

How Programming Languages Work (Basics)

Kenjiro Taura

2024/05/26

Contents

Introduction	2
CPU and machine code : an overview	9
A glance at ARM64 machine (assembly) code	18
ARM64 instructions	26

Introduction



Why you want to make a language, today?

- new hardware
 - GPUs, AI chips, Quantum, ...
 - new instructions (e.g., SIMD, matrix, ...)
- new general purpose languages
 - Scala, Julia, Go, Rust, etc.

Why you want to make a language, today?

- special purpose (domain specific) languages
 - statistics (R, MatLab, etc.)
 - data processing (SQL, NoSQL, SPARQL, etc.)
 - deep learning
 - constraint solving, proof assistance (Coq, Isabelle, etc.)
 - macro (Visual Basic (MS Office), Emacs Lisp (Emacs), Javascript (web browser), etc.)

Taxonomy : interaction mode

- **interactive / read-eval-print-loop (REPL)**
 - type code directly or load source code in a file interactively
 - Julia
- **batch compile**
 - convert source into an executable file
 - and run it (typically the “main” function)
 - Go, Rust
- some language implementations provide both
 - OCaml

Taxonomy : execution strategy

- **interpreter** executes source code directly with its input
 - interpreter (*source-code, input*) \rightarrow *output*
- **compiler** first converts source code into **a machine (assembly) code** that is directly executed by the CPU
 - compiler (*source-code*) \rightarrow *machine-code*;
 - *machine-code (input)* \rightarrow *output*
- **translator** or **transpiler** are like compiler, but convert into another language, not machine (assembly) code

A (minor) note: machine code vs. assembly code

- in many contexts, they are used almost interchangeably
- machine (assembly) *languages* are almost interchangeable, too
- if asked a difference,
 - **machine** code is *the* real encoding of instructions interpretable by a CPU
 - **assembly** code refers to a *textual (human-readable) representation* of machine code

Taxonomy : compiler/translator

- **ahead-of-time (AOT)** compiler converts all the program parts into assembly before execution
- **just-in-time (JIT)** compiler converts program parts incrementally as they get executed (e.g., a function at a time)

CPU and machine code : an overview

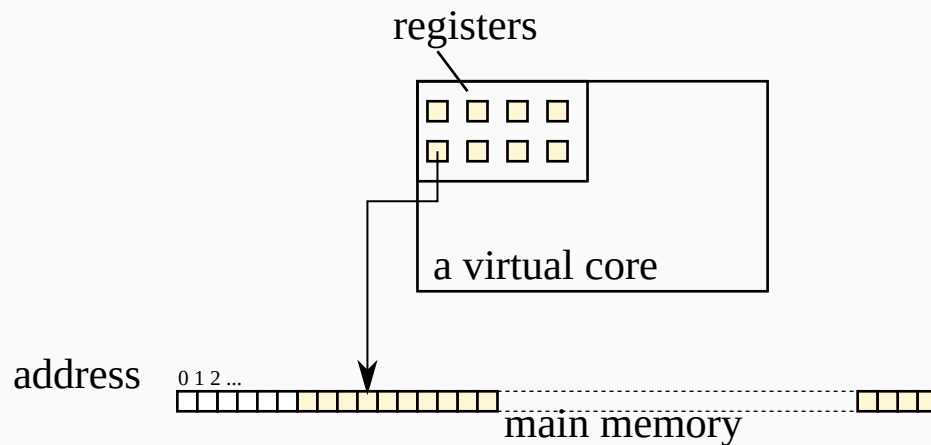
High-level (programming) languages vs. assembly languages

- assembly *is* just another programming language
- it has many features present in programming languages

high-level language	assembly language
variables	<i>registers</i> and <i>memory</i>
structs and arrays	memory and load/store instructions
expressions	arithmetic instructions
if / loop	compare, conditional branch instructions
functions	branch and link instructions

What a CPU looks like

- has a small number (typically < 100) of *registers*
 - each register can hold a small amount of (e.g., 64-bit) data
- \Rightarrow majority of data are stored in the *main memory*
 - a few GB to >1000 GB

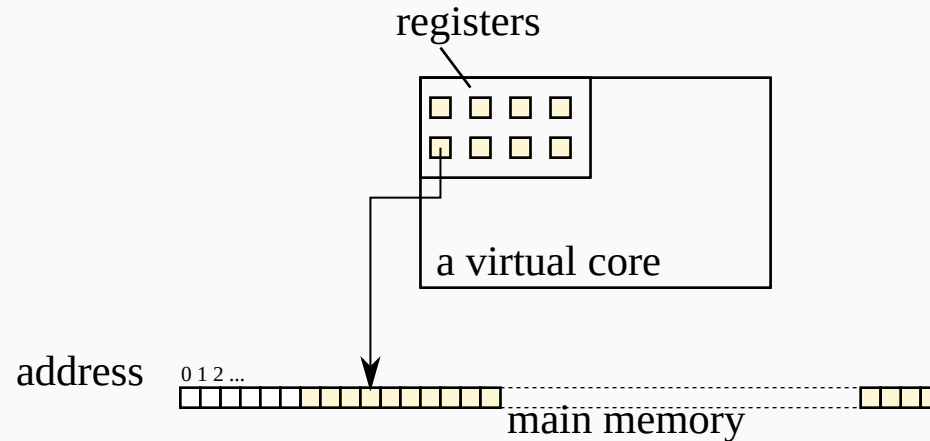


Terminology note : CPU \ni core \ni virtual core

- a *CPU* has multiple (typically, 2 to >100) *cores*
- a core has multiple (typically, 1 to a few) *virtual cores* or *hardware threads*
- each virtual core has its own registers and is capable of fetching and executing instructions
- a single instruction sequence is executed by a single OS-level *thread*, which is on a single virtual core at any given time
- this course is only concerned about single-thread execution

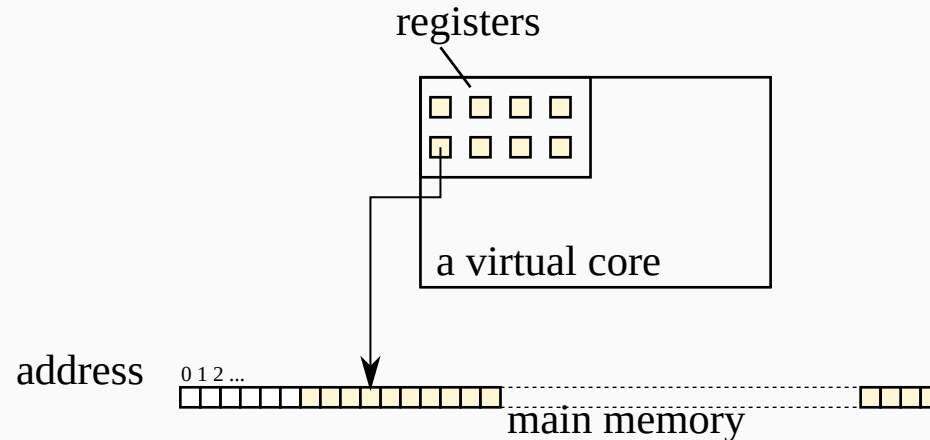
Main memory

- \approx a large array indexed by integers, called *addresses*
- in machine code level, *an address is just an integer*



Main memory

- each address typically stores 8 bits (a *byte*) of data
- a larger word is stored in consecutive addresses. e.g.,
 - 32 bit (4 byte) word occupies 4 consecutive addresses
 - 64 bit (8 byte) word occupies 8 consecutive addresses



What a virtual core does

- a virtual core is a machine that does the following:

repeat:

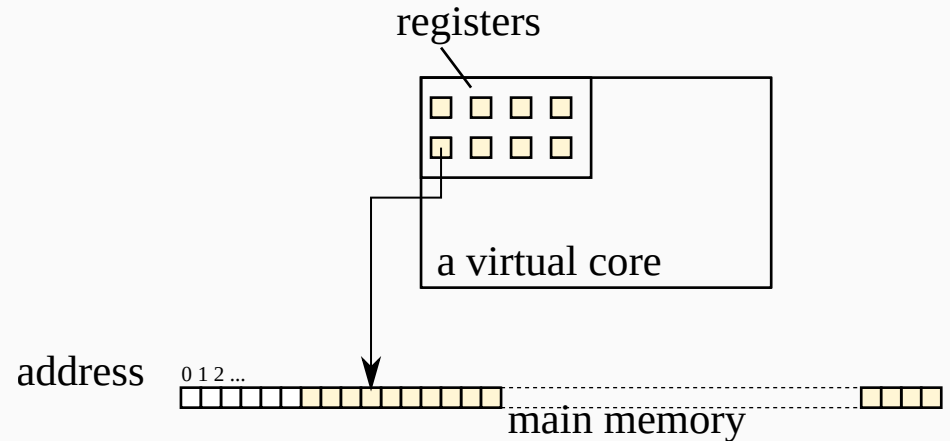
```
instruction = memory[program counter]
```

```
execute instruction
```

- *program counter* (or *instruction pointer*) is the register that specifies the address to fetch the next instruction from

What an instruction does

1. perform some computation on specified register(s) or a memory address
2. change the program counter,
 - typically to the address of the instruction immediately following it
 - except branch / jump instructions



Exercise objectives

- `pl06_how_it_gets_compiled`
- learn how a *compiler* does the job
- by inspecting assembly code generated from functions of the source language

A glance at ARM64 machine (assembly) code

A glance at ARM64 machine (assembly) code

Rust (add123.rs)

```
#[no_mangle]
pub fn add123(x:i64, y:i64) -> i64 {
    y + 123
}
```

```
$ rustc -O --emit asm --crate-type lib
add123.rs -o add123.s
```

assembly (add123.s)

```
.text
.file    "pl06.lebfa1..."
.section .text.add123,...
.globl   add123
.p2align        2
.type     add123,@function

add123:
.cfi_startproc
add      x0, x1, #123
ret

.Lfunc_end0:
.size    add123, .Lfunc_end0-.Lfunc_start0
.cfi_endproc
```

Insignificant lines

- indented lines starting with a dot (e.g., `.file`, `.section`, `.text`, etc.) are *directives* (not instructions) and largely not important
- unindented lines ending in a colon (e.g., `add123:`) are *labels* used to human-readably specify jump targets

```
.text
.file    "pl06.lebfa1..."
.section .text.add123,...
.globl   add123
.p2align          2
.type     add123,@function

add123:
    .cfi_startproc
    add     x0, x1, #123
    ret
.Lfunc_end0:
    .size
add123, .Lfunc_end0-add123
    .cfi_endproc
```

Insignificant lines

- indented lines starting with a dot (e.g., `.file`, `.section`, `.text`, etc.) are *directives* (not instructions) and largely not important
- unindented lines ending in a colon (e.g., `add123:`) are *labels* used to human-readably specify jump targets

```
add123:
```

```
    add    x0, x1, #123
```

```
    ret
```

```
.Lfunc_end0:
```

How to read assembly

- focus on lines that are *instructions*
- look for a label *similar to* the function name — where its instructions start
 - the label may not be exactly the same as the function name (*name mangling*)

add123:

add x0, x1, #123

ret

.Lfunc_end0:

How to read instructions

- ex.

```
add    x0, x1, #123
```

performs $x0 = x1 + 123$

- `add` is an *instruction name* or *mnemonic*
- takes a few *operands* (`x0`, `x1`, and `#123`)
 - `x0`, `x1` : register
 - `#123` : constant (*immediate value* or *literal*)

ARM64 registers

- integer registers $\times 32$
 - 64 bit : $x0, x1, \dots, x31$
 - 32 bit : $w0, w1, \dots, w31$
 - the lower 32 bits of $x0, x1, \dots, x31$
- floating point registers $\times 32$
 - 64 bit (double precision) : $d0, d1, \dots, d31$
 - 32 bit (single precision): $s0, s1, \dots, s31$
 - the lower 32 bits of $d0, d1, \dots, d31$

Implicit registers

- *condition code register (CC)* — holds the result of the last compare instruction
- *program counter register (PC)* — holds the address of the next instruction

SIMD registers

- pack a few floating point numbers / integers in a register
- a SIMD instruction can perform an operation on all values on SIMD register(s)
- important for performance, but omitted in this course for brevity

ARM64 instructions

ARM64 instruction categories

- arithmetic / move
- load / store
- compare and conditional branches
- unconditional branch
- branch-and-link and return

Sources

- when you encounter unfamiliar instructions, see [Arm A-profile A64 Instruction Set Architecture](#)
- Cheat sheet
- Google / AI

Arithmetic / move

assembly	pseudo C
<code>sub x0, x1, x2</code>	<code>x0 = x1 - x2</code>
<code>add x0, x1, x2</code>	<code>x0 = x1 + x2</code>
<code>mov x0, x1</code>	<code>x0 = x1</code>
<code>. . .</code>	<code>. . .</code>

- typically takes three operands
- the result is written to the first operand

Load / store

	assembly	pseudo C
basic load	<code>ldr x0, [x1]</code>	<code>x0 = *(long*)x1</code>
basic store	<code>str x0, [x1]</code>	<code>*(long*)x1 = x0</code>
offset	<code>ldr x0, [x1, #8]</code>	<code>x0 = *(long*)(x1+8)</code>
scaled offset	<code>ldr x0, [x1, x2, lsl #3]</code>	<code>x0 = *(long*)(x1 + (x2<<3))</code>
pre-increment	<code>ldr x0, [x1, #8]!</code>	<code>x1 += 8; x0 = *(long*)x1</code>
post-increment	<code>ldr x0, [x1], #8</code>	<code>x0 = *(long*)x1; x1 += 8</code>
negative offset	<code>ldur x0, [x1, #-8]</code>	<code>x0 = *(long*)(x1 - 8)</code>
load pair	<code>ldp x0, x1, [x2]</code>	<code>x0 = *(long*)x2; x1 = *(long*)(x2 + 8)</code>

- there are similar variations for store

Compare and conditional branches

<code>cmp x0, x1</code>	$CC = x0 - x1$
<code>b.eq label</code>	if $CC = 0$, goto <i>label</i>
<code>b.lt label</code>	if $CC < 0$ (signed), goto <i>label</i>
<code>b.le label</code>	if $CC \leq 0$ (signed), goto <i>label</i>
<code>b.lo label</code>	if $CC < 0$ (unsigned), goto <i>label</i>
...	...

- notes:
 - $CC = \textit{condition code register}$
 - CC does not hold the value of $x0 - x1$ itself
 - but holds whether it is > 0 , $= 0$, < 0 , etc. as a bit sequence

Other jump variants

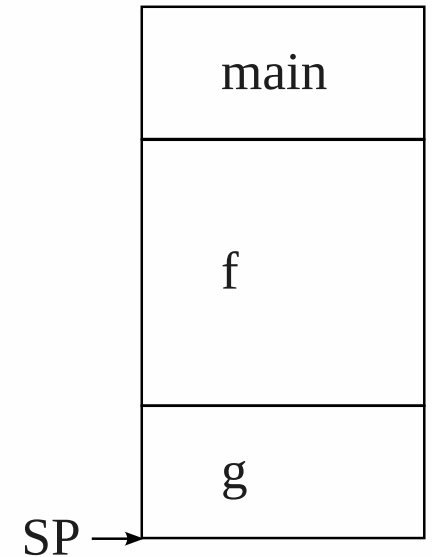
b <i>label</i>	goto <i>label</i>	
bl <i>label</i>	goto <i>label</i> ; <code>x30</code> = the next address of the bl	(*)
ret	goto the address in <code>x30</code>	(†)

- (*) used for calling a function; set `x30` to the address to return after the function
- (†) used for returning from a function; jump to the “return address”, presumably set by the **bl** instruction that called it

How function calls work — overview

- memory may be required to execute a function, to remember local variables
 - *activation frame* or *stack frame* of a function call
- since life time of a function call is nested inside that of the caller, they are typically LIFO data structure (i.e., stack)

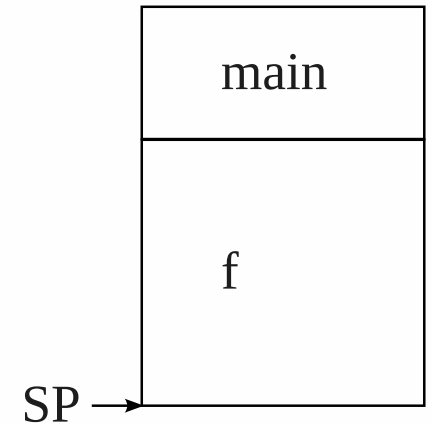
```
int main() {  
    ... f(...) ...  
    ... i(...) ...  
}  
void f(...) {  
    ... g(...) ...  
    ... h(...) ...  
}
```



How function calls work — overview

- memory may be required to execute a function, to remember local variables
 - *activation frame* or *stack frame* of a function call
- since life time of a function call is nested inside that of the caller, they are typically LIFO data structure (i.e., stack)

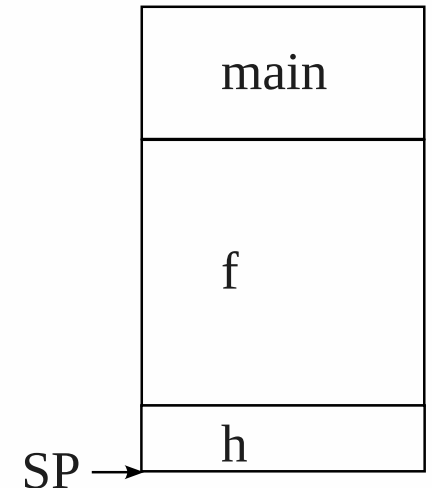
```
int main() {  
    ... f(...) ...  
    ... i(...) ...  
}  
void f(...) {  
    ... g(...) ...  
    ... h(...) ...  
}
```



How function calls work — overview

- memory may be required to execute a function, to remember local variables
 - *activation frame* or *stack frame* of a function call
- since life time of a function call is nested inside that of the caller, they are typically LIFO data structure (i.e., stack)

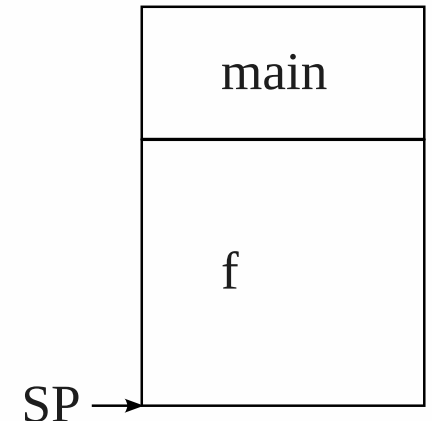
```
int main() {  
    ... f(...) ...  
    ... i(...) ...  
}  
void f(...) {  
    ... g(...) ...  
    ... h(...) ...  
}
```



How function calls work — overview

- memory may be required to execute a function, to remember local variables
 - *activation frame* or *stack frame* of a function call
- since life time of a function call is nested inside that of the caller, they are typically LIFO data structure (i.e., stack)

```
int main() {  
    ... f(...) ...  
    ... i(...) ...  
}  
void f(...) {  
    ... g(...) ...  
    ... h(...) ...  
}
```



How function calls work — overview

- memory may be required to execute a function, to remember local variables
 - *activation frame* or *stack frame* of a function call
- since life time of a function call is nested inside that of the caller, they are typically LIFO data structure (i.e., stack)

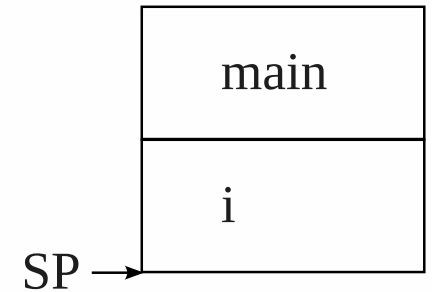
```
int main() {  
    ... f(...) ...  
    ... i(...) ...  
}  
void f(...) {  
    ... g(...) ...  
    ... h(...) ...  
}
```



How function calls work — overview

- memory may be required to execute a function, to remember local variables
 - *activation frame* or *stack frame* of a function call
- since life time of a function call is nested inside that of the caller, they are typically LIFO data structure (i.e., stack)

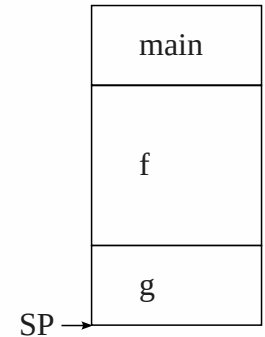
```
int main() {  
    ... f(...) ...  
    ... i(...) ...  
}  
void f(...) {  
    ... g(...) ...  
    ... h(...) ...  
}
```



Note about the stack data structure

- a register, typically called *stack pointer* or *SP*, points to the end of the used part of the stack
- the entire stack is typically a single contiguous region, but it doesn't have to
- if it is, allocating an activation frame from the stack just entails bumping a stack pointer

```
int main() {  
    ... f(...) ...  
    ... i(...) ...  
}  
void f(...) {  
    ... g(...) ...  
    ... h(...) ...  
}
```



Looking into a function call

- rules must exist for function calls to work
 1. where are arguments
 2. where to jump after finished (*return address*)
 3. where to use if the function wants to use memory
 4. which registers must be preserved across a call
 5. where to pass the return value
- they are variously called
 - *calling convention, register usage convention, or*
 - *Application Binary Interface (ABI)*

Application Binary Interface (ABI) of ARM64

Omitting details you should learn through experiments (and which may be language-dependent),

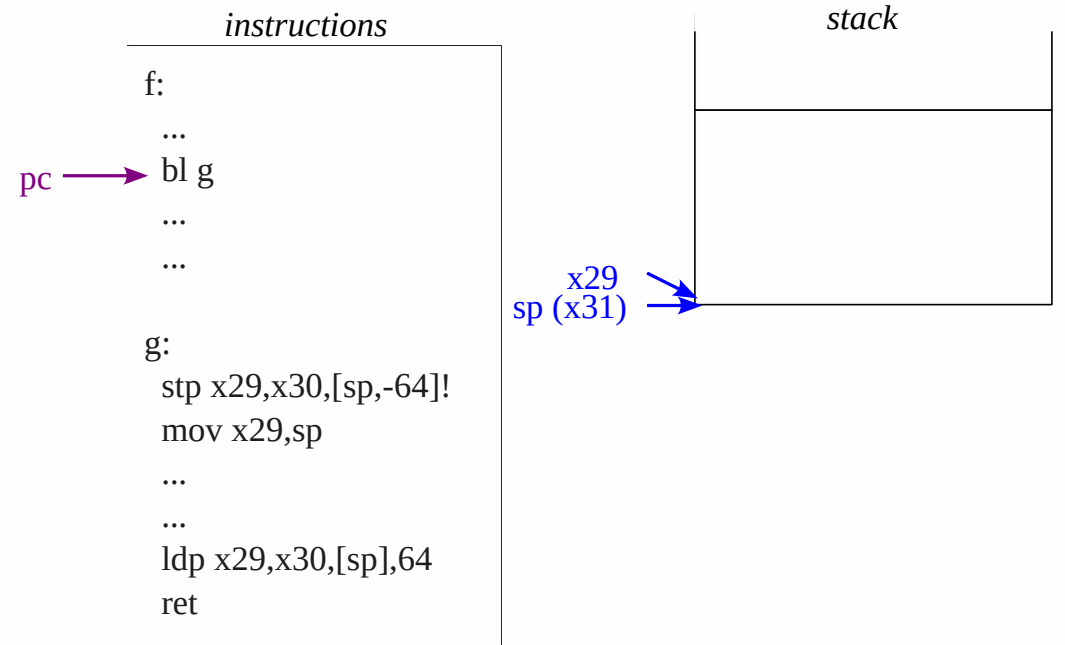
- upon entry:
 - arguments: `x0`, `x1`, ... (integers), `d0`, `d1`, ... (floats)
 - return address: `x30`
 - `sp` points to the end of the stack, below which the callee can use for its purpose

Application Binary Interface (ABI) of ARM64

- upon return:
 - return value: `x0` (integer), `d0` (float)
 - `x19` - `x28` must hold the same values as those upon entry
(callee-save registers)
 - the following registers must also be preserved
 - `x29` (*frame pointer*),
 - `x30` (*link register or return address register*),
 - `sp` (*stack pointer*)
- `x0` - `x15` can be destroyed by the callee *(caller-save registers)*

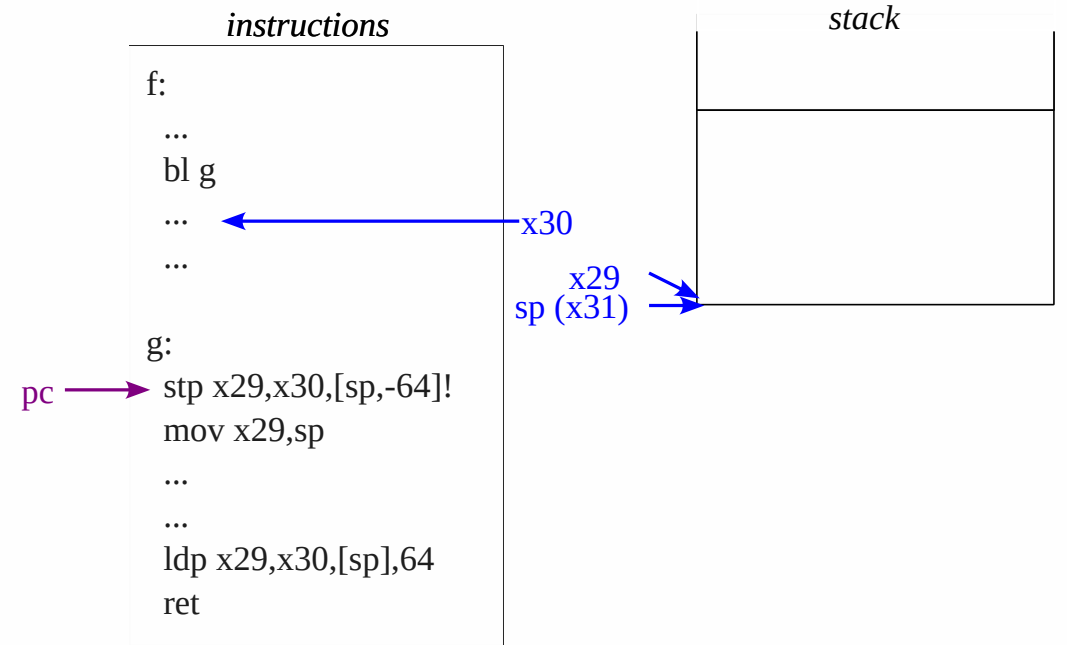
A function call in action

- say `f` is calling `g`
- `f` puts arguments at right places and executes `bl g` instruction



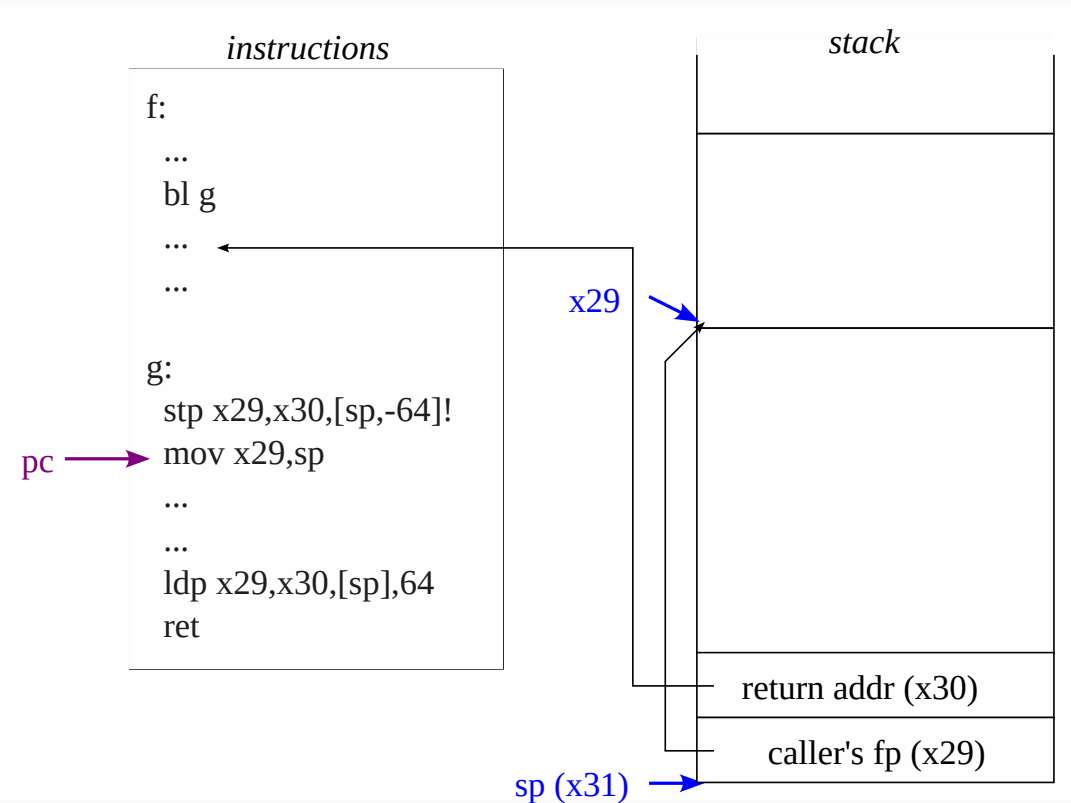
A function call in action

- x30 now holds the **return address** (address of the instruction immediately following `bl g`)
- if `g` calls another function, it first extends the stack and saves x30 and x29 (typically by `stp x29, x30, [sp, #-??]`)



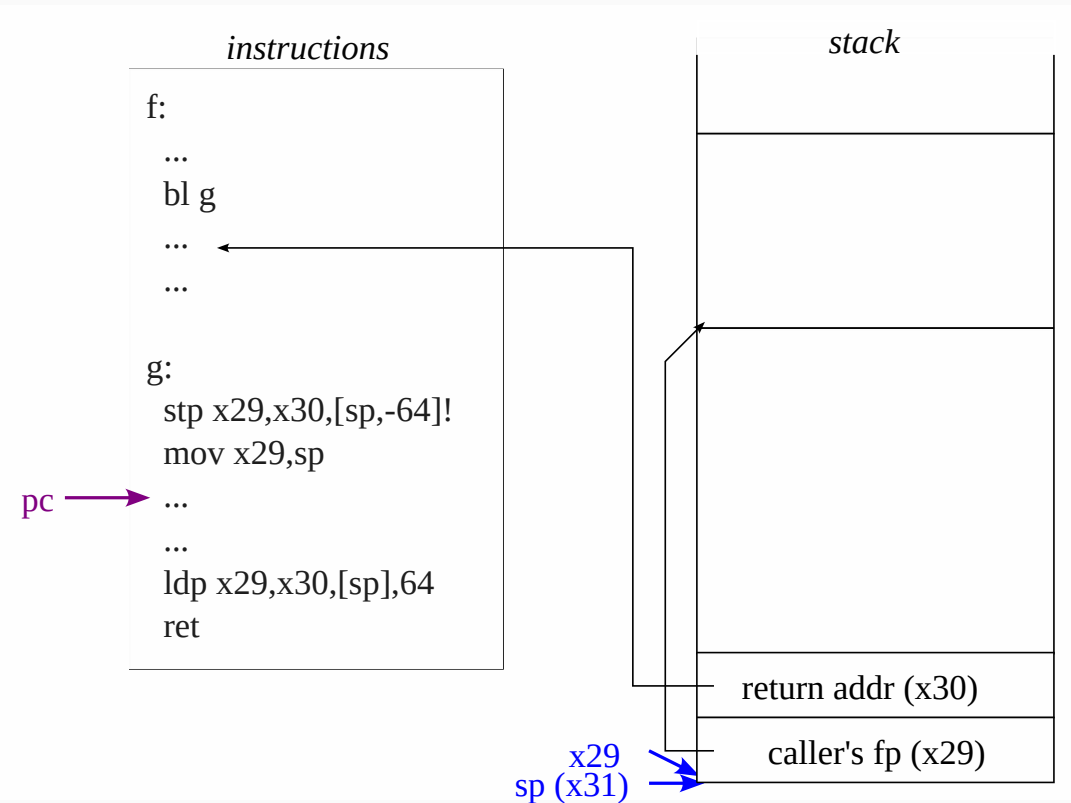
A function call in action

- it then sets up *frame pointer* (x29), typically to the same as sp (more on this later)



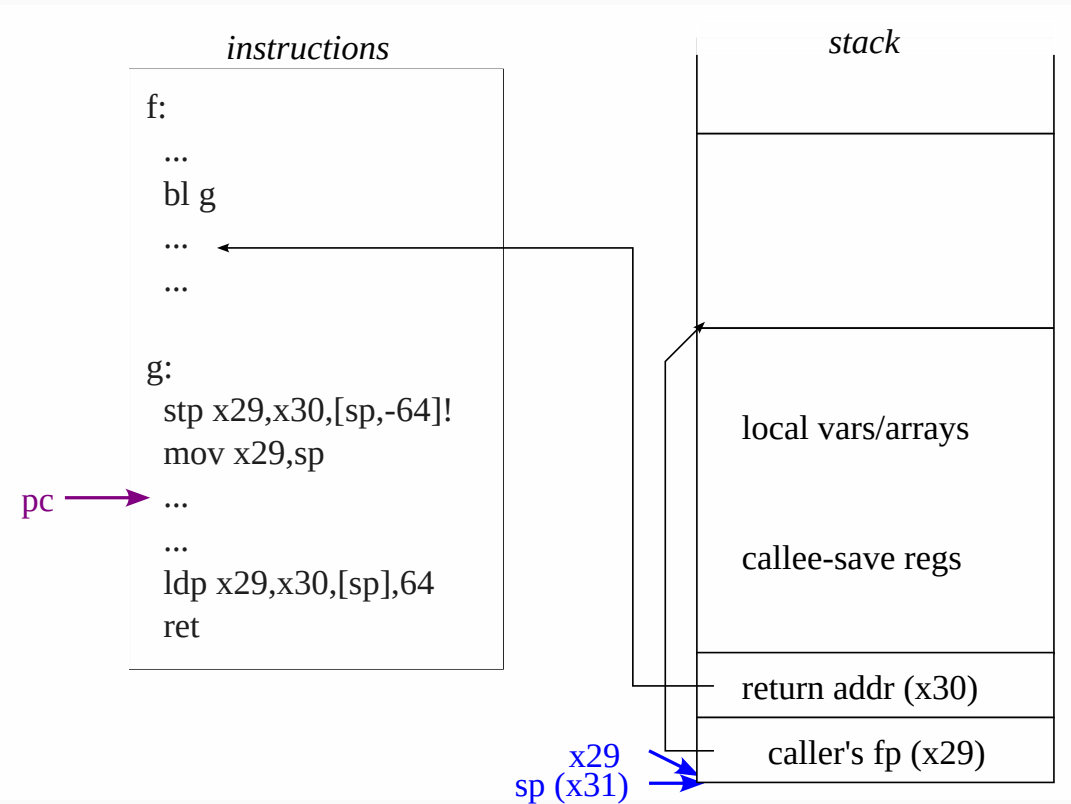
A function call in action

- if it uses some callee-save registers, it saves them on stack



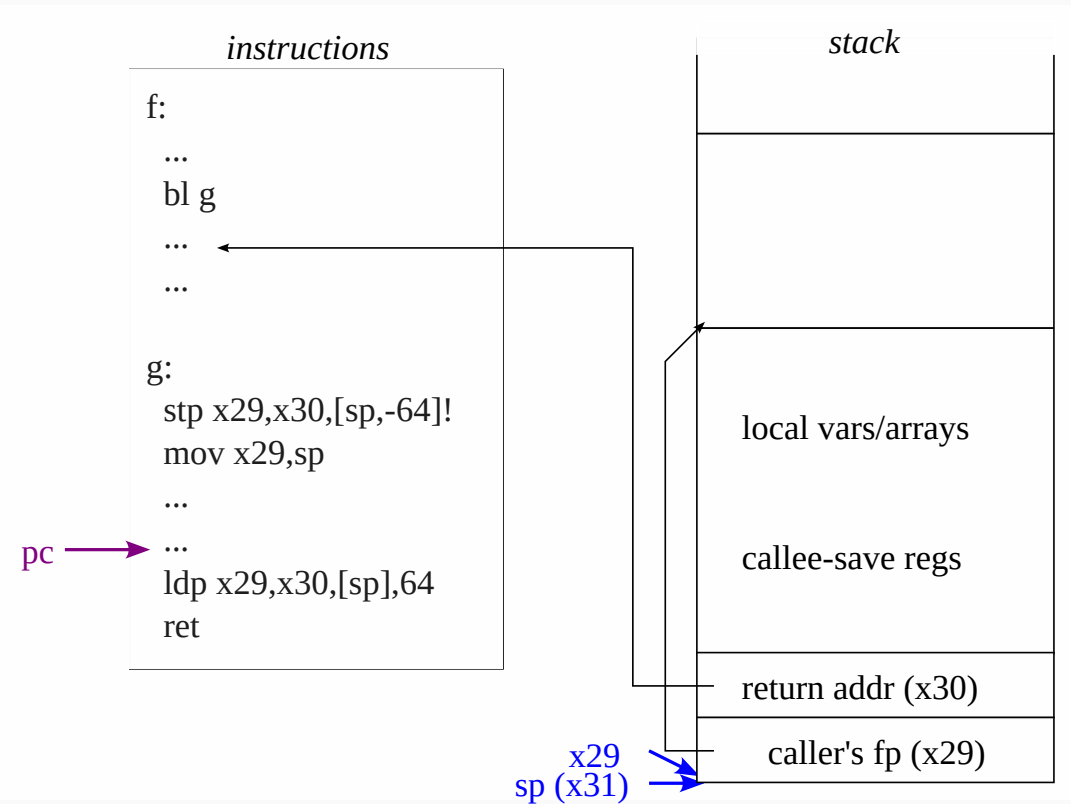
A function call in action

- `g` is now really ready to execute ...



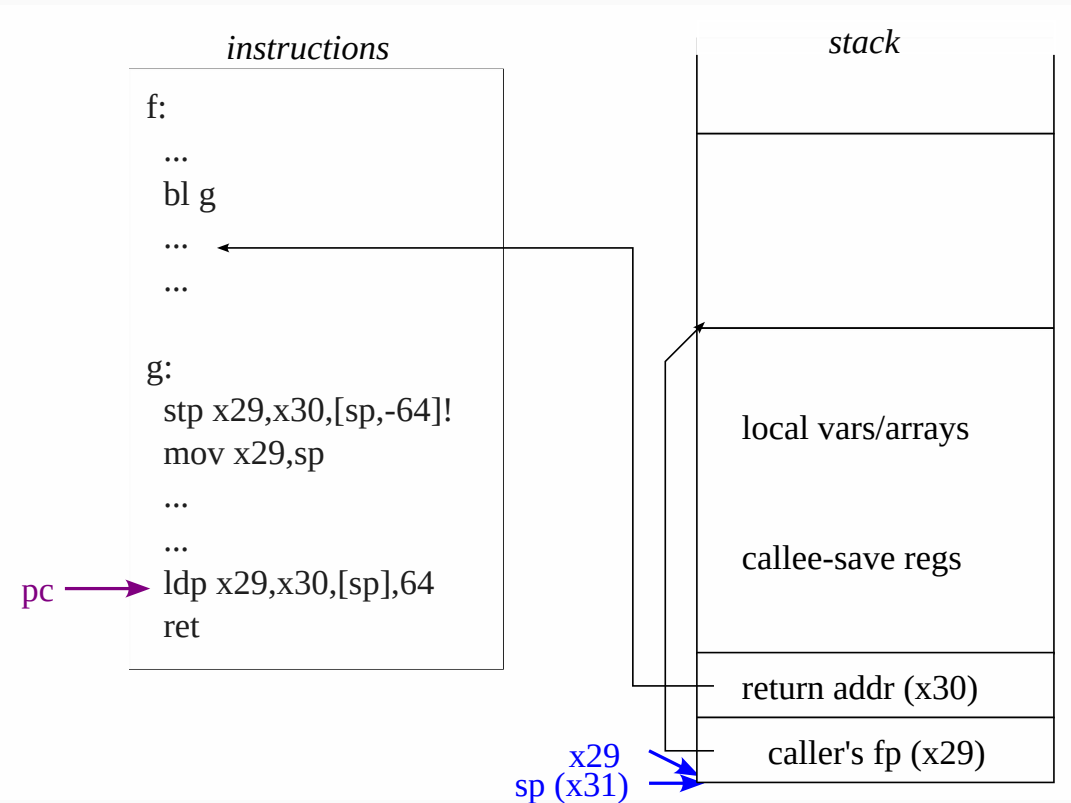
A function call in action

- ... before it returns,
- it restores callee-save registers, if any



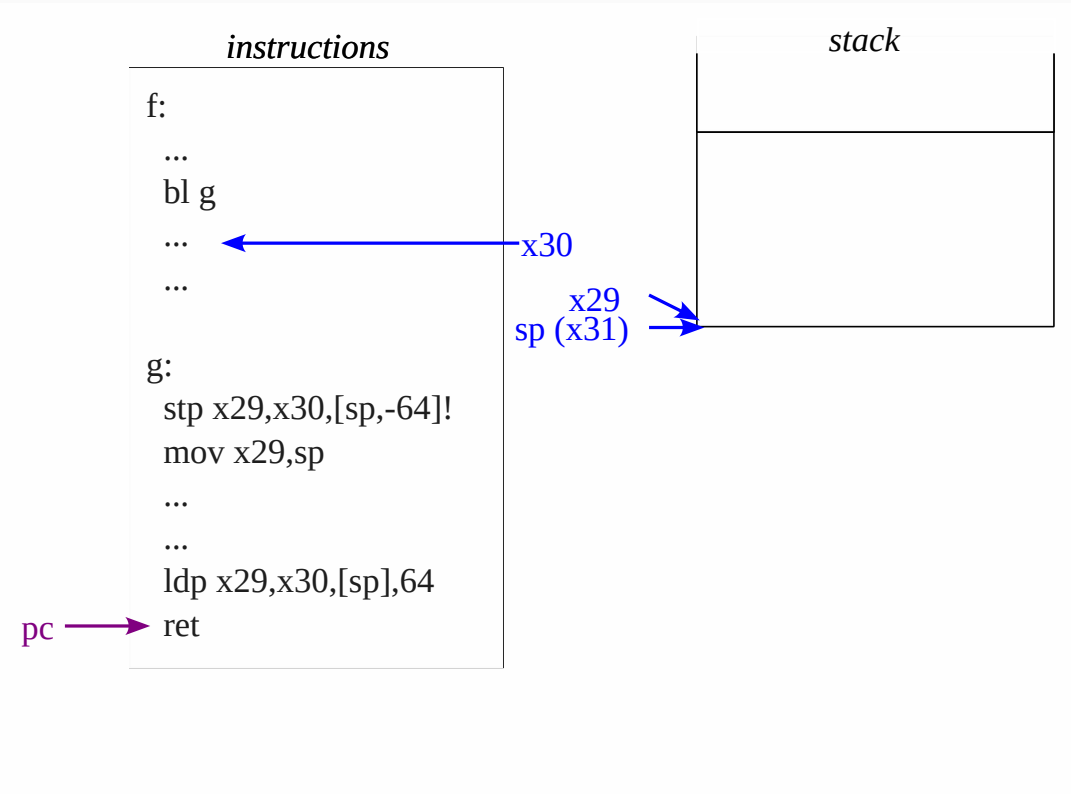
A function call in action

- it then restores `x29` and `x30` and shrinks the stack (typically by `ldp x29,x30,[sp],64`, `[sp],??`)



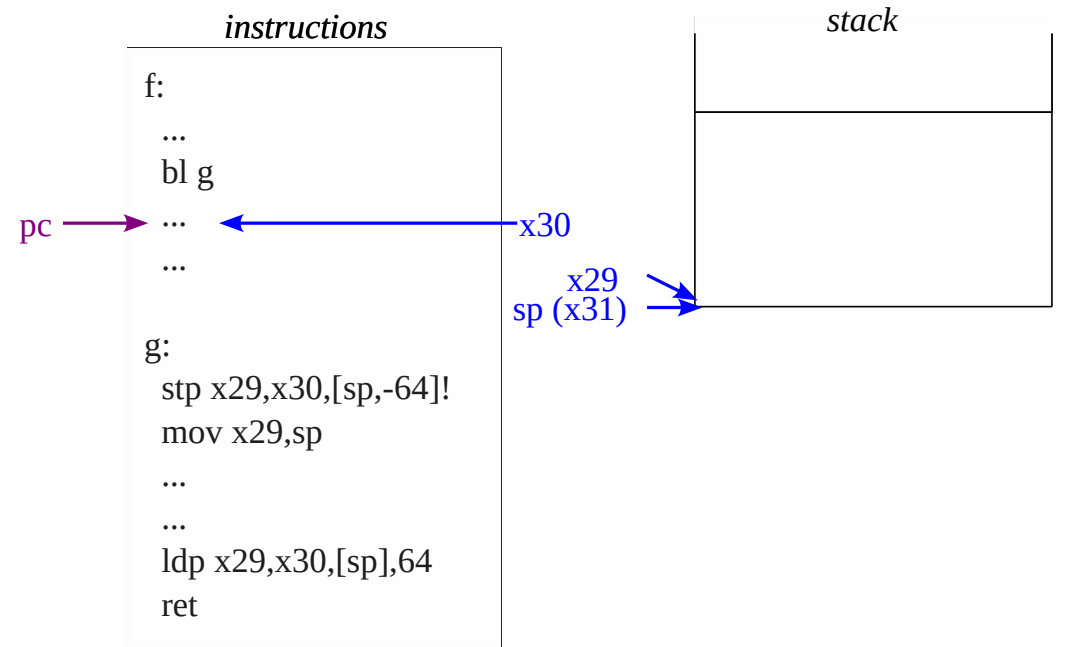
A function call in action

- it finally executes `ret`, which jumps to the address in `x30`, which should hold the return address just restored



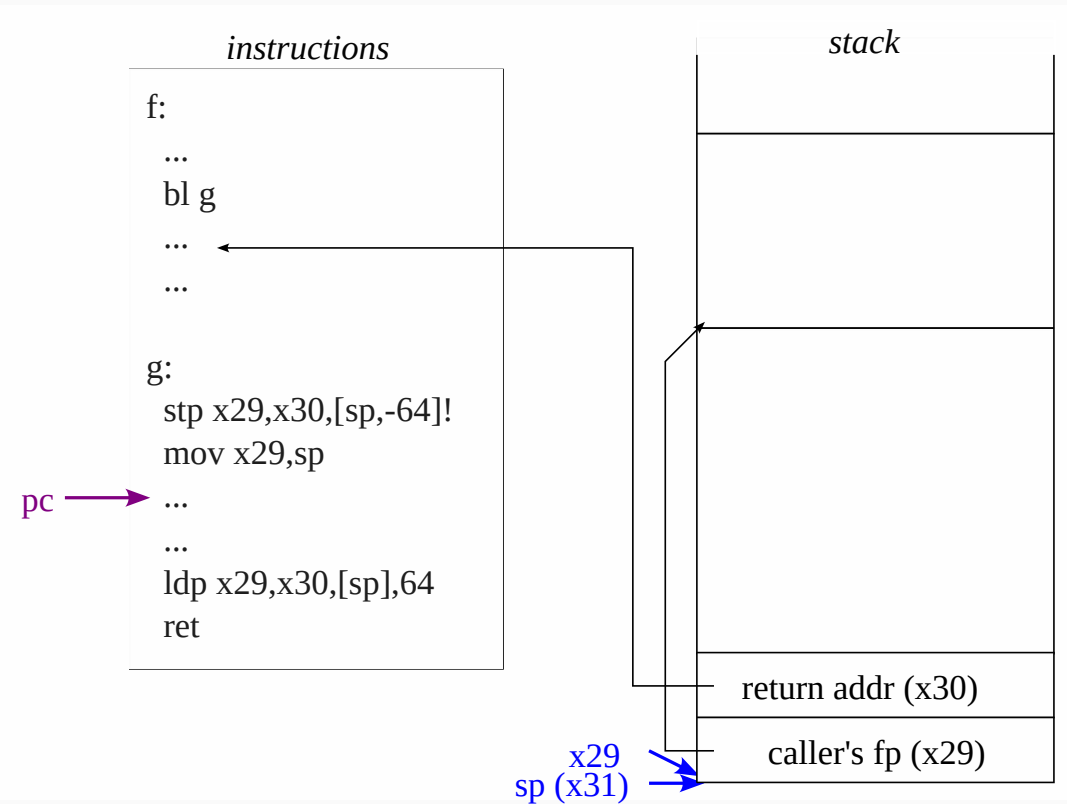
A function call in action

- execution of `f` continues, from the instruction immediately following the `bl`



Note: stack pointer (SP) and frame pointer (FP)

- they are usually the same
- more generally,
 - SP may change while executing a function (e.g., when dynamically allocating memory from stack with `alloca`)
 - FP is fixed and always points to where the caller's FP is stored



Things to learn in the exercise

1. **calling convention / ABI:** how a function call works
2. **control flow:** how conditionals and loops are implemented
3. **data representation:** how various data types (ints, floats, structs, pointers, arrays) are represented