HARVARD
School of Engineering
and Applied Sciences

Assembly Programming: More control flow and Procedures

CS61, Lecture 4

Prof. Stephen Chong

September 14, 2010

Announcements

- Sections start this week
  - Everyone should have been assigned to a section
  - Email cs61-staff@eecs.harvard.edu if not
- Office hours start this week
  - See web site for details
- Lab 1 due in one week!
- Auditors: Email cs61-staff@eecs.harvard.edu to let us know
  - Otherwise we will hunt you down for missed labs and quizzes

Announcements

- Name tags
  - Fill out a name tag, put it in front of you!
  - Leave after class, and collect at start of next class.

- Weekend server outages
  - cs61.seas was offline for a few hours on the weekend
  - We're monitoring this

Topics for today

- Control flow ctd.
  - Loops
  - Switch statements
- Procedures
  - Implementing procedure calls
  - Using the stack
  - Storing and accessing local variables
  - Saving and restoring registers
  - Recursive procedures

Last lecture

- Condition flags
  - Zero Flag, Carry Flag, Overflow Flag, Sign Flag
  - Updated by every arithmetic operation and `cmpl` and `testl` instructions
- Conditional jumps
  - E.g., `jz`, `je`, `jne`, `jg`, `jle`, …
  - Update the instruction pointer if condition flags set appropriately
- Control flow
  - Conditional jumps to implement `if` statements

Implementing loops

```
int fact_do(int x)
{
  int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);

  return result;
}
```

```
int fact_goto(int x)
{
  int result = 1;
Loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto Loop;
  return result;
}
```

- Two equivalent programs to compute factorial
- Goto version uses backwards branch to continue loop
  - Only takes branch if while condition (x > 1) is true

Do-while loop compilation

```
int fact_goto(int x)
{
  int result = 1;
Loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto Loop;
  return result;
}
```

```
fact_goto:
    pushl %ebp           # Setup
    movl %esp,%ebp       # Setup
    movl $1,%eax         # eax = 1
    movl 8(%ebp),%edx    # edx = x

L11:
    imull %edx,%eax      # result *= x
    decl %edx            # x--
    cmpl $1,%edx         # Compare x : 1
    jg L11               # if > goto loop

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

While loops version 1

C code

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {


    result *= x;
    x = x-1;
  };

  return result;
}
```

Goto version 1

```
int fact_while_goto(int x)
{
  int result = 1;
Loop:
  if (!(x > 1))
    goto Done;
  result *= x;
  x = x-1;
  goto Loop;
Done:
  return result;
}
```

- How is this different from the do-while version?

While loops version 2

C code

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {


    result *= x;
    x = x-1;
  };

  return result;
}
```

Goto version 2

```
int fact_while_goto2(int x)
{
  int result = 1;
  if (!(x > 1))
    goto Done;
Loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto Loop;
Done:
  return result;
}
```

- Historically used by GCC
- Uses same inner loop as do-while version
- Guards loop entry with extra test

While loops version 2

While version

```
while (test)
   body
```

Do-While version

```
if (!test) goto done;
do
   body
while (test)

done:
```

Goto version

```
if (!test) goto done;
loop:
   body
if (test) goto loop;

done:
```

Compiling for loops

For version

```
for (init; test; update)
  body
```

While version

```
init
while (test)
  body
  update
```

Do-While version

```
init
if (!test) goto done;
do
  body
  update
while (test)

done:
```

…

Switch statements

- Switch statements can be complex…
  - Many cases to consider
  - Can have "fall through"
    - No break at end of case 2
  - Can have missing cases
  - Can have `default` case
- How to compile?
  - Series of conditionals?
    - Works, but a lot of code, and expensive
  - **Jump table**
    - List of jump targets indexed by x
    - Less code, and fast!

```
int switchexample(int x) {
    int y;
    switch(x) {
      case 1:
        y = x; break;
      case 2:
        y = 2*x;
        /* Fall through! */
      case 3:
        y = 3*x; break;
      /* No case 4! */
      case 5:
         y = 5*x; break;
      default:
         y = x; break;
    }
    return y;
}
```

Jump table structure

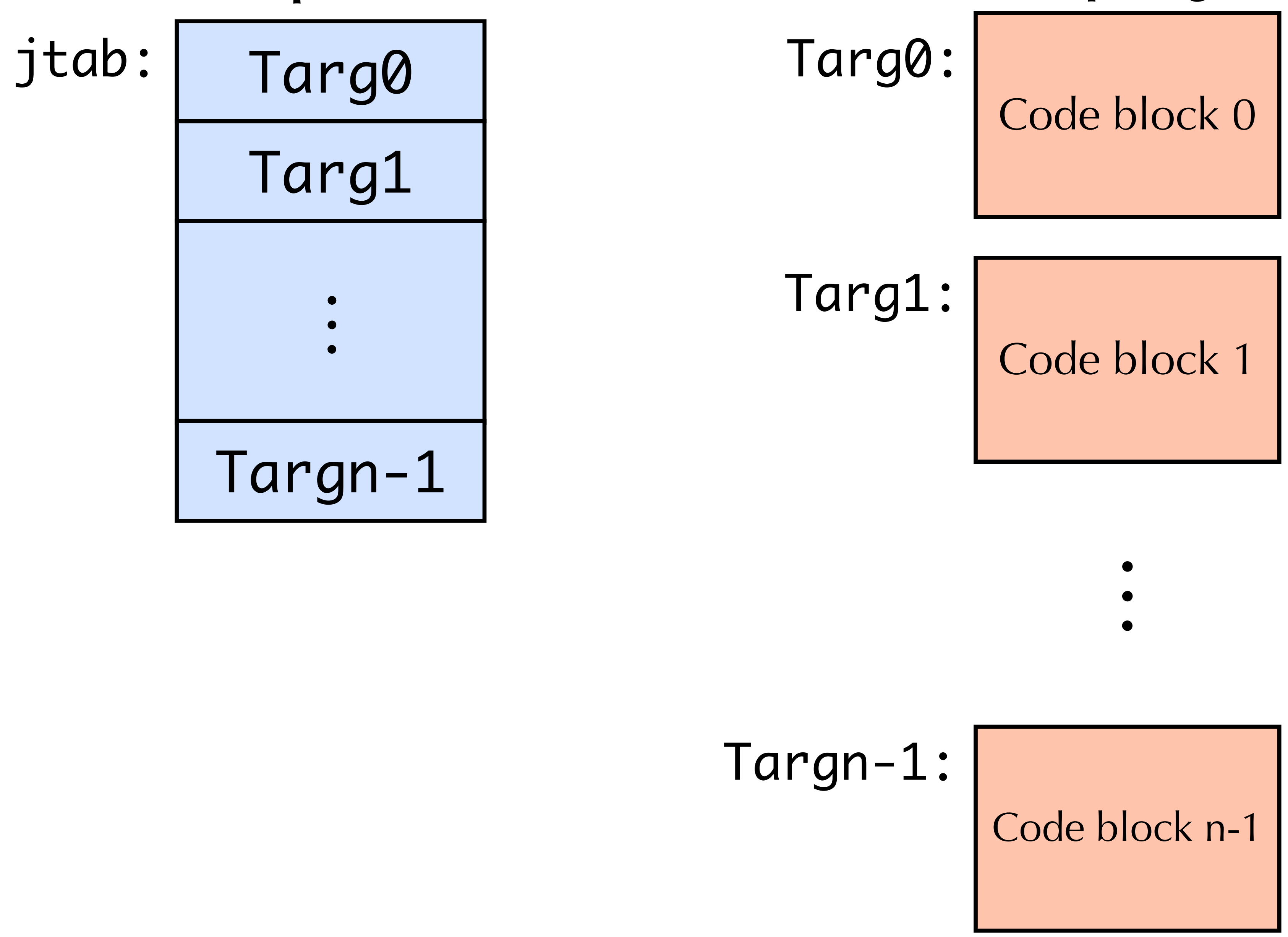
Switch code

```
switch(x) {
  case val_0:
    Block_0
  case val_1:
    Block_1
  ...
  case val_n-1:
    Block_n-1
}
```

Approximate translation

```
target = jtab[x];
goto *target;
```

Jump table **Jump targets**



Using a jump table

```
int switchexample(int x) {
    int y;
    switch(x) {
      case 1:  y =   x; break;
      case 2:  y = 2*x;
      case 3:  y = 2*x; break;
      /*no case 4*/
      case 5:
      case 6:  y = 2*x; break;
      default: y =   0; break;
    }
    return y;
}
```

`jmp *`*src* is an **indirect jump**. Always jumps to the address that *src* evaluates to.

Why multiply x by 4?

```
pushl   %ebp                  # Setup
movl    %esp, %ebp
subl    $16, %esp
cmpl    $6, 8(%ebp)           # Check if 'x' is > 6
ja      .L38                  # If so, jump to .L38 (default case)
movl    8(%ebp), %eax         # %eax = x
sall    $2, %eax              # Shift left by 2 (multiply by 4)
movl    .L39(%eax), %eax      # Move jumptable[x] to eax
jmp     *%eax                 # Jump to this address
```

Using a jump table

```
int switchexample(int x) {
    int y;
    switch(x) {
      case 1:  y =   x; break;
      case 2:  y = 2*x;
      case 3:  y = 2*x; break;
      /*no case 4*/
      case 5:
      case 6:  y = 2*x; break;
      default: y =   0; break;
    }
    return y;
}
```

```
.L39:               # Jumptable starts here
     .long    .L38  #   Entry 0 is symbol .L38 (default)
     .long    .L34  #   Entry 1 is symbol .L34
     .long    .L35  #   Entry 2 is symbol .L35
     .long    .L36  #   Entry 3 is symbol .L36
     .long    .L38  #   Entry 4 is symbol .L38 (default)
     .long    .L37  #   Entry 5 is symbol .L37
     .long    .L37  #   Entry 6 is symbol .L37
```

```
    pushl   %ebp                # Setup
    movl    %esp, %ebp
    subl    $16, %esp
    cmpl    $6, 8(%ebp)         # Check if 'x' is > 6
    ja      .L38                # If so, jump to .L38 (default case)
    movl    8(%ebp), %eax       # %eax = x
    sall    $2, %eax            # Shift left by 2 (multiply by 4)
    movl    .L39(%eax), %eax    # Move jumptable[x] to eax
    jmp     *%eax               # Jump to this address
```

Using a jump table

```c
int switchexample(int x) {
    int y;
    switch(x) {
      case 1:  y =   x; break;
      case 2:  y = 2*x;
      case 3:  y = 2*x; break;
      /*no case 4*/
      case 5:
      case 6:  y = 2*x; break;
      default: y =   0; break;
    }
    return y;
}
```

```
.L39:                 # Jumptable starts here
   .long    .L38  #    Entry 0 is symbol .L38 (default)
   .long    .L34  #    Entry 1 is symbol .L34
   .long    .L35  #    Entry 2 is symbol .L35
   .long    .L36  #    Entry 3 is symbol .L36
   .long    .L38  #    Entry 4 is symbol .L38 (default)
   .long    .L37  #    Entry 5 is symbol .L37
   .long    .L37  #    Entry 6 is symbol .L37
```

```
.L34:                              # Case for Entry 1 (x == 1)
   movl     8(%ebp), %eax          # %eax = x
   movl     %eax, -4(%ebp)         # y = %eax
   jmp      .L40                   # Jump out of 'switch'
```

Topics for today

- Control flow ctd.
  - Loops
  - Switch statements
- Procedures
  - Implementing procedure calls
  - Using the stack
  - Storing and accessing local variables
  - Saving and restoring registers
  - Recursive procedures

Procedure calls

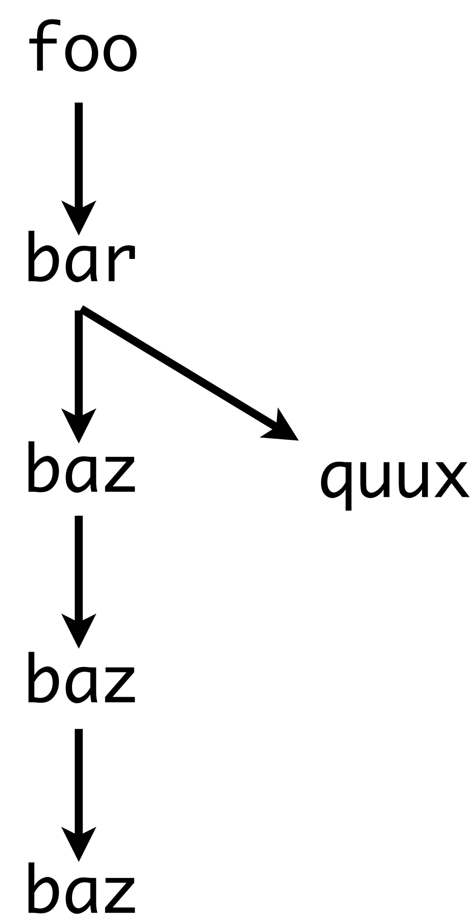
```
void foo(...) {
    ...
    bar();
    ...
}
```

```
void bar(...) {
    int x, y;
     x = baz();
     ...
     y = quux();
     ...
}
```

```
int baz(...) {
    int z;
    ...
    z = baz();
    ...
    return z;
}
```

```
int quux(...) {
    ...
    return 42;
}
```

Call chain

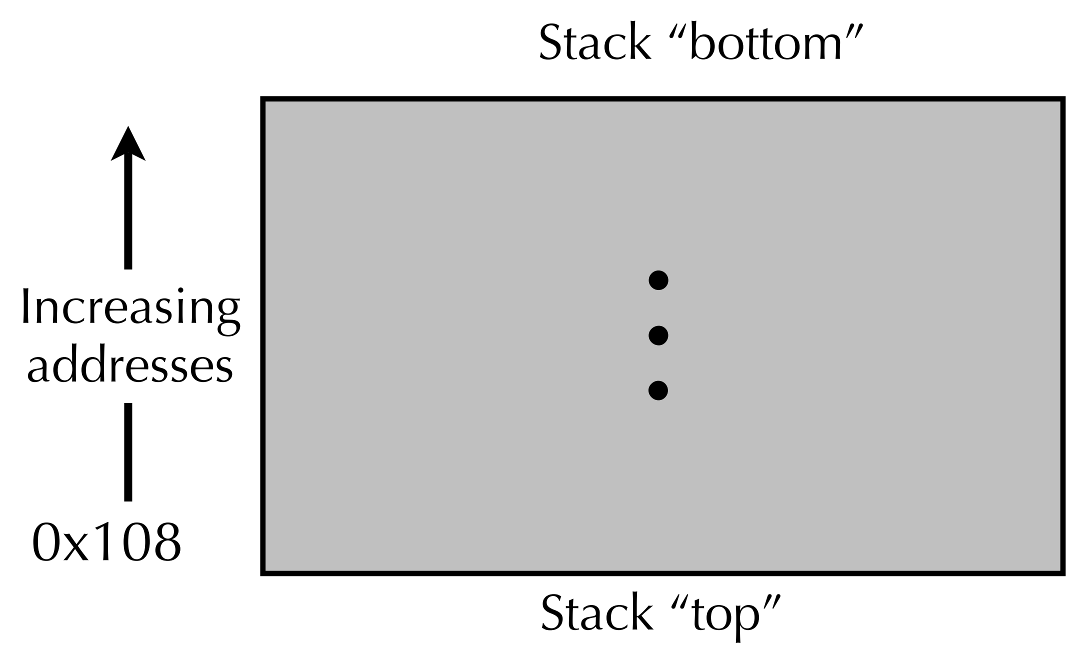


- How do we call procedures?
- Where do we store local variables (e.g., x,y,z)?
- How do we return values from procedures?
- How do we support recursion?

Stack

- Stack is used for handling function calls and local storage
  - Stores local variables, return address, saved registers, …
- Stack pointer `%esp` always holds address of top stack element
- Stack grows **downwards**!

`%eax`	`0x123`
`%edx`	`0`
`%esp`	`0x108`



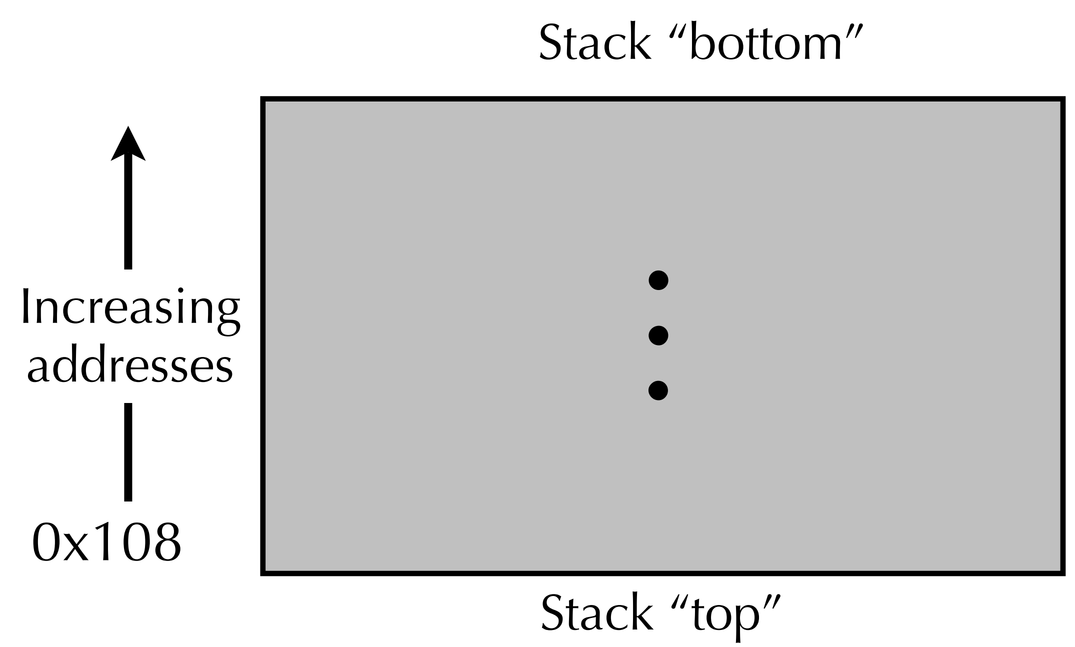
Pushing and popping

`%eax`	`0x123`
`%edx`	`0`
`%esp`	`0x108`

- Two data movement instructions for stack: `pushl` and `popl`
- `pushl` *src*
  - Push four bytes onto stack
  - Effect is

    $R[\texttt{\%esp}] \leftarrow R[\texttt{\%esp}] - 4$

    $M[R[\texttt{\%esp}]] \leftarrow src$
- E.g., `pushl %eax`



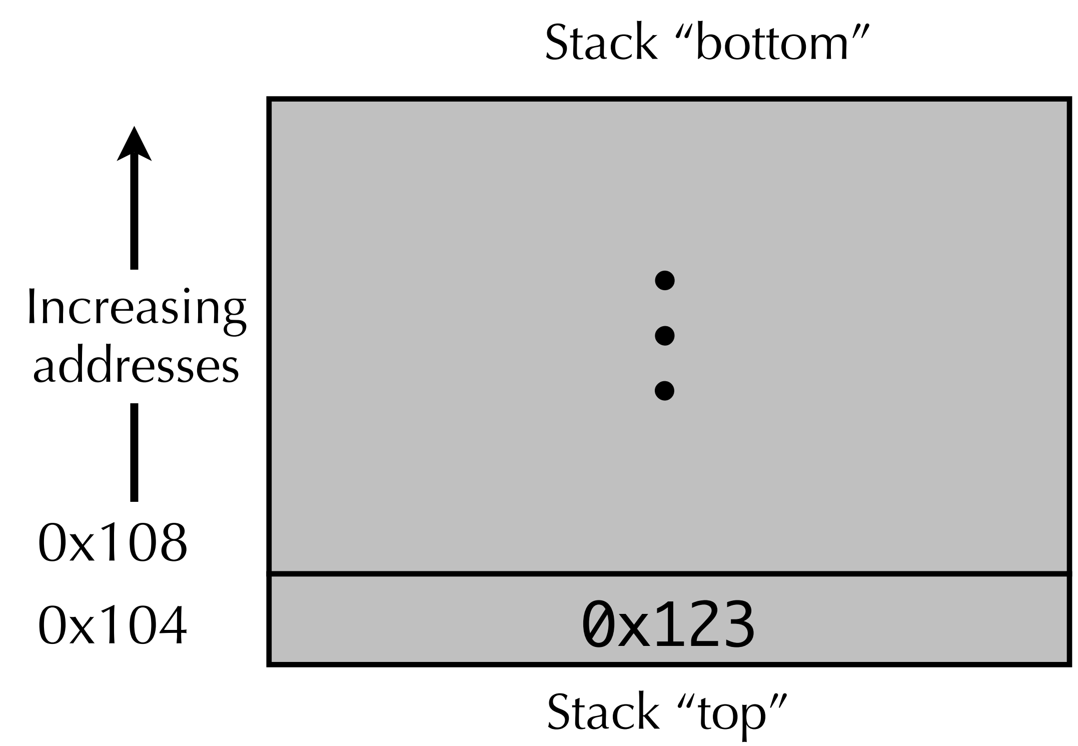
Pushing and popping

`%eax`	`0x123`
`%edx`	`0`
`%esp`	`0x104`

- Two data movement instructions for stack: `pushl` and `popl`
- `pushl` *src*
  - Push four bytes onto stack
  - Effect is

    $R[\texttt{\%esp}] \leftarrow R[\texttt{\%esp}] - 4$

    $M[R[\texttt{\%esp}]] \leftarrow src$
- E.g., `pushl %eax`



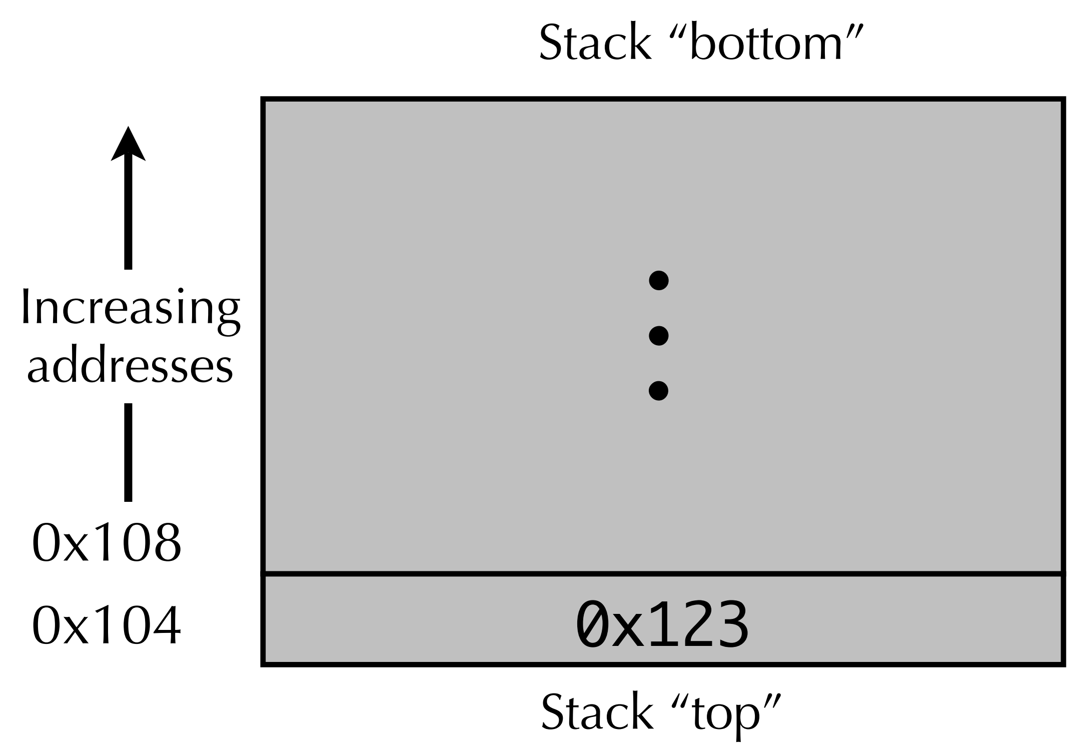
Pushing and popping

- `popl` *dest*
  - Pops four bytes from stack
  - Effect is

    dest ← M[R[`%esp`]]

    R[`%esp`] ← R[`%esp`] + 4
- E.g., `popl %edx`

%eax	0x123
%edx	0
%esp	0x104



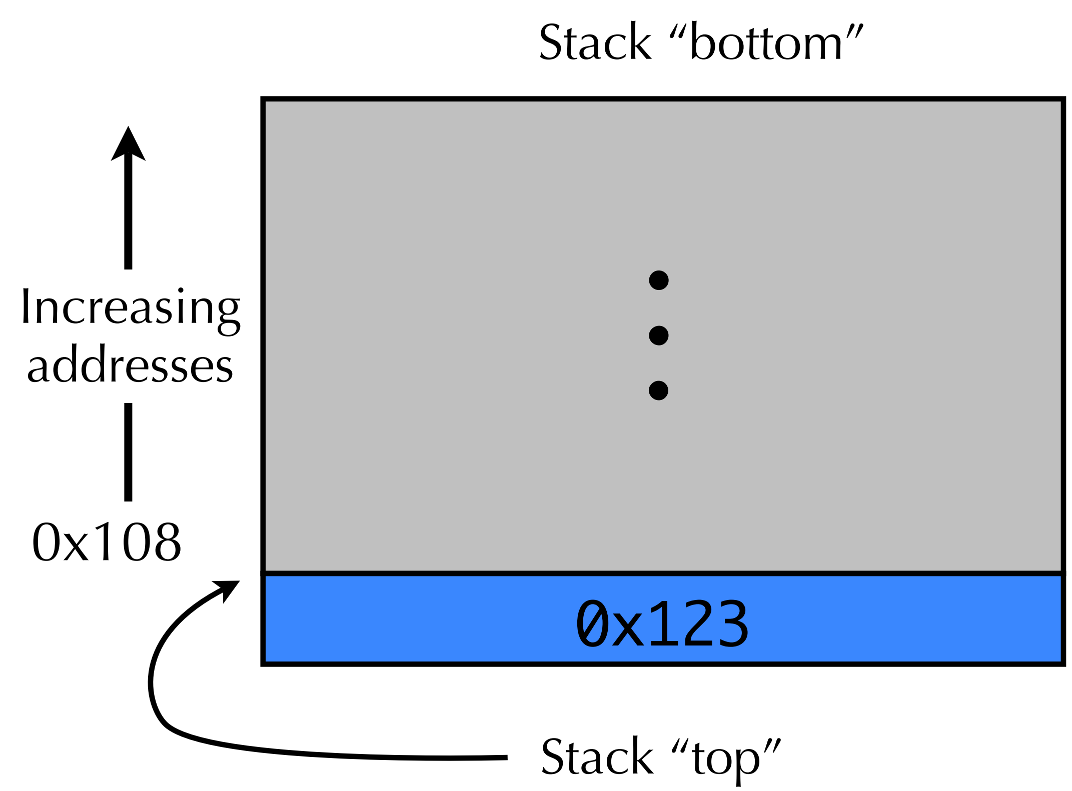
Pushing and popping

- `popl` *dest*
  - Pops four bytes from stack
  - Effect is

    dest ← M[R[`%esp`]]

    R[`%esp`] ← R[`%esp`] + 4
- E.g., `popl %edx`

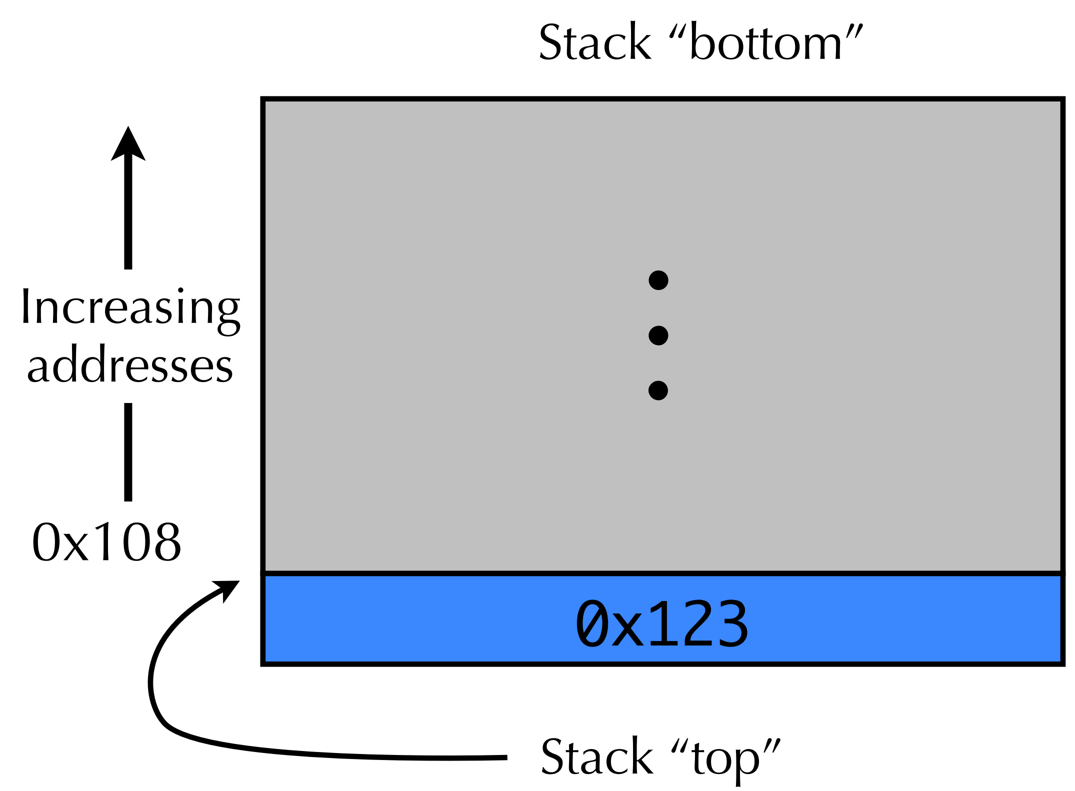
%eax	0x123
%edx	0x123
%esp	0x108



Examining the stack

%eax	0x123
%edx	0x123
%esp	0x108

- Can use `movl` to access and modify arbitrary values on the stack
  - No need to access just top element
  - Can "peek" at stack:
    - `movl 12(%esp), %eax`
  - Can "poke" stack:
    - `movl $0xdeadbeef, 12(%esp)`



Procedure control flow

- Stack is used to implement procedure call and return
- Procedure call
  - x86 instruction: `call` *address*
  - Pushes **return address** on stack, then jumps to *address*
  - What is the return address?
    - Address of instruction **after** the `call` instruction
    - E.g.,

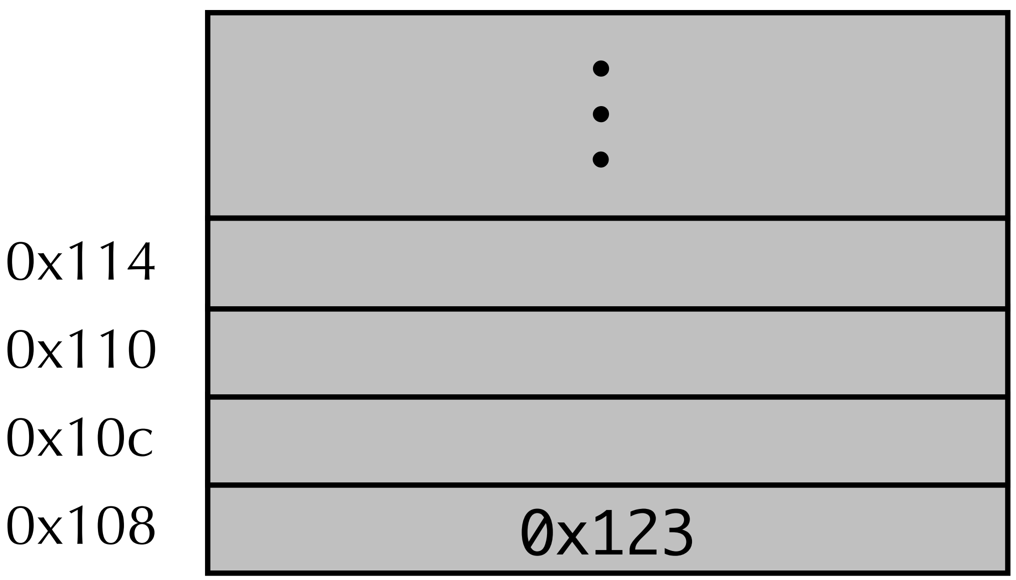
```
804854e: e8 3d 06 00 00    call   8048b90 <main>
8048553: 50                pushl  %eax
```

    - Return address is 0x8048553
- Procedure return
  - x86 instruction: `ret`
  - Pops return address from stack, and jumps to it

Procedure call example

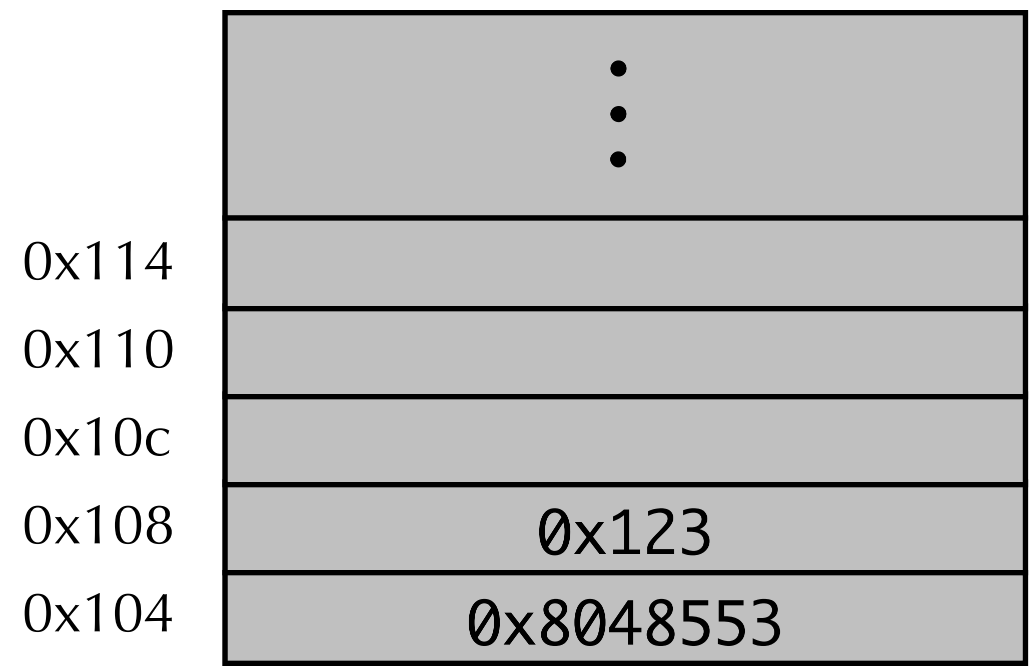
```
804854e: e8 3d 06 00 00    call   8048b90 <main>
8048553: 50                pushl  %eax
```

Before call



%esp	0x108
%eip	0x804854e

After call



%esp	0x104
%eip	0x8048b90

Procedure call example

```
804854e: e8 3d 06 00 00     call    8048b90 <main>
8048553: 50                 pushl   %eax
```

Before call

	⋮
0x114	
0x110	
0x10c	
0x108	0x123

%esp	0x108
%eip	0x804854e

After call

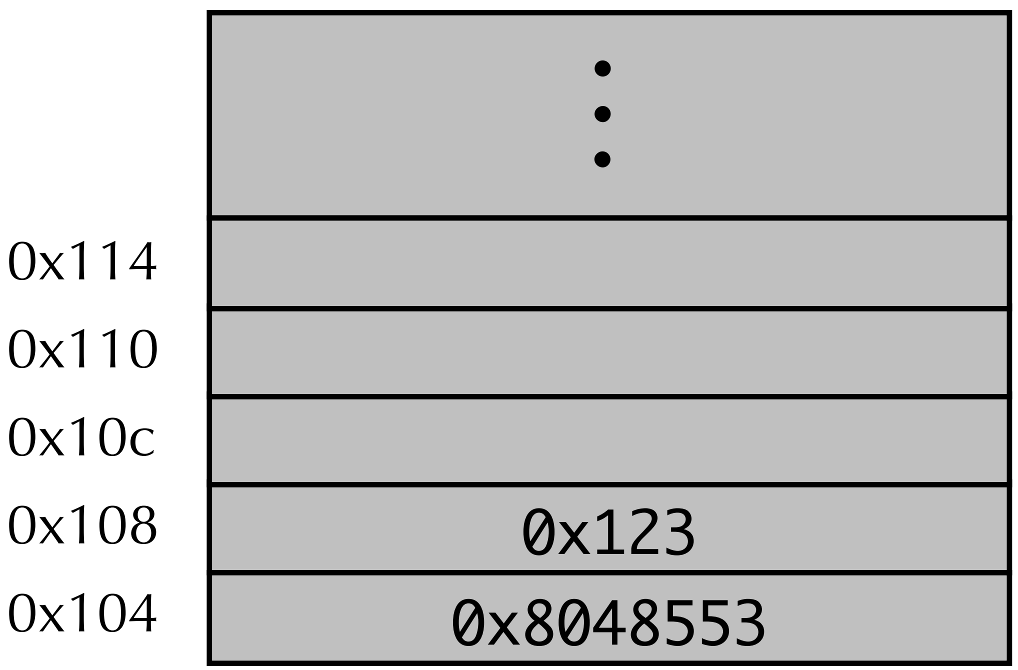
	⋮
0x114	
0x110	
0x10c	
0x108	0x123
0x104	0x8048553

%esp	0x104
%eip	0x8048b90

Procedure return example

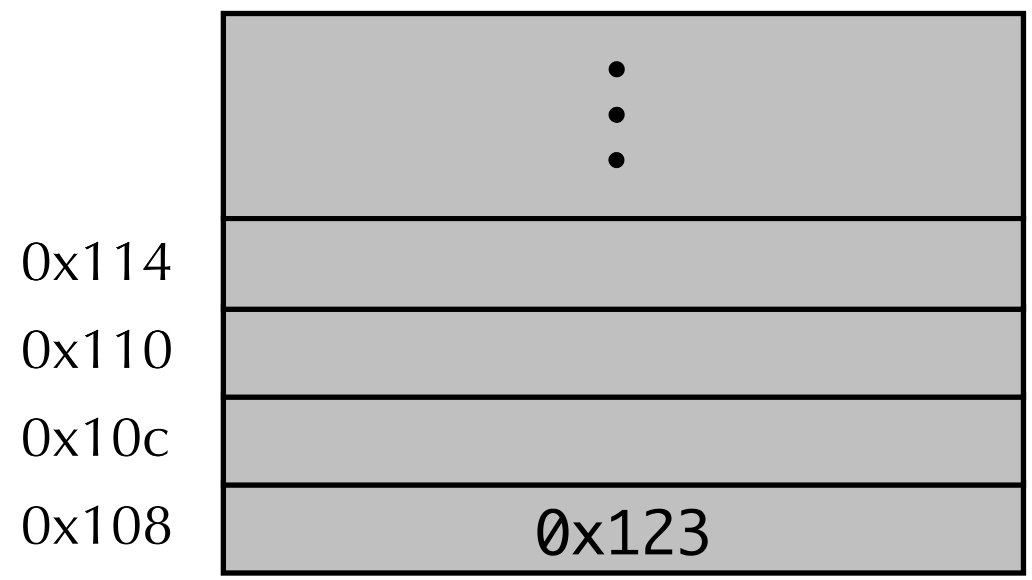
```
8048591: c3                    ret
```

Before return



%esp	0x104
%eip	0x8048b91

After return



%esp	0x108
%eip	0x8048553

Stack-based languages

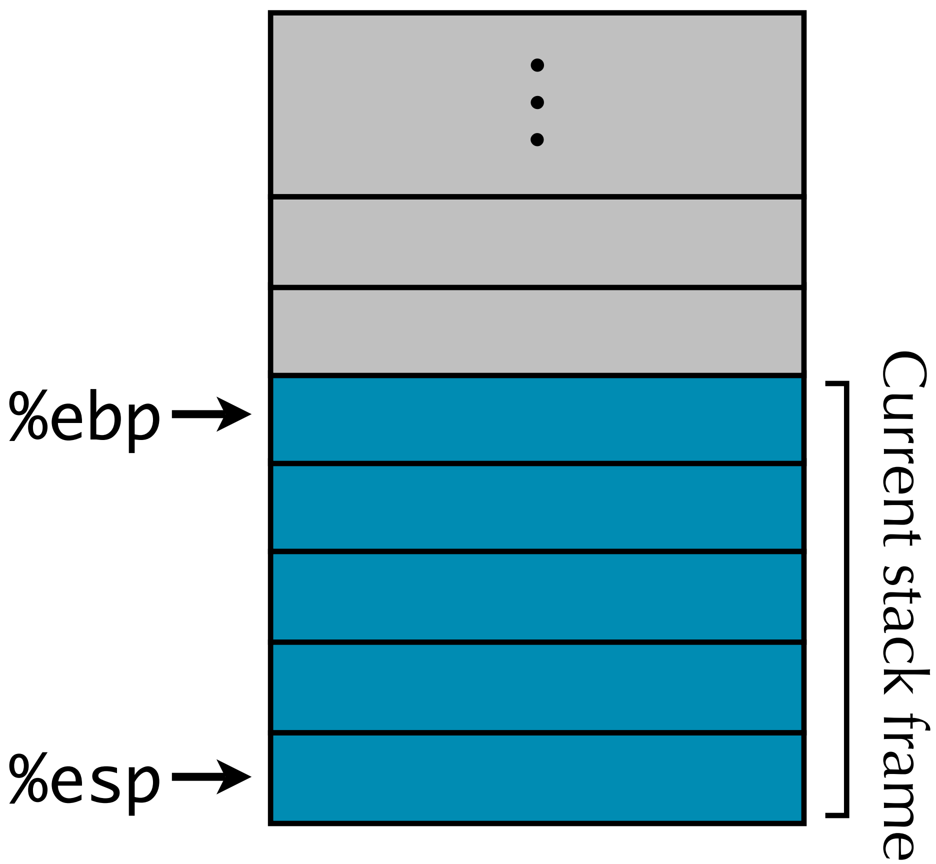
- Languages that support recursion
  - E.g., C, Pascal, Java
  - Must be able to support multiple instantiations of single procedure
    - Code must be **reenterant**

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

- Each invocation of a procedure has its own local state
  - Arguments to the procedure (e.g., `x`)
  - Local variables within the procedure (e.g., `rval`)
  - Return address
- Where are these stored?

Stack frame

- Each procedure invocation has an associated **stack frame**
  - The "chunk" of the stack for that procedure invocation
  - Contains local variables, arguments to functions, and return address
  - Needed from when procedure called to when it returns
- Stack discipline
  - Stack frame released when procedure returns
  - Callee must return before caller does
- Current stack frame described by two registers
  - `%ebp`: frame pointer
    - Points to base (or "bottom") of current stack frame
  - `%esp`: stack pointer
    - Points to stop of stack (i.e., top of current stack frame)



Stack frame example

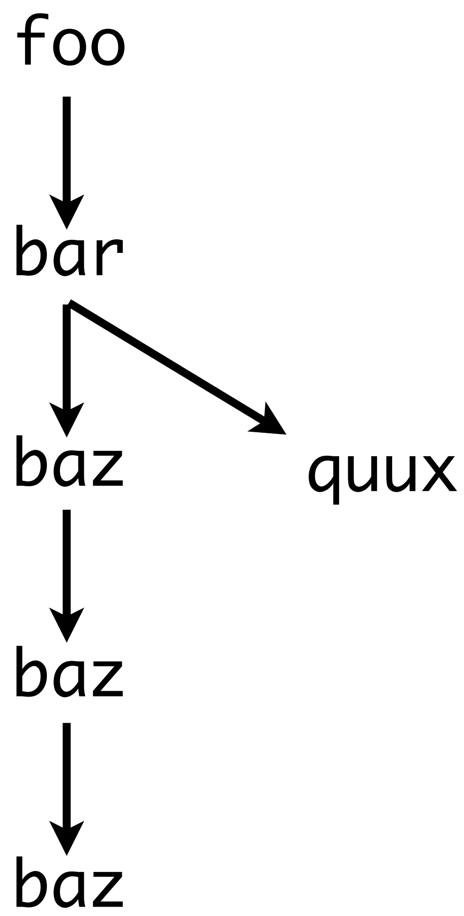
```
void foo(...) {
    ...
    bar();
    ...
}
```

```
void bar(...) {
   int x, y;
    x = baz();
    ...
    y = quux();
    ...
}
```

```
int baz(...) {
    int z;
    ...
    z = baz();
    ...
    return z;
}
```

```
int quux(...) {
    ...
    return 42;
}
```

Call chain



Stack frame example

Call chain

foo

```
void foo(...) {
    ...
    bar();
    ...
}
```

Stack frame example

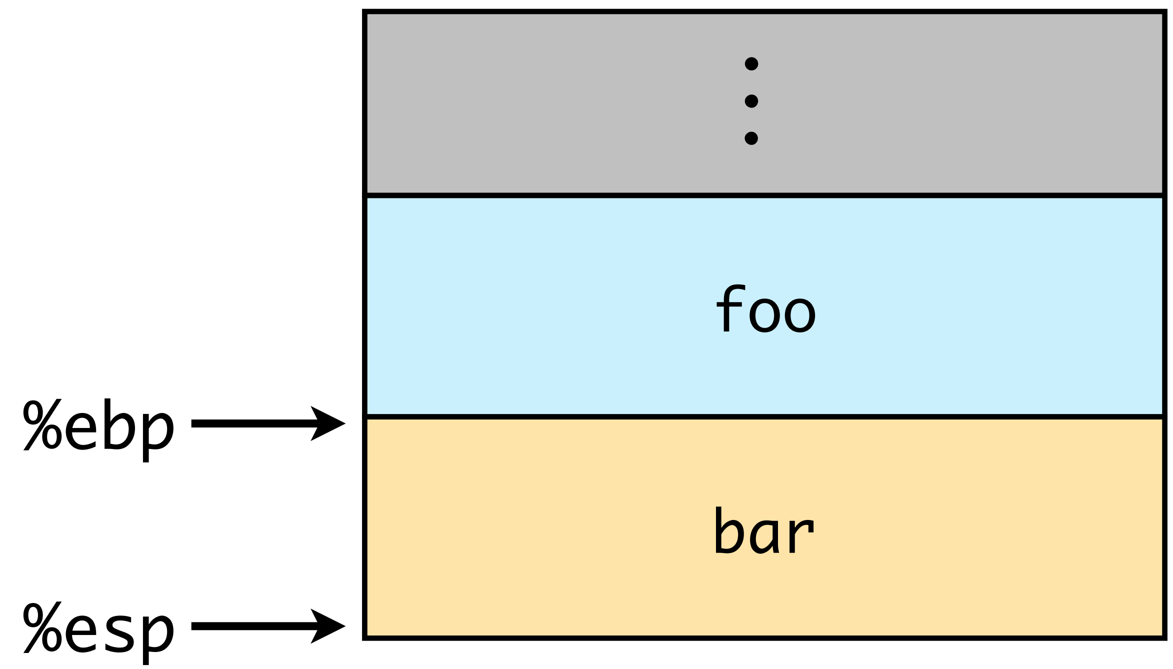
```
void bar(...) {
   int x, y;
    x = baz();
    ...
    y = quux();
    ...
}
```

Call chain

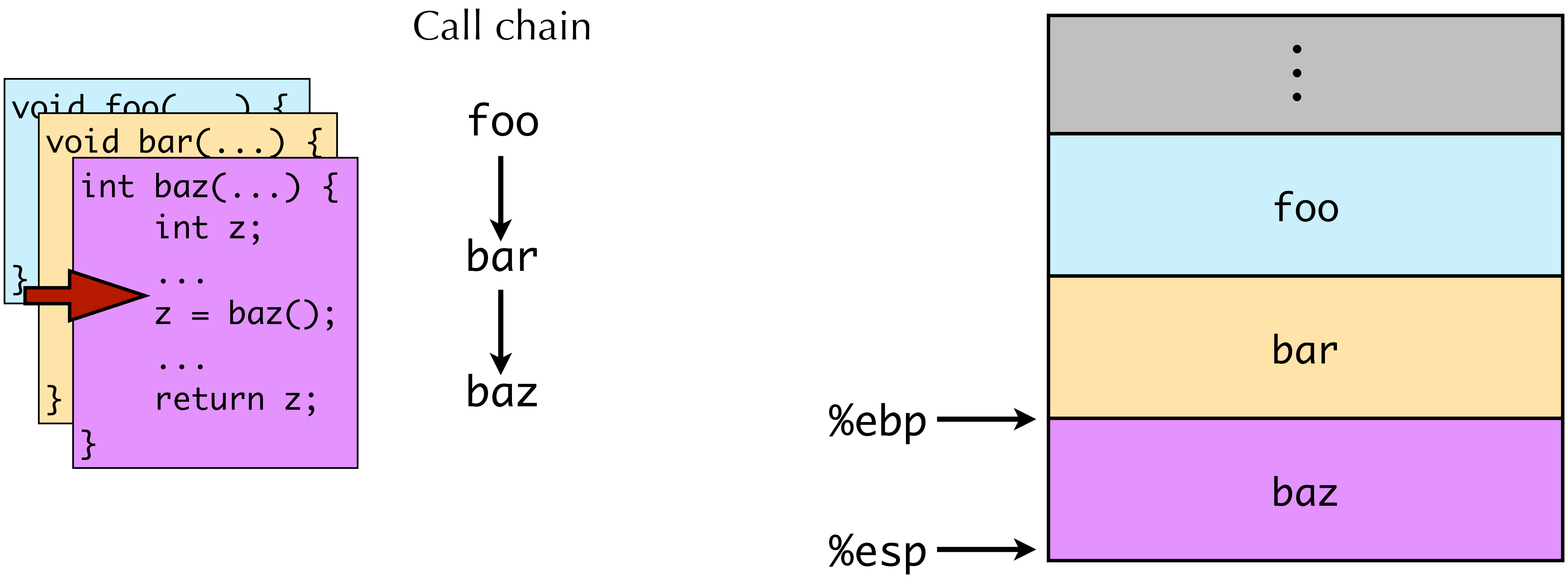
foo

↓

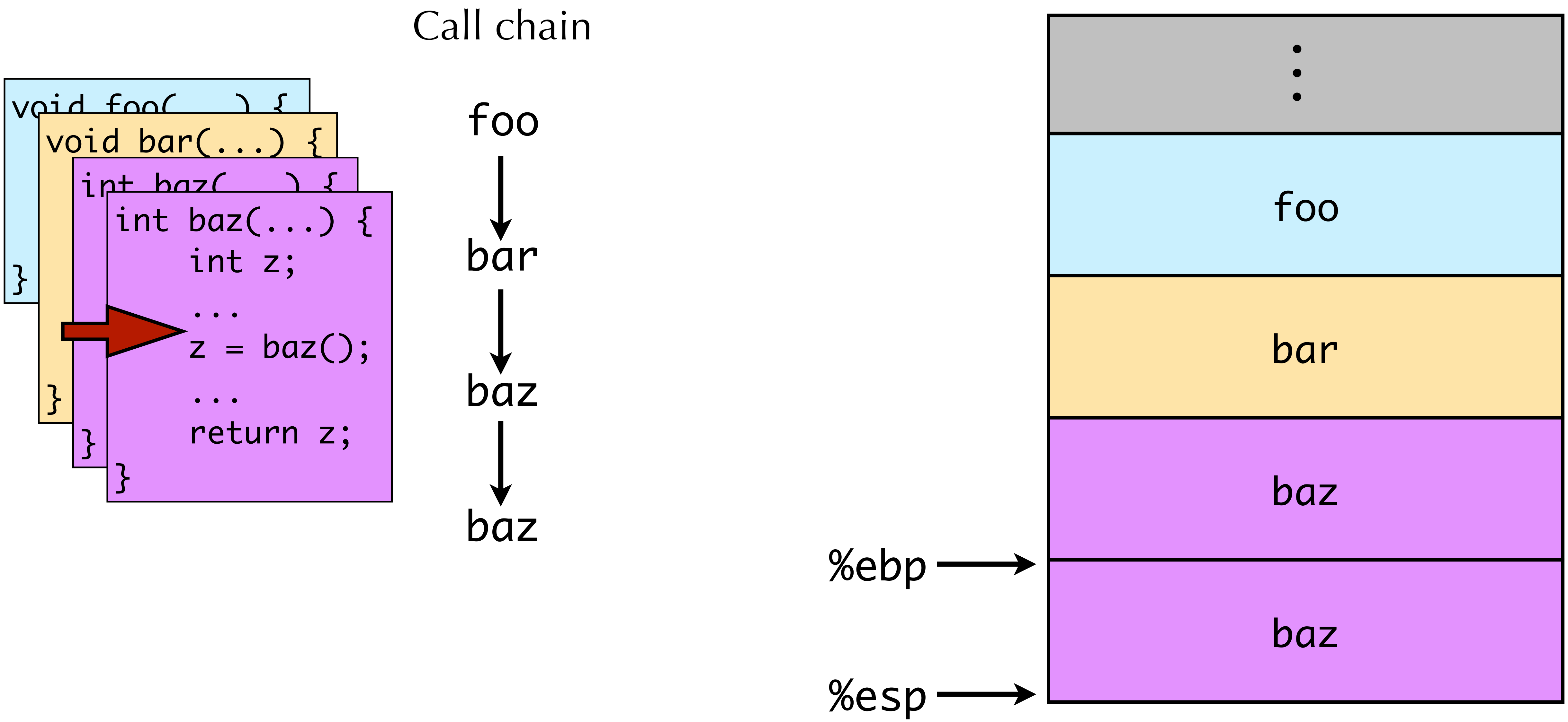
bar



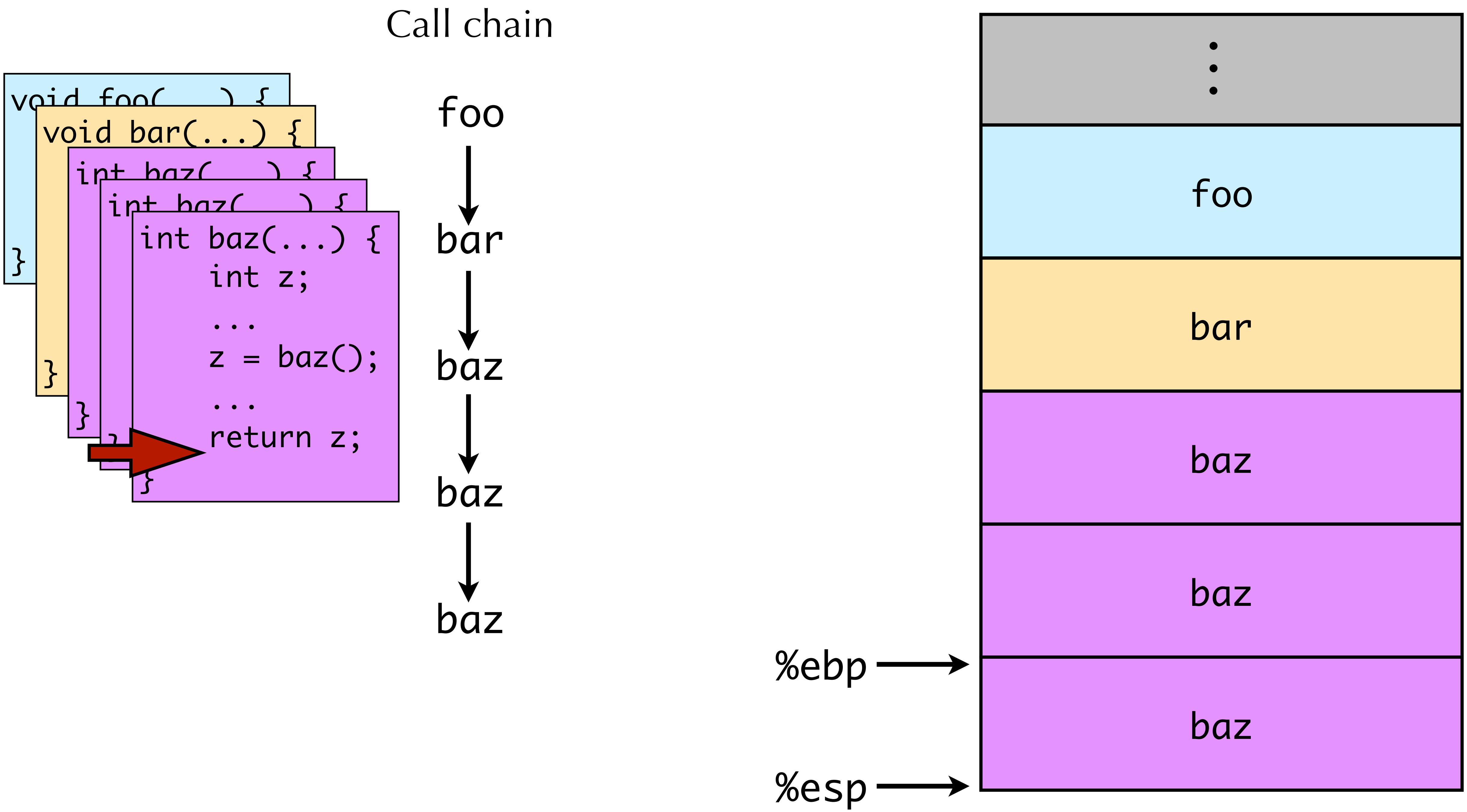
Stack frame example



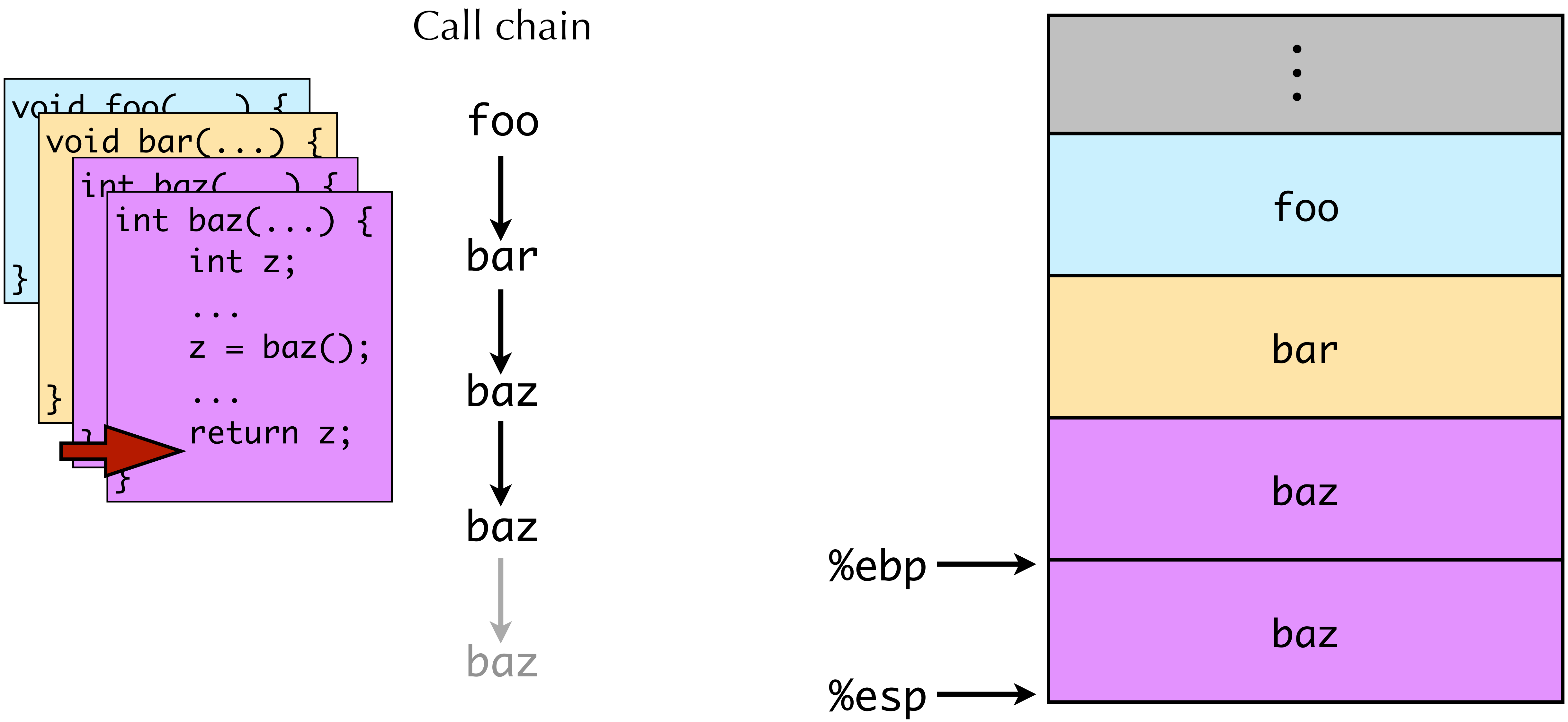
Stack frame example



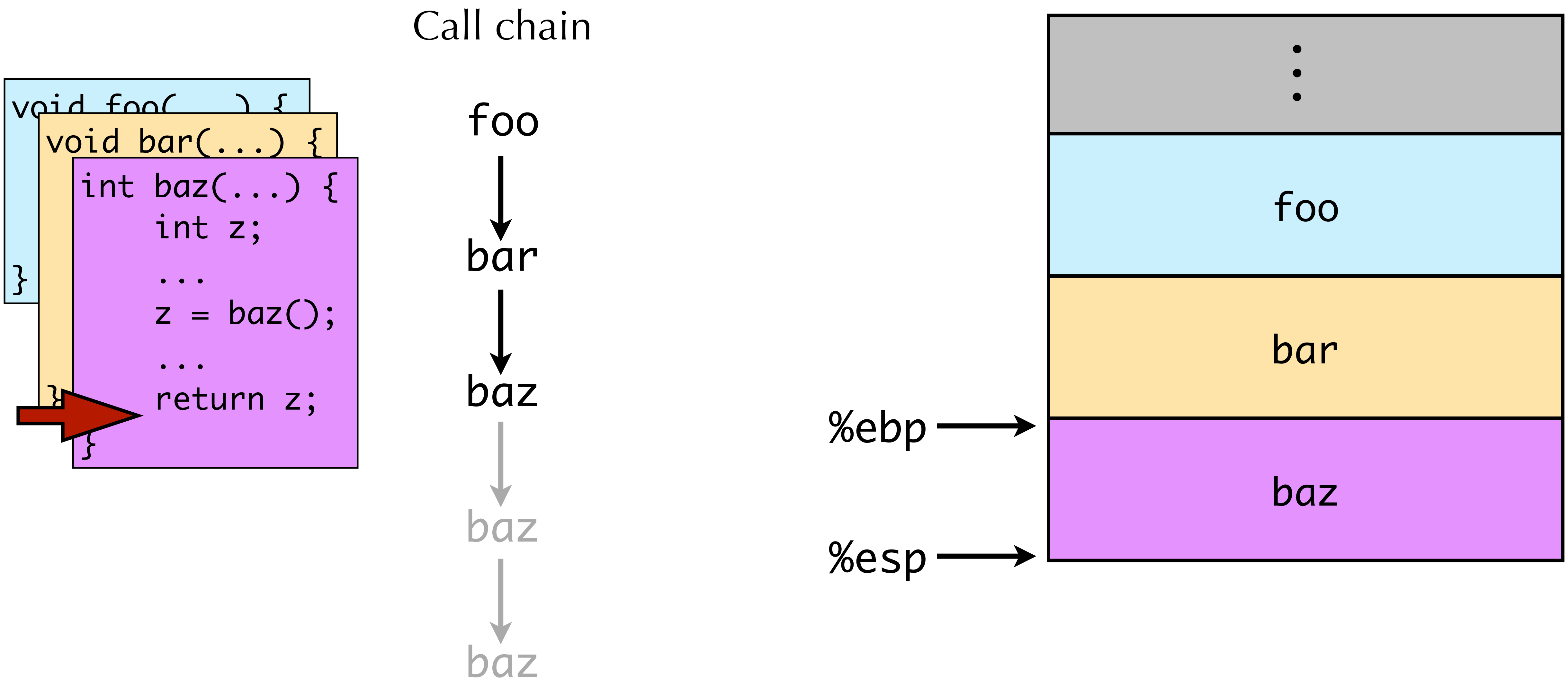
Stack frame example



Stack frame example



Stack frame example

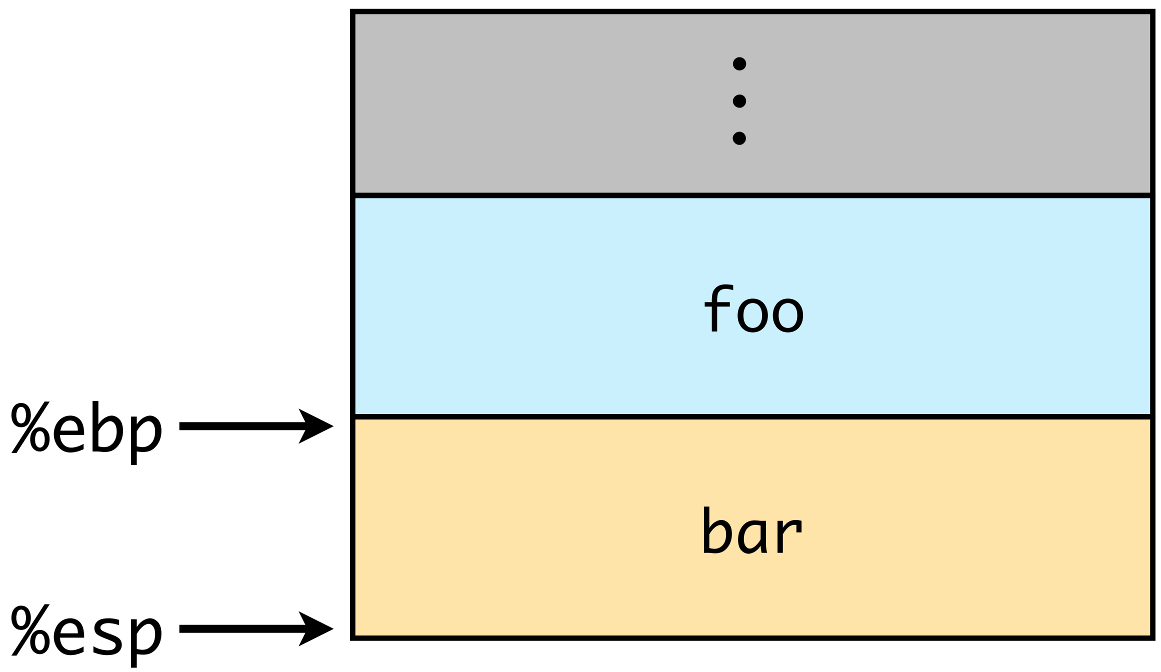


Stack frame example

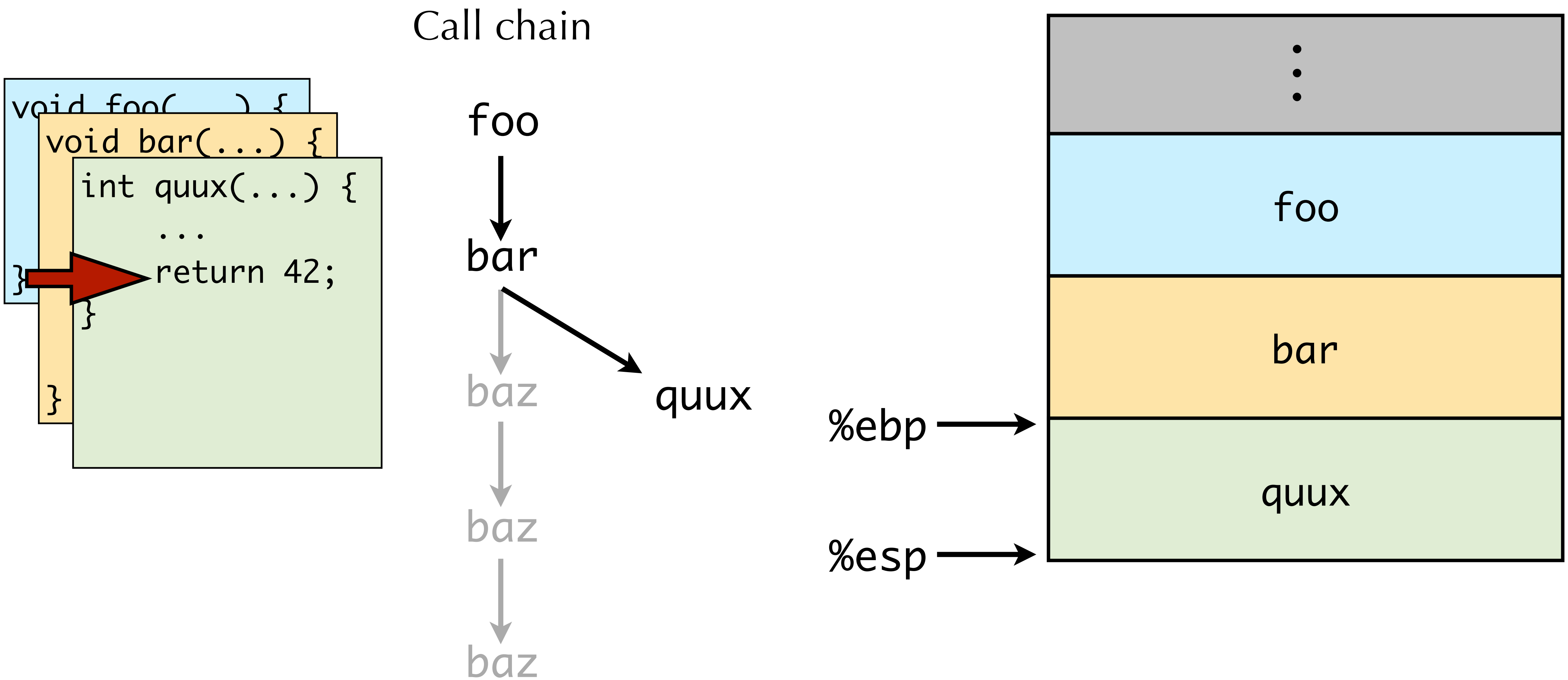
```
void bar(...) {
   int x, y;
    x = baz();
    ...
    y = quux();
    ...
}
```

Call chain

foo → bar → baz → baz → baz



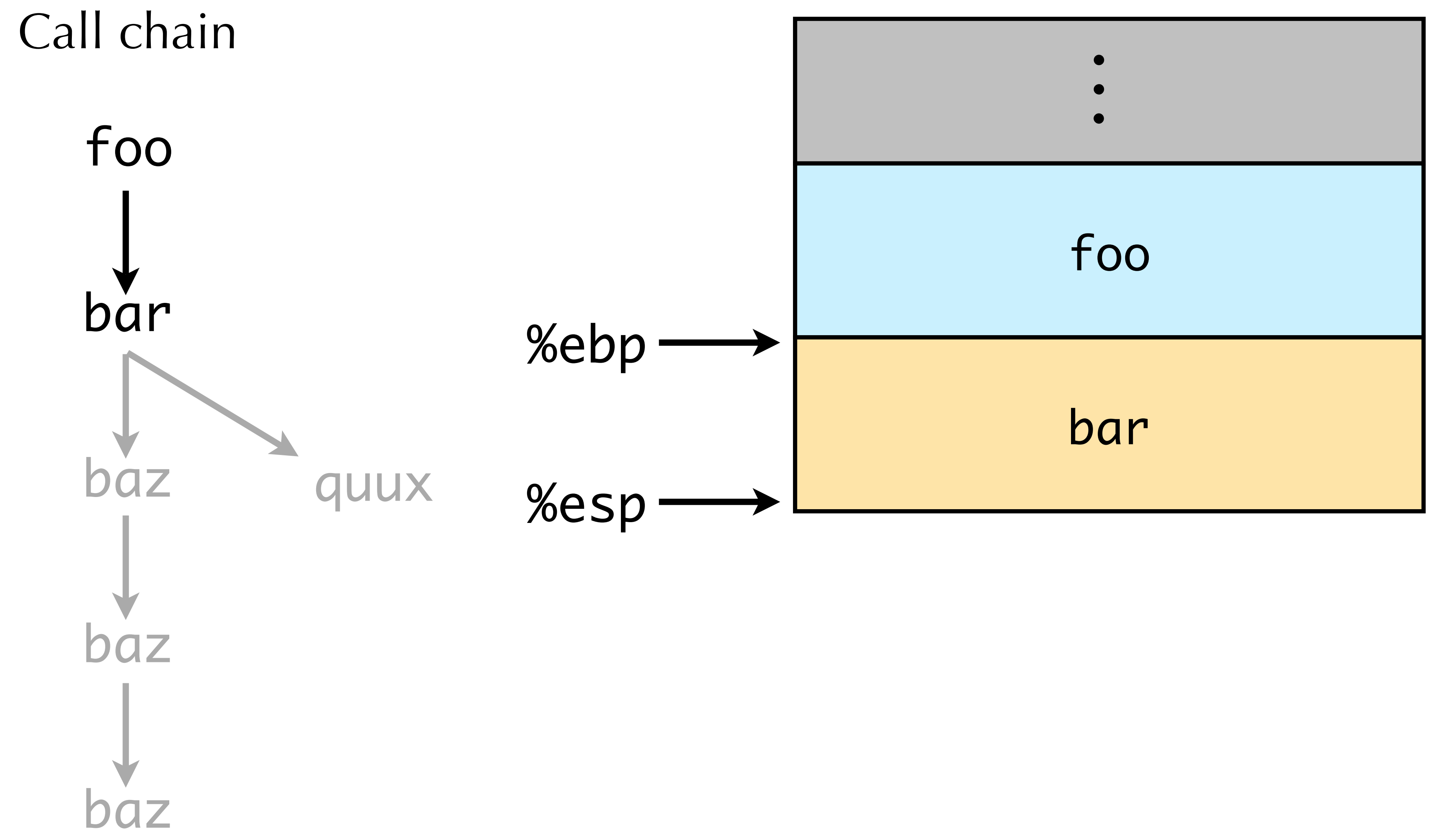
Stack frame example



Stack frame example

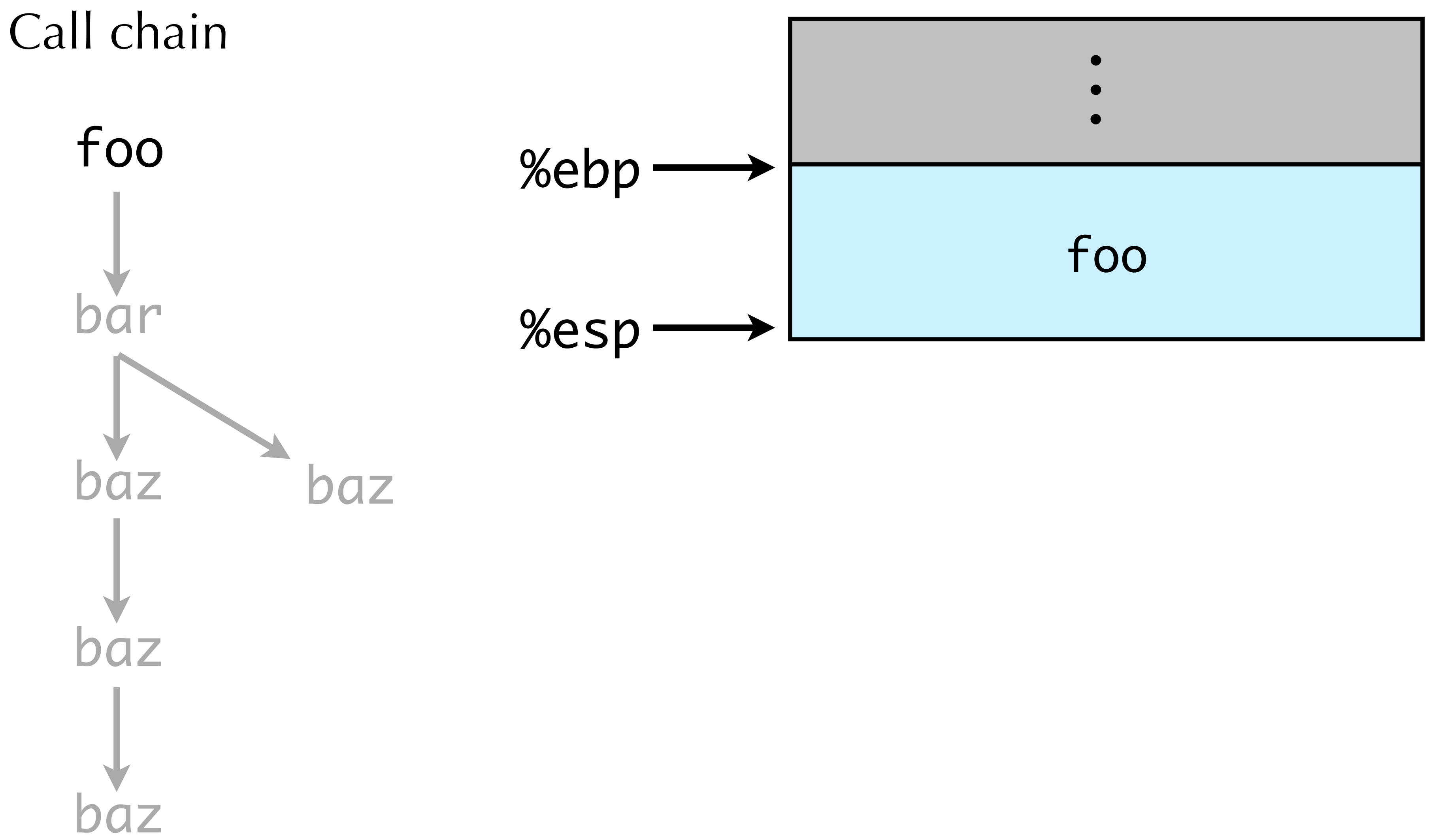
```
void foo(...) {
}
```

```
void bar(...) {
   int x, y;
    x = baz();
    ...
    y = quux();
    ...
}
```



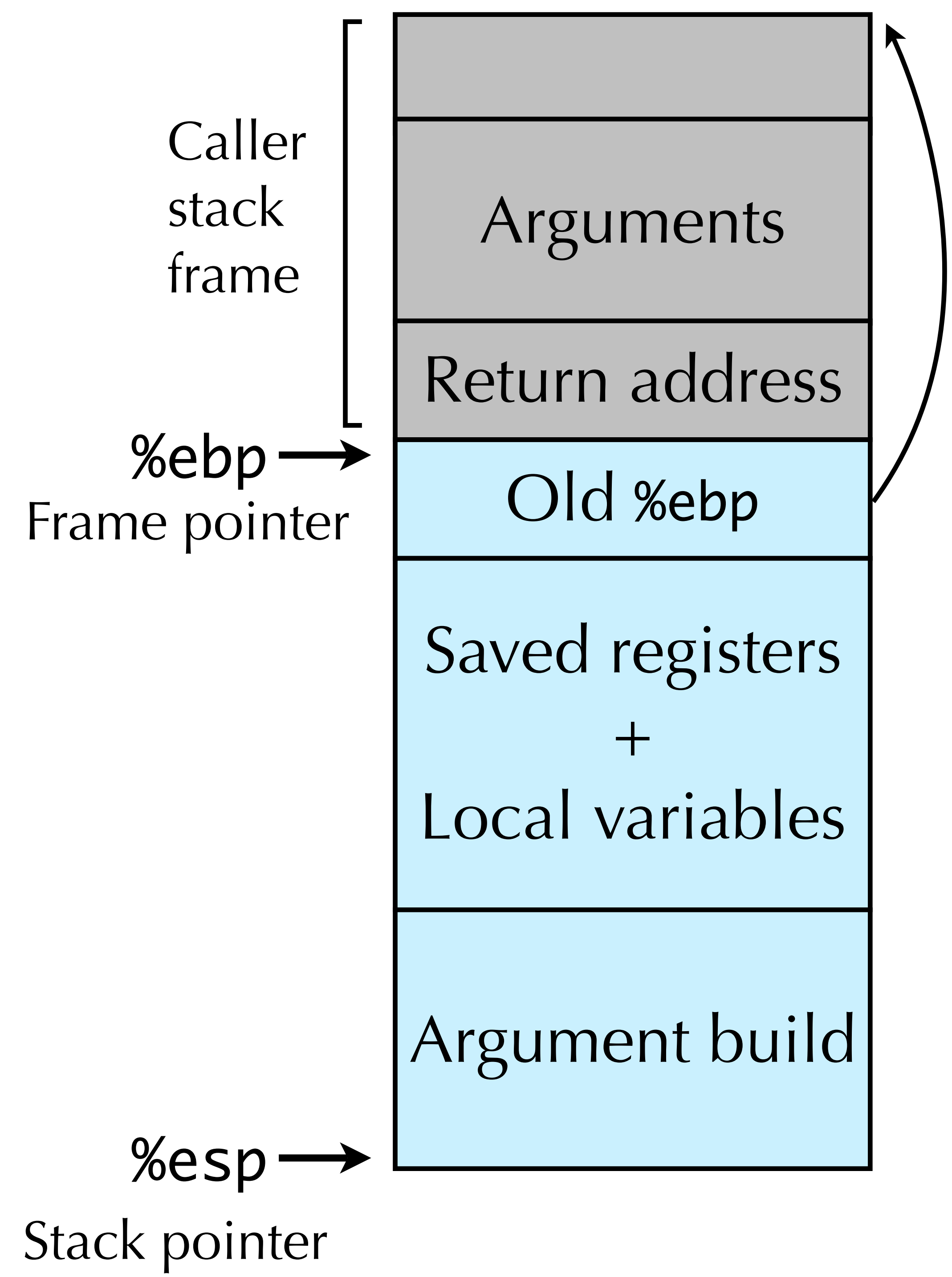
Stack frame example

```
void foo(...) {
    ...
    bar();
    ...
}
```



x86/Linux stack frame

- The exact layout of a stack frame is a convention.
  - Depends on hardware, OS, and compiler used.
- x86/Linux stack frame contains:
  - Old value of `%ebp` (from previous frame)
  - Any saved registers (more later)
  - Local variables (if not kept in registers)
  - Arguments to function about to be called
- The **caller's** stack frame contains:
  - Return address – pushed by call instruction
  - Arguments for this function call



Swap revisited

```
/* Global vars */
int zip1 = 15213;
int zip2 = 91125;

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

```
call_swap:
   ...
   pushl $zip2    # Push args
   pushl $zip1    #   on stack
   call swap      # Do the call
   ...
```

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

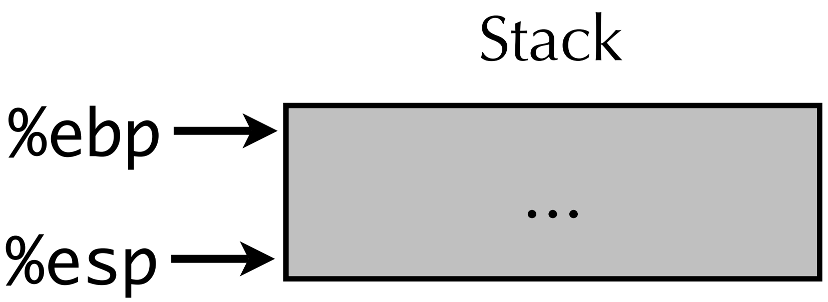
Swap revisited

```
/* Global vars */
int zip1 = 15213;
int zip2 = 91125;

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

```
call_swap:
   ...
   pushl $zip2    # Push args
   pushl $zip1    #   on stack
   call swap      # Do the call
   ...
```

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



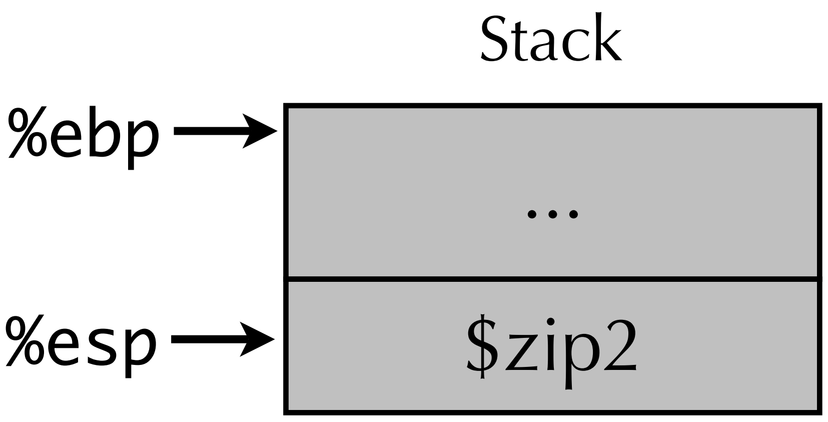
Swap revisited

```
/* Global vars */
int zip1 = 15213;
int zip2 = 91125;

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

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

```
call_swap:
    ...
    pushl $zip2     # Push args
    pushl $zip1     #    on stack
    call swap       # Do the call
    ...
```



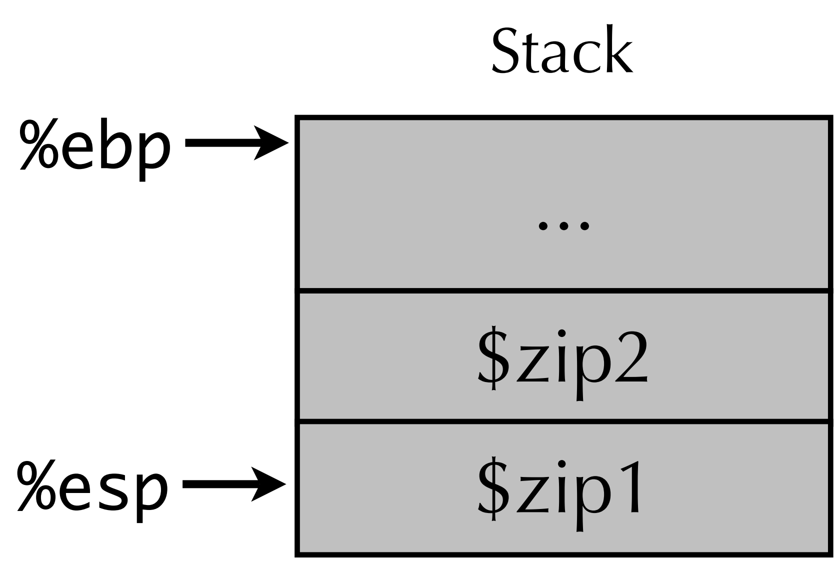
Swap revisited

```
/* Global vars */
int zip1 = 15213;
int zip2 = 91125;

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

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

```
call_swap:
    ...
    pushl $zip2    # Push args
    pushl $zip1    #    on stack
    call swap      # Do the call
    ...
```



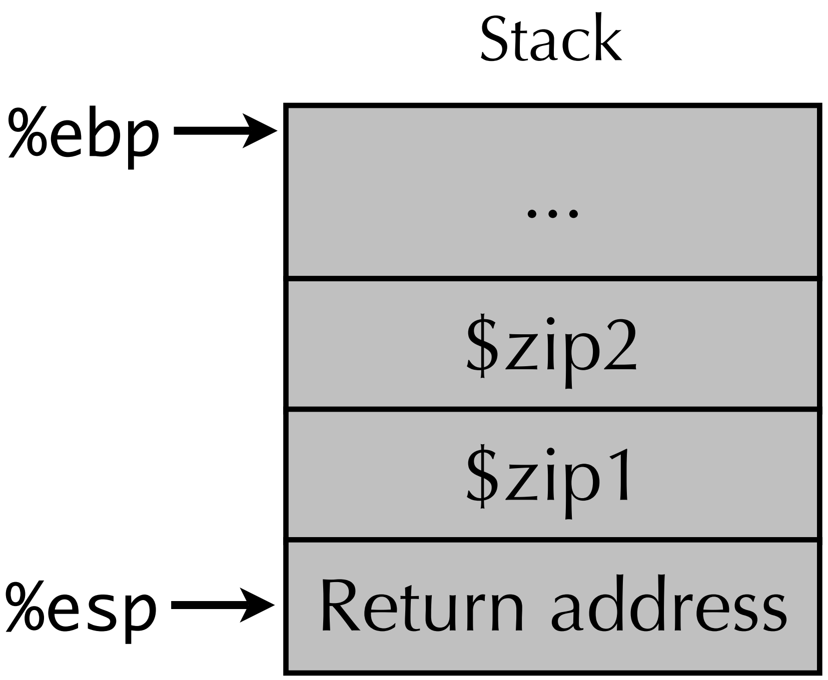
Swap revisited

```
/* Global vars */
int zip1 = 15213;
int zip2 = 91125;

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

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

```
call_swap:
    ...
    pushl $zip2    # Push args
    pushl $zip1    #   on stack
    call swap      # Do the call
    ...
```



Code for swap

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

```
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
```

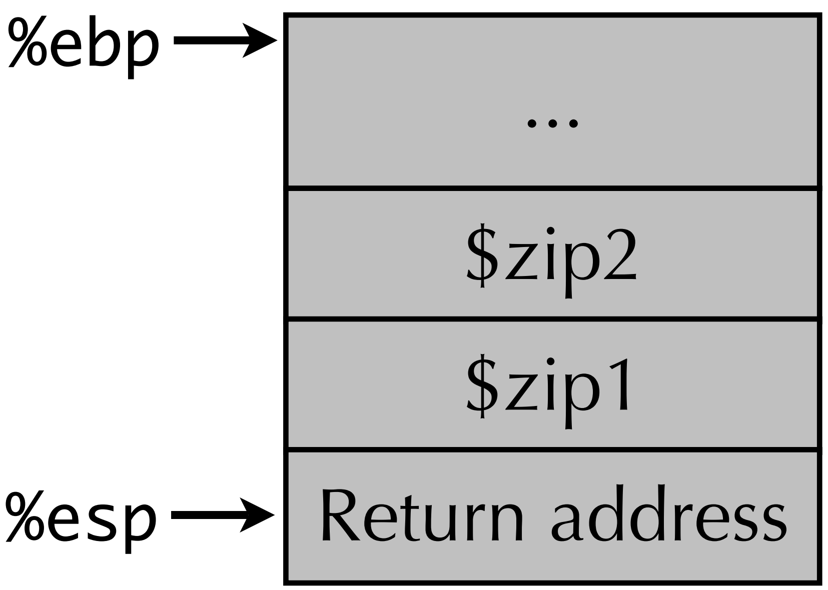
Set up

Body

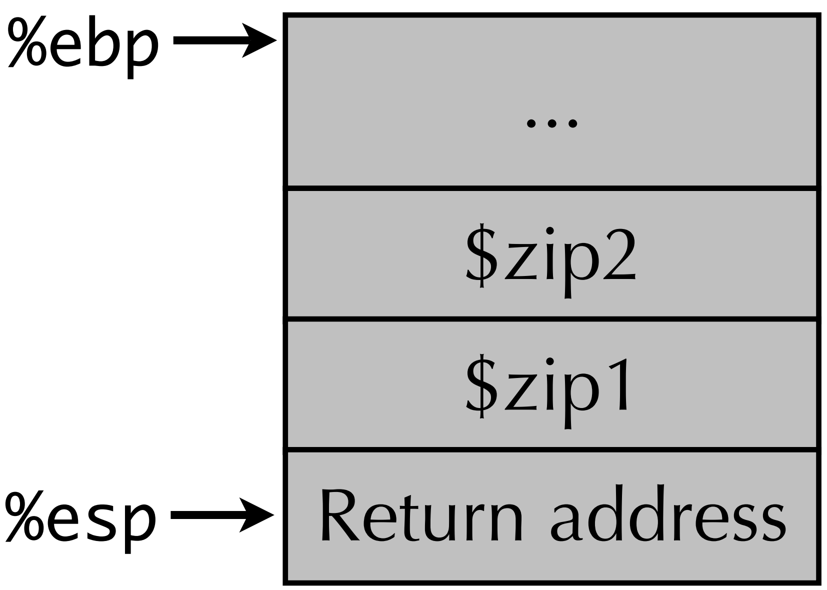
Finish

Swap setup

Stack entering swap



Resulting stack

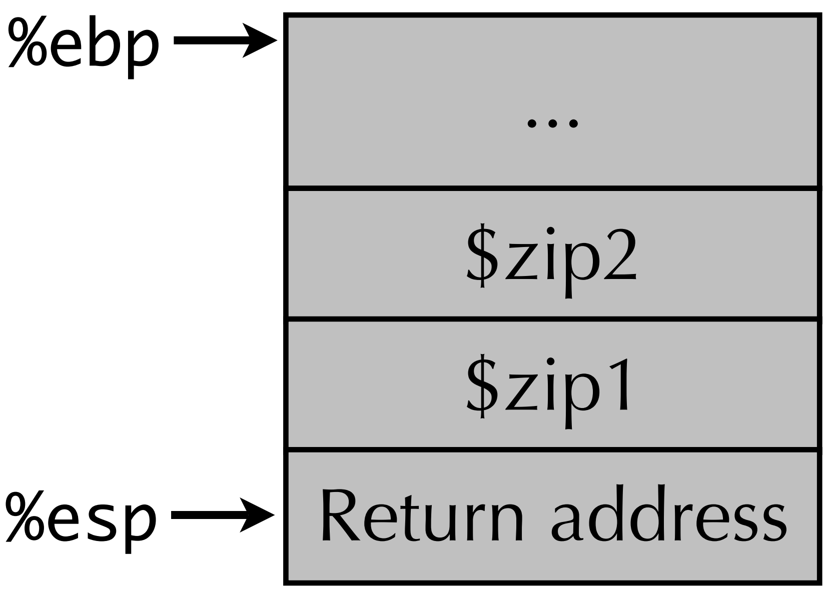



```
pushl %ebp
movl %esp,%ebp
pushl %ebx
```

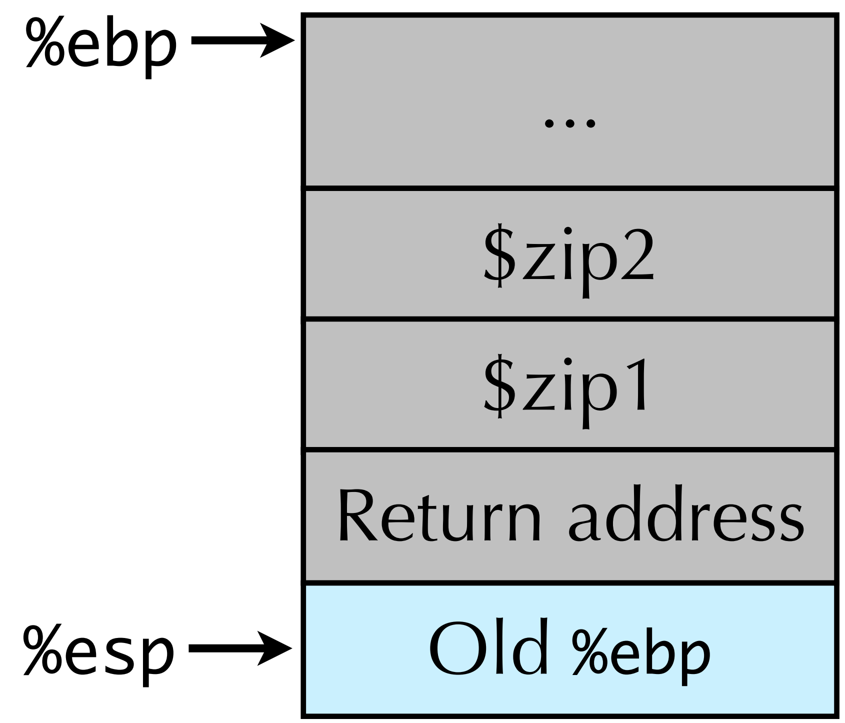
Set up

Swap setup

Stack entering swap



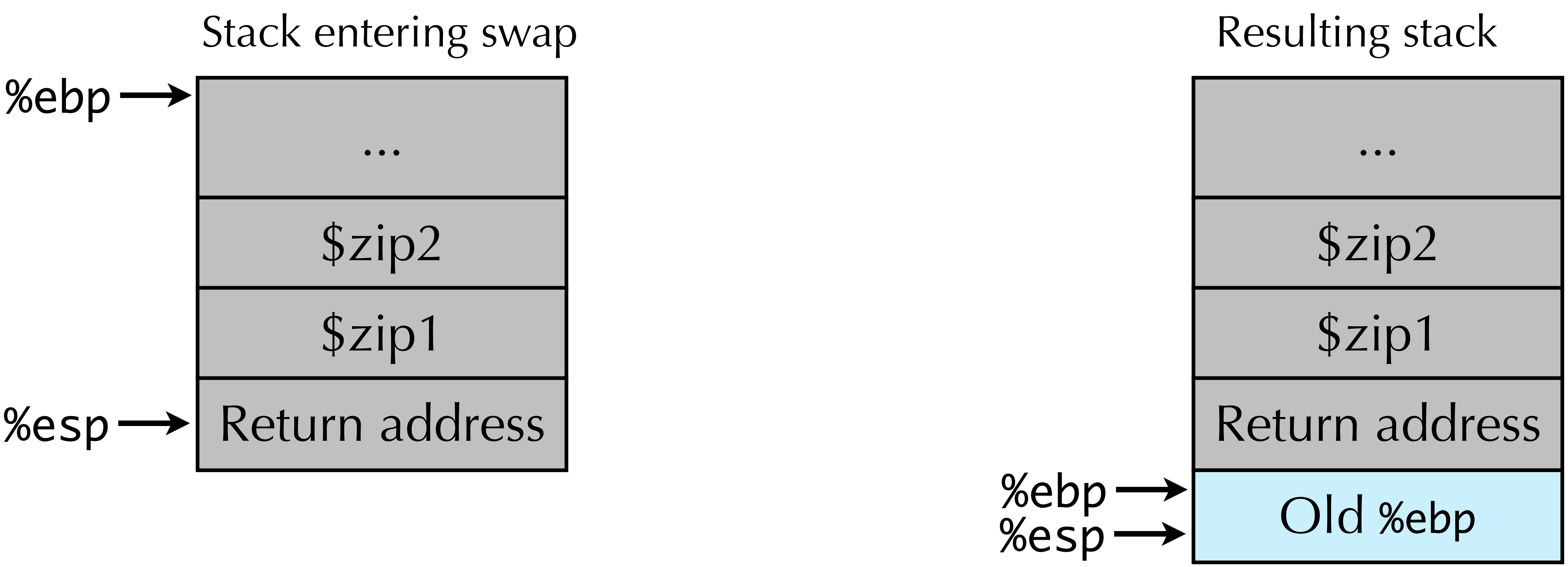
Resulting stack




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

Set up

Swap setup

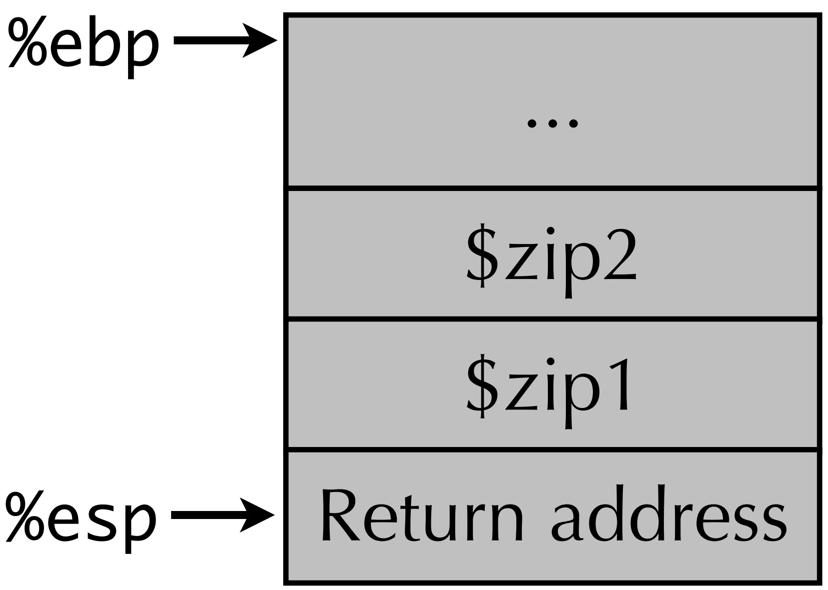



```
pushl %ebp
movl %esp,%ebp
pushl %ebx
```

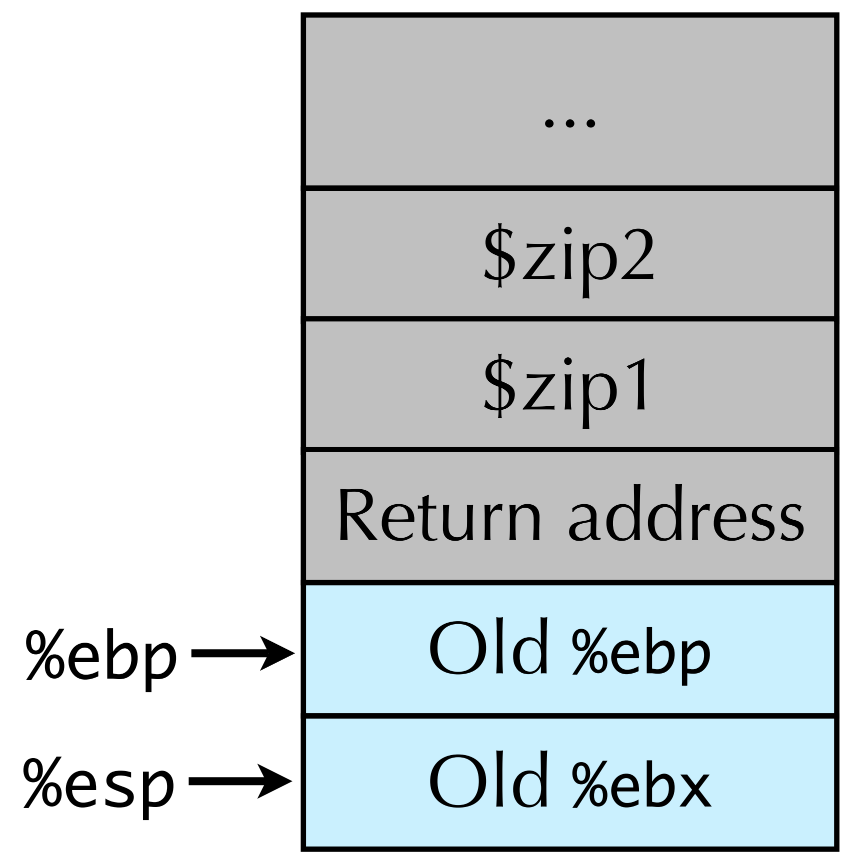
Set up

Swap setup

Stack entering swap



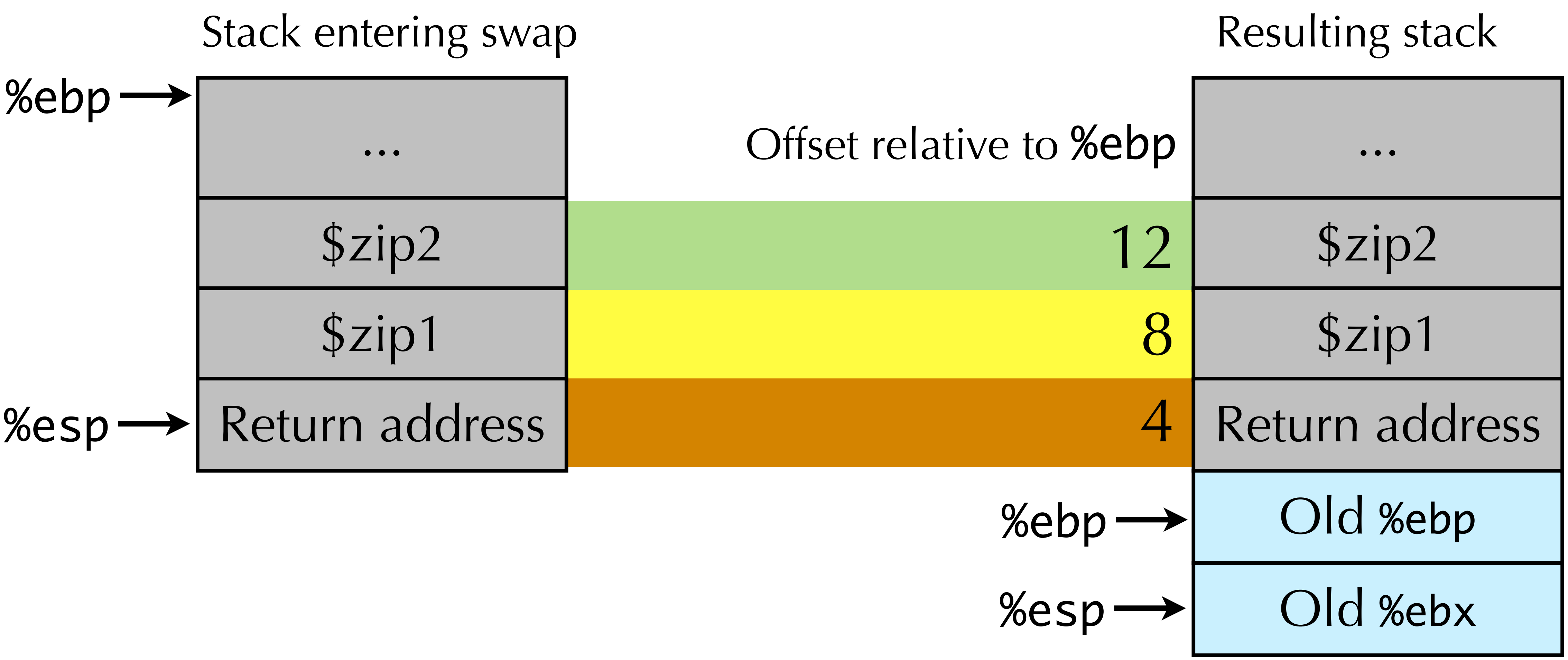
Resulting stack




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

Set up

Swap body

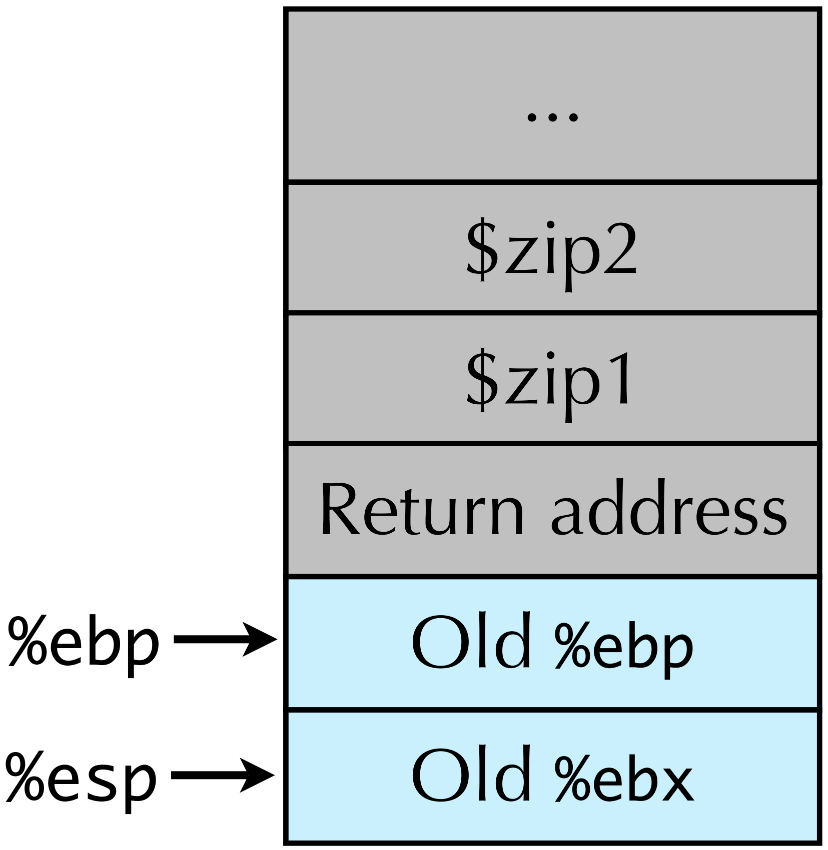



```
movl 12(%ebp),%ecx
movl 8(%ebp),%edx
movl (%ecx),%eax
movl (%edx),%ebx
movl %eax,(%edx)
movl %ebx,(%ecx)
```

Body

Swap finish

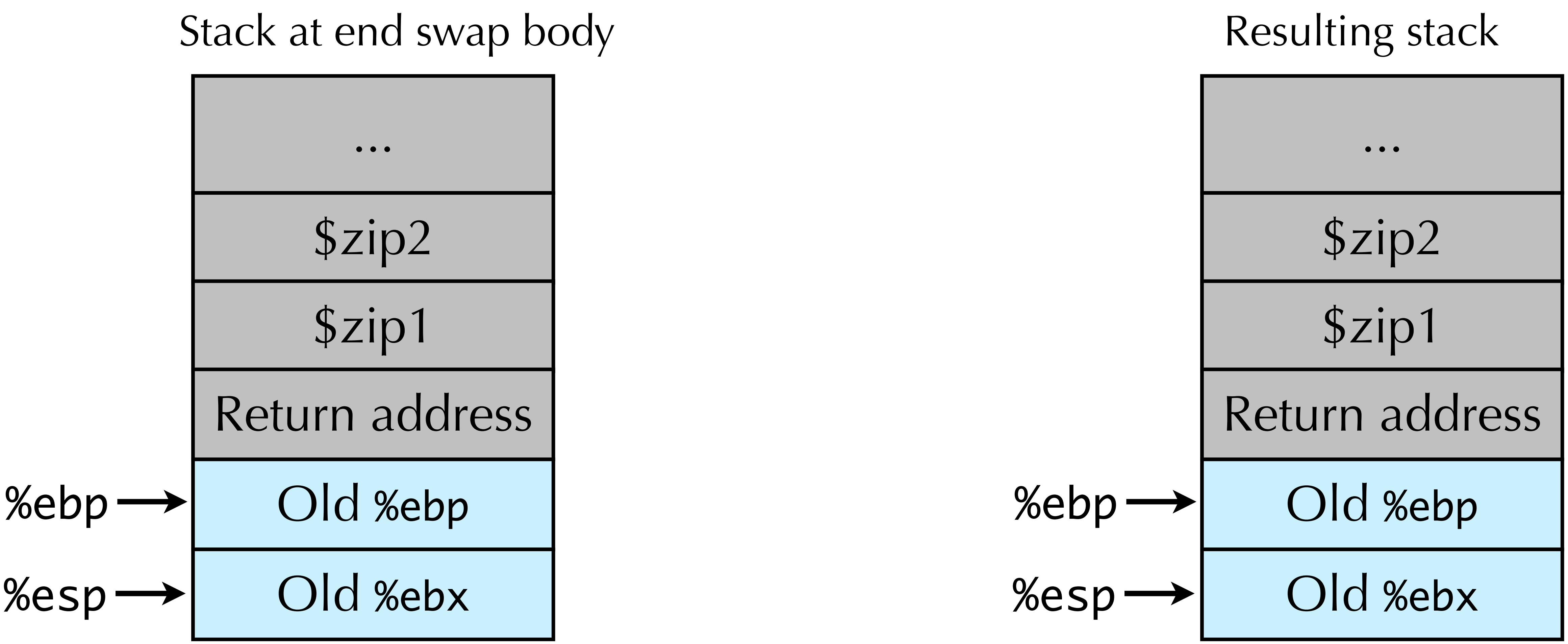
Stack at end swap body




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

Finish

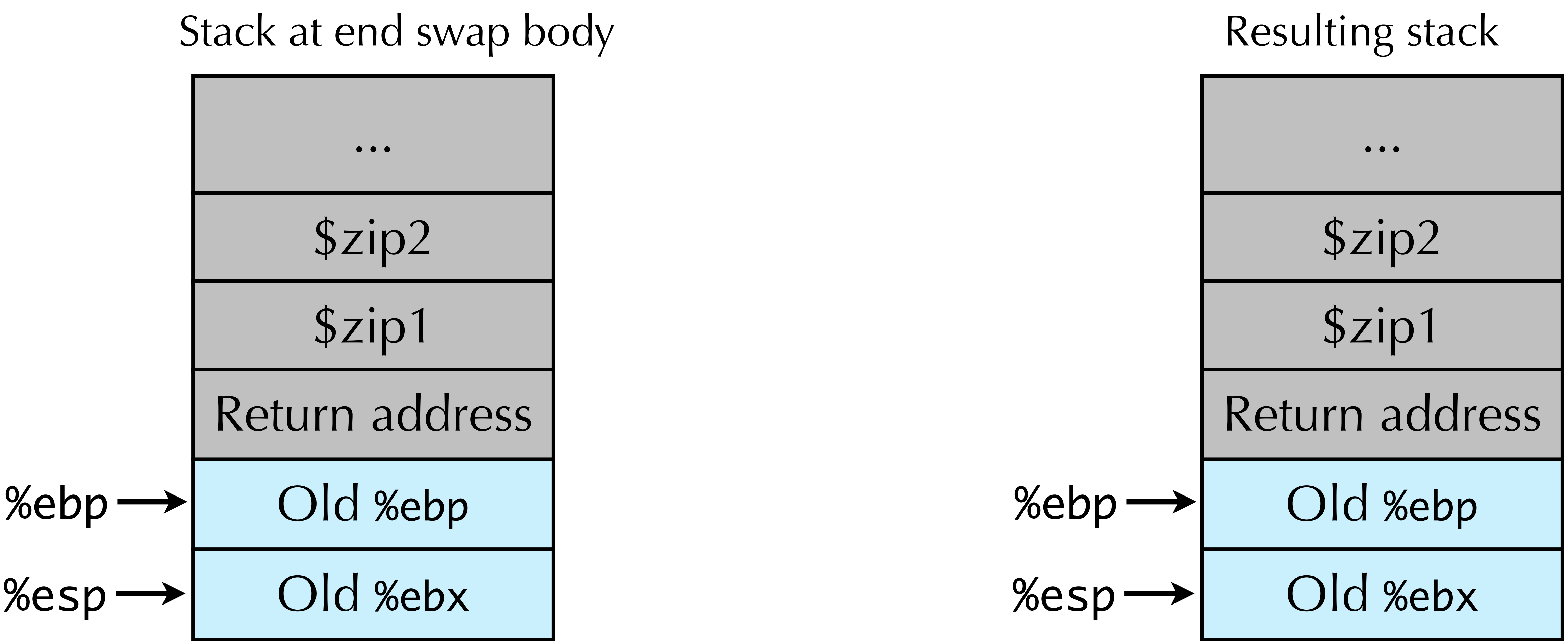
Swap finish




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

Finish

Swap finish



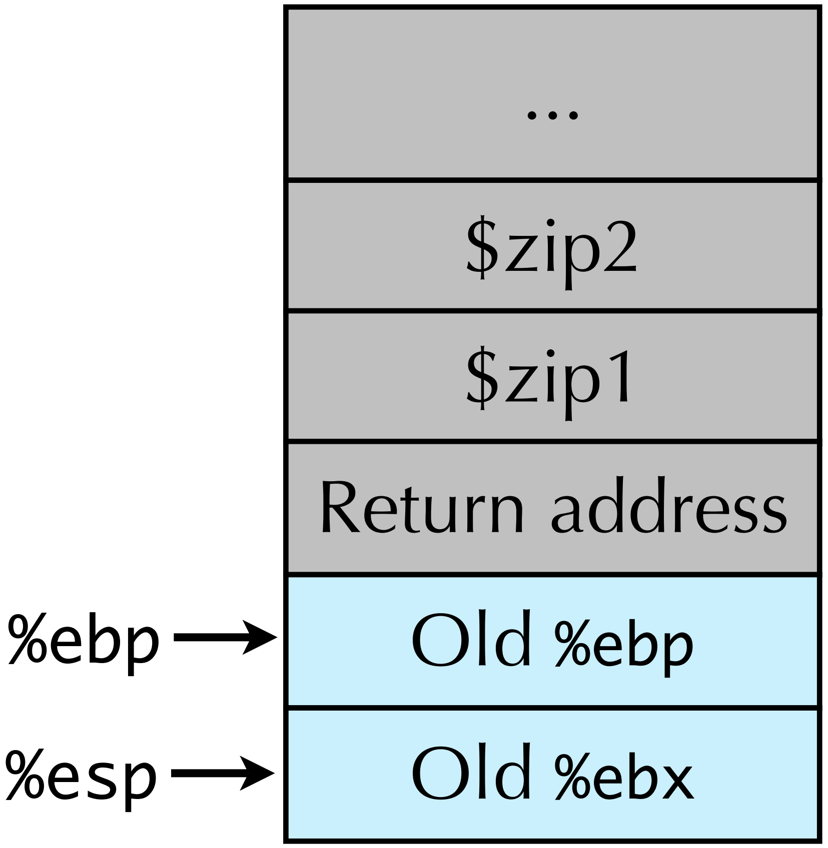

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Finis

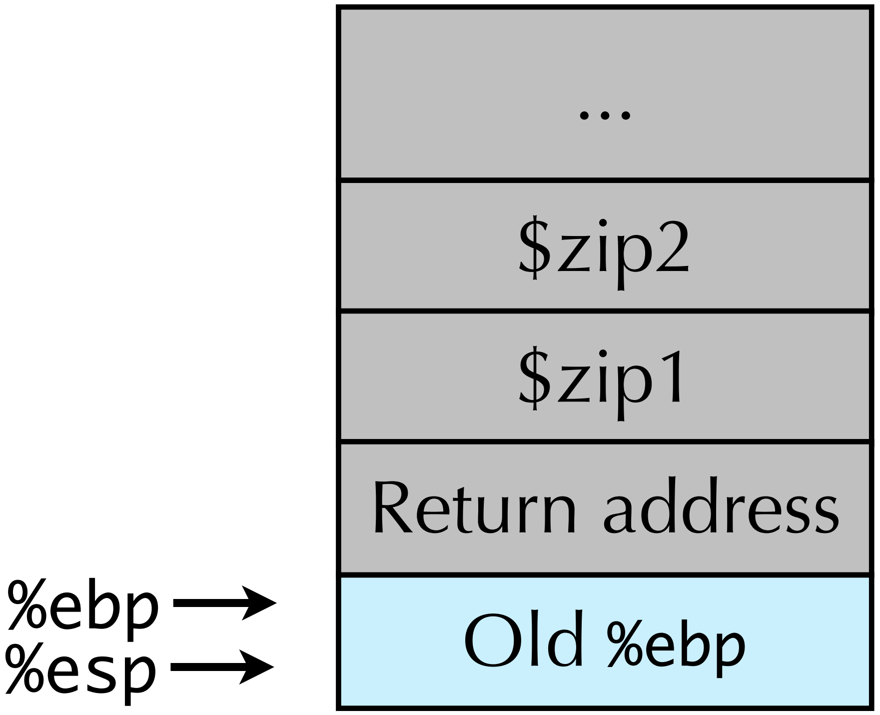
Restores old value of %ebx!

Swap finish

Stack at end swap body



Resulting stack

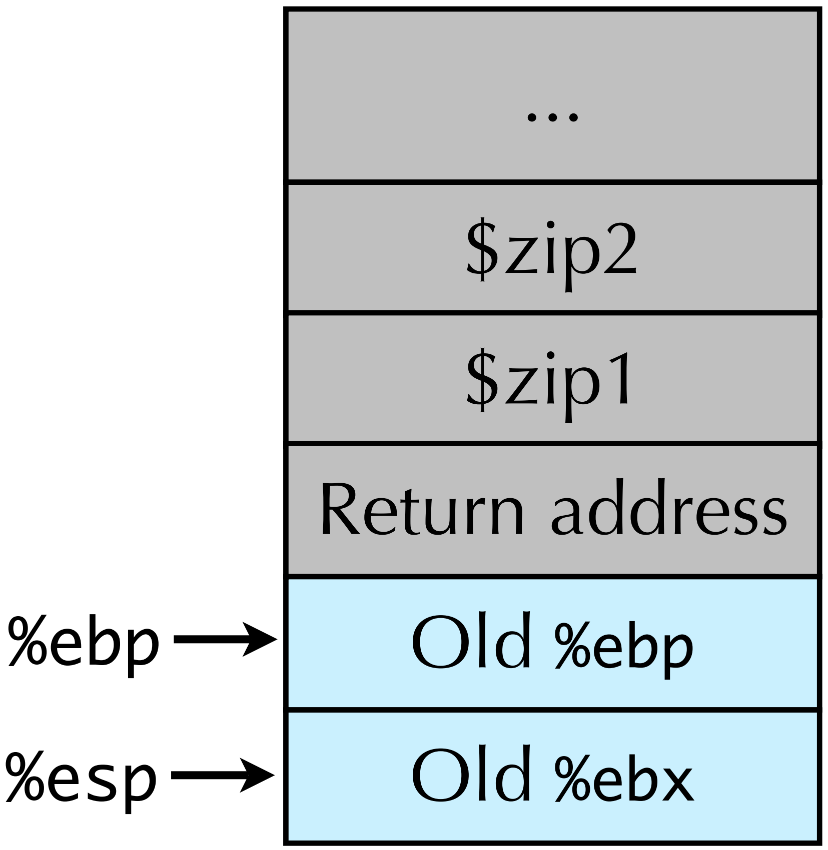



```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

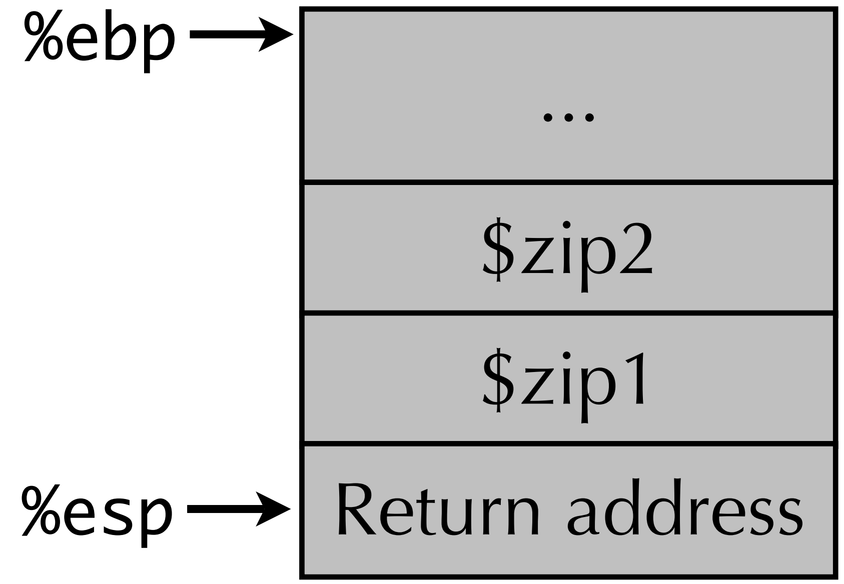
Finish

Swap finish

Stack at end swap body



Resulting stack

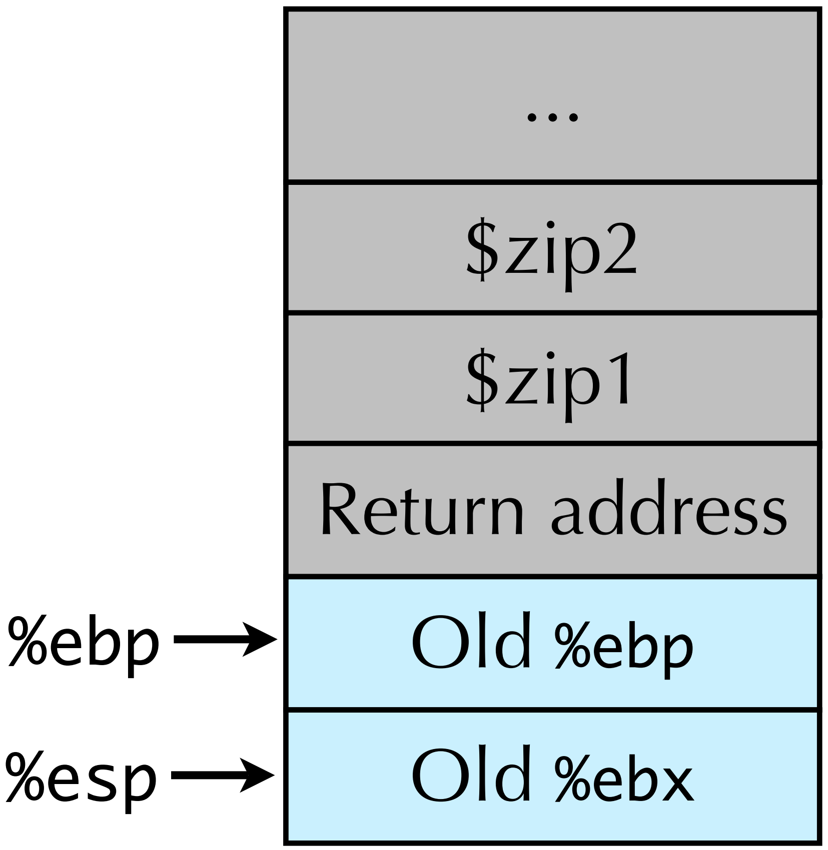



```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

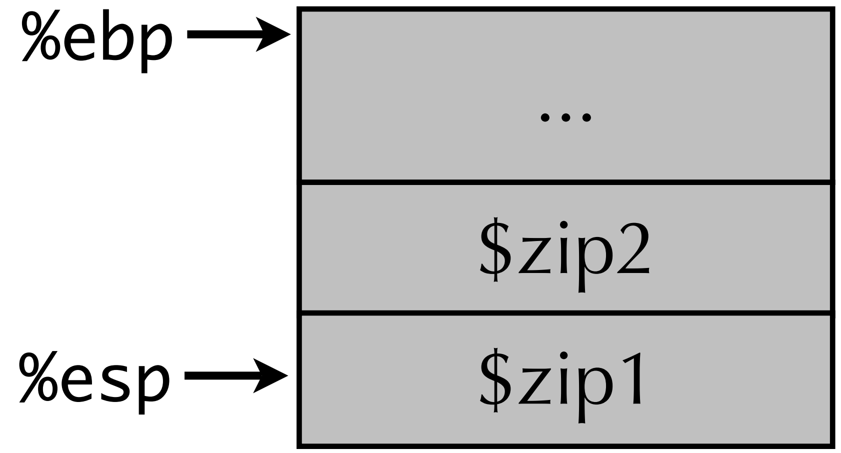
Finish

Swap finish

Stack at end swap body



Resulting stack




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

Finish

`leave` instruction

- Actual disassembly of swap

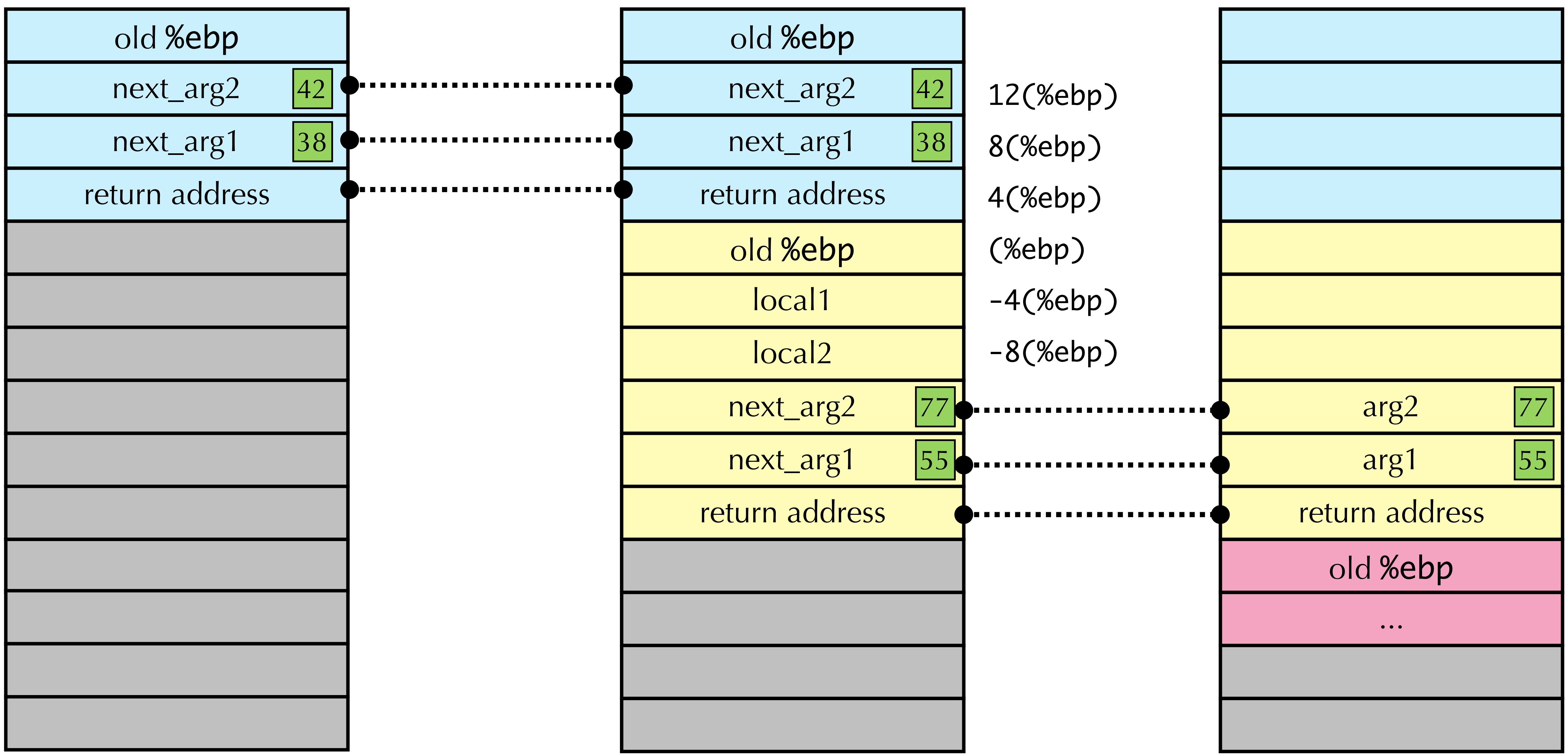
```
080483a4 <swap>:
 80483a4:   55          push   %ebp
 80483a5:   89 e5       mov    %esp,%ebp
 80483a7:   53          push   %ebx
 80483a8:   8b 55 08    mov    0x8(%ebp),%edx
 80483ab:   8b 4d 0c    mov    0xc(%ebp),%ecx
 80483ae:   8b 1a       mov    (%edx),%ebx
 80483b0:   8b 01       mov    (%ecx),%eax
 80483b2:   89 02       mov    %eax,(%edx)
 80483b4:   89 19       mov    %ebx,(%ecx)
 80483b6:   5b          pop    %ebx
 80483b7:   c9          leave
 80483b8:   c3          ret
```

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

- `leave` prepares the stack for returning
- `leave` is equivalent to

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

Stack frame cheat sheet




```
foo() {
  bar(38,42);
}
```

```
bar(int arg1, int arg2) {
  int local1, local2;
  ...
  baz(55,77);
}
```

```
baz(int arg1, int arg2) {
  ...
}
```

Return values

- By convention, the compiler leaves return value in `%eax`

```
int absdiff(int x, int y)
{
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

```
absdiff:
    pushl  %ebp
    movl   %esp, %ebp
    movl   8(%ebp), %edx
    movl   12(%ebp), %eax
    cmpl   %eax, %edx
    jle    .L7
    subl   %eax, %edx
    movl   %edx, %eax
.L8:
    leave
    ret
.L7:
    subl   %edx, %eax
    jmp    .L8
```

Return values

- By convention, the compiler leaves return value in %eax

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

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

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

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

- Works fine for 32-bit values
- For floating point values: other registers used
- For structs: return value is left on stack, caller must copy data elsewhere
    - Why must caller copy the data?

Register saving conventions

- When procedure `foo()` calls `bar()`

  `foo()` is the **caller**, `bar()` is the **callee**
- Suppose `bar()` needs to modify some registers when it run
  - But `foo()` is using some of the same registers for its own purposes

```
foo:
    ...
    movl $15213, %edx
    call bar
    addl %edx, %eax
    ...
    ret
```

```
bar:
    ...
    movl 8(%ebp), %edx
    addl $91125, %edx
    ...
    ret
```

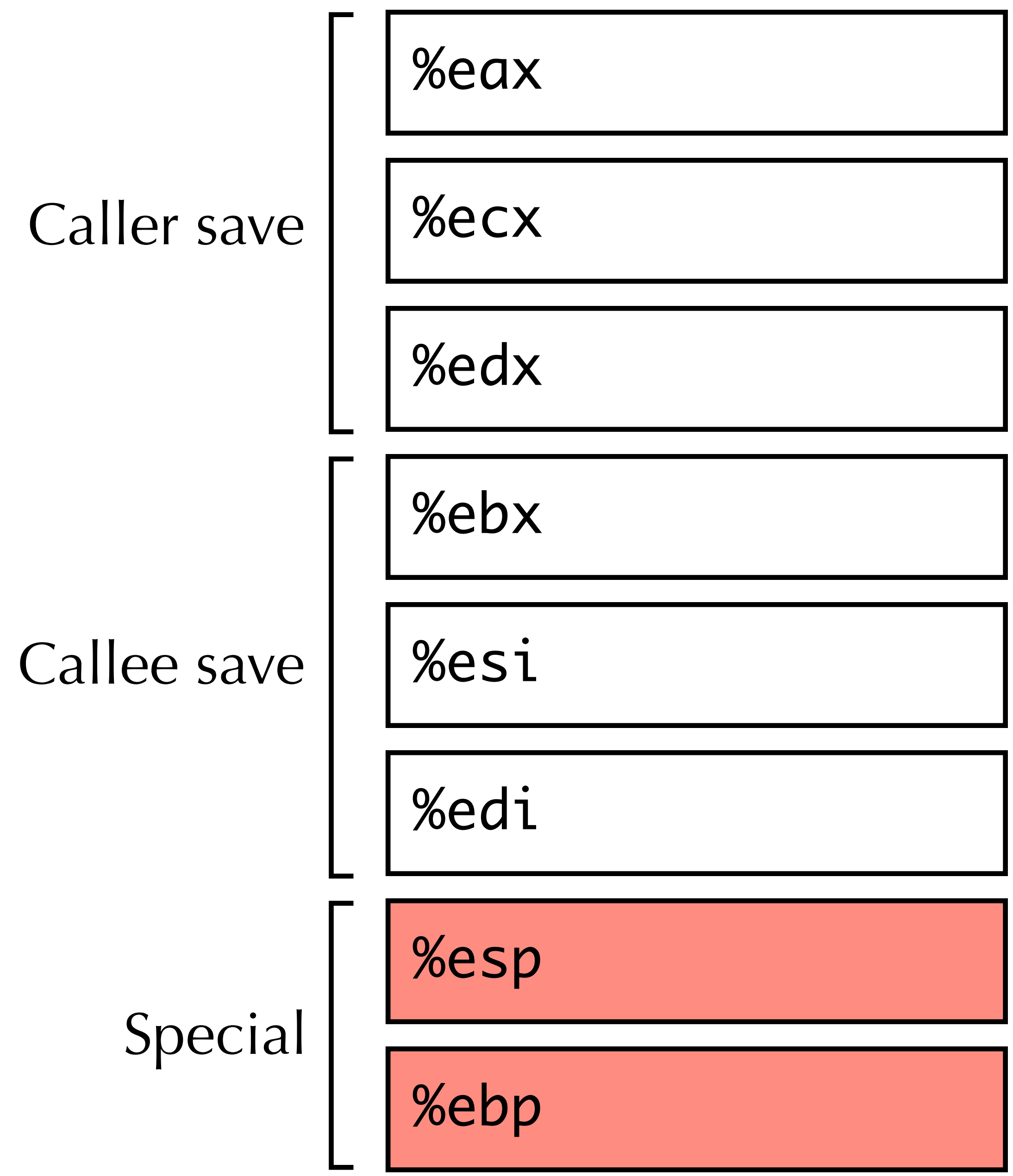
- Contents of `%edx` clobbered by `bar()`!

Register saving conventions

- Need to save some of the clobbered registers on the stack.
- Who saves the registers? The caller? The callee?
  - **Caller save:** caller saves registers in its stack frame before call
  - **Callee save:** callee saves registers it will clobber in its stack frame, and restores them before return
- What are advantages and disadvantages of each?
  - Caller save: caller must be conservative and save everything, since it doesn't know what callee will clobber.
  - Callee save: callee must be conservative and save everything, since it doesn't know what caller wants preserved.

x86/Linux register conventions

- x86/Linux uses a mixture of caller-save and callee-save!
- Three registers managed as caller-save
    - `%eax`, `%ecx`, `%edx`
- Three registers managed as callee-save
    - `%ebx`, `%esi`, `%edi`
- Frame and stack registers managed specially
    - `%esp`, `%ebp`



Procedures summary

- The stack makes function calls work!
  - Private storage for each invocation of a procedure call
  - Multiple function invocations don't clobber each other
  - Addressing of local variables and arguments is relative to stack frame `%ebp`
  - Recursion works too
  - Requires that procedures return in order of invocations (nesting is preserved)
- Procedures implemented using a combination of **hardware support** plus **software conventions**
  - Hardware support: `call`, `ret`, `leave`, `pushl`, `popl`
  - Software conventions: Register saving conventions, managing `%esp` and `%ebp`, managing layout of stack
    - Software conventions defined by the OS and the compiler.
    - No guarantee it will be the same on a different software platform.

Next lecture

- Structured data
  - Arrays
  - Arrays of arrays
  - Structs
  - Arrays of structs…
- (Please leave name tags!)