (32bits) register with a *word* (16bits) register and producing a result which is *word* in size.

DIV BL

This is interpreted as:-

$AX = AX / BL$

whereby *AX* is the *implied register*, *AL* contains the *quotient* and *AH* contains the *remainder*.

DIV CL

This is interpreted as:-

$$AX = AX / CL$$

whereby AX is the *implied register*, AL contains the *quotient* and AH contains the *remainder*.

DIV CX

This is interpreted as:-

$$DX:AX = DX:AX / CX$$

whereby DX:AX is the *implied register*, AX contains the *quotient* and DX contains the *remainder*.

DIV BX

This is interpreted as:-

$$DX:AX = DX:AX / BX$$

whereby DX:AX is the *implied register*, AX contains the *quotient* and DX contains the *remainder*.

INC CX

This is interpreted as:-
 $CX = CX + 1$

INC AL

This is interpreted as:-
 $AL = AL + 1$

INC

Example:-

```
MOV    AH,198
INC    AH
```

Before (AH contents)

$198_{10} = 11000110_2$

After (AH contents)

$199_{10} = 11000111_2$

Decrement Commands

DEC CX

This is interpreted as:-
 $CX = CX - 1$

DEC AL

This is interpreted as:-
 $AL = AL - 1$

DEC

Example:-

```
MOV    AH,198
DEC    AH
```

Before (AH contents)

$198_{10} = 11000110_2$

After (AH contents)

$197_{10} = 11000101_2$

Bit
Manipulation
Commands



Bit Manipulation

In assembly language, data within a register is normally operated on *bitwise* (in terms of bits). Some examples of bitwise instructions are as follows:-

AND
OR
XOR
SHL
SHR
ROL
ROR



AND

	<u>Decimal</u>	<u>Binary</u>
	198	11000110
AND	117	01110101
	68	01000100

Example:-

```
MOV    AH,198
AND    AH,117
```



OR

	<u>Decimal</u>	<u>Binary</u>
	198	11000110
OR	117	01110101
	247	11110111

Example:-

```
MOV    AH,198
OR     AH,117
```



XOR

	<u>Decimal</u>	<u>Binary</u>
	198	11000110
XOR	117	01110101
	179	10110011

Example:-

```
MOV    AH,198
XOR    AH,117
```

Example:-

```
MOV    AH,198
SHL    AH,2
```

Before Shift (AH contents)

11000110
← Shift Direction

After Shift (AH contents)

1100011000

High order bits
are lost

Zero Bits are
appended

Example:-

```
MOV    AH,198
SHR    AH,2
```

Before Shift (AH contents)

11000110
Shift Direction →

After Shift (AH contents)

0011000110
Zero Bits are appended Low Order bits are lost

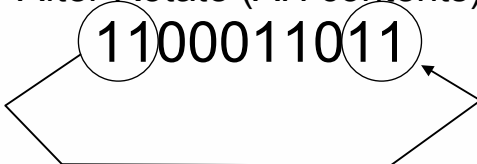
Example:-

```
MOV    AH,198
ROL    AH,2
```

Before Rotate (AH contents)

11000110
Rotate Direction ←

After Rotate (AH contents)



High order bits are
appended to low end
of AH

Example:-

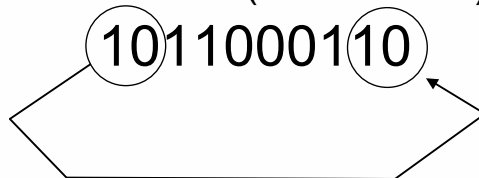
```
MOV    AH,198
ROR     AH,2
```

Before Rotate (AH contents)

11000110

Rotate Direction

After Rotate (AH contents)



Low order bits are
appended to high end
of AH

You are required to write a complete program that makes use of the common registers (AH, AL, BH, BL,.....) to perform the following mathematical calculations:-

- $(5 + 2) * 5$
- $(12 * 60) / 8 - 120$
- $(7 - 2) + (8 * 3) + 5$
- $(78 + 4) * 5 + 12$
- $(75 * 5) / 4$

Ensure that your answer is stored in the AX register after performing the calculations. You are not required to have any inputs or outputs in your program.

Flow Control (Looping Constructs)

Like any other conventional programming language, the Intel assembly language provides us with *flow control*.

Flow control allows us to dictate what statements are to be executed.

Assembly language provides us with the following *two* methods of controlling the program flow:-

- 1) loops
- 2) jumps

```
.model small
.stack 100h
.code
MAIN PROC
    mov cx,80    ; set counter to 80
    mov ah,2     ; DOS function to display
    mov dl,'*'   ; character to display
    top: int 21h  ; call DOS interrupt
    loop top     ; repeat

    mov ah,4Ch   ; DOS terminate
    int 21h      ; return to DOS
MAIN ENDP
end MAIN
```



Jumps

The Intel Assembly Language provides us with *two* kinds of jump statements;

- 1) conditional jumps
- 2) unconditional jumps



Flags

Flags are called *status indicators* which are located within the CPU and are used to represent various states which the CPU may go through.



Jumps

Conditional Jumps

Jump statements which moves control to another part of the program *depending* on the value of the *flags*.

Unconditional Jumps

Jump statements which moves control to another part of the program *regardless* of the value of the *flags*.



Conditional Jumps - An Example

```

.model small
.stack 100h
.code
MAIN PROC
    mov  bl,80    ; set counter to 80
    mov  ah,2     ; DOS function to display
    mov  dl,'*'   ; character to display
top:   int  21h    ; call DOS interrupt
       dec  bl     ; decrease BL by one
       jnz  top    ; jump if not zero

    mov  ah,4Ch   ; DOS terminate
    int  21h     ; return to DOS
MAIN ENDP
end MAIN

```

JA/JNBE	Jump If Above/Jump If Not Below Or Equal To
JAЕ/JNB	Jump If Above Or Equal To/Jump If Not Below
JB/JNAE/JC	Jump If Below/Jump If Not Above Or Equal To/Jump If Carry Flag Set
JBE/JNA	Jump If Below Or Equal To/Jump If Not Above
JCXZ	Jump If CX Is Zero
JE/JZ	Jump If Equal/Jump If Zero Flag Is Set
JG/JNLE	Jump If Greater/Jump If Not Less Than Or Equal To
JGE/JNL	Jump If Greater Than Or Equal To/Jump If Not Less Than
JL/JNGE	Jump Is Less Than/Jump If Not Greater Than Or Equal To
JLE/JNG	Jump If Less Than Or Equal To/Jump Is Not Greater Than
JNC	Jump If Carry Flag Not Set
JNE/JNZ	Jump If Not Equal To/Jump If Zero Flag Not Set
JNO	Jump If No Overflow
JNP/JPO	Jump If No Parity/Jump If Parity Is Odd
JNS	Jump If No Sign
JO	Jump On Overflow
JP/JPE	Jump On Parity (Even)/Jump If Parity Is Even
JS	Jump If Sign Bit Is Set

CS-01-ASM March 04, 2001

93

Questions

```
.model small
.stack 100h
.code
MAIN PROC
    mov ah,1      ; call DOS to read a key
    int 21h       ; from the keyboard into AL
    mov bl,al     ; save character in BL
    mov ah,1      ; call DOS to read next key
    int 21h       ; from keyboard into AL
    mov ah,2      ; DOS function to display
    cmp al,bl     ; check if AL <= BL
    jnbe ELSE     ; go display char in BL

    mov dl,al     ; set DL to have AL
    imp DISP     ; go display char in AL
```

CS-01-ASM

March 04, 2001

94

Write a complete program to read in a character and then to display the upper case equivalent (if it exist). If there is no upper case equivalent, your program must display the original input character.

Hint:-

The ASCII value of "A" is 65 and the ASCII value of "a" is 97.

The ASCII value of "Z" is 90 and the ASCII value of "z" is 122.

Question - Assembly Language

Write a complete program that will read in a character from the keyboard. Based on the character that was entered, your program must then display on the screen the following corresponding output message on the screen and then end.

- A This is the first option
- B This is the second option
- C This is the third option

For any other keys that are pressed, your program must display “Invalid Input” and to then continue reading in the next character.

Procedures

Procedures are routines which are self contained, and is created to perform a particular task.

Assembly language is not unlike other higher level languages such as Pascal, COBOL or C.

There *are* constructs which allows **procedures** to be created in assembly language.

Procedures

Procedures - An Example

```
.model small
.stack 300h
.data
    msg DB 'I Love Assembly Language!!$'

.code
MAIN    PROC
; ***** display a message
    mov  ax,@Data        ; get the data segment
                        ; address
    mov  ds,ax            ; point DS to data segment

    mov  dx,offset msg    ; get address of message
    call ShowMess        ; display message

; *****
```

Macros

A construct within assembly language which allows us to attach names to frequently used blocks of codes.

The concept of **macros** are very similar to *procedures*.

General Format of a macro:-

```
name MACRO parameter_list
:
:
ENDm
```

Macro - An Example

```
.model small
.stack 100h
.data
    CRLF    DB    13,10,'$'
    msg1    DB    'I Love Assembly Language!!$'
    msg2    DB    'I Think I Love Assembly Language!!$'

.code
; macro to display the message on the screen
ShowMess Macro Mess
    lea dx,Mess        ; load message address
    mov ah,9h          ; function to display string
    int 21h            ; call DOS
EndM
```

Macros VS Procedures

- During the assembly stage, *every* statement that invokes (calls) the macro is *replaced* with the codes found under the macro (code expansion).
- Since macros are *expanded*, they do not have overheads on the stack, such as that found with procedures.

```
.model small
.stack 100h
.data
    CRLF DB 13,10,'$'
    msg1 DB 'I Love Assembly Language!!$'
    msg2 DB 'I Think I Love Assembly Language!!$'

.code
MAIN PROC
; ***** display a message
    mov ax,@Data ; get the data segment
                ; address
    mov ds,ax    ; point DS to data segment

    lea dx,Msg1  ; load message address
```

Arrays

Arrays

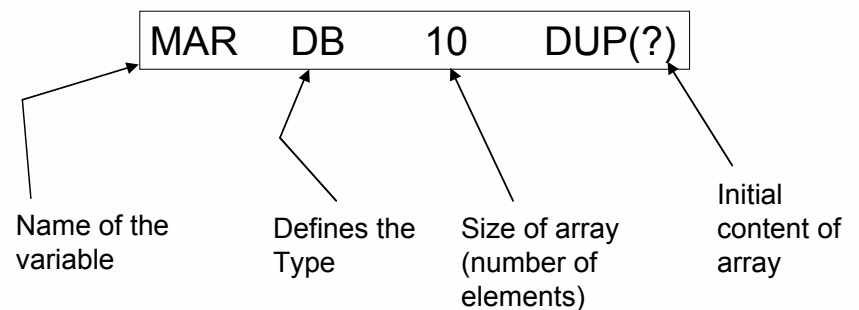
Arrays are *contiguous* memory locations whereby data is stored or located.

Most high level languages provides access to arrays.

Assembly language, through the **Intel** microprocessor, also provides us with the ability to utilise (or make use of) *arrays*.

Declaring Arrays

An example of how arrays can be *declared* (or created) in assembly language is as follows:-



MAR	DB	10	DUP(?)
-----	----	----	--------

RAM	MAR
	0
	1
	2
	3
	4
	5
	6
	7
	8
	9

Register Indirect Access

In this method of accessing an array, a register is used and contains the address of the array.

```
.model small
.stack 100h
.data
    mar db 5,10,11,12,13,14,15
    msg db 'Yes, It Is!$'
.code
MAIN PROC
    mov ax,@Data ; get data segment
    mov ds,ax ; point DS to segment

    lea si,mar ; get MAR address
    mov ah,[si] ; get value of first location
    cmp ah,05h ; check if it is 5
    jne l1 ; jmp if it is not
```

Base and Index Addressing Mode

In this method of accessing an array, a *base address* and a *displacement* (or index) is used to access individual elements in an array.

```
.model small
.stack 100h
.data
    mar    db    5,10,11,12,13,14,15
    msg    db    'Yes, It Is!$'
.code
MAIN    PROC
    mov    ax,@Data    ; get data segment
    mov    ds,ax        ; point DS to segment

    mov    bx,00h        ; get first index
    mov    ah,mar[bx]    ; get value of first location
    cmp    ah,05h        ; check if it is 5
    jne    l1            ; jmp if it is not
```

Questions

The statement,

```
mov ah,mar[bx]
```

can also be written in any one of the following format,

```
mov ah,[bx+mar]
mov ah,[bx]+mar
```

Question - Assembly Language

Write a complete assembly language program which will enable you to read in the name of a person and display the following message on the screen (followed by the person's name):-

Hello [person's name]



Question - Assembly Language

Write a complete program which will receive two inputs, one being a string and the other, a character.

The program is then to search through the string and to display on the screen whether the character was found within it.



Question - Assembly Language

You are required to write a complete assembly language program that can display on the screen any number that is stored in the AX register (or AL, for 8 bit values). For example, if the AX register has a value of 1234 (this is the decimal value), it should be displayed on the computer's screen as a string of characters, "1234".

Therefore in order to display the numerical value of 1234 on the screen the computer must display character "1" followed by character "2" followed by character "3" followed by character "4". Write a program that moves a fixed value into the AX register and from there, use your routine to display the answer on the screen. The ASCII value of character "0" is 48 and "1" is 49.



Question - Assembly Language

Write a complete assembly language program that can perform the following:-

- 1) Read in a string from the keyboard (this is regarded as plaintext, human readable form).
- 2) Based on this input string, you are required to produce the encrypted version (cyphertext) of the string using the either of the ROL/ROR command.
- 3) The program must then display the cyphertext on the screen.
- 4) After displaying the cyphertext, your program must then convert it back to plaintext.
- 5) Your program must display the plaintext on the screen.



Question - Assembly Language

You are required to write a complete assembly language program that can read in *two integer* values (maximum of four digits each) and to then perform a *mathematical addition* on these values.

After obtaining the result of the addition, your program must then display the result on the screen.

Take note that DOS and BIOS *do not* provide any routines to input or output numbers (in numerical values). The PC deals with input and output in terms of characters (ASCII).

The ASCII value of character "0" is 48 and "1" is 49.