MACHINE PROG:CONTROL

程序的机器级表示：控制

- 条件码
  - 条件码的设置
  - 条件码的使用
- 跳转指令及其编码
- 条件分支的两种实现
  - 条件控制
  - 条件传送
- Switch语句
- 循环
  - 三种常用的循环：do-while循环,while循环,for循环

回顾：x86-64机器代码中有些在C语言中对程序员**不可见**的状态变为**可见**

- 程序计数器：通常称“PC”，%rip，下一条指令的地址
- 整数寄存器：16个有命名的位置，分别存储64位值（可以是地址或整数）
- **条件码寄存器：保存最近执行的运算的结果之状态信息，可用于if条件分支、while循环等**
- 向量寄存器：存放一个或多个整数、浮点数值

条件码都只有1个bit

描述的是 **最近** 的算数或逻辑操作的属性

如何设置条件码 ：每个条件码都有一个判断条件（教材P135），对最近的算数或逻辑运算的结果进行判断，当成立时置为1

图：x64官方文档中对条件码的描述

只需要掌握课本上提到的CF、ZF、SF、OF

CF (bit 0)	**Carry flag** — Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared otherwise. This flag indicates an overflow condition for unsigned-integer arithmetic. It is also used in multiple-precision arithmetic.
PF (bit 2)	**Parity flag** — Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise.
AF (bit 4)	**Auxiliary Carry flag** — Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic.
ZF (bit 6)	**Zero flag** — Set if the result is zero; cleared otherwise.
SF (bit 7)	**Sign flag** — Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.)
OF (bit 11)	**Overflow flag** — Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two's complement) arithmetic.

4个需要掌握的条件码

- CF：进位标志，最近执行的操作中最高位发生进位时置CF为1，可用于检测无符号溢出
  - Carry表示进位
- ZF：零标志，最近执行的操作结果为0时置ZF为1
  - Zero表示0
- SF：符号标志，最近执行的操作结果为负数时置SF为1
  - Sign表示符号
- OF：溢出标志，最近的操作中出现了补码溢出（正或负溢出）时置OF为1
  - Overflow表示溢出

- 个人推荐记忆方式：ZF、SF较好记忆，对OF与CF，记住一般讨论最高位进位时，讨论的数是无符号数（补码数最高位是符号位，而无符号数最高位表示2的w-1次方，w是位数），若出现补码数相加时最高位是两个1，则只可能产生负溢出，而不会说发生了进位

条件码的设置

- Implicit 隐式设置条件码
  - 更新寄存器或内存中的值，并设置根据结果条件码，包括各种整数算术运算：

指令名	效果	指令名	效果
INC	自加1	XOR	两数亦或
DEC	自减1	OR	两数或
NEG	取负（非）	AND	两数且
NOT	取补	SAL	左移
ADD	两数相加	SHL	左移
SUB	两数相减	SAR	算术右移
IMUL	两数相乘	SHR	逻辑右移

硬件中条件码的计算方式

- OF：最高位的进位^次高位的进位
  - 仅对补码加减法有意义
- SF：最高位的数字之和,包括进位
- ZF：仅当两数的运算结果中所有位全为0时才置为1
- CF：最高位产生的进位^sub，做减法时sub为1，加法时sub为0
  - 仅对无符号数加减法有意义

进位是指某个位上出现1+1，或是1+0的同时再加上从次一位进位产生的1

例如：假设用8位表示，11111111+00000001、10001010+10000000都会在最高位发生进位，但前者会在次高位发生进位，后者不会

设置条件码的一些细节

- 如果无符号数减法发生了借位，也会设置CF为1（类似于负溢出，但不是）
- leaq指令不改变任何条件码(只对地址进行操作)
- 逻辑操作会把CF和OF设置成0（逻辑运算结果不会超过表示范围，也就不会溢出）
- 移位操作会把CF设置为最一个被移出的位，OF设置为0
- INC和DEC指令会设置OF和ZF，但是不会改变CF
- -1-TMIN的行为0xF..F - 0x10..0

- **思考：无符号数做加减法，会设置OF标志位吗?**
  - 会，因为标志位的设置只与操作数的位级操作有关，回忆第二章，对机器而言，无符号数与补码数在计算时遵循同样的规则，机器并不会区分操作数的类型，只是对每一个位上的0、1进行操作

条件码的设置

- Explicit 显式设置条件码
  - 只更新条件码，而不改变寄存器
- 两个重要指令：CMPX，TESTX，X可以是b(字节),w(字),l(2字),q(4字)
- CMP
  - 除了只设置条件码而不改变目的寄存器之外，CMP指令与SUB指令的行为是一样的
  - 注意 cmpq S1, S2 是取 S2-S1 用来设置条件码（ATT格式）
- TEST
  - 除了只设置条件码而不改变目的寄存器之外，TEST指令与AND指令的行为是一样的

两种典型用法

1. 两个操作数一样，如 testq %rax, %rax

看 %rax 中的数和 0 的关系。e.g. 上一句话后面跟 setg %al ，就表明当 %rax 值大于0时把 %al 的值设置为 1 ，否则设成 0

2. 其中的一个操作数是一个掩码

与另一个操作数与完了之后，得到的结果就是让另一个操作数只剩下你想保留的那些位所得到的值，之后就可以对这些特定的位进行判断了

练习题

2、条件码描述了最近一次算术或逻辑操作的属性。下列关于条件码的叙述中，哪一个是不正确的？（ C ）

A. set 指令可以根据条件码的组合将一个字节设置为 0 或 1

B. cmp 指令和 test 指令可以设置条件码但不更改目的寄存器

C. leaq 指令可以设置条件码 CF 和 OF

D. 除无条件跳转指令 jmp 外，其他跳转指令都是根据条件码的某种组合跳转到标号指示的位置

条件码的使用

- set、jxx、cmov等指令都会使用到条件码
- set指令：
  - 以一个字节为目标，将其设置为0或1，这个字节可以是某寄存器的低位单字节，也可以是占1字节的内存
  - set指令会有各种后缀，表示其用于判断的条件，如sets，意为当SF为1时，把本sets指令的目标字节设为1，否则设为0
- 格式：setx D，x可以是表中的各后缀，D是目标内存或目标寄存器

（附表）

注：后缀g、l、a、b的区别：g和l对应补码数的大、小，即greater、less；a和b对应无符号数的大、小，即above、below。因为无符号数只需考虑两数的位级表示谁更大，而补码数需要考虑符号位才能确定大小，所以无符号数就用above(在…之上)，而补码数则用表示大小的greater与less

后缀	同义后缀	结果	后缀含义
e	z	ZF	equal/zero
ne	nz	~ZF	not equal/not zero
s		SF	sign为1，标志负数
ns		~SF	sign不为1，标志非负数
g	nle	~(SF^OF)&~ZF	greater/not less or equal
ge	nl	~(SF^OF)	greater or equal/not less
l	nge	SF^OF	less/not greater or equal
le	ng	(SF^OF)|ZF	less or equal/not greater
a	nbe	~CF&~ZF	above/not below or equal
ae	nb	~CF	above or equal/not below
b	nae	CF	below/not above or equal
be	na	CF|ZF	below or equal/not above

跳转指令及其编码

- 跳转指令：用于在一定条件下将当前命令执行的过程跳转至指定目标，包括两类：直接跳转与间接跳转
- 格式：jx *Label*(直接)或jx **Operand*(间接)，x可以是表中的各后缀，还包括mp这一特殊后缀(跳转指令特有)，即jmp指令，Label是目的指令标号，*Operand是目的指令地址
- jmp指令无需条件判断即可跳转，而其余所有指令都只有在条件码符合其后缀对应的条件时才跳转，比如jl指令只有在SF^OF为1时才跳转

跳转指令及其编码

- 编码
  - PC 相对的 (PC-relative)
    - 将目标指令的地址与**紧跟在跳转指令后面的那条指令**的地址之间的差作为编码
    - 当执行PC相对寻址时，程序计数器的值是跳转指令后面的那条指令的地址
    - 地址偏移量可以编码为1、2、4个字节
      - 偏移量超过4字节(4GB)？使用直接跳转，先计算出目标指令地址，存入寄存器中，再直接跳转即可
  - 绝对地址
    - 用4个字节直接指定目标

- 使用相对地址的一些好处
  - 指令编码更简洁（很多情况下只需要1、2个字节指代地址）
  - 代码可以不做任何改变就移到内存中不同的位置

跳转指令及其编码

举例：

```
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
```

红色标注的d8为目标指令的地址，d5为下一条指令的地址，d8-d5=03是跳转指令的编码

黄色标注的d5为目标指令的地址，dd为下一条指令的地址，d5-dd=f8是跳转指令的编码，注意这里f8意为-8

练习题

3. 在如下代码段的跳转指令中，目的地址是：**B**

```
400020: 74 F0     je _________
400022: 5d     pop %rbp
```

A. 400010　　B. 400012　　C. 400110　　D. 400112

条件控制

- if语句的两种实现，其一，根据条件决定选择执行哪些代码
- C语言中的通用形式：

```
if(test_expr)
    then_statement
else
    else_statement
```

- 汇编实现的C形式：

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

条件控制

- 对应的汇编代码

```
function:
    (initialize)          初始化
    jx        Label       根据x对应的条件决定是否跳转至Label所对应的代码段
    (else-statement)      条件不满足时执行的代码
      ret                 结束
Label:
    (then-statement)      条件满足时执行的代码
    ret                   结束
```

条件传送

- if语句的两种实现，其二，先计算一个条件满足与不满足这两种情况的结果，再根据条件决定选取哪个
- 例如计算两数之差绝对值的代码：

```
long cmovdiff(long x,long y)
{
    long rval=y-x;
    long eval=x-y;
    long ntest=x>=y;
    if(ntest)ravl=eval;
    return rval;
}
```

条件传送

- 在汇编代码中，需要一个新的指令：cmovx，x是前面的表中的后缀
- 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
- 格式：cmovx *S，R* 其中S是源(寄存器或内存地址)，R是目标寄存器
- 在汇编代码中，先计算出两种情况下的值，分别临时放在寄存器中，最后根据后缀对应的条件决定是否用另一寄存器的值来覆盖返回值所在的寄存器，即%rax
- 例如：*x in %rdi,y in %rsi*

```
absdiff:
    movq    %rsi,%rax
    subq    %rdi,%rax       rval=y-x
    movq    %rdi,%rdx
    subq    %rsi,%rdx       eval=x-y
    cmpq    %rsi,%rdi       compare x:y
    cmovge  %rdx,%rax       if x>=y,rval=eval
    ret                     return rval
```

条件传送

- 为什么有些情况下条件传送优于条件控制？
- 简而言之，每条指令的执行分很多步，以流水线方式执行。但处理器会以一定逻辑猜测跳转指令的走向，以使得流水线中充满待执行的指令。一旦猜测错误，关于该分支所做的工作就付诸东流，只能从正确位置开始执行正确的指令。那些被浪费的指令工作的时间开销是不可忽略的，那么当计算开销较小时，就可以用条件传送来代替条件控制，以此避免除计算指令之外的可能的无用开销。

cmov可能带来的问题

当使用条件传送时，不可避免地需要用到cmov指令，在以下3种情况使用cmov会带来问题

1. 表达式的求值需要大量的运算，此时时间开销较大
2. 表达式的求值可能导致错误

   例：val = p ? *p : 0;

3. 表达式的求值可能导致不期望产生的副作用

   例：val = x > 0 ? x*=7 : x+=3;

就像if语句的条件控制与条件传送，三目表达式 a ? b : c可以有两种形式的代码，a作为条件，b作为*then-statement*，c作为*else-statement*

练习题

9．对于下列四个函数，假设 gcc 开了编译优化，判断 gcc 是否会将其编译为条件传送。

long f1(long a, long b) {
  return (++a > --b) ? a : b;
}
	Y N

long f2(long *a, long *b) {
  return (*a > *b) ? --(*a) : (*b)--;
}
	Y N

long f3(long *a, long *b) {
  return a ? *a : (b ? *b : 0);
}
	Y N

long f4(long a, long b) {
  return (a > b) ? a++ : ++b;
}
	Y N

练习题

9．对于下列四个函数，假设 gcc 开了编译优化，判断 gcc 是否会将其编译为条件传送。

`long f1(long a, long b) {`
`  return (++a > --b) ? a : b;`
`}`
	(Y) N

`long f2(long *a, long *b) {`
`  return (*a > *b) ? --(*a) : (*b)--;`
`}`
	Y (N)

`long f3(long *a, long *b) {`
`  return a ? *a : (b ? *b : 0);`
`}`
	Y (N)

`long f4(long a, long b) {`
`  return (a > b) ? a++ : ++b;`
`}`
	(Y) N

【答】f1 由于比较前计算出的 a 与 b 就是条件传送的目标，因此会被编译成条件传送；f2 由于比较结果会导致 a 与 b 指向的元素发生不同的改变，因此会被编译成条件跳转；f3 由于指针 a 可能无效，因此会被编译为条件跳转；f4 会被编译成条件传送，注意到 a 和 b 都是局部变量，return 的时候对 a 和 b 的操作都是没有用的。使用-O1 可以验证 gcc 的行为。

switch语句

- 根据一个整数索引值进行多重分支
- 需要用到跳转表，形如：

```
.L4
    .quad       .L3
    .quad       .L2
    .quad       .L5
    .quad       .L2
    .quad       .L6
    .quad       .L7
    .quad       .L2
    .quad       .L5
```

switch语句

```
switcher:
  ……
  jmp              *.L4(,%xxx,8)
  .section                  .rodata
  .L7:
    ……
  .L3:
    ……
  .L5:
    ……
  .L2:
    ……
  .L6:
    ……
```

表示根据寄存器%xxx中的值取.L4中的对应标号，进行跳转

比如%xxx中为3，那么取上表中的第三个标号，跳转至.L5

*.L4(,%xxx,8)表示取 与指定的标号相关联的指令地址

switch语句

- 在C代码中每个case下的代码段末尾应有break，对应在汇编代码中则是jmp指令跳转至末尾处，结束该switch语段，若没有jmp，则意味着会顺延至下一标号对应的代码中继续执行。
- 例如：

```
.L7:
  xorq      $15,%rsi
  movq      %rsi,rdx          向下顺延
.L3:
  leaq      112(%rdx),%rdi
  jmp       .L6               跳转至末尾，结束
.L5
  leaq      (%rdx,%rsi),%rdi
  salq      $2,%rdi
  jmp       .L6               跳转至末尾，结束
.L2
  movq      %rsi,%rdi         向下顺延
.L6:
  movq      %rdi,(%rcx)
  ret
```

三种循环 do-while循环,while循环,for循环

- do-while循环的C语言形式：

```
do
    body-statement
    while(test-expr)
```

- goto语句：

```
loop:
    body-statement
    t=test-expr;
    if (t)
       goto loop;
```

do-while循环

- 以阶乘为例

```
fact_do:                        n in %rdi
  movl        $1,%eax
.L2:
  imulq       %rdi,%rax         执行循环
  subq        $1,%rdi
  cmpq        $1,%rdi
  jg          .L2               判断循环是否继续
  rep;ret
```

while循环

- while的两种实现，其一(jump to middle)：

```
    goto test;
loop:
    body-statement
test:
    t=test-expr;
    if (t)
        goto loop;
```

- 汇编代码(仍以阶乘为例)

```
fact_while:
  movl          $1,%eax
  jmp           .L5                 先执行测试
.L6:
  imulq         %rdi,%rax           执行循环
  subq          $1,%rdi
.L5:
  cmpq          $1,%rdi
  jg            .L6                 判断循环是否继续
  rep;ret
```

while循环

- while的两种实现，其二(guarded-do)：

```
t=test-expr;
if (! t) goto done;
loop:
    body-statement
    t=test-expr;
    if(t)
        goto loop;
done:
```

- 汇编形式：

```
fact_while:
  cmpq $1,%rdi
  jle  .L7                    初始判断，若不符合则直接跳转到结束语句
  movl $1,%eax
.L6:
  imulq %rdi,%rax             执行循环
  subq  $1,%rdi
  cmpq  $1,%rdi
  jne   .L6                   判断循环是否继续
  rep;ret
.L7
  movl $1,%eax          结束
  ret
```

for循环

- for循环的C语言形式：

```
for(init-expr;test-expr;update-expr)
    body-statement
```

- 可以按如下形式转换为while循环

```
init-expr;
while(test-expr){
    body-statement
    update-expr;
}
```

- 因此也可以用jump to middle、guarded-do这两种汇编代码来对应，但是有一个小例外

for循环

- 当循环体中出现continue语句时，单纯地把for循环转换为while循环就会出现问题
- 例如：

```
long sum=0;
long i;
for(i=0;i<10;i++){
   if(i&1)
     continue;
   sum+=i;
}
```

- 这个例子若直接写成while循环,会导致死循环，因为continue语句跳过了update语句

```
while(i<10){
     if(i&1)
       continue;
     sum+=i;
     i++;
}
```

- 此时可以通过把continue语句换为goto语句，goto的目标是update语句，来解决问题，汇编代码也是同理