The C - Assembly connection

Systems Software

Intel IA32 Architecture

- **Often referred to as “x86”**
  - Started with Intel 80386 in 1985 … then 80486, Pentium, Intel Core, Intel Core 2, ---
  - The most popular ISA by far

- **Has a large and varied set of instructions**
  - Fortunately we won’t need to use everything!

Compiling into assembly

- **C Code**

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Generated IA32 Assembly

```
_sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

Obtain with command

```
gcc -O -S code.c
```

Produces file `code.s`

Disassembling Object Code

```
00401040 <_sum>:
   0:      55                push   %ebp
   1:      89 e5             mov    %esp,%ebp
   3:      8b 45 0c          mov    0xc(%ebp),%eax
   6:      03 45 08          add    0x8(%ebp),%eax
   9:      89 ec             mov    %ebp,%esp
   b:      5d                pop    %ebp
   c:      c3                ret
```

- **Disassembler: `objdump -d code.o`**
  - Produces interpretation of assembly code
  - Can be run on either `a.out` (complete executable) or `.o` file

Alternate Disassembly

Object

```
0x401040:
      0x55
      0x89
      0xe5
      0x8b
      0x45
      0x0c
      0x03
      0x45
      0x08
      0x89
      0xec
      0x5d
      0xc3
```

Disassembled

```
0x401040 <sum>:      push    %ebp
0x401041 <sum+1>:    mov     %esp,%ebp
0x401043 <sum+3>:    mov     0xc(%ebp),%eax
0x401046 <sum+6>:    add     0x8(%ebp),%eax
0x401049 <sum+9>:    mov     %ebp,%esp
0x40104b <sum+11>:   pop     %ebp
0x40104c <sum+12>:   ret
```

- **Within gdb Debugger**

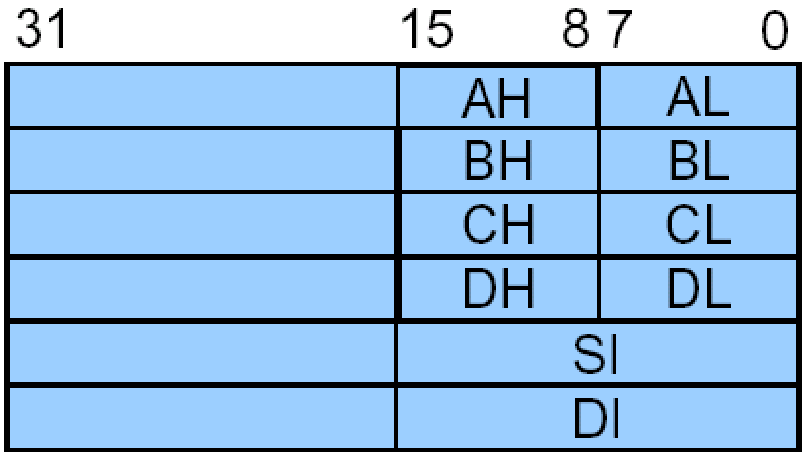
  ```
  gdb code
  disassemble sum
  ```

  - Disassemble procedure

  ```
  x/13b sum
  ```

  - Examine the 13 bytes starting at `sum`

Some x86 registers



General-purpose registers

16-bit	32-bit	64-bit
AX	EAX	RAX
BX	EBX	RBX
CX	ECX	RCX
DX	EDX	RDX
	ESI	…
	EDI	

GNU Assembly Syntax

GDB: GNU Debugger
GCC: GNU Compiler Collection
GAS: GNU Assembler

source first

Meaning	GAS
ebx := eax	movl %eax, %ebx
eax := eax + ebx	addl %ebx, %eax
ecx := ecx << 2	shl $2, %ecx

- Referring to a register: percent sign ("%")
  - E.g., "%ecx"
- Referring to a constant: dollar sign ("$")
  - E.g., "$1" for the number 1

Addressing modes

- **Most instructions have several ways of addressing source and destination operands**
  - Inputs can be registers, memory location, or immediate (constant) values
  - Outputs can be saved to registers or memory locations

- **Example: “movl” instruction (copy 32-bit values) supports…**

Immediate to register	movl $0x1000, %eax
Register to Register	movl %eax, %ebx
Memory to register (a.k.a. “load”)	movl (%eax), %ebx
Register to memory (a.k.a. “store”)	movl %eax, (%ebx)

Memory references

Addresses are indicated by operands that have a paren “( )”

What does mov (%al), %dl do?

Moves 0xcc into dl

	Addr
0xff	6
0xee	
0xdd	
0xcc	
0xbb	
0xaa	
0x00	0

Register	Value
eax	0x3
edx	0x0
ebx	0x5

Memory references

Addresses are indicated by operands that have a paren "( )"

What does mov (%eax), %edx do?

Register	Value
eax	0x3
edx	0xcc
ebx	0x5

Which 4 bytes get moved, and which is the LSB in edx?

	Addr
0xff	6
0xee	
0xdd	
0xcc	
0xbb	
0xaa	
0x00	0

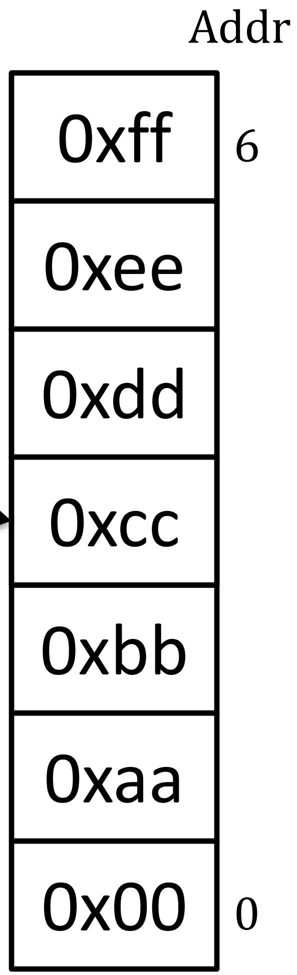
mov (%eax), %edx

EDX

Register	Value
eax	0x3
edx	0xcc
ebx	0x5

Bit 0

EDX = 0xffeeddcc!



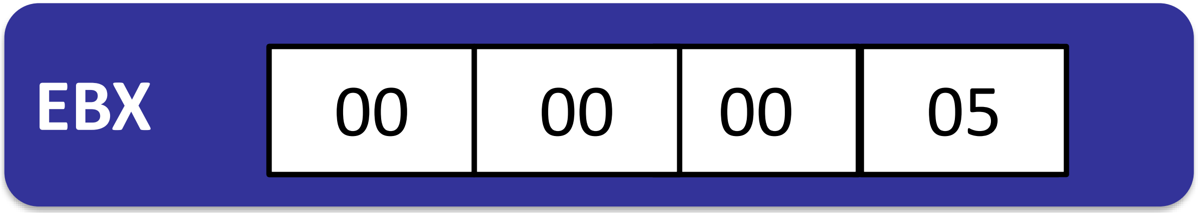
	Addr
0xff	6
0xee	
0xdd	
0xcc	
0xbb	
0xaa	
0x00	0

Little Endian: Least significant byte first

… so …

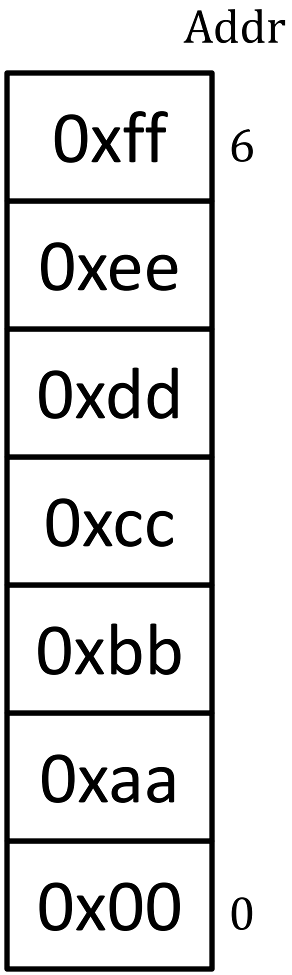
address *a* goes in the least significant byte (the littlest bit) *a+1* goes into the next byte, and so on.

mov %ebx, (%eax)



Bit 0

Register	Value
eax	0x3
edx	0xcc
ebx	0x5



Little Endian: Least significant byte first

... so ...

address *a* goes in the least significant byte (the littlest bit) *a+1* goes into the next byte, and so on.

Program Memory Layout

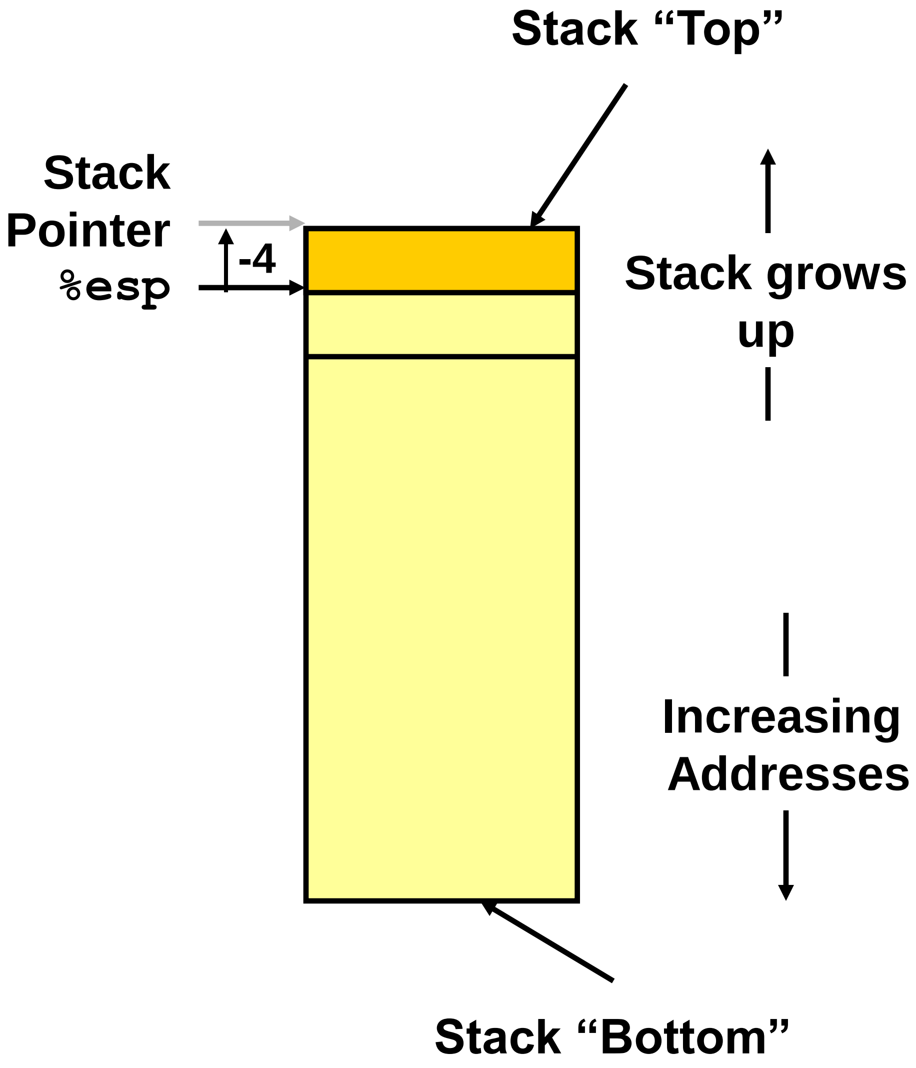
- Text region
  - Program’s executable code
- Heap
  - Dynamically allocated data
- Stack
  - Stores all temporary data related to each function call
    - Return address
    - Arguments
    - Local variables of function

Registers for Executing the Code

- Instruction pointer (EIP)
  - Address in memory of the next instruction
- Interesting pointers to the stack
  - Stack register (ESP)
    - Address of the top of the stack
  - Base pointer (EBP)
    - Used for relative references to local variables and arguments

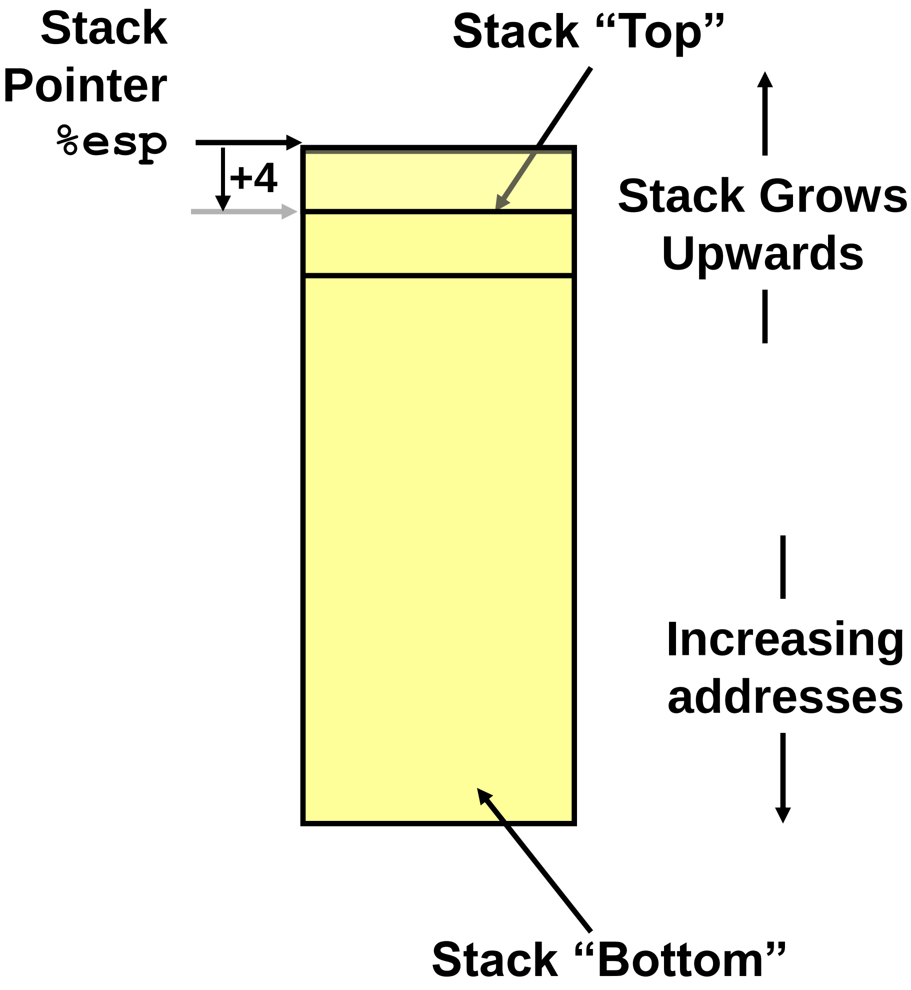
Push Operation

- Push instruction
  - pushl *Src*
- Fetch operand at *Src*
- **Decrement** %esp by 4
- Write operand at address given by %esp



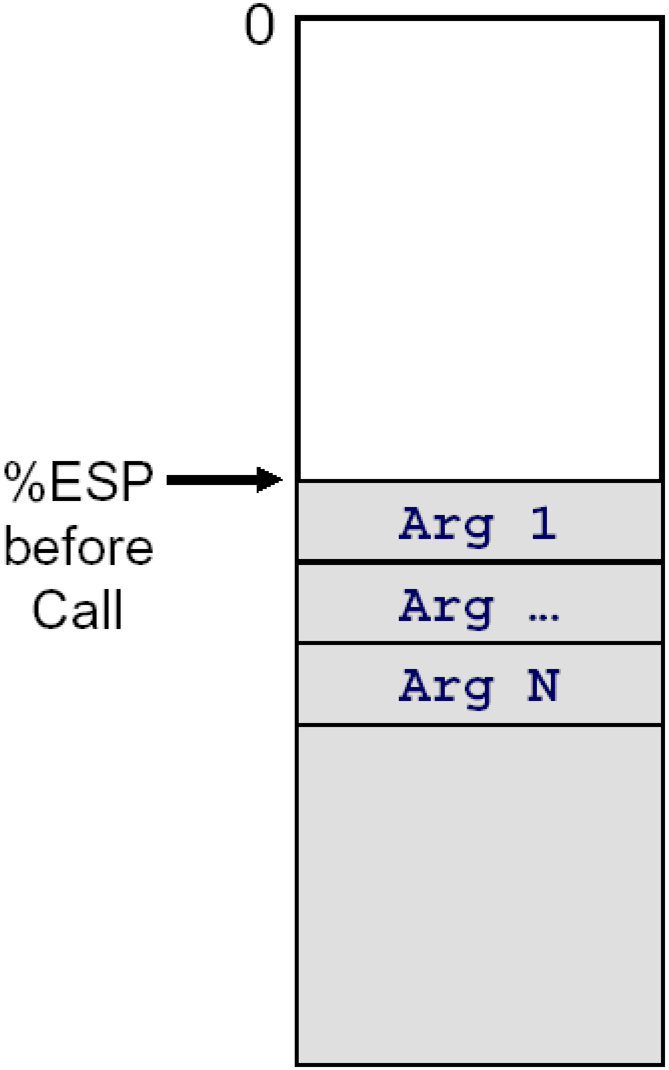
Pop Operation

- Popping
  - popl *Dest*
  - Read operand at address given by %esp
  - **Increment** %esp by 4
  - Write to *Dest*



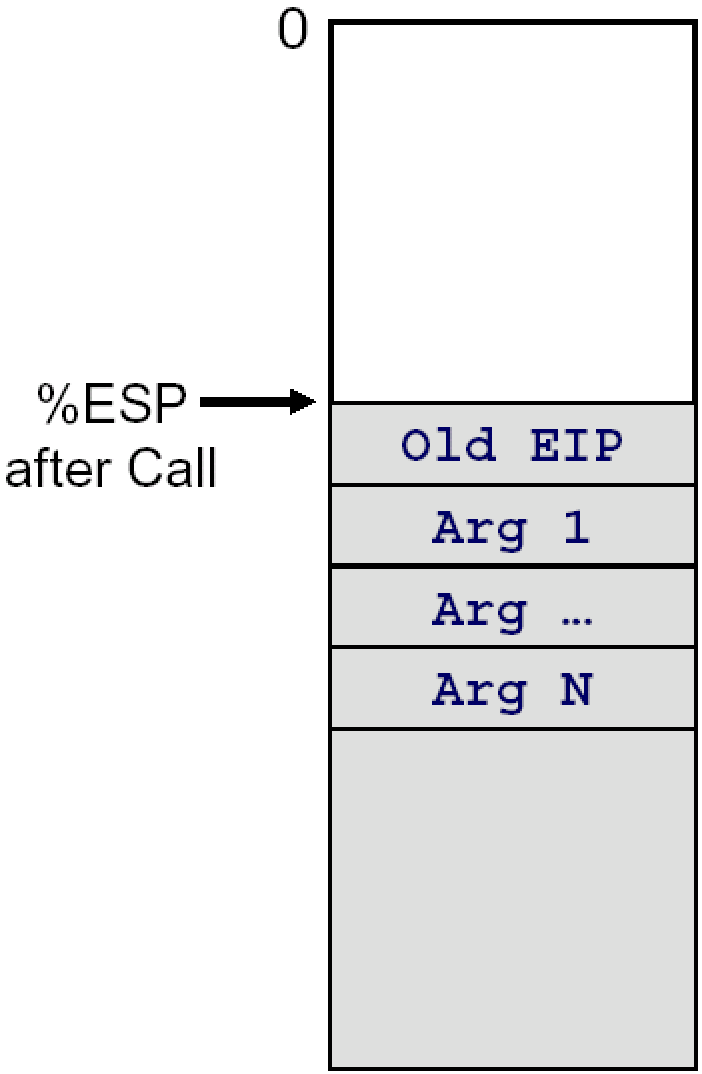
Input parameters

- Caller pushes input parameters before executing call instruction
- Parameters are pushed in reverse order
  - – So that the first argument is at the top of the stack at the time of the call
- When done, Caller invokes the function with the “call”



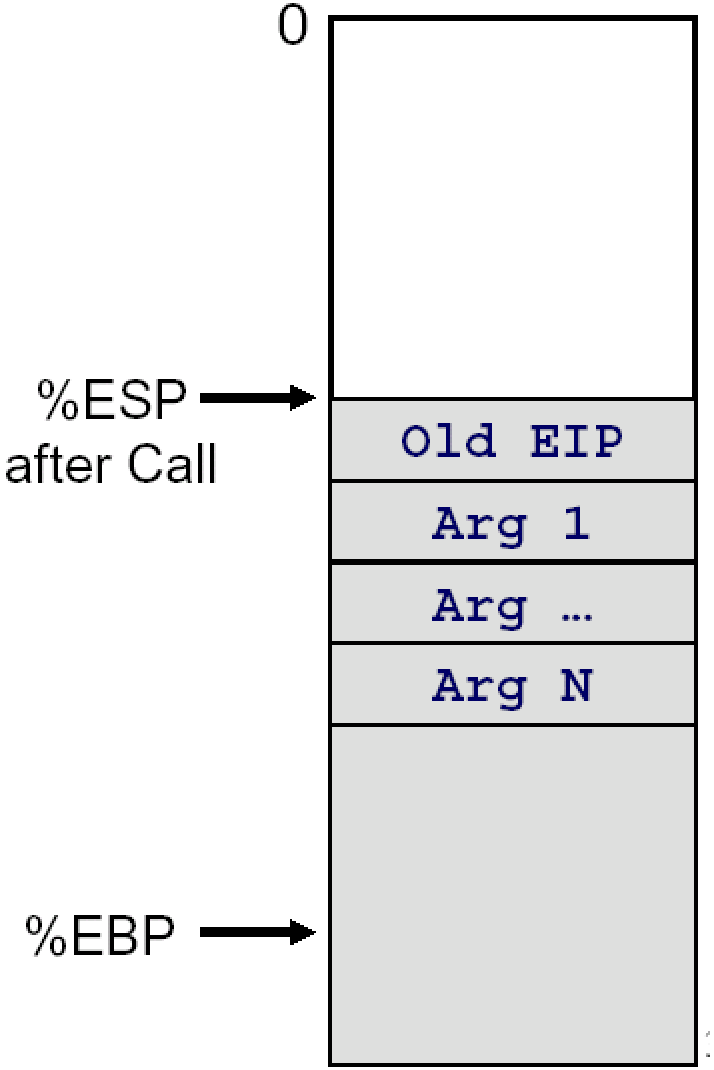
Input parameters

- Call instruction pushes %eip onto stack (return address)
- Call instruction causes execution to shift to the callee
- Callee can address arguments relative to ESP: Arg1 as 4(%esp)



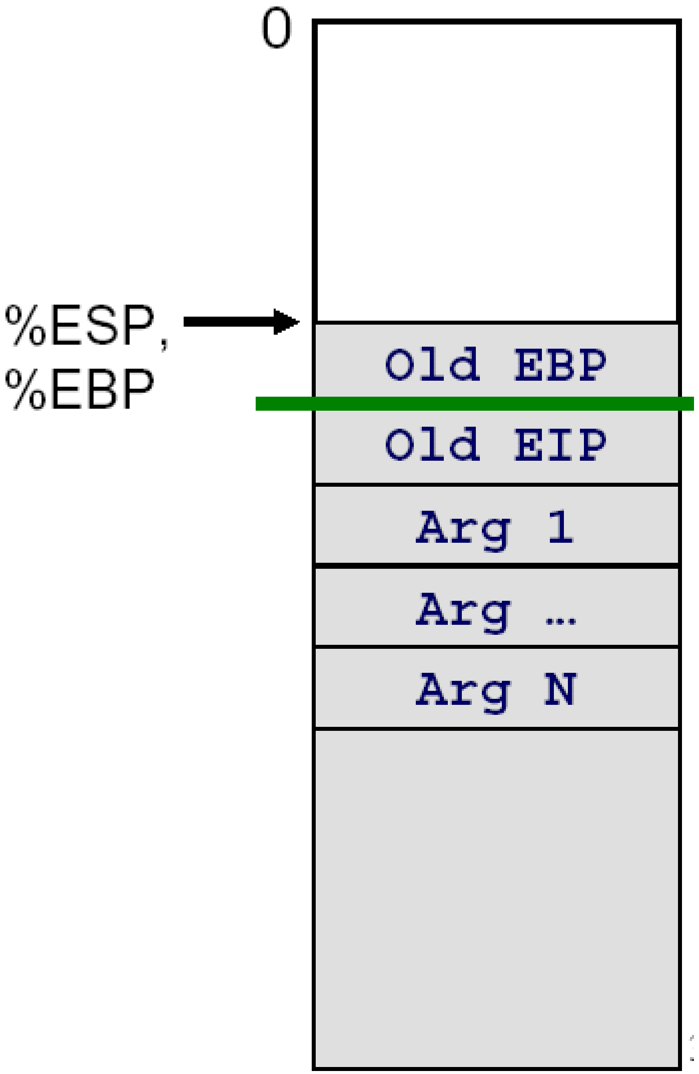
Base Pointer: EBP

- As callee executes, ESP may change
- Use EBP as a fixed reference point to access arguments and other local variables
- Need to save caller’s value of EBP before using EBP



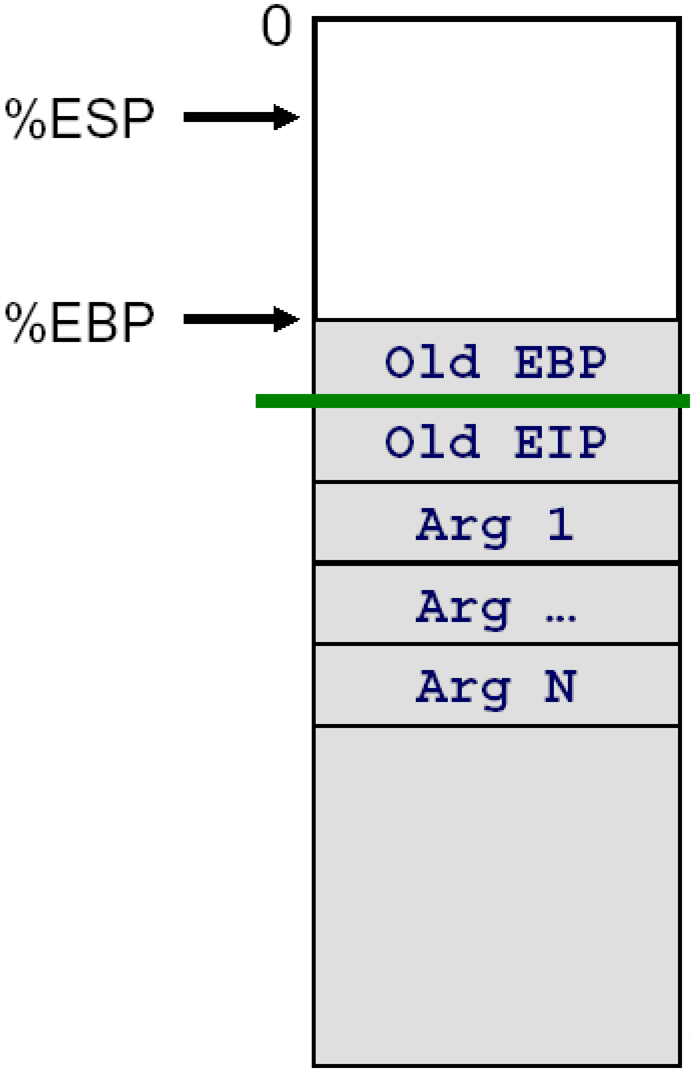
Base Pointer: EBP

- Save caller's base pointer
  - pushl %ebp
- Set value of EBP for callee's use
  - Current value of stack pointer used as callee's base pointer
  - movl %esp, %ebp



Base Pointer: EBP

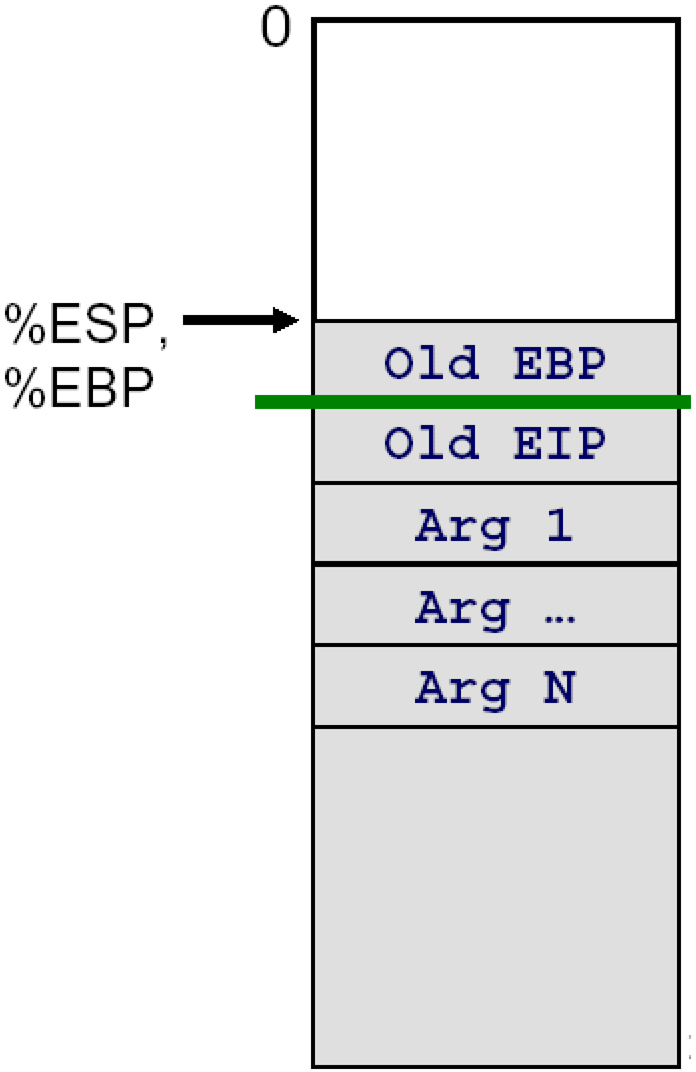
- As Callee executes, ESP may change
- Regardless of ESP, Callee can address Arg1 as 8(%ebp)



Base Pointer: EBP

- Before returning, Callee must restore EBP to its old value
- Executes

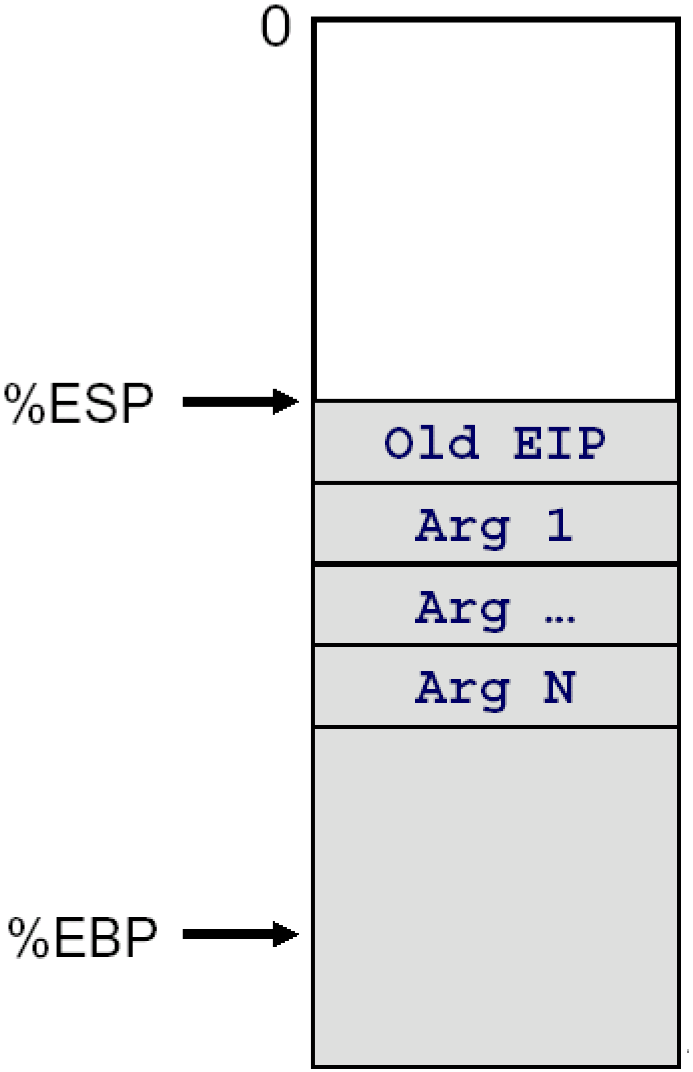
```
movl %ebp, %esp
popl %ebp
ret
```



Base Pointer: EBP

- Before returning, Callee must restore EBP to its old value
- Executes

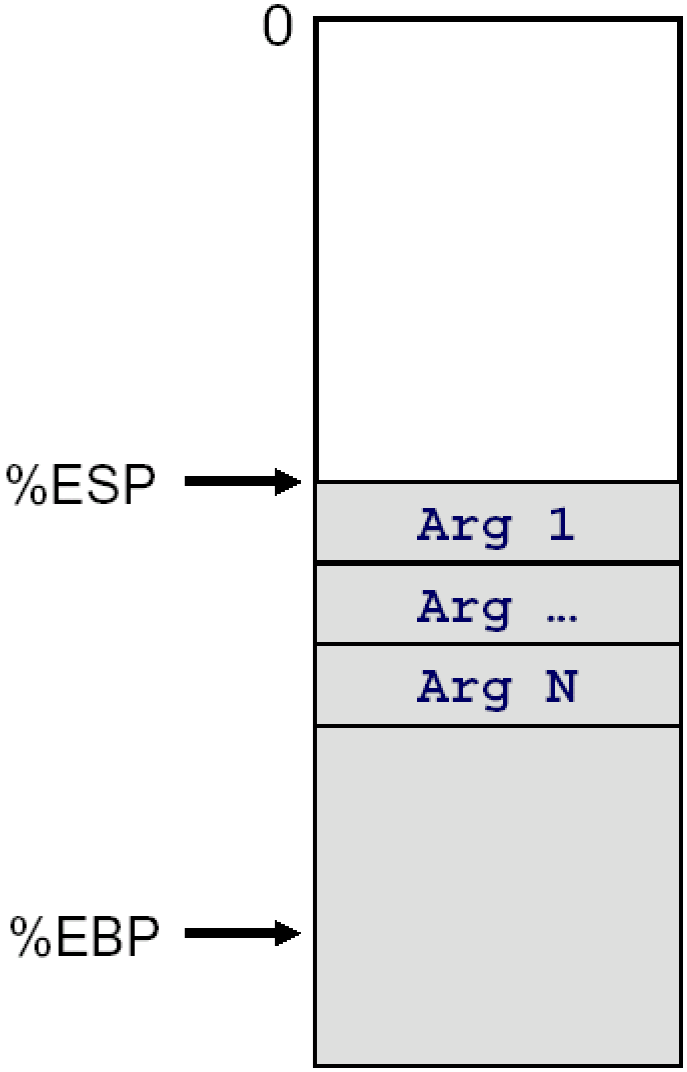
```
movl %ebp, %esp
popl %ebp
→ ret
```



Base Pointer: EBP

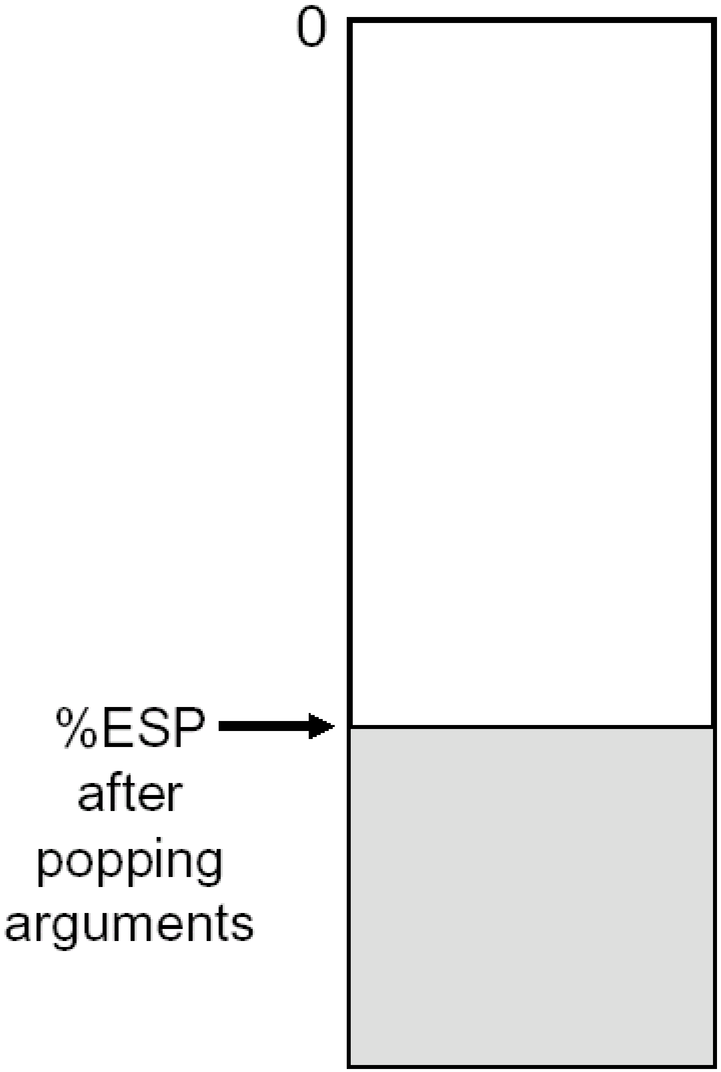
- Before returning, Callee must restore EBP to its old value
- Executes

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



Input parameters

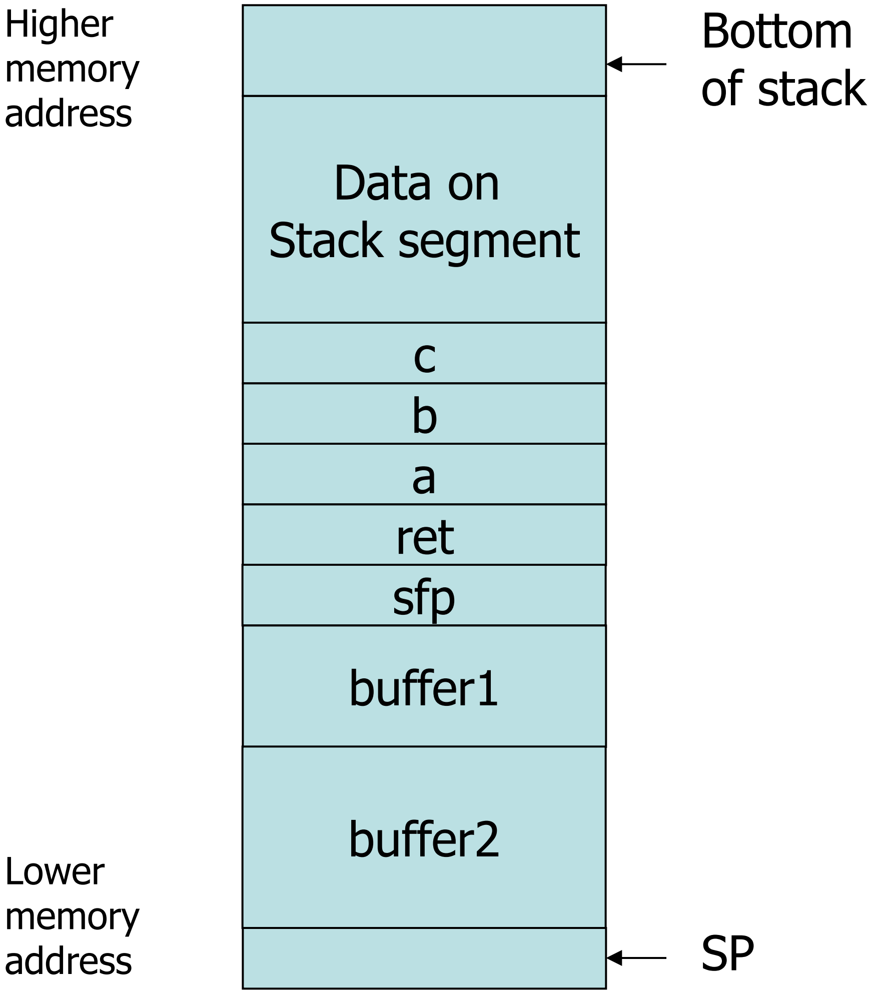
- After the function call is finished, the caller pops the pushed arguments from the stack



Key Point

- Function arguments have fixed positive offsets relative to frame pointer
  - +8(%ebp)
  - +12(%ebp)
- Local variable have fixed negative offsets relative to frame pointer
  - -4(%ebp)
  - -8(%ebp)

Example



Higher memory address

Bottom of stack

Data on Stack segment

c

b

a

ret

sfp

buffer1

buffer2

Lower memory address

SP

Example1.c

example.s

Teaser Problem

Practice problem 3.14 from Bryant

You will see this code all over if you look at library routines

```
     call next
   next:
      popl %eax
```

- **What value does `%eax` end up with?**
- **Why is there no `ret` to match the call?**
- **What useful purpose might this code serve?**

Summary

- Invoking a function
  - Call: call the function
  - Ret: return from the instruction
- Stack Frame for a function invocation includes
  - Return address
  - Procedure arguments
  - Local variables
- Base pointer EBP
  - Fixed reference point in the Stack Frame
  - Used for referencing arguments and local variables