This is not the current version of the class.


# Assembly

## Contents



## Registers

Registers are the fastest kind of memory available in the machine. x86-64 has 14 general-purpose registers and several special-purpose registers. This table gives all the basic registers, with special-purpose registers highlighted in yellow. You'll notice different naming conventions, a side effect of the long history of the x86 architecture (the 8086 was first released in 1978).

| **Full register** (bits 0-63) | 32-bit (bits 0–31) | 16-bit (bits 0–15) | 8-bit low (bits 0–7) | 8-bit high (bits 8–15) | **Use in calling convention** | **Callee-saved?** |
|---|---|---|---|---|---|---|
| **General-purpose registers:** | | | | | | |
| **%rax** | %eax | %ax | %al | %ah | Return value (accumulator) | No |
| **%rbx** | %ebx | %bx | %bl | %bh | – | **Yes** |
| **%rcx** | %ecx | %cx | %cl | %ch | 4th function argument | No |
| **%rdx** | %edx | %dx | %dl | %dh | 3rd function argument | No |
| **%rsi** | %esi | %si | %sil | – | 2nd function argument | No |
| **%rdi** | %edi | %di | %dil | – | 1st function argument | No |
| **%r8** | %r8d | %r8w | %r8b | – | 5th function argument | No |
| **%r9** | %r9d | %r9w | %r9b | – | 6th function argument | No |
| **%r10** | %r10d | %r10w | %r10b | – | – | No |
| **%r11** | %r11d | %r11w | %r11b | – | – | No |
| **%r12** | %r12d | %r12w | %r12b | – | – | **Yes** |

| Full register (bits 0-63) | 32-bit (bits 0–31) | 16-bit (bits 0–15) | 8-bit low (bits 0–7) | 8-bit high (bits 8–15) | Use in calling convention | Callee-saved? |
|---|---|---|---|---|---|---|
| **%r13** | %r13d | %r13w | %r13b | – | – | **Yes** |
| **%r14** | %r14d | %r14w | %r14b | – | – | **Yes** |
| **%r15** | %r15d | %r15w | %r15b | – | – | **Yes** |
| **Special-purpose registers:** | | | | | | |
| **%rsp** | %esp | %sp | %spl | – | Stack pointer | **Yes** |
| **%rbp** | %ebp | %bp | %bpl | – | Base pointer (general-purpose in some compiler modes) | **Yes** |
| **%rip** | %eip | %ip | – | – | Instruction pointer (Program counter; called $pc in GDB) | * |
| **%rflags** | %eflags | %flags | – | – | Flags and condition codes | No |

Note that unlike *primary* memory (which is what we think of when we discuss memory in a C/C++ program), registers have no addresses! There is no address value that, if cast to a pointer and dereferenced, would return the contents of the `%rax` register. Registers live in a separate world from the memory whose contents are partially prescribed by the C abstract machine.

The %rbp register has a special purpose: it points to the bottom of the current function's stack frame, and local variables are often accessed relative to its value. However, when optimization is on, the compiler may determine that all local variables can be stored in registers. This frees up %rbp for use as another general-purpose register.

The relationship between different register bit widths is a little weird.

1. Modifying a 32-bit register name sets the *upper* 32 bits of the register to zero. Thus, after `movl $-1, %eax`, the `%rax` register has value 0x0000'0000'FFFF'FFFF. The same is true after `movq $-1, %rax; addl $0, %eax`! (The `movq` sets `%rax` to 0xFFFF'FFFF'FFFF'FFFF; the `addl` sets its upper 32 bits to zero.)
2. Modifying a 16- or 8-bit register name *leaves all other bits of the register unchanged.*

There are special instructions for loading signed and unsigned 8-, 16-, and 32-bit quantities into registers, recognizable by instruction suffixes. For instance, `movzbl` moves an 8-bit quantity (a **b**yte) into a 32-bit register (a **l**ongword) with **z**ero extension; `movslq` moves a 32-bit quantity (**l**ongword) into a 64-bit register (**q**uadword) with **s**ign extension. There's no need for `movzlq` (why?).

## Instruction format

The basic kinds of assembly instructions are:

1. **Computation.** These instructions perform computation on values, typically values stored in registers. Most have zero or one *source operands* and one *source/destination operand*, with the source operand coming first. For example, the instruction `addq %rax, %rbx` performs the computation `%rbx := %rbx + %rax`.
2. **Data movement.** These instructions move data between registers and memory. Almost all have one source operand and one destination operand; the source operand comes first.
3. **Control flow.** Normally the CPU executes instructions in sequence. Control flow instructions change the instruction pointer in other ways. There are unconditional branches (the instruction pointer is set to a new value), conditional branches (the instruction pointer is set to a new value if a condition is true), and function call and return instructions.

(We use the "AT&T syntax" for x86-64 assembly. For the "Intel syntax," which you can find in online documentation from Intel, see the Aside in CS:APP3e §3.3, p177, or Wikipedia, or other online resources. AT&T syntax is distinguished by several features, but especially by the use of percent signs for registers. Sadly, the Intel syntax puts destination registers *before* source registers.)

Some instructions appear to combine computation and data movement. For example, given the C code `int* ip; ... ++(*ip);` the compiler might generate `incl (%rax)` rather than `movl (%rax), %ebx; incl %ebx; movl %ebx, (%rax)`. However, the processor actually divides these complex instructions into tiny, simpler, invisible instructions called microcode, because the simpler instructions can be made to execute faster. The complex `incl` instruction actually runs in three phases: data movement, then computation, then data movement. This matters when we introduce parallelism.

## Directives

Assembly generated by a compiler contains instructions as well as *labels* and *directives*. Labels look like `labelname:` or `labelnumber:`; directives look like `.directivename arguments`. Labels are markers in the generated assembly, used to compute addresses. We usually see them used in control flow instructions, as in `jmp L3` ("jump to L3"). Directives are instructions to the assembler; for instance, the `.globl L` instruction says "label `L` is globally visible in the executable", `.align` sets the alignment of the following data, `.long` puts a number in the output, and `.text` and `.data` define the current segment.

We also frequently look at assembly that is *disassembled* from executable instructions by GDB, `objdump -d`, or `objdump -S`. This output looks different from compiler-generated assembly: in disassembled instructions, there are no intermediate labels or directives. This is because the labels and directives disappear during the process of generating executable instructions.

For instance, here is some compiler-generated assembly:

```
    .globl  _Z1fiii
    .type   _Z1fiii, @function
_Z1fiii:
.LFB0:
    cmpl    %edx, %esi
    je  .L3
    movl    %esi, %eax
    ret
.L3:
    movl    %edi, %eax
    ret
.LFE0:
    .size   _Z1fiii, .-_Z1fiii
```

And a disassembly of the same function, from an object file:

```
0000000000000000 <_Z1fiii>:
   0:   39 d6                   cmp    %edx,%esi
   2:   74 03                   je     7 <_Z1fiii+0x7>
   4:   89 f0                   mov    %esi,%eax
   6:   c3                      retq
   7:   89 f8                   mov    %edi,%eax
   9:   c3                      retq
```

Everything but the instructions is removed, and the helpful `.L3` label has been replaced with an actual address. The function appears to be located at address 0. This is just a placeholder; the final address is assigned by the linking process, when a final executable is created.

Finally, here is some disassembly from an executable:

```
0000000000400517 <_Z1fiii>:
  400517:   39 d6                   cmp    %edx,%esi
  400519:   74 03                   je     40051e <_Z1fiii+0x7>
  40051b:   89 f0                   mov    %esi,%eax
  40051d:   c3                      retq
  40051e:   89 f8                   mov    %edi,%eax
  400520:   c3                      retq
```

The instructions are the same, but the addresses are different. (Other compiler flags would generate different addresses.)

## Address modes

Most instruction operands use the following syntax for values. (See also CS:APP3e Figure 3.3 in §3.4.1, p181.)

| Type | Example syntax | Value used |
|---|---|---|
| Register | `%rbp` | Contents of `%rbp` |
| Immediate | `$0x4` | 0x4 |
| Memory | `0x4` | Value stored at address 0x4 |
| | `symbol_name` | Value stored in global `symbol_name`<br>(the compiler resolves the symbol name to an address when creating the executable) |
| | `symbol_name(%rip)` | `%rip`-relative addressing for global |
| | `symbol_name+4(%rip)` | Simple computations on symbols are allowed<br>(the compiler resolves the computation when creating the executable) |
| | `(%rax)` | Value stored at address in `%rax` |
| | `0x4(%rax)` | Value stored at address `%rax + 4` |
| | `(%rax,%rbx)` | Value stored at address `%rax + %rbx` |
| | `(%rax,%rbx,4)` | Value stored at address `%rax + %rbx*4` |
| | `0x18(%rax,%rbx,4)` | Value stored at address `%rax + 0x18 + %rbx*4` |

The full form of a memory operand is `offset(base,index,scale)`, which refers to the address `offset + base + index*scale`. In `0x18(%rax,%rbx,4)`, `%rax` is the base, `0x18` the offset, `%rbx` the index, and `4` the scale. The offset (if used) must be a constant and the base and index (if used) must be registers; the scale must be either 1, 2, 4, or 8. The default offset, base, and index are 0, and the default scale is 1.

`symbol_name`s are found only in compiler-generated assembly; disassembly uses raw addresses (`0x601030`) or `%rip`-relative offsets (`0x200bf2(%rip)`).

Jumps and function call instructions use different syntax 🤷🏾‍♀️: `*`, rather than `()`, represents indirection.

| Type | Example syntax | Address used |
|---|---|---|
| Register | `*%rax` | Contents of `%rax` |
| Immediate | `.L3` | Address of `.L3` (compiler-generated assembly) |
| | `400410` or `0x400410` | Given address |
| Memory | `*0x200b96(%rip)` | Value stored at address `%rip + 0x200b96` |
| | `*(%r12,%rbp,8)` | Other address modes accepted |

## Address computations

The `offset(base,index,scale)` form compactly expresses many array-style address computations. It's typically used with a `mov`-type instruction to dereference memory. However, the compiler can use that form to compute addresses, thanks to the `lea` (Load Effective Address) instruction.

For instance, in `movl 0x18(%rax,%rbx,4), %ecx`, the address `%rax + 0x18 + %rbx*4` is computed, then immediately dereferenced: the 4-byte value located there is loaded into `%ecx`. In `leaq 0x18(%rax,%rbx,4), %rcx`, the same address is computed, but it is *not* dereferenced. Instead, the *computed address* is moved into register `%rcx`.

Thanks to `lea`, the compiler will also prefer the `offset(base,index,scale)` form over `add` and `mov` for certain computations on integers. For example, this instruction:

```
leaq (%rax,%rbx,2), %rcx
```

performs the function `%rcx := %rax + 2 * %rbx`, but in one instruction, rather than three (`movq %rax, %rcx; addq %rbx, %rcx; addq %rbx, %rcx`).

## `%rip`-relative addressing

x86-64 code often refers to globals using **%rip-relative** addressing: a global variable named `a` is referenced as `a(%rip)` rather than `a`.

This style of reference supports *position-independent code* (PIC), a security feature. It specifically supports *position-independent executables* (PIEs), which are programs that work independently of where their code is loaded into memory.

To run a conventional program, the operating system loads the program's instructions into memory *at a fixed address that's the same every time*, then starts executing the program at its first instruction. This works great, and runs the program in a predictable execution environment (the addresses of functions and global variables are the same every time). Unfortunately, the very predictability of this environment makes the program easier to attack.

In a position-independent executable, the operating system loads the program at *varying* locations: every time it runs, the program's functions and global variables have different addresses. This makes the program harder to attack (though not impossible).

Program startup performance matters, so the operating system doesn't recompile the program with different addresses each time. Instead, the compiler does most of the work in advance by using *relative addressing*.

When the operating system loads a PIE, it picks a starting point and loads all instructions and globals relative to that starting point. The PIE's instructions never refer to global variables using direct addressing: you'll never see `movl global_int, %eax`. Globals are referenced *relatively* instead, using deltas relative to the next `%rip`: `movl global_int(%rip), %eax`. These relative addresses work great independent of starting point! For instance, consider an instruction located at (starting-point + 0x80) that loads a variable `g` located at (starting-point + 0x1000) into `%rax`. In a non-PIE, the instruction might be written `movq 0x400080, %rax` (in compiler output, `movq g, %rax`); but this relies on `g` having a fixed address. In a PIE, the instruction might be written `movq 0xf79(%rip), %rax` (in compiler output, `movq g(%rip), %rax`), which works out beautifully no matter the starting point.

| At starting point... | The `mov` instruction is at... | The next instruction is at... | And `g` is at... | So the delta (`g` - next `%rip`) is... |
|---|---|---|---|---|
| 0x400000 | 0x400080 | 0x400087 | 0x401000 | 0xF79 |
| 0x404000 | 0x404080 | 0x404087 | 0x405000 | 0xF79 |
| 0x4003F0 | 0x400470 | 0x400477 | 0x4013F0 | 0xF79 |

## 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
```

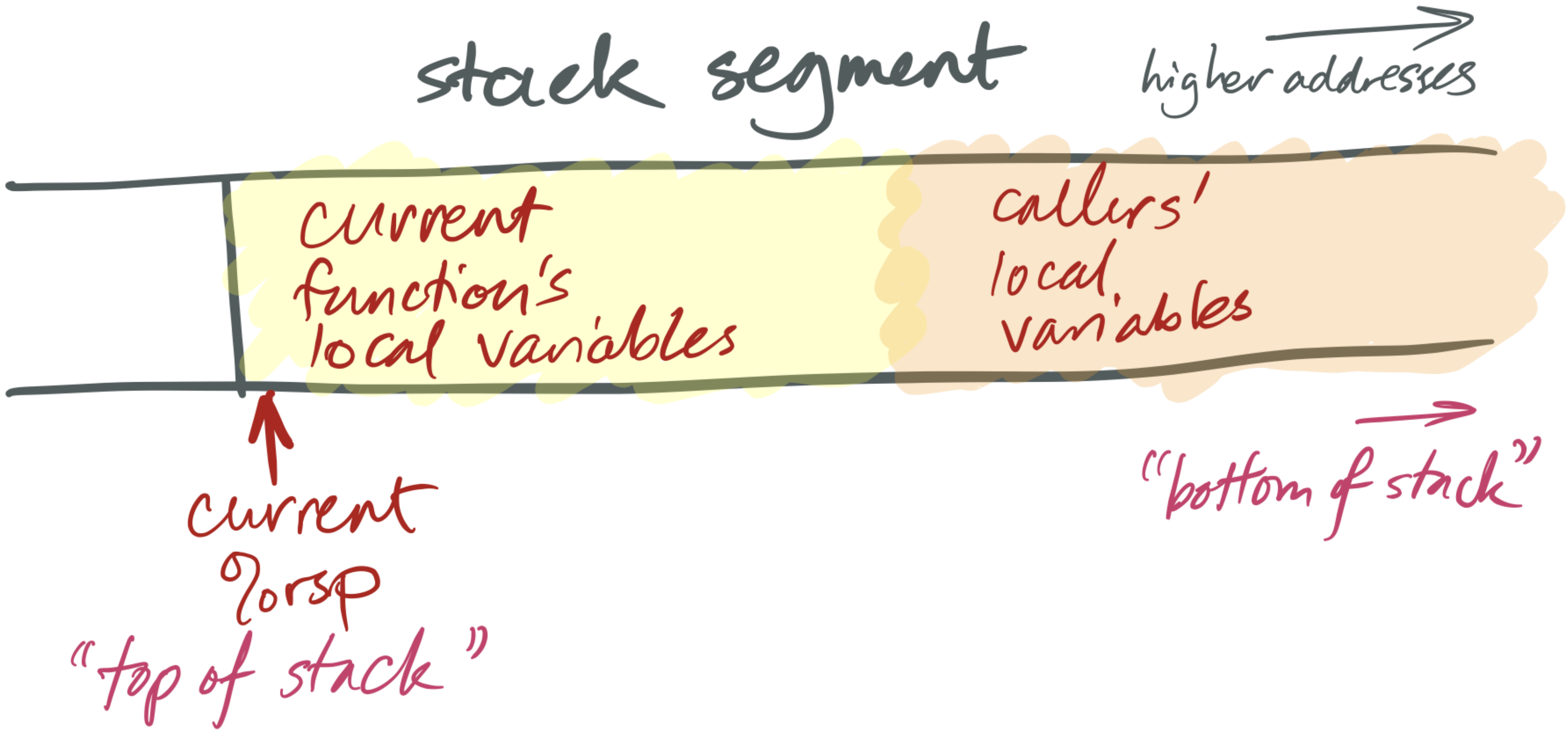
## Stack

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`, `call`, and `ret`. 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.

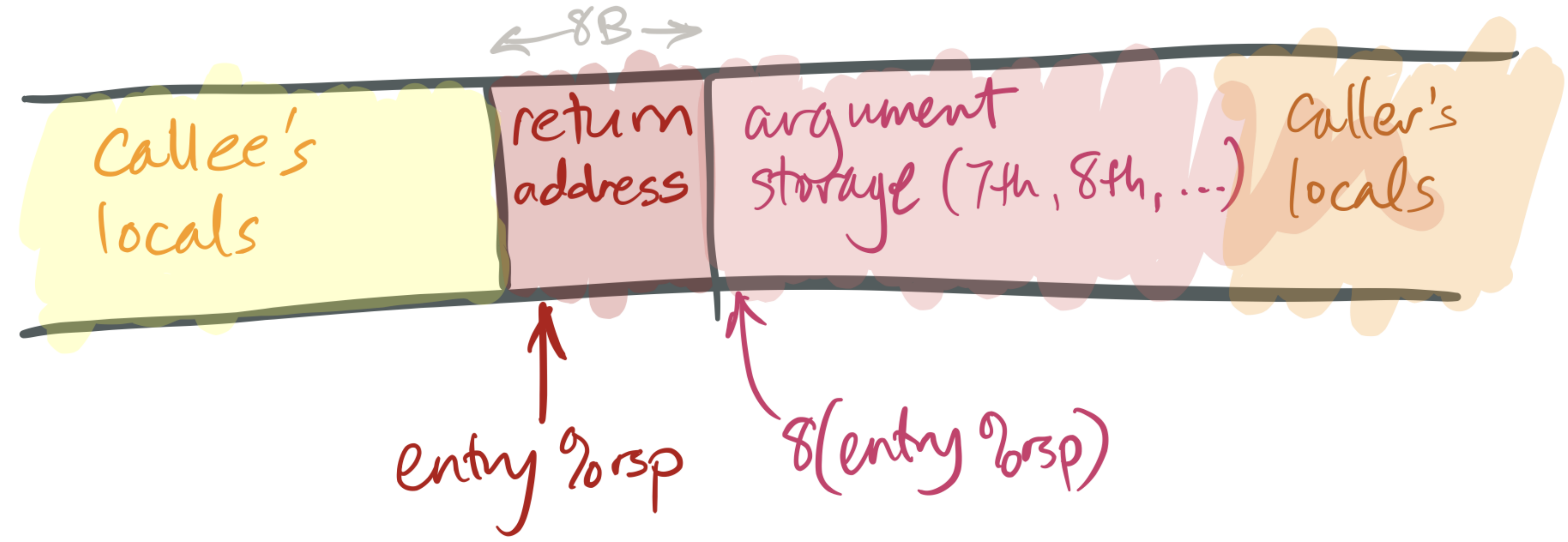
## Return address and entry and exit sequence

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.

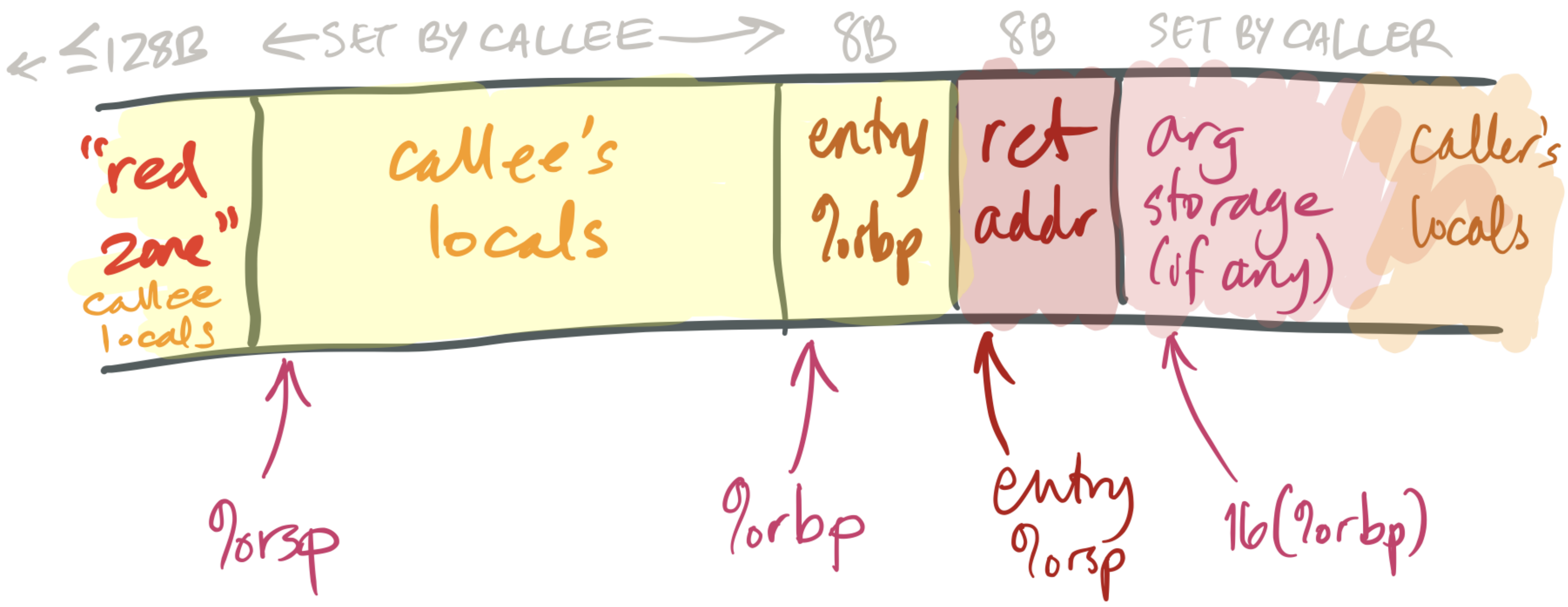
The most important responsibility is that certain registers' values *must be preserved across function calls*. A callee may use these registers, but if it changes them, it must restore them to their original values before returning. These registers are called **callee-saved registers**. All other registers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



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 |

| Instruction | Mnemonic | C example | Flags |
|---|---|---|---|
| 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).

## Compiler optimizations

### Argument elision

A compiler may decide to elide (or remove) certain operations setting up function call arguments, if it can decide that the registers containing these arguments will hold the correct value before the function call takes place. Let's see an example of a function disassembled function `f` in `f31.s`:

```
subq    $8, %rsp
call    _Z1gi@PLT
addq    $8, %rsp
addl    $1, %eax
ret
```

This function calls another function `g`, adds 1 to `g`'s return value, and returns that value.

It is possible that the function has the following definition in C++:

```
int f() {
    return 1 + g();
}
```

However, the actual definition of `f` in `f31.cc` is:

```
int f(x) {
    return 1 + g(x);
}
```

The compiler realizes that the argument to function `g`, which is passed via register `%rdi`, already has the right value when `g` is called, so it doesn't bother doing anything about it. This is one example of numerous optimizations a compiler can perform to reduce the size of generated code.

## Inlining

A compiler may also copy the body of function to its call site, instead of doing an explicit function call, when it decides that the overhead of performing a function call outweights the overhead of doing this copy. For example, if we have a function `g` defined as `g(x) = 2 + x`, and `f` is defined as `f(x) = 1 + g(x)`, then the compiler may actually generate `f(x)` as simply `3 + x`, without inserting any `call` instructions. In assembly terms, function `g` will look like

```
leal    2(%rdi), %eax
ret
```

and `f` will simply be

```
leal    3(%rdi), %eax
ret
```

## Tail call elimination

Let's look at another example in `f32.s`:

```
addl    $1, %edi
jmp _Z1gi@PLT
```

This function doesn't even contain a `ret` instruction! What is going on? Let's take a look at the actual definition of `f`, in `f32.cc`:

```
int f(int x) {
    return g(x + 1);
}
```

Note that the call to function `g` is the last operation in function `f`, and the return value of `f` is just the return value of the invocation of `g`. In this case the compiler can perform a *tail call elimination*: instead of calling `g` explicitly, it can simply jump to `g` and have `g` return to the same address that `f` would have returned to.

> A tail call elimination may occur if a function (caller) ends with another function call (callee) and performs no cleanup once the callee returns. In this case the caller and simply jump to the callee, instead of doing an explicit call.

## Loop unrolling

Before we jump into loop unrolling, let's take a small excursion into an aspect of calling conventions called caller/callee-saved registers. This will help us under the sample program in `f33.s` better.

### *Calling conventions: caller/callee-saved registers*

Let's look at the function definition in `f33.s`:

```
    pushq   %r12
    pushq   %rbp
    pushq   %rbx
    testl   %edi, %edi
    je  .L4
    movl    %edi, %r12d
    movl    $0, %ebx
    movl    $0, %ebp
.L3:
    movl    %ebx, %edi
    call    _Z1gj@PLT
    addl    %eax, %ebp
    addl    $1, %ebx
    cmpl    %ebx, %r12d
    jne .L3
.L1:
    movl    %ebp, %eax
    popq    %rbx
    popq    %rbp
    popq    %r12
    ret
.L4:
    movl    %edi, %ebp
    jmp .L1
```

From the assembly we can tell that the backwards jump to `.L3` is likely a loop. The loop index is in `%ebx` and the loop bound is in `%r12d`. Note that upon entry to the function we first moved the value `%rdi` to `%r12d`. This is necessary because in the loop `f` calls `g`, and `%rdi` is used to pass arguments to `g`, so we must move its value to a different register to used it as the loop bound (this case `%r12`). But there is more to this: the compiler also needs to ensure that this register's value is preserved across function calls. Calling conventions dictate that certain registers always exhibit this property, and they are called **callee-saved registers**. If a register is callee-saved, then the caller doesn't have to save its value before entering a function call.

We note that upon entry to the function, `f` saved a bunch of registers by pushing them to the stack: `%r12`, `%rbp`, `%rbx`. It is because all these registers are callee-saved registers, and `f` uses them during the function call. In general, the following registers in x86_64 are callee-saved:
`%rbx`, `%r12-%r15`, `%rbp`, `%rsp` (`%rip`)

All the other registers are **caller-saved**, which means the callee doesn't have to preserve their values. If the caller wants to reuse values in these registers across function calls, it will have to explicitly save and restore these registers. In general, the following registers in x86_64 are caller-saved:
`%rax`, `%rcx`, `%rdx`, `%r8-%r11`

Now let's get back to loop unrolling. Let us a look at the program in `f34.s`:

```
    testl   %edi, %edi
    je  .L7
    leal    -1(%rdi), %eax
    cmpl    $7, %eax
    jbe .L8
    pxor    %xmm0, %xmm0
    movl    %edi, %edx
    xorl    %eax, %eax
    movdqa  .LC0(%rip), %xmm1
    shrl    $2, %edx
    movdqa  .LC1(%rip), %xmm2
.L5:
    addl    $1, %eax
    paddd   %xmm1, %xmm0
    paddd   %xmm2, %xmm1
    cmpl    %edx, %eax
    jb  .L5
    movdqa  %xmm0, %xmm1
    movl    %edi, %edx
    andl    $-4, %edx
    psrldq  $8, %xmm1
    paddd   %xmm1, %xmm0
    movdqa  %xmm0, %xmm1
    cmpl    %edx, %edi
    psrldq  $4, %xmm1
    paddd   %xmm1, %xmm0
    movd    %xmm0, %eax
    je  .L10
.L3:
    leal    1(%rdx), %ecx
    addl    %edx, %eax
    cmpl    %ecx, %edi
    je  .L1
    addl    %ecx, %eax
    leal    2(%rdx), %ecx
    cmpl    %ecx, %edi
    je  .L1
    addl    %ecx, %eax
    leal    3(%rdx), %ecx
    cmpl    %ecx, %edi
    je  .L1
    addl    %ecx, %eax
    leal    4(%rdx), %ecx
    cmpl    %ecx, %edi
    je  .L1
    addl    %ecx, %eax
    leal    5(%rdx), %ecx
    cmpl    %ecx, %edi
    je  .L1
    addl    %ecx, %eax
    leal    6(%rdx), %ecx
    cmpl    %ecx, %edi
    je  .L1
    addl    %ecx, %eax
    addl    $7, %edx
    leal    (%rax,%rdx), %ecx
    cmpl    %edx, %edi
    cmovne  %ecx, %eax
    ret
.L7:
    xorl    %eax, %eax
.L1:
    rep ret
.L10:
    rep ret
.L8:
    xorl    %edx, %edx
    xorl    %eax, %eax
    jmp .L3
```

Wow this looks long and repetitive! Especially the section under label `.L3`! If we take a look at the original function definition in `f34.cc`, we will find that it's almost the same as `f33.cc`, except that in `f34.cc` we know the definition of `g` as well. With knowledge of what `g` does the compiler's optimizer decides that unrolling the loop into 7-increment batches results in faster code.

Code like this can become difficult to understand, especially when the compiler begins to use more advanced registers reserved for vector operations. We can fine-tune the optimizer to disable certain optimizations. For example, we can use the `-mno-sse -fno-unroll-loops` compiler options to disable the use of SSE registers and loop unrolling. The resulting code, in `f35.s`, for the same function definitions in `f34.cc`, becomes much easier to understand:

```
    testl   %edi, %edi
    je  .L4
    xorl    %edx, %edx
    xorl    %eax, %eax
.L3:
    addl    %edx, %eax
    addl    $1, %edx
    cmpl    %edx, %edi
    jne .L3
    rep ret
.L4:
    xorl    %eax, %eax
    ret
```

Note that the compiler still performed inlining to eliminate function `g`.

## Optimizing recursive functions

Let's look at the following recursive function in `f36.cc`:

```
int f(int x) {
    if (x > 0) {
        return x * f(x - 1);
    } else {
        return 0;
    }
}
```

At the first glance it may seem that the function returns factorial of `x`. But it actually returns 0. Despite it doing a series of multiplications, in the end it always multiplies the whole result with 0, which produces 0.

When we compile this function to assembly without much optimization, we see the expensive computation occurring:

```
    movl    $0, %eax
    testl   %edi, %edi
    jg  .L8
    rep ret
.L8:
    pushq   %rbx
    movl    %edi, %ebx
    leal    -1(%rdi), %edi
    call    _Z1fi
    imull   %ebx, %eax
    popq    %rbx
    ret
```

In `f37.cc` there is an actual factorial function:

```
int f(int x) {
    if (x > 0) {
        return x * f(x - 1);
    } else {
        return 1;
    }
}
```

If we compile this function using level-2 optimization (`-O2`), we get the following assembly:

```
    testl   %edi, %edi
    movl    $1, %eax
    jle .L4
.L3:
    imull   %edi, %eax
    subl    $1, %edi
    jne .L3
    rep ret
.L4:
    rep ret
```

There is no `call` instructions again! The compiler has transformed the recursive function into a loop.

If we revisit our "fake" factorial function that always returns 0, and compile it with `-O2`, we see yet more evidence of compiler's deep understanding of our program:

```
xorl    %eax, %eax
ret
```

## Optimizing arithmetic operations

The assembly code in `f39.s` looks like this:

```
leal    (%rdi,%rdi,2), %eax
leal    (%rdi,%rax,4), %eax
ret
```

It looks like some rather complex address computations! The first `leal` instruction basically loads `%eax` with value `3*%rdi` (or `%rdi + 2*%rdi`). The second `leal` multiplies the previous result by another 4, and adds another `%rdi` to it. So what it actually does is `3*%rdi*4 + %rdi`, or simply `13*%rdi`. This is also revealed in the function name in `f39.s`.

The compiler choose to use `leal` instructions instead of an explicit multiply because the two `leal` instructions actually take less space.

# Buffer overflow attacks

The `storage1/attackme.cc` file contains a particularly dangerous kind of undefined behavior, a **buffer overflow**. In a buffer overflow, an untrusted input is transferred into a *buffer*, such as an array of characters on the stack, without checking to see whether the buffer is big enough for the untrusted input.

Historically buffer overflows have caused some of the worst, and most consequential, C and C++ security holes known. They are particularly bad when the untrusted input derives from the network: then breaking into a network can be as simple as sending a packet.

The buffer overflow in `attackme.cc` derives from a **checksum function**. Our simple `checksum` takes in a pointer to the buffer, then copies that buffer to a local variable, `buf`, and processes the copy. Unfortunately, `checksum` doesn't verify that the input fits its buffer, and if the user supplies too big an argument, `checksum`'s buffer will overflow!

Buffer overflows cause undefined behavior—it's illegal to access memory outside any object. Undefined behavior is always bad, but some is worse than others; and this particular buffer overflow allows the attacker to completely own the victim program, causing it to execute *any shell command*!

The attack works as follows.

- The function's entry sequence allocates local variable space with `subq $112, %rsp`. Examining the assembly we can infer that `buf` is stored at this `%rsp`.
- The function's return address is stored at `112(%rsp)`, which is `buf.c + 112`.
- Any input `arg` with `strlen(arg) > 99` will overflow the buffer. (The 100th character is the null character that ends the string). But `arg`s with 100 to 111 characters won't cause problems, because the remaining 12 bytes of local variable space are unused (they're present to ensure stack frame alignment).
- `arg`s of 112 or more characters, however, will run into the stack slot reserved for `checksum`'s return address! An attacker can put any value they want in that location, as long as the value contains no null characters (since `checksum`'s buffer copy stops when it encounters the end of the input string).
- Overflowing the return address slot causes harm when `checksum` returns. Examining the state during the exit sequence, we see that the immediately previous instructions load `%rdi` with `buf` and call `finish_checksum`. But what a coincidence—`finish_checksum` does not modify `%rdi`! Thus, when `checksum` returns, `%rdi` will be loaded with the address of `buf`.
- Our attacker therefore supplies a carefully-chosen string that overwrites `checksum`'s return address with *the address of the `run_shell` function*. This will cause `run_shell` to run the programs defined in the initial portion of the string!

Thus, the attacker has taken control of the victim by **returning** to an unexpected library function, with carefully-chosen arguments. This attack is called a **return-to-libc attack**.

> A version of `checksum`'s copy-to-aligned-buffer technique is actually useful in practice, but real code would use a smaller buffer, processed one slice at a time.

# Attack defenses

Return-to-libc attacks used to be pretty trivial to find and execute, but recent years have seen large improvements in the robustness of typical C and C++ programs to attack. Here are just a few of the defenses we had to disable for the simple `attackme` attack to work.

**Register arguments**: In older architectures, function arguments were *all* passed on the stack. In those architectures attackers could easily determine not only the *function* returned to, but also *its arguments* (a longer buffer overflow would overwrite the stack region interpreted by the "return-callee" as arguments). `attackme` only works because `finish_checksum` happens not to modify its argument.

Modern operating systems and architectures support **position-independent code** and **position-independent executables**. These features make programs work independent of specific addresses for functions, and the operating system can put a program's functions at different addresses each time the program runs. When a program is position-independent, the address of the attacker's desired target function can't be predicted, so the attacker has to get lucky. (The x86-64 designers were smart to add `%rip`-relative addressing, since that's what enables efficient position-independent executables!)

Finally, modern compilers use a technique called **stack protection** or **stack canaries** to detect buffer overflows and stop `retq` from returning to a corrupted address. This is a super useful technique.

> It's called a "canary" in reference to the phrase "canary in a coal mine".

1. The operating system reserves a special tiny memory segment when the program starts running. This tiny memory segment is used for special purposes, such as **thread-local storage**. Although this segment can be addressed normally—it exists in memory—the operating system also ensures it can be accessed through special instruction formats like this:

   ```
   movq %fs:0x28, %rax
   ```

   The `%fs:` prefix tells the processor that `0x28` (a memory offset) is to be measured relative to the start of the thread-local storage segment.
2. A specific memory word in thread-local storage is reserved for a canary value. The operating system sets this value to a random word when starting the program.
3. The compiler adds code to function entry and exit sequences that use this value. Specifically, something like this is added to entry:

   ```
   movq %fs:0x28, REG    # load true canary to register
   pushq REG             # push canary to stack
   ```

   And something like this is added to exit:

   ```
   popq REG              # pop canary from stack
   xorq %fs:0x28, REG    # xor with true canary
   jne fail              # fail if they differ
   ```

   where the `fail` branch calls a library-provided function such as `__stack_chk_fail`.

The pushed canary is located between the function's buffers (its local variables) and the function's return address. Given that position, any buffer overflow that reaches the return address will also overwrite the canary! And the attacker can't easily guess the canary or overwrite the true canary's memory location, since both are random.

If the stack canary and the true canary don't match, the function can infer that it executed some undefined behavior. Maybe the return address was corrupted and maybe it wasn't; either way, executing undefined behavior means the program is broken, and it is safe to exit immediately.

These techniques, and others like them, are incredibly useful, important, fun, and good to understand. But they don't make C programming safe: attackers are persistent and creative. (Check out return-oriented programming.) Memory-unsafe languages like C and C++ make programming inherently risky; only programs with no undefined behavior are safe. (And they're only safe if the underlying libraries and operating system have no undefined behavior either!) Good C and C++ programmers need a healthy respect for—one might say fear of—their tools. Code cleanly, use standard libraries where possible, and test extensively under sanitizers to catch bugs like buffer overflows.

Or listen to cranky people like Trevor Jim, who says that "C is bad and you should feel bad if you don't say it is bad" and also "Legacy C/C++ code is a nuclear waste nightmare that will make you WannaCry".

## Aside: Checksums

A **checksum** is a short, usually fixed-length summary of a data buffer. Checksums have two important properties:

1. If two buffers contain the same data (the same bytes in the same order), then their checksums **must** equal. (This property is mandatory.)
2. If two buffers contain *different* data, then their checksums **should not** be equal. (This property is optional: it is OK for distinct data to have equal checksums, though in a good checksum function, this should be rare.)

Checksums have many uses, but they are often used to detect errors in data transcription. For example, most network technologies use checksums to detect link errors (like bursts of noise). To send data over a network, the sender first computes a checksum of the data. Then it transmits both data and checksum over the possibly-noisy network channel. The receiver computes its own checksum of the received data, and compares that with the sender's checksum. If the checksums match, that's a good sign that the data was transmitted correctly. If the checksums don't match, then the data can't be trusted and the receiver throws it away.

Most checksum functions map from a large domain to a smaller range—for instance, from the set of all variable-length buffers to the set of 64-bit numbers. Such functions *must* sometimes map unequal data to equal checksums, so some errors must go undetected; but a good checksum function detects common corruption errors all, or almost all, the time. For example, some checksum functions can *always* detect a single flipped bit in the buffer.

The requirements on checksums are the same as the requirements for hash codes: equal buffers have equal checksums (hash codes), and distinct buffers should have distinct checksums (hash codes). A good hash code can be a good checksum and vice versa.

The most important characteristics of a checksum function are speed and strength. Fast checksums are inexpensive to compute, but easy to break, meaning that many errors in the data aren't detected by the checksum function. Widely-used fast checksum functions include the IPv4/TCP/UDP checksum and cyclic redundancy checks; our `checksum` function is a lot like the IPv4 checksum. Strong checksums, also known as cryptographic hash functions, are hard to break. Ideally they are *infeasible* to break, meaning that no one knows a procedure for finding two buffers that have the same checksum. Widely-used strong checksum functions include SHA-256, SHA-512, and SHA-3. There are also many formerly-strong checksum functions, such as SHA-1, that have been broken—meaning that procedures are known for finding two buffers with the same checksum—but are still stronger in practice than fast checksums.

References: Checksums on Wikipedia; CS 225 explores some related theory.