Author: Kunal Varudkar
LinkedIn: https://www.linkedin.com/in/kunal-varudkar

x86 Assembly language :- Prologue and Epilogue

A) call Instruction

In simple term, call instruction is used to call the procedure which you have defined in your assembly code. Call perform two operations, which are as follow

- Push EIP to stack
- Perform jump to the procedure.

Syntax:-

call <label>
ret

The main concept of call instruction:

- Before performing a call operation, the EIP register is pushed onto the stack, so that it can execute the next instruction after executing procedure.
- After the execution of procedure is completed, the EIP register is PUSHED from stack and the program jump to that memory address to which the EIP was pointing.

```
global _start

section .text

proc1:

        mov ebx, 0x1
        ret

_start:

        call proc1
        mov eax, 0x1
        int 0x80
```

Figure 1: simple code for illustrating use of **call**

Author: Kunal Varudkar
LinkedIn: https://www.linkedin.com/in/kunal-varudkar

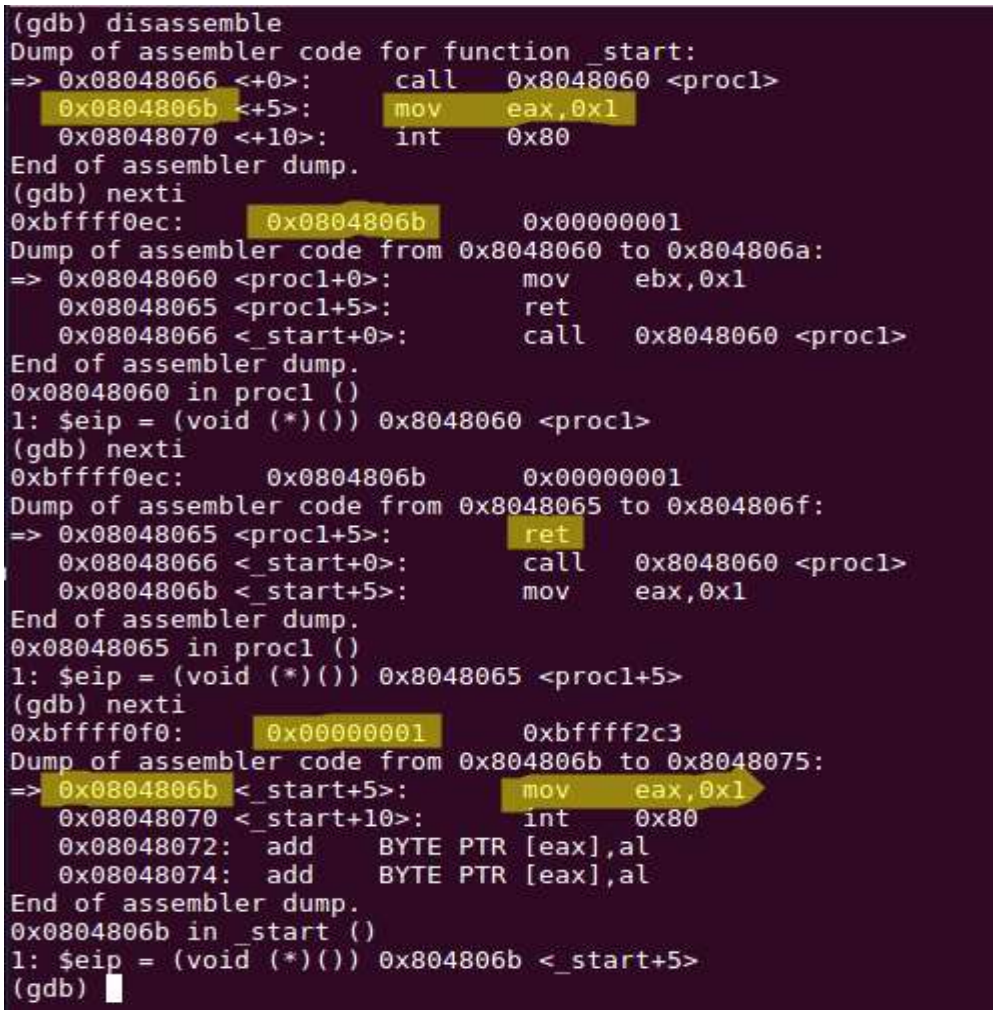

Figure 2: PUSH & POP of EIP

B) Prologue and Epilogue

The concept of prologue and epilogue comes into picture when we use stack in our function, because when we use the stack it becomes important for use to preserve the state (value) of stack pointer (esp) as we want to execute our code in which it is intended to run.

Use of stack in function may change the value of stack pointer, hence to avoid this we use prologue and epilogue. For doing so we use EBP register

- For preserving state of stack we use a register EBP (base pointer)
- EBP is going to hold the value of ESP which is pointing, top of the stack.

Author: Kunal Varudkar
LinkedIn: https://www.linkedin.com/in/kunal-varudkar

Consider the program

```
global _start

section .text

proc1:

        mov ebp, esp
        sub esp, 2
        mov [esp], byte 'P'
        mov [esp+1], byte 'D'
        mov eax, 0x4
        mov ebx, 0x1
        mov ecx, esp
        mov edx, 0x2
        int 0x80
        mov esp, ebp
        ret

_start:

        call proc1
        mov eax, 0x1
        mov ebx, 0x1
        int 0x80
```

Figure 3

- In the above code, we are preserving the value of ESP in EBP (mov ebp, esp), because from next instruction, the value of stack is going to be changed (sub esp, 2).
- Later we again, restore the value of ESP which we have stored in EBP (mov esp, ebp)

The above mentioned technique for storing the top of the stack will be complicated by the fact that if function call another function that stores the ESP into EBP like the above mentioned case does, it will alter the EBP before we return and this will affect our program execution.

So a common technique is to PUSH the value of previous value of EBP onto the stack, when you enter the function. This is going to preserve the old value of EBP by just pushing it onto the stack and then you POP it back into EBP, before you return from function

This technique of pushing and poping will be used for nested function call without the function interfering each other's stack.

Author: Kunal Varudkar
LinkedIn: https://www.linkedin.com/in/kunal-varudkar

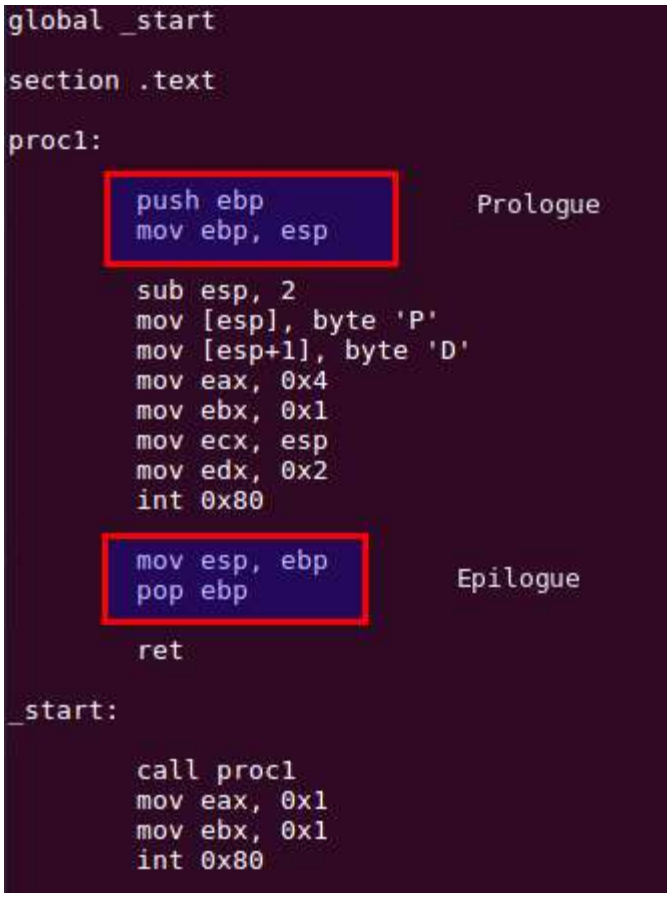

Figure 4: Prologue and Epilogue

- Technique of preserving and allocating space is known as **Prologue**
- Where the portion of restoring the stack and returning is known as **Epilogue**

Find the code on the given link
GitHub https://github.com/kunalvarudkar/x86-Assembly-Language-and-Shellcoding-on-Linux#x86-assembly-language-and-shellcoding-on-linux