Procedures

Procedures

A *Procedure* is a unit of code that we can call

- Depending on the programming language, it may be called a procedure, function, subroutine, or method
- A call is like a jump, except it can return.

The hardware provides machine instructions for this:

`call` *`label`*

Push return address on stack; Jump to *`label`*

`ret`

Pop address from stack; Jump to address

First of all, we have to understand how a stack works…

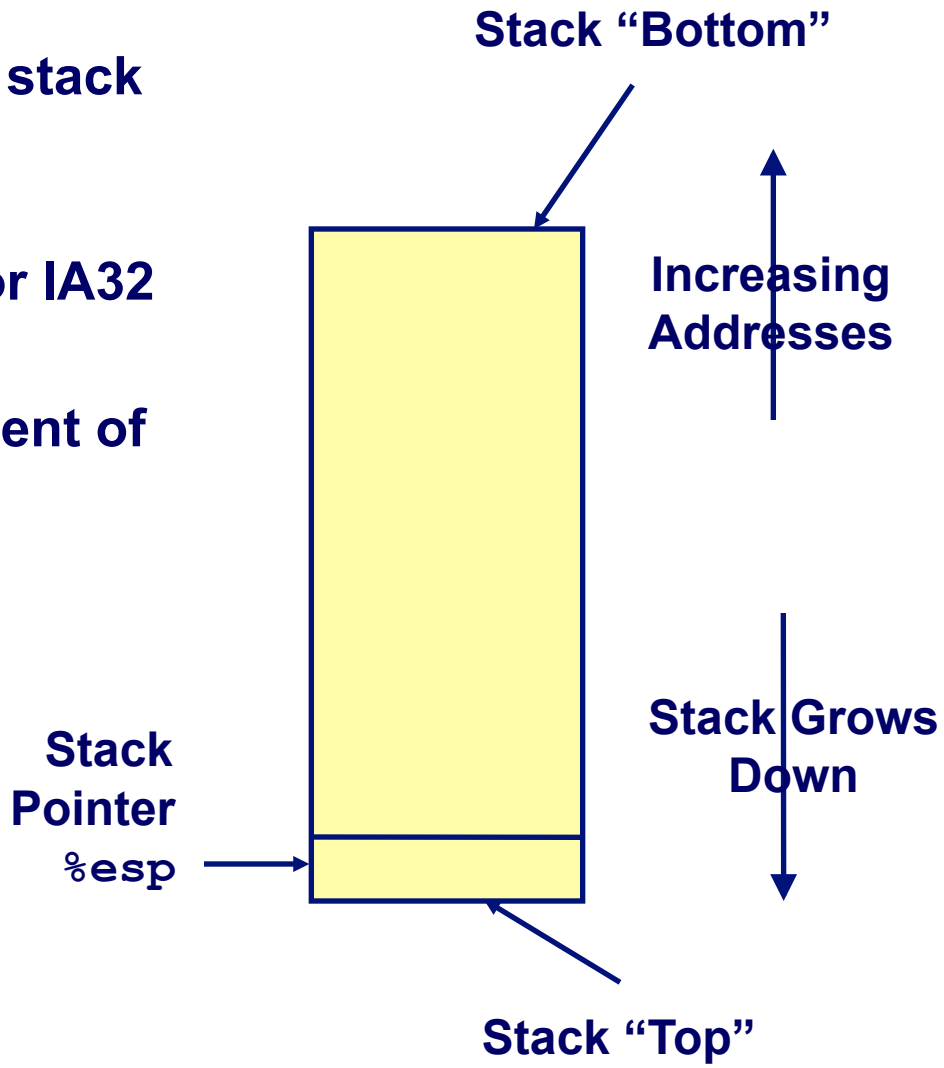
IA32 Stack

Region of memory managed with stack discipline

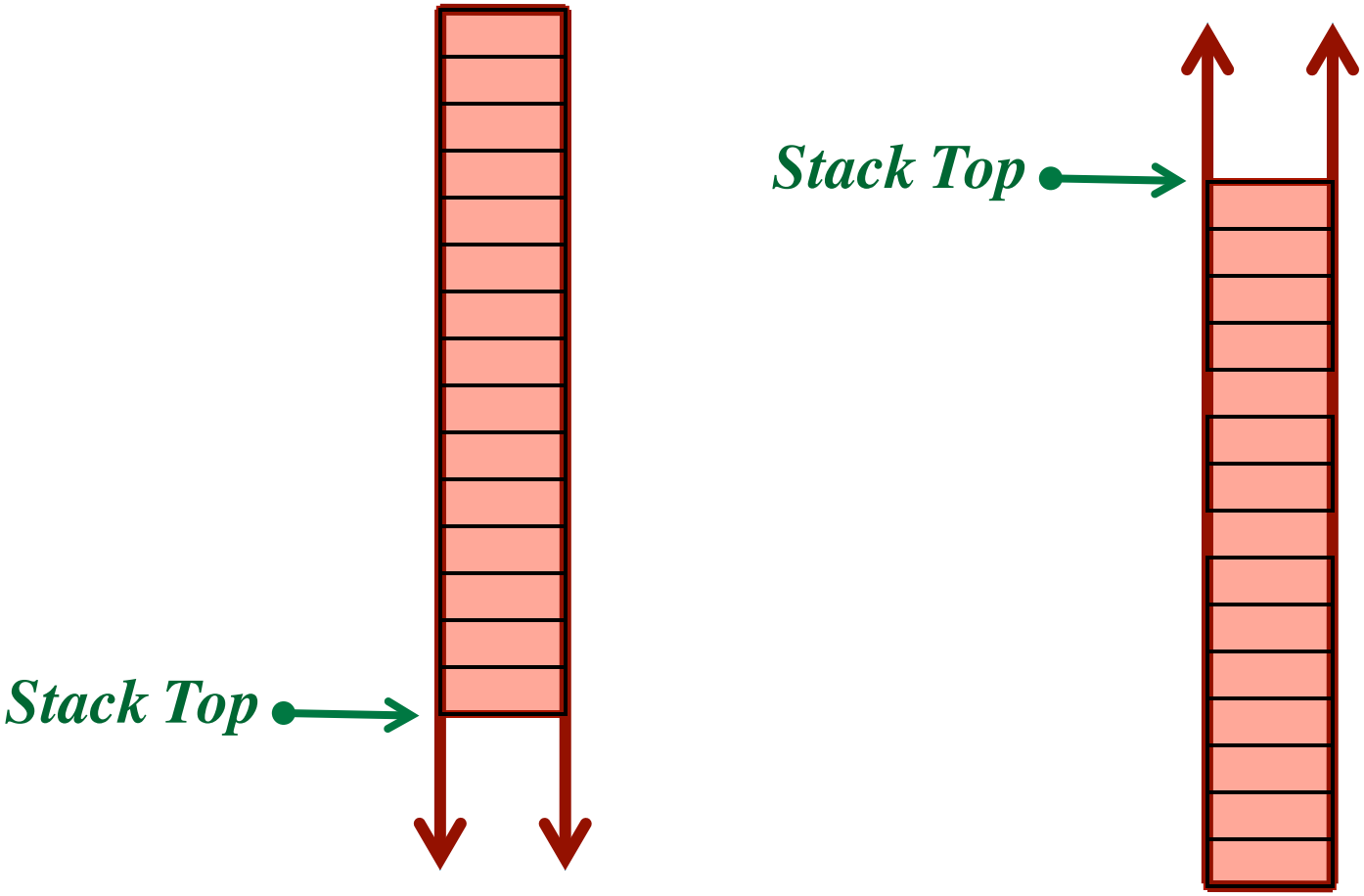
Grows toward lower addresses for IA32

Register %esp indicates top element of stack

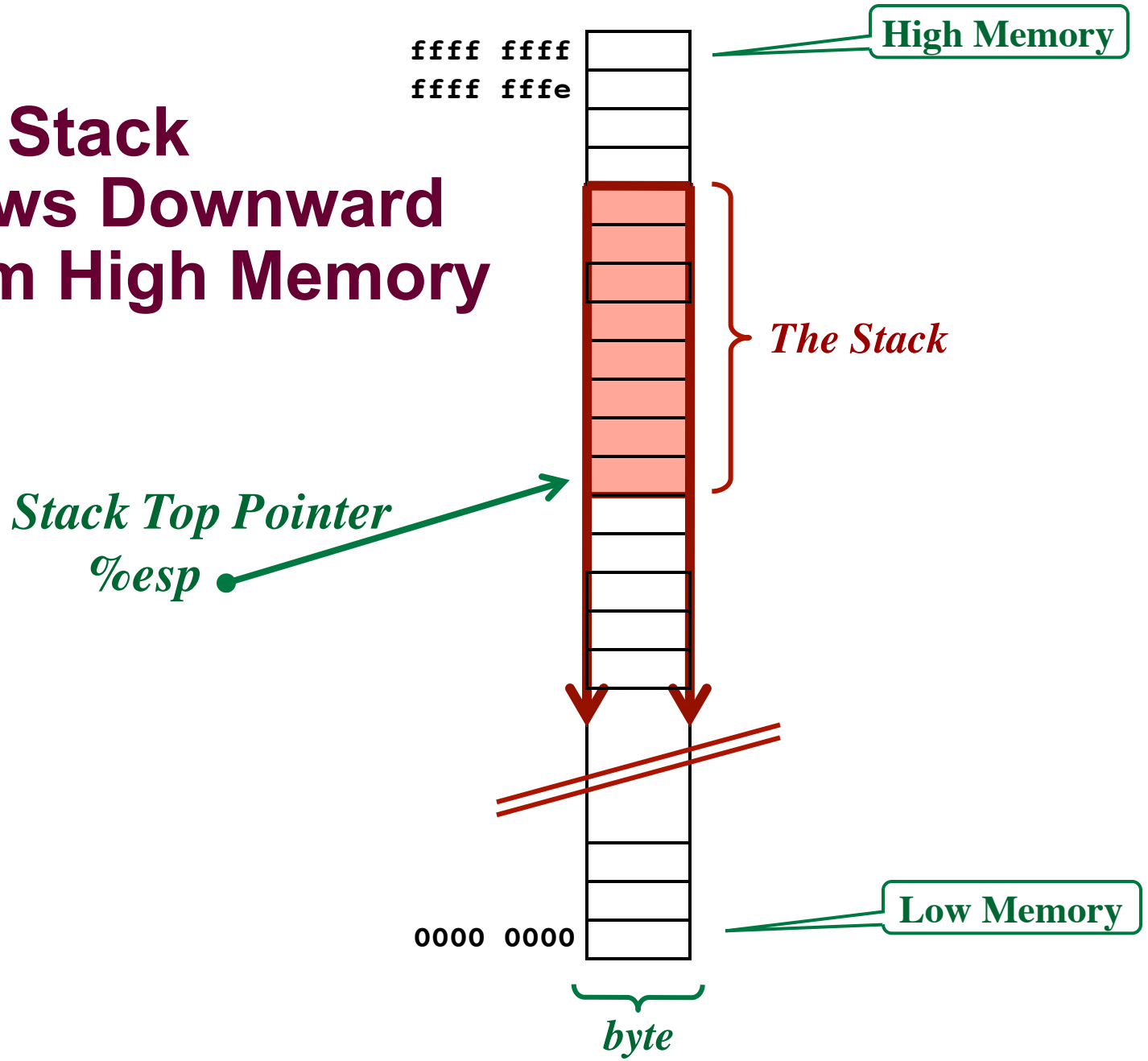
Top element has lowest address



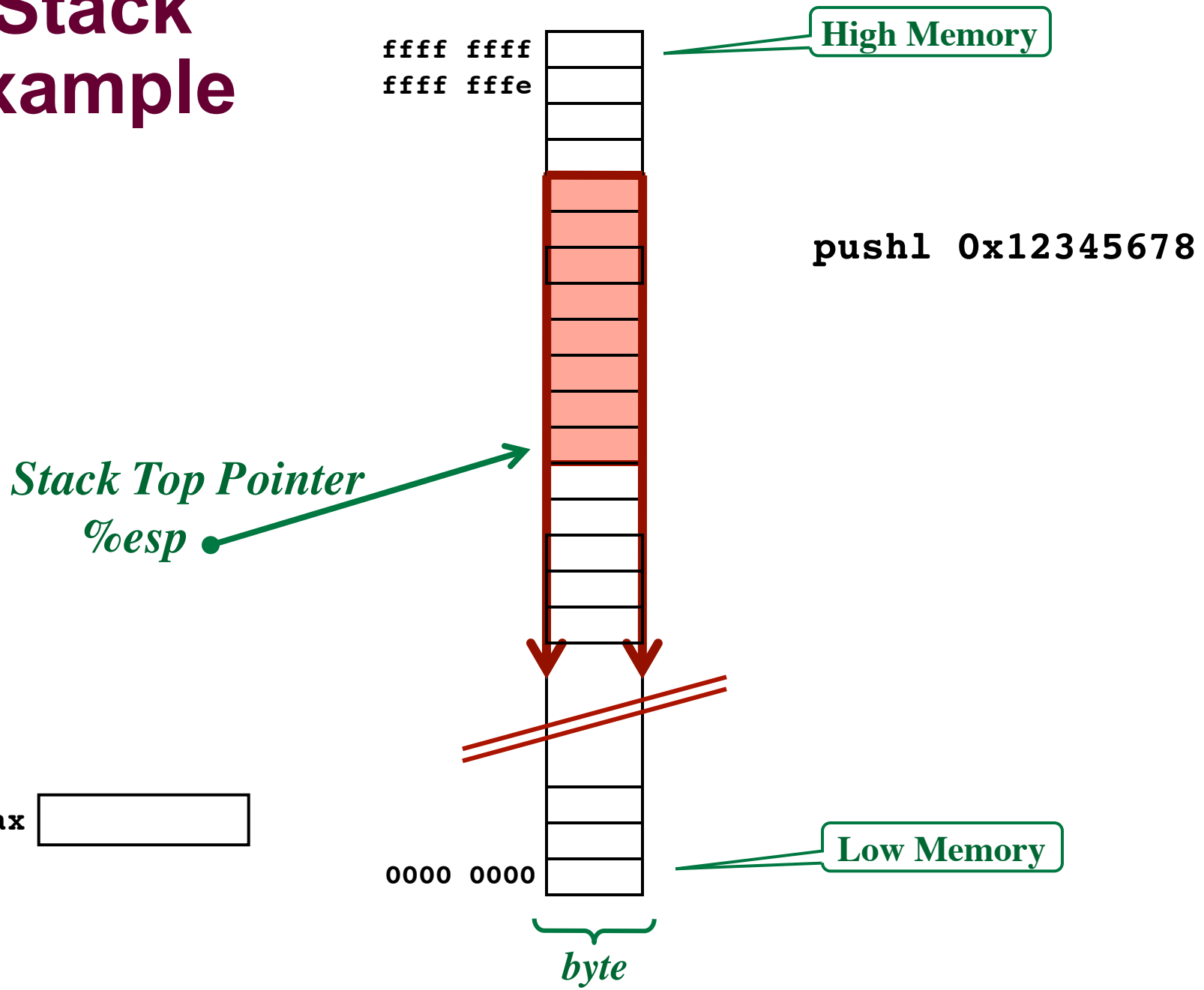
How shall we imagine a stack?



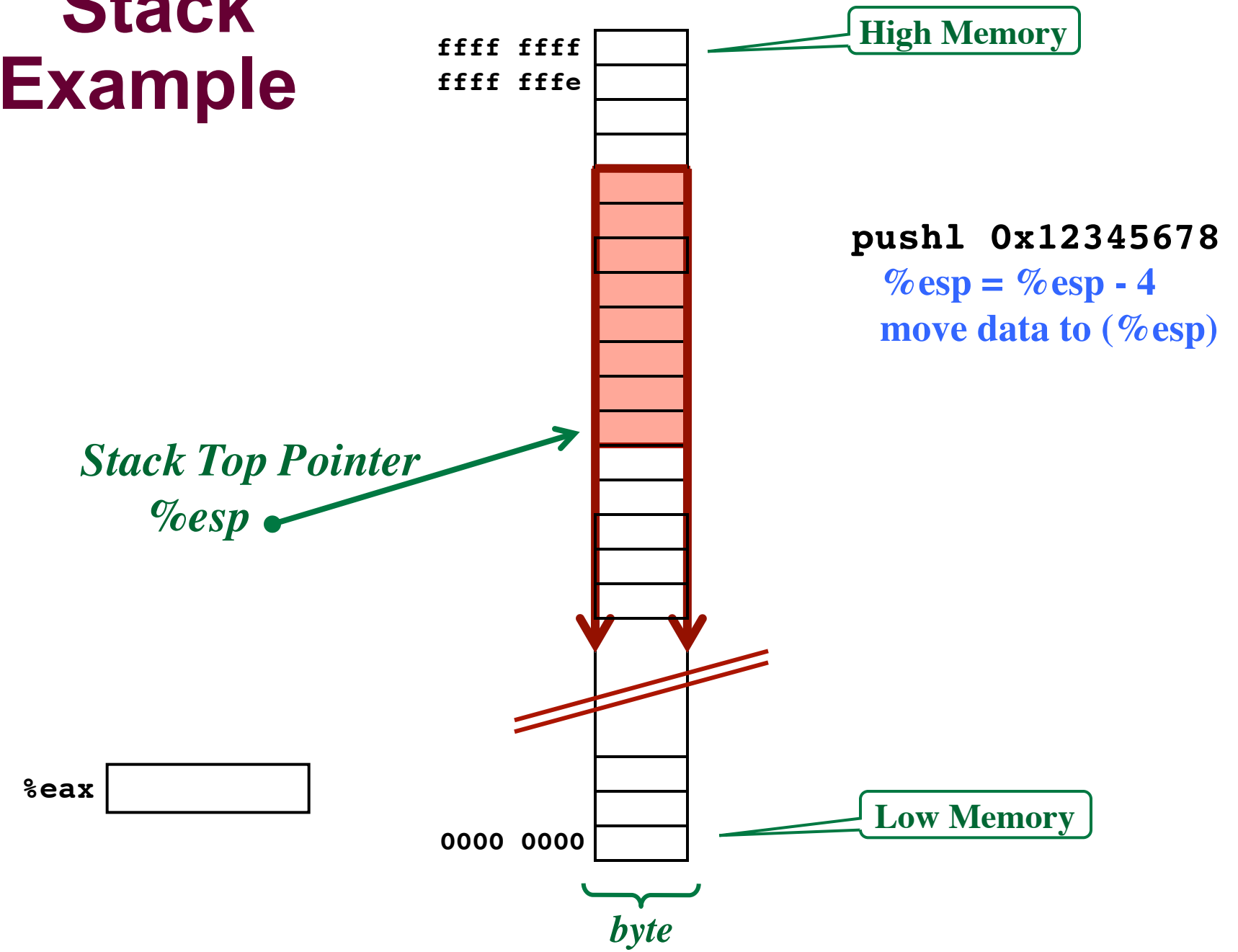
The Stack Grows Downward From High Memory



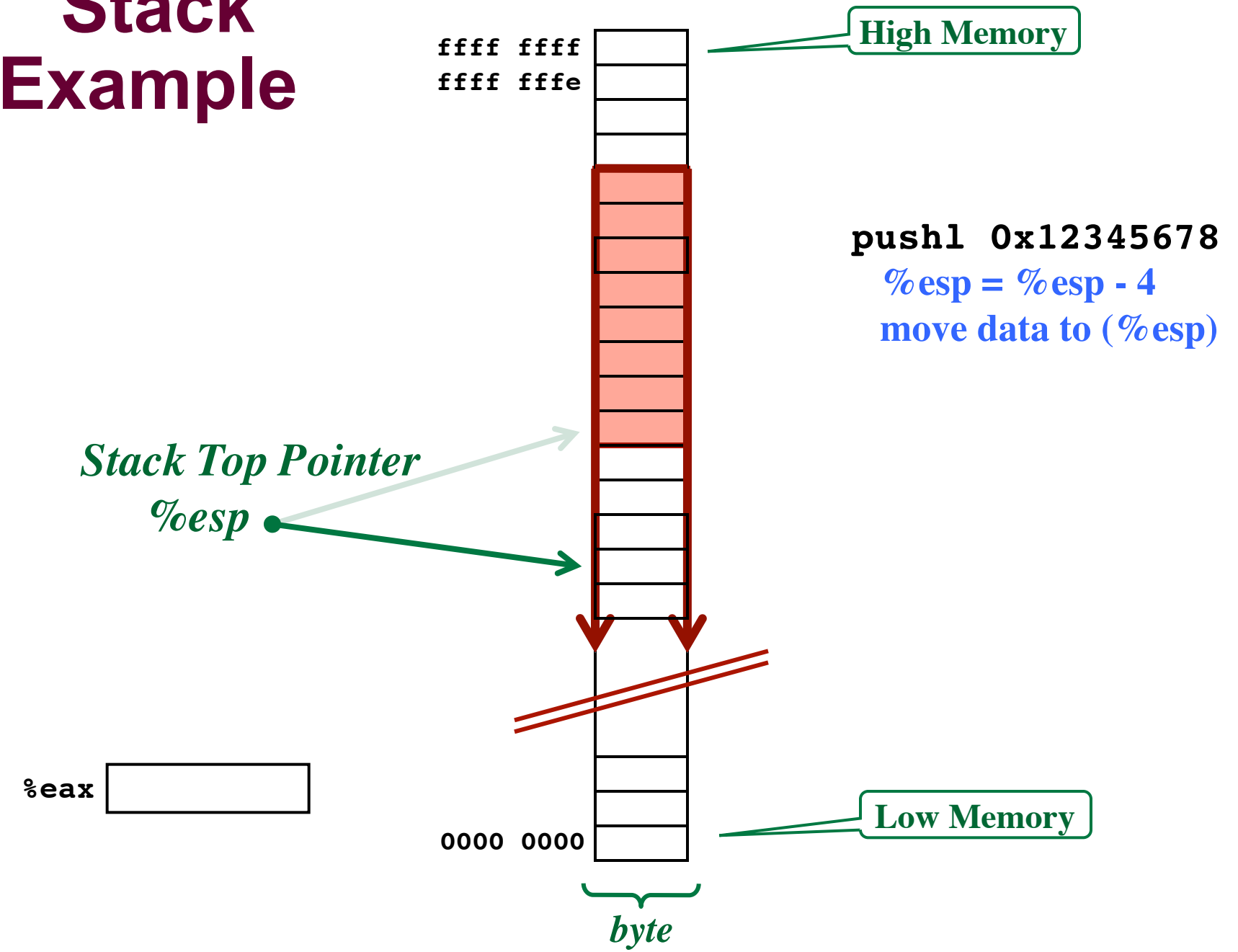
Stack Example



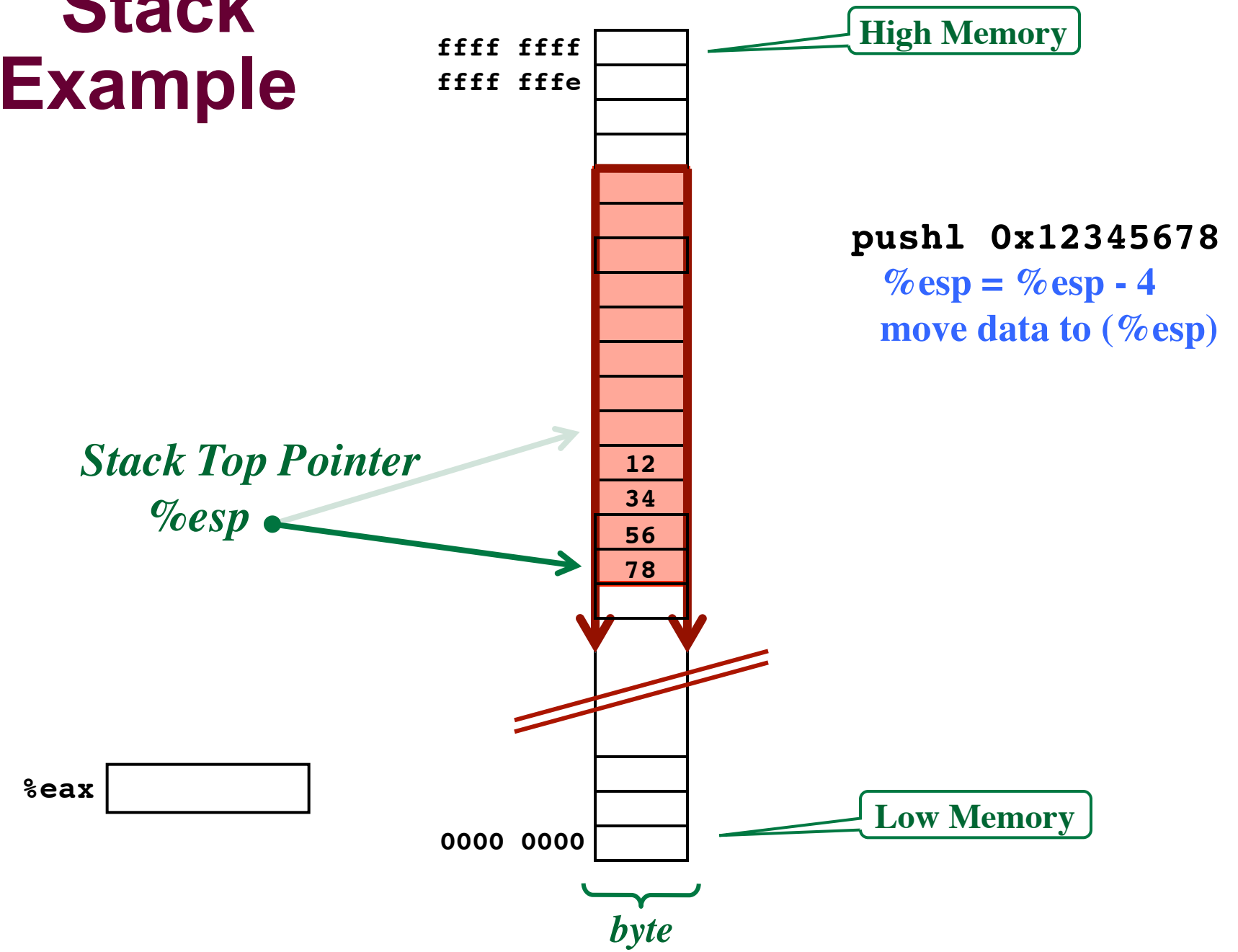
Stack Example



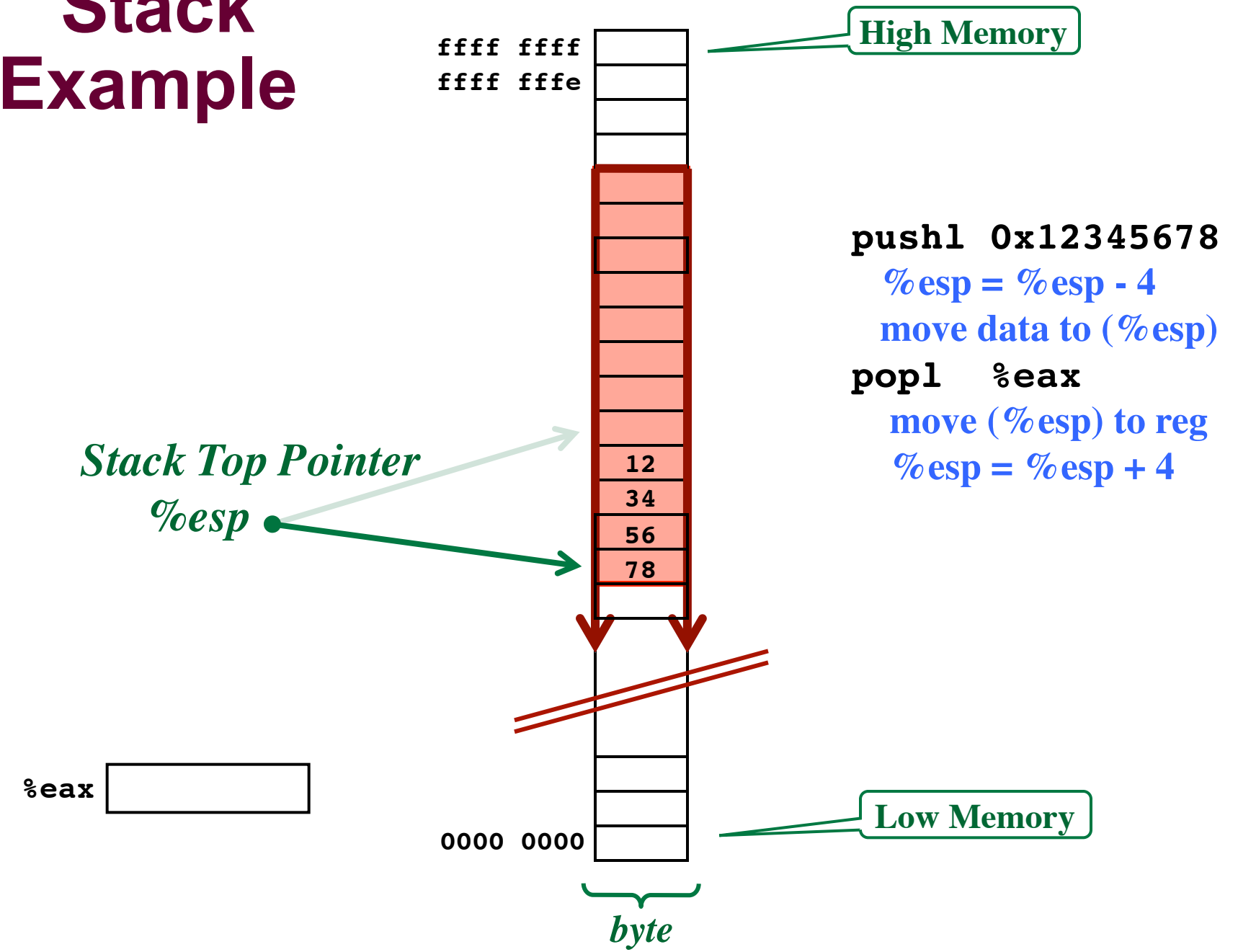
Stack Example



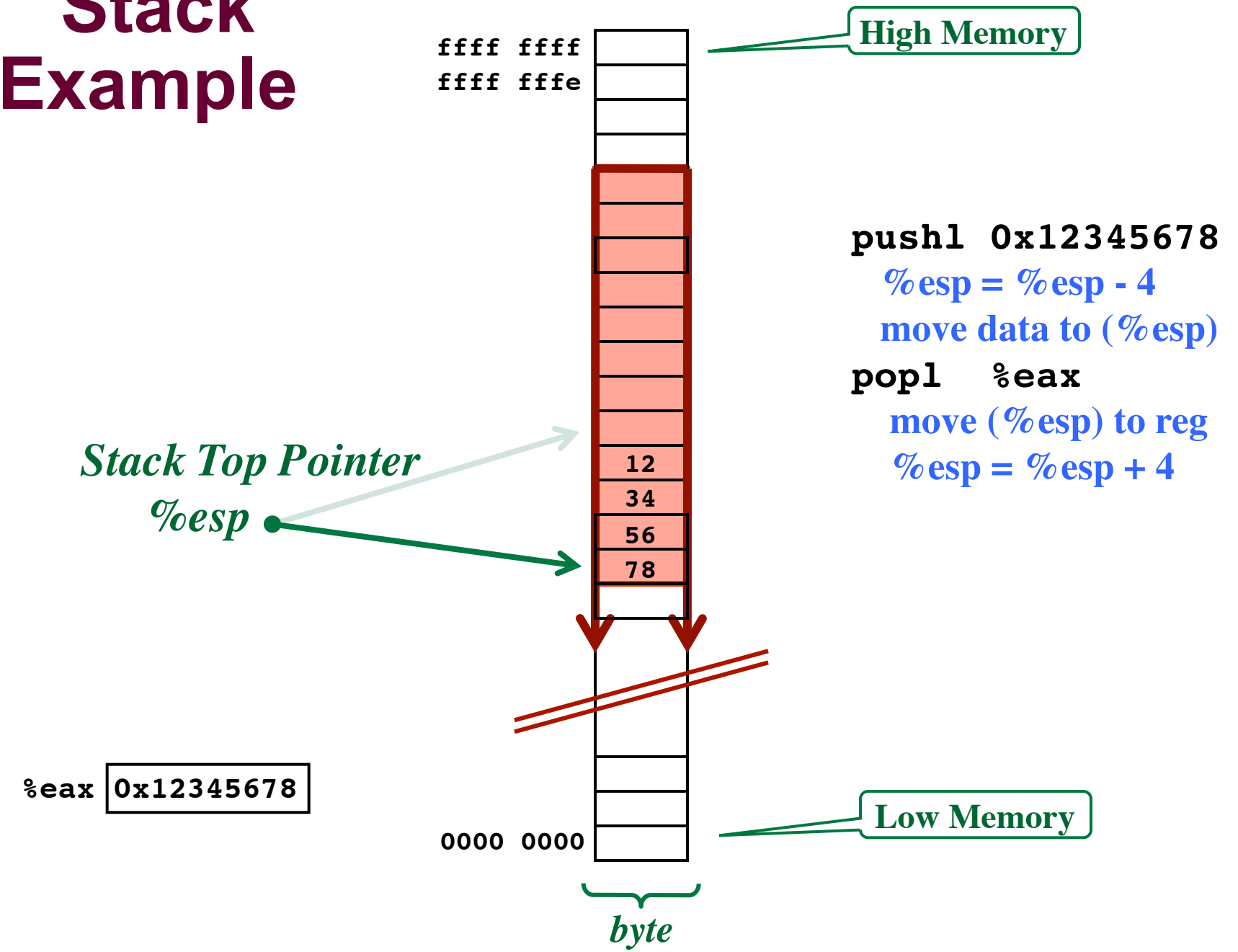
Stack Example



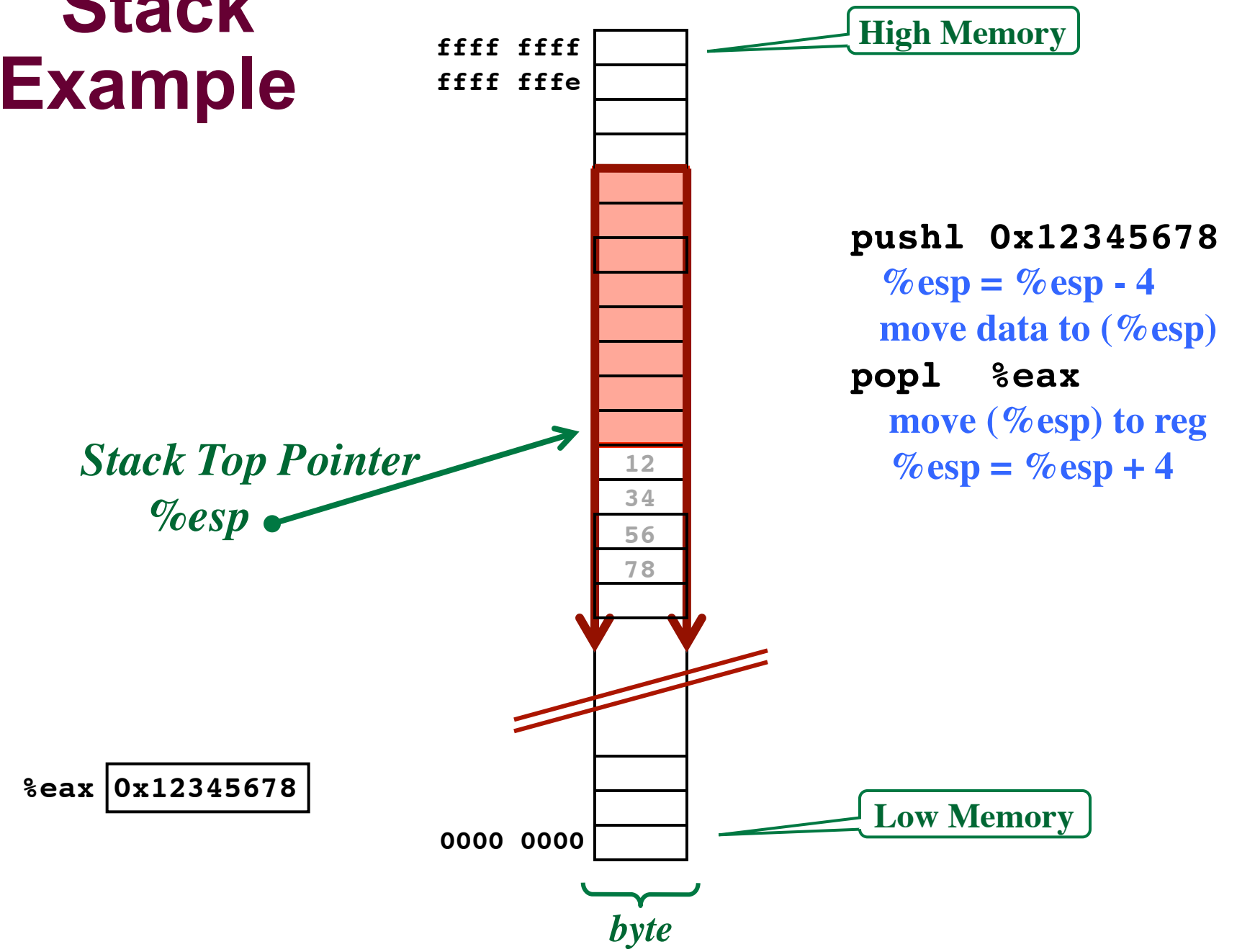
Stack Example



Stack Example



Stack Example



Stack instructions (push and pop)

Stack manipulation is just data movement that updates the stack pointer register

- **Move data onto the stack (pushl)**
- **Move data off of the stack (popl)**

The stack is essential for implementing function calls

- **Function parameters**
- **Return address**
- **Prior stack frame information**
- **Local function variables**

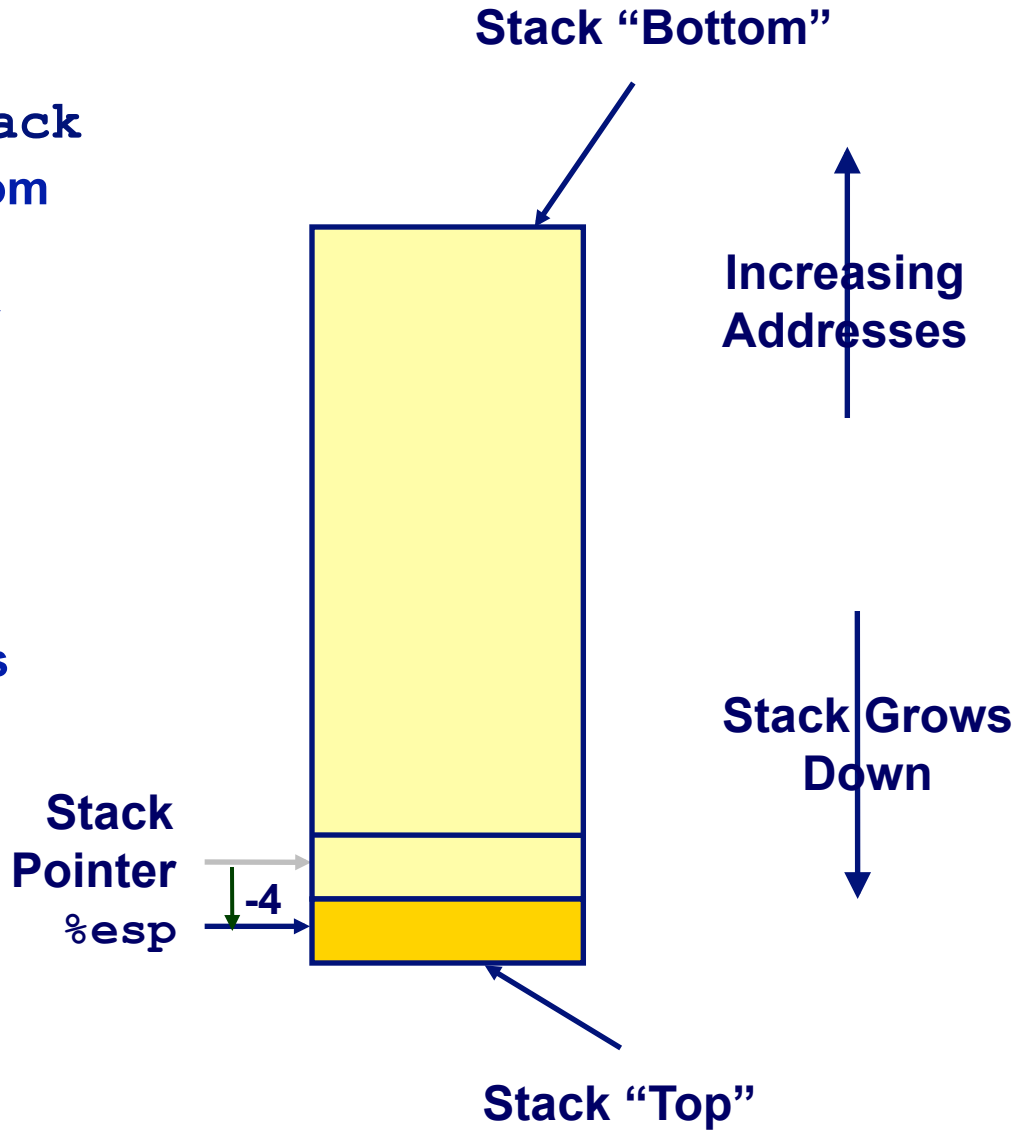
Instruction		Effect	Description
`pushl`	`S`	R[%esp] ← R[%esp] – 4; M[R[%esp]] ← S	Push
`popl`	`D`	D ← M[R[%esp]]; R[%esp] ← R[%esp] + 4	Pop

IA32 Stack Pushing

Pushing

- `Pushing data onto stack`
  - Subtract the data size from the stack pointer (%esp)
  - Move data into the newly created space
- `pushl` *Src*
  - Fetch operand at *Src*
  - Decrement `%esp` by 4
  - Write operand at address given by `%esp`
- *e.g.* `pushl %eax`

```
subl $4, %esp
movl %eax,(%esp)
```

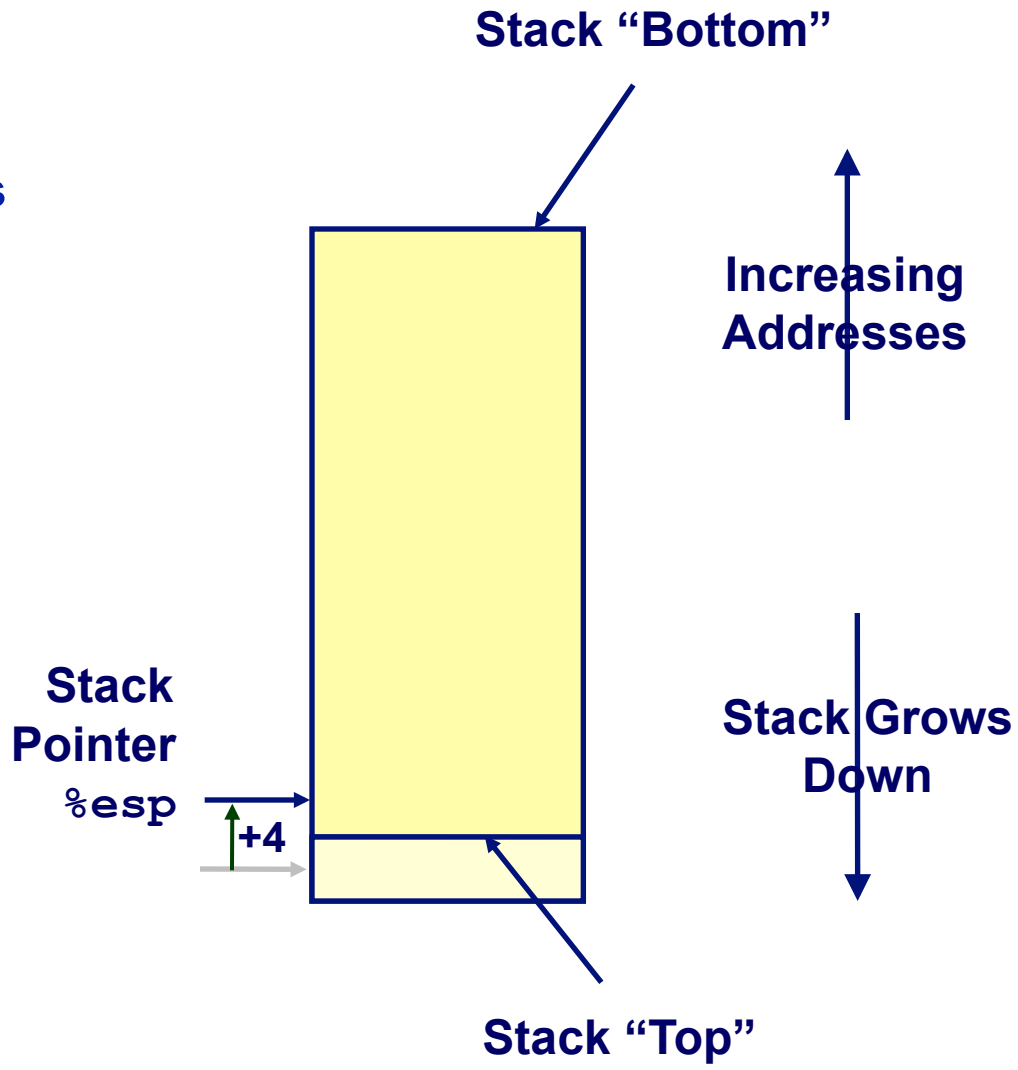


IA32 Stack Popping

Popping

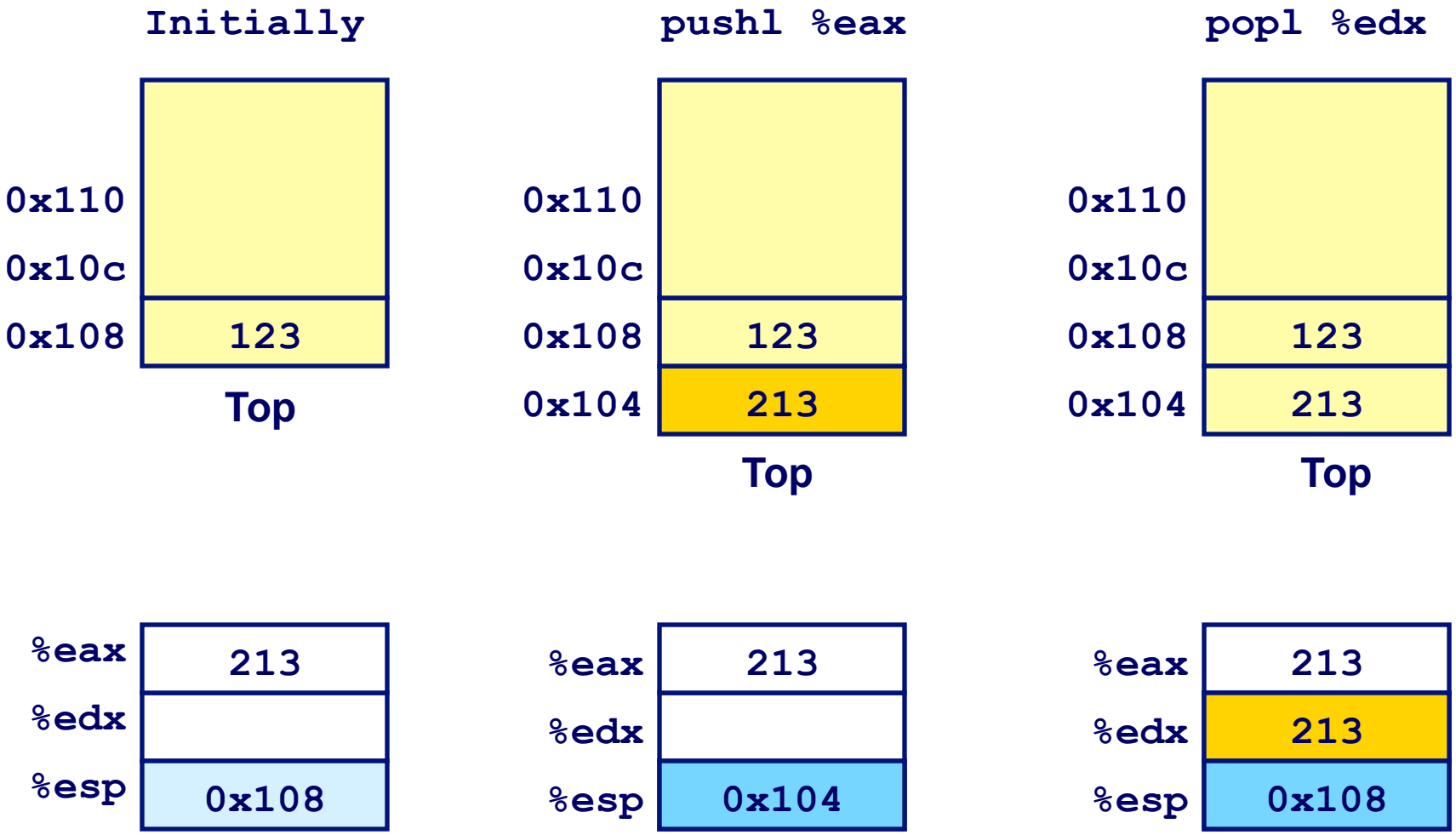
- `popl` *Dest*
  - Read operand at address given by `%esp`
  - Write to *Dest*
  - Increment `%esp` by 4
- *e.g.* `popl %eax`

```
movl (%esp),%eax
addl $4,%esp
```



Stack Operation Examples

animation



Procedure Control Flow

Procedure call:

`call` *`label`*

- Push address of next instruction (after the call) on stack
- This is your return address
- Jump to *`label`*

Procedure return:

- `ret` Pop return address from stack into eip register

Procedure Control Flow

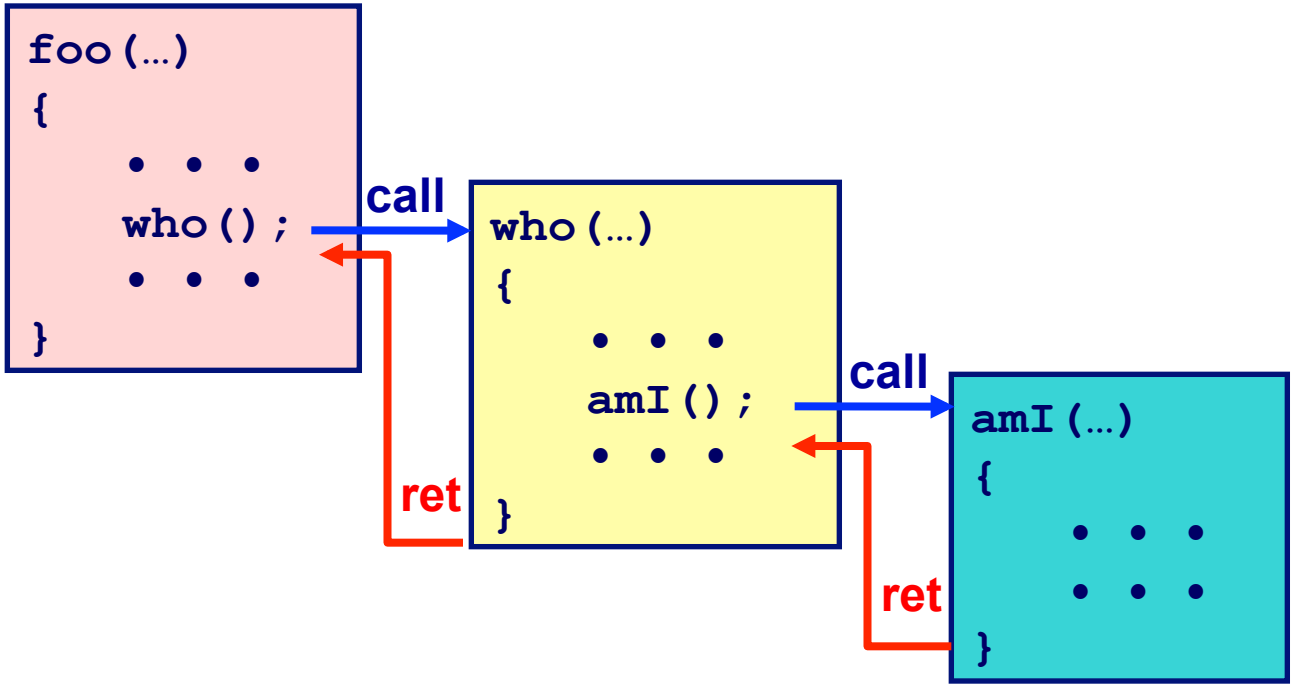
When procedure `foo` calls `who`:

- `foo` is the *caller*, `who` is the *callee*
- Control is transferred to the 'callee'

When procedure returns

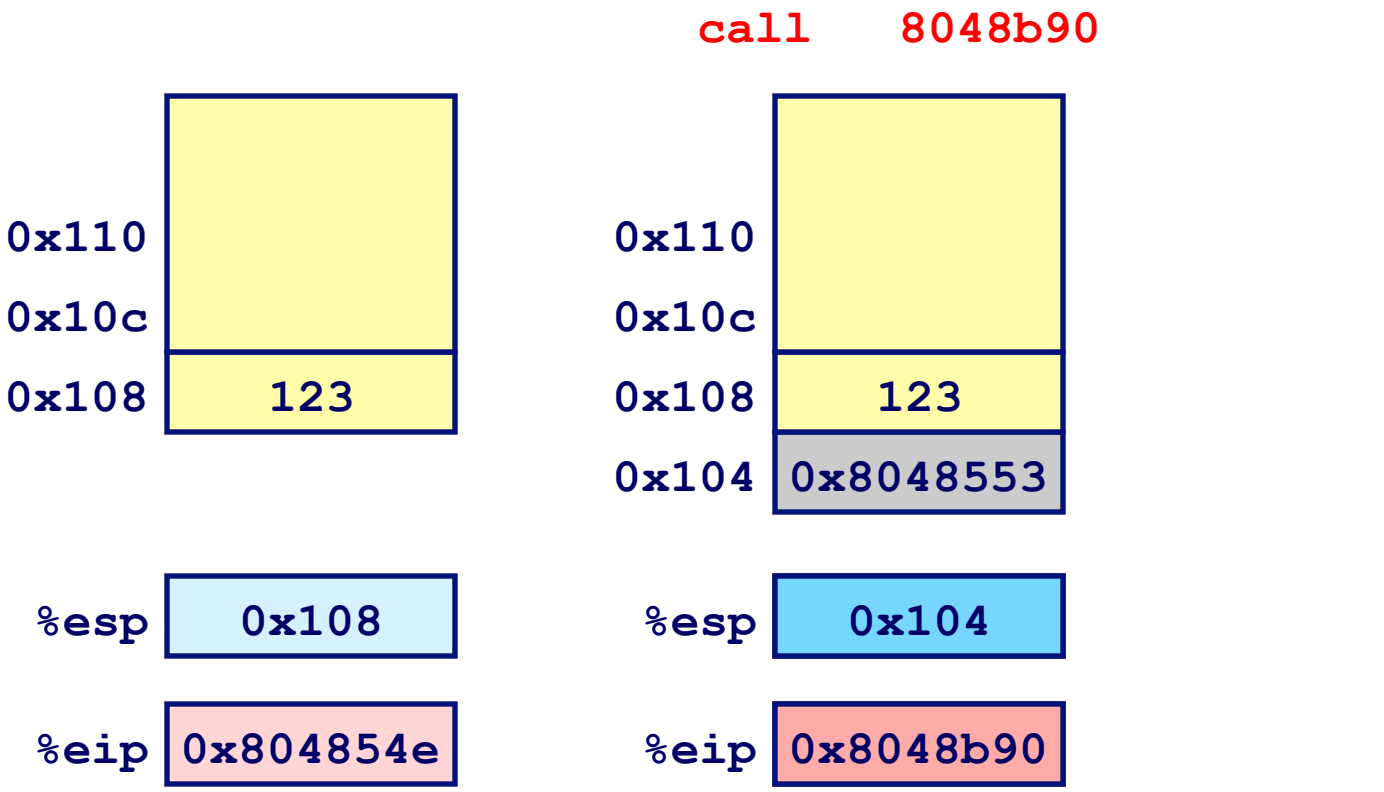
- Control is transferred back to the 'caller'

Last-called, first-return (LIFO) order naturally implemented via stack



Procedure Call Example

```
804854e:  e8 3d 06 00 00        call   8048b90 <main>
8048553:  50                    next instruction
```

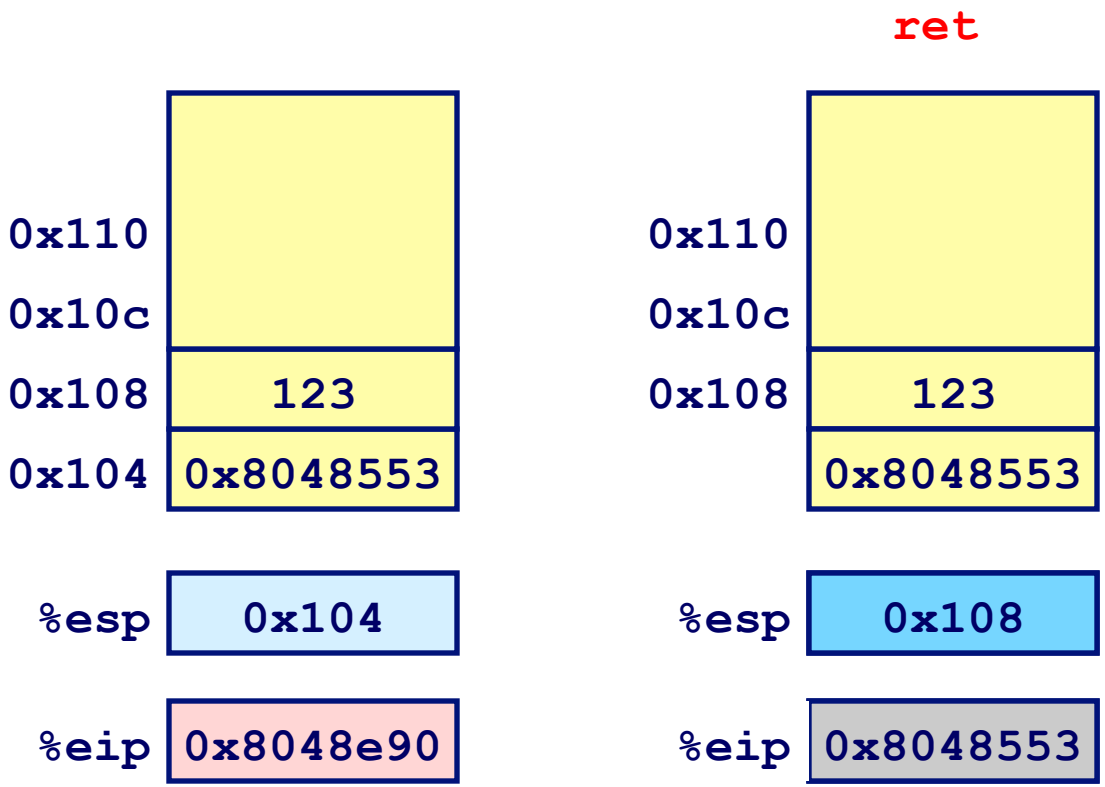


`%eip` *is program counter*

Procedure Return Example

animation

```
8048e90:   c3                    ret
```



`%eip` *is program counter*

Keeping track of stack frames

The stack pointer (%esp) moves around

- Can be changed within a procedure
- Problem
  - How can we consistently find our local variables, return address and parameters?
- The *base pointer* (%ebp)
  - Points to the base of our current stack frame
  - Also called the frame pointer
  - Within each function, %ebp stays constant

Most information on the stack is referenced relative to the base pointer

- Base pointer setup is the programmer's job
  - Actually usually the compiler's job

Procedure calls and stack frames

How does the 'callee' know where to return later?

- Return address placed in a well-known location on stack within a "stack frame"

How are arguments passed to the 'callee'?

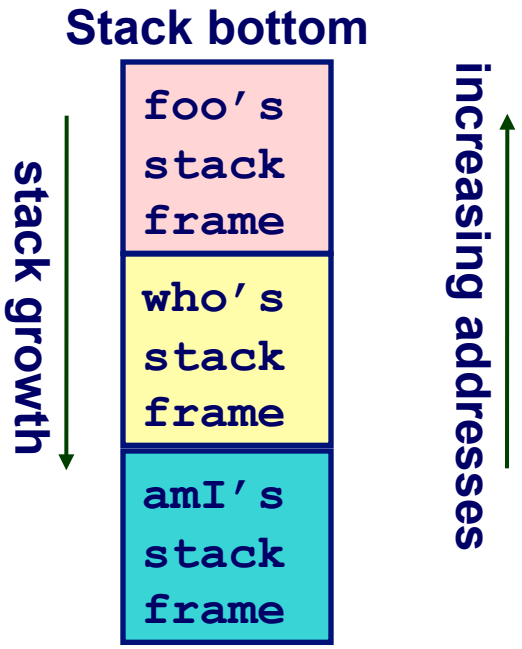
- Arguments placed in a well-known location on stack within a "stack frame"

Upon procedure invocation

- Stack frame created for the procedure
- Stack frame is pushed onto program stack

Upon procedure return

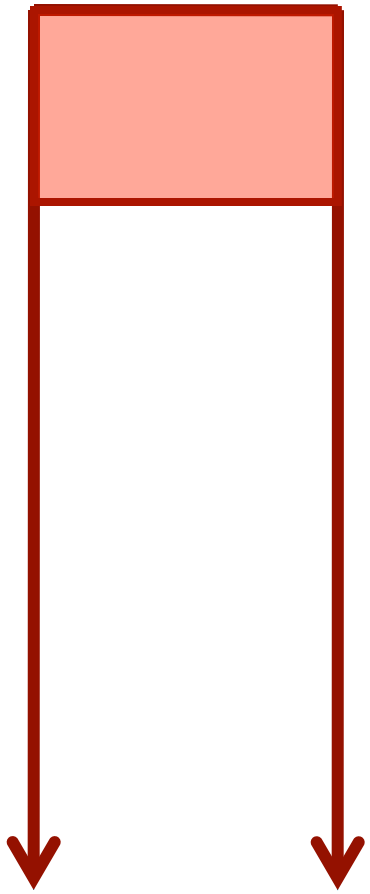
- Its frame is popped off of stack
- Caller's stack frame is recovered



Call chain: foo => who => aml

Functions and Stack Frames

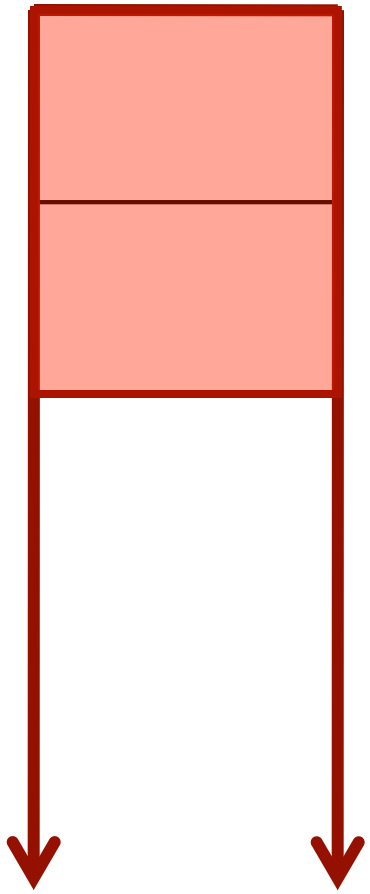
```
main() {
   … call f1()
}

f1() {
   … call f2()
}

f2() {
   … call f3()
}

f3 () {
    …
}
```



Frame of **main**

Functions and Stack Frames

```
main() {
   … call f1()
}

f1() {
   … call f2()
}

f2() {
   … call f3()
}

f3 () {
    …
}
```

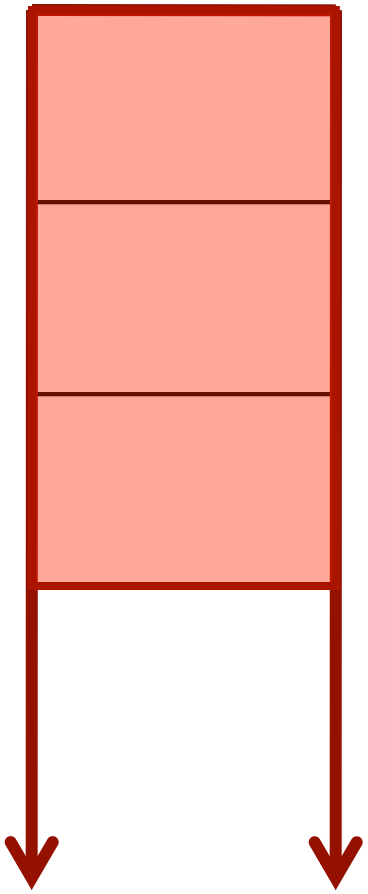


Frame of **`main`**

Frame of **`f1`**

Functions and Stack Frames

```
main() {
   … call f1()
}

f1() {
   … call f2()
}

f2() {
   … call f3()
}

f3 () {
    …
}
```



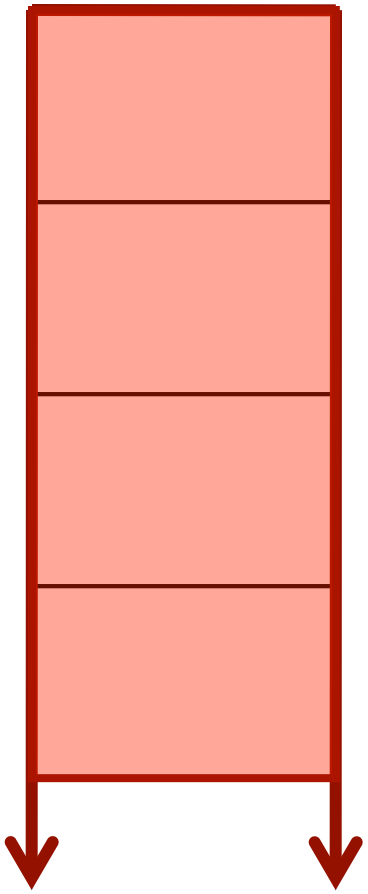
Frame of **main**

Frame of **f1**

Frame of **f2**

Functions and Stack Frames

```
main() {
   … call f1()
}

f1() {
   … call f2()
}

f2() {
   … call f3()
}

f3 () {
    …
}
```



Frame of **`main`**

Frame of **`f1`**

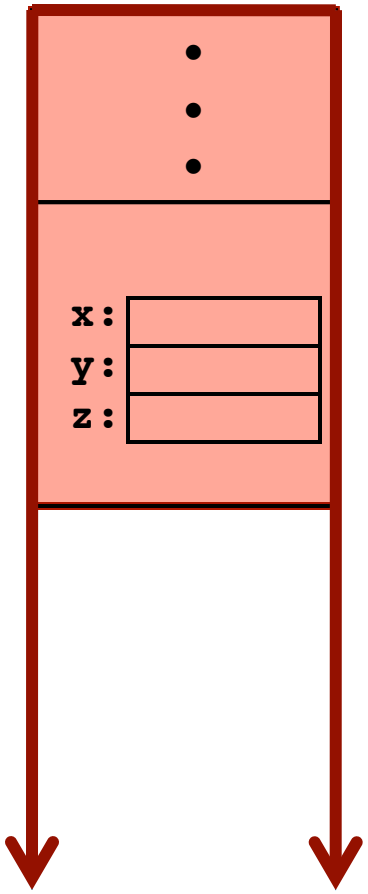
Frame of **`f2`**

Frame of **`f3`**

Functions and Stack Frames

```
f1() {
  int z,y,z;
  … call f2()
}
```

```
f2() {
  int a,b,c;
  … call f3()
}
```

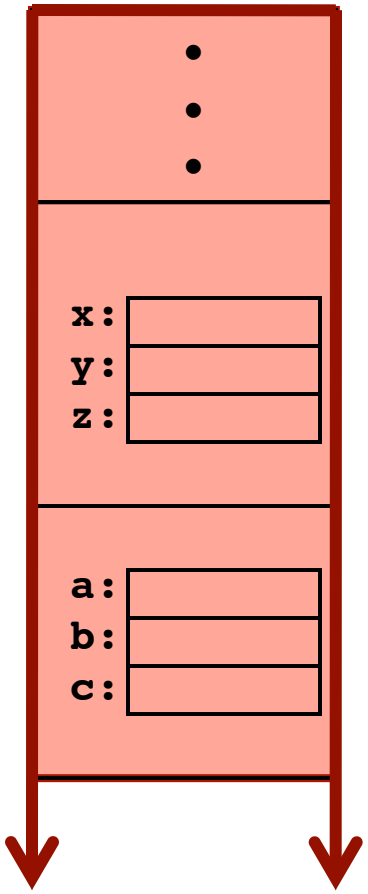


Frame of **f1**

Functions and Stack Frames

```
f1() {
   int z,y,z;
   … call f2()
}
```

```
f2() {
   int a,b,c;
   … call f3()
}
```



Frame of **f1**

Frame of **f2**

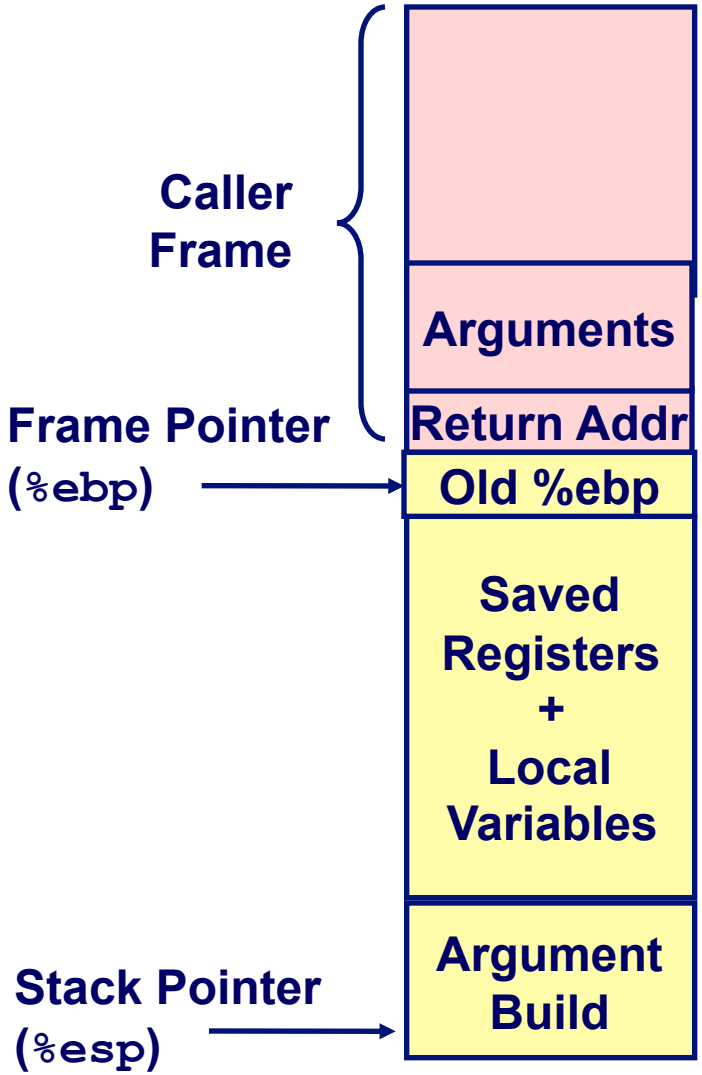
IA32/Linux Stack Frame

Caller Stack Frame (Pink)

- Argument build for callee
  - i.e. function parameters of callee
- Return address
  - Pushed by `call` instruction

Callee Stack Frame (Yellow) (From Top to Bottom)

- Old frame pointer
- Local variables
  - If can't keep in registers
- Saved register context
- Argument build for next call
  - i.e. function parameters for next call



Stack Frame Creation and Destruction

```
f2(…) {
   …
}
```

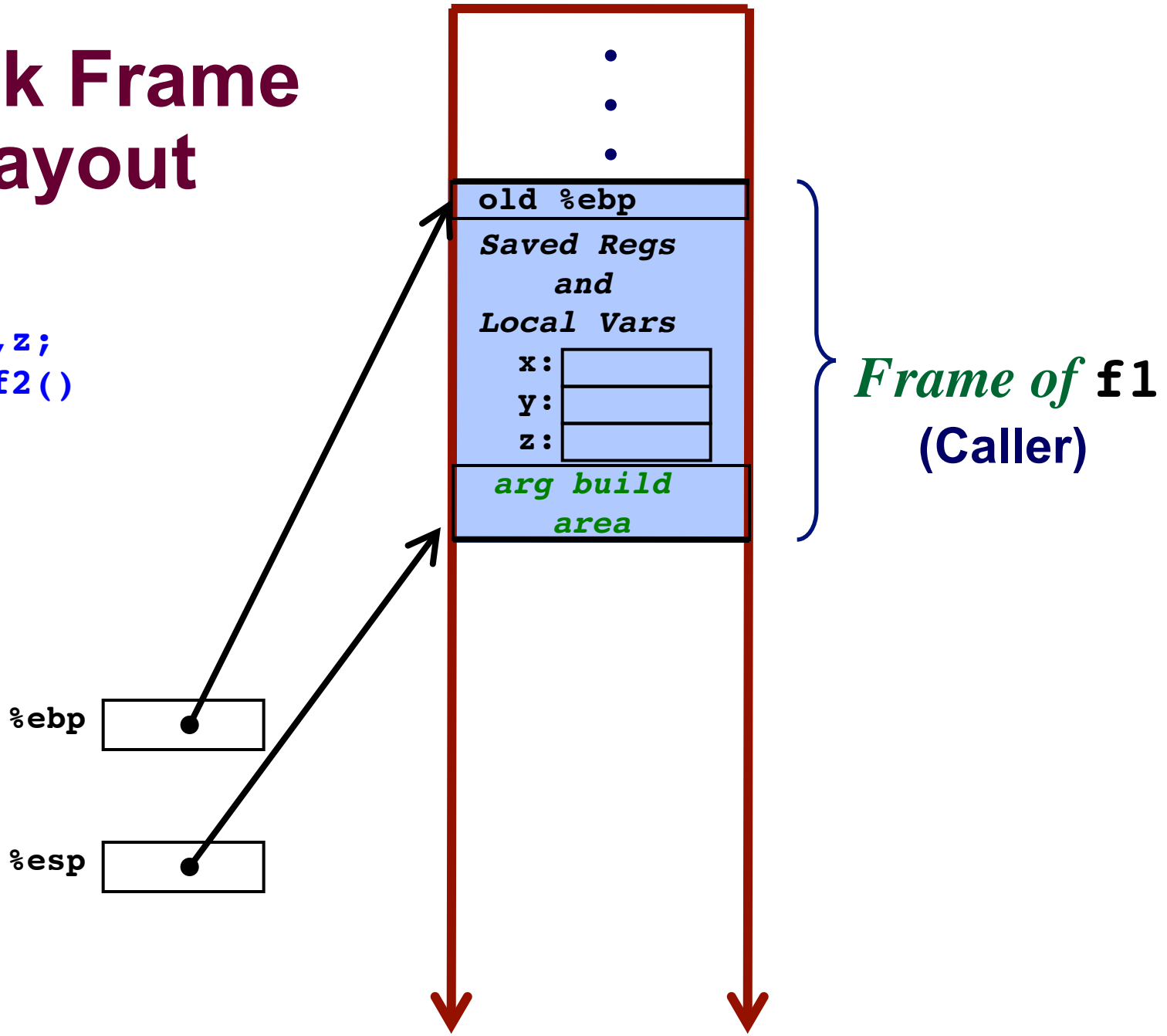
```
f2:
    pushl   %ebp
    movl    %esp,%ebp

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

Prologue

Epilogue

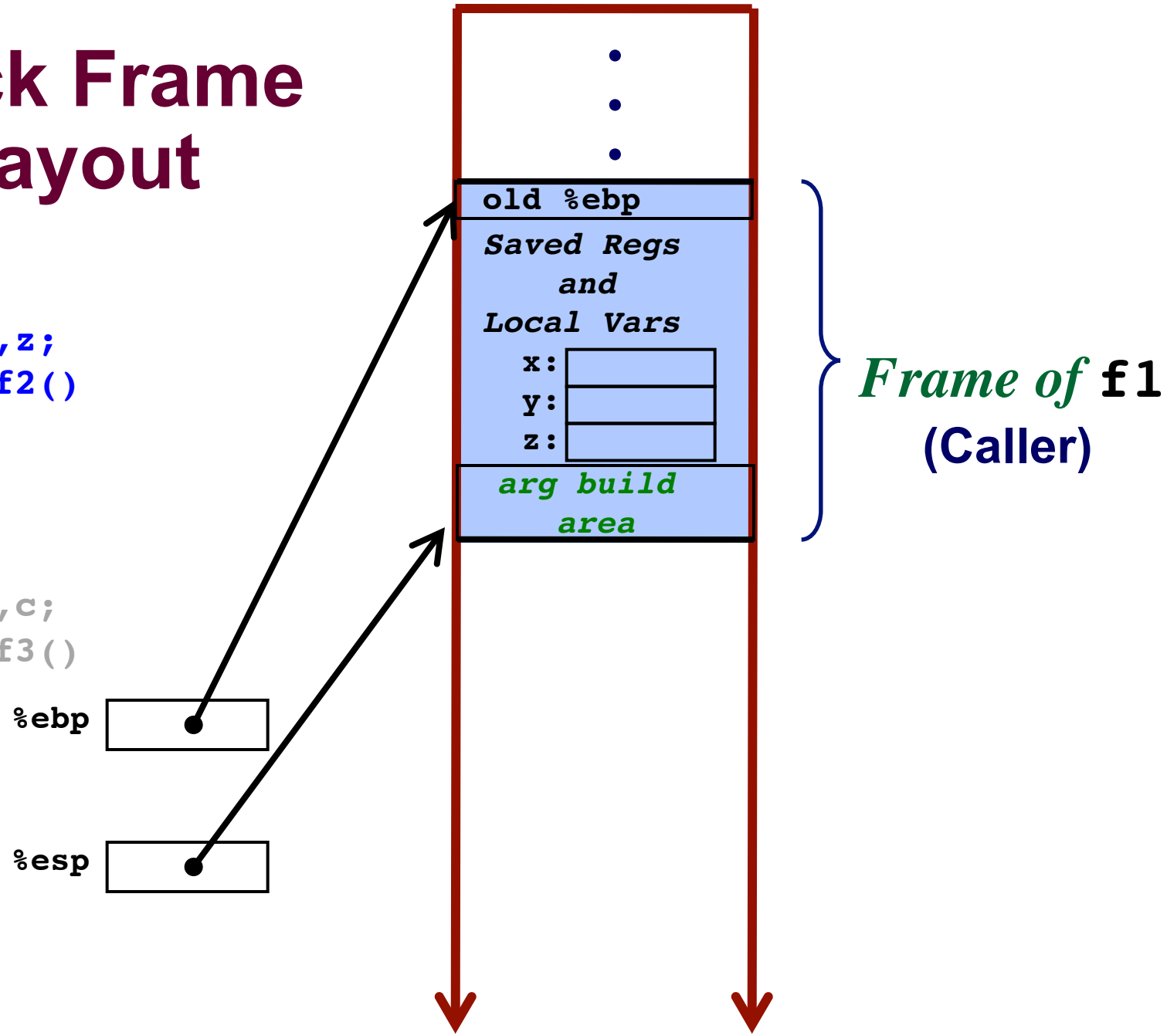
Stack Frame Layout

```
f1() {
   int z,y,z;
   … call f2()
}
```



Stack Frame Layout

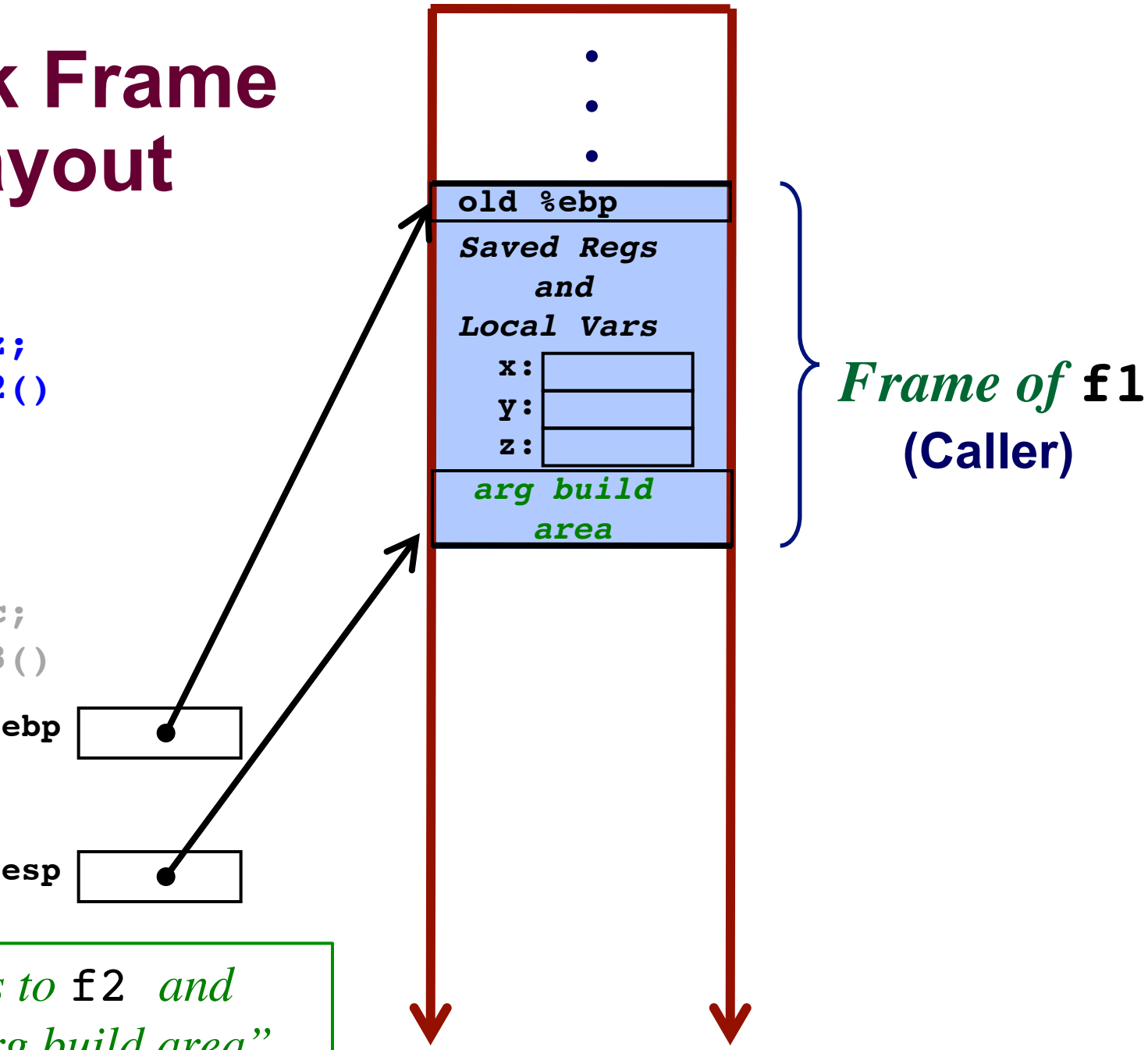
```
f1() {
  int z,y,z;
  … call f2()
}

f2() {
  int a,b,c;
  … call f3()
}
```



Stack Frame Layout

```
f1() {
  int z,y,z;
  … call f2()
}
```

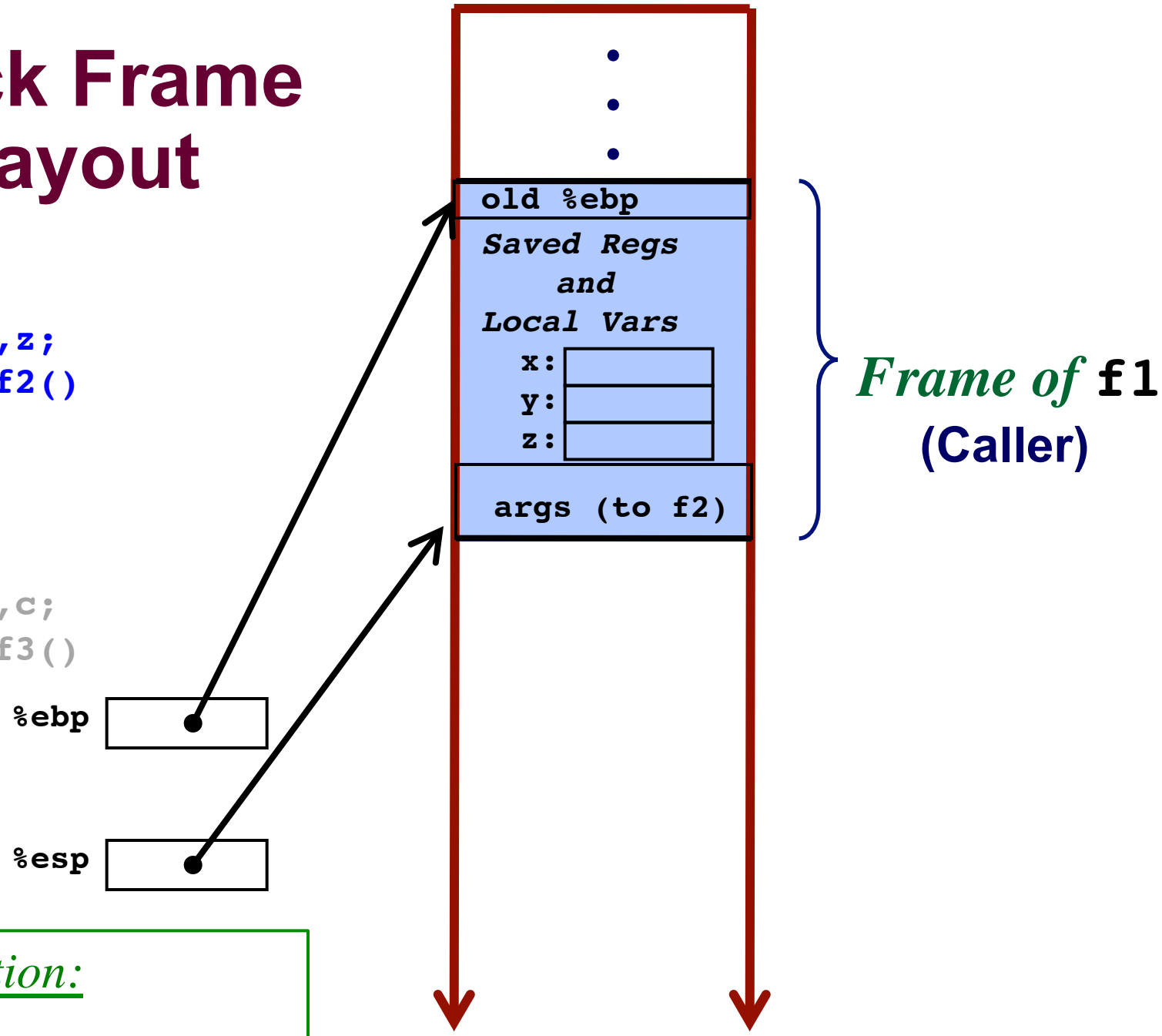
```
f2() {
  int a,b,c;
  … call f3()
}
```



Compute args to `f2` *and move into "arg build area"*

Stack Frame Layout

```
f1() {
  int z,y,z;
  … call f2()
}

f2() {
  int a,b,c;
  … call f3()
}
```

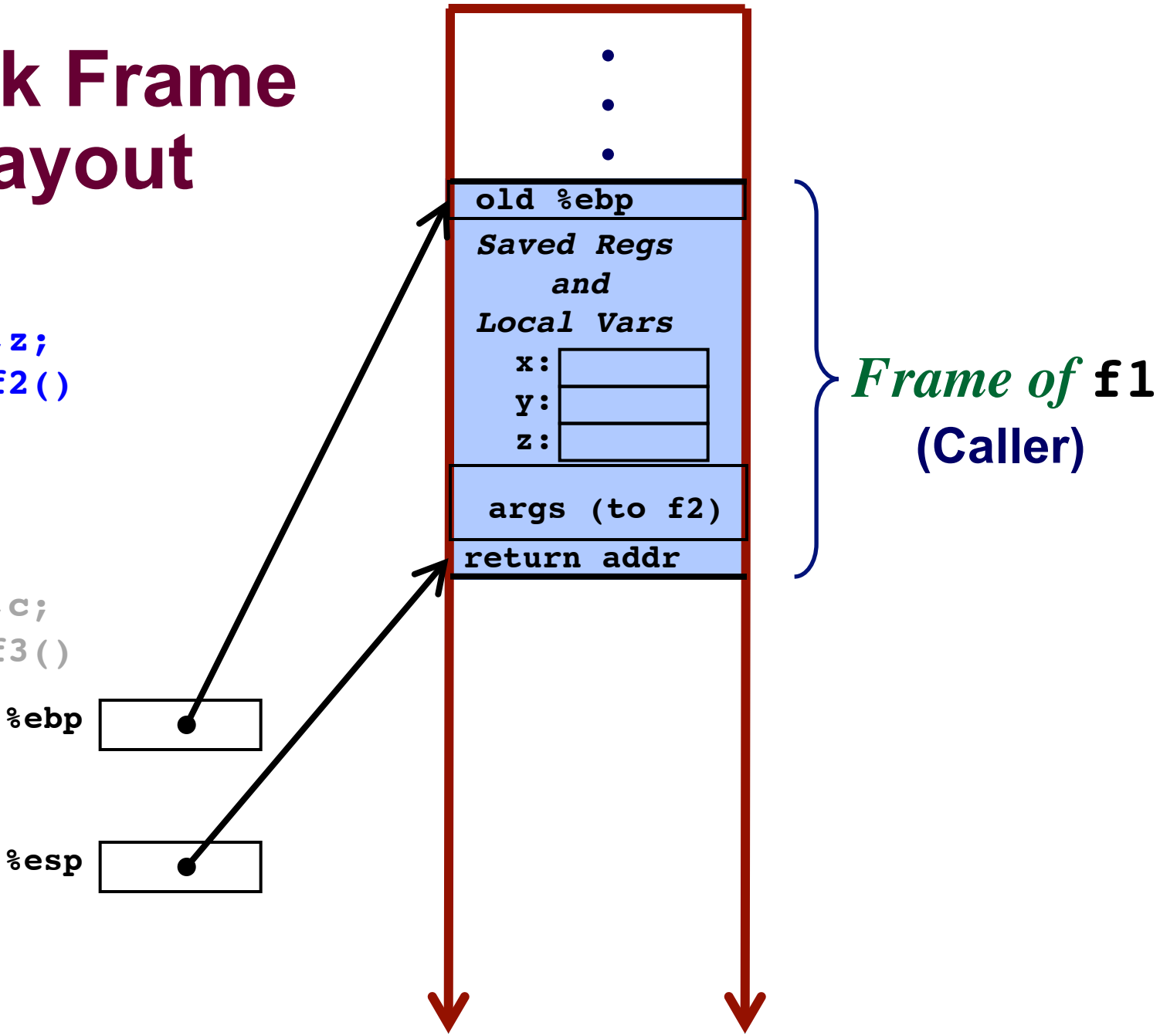


next instruction:

```
call   f2
```

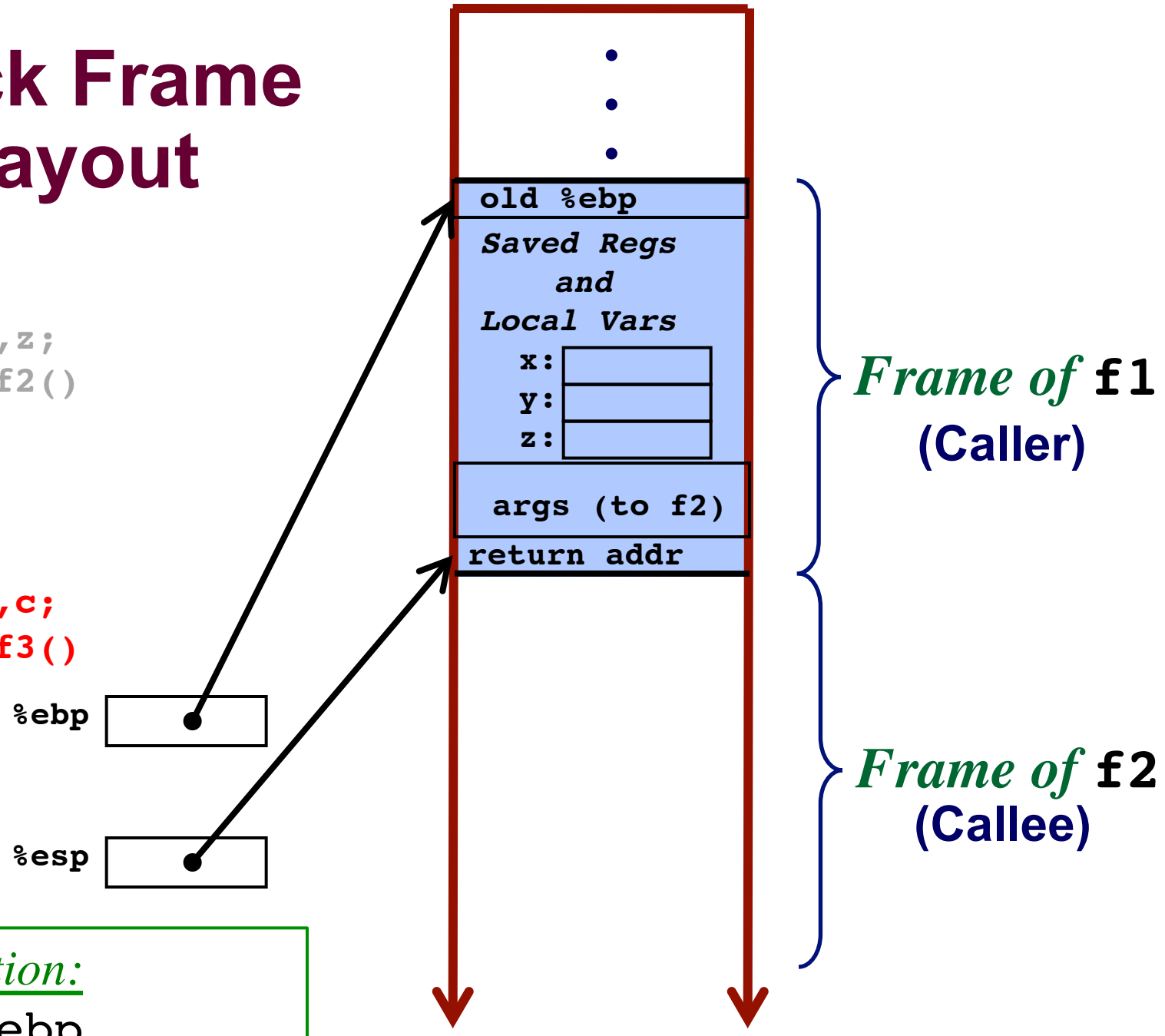
Stack Frame Layout

```
f1() {
  int z,y,z;
  … call f2()
}

f2() {
  int a,b,c;
  … call f3()
}
```



Stack Frame Layout

```
f1() {
   int z,y,z;
   … call f2()
}

f2() {
   int a,b,c;
   … call f3()
}
```

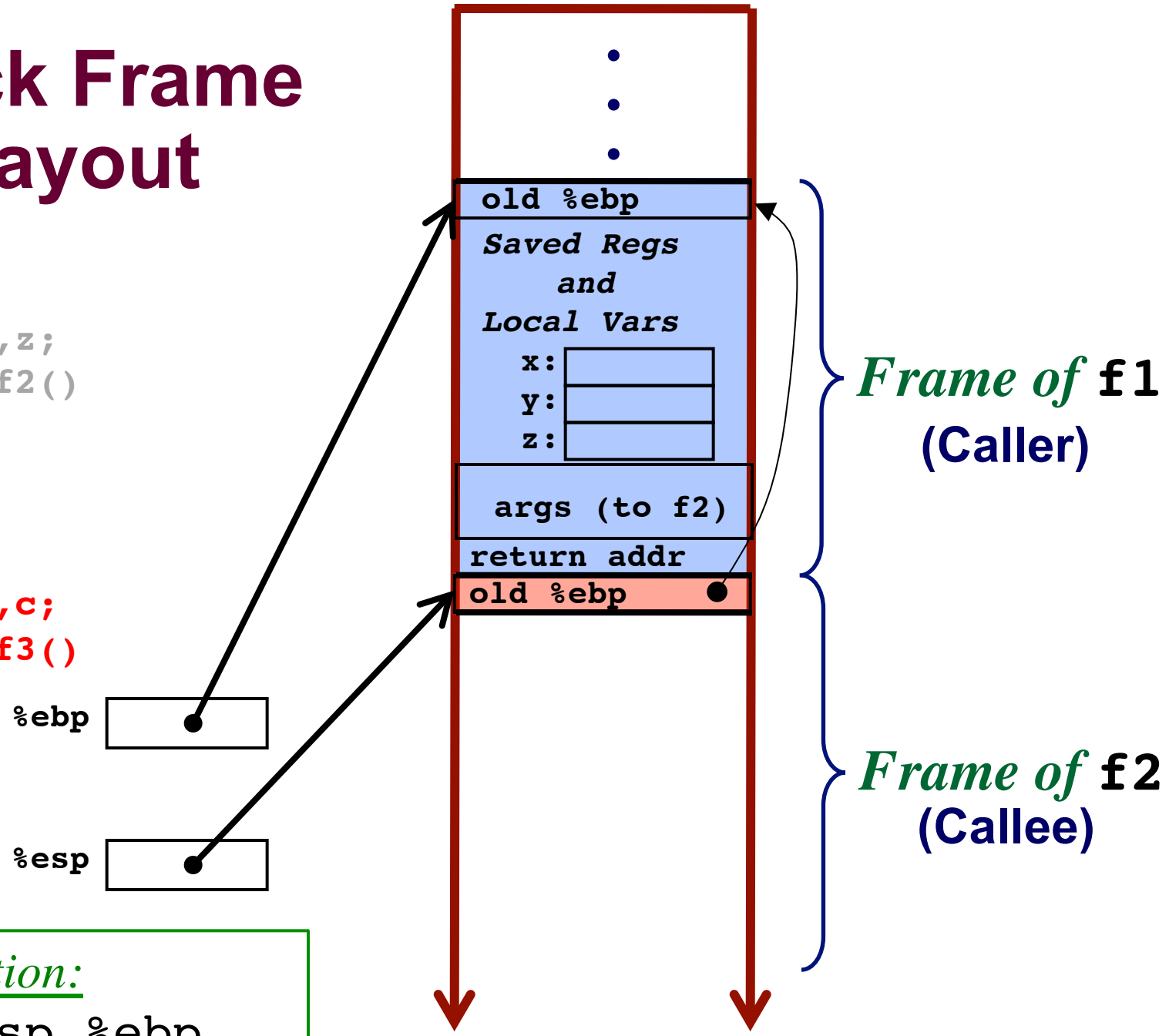


next instruction:

```
pushl   %ebp
```

Stack Frame Layout

```
f1() {
   int z,y,z;
   … call f2()
}

f2() {
   int a,b,c;
   … call f3()
}
```



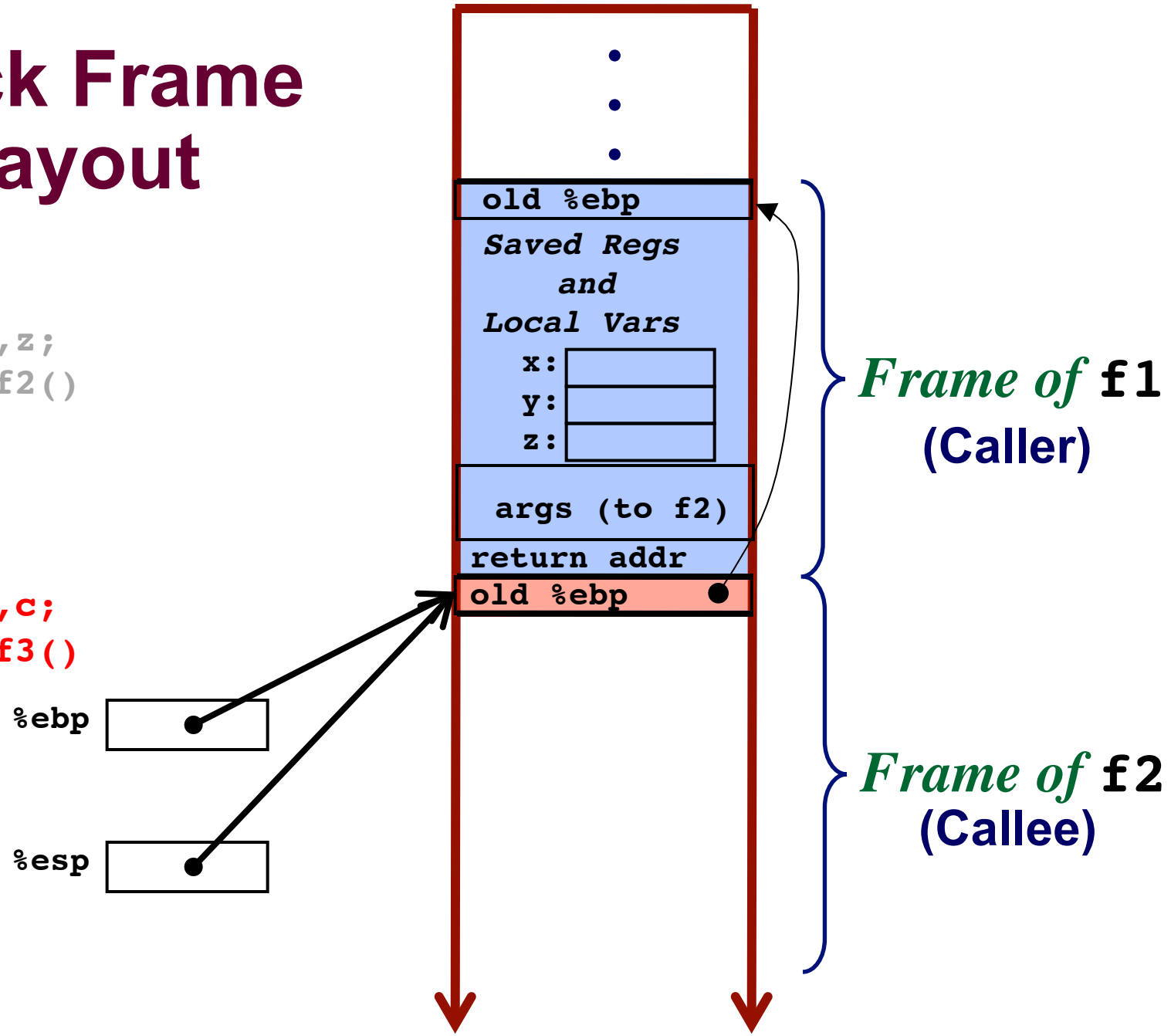
next instruction:

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

Stack Frame Layout

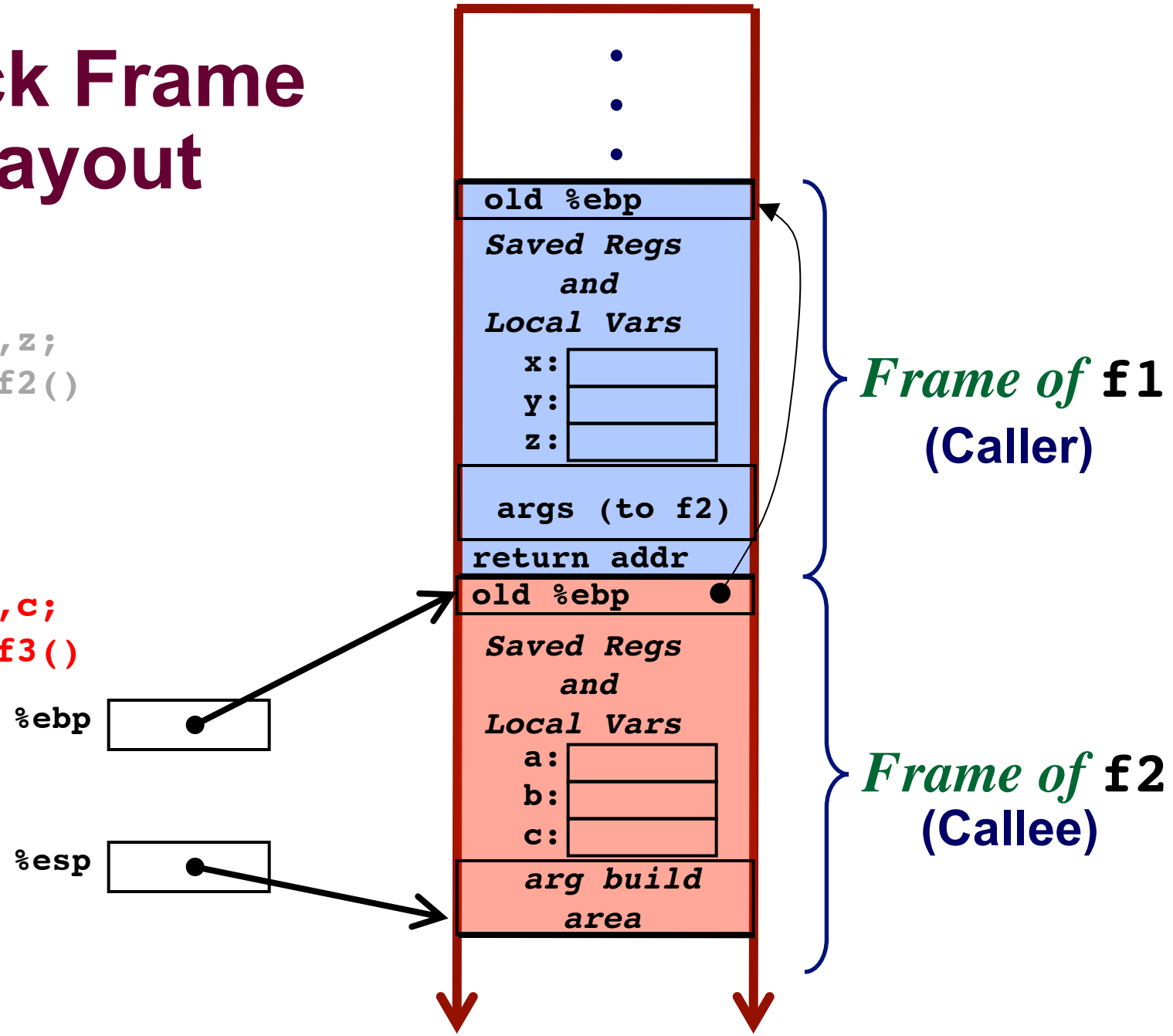
```
f1() {
  int z,y,z;
  … call f2()
}
```

```
f2() {
  int a,b,c;
  … call f3()
}
```



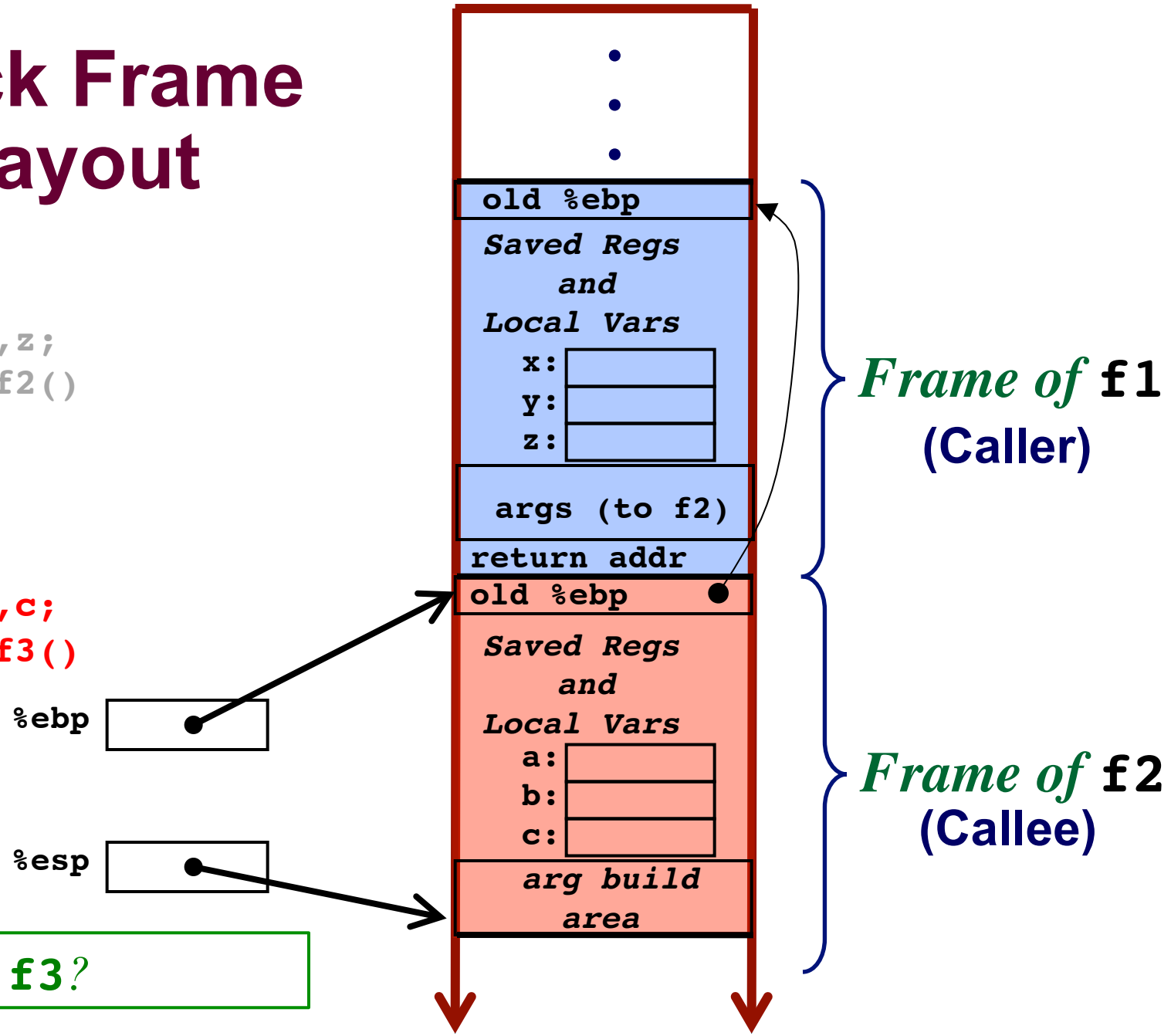
Stack Frame Layout

```
f1() {
  int z,y,z;
  … call f2()
}

f2() {
  int a,b,c;
  … call f3()
}
```



Stack Frame Layout

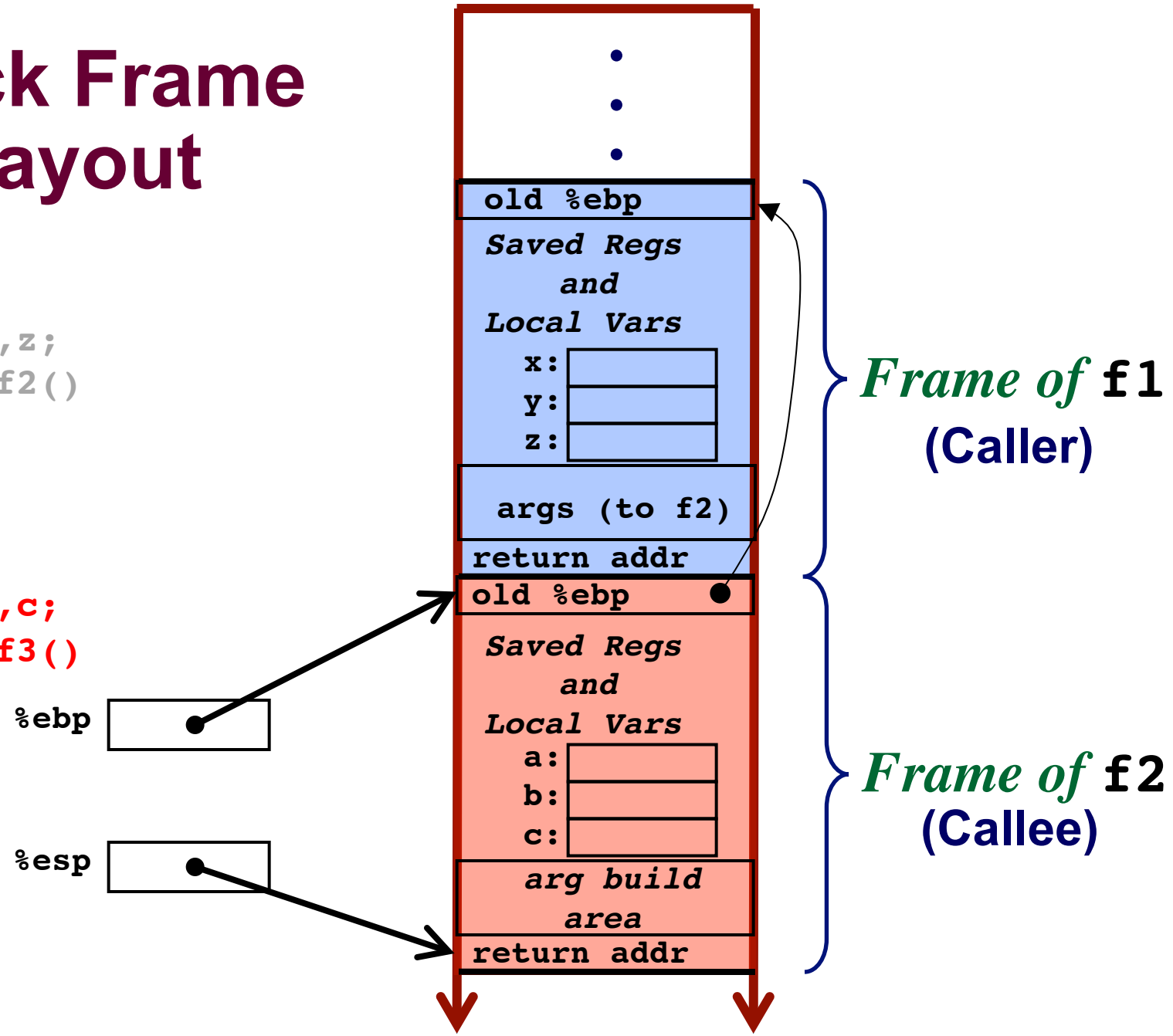
```
f1() {
   int z,y,z;
   … call f2()
}

f2() {
   int a,b,c;
   … call f3()
}
```



Want to call **f3**?

Stack Frame Layout

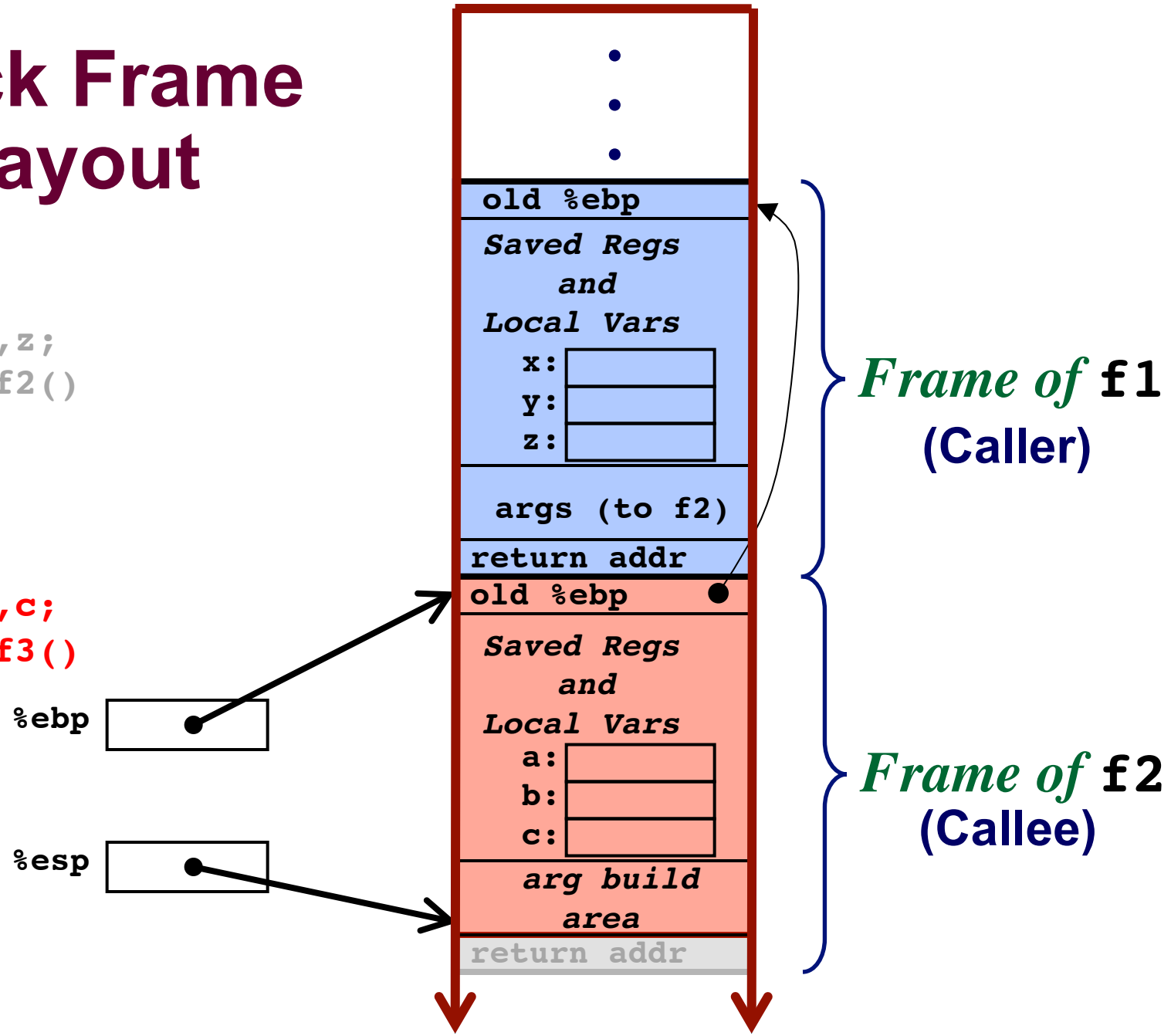
```
f1() {
   int z,y,z;
   … call f2()
}

f2() {
   int a,b,c;
   … call f3()
}
```



Stack Frame Layout

```
f1() {
  int z,y,z;
  … call f2()
}
```

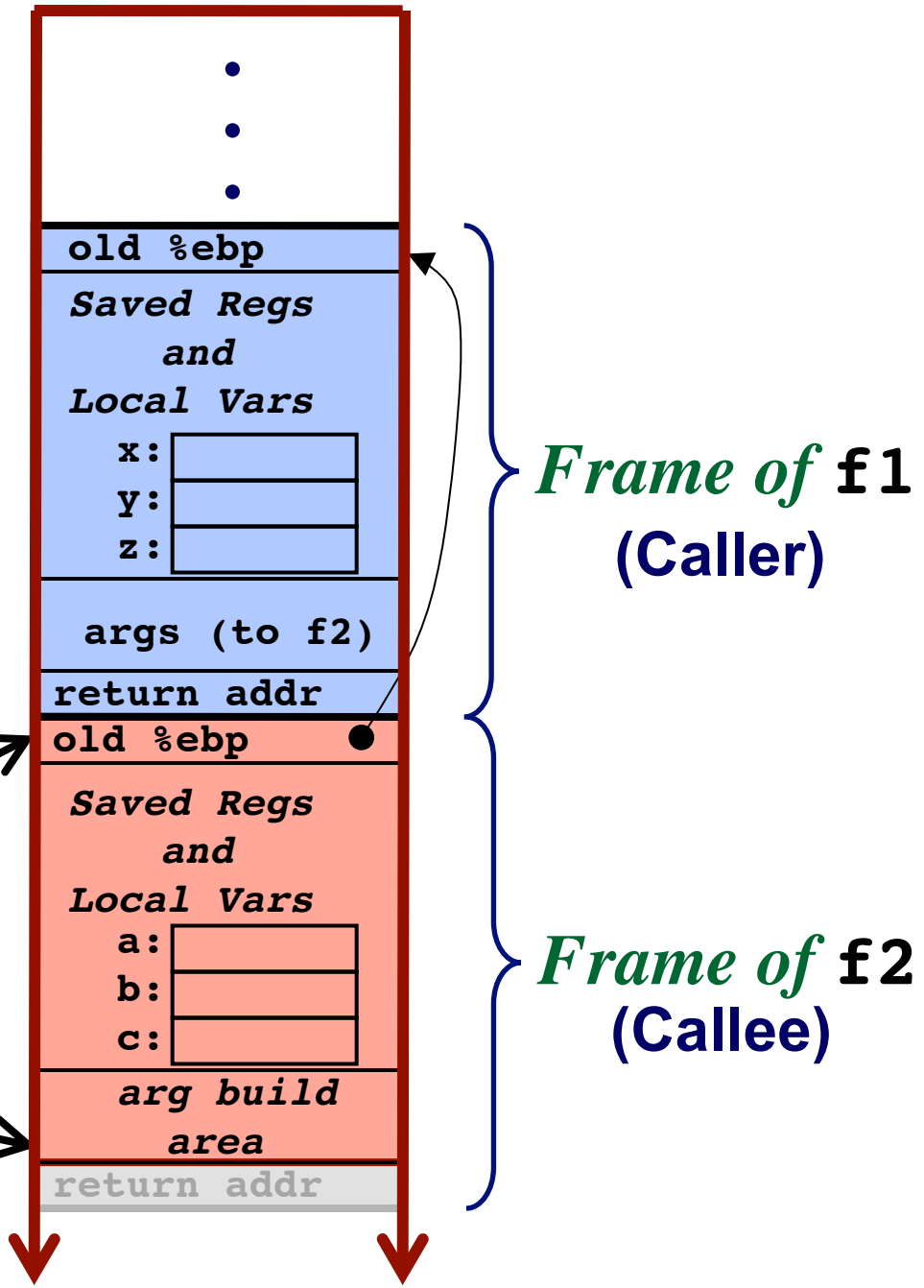
```
f2() {
  int a,b,c;
  … call f3()
}
```



Stack Frame Layout

```
f1() {
   int z,y,z;
   … call f2()
}
```

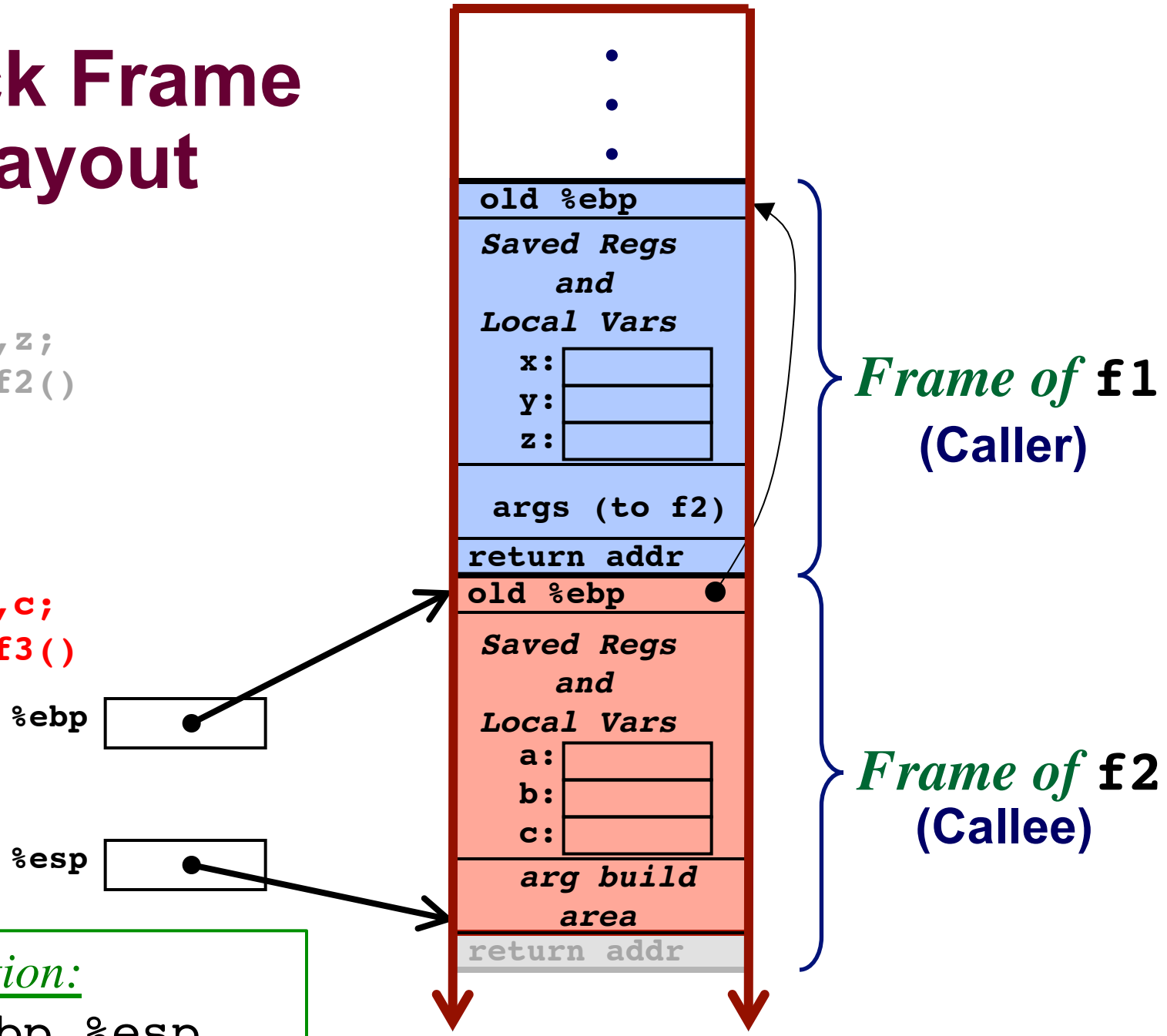
```
f2() {
   int a,b,c;
   … call f3()
}
```



Ready to return?

Stack Frame Layout

```
f1() {
   int z,y,z;
   … call f2()
}

f2() {
   int a,b,c;
   … call f3()
}
```

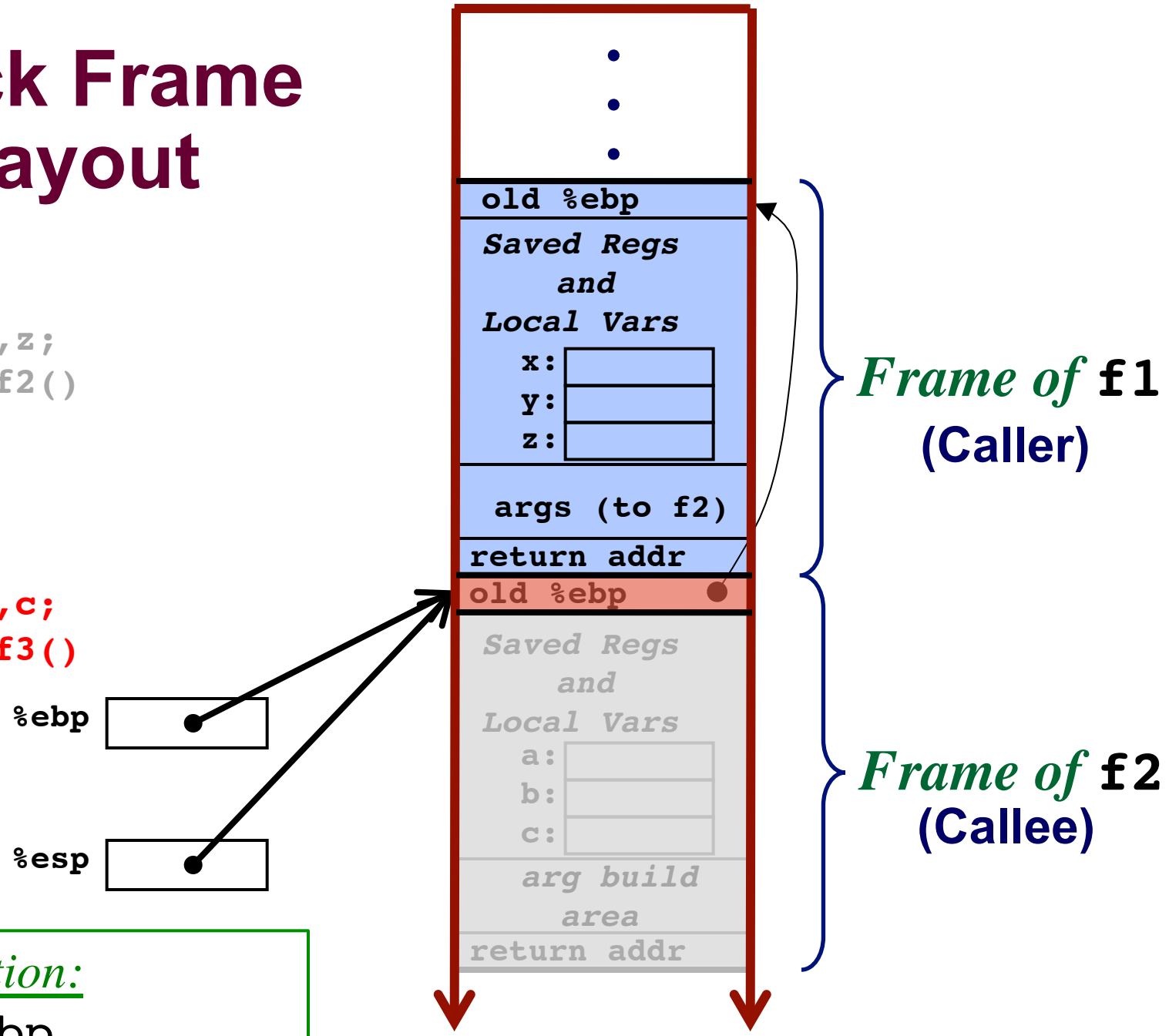


next instruction:

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

Stack Frame Layout

```
f1() {
   int z,y,z;
   … call f2()
}

f2() {
   int a,b,c;
   … call f3()
}
```

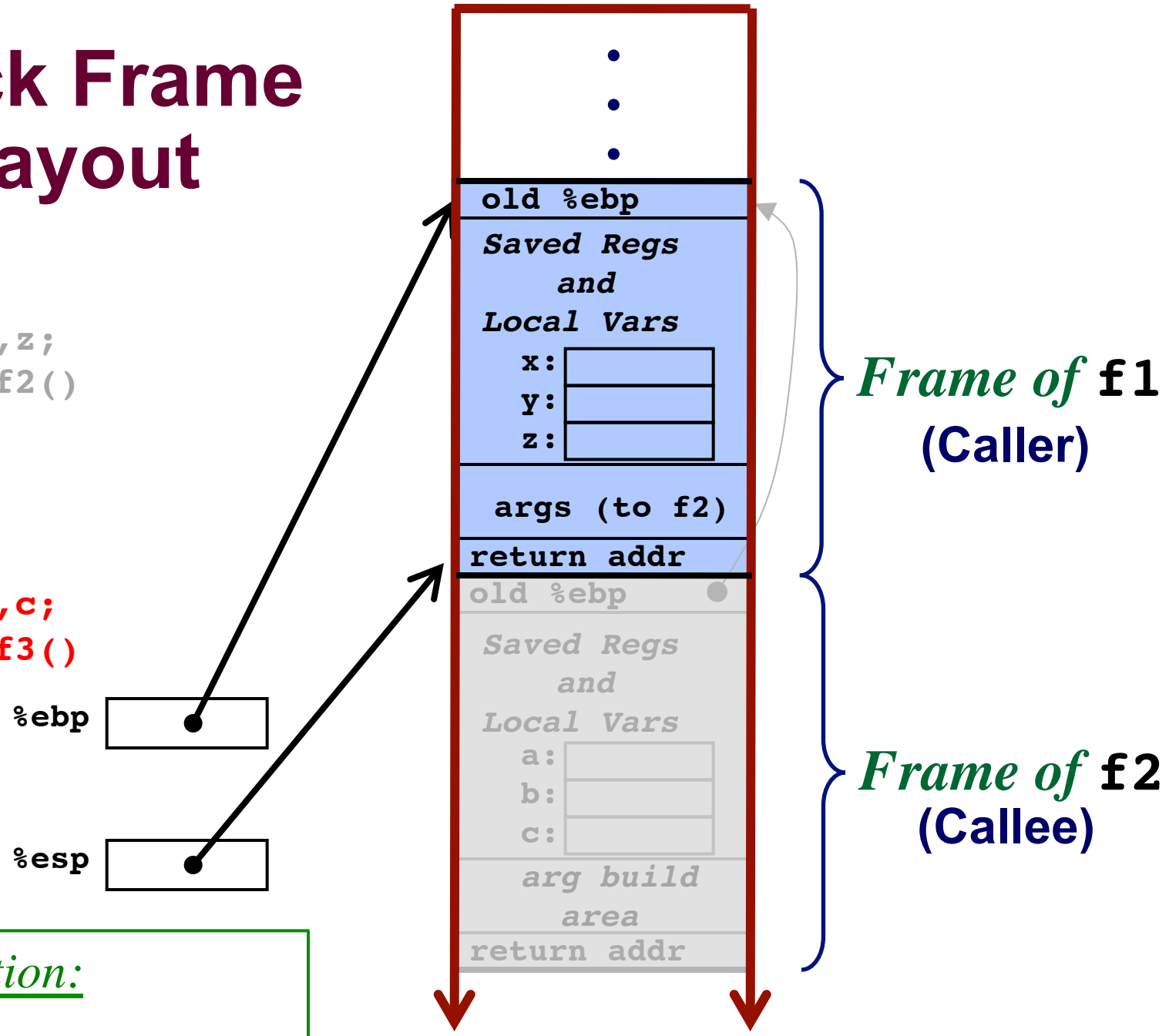


next instruction:

```
popl   %ebp
```

Stack Frame Layout

```
f1() {
   int z,y,z;
   … call f2()
}

f2() {
   int a,b,c;
   … call f3()
}
```

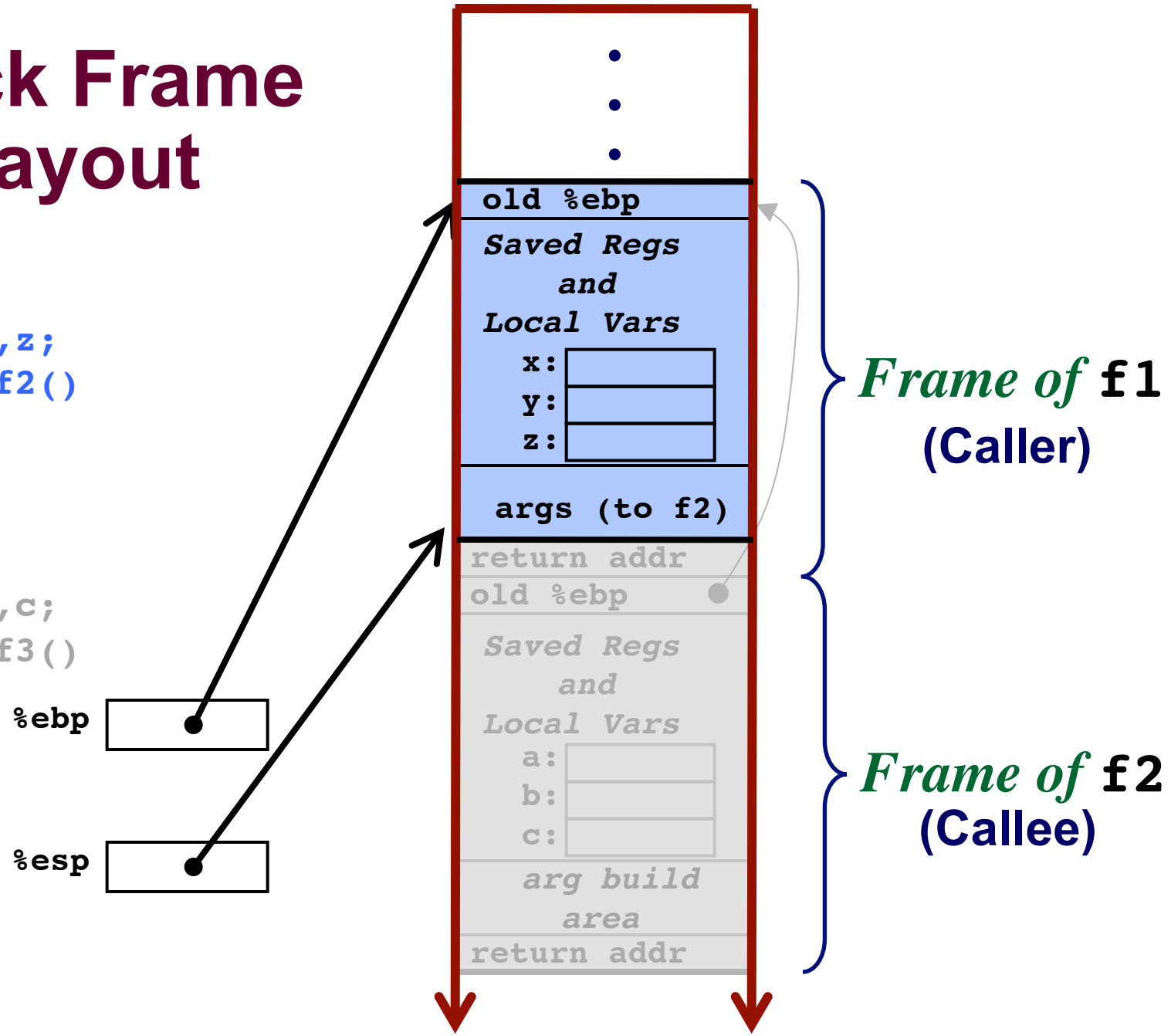


next instruction:

`ret`

Stack Frame Layout

```
f1() {
  int z,y,z;
  … call f2()
}

f2() {
  int a,b,c;
  … call f3()
}
```



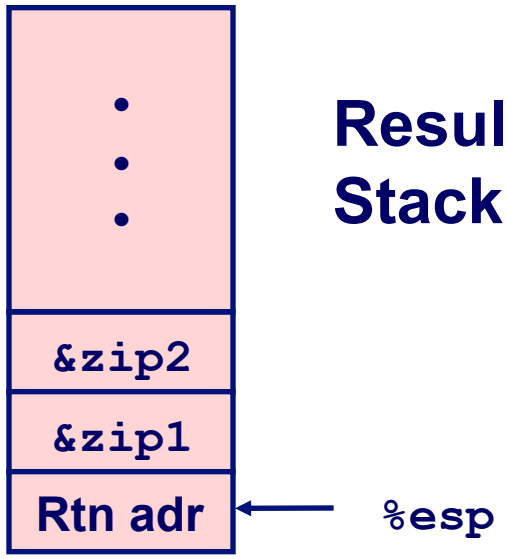
Revisiting swap

```
int zip1 = 15213;
int zip2 = 91125;

void call_swap()
{
  swap(&zip1, &zip2);
}
```

Calling swap from call_swap

```
call_swap:
    • • •
    pushl $zip2   # Global Var
    pushl $zip1   # Global Var
    call swap
    • • •
```

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Resulting Stack

Revisiting swap

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Setup

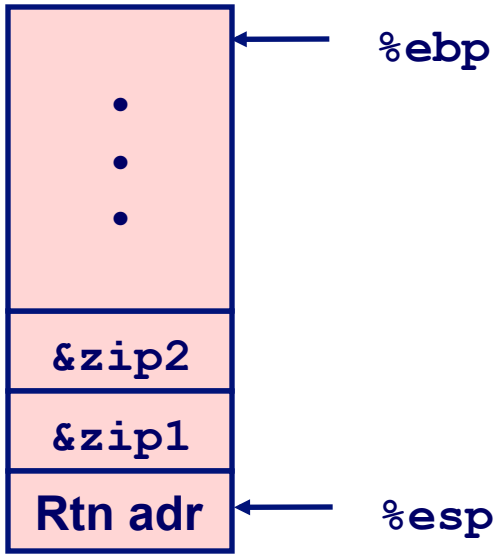
Body

Finish

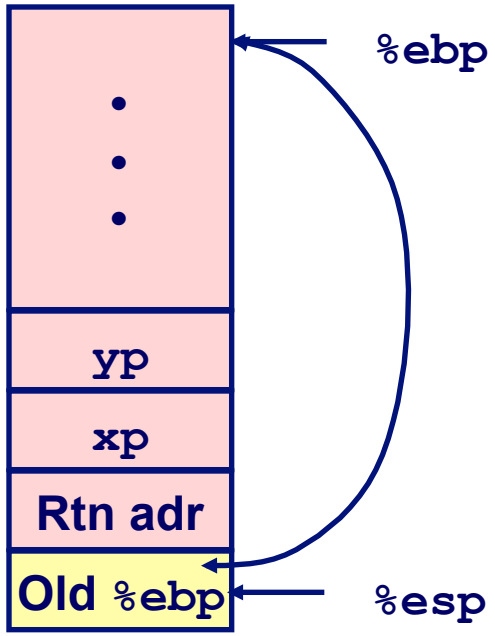
```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

swap Setup #1

Entering Stack



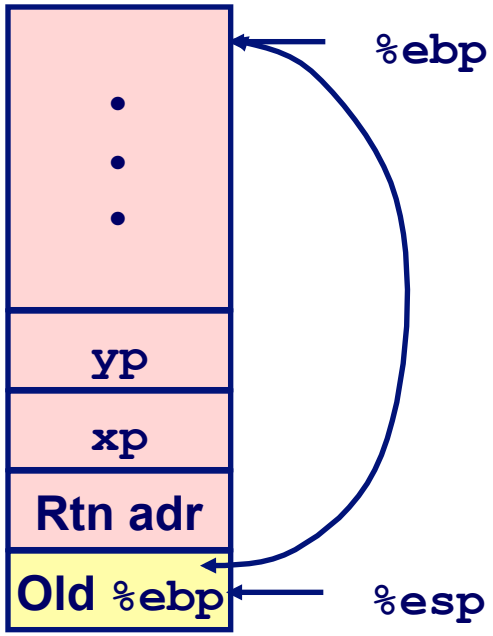
Resulting stack



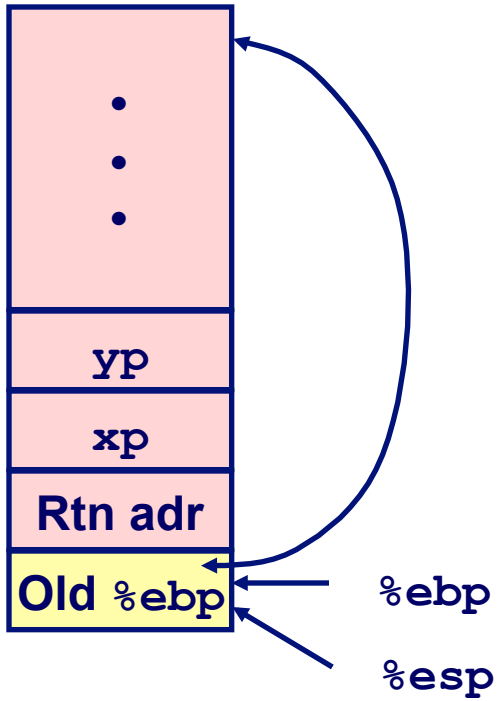

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #2

Stack before instruction



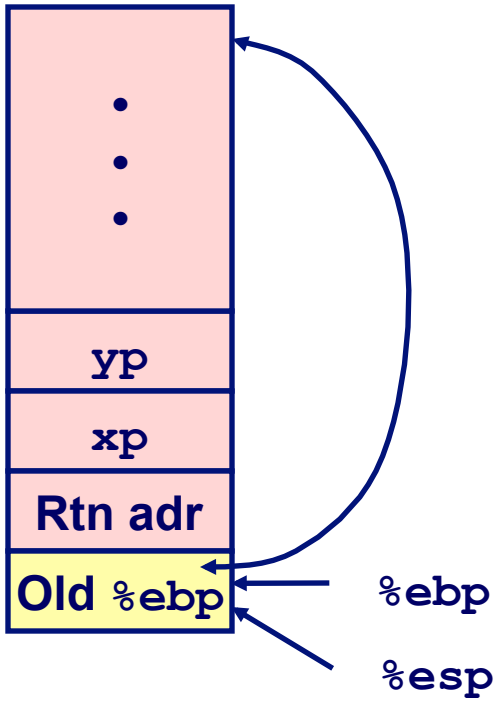
Resulting stack



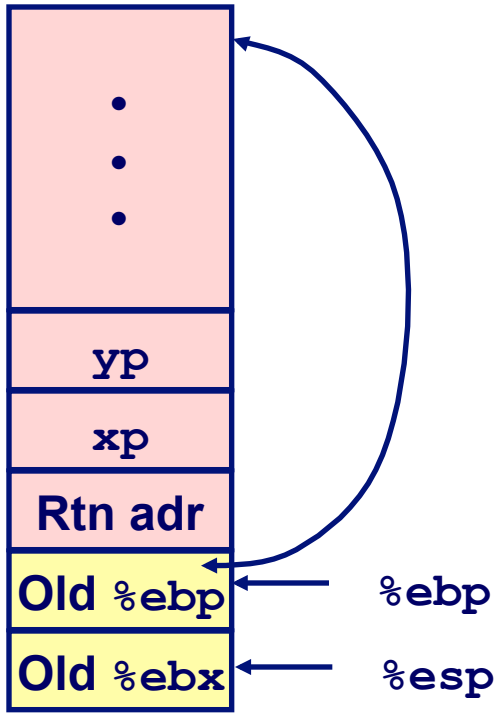

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #3

Stack before instruction

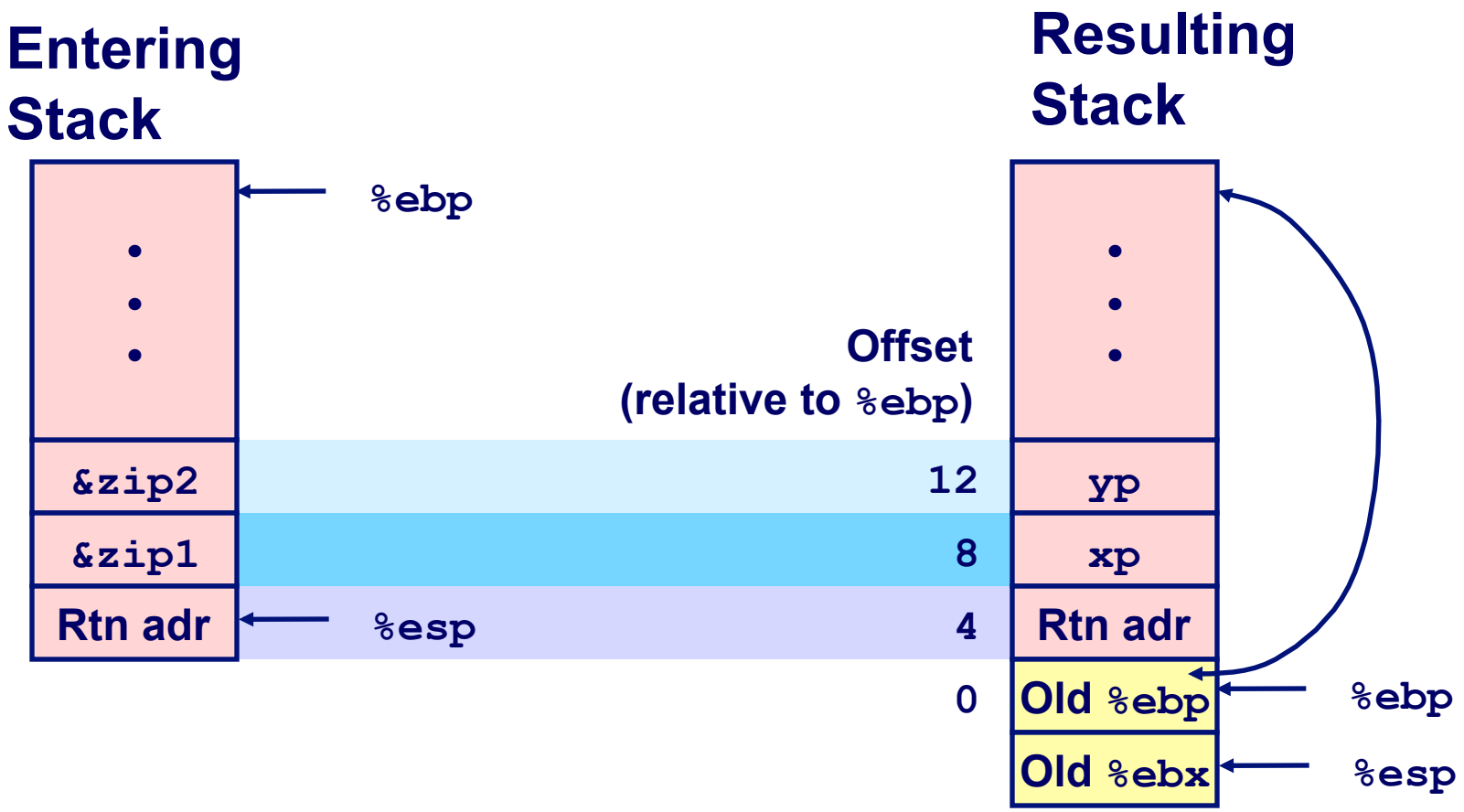


Resulting Stack




```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

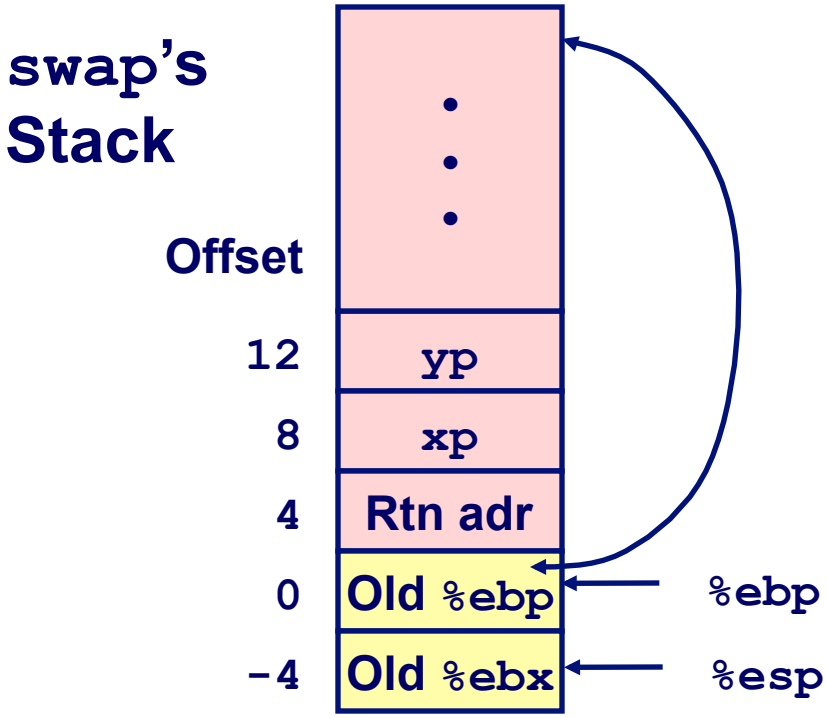
Effect of `swap` Setup

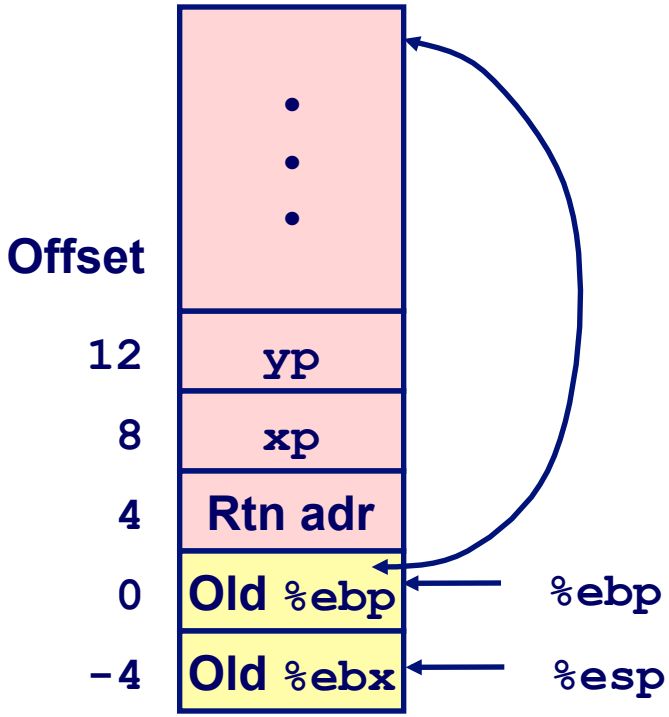



```
movl 12(%ebp),%ecx # get yp
movl 8(%ebp),%edx  # get xp
. . .
```

Body

swap Finish #1



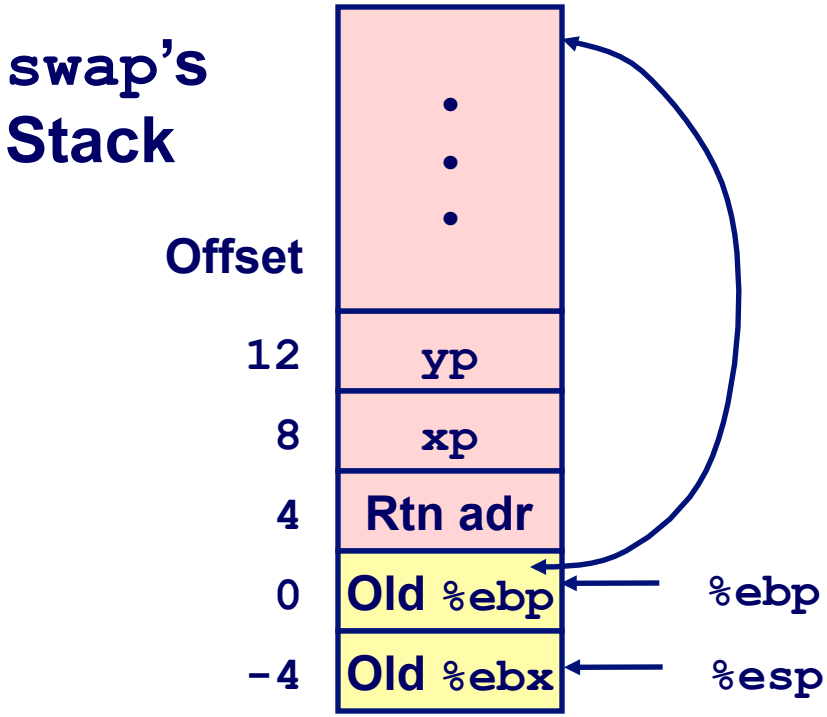


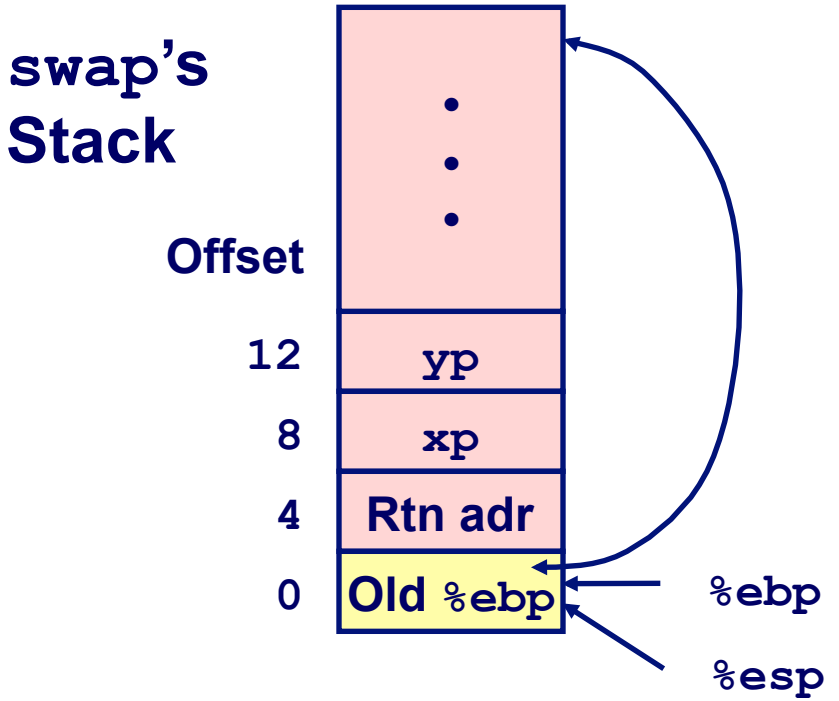

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Observation

- Saved & restored register `%ebx`

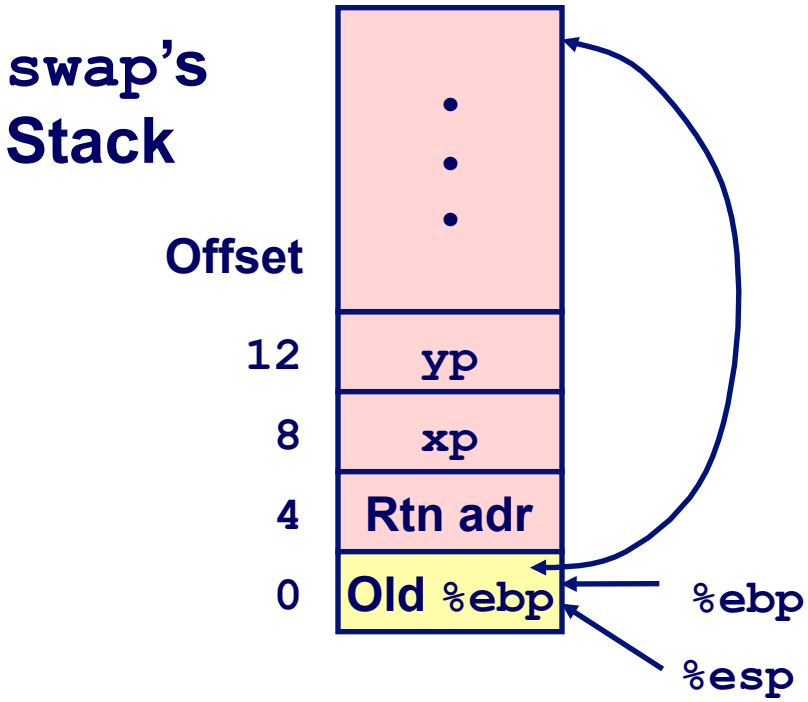
swap Finish #2

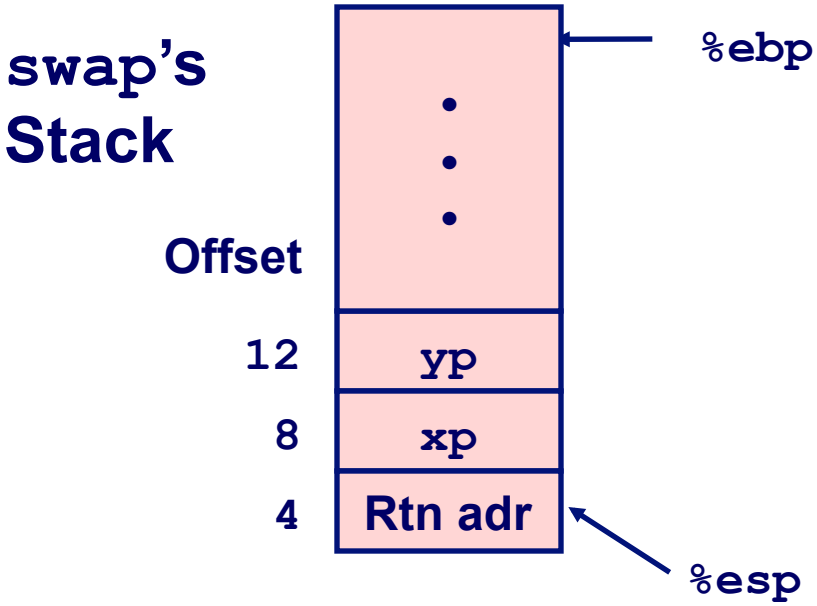





```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

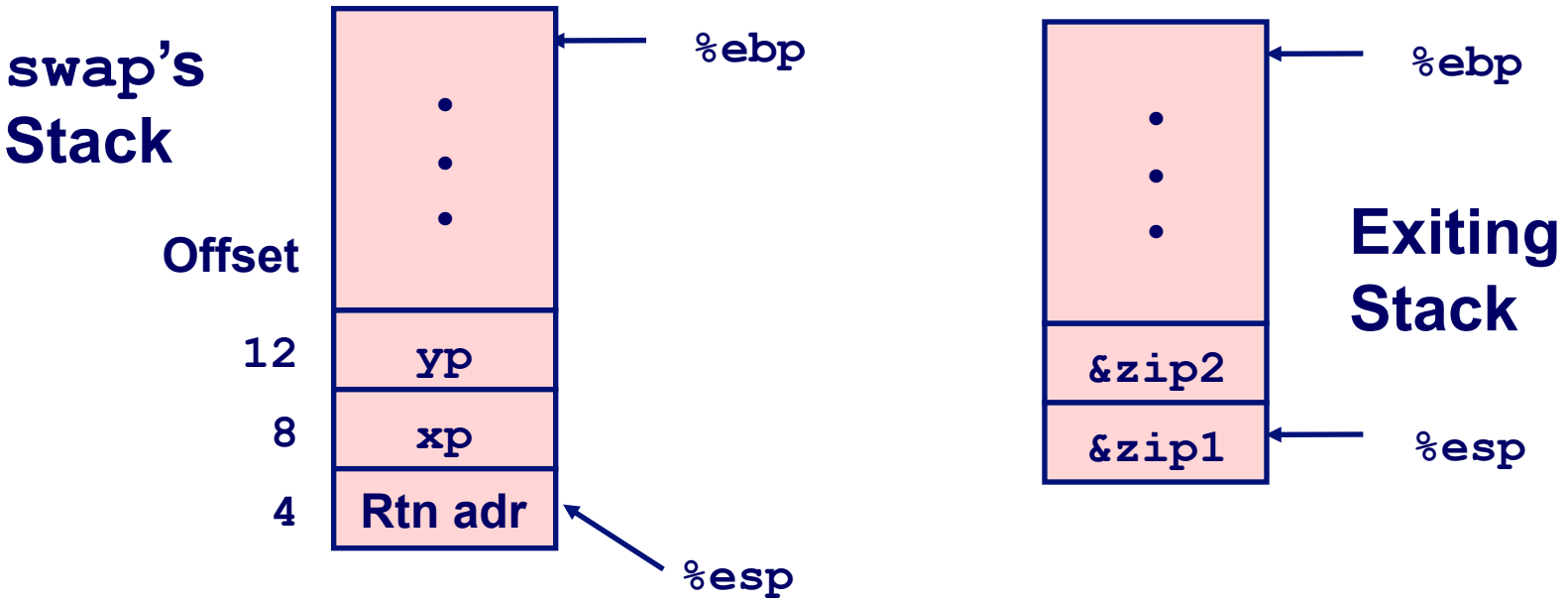
swap Finish #3






```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap Finish #4




```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Observation

- Saved & restored register `%ebx`
- Didn't do so for `%eax`, `%ecx`, or `%edx`

Revisiting swap

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Setup

Save old %ebp of caller frame
Set new %ebp for callee (current) frame
Save state of %ebx register from caller

Body

Retrieve parameter yp from caller frame
Retrieve parameter xp from caller frame

Perform swap

Replaced by

```
leave
```

Finish

Restore the state of caller's %ebx register
Set stack pointer to bottom of callee frame (%ebp)
Restore %ebp to original state

Pop return address from stack to %eip

Local variables

Where are they in relation to ebp?

- Stored "above" %ebp (at lower addresses)

How are they preserved if the current function calls another function?

- Compiler updates %esp beyond local variables before issuing "`call`"

What happens to them when the current function returns?

- Are lost (i.e. no longer valid)

What's the matter with this?

List two problems with the following code

```
int *foo(.. .)
{
    int *val;
    ...
    *val = 34;
    return val;
}
```

Exercise

```
int *func(int x)
{
  int n;
  n = x;
  return &n;
}
```

Will the C compiler give you an error on this function?

What will happen when it returns?

What if the pointer it returns is dereferenced?

Exercise

```
int *func(int x)
{
  int n;
  n = x;
  return &n;
}
```

Will the C compiler give you an error on this function?

- A warning is given

What will happen when it returns?

- Returns an address that is no longer part of the stack

What if the pointer it returns is dereferenced?

- If pointer is outside of valid stack page, segmentation fault
- Otherwise, returns whatever is at location

Register Saving Conventions

When procedure `foo` calls `who`:

`foo` is the *caller*, `who` is the *callee*

Can a Register be Used for Temporary Storage?

```
foo:
    • • •
    movl $15213, %edx
    call who
    addl %edx, %eax
    • • •
    ret
```

```
who:
    • • •
    movl 8(%ebp), %edx
    addl $91125, %edx
    • • •
    ret
```

Contents of register `%edx` overwritten by `who`

Register Saving Conventions

When procedure `foo` calls `who`:

`foo` is the *caller*, `who` is the *callee*

Can a Register be Used for Temporary Storage?

Conventions

"Caller Save"

- Caller saves temporary in its frame before calling

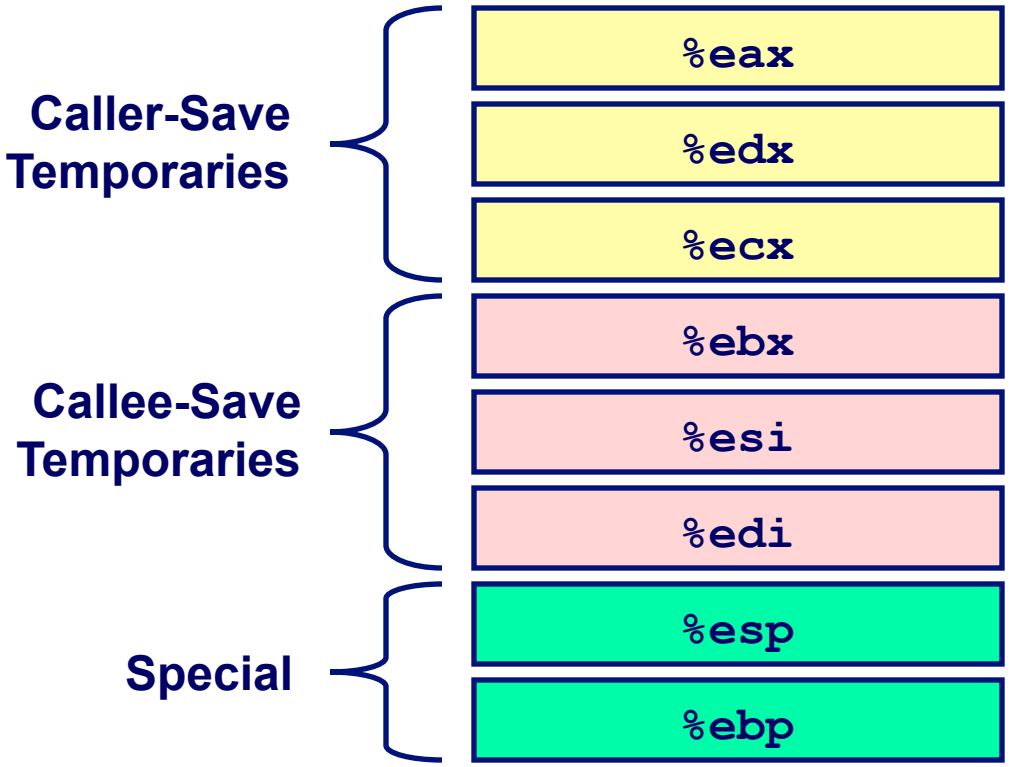
"Callee Save"

- Callee saves temporary in its frame before using

IA32 Register Usage

Integer Registers

- Two have special uses
  - `%ebp`, `%esp`
- Three managed as callee-save
  - `%ebx`, `%esi`, `%edi`
  - Old values should be saved on stack prior to using
- Three managed as caller-save
  - `%eax`, `%edx`, `%ecx`
  - Do what you please, but expect any callee to do so, as well
- Return value in %eax



IA32 register usage

Intel publishes these register usage conventions.

Compiler writers are expected to comply.

Why shouldn't each compiler writer make up their own conventions?

Practice problem 3.30

What does this code do?

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

What does the call do to the stack?

What is the value of %eax?

Where'd the ret go?

What would this be useful for?

Recursive Procedures

Since each `call` results in a new stack frame, recursive calls become natural

A recursive call is just like any other call, as far as IA32 assembly code is concerned

- Of course, the a recursive algorithm needs a termination condition, but that's the programmer's problem

http://thefengs.com/wuchang/courses/cs201/class/08/stack.c

Recursive Factorial

```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

x! = (x-1)! * x

Registers

- `%eax` used without first saving
- `%ebx` saved at beginning & restored at end

%eax is pushed before call to rfact and never popped. Is that a problem?

```
    .globl rfact
    .type  rfact,@function
rfact:
    pushl  %ebp
    movl   %esp,%ebp
    pushl  %ebx
    movl   8(%ebp),%ebx
    cmpl   $1,%ebx
    jle    .L78
    leal   -1(%ebx),%eax
    pushl  %eax
    call   rfact
    imull  %ebx,%eax
    jmp    .L79
    .align 4
.L78:
    movl   $1,%eax
.L79:
    movl   -4(%ebp),%ebx
    movl   %ebp,%esp
    popl   %ebp
    ret
```

Practice problem 3.33

```
int proc(void) {
        int x,y;
        scanf("%x %x", &y, &x);
        return (x-y)
}
```

```
1 proc:
2       pushl %ebp
3       movl %esp,%ebp
4       subl $24,%esp
5       addl $-4,%esp
6       leal –4(%ebp),%eax
7       pushl %eax
8       leal –8(%ebp),%eax
9       pushl %eax
10      pushl $.LC0          /* "%x %x" */
11      call scanf
12      movl –8(%ebp),%eax
13      movl –4(%ebp),%edx
14      subl %eax,%edx
15      movl %edx,%eax
16      movl %ebp,%esp
17      popl %ebp
18      ret
```

%esp initially 0x800040
%ebp initially 0x800060
scanf reads 0x46 and 0x53 from stdin
"%x %x" stored at 0x300070

What value does %ebp get set to in line 3?

At what addresses are local variables x and y stored?

What is the value of %esp after line 10?

Practice problem 3.33

```
int proc(void) {
        int x,y;
        scanf("%x %x", &y, &x);
        return (x-y)
}
```

```
1 proc:
2       pushl %ebp
3       movl %esp,%ebp
4       subl $24,%esp
5       addl $-4,%esp
6       leal –4(%ebp),%eax
7       pushl %eax
8       leal –8(%ebp),%eax
9       pushl %eax
10      pushl $.LC0          /* "%x %x" */
11      call scanf
12      movl –8(%ebp),%eax
13      movl –4(%ebp),%edx
14      subl %eax,%edx
15      movl %edx,%eax
16      movl %ebp,%esp
17      popl %ebp
18      ret
```

%esp initially 0x800040
%ebp initially 0x800060
scanf reads 0x46 and 0x53 from stdin
"%x %x" stored at 0x300070

What value does %ebp get set to in line 3?

0x80003C

At what addresses are local variables x and y stored?

0x800038 and 0x800034

What is the value of %esp after line 10?

0x800040 – 4 – 24 – 4 – 4 – 4 – 4
0x800040 – 0x2C = 0x800014

Practice problem 3.33

```
int proc(void) {
        int x,y;
        scanf("%x %x", &y, &x);
        return (x-y)
}
```

```
1 proc:
2       pushl %ebp
3       movl %esp,%ebp
4       subl $24,%esp
5       addl $-4,%esp
6       leal –4(%ebp),%eax
7       pushl %eax
8       leal –8(%ebp),%eax
9       pushl %eax
10      pushl $.LC0          /* "%x %x" */
11      call scanf
12      movl –8(%ebp),%eax
13      movl –4(%ebp),%edx
14      subl %eax,%edx
15      movl %edx,%eax
16      movl %ebp,%esp
17      popl %ebp
18      ret
```

%esp initially 0x800040
%ebp initially 0x800060
scanf reads 0x46 and 0x53 from stdin
"%x %x" stored at 0x300070

Draw a diagram of the stack frame for proc right after scanf returns and indicate the regions of the stack frame that are not used.

0x80003C	0x800060
0x800038	0x53
0x800034	0x46
0x800030	
0x80002C	
0x800028	
0x800024	
0x800020	
0x80001C	0x800038
0x800018	0x800034
0x800014	0x300070
0x800010	

Other calling conventions

Previous slides describe the “cdecl” calling convention

- Two other calling conventions “stdcall” and “fastcall”

cdecl

- Caller pushes arguments on stack before call
- Caller clears arguments off stack after call

stdcall

- Standard Win 32 API
- Same as cdecl except …
- Caller pushes arguments on stack before call
- Callee clears arguments off stack before returning from call
  - Saves some instructions since callee is already restoring the stack at the end of the function

Other calling conventions

fastcall

- Save memory operations by passing arguments in registers
- Many versions based on the compiler
  - Combining code that has been compiled with different conventions ugly
  - Must declare in source code if calling convention is different for function
- Microsoft implementation
  - First two arguments passed in registers %ecx and %edx
  - Code written on Windows must deal with stdcall and fastcall conventions
- Linux
  - Must declare in function prototype which calling convention is being used
  - http://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html

x86-64

- Function arguments placed in registers
  - 4 integer register arguments
  - 4 floating point register arguments
  - Rest on stack

Other calling conventions

thiscall

- Used for C++
- Linux
  - Same as cdecl, but first argument assumed to be “this” pointer
- Windows/Visual C++
  - “this” pointer passed in %ecx
  - Callee cleans the stack when arguments are not variable length
  - Caller cleans the stack when arguments are variable length

More information

- http://www.programmersheaven.com/2/Calling-conventions

Extra slides

Summary

The Stack Makes Recursion Work

- Private storage for each *instance* of procedure call
  - Instantiations don't clobber each other
  - Addressing of locals + arguments can be relative to stack positions
- Can be managed by stack discipline
  - Procedures return in inverse order of calls

IA32 Procedures Combination of Instructions + Conventions

- Call / Ret instructions
- Register usage conventions
  - Caller / Callee save
  - `%ebp` and `%esp`
- Stack frame organization conventions

The Towers of Hanoi

```
hanoi (int N, int src, int aux, int dst)
{
  if (N == 0) return;
  hanoi(N-1, src, dst, aux);
  move(src, dst);
  hanoi(N-1, aux, src, dst);
}
```

where “move” does the graphical or physical moving of a ring to another post.

Does this code really work?