

Contents

PROCEDURES IN ASSEMBLY LANGUAGE.....	2
QUICK REVIEW: STACK QUESTIONS MADE SIMPLE	12
NESTED PROCEDURE CALLS.....	18
PROGRAM WALKTHROUGH: TESTING THE ARRAYSUM PROCEDURE	27
PROCEDURES, RET, AND EXTERNAL LIBRARIES	31
INDIVIDUAL PROCEDURE DESCRIPTIONS.....	33
IMPLEMENTING THE PROCEDURES: USING IRVINE32	47
IRVINE 64.....	51
QUESTIONS.....	58
GOD LEVEL QUESTIONS.....	63

PROCEDURES IN ASSEMBLY LANGUAGE

Linking External Libraries in Assembly: A Friendly Guide

Want to use an external library in your assembly code? The linker is your best friend! Just use the `-l` and `-L` options on the command line.

- `-l` tells the linker *which* library to use.
- `-L` tells it *where* to find the library file.

Example: Let's say you want to link your `hello.obj` file to the Irvine32 library. Here's how you'd do it:

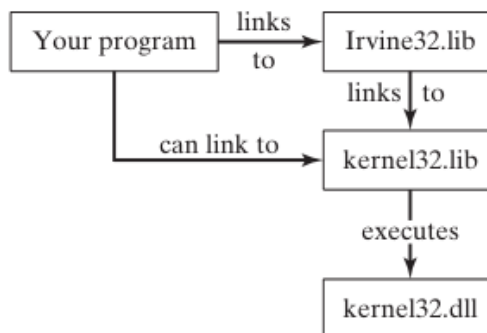
```
link hello.obj -l Irvine32.lib kernel32.lib
```

This command links your object file to both the Irvine32 library and the kernel32 library.

Why kernel32? It's a system library packed with essential functions, like reading and writing to the console—super handy!

Putting It All Together Here's a simple assembly program that calls a procedure from the Irvine32 library:

FIGURE 5-9 Linking 32-bit programs.



```

; This program displays the string "Hello, world!" on the console.

include Irvine32.inc

.data
msg db "Hello, world!", 0

.code
start:
    mov eax, OFFSET msg
    call WriteString
    mov eax, 1 ; exit program
    int 0x80

```

This little program prints “Hello, world!” to the console. Here’s how it works:

How Linking Works: Your Flowchart Explained

Your flowchart does a great job showing how the linker turns your code into a working program! Here’s a quick breakdown:

1. **Reads Your Code:** The linker starts by reading your program’s object file and building a *symbol table*—a list of all the names (like functions or variables) and their addresses.
2. **Searches Libraries:** It then looks through Irvine32.lib and kernel32.lib for any functions or data your program needs. If it finds a match, it copies the code into your final program.
3. **Updates Addresses:** The linker makes sure everything points to the right place, updating the symbol table as it goes.
4. **Repeats Until Done:** It keeps going until every reference is resolved.
5. **Creates the Executable:** Finally, it writes your finished program to disk—ready to run!

Bonus: The flowchart skips some details (like fixing up addresses for different memory locations), but it’s a perfect big-picture view!

The Runtime Stack: Your Program's Scratch Pad

The stack is like a stack of plates—you can only add or remove from the top. It's where your program stores function arguments, return values, and local variables.

- **Push:** Decrements the stack pointer (ESP) by 4 bytes and stores a 32-bit value at the new top.
- **Pop:** Grabs the top 32-bit value and increments ESP by 4 bytes.

Example: Here's how you'd push and pop a value in assembly:

```
; Push the value 10 onto the stack.  
push 10  
  
; Pop the value off of the stack and store it in the register EAX.  
pop eax
```

How the Stack Grows in Memory

Yes, you've got it! The stack grows *downward*—meaning it starts at a high memory address and moves toward lower ones as you push more data.

Your Example:

- Suppose your stack starts at address 4096 (the top).
- You push 10: the stack pointer moves down to 4092.
- You push 11: the stack pointer moves down to 4088.
- You pop: the stack pointer moves back up to 4092.

In short:

- **Push:** Stack pointer moves to a lower address.
- **Pop:** Stack pointer moves back up.

This “last in, first out” behavior is what makes the stack so useful for managing function calls and local variables.

FIGURE 5-3 Pushing integers on the stack.

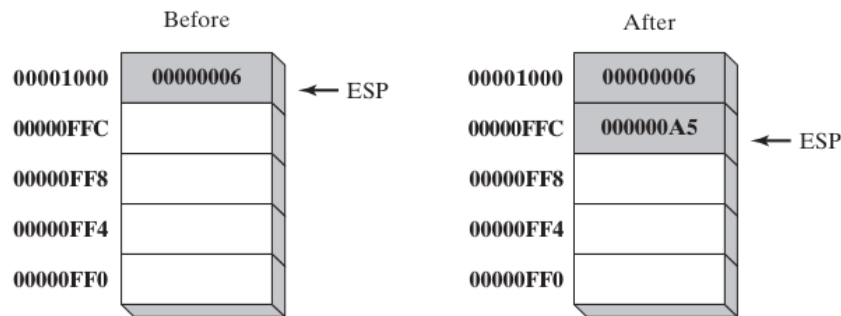
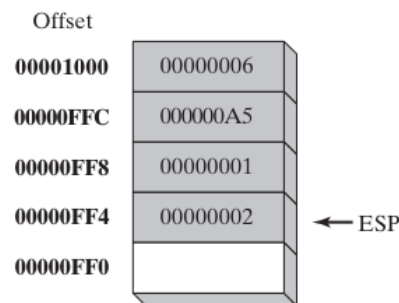


FIGURE 5-4 Stack, after pushing 00000001 and 00000002.



Push Operation: Adding to the Stack

In a 32-bit system, pushing a value onto the stack is a two-step process:

1. **Move the Stack Pointer:** The stack pointer (ESP) is decremented by 4 bytes, making room for the new value.
2. **Store the Value:** The value is copied into the memory location now pointed to by ESP.

Key Point: The stack pointer always points to the last item pushed, and the stack grows *downward*—from higher memory addresses to lower ones.

Pop Operation: Removing from the Stack

Popping a value off the stack is just as simple:

1. **Remove the Value:** The value at the memory location pointed to by ESP is “removed” (i.e., it’s read and no longer considered part of the stack).
2. **Move the Stack Pointer:** ESP is incremented by 4 bytes, pointing to the next item on the stack.

Example:

- Before pop: ESP points to 00000002.
- After pop: 00000002 is removed, and ESP moves up by 4 bytes.

Saving and Restoring Flags

In assembly, you can save and restore the processor flags using PUSHFD and POPFD. This is handy for preserving the state of your program, especially before and after interrupts or critical sections.

Tip: If you want to be extra safe, you can push the flags onto the stack and immediately pop them into a variable. This makes your code clearer and less prone to errors—you can always see where the flags are stored and when they're restored.

Just remember: Always make sure your POPFD (or flag restore) happens at the right time, or your program might not behave as expected!

PUSHFD and POPFD Operation

```
.data
savedFlags DWORD ?

.code
main PROC
    ; Save the flags using PUSHFD
    pushfd

    ; Store the saved flags into the 'savedFlags' variable
    pop savedFlags

    ; Modify some flags (for demonstration purposes)
    cli ; Clear Interrupt Flag
    sti ; Set Interrupt Flag

    ; Restore the original flags using 'savedFlags'
    push savedFlags
    popfd

    ; Your code continues here

    ; Exit the program
    mov eax, 0x4C00 ; Exit code 0
    int 0x21        ; Call DOS function to exit
main ENDP

END main
```

PUSHFD in x86 Assembly: Your Friendly Flags Saver

The PUSHFD instruction is like a helpful assistant for your flags register: it knows exactly where to find all your flags and automatically pushes the entire 32-bit EFLAGS register onto the stack. No need to tell it which flags to grab—it just does it all for you!

This means it saves everything: status flags like the Zero Flag (ZF), Carry Flag (CF), and Overflow Flag (OF), plus control flags such as the Direction Flag (DF) and Interrupt Flag (IF). When you're ready to restore things, just use POPFD to pull those flags right back where they belong.

How It Works in Practice

Let's say you want to temporarily tweak some flags, but you don't want to lose the original settings. Here's how you can do it safely:

1. **Save the Flags:** Use PUSHFD to push the current flags onto the stack.
2. **Store Them:** Pop the flags off the stack and tuck them away in a variable (like savedFlags).
3. **Play Around:** Clear or set flags as needed—maybe disable interrupts with CLI or re-enable them with STI.
4. **Restore Order:** When you're done, push your saved flags back onto the stack and use POPFD to restore the original state.

Example Walkthrough

```
.data
    savedFlags DWORD 0 ; A cozy spot to store your flags

.code
main PROC
    PUSHFD             ; Save all flags to the stack
    POP savedFlags     ; Store them in savedFlags

    ; --- Do your thing ---
    CLI                ; Disable interrupts (just for fun)
    STI                ; Re-enable interrupts

    ; --- Restore the original flags ---
    PUSH savedFlags    ; Put your saved flags back on the stack
    POPFD              ; Restore the flags register

    ; Exit gracefully
    MOV eax, 0x4C00
    INT 0x21
main ENDP
```

or

```

; Reversing a String (RevStr.asm)
.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO,dwExitCode:DWORD

.data
aName BYTE "Abraham Lincoln",0
nameSize = ($ - aName) - 1

.code
main PROC
    ; Initialize registers and counters
    mov ecx,nameSize
    mov esi,0

L1:
    ; Get the next character from the 'aName' string
    movzx eax,aName[esi]

    ; Push the character onto the stack
    push eax

    ; Move to the next character
    inc esi

    ; Continue the loop until all characters are processed
    loop L1

    ; Pop the characters from the stack (in reverse order)
    ; and store them back into the 'aName' array

    ; Reset counters
    mov ecx,nameSize
    mov esi,0

L2:
    ; Get the character from the stack
    pop eax

    ; Store the character in the 'aName' string
    mov aName[esi],al

    ; Move to the next character
    inc esi

    ; Continue the loop until all characters are restored
    loop L2

    ; Exit the program
    INVOKE ExitProcess,0

main ENDP
END main

```

A Word of Caution While PUSHFD and POPFD make flag management a breeze, always double-check that you're saving and restoring them correctly. Messing up here can lead to unexpected program behavior—so handle with care!

How This Code Reverses a String Using the Stack

This assembly program reverses the string "Abraham Lincoln" using the stack's natural Last-In-First-Out (LIFO) behavior. Here's the breakdown:

1. Setup

- **.data Section:**
 - aName holds the string "Abraham Lincoln" (plus a null terminator).
 - nameSize calculates the string's length (excluding the null terminator).
- **.code Section:**
 - ecx is loaded with the string length (loop counter).
 - esi starts at 0 (string index).

2. Pushing Characters onto the Stack (Loop L1)

- **What happens:**
 - Load each character from aName[esi] into eax.
 - Push eax onto the stack.
 - Move to the next character (inc esi).
 - Repeat until all characters are pushed.
- **Why it works:** The stack's LIFO order means the first character pushed will be the last one popped—perfect for reversing!

3. Popping Characters Back (Loop L2)

- **What happens:**
 - Pop each character from the stack into eax.
 - Store it back into aName[esi].
 - Move to the next position (inc esi).
 - Repeat until all characters are restored.
- **Result:** The string is now reversed in place!

4. Wrap-Up

- The program exits cleanly with INVOKE ExitProcess, 0.

In short: The code uses the stack to flip the string—simple, clever, and efficient!

The Tricky Bits: What Makes This Assembly Code Challenging

```
; Example: Reverse "Abraham Lincoln" using stack operations - MASM syntax, Windows environment
.386
.model flat, stdcall
option casemap:none

include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib

.data
    aName db "Abraham Lincoln",0        ; original string with null terminator
    nameSize equ ($ - aName) - 1        ; length of string excluding null terminator

.code
main PROC
    ; Initialize registers
    mov esi, 0                          ; index into string
    mov ecx, nameSize                    ; loop counter = string length

L1: ; Push each character onto stack
    movzx eax, byte ptr aName[esi]      ; load character into eax (zero-extended)
    push eax                             ; push onto stack
    inc esi                              ; move to next character
    loop L1                             ; repeat until ecx = 0

    ; Reset index and counter for popping
    mov esi, 0
    mov ecx, nameSize

L2: ; Pop characters back into string (reversing order)
    pop eax                             ; get character from stack
    mov al, byte ptr eax                 ; use only low 8 bits
    mov aName[esi], al                  ; store back into string
    inc esi
    loop L2

    ; Exit program gracefully
    INVOKE ExitProcess, 0
main ENDP
END main
```

1. Playing with Strings

We're working with the string "Abraham Lincoln" as a bunch of bytes in memory. By using `aName[esi]`, the code steps through each character one at a time. The star of the show here is `movzx`: it grabs a single byte (your character) and safely drops it into a 32-bit register, padding the rest with zeros so you don't end up with random junk data.

2. Loops Made Simple

`ecx` is our countdown timer. It starts at the length of the string and ticks down to zero. The loop instruction is like a built-in shortcut—it automatically subtracts one from `ecx` and jumps back to the loop until we're done. If you're new to assembly, this feels a little magical at first, but once you see it, it clicks.

3. Stack Magic

Think of the stack as a "last in, first out" box. Push characters in order, then pop them back out, and voilà—you've reversed the string! It's clever, but it can be tricky to picture until you imagine the stack as a spring-loaded box that spits out the last thing you put in.

4. Register Juggling

Each register has its own job:

- **eax**: Holds the current character.
- **esi**: Points to where we are in the string.
- **ecx**: Counts how many times we loop.
- **al**: The tiny lower 8 bits of `eax`, perfect for writing characters back.

It's like juggling—you've got to keep track of which ball (register) is doing what.

5. Ending Gracefully

INVOKE `ExitProcess`, 0 is our polite way of saying "all done!" The 0 means everything went fine. No drama, just a clean exit.

6. Measuring the String

`nameSize` is calculated as $(\$ - \text{aName}) - 1$. Here, `$` is the current address, and `aName` is the start of the string. Subtracting gives us the length, and we shave off one to ignore the null terminator. If pointer math feels weird, don't worry—it's just subtraction with addresses.

7. Control Flow Adventures

The code hops between loops (L1 and L2), pushing and popping characters. Understanding how the program jumps around is key—it's like following a treasure map. Once you see the flow, debugging and writing your own assembly gets way easier.

🌟 Why It's Worth the Effort

Assembly is all about precision and control. At first, these tricks feel like riddles, but once you get the hang of them, you'll see how powerful (and even fun!) low-level programming can be. Keep practicing, and soon these “tricky bits” will feel like second nature.

QUICK REVIEW: STACK QUESTIONS MADE SIMPLE

Which register manages the stack in 32-bit mode?

That's **ESP (Extended Stack Pointer)**. It always points to the top of the stack, and every time you push or pop something, ESP moves to keep track.

📖 Runtime Stack vs. Stack Data Type

- **Runtime stack:** A real chunk of memory used by programs to handle function calls, local variables, and flow control. It grows when you call a function and shrinks when you return.
- **Stack data type:** A higher-level idea in computer science—just a structure that supports push and pop operations, following the **Last-In-First-Out (LIFO)** rule. Think of it this way: the runtime stack is the *real-world implementation*, while the stack data type is the *concept*.

🔄 Why is the stack called LIFO?

Because the **last thing you put in is the first thing that comes out**. Just like stacking plates—you grab the top one first.

What happens to ESP when you push a 32-bit value?

ESP moves down by **4 bytes** (since 32 bits = 4 bytes). That way, ESP always points to the new top of the stack.

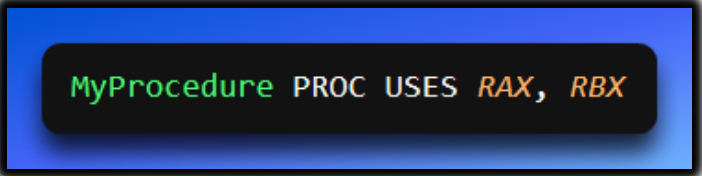
✓ True/False Quickies

- **Local variables live on the stack?** → **True**. That's how functions keep their own private workspace.
- **PUSH can't take an immediate operand?** → **False**. You *can* push a value directly, like `PUSH 42`.

⚙️ Assembly Instructions & Directives

PROC / ENDP → Mark the start and end of a procedure. Think of them as bookends for your function.

- **PUSH** → Put a value on the stack.
- **POP** → Take the top value off the stack and store it somewhere.
- **PUSHA / PUSHAD** → Push *all* general-purpose registers at once (16-bit vs. 32-bit versions).
- **POPA / POPAD** → Pop them all back in the same order.
- **PUSHFD / POPFD** → Save and restore the processor's flags (EFLAGS). Handy for preserving state.
- **RET** → Return from a procedure. It pops the return address off the stack and jumps back.
- **USES** → A directive that documents which registers a procedure touches. Example:



```
MyProcedure PROC USES RAX, RBX
```

- This makes it clear to other programmers which registers are affected.

✨ Why This Matters

These instructions are the building blocks of how assembly manages memory, flow, and state. Once you see them as simple “stack tricks” and “procedure bookends,” they stop feeling intimidating and start feeling like neat little tools.

```
477 .model flat, stdcall
478 .stack 4096
479
480 .data
481     message db "Hello, World!", 0
482
483 .code
484     MyProcedure PROC USES EAX, EBX
485         ; Your code here that uses EAX and EBX registers
486         mov eax, 42
487         mov ebx, 24
488         add eax, ebx
489         ; Rest of your code
490         ret
491     MyProcedure ENDP
492
493     main PROC
494         ; Push all general-purpose registers onto the stack
495         pushad
496         ; Call MyProcedure
497         call MyProcedure
498         ; Pop all general-purpose registers from the stack
499         popad
500         ; Push processor flags onto the stack
501         pushfd
502         ; Modify some flags (for demonstration purposes)
503         cld ; Clear direction flag
504         ; Pop processor flags from the stack
505         popfd
506         ; Display a message
507         mov eax, 4          ; syscall number for sys_write (Linux)
508         mov ebx, 1          ; file descriptor 1 (stdout)
509         mov ecx, offset message ; pointer to the message
510         mov edx, 13         ; message length
511         int 0x80            ; invoke syscall
512         ; Exit program
513         mov eax, 1          ; syscall number for sys_exit (Linux)
514         int 0x80            ; invoke syscall
515     main ENDP
516 END main
```

✂ PROCEDURES IN ASSEMBLY LANGUAGE

Procedures are like mini-programs inside your program.

They break big tasks into smaller, manageable pieces, making your code easier to understand, test, and reuse.

In assembly, a **procedure** is simply a named block of code that you can call from anywhere.

Unlike higher-level languages, assembly doesn't enforce structure, so programmers use procedures to bring order and modularity to their work.

✨ The PROC Directive

The PROC directive defines a procedure. You give it a name, write the code inside, and close it with ENDP. Think of PROC and ENDP as the "start" and "end" markers for your subroutine.

Example: A Simple Sum Procedure

```
SumOf PROC
add
    eax,ebx
add
    eax,ecx
ret
SumOf ENDP
```

This procedure adds three 32-bit integers together and leaves the result in the EAX register.

☎ Calling a Procedure

To use a procedure, you just call it by name. For example:

```
mov    eax, 1
mov    ebx, 2
mov    ecx, 3

call   SumOf

; eax now contains the sum of 1, 2, and 3, which is 6
```

Here, the program sets up values in registers, calls SumOf, and gets the result back in EAX.

Returning from a Procedure

Procedures return control with the RET instruction. RET pops the return address off the stack and jumps back to wherever the procedure was called. Simple, clean, and efficient.

```
.code
main PROC
    ; Call the MyProcedure
    call MyProcedure

    ; Continue executing after the call

    ; Exit the program
    exit

MyProcedure PROC
    ; Your procedure code here

    ; Return to the caller
    ret

MyProcedure ENDP

main ENDP
END main
```

✴ Why Procedures Matter

- **Clarity:** Breaks complex programs into smaller tasks.
- **Reusability:** Write once, call anywhere.
- **Structure:** Helps assembly programmers organize code in a language that doesn't enforce it.

Understanding the Example Step by Step

.code Section

This is where the actual program lives. It contains both the **main program** and any procedures you define, like MyProcedure.

main PROC

This marks the start of the main program. Think of it as the entry point where execution begins.

call MyProcedure

The CALL instruction is how you jump into a procedure. When the CPU executes CALL, two things happen:

1. **Return address pushed** → The address of the next instruction (right after CALL) is saved on the stack.
2. **Instruction pointer updated** → The CPU loads the address of MyProcedure into EIP (the instruction pointer), so execution continues inside the procedure.

After the procedure finishes, the program will pick up right where it left off—thanks to that saved return address.

MyProcedure PROC

This is the start of the procedure itself. Inside here, you write the code that defines what the procedure does.

ret

The RET instruction is how you exit a procedure. It pops the saved return address off the stack and loads it back into EIP. That means the CPU “jumps back” to the instruction immediately after the CALL in the main program.

MyProcedure ENDP

Marks the end of the procedure. It's the closing bracket for your subroutine.

main ENDP

Marks the end of the main program.

END main

Tells the assembler: “This is the end of the program, and execution starts at main.”

Big Picture Summary

- **CALL** → jumps into a procedure and saves the return address on the stack.
- **RET** → pops that return address off the stack and jumps back to the caller. Together, they make structured program flow possible in assembly—like function calls in higher-level languages.

NESTED PROCEDURE CALLS

A **nested procedure call** happens when one procedure calls another before finishing. In other words, Procedure A calls Procedure B, and only after B returns does A continue.

```
.code
main PROC
    ; Call the OuterProcedure
    call OuterProcedure
    ; Continue executing after the call to OuterProcedure
    ; Exit the program
    exit

OuterProcedure PROC
    ; Some code in the OuterProcedure
    ; Call the InnerProcedure
    call InnerProcedure
    ; Continue executing in OuterProcedure after the call to InnerProcedure
    ; Return from OuterProcedure
    ret

OuterProcedure ENDP
InnerProcedure PROC
    ; Some code in the InnerProcedure
    ; Return from InnerProcedure
    ret

InnerProcedure ENDP
main ENDP
END main
```

.code Section

This is where the program lives. It contains the **main program** and the two procedures: OuterProcedure and InnerProcedure.

main PROC

Marks the start of the main program. This is the entry point where execution begins.

call OuterProcedure

The main program calls OuterProcedure. When the CPU executes CALL, it:

1. Pushes the return address (the spot right after the CALL) onto the stack.
2. Jumps into OuterProcedure by loading its address into the instruction pointer (EIP).

So, control moves from main into OuterProcedure.

OuterProcedure PROC

This is the start of OuterProcedure. It runs its own code, then calls another procedure—InnerProcedure.

call InnerProcedure

Inside OuterProcedure, the CALL instruction jumps into InnerProcedure. Again, the return address is saved on the stack, and execution continues inside InnerProcedure.

InnerProcedure PROC

This is the start of InnerProcedure. It runs its code, then finishes with RET.

ret in InnerProcedure

The RET pops the saved return address off the stack and jumps back to the instruction immediately after the CALL InnerProcedure in OuterProcedure. Execution resumes in OuterProcedure.

ret in OuterProcedure

Once OuterProcedure finishes, its RET pops the return address saved earlier and jumps back to the instruction immediately after CALL OuterProcedure in main. Execution resumes in the main program.

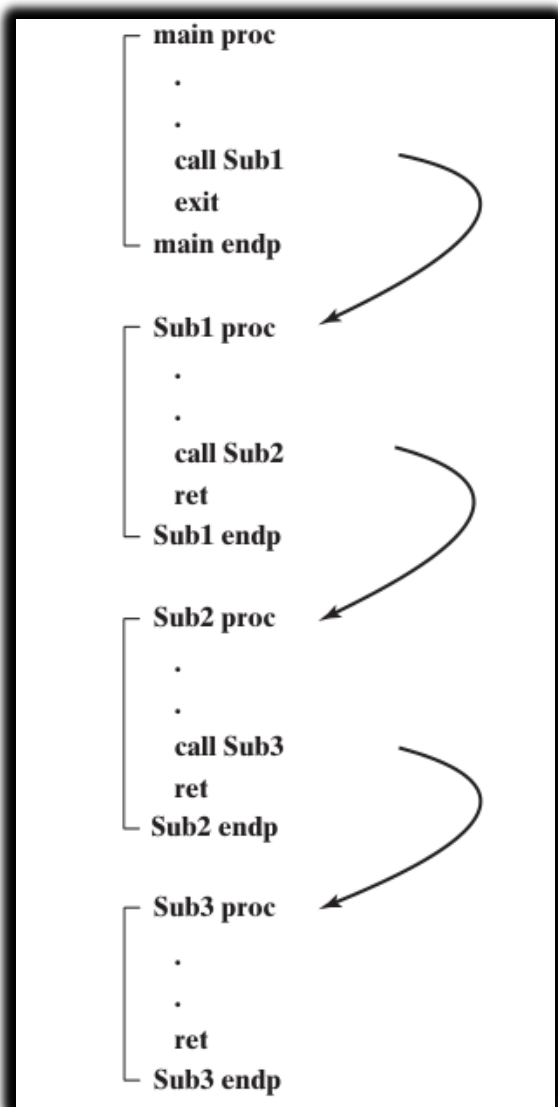
Big Picture

- The **main program** calls OuterProcedure.
- **OuterProcedure** does its work, then calls InnerProcedure.
- **InnerProcedure** runs, returns to OuterProcedure.
- **OuterProcedure** finishes, returns to main.

The stack keeps track of all these return addresses, so the CPU always knows exactly where to go back.

Why It Matters

Nested procedure calls show how assembly supports structured program flow. With CALL and RET, you can build layered tasks—procedures that call other procedures—while the stack quietly manages all the “return-to-here” addresses behind the scenes.



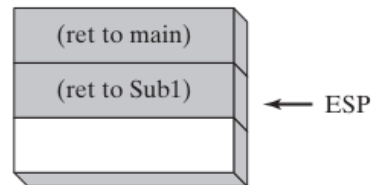
The main procedure calls the Sub1 procedure. Sub1 calls Sub2, and Sub2 calls Sub3. Sub3 then returns to Sub2, Sub2 returns to Sub1, and Sub1 returns to main.

Here is a more detailed explanation of the flow:

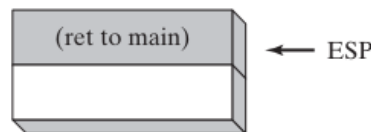
- The main procedure calls Sub1 by executing the **CALL Sub1** instruction.
- The **CALL** instruction pushes the address of the instruction following the **CALL** instruction onto the stack.
- This is the return address for Sub1.
- The **CALL** instruction then loads the address of Sub1 into the instruction pointer.
- The Sub1 procedure begins executing. Sub1 calls Sub2 by executing the **CALL Sub2** instruction.

- The CALL instruction pushes the address of the instruction following the CALL instruction onto the stack.
- This is the return address for Sub2.
- The CALL instruction then loads the address of Sub2 into the instruction pointer.
- The Sub2 procedure begins executing.
- Sub2 calls Sub3 by executing the CALL Sub3 instruction.
- The CALL instruction pushes the address of the instruction following the CALL instruction onto the stack.
- This is the return address for Sub3.
- The CALL instruction then loads the address of Sub3 into the instruction pointer.
- The Sub3 procedure begins executing. Sub3 returns by executing the RET instruction.
- The RET instruction pops the return address for Sub2 from the stack into the instruction pointer.
- Sub2 begins executing again.
- Sub2 returns by executing the RET instruction.
- The RET instruction pops the return address for Sub1 from the stack into the instruction pointer.
- Sub1 begins executing again.
- Sub1 returns by executing the RET instruction.
- The RET instruction pops the return address for main from the stack into the instruction pointer.
- The main procedure begins executing again.
- The stack is used to keep track of the return addresses for the nested procedure calls.
- When a procedure calls another procedure, it pushes its return address onto the stack.
- When a procedure returns, it pops its return address from the stack.
- This ensures that the procedures return to the correct location in the program.

After the return, ESP points to the next-highest stack entry. When the RET instruction at the end of **Sub2** is about to execute, the stack appears as follows:



Finally, when **Sub1** returns, stack[ESP] is popped into the instruction pointer, and execution resumes in **main**:



Clearly, the stack proves itself a useful device for remembering information, including nested procedure calls. Stack structures, in general, are used in situations where programs must retrace their steps in a specific order.

Here is a more detailed explanation of the image:

- The ret to main entry contains the address of the instruction following the CALL Sub1 instruction in the main procedure.
- This is the return address for the main procedure.
- The ret to Sub1 entry contains the address of the instruction following the CALL Sub2 instruction in the Sub1 procedure.
- This is the return address for the Sub1 procedure.
- The ret to Sub2 entry contains the address of the instruction following the CALL Sub3 instruction in the Sub2 procedure.
- This is the return address for the Sub2 procedure.
- The stack is used to keep track of the return addresses for the nested procedure calls.
- When a procedure calls another procedure, it pushes its return address onto the stack.
- When a procedure returns, it pops its return address from the stack.
- This ensures that the procedures return to the correct location in the program.
- In the image, Sub3 has just returned.

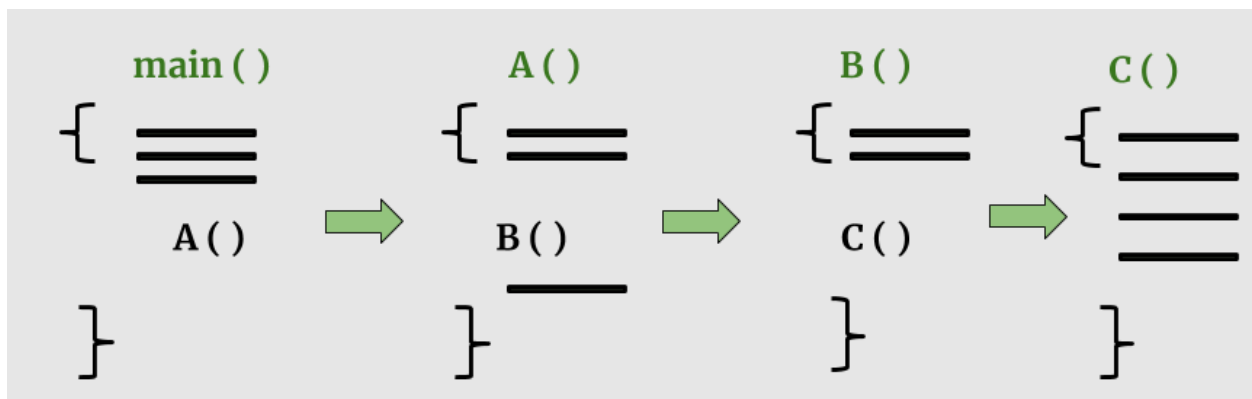
- The RET instruction in Sub3 popped the return address for Sub2 from the stack and loaded it into the instruction pointer.
- This caused Sub2 to begin executing again.
- The next instruction to be executed is the instruction after the CALL Sub3 instruction in the Sub2 procedure.
- After Sub2 finishes executing, it will return to Sub1 by executing the RET instruction.
- The RET instruction in Sub2 will pop the return address for Sub1 from the stack and load it into the instruction pointer.
- This will cause Sub1 to begin executing again.
- The same process will continue until all of the procedures have returned.
- The last procedure to return will be the main procedure.

So, this nesting is basically, putting the address of the procedure that called me onto the stack, executing my code, then returning causes a pop of the address of the procedure that called me? And I will only press return if I don't call another procedure within me?

Yes, your understanding of nested procedure calls is correct. When a procedure is called, the address of the instruction immediately following the CALL instruction is pushed onto the stack as the return address.

This allows the called procedure to execute its code and, when it's ready to return, it uses the RET instruction. The RET instruction pops the return address from the stack and transfers control back to the instruction with that address, effectively "returning" to the calling procedure.

Here's a step-by-step explanation of the process:



The calling procedure (let's call it Procedure A) encounters a CALL instruction to call another procedure (Procedure B).

Before transferring control to Procedure B, Procedure A pushes its own return address (the address of the instruction immediately following the CALL) onto the stack.

Procedure B is executed. When Procedure B is ready to return, it uses the RET instruction.

The RET instruction pops the return address (which is the address of the instruction following the CALL in Procedure A) from the stack.

Control is transferred back to Procedure A, continuing from where it left off.

Your statement, "**I will only press return if I don't call another procedure within me,**" is also accurate.

When a procedure calls another procedure within it (nested calls), it won't use RET until all nested calls are completed.

Each RET instruction corresponds to a single CALL instruction, and they must match.

If you call another procedure within the current one, you need to ensure that all inner procedure calls are resolved before the outer procedure returns.

This maintains the proper stack structure and allows for correct program execution.

```
ArraySum PROC
    ; Input:
    ;   ecx = array size
    ;   esi = pointer to the start of the array
    ; Output:
    ;   eax = sum of array elements

    xor eax, eax          ; Initialize eax to 0 (sum)
L1:
    add eax, [esi]         ; Add the current element to the sum
    add esi, 4             ; Move to the next element (assuming 32-bit integers)
    loop L1               ; Repeat for the remaining elements
    ret

ArraySum ENDP
```

The ArraySum procedure is designed to calculate the sum of an array of 32-bit integers. It's not about nested calls—it's more about **showing how procedures** can make code modular and reusable.

The Loop (loop L1)

- The loop L1 instruction runs through the array.
- Each time, it decrements the ecx register (which holds the loop counter).
- If ecx isn't zero yet, execution jumps back to the L1 label, continuing the iteration.
- This way, the procedure processes every element in the array.

Restoring Registers

- pop ecx → After the loop finishes, the original value of ecx is restored from the stack. This ensures the caller still has access to the loop counter it passed in.
- pop esi → Similarly, the pointer/index stored in esi is restored. This keeps the caller's register values intact after the procedure runs.

Returning (ret)

- The ret instruction signals the end of the procedure.
- At this point, the sum of all array elements is stored in the eax register, ready for the caller to use.

How It Works in Practice

In this version of ArraySum:

Inputs:

- ecx → the size of the array (number of elements).
- esi → the pointer to the start of the array.

Process:

- eax is initialized to 0.
- The loop runs through the array, adding each element to eax.

Output:

- When the procedure returns, eax contains the total sum.

✴ Why This Is Useful

By passing in the array size (ecx) and the starting address (esi), this procedure becomes flexible—it can handle arrays of different lengths and starting points.

That makes it reusable across different parts of a program, instead of being hard-coded for one specific case.

PROGRAM WALKTHROUGH: TESTING THE ARRAYSUM PROCEDURE

```
; Testing the ArraySum procedure (TestArraySum.asm)
.386
.model flat, stdcall
.stack 4096
ExitProcess PROTO, dwExitCode:DWORD
.data
    array DWORD 10000h, 20000h, 30000h, 40000h, 50000h ; An array of 32-bit integers
    theSum DWORD ? ; Variable to store the sum
.code
main PROC
    mov esi, OFFSET array ; ESI points to the array
    mov ecx, LENGTHOF array ; ECX = number of elements in the array
    call ArraySum ; Call the ArraySum procedure to calculate the sum
    mov theSum, eax ; Store the returned sum in theSum variable
    INVOKE ExitProcess, 0 ; Exit the program

main ENDP

;-----
; ArraySum
; Calculates the sum of an array of 32-bit integers.
; Receives: ESI = the array offset
;           ECX = number of elements in the array
; Returns:  EAX = sum of the array elements
;-----
ArraySum PROC
    push esi ; Save ESI (pointer to the array)
    push ecx ; Save ECX (number of elements)
    mov eax, 0 ; Initialize the sum to zero

L1:
    add eax, [esi] ; Add each integer to the sum
    add esi, TYPE DWORD ; Move to the next integer (assuming 32-bit integers)
    loop L1 ; Repeat for the remaining elements

    pop ecx ; Restore ECX
    pop esi ; Restore ESI
    ret ; Return with the sum in EAX
ArraySum ENDP
```

This program demonstrates how to use the ArraySum procedure to calculate the sum of an array of 32-bit integers. Let's break it down step by step.

Setup Directives

- `.386 and .model flat` → Tell the assembler we're targeting the 80386 processor and using the flat memory model.
- `.stack 4096` → Reserves 4096 bytes of stack space for the program.
- `ExitProcess PROTO, dwExitCode:DWORD` → Declares the Windows API function `ExitProcess`, which we'll use to end the program cleanly.

Data Section

```
array    DWORD 10000h, 20000h, 30000h, 40000h, 50000h
theSum   DWORD ?
```

- `array` → A list of five 32-bit integers.
- `theSum` → A variable to store the result. The `?` means it's uninitialized.

Main Procedure

```
main PROC
    mov esi, OFFSET array    ; point to start of array
    mov ecx, LENGTHOF array  ; number of elements (5)
    call ArraySum             ; calculate sum
    mov theSum, eax           ; store result
    INVOKE ExitProcess, 0     ; exit program
main ENDP
```

- `mov esi, OFFSET array` → `esi` points to the first element of the array.
- `mov ecx, LENGTHOF array` → `ecx` holds the number of elements (5).
- `call ArraySum` → Calls the procedure to do the work.
- `mov theSum, eax` → After the procedure returns, the sum is in `eax`. We copy it into `theSum`.
- `ExitProcess, 0` → Ends the program with exit code 0 (success).

Inside the ArraySum Procedure

```

ArraySum PROC
    push esi
    push ecx
    mov eax, 0                ; initialize sum

L1:
    add eax, [esi]            ; add current element
    add esi, TYPE DWORD      ; move to next element
    loop L1                  ; repeat until ecx = 0

    pop ecx
    pop esi
    ret
ArraySum ENDP

```

Step-by-step:

- `push esi / push ecx` → Save the caller's registers so we don't overwrite them.
- `mov eax, 0` → Start the sum at zero.
- `L1:` → Loop label.
- `add eax, [esi]` → Add the current array element to the sum.
- `add esi, TYPE DWORD` → Move `esi` to the next element (advance by 4 bytes).
- `loop L1` → Decrement `ecx`. If not zero, jump back to `L1`.
- `pop ecx / pop esi` → Restore the original register values.
- `ret` → Return to the caller with the sum in `eax`.

🌟 Big Picture

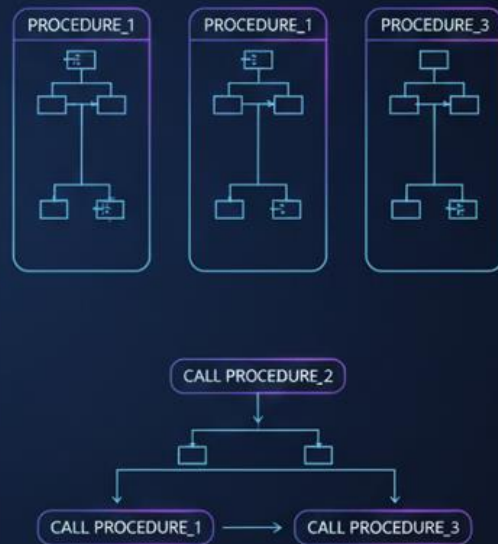
- The **main program** sets up the array pointer (esi) and size (ecx).
- **ArraySum** loops through the array, adding each element into eax.
- When finished, eax holds the total sum, which is passed back to the caller.
- The result is stored in theSum, and the program exits cleanly.

Assembly Procedures: Modularizing Code

Before Procedures: Monolithic Code
One Big Task (Hard to Manage)



With Procedures: Modular Code
Break into Smaller Tasks (Easy to Reuse)



PROCEDURES, RET, AND EXTERNAL LIBRARIES

✓ True/False Quick Review

- **PROC begins a procedure, ENDP ends it → True.** They're the bookends of a procedure.
- **You can define a procedure inside another → True.** These are called *nested procedures*. Just be careful with stack and register management.
- **What if RET is missing? → Without RET, control never returns to the caller.** The stack isn't fixed, the program counter isn't updated, and you'll likely crash or get unpredictable behavior.
- **CALL pushes the offset of the CALL instruction? → False.** It pushes the *return address*.
- **CALL pushes the offset of the instruction after CALL? → True.** That's how the CPU knows where to resume after the procedure finishes.

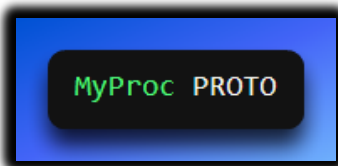
📖 Documentation Terms: Receives vs. Returns

- **Receives** → Inputs the procedure expects (like parameters or registers).
- **Returns** → The output the procedure provides (often in EAX).

This makes procedure documentation clear: what goes in, and what comes out.

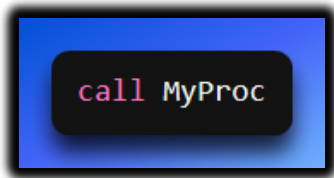
📦 Link Libraries

- **Do link libraries contain source code? → False.** They contain *precompiled machine code* you can reuse.
- **Declaring external procedures** → Use the `PROTO` directive:



This tells the assembler "MyProc exists," and the linker fills in the details later.

Calling external procedures → Just use:



- The linker ensures the right machine code is pulled in.
- **Library from the book** → Irvine32.lib, which connects to Windows API functions for input/output and more.
- **What is kernel32.dll??** → A **dynamic link library (DLL)**. It contains system-level functions (like console I/O) that Windows programs can call.

Handy Procedures from Irvine32.lib

Here are some of the most useful ones:

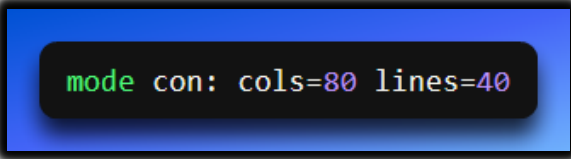
- **Input:** ReadInt, ReadKey, ReadString
- **Output:** WriteDec, WriteHex, WriteInt, WriteString, WriteChar
- **Strings:** Str_length, Str_copy, Str_compare, Str_trim, Str_ucase
- **Formatting:** SetTextColor, WriteBin, WriteBinB, WriteHexB
- **Debugging:** WriteStackFrame, WriteStackFrameName
- **File/Windows:** WriteToFile, WriteWindowsMsg, WaitMsg

These procedures save you from writing repetitive low-level code, letting you focus on the logic of your program.

Console Window Basics

The console window is the text-only interface you see when running command-line programs in Windows. You can customize it:

- **Size & layout** → Use the mode command to change columns and rows. Example:



```
mode con: cols=80 lines=40
```

- **Appearance** → Adjust font size and colors for readability.

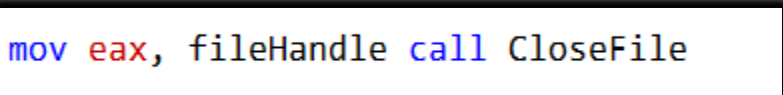
✦ **In short:** Procedures (PROC/ENDP + RET) give structure to assembly programs, CALL/RET manage control flow via the stack, and external libraries like Irvine32.lib or kernel32.dll provide ready-made building blocks for input, output, and system tasks.

INDIVIDUAL PROCEDURE DESCRIPTIONS

Here are descriptions of some of the procedures in the Irvine32 library:

1. CloseFile

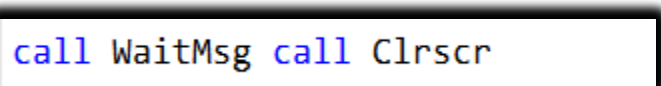
Closes a file that was previously created or opened using a file handle. Pass the file handle in EAX.



```
mov eax, fileHandle call CloseFile
```

2. Clrscr

Clears the console window. Typically called at the beginning and end of a program to clear the screen. If called at other times, you may need to pause using WaitMsg to allow the user to view existing screen contents.



```
call WaitMsg call Clrscr
```

3. CreateOutputFile

Creates a new disk file and opens it for writing. Provide the filename offset in EDX. Returns a valid file handle (32-bit integer) in EAX if successful, otherwise INVALID_HANDLE_VALUE.

```
mov edx, offset filename call CreateOutputFile
```

4. Crlf

The Crlf procedure advances the cursor to the beginning of the next line in the console window. It essentially moves to the next line. Sample Call:

```
call Crlf
```

This instruction calls the Crlf procedure. The Crlf procedure stands for "Carriage Return" and "Line Feed." It is used to advance the cursor to the beginning of the next line in the console window. Essentially, it simulates pressing the Enter key, which moves the cursor to the next line as in a typical text editor.

5. Delay

The Delay procedure pauses the program for a specified number of milliseconds. You need to set EAX to the desired interval before calling it. Sample Call:

```
mov eax, 1000 ; 1 second call Delay
```

This instruction calls the Delay procedure. The Delay procedure is used to pause the program for a specified number of milliseconds. In this case, it's designed to create a 1-second delay since EAX is set to 1000 (milliseconds).

6. DumpMem

The DumpMem procedure writes a range of memory to the console window in hexadecimal format. It requires you to pass the starting address in ESI, the number of units in ECX, and the unit size in EBX.

```
mov esi, OFFSET array ; starting OFFSET  
mov ecx, LENGTHOF array ; number of units  
mov ebx, TYPE array ; doubleword format  
call DumpMem
```

This call displays the content of the array in hexadecimal.

Here's an explanation of the provided assembly code:

mov esi, OFFSET array: This instruction loads the memory address (offset) of the array into the ESI register. It's setting up ESI to point to the starting address of the array in memory.

mov ecx, LENGTHOF array: This instruction loads the length of the array in terms of the number of elements into the ECX register. It determines how many units of data will be displayed when calling DumpMem.

mov ebx, TYPE array: This instruction loads the data type of the array elements into the EBX register. It specifies the format of the data when calling DumpMem. In this case, it indicates that the elements are in doubleword format.

call DumpMem: This instruction calls the DumpMem procedure. The DumpMem procedure is responsible for displaying a range of memory in hexadecimal format. It takes the starting address in ESI, the number of units in ECX, and the unit size in EBX (in this case, doubleword format). It will display the contents of the array specified by ESI, ECX, and EBX in hexadecimal format.

7. DumpRegs

The DumpRegs procedure displays the values of various registers (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP, EFL), as well as the Carry, Sign, Zero, Overflow, Auxiliary Carry, and Parity flags in hexadecimal.

```
call DumpRegs
```

This instruction calls the DumpRegs procedure. The DumpRegs procedure displays the values of various CPU registers and flags in hexadecimal format. It provides a snapshot of the CPU's current state, which can be helpful for debugging and understanding the program's execution.

8. GetCommandTail

GetCommandTail copies the program's command line into a null-terminated string and checks if the command line is empty.

To use it, you must provide the offset of a buffer in EDX where the command line will be stored. Sample Call:

```
.data
    cmdTail BYTE 129 DUP(0) ; empty buffer
.code
    mov edx, OFFSET cmdTail
    call GetCommandTail ; fills the buffer
```

9. GetMaxXY

The GetMaxXY procedure retrieves the size of the console window's buffer. The number of buffer columns is stored in DX, and the number of buffer rows is stored in AX. Sample Call:

```
.data
    rows BYTE ?
    cols BYTE ?
.code
    call GetMaxXY
    mov rows, al
    mov cols, dl
```

10. GetMseconds

GetMseconds returns the number of milliseconds elapsed since midnight on the host computer in the EAX register. It's useful for measuring time between events. Sample Call:

```
.data
    startTime DWORD ?
.code
    call GetMseconds
    mov startTime, eax
L1:
    ; (loop body)
    loop L1
    call GetMseconds
    sub eax, startTime ; EAX = loop time, in milliseconds
```

This example measures the execution time of a loop.

11. GetTextColor

The GetTextColor procedure retrieves the current foreground and background colors of the console window. It has no input parameters. The background color is stored in the upper four bits of AL, and the foreground color is stored in the lower four bits. Sample Call:

```
.data
    color BYTE ?
.code
    call GetTextColor
    mov color, al
```

Here's a breakdown of what this code does:

1. **.data color BYTE ?**: This declares a byte-sized variable named color in the data section. This variable will be used to store the color information retrieved from the console.
2. **.code call GetTextColor**: This line calls the GetTextColor procedure, which retrieves the current text color attributes of the console window. The result is returned in the AL register.

3. **mov color, al:** This instruction moves the value in the AL register (which contains the retrieved color information) into the color variable declared earlier. It stores the current text color in the color variable.

So, after executing these instructions, the color variable will contain the current text color attribute of the console window, allowing you to use or manipulate it in your program as needed.

12. Gotoxy

The Gotoxy procedure positions the cursor at a specific row and column in the console window. You should pass the row (Y-coordinate) in DH and the column (X-coordinate) in DL. Sample Call:

```
mov dh, 10 ; row 10
mov dl, 20 ; column 20
call Gotoxy ; position cursor
```

Here's a breakdown of what this code does:

mov dh, 10: This instruction moves the value 10 into the DH (Destination High) register, which represents the row number where you want to position the cursor. In this case, it sets the row to 10.

mov dl, 20: This instruction moves the value 20 into the DL (Destination Low) register, which represents the column number where you want to position the cursor. It sets the column to 20.

call Gotoxy: This is a procedure call to the Gotoxy procedure. When called, it takes the values in DH (row) and DL (column) and uses them to position the cursor in the console window.

So, after executing these instructions, the cursor will be moved to row 10 and column 20 in the console window. This can be useful for controlling the cursor's position when you want to display text or interact with the user at specific locations on the screen.

13. IsDigit

IsDigit checks whether the value in AL is the ASCII code for a valid decimal digit. If AL contains a valid decimal digit, it sets the Zero flag (ZF); otherwise, it clears ZF. Sample Call:

```
mov AL, somechar
call IsDigit
```

- Load the character into AL with `mov AL, somechar`.
- Call `IsDigit`.
- `IsDigit` checks if AL holds an ASCII digit ('0'-'9').
- If it is, the **Zero Flag (ZF)** is set.
- If not, ZF is cleared.
- After the call, just test ZF:
 - **ZF = 1** → **valid digit**
 - **ZF = 0** → **not a digit**

This is a quick way to validate input characters in assembly without writing extra comparison logic yourself. It leverages the processor's flags to give you a simple yes/no answer.

14. MsgBox

The `MsgBox` procedure displays a popup message box with an optional caption. Pass it the offset of a string in EDI to appear inside the box, and optionally, pass the offset of a string for the box's title in EBX. To leave the title blank, set EBX to zero.

```
.data
    caption BYTE "Dialog Title", 0
    HelloMsg BYTE "This is a pop-up message box.", 0dh, 0ah, "Click OK to continue...", 0
.code
    mov ebx, OFFSET caption
    mov edi, OFFSET HelloMsg
    call MsgBox
```

- MsgBox shows a popup with a message and an optional title.
- Put the **message string address** in EDX.
- Put the **title string address** in EBX (or set EBX = 0 for no title).
- Call MsgBox.
- The box appears with an **OK button**. When the user clicks it, execution continues.

15. MsgBoxAsk

The MsgBoxAsk procedure displays a popup message box with Yes and No buttons. It returns an integer in EAX that indicates which button was selected by the user (IDYES or IDNO).

```
.data
    caption BYTE "Survey Completed", 0
    question BYTE "Thank you for completing the test.", 0dh,0ah, "Receive results?", 0
.code
    mov ebx, OFFSET caption
    mov edx, OFFSET question
    call MsgBoxAsk ; check return value in EAX
```

Strings setup: In the .data section, you define two null-terminated strings:

- caption → the window title ("Survey Completed")
- question → the message ("Thank you for completing the test. Receive results?")

Register setup:

- mov ebx, OFFSET caption → loads the title's address.
- mov edx, OFFSET question → loads the message's address.

Procedure call: call MsgBoxAsk → shows a popup with **Yes** and **No** buttons.

Return value: When the user clicks a button, the procedure returns a code in EAX:

- IDYES (6) → user clicked **Yes**
- IDNO (7) → user clicked **No**

Next steps: You can branch your program logic based on EAX. For example:

```
cmp eax, IDYES
je ShowResults
cmp eax, IDNO
je ExitProgram
```

This gives you a simple way to ask the user a **binary choice** (Yes/No) and immediately act on their response without building a full UI.

16. OpenInputFile

```
.data
    filename BYTE "myfile.txt",0
.code
    mov edx, OFFSET filename
    call OpenInputFile
```

- OpenInputFile takes the filename address in EDX.
- On success, EAX holds a valid file handle.
- On failure, EAX = INVALID_HANDLE_VALUE.
- You typically define a string like "myfile.txt" in .data, load its address into EDX, then call the procedure.
- Afterward, check EAX to decide whether to proceed with reading or handle the error.

17. ReadString:

```
.data
    buffer BYTE 21 DUP(0) ; input buffer
    byteCount DWORD ? ; holds counter
.code
    mov edx, OFFSET buffer ; point to the buffer
    mov ecx, SIZEOF buffer ; specify max characters
    call ReadString ; input the string
    mov byteCount, eax ; number of characters
```

- ReadString reads input from the keyboard until **Enter** is pressed.
- You pass it:
 - ❖ EDX → address of the buffer where the text will go
 - ❖ ECX → maximum number of characters allowed
- When it finishes, the string is stored in the buffer and the **number of characters typed** is returned in EAX.
- You can save that count (e.g., into byteCount) for later use.

EAX comes directly from the ReadString procedure—it's how the procedure reports back how many characters the user entered.

18. SetTextColor:

```
mov eax, white + (blue SHL 4) ; white on blue
call SetTextColor
```

- SetTextColor changes how text appears in the console.
- You load EAX with a color attribute that combines **foreground** (text) and **background** colors.
- Example:
 - ❖ white → text color
 - ❖ (blue SHL 4) → background color (blue shifted into the background bits)
 - ❖ Together: white + (blue SHL 4) = white text on blue background.
- call SetTextColor applies this setting, so all text printed afterward uses that color scheme.

Foreground is the text color, background is the console backdrop. Shifting the background value ensures it's placed in the right bits before combining.

19. Str_length

```
.data
    buffer BYTE "abcde", 0
    bufLength DWORD ?
.code
    mov edx, OFFSET buffer ; point to string
    call Str_length ; EAX = 5
    mov bufLength, eax ; save length
```

- Str_length finds the length of a null-terminated string.
- You pass the string's address in EDX.
- It returns the length in EAX.
- Example: with "abcde", Str_length returns 5.
- That value can then be stored in a variable (like bufLength) for later use.

The procedure doesn't give you the string itself—it gives you the **count of characters**

20. WaitMsg:

```
call WaitMsg
```

- WaitMsg shows "**Press any key to continue...**" in the console.
- It pauses program execution until the user presses a key.
- Useful for giving the user time to read output before the program moves on.

That's all it does: display the prompt, wait for a key, then continue.

21. WriteInt:

```
mov eax, -12345  
call WriteInt ; displays: "-12345"
```

- WriteInt prints a 32-bit signed integer (from EAX) to the console in decimal format.
- If the value is negative, it shows the minus sign.
- Example:
 - ❖ `mov eax, -12345`
 - ❖ `call WriteInt` → Console displays **-12345**.

Whatever number you load into EAX is exactly what WriteInt outputs in decimal form.

22. WriteString:

Use WriteString to write a null-terminated string to the console window.

Pass the offset of the string in EDX.

```
.data
    greeting BYTE "Hello, Assembly Programmer!", 0
.code
    mov edx, OFFSET greeting
    call WriteString
```

- `mov edx, OFFSET greeting` → points EDX to the string's location.
- `call WriteString` → prints that string to the console.
- Result: the message **"Hello, Assembly Programmer!"** appears on screen.

WriteString simply takes the string's address in EDX and displays its text.

23. WriteToFile:

This procedure is used to write the contents of a buffer to an output file.

- Parameters: EAX (file handle), EDX (buffer address), ECX (number of bytes to write).
- Returns the number of bytes written if successful; otherwise, an error code.

```
.data
    fileHandle DWORD ?
    buffer BYTE "This is a test.", 0
.code
    mov eax, fileHandle
    mov edx, OFFSET buffer
    mov ecx, LENGTHOF buffer - 1 ; Length excluding the null terminator
    call WriteToFile
```

24. WriteWindowsMsg:

```
.code
; Some code that might generate an error
; ...

; After an error occurs, call WriteWindowsMsg to display the error message
call WriteWindowsMsg
```

This procedure outputs the **latest system error** to the console.

- It retrieves the error string generated when a system function fails.
- The string is then displayed so the user knows what went wrong.

It's mainly used for **diagnostics**—helping you see why a function call failed by showing the error message directly.

IMPLEMENTING THE PROCEDURES: USING IRVINE32

```
300 ; Library Test #1: Integer I/O (InputLoop.asm)
301 ; Tests the Clrscr, Crlf, DumpMem, ReadInt, SetTextColor,
302 ; WaitMsg, WriteBin, WriteHex, and WriteString procedures.
303
304 include Irvine32.inc
305
306 .data
307 COUNT = 4
308 BlueTextOnGray = blue + (lightGray * 16)
309 DefaultColor = lightGray + (black * 16)
310 arrayD SDWORD 12345678h, 1A4B2000h, 3434h, 7AB9h
311 prompt BYTE "Enter a 32-bit signed integer: ", 0
312
313 .code
314
315 main PROC
316     ; Select blue text on a light gray background
317     mov eax, BlueTextOnGray
318     call SetTextColor
319
320     ; Clear the screen
321     call Clrscr
322
323     ; Display an array using DumpMem.
324     mov esi, OFFSET arrayD      ; starting OFFSET
325     mov ebx, TYPE arrayD        ; doubleword = 4 bytes
326     mov ecx, LENGTHOF arrayD    ; number of units in arrayD
327     call DumpMem                ; display memory
328
329     ; Ask the user to input a sequence of signed integers
330     call Crlf                   ; new line
331     mov ecx, COUNT
332
333 L1:
334     mov edx, OFFSET prompt
335     call WriteString            ; prompt the user
336     call ReadInt                ; input integer into EAX
337     call Crlf                  ; new line
338
339     ; Display the integer in decimal, hexadecimal, and binary
340     call WriteInt                ; display in signed decimal
341     call Crlf
342     call WriteHex                ; display in hexadecimal
343     call Crlf
344     call WriteBin                ; display in binary
345     call Crlf
346
347     Loop L1                    ; repeat the loop
348
349     ; Return the console window to default colors
350     call WaitMsg                ; "Press any key..."
351     mov eax, DefaultColor
352     call SetTextColor
353     call Clrscr
354
355     exit
356 main ENDP
357 END main
```

This program is a showcase of how to use the **Irvine32 library** to handle console input/output, memory display, and random number generation. Let's break it down step by step.

Setup and Definitions

The program begins with comments describing its purpose and includes Irvine32.inc.

It defines constants like:

- **COUNT** → how many times the user will enter integers.
- **BlueTextOnGray** → a color setting for text.
- **DefaultColor** → the standard console color.

An array named arrayD is declared with four signed doubleword integers.

A **prompt message** is also defined to guide user input.

Main Procedure (main PROC)

1. **SetTextColor** → Changes text color to blue on a light gray background.
2. **Clsrscr** → Clears the console screen.
3. **DumpMem** → Displays the contents of arrayD in memory.
4. **Loop (L1)** → Repeats COUNT times to collect user input.
 - ❖ **WriteString** → Prompts the user to enter an integer.
 - ❖ **ReadInt** → Reads the integer into EAX.
 - ❖ **WriteInt, WriteHex, WriteBin** → Displays the entered integer in decimal, hexadecimal, and binary formats.
5. After the loop:
 - ❖ **WaitMsg** → Pauses until a key is pressed.
 - ❖ **Clsrscr** → Clears the screen again.
 - ❖ Console colors are reset to default.

Random Number Generation

Two procedures demonstrate different random number functions:

- **Rand1** → Generates 10 pseudo-random *unsigned* integers using Random32 (range: 0 to 4,294,967,294).
- **Rand2** → Generates 10 pseudo-random *signed* integers using RandomRange (range: -50 to +49).

Before calling these, the program runs **Randomize** to initialize the random number generator.

Program End

After running both random number procedures, the program finishes with END main.

Big Picture

This program demonstrates:

- **Console control** → Changing colors, clearing the screen, waiting for input.
- **Memory display** → Using DumpMem to show array contents.
- **User interaction** → Reading integers and displaying them in multiple formats.
- **Random numbers** → Generating both unsigned and signed values with Irvine32's built-in procedures.

```
; Library Test #3: Performance Timing (TestLib3.asm)
; Calculate the elapsed execution time of a nested loop.
```

```
include Irvine32.inc
```

```
.data
```

```
OUTER_LOOP_COUNT = 3
```

```
startTime DWORD ?
```

```
msg1 BYTE "Please wait...", 0dh, 0ah, 0
```

```
msg2 BYTE "Elapsed milliseconds: ", 0
```

```
.code
```

```
main PROC
```

```
    mov edx, OFFSET msg1 ; "Please wait..."
```

```
    call WriteString
```

```
    ; Save the starting time
```

```
    call GetMSeconds
```

```
    mov startTime, eax
```

```
    ; Start the outer loop
```

```
    mov ecx, OUTER_LOOP_COUNT
```

```
L1:
```

```
    call innerLoop
```

```
    loop L1
```

```
    ; Calculate the elapsed time
```

```
    call GetMSeconds
```

```
    sub eax, startTime
```

```
    ; Calculate the elapsed time
```

```
    call GetMSeconds
```

```
    sub eax, startTime
```

```
    ; Display the elapsed time
```

```
    mov edx, OFFSET msg2 ; "Elapsed milliseconds: "
```

```
    call WriteString
```

```
    call WriteDec ; Write the milliseconds
```

```
    call Crlf
```

```
    exit
```

```
main ENDP
```

```
innerLoop PROC
```

```
    push ecx ; Save current ECX value
```

```
    mov ecx, 0FFFFFFh ; Set the loop counter
```

```
L1:
```

```
    mul eax ; Use up some cycles
```

```
    mul eax
```

```
    mul eax
```

```
    loop L1 ; Repeat the inner loop
```

```
    pop ecx ; Restore ECX's saved value
```

```
    ret
```

```
innerLoop ENDP
```

```
END main
```

Irvine32 vs Irvine64

- **Irvine32** → built for 32-bit assembly, uses registers like EAX, EBX, EDX.
- **Irvine64** → built for 64-bit assembly, uses registers like RAX, RBX, RDX.

Similar Procedures, Different Context

- **WriteHex32 vs WriteHex64** → both print integers in hex, but one works with 32-bit values (EAX), the other with 64-bit (RAX).
- **WriteHexB** → same idea, but in Irvine64 you can choose the display size (1, 2, 4, or 8 bytes) via RBX.
- **WriteString** → conceptually identical, but now you pass the string's 64-bit offset in RDX.

Key Difference

The **functionality feels familiar**, but the **calling conventions and register preservation rules** change in 64-bit assembly. For example:

- Volatile (not preserved): RAX, RCX, RDX, R8–R11.
- Preserved: RBX, RBP, RSI, RDI, R12–R15.

IRVINE 64

In 64-bit assembly, calling a subroutine means putting your input parameters into the right registers, then using the CALL instruction e.g.

```
mov rax, 12345678h
call WriteHex64
```

Here, WriteHex64 is a subroutine that prints the value in RAX. The parameter is passed via a register, not the stack.

Using the PROTO Directive

When you want to call external procedures (like Windows API functions or Irvine64 library routines), you declare them at the top of your program with PROTO.

This tells the assembler: *“These procedures exist, even though I’m not defining them here.”*

```
ExitProcess PROTO      ; Located in the Windows API
WriteHex64 PROTO       ; Located in the Irvine64 library
```

- **ExitProcess** → comes from the Windows API.
- **WriteHex64** → comes from the Irvine64 library.

Declaring them ensures the assembler knows how to generate the correct call instructions.

Microsoft x64 Calling Convention

This is the standard way parameters are passed in 64-bit Windows programs (used by C/C++ compilers and the Windows API).

Key Rules

- **CALL** → pushes an 8-byte return address onto the stack (since addresses are 64-bit).
- **First four parameters** → go into registers:
 - ❖ RCX → 1st parameter
 - ❖ RDX → 2nd parameter
 - ❖ R8 → 3rd parameter
 - ❖ R9 → 4th parameter
- **Shadow space** → Caller must reserve 32 bytes on the stack for potential register saves.
- **Stack alignment** → RSP must be aligned to a 16-byte boundary before a call.

Example: AddFour Subroutine

```
AddFour PROC
    ; Input parameters: RCX, RDX, R8, R9
    ; Output: RAX contains the sum of the input parameters

    ; Add the input parameters
    mov rax, rcx
    add rax, rdx
    add rax, r8
    add rax, r9

    ret
AddFour ENDP
```

- **Inputs:** RCX, RDX, R8, R9 (four integers).
- **Output:** RAX (the sum).
- **RET:** returns control to the caller.

This demonstrates how parameters flow through registers and how results are returned.

Stack Alignment

Before calling a subroutine, the stack pointer (**RSP**) must be aligned on a 16-byte boundary. This ensures proper memory access and avoids runtime errors. Remember: the **CALL** instruction itself pushes 8 bytes, so you need to account for that when aligning.

Irvine64 vs. Windows API

- **Irvine64 library** → You don't need to worry about the Microsoft x64 calling convention when using its procedures; they're already set up for you.
- **Windows API / C/C++ functions** → You *must* follow the x64 calling convention rules (registers, shadow space, stack alignment).

🌟 Big Picture

- Use **registers** (RCX, RDX, R8, R9) for parameters.
- Use CALL to jump into a subroutine.
- Use RET to return.
- Declare external procedures with PROTO.
- Always keep the **stack aligned**.

This structure makes 64-bit assembly predictable and consistent, especially when mixing your code with Windows API or external libraries.

Breaking Down Proto Directive

Certainly, here's a short assembly code snippet that demonstrates the use of the PROTO directive and stack alignment for a simple subroutine 🌟:

```
.data
    result QWORD 0    ; Declare a variable to store the result
.code
main PROC
    sub rsp, 8        ; Align the stack pointer to a 16-byte boundary
    mov rcx, 5        ; First parameter
    mov rdx, 3        ; Second parameter

    call AddNumbers   ; Call the subroutine
    add rsp, 8        ; Restore the stack pointer

    ; The result is now stored in the "result" variable
    ; You can use it as needed
    ; Exit the program
    mov ecx, 0
    call ExitProcess

AddNumbers PROC
    ; Parameters are already in RCX and RDX
    add rax, rcx      ; Add the first parameter to RAX
    add rax, rdx      ; Add the second parameter to RAX
    mov [result], rax ; Store the result in the "result" variable
    ret
AddNumbers ENDP

END main
```

I. Declaring External Procedures

We use the `PROTO` **directive** to declare `ExitProcess`. This tells the assembler that `ExitProcess` exists, even though it's defined outside our program (in the Windows API). That way, when we call it later, the assembler knows how to handle it.

II. Stack Alignment

At the start of the main procedure, we subtract **8 bytes** from `rsp`.

- Why? Because in 64-bit assembly, the stack pointer must be aligned on a **16-byte boundary** before making a call.
- Since the `CALL` instruction itself pushes 8 bytes (the return address), subtracting 8 ensures proper alignment. After the subroutine call, we add 8 back to `rsp` to restore the stack.

III. Calling the Subroutine

We call `AddNumbers`, passing two parameters:

- `rcx` → first parameter
- `rdx` → second parameter

Inside `AddNumbers`, these values are added together, and the result is stored in the variable `result`.

IV. Exiting the Program

Finally, we call `ExitProcess` to end the program cleanly. Passing 0 as the exit code signals that everything completed successfully.

✨ Big Picture

- **PROTO** → declares external procedures.
- **Stack alignment** → ensures calls work correctly in 64-bit mode.
- **CALL + RET** → handle subroutine execution and return.
- **ExitProcess** → ends the program.

This is a simplified example, but the same principles scale up. For more complex scenarios—especially when calling Windows API functions or C/C++ routines—you’d follow the full **Microsoft x64 calling convention** (parameters in RCX, RDX, R8, R9, shadow space reserved, stack aligned).

Runtime Stack in the CallProc_64 Program ✂

The runtime stack is a special memory region used to keep track of function calls, return addresses, and local variables. In the CallProc_64 program, the stack pointer (**RSP**) changes as the program executes, and the diagram illustrates these changes step by step.

FIGURE 5–11 Runtime stack for the CallProc_64 program.

(return to OS)	0 1 A F E 4 8
	0 1 A F E 4 0
shadow p1	0 1 A F E 3 8
shadow p2	0 1 A F E 3 0
shadow p3	0 1 A F E 2 8
shadow p4	0 1 A F E 2 0
(return to main)	0 1 A F E 1 8

📖 Step-by-Step Stack Behavior

1. Before the program starts

- ❖ RSP = 01AFE48
- ❖ The operating system has already pushed the return address (subtracting 8 from the stack pointer). This ensures the program knows where to return when it finishes.

2. After line 10 executes

- ❖ RSP = 01AFE40
- ❖ The stack is now aligned on a **16-byte boundary**, which is required for performance and correctness in 64-bit programs.
- ❖ Alignment is often achieved by pushing a dummy value before a call and popping it afterward.

3. After line 11 executes

- ❖ RSP = 01AFE20
- ❖ **32 bytes of shadow space** have been reserved.
- ❖ Shadow space is used to store register parameters temporarily (RCX, RDX, R8, R9) and to preserve callee-saved registers if needed.

4. Inside the AddFour procedure

- ❖ RSP = 01AFE18
- ❖ The caller's return address has been pushed onto the stack. This is how the CPU knows where to jump back after the procedure finishes.

5. After AddFour returns

- ❖ RSP = 01AFE20
- ❖ The stack pointer is restored to its previous value. The return address has been popped, and execution continues in the caller.

6. At the end of main

- ❖ The program executes RET, returning control to the operating system.
- ❖ If instead it had called ExitProcess, the program itself would have been responsible for restoring the stack pointer before exiting.

🌟 Big Picture

- **CALL** → pushes the return address onto the stack.
- **RET** → pops the return address and resumes execution at that location.
- **Shadow space** → ensures parameters and registers have reserved memory slots.
- **Alignment** → keeps the stack pointer on a 16-byte boundary for efficiency.

The runtime stack diagram for CallProc_64 shows how carefully the stack is managed in 64-bit assembly: every call, return, and alignment step ensures structured program flow and stable execution.

QUESTIONS

Question 1: Which instruction pushes all of the 32-bit general-purpose registers on the stack?

Answer 1: The instruction that pushes all of the 32-bit general-purpose registers on the stack is PUSHA.

Question 2: Which instruction pushes the 32-bit EFLAGS register on the stack?

Answer 2: The instruction that pushes the 32-bit EFLAGS register on the stack is PUSHFD.

Question 3: Which instruction pops the stack into the EFLAGS register?

Answer 3: The instruction that pops the stack into the EFLAGS register is POPFD.

Question 4: Challenge: Another assembler (called NASM) permits the PUSH instruction to list multiple specific registers. Why might this approach be better than the PUSHAD instruction in MASM? Here is a NASM example: PUSH EAX EBX ECX

Answer 4: NASM's approach of allowing the PUSH instruction to list multiple specific registers can be better in some cases because it provides more flexibility. It allows you to choose which registers to push onto the stack, whereas PUSHAD in MASM pushes all the general-purpose registers. This can save stack space and execution time when you only need to save a subset of registers.

Question 5: Challenge: Suppose there were no PUSH instruction. Write a sequence of two other instructions that would accomplish the same as push eax.

Answer 5: If there were no PUSH instruction, you could achieve the same result as PUSH EAX using the following two instructions:

```
sub esp, 4  
mov [esp], eax
```

Question 6: (True/False): The RET instruction pops the top of the stack into the instruction pointer.

Answer 6: False. The RET instruction pops the return address from the stack into the instruction pointer (EIP).

Question 7: (True/False): Nested procedure calls are not permitted by the Microsoft assembler unless the NESTED operator is used in the procedure definition.

Answer 7: False. Nested procedure calls are permitted without the need for the NESTED operator in the Microsoft assembler.

Question 8: (True/False): In protected mode, each procedure call uses a minimum of 4 bytes of stack space.

Answer 8: False. In protected mode, each procedure call doesn't necessarily use a minimum of 4 bytes of stack space. The actual stack space used depends on the number of parameters and local variables.

Question 9: (True/False): The ESI and EDI registers cannot be used when passing 32-bit parameters to procedures.

Answer 9: False. The ESI and EDI registers can be used when passing 32-bit parameters to procedures.

Question 10: (True/False): The ArraySum procedure (Section 5.2.5) receives a pointer to any array of doublewords.

Answer 10: False. The ArraySum procedure from Section 5.2.5 expects a pointer to an array of doublewords specifically.

Question 11: (True/False): The USES operator lets you name all registers that are modified within a procedure.

Answer 11: True. The USES operator lets you specify all registers that are modified within a procedure.

Question 12: (True/False): The USES operator only generates PUSH instructions, so you must code POP instructions yourself.

Answer 12: True. The USES operator generates PUSH instructions, so you need to code the corresponding POP instructions yourself.

Question 13: (True/False): The register list in the USES directive must use commas to separate the register names.

Answer 13: True. The register list in the USES directive must use commas to separate the register names.

Question 14: Which statement(s) in the ArraySum procedure (Section 5.2.5) would have to be modified so it could accumulate an array of 16-bit words? Create such a version of ArraySum and test it.

Answer 14: To accumulate an array of 16-bit words, you would need to modify the mov eax, [esi] and add esi, 4 statements to work with 16-bit words, like this:

```
mov ax, [esi]
add esi, 2
```

**Question 15: What will be the final value in EAX after these instructions execute?
push 5 push 6 pop eax pop eax**

Answer 15: EAX will equal 5 after these instructions execute. The second pop eax instruction will overwrite the previous value of EAX.

Question 16: Which statement is true about what will happen when the example code runs?

```
1: main PROC
2: push 10
3: push 20
4: call Ex2Sub
5: pop eax
6: INVOKE ExitProcess,0
7: main ENDP

10: Ex2Sub PROC
11: pop eax
12: ret
13: Ex2Sub ENDP
```

Answer 16: a. EAX will equal 10 on line 6.

Question 17: Which statement is true about what will happen when the example code runs?

```
1: main PROC
2: mov eax,30
3: push eax
4: push 40
5: call Ex3Sub
6: INVOKE ExitProcess,0
7: main ENDP

10: Ex3Sub PROC
11: pusha
12: mov eax,80
13: popa
14: ret
15: Ex3Sub ENDP
```

Answer 17: d. The program will halt with a runtime error on Line 11 because there's no matching pop for the pusha instruction.

Question 18: Which statement is true about what will happen when the example code runs?

```
1: main PROC
2: mov eax,40
3: push offset Here
4: jmp Ex4Sub
5: Here:
6: mov eax,30
7: INVOKE ExitProcess,0
8: main ENDP

10: Ex4Sub PROC
11: ret
12: Ex4Sub ENDP
```

Answer 18: b. The program will halt with a runtime error on Line 4 because there's no matching pop for the push instruction.

Question 19: Which statement is true about what will happen when the example code runs?

```
1: main PROC
2: mov edx,0
3: mov eax,40
4: push eax
5: call Ex5Sub
6: INVOKE ExitProcess,0
7: main ENDP

10: Ex5Sub PROC
11: pop eax
12: pop edx
13: push eax
14: ret
15: Ex5Sub ENDP
```

Answer 19: a. EDX will equal 40 on line 6.

Question 20: What values will be written to the array when the following code executes?

```
592 .data
593 array DWORD 4 DUP(0)
594
595 .code
```

In the provided code, you've declared an array named array with four double word (DWORD) elements, and you've initialized each element to 0 using 4 DUP(0). Therefore, when this code is executed, the array will contain the following values:

array[0] will be 0.

array[1] will be 0.

array[2] will be 0.

array[3] will be 0.

So, all elements in the array will have the value 0.

GOD LEVEL QUESTIONS

Exercise 1: Draw Text Colors

```
598 include Irvine32.inc
599
600 .data
601     colors BYTE 2, 4, 6, 14    ; Colors: Green, Red, Yellow, White
602     message BYTE "Hello, Colors!",0
603
604 .code
605 main PROC
606     mov ecx, 4    ; Number of colors
607     mov esi, 0    ; Index for colors array
608
609 loop_colors:
610     mov eax, colors[esi]
611     call SetTextColor
612     mov edx, OFFSET message
613     call WriteString
614     call Crlf
615
616     inc esi
617     loop loop_colors
618
619     call WaitMsg    ; Wait for a key press
620     call Clrscr     ; Clear the screen
621     call ExitProcess
622 main ENDP
623 END main
```

Exercise 2: Linking Array Items

```
.data
start DWORD 1
chars BYTE 'H', 'A', 'C', 'E', 'B', 'D', 'F', 'G'
links DWORD 0, 4, 5, 6, 2, 3, 7, 0
outputArray BYTE 8 DUP(?) ; To store the characters in order
.code
main PROC
    mov edi, OFFSET outputArray ; Destination for output characters
    mov esi, start ; Start index
    mov ecx, 8 ; Number of characters to locate
    traverse_links:
        mov al, chars[esi] ; Load character
        mov [edi], al ; Store it in outputArray
        inc edi ; Move to the next position in outputArray
        ; Get the next link index
        mov eax, esi
        mov ebx, 4 ; Size of DWORD (4 bytes)
        mul ebx ; Multiply esi by 4
        mov esi, links[eax] ; Get the next link index

        loop traverse_links ; Repeat for all characters

    ; Display the characters in outputArray
    mov edx, OFFSET outputArray
    call WriteString
    call Crlf

    call WaitMsg ; Wait for a key press
    call ExitProcess

main ENDP
END main
```

Exercise 3: Simple Addition (1)

```
662 include Irvine32.inc
663 .data
664     prompt1 BYTE "Enter the first integer: ", 0
665     prompt2 BYTE "Enter the second integer: ", 0
666     resultMsg BYTE "The sum is: ", 0
667     buffer1 DWORD ?
668     buffer2 DWORD ?
669     result DWORD ?
670 .code
671     main PROC
672         call Clrscr
673         mov edx, OFFSET prompt1
674         call WriteString
675         call ReadInt
676         mov buffer1, eax
677
678         mov edx, OFFSET prompt2
679         call WriteString
680         call ReadInt
681         mov buffer2, eax
682
683         ; Add the integers
684         mov eax, buffer1
685         add eax, buffer2
686         mov result, eax
687
688         mov edx, OFFSET resultMsg
689         call WriteString
690         mov eax, result
691         call WriteInt
692
693         call WaitMsg ; Wait for a key press
694         call ExitProcess
695     main ENDP
696 END main
```

Exercise 4: Simple Addition (2)

```
700 include Irvine32.inc
701 .data
702     prompt1 BYTE "Enter the first integer: ", 0
703     prompt2 BYTE "Enter the second integer: ", 0
704     resultMsg BYTE "The sum is: ", 0
705     buffer1 DWORD ?
706     buffer2 DWORD ?
707     result DWORD ?
708
709 .code
710     main PROC
711         mov ecx, 3 ; Repeat the process three times
712         loop_repeat:
713             call Clrscr
714
715             mov edx, OFFSET prompt1
716             call WriteString
717             call ReadInt
718             mov buffer1, eax
719
720             mov edx, OFFSET prompt2
721             call WriteString
722             call ReadInt
723             mov buffer2, eax
724
725             ; Add the integers
726             mov eax, buffer1
727             add eax, buffer2
728             mov result, eax
729
730             mov edx, OFFSET resultMsg
731             call WriteString
732             mov eax, result
733             call WriteInt
734
735             call WaitMsg ; Wait for a key press
736
737             dec ecx
738             jnz loop_repeat ; Repeat the process three times
739
740             call ExitProcess
741     main ENDP
742 END main
```

Exercise 5: BetterRandomRange Procedure

```
746 include Irvine32.inc
747
748 .data
749
750 .code
751 BetterRandomRange PROC
752     ; Input: EBX = Lower bound (M)
753     ;         EAX = Upper bound (N)
754     ; Output: EAX = Random number between M and N-1
755
756     sub eax, ebx ; Calculate the range (N - M)
757     add eax, 1   ; Include the upper bound itself
758     call RandomRange
759     add eax, ebx ; Offset the result by M (lower bound)
760     ret
761 BetterRandomRange ENDP
762
763 main PROC
764     mov ecx, 50 ; Repeat 50 times
765     loop_repeat:
766         call Clrscr
767
768         mov ebx, -300 ; Lower bound
769         mov eax, 100 ; Upper bound
770         call BetterRandomRange
771
772         ; Display the randomly generated value
773         mov edx, eax
774         call WriteInt
775         call Crlf
776
777         call WaitMsg ; Wait for a key press
778         dec ecx
779         jnz loop_repeat ; Repeat the process 50 times
780
781     call ExitProcess
782 main ENDP
783 END main
```

Exercise 6: Draw Text Colors (Part 2)

The Challenge: Write a program that displays the same string in four different colors using a loop. Use the SetTextColor procedure from the Irvine32 library.

```
INCLUDE Irvine32.inc

.data
message BYTE "Colorful Text", 0
; Array of color constants to make the loop cleaner
colors BYTE lightRed, lightBlue, lightGreen, yellow

.code
main PROC
    mov ecx, 4           ; Loop counter: 4 colors
    mov esi, 0           ; Index for our colors array

colorLoop:
    ; 1. Prepare and set the color
    movzx eax, colors[esi] ; Move color value into EAX (zero-extended)
    call SetTextColor      ; Irvine32 library procedure

    ; 2. Display the string
    mov edx, OFFSET message
    call WriteString
    call Crlf

    ; 3. Update index and loop
    inc esi               ; Move to next color in array
    loop colorLoop

    ; Reset color to white before exiting so the console isn't messy
    mov eax, white
    call SetTextColor

    exit
main ENDP
END main
```

🌟 Big Picture

- **Procedures** make the program modular: one handles looping, one handles color selection, and others handle specific colors.
- **Reusability:** You can easily expand this to more colors or reuse SetTextColor in other programs.
- **Clarity:** Breaking tasks into procedures makes the program easier to read, test, and maintain.

👉 This closes the **procedures topic** beautifully: you've seen how PROC/ENDP, CALL, and RET work together, how parameters flow, and how modular design makes assembly programs more powerful.