# Motivation



- We will cover 3 main topics: memory safety, cryptography functions, and web/network securities
  - Fact check: a lot of software was written without taking security into consideration from the very beginning
- All topics lay foundations for future topics
  - Memory safety focuses on writing more secure programs (especially on creating smart contracts, where insecure code can cause catastrophic damage and loss)
  - Cryptography functions lay the foundation for block chains to verify identities, transfer fund, and execute smart contracts
  - Web/network securities topics provide guidelines and real examples to avoid exploits when developing fintech applications

# x86 Calling Convention

Adapted from CS161 Lecture 4

# Recap: Stack Layout

Textbook Chapter 2.6

# Stack Frames

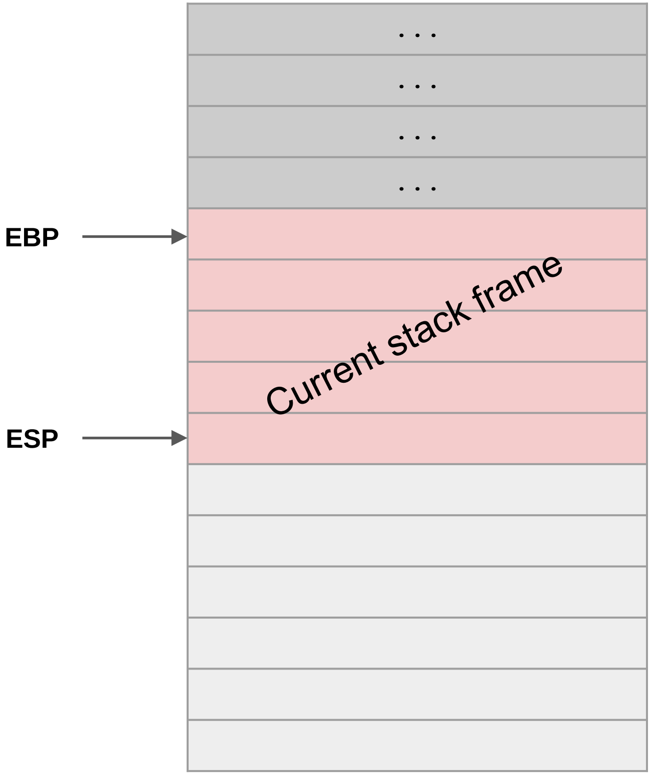


- When your code calls a function, space is made on the stack for local variables
  - This space is known as the **stack frame** for the function
  - The stack frame goes away once the function returns
- The stack starts at higher addresses. Every time your code calls a function, the stack makes extra space by growing down
  - Note: Data on the stack, such as a string, is still stored from lowest address to highest address. “Growing down” only happens when extra memory needs to be allocated.

# Stack Frames



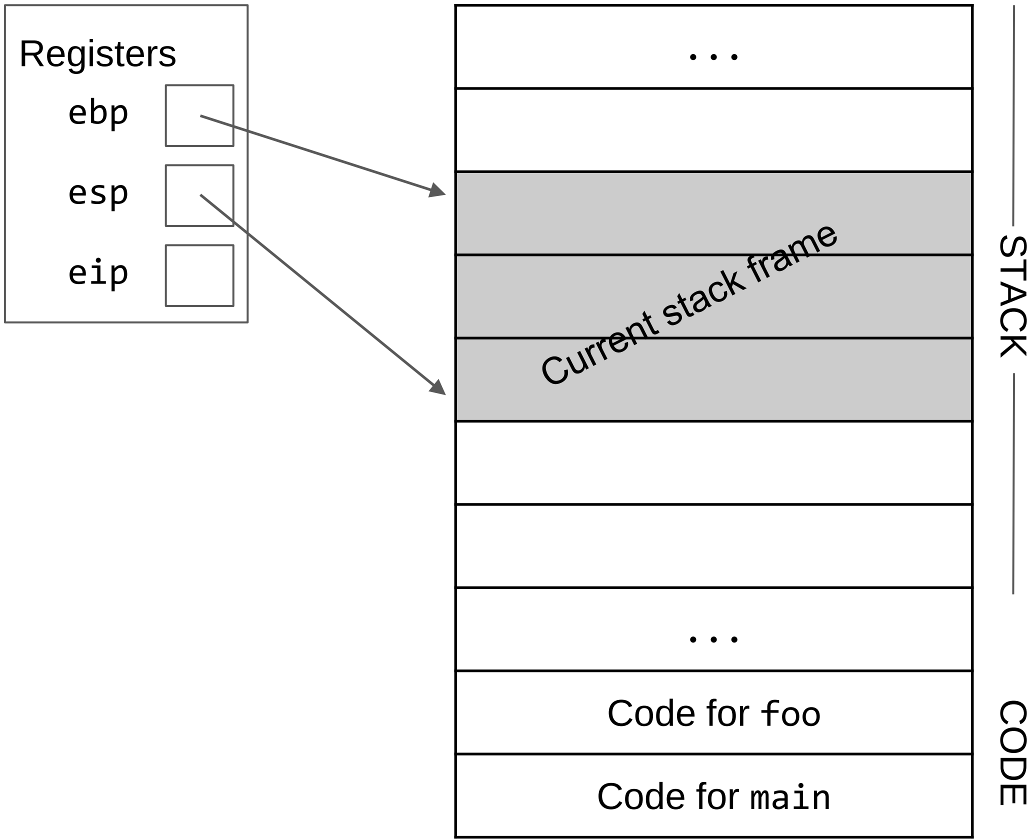
- To keep track of the current stack frame, we store two pointers in registers
  - The EBP (base pointer) register points to the ~~top~~ base of the current stack frame
    - Equivalent to RISC-V **`fp`**
  - The ESP (stack pointer) register points to the ~~bottom~~ moving part of the current stack frame
    - Equivalent to RISC-V **`sp`** (but x86 moves the stack pointer up and down a lot more than RISC-V does)



# Quick detour: storing pointers

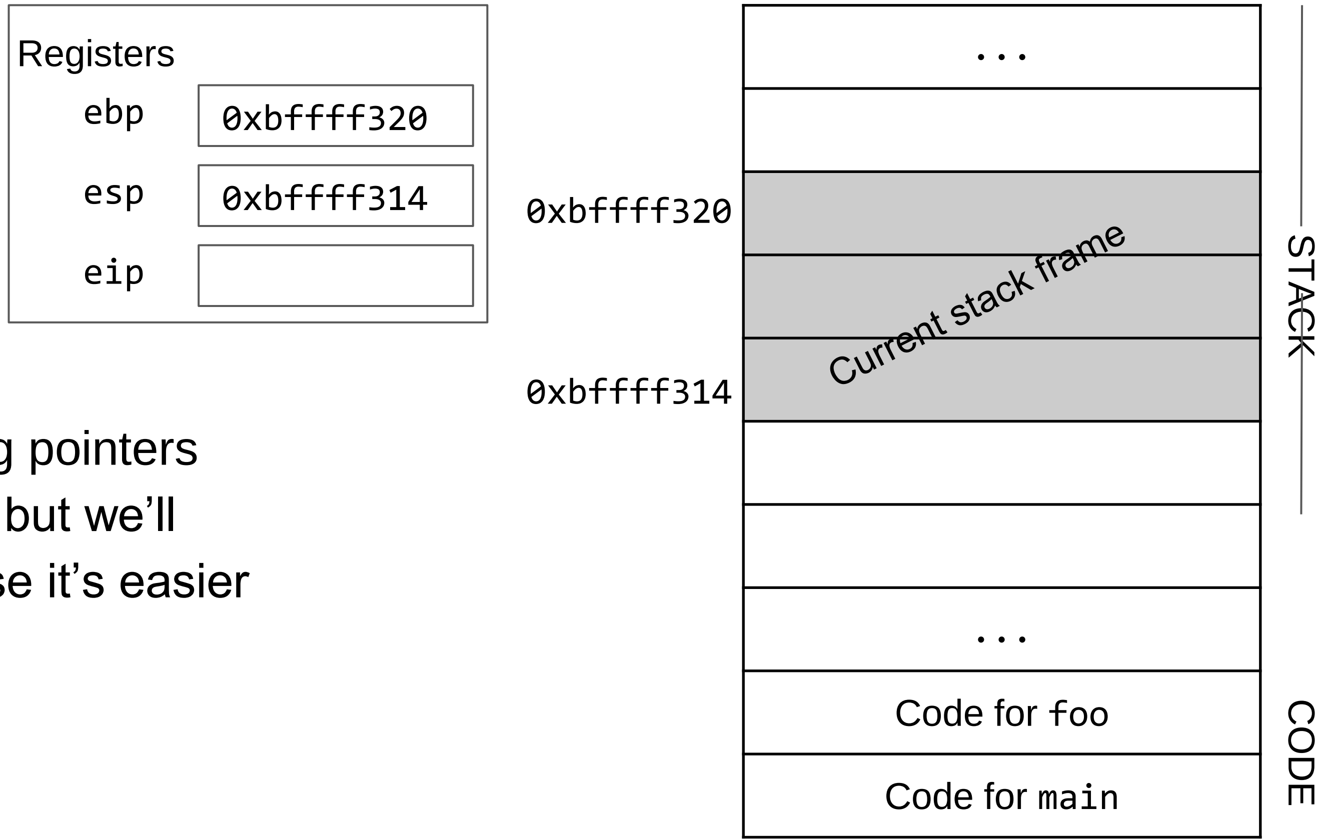


- In this diagram, the `ebp` and `esp` registers are drawn as arrows. What is actually being stored in the register?
- The register is storing the **address** of where the arrow is pointing.
- This works because registers are 32 bits, and addresses are 32 bits.



# Quick detour: storing pointers



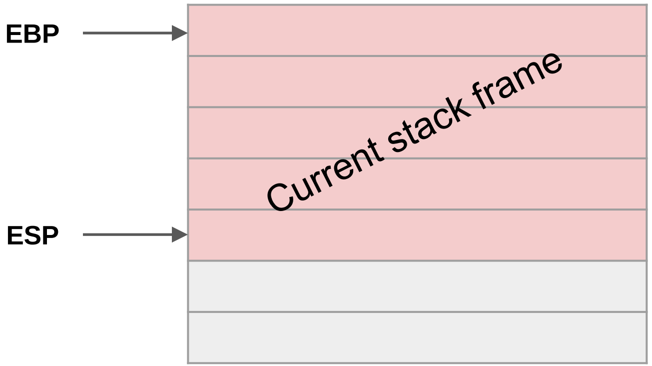



- This is what storing pointers actually looks like, but we'll use arrows because it's easier to look at.

# Pushing and Popping

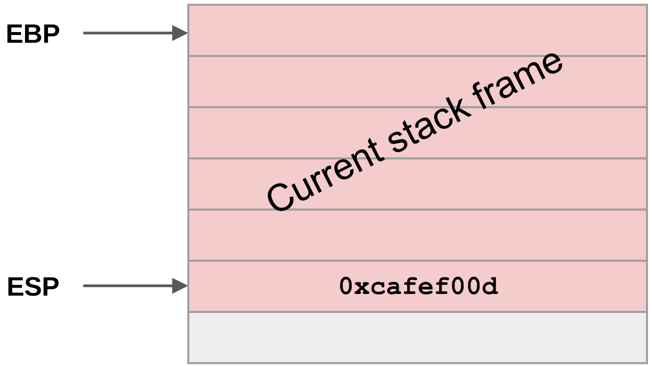


- The **`push`** instruction adds an element to the stack
    - Decrement ESP to allocate more memory on the stack
    - Save the new value on the lowest value of the stack




EAX = `0xcafef00d`
EBX = ...

Before `push %eax`



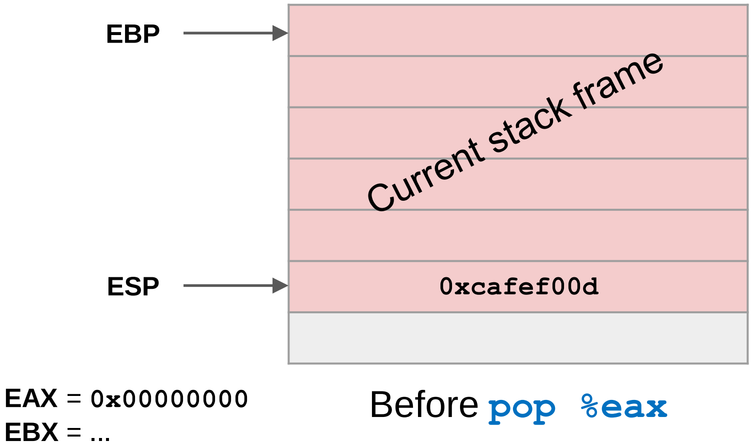

EAX = `0xcafef00d`
EBX = ...

After `push %eax`

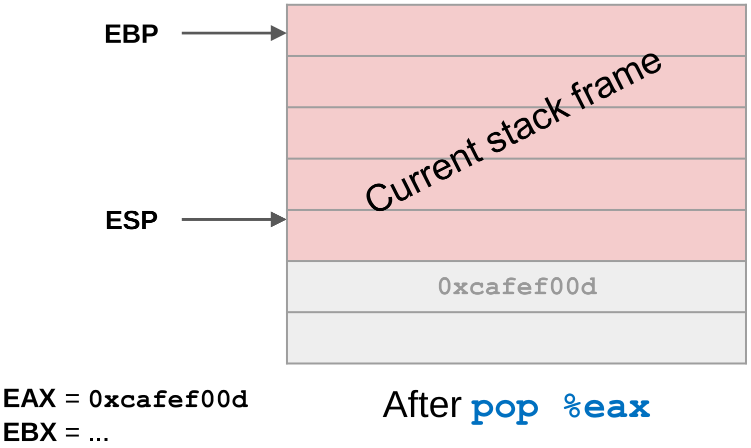
# Pushing and Popping



- The **pop** instruction removes an element from the stack
  - Load the value from the lowest value on the stack and store it in a register
  - Increment ESP to deallocate the memory on the stack




Before `pop %eax`




After `pop %eax`

# x86 Stack Layout



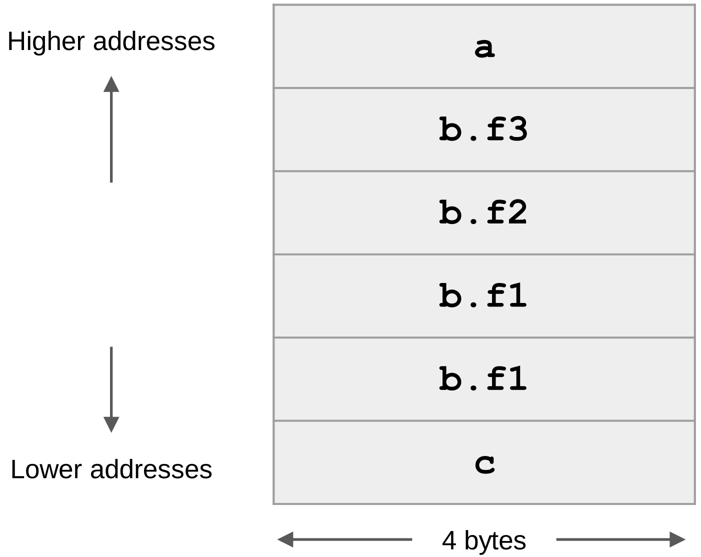
- Local variables are always allocated on the stack
  - Contrast with RISC-V, which has plenty of registers that can be used for variables
- Individual variables within a stack frame are stored with the first variable at the *highest* address
- Members of a struct are stored with the first member at the *lowest* address
- Global variables (not on the stack) are stored with the first variable at the *lowest* address

# Stack Layout



```
struct foo {
    int64_t f1;    // 8 bytes
    int32_t f2;    // 4 bytes
    uint32_t f3;   // 4 bytes
};

void func(void) {
    int a;         // 4 bytes
    struct foo b;
    int c;         // 4 bytes
}
```




How would you fill out the boxes in this stack diagram?
Options:
**a** **b.f1** **b.f2** **b.f3** **c**

# x86 Calling Convention

Textbook Chapter 2.8 & 2.9

# Function Calls



Before function call

```
int main() {
    int a = 1;
    foo();
    return 0;
}
```

Caller

During function call

```
void foo() {
    int b = 0;
    return;
}
```

Callee

After function returns

```
int main() {
    int a = 1;
    foo();
    return 0;
}
```

Caller

The **caller** function (`main`) calls the **callee** function (`foo`).

The callee function executes and then returns control to the caller function.

# x86 Calling Convention



- An understood way for functions to call other functions and know what state the processor will return in
- How to pass arguments
  - Arguments are pushed onto the stack in reverse order, so **`func(val1, val2, val3)`** will place **`val3`** at the highest memory address, then **`val2`**, then **`val1`**
  - Contrast with RISC-V, which passes arguments in argument registers (**`a0-a7`**)
- How to receive return values
  - Return values are passed in EAX
  - Similar to RISC-V, which passes return values in **`a0-a1`**
- Which registers are caller-saved or callee-saved
  - **Callee-saved**: The callee must not change the value of the register when it returns
  - **Caller-saved**: The callee may overwrite the register without saving or restoring it

# x86 Calling Convention

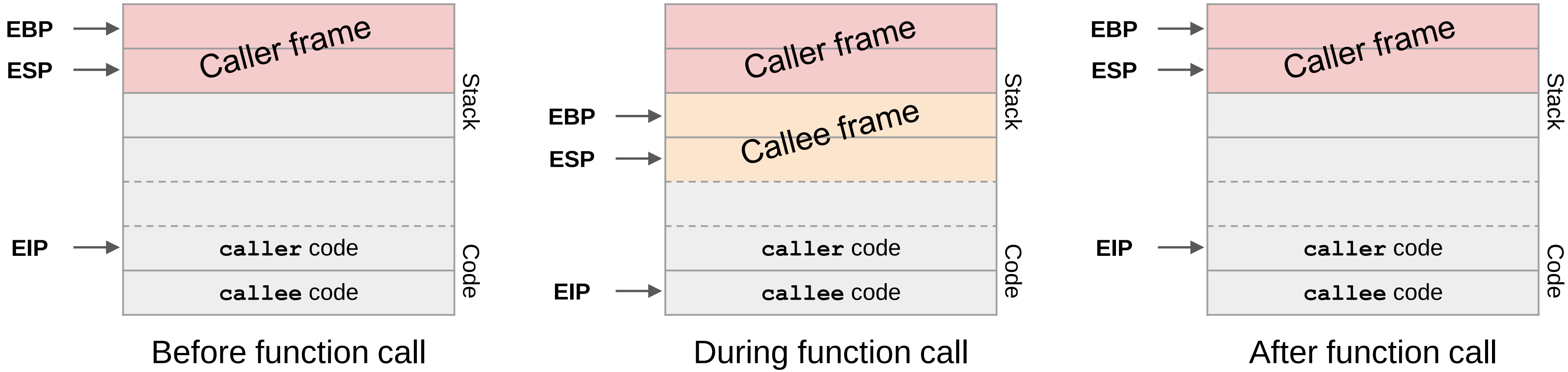


- Which registers are caller-saved or callee-saved
  - **Callee-saved**: The callee must not change the value of the register when it returns
  - **Caller-saved**: The callee may overwrite the register without saving or restoring it
- Caller-Saved Registers: Examples: EAX, ECX, EDX
  - Also known as "volatile" registers.
  - If a function (caller) wants to use these registers, it must save their original values before making a function call and restore them afterward.
  - These registers are not preserved across function calls.
- Callee-Saved Registers: EBP, ESP, EIP
  - Also known as "non-volatile" registers.
  - If a function (callee) uses these registers, it must save their original values upon entry and restore them before exiting the function.
  - These registers are preserved across function calls.
- The distinction between caller-saved and callee-saved registers helps manage the state of registers across function calls, ensuring that a function doesn't unintentionally overwrite values in registers that the calling code expects to remain unchanged.

# Calling a Function in x86



- When calling a function, the ESP and EBP need to shift to create a new stack frame, and the EIP must move to the callee's code
- When returning from a function, the ESP, EBP, and EIP must return to their old values （Callee-Saved）



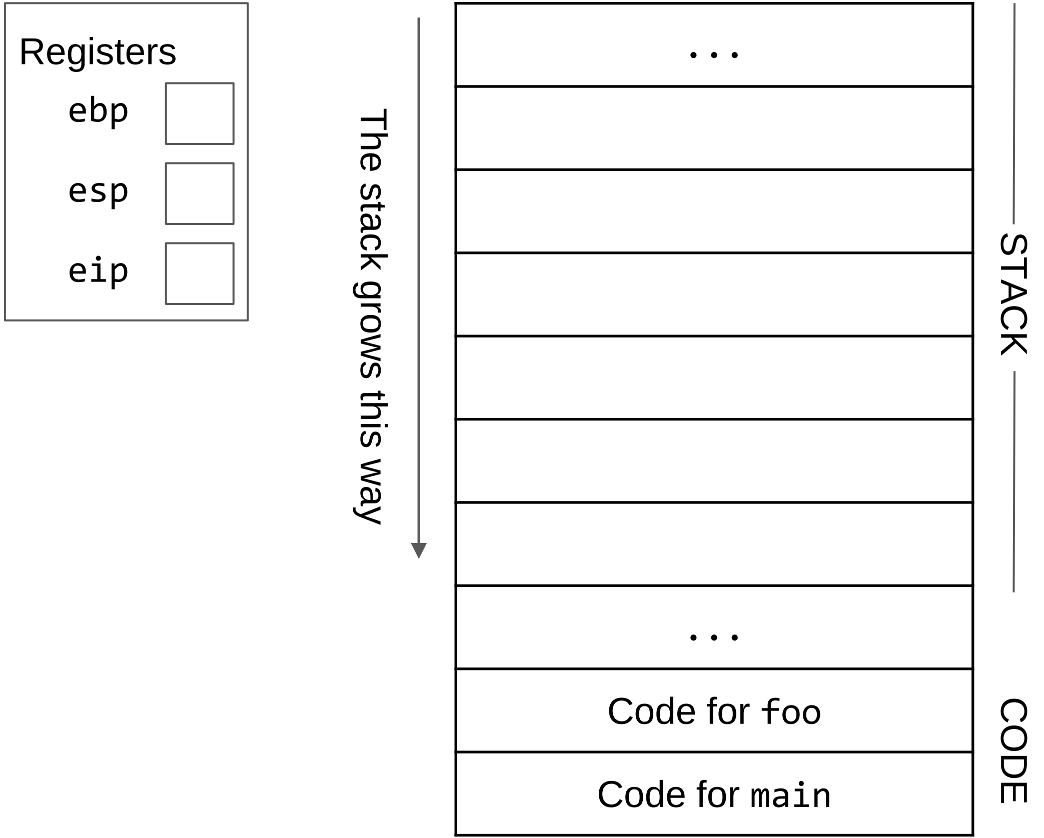
# x86 Calling Convention Design

Textbook Chapter 2.6

# Review: stack, registers



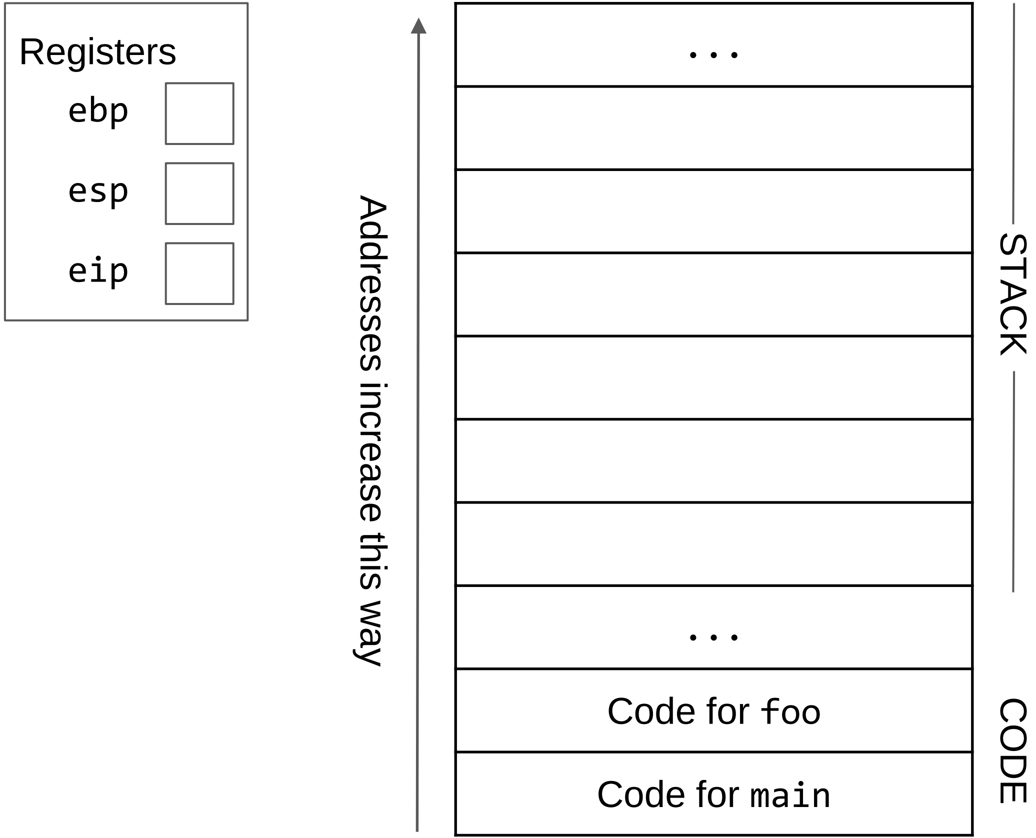
- Any time your code calls a function, space is made on the stack for local variables. The space goes away once the function returns.
- The stack starts at higher addresses and grows down.
- Registers are 32-bit (or 4-byte, or 1-word) units of memory located on CPU.



# Review: words, code section



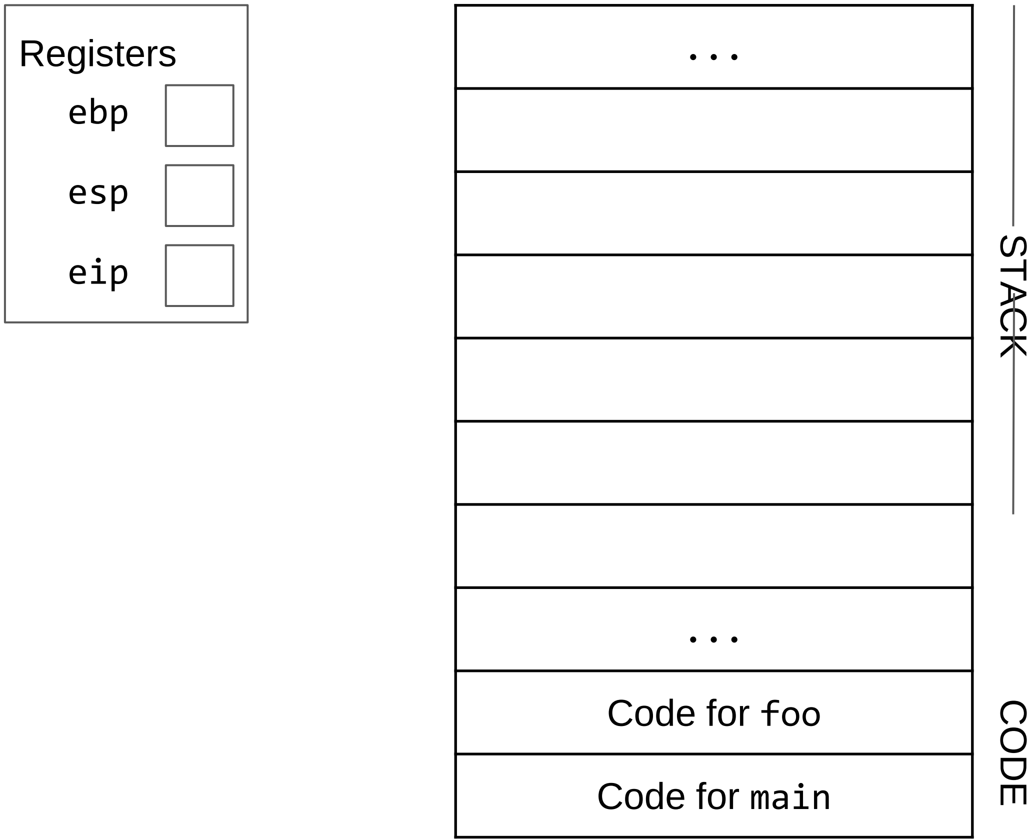
- The code section contains raw bytes that represent assembly instructions.
- We omit the static and heap sections to save space.
- Each row of the diagram is 1 word = 4 bytes = 32 bits.
- Addresses increase as you move up the diagram.



# Stack frames



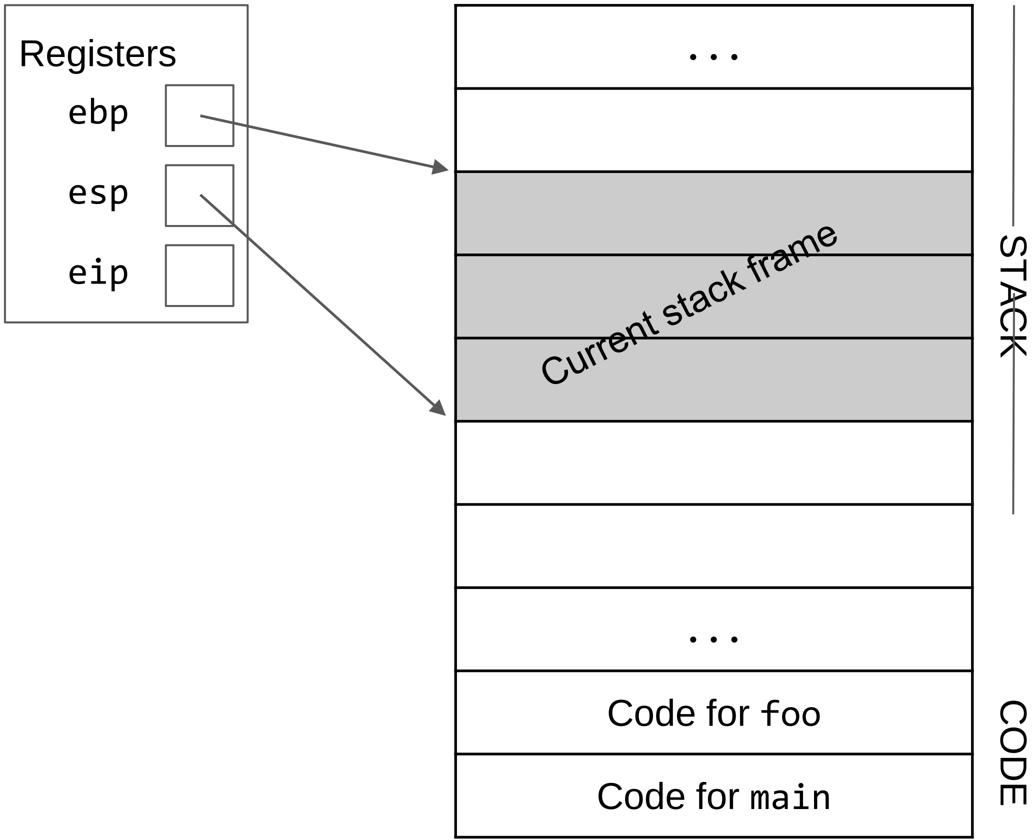
- We'll use two pointers to tell us which part of the stack is being used by the current function.
- On the stack, this is called a **stack frame**. One stack frame corresponds to one function being called.
- You might recall stack frames from environment diagrams in CS 61A.



# ebp and esp



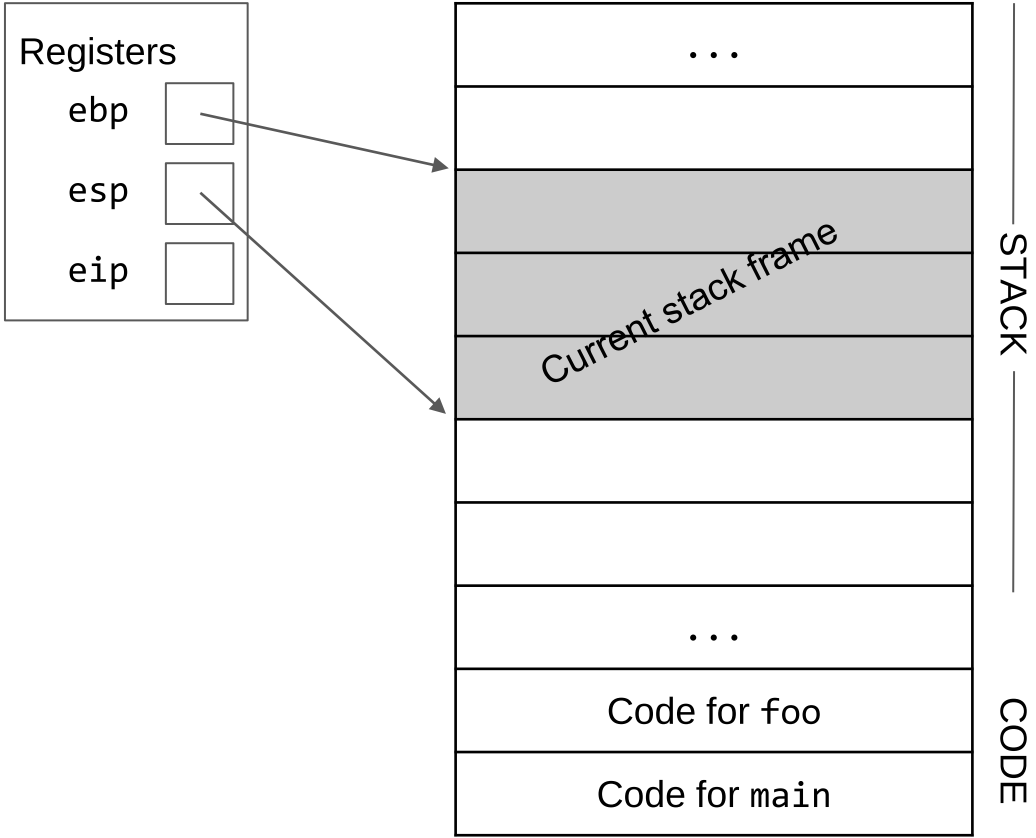
- We store two pointers to remind us the extent of the current stack frame.
- `ebp` is used for the top of the stack frame, and `esp` is used for the bottom of the stack frame.



# esp

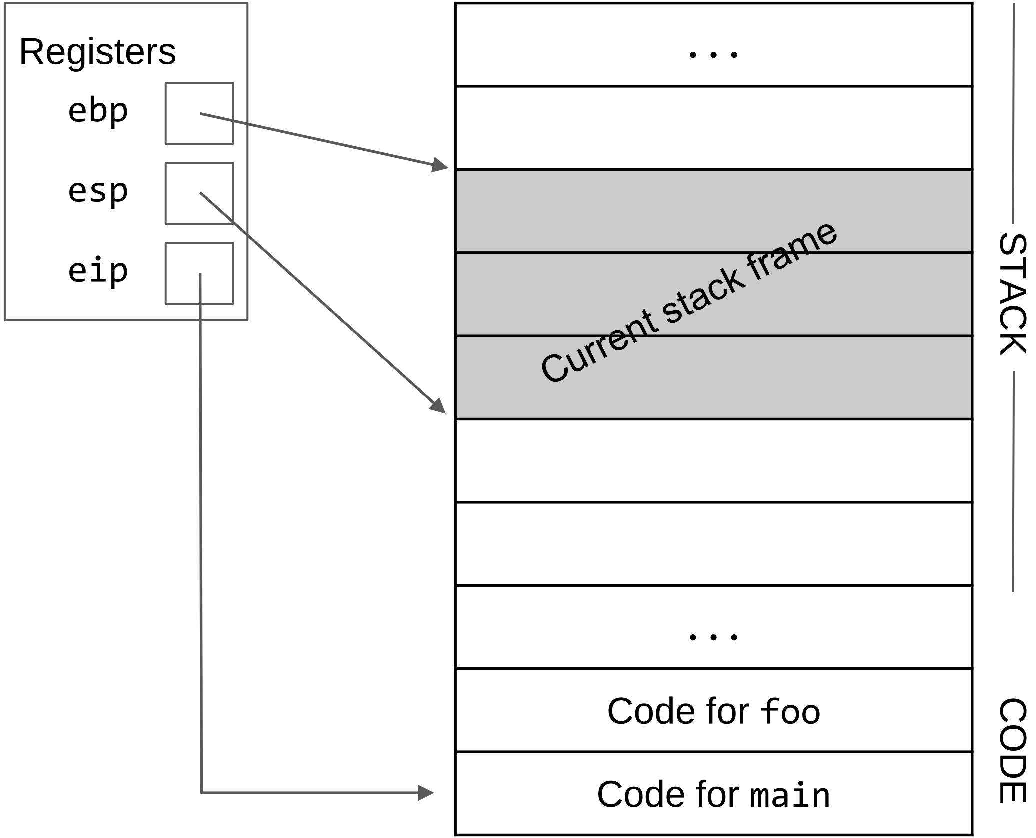


- esp also denotes the current lowest value on the stack.
- Everything below `esp` is undefined
- If you ever **push** a value onto the stack, `esp` must adjust to match the lowest value on the stack.



# eip

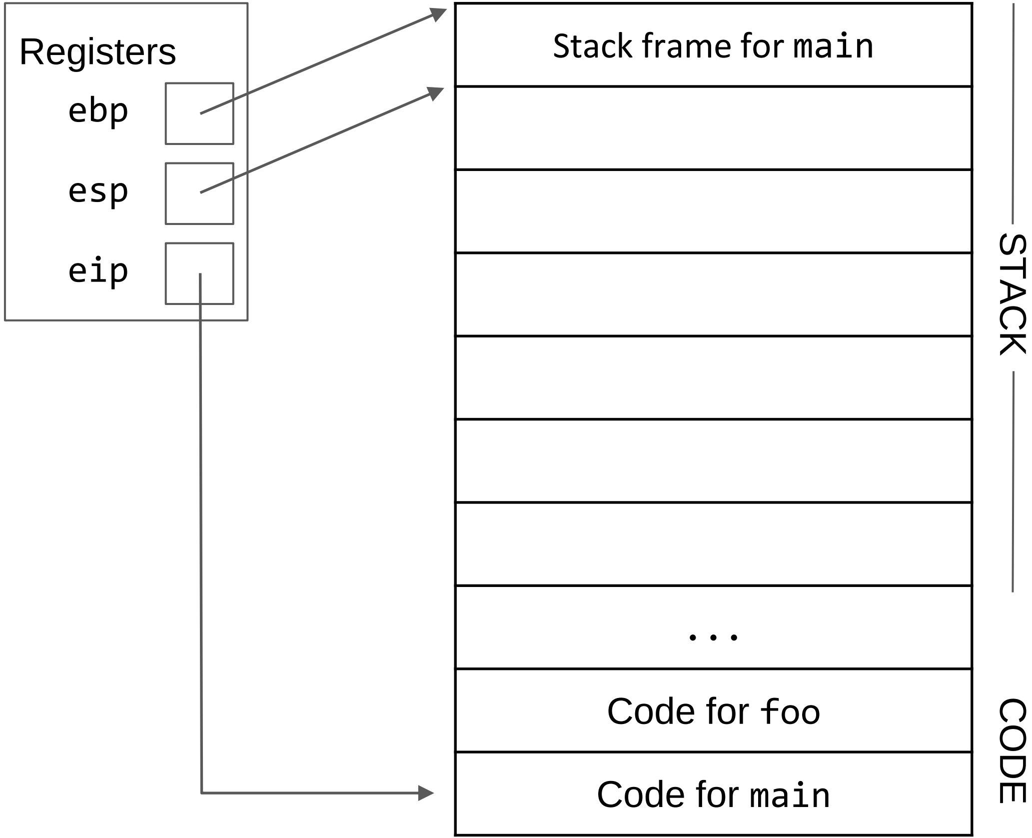
- We need some way to keep track of what step we're at in the instructions.
- We use the `eip` register to store a pointer to the next instruction to be executed.



# Designing the stack: requirements



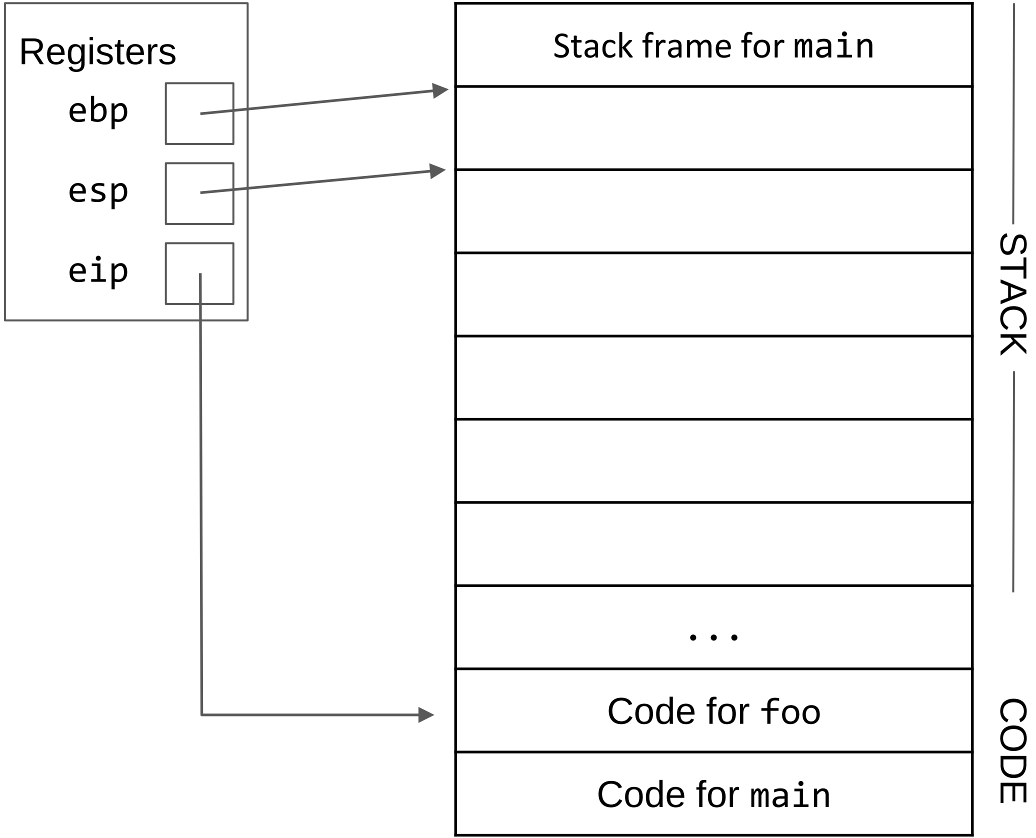
- Every time a function is called, a new stack frame must be created. When the function returns, the stack frame must be discarded.
- Each stack frame needs to have space for local variables.
- We also need to figure out how to pass arguments to functions using the stack.



# Designing the stack: requirements



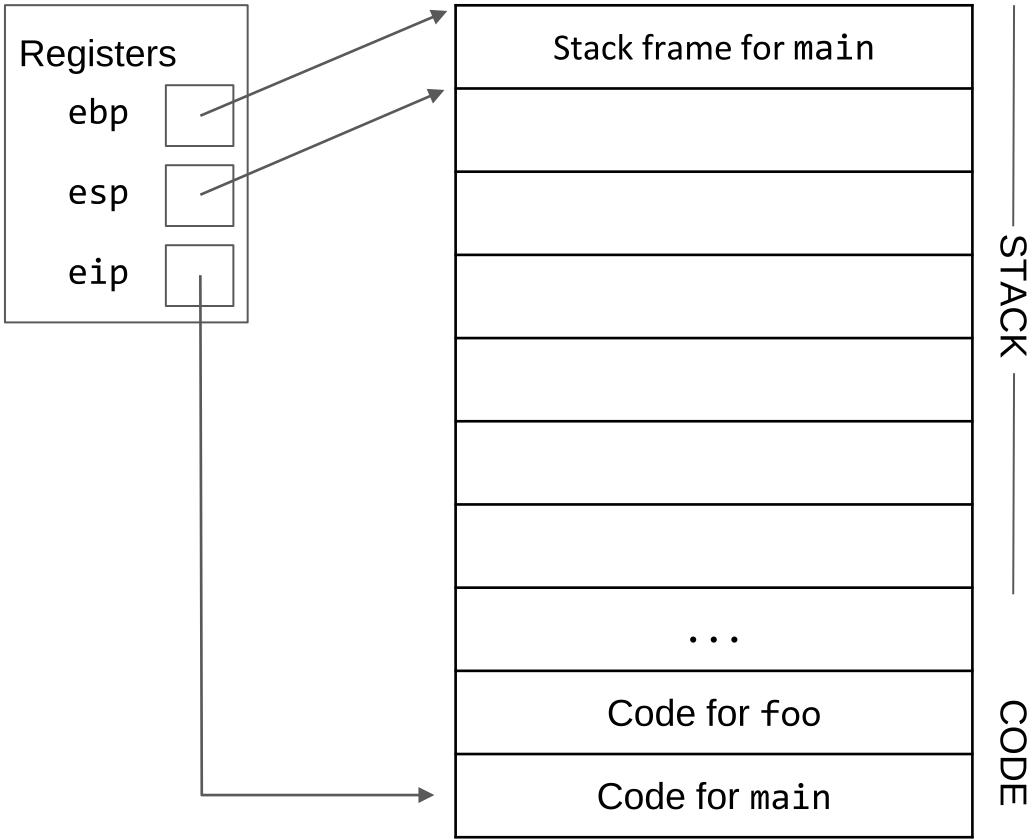
- For example, this is what the stack might look like after a function `foo` is called.
- The `ebp` and `esp` registers should adjust to give us a stack frame for `foo` with the correct size.
- The `eip` register should adjust to let us execute the instructions for `foo`.



# Designing the stack: requirements



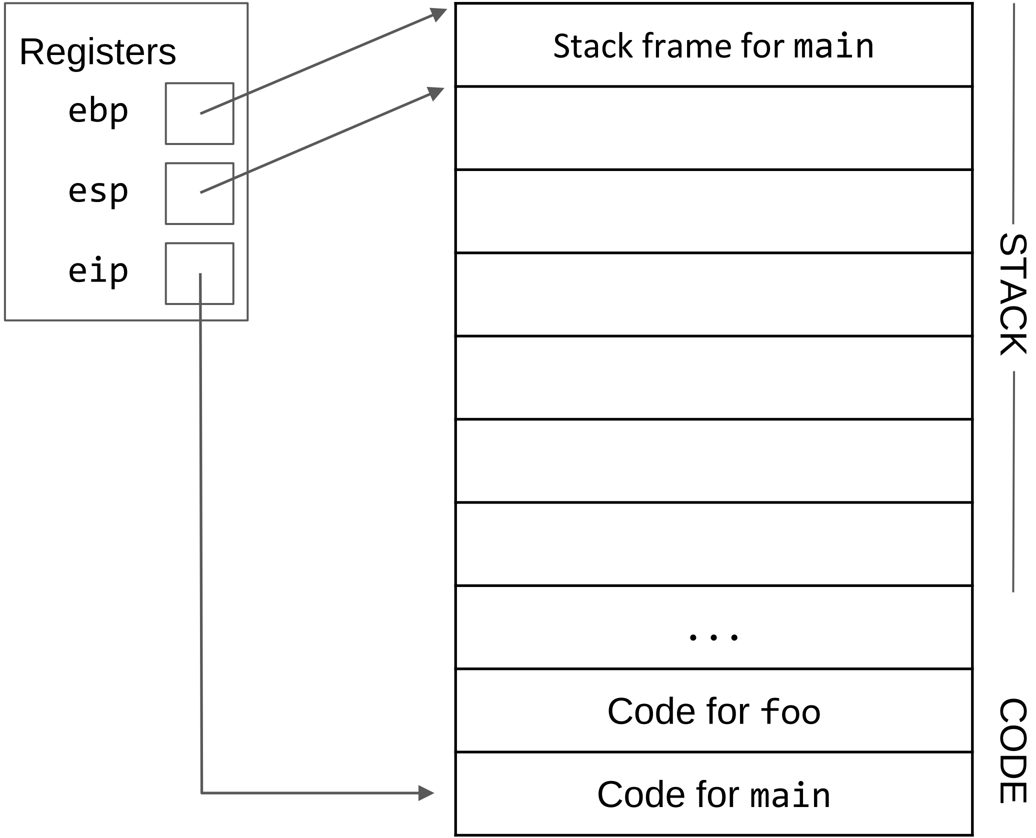
- Then after `foo` returns, the stack should look exactly like it did before `foo` was called.



# Remember to save your work as you go



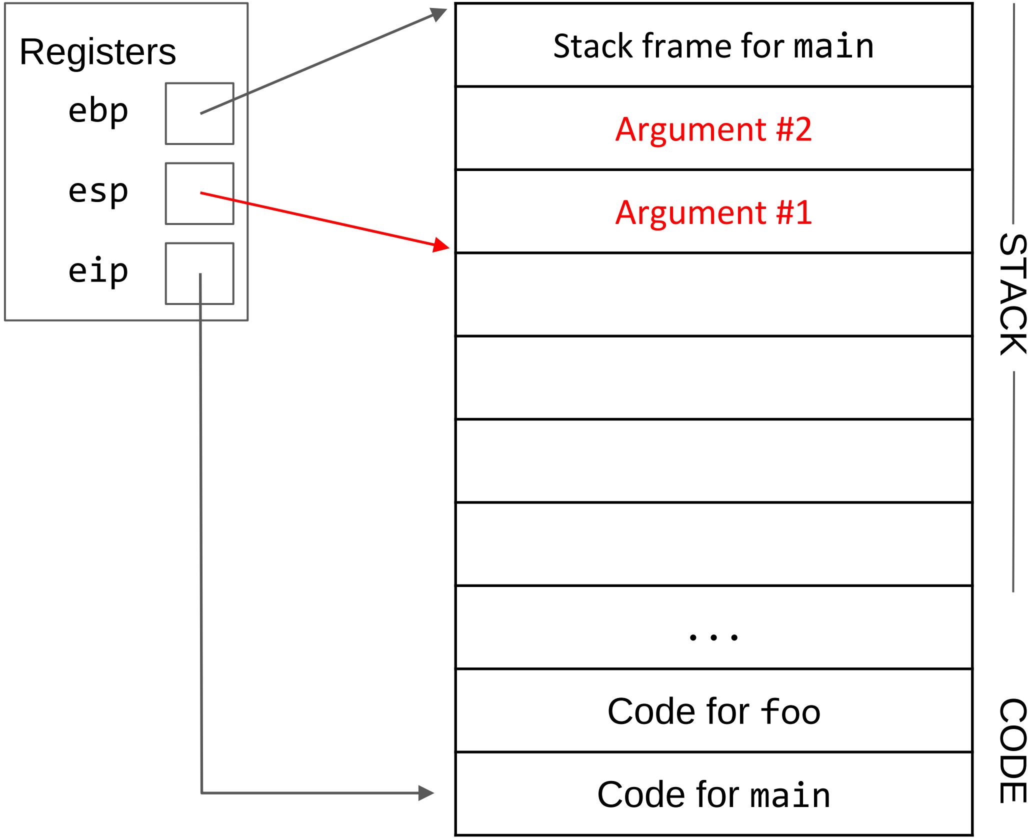
- Don't forget calling convention: if we ever overwrite a saved register, we should remember its old value by putting it on the stack.



# 1. Arguments



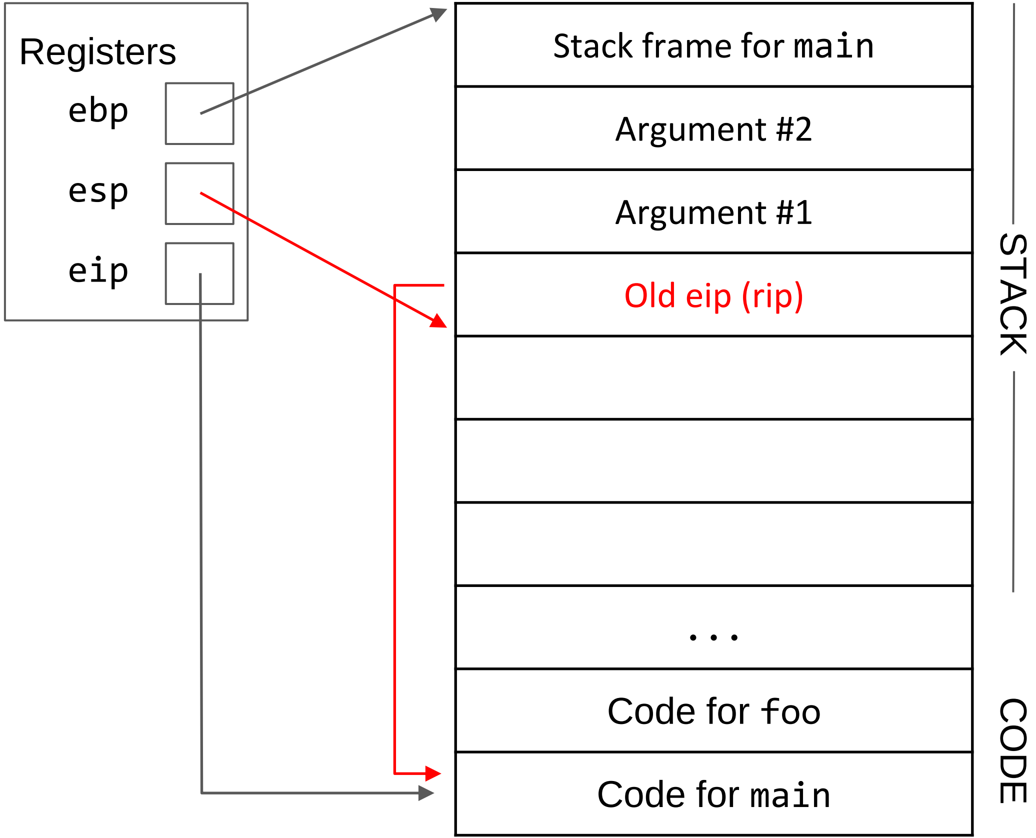
- First, we push the arguments onto the stack.
- Remember to adjust `esp` to point to the new lowest value on the stack.
- Arguments are added to the stack in reverse order.



# 2. Remember eip



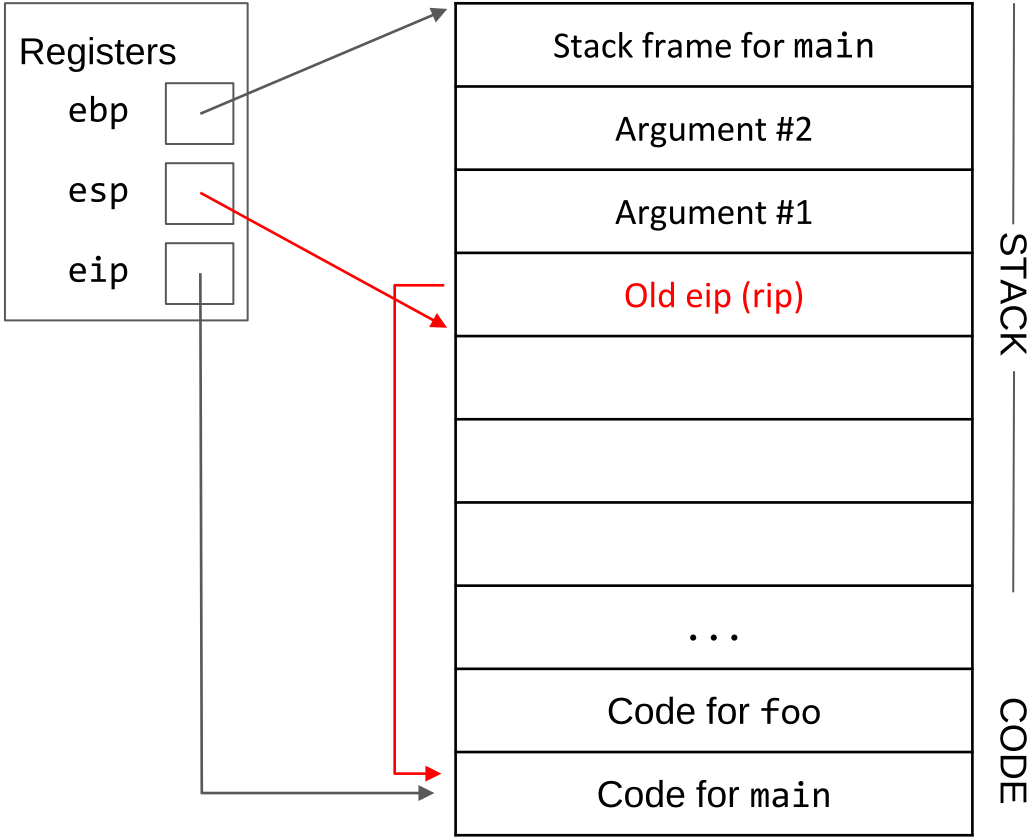
- Next, push the current value of `eip` on the stack.
  - This tells us what code to execute next after the function returns
  - Similar to putting a return address in `ra` in RISC-V
- Remember to adjust `esp` to point to the new lowest value on the stack.



# 2. Remember eip



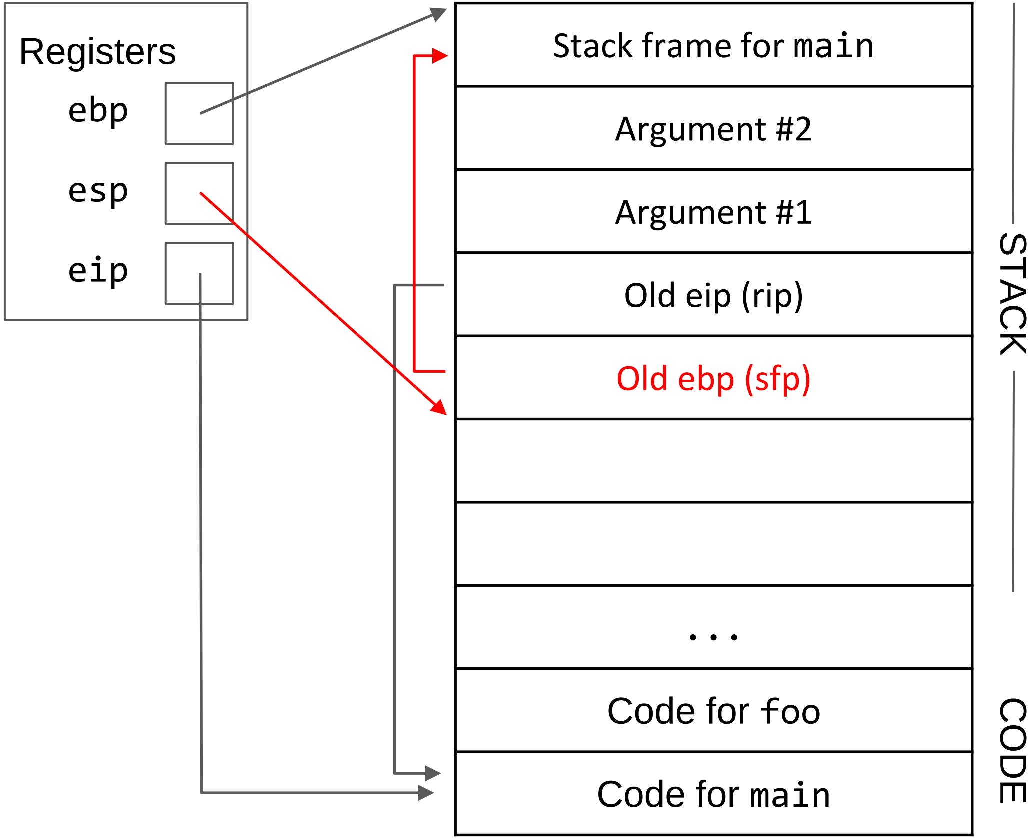
- This value is sometimes known as the `rip` (return instruction pointer), because when we're finished with the function, this pointer tells us where in the instructions to go next.



# 3. Remember ebp



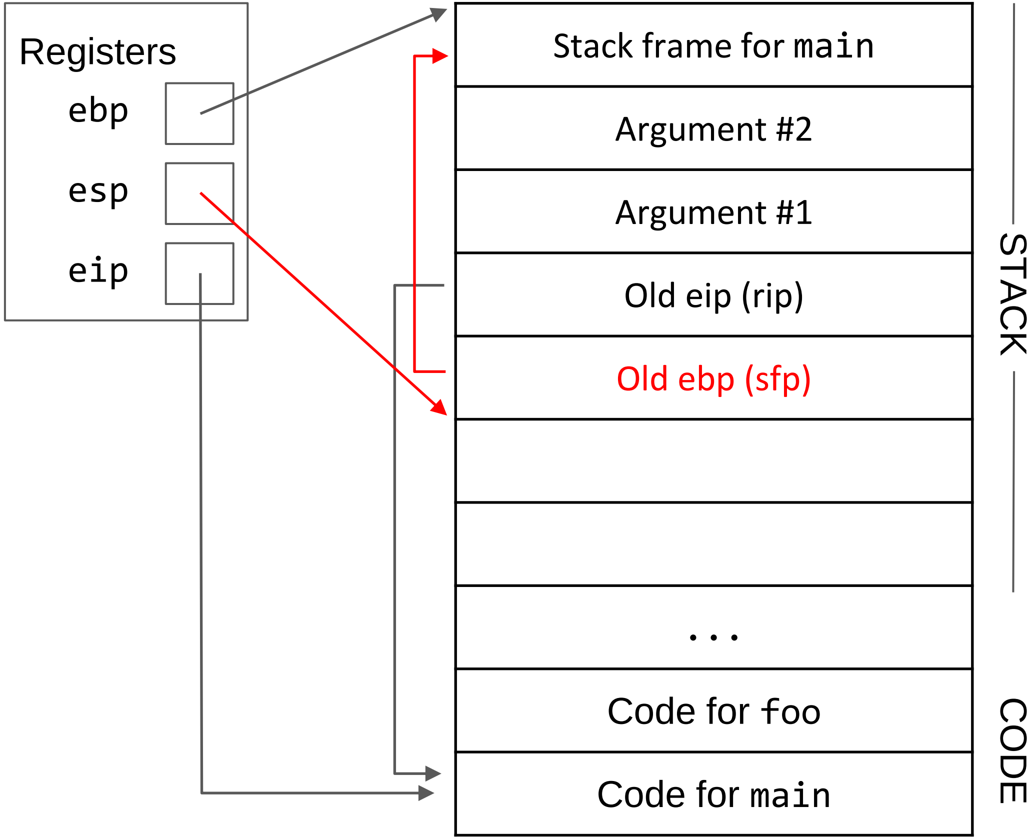
- Next, push the current value of `ebp` on the stack.
  - This will let us restore the top of the previous stack frame when we return
  - Alternate interpretation: `ebp` is a saved register. We store its old value on the stack before overwriting it.
- Remember to adjust `esp` to point to the new lowest value on the stack.



# 3. Remember ebp



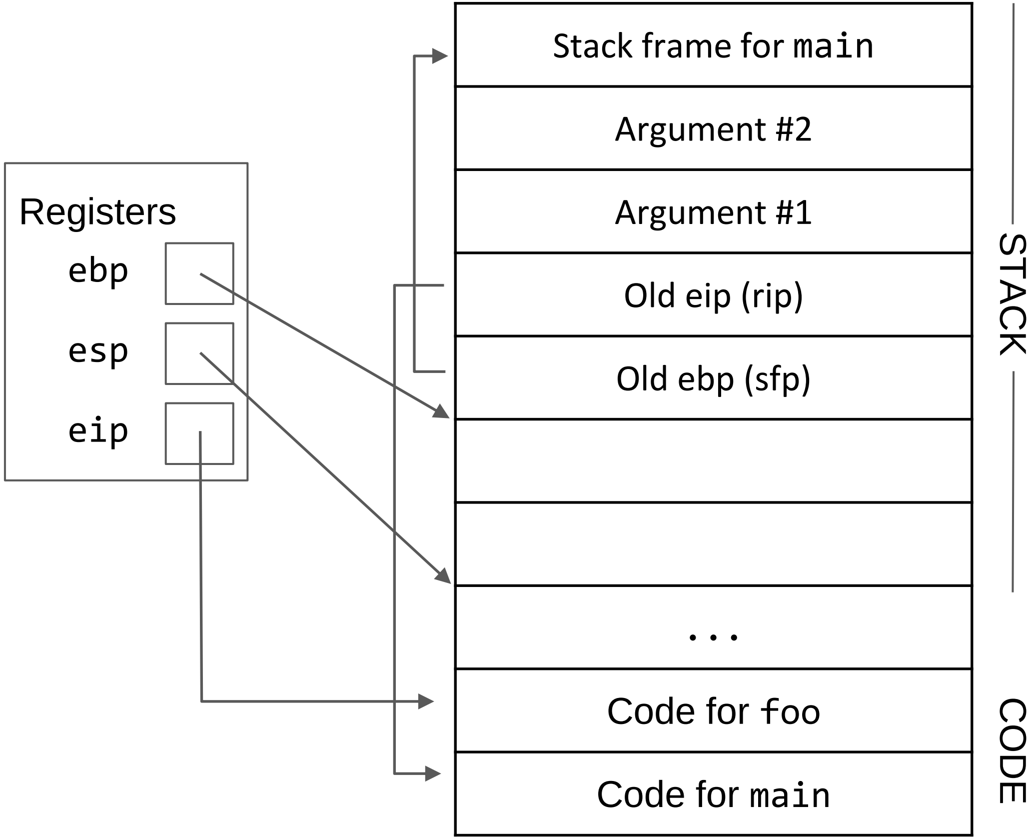
- This value is sometimes known as the `sfp` (saved frame pointer), because it reminds us where the previous frame was.



# 4. Adjust the stack frame



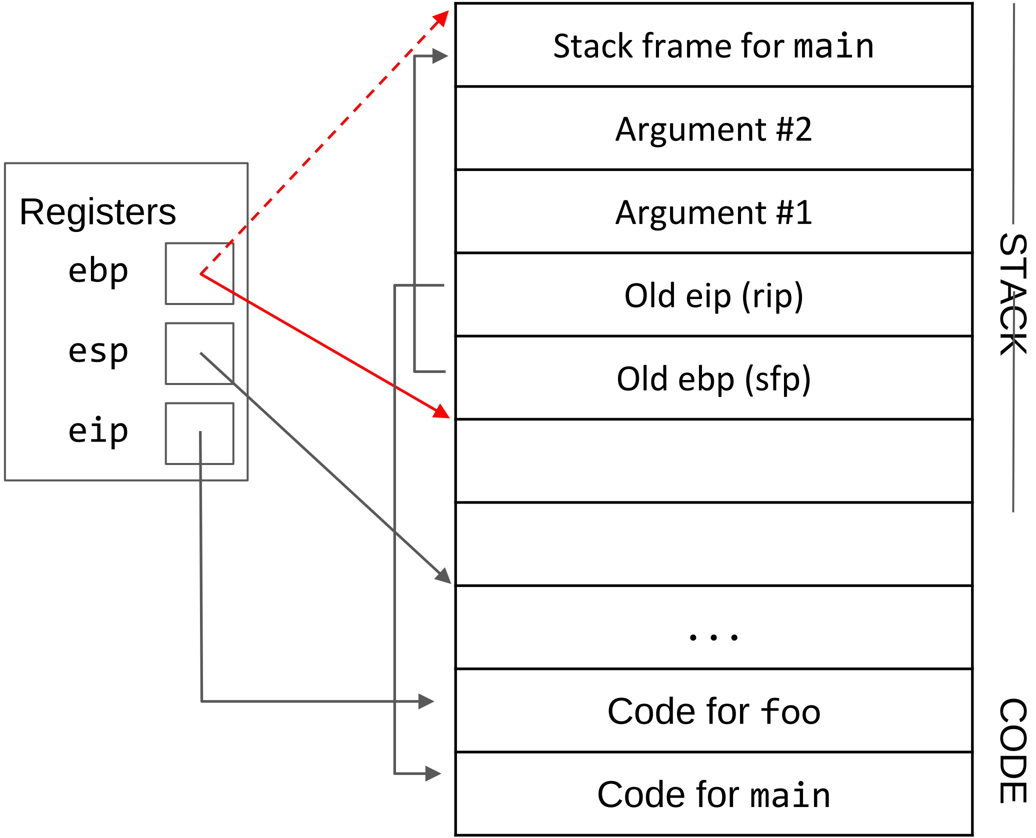
- To adjust the stack frame, we need to update all three registers.
- We can safely do this because we've just saved the old values of `ebp` and `eip`. (`esp` will always be the bottom of the stack, so there's no need to save it).



# 4. Adjust the stack frame



- ebp now points to the top of the current stack frame, which is always the `sfp`. (Easy way to remember this: ebp points to old value of ebp.)

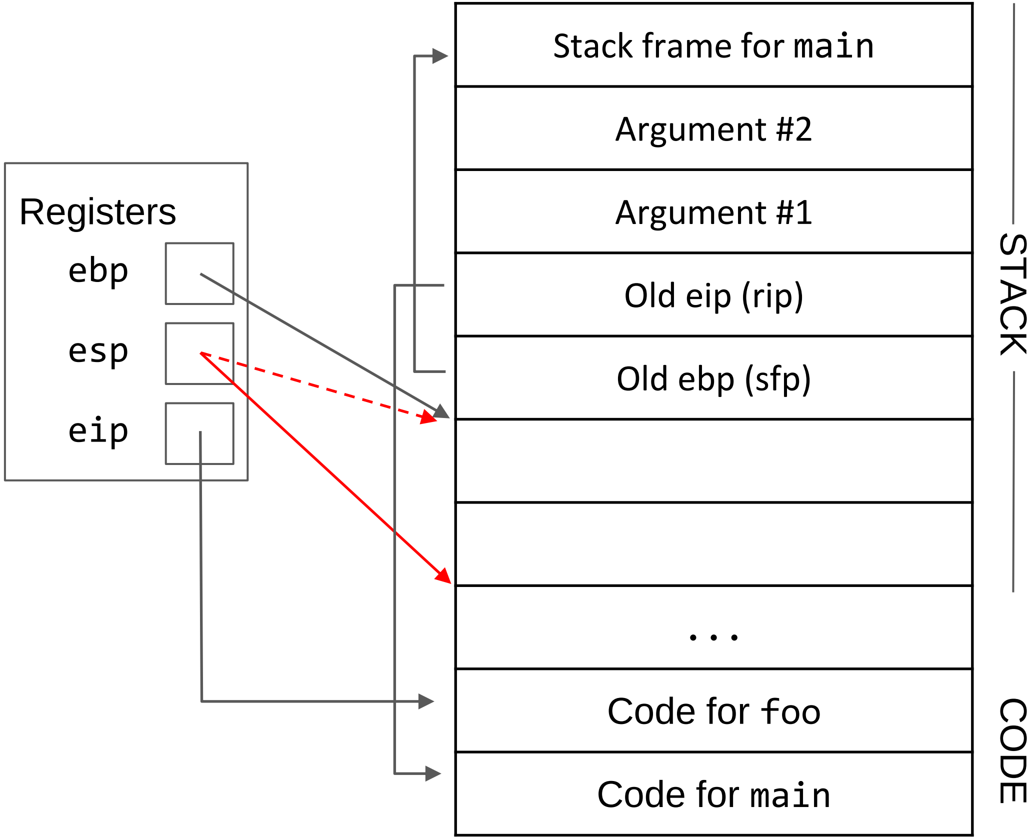



dashed line = ebp pointer before this step

# 4. Adjust the stack frame



- `esp` now points to the bottom of the current stack frame. The compiler determines the size of the stack frame by checking how much space the function needs (how many local variables it has).

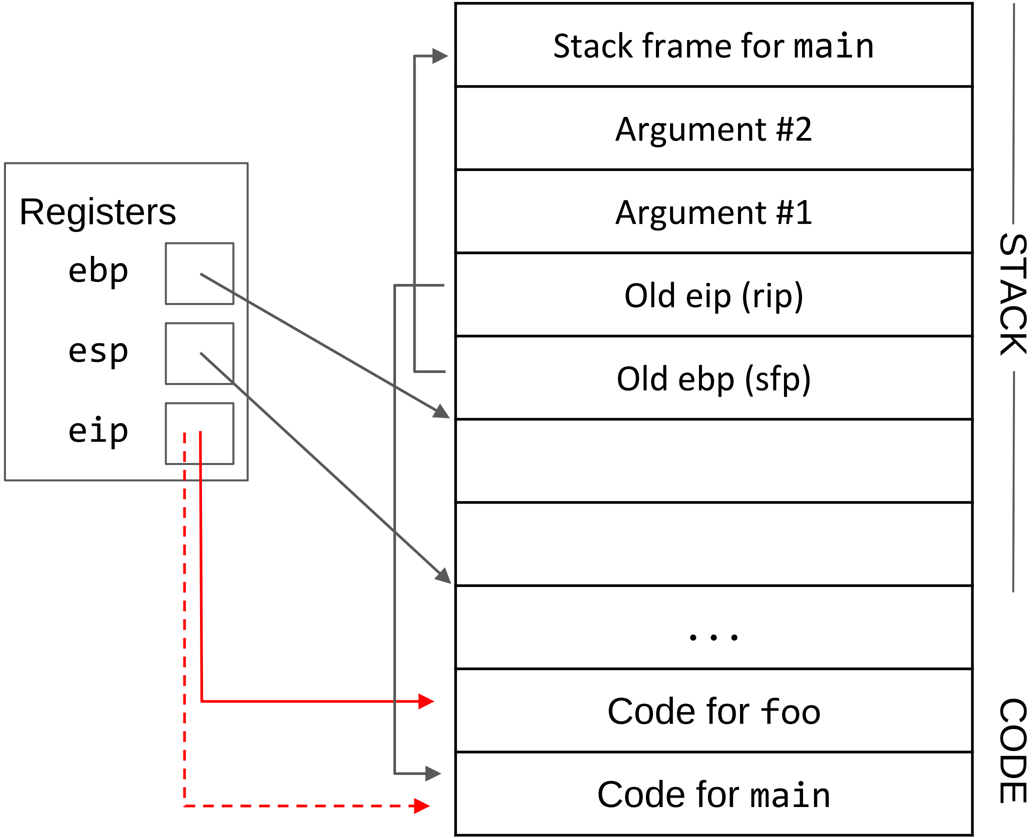



dashed line = `esp` pointer before this step

# 4. Adjust the stack frame



- `eip` now points to the instructions for `foo`.

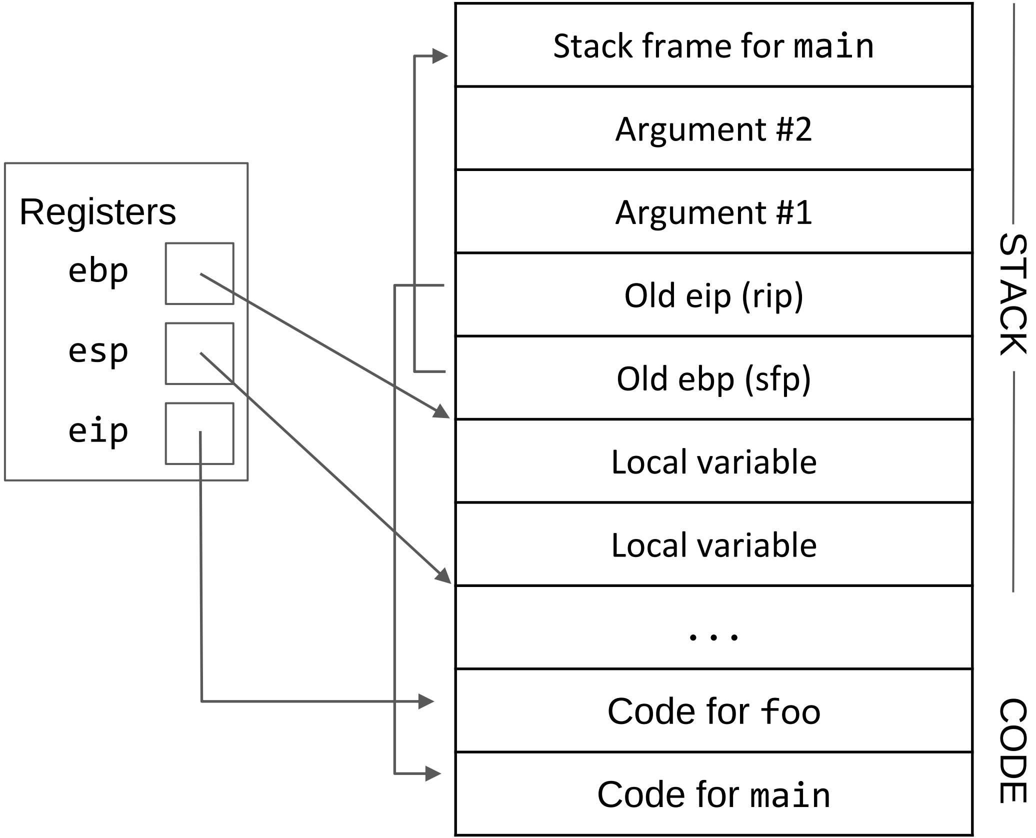



dashed line = `eip` pointer before this step

# 5. Execute the function



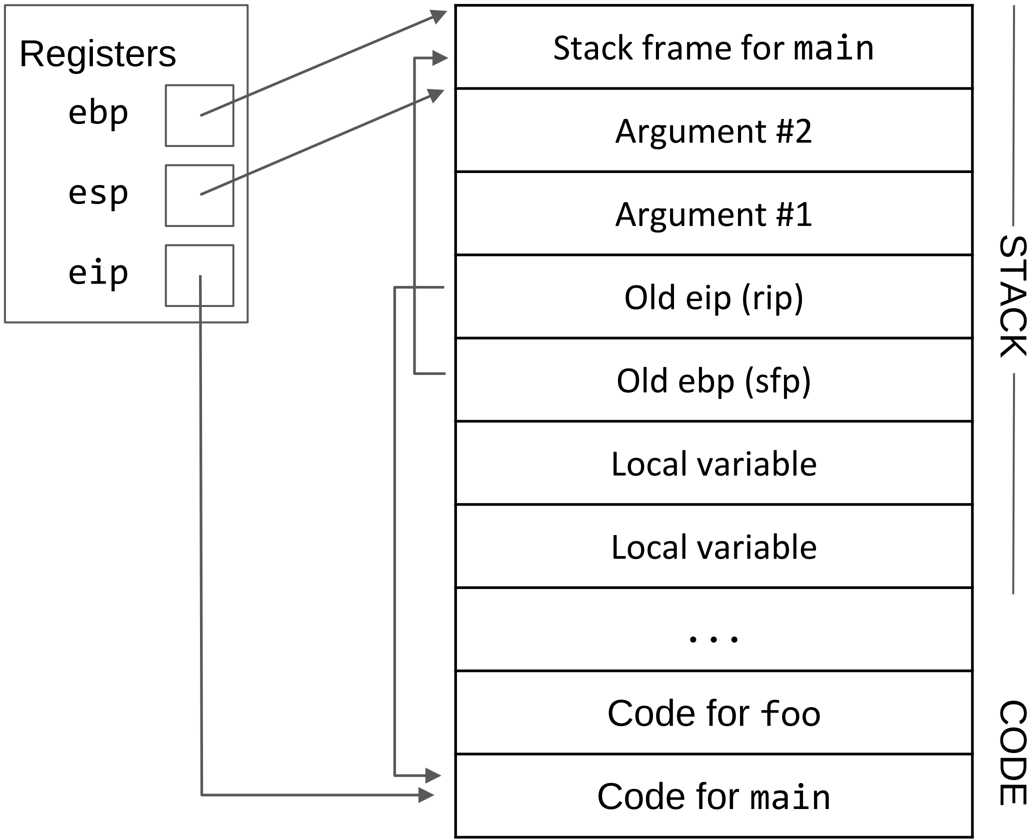
- Now the stack frame is ready to do whatever the function instructions say to do.
- Any local variables can be moved onto the stack now.



# 6. Restore everything



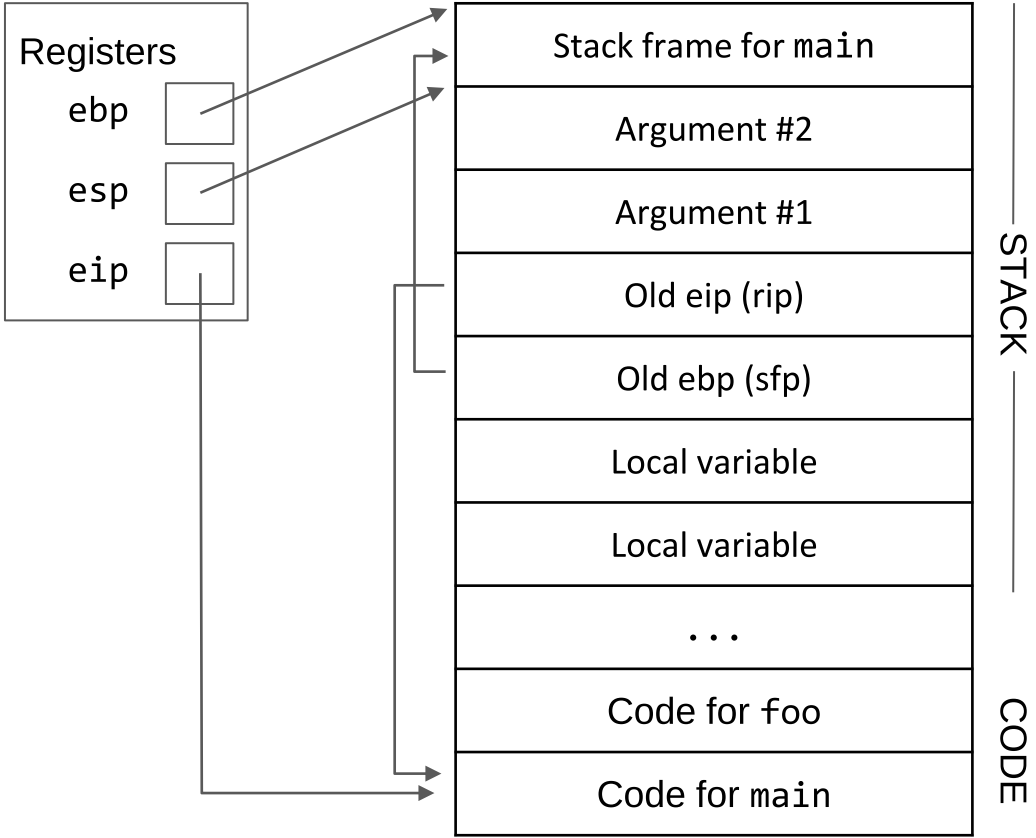
- After the function is finished, we put all three registers back where they were.
- We use the addresses stored in `rip` and `sfp` to restore `eip` and `ebp` to their old values.



# 6. Restore everything



- `esp` naturally moves back to its old place as we undo all our work, which involves **popping** values off the stack.
- Note that the values we pushed on the stack are still there (we don't overwrite them to save time), but they are below `esp` so they cannot be accessed by memory.



# Review: steps of a function call



1. Push arguments on the stack
2. Push old eip (rip) on the stack
3. Push old ebp (sfp) on the stack
4. Adjust the stack frame
5. Execute the function
6. Restore everything

# Steps of a function call (complete)



1. Push arguments on the stack
2. Push old eip (rip) on the stack
3. Move eip
4. Push old ebp (sfp) on the stack
5. Move ebp
6. Move esp
7. Execute the function
8. Move esp
9. Restore old ebp (sfp)
10. Restore old eip (rip)
11. Remove arguments from stack

# Steps of an x86 Function Call



caller

1. Push arguments on the stack
2. Push old EIP (RIP) on the stack
3. Move EIP

callee

4. Push old EBP (SFP) on the stack
5. Move EBP
6. Move ESP
7. Execute the function
8. Move ESP
9. Pop (restore) old EBP (SFP)
10. Pop (restore) old EIP (RIP)

caller

11. Remove arguments from stack

# Steps of a function call (complete) Example



1. Push arguments on the stack
2. Push old eip (rip) on the stack
3. Move eip
4. Push old ebp (sfp) on the stack
5. Move ebp
6. Move esp
7. Execute the function
8. Move esp
9. Restore old ebp (sfp)
10. Restore old eip (rip)
11. Remove arguments from stack

main

foo

main

Moving eip transfers control from main to foo.

Restoring eip transfers control back to main.

# x86 Calling Convention Walkthrough

Textbook Chapter 2.6

# x86 Function Call

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```



Here is a snippet of C code

Here is the code compiled into x86 assembly

```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

# x86 Function Call

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```



The instruction that was just executed is in **red**

The EIP points to the address of the *next* instruction!

```
caller:
    ...
EIP → push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

# x86 Function Call

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```



Here is a diagram of the stack. Remember, each row represents 4 bytes (32 bits).

```
caller:
    ...
EIP → push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

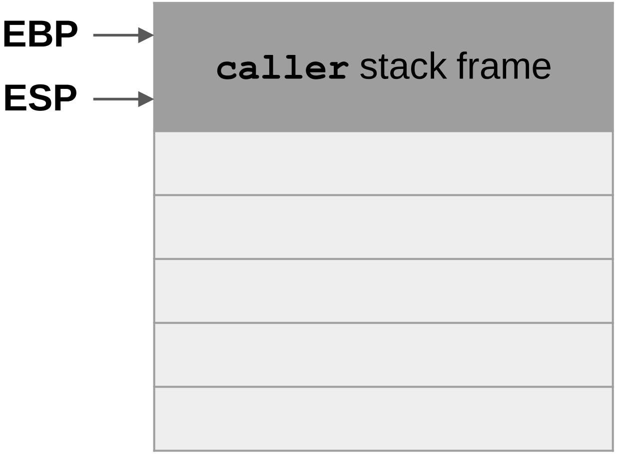
# x86 Function Call

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```



- The EBP and ESP registers point to the top and bottom of the current stack frame.




EIP →

```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

# x86 Function Call

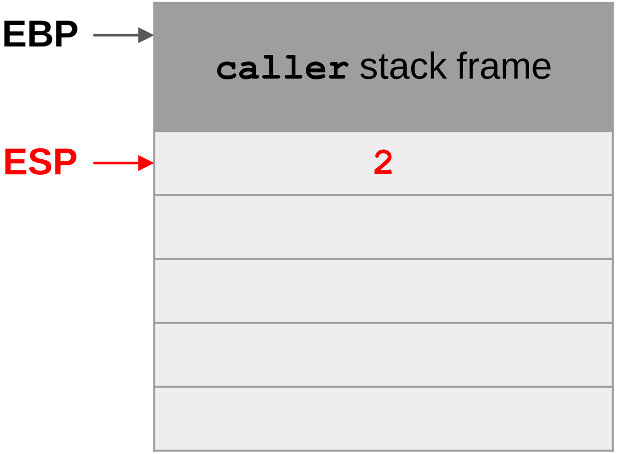
```c
void caller(void) {
    callee(1, 2);
}
```

```c
int callee(int a, int b) {
    int local;
    return 42;
}
```



**1. Push arguments on the stack**

- The **`push`** instruction decrements the ESP to make space on the stack
- Arguments are pushed in reverse order




```
caller:
        ...
        push $2
EIP     push $1
        call callee
        add $8, %esp
        ...

callee:
        push %ebp
        mov %esp, %ebp
        sub $4, %esp

        mov $42, %eax

        mov %ebp, %esp
        pop %ebp
        ret
```

# x86 Function Call

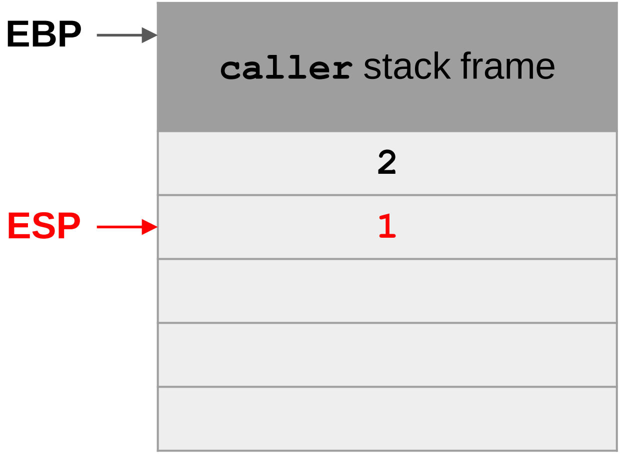
```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```



**1. Push arguments on the stack**

- The **`push`** instruction decrements the ESP to make space on the stack
- Arguments are pushed in reverse order




```
caller:
    ...
    push $2
    push $1
EIP → call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

# x86 Function Call

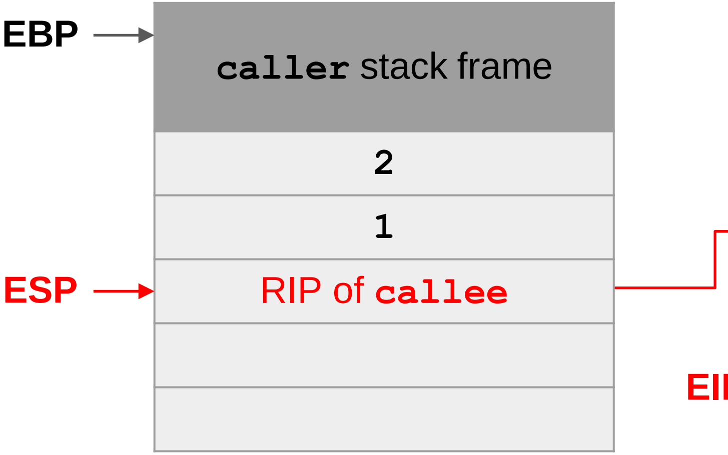
```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```



**2. Push old EIP (RIP) on the stack**
**3. Move EIP**

- The `call` instruction does 2 things
- First, it pushes the current value of EIP (the address of the next instruction in `caller`) on the stack.
- The saved EIP value on the stack is called the RIP (return instruction pointer).
- Second, it changes EIP to point to the instructions of the callee.




By loading the memory address of the callee's code into the EIP register

```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
EIP → push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

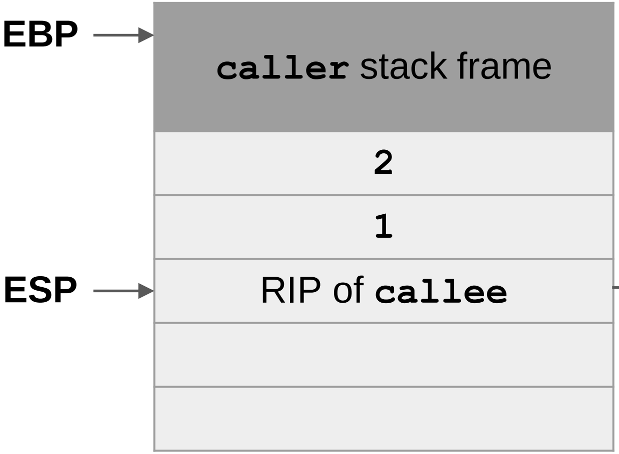
# x86 Function Call

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```



- The next 3 steps set up a stack frame for the callee function.
- These instructions are sometimes called the function prologue, because they appear at the start of every function.




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:    Function prologue
EIP →
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

# x86 Function Call

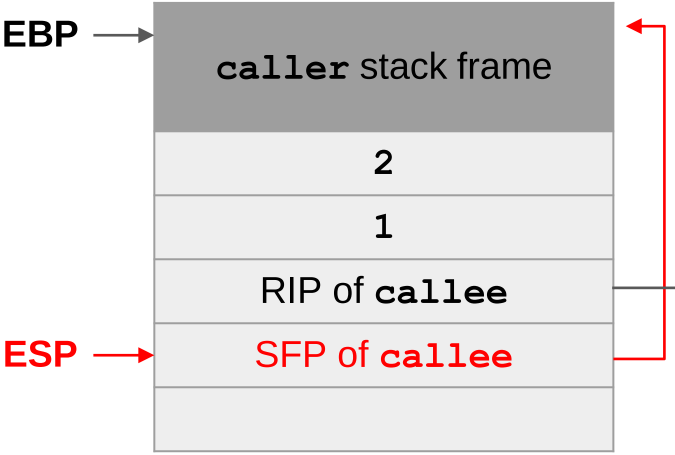
```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```



**4. Push old EBP (SFP) on the stack**

- We need to restore the value of the EBP when returning, so we push the current value of the EBP on the stack.
- The saved value of the EBP on the stack is called the SFP (saved frame pointer).




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

EIP

# x86 Function Call

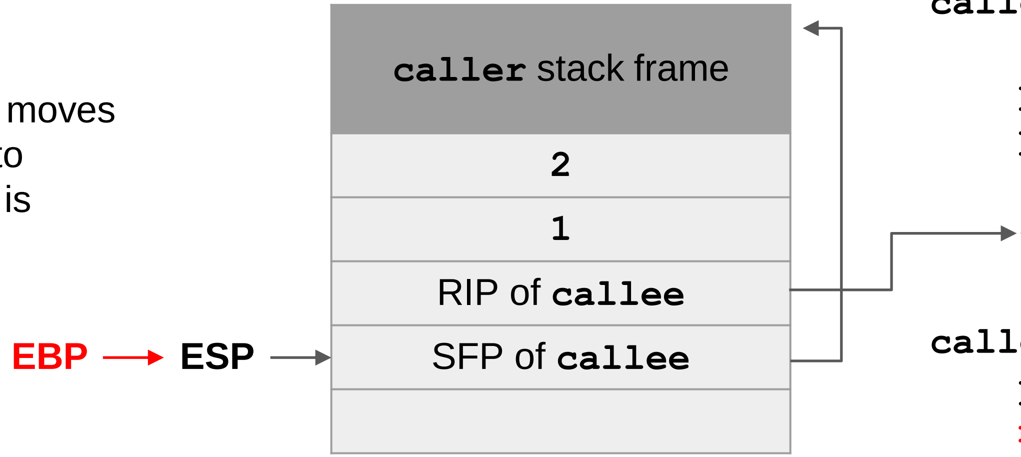
```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```



### 5. Move EBP

- This instruction moves the EBP down to where the ESP is located.




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

# x86 Function Call

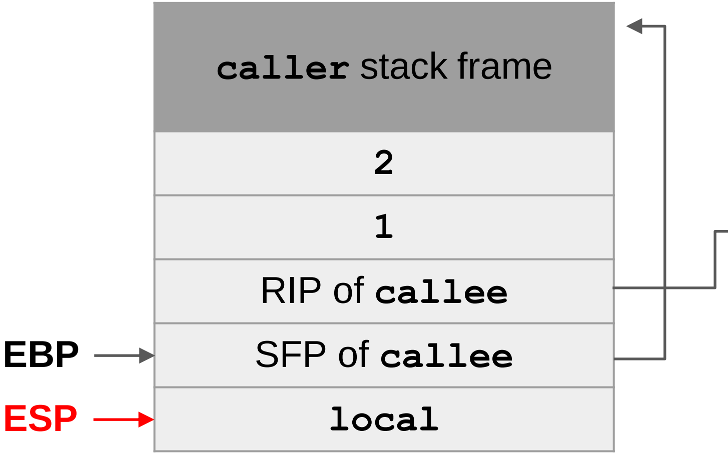
```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```



## 6. Move ESP

- This instruction moves **esp** down to create space for a new stack frame.




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp
EIP → mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

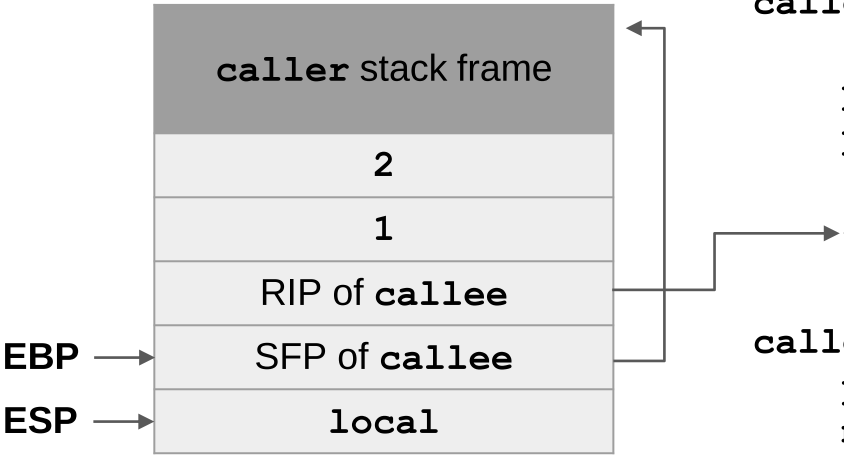
# x86 Function Call

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```

**7. Execute the function**

- Now that the stack frame is set up, the function can begin executing.
- This function just returns 42, so we put 42 in the EAX register. (Recall the return value is placed in EAX.)




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

EIP → mov %ebp, %esp
    pop %ebp
    ret
```

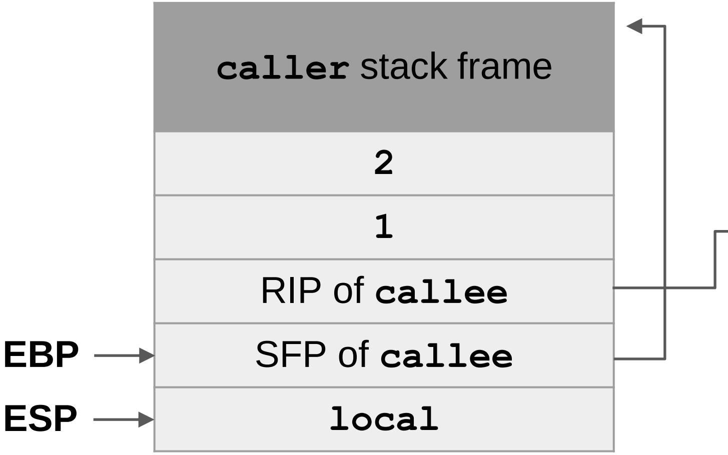
# x86 Function Call

```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```



- The next 3 steps restore the caller's stack frame.
- These instructions are sometimes called the function epilogue, because they appear at the end of every function.
- Sometimes the **`mov`** and **`pop`** instructions are replaced with the **`leave`** instruction.




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax
```

```
leave
ret
```

Function epilogue

EIP →

```
mov %ebp, %esp
pop %ebp
ret
```

# x86 Function Call

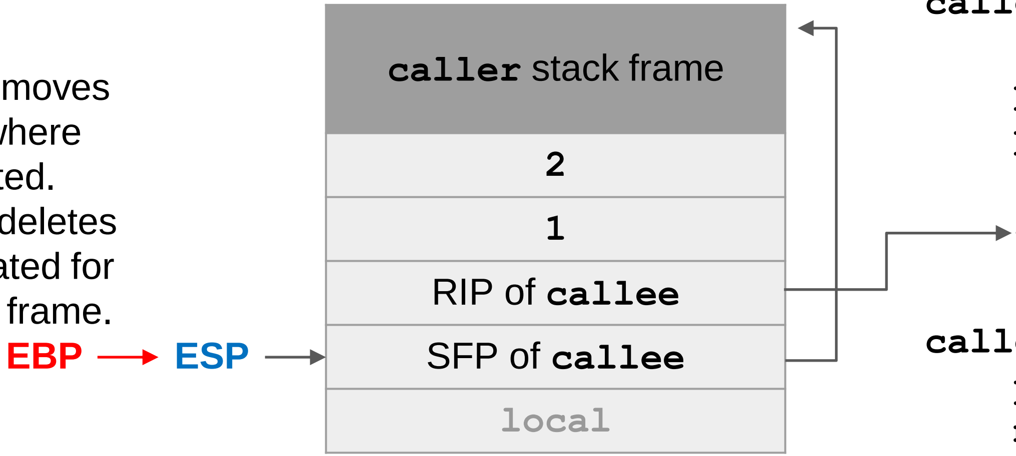
```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```



**8. Move ESP**

- This instruction moves the ESP up to where the EBP is located.
- This effectively deletes the space allocated for the callee stack frame.




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

EIP → pop %ebp

# x86 Function Call

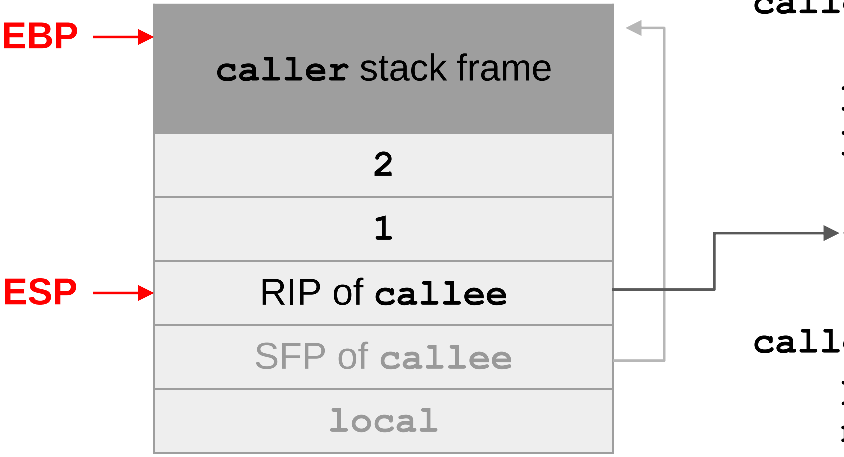
```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```



**9. Pop (restore) old EBP (SFP)**

- The `pop` instruction puts the SFP (saved EBP) back in EBP.
- It also increments ESP to delete the popped SFP from the stack.




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
EIP → ret
```

# x86 Function Call

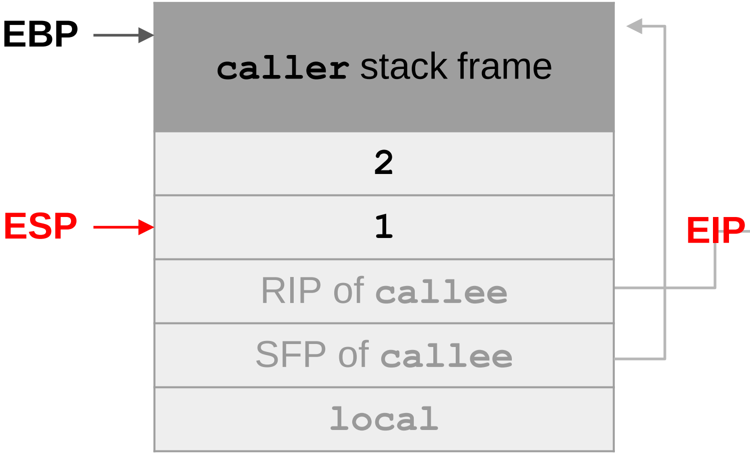
```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```



**10. Pop (restore) old EIP (RIP)**

- The `ret` instruction acts like `pop %eip`.
- It puts the next value on the stack (the RIP) into the EIP, which returns program execution to the caller.
- It also increments ESP to delete the popped RIP from the stack.




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

# x86 Function Call

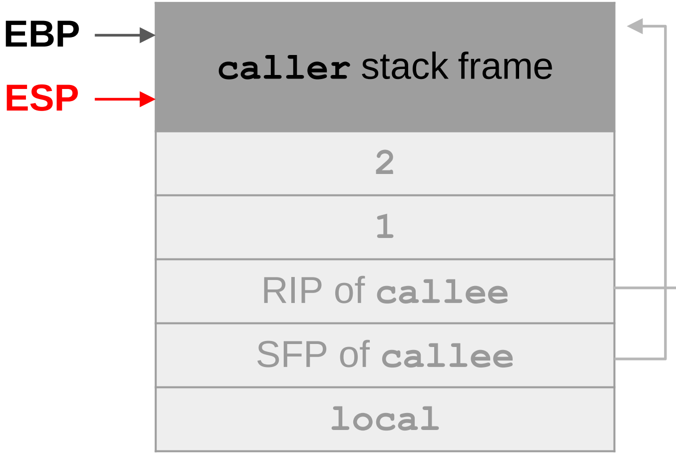
```
void caller(void) {
    callee(1, 2);
}
```

```
int callee(int a, int b) {
    int local;
    return 42;
}
```



**11. Remove arguments from stack**

- Back in the caller, we increment ESP to delete the arguments from the stack.
- The stack has returned to its original state before the function call!




```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
EIP → ...

callee:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

    mov %ebp, %esp
    pop %ebp
    ret
```

# Summary: x86 Assembly and Call Stack



- C memory layout
  - **Code** section: Machine code (raw bits) to be executed
  - **Static** section: Static variables
  - **Heap** section: Dynamically allocated memory (e.g. from `malloc`)
  - **Stack** section: Local variables and stack frames
- x86 registers
  - **EBP** register points to the top of the current stack frame
  - **ESP** register points to the bottom of the stack
  - **EIP** register points to the next instruction to be executed
- x86 calling convention
  - When calling a function, the old EIP (RIP) is saved on the stack
  - When calling a function, the old EBP (SFP) is saved on the stack
  - When the function returns, the old EBP and EIP are restored from the stack