第9讲
控制操作

吴海军
南京大学计算机科学与技术系

主要内容

- 条件

控制

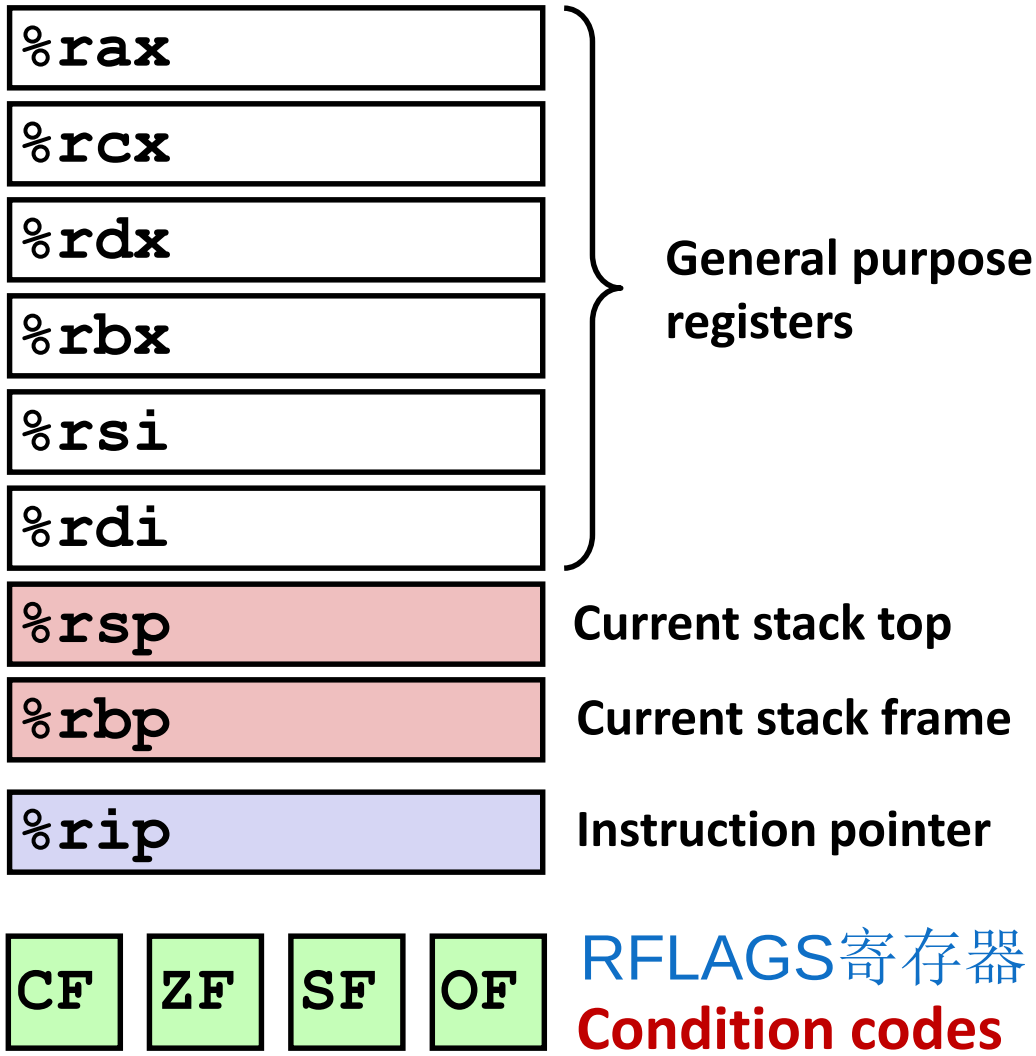
- C语言中的某些结构，比如条件语句、循环语句和分支语句，要求有条件的执行，根据数据测试的结果来决定操作执行的顺序。
- 实现有条件的行为： 测试数据值，然后根据测试的结果来改变控制流或者数据流。
- 用jump指令可以改变一组机器代码指令的执行顺序。

条件码

- CPU维护着一组单个位的条件码(condition code)寄存器，它们描述了最近的算术或逻辑操作的属性。可以检测这些寄存器来执行条件分支指令。
  - CF: 进位标志。 最近的操作使最高位产生了进位。 可用来检查无符号操作的溢出。
  - ZF: 零标志。 最近的操作得出的结果为0。
  - SF: 符号标志。 最近的操作得到的结果为负数。
  - OF: 溢出标志。 最近的操作导致一个补码溢出 正溢出或负溢出。

寄存器组保存处理器状态

- 记录当前运行程序状态
  - 暂存数据 (**%rax**, …)
  - 堆栈实时地址 (**%rbp**,**%rsp**)
  - 代码指针 (**%rip**, …)
  - 当前的测试状态，包含在RFLAGS寄存器中(**CF, ZF, SF, OF**)



条件标志位设置

- RFLAGS寄存器中的单独1位标志位，最常用有4个：

CF	进位标志	SF	符号标志
ZF	零标志	OF	溢出标志

- 由算术或逻辑操作隐式设置：**add** *Src,Dest* ↔ **t = a+b**
  - CF：(unsigned) t <(unsigned) a　　无符号数溢出
  - ZF：(t==0)　　零
  - SF：(t<0)　　负数
  - OF：(a<0==b<0)&&(t<0!=a<0)　　有符号溢出
- leaq指令：不改变任何条件码，只计算有效地址
- 逻辑运算：指令OF=CF=0；若结果全0，则ZF=1；若结果最高位为1，则SF=1。
- 移位运算：移出位送CF，OF=0。
- INC和DEC指令：设置溢出和零标志，不改变进位标志。

详细的标志位设置文档

条件标志位设置(续)

- CMP和TEST指令只设置条件码而不改变其他寄存器。
- CMP和SUB指令的行为一致，只是不改变目的寄存器的值。
- CMP *b*,*a* 相当于计算 t=a-b

无符号数a>b的判别条件是：CF=0 并且ZF=0

带符号数a>b的判别条件是：

1、没有溢出OF=0，则SF=0时，a>=b； SF=1时，a<b
2、有溢出OF=1，则SF=0(负溢出)，a<b；
SF=1(正溢出)时,a>b。
3、a=b时，ZF=1,OF=0,SF=0,没有溢出。

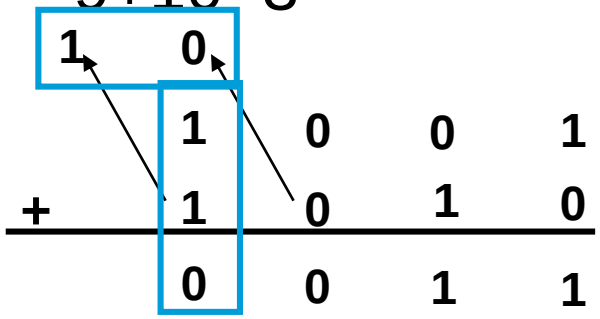
a>b 的条件 OF=SF &ZF=0，a<b的条件OF^SF

标志信息是干什么的？

Ex1: -7- 6 = -7 + (-6) = +3

9 - 6 = 3

9+10=3



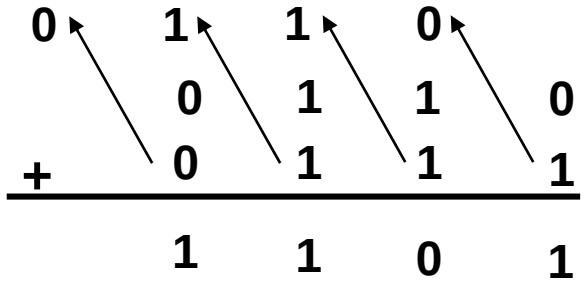
OF=1、ZF=0

SF=0、借位CF=0 ？

6 - (-7) = 6 + 7 = -3

6 - 9 = 13

6+7=13



OF=1、ZF=0

SF=1、借位CF=1？

同样的计算同样的标志，不同的含义；取决于表达式的含义

做减法以比较大小，规则：

Unsigned: CF=0且ZF=0时，大于

Signed：OF=SF且ZF=0时，大于

条件标志位设置(续)

- TEST指令和AND指令一样，但只设置条件码而不改变目的寄存器。
- 由Test指令显式设置：TEST b，a
  - 条件标志位的置位是根据b & a运算结果
  - ZF 当a&b == 0时置位1
  - SF 当a&b 最高位为1时置位1，看成a&b<0
- testq %rax,%rax用来检查%rax是负数、零、正数

访问条件码

- 条件码通常不会直接读取，使用方法：
  - 1)可以根据条件码的某种组合，将1个字节设置为 0或者 1，使用SET指令。
  - 2)可以条件跳转到程序的某个其他的部分
  - 3)可以有条件地传送数据。
- SET指令的目的操作数是低位单字节寄存器元素之一，或是一个字节的内存位置，指令会将这个字节设置成0或者1。
- setg(设置大于)和setnle(设置不小于等于)指的就是同一 条机器指令。编译器和反汇编器会随意决定使用哪个名字。

读取条件标志位

- SetX 指令:根据条件标志位的组合来设置一个字节的值为0或者1

指令	同名	效果	设置条件
sete D	setz	D←ZF	相等 / 零
setne D	setnz	D←~ZF	不等 / 非零
sets D		D←SF	负数
setns D		D←~SF	非负数
setg D	setnle	D←~(SF^OF)&~ZF	大于 (有符号>)
setge D	setnl	D←~(SF^OF)	大于等于 (有符号>=)
setl D	setnge	D←(SF^OF)	小于(有符号<)
setle D	setng	D←(SF^OF)|ZF	小于等于 (有符号<=)
seta D	setnbe	D←~CF&~ZF	超过 (无符号>)
setae D	setnb	D←~CF	超过或等于 (无符号>=)
setb D	setnae	D←CF	低于 (无符号<)
setbe D	setna	D←CF|ZF	低于或等于 (无符号<=)

指令后缀X不表示操作数的长度，而是条件标志位的组合。

读取条件标志位

- SetX 指令：根据条件标志位的组合来设置一个字节的值为0或者1
  - 目的操作数是低位个单字节寄存器之一或内存中单字节地址
    - 只修改整数寄存器的最低位1个字节，不修改高位3个字节
    - 经常用movzbl 指令完成对高位字节的清0。

```
c表达式 long a,b; a<b;
int comp(data_t a,data_t b)
a in %rdi, b in %rsi
```

comp:

```
cmpq %rsi,%rdi        #比较a-b      ←——— 注意次序反转!
setl %al              #%al=0
mozbl %al,%eax        #%eax=0,并且%rax高4个字节也清0
ret
```

虽然所有的算术和逻辑操作都会设置条件码， 但是各个SET命令的描述都适用的情况是： 执行CMP比较指令， 根据计算t=a-b设置条件码。

分支、跳转指令

控制转移指令

- 控制转移指令：终止指令的**顺序执行**，切换到程序中一个新的位置地址重新执行。目的地址可以用标号（label）指明。
  - 无条件跳转指令

    JMP DST：无条件转移到目标指令DST处执行
  - 有条件跳转指令

    Jcc DST：cc为条件码，根据标志（条件码）判断是否满足条件，若满足，则转移到目标指令DST处执行，否则按顺序执行
  - 条件设置

    SETcc DST：将条件码cc保存到DST（通常是一个8位寄存器 ）
  - 调用和返回指令 （用于过程调用）

    CALL DST：返回地址RA入栈，转DST处执行

    RET：从栈中取出返回地址RA，转到RA处执行
  - 中断指令

跳转指令

- jX 指令：根据条件码跳转到程序不同的部分

指令	同义指令	跳转条件	描述
jmp label		1	直接跳转
jmp *Operand		1	间接跳转
je label	jz	ZF	相等 / 零
jne label	jnz	~ZF	不等 / 非零
js label		SF	负数
jns label		~SF	非负数
jg label	jnle	~(SF^OF)&~ZF	大于 (有符号)
jge label	jnl	~(SF^OF)	大于等于 (有符号)
jl label	jnge	(SF^OF)	小于(有符号)
jle label	jng	(SF^OF)|ZF	小于等于(有符号)
ja label	jnbe	~CF&~ZF	超过(无符号)
jae label	jnb	~CF	超过或相等(无符号)
jb label	jnae	CF	低于(无符号)
jbe label	jna	CF|ZF	低于或相等(无符号)

分三类:

(1)根据单个标志的值转移

(2)按有符号整数比较转移

(3)按无符号整数比较转移

跳转指令编码

- 直接跳转：目的地址是作为指令的一部分编码。在汇编中直接给出一个标号作为跳转目标。
- 间接跳转：目的地址是从寄存器或存储器位置中读取的。使用“*”后接一个操作数指示符，如jmp *%eax, jmp *(%eax)
- 跳转指令中跳转地址的编码方法：
  - 相对地址：将目标指令的地址与*紧跟跳转指令后面的地址（PC计数器）*之间的偏移值(1、2、4个字节)作为编码。
  - 绝对地址：用4个字节直接指定目标地址
- 汇编器和链接器选择适当的跳转地址编码方法。

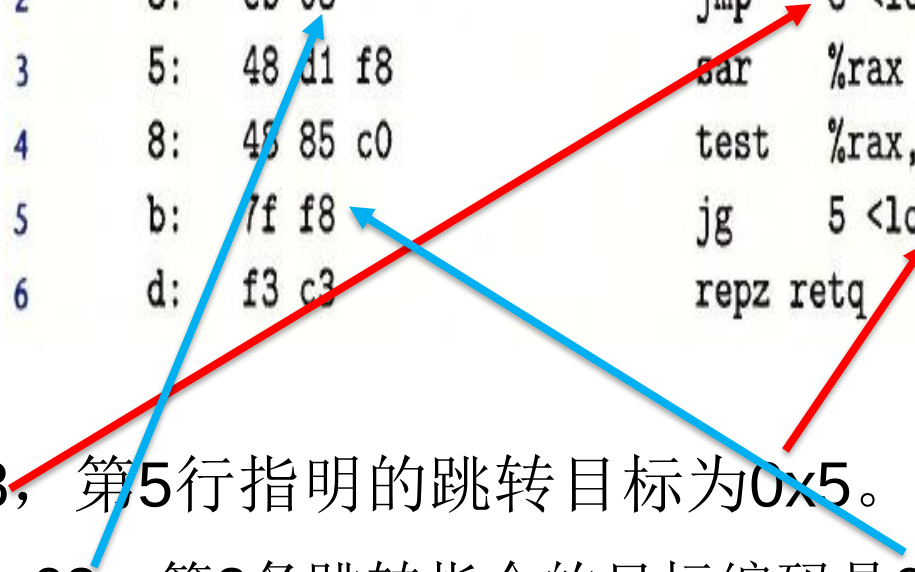
PC相对寻址

- 汇编程序与可重定位代码反汇编代码

```
1     movq    %rdi, %rax
2     jmp     .L2
3   .L3:
4     sarq    %rax
5   .L2:
6     testq   %rax, %rax
7     jg      .L3
8     rep; ret
```

```
1    0:   48 89 f8
2    3:   eb 03
3    5:   48 d1 f8
4    8:   48 85 c0
5    b:   7f f8
6    d:   f3 c3
```

```
mov    %rdi,%rax
jmp    8 <loop+0x8>
sar    %rax
test   %rax,%rax
jg     5 <loop+0x5>
repz retq
```



第2行指明的跳转目标为0x8，第5行指明的跳转目标为0x5。

第1条跳转指令的目标编码是0x03，第2条跳转指令的目标编码是0xf8?

执行第1条跳转指令时(PC)=0x05，0x05+0x03=0x08

执行第2条跳转指令时(PC)=0x0d，0x0d-0x08=0x05，$0xf8_{补}$=-8

PC相对寻址

- 指令被重定位不同地址后，跳转编码没有改变

```
1      4004d0:  48 89 f8          mov    %rdi,%rax
2      4004d3:  eb 03             jmp    4004d8 <loop+0x8>
3      4004d5:  48 d1 f8          sar    %rax
4      4004d8:  48 85 c0          test   %rax,%rax
5      4004db:  7f f8             jg     4004d5 <loop+0x5>
6      4004dd:  f3 c3             repz retq
```

下面 je 指令的目标是什么？

```
40042f: 74 f4                    je     XXXXXX
400431: 5d                       pop    %rbp
```

ja 和 pop 指令的地址是多少？

```
XXXXXX: 77 02                    ja     400547
XXXXXX: 5d                       pop    %rbp
```

用条件控制来实现条件分支

- 将条件表达式和语句从C语言翻译成机器代码，最常用的方式是结合有条件和无条件 跳转。
- 把汇编代码再转换成 C语言使用了的 goto语句。

用条件控制来实现条件分支

- 给出了一个计算两数之差绝对值的函数

```
long lt_cnt = 0;
long ge_cnt = 0;

long absdiff_se(long x, long y)
{
    long result;
    if (x < y) {
        lt_cnt++;
        result = y - x;
    }
    else {
        ge_cnt++;
        result = x - y;
    }
    return result;
}
```

```
    long absdiff_se(long x, long y)
    x in %rdi, y in %rsi
1   absdiff_se:
2     cmpq    %rsi, %rdi          Compare x:y
3     jge     .L2                 If >= goto x_ge_y
4     addq    $1, lt_cnt(%rip)    lt_cnt++
5     movq    %rsi, %rax
6     subq    %rdi, %rax          result = y - x
7     ret                         Return
8   .L2:                        x_ge_y:
9     addq    $1, ge_cnt(%rip)    ge_cnt++
10    movq    %rdi, %rax
11    subq    %rsi, %rax          result = x - y
12    ret                         Return
```

用条件控制来实现条件分支

- 使用 goto 语句， 是为了构造描述汇编代码程序控制流的 C 程 序。

```
    long absdiff_se(long x, long y)
    x in %rdi, y in %rsi
1   absdiff_se:
2     cmpq    %rsi, %rdi          Compare x:y
3     jge     .L2                 If >= goto x_ge_y
4     addq    $1, lt_cnt(%rip)    lt_cnt++
5     movq    %rsi, %rax
6     subq    %rdi, %rax          result = y - x
7     ret                         Return
8   .L2:                        x_ge_y:
9     addq    $1, ge_cnt(%rip)    ge_cnt++
10    movq    %rdi, %rax
11    subq    %rsi, %rax          result = x - y
12    ret                         Return
```

```
1   long gotodiff_se(long x, long y)
2   {
3       long result;
4       if (x >= y)
5           goto x_ge_y;
6       lt_cnt++;
7       result =  y - x;
8       return result;
9    x_ge_y:
10      ge_cnt++;
11      result = x - y;
12      return result;
13  }
```

一般条件表达式的转换

- C语言中的条件语言通用模板
- 汇编器为 then-statement 和 else-statement 产生各自的代码块。它会插入条件和无条件分支，以保证能执行正确的代码块。

```
if (test-expr)
     then-statement
else
    else-statemen
```

```
t = test-expr;
if (!t)
        goto false;
        then-statement
        goto done;
false:
         else-statement
done:
```

汇编实现形式

用条件传送来实现条件分支

- 传统方法实现条件操作是通过使用控制的条件转移。当条件满足时，程序沿着一条执行路径执行，而当条件不满足时，就走另一条。
- 一种替代的策略是使用数据的条件转移。这种方法计算一个条件操作的两种结果，然后再根据条件是否满足从中选取一个。
- 条件传送指令更符合现代处理器的性能特性。
- 不是所有的条件表达式都可以用条件传送来编译。

条件传送指令

- 这个函数计算参数x和y差的绝对值

```
long absdiff(long x, long y)
{
    long result;
   if (x < y)
       result = y - x;
   else
       result = x - y;
   return result;
}
```

```
long cmovdiff(long x, long y)
{
   long rval = y-x;
   long eval = x-y;
   long ntest = x >= y;
   /* Line below requires
        single instruction: */
    if (ntest)   rval = eval;
    return rval;
}
```

它既计算了y-x,也计算了x-y, 分别命名为rval和eval。然后它再测试x是否大于等千y, 如果是，就在函数返回rval 前，将eval复制到rval中。

条件传送指令

- 汇编代码

```
long absdiff(long x, long y)
 x in %rdi , y in %rsi
absdiff:
   movq      %rsi, %rax
   subq       %rdi, %rax
   movq       %rdi, %rdx
   subq       %rsi, %rdx
   cmpq       %rsi, %rdi
   cmovge     %rdx, %rax
   ret
```

rval = y-x

eval = x-y

Compare x:y

If >=, rval = eval

Return rval

条件数据传送代码

- 基于条件数据传送的代码会比基于条件控制转移的代码性能要好。
  - 处理器通过使用流水线(pipelining)来获得高性能
  - 要求能够事先确定要执行的指令序列
  - 当遇到条件跳转时，只有当分支条件求值完成之后，才能决定分支往哪边走。
  - 错误预测一个跳转，要求处理器丢掉它为该跳转指令后所有指令已做的工作，然后再开始用从正确位置处起始的指令去填充流水线。

基于条件控制转移的代码需要8到27个时钟周期，
使用条件传送的代码所需的时间都是大约8个时钟周期。

条件传送指令

指令	同义名	传送条件	描述
cmove S, R	cmovz	ZF	相等 / 零
cmovne S, R	cmovnz	~ZF	不相等 / 非零
cmovs S, R		SF	负数
cmovns S, R		~SF	非负数
cmovg S, R	cmovnle	~(SF ^ OF) & ~ZF	大于（有符号 >）
cmovge S, R	cmovnl	~(SF ^ OF)	大于或等 于（有符号 > =）
cmovl S, R	cmovnge	SF ^ OF	小于（有符号 <）
cmovle S, R	cmovng	(SF ^ OF) | ZF	小于或等 于（有符号 < =）
cmova S, R	cmovnbe	~CF & ~ZF	超过（无符号 >）
cmovae S, R	cmovnb	~CF	超过或相等（无符号 > =）
cmovb S, R	cmovnae	CF	低于（无符号 <）
cmovbe S, R	cmovna	CF | ZF	低于或相等（无符号 < =）

当条件满足时，指令将S值复制到R中。

条件传送的代码

- 用条件控制转移的标准方法

v = test-expr ? then-expr : else-expr;

```
if (! test-expr)
    goto false;
  y = then-expr;
    goto done;
false:
    v = else-expr;
done:
```

基于条件传送的代码， 会对 then-expr 和 else-expr 都求值， 最终值的选择基于对 testexpr 的求值。 可以用下而的抽象代码描述：

```
v = then-expr;
ve = else-expr;
t = test-expr;
if (!t) v = ve;
```

条件传送的代码

- 不是所有的条件表达式都可以用条件传送来编译。
- 抽象代码会对then-expr 和 else-expr都求值。 如果这两个表达式中的任意一个可能产生错误条件或者副作用， 就会导致非法的行为

```
long cread(long *xp)
 {
    return (xp? *xp : 0)
}
```

```
long cread(long *Xp)
Invalid implementation of function cread
xp in register %rdi
cread:
      movq    (%rdi), %rax
      testq   %rdi, %rdi
      movl    $0, %edx
      cmove   % rdx, %rax
ret
```

movq 指令对xp的间接引用
可能导致一个间接引用空指针的错误。 所以，必须用分支代码来编译这段代码。

条件转移语句中不良应用

分支语句计算的复杂性

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations are very simple

分支语句计算的风险性

```
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

分支语句计算影响条件判断

```
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- Must be side-effect free