Assembly Language:
Function Calls

Goals of this Lecture

Help you learn:

- Function call problems
- IA-32 solutions
    - Pertinent instructions and conventions

Function Call Problems

Calling and returning

- How does caller function **jump** to callee function?
- How does callee function **jump back** to the right place in caller function?

Passing arguments

- How does caller function pass **arguments** to callee function?

Storing local variables

- Where does callee function store its **local variables**?

Returning a value

- How does callee function send **return value** back to caller function?

Handling registers

- How do caller and callee functions use same **registers** without interference?

Running Example

Caller

```
void f(void)
{   …
    n = add3(3, 4, 5);
    …
}
```

Callee

```
int add3(int a, int b, int c)
{   int d;
    d = a + b + c;
    return d;
}
```

Agenda

Calling and returning

Passing arguments

Storing local variables

Returning a value

Handling registers

An example

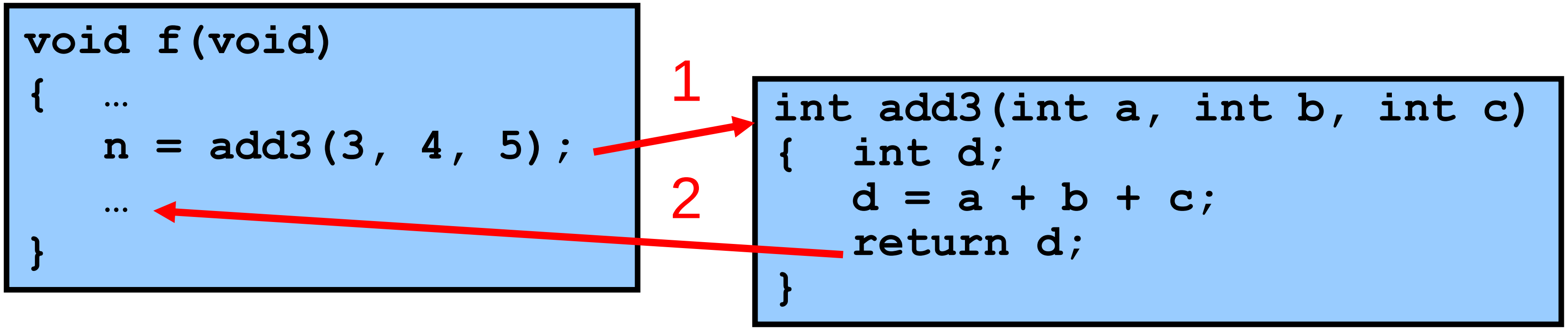
Problem 1: Calling and Returning

How does caller *jump* to callee?

- I.e., Jump to the address of the callee’s first instruction

How does the callee *jump back* to the right place in caller?

- I.e., Jump to the instruction immediately following the most-recently-executed call instruction



```
void f(void)
{   …
    n = add3(3, 4, 5);
    …
}
```

```
int add3(int a, int b, int c)
{   int d;
    d = a + b + c;
    return d;
}
```

Attempted Solution: `jmp` Instruction

Attempted solution: caller and callee use `jmp` instruction

```
f:
   …
   jmp g        # Call g
fReturnPoint:
   …
```

```
g:
   …
   jmp fReturnPoint   # Return
```

Attempted Solution: `jmp` Instruction

Problem: callee may be called by multiple callers

```
f1:
   …
   jmp g       # Call g
f1ReturnPoint:
   …
```

```
f2:
   …
   jmp g       # Call g
f2ReturnPoint:
   …
```

```
g:
   …
   jmp ???   # Return
```

Attempted Solution: Use Register

Attempted solution: Store return address in register

```
f1:
   movl $f1ReturnPoint, %eax
   jmp g      # Call g
f1ReturnPoint:
   …
```

```
f2:
   movl $f2ReturnPoint, %eax
   jmp g      # Call g
f2ReturnPoint:
   …
```

```
g:
   …
   jmp *%eax  # Return
```

Special form of
`jmp` instruction

Attempted Solution: Use Register

Problem: Cannot handle nested function calls

```
f:
   movl $fReturnPoint, %eax
   jmp g       # Call g
fReturnPoint:
   …
```

```
g:
   movl $gReturnPoint, %eax
   jmp h       # Call h
gReturnPoint:
   …
   jmp *%eax   # Return
```

Problem if f() calls g(), and g() calls h()

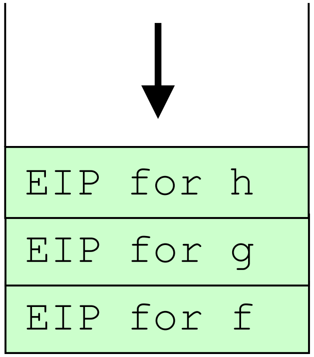
Return address g() -> f() is lost

```
h:
   …
   jmp *%eax   # Return
```

IA-32 Solution: Use the Stack

Observations:

- May need to store many return addresses
  - The number of nested function calls is not known in advance
  - A return address must be saved for as long as the invocation of this function is live, and discarded thereafter
- Stored return addresses are destroyed in reverse order of creation
  - f() calls g() => return addr for g is stored
  - g() calls h() => return addr for h is stored
  - h() returns to g() => return addr for h is destroyed
  - g() returns to f() => return addr for g is destroyed
- LIFO data structure (stack) is appropriate

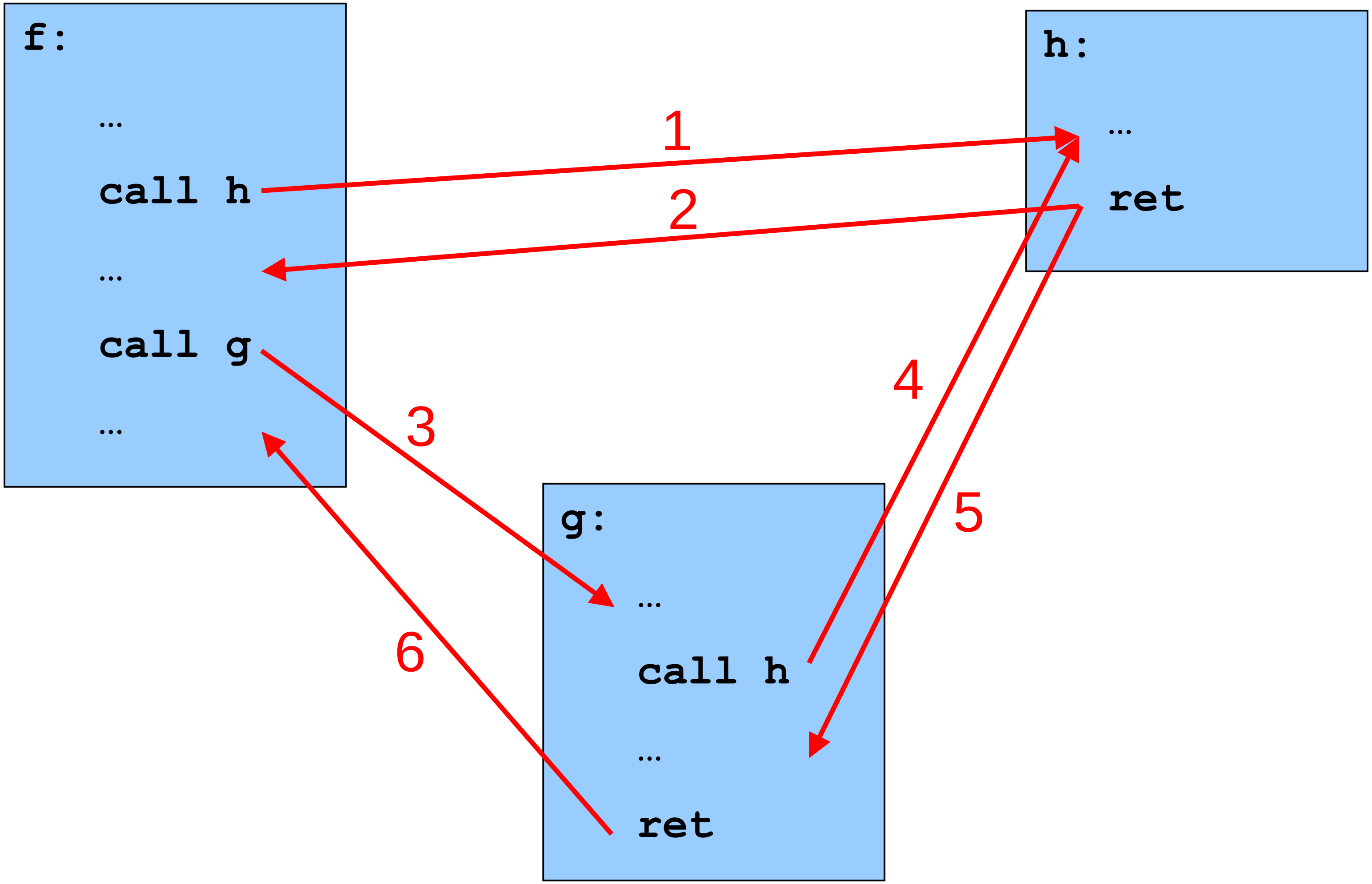


IA-32 solution:

- Use the STACK section of memory
- Via **`call`** and **`ret`** instructions

`call` and `ret` Instructions

`ret` instruction "knows" the return address



```
f:
    …
    call h
    …
    call g
    …
```

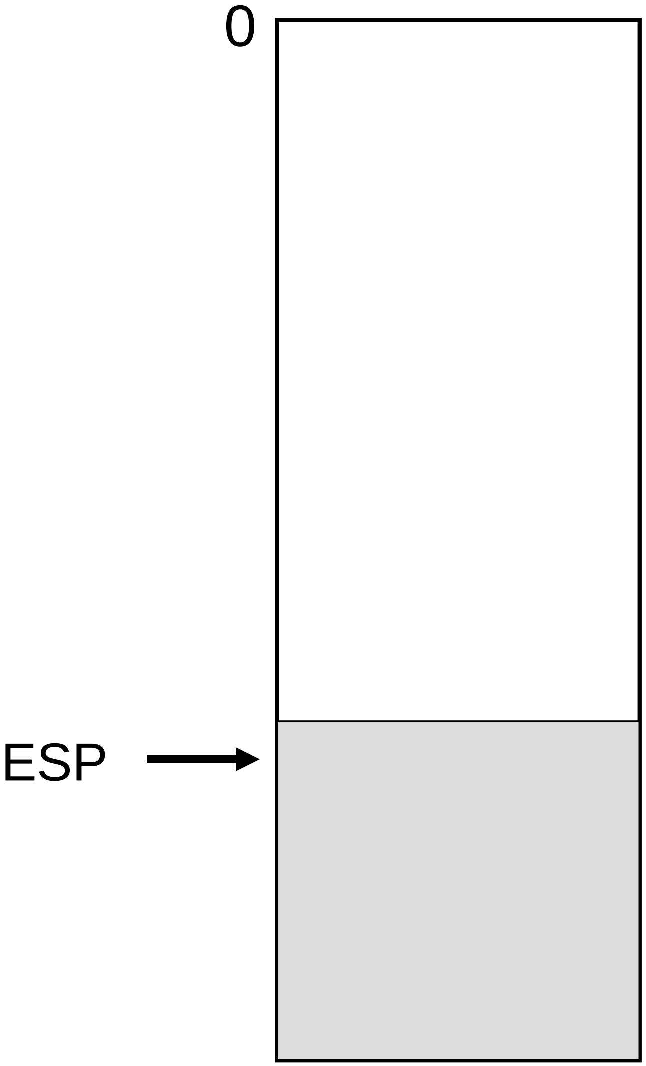
```
h:
    …
    ret
```

```
g:
    …
    call h
    …
    ret
```

Implementation of `call`

- **ESP** (stack pointer) register points to top of stack

Instruction	**Effective Operations**
`pushl src`	`subl $4, %esp` `movl src, (%esp)`
`popl dest`	`movl (%esp), dest` `addl $4, %esp`

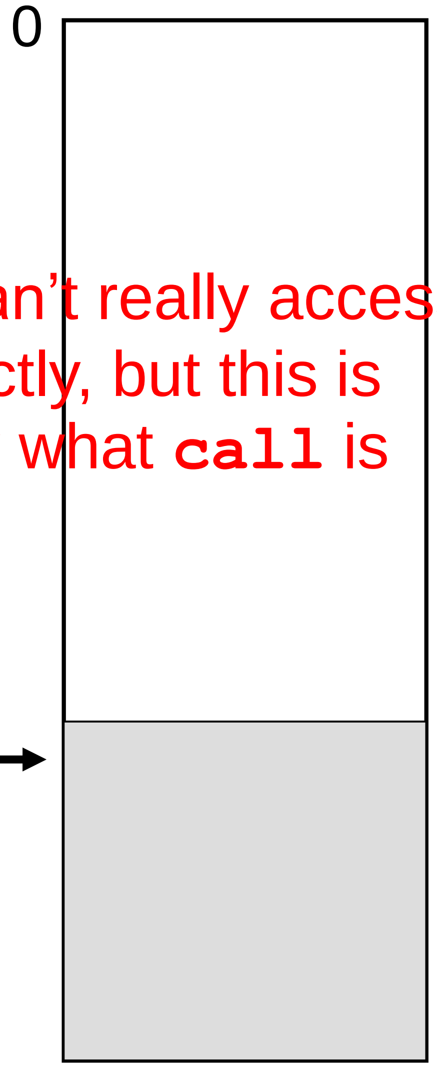


Implementation of `call`

EIP (instruction pointer) register points to next instruction to be executed

Instruction	Effective Operations
`pushl src`	`subl $4, %esp` `movl src, (%esp)`
`popl dest`	`movl (%esp), dest` `addl $4, %esp`
`call addr`	`pushl %eip` `jmp addr`

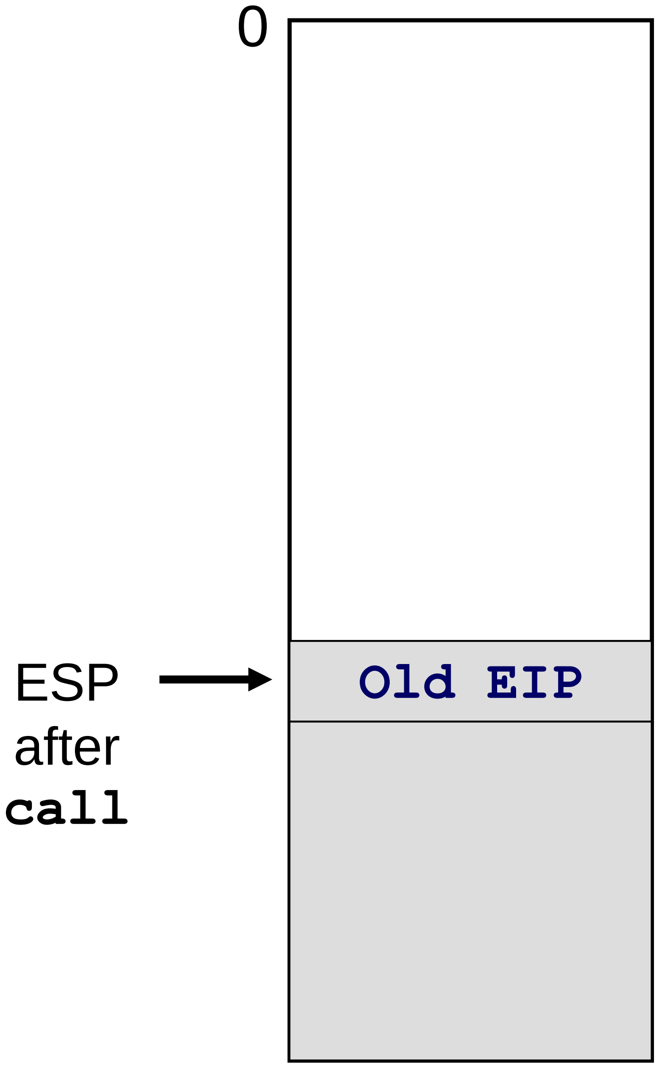
Note: Can't really access EIP directly, but this is implicitly what `call` is doing



`call` instruction pushes return addr (old EIP) onto stack, then jumps

Implementation of `call`

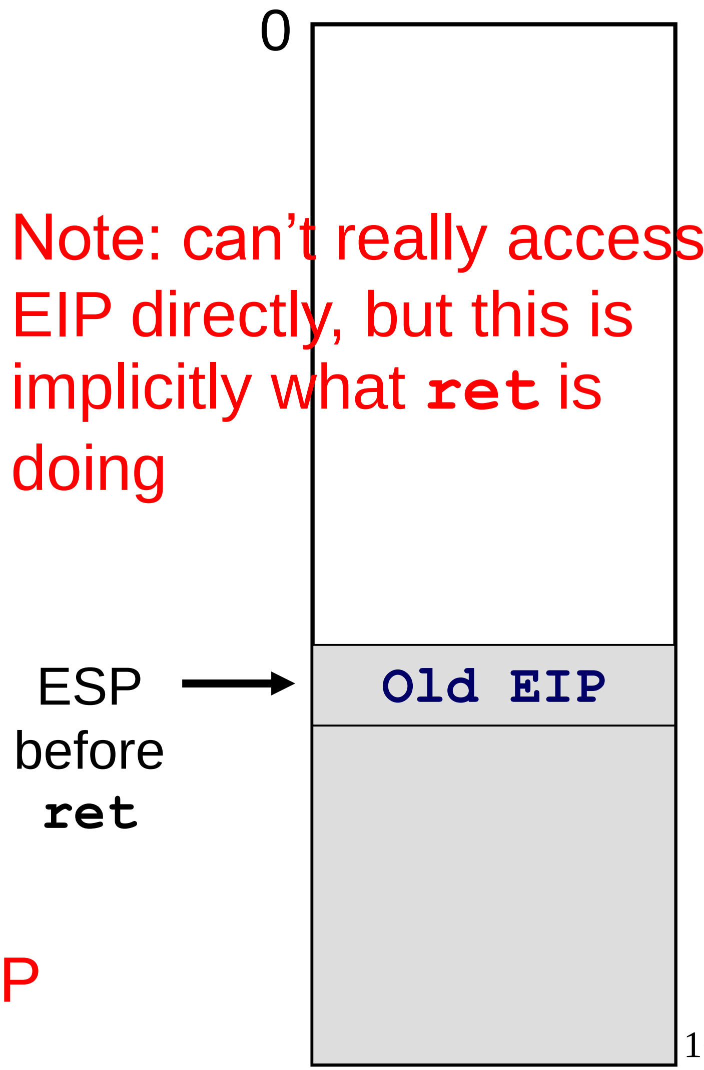
Instruction	**Effective Operations**
`pushl src`	`subl $4, %esp` `movl src, (%esp)`
`popl dest`	`movl (%esp), dest` `addl $4, %esp`
`call addr`	`pushl %eip` `jmp addr`



Implementation of `ret`

Instruction	Effective Operations
`pushl src`	`subl $4, %esp` `movl src, (%esp)`
`popl dest`	`movl (%esp), dest` `addl $4, %esp`
`call addr`	`pushl %eip` `jmp addr`
`ret`	`popl %eip`

Note: can't really access EIP directly, but this is implicitly what **`ret`** is doing



`ret` instruction pops stack, thus placing return addr (old EIP) into EIP

Implementation of `ret`

Instruction	Effective Operations
`pushl src`	`subl $4, %esp` `movl src, (%esp)`
`popl dest`	`movl (%esp), dest` `addl $4, %esp`
`call addr`	`pushl %eip` `jmp addr`
`ret`	`popl %eip`

Running Example

Caller

```
f:
    …
    # Call the function
    call add3
    …
```

Callee

```
add3:
    …
    ret
```

Agenda

Calling and returning

Passing arguments

Storing local variables

Returning a value

Handling registers

An example

Problem 2: Passing Arguments

Problem: How does caller pass *arguments* to callee?

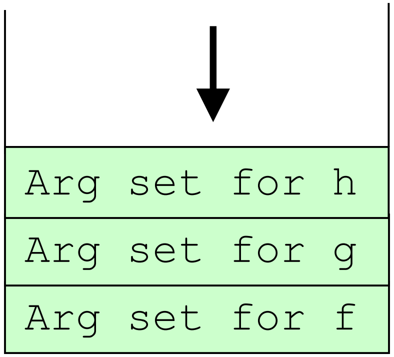
```
void f(void)
{  …
   n = add3(3, 4, 5);
   …
}
```

```
int add3(int a, int b, int c)
{  int d;
   d = a + b + c;
   return d;
}
```

IA-32 Solution: Use the Stack

Observations (déjà vu):

- May need to store many arg sets
    - The number of arg sets is not known in advance
    - Arg set must be saved for as long as the invocation of this function is live, and discarded thereafter
- Stored arg sets are destroyed in reverse order of creation
    - f() calls g() => arg set for g is created
    - g() calls h() => arg set for h is created
    - h() returns to g() => arg set for h is destroyed
    - g() returns to f() => arg set for g is destroyed
- LIFO data structure (stack) is appropriate

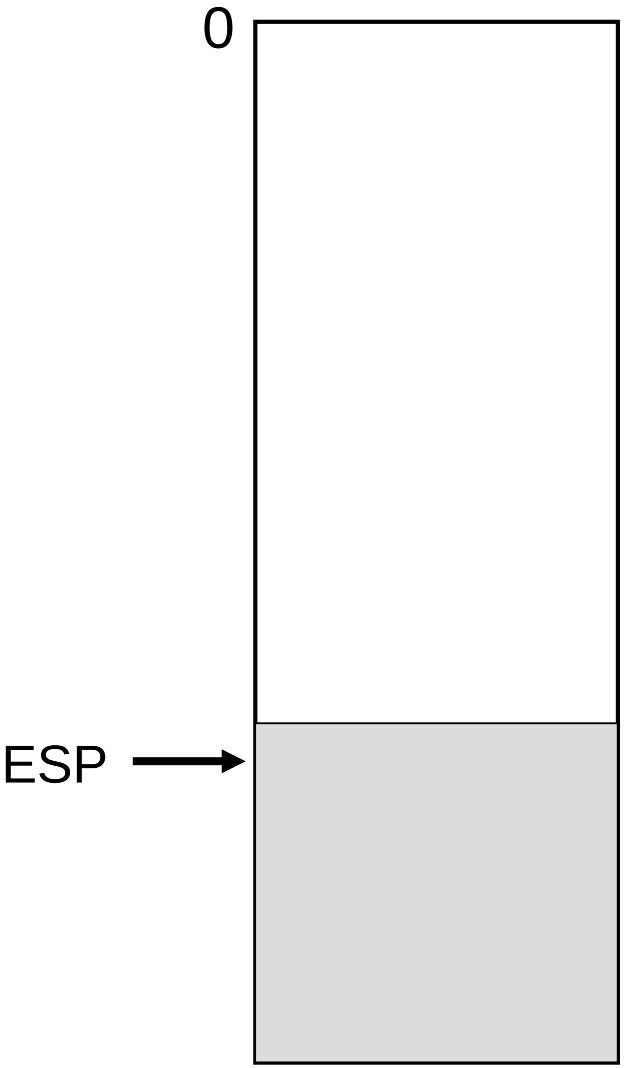


IA 32 solution:

- Use the STACK section of memory

Passing Args on the Stack

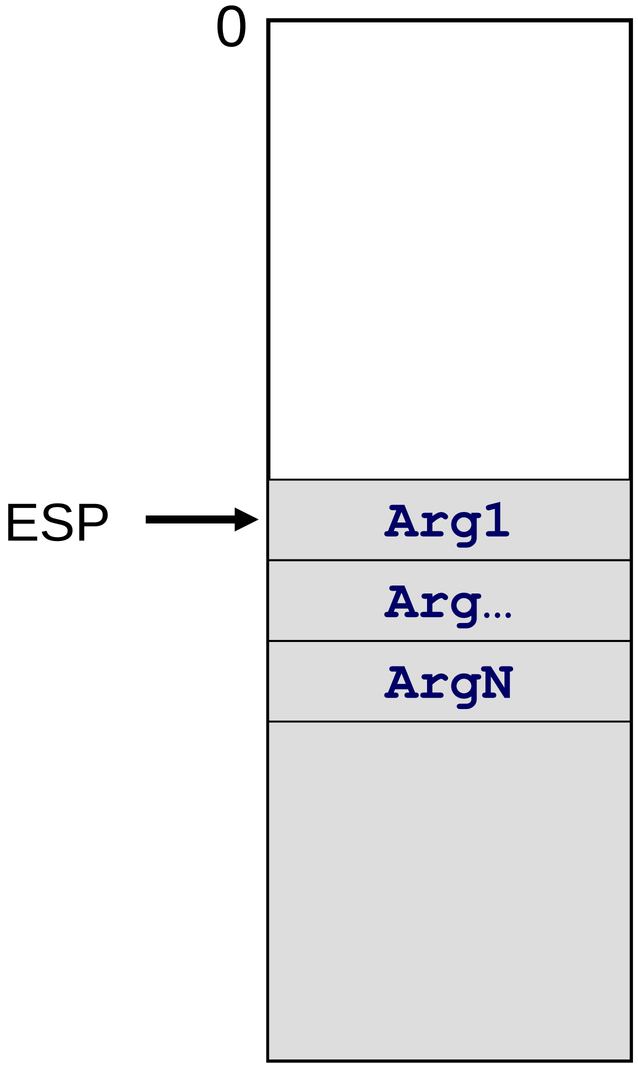
Before executing `call` instruction…



Passing Args on the Stack

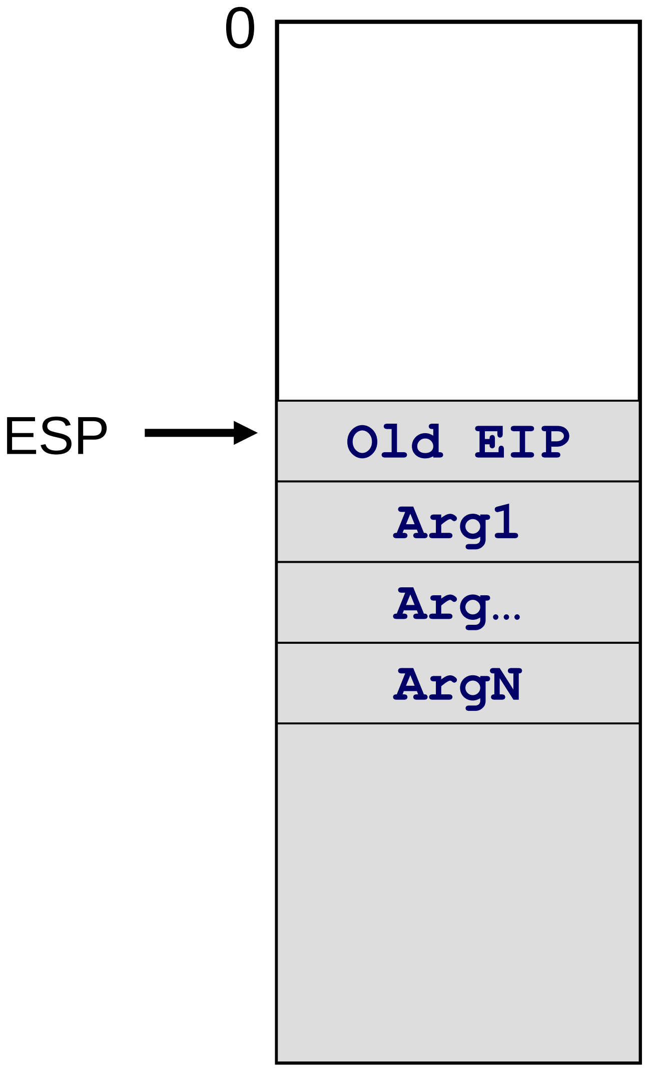
Caller pushes args in reverse order

- Push Nth arg first
- Push 1st arg last
- So 1st arg is at top of the stack at the time of the **`call`**



Passing Args on the Stack

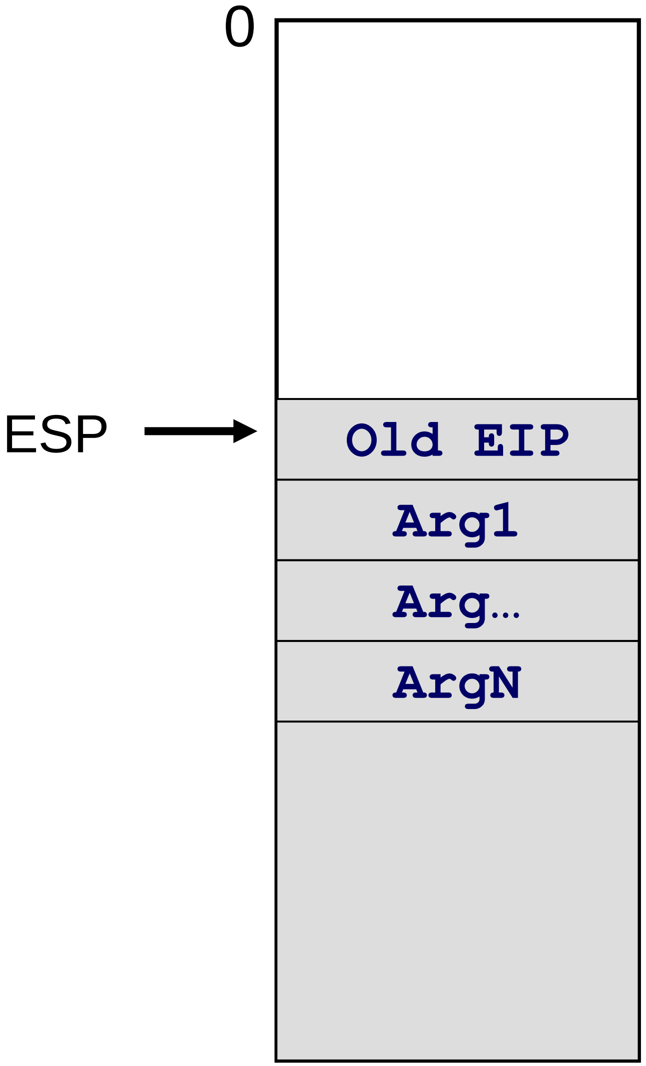
Caller executes **`call`** instruction



Passing Args on the Stack

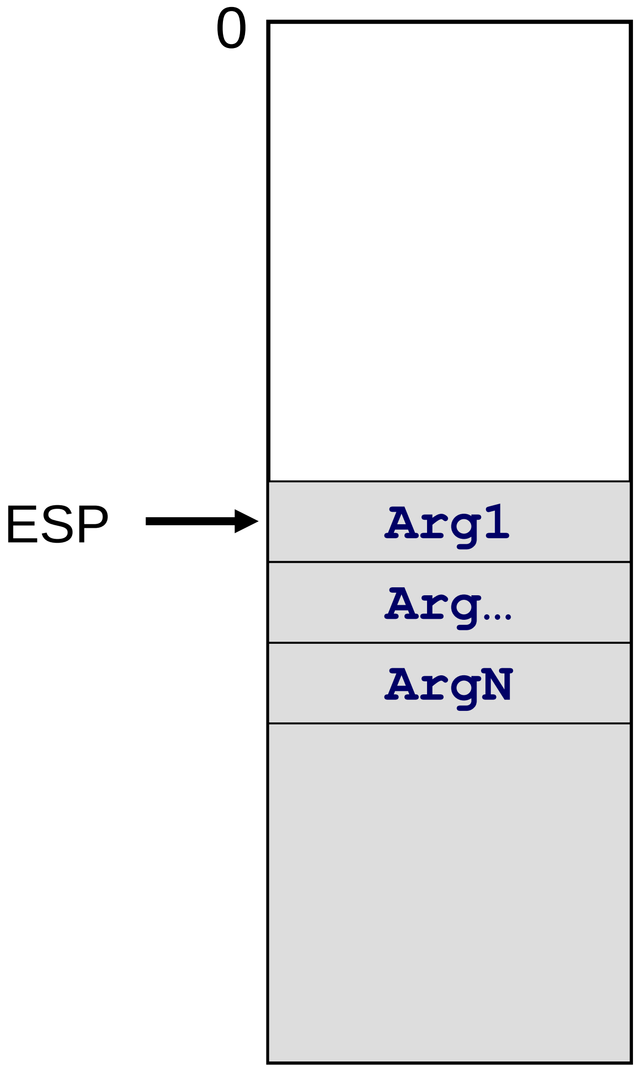
Callee accesses args relative to ESP

- Arg1 as **`4(%esp)`**
- Arg2 as **`8(%esp)`**
- …



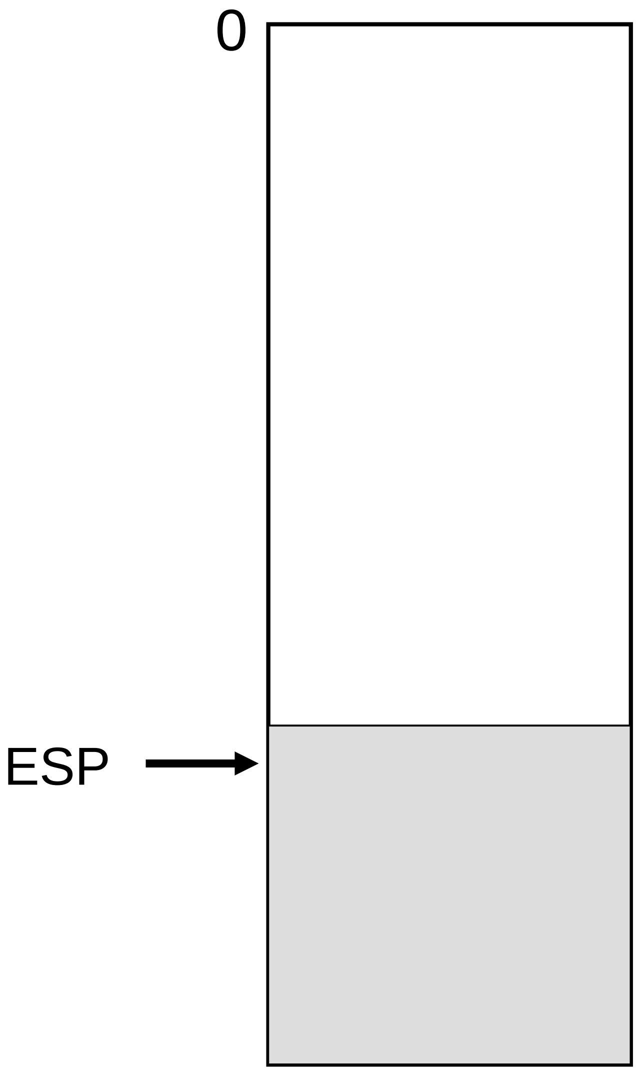
Passing Args on the Stack

Callee executes **`ret`** instruction



Passing Args on the Stack

Caller pops args from the stack



Running Example

```
f:
    …
    # Push arguments
    pushl $5
    pushl $4
    pushl $3

    # Call the function
    call add3

    # Pop arguments
    addl $12, %esp
    …
```

```
add3:
    …
    # Use arguments
    movl 4(%esp), %eax
    addl 8(%esp), %eax
    addl 12(%esp), %eax
    …
    ret
```

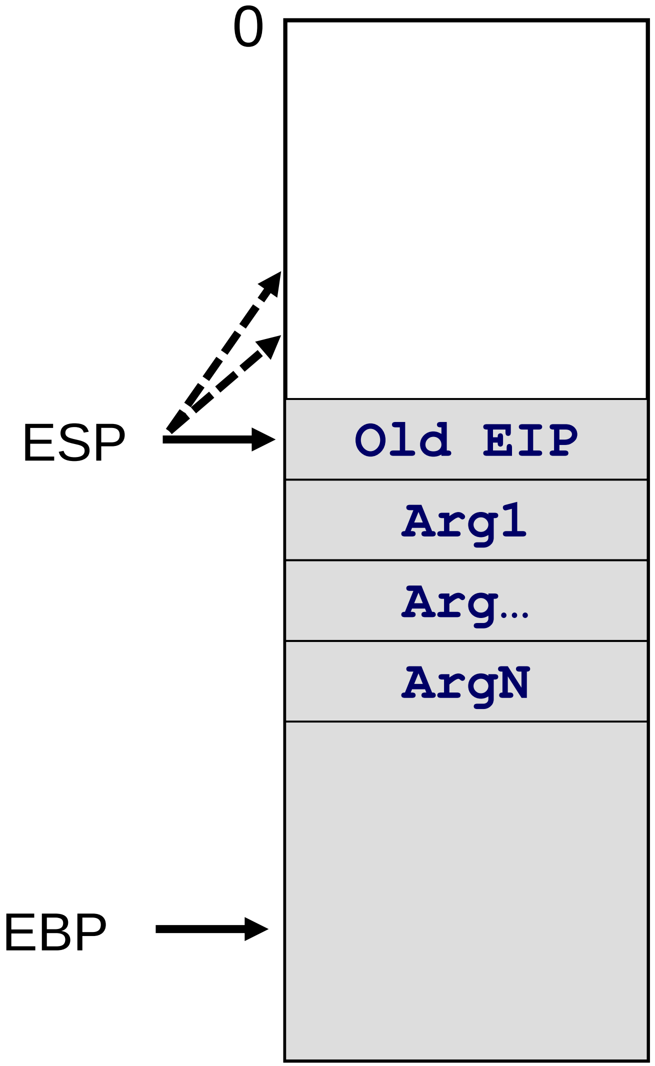
Base Pointer Register: EBP

Problem:

- As callee executes, ESP may change
  - E.g., preparing to call another function
- Error-prone for callee to reference args as offsets relative to ESP

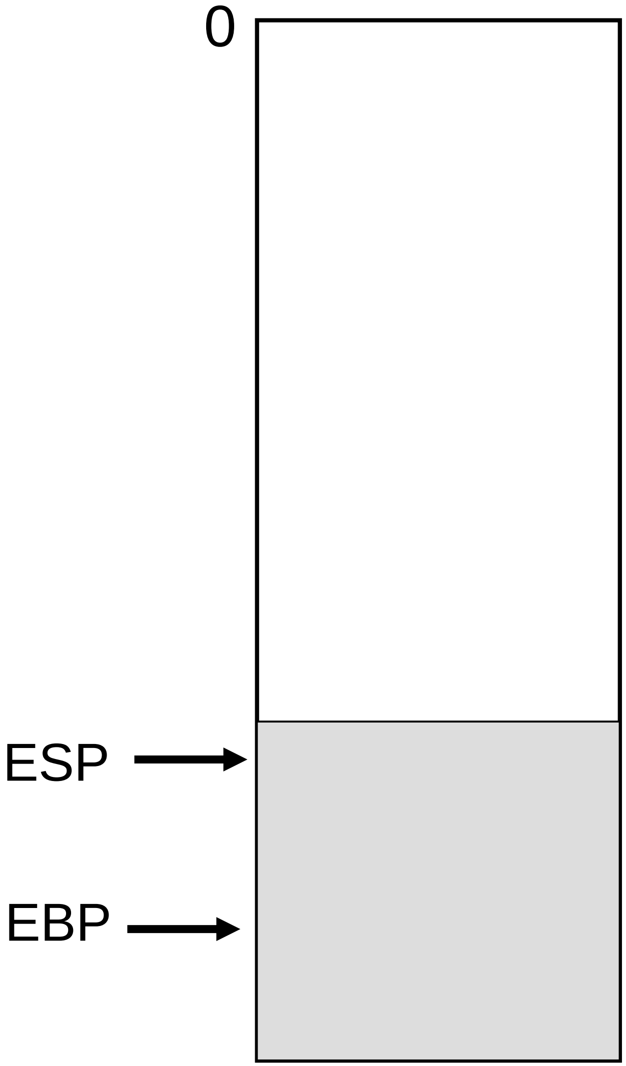
Solution:

- Use **EBP** (base pointer) register
  - EBP doesn't change during callee's execution
- Use EBP as fixed reference point to access args



Passing Args on the Stack (v2)

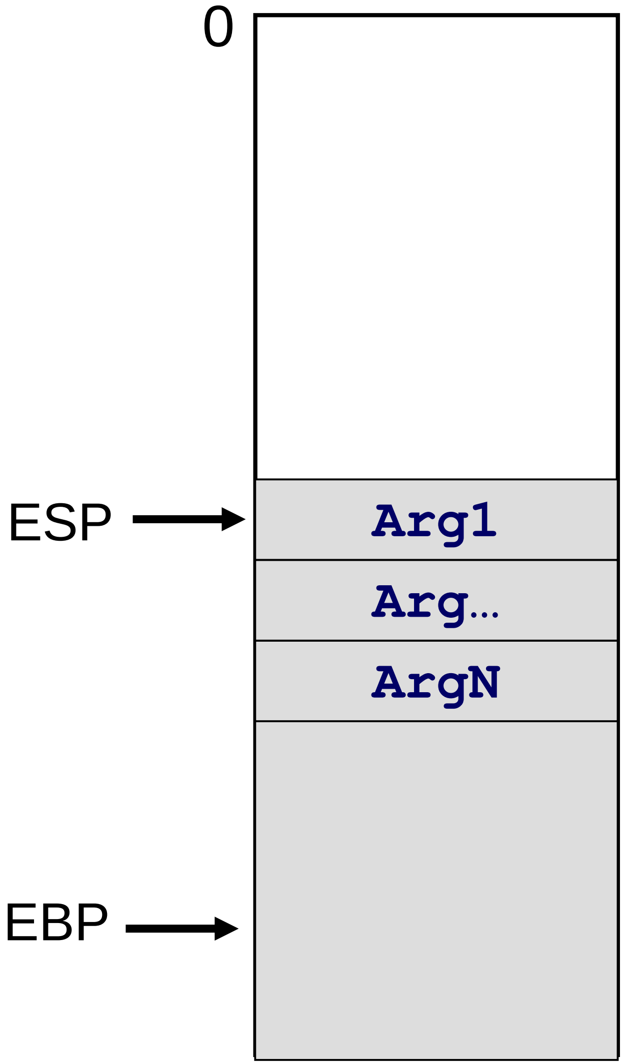
Before executing **`call`** instruction…



Passing Args on the Stack (v2)

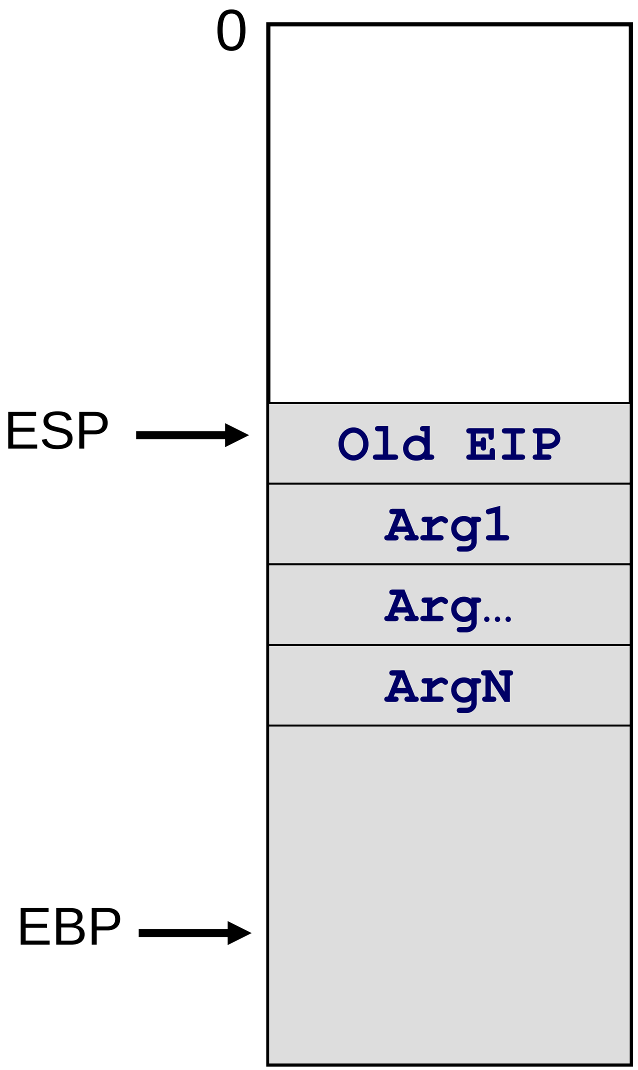
Caller pushes args in reverse order

- Push Nth arg first
- Push 1st arg last
- So 1st arg is at top of the stack at the time of the **`call`**



Passing Args on the Stack (v2)

Caller executes **`call`** instruction



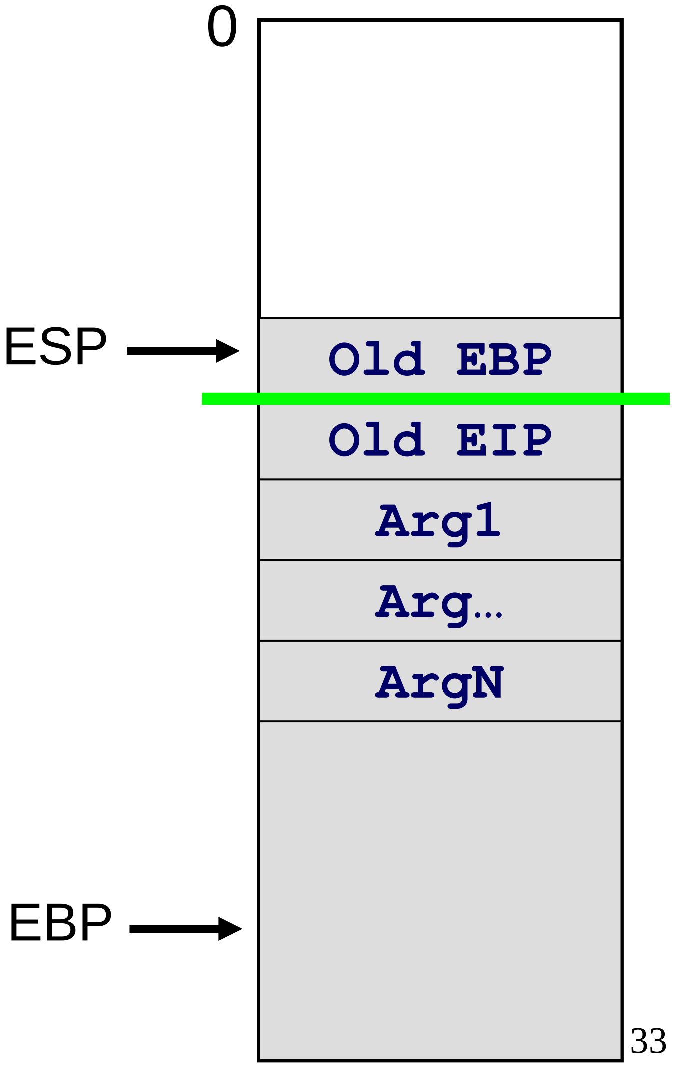
Passing Args on the Stack (v2)

Need to save old value of EBP

- Before overwriting EBP register

Callee executes “prolog”

```
pushl %ebp
movl %esp, %ebp
```

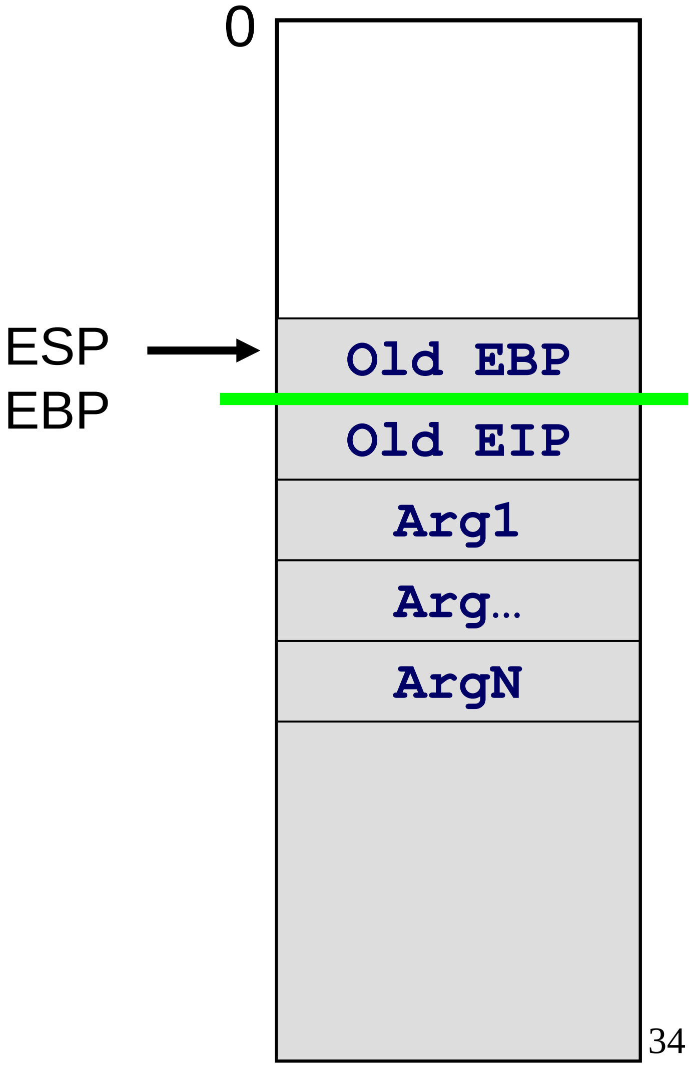


Passing Args on the Stack (v2)

Callee executes "prolog"

```
pushl %ebp
movl %esp, %ebp
```

Regardless of ESP, callee can reference Arg1 as **`8(%ebp)`**, Arg2 as **`12(%ebp)`**, etc.



0

ESP

EBP

Old EBP

Old EIP

Arg1

Arg…

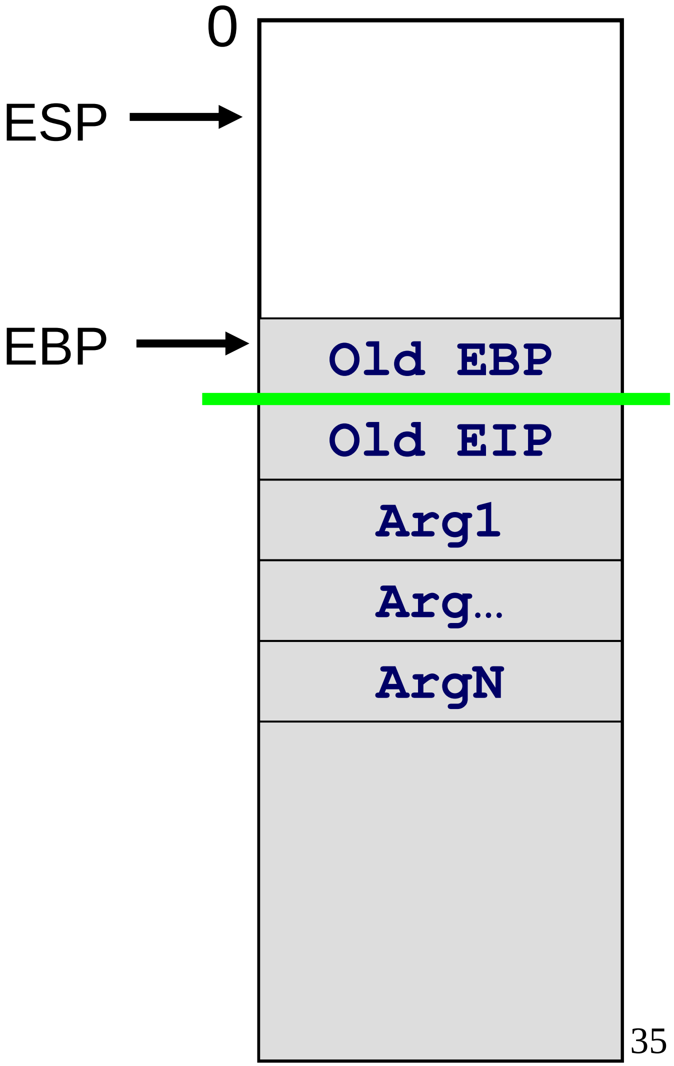
ArgN

Passing Args on the Stack (v2)

Before returning, callee must restore ESP and EBP to their old values

Callee executes "epilog"

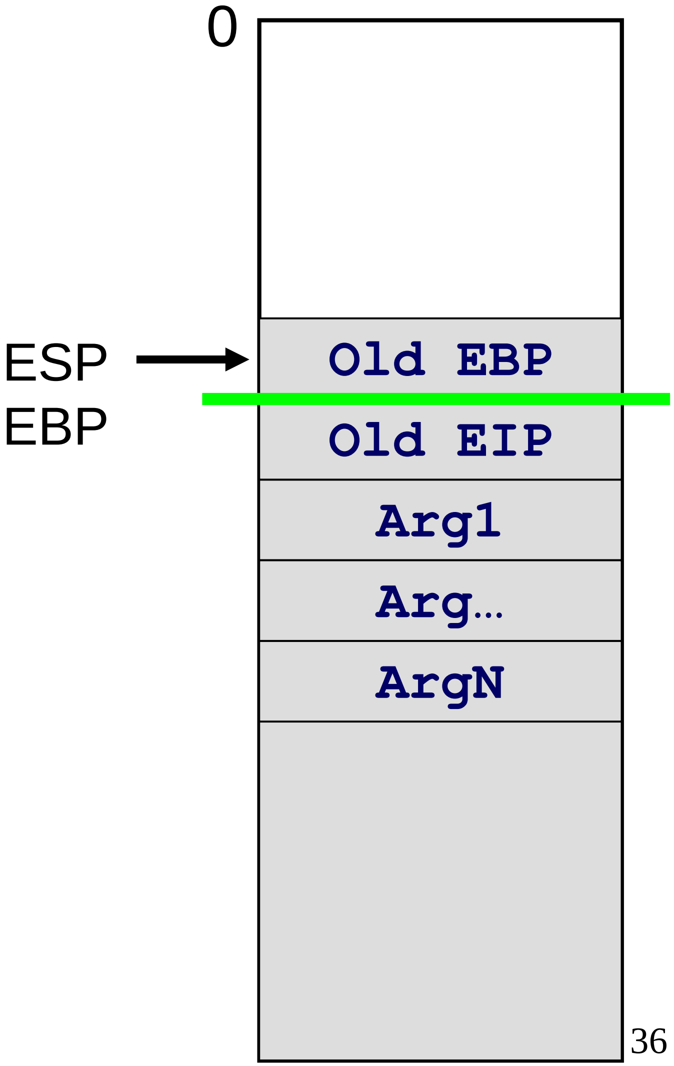
```
movl %ebp, %esp
popl %ebp
```



Passing Args on the Stack (v2)

Callee executes "epilog"

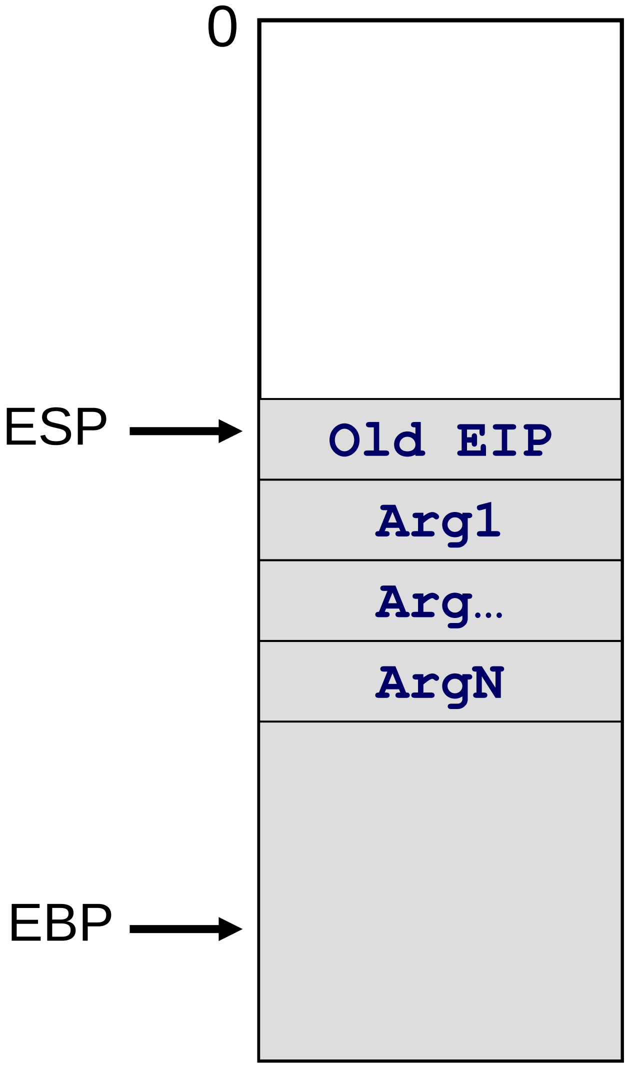
```
movl %ebp, %esp
popl %ebp
```



Passing Args on the Stack (v2)

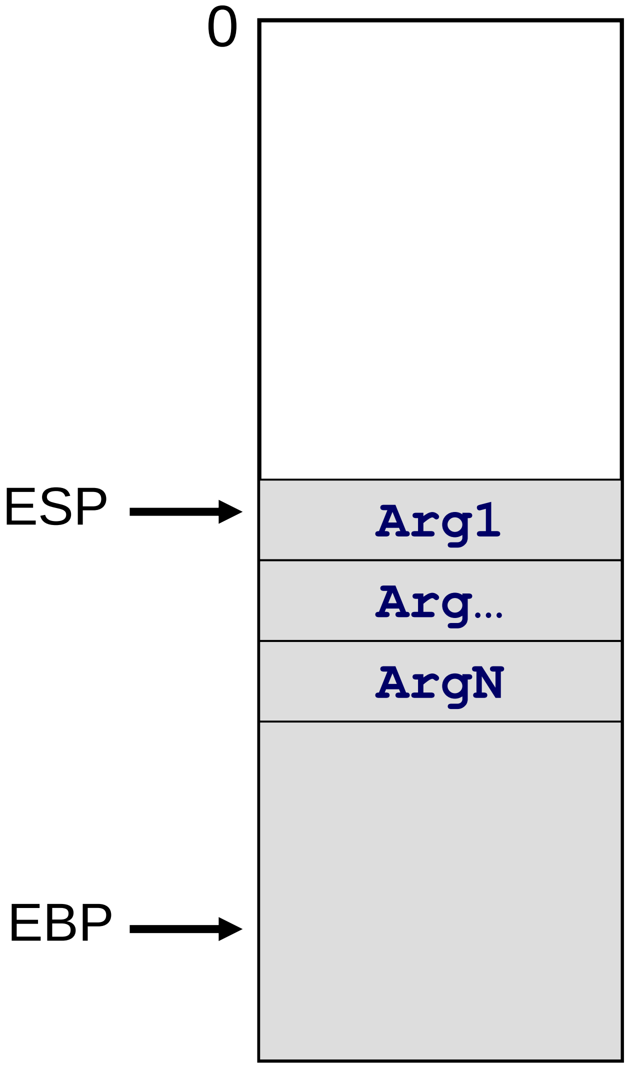
Callee executes “epilog”

```
movl %ebp, %esp
popl %ebp
```



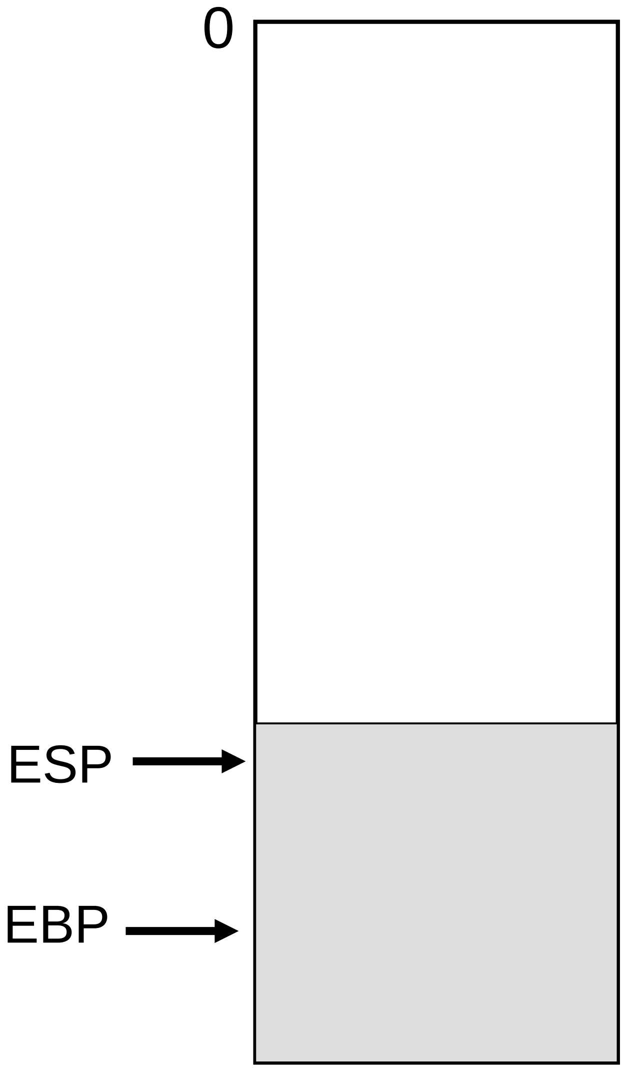
Passing Args on the Stack (v2)

Callee executes **`ret`** instruction



Passing Args on the Stack (v2)

Caller pops args from the stack



Running Example

```
f:
    …
    # Push arguments
    pushl $5
    pushl $4
    pushl $3

    # Call the function
    call add3

    # Pop arguments
    addl $12, %esp
    …
```

```
add3:
    pushl %ebp
    movl %esp, %ebp
    …
    # Use arguments
    movl 8(%ebp), %eax
    addl 12(%ebp), %eax
    addl 16(%ebp), %eax
    …
    movl %ebp, %esp
    popl %ebp
    ret
```

Agenda

Calling and returning

Passing arguments

Storing local variables

Returning a value

Handling registers

An example

Problem 3: Storing Local Variables

Where does callee function store its *local variables?*

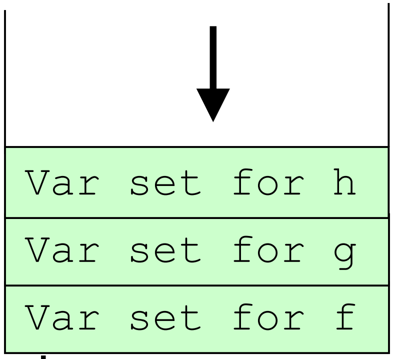
```
void f(void)
{   …
    n = add3(3, 4, 5);
    …
}
```

```
int add3(int a, int b, int c)
{   int d;
    d = a + b + c;
    return d;
}
```

IA-32 Solution: Use the Stack

Observations (déjà vu again!):

- May need to store many local var sets
  - The number of local var sets is not known in advance
  - Local var set must be saved for as long as the invocation of this function is live, and discarded thereafter
- Stored local var sets are destroyed in reverse order of creation
  - f() calls g() => local vars set for g is created
  - g() calls h() => local vars set for h is created
  - h() returns to g() => local vars set for h is destroyed
  - g() returns to f() => local vars set for g is destroyed
- LIFO data structure (stack) is appropriate



IA 32 solution:

- Use the STACK section of memory

Storing Variables on the Stack

Callee allocates space for its local variables on the stack

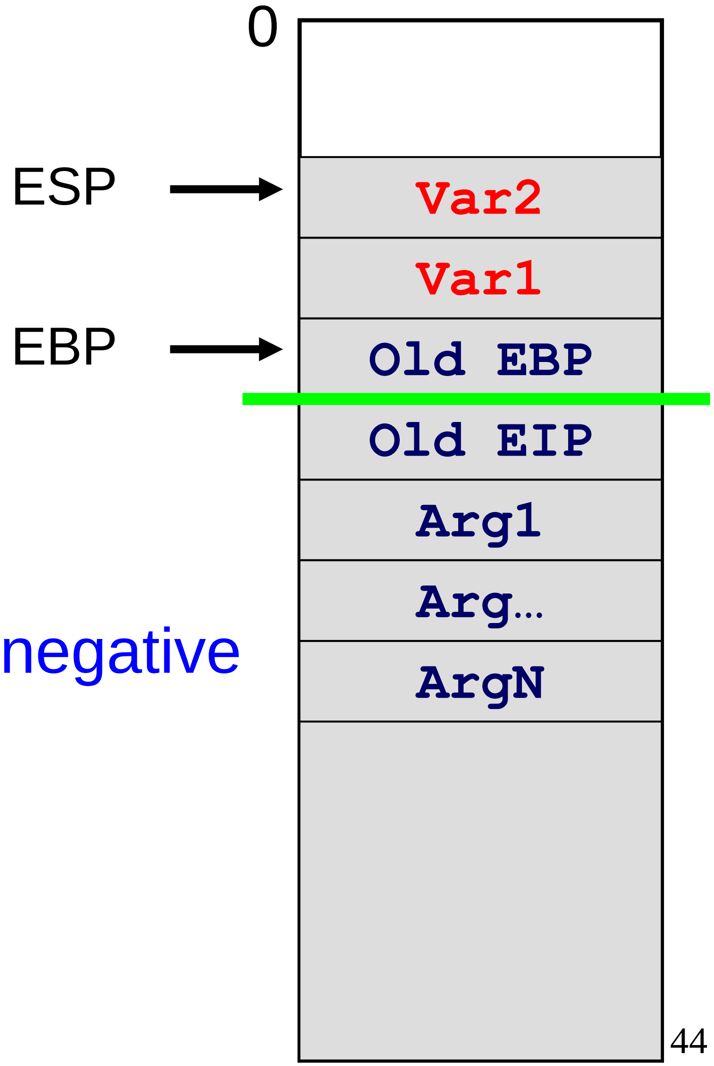
- Via **`pushl`** instructions
- Via **`subl $4,%esp`** instructions

Example: allocate memory for two integers

- **`subl   $4,%esp # int i;`**
- **`pushl $5       # int j = 5;`**

Callee references local variables as negative offsets relative to EBP

- **`-4(%ebp)   # Access i`**
- **`-8(%ebp)   # Access j`**



Running Example

```
f:
    …
    # Push arguments
    pushl $5
    pushl $4
    pushl $3

    # Call the function
    call add3

    # Pop arguments
    addl $12, %esp
    …
```

```
add3:
    pushl %ebp
    movl %esp, %ebp

    # Allocate mem for local var
    subl $4, %esp

    # Use arguments
    movl 8(%ebp), %eax
    addl 12(%ebp), %eax
    addl 16(%ebp), %eax

    # Use local variable
    movl %eax, -4(%ebp)
    …
    movl %ebp, %esp
    popl %ebp
    ret
```

Agenda

Calling and returning

Passing arguments

Storing local variables

Returning a value

Handling registers

An example

Problem 4: Return Values

Problem: How does callee function send return value back to caller function?

```
void f(void)
{   …
    n = add3(3, 4, 5);
    …
}
```

```
int add3(int a, int b, int c)
{   int d;
    d = a + b + c;
    return d;
}
```

IA-32 Solution: Use EAX

In principle

- Store return value in stack frame of caller

Or, for efficiency

- Known small size => store return value in register
- Other => store return value in stack

IA-32 convention

- Integer or pointer:
  - Store return value in EAX
- Floating-point number:
  - Store return value in floating-point register
  - (Beyond scope of COS 217)
- Structure:
  - Store return value on stack
  - (Beyond scope of COS 217)

Running Example

```
f:
    …
    # Push arguments
    pushl $5
    pushl $4
    pushl $3

    # Call the function
    call add3

    # Pop arguments
    addl $12, %esp

    # Use return value
    movl %eax, n
    …
```

```
add3:
    pushl %ebp
    movl %esp, %ebp

    # Allocate mem for local var
    subl $4, %esp

    # Use arguments
    movl 8(%ebp), %eax
    addl 12(%ebp), %eax
    addl 16(%ebp), %eax

    # Use local variable
    movl %eax, -4(%ebp)

    # Indicate return value
    movl -4(%ebp), %eax

    movl %ebp, %esp
    popl %ebp
    ret
```

Agenda

Calling and returning

Passing arguments

Storing local variables

Returning a value

Handling registers

An example

Problem 5: Handling Registers

Observation: Registers are a finite resource

- In principle: Each function should have its own registers
- In reality: All functions share same small set of registers

Problem: How do caller and callee use same set of registers without interference?

- Callee may use register that the caller also is using
- When callee returns control to caller, old register contents may have been lost
- Caller function cannot continue where it left off

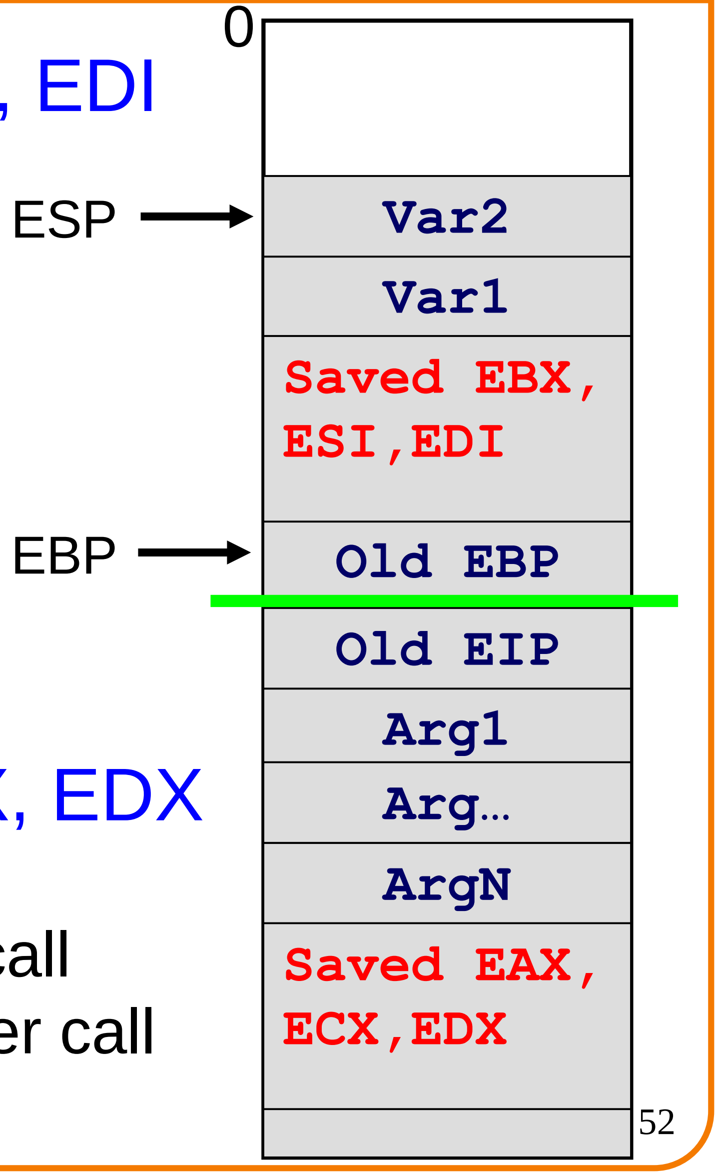
IA-32 Solution: Register Conventions

Callee-save registers: EBX, ESI, EDI

- If necessary…
  - Callee saves to stack after prolog
  - Callee restores from stack before epilog
- Caller can assume that values in EBX, ESI, EDI will not be changed by callee

Caller-save registers: EAX, ECX, EDX

- If necessary…
  - Caller saves to stack before call
  - Caller restores from stack after call



Running Example

```
f:
   …
   # Save EAX, ECX, EDX
   pushl %eax
   pushl %ecx
   pushl %edx

   # Push arguments
   pushl $5
   pushl $4
   pushl $3

   # Call the function
   call add3
```

```
   # Pop arguments
   addl $12, %esp

   # Use return value
   movl %eax, n

   # Restore EAX, ECX, EDX
   popl %edx
   popl %ecx
   popl %eax
   …
```

Running Example

```
add3:
   pushl %ebp
   movl %esp, %ebp

   # Save EBX, ESI, EDI
   pushl %ebx
   pushl %esi
   pushl %edi

   # Allocate mem for local var
   subl $4, %esp

   # Use arguments
   movl 8(%ebp), %eax
   addl 12(%ebp), %eax
   addl 16(%ebp), %eax

   # Use local variable
   movl %eax, -16(%ebp)
```

Not necessary to save callee-save registers in this particular function

```
   # Indicate return value
   movl -4(%ebp), %eax

   # Restore EBX, ESI, EDI
   movl -12(%ebp), %edi
   movl -8(%ebp), %esi
   movl -4(%ebp), %ebx

   movl %ebp, %esp
   popl %ebp
   ret
```

Stack Frames

Summary of IA-32 function handling:

Stack has one **stack frame** per active function invocation

ESP points to top (low memory) of current stack frame

EBP points to bottom (high memory) of current stack frame

Stack frame contains:

- Values of caller-save registers
- Arguments to be passed to callee function
- Return address (old EIP)
- Old EBP
- Values of callee-save registers
- Local variables

Agenda

Calling and returning

Passing arguments

Storing local variables

Returning a value

Handling registers

An example

Trace of Running Example

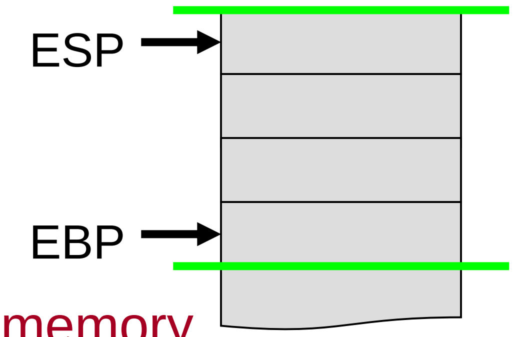
```
int add3(int a, int b, int c)
{   int d;
    d = a + b + c;
    return d;
}
```

```
void f(void)
{   …
    n = add3(3, 4, 5);
    …
}
```

Trace of Running Example 1

```
n = add3(3, 4, 5);
```

Low memory



High memory

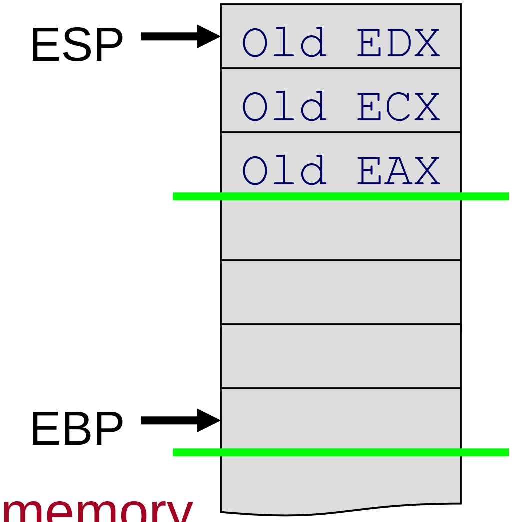
Trace of Running Example 2

```
n = add3(3, 4, 5);
```

Low memory

 # Save caller-save registers if necessary

```
pushl %eax
pushl %ecx
pushl %edx
```

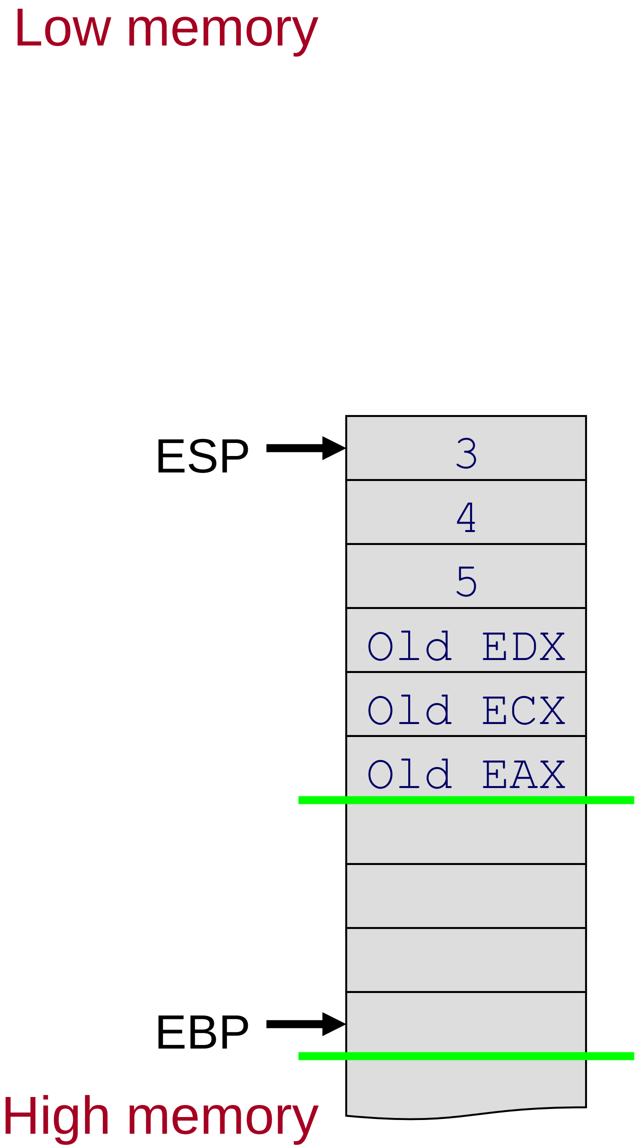


High memory

Trace of Running Example 3

`n = add3(3, 4, 5);`

```
# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push arguments
pushl $5
pushl $4
pushl $3
```

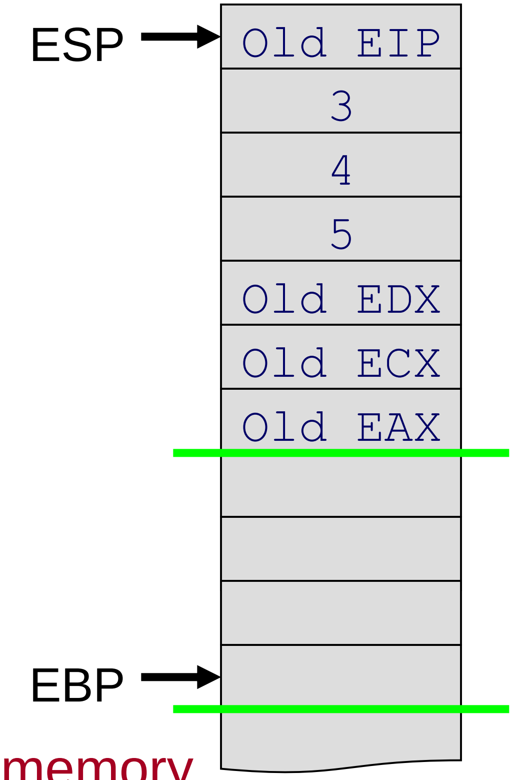


Trace of Running Example 4

```
n = add3(3, 4, 5);
```

Low memory

```
# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push arguments
pushl $5
pushl $4
pushl $3
# Call add3
call add3
```



High memory

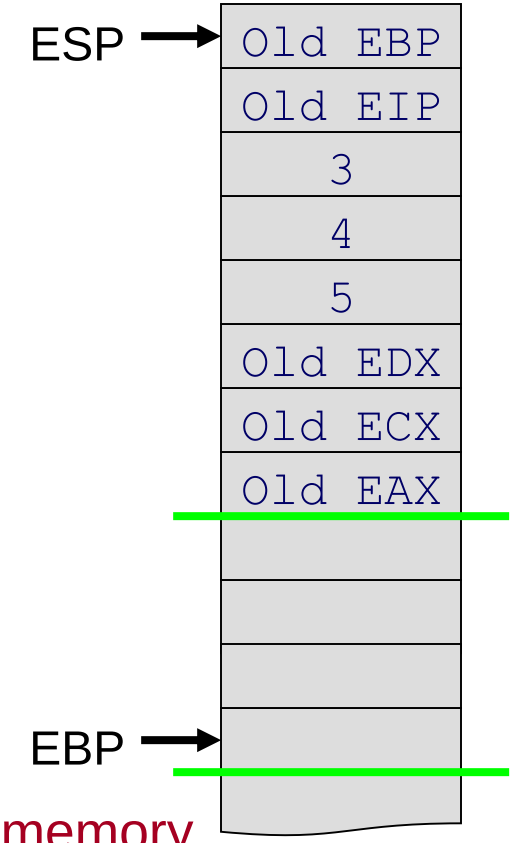
Trace of Running Example 5

```
int add3(int a, int b, int c)
{   int d;
    d = a + b + c;
    return d;
}
```

 # Save old EBP
pushl %ebp

Prolog

Low memory



High memory

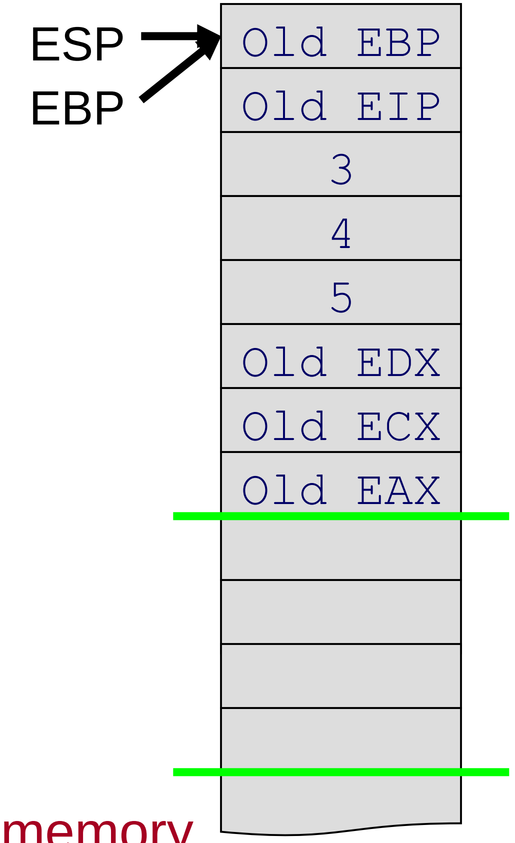
Trace of Running Example 6

```
int add3(int a, int b, int c)
{   int d;
    d = a + b + c;
    return d;
}
```

```
# Save old EBP
pushl %ebp
# Change EBP
movl %esp, %ebp
```

Prolog

Low memory



ESP

EBP

Old EBP

Old EIP

3

4

5

Old EDX

Old ECX

Old EAX

High memory

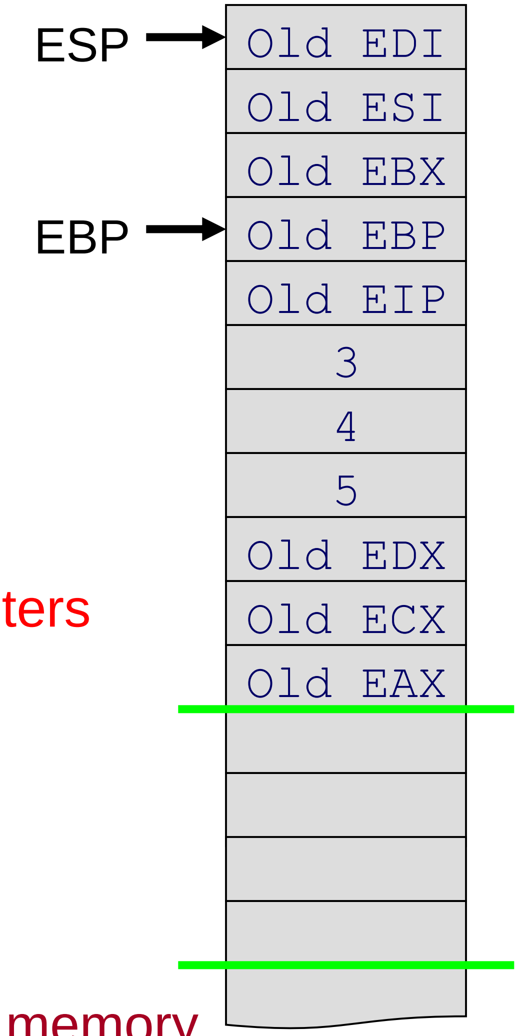
Trace of Running Example 7

```
int add3(int a, int b, int c)
{   int d;
    d = a + b + c;
    return d;
}
```

```
# Save old EBP
pushl %ebp
# Change EBP
movl %esp, %ebp
# Save callee-save registers if necessary
pushl %ebx
pushl %esi
pushl %edi
```

Unnecessary here; add3 will not change the values in these registers

Low memory

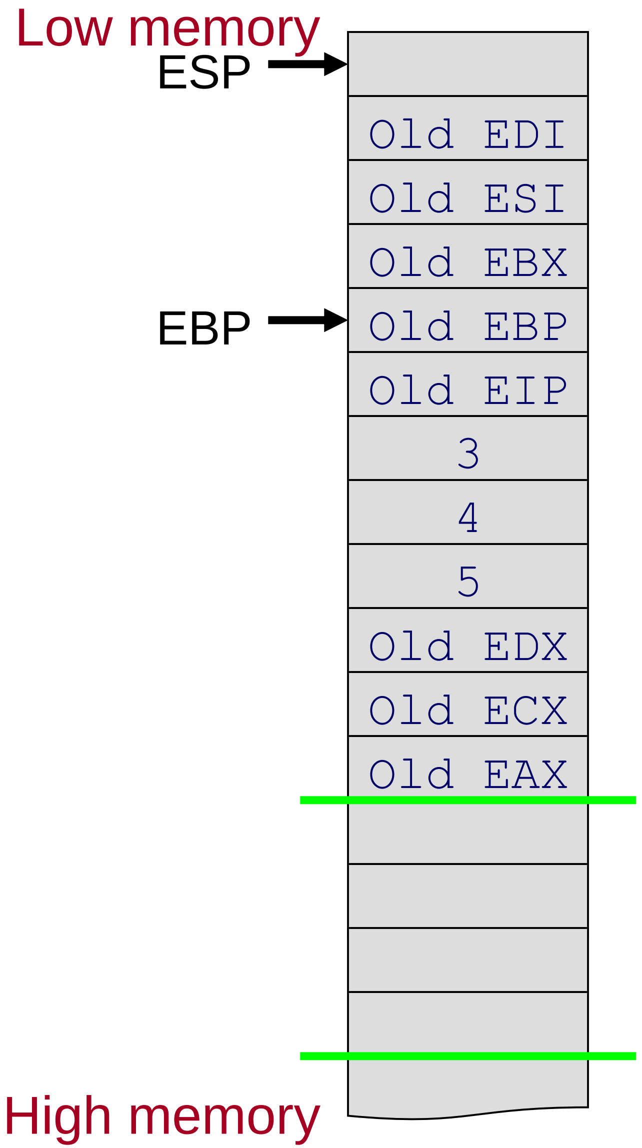


High memory

Trace of Running Example 8

```
int add3(int a, int b, int c)
{   int d;
    d = a + b + c;
    return d;
}
```

```
# Save old EBP
pushl %ebp
# Change EBP
movl %esp, %ebp
# Save caller-save registers if necessary
pushl %ebx
pushl %esi
pushl %edi
# Allocate space for local variable
subl $4, %esp
```



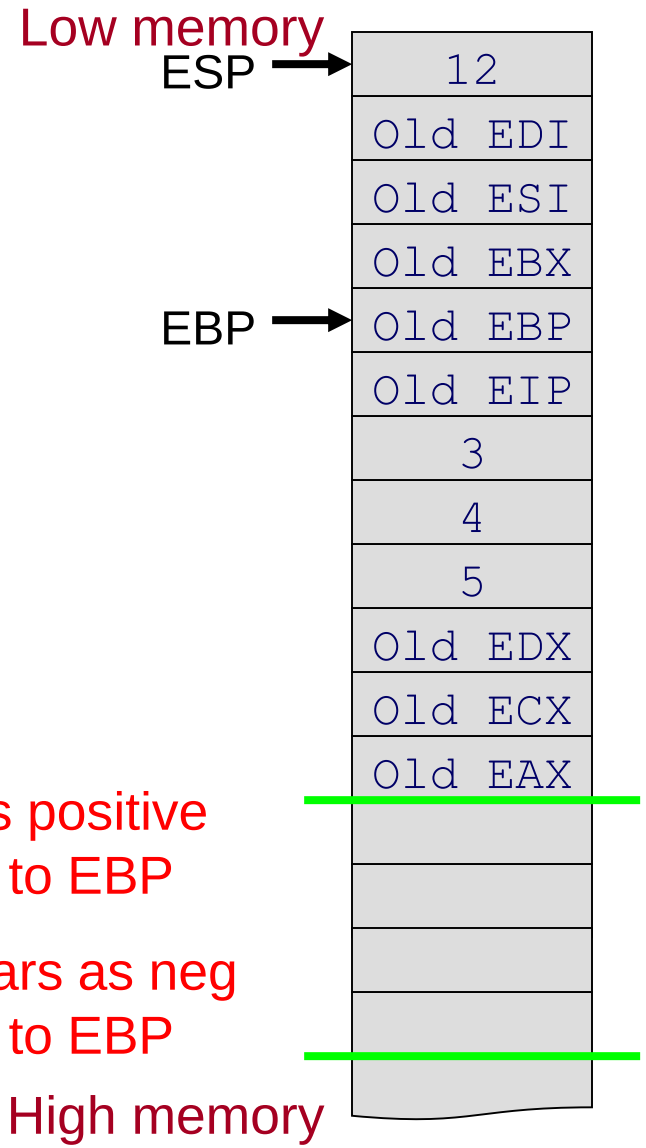
Trace of Running Example 9

```
int add3(int a, int b, int c)
{  int d;
   d = a + b + c;
   return d;
}
```

```
# Save old EBP
pushl %ebp
# Change EBP
movl %esp, %ebp
# Save caller-save registers if necessary
pushl %ebx
pushl %esi
pushl %edi
# Allocate space for local variable
subl $4, %esp
# Perform the addition
movl 8(%ebp), %eax
addl 12(%ebp), %eax
addl 16(%ebp), %eax
movl %eax, -16(%ebp)
```

Access args as positive offsets relative to EBP

Access local vars as neg offsets relative to EBP



Low memory

ESP → 12

Old EDI

Old ESI

Old EBX

EBP → Old EBP

Old EIP

3

4

5

Old EDX

Old ECX

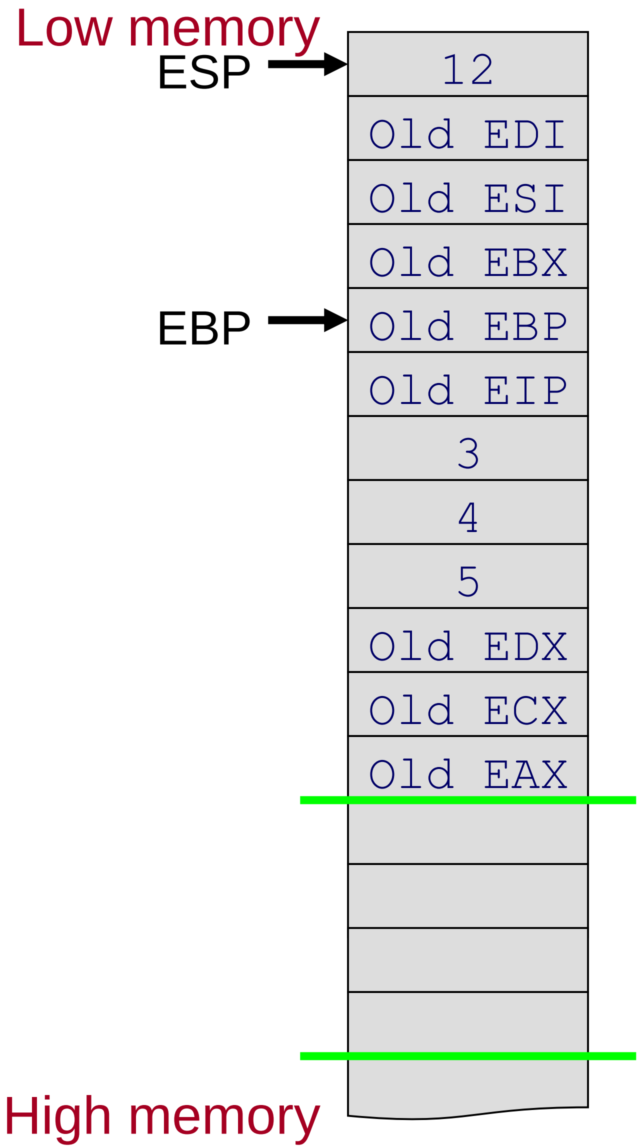
Old EAX

High memory

Trace of Running Example 10

```
int add3(int a, int b, int c)
{   int d;
    d = a + b + c;
    return d;
}
```

```
# Copy the return value to EAX
movl -16(%ebp), %eax
# Restore callee-save registers if necessary
movl -12(%ebp), %edi
movl -8(%ebp), %esi
movl -4(%ebp), %ebx
```

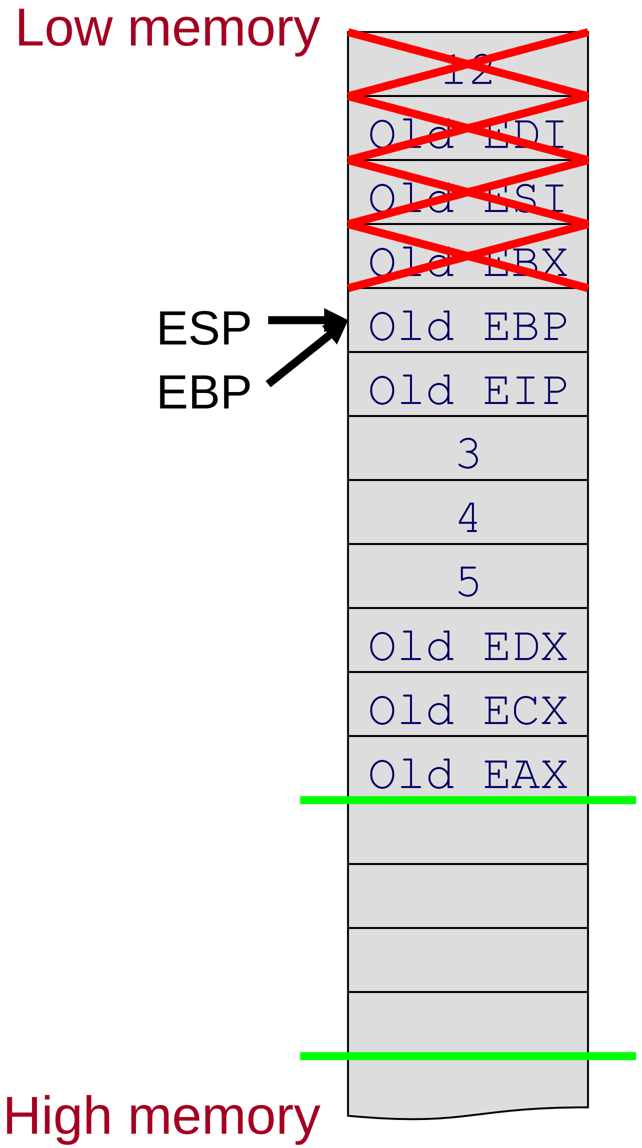


Trace of Running Example 11

```
int add3(int a, int b, int c)
{   int d;
    d = a + b + c;
    return d;
}
```

 # Copy the return value to EAX
movl -16(%ebp), %eax
 # Restore callee-save registers if necessary
movl -12(%ebp), %edi
movl -8(%ebp), %esi
movl -4(%ebp), %ebx
 # Restore ESP
movl %ebp, %esp

Epilog

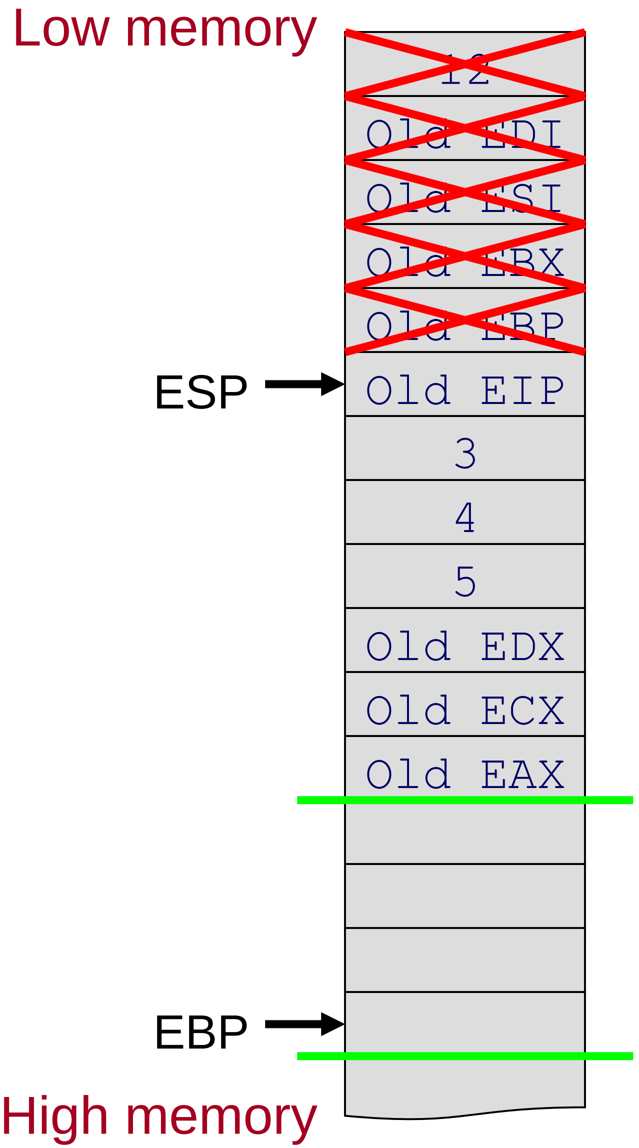


Trace of Running Example 12

```
int add3(int a, int b, int c)
{   int d;
    d = a + b + c;
    return d;
}
```

```
# Copy the return value to EAX
movl -16(%ebp), %eax
# Restore callee-save registers if necessary
movl -12(%ebp), %edi
movl -8(%ebp), %esi
movl -4(%ebp), %ebx
# Restore ESP
movl %ebp, %esp
# Restore EBP
popl %ebp
```

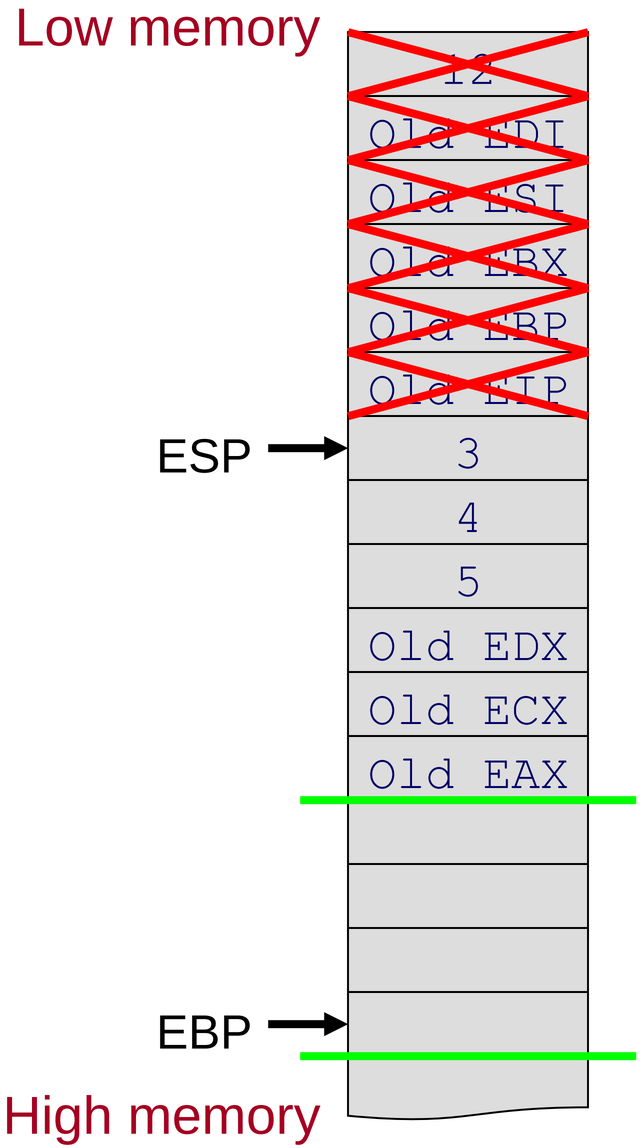
Epilog



Trace of Running Example 13

```
int add3(int a, int b, int c)
{   int d;
    d = a + b + c;
    return d;
}
```

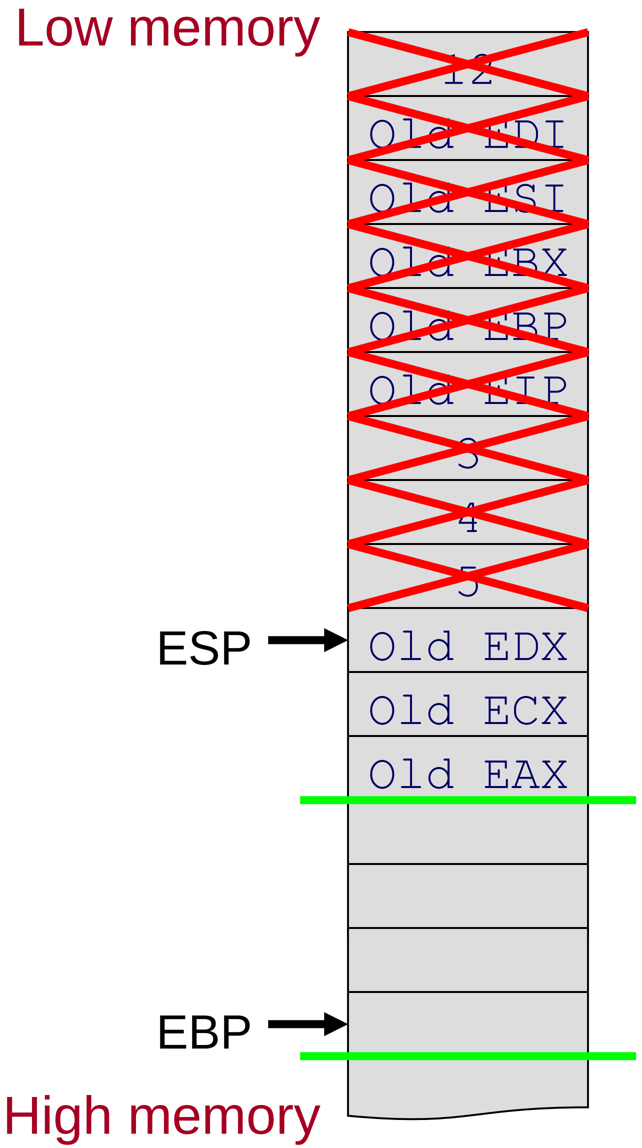
```
# Copy the return value to EAX
movl -16(%ebp), %eax
# Restore callee-save registers if necessary
movl -12(%ebp), %edi
movl -8(%ebp), %esi
movl -4(%ebp), %ebx
# Restore ESP
movl %ebp, %esp
# Restore EBP
popl %ebp
# Return to calling function
ret
```



Trace of Running Example 14

```
n = add3(3, 4, 5);
```

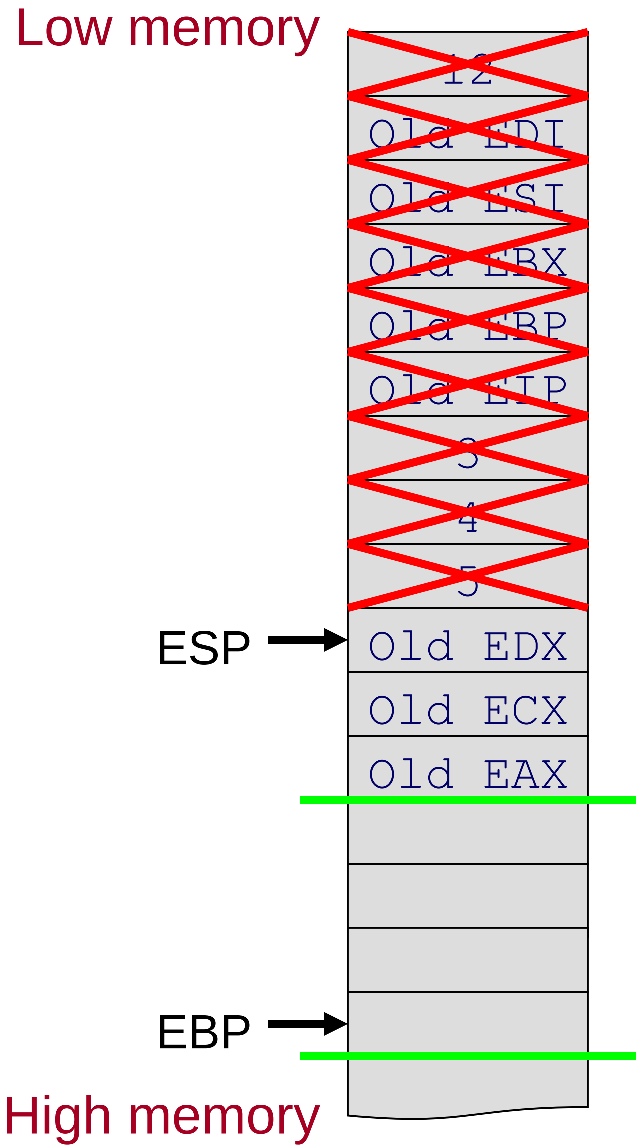
```
# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push arguments
pushl $5
pushl $4
pushl $3
# Call add3
call add3
# Pop arguments
addl $12, %esp
```



Trace of Running Example 15

```
n = add3(3, 4, 5);
```

```
# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push arguments
pushl $5
pushl $4
pushl $3
# Call add3
call add3
# Pop arguments
addl %12, %esp
# Use return value
movl %eax, n
```



Low memory

~~12~~
~~Old EDI~~
~~Old ESI~~
~~Old EBX~~
~~Old EBP~~
~~Old EIP~~
~~3~~
~~4~~
~~5~~
ESP → Old EDX
Old ECX
Old EAX

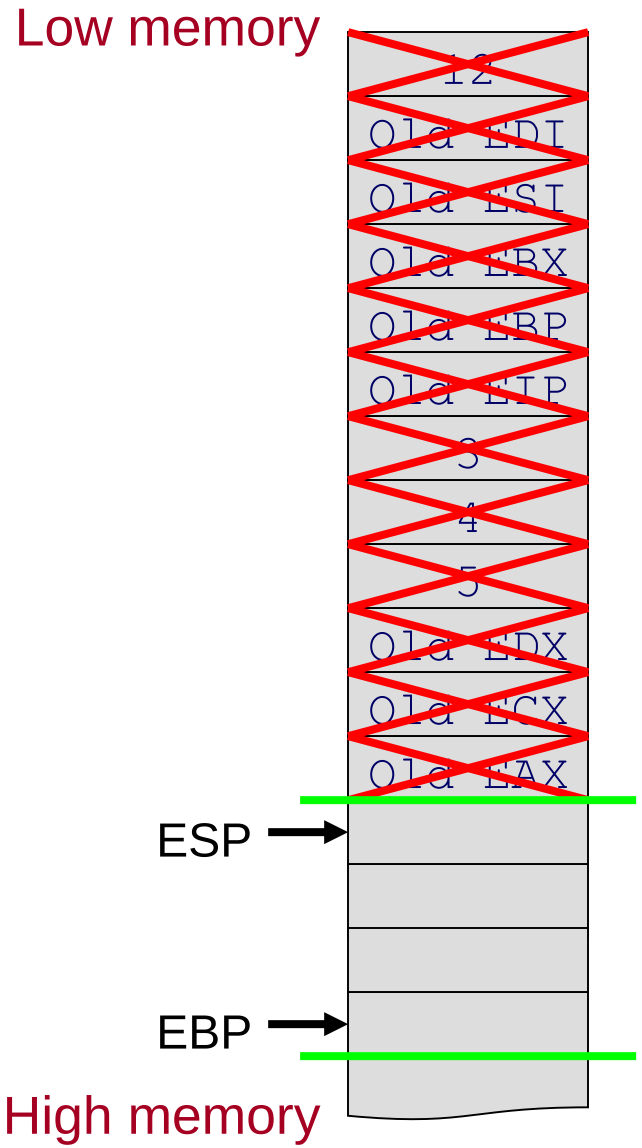
EBP →

High memory

Trace of Running Example 16

```
n = add3(3, 4, 5);
```

```
# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push arguments
pushl $5
pushl $4
pushl $3
# Call add3
call add3
# Pop arguments
addl %12, %esp
# Use return value
movl %eax, n
# Restore caller-save registers if necessary
popl %edx
popl %ecx
popl %eax
```

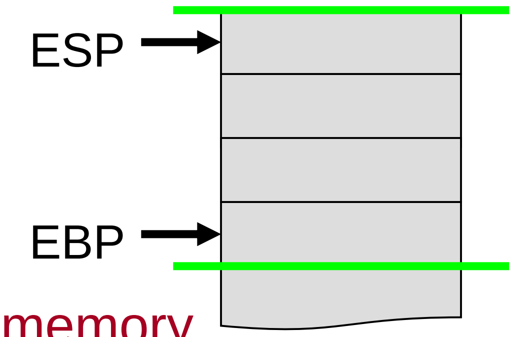


Trace of Running Example 17

`n = add3(3, 4, 5);`

Low memory

```
# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push arguments
pushl $5
pushl $4
pushl $3
# Call add3
call add3
# Pop arguments
addl %12, %esp
# Use return value
movl %eax, n
# Restore caller-save registers if necessary
popl %edx
popl %ecx
popl %eax
# Proceed!
…
```



High memory

Summary

Function calls in IA-32 assembly language

Calling and returning

- **`call`** instruction: push EIP onto stack and jump
- **`ret`** instruction: pop from stack to EIP

Passing arguments

- Caller pushes onto stack
- Callee accesses as positive offsets from EBP
- Caller pops from stack

Summary (cont.)

Storing local variables

- Callee pushes onto stack
- Callee accesses as negative offsets from EBP
- Callee pops from stack

Handling registers

- Caller saves and restores EAX, ECX, EDX if necessary
- Callee saves and restores EBX, ESI, EDI if necessary

Returning values

- Callee places data of integer types and addresses in EAX