CS 31: Intro to Systems Functions and the Stack

Kevin Webb

Swarthmore College

October 4, 2022

Reading Quiz

Overview

- Stack data structure, applied to memory
- Behavior of function calls
- Storage of function data, at assembly level

“A” Stack

- A stack is a basic data structure
  - Last in, first out behavior (LIFO)
  - Two operations
    - Push (add item to top of stack)
    - Pop (remove item from top of stack)

Pop (remove and return item)

Push (add data item)

Newest data

Oldest data

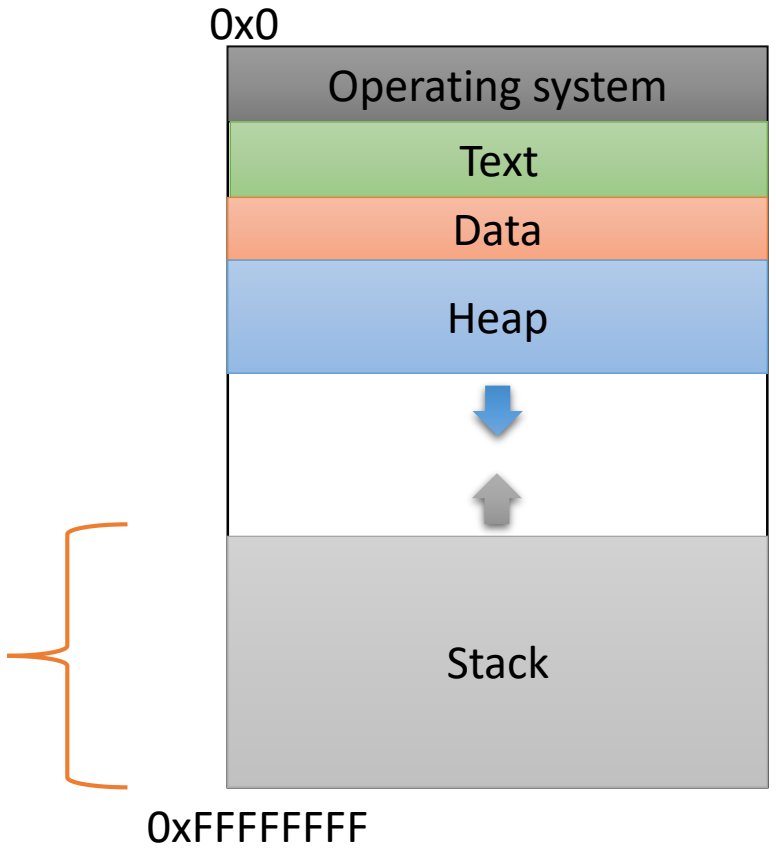
“The” Stack

- Apply stack data structure to memory
  - Store local (automatic) variables
  - Maintain state for functions (e.g., where to return)

- Organized into units called *frames*
  - One frame represents all of the information for one function.
  - Sometimes called *activation records*

Memory Model

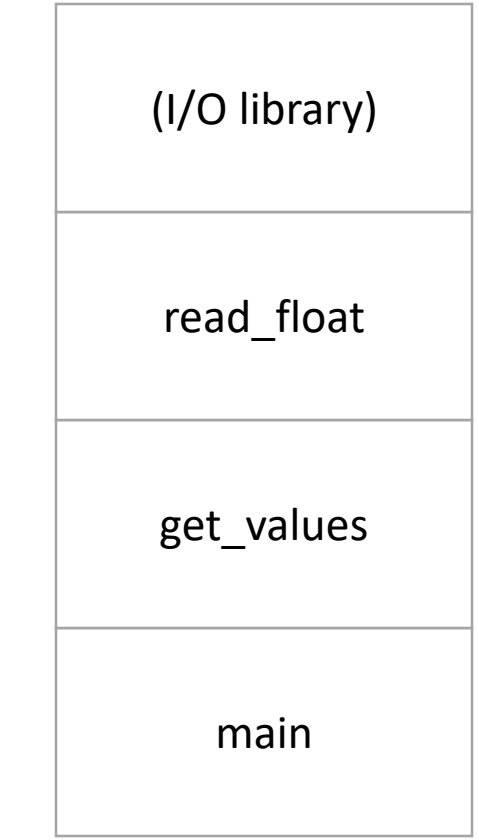
- Starts at the highest memory addresses, grows into lower addresses.



Stack Frames

- As functions get called, new frames added to stack.

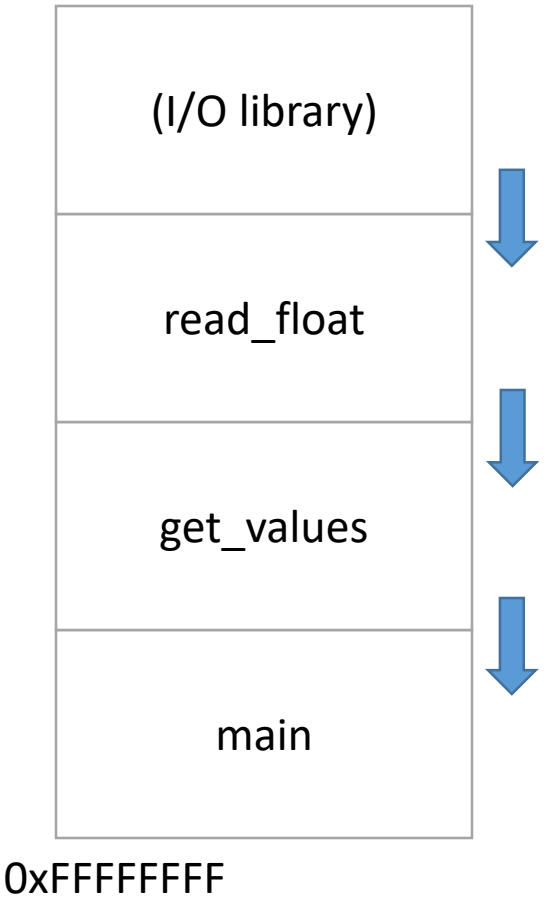
- Example: Lab 4
  - main calls get_values()
  - get_values calls read_float()
  - read_float calls I/O library



Stack Frames

- As functions return, frames removed from stack.

- Example: Lab 4
  - I/O library returns to read_float
  - read_float returns to get_values
  - get_values returns to main



All of this stack growing/shrinking happens automatically (from the programmer's perspective).

What is responsible for creating and removing stack frames?

A. The user

B. The compiler

C. C library code

D. The operating system

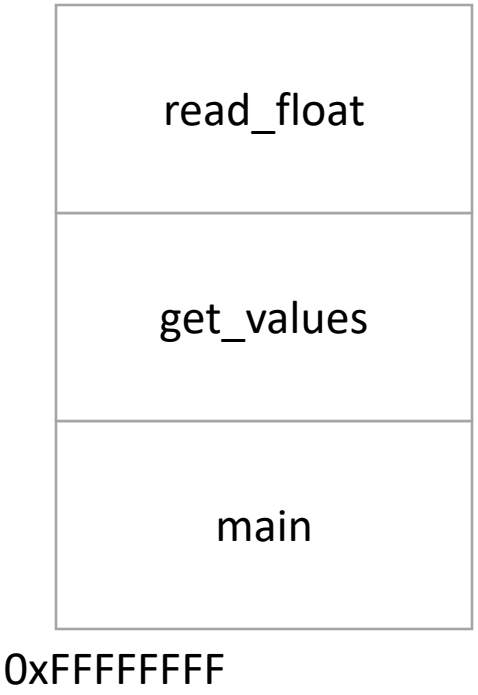
E. Something / someone else

Insight: EVERY function needs a stack frame. Creating / destroying a stack frame is a (mostly) generic procedure.

Stack Frame Contents

- What needs to be stored in a stack frame?
  - Alternatively: What *must* a function know / access?

- Local variables



Local Variables

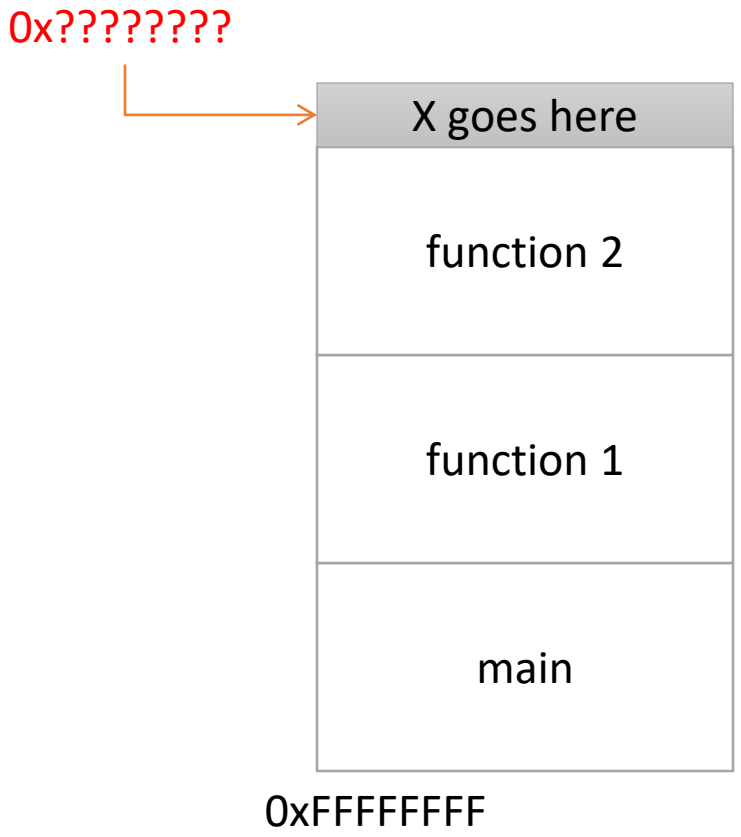
If the programmer says:

```
int x = 0;
```

Where should x be stored?

(Recall basic stack data structure)

Which memory address is that?



How should we determine the address to use for storing a new local variable?

A. The programmer specifies the variable location.

B. The CPU stores the location of the current stack frame.

C. The operating system keeps track of the top of the stack.

D. The compiler knows / determines where the local data for each function will be as it generates code.

E. The address is determined some other way.

Program Characteristics

- Compile time (static)
  - Information that is known by analyzing your program
  - Independent of the machine and inputs

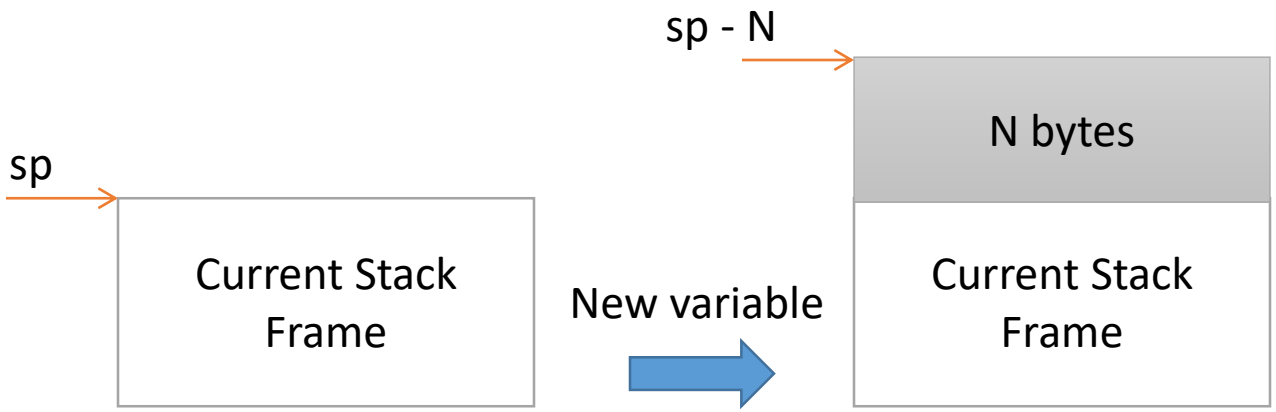
- Run time (dynamic)
  - Information that isn’t known until program is running
  - Depends on machine characteristics and user input

The Compiler Can…

- Perform type checking.
- Determine how much space you need on the stack to store local variables.
- Insert assembly instructions for you to set up the stack for function calls.
  - Create stack frames on function call
  - Restore stack to previous state on function return

Local Variables

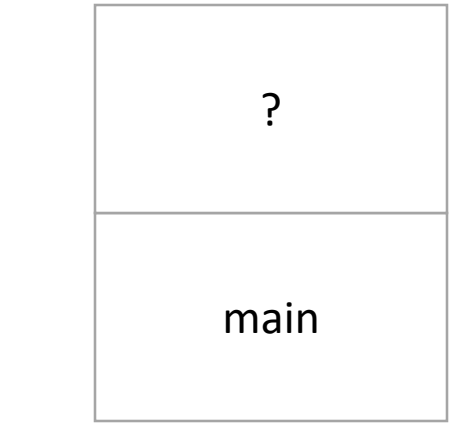
- Compiler can allocate N bytes on the stack by subtracting N from the “stack pointer”: sp



The Compiler Can't…

- Predict user input.

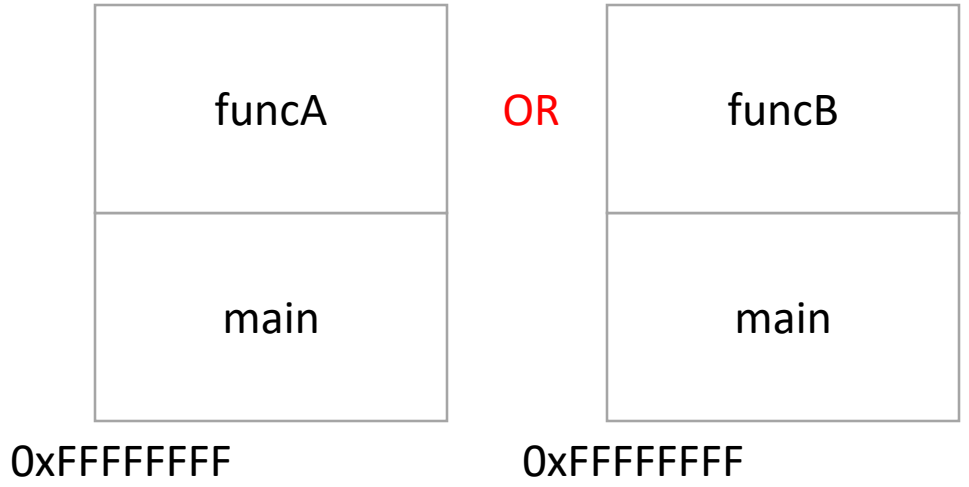
```
int main() {
  int decision = [read user input];
  if (decision > 5) {
    funcA();
  } else {
    funcB();
  }
}
```



The Compiler Can't…

- Predict user input.

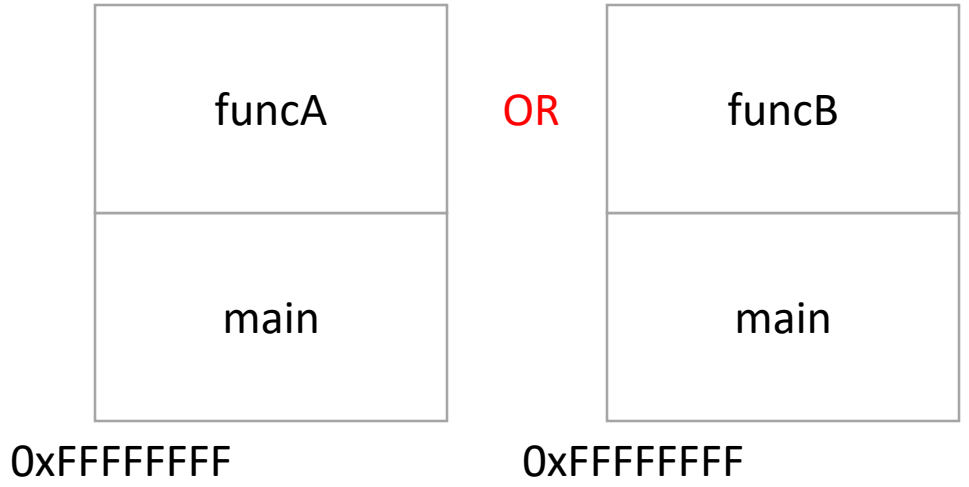
```
int main() {
   int decision = [read user input];
   if (decision > 5) {
      funcA();
   } else {
      funcB();
   }
}
```



The Compiler Can't…

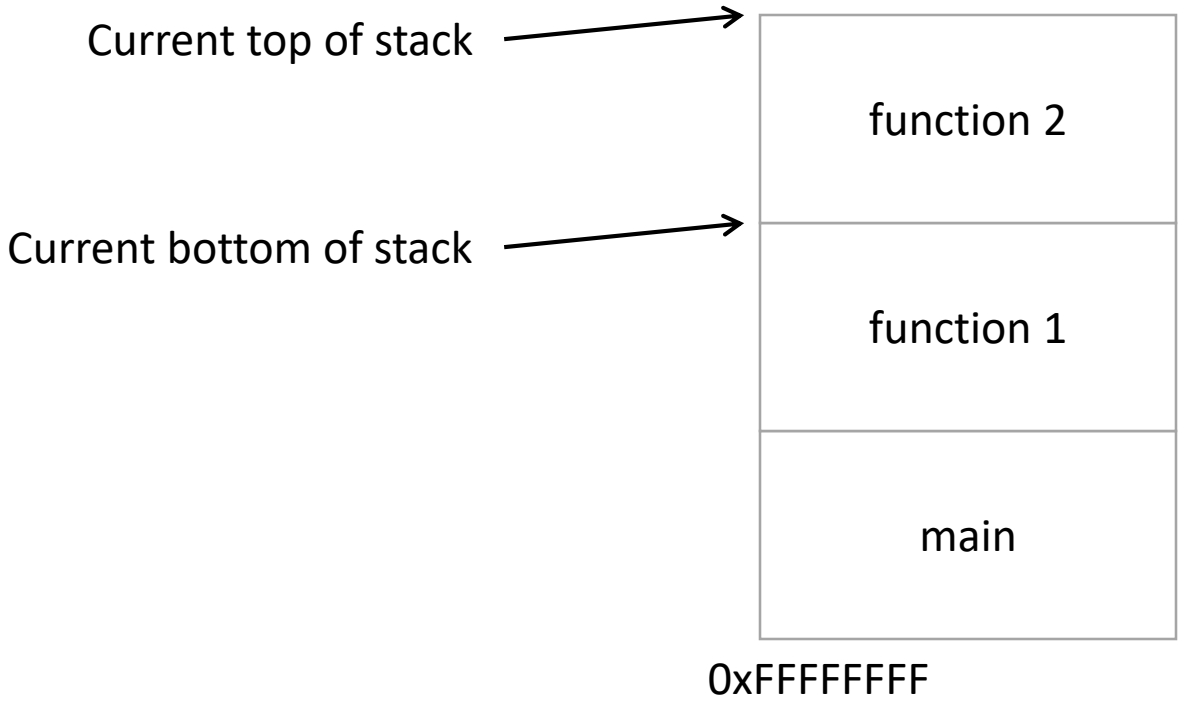
- Predict user input.
- Can't assume a function will always be at a certain address on the stack.

Alternative: create stack frames relative to the current (dynamic) state of the stack.



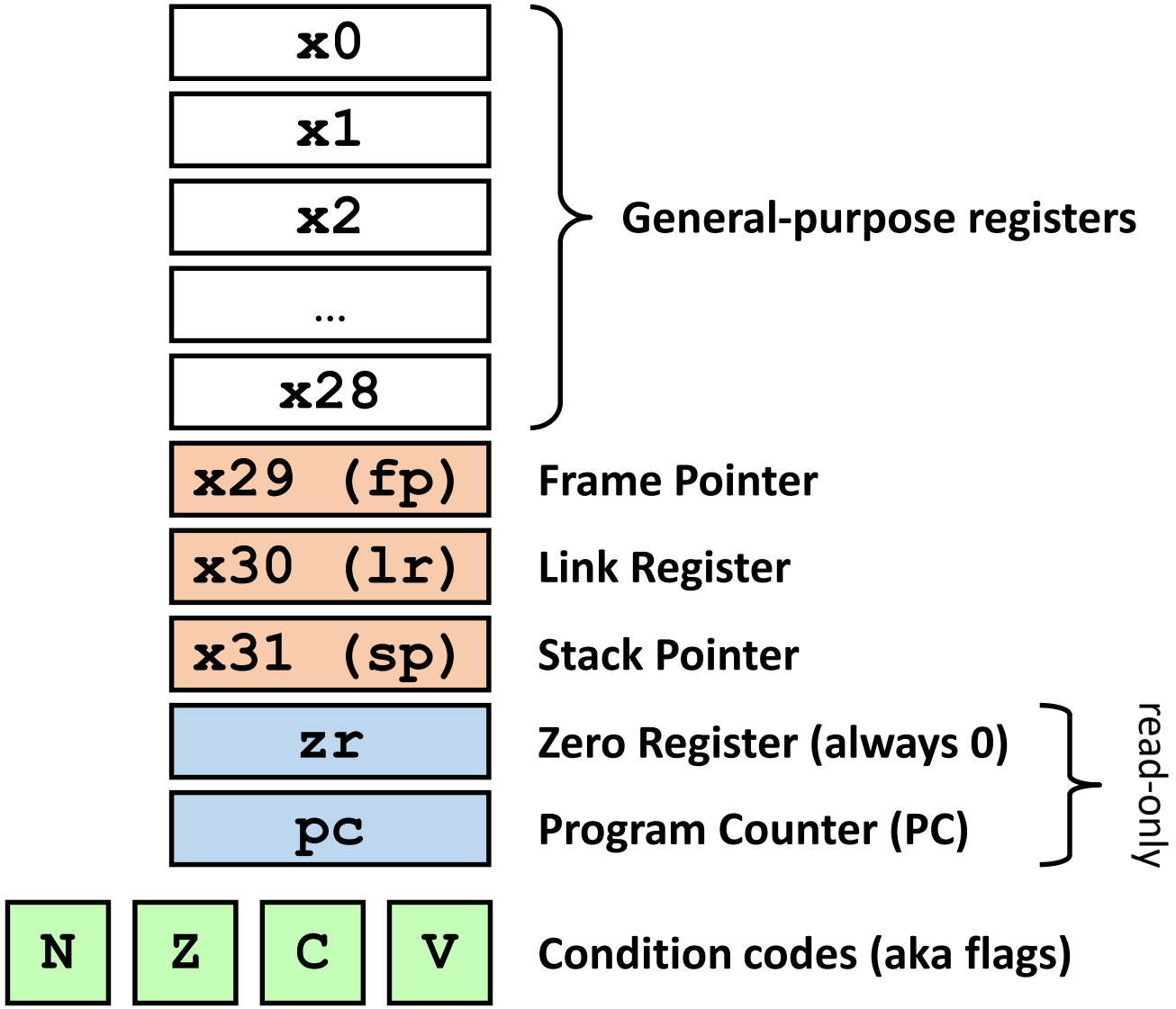
Stack Frame Location

- Where in memory is the current stack frame?



Recall: ARM64 Register Conventions

- Even though x0 - x31 are general-purpose, some are unofficially reserved for specific purposes…
- x29 - x31 used to keep track of function / stack information (more on this later)

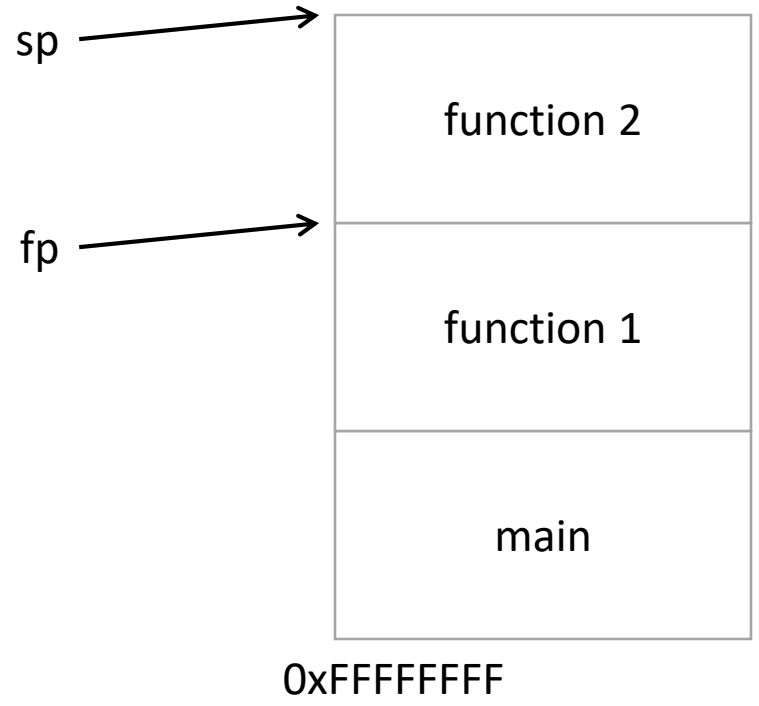


Stack Frame Location

- Where in memory is the current stack frame?

- Maintain invariant:
  - The current function's stack frame is always between the addresses stored in sp and fp

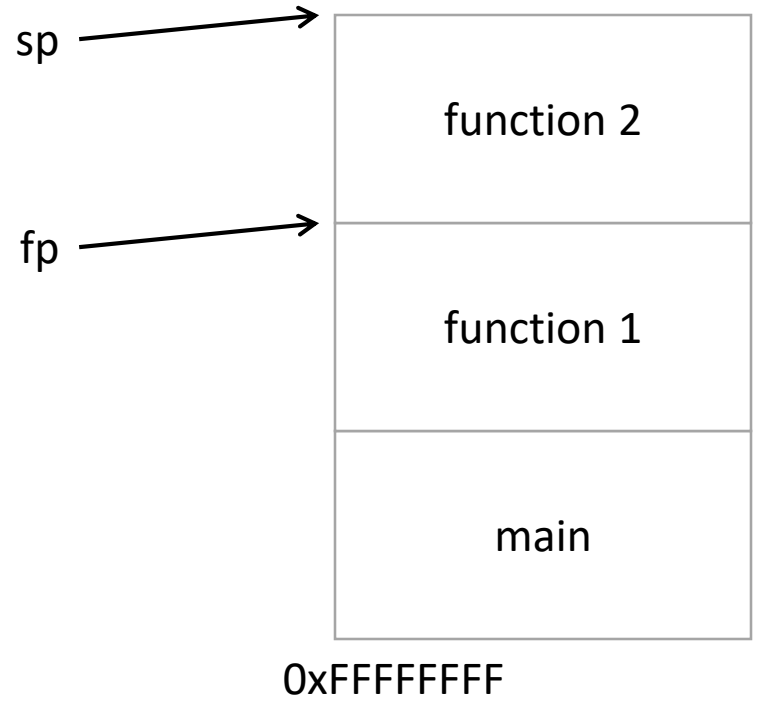
- sp: stack pointer
- fp: frame pointer (base pointer)



Stack Frame Location

- Compiler ensures that this invariant holds.
  - We'll see how a bit later.

- This is why all local variables we've seen in assembly are relative to `fp` or `sp`!



How would we implement pushing x to the top of the stack in ARM64?

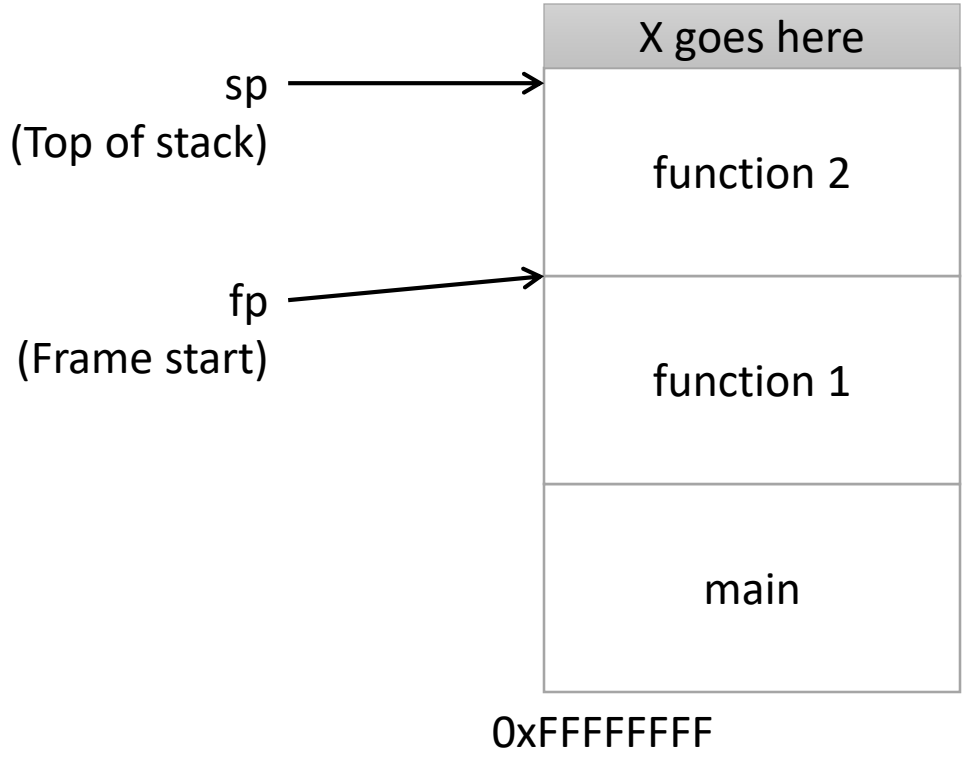
A. Increment sp
   Store x at [sp]

B. Store x at [sp]
   Increment sp

C. Decrement sp
   Store x at [sp]

D. Store x at [sp]
   Decrement sp

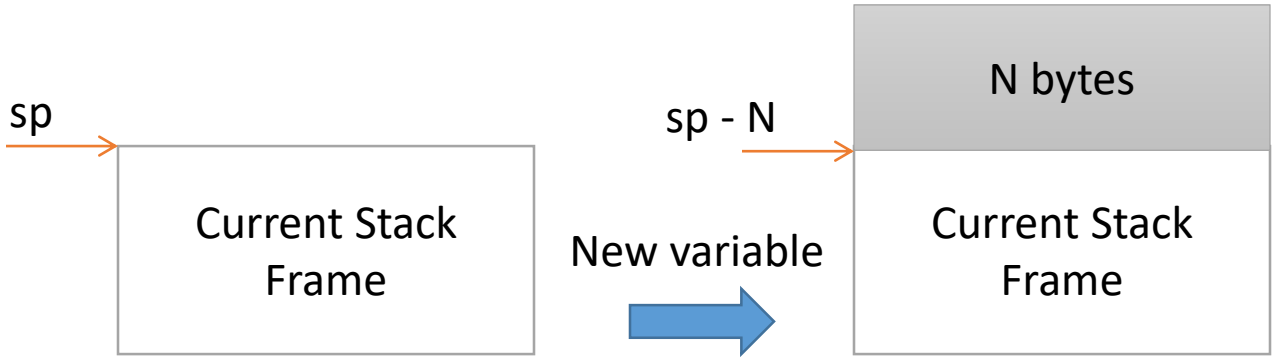
E. Copy sp to fp
   Store x at fp



Local Variables

- More generally, we can make space on the stack for N bytes by subtracting N from sp

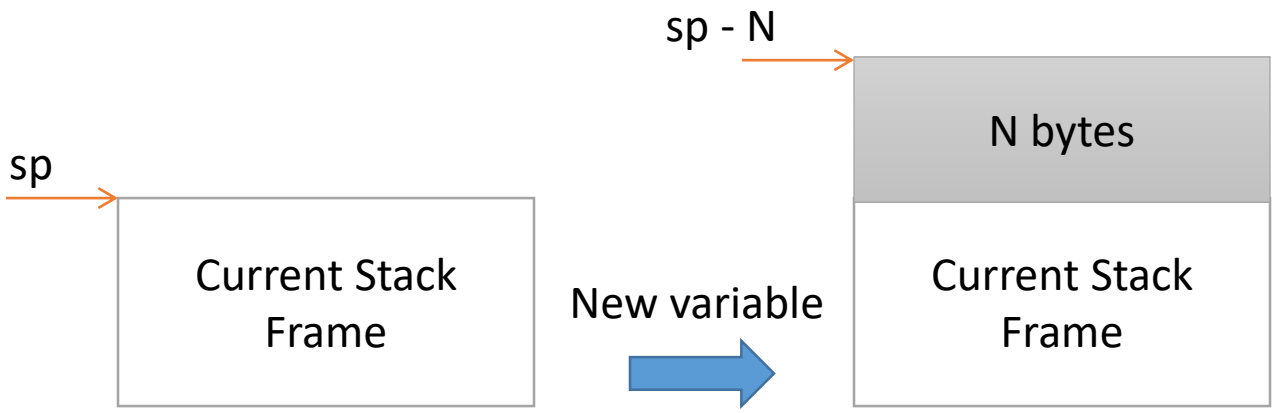
- NOTE: for ARM64, the sp register must hold a multiple of 16 (it's said to be "16-byte aligned")



Local Variables

- More generally, we can make space on the stack for N bytes by subtracting N from sp

- When we're done, free the space by adding N back to sp

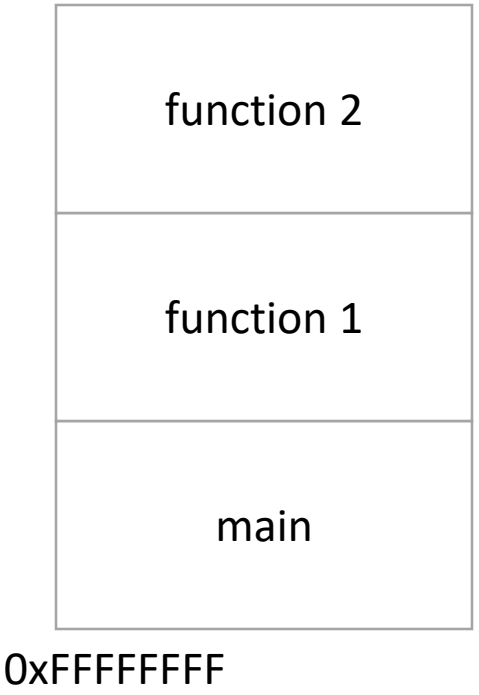


Stack Frame Contents

- What needs to be stored in a stack frame?
  - Alternatively: What *must* a function know?

- Local variables
- Previous stack frame base address
- Function arguments
- Return value
- Return address

- Saved registers
- Spilled temporaries

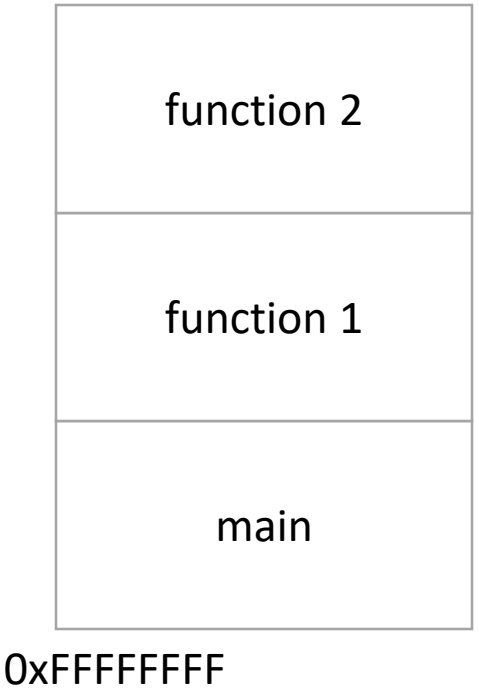


Stack Frame Contents

- What needs to be stored in a stack frame?
  - Alternatively: What *must* a function know?

- Local variables
- Previous stack frame base address
- Function arguments
- Return value
- Return address

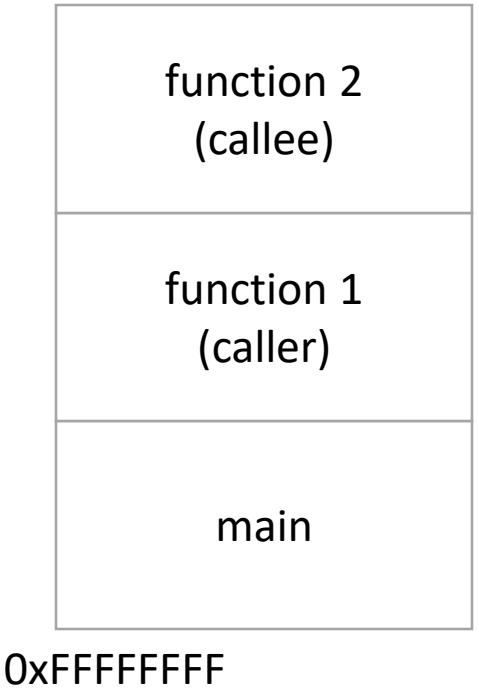
- Saved registers
- Spilled temporaries



Stack Frame Relationships

- If function 1 calls function 2:
  - function 1 is the caller
  - function 2 is the callee

- With respect to main:
  - main is the caller
  - function 1 is the callee



Where should we store all this stuff?

Previous stack frame base address
Function arguments
Return value
Return address

A. In registers
B. On the heap
C. In the caller’s stack frame
D. In the callee’s stack frame
E. Somewhere else

Calling Convention

- You could store this stuff wherever you want!
  - The hardware does NOT care.
  - What matters: everyone agrees on where to find the necessary data.

- Calling convention: agreed upon system for exchanging data between caller and callee

ARM64 Calling Convention

- When possible, keep values in registers
  - ARM has lots of registers available
  - Accessing registers is faster than memory (stack)

- When a function (caller) calls another (callee):
  - The caller saves the frame pointer (fp / x29) and return address (lr / x30) on the stack.
  - The caller passes the first eight arguments in registers x0 - x7
    (If more arguments are needed, they go on the stack)
  - If the callee produces a result, it returns it via register x0

ARM64 Calling Convention

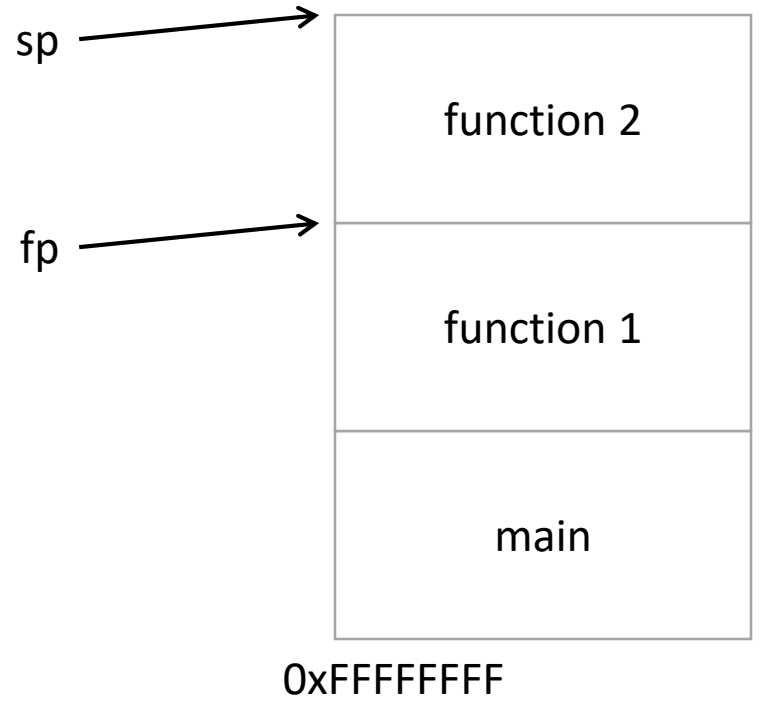
- When possible, keep values in registers
  - ARM has lots of registers available
  - Accessing registers is faster than memory (stack)

- When a function (caller) calls another (callee):
  - **The caller saves the frame pointer (fp / x29) and return address (lr / x30) on the stack.**
  - The caller passes the first eight arguments in registers x0 - x7
    (If more arguments are needed, they go on the stack)
  - If the callee produces a result, it returns it via register x0

Stack Frame Location

- Compiler ensures that this invariant holds.
  - We'll see how a bit later.

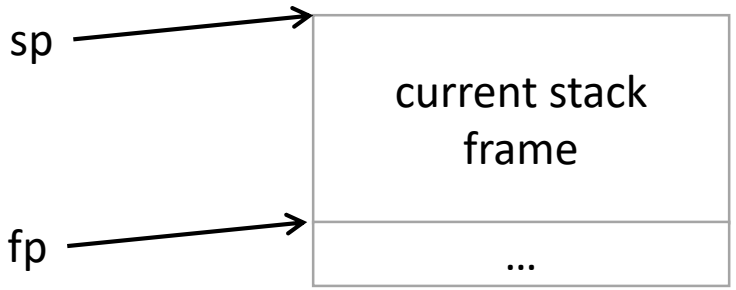
- This is why all local variables we've seen in assembly are relative to `fp` or `sp`!



Two important pieces of state…

1. Keeping track of top/bottom of stack

- Dedicate CPU registers for stack bookkeeping
  - sp (stack pointer, x31):
    Top of current stack frame
  - fp (frame pointer, x29):
    Base of current stack frame



ARM64 Calling Convention

- When possible, keep values in registers
  - ARM has lots of registers available
  - Accessing registers is faster than memory (stack)

- When a function (caller) calls another (callee):
  - **The caller saves the frame pointer (fp / x29) and return address (lr / x30) on the stack.**
  - The caller passes the first eight arguments in registers x0 - x7
    (If more arguments are needed, they go on the stack)
  - If the callee produces a result, it returns it via register x0

Return Address

- When you call a function, you expect that when the function returns, you proceed with the next statement after the function call.

- To make this behavior happen, we must store at address of the next instruction before branching (changing the PC) to the callee.

- ARM64 calls this a "branch with link"

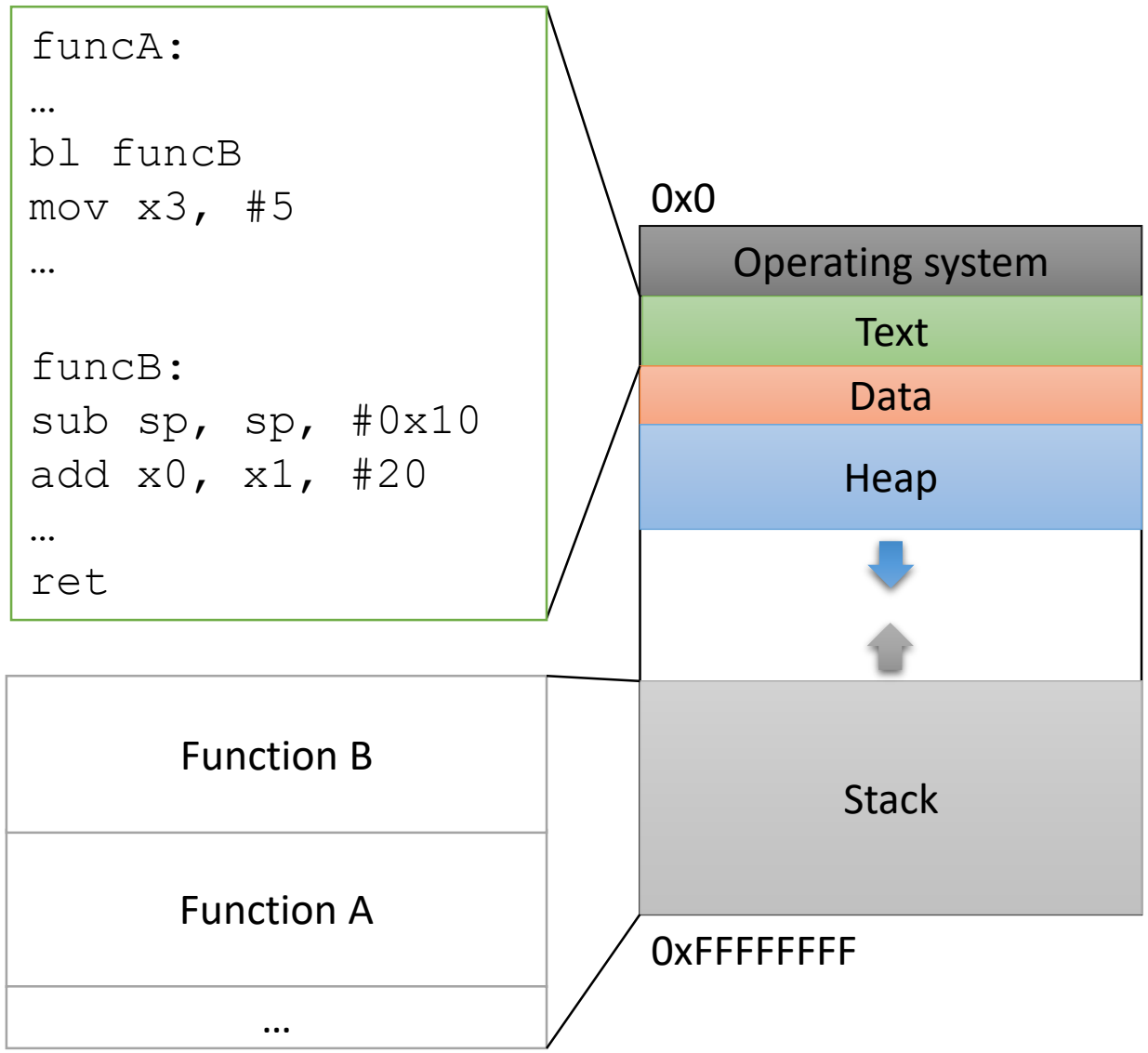
Branch with link (`bl`)

1. Save PC+4 (address of *next* instruction after the `bl`) into x30 (LR)

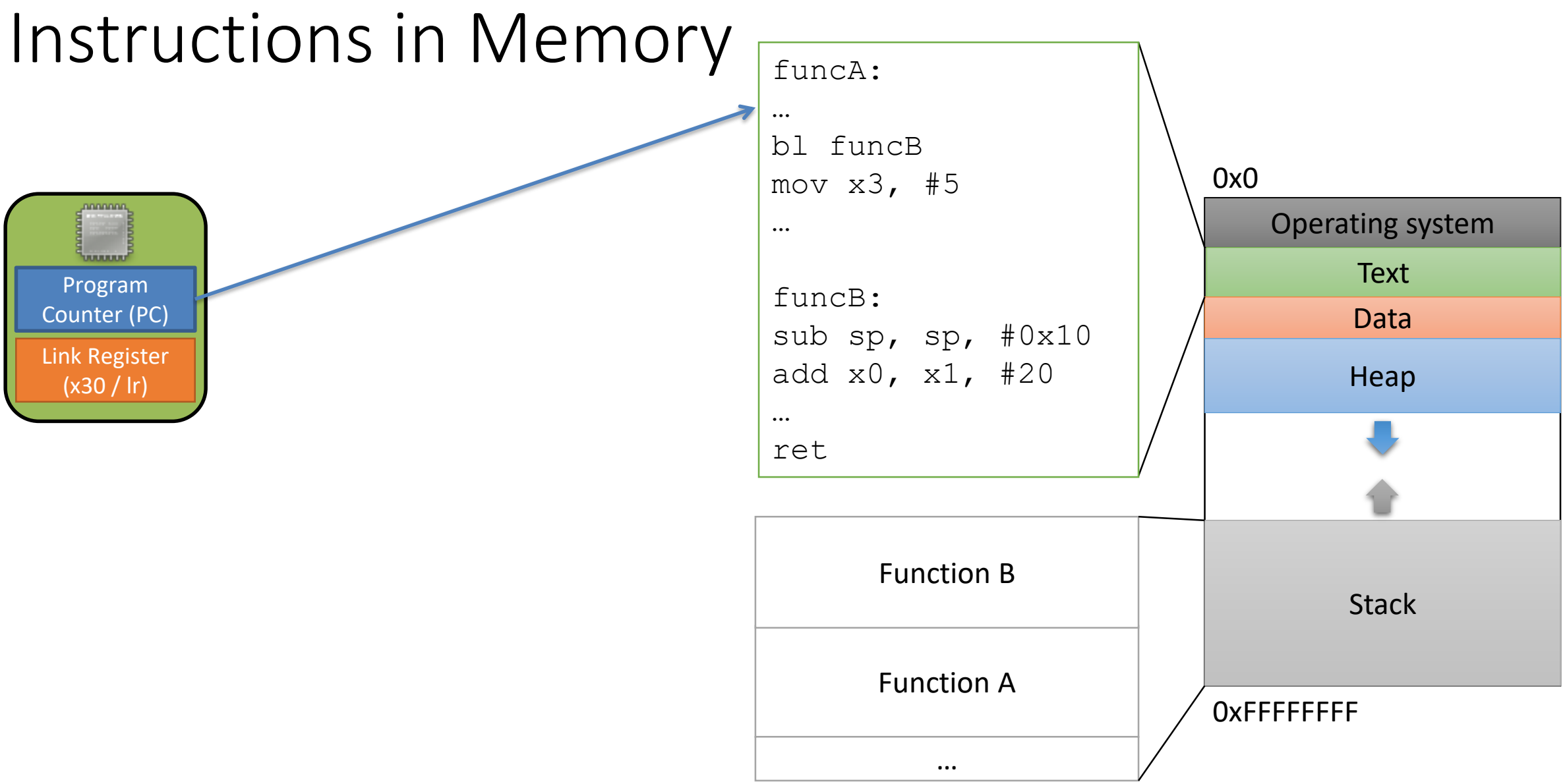
2. Unconditionally change the PC to the start of the callee's code

- Later, when callee executes a ret instruction, set PC back to the value in the link register (x30 / LR)

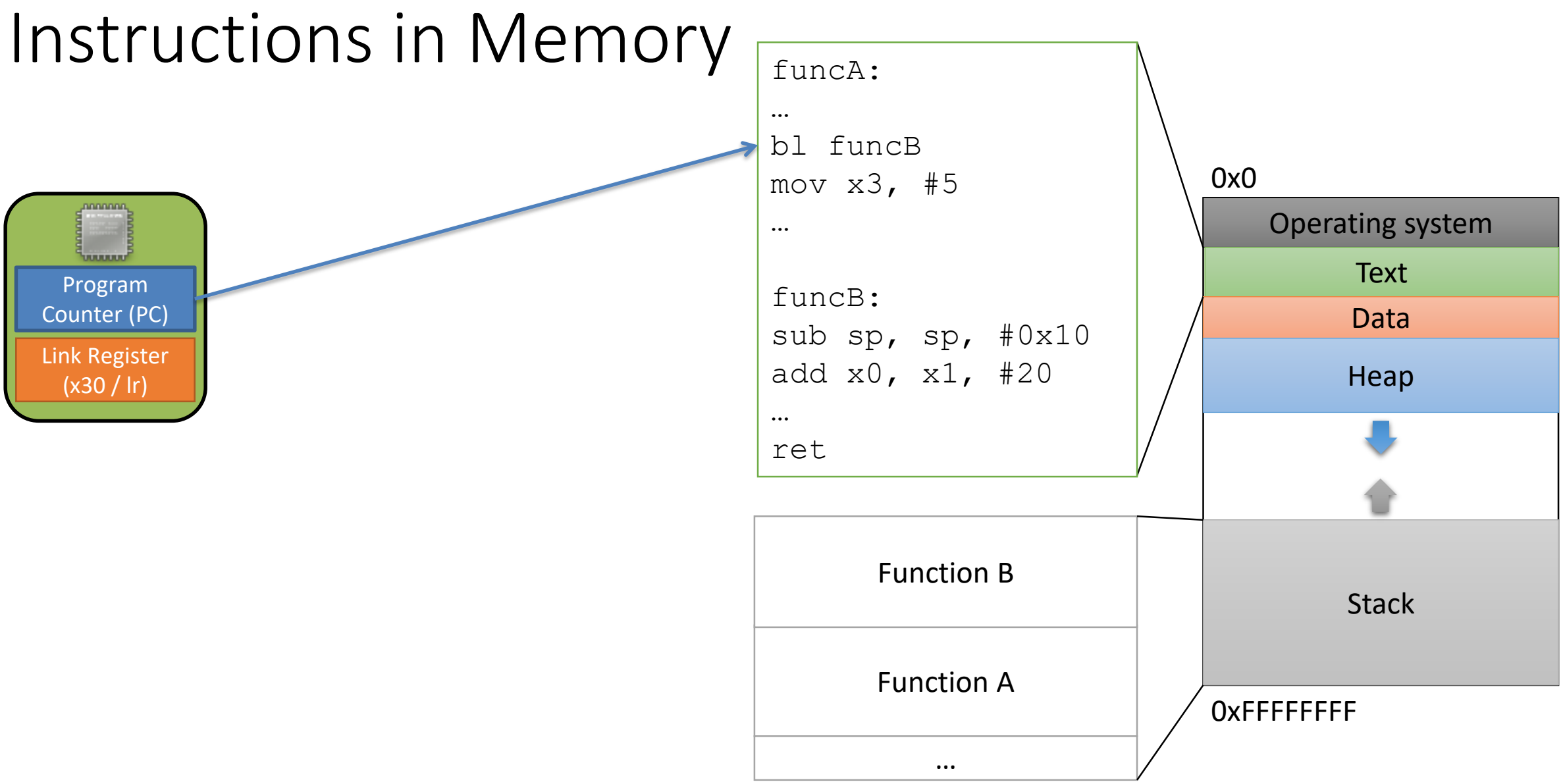
Instructions in Memory



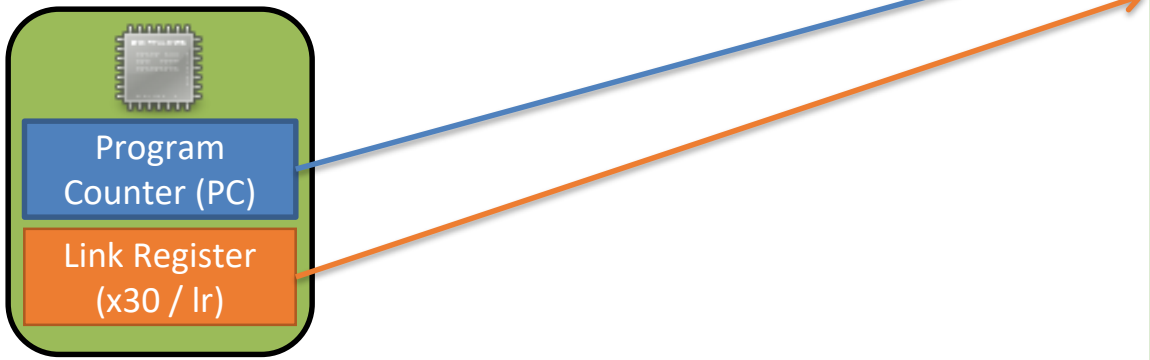
Instructions in Memory



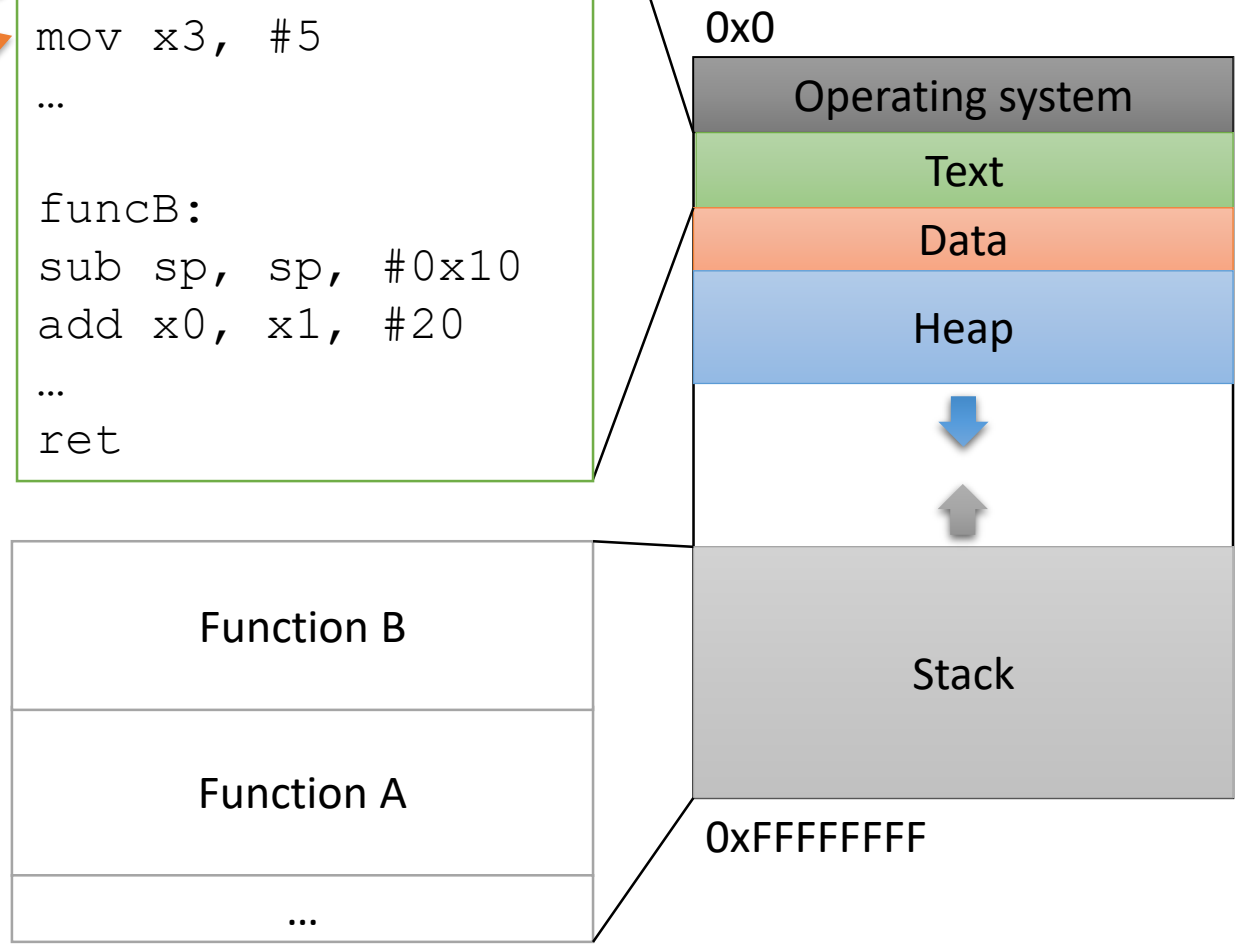
Instructions in Memory



Instructions in Memory



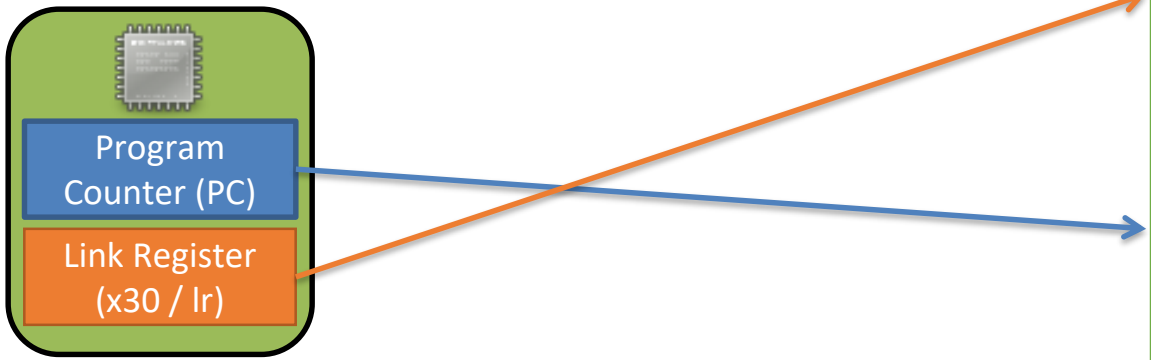

```
funcA:
…
bl funcB
mov x3, #5
…

funcB:
sub sp, sp, #0x10
add x0, x1, #20
…
ret
```



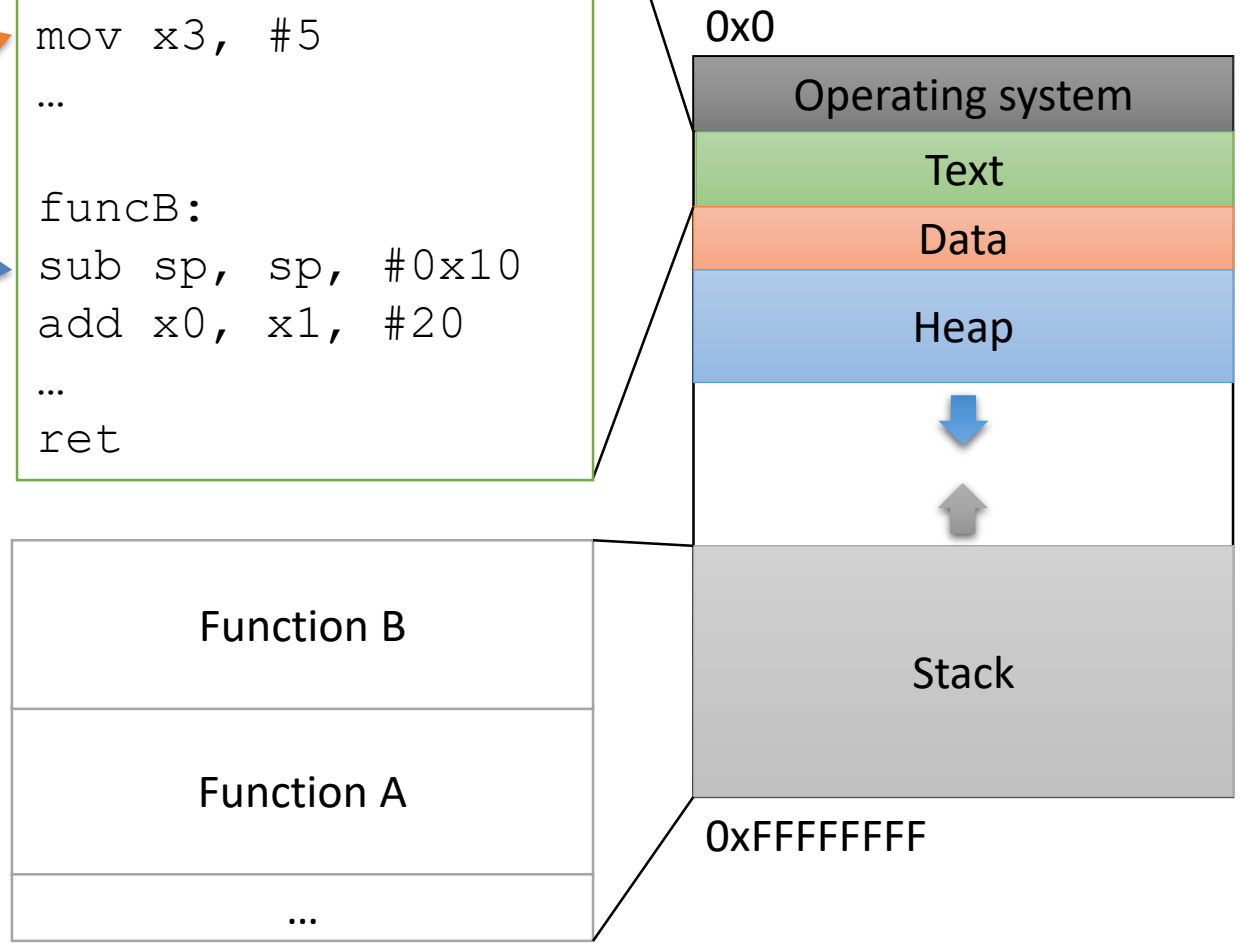
1. Save PC+4 (address of *next* instruction after the `bl`) into x30 (LR)

2. Unconditionally change the PC to the start of the callee's code

Instructions in Memory



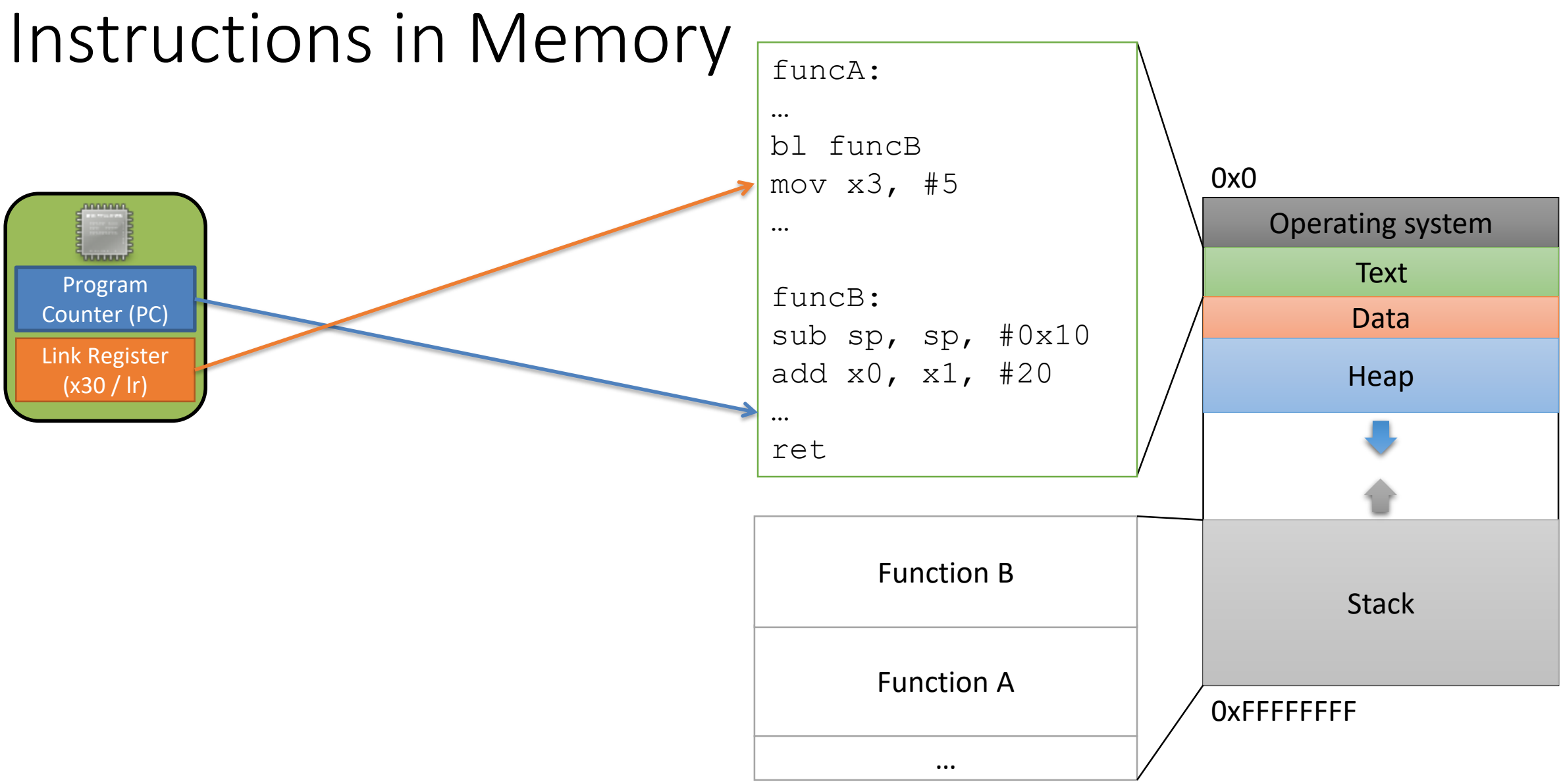

```
funcA:
…
bl funcB
mov x3, #5
…

funcB:
sub sp, sp, #0x10
add x0, x1, #20
…
ret
```



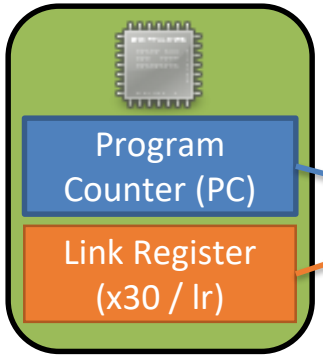
1. Save PC+4 (address of *next* instruction after the `bl`) into x30 (LR)

2. Unconditionally change the PC to the start of the callee's code

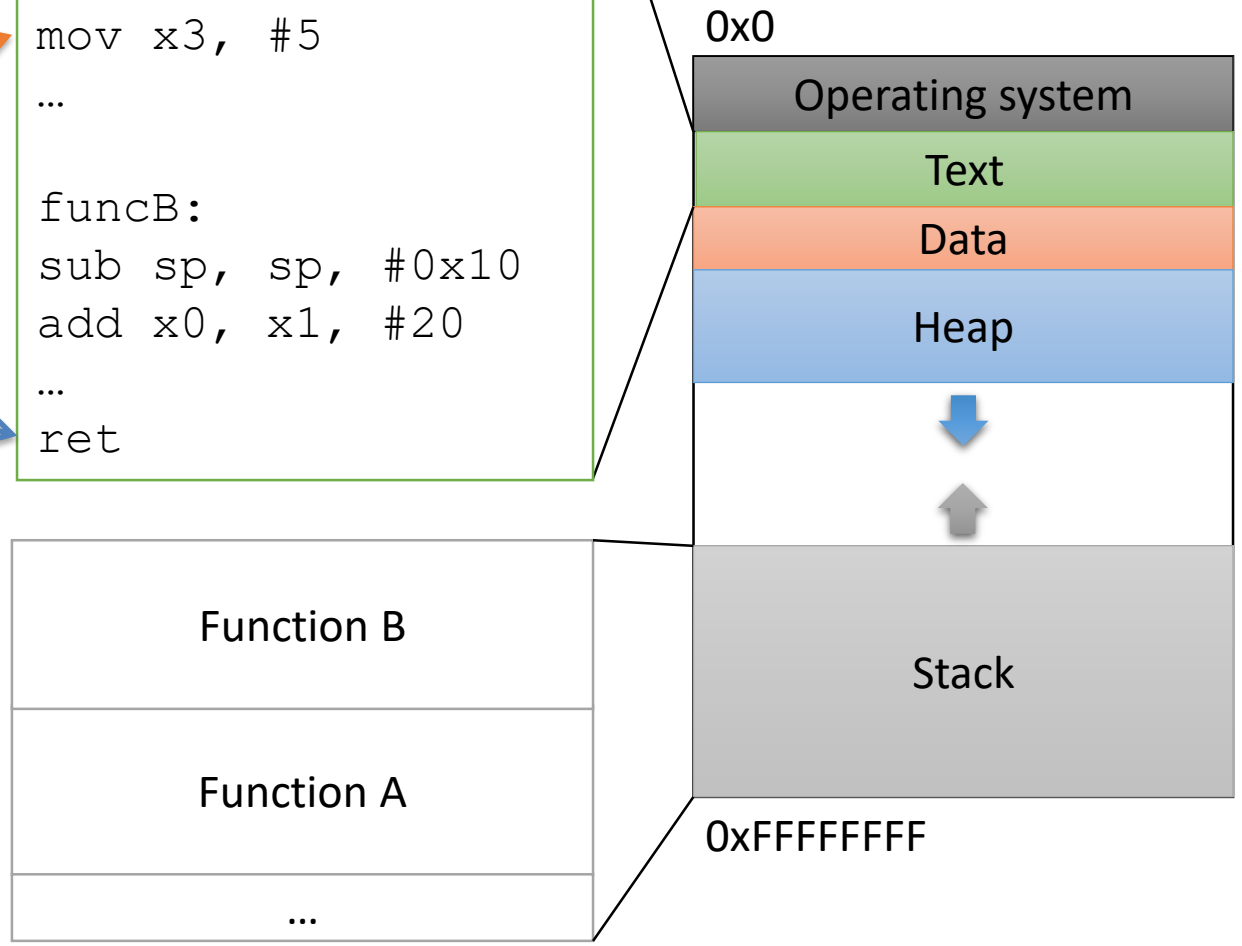
Instructions in Memory



Instructions in Memory

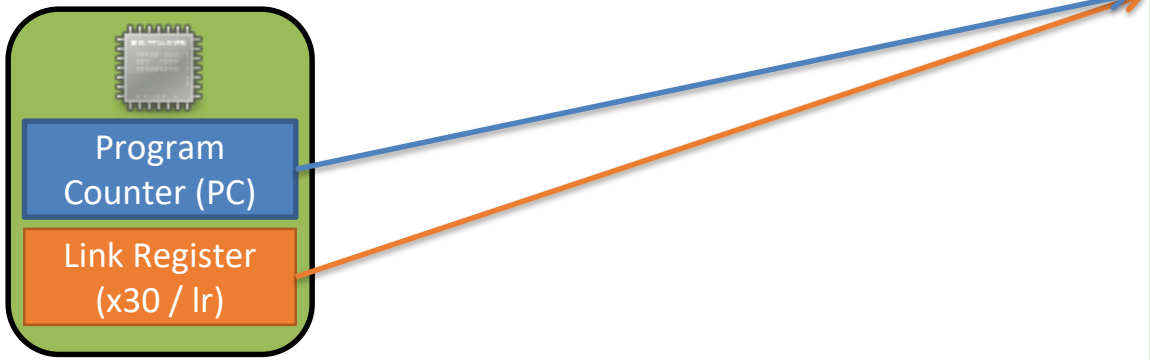



```
funcA:
…
bl funcB
mov x3, #5
…

funcB:
sub sp, sp, #0x10
add x0, x1, #20
…
ret
```

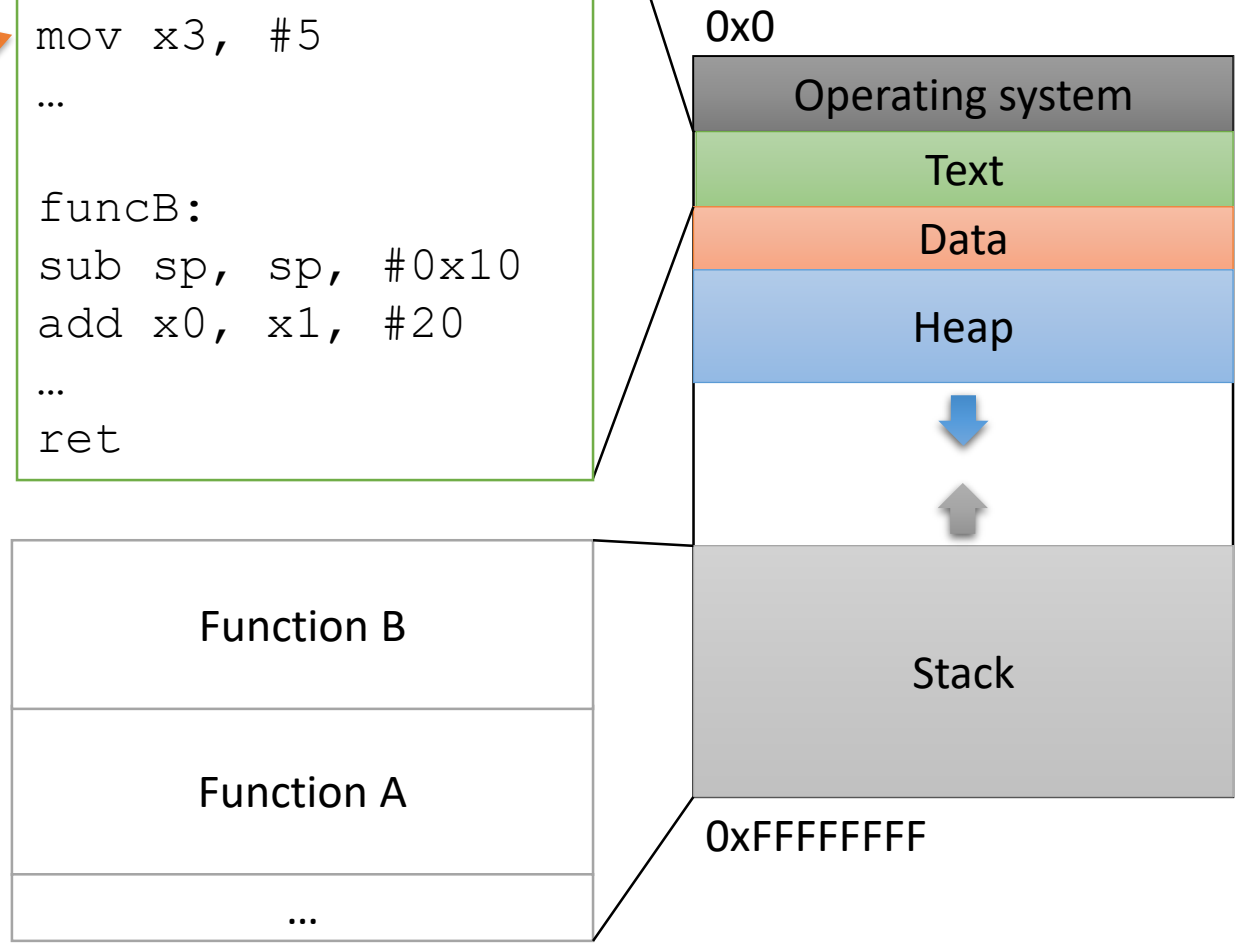


Restore PC to the address in x30 (LR)

Instructions in Memory




```
funcA:
…
bl funcB
mov x3, #5
…

funcB:
sub sp, sp, #0x10
add x0, x1, #20
…
ret
```

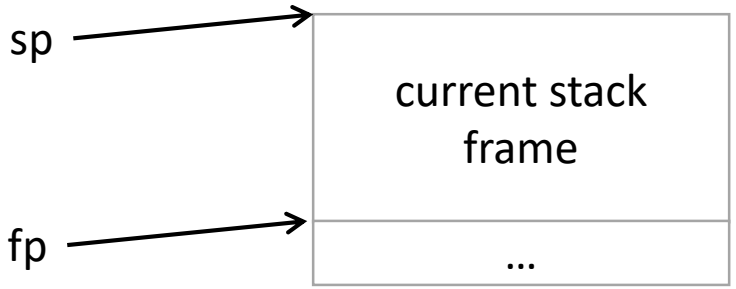


Restore PC to the address in x30 (LR)

Two important pieces of state…

1. Keeping track of top/bottom of stack

- Dedicate CPU registers for stack bookkeeping
  - sp (stack pointer, x31): Top of current stack frame
  - fp (frame pointer, x29): Base of current stack frame



2. Keeping track of return address

- Dedicate CPU register for storing the address to jump back to (x30)

This all works fine if just one function calls one other function. What if we want to chain multiple function calls though? (e.g., A calls B calls C, calls …)

There's only ONE of: x29, x30, x31

ARM64 Calling Convention

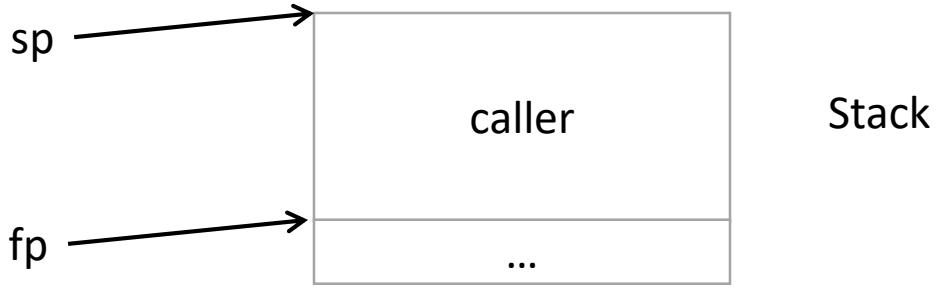
- When possible, keep values in registers
  - ARM has lots of registers available
  - Accessing registers is faster than memory (stack)

- When a function (caller) calls another (callee):
  - **The caller saves the frame pointer (fp / x29) and return address (lr / x30) on the stack.**
  - The caller passes the first eight arguments in registers x0 - x7
    (If more arguments are needed, they go on the stack)
  - If the callee produces a result, it returns it via register x0

Function Call Sequence

- Must maintain invariant:
  - The current function's stack frame is always between the addresses stored in sp and fp
  - The link register contains the return address

- Must adjust sp, fp on call / return.



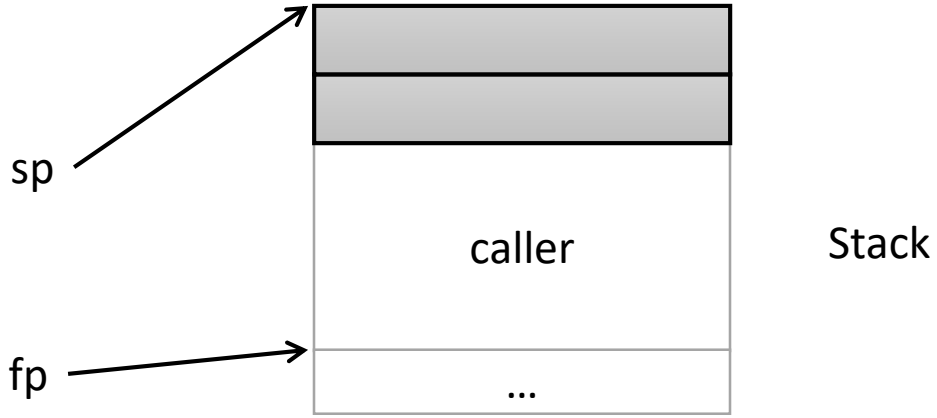
Function Call Sequence

- Must maintain invariant:
  - The current function’s stack frame is always between the addresses stored in sp and fp
  - The link register contains the return address

- Before calling a function, the caller:
  1. Raises the stack pointer to make room for x29 (fp) and x30 (lr)

Explanation:

We need space on the stack to store the caller's return address and the caller's frame pointer so that we can restore them later.



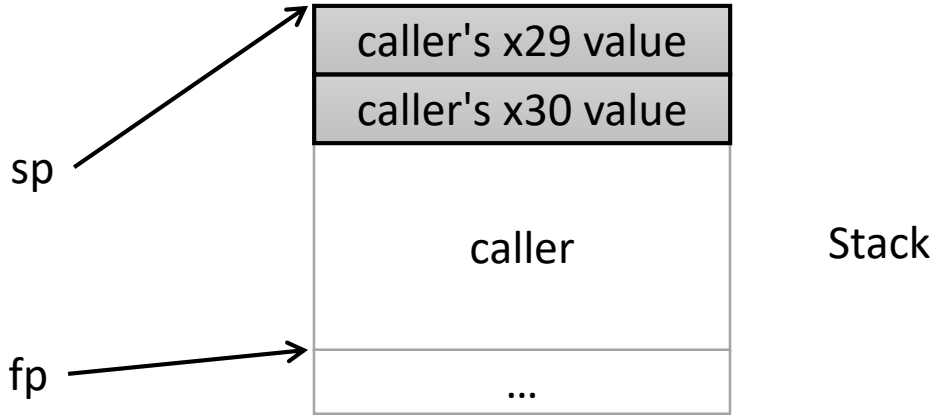
Function Call Sequence

- Must maintain invariant:
  - The current function’s stack frame is always between the addresses stored in sp and fp
  - The link register contains the return address

- Before calling a function, the caller:
  1. Raises the stack pointer to make room for x29 (fp) and x30 (lr)
  2. Saves x29 and x30 in the new space

Explanation:

We need space on the stack to store the caller's return address and the caller's frame pointer so that we can restore them later.



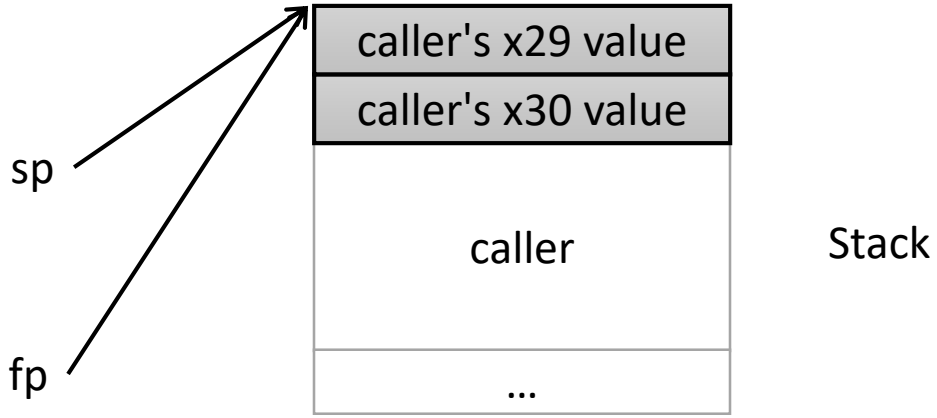
Function Call Sequence

- Must maintain invariant:
  - The current function's stack frame is always between the addresses stored in sp and fp
  - The link register contains the return address

- Before calling a function, the caller:
  1. Raises the stack pointer to make room for x29 (fp) and x30 (lr)
  2. Saves x29 and x30 in the new space
  3. Set x29 (fp) to match sp

Explanation:

We're building a new stack frame for the callee, and fp should point to the bottom of it. The top of the caller's stack frame is the bottom of the callee's.



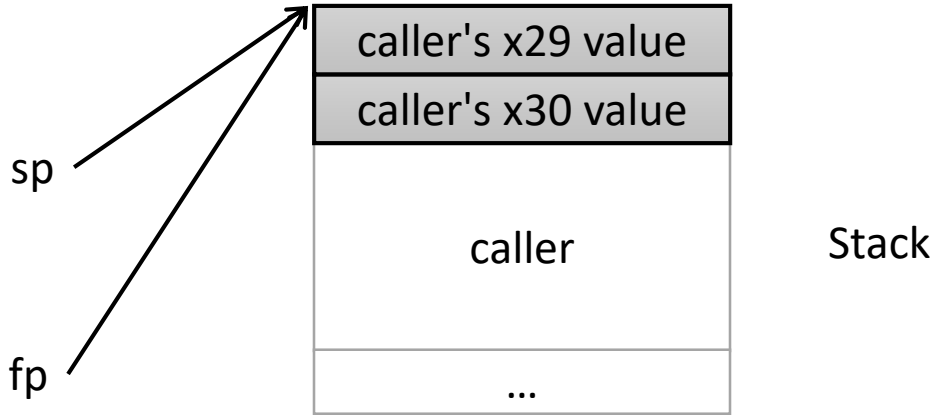
Function Call Sequence

- Must maintain invariant:
  - The current function's stack frame is always between the addresses stored in sp and fp
  - The link register contains the return address

- Before calling a function, the caller:
  1. Raises the stack pointer to make room for x29 (fp) and x30 (lr)
  2. Saves x29 and x30 in the new space
  3. Set x29 (fp) to match sp
  4. Execute a `bl` instruction to begin the callee. This overwrites x30 to be the address of the caller's next instruction.

Explanation:

When the callee returns, we need to know where to resume execution in the caller. Register x29 stores the address of the caller's next instruction after executing `bl`.

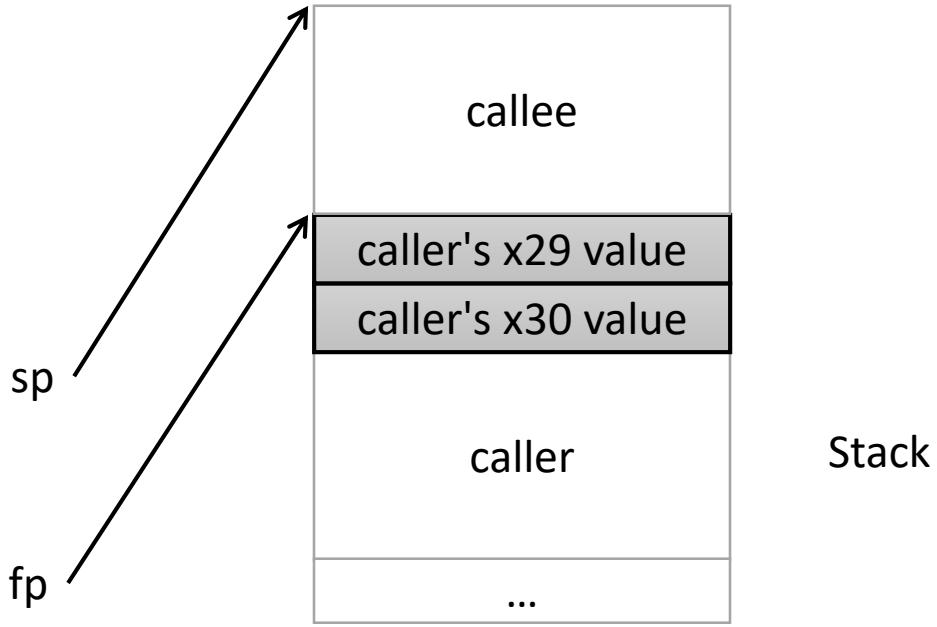


Callee Executes…

- Must maintain invariants:
  - The current function's stack frame is always between the addresses stored in sp and fp
  - The link register contains the return address

- At this point, the callee has begun execution. It can do what it wants.

- Callee will generally raise the stack pointer to give itself space for local variables

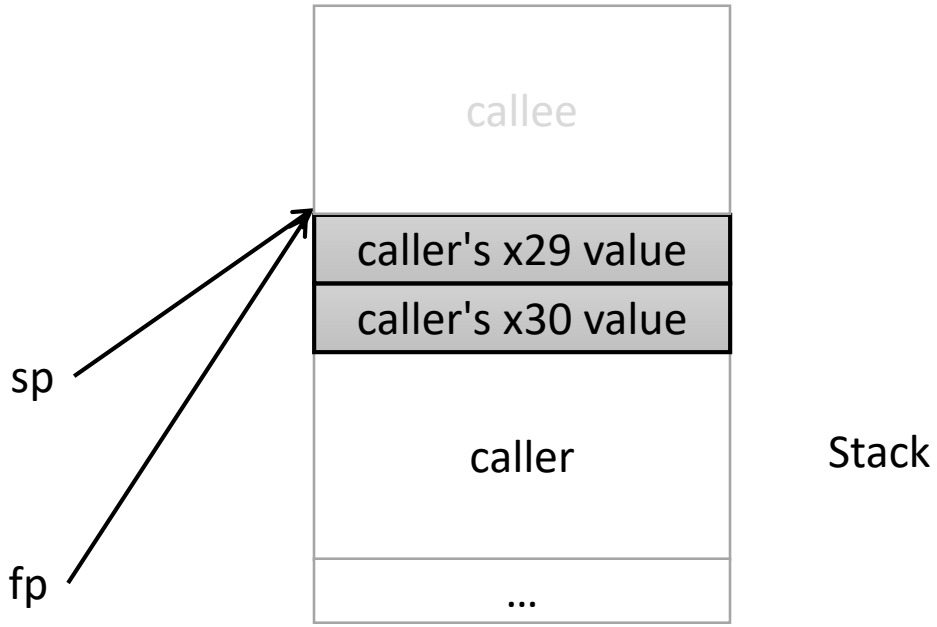


Callee Executes…

- Must maintain invariants:
  - The current function's stack frame is always between the addresses stored in sp and fp
  - The link register contains the return address

- At this point, the callee has begun execution. It can do what it wants.

- Before returning, callee resets sp back to what it was when it started

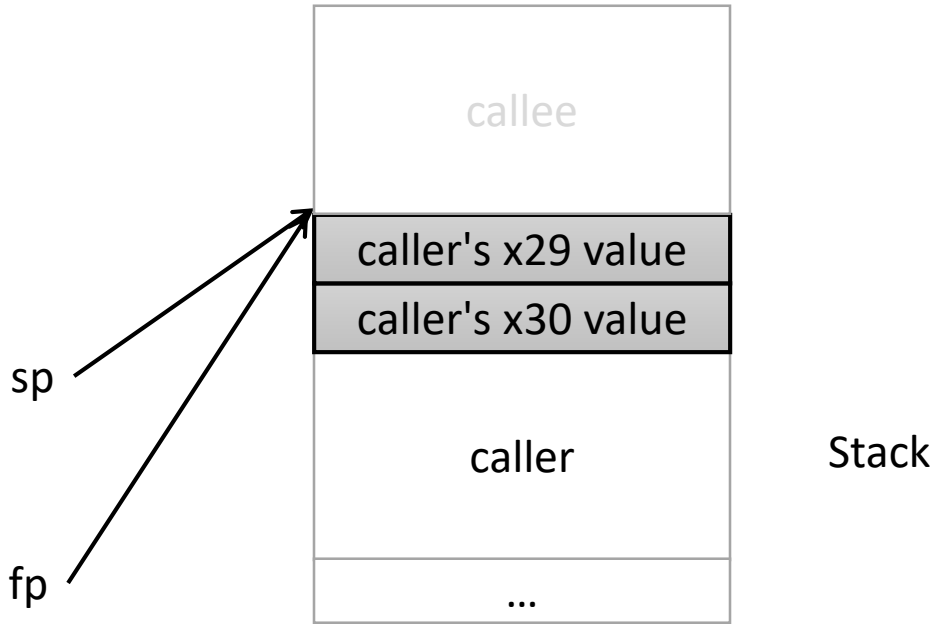


Callee Executes…

- Must maintain invariants:
  - The current function's stack frame is always between the addresses stored in sp and fp
  - The link register contains the return address

- At this point, the callee has begun execution. It can do what it wants.

- Before returning, callee resets sp back to what it was when it started

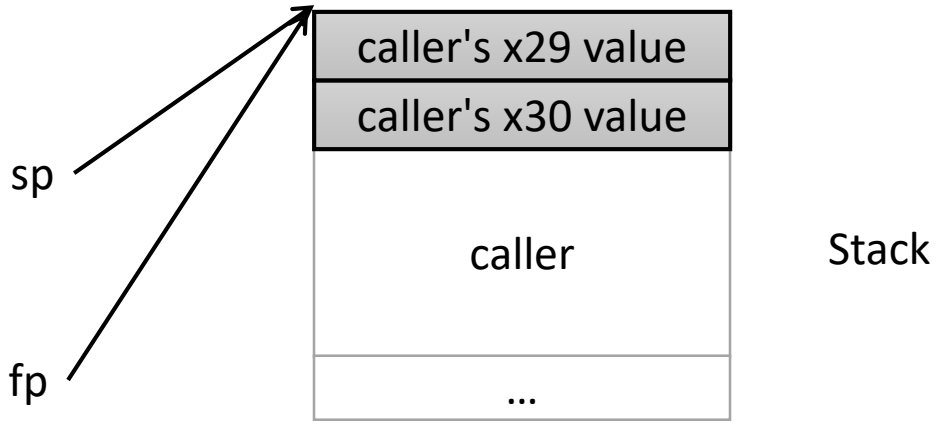


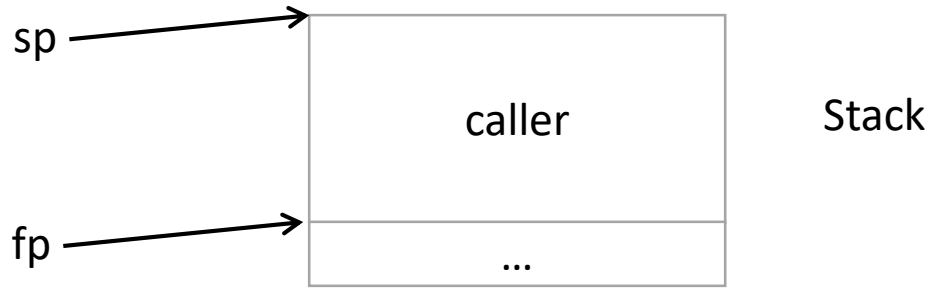
Callee Returns…

- Must maintain invariants:
  - The current function's stack frame is always between the addresses stored in sp and fp
  - The link register contains the return address

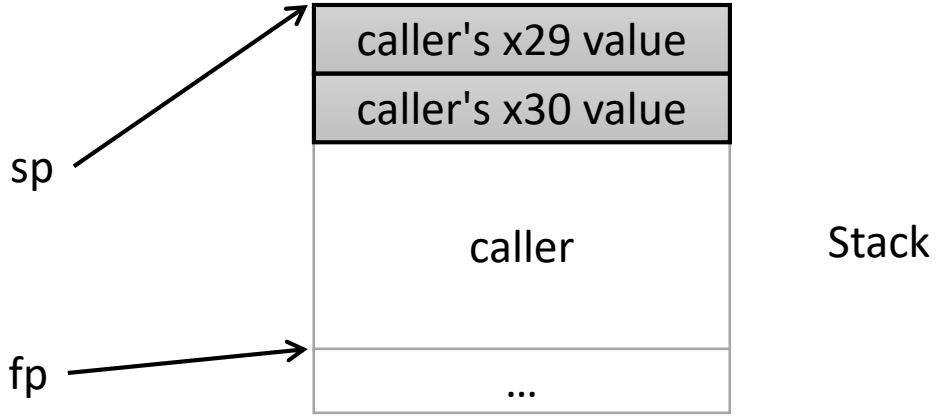
- Caller resumes execution

- Caller (eventually) restores x29 and x30 from values saved on stack

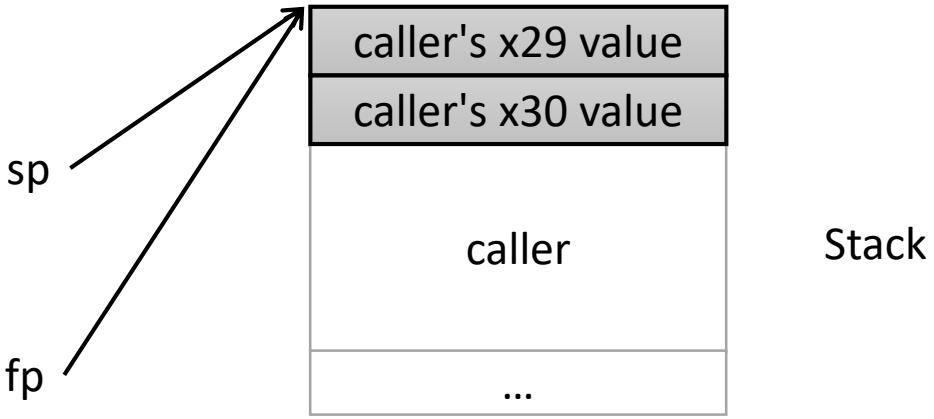




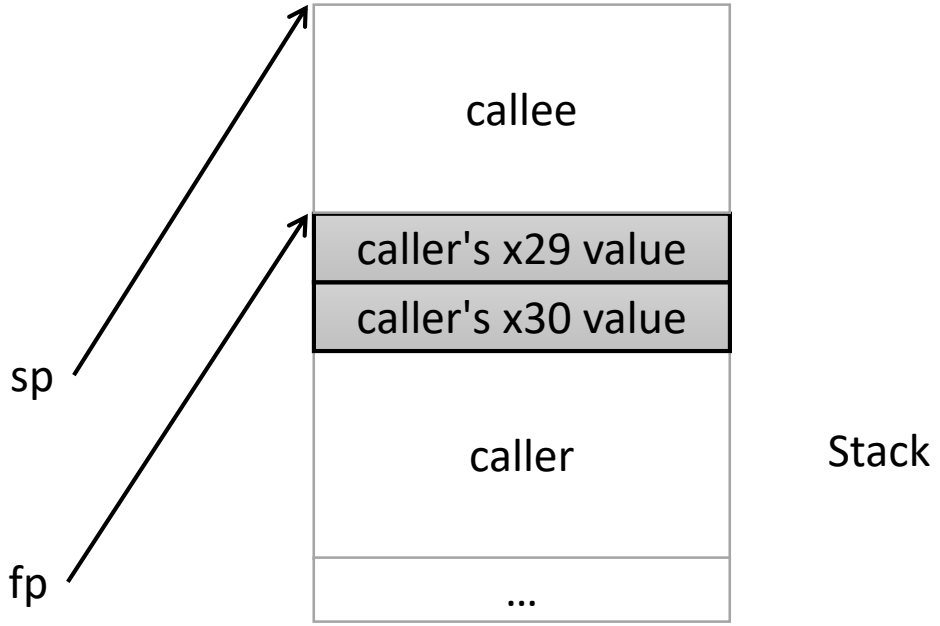
Caller prior to function call.



Caller saves its fp and lr registers.



Caller sets fp to bottom of callee's new frame.



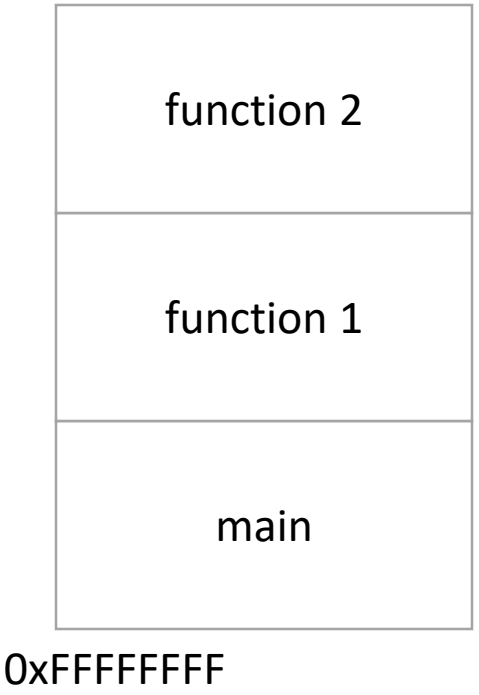
Callee executes.

Stack Frame Contents

- What needs to be stored in a stack frame?
  - Alternatively: What *must* a function know?

- Local variables
- Previous stack frame base address
- Function arguments
- Return value
- Return address

- Saved registers
- Spilled temporaries



Saving Registers

- Registers are a relatively scarce resource, but they're fast to access. Memory is plentiful, but slower to access.

- Should the caller save its registers to free them up for the callee to use?
- Should the callee save the registers in case the caller was using them?
- Who needs more registers for temporary calculations, the caller or callee?

- Clearly the answers depend on what the functions do…

Splitting the difference…

- We can’t know the answers to those questions in advance…

- Divide registers into two groups:
  - Caller-saved: x9-x15
    - If the caller wants to preserve these registers, it must save them prior to calling callee
    - callee free to trash these, caller will restore if needed
  - Callee-saved: x19-x29
    - If the callee wants to use these registers, it must save them first, and restore them before returning
    - caller can assume these will be preserved

Running Out of Registers

- Some computations require more than 29 general-purpose registers to store temporary values.

- *Register spilling*: The compiler will move some temporary values to memory, if necessary.
  - Values pushed onto stack, popped off later
  - No explicit variable declared by user

Up next…

- Arrays, Structs, and Pointers