Topic 8
Lecture 8a
Advanced Procedures

CSCI 150

Assembly Language / Machine Architecture

Prof. Dominick Atanasio

Chapter Overview

- **Stack Frames**
- Recursion
- Creating Multimodule Programs
- Advanced Use of Parameters

Stack Frames

- Stack Parameters
- Local Variables
- ENTER and LEAVE Instructions

Stack Frame

- Also known as an *activation record*
- Area of the stack set aside for a procedure's return address, passed parameters, saved registers, and local variables
- Created by the following steps (this is new):
  - Calling program pushes arguments on the stack and calls the procedure.
  - The called procedure pushes EBP on the stack, and sets EBP to ESP.
    - Establishes the base of the stack frame.
  - If local variables are needed, a constant is subtracted from ESP to make room on the stack.4

Stack Parameters

- More convenient than register parameters
- Two possible ways of calling a procedure called DumpMem. Which is easier?

```
push esi
mov esi, array
mov ecx, arrayLen
mov ebx, byteQty
call DumpMem
pop esi
```

```
push array
push arrayLen
push byteQty
call DumpMem
```

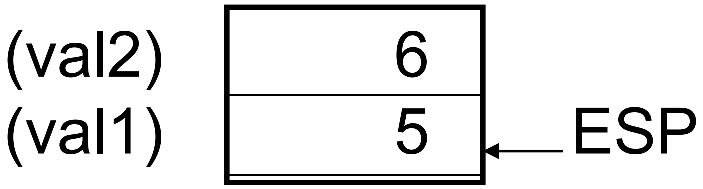
Passing Arguments by Value

- Push argument values on stack
  - (Use only 32-bit values to keep the stack “aligned”)
- Call the called-procedure
- Accept a return value in EAX, if any
- Remove arguments from the stack

Example (1 of 2)

```
section .data
val1  dw 5
val2  dw 6

section .text
push val2
push val1
```



Stack prior to CALL

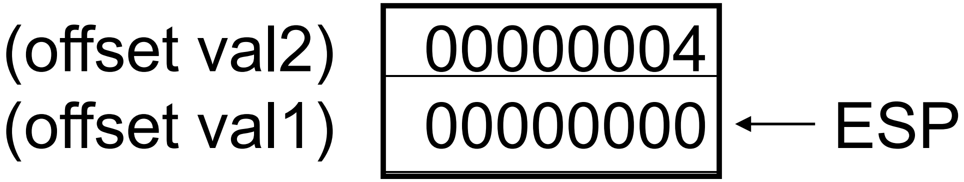
Passing by Reference

- Push the offsets (address) of arguments on the stack
- Call the procedure
- Accept a return value in EAX, if any
- Remove arguments from the stack

Example (2 of 2)

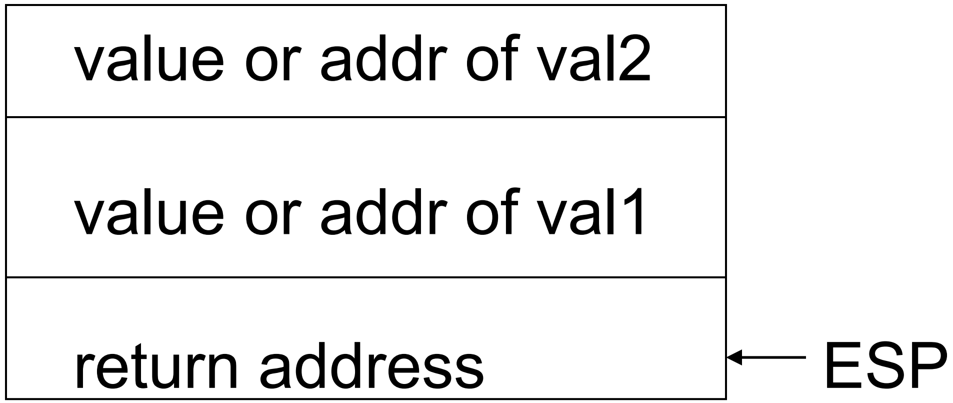
```
section .data
val1  dw 5
val2  dw 6

section .text
push val2
push val1
```



Stack prior to CALL

Stack after the CALL



value or addr of val2

value or addr of val1

return address

ESP

Passing an Array by Reference (1 of 2)

- The array_fill procedure fills an array with 16-bit random integers
- The calling program passes the address of the array (argument 1), along with a count of the number of array elements (argument 2):

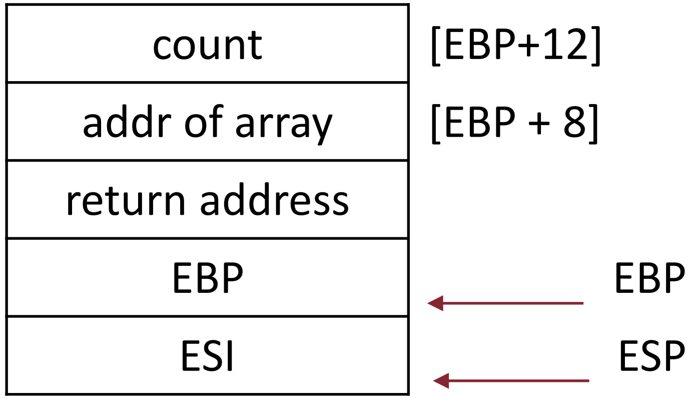
```
section .bss
        count: equ 100
        array: times count resw
section .text
        push DWORD count
        push array
        call array_fill
```

Passing an Array by Reference (2 of 2)

array_fill can reference an array without knowing the array's name:

```
array_fill:
      push ebp
      mov  ebp, esp

      push esi
      mov ecx, [ebp+12]
      mov esi,  [ebp+8]
      ...
      pop esi
      pop ebp
```



count [EBP+12]

addr of array [EBP + 8]

return address

EBP ← EBP

ESI ← ESP

ESI points to the beginning of the array, so it's easy to use a loop to access each array element.

Accessing Stack Parameters (C/C++)

- C and C++ functions access stack parameters using constant offsets from EBP.
  - Example: [ebp + 8]
- EBP is called the base pointer or frame pointer because it holds the base address of the base of the stack frame.
- EBP does not change value during the function.
- EBP must be restored to its original value when a function returns.

RET Instruction

- *Return from subroutine*
- Pops stack into the instruction pointer (EIP or IP). Control transfers to the target address.
- Syntax:
  - RET
  - RET *n*
- Optional operand *n* causes *n* bytes to be added to the stack pointer after EIP (or IP) is assigned a value.
  - This could be used by the callee to “pop” parameters

Who removes parameters from the stack?

Caller (C)

```
push val2
push val1
call add_two
add   esp, 8
```

...... or ......

Called-procedure

```
add_two:
push  ebp
mov   ebp, esp

mov   eax, [ebp+12]
add   eax, [ebp+8]

pop   ebp
ret   8
```

It should be the responsibility of the caller to remove the parameters.

Your turn . . .

- Create a procedure named d*ifference* that subtracts the first argument from the second one. Following is a sample call:

  pseudocode:

```
PROC difference(integer minuend, integer subtrahend)
      return minuend – subtrahend
ENDPROC difference
```

Passing 8-bit and 16-bit Arguments

- Cannot push 8-bit values on stack
- Pushing 16-bit operand may cause page fault or ESP alignment problem
- Expand smaller arguments into 32-bit values, using MOVZX or MOVSX:

Passing Multiword Arguments

- Push high-order values on the stack first; work backward in memory
- Results in little-endian ordering of data
- Example:

```
section .data
   longVal:  DQ 1234567800ABCDEFh
section .text
     push    DWORD [longVal + 4]          ; high doubleword
     push    DWORD [longVal]              ; low doubleword
     call    write_hex_64
```

Saving and Restoring Registers

- Push registers on stack just after assigning ESP to EBP
  - local registers are modified inside the procedure

```
my_sub:
        push    ebp
        mov     ebp, esp
        push    ecx                     ; save local registers
        push    edx
```

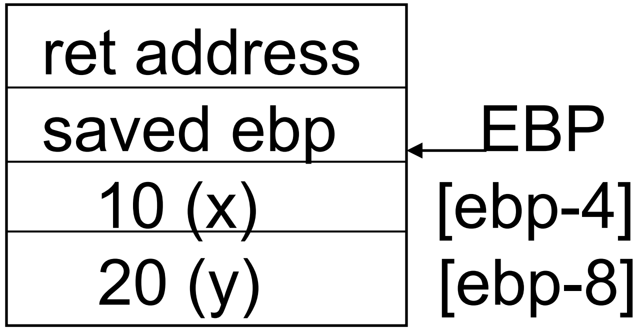
Local Variables

- Only statements within subroutine can view or modify local variables
- Storage used by local variables is released when subroutine ends
- local variable name can have the same name as a local variable in another function without creating a name clash
- Essential when writing recursive procedures, as well as procedures executed by multiple execution threads

Creating LOCAL Variables

Example - create two DWORD local variables:

Say: int x=10, y=20;



ret address

saved ebp ← EBP

10 (x) [ebp-4]

20 (y) [ebp-8]

```
MySub PROC

        push    ebp
        mov     ebp,esp
        sub     esp,8                ;create 2 DWORD variables

        mov     DWORD [ebp-4], 10 ; initialize x=10
        mov     DWORD [ebp-8], 20 ; initialize y=20
```

LEA Instruction

- Load Affective Address
- LEA returns addresses of direct and indirect operands
- LEA required when obtaining addresses of stack parameters & local variables
- Example

```
copy_string:
    push ebp
    mov ebp, esp
    sub esp, 4                      ; create 32bit local storage
    sub esp, 20                     ; create local array of 5 32bit elements
    ...
    mov edi, [ebp - 4]              ; invalid operand, copies the value stored in first
    mov esi, [ebp - 24]             ; invalid operand, copies first value stored
    lea edi, [ebp - 4]              ; ok, moves address of offset int edi
    lea esi, [ebp - 24]             ; ok, moves address of offset into esi
```

LEA Example (1 of 2)

Suppose you have a Local variable at [ebp-8]

And you need the address of that local variable in ESI

You cannot use this:

```
mov esi, [ebp-8]          ; error
```

Use this instead:

```
lea esi, [ebp-8]
```

ENTER Instruction

- ENTER instruction creates stack frame for a called procedure
  - pushes EBP on the stack
  - sets EBP to the base of the stack frame
  - reserves space for local variables
  - Example:

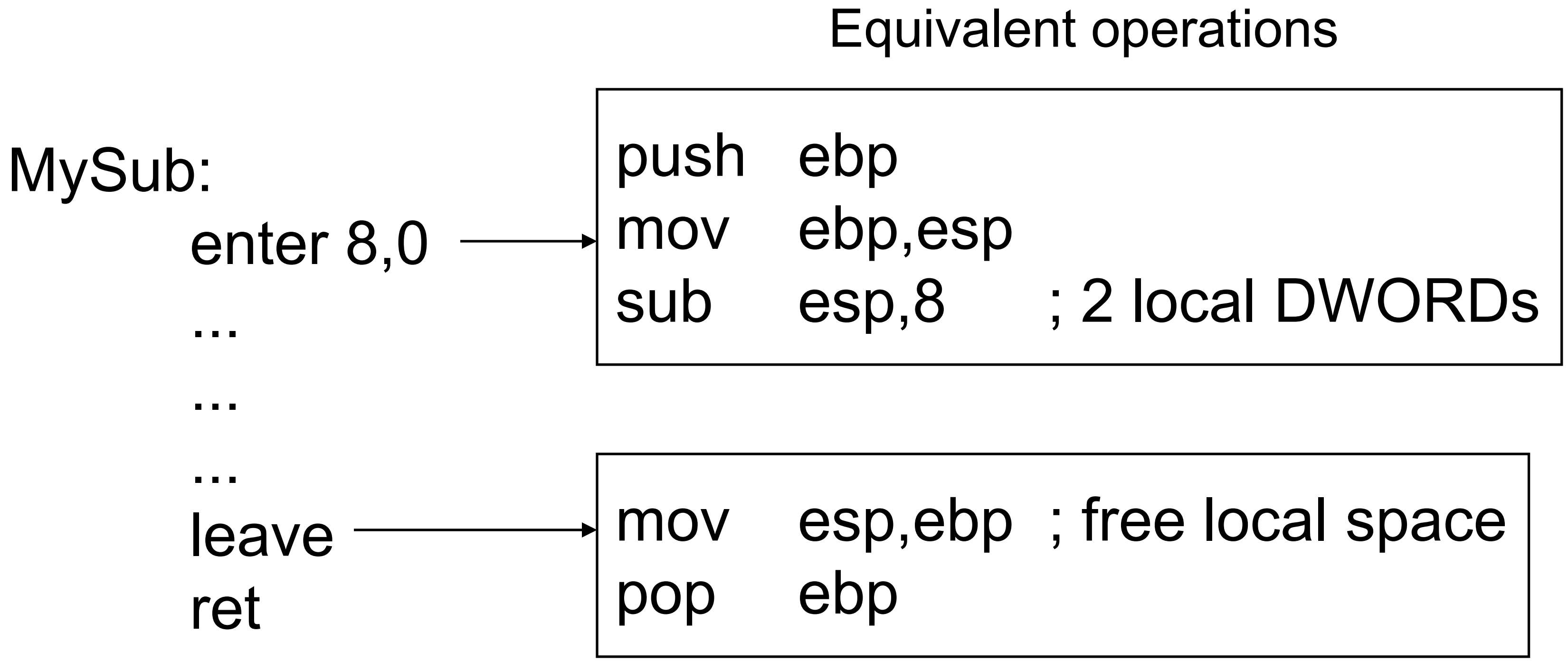
```
my_sub:
enter 8,0
```

  - Equivalent to:

```
my_sub:
  push ebp
  mov ebp,esp
  sub esp,8
```

LEAVE Instruction

Terminates the stack frame for a procedure.



Equivalent operations

```
MySub:
        enter 8,0
        ...
        ...
        ...
        leave
        ret
```

```
push   ebp
mov    ebp,esp
sub    esp,8      ; 2 local DWORDs
```

```
mov    esp,ebp  ; free local space
pop    ebp
```

What's Next (1 of 4)

- Stack Frames
- **Recursion**
- Creating Multimodule Programs

53 68 75 72 79 6F