Machine-Level Programming II:Control

1 条件码

2 跳转指令

3 条件分支

4 循环

5 switch

条件码

CF	进位标志
ZF	零标志
SF	符号标志
OF	溢出标志

更改条件码的操作

- 各种算数和逻辑操作
- **不包含leaq 因为它本身不是被设计去算数的**

CF	无符号溢出
ZF	零
SF	负数
OF	有符号溢出

对于ADD操作设置条件码，计算机不知道二进制码被解释成有符号和无符号但是因为二者表示成位向量时操作一样，所以不区分的使用ADD操作，但是在设计条件码时就要考虑到不同情况分别判断是否溢出

条件码

更改条件码的操作

- 各种算数和逻辑操作
- CMP类指令和TEST类指令
- 只会改变条件码，不更新目的寄存器
- 在条件码的设置上CMP=SUB，TEST=AND
- CMP
  （1）利用符号标志和零标志判断大小（后面减前面）
- TEST
  （1）自己and自己，利用符号和零标志判断自己的符号
  （2）掩码测试

条件码

访问条件码

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF) &~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF)|ZF	Less or Equal (Signed)
seta	~CF&~ZF	Above (unsigned)
setb	CF	Below (unsigned)

- 操作对象只能是单字节的寄存器或内存，所以后缀不表示操作数大小而表示不同的条件
- 有符号和无符号比大小是不同的
- 实际操作中往往需要用movzbl高位清零

跳转方式

- 按照获得跳转目的地的方式分为 直接跳转 和 间接跳转
- 直接跳转：jump .Label
- 间接跳转：jump *(%rax)（区分jump *%rax）

跳转指令

按照是否需要满足条件有分为条件跳转和无条件跳转

条件跳转只能是直接跳转（用label的）

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

条件分支

计算机只能顺序执行命令，因此在遇到条件分支的时候需要跳转，而跳转的方式有两种：控制的转移和数据的转移，称它们为条件控制和条件传送

条件控制实现

高级语言中的条件分支：

```
if(t) A
else B
```

条件控制实现：

```
    jmpif(!t) .L1
    A
    jmp .L2
L1:
    B
L2:
```

(jmpif(!t)是指根据t找到相反的条件跳转汇编代码)

条件控制实现

高级语言中的条件分支：

```
if(t) A
else B
```

条件控制实现：

```
    jmpif(!t) .L1
    A
    jmp .L2
L1:
    B
L2:
```

(jmpif(!t)是指根据t找到相反的条件跳转汇编代码)

翻译的方式有很多种，比如可以把jumif(!t)改为jumif(t)并交换AB，或者把一个条件跳转拆成两个

条件传送实现

和条件跳转不同条件传送不会因为条件满足与否改变程序的运行顺序，因此cpu可以正确预测接下来该干什么从而利用流水线获得更高的性能。而条件跳转中的判断任务被交给了条件传送：即根据条件满足与否判断是否传送数据。

如cmove cmovne cmovs等

它们的形式都是cmov+代表条件的后缀（和前面的SET J 的后缀保持一致）相对原先的无条件传送，指令缺少了操作数长度的信息，目的只能是寄存器，程序通过目标寄存器名字判断传送长度

条件传送实现

但是相应的条件传送在一些场合不适合被应用：

（1）计算带来副作用

在程序员利用高级语言写代码时，他们认为的条件分支应该是只会执行其中then else 中一个的。因此如果其中一个的执行会对另一个需要的参数造成影响，或者是满足t的时候执行!t应该执行的操作会出问题，这些都是高级语言的书写者不会考虑到的，因为其实条件转移才是对我们直觉中的条件分支的完全还原，而条件传送只在某些时候能够产生等价的结果。

（2）可能会降低效率

如果then else 中的操作耗时太长，那么多执行一个带来的开销可能会超过流水线对处理的优化节约的时间。

循环

- do-while循环
- while循环
- for循环

循环

do-while循环

```
do
    A
    while(t)
```

利用条件跳转实现

```
.L1:
    A
    jmpif(t) .L1
```

循环

while循环

```
while(t)
    A
```

利用条件跳转实现（方法1）

```
    jmp .L2
.L1:
    A
.L2:
    jmpif(t) .L1
```

编译器有时会根据实际情况对汇编代码进行一些优化，比如判断某些条件一定满足或者不满足的时候

循环

while循环

```
while(t)
    A
```

利用条件跳转实现（方法2）

```
    jmpif(!t) .L2
.L1:
    A
    jmpif(t) .L1
.L2
```

循环

for循环

for(A;t;B)
    C
等价于这样的while循环
A
while(t)
    C
    B
然后利用前面的两种方式即可

在C中含有continue 的时候，不能直接跳到下一次循环，而应到C B之间

switch和if-else

switch和多重if-else 可以完成相同的任务，区别在于switch通过跳转表可以查询到相应的指令地址并通过间接跳转（jmp *%rax）转移控制，这个跳转需要的时间和情况的数目没有关系，而if-else则不同，而相应的switch 需要存储一个跳转表其中连续地记录了各种情况应该跳往的指令地址（因为switch的条件只能是整数因此连续是可以实现的）（顺序表-链表）

因此当我们在c中使用switch 时，编译器会根据情况是否足够多（时间优势是否明显），情况的稀疏程度（空间劣势是否明显）来判断使用switch还是多重if-else

Switch

switch

在汇编代码中，设case分别为a1<a2<a3...an，那么它会先把t减a1把合法区域平移到0～an-a1，然后所有无符号大于（ja .Li）an-a1的t都跳到default对应的指令中（为了处理t<a1的情况）

然后利用间接跳转到对应的指令，如 jmp .L4(,%rsi,8)

（这里的L4是在汇编代码中声明的跳转表在内存中存储位置的起始地址）

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

跳转表的设置

正常情况下只要按照case的大小顺序把相应代码块的地址连续得存放在跳转表中即可，然而考得就是特殊情况。

- case不连续

例如case 3 下来直接就到case 5，就在case 4 的位置上填default代码块所在的地址

- 没加break

在相应的代码块结尾不加jmp

- 不仅不加break 连代码也没有

case 4:

case 5:

那么跳转表中两个case会拥有相同的地址