



第3章 程序的机器级表示

Machine-Level Programming II: Control

100076202: 计算机系统导论

II: 控制

II: Control



任课教师:

宿红毅 张艳 黎有琦 李秀星

原作者:

Randal E. Bryant and David R. O'Hallaron

**Carnegie
Mellon
University**



回忆：内存操作数和LEA

Recall: Memory operands and LEA

在大多数指令中，内存操作数访问内存 In most instructions, a memory operand accesses memory

汇编 Assembly	等价C语言 C equivalent
<code>mov 6(%rbx,%rdi,8), %ax</code>	<code>ax = *(rbx + rdi*8 + 6)</code>
<code>add 6(%rbx,%rdi,8), %ax</code>	<code>ax += *(rbx + rdi*8 + 6)</code>
<code>xor %ax, 6(%rbx,%rdi,8)</code>	<code>*(rbx + rdi*8 + 6) ^= ax</code>

LEA特殊性：它不访问内存 LEA is special: it *doesn't* access memory

汇编 Assembly	等价C语言 C equivalent
<code>lea 6(%rbx,%rdi,8), %rax</code>	<code>rax = rbx + rdi*8 + 6</code>

为何使用LEA Why use LEA?



- CPU设计师倾向使用：计算一个对象的指针 CPU designers' intended use: calculate a pointer to an object
 - 数组元素，或许 An array element, perhaps
 - 例如 传递一个数组元素给另一个函数 For instance, to pass just one array element to another function

汇编 Assembly

```
lea (%rbx,%rdi,8), %rax
```

等价C语言 C equivalent

```
rax = &rbx[rdi]
```

- 编译器设计人员喜欢用它实现普通计算 Compiler authors like to use it for ordinary arithmetic
 - 可以在一条指令中做复杂的计算 It can do complex calculations in one instruction
 - x86仅有的几个三操作数指令之一 It's one of the only three-operand instructions the x86 has
 - 并不影响条件码（我们后面再讨论） It doesn't touch the condition codes (we'll come back to this)

汇编 Assembly

```
lea (%rbx,%rbx,2), %rax
```

等价C语言 C equivalent

```
rax = rbx * 3
```

旁注：指令后缀



Sidebar: instruction suffixes

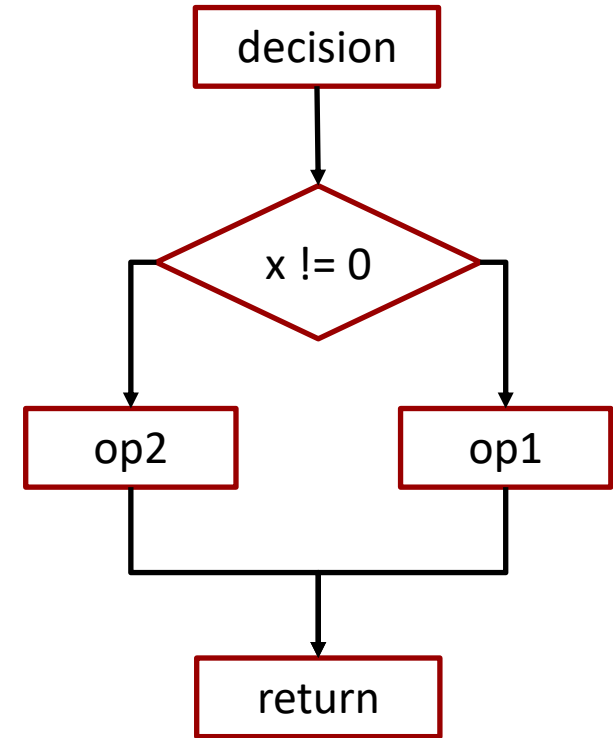
- 多数x86指令可以写或不写后缀 Most x86 instructions can be written with or without a suffix
 - `imul %rcx, %rax`
 - `imulq %rcx, %rax`
- 没有区别**
There's no difference!
- 后缀指明操作的大小 The suffix indicates the operation size
 - b=byte, w=short, l=int, q=long
 - 如果出现，必须和寄存器名字相匹配 If present, must match register names
 - 编译器产生的汇编输出(gcc -S)通常有后缀 Assembly output from the compiler (gcc -S) usually has suffixes
 - 反汇编转储通常省略后缀 Disassembly dumps (objdump -d, gdb 'disas') usually omit suffixes
 - Intel手册总是省略后缀 Intel's manuals always omit the suffixes





控制流 Control flow

```
extern void op1(void);  
extern void op2(void);  
  
void decision(int x) {  
    if (x) {  
        op1();  
    } else {  
        op2();  
    }  
}
```



汇编语言的控制流

Control flow in assembly language



```
extern void op1(void);  
extern void op2(void);
```

```
void decision(int x) {  
    if (x) {  
        op1();  
    } else {  
        op2();  
    }  
}
```

```
decision:  
    subq    $8, %rsp  
    testl   %edi, %edi  
    je      .L2  
    call    op1  
    jmp     .L1  
    .L2:  
    call    op2  
    .L1:  
    addq    $8, %rsp  
    ret
```



汇编语言的控制流

Control flow in assembly language

```
extern void op1(void);  
extern void op2(void);
```

```
void decision(int x) {  
    if (x) {  
        op1();  
    } else {  
        op2();  
    }  
}
```

decision:

```
subq    $8, %rsp  
testl   %edi, %edi  
je       .L2  
call     op1  
jmp      .L1  
  
.L2:  
call     op2  
  
.L1:  
addq    $8, %rsp  
ret
```

用GOTO语句来实
现 It's all done with
GOTO!



议题

- **控制：条件码** Control: Condition codes
- **条件分支** Conditional branches
- **循环** Loops
- **Switch语句** Switch Statements



处理器状态 (x86-64, 部分)

Processor State (x86-64, Partial)

■ 关于当前执行程序的信息

Information about currently executing program

- 临时数据 Temporary data
(`%rax`, ...)
- 运行时栈位置 Location of runtime stack
(`%rsp`)
- 当前代码控制点位置 Location of current code control point
(`%rip`, ...)
- 最近测试的状态 Status of recent tests
(`CF`, `ZF`, `SF`, `OF`)

当前栈顶
Current stack top

寄存器 Registers

<code>%rax</code>	<code>%r8</code>
<code>%rbx</code>	<code>%r9</code>
<code>%rcx</code>	<code>%r10</code>
<code>%rdx</code>	<code>%r11</code>
<code>%rsi</code>	<code>%r12</code>
<code>%rdi</code>	<code>%r13</code>
<code>%rsp</code>	<code>%r14</code>
<code>%rbp</code>	<code>%r15</code>

`%rip`

指令指针
Instruction pointer

<code>CF</code>	<code>ZF</code>	<code>SF</code>	<code>OF</code>
-----------------	-----------------	-----------------	-----------------

条件码
Condition codes

在授课过程中需记住什么

What to remember during lecture



设置条件码

Set Condition Codes

- 操作: 例如 `addq`
Operations: e.g. `addq`
- 比较: Compare: `cmp a, b`
类似做 `b-a` like doing `b-a`
- 测试: Test: `test a, b`
类似做 `a&b` like doing `a&b`

根据条件码跳转: `je` (相等跳转) `jg` (大于跳转) 等

Jump based on condition

codes: `je` (jump if equal), `jg` (greater), etc.

根据条件码设置寄存器的低字节为0/1 Set low order byte of a register to 0/1 based on condition codes

如果条件码置位则传送一个值 `mov a, value` if a condition code is set

我们将深入研究, 但是请像做炸弹实验一样阅读
We'll dive in, but read as you do bomb lab!



条件码（隐式设置）

Condition Codes (Implicit Setting)

■ 单个比特寄存器 Single bit registers

- **CF** 进位标志 Carry Flag (对无符号数 for unsigned) **SF** 符号标志 Sign Flag (对有符号数 for signed)
- **ZF** 零标志 Zero Flag **OF** 溢出标志 Overflow Flag (对有符号数 for signed)

■ 由算术运算隐式设置（看成副作用） Implicitly set (think of it as side effect) by arithmetic operations

举例： Example: `addq Src, Dest` \leftrightarrow `t = a+b`

CF set 如果从最高有效位进位（无符号溢出） if carry out from most significant bit (unsigned overflow)

ZF set 如果结果为零 if `t == 0`

SF set 如果结果小于零（有符号数） if `t < 0` (as signed)

OF set 如果补码（有符号数）溢出 if two's-complement (signed) overflow
(`a>0 && b>0 && t<0`) || (`a<0 && b<0 && t>=0`)

■ `leaq`指令不设置条件码 Not set by `leaq` instruction



ZF set when

000000000000...000000000000



SF set when

$$\begin{array}{r} \boxed{\text{yxxxxxxxxxxxxxxxxx} \dots} \\ + \boxed{\text{yxxxxxxxxxxxxxxxxx} \dots} \\ \hline \boxed{\text{1xxxxxxxxxxxxxxxxx} \dots} \end{array}$$

对于有符号数计算，该标志报告结果为负数

For signed arithmetic, this reports when result is a negative number



CF set when

$$\begin{array}{r} \text{+} \quad \begin{array}{|l|} \hline 1\text{xxxxxxxxxxxxxxxxx} \dots \\ \hline 1\text{xxxxxxxxxxxxxxxxx} \dots \\ \hline \end{array} \\ \hline \text{1} \quad \begin{array}{|l|} \hline \text{xxxxxxxxxxxxxxxxx} \dots \\ \hline \end{array} \end{array}$$

$$\begin{array}{r} \text{1} \quad \begin{array}{|l|} \hline 0\text{xxxxxxxxxxxxxxxxx} \dots \\ \hline 1\text{xxxxxxxxxxxxxxxxx} \dots \\ \hline \end{array} \\ \text{-} \quad \begin{array}{|l|} \hline 1\text{xxxxxxxxxxxxxxxxx} \dots \\ \hline \end{array} \\ \hline \begin{array}{|l|} \hline 1\text{xxxxxxxxxxxxxxxxx} \dots \\ \hline \end{array} \end{array}$$

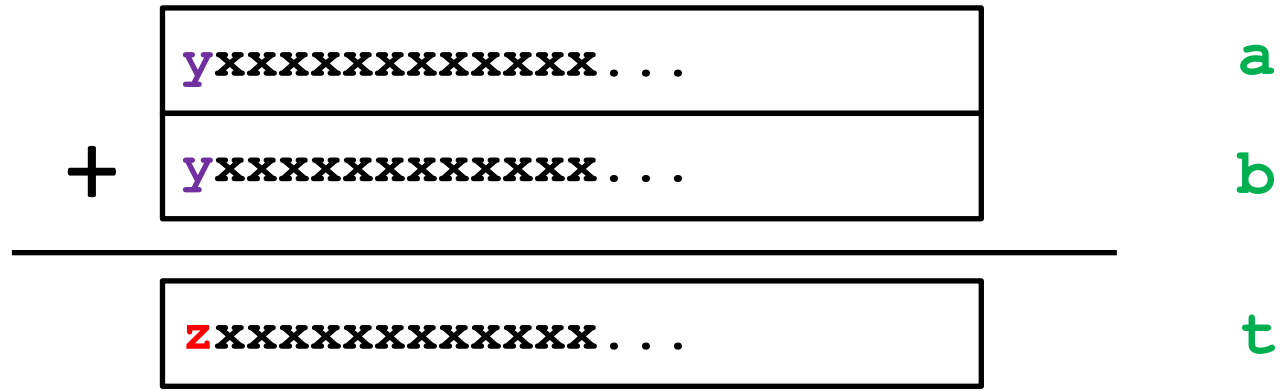
进位 Carry

借位 Borrow

对于无符号数计算，该标志报告产生溢出
For unsigned arithmetic, this reports overflow



OF set when



$$z = \sim y$$

$(a > 0 \ \&\& \ b > 0 \ \&\& \ t < 0) \ || \ (a < 0 \ \&\& \ b < 0 \ \&\& \ t \geq 0)$

对于有符号数计算，该标志报告溢出
For signed arithmetic, this reports overflow

条件码（显式设置：比较指令）

Condition Codes (Explicit Setting: Compare)



■ 由比较指令显式设置 Explicit Setting by Compare Instruction

- `cmpq Src2, Src1`
- `cmpq b, a` 类似计算 $a-b$ ，只是不设置目的操作数 like computing $a-b$ without setting destination
- **CF set** 如果从最高有效位进位（用于无符号数比较） if carry out from most significant bit (used for unsigned comparisons)
- **ZF set** 如果相等 if $a == b$
- **SF set** 如果小于（有符号数） if $(a-b) < 0$ (as signed)
- **OF set** 如果补码（有符号数）溢出 if two's-complement (signed) overflow $(a>0 \ \&\& \ b<0 \ \&\& \ (a-b)<0) \ || \ (a<0 \ \&\& \ b>0 \ \&\& \ (a-b)>0)$



条件码 (显式设置: 测试指令)

Condition Codes (Explicit Setting: Test)

■ 由测试指令显式设置 Explicit Setting by Test instruction

- `testq Src2, Src1`

- `testq b, a` 类似计算与操作, 但是不设置目的操作数 like computing `a&b` without setting destination

- 根据与运算的值设置条件码 Sets condition codes based on value of `Src1` & `Src2`

- 对于用一个操作数作为掩码很有用 Useful to have one of the operands be a mask

- **ZF set** 当与结果为0时 when `a&b == 0`

- **SF set** 当与结果小于0时 when `a&b < 0`

非常常用 Very often:

```
testq    %rax, %rax
```



读取条件码 Reading Condition Codes

■ SetX指令 SetX Instructions

- 根据条件码组合设置目的操作数低字节成0或1 Set low-order byte of destination to 0 or 1 based on combinations of condition codes
- 不会改变剩余的7个字节 Does not alter remaining 7 bytes

SetX	条件 Condition	描述 Description
sete	ZF	等于/零 Equal / Zero
setne	$\sim ZF$	不等/不为零 Not Equal / Not Zero
sets	SF	负数 Negative
setns	$\sim SF$	非负 Nonnegative
setg	$\sim (SF \wedge OF) \ \& \ \sim ZF$	大于（有符号） Greater (Signed)
setge	$\sim (SF \wedge OF)$	大于或等于（有符号数） Greater or Equal (Signed)
setl	$(SF \wedge OF)$	小于（有符号数） Less (Signed)
setle	$(SF \wedge OF) \mid ZF$	小于或等于（有符号数） Less or Equal (Signed)
seta	$\sim CF \ \& \ \sim ZF$	高于（无符号数） Above (unsigned)
setb	CF	低于（无符号数） Below (unsigned)



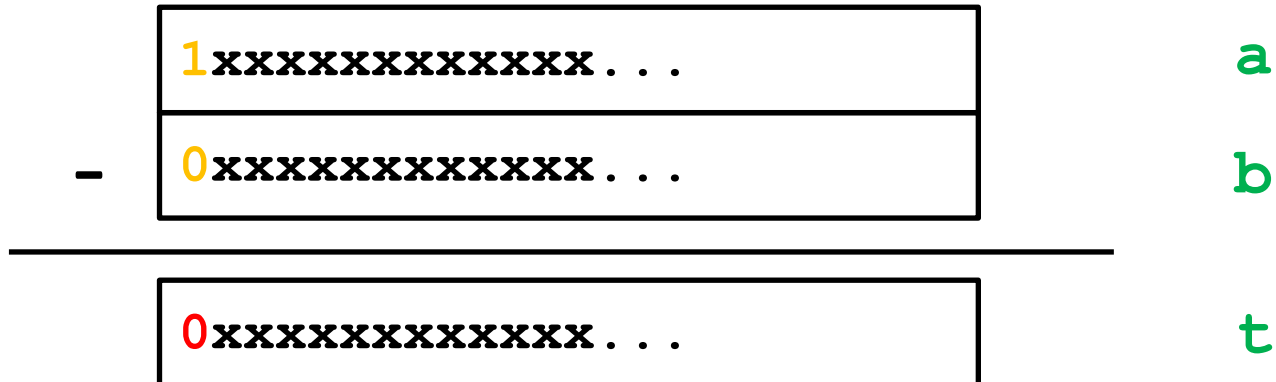
示例: setl (有符号数小于)

Example: setl (Signed <)

■ 情况 Condition: SF^OF

SF	OF	SF ^ OF	隐含 Implication
0	0	0	没有溢出, SF隐含着不小于 No overflow, so SF implies not <
1	0	1	没有溢出, SF隐含着小于 No overflow, so SF implies <
0	1	1	溢出, SF隐含着负溢, 即小于 Overflow, so SF implies negative overflow, i.e. <
1	1	0	溢出, SF隐含着正溢, 即不小于 Overflow, so SF implies positive overflow, i.e. not <

负溢的情况 negative overflow case



x86-64整数寄存器 x86-64 Integer Registers



%rax	%al
%rbx	%bl
%rcx	%cl
%rdx	%dl
%rsi	%sil
%rdi	%di1
%rsp	%spl
%rbp	%bpl

%r8	%r8b
%r9	%r9b
%r10	%r10b
%r11	%r11b
%r12	%r12b
%r13	%r13b
%r14	%r14b
%r15	%r15b

- 可以引用低字节 Can reference low-order byte

读取条件码 Reading Condition Codes (Cont.)



■ SetX指令 SetX Instructions:

- 根据条件码的组合设置单个字节 Set single byte based on combination of condition codes

■ 可寻址的字节寄存器之一 One of addressable byte registers

- 不会修改剩余的字节 Does not alter remaining bytes
- 典型地使用movzbl(0扩展字节到双字)来完成工作 Typically use **movzbl** to finish job
 - 32位指令也设置高32位为0 32-bit instructions also set upper 32 bits to 0

```
int gt (long x, long y)
{
    return x > y;
}
```

寄存器 Register	用途 Use(s)
%rdi	参数x Argument x
%rsi	参数y Argument y
%rax	返回值 Return value

```
cmpq    %rsi, %rdi    # Compare x:y
setg    %al           # Set when >
movzbl  %al, %eax      # Zero rest of %rax
ret
```

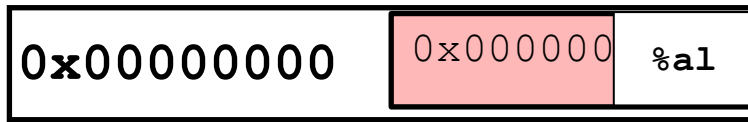
显式读取条件码 (续)



Explicit Reading Condition Codes (Cont.)

当心怪异 `movzbl` (和其它) Beware
weirdness `movzbl` (and others)

```
movzbl %al, %eax
```



全部归零

Zapped to all 0's

Use(s)

Argument **x**

Argument **y**

Return value

```
cmpq    %rsi, %rdi    # Compare x:y
setg     %al           # Set when >
movzbl   %al, %eax     # Zero rest of %rax
ret
```



议题

- **控制：条件码** Control: Condition codes
- **条件分支** Conditional branches
- **循环** Loops
- **Switch语句** Switch Statements



跳转指令 Jumping

■ 跳转指令 jX Instructions

- 根据条件码跳转到代码的不同部分 Jump to different part of code depending on condition codes

jX	条件 Condition	描述 Description
jmp	1	无条件 Unconditional
je	ZF	相等/零 Equal / Zero
jne	$\sim ZF$	不等/非零 Not Equal / Not Zero
js	SF	负数 Negative
jns	$\sim SF$	非负数 Nonnegative
jg	$\sim (SF \wedge OF) \ \& \ \sim ZF$	大于（有符号数） Greater (Signed)
jge	$\sim (SF \wedge OF)$	大于或等于（有符号数） Greater or Equal (Signed)
jl	$(SF \wedge OF)$	小于（有符号数） Less (Signed)
jle	$(SF \wedge OF) \mid ZF$	小于或等于（有符号数） Less or Equal (Signed)
ja	$\sim CF \ \& \ \sim ZF$	高于Above（无符号数） (unsigned)
jb	CF	低于Below（无符号数） (unsigned)

条件分支示例 (旧版风格)



Conditional Branch Example (Old Style)

■ 生成 Generation

```
shark> gcc -Og -S -fno-if-conversion control.c
```

Get to this shortly

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

```
absdiff:
    cmpq    %rsi, %rdi    # x:y
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:       # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

寄存器 Register	用途 Use(s)
%rdi	参数x Argument x
%rsi	参数y Argument y
%rax	返回值 Return value

用Goto代码表达 Expressing with Goto Code



- C语言允许使用goto语句 C allows goto statement
- 跳转到标号指示的位置 Jump to position designated by label

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

```
long absdiff_j
(long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
Else:
    result = y-x;
Done:
    return result;
}
```



通用条件表达式翻译（使用分支）

General Conditional Expression Translation (Using Branches)

C语言代码 C Code

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x > y ? x - y : y - x;
```

Goto版本 Goto Version

```
n_test = !Test;  
if (n_test) goto Else;  
val = Then_Expr;  
goto Done;  
Else:  
    val = Else_Expr;  
Done:  
    . . .
```

- 为then和else表达式分别创建代码区 Create separate code regions for then & else expressions
- 执行适当的一个代码区 Execute appropriate one

使用条件传送指令 Using Conditional Moves



■ 条件传送指令 Conditional Move

Instructions

- 指令支持: Instruction supports:
if (Test) Dest \leftarrow Src
- 在1995年之后的x86处理器得到支持
Supported in post-1995 x86 processors
- GCC尝试使用这些指令 GCC tries to use them
 - 但是, 仅仅当知道这样是在安全的情况下才行 But, only when known to be safe

■ 为何? Why?

- 指令流通过流水线时分支是非常容易引起混乱的 Branches are very disruptive to instruction flow through pipelines
- 条件传送不需要控制转移 Conditional moves do not require control transfer

C代码 C Code

```
val = Test  
    ? Then_Expr  
    : Else_Expr;
```

Goto版本 Goto Version

```
result = Then_Expr;  
eval = Else_Expr;  
nt = !Test;  
if (nt) result = eval;  
return result;
```



条件传送指令

- 只有条件成立时，才进行传送；否则，无操作

指令	条件	描述
cmove	ZF	相等/零
cmovne	$\sim ZF$	不相等/非零
cmovs	SF	负数
cmovns	$\sim SF$	非负数
cmovg	$\sim (SF \wedge OF) \& \sim ZF$	大于（有符号>）
cmovge	$\sim (SF \wedge OF)$	大于或等于（有符号 \geq ）
cmovl	$SF \wedge OF$	小于（有符号<）
cmovle	$(SF \wedge OF) \mid ZF$	小于或等于（有符号 \leq ）
cmova	$\sim CF \& \sim ZF$	高于（无符号>）
cmovae	$\sim CF$	高于或相等（无符号 \geq ）
cmovb	CF	低于（无符号<）
cmovbe	$CF \mid ZF$	低于或相等（无符号 \leq ）

条件传送示例 Conditional Move Example



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

寄存器 Register	用途 Use(s)
%rdi	参数x Argument x
%rsi	参数y Argument y
%rax	返回值 Return value

absdiff:

```
movq    %rdi, %rax    # x
subq    %rsi, %rax    # result = x-y
movq    %rsi, %rdx
subq    %rdi, %rdx    # eval = y-x
cmpq    %rsi, %rdi    # x:y
cmovle  %rdx, %rax    # if <=, result = eval
ret
```

何时这样做比较糟糕?

When is this bad?



对于条件传送糟糕的情况

Bad Cases for Conditional Move

需大量的计算 Expensive Computations

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

糟糕的性能

- 两个值都需要计算 Both values get computed
 - 仅当计算非常简单时才有意义 Only makes sense when computations are very simple
- Bad Performance

计算存在风险 Risky Computations

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

不安全 Unsafe

- 两个值都得到计算 Both values get computed
- 可能有不期望的效果 May have undesirable effects

计算有副作用 Computations with

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

不正确 Illegal

- 两个值都得到计算 Both values get computed
- 必须保证没有副作用 Must be side-effect free



练习

`cmpq b, a` like computing $a-b$ w/o setting dest

subq	Src, Dest	Dest = Dest - Src
xorq	Src, Dest	Dest = Dest ^ Src

■ **CF set** if carry/borrow out from most significant bit (used for unsigned comparisons)

■ **ZF set** if $a == b$

■ **SF set** if $(a-b) < 0$ (as signed)

■ **OF set** if two's-complement (signed) overflow

SetX	Condition	Description
<code>setl</code>	$SF \wedge OF$	Less (signed)

```
xorq    %rax, %rax
subq    $1, %rax
cmpq    $2, %rax
setl    %al
movzbl  %al, %eax
```

%rax	SF	CF	OF	ZF
0x0000 0000 0000 0000	0	0	0	1
0xFFFF FFFF FFFF FFFF	1	1	0	0
0xFFFF FFFF FFFF FFFF	1	0	0	0
0xFFFF FFFF FFFF FF01	1	0	0	0
0x0000 0000 0000 0001	1	0	0	0

Note: **setl** and **movzblq** do not modify condition codes



议题

- 控制：条件码 Control: Condition codes
- 条件分支 Conditional branches
- 循环 Loops
- Switch语句 Switch Statements



Do-While循环示例

“Do-While” Loop Example

C代码 C Code

```
long pcount_do
(unsigned long x) {
    long result = 0;
    do {
        result += x & 0x1;
        x >>= 1;
    } while (x);
    return result;
}
```

Goto版本 Goto Version

```
long pcount_goto
(unsigned long x) {
    long result = 0;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    return result;
}
```

- 计算参数x中1的个数 Count number of 1's in argument x (“popcount”)
- 使用条件分支要么继续循环要么退出循环
branch to either continue looping or to exit

X86作为CISC有
popcount指令 x86
being CISC has a
popcount instruction



通用 “Do-While” 翻译

General “Do-While” Translation

C代码 C Code

```
do  
    Body  
while (Test) ;
```

Goto版本 Goto Version

```
loop:  
    Body  
    if (Test)  
        goto loop
```

■ 循环体 Body:

```
{  
    Statement1;  
    Statement2;  
    ...  
    Statementn;  
}
```



Do-While循环翻译

“Do-While” Loop Compilation

Goto版本 Goto Version

```
long pcount_goto
(unsigned long x) {
    long result = 0;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    return result;
}
```

寄存器 Register	用途 Use(s)
%rdi	参数x Argument x
%rax	结果 result

```
        movl    $0, %eax    # result = 0
.L2:                                # loop:
        movq    %rdi, %rdx
        andl    $1, %edx    # t = x & 0x1
        addq    %rdx, %rax  # result += t
        shrq    %rdi        # x >>= 1
        jne     .L2         # if (x) goto loop
        rep; ret
```



通用 “While” 循环翻译方法#1

General “While” Translation #1

- “跳转到中间” 翻译方法 “Jump-to-middle” translation
- 使用编译参数 Used with -Og

While版本 While version

```
while (Test)  
    Body
```



Goto版本 Goto Version

```
goto test;  
loop:  
    Body  
test:  
    if (Test)  
        goto loop;  
done:
```

While循环示例 While Loop Example #1



跳转到中间版本

C代码 C Code

```
long pcount_while
(unsigned long x) {
    long result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```

Jump to Middle Version

```
long pcount_goto_jtm
(unsigned long x) {
    long result = 0;
    goto test;
loop:
    result += x & 0x1;
    x >>= 1;
test:
    if(x) goto loop;
    return result;
}
```

- 相较于该函数的do-while版本 Compare to do-while version of function
- 初始跳转到从测试开始循环 Initial goto starts loop at test

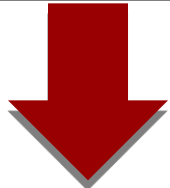


通用 “While” 翻译方法#2

General “While” Translation #2

While版本 While version

```
while (Test)  
    Body
```



Do-While版本 Do-While

```
if (!Test)  
    goto done;  
do  
    Body  
    while (Test) ;  
done:
```



Goto版本 Goto Version

```
if (!Test)  
    goto done;  
loop:  
    Body  
    if (Test)  
        goto loop;  
done:
```

- “Do-while”转换 “Do-while” conversion
- 使用编译参数 Used with -O1



While循环示例 While Loop Example #2

C代码 C Code

```
long pcount_while
(unsigned long x) {
    long result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```

Do-While版本 Do-While

```
long pcount_goto_dw
(unsigned long x) {
    long result = 0;
    if (!x) goto done;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
done:
    return result;
}
```

- 初始条件检测在循环入口 Initial conditional guards entrance to loop
- 相较于该函数的do-while版本 Compare to do-while version of function
 - 删除跳转到中间这个过程。是好是坏? Removes jump to middle. When is this good or bad?

“For”循环形式 “For” Loop Form



通用格式 General Form

```
for (Init; Test; Update )  
    Body
```

```
#define WSIZE 8*sizeof(int)  
long pcount_for  
    (unsigned long x)  
{  
    size_t i;  
    long result = 0;  
    for (i = 0; i < WSIZE; i++)  
    {  
        unsigned bit =  
            (x >> i) & 0x1;  
        result += bit;  
    }  
    return result;  
}
```

初始 Init

```
i = 0
```

测试 Test

```
i < WSIZE
```

更新 Update

```
i++
```

循环体 Body

```
{  
    unsigned bit =  
        (x >> i) & 0x1;  
    result += bit;  
}
```

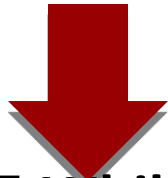


“For” 循环转换成While循环

“For” Loop → While Loop

For循环版本 For Version

```
for (Init; Test; Update)  
    Body
```



While循环版本 While Version

```
Init;  
while (Test) {  
    Body  
    Update;  
}
```

For-While转换 For-While Conversion



初始 Init

```
i = 0
```

测试 Test

```
i < WSIZE
```

更新 Update

```
i++
```

循环体 Body

```
{  
    unsigned bit =  
        (x >> i) & 0x1;  
    result += bit;  
}
```

```
long pcount_for_while  
(unsigned long x)  
{  
    size_t i;  
    long result = 0;  
    i = 0;  
    while (i < WSIZE)  
    {  
        unsigned bit =  
            (x >> i) & 0x1;  
        result += bit;  
        i++;  
    }  
    return result;  
}
```



“For”循环到Do-While循环转换

“For” Loop Do-While Conversion

Goto版本 Goto

C代码 C Code

```
long pcount_for
(unsigned long x)
{
    size_t i;
    long result = 0;
    for (i = 0; i < WSIZE; i++)
    {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    return result;
}
```

- 初始测试可以优化掉 Initial test can be optimized away

```
long pcount_for_goto_dw
(unsigned long x) {
    size_t i;
    long result = 0;
    i = 0; 初始 Init
    if (!(i < WSIZE)) 非终止测试 ! Test
    goto done;
loop:
{
    unsigned bit =
        (x >> i) & 0x1; 循环体 Body
    result += bit;
}
i++ 更新 Update
if (i < WSIZE) 测试 Test
    goto loop;
done:
    return result;
}
```



议题

- 控制：条件码 Control: Condition codes
- 条件分支 Conditional branches
- 循环 Loops
- Switch（开关）语句 Switch Statements



Switch语句示例

Switch Statement Example

```
long switch_eg
(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
            /* Fall Through */
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

- 多个情况标签 **Multiple case labels**
 - 此处： 5和6 Here: 5 & 6
- 落入其它情况 **Fall through cases**
 - 此处： 2 Here: 2
- 缺失的情况 **Missing cases**
 - 此处： 4 Here: 4

跳转表结构 Jump Table Structure



开关形式 Switch Form

```
switch(x) {  
  case val_0:  
    Block 0  
  case val_1:  
    Block 1  
    . . .  
  case val_n-1:  
    Block n-1  
}
```

跳转表 Jump Table

jtab:	Targ0
	Targ1
	Targ2
	•
	•
	•
	Targn-1

跳转目标 Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

Targ2: Code Block 2

•
•
•

Targn-1: Code Block n-1

翻译 Translation (Extended C)

```
goto *JTab[x];
```



开关语句示例 Switch Statement Example



```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        . . .
    }
    return w;
}
```

组织方式 Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi    # x:6
    ja     .L8
    jmp     *.L4(, %rdi, 8)
```



默认值的范围是多少? What range of values takes default?

寄存器 Register	用途 Use(s)
%rdi	参数x Argument x
%rsi	参数y Argument y
%rdx	参数z Argument z
%rax	返回值 Return value

注意w此处没有初始化 Note that **w** not initialized here



开关语句示例 Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        . . .
    }
    return w;
}
```

跳转表 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
```

组织方式 Setup:

```
switch_eg:
    movq      %rdx, %rcx
    cmpq      $6, %rdi      # x:6
    ja       .L8            # Use default
    jmp       *.L4(, %rdi, 8) # goto *JTab[x]
```

间接跳转
Indirect
jump



汇编程序组织方式解释

Assembly Setup Explanation



跳转表 Jump table

■ 跳转表结构 Table Structure

- 每个目标需要8字节 Each target requires 8 bytes
- 基地址在.L4处 Base address at .L4

■ 跳转 Jumping

- **直接跳转 Direct:** `jmp .L8`
- 跳转目标由标号.L8指示 Jump target is denoted by label .L8
- **间接跳转 Indirect:** `jmp *.L4(,%rdi,8)`
- 跳转表从.L4开始 Start of jump table: .L4
- 必须用8做比例因子（地址是8字节） Must scale by factor of 8 (addresses are 8 bytes)
- 从有效地址获取目标 Fetch target from effective Address $.L4 + x * 8$
 - 仅在范围内 Only for $0 \leq x \leq 6$

```
.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 Jump Table



跳转表 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
```

```
switch(x) {
case 1:      // .L3
    w = y*z;
    break;
case 2:      // .L5
    w = y/z;
    /* Fall Through */
case 3:      // .L9
    w += z;
    break;
case 5:
case 6:      // .L7
    w -= z;
    break;
default:    // .L8
    w = 2;
}
```

代码块 (x等于1时) Code Blocks (x == 1)



```
switch(x) {  
case 1:      // .L3  
    w = y*z;  
    break;  
    . . .  
}
```

```
.L3:  
    movq    %rsi, %rax    # y  
    imulq   %rdx, %rax    # y*z  
    ret
```

寄存器 Register	用途 Use(s)
%rdi	参数x Argument x
%rsi	参数y Argument y
%rdx	参数z Argument z
%rax	返回值 Return value



处理落入其它情况 Handling Fall-Through

```
long w = 1;  
.  
.  
.  
switch(x) {  
.  
.  
.  
case 2:   
    w = y/z;  
    /* Fall Through */  
case 3:   
    w += z;  
    break;  
.  
.  
.  
}
```

```
case 2:  
    w = y/z;  
    goto merge;
```

```
case 3:  
    w = 1;  
merge:  
    w += z;
```

代码块 (当x为2, 3时)

Code Blocks (x == 2, x == 3)



```
long w = 1;
. . .
switch(x) {
. . .
case 2:
    w = y/z;
    /* Fall Through */
case 3:
    w += z;
    break;
. . .
}
```

```
.L5:                                # Case 2
    movq    %rsi, %rax
    cqto                                # 扩展为8字节
    idivq    %rcx                       # y/z
    jmp     .L6                         # goto merge
.L9:                                # Case 3
    movl     $1, %eax                   # w = 1
.L6:                                # merge:
    addq     %rcx, %rax                 # w += z
    ret
```

寄存器 Register	用途 Use(s)
%rdi	参数x Argument x
%rsi	参数y Argument y
%rdx	参数z Argument z
%rax	返回值 Return value

代码块 (当x为5, 6时)

Code Blocks (x == 5, x == 6, default)



```
switch(x) {  
    . . .  
    case 5:  // .L7  
    case 6:  // .L7  
        w -= z;  
        break;  
    default: // .L8  
        w = 2;  
}
```

```
.L7:                                # Case 5,6  
    movl    $1, %eax               # w = 1  
    subq    %rdx, %rax             # w -= z  
    ret  
.L8:                                # Default:  
    movl    $2, %eax               # 2  
    ret
```

寄存器 Register	用途 Use(s)
%rdi	参数x Argument x
%rsi	参数y Argument y
%rdx	参数z Argument z
%rax	返回值 Return value

练习 Exercise



```
void switch2(long x, long *dest){
    long val = 0;
    switch (x) {
        ...
        Body of switch statement omitted
    }
    *dest = val;
}
```

- switch 语句内情况标号的值分别是多少?
- C代码中哪些情况有多个标号?

```
1  switch2:
2      addq    $1, %rdi
3      cmpq    $8, %rdi
4      ja      .L2
5      jmp     *.L4(, %rdi, 8)
```

```
1  .L4:
2      .quad   .L9    # x = -1
3      .quad   .L5    # x = 0
4      .quad   .L6    # x = 1
5      .quad   .L7    # x = 2
6      .quad   .L2    # x = 3
7      .quad   .L7    # x = 4
8      .quad   .L8    # x = 5
9      .quad   .L2    # x = 6
10     .quad   .L5    # x = 7
```

小结 Summarizing



■ C语言控制 C Control

- if-then-else
- do-while
- while, for
- switch

■ 汇编器控制 Assembler Control

- 条件跳转 Conditional jump
- 条件传送 Conditional move
- 间接跳转（通过跳转表） Indirect jump (via jump tables)
- 编译器生成代码序列实现更复杂的控制 Compiler generates code sequence to implement more complex control

■ 标准技术 Standard Techniques

- 循环转换成do-while或跳转到中间的形式 Loops converted to do-while or jump-to-middle form
- 大型switch语句使用跳转表 Large switch statements use jump tables
- 稀疏switch语句可能使用决策树（if-elseif-elseif-else） Sparse switch statements may use decision trees (if-elseif-elseif-else)



小结 Summary

■ 本次议题

- 控制：条件码 Control: Condition codes
- 条件分支和条件传送 Conditional branches & conditional moves
- 循环 Loops
- Switch语句 Switch statements

■ 下次议题 Next Time

- 栈 Stack
- 调用/返回 Call / return
- 过程调用准则 Procedure call discipline

找到二进制跳转表

Finding Jump Table in Binary



```
00000000004005e0 <switch_eg>:
4005e0:    48 89 d1                mov     %rdx,%rcx
4005e3:    48 83 ff 06            cmp     $0x6,%rdi
4005e7:    77 2b                  ja      400614 <switch_eg+0x34>
4005e9:    ff 24 fd f0 07 40 00   jmpq    *0x4007f0(,%rdi,8)
4005f0:    48 89 f0                mov     %rsi,%rax
4005f3:    48 0f af c2            imul    %rdx,%rax
4005f7:    c3                     retq
4005f8:    48 89 f0                mov     %rsi,%rax
4005fb:    48 99                  cqto
4005fd:    48 f7 f9                idiv    %rcx
400600:    eb 05                  jmp     400607 <switch_eg+0x27>
400602:    b8 01 00 00 00        mov     $0x1,%eax
400607:    48 01 c8                add     %rcx,%rax
40060a:    c3                     retq
40060b:    b8 01 00 00 00        mov     $0x1,%eax
400610:    48 29 d0                sub     %rdx,%rax
400613:    c3                     retq
400614:    b8 02 00 00 00        mov     $0x2,%eax
400619:    c3                     retq
```



找到二进制跳转表 (续)

Finding Jump Table in Binary (cont.)

```
00000000004005e0 <switch_eg>:  
. . .  
4005e9:      ff 24 fd f0 07 40 00      jmpq    *0x4007f0(,%rdi,8)  
. . .
```

```
% gdb switch  
(gdb) x /8xg 0x4007f0  
0x4007f0:      0x0000000000400614      0x00000000004005f0  
0x400800:      0x00000000004005f8      0x0000000000400602  
0x400810:      0x0000000000400614      0x000000000040060b  
0x400820:      0x000000000040060b      0x2c646c25203d2078  
(gdb)
```

找到二进制跳转表 (续)

Finding Jump Table in Binary (cont.)



```
% gdb switch
(gdb) x /8xg 0x4007f0
0x4007f0:      0x000000000000400614      0x0000000000004005f0
0x400800:      0x0000000000004005f8      0x000000000000400602
0x400810:      0x000000000000400614      0x00000000000040060b
0x400820:      0x00000000000040060b      0x2c646c25203d2078
```

...		
4005f0:	48 89 f0	mov %rsi,%rax
4005f3:	48 0f af c2	imul %rdx,%rax
4005f7:	c3	retq
4005f8:	48 89 f0	mov %rsi,%rax
4005fb:	48 99	cqto
4005fd:	48 f7 f9	idiv %rcx
400600:	eb 05	jmp 400607 <switch_eg+0x27>
400602:	b8 01 00 00 00	mov \$0x1,%eax
400607:	48 01 c8	add %rcx,%rax
40060a:	c3	retq
40060b:	b8 01 00 00 00	mov \$0x1,%eax
400610:	48 29 d0	sub %rdx,%rax
400613:	c3	retq
400614:	b8 02 00 00 00	mov \$0x2,%eax
400619:	c3	retq