



HARVARD

John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers

Lecture 3: Assembly ctd.

Stephen Chong

<https://www.seas.harvard.edu/courses/cs153>

Contains content from lecture notes by Steve Zdancewic

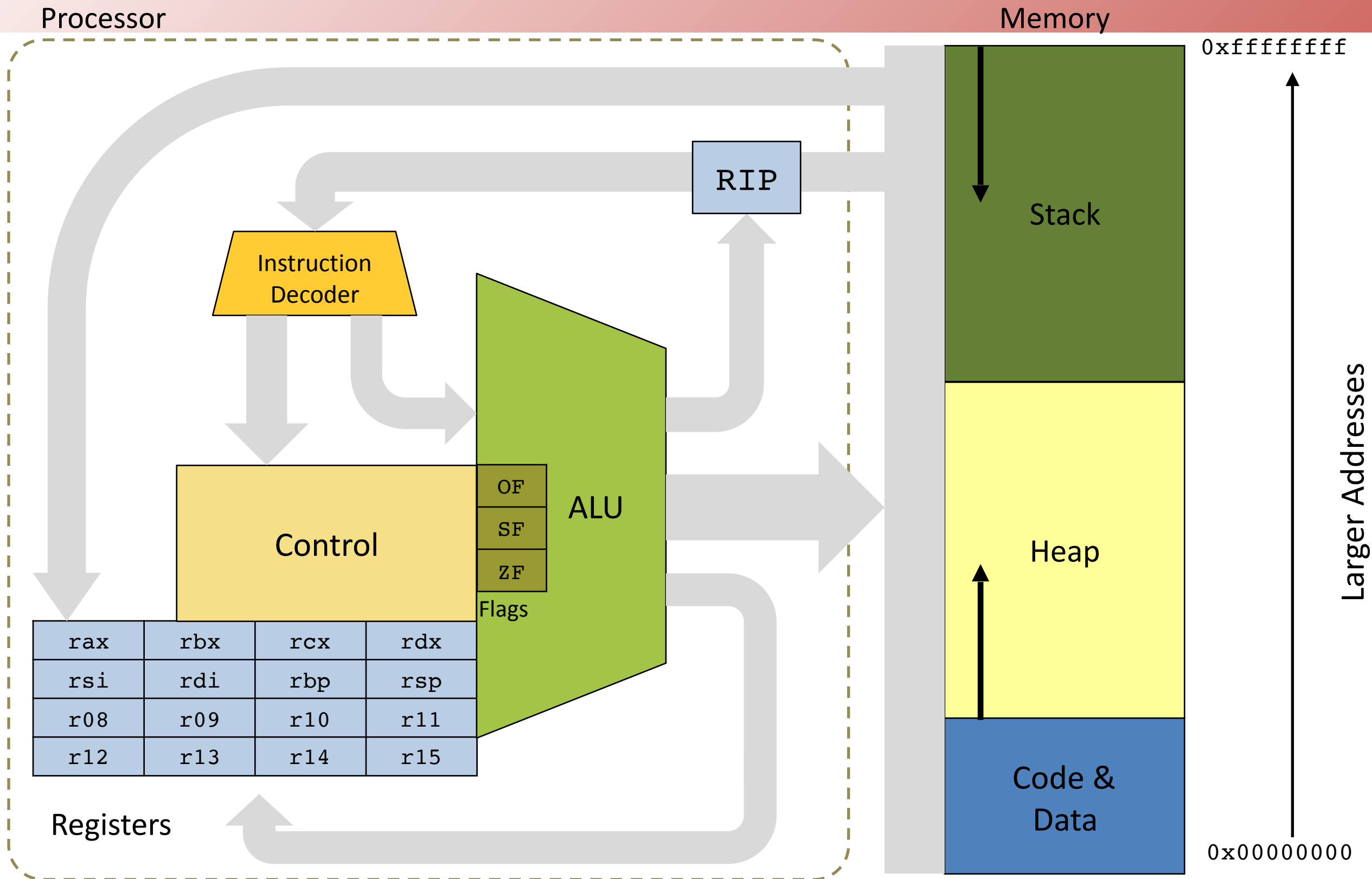
Announcements

- Office hours started
 - See website for details
- Homework 1 (HelloCaml) due today
 - Recall: at most 3 days worth of late minutes can be used per homework
 - You have 10 days worth of late minutes in total
- Homework 2 X86lite out today
 - Due Tuesday Sept 24

Today

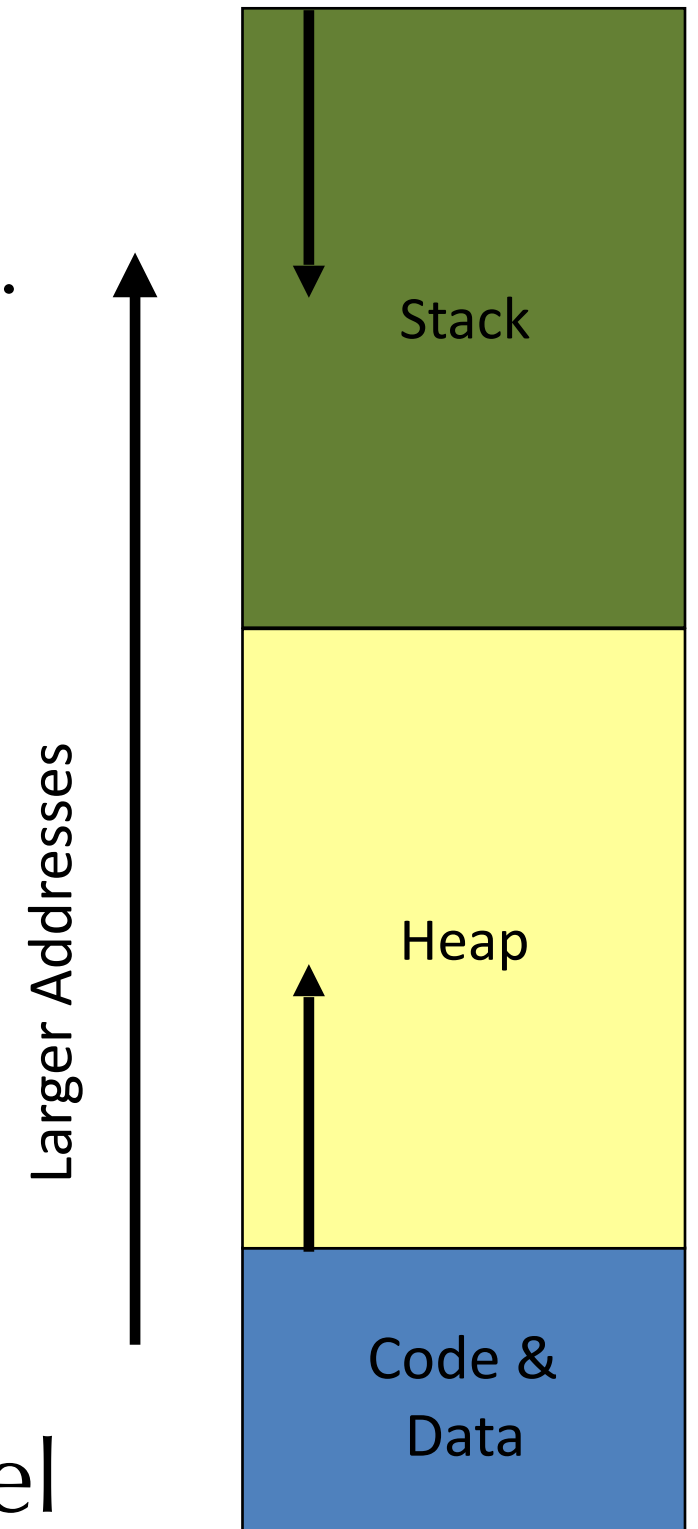
- Continue looking at representation of x86 code
 - From previous lecture
- C memory layout
- Calling convention

X86 Schematic



3 parts of the C memory model

- The code & data (or “text”) segment
 - contains compiled code, constant strings, etc.
- The Heap
 - Stores dynamically allocated objects
 - Allocated via “`malloc`”
 - Deallocated via “`free`”
 - C runtime system
- The Stack
 - Stores local variables
 - Stores the return address of a function
- In practice, most languages use this model



Local/Temporary Variable Storage

- Need space to store:
 - Global variables
 - Values passed as arguments to procedures
 - Local variables (either defined in the source program or introduced by the compiler)
- Processors provide two options
 - Registers: fast, small size (32 or 64 bits), very limited number
 - Memory: slow, very large amount of space (2+ GB)
 - caching important
- In practice on X86:
 - Registers are limited (and have restrictions)
 - Divide memory into regions including the stack and the heap

Calling Conventions

- Specify the locations (e.g. register or stack) of arguments passed to a function and returned by the function
- Designate registers either:
 - Caller Save – e.g. freely usable by the called code
 - Callee Save – e.g. must be restored by the called code
- Define the protocol for deallocating stack-allocated arguments
 - Caller cleans up
 - Callee cleans up (makes variable arguments harder)

32-bit cdecl calling conventions

- “Standard” on X86 for many C-based operating systems (i.e. almost all)
 - Still some wrinkles about return values (e.g. some compilers use `EAX` and `EDX` to return small values)
 - 64-bit allows for packing multiple values in one register
- Arguments are passed on the stack in right-to-left order
- Return value is passed in `EAX`
- Registers `EAX`, `ECX`, `EDX` are caller save
- Other registers are callee save
 - Ignoring these conventions will cause havoc (bus errors or seg faults)

Stack frame example

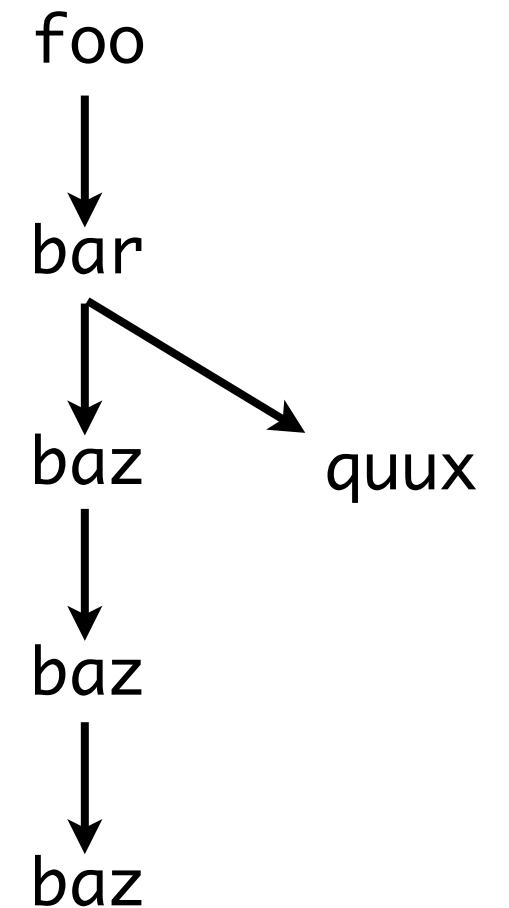
```
void foo(...) {  
    ...  
    bar();  
    ...  
}
```

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

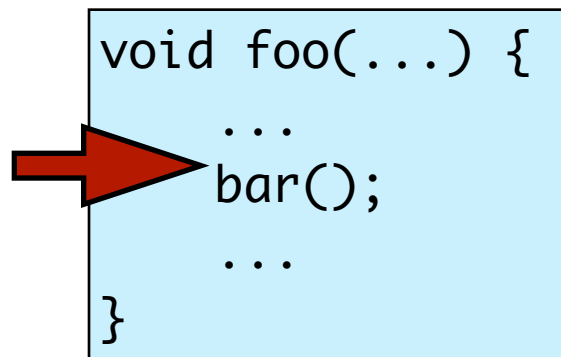
```
int baz(...) {  
    int z;  
    ...  
    z = baz();  
    ...  
    return z;  
}
```

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

Call chain

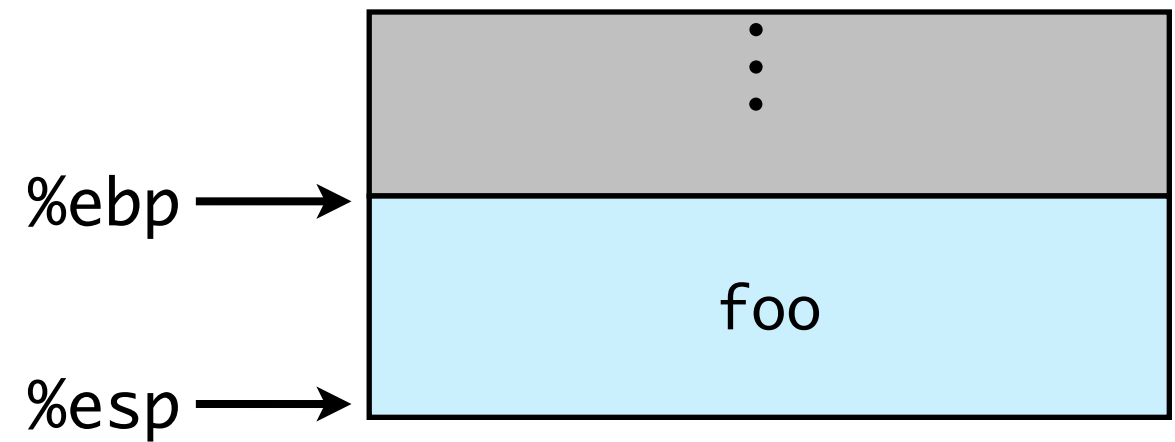


Stack frame example



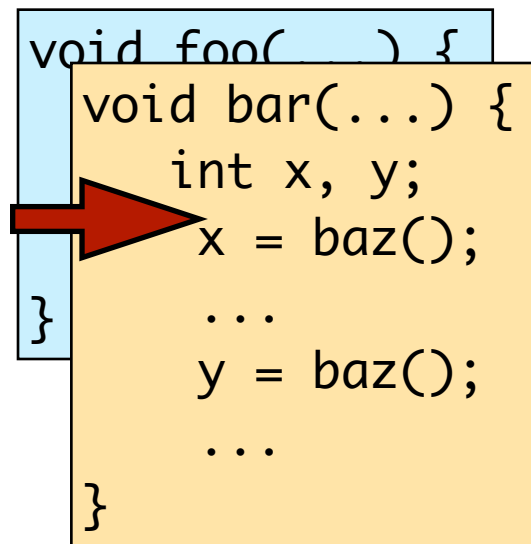
Call chain

foo

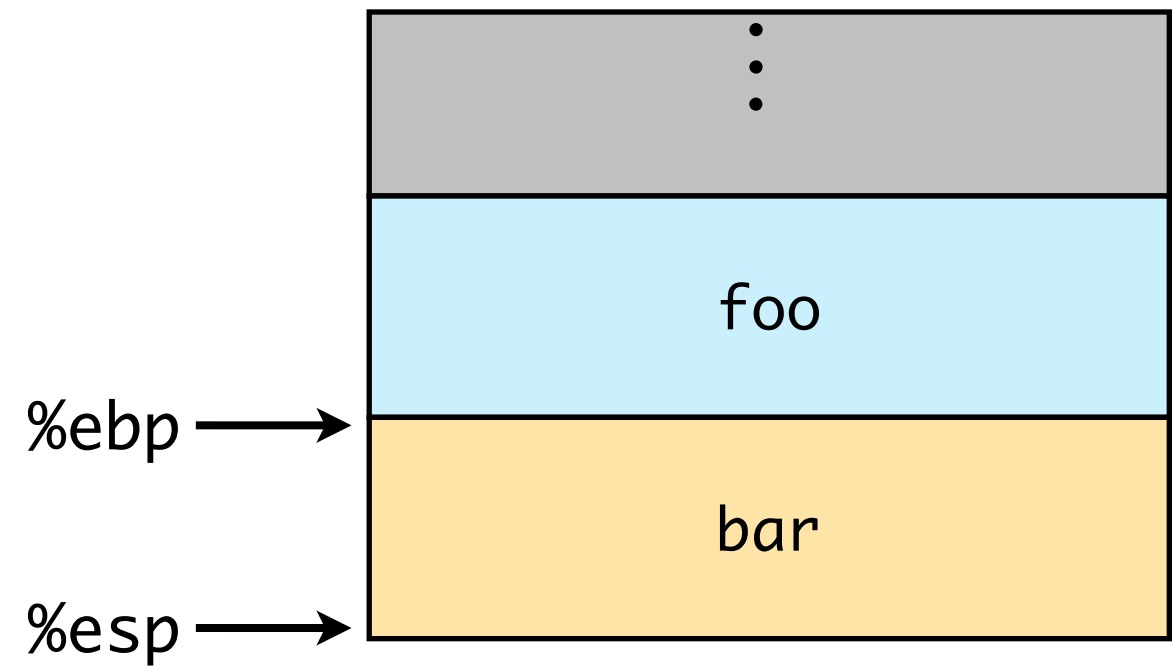


Stack frame example

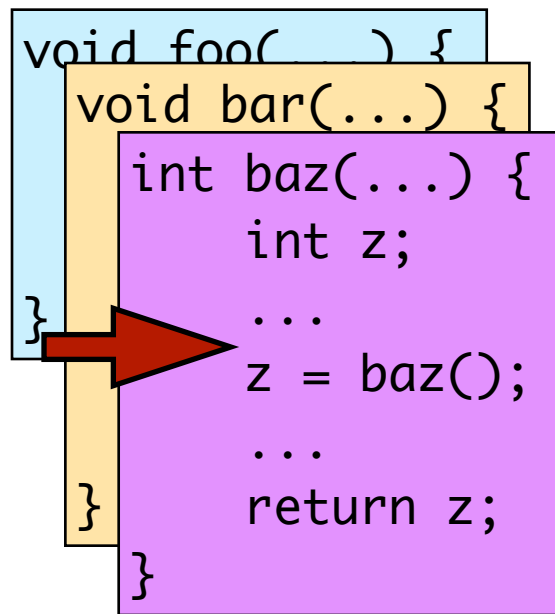
Call chain



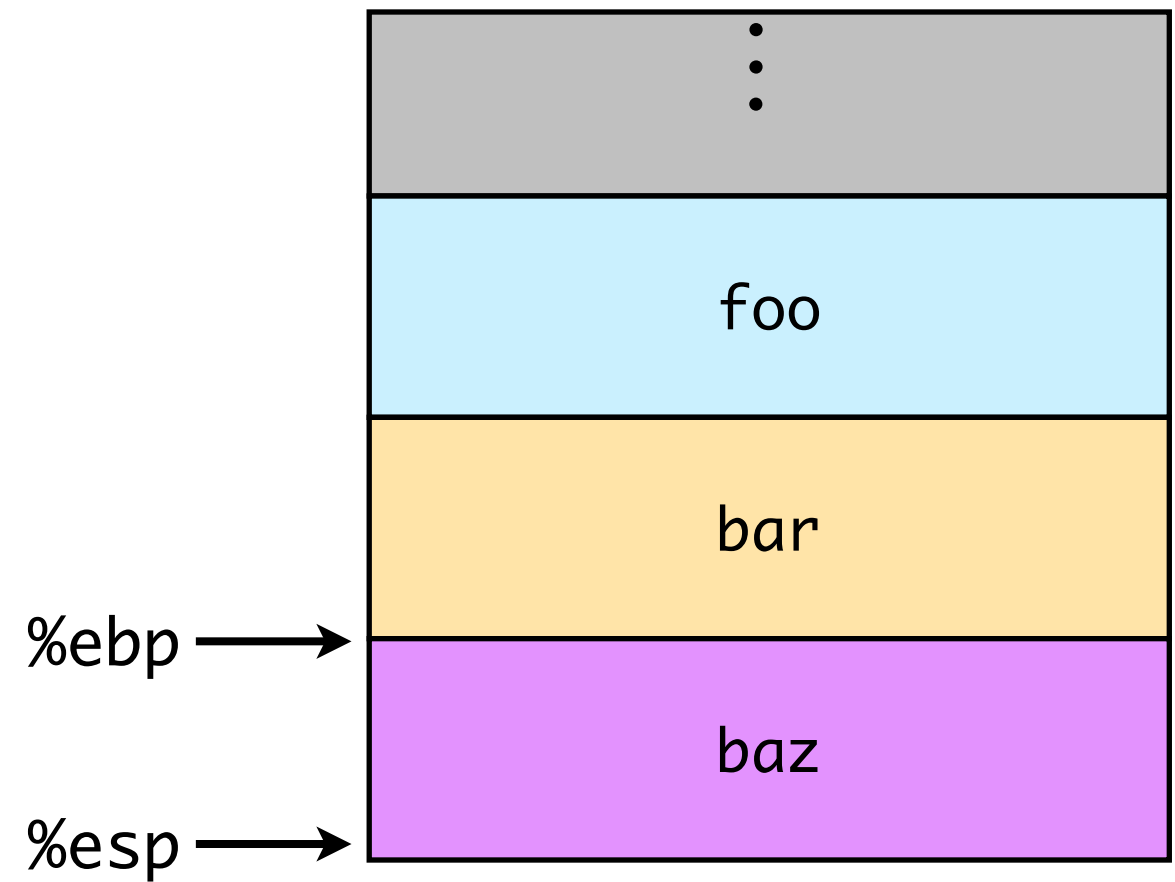
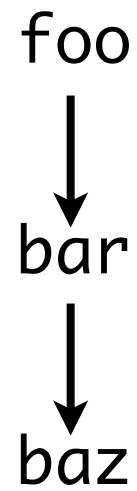
foo
↓
bar



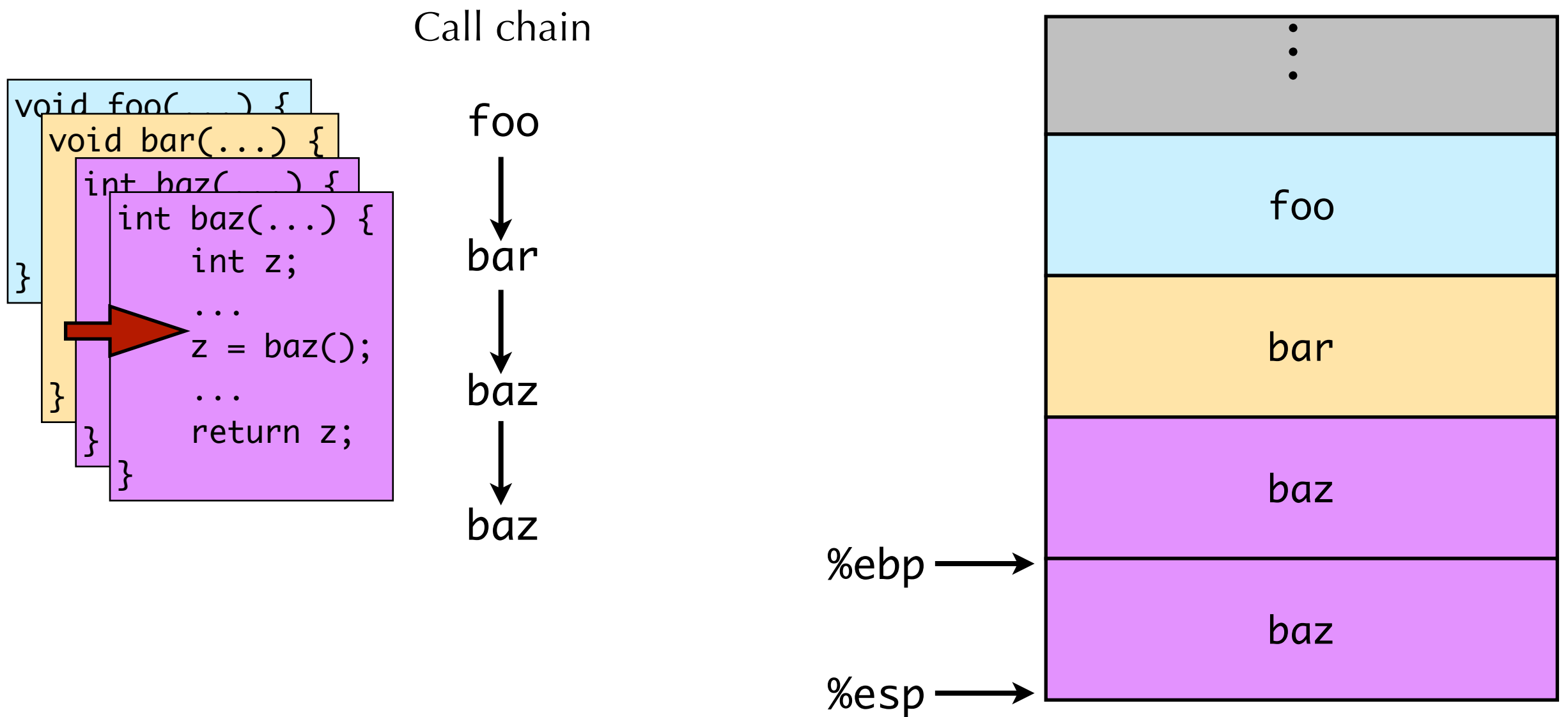
Stack frame example



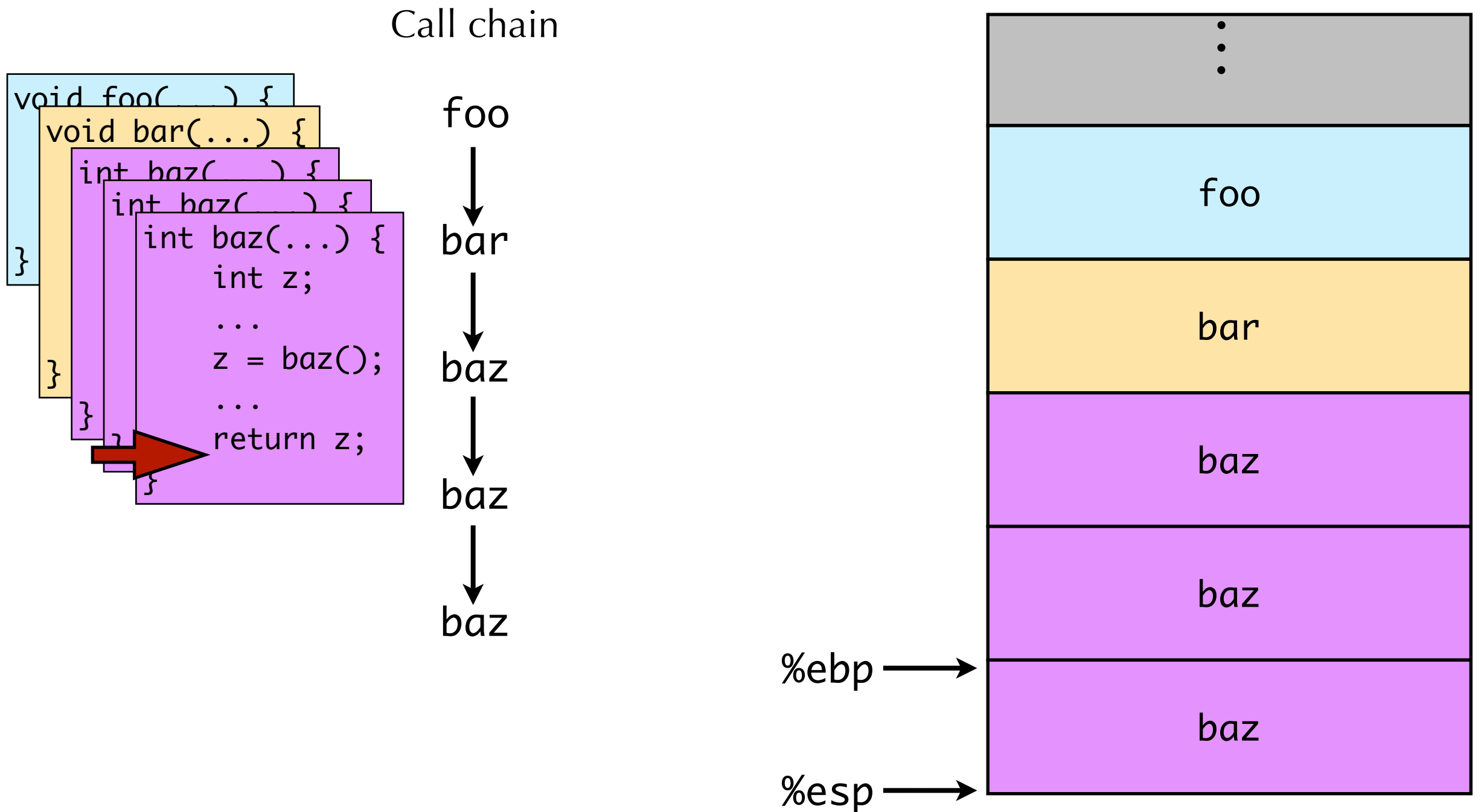
Call chain



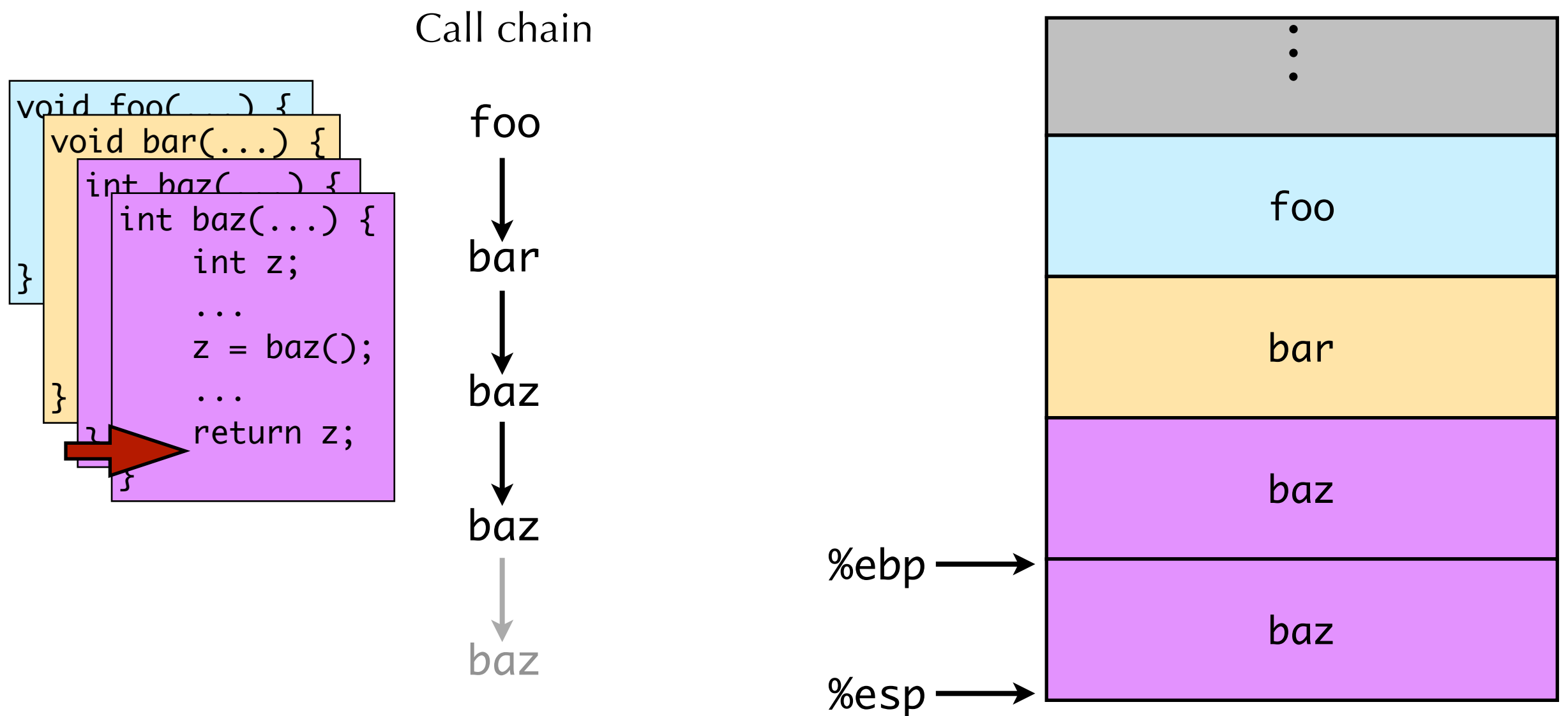
Stack frame example



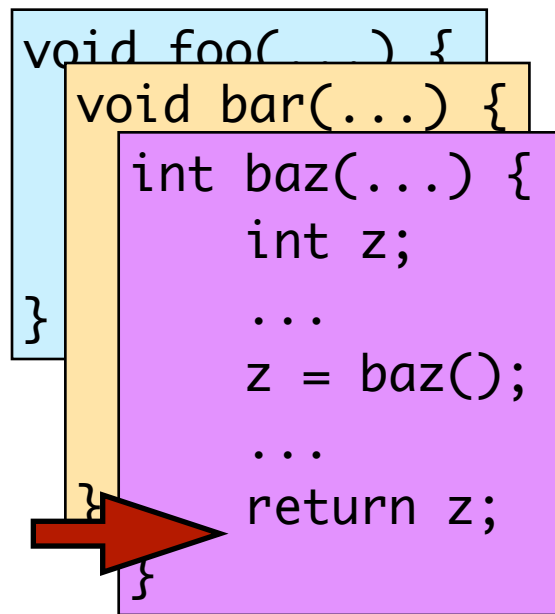
Stack frame example



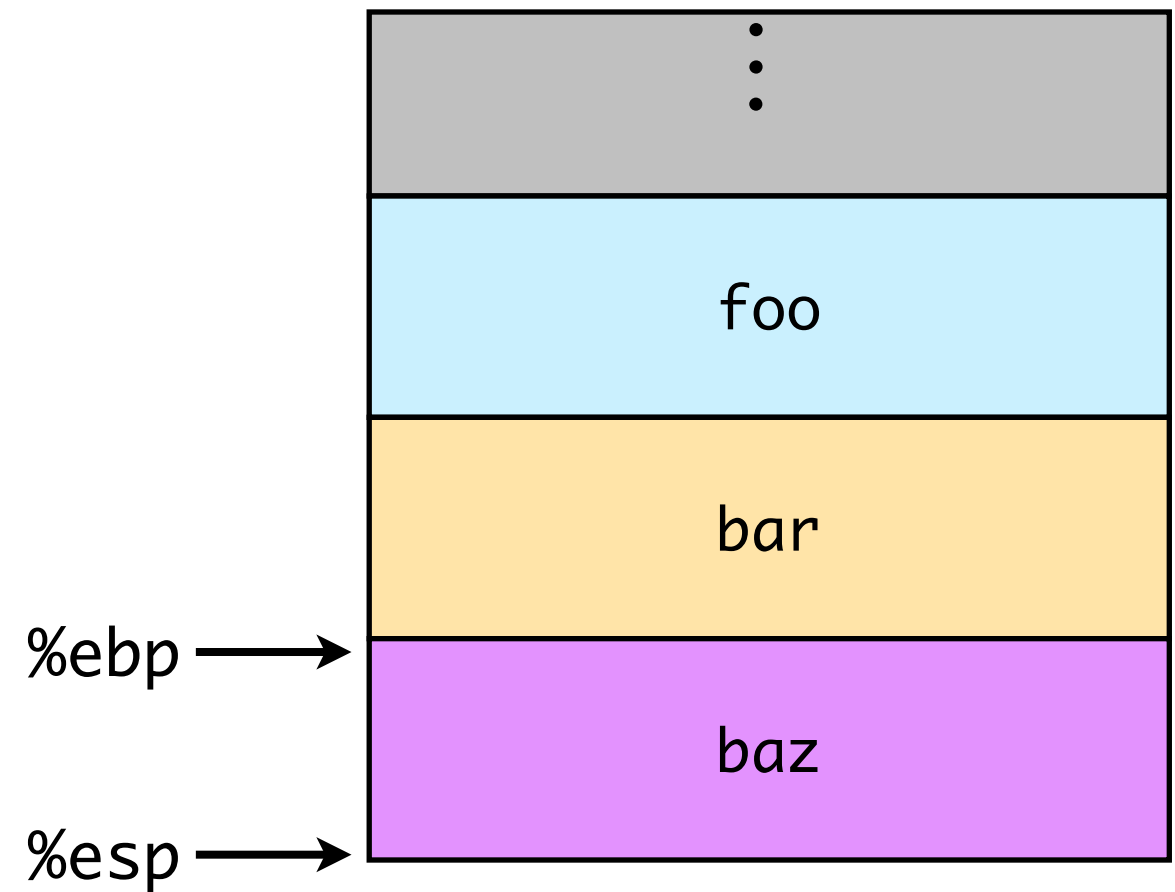
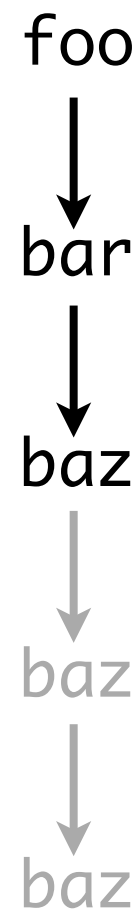
Stack frame example



Stack frame example

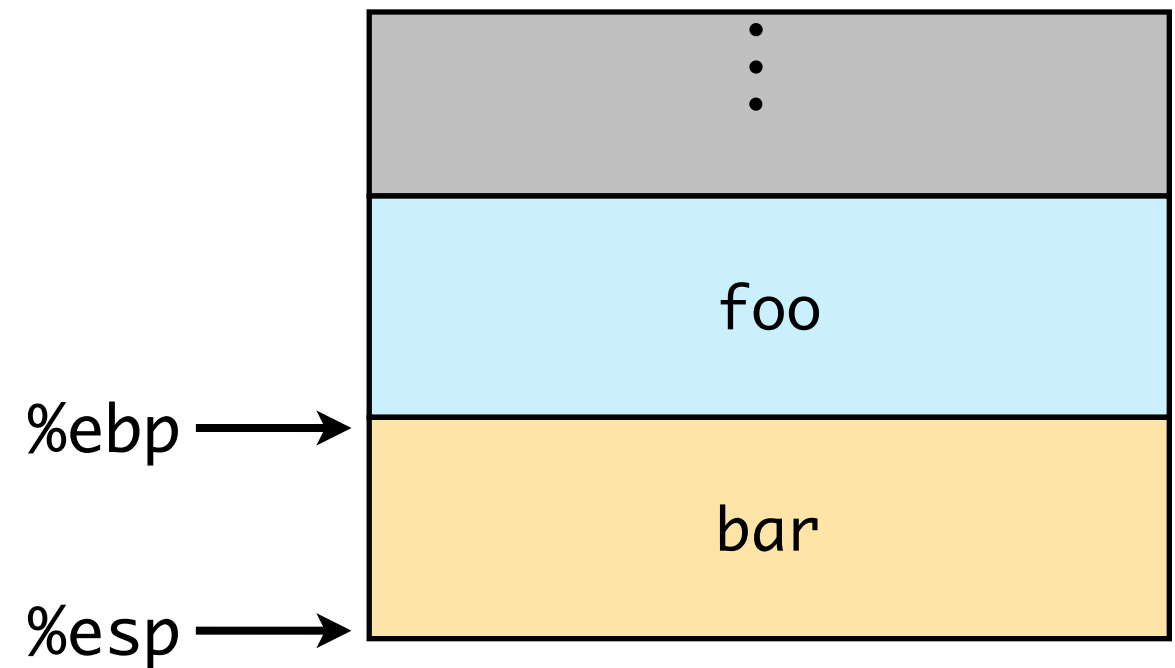
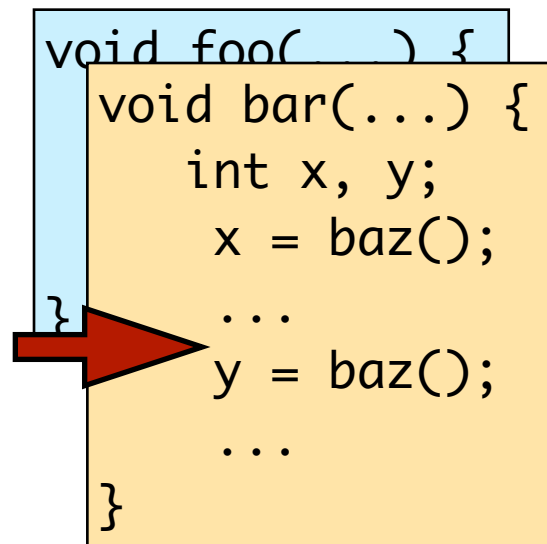


Call chain

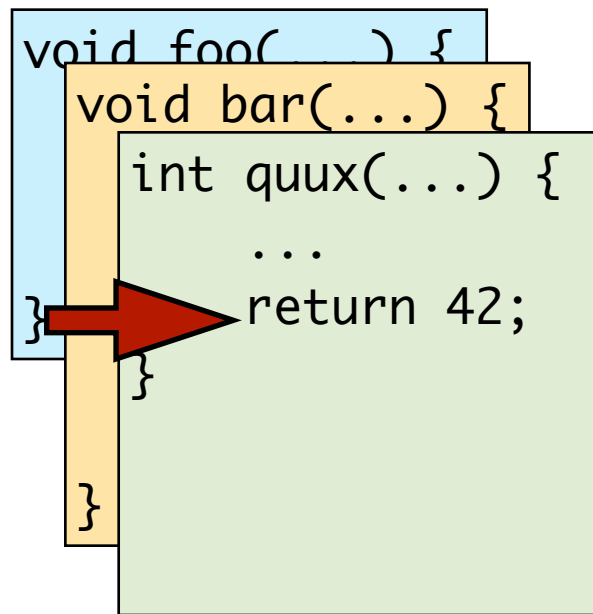


Stack frame example

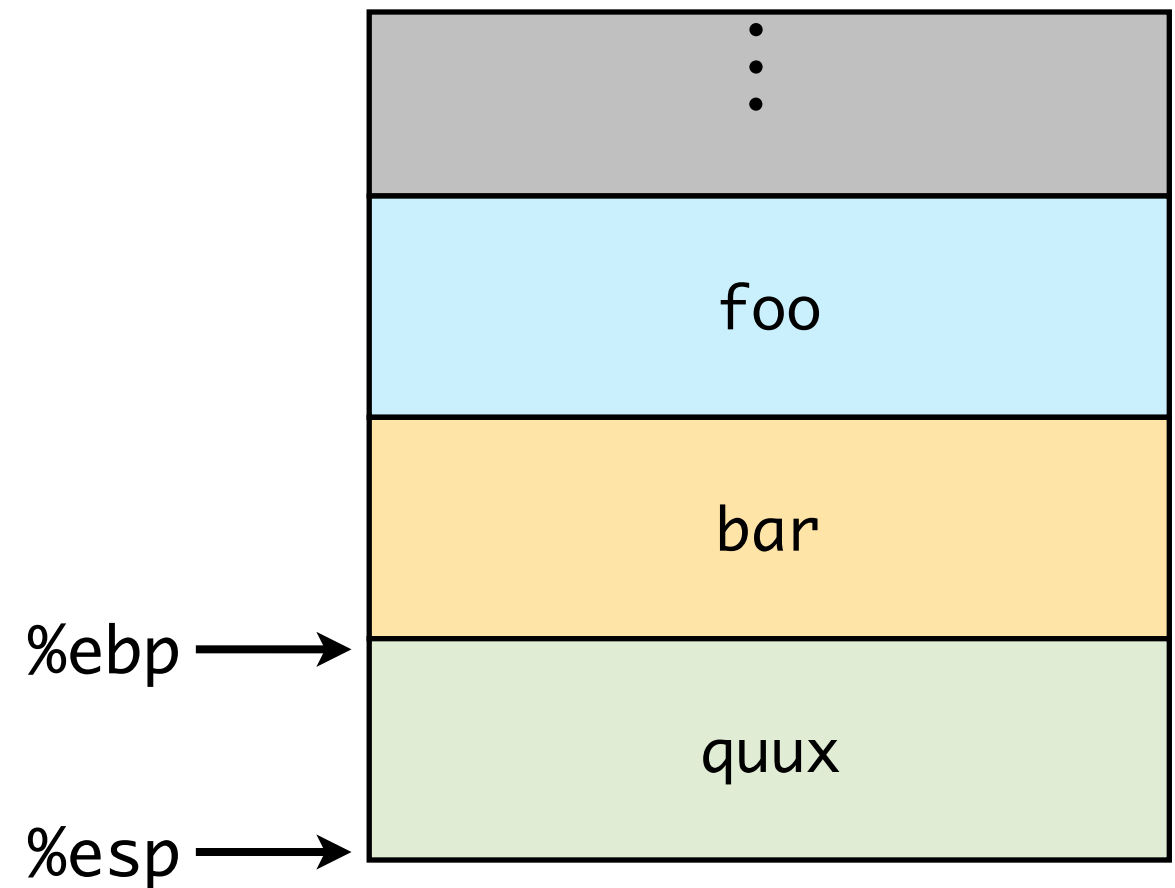
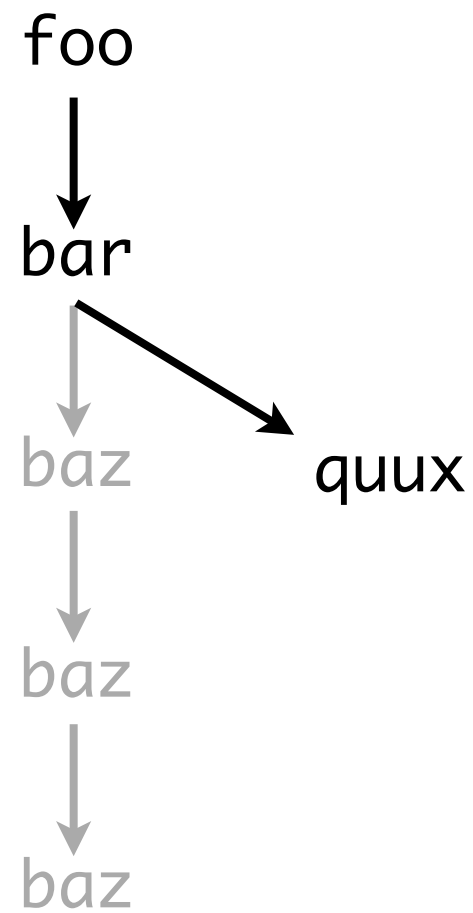
Call chain



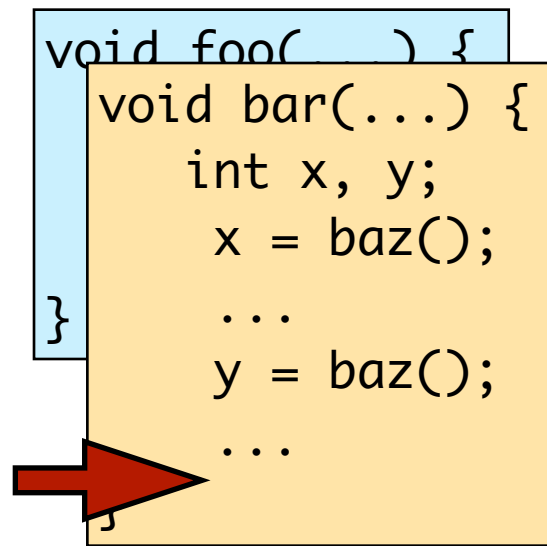
Stack frame example



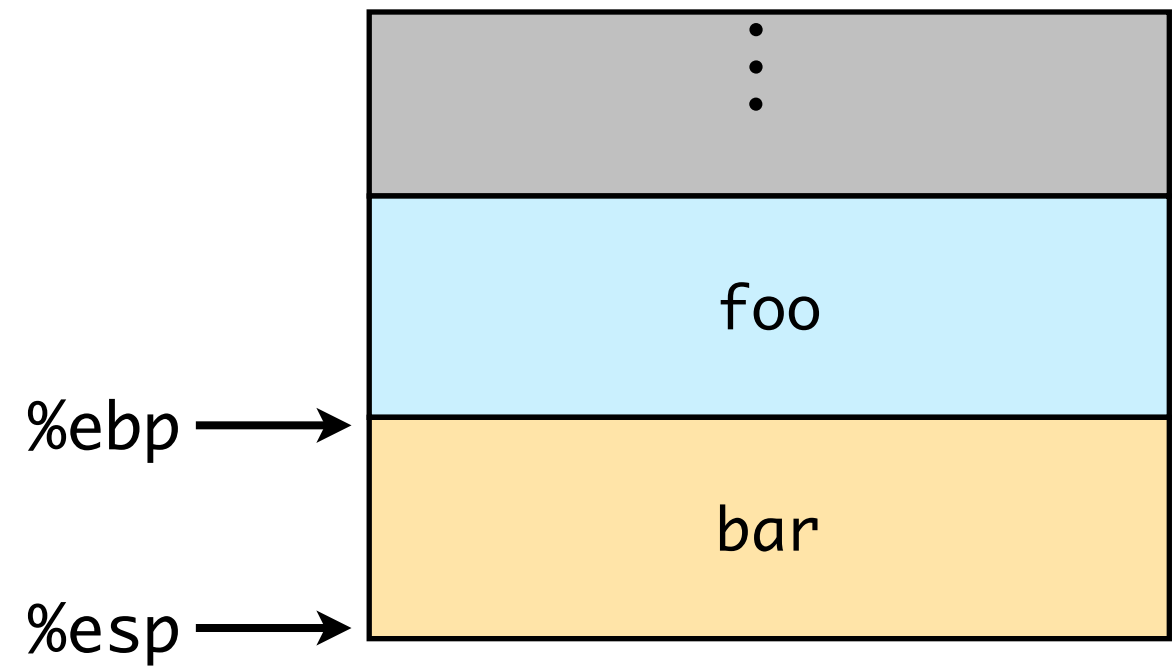
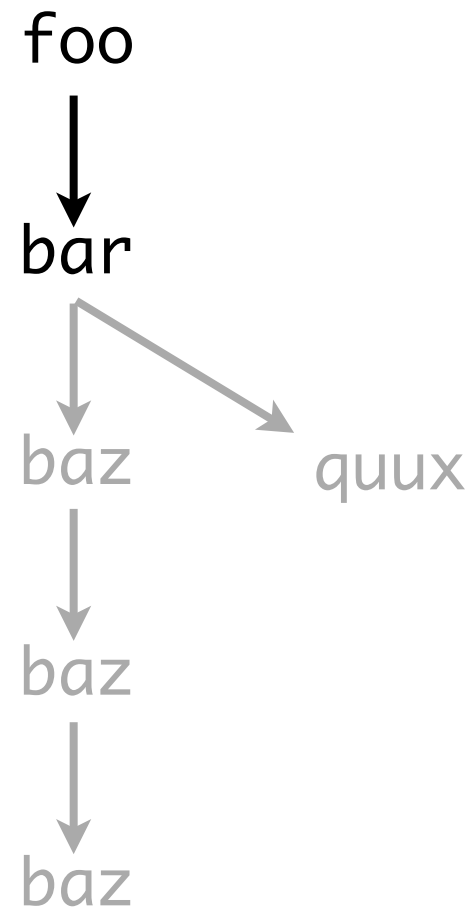
Call chain



Stack frame example




Call chain

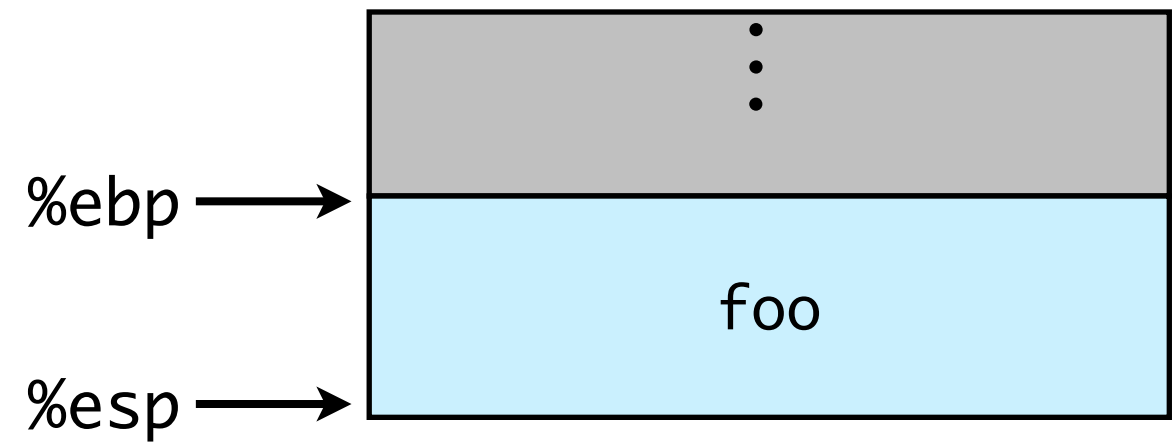
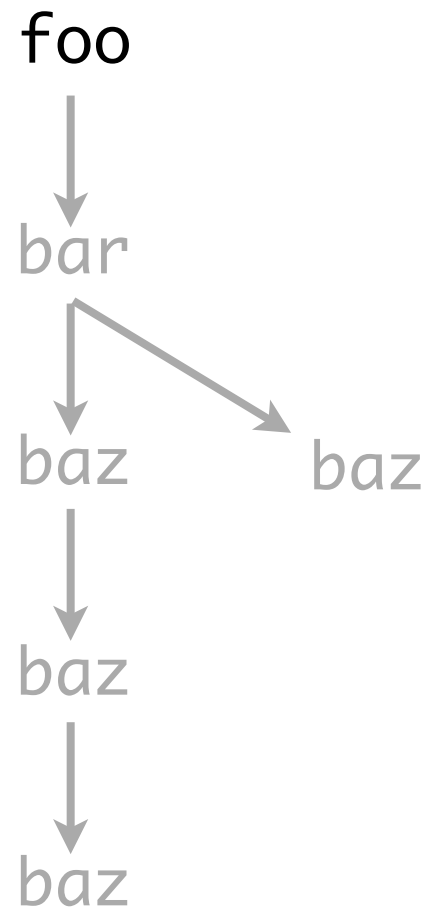


Stack frame example

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

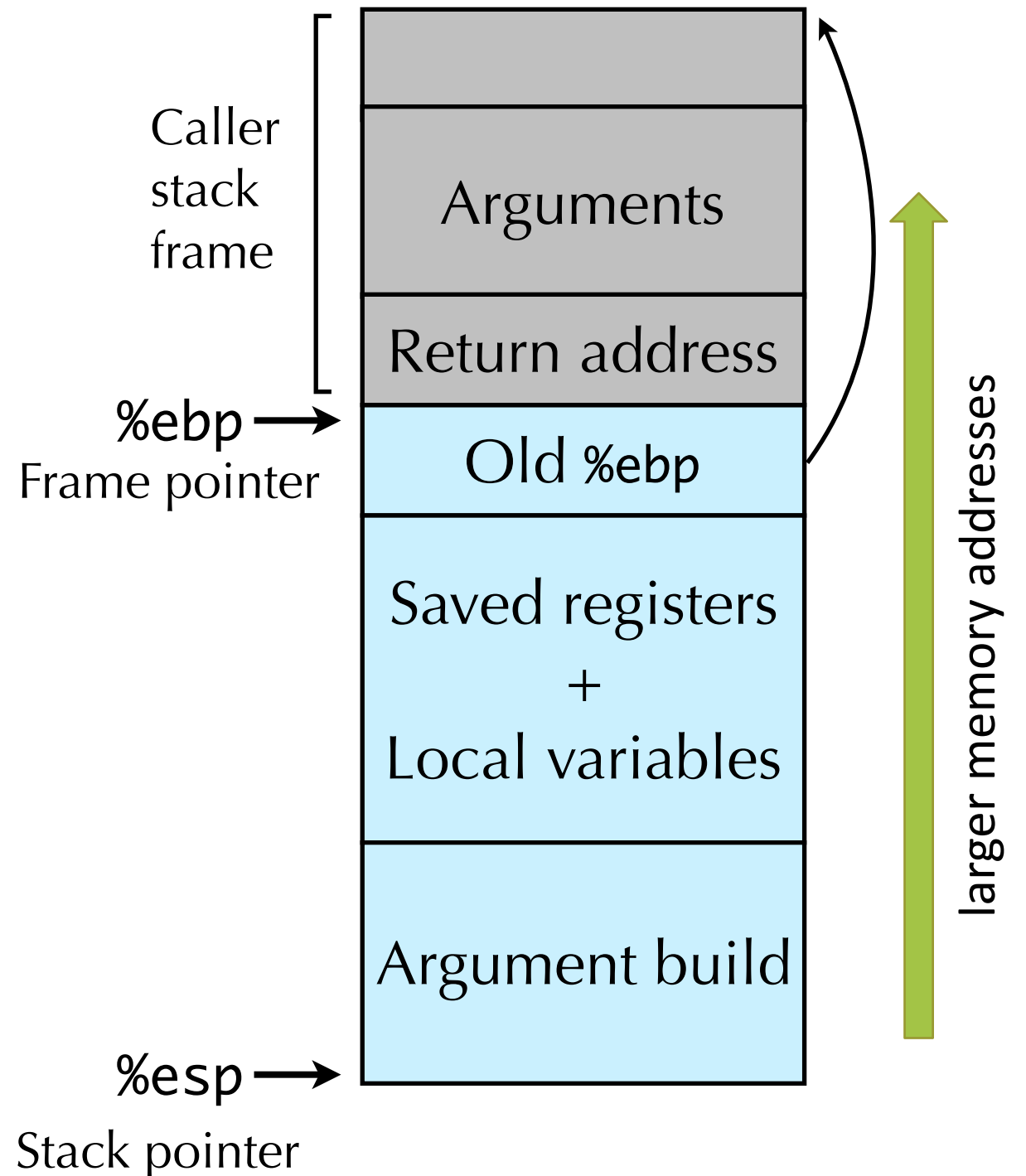


Call chain



Call Stacks: Example

- Stack frame:
 - Functions arguments
 - Local variable storage
 - Return address
 - Link (or “frame”) pointer



Call Stacks: Caller's protocol

- Function call:

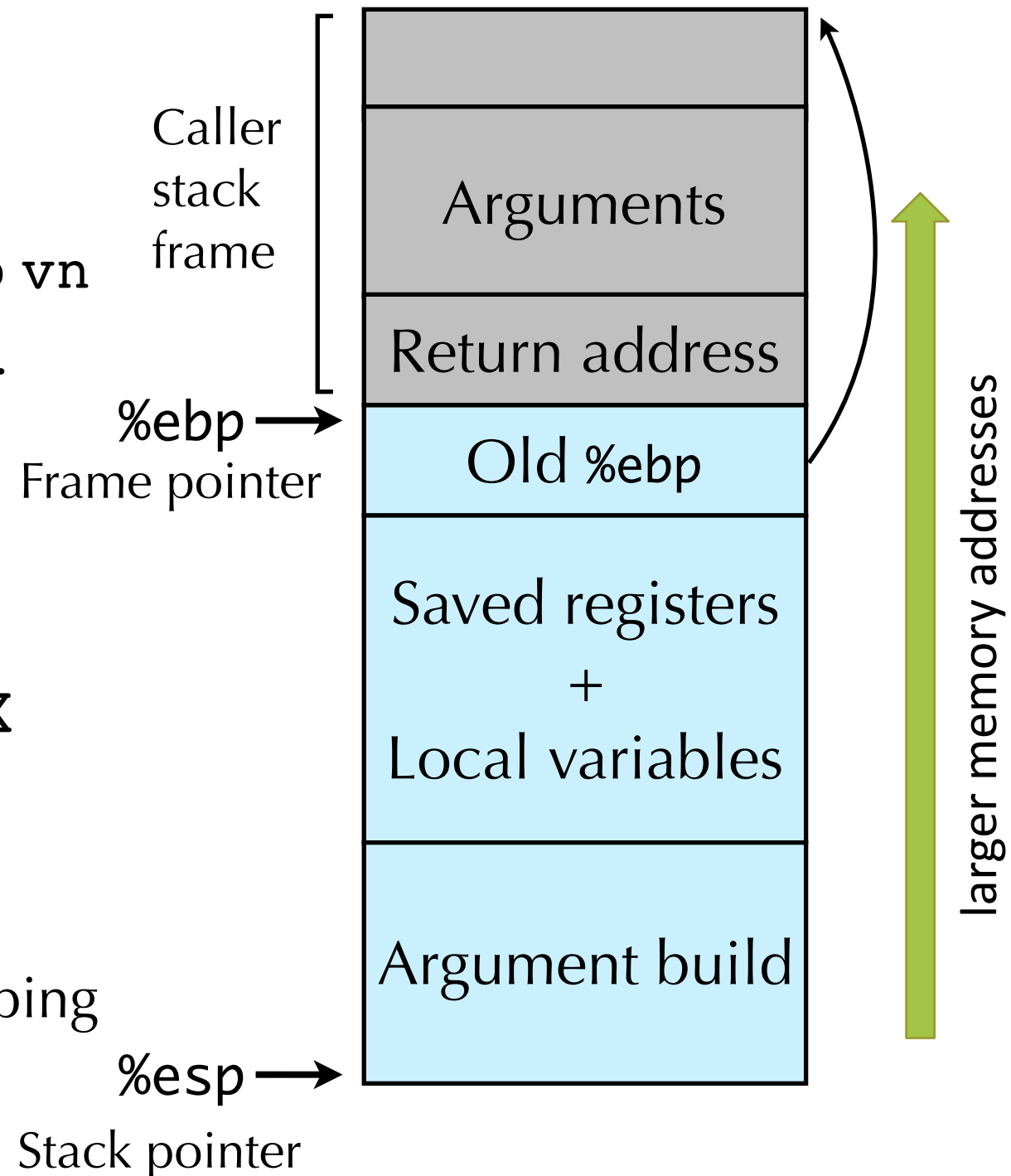
$f(e_1, e_2, \dots, e_n);$

- 1. Save caller-save registers
- 2. Evaluate e_1 to v_1 , e_2 to v_2 , ..., e_n to v_n
- 3. Push v_n to v_1 onto the top of the stack.
- 4. Use `Call` to jump to the code for f
 - pushing the return address onto the stack.

- Invariant: returned value passed in `EAX`

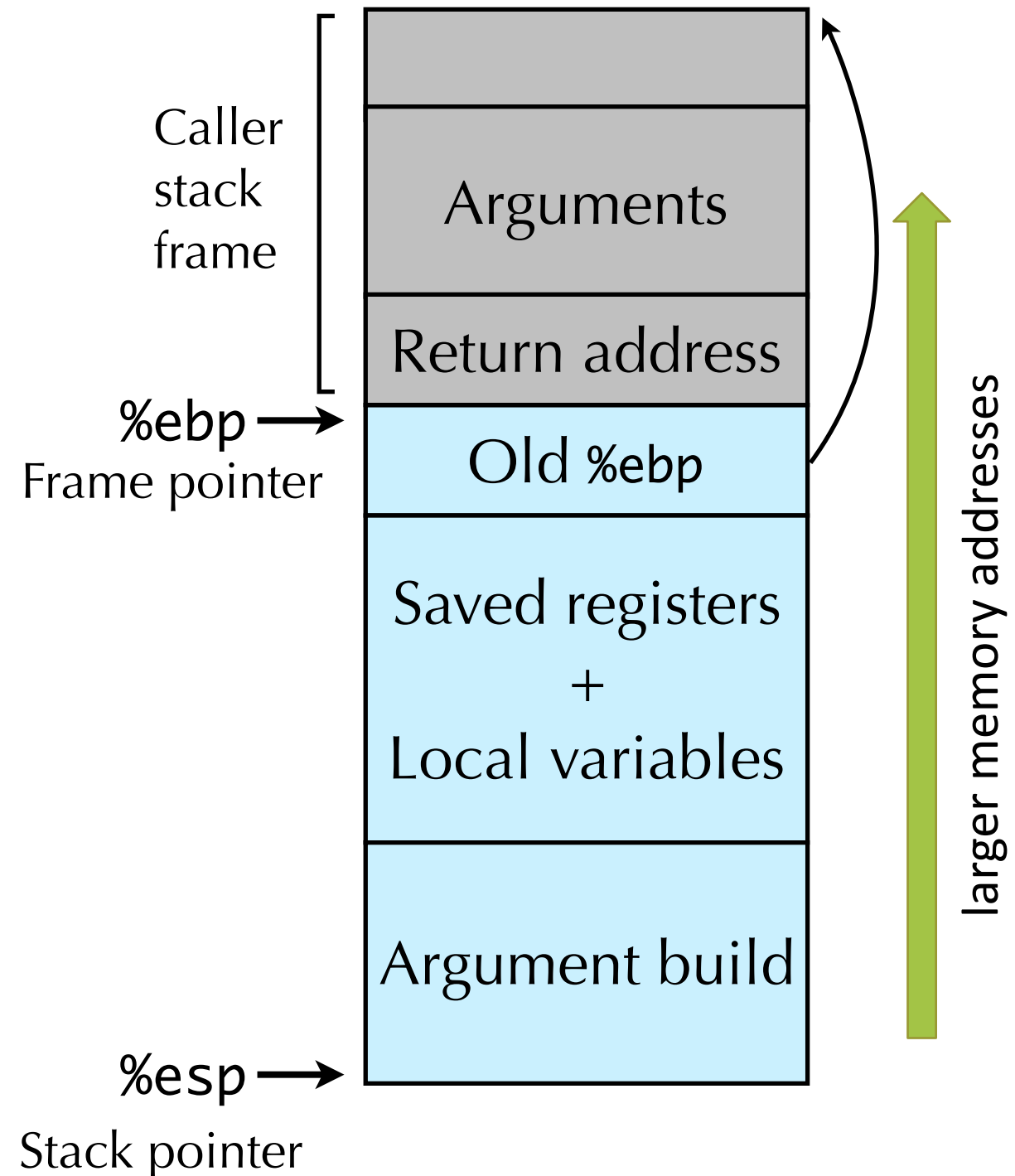
- After call:

- 1. clean up the pushed arguments by popping the stack.
- 2. Restore caller-saved registers



Call Stacks: Callee's protocol

- On entry:
 - 1. Save old frame pointer
 - EBP is callee save
 - 2. Create new frame pointer
 - `Mov(Esp, Ebp)`
 - 3. Allocate stack space for local variables.
- Invariants: (assuming word-size values)
 - Function argument n is located at:
$$\text{EBP} + (1 + n) * 4$$
 - Local variable j is located at:
$$\text{EBP} - j * 4$$
- On exit:
 - 1. Pop local storage
 - 2. Restore EBP



Example: swap

```
/* 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 = t1;  
    *yp = t0;  
}
```

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


Example: swap

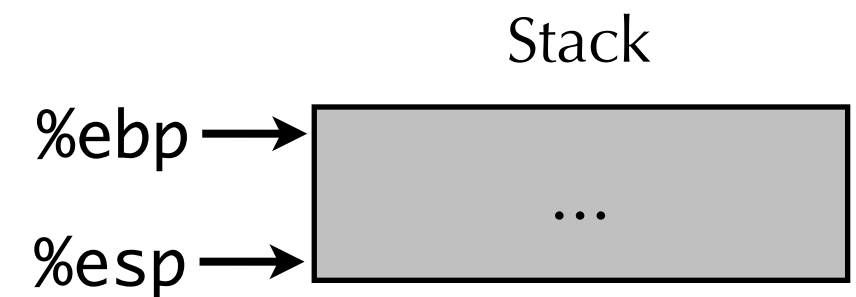
```
/* Global vars */
int zip1 = 15213;
int zip2 = 91125;

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

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

```
call_swap:
```

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



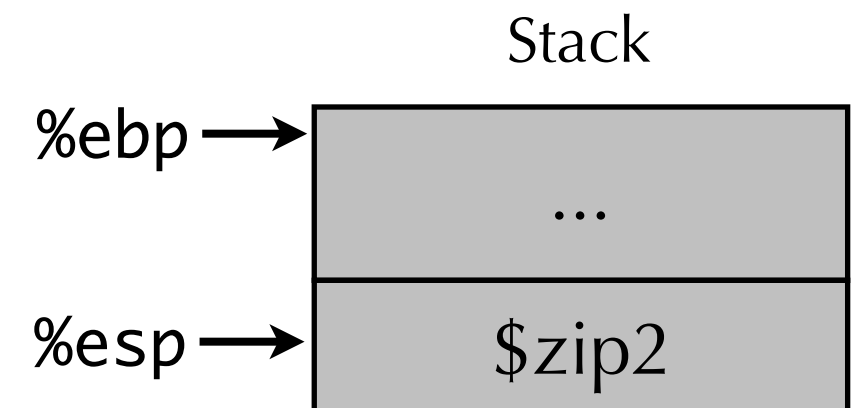
Example: swap

```
/* 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 = t1;
    *yp = t0;
}
```

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

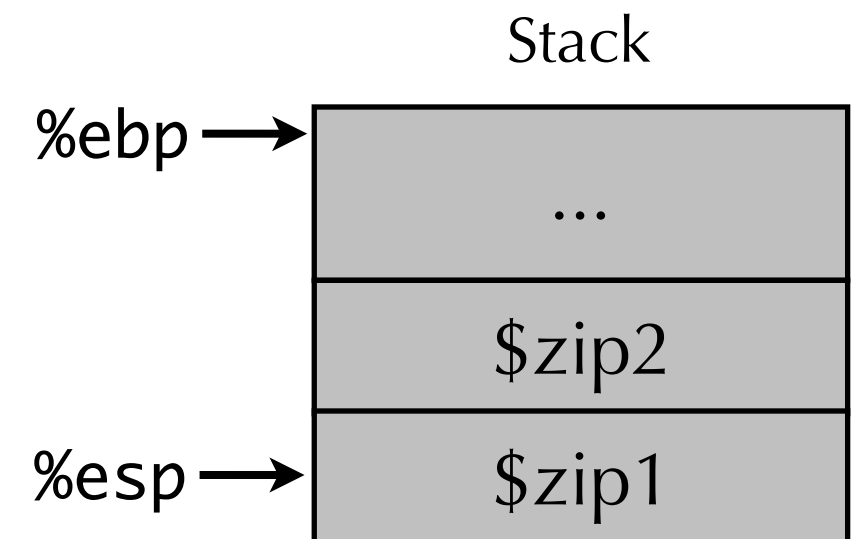


Example: swap

```
/* 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 = t1;  
    *yp = t0;  
}
```

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

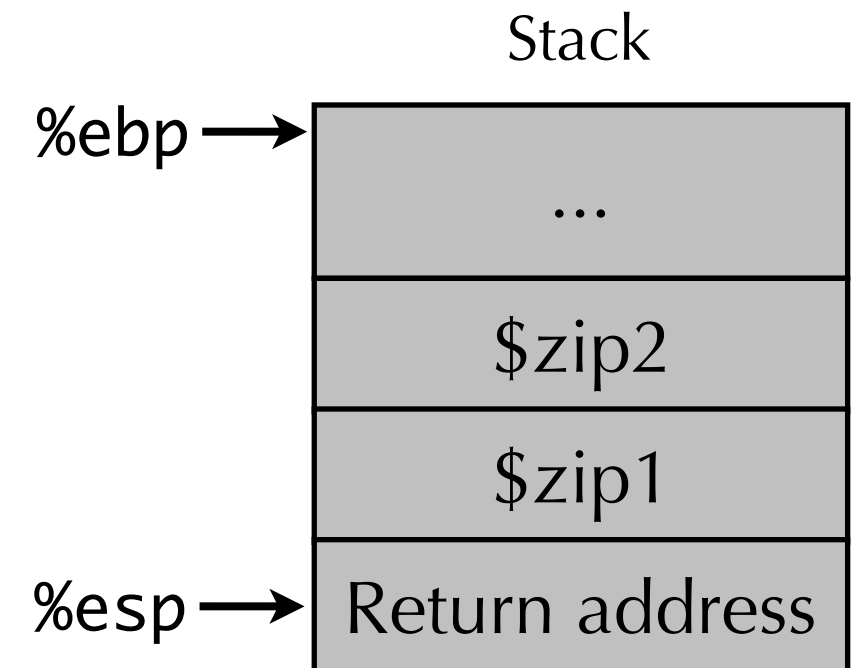


Example: swap

```
/* 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 = t1;  
    *yp = t0;  
}
```

```
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 = 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
```

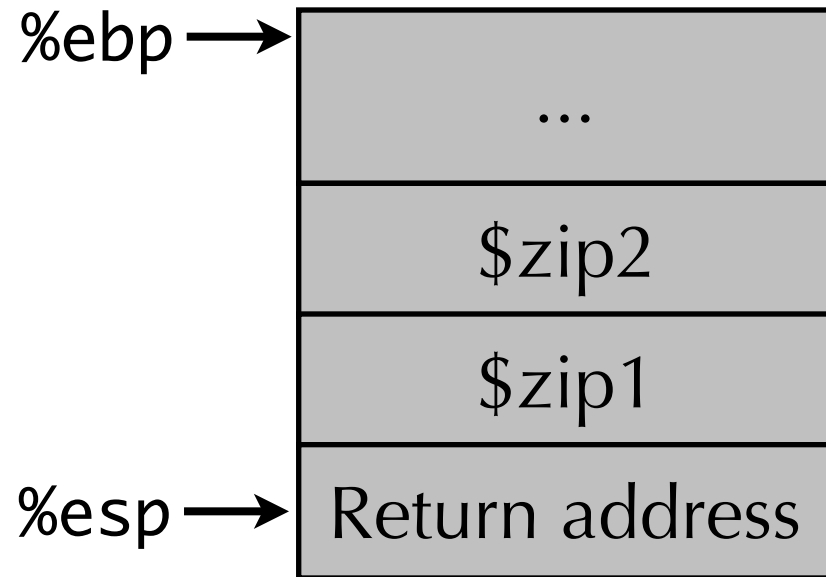
Set up

Body

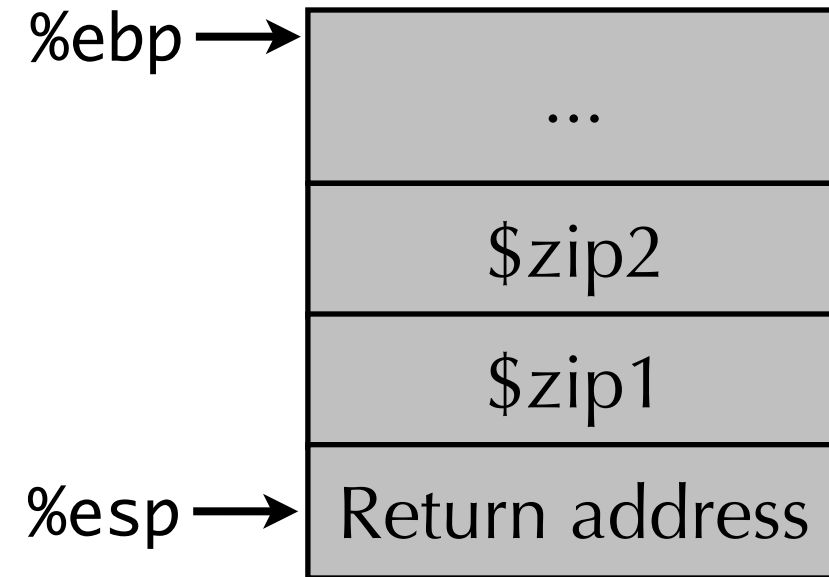
Finish

Swap setup

Stack entering swap



Resulting stack

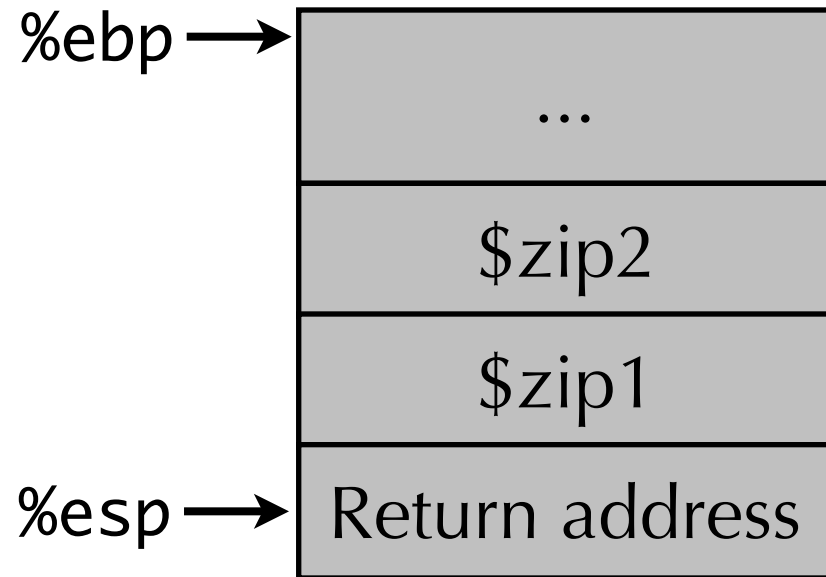


```
pushl %ebp  
movl %esp,%ebp  
pushl %ebx
```

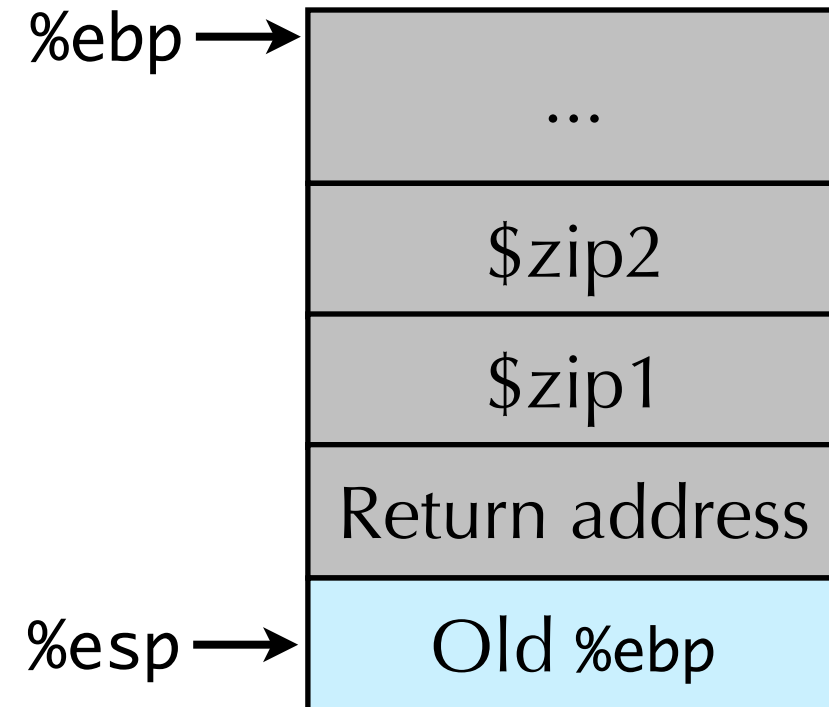
Set up

Swap setup

Stack entering swap



Resulting stack

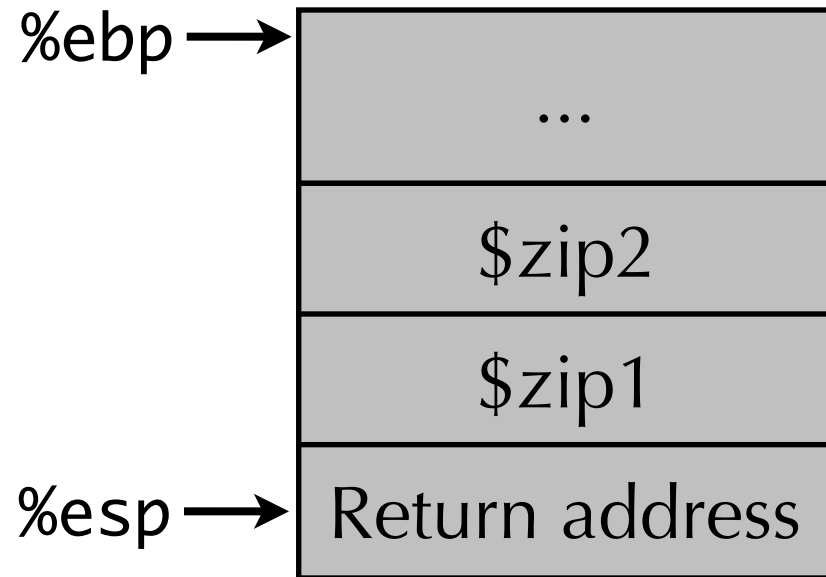


```
pushl %ebp  
movl %esp,%ebp  
pushl %ebx
```

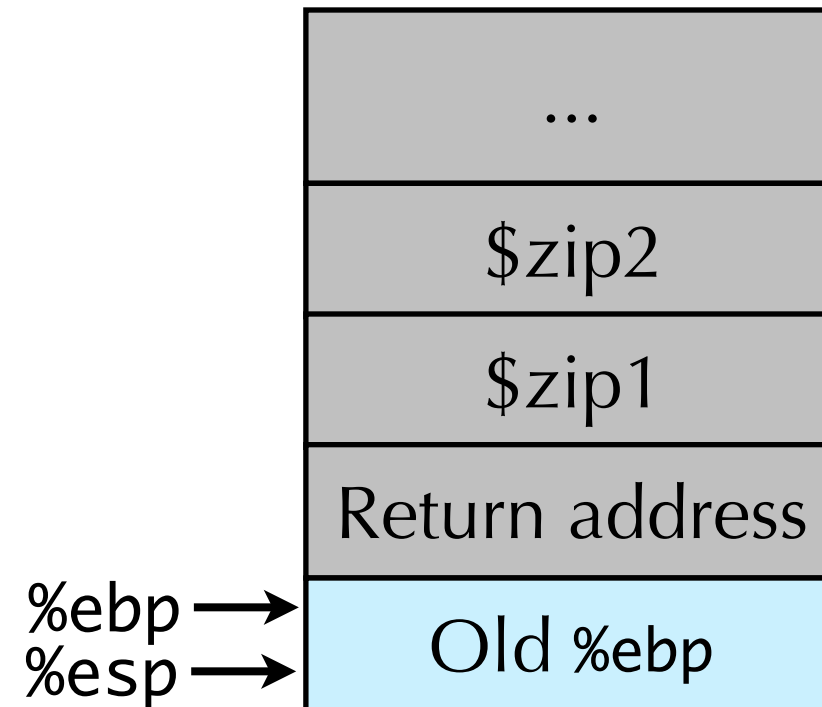
Set up

Swap setup

Stack entering swap



Resulting stack

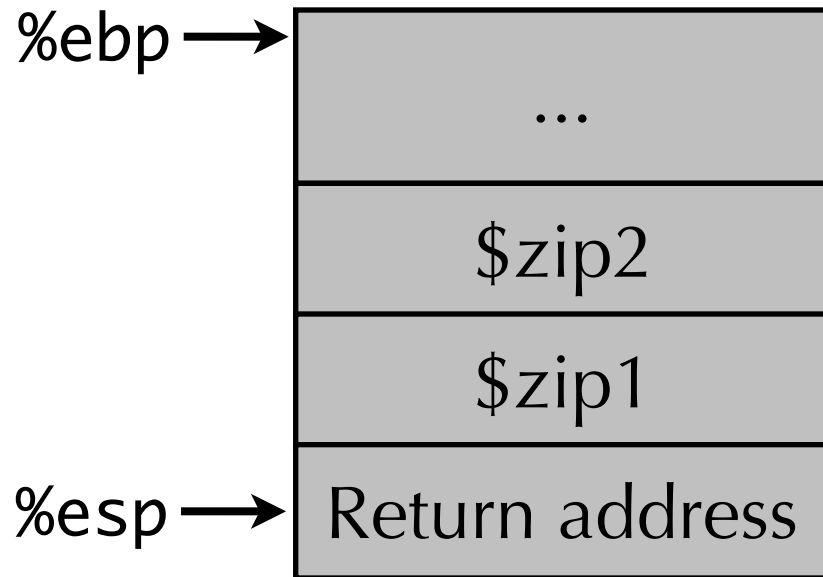


```
pushl %ebp  
movl %esp,%ebp  
pushl %ebx
```

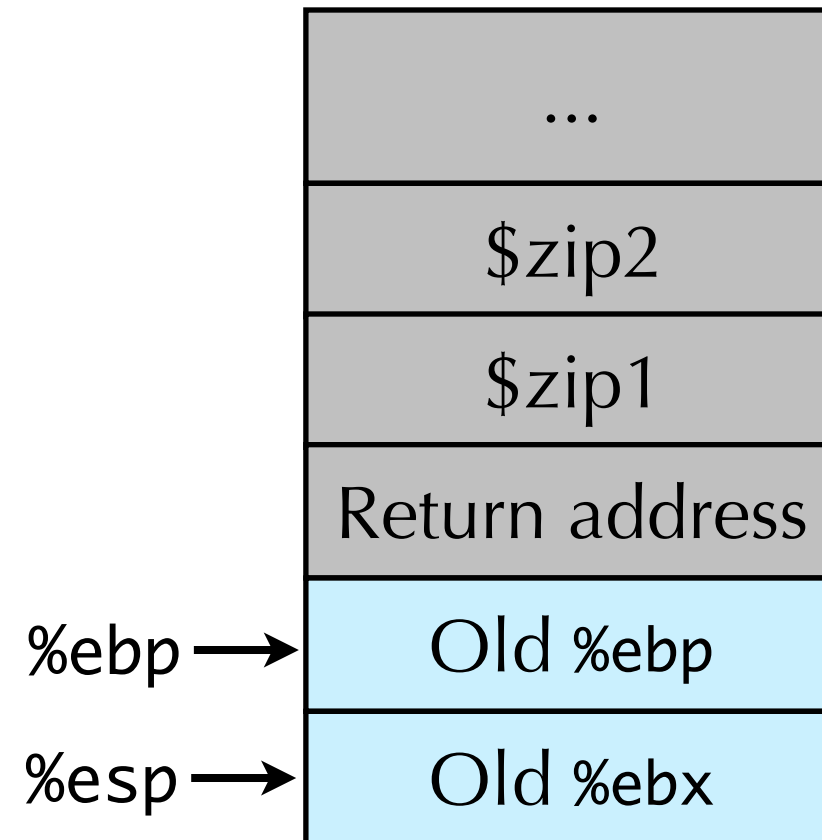
Set up

Swap setup

Stack entering swap



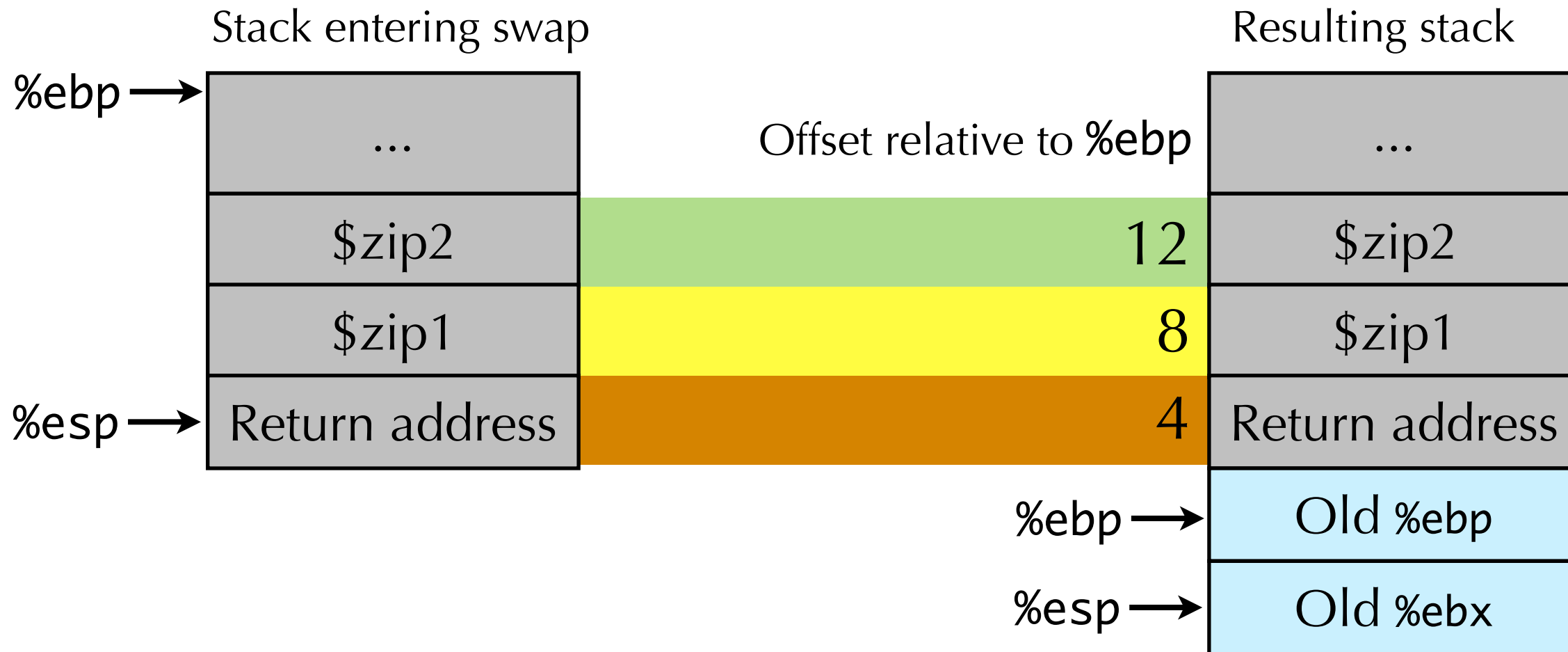
Resulting stack



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

Set up

Swap body

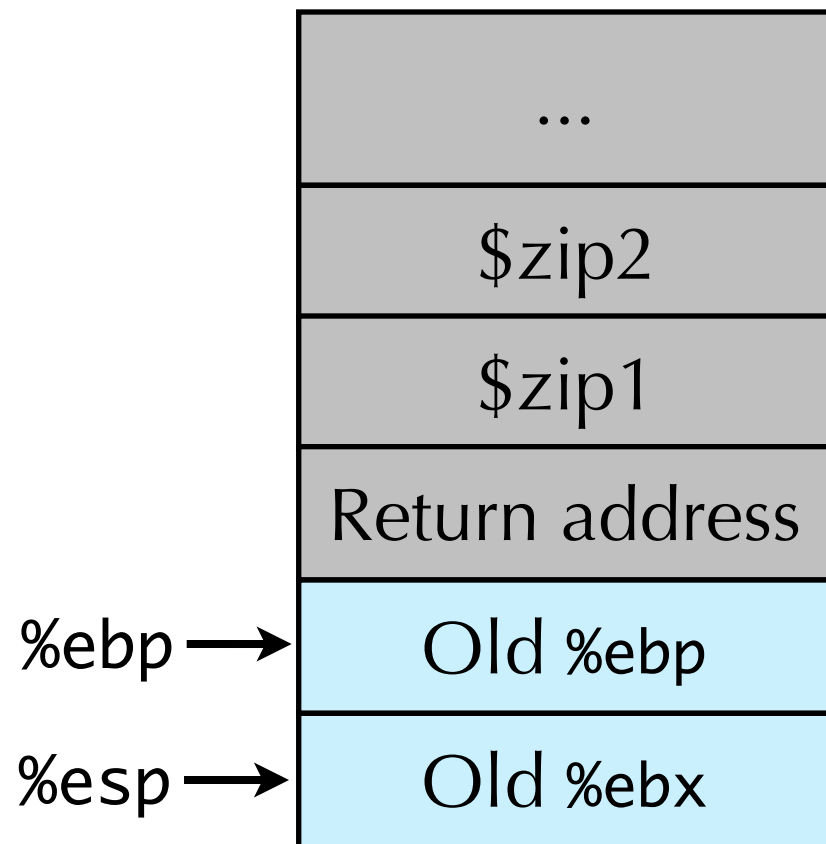


```
movl 12(%ebp),%ecx
movl 8(%ebp),%edx
movl (%ecx),%eax
movl (%edx),%ebx
movl %eax,(%edx)
movl %ebx,(%ecx)
```

Body

Swap finish

Stack at end swap body

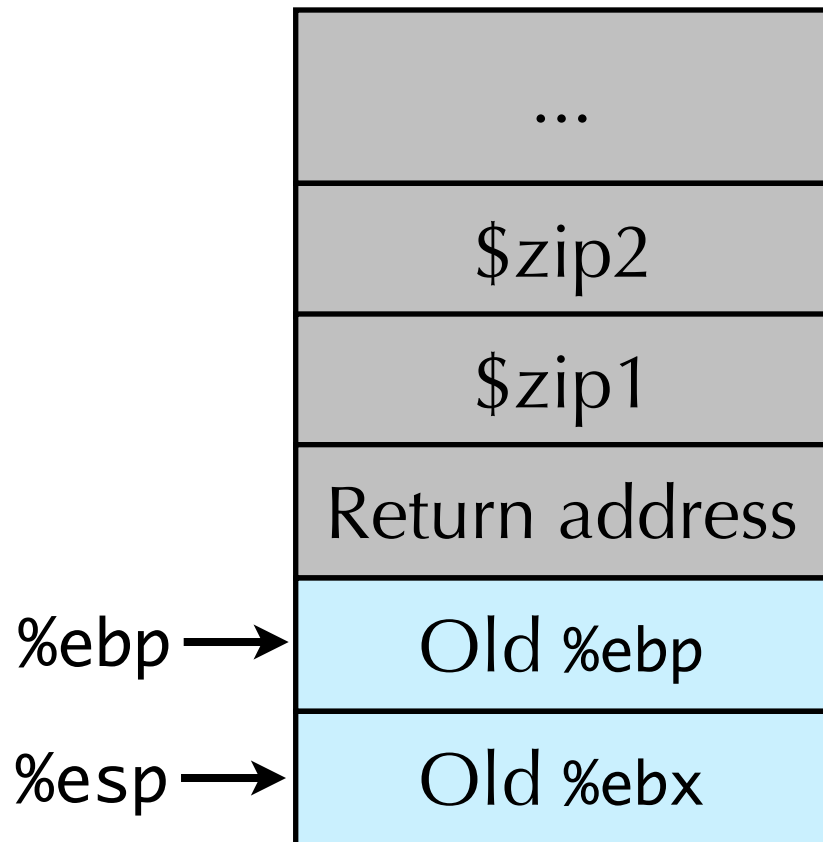


```
movl -4(%ebp),%ebx  
movl %ebp,%esp  
popl %ebp  
ret
```

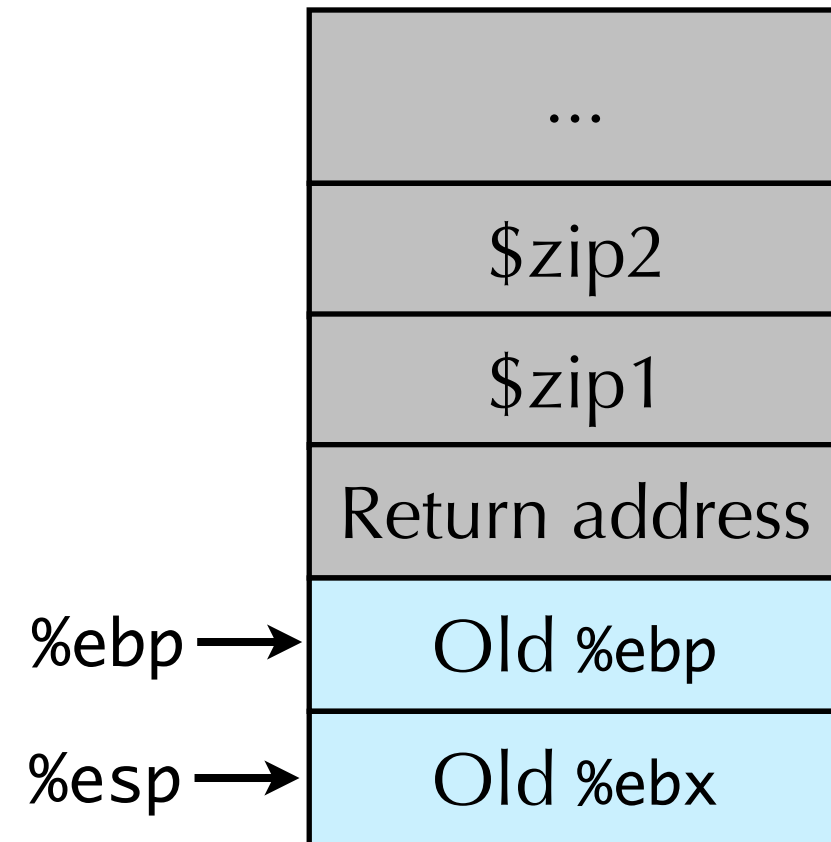
Finish

Swap finish

Stack at end swap body



Resulting stack

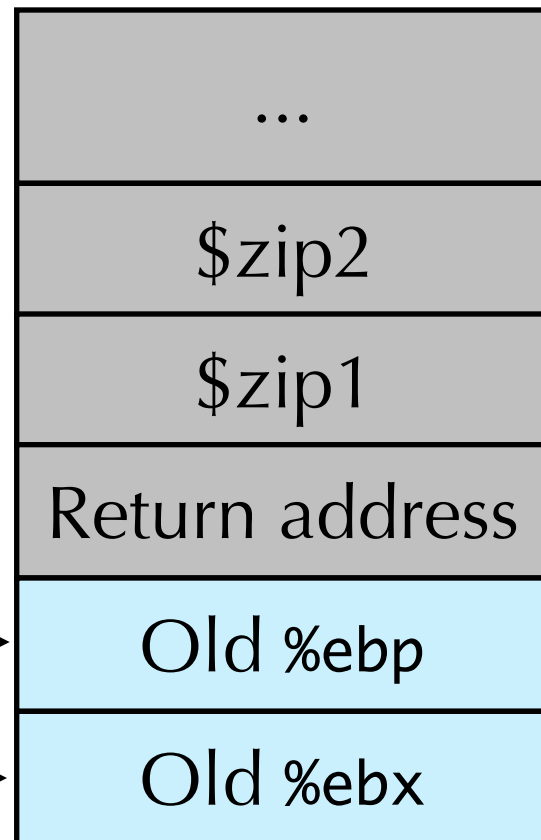


```
movl -4(%ebp),%ebx  
movl %ebp,%esp  
popl %ebp  
ret
```

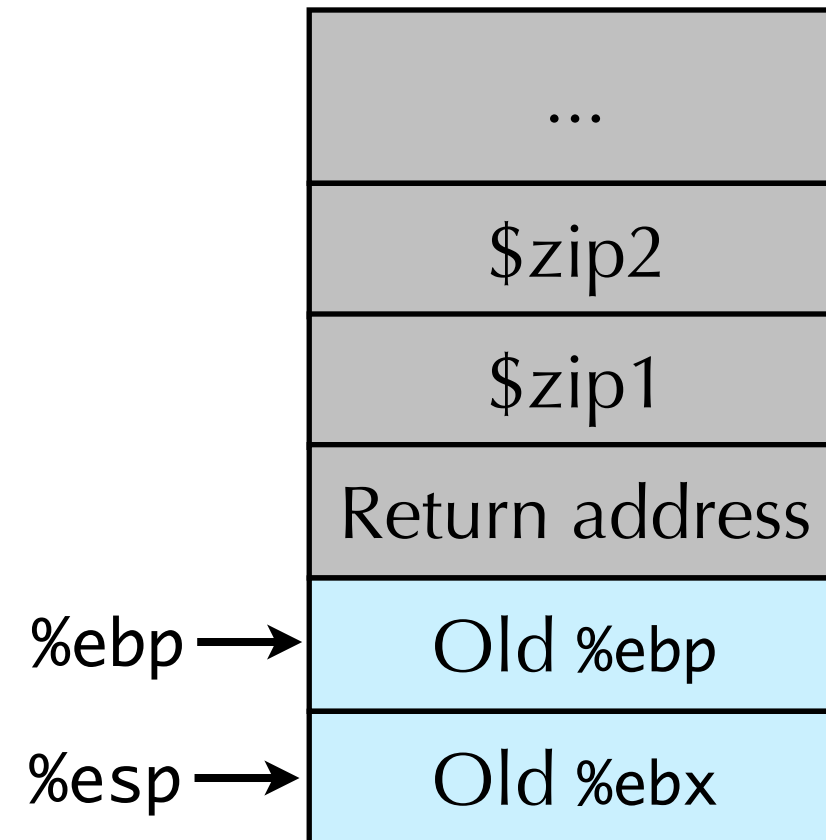
Finish

Swap finish

Stack at end swap body



Resulting stack



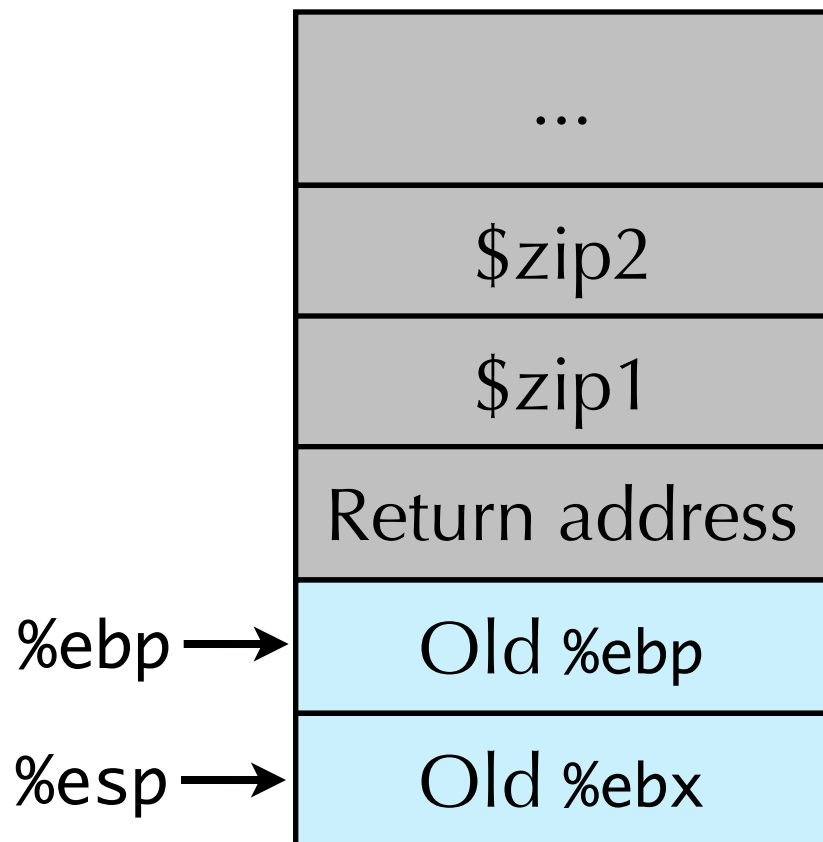
```
movl -4(%ebp),%ebx  
movl %ebp,%esp  
popl %ebp  
ret
```

Finish

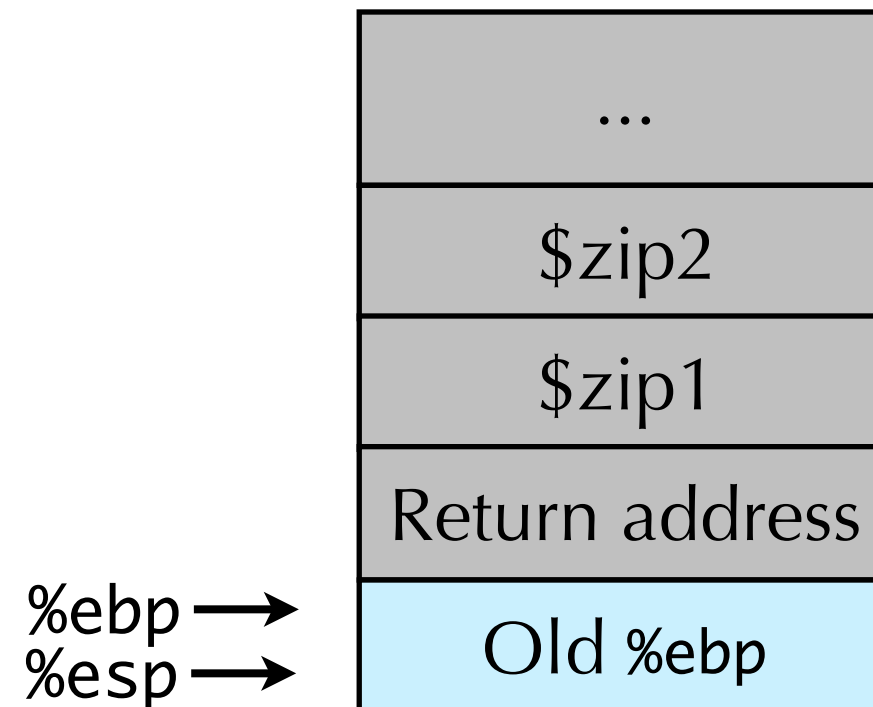
Restores old value of %ebx!

Swap finish

Stack at end swap body



Resulting stack

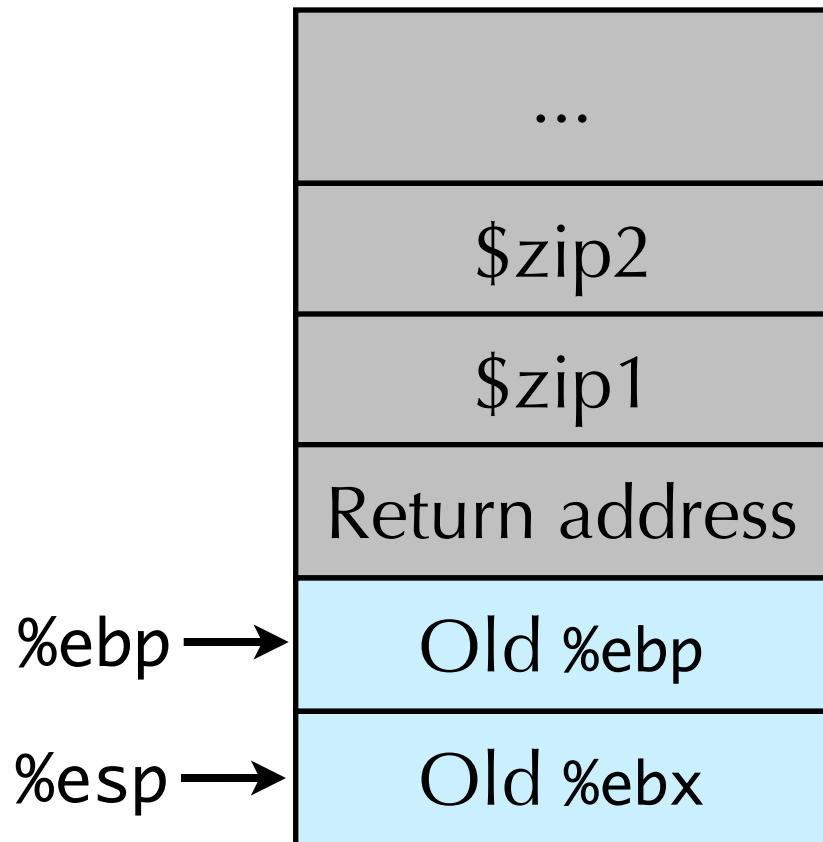


```
movl -4(%ebp),%ebx  
movl %ebp,%esp  
popl %ebp  
ret
```

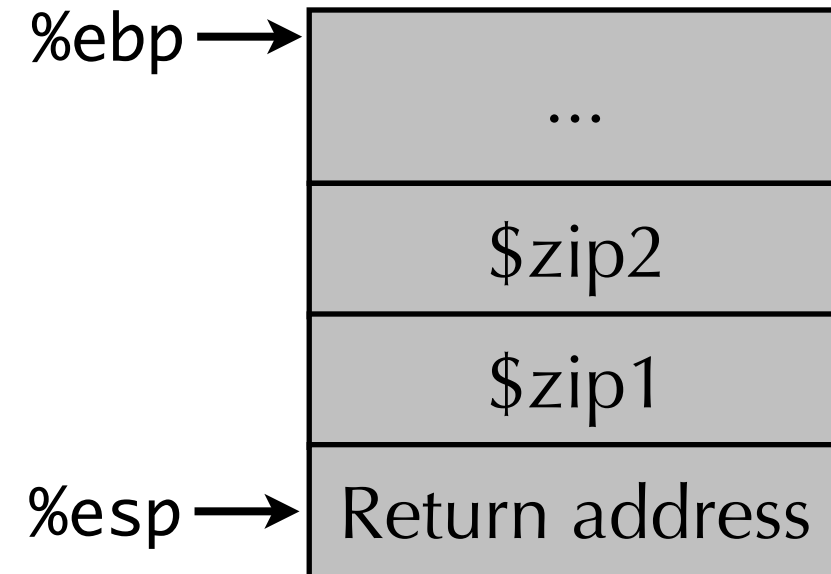
Finish

Swap finish

Stack at end swap body



Resulting stack

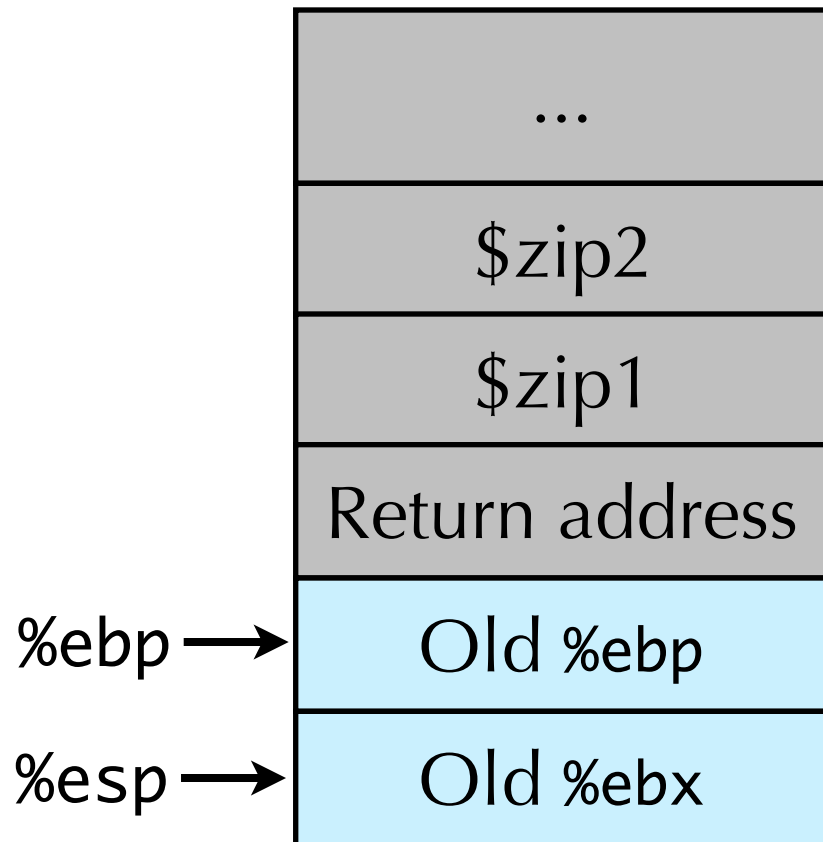


```
movl -4(%ebp),%ebx  
movl %ebp,%esp  
popl %ebp  
ret
```

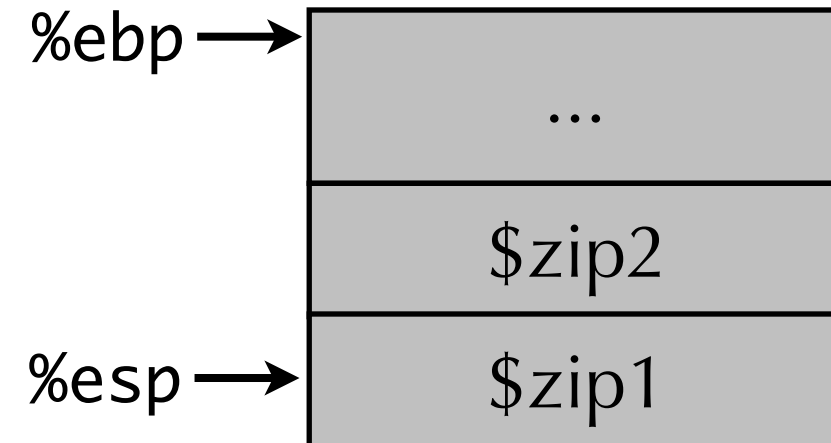
Finish

Swap finish

Stack at end swap body



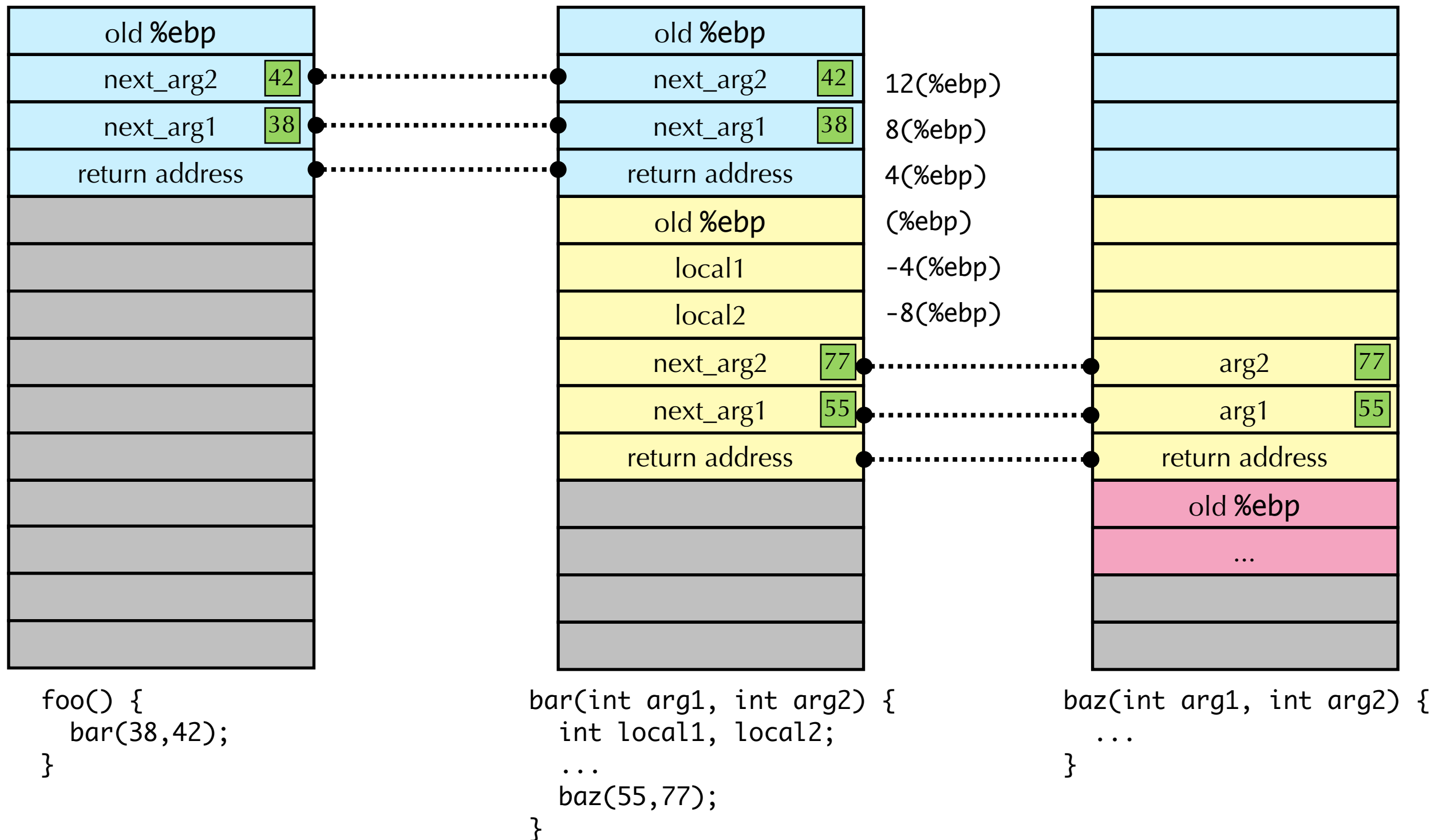
Resulting stack



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

Finish

Stack frame cheat sheet



X86-64 SYSTEM V AMD 64 ABI

- More modern variant of C calling conventions
 - used on Linux, Solaris, BSD, OS X
- Callee save: `rbp`, `rbx`, `r12-r15`
- Caller save: all others
- Parameters 1 .. 6 go in: `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`
- Parameters 7+ go on the stack (in right-to-left order)
 - so: for $n > 6$, the n th argument is located at $((n-7)+2)*8 + \text{rbp}$
- Return value: in `rax`
- 128 byte “red zone” – scratch pad for the callee's data
 - 128 bytes beyond `rsp` is considered reserved memory
 - Not modified by signal or interrupt handlers
 - Callee can use this for temporary data not needed across function calls