# Machine Programming Basic

汪之立

## 课本章节

- ▶ 引言 机器代码与汇编代码, x86机器和指令集的发展
- ▶ 程序编码 C程序的编译过程与反汇编
- ▶ 数据格式 x86-64指令集中的数据类型

### 理解&应用

- ▶ 访问信息
  - 通用目的寄存器
  - 操作数指示符
  - 数据传输指令
  - 入栈出栈指令
- ▶ 算术和逻辑操作
  - 加载有效地址
  - +, -, \*, /, &, ^, |, ++, --, ~, -
  - 算术/逻辑左右移
  - 特殊算术

### 目录

- ► C程序的编译过程与反汇编概述
  - 指令集&虚拟内存
  - 编译&反汇编
- ▶ x86-64指令集中的数据类型
  - 数据格式
  - 通用寄存器
  - 操作数指示符
- ▶ 汇编代码的基本操作
  - 数据传输、压入和弹出栈数据、算术和逻辑操作
- **Utils** 
  - ATT&Intel汇编区别
  - 调用者保存与被调用者保存
  - CISC&RISC

### 指令集&虚拟内存 计算机系统的两种抽象

- ▶指令集体系结构或指令集架构(Instruction Set Architecture, ISA)
  - 定义机器级程序的格式和行为,包括处理器状态、指令的格式、以及每条指令对状态的影响
  - 大多数ISA,包括x86-64,将程序的行为描述成好像每条指令都是按顺序执行的,实际硬件实现上是并发的
  - 举例: Intel: IA32, x86-64; ARM; RISC V等

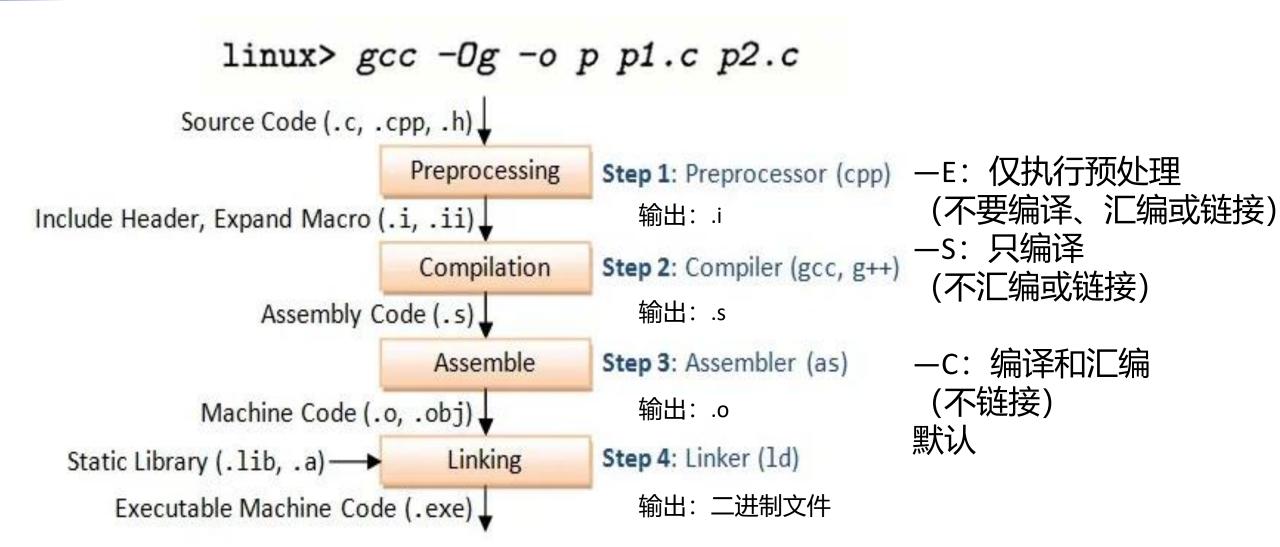
### ▶虚拟内存

- 机器级程序使用的内存地址是虚拟地址,类似巨大的字节数组
- 实际实现是将多个硬件存储器和操作系统软件组合起来,虚拟的地址空间 是由操作系统管理的

## 机器代码中展示的处理器状态

- ▶程序计数器 (PC/x86-64:%rip):下一条指令地址
- ▶整数寄存器(16个):地址/整数数据,存储临时变量等
- ▶条件码寄存器: 算术或逻辑指令的状态信息, 控制条件变化、
- ▶ 向量寄存器: 一个或多个整数或浮点数值

## GCC编译过程



图片源自https://zhuanlan.zhihu.com/p/111500914

## GCC编译过程

### 优化等级

-O0/Og: 不做任何优化,这是默认的编译选项。

- -O1: 优化会消耗少多的编译时间,它主要对代码的分支,常量以及表达式等进行优化。(比如使用条件传送)
- -O2:会尝试更多的寄存器级的优化以及指令级的优化,它会在编译期间占用更多的内存和编译时间。
- -O3: 在O2的基础上进行更多的优化(比如内联简单的函数到被调用函数中)

### 编译&反汇编

### 生成.s文件

linux> gcc -Og -S mstore.c

```
multstore:

pushq %rbx

movq %rdx, %rbx

call mult2

movq %rax, (%rbx)

popq %rbx

ret
```

```
linux> objdump -d mstore.o
            -l: 显示行号 重定向-d <file> > <file.txt>
Disassembly of function multstore in binary file mstore.o
00000000000000000 <multstore>:
                              Equivalent assembly language
Offset
      Bytes
       53
                                     %rbx
                              push
  1: 48 89 d3
                                     %rdx,%rbx
                              mov
  4: e8 00 00 00 00
                              callq 9 <multstore+0x9>
  9: 48 89 03
                                     %rax,(%rbx)
                              mov
                                     %rbx
  c:
      5b
                              pop
  d:
                              retq
```

- 反汇编器只是基于机器代码文件中的字节序列来确定汇编代码
- 字节序列从某个给定位置开始,可以将字节唯一地解码成机器指令
- 部分指令代表字数的符号会有增删
- 链接器会为机器找到函数调用的可执行代码的位置,并优化文件的存储
- 实际GCC产生的汇编代码有'.'开头的指导汇编器和链接器工作的伪指令

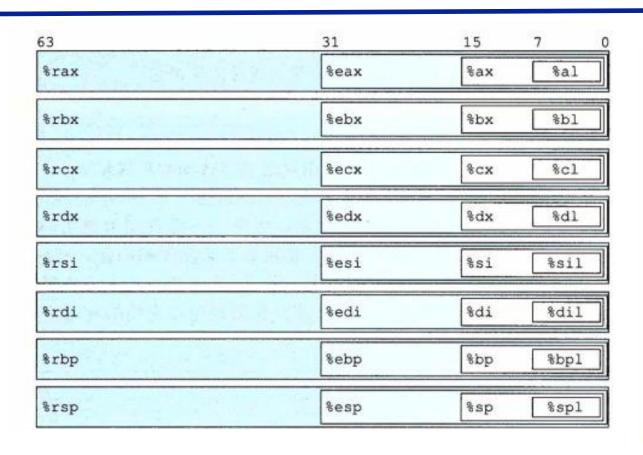
### 数据格式

| C声明    | Intel 数据类型 | 汇编代码后缀      | 大小(字节) |
|--------|------------|-------------|--------|
| char   | 字节         | b byte      | 1      |
| short  | 字          | w word      | 2      |
| int    | 双字         | 1 long word | 4      |
| long   | 四字         | q quad word | 8      |
| char*  | 四字         | g quad word | 8      |
| float  | 单精度        | s single    | 4      |
| double | 双精度        | 1 long      | 8      |

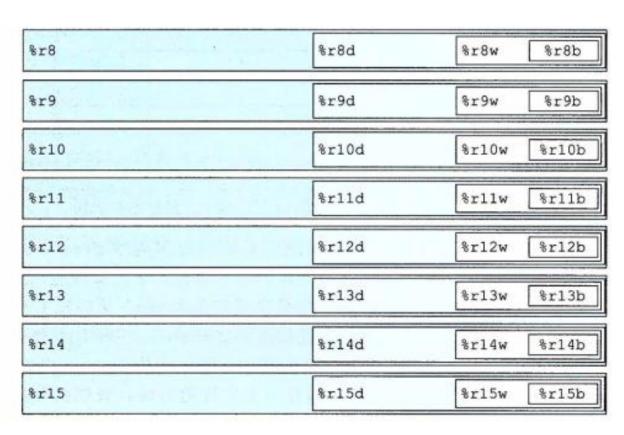
图 3-1 C语言数据类型在 x86-64 中的大小。在 64 位机器中,指针长 8 字节

- 某类指令后加上汇编后缀代表操作的字节数
- 浮点数和整数使用不同的指令和寄存器 (所以后缀不会混)

### 16个64位通用目的寄存器 地址&整数



现今除了%rsp作为栈指针保留 其他没有专用



%rax:返回值

%rdi: 第一个参数 %rsi: 第二个参数

%rdx: 第三个参数 %rcx: 第四个参数

%r8: 第五个参数 %r9: 第六个参数

## 指令字节操作

### 对于生成小于8字节的指令

- 生成1字节和2字节的指令会保持剩下的字节不变
- 生成4字节数字的指令会把高位4个字节置为0 (对 应movl)

### 操作数指示符 ATT格式

- ▶ 操作数 指示出执行一个操作中要使用的源数据值,以及放置结果的目的位置。
- ▶ 数据形式
  - 源数据值:常数、寄存器或内存中读出 结果:寄存器或内存
- ▶ 类型
  - 立即数:用来表示常数值 \$组合C表示法表示的数字 例如:\$0x400,\$-533
  - 寄存器:表示某个寄存器的内容,低位1字节、2字节、4字节或8字节中的一个
  - 内存引用:根据计算出来的地址(通常称为有效地址)访问某个内存位置。
  - 用ra表示寄存器,R[ra]表示寄存器里的值,用add表示地址,M[add]表示内存中 地址add处的值
- ▶ 寻址模式  $Imm(r_b, r_i, s) := Imm + R[r_b] + R[r_i] * s$  配合leaq可能仅仅是简化的算术运算 比例因子s必须为1, 2, 4, 8; 基址寄存器 $r_b$ ; 变址寄存器 $r_i$ ; 立即数偏移Imm

### 数据传输指令 MOV类

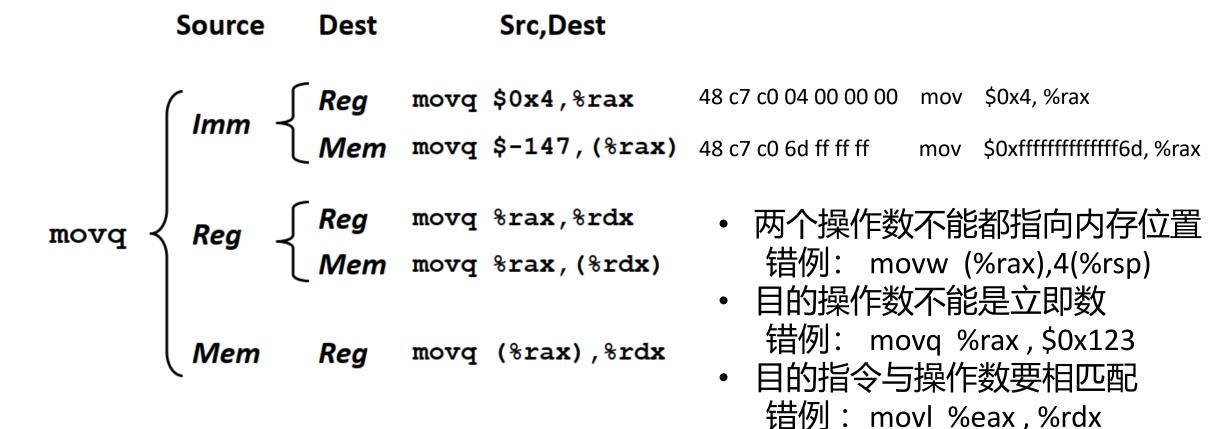
| 打       | 令    | 效果   | 描述      |
|---------|------|------|---------|
| MOV     | S, D | D←S  | 传送      |
| movb    |      |      | 传送字节    |
| movw    |      |      | 传送字     |
| movl    |      |      | 传送双字    |
| movq    |      | 2 2  | 传送四字    |
| movabsq | I, R | R∗-I | 传送绝对的四字 |

图 3-4 简单的数据传送指令

- movl指令以寄存器为目标时,它会把该寄存器的高位4字节设置为0
- movabsq能以任意64位立即数值作为源操作数,只能以寄存器作为目的;
- movq当源操作数是立即数时,只能用表示为32位补码数字的立即数,然后把这个 值符号扩展到64位

### 数据传输指令 MOV类

### 操作数分类



### 数据传输指令 MOVZ类&MOVS类

| 指令     |      | 效果            | 描述             |  |
|--------|------|---------------|----------------|--|
| MOVZ   | S, R | R←零扩展(S)      | 以零扩展进行传送       |  |
| movzbw |      |               | 将做了零扩展的字节传送到字  |  |
| movzbl |      | 将做了零扩展的字节传送到双 |                |  |
| movzwl |      |               | 将做了零扩展的字传送到双字  |  |
| pdsvom |      |               | 将做了零扩展的字节传送到四字 |  |
| movzwq |      |               | 将做了零扩展的字传送到四字  |  |

图 3-5 零扩展数据传送指令。这些指令以寄存器或内存地址作为源,以寄存器作为目的

| 指令     |      | 效果               | 描述              |  |
|--------|------|------------------|-----------------|--|
| MOVS   | S, R | R←符号扩展(S)        | 传送符号扩展的字节       |  |
| movsbw |      |                  | 将做了符号扩展的字节传送到字  |  |
| movsbl |      | 1                | 将做了符号扩展的字节传送到双字 |  |
| movswl |      |                  | 将做了符号扩展的字传送到双字  |  |
| movsbq |      |                  | 将做了符号扩展的字节传送到四字 |  |
| movswq |      |                  | 将做了符号扩展的字传送到四字  |  |
| movslq |      |                  | 将做了符号扩展的双字传送到四字 |  |
| cltq   |      | %rax ←符号扩展(%eax) | 把%eax 符号扩展到%rax |  |

图 3-6 符号扩展数据传送指令。MOVS指令以寄存器或内存地址作为源,以寄存器 作为目的。cltq指令只作用于寄存器%eax和%rax

- 不存在movzlq:movl已经实现
- cltq:编码更为紧致

### 压入和弹出栈数据

| 指令    |   | 效果                                    | 描述     |
|-------|---|---------------------------------------|--------|
| pushq | s | R[%rsp] -R[%rsp] -8;<br>M[R[%rsp]] -S | 将四字压人栈 |
| popq  | D | D←M[R[%rsp]]; R[%rsp]←R[%rsp]+8       | 将四字弹出栈 |

subq \$8, %rsp movq %rbp, (%rsp)

movq (%rsp), %rax addq \$8, %rsp

图 3-8 入栈和出栈指令

- 在x86-64中,程序栈向下增长,栈顶元素的地址是所有栈中 元素地址中最低的
- 栈指针%rsp保存着栈顶元素的地址(压减出加)
- 程序可以用标准的内存寻址方法访问栈内的任意位置
- 此指令是单字节编码的

pushq和popq分别 等价于用mov和算 术指令组合成的哪 两条指令?

### 加载有效地址(load effective address)

| 指令        | 效果     | 描述     |  |
|-----------|--------|--------|--|
| leaq S, D | D ← &S | 加载有效地址 |  |

• leaq的指令形式是从内存读数据到寄存器,但实际上它根本没有引用内存,仅仅是

将有效地址写入目标操作数

• leaq有时仅仅是简介描述普通的算术操作(利用寻址)

```
5 cal:
6 leaq (%rdi,%rsi,8), %rax
7 ret
```

对应的long cal(long x, long y )函数返回了什么值

$$x+8y$$
 {long t = x + y \* 8; return t;}

```
cal:
 6
         pushq
                  %rbp
                  %rsp, %rbp
         movq
                 %rdi, -24(%rbp)
 8
         mova
                 %rsi, -32(%rbp)
 9
         mova
                  -32(%rbp), %rax
10
         movq
                  0(,%rax,8), %rdx
11
         leaq
12
                  -24(%rbp), %rax
         movq
13
                  %rdx, %rax
         addq
                  %rax, -8(%rbp)
14
         movq
                  -8(%rbp), %rax
15
         movq
                  %rbp
16
         popq
17
         ret
```

### 一元&二元&位移操作

| INC  | D    | $D \leftarrow D + 1$        | 加1         |
|------|------|-----------------------------|------------|
| DEC  | D    | $D \leftarrow D - 1$        | 减1         |
| NEG  | D    | $D \leftarrow -D$           | 取负         |
| NOT  | D    | D ← ~D                      | 取补         |
| ADD  | S, D | $D \leftarrow D + S$        | DIT I      |
| SUB  | S, D | $D \leftarrow D - S$        | 减          |
| IMUL | S, D | $D \leftarrow D * S$        | 乘          |
| XOR  | S, D | $D \leftarrow D \cap S$     | 异或         |
| OR   | S, D | $D \leftarrow D \mid S$     | 或          |
| AND  | S, D | $D \leftarrow D \& S$       | 与          |
| SAL  | k, D | $D \leftarrow D \lessdot k$ | 左移         |
| SHL  | k, D | $D \leftarrow D \lessdot k$ | 左移(等同于SAL) |
| SAR  | k, D | $D \leftarrow D >>_A k$     | 算术右移       |
| SHR  | k, D | $D \leftarrow D >>_L k$     | 逻辑右移       |

### ▶ 一元操作

- 只有一个操作数,可以为寄存器也可以为内存
- ▶ 二元操作
  - 结果写入目标操作数,目的操作数不能为立即数
  - 如果目的操作数为内存位置,处理器需要先从内存读出值,再执行操作,最后把结果写回内存
- ▶ 位移操作
  - 移位量(k)可以是立即数,或者存储在 %c1 (单字节寄存器) 中的值
  - 移位操作对于w位长的数据值进行操作的时候,移位量是由%c1的低m位决定的,这里 $2^m = w$

## 特殊算数操作(八字)

| 指令      |     | 效果   | 描述     |  |
|---------|-----|--|--------|--|
| imulq   | S   | $R[%rdx]: R[%rax] \leftarrow S \times R[%rax]$   | 有符号全乘法 |  |
| mulq    | S   | $R[\$rdx]: R[\$rax] \leftarrow S \times R[\$rax]$  | 无符号全乘法 |  |
| cqto(cc | lo) | R[%rdx]: R[%rax]←符号扩展(R[%rax])   | 转换为八字  |  |
| idivq   | S   | $R[\rdx] \leftarrow R[\rdx]: R[\rdx] \mod S$<br>$R[\rdx] \leftarrow R[\rdx]: R[\rdx] \div S$       | 有符号除法  |  |
| divq    | s   | $R[%rdx] \leftarrow R[%rdx]$ : $R[%rax] \mod S$<br>$R[%rdx] \leftarrow R[%rdx]$ : $R[%rax] \div S$ | 无符号除法  |  |

图 3-12 特殊的算术操作。这些操作提供了有符号和无符号数的全 128 位乘法和除法。

一对寄存器%rdx 和%rax 组成一个 128 位的八字

单操作数,乘法要求一个乘数放在%rax,除法要求被除数%rdx(高64位),%rax(低64位)

结果 乘法: %rdx (高64位), %rax (低64位)

除法: %rax (商), %rdx (余数)

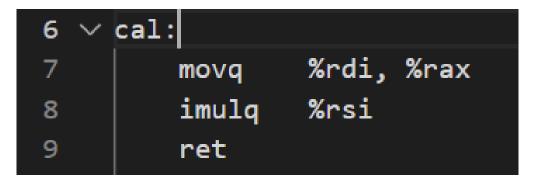
## 特殊算数操作(八字)

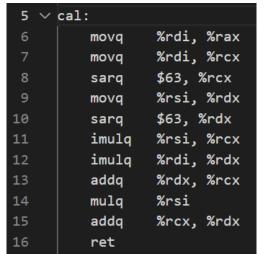
```
void cal(unsigned ___int128 *dest,
unsigned long x, unsigned long y)
dest: %rdi x: %rsi y: %rdx
{
   *dest = x * (unsigned ___int128)y;
}
```

### 改为有符号版本 (再简化一点)

\_\_int128 cal(long x, long y) {return x \* (\_\_int128)y;}

#### 预期结果:





上述结果要 -O2 才行 实际默认是将高 低位分开计算

## 特殊算数操作(八字)

### 改为无符号版本

```
5 ∨ cal:
 6
          endbr64
                   %rdi, %rax
          movq
                   %rdx, %r8
 8
          movq
 9
          cqto
10
          idivq
                  %rsi
                   %rax, (%r8)
11
          movq
                  %rdx, (%rcx)
12
          movq
13
          ret
```

```
cal:
          endbr64
 6
                  %rdi, %rax
          movq
                  %rdx, %r8
 8
          movq
                  $0, %edx
          movl
10
          divq
                  %rsi
11
                  %rax, (%r8)
          movq
12
                  %rdx, (%rcx)
          movq
13
          ret
```

### ATT VS Intel 汇编

- Intel 代码省略了指示大小的后缀。我们看到指令push和mov,而不是pushq和movq。
- Intel代码省略了寄存器名字前面的"%'符号,用的是rbx,而不是%rbx。
- Intel代码用不同的方式来描述内存中的位置,例如是"QWORD PTR [rbx] '而不是'(%rbx)'。
- 在带有多个操作数的指令情况下,列出操作数的顺序相反。

ATT: instuction sourse destination

Intel: instuction destination sourse

### 调用者保存与被调用者保存

函数A调用了函数B,寄存器rbx在函数B中被修改了,逻辑上%rbx内容在调用函数B的前后应该保持一致,解决这个问题有两个策略,

- (1)在函数A在调用函数B之前提前保存寄存器%rbx的内容,执行完函数B之后再恢复%rbx的内容,这个 策略就称为调用者保存;
- (2)函数B在使用寄存器%rbx, 先保存寄存器%rbx的值, 在函数B返回之前, 要恢复寄存器%rbx原来存储的内容, 这种策略被称之为被调用者保存。

#### 4.X86-64中除了%rsp以外的十五个通用寄存器的保存策略

#### callee saved(被调用者保存)

%rbx , %rbp, %r12, %r13, %r14, %r15

#### caller saved(调用者保存)

%r10, %r11, %rax, %rdi, %rsi, %rdx, %rcx, %8, %r9

### CISC & RISC

- ▶ CISC (Complex Instruction Set Computer) , 即 "复杂指令系统计算机"
- 从计算机诞生一直沿用的指令集方式
- 指令体系比较丰富,有专用指令来完成特定的功能,但计算机各部分的利用率不高,执行速度慢
- 桌面通用计算机流行的x86体系结构
- ▶ RISC(Reduced Instruction Set Computer),即"精简指令集计算机"
- 执行较少类型计算机指令的微处理器
- 以更快的速度执行常规操作,特殊指令只能由常规指令组合执行
- 数据中心等专用领域,移动设备上ARM架构

### CISC & RISC

#### **RISC**

- 指令长度固定,指令格式种类少,寻址方式种类少
- 只有取数/存数指令访问存储器,其余指令的操作都在• 寄存器内完成
- 一定采用流水线,大部分指令在一个时钟周期内完成 •
- 控制器采用组合逻辑控制,不用微程序控制
- 采用优化的编译程序
- CPU中有更多个通用寄存器

#### 优势

- 更能提高计算机运算速度
- 更便于设计,可降低成本,提高可靠性。
- 有效支持高级语言程序。

#### **CICS**

- 指令长度不固定,指令格式种类多,寻址方式种类多
- 可以访存的指令不受限制
- 各种指令执行时间相差很大,大多数指令未 经优化需多个时钟周期才能完成。
- 控制器大多数采用微程序控制。
- 难以用优化编译生成高效的目标代码程序

#### 优势

• 处理特殊任务效率高

### Acknowledgment

感谢陈向群老师、杨智老师、邹磊老师对课程的教授 感谢孙英博学长、刘逸兴学长、仝昊学长对学习以及ppt制作提供的帮助

# Thanks

Litchi-w