National University of computer and emerging sciences

Computer Organization and Assembly Language (EL-2003)

Semester: Fall 2025
Section: Section: BSSE-3D / BSSE-3H
Course Instructor: Muhammad Owais

LAB # 08

Stack Operations & Procedures

Lab Objectives:

By the end of this lab, students will be able to:

- Understand the concept and working of the Runtime Stack in assembly language and its role in function calls.
- Learn the purpose and functionality of the PUSH instruction for storing data temporarily on the stack.
- Explore the usage of the POP instruction to retrieve data from the stack efficiently.
- Understand the PROC directive and its significance in defining modular procedures in assembly language.
- Implement CALL and RET instructions to control program flow and manage procedure calls and returns.
- Develop and execute programs using nested procedures, demonstrating proper stack management and parameter passing.
- Enhance problem-solving and logical thinking skills through structured and modular assembly programming.

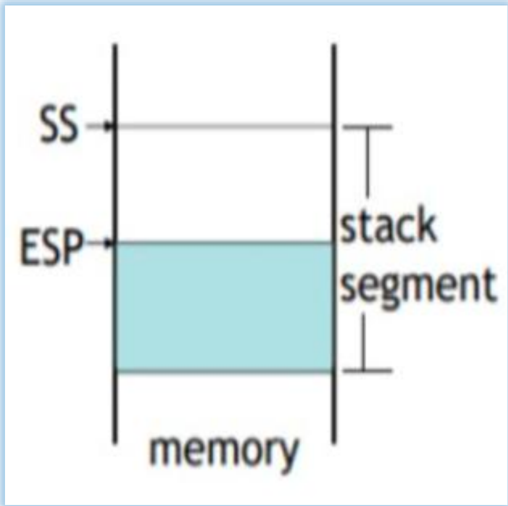
STACK

- LIFO (Last-In, First-Out) data structure.
- PUSH/ POP operations
- You probably have had experiences on implementing it in high-level languages.
- Here, we concentrate on runtime stack, directly supported by hardware in the
- CPU. It is essential for calling and returning from procedures.

RUNTIME STACK

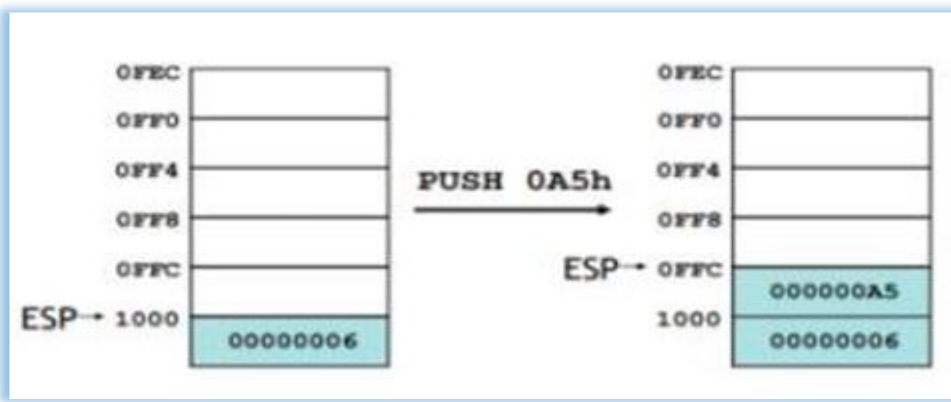
Managed by the CPU, using two registers

- SS (stack segment)
- ESP (stack pointer): point the last value to be added to, or pushed on, the top of stack usually modified by instructions:
    - **CALL, RET, PUSH and POP**



PUSH OPERATION

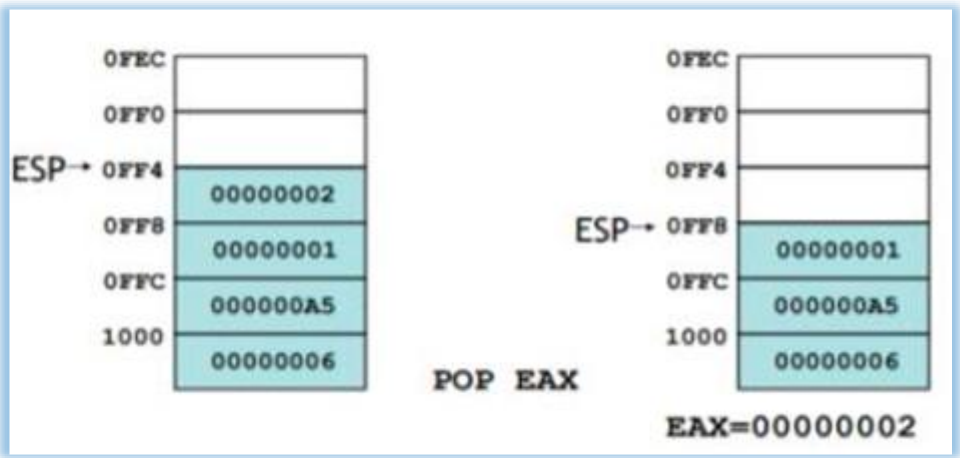
Push Operation A 32-bit push operation decrements the stack pointer by 4 and copies a value into the location in the stack pointed to by the stack pointer.



The same stack after pushing two more integers:

POP OPERATION

Pop Operation A pop operation removes a value from the stack. After the value is popped from the stack, the stack pointer is incremented (by the stack element size) to point to the next- highest location in the stack. It copies value at stack [ESP] into a register or variable



PUSH and POP INSTRUCTIONS

PUSH syntax:

- **PUSH r/m16**
- **PUSH r/m32**
- **PUSH imm32**

POP syntax:

- **POP r/m16**
- **POP r/m32**

EXAMPLE # 01:

STACK AND NESTED LOOPS

```
INCLUDE Irvine32.inc
.data
msg1 BYTE "Example: Stack and Nested Loops", 0
msg2 BYTE "Final value of EBX after nested loops: ", 0

.code
main PROC
    mov ebx, 0
    mov ecx, 5
L1:
    push ecx
    mov ecx, 10
L2:
    inc ebx
    loop L2

    pop ecx
    loop L1

    call Crlf
    mov edx, OFFSET msg1
    call WriteString
    call Crlf
    mov edx, OFFSET msg2
    call WriteString
    mov eax, ebx
    call WriteDec
    call Crlf

    exit
main ENDP
END main
```

Microsoft Visual Studio Debug Console

```
Example: Stack and Nested Loops
Final value of EBX after nested loops: 50

C:\Users\Muhammad Nadeem\source\repos\Sunday\Debug\Sunday.exe (process 7916) exited with cod
e 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->A
utomatically close the console when debugging stops.
Press any key to close this window . . .
```

EXAMPLE # 02:

HOW VALUES ARE TEMPORARILY STORED AND RETRIEVED FROM THE STACK DURING ARITHMETIC OPERATIONS.

You are a **Software Engineering student** developing a calculator program that needs to temporarily store intermediate results during arithmetic computations.
Write an Assembly program that:

1. Pushes two numbers onto the stack.
2. Pops them back one by one.
3. Adds them and displays the sum.
4. Again pushes the result, then pops and displays it as the **final stored result**.

```
INCLUDE Irvine32.inc
.data
num1 WORD 15
num2 WORD 25
msg1 BYTE "Initial Numbers: ", 0
msg2 BYTE "Sum after POP (A + B): ", 0
msg3 BYTE "Final Result after PUSH & POP: ", 0
space BYTE " ", 0

.code
main PROC
    call Crlf
    mov edx, OFFSET msg1
    call WriteString
    call Crlf

    movzx eax, num1
    call WriteDec
    mov edx, OFFSET space
    call WriteString
    movzx eax, num2
    call WriteDec
    call Crlf

    mov ax, num1
    push ax
    mov ax, num2
    push ax

    pop bx
    pop ax
    add ax, bx

    call Crlf
    mov edx, OFFSET msg2
    call WriteString
    movzx eax, ax
    call WriteDec
    call Crlf

    push ax
    pop bx

    call Crlf
    mov edx, OFFSET msg3
    call WriteString
    movzx eax, bx
    call WriteDec
    call Crlf

    exit
main ENDP
END main
```

```
Initial Numbers:
15 25

Sum after POP (A + B): 40

Final Result after PUSH & POP: 40

C:\Users\Muhammad Nadeem\source\repos\Sunday\Debug\S
```

EXAMPLE # 03:

DEMONSTRATES HOW STACK-BASED MEMORY MANAGEMENT SUPPORTS REAL-WORLD FEATURES LIKE UNDO/REDO OR VERSION ROLLBACK.
REINFORCES UNDERSTANDING OF LAST-IN, FIRST-OUT (LIFO) PRINCIPLE CRUCIAL FOR RECURSION, FUNCTION CALLS, AND BACKTRACKING ALGORITHMS.

- Store 5 version numbers in an array (representing software updates).
- Push all version numbers onto the stack (latest version last).
- Pop them into another array in reverse order, simulating a rollback feature.
- Display both arrays to show before and after rollback versions.

```
INCLUDE Irvine32.inc
.data
versionHistory  WORD 101, 102, 103, 104, 105
rollbackHistory WORD 5 DUP(?)
msg1 BYTE "Version History (Latest Last):", 0
msg2 BYTE "Rollback Order (After Using PUSH & POP):", 0
space BYTE " ", 0

.code
main PROC
    call    Crlf
    mov     edx, OFFSET msg1
    call    WriteString
    call    Crlf

```

Display original version history

```
    mov     ecx, LENGTHOF versionHistory
    mov     esi, OFFSET versionHistory
display_original:
    movzx   eax, WORD PTR [esi]
    call    WriteDec
    mov     edx, OFFSET space
    call    WriteString
    add     esi, TYPE versionHistory
    loop    display_original
```

Push all version numbers to stack

```
    mov     ecx, LENGTHOF versionHistory
    mov     esi, OFFSET versionHistory
push_versions:
    mov     ax, [esi]
    push    ax
    add     esi, TYPE versionHistory
    loop    push_versions

```

Pop all version numbers in reverse order

```
    mov     ecx, LENGTHOF rollbackHistory
    mov     edi, OFFSET rollbackHistory
pop_versions:
    pop     ax
    mov     [edi], ax
    add     edi, TYPE rollbackHistory
    loop    pop_versions

    call    Crlf
    mov     edx, OFFSET msg2
    call    WriteString
    call    Crlf
```

Display rollback array (reversed)

```
    mov     ecx, LENGTHOF rollbackHistory
    mov     esi, OFFSET rollbackHistory
display_rollback:
    movzx   eax, WORD PTR [esi]
    call    WriteDec
    mov     edx, OFFSET space
    call    WriteString
    add     esi, TYPE rollbackHistory
    loop    display_rollback

    call    Crlf
    exit
main ENDP
END main
```

```
Version History (Latest Last):
101 102 103 104 105
Rollback Order (After Using PUSH & POP):
105 104 103 102 101

C:\Users\Muhammad Nadeem\source\repos\Sunday
```

EXAMPLE # 04:

THIS PROGRAM PUSHES THREE INTEGERS ONTO THE STACK AND THEN POPS THEM TO COMPUTE THEIR PRODUCT.

```
INCLUDE Irvine32.inc
.data
    multp DWORD 2
    msg BYTE "Hello Fastian! The product of the three integers is: ", 0
.code
main PROC
    mov eax, 1
    mov ecx, 3
PushLoop:
    push multp
    add multp, 2
    loop PushLoop
    mov ecx, 3
MultiplyLoop:
    pop ebx
    mul ebx
    loop MultiplyLoop
    mov edx, OFFSET msg
    call WriteString
    call WriteDec
    call CrLf
    exit
main ENDP
END main
```

Select Microsoft Visual Studio Debug Console

```
Hello Fastian! The product of the three integers is: 48

C:\Users\Muhammad Nadeem\source\repos\Sunday\Debug\Sunday.exe (process 7872) exited with code 0.
```

EXAMPLE # 05:

TO FIND THE LARGEST NUMBER THROUGH A STACK.

```
INCLUDE Irvine32.inc
.data
    msg BYTE "Hello Fastian! The largest number is:", 0
.code
main PROC
    push 5
    push 7
    push 3
    push 2
    mov eax, 0
    mov ecx, 4
L1:
    pop edx
    cmp edx, eax
    jle Next
    mov eax, edx
Next:
    loop L1
    mov edx, OFFSET msg
    call WriteString
    call WriteDec
    call CrLf
    exit
main ENDP
END main
```

Select Microsoft Visual Studio Deb...

```
Hello Fastian! The largest number is:7

C:\Users\Muhammad Nadeem\source\repos\Sund
ay\Debug\Sunday.exe (process 13316) exited
```

PUSHFD and POPFD INSTRUCTIONS

The MOV instruction cannot be used to copy the flags to a variable.

The PUSHFD instruction pushes the 32-bit EFLAGS register on the stack, and POPFD pops the stack into EFLAGS:

PUSHFD

POPFD

EXAMPLE # 06:

TO FIND THE LARGEST NUMBER THROUGH A STACK.

```
INCLUDE Irvine32.inc
.data
    msg1 BYTE "Original Flags saved on stack.", 0
    msg2 BYTE "Flags restored from stack.", 0
.code
main PROC
    mov eax, 5
    sub eax, 5
    pushfd
    mov edx, OFFSET msg1
    call WriteString
    call CrLf
    mov eax, 10
    add eax, 1
    popfd
    mov edx, OFFSET msg2
    call WriteString
    call CrLf
    call DumpRegs
    exit
main ENDP
END main
```

Microsoft Visual Studio Debug Console

```
Original Flags saved on stack.
Flags restored from stack.

  EAX=0000000B  EBX=00C1F000  ECX=001A10AA  EDX=001A601F
  ESI=001A10AA  EDI=001A10AA  EBP=00EFFB34  ESP=00EFFB28
  EIP=001A3695  EFL=00000202  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=0
```

Explanation:

1. After sub eax, 5, Zero Flag (ZF) becomes 1.

2. PUSHFD saves this flag state to the stack.

3. Some operations are performed that modify flags (e.g., add eax, 1).

4. POPFD restores the original flags from the stack — so ZF=1 again, exactly as before the modification.

PUSHFD and POPFD are used to preserve and restore the CPU's status flags, ensuring that temporary operations don't unintentionally affect conditional logic or program flow.

PROCEDURES

- Procedures or subroutines are very important in assembly language, as the assembly language programs tend to be large in size.
- Procedures are identified by a name. Following this name, the body of the procedure is described which performs a well-defined job.
- End of the procedure is indicated by a return statement.

PROC Directive

We can define a procedure as a named block of statements that ends in a return statement. A procedure is declared using the **PROC** and **ENDP** directives. It must be assigned a name (a valid identifier). When we create a procedure other than your program's startup procedure, end it with a **RET** instruction. **RET** forces the CPU to return to the location from where the procedure was called:

Let us say, sample is the name of procedure.

```
sample PROC
      .
      .
      ret
sample ENDP
```

The procedure is called from another function by using the CALL instruction. The CALL instruction should have the name of the called procedure as an argument as shown below

CALL Sample

The called procedure returns the control to the calling procedure by using the RET instruction.

Call & RET INSTRUCTIONS

CALL instruction is used whenever we need to make a call to some procedure or a subprogram. Whenever a **CALL** is made, the following process takes place inside the microprocessor:

- The address of the next instruction that exists in the caller program (after the program **CALL** instruction) is stored in the stack.
- The instruction queue is emptied for accommodating the instructions of the procedure. Then, the contents of the instruction pointer (IP) is changed with the address of the first instruction of the procedure.
- The subsequent instructions of the procedure are stored in the instruction queue for execution.
- The Syntax for the CALL instruction is mentioned above.

RET instruction stands for return. This instruction is used at the end of the procedures or the subprograms. This instruction transfers the execution to the caller program. Whenever the RET instruction is called, the following process takes place inside the microprocessor:

- The address of the next instruction in the mainline program which was previously stored inside the stack is now again fetched and is placed inside the instruction pointer (IP).
- The instruction queue will now again be filled with the subsequent instructions of the mainline program.

EXAMPLE # 07:

ADD TWO NUMBERS USING PROCEDURES

```
INCLUDE Irvine32.inc
.data
    var1 DWORD 5
    var2 DWORD 6
    msg  BYTE "The sum of the two numbers is: ", 0
.code
main PROC
    call AddTwo
    mov edx, OFFSET msg
    call WriteString
    call WriteDec
    call CrLf
    exit
main ENDP

AddTwo PROC
    mov eax, var1
    add eax, var2
    ret
AddTwo ENDP
END main
```

```
Select Microsoft Visual Studio ...
The sum of the two numbers is: 11

C:\Users\Muhammad Nadeem\source\repos\Su
```

EXAMPLE # 08:

THE SUM OF INTEGERS USING PROCEDURES

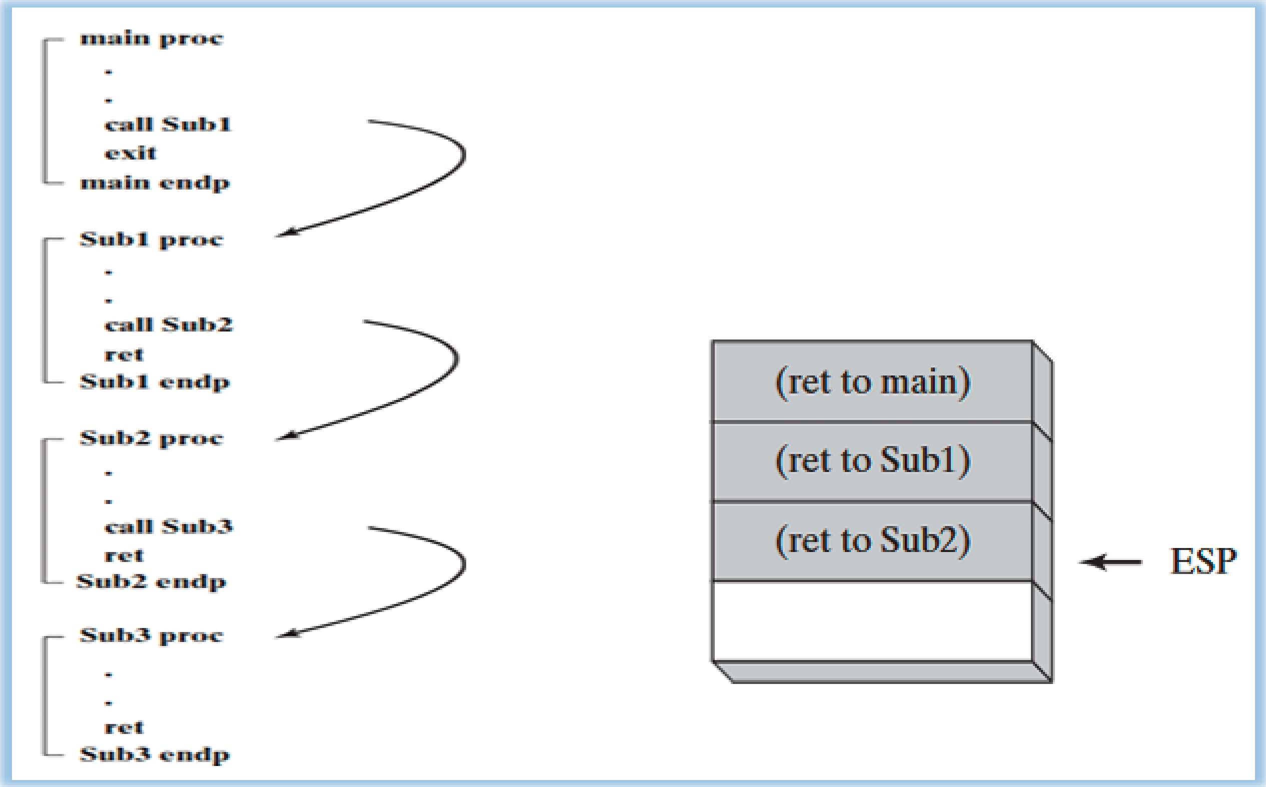
```
INCLUDE Irvine32.inc

INTEGER_COUNT = 3

.data
    str1  BYTE "Enter a signed integer: ", 0
    str2  BYTE "The sum of the integers is: ", 0
    array DWORD INTEGER_COUNT DUP(?)

.code
main PROC
    call Clrscr
    mov esi, OFFSET array
    mov ecx, INTEGER_COUNT
    call PromptForIntegers
    call ArraySum
    call DisplaySum
    exit
main ENDP

PromptForIntegers PROC USES ecx edx esi
    mov edx, OFFSET str1
L1:
    call WriteString
    call ReadInt
    call Crlf
    mov [esi], eax
    add esi, TYPE DWORD
    loop L1
    ret
PromptForIntegers ENDP

ArraySum PROC USES esi ecx
    mov eax, 0
L1:
    add eax, [esi]
    add esi, TYPE DWORD
    loop L1
    ret
ArraySum ENDP
```

```
DisplaySum PROC USES edx
    mov edx, OFFSET str2
    call WriteString
    call WriteInt
    call Crlf
    ret
DisplaySum ENDP

END main
```

```
Enter a signed integer: 5

Enter a signed integer: 10

Enter a signed integer: 15

The sum of the integers is: +30

C:\Users\Muhammad Nadeem\source\repos\Sunday\De
```

NESTED PROCEDURE CALLS

A nested procedure call occurs when a called procedure calls another procedure before the first procedure returns.



EXAMPLE # 09:

HOW NESTED PROCEDURE CALLS WORK

A **nested procedure call** occurs when a **procedure calls another procedure** before returning to the procedure that called it.
Think of it like **one function calling another** before finishing its own work.

```
INCLUDE Irvine32.inc
.data
    var1 DWORD 5
    var2 DWORD 6
    msg1 BYTE "The sum calculated in AddTwo is: ", 0
    msg2 BYTE "Values printed inside AddTwo1:", 0
.code
main PROC
    call AddTwo
    call CrLf
    exit
main ENDP

AddTwo PROC
    mov eax, var1
    mov ebx, var2
    add eax, var2

    mov edx, OFFSET msg1
    call WriteString
    call WriteInt
    call CrLf

    call AddTwo1
    ret
AddTwo ENDP
```

```
AddTwo1 PROC
    mov ecx, var1
    mov edx, var2

    mov ebx, OFFSET msg2
    call WriteString
    call CrLf

    mov eax, ecx
    call WriteInt
    call CrLf

    mov eax, edx
    call WriteInt
    call CrLf
    ret
AddTwo1 ENDP
END main
```

```
The sum calculated in AddTwo is: +11

C:\Users\Muhammad Nadeem\source\repos\Sunday
\Debug\Sunday.exe (process 15276) exited wit
```

(How It Works):

1 main starts and calls AddTwo → return address is pushed on stack.
2 AddTwo adds var1 and var2, prints the sum, then calls AddTwo1 → another return address pushed.
3 AddTwo1 prints values of both variables, executes RET → returns to AddTwo.
4 AddTwo executes RET → returns to main.
5 main regains control, finishes execution, and exits.

Thank you ☺