# X86 Calling Convention

Spring, 2018

# x86 Data Declarations

- Must be in a data section
- Uninitialized data usually with .comm

```
.comm    Speed,4,4
.comm    Velocity,8,8
.comm    Vector,40,32
```

- First arg is name, second is size (in bytes), optional 3rd is alignment
  - Alignment must be power of 2, an alignment of 8 says double word aligned (i.e., low 3 bits of address is zero)

# x86 Data Declarations

- Must be in a data section
- Initialized

```
            .globl  Time
            .data
            .align  4
            .size   Time, 4
Time:
            .long   60
             .text
```

# x86 Memory Operations

- "lea" instruction means "load effective address:

```
  leal  [count],eax  ; eax := address of
count
```

- Can move through an address pointer

```
  lea ebx,[count] ; ebx := address of count
  mov edx,[ebx]   ; count := edx
                  ; ebx is a pointer
                  ; [ebx] dereferences it
```

- We also will see the stack used as memory

# x86 Stack Operations

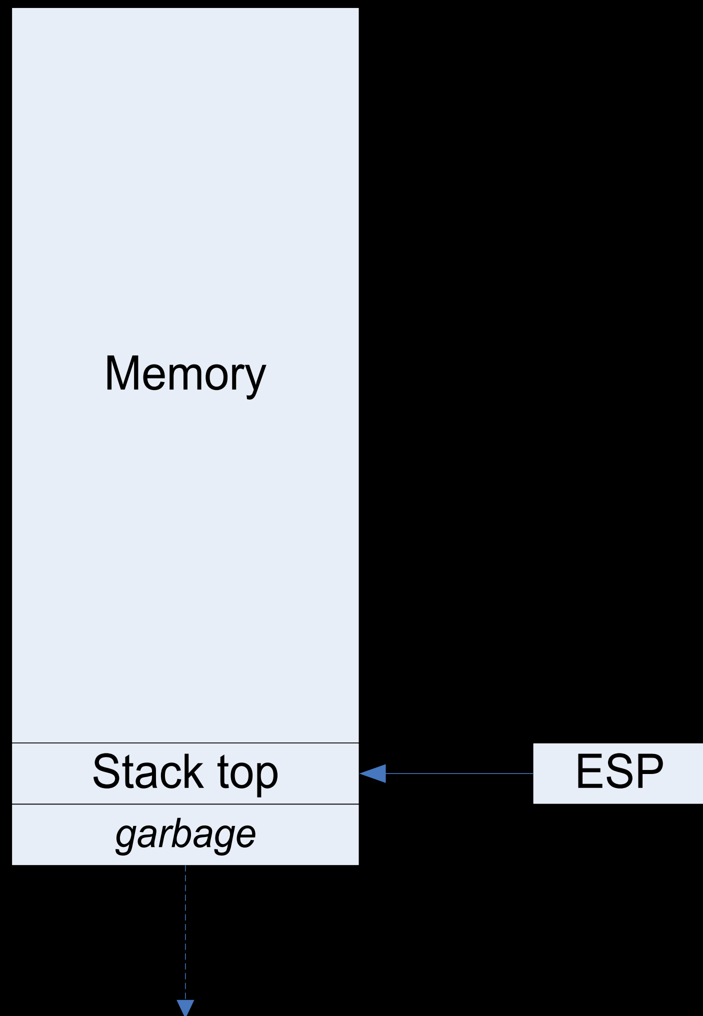
- The x86 stack is managed using the ESP (stack pointer) register, and specific stack instructions:

```
1. push ecx   ; push ecx onto stack
2. pop ebx    ; pop top of stack into register ebx
3. call foo   ; push address of next instruction on
               ; stack, then jump to label foo
4. ret        ; pop return address off stack, then
               ; jump to it
```

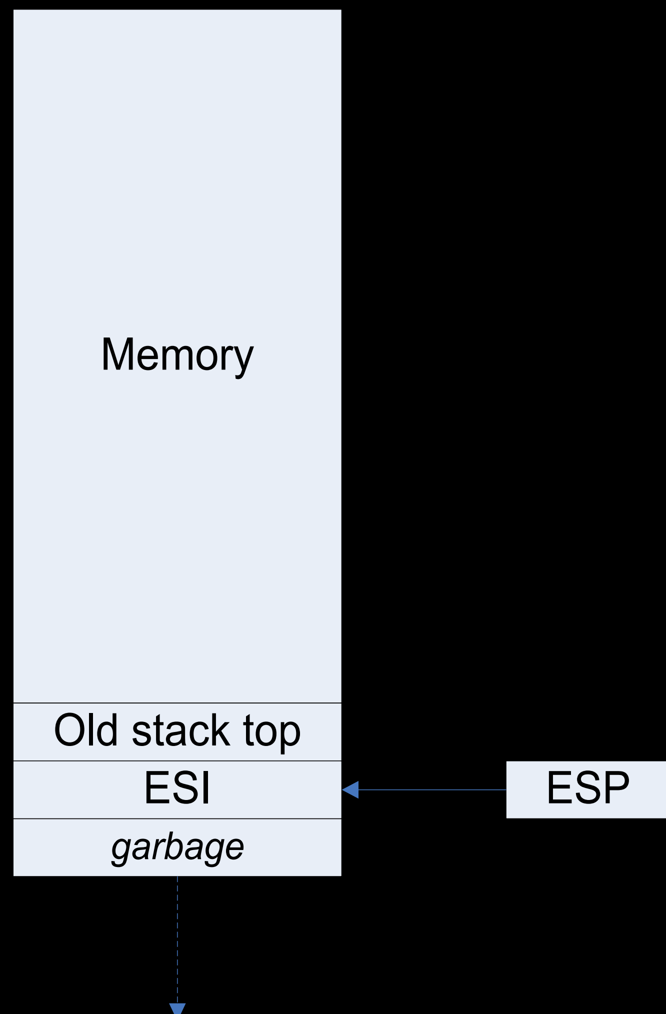
# x86 Hardware Stack

- The x86 stack grows downward in memory addresses
- Decrementing ESP increases stack size; incrementing ESP reduces it

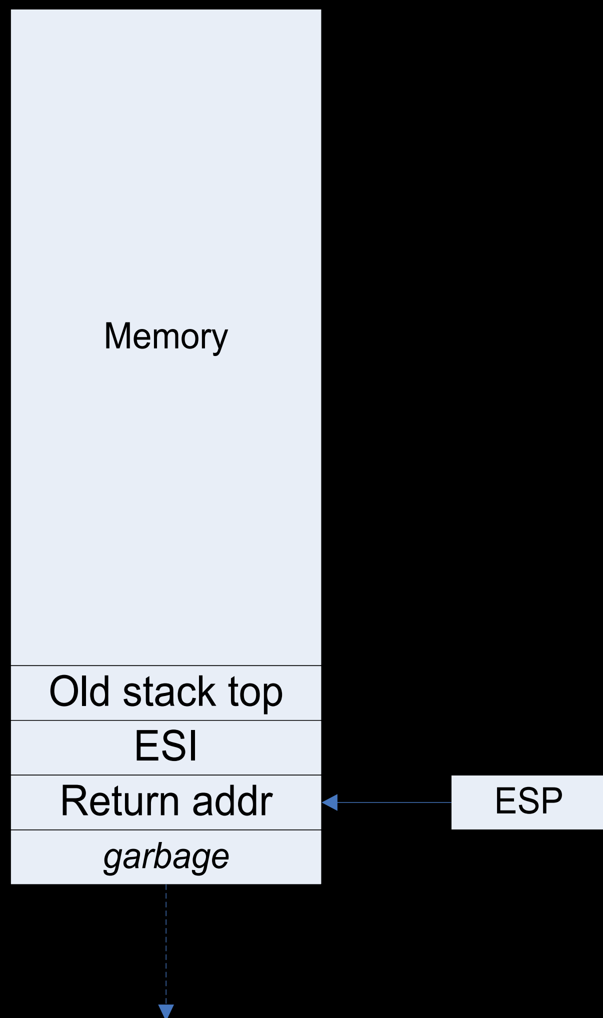
# x86 Hardware Stack



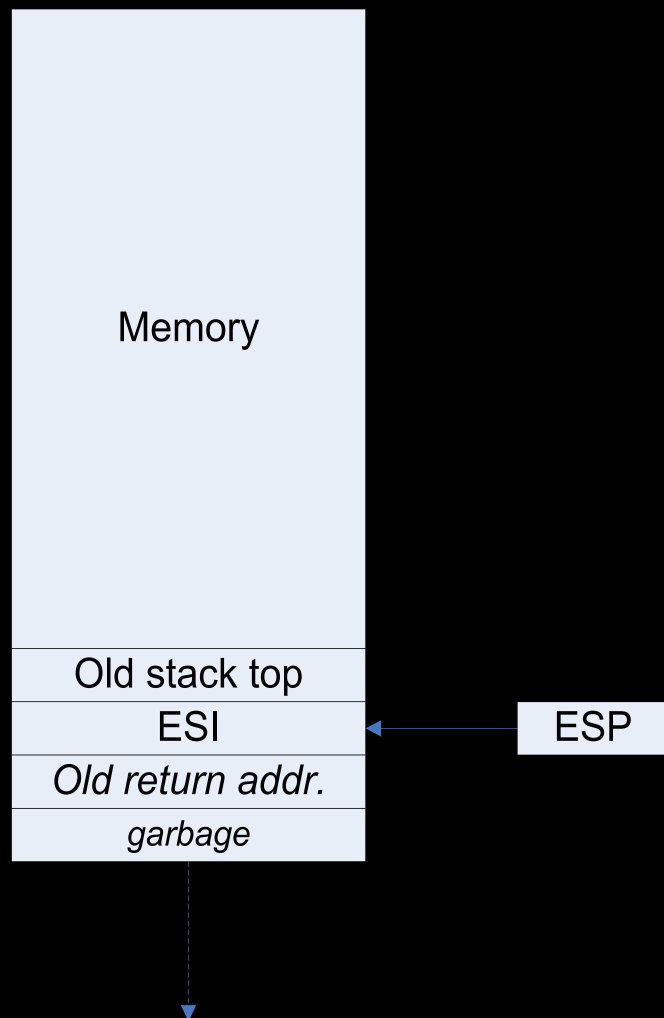
# x86 Stack after "push ESI"



# x86 Stack after `call`



# x86 Stack after `ret`



# x86 C Calling Convention

- A calling convention is an *agreement* among software designers (e.g. of compilers, compiler libraries, assembly language programmers) on how to use registers and memory in subroutines
- NOT enforced by hardware
- Allows software pieces to interact compatibly, e.g. a C function can call an ASM function, and vice versa

# C Calling Convention cont.

- Questions answered by a calling convention:
  1. How are parameters passed?
  2. How are values returned?
  3. Where are local variables stored?
  4. Which registers must the caller save before a call, and which registers must the callee save if it uses them?

# How Are Parameters Passed?

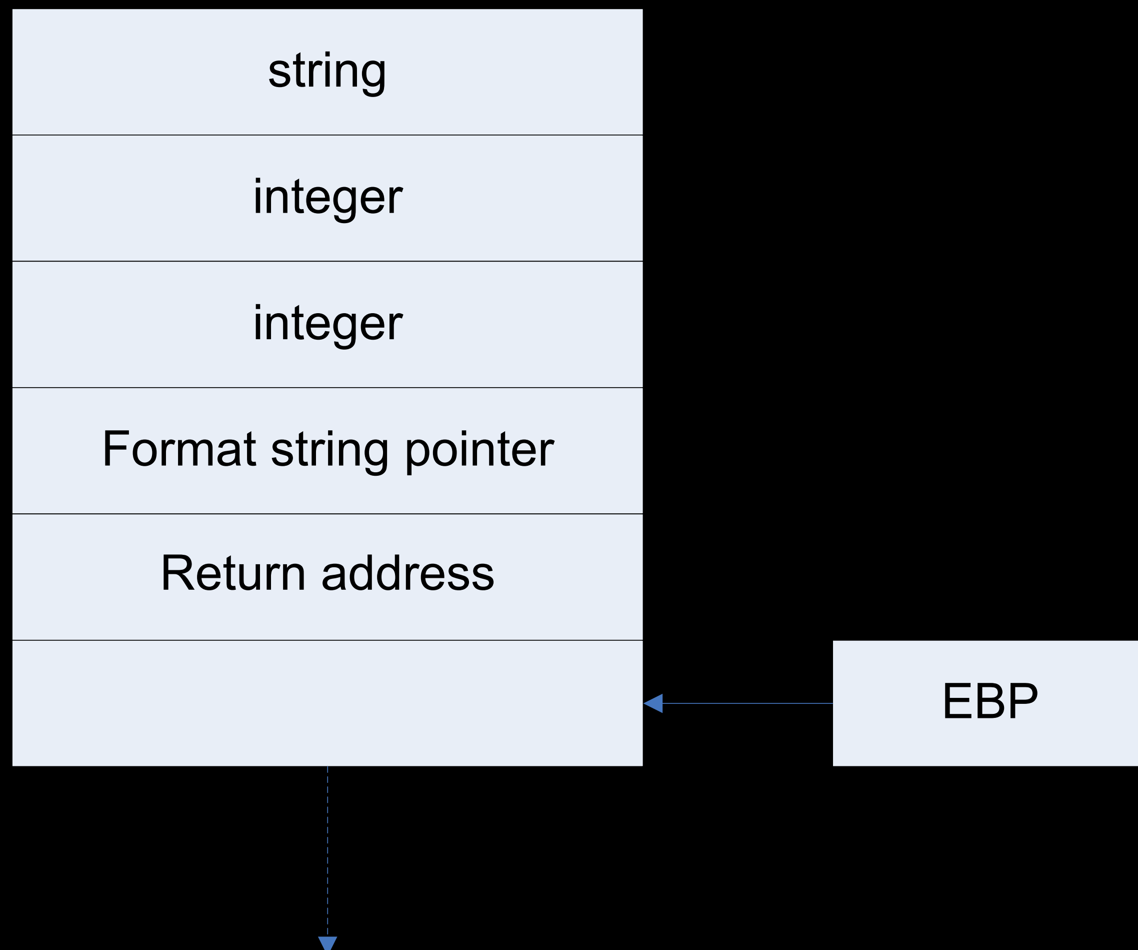
- Most machines use registers, because they are faster than memory
- x86 has too few registers to do this
- Therefore, the stack must be used to pass parameters
- Parameters are pushed onto the stack in reverse order

# Why Pass Parameters in Reverse Order?

- Some C functions have a variable number of parameters
- First parameter determines the number of remaining parameters
- Example: `printf("%d %d %s\n", ….);`
- `Printf()` library function reads first parameter, then determines that the number of remaining parameters is 3

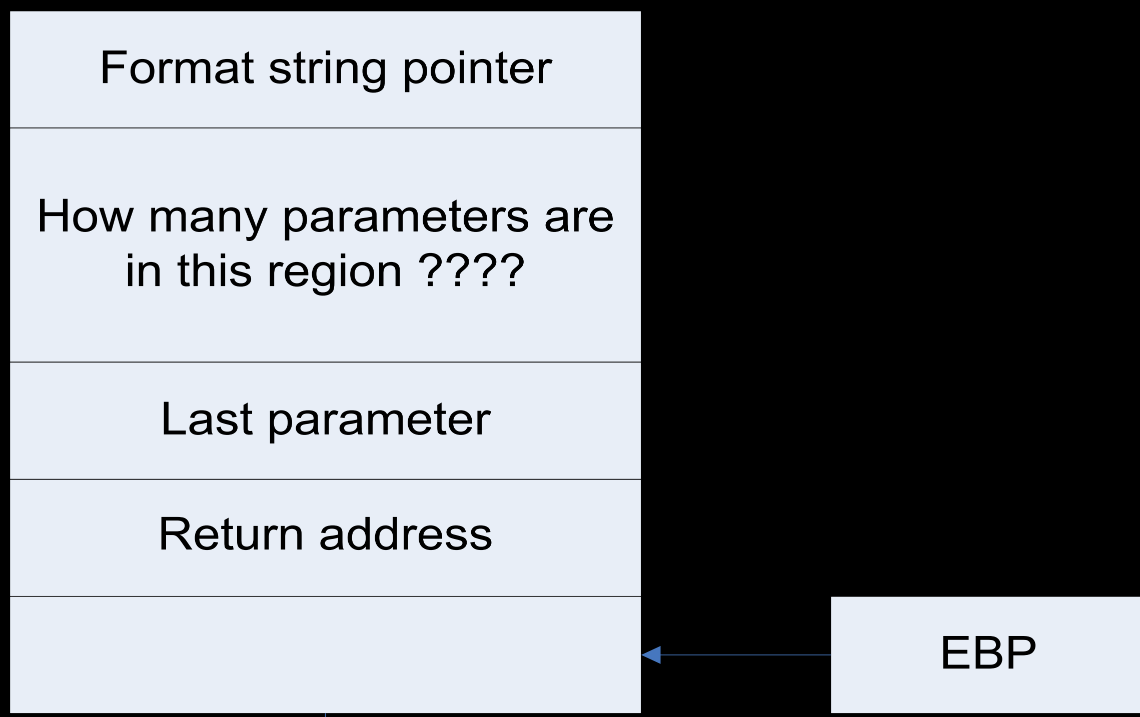
# Reverse Order Parameters cont.

- `printf()` will always find the first parameter at [EBP + 8]



# What if Parameter Order was NOT Reversed?

- `printf()` will always find the LAST parameter at `[EBP + 8]`; not helpful



# C Calling Convention cont.

- Questions answered by a calling convention:
  1. How are parameters passed?
  2. How are values returned?
  3. Where are local variables stored?
  4. Which registers must the caller save before a call, and which registers must the callee save if it uses them?

# How are Values Returned?

- Register eax contains the return value for integer values
- This means x86 can only return a 32-bit value from a function
- Smaller values are zero extended or sign extended to fill register eax
- If a programming language permits return of larger values (structures, objects, arrays, etc.), a pointer to the object is returned in register eax

# How are Values Returned?

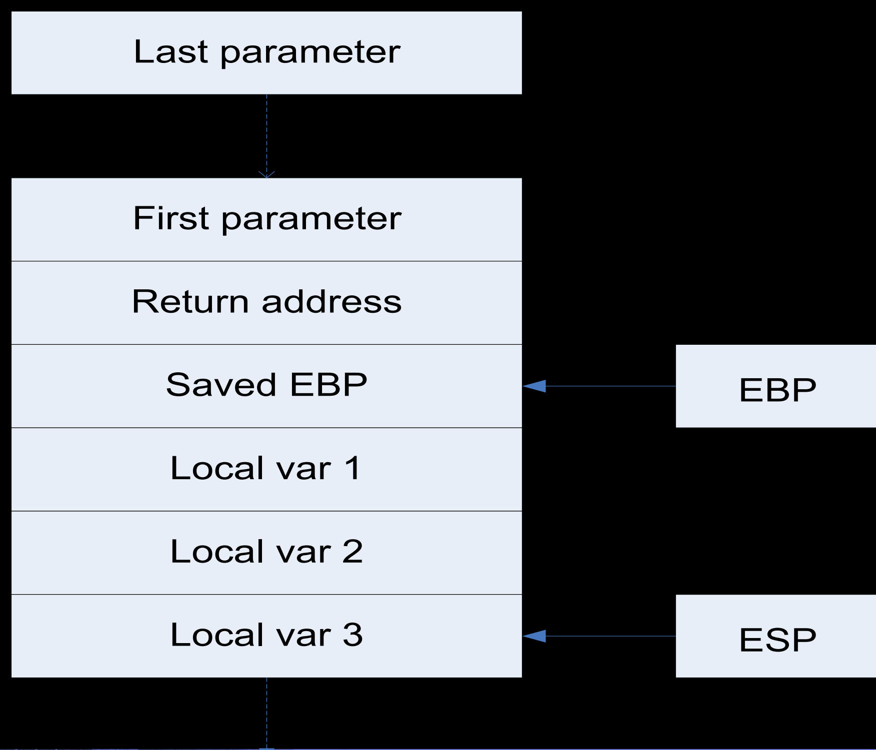
- Floating-point values (e.g., double and float) are returned on top of the floating-point stack

# C Calling Convention cont.

- Questions answered by a calling convention:
  1. How are parameters passed?
  2. How are values returned?
  3. Where are local variables stored?
  4. Which registers must the caller save before a call, and which registers must the callee save if it uses them?

# Where are Local Variables Stored?

- Stack frame for the currently executing function is between where EBP and ESP point in the stack



# C Calling Convention cont.

- Questions answered by a calling convention:
  1. How are parameters passed?
  2. How are values returned?
  3. Where are local variables stored?
  4. Which registers must the caller save before a call, and which registers must the callee save if it uses them?

# Who Saves Which Registers?

- It is efficient to have the caller save some registers before the call, leaving others for the callee to save
- x86 only has 8 general registers; 2 are used for the stack frame (ESP and EBP)
- The other 6 are split between callee-saved (ESI, EDI) and caller-saved
- Remember: Just a *convention*, or agreement, among software designers

# What Does the Caller Do?

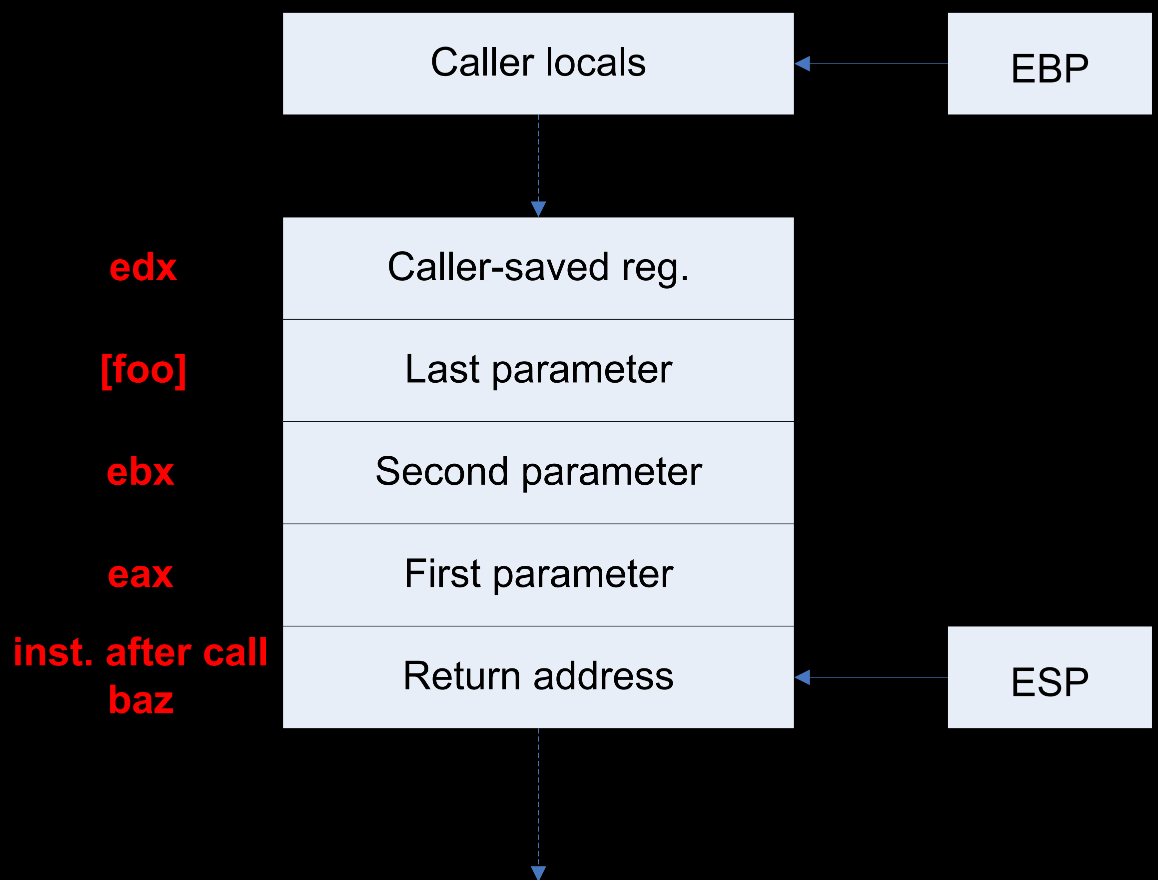
- Example: Call a function and pass 3 integer parameters to it

```
push edx      ; caller-saved register
push [foo]    ; Var foo is last parameter
push ebx      ; ebx is second parameter
push eax      ; eax is first parameter
call baz      ; push return address, jump
add esp,12    ; toss old parameters
pop edx       ; restore caller-saved edx
              ; eax holds return value
```

- eax, ebx did not need to be saved here

# Stack after Call

- x86 stack immediately after `call baz`



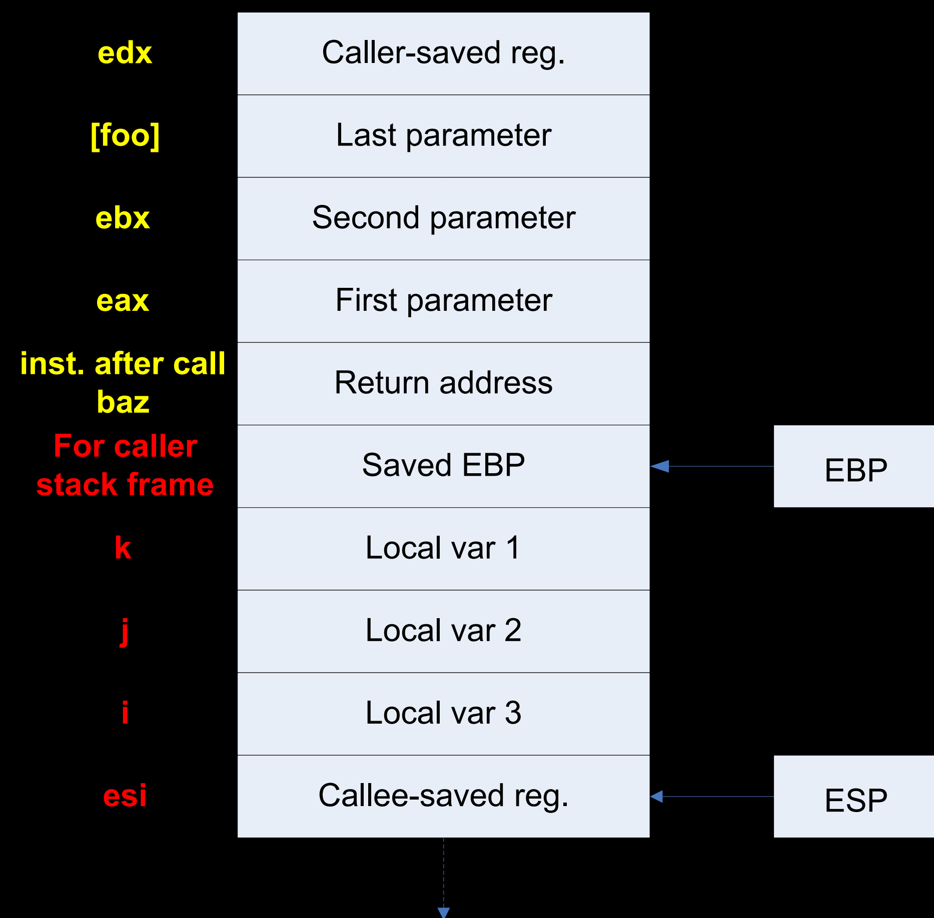
# Callee Stack Frame Setup

- The standard subroutine prologue code sets up the new stack frame:

```
; Prologue code at top of function
push ebp          ; save old base pointer
move ebp,esp      ; Set new base pointer
sub esp,12        ; Make room for locals
push esi          ; Func uses ESI, so save
:
:
```

# Stack After Prologue Code

- After the prologue code sets up the new stack frame:



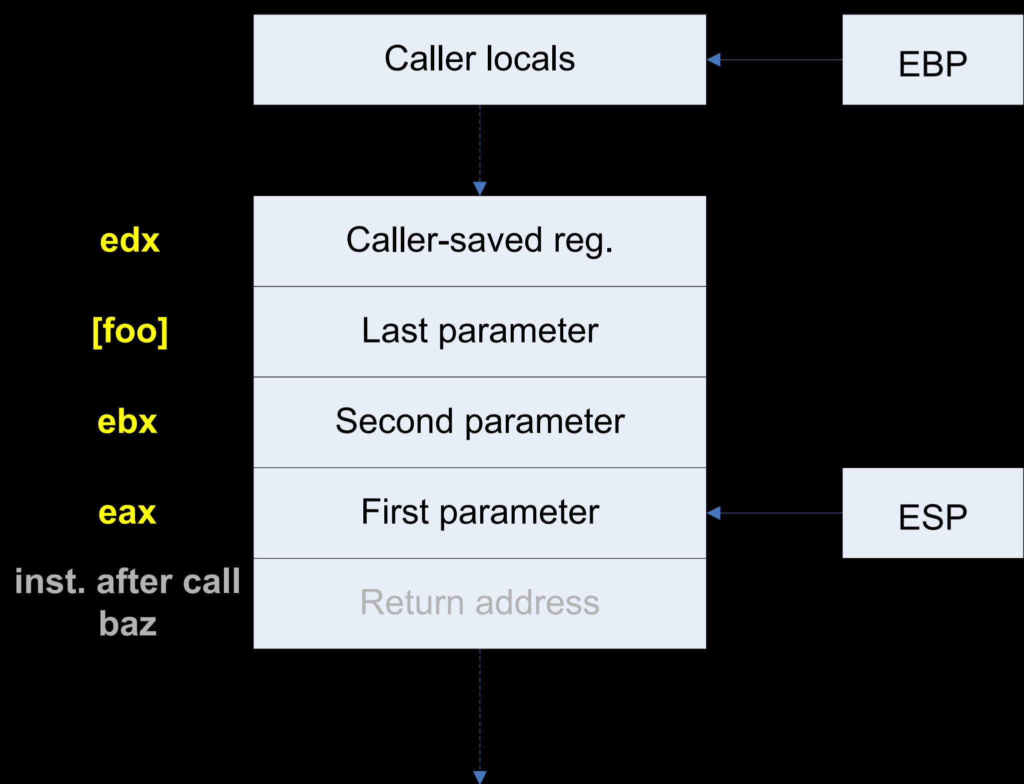
# Callee Stack Frame Cleanup

- Epilogue code at end cleans up frame (mirror image of prologue):

```
; Epilogue code at bottom of function
pop esi            ; Restore callee-saved ESI
move esp,ebp       ; Deallocate stack frame
pop ebp            ; Restore caller's EBP
ret                ; return
```

# Stack After Return

- After epilogue code and return:



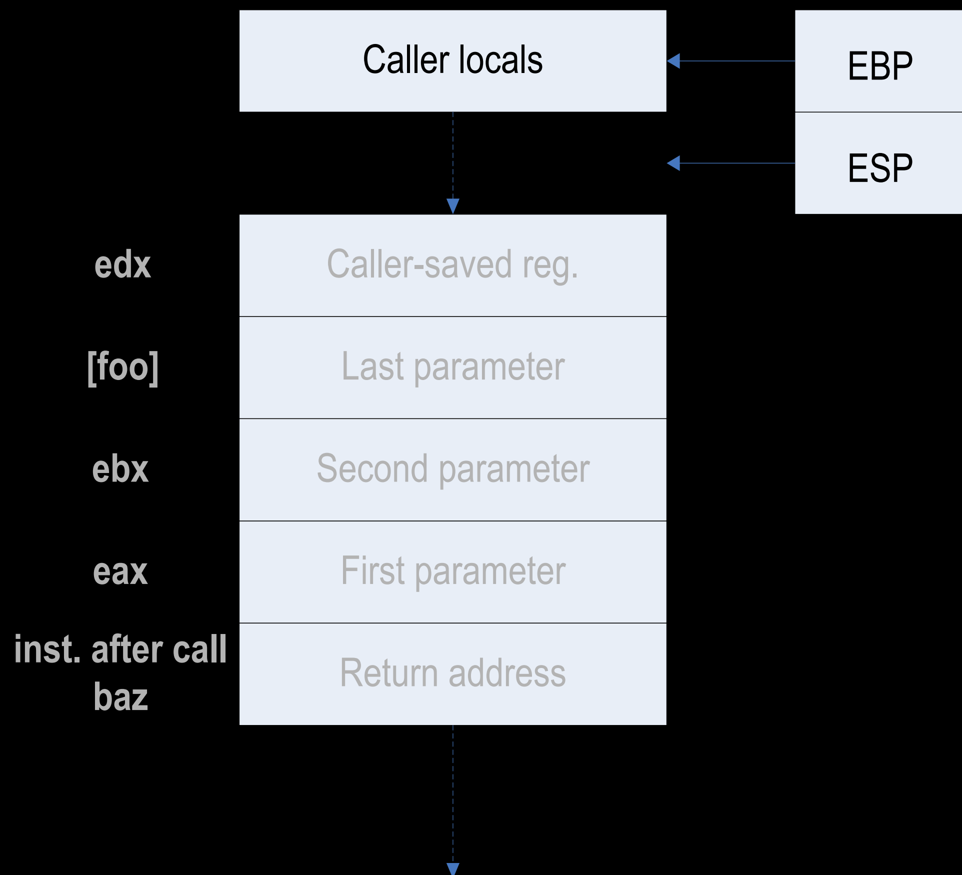
# Caller Stack Cleanup

- After the return, caller has a little cleanup code:

```
add esp,12    ; deallocate parameter space
pop edx       ; restore caller-saved register
```

# Caller Stack After Cleanup

- After the caller's cleanup code:



# Register Save Question

- Why would it be less efficient to have all registers be callee-saved, rather than splitting the registers into caller-saved and callee-saved? (Just think of one example of inefficiency.)

# Who Saves Which Register

| Register | Who Saves | How to Save |
|---|---|---|
| EAX, ECX, EDX | Caller | Considered volatile, caller must save; manually push to stack, e.g., push %eax |
| EBX, EDI, ESI | Callee | Considered non-volatile, callee saves if used; Manually push to stack; e.g., push %edi |
| EBP | Callee | Stack frame pointer; caller stack status; Manually push to stack; e.g., push %ebp |
| ESP | Callee | Stack Pointer; Current top of the stack; Must reset to point to return address before “ret” |
| EIP | CPU | Automatically push to stack; “call” pushes EIP, while “ret” pops EIP |

# Call Stack: Malware Implications

- The return address is a primary target of malware
- If the malware can change the return address on the stack, it can cause control to pass to malware code
- We saw an example with buffer overflows
- Read “Tricky Jump” document on web page for another virus technique

# x86 Executable Files

- On Linux, the standard format of a *.exe file, produced by compiling and linking, is called ELF (Extensible and Linking Format)
- On Windows, the standard format for an executable is called PE32 (Portable Executable) newer format is PE64
- Older formats exist for 16-bit DOS and Windows 3.1
- We will always be talking about ELF