程序的机器级表示：控制

Machine-Level Programming : Control

本章内容

Topic

- 条件码

  Condition codes

- 条件分支

  Conditional branches

- 循环

  Loops

- switch语句

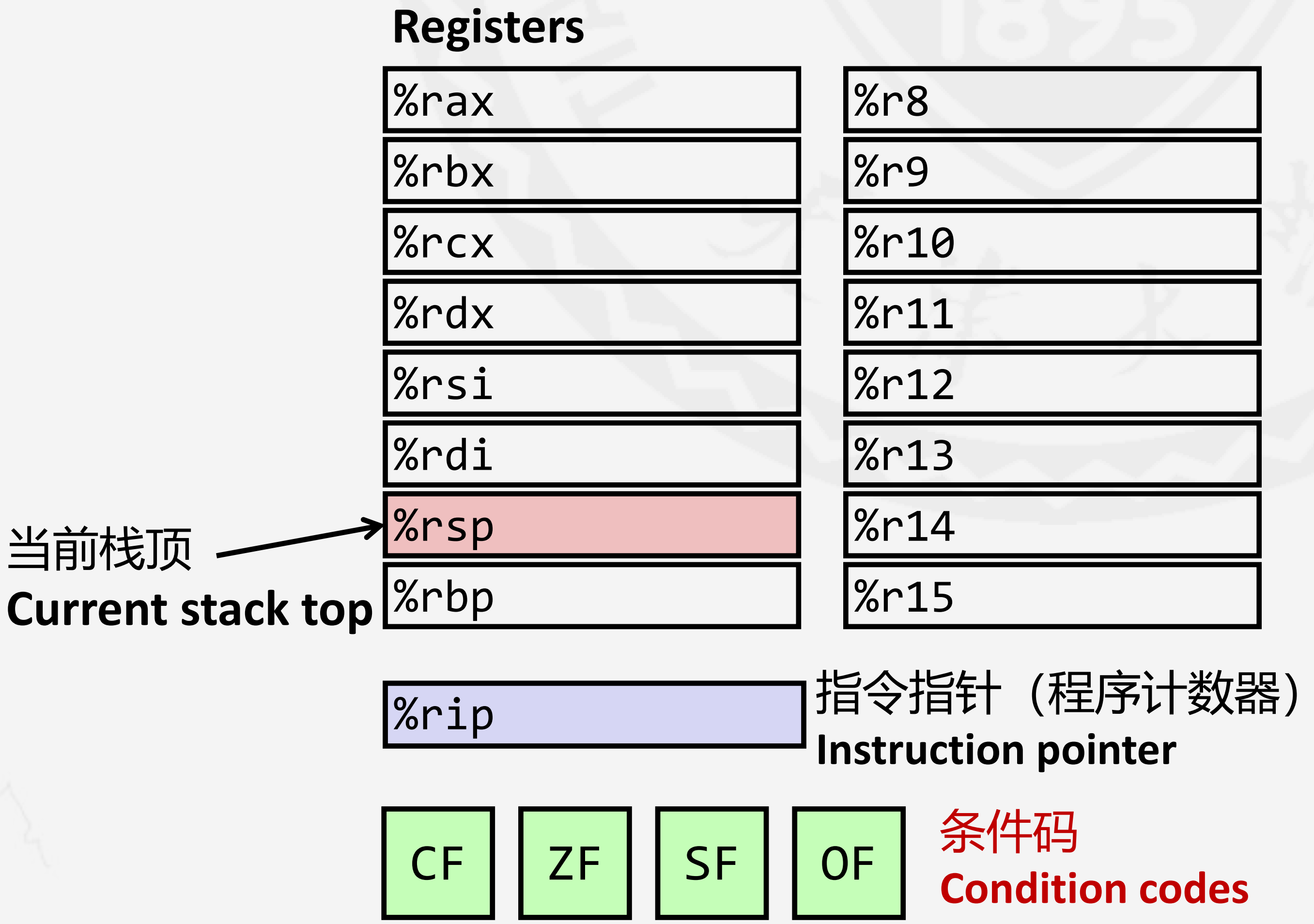
  Switch Statements

条件码

Condition codes

处理器状态（x86-64，部分的）
Processor State (x86-64, Partial)

- 当前程序的执行信息
  Information about currently executing program
  - 临时数据
    Temporary data
    (%rax, …)
  - 运行时栈的位置（栈顶）
    Location of runtime stack
    (%rsp)
  - 当前代码控制点的位置（即将要执行的指令地址）
    Location of current code control point
    (%rip, …)
  - 最近一次指令执行的状态
    Status of recent tests
    (CF, ZF, SF, OF)



条件码

Condition codes

条件码（隐式设置）
Condition Codes (Implicit Setting)

`leaq` 指令不修改条件码
Not set by `leaq` instruction

- 位寄存器

  Single bit registers

CF Carry Flag (for unsigned)
进位标志（无符号数）

SF Sign Flag (for signed)
符号标志（有符号数）

ZF Zero Flag
零标志

OF Overflow Flag (for signed)
溢出标志（有符号数）

- 通过算术运算可以隐式设置条件码（可以把它看做是运算的副作用）

  Implicitly set (think of it as side effect) by arithmetic operations

  - 例如： `addq Src,Dest` ↔ `t = a+b`

    Example: `addq Src,Dest` ↔ `t = a+b`

    - **CF** 被置位，如果运算时出现了超出最高位的进位（无符号数运算溢出）

      CF set if carry out from most significant bit (unsigned overflow)

    - **ZF** 被置位，如果 t == 0

      ZF set if t == 0

    - **SF** 被置位，如果 t<0 （看做是有符号数）

      SF set if t < 0 (as signed)

    - **OF** 被置位，如果有符号数运算出现了溢出

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

      (a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)

条件码（显式设置：比较指令）
Condition Codes (Explicit Setting：Compare)

- 使用比较指令可以显式设置条件码
  Explicit Setting by Compare Instruction
  - `cmpq Src2, Src1`
  - `cmpq b,a` 这条指令和a-b的作用类似，但不需要将结果写入目标寄存器
    `cmpq b,a` like computing **a-b** without setting destination
    - **CF** 被置位，如果运算时出现了超出最高位的借位（用于无符号数比较）
      CF set if carry out from most significant bit (used for unsigned comparisons)
    - **ZF** 被置位，如果 a == b
      ZF set if a == b
    - **SF** 被置位，如果 (a-b) < 0 （看做是有符号数）
      SF set if (a-b) < 0 (as signed)
    - **OF** 被置位，如果有符号数运算出现了溢出
      OF set if two's-complement (signed) overflow
      (a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)

条件码

Condition codes

条件码（显式设置：测试指令）
Condition Codes (Explicit Setting：Test)

- 使用测试指令也可以显式设置条件码
  Explicit Setting by Test Instruction
  - `testq Src2, Src1`
  - `testq b,a` 这条指令和**a&b**的作用类似，但不需要将结果写入目标寄存器
    `testq b,a` like computing **a&b** without setting destination
  - 根据 `Src1&Src2` 的结果设置条件码
    Sets condition codes based on value of Src1 & Src2
  - 用于对一个操作数的某几个位进行掩码检测
    Useful to have one of the operands be a mask
    - ZF 被置位，当 a&b == 0
      ZF set if a&b == 0
    - SF 被置位，如果 (a&b) < 0
      SF set if (a&b) < 0

条件码

Condition codes

读取条件码
Reading Condition Codes

- SetX指令
  SetX Instructions
  - 根据条件码表达式将目标寄存器的最后一个字节修改为0或1
    Set low-order byte of destination to 0 or 1 based on combinations of condition codes
  - 不会影响目标寄存器最高7个字节的值
    Does not alter remaining 7 bytes

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)

条件码

Condition codes

x86-64 各寄存器中最后一个字节的名称
Referencing low-order byte of x86-64 Register

Register	Low byte	Register	Low byte
%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
Reading Condition Codes

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

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rax	Return value

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

在x86-64指令集中，32位操作指令会将目标寄存器的高32位清0

In x86-64 ISA, 32-bit instructions also set upper 32 bits to 0

本章内容

Topic

- 条件码

  Condition codes

- 条件分支

  Conditional branches

- 循环

  Loops

- switch语句

  Switch Statements

条件分支

Conditional branches

跳转
Jumping

- jX指令
  jX Instruction
  - 根据条件码跳转到代码的其他位置执行
    Jump to different part of code depending on condition codes

jX	Condition	Description
jmp	1	**Unconditional （无条件）**
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

Conditional branches

跳转指令的编码
Jump Instruction Encoding

```
loop:
        movq    %rdi, %rax
        jmp             .L2
.L3:
        sarq    %rax
.L2:
        testq   %rax, %rax
        jg              .L3
        rep; ret
```

```
4004d0:     48 89 f8          movq    %rdi, %rax
4004d3:     eb 03             jmp     4004d8<loop+0x8>
4004d5:     48 d1 f8          sarq    %rax
4004d8:     48 85 c0          testq   %rax, %rax
4004db:     7f f8             jg      4004d5<loop+0x5>
4004dd:     f3 c3             repz retq
```

条件分支

Conditional branches

条件分支示例（早期模式）
Conditional Branch Example (Old Style)

生成汇编代码

Generation

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

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

control.c

```
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
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rax	Return value

条件分支

Conditional branches

使用goto语句等价表示
Expressing with goto Code

- C语言允许使用goto语句
  C allows goto statement
- 跳转至标签所在位置的语句继续执行
  Jump to position designated by label

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

```c
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;
}
```

条件分支

Conditional branches

条件表达式的翻译（使用分支）
General Conditional Expression Translation (Using Branches)

C 代码

C Code

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

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

- 为Then和Else表达式创建独立的代码块
  separate code regions for then & else expressions
- 根据条件选择合适的一个代码块并执行
  Execute appropriate one

goto 语句版本

goto Version

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

条件分支

Conditional branches

使用条件数据移动指令
Using Conditional Moves

- 条件数据移动指令
  Conditional Move Instructions
  - 指令的功能： `if (Test) Dest ← Src`
    Instruction supports: `if (Test) Dest ← Src`
  - 1995年后的x86处理器开始支持
    Supported in post-1995 x86 processors
  - GCC在编译时会尝试使用这个指令翻译条件分支
    GCC tries to use them
    - 仅当保证逻辑安全的时候使用
      But, only when known to be safe
- 为什么使用条件数据移动指令?
  Why?
  - 分支会破坏流水线的指令流，影像处理器性能
    Branches are very disruptive to instruction flow through pipelines
  - 条件数据移动指令不需要改变控制流
    Conditional moves do not require control transfer

C 代码
C Code

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

goto 语句版本
goto Version

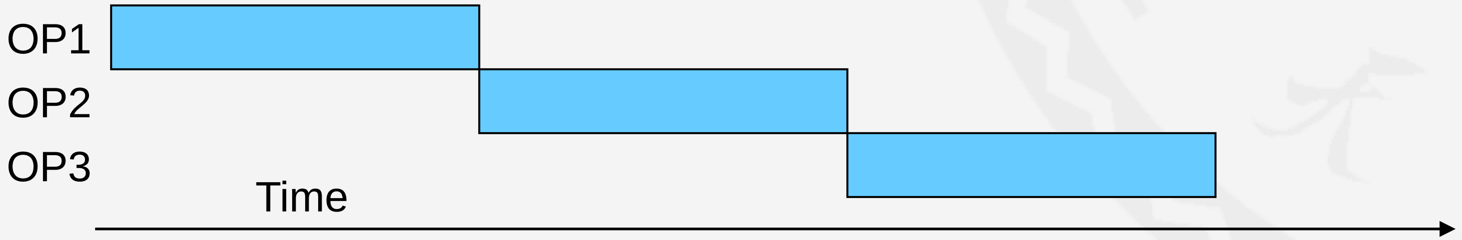
```
result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
return result;
```

条件分支

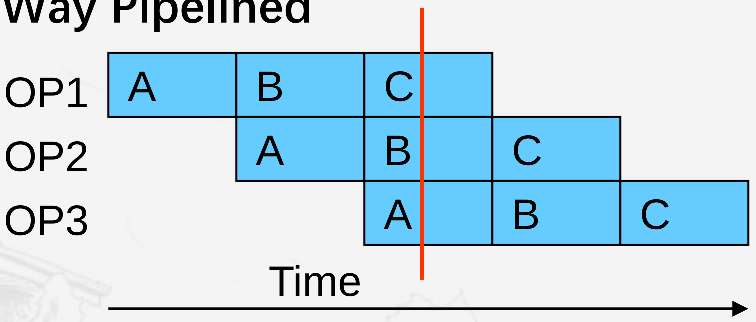
Conditional branches

流水线
Pipeline

无流水线
Unpipelined



3阶段流水线
3-Way Pipelined



最多可以有三条指令同时执行
Up to 3 operations in process simultaneously

举例：条件数据移动指令
Conditional Move Example

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rax	Return value

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

```
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
```

不能使用条件数据移动指令的情况
Bad Cases for Conditional Move

- 大量的计算
  Expensive Computations
  - 条件数据移动指令会将所有的结果提前计算出来
    Both values get computed
  - 只有计算都非常简单的时候，使用条件数据移动指令才会有意义
    Only makes sense when computations are very simple

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

- 存在风险的计算
  Risky Computations
  - 可能导致程序出错
    May have undesirable effects

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

- 有副作用的计算
  Computations with side effects

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

本章内容

Topic

- 条件码
  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 Code

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

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x编码中“1” 的个数
Count number of 1’s in argument x

使用条件分支决定继续或退出循环
Use conditional branch to either continue looping or to exit loop

Do-While循环编译后的结果
"Do-While" Loop Compilation

Register	Use(s)
%rdi	Argument x
%rax	result

goto Version

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

Do-While循环通用的翻译方式
General "Do-While" Translation

C Code

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

Goto Version

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

```
Body:  {
         Statement1;
         Statement2;
           …
         Statementn;
       }
```

循环

Loops

While循环通用的翻译方式 #1
General “While” Translation #1

- “跳转到中间”翻译方法
  “Jump-to-middle” translation
- 使用 –Og 编译优化选项生成代码
  Used with -Og

While version

```
while (Test)
  Body
```

Goto Version

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

循环

Loops

与 do-while 循环相比，循环开始前先跳转至循环条件检测的位置

Compared with do-while version of function, initial goto starts loop at test

举例：While循环 #1
While Loop Example #1

C Code

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

Jump to Middle

```
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;
}
```

While循环通用的翻译方式 #2
General “While” Translation #2

While version

```
while (Test)
  Body
```

Do-While Version

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

- 使用 –O1 编译优化选项生成代码

  Used with –O1

Goto Version

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

循环

Loops

与 do-while 循环相比，循环开始前先检测循环条件，再进入循环

Compared with do-while version of function, initial conditional guards entrance to loop

举例：While循环 #2
While Loop Example #2

C Code

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

Do-While Version

```
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;
}
```

循环

Loops

For循环一般形式
“For” Loop Form

一般形式

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;
}
```

初始化

Init

```
i = 0
```

检测

Test

```
i < WSIZE
```

更新

Update

```
i++
```

循环体

Body

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

循环

Loops

“For” Loop → While Loop → Goto

For Version

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

While Version

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

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

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

For与While之间的转换
For-While Conversion

Init
```
i = 0
```

Test
```
i < WSIZE
```

Update
```
i++
```

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;
}
```

循环

Loops

举例：For循环编译后的结果
“For” Loop Conversion Example

在这个例子中，初始的循环条件检测可以编译器被优化掉

Initial test can be optimized away

C Code

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
  int i;
  int result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  return result;
}
```

Goto Version

```
int pcount_for_gt(unsigned x)
{
  int i;
  int result = 0;
  i = 0;                          Init
  if (!(i < WSIZE))               ! Test
    goto done;
 loop:
  {                                        Body
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  i++;                            Update
  if (i < WSIZE)
    goto loop;                    Test
 done:
  return result;
}
```

本章内容

Topic

- 条件码

  Condition codes

- 条件分支

  Conditional branches

- 循环

  Loops

- switch语句

  Switch Statements

switch语句

Switch Statements

Switch语句示例
Switch Statement Example

- 多个case共用同一语句块
  Multiple case labels
  - 5 & 6
- Case贯穿
  Fall through cases
  - 2
- Case缺失（case值不连续）
  Missing cases
  - 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;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

跳转表的结构
Jump Table Structure

Switch的一般形式
Switch Form

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

翻译后 （扩展C）
Translation (Extended C)

```
goto *JTab[op];
```

跳转表
Jump Table

```
jtab: Targ0
      Targ1
      Targ2
      •
      •
      •
      Targ n-1
```

跳转目标（语句块）
Jump Targets

Targ 0:	Code Block 0
Targ 1:	Code Block 1
Targ 2:	Code Block 2
	• • •
Targ *n*-1:	Code Block *n*–1

switch语句

Switch Statements

分析跳转表 1
Understanding Jump Table 1

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

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rdx	Argument **z**
%rax	Return value

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

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

注意：w 并没有在switch开始前初始化
Note that **w** not initialized here

分析跳转表 2
Understanding Jump Table 2

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

```
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

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语句

Switch Statements

Setup部分汇编语句说明
Reading Condition Codes

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
```

- 跳转表结构
  Jump Table Structure
  - 基地址是 .L4
    Base address at .L4
  - 每个跳转目标需要8个字节（指向目标语句块的地址）
    Each target requires 8 bytes
- **直接跳转**： `jmp .L8`
  Direct: `jmp .L8`
  - 直接跳转至.L8标签所指向地址的指令
    Jump target is denoted by label .L8
- **间接跳转**： `jmp *.L4(,%rdi,8)`
  Indirect: `jmp *.L4(,%rdi,8)`
  - 跳转表起始地址.L4
    Start of jump table: .L4
  - 缩放因子必须是8的整倍数（每个地址是8个字节）
    Must scale by factor of 8 (addresses are 8 bytes)
  - 从地址 .L4 + x*8 处获得跳转目标的位置
    Fetch target from effective Address `.L4 + x*8`
    - 仅限于 $0 \leqslant x \leqslant 6$的情况
      Only for $0 \leqslant x \leqslant 6$

分析跳转表 3
Understanding Jump Table 3

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;
}
```

switch语句

Switch Statements

代码块（x==1）
Code Blocks (x == 1)

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

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

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rdx	Argument **z**
%rax	Return value

switch语句

Switch Statements

处理落入另一个case的情况
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;
```

switch语句

Switch Statements

代码块（x==2，x==3）
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
    idivq   %rcx            #  y/z
    jmp     .L6             #  goto merge
.L9:                        # Case 3
    movl    $1, %eax        #  w = 1
.L6:                        # merge:
    addq    %rcx, %rax #  w += z
    ret
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rdx	Argument **z**
%rax	Return value

switch语句

Switch Statements

代码块（x==5，x==6，缺省）
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
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rdx	Argument **z**
%rax	Return value

没有从0处开始的情况
Case does not Start From 0

```
void switch_eg_2 (long x) {
    switch(x) {
    case 10000:
       ……
    case 10002:
       ……
    case 10003:
       ……
    case 10005:
       ……
    case 10006:
       ……
    default:
       ……
    }
}
```

```
switch_eg_2:
    leaq -$10000(%rdi), %rsi # %rsi=%rdi-10000
    cmpq $6, %rsi            # x:6
    ja   .L8                 # Use default
    jmp  *.L4(,%rsi,8)       # goto *JTab[x]
```

稀疏的switch语句
Sparse Switch Statements

```
switch(x) {
  case 0:
      . . .
  case 1000:
      . . .
  case 92027:
      . . .
}
```

```
if () {
   if () {
      . . .
   } else {
      . . .
   }
} else {
   if () {
      . . .
   } else {
      . . .
   }
}
```

- 将翻译为二分查找的语句 O(log n)
  may translate into binary search
- 而不是退化为 if-elseif-elseif-else O(n)
  not if-elseif-elseif-else O(n)

switch语句

Switch Statements

总结
Summarizing

- C语言的控制方法
  C Control
  - if-then-else
  - do-while
  - while, for
  - switch
- 汇编语言的控制方法
  Assembler Control
  - 条件跳转
    Conditional jump
  - 条件数据移动
    Conditional move
  - 间接跳转（利用跳转表实现）
    Indirect jump (via jump tables)
  - 编译器通过自动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
  - 将循环转换为do-while 或 jump-to-middle 形式
    Loops converted to do-while or jump-to-middle form
  - 大规模的switch语句可以使用跳转表实现
    Large switch statements use jump tables
  - 稀疏的switch语句可以使用决策树（二分查找）实现
    Sparse switch statements may use decision trees