Machine prog: Procedures

韩汶辰

2019 年 9 月 26 日

Contents

Contents

Overview

- 在高级语言中，一个函数调用另一个函数时，底层的汇编需要做什么？
  - passing control, 确保过得去，回得来
  - passing data between caller and callee，包括参数和返回值
  - managing local data，存储局部变量，保证执行完子函数后局部变量还好好的.

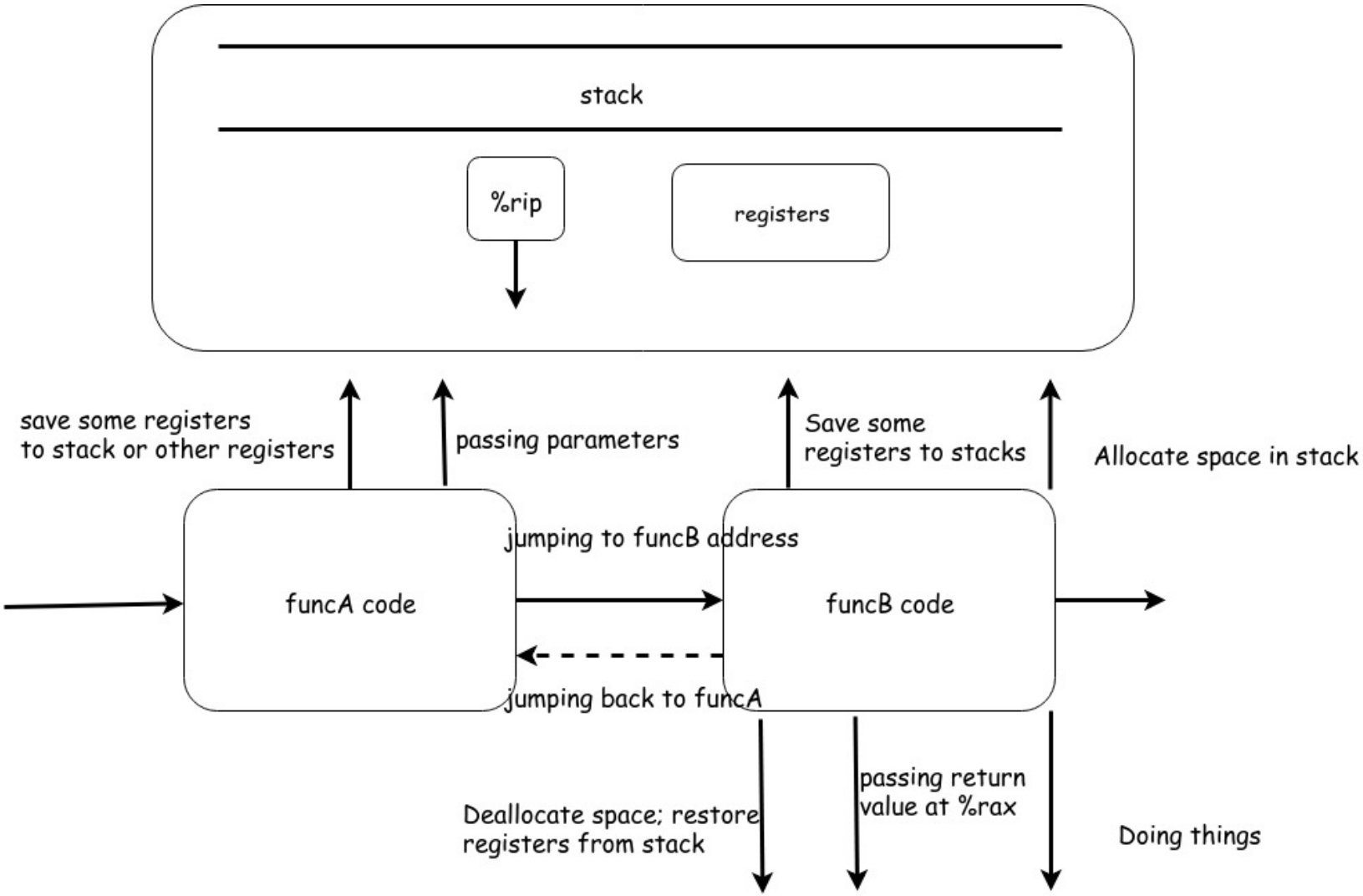
Overview

- 在高级语言中，一个函数调用另一个函数时，底层的汇编需要做什么？
  - passing control, 确保过得去，回得来
  - passing data between caller and callee，包括参数和返回值
  - managing local data，存储局部变量，保证执行完子函数后局部变量还好好的.
- 函数的调用过程本身就是 FILO 的过程.

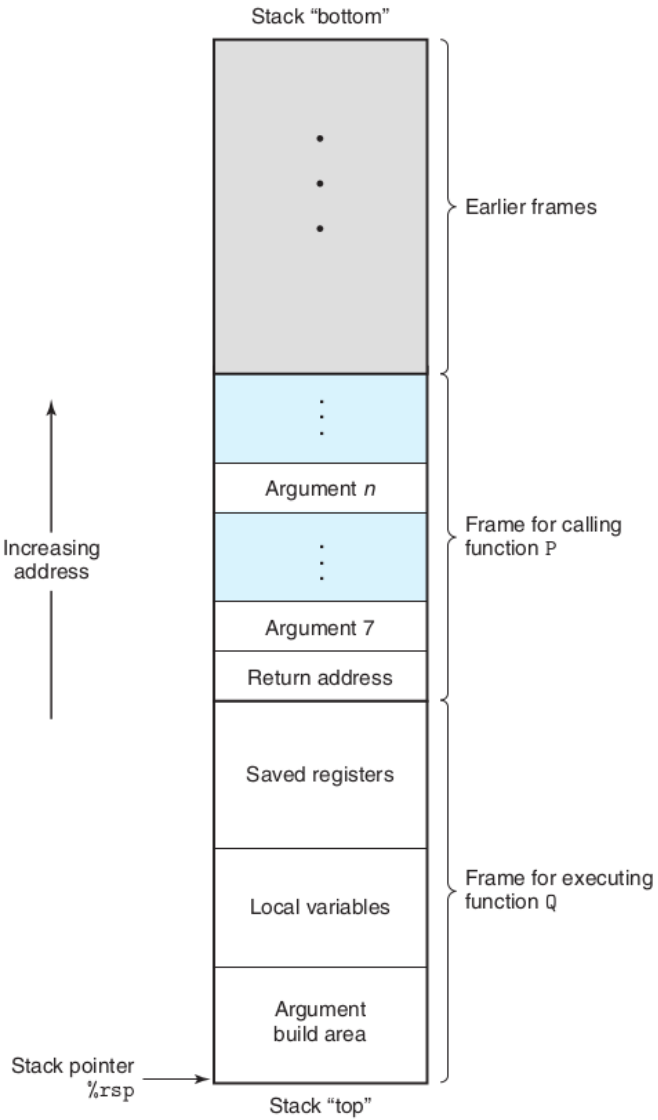
Overview

- 在高级语言中，一个函数调用另一个函数时，底层的汇编需要做什么？
  - passing control, 确保过得去，回得来
  - passing data between caller and callee，包括参数和返回值
  - managing local data，存储局部变量，保证执行完子函数后局部变量还好好的.
- 函数的调用过程本身就是 FILO 的过程.
- 需要 run-time stack 来进行存储工作

Overview

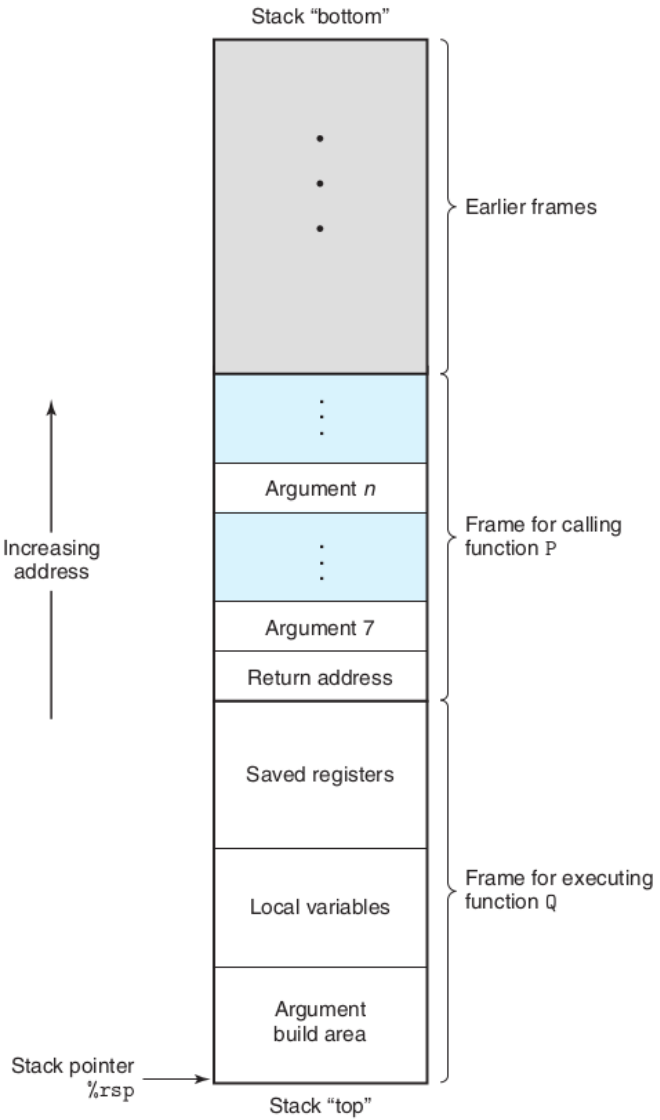

stack
%rip
registers
save some registers
to stack or other registers
passing parameters
Save some
registers to stacks
Allocate space in stack
funcA code
jumping to funcB address
funcB code
jumping back to funcA
Deallocate space; restore
registers from stack
passing return
value at %rax
Doing things

Run-time Stack



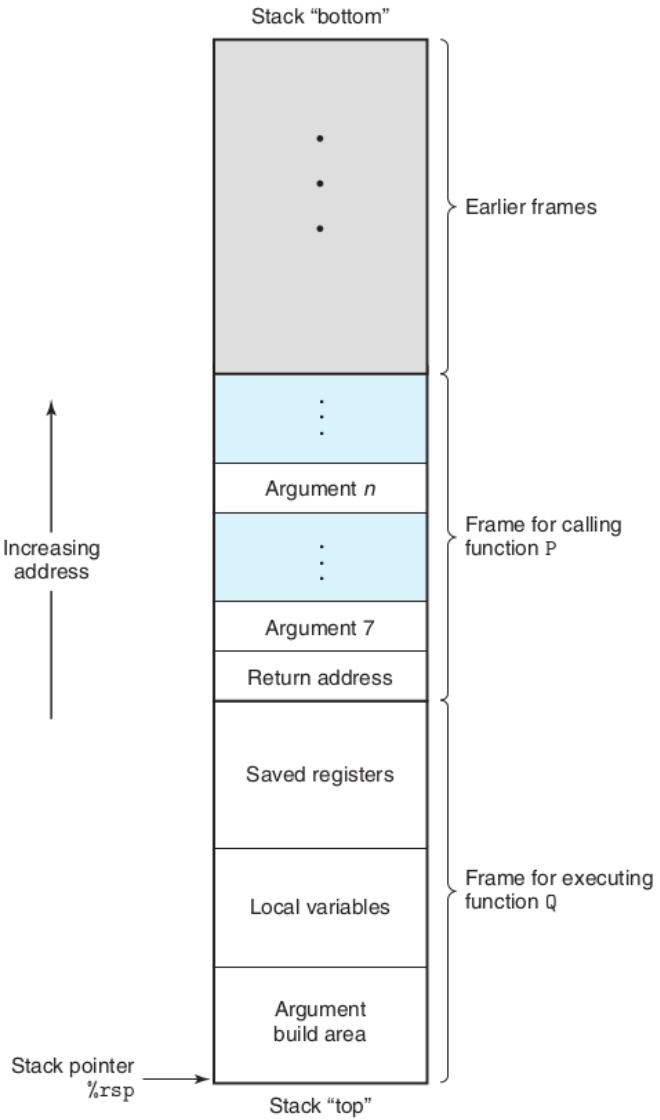
- 运行时栈的栈底在高地址，向低地址生长.

Run-time Stack



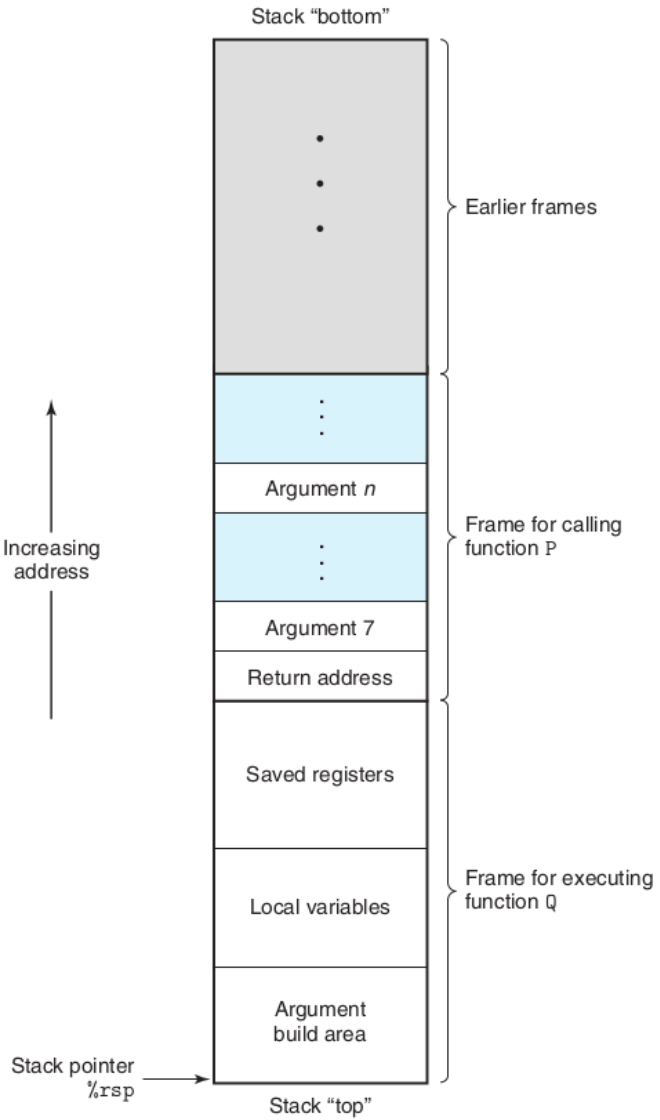
- 运行时栈的栈底在高地址，向低地址生长.
- 一个函数实例在栈中会划分出一片区域存放数据，叫做一个栈帧 (frame). `subq xxx %rsp`

Run-time Stack



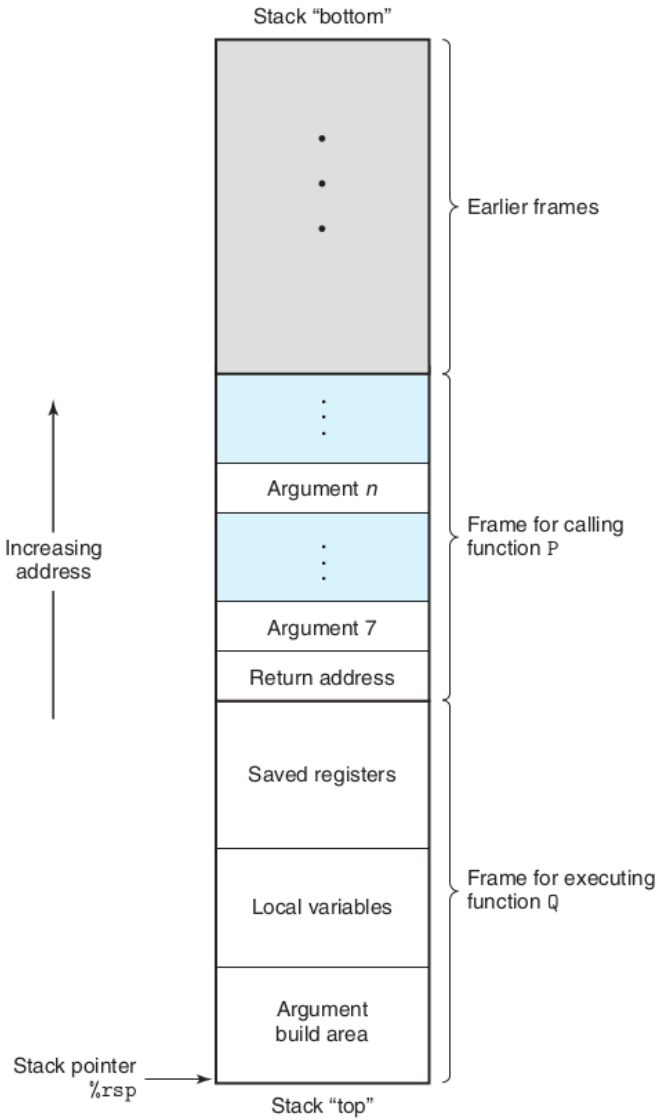
- 运行时栈的栈底在高地址，向低地址生长.
- 一个函数实例在栈中会划分出一片区域存放数据，叫做一个栈帧 (frame). `subq xxx %rsp`
- 栈顶指针是%rsp, 有时还会用%rbp 指向栈帧的底部.

Run-time Stack



- 运行时栈的栈底在高地址，向低地址生长.
- 一个函数实例在栈中会划分出一片区域存放数据，叫做一个栈帧 (frame). `subq xxx %rsp`
- 栈顶指针是%rsp, 有时还会用%rbp 指向栈帧的底部.
- 我们可以通过 `push(q) src` 和 `pop(q) dst` 来将 src 压入栈中以及弹出栈顶元素存在 dst 中.

Run-time Stack



- 运行时栈的栈底在高地址，向低地址生长.
- 一个函数实例在栈中会划分出一片区域存放数据，叫做一个栈帧 (frame). `subq xxx %rsp`
- 栈顶指针是`%rsp`, 有时还会用`%rbp` 指向栈帧的底部.
- 我们可以通过 `push(q) src` 和 `pop(q) dst` 来将 src 压入栈中以及弹出栈顶元素存在 dst 中.
- 当然，更可以像平常写入内存一样根据`%rsp` 直接写入 (比如 `mov`).

Contents

How CPU Executes Code Step by Step

- CPU 其实几乎是一个盲人，看不到整个 code。在每个时钟周期内，它只会看到 program counter `%rip`，读取对应的指令执行。

How CPU Executes Code Step by Step

- CPU 其实几乎是一个盲人，看不到整个 code。在每个时钟周期内，它只会看到 program counter `%rip`, 读取对应的指令执行。
- 在顺序结构中，CPU 每执行一次指令，`%rip` 就会跳转到下一条指令 (由硬件自动实现)，这样 CPU 就能一条一条地执行指令了.

How CPU Executes Code Step by Step

- CPU 其实几乎是一个盲人，看不到整个 code。在每个时钟周期内，它只会看到 program counter %rip，读取对应的指令执行。
- 在顺序结构中，CPU 每执行一次指令，%rip 就会跳转到下一条指令 (由硬件自动实现)，这样 CPU 就能一条一条地执行指令了.
- jmp 操作，实质上就是将%rip 设成跳转指令的地址.

When Calling a Function

- 调用子函数跟 jmp 略有不同，因为程序 return 的时候要跳转到原函数的下一条语句继续执行.

When Calling a Function

- 调用子函数跟 jmp 略有不同，因为程序 return 的时候要跳转到原函数的下一条语句继续执行.
- 所以我们需要在栈中存储返回时的地址.

When Calling a Function

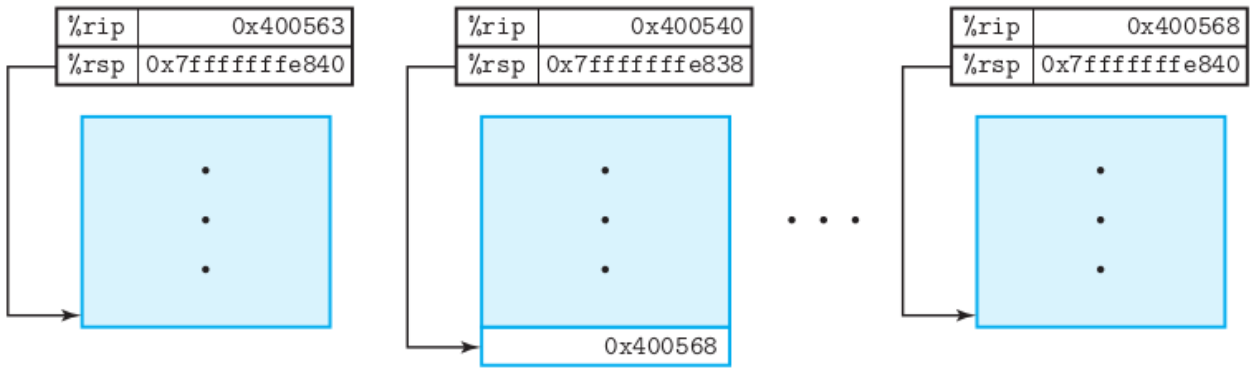
- 调用子函数跟 jmp 略有不同，因为程序 return 的时候要跳转到原函数的下一条语句继续执行.
- 所以我们需要在栈中存储返回时的地址.
- 所以 callq 分解后等价于：将返回地址压入栈中，将%rip 设置成子函数的起始命令.

When Calling a Function

- 调用子函数跟 jmp 略有不同，因为程序 return 的时候要跳转到原函数的下一条语句继续执行.
- 所以我们需要在栈中存储返回时的地址.
- 所以 callq 分解后等价于：将返回地址压入栈中，将%rip 设置成子函数的起始命令.
- ret 等价于，弹出栈顶的地址，赋给%rip.

An Example from the Textbook

```
     Beginning of function multstore
1    0000000000400540 <multstore>:
2      400540:   53                    push   %rbx
3      400541:   48 89 d3              mov    %rdx,%rbx
     . . .
     Return from function multstore
4      40054d:   c3                    retq
     . . .
     Call to multstore from main
5      400563:   e8 d8 ff ff ff        callq  400540 <multstore>
6      400568:   48 8b 54 24 08        mov    0x8(%rsp),%rdx
```



(a) Executing `call` (b) After `call` (c) After `ret`

Contents

Data Transfer: Passing Parameters and Return Value

- 我们知道，call 和 ret 只是简单的跳转到另一个指令的地址，并不含传参的功能。

Data Transfer: Passing Parameters and Return Value

- 我们知道，call 和 ret 只是简单的跳转到另一个指令的地址，并不含传参的功能。
- 所以传参和传 return value 需要将参数事先放在**指定**的寄存器中，这样跳过去的话子函数要使用参数只需要从对应的寄存器中拿数据，就正好对应起来了. return value 同理.

Some Rules

- ≤ 6 个参数时，参数依次放置在%rdi，%rsi，%rdx，%rcx，%r8，%r9

Some Rules

- ≤ 6 个参数时，参数依次放置在%rdi，%rsi，%rdx，%rcx，%r8，%r9
- 多于 6 个参数时，多余的参数依次放置在栈中

Some Rules

- ≤ 6 个参数时，参数依次放置在%rdi，%rsi，%rdx，%rcx，%r8，%r9
- 多于 6 个参数时，多余的参数依次放置在栈中
  - 参数 7 靠近栈顶，然后依次排参数 8,...,n.
  - 栈传递参数时，数据大小要向 8 的倍数对齐.

Some Rules

- ≤ 6 个参数时，参数依次放置在%rdi，%rsi，%rdx，%rcx，%r8，%r9
- 多于 6 个参数时，多余的参数依次放置在栈中
  - 参数 7 靠近栈顶，然后依次排参数 $8,\ldots,n$.
  - 栈传递参数时，数据大小要向 8 的倍数对齐.
- return value 放在%rax 中.

Some Rules

- ≤ 6 个参数时，参数依次放置在%rdi，%rsi，%rdx，%rcx，%r8，%r9
- 多于 6 个参数时，多余的参数依次放置在栈中
  - 参数 7 靠近栈顶，然后依次排参数 8,...,n.
  - 栈传递参数时，数据大小要向 8 的倍数对齐.
- return value 放在%rax 中.
- See here for more details.
- 其中 6 个参数放在寄存器中是因为访问起来比内存更快.
  - 历史上，一些 32 位系统只有 8 个 registers，所有参数都放在栈中.

An Example

```
.cfi_startproc
movq    16(%rsp), %rax # %rax = a8
movl    8(%rsp), %r10d # %r10d = a7
addw    %r10w, (%rax) # *a8 += a7
addl    %r8d, (%r9) # *a6 += a5
addl    %edx, (%rcx) # *a4 += a3
addl    (%rsi), %edi # a1 = a1 + (*a2)
movl    %edi, (%rsi) # *a2 = a1
movl    (%rcx), %eax # eax = *a4
addl    (%r9), %eax # eax += *a6
addl    %edi, %eax # eax += a2
ret
.cfi_endproc
```

```
7  int proc(int a1, int* a2,
8       int a3, int* a4,
9       int a5, int* a6,
10      short a7, short* a8){
11      *a8 += a7;
12      *a6 += a5;
13      *a4 += a3;
14      *a2 += a1;
15      return (*a6) + (*a4) + (*a2);
16 }
17
18
19
```

An Example

```
.cfi_startproc
movq    16(%rsp), %rax # %rax = a8
movl    8(%rsp), %r10d # %r10d = a7
addw    %r10w, (%rax) # *a8 += a7
addl    %r8d, (%r9) # *a6 += a5
addl    %edx, (%rcx) # *a4 += a3
addl    (%rsi), %edi # a1 = a1 + (*a2)
movl    %edi, (%rsi) # *a2 = a1
movl    (%rcx), %eax # eax = *a4
addl    (%r9), %eax # eax += *a6
addl    %edi, %eax # eax += a2
ret
.cfi_endproc
```

```
7  int proc(int a1, int* a2,
8       int a3, int* a4,
9       int a5, int* a6,
10      short a7, short* a8){
11      *a8 += a7;
12      *a6 += a5;
13      *a4 += a3;
14      *a2 += a1;
15      return (*a6) + (*a4) + (*a2);
16 }
17
18
19
```

- 注意 a_7 和 a_8 在栈中所处的位置。
- %rsp 存放的应该是返回地址，8(%rsp) 存放 a_7，16(%rsp) 存放 a_8.

Contents

Remaining Problems

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

- 一个函数很长，用到很多局部变量 -> 存到栈中

Remaining Problems

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

- 一个函数很长，用到很多局部变量 -> 存到栈中
- 需要使用一个局部变量的地址，那么这个新的局部变量就应该存在栈中而非寄存器中

Remaining Problems

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

- 一个函数很长，用到很多局部变量 -> 存到栈中
- 需要使用一个局部变量的地址，那么这个新的局部变量就应该存在栈中而非寄存器中
- Most importantly, 16 个寄存器并不是一个函数独享的；当调用下一级函数时，一些局部变量要藏好，一些局部变量需要下一级函数保持不动.

Caller or Callee saved

- Caller saved 就是说，在调用函数之前，caller 要先将需要的局部变量存好，callee 可以任意修改

Caller or Callee saved

- Caller saved 就是说，在调用函数之前，caller 要先将需要的局部变量存好，callee 可以任意修改
- Callee saved 就是说，callee 需要保证函数 return 时的这些寄存器的值与刚开始完全一样。一般来说，callee 需要哪些 callee saved 寄存器，就会在一开始将这些寄存器 push 到栈中 (**不用全部存**)。同时 caller 也不必担心这些变量会被修改.

Register Saving Conventions

registers	caller or callee saved	special uses
%rdi, %rsi, %rdx, %rcx, %r8, %r9	caller saved	传参用
%rax	caller saved	传递返回值
%rbx, %rbp, %r12 ~ %r15	callee saved	
%r10, %r11	caller saved	
%rsp	算作 callee saved	栈顶指针

An Example

```c
int factorial(int x){
    if (!x){
        return 1;
    }
    return x * factorial(x - 1);
}
```

```
    .cfi_startproc
    testl   %edi, %edi # if (!x)
    jne .L8  # not equal, jump to .L8
    movl    $1, %eax # %eax = 1
    ret
.L8:
    pushq   %rbx #Later we use %rbx, so we save %rbx first
    .cfi_def_cfa_offset 16
    .cfi_offset 3, -16
    movl    %edi, %ebx # %ebx = x;before calling,save first
    leal    -1(%rdi), %edi # x -= 1
    call    _Z9factoriali # call factorial(x - 1)
    imull   %ebx, %eax # %eax *= %ebx (%ebx = x)
    popq    %rbx # pop element from top and save to %rbx
    .cfi_def_cfa_offset 8
    ret
    .cfi_endproc
```

Another Example

```c
int func(int x, short y){
    int* p = &x;
    return proc(x, p, x, p, x, p, y) + y;
}
```

```
24      pushq   %rbx        # save %rbx
25      .cfi_def_cfa_offset 16
26      .cfi_offset 3, -16
27      subq    $8, %rsp    # %rsp -= 8
28      .cfi_def_cfa_offset 24
29      movl    %edi, 4(%rsp) # 4(%rsp) = x
30      movswl  %si, %ebx     # %ebx = y
31      leaq    4(%rsp), %rsi # %rsi = p = &x
32      pushq   %rbx          # the 7th param
33      .cfi_def_cfa_offset 32
34      movq    %rsi, %r9     # the 6th param: p
35      movl    %edi, %r8d    # the 5th param: x
36      movq    %rsi, %rcx
37      movl    %edi, %edx
38      call    _Z4prociPiiS_iS_s
39      addl    %ebx, %eax    # ans += %ebx (y)
40      addq    $16, %rsp
41      .cfi_def_cfa_offset 16
42      popq    %rbx          # restore %rbx
43      .cfi_def_cfa_offset 8
44      ret
```