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

<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic and Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push onto Stack</td>
<td>push src</td>
<td>(sp)←(sp)-2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(sp)←(src)</td>
</tr>
<tr>
<td>Pop from Stack</td>
<td>pop dst</td>
<td>(dst)←((sp))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(sp)←(sp)+2</td>
</tr>
<tr>
<td>Push Flags</td>
<td>pushf</td>
<td>(sp)←(sp)-2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(sp)←(psw)</td>
</tr>
<tr>
<td>Pop Flags</td>
<td>popf</td>
<td>(psw)←((sp))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(sp)←(sp)+2</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Stack:</th>
<th>Initially: (ss) = F000, (sp)=0008</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>F0010</td>
<td>pushf</td>
</tr>
<tr>
<td>F000E</td>
<td>mov ax,2211h</td>
</tr>
<tr>
<td>F000C</td>
<td>push ax</td>
</tr>
<tr>
<td>F000A</td>
<td>add ax,1111h</td>
</tr>
<tr>
<td>F0008</td>
<td>push ax</td>
</tr>
<tr>
<td>F0006</td>
<td></td>
</tr>
<tr>
<td>F0004</td>
<td></td>
</tr>
<tr>
<td>F0002</td>
<td></td>
</tr>
<tr>
<td>F0000</td>
<td>pop cx</td>
</tr>
<tr>
<td></td>
<td>pop ds</td>
</tr>
<tr>
<td></td>
<td>popf</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic and Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intrasegment Direct Call</td>
<td>call opr</td>
<td>(sp)←(sp)-2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>((sp))←(ip)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(ip)←(ip)+16-bit Disp.</td>
</tr>
<tr>
<td>Intrasegment Indirect Call</td>
<td>call opr</td>
<td>(sp)←(sp)-2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>((sp))←(ip)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(ip)←(Eff. Addr.)</td>
</tr>
<tr>
<td>Intersegment Direct Call</td>
<td>call opr</td>
<td>(sp)←(sp)-2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>((sp))←(cs)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(sp)←(sp)-2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>((sp))←(ip)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(ip)←16-bit Disp.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(cs)←Segment Address</td>
</tr>
<tr>
<td>Intersegment Indirect Call</td>
<td>call opr</td>
<td>(sp)←(sp)-2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>((sp))←(cs)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(sp)←(sp)-2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>((sp))←(ip)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(ip)←(Eff. Addr.)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(cs)←(Eff. Addr. + 2)</td>
</tr>
<tr>
<td>Intrasegment Return</td>
<td>ret</td>
<td>(ip)←((sp))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(sp)←(sp)+2</td>
</tr>
<tr>
<td>Intrasegment Return with immediate data</td>
<td>ret expression</td>
<td>(ip)←((sp))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(sp)←(sp)+2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(sp)←(sp)+expression</td>
</tr>
<tr>
<td>Intrasegment Return</td>
<td>ret</td>
<td>(ip)←((sp))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(sp)←(sp)+2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(cs)←((sp))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(sp)←(sp)+2</td>
</tr>
<tr>
<td>Intrasegment Return with immediate data</td>
<td>ret expression</td>
<td>(ip)←((sp))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(sp)←(sp)+2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(cs)←((sp))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(sp)←(sp)+2</td>
</tr>
</tbody>
</table>
(sp)←(sp)+expression

Flags: Not affected.

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

.MODEL MEDIUM
.DAT
   VECTOR1 DW ACTION1
   VECTOR2 DD ACTION2
.CODE

ACTION1 PROC NEAR
   ...
   ...
   RET
ACTION1 ENDP

ACTION2 PROC FAR
   ...
   ...
   RET
ACTION2 ENDP

START:
   ...
   ...
   ;INTRASECTION DIRECT
   CALL ACTION1
   ...
   ...
   ;INTRASECTION INDIRECT
   CALL VECTOR1
   ...
   ...
   ;INTERSECTION DIRECT
   CALL ACTION2
   ...
   ...
   ;INTERSECTION INDIRECT
   CALL VECTOR2
   ...
   ...
END START
### Exercise: Fill-in the Stack

#### Stack:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>(ss) = F000h, (sp)=0012h, (cs)=2000h, done=6050h</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F0022</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F0020</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F001E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F001C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F001A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F0018</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F0016</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F0014</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F0012</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F0010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F000E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F000C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F000A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F0008</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F0006</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F0004</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F0002</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F0000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- \( \text{mov} \ ax, 2211h \)
- \( \text{push} \ ax \)
- \( \text{call} \ \text{junk} \)

#### done: mov var1, ax

- \( (cs)=3000h, \text{junk}=8000h \)

- \( \text{junk proc far} \)
- \( \text{push} \ bp \)
- \( \text{push} \ bp \)
- \( \text{pop} \ bp \)
- \( \text{ret} 2 \)

- \( \text{junk endp} \)
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, etc.) 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:

```assembly
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

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

Stack:

- [bp+6] offset var1
- [bp+4] offset var2
- [bp+2] caller ip
- [bp] saved bp
- [bp-2] saved bx
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:

- [bp+6] offset Var1
- [bp+4] offset Var2
- [bp+2] caller ip
- [bp] saved bp
- [bp-2] saved bx
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**

```assembly
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:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0010</td>
<td></td>
</tr>
<tr>
<td>F000E</td>
<td>[BP+6]</td>
</tr>
<tr>
<td>F000C</td>
<td>[BP+4]</td>
</tr>
<tr>
<td>F0008</td>
<td>[BP+2]</td>
</tr>
<tr>
<td>F0006</td>
<td>[BP]</td>
</tr>
<tr>
<td>F0004</td>
<td>[BP-2]</td>
</tr>
<tr>
<td>F0002</td>
<td>[BP-4]</td>
</tr>
<tr>
<td>F0000</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Returned Data Type</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>AL</td>
</tr>
<tr>
<td>Word</td>
<td>AX</td>
</tr>
<tr>
<td>Double Word</td>
<td>DX:AX</td>
</tr>
</tbody>
</table>
EXAMPLE: Calling ASM from C

```assembly
_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
```

```c
void main()
{
    total1 = _add(1,2);
    ...
    total2 = _sub(3,4)
}
```
EXAMPLE: Calling C from ASM

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

int _sub(int a, int b)
{
    return a - b;
}
```

```assembly
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
```