程序的机器级表示（4）

谢旻晖

2025.10

Control

Outline

- **Jump instructions**
  - PC-relative Address
- Translate Control constructs in C to assembly
  - Goto、Branch、Loop
- Conditional Move
- Translate switch in C to assembly

Control

- Two of the most important parts of program execution
  - Data flow (Accessing and operating data)
  - Control flow (control the sequence of operations)

Control

- Sequential execution is default
  - the instructions are executed in the order they appear in the program
- Chang the control flow
  - Jump instructions
  - 分支语句对性能有一定影响，分支预测失败会有较大的惩罚（根源在于内存访问速度远低于CPU运算速度）

Jump Instructions

```
1     xorl   %eax, %eax          Set %eax to 0
2     jmp    .L1                 Goto .L1
3     movl   (%eax), %edx        Null pointer dereference
4  .L1:
5     popl   %edx
```

Unconditional jump

- Jumps unconditionally
- Direct jump: jmp label
  - jmp .L
- Indirect jump: jmp *Operand
  - jmp *%eax
  - jmp *(%eax)

Conditional jump

- 或者跳转，或者继续顺序执行
  - 取决于一定的condition codes
  - Jz(je), jnz(jne)
  - Js, jns
  - 有符号数：jl(jnge), jg(jnle), jle(jng), jge(jnl)
  - 无符号数：ja(jnbe), je(jnae), jae(jnb), jbe(jna)
- 都是direct jump类型

Accessing Conditional Codes

Instruction	Synonym	Effect	Set Condition
Sete	Setz	ZF	Equal/zero
Setne	Setnz	~ZF	Not equal/not zero
Sets		SF	Negative
Setns		~SF	Nonnegative
Setl	Setnge	SF^OF	Less
Setle	Setng	(SF^OF)|ZF	Less or Equal
Setg	Setnle	~(SF^OF)&~ZF	Greater
Setge	Setnl	~(SF^OF)	Greater or Equal
Seta	Setnbe	~CF&~ZF	Above
Setae	Setnb	~CF	Above or equal
Setb	Setnae	CF	Below
Setbe	Setna	CF|ZF	Below or equal

有符号数

无符号数

Jump Instructions

```
1     jle  .L2
2.L5:
3     movl %edx, %eax
4     sarl   $1, %eax
5     subl   %eax, %edx
6     Leal (%edx, %edx, 2), %edx
7     testl   %edx, %edx
8     jg      .L5
9 .L2:
10    movl   %edx, %eax
```

这个结构的程序是在做什么？

Jump Targets in Binary

- Two forms to represent a jump target
- PC-relative (direct jump)
  - 实质上是在指令中记录下来：跳转的目标地址（标号所在地址）与当前指令下一条指令的差值（可正可负）
  - 由于硬件每执行完一条指令，都会自动将Pc执行当前指令的下一条指令。如果决定跳转，只需要PC += 指令中的offset即可
- Absolute address (indirect jump)
  - Jump target is an absolute address

Example

```
1     jle  .L2
2.L5:
3     movl %edx, %eax
4     sarl   $1, %eax
5     subl   %eax, %edx
6     Leal (%edx, %edx, 2), %edx
7     testl   %edx, %edx
8     jg      .L5
9 .L2:
10    movl   %edx, %eax
```

PC-relative in the Relocatable Object

1	8: 7e 0d	jle	17<silly+0x17>	
2	a: 89 d0	mov	%edx, %eax	*dest1:*
3	c: c1 f8	sar	%eax	
4	e: 29 c2	sub	%eax, %edx	
5	10: 8d 14 52	lea	(%edx, %edx, 2), %edx	
6	13: 85 d2	test	%edx, %edx	
7	15: 7f f3	jg	a<silly+0x10>	
8	17: 89 d0	movl	%edx, %eax	*dest2:*

d+a = 17

17+f3(-d) =a

PC-relative in the Executable Object

```
1   804839:    7e 0d        jle   17<silly+0x17>
2   804839e:   89 d0        mov   %edx, %eax      dest1:
3   80483a0:   c1 f8        sar   %eax
4   80483a2:   29 c2        sub   %eax, %edx
5   80483a4: 8d 14 52       lea   (%edx, %edx, 2), %edx
6   80483a7: 85 d2          test  %edx, %edx
7   80483a9: 7f f3          jg    a<silly+0x10>
8   80483ab: 89 d0          movl  %edx, %eax      dest2:
```

d+804849e = 80483ab

80483ab+f3(-d) = 804839e

填空题 2分

设置

- 下列XXXX的值是什么？
  - 1)
    - 0x804828f: 74 05 je XXXX
    - 0x8048291: e8 1e 00 00 00 call 80482b4
  - 2)
    - 0x8048357: 72 e7 jb XXXX
    - 0x8048359: c6 05 10 a0 04 08 01 movb $0x1, 0x804a010

1) [填空1]
2) [填空2]

作答

练习答案

- 下列XXXX的值是什么？
  - 1)
    - 0x804828f: 74 05 je XXXX (0x8048296)
    - 0x8048291: e8 1e 00 00 00 call 80482b4
  - 2)
    - 0x8048357: 72 e7 jb XXXX (0x8048340)
    - 0x8048359: c6 05 10 a0 04 08 01 movb $0x1, 0x804a010

填空题 3分

设置

- 3)
  - XXXX: 74 12 je 8048391
  - XXXX: b8 00 00 00 00 mov $0x0, %eax
- 4)
  - 80482bf: e9 e0 ff ff ff jmp XXXX
  - 80482c4: 90 nop

1) [填空1] [填空2]
2) [填空3]

作答

练习答案

- 3)
  - 0x804837d: 74 12 je 8048391
  - 0x804837f: b8 00 00 00 00 mov $0x0, %eax
- 4)
  - 80482bf: e9 e0 ff ff ff jmp 0x80482a4
  - 80482c4: 90 nop

Outline

- Jump instructions
- Translate Control constructs in C to assembly
  - Goto, Branch, Loops
- Conditional Move
- Translate switch in C to assembly

Control Constructs in C

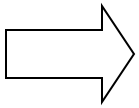
- Gotos
  - goto L
  - break
  - continue
- Branch
  - if () { } else if () { } else { }
  - switch () {case }

Control Constructs in C

- Loop
  - while () { }
  - do { } while ()
  - for (init; test; incr) { }

Translating Conditional Branches

```
if ( test-expr )
        then-statement
else
        else-statement
```



```
        t = test-expr ;
        if ( t )
                goto true ;
        else-statement
        goto done
true:
        then-statement
done:
```

Translating Conditional Branches

```
1.    int absdiff(int x, int y)
2.  {
3.        if  (x < y)
4.             return y – x;
5.        else
6.             return x – y;
7.  }
```

```
1.   int absdiff(int x, int y)
2.  {
3.        int rval ;
4.        if  (x < y)
5.            goto less
6.        rval = x – y ;
7.        goto done;
8.    less:
9.             rval = y – x;
10.   done:
11.            return rval;
12. }
```

Jump Instructions

```
1.      movl 8(%ebp), %edx      # get x
2.      movl 12(%ebp), %eax     # get y
3.      cmpl %eax, %edx         # cal x - y
4.      jl .L3                  # if  x < y goto less
5.      subl %eax, %edx         # compute x - y
6.      movl %edx, %eax         # set return val
7.      jmp .L5                 # goto done
8.  .L3:                        # less:
9.      subl %edx, %eax         # compute y – x
10. .L5:                        # done: Begin Completion code
```

课堂练习

- 填空(eax为函数返回值)

```
Int test(int x, int y) {
  int val = _______;
  if (    ) {
    if (     )
       val = ______;
    else
       val = ______;
  else if (   )
    val = ______;
   else
     val = ______;
  return val;
}
```

```
• Movl 8(%ebp), %eax ;x
• Movl 12(%ebp), %edx ;y
• Cmpl $-3, %eax
• Jge .L2
• Cmpl %edx, %eax
• Jle .L3
• Imull %edx, %eax
• Jmp .L4
• .L3: leal (%edx, %eax), %eax
• Jmp .L4
• .L2: cmpl $2, %eax
• Jg .L5
• Xorl %edx, %eax
• Jmp .L4
• .L5: subl %edx, %eax
• .L4
```

课堂答案

- 填空(eax为函数返回值)

```
Int test(int x, int y) {
  int val = _x^y或x-y____;
  if (x<-3  ) {
    if (x>y   )
      val = __ x*y ____;
    else
      val = __ x+y ____;
  else if ( x>2 或x<=2  )
    val = __x-y或x^y____;
  return val;
}
```

- Movl 8(%ebp), %eax ;x
- Movl 12(%ebp), %edx ;y
- Cmpl $-3, %eax
- Jge .L2
- Cmpl %edx, %eax
- Jle .L3
- Imull %edx, %eax
- Jmp .L4
- .L3: leal (%edx, %eax), %eax
- Jmp .L4
- .L2: cmpl $2, %eax
- Jg .L5
- Xorl %edx, %eax
- Jmp .L4
- .L5: subl %edx, %eax
- .L4

课堂练习

- void cond(int a, int *p) {
  if (p && a > 0)
  *p += a;
  }
- Gcc产生的汇编为：
  -Movl 8(%ebp), %edx ;a
  -Movl 12(%ebp), %eax ;p
  -Testl %eax, %eax
  -Je .L3
  -Testl %edx, %edx
  -Jle .L3
  -Addl %edx, (%eax)
  -.L3

1.写一个C语言的go-to版本；
2. 为什么C语言原版只有一个if语句，而汇编包含两个条件分支？

Do-while Translation

```
do
   body-statement
while (test-expr)

loop:
   body-statement
   t = test-expr;
   if ( t )
         goto loop ;
```

Do-while Translation

```
int fib_dw(int n)
{
    int i = 0;
    int val = 0 ;
    int nval = 1 ;

    do {
       int t = val + nval ;
       val = nval ;
       nval = t ;
       i++;
    } while ( i<n) ;
    return val ;
}
```

```
.L6:
    lea    (%ebx, %edx), %eax
    movl   %edx, %ebx
    movl   %eax, %edx
    incl   %ecx
    cmpl   %esi, %ecx
    jl     .L6
    movl   %ebx, %eax
```

register	value	initially
%ecx	i	0
%esi	n	n
%ebx	val	0
%edx	nval	1
%eax	t	-

While Loop Translation

```
while (test-expr)
        body-statement
```

```
loop:
   t = test-expr
   if ( !t )
        goto done;
   body-statement
   goto loop;
done:
```

```
   if ( !test-expr)
        goto done;
   do
        body-statement
   while(test-expr)
done:
```

While Loop Translation

```
int fib_w(int n)
{
    int i=1;
    int val=1;
    int nval=1;

    while ( i<n ) {
        int t=val+nval ;
        val = nval ;
        nval = t ;
        i++;
    }
    return val ;
}
```

```
int fib_w_goto(int n)
{
    int val=1;
    int nval=1;
    int nmi, t ;

    if ( val >= n )
        goto done ;
    nmi = n-1;

loop:
    t=val+nval ;
    val = nval ;
    nval = t ;
    nmi--;
    if ( nmi )
        goto loop
done:
    return val
}
```

While Loop Translation

Register usage		
Register	**Variable**	**Initially**
%edx	nmi	n-1
%ebx	val	1
%ecx	nval	1

```
int fib_w_goto(int n)
{
        int val=1;
        int nval=1;
        int nmi, t ;

        if ( val >= n )
                goto done ;
        nmi = n-1;

loop:
        t=val+nval ;
        val = nval ;
        nval = t ;
        nmi--;
        if ( nmi )
                goto loop
done:
        return val
}
```

```
1.      movl 8(%ebp), %eax
2.      movl $1, %ebx
3.      movl $1, %ecx
4.      cmpl %eax, ebx
5.      jge .L9
6.      lea –1(%eax), %edx
7.   .L10:
8.      lea (%ecx, %ebx), %eax
9.      movl %ecx, %ebx
10.     movl %eax, %ecx
11.     decl %edx
12.     jnz .L10
13.  .L9:
```

For Loop Translation

For Version

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

While Version

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

"For" Loop Do-While Conversion

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;
}
```

- Initial test can be optimized away

Goto Version

```
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;
}
```

多判断条件的短路性

- 逻辑与运算

while (sum < 1000) and (count ≤ 24) loop

... { body of loop }

end while;

两个条件必须都成立<==>任一个不成立都退出

```
whileSum:     cmpw $1000, $sum    ; sum < 1000 ?
              jnl  endWhileSum    ; exit if not
              cmpw $24, %cx       ; count <= 24 ?
              jnle  endWhileSum   ; exit if not
              . . .               ; body of loop . .
              jmp whileSum        ; go check condition again
endWhileSum:
```

多判断条件的短路性

- 逻辑或运算

```
while (sum < 1000) or (flag=1) loop
        ... { body of loop }
end while;
```

```
whileSum:   cmpw $1000, $sum    ; sum < 1000 ?
            jl  body            ; execute body if so
            cmpb $1, %dh                ; flag=1 ?
            jne  endWhileSum    ; exit if not
body:       . . .               ; body of loop . .
            ....
            jmp  whileSum       ; go check condition again
endWhileSum:
```

Loop 指令实现 for 循环

- LOOP label
  - 利用 ECX 计数器自动减1，方便实现计数循环的程序结构。
  - ECX ← ECX－1 ;dec ecx
  - 如果 ECX≠0，转标号 label 继续循环 ;jnz label
  - 否则继续执行循环指令下面的语句
  - 操作数 label 采用相对短转移寻址方式，及向后128个字节或向前127个字节。

使用“loop”的80x86汇编代码

```
for  count := 20 downto 1   loop
        ... { body of loop }
end for
```

```
                movl $20, %ecx          ; number of iterations
forCount:       . . .                   ; body of loop
                . . .
                loop forCount           ; repeat body 20 times
```

```
                movl $number, %ecx      ; number of iterations
forIndex:       . . .                   ; body of loop
                . . .
                loop forIndex           ; repeat body number times
```

可靠的 for 循环

- 当ecx中的值为0时，循环不会执行

```
        movl $number, %ecx      ; number of iterations
        cmpl $0, %ecx           ; number = 0 ?
        jle  endFor             ; skip loop if number = 0
forIndex: . . .                 ; body of loop
        . . .
        loop forIndex           ; repeat body number times
endFor:
```

```
        movl $number, %ecx      ; number of iterations
        jecxz  endFor           ; skip loop if number = 0
forIndex: . . .                 ; body of loop
        . . .
        loop forIndex           ; repeat body number times
endFor:
```

主观题 10分

设置

程序填空，变量映射关系，并解释函数功能

```
Int fun_a(unsigned x) {
  int val = 0;
  while (          ) {

  }
  return _________;
}
```

- x@$ebp+8

```
  Movl 8(%ebp), %edx
  Movl $0, %eax
  Testl %edx, %edx
  Je .L7
.L10:
  xorl %edx, %eax
  shrl %edx  ;逻辑右移1位
  jne .L10
.L7:
  andl $1, %eax
```

作答

课堂练习

程序填空，变量映射关系，并解释函数功能

```
Int fun_a(unsigned x){
  int val = 0;
  while (    x     ){
    val ^= x;
    x>>=1
  }
  return val&0x01;
}
```

- x@$ebp+8

```
  Movl 8(%ebp), %edx
  Movl $0, %eax
  Testl %edx, %edx
  Je .L7
.L10:
  xorl %edx, %eax
  shrl %edx  ;逻辑右移1位
  jne .L10
.L7:
  andl $1, %eax
```

课堂练习

- 函数fun_b(unsigned long x) {
  - Long val = 0;
  - Long i;
  - For (… ; … ; …) {
    - …
  - }
  - Return val;
- }
- Gcc生成的汇编代码如右所示。
- 完成下面工作：
  - A. 根据汇编代码，填写C代码缺失的部分；
  - B. 解释循环前为什么没有初始测试，也没有初始跳转到循环内部的测试部分；
  - C. 用自然语言描述这个函数的功能

```
# x in %rdi
1 func_b:
2    Movl $64, %edx
3    movl $0, %eax
4  .L10:
5    movq %rdi, %rcx
6    andl  $1, %ecx
7    addq %rax, %rax
8    orq $rcx, %rax
9    shrq %rdi
10   subq $1, %rdx
11   jne .L10
12   ret
```

练习答案

- fun_b(unsigned long x) {
  - Long val = 0;
  - Long i;
  - For (i=64; i!=0 ; i--) {　　// 尽量忠于原意
    - Val = (val << 1) | (x&1)
    - X >>= 1;
  - }
  - Return val;
- }
- 2）因为循环次数是常量
- 3）将x镜像反转

Outline

- Jump instructions
- Translate Control constructs in C to assembly
  - Goto, Branch, Loops
- **Conditional Move**
- Translate switch in C to assembly

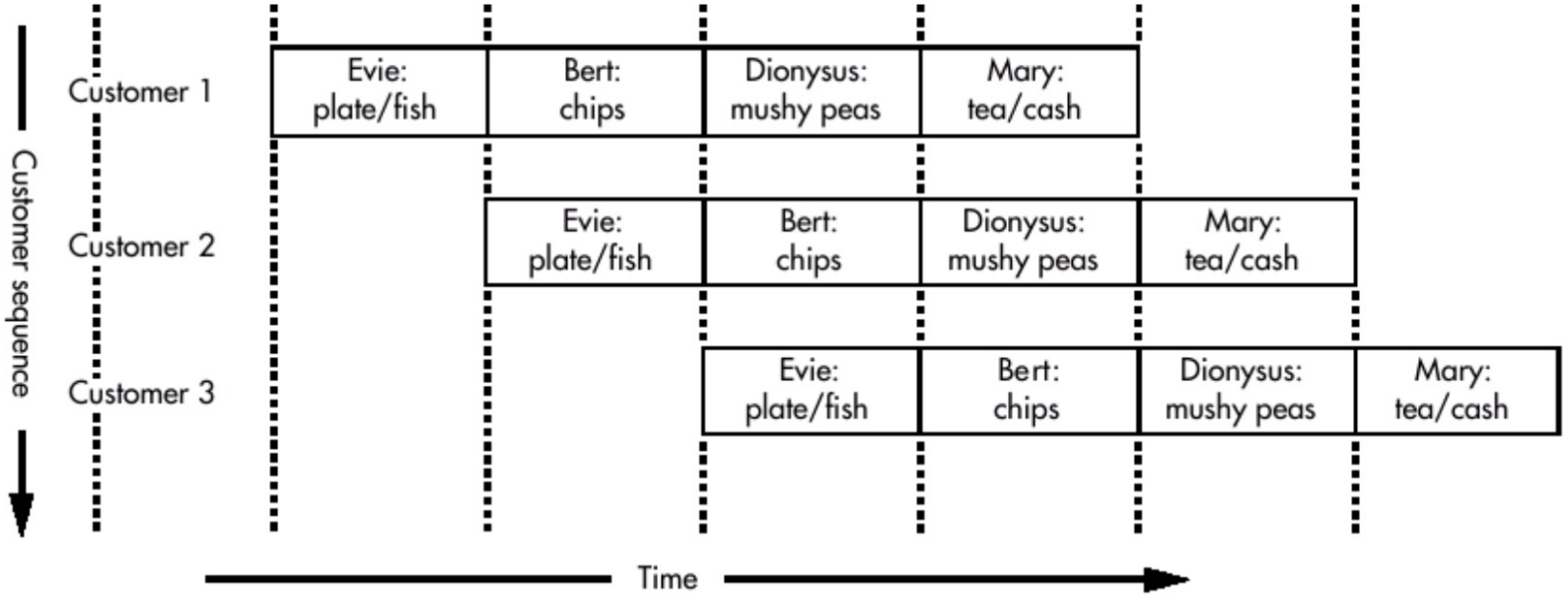
Conditional Move

- 为什么需要Conditional Move
  - 例子：求x和y之间差值的绝对值
  - 1 int absdiff(int x, int y) {
  - 2 return x < y ? y-x : x-y;
  - 3 }

  - 正常思路：采用分支if/else
  - 挑战：一旦分支预测失败，代价较大
    - 因为需要从内存加载指令，很慢；而预测成功时，可以从很快的cache中加载指令（指令预取）
    - 破坏CPU流水线

流水线（pipeline）

- Evie炸鱼店
  - 提供：炸鳕鱼、炸薯片、豌豆粥、茶
  - 排队问题
  - 加长柜台，排队移动中依次得到各种食物

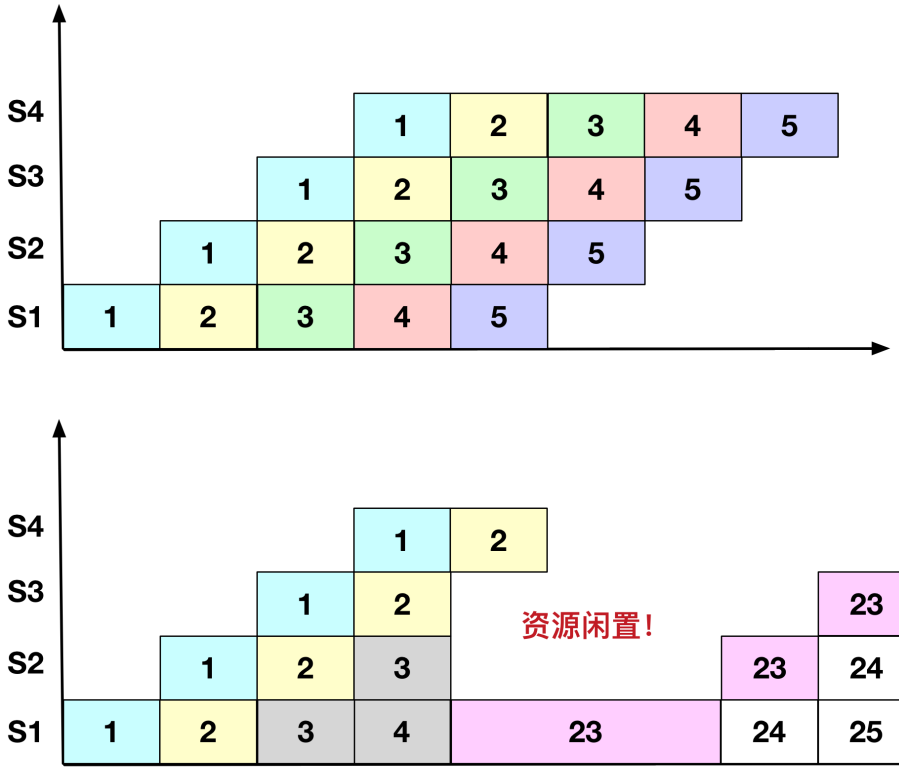


流水线（pipeline）

- 优点
  - 资源充分利用，减少排队时间
  - 每个工人的工作简单，容易培训，成本低，不容易出错
- 挑战：
  - Mary一边倒茶，一边找零钱，成为4个环节中的瓶颈（拆分）
  - Daphne和Lola一起来，Daphne不买茶，则Lola不买薯片。所以Lola驻足观望Daphne是否买茶（气泡）
  - 对信誉不好的Cyril，要等Mary点完钱，Evie才给他盛炸鱼（Cyril要用前面三个顾客用的资源—Mary，资源冲突）

流水线（pipeline）

- 如果分支预测失败
  - 需要再去加载新的指令，很长时间内，流水线各个环节没有指令需要运行，处于空闲状态，性能下降



Conditional Move

(a) Original C code

```
1 int absdiff(int x, int y) {
2     return x < y ? y-x : x-y;
3 }
```

(b) Implementation using conditional assignment

```
1 int cmovdiff(int x, int y) {
2     int tval = y-x;
3     int rval = x-y;
4     int test = x < y;
5     /* Line below requires
6        single instruction: */
7     if (test) rval = tval;
8     return rval;
9 }
```

Conditional Move

(c) Generated assembly code

(x at %ebp+8, y at %ebp+12)

%ecx	x
%edx	y
%ebx	y-x
%eax	x-y

```
1. movl   8(%ebp),  %ecx     Get x
2. movl   12(%ebp), %edx     Get y
3. movl   %edx,  %ebx        Copy y
4. subl   %ecx,  %ebx        Compute y-x
5. movl   %ecx, %eax         Copy x
6. subl   %edx,  %eax        Compute x-y and set as return value
7. cmpl   %edx,  %ecx        Compare x:y
8. cmovl %ebx, %eax          If < , replace return value with y-x
```

Conditional Move

Instruction		Synonym	Move condition	Description
`cmove`	S, R	`cmovz`	`ZF`	Equal / zero
`cmovne`	S, R	`cmovnz`	`~ZF`	Not equal / not zero
`cmovs`	S, R		`SF`	Negative
`cmovns`	S, R		`~SF`	Nonnegative
`cmovg`	S, R	`cmovnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`cmovge`	S, R	`cmovnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`cmovl`	S, R	`cmovnge`	`SF ^ OF`	Less (signed <)
`cmovle`	S, R	`cmovng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`cmova`	S, R	`cmovnbe`	`~CF & ~ZF`	Above (unsigned >)
`cmovae`	S, R	`cmovnb`	`~CF`	Above or equal (Unsigned >=)
`cmovb`	S, R	`cmovnae`	`CF`	Below (unsigned <)
`cmovbe`	S, R	`cmovna`	`CF | ZF`	below or equal (unsigned <=)

Conditional Move

- Conditional Move的问题
  - 与真正的分支语句相比，Conditional Move需要实现对两个分支情况都求值，然后根据情况选择一个
  - 而分支语句只需要对其中一个进行求值
  - 所以，等某个分支语句有“副作用”（会修改某个寄存器或变量的值）时，不能使用Conditional Move

Invalid Situation

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

如果使用conditional move，翻译的汇编为：

```
# xp@%rdi
1 cread:
2 movq (%rdi), %rax   # v=*xp
3 testq %rdi, %rdi    # xp==0?
4 movl $0, %edx
5 cmove %rdx, %rax    # if xp==0, return 0
6 ret
```

如果xp=0，那么第2行用NULL指针寻址出现错误

主观题 10分

设置

填写C语言代码：

```
Long test (long x, long y) {
    long val = _____;
    if (_____) {
        if (_____) {
            val = _____;
        else
            val = _____;
    } else if (_____)
        val = _____;
    return val;
}
```

```
# x @ %rdi, y @ %rsi
Test:
  leaq    0(,%rdi,8), %rax
  testq   %rsi, %rsi
  jle     .L2
  movq    %rsi, %rax
  subq    %rdi, %rax
  movq    %rdi, %rdx
  andq    %rsi, %rdx
  cmpq    %rsi, %rdi
  cmovge %rdx, %rax
.L2:
  addq    %rsi, %rdi
  cmpq    $-2, %rsi
  cmovle %rdi, %rax
  ret
```

作答

练习答案

```
Long test (long x, long y) {
    long val = _____8x_____;
    if (____y>0_____) {
        if (____x<y_____) {
            val = ____y-x_____;
        else
            val = ____x&y_____;
    } else if (____y<=-2_____)
        val = ___x+y_____;
    return val;
}
```

```
# x @ %rdi, y @ %rsi
Test:
  leaq   0(,%rdi,8), %rax
  testq  %rsi, %rsi
  jle    .L2
  movq   %rsi, %rax
  subq   %rdi, %rax
  movq   %rdi, %rdx
  andq   %rsi, %rdx
  cmpq   %rsi, %rdi
  cmovge %rdx, %rax
.L2:
  addq   %rsi, %rdi
  cmpq   $-2, %rsi
  cmovle %rdi, %rax
  ret
```

Outline

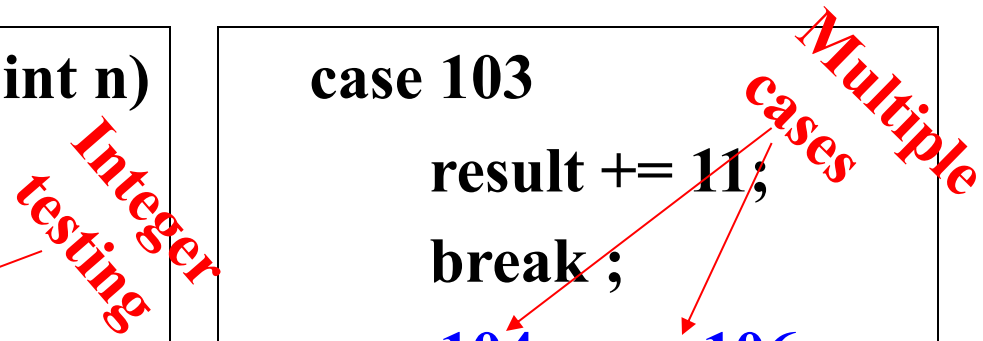
- Jump instructions
- Translate Control constructs in C to assembly
  - Goto, Branch, Loops
- Conditional Move
- **Translate switch in C to assembly**

Switch Construct

- Properties of Switch Construct
  - Integer testing
  - Multiple outcomes (may be a large number)
    - 如果采用if/elseif分支，分支很多，需要做多次判断，产生计算开销（cmp）
    - 一次性处理所有分支，效率更高！
  - Improve the readability of the source code

Switch Statements

```
int switch_eg(int x, int n)
{
   int result = x ;
   switch ( n ) {
   case 100:
         result *= 13 ;
         break ;
   case 102:
         result += 10 ;
         /* fall through */
   case 103
         result += 11;
         break ;
   case 104: case 106:
         result *= result ;
         break ;
   default:
         result = 0 ;
   }
   return result ;
}
```



Switch Form

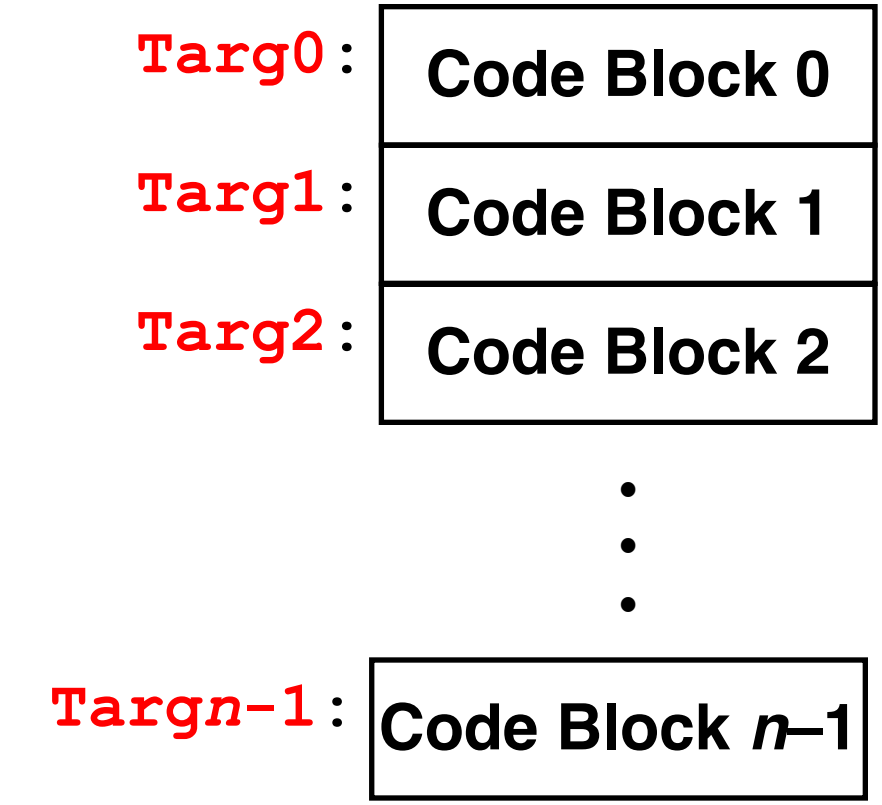
```
switch(op) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    . . .
  case val_n-1:
    Block n-1
}
```

Jump Table

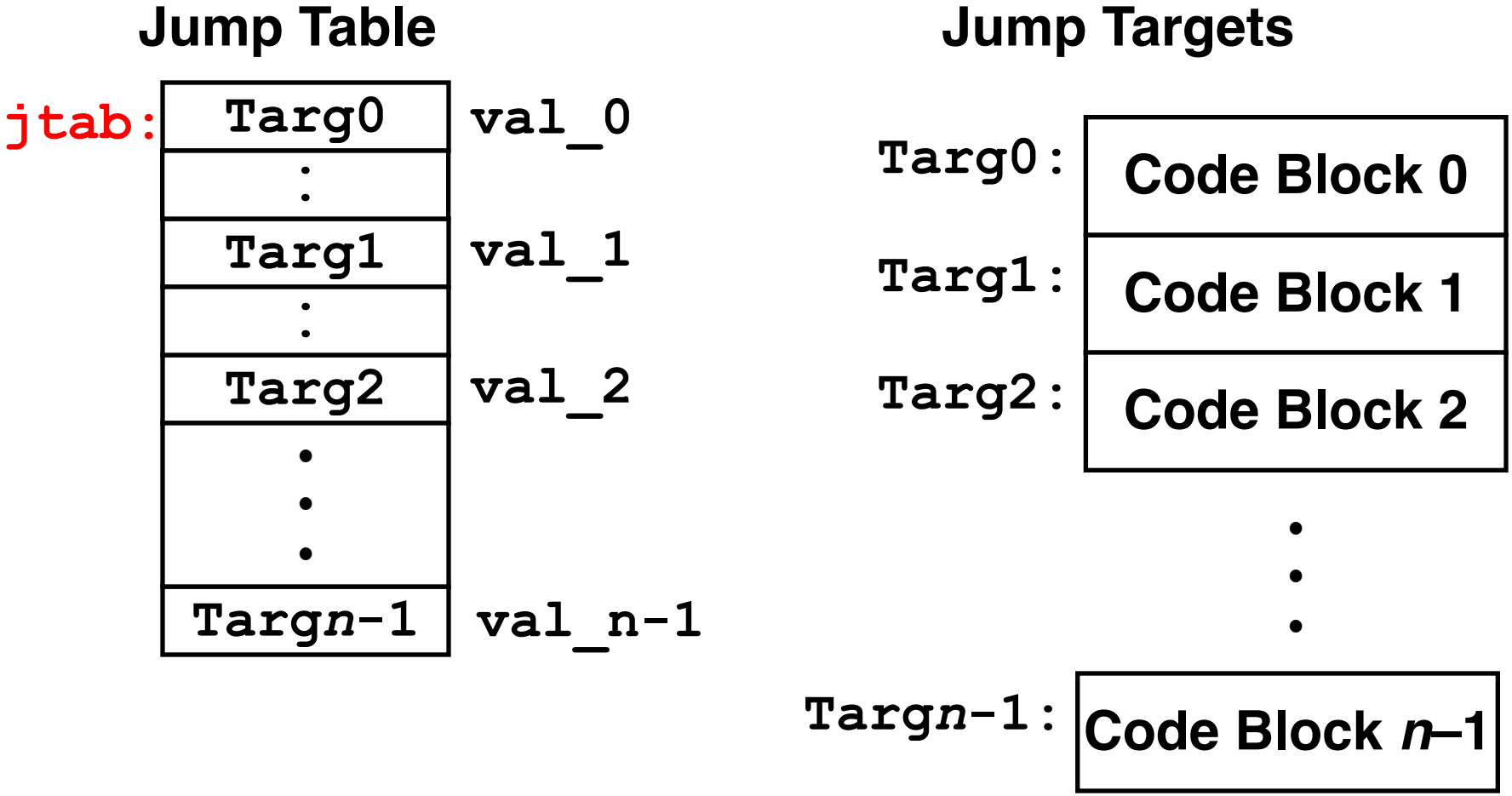
- 高效的实现方式
- 避免很多if-else语句
- 标准
  - Case的数量，以及case值的分布
    - Case数量太少，还是用if-else方便
    - Case值分布太广、太稀疏，jump table开销过大

From Cases to Targets

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



Construct a Jump Table

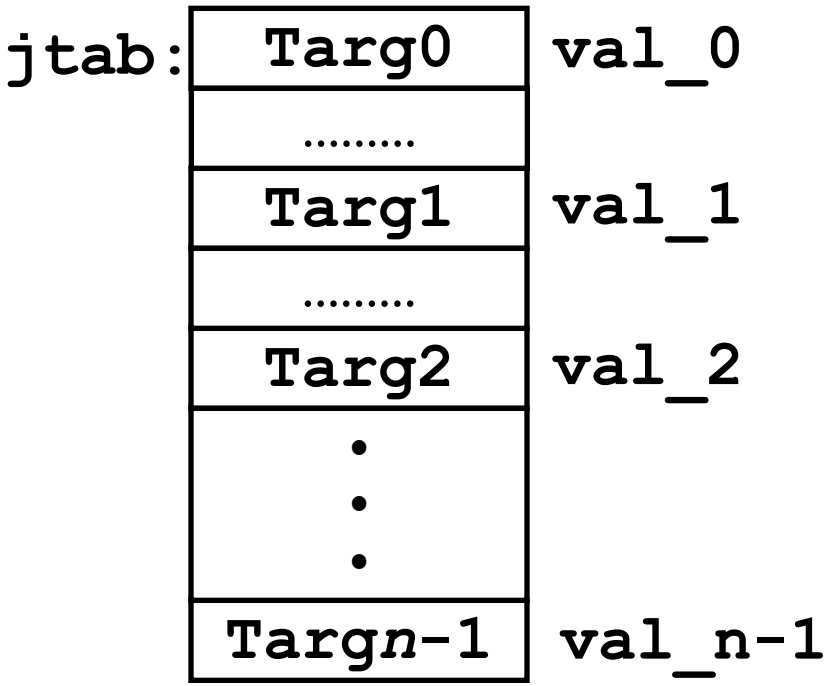


Implement the Switch

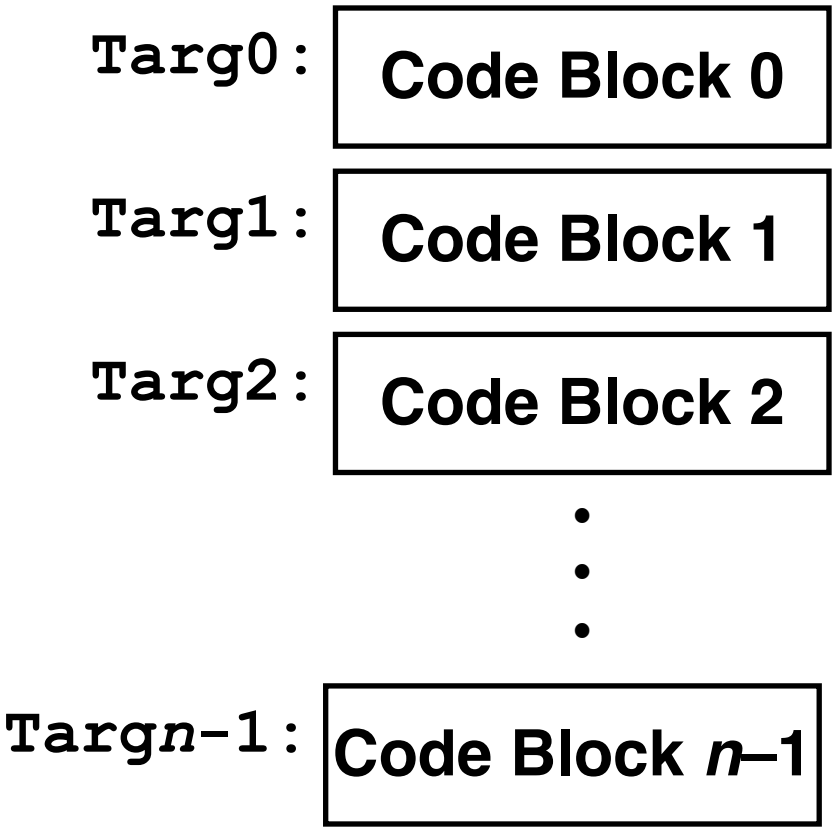
Approx. Translation

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

Jump Table



Jump Targets



```
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;
}
```

Switch Statement Example

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

Jump Table Structure

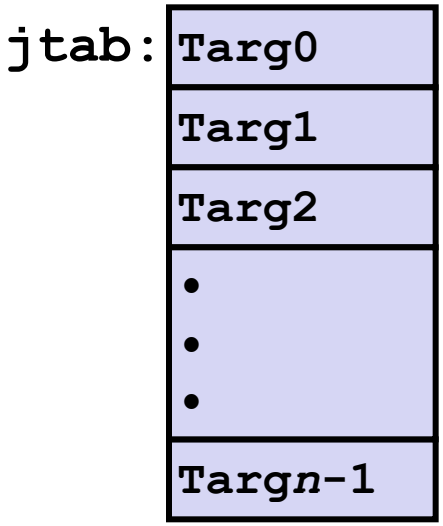
Switch Form

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

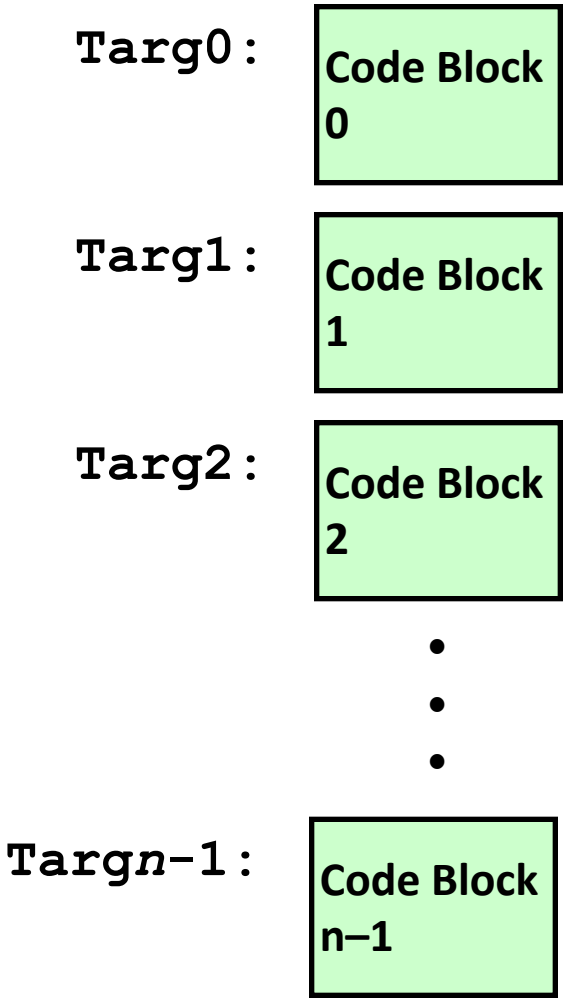
Translation (Extended C)

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

Jump Table



Jump Targets



Switch Statement Example

```
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)
```

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

Note that w not initialized here

Switch Statement Example

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

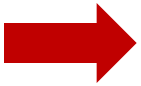
Jump table

```
.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



Assembly Setup Explanation

- Table Structure
  - Each target requires 8 bytes
  - Base address at `.L4`

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
```

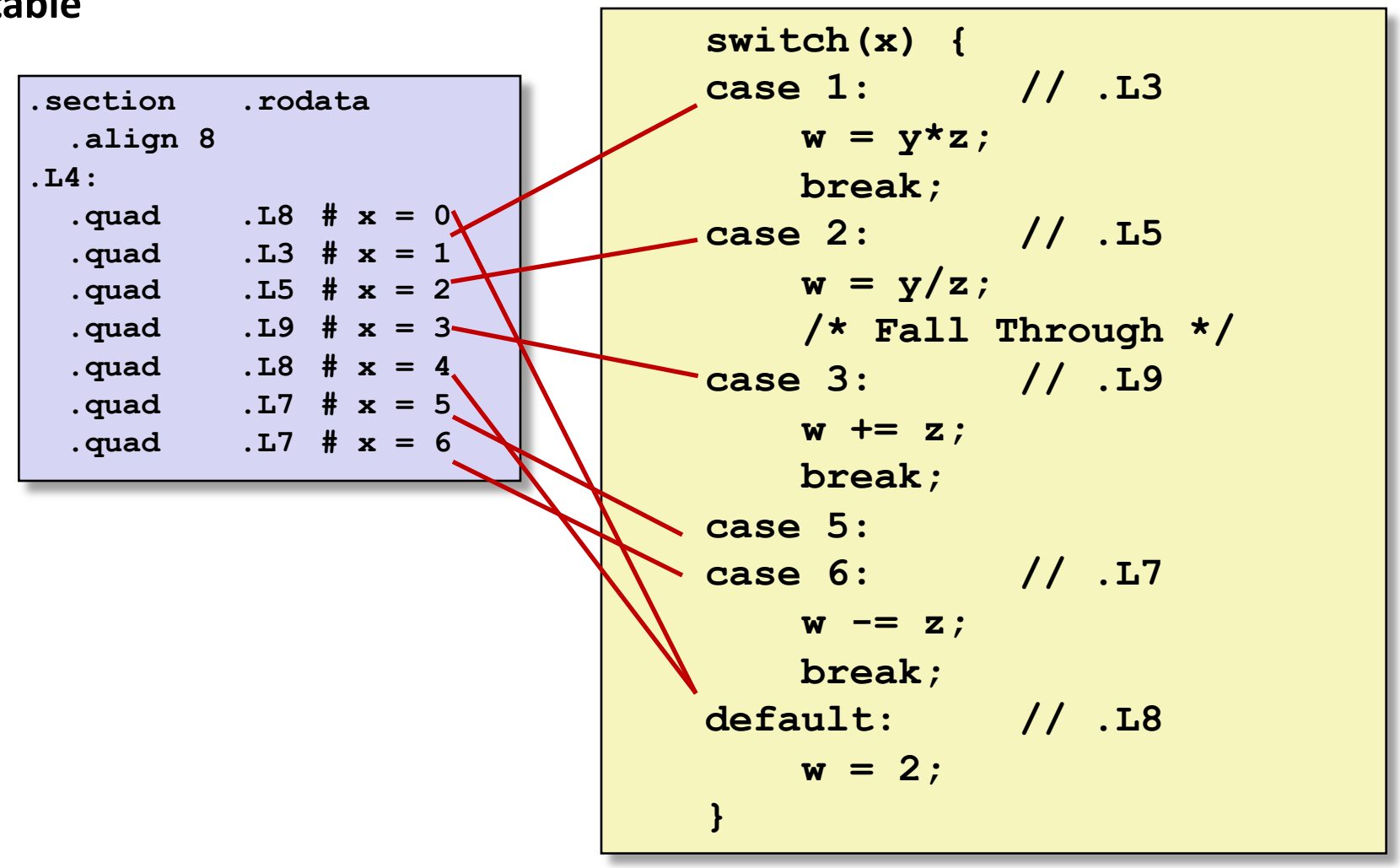
- Jumping
  - Direct: `jmp .L8`
  - Jump target is denoted by label `.L8`
  - Indirect: `jmp *.L4(,%rdi,8)`
  - Start of jump table: `.L4`
  - Must scale by factor of 8 (addresses are 8 bytes)
  - Fetch target from effective Address `.L4 + x*8`
    - Only for $0 \le x \le 6$

Jump Table

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
```

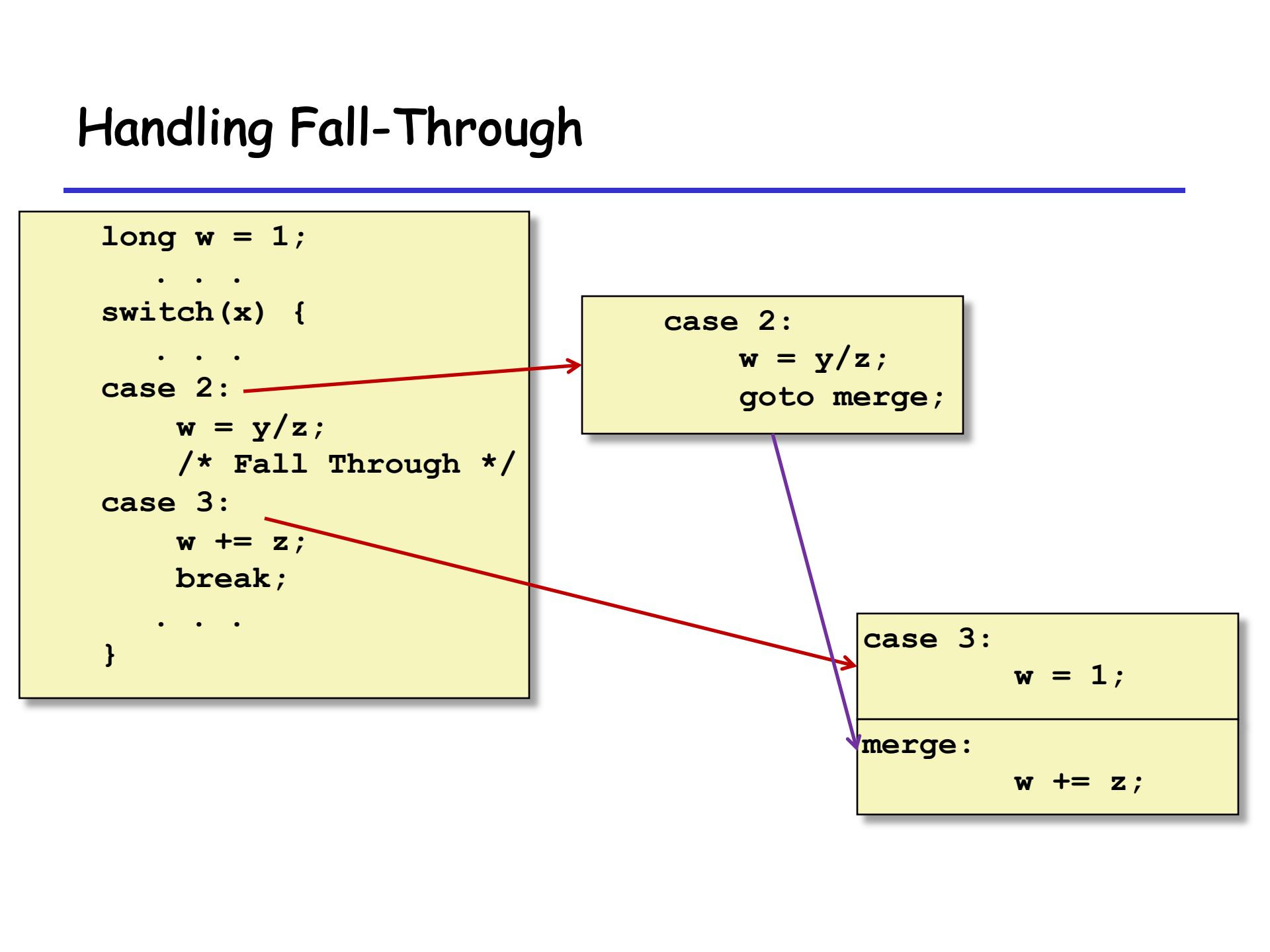
Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

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)

```
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

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