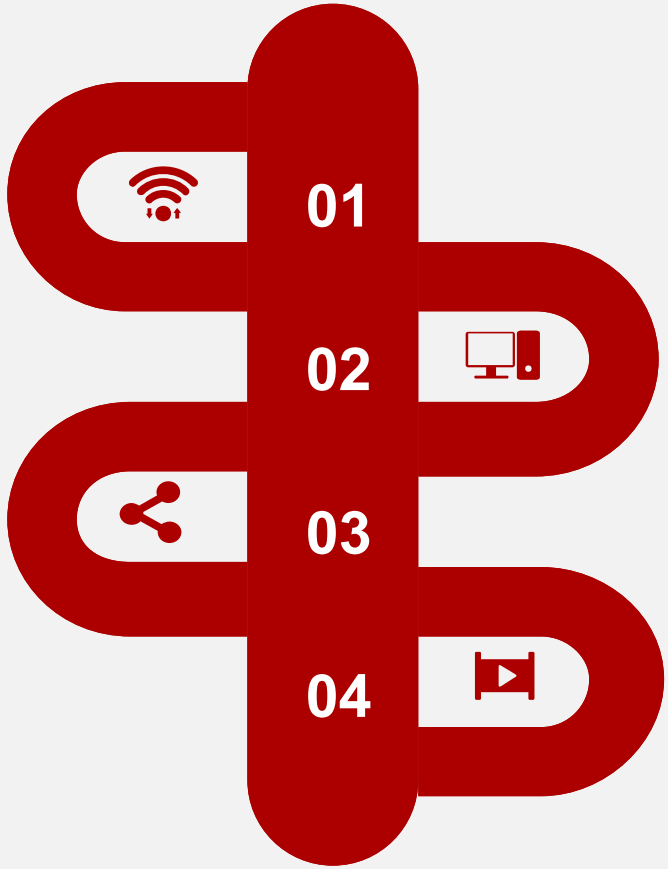
计算机系统
程序的机器级表示：控制

湖南大学

《计算机系统》课程教学组

内容提要



控制：条件码

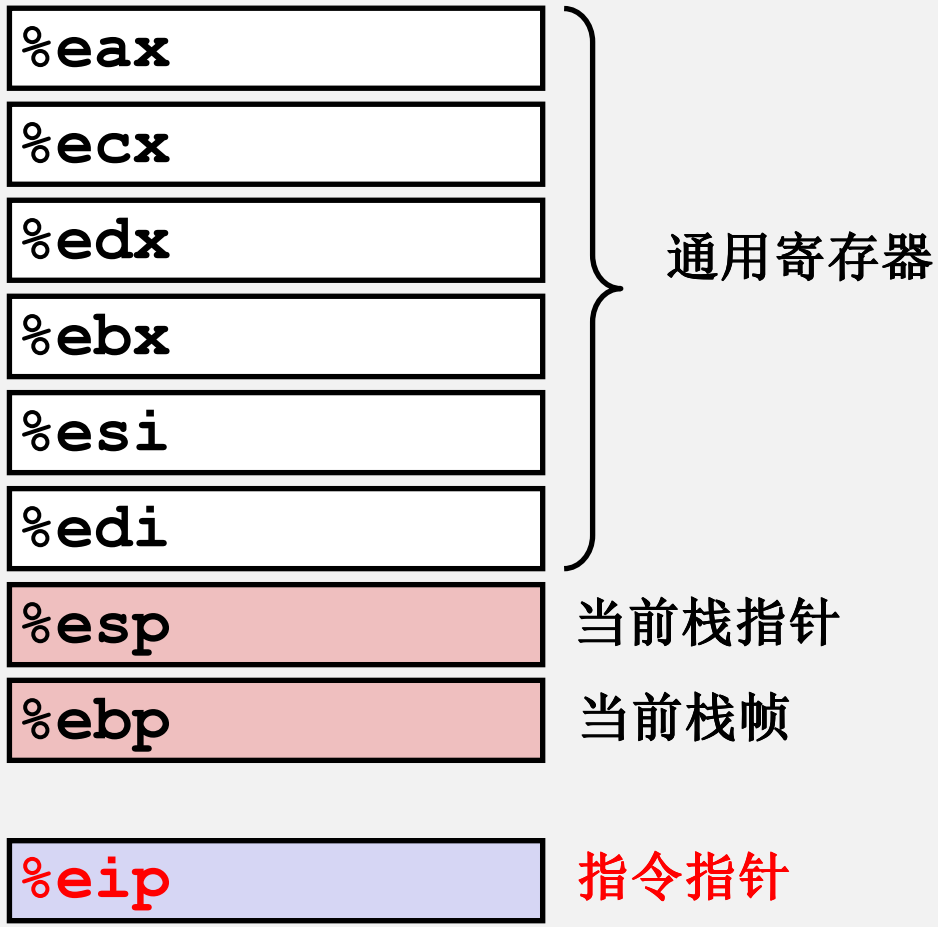
条件分支

循环

switch语句

处理器状态

- **当前运行程序的相关信息**
  - 临时数据
    (%eax, …)
  - 运行栈帧的地址
    (%ebp,%esp)
  - 即将要执行的指令地址
    (%eip, …)
  - **标志位**
    (CF, ZF, SF, OF)



条件码

- **每个条件码占一个bit**
  - **CF** 最高位产生了进位，无符号操作数的溢出
  - **SF** 符号标志，操作结果为负数
  - **ZF** 零标志
  - **OF** 溢出标志 (有符号数 signed)

例如: **`addl/addq`** *Src*, *Dest* ↔ **`t = a+b`**

CF set ，如果t溢出

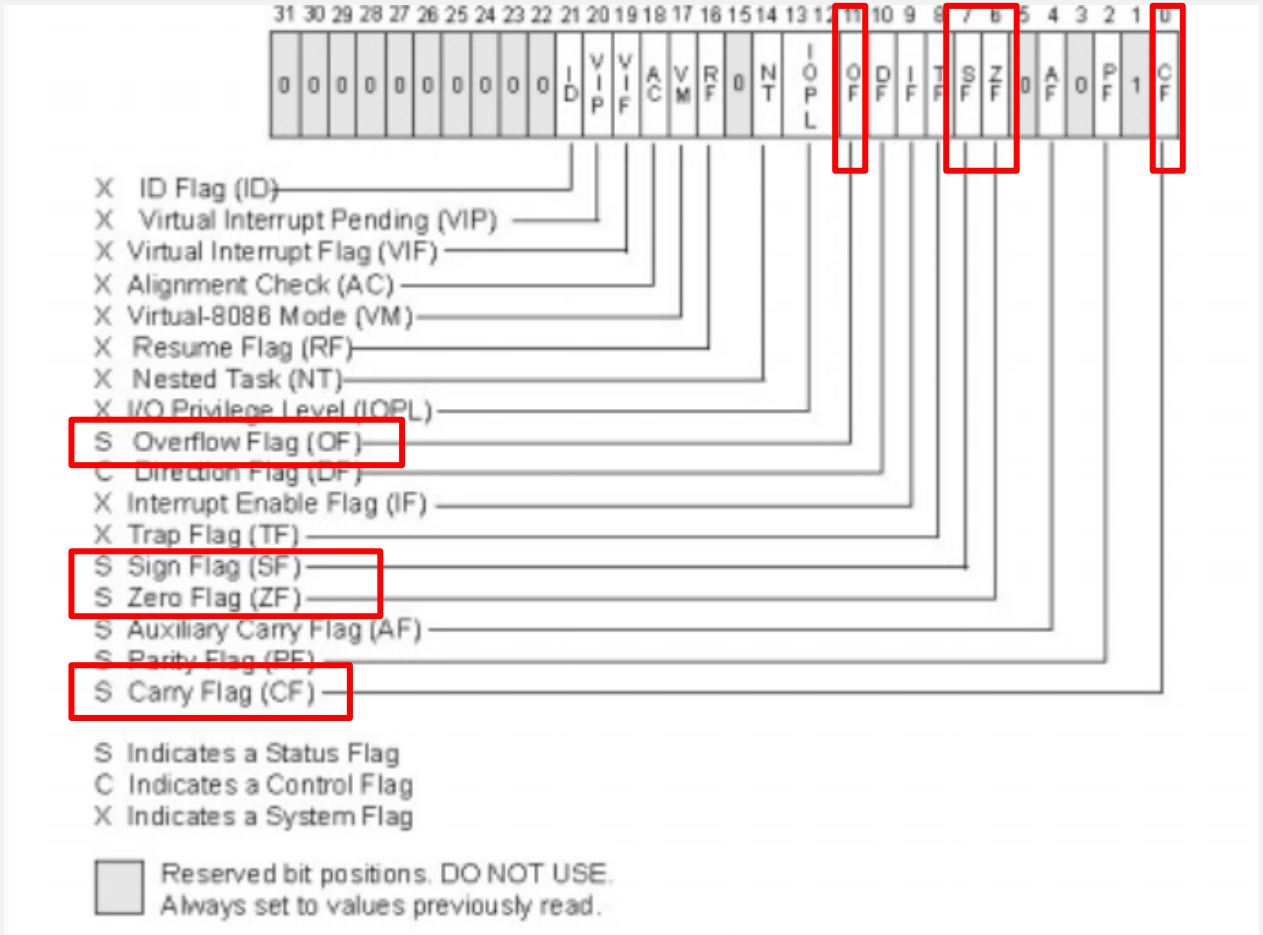
ZF set ，如果 **`t == 0`**

SF set ，如果 **`t < 0`** (as signed)

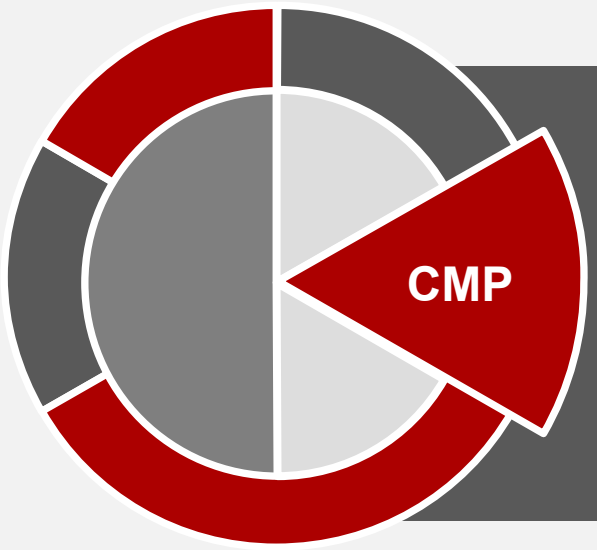
OF set，如果有符号数溢出

`lea/mov` 指令不设置条件码

条件码寄存器


31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)
S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
Reserved bit positions. DO NOT USE.
Always set to values previously read.

条件码设置



`cmpl` *Src, Dest;* *Dest – Src,* 影响标志位

`cmpl` `b,a` 等价于计算 `a-b`，但不改变a与b的值。

CF set	无符号数运算时有进位
ZF set	如果 `a == b`
SF set	如果 有符号数`(a-b)< 0`
OF set	如果有符号数运算溢出

条件码设置

以四个bit的数为例,有符号数能表示-8~7（ 1000~0111 ），无符号数能表示0~15（0000~1111）

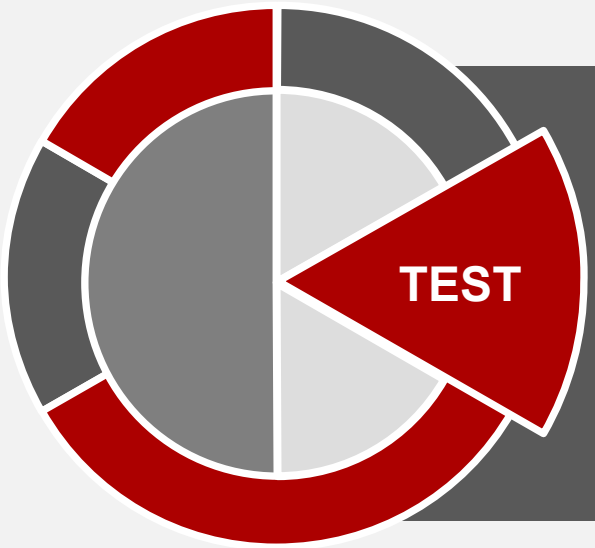
6 >3	6-3=3,	SF=0,	OF=0，	SF⊕OF=0
4 >-2	4-(-2)=6,	SF=0,	OF=0	
-2 <-4	-2-(-4)=2,	SF=0,	OF=0	
6 <7	6-7=-1,	SF=1,	OF=0,	SF⊕OF=1
-4 >-6	-4-(-6)=2,	SF=0,	OF=0	
-4 < 7	-4-7=-11=10101=0101(正),	SF=0,	OF=1,	SF⊕OF=1
7 >-5	7-(-5)=12=01100=1100(负),	SF=1,	OF=1,	SF⊕OF=0

当出现：

正+正=负、负+负=正、正-负=负、负-正=正

的情况，就是产生了溢出，OF=1

条件码设置



testl/testq Src, Dest；Dest & Src，影响标志位

testl b,a 等价于计算a&b（但不改变a或b的值）

ZF set	如果 a&b == 0
SF set	如果 a&b < 0

条件码设置

SetX 指令：根据条件码的组合将一个字节设置为0或1

	SetX	Condition	Description
	sete/setz	ZF	Equal / Zero
	setne/setnz	~ZF	Not Equal / Not Zero
	sets	SF	Negative
	setns	~SF	Nonnegative
有符号数	setg	~(SF^OF)&~ZF	Greater (Signed)
有符号数	setge	~(SF^OF)	Greater or Equal (Signed)
有符号数	setl	(SF^OF)	Less (Signed)
有符号数	setle	(SF^OF)|ZF	Less or Equal (Signed)
无符号数	seta	~CF&~ZF	Above (unsigned)
无符号数	setb	CF	Below (unsigned)

设置条件码

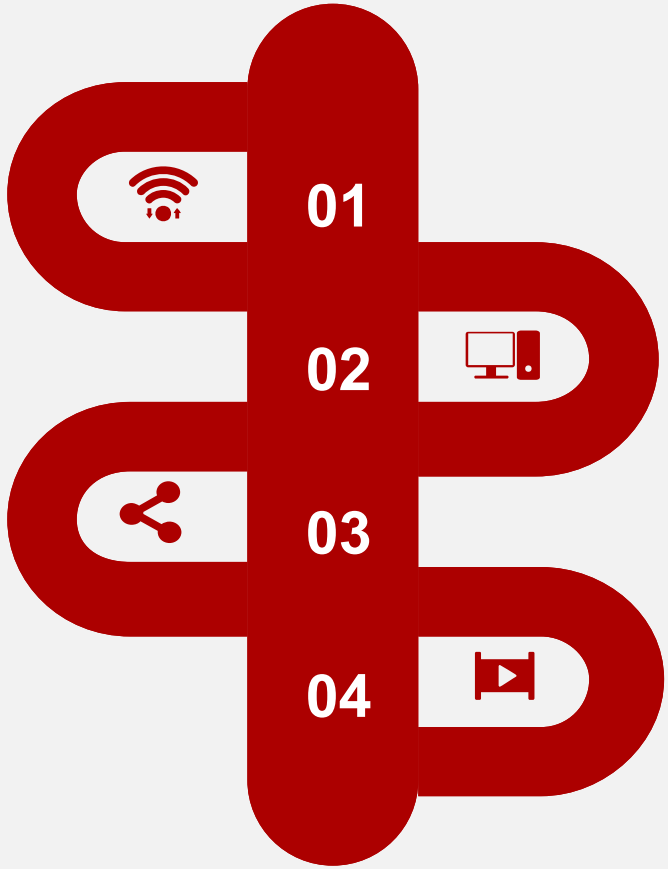
```
int gt (int x, int y)
{
  return x > y;
}
```

Body

```
movl 12(%ebp),%eax # eax = y
cmpl %eax,8(%ebp)  # Compare x : y
setg %al           # al = x > y
movzbl %al,%eax    # Zero rest of %eax
```

%eax	%ah	%al

内容提要



控制：条件码

条件分支

循环

switch语句

跳转指令

jX 指令： 根据不同的条件跳转到某条指令处执行

jX	Condition	Description
jmp	1	无条件跳转
je/jz	ZF	Equal / Zero
Jne/jnz	~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)

有符号数

无符号数

条件跳转

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

```
    absdiff:
    pushl   %ebp                    } Setup
    movl    %esp, %ebp              }
%edx=xmovl  8(%ebp), %edx           } Body1
%eax=ymovl  12(%ebp), %eax          }
    cmpl    %eax, %edx              }
    jle     .L6                     } Body2a
    subl    %eax, %edx              }
    movl    %edx, %eax              }
    jmp     .L7                     }
    .L6:                            } Body2b
    subl    %edx, %eax              }
    .L7:                            }
    popl    %ebp                    } Finish
    ret                             }
```

条件跳转

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
  goto Exit;
Else:
  result = y-x;
Exit:
  return result;
}
```

- C 中可以采用goto语句进行跳转，与机器级的语言风格类似
- 但通常被认为是一种比较“low”的编程风格

```
        absdiff:
        pushl   %ebp                    } Setup
        movl    %esp, %ebp              }
%edx=x  movl    8(%ebp), %edx           } Body1
%eax=y  movl    12(%ebp), %eax          }
        cmpl    %eax, %edx              }
        jle     .L6                     } Body2a
        subl    %eax, %edx              }
        movl    %edx, %eax              }
        jmp     .L7                     }
        .L6:                            } Body2b
        subl    %edx, %eax              }
        .L7:                            }
        popl    %ebp                    } Finish
        ret                             }
```

分支跳转

C Code

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

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

等价

Goto Version

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

Test 是一个返回整数值的表达式

= 0 逻辑假

≠ 0 逻辑真

- 为每一个分支都产生一段代码
- 根据条件执行合适的代码段

条件传送

条件传送指令——满足条件才传送

Instruction supports:

if (Test) Dest← Src

- **先计算一个条件操作的两种结果，然后根据条件选择某一个**
- **优势：能够更好的匹配现代处理器的特性**
  - **流水线**
  - **分支预测**

C Code

```
tval = Then_Expr;
result = Else_Expr;
t = Test;
if (t) result = tval;
return result;
```

条件传送

C Code

```
int comvdiff(int x, int y)
{
  int tval = y-x;
  int rval = x-y;
  int test = x < y;
  if (test) rval = tval;
  result rval;
}
```

条件传送

```
comvdiff:
movl    8(%ebp), %ecx
movl    12(%ebp), %edx
movl    %edx, %ebx
subl    %ecx, %ebx
movl    %ecx, %eax
subl    %edx, %eax
cmpl    %edx, %ecx
cmovl   %ebx, %eax
```

避免了跳转指令

- CPU无需做分支预测，避免预测错误的代价
- 流水线效率更高

条件跳转

```
absdiff:
pushl   %ebp
movl    %esp, %ebp
movl    8(%ebp), %edx
movl    12(%ebp), %eax
cmpl    %eax, %edx
jle     .L6
subl    %eax, %edx
movl    %edx, %eax
jmp .L7
.L6:
subl %edx, %eax
.L7:
popl %ebp
ret
```

条件传送

C 代码

```
int absdiff(int a, int b)
{
        return a>b ? a-b : b-a;
}
```

非优化编译：gcc -S test.c：

```
        movl    8(%ebp), %eax
        cmpl    12(%ebp), %eax
        jle     .L2
        movl    12(%ebp), %eax
        movl    8(%ebp), %edx
        movl    %edx, %ecx
        subl    %eax, %ecx
        movl    %ecx, %eax
        jmp     .L3
.L2:
        movl    8(%ebp), %eax
```

汇编代码出现了jle这样的跳转指令。对于使用了流水线的CPU，这样的跳转是存在隐患的（P140），分支预测失败就会刷新掉所有流水线中取到而未执行的指令，影响运行性能。

优化：gcc -S O1 test.c

```
movl    8(%esp), %ecx
movl    12(%esp), %edx
movl    %ecx, %eax
subl    %edx, %eax
movl    %edx, %ebx
subl    %ecx, %ebx
cmpl    %edx, %ecx
cmovle  %ebx, %eax
popl    %ebx
```

经过优化后的代码没有了跳转指令，取而代之的是一个条件传送指令——cmovle。使得控制流不依赖于数据，流水线也更容易保持满状态。

条件传送

计算代价

- 两个计算过程都需要运行
- 一般来说，只有两个计算过程都比较简单的时候，才能够发挥优势

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

非法操作

在p为0的时候，仍然会去引用 *p，从而产生非法操作

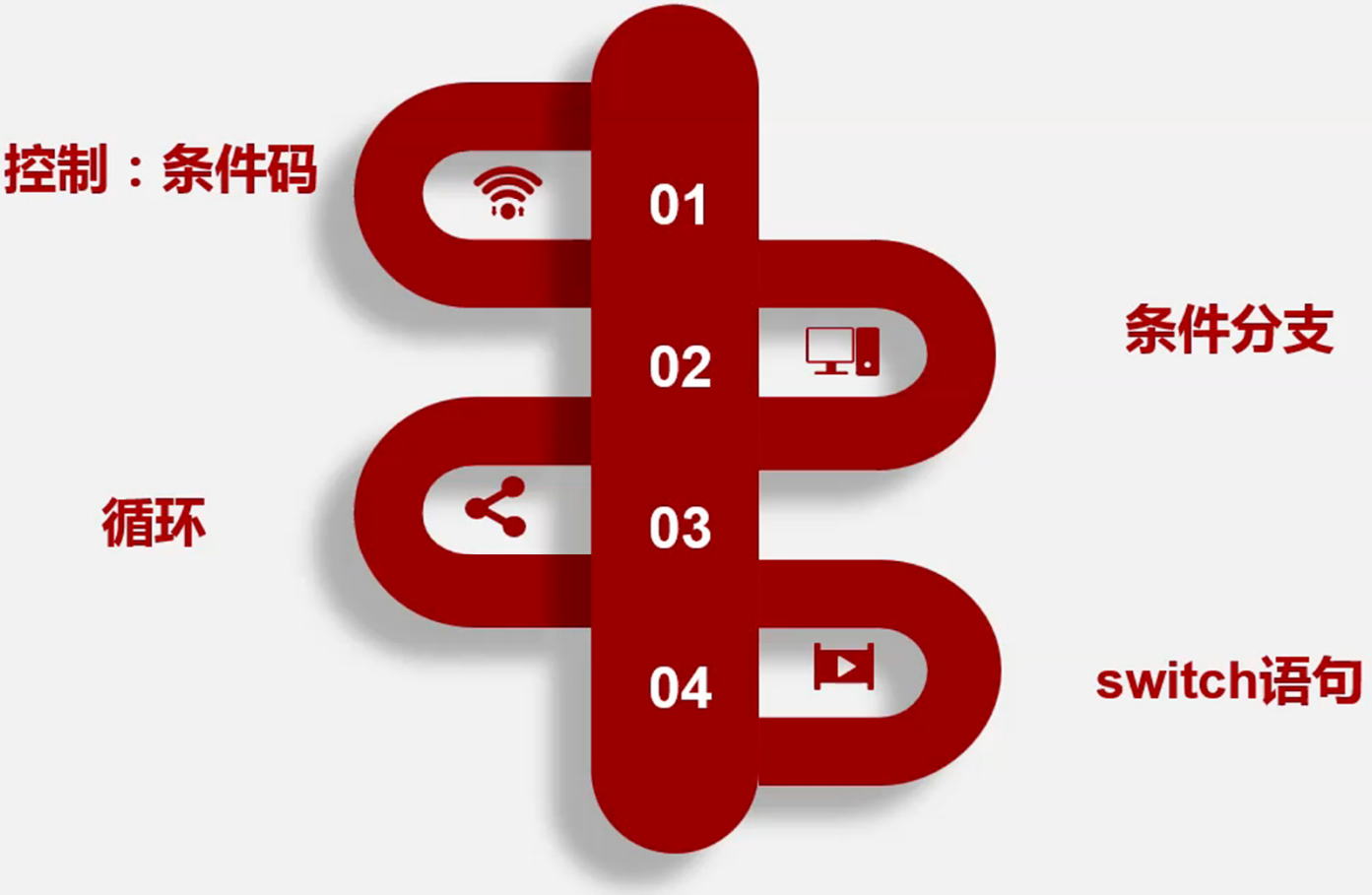
```
val = p? *p : 0;
```

副作用

- 两个表达式都进行了计算
- 产生了意料之外的赋值过程

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

内容提要



do-while 循环

C Code

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

寄存器

%edx x

%ecx result

- 计算x中有多少个1(“popcount”)
- 使用条件跳转指令来进行条件判断

```
movl    $0,%ecx       # result = 0
.L2:                  # loop:
movl    %edx,%eax
andl    $1,%eax       # t = x & 1
addl    %eax,%ecx     # result += t
shrl    %edx          # x >>= 1
jne     .L2           # If !0, goto loop
```

while 循环

C Code for while loop

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

C code for do loop

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

While 和 do-while二者的代码是否完全一致?

- 都是条件测试失败退出循环
- do-while循环至少执行一次循环体

for 循环

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;
}
```

各循环对比

For Version

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

While Version

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

Do-while Version

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

Goto Version

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

内容提要

switch语句

```
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;
}
```

- 多重分支
- 使用跳转表
- x = 2时无break

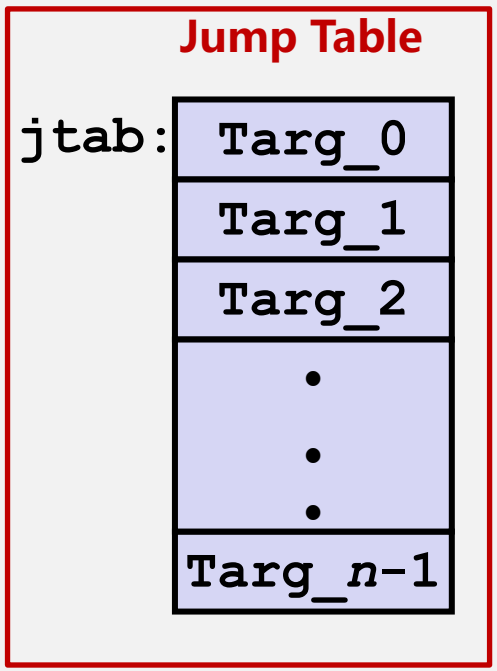
跳转表

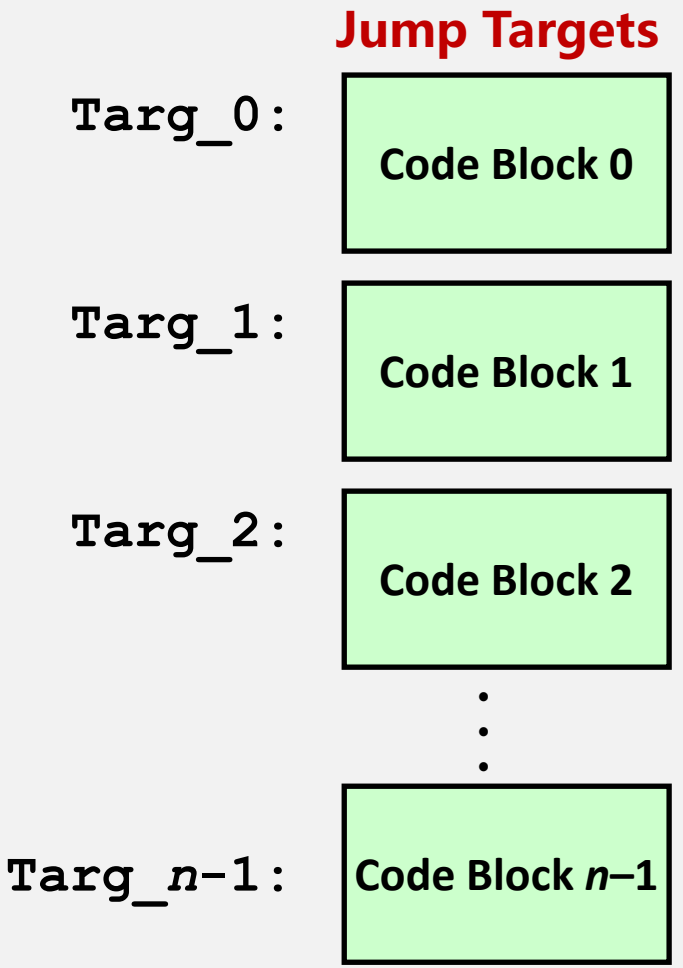
Switch Form

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

Approximate Translation

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

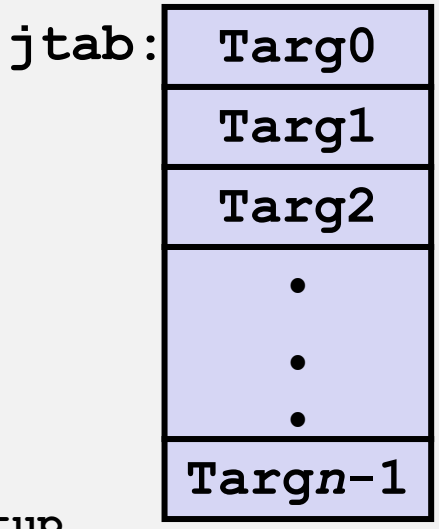




switch语句

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

Jump Table



Jump table

```
.section  .rodata
  .align 4
.L4:
  .long    .L8 # x = 0
  .long    .L3 # x = 1
  .long    .L5 # x = 2
  .long    .L9 # x = 3
  .long    .L8 # x = 4
  .long    .L7 # x = 5
  .long    .L7 # x = 6
```

setup:

```
switch_eg:  注意：此处W没有进行初始化
pushl       %ebp              # Setup
movl        %esp,%ebp         # Setup
movl        8(%ebp),%eax      # %eax = x
cmpl        $6,%eax           # Compare x:6
ja          .L8               # If unsigned '>' goto default
jmp         *.L4(,%eax,4)     # Goto *JTab[x]
```

跳转表结构

- **每个跳转地址需要4字节**
- **基址在 .L4**

跳转

- Direct：**jmp .L2**
- 直接跳转，地址为.L2
- Indirect：**jmp *.L4(,%eax,4)**
- 跳转表基地址：**.L4**
- **4**是因为每个地址占4个字节
- 取到有效地址.L4+ eax*4
  - $0 \le x \le 6$

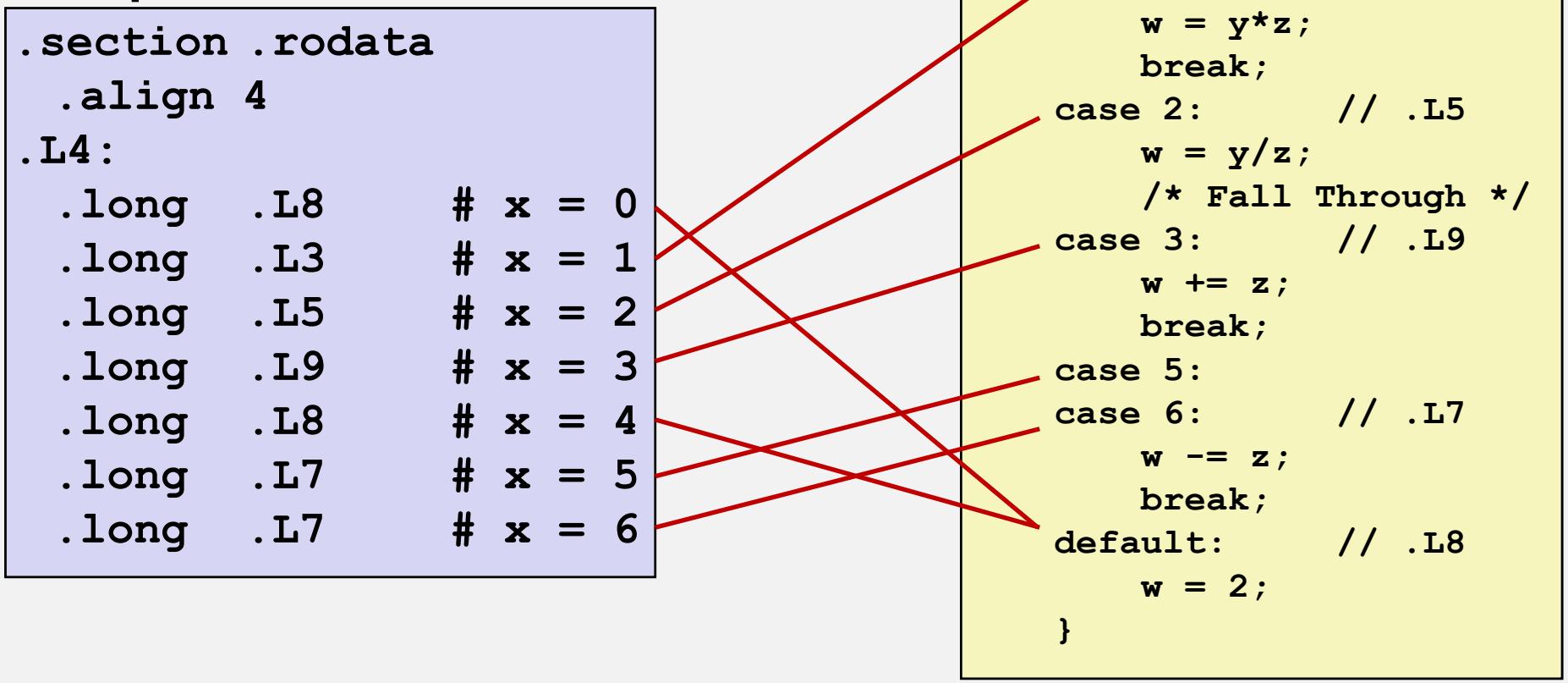
Jump Table

```
.section .rodata
  .align 4
.L4:
  .long   .L8      # x = 0
  .long   .L3      # x = 1
  .long   .L5      # x = 2
  .long   .L9      # x = 3
  .long   .L8      # x = 4
  .long   .L7      # x = 5
  .long   .L7      # x = 6
```

跳转表

Jump Table

```
.section .rodata
  .align 4
.L4:
  .long   .L8      # x = 0
  .long   .L3      # x = 1
  .long   .L5      # x = 2
  .long   .L9      # x = 3
  .long   .L8      # x = 4
  .long   .L7      # x = 5
  .long   .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;
}
```

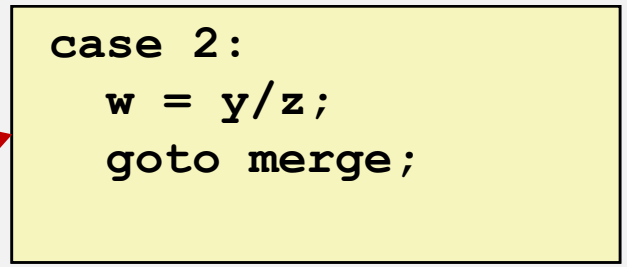
代码块 1

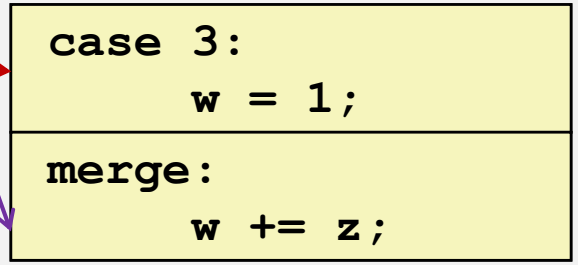
```
switch(x) {
case 1:    // .L3
     w = y*z;
     break;
. . .
}
```

```
.L3:  # x == 1
  movl   12(%ebp),%eax   # y
  imull  16(%ebp),%eax   # w = y*z
  jmp    .L2             # Goto done
```

Fall Through

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





代码块 2&3

```
long w = 1;
   . . .
switch(x) {
   . . .
case 2:          // .L5
   w = y/z;
   /* Fall Through */
case 3:        // .L9
   w += z;
   break;
   . . .
}
```

```
.L5:                              # x == 2
  movl  12(%ebp),%eax             # y
  cltd
  idivl 16(%ebp)                  # y/z
  jmp   .L6
.L9:                              # x == 3
  movl  $1, %eax                  # w = 1
.L6:                              # merge:
  addl  16(%ebp),%eax             # +=z
  jmp   .L2                       # Goto
done
```

代码块
5&6&default

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

```
.L7:
                                # x == 5,6
   movl $1,%eax                 # w = 1
   subl 16(%ebp),%eax           # w = 1-z
   jmp    .L2                   # goto done
.L8:                            # default
   movl   $2,%eax               # w = 2
.L2:                            # done
```

代码块：结束

```
return w;
```

```
.L2:                    # done:
   popl %ebp
   ret
```

使用跳转表是一种非常有效的实现多重分支的方法

目标代码

准备阶段

- Label `.L8` becomes address `0x80484b8`
- Label `.L4` becomes address `0x8048680`

Assembly Code

```
switch_eg:
  . . .
  ja     .L8                    # If unsigned >goto default
  jmp   *.L4(,%eax,4)           # Goto *JTab[x]
```

Disassembled Object Code

```
08048480 <switch_eg>:
. . .
8048489:   77 2d                          ja    80484b8 <switch_eg+0x38>
804848b:   ff 24 85 80 86 04 08           jmp   *0x8048680(,%eax,4)
```

目标代码：跳转表

跳转表

- 在反汇编代码中无法直接看到，可以通过 GDB来观察
- **gdb switch**
- **(gdb) x/7xw 0x8048680**
  - Examine 7 hexadecimal format “words” (4-bytes each)—**x/7xw**

```
0x8048680:  0x080484b8  0x08048492  0x0804849b  0x080484a4
0x8048690:  0x080484b8  0x080484ae  0x080484ae
```

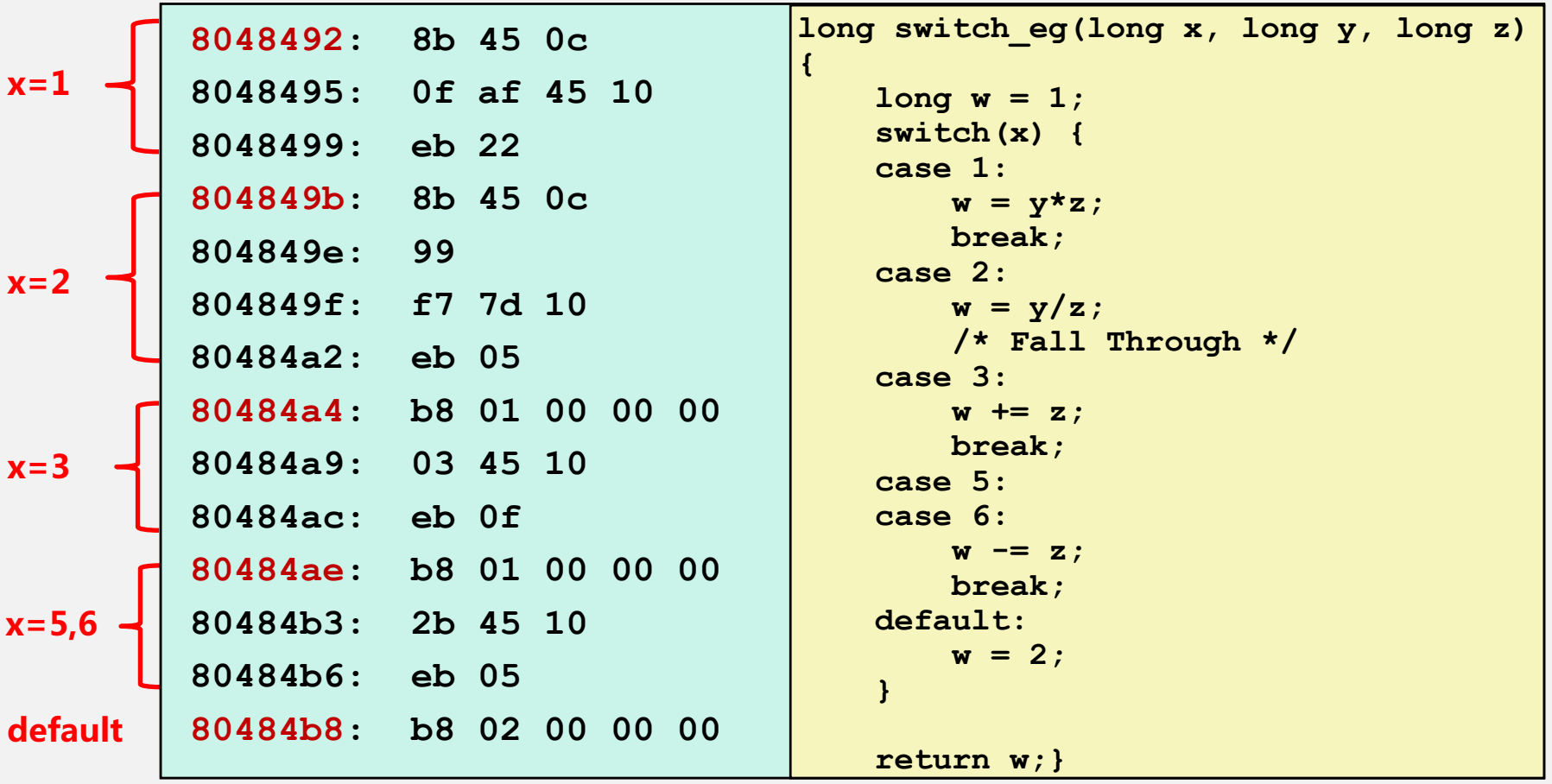
跳转表解释

```
0x8048680:   0x080484b8   0x08048492   0x0804849b   0x080484a4
0x8048690:   0x080484b8   0x080484ae   0x080484ae
```

Address	Value	x
0x8048680	0x80484b8	0
0x8048684	0x8048492	1
0x8048688	0x804849b	2
0x804868c	0x80484a4	3
0x8048690	0x80484b8	4
0x8048694	0x80484ae	5
0x8048698	0x80484ae	6

```
.section   .rodata
  .align 4
.L4:
  .long     .L8 # x = 0
  .long     .L3 # x = 1
  .long     .L5 # x = 2
  .long     .L9 # x = 3
  .long     .L8 # x = 4
  .long     .L7 # x = 5
  .long     .L7 # x = 6
```

反汇编


x=1
8048492: 8b 45 0c
8048495: 0f af 45 10
8048499: eb 22
x=2
804849b: 8b 45 0c
804849e: 99
804849f: f7 7d 10
80484a2: eb 05
x=3
80484a4: b8 01 00 00 00
80484a9: 03 45 10
80484ac: eb 0f
x=5,6
80484ae: b8 01 00 00 00
80484b3: 2b 45 10
80484b6: eb 05
default
80484b8: b8 02 00 00 00
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;}

反汇编

Value
0x80484b8
0x8048492
0x804849b
0x80484a4
0x80484b8
0x80484ae
0x80484ae

```
8048492:   8b 45 0c         mov  0xc(%ebp),%eax
8048495:   0f af 45 10      imul 0x10(%ebp),%eax
8048499:   eb 22            jmp  80484bd
804849b:   8b 45 0c         mov  0xc(%ebp),%eax
804849e:   99               cltd
804849f:   f7 7d 10         idivl 0x10(%ebp)
80484a2:   eb 05            jmp  80484a9
80484a4:   b8 01 00 00 00   mov  $0x1,%eax
80484a9:   03 45 10         add  0x10(%ebp),%eax
80484ac:   eb 0f            jmp  80484bd
80484ae:   b8 01 00 00 00   mov  $0x1,%eax
80484b3:   2b 45 10         sub  0x10(%ebp),%eax
80484b6:   eb 05            jmp  80484bd
80484b8:   b8 02 00 00 00   mov  $0x2,%eax
```

下一节：过程

湖南大学

《计算机系统》课程教学组