



# 编译原理 第九章 代码生成

哈尔滨工业大学 陈鄞



The background of the slide features a 3D metallic staircase graphic that ascends from the bottom left towards the top right. The staircase is composed of several steps, each with a flat top and a vertical riser, giving it a three-dimensional appearance. The background is a light blue-grey color, overlaid with a pattern of binary code (0s and 1s) in a slightly darker shade, creating a digital or technological theme.

# 提纲

- 9.1 代码生成器的主要任务
- 9.2 一个简单的目标机模型
- 9.3 指令选择
- 9.4 寄存器的选择
- 9.5 窥孔优化

# 代码生成器的主要任务

## ➤ 指令选择

➤ 选择适当的 **目标机指令** 来实现 **中间表示(IR)语句**

➤ 例：

➤ 三地址语句

➤  $x = y + z$

➤ 目标代码

➤ **LD**    **R0**, **y**                    /\* 把y的值**加载**到寄存器**R0**中 \*/

➤ **ADD** **R0**, **R0** , **z**                /\* **z****加**到**R0**上 \*/

➤ **ST**    **x** , **R0**                    /\* 把**R0**的值**保存**到**x**中 \*/

# 代码生成器的主要任务

## ➤ 指令选择

➤ 选择适当的 **目标机指令** 来实现 **中间表示(IR)语句**

➤ 例：

### 三地址语句序列

➤  $a = b + c$

➤  $d = a + e$

### 目标代码

➤ *LD R0, b* //  $R0 = b$

➤ *ADD R0, R0, c* //  $R0 = R0 + c$

➤ *ST a, R0* //  $a = R0$

➤ ***LD R0, a*** //  $R0 = a$

➤ *ADD R0, R0, e* //  $R0 = R0 + e$

➤ *ST d, R0* //  $d = R0$

# 代码生成器的主要任务

## ➤ 指令选择

➤ 选择适当的 **目标机指令** 来实现 **中间表示(IR)语句**

## ➤ 寄存器分配 (allocation) 和指派 (assignment)

➤ 把哪个值放在哪个寄存器中

## ➤ 指令排序

➤ 按照什么顺序来安排指令的执行



# 提纲

9.1 代码生成器的主要任务

9.2 一个简单的目标机模型

9.3 指令选择

9.4 寄存器的选择

9.5 窥孔优化

## 9.2 一个简单的目标机模型

### ➤ 三地址机器模型

➤ 加载、保存、运算、跳转等操作

➤ 内存按字节寻址

➤  $n$  个通用寄存器  $R0, R1, \dots, Rn-1$

➤ 假设所有的运算分量都是整数

➤ 指令之间可能有一个标号

# 目标机器的主要指令

- 加载指令  $LD\ dst, addr$ 
  - $LD\ r, x$
  - $LD\ r_1, r_2$
- 保存指令  $ST\ x, r$
- 运算指令  $OP\ dst, src1, src2$
- 无条件跳转指令  $BR\ L$
- 条件跳转指令  $Bcond\ r, L$ 
  - 例:  $BLTZ\ r, L$



# 寻址模式

➤ 变量名  $a$

➤ 例:  $LD\ R1, a$

# 寻址模式

➤ 变量名  $a$

➤  $a(r)$

➤  $a$  是一个变量， $r$  是一个寄存器

➤ 这个寻址方式对于数组访问是很有用的

➤ 其中， $a$  是数组的基地址， $r$  中存放了数组元素的偏移地址

➤ 例：LD R1,  $a(R2)$

# 寻址模式

➤ 变量名 $a$

➤  $a(r)$

➤  $c(r)$

➤  $c$ 是一个整数，寄存器 $r$ 中存放的是一个地址

➤  $c(r)$ 所表示的内存地址是寄存器 $r$ 中的值加上整数 $c$

➤ 这个寻址方式可以用于沿指针取值

➤ 例：  $LD\ R1, 100(R2)$

# 寻址模式

➤ 变量名 $a$

➤  $a(r)$

➤  $c(r)$

➤  $*r$

➤ 在寄存器 $r$ 的内容所表示的位置上存放的内存位置

# 寻址模式

➤ 变量名  $a$

➤  $a(r)$

➤  $c(r)$

➤  $*r$

➤  $*c(r)$

➤ 在寄存器  $r$  中内容加上  $c$  后所表示的位置上存放的内存位置

➤ 例:  $LD\ R1, *100(R2)$

# 寻址模式

➤ 变量名  $a$

➤  $a(r)$

➤  $c(r)$

➤  $*r$

➤  $*c(r)$

➤  $\#c$

➤ 例:  $LD\ R1,\ \#100$



# 提纲

9.1 代码生成器的主要任务

9.2 一个简单的目标机模型

9.3 指令选择

9.4 寄存器的选择

9.5 窥孔优化

# 运算语句的目标代码

## ➤ 三地址语句

➤  $x = y - z$

## ➤ 目标代码

➤ *LD R1, y*                      //  $R1 = y$

➤ *LD R2, z*                      //  $R2 = z$

➤ *SUB R1, R1, R2*                //  $R1 = R1 - R2$

➤ *ST x, R1*                      //  $x = R1$

尽可能避免使用上面的全部四个指令，如果

✓ 所需的运算分量已经在寄存器中了

✓ 运算结果不需要存放回内存



# 数组寻址语句的目标代码

## ➤ 三地址语句

➤  $b = a[i]$

➤  $a$  是一个实数数组，每个实数占8个字节

## ➤ 目标代码

➤ ***LD     R1 , i         // R1 = i***

➤ ***MUL R1 , R1, 8    // R1=R1 \* 8***

➤ ***LD     R2 , a(R1)    // R2=contents ( a + contents(R1) )***

➤ ***ST     b   , R2        // b = R2***

## ➤ 寻址模式

➤ 变量名  $a$

➤  $a(r)$

➤  $c(r)$

➤  $*r$

➤  $*c(r)$

➤  $\#c$

# 数组寻址语句的目标代码

## ➤ 三地址语句

➤  $a[j] = c$

➤  $a$  是一个实数数组，每个实数占8个字节

## ➤ 目标代码

➤  $LD \quad R1, \quad c \qquad \qquad // R1 = c$

➤  $LD \quad R2, \quad j \qquad \qquad // R2 = j$

➤  $MUL \quad R2, \quad R2, 8 \qquad // R2 = R2 * 8$

➤  $ST \quad a(R2), R1 \qquad // contents(a+contents(R2))=R1$

# 指针存取语句的目标代码

## ➤ 三地址语句

➤  $x = *p$

## ➤ 目标代码

➤  $LD\ R1, p$                        $//\ R1 = p$

➤  $LD\ R2, 0(R1)$                    $//\ R2 = contents\ (0 + contents\ (R1))$

➤  $ST\ x, R2$                        $//\ x = R2$

## ➤ 寻址模式

➤ 变量名  $a$

➤  $a(r)$

➤  $c(r)$

➤  $*r$

➤  $*c(r)$

➤  $\#c$

# 指针存取语句的目标代码

## ➤ 三地址语句

➤  $*p = y$

## ➤ 目标代码

➤  $LD \ R1, \ p \quad // \ R1 = p$

➤  $LD \ R2, \ y \quad // \ R2 = y$

➤  $ST \ 0(R1), R2 \quad // \text{contents} ( 0 + \text{contents} ( R1 ) ) = R2$

# 条件跳转语句的目标代码

## ➤ 三地址语句

➤ *if  $x < y$  goto  $L$*

## ➤ 目标代码

➤ *LD      $R1$ ,  $x$*                                  *//  $R1 = x$*

➤ *LD      $R2$ ,  $y$*                                  *//  $R2 = y$*

➤ *SUB    $R1$ ,  $R1$ ,  $R2$*                            *//  $R1 = R1 - R2$*

➤ *BLTZ  $R1$ ,  $M$*                                  *// if  $R1 < 0$  jump to  $M$*

*$M$ 是标号为 $L$ 的三地址指令所产生的  
目标代码中的第一个指令的标号*

# 过程调用和返回的目标代码

- 使用静态内存分配的方式
- 使用栈式内存分配的方式

# 使用静态内存分配的方式

三地址语句	call callee	return
目标代码 (静态方式)	ST callee.staticArea, #here + 20 BR callee.codeArea	BR *callee.staticArea

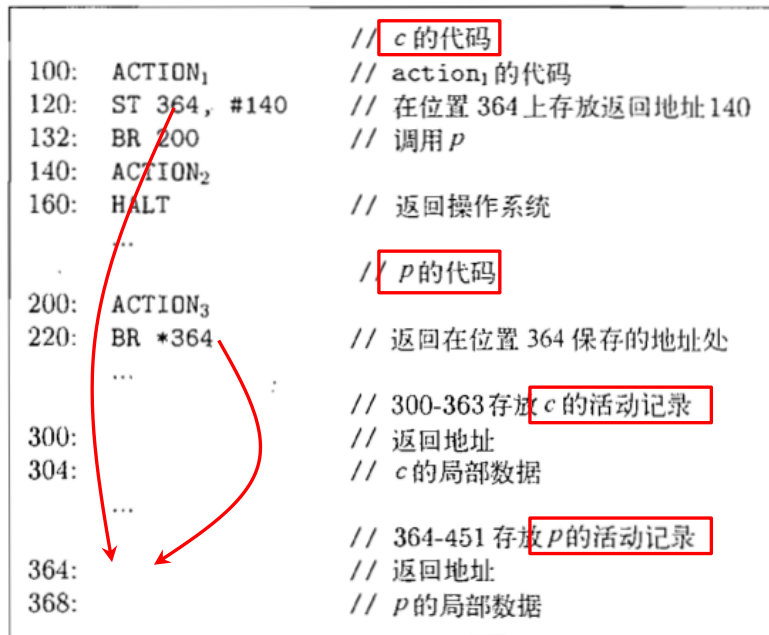
返回地址

callee的活动记录在静态区中的起始位置

callee的目标代码在代码区中的起始位置

```
                // code for c
action1
call p
action2
halt

                // code for p
action3
return
```



# 使用栈式内存分配的方式

三地址语句	call callee	return
目标代码 (静态方式)	ST callee.staticArea, #here + 20 BR callee.codeArea	BR *callee.staticArea
目标代码 (栈式)	ADD SP, SP, #caller.recordsize ST 0(SP), #here + 16 BR callee.codeArea	<ul style="list-style-type: none"><li>➤ 被调用过程 BR *0(SP)</li><li>➤ 调用过程 SUB SP, SP, #caller.recordsize</li></ul>

## 寻址模式

- 变量名  $a$
- $a(r)$
- $c(r)$
- $*r$
- $*c(r)$
- $\#c$



# 使用栈式内存分配的方式

三地址语句	call callee	return
目标代码 (栈式)	ADD SP, SP, #caller.recordsize ST 0(SP), #here + 16 BR callee.codeArea	▪ 被调用过程 BR *0(SP) ▪ 调用过程 SUB SP, SP, #caller.recordsize

```
                                // code for m
action1
call q
action2
halt

                                // code for p
action3
return

                                // code for q
action4
call p
action5
call q
action6
call q
return
```

```
                                // m的代码
100: LD SP, #600                // 初始化栈
108: ACTION1                   // action1的代码
128: ADD SP, SP, #msize         // 调用指令序列的开始
136: ST 0(SP), #152            // 将返回地址压入栈
144: BR 300                    // 调用q
152: SUB SP, SP, #msize        // 恢复SP的值
160: ACTION2
180: HALT
...

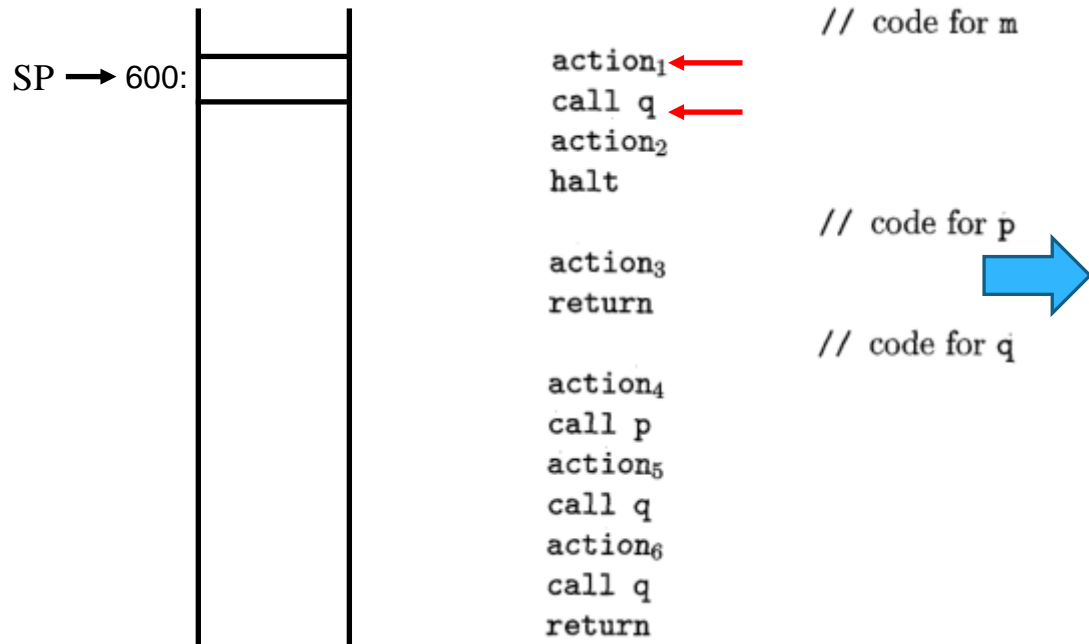
                                // p的代码
200: ACTION3
220: BR *0(SP)                 // 返回
...

                                // q的代码
300: ACTION4                   // 包含有跳转到456的条件转移指令
320: ADD SP, SP, #qsize
328: ST 0(SP), #344            // 将返回地址压入栈
336: BR 200                    // 调用p
344: SUB SP, SP, #qsize
352: ACTION5
372: ADD SP, SP, #qsize
380: ST 0(SP), #396            // 将返回地址压入栈
388: BR 300                    // 调用q
396: SUB SP, SP, #qsize
404: ACTION6
424: ADD SP, SP, #qsize
432: ST 0(SP), #440            // 将返回地址压入栈
440: BR 300                    // 调用q
448: SUB SP, SP, #qsize
456: BR *0(SP)                 // 返回
...

600: ...                       // 栈区的开始处
```

# 使用栈式内存分配的方式

三地址语句	call callee	return
目标代码 (栈式)	ADD SP, SP, #caller.recordsize ST 0(SP), #here + 16 BR callee.codeArea	▪ 被调用过程 BR *0(SP) ▪ 调用过程 SUB SP, SP, #caller.recordsize



```

// m的代码
100: LD SP, #600 // 初始化栈
108: ACTION1 // action1的代码
128: ADD SP, SP, #msize 20 // 调用指令序列的开始
136: ST 0(SP), #152 // 将返回地址压入栈
144: BR 300 // 调用q
152: SUB SP, SP, #msize // 恢复SP的值
160: ACTION2
180: HALT
...

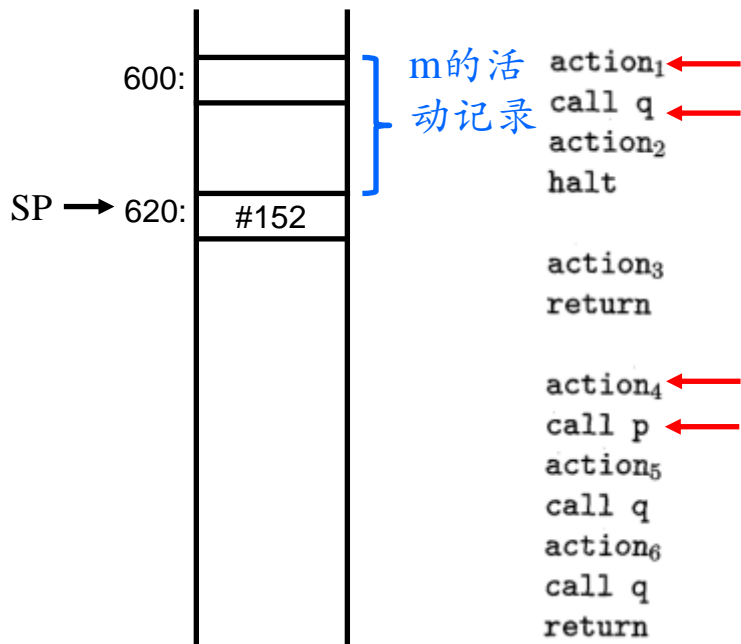
// p的代码
200: ACTION3
220: BR *0(SP) // 返回
...

// q的代码
300: ACTION4 // 包含有跳转到456的条件转移指令
320: ADD SP, SP, #qsize 60
328: ST 0(SP), #344 // 将返回地址压入栈
336: BR 200 // 调用p
344: SUB SP, SP, #qsize
352: ACTION5
372: ADD SP, SP, #qsize
380: ST 0(SP), #396 // 将返回地址压入栈
388: BR 300 // 调用q
396: SUB SP, SP, #qsize
404: ACTION6
424: ADD SP, SP, #qsize
432: ST 0(SP), #440 // 将返回地址压入栈
440: BR 300 // 调用q
448: SUB SP, SP, #qsize
456: BR *0(SP) // 返回
...

600: // 栈区的开始处
```

# 使用栈式内存分配的方式

三地址语句	call callee	return
目标代码 (栈式)	ADD SP, SP, #caller.recordsize ST 0(SP), #here + 16 BR callee.codeArea	▪ 被调用过程 BR *0(SP) ▪ 调用过程 SUB SP, SP, #caller.recordsize



// code for m

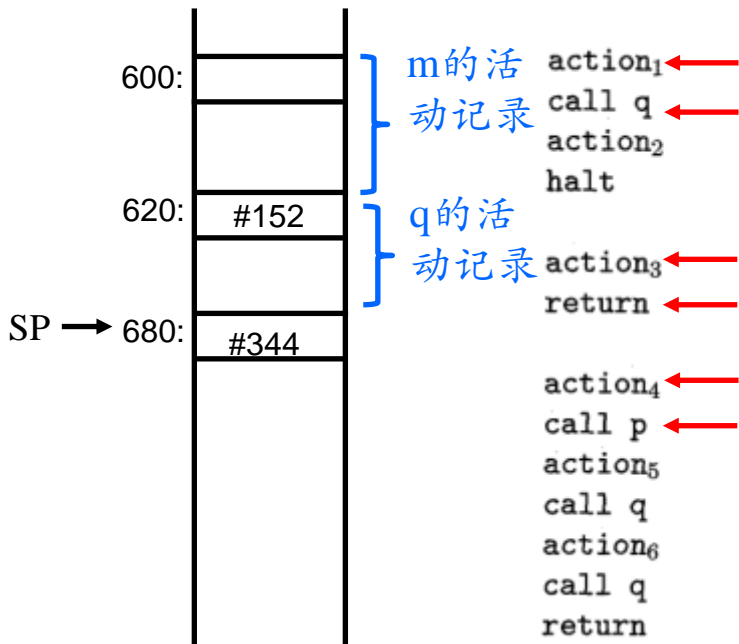
// code for p

// code for q

100:	LD SP, #600	// m的代码
108:	ACTION <sub>1</sub>	// 初始化栈
128:	ADD SP, SP, #msize	// action <sub>1</sub> 的代码
136:	ST 0(SP), #152	// 调用指令序列的开始
144:	BR 300	// 将返回地址压入栈
152:	SUB SP, SP, #msize	// 调用q
160:	ACTION <sub>2</sub>	// 恢复SP的值
180:	HALT	
...		
200:	ACTION <sub>3</sub>	// p的代码
220:	BR *0(SP)	// 返回
...		
300:	ACTION <sub>4</sub>	// q的代码
320:	ADD SP, SP, #qsize	// 包含有跳转到456的条件转移指令
328:	ST 0(SP), #344	// 将返回地址压入栈
336:	BR 200	// 调用p
344:	SUB SP, SP, #qsize	
352:	ACTION <sub>5</sub>	
372:	ADD SP, SP, #qsize	
380:	ST 0(SP), #396	// 将返回地址压入栈
388:	BR 300	// 调用q
396:	SUB SP, SP, #qsize	
404:	ACTION <sub>6</sub>	
424:	ADD SP, SP, #qsize	
432:	ST 0(SP), #440	// 将返回地址压入栈
440:	BR 300	// 调用q
448:	SUB SP, SP, #qsize	
456:	BR *0(SP)	// 返回
...		
600:		// 栈区的开始处

# 使用栈式内存分配的方式

三地址语句	call callee	return
目标代码 (栈式)	ADD SP, SP, #caller.recordsize ST 0(SP), #here + 16 BR callee.codeArea	<ul style="list-style-type: none"> <li>被调用过程 BR *0(SP)</li> <li>调用过程 SUB SP, SP, #caller.recordsize</li> </ul>



// code for m

action<sub>1</sub>  
call q  
action<sub>2</sub>  
halt

// code for p

action<sub>3</sub>  
return

// code for q

action<sub>4</sub>  
call p  
action<sub>5</sub>  
call q  
action<sub>6</sub>  
call q  
return

```

// m的代码
100: LD SP, #600           // 初始化栈
108: ACTION1              // action1的代码
128: ADD SP, SP, #msize 20 // 调用指令序列的开始
136: ST 0(SP), #152      // 将返回地址压入栈
144: BR 300              // 调用q
152: SUB SP, SP, #msize  // 恢复SP的值
160: ACTION2
180: HALT
...

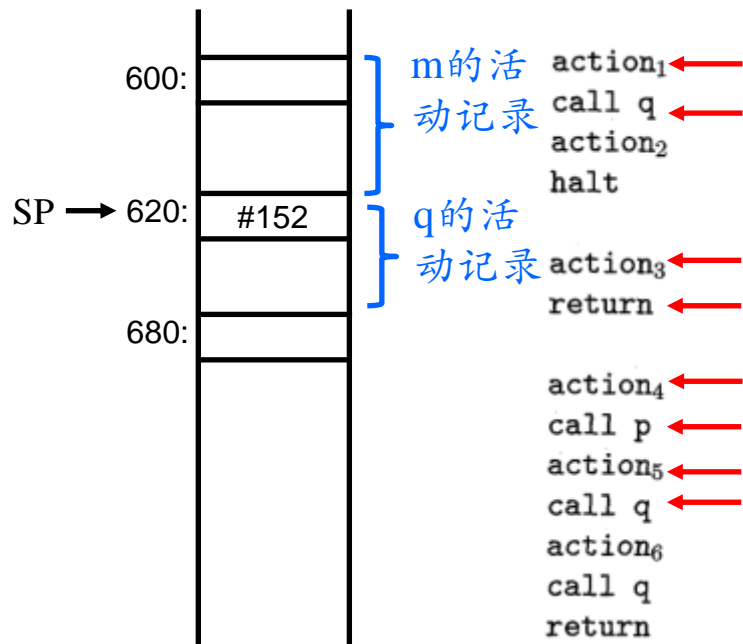
// p的代码
200: ACTION3
220: BR *0(SP)           // 返回
...

// q的代码
300: ACTION4           // 包含有跳转到456的条件转移指令
320: ADD SP, SP, #qsize 60
328: ST 0(SP), #344      // 将返回地址压入栈
336: BR 200              // 调用p
344: SUB SP, SP, #qsize
352: ACTION5
372: ADD SP, SP, #qsize
380: ST 0(SP), #396      // 将返回地址压入栈
388: BR 300              // 调用q
396: SUB SP, SP, #qsize
404: ACTION6
424: ADD SP, SP, #qsize
432: ST 0(SP), #440      // 将返回地址压入栈
440: BR 300              // 调用q
448: SUB SP, SP, #qsize
456: BR *0(SP)           // 返回
...

600: // 栈区的开始处
    
```

# 使用栈式内存分配的方式

三地址语句	call callee	return
目标代码 (栈式)	ADD SP, SP, #caller.recordsize ST 0(SP), #here + 16 BR callee.codeArea	<ul style="list-style-type: none"> <li>被调用过程 BR *0(SP)</li> <li>调用过程 SUB SP, SP, #caller.recordsize</li> </ul>



// code for m

action<sub>1</sub>  
call q  
action<sub>2</sub>  
halt

// code for p

action<sub>3</sub>  
return

// code for q

action<sub>4</sub>  
call p  
action<sub>5</sub>  
call q  
action<sub>6</sub>  
call q  
return

```

100: LD SP, #600           // m的代码
108: ACTION1              // 初始化栈
128: ADD SP, SP, #msize 20 // 调用指令序列的开始
136: ST 0(SP), #152       // 将返回地址压入栈
144: BR 300               // 调用q
152: SUB SP, SP, #msize   // 恢复SP的值
160: ACTION2
180: HALT
...

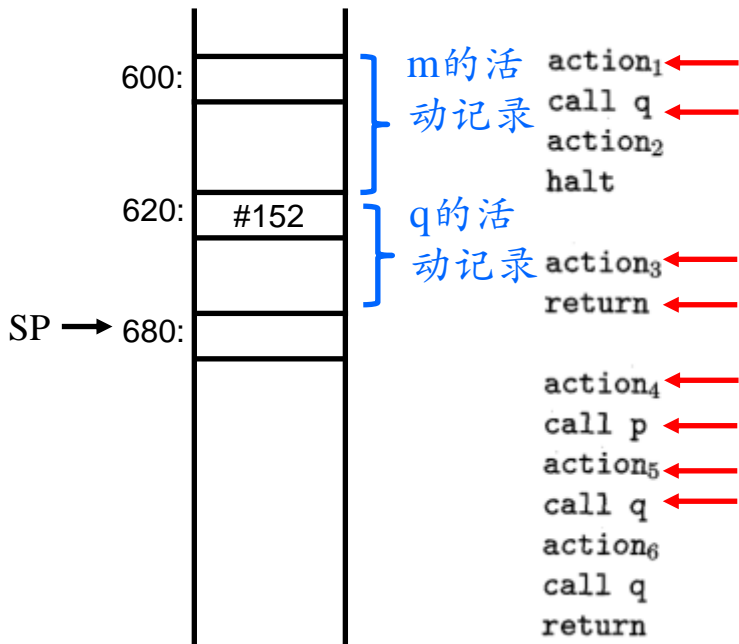
200: ACTION3              // p的代码
220: BR *0(SP)            // 返回
...

300: ACTION4              // q的代码
320: ADD SP, SP, #qsize 60 // 包含有跳转到456的条件转移指令
328: ST 0(SP), #344       // 将返回地址压入栈
336: BR 200               // 调用p
344: SUB SP, SP, #qsize
352: ACTION5
372: ADD SP, SP, #qsize
380: ST 0(SP), #396       // 将返回地址压入栈
388: BR 300               // 调用q
396: SUB SP, SP, #qsize
404: ACTION6
424: ADD SP, SP, #qsize
432: ST 0(SP), #440       // 将返回地址压入栈
440: BR 300               // 调用q
448: SUB SP, SP, #qsize
456: BR *0(SP)            // 返回
...

600:                      // 栈区的开始处
    
```

# 使用栈式内存分配的方式

三地址语句	call callee	return
目标代码 (栈式)	ADD SP, SP, #caller.recordsize ST 0(SP), #here + 16 BR callee.codeArea	<ul style="list-style-type: none"> <li>被调用过程 BR *0(SP)</li> <li>调用过程 SUB SP, SP, #caller.recordsize</li> </ul>



// code for m

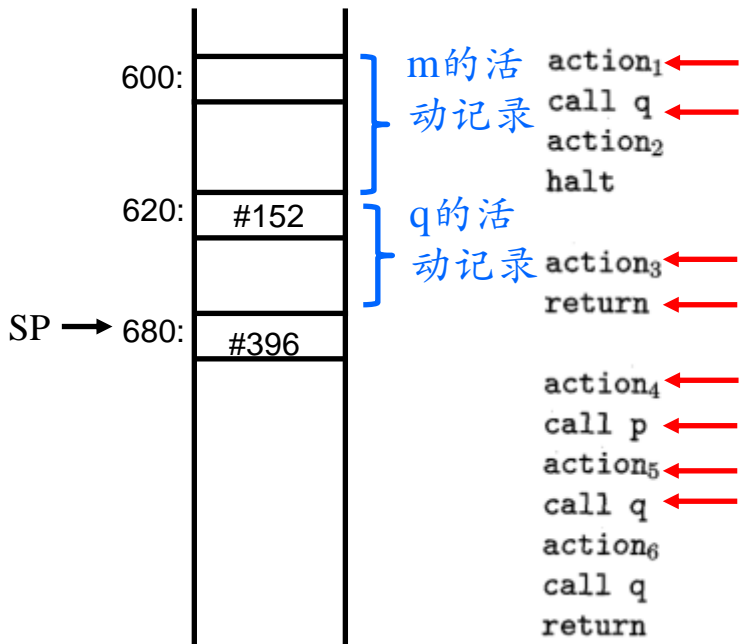
// code for p

// code for q

100:	LD SP, #600	// m的代码
108:	ACTION <sub>1</sub>	// 初始化栈
128:	ADD SP, SP, #msize	// 调用指令序列的开始
136:	ST 0(SP), #152	// 将返回地址压入栈
144:	BR 300	// 调用q
152:	SUB SP, SP, #msize	// 恢复SP的值
160:	ACTION <sub>2</sub>	
180:	HALT	
...		
200:	ACTION <sub>3</sub>	// p的代码
220:	BR *0(SP)	// 返回
...		
300:	ACTION <sub>4</sub>	// q的代码
320:	ADD SP, SP, #qsize	// 包含有跳转到456的条件转移指令
328:	ST 0(SP), #344	// 将返回地址压入栈
336:	BR 200	// 调用p
344:	SUB SP, SP, #qsize	
352:	ACTION <sub>5</sub>	
372:	ADD SP, SP, #qsize	
380:	ST 0(SP), #396	// 将返回地址压入栈
388:	BR 300	// 调用q
396:	SUB SP, SP, #qsize	
404:	ACTION <sub>6</sub>	
424:	ADD SP, SP, #qsize	
432:	ST 0(SP), #440	// 将返回地址压入栈
440:	BR 300	// 调用q
448:	SUB SP, SP, #qsize	
456:	BR *0(SP)	// 返回
...		
600:		// 栈区的开始处

# 使用栈式内存分配的方式

三地址语句	call callee	return
目标代码 (栈式)	ADD SP, SP, #caller.recordsize ST 0(SP), #here + 16 BR callee.codeArea	<ul style="list-style-type: none"> <li>被调用过程 BR *0(SP)</li> <li>调用过程 SUB SP, SP, #caller.recordsize</li> </ul>



// code for m

// code for p

// code for q

100:	LD SP, #600	// m的代码
108:	ACTION <sub>1</sub>	// 初始化栈
128:	ADD SP, SP, #msize	// 调用指令序列的开始
136:	ST 0(SP), #152	// 将返回地址压入栈
144:	BR 300	// 调用q
152:	SUB SP, SP, #msize	// 恢复SP的值
160:	ACTION <sub>2</sub>	
180:	HALT	
...		
200:	ACTION <sub>3</sub>	// p的代码
220:	BR *0(SP)	// 返回
...		
300:	ACTION <sub>4</sub>	// q的代码
320:	ADD SP, SP, #qsize	// 包含有跳转到456的条件转移指令
328:	ST 0(SP), #344	// 将返回地址压入栈
336:	BR 200	// 调用p
344:	SUB SP, SP, #qsize	
352:	ACTION <sub>5</sub>	
372:	ADD SP, SP, #qsize	
380:	ST 0(SP), #396	// 将返回地址压入栈
388:	BR 300	// 调用q
396:	SUB SP, SP, #qsize	
404:	ACTION <sub>6</sub>	
424:	ADD SP, SP, #qsize	
432:	ST 0(SP), #440	// 将返回地址压入栈
440:	BR 300	// 调用q
448:	SUB SP, SP, #qsize	
456:	BR *0(SP)	// 返回
...		
600:		// 栈区的开始处





# 提纲

9.1 代码生成器的主要任务

9.2 一个简单的目标机模型

9.3 指令选择

9.4 寄存器的选择

9.5 窥孔优化



## 三地址语句的目标代码生成

- 对每个形如  $x = y \text{ op } z$  的三地址指令  $I$ ，执行如下动作
  - 调用函数  $getReg(I)$  来为  $x$ 、 $y$ 、 $z$  选择寄存器，把这些寄存器称为  $R_x$ 、 $R_y$ 、 $R_z$
  - 如果  $R_y$  中存放的不是  $y$ ，则生成指令 “ $LD R_y, y'$ ”。 $y'$  是存放  $y$  的内存位置之一
  - 类似的，如果  $R_z$  中存放的不是  $z$ ，生成指令 “ $LD R_z, z'$ ”
  - 生成目标指令 “ $OP R_x, R_y, R_z$ ”

# 寄存器描述符和地址描述符

## ➤ 寄存器描述符 (*register descriptor*)

- 记录每个寄存器当前存放的是哪些变量的值

## ➤ 地址描述符 (*address descriptor*)

- 记录运行时每个名字的当前值存放在哪个或哪些位置

- 该位置可能是寄存器、栈单元、内存地址或者是它们的某个集合

- 这些信息可以存放在该变量名对应的符号表条目中

## 基本块的收尾处理

- 在基本块结束之前，基本块中使用的变量可能仅存放在某个寄存器中
- 如果这个变量是一个只在基本块内部使用的临时变量，当基本块结束时，可以忘记这些临时变量的值并假设这些寄存器是空的
- 对于一个在基本块的出口处可能活跃的变量 $x$ ，如果它的地址描述符表明它的值没有存放在 $x$ 的内存位置上，则生成指令“ $ST\ x, R$ ”（ $R$ 是在基本块结尾处存放 $x$ 值的寄存器）

# 管理寄存器和地址描述符

- 当代码生成算法生成加载、保存和其他指令时，它必须同时更新寄存器和地址描述符
- 对于指令 “ $LD\ R, x$ ”
  - 修改  $R$  的寄存器描述符，使之只包含  $x$
  - 修改  $x$  的地址描述符，把  $R$  作为新增位置加入到  $x$  的位置集合中
  - 从任何不同于  $x$  的地址描述符中删除  $R$

# 管理寄存器和地址描述符

- 当代码生成算法生成加载、保存和其他指令时，它必须同时更新寄存器和地址描述符
- 对于指令 “ $LD\ R, x$ ”
- 对于指令 “ $OP\ R_x, R_y, R_z$ ”
  - 修改  $R_x$  的寄存器描述符，使之只包含  $x$
  - 从任何不同于  $R_x$  的寄存器描述符中删除  $x$
  - 修改  $x$  的地址描述符，使之只包含位置  $R_x$
  - 从任何不同于  $x$  的地址描述符中删除  $R_x$

# 管理寄存器和地址描述符

- 当代码生成算法生成**加载**、**保存**和其他指令时，它必须同时更新寄存器和地址描述符
- 对于指令 “ $LD\ R, x$ ”
- 对于指令 “ $OP\ R_x, R_y, R_z$ ”
- 对于指令 “ $ST\ x, R$ ”
  - 修改 **$x$** 的**地址描述符**，使之包含自己的内存位置

# 管理寄存器和地址描述符

- 当代码生成算法生成**加载**、**保存**和其他指令时，它必须同时更新寄存器和地址描述符
  - 对于指令 “ $LD\ R, x$ ”
  - 对于指令 “ $OP\ R_x, R_y, R_z$ ”
  - 对于指令 “ $ST\ x, R$ ”
  - 对于复制语句  $x=y$ ，如果需要生成加载指令 “ $LD\ R_y, y'$ ” 则
    - 修改  $R_y$  的寄存器描述符，使之只包含  $y$
    - 修改  $y$  的地址描述符，把  $R_y$  作为新增位置加入到  $y$  的位置集合中
    - 从任何不同于  $y$  的变量的地址描述符中删除  $R_y$
    - **修改  $R_y$  的寄存器描述符，使之也包含  $x$**
    - **修改  $x$  的地址描述符，使之只包含  $R_y$**

# 例

$$t = a - b$$

$$u = a - c$$

$$v = t + u$$

$$a = d$$

$$d = v + u$$



```
LD R1 , a
LD R2 , b
SUB R2 , R1 , R2
```

<i>R1</i>	<i>R2</i>	<i>R3</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>t</i>	<i>u</i>	<i>v</i>
<i>a</i>	<i>t</i>		<i>a, R1</i>	<i>b, R2</i>	<i>c</i>	<i>d</i>	<i>R2</i>		



# 例

$$t = a - b$$

$$u = a - c$$

$$v = t + u$$

$$a = d$$

$$d = v + u$$

*LD R3 , c*  
*SUB R1 , R1 , R3*

<i>R1</i>	<i>R2</i>	<i>R3</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>t</i>	<i>u</i>	<i>v</i>
<i>u</i>	<i>t</i>	<i>c</i>	<i>a, R1</i>	<i>b</i>	<i>c, R3</i>	<i>d</i>	<i>R2</i>	<i>R1</i>	

# 例

$$t = a - b$$

$$u = a - c$$

$$v = t + u \longrightarrow \text{ADD } R3, R2, R1$$

$$a = d$$

$$d = v + u$$

<i>R1</i>	<i>R2</i>	<i>R3</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>t</i>	<i>u</i>	<i>v</i>
<i>u</i>	<i>t</i>	<i>v</i>	<i>a</i>	<i>b</i>	<i>c, R3</i>	<i>d</i>	<i>R2</i>	<i>R1</i>	<i>R3</i>

# 例

$$t = a - b$$

$$u = a - c$$

$$v = t + u$$

$$a = d$$



*LD R2, d*

$$d = v + u$$

<i>R1</i>	<i>R2</i>	<i>R3</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>t</i>	<i>u</i>	<i>v</i>
<i>u</i>	<i>d, a</i>	<i>v</i>	<i>R2</i>	<i>b</i>	<i>c</i>	<i>d, R2</i>	<i>R2</i>	<i>R1</i>	<i>R3</i>

# 例

$$t = a - b$$

$$u = a - c$$

$$v = t + u$$

$$a = d$$

$$d = v + u$$

*ADD R1 , R3 , R1*

<i>R1</i>	<i>R2</i>	<i>R3</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>t</i>	<i>u</i>	<i>v</i>
<i>d</i>	<i>d , a</i>	<i>v</i>	<i>R2</i>	<i>b</i>	<i>c</i>	<i>R1</i>		<i>R1</i>	<i>R3</i>

# 例

$$t = a - b$$

$$u = a - c$$

$$v = t + u$$

$$a = d$$

$$d = v + u$$

*exit*

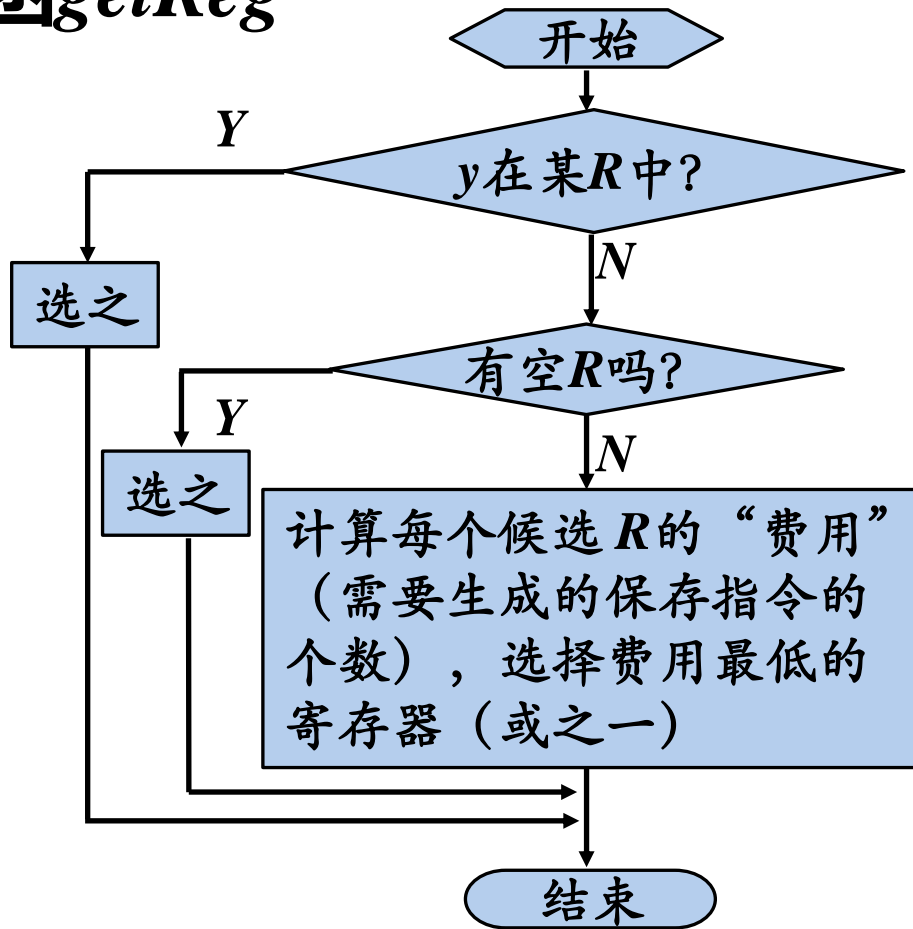
*ST a , R2*  
*ST d , R1*

<i>R1</i>	<i>R2</i>	<i>R3</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>t</i>	<i>u</i>	<i>v</i>
<i>d</i>	<i>a</i>	<i>v</i>	<i>R2 ,a</i>	<i>b</i>	<i>c</i>	<i>R1 ,d</i>			<i>R3</i>

# 寄存器选择函数 *getReg*

$x = y \text{ op } z$

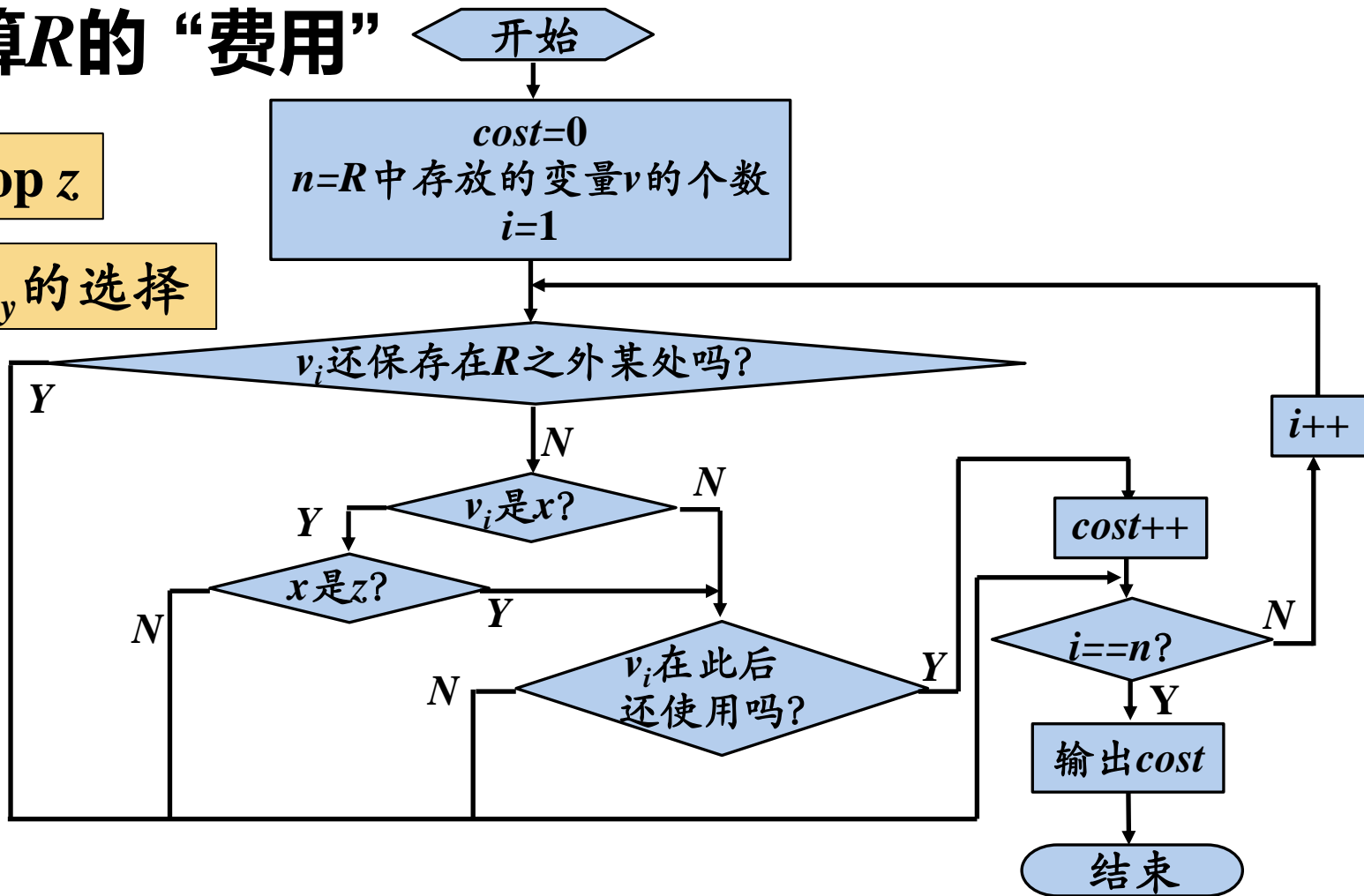
寄存器  $R_y$  的选择



# 计算 $R$ 的“费用”

$x = y \text{ op } z$

寄存器 $R_y$ 的选择



## 寄存器 $R_x$ 的选择

$$x = y \text{ op } z$$

- 选择方法与 $R_y$ 类似，区别之处在于
  - 因为 $x$ 的一个新值正在被计算，因此只存放了 $x$ 的值的寄存器对 $R_x$ 来说总是可接受的，即使 $x$ 就是 $y$ 或 $z$ 之一(因为我们的机器指令允许一个指令中的两个寄存器相同)
  - 如果 $y$ 在指令 $I$ 之后不再使用，且(在必要时加载 $y$ 之后) $R_y$ 仅仅保存了 $y$ 的值，那么， $R_y$ 同时也可以用作 $R_x$ 。对 $z$ 和 $R_z$ 也有类似选择

当 $I$ 是复制指令 $x=y$ 时，选择好 $R_y$ 后，令 $R_x = R_y$





# 提纲

9.1 代码生成器的主要任务

9.2 一个简单的目标机模型

9.3 指令选择

9.4 寄存器的选择

9.5 窥孔优化

## 9.5 窥孔优化

- **窥孔**(*peephole*)是程序上的一个小的滑动窗口
- **窥孔优化**是指在优化的时候，检查目标指令的一个滑动窗口(即窥孔)，并且只要有可能就在窥孔内用更快或更短的指令来替换窗口中的指令序列
- 也可以在**中间代码生成**之后直接应用窥孔优化来提高中间表示形式的质量

# 具有窥孔优化特点的程序变换的例子

- 冗余指令删除
- 控制流优化
- 代数优化
- 机器特有指令的使用

# 冗余指令删除

## ➤ 消除冗余的加载和保存指令

### ➤ 例

#### 三地址指令序列

➤  $a = b + c$

➤  $d = a + e$

#### 目标代码

➤  $LD \quad R0, \quad b \quad \quad \quad // \quad R0 = b$

➤  $ADD \quad R0, \quad R0, \quad c \quad \quad // \quad R0 = R0 + c$

➤  $ST \quad a \quad , \quad R0 \quad \quad \quad // \quad a = R0$

➤  $LD \quad R0, \quad a \quad \quad \quad // \quad R0 = a$

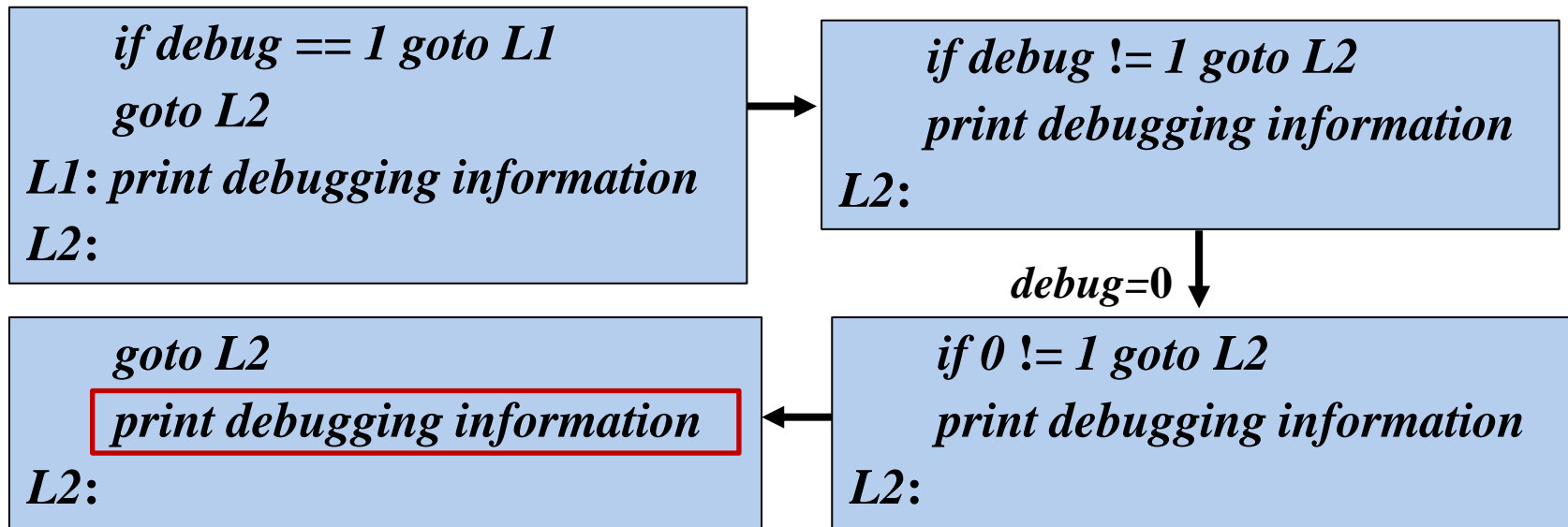
➤  $ADD \quad R0, \quad R0, \quad e \quad \quad // \quad R0 = R0 + e$

➤  $ST \quad d \quad , \quad R0 \quad \quad \quad // \quad d = R0$

如果第四条指令有标号，则不可以删除

# 冗余指令删除

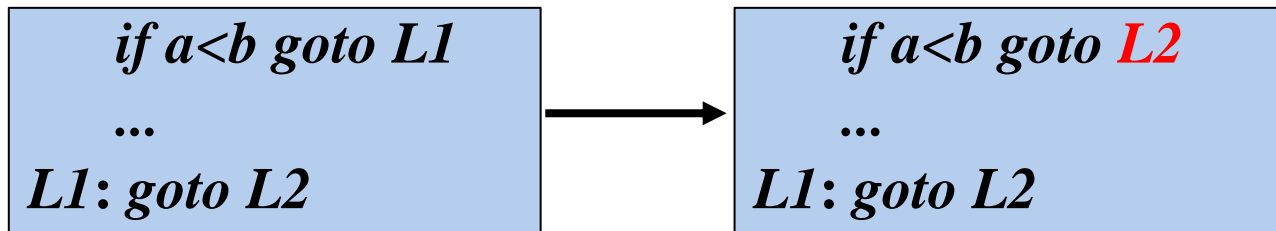
- 消除冗余的加载和保存指令
- 消除不可达代码
  - 一个紧跟在无条件跳转之后的不带标号的指令可以被删除
  - 例



# 控制流优化

➤ 在代码中出现**跳转到跳转指令的指令**时，某些条件下可以使用**一个跳转指令**来代替

➤ 例



如果不再有跳转到*L1*的指令，并且语句*L1: goto L2*之前是一个**无条件跳转指令**，则可以删除该语句

# 代数优化

## ➤ 代数恒等式

➤ 消除窥孔中类似于 $x=x+0$ 或 $x=x*1$ 的运算指令

## ➤ 强度削弱

➤ 对于乘数(除数)是2的幂的定点数乘法(除法)，用移位运算实现代价比较低

➤ 除数为常量的浮点数除法可以通过乘数为该常量倒数的乘法来求近似值

# 特殊指令的使用

- 充分利用目标系统的某些高效的特殊指令来提高代码效率
- 例如：*INC*指令可以用来替代加1的操作





# 结束

