Machine-level Programming: Basics & Control

机器级语言基础知识及控制部分复习

1900012959 李宁远

第3章 程序的机器级表示

3.1 历史（略）P110~112

基于x86-64机器

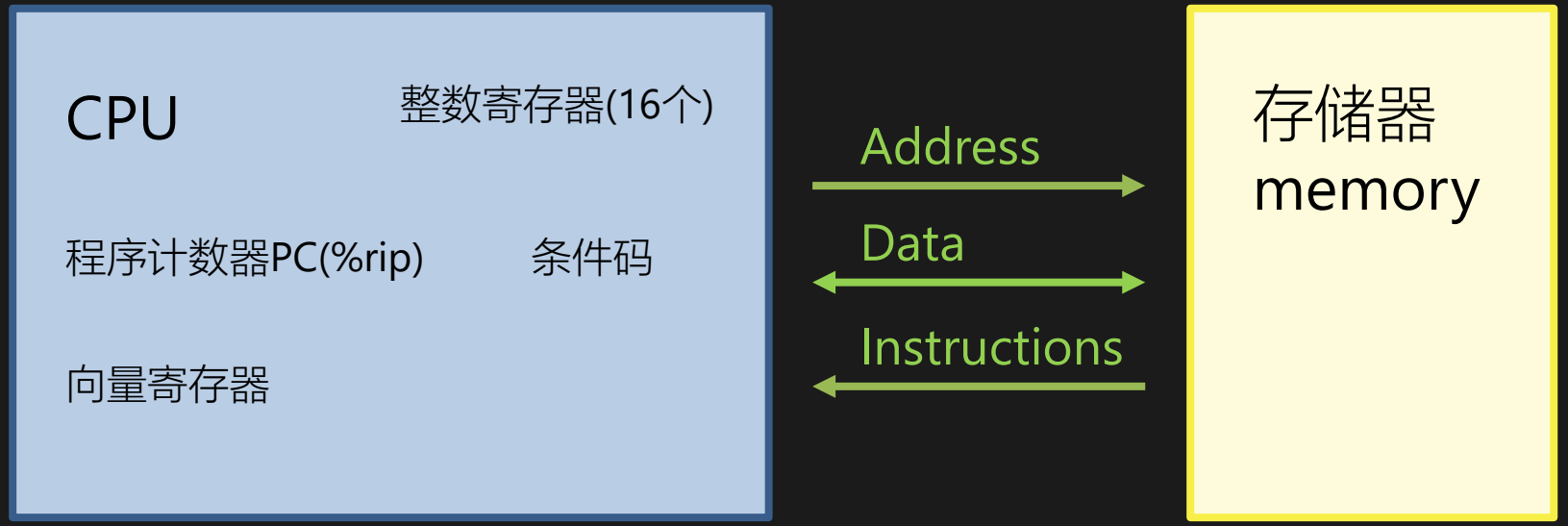
CISC和RISC

3.2 程序编码

gcc编译C程序p1.c的过程：
1.预处理器扩展源代码，插入#include，扩展#define
2.编译器产生汇编代码p1.s
3.汇编器将汇编代码转换成二进制目标代码p1.o
4.链接器将目标代码文件与库函数代码合并，产生可执行文件。

机器级语言的两种抽象：
1.指令集体系结构ISA规定程序的格式和行为，结果与顺序执行一致
2.内存地址是虚拟地址

处理器状态


CPU
整数寄存器(16个)
程序计数器PC(%rip)
条件码
向量寄存器
Address
Data
Instructions
存储器
memory

3.3数据格式

汇编代码后缀b,w,l,q分别表示1,2,4,8字节长度的数据
意思是字节byte，字word，长字long word，四字quad words
例如movb,movw,movl,movq
浮点数分别用s表示float，l表示double

区分：寄存器名的后缀
%r8~%r15的低8,16,32位分别有后缀b,w,d
例如%r10b,%r10w,%r10d
%rax~%rsp的低8位是%al,%bl,%cl,%dl,%sil,%dil,%bpl,%spl

P120 整数寄存器

共16个，以%开头

可以按书上顺序记忆：

%rax,%rbx,%rcx,%rdx,

%rsi,%rdi,//本义是source/destination index，源/目标变址寄存器

%rbp,%rsp,

%r8,...,%r15

（还有%rip）

惯例：调用者(被调用者)保存

P173 3.7.5

寄存器%rbx,%rbp,%r12~%r15为**被调用者保存**寄存器；

所有其他寄存器，除了栈指针%rsp，

都是**调用者保存**寄存器，包括:

%rax(函数返回值),

%rdi,%rsi,%rdx,%rcx,%r8,%r9(函数参数),

%r10,%r11。

注：在P173图3-34的例子中%rbx和%rbp都被作为普通的被调用者保存寄存器使用。

3.4.1 操作数

指令的操作数分三种类型：

1.立即数$Imm

格式为$后面跟一个用标准C表示法表示的整数，比如$-577或$0x1F

2.寄存器，例如%rax,%ecx等

3.存储器

- Imm不带$的立即数表示地址。绝对寻址
- (r_a)寄存器外加括号，如(%rsp)。间接寻址
- Imm(r_b,r_i,s)比例变址寻址，s=1,2,4,8，去掉一些部分就得到 (r_b,r_i,s)、Imm(r_b)、(r_b,r_i)、Imm(r_b,r_i)、(,r_i,s)、Imm(,r_i,s)
- 没有ri时称为（基址+偏移量）寻址，没有s时称为变址寻址

3.4.2 mov

MOV S,D

movb,movw,movl,movq,movabsq

目标不能是立即数

寄存器大小要与指令末尾的字符匹配

movl以寄存器为目的时会将高32位置0。任何为寄存器生成32位值的指令都会把该寄存器的高位部分置成0。

不过改变内存里的低位时不会清空高位。

movabsq的S是64位立即数，目的只能是寄存器。其他mov只能用32位立即数为操作数。

3.4.2 mov

MOVZ S,R 用0填充高位，传送到较长的寄存器

MOVS S,R 用符号扩展填充高位，传送到较长的寄存器

以寄存器或内存为源，以寄存器为目的。

movz有5个，movs有6个。bw,bl,bq,wl,wq,lq

假如有movzlq，movzlq %ecx,%rdx相当于movl %ecx,%edx，所以没有movzlq。

cltq=movslq %eax,%rax 没有操作数

后面还有cqto,向%rdx:%rax填入%rax的符号扩展，即用%rax的符号填充%rdx

3.4.4 压入和弹出栈数据

pushq S:　　%rsp-=8;(%rsp)=S;

popq D:　　D=(%rsp);%rsp+=8;

网上搜索得知popq的目标只能是寄存器

3.5 算数和逻辑操作

加载有效地址：leaq
除了leaq都有4种后缀
单操作数：op D
INC　　DEC　　NEG(取负)　　NOT(按位取反）
两操作数：op S,D 注意第二个操作数是目标且在运算符左边 D=D op S
ADD　　SUB　　IMUL　　XOR　　OR　　AND
SAL　　SHL　　SAR　　SHR
只有leaq不会设置条件码。
leaq和xor的特殊用法
除法没有列在P129的表中，在后面介绍。

3.5.3 移位操作

SHL SHR：逻辑左移右移
SAL SAR：算数左移右移
SAL和SHL等效（右边填0）
SHR在高位填0，SAR在高位填符号位。
移位量可以是立即数或%cl，忽略%cl的高位
只有一个操作数时默认移位量是1。

注意C++中移位量不在[0,位数-1]范围时程序行为是未定义的。
我测试时如果k为>=32的常数，x是unsigned int，C++编译时好像会将x<<k和x>>k直接当成0，但先将k赋值给整型变量n后x<<n和x>>n编译结果是以%cl存放移位量，也就是移位(n&31)位。
好像没办法用C++编译出shlq $100,%eax这样的指令，所以不知道会是什么结果。

3.5.5 乘除法与128位数

imulq/mulq S

idivq/divq S

单操作数，乘法计算%rax*S，除法计算(%rdx:%rax)/S和mod S

需要区分有无符号

cqto：符号扩展%rax到%rdx:%rax

连起来时%rdx存高位%rax存低位

除法结果商放在%rax余数放在%rdx

P134 GCC提供__int128 不知道考不考

P134 出现了idivl不知道是什么意思

3.6 控制

goto,if,while,switch

3.6.1 条件码

CPU维护一组单个位的条件码寄存器，描述了最近的算数或逻辑操作的属性。可检测这些寄存器来执行条件分支指令。最常用的条件码有：

CF：进位标志，在无符号意义下发生进位（包括减法借位）

ZF：零标志，得到结果为0

SF：符号标志，得到结果为负数

OF：溢出标志，补码意义下发生溢出（正或负）

P136 一些操作的行为：

xor将CF和OF设为0

移位将CF设为最后一个被移出的位，OF设为0

inc和dec设置ZF,OF但不改变CF

没写到的不知道考不考

3.6.1 条件码

CMP S1,S2和TEST S1,S2只设置条件码不更新目的寄存器
也有后缀b,w,l,q
CMP S1,S2：S2-S1
TEST S1,S2：S2&S1
典型用法testq %rax,%rax检查某个寄存器是正数/负数/0
也可以其中一个操作数是掩码

3.6.2 访问条件码

条件码通常不直接读取，常用使用方法有3种：
(1)根据条件码的组合设置一个字节为0或1；(2)条件跳转；(3)条件传送
SET指令的后缀表示条件码组合 12种：

```
e(ZF) ne(~ZF) s(SF) ns(~SF)
g(~(SF^OF)&~ZF) ge(~(SF^OF)) l(SF^OF) le((SF^OF)|ZF)
a(~CF&~ZF) ae(~CF) b(CF) be(CF|ZF)
```

注意到当cmp y,x后，SF^OF表示有符号x<y，CF表示无符号x<y
同义名：

```
z=e nz=ne
nle=g nl=ge nge=l ng=le
nbe=a nb=ae nae=b na=be
```

3.6.3 跳转指令

jmp 无条件跳转

jmp .L1 给出标号，汇编器确定带标号指令的地址并编码跳转目标

直接跳转：jmp Label跳转目标作为指令的一部分编码

间接跳转：jmp *Operand 从寄存器或内存读取跳转目标

条件跳转：jX Label 指令名后缀X和跳转条件与SET指令相匹配

条件跳转只能是直接跳转。

跳转指令的编码：P140

最常用的是PC相对的，以目标指令与跳转指令后一条指令的地址差为编码，偏移量可以为1,2,4字节，有符号

也可以4个字节给出绝对地址

3.6.5 条件分支

```
if(test_expr){
    then_statement;
}else{
    else_statement;
}
```

```
    t=test_expr;
    if(!t) goto False;
    then_statement;
    goto Done;
False:
    else_statement;
Done:
```

```
    t=test_expr;
    if(t) goto True;
    else_statement;
    goto Done;
True:
    then_statement;
Done:
```

练习题3.16 类似if(a&&b){...}生成的汇编代码可能包含多个条件跳转以实现逻辑短路。
练习题3.17 两个分支的前后顺序也可以交换，但对常见的没有else的情形，then在前更友好。

```
    t=test_expr;
    if(!t) goto Done;
    then_statement;
Done:
```

3.6.6 条件传送cmov

条件跳转可能非常低效（P146:分支预测错误的惩罚），在满足一些限制的情况下可以用条件传送指令代替。执行条件传送不需要预测测试结果。控制流不依赖于数据，处理器更容易保持流水线是满的。

```
if(test_expr)
    res=expr_then;
else
    res=expr_else;
return res;
```

```
rval=expr_then;
eval=expr_else;
t=test_expr;
if(t)rval=eval;
return rval;
```

cmov_ S R　S是源寄存器或内存地址，R是目的寄存器。

cmov的后缀与SET和条件跳转指令一样，表示传送的条件。

汇编器可以从目标寄存器的名字推断条件传送指令的操作数长度。

例如cmovl是当条件码表示小于时传送，而不是传送长字

cmov的源和目的只支持16位,32位,64位，不支持单字节。

3.6.6 cmov

无论测试结果，cmov需要对then_expr和else_expr都求值

因此有些条件表达式不能使用条件传送编译：

1.求值造成非法操作 x=p?*p:0;

2.求值造成副作用 x=t?(y+=1):(y-=1);

也可能降低代码效率 test?hard1():hard2();

```
int f(int n){
    return n>0?f(n-1):1;
}
```

```
int f(int x,int s){
    if(x<=0)return s;
    return (x&1)?f(x>>1,s+1):f(x>>1,s);
}
```

编译器并没有足够的信息来作出可靠的决定。

（不过写代码时可以先把两个值求好来提示编译器使用cmov？）

P148练习题3.20 一个应用cmov的有趣的例子

3.6.7 循环

C语言中的循环：do-while, while, for

```
//do-while
loop:
    body_statement
    t=test_expr;
    if(t)goto loop;
```

```
//while (jump to middle)
    goto Test;
Loop:
    body_statement
Test:
    t=test_expr;
    if(t)goto Loop;
```

```
//while (guarded-do)
    t=test_expr;
    if(!t)goto Done;
Loop:
    body_statement
    t=test_expr;
    if(t)goto Loop;
Done:
```

编译器有时会根据已知信息改变测试条件，如将>变为==

for循环

for循环可改写为while循环，注意存在continue语句时跳转目标是update_expr而不是循环体结尾。

```
for(init_expr;test_expr;update_expr){
    body_statement
}
```

```
init_expr;
while(test_expr){
    body_statement
    update_expr;
}
```

3.6.8 switch

switch能根据一个整数索引值进行多重分支。
通过使用跳转表使得实现高效。
当switch情况比较多且值的范围跨度比较小时就会使用跳转表。
jmp *.L4(,%rsi,8)
间接跳转到.L4+%rsi*8处存储的地址处（详见P161）
跳转表存在目标代码文件中的.rodata段中

Q&A

趁机提问：根据老师的描述，CPU的行为可以看作每个时钟周期取出PC指向的指令然后去完成，但是有很多指令需要的时间超过一个时钟周期，CPU怎么判断上一个指令有没有完成？

Thank you!