## Lecture J:

# x86-64 Review/Intro

CSE401/501m:

Introduction to Compiler Construction

*Instructor: Gilbert Bernstein*

# Administrivia

- Short hw3 due tonight
- Midterm on Friday in Class — topics & old exams online; blank 5x8 index card available at the end of class
  - Review in sections this week; come with questions!
- Checking Project **due May 20**
  - **Project released online** — please look at it early. Make sure to **get lots done this weekend or early next week**, or you will be in trouble!
  - There is a required check-in to review data structure design (symbol tables, classes for types, methods, etc.) during next Thursday section — worth a point on the project grades.

# Administrivia (Wed)

- Midterm in Class (Friday!)
  - Everything up to and including Checking (not IRs)
  - You can have **one** 5x8 notecard with any **hand-written** notes that you want
  - Topic list and exams on the web
  - Review in sections tomorrow — **Come with questions!**
- HW 3 sample solutions + blank 5x8 notecards available after class

# Administrivia (Wed — slide 2)

- Checking project due in just under 2 weeks!
  - Don't forget about it! Start right after the exam!
  - Sections next week are a project work-session
  - Required Check-in with TAs (1pt) to make sure your Symbol Table and Type data structures are reasonable designs
  - Attend section with your partner, even if you normally attend different sections!

# Outline

x86

Assembly Language

Memory Model

Arithmetic & Data Movement

Control Flow, Jumps & Branches

Function Calls

# Outline

**x86**

Assembly Language

Memory Model

Arithmetic & Data Movement

Control Flow, Jumps & Branches

Function Calls

# Some x86-64 Resources

## From most to least helpful for you in this course

- x86-64 Instructions and ABI Summary
  - Handout for University of Chicago CMSC 22620, Spring 2009 by John Reppy
- x86-64 Machine-Level Programming
  - Earlier version of sec. 3.13 of Computer Systems: A Programmer's Perspective, 2nd ed. by Bryant & O-Hallaron (CSE 351 textbook)
- Intel architecture processor manuals
  - Undoubtedly way more than we'll need

# x86 Selected History

- Over 45 Years of x86
  - 1978: 8086 ———————— 16-bit, 5 MHz, $3\mu m$, segmented memory
  - 1982: 80286 ——————— protected mode, floating point
  - 1985: 80386 ——————— 32-bit, VM, 8 "general" registers
  - 1993: Pentium —————— MMX
  - 1999: Pentium III ———— SSE
  - 2000: Pentium IV ———— SSE2, SSE3, Hyperthreading
  - 2006: Core Duo, Core 2 —— Multicore, SSE4, x86-64
  - 2013: Haswell —————— 64-bit, 4-8 core, ~3 GHz, 22 nm, AVX2
  - …
- Many micro-architecture changes over the years
  - pipelining, super-scalar, out-of-order, caching, multicore, …

# Backwards-Compatibility

- Current processors can run 8086 code
  - You can get VisiCalc 1.0 on the web & run it!!!
  - Until 2023, UW's HR and financial infrastructure ran on code from the 80s!
- The full ISA is incredibly complicated because no instruction or feature is ever removed
  - In the processor, this is all translated into internal micro-code, implementing a RISC-like core
- We will focus on basic 64-bit instructions and ignore the rest of the ISA — only target a small set of simple instructions

# x86-64 Main Features

- 16 64-bit general registers (i.e. uses 64-bit integers)
  - *(note: int is 32-bits usually, and long 64 bits ¯_(ツ)_/¯ )*
- 64-bit address space; pointers are 8 bytes
- 16 SSE registers for SIMD & floating point
- Register-based function call conventions (the ABI)
- Additional addressing modes (e.g. pc relative)
- 32-bit legacy mode
- some pruning of old features

# Outline

x86

**Assembly Language**

Memory Model

Arithmetic & Data Movement

Control Flow, Jumps & Branches

Function Calls

# Processor Languages

- What two fundamental things do you need to know in order to learn a new programming language?
  - What is the model of memory/data?
  - What are the operations you can perform on that data?
- Instruction Set Architectures (ISAs) are *just languages*, whether presented via binary or via assembly
- btw, the automata we used earlier follow the same idea
  - FSM — memory is a finite set of states; transition function specifies how to interpret inputs (instructions)
  - PDA — same idea but augment memory with a stack

# x86-64 Assembler Language

- This is the target for our project
- x86-64 has a standard assembly language, but *the nice thing about standards is…*
- There are two main syntactic variants for x86-64
    - Intel/Microsoft syntax — used in the Intel docs
    - AT&T/GNU syntax — what we're generating, what's in the linked handouts and in the 351 book
    - *note: you can use* `gcc -S` *to generate asm code from C/C++ code to see more examples*
- **These slides use gcc/AT&T/GNU syntax**

# Intel vs. GNU Assembler

- Summary of key differences between Intel docs and gcc assembly

| | Intel/Microsoft asm | AT&T/GNU asm |
|---|---|---|
| Operand order: `op a,b` | `a = a op b (dst first)` | `b = a op b (dst last)` |
| Memory address | `[baseregister+offset]` | `offset(baseregister)` |
| Instruction mnemonics | `mov, add, push, …` | `movq, addq, pushq` [explicit operand size added to end] |
| Register names | `rax, rbx, rbp, rsp, …` | `%rax, %rbx, %rbp, %rsp, …` |
| Constants | `17, 42` | $17, $14 |
| Comments | `; to end of line` | `# to end of line` or `/* … */` |

- Intel docs include many complex, historical instructions and artifacts that aren't commonly used by modern compilers — and we won't use them either

# Outline

x86

Assembly Language

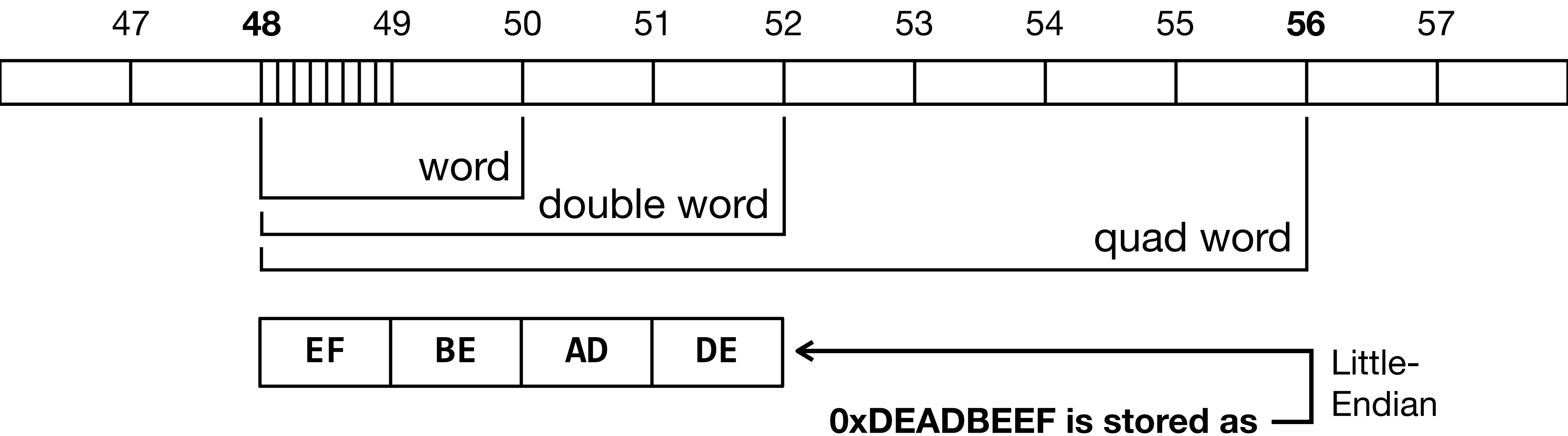
**Memory Model**

Arithmetic & Data Movement

Control Flow, Jumps & Branches

Function Calls

# x86-64 Memory Model




- 8-bit bytes, byte addressable
- terminology for 16-, 32-, 64-bit “words”
  - hence the ‘q’ in instructions like `movq`, `addq`, etc.
- Data should be aligned to “natural” boundaries. Unaligned accesses are usually supported, but can incur large performance penalties
- Little-endian — low-order byte of integers goes in lowest address

# x86-64 Registers

- 16 64-bit general registers
- Registers can be used as 64-bit integers or pointers, or as 32-bit integers
  - You can reference sub-words/double-words, but we won't
- To simplify our project, we'll use only 64-bit data (ints, pointers, even booleans!)

| | |
|---|---|
| `%rax` | *orig. general reg.* |
| `%rbx` | *orig. general reg.* |
| `%rcx` | *orig. general reg.* |
| `%rdx` | *orig. general reg.* |
| `%rsp` | **s**tack **p**ointer |
| `%rbp` | **b**ase (aka. frame) **p**ointer |
| `%rsi` | **s**ource **i**ndex; *generic use* |
| `%rdi` | **d**estination **i**ndex; *generic use* |
| `%r8` | *8th register* |
| `%r9` | *9th register* |
| … | |
| `%r14` | *14th register* |
| `%r15` | *15th register* |

# Processor Fetch/Execute Cycle

- Basic idea

```
while (running) {
  fetch instruction at %rip address
  %rip ← %rip + instruction length
  execute instruction
}
```

- Sequential execution unless a jump instruction modifies the address stored in `%rip`
  - `%rip` is a hidden register. It cannot be directly accessed from assembly code, but still part of the memory/state model of the machine.

# Format Of Assembly File

- Most lines are
  - blank / comment
  - **an instruction**
  - a label (a possible place to jump to)
- …inside a `.text` segment
- Assembly files also have `.data` segments, and other `.xyz` directives
  - We will revisit these later in the context of the project

# Instruction Format

- Typical data manipulation instruction

```
opcode  src,dst    # comment
```

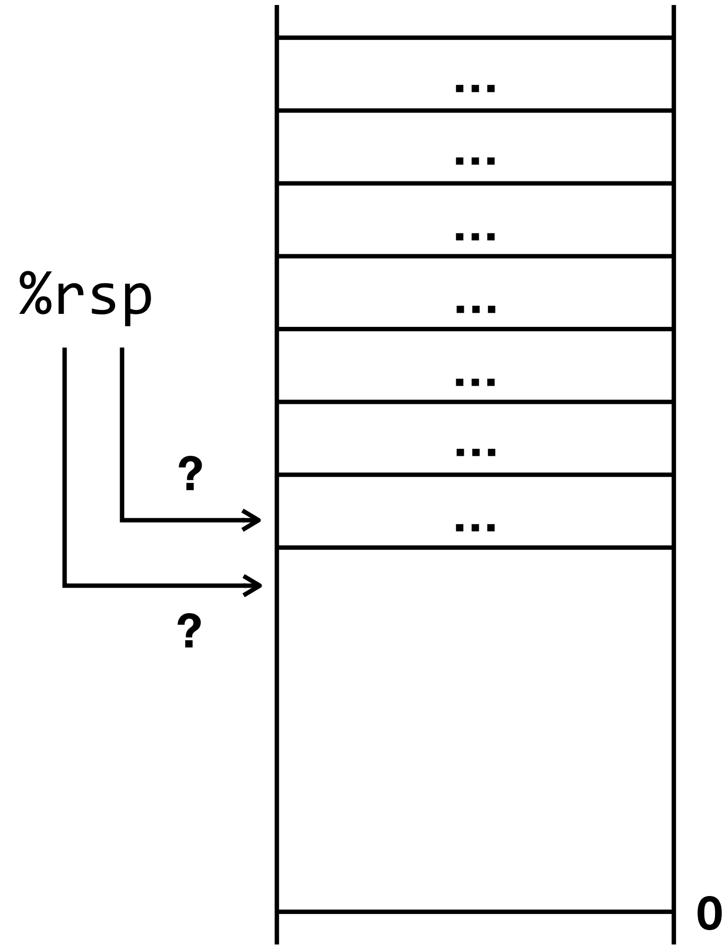
- Meaning is

```
dst ← dst op src
```

- Operands — registers, memory locations, or constants
  - ✦ constants — may be used as `src`, but not as `dst`
  - ✦ `memory locations` — `dst` or `src`, but not both at once
  - ✦ registers — no restrictions on use

# x86-64 Memory Stack

- Register %rsp points to the *"top"* of the stack; (though stack grows *down* — towards lower addresses)
- %rsp is dedicated to this use; don't use for any other purpose
- Points to the last 64-bit quadword *pushed onto the stack*, not the *next "free" quadword*.
- You ***must ensure*** this address remains (8-byte) aligned, and ***should ensure*** that it is 16-byte aligned when performing a function call




**Part of the Application Binary Interface (ABI), not the ISA**

# Stack Instructions

- `pushq src`

```
        %rsp ← %rsp - 8
memory[%rsp] ← src
```

  - meaning is to push `src` onto the stack

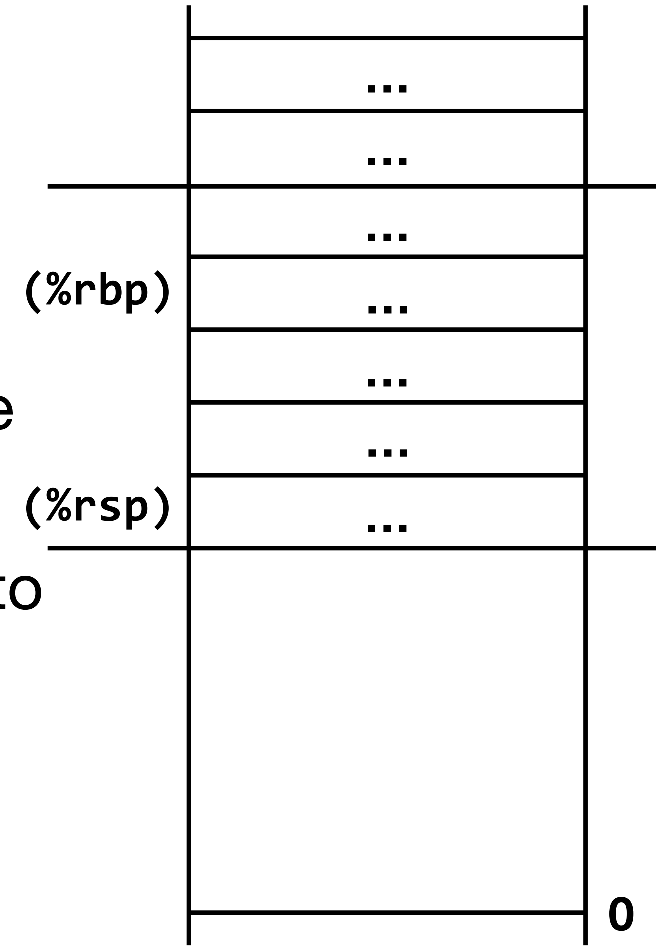
- `popq dst`

```
 dst ← memory[%rsp]
%rsp ← %rsp + 8
```

  - meaning is to pop the top of the stack into `dst`

# Stack *Frames*

- When a method is called, a **stack frame** is allocated on the "top" of the stack
- Frame is popped on method return
- By convention `%rbp` (base pointer) points to a *known offset* in the currently active stack frame (which offset? see ABI)
  - local variables are stored/accessed relative to `%rbp`
- Use of a base pointer is common in 32-bit x86 code; less so in x86-64 code (rely on a fixed-size stack frame instead, and index relative to `%rsp`)
  - Just use `%rbp` in our project to simplify things



# Operand Address Modes

## Cheat Sheet

- These common cases will cover most of what you need

```
movq $17,%rax          # store 17 in %rax
movq %rcx,%rax         # copy %rcx to %rax
movq 16(%rbp),%rax     # copy memory[%rbp + 16] to %rax
movq %rax,-24(%rbp)    # copy %rax to memory[%rbp - 24]
```

- References to object fields work similarly — put the object's memory address into a register and use that address plus an offset
- Remember! — can only use this for `src` or `dst`, not both

# Operand Address Modes

## in general…

- General forms of memory addressing *(note: `s` must be one of 2,4,8)*

| Syntax | Address |
|---|---|
| `(reg)` | *reg* |
| `d(reg)` | *reg + d* |
| `d(reg,s)` | *(s • reg) + d* |
| `d(reg1,reg2,s)` | *reg1 + (s • reg2) + d* |

- Main use of general form is for array subscripting *(or small computations — if the compiler is clever)*
- e.g. — suppose we want to perform an array access **`A[i]`**, where **`A`** holds 8-byte (64-bit) values, the address of **`A`** is in `%rcx` and the value of **`i`** is in `%rax`…

```
0(%rcx,%rax,8)
```

# Load Effective Address

- Essentially the unary & (address of) operator in C/C++

```
leaq src,dst
```

```
dst ← address of src
```

- `dst` must be a register
- Address of `src` includes any address arithmetic or indexing used
- Useful to capture addresses for pointers, parameters, etc.
- Also useful for computing arithmetic that matches the pattern *const + r1 + scale * r2*

# Outline

x86

Assembly Language

Memory Model

**Arithmetic & Data Movement**

Control Flow, Jumps & Branches

Function Calls

# Basic Data Movement and Arithmetic Instructions

```
movq src,dst
```
dst ← dst + src

```
incq dst
```
dst ← dst + 1

```
addq src,dst
```
dst ← dst + src

```
decq dst
```
dst ← dst - 1

```
subq src,dst
```
dst ← dst - src

```
negq dst
```
dst ← -dst

*note: 2s complement arithmetic negation*

# Integer Multiply & Divide

```
imulq src,dst
```

```
dst ← dst * src
```

*dst must be a register*

**c**onvert **q**uad **t**o **o**ct

```
cqto
```

```
%rdx:%rax ← %rax
```

*creates a sign-extended 128-bit "oct" word version of the contents of %rax*

```
idivq src
```

```
let N = %rdx:%rax
let Q = N / src
let R = N % src
%rax ← Q (quotient)
%rdx ← R (remainder)
```

**No Division in MiniJava!**

# Bitwise Operations

```
andq src,dst
```

```
dst ← dst & src
```

```
xorq src,dst
```

```
dst ← dst ^ src
```

```
orq src,dst
```

```
dst ← dst | src
```

```
notq dst
```

```
dst ← ~dst
```

*logical negation, a.k.a. 1s complement*

# Shifts & Rotates

`shlq dst,count`

`dst` ← `dst` shifted left by `count` bits (0 fill)

`shrq dst,count`

`dst` ← `dst` shifted right by `count` bits (0 fill)

`sarq dst,count`

`dst` ← `dst` shifted right by count bits (sign fill)

`rolq dst,count`

`dst` ← `dst` rotated left by `count` bits

`rorq dst,count`

`dst` ← `dst` rotated right by `count` bits

# Uses for Shift & Rotate

- Can be used to optimize multiplication and division by small constants (e.g. x / 2, x / 4, x / 8, etc.)
  - see **Hacker's Delight.** *Henry Warren. Addison-Wesley Professional. 2nd edition (2012)*
- Be careful when using tricks — many edge cases may not work
- Useful for compressing and decompressing encodings of data (e.g. packing many values into a single 64-bit value)
  - e.g. pixel colors and many other optimizations in graphics code

# Outline

x86

Assembly Language

Memory Model

Arithmetic & Data Movement

**Control Flow, Jumps & Branches**

Function Calls

# Control Flow — Jump/Go-to

- In assembly, all control flow is handled with go-to (jump) and conditional go-to (branch)
- Loops and conditional statements are built from these primitives
- *Note* — truly random jumps tank performance. Modern CPUs have very deep pipelines that rely on highly accurate branch predictors b/c the cost of a wrong branch is very high! (mostly hard to address as a compiler optimization)

# Unconditional Jumps

`jmp dst`

`%rip ← address of dst`

- `dst` is usually a label in the code (which can be on a line by itself)
- `dst` address can also be indirect using the address in a reigster or memory location. This uses special syntax
  - `jump *reg` — jump to addr. stored in `reg`
  - `jump *(reg)` — jump to addr. stored in `mem[reg]`
- Can be used for method calls or switch statements

# Compare and Jump Conditionally

- What we want — an instruction that compares two operands and jumps if the comparison succeeds
  - e.g. a "jump to LABEL if REG1 is less than REG2" instruction
- You can't always get what you want — this instruction would take *three* operands rather than just *two*; not allowed in x86-64!
  - e.g. `jump_less_than op1,op2,label`
- Furthermore, we might have already computed `op1 - op2`. It would be great if we could ***reuse*** that computation

# 2 Kinds of Conditional Jumps

- Most **arithmetic instructions** set ***condition code*** bits (part of the processor state) to record information about the result of the arithmetic op (e.g. zero, non-zero, >0)
  - True of `addq, subq, andq, orq`;
    but not `imulq, idivq, leaq`
- There are two other **test & compare instructions** that set the ***condition codes***
  - `cmpq  src,dst`  # compare `dst` to `src` (i.e. `dst-src`)
  - `testq src,dst`  # calculate `dst` & `src` (logical and)
  - Crucially! these instructions do not alter `src` or `dst`

# Conditional Jumps After Arith. Ops

```
jz    label      # jump if result == 0
jnz   label      # jump if result != 0
jg    label      # jump if result > 0
jng   label      # jump if result <= 0
jge   label      # jump if result >= 0
jnge  label      # jump if result < 0
jl    label      # jump if result < 0
jnl   label      # jump if result >= 0
jle   label      # jump if result <= 0
jnle  label      # jump if result > 0
```

- Obviously, the assembler is mapping multiple opcode mnemonics to some of the same actual instructions

# `cmp` and then jump

- Alternately, we can use a comparison to set the condition codes before executing a jump

  `cmpq op1,op2` # set cond. codes as if `op2 - op1`

  `jcc label`

- where jcc is a conditional jump instruction that is taken if the result of the comparison matches the condition cc
- **Warning!** Because the dst is on the right in GCC/AT&T syntax, we write `op1,op2` but expect `op2 - op1`
  - **This is very easy to get wrong!**

# Conditional Jumps After `cmp` or `tst`

```
je     label     # jump if op2 == op1
jne    label     # jump if op2 != op1
jg     label     # jump if op2 > op1
jng    label     # jump if op2 <= op1
jge    label     # jump if op2 >= op1
jnge   label     # jump if op2 < op1
jl     label     # jump if op2 < op1
jnl    label     # jump if op2 >= op1
jle    label     # jump if op2 <= op1
jnle   label     # jump if op2 > op1
```

- Assembler is mapping multiple mnemonics to the same instruction here as well
- **Note!** what order are op2 and op1 in above?

# Outline

x86

Assembly Language

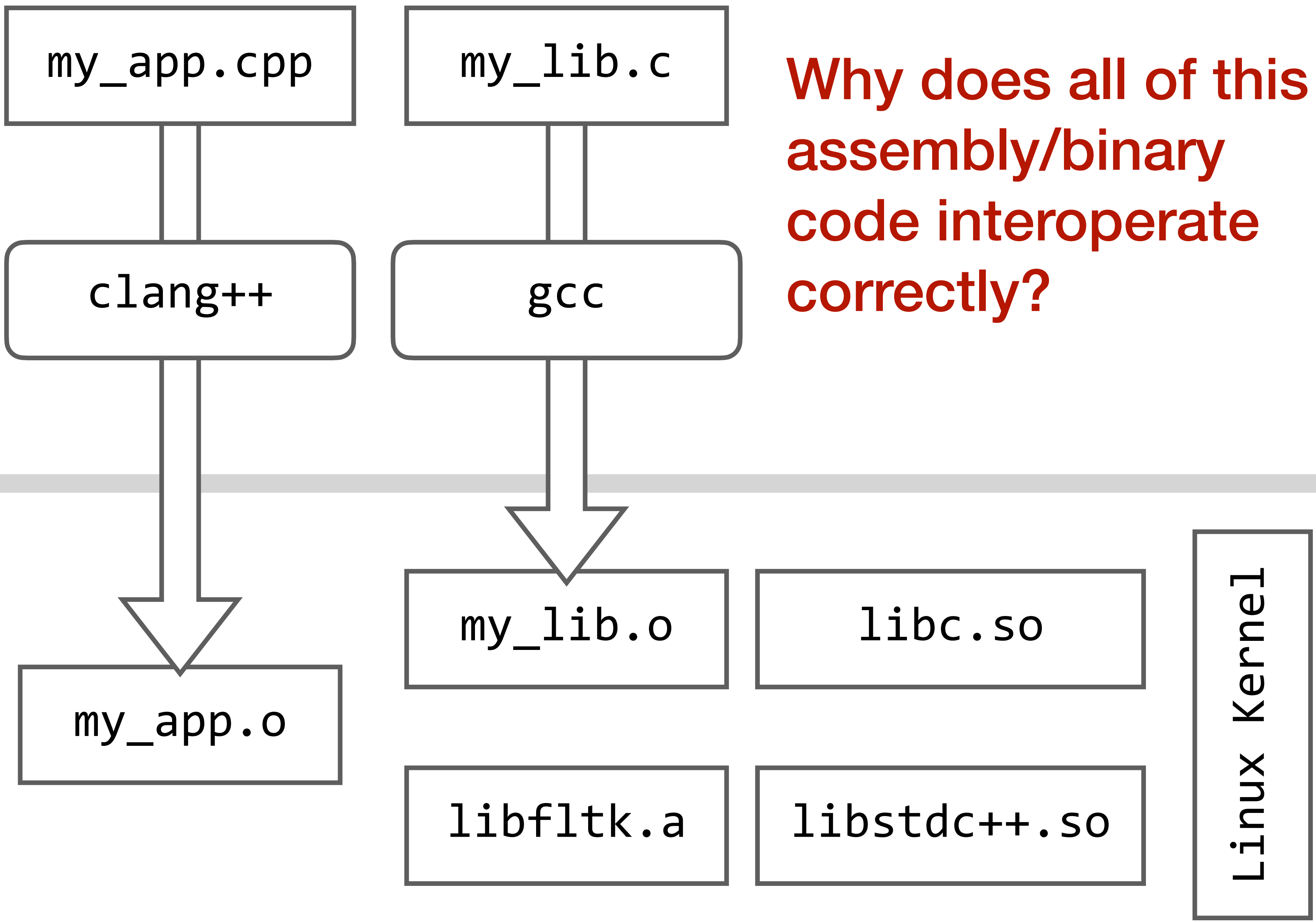
Memory Model

Arithmetic & Data Movement

Control Flow, Jumps & Branches

**Function Calls**

# Binary Compatibility




**Why does all of this assembly/binary code interoperate correctly?**

**Compiled Executables (OS, apps, & libraries)**

# Application Binary Interface

- The **application binary interface (ABI)** defines ***standard conventions*** for interoperation of different pieces of binary code
  - ✦ and the nice thing about standards is…
- Who & what needs to agree on the ABI standard?
  - ✦ (*an incomplete list*) compilers, languages, operating systems, programmers directly writing assembly
  - ✦ Some other strange possibilities — device manufacturers (drivers), developers of software engineering tools (e.g. debuggers and profilers)
- The following slides describe the **System V/AMD64 ABI** used by Linux and Mac

# ABI — Calling Convention

- The ABI covers many issues
  - Here, we will focus on the most common issue — function calls
- **Calling Convention** — the ABI needs to specify how the method performing the call (caller) and the method being called (callee) correctly orchestrate a function call.
- **Stack discipline** — the ABI needs to specify how the stack is supposed to be managed
- If your code doesn't interact with other code, ***do you need*** to obey these invariants? ***Should you***? Why? Why not?

# `call` & `ret` Instructions

```
call label
```

```
     %rsp ← %rsp - 8
mem[%rsp] ← %rip
     %rip ← addr of label
```

*Note: label can be replaced by the* `*reg` *and* `*(reg)` *forms, which work the same as for an unconditional jump. Will be very useful for dynamic dispatch of method calls*

```
ret
```

```
%rip ← mem[%rsp]
%rsp ← %rsp + 8
```

**Warning!** *The quad-word on the top of the stack needs to be the address we want to jump to; and not garbage data!*

# `enter` & `leave` Instructions

```
enter …
```

(for us) *deprecated*

slow on current processors — best to avoid; don't use in your project!

```
leave
```

equivalent to…

```
movq %rbp,%rsp
popq %rbp
```

`leave` is generated by many compilers. It fits in 1 byte, and saves space. Not clear if it's any faster.
You can take it or *leave* it.

# x86-64-Register Convention

- *note* — the first call argument `%rdi` will always hold the `this` pointer in MiniJava
- **Callee Save** — Does the function jumped to have to keep the register the same when returning?
- **Debugging Tip** — write code to mess up every ***unsaved*** register on return

| Callee Save? | Reg | Role / Usage |
|---|---|---|
| | `%rax` | function result |
| *save* | `%rbx` | |
| | `%rcx` | **4th** call argument |
| | `%rdx` | **3rd** call argument |
| *special* | `%rsp` | stack pointer |
| *save* | `%rbp` | base (aka. frame) pointer |
| | `%rsi` | **2nd** call argument |
| | `%rdi` | **1st** call argument |
| | `%r8` | **5th** call argument |
| | `%r9` | **6th** call argument |
| | `%r10` | |
| *%r12 to %r15 save* | | ... |
| | `%r15` | |

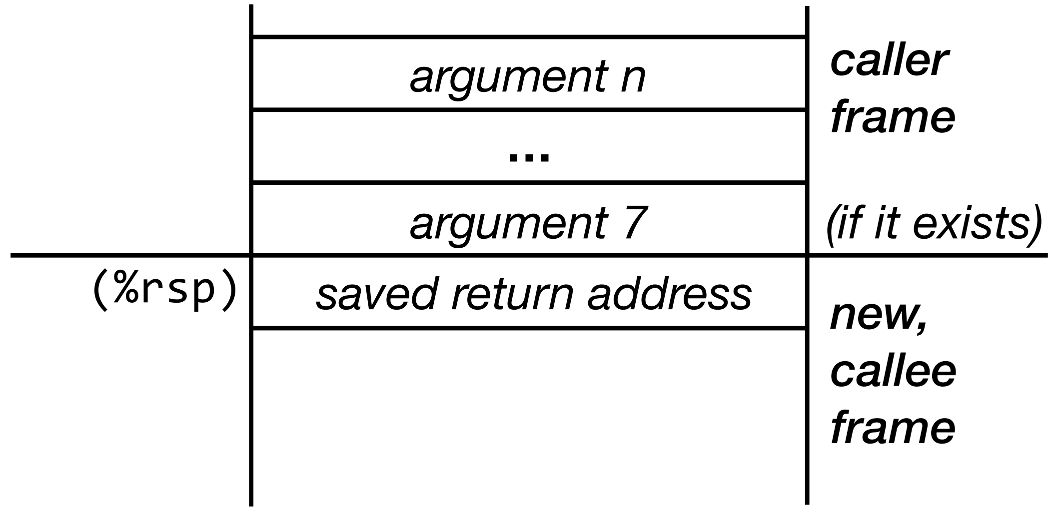
# Call-Site (Caller) Code

- Caller places up to 6 arguments into the designated registers, (more arguments go on stack) then executes a call instruction (pushes 8-byte return address onto stack)
- On return, the result is in register `%rax`
- e.g. assembly for `n = sumOf(17, 42)`

```
movq  $42,%rsi      # load arguments in
movq  $17,%rdi      #   either order, but use
                    #   correct registers
call  sumOf         # jump & push ret addr
movq  %rax,offsetn(%rbp) # store result
```

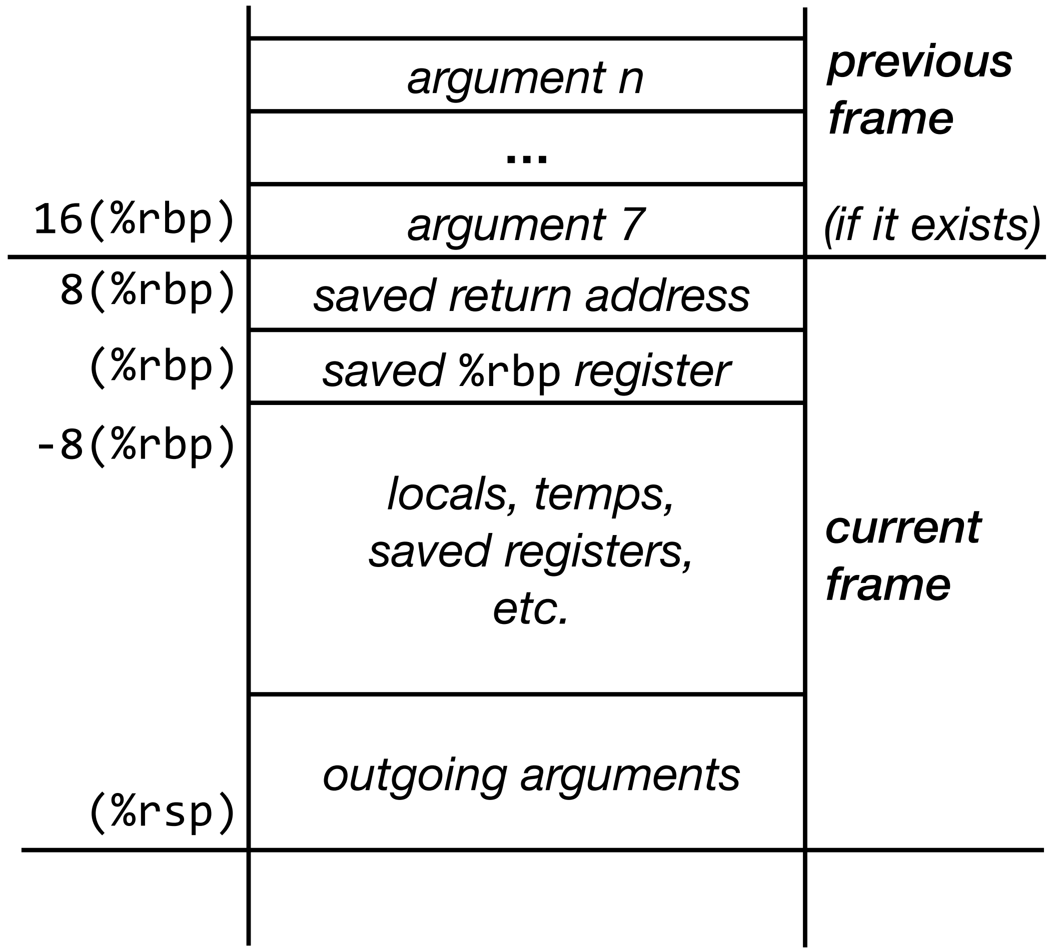
# Function Prologue (Enter)

- On entry, arguments in registers or on stack; return address on the top of the stack
- Prologue must set up the stack frame correctly



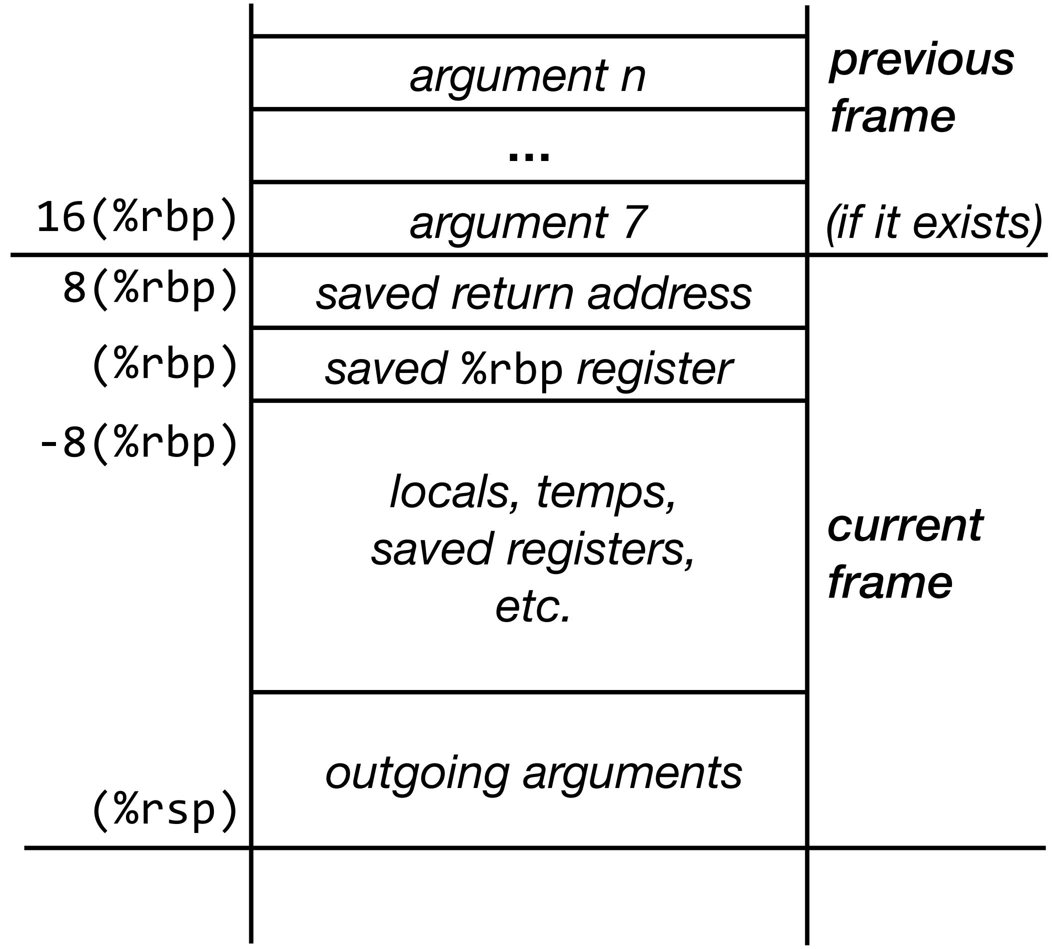

```
pushq %rbp                # save old frame ptr
movq  %rsp,%rbp           # new frame ptr is top of
                          #   stack after ret addr and
                          #   old %rbp pushed
subq  $framesize,%rsp     # allocate stack frame
                          #   (size should be multiple
                          #    of 16 normally)
```

# Stack Frame Layout



# Function Epilogue (Return)

- At the end of the function, the stack frame is in normal order; the return value must be in %rax, and callee-save registers must be restored
- Epilogue must now unwind the stack frame correctly, and then return




```
movq %rbp,%rsp   # or just use
popq %rbp        #   leave
ret
```

# Example C Function

```c
int sumOf(int x, int y) {
    int a; int b;
    a = x;
    b = a + y;
    return b;
}
```

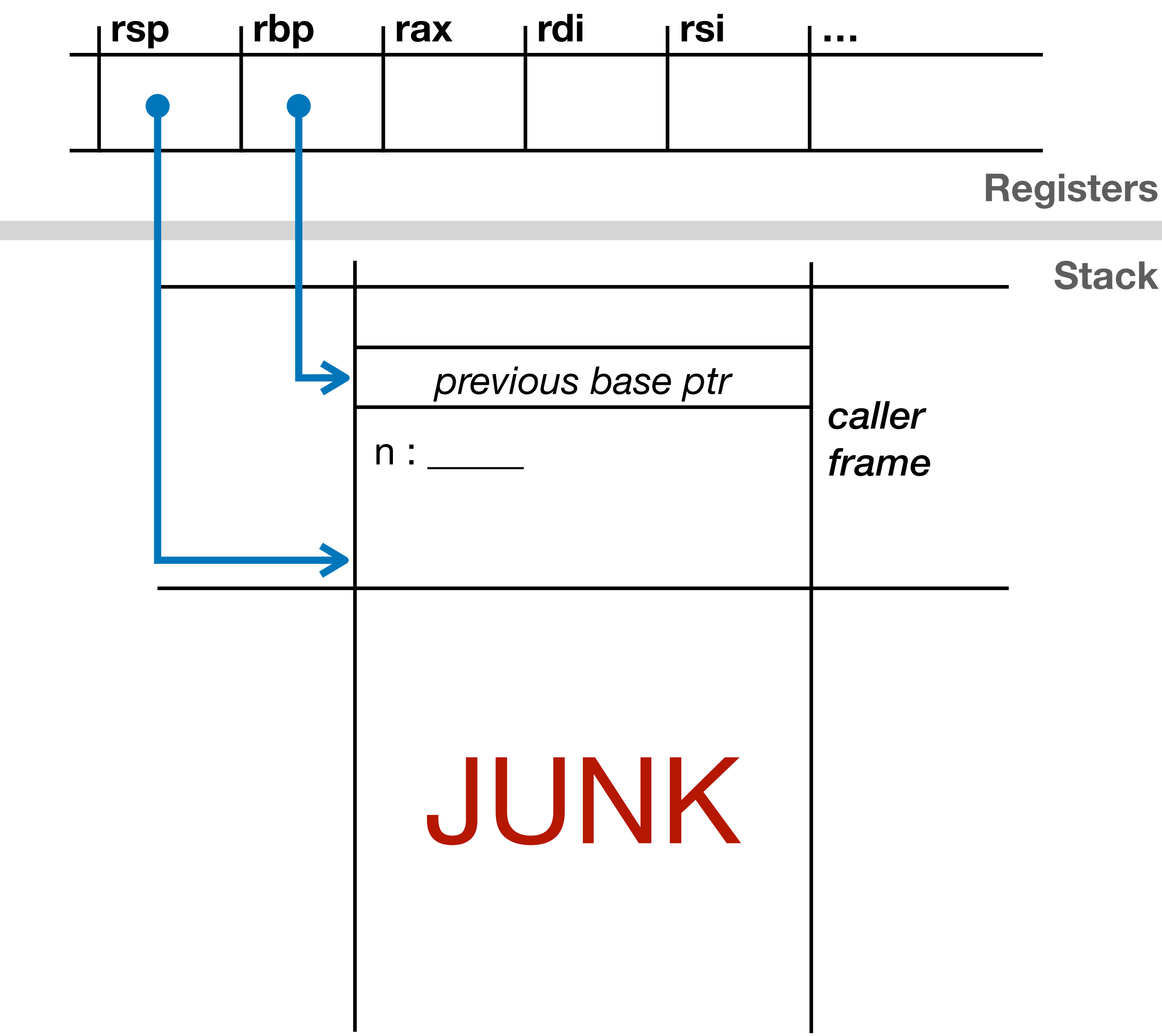
```
# int sumOf(int x, int y) {
#   int a; int b;
sumOf:
  pushq %rbp
  movq  %rsp,%rbp
  subq  $16,%rsp

# a = x;
  movq  %rdi,-8(%rbp)

# b = a + y;
  movq  -8(%rbp),%rax
  addq  %rsi,%rax
  movq  %rax,-16(%rbp)

# return b;
  movq  -16(%rbp),%rax
  movq  %rbp,%rsp
  popq  %rbp
  ret
# }
```

# Stack Frame for `sumOf(...)`



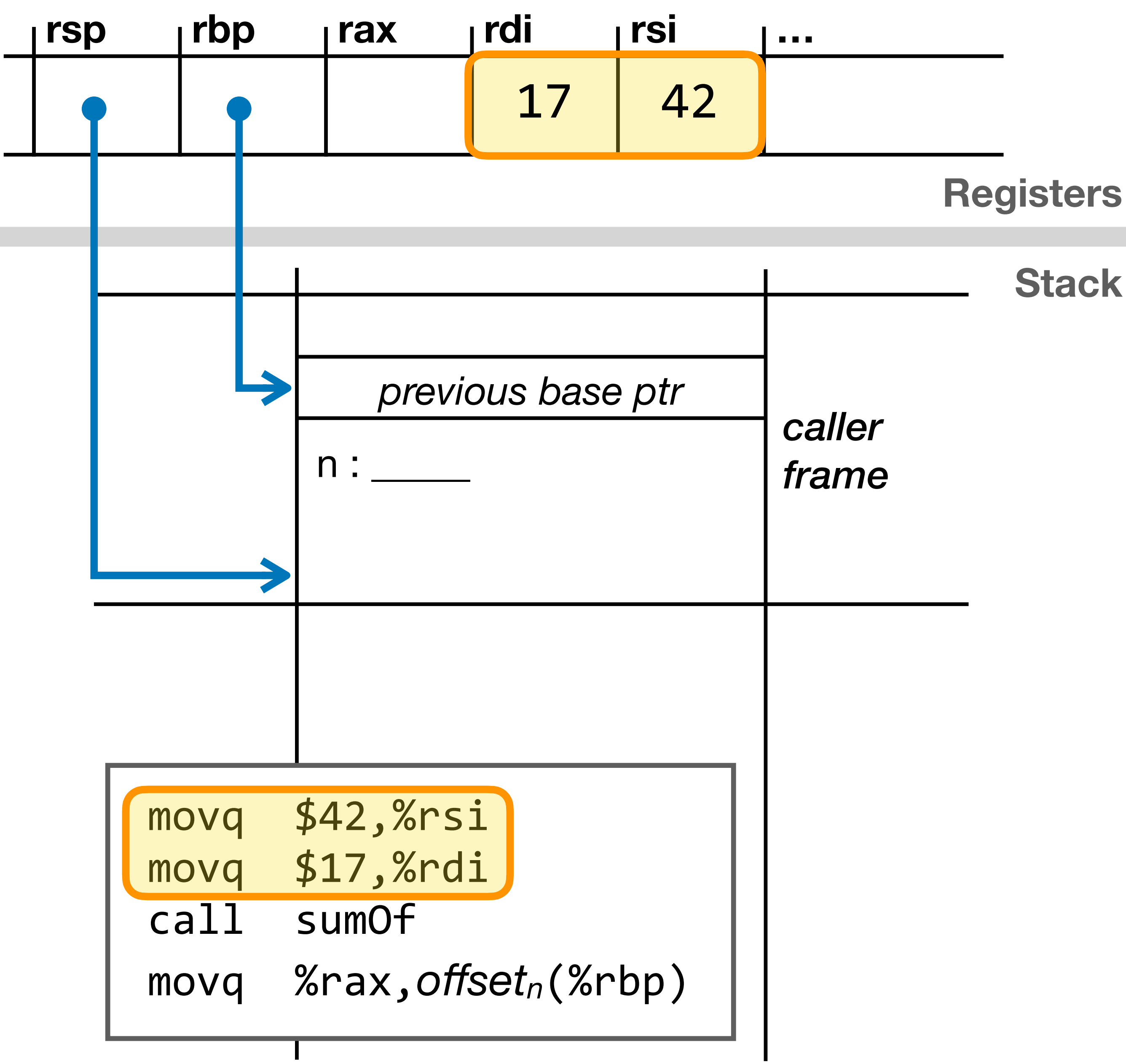

```
# int sumOf(int x,
#               int y) {
#   int a; int b;
sumOf:
  pushq %rbp
  movq  %rsp,%rbp
  subq  $16,%rsp

# a = x;
  movq  %rdi,-8(%rbp)

# b = a + y;
  movq  -8(%rbp),%rax
  addq  %rsi,%rax
  movq  %rax,-16(%rbp)

# return b;
  movq  -16(%rbp),%rax
  movq  %rbp,%rsp
  popq  %rbp
  ret
# }
```

# Stack Frame for `sumOf(…)`



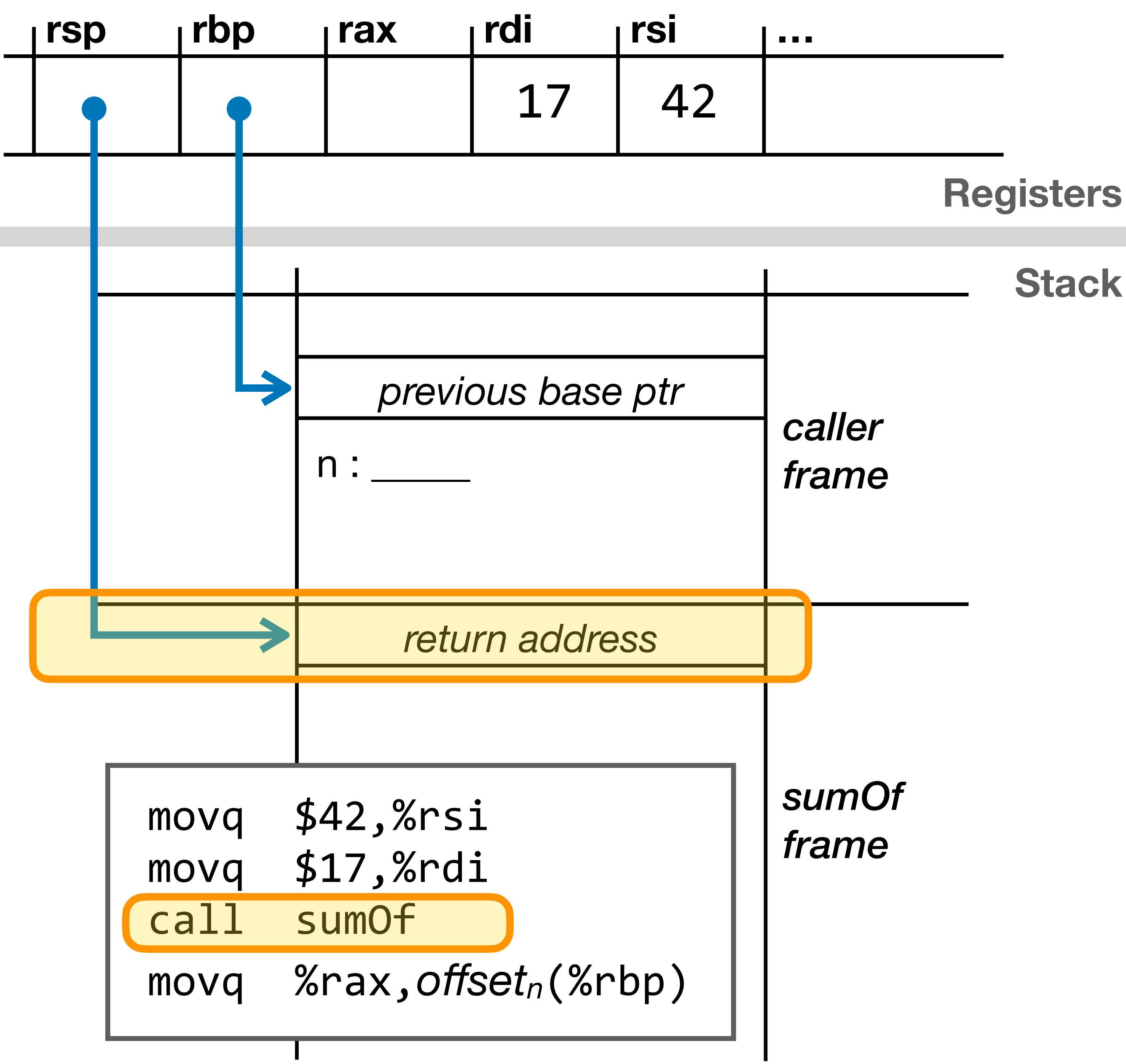

```
# int sumOf(int x,
#           int y) {
#   int a; int b;
sumOf:
  pushq %rbp
  movq  %rsp,%rbp
  subq  $16,%rsp

# a = x;
  movq  %rdi,-8(%rbp)

# b = a + y;
  movq  -8(%rbp),%rax
  addq  %rsi,%rax
  movq  %rax,-16(%rbp)

# return b;
  movq  -16(%rbp),%rax
  movq  %rbp,%rsp
  popq  %rbp
  ret
# }
```

# Stack Frame for `sumOf(…)`



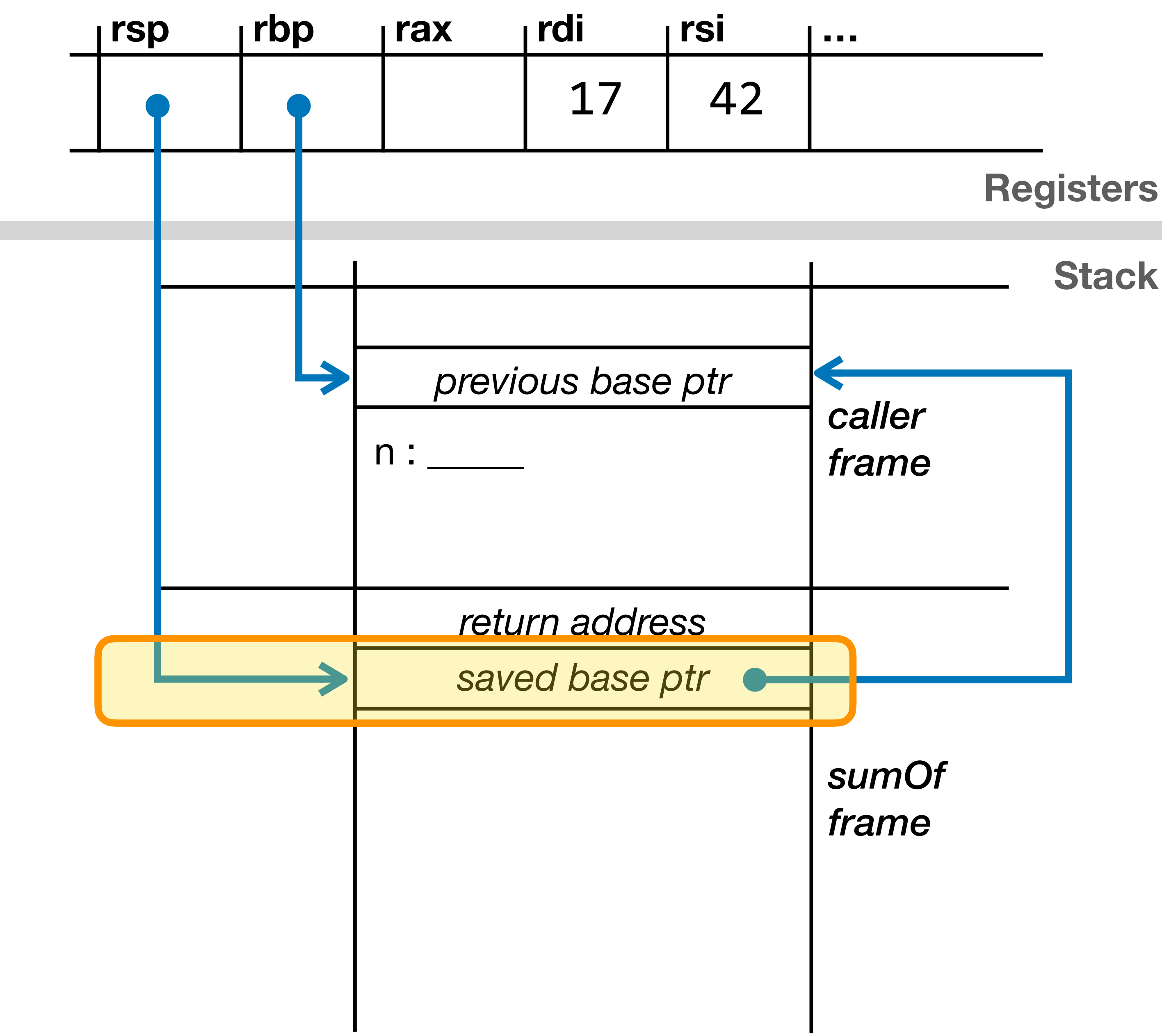

```
# int sumOf(int x,
#           int y) {
#   int a; int b;
sumOf:
  pushq %rbp
  movq  %rsp,%rbp
  subq  $16,%rsp

# a = x;
  movq  %rdi,-8(%rbp)

# b = a + y;
  movq  -8(%rbp),%rax
  addq  %rsi,%rax
  movq  %rax,-16(%rbp)

# return b;
  movq  -16(%rbp),%rax
  movq  %rbp,%rsp
  popq  %rbp
  ret
# }
```

# Stack Frame for `sumOf(…)`



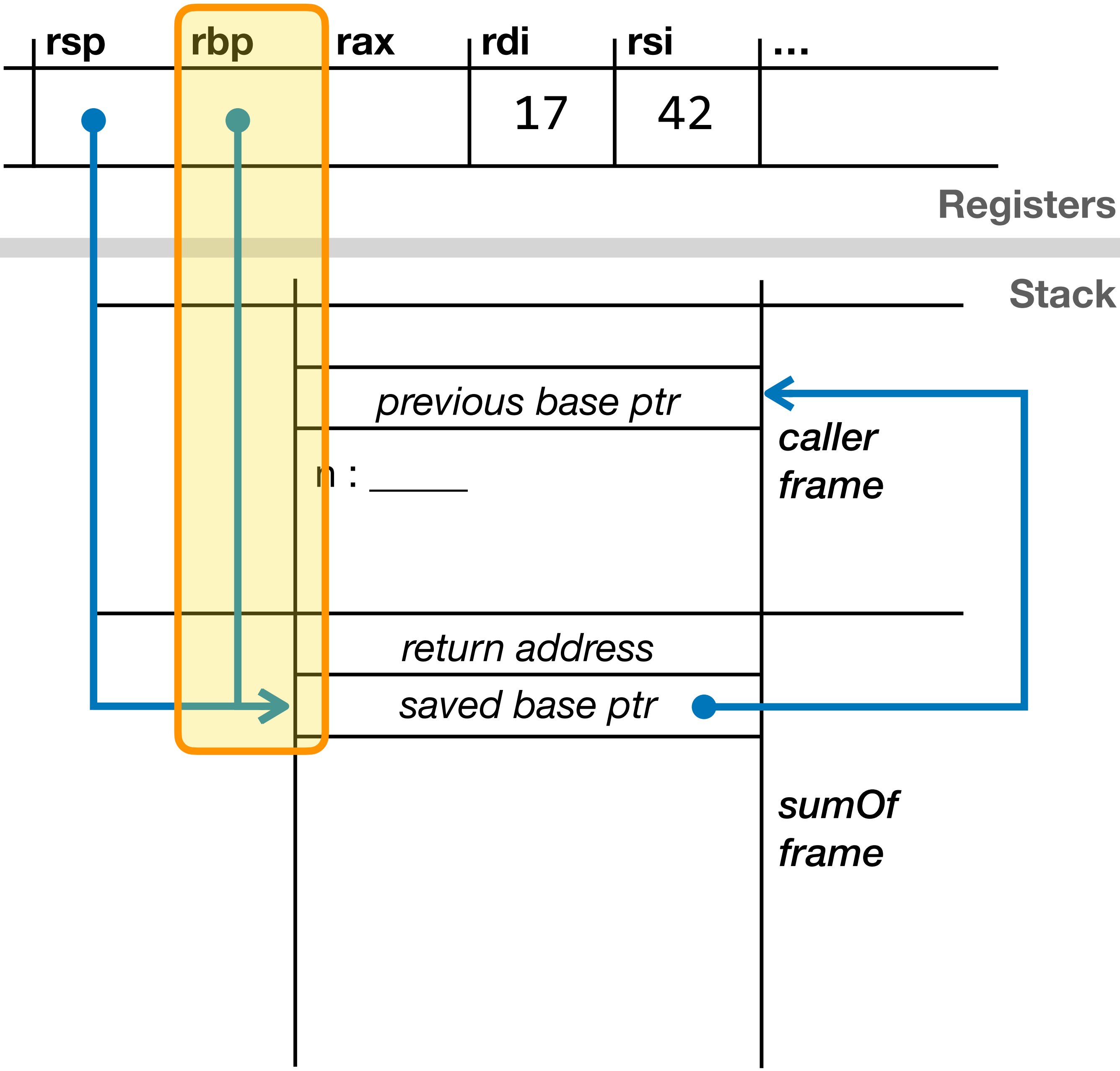

```
# int sumOf(int x,
#             int y) {
#   int a; int b;
sumOf:
  pushq %rbp
  movq  %rsp,%rbp
  subq  $16,%rsp

# a = x;
  movq  %rdi,-8(%rbp)

# b = a + y;
  movq  -8(%rbp),%rax
  addq  %rsi,%rax
  movq  %rax,-16(%rbp)

# return b;
  movq  -16(%rbp),%rax
  movq  %rbp,%rsp
  popq  %rbp
  ret
# }
```

# Stack Frame for `sumOf(…)`



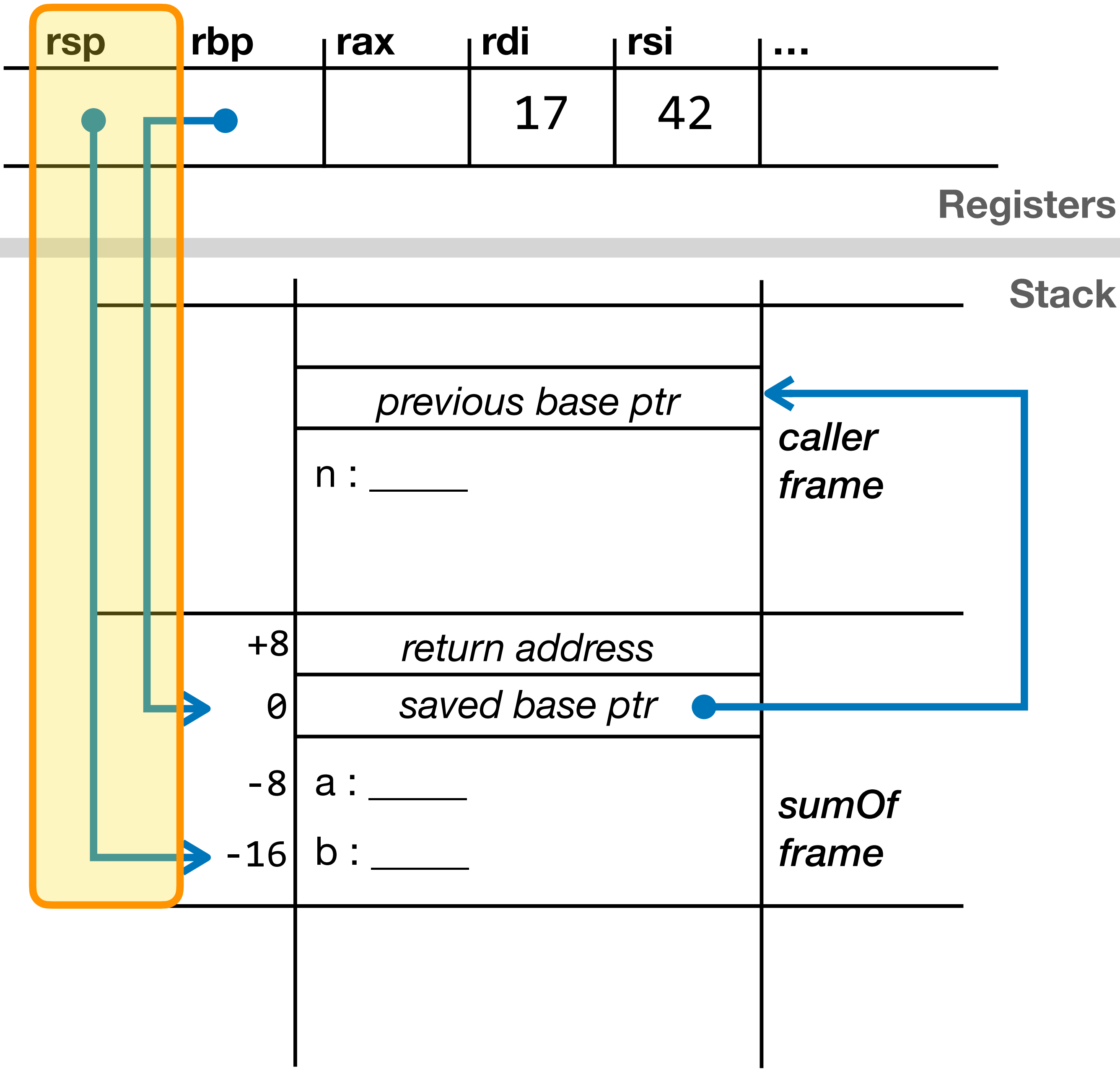

```
# int sumOf(int x,
#           int y) {
#   int a; int b;
sumOf:
  pushq %rbp
  movq  %rsp,%rbp
  subq  $16,%rsp

# a = x;
  movq  %rdi,-8(%rbp)

# b = a + y;
  movq  -8(%rbp),%rax
  addq  %rsi,%rax
  movq  %rax,-16(%rbp)

# return b;
  movq  -16(%rbp),%rax
  movq  %rbp,%rsp
  popq  %rbp
  ret
# }
```

# Stack Frame for `sumOf(…)`



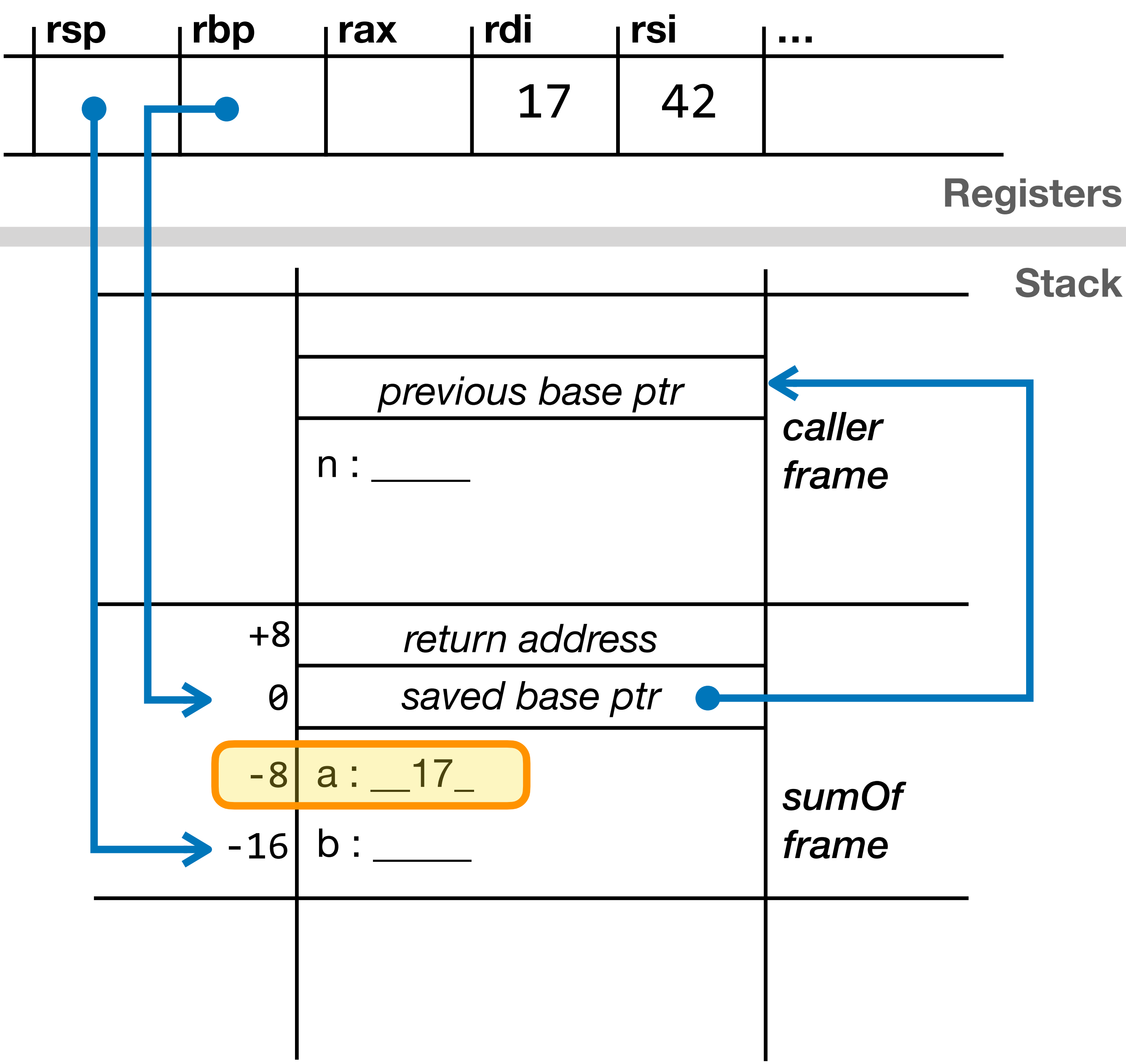

```
# int sumOf(int x,
#           int y) {
#   int a; int b;
sumOf:
  pushq %rbp
  movq  %rsp,%rbp
  subq  $16,%rsp

# a = x;
  movq  %rdi,-8(%rbp)

# b = a + y;
  movq  -8(%rbp),%rax
  addq  %rsi,%rax
  movq  %rax,-16(%rbp)

# return b;
  movq  -16(%rbp),%rax
  movq  %rbp,%rsp
  popq  %rbp
  ret
# }
```

# Stack Frame for `sumOf(…)`



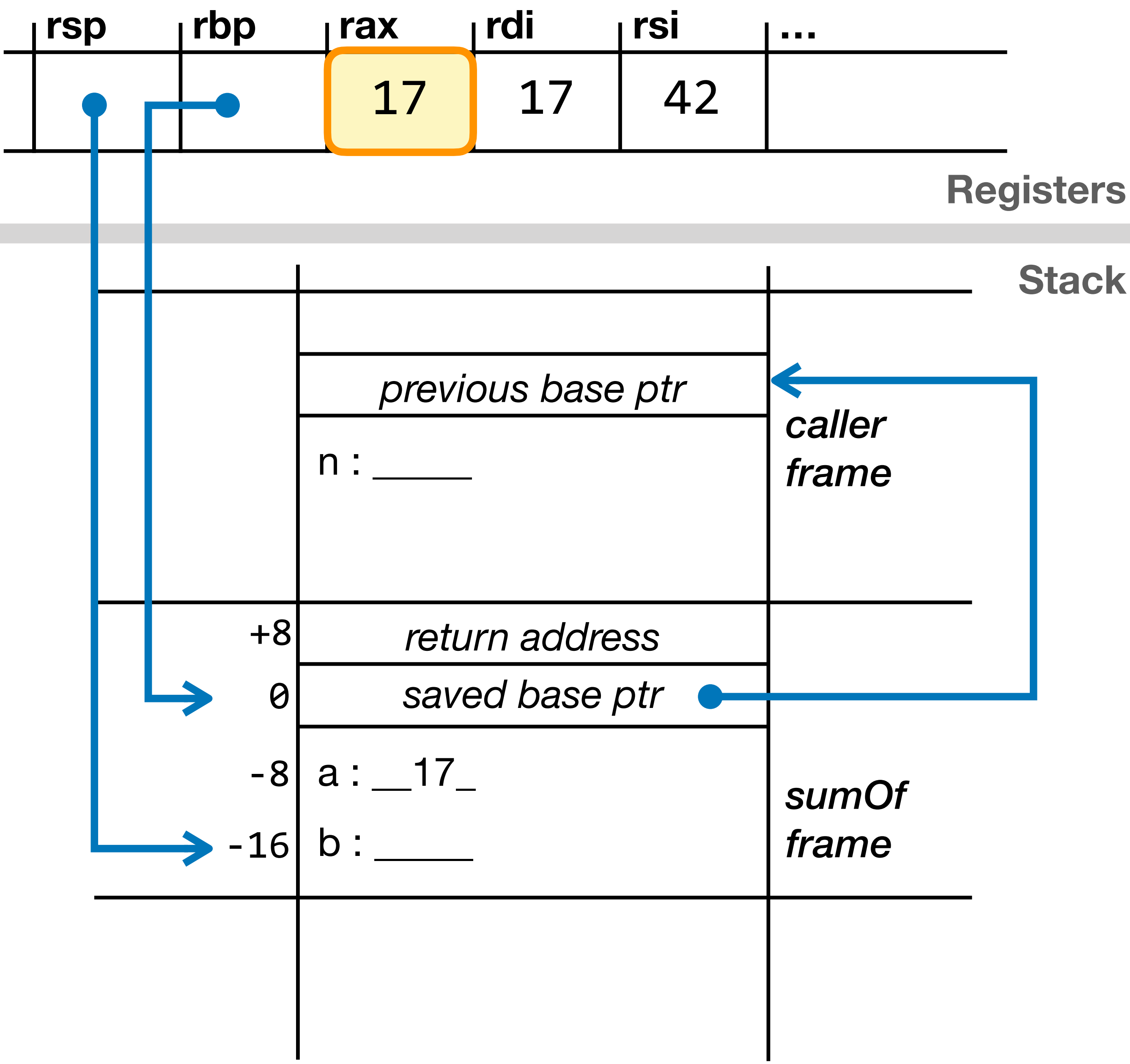

```
# int sumOf(int x,
#           int y) {
#   int a; int b;
sumOf:
  pushq %rbp
  movq  %rsp,%rbp
  subq  $16,%rsp

# a = x;
  movq  %rdi,-8(%rbp)

# b = a + y;
  movq  -8(%rbp),%rax
  addq  %rsi,%rax
  movq  %rax,-16(%rbp)

# return b;
  movq  -16(%rbp),%rax
  movq  %rbp,%rsp
  popq  %rbp
  ret
# }
```

# Stack Frame for `sumOf(…)`



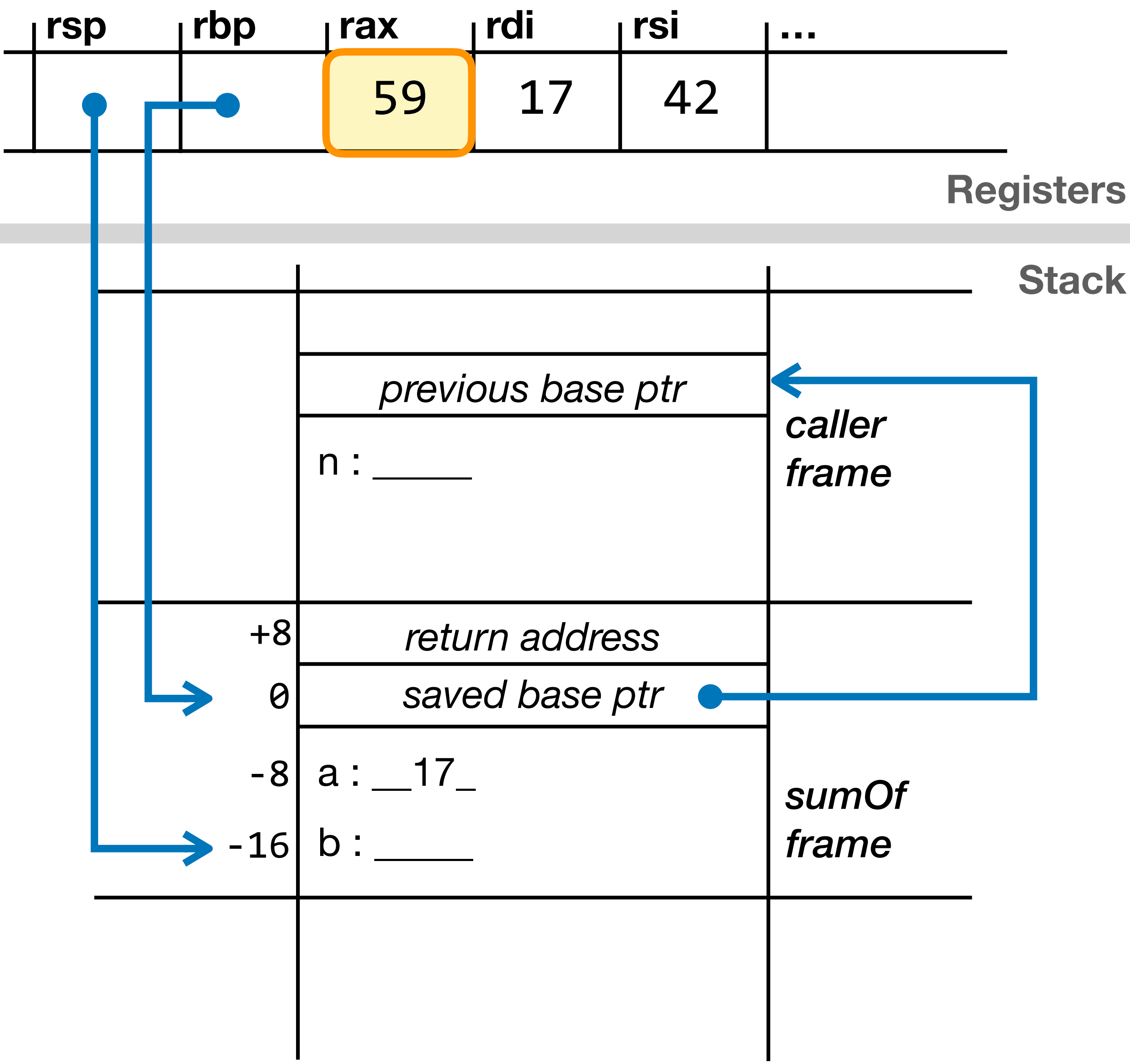

```
# int sumOf(int x,
#           int y) {
#   int a; int b;
sumOf:
  pushq %rbp
  movq  %rsp,%rbp
  subq  $16,%rsp

# a = x;
  movq  %rdi,-8(%rbp)

# b = a + y;
  movq  -8(%rbp),%rax
  addq  %rsi,%rax
  movq  %rax,-16(%rbp)

# return b;
  movq  -16(%rbp),%rax
  movq  %rbp,%rsp
  popq  %rbp
  ret
# }
```

# Stack Frame for `sumOf(...)`



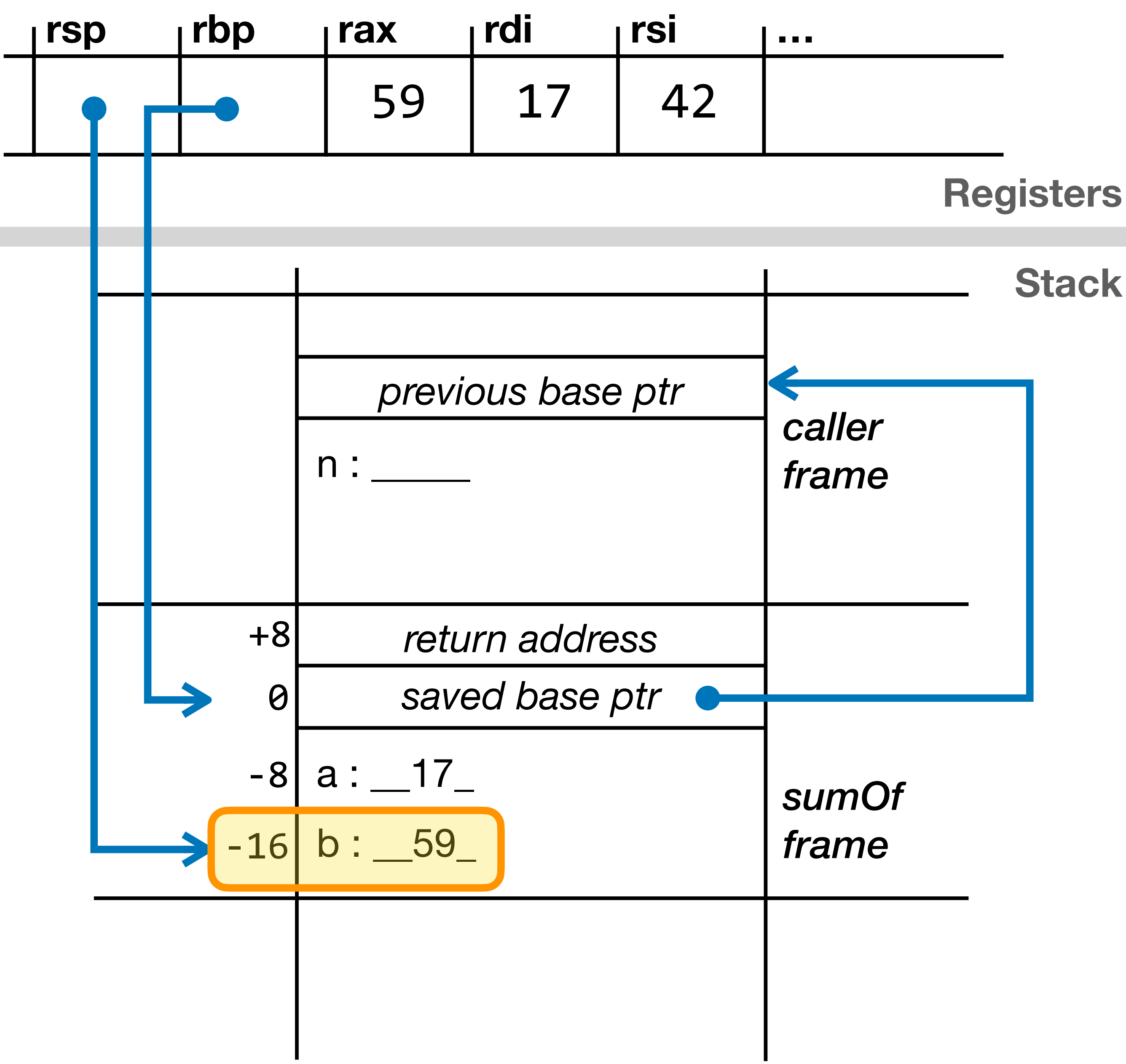

```
# int sumOf(int x,
#           int y) {
#   int a; int b;
sumOf:
  pushq %rbp
  movq  %rsp,%rbp
  subq  $16,%rsp

# a = x;
  movq  %rdi,-8(%rbp)

# b = a + y;
  movq  -8(%rbp),%rax
  addq  %rsi,%rax
  movq  %rax,-16(%rbp)

# return b;
  movq  -16(%rbp),%rax
  movq  %rbp,%rsp
  popq  %rbp
  ret
# }
```

# Stack Frame for `sumOf(…)`



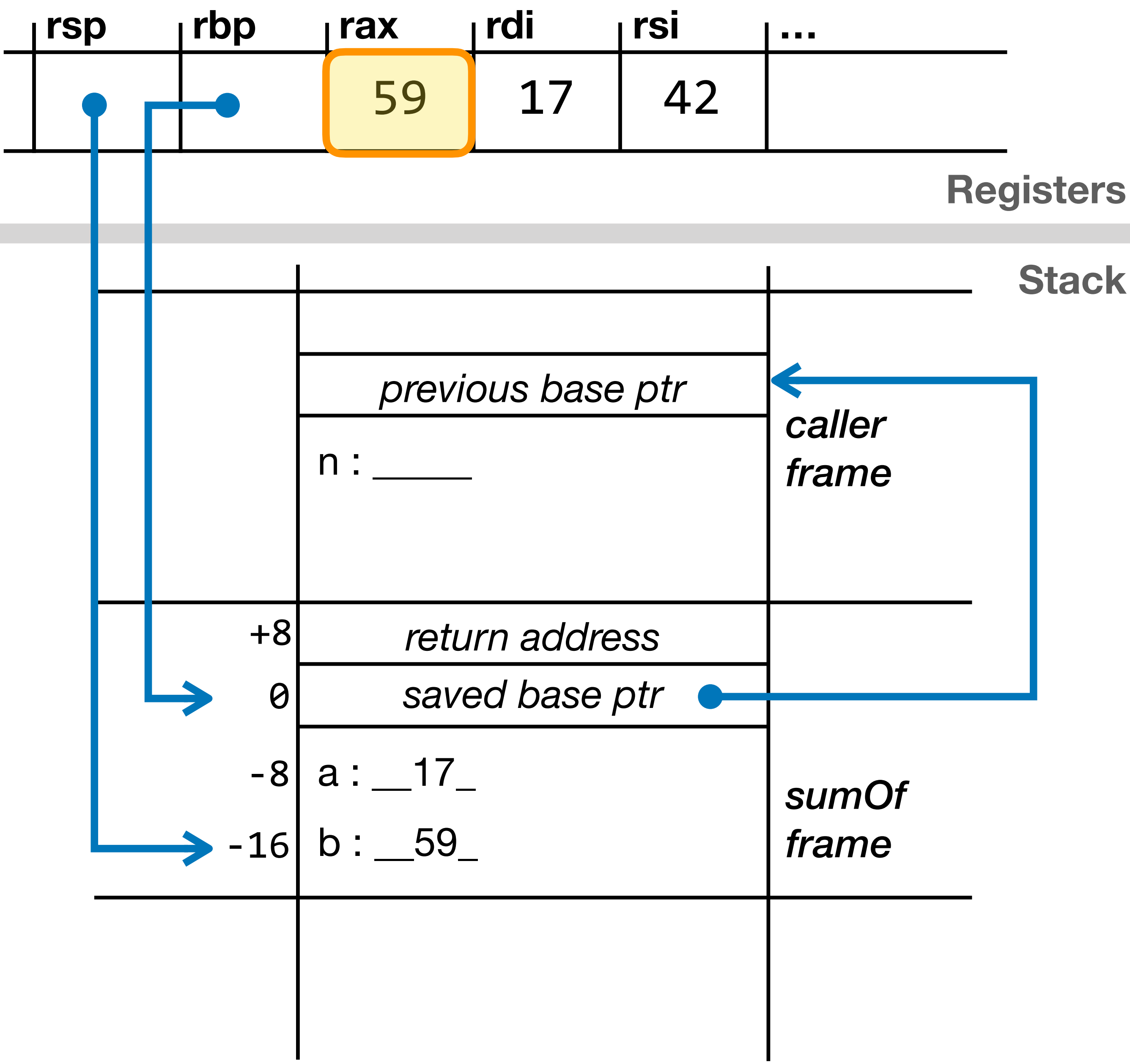

```
# int sumOf(int x,
#           int y) {
#   int a; int b;
sumOf:
  pushq %rbp
  movq  %rsp,%rbp
  subq  $16,%rsp

# a = x;
  movq  %rdi,-8(%rbp)

# b = a + y;
  movq  -8(%rbp),%rax
  addq  %rsi,%rax
  movq  %rax,-16(%rbp)

# return b;
  movq  -16(%rbp),%rax
  movq  %rbp,%rsp
  popq  %rbp
  ret
# }
```

# Stack Frame for sumOf(…)



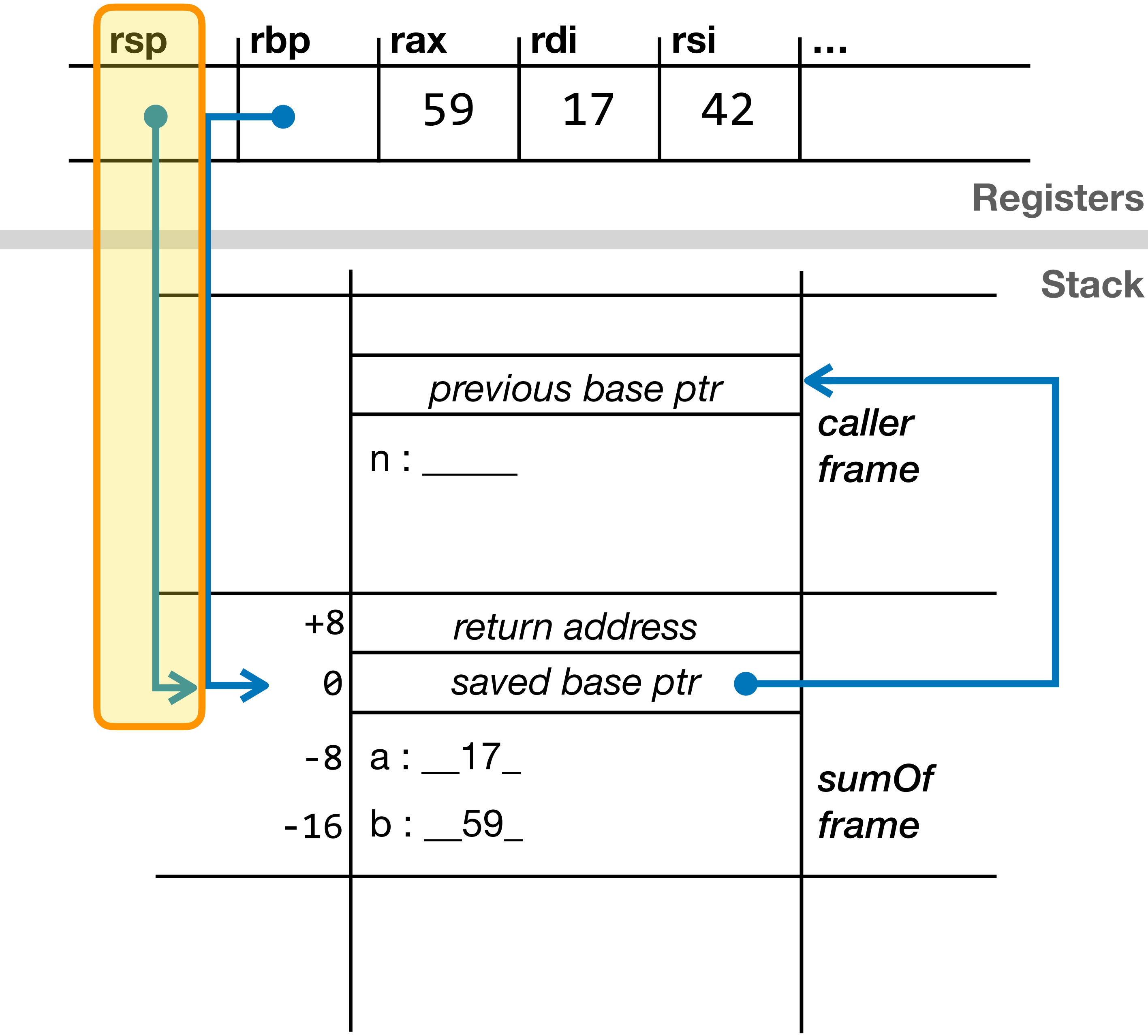

```
# int sumOf(int x,
#           int y) {
#   int a; int b;
sumOf:
  pushq %rbp
  movq  %rsp,%rbp
  subq  $16,%rsp

# a = x;
  movq  %rdi,-8(%rbp)

# b = a + y;
  movq  -8(%rbp),%rax
  addq  %rsi,%rax
  movq  %rax,-16(%rbp)

# return b;
  movq  -16(%rbp),%rax
  movq  %rbp,%rsp
  popq  %rbp
  ret
# }
```

# Stack Frame for `sumOf(…)`



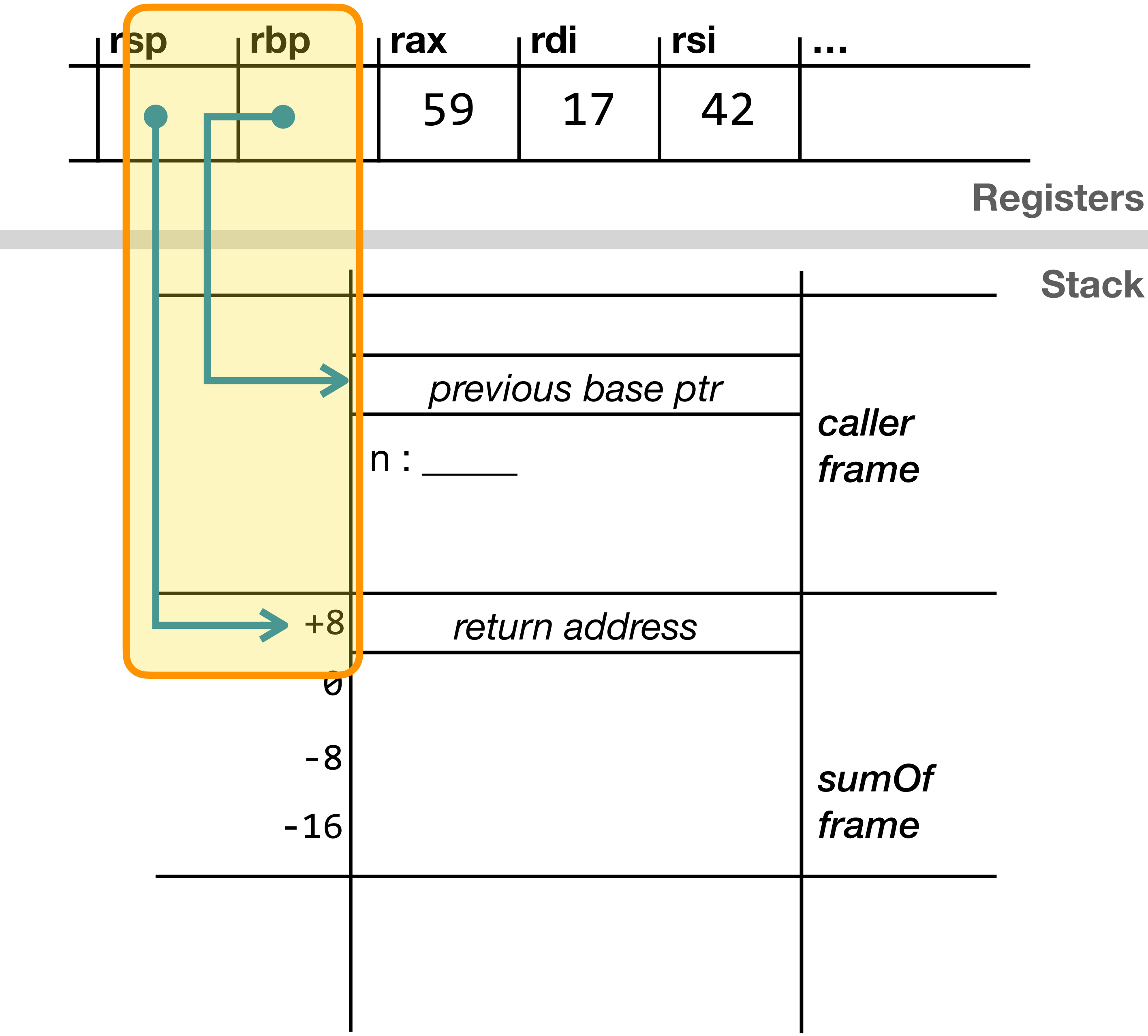

```
# int sumOf(int x,
#           int y) {
#   int a; int b;
sumOf:
  pushq %rbp
  movq  %rsp,%rbp
  subq  $16,%rsp

# a = x;
  movq  %rdi,-8(%rbp)

# b = a + y;
  movq  -8(%rbp),%rax
  addq  %rsi,%rax
  movq  %rax,-16(%rbp)

# return b;
  movq  -16(%rbp),%rax
  movq  %rbp,%rsp
  popq  %rbp
  ret
# }
```

# Stack Frame for `sumOf(…)`



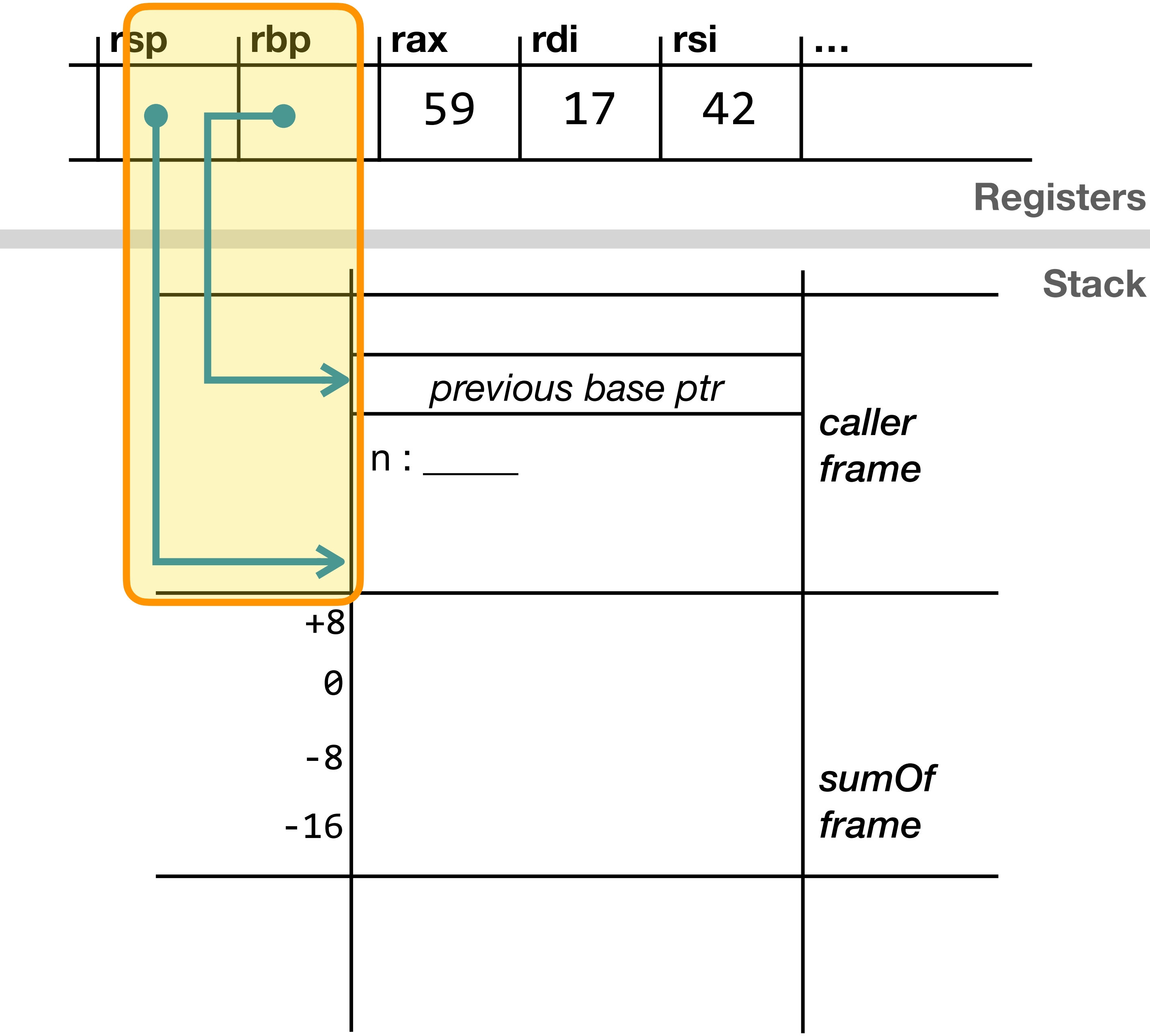

```
# int sumOf(int x,
#           int y) {
#   int a; int b;
sumOf:
  pushq %rbp
  movq  %rsp,%rbp
  subq  $16,%rsp

# a = x;
  movq  %rdi,-8(%rbp)

# b = a + y;
  movq  -8(%rbp),%rax
  addq  %rsi,%rax
  movq  %rax,-16(%rbp)

# return b;
  movq  -16(%rbp),%rax
  movq  %rbp,%rsp
  popq  %rbp
  ret
# }
```

# Stack Frame for `sumOf(…)`



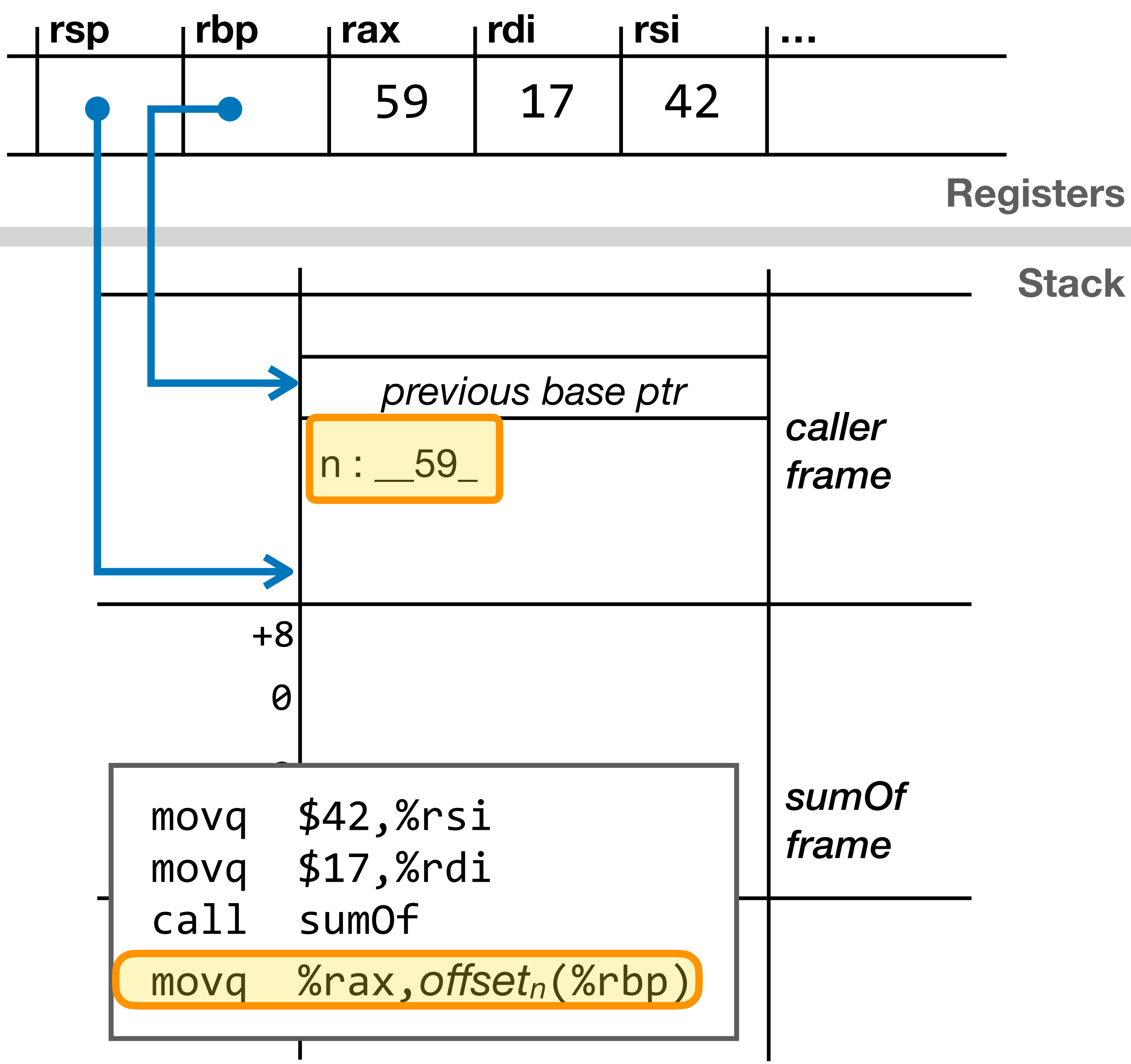

```
# int sumOf(int x,
#           int y) {
#   int a; int b;
sumOf:
  pushq %rbp
  movq  %rsp,%rbp
  subq  $16,%rsp

# a = x;
  movq  %rdi,-8(%rbp)

# b = a + y;
  movq  -8(%rbp),%rax
  addq  %rsi,%rax
  movq  %rax,-16(%rbp)

# return b;
  movq  -16(%rbp),%rax
  movq  %rbp,%rsp
  popq  %rbp
  ret
# }
```

# Stack Frame for `sumOf(…)`




```
# int sumOf(int x,
#           int y) {
#   int a; int b;
sumOf:
  pushq %rbp
  movq  %rsp,%rbp
  subq  $16,%rsp

# a = x;
  movq  %rdi,-8(%rbp)

# b = a + y;
  movq  -8(%rbp),%rax
  addq  %rsi,%rax
  movq  %rax,-16(%rbp)

# return b;
  movq  -16(%rbp),%rax
  movq  %rbp,%rsp
  popq  %rbp
  ret
# }
```

# The Nice Thing About Standards…

- Again, the preceding discussion explains the System V/AMD64 ABI convention (used by Linux, MacOS X)
- Microsoft's x64 calling conventions are different 😩
  - First four arguments go `%rcx, %rdx, %r8, %r9` and then go to the stack after that
  - Called function stack frame must include empty space for the called function to save values passed as arguments, if it so desires
- Compilers can generate many, many more calling conventions — can be used internally, or externally within certain language ecosystems…

# Some LLVM Calling Conventions

`ccc` — the C calling convention (matches target)

`fastcc` — optimize to make calls as fast as possible

`coldcc` — optimize for less work by caller

`ghccc` — special calling convention for Haskell language

`cc 11` — special calling convention for Erlang language

`tailcc` — calling convention for tail call optimizations

`swiftcc` — special calling convention for Swift language

...



*https://llvm.org/docs/LangRef.html#calling-conventions*

# Next Week…

- Now that we're done reviewing our target language…
- How do we transform from ASTs down to an assembly-level linear IR (i.e. x86)
  - ✦ We'll call this **Code Shape**
  - ✦ Next 3 lectures (MWF next week)
- Finally, we'll have a short half-lecture on how to wire it all together and bootstrap our programs into running x86
- (then on to the backend)