x86 Architecture

CMPSC 403 Fall 2021

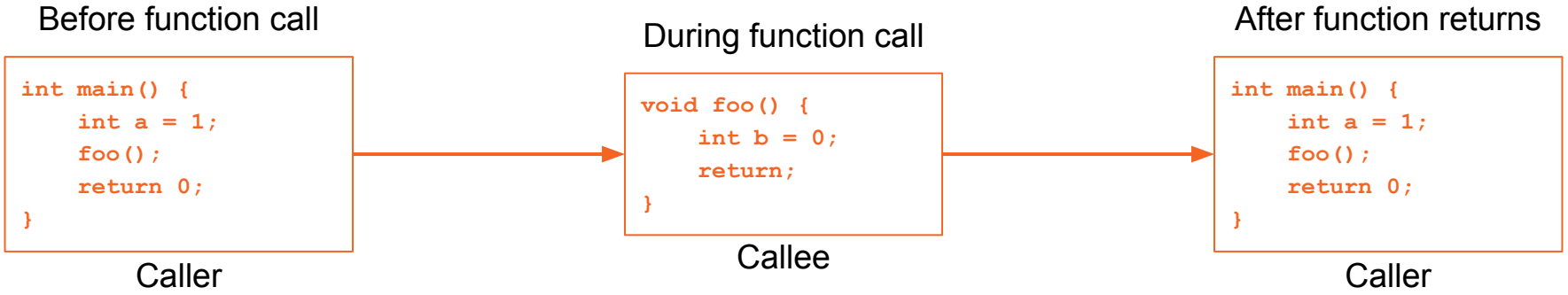
September 16, 2021

Take the Reflection Quiz 4

Check Discord for the link

x86 Calling Convention

Function Calls



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

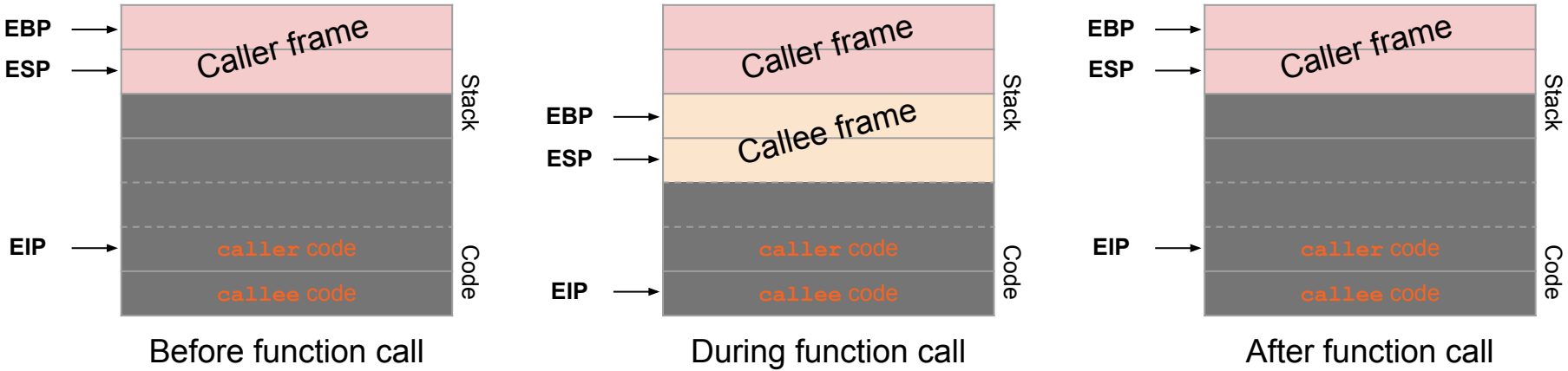
The callee function executes and then returns control to the caller function.

x86 Calling Convention

- 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**
- How to receive return values:
  - Return values are passed in EAX
- 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

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.



Steps of x86 Function Call

caller

1. Push arguments on the stack
2. Push old EIP, *instruction pointer*, on the stack (this value becomes RIP, return instruction pointer)
3. Move EIP

callee

4. Push old EBP, *base pointer*, on the stack (this becomes SFP, saved frame pointer)
5. Move EBP
6. Move ESP, *stack pointer*
7. Execute the function
8. Move ESP
9. Pop (restore) old EBP (SFP)

caller

10. Pop (restore) old EIP (RIP)
11. Remove arguments from stack

X86 Function Call

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

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

Here is a snippet of C code

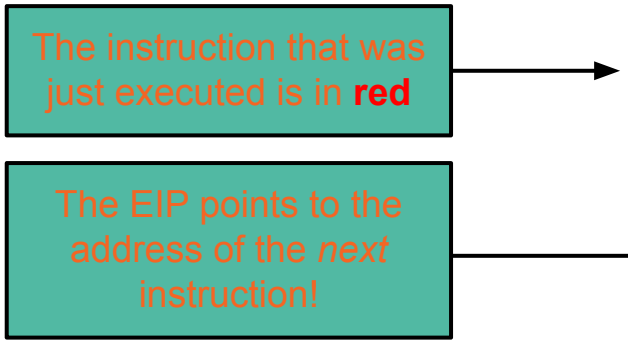
The code compiled into x86 assembly

```
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;
}
```



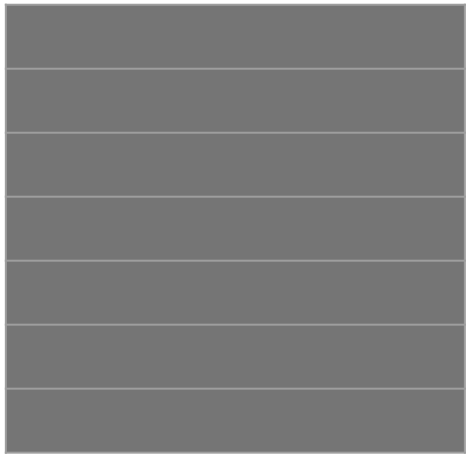

```
caller:
    ...
EIP → push $2
    push $1
    call callee
    add $8, %esp
    ...

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

    mov $42, %eax

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

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

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

X86 Function Call

Diagram of the stack. Remember, each row represents 4 bytes (32 bits).



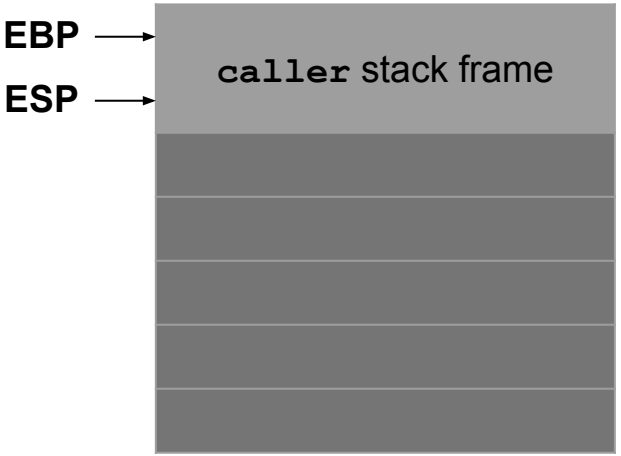
```
caller:
    ...
EIP → push $2
    push $1
    call callee
    add $8, %esp
    ...

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

    mov $42, %eax

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

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

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

X86 Function Call

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




```
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
```

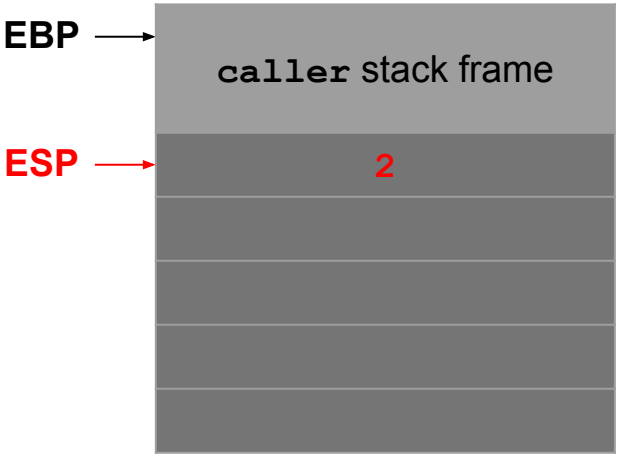
```
void caller(void) {
    callee(1, 2);
}
```

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

X86 Function Call

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
```

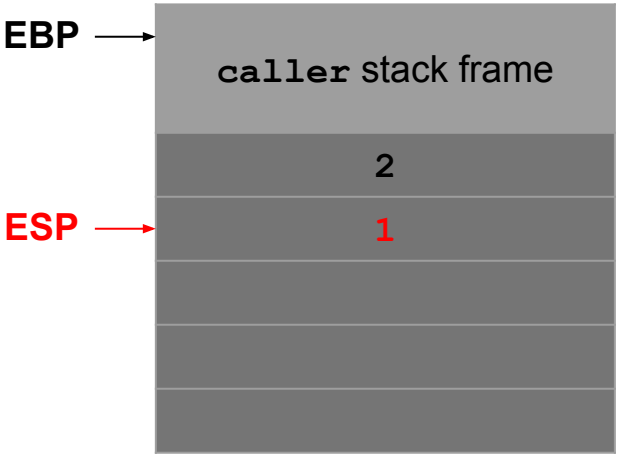
```
void caller(void) {
    callee(1, 2);
}
```

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

X86 Function Call

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
```

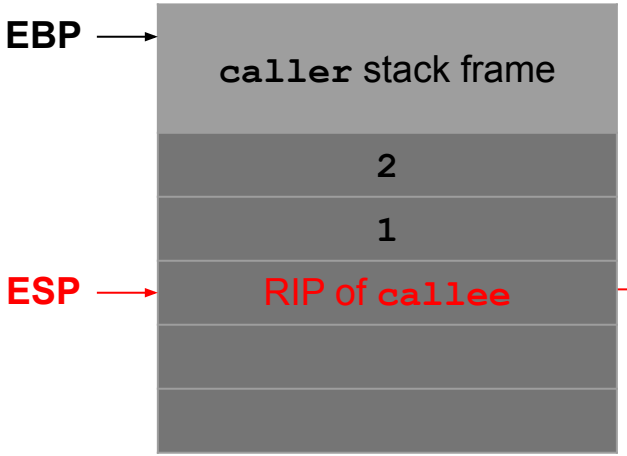
```
void caller(void) {
    callee(1, 2);
}
```

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

X86 Function Call

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.



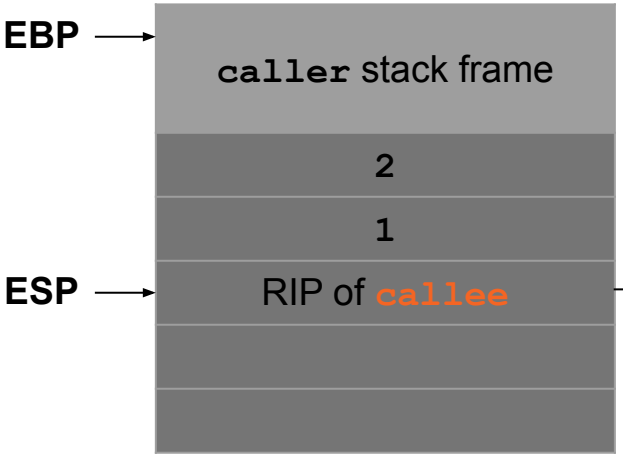

```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

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

    mov $42, %eax

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

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

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

X86 Function Call

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

      mov $42, %eax

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

Function prologue

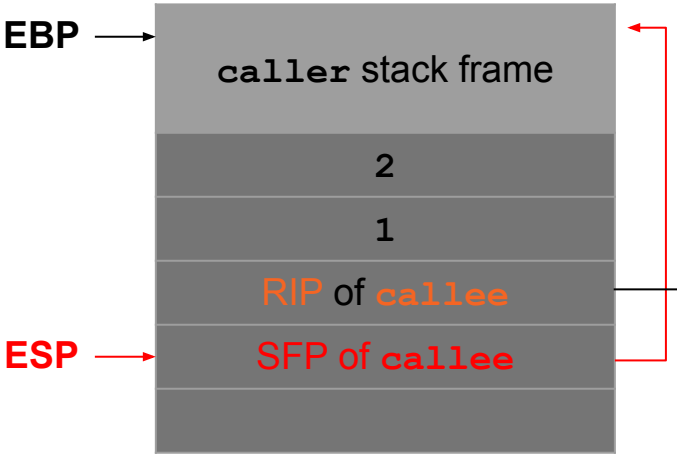
```
void caller(void) {
    callee(1, 2);
}
```

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

X86 Function Call

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
EIP → mov %esp, %ebp
    sub $4, %esp

    mov $42, %eax

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

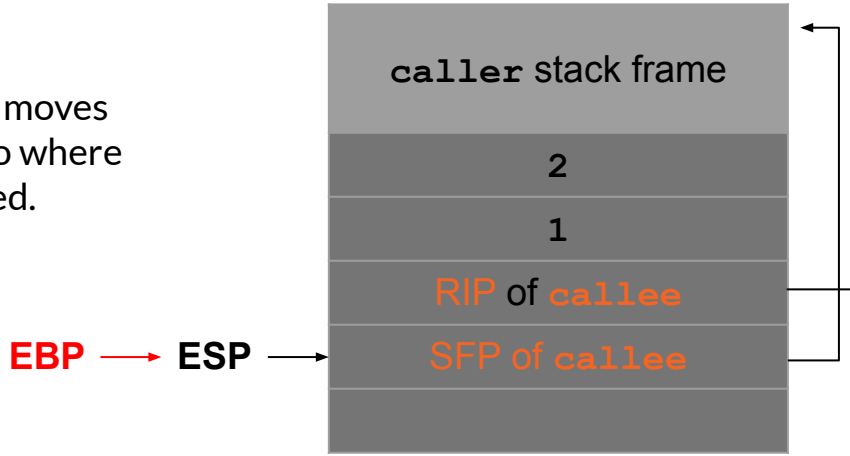
```
void caller(void) {
    callee(1, 2);
}
```

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

X86 Function Call

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
EIP → sub $4, %esp

    mov $42, %eax

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

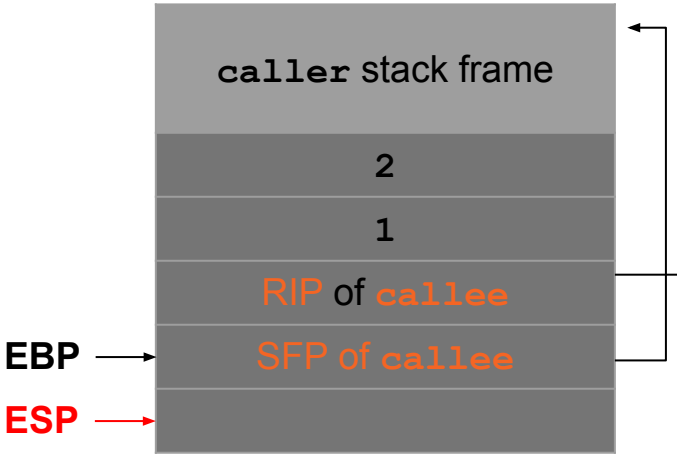
```
void caller(void) {
    callee(1, 2);
}
```

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

X86 Function Call

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
```

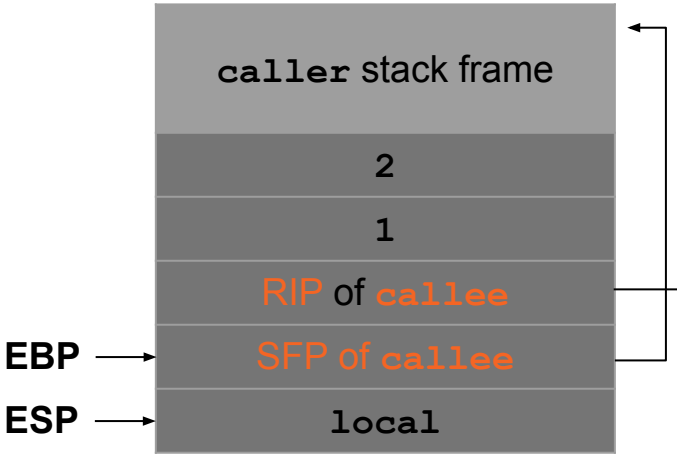
```
void caller(void) {
    callee(1, 2);
}
```

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

X86 Function Call

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.)



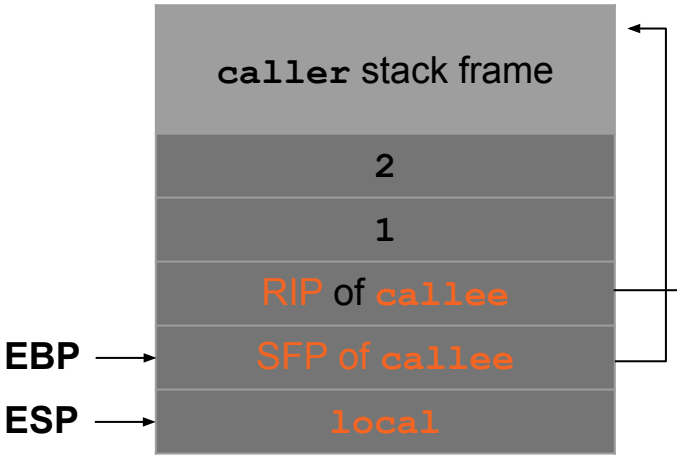

```
caller:
    ...
    push $2
    push $1
    call callee
    add $8, %esp
    ...

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

    mov $42, %eax

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

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

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

X86 Function Call

- 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 pseudo-instruction **leave**.




```
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 →

```
leave
ret
```

Function epilogue

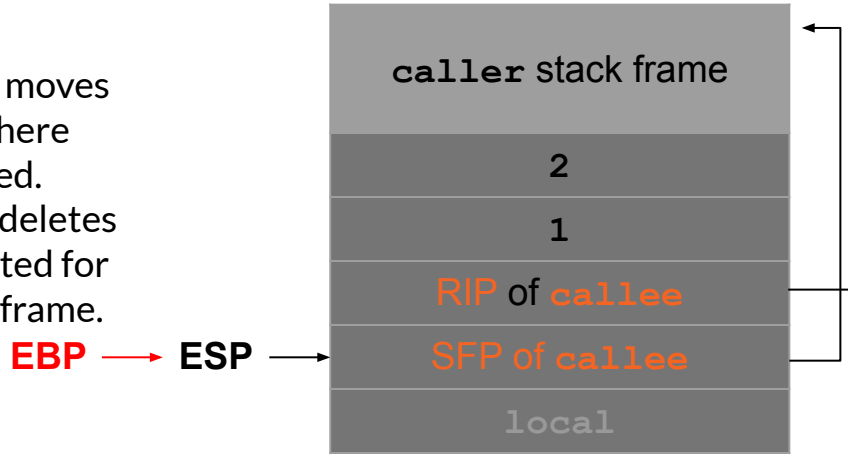
```
void caller(void) {
    callee(1, 2);
}
```

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

X86 Function Call

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 →

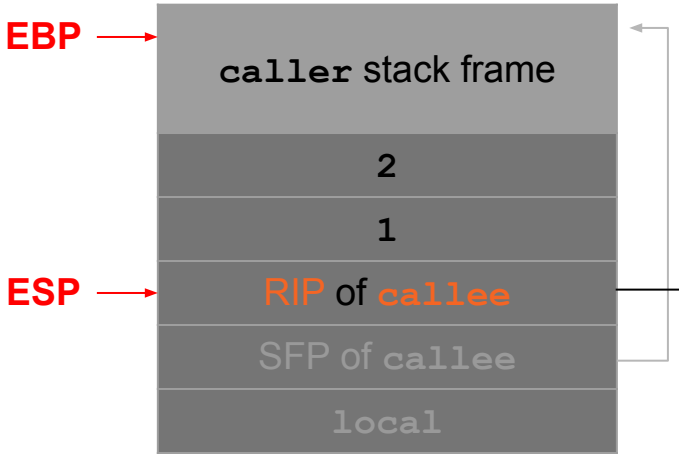
```
void caller(void) {
    callee(1, 2);
}
```

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

X86 Function Call

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
```

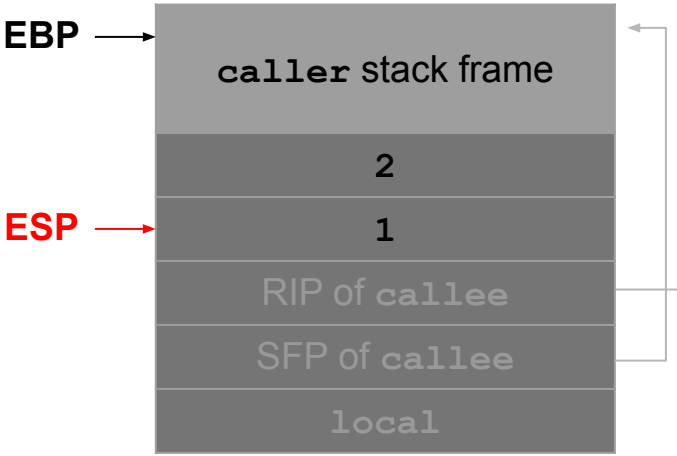
```
void caller(void) {
    callee(1, 2);
}
```

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

X86 Function Call

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
```

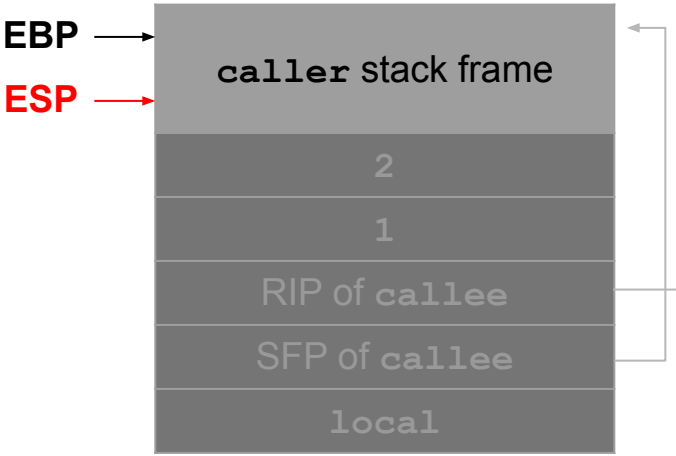
```
void caller(void) {
    callee(1, 2);
}
```

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

X86 Function Call

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
```