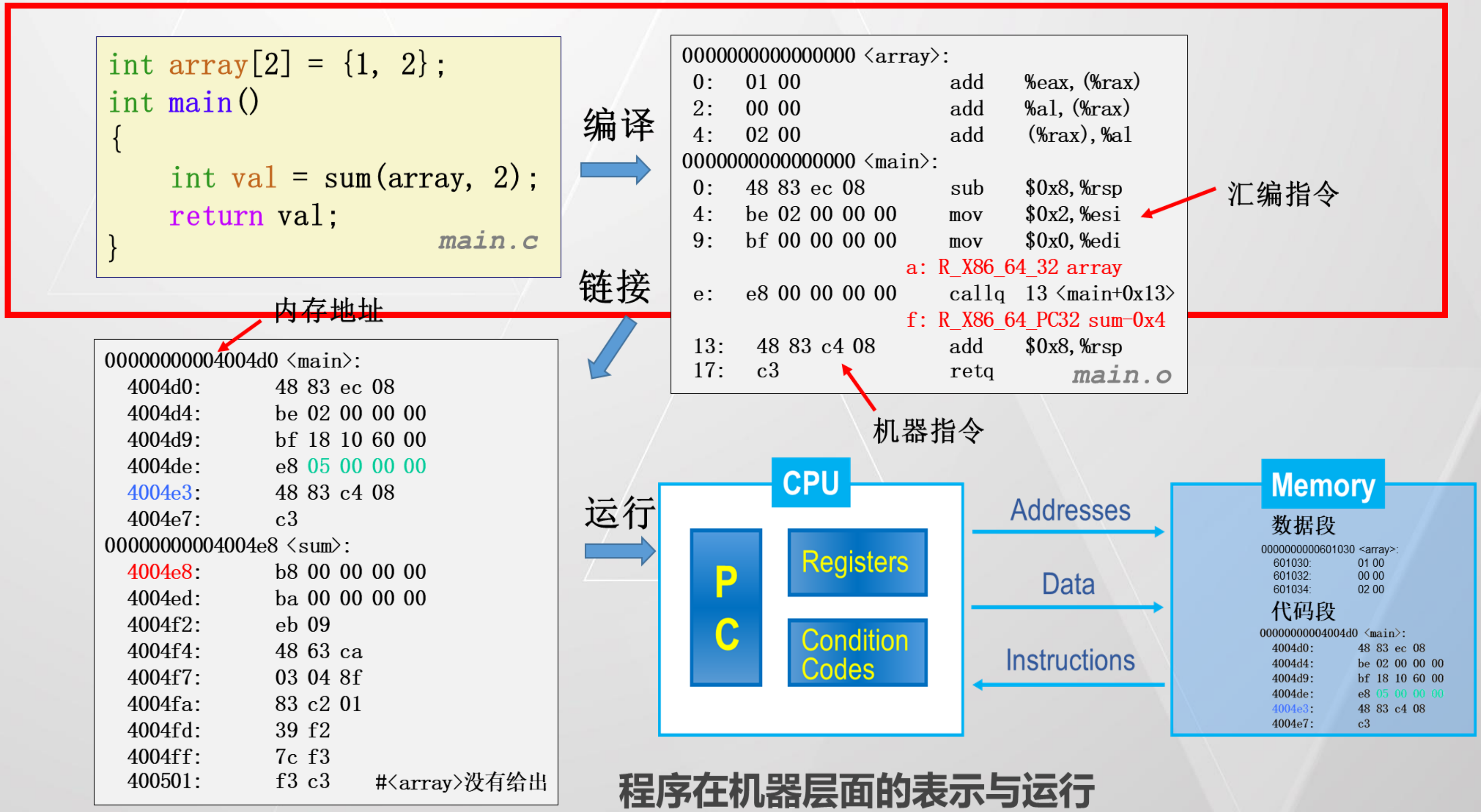
汇编语言程序设计

第6节 80X86汇编语言与C语言-2

C程序在硬件层面的表示

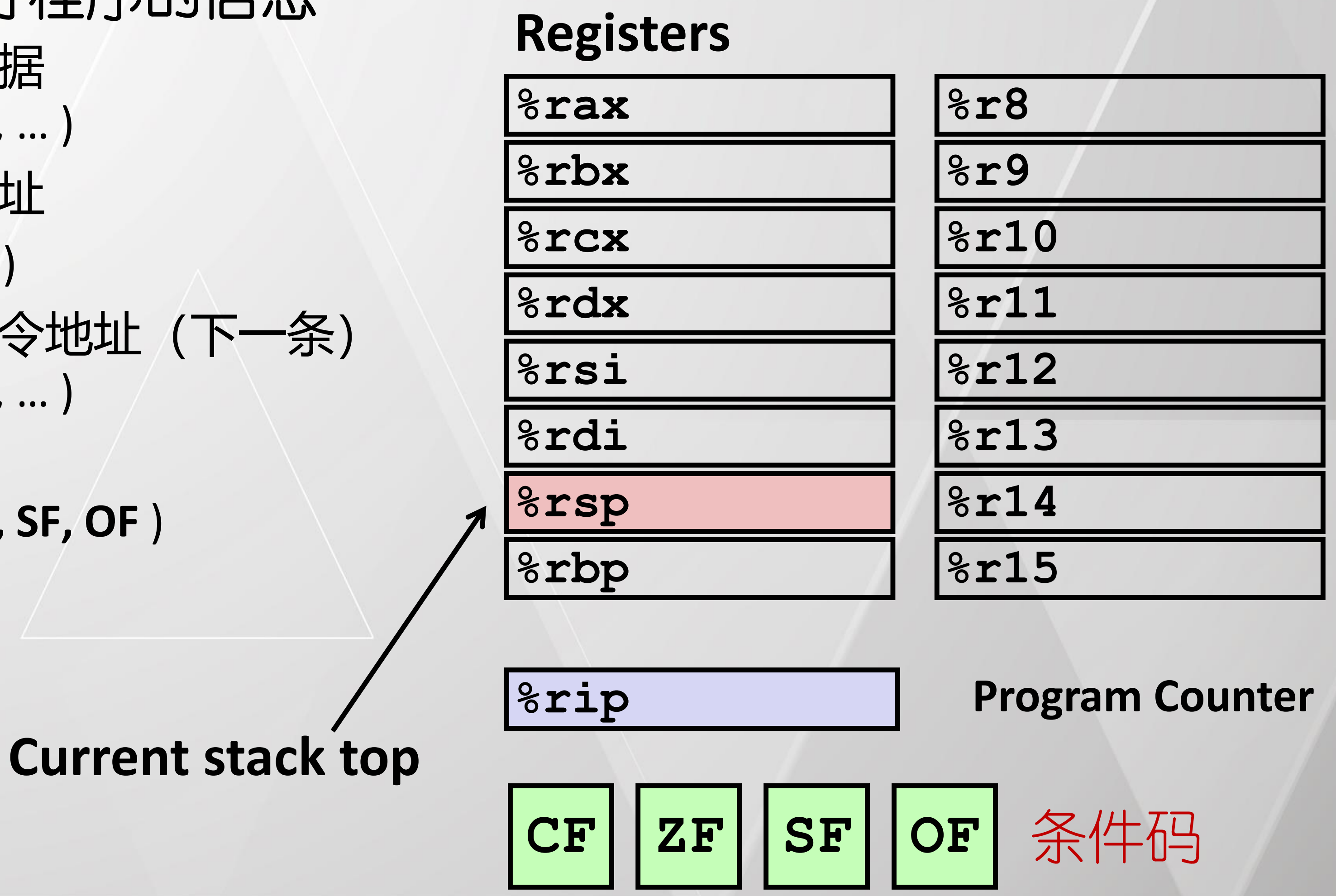
- 数据
  - 整数（第二讲）
  - 浮点数（第三讲）
  - 数组、结构（第八讲）
- 代码
  - 基本概念/基本指令/寻址方式（第五讲）
  - 程序控制流与相关指令（第六讲）
  - 函数调用与相关指令（第七讲）



程序在机器层面的表示与运行

处理器状态 (x86-64, 部分)

- 当前执行程序的信息
  - 临时数据
    (`%rax`, …)
  - 栈顶地址
    (`%rsp`)
  - 当前指令地址（下一条）
    (`%rip`, …)
  - 条件码
    (CF, ZF, SF, OF)



条件码（由指令隐式设置）

CF	Carry（进位）Flag	SF	Sign Flag
ZF	Zero Flag	OF	Overflow Flag

- 这些条件码由算术指令隐含设置

  addq *Src,Dest*　　　　**addl** *Src,Dest*

  类似的C语言表达式： t = a + t　　(a = Src, t = Dest)

  CF 进位标志
  - 可用于检测无符号整数运算的溢出

  ZF set if t == 0

  SF set if t < 0

  OF set if 补码运算溢出（即带符号整数运算）

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

leaq 指令不设置条件码

⊙比较（Compare）指令

`cmpq` *Src2,Src1*　　`cmpl` *Src2,Src1*

`cmpq b,a` 类似于计算`a-b`（但是不改变目的操作数）

CF set if carry out from most significant bit

- 可用于无符号数的比较

ZF set if `a == b`

SF set if `(a-b) < 0`

OF set if two’s complement overflow （补码计算溢出）

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

测试（Test）指令

`testq` *Src2,Src1*
`testl` *Src2,Src1*

- 计算*Src1* & *Src2*并设置相应的条件码，但是不改变目的操作数

- ZF set when a&b == 0
- SF set when a&b < 0

test指令使CF，OF为0

读取条件码

- **Set*X* 指令**

读取当前的条件码（或者某些条件码的组合），并存入目的字节寄存器

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)

如果补码计算溢出，则**OF**置**1**

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

x86-64 通用寄存器（低8位可独立访问）

%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

⊙SetX 指令

读取当前的条件码（或者某些条件码的组合），并存入目的“字节”寄存器

- 余下的7个字节不会被修改
- 通常使用“ movzbl”指令对目的寄存器进行“0”扩展

```
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
```

“64-bit operands generate a 64-bit result in the destination general-purpose register. 32-bit operands generate a 32-bit result, zero-extended to a 64-bit result in the destination general-purpose register.”

摘自”Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 1: Basic Architecture”

跳转指令

- jX 指令

依赖当前的条件码选择下一条执行语句（是否顺序执行）

jX	Condition	Description
`jmp`	`1`	Unconditional
`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)

条件跳转示例 (Old Style)

- **Generation**

```
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;
}
```

```
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

Goto风格表示

- C allows `goto` statement
- 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;
}
```

条件表达式

C Code

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

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

Goto Version

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

- Create separate code regions for *then* & *else* expressions
- Execute appropriate one

条件移动指令

- 条件移动
  - 语义： if (Test) Dest ← Src
  - Supported in post-1995 x86 processors
  - GCC tries to use them
    - But, only when known to be safe
- Why?
  - 条件跳转指令对于现代流水线处理器的执行效率有很大的负面影响
  - 条件移动指令可以避免这一现象

C Code

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

Goto Version

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

条件移动指令示例

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

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

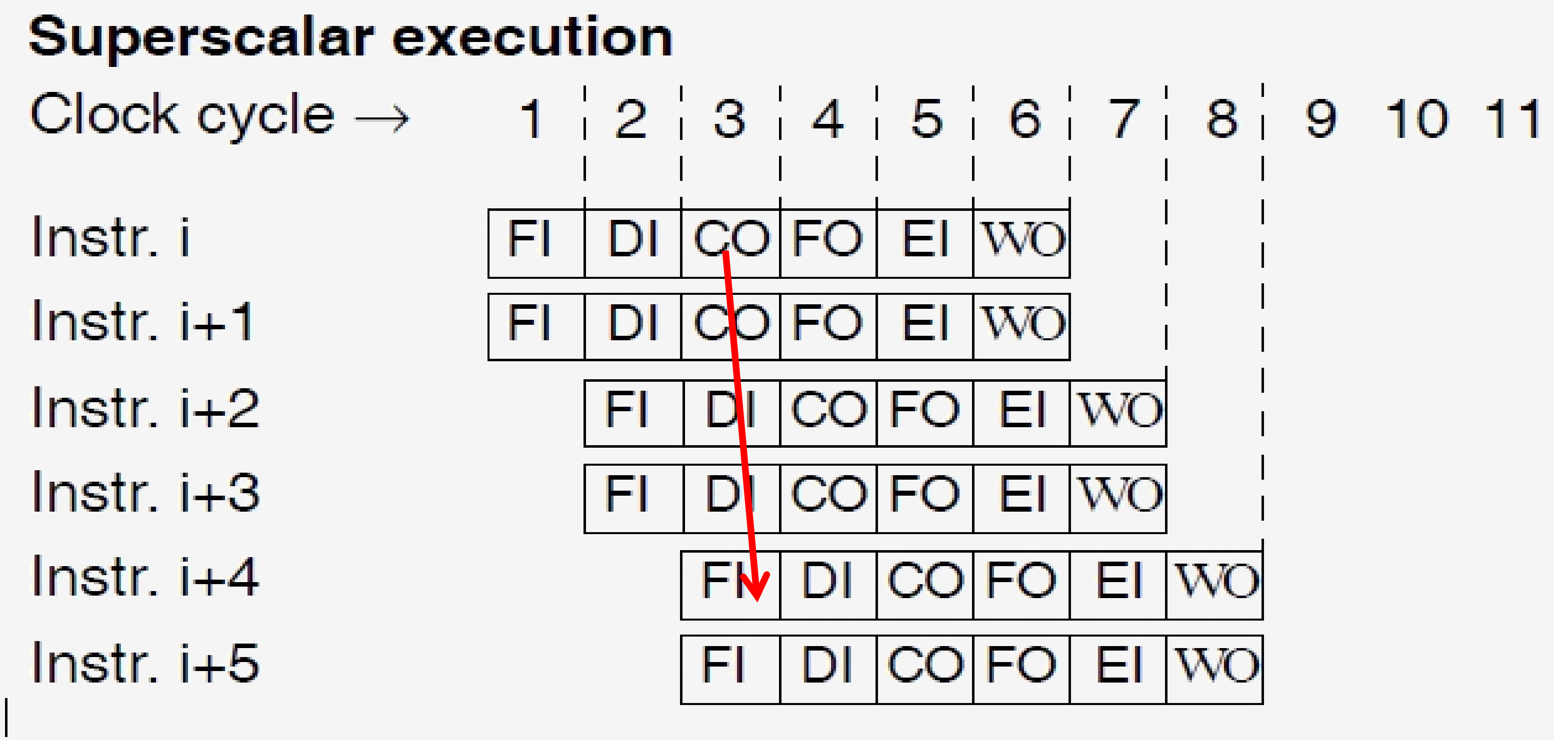
```
absdiff:
   movq     %rdi, %rax   # x
   subq     %rsi, %rax   # result = x-y
   movq     %rsi, %rdx
   subq     %rdi, %rdx   # eval = y-x
   cmpq     %rsi, %rdi   # x:y
   cmovle   %rdx, %rax   # if <=, result = eval
   ret
```

Δ微体系结构背景*

- 处理器流水线（五级流水示例）
  - Instruction Fetch (IF)
  - Read Registers (RD)
  - Arithmetic Operation (ALU)
  - Memory Access (MEM)
  - Write Back (WB)

Δ微体系结构背景*

现代的通用处理器 支持深度流水线以及多发射结构，如Pentium 4： >= 20 stages, up to 126 instructions on-fly



条件跳转指令往往会引起一定的性能损失，因此需要尽量消除。

条件转移指令的局限性

```
val  = Then-Expr;
vale = Else-Expr;
val  = vale if !Test;
```

```
int xgty  = 0, xltey = 0;

int absdiff_se(
    int x, int y)
{
    int result;
    if (x > y) {
        xgty++;  result = x-y;
    } else {
        xltey++; result = y-x;
    }
    return result;
}
```

- 限制使用的场合:
  - Then-Expr 或Else-Expr 表达式有“副作用”
  - Then-Expr 或 Else-Expr 表达式的计算量较大

主观题 0.5分

设置

使用条件移动指令来完成以下功能。

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

是否可以用如下汇编代码段来完成?

```
   Invalid implementation of function cread
   xp in register %edx
1      movl    $0, %eax          Set 0 as return value
2      testl   %edx, %edx        Test xp
3      cmovne  (%edx), %eax      if !0, dereference xp to get return value
```

作答

```c
int cread_alt(int *xp) {
    int t = 0;
    return *(xp ? xp : &t);
}
```

编译时加上-fif-conversion -Og

```
_cread_alt:
    …
    movl    $0, -4(%rsp)      #t=0
    leaq    -4(%rsp), %rax    #&t
    testq   %rdi, %rdi
    cmove   %rax, %rdi
    movl    (%rdi), %eax
    ret
```

“Do-While” 循环示例

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;
}
```

- Count number of 1’s in argument **x**
- Use conditional branch to either continue looping or to exit loop

"Do-While" 循环的编译后代码

Goto Version

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

Register	Use(s)
%rdi	Argument **x**
%rax	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
    ret
```

通用的“Do-While” 转换

C Code

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

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

Goto Version

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

通用的“While” 转换-1

- “Jump-to-middle” translation
- Used with `-Og`

While version

```
while (Test)
  Body
```

Goto Version

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

While 循环示例-1

C Code

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

Jump to Middle

```c
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;
}
```

- Compare to do-while version of function
- Initial goto starts loop at test

通用的“While” 转换-2

While version

```
while (Test)
  Body
```

Do-While Version

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

- “Do-while” conversion
- Used with `-O1`

Goto Version

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

While循环示例-2

C Code

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

Do-While Version

```c
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;
}
```

- Compare to do-while version of function
- Initial conditional guards entrance to loop

"For" 循环的形式

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;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

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

“For” 循环→ While 循环

For Version

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

While Version

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

Switch语句

- 依据不同情况来采用不同的实现技术
  - 使用一组if-then-else语句来实现
  - 使用跳转表

Switch语句示例

```
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;
}
```

- **Multiple case labels**
  - Here: 5 & 6
- **Fall through cases**
  - Here: 2
- **Missing cases**
  - Here: 4

跳转表

Switch Form

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

Jump Table

jtab:

Targ0
Targ1
Targ2
• • •
Targn-1

Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

Targ2: Code Block 2

• • •

Targn-1: Code Block *n*–1

Translation (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;
}
```

Setup:

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

What range of values takes default?

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

Note that `w` not initialized here

Switch语句示例

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

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
```

Setup:

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

Indirect jump

跳转表的构建与访问

- **Table Structure**
  - Each target requires 8 bytes
  - Base address at `.L4`
- **Jumping**
  - Direct: `jmp .L8`
  - Jump target is denoted by label `.L8`
  - Indirect: `jmp *.L4(,%rdi,8)`
  - Start of jump table: `.L4`
  - Must scale by factor of 8 (addresses are 8 bytes)
  - Fetch target from effective Address `.L4 + x*8`
    - Only for $0 \le x \le 6$

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
```

Jump Table

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;
}
```

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

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;
```

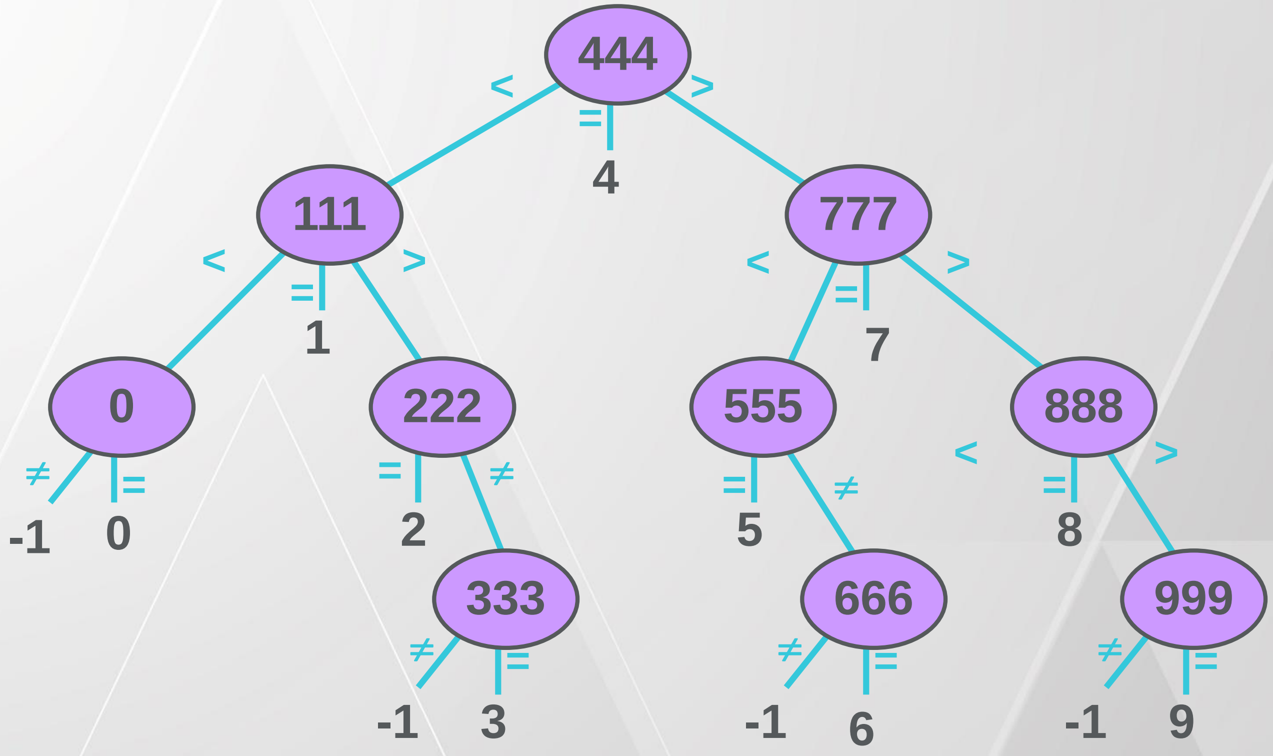
Code Blocks (x == 2, x == 3)

Signed divide RDX:RAX by *r/m64*, with result stored in RAX ← Quotient, RDX ← Remainder.

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

```
.L5:                         # Case 2
   movq     %rsi, %rax
   cqto                      # sign extend rax to
                             # rdx:rax
   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



- 以二叉树的结构组织，提升性能

小结

- 条件码
  - 设置
  - 读取
  - 条件跳转指令
  - 条件传送指令
- 程序控制流
  - If-then-else
  - 循环结构
    - Do-while
    - While
    - for
  - switch语句