程序的机器级表示

数据格式 & 访问信息

数据格式

下面我们给出 `x86-64` 指令集中关于数据大小的指令

C声明	Intel数据类型	汇编代码后缀	大小（字节）
`char`	字节	`b`	1
`short`	字	`w`	2
`int`	双字	`l`	4

C声明	Intel数据类型	汇编代码后缀	大小（字节）
long	四字	q	8
char*	四字	q	8
float	单精度	s	4
double	双精度	l	8

这当中，字 word 表示 2 个字节；双字 double word\long word 表示 4 个字节；四字 quad word 表示 8 个字节

需要说明的是，汇编代码用 l 表示双字与双精度，这并不会造成歧义，因为整数与浮点数使用的是不同的两套指令和寄存器

访问信息

一个 x86-64 的 CPU 包含一组 16 **个存储** 64 **位的*通用目的寄存器***，以下为这 16 个寄存器的名称及其作用

表中从左往右依次为：64 位、32 位、16 位、8 位

Register	Function	Register	Function
%rax %eax %ax %al	保存返回值	%rbx %ebx %bx %bl	被调用者保存
%rcx %ecx %cx %cl	第 4 个参数	%rdx %edx %dx %dl	第 3 个参数
%rsi %esi %si %sil	第 2 个参数	%rdi %edi %di %dil	第 1 个参数
%rbp %ebp %bp %bpl	被调用者保存	%rsp %esp %sp %spl	栈指针
%r8 %r8d %r8w %r8b	第 5 个参数	%r9 %r9d %r9w %r9b	第 6 个参数
%r10 %r10d %r10w %r10b	调用者保存	%r11 %r11d %r11w %r11b	调用者保存
%r12 %r12d %r12w %r12b	被调用者保存	%r13 %r13d %r13w %r13b	被调用者保存
%r14 %r14d %r14w %r14b	被调用者保存	%r15 %r15d %r15w %r15b	被调用者保存

不同的指令可以对单个寄存器中不同大小的数据进行操作：

- 字节级操作可以访问最低的字节
- 16 位操作可以访问最低的 2 个字节
- 32 位操作可以访问最低的 4 个字节
- 64 位操作可以访问整个寄存器

如果某条指令以这些寄存器为目标，对于生成小于 8 字节结果的指令，寄存器中剩余字节由以下两条规则指定：

- 生成 1 字节或 2 字节结果的指令会保持剩下的字节不变
- 生成 4 字节结果的指令，会使**高位 4 字节全部清零**

操作数指示符

大多数指令有一个或多个操作数 `operand` ，用以表示该操作中**要使用的源数据值**和**放置目标结果的位置**

源数据值可以是**常数**给出或者从**寄存器**或**内存**当中读出，其结果可以存放在**寄存器**或**内存**

因此，操作数可以被分为三种类型：

- 立即数 `immediate` ，用以表示常数值，通常用 `$` 后面跟一个 C 标准的整数
- 寄存器 `register` ，用以表示寄存器的**内容**，我们用 r_a 来表示任意寄存器 a ，而引用 $R[r_a]$ 则可以表示该寄存器的值
- 内存引用 `memory` ，它会根据**计算出来的地址来访问某个内存位置**，我们用 $M_b[Addr]$ 表示从地址 $Addr$ 开始的连续 b 个字节的引用（通常我们省去下标 b）

关于在内存当中的寻址模式，在这里我们给出一种最通用的：

$$Imm(r_b,r_i,s)=M[Imm+R[r_b]+s\cdot R[r_i]]$$

在这里，Imm 为**立即数偏移**，r_b 为**基址寄存器**，r_i 为**变址寄存器**，s 为比例因子，必须是 $1,2,4,8$

需要注意的是，当采用**基址**与**变址**寄存器来进行内存寻址时，二者**都必须要是 64 位寄存器**

指令 & 算术和逻辑操作 & 控制

数据传送指令

MOV 类指令可以将数据从源位置**复制**目的位置，此过程中不做任何改变。MOV 类由四条基本指令构成，它们的区别仅仅在于操作数据的大小不同，分别是 $1,2,4,8$ 字节

Command	Effect	Description
$MOV\quad S,D$	$S\rightarrow D$	传送
`movb\movw\movl\movq`		分别传送字、字节、二字、四字（从补码角度看待）
`movabsq`		传送四字（从无符号的角度看待）

> 下面用 `movx` 来表示 `movb\movw\movl\movq`

源操作数要么是立即数、寄存器或者内存引用，目的操作数要么是寄存器或者内存引用,这里有一条限制是：**两个操作数不能都是内存引用**

这些指令的操作数可以是 16 个寄存器当中有标号部分的任意一个，但需要注意的是，**寄存器的大小必须与指令所指示的大小相一致**

无论操作数当中有几个寄存器（一个或两个），它们的**大小都需要与指令所指示的相一致**

例如：`movl %eax %rdx` 这是错误的，因为源寄存器为 32 位，目的寄存器为 64 位，二者并不相同

需要额外说明的是，不能以两个内存引用作为操作数的原因在于，内存引用是**无法确定大小的**，但寄存器或立即数却可以确定大小。因此对于内存到内存的复制，需要先从内存到寄存器，再从寄存器到内存

MOV 指令只会更新**目的操作数**指定的寄存器字节或内存，其他位置不会发生改变，唯一的例外是使用 `movl` 且目的地为寄存器时，这会使得高位的 4 字节全部清零

常规的 `movx` 指令只能以 32 位**补码**立即数作为源操作数，然后将该数字**符号拓展**得到 64 位的值（也就是前面写的从补码的角度考虑），而 `movabsq` 则可以以任意 64 位立即数作为源操作数，并且**只能以寄存器作为目的地**

上面的指令如果是以 $1,2$ 为字节单位的指令，则不会对高位进行填充，例如：

```
movabsq $0x0011223344556677, %rax   %rax = 0011223344556677
movb    $-1, %al                    %rax = 00112233445566FF
movw    $-1, %ax                    %rax = 001122334455FFFF
movl    $-1, %eax                   %rax = 00000000FFFFFFFF
movq    $-1, %rax                   %rax = FFFFFFFFFFFFFFFF
```

可以看到，以 `b` 和 `w` 为大小指定的指令不会对高位进行修改，下面两个 MOV 的拓展类可以实现**对剩余字节的填充**

- $MOVZ$ 通过**零拓展**进行填充，将剩余字节填充为 0
- $MOVS$ 通过**符号拓展**进行填充，将剩余字节填充为**源操作数的最高位**

Command	Effect	Description
$MOVZ \quad S,R$	零拓展$(S) \rightarrow R$	以零拓展进行传送

`movzbw\movzbl\movzwl\movzbq\movzwq`

指令	效果	描述
$MOVS \quad S,R$	符号拓展$(S) \rightarrow R$	以符号拓展进行传送
`movsbw\movsbl\movswl\movsbq\movswq\movslq`		
`cltq`	符号拓展(	将 `%eax` 符号拓展后的结果放到 `%rax` 中

$MOVZ$ 与 $MOVS$ 均以**寄存器或内存作为源**，以**寄存器作为目的地**

正如上面指令所指示的那样，最后两个字符用以表示大小，第一个指示源操作数的大小，第二个指示目的操作数的大小

需要说明的是，这里面没有 `movzlq` ，这是因为对于以 4 字节为操作数的 MOV 指令，会将高位全部置零，因此这条指令没有必要出现

`ctlq` 指令**没有操作数**，它以 `%eax` 作为源，将其**符号拓展**后的结果放到 `%rax` 中，本质上与 `movslq %eax, %rax` 完全一致

数据传送之于指针

C 语言代码：

```
long wxchange(long* xp, long y)
{
    long x = *xp;
    *xp = y;
    return x;
}
```

对应汇编代码为：

```
xp in %rdi, y in %rsi
movq    (%rdi), %rax
movq    %rsi, (%rdi)
ret
```

注意到两点：

- 局部变量均**保存在寄存器中**，而不是在内存当中。这是显而易见的，因为访问寄存器比访问内存要快
- 由于 xp 传入的是指针，因此**寄存器当中保存的是一个内存引用**，而不是直接将该指针所指对象的值直接放到寄存器中（xp 是从内存当中读取出来的）

数据传送之于转型

有如下代码

```
//声明
src_t *sp;
dest_t *dp;
//sp in %rdi, dp in %rsi

//转型
*dp = (dest_t)*sp;
```

其中 src_t 和 dest_t 均为 typedef 声明的不同数据类型，我们考虑的是不同数据大小之间的转型是如何进行的

把握两个原则：

- 从小到大需要拓展
  - 有符号数需要做**符号拓展**，使用 $MOVS$ 类指令
  - 无符号数需要做**零拓展**，使用 $MOVZ$ 类指令
- 从大到小直接截断

具体如下表：

src_t	dest_t	Command

src_t	dest_t	Command
long	long	movq (%rai), %rax; movq %rax, (%rsi)
char	int	movsbl; movl
char	unsigned	movzbl; movl
unsigned char	int	movzbl; movl
int	char	movl; movb
unsigned	unsigned char	movl; movb

压栈与出栈

在 x86-64 中，栈是**向下增长**的，因此**栈顶元素的地址是栈中所有元素当中最低的**

压栈与出栈主要涉及两条指令：

Command	Effect	Description
$pushq \quad S$	$(R[$	将 quad word 压入栈
$popq \quad D$	$M[R[$	将 quad word 弹出栈

> pushq %rbp 等价于：
>
> ```
> subq $8, %rsp
> movq %rbp, (%rsp)
> ```
>
> popq %rax 等价于：
>
> ```
> movq (%rsp), %rax
> addq $8, %rsp
> ```

注意到：

- push 指令是先将**栈指针减小**，然后再将**寄存器**的值压入栈中，且减少的数值是固定的
- pop 指令会先将**内存**当中的值传递到**寄存器**中，然后再**增加栈指针**

算术和逻辑操作

基本操作

下面我们直接给出 x86-64 的一些整数与逻辑操作。这里面当中，除了 leaq 以外，其余的都代表一个指令类，用以对**不同大小的数据进行操作**。例如，ADD 类由 addb; addw; addl; addq 四种组成

Command	Effect	Description
$leaq \quad S, D$	D\ \gets\ &S	加载有效地址
$INC \quad D$	$D \leftarrow D+1$	加 1
$DEC \quad D$	$D \leftarrow D-1$	减 1
$NEG \quad D$	$D \leftarrow -D$	取负
$NOR \quad D$	$D \leftarrow \sim D$	取补
$ADD \quad S, D$	$D \leftarrow D+S$	加
$SUB \quad S, D$	$D \leftarrow D-S$	减
$IMUL \quad S, D$	$D \leftarrow D \times S$	乘
$XOR \quad S, D$	$D \leftarrow D \oplus S$	异或
$OR \quad S, D$	$D \leftarrow D \mid S$	或
$AND \quad S, D$	D\ \gets\ D\ &\ S	与
$SAL \quad k, D$	$D \leftarrow D << k$	左移
$SHL \quad k, D$	$D \leftarrow D << k$	左移，等同于 SAL
$SAR \quad k, D$	$D \leftarrow D >>_A k$	算术右移
$SHR \quad k, D$	$D \leftarrow D >>_L k$	逻辑右移

加载有效地址 `load effective address` 指令 `leaq`，可以将数据**从内存读到寄存器中**，需要注意的是，这里的读取内存并不是将那个值读取出来，而是将地址写入到寄存器中，因此用 `&` 来描述更为合适

其次 `leaq` 可以简洁地描述一些算术操作。例如，如果 `%rdx` 的值为 x，那么 `leaq 7(%rdx, %rdx, 4), %rax` 会将 `%rax` 的值设置成 $5x+7$（这里需要说明的是，x 的值并没有指定，既可以是地址，也可以是数值）

`leaq` 的目的操作数必须是**寄存器**

移位操作中，需要先给出移位量，再给出需要移位的数。移位量可以是**立即数**，也可以是**字节寄存器 `%cl` 当中的数值**

需要说明的是，在 `x86-64` 中，对 ω 位的数据进行移位操作时，移位量为 `%cl` 的低 m 个字节，这里 $2^m = \omega$

举例来说就是，如果 `%cl` 的值被置为 `0xFF`，那么有：

- 若 $\omega = 8$（对应 `salb`），则 `%cl` 中由低 3 位来决定位移量，这里为 7
- 若 $\omega = 16$（对应 `salw`），则 `%cl` 中由低 4 位来决定位移量，这里位 15
- 若 $\omega = 32$（对应 `sall`），则 `%cl` 中由低 5 位来决定位移量，这里位 31
- 若 $\omega = 64$（对应 `salq`），则 `%cl` 中由低 6 位来决定位移量，这里位 63

特殊操作

正如前面所提到的那样，两个 64 位的无符号数或有符号数的结构需要 128 位来进行存储

x86-64 指令集对 128 位（16 字节）同样提供支持，延续之前的命名方式，16 字节被称为**八字**，以下描述的是支持两个 64 位数字的全 128 位乘法或除法的指令：

Command	Effect	Description
$imulq \quad S \backslash \text{mulq} \quad S$	$S \times R[$	有符号全乘法\无符号全乘法
$cqto$	符号拓展$(R[$	转换为八字
$idivq \quad S \backslash \text{divq} \quad S$	$R[$	有符号除法\无符号除法

前面给出过一种双操作数的 $IMUL$ 类指令，用于实现两个 64 位数的乘积，其结果还是 64 位数，本质上是实现 $_{64}^{u}与_{64}^{t}$

对于乘法，x86-64 提供一种单操作数的指令用于计算无符号数和有符号数的全 128 位乘积。两条指令均要求一个数由**寄存器** %rax 给出，另一个由**操作数的形式给出。**而最终的结果，低 64 位存储于 $rax 中，高 64 位存储于 %rdx 中

对于 128 位的全除法则略有不同，两条指令以**寄存器** %rdx （低 64 位）和 %rax （高 64 位）中的 128 位数作为**被除数**，以**操作数的形式给出除数。**最终将商存储在 %rax ，余数存储在 %rdx （商存在低 64 位，余数存储在高 64 位）

除法的 $divq$ 指令支持 128 位的被除数，但很多时候被除数一般为 64 位，存放于 %rax 中，此时 %rdx 应当被设为**全零**（无符号除法）或者 %rax **的符号位**（有符号除法）。而该操作可以用 $cqto$ 完成

该指令不需要操作数，它可以隐式读出 %rax 的符号位并将其复制到 %rdx 中

控制

当遇到条件语句、循环语句和分支语句时，就需要有条件地执行指令了，也就是改变控制流或者数据流，而实现控制流改变的方式有两种：「条件控制」和「条件传送」，下面我们逐一介绍

条件控制

CPU 除了维护整数寄存器以外，还会维护一组单个位的条件码 condition code ，这些条件码用于描述**最近**的算术或逻辑操作的属性。我们通过检测这些条件码的方式来实现控制流的改变这个过程，称为**条件控制**

常用的条件码有：

- CF：进位标志——最近的操作使最高位产生了进位，用于检查无符号操作的溢出
- ZF：零标志——最近的操作得出的结果为零
- SF：符号标志——最近的操作得到的结果为负数
- OF：溢出标志——最近的操作导致补码的溢出，正溢出或负溢出

上面列出的指令中，除了 lea 不会设置标志位外，其余的都会设置标志位并且会改变寄存器的值，下面介绍两种**只设置标志位而不改变寄存器数值**的指令

Command	Base On	Description

Command	Base On	Description
$CMP \quad S_1, S_2$	$S_2 - S_1$	比较
cmpb\cmpw\cmpl\cmpq		比较字节\字\双字\四字
$TEST \quad S_1, S_2$	$S_2 \& S_1$	测试
testb\testw\testl\testq		测试字节\字\双字\四字

$TEST$ 指令的**两个操作数是一样的**，用于判断是小于零，等于零，还是大于零

关于条件码设置的具体细节（设置了哪些条件码），我们可以不用关注，因为这不重要，重要的是基于条件码所产生的条件控制才是我们应当关注的重点

访问条件码

条件码不会直接读取，常用的使用方式有三种：

- 将条件码的组合设置某个字节为 0 或者 1
- 根据条件码跳转到程序的某个其他部分
- 根据条件码来传送数据

下面我们分别介绍这三者的使用

字节设置

我们可以用一整组 SET 指令来对某个字节设置成 0 或者 1 ，指令当中的后缀指的是**不同的条件**，而不是像之前那样表示数据的大小

SET 指令的目的操作数是**低位单字节寄存器**或者**一个字节的内存地址**，而对于寄存器而言，为了得到一个 32 位或 64 位的结果，我们必须对高位进行**清零**

具体指令如下：

Command	Synonyms	Effect	Condition
$sete\ D$	$setz$	$D \leftarrow ZF$	相等、零
$setne\ D$	$setnz$	$D \leftarrow\ ZF$	不等、非零
$sets\ D$		$D \leftarrow SF$	负数
$setns\ D$		$D \leftarrow\ SF$	非负数
$setg\ D$	$setnle$	$D \leftarrow\sim (SF \oplus OF) \& ZF$	大于（有符号 > ）
$setge\ D$	$setnl$	$D \leftarrow\sim (SF \oplus OF)$	大于等于（有符号 >= ）
$setl\ D$	$setnge$	$D \leftarrow SF \oplus OF$	小于（有符号 < ）
$setle\ D$	$setng$	$D \leftarrow (SF \oplus OF) \vee ZF$	小于等于（有符号 <= ）
$seta\ D$	$setnbe$	$D \leftarrow\sim CF \& ZF$	超过（无符号 > ）
$setae\ D$	$setnb$	$D \leftarrow\sim CF$	超过或相等（无符号 >=）

Command	Synonyms	Effect	Condition
$setb\ D$	$setnae$	$D \leftarrow CF$	低于（无符号 < ）
$setbe\ D$	$setna$	$D \leftarrow CF \vee ZF$	低于或相等（无符号 <= ）

在实际的汇编代码中，我们并不需要关注这些是如何推导的，我们只需要知道**根据什么条件**设置了**哪个位置的值即可**

举个例子：

```
int comp(data_t a, data_t b)
a in %rdi, b in %rsi

comp:
  cmpq    %rsi, %rdi    compare a : b
  setl    %al           set low-order byte of %eax
  movzbl  %al, %eax     clear rest of %eax
  ret
```

注意到，汇编代码中只是用 `setl` 指令根据 a 与 b 的值设置了 `%al` 的值，我们并**不需要关注标志位的情况**

条件跳转

跳转指令会导致程序的执行切换到一个全新的位置，跳转的目的地通常用一个标号 `label` 指明，比如：

```
  jmp .L1
  movq
.L1
  pushq
```

在产生目标代码文件时，编译器会确定所有**带标号的指令的地址**，也就是将跳转目标作为跳转指令编码的一部分进行考虑

下面给出不同的跳转指令，总共分为**无条件跳转**和**条件跳转**

Command	Synonyms	Condition	Description
$jmp\ Lable$		无	直接跳转（直接跳转到以寄存器的值为地址的地方）
$jmp\ *Operand$		无	间接跳转（以寄存器的值作为地址再次跳转）
$je\ Lable$	jz	ZF	相等、零
$jne\ Lable$	jnz	$\sim ZF$	不相等、非零
$js\ Lable$		SF	负数

Command	Synonyms	Condition	Description
$jns\ Lable$		$\sim SF$	非负数
$jg\ Lable$	$jnle$	$\sim (SF \oplus OF) \& ZF$	有符号大于
$jge\ Lable$	jnl	$\sim (SF \oplus OF)$	有符号大于等于
$jl\ Lable$	$jnge$	$SF \oplus OF$	有符号小于
$jle\ Lable$	jne	$(SF \oplus OF) \vee ZF$	有符号小于等于
$ja\ Lable$	$jnbe$	$\sim CF \& ZF$	无符号超过
$jae\ Lable$	jnb	$\sim CF$	无符号超过或等于
$jb\ Lable$	$jnae$	CF	无符号低于
$jbe\ Lable$	jna	$CF \vee ZF$	无符号低于或等于

jmp 为无条件跳转，分为**直接跳转**和**间接跳转**两种

直接跳转是以标号作为跳转目标，就如上面所演示的那样

间接跳转以寄存器或内存当中的某个值作为跳转目标，通常写法带有 `*`

例如：

`jmp *%rax` 以寄存器 `%rax` 的值作为跳转目标；`jmp *(%rax)` 以寄存器 `%rax` 的值作为地址，以该地址所对应的值为跳转目标

需要说明的是，只有无条件跳转才可以**间接跳转**，有条件跳转不能间接跳转

关于跳转指令的编码，规则为：以目标指令地址与跳转指令后面那条指令的地址之间的差值作为编码

基于条件跳转实现条件分支

对于 C 语言当中的 `if-else` 语句，其通用形式如下：

```
if(test-expr)
    then-statement
else
    else-statement
```

这里的 *test-expr* 是一个整数表达式，其值只有零（解释为假）或非零（解释为真），下面的两条分支语句 *then-statement* 和 *else-statement* **只会执行一个**

对于这种形式，汇编通常会产生如下代码，我们用 `C` 语言来描述其程序控制流

```
t = test-expr
if(!t)
    goto false;
```

```
    then-statement
    goto done;
false:
    else-statement
done:
```

也就是说，汇编器会为 *then-statement* 和 *else-statement* **生成各自的代码块，但只会执行一个**

举个例子，考虑如下代码：

```
int test(int x, int y)
{
    int result = 0;
    if (x > y)
        result = x - y;
    else
        result = y - x;
    return result;
}
```

我们将其汇编代码翻译为等效的 C 语言代码，如下：

```
int test_goto(int x, int y)
{
    int result = 0;
    bool t = x > y;
    if (!t)
        goto x_g_y;
    result = y - x;
    return result;
x_g_y:
    result = x - y;
    return result;
}
```

条件传送

我们前面介绍过 MOV 类指令，那是**无条件传送指令**，我们在此给出条件传送的版本

Command	Synonyms	Condition	Description
$cmove \quad S, R$	$cmovz$	ZF	相等、零
$cmovne \quad S, R$	$cmovnz$	$\sim ZF$	不相等、非零
$cmovs \quad S, R$		$\sim SF$	负数
$cmovns \quad S, R$		$\sim SF$	非负数

Command	Synonyms	Condition	Description
$cmovg \quad S,R$	$cmovnle$	$\sim (SF \oplus OF) \& ZF$	有符号大于
$cmovge \quad S,R$	$cmovnl$	$\sim (SF \oplus OF)$	有符号大于等于
$cmovl \quad S,R$	$cmovnge$	$SF \oplus OF$	有符号小于
$cmovle$	$cmovng$	$(SF \oplus OF) \vee ZF$	有符号小于等于
$cmova$	$cmovnbe$	$\sim CF \& ZF$	无符号超过
$cmovae$	$cmovnb$	$\sim CF$	无符号超过或等于
$cmovb$	$cmovnae$	CF	无符号低于
$cmovbe$	$cmovna$	$CF \vee ZF$	无符号低于或等于

条件传送的源操作数为**寄存器**或**内存地址**，目的操作数为**寄存器**。条件传送的源和目的地可以是 16 位，32 位，64 位，但**不支持单字节传送**（而无条件传送指令则会显式指定大小）

基于条件传送实现条件分支

需要事先说明的是，基于条件传送的实现应用场景**十分有限**

处理器通过使用流水线 `pipelining` 设计来获取高性能，这要求能够**事先确定要执行的指令序列**，这样能够保证流水线中充满了指令

在遇到分支时，处理器会猜测一条路线，将该路线的指令全部加载到流水线中。显然如果猜测失败则会相应的时间惩罚，这便是条件控制实现的问题

> **如何确定分支预测错误的惩罚**
>
> 设预测错误的概率为 p ，如果没有预测错误所需时间为 T_{ok} ，预测错误的处罚为 T_{mp}
>
> 因此，代码的平均执行时间为：$T_{avg}(p) = (1-p)T_{ok} + p(T_{ok} + T_{mp}) = T_{ok} + pT_{mp}$
>
> 如果已知 T_{ok} （预测可靠）和 T_{ran} （预测随机，$p = 0.5$），我们要确定 T_{mp}
>
> 将其带入得：$T_{ran} = T_{ok} + 0.5T_{mp}$ ，所以有 $T_{mp} = 2(T_{ran} - T_{ok})$

基于条件传送是指，处理器会对两条支路均进行计算，最后通过判断来确定该返回哪个值（此汇编代码由编译器生成）

还是上面的例子，基于条件传送生成的汇编代码对应的 `C` 代码如下：

```
int test_goto(int x, int y)
{
    int rval = x - y;
    int eval = y - x;
    int test = x > y;
    if (!test)//当条件不满足是返回eval
        rval = eval;
```

```
    return rval;
}
```

更一般地，我们考虑下面条件和赋值表达式的通用形式

```
v = test-expr ? then-statement : else-statement
```

基于条件控制的汇编器会得到如下代码：

```
t = test-expr;
if(!t)
    goto false;
    v = then-statement;
    goto done;
false:
    v = else-statement;
done:
```

而基于条件传送的汇编器会得到如下代码：

```
v = then-statement;
ve = else-statement;
t = test-expr;
if(!t)
    v = ve;
```

在开头已经说明的是，条件传送的应用场景十分有限，如果 *then-expr* 和 *else-expr* 需要做大量的计算，那么当条件不满足时，所作出的工作也就白费了，因此哪怕冒着预测错误招致的处罚，编译器也会更倾向于生成**基于条件跳转**的条件分支指令

循环

---

switch 语句

过程

过程提供了一种用于封装代码方式，用**一组指定的参数**和**一个可选的返回值**实现某种功能，然后我们可以在函数的任何部分调用该过程

在不同的语言中，过程的表现形式各有不同：函数 `function` 、方法 `methor` 、子例程 `subroutine` 、处理函数 `handler` 等，这些均可以统称*过程*

假设过程 P 调用过程 Q，Q 执行完之后返回 P，为了实现过程的机器级支持，需要提供以下机制：

- 传递控制。当进入过程 Q 时，程序计数器需要设置为过程 Q 的起始地址；在返回时，程序计数器需要设置为过程 P 中调用过程 Q 后面那条语句的地址
- 传递数据。P 需要向 Q 传递一个或多个数据，Q 需要向 P 返回至多一个数据
- 分配和释放内存。在开始时，可能需要为 Q 的局部变量分配空间；在返回时，又需要释放这些空间

下面，我们按照传递控制、传递数据、内存管理的顺序分别介绍

控制转移

运行时栈

在过程 P 调用过程 Q 的例子中，我们发现当 Q 执行时，P 以及所有向上追溯到 P 的过程均处于**被挂起**的状态。另一方面，当执行 Q 时，我们需要为其分配空间用于存储局部变量，或者调用下一个过程。当 Q 返回时，我们需要将 Q 所占用的空间全部释放掉，此时回到调用 Q 的上一级过程

上述这整个调用过程，恰好可以用栈来描述。当过程 P 执行时，控制流处于 P 所在的区域，当调用 Q 时，相当于将 Q **压入** P 的上方，此时控制流处于 Q 所在的区域。因此我们将这个栈称为**运行时栈**

`x86-64` 的栈向低地址增长，因此栈顶在低地址，栈底在高地址。栈指针 `%rsp` 指向栈顶，我们可以用 `pushq` 或 `popq` 向栈顶压入或弹出元素

当执行一个过程所需的空间**超出寄存器所能存放的大小时**，就需要在栈上分配空间，此空间被称为该过程的**栈帧**（如果一个过程不需要栈帧，那么就不会分配），需要注意的是，栈帧一般都是**定长的**，在**过程开始**时通过以下指令分配好：

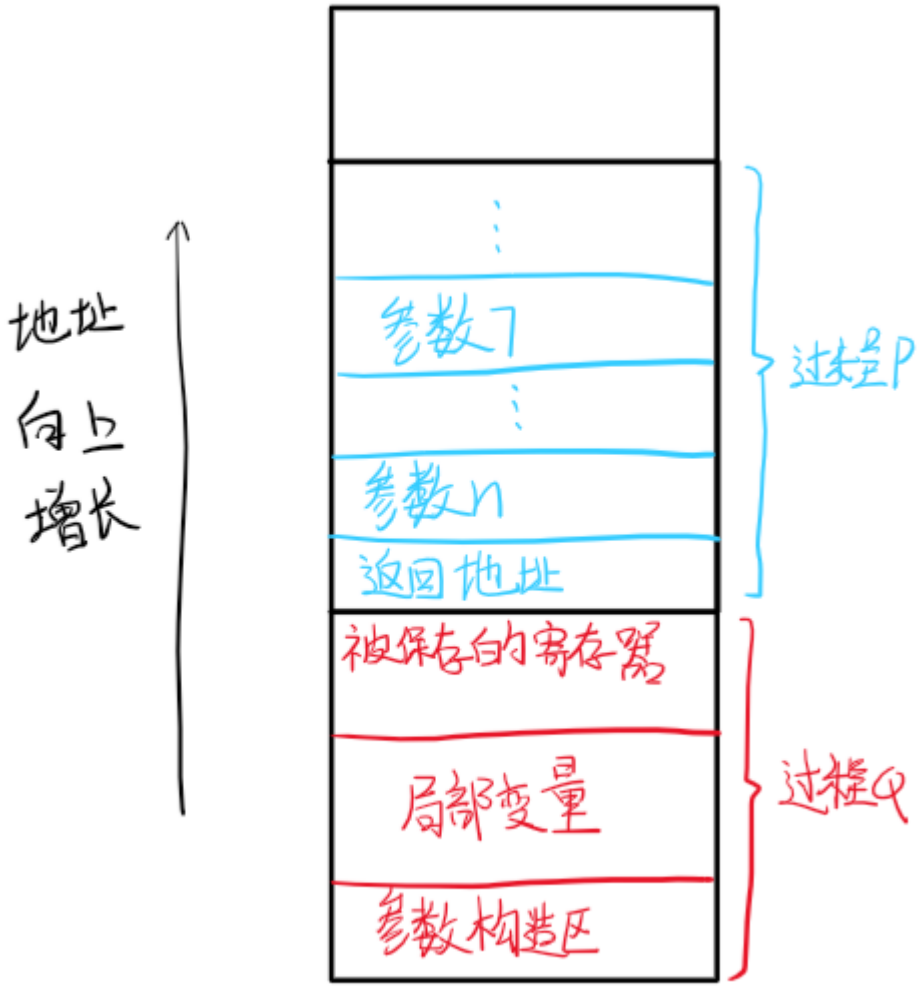
```
subq $16, %rsp   分配 16 字节的栈帧
```

而过程结束时则通过以下指令回收：

```
addq $16, %rsp   回收栈帧
```

当前正在执行的过程的栈帧总是位于栈顶，栈帧主要是用于保存该过程的相关信息。当过程 P 调用过程 Q 时，会先将返回地址**压入** P **的栈帧中**，表明当 Q 返回时需要从哪里开始继续执行代码（通常是调用语句的下一条语句的地址）。这个返回地址是过程 P 栈帧的一部分，因为它指明了过程 P 的相关属性

而对于过程 Q 而言，则会拓展当前栈的边界（向下拓展），用于分配其栈帧所需要的空间。在这个空间中，过程可以**保存寄存器的值**、**分配局部变量**、**为其他调用过程分配参数**。对于一个栈帧的通用结构，如下图所示：



也就是说，处于栈顶的栈帧总是当前正在执行的过程

转移控制

将控制从过程 P 转移到过程 Q 只需要简单地将程序计数器的值设为 Q 起始位置的地址即可。在 Q 返回的时候，处理器必须知道在 P 中继续执行指令的地址

在 x86-64 中，这个过程是通过一组指令 $call\quad Q$ 和 ret 执行的

$call\quad Q$ 会将地址 A **压入栈中**，并将程序计数器的值设为 Q 的起始地址。这里的 A 被称为**返回地址**，也就是 $call$ 指令后面的那条指令的地址

ret 会将返回地址 A 从栈中**弹出**，并将程序计数器设置为 A

下面给出两条指令的一般形式：

Command	Description
$call\quad Label$	过程调用（直接）

Command		Description
$call$	$*Operand$	过程调用（间接）
	ret	从调用过程中返回

$call$ 指令的目标，为被调用过程的起始地址。同跳转一样，可以是直接的（用已确定地址的标号），也可以是间接的（从寄存器或内存中读取）

需要说明的是：在返回时，**返回值保存在寄存器 `%rax` 中**

数据传送

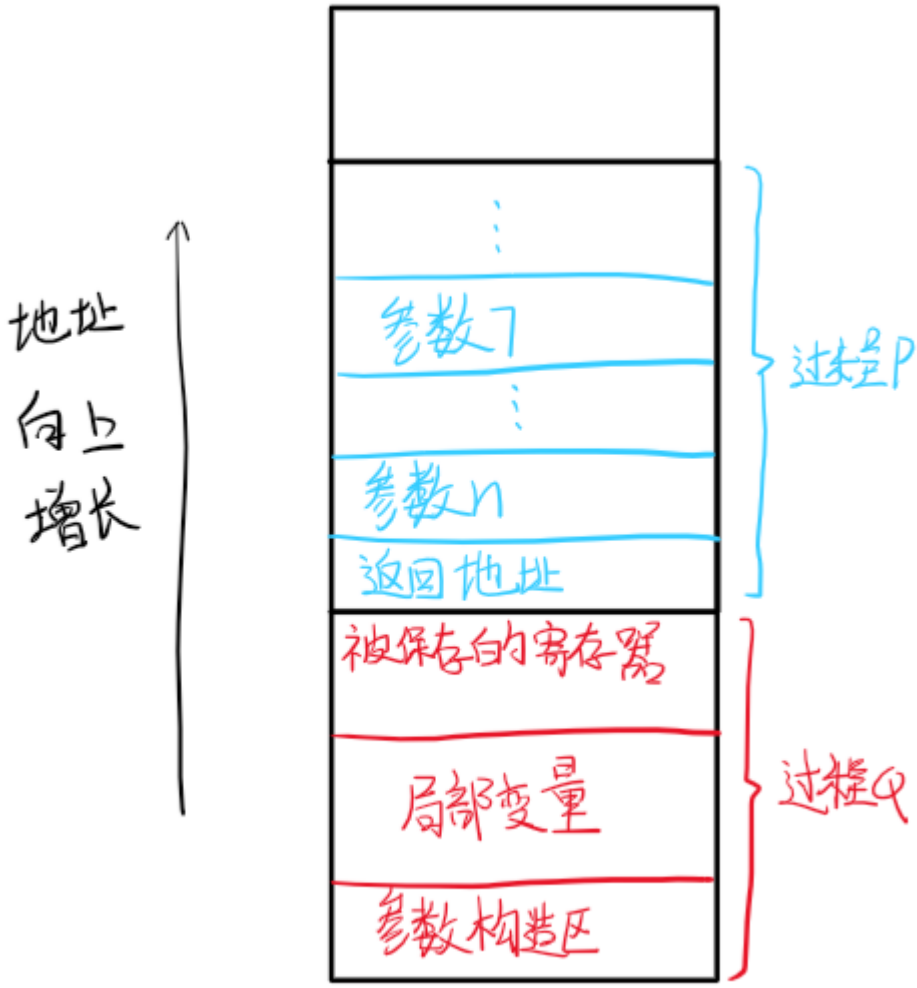
当调用一个过程时，除了要将控制传递外，还需要传递相应的参数，而该过程也可能会返回相应的参数。在 `x86-64` 中，数据的传送大多是通过寄存器实现的

在访问信息那里我们看到，16 个寄存器中有专门用于传递参数的，分别为：

Parameter Order	Register
1	%rdi
2	%rsi
3	%rdx
4	%rcx
5	%r8
6	%r9

说明：这些寄存器分别可用 64、32、16、8 位的，这里并没有列出

如果一个过程有超过 6 个参数，那么**超过的部分就需要通过栈来传递。**假设过程 P 调用过程 Q，这当中需要传递 n $(n > 6)$ 个参数，那么 P **的栈帧**必须要存储第 7 到第 n 号参数



如上图所示，P 需要将参数 $1 \sim 6$ 放入寄存器中，把参数 $7 \sim n$ 放到**自己**的栈帧中，而参数 7 位于栈顶。通过栈传递参数时，所有的数据大小都需要按 8 的倍数对齐。参数复制完毕后就可以将控制转移到 Q 了

相应地，如果 Q 也调用了另一个超过 6 个参数的过程，那么它也需要在**自己**的栈帧中为超出 6 个参数的部分分配空间，这对应图中的*参数构造区*

内存管理

我们知道在过程的开始时，会通过 `add` 指令分配栈帧，在过程结束时会通过 `sub` 指令回收栈帧，但我们还需要细化这个过程

栈上的局部存储