程序的机器级表示II: 控制

教师：郑贵滨
计算机科学与技术学院
哈尔滨工业大学

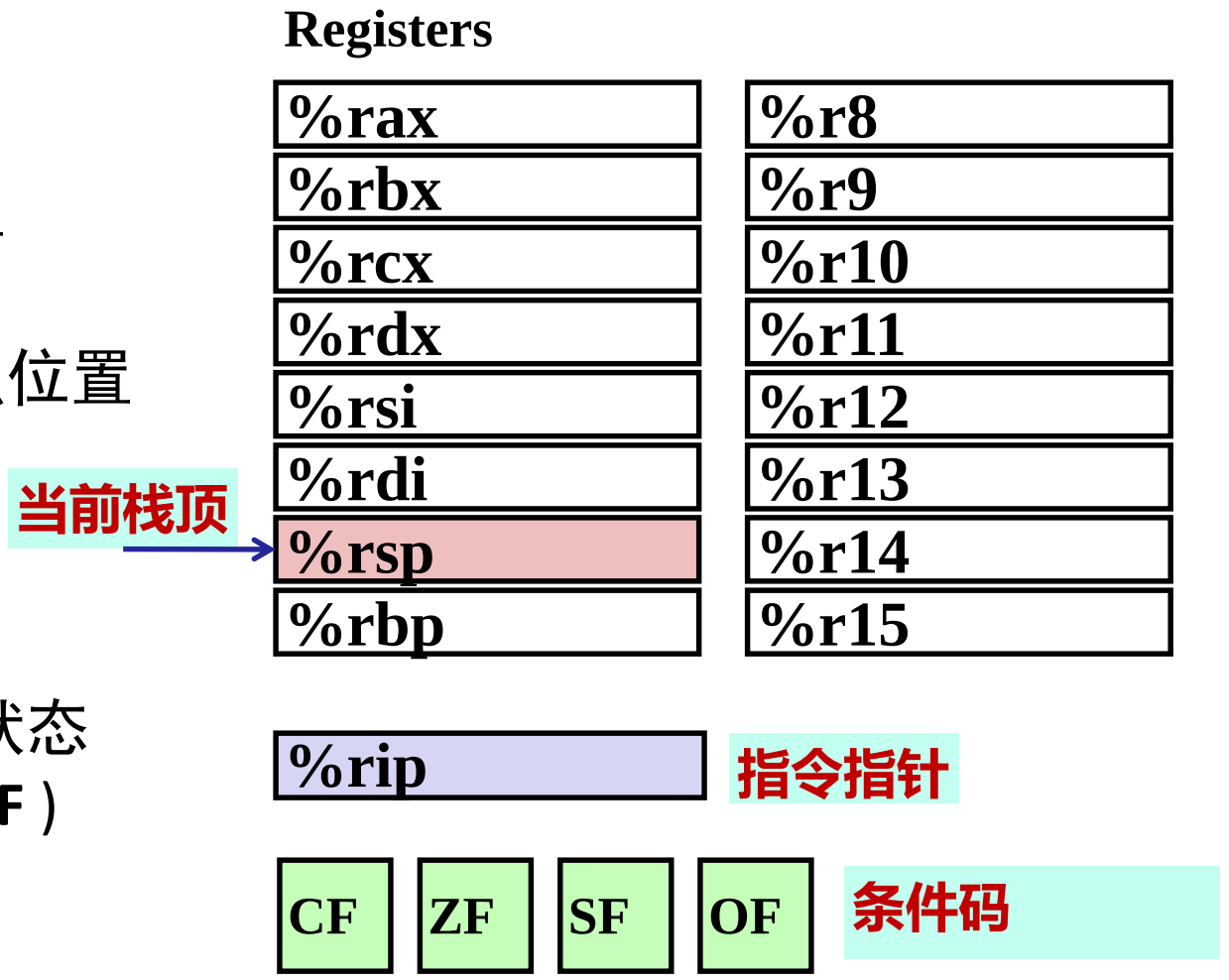
主要内容

- 流程控制:条件码(状态标志位)
- 条件分支
- 循环结构的实现
- Switch语句

- Chap 3.2, 3.3

处理器状态(x86-64, 一部分)

- 当前程序的运行状态信息
  - 临时数据
    (`%rax`, ...)
  - 运行时栈的位置
    (`%rsp`)
  - 当前程序控制点位置
    (`%rip`, ...)
  - 条件码
    - 状态标志位
    - 最近测试的状态
      (CF, ZF, SF, OF)



条件码——被隐含赋值

- 单个位的寄存器
  - **CF** 进位标志位(Carry Flag, 无符号数)
  - **SF** 符号标志识(位Sign Flag, 有符号数)
  - **ZF** 零标志位 (Zero Flag)
  - **OF** 溢出标志位(Overflow Flag , 有符号数)
- 数值由算术运算隐式自动赋值

  例: **addq** *Src*,*Dest* ↔ **t = a+b**

  CF=1 如果最高有效位有进位(无符号数溢出)，否则**CF=0**

  ZF=1 如 t == 0，否则**ZF=0**

  SF=1 如 **t < 0** (结果看做有符号数)，否则**SF=0**

  OF=1 如 有符号数(补码)溢出，否则**OF=0**

  (a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)
- 指令**leaq**不设置条件码

条件码(隐含赋值: Compare指令)

- Compare指令对条件码的隐含赋值
  - **cmpq** *Src1*, *Src2*

  cmpq *Src1*，*Src2* 计算***Src2*** – ***Src1*** 但不改变目的操作数，仅用结果设置条件码

- **cmpq** ***b*, *a***
  - **CF=1** 如果最高有效位有借位(无符号数比较)
  - **ZF=1** 如 a== **b**
  - **SF=1** 如 **(a-b) < 0** (有符号数比较)
  - **OF=1** 如补码 (有符号数)溢出

  (a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)

条件码

- Test指令对条件码的隐含赋值
  - **`testq`** *Src2*, *Src1*
    - 根据*Src1* & *Src2*的数值，设置条件码
    - 常用：一个操作数看做是一个掩码
  - **`testq b,a`**
    - 计算**`a&b`** 的结果后仅用于设置条件码，并不保存
    - **ZF=1** 如 **`a&b == 0`**
    - **SF=1** 如 **`a&b < 0`**
- 读取条件码：setX 指令 (X表示条件)
  - 根据条件码组合将目的操作数的低位字节设置为**0**或**1**
  - 不改变其余7字节

条件码

指令	同义词	作用	设置条件
sete	Setz	ZF	相等 / 结果为0
setne	setnz	~ZF	不相等 / 结果不为0
sets		SF	结果为负数
setns		~SF	结果为非负数
setl	setnge	SF^OF	小于 (符号数)
setle	setng	(SF^OF)|ZF	小于等于 (符号数)
setg	setnle	~(SF^OF)&~ZF	大于 (符号数)
setge	setnl	~(SF^OF)	大于等于 (符号数)
seta	setnbe	~CF&~ZF	大于 (无符号数)
setae	setnb	~CF	大于等于 (无符号数)
setb	setnae	CF	小于 (无符号数)
setbe	setna	CF|ZF	小于等于 (无符号数)

x86-64 整型数寄存器

- 可以引用低位字节

%rax	%al	%r8	%r8b
%rbx	%bl	%r9	%r9b
%rcx	%cl	%r10	%r10b
%rdx	%dl	%r11	%r11b
%rsi	%sil	%r12	%r12b
%rdi	%dil	%r13	%r13b
%rsp	%spl	%r14	%r14b
%rbp	%bpl	%r15	%r15b

读取条件码(续...)

- setX 指令:
  - 根据条件码组合,设置单个字节的数值
- 可寻址的单字节寄存器
  - 不改变寄存器其他字节的数值
  - 常用指令 **movzbl**将单字节值零扩展至整个8字节寄存器
    - 或者用32位指令将寄存器高位置0

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

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

寄存器	用途
%rdi	参数x
%rsi	参数y
%rax	返回值

主要内容

- 流程控制:条件码(状态标志位)
- 条件分支
- 循环结构的实现
- Switch语句

跳转

- jX 指令: 根据条件码跳转

jX指令	条件	描述
jmp	1	无条件
je	ZF	相等 / 结果为0
jne	~ZF	不相等 / 结果不为0
js	SF	结果为负数
jns	~SF	结果为非负数
jg	~(SF^OF)&~ZF	大于 (符号数)
jge	~(SF^OF)	大于等于 (符号数)
jl	(SF^OF)	小于 (符号数)
jle	(SF^OF)|ZF	小于等于 (符号数)
ja	~CF&~ZF	大于 (无符号数)
jb	CF	小于 (无符号数)

跳转

- 跳转指令的编码

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

关键：RIP/EIP 的含义

.o文件的反汇编

```
1  0: 48 89  f8     mov %rdi,%rax
2  3: eb 03         jmp 8 <loop+0x8>
3  5: 48 d1 f8      sar %rax
4  8: 48 85 c0      test %rax,%rax.
5  b: 7f f8         jg 5 <loop+0x5>
6  d: f3 c3         repz retq
```

链接后的程序反汇编

```
1  4004d0: 48 89 f8     mov %rdi,%rax
2  4004d3: eb 03        jmp 4004d8 <loop+0x8>
3  4004d5: 48 dl 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
```

条件分支(旧风格)

- 生成

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

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

寄存器	用途
%rdi	参数x
%rsi	参数y
%rax	返回值

```
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
```

用goto表述

- C 允许使用goto语句
- 跳转到标号指定的位置

```
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;
}
```

用分支翻译一般条件表达式

C 代码

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

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

使用goto的版本

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

- 为 then 表达式和 else表达式生成独立的代码段
- 执行合适的代码段

条件传送指令

- 条件传送指令
  - 指令支持的操作:

    if (Test) Dest ← Src
  - 1995年之后的 x86处理器支持条件传送
  - GCC 尽量使用它们
    - 前提条件：安全
- Why?
  - 分支对流水线中的指令流非常有害
  - 条件传送不需要控制转移

C代码

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

不使用goto的版本

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

条件传送——例子

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

寄存器	用途
%rdi	参数x
%rsi	参数y
%rax	返回值

```
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
```

条件传送的不良案例

计算代价大

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

- 两个数值均被计算
- 只有计算非常简单时才有意义。

危险的计算

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

- 两个数值均被计算
- 可能获得期望的效果

有副作用的计算

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

- 两个数值均被计算
- 副作用！

主要内容

- 流程控制:条件码(状态标志位)
- 条件分支
- 循环结构的实现
- Switch语句

“Do-While” 循环示例

- 计算参数x中1的个数 (popcount) (参考程序loop.c)
- 使用条件分支实现继续循环或退出循环

C代码

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

goto版本

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

“Do-While” 循环的汇编实现

goto版本

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

寄存器	用途
%rdi	参数x
%rax	返回值

```
    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
```

“Do-While”的通常实现方法

C代码

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

goto版本

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

Body:

```
{
    语句1;
    语句2;
     …
    语句n;
}
```

“While” 循环的实现方法

- “跳到中间”翻译法
- 使用选项 –Og

While版本

```
while (Test)
   Body
```

goto版本

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

While循环实现# 1

C代码

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

跳到中间版本

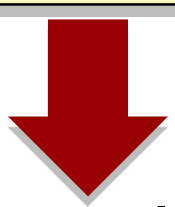
```
long pcount_goto_jtm
  (unsigned long x) {
  long result = 0;
  goto test;
 loop:
  result += x & 0x1;
  x >>= 1;
 test:
  if(x) goto loop;
  return result;
}
```

- 与do-while版本的函数比较：
  第一个goto语句跳到test处启动循环

While循环实现#2

While版本

```
while (Test)
  Body
```



Do-While版本

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

goto版本

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

- “Do-while” 转换
- 使用选项 –O1

While循环——例2

C代码

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

Do-While版本

```
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;
}
```

- 与do-while版本的函数对比
  初始条件守护循环的入口

“For” 循环形式

一般形式

```
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;
}
```

循环初始化(Init)

```
i = 0
```

循环条件测试(Test)

```
i < WSIZE
```

更新(Update)

```
i++
```

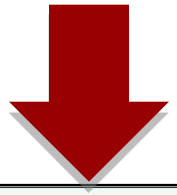
循环体(Body)

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

“For” 循环→ While循环

For版本

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



While版本

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

For-While 转换

循环初始化

```
i = 0
```

循环条件测试

```
i < WSIZE
```

更新

```
i++
```

循环体

```
{
  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 循环的Do-While实现

C代码

```
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;
}
```

- #define WSIZE 8*sizeof(int)
- “初始化测试”被优化删除

goto版本

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

主要内容

- 流程控制:条件码(状态标志位)
- 条件分支
- 循环结构的实现
- **Switch语句**

Switch语句

- 多case标号
  - 本例：5 & 6
- 下穿case语句
  - 本例: 2
- 缺失的case
  - 本例: 4

```
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;
        /* 下穿Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

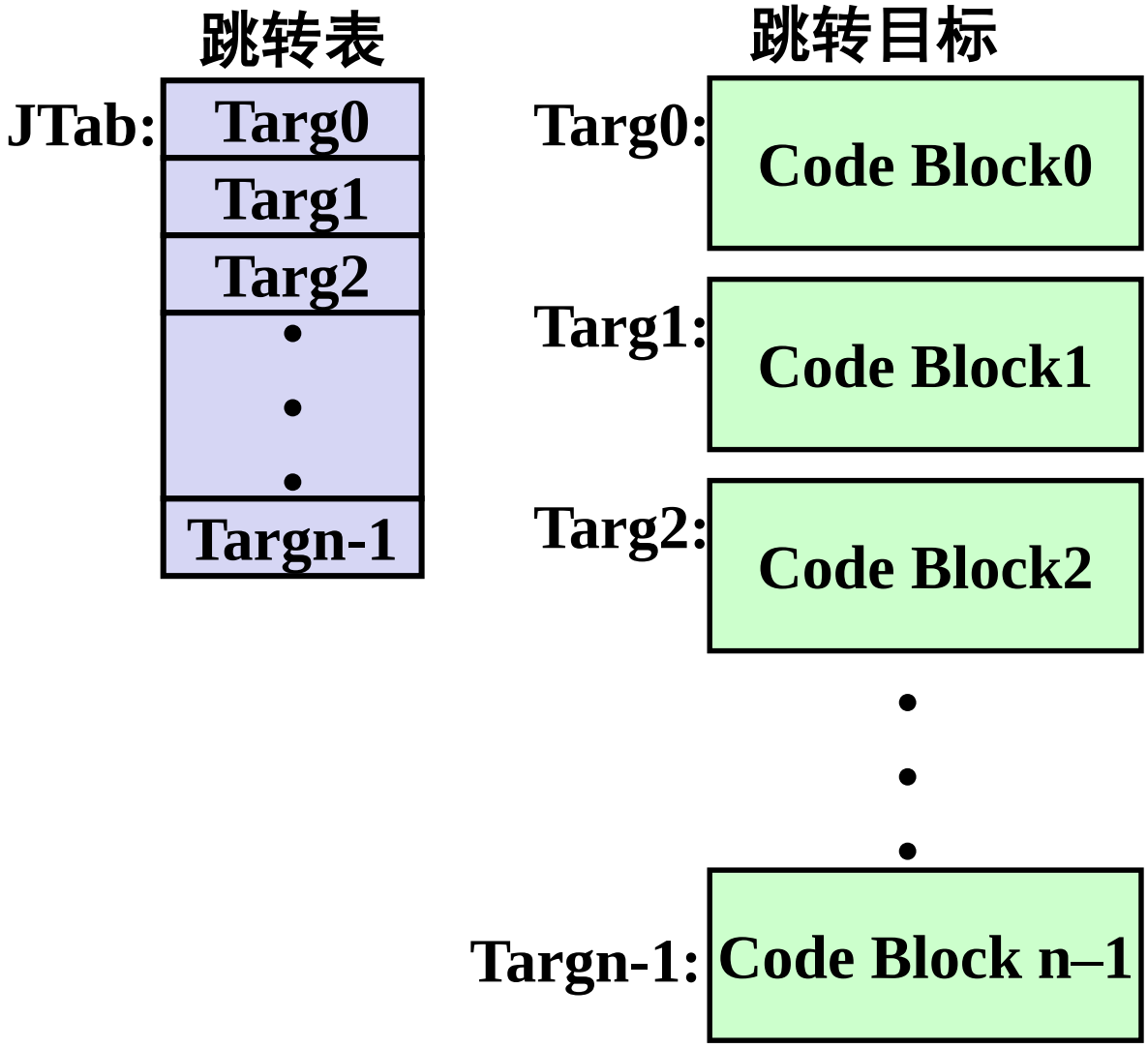
跳转表结构

Switch语句

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

翻译 (Extended C)

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



Switch语句示例

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

寄存器	用途
%rdi	参数x
%rsi	参数y
%rdx	参数z
%rax	返回值

汇编实现:

注意：此处没有w及初始化

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

属于default的数值范围?

Switch语句示例

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

跳转表

```
.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
```

汇编实现:

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

汇编实现的解释

- **表结构**
  - 每个目标需8字节
  - 基地址为.L4

跳转表

```
.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
```

- **跳转指令**
  - 直接跳转：**jmp .L8**
    跳转到标号.L8处
  - 间接跳转：**jmp *.L4(,%rdi,8)**
    跳转表的起始地址：.L4
  - 必须以8为比例因子 (地址是8字节)
  - $0 \le x \le 6$时，从有效地址**.L4 + x*8**处获取目标地址

跳转表

跳转表

```
.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 == 1)

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

寄存器	用途
%rdi	参数x
%rsi	参数y
%rdx	参数z
%rax	返回值

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

处理下穿(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;
```

```
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 == 2, x == 3)

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

寄存器	用途
%rdi	参数x
%rsi	参数y
%rdx	参数z
%rax	返回值

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

代码块(x == 5, x == 6, default)

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

寄存器	用途
%rdi	参数x
%rsi	参数y
%rdx	参数z
%rax	返回值

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

Switch语句

- 条件值稀疏的switch语句

```
        ...
movq    %rdx, %rcx
        cmpq    $300, %rdi
        je      .L18
        cmpq    $300, %rdi
        jg      .L13
        cmpq    $100, %rdi
        je      .L14
        cmpq    $200, %rdi
        jne     .L17
        movq    %rsi, %rax
        cqto
        idivq   %rcx
.L12:
        addq    %rcx, %rax
        ret
.L13:
        cmpq    $500, %rdi
        je      .L16
        cmpq    $600, %rdi
        je      .L16
```

```
long sparse_switch_eg
(long x, long y, long z) {
  long w = 1;
  switch(x) {
        case 100:
                w = y*z;
                break;
        case 200:
                w = y/z;
                /* Fall through */
        case 300:
                w += z;
                break;
        case 500:
        case 600:
                w -= z;
                break;
        default:
                w = 2;
  }
  return w;
}
```

小结

- C 控制语句
  - if-then-else
  - do-while
  - while, for
  - switch
- 汇编控制语句
  - 条件jump
  - 条件传送
  - 间接跳转(使用跳转表)
  - 汇编器生成代码序列实现复杂的流程控制

小结

- 标准技术
  - 循环结构的实现
    - 转换成do-while
    - 转换成“跳到中间”形式
  - **大的Switch语句使用跳转表**
  - **稀疏的Switch语句 使用决策树 (if-elseif-elseif-else)**

小结

- 主要内容
  - 流程控制:条件码(状态标志位)
  - 条件分支 & conditional moves
  - 循环结构的实现
  - Switch语句
- 下一讲
  - 栈(Stack)
  - 调用/返回
  - 过程调用的原则