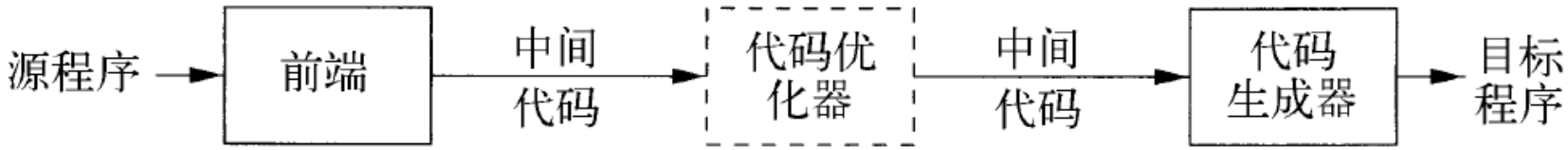
第八章 代码生成

代码生成器的位置



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
- 目标机模型
- 静态/栈式数据区分配
- 基本块相关的代码生成及优化
- 简单的代码生成算法
- 窥孔优化
- 寄存器分配和指派

8.1代码生成器设计中的问题

设计目标：

- 生成代码的正确性（最重要）
- 易于实现、测试和维护

✓输入

✓输出

✓指令选择

✓寄存器分配

✓计算顺序

代码生成器设计中的问题

输入

- 前端生成的源代码的IR(中间表示形式）及符号表信息
- 中间表示形式的选择
  - 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

- 绝对的机器语言
- 可重定向代码、汇编语言

代码生成器设计中的问题

指令选择

- 代码生成器将中间表示形式映射为目标机代码
- 映射的复杂性由下列因素决定：
- IR的层次
- 指令集体系结构本身的特性
- 期望的目标代码质量

8.2 目标机模型

使用三地址机器模型，指令如下：

- 加载
  - LD dst, addr；把地址addr中的内容加载到dst所指寄存器。addr： 内存地址/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（x的左值）
- 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中内容加上常量的和所指地址中存放的值为其地址
- 常量：#56

例子

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
- LD R2, a(R1) //R2=contents(a+contents(R1))
- ST b, R2 //b = R2

a为实数数组

实数
占8个字节

程序及指令代价

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

不可判定一个目标程序是否最优

指令代价＝指令固定代价(设为1)＋运算分量寻址模式代价，例：

- LD R0, R1；代价为1
- LD R0, M；代价是2
- LD R1, *100(R2)；代价为2

程序及指令代价

地址模式和它们的汇编语言形式及附加代价

模式	形式	地址	附加代价
绝对地址	M	M	1
寄存器	R	R	0
变址	*c*(R)	*c* + *contents*(R)	1
间接寄存器	* R	*contents*(R)	0
间接变址	* *c*(R)	*contents*(*c* + *contents*(*R*))	1
直接量	#*c*	*c*	1

常数和内存地址增加代价1，寄存器增加代价为0

例子

a、b和c都静态分配内存单元

若R0，R1和R2分别含a，b和c的地址，则

```
MOV * R1, * R0
ADD * R2, * R0
```

代价= 2

若R1和R2分别含b和c的值，并且b的值在这个赋值后不再需要，则

```
ADD R2, R1
MOV R1, a
```

代价= 3

8.3 目标代码中的地址

- Q:如何将IR中的名字转换成目标代码中的地址？

  A：程序运行时环境划分为4个区域：代码区Code、静态区Static、栈区Stack和堆区Heap。不同区域中的名字采用不同寻址方式。

如何为过程调用和返回生成代码

- 静态分配
- 栈式分配

活动记录静态分配

每个过程静态地分配一个数据区域，开始位置用staticArea表示

call callee的实现

- ST　callee.staticArea, #here+20
- //保存返回地址
- BR　callee.codeArea

callee中的语句return

- BR　*callee.staticArea

一个指令1字长

一个指令1字长

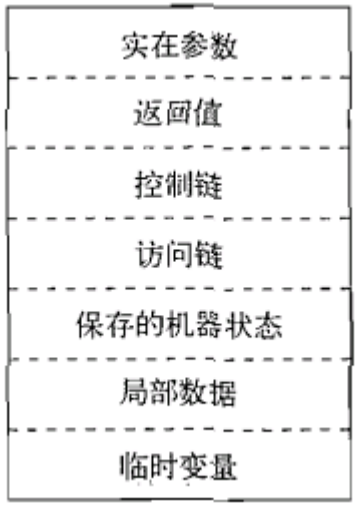
常量1字长

实际为一个常量1字长

常量1字长

返回地址

5×4



例子

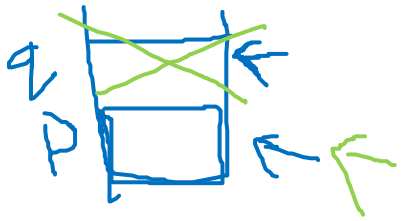
三地址代码

- action1
- call p
- action 2
- halt

//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:  140                 // 返回地址
368:                      // p 的局部数据
```



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,越过调用者自身的活动记录
- 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)　　//被调用者最后执行，返回调用者
- SUB SP, SP, #caller.recordSize
  //调用者中减低栈指针

例：快速排序

```
// 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
```

图　　栈式分配时的目标代码

例：快速排序

```
                    // 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
```

```
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
```

例：快速排序

```
                    // 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
```

```
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
```

名字的运行时刻地址

在三地址语句中使用名字（实际上是按符号表条目提供的信息）来引用变量

三地址语句x=0

- 如果x分配在开始位置为static静态区域，且符号表中保存的相对地址为12，则可以译为：
  - static[12] = 0
  - 设静态区从100开始，则也可译为
    LD　112　#0
- 如果x分配在栈区，且相对地址为12，则
  - LD 12(SP) #0

8.5 代码生成器

根据三地址序列生成指令

- 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；

寄存器主要使用方法

- 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）

算法基本思想及数据结构

尽可能留

- 依次考虑各个指令，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

尽可能用

- 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；
- 尽量保证寄存器中值不被使用时，才把它覆盖掉。

数据结构：记录各个值对应的位置

- 寄存器描述符：跟踪哪些变量的当前值存放在寄存器内；有M个对应于M个寄存器
- 地址描述符：在哪个或哪些位置上可以找到该变量的当前值；有N个对应于N个变量

代码生成算法（1）

重要子函数：getReg(I)

- 为三地址指令I选择寄存器；
- 根据寄存器描述符和地址描述符、以及数据流信息来分配最佳的寄存器；
- 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）

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’是存放y的内存位置之一)
- 类似地，确定是否生成LD Rz,z’
- 生成指令ADD Rx, Ry, Rz

复制语句：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）

代码生成时必须更新寄存器描述符和地址描述符

LD R x

- R的寄存器描述符：只包含x（因为R被修改了）
- x的地址描述符：R作为x的新位置，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，（因为R中仅有x了）

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！因为最新的x值只这有）
- 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

- LD按照第一个规则处理；
- 把x加入到Ry中；**
- 修改x的地址描述符，使它只包含Ry。

例：

已知：

- a、b、c、d在出口处活跃
- t、u、v是局部临时变量

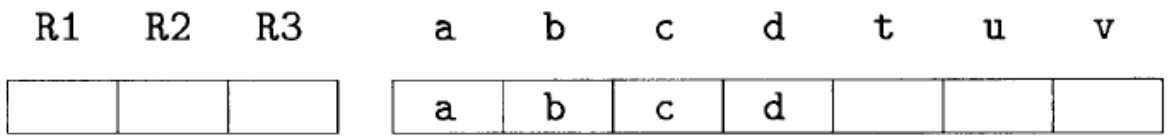
将基本块翻译成代码,只有三个寄存器

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

例：

寄存器描述符　　地址描述符

初始




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

图8-16 生成的指令及寄存器和地址描述符的改变过程

例：

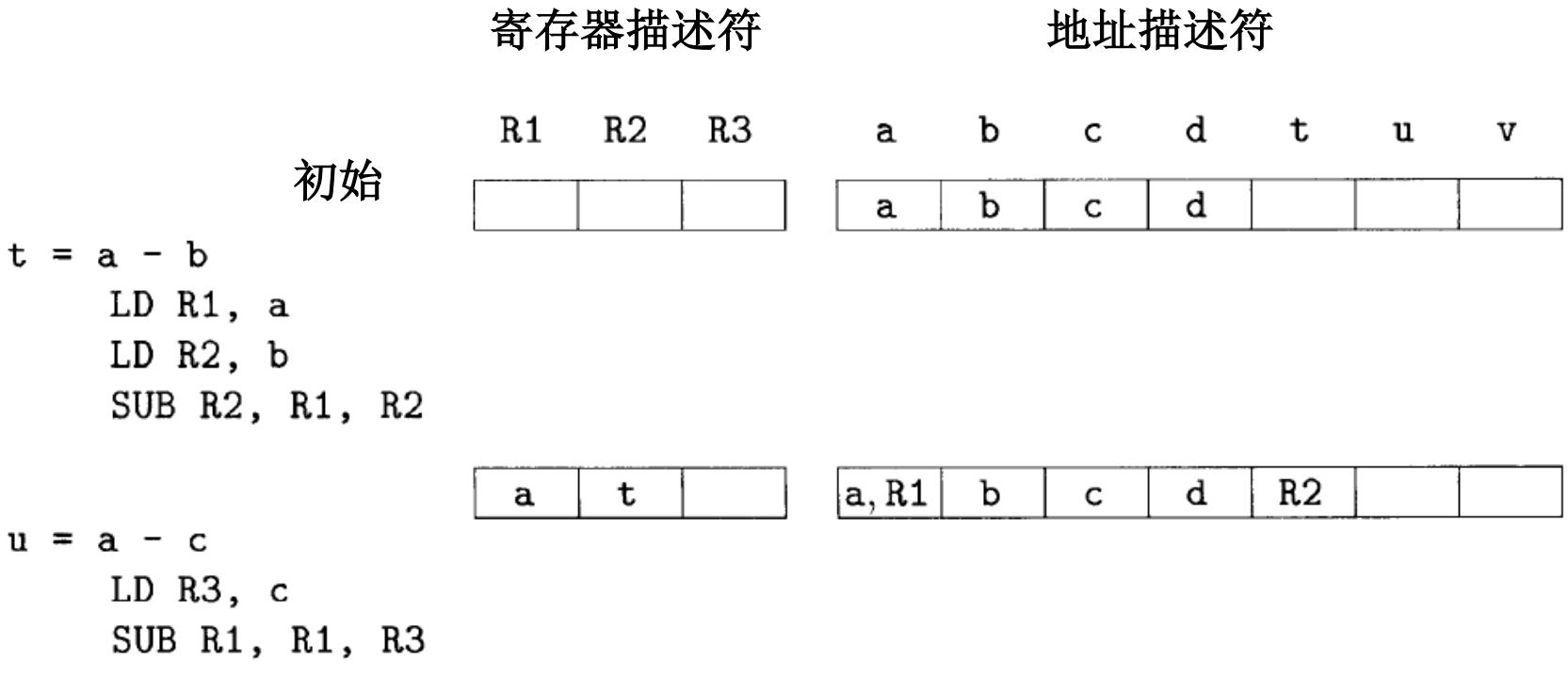


图8-16 生成的指令及寄存器和地址描述符的改变过程

例：

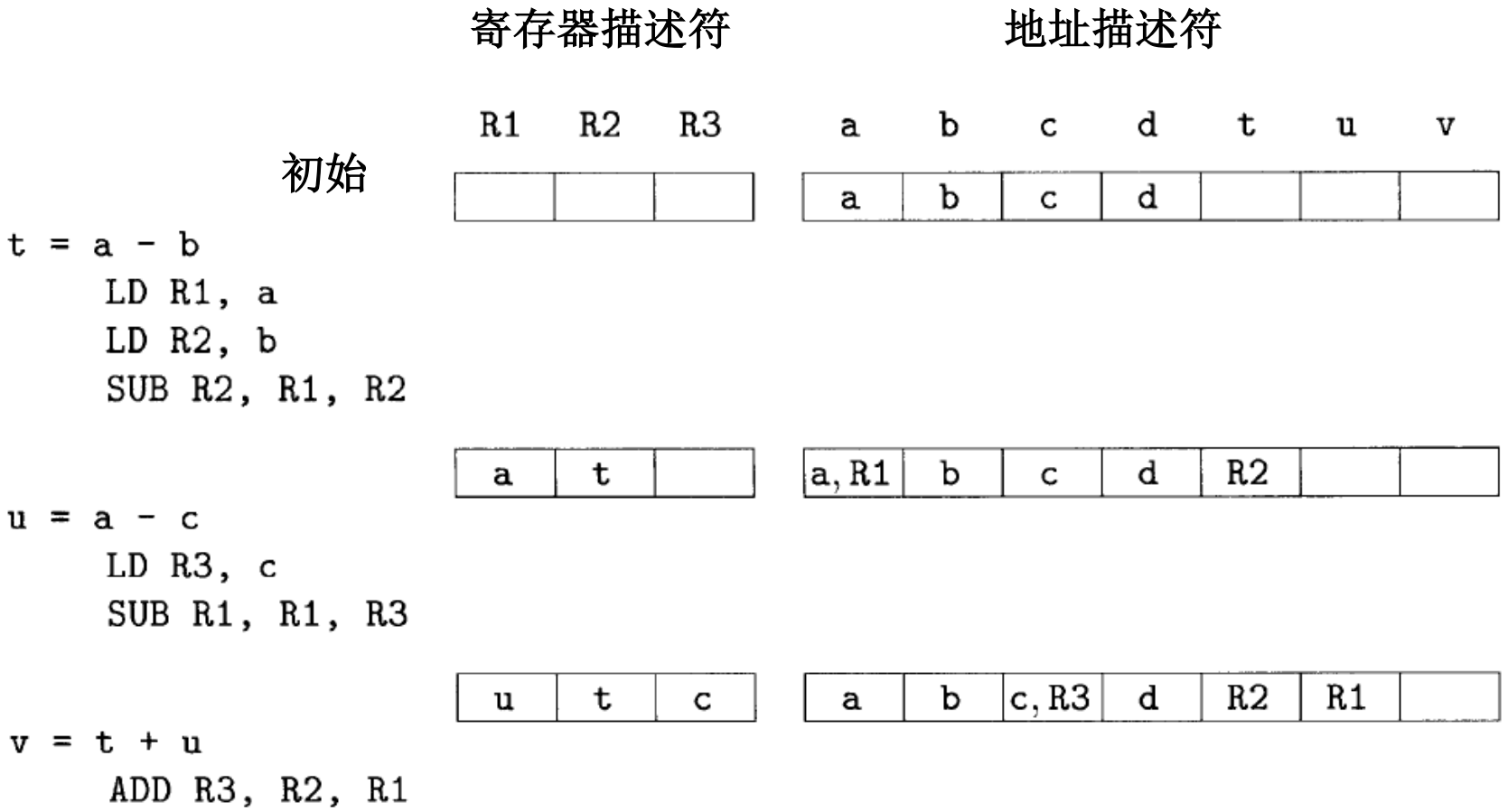


图8-16 生成的指令及寄存器和地址描述符的改变过程

例：

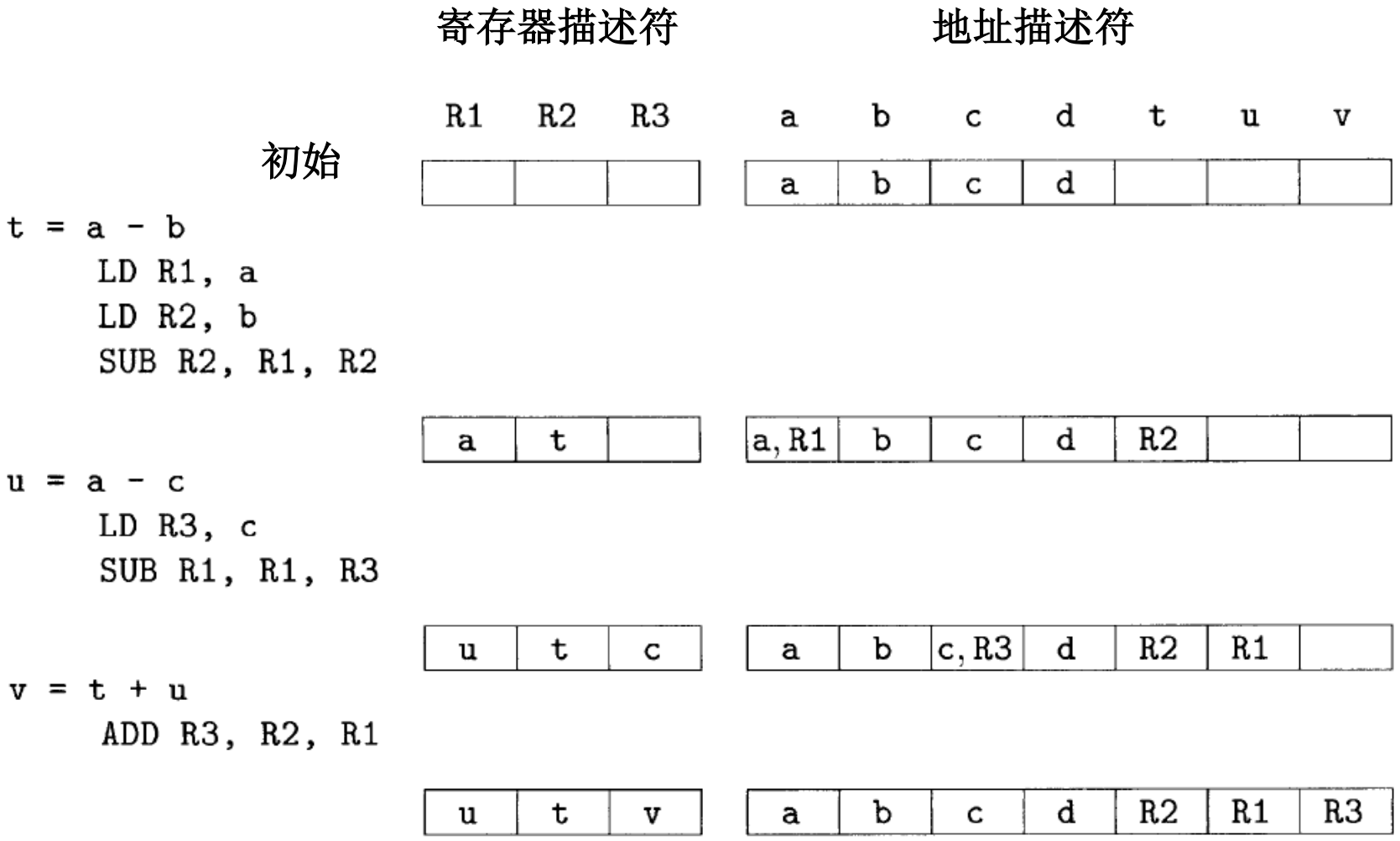


图8-16 生成的指令及寄存器和地址描述符的改变过程

例：

寄存器描述符　　地址描述符

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

图8-16 生成的指令及寄存器和地址描述符的改变过程

例：

寄存器描述符　　地址描述符

```
a = d
      LD R2, d

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

u	a, d	v

R2	b	c	d, R2		R1	R3

图8-16 生成的指令及寄存器和地址描述符的改变过程

例：

寄存器描述符　　地址描述符

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

u	a, d	v

R2	b	c	d, R2		R1	R3

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

d	a	v

R2	b	c	R1			R3

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

图8-16 生成的指令及寄存器和地址描述符的改变过程

例：

寄存器描述符　　地址描述符

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

u	a, d	v

R2	b	c	d, R2		R1	R3

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

d	a	v

R2	b	c	R1			R3

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

d	a	v

a, R2	b	c	d, R1			R3

图8-16 生成的指令及寄存器和地址描述符的改变过程

8.6 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

消除冗余指令

多余的LD/ST指令

- LD R0 a
- ST a R0
- 且没有指令跳到第二条指令处，即第二指令没有标号。可以删除第二条指令

消除不可达指令

紧跟在无条件转移指令后不带标号的指令

- 重复这个操作，可删除一个指令序列

级联跳转代码

- 比如示例：调试代码（当debug＝1时才运行的程序片断）的原始中间代码可能如下：
  - if debug==1 goto L1
  - goto L2
  - L1:print debugging information
  - L2:

  有两个GOTO

消除不可达指令

级联跳转代码消除

- 调试代码（当debug＝1时才运行的程序片断）的中间代码形如：
  - if debug==1 goto L1
  - goto L2
  - L1:print debugging information
  - L2:
- 可替换为：
  - if debug!=1 goto L2
  - print debugging information
  - L2:

消除不可达指令

级联跳转代码

- 调试代码可替换为：
  - if debug!=1 goto L2
  - print debugging information
  - L2:
- 如果已知debug一定是0，那么第一条指令替换成为goto L2；（比如，生成release版本）
- 替换后，打印调试信息的所有指令变为不可达指令，可删除

控制流优化

goto　L1;

… …;

L1: goto L2

替换为：

goto　L2;

… …;

L1: goto L2

若没有跳转到L1的指令，且指令L1: goto L2之前是一个无条件跳转指令，则L1该行指令可删除。

控制流优化

if a<b goto L1

… …

L1: goto L2

替换为：

if a<b goto L2

… …

L1: goto L2

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

- 应用强度消减转换进行优化

  x*x替换x^2

- 使用机器特有指令
  - INC，DEC，…

8.7 寄存器分配和指派

寄存器操作比内存操作代码短，速度快

分配：确定哪些值保存在寄存器中

指定：确定每个值具体保存在哪个寄存器中

寄存器分组

- 基地址、数学运算、栈地址…
- 简单、低效

8.7.1 全局寄存器分配

基本块出口，寄存器→内存

- 避免复制，将最常用的变量保持在寄存器中
- 跨越基本块边界——全局
- 使用固定一组寄存器保存每个内循环中最活跃的变量
- C编译器中，程序员指定寄存器分配

8.7.2 引用计数

假定访问寄存器比访问内存节省开销1

循环L，定义x

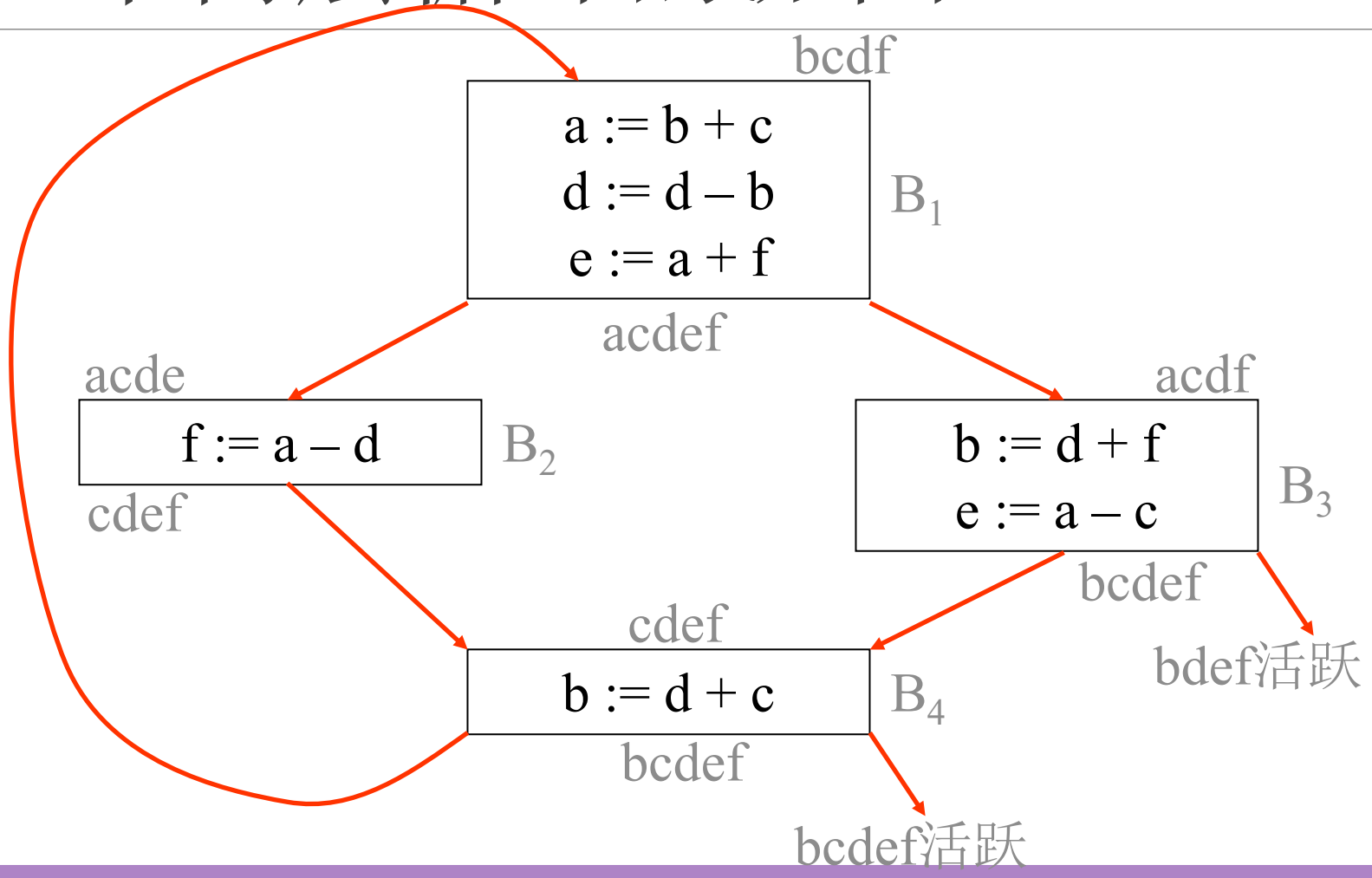
- 之前：定义之后若有引用，x将保留在寄存器中，而定义之前的引用需访问内存
- 优化：x一直保存在寄存器中，定义之前的引用变为访问寄存器，节省开销1

8.7.2 引用计数

- 块结束时变量活跃，在块中被定义，后继块中被引用
  - 之前：保存至内存，后继块又需读出到寄存器
  - 优化：无需保存和读出，节省开销2

$$\sum_{L\text{中的块}B}(\text{use}(x,B)+2*live(x,B))$$

例:一个内层循环的流图


bcdf
a := b + c
d := d – b
e := a + f
B1
acdef
acde
f := a – d
B2
cdef
acdf
b := d + f
e := a – c
B3
bcdef
bdef活跃
cdef
b := d + c
B4
bcdef
bcdef活跃

例:

R0，R1，R2：存放整个循环内的值

对变量a

- 只在B_1出口活跃，
- use(a, B_1) = use(a, B_4) = 0,
  use(a, B_2) = use(a, B_3) = 1
- 总共节省4

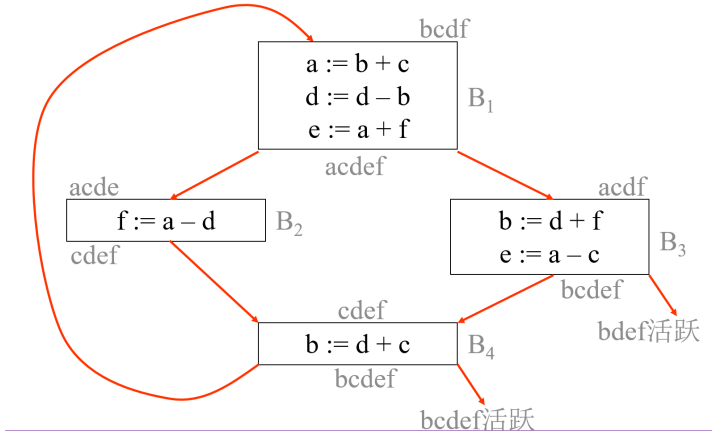
$$\sum_{L\text{中的}B} 2 * live(a, B) = 2$$

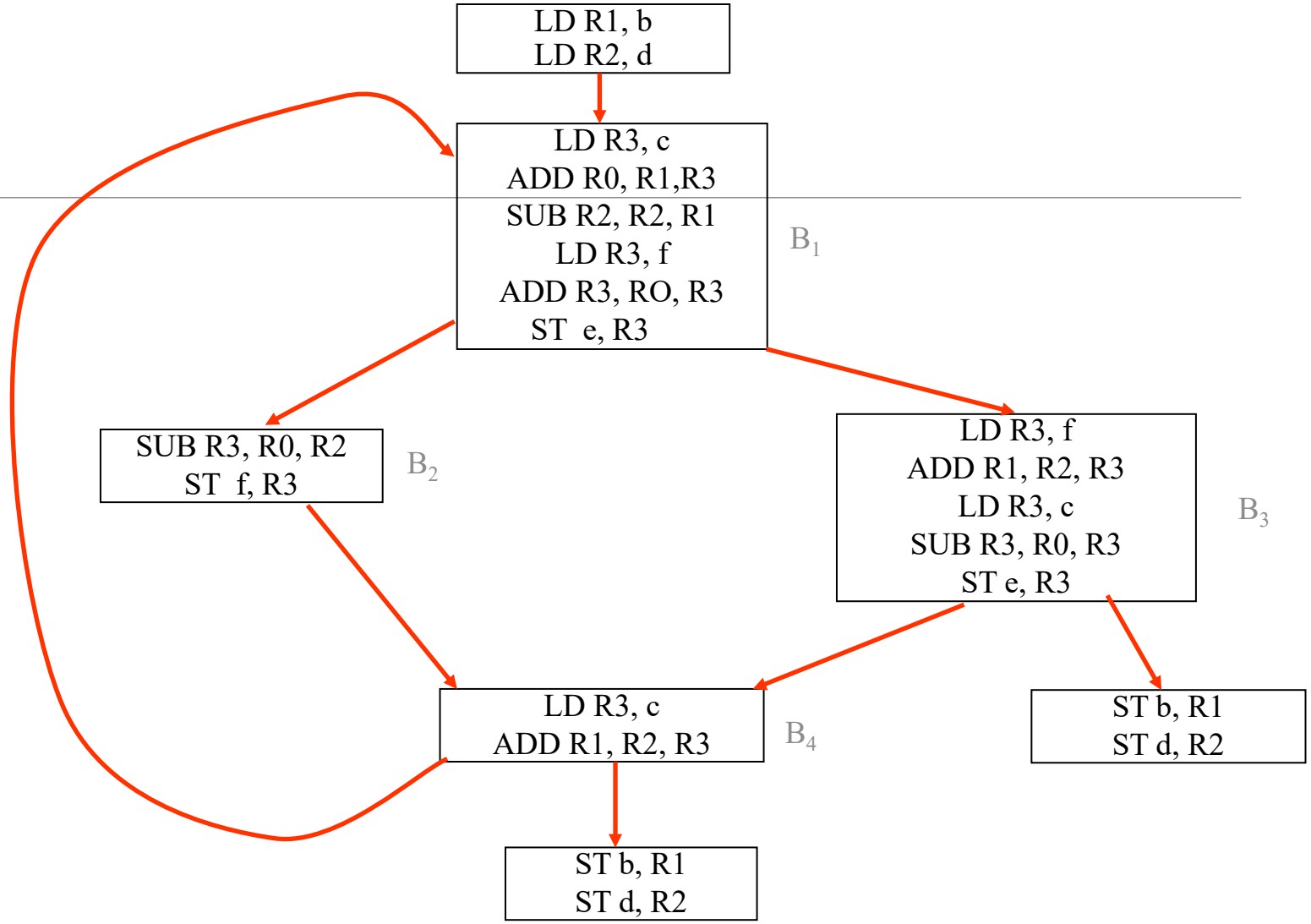
$$\sum_{L\text{中的}B} use(a, B) = 2$$

b, c, d, e, f——5, 3, 6, 4, 4

可将a、b、d保存在R0、R1、R2中

例:（续）


bcdf
a := b + c
d := d – b
e := a + f
B1
acdef
acde
f := a – d
B2
cdef
acdf
b := d + f
e := a – c
B3
bcdef
bdef活跃
cdef
b := d + c
B4
bcdef
bcdef活跃


LD R1, b
LD R2, d
LD R3, c
ADD R0, R1,R3
SUB R2, R2, R1
LD R3, f
ADD R3, RO, R3
ST e, R3
B1
SUB R3, R0, R2
ST f, R3
B2
LD R3, f
ADD R1, R2, R3
LD R3, c
SUB R3, R0, R3
ST e, R3
B3
LD R3, c
ADD R1, R2, R3
B4
ST b, R1
ST d, R2
ST b, R1
ST d, R2

8.7.3 外层循环的寄存器分配

内层循环相同思想，L_1包含L_2

- x在L_2中分配了寄存器，则在$L_1 - L_2$不必再分配
- x在L_1中分配了寄存器，而L_2中没有，则在L_2入口需保存，出口需读出
- x在L_2中分配了寄存器，而L_1中没有，则在L_2入口需读取，出口需保存

8.7.4 图着色法进行寄存器分配

两次扫描

1. 假定寄存器数目是无限的，选择目标机器指令翻译中间代码——每个变量一个寄存器，符号寄存器
2. 分配物理寄存器
   - 寄存器冲突图，register-interference graph
     - 节点——符号寄存器，R_1—R_2，R_2定义的位置上R_1活跃
     - k——可用物理寄存器数，用k个颜色为图着色
     - 相邻节点不同颜色——干扰变量使用不同寄存器
     - 启发式算法
       - n邻居数<k，去掉n → G’，G’可k着色→ G可k着色
       - 最终——空图（成功）或所有节点邻居数>=n（失败）

总结

代码生成器设计中的问题

目标机模型

静态/栈式数据区分配

基本块相关的代码生成及优化

简单的代码生成算法

窥孔优化

寄存器的分配与指派

8.4 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)，其特点是单入口单出口，即：
  - 控制流只能从第一个指令进入
  - 除了基本块最后一个指令，控制流不会跳转/停机
- 流图中结点是基本块，边指明了哪些基本块可以跟在一个基本块之后运行

流图可作为优化的基础

- 它指出了基本块之间的控制流
- 可根据流图了解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leader指令（基本块的第一个指令）符合以下任一条：
  - 中间代码的第一个三地址指令
  - 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（包含）开始到下一个首指令（不含）

基本块划分举例

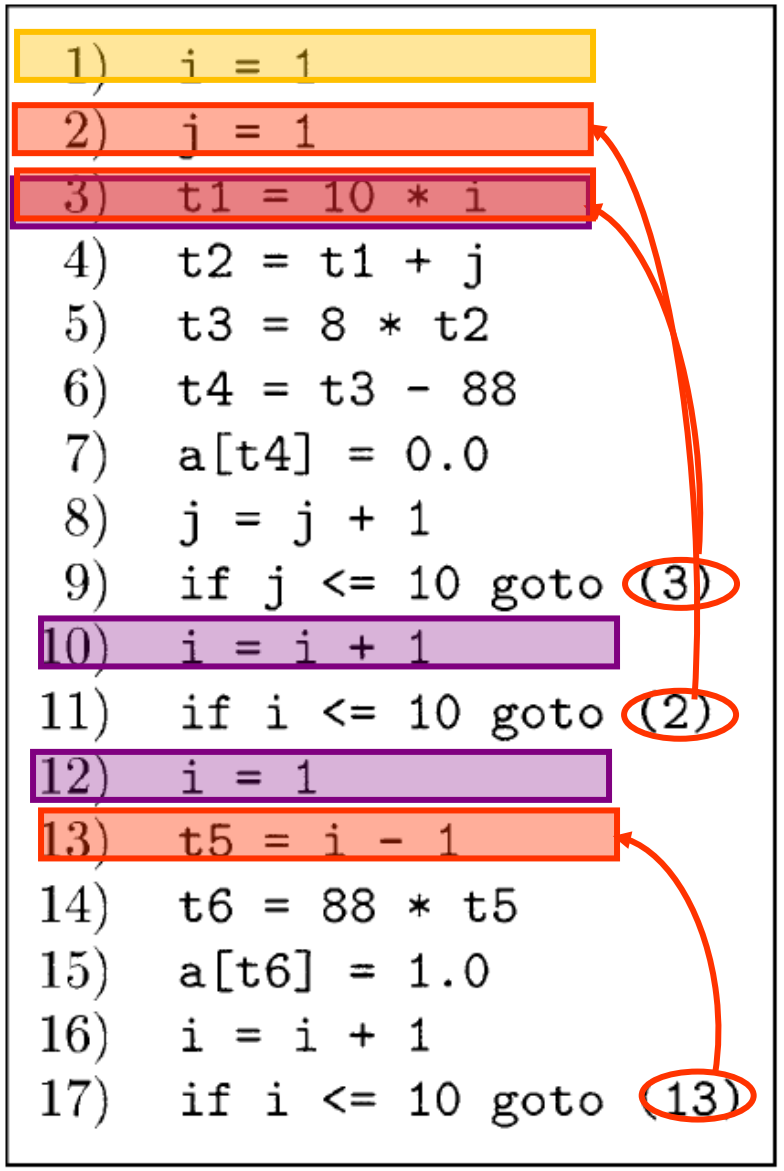
第一个指令
- 1

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(包括跳转语句)；
10-11(包括跳转语句)；
12-12；
13-17 (包括跳转语句)



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的第一个指令有可能在B的最后一个指令之后执行。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
- 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
- B是C的前驱，C是B的后继

流图中增加额外的入口，出口结点各一个

- 不对应于实际的中间指令（是增加）
- 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

流图绘制

```
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)
```

流图的例子

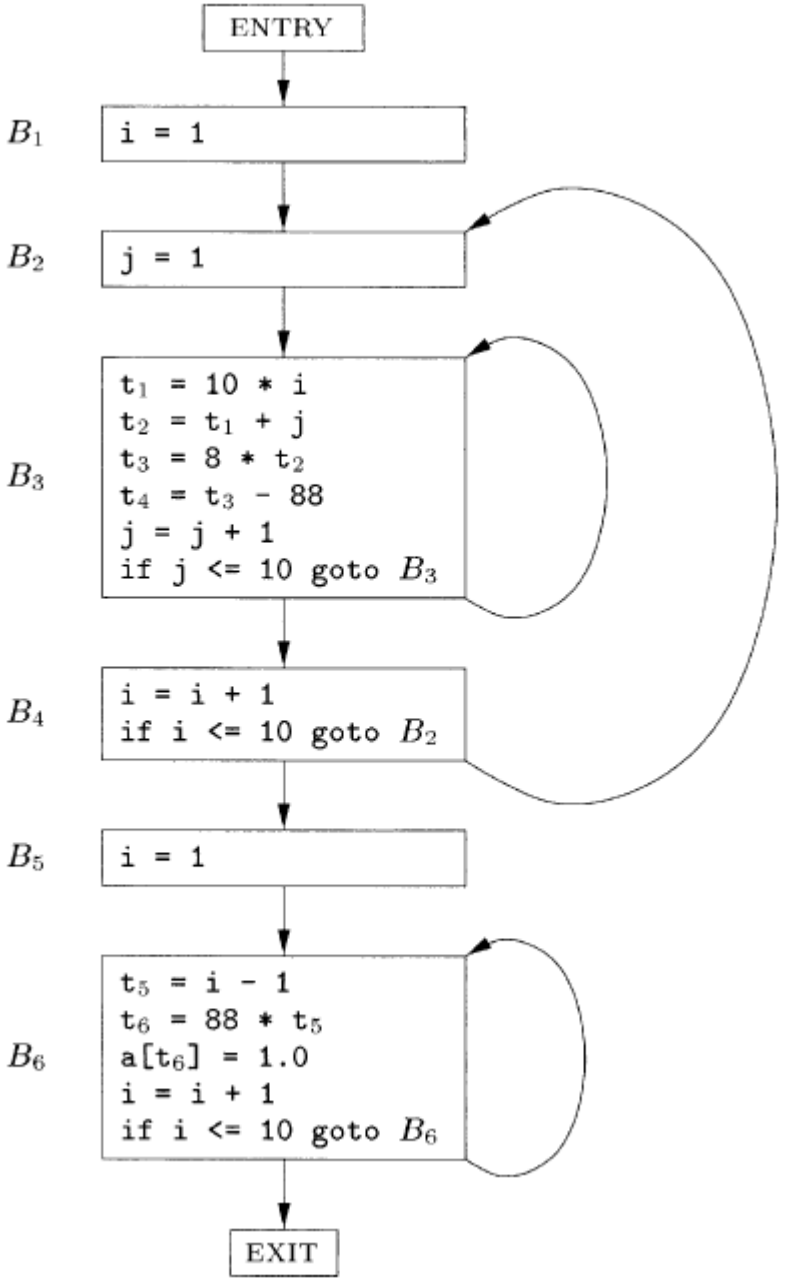
因跳转而生成的边

- B3 → B3
- B4 → B2
- B6 → B6

因为顺序而生成的边

- 其它

```
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)
```



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

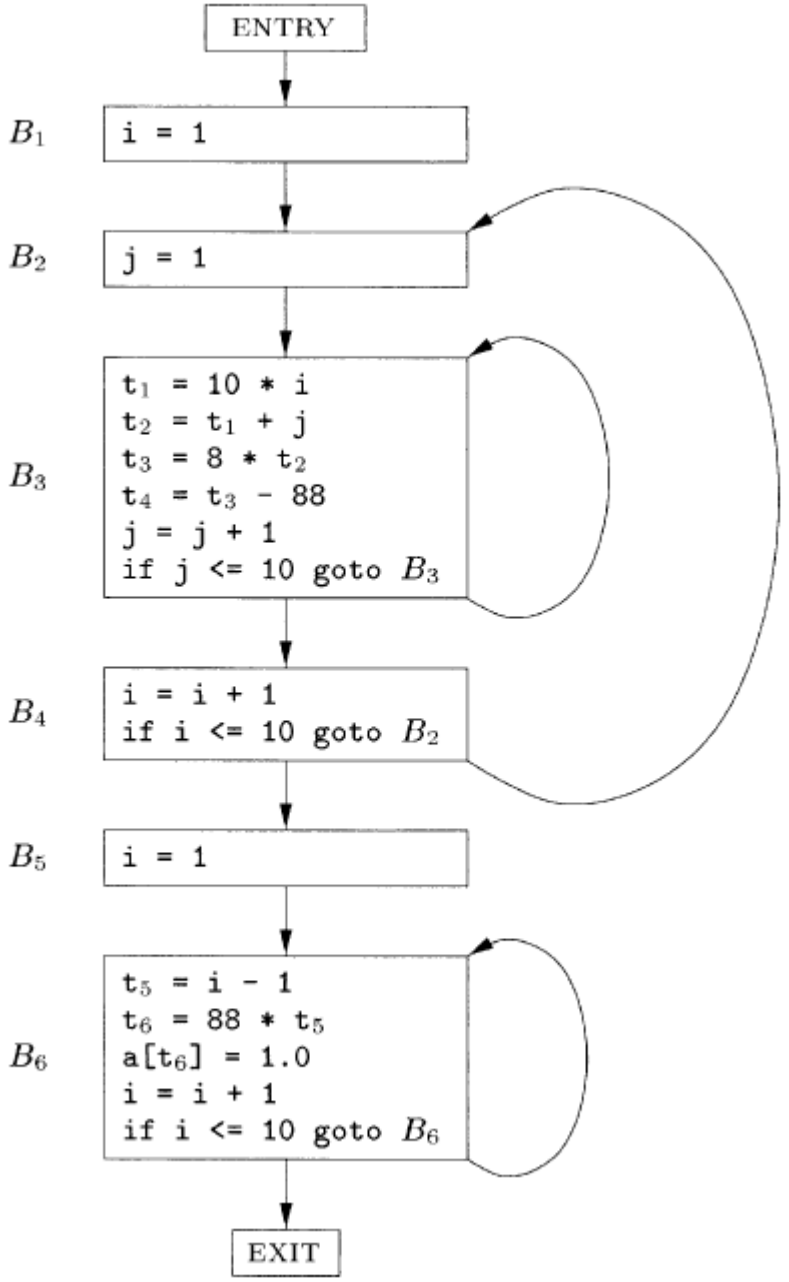
循环的定义

- 循环L是一个结点集合
- 存在一个循环入口（loop entry）节点，其唯一的前驱可以是循环L之外的结点
- 其余结点都存在到达L的入口的非空路径，且路径都在L中。

循环的例子

循环

- {B3}
- {B6}
- {B2,B3,B4}

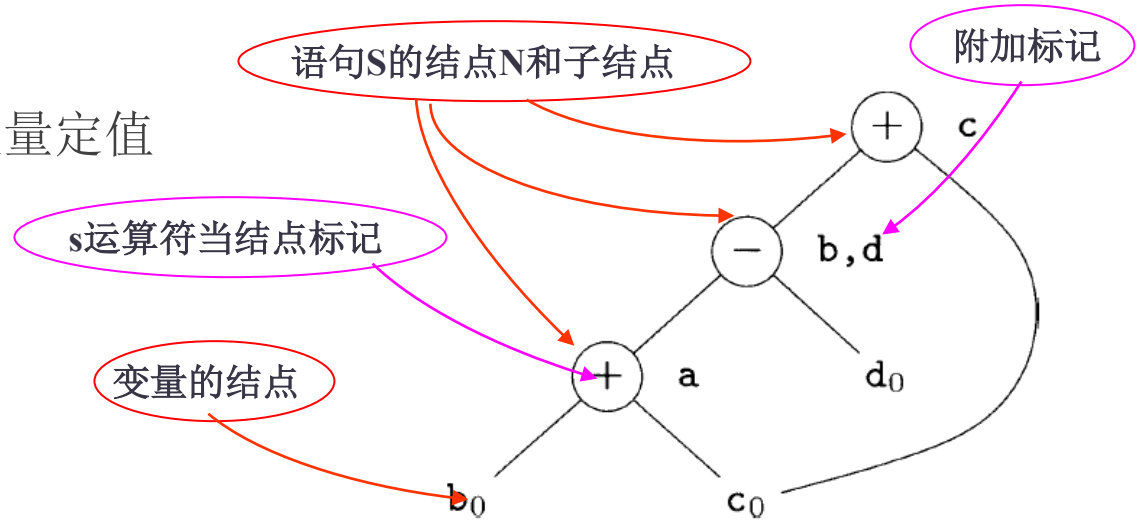


基本块的几种优化

一个基本块可用一个DAG图表示(基本的DAG图)

- 每个变量对应于一个结点，表示初始值
- 每条语句s有一个相关结点N,具有
  - 子结点：对应于其它语句,是在s之前，最后一个对s所使用的某个运算分量进行定值的语句。
  - 标记：s的运算符
  - 附加标记：一组变量，表明s是在此基本块内最晚对该变量定值
- 某些输出结点：结点对应的变量在基本块出口处活跃

（出口活跃属于全局数据流分析）



DAG示例

DAG图的构造

为基本块中出现的每个变量建立结点（表示基本值）

顺序扫描各个三地址指令

- 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；
  - 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图描述了基本块运行时各个值之间的关系。

可以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
- 重新排序运算分量的顺序

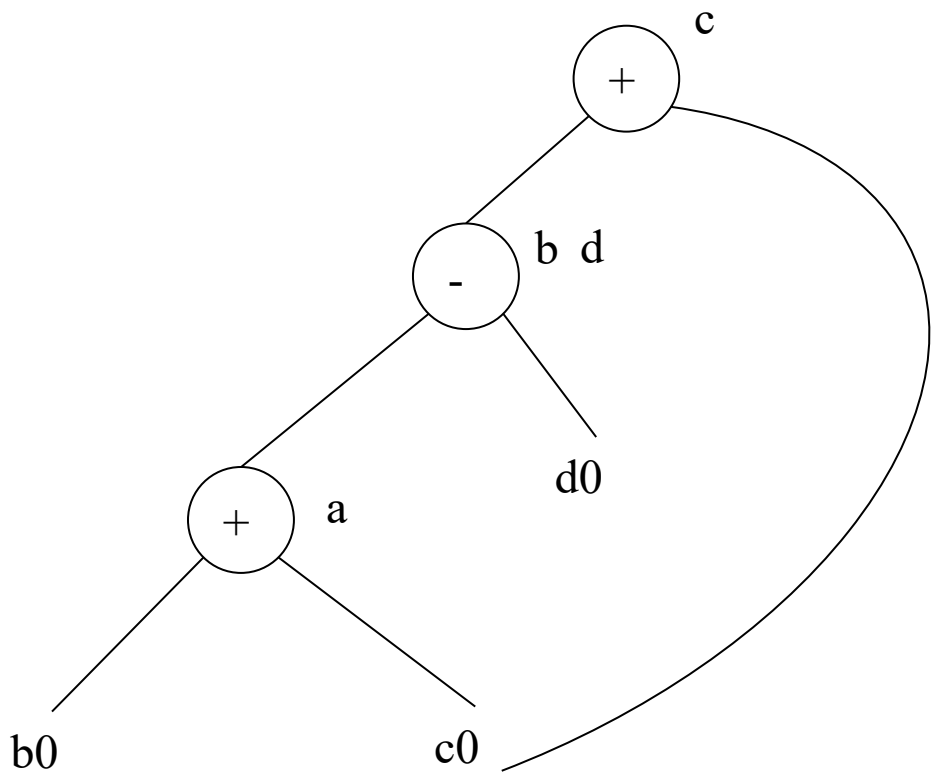
局部公共子表达式

局部公共子表达式的发现

- 建立某个结点M之前，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
- 找出公共的表达式？



注意：两个b+c实际上并不是公共子表达式

消除死代码

死代码：是指在程序操作过程中永远不可能被执行到的代码。

在DAG图上消除没有附加活跃变量的根结点（没有父结点的结点），即可消除死代码

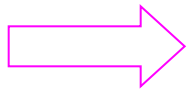
如果图中c、e不是活跃变量，则可以删除标号为e、c的结点。

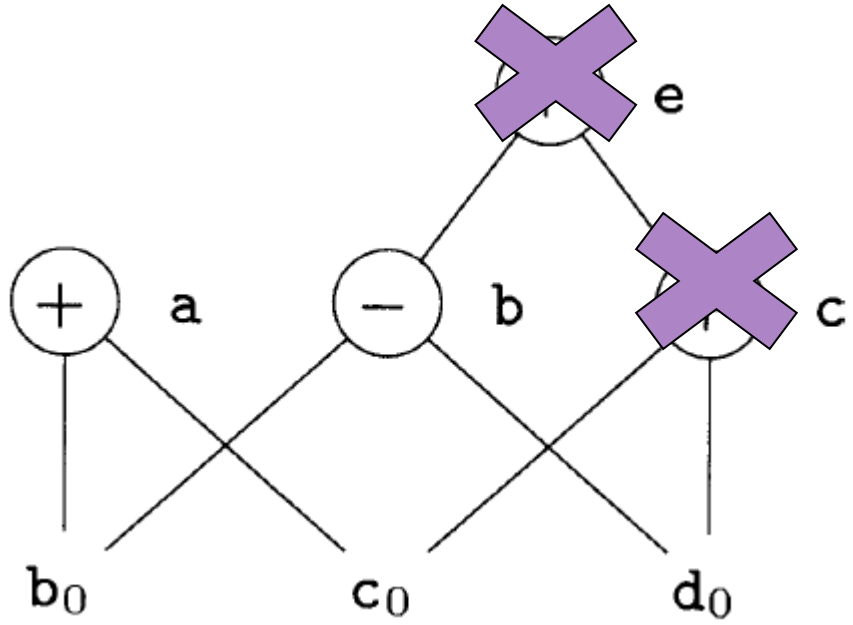
$$a = b + c$$

$$b = b - d$$

$$c = c + d$$

$$e = b + c$$





DAG方法的不足

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

为了计算第四式的e值，而有的2,3式实际上是冗余的，也就是第一式和第四式等价。该代码本可优化，但发现不了

$$b + c = (b - d) + (c + d)$$

在DAG上应用代数恒等式的优化

消除计算步骤

- $x+0=0+x=x$　　$x-0=x$
- $x*1=1*x=x$　　$x/1=x$

降低计算强度

- $x^2=x*x$　　$2*x=x+x$

常量合并

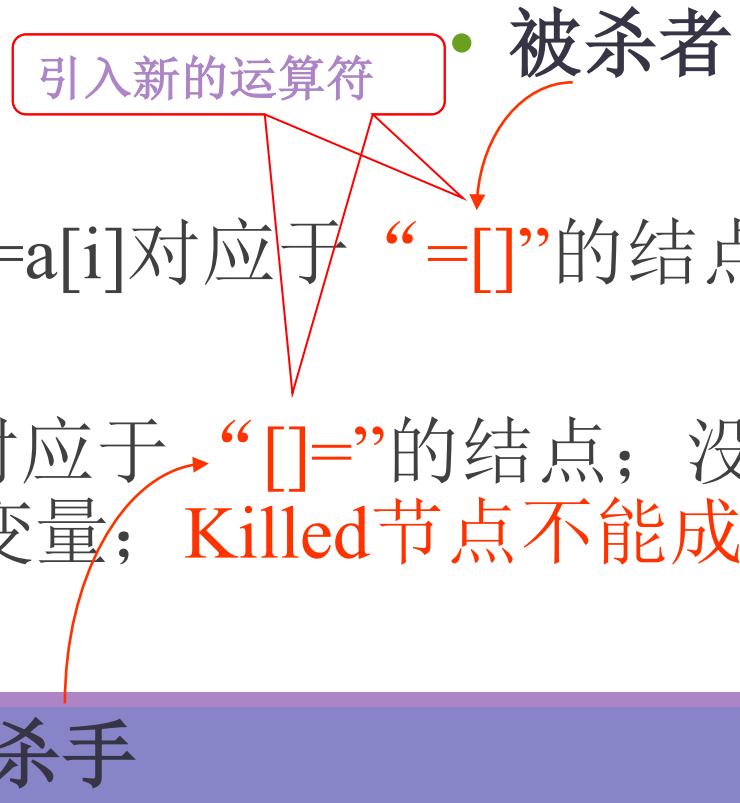
- 2*3.14可以用6.28替换

实现这些优化时，只需要在DAG图上寻找特定的模式

数组引用—避免误优化

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

- 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，x作为这个结点的标号之一；

对数组赋值的运算对应于“[]=”的结点；没有关联的变量、且杀死所有依赖于a的变量；Killed节点不能成为公共子表达式

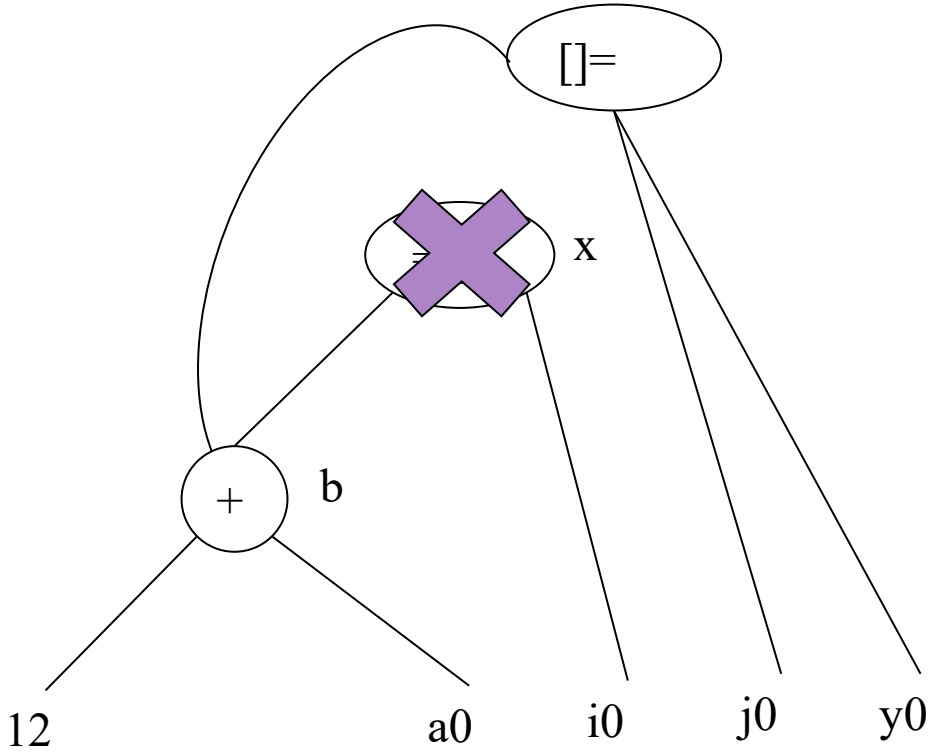
数组引用的DAG的例子

b=12+a

x=b[i]

b[j]=y

注意a是被杀死结点的孙结点。



- 如果没有杀手的出现则被杀者本可以象前述一样进行优化。
- 杀死的意思——指不能参加优化

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(可类似引出新的运算符“=*”与“*=”帮助分析)

杀的范围过大。可以通过（全局/局部）指针分析,缩小范围；比如针对

- p=&x
- *p=y可以只杀死那些以x为附加变量的结点

过程调用也类似，为了安全：

- 必须假设它使用了访问范围内所有变量
- 假设修改了访问范围内的所有变量 。杀谁？
- 全杀 !!

从DAG到基本块

重构的方法

- 对每个结点构造一个三地址语句来计算对应的值
- 结果应该尽量赋给一个活跃的变量
  - 一般为出口活跃，如果不确定则假设所有非临时变量都出口活跃
- 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

原三地址码

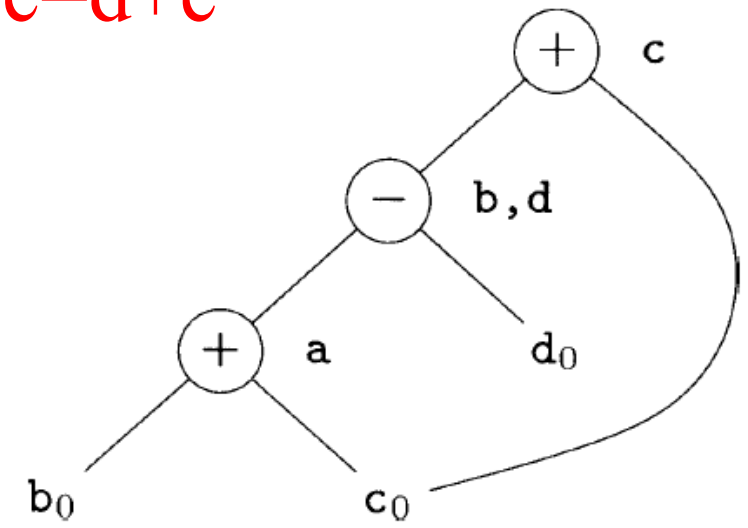
a=b+c

b=a-d

c=b+c

d=a-d

- 根据DAG构造是结点产生的顺序
- a=b+c
- d=a-d
- b=d
- c=d+c



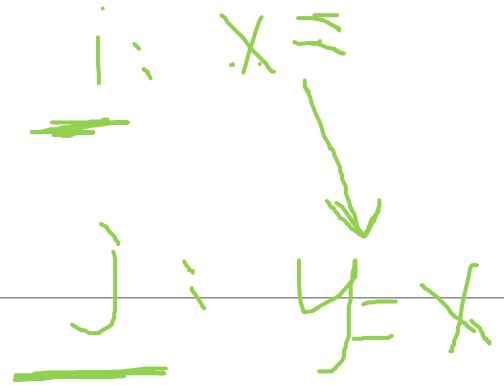
重组的规则

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。

如果两个指令之间可能相互影响，那么它们的顺序就不应该改变。

下次引用信息



变量值的使用

- 如果三地址语句i对x赋值、三地址语句j的运算分量含x，且从i到j有一条路径，且路径上无对x的重新赋值，则称j使用了语句i计算得到的x值，称x在语句i处活跃。

下次引用和活跃信息可用于代码生成

- 如果x在i处不活跃，且x占用了一个寄存器，则该寄存器在i后可用于其它目的。