**CHAPTER 5**

**Stack Operations**

**1. What is a Stack?**

* A **stack** is like a pile of plates where new items are added to the top and removed from the top.
* It follows the **LIFO (Last-In, First-Out)** principle.
* In programming, stacks are useful for managing data efficiently.

**2. Runtime Stack (32-Bit Mode)**

* The **runtime stack** is a memory area managed by the CPU.
* It is controlled using the **ESP (Extended Stack Pointer) register**.
* Instructions like CALL, RET, PUSH, and POP modify the stack.
* The **stack grows downward** in memory (from high to low addresses).
* The runtime stack differs from a **Stack ADT**, which is used in high-level programming.

**3. Push Operation**

* The **PUSH** instruction:
  + Decreases **ESP** by **4 bytes (in 32-bit mode)**.
  + Copies the value into the new location at the top of the stack.
* Example:
  + Before: ESP = 00001000h
  + After PUSH operation: ESP = 00000FFCh

**4. Pop Operation**

* The **POP** instruction:
  + Removes the top value from the stack.
  + Increases **ESP** by **4 bytes** to point to the next-highest value.
* The area below ESP is now **available for new values**.

**5. Uses of the Runtime Stack**

* **Saving Register Values**: Stores temporary values for registers.
* **Handling Function Calls**: The **CALL** instruction saves return addresses on the stack.
* **Passing Function Arguments**: Input values are pushed onto the stack.
* **Storing Local Variables**: The stack holds temporary variables used inside subroutines.

**6. PUSH and POP Instructions**

* **PUSH Instruction Formats**:
  + PUSH reg/mem16 (16-bit operand, decrements ESP by 2).
  + PUSH reg/mem32 (32-bit operand, decrements ESP by 4).
  + PUSH imm32 (pushes a 32-bit immediate value).
* **POP Instruction Formats**:
  + POP reg/mem16 (increments ESP by 2).
  + POP reg/mem32 (increments ESP by 4).

**7. PUSHFD and POPFD Instructions**

* PUSHFD saves the **EFLAGS register** on the stack.
* POPFD restores **EFLAGS** from the stack.
* Used to **backup and restore CPU flags** during execution.
* Example:

pushfd ; Save flags

; Some code execution

popfd ; Restore flags

**8. PUSHAD, PUSHA, POPAD, and POPA**

* **PUSHAD**: Saves all **32-bit registers** on the stack.
* **POPAD**: Restores all **32-bit registers** from the stack.
* **PUSHA** and **POPA**: Similar but used for **16-bit registers**.
* **Note:** Avoid using PUSHAD and POPAD in functions that return values in registers like **EAX**.

**9. Example: Reversing a String Using Stack**

* **Process:**
  1. Loop through the string, pushing each character onto the stack.
  2. Pop characters from the stack and place them back in the string.
  3. This reverses the string since the stack follows **LIFO**.
* **Code snippet:**

mov ecx, nameSize ; Load string length

mov esi, 0 ; Start index

L1:

movzx eax, aName[esi] ; Get character

push eax ; Push onto stack

**Section 5.1.3 Review Questions**

1. **Which register (in 32-bit mode) manages the stack?**
   * **ESP (Extended Stack Pointer)** is the register that manages the stack in 32-bit mode.
2. **How is the runtime stack different from the stack abstract data type?**
   * The **runtime stack** is used by the computer to keep track of function calls and local variables, while the **stack abstract data type** is a general concept of a LIFO (Last In, First Out) structure used in programming.
3. **Why is the stack called a LIFO structure?**
   * The stack is called a **LIFO (Last In, First Out) structure** because the last thing you put in (**push**) is the first thing you take out (**pop**), just like a stack of plates.
4. **When a 32-bit value is pushed on the stack, what happens to ESP?**
   * When a 32-bit value is **pushed** onto the stack, **ESP decreases by 4** because the stack grows downward in memory.
5. **(True/False): Local variables in procedures are created on the stack.**
   * **True** – Local variables in procedures are stored on the stack.
6. **(True/False): The PUSH instruction cannot have an immediate operand.**
   * **False** – The **PUSH** instruction **can** have an immediate operand (like a number).

**Assembly Language Procedures**

**5.2 Defining and Using Procedures**

* **Procedures** are like functions in other programming languages (C, C++, Java, etc.).
* They help break a large program into smaller, manageable tasks.
* Assembly language does not have built-in object-oriented features, so we create structure manually.

**5.2.1 PROC Directive**

* A **procedure** is a named block of instructions ending with a **RET** (return) statement.
* Declared using **PROC** (start) and **ENDP** (end).
* Example:
* main PROC
* ; Code here
* main ENDP
* When calling a procedure, it must end with **RET** to return control to the caller.
* Example:
* sample PROC
* ; Code here
* ret
* sample ENDP

**Labels in Procedures**

* Labels inside procedures are **only visible** within that procedure.
* **Global labels** (visible everywhere) can be created using **::** (double colon).
* Avoid jumping out of a procedure, as it can **corrupt the stack**.

**Example: Summing Three Integers**

SumOf PROC

add eax, ebx

add eax, ecx

ret

SumOf ENDP

* This procedure adds three 32-bit numbers stored in **EAX, EBX, and ECX** and returns the sum in **EAX**.

**Documenting Procedures**

Always add comments to describe:

1. **What the procedure does**.
2. **What inputs it receives** (e.g., Receives: EAX, EBX, ECX).
3. **What it returns** (e.g., Returns: EAX = sum).
4. **Any special conditions** (e.g., Requires: must be in graphics mode before calling).

**5.2.2 CALL and RET Instructions**

* **CALL**: Transfers execution to a procedure.
* **RET**: Returns control back to the caller.
* The CALL instruction:
  + Pushes the **return address** on the stack.
  + Jumps to the procedure.
* The RET instruction:
  + Pops the return address from the stack.
  + Resumes execution from where the procedure was called.

**Example: Calling a Procedure**

main PROC

call MySub ; Call procedure MySub

mov eax, ebx ; Continue execution after MySub

main ENDP

MySub PROC

mov eax, edx ; Some operation

ret ; Return to main

MySub ENDP

**5.2.3 Nested Procedure Calls**

* **Nested calls** happen when one procedure calls another before returning.
* Example:

main PROC

call Sub1

exit

main ENDP

Sub1 PROC

call Sub2

ret

Sub1 ENDP

Sub2 PROC

call Sub3

ret

Sub2 ENDP

Sub3 PROC

; Code

ret

Sub3 ENDP

* The stack helps keep track of return addresses.
* Each **RET** pops the last return address from the stack, resuming the correct place.

**5.2.4 Passing Arguments in Registers**

* Instead of using specific variable names, **pass values in registers**.
* Example: Passing three numbers before calling a sum procedure.

.data

sumResult DWORD ?

.code

main PROC

mov eax, 10000h ; First number

mov ebx, 20000h ; Second number

mov ecx, 30000h ; Third number

call SumOf ; Call procedure

mov sumResult, eax ; Store the result

main ENDP

* After calling **SumOf**, the sum in **EAX** is stored in **sumResult**.

**5.2.5 Summing an Integer Array**

* A common loop calculates the sum of an array.
* Implemented easily in assembly language.

**Summary**

* **Procedures** organize assembly code like functions in high-level languages.
* Use **PROC** and **ENDP** to define them.
* **CALL** jumps to a procedure, and **RET** brings execution back.
* Use **registers** to pass values instead of fixed variable names.
* The **stack** manages return addresses and nested calls efficiently.
* Always **document procedures** for clarity and debugging.

This makes assembly procedures easy to understand and use efficiently!

**Assembly Language Procedures**

**Saving and Restoring Registers**

* When a procedure modifies registers, save them at the beginning and restore them at the end.
* This ensures that the calling program’s register values remain unchanged.
* Exception: The return register (usually **EAX**) should not be saved/restored.

**USES Operator**

* The **USES** operator simplifies saving/restoring registers in procedures.
* It automatically generates **PUSH** (save) and **POP** (restore) instructions.
* Example:
* ArraySum PROC USES esi ecx
* mov eax, 0
* L1:
* add eax, [esi]
* add esi, TYPE DWORD
* loop L1
* ret
* ArraySum ENDP
* This is equivalent to manually pushing and popping registers.

**Debugging Tip**

* In **Microsoft Visual Studio**, view hidden machine instructions:
  + Right-click in **Debugging window** → Select **Go to Disassembly**.

**Exception for Return Registers**

* If a procedure returns a value in **EAX**, do not **PUSH/POP** it.
* Example of incorrect usage:
* SumOf PROC
* push eax
* add eax, ebx
* add eax, ecx
* pop eax ; This removes the sum from EAX!
* ret
* SumOf ENDP

**5.2.7 Section Review – Easy Answers**

**1. Does the PROC directive start a procedure, and the ENDP directive end it?**

Answer: ✅ True – The PROC directive marks the start of a procedure, and ENDP marks the end of it.

**2. Can a procedure be defined inside another procedure?**

Answer: ❌ False – You cannot define one procedure inside another; each must be separate.

**3. What happens if the RET instruction is missing from a procedure?**

Answer: If RET is missing, the program won’t know where to return, which can cause errors or unexpected behavior.

**4. How are "Receives" and "Returns" used in procedure documentation**?

Answer:

* Receives – Describes the input values or parameters a procedure gets.
* Returns – Describes the output or result the procedure gives back.

**5. Does the CALL instruction push the offset of the CALL instruction onto the stack?**

Answer: ❌ False – The CALL instruction does not push its own offset; it pushes the next instruction’s offset so the program knows where to continue after returning.

**6. Does the CALL instruction push the offset of the instruction after it onto the stack?**

Answer: ✅ True – When CALL is used, the next instruction’s offset is saved so the program knows where to resume execution after the procedure ends.

**Linking to an External Library**

* Writing input-output (I/O) code in assembly is complex and time-consuming.
* Instead, use **link libraries** to simplify tasks.
* The book provides **Irvine32.lib** (for 32-bit programs) and **Irvine64.obj** (for 64-bit programs).

**What is a Link Library?**

* A **file** that contains pre-assembled procedures.
* When you **CALL** a procedure from a library, the assembler doesn’t know its address.
* The **linker** finds the procedure in the library and adds it to the final program.

**Using an External Procedure**

1. **Declare the procedure** using **PROTO**:
2. WriteString proto
3. **Call the procedure**:
4. call WriteString
5. **Link the library** using the linker command:
6. link hello.obj irvine32.lib kernel32.lib

**Linking 32-bit Programs**

* **kernel32.lib** helps link system functions from **kernel32.dll** (a core Windows file).
* The Irvine32 library links to **kernel32.lib**, which connects to **kernel32.dll**.
* Example: Your program → Irvine32.lib → kernel32.lib → kernel32.dll (Executes commands)

**Chapter 11**

* Later, we will directly link programs to **kernel32.lib** without Irvine32.lib.

These notes cover everything in **super simple wording**. Hope they help! 😊

**5.3.2 Section Review – Easy Answers**

**1. Does a link library contain assembly language source code?**

**Answer:** ❌ **False** – A link library contains **precompiled machine code**, not raw assembly language source code.

**2. How do you declare a procedure named MyProc in an external link library?**

**Answer:** Use the PROTO directive like this:

MyProc PROTO

This tells the assembler that MyProc exists in an external library.

**3. How do you call a procedure named MyProc in an external link library?**

**Answer:** Use the CALL instruction like this:

CALL MyProc

This tells the program to execute the MyProc procedure from the library.

**4. What is the name of the 32-bit link library provided with this book?**

**Answer:** The **Irvine32.lib** library is provided for 32-bit assembly programs.

**5. What type of file is kernel32.dll?**

**Answer:** kernel32.dll is a **Dynamic Link Library (DLL)** file. It contains important system functions for Windows programs.

**5.4 The Irvine32 Library**

**Why Was the Irvine32 Library Created?**

* There is **no official** standard library for assembly language (unlike in C or Python).
* In the early 1980s, **MS-DOS** was the main operating system.
* Programmers used **INT 21h services** to handle input and output.
* Even simple tasks, like printing a number on the screen, were **very complex** in assembly.

**How Was Printing a Number Done? (WriteInt Logic)**

To display a number in assembly, you had to:

1. Convert the binary number into ASCII characters.
2. Store the characters in a buffer **in reverse order**.
3. Print the buffer **in forward order**.

**WriteInt Algorithm (Simplified Explanation)**

* **Step 1**: Take the number and divide it by 10.
* **Step 2**: Store the remainder (this gives us the last digit).
* **Step 3**: Convert the remainder into an ASCII character.
* **Step 4**: Repeat until all digits are processed.
* **Step 5**: If the number is negative, add a "-" sign.
* **Step 6**: Print the digits in the correct order.

💡 **Why is this hard?**

* In C/C++, this is **easy** using printf().
* In assembly, you need to **manually** do these steps.

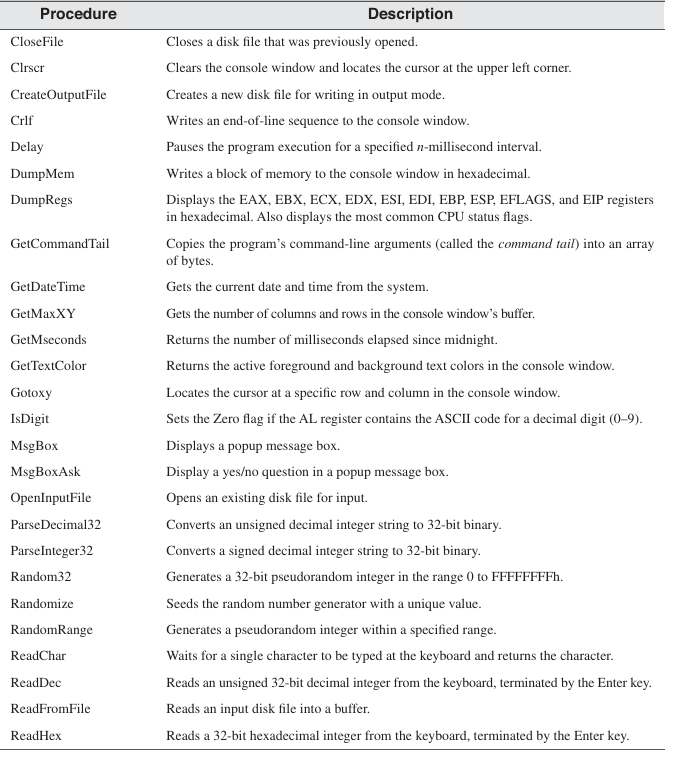
**Why Do Programmers Create Their Own Libraries?**

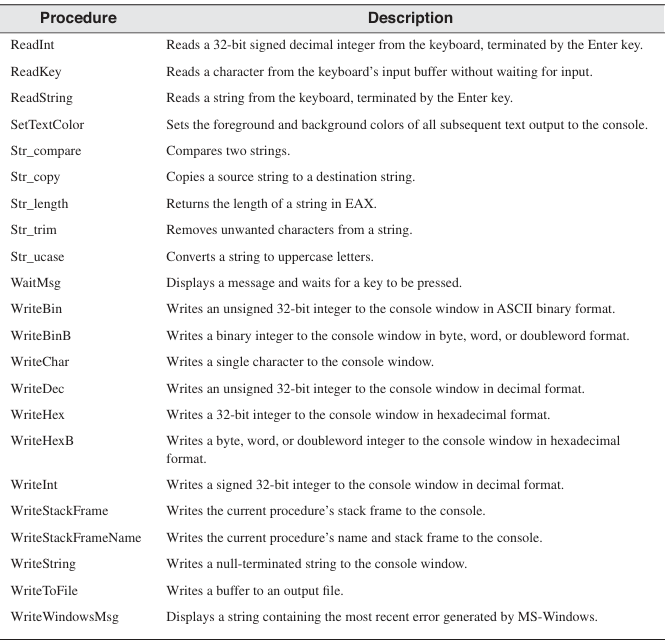
* Helps **reuse** code and avoid writing the same functions again.
* Great for **learning** how assembly works.
* Makes programming **easier** by providing ready-made functions.

**Why Use the Irvine32 Library?**

* Provides **easy functions** for beginners to do input/output.
* Works in **32-bit mode** on Windows.
* Saves time because you **don’t have to write everything from scratch**.

📌 **Fun Fact**: You can modify the Irvine32 library, but you must give credit to its creator.





**Section 5.4.5 Review – Super Easy Answers**

**1. Which procedure generates a random integer within a selected range?**

👉 RandomRange generates a random number within a given range.

**2. Which procedure displays “Press [Enter] to continue...” and waits for the user?**

👉 WaitMsg displays the message and waits for the user to press **Enter**.

**3. How to pause a program for 700 milliseconds?**

👉 Use the Delay procedure like this:

mov eax, 700

call Delay

**4. Which procedure writes an unsigned integer in decimal format?**

👉 WriteDec displays an unsigned integer in **decimal** form.

**5. Which procedure places the cursor at a specific location in the console window?**

👉 Gotoxy moves the cursor to a **specific row and column**.

**6. What INCLUDE directive is needed for Irvine32 library?**

👉 Add this line at the beginning of your program:

INCLUDE Irvine32.inc

**7. What types of statements are inside Irvine32.inc?**

👉 It contains:

* **Procedure declarations** (for input, output, math, etc.).
* **Constants** and **macro definitions**.

**8. What are the required input parameters for DumpMem?**

👉 DumpMem needs:

* **Memory address** to display.
* **Number of bytes to dump**.
* **Bytes per line** for formatting.

**9. What are the required input parameters for ReadString?**

👉 ReadString needs:

* **Memory address** (where to store the input).
* **Maximum number of characters** allowed.

**10. Which processor status flags are displayed by DumpRegs?**

👉 DumpRegs shows:

* **Zero flag (ZF)**
* **Carry flag (CF)**
* **Sign flag (SF)**
* **Overflow flag (OF)**

**11. Challenge: Prompt the user for an ID number and store it in an array of bytes**

👉 Sample code:

.data

idArray BYTE 20 DUP(0) ; Reserve space for 20 characters

maxLen DWORD 20 ; Max length of input

.code

mov edx, OFFSET idArray

mov ecx, maxLen

call ReadString ; Read user input and store in idArray

**Super Easy Answers for 5.8.1 Short Answer Questions**

**1. Which instruction pushes all 32-bit registers on the stack?**

👉 PUSHAD

**2. Which instruction pushes the EFLAGS register on the stack?**

👉 PUSHFD

**3. Which instruction pops the stack into the EFLAGS register?**

👉 POPFD

**4. Why is PUSH EAX EBX ECX better than PUSHAD in NASM?**

👉 It **only pushes selected registers**, saving stack space and avoiding unnecessary saves.

**5. How to replace PUSH EAX without using PUSH?**

👉 Use:

sub esp, 4 ; Make space on stack

mov [esp], eax ; Store EAX at the top of the stack

**6. (True/False) RET pops the top of the stack into the instruction pointer?**

✅ **True**

**7. (True/False) Nested procedure calls need the NESTED operator in MASM?**

❌ **False** – MASM allows nested calls without it.

**8. (True/False) In protected mode, procedure calls use at least 4 bytes of stack space?**

✅ **True**

**9. (True/False) ESI and EDI can’t be used for passing 32-bit parameters?**

❌ **False** – They **can** be used.

**10. (True/False) ArraySum gets a pointer to an array of doublewords?**

✅ **True**

**11. (True/False) USES lists all registers modified in a procedure?**

✅ **True**

**12. (True/False) USES only generates PUSH, so you must write POP manually?**

❌ **False** – It generates both PUSH and POP.

**13. (True/False) The register list in USES needs commas?**

❌ **False** – MASM doesn’t require commas.

**14. Modify ArraySum for 16-bit words**

👉 Change **DWORD** to **WORD** and update summation logic.

**15. What is the final value in EAX after this?**

push 5

push 6

pop eax ; EAX = 6

pop eax ; EAX = 5

✅ **EAX = 5**

**16. What happens when this code runs?**

push 10

push 20

call Ex2Sub

pop eax

👉 Ex2Sub does POP EAX, removing **20**.  
👉 Then POP EAX again removes **10**.  
✅ **EAX = 10 on line 6** (Answer: **a**)

**17. What happens when this code runs?**

mov eax,30

push eax

push 40

call Ex3Sub

👉 PUSHA saves **all registers**.  
👉 POPA restores them, **resetting EAX to 30**.  
✅ **EAX = 30 on line 6** (Answer: **c**)

**18. What happens when this code runs?**

push offset Here

jmp Ex4Sub

👉 JMP skips Here, so RET jumps to a bad address.  
✅ **Runtime error on Line 4** (Answer: **b**)

**19. What happens when this code runs?**

assembly

CopyEdit

mov eax,40

push eax

call Ex5Sub

👉 POP EAX removes **40**.  
👉 POP EDX tries to remove **nothing**, causing an error.  
✅ **Runtime error on Line 11** (Answer: **d**)

**20. What values will be stored in the array?**

👉 EAX starts at **10**, and every function adds **10** before storing it.

assembly

CopyEdit

array[0] = 10

array[4] = 20

array[8] = 30

array[12] = 40

✅ **Final array = {10, 20, 30, 40}**