

# 第八章 代码生成

许畅

南京大学计算机学院

2025年春季

# 主要内容

---

- 代码生成器的设计
- 目标语言
- 目标代码中的地址
- 基本块和流图
- 基本块优化
- 代码生成器
- 寄存器分配
- 选择指令

# 代码生成器的位置

- 根据 **中间表示 (IR)** 生成代码
- 代码生成器之前可能有一个优化组件
- 代码生成器的三个任务
  - **指令选择**: 选择适当的指令实现IR语句
  - **寄存器分配和指派**: 把哪个值放在哪个寄存器中
  - **指令排序**: 按照什么顺序安排指令执行

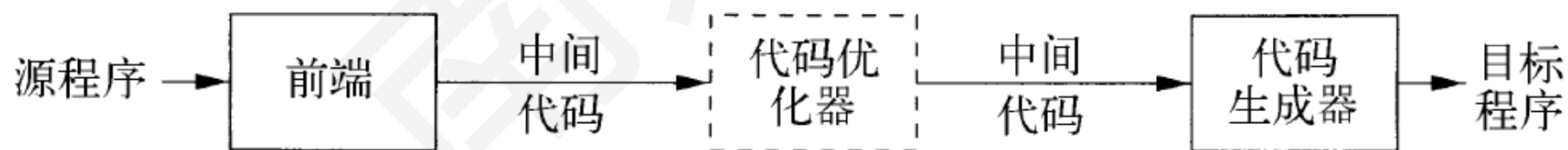


图 8-1 代码生成器的位置

# 要解决的问题

- 正确性：正确的机器指令
- 易于实现、测试和维护
- 输入IR的选择
  - 四元式、三元式、字节代码、堆栈机代码、后缀表示、抽象语法树、DAG图...
- 输出
  - **RISC** (目标机)、**CISC** (寻址方式)
  - 可重定向代码、**汇编语言**

# 目标机模型

- 使用三地址机器的模型
- 指令
  - 加载:  $LD\ dst, addr$  (把地址 $addr$ 中的内容加载到 $dst$ 所指的寄存器)
  - 保存:  $ST\ x, r$  (把寄存器 $r$ 中的内容保存到 $x$ 中)
  - 计算:  $OP\ dst, src_1, src_2$  (把 $src_1$ 和 $src_2$ 中的值运算后将结果存放到 $dst$ 中)
  - 无条件跳转:  $BR\ L$  (控制流转向标号 $L$ 的指令)
  - 条件跳转:  $Bcond\ r, L$  (对 $r$ 中的值进行测试, 如果为真则转向 $L$ )

# 寻址模式

- 变量 $x$ : 指向分配 $x$ 的内存位置
- $a(r)$ : 地址是 $a$ 的左值加上寄存器 $r$ 中的值
- $\text{constant}(r)$ : 寄存器 $r$ 中内容加上前面的常数即其地址
- $*r$ : 寄存器 $r$ 的内容所表示的位置上存放的内容位置
- $*\text{constant}(r)$ : 寄存器 $r$ 中内容加上常数所代表的位置上的内容所表示的位置
- 常数 $\# \text{constant}$

# 例子 (1)

- $x = y - z$ 
  - LD R1, y // R1 = y
  - LD R2, z // R2 = z
  - SUB R1, R1, R2 // R1 = R1 - R2
  - ST x, R1 // x = R1
- $b = a[i]$ 
  - LD R1, i // R1 = i
  - MUL R1, R1, 8 // R1 = R1 \* 8 (8字节长元素)
  - LD R2, a(R1) // R2 = contents(a + contents(R1))
  - ST b, R2 // b = R2

## 例子 (2)

- $a[j] = c$ 
  - LD R1, c                   // R1 = c
  - LD R2, j                   // R2 = j
  - MUL R2, R2, 8           // R2 = R2 \* 8 (8字节长元素)
  - ST a(R2), R1           // contents(a + contents(R2)) = R1
- $x = *p$ 
  - LD R1, p                   // R1 = p
  - LD R2, 0(R1)           // R2 = contents(0 + contents(R1))
  - ST x, R2                 // x = R2



## 例子 (3)

- $*p = y$ 
  - LD R1, p                   // R1 = p
  - LD R2, y                   // R2 = y
  - ST 0(R1), R2               // contents(0 + contents(R1)) = R2
- if  $x < y$  goto L
  - 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

# 程序和指令的代价

- 不同的目的有不同的度量
  - 最短编译时间、运行时间、目标程序大小、能耗
- 不可判定一个目标程序是否最优
- 假设每个指令有固定的代价，设定为1加上运算分量寻址模式的代价
  - LD R0, R1: 代价为1
  - LD R0, M: 代价是2
  - LD R1, \*100(R2): 代价为2

# 目标代码中的地址

- 如何为过程调用和返回生成代码？
  - 静态分配 (活动记录)
  - 栈式分配 (活动记录)
- 如何将IR中的名字 (过程名或变量名) 转换成为目标代码中的地址？
  - 不同区域中的名字采用不同的寻址方式

# 活动记录的静态分配

- 每个过程静态地分配一个数据区域，其开始位置用 *staticArea* 表示
- *call callee* 的实现
  - *ST callee.staticArea, #here + 20* // 存放返回地址
  - *BR callee.codeArea*
- *callee* 中的语句 *return*
  - *BR \*callee.staticArea*

# 例子

- 三地址代码

- $c$ 的代码

- $\text{action}_1$
    - $\text{call } p$
    - $\text{action}_2$
    - halt

- $p$ 的代码

- $\text{action}_3$
    - return

```
100: ACTION1
120: ST 364, #140
132: BR 200
140: ACTION2
160: HALT
```

...

```
200: ACTION3
220: BR *364
```

...

```
300:
304:
```

...

```
364:
368:
```

```
//  $c$  的代码
//  $\text{action}_1$  的代码
// 在位置 364 上存放返回地址 140
// 调用  $p$ 

// 返回操作系统
```

```
//  $p$  的代码

// 返回在位置 364 保存的地址处
```

```
// 300-363 存放  $c$  的活动记录
// 返回地址
//  $c$  的局部数据
```

```
// 364-451 存放  $p$  的活动记录
// 返回地址
//  $p$  的局部数据
```

# 活动记录的栈式分配

- 寄存器SP指向栈顶
- 第一个过程 (main) 初始化栈区
- 过程调用指令序列
  - `ADD SP, SP, #caller.recordSize` // 增大栈指针
  - `ST 0(SP), #here + 16` // 保存返回地址
  - `BR callee.codeArea` // 转移到被调用者
- 返回指令序列
  - `BR *0(SP)` // 被调用者执行，返回调用者
  - `SUB SP, SP, #caller.recordSize` // 调用者减小栈指针

# 例子

```
100: LD SP, #600           // m的代码
108: ACTION1              // 初始化栈
128: ADD SP, SP, #msize    // action1的代码
136: ST *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: ACTION12             // 恢复SP的值
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    // 包含有跳转到456的条件转移指令
328: ST *SP, #344          // 将返回地址压入栈
336: BR 200                // 调用p
344: SUB SP, SP, #qsize
352: ACTION5
```

m调用q, q调用p

```
372: ADD SP, SP, #qsize
380: BR *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 *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
```

图 8-6 栈式分配时的目标代码

# 名字的运行时刻地址

- 在三地址语句中使用名字 (实际上是指向符号表条目) 来引用变量
- 语句  $x = 0$ 
  - 如果  $x$  分配在静态区域, 且静态区开始位置为 *static*
    - `static[12] = 0`      `LD 112, #0 // static = 100`
  - 如果  $x$  分配在栈区, 且相对地址为 12, 则
    - `LD 12(SP), #0`



# 基本块和流图

- 中间代码的**流图 (flow graph)** 表示法
  - 中间代码划分成为**基本块 (basic block)**
    - 控制流只能从基本块的**第一条**指令进入
    - 除基本块的最后一条指令外，控制流不会跳转/停机
  - 流图的**结点**是基本块，流图的**边**指明了哪些基本块可以跟在一个基本块之后运行
- 流图可以作为优化的基础
  - 它指出了基本块之间的**控制流**
  - 可以根据流图了解到一个值是否会被使用等信息

# 划分基本块的算法

- 输入：三地址指令序列
- 输出：基本块的列表
- 方法
  - 确定**首指令leader** (基本块的第一个指令)
    - 第一个三地址指令
    - 任意一个 (条件或无条件) 转移指令的**目标**指令
    - 紧跟在一个 (条件或无条件) 转移指令**之后**的指令
  - 确定基本块
    - 每个首指令对应于一个基本块：从首指令开始到下一个首指令

# 例子

- 第一个指令
  - 1
- 跳转指令的目标指令
  - 3, 2, 13
- 跳转指令的下一条指令
  - 10, 12
- 基本块
  - 1-1, 2-2, 3-9, 10-11, 12-12, 13-17

```
1)  i = 1
2)  j = 1
3)  t1 = 10 * i
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

# 后续使用信息

- 变量值的使用 (use)

- 三地址语句 $i$ 向变量 $x$ 赋值，如果另一个语句 $j$ 的运算分量为 $x$ ，且从 $i$ 开始有一条路径到达 $j$ ，且路径上没有对 $x$ 赋值，那么 $j$ 就使用了 $i$ 处计算得到的 $x$ 的值
- 我们说变量 $x$ 在语句 $i$ 后的程序点上活跃 (live)
  - 程序执行完语句 $i$ 时， $x$ 中存放的值将被后面的语句使用
  - 不活跃是指变量的值不会被使用，而不是变量不会被使用

- 这些信息可以用于代码生成

- 如果 $x$ 在 $i$ 处不活跃，且 $x$ 占用了一个寄存器，我们可以把这个寄存器用于其它目的

# 确定基本块中的活跃性、后续使用

- 输入
  - 基本块 $B$ ，开始时 $B$ 中的所有非临时变量都是活跃的
- 输出
  - 各个语句 $i$ 上变量的活跃性、后续使用信息
- 方法
  - 从 $B$ 的最后一个语句开始反向扫描
  - 对于每个语句 $i$ :  $x = y + z$ 
    - 令语句 $i$ 和 $x$ 、 $y$ 、 $z$ 的当前活跃性信息/使用信息关联
    - 设置 $x$ 为“不活跃”和“无后续使用”
    - 设置 $y$ 和 $z$ 为“活跃”，并指明它们的下一次使用设置为语句 $i$

# 例子

- $i, j, a$  非临时变量 (出口处活跃), 其余变量不活跃
  - 8)  $i, j, a$  活跃,  $j$  在 8 上被使用
  - 7)  $i, j, a, t4$  活跃,  $a$  和  $t4$  被 7 使用
  - 6)  $i, j, a, t3$  活跃,  $t4$  不活跃,  $t3$  被 6 使用
  - 5)  $i, j, a, t2$  活跃,  $t4, t3$  不活跃,  $t2$  被 5 使用
  - 4)  $i, j, a, t1$  活跃,  $t4, t3, t2$  不活跃,  $t1$  和  $j$  被 4 使用
  - 3)  $i, j, a$  活跃,  $t4, t3, t2, t1$  不活跃,  $i$  被 3 使用

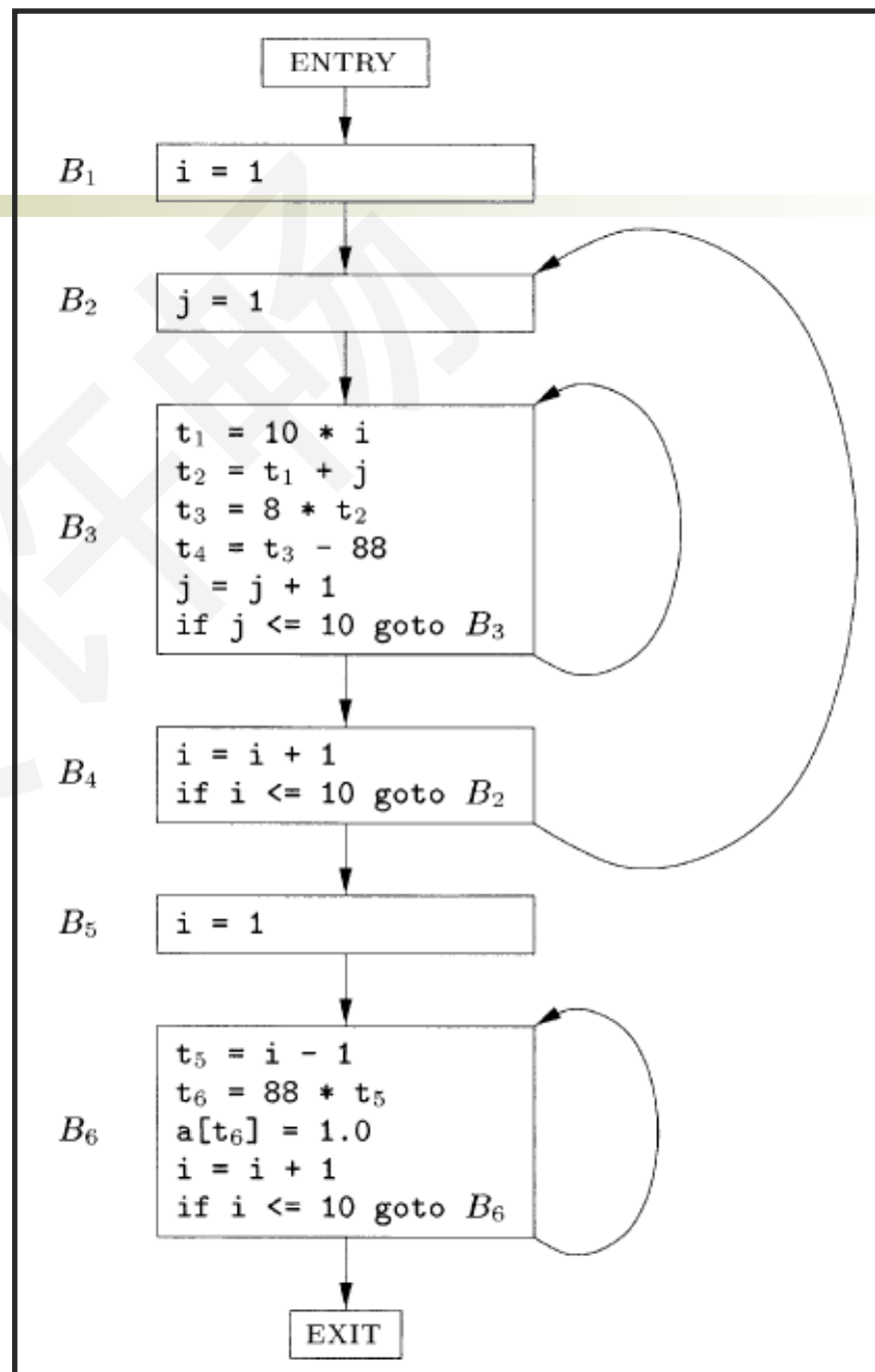
3)	$t1 = 10 * i$
4)	$t2 = t1 + j$
5)	$t3 = 8 * t2$
6)	$t4 = t3 - 88$
7)	$a[t4] = 0.0$
8)	$j = j + 1$

# 流图的构造

- 流图的**结点**是基本块
  - 两个结点 $B$ 和 $C$ 之间有一条有向边 iff 基本块 $C$ 的第一个指令**可能**在 $B$ 的最后一个指令之后执行
- 存在边的原因
  - $B$ 的结尾指令是一条跳转到 $C$ 的开头的条件/无条件语句
  - $C$ 紧跟在 $B$ 之后，且 $B$ 的结尾不是无条件跳转语句
  - 称 $B$ 是 $C$ 的**前驱** (predecessor),  $C$ 是 $B$ 的**后继** (successor)
- **入口 (entry) / 出口 (exit) 结点**
  - 不和任何中间指令对应；入口到第一条指令有一条边；**任何**可能最后执行的基本块到出口有一条边

# 流图的例子

- 因跳转而生成的边
  - $B_3 \rightarrow B_3$
  - $B_4 \rightarrow B_2$
  - $B_6 \rightarrow B_6$
- 因为顺序而生成的边
  - 其它



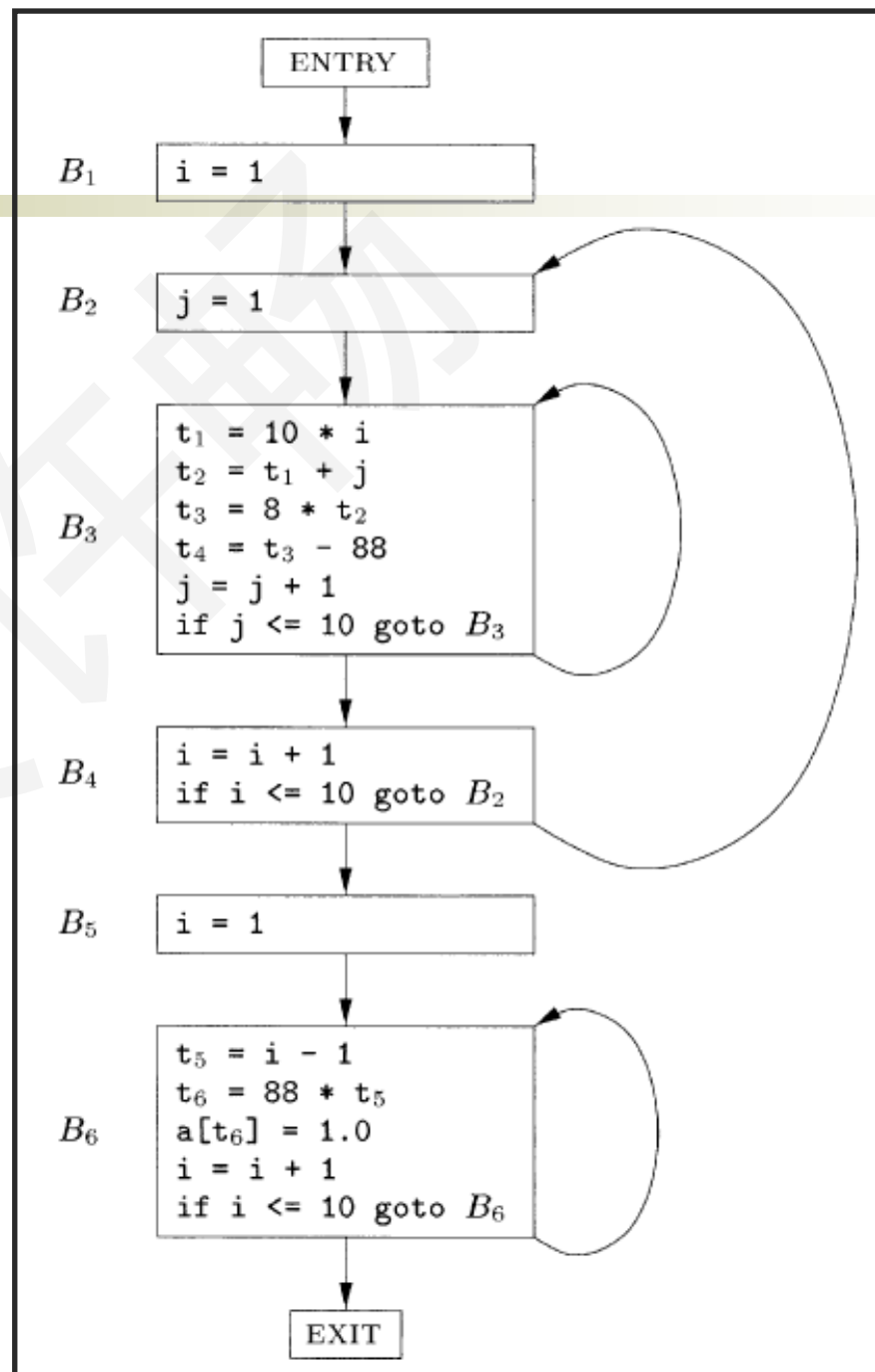


# 循环

- 程序的大部分运行时间花费在循环上，因此循环是识别的重点（优化的目标）
- 循环的定义
  - 循环 $L$ 是一个结点集合
  - 存在一个**循环入口 (loop entry)** 结点，是**唯一**的前驱可以在循环 $L$ 之外的结点，到达其余结点的路径必然先经过这个入口结点
  - 其余结点都存在到达入口结点的非空路径，且**路径都在 $L$ 中**

# 循环的例子

- 循环
  - $\{ B_3 \}$
  - $\{ B_2, B_3, B_4 \}$
  - $\{ B_6 \}$
- 对于 $\{ B_2, B_3, B_4 \}$ 的解释
  - $B_2$ 为入口结点
  - $B_1, B_5, B_6$ 不在循环内
    - 到达 $B_1$ 可不经过 $B_2$
    - $B_5, B_6$ 没有到达 $B_2$ 的路径



# 基本块的优化

- 针对基本块的优化可以有很好的效果 (局部优化)
- DAG图可反映变量及其值对其他变量的依赖关系
- 构造方法
  - 每个变量都有一个对应的DAG结点表示其初始值
  - 每个语句 $s$ 有一个相关的结点 $N$ ，代表此计算得到的值
    - $N$ 的子结点对应于 (得到其运算分量当前值的) 其它语句
    - $N$ 的标号是 $s$ 中的运算符，同时还有一组变量被关联到 $N$ ，表示 $s$ 是最晚对这些变量进行定值的语句

# DAG图的构造

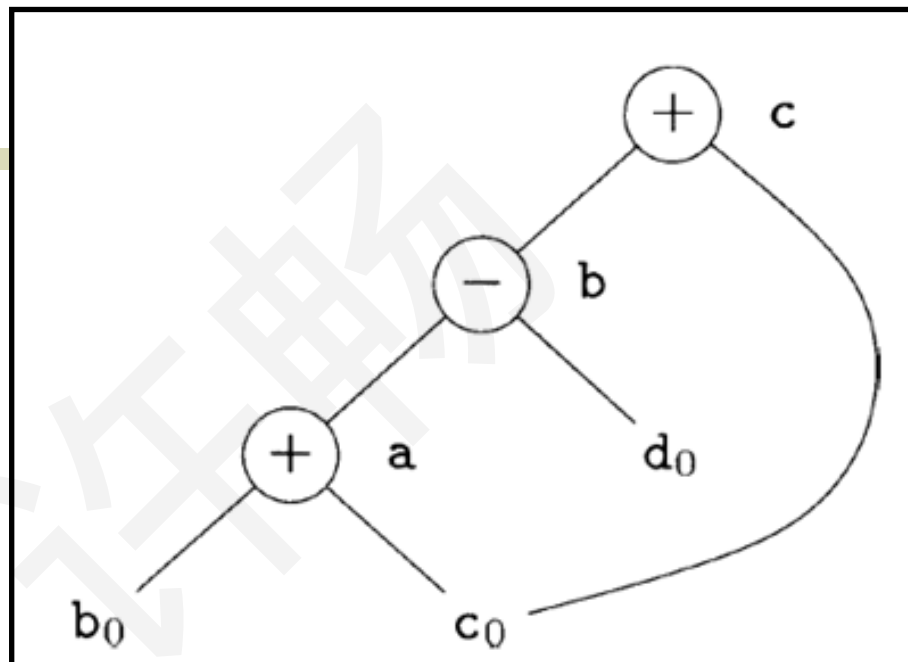
- 为基本块中出现的每个变量建立结点 (表示初始值), 各变量和相应结点关联
- 顺序扫描各三地址指令, 进行如下处理
  - 指令  $x = y \text{ op } z$ 
    - 为该指令建立结点  $N$ , 标号为  $op$ , 令  $x$  和  $N$  关联
    - $N$  的子结点为  $y$  和  $z$  当前关联的结点
  - 指令  $x = y$ 
    - 假设  $y$  关联到  $N$ , 那么  $x$  现在也关联到  $N$
- 扫描结束后, 对所有在出口处活跃的变量  $x$ , 将  $x$  所关联的结点设置为输出结点 (output node)

# 例子

- 指令序列
  - $a = b + c$
  - $b = a - d$
  - $c = b + c$

- 过程

- 结点 $b_0$ 、 $c_0$ 和 $d_0$ 对应于 $b$ 、 $c$ 和 $d$ 的初始值
- $a = b + c$ : 构造第一个加法结点,  $a$ 与之关联
- $b = a - d$ : 构造减法结点,  $b$ 与之关联
- $c = b + c$ : 构造第二个加法结点,  $c$ 与之关联 (注意第一个子结点对应于减法结点)



# DAG的作用

- DAG图描述了基本块运行时各变量的值 (和初始值) 之间的关系
- 以DAG为基础，对代码进行转换
  - 寻找局部公共子表达式 (local common subexpression)
  - 消除死代码 (dead code)
  - 代数恒等式的使用
  - 数组引用的表示
  - 指针赋值和过程调用

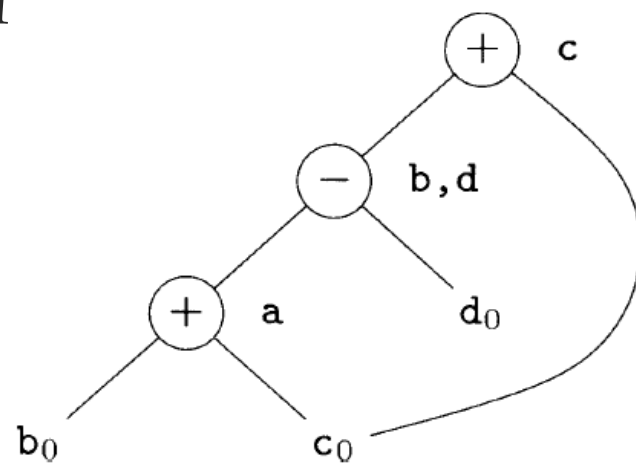
# 局部公共子表达式

- 局部公共子表达式的发现
  - 建立某个结点 $M$ 之前，检查是否存在一个结点 $N$ ，它和 $M$ 具有相同的运算符和子结点 (顺序也相同)
  - 如果存在，则不需要生成新的结点，用 $N$ 代表 $M$

- 例如

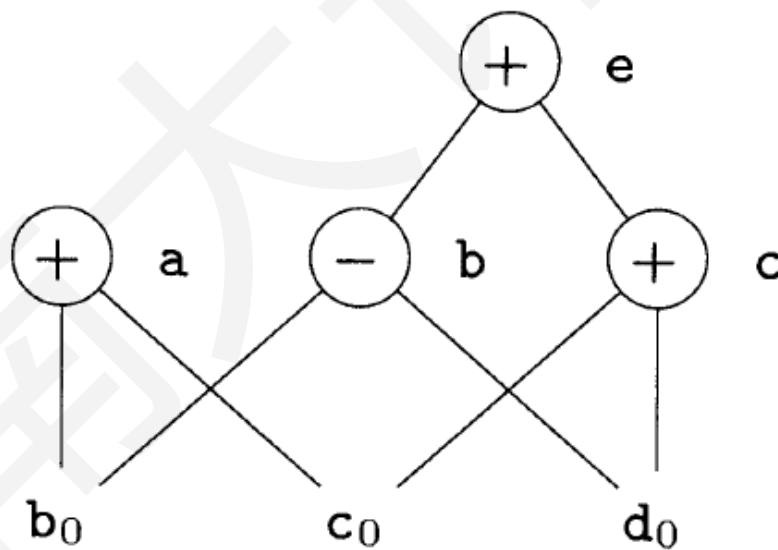
- $a = b + c$
- $b = a - d$
- $c = b + c$
- $d = a - d$

- 注意：两个 $b + c$ 不是公共子表达式 (但 $a - d$ 是)



# 消除死代码

- 在DAG图上消除没有附加活跃变量的根结点，即消除死代码
- 如果图中c、e不是活跃变量(但a、b是)，则可以删除标号为e、c的结点





# 基于代数恒等式的优化

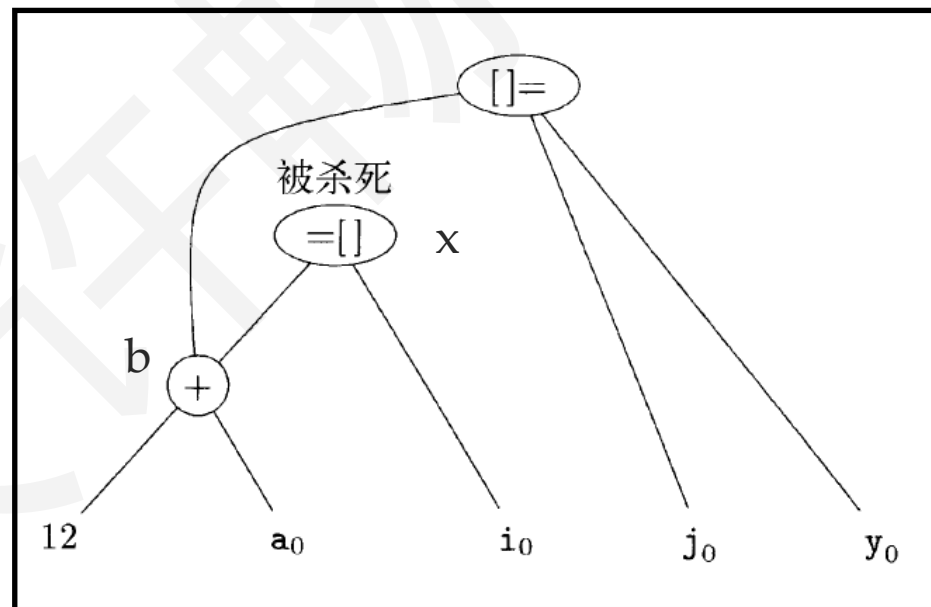
- 恒等式
  - $x + 0 = 0 + x = x$ ,  $x - 0 = x$ ,  $x * 1 = 1 * x = x$ ,  $x / 1 = x$
- 强度消减 (reduction in strength)
  - $x^2 = x * x$ ,  $2 * x = x + x$ ,  $x / 2 = x * 0.5$
- 常量合并/折叠 (constant folding)
  - $2 * 3.14$  可以用  $6.28$  替换
- 通用代数转换规则
  - 交换律和结合律等, 如  $x * y = y * x$
- 实现这些优化, 只需在DAG图上寻找特定的模式

# 数组引用

- $a[j]$ 可能改变 $a[i]$ 的值，因此不能像普通运算符一样构造结点
  - $x = a[i]$        $a[j] = y$        $z = a[i]$
- 从数组取值的运算 $x = a[i]$ 对应于 $=[]$ 的结点
  - 这个结点的左右子节点是数组初始值 $a_0$ 和下标 $i$
  - 变量 $x$ 是这个结点的标号之一
- 对数组赋值的运算 $a[j] = y$ 对应于 $[] =$ 的结点
  - 这个结点的三个子节点分别表示 $a_0$ 、 $j$ 和 $y$
  - 杀死所有依赖于 $a_0$ 的变量

# 数组引用DAG的例子

- 设 $a$ 是数组， $b$ 是指针
  - $b = 12 + a$
  - $x = b[i]$
  - $b[j] = y$



- 一个结点被杀死，意味着它不能被复用
  - 考虑再有指令 $m = b[i]$

# 指针赋值/过程调用

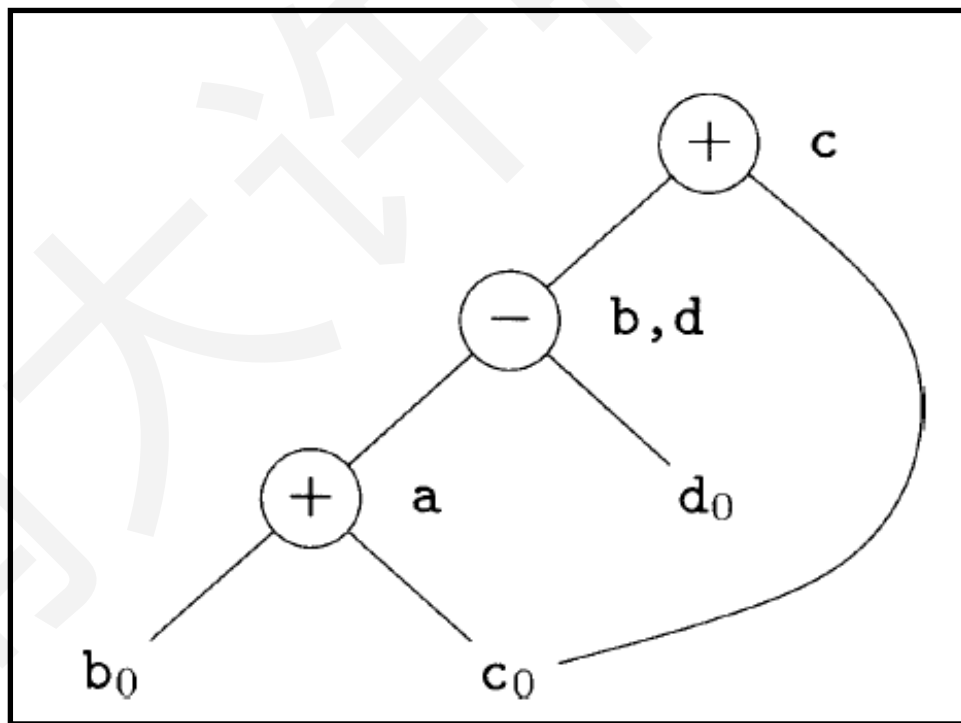
- 通过指针进行取值/赋值： $x = *p$ 、 $*q = y$ 
  - $x$ 使用了任意变量，因此无法消除死代码
  - $*q = y$ 对任意变量赋值，因此杀死了全部其它结点
- 可通过 (全局/局部) 指针分析部分地解决这个问题
- 过程调用也类似，必须安全地假设它
  - 使用了可访问范围内的所有变量
  - 修改了可访问范围内的所有变量

# 从DAG到基本块的重组

- 重组的方法
  - 每个结点构造一个三地址语句，计算对应的值
  - 结果应该尽量赋给一个活跃的变量
  - 如果结点有多个关联的变量，则需要用复制语句进行赋值

# 例子

- 根据DAG构造时结点产生的顺序
  - $a = b + c$
  - $d = a - d$
  - $b = d$
  - $c = \textcolor{red}{d} + c$



# 重组的规则

- 注意求值顺序
  - 指令顺序必须遵守DAG中结点的顺序
  - 对数组赋值 (**write**) 要跟在原来之前的赋值/求值之后
  - 对数组求值 (**read**) 要跟在原来之前的赋值指令之后
  - 对变量的使用必须跟在所有原来在它之前的过程调用和指针间接赋值之后
  - 任何过程调用或指针间接赋值必须跟在原来在它之前的变量求值之后
- 即保证
  - 如果两个指令之间相互**影响**，它们的顺序就不该改变

# 代码生成器

- 根据三地址指令序列生成机器指令
  - 假设每个三地址指令只有一个对应的机器指令
  - 有一组寄存器用于计算基本块内部的值
- 主要的目标是减少加载 (LD) 和保存 (ST) 指令，即最大限度地利用寄存器
- 寄存器的使用方法
  - 执行运算时，运算分量必须放在寄存器中
  - 存放临时变量
  - 存放全局的值
  - 进行运行时刻管理 (比如栈顶指针)



# 算法的基本思想和数据结构

- 依次考虑各三地址指令，尽可能把值保留在寄存器中，以减少寄存器/内存之间的数据交换
- 为一个三地址指令生成机器指令时
  - 只有当运算分量不在寄存器中时，才从内存载入
  - 尽量保证只有当寄存器中值不被使用时，才覆盖掉
- 数据结构
  - **寄存器描述符 (register descriptor)**: 跟踪各个寄存器都存放了哪些变量的当前值
  - **地址描述符 (address descriptor)**: 各个变量的当前值存放在哪些位置 (包括内存位置和寄存器) 上

# 代码生成算法 (1)

- 重要子函数: *getReg(I)*
  - 根据寄存器描述符和地址描述符等数据流信息, 为三地址指令*I*选择最佳的寄存器
  - 得到的机器指令的质量依赖于*getReg*函数选取寄存器的算法
- 代码生成算法逐个处理三地址指令

# 代码生成算法 (2)

- 运算语句:  $x = y + z$ 
  - $getReg(x = y + z)$  为  $x, y, z$  选择寄存器  $R_x, R_y, R_z$
  - 检查  $R_y$  的寄存器描述符, 如果  $y$  不在  $R_y$  中则生成指令
    - $LD R_y, y'$  //  $y'$  表示存放  $y$  值的当前位置
    - 类似地确定是否生成  $LD R_z, z'$
  - 生成指令  $ADD R_x, R_y, R_z$
- 复制语句:  $x = y$ 
  - $getReg(x = y)$  为  $x$  和  $y$  选择相同的寄存器
  - 如果  $y$  不在  $R_y$  中, 则生成指令  $LD R_y, y$
- 基本块的收尾
  - 如果变量  $x$  活跃, 且不在内存中, 则生成指令  $ST x, R_x$

# 代码生成算法 (3)

- 代码生成同时更新寄存器和地址描述符
- 处理指令时生成的LD  $R, x$ 
  - $R$ 的寄存器描述符：只包含 $x$
  - $x$ 的地址描述符： $R$ 作为新位置加入到 $x$ 的位置集合中
  - 从任何不同于 $x$ 的变量的地址描述符中删除 $R$
- 生成的ST  $x, R$ 
  - $x$ 的地址描述符：包含自己的内存位置(新增)

# 代码生成算法 (4)

- $\text{ADD } R_x, R_y, R_z$ 
  - $R_x$ 的寄存器描述符: 只包含 $x$
  - $x$ 的地址描述符: 只包含 $R_x$  (不包含 $x$ 的内存位置)
  - 从任何不同于 $x$ 的变量的地址描述符中删除 $R_x$
- 处理 $x = y$ 时
  - 如果生成 $\text{LD } R_y, y$ , 按照第一个规则处理
  - 把 $x$ 加入到 $R_y$ 的寄存器描述符中 (即 $R_y$ 同时存放了 $x$ 和 $y$ 的当前值)
  - $x$ 的地址描述符: 只包含 $R_y$  (不包含 $x$ 的内存位置)

# 例子 (1)

- a、b、c、d在出口处活跃
- **t**、**u**、**v**是局部临时变量

$$\mathbf{t} = a - b$$

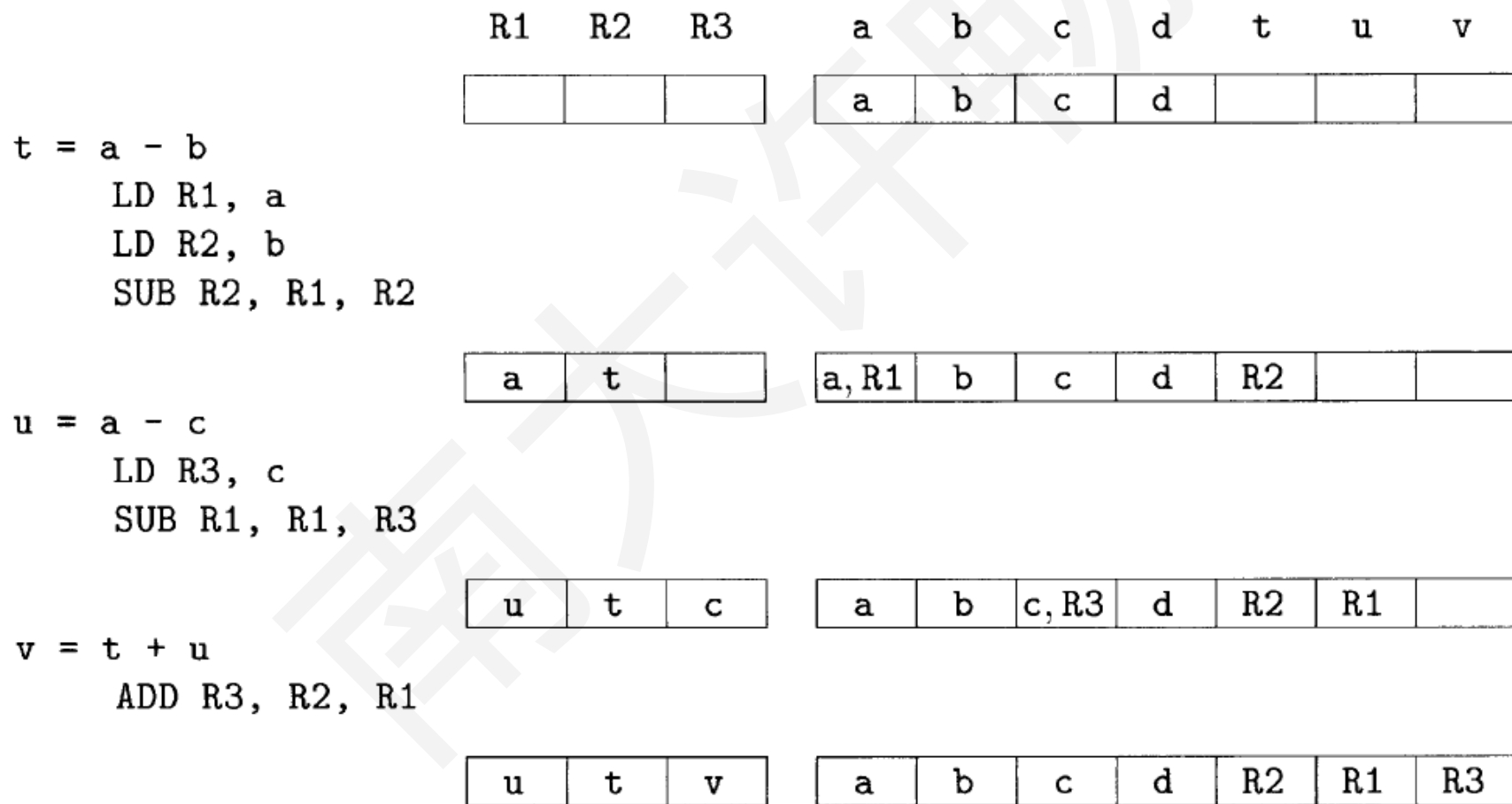
$$\mathbf{u} = a - c$$

$$\mathbf{v} = \mathbf{t} + \mathbf{u}$$

$$a = d$$

$$d = \mathbf{v} + \mathbf{u}$$

## 例子 (2)



# 例子 (3)

a = d

LD R2, d

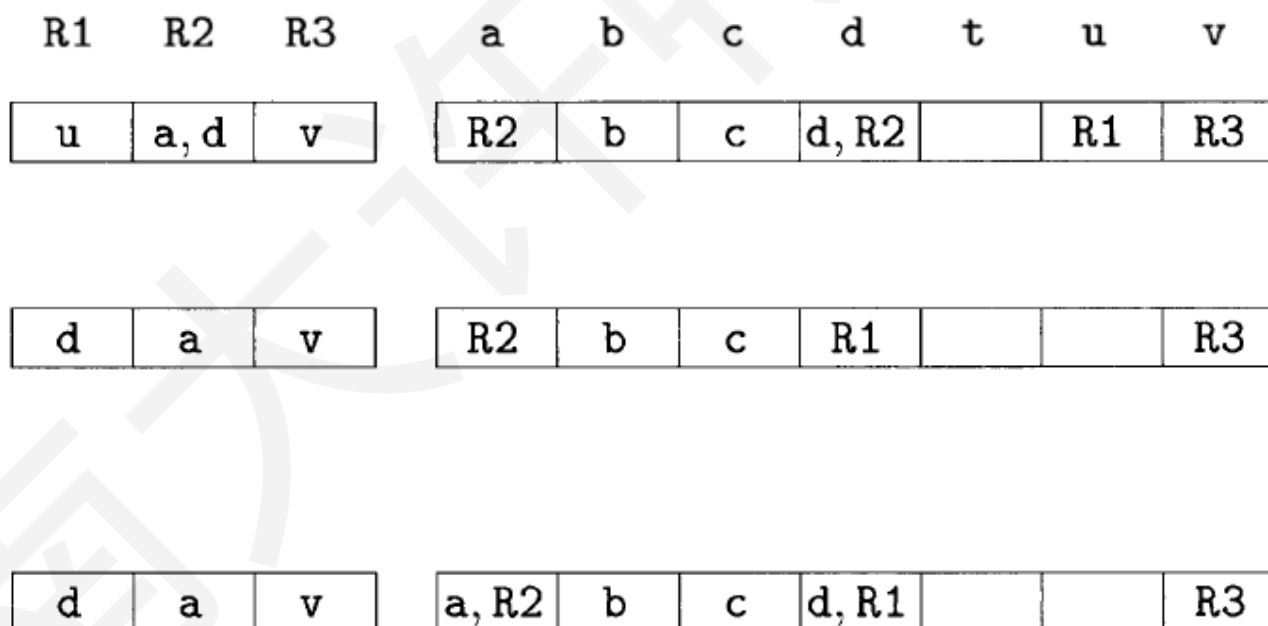
d = v + u

ADD R1, R3, R1

exit

ST a, R2

ST d, R1





# *getReg*函数 (1)

- 目标：减少LD/ST指令
- 任务：为运算分量和结果分配寄存器
- 为  $x = y \text{ op } z$  的运算分量  $y$  和  $z$  分配寄存器
  - 如果  $y$  已经在某个寄存器中，不需要进行处理，选择这个寄存器作为  $R_y$
  - 如果  $y$  不在寄存器中，且有空闲寄存器，选择一个空闲寄存器作为  $R_y$
  - 如果不在寄存器中，且没有空闲寄存器？

## *getReg*函数 (2)

- 如果不在寄存器中，且没有空闲寄存器？
- 寻找一个寄存器 $R$ ，其寄存器描述符表示 $v$ 在 $R$ 中
  - 如果 $v$ 的地址描述符表明还可以在别的地方找到 $v$ ，DONE
  - $v$ 就是 $x$  (即 **结果**)，且 $x$ 不是运算分量 $z$ ，DONE
  - 如果 $v$ 在此之后不会被使用 (**不活跃**)，DONE
  - **溢出操作 (spill)**: 生成保存指令 $ST\ v, R$ 并修改 $v$ 的地址描述符；如果 $R$ 中存放了多个变量的值，那么需要生成多条 $ST$ 指令

## *getReg*函数 (3)

- 为  $x = y \text{ op } z$  的结果  $x$  选择寄存器  $R_x$  的方法基本上和上面要把  $y$  从内存 LD 时一样，但是
  - 只存放  $x$  值的寄存器总是可接受的
  - 如果  $y$  在指令之后不再使用，且  $R_y$  仅仅保存了  $y$  的值，那么  $R_y$  同时也可以作为  $R_x$  (对  $z$  也一样)
- 处理  $x = y$  时
  - 先选择  $R_y$
  - 然后让  $R_x = R_y$

# 窥孔优化

- 窥孔优化 (peephole optimization): 使用一个滑动窗口 (窥孔) 来检查目标指令, 在窥孔内实现优化
  - 冗余指令消除
  - 控制流优化
  - 代数化简
  - 机器特有指令的使用

# 冗余指令消除

- 多余的LD/ST指令
  - LD  $R_0, a$
  - ST  $a, R_0$       // 无标号; 可删除
- 级联跳转指令
  - if debug == 1 goto L1; goto L2; L1: ...; L2: ...;
    - => if debug != 1 goto L2; L1: ...; L2: ...;
  - 如果已知debug一定是0, 那么替换成为goto L2

# 控制流优化

- 不必要跳转指令
  - goto L1; ... L1: goto L2
    - $\Rightarrow$  goto **L2**; ... L1: goto L2
  - if a < b goto L1; ... L1: goto L2
    - $\Rightarrow$  if a < b goto **L2**; ... L1: goto L2

# 代数化简和机器特有指令

- 应用代数恒等式
  - 消除  $x = x + 0$ ,  $x = x * 1$ , ...
  - 用  $x * x$  替换  $x^2$
- 使用机器特有指令
  - INC, DEC, ...

# 寄存器分配和指派

- 寄存器分配
  - 确定在程序的每个点上，**哪个值**应该存放在寄存器中
- 寄存器指派
  - 各个值应该存放在**哪个寄存器**中
- 简单方法：把特定类型的值分配给特定的寄存器
  - 数组基地址指派给一组寄存器，算术计算分配给一组寄存器，栈顶指针分配一个寄存器，循环，...
  - 缺点：寄存器的使用效率较低



# 全局寄存器分配

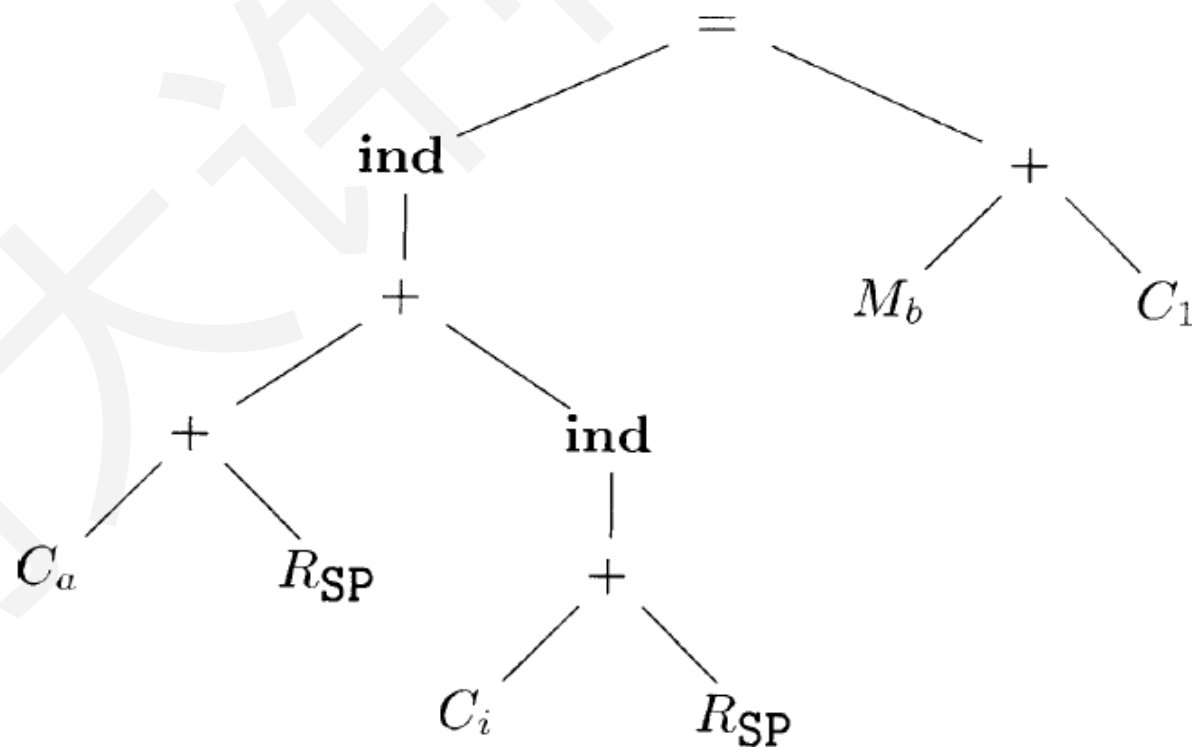
- 在循环中频繁使用的值存放在固定寄存器
  - 分配固定多个寄存器来存放内部循环中最活跃的值
- 可以通过使用计数 (保存和加载成本) 的方法来估算把一个变量放到寄存器中会带来多大好处，然后根据这个估算来分配寄存器

# 树重写实现指令选择

- 在某些机器上，同一个三地址指令可以使用多种机器指令实现，有时多个三地址指令可以使用一个机器指令实现
- 指令选择
  - 为实现中间表示形式中出现的运算符选择适当的机器指令
  - 用树来表示中间代码，按照特定的规则不断覆盖这棵树并生成机器指令

# 例子

- $a[i] = b + 1$ 
  - ind: 把参数作为内存地址
  - $a, i$ : 局部变量
  - $b$ : 全局变量
  - SP: 栈顶指针



# 目标指令选择

- 通过应用一个树重写规则序列来生成

$$replacement \leftarrow template \{ action \}$$

其中,  $replacement$  (替换结点) 是一个结点,  $template$  (模板) 是一棵树,  $action$  (动作) 是一个像语法制导翻译方案中那样的代码片断。

- 一组树重写规则被称为一个树翻译方案 (tree-translation scheme)
- 树重写规则示例

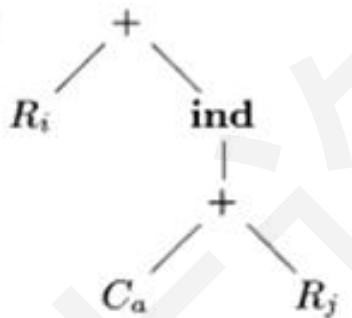


$$R_i \leftarrow \begin{array}{c} + \\ / \quad \backslash \\ R_i \quad R_j \end{array} \quad \{ \text{ADD } R_i, R_i, R_j \}$$

# 一些重写规则 (1)

1)	$R_i \leftarrow C_a$	$\{ \text{LD } R_i, \#a \}$
2)	$R_i \leftarrow M_x$	$\{ \text{LD } R_i, x \}$
3)	$M \leftarrow \begin{array}{c} = \\ / \quad \backslash \\ M_x \quad R_i \end{array}$	$\{ \text{ST } x, R_i \}$
4)	$M \leftarrow \begin{array}{c} = \\ / \quad \backslash \\ \text{ind} \quad R_j \\   \\ R_i \end{array}$	$\{ \text{ST } *R_i, R_j \}$
5)	$R_i \leftarrow \begin{array}{c} \text{ind} \\   \\ + \\ / \quad \backslash \\ C_a \quad R_j \end{array}$	$\{ \text{LD } R_i, a(R_j) \}$

一些目标机器指令的树重写规则

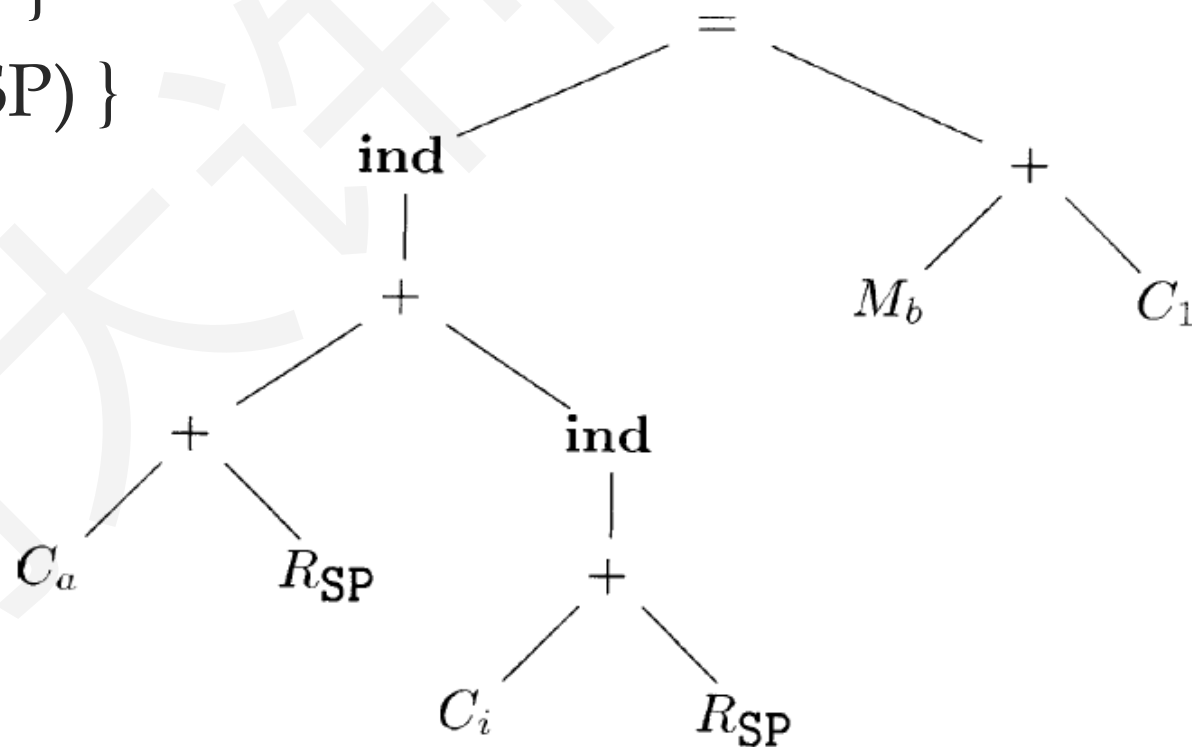
## 一些重写规则 (2)

6)	$R_i \leftarrow$ 	$\{ \text{ADD } R_i, R_i, a(R_j) \}$
7)	$R_i \leftarrow$ 	$\{ \text{ADD } R_i, R_i, R_j \}$
8)	$R_i \leftarrow$ 	$\{ \text{INC } R_i \}$

一些目标机器指令的树重写规则

# 覆盖重写过程

- 规则1):  $\{ \text{LD } R_0, \#a \}$
- 规则7):  $\{ \text{ADD } R_0, R_0, \text{SP} \}$
- 规则6):  $\{ \text{ADD } R_0, R_0, i(\text{SP}) \}$
- 规则2):  $\{ \text{LD } R_1, b \}$
- 规则8):  $\{ \text{INC } R_1 \}$
- 规则4):  $\{ \text{ST } *R_0, R_1 \}$



# 树翻译方案的工作模式

- 给定一颗输入树，树重写规则中的模板被用来匹配输入树的子树
- 如果找到一个匹配的模板，那么输入树中匹配的子树将被替换为相应规则中的替换结点，并执行相应的动作，这可能是生成相应的机器指令序列
- 不断匹配，直到这颗树被规约成单个结点，或找不到匹配的模板为止
- 在此过程中生成的机器指令代码序列就是树翻译方案作用于给定输入树而得到的输出



# 通过扫描进行模式匹配

- 如何完成树匹配？
  - 把树重写规则替换成相应的上下文无关文法的产生式
  - 产生式的右部是其指令模板的**前缀表示**
- 如果在某个时刻有多个模板可以匹配
  - 匹配到**大树**优先

1)	$R_i \rightarrow c_a$	$\{ LD \ R_i, \#a \}$
2)	$R_i \rightarrow M_x$	$\{ LD \ R_i, x \}$
3)	$M \rightarrow = M_x R_i$	$\{ ST \ x, R_i \}$
4)	$M \rightarrow = \mathbf{ind} \ R_i \ R_j$	$\{ ST \ *R_i, R_j \}$
5)	$R_i \rightarrow \mathbf{ind} + c_a R_j$	$\{ LD \ R_i, a(R_j) \}$
6)	$R_i \rightarrow + R_i \mathbf{ind} + c_a R_j$	$\{ ADD \ R_i, R_i, a(R_j) \}$
7)	$R_i \rightarrow + R_i R_j$	$\{ ADD \ R_i, R_i, R_j \}$
8)	$R_i \rightarrow + R_i c_1$	$\{ INC \ R_i \}$
9)	$R \rightarrow \mathbf{sp}$	
10)	$M \rightarrow \mathbf{m}$	

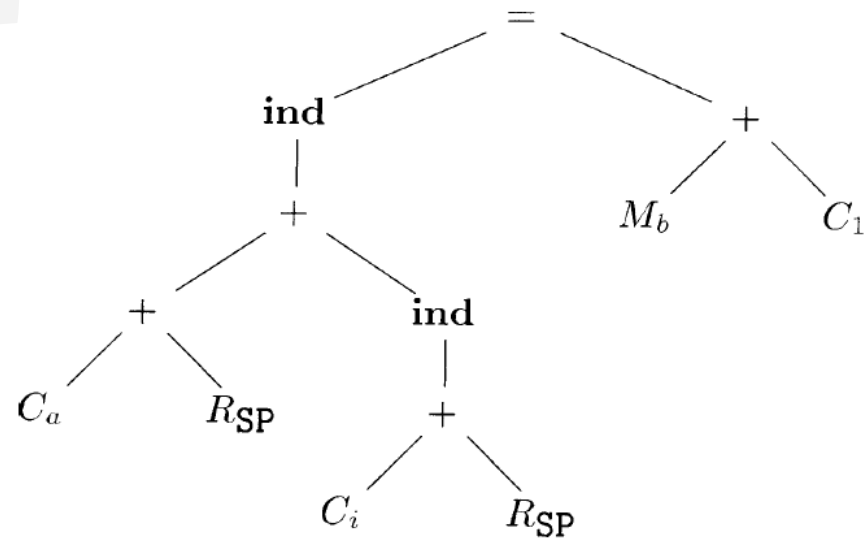


图 8-21 由图 8-20 构造得到的语法制导翻译方案