CS 31: Introduction to Computer Systems

13: Assembly Functions

03-06-2025

SWARTHMORE COLLEGE

Announcements

Midterm Scores up on gradescope!

Four Types of Assembly Instructions

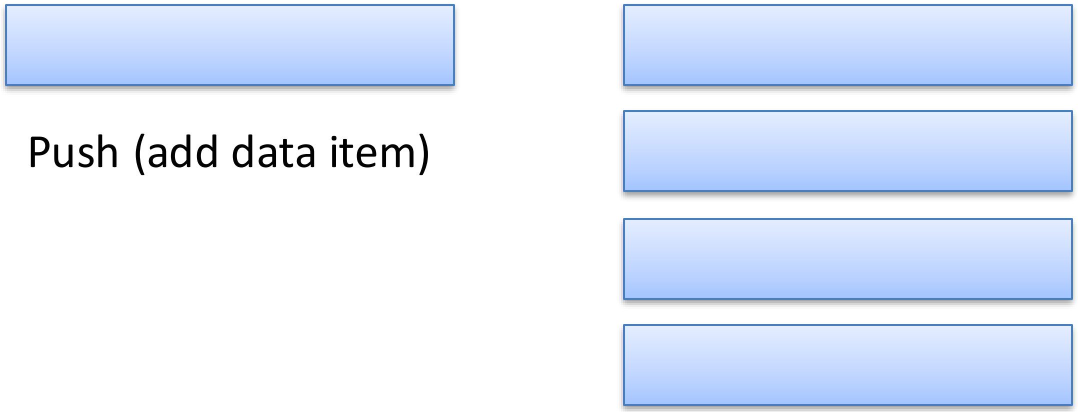
1. Arithmetic: use ALU to compute a value
2. Data movement: load and store
3. Control Flow: branch, jump, etc.
4. Stack Instructions: push and pop stack frames
   – Shortcut instructions for common operations (we’ll cover these in detail later)

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)



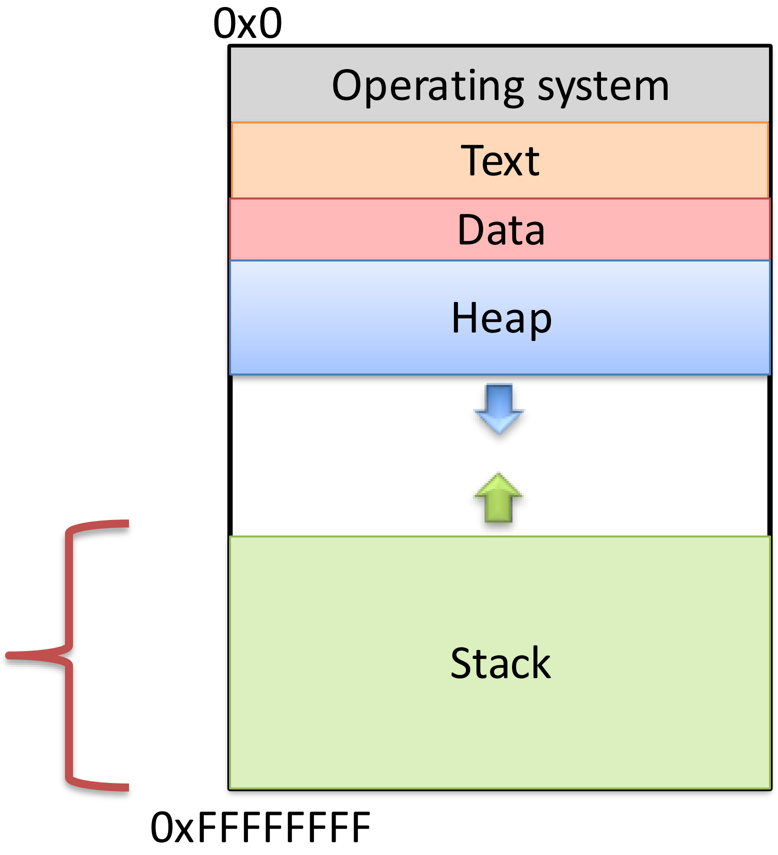
“The” Stack

- 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

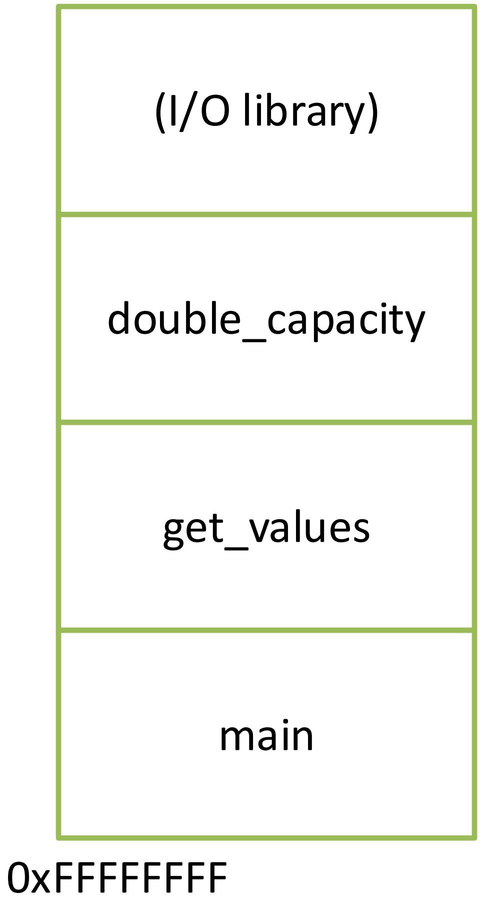
- Starts at the highest memory addresses, grows into lower addresses.



Stack Frames

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

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

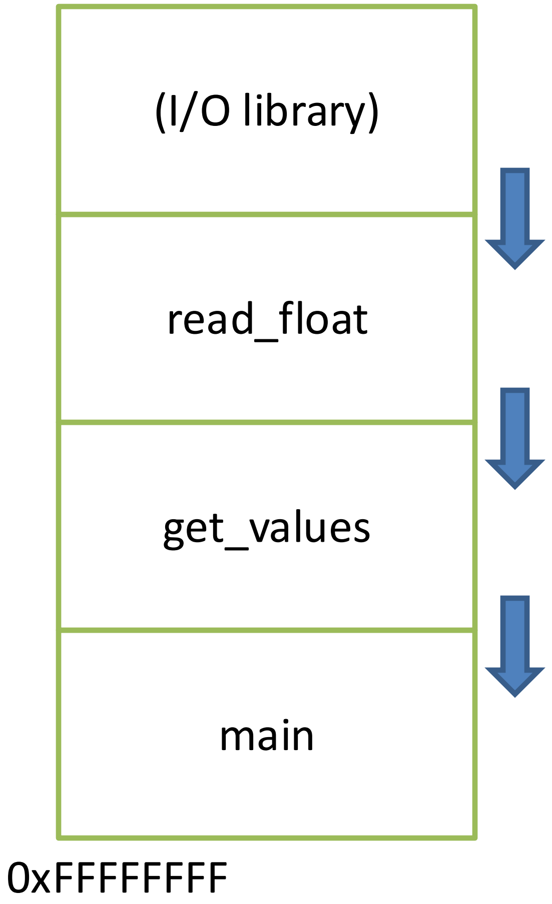


Stack Frames

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

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

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.

What is responsible for creating and removing stack frames?

A. The user

B. The compiler

C. C library code

D. The operating system

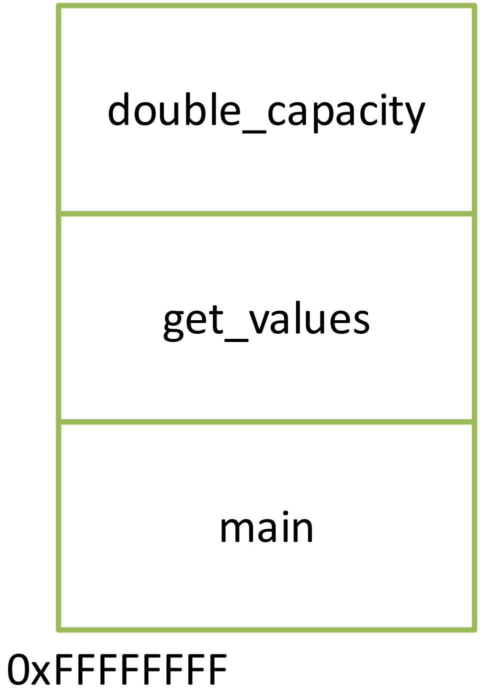
E. Something / someone else

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

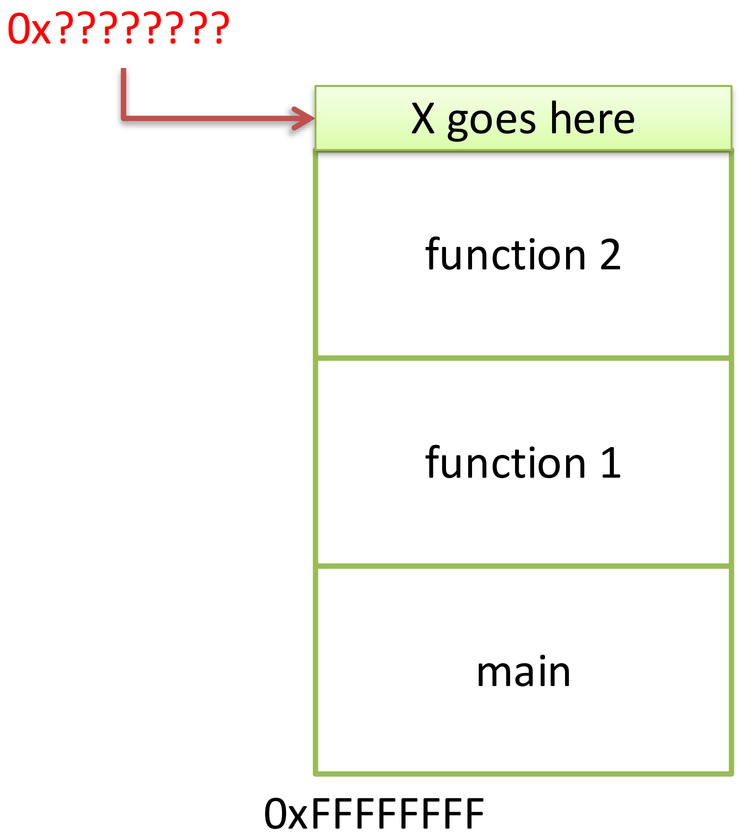
If the programmer says:

```
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 or a new stack frame?

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.

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

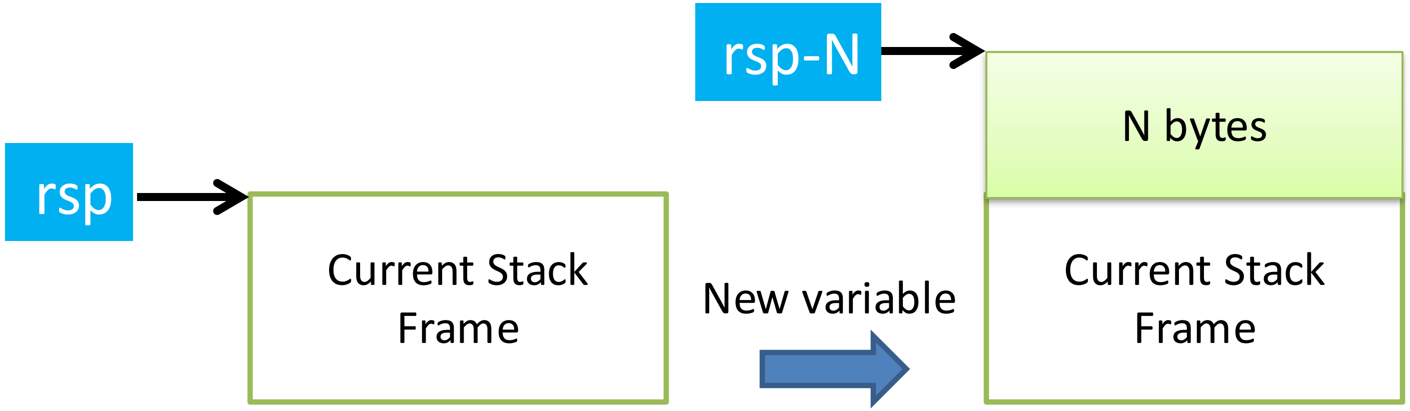
- Run time (dynamic)
  - Information that isn't known until program is running
  - Depends on machine characteristics and user input

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

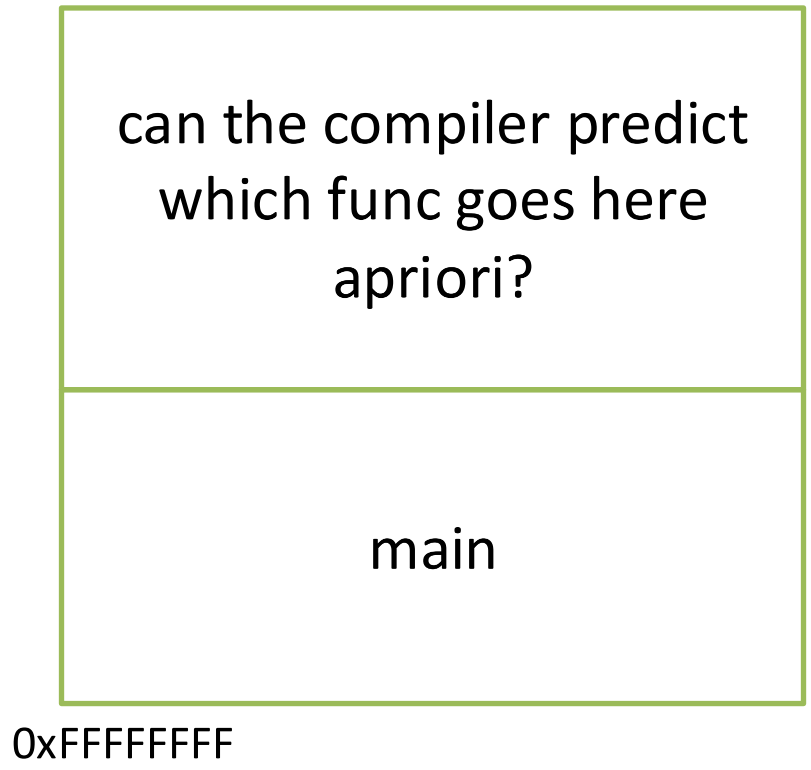
Local Variables

Compiler can allocate N bytes on the stack by subtracting N from the stack pointer: (rsp)



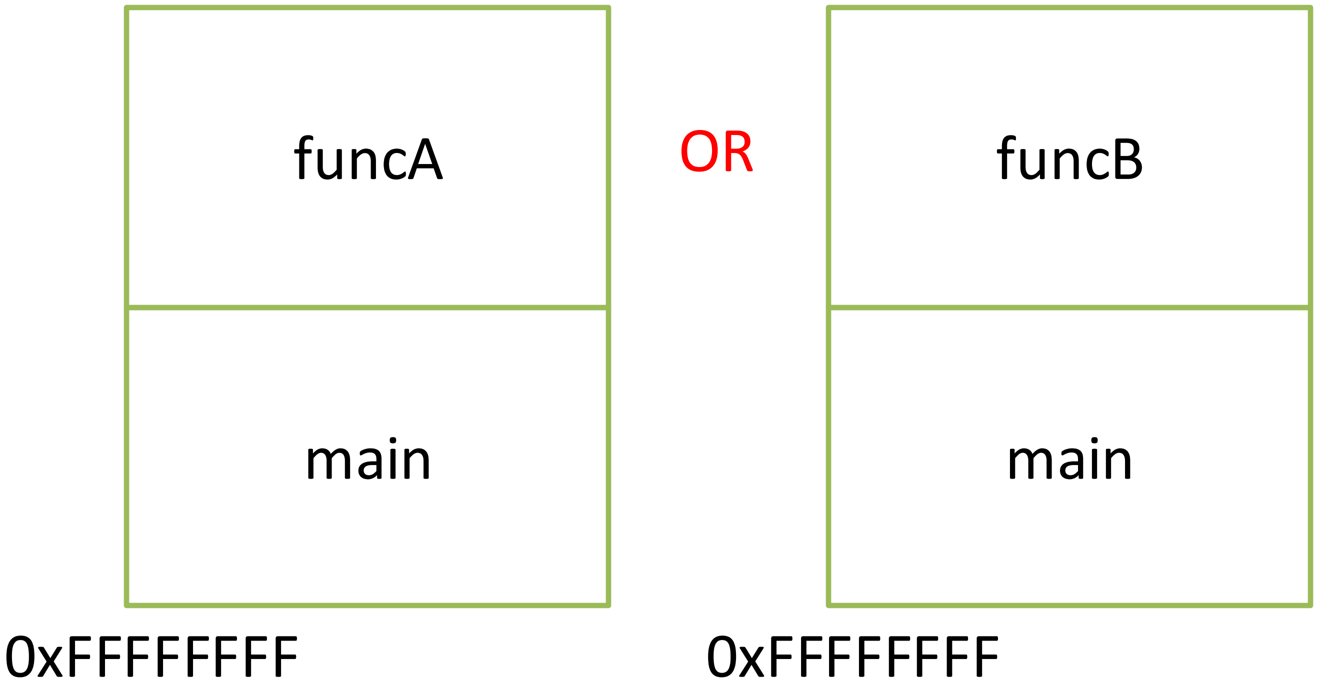
The Compiler Can't…Predict User Input

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



The Compiler Can't…Predict User Input

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



The Compiler Can't…

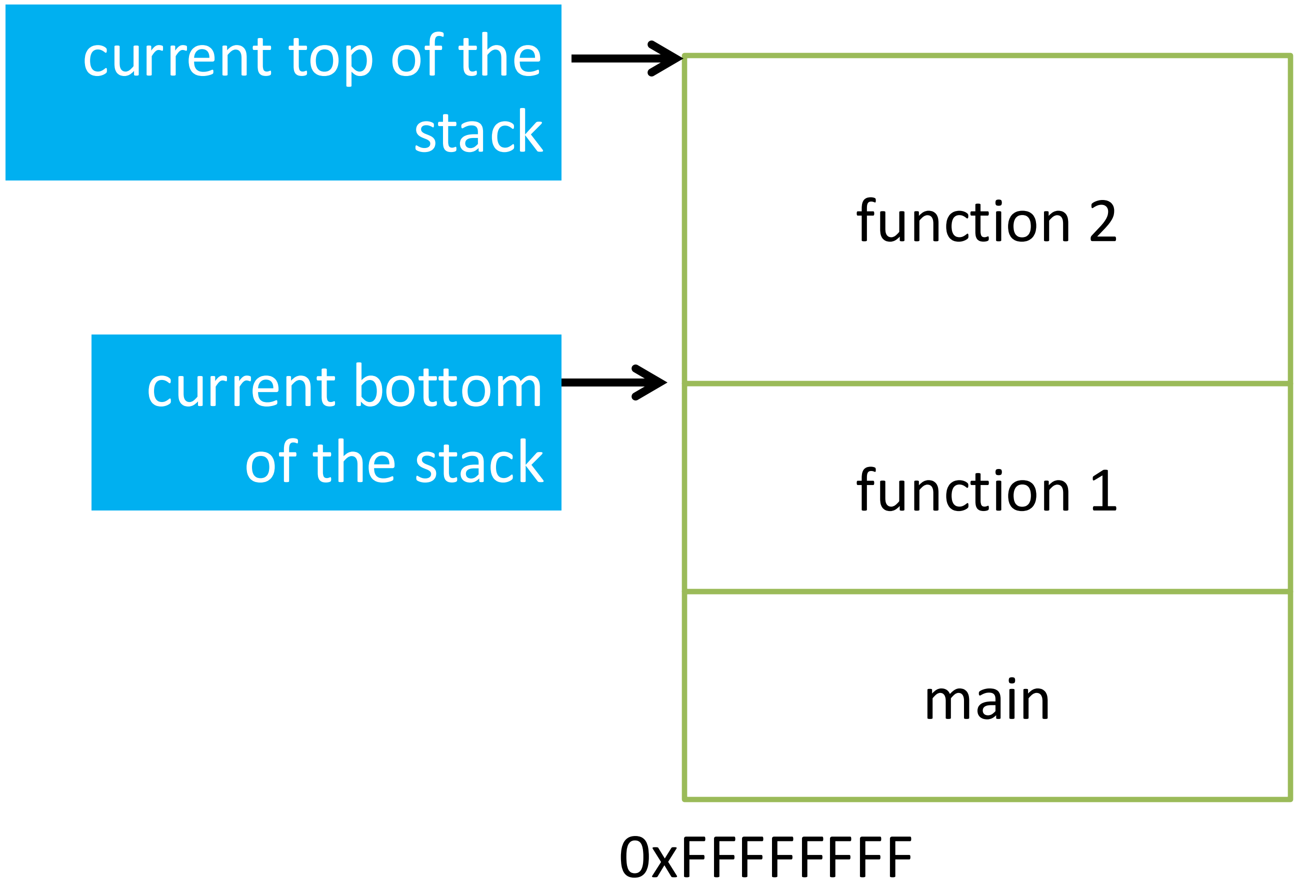
Predict user input.

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.

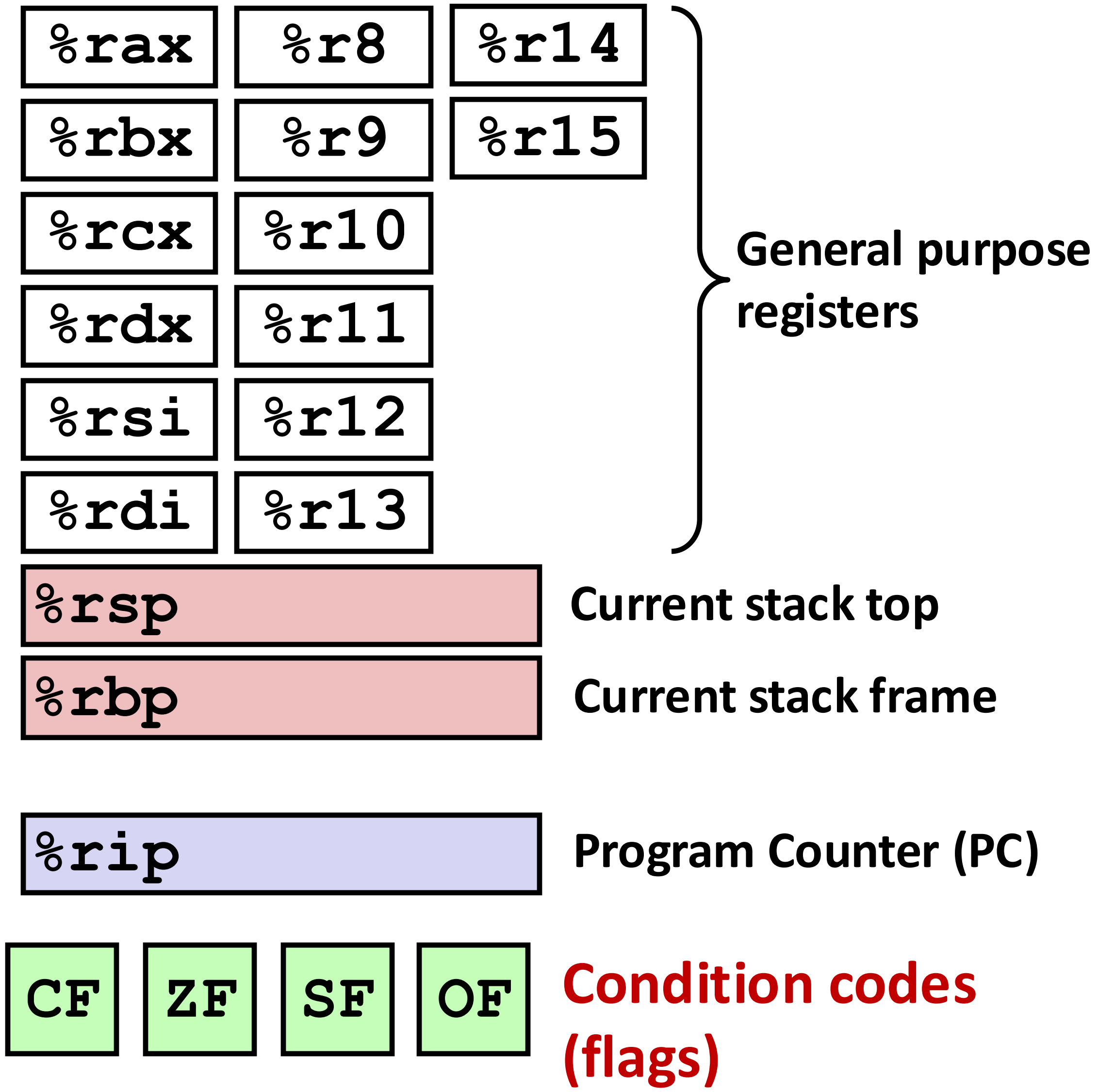
Stack Frame Location

Where in memory is the current stack frame?



Recall: x86_64 Register Conventions

- Working memory for currently executing program
  - Address of next instruction to execute (%rip)
  - <u>Location of runtime stack (%rbp, %rsp)</u>
  - Temporary data (%rax - %r15)
  - Status of recent ALU tests (CF, ZF, SF, OF)

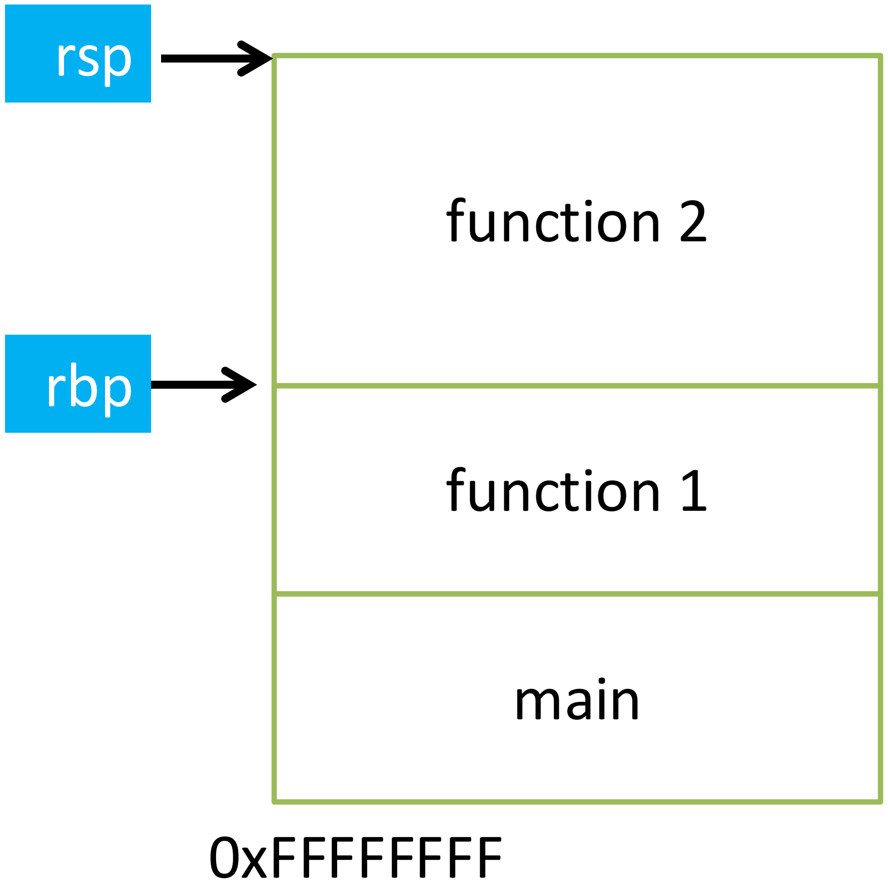


Stack Frame Location

Where in memory is the current stack frame?

- rsp: stack pointer
- rbp: frame pointer (base pointer)

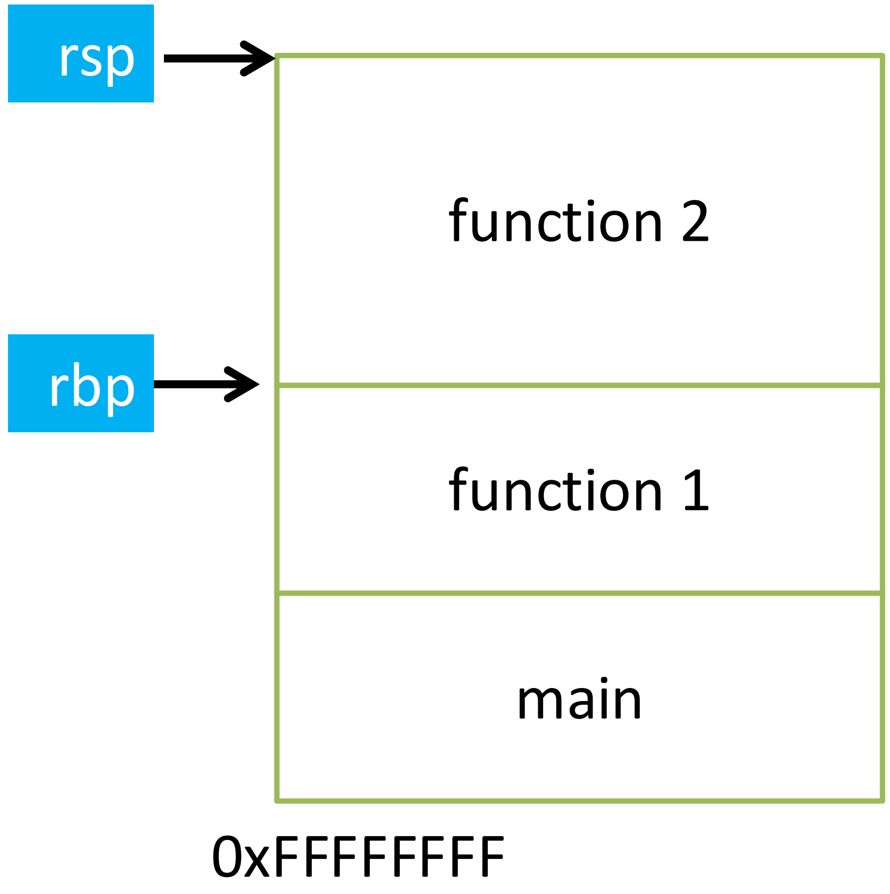
invariant:
The current function's stack frame is always between the addresses stored in `rsp` and `rbp`



Stack Frame Location

- Compiler ensures that this invariant holds.
- This is why all local variables we've seen in assembly are relative to `rbp` or `rsp`!

invariant:
The current function's stack frame is always between the addresses stored in `rsp` and `rbp`



How would we implement pushing x to the top of the stack in x86_64?

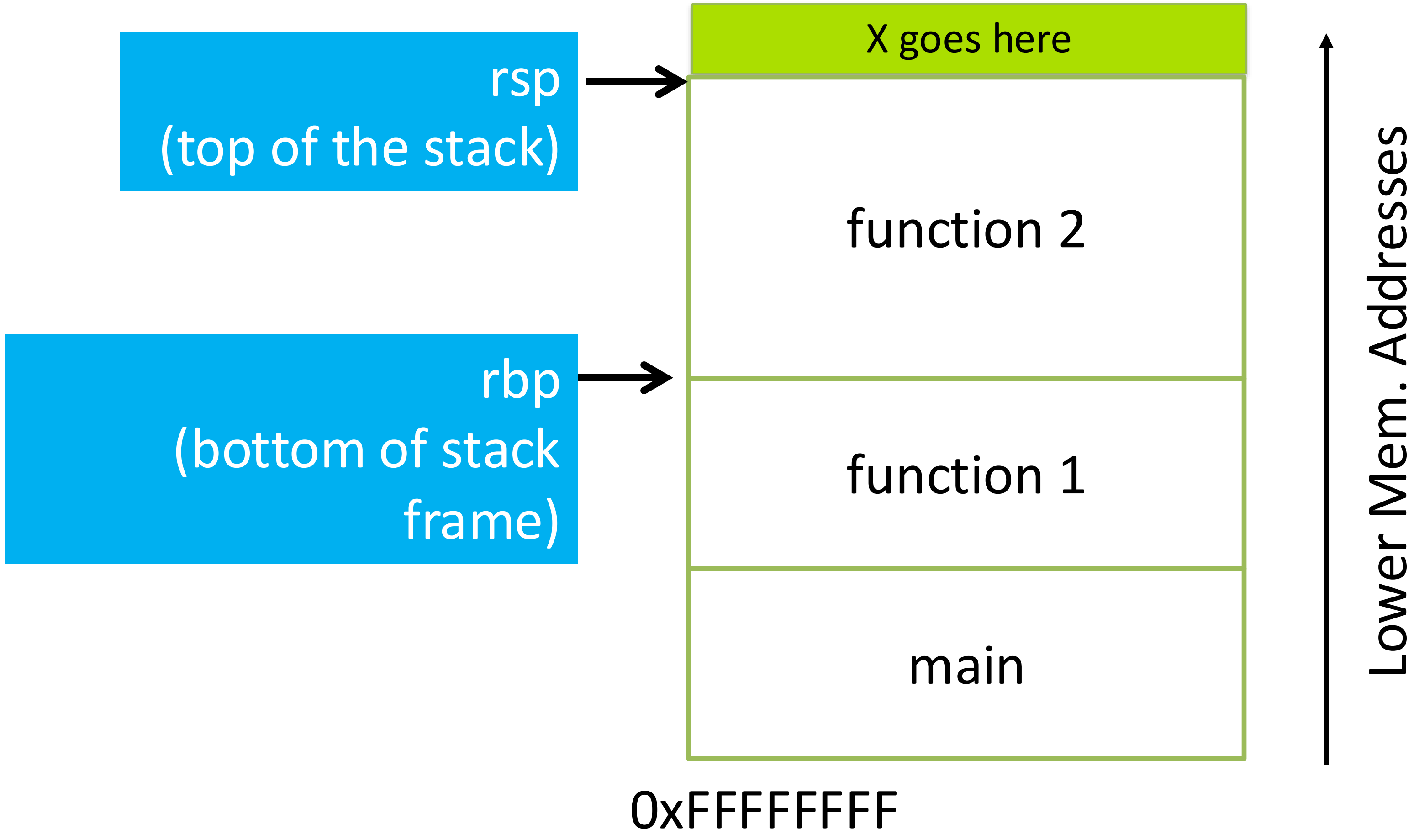
A. Increment rsp
   Store x at (rsp)

B. Store x at (rsp)
   Increment rsp

C. Decrement rsp
   Store x at (rsp)

D. Store x at (rsp)
   Decrement rsp

E. Copy rsp to rbp
   Store x at rbp



How would we implement pushing x to the top of the stack in x86_64?

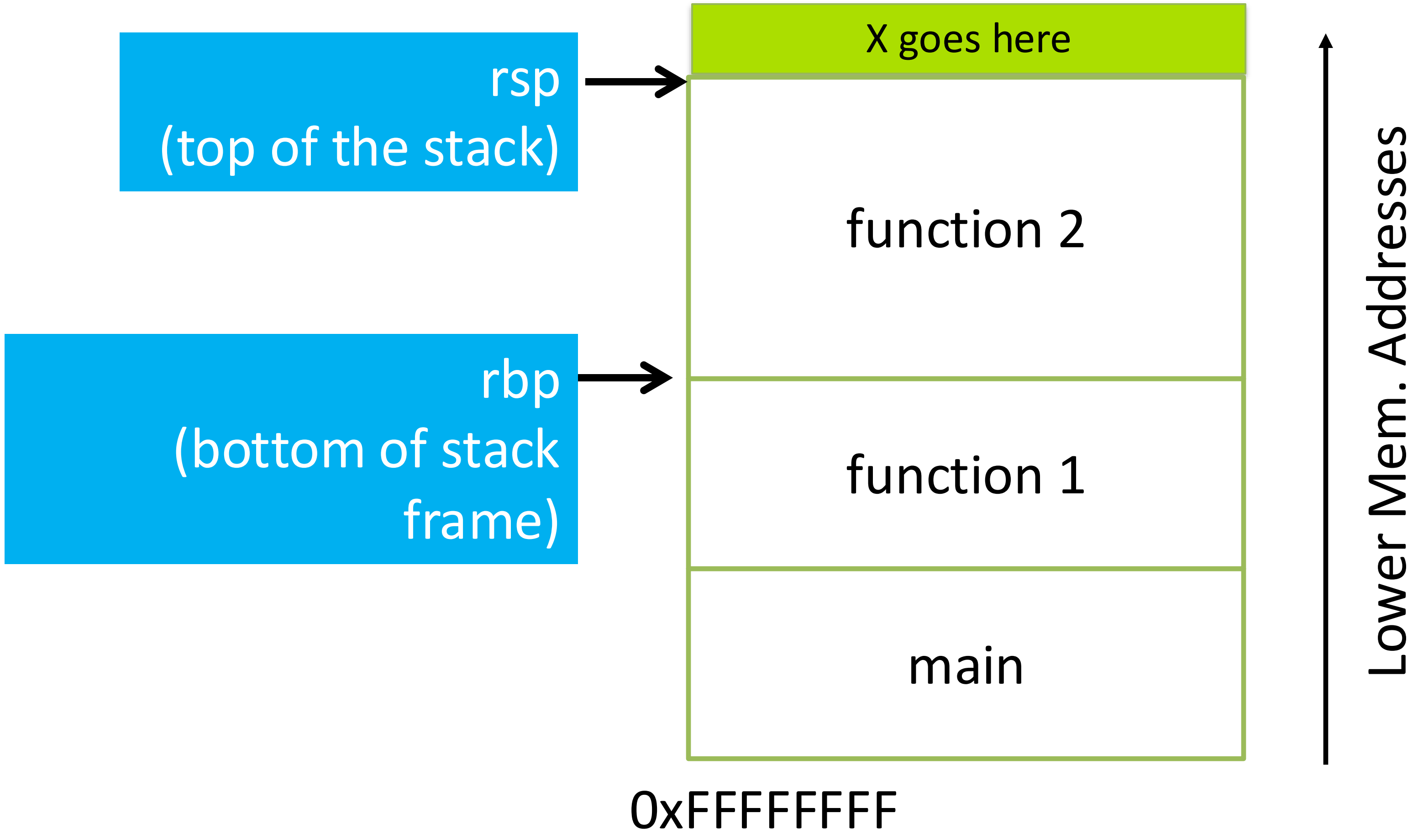
A. Increment rsp
   Store x at (rsp)

B. Store x at (rsp)
   Increment rsp

C. Decrement rsp
   Store x at (rsp)

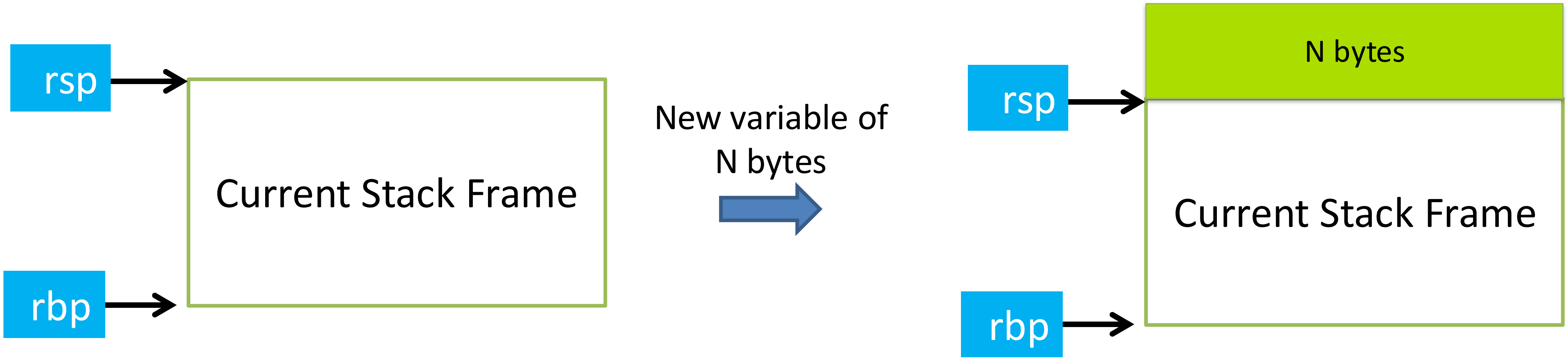
D. Store x at (rsp)
   Decrement rsp

E. Copy rsp to rbp
   Store x at rbp



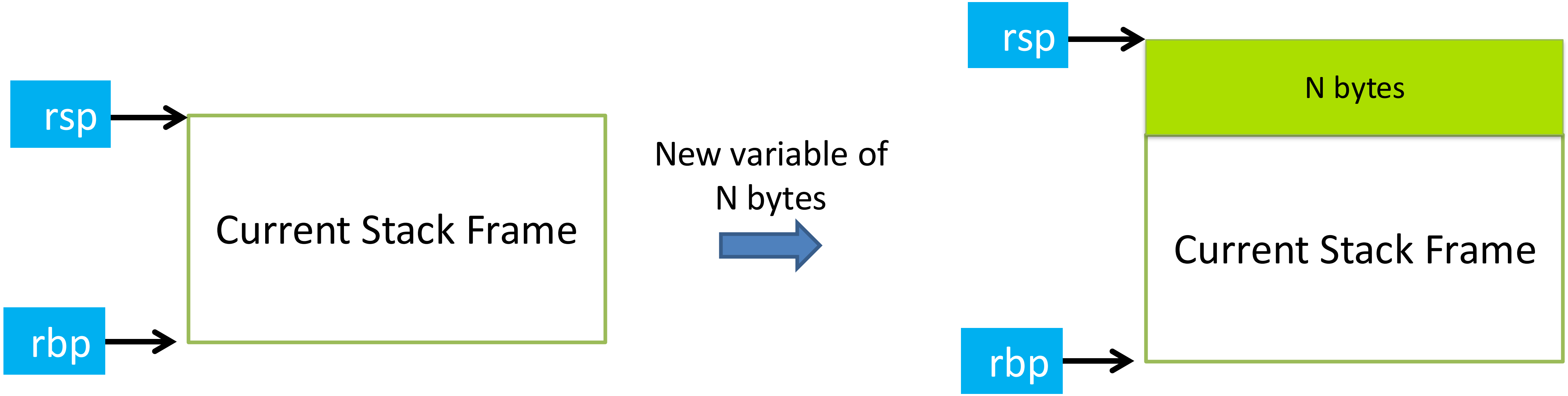
Local Variables

- Generally, we can make space on the stack for N bytes by:
  - <u>subtracting N from rsp</u>



Local Variables

- When we're done, free the space by adding N back to `rsp`
  - `rsp + N`

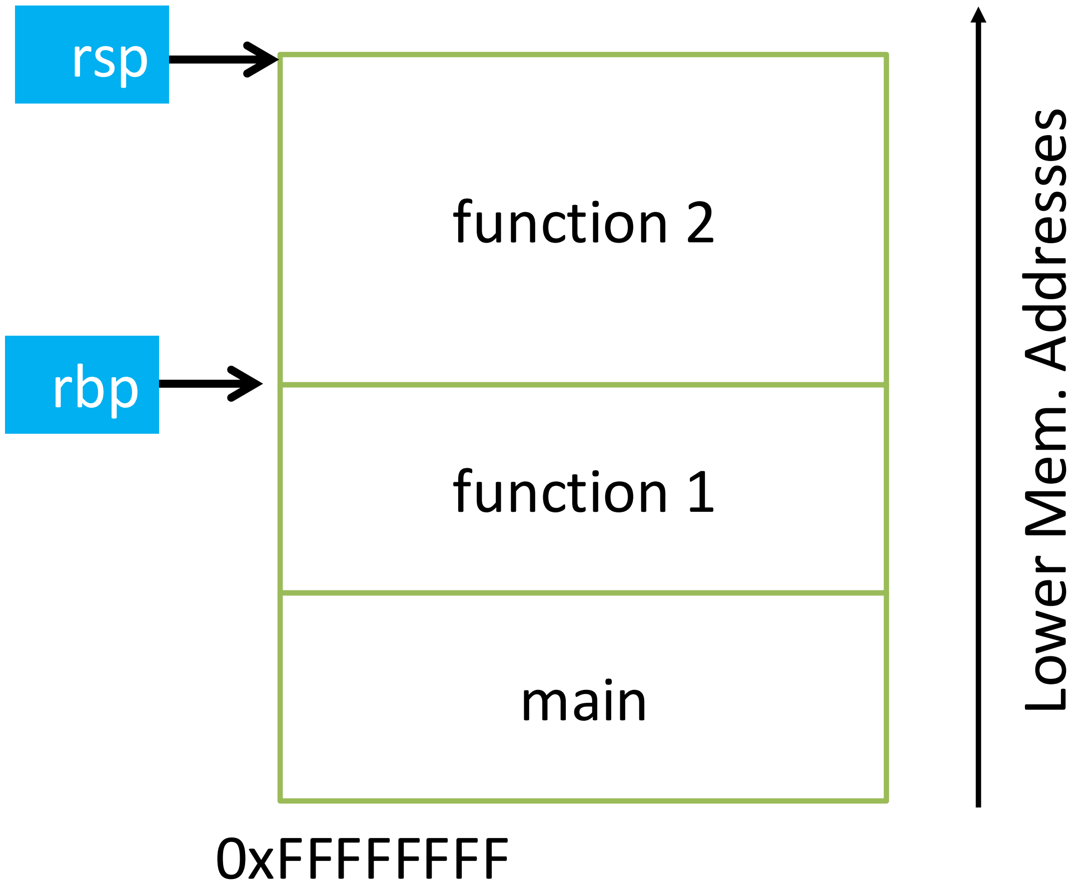


Stack Frame Contents

What needs to be stored in a stack frame? <u>What *must* a function know?</u>

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

- Saved registers
- Spilled temporaries

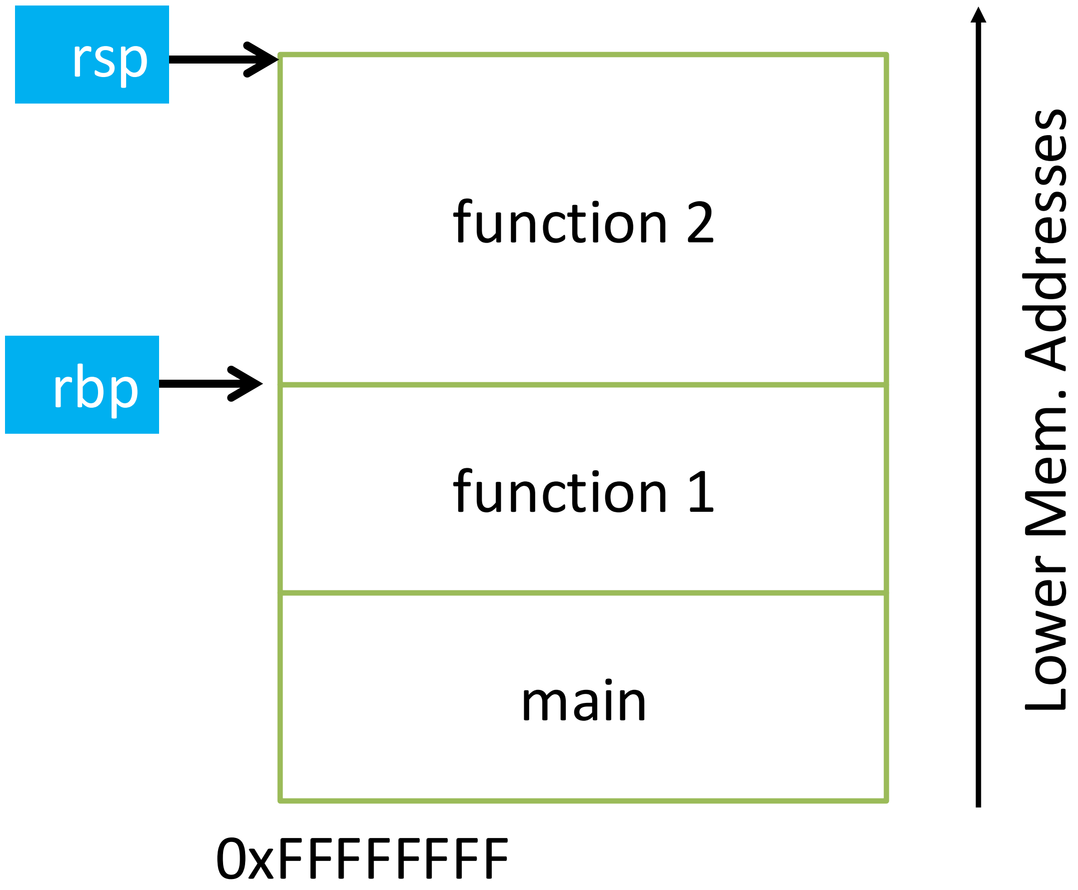


Stack Frame Contents

What needs to be stored in a stack frame? <u>What *must* a function know?</u>

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

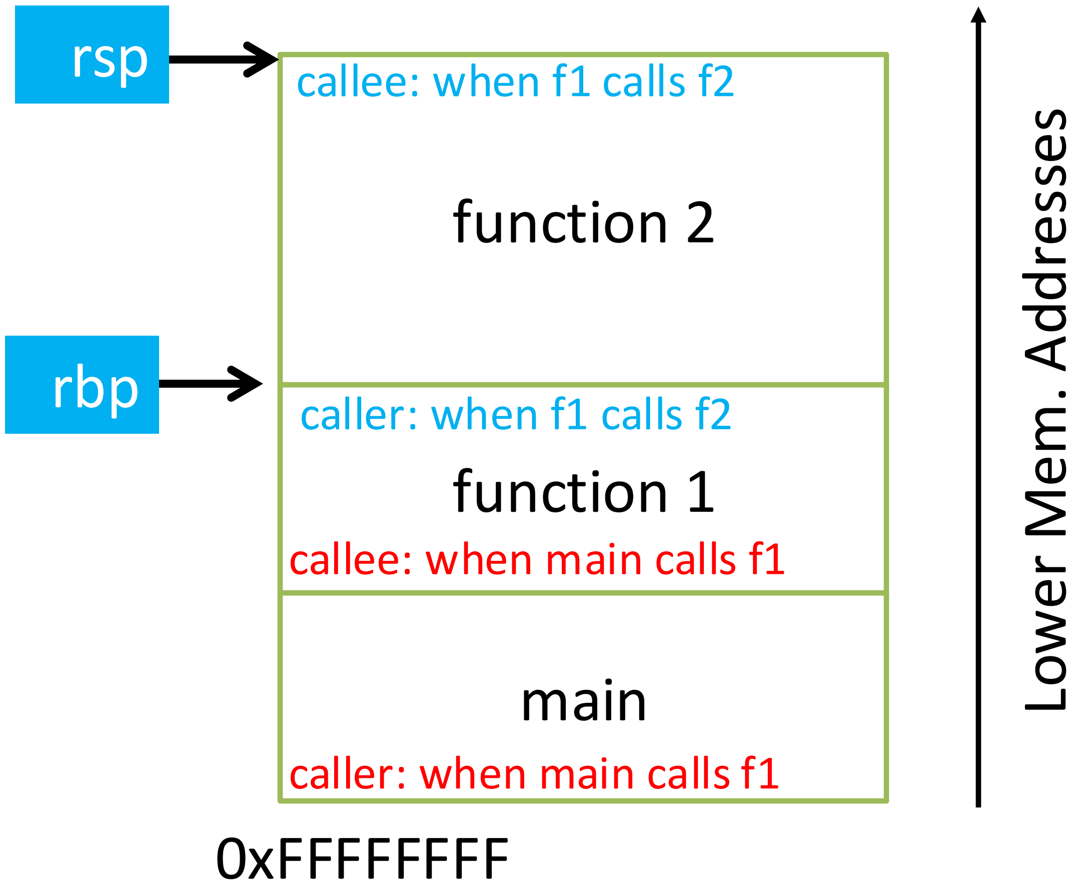
- Saved registers
- Spilled temporaries



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 the following 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

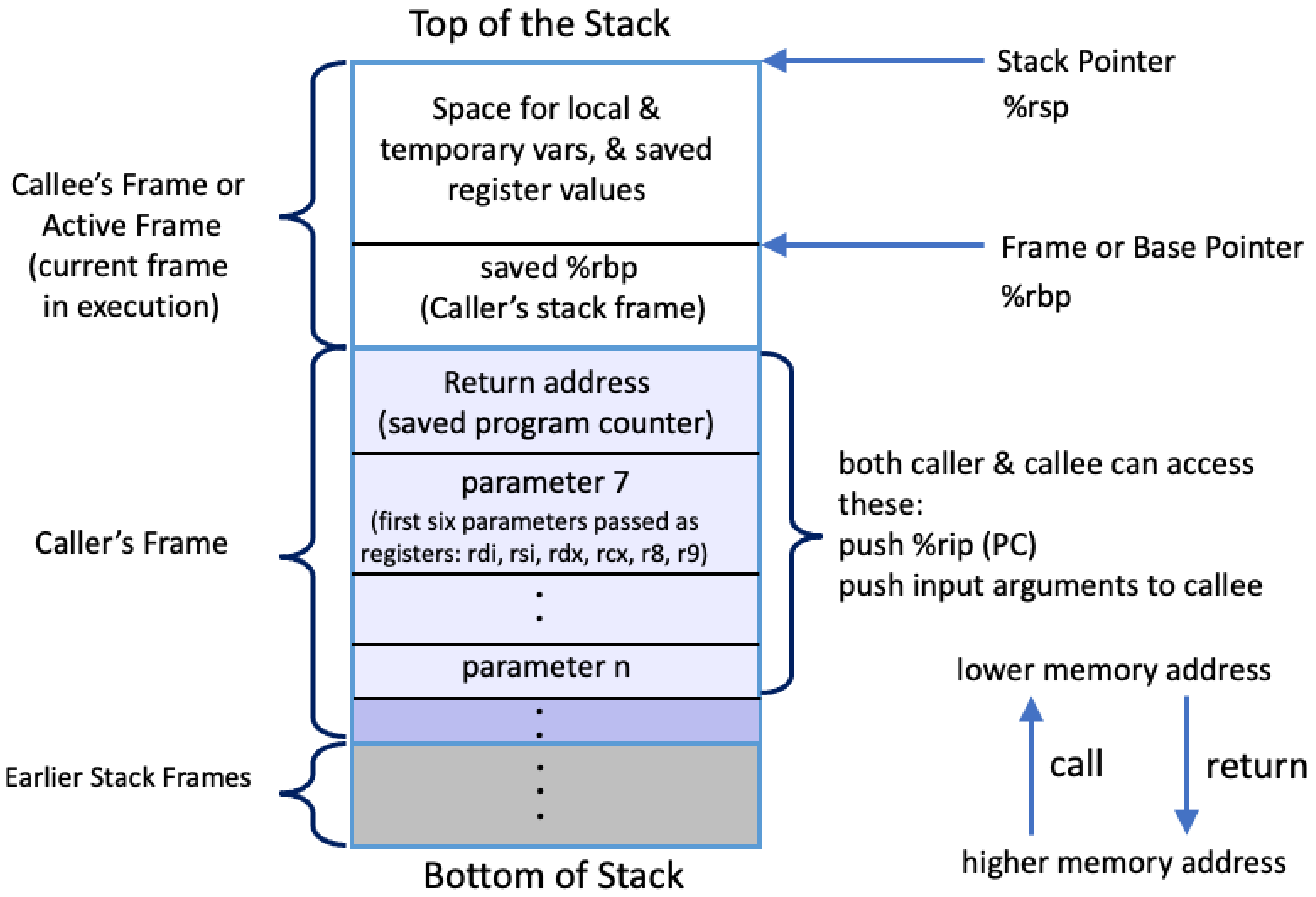
- You could store this stuff wherever you want!
  - The hardware does NOT care.
  - What matters: everyone agrees on where to find the necessary data.

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

- When possible, keep values in registers (why?)
  - Accessing registers is faster than memory (stack)

x86_64 Calling Convention

- The function's return value: In register %rax
- The caller's %rbp value (caller's saved frame pointer)
  - Placed on the stack in the callee's stack frame
- The return address (saved PC value to resume execution on return)
  - Placed on the stack in the caller's stack frame
- Arguments passed to a function:
  - First six passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
  - Any additional arguments stored on the caller's stack frame (shared with callee)


Top of the Stack
Stack Pointer
%rsp
Space for local & temporary vars, & saved register values
Callee's Frame or Active Frame (current frame in execution)
saved %rbp (Caller's stack frame)
Frame or Base Pointer
%rbp
Return address (saved program counter)
parameter 7
(first six parameters passed as registers: rdi, rsi, rdx, rcx, r8, r9)
parameter n
Caller's Frame
both caller & callee can access these:
push %rip (PC)
push input arguments to callee
lower memory address
call
return
higher memory address
Earlier Stack Frames
Bottom of Stack

x86_64 Calling Convention

- The function's return value: In register %rax
- The caller's %rbp value (caller's saved frame pointer)
  - Placed on the stack in the callee's stack frame
- The return address (saved PC value to resume execution on return)
  - Placed on the stack in the caller's stack frame
- Arguments passed to a function:
  - First six passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
  - Any additional arguments stored on the caller's stack frame (shared with callee)

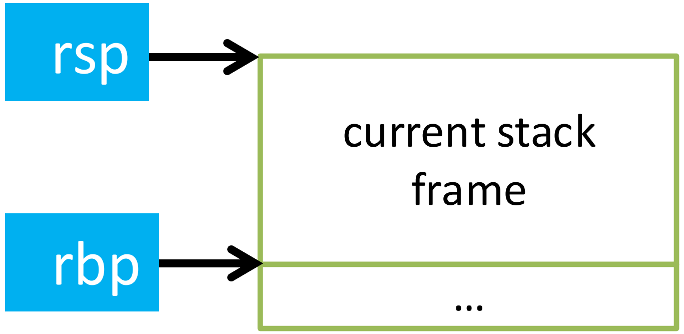
Return Value

- If the callee function produces a result, the caller can find it in %rax

- We saw this when we wrote our function in the weekly lab last friday
  - Copy the result to %rax before we finishing up

Dynamic Stack Accounting

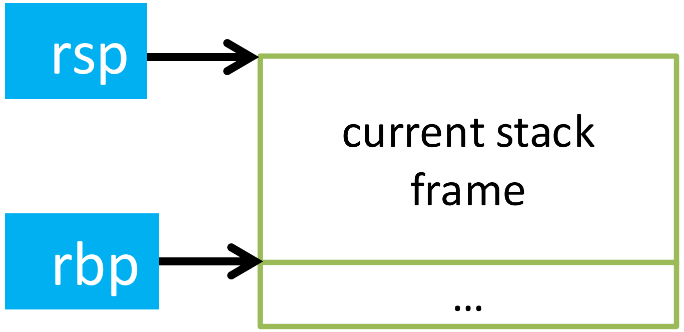
- Dedicate CPU registers for stack bookkeeping
  - %rsp (stack pointer): Top of current stack frame
  - %rbp (frame pointer): Base of current stack frame



- Compiler maintains these pointers
  - Does the compiler know the exact address they point to?
  - Compiler doesn't know or care! (job of the OS to figure that out)
- To the compiler: every variable access is relative to %rsp and %rbp!

Compiler: updates to rsp/rbp on function call/return

invariant:
The current function's stack frame is always between the addresses stored in `rsp` and `rbp`

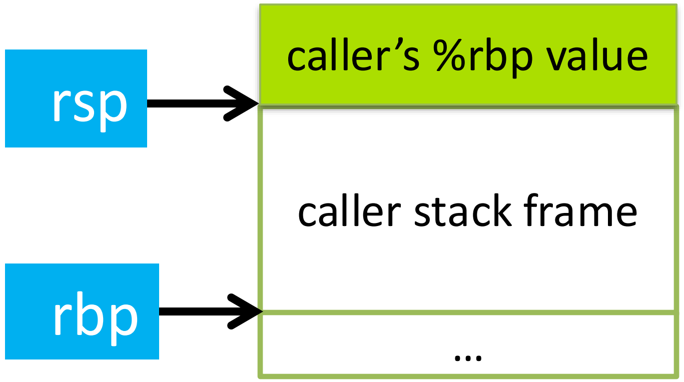


Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp

invariant:
The current function's stack frame is always between the addresses
stored in `rsp` and `rbp`

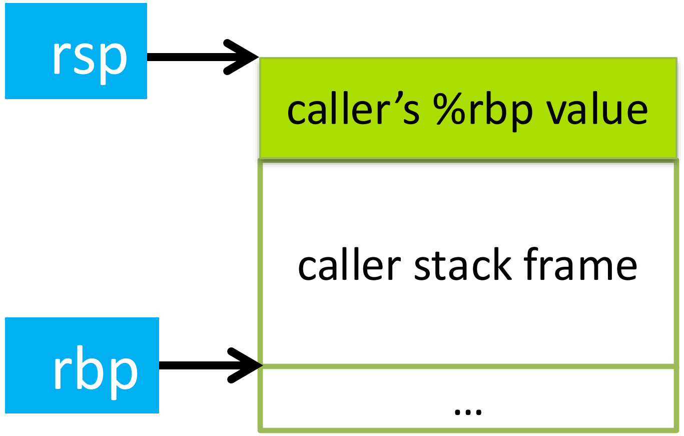


Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp

invariant:
The current function's stack frame is always between the addresses stored in rsp and rbp

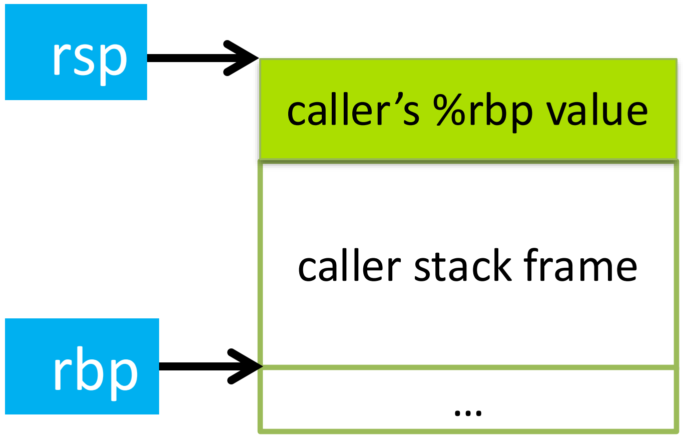


Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp
2. Set %rbp = %rsp

invariant:
The current function's stack frame is always between the addresses
stored in `rsp` and `rbp`

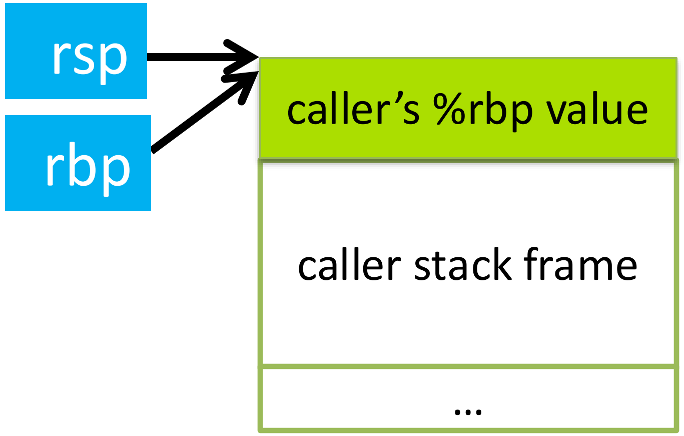


Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp
2. Set %rbp = %rsp

invariant:
The current function's stack frame is always between the addresses stored in `rsp` and `rbp`

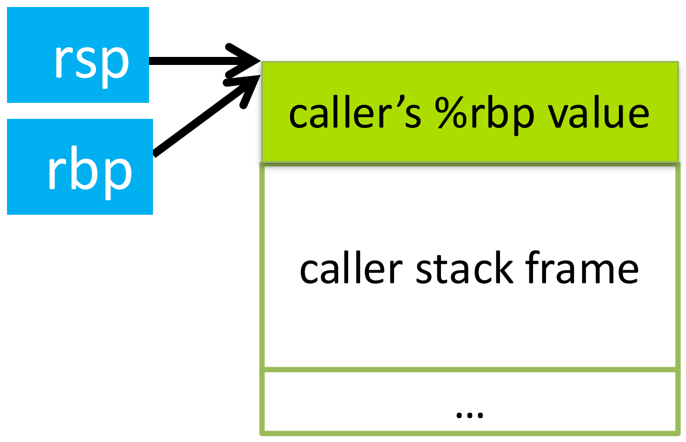


Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp
2. Set %rbp = %rsp
3. Subtract N from %rsp

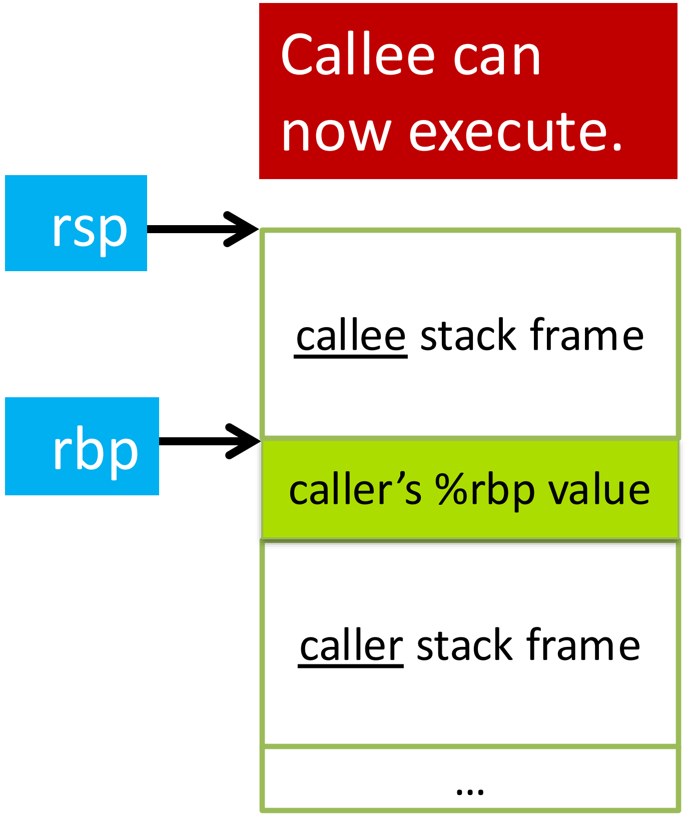
invariant:
The current function's stack frame is always between the addresses stored in rsp and rbp



Compiler: Upon a new Function Call..

Immediately upon calling a new function:

1. push current %rbp
2. Set %rbp = %rsp
3. Subtract N from %rsp



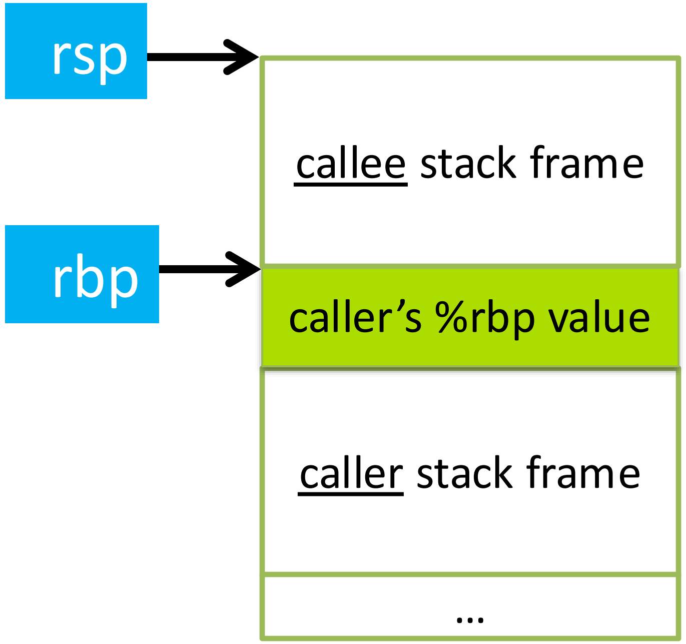
invariant:
The current function's stack frame is always between the addresses stored in rsp and rbp

Compiler: Returning from a function call..

Returning from a function:

1. Set %rsp = %rbp

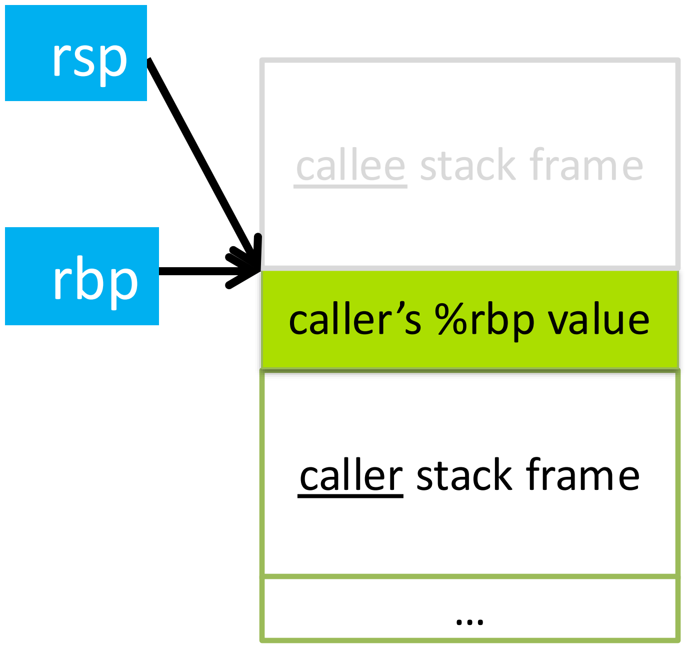
invariant:
The current function's stack frame is always between the addresses stored in rsp and rbp



Compiler: Returning from a function call..

Returning from a function:

1. Set %rsp = %rbp (callee stack frame no longer exists)



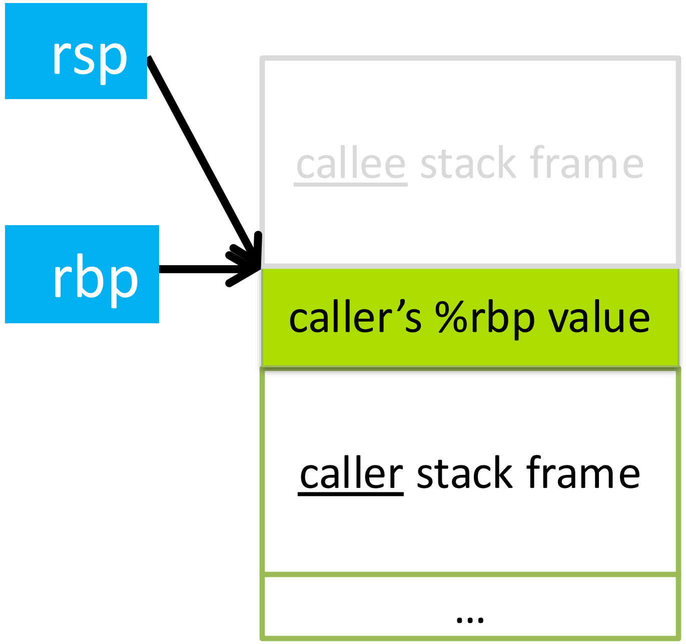
invariant:
The current function's stack frame is always between the addresses stored in rsp and rbp

Compiler: Returning from a function call..

Returning from a function:

1. Set %rsp = %rbp (callee stack frame no longer exists)
2. pop %rbp

invariant:
The current function's stack frame is always between the addresses stored in rsp and rbp



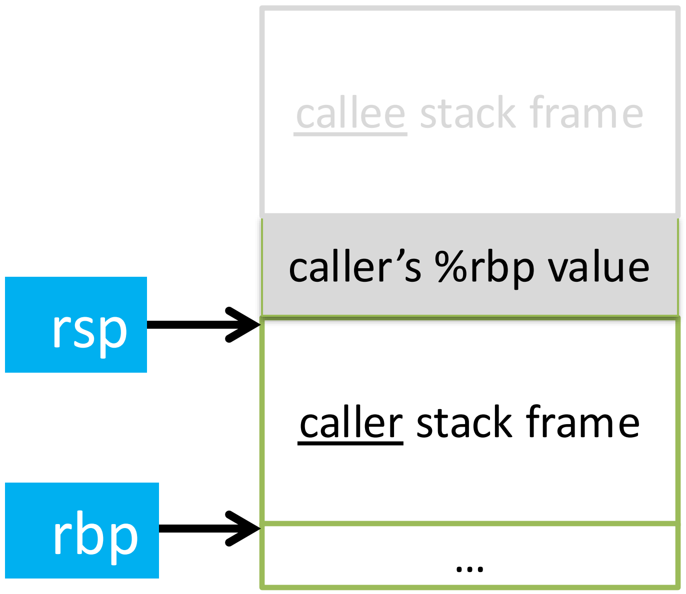
Compiler: Returning from a function call..

Returning from a function:

1. Set %rsp = %rbp
2. pop %rbp
   - pop caller's rbp off the stack and set it to the value of rbp
   - decrement rsp

X86_64 has another convenience instruction for this: leaveq

invariant:
The current function's stack frame is always between the addresses stored in rsp and rbp



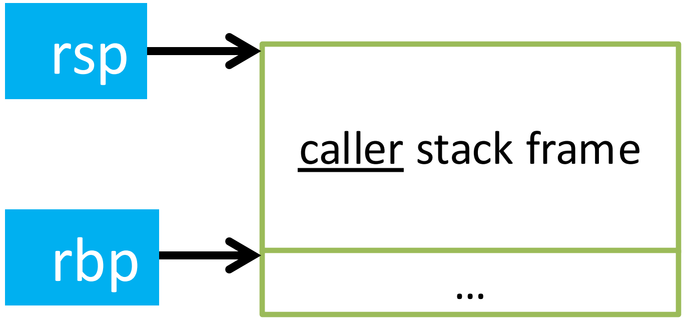
Compiler: Returning from a function call..

Returning from a function:

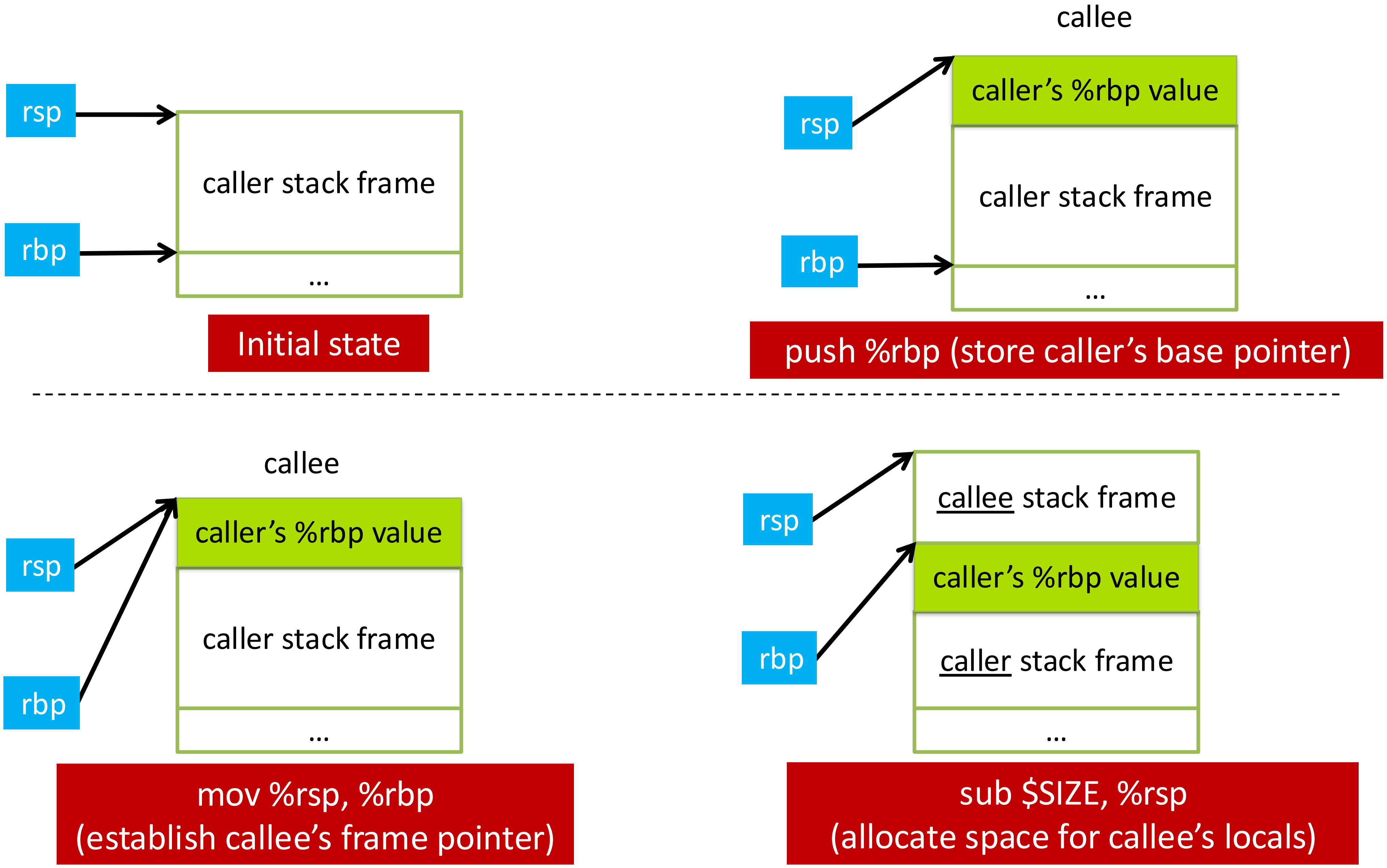
1. Set %rsp = %rbp
2. pop %rbp
   - pop caller's rbp off the stack and set it to the value of rbp
   - decrement rsp

invariant:
The current function's stack frame is always between the addresses stored in rsp and rbp

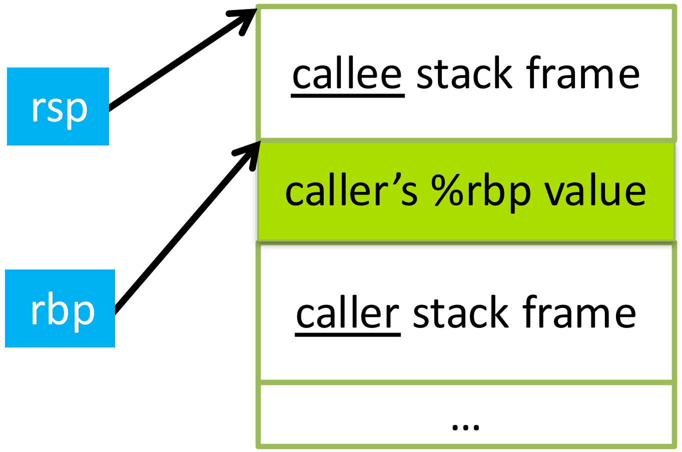
Back to where we started



x86 Calling Conventions: Function Call

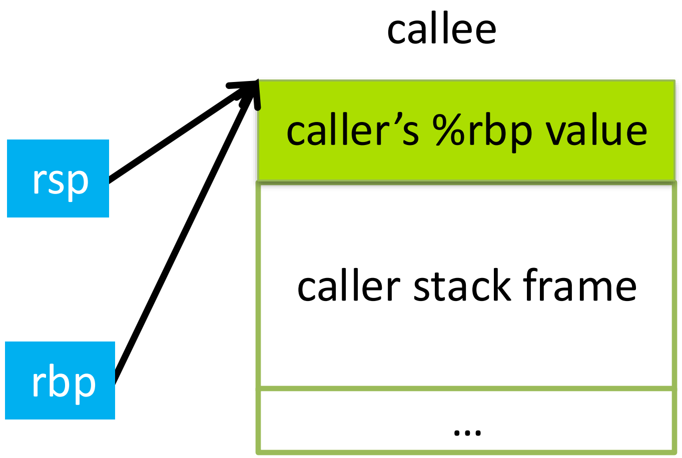


x86 Calling Conventions: Function Return

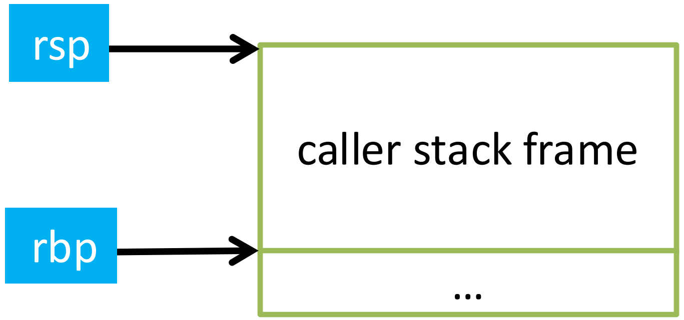


we want to restore the caller's frame

x86_64 provides a convenience instruction that does all of this:
`leaveq`



mov %rbp, %rsp
(restore caller's stack pointer)

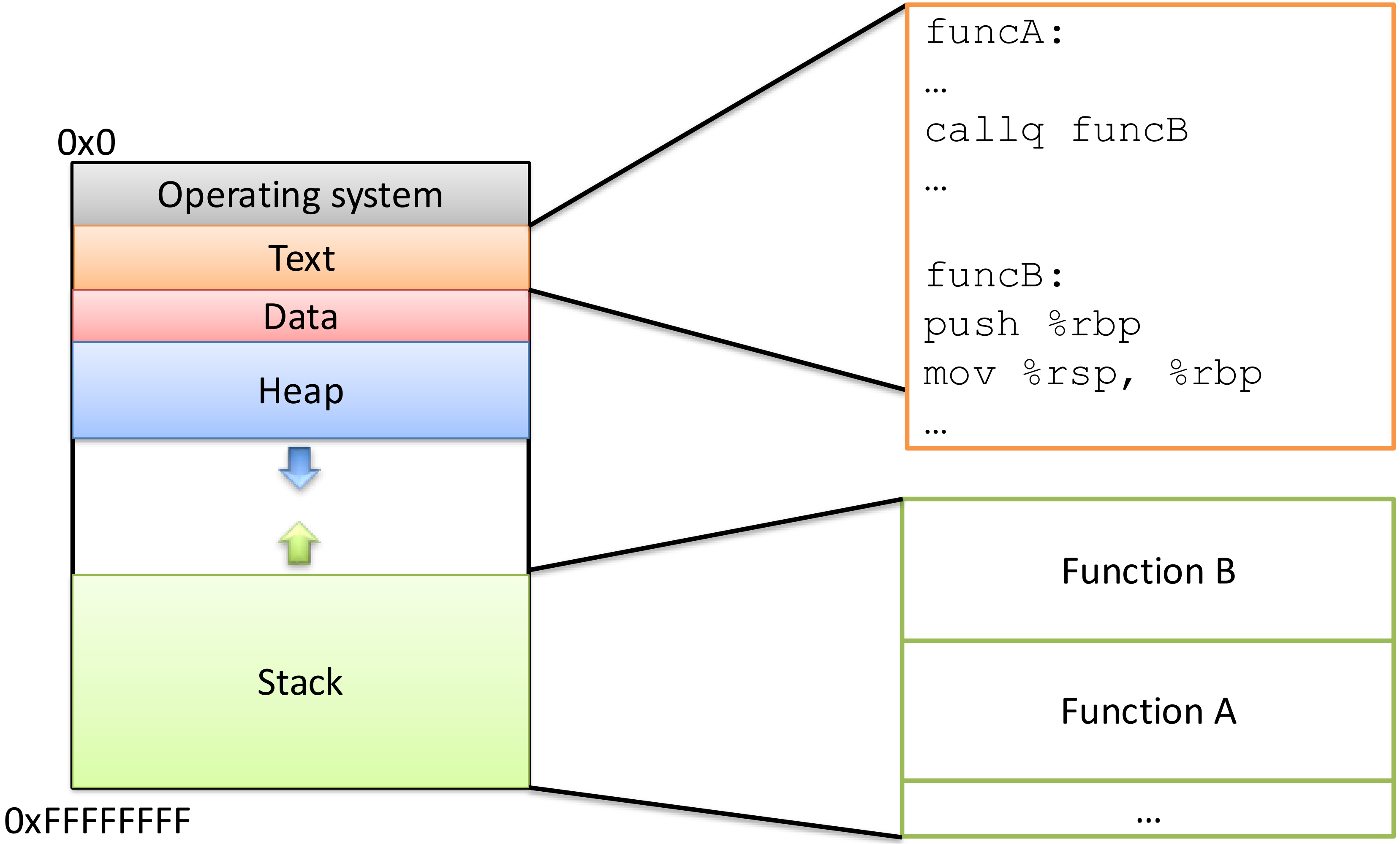


pop %rbp (restore caller's frame pointer)

x86_64 Calling Convention

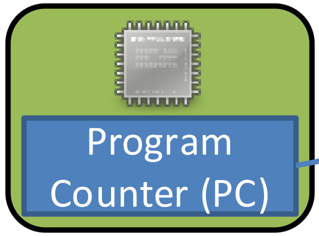
- The function's return value:
  - In register %rax
- The caller's %rbp value (caller's saved frame pointer)
  - Placed on the stack in the callee's stack frame
- The return address (saved PC value to resume execution on return)
  - Placed on the stack in the caller's stack frame
- Arguments passed to a function:
  - First six passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
  - Any additional arguments stored on the caller's stack frame (shared with callee)

Instructions in Memory



Program Counter

Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)



Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

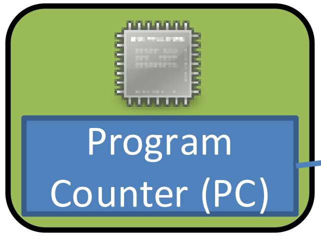
What do we do now?

Follow PC, fetch instruction:

`add $5, %rcx`

Program Counter

Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)



Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

What do we do now?

Follow PC, fetch instruction:

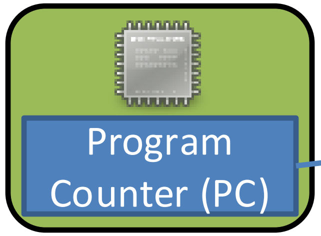
`add $5, %rcx`

Update PC to next instruction.

Execute the `addl`.

Program Counter

Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)



What do we do now?

Follow PC, fetch instruction:

```
mov $rcx, -8(%rbp)
```

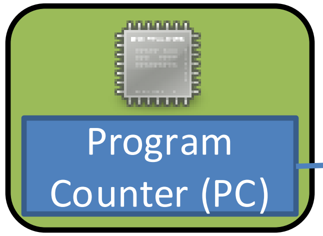
Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Program Counter

Recall: PC stores the address of the next instruction. (A pointer to the next instruction.)



What do we do now?

Follow PC, fetch instruction:

```
mov $rcx, -8(%rbp)
```

Update PC to next instruction.

Execute the `mov`.

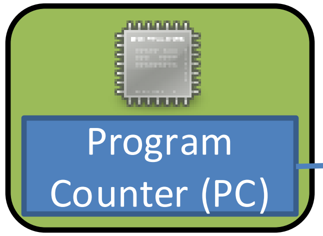
Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Program Counter

Recall: PC stores the address of the next instruction.
(A pointer to the next instruction.)



Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

What do we do now?

Keep executing in a straight line downwards like this until:

We hit a jump instruction.
We call a function.

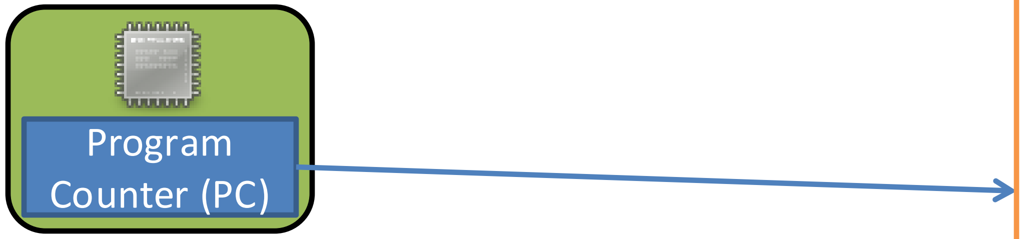
Changing the PC: Jump

- On a (non-function call) jump:
  - Check condition codes
  - Set PC to execute elsewhere (usually not the next instruction)

- Do we ever need to go back to the instruction after the jump?

Maybe (and if so, we'd have a label to jump back to), but usually not.

Changing the PC: Functions



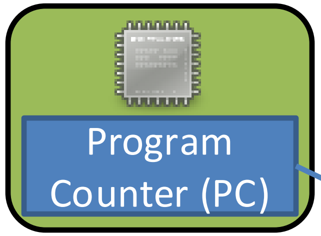
Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

What we'd like this to do:

Changing the PC: Functions



What we'd like this to do:

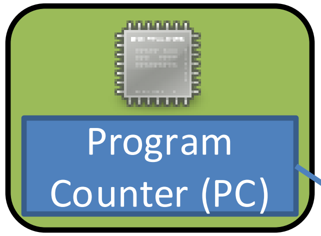
Set up function B's stack.

Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Changing the PC: Functions



What we'd like this to do:

Set up function B's stack.

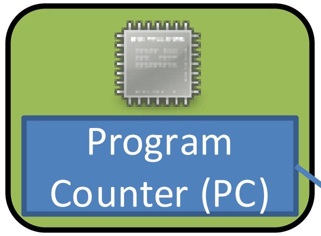
Execute the body of B, produce result (stored in %rax).

Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Changing the PC: Functions



What we'd like this to do:

Set up function B's stack.

Execute the body of B, produce result (stored in %rax).

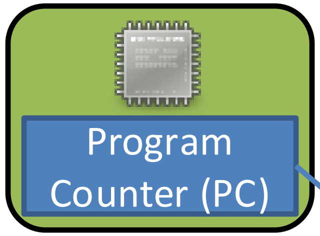
Restore function A's stack.

Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Changing the PC: Functions



What we'd like this to do:

Return:
Go back to what we were doing before funcB started.

Unlike jumping, we intend to go back!

Text Memory Region

```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Like `push`, `pop`, and `leave`, `call` and `ret` are convenience instructions. What should they do to support the PC-changing behavior we need? (The PC is %rip.)

call

In words:

In instructions:

ret

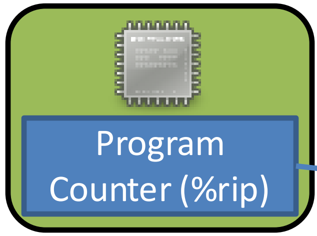
In words:

In instructions:

Functions and the Stack

Executing instruction:
`callq funcB`

PC points to <u>next instruction</u>



Text Memory Region

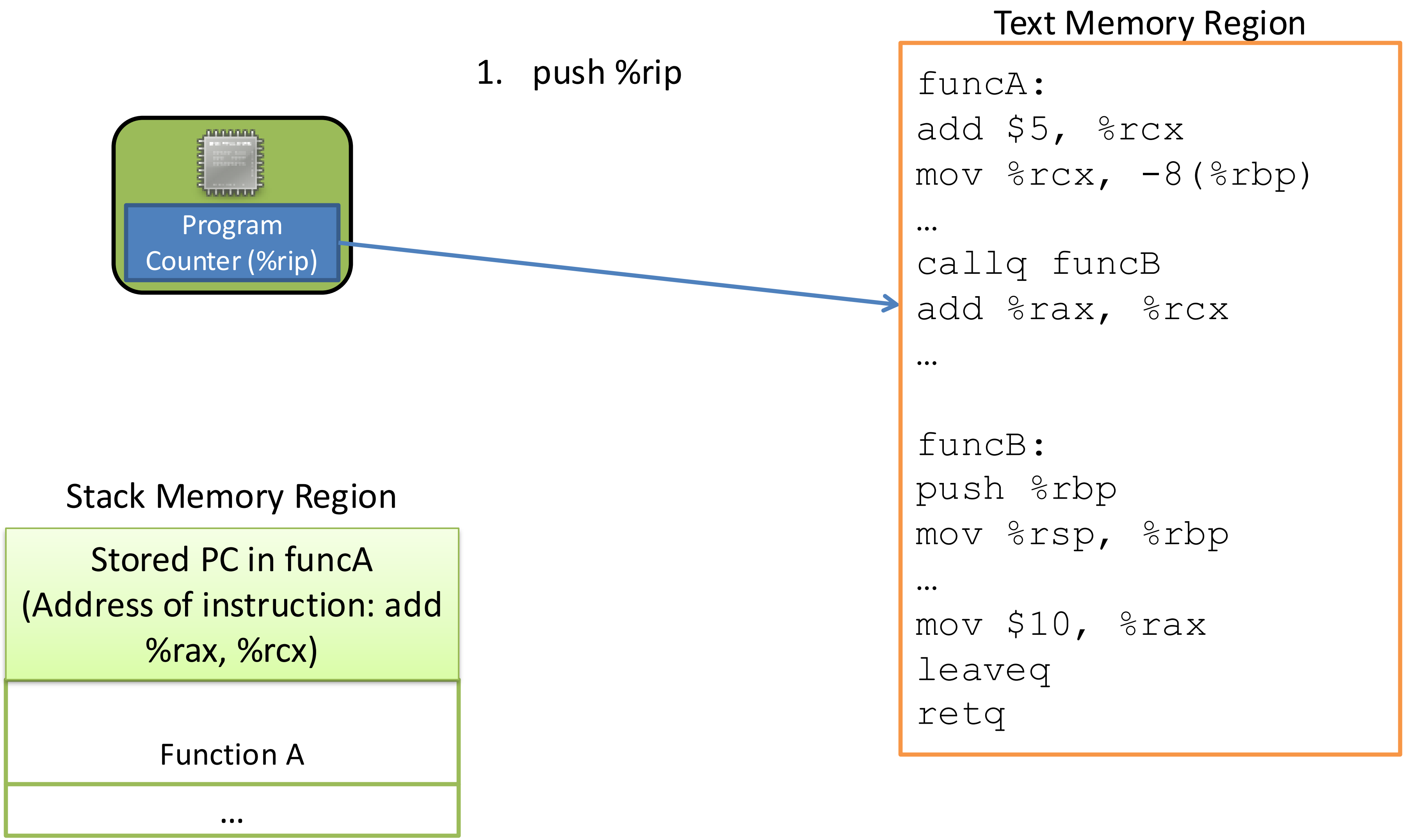
```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Stack Memory Region

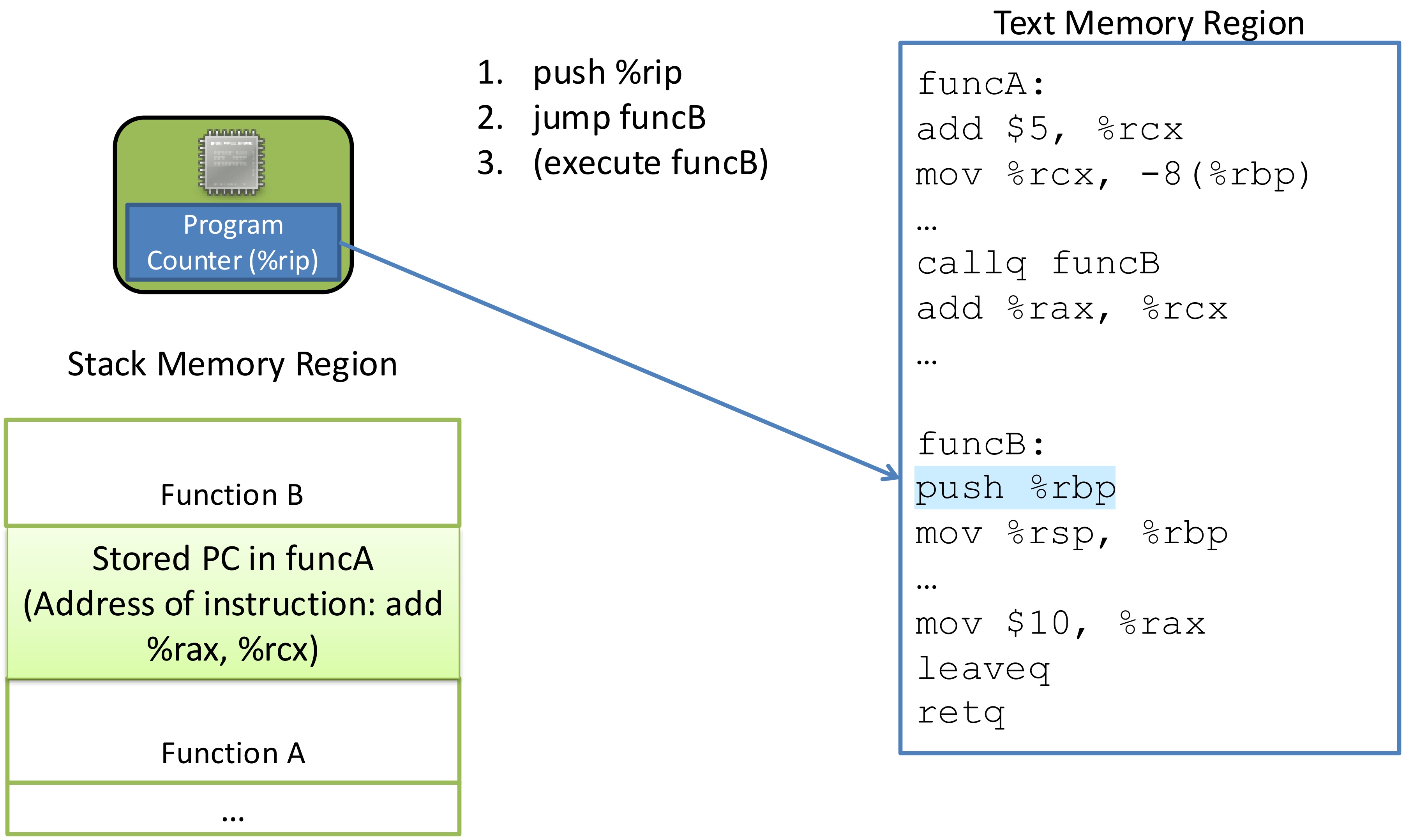
Function A
…

Functions and the Stack



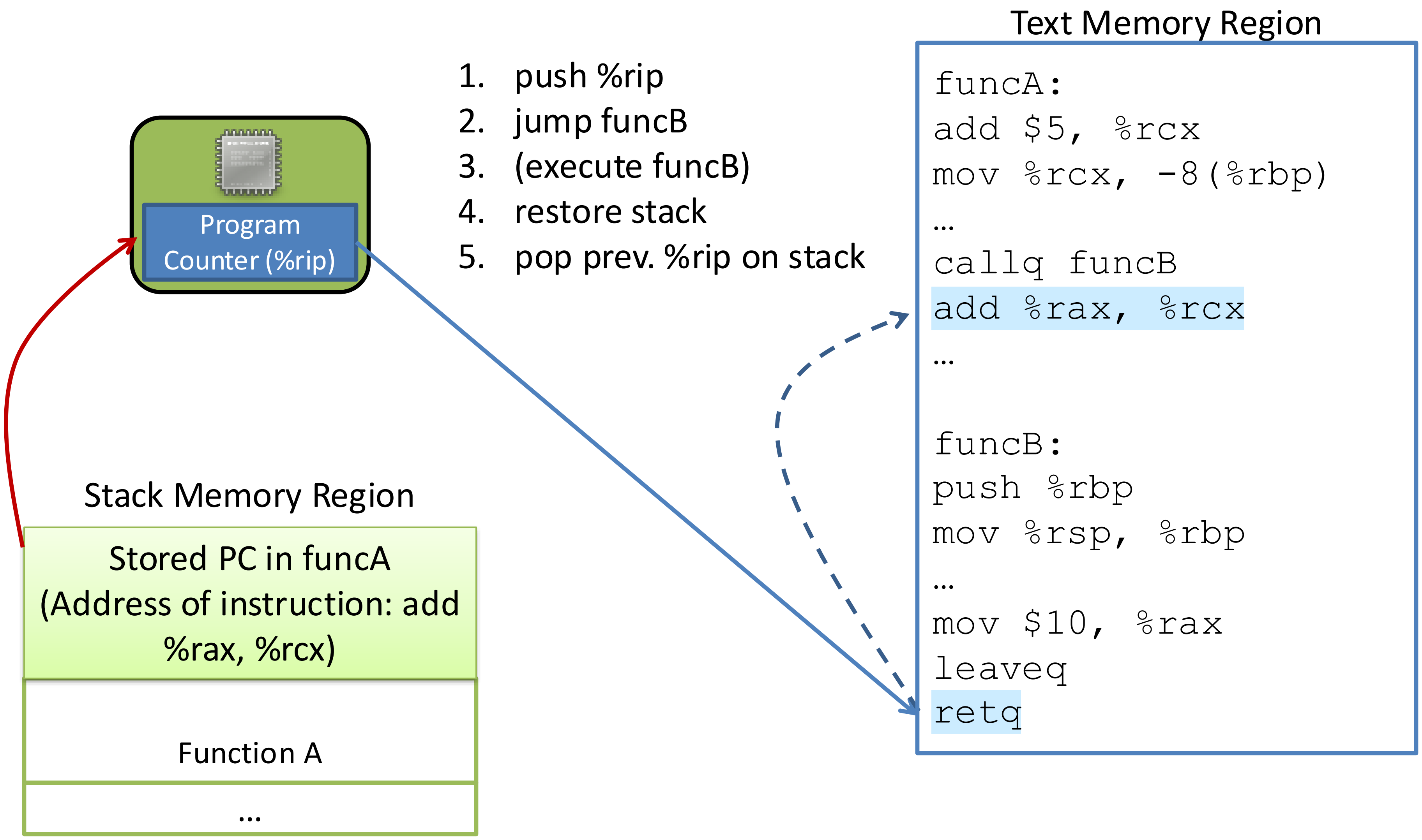
Functions and the Stack

1. push %rip
2. jump funcB
3. (execute funcB)



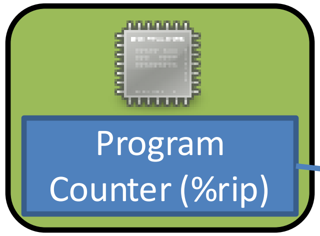
Functions and the Stack

1. push %rip
2. jump funcB
3. (execute funcB)
4. restore stack
5. pop prev. %rip on stack



Functions and the Stack

6. (resume funcA)



Text Memory Region

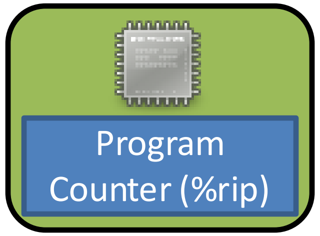
```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Stack Memory Region

Function A
…

Recap: PC upon a Function Call



1. push %rip
2. jump funcB
3. (execute funcB)
4. restore stack
5. pop prev. %rip on stack
6. (resume funcA)

Text Memory Region

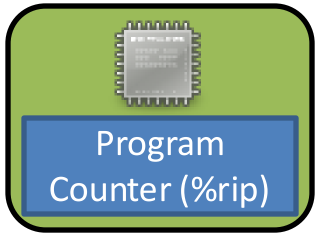
```
funcA:
add $5, %rcx
mov %rcx, -8(%rbp)
…
callq funcB
add %rax, %rcx
…

funcB:
push %rbp
mov %rsp, %rbp
…
mov $10, %rax
leaveq
retq
```

Stack Memory Region

Stored PC in funcA (Address of instruction: add %rax, %rcx)
Function A
…

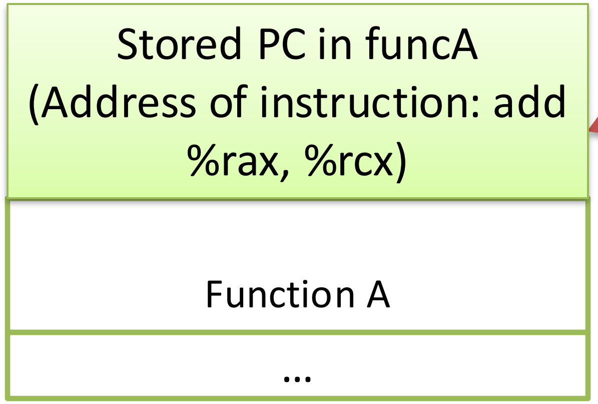
Functions and the Stack



1. push %rip
2. jump funcB
3. (execute funcB)
4. restore stack
5. pop prev. %rip on stack
6. (resume funcA)

Steps 1–2: callq
Step 4: leaveq
Step 5: retq

Stack Memory Region



Return address:

Address of the instruction we should jump back to when we finish (return from) the currently executing function.

x86_64 Stack / Function Call Instructions

`push`	Create space on the stack and place the source there.	`sub $8, %rsp` `mov src, (%rsp)`
`pop`	Remove the top item off the stack and store it at the destination.	`mov (%rsp), dst` `add $8, %rsp`
`callq`	1. Push return address on stack 2. Jump to start of function	`push %rip` `jmp target`
`leaveq`	Prepare the stack for return (restoring caller's stack frame)	`mov %rbp, %rsp` `pop %rbp`
`retq`	Return to the caller, PC ← saved PC (pop return address off the stack into PC (rip))	`pop %rip`

x86_64 Calling Convention

- The function's return value:
  - In register %rax
- The caller's %rbp value (caller's saved frame pointer)
  - Placed on the stack in the callee's stack frame
- The return address (saved PC value to resume execution on return)
  - Placed on the stack in the caller's stack frame
- Arguments passed to a function:
  - First six passed in registers (%rdi, %rsi, %rdx, %rcx, %r8, %r9)
  - Any additional arguments stored on the caller's stack frame (shared with callee)

Function Arguments

- Most functions don't receive more than 6 arguments, so x86_64 can simply use registers most of the time.

- If we *do* have more than 6 arguments though (e.g., perhaps a `printf` with lots of placeholders), we can't fit them all in registers.

- In that case, we need to store the extra arguments on the stack.
  By convention, they go in the caller's stack frame.

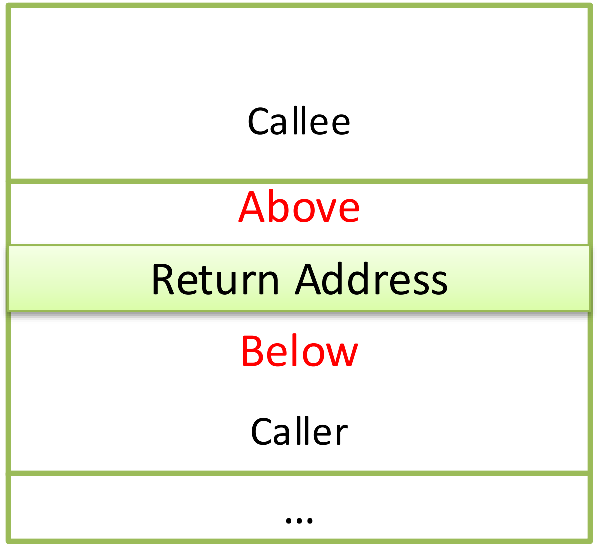
If we need to place arguments in the caller's stack frame, should they go above or below the return address?

A. Above

B. Below

C. It doesn't matter

D. Somewhere else



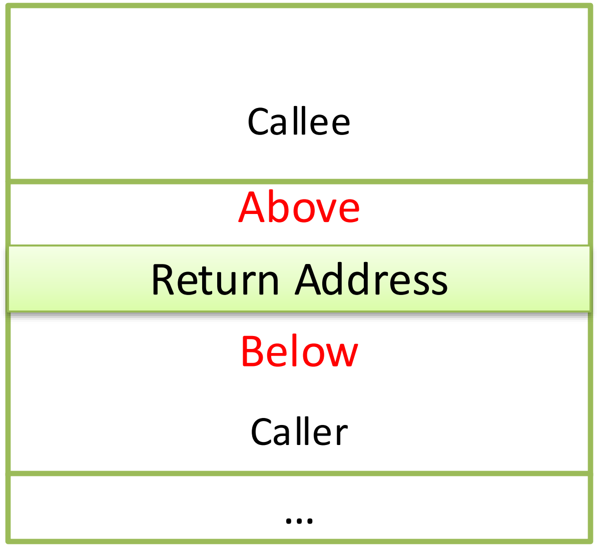
If we need to place arguments in the caller's stack frame, should they go above or below the return address?

A. Above

B. Below

C. It doesn't matter

D. Somewhere else



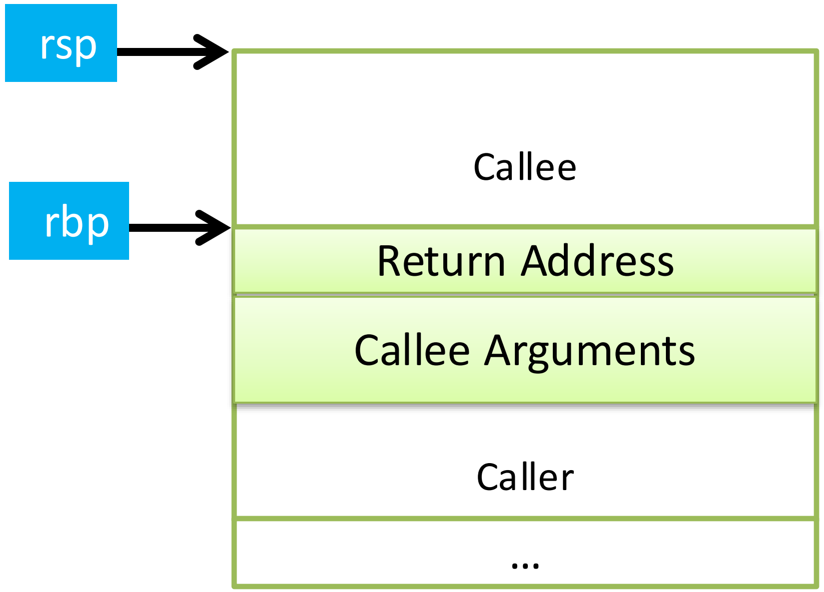
x86_64 Stack / Function Call Instructions

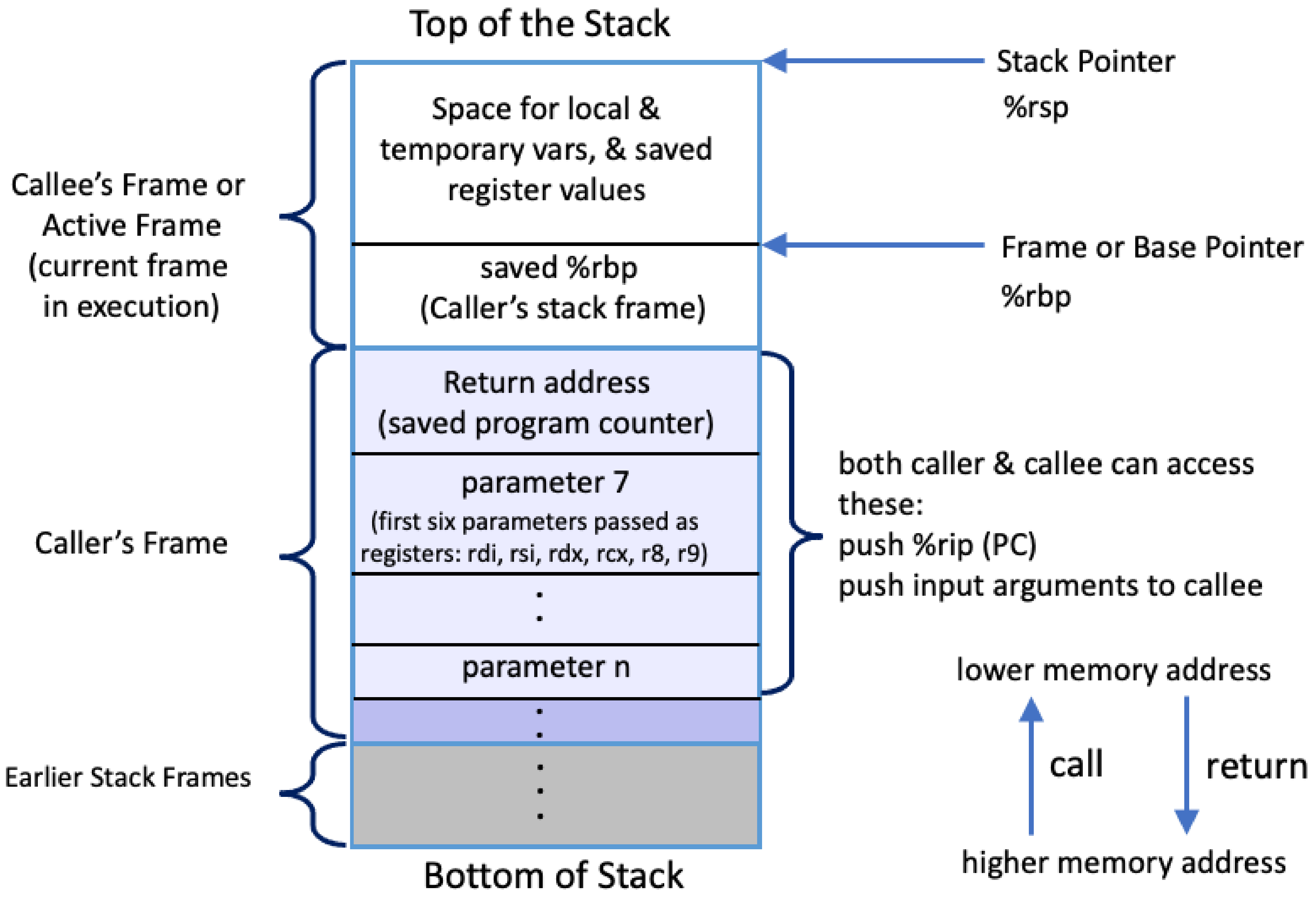
`push`	Create space on the stack and place the source there.	`sub $8, %rsp` `mov src, (%rsp)`
`pop`	Remove the top item off the stack and store it at the destination.	`mov (%rsp), dst` `add $8, %rsp`
`callq`	1. Push return address on stack 2. Jump to start of function	`push %rip` `jmp target`
`leaveq`	Prepare the stack for return (restoring caller's stack frame)	`mov %rbp, %rsp` `pop %rbp`
`retq`	Return to the caller, PC ← saved PC (pop return address off the stack into PC (rip))	`pop %rip`

Arguments

- Extra arguments to the callee are stored just underneath the return address.
- Does it matter what order we store the arguments in?
- Not really, as long as we're consistent (follow conventions).

This is why arguments can be found at positive offsets relative to %rbp.



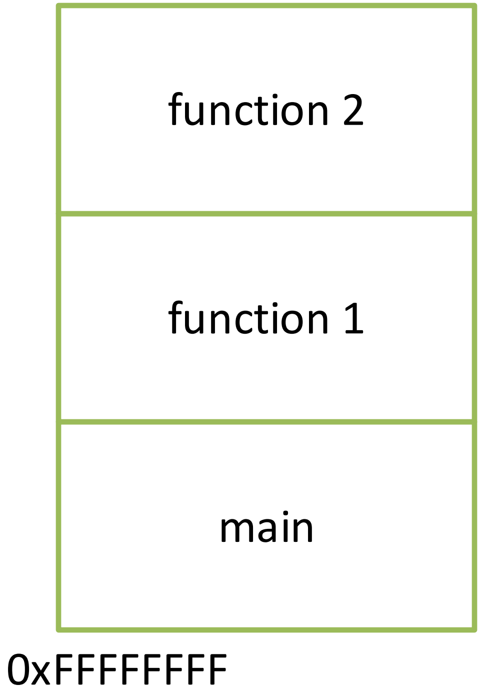

Top of the Stack
Stack Pointer
%rsp
Space for local & temporary vars, & saved register values
Callee's Frame or Active Frame (current frame in execution)
saved %rbp
(Caller's stack frame)
Frame or Base Pointer
%rbp
Return address
(saved program counter)
parameter 7
(first six parameters passed as registers: rdi, rsi, rdx, rcx, r8, r9)
parameter n
Caller's Frame
both caller & callee can access these:
push %rip (PC)
push input arguments to callee
lower memory address
call
return
higher memory address
Earlier Stack Frames
Bottom of Stack

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: %rax, %rdi, %rsi, %rdx, %rcx, %r8, %r9, %r10, %r11

Caller must save them prior to calling callee
callee free to trash these,
Caller will restore if needed

Callee-saved: %rbx, %r12, %r13, %r14, %r15

Callee 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 16 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
  - This is getting to the limits of CS 31!
    - – take CS 75 (compilers) for more details.

Up next…

- Connecting Arrays, Structs, and Pointers with assembly