Irvine64 library

To call a subroutine in 64-bit assembly, you need to place input parameters in registers and then use the CALL instruction. For example:

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

In this example, WriteHex64 is a subroutine being called with rax as an input parameter.

Using the PROTO Directive: To inform the assembler about external procedures (subroutines) that you plan to call but are not defined within your program, you should use the PROTO directive at the top of your program.

This helps the assembler understand the calling conventions and parameter types for these external procedures. For instance:

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

In this code, ExitProcess and WriteHex64 are identified as external procedures, and the PROTO directive provides information about their usage.

These guidelines are essential for proper procedure calls in 64-bit assembly with the Irvine64 library, ensuring that the correct calling conventions are followed and that the assembler can generate the appropriate code for calling external procedures.

The additional content you've mentioned explains the x64 calling convention, which is used in 64-bit programs on the x86-64 architecture, such as Windows API functions and C/C++ programs. Here are the key points from this section:

Microsoft x64 Calling Convention: This is a consistent scheme for

passing parameters and calling procedures in 64-bit programs. It's followed by C/C++ compilers and the Windows API. Some of its characteristics include:

The CALL instruction subtracts 8 from the **RSP (stack pointer) register** because addresses are 64 bits long.

The first four parameters passed to a procedure are placed in the **RCX , RDX, R8** and **R9** registers, in that order.

If only one parameter is passed, it goes in **RCX**, and so on.

The caller's responsibility is to **allocate at least 32 bytes of shadow space** on the runtime stack to optionally save register parameters.

When calling a **subroutine, the stack pointer (RSP) must be aligned on a 16-byte boundary**, considering the 8 bytes pushed by the CALL instruction.

Sample Program: The provided sample program demonstrates how to call a subroutine (AddFour) using the Microsoft x64 calling convention.

```
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
```

It adds four input parameters placed in RCX, RDX, R8, and R9 and saves the sum in RAX. The program aligns the stack pointer, sets the parameters, and calls the subroutine.

This code defines the "AddFour" subroutine, which takes four input parameters (RCX, RDX, R8, and R9) and calculates their sum, storing the result in RAX. The ret instruction is used to return from the subroutine.

You can include this code in your assembly program to call the "AddFour" subroutine using the Microsoft x64 calling convention, as demonstrated in the previous content.

PROTO Directive: The PROTO directive is used to declare external procedures (subroutines) at the top of your program. It helps the assembler understand how to call these procedures correctly, including their names and parameter types.

Stack Alignment: Ensuring that the stack pointer (RSP) is properly aligned on a 16-byte boundary is crucial when working with the x64 calling convention. This alignment is essential to maintain proper stack integrity during procedure calls.

Usage in Irvine64 Library: It's worth noting that when you work with the Irvine64 library (used in these examples), you don't need to adhere to the Microsoft x64 calling convention for library procedures. It's primarily required when calling Windows API functions or external C/C++ functions.

These explanations provide a detailed understanding of how the Microsoft x64 calling convention works and how to apply it in your 64-bit assembly programs, particularly when interfacing with external functions and libraries.

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
```

In this code:

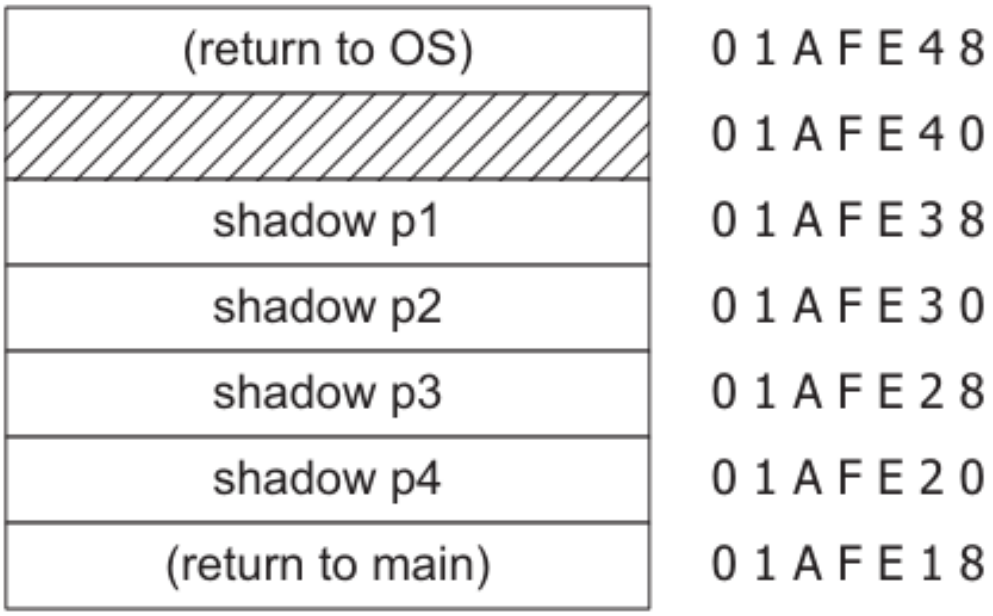
We use the PROTO directive for the ExitProcess procedure, which is an external procedure used to exit the program.

We ensure proper stack alignment by subtracting 8 from rsp at the beginning of the main procedure and then adding 8 to rsp to restore it after the subroutine call.

We call a subroutine AddNumbers with two parameters (rcx and rdx), which adds these parameters and stores the result in the result variable.

Finally, we exit the program using ExitProcess.
Please note that this is a simplified example, and in practice, you would use the Microsoft x64 calling convention for more complex scenarios or when calling external functions.

Figure 5–11 Runtime stack for the CallProc_64 program.



The runtime stack for the CallProc_64 program is a diagram that shows how the stack changes as the program executes. The stack is a region of memory that is used to store function calls, return addresses, and local variables.

The diagram shows the following:

The stack pointer (RSP) starts at address 01AFE48 before the program is called. When the OS calls the program, it subtracts 8 from the stack pointer to push the return address onto the stack.

After line 10 of the program executes, the stack pointer is at address 01AFE40, showing that the stack has been properly aligned on a 16-byte boundary.

After line 11 of the program executes, the stack pointer is at address 01AFE20, showing that 32 bytes of shadow space have been reserved on the stack.

Inside the AddFour procedure, the stack pointer is at address 01AFE18, showing that the caller's return address has been pushed onto the stack.

After AddFour returns, the stack pointer is again at address 01AFE20, the same value it had before calling AddFour. When the program

reaches the end of the main procedure, it returns to the OS by executing a RET instruction.

If the program had chosen to execute an ExitProcess instruction instead, it would have been responsible for restoring the stack pointer to the way it was when the main procedure began to execute.

Here is a more detailed explanation of each line in the diagram:

Before line 10 executed, RSP = 01AFE48.

This tells us that RSP was equal to 01AFE50 before the OS called our program. (The CALL instruction subtracts 8 from the stack pointer.)

The RSP register is the stack pointer register. It contains the address of the top of the stack. The CALL instruction pushes the return address onto the stack, which is the address of the instruction that will be executed after the called function returns.

After line 10 executed, RSP = 01AFE40, showing that the stack was properly aligned on a 16-byte boundary.

The stack must be aligned on a 16-byte boundary for performance reasons. The stack is aligned by pushing a dummy value onto the stack before calling a function and popping the dummy value off the stack after the function returns.

After line 11 executed, RSP = 01AFE20, showing that 32 bytes of shadow space were reserved at addresses 01AFE20 through 01AFE3F.

Shadow space is used to store the shadow registers of a function. Shadow registers are used to save the values of the callee-saved registers before a function is called and restore them after the function returns.

Inside the AddFour procedure, RSP = 01AFE18, showing that the caller’s return address had been pushed on the stack. When a function is called, the caller's return address is pushed onto the stack.

This is so that the function knows where to return to when it is finished.

After AddFour returned, RSP again was equal to 01AFE20, the same value it had before calling AddFour. When a function returns, the stack pointer is restored to the value it had before the function was called.

This is done by popping the caller's return address off the stack.

Overall, the runtime stack diagram for the CallProc_64 program shows how the stack is used to store function calls, return addresses, and local variables.