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
`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	Functione	Register	Function

Register	Functione	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
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

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