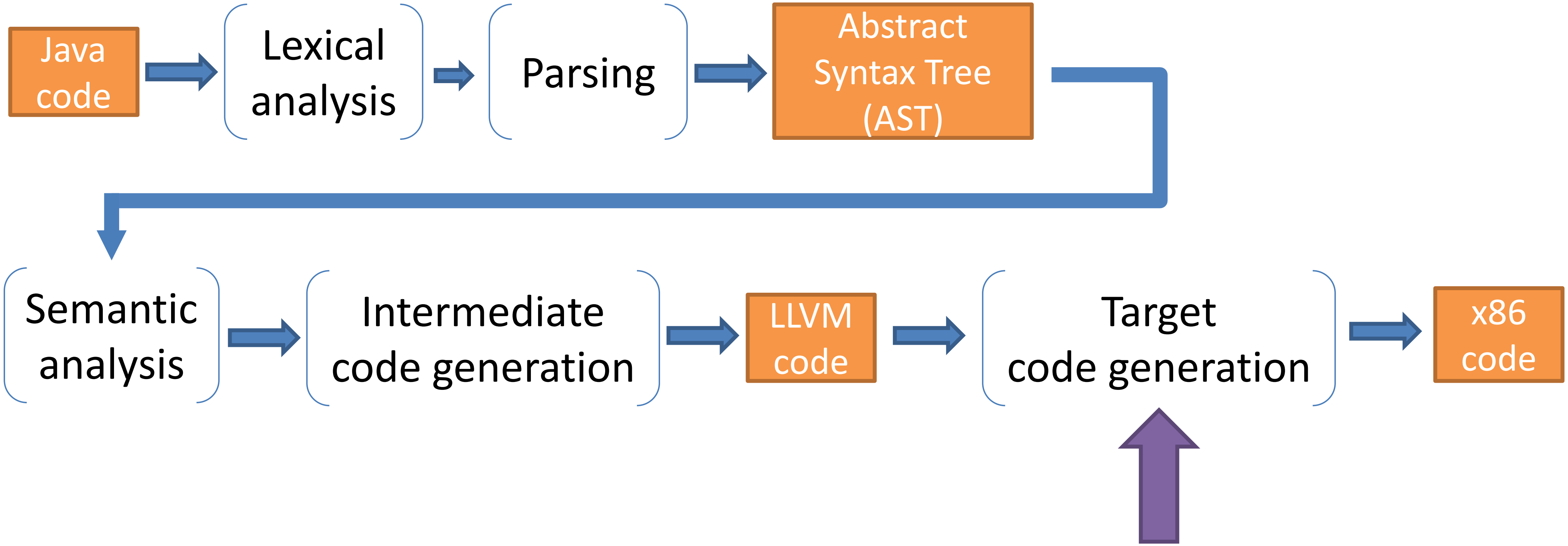
Compiler Construction
Winter 2020

Recitation 12:
Activation Records

Yotam Feldman

Based on slides by Technion compilers class’ staff
and Guy Golan-Gueta

Assembly Code Generation



Lowering (LLVM) to Assembly

- Different instruction set
- Unbounded number of registers
  - Register allocation & spilling
- Function calls
  - Activation records

What’s in a Procedure

- A procedure needs access to
  - Its local variables
  - Its parameters
  - Return address

```
int add(int x, int y)
{
   int inc = x;
   inc = inc + y;
   return inc;
}
```

The Deep Dive: Recursion

- Where are the arguments / local variables of each invocation stored?
- How do we know to access the correct ones?

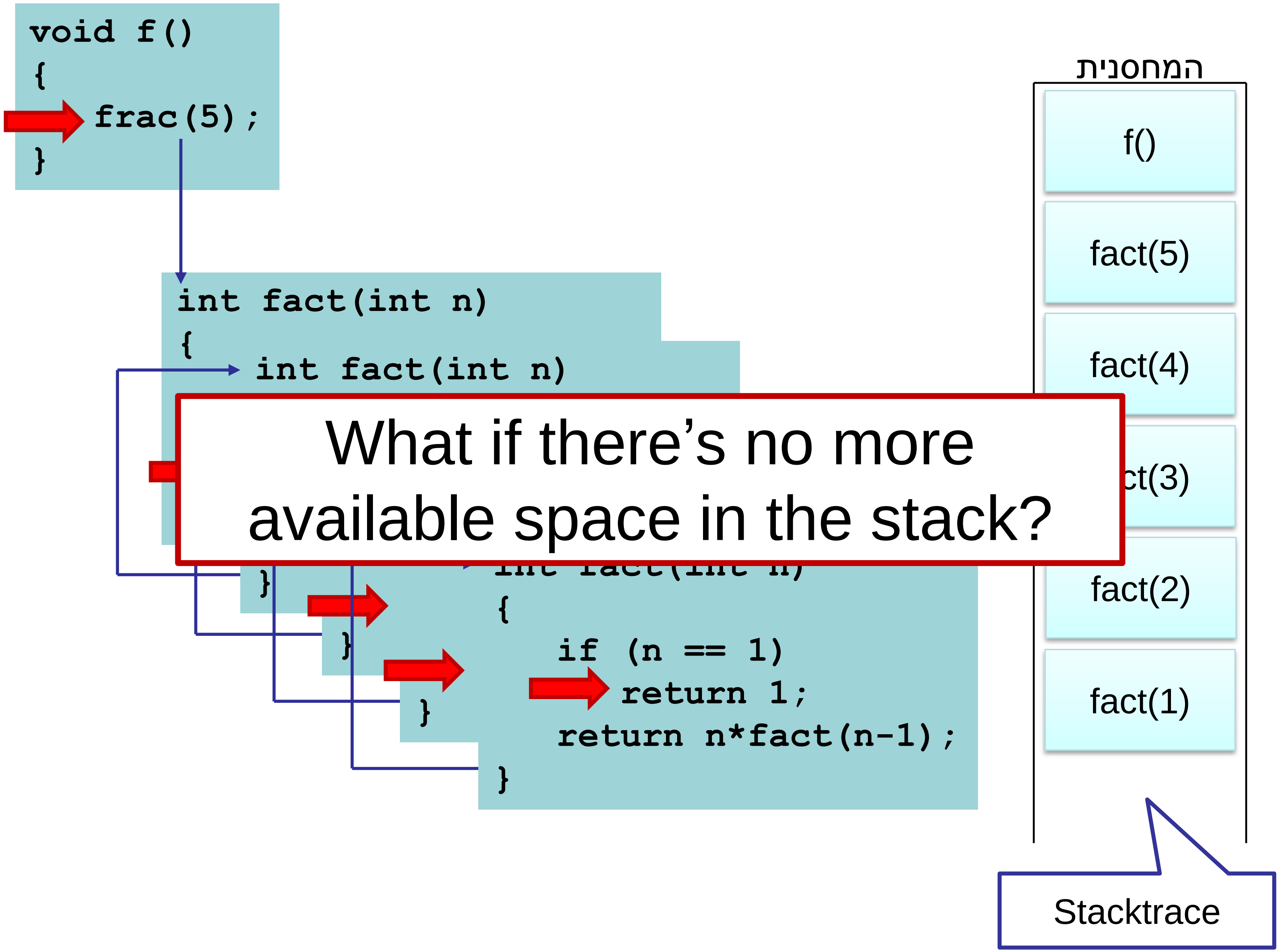
- How do we know to which **`fact`** invocation to return? Or to **`f`**?

```
int fact(int n)
{
    if (n == 1)
        return 1;
    return n*fact(n-1);
}

void f()
{
    fact(5);
}
```

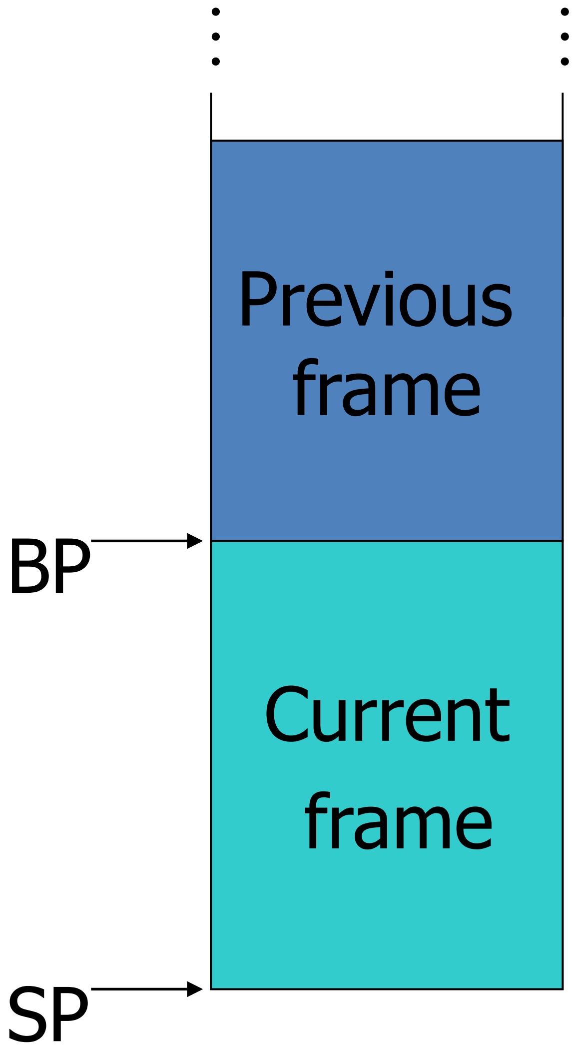
Activation Records / Stack Frames

- Data structure **per procedure invocation**
- Records all the necessary information
- Stored in the stack
- At **runtime**, an activation record is allocated for each invocation
  - Allocated when the procedure is called
  - Released when the procedure terminates


void f()
{
frac(5);
}
int fact(int n)
{
int fact(int n)
What if there’s no more available space in the stack?
{
if (n == 1)
return 1;
return n*fact(n-1);
}
המחסנית
f()
fact(5)
fact(4)
fact(3)
fact(2)
fact(1)
Stacktrace

Runtime Stack

- Stack grows downwards (towards smaller addresses)
- ***BP*** – base / frame pointer
  - base of current frame
- ***SP*** – stack pointer
  - top of current frame
  - last allocated value



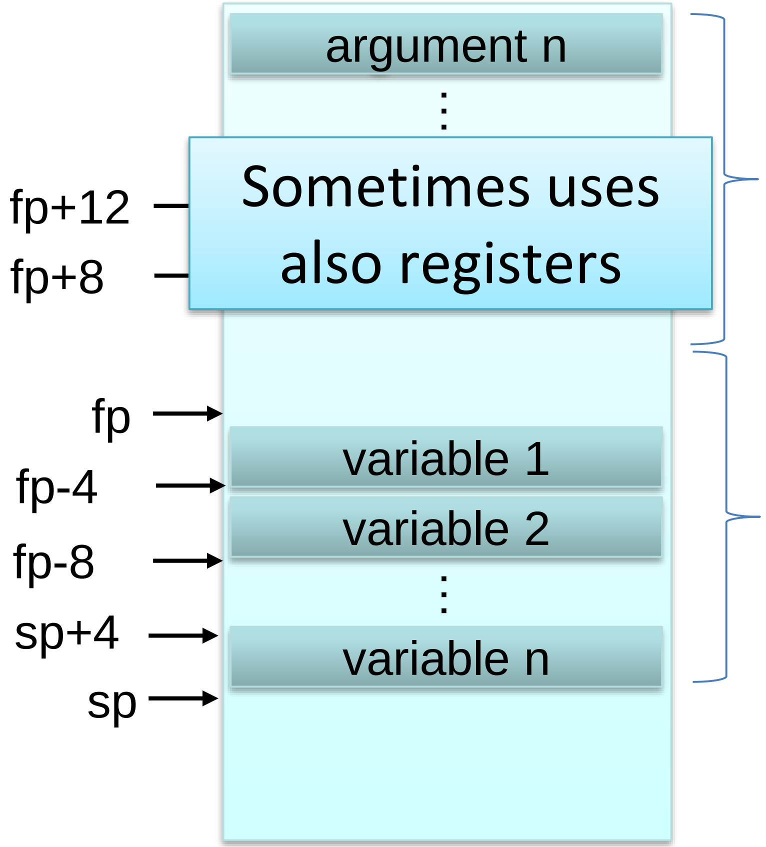
Activation Record's Contents

How can we execute our code while…

- Finding arguments?
- Finding local variables?

Global variables access via their fixed address

Heap variables by following pointers from other variables



* 32 bit addresses

* Layout may change between architectures and operating systems

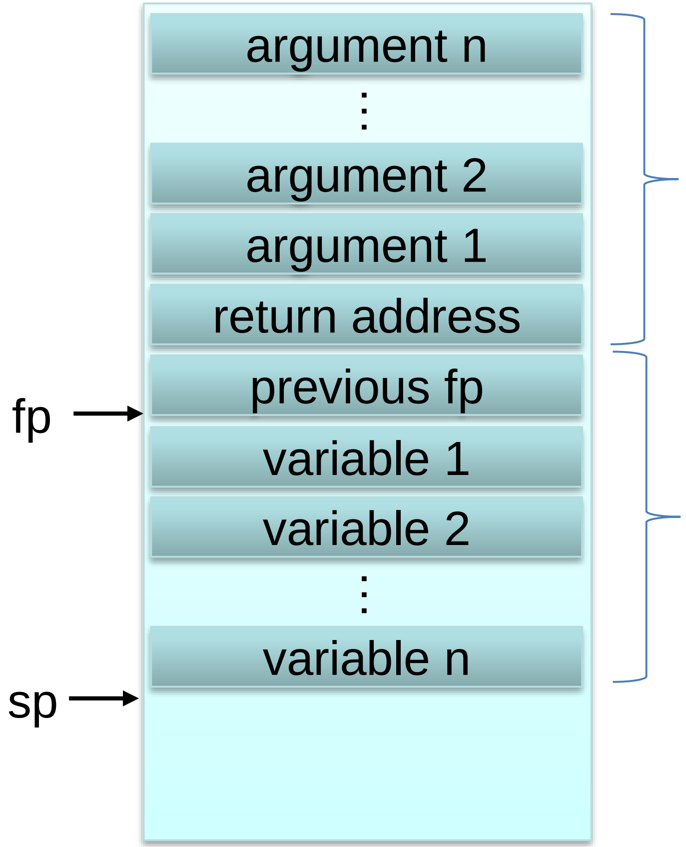
Activation Record’s Contents

How can we execute our code while…

- Finding arguments?
- Finding local variables?

How can we return to the caller’s context…

- Instruction pointer?
- Activation record?
- Registers?



* Layout may change between architectures and operating systems

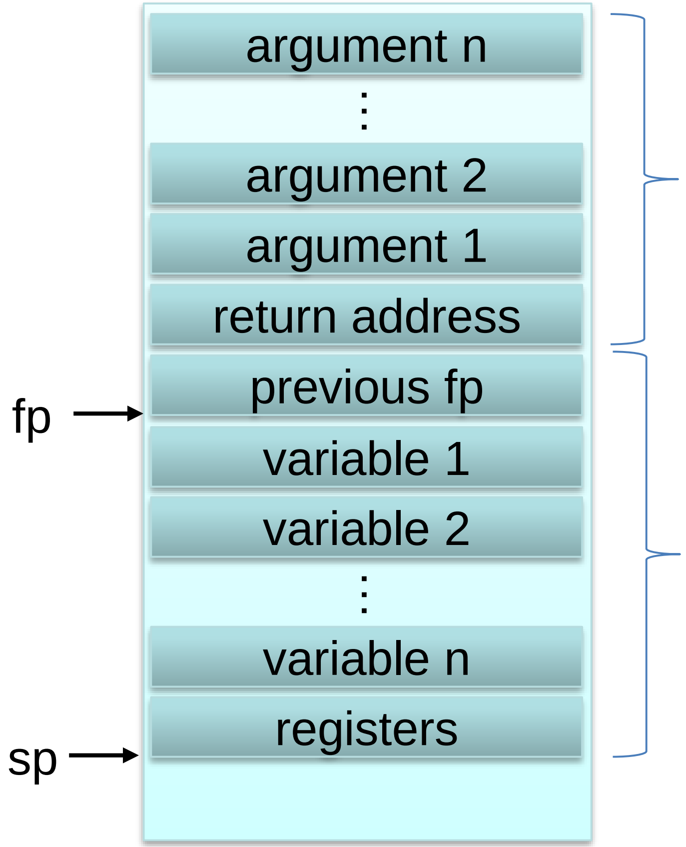
Activation Record's Contents

How can we execute our code while…

- Finding arguments?
- Finding local variables?

How can we return to the caller's context…

- Instruction pointer?
- Activation record?
- Registers?



* Layout may change between architectures and operating systems

Application Binary Interface: Things to Be Done (and By Whom) (and How)

caller

callee

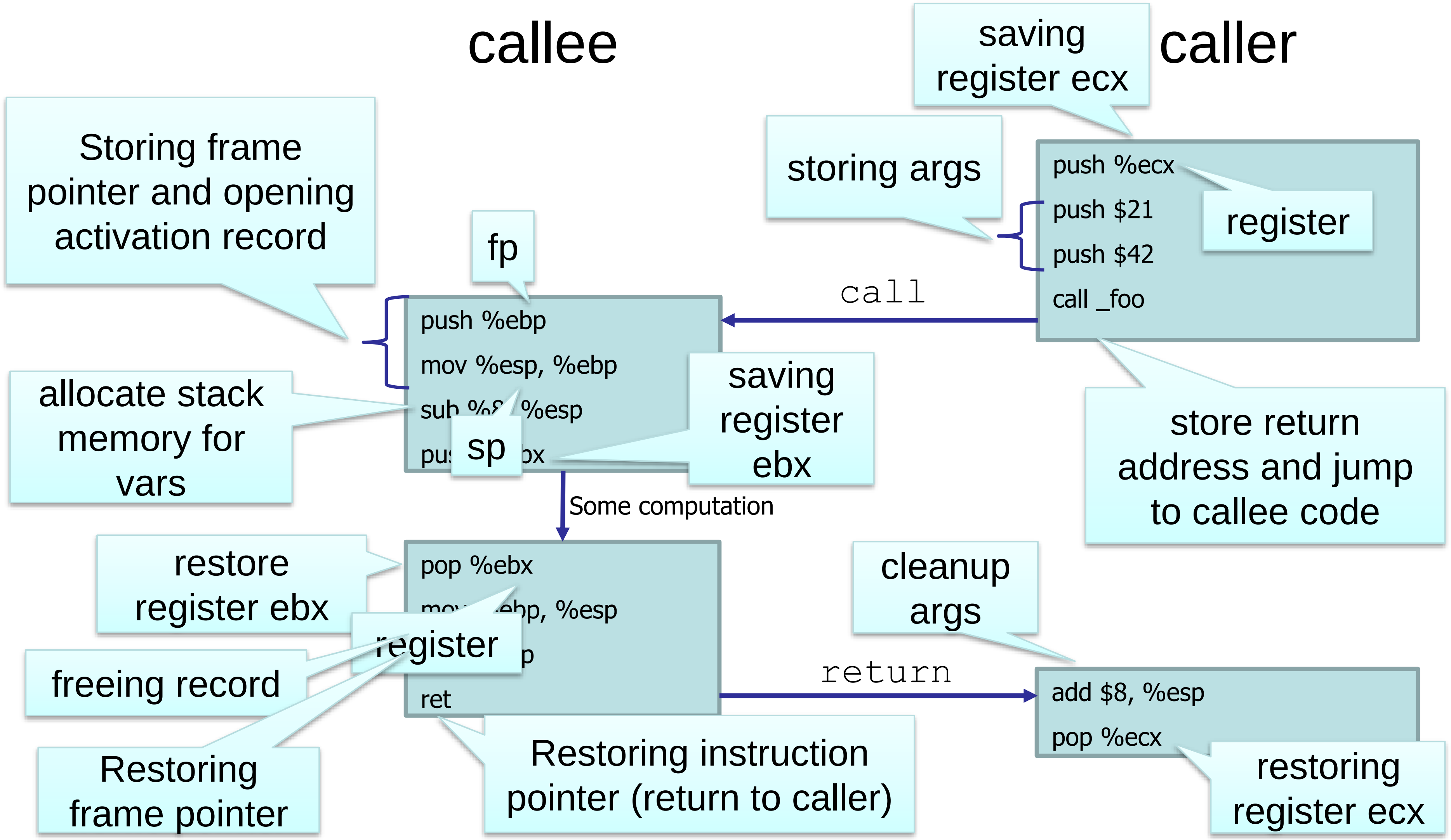
Upon call:

- Storing arguments
- Storing return address
- Storing frame pointer
- Allocating stack space for registers
- Storing registers
- Allocating stack space for local variables

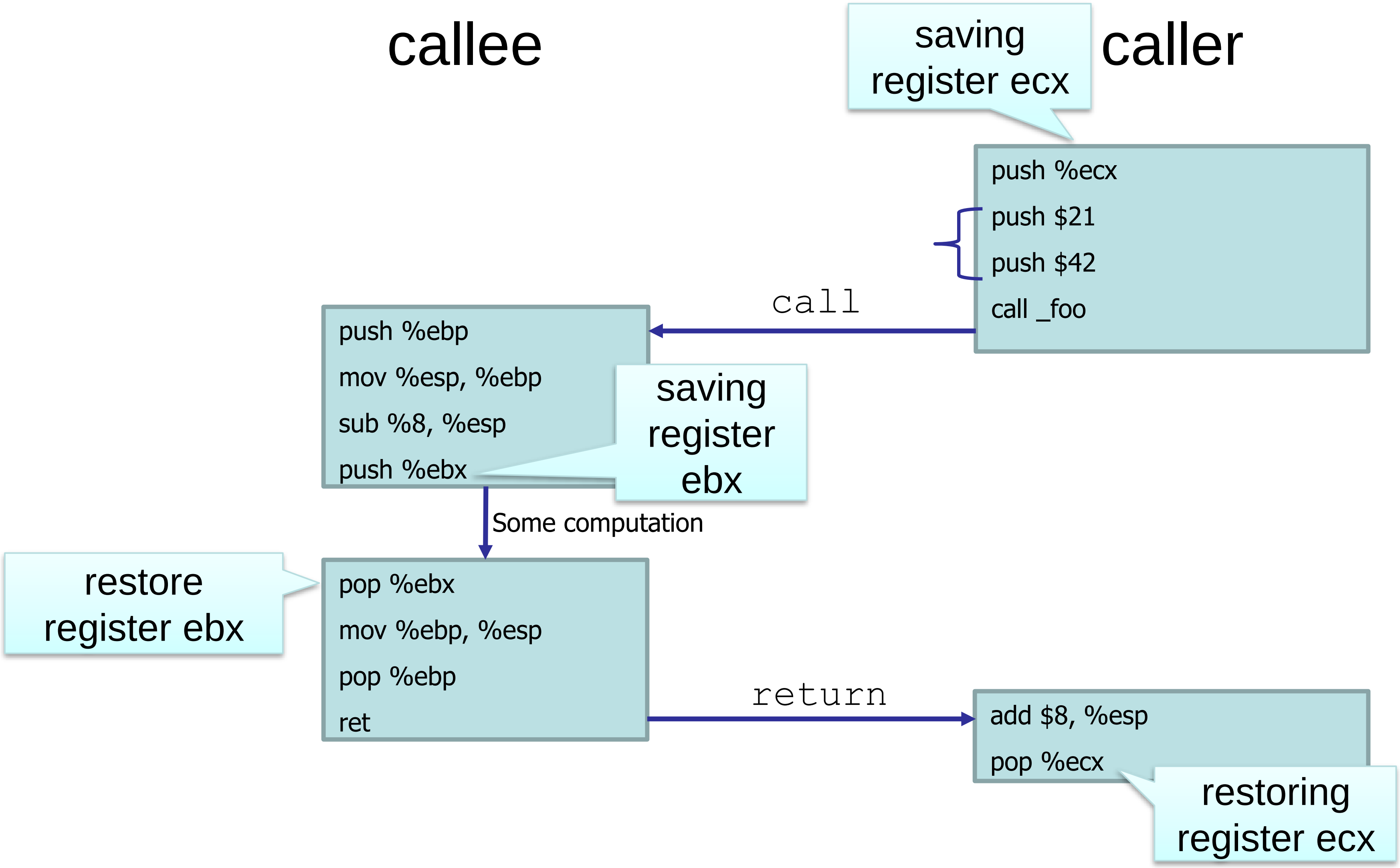
Upon return:

- Deallocating stack space for registers
- Deallocating stack space for local variables
- “Cleanup” arguments
- Storing return value
- Restoring base pointer
- Restoring instruction pointer

Example Application Binary Interface (ABI) in x86



Caller- and Callee-Saved Resigters



Register Preservation

Who's responsible to store and backup important registers?

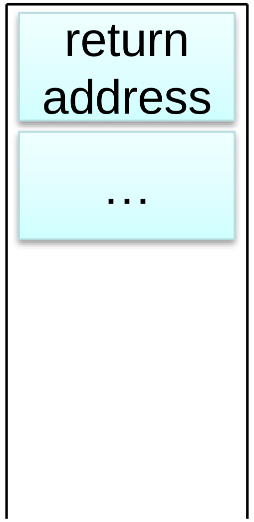
- Caller knows which registers need to be preserved
- Callee knows which registers it overwrites

- Callee-saved: Caller guaranteed that they are not modified by the callee, or restored before callee returns
  - In x86: ebp, esp, ebx, edi, …
- Caller-saved: Can be modified by the callee, the caller needs to store them before the call if it needs them
  - In x86: eax, ecx, edx, …

- The compiler's register allocation chooses between callee- and caller-saved
  - And generate code that respects the rules

Passing Arguments

<u>In a register</u>




```
mov eax, 5
mov ebx,37
call f
```

<u>On the stack</u>


```
push 5
push 37
call f
```

```
int f(int a, int b)
{
    …
}
```

```
void g()
{
    f(5,37);
}
```

Passing Arguments

In a register

- Limited number of registers
- Register preservation

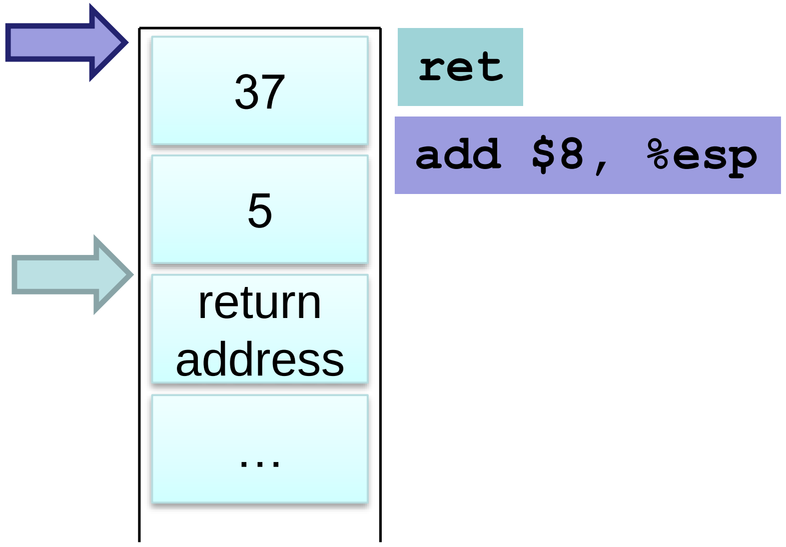
On the stack

- Slower access
- Need to cleanup

- Most x86 (cdecl,stdcall): arguments on the stack
- x86_64: first arguments in designated (caller-saved) registers,
  rest on the stack

Argument Cleanup

By caller

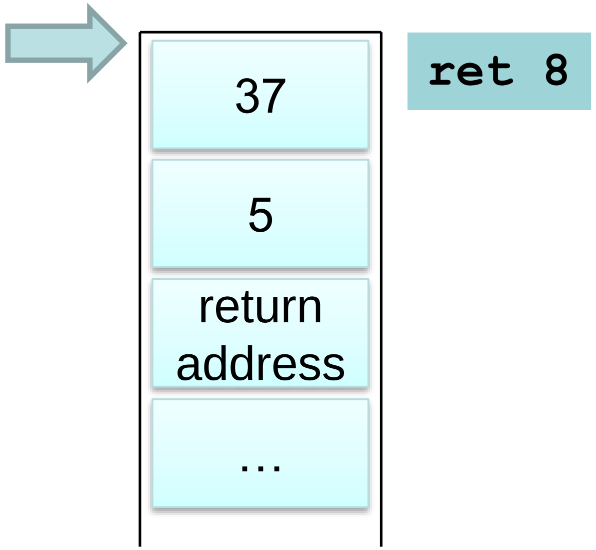


- e.g. cdecl, …

vararg

```
printf("%d",1);
printf("%d,%d",1,2);
```

By Callee



- e.g. stdcall, …

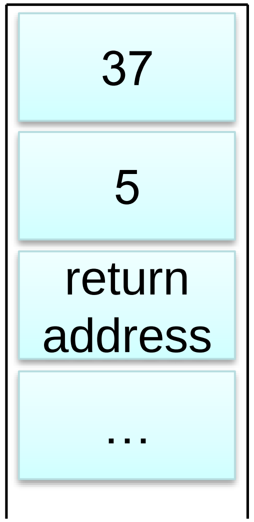
Smaller binaries

Order of Arguments on the Stack

<u>Left to right</u>


```
push 5
push 37
call f
```

<u>Right to left</u>




```
push 37
push 5
call f
```

- e.g. cdecl, stdcall, …

Return Value

<u>In a register</u>

- Limited number of registers
- Register preservation

<u>On the stack</u>

- Slower access
- Need to cleanup

- What if we want to return something that doesn’t fit in a register?

Return Address

- In a designated register or on the stack?
- Store the current instruction or the next instruction?

- ❖ In practice, this is decided by the architecture's "call" operation

Which is Best?

- No “correct” answer
- Depends on
  - Processor capabilities,
  - Applications’ characteristics
  - Conventions
- Caller & callee must agree on the calling convention!
  - Interoperability between compilers
  - Or with explicit directives:

```
int __cdecl system(const char *);
```

Summary

- Runtime stack
- Activation records
- Frame pointer, stack pointer
- Calling conventions