第八章 代码生成

陈 林

代码生成器的位置

图 8-1　代码生成器的位置

- 根据中间表示生成代码
- 代码生成器之前可能有一个优化组件
- 代码生成器的三个主要任务
  - 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

主要内容

- 代码生成器设计中的问题
- 目标语言
- 目标代码中的地址
- 基本块和流图
- 基本块优化方法
- 寄存器分配和指派

代码生成器中的问题

- 正确性、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、…
- 目标程序
  - RISC、CISC、可重定向代码、汇编语言
- 指令选择
  - 影响因素：IR层次、指令集特性、目标代码质量
- 寄存器分配和指派
- 求值顺序

指令选择

```
x = y + z;

LD  R0, y        // R0 = y          (load y into register R0)
ADD R0, R0, z    // R0 = R0 + z     (add z to R0)
ST  x, R0        // x = R0          (store R0 into x)
```

```
a = b + c
d = a + e
```

如果变量a不再被使用了呢？

```
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
```

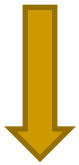
冗余的指令

指令选择

得考虑
指令代价

```
a = a + 1
```

```
LD   R0, a          // R0 = a
ADD  R0, R0, #1     // R0 = R0 + 1
ST   a, R0          // a = R0
```



```
INC a
```

目标语言

- 三地址机器的模型
- 指令集
  - 加载：LD dst, addr
    - 把地址addr中的内容加载到dst所指寄存器
  - 保存：ST x, r
    - 把寄存器r中的内容保存到x中
  - 计算：OP dst, src1, src2
    - 把src1和scr2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r中的值
- constant(r)：寄存器中内容加上前面的常数即其地址
- *r：寄存器r的内容为其地址
- *constant(r)：r中内容加上常量所指地址中存放的值为其地址
- 常量#constant

实例

- 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
- 一些思考
  - 如果y已经装载进某个寄存器
  - R1里面的内容是什么
  - 如果z已经装载进某个寄存器
  - R2里面的内容是什么
  - 如果x再也没有被使用

实例

- b = a[i]
  - 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
- a[j] = c
  - LD R1, c //R1=c
  - LD R2, j //R2=j
  - MUL R2, R2, 8 //R2=R2*8
  - ST a(R2), R1 //contents(a+contents(R2))=R1

实例

```
x = *p
  LD   R1, p          // R1 = p
  LD   R2, 0(R1)      // R2 = contents(0 + contents(R1))
  ST   x, R2          // x = R2

*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、目标程序大小、运行时间、能耗
- 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
- 为给定源程序生成最优目标程序
  - 不可判定

目标代码中的地址

- 如何将IR中的名字转换成为目标代码中的地址
  - 不同的区域中的名字采用不同的寻址方式
- 如何为过程调用和返回生成代码
  - 静态分配
  - 栈式分配

活动记录的静态分配

- 活动记录的大小和布局由符号表决定
- 每个过程静态地分配一个数据区域
  - 开始位置用staticArea表示

- 过程调用时
  - 在活动记录中存放返回地址
- 过程调用结束后
  - 控制权返回

活动记录的静态分配

- 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

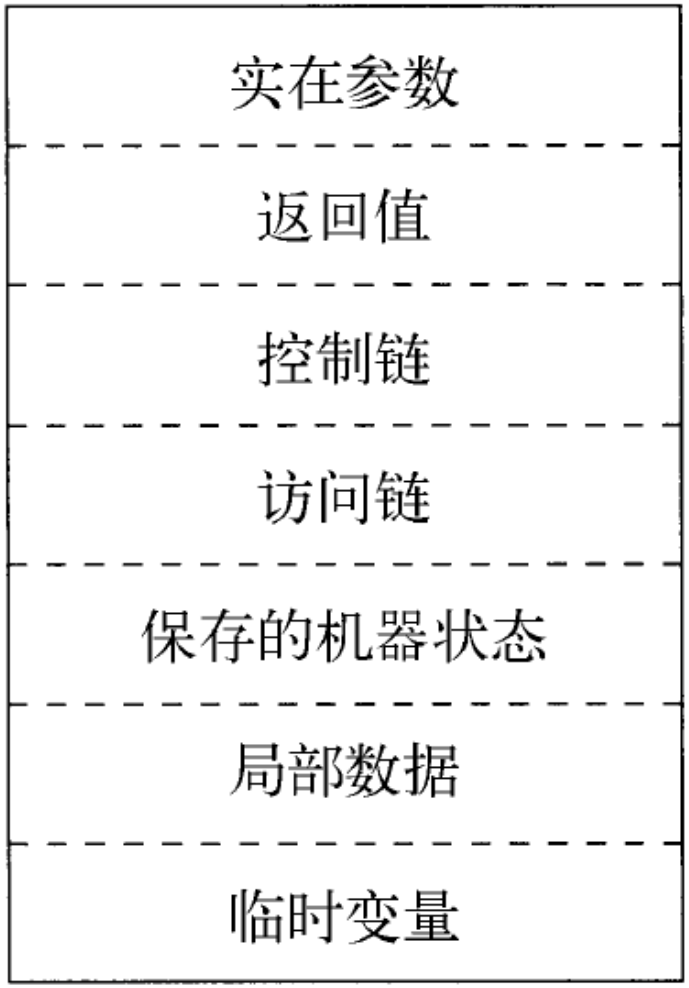
例子

- 三地址代码

```
//过程c的代码
action1
call p
action 2
halt

//过程p的代码
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
```



静态分配会
使得语言受
到什么限制？

活动记录栈式分配

- 问题
  - 运行时刻才能知道一个过程的活动记录的位置
- 解决
  - 活动记录的位置存放在寄存器中
  - 偏移地址

活动记录栈式分配

- 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　　//增加栈指针
  - ST　0(SP), #here+16　　//保存返回地址
  - BR callee.codeArea　　//转移到被调用者
- 过程返回指令序列
  - BR　　*0(SP)　//被调用者执行，返回调用者
  - SUP SP, SP, #caller.recordSize　　//调用者减少栈指针值

例子

```
                    // 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 *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:  ACTION12
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 *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:  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 （相对地址）
- 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　　ST　112　#0
  - 如果x分配在栈区，且相对地址为12，则
    - ST　12(SP)　　#0

基本块和流图

- 中间代码的流图表示法
  - 中间代码划分成为基本块(basic block)
    - 控制流只能从第一个指令进入
    - 除基本块的最后一个指令外，控制流不会跳转/停机
  - 结点：基本块
  - 边：指明了基本块的执行顺序
- 流图可以作为优化的基础
  - 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：
  - 确定基本块的首指令
    - 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/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

基本块划分的例子

- 第一个指令
  - 1
- 跳转指令的目标
  - 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)
```

练习

(1) b = 1
(2) n = 10
(3) d = l + n
(4) c = 2
(5) t1 = 4 * a
(6) t2 = l + n
(7) c = c + b
(8) if c < n goto (6)
(9) a = a + b
(10) if a < n goto (4)
(11) c = a – b

后续使用信息

- 可用于优化
  - 寄存器指派

如果当前某个变量存放于一个寄存器中，之后不会再被使用 ----> 寄存器可被分派给别的变量

```
a = b + c
d = a + e
```

```
LD  R0, b
ADD R1, R0, c
ST  a, R1

LD  R1, a
ADD R2, R1, e
ST  d,  R2
```

```
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
```

后续使用信息

- 变量值的使用（use）
  - 假设三地址语句i向x赋值，如果另一语句j的一个运算分量为x，且从i开始有一条没有对x进行赋值的路径到达j，那么j就使用了i处计算得到的x的值
  - 称x在语句i后的程序点上活跃（live）
    - 即在程序执行完语句i的时刻，x中存放的值将被后面的语句使用
    - 不活跃是指变量中存放的值不会被使用，而不是变量不会被使用

确定基本块中的活跃性、后续使用

- 输入：基本块B，开始时B的所有非临时变量都是活跃的
- 输出：各语句i上的变量的活跃性、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
    1. 令语句i和x、y、z的当前活跃性信息/使用信息关联
    2. 设置x为不活跃、无后续使用
    3. 设置y和z为活跃，并指明它们的下一次使用为语句i
  - 注意：2和3的顺序

例子

- 假设i, j, a不是临时变量，它们在出口处活跃，其余变量不活跃
  - 在8之前的程序点上，i, j, a仍然活跃；且j在8上被使用
  - 在7之前的程序点上，i, j, a, t4活跃；且t4被7使用
  - 在6之前的程序点上，i, j, a, t3活跃，t4不活跃，t3被6使用
  - 在5之前的程序点上，i, j, a, t2活跃，t3不活跃
  - 在4之前的程序点上，…

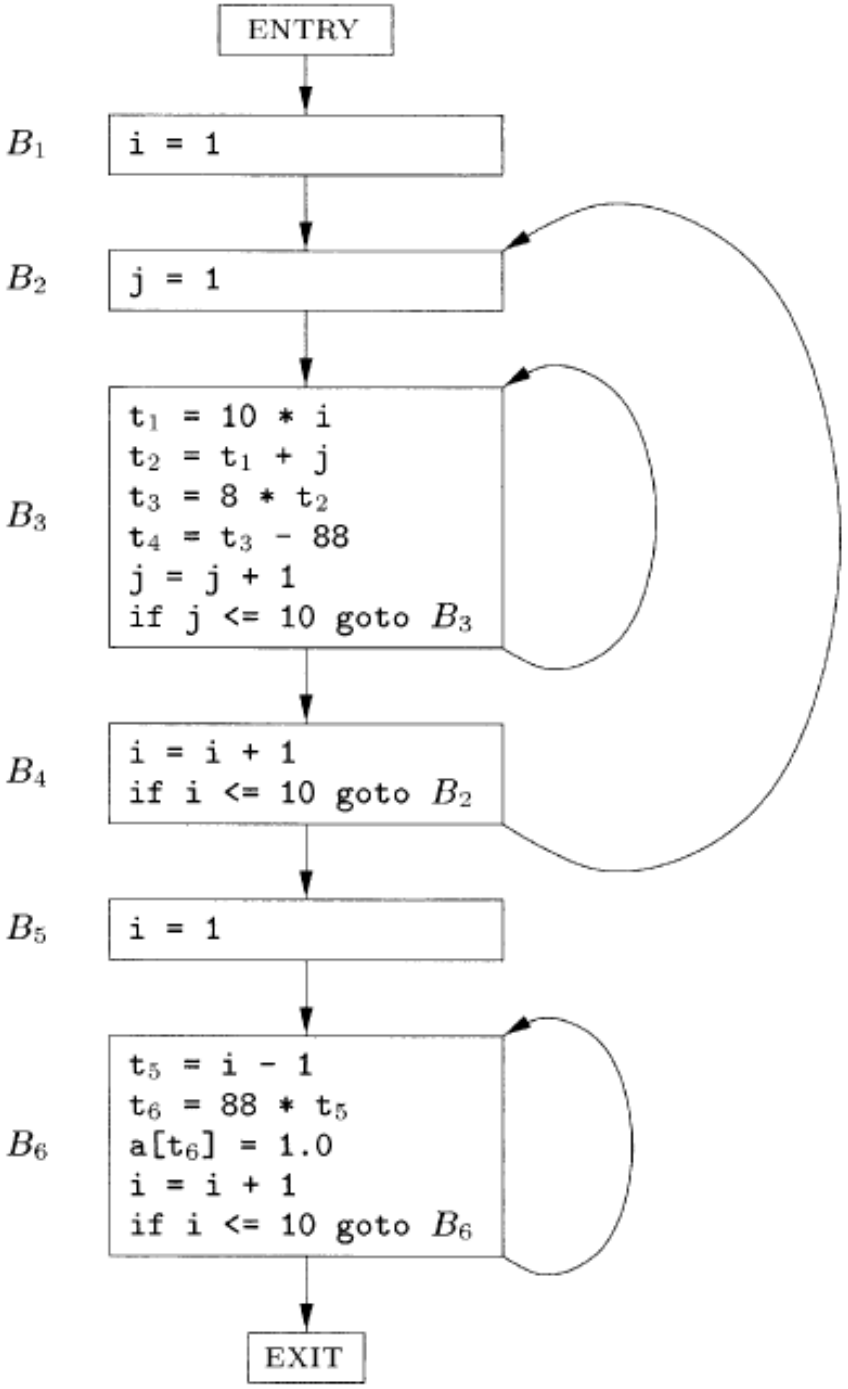
```
3)   t1 = 10 * i
4)   t2 = t1 + j
5)   t3 = 8 * t2
6)   t4 = t3 - 88
7)   a[t4] = 0.0
8)   j = j + 1
```

流图的构造

- 流图的顶点是基本块
- 两个顶点B和C之间有一条有向边当且仅当基本块C的第一个指令可能在B的最后一个指令之后执行
  - 从B的结尾指令是一条跳转到C的开头的条件/无条件语句
  - 在原来的序列中，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，C是B的后继
- 入口和出口结点
  - 流图中额外添加的边，不与中间代码（基本块）对应
  - 入口到第一条指令有一条边
  - 从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
  - B3→B3
  - B4→B2
  - B6→B6
- 因为顺序而生成的边
  - 其它

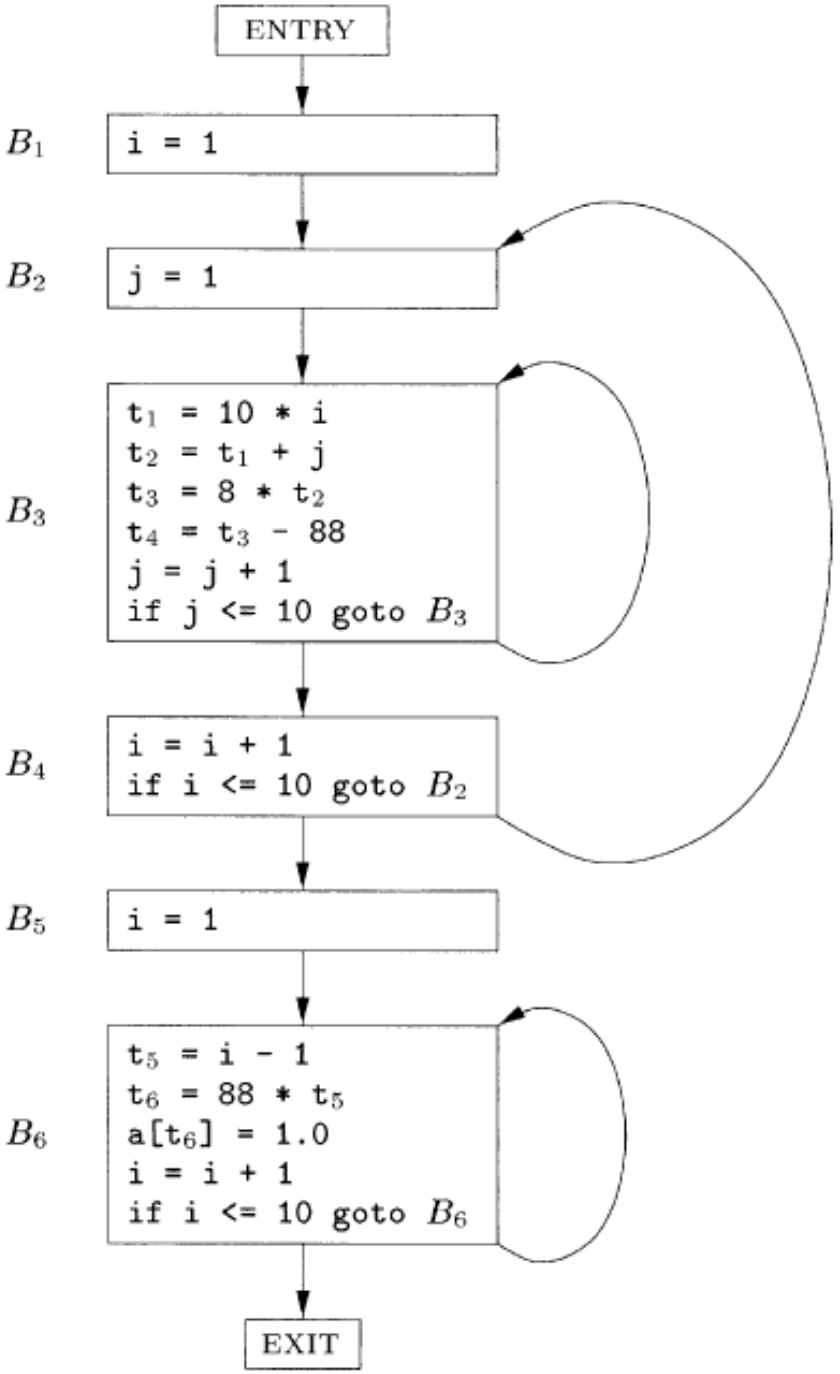


循环

- 程序的大部分运行时间花费在循环上
  - 循环是识别的重点
- 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
  - $\{B_3\}$
  - $\{B_6\}$
  - $\{B_2, B_3, B_4\}$
- 对于$\{B_2, B_3, B_4\}$的解释
  - B_2为入口结点
  - B_1，B_5，B_6不在循环内的理由
    - 到达B_1可不经过B_2
    - B_5，B_6没有到达B_2的结点



基本块的优化

- 针对基本块的局部优化
  - 针对基本块的优化可以有很好的效果
  - 基本块中各个指令要么都执行，要么都不执行

回顾：表达式的DAG

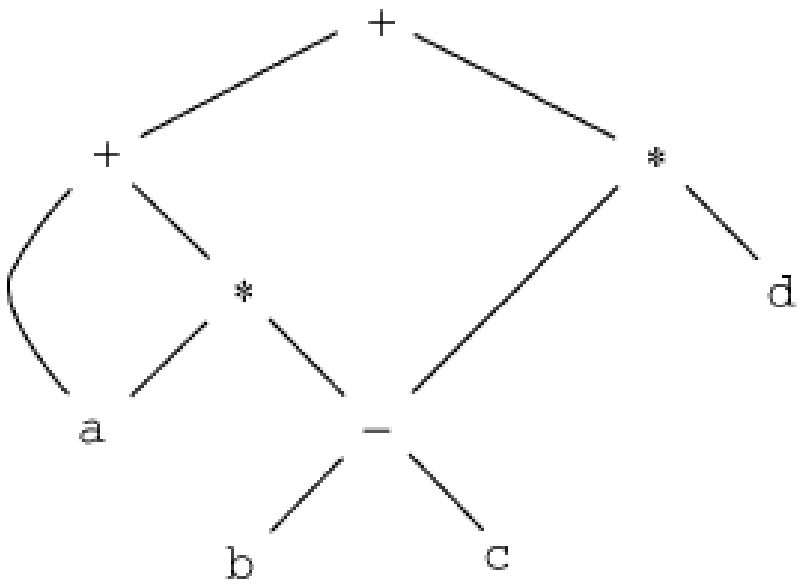
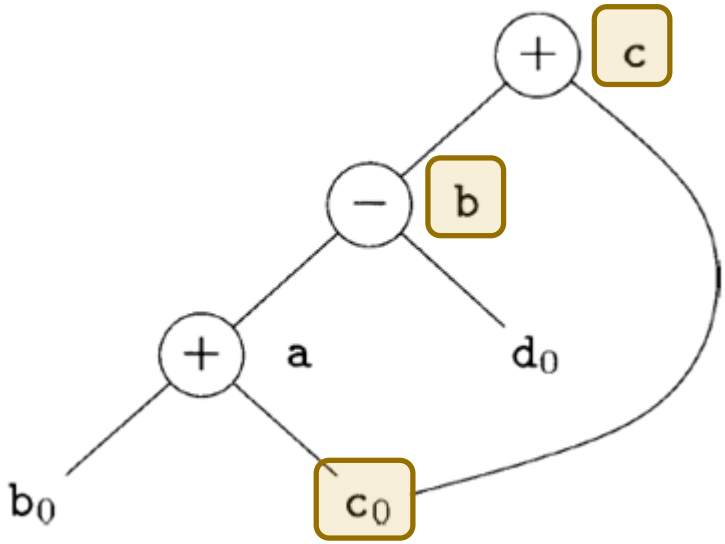


图 6-3　表达式 a + a * (b - c) + (b - c) * d 的 DAG

基本块的DAG表示

- 基本块可以用DAG表示
  - 每个变量有对应的DAG的结点，代表初始值
  - 每个语句s有一个相关的结点N，代表计算得到的值
    - N的子结点对应于（其运算分量当前值的）其它语句
    - 结点N的标号是s的运算符
    - N和一组变量关联，表示s是在此基本块内最晚对它们定值的语句
- 输出结点：结点对应的变量在基本块出口处活跃
- 从DAG，我们可以知道各个变量最后的值和初始值的关系

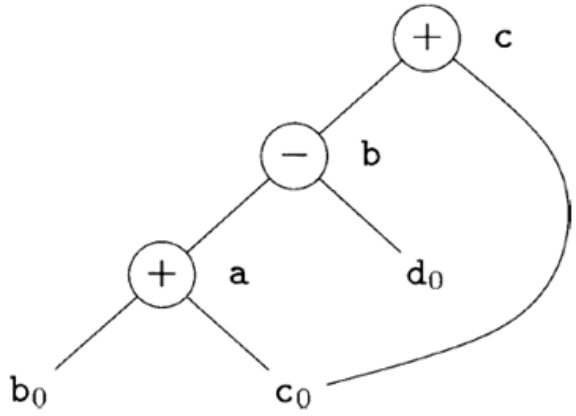
例子



- 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，它们和相应结点关联；
  - 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）
- 如果还有第四条指令c=a，如何处理DAG？

DAG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
- 顺序扫描各个三地址指令，进行如下处理
  - 如果指令为x=y op z
    - 为这个指令建立结点N，标号为op
    - N的子结点为y、z当前关联的结点
    - 令x和N关联
  - 如果指令为x=y
    - 不建立新结点
    - 设y关联到N，那么x现在也关联到N



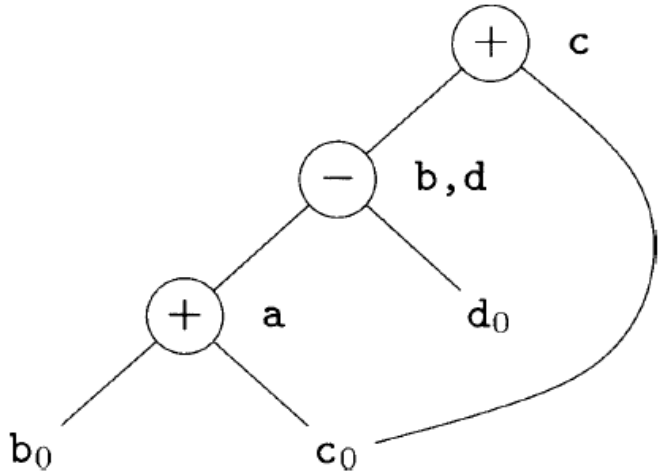
- 扫描结束后，对于所有在出口处活跃的变量x，将x所关联的结点设置为输出结点

DAG的作用

- DAG描述了基本块运行时各变量的值（和初始值）之间的关系
- 我们可以以DAG为基础，对代码进行转换
  - 消除局部公共子表达式
  - 消除死代码
  - 对语句重新排序
  - 对运算分量的顺序进行重排

局部公共子表达式

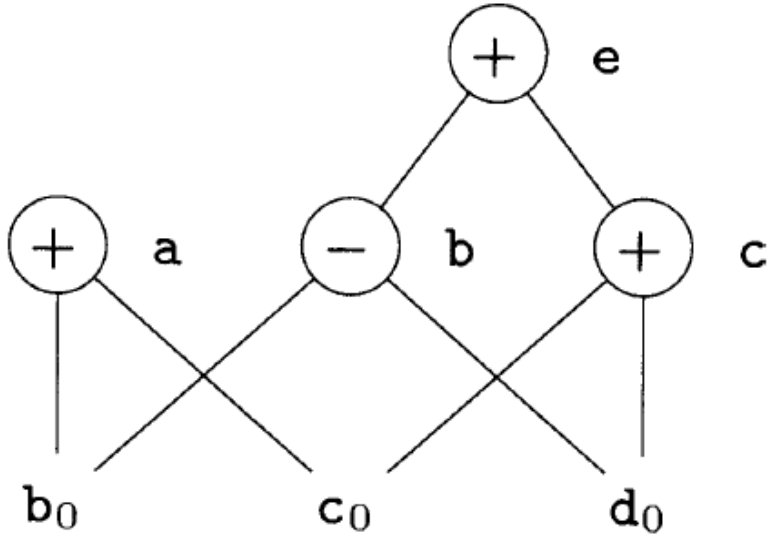
- 检测局部公共子表达式
  - 建立某个结点M之前，首先检查是否存在一个结点N，它和M具有相同的运算符和子结点（顺序也相同）
  - 如果存在，则不需要生成新的结点，用N代表M
- 例如
  - a=b+c
  - b=a-d
  - c=b+c
  - d=a-d



- 注意：两个b+c实际上并不是公共子表达式

消除死代码

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



应用代数恒等式的优化

- 消除计算步骤
  - $x+0=0+x=x$　　$x-0=x$
  - $x*1=1*x=x$　　$x/1=x$
- 强度消减
  - $x^2=x*x$　　$2*x=x+x$
- 常量合并
  - 2*3.14可以用6.28替换
- 实现这些优化时，只需要在DAG上寻找特定的模式

其他一些代数恒等式

- 条件表达式和算术表达式
  - x>y
  - x-y
- 结合律
  - a=b+c
  - e=c+d+b

```
a = b + c
t = c + d
e = t + b
```

  - 如果t只在基本块

```
a = b + c
e = a + d
```

数组引用

- 数组引用实例
  - x=a[i]
  - a[j]=y
  - z=a[i]

a[i]是公共子表达式吗？

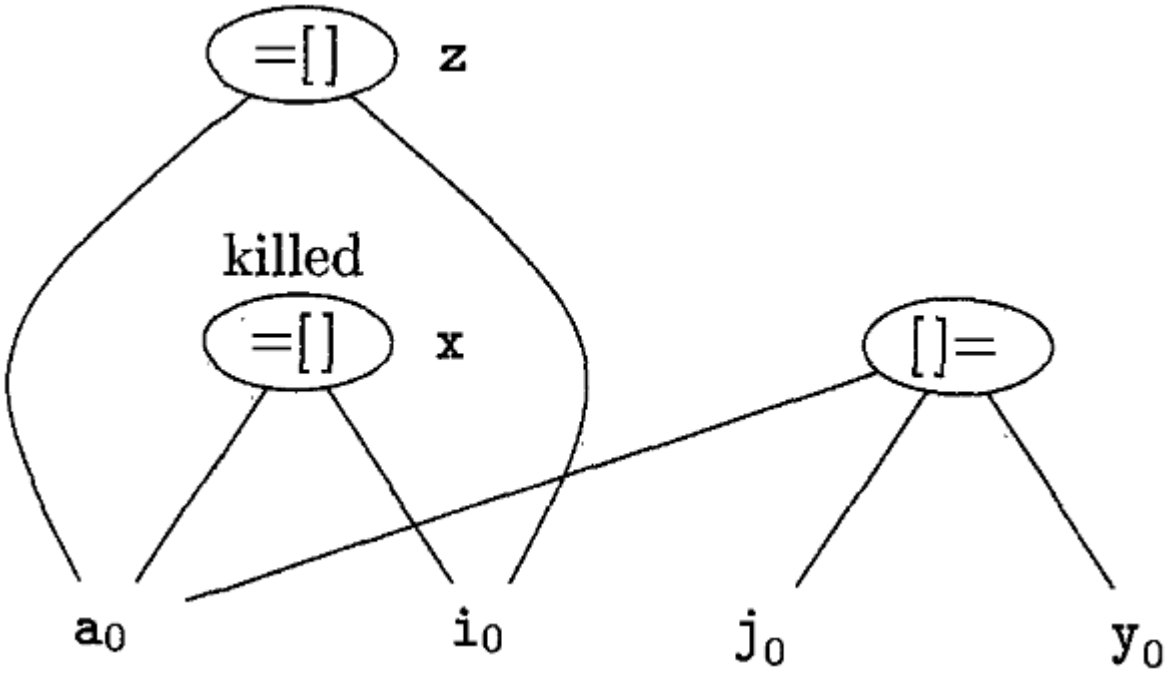
- 注意：a[j]可能改变a[i]的值，因此不能和普通的运算符一样构造相应的结点

数组引用

- 数组引用的DAG表示
  - 从数组取值的运算x=a[i]对应于=[]的结点
    - x作为这个结点的标号之一
    - 该节点的左右子节点分别代表数组初始值和下标
  - 对数组赋值（例如a[j]=y）的运算对应于[]=的结点，没有关联的变量、且杀死所有依赖于a的变量
    - 三个子节点分别表示a_0、j和y
  - 数组赋值隐含的意思：将数组作为整体考虑，对数组元素赋值即改变整个数组

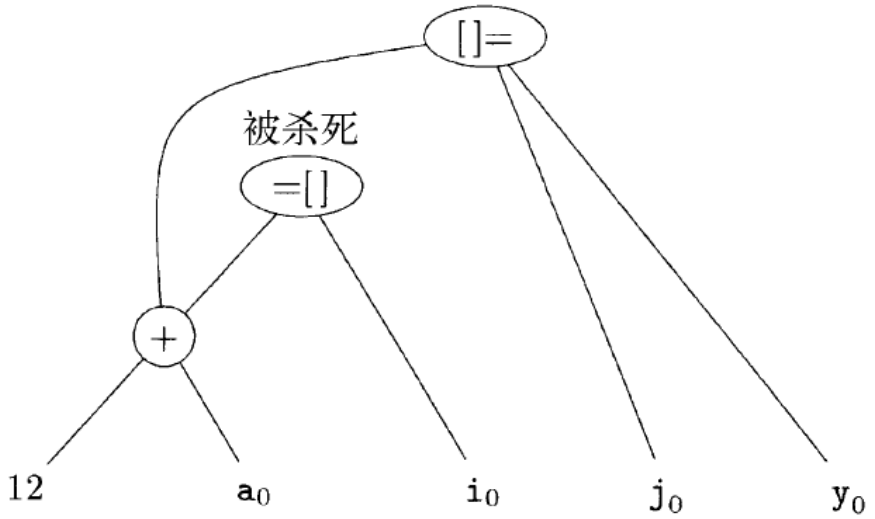
数组引用的DAG表示实例

- 基本块
  - x = a[i]
  - a[j] = y
  - z = a[i]



数组引用的DAG的例子

- 设a是数组，b是指针
  - b=12+a
  - x=b[i]
  - b[j]=y
- 注意a是被杀死的结点的孙结点
- 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]?

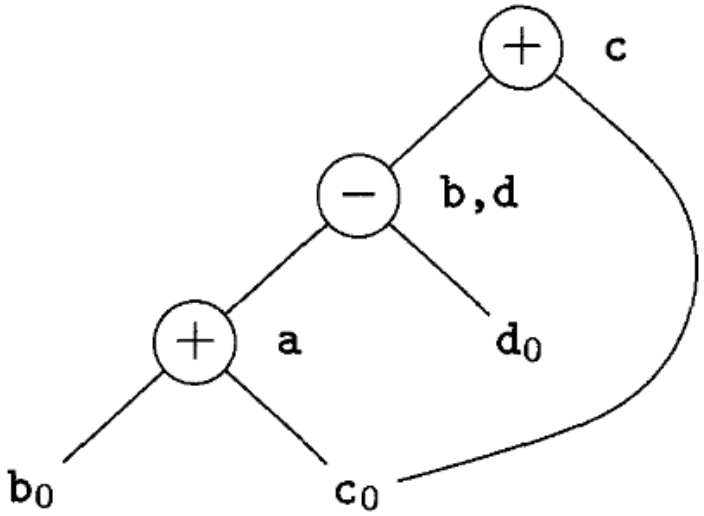


指针赋值/过程调用

- 通过指针进行取值/赋值：x=*p　　*q=y。最粗略地估计：
  - x使用了任意变量，因此无法消除死代码
  - 而*q=y对任意变量赋值，因此杀死了全部其他结点
- 可以通过（全局/局部）指针分析部分解决这个问题
- 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

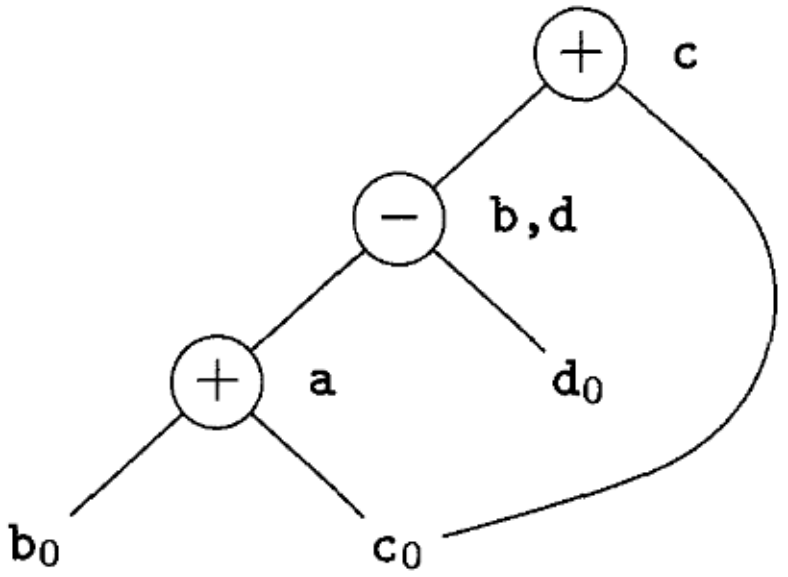
从DAG到基本块

- 重构的方法
  - 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



重组基本块的例子

```
a = b + c
b = a - d
c = b + c
d = a - d
```



- 如果b在基本块出口不活跃

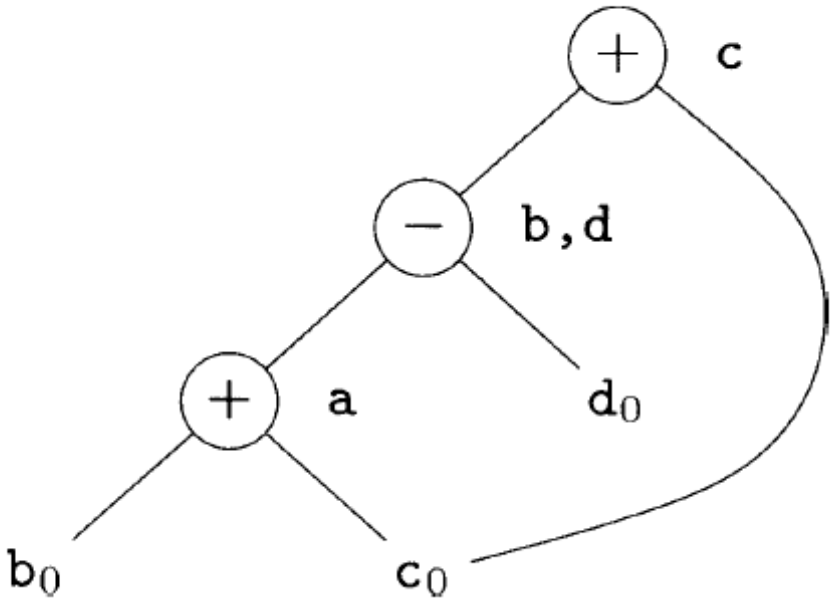
```
a = b + c
d = a - d
c = d + c
```

结果应该尽量赋给一个活跃的变量

重组基本块的例子

```
a = b + c
b = a - d
c = b + c
d = a - d
```

- 如果b和d都在基本块出口处活跃
  - a = b + c
  - d = a − d
  - b = d
  - c = d + c



重组的规则

- 重组时应该注意求值的顺序
  - 指令的顺序必须遵守DAG中结点的顺序
  - 对数组的赋值必须跟在所有原来在它之前的赋值/求值操作之后
  - 对数组元素的求值必须跟在所有原来在它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
  - 任何过程调用或者指针间接赋值必须跟在原来在它之前的变量求值之后
- 总的来说，我们必须保证：如果两个指令之间可能相互影响，那么他们的顺序就不应该改变

基本块的代码生成器

- 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

- 主要目标
  - 尽量减少加载和保存指令，即最大限度利用寄存器

代码生成器

- 重点考虑寄存器的使用方法
  - 执行运算时，运算分量必须放在寄存器中
  - 用于临时变量
  - 存放全局的值
  - 进行运行时刻管理（比如：栈顶指针）

简单方法

实际情况

- a = b + c
  - LD R0, b
  - LD R1, c
  - ADD R2, R0, R1
  - ST a, R2
- 隐含的假设
  - 无限多寄存器
  - 不知道变量是否在寄存器中
  - 所有变量以后都有用

预先判断需要哪些加载指令把必须的运算分量加载进寄存器

有限的寄存器，加载时预判选用哪个寄存器

如果必要才把结果存入内存

基本思路

- 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的值不被使用时，才把它覆盖掉

基本数据结构

- 记录各个值对应的位置
  - 寄存器描述符：跟踪各个寄存器都存放了哪些变量的当前值
  - 地址描述符：某个变量的当前值存放在哪个或哪些位置(包括内存位置和寄存器）上

代码生成算法（1）

- 代码生成算法框架
  - 遍历三地址指令
    1. 使用getReg()函数选择寄存器
    2. 生成指令

- 重要子函数：getReg(I)
  - 根据寄存器描述符和地址描述符、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（2）

- 运算语句：x = y + z
  - 调用getReg(x=y+z)，为x,y,z选择寄存器Rx, Ry, Rz
  - 查Ry的寄存器描述符，如果y不在Ry中则生成指令
    - LD Ry y’　　(y’表示存放y值的当前位置)
  - 类似地，确定是否生成LD Rz, z’
  - 生成指令ADD Rx, Ry, Rz
- 赋值语句：x=y
  - getReg(x=y)总是为x和y选择相同的寄存器
  - 如果y不在Ry中，生成机器指令LD Ry, y
- 基本块的收尾
  - 如果变量x在出口处活跃，且x现在不在内存，那么生成指令ST x, Rx

代码生成算法（3）

- 代码生成同时更新寄存器和地址描述符
- 处理普通指令时生成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
- ST x，R
  - 修改x的地址描述符，包含自己的内存位置

代码生成算法（4）

- ADD Rx, Ry, Rz
  - Rx的寄存器描述符只包含x
  - x的地址描述符只包含Rx（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Rx。
- 处理x=y时，
  - 如果生成LD Ry y，按照第一个规则处理；
  - 把x加入到Ry的寄存器描述符中（Ry存放了x和y的当前值）；
  - 修改x的地址描述符，使它只包含Ry

例子（1）

- a、b、c、d在出口处活跃
- t、u、v是局部临时变量

```
t = a - b
u = a – c
v = t + u
a = d
d = v + u
```

```
t = a - b
u = a - c
v = t + u
a = d
d = v + u
```

	R1	R2	R3	a	b	c	d	t	u	v
				a	b	c	d			

```
t = a - b
     LD R1, a
     LD R2, b
     SUB R2, R1, R2
```

	R1	R2	R3	a	b	c	d	t	u	v
	a	t		a,R1	b	c	d	R2		

```
u = a - c
     LD R3, c
     SUB R1, R1, R3
```

	R1	R2	R3	a	b	c	d	t	u	v
	u	t	c	a	b	c,R3	d	R2	R1	

```
v = t + u
     ADD R3, R2, R1
```

	R1	R2	R3	a	b	c	d	t	u	v
	u	t	v	a	b	c	d	R2	R1	R3

```
a = d
     LD R2, d
```

	R1	R2	R3	a	b	c	d	t	u	v
	u	a,d	v	R2	b	c	d,R2		R1	R3

```
d = v + u
     ADD R1, R3, R1
```

	R1	R2	R3	a	b	c	d	t	u	v
	d	a	v	R2	b	c	R1			R3

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

	R1	R2	R3	a	b	c	d	t	u	v
	d	a	v	a,R2	b	c	d,R1			R3

图 8-16　生成的指令以及寄存器和地址描述符的改变过程

getReg函数（1）

- getReg的目标：减少LD/ST指令
- 任务：为运算分量和结果分配寄存器
- 为 x = y + z 指令选择寄存器
- 为运算分量y分配寄存器
  - 如果已经在某个寄存器中，不需要进行处理，选择这个寄存器
  - 如果不在寄存器中，且有空闲寄存器，选择一个空闲寄存器
  - 如果不在寄存器中，且没有空闲寄存器？

getReg函数（2）

- 为运算分量y分配寄存器（如果不在寄存器中，且没有空闲寄存器）
  - 寻找一个寄存器R，假设已经有某个变量v在R中了（R的寄存器描述符中包含了v）
    - 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不是运算分量之一，DONE
    - 如果v在此之后不会被使用，DONE
    - 生成保存指令ST v R（溢出操作）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（选尽可能少ST的）；
- 关键：借用了v所在的寄存器，不能随意把v冲掉

getReg函数（3）

- 为x选择寄存器Rx（基本方法和把y从内存LD时一样）
- 区别
  - 只存放x的值的寄存器总是可接受的，因为x计算出了一个新的值
  - 如果y在指令I之后不再使用，且Ry仅仅保存了y的值，那么Ry同时也可以作为Rx
- 处理x=y时，我们总是让Rx=Ry

示例

- d := (a-b)+(a-c)+(a-c)的三地址代码序列:

  t_1 := a-b

  t_2 := a-c

  t_3 := t_1 + t_2

  d := t_3 + t_2

假设只有两个寄存器

statements	code generated	register descriptor	address descriptor
		registers empty	
$t_1 := a - b$	LD R0,a	R0含a	a in R0
	LD R1, b	R1含t_1	t_1 in R1
	SUB R1, R0,R1		
$t_2 := a - c$	ST t1, R1	R1含c	c in R1
	LD R1, c	R0含t_2	t_2 in R0
	SUB R0, R0, R1		
$t_3 := t_1 + t_2$	LD R1,t1	R1含t_3	t_3 in R1
	ADD R1,R1,R0		t_2 in R0
$d := t_3 + t_2$	ADD R1,R1,R0	R1含d	d in R1
		R0含t_2	t_2 in R0
	ST d, R1		d在R1和内存中

习题

假设有三个寄存器可用R1、R2、R3，使用课本介绍的getReg()分配寄存器；其中a、b、c、d是全局变量；t、u是临时变量，不需要保存）。

t = a * b

u = a – c

d = t + u

窥孔优化

- 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
  - 代数简化
  - 机器特有指令的使用

冗余指令

- 多余的LD/ST指令
  - LD a R0
  - ST R0 a
  - 且没有指令跳转到第二条指令处
- 级联跳转代码
  - if debug==1 goto L1; goto L2
  - if debug!=1 goto L2;
  - 如果已知debug一定是0，那么替换成为goto L2;

控制流优化

- goto　L1; … …;　L1：goto L2
  - goto L2; … …; goto L2
- if a<b goto L1; … … ; L1: goto L2
  - if a<b goto L2; … … ; L1: goto L2
- goto L1; … ; L1: if a<b goto L2; L3:
  - if a<b goto L2; goto L3; …; L3:

代数化简/强度消减

- 应用代数恒等式进行优化
- 消除x=x+0　　x=x*1
- x*x替换x^2

- 使用机器特有指令
  - INC，DEC，…

寄存器分配和指派

- 目的：有效利用寄存器
- 简单的基本方法：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
  - ……
- 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

寄存器分配优化

- 循环执行图示指令
- 可否优化？

- 基本块代码生成的隐含假设
  - 变量使用前要先Load
  - 活跃变量在出口处要Store

```
t = a - b
    LD R1, a
    LD R2, b
    SUB R2, R1, R2

u = a - c
    LD R3, c
    SUB R1, R1, R3

v = t + u
    ADD R3, R2, R1

a = d
    LD R2, d

d = v + u
    ADD R1, R3, R1

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

使用计数

where $use(x, B)$ is the number of times x is used in B prior to any definition of

- 如果把x放在寄存器中，每一次引用可以节省一个单位成本
- 如果在某个基本块的结尾不把x保存回内存，可以省略两个单位的开销。即如果x被分配在某个寄存器中，对于每个向x赋值且x在其出口处活跃的基本块，可以节省两个单位的开销
- 进入循环和离开循环的支出成本可以忽略
- 在循环L中把一个寄存器分配给x所得到的好处大约：

$$\sum_{L\text{中的全部基本块}B} use(x,B) + 2 * live(x,B)$$

其中，use(x,B)是x在B中被定值之前就被引用的次数。如果x在B的出口处活跃并在B中被赋予一个值，则live(x,B)的取值为1，否则取值为0。

使用计数（续）

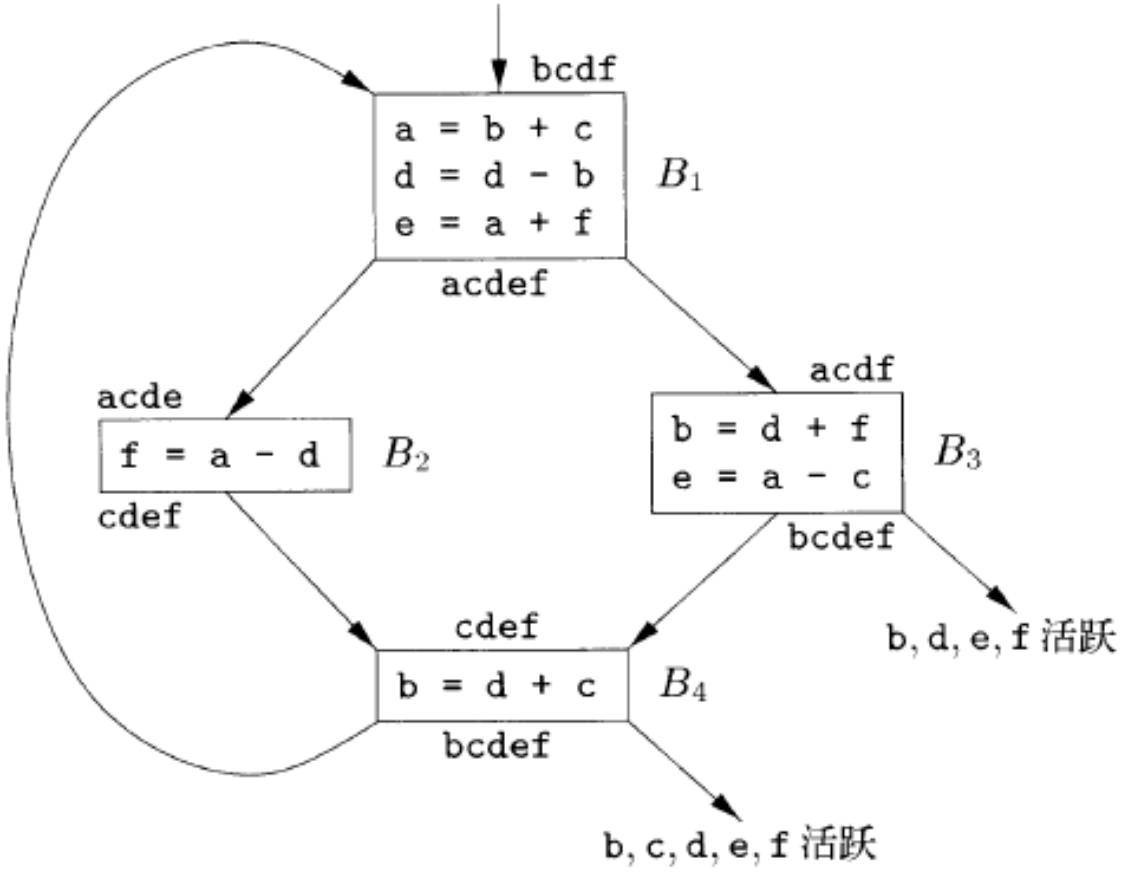


图 8-17　一个内层循环的流图

- 使用全局寄存器存放a/b/c/d/e/f，分别可以节约4/5/3/6/4/4个单位成本。
- 为a b d指派三个全局寄存器R0 R1 R2

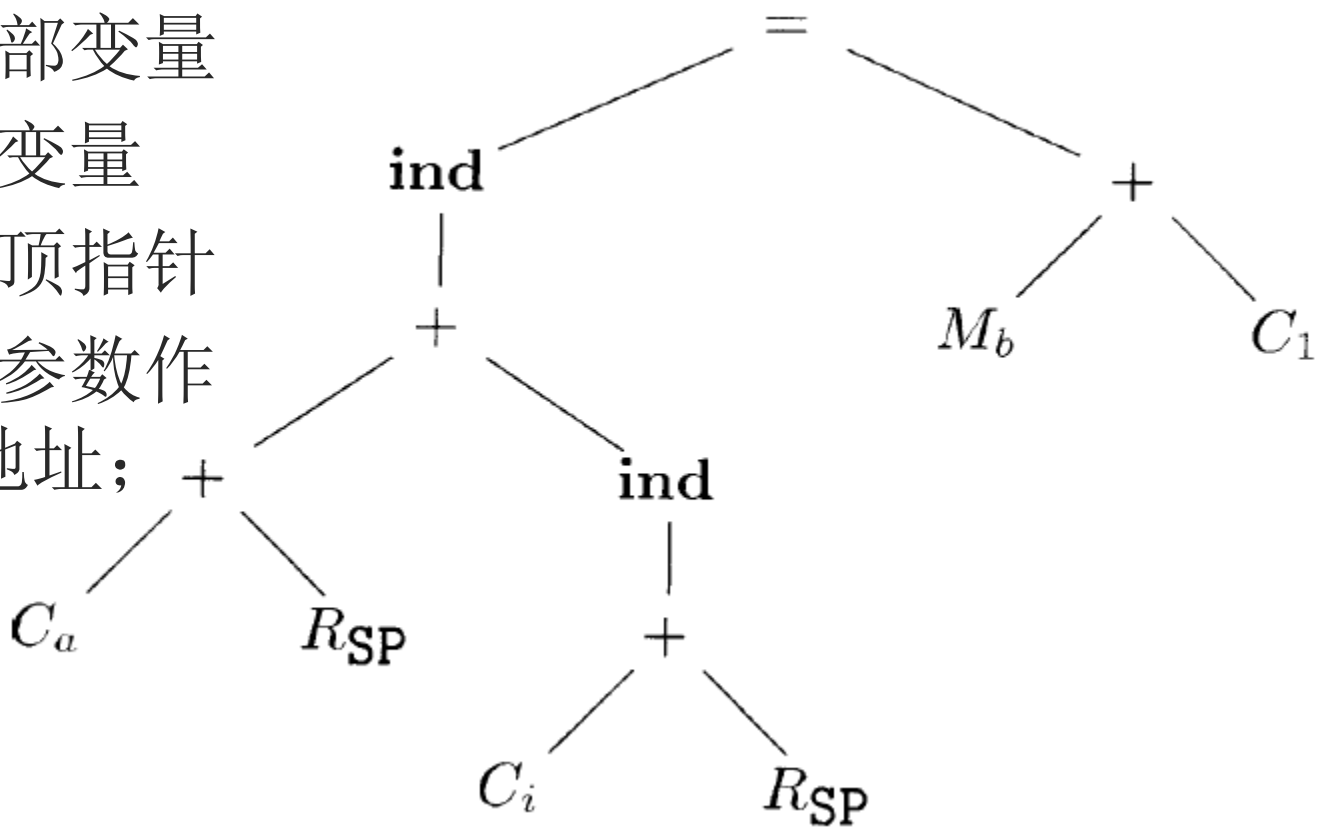
$$\sum_{L\text{中的全部基本块}B} use(x,B) + 2 * live(x,B)$$

树重写实现指令选择

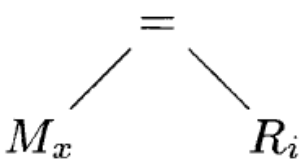
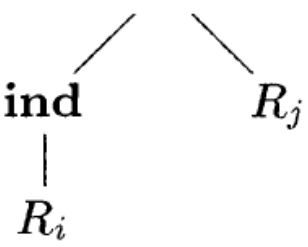
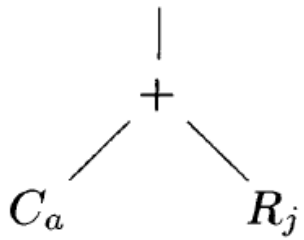
- 在某些机器上，同一个三地址指令可以使用多种机器指令实现；有时多个三地址指令可以使用一个机器指令实现
- 指令选择
  - 为实现中间表示形式中出现的运算符选择适当的机器指令
- 用树来表示中间代码，按照特定的规则不断覆盖这棵树并生成机器指令

例子（中间表示）

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



一些重写规则

1)	$R_i \leftarrow C_a$	{ LD Ri, #a }
2)	$R_i \leftarrow M_x$	{ LD Ri, x }
3)	$M \leftarrow$ 	{ ST x, Ri }
4)	$M \leftarrow$ 	{ ST *Ri, Rj }
5)	$R_i \leftarrow$ 	{ LD Ri, a(Rj) }

覆盖重写过程

- 规则1：{LD R0, #a}
- 规则7：{ADD R0 R0 SP}
- 规则6：{ADD R0 R0 i(SP)}
- 规则2：{LD R1, b}
- 规则8：{INC R1}
- 规则4：{ST *R0 R1}