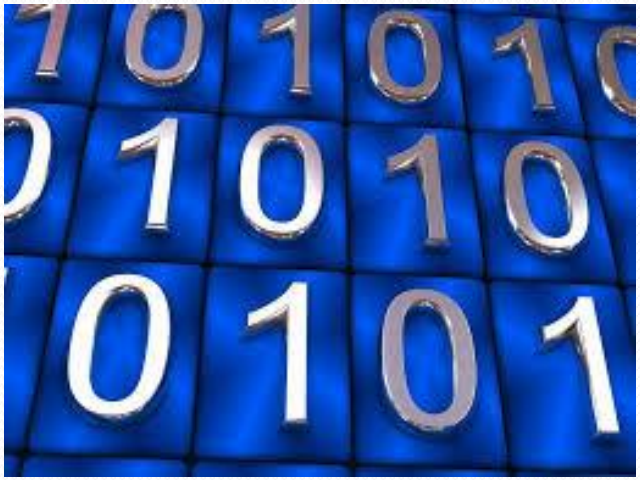
第八章　代码生成

南京大学 戴新宇

Overview



- if(x<100 || x>200 && x!=y) x=0;

```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:     x = 0
L1:
```

0000,0000,000000010000
0000,0001,000000000001
0001,0001,000000010000
0001,0001,000000000001

代码生成器概述

```
while (i<10){
    a = a + i;
    i++;
}
```

```
L1: ifFalse i < 10 goto L2
    t1 = a + i
    a = t1
    t2 = i + 1
    i = t2
    goto L1
L2:
```

```
LBB0_1:
    .loc    1 17 5
    cmpl    $10, -20(%rbp)
    jge LBB0_3
## BB#2:
    .loc    1 18 9
Ltmp6:
    movl    -24(%rbp), %eax
    addl    -20(%rbp), %eax
    movl    %eax, -24(%rbp)
    .loc    1 19 9
    movl    -20(%rbp), %eax
    addl    $1, %eax
    movl    %eax, -20(%rbp)
    .loc    1 20 5
    jmp LBB0_1
```

```
0000,0000,000000010000
0000,0001,000000000001
0001,0001,000000010000
0001,0001,000000000001
```

例如执行指令：MOV A,#0E0H，其机器码为“74H E0H”，该指令的功能是把操作数E0H送入累加器，

代码生成器概述

图 8-1　代码生成器的位置

- 根据中间表示生成目标机代码
- 代码生成器之前可能有一个优化组件
- 设计目标：
  - 保证源程序的语义
  - 充分利用目标机器上的可用资源，能够高效运行
  - 生成相对较优代码
- 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

本章内容

- 任务描述
- 目标机模型
- 静态/栈式数据区分配
- 基本块优化
- 基本块相关代码生成算法
  - 寄存器分配

代码生成器的主要任务

- 输入中间代码(IR)：
  - 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、...
- 输出目标机器指令：
  - RISC、CISC；
  - 汇编语言

代码生成器设计

- 需要考虑的问题
  - 代码生成器的输入
    - 由前端生成的源语言的中间表示和符号表信息
    - 中间表示形式，在本书中主要是三地址代码、DAG等
  - 目标程序
    - 目标机器的指令集体系结构
    - RISC(精简指令集计算机)，CISC(复杂指令集计算机)，基于堆栈的结构
    - 本章中，采用一个非常简单的类RISC计算机作为目标机，加入一些类CISC的寻址方式。为增加可读性，把汇编代码作为目标语言。

代码生成器设计（续）

- 需要考虑的问题（续）
  - 指令选择
    - 指令集体系结构中本身的特性
    - 生成代码的质量，要考虑到目标代码的效率

```
a = a + 1
```

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

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

```
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
```

```
INC a
```

目标机器模型

- 本章中
  - 三地址机器模型
  - 按照字节寻址
  - n个通用寄存器
  - 指令格式：一个运算符，一个目标地址，一个源运算分量的列表。
  - 一个有限个指令的集合

指令集

- 加载运算：LD *dst, addr*　　LD *r1,r2*
- 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*　　BLTZ *r,L*

指令中的寻址模式

- 变量名x，指向x的内存位置
- 带有下标的形如a(r)的地址，a是一个变量，r是一个寄存器。a(r)的内存位置:a的左值加上存放在寄存器r中的值。
  - LD R1, a(R2)表示R1=contents(a+contents(R2))。
  - contents(x)表示x所代表的寄存器或内存中存放的值。
- constant(r)，寄存器r中的值加上前面的常数
  - LD R1, 100(R2)表示R1=contents(100+contents(R2))
- 两种间接寻址模式
  - *r表示r的内容所表示的位置上存放的位置中的值
  - *100(r)。LD R1, *100(R2)表示 R1=contents(contents(100+content(R2)))
- 直接常数，在常数前面加上#。 LD R1,#100 ADD R1,R1,#100

目标机指令序列示例

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
```

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
```

x = *p

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

目标机指令序列示例（续）

```
*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
```

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
  - 栈式分配
- 程序逻辑地址空间的划分
  - 静态代码区
  - Static区
  - Heap区
  - Stack区
- 过程调用相关的目标代码：静态分配
  - 过程调用相关的三地址语句 call *callee*, return, halt, action
  - call *callee*对应的目标指令实现（静态分配）

    ST callee.staticArea, #here+20　　//把返回地址保存到callee的活动记录的开始处

    BR callee.codeArea　　// 把控制传递给被调用过程callee的目标代码上
  - Return指令

    BR **callee.staticArea*　//控制转向保存在callee的活动记录开始位置的地址
  - Halt指令，没有调用者的第一个过程的最后一个指令是halt，把控制返回给操作系统。

过程调用示例

```
                    // c 的代码
action1
call p

action2
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:                    // 返回地址
368:                    // p 的局部数据
```

图 8-4　静态分配的目标代码

过程调用相关的栈式分配

- 在保存活动记录时，使用相对地址
- 在寄存器SP中存放一个指向栈顶的活动记录的开始处的指针。活动记录中的其他信息可以通过相对于SP值的偏移量来访问。
- 发生过程调用时，调用过程增加SP值，并把控制转移至被调用过程。返回时，减少SP的值，从而释放被调用过程的活动记录。
- 第一个过程的代码把SP设置成内存中栈区的开始位置，完成对栈的初始化。

```
LD    SP, #stackStart          // 初始化栈
code for the first procedure
HALT                           // 结束执行
```

过程调用相关的栈分配（续）

- 一个过程调用指令序列（调用者）增加SP的值，保存返回地址，并把控制传递给被调用者。

```
ADD  SP, SP, #caller.recordSize // 增加栈指针， #caller.recordSize表示一个活动记录的大小
ST  o(SP), #here+16  // 保存返回地址， 返回地址是BR之后的指令的地址
BR  callee.codeArea  // 转移到被调用过程
```

- 返回指令序列包括两个部分
  - 被调用者把控制传递给返回地址 BR *o(SP)
  - 调用者把SP恢复为以前的值 SUB SP,SP,#caller.recordSize

过程调用栈

```
                    // m 的代码
action1
call q
action2
halt
                    // p的代码
action3
return
                    // q 的代码
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                 //压入返回地址
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                     //压入返回地址
336: BR 200                             //调用p
344: SUB SP, SP, #qsize
352: ACTION5
372: ADD SP, SP, #qsize
380: ST 0(SP), #396                     //压入返回地址
388: BR 300                             //调用q
396: SUB SP, SP, #qsize
404: ACTION6
424: ADD SP, SP, #qsize
432: ST 0(SP), #440                     //压入返回地址
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

- 考虑名字是指向符号表条目中该名字的指针。
- 语句x=0，x在符号表中的offset是12
  - 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

- 为了更好的分配寄存器和完成指令选择，按照如下方法组织中间代码：
  - 把中间代码划为基本块。每个基本块是满足下列条件的最大的连续三地址指令序列
    - 控制流只能从基本块中的第一个指令进入该块。没有跳转到基本块中间的转移指令
    - 除了基本块的最后一个指令，控制流在离开基本块之前不会停止或者跳转。
  - 基本块构成了流图中的结点。流图的边指明了哪些基本块可能紧随一个基本块之后运行。

基本块的划分

- 输入：一个三地址指令序列
- 输出：输入序列对应的一个基本块列表，其中每个指令恰好被分配给一个基本块。
- 方法：
  - 确定基本块的首指令
    - 中间代码的第一个三地址指令是一个首指令
    - 任意一个条件或无条件转移指令的目标指令是一个首指令
    - 紧跟在一个条件或无条件转移指令之后的指令是一个首指令
  - 每个首指令对应的基本块包括了从它自己开始，直到下一个首指令（不含）或者结尾指令之间的所有指令。

划分示例

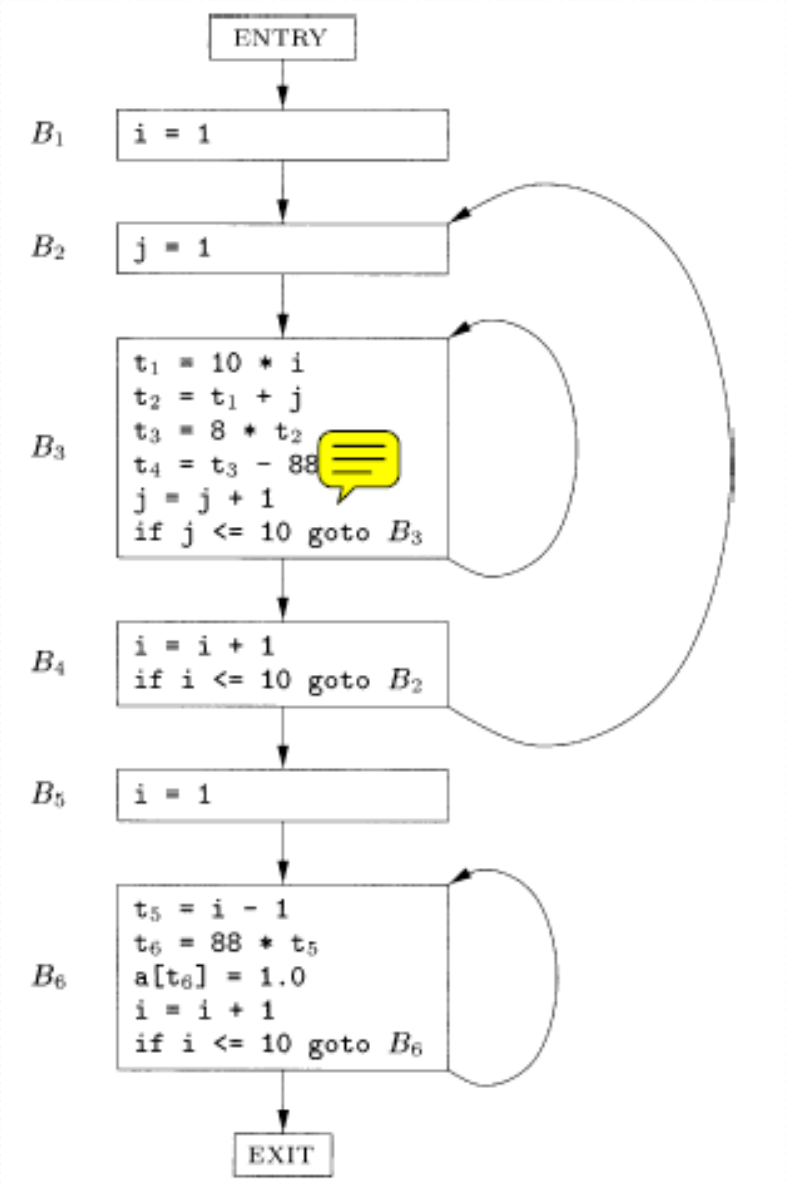
```
for i from 1 to 10 do
        for j from 1 to 10 do
                a[i, j] = 0.0;
for i from 1 to 10 do
        a[i, i] = 1.0;
```

```
 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

- 可以用来表示基本块之间的控制流
- 流图的结点是基本块，从基本块B到基本块C之间有一条边当且仅当基本块C的第一个指令可能紧跟在B的最后一条指令之后执行。
  - 有一个从B的结尾跳转到C的开头的条件或无条件跳转语句
  - 按照原来的三地址语句序列中的顺序，C紧跟在B之后，且B的结尾不存在跳转语句。
- B是C的前驱，C是B的后继
- 增加一个入口和出口。入口到流图的第一个基本块有一条边。从任何可能是程序的最后执行指令的基本块到出口有一条边。

流图示例


ENTRY
B_1
i = 1
B_2
j = 1
B_3
t_1 = 10 * i
t_2 = t_1 + j
t_3 = 8 * t_2
t_4 = t_3 - 88
j = j + 1
if j <= 10 goto B_3
B_4
i = i + 1
if i <= 10 goto B_2
B_5
i = 1
B_6
t_5 = i - 1
t_6 = 88 * t_5
a[t_6] = 1.0
i = i + 1
if i <= 10 goto B_6
EXIT

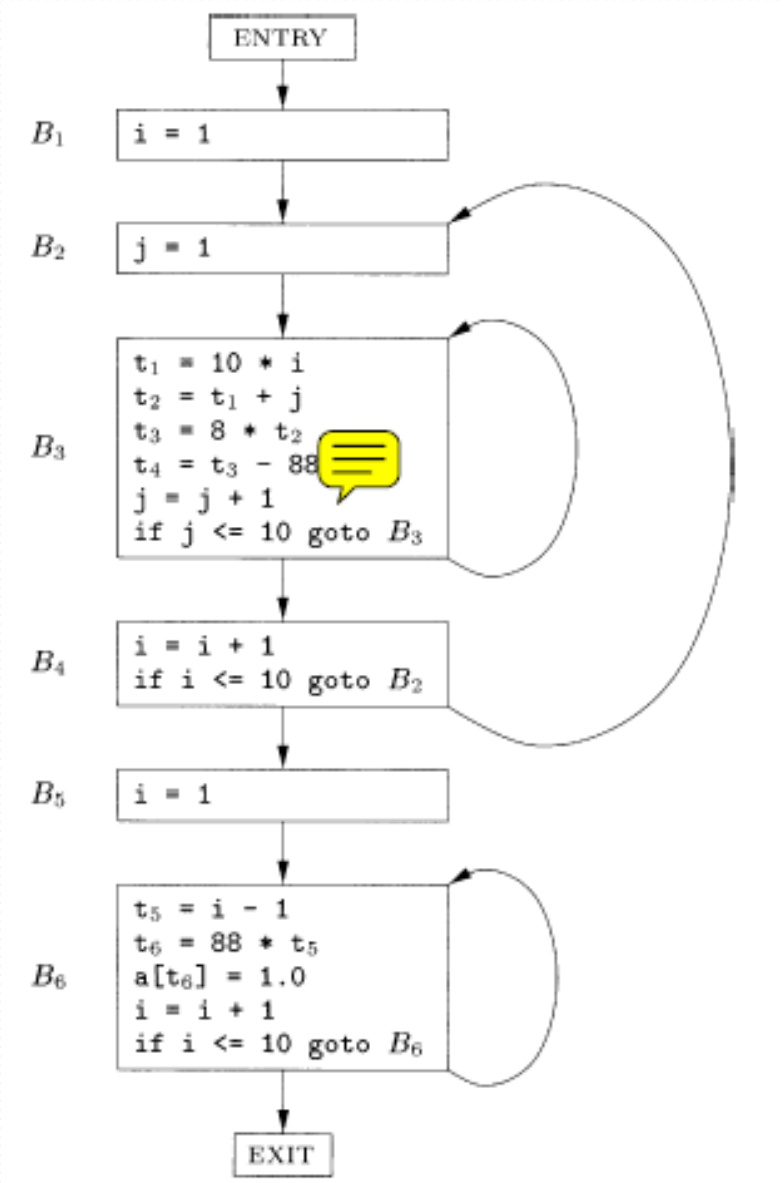
流图的表示方式

- 把到达指令的标号或序号替换为到达基本块的跳转
- 这样在改变某些指令的时候，可以不修改跳转指令的目标

循环

- 通过流图识别“循环”
- 若满足以下条件，则流图中的一个结点集合L是一个循环。
  - L中有一个循环入口结点，它是唯一的前驱可能在循环外的结点。从整个流图的入口结点开始到L中的任何结点的路径都必然经过循环入口结点。
  - L的每个结点都有一个到达L的入口结点的非空路径，并且该路径都在L中。

循环示例


ENTRY
B_1
i = 1
B_2
j = 1
B_3
t_1 = 10 * i
t_2 = t_1 + j
t_3 = 8 * t_2
t_4 = t_3 - 88
j = j + 1
if j <= 10 goto B_3
B_4
i = i + 1
if i <= 10 goto B_2
B_5
i = 1
B_6
t_5 = i - 1
t_6 = 88 * t_5
a[t_6] = 1.0
i = i + 1
if i <= 10 goto B_6
EXIT

基本块的优化

- 基本块的优化又称为局部优化
- 全局优化是一个程序的优化，需要考虑基本块之间的数据分析。
- DAG图可以显式地反映变量及其值对其他变量的依赖关系
- 构造方法
  - 基本块中出现的每个变量有一个对应的DAG结点表示其初始值
  - 基本块每个语句s有一个节点N，N的子节点是基本块中的其它语句对应的结点。这些节点对应的是最后一个对s中的运算分量进行定值的语句
- 结点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

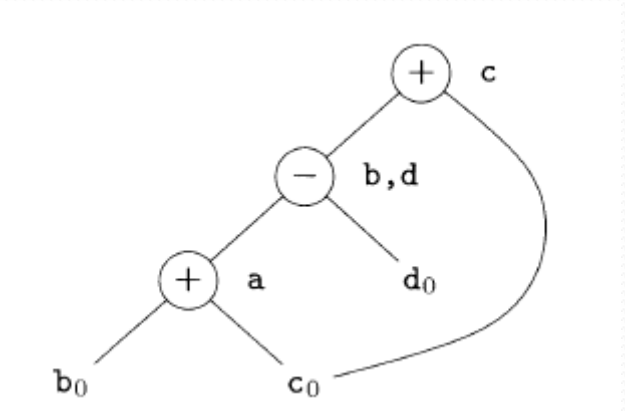
基本块的DAG表示

- 基本块可以用DAG来表示，可以帮助我们
  - 消除局部公共子表达式
  - 消除死代码
  - 语句重排序
  - 对运算分量进行符合代数规则重排序

DAG中的局部公共子表达式

- 所谓公共子表达式就是重复计算一个已经计算得到的值的指令。
- 构造过程中，我们就会检测是否存在具有同样运算符和同样子节点的节点。

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




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

消除死代码

- 删除没有附加活跃变量且没有父节点的结点
- 假设c和e不是活跃变量

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

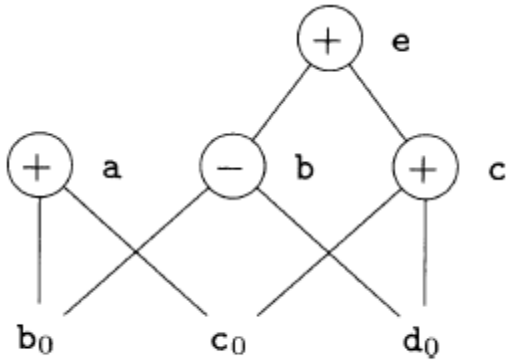


图 8-13　例 8.11 中的基本块的 DAG

代数恒等式的使用

- 消除计算步骤

$$x+0=0+x=x \qquad x-0=x$$
$$x\times 1=1\times x=x \qquad x/1=x$$

- 强度消减

代价较高的		代价较低的
x^2	=	$x\times x$
$2\times x$	=	$x+x$
$x/2$	=	$x\times 0.5$

- 常量合并，将常量表达式替换为求出的值
- 使用代数转换规则，比如交换律和结合律，可以发现公共子表达式。
  - x*y=y*x
  - a=b+c, e=c+d+b
- 编译器的设计者应该仔细阅读语言手册，以免其不一定遵守数学上的代数恒等式。

数组引用的DAG表示

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

- x和z不能当成公共子表达式
- 在DAG图中的数组访问表示方法
  - 从一个数组取值并赋给其他变量的运算（x=a[i]），用运算符为=[]的结点表示。这个结点的左右子节点是数组初始值a_o和下标i。变量x是这个结点的标号之一。
  - 对数组的赋值(比如a[j]=y)用一个运算符[]=来表示。这个结点的三个子节点分别表示a_o、j和y。没有其它变量用这个结点标号。此结点创建后，当前已经建立的、其值依赖于的a_o所有结点被杀死。

数组DAG表示

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

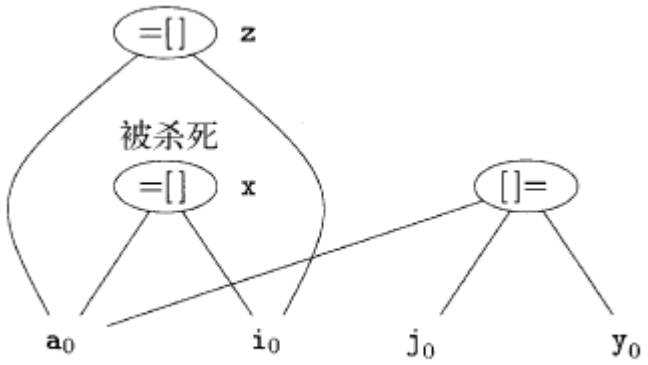
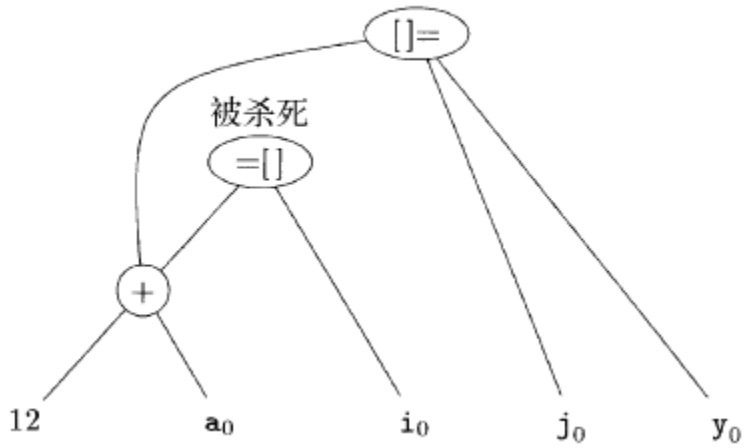


图 8-14　一个数组赋值序列的 DAG

```
b = 12 + a
x = b[i]
b[j] = y
```



一个简单的代码生成器

- 为基本块生成代码。依次考虑各个三地址指令，并跟踪记录哪个值存放在哪个寄存器中，从而可以避免生成不必要的加载和保存指令
- 如何最大限度的利用寄存器？
- 通常，寄存器的使用方法：
  - 各个运算分量必须存放在寄存器中
  - 寄存器适合存放临时变量（只在基本块中使用的变量的值）
  - 寄存器用来存放在一个基本块中计算而在另一个基本块中使用的值。例如循环的下标。
  - 寄存器用来帮助进行运行时刻存储管理。如运行时刻栈的指针。
- 上述寄存器使用有竞争关系

寄存器描述符和地址描述符

- 用来记录跟踪程序变量的值所在的位置
- 寄存器描述符：记录寄存器当前存放了哪个变量的值。一个寄存器可以存放一个或者多个变量的值。
- 地址描述符：记录每个名字的当前值的存放处所，可以是寄存器，也可以是内存地址，或者它们的集合（当值被赋值传输的时候）。

- LD *reg*, *mem*
- ST *mem*, *reg*
- OP *reg*, *reg*, *reg*

代码生成算法

- 本书假定一组寄存器存放基本块内使用的值。每个运算符有唯一的运算指令，且运算指令对存放在寄存器中的运算分量进行运算。
- 该算法中的重要函数getreg(i), 这个函数为每个与三地址指令i有关的内存位置选择寄存器
- 对于每个i: x=y op z指令。
  - 调用getreg(i)，该函数为x、y、z选择寄存器Rx Ry Rz
  - 如果y不在Ry（查看Ry的寄存器描述符）中，那么生成指令“LD Ry,y’”，其中y’是存放y的内存地址之一（由y的地址描述符得到）。
  - 类似的处理z
  - 生成指令“op Rx, Ry, Rz”

代码生成算法(续)

- 对于赋值语句i: x=y
  - 调用getreg(i)
  - 如果y不在Ry（查看Ry的寄存器描述符）中，那么生成指令“LD Ry,y’”，其中y’是存放y的内存地址之一（由y的地址描述符得到）。
  - 修改Ry的寄存器描述符，表明Ry中也存放了x的值。
- 基本块的结尾，为每个活跃变量x生成指令“ST x R”，其中R是存放x值的寄存器。

寄存器和地址描述符管理

- 在生成加载、保存和其他指令时，还需要更新寄存器和地址描述符。
- 修改规则
  - 对于指令“LD R, x”
    - 修改寄存器R描述符，使之只包含x
    - 修改x的地址描述符，把寄存器R作为新增位置加入其中
    - 从不同于x的其他变量的地址描述符中删除R
  - 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
  - 对于x=y op z的指令”OP Rx,Ry,Rz”
    - 修改Rx的寄存器描述符，使之只包含x
    - 修改x的地址描述符，使之只包含位置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
  - 对于赋值语句x=y。除了可能的y加载指令处理之外，
    - 把x加入到Ry的寄存器描述符中
    - 修改x的地址描述符，使得它只包含唯一的位置Ry

代码生成示例

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

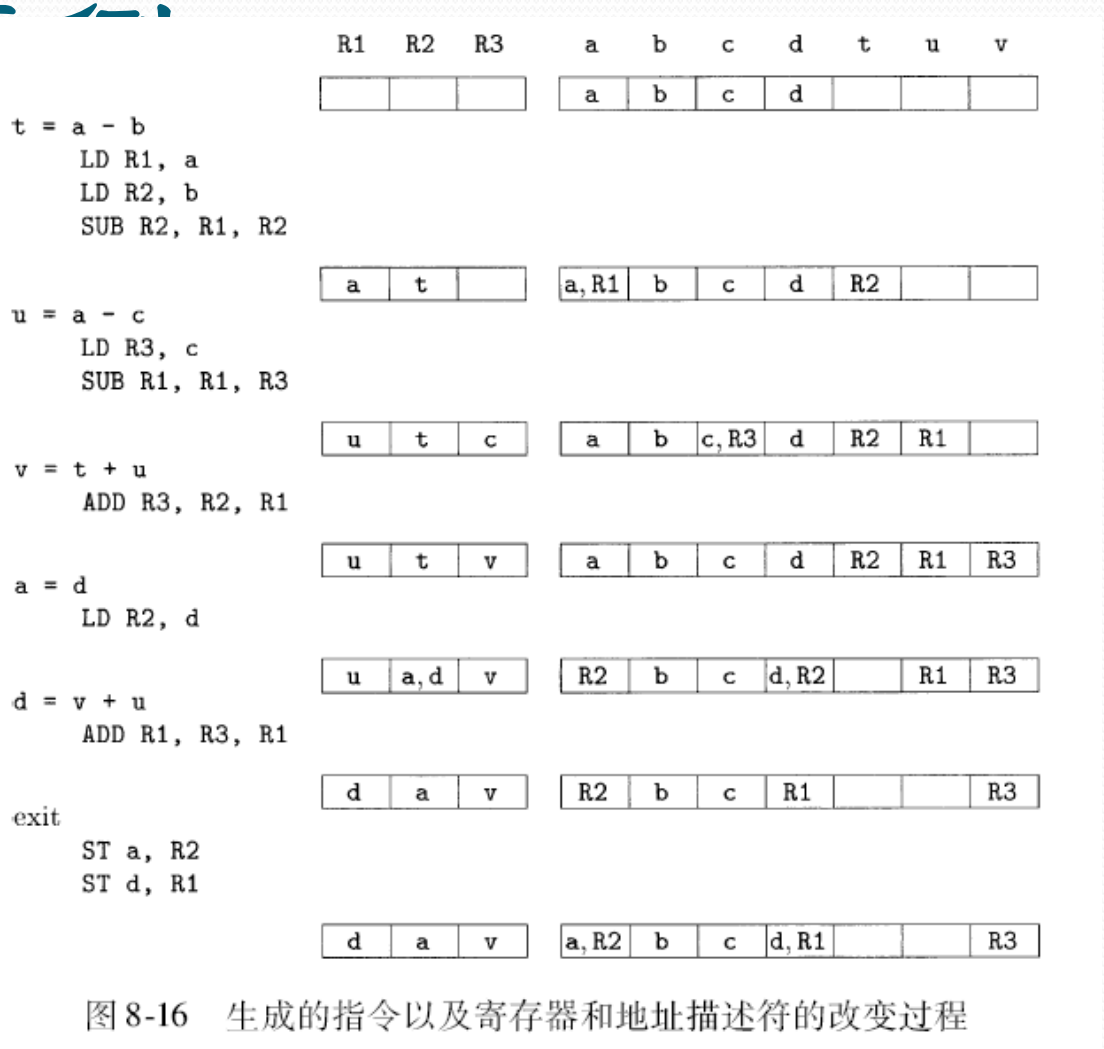


图 8-16　生成的指令以及寄存器和地址描述符的改变过程

getreg的设计

Getreg(x= y op z)：Ry or Rz

1. 如果y当前就在一个寄存器中，则选择这个已经包含了y的寄存器作为Ry。
2. 如果y不在寄存器中，选择一个空寄存器作为Ry。
3. 如果不在寄存器中，又没有空寄存器可用，则需要复用一个寄存器，该寄存器描述符说明v是保存在其中的变量：
   - 如果v的地址描述符说v还在其他地方，ok
   - 如果v是x, x!=z，ok
   - 如果 v不用（not alive), ok
   - Spill: ST v, R
4. 对每个寄存器描述符中的v，重复以上操作。R的代价是spill的次数。从所有可能的R中选择代价最小的R

getreg的设计

Getreg(x= y op z)： Rx

- 选择只存放x的值的寄存器
- 如果y或z之后不会被使用，也可以选择Ry或Rz
- 否则，可以按照选择Ry和Rz的2、3、4步骤选择

Getreg(x= y)： Rx

- 先选择Ry，然后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** **LD R1, b** **SUB R1, R0,R1**	**R0含a** **R1含t_1**	**a in R0** **t_1 in R1**
$t_2 := a - c$	**ST t1, R1** **LD R1, c** **SUB R0, R0, R1**	**R1含c** **R0含t_2**	**c in R1** **t_2 in R0**
$t_3 := t_1 + t_2$	**LD R1,t1** **ADD R1,R1,R0**	**R1含t_3**	**t_3 in R1** **t_2 in R0**
$d := t_3 + t_2$	**ADD R1,R1,R0**	**R1含d** **R0含t_2**	**d in R1** **t_2 in R0**
	ST d, R1		**d在R1和内存中**

寄存器的分配和指派

- 寄存器分配：确定在程序的每个点上，哪些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
- 最简单的分配和指派方法：把目标程序中的特定值分配给特定的寄存器。比如把基地址指派给一组寄存器，算术运算则使用另一组寄存器，栈顶指针指派给一个固定的寄存器。
- 优点：设计简单
- 缺点：使用效率低

第8章总结

- ……