8. Procedures

Stack Operation

A *stack* is a region of memory used for temporary storage of information. Memory space should be allocated for stack by the programmer.

The last value placed on the stack is the 1st to be taken off. This is called LIFO (Last In, First Out) queue. Values placed on the stack are stored from the highest memory location down to the lowest memory location. SS is used as a segment register for address calculation together with SP.

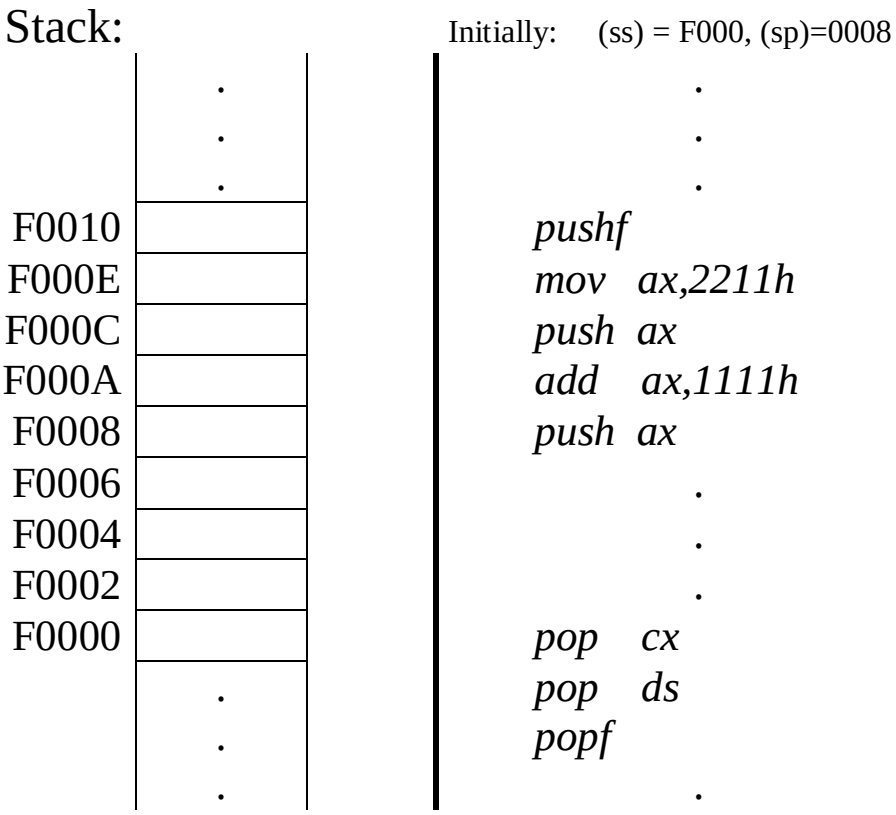
Stack Instructions

Name	Mnemonic and Format	Description
Push onto Stack	*push src*	(sp)←(sp)-2 ((sp))←(src)
Pop from Stack	*pop dst*	(dst)←((sp)) (sp)←(sp)+2
Push Flags	*pushf*	(sp)←(sp)-2 ((sp))←(psw)
Pop Flags	*popf*	(psw)←((sp)) (sp)←(sp)+2

Flags: Only affected by the *popf* instruction.

Addressing Modes: *src* & *dst* should be Words and cannot be immediate. *dst* cannot be the *ip* or *cs* register.

Exercise: Fill-in the Stack



Procedure Definition

PROC is a statement used to indicate the beginning of a procedure or subroutine.

ENDP indicates the end of the procedure.

Syntax:

```
ProcedureName      PROC     Attribute
        .
        .
        .
ProcedureName      ENDP
```

ProcedureName may be any valid identifier.

Attribute is *NEAR* if the Procedure is in the same code segment as the calling program; or *FAR* if in a different code segment.

Call and Return Instructions

Name	Mnemonic and Format	Description
Intrasegment Direct Call	*call opr*	(sp)←(sp)-2 ((sp))←(ip) (ip)←(ip)+16-bit Disp.
Intrasegment Indirect Call	*call opr*	(sp)←(sp)-2 ((sp))←(ip) (ip)←(Eff. Addr.)
Intersegment Direct Call	*call opr*	(sp)←(sp)-2 ((sp))←(cs) (sp)←(sp)-2 ((sp))←(ip) (ip)←16-bit Disp. (cs)←Segment Address
Intersegment Indirect Call	*call opr*	(sp)←(sp)-2 ((sp))←(cs) (sp)←(sp)-2 ((sp))←(ip) (ip)←(Eff.Addr.) (cs)←(Eff. Addr. + 2)
Intrasegment Return	*ret*	(ip)←((sp)) (sp)←(sp)+2
Intrasegment Return with immediate data	*ret expression*	(ip)←((sp)) (sp)←(sp)+2 (sp)←(sp)+*expression*
Intersegment Return	*ret*	(ip)←((sp)) (sp)←(sp)+2 (cs)←((sp)) (sp)←(sp)+2
Intersegment Return with immediate data	*ret expression*	(ip)←((sp)) (sp)←(sp)+2 (cs)←((sp)) (sp)←(sp)+2

		(sp)←(sp)+*expression*

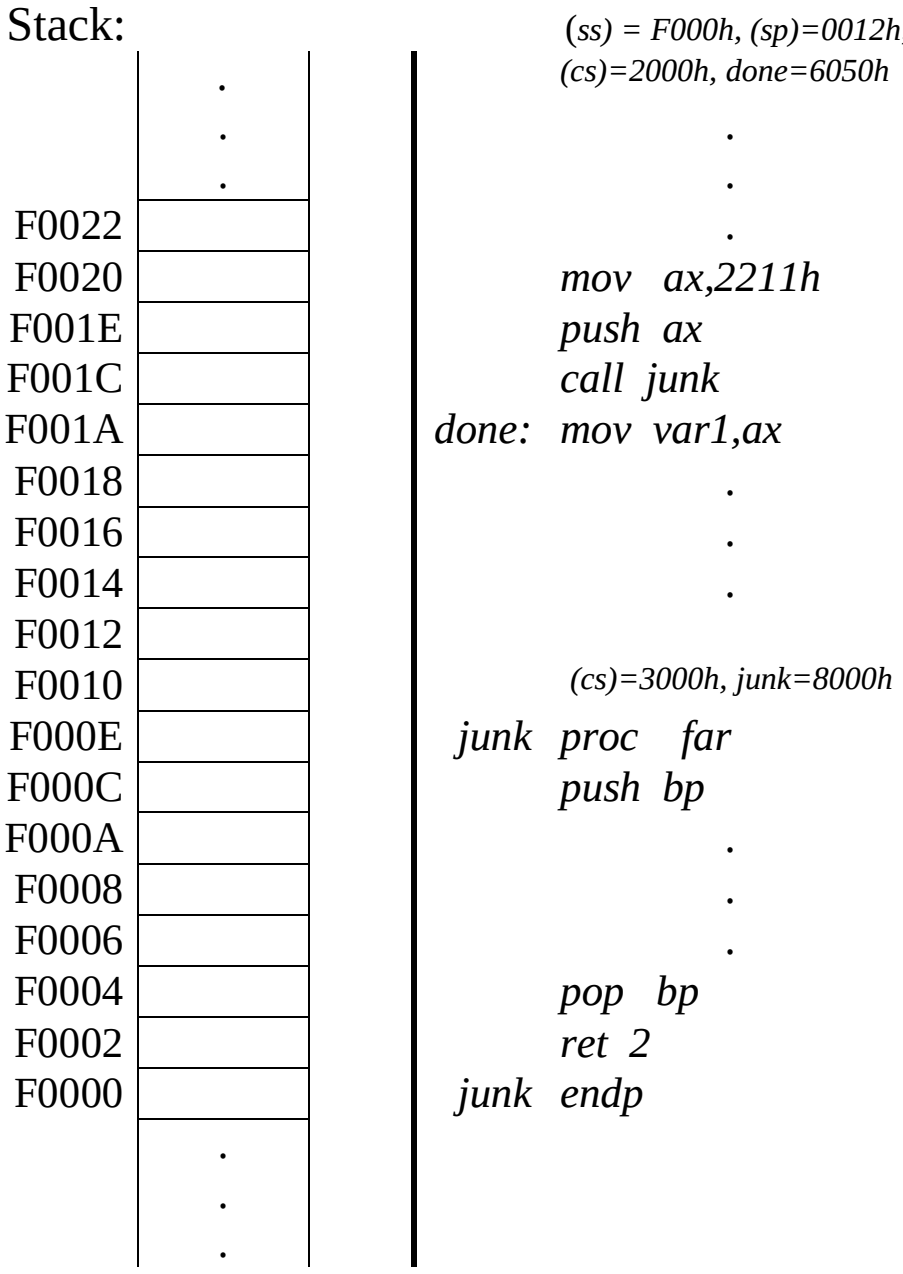
Flags: Not affected.

Addressing Modes: Any branch addressing mode except *short*.

EXAMPLE:

```
.model medium

.data
     vector1        dw        action1
     vector2        dd        action2

.code

action1 proc near
     …
     …
     ret
action1 endp

action2 proc far
     …
     …
     ret
action2 endp


start:
     …
     …
     ;Intrasegment Direct
     call action1
     …
     …
     ;Intrasegment Indirect
     call vector1
     …
     …
     ;Intersegment Direct
     call action2
     …
     …
     ;Intersegment Indirect
     call vector2
     …
     …
end start
```

Exercise: Fill-in the Stack



Exercise

Write a procedure named *multiply* that computes the product of two signed 16-bit operands. The operands will be passed in registers *si* and *di*. The procedure should return the result on *ax*. Write a program that uses the *multiply* procedure

Procedure Parameters

Few procedures perform activities without requiring some input parameters that can be passed:

1. in registers
2. in memory variables
3. on the stack

- By convention, high-level languages (like C, Pascal, PL/1, ect.) pass parameters by placing them on the stack.

- Parameter on the stack can be passed *by Value* or *by Reference*. *Passing by Value* means to put a copy of each parameter value on the stack. *Passing by Reference* means to put a copy of each parameter offset (effective address) on the stack.

- Parameters on the stack can then be accessed by procedures by using displacements or a stack-frame structure.

EXAMPLE: Passing Parameters

```
.model medium

.data
     var1      dw   ?
     var2      dw   ?

.code

action1 proc near
     …
     …
     ret 4
action1 endp

action2 proc near
     …
     …
     ret 4
action2 endp


start:
     …
     …
     ;Pass by Value
     push var1
     push var2
     call action1
     …
     …
     ;Pass by Reference
     push offset var1
     push offset var2
     call action2
     …
     …
end start
```

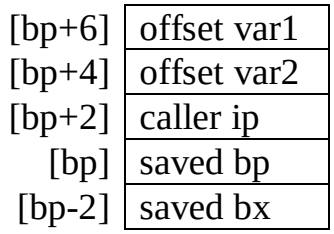
Using Displacement

To access parameters from the stack, a marker to the stack frame is required. BP & SP default to the stack if used as base registers. BP is commonly used by procedures, but need to be pushed before. Parameters are accessed at [BP+Disp.] after a push of bp and a mov of SP to BP.

EXAMPLE:

```
clear   proc        near
        push        bp
        mov         bp,sp
        push        bx
        mov         bx,[bp+4]
        mov         word ptr [bx],0
        mov         bx,[bp+6]
        mov         word ptr [bx],0
        pop         bx
        pop         bp
        ret         4
clear   endp

main:
        push        offset var1
        push        offset var2
        call        clear
        .
```

Stack:



Exercise

Write a procedure named *multiply* that computes the product of two signed 16-bit operands. The operands will be passed on the stack, by-value. The procedure should return the result on *ax*. Write a program that uses the *multiply* procedure

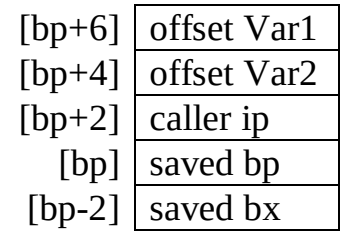
Using a Stack Frame Structure

The stack frame structure can be used as a template over the stack. Based addressing can be used after a push of bp and a mov of SP to BP. The displacement is then from the structure definition (not memory allocation is required).

EXAMPLE:

```
stack_frame struc
        saved_bp    dw     ?
        caller_ip   dw     ?
        var2_ptr    dw     ?
        var1_ptr    dw     ?
stack_frame ends

clear           proc            near
        push            bp
        mov             bp,sp
        push            bx
        mov             bx,[bp].var2_ptr
        mov             word ptr [bx],0
        mov             bx,[bp].var1_ptr
        mov             word ptr [bx],0
        pop             bx
        pop             bp
        ret             4
clear           endp

main: push              offset Var1
        push            offset Var2
        call            clear
```

Stack:



Procedure Variables

Procedures often need local memory space. The stack area can be used to allocate space dynamically for the procedure with the space de-allocated when the procedure concludes.

To allocate space for local variables, subtract from SP the number of bytes needed after setting-up the stack frame marker (BP). Then, local variables can be accessed at [BP-*number*] and the parameters at [BP+*number*].

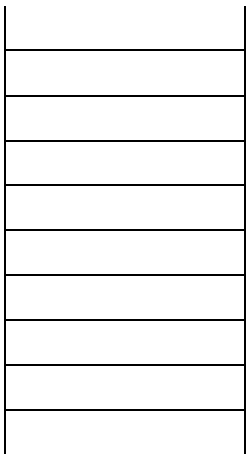
Local variables are released by moving BP back to SP (*mov sp,bp*).

Exercise: Fill-in the Stack

```
junk   proc        near
       push        bp
       mov         bp,sp
       sub         sp,4          ;allocate local variables
       push        ax
       mov         ax,[bp+4]     ;parameter var2
       mov         [bp-2],ax     ;local variable
       mov         ax,[bp+6]     ;parameter var1
       mov         [bp-4],ax     ;local variable
       pop         ax
       mov         sp,bp
       pop         bp
       ret         4             ;return & clean-up stack
junk   endp

main:  push var1
       push  var2
       call   junk
.
.
```

Stack:

Initially: (ss)=F000, (sp)=0010



F0010		
F000E		[BP+6]
F000C		[BP+4]
F0008		[BP+2]
F0006		[BP]
F0004		[BP-2]
F0002		[BP-4]
F0000		

C-Language Interfacing

- C-Language passes parameters to a procedure on the stack from right-to-left order
- The calling program is responsible of cleaning up the stack
- The procedure is free to modify the following registers without preserving: *AX*, *CX*, *DX*.
- Values are returned in the following registers:

Returned Data Type	Register
Byte	AL
Word	AX
Double Word	DX:AX

EXAMPLE: Calling ASM from C

```
_add  proc   near
      push   bp
      mov    bp,sp
      mov    ax,[bp+4]
      add    ax,[bp+6]
      pop    bp
      ret
_add  endp

_sub  proc   near
      push   bp
      mov    bp,sp
      mov    ax,[bp+4]
      sub    ax,[bp+6]
      pop    bp
      ret
_sub  endp
```

```
void main()
{
    total1 =_add(1,2);
    …
    total2 =_sub(3,4)
}
```

EXAMPLE: Calling C from ASM

```
int _add(int a, int b)
{
     return a + b;
}


int _sub(int a, int b)

{
     return a - b;
}
```

```
mov   ax,2
push  ax
mov   ax,1
push  ax
call  _add
add sp,4
mov  total1,ax

          .
          .
mov   ax,4
push  ax
mov   ax,3
push  ax
call  _sub
add sp,4
mov  total2,ax
          .
          .
```