Assembly 2: Calling convention

Calling convention

A **calling convention** governs how functions on a particular architecture and operating system interact. This includes rules about includes how function arguments are placed, where return values go, what registers functions may use, how they may allocate local variables, and so forth. Calling conventions ensure that functions compiled by different compilers can interoperate, and they ensure that operating systems can run code from different programming languages and compilers. Some aspects of a calling convention are derived from the instruction set itself, but some are conventional, meaning decided upon by people (for instance, at a convention).

Calling conventions constrain both *callers* and *callees*. A caller is a function that calls another function; a callee is a function that was called. The currently-executing function is a callee, but not a caller.

For concreteness, we learn the x86-64 calling conventions for Linux. These conventions are shared by many OSes, including MacOS (but not Windows), and are officially called the "System V AMD64 ABI."

The official specification: AMD64 ABI

Argument passing and stack frames

One set of calling convention rules governs how function arguments and return values are passed. On x86-64 Linux, the first six function arguments are passed in registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9`, respectively. The seventh and subsequent arguments are passed on the stack, about which more below. The return value is passed in register `%rax`.

The full rules more complex than this. You can read them in the AMD64 ABI, section 3.2.3, but they're quite detailed. Some highlights:

1. A structure argument that fits in a single machine word (64 bits/8 bytes) is passed in a single register.

   Example: `struct small { char a1, a2; }`
2. A structure that fits in two to four machine words (16–32 bytes) is passed in sequential registers, as if it were multiple arguments.

   Example: `struct medium { long a1, a2; }`
3. A structure that's larger than four machine words is always passed on the stack.

   Example: `struct large { long a, b, c, d, e, f, g; }`
4. Floating point arguments are generally passed in special registers, the "SSE registers," that we don't discuss further.
5. If the return value takes more than eight bytes, then the *caller* reserves space for the return value, and passes the *address* of that space as the first argument of the function. The callee will fill in that space when it returns.

Writing small programs to demonstrate these rules is a pleasant exercise; for example:

```
struct small { char a1, a2; };
int f(small s) {
    return s.a1 + 2 * s.a2;
}
```

compiles to:

```
movl %edi, %eax           # copy argument to %eax
movsbl %dil, %edi         # %edi := sign-extension of lowest byte of argument (s.a1)
movsbl %ah, %eax          # %eax := sign-extension of 2nd byte of argument (s.a2)
movsbl %al, %eax
leal (%rdi,%rax,2), %eax  # %eax := %edi + 2 * %eax
ret
```

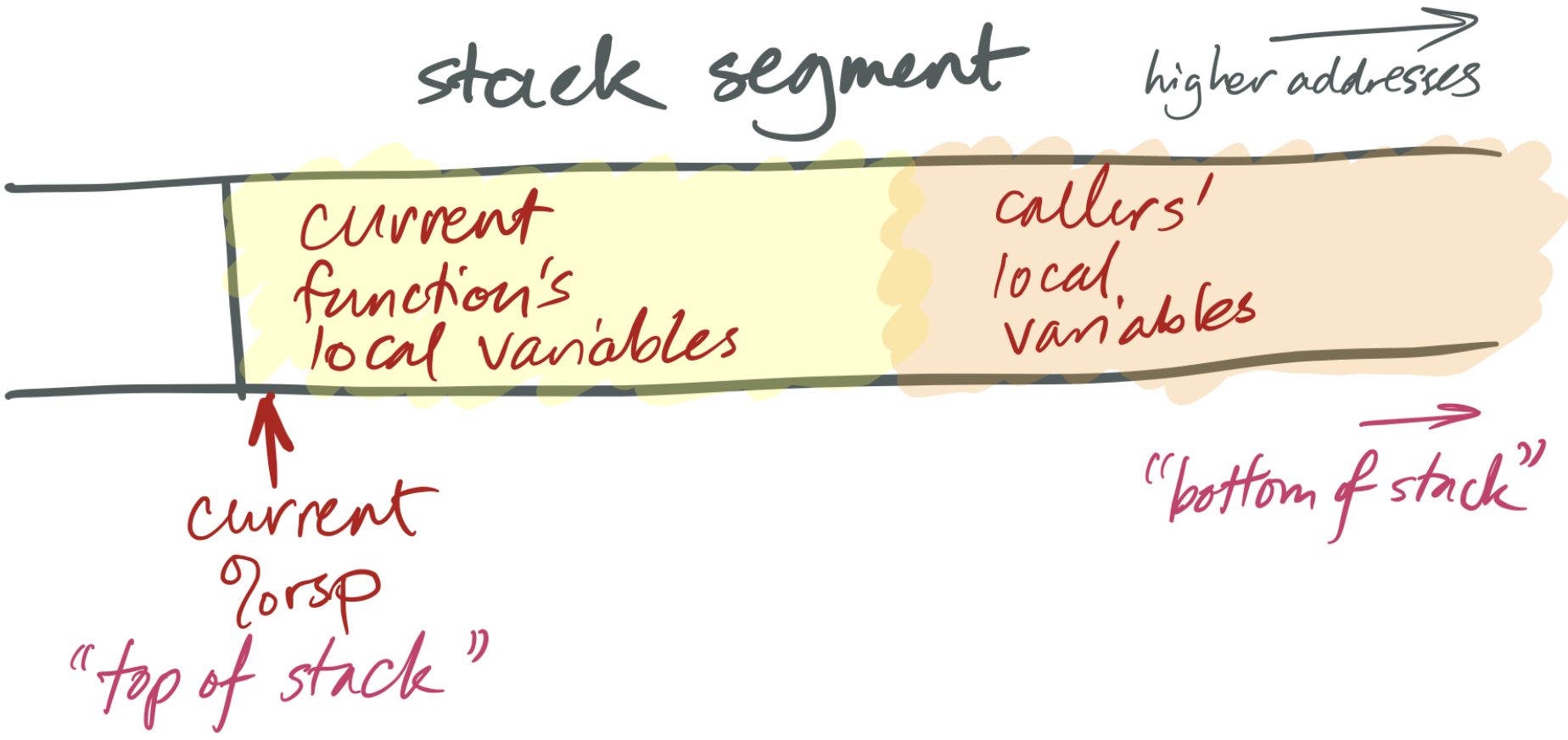
Stack

Recall that the stack is a segment of memory used to store objects with automatic lifetime. Typical stack addresses on x86-64 look like `0x7ffd'9f10'4f58`—that is, close to 2^{47}.

The stack is named after a data structure, which was sort of named after pancakes. Stack data structures support at least three operations: **push** adds a new element to the "top" of the stack; **pop** removes the top element, showing whatever was underneath; and **top** accesses the top element. Note what's missing: the data structure does not allow access to elements other than the top. (Which is sort of how stacks of pancakes work.) This restriction can speed up stack implementations.

Like a stack data structure, the stack memory segment is only accessed from the top. The currently running function accesses *its* local variables; the function's caller, grand-caller, great-grand-caller, and so forth are dormant until the currently running function returns.

x86-64 stacks look like this:



The x86-64 `%rsp` register is a special-purpose register that defines the current "stack pointer." This holds the address of the current top of the stack. On x86-64, as on many architectures, stacks grow *down*: a "push" operation adds space for more automatic-lifetime objects by moving the stack pointer left, to a numerically-smaller address, and a "pop" operation recycles space by moving the stack pointer right, to a numerically-larger address. This means that, considered numerically, the "top" of the stack has a smaller address than the "bottom."

This is built in to the architecture by the operation of instructions like `pushq`, `popq`, `callq`, and `retq`. A `push` instruction pushes a value onto the stack. This both modifies the stack pointer (making it smaller) and modifies the stack segment (by moving data there). For instance, the instruction `pushq X` means:

```
subq $8, %rsp
movq X, (%rsp)
```

And `popq X` undoes the effect of `pushq X`. It means:

```
movq (%rsp), X
addq $8, %rsp
```

`X` can be a register or a memory reference.

The portion of the stack reserved for a function is called that function's **stack frame**. Stack frames are aligned: x86-64 requires that each stack frame be a multiple of 16 bytes, and when a `callq` instruction begins execution, the `%rsp` register must be 16-byte aligned. This means that every function's entry `%rsp` address will be 8 bytes off a multiple of 16.

Return address and entry and exit sequence

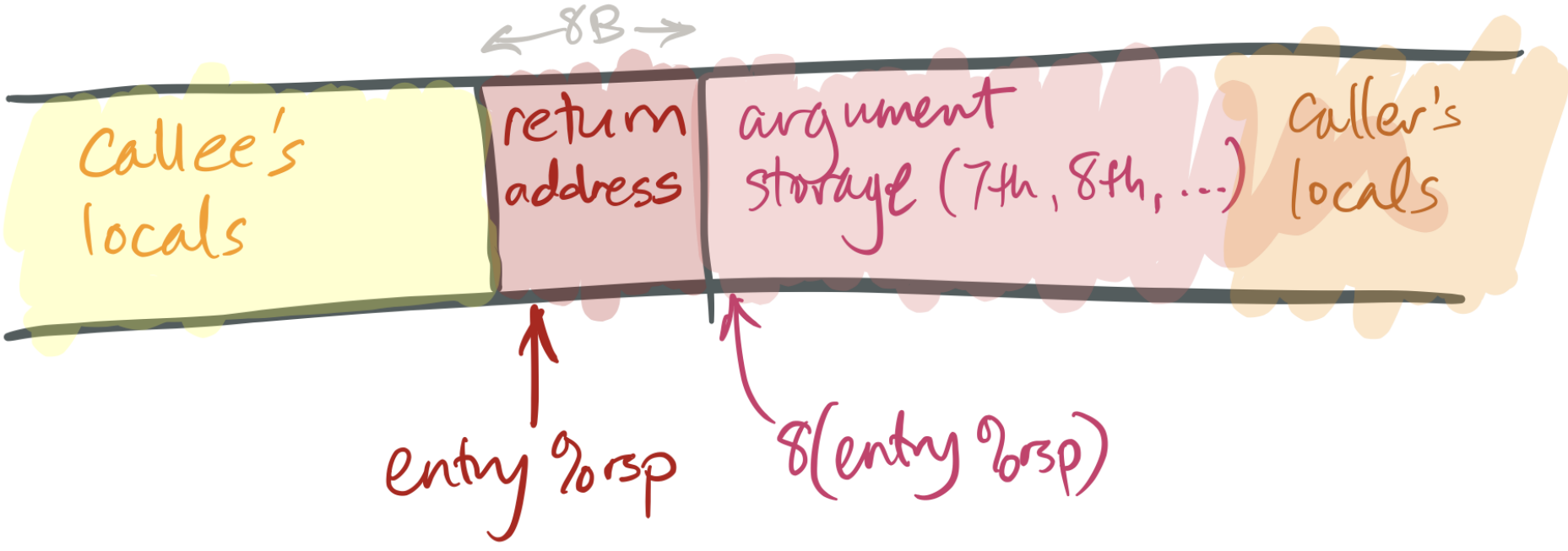
The steps required to call a function are sometimes called the *entry sequence* and the steps required to return are called the *exit sequence*. Both caller and callee have responsibilities in each sequence.

To prepare for a function call, the caller performs the following tasks in its entry sequence.

1. The caller stores the first six arguments in the corresponding registers.
2. If the callee takes more than six arguments, or if some of its arguments are large, the caller must store the surplus arguments on its stack frame. It stores these in increasing order, so that the 7th argument has a smaller address than the 8th argument, and so forth. The 7th argument must be stored at `(%rsp)` (that is, the top of the stack) when the caller executes its `callq` instruction.
3. The caller saves any caller-saved registers (see below).

4. The caller executes `callq FUNCTION`. This has an effect like `pushq $NEXT_INSTRUCTION; jmp FUNCTION` (or, equivalently, `subq $8, %rsp; movq $NEXT_INSTRUCTION, (%rsp); jmp FUNCTION`), where `NEXT_INSTRUCTION` is the address of the instruction immediately following `callq`.

This leaves a stack like this:



To return from a function:

1. The callee places its return value in `%rax`.
2. The callee restores the stack pointer to its value at entry ("entry `%rsp`"), if necessary.
3. The callee executes the `retq` instruction. This has an effect like `popq %rip`, which removes the return address from the stack and jumps to that address.
4. The caller then cleans up any space it prepared for arguments and restores caller-saved registers if necessary.

Particularly simple callees don't need to do much more than return, but most callees will perform more tasks, such as allocating space for local variables and calling functions themselves.

Callee-saved registers and caller-saved registers

The calling convention gives callers and callees certain guarantees and responsibilities about the values of registers across function calls. Function implementations may expect these guarantees to hold, and must work to fulfill their responsibilities.

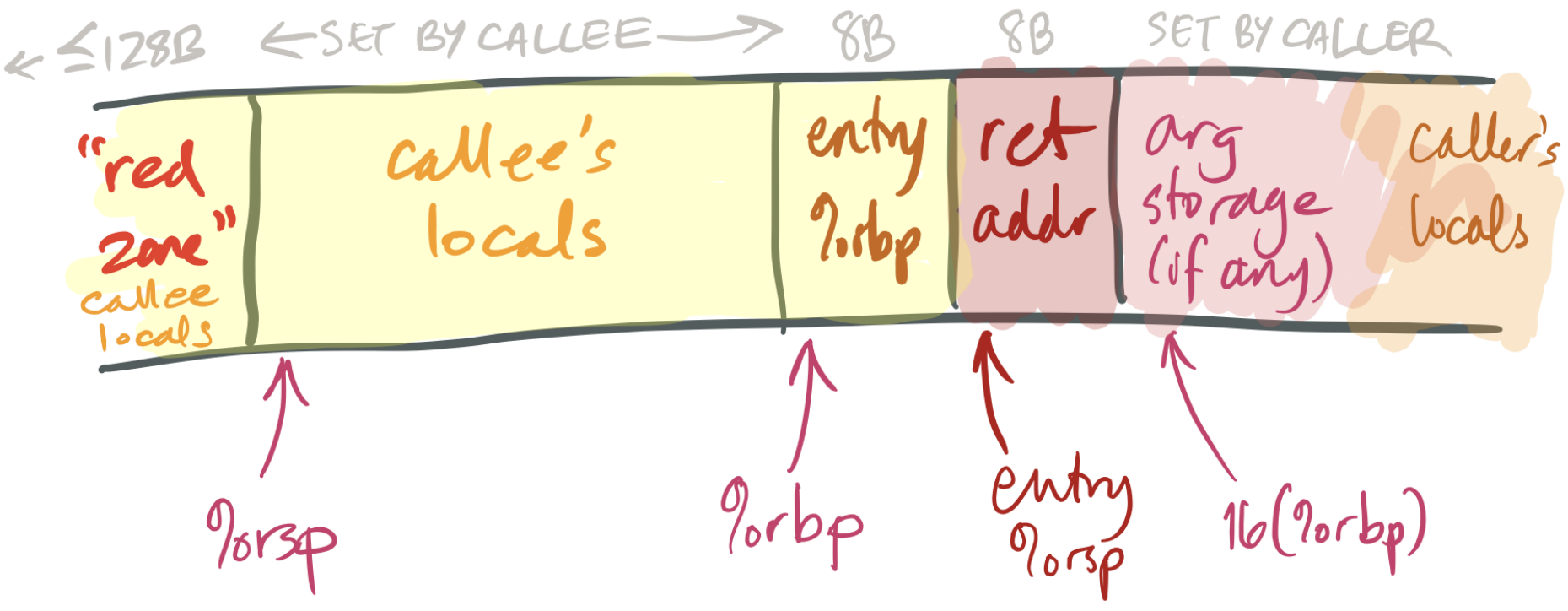
The most important responsibility is that certain registers' values *must be preserved across function calls*. A callee may use these registers, but if it changes them, it must restore them to their original values before returning. These registers are called **callee-saved registers**. All other registers are **caller-saved**.

Callers can simply use callee-saved registers across function calls; in this sense they behave like C++ local variables. Caller-saved registers behave differently: if a caller wants to preserve the value of a caller-saved register across a function call, the caller must explicitly save it before the `callq` and restore it when the function resumes.

On x86-64 Linux, `%rbp`, `%rbx`, `%r12`, `%r13`, `%r14`, and `%r15` are callee-saved, as (sort of) are `%rsp` and `%rip`. The other registers are caller-saved.

Base pointer (frame pointer)

The `%rbp` register is called the *base pointer* (and sometimes the *frame pointer*). For simple functions, an optimizing compiler generally treats this like any other callee-saved general-purpose register. However, for more complex functions, `%rbp` is used in a specific pattern that facilitates debugging. It works like this:



1. The first instruction executed on function entry is `pushq %rbp`. This saves the caller's value for `%rbp` into the callee's stack. (Since `%rbp` is callee-saved, the callee must save it.)
2. The second instruction is `movq %rsp, %rbp`. This saves the current stack pointer in `%rbp` (so `%rbp` = entry `%rsp` - 8).

   This adjusted value of `%rbp` is the callee's "frame pointer." The callee will not change this value until it returns. The frame pointer provides a stable reference point for local variables and caller arguments. (Complex functions may need a stable reference point because they reserve varying amounts of space for calling different functions.)

   Note, also, that the value stored at `(%rbp)` is the *caller's* `%rbp`, and the value stored at `8(%rbp)` is the return address. This information can be used to trace backwards through callers' stack frames by functions such as debuggers.
3. The function ends with `movq %rbp, %rsp; popq %rbp; retq`, or, equivalently, `leave; retq`. This sequence restores the caller's `%rbp` and entry `%rsp` before returning.

Stack size and red zone

Functions execute fast because allocating space within a function is simply a matter of decrementing `%rsp`. This is much cheaper than a call to `malloc` or `new`! But making this work takes a lot of machinery. We'll see this in more detail later; but in brief: The operating system knows that `%rsp` points to the stack, so if a function accesses nonexistent memory near `%rsp`, the OS assumes it's for the stack and transparently allocates new memory there.

So how can a program "run out of stack"? The operating system puts a limit on each function's stack, and if `%rsp` gets too low, the program segmentation faults.

The diagram above also shows a nice feature of the x86-64 architecture, namely the **red zone**. This is a small area *above* the stack pointer (that is, at lower addresses than `%rsp`) that can be used by the currently-running function for local variables. The red zone is nice because it can be used without mucking around with the stack pointer; for small functions `push` and `pop` instructions end up taking time.

Branches

The processor typically executes instructions in sequence, incrementing `%rip` each time. Deviations from sequential instruction execution, such as function calls, are called **control flow transfers**.

Function calls aren't the only kind of control flow transfer. A *branch* instruction jumps to a new instruction without saving a return address on the stack.

Branches come in two flavors, unconditional and conditional. The `jmp` or `j` instruction executes an unconditional branch (like a `goto`). All other branch instructions are conditional: they only branch if some condition holds. That condition is represented by **condition flags** that are set as a side effect of every arithmetic operation.

Arithmetic instructions change part of the `%rflags` register as a side effect of their operation. The most often used flags are:

- **ZF** (zero flag): set iff the result was zero.
- **SF** (sign flag): set iff the most significant bit (the sign bit) of the result was one (i.e., the result was negative if considered as a signed integer).
- **CF** (carry flag): set iff the result overflowed when considered as unsigned (i.e., the result was greater than 2^W-1).

- **OF** (overflow flag): set iff the result overflowed when considered as signed (i.e., the result was greater than $2^{W-1}-1$ or less than -2^{W-1}).

Although some instructions let you load specific flags into registers (e.g., `setz`; see CS:APP3e §3.6.2, p203), code more often accesses them via conditional jump or conditional move instructions.

Instruction	Mnemonic	C example	Flags
j (jmp)	Jump	`break;`	(Unconditional)
je (jz)	Jump if equal (zero)	`if (x == y)`	ZF
jne (jnz)	Jump if not equal (nonzero)	`if (x != y)`	!ZF
jg (jnle)	Jump if greater	`if (x > y)`, signed	!ZF && !(SF ^ OF)
jge (jnl)	Jump if greater or equal	`if (x >= y)`, signed	!(SF ^ OF)
jl (jnge)	Jump if less	`if (x < y)`, signed	SF ^ OF
jle (jng)	Jump if less or equal	`if (x <= y)`, signed	(SF ^ OF) || ZF
ja (jnbe)	Jump if above	`if (x > y)`, unsigned	!CF && !ZF
jae (jnb)	Jump if above or equal	`if (x >= y)`, unsigned	!CF
jb (jnae)	Jump if below	`if (x < y)`, unsigned	CF
jbe (jna)	Jump if below or equal	`if (x <= y)`, unsigned	CF || ZF
js	Jump if sign bit	`if (x < 0)`, signed	SF
jns	Jump if not sign bit	`if (x >= 0)`, signed	!SF
jc	Jump if carry bit	N/A	CF
jnc	Jump if not carry bit	N/A	!CF
jo	Jump if overflow bit	N/A	OF
jno	Jump if not overflow bit	N/A	!OF

The `test` and `cmp` instructions are frequently seen before a conditional branch. These operations perform arithmetic but throw away the result, except for condition codes. `test` performs binary-and, `cmp` performs subtraction.

`cmp` is hard to grasp: remember that `subq %rax, %rbx` performs `%rbx := %rbx - %rax`—the source/destination operand is on the left. So `cmpq %rax, %rbx` evaluates `%rbx - %rax`. The sequence `cmpq %rax, %rbx; jg L` will jump to label `L` if and only if `%rbx` is greater than `%rax` (signed).

The weird-looking instruction `testq %rax, %rax`, or more generally `testq REG, SAMEREG`, is used to load the condition flags appropriately for a single register. For example, the bitwise-and of `%rax` and `%rax` is zero if and only if `%rax` is zero, so `testq %rax, %rax; je L` jumps to `L` if and only if `%rax` is zero.

Sidebar: C++ data structures

C++ compilers and data structure implementations have been designed to avoid the so-called *abstraction penalty*, which is when convenient data structures compile to more and more-expensive instructions than simple, raw memory accesses. When this works, it works quite well; for example, this:

```
long f(std::vector<int>& v) {
    long sum = 0;
    for (auto& i : v) {
        sum += i;
    }
    return sum;
}
```

compiles to this, a very tight loop similar to the C version:

```
        movq    (%rdi), %rax
        movq    8(%rdi), %rcx
        cmpq    %rcx, %rax
        je      .L4
        movq    %rax, %rdx
        addq    $4, %rax
        subq    %rax, %rcx
        andq    $-4, %rcx
        addq    %rax, %rcx
        movl    $0, %eax
.L3:
        movslq  (%rdx), %rsi
        addq    %rsi, %rax
        addq    $4, %rdx
        cmpq    %rcx, %rdx
        jne     .L3
        rep ret
.L4:
        movl    $0, %eax
        ret
```

We can also use this output to infer some aspects of `std::vector`'s implementation. It looks like:

- The first element of a `std::vector` structure is a pointer to the first element of the vector;
- The elements are stored in memory in a simple array;
- The second element of a `std::vector` structure is a pointer to *one-past-the-end* of the elements of the vector (i.e., if the vector is empty, the first and second elements of the structure have the same value).