

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

```
linux> gcc -Og -o p p1.c p2.c
```



- 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
```

```
linux> objdump -d mstore.o
```

-l: 显示行号

重定向-d <file> > <file.txt>

```
multstore:
```

```
pushq    %rbx
movq     %rdx, %rbx
call     mult2
movq     %rax, (%rbx)
popq     %rbx
ret
```

Disassembly of function multstore in binary file mstore.o

```
1 0000000000000000 <multstore>:
```

Offset Bytes

```
2 0: 53
3 1: 48 89 d3
4 4: e8 00 00 00 00
5 9: 48 89 03
6 c: 5b
7 d: c3
```

Equivalent assembly language

```
push    %rbx
mov     %rdx,%rbx
callq   9 <multstore+0x9>
mov     %rax, (%rbx)
pop     %rbx
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	l long word	4
long	四字	q quad word	8
char*	四字	q quad word	8
float	单精度	s single	4
double	双精度	l 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

63	31	15	7	0
%rax	%eax	%ax	%al	
%rbx	%ebx	%bx	%bl	
%rcx	%ecx	%cx	%cl	
%rdx	%edx	%dx	%dl	
%rsi	%esi	%si	%sil	
%rdi	%edi	%di	%dil	
%rbp	%ebp	%bp	%bpl	
%rsp	%esp	%sp	%spl	

%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

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 r_a 表示寄存器， $R[r_a]$ 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

指令	效果	描述
MOV S, D	$D \leftarrow S$	传送
movb		传送字节
movw		传送字
movl		传送双字
movq		传送四字
movabsq I, R	$R \leftarrow I$	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

	Source	Dest	Src, Dest	
movq	Imm	Reg	movq \$0x4, %rax	48 c7 c0 04 00 00 00 mov \$0x4, %rax
		Mem	movq \$-147, (%rax)	48 c7 c0 6d ff ff ff mov \$0xffffffffffff6d, %rax
	Reg	Reg	movq %rax, %rdx	
		Mem	movq %rax, (%rdx)	
	Mem	Reg		
			movq (%rax), %rdx	

- 两个操作数不能都指向内存位置
错例：movw (%rax), 4(%rsp)
- 目的操作数不能是立即数
错例：movq %rax, \$0x123
- 目的指令与操作数要相匹配
错例：movl %eax, %rdx

数据传输指令 MOVZ 类&MOVS 类

指令	效果	描述
MOVZ S, R	$R \leftarrow \text{零扩展}(S)$	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字
movzwl		将做了零扩展的字传送到双字
movzbq		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

- 不存在movzld:movl已经实现
- cltq: 编码更为紧致

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	$R \leftarrow \text{符号扩展}(S)$	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		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	$\%rax \leftarrow \text{符号扩展}(\%eax)$	将做了符号扩展的双字传送到四字
cltq		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

压入和弹出栈数据

指令	效果	描述
pushq S	$R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M[R[\%rsp]] \leftarrow S$	将四字压入栈
popq D	$D \leftarrow M[R[\%rsp]];$ $R[\%rsp] \leftarrow R[\%rsp] + 8$	将四字弹出栈

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

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

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

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
<code>leaq S, D</code>	$D \leftarrow \&S$	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
```

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

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	$D \leftarrow \sim D$	取补
ADD	S, D	$D \leftarrow D + S$	加
SUB	S, D	$D \leftarrow D - S$	减
IMUL	S, D	$D \leftarrow D * S$	乘
XOR	S, D	$D \leftarrow D \wedge S$	异或
OR	S, D	$D \leftarrow D \vee S$	或
AND	S, D	$D \leftarrow D \& S$	与
SAL	k, D	$D \leftarrow D \ll k$	左移
SHL	k, D	$D \leftarrow D \ll k$	左移（等同于SAL）
SAR	k, D	$D \leftarrow D \gg_A k$	算术右移
SHR	k, D	$D \leftarrow D \gg_L k$	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(cqo)	$R[\%rdx]: R[\%rax] \leftarrow \text{符号扩展}(R[\%rax])$	转换为八字
idivq S	$R[\%rdx] \leftarrow R[\%rdx]: R[\%rax] \bmod S$ $R[\%rdx] \leftarrow R[\%rdx]: R[\%rax] \div S$	有符号除法
divq S	$R[\%rdx] \leftarrow R[\%rdx]: R[\%rax] \bmod S$ $R[\%rdx] \leftarrow R[\%rdx]: R[\%rax] \div S$	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;  
}
```

```
5  ∨ cal:  
6      movq    %rsi, %rax  
7      mulq    %rdx  
8      movq    %rax, (%rdi)  
9      movq    %rdx, 8(%rdi)  
10     ret
```

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：

```
6  ∨ cal:  
7      movq    %rdi, %rax  
8      imulq   %rsi  
9      ret
```

```
5  ∨ cal:  
6      movq    %rdi, %rax  
7      movq    %rdi, %rcx  
8      sarq    $63, %rcx  
9      movq    %rsi, %rdx  
10     sarq    $63, %rdx  
11     imulq   %rsi, %rcx  
12     imulq   %rdi, %rdx  
13     addq    %rdx, %rcx  
14     mulq    %rsi  
15     addq    %rcx, %rdx  
16     ret
```

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）

```
void cal(long x, long y, long *des1, long *des2)
{
    %rdi %rsi    %rdx    %rcx
    long q = x / y;
    long r = x % y;
    *des1 = q;
    *des2 = r;}

```

改为无符号版本

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

```

```
5      cal:
6      endbr64
7      movq    %rdi, %rax
8      movq    %rdx, %r8
9      movl    $0, %edx
10     divq    %rsi
11     movq    %rax, (%r8)
12     movq    %rdx, (%rcx)
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 source destination

Intel: instuction destination source

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。

CISC

- 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