Lab: 14

Department of Computer Science

Iqra University Islamabad

Computer Organization and Assembly Language

Maqsood Ahmed

ID: 38186

7.1.4 Lab Work: Demonstrating the Stack Instructions

```
TITLE Demonstrating Stack Instructions  (stack.asm)

.686
.MODEL flat, stdcall
.STACK 4096
INCLUDE Irvine32.inc
 .data
var1    DWORD   01234567h
var2    DWORD   89ABCDEFh
 .code main
PROC
    pushad  ; Save general-purpose registers

    ;  PUSH  and  POP
push var1
push var2
push        6A6A4C4Ch
pop  eax
pop  ebx
pop  cx
pop  dx

    popad   ; restore general-purpose registers

    ; Exchanging  2  variables  in  memory
push var1      push var2      pop   var1
pop  var2
     exit
main ENDP
END main
```

Analyze the above program and guess the values of the eax, ebx, cx, and dx registers after executing the pop dx instruction. Write these values in hexadecimal in the shown boxes:

EAX (hex) = 6A6A4C4C	EBX (hex) = 89ABCDEF
CX (hex) = 01234567	DX (hex) = Orginal value of dx

Also guess and write the values of var1 and var2 after executing the pop var2 instruction:

var1 (hex) = 89ABCDEF	var2 (hex) = 01234567

7.2.2 Demonstrating Procedures

```
TITLE Demonstrating Procedures  (procedure.asm)

.686
.MODEL flat, stdcall
.STACK 4096
INCLUDE Irvine32.inc

.data
 .code
main PROC
mov  eax, 9876
mov  ebx, 12
mov  ecx, -5
call sort3
exit
main ENDP

; Sorts 3 integers in EAX, EBX, and ECX
sort3 PROC
cmp  eax, ebx
jle  L1
call swapAB
L1:
cmp  eax, ecx
jle  L2
call swapAC
L2:
cmp  ebx, ecx
jle  L3
call swapBC
L3:
ret
sort3 ENDP

; Swaps the values of the EAX and EBX registers
swapAB PROC
Push eax
Push ebx
Pop  eax
Pop  ebx
ret

swapAB ENDP

; Swaps the values of the EAX and ECX registers
swapAC PROC
Push eax
Push ecx
Pop  eax
Pop  ecx
ret
swapAC ENDP
```

```
; Swaps the values of the EBX and ECX registers
swapBC PROC
Push ebx
Push ecx
Pop  ebx

Pop  ecx

ret

swapBC    ENDP
END main
```

Summary of Return Addresses and Stack Locations

1. **Return address of call sort3**: 0040101A
    - Stack address**: 12ffc0**
2. **Return address of call swapAB**: 00401030`
    - Stack address**: 12ffbc**
3. **Return address of call swapAC**: 0040103A`
    - Stack address**: 12ffb8**
4. **Return address of call swapBC**: 00401044`
    - Stack address**: 12ffb4**

Review Questions

1. **(True/False) The push instruction decreases the esp register and pop increases it.**
   True. The `push` instruction decreases the `esp` register, allocating space on the stack, while the `pop` instruction increases the `esp` register, deallocating space from the stack.

2. How does the call instruction work?

The `call` instruction saves the address of the next instruction (return address) onto the stack and then transfers control to the target function or procedure. The return address is stored on the stack to allow the function to return to the correct location in the code after execution.

3. How does the ret n instruction work (where n is an integer constant)?

The `ret n` instruction pops the return address from the stack and transfers control to that address. Additionally, it adds the integer constant `n` to the `esp` register, effectively cleaning up `n` bytes from the stack. This is often used for cleaning up parameters pushed onto the stack by the caller.

4. Why is it better to use the ebp than the esp register to locate parameters on the stack?

The `ebp` register is used as a stable base pointer to access function parameters and local variables. Unlike `esp`, which changes frequently during function execution (due to push/pop operations), `ebp` remains constant if properly set up at the beginning of a function. This makes it easier and safer to access parameters and local variables.

5. Which instruction should be used to allocate local variables on the stack?

The `sub` instruction is typically used to allocate local variables on the stack. For example, `sub esp, 16` allocates 16 bytes of space on the stack.

6. How does the leave instruction work?

The `leave` instruction is used to clean up the stack frame before returning from a function. It performs two operations: it copies the value of `ebp` into `esp`, effectively deallocating local variables, and then pops the top value of the stack into `ebp`, restoring the previous base pointer. This is typically used in conjunction with the `ret` instruction.

7. What is the use of an INVOKE directive, and how is it translated?

The `INVOKE` directive is used in assembly language to call a procedure with parameters. It simplifies the calling process by automatically generating the necessary push instructions for each argument and issuing the call instruction. It translates into a series of `push` instructions followed by a `call` instruction.

8. What is the use of a LOCAL directive, and how is it translated?

The `LOCAL` directive is used to declare local variables within a procedure. It is translated into adjustments to the `esp` register to allocate space for the local variables on the stack at the beginning of the procedure, and adjustments to `esp` to deallocate that space at the end of the procedure.

9. What is the use of a USES directive and how is it translated?

The `USES` directive specifies which registers will be preserved (pushed onto the stack) and restored (popped off the stack) within a procedure. It simplifies the task of saving and restoring register states, ensuring that the specified registers are not altered by the procedure. It is translated into `push` instructions at the beginning and `pop` instructions at the end of the procedure.

Programming Exercises

1. Procedure to fill an array with random integers:

```
; Procedure to fill an array with random integers (0-999)
FillArray PROC
    push ebp
    mov ebp, esp
    push edi

    mov edi, eax          ; EDI points to the start of the array
    mov ecx, [ebp+8]      ; ECX contains the count of elements

    fill_loop:
        INVOKE RandomRange, 1000
        mov [edi], eax   ; Store the random integer in the array
        add edi, 4       ; Move to the next element
        loop fill_loop

    pop edi
    pop ebp
    ret
FillArray ENDP
```

2. Procedure to display an array of integers:

```
; Procedure to display an array of integers
DisplayArray PROC
   push ebp
   mov ebp, esp
   push edi

   mov edi, [ebp+8]     ; EDI points to the start of the array
   mov ecx, [ebp+12]    ; ECX contains the count of elements

   display_loop:
      mov eax, [edi]   ; Load the current element
      call WriteInt    ; Display the integer
      call Crlf        ; Newline for better readability
      add edi, 4       ; Move to the next element
      loop display_loop

   pop edi
   pop ebp
   ret
DisplayArray ENDP
```

3. Procedure to sort an array of integers (Bubble Sort):

```
; Procedure to sort an array of integers using Bubble Sort
SortArray PROC
    push ebp
    mov ebp, esp
    push esi
    push edi

    mov esi, [ebp+8]     ; ESI points to the start of the array
    mov ecx, [ebp+12]    ; ECX contains the count of elements
    dec ecx              ; ECX is now the index of the last element

    sort_outer:
        mov edi, 0       ; Start inner loop from the beginning
        mov ebx, ecx     ; Set the limit for the inner loop
        sort_inner:
            mov eax, [esi + edi*4]       ; Load current element
            cmp eax, [esi + edi*4 + 4]   ; Compare with the next element
            jle no_swap
            ; Swap elements if out of order
            mov edx, [esi + edi*4 + 4]
            mov [esi + edi*4 + 4], eax
            mov [esi + edi*4], edx

        no_swap:
            inc edi                      ; Move to the next element
```

```
               cmp edi, ebx
               jl sort_inner

           dec ecx                         ; Reduce the limit for the next pass
           cmp ecx, 0
           jg sort_outer

       pop edi
       pop esi
       pop ebp
       ret
   SortArray ENDP
```

Main Procedure to Test the Procedures

```
main PROC
    ; Allocate space for the array (10 elements)
    sub esp, 40
    mov eax, esp         ; EAX points to the array
    mov ecx, 10          ; ECX contains the number of elements
    call FillArray       ; Fill the array with random numbers

    ; Display the unsorted array
    push ecx
    push eax
    call DisplayArray
    ; Sort the array
    push ecx
    push eax
    call SortArray

    ; Display the sorted array
    push ecx
    push eax
    call DisplayArray

    add esp, 40          ; Clean up the allocated space
    exit
main ENDP
END main
```

Explanation:

FillArray: Fills an array with random integers between 0 and 999.

DisplayArray: Displays an array of integers.

SortArray: Sorts an array of integers using the Bubble Sort algorithm.

Main Procedure: Calls the above procedures to demonstrate their functionality.

THE END