Computer Organization and Assembly Language (COAL)

Lecture 7

Dr. Naveed Anwar Bhatti

Webpage: naveedanwarbhatti.github.io

Advanced Procedures

- **Stack Frames**
- Recursion
- INVOKE, ADDR, PROC, and PROTO
- Creating Multimodule Programs
- Advanced Use of Parameters (optional)
- Java Bytecodes (optional)

Stack Frames

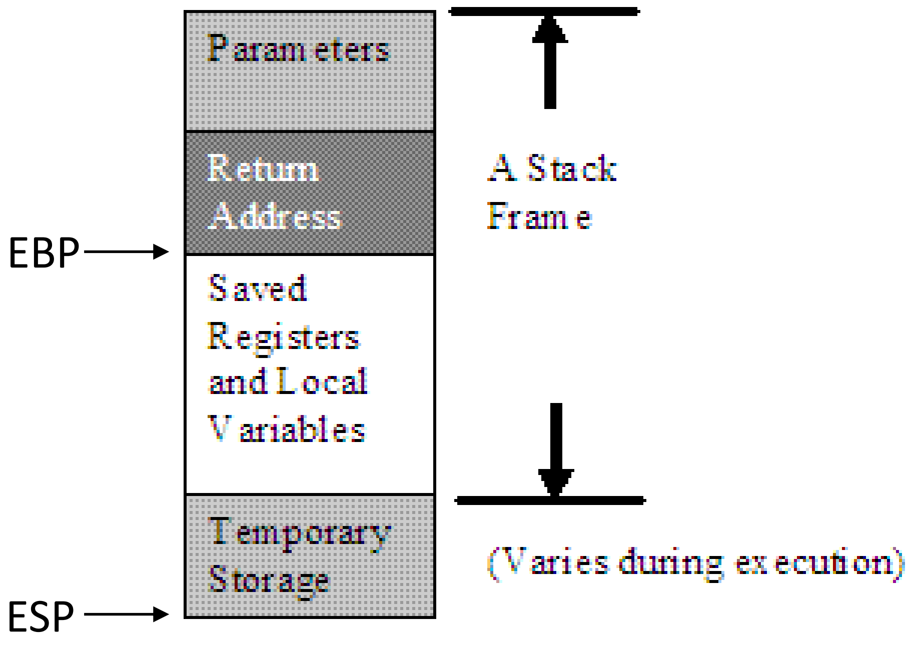
- Stack Parameters
- Local Variables
- ENTER and LEAVE Instructions
- LOCAL Directive

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:
  - Calling program pushes arguments on the stack and calls the procedure.
  - The called procedure pushes EBP on the stack, and sets EBP to ESP.
  - If local variables are needed, a constant is subtracted from ESP to make room on the stack.

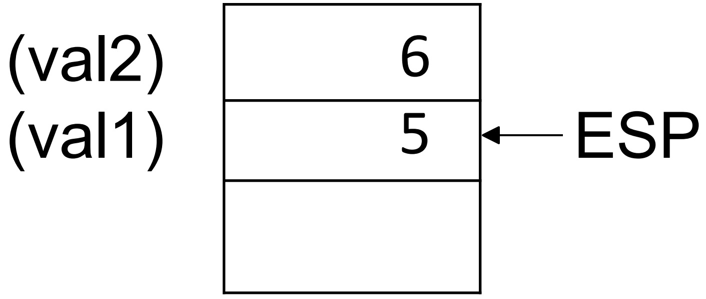

Parameters
Return Address
Saved Registers and Local Variables
Temporary Storage
EBP
ESP
A Stack Frame
(Varies during execution)

Passing Arguments by Value

- Push argument values on stack
  - (Use only **32-bit values** in protected mode to keep the stack aligned)
- Call the called-procedure
- Accept a return value in EAX, if any
- Remove arguments from the stack if the called- procedure did not remove them

Example

```
.data
val1   DWORD 5
val2   DWORD 6

.code
push val2
push val1
```



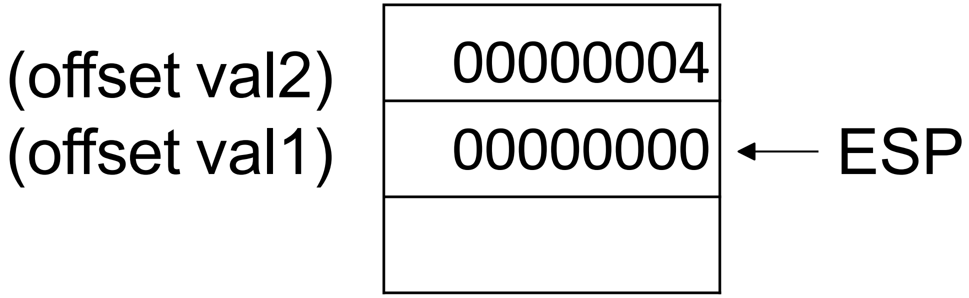
Stack prior to CALL

Passing by Reference

- Push the offsets of arguments on the stack
- Call the procedure
- Accept a return value in EAX, if any
- Remove arguments from the stack if the called procedure did not remove them

Example

```
.data
val1   DWORD 5
val2   DWORD 6

.code
push OFFSET val2
push OFFSET val1
```

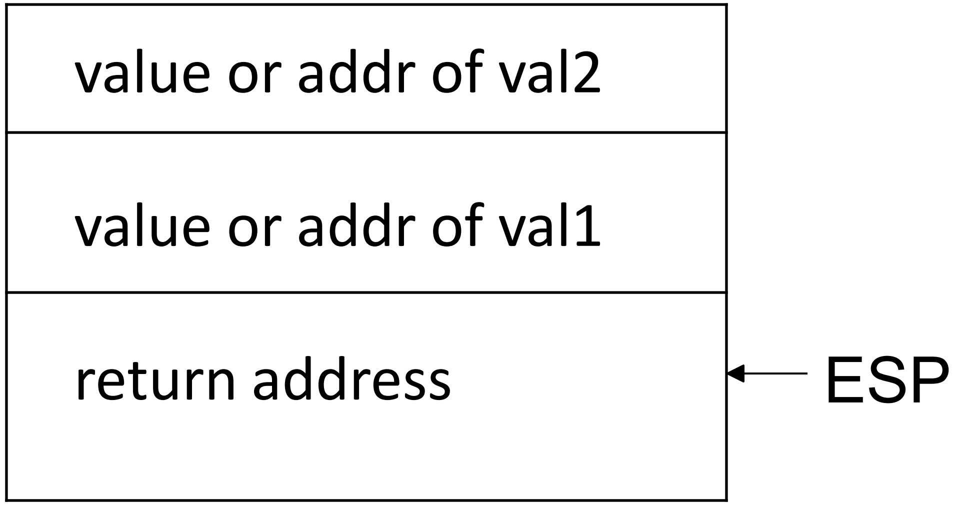


(offset val2) 00000004

(offset val1) 00000000 ← ESP

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)

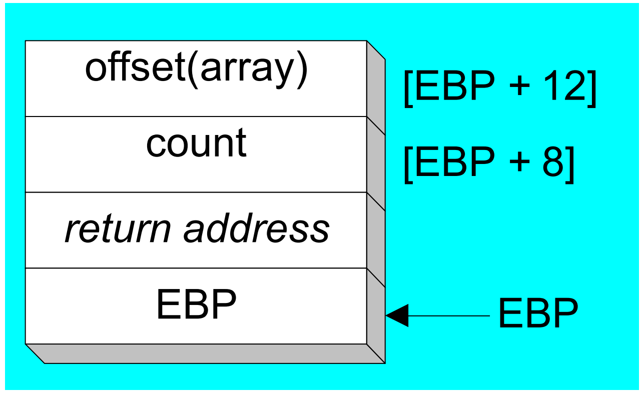
- The **ArrayFill** procedure fills an array with 16-bit random integers
- The calling program passes the address of the array, along with a count of the number of array elements:

```
.data
count = 100
array WORD count DUP(?)
.code
   push OFFSET array
   push COUNT
   call ArrayFill
```

Passing an Array by Reference (2 of 2)

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

```
ArrayFill PROC
    push ebp
    mov  ebp,esp
    pushad
    mov  esi,[ebp+12]
    mov  ecx,[ebp+8]
    .
    .
```



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[1].
  - Example: [ebp + 8]
- EBP is called the **base pointer** or **frame pointer** because it holds the base address of the stack frame.
- EBP does not change value during the function.
- EBP must be restored to its original value when a function returns.

[1] BP in Real-address mode

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.

Who removes parameters from the stack?

Caller (C) ...... or ...... Called-procedure (STDCALL):

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

```
AddTwo PROC

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

    pop   ebp
    ret   8
```

(Covered later: The MODEL directive specifies calling conventions)

Your turn . . .

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

```
push 14                         ; first argument
push 30                         ; second argument
call Difference                 ; EAX = 16
```

```
Difference PROC
    push ebp
    mov  ebp,esp
    mov  eax,[ebp + 8]       ; second argument
    sub  eax,[ebp + 12]      ; first argument
    pop  ebp
    ret  8
Difference ENDP
```

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
  - incompatible with Windows API functions
- Expand smaller arguments into 32-bit values, using MOVZX or MOVSX:

```
.data
charVal BYTE 'x'
.code
  movzx eax,charVal
  push  eax
  call  Uppercase
```

Passing Multiword Arguments

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

```
.data
longVal DQ 1234567800ABCDEFh
.code
  push  DWORD PTR longVal + 4       ; high doubleword
  push  DWORD PTR longVal           ; low doubleword
  call  WriteHex64
```

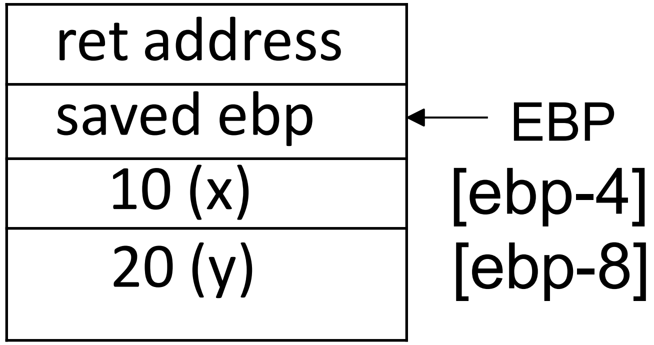
Local Variables

- **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

10 (x)

20 (y)

EBP

[ebp-4]

[ebp-8]

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

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

LEA Instruction

- LEA returns offsets of direct and indirect operands
    - OFFSET operator only returns constant offsets
- LEA required when obtaining offsets of stack parameters & local variables
- Example

```
CopyString PROC,
    LOCAL temp[20]:BYTE, count:DWORD


    mov edi,OFFSET count       ; invalid operand
    mov esi,OFFSET temp        ; invalid operand
    lea edi,count              ; ok
    lea esi,temp               ; ok
```

LEA Example

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, OFFSET [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:

```
MySub PROC
   enter 8,0
```

  - Equivalent to:

```
MySub PROC
   push ebp
   mov ebp,esp
   sub esp,8
```

LEAVE Instruction

Terminates the stack frame for a procedure.

```
MySub PROC
          enter 8,0
          ...
          ...
          ...
          leave
          ret
MySub ENDP
```

Equivalent operations

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

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

LOCAL Directive

- The LOCAL directive declares a list of local variables
  - immediately follows the PROC directive
  - each variable is assigned a type
- Syntax:

```
LOCAL varlist
```

Example:

```
MySub PROC
   LOCAL var1:BYTE, var2:WORD, var3:SDWORD
```

LOCAL Example (1 of 2)

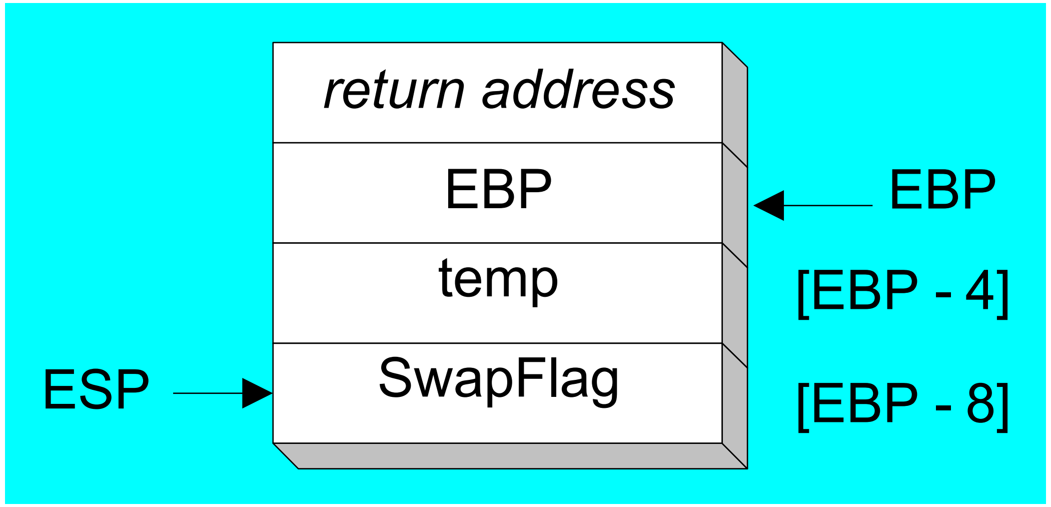
```
BubbleSort PROC
    LOCAL temp:DWORD, SwapFlag:BYTE
    . . .
    ret
BubbleSort ENDP
```

MASM generates the following code:

```
BubbleSort PROC
    push ebp
    mov  ebp,esp
    sub  esp,8
    . . .
    mov  esp,ebp
    pop  ebp
    ret
BubbleSort ENDP
```

LOCAL Example (2 of 2)

Diagram of the stack frame for the ***BubbleSort*** procedure:



What's Next

- Stack Frames

Recursion

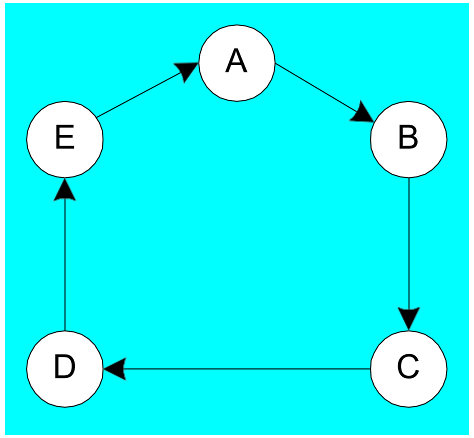
- INVOKE, ADDR, PROC, and PROTO
- Creating Multimodule Programs
- Advanced Use of Parameters (optional)
- Java Bytecodes (optional)

Recursion

- What is Recursion?
- Recursively Calculating a Sum
- Calculating a Factorial

What is Recursion?

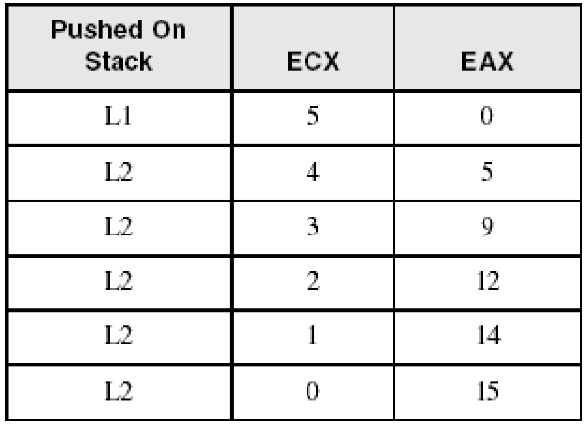
- The process created when . . .
  - A procedure calls itself
  - Procedure A calls procedure B, which in turn calls procedure A
- Using a graph in which each node is a procedure and each edge is a procedure call, recursion forms a **cycle**:



Recursively Calculating a Sum

The CalcSum procedure recursively calculates the sum of an array of integers. Receives: ECX = count. Returns: EAX = sum

```
    CalcSum PROC

    cmp ecx,0                   ; check counter value
    jz L2                       ; quit if zero
    add eax,ecx                 ; otherwise, add to sum
    dec ecx                     ; decrement counter
    call CalcSum                ; recursive call
L2: ret
CalcSum ENDP
```



Pushed On Stack	ECX	EAX
L1	5	0
L2	4	5
L2	3	9
L2	2	12
L2	1	14
L2	0	15

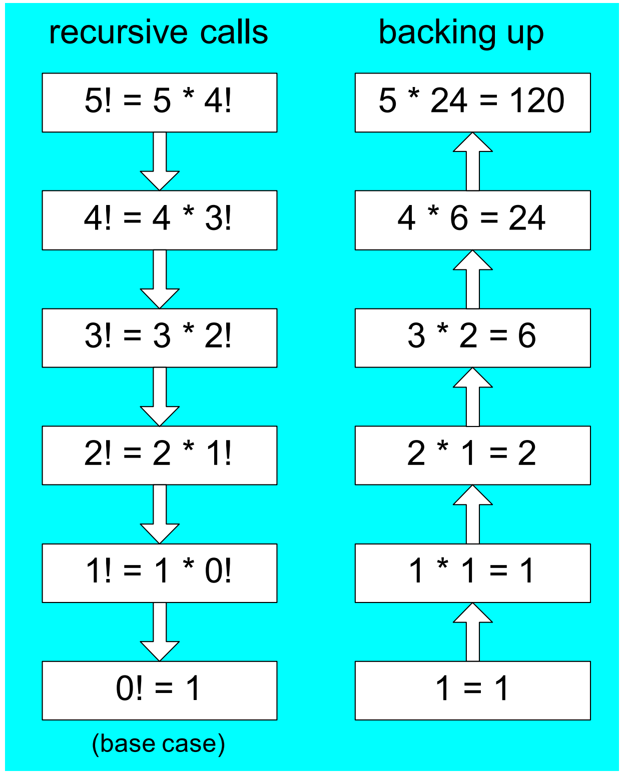
View the complete program

Calculating a Factorial (1 of 3)

This function calculates the factorial of integer *n*. A new value of *n* is saved in each stack frame:

```
int function factorial(int n)
{
    if(n == 0)
      return 1;
    else
      return n * factorial(n-1);
}
```

As each call instance returns, the product it returns is multiplied by the previous value of n.



recursive calls

5! = 5 * 4!

4! = 4 * 3!

3! = 3 * 2!

2! = 2 * 1!

1! = 1 * 0!

0! = 1

(base case)

backing up

5 * 24 = 120

4 * 6 = 24

3 * 2 = 6

2 * 1 = 2

1 * 1 = 1

1 = 1

Calculating a Factorial (2 of 3)

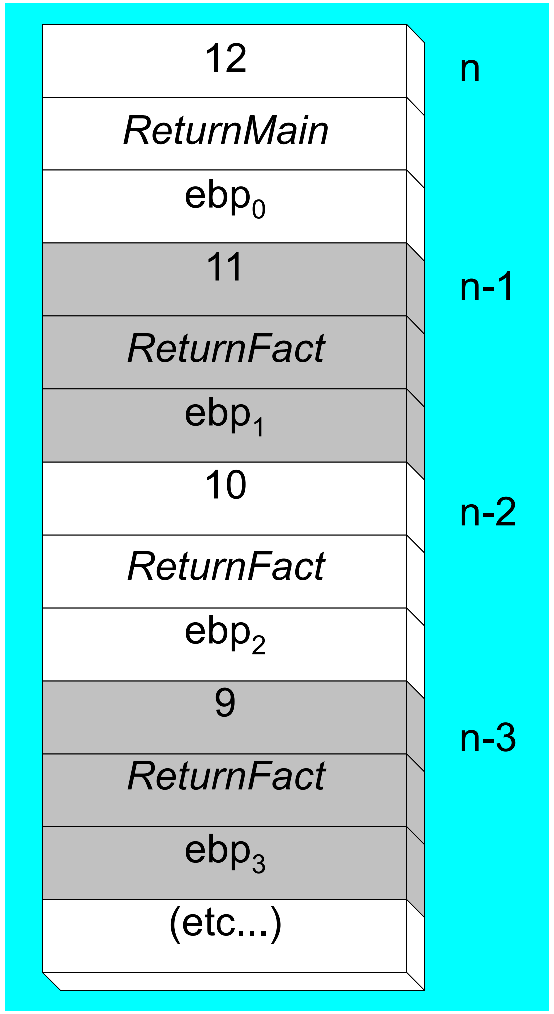
```
Factorial PROC
    push ebp
    mov  ebp,esp
    mov  eax,[ebp+8]              ; get n
    cmp  eax,0                    ; n < 0?
    ja   L1                       ; yes: continue
    mov  eax,1                    ; no: return 1
    jmp  L2

L1: dec  eax
    push eax                      ; Factorial(n-1)
    call Factorial

; Instructions from this point on execute when each
; recursive call returns.

ReturnFact:
    mov  ebx,[ebp+8]              ; get n
    mul  ebx                      ; eax = eax * ebx

L2: pop  ebp                      ; return EAX
    ret  4                        ; clean up stack
Factorial ENDP
```

Calculating a Factorial (3 of 3)

Suppose we want to calculate 12!

This diagram shows the first few stack frames created by recursive calls to Factorial

Each recursive call uses 12 bytes of stack space.



12

ReturnMain

ebp$_0$

11

ReturnFact

ebp$_1$

10

ReturnFact

ebp$_2$

9

ReturnFact

ebp$_3$

(etc...)