ANNOUNCEMENTS

----

1. Sign up sheet for learning about reading and posting to newsgroups.

2. Exam 2 on 14th or 16th March

3. Exam 1 stats:

Mean	Median	Midpoint	Geometric	Harmonic
68.846	74.000	61.500	66.848	64.377

SD	Quart Dev	Range	SE mean
15.220	9.500	57.000	2.985

Minimum	Quartile 1	Quartile 2	Quartile 3	Maximum
33.000	59.000	74.000	78.000	90.000

Stack manipulation Instructions:

LIFO : Last In First Out
Bottom of Stack is in High Mem
Top of stack is in Low Mem

Only WORD sized operands (16-bit for 8086) can be stored on or removed from the stack

SS and SP registers are dedicated to the run-time stack
SS:SP points to ==> Top of Stack
(BP is also used for stack manipulation.)

Instructions that manipulate the stack:

```
push    OP
pop     OP
```

Push and Pop automatically do the right things to SP
Push:
SP is DECREMENTED by 2 (why?) and OP is stored at location pointed to by SS:SP

Pop:
One word (two bytes) popped of the stack and stored in OP
AND
SP is INCREMENTED by 2

OP must be register or Memory-ref AND
MUST be of size: One word = two bytes = 16 bits

Examples

1)

```
        push    bx
        pop     ax
```

same as

```
        mov     ax, bx
```

Well a better use might be to move the value of one segment register to another. Remember segment reg to segment reg moves are not allowed. we want to mov contents of DS to ES

```
        mov     es, ds              ; ILLEGAL
```

instead

```
        mov     ax, ds
        mov     es, ax
```

but AX might be in use (that is, contain a value that the program is using)

therefore

```
        push    ds
        pop     es
```

would do the job.

Addressing Modes:

Thinking about Pointers.....

In Intel Assembly you can store the address (offset) of a memory location in a register and refer to that memory location using the register

```
.DATA
        BUF     db      80 dup('$')

        .
        .
        .
.CODE
        .
        .
        .
        lea     dx, BUF
        call    READ

        lea     bx, BUF
        mov     ax, [bx]                ; moves a WORD
        mov     al, [bx+1]              ; moves a BYTE

; or you could try

        mov     al, [bx]                ; moves a BYTE
        mov     [bx + 1], al            ; moves a BYTE
```

LOOPS

Syntax

```
        loop        ShortLabel
```

Assumes that the loop counter is CX

```
                mov         cx, 10h             ; init cx with 16

ShortLabel:     .
                .
                .
                .
                loop        ShortLabel

AfterLoop:
```

What does loop do?

1) Decrement CX

2) IF (CX NOTEQUALTO 0)
   THEN
        GOTO ShortLabel
   ELSE
        Next Instruction (labeled AfterLoop)

Example

```
        mov     AX, 0
        mov     CX, 10

LoopLabel:

        inc     ax
        loop    LoopLabel

AfterLoop:
```

What is the value in AX at the end of this loop?

What if CX EQUALS 0

CX - 1 = FFFFh = 65535 !!!

Loop is done 65536 times !!!

Special instruction to handle such cases

```
        jcxz    ShortTarget
```

Example

```
        mov     AX, 0
        jcxz    AfterLoop

LoopLabel:

        inc     ax
        loop    LoopLabel

AfterLoop:

        .
        .
        .
```

PROCEDURE CALLS

Two instructions:

```
        call        ProcName
```

CALL causes control to be transferred to the first instruction of the procedure ProcName

When the procedure finishes we want control to return to the instruction after the CALL. Use

```
        ret
```

as the last instruction in the procedure to accomplish this.

```
        CALL    ProcName

        pushes CS and IP on stack
        Jumps to first instruction of ProcName

        RET

        pops CS and IP from stack
```

Software Interrupts

Routines for interfacing with other hardware

ROM BIOS and DOS interrupts, ship with the system or DOS

S/W interrupts are like procedure calls.
There are conventions/rules in using these interrupts just like there are rules to using READ/WRITE/EXIT/DECIN/DECOUT

Syntax

```
            int         XXH

interrupt               interrupt number in HEX (usually)

            int         10h         ; activates the video routine
                                    ; which video routine?
```

example:

```
            int         20h         ; terminate executing program
```

See APPENDIX 2

**int 21h is the most frequently used.
The DOS function dispatcher
Has a whole set of functions:**

read/write/disk i/o etc.....

See appendix 3.

int 20 must be told which function is being requested

**You give int 21 this info by placing the
FUNCTION NUMBER
in AL**

**Function 02h OUTPUT a character to video screen
Inputs: dl = character**

use the 02h function of int 12h to print an A to the screen

```
        mov     dl, 'A'

        mov     ah, 02h
        int     21h
```

**FUNCTION 09h
Prints a $ terminated string to video screen
Now you know how WRITE works!**

Programs

```
TITLE   ProgName

DOSSEG
.MODEL SMALL

.STACK 100H

.DATA

        (put data definitions  here)

.CODE


Start:

        mov     ax, @DATA
        mov     ds, ax

        (put code here)


        mov     ah, 4cH         ; CALL  EXIT
        int     21H             ;

END     Start
```

```
TITLE   AS2.ASM

DOSSEG
.MODEL  SMALL

.STACK  100h

.DATA
        buf     db      80 dup('$')
        NEWLINE db      0Dh, 0Ah, 24h

.CODE

EXTRN  EXIT:NEAR, READ:NEAR, WRITE:NEAR

START:
        mov     ax, @DATA
        mov     ds, ax

        lea     dx, buf
        call    READ

        mov     cx, dx

        lea     dx, NEWLINE
        call    WRITE
        mov     dx, cx
        call    WRITE
        lea     dx, NEWLINE
        call    WRITE
```

```
        lea     bx, buf

WhileLabel:
        mov     al,[bx]

;I can only use BX, SI, or DI for this kind of
;INDIRECT ADDRESSING

        cmp     al, '$'
        je      STOP

        add     al, 20h
        mov     [bx],al

        inc     bx
        jmp     WhileLabel

STOP:
        mov     dx, cx
        CALL    WRITE
        lea     dx, NEWLINE
        CALL    WRITE

        CALL    EXIT

END     START
```