第十一章：代码生成

提纲

代码生成器的主要任务

- 指令选择
  - 代码生成器必须把中间表示(*IR*)语句映射成为可以在目标机上运行的代码序列。
    - 决定该映射复杂性的因素
      - *IR*的层次

        如果IR是高层次的，代码生成器就要使用代码模板把每个IR语句翻译成机器指令序列-通常质量不佳。

        如果IR中反映了相关计算机的某些低层次细节，就可以生成更加高效的代码。

代码生成器的主要任务

- 指令选择
  - 代码生成器必须把中间表示(*IR*)语句映射成为可以在目标机上运行的代码序列。
    - 决定该映射复杂性的因素
      - *IR*的层次
      - 指令集体系结构本身的特性
        - 指令集的统一性和完整性是两个很重要的因素。否则总体规则的每个例外都要需要进行特别处理。

          eg:某些机器上浮点数运算使用单独的寄存器完成。
        - 指令速度和机器的特有用法是另外一些重要因素

代码生成器的主要任务

- 指令选择
  - 代码生成器必须把中间表示(IR)语句映射成为可以在目标机上运行的代码序列。
    - 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
  - 代码生成器必须把中间表示(*IR*)语句映射成为可以在目标机上运行的代码序列。
    - 决定该映射复杂性的因素
      - *IR*的层次
      - 指令集体系结构本身的特性
      - 想要达到的生成代码的质量

        通常由生成代码的运行速度和大小来确定。

例： **a = a+1**

INC a（加一指令）

```
LD   R0, a
ADD  R0 = R0 + 1
ST   a, 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
- 寄存器分配和指派
  - 把哪个值放在哪个寄存器中

几十个~上百个寄存器

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
- 寄存器分配和指派
  - 把哪个值放在哪个寄存器中
  - 分配：对于源程序中的每个点，我们选择一组将被存放在寄存器中的变量
  - 指派：我们指定一个变量被存放在哪个寄存器中

即使对于单寄存器机器，找到一个从寄存器到变量的最优指派也是很困难的。NP完全问题。

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
- 寄存器分配和指派
  - 把哪个值放在哪个寄存器中
- 求值顺序
  - 按照什么顺序来安排指令的执行

    找到最好的顺序是一个困难的NP完全问题

提纲

一个简单的目标机模型

- 三地址机器模型
  - 加载、保存、计算、跳转和条件跳转操作
  - 内存按字节寻址
  - 有*n*个通用寄存器*R0, R1, …, Rn-1*
  - 假设所有的运算分量都是整数
  - 指令之前可能有一个标号

目标机器的主要指令

- 加载运算　　*LD dst, addr*
  - *LD r, x*
  - *LD r_1, r_2*
- 保存运算　　*ST x, r*
- 计算运算　　*OP dst, src1, src2*
  - 例: *SUB r1, r2, r3*　　*r1 = r2 - r3*
  - 只需要一个运算分量的单目运算符没有*src2*

目标机器的主要指令

- 加载运算　　*LD dst, addr*
  - *LD r, x*
  - *LD r_1, r_2*
- 保存运算　　*ST x, r*
- 计算运算　　*OP dst, src1, src2*
- 无条件跳转　　*BR L*
- 条件跳转　　*Bcond r, L*
  - 例: *BLTZ r, L*

寻址模式

- 一个位置可以是变量名*a*
  - 例：*LD R1 , a*
    - *R1 = contents (a)*

寻址模式

- 一个位置可以是变量名*a*
- 一个位置可以是带下标的地址*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*)
    - *R1* = *contents* (*a* + *contents*(*R2*))

寻址模式

- 一个位置可以是变量名*a*
- 一个位置可以是带下标的地址*a*(*r*)
- 一个内存位置可以是一个以寄存器作为下标的整数*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)*
    - *R1 = contents (contents(R2) + 100)*

寻址模式

- 一个位置可以是变量名*a*
- 一个位置可以是带下标的地址*a*(*r*)
- 一个内存位置可以是一个以寄存器作为下标的整数*c*(*r*)
- 在寄存器*r*的内容所表示的位置上存放的内存位置**r*
  - 例：*LD R1 , * R2*
    - *R1 = conents (contents (contents (R2)))*

寻址模式

- 一个位置可以是变量名*a*
- 一个位置可以是带下标的地址*a*(*r*)
- 一个内存位置可以是一个以寄存器作为下标的整数*c*(*r*)
- 在寄存器r的内容所表示的位置上存放的内存位置**r*
- 在*r*中的内容+100的和所表示的位置上的内容代表的位置 **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*)
    - *R1 = conents (contents (contents*(*R2*) + *100*))

寻址模式

- 一个位置可以是变量名*a*
- 一个位置可以是带下标的地址*a*(*r*)
- 一个内存位置可以是一个以寄存器作为下标的整数*c*(*r*)
- 在寄存器r的内容所表示的位置上存放的内存位置**r*
- 在*r*中内容+100的和所表示的位置上的内容代表的位置**c*(*r*)
- 直接常数寻址#*c*
  - 例：*LD R1，#100*
    - *R1 = 100*
  - 例：*ADD R1，R1 ， #100*
    - *R1 = R1 + 100*

提纲

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*是一个整数数组，每个整数占4个字节
- 目标代码
  - *LD R1 , i // R1 = i*
  - *MUL R1 , R1, 4 // R1=R1 * 4*
  - *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*是一个整数数组，每个整数占4个字节
- 目标代码
  - *LD R1 , c* // *R1 = c*
  - *LD R2 , j* // *R2 = j*
  - *MUL R2 , R2 , 4* // *R2 = R2 * 4*
  - *ST a(R2) , R1* // *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 // R1 = p
  - LD R2 , y // R2 = y
  - ST 0(R1), R2 //contents (0 + 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

例

- 假设a和b是元素为4字节值的数组，为下面的三地址语句序列生成代码：

```
x = a [i]
y = b [j]
a [i] = y
b [j] = x
```

例

- 假设p和q都存放在内存位置中，为下面的三地址语句序列生成代码：

```
y = * q
q = q +4
* p = y
p = p + 4
```

目标代码中的地址

如何使用静态和栈式内存分配为简单的过程调用和返回生成代码，依次将IR中的名字转换为目标代码中的地址

- 一个静态确当的代码区*Code*
- 一个静态确定的静态数据区*Static*
- 一个动态管理的堆区*Heap*
- 一个动态管理的栈区*Stack*

使用静态分配时的代码生成

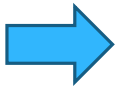
返回地址

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

*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
```

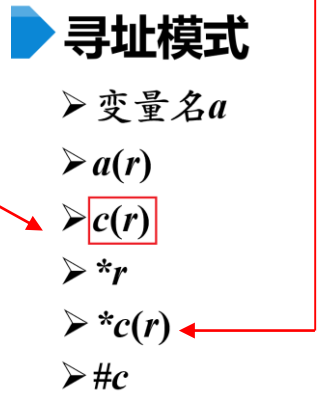


```
                    // c的代码
100:  ACTION1       // action1的代码
120:  ST 364, #140  // 在位置364上存放返回地址140
132:  BR 200        // 调用p
140:  ACTION2
160:  HALT          // 返回操作系统
      ...
                    // p的代码
200:  ACTION3
220:  BR *364       // 返回在位置364保存的地址处
      ...
                    // 300-363存放c的活动记录
300:                // 返回地址
304:                // c的局部数据
      ...
                    // 364-451存放p的活动记录
364:                // 返回地址
368:                // p的局部数据
```

静态分配的目标代码

使用栈式内存分配的代码生成

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	➢ 被调用过程 BR *0(SP) ➢ 调用过程 SUB SP , SP , #caller.recordsize



使用栈式内存分配的代码生

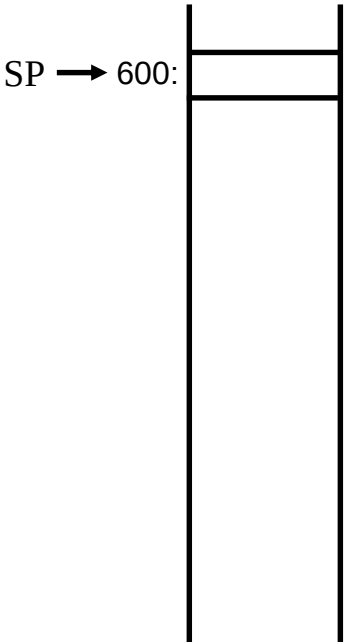
三地址语句	call callee	return
目标代码（栈式）	ADD SP, SP, #caller.recordsize ST 0(SP), #here + 16 BR callee.codeArea	▪ 被调用过程 BR *0(SP) ▪ 调用过程 SUB SP , SP , #caller.recordsize

```
                    // 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代码生

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

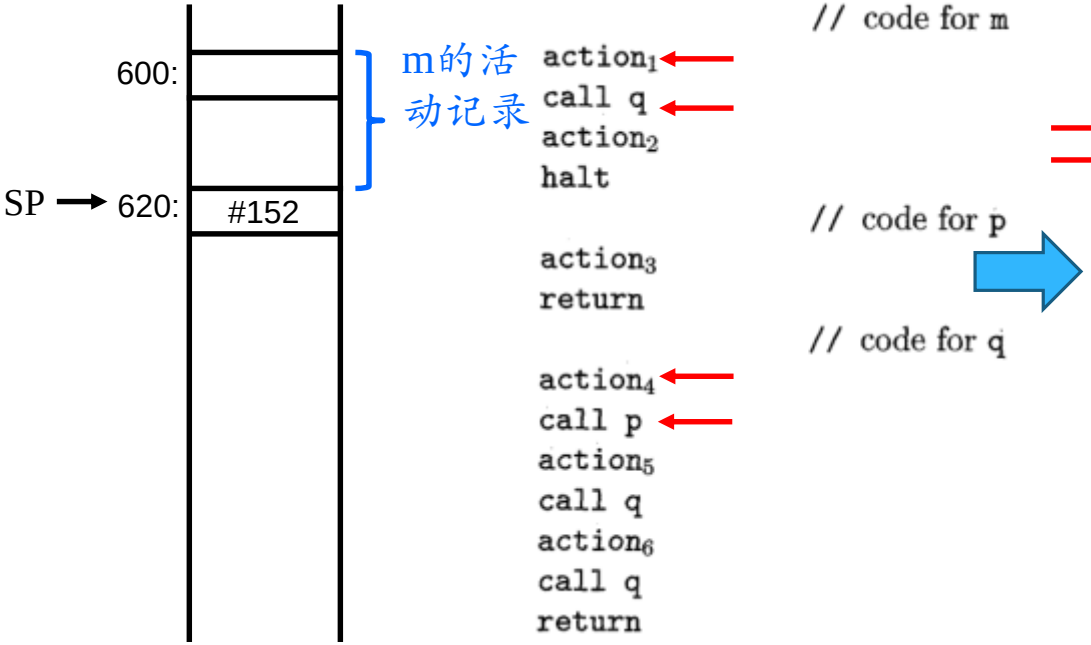



```
// code for m
action1
call q
action2
halt

// 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 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代码生

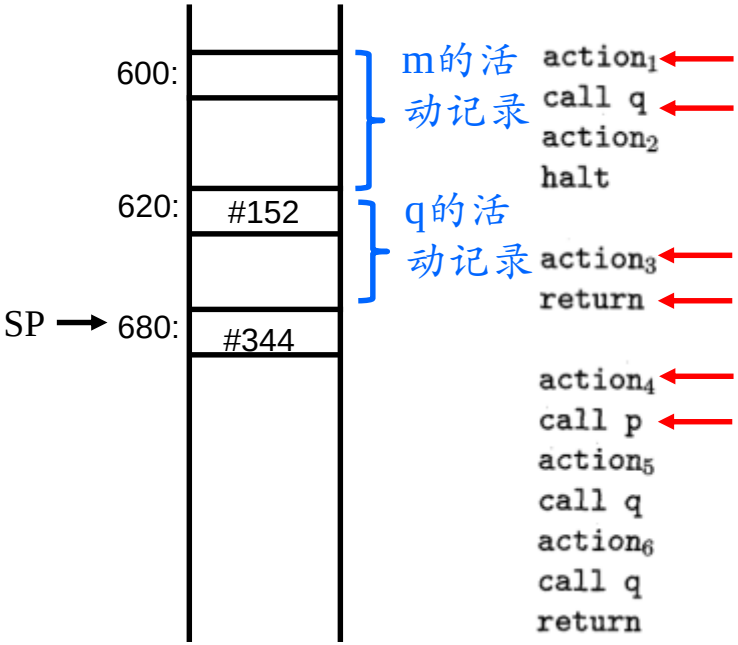
三地址语句	call callee	return
目标代码（栈式）	ADD SP, SP, #caller.recordsize ST 0(SP), #here + 16 BR callee.codeArea	▪ 被调用过程 BR *0(SP) ▪ 调用过程 SUB SP , SP , #caller.recordsize




```
                          // 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代码生

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

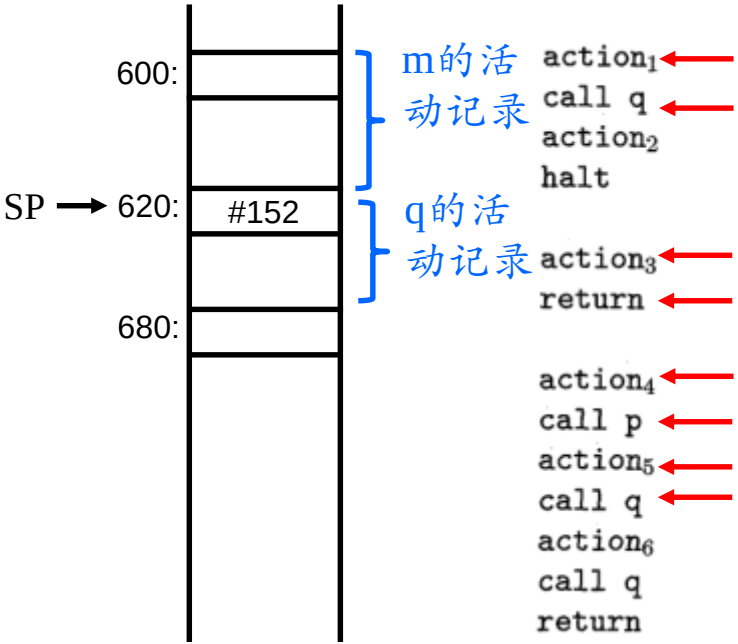



```
// code for m
action1
call q
action2
halt

// 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 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代码生

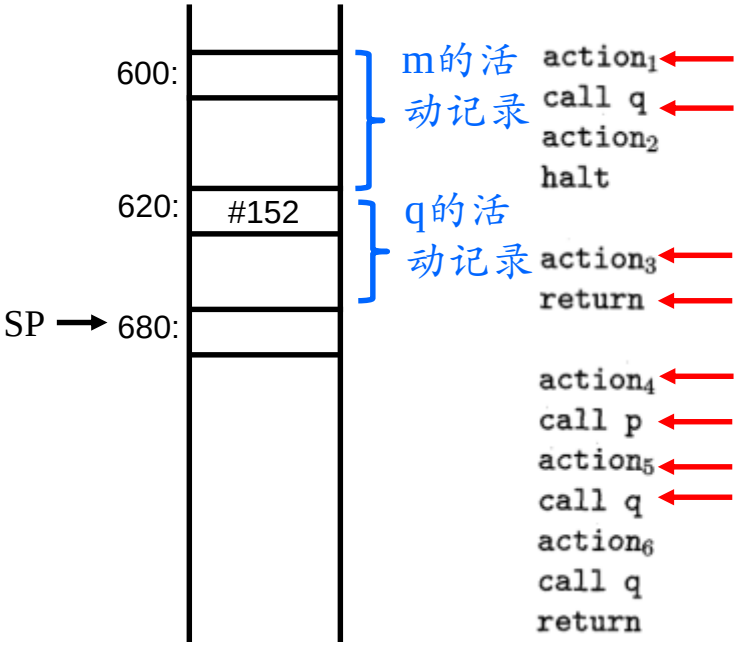
三地址语句	call callee	return
目标代码（栈式）	ADD SP, SP, #caller.recordsize ST 0(SP), #here + 16 BR callee.codeArea	▪ 被调用过程 BR *0(SP) ▪ 调用过程 SUB SP , SP , #caller.recordsize




```
                        // 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代码生

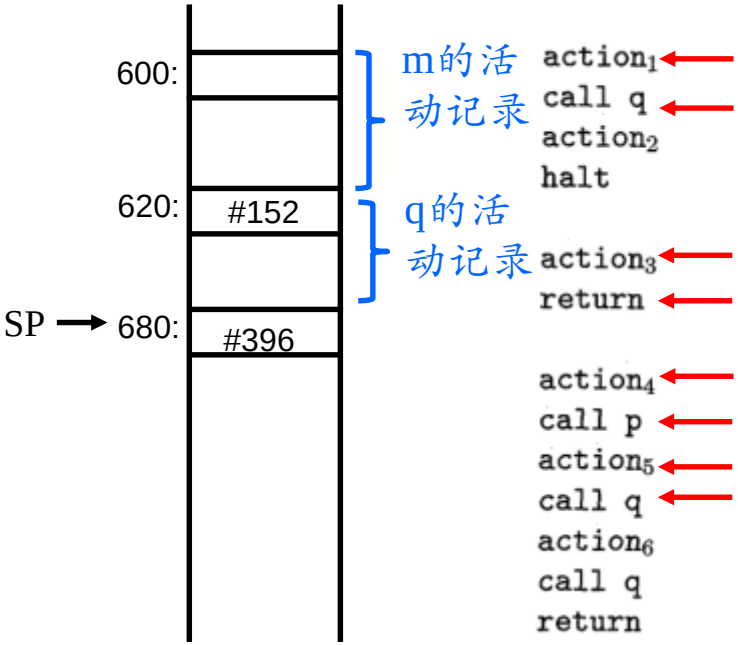
三地址语句	call callee	return
目标代码（栈式）	ADD SP, SP, #caller.recordsize ST 0(SP), #here + 16 BR callee.codeArea	▪ 被调用过程 BR *0(SP) ▪ 调用过程 SUB SP , SP , #caller.recordsize




```
                                // code for m
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代码生

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          // 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
```

20

60

提纲

三地址语句的目标代码生成

- 对每个形如$x = y\ \mathrm{op}\ z$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）

管理寄存器描述符和地址描述符

- 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描述符和地址描述符
  - 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

$t1 = a - b$

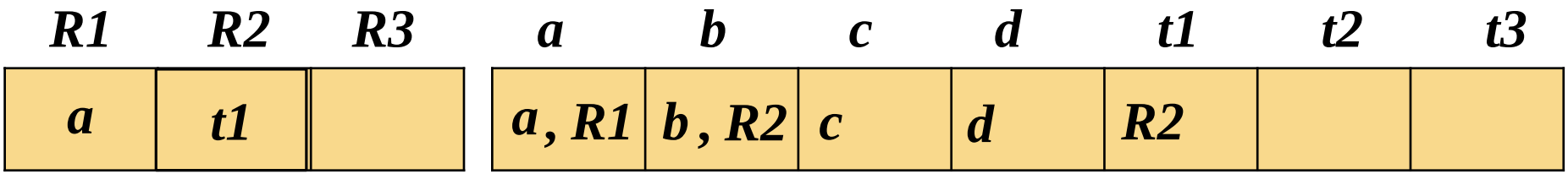
$t2 = a - c$

$t3 = t1 + t2$

$a = d$

$d = t3 + t2$

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



例

$t1 = a - b$

$t2 = a - c$

$t3 = t1 + t2$

$a = d$

$d = t3 + t2$

```
LD   R3 , c
SUB R1 , R1 , R3
```

R1	R2	R3
t2	t1	c

a	b	c	d	t1	t2	t3
a, R1	b	c ,R3	d	R2	R1	

例

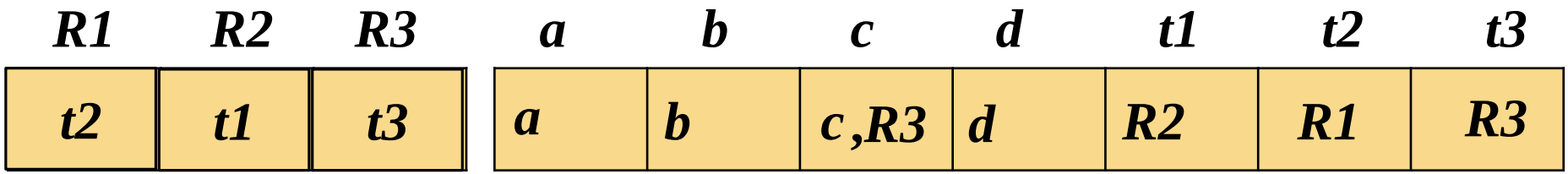
$t1 = a - b$

$t2 = a - c$

$t3 = t1 + t2$ → `ADD R3 , R2 , R1`

$a = d$

$d = t3 + t2$



例

$t1 = a - b$

$t2 = a - c$

$t3 = t1 + t2$

$a = d$ → LD R2 , d

$d = t3 + t2$

R1	R2	R3
t2	d , a	t3

a	b	c	d	t1	t2	t3
R2	b	c	d ,R2	R2	R1	R3

例

$t1 = a - b$

$t2 = a - c$

$t3 = t1 + t2$

$a = d$

$d = t3 + t2$ → `ADD R1 , R3 , R1`

R1	R2	R3	a	b	c	d	t1	t2	t3
d	d , a	t3	R2	b	c	R1		R1	R3

例

$t1 = a - b$

$t2 = a - c$

$t3 = t + t2$

$a = d$

$d = t3 + t2$

exit

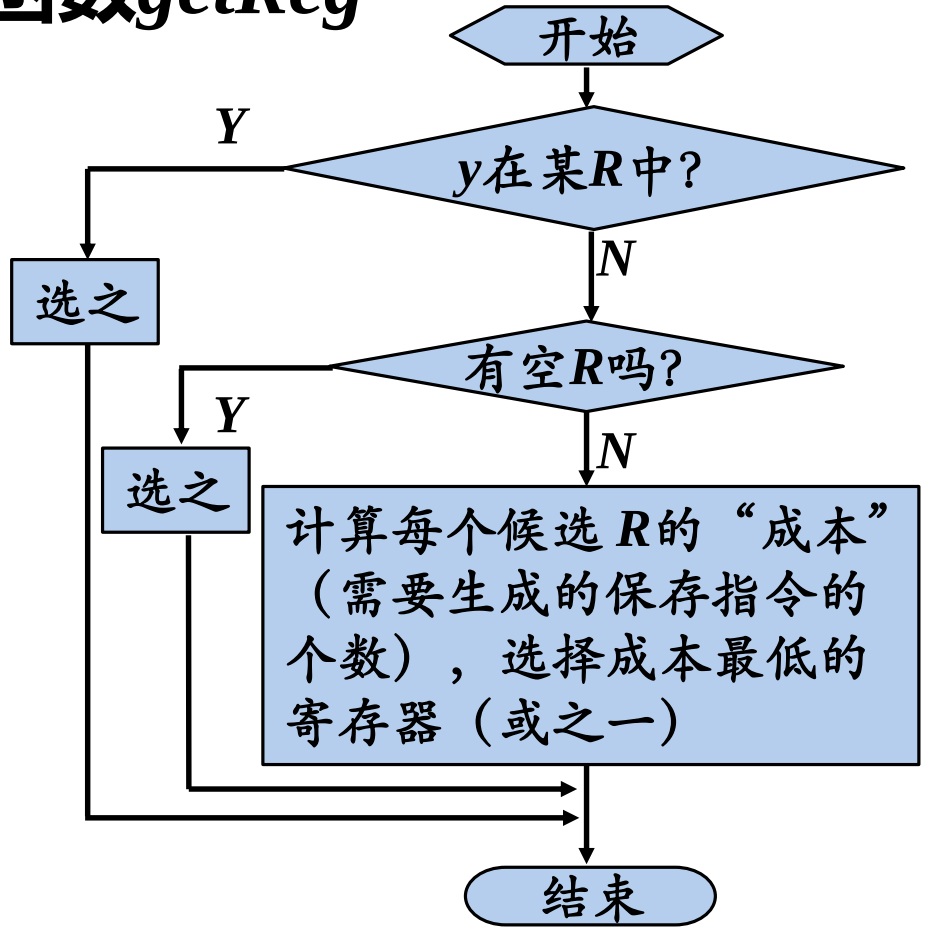
```
ST   a , R2
ST   d , R1
```

R1	R2	R3	a	b	c	d	t1	t2	t3
d	a	t3	R2 ,a	b	c	R1 ,d			R3

寄存器选择函数*getReg*

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

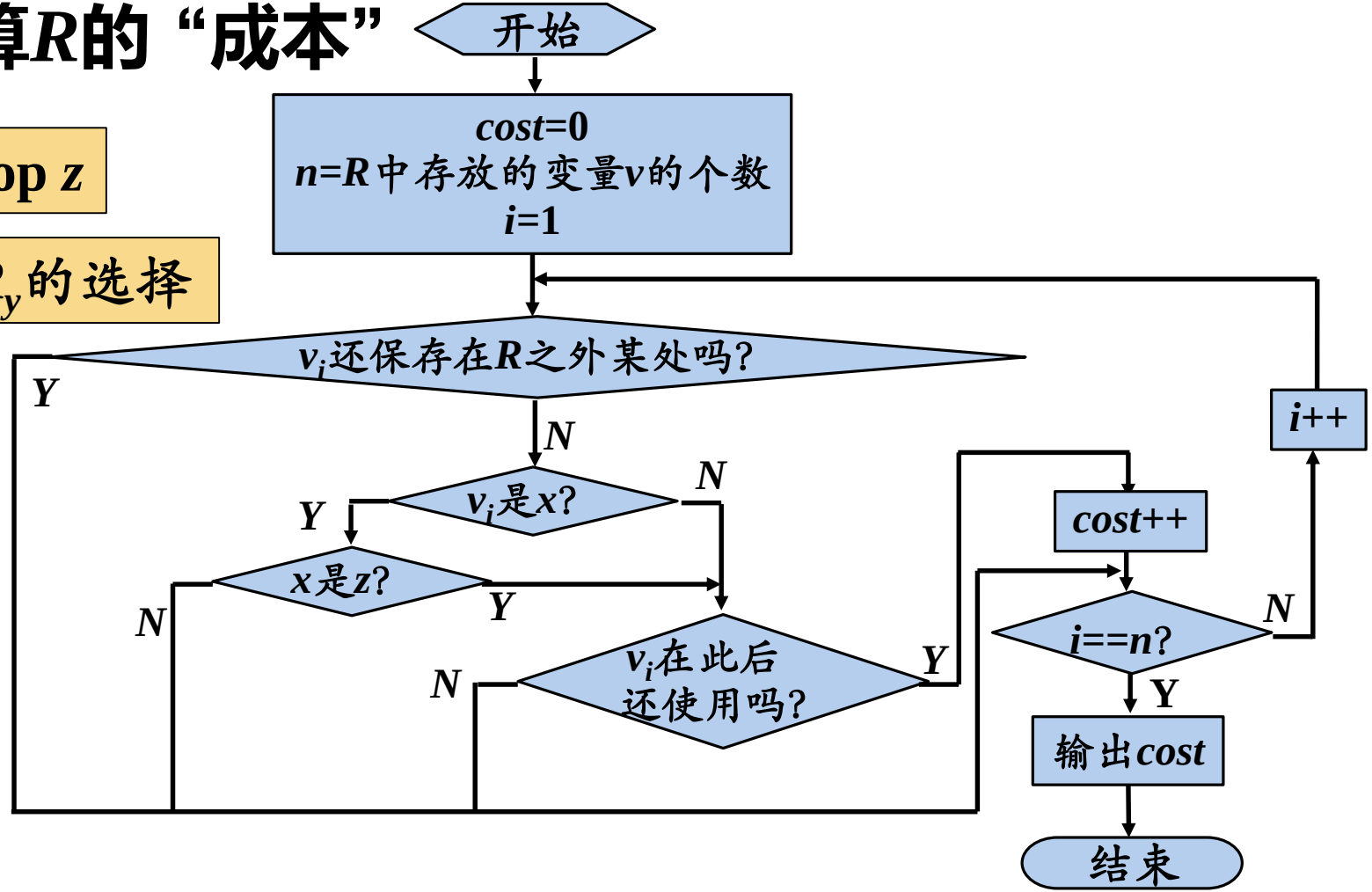
寄存器R_y的选择



计算R的“成本”

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

寄存器R_y的选择



寄存器R_x的选择

$x = y$ 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

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

- ①冗余指令删除
- ②控制流优化
- ③代数优化
- ④机器特有指令的使用

①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

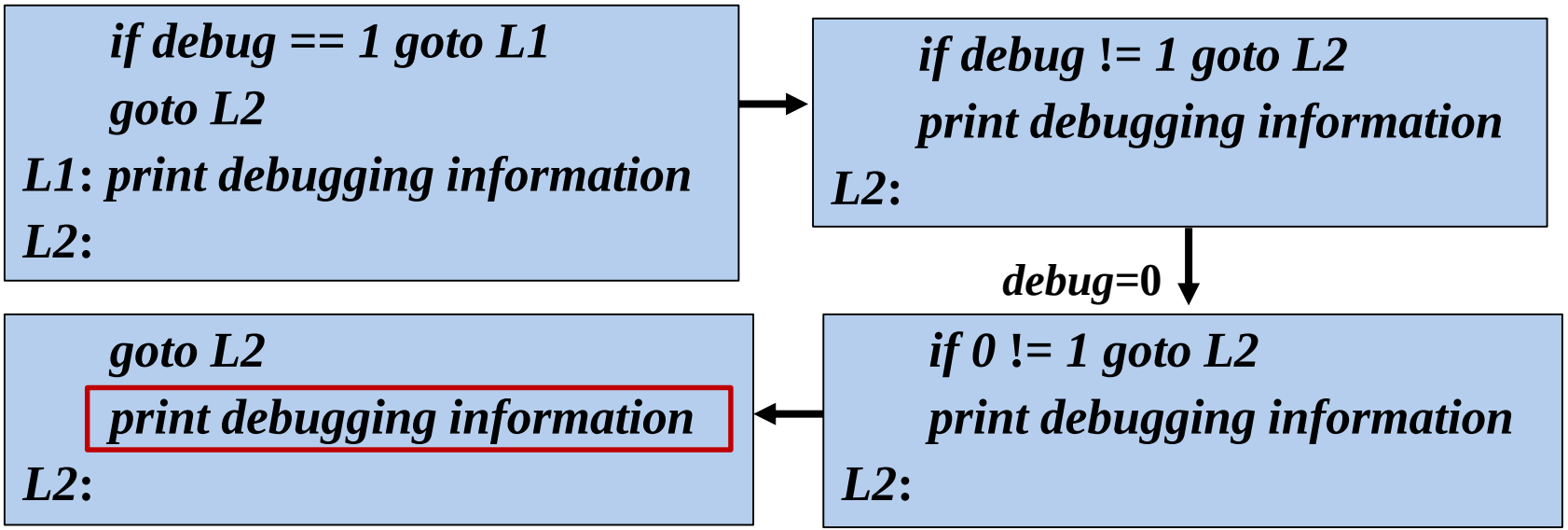
目标代码

- LD 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

如果第四条指令有标号，则不可以删除

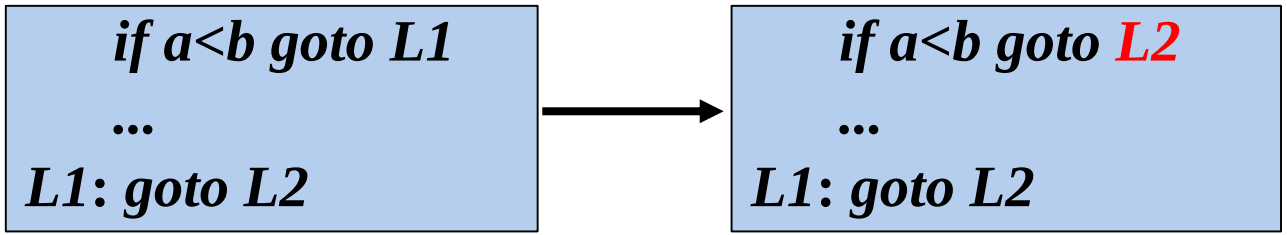
①冗余指令删除

- 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



②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

③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

④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