Stack Frames and Recursion:

The part of the stack that is used for a single procedure call is referred to as the STACK FRAME

Question to think about (pgs 181 ...)

Where do we store local variables?

We can have a procedure (procA) call another procedure (procB) which then calls another and so on.

We can nest procedures.

We could also have ProcA call ProcA which calls ProcA and so on. In this extreme case when we have nested procedure calls to the same procedure, we have RECURSION.

If ProcA calls ProcA, program control reenters a procedure that is already active (that is we havn't returned from it yet). Code that supports this kind of procedures is called REENTRANT code.

many MS-DOS procedures are NOT reentrant.

Reentrant procedures must maintain seperate copies of all pertinent information for each active call.

Why?
Well, otherwise later calls to such procedures will overwrite information from previous calls needed to complete the procedure.

What information is "pertinent" ?

Return Address
Parameter values
perhaps Local variables

Where do we keep this info?

On the stack in a stack frame.

This provides a significant reason why passing parameters on the stack is better.

Code in which all pertinent information about procedure calls is maintained on a stack frame is REENTRANT

```
TITLE     addFirst.asm: demonstrates simple recursion
INCLUDE   console.mac
DOSSEG
.MODEL SMALL
.STACK 100h
.DATA
Prompt1   DB   CR, LF, LF, "Input an Integer: ", 0
intStr    DB   6 DUP(?)
ansStr1   DB   CR, LF, "The sum of the integers from 1 to ", 0
ansStr2   DB   " is ", 0
.CODE
Start:
   mov     ax, @DATA
   mov     ds, ax
Again:
   outStr  Prompt1
   inpStr  intStr                 ;value of highIndex
   aToI    intStr

   push    ax                     ;push parameter
   call    addFirst               ;start recursive chain

   outStr  ansStr1                ;print answer
   outStr  intStr                 ;  including value of highIndex
   outStr  ansStr2
   iToA    intStr, ax
                                  ;reuse intStr for printing of answer,
   outStr  intStr                 ; which has been returned in ax
   Skip    2

   jmp     Again                  ;will need a CONTROL C to exit
```

```
   mov     ax, 4c00h           ;these lines are never reached and
   int     21h                 ;  so could be eliminated

;-------------------------------------
; recursively adds 0 + 1 + 2 + ... + highIndex,
; returning answer in ax

addFirst   PROC   NEAR
   push    bp                  ;establish base pointer for
   mov     bp, sp              ;  indirect addressing

   cmp     WORD PTR [bp + 4], 0
                               ;is highIndex = 0? If so,
   jne     Recurse             ;stop the recursion, but
   xor     ax, ax              ; before doing so, record 0
                               ; as answer in
   jmp     SHORT Done          ;     ax
Recurse:                       ;here if recursion is
                               ;necessary, if highIndex>=1
   mov     ax, [bp + 4];presentvalue of highIndex
   dec        ax               ;decrement it for next call
   push    ax                  ;push parameter for next call
   call    addFirst            ;the recursive call
   add     ax, [bp + 4];the actual computation: update
                               ; the answer in ax
Done:
   pop     bp
   ret     2
addFirst   ENDP
;--------------------------------------
END   Start
```

DISASSEMBLY
----

inc ax	===	0100 0000
inc cx	===	0100 0001
inc dx	===	0100 0010
inc bx	===	0100 0011
inc sp	===	0100 0100
inc bp	===	0100 0101
inc si	===	0100 0110
inc di	===	0100 0111

16 bit register table:

ax	===	0000
cx	===	0001
dx	===	0010
bx	===	0011
sp	===	0100
bp	===	0101
si	===	0110
di	===	0111

The increment instruction looks like

01000 reg   where reg is specified by 3 bits

The DECREMENT instruction opcode is

01001 reg

16 bit register table:
ax === 0000
cx === 0001
dx === 0010
bx === 0011
sp === 0100
bp === 0101
si === 0110
di === 0111
What is the opcode in hex for

dec bx
dec si
dec ax == 48h

Assembly is so much easier to remember than
48h40h which is the same as

dec ax
inc ax

The opcode must specify:
what to do *and*
to whom it must be done to.

inc and dec are one byte assembly instructions
most instructions are more than one byte long
They have too much to say to be expressed in one byte.

example:

mov ax, F0C6 === B8 C6 F0

requires 3 bytes.

mov reg, Immed opcodes

mov ax, Immed	**B8**	**1011 1000**
mov cx, Immed	**B9**	**1011 1001**
mov dx, Immed	**BA**	**1011 1010**
mov bx, Immed	**BB**	**1011 1011**
mov sp, Immed	**BC**	**1011 1100**
mov bp, Immed	**BD**	**1011 1101**
mov si, Immed	**BE**	**1011 1110**
mov di, Immed	**BF**	**1011 1111**

mov register, immed8/16

1011 w reg | data | data if w = 1

mov al, FF

1011 0 000 | 1111 1111 ->   B0FF

mov ax, FFFF

1011 1 000 | 1111 1111 | 1111 1111

The 5th bit actually specifies the size of the operand in the "mov reg, immed" instruction

DEC reg = 0100 1 reg
INC reg = 0100 0 reg

mov reg, immed

1011 w reg | data | data

16 bit register table 8 bit register table:

ax	===	0000	al	===	0000
cx	===	0001	cl	===	0001
dx	===	0010	dl	===	0010
bx	===	0011	bl	===	0011
sp	===	0100	ah	===	0100
bp	===	0101	ch	===	0101
si	===	0110	dh	===	0110
di	===	0111	bh	===	0111

mov ah, FF

mov ax, 134C

mov ax, 14

mov cx, 0

dec cx

dec al

inc cx

And now for something completely different

mov reg, reg

is VERY different at the assembly level.

mov reg/mem to/from reg

1000 10 d w || mod | reg | r/m ||

2 byte instruction
2nd byte is called the EA byte
Effective Address byte

mod is a 2 bit field

reg is a 3 bit field

r/m is a 3 bit field

1000 10 specifies that this is a register to register move
if the mod bits are set to 11

1000 10 d w || 11 | reg | r/m ||

reg and r/m specify the two registers involved in the mov

16 bit register table 8 bit register table:

ax ===	**0000**	**al ===**	**0000**
cx ===	**0001**	**cl ===**	**0001**
dx ===	**0010**	**dl ===**	**0010**
bx ===	**0011**	**bl ===**	**0011**
sp ===	**0100**	**ah ===**	**0100**
bp ===	**0101**	**ch ===**	**0101**
si ===	**0110**	**dh ===**	**0110**
di ===	**0111**	**bh ===**	**0111**

if "d" == 1 the reg field specifies the DESTINATION register
and the r/m field specifies the SOURCE register

if "d" == 0 the r/m field specifies the DESTINATION register
and the reg field specifies the SOURCE register

MOV bx, bp

100010 1 1 || 11 | bx | bp ||

100010 1 1 || 11 | 100 | 101 ||

MOV bp, bx

100010 0 1 || 11 | 100 | 101 ||

**if "d" == 1 the reg field specifies the DESTINATION register
and the r/m field specifies the SOURCE register**

**if "d" == 0 the r/m field specifies the DESTINATION register
and the reg field specifies the SOURCE register**

the "w" flag is the word flag

16 bit register table 8 bit register table:

ax	**===**	**0000**	**al**	**===**	**0000**
cx	**===**	**0001**	**cl**	**===**	**0001**
dx	**===**	**0010**	**dl**	**===**	**0010**
bx	**===**	**0011**	**bl**	**===**	**0011**
sp	**===**	**0100**	**ah**	**===**	**0100**
bp	**===**	**0101**	**ch**	**===**	**0101**
si	**===**	**0110**	**dh**	**===**	**0110**
di	**===**	**0111**	**bh**	**===**	**0111**

**MOV BL, CH
100010 0 0 || 11 | bl | ch
100010 0 0 || 11 | 011 | 101
MOV ch, bl
100010 1 0 || 11 | 011 | 101**

**MOV al, ah
MOV bx, cx
MOV al, cx
MOV bp, sp
MOV bh, dl**