# 第3章 程序的机器级表示

* **简介**

32位机器可使用约4GB（232字节）的随机访问存储器（内存）

64位机器可使用256TB（248字节）的随机访问存储器，并且可扩展至16EB（264字节）

* **3.1 历史观点**
* **3.2 程序编码**
* 两种抽象：ISA（Intel: IA32, x86-64; ARM; RISC V），虚拟地址
* 任意给定的时刻，只有有限一部分虚拟地址被认为是合法的
* 一般的实现中虚拟地址的高16位必须为0（一个地址实际上能够指定的是256TB范围内的一个字节）
* 机器最终执行字节序列
* x86-64指令长度：1-15byte(s)

从某个给定位置开始，可以将字节序列唯一地解码成机器指令

* nop指令：占位，使函数代码变为16字节
* 运行链接器时对应的一组目标代码文件中必须含有一个main函数
* PF(parity flag，条件码) ：处理器执行算数或逻辑运算时，若结果低8位中有偶数个1则设为1，若有奇数个1则设为0

C中计算该信息至少需要7次位运算操作

* 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）

* 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
* **3.3 数据格式**
  + C语言数据类型在x86-64中的大小

|  |  |  |  |
| --- | --- | --- | --- |
| C声明 | Intel数据类型 | 汇编代码后缀 | 大小（字节） |
| float | 双字 | s | 4 |
| double | 四字 | l | 8 |

* + 浮点数使用的是一组完全不同的指令和寄存器
* **3.4 访问信息**
  + 16个通用目的（整数）寄存器
    - 生成1、2字节指令保持剩余字节不变，生成4字节的指令使高位4字节置0
  + 操作数格式与寻址模式

基址寄存器rb；变址寄存器ri；立即数偏移Imm；比例因子s

注意Imm前不加$，只有单独使用时加$，$Imm!=Imm

* **数据传送指令MOV**
  + MOV S,D S，D不能同时指向内存
  + movb,movw,movl,movq: S指定一个立即数，D指定一个位置（寄存器/内存，不能是立即数）
  + movabsq: 以任意64位立即数值作为源操作数，只能以寄存器做为目的

常规的movq指令只能以表示为32位补码数字的立即数作为源操作数，符号扩展得到64位的值

* + 寄存器部分大小必须与指令后缀指定的大小匹配
* **零扩展数据传送指令MOVZ**
  + 不能以立即数作为源，目的只能是寄存器
  + 没有movzlq，不需要，用movl即可实现（高4字节自动清零）
* **符号扩展数据传送指令MOVS**
  + 多了一个movslq
  + cltq无操作数，与指令movslq %eax,%rax完全一致，但编码更紧凑
  + 不能以立即数作为源，目的只能是寄存器
* **入栈和出栈指令**
  + 栈保存在内存中
  + popq后被pop出的数据仍在栈的原位置，只是%rsp+8
  + 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源操作数：立即数/寄存器/内存
  + pop目的操作数：寄存器/内存
* 其他内容
  + 间接引用指针（如作为函数参数）就是将该指针放在一个寄存器中，然后在内存引用中应用这个寄存器
  + 局部变量通常保存在寄存器中而非内存中（访问寄存器比访问内存快得多）
  + 执行强制类型转换时如果既要改变位长又要改变符号（无/有符号），则先扩展位长（第二章）
  + unsigned char->long 先用movzbl而不是movzbq（无论什么指令，只要是生成4字节值并以寄存器为目的，寄存器高4位字节都置为0）
  + \*p p的间接引用；&a 创建指针
  + 对于64位机器，(%ebx)的寻址模式是错误的，因为地址都是8字节的
* **3.5 算术和逻辑操作**
  + **加载有效地址**
    - 源操作数是内存引用的形式
    - 目的操作数必须是寄存器
    - 只有leaq，没有leab,leaw,leal
  + **一元操作**
    - 一个操作数，内存/寄存器
  + **二元操作**
    - 第二个操作数即是源又是目的
    - 第一个操作数：立即数/寄存器/内存
    - 第二个操作数：寄存器/内存（处理器先从内存中读出值，执行操作，再把结果写回内存）
  + **移位操作**
    - 第一个操作数是移位量，第二个操作数是要移位的数
    - 移位量：立即数/存放在%cl中的单字节数

w位的数据，移位量由%cl的低log2w位决定，即移位量=%cl mod w

当%cl=0xFF时，salb移7位，salw移15位…

* 第二个操作数：寄存器/内存
  + **特殊的算术操作**
    - 乘法：%rdx（高64位），%rax（低64位）

除法：%rax（商），%rdx（余数）

* + - 乘法的结果传送到内存时要考虑大端法和小端法
    - 对于大多数除法应用，被除数常常也是64位的，要先用cqto扩展到128位

使用divq指令时，%rdx通常会事先设为0（无符号）

* + - 源操作数：立即数/寄存器/内存
* **3.6 控制**
  + **条件码**
    - 单个位，条件码寄存器
    - CF: 进位/借位标志

ZF: 零标志

SF: 符号标志

OF: 补码溢出标志

* leaq指令不改变条件码
* 除leaq以外的**整数算数操作（包括逻辑操作）**均会设置条件码

逻辑操作: CF=0, OF=0,

移位操作: CF=最后一个被移出的位，OF=0

INC,DEC: 设置OF, ZF，不设置CF

无符号数做加减法也会设置OF标志位（机器不区分有无符号）

无符号数减法发生了借位，也会设置CF为1

* CMP指令和TEST指令

只设置条件码而不改变寄存器

* SET指令：根据条件码的某种组合设置寄存器/内存字节

为了得到32/64位结果，要对高位清零

有符号<: SF^OF

无符号<: CF

* **跳转指令**
* 无条件跳转jmp有直接/间接跳转，条件跳转只有直接跳转
* 汇编器确定所有带标号指令的地址，并将跳转目标（目的指令的地址）编码为跳转指令的一部分
* jmp间接跳转：\*操作数格式（任意一种）
* 跳转指令的编码：PC相对的
  + 编码（1/2/4字节）=目标指令地址-跳转指令后面一条指令的地址（%rip）
  + 编码超过4字节（4GB），则使用间接跳转，先计算出目标指令地址，存入寄存器中，再间接跳转即可
  + 早期的处理器会将**更新程序计数器**作为执行一条指令的第一步
  + 指令编码简洁且可以不做改变就移到内存中的不同位置
* rep(z)指令作为一种空操作放在ret指令前

当ret指令通过跳转指令到达（包括判断后不跳转而到达）时，处理器不能正确预测ret指令的目的

* + **条件控制实现条件分支**
    - 条件控制=控制的条件转移，根据条件决定选择执行哪些代码
    - 汇编代码通用形式
  + **条件传送实现条件分支**
    - 条件传送=数据的条件转移，开优化后使用
    - 汇编代码通用形式
    - cmov可以在满足后缀对应的条件时，传送16位、32位、64位值，但不支持单字节传送，编译器可以从目标寄存器大小来推断传送的数据的位数
    - cmovx *S，R* 其中S是源(寄存器或内存地址)，R是目标寄存器
    - 优点

可以更好地填充流水线（避免分支预测错误的惩罚，计算）

* + - 缺点

不是所有条件表达式都可以用条件传送编译

无论如何都会对两种情况的表达式求值，可能导致非法行为(xp?\*xp:0)

不一定省时间

* + **循环**
    - do-while循环
      * 汇编代码通用形式
    - while循环
      * 汇编代码通用形式

jump to middle. *-Og*

guarded do。 *-O1*

当循环开始时测试条件一定满足时，编译器会优化掉初始判断

编译器常常可以优化初始测试

* + - for循环
      * 汇编代码通用形式（和while类似）
      * 注意continue
  + **switch语句**
    - 跳转表
      * 当开关情况比较多且值的跨度比较小时就会使用跳转表
      * case 100-106 映射到无符号整数类型（简化了分支的可能性） index， \*jt[index]
* **3.7过程**
  + **运行时栈**
    - P的返回地址是P的栈帧的一部分
    - 大多数过程的栈帧都是定长的，在过程的开始就分配好了
    - 寄存器传参最多6个，剩下的要在调用之前在自己的栈帧里存储好这些参数
    - 许多函数根本不需要栈帧
      * 不会调用其它过程
      * 所有局部变量都可以保存在寄存器里
    - 栈帧从高地址到低地址的结构
      * 被保存的寄存器
      * 局部变量
      * 参数构造区（n,n-1,…,7）
      * （返回地址）
  + **转移控制**
    - call指令
      * push+jmp（设置pc）
      * 支持直接跳转和间接跳转
    - ret指令
      * pop+jmp（设置pc）
  + **参数传递**
    - 多的参数放到自己的栈帧里（参数构造区）
    - 传递参数时所有数据大小都向8字节（64位）的倍数对齐（书p170图）
  + **栈上的局部存储**
    - 局部变量（指**当前过程**中的）必须放到自身栈帧上的情况
      * 寄存器不够用
      * &局部变量，产生地址；局部变量要入栈
      * 局部变量是数组/结构，必须能通过**数组/结构引用**访问
  + **寄存器中的局部存储**
    - 调用者保存 %rax,%rcx,%rdx,%rsi,%rdi,%rbp,%r8,%r9,%r10,%r11
    - 被调用者保存寄存器 %rbx,%rbp,%r12,%r13,%r14,%r15
    - %rsp哪个都不是
    - 被调用者保存寄存器入栈保存的顺序：上面的反过来
    - 被调用者保存寄存器不一定总是要入栈，只有当前过程需要修改的时候才入栈
  + 递归过程
    - 注意逻辑关系，在当前栈帧要保存的**被调用者保存寄存器**存的是上次调用的数据，因为返回时要通过pop恢复上次调用的“状态”
    - 所以在调用者运行的过程中，要把**需要保留的数据**传入被调用者保存寄存器里
* **3.8 数组分配和访问**
  + **基本原则**
    - T A[N]; 分配内存+引入标识符A（指针）
  + **指针运算**
    - p（指针）+i
    - &产生指针，\*间接引用指针
    - 数组引用A[i]等价于\*(A+i)
    - 同一个数据结构两个指针之差：&E[i]-E=i(long)
  + **嵌套的数组**
    - 行优先存储
    - 对于定长数组，很多时候编译器会进行优化
    - 变长数组
      * 可以作为局部变量/函数参数
      * 参数n要放在A[n][n]之前
      * 访问元素A[i][j]

不优化的情况下要使用imulq计算n\*i，耗时间

优化的情况下GCC能够识别出程序访问多维数组元素的步长，避免计算n\*i？

* **3.9 异质的数据结构**
  + **结构**
    - 编译器维护关于每个结构类型的信息，指示每个字段的字节偏移量
    - 机器代码不包含关于字段声明或字段名字的任何信息
  + **联合**
    - 内部节点无数据的二叉树
    - 访问不同数据类型的位模式
      * 注意字节顺序问题（大端小端）
  + **数据对齐**
    - 任何K字节的基本对象的地址必须是K的倍数
* **3.10 在机器级程序中将控制与数据结合起来**
  + **指针**
    - 指针也有类型，不过不是机器代码中的一部分，是C语言提供的一种抽象
    - 指针的值就是某个对象的地址，NULL指针值
    - &创建指针，&lvalue -> leaq
    - \*间接引用指针，通过内存引用实现
    - 数组名也是指针
    - 指针强制类型转换不改变指针值，改变解读方式和步长
      * char\*p

(int\*) p+7 -> p+4\*7

(int\*) (p+7) ->p+1\*7

* + - 函数指针（函数名也是指针）
      * int (\*fp) (int,int\*);

fp=fun;

* + **内存越界引用和缓冲区溢出**
    - C对数组引用不做任何边界检查
    - gets, strcpy, strcat, sprintf
    - 两种攻击形式
      * 使用系统调用启动一个shell程序，给攻击者提供一组操作系统函数
      * 攻击代码执行一些未授权的任务，之后修复对栈的破坏，然后第二次执行ret，（表面上）正常返回到调用者函数
  + **对抗缓冲区溢出攻击**
    - 栈随机化
      * 安全单一化
      * 地址空间布局随机化，ASLR

程序代码、库代码、栈、全局变量、堆数据

程序开始时，在栈上随机分配一段随机大小的空间（栈随机化），程序不使用这段空间

* + - * 空操作雪橇

由于栈地址在一定范围的随机性，攻击者不能够知道攻击代码注入的地址，而要执行攻击代码需要将函数的返回地址更改为攻击代码的地址（可通过缓冲区溢出的方式改写函数返回地址）。所以，只能在一定范围内（栈随机导致攻击代码地址一定范围内随机）枚举

* + - 栈破坏检测
      * canary，程序每次运行时随机产生，通过段寻址从内存中读入，标记为只读
    - 限制可执行代码区域
      * 三种访问模式：读、写、执行
      * 以前，x86体系结构：读和控制合并成一位；栈必须是可读且可写的
      * NX：不执行位，区分读和控制
  + **变长栈帧**
    - alloca，变长数组
    - %rbp 帧指针，基指针
    - leave指令
      * movq %rbp,%rsp

popq %rbp

* 变长数组也要保持对齐
* **补充内容**
  + 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

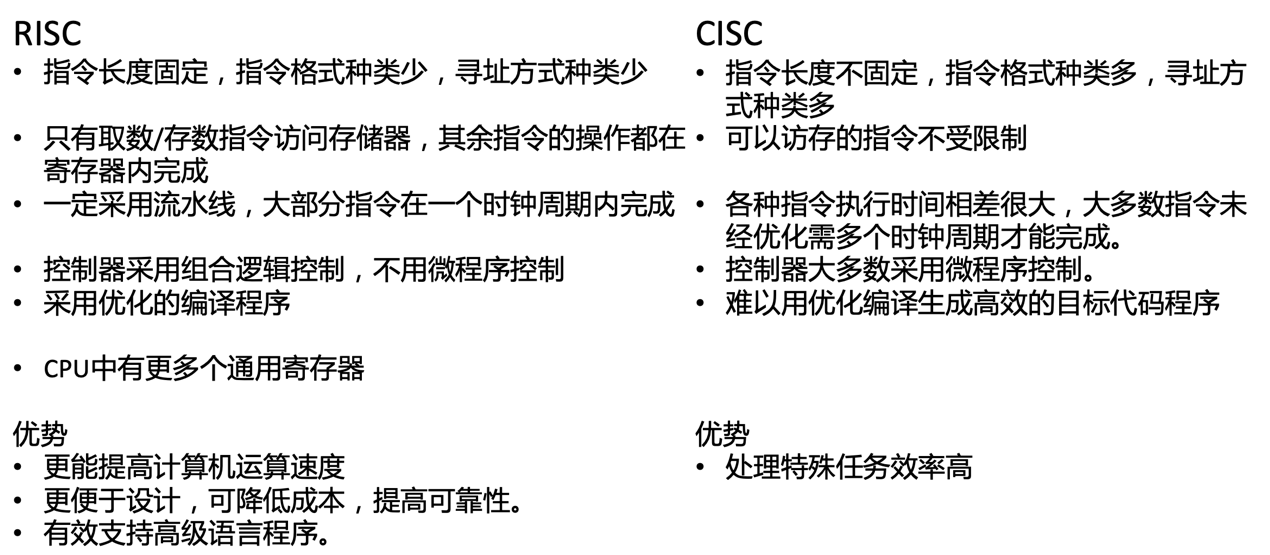
指令体系比较丰富，有专用指令来完成特定的功能，但计算机各部分的利用率不高，执行速度慢

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

* + 指针不能相加
  + 局部数组为什么是随机值？
  + fix\_matrix a[N][N] sizeof(a), sizeof(\*a)
  + \*运算优先级最低，[],()一样
  + union u{char s[7];int a;} sizeof(u)=8