Lab 8: *Advanced Procedures*	**Session:** Fall 2018
Instructor(s): Sumaiyah Zahid & Fahim Ahmed	

Stack Applications

• A stack makes a convenient temporary save area for registers when they are used for more than one purpose. After they are modified, they can be restored to their original values.
• When the CALL instruction executes, the CPU saves the current subroutine's return address on the stack.
• When calling a subroutine, you pass input values called arguments by pushing them on the stack.
• The stack provides temporary storage for local variables inside subroutines.

Stack Parameters

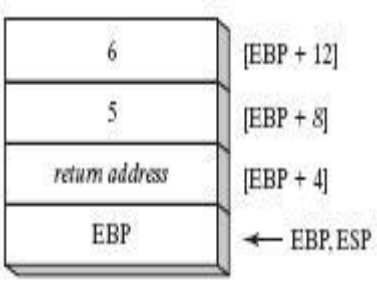
1. Pass by value

When an argument is passed by value, a copy of the value is pushed on the stack.

Example:

```
.data
        var1    DWORD       5
        var2    DWORD       6
.code
main PROC
        push var2
        push var1
        call AddTwo
        call DumpRegs
        exit
main ENDP

AddTwo PROC
        push    ebp
        mov     ebp, esp
        mov     eax, [ebp  + 12]
        add     eax, [ebp + 8]
        pop     ebp
        ret
AddTwo ENDP
END main
```



2. Explicit stack parameters

When stack parameters are referenced with expressions such as [ebp+8], we call them explicit stack parameters.

Example:

```
.data
        var1    DWORD       5
        var2    DWORD       6
        y_param         EQU     [ebp + 12]
        x_param         EQU     [ebp+ 8]

.code
        push var2
        push var1
```

```
        call AddTwo
        exit

AddTwo PROC
        push    ebp
        mov     ebp, esp
        mov     eax, y_param
        add     eax, x_param
        pop     ebp
        ret
AddTwo ENDP
END main
```

3. Pass by reference

An argument passed by reference consists of the offset of an object to be passed.

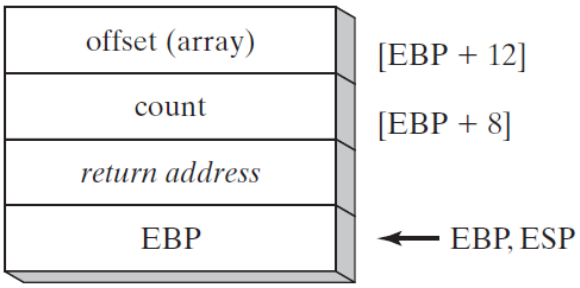
Example:

```
.data
        count = 10
        arr WORD  count DUP (?)

.code
        push    OFFSET arr
        push    count
        call    ArrayFill
exit

ArrayFill       PROC
        push    ebp
        mov     ebp, esp
        pushad
        mov     esi, [ebp + 12]
        mov     ecx, [ebp + 8]
        cmp     ecx, 0
        je      L2

L1:
        mov     eax, 100h
        call    RandomRange
        mov     [esi], ax
        add     esi, TYPE WORD
        loop    L1
L2:
        popad
        pop     ebp
        ret     8
ArrayFill       ENDP
```



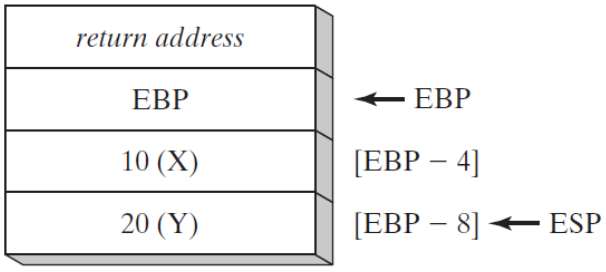
Local Variables

In MASM Assembly Language, local variables are created at runtime stack, below the base pointer (EBP).

Example:

```
.code
        call    MySub
        exit

MySub           PROC
        push    ebp
```



```
        mov     ebp, esp
        sub     esp, 8
        mov     DWORD       PTR [ebp - 4],   10      ; first parameter
        mov     DWORD       PTR [ebp - 8],   20      ; second parameter
        mov     esp, ebp
        pop     ebp
        ret
MySub   ENDP
```

LEA Instruction

LEA instruction returns the effective address of an indirect operand. Offsets of indirect operands are calculated at runtime.

Example:

```
.code
        call    makeArray
        exit

makeArray   PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 32
        lea     esi, [ebp - 30]
        mov     ecx,30
L1:
        mov     BYTE PTR [esi], '*'
        inc     esi
        loop    L1
        add     esp, 32
        pop     ebp
        ret
makeArray   ENDP
```

ENTER & LEAVE Instructions

Enter instruction automatically creates stack frame for a called Procedure. Leave instruction reverses the effect of enter instruction.

Example:

```
.data
        var1    DWORD       5
        var2    DWORD       6

.code
        push var2
        push var1
        call AddTwo
        exit


AddTwo PROC
        enter   0, 0
        mov     eax, [ebp  + 12]
        add     eax, [ebp + 8]
        leave
        ret
AddTwo ENDP
```

LOCAL Directive

LOCAL directive declares one or more local variables by name, assigning them size attributes.

Example:

```
.code
        call LocalProc
        exit


LocalProc       PROC
        LOCAL temp : DWORD
        mov     temp, 5
        mov     eax, temp
        ret
        LocalProc       ENDP
```

Recursive Procedures

Recursive procedures are those that call themselves to perform some task.

Example:

```
        .code
                mov     ecx, 5
                mov     eax, 0
                call    CalcSum
        L1:
                call    WriteDec
                call    crlf
                exit
        CalcSum         PROC
                cmp     ecx, 0
                jz      L2
                add     eax, ecx
                dec     ecx
                call    CalcSum
        L2:
                ret
        CalcSum         ENDP
```

Invoke Directive

```
INVOKE procedureName [, argumentList]

  INVOKE DumpArray, OFFSET array, LENGTHOF array, TYPE array
```

Proc Directive

```
label PROC [attributes] [USES reglist],
     parameter_1,
     parameter_2,
     .
     .
     parameter_n
```

ADDR Directive

```
INVOKE FillArray, ADDR myArray
```

Using INVOKE and PROTO

```
; (sum.inc)
INCLUDE Irvine32.inc

PromptForIntegers PROTO,
     ptrPrompt:PTR BYTE,          ; prompt string
     ptrArray:PTR DWORD,          ; points to the array
     arraySize:DWORD              ; size of the array

ArraySum PROTO,
     ptrArray:PTR DWORD,          ; points to the array
     arraySize:DWORD              ; size of the array

DisplaySum PROTO,
     ptrPrompt:PTR BYTE,          ; prompt string
     theSum:DWORD                 ; sum of the array

TITLE Integer Summation Program (Sum_main.asm)

INCLUDE sum.inc
Count = 3
.data
prompt1 BYTE "Enter a signed integer: ",0
prompt2 BYTE "The sum of the integers is: ",0
array   DWORD  Count DUP(?)
sum     DWORD  ?

.code
main PROC

     call Clrscr

     INVOKE PromptForIntegers, ADDR prompt1, ADDR array, Count
     INVOKE ArraySum, ADDR array, Count
     mov   sum,eax
     INVOKE DisplaySum, ADDR prompt2, sum

     call  Crlf
     exit
 main ENDP
 END main
```

Activity:

1. Write a program which contains a procedure named *ThreeProd* that displays the product of three numeric parameters passed through a stack.

2. Write a program which contains a procedure named *MinMaxArray* that displays the minimum & maximum values in an array. Pass a size-20 array by reference to this procedure.

3. Write a program which contains a procedure named *LocalSquare*. The procedure must declare a local variable. Initialize this variable by taking an input value from the user and then display its square. Use ENTER& LEAVE instructions to allocate and de-allocate the local variable.

4. Write a program that calculates factorial of a given number n. Make a recursive procedure named Fact that takes n as an input parameter.

5. Write a non-recursive version of the procedure Fact that uses a loop to calculate factorial of given number n. Compare efficiency of both versions of the Fact procedure using *GetMSeconds*.

6. Write a program to take 4 input numbers from the users. Then make two procedures *CheckPrime* and *LargestPrime*. The program should first check if a given number is a prime number or not. If all of the input numbers are prime numbers then the program should call the procedure *LargestPrime*.

CheckPrime: This procedure tests if a number is prime or not
LargestPrime: This procedure finds and displays the largest of the four prime numbers.