

编译原理

第九章 目标代码生成

方徽星

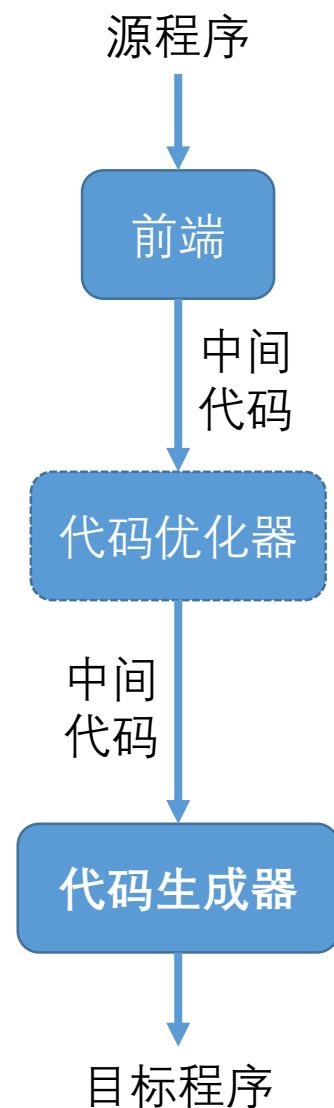
扬州大学 信息工程学院(505)

fanghuixing@yzu.edu.cn

2018年春季学期

本章主要内容

1. 代码生成器设计中的问题
2. 目标语言
3. 目标代码中的地址
4. 基本块和流图(Flow Graph)
5. 基本块的优化
6. 一个简单的代码生成器
7. 窥孔优化(Peephole Optimization)



第一节 代码生成器设计中的问题

1 代码生成器设计中的问题

- 代码生成器的输入

- 由前端生成的源程序的**中间表示形式** + **符号表信息**
- **用来确定**数据对象的运行时刻地址

IR { 三地址表示方式：四元式、三元式、间接三元式
虚拟机表示方式：字节代码、堆栈代码
线性表示方式：后缀表示
图形表示方式：抽象语法树、**DAG**

IR: Intermediate Representation

1 代码生成器设计中的问题

- 目标程序

- 构造代码生成器会受目标机器的指令集体系结构影响

常见的目标机
体系结构

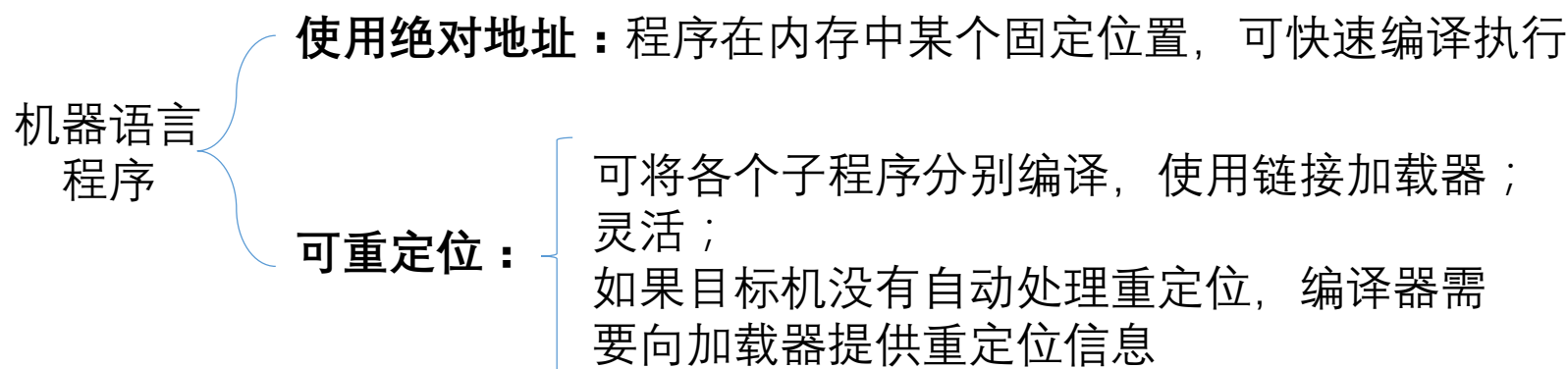
{ RISC：精简指令集计算机
CISC：复杂指令集计算机
基于堆栈的结构

RISC	CISC	基于堆栈的结构
大量的通用寄存器 三地址指令 寻址方式简单 指令集体系结构简单	较少的寄存器 两地址指令 寻址方式复杂多样 寄存器种类多 可变长度的指令 具有副作用的指令	运算通过运算分量压栈 再对栈顶运算分量进行 运算来完成 栈顶元素常在寄存器中 操作限制多

1 代码生成器设计中的问题

- 目标程序

- 构造代码生成器会受目标机器的指令集体系结构影响



输出汇编程序可以简化代码生成过程
生成符号指令，使用宏机制帮助生成代码

1 代码生成器设计中的问题

- 指令选择
 - 从IR程序到目标机代码序列的映射复杂性由如下因素决定
 - **IR的层次**
 - 指令集体系结构本身特性
 - 期望代码质量

IR是高层次的	IR是低层次的
使用代码模板把每个IR语句翻译成机器指令序列，会产生质量不佳的代码	代码生成器可以使用底层次信息来生成更加高效的代码序列

1 代码生成器设计中的问题

- 指令选择
 - 从IR程序到目标机代码序列的映射复杂性由如下因素决定
 - IR的层次
 - 指令集体系统结构本身特性
 - 期望代码质量

如果目标机没有统一方式来支持每种数据类型
则总体规则的每个例外都需要进行特别处理

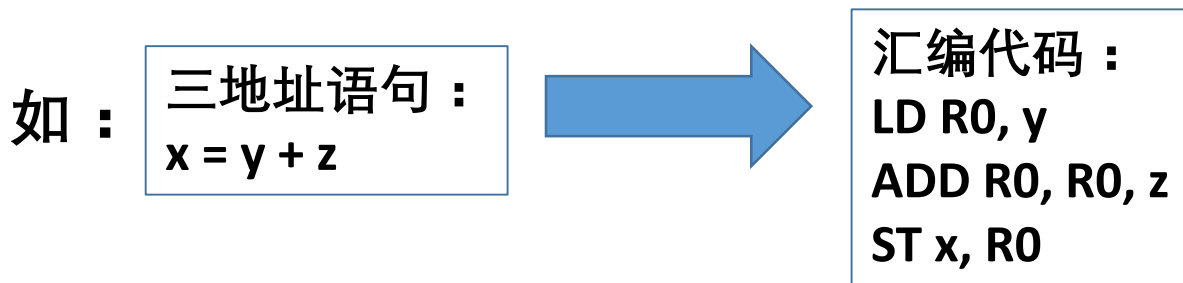
1 代码生成器设计中的问题

- 指令选择

- 从IR程序到目标机代码序列的映射复杂性由如下因素决定

- IR的层次
- 指令集体系结构本身特性
- **期望代码质量**

如果不考虑目标程序的效率，则指令选择是简单的
对于每一种三地址语句，可以生成一个代码骨架



但常常会产生冗余的加载和存储运算

1 代码生成器设计中的问题

- 指令选择

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



```
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
```

- ✓ 第四个语句是冗余的
- ✓ 如果a以后不再被使用，则第三个与语句也是冗余的

1 代码生成器设计中的问题

- 指令选择

- 一个给定的IR程序可以用多种不同的目标代码序列来实现，不同的实现代码差异可能显著
- 如果目标机有“加一”指令(**INC**)，三地址语句 $a = a + 1$

INC a

vs

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

1 代码生成器设计中的问题

- 寄存器分配
 - 资源有限：通常没有足够的寄存器来存放所有的值
 - 有效利用很重要
 - 寄存器分配：选择一组变量，待存放入寄存器
 - 寄存器指派：选择一个寄存器安放某个变量
- 求值顺序
 - 计算执行的顺序会影响目标代码的效率
 - 某些计算顺序对用于存放中间结果的寄存器需求更少
 - 一般情况下，找到最好的顺序是一个NP完全问题

第二节 目标语言

2 目标语言

- 三地址机器目标机模型
 - 内存按照字节寻址
 - 具有 n 个通用寄存器 R_0, \dots, R_{n-1}
- 可用指令
 - 加载操作：指令LD dst, addr把位置addr上的值加载到位置dst,
 - 该指令表示赋值 $\text{dst} = \text{addr}$
 - LD r, x 把位置 x 中的值加载到寄存器 r 中
 - LD r_1, r_2 把寄存器 r_2 的内容复制到寄存器 r_1
 - 保存操作：指令ST x, r 把寄存器 r 中的值保存到位置 x
 - 该指令表示赋值 $x = r$

2 目标语言

- 可用指令

- 计算操作：形如 $OP\ dst, src_1, src_2$ ，其中

- OP 是运算符，如：ADD、SUB

- dst 、 src_1 和 src_2 是位置

- 把运算作用在位置 src_1 和 src_2 中的值上，然后把结果放在位置 dst 中

- SUB r_1, r_2, r_3 计算了 $r_1 = r_2 - r_3$

- 单目运算没有 src_2

2 目标语言

- 可用指令

- 无条件跳转：

- 指令BR L 使得控制流转向标号为 L 的机器指令

- 条件跳转：Bcond r, L ，其中

- r 是一个寄存器，

- L 是一个标号

- cond代表了对寄存器 r 中的值所做的测试

- 如：当寄存器 r 中的值小于0时，BLTZ r, L 使得控制流跳转到标号 L ；否则，控制流传递到下一个机器指令

2 目标语言

- 寻址模式，指令中的位置可以是：
 - 变量名 x ，指向分配给 x 的内存位置，即 x 的左值
 - 带有下标的形如 $a(r)$ 的地址，其中
 - a 是一个变量，
 - r 是一个寄存器
 - $a(r)$ 所表示的内存地址 = a 的左值 + r 中的值
 - **LD R1, a(R2)**的效果是
$$R1 = \text{contents}(a + \text{contents}(R2)),$$
contents(x)表示 x 所代表的寄存器或内存位置中存放的内容

2 目标语言

- 寻址模式，指令中的位置可以是：

- 以寄存器为下标的整数

- `LD R1, 100(R2)`的效果是

$$R1 = \text{contents}(100 + \text{contents}(R2))$$

✓ 首先计算寄存器R2中的值加上100得到的和，

✓ 然后把这个和所指向的位置中的值加载到R1中

- 直接常数寻址

- `LD R1, #100`把整数100加载到R1中，

- `ADD R1, R1, #100`把100加到寄存器R1中去

2 目标语言

- 寻址模式，指令中的位置可以是：
 - $*r$ 表示在寄存器 r 的内容所表示的位置上存放的内存位置
 - $*100(r)$ 表示 r 中内容加上100的和所代表的位置上的内容所代表的位置
 - **LD R1, *100(R2)**的效果是把**R1**设置为
contents(contents(100 + contents(R2)))

2 目标语言

- 三地址代码 $x = y - z$ 可以使用下面的机器指令序列实现：

LD R1, y	//R1 = y
LD R2, z	//R2 = z
SUB R1, R1, R2	//R1 = R1 - R2
ST x, R1	//x = R1

可能的优化：

- 如果y或z已经存放在寄存器中，则可以避免相应的LD步骤
- 如果x的值被使用时都存放在寄存器中，且之后不会再被用到，则不需要把值保存回x

2 目标语言

- 假设a是数组，下标从0开始，每个元素占8字节
如下指令序列实现三地址语句 **$b = a[i]$** :

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

2 目标语言

- 三地址指令 $a[j] = c$ 可以实现为:

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

2 目标语言

- 三地址指令 $x = *p$ 可以实现为:

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

- 三地址指令 $*p = x$ 可以实现为:

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

2 目标语言

- 带条件跳转的三地址指令 **if $x < y$ goto L**可以实现为:

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是从标号为L的三地址指令所产生的机器指令序列中的第一个指令的标号

2 目标语言

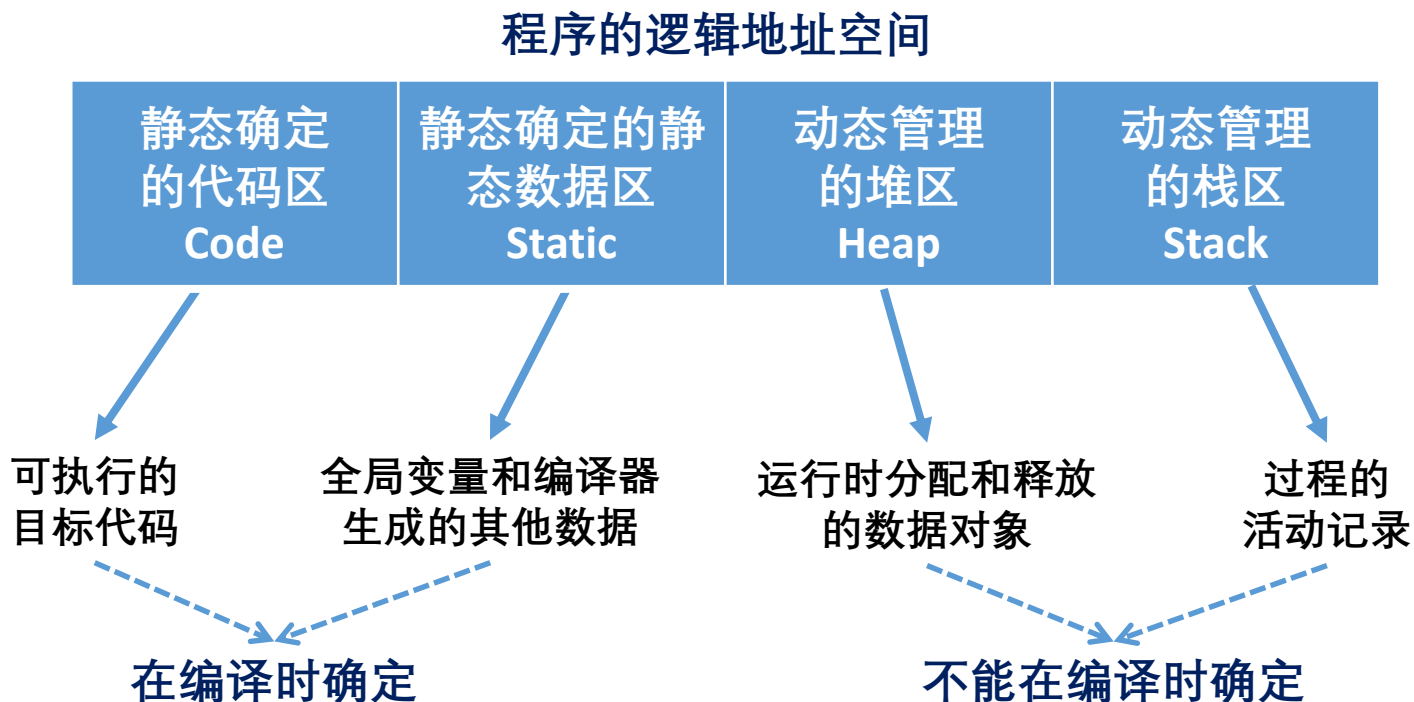
- **指令的代价**

- 指令的代价 = 1 + 运算分量寻址模式相关代价
- 寄存器寻址模式具有附件代价为0
- 内存位置或常数的寻址方式的附件代价为1
- LD R0, R1把寄存器R1中的内容复制到寄存器R0中，指令代价为**1**
- LD R0, M把内存位置M中的内容加载到寄存器R0中，指令代价为**2**
- LD R1, *100(R2)把值
 contents(contents(100+contents(R2)))
加载到寄存器R1中，代价为**3**

第三节 目标代码中的地址

3 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？



3.1 静态分配

- 假设活动记录的开始位置存放返回地址
- 关注如下三地址语句
 - call *callee*、**return**、**halt** 和 **action**



```
ST  callee.staticArea, #here+20  
BR  callee.codeArea
```

- ✓ **ST**把返回地址保存在***callee***的活动记录开始处
- ✓ **BR**把控制传递到被调用过程***callee***的目标代码上
- ✓ *staticArea*是活动记录的开始地址
- ✓ *codeArea*是指向***callee***的第一个指令的地址

3.1 静态分配

- 假设活动记录的第一位置存放返回地址
- 关注如下三地址语句
 - **call** *callee*、**return**、**halt** 和 **action**



BR **callee.staticArea*

把控制流转到保存在*calle*的活动记录开始地址上

3.1 静态分配

- 例：三地址代码及静态分配的目标代码

```
//过程c的代码  
action1  
call p  
action2  
halt  
//过程p的代码  
action3  
return
```

```
100: ACTION1 // code for c  
120: ST 364, #140 // code for action1  
132: BR 200 // save return address 140 in location 364  
140: ACTION2 // call p  
160: HALT // return to operating system  
...  
// code for p  
200: ACTION3  
220: BR *364 // return to address saved in location 364  
...  
// 300-363 hold activation record for c  
300: // return address  
304: // local data for c  
...  
// 364-451 hold activation record for p  
364: // return address  
368: // local data for p
```

3.2 栈分配

- 在保存活动记录时使用相对地址
- 在寄存器SP中维护一个指向栈顶的活动记录的开始处的指针，偏移量为正数
 - 发生**过程调用**时，**增加SP的值**，并把控制传递到被调用过程
 - 控制**返回**到调用者时，**减少SP的值**，释放被调用过程的活动记录

3.2 栈分配

- 第一个过程的代码把寄存器SP设置为内存中栈区的开始位置，完成对栈的初始化

1. LD SP, #stackStart //初始化栈

2. Code for the first procedure

3. HALT //结束执行

3.2 栈分配

- 一个过程调用指令序列增加SP的值，保存返回地址，并把控制传递到被调用过程

1. **ADD SP, SP, #caller.recordSize** **//增加栈指针**
2. **ST 0(SP), #here+16** **//保存返回地址**
3. **BR callee.codeArea** **//转移到被调用过程**

- 运算分量#caller.recordSize表示一个活动记录的大小
- 运算分量#here+16是跟随在BR之后的指令的地址

3.2 栈分配

- 返回指令序列

- 被调用过程使用如下指令把控制流传递到返回地址

BR *0(SP) //控制返回给调用者

➤ **0(SP)**是活动记录的第一个字所在的位置

➤ ***0(SP)**是存放在该位置上的返回地址

- 调用者恢复SP

SUB SP, SP, #caller.recordSize

➤ 该指令指向后，**SP**指向调用者活动记录开始处

3.2 栈分配

- 例：假设过程m、p和q的活动记录大小已经确定，分别是msize、psize和qsize

```
                                // 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
```

```
100 : LD SP, #600           //初始化栈
108 : ACTION1                //action1的代码
128 : ADD SP, SP, #msize
136 : ST 0(SP), #152        //返回地址入栈
144 : BR 300                //调用q
152 : SUB SP, SP, #msize    //恢复SP
160 : ACTION2
180 : HALT
.....
200 : ACTION3
220 : BR *0(SP)
.....
300 : ACTION4
.....
```

3.2 栈分配

- 例：假设过程m、p和q的活动记录大小已经确定，分别是msize、psize和qsize

```
                                // 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
```

```
300 : ACTION4
320 : ADD SP, SP, #qsize
328 : ST 0(SP), #344      //返回地址入栈
336 : BR 200              //调用p
344 : SUB SP, SP, #qsize
352 : ACTION5
372 : ADD SP, SP, #qsize
380 : ST 0(SP), #396      //返回地址入栈
388 : BR 300              //调用q
...
432 : ST 0(SP), #448      //返回地址入栈
440 : BR 300              //调用q
448 : SUB SP, SP, #qsize
456 : BR *0(SP)           //返回
...
600 :                     //栈区的开始处
```

第四节 基本块和流图

4 基本块和流图

- 用图表示中间代码：

- 把中间代码划分为**基本块(basic block)**，每个基本块满足如下条件的**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
 - 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机或跳转**
- 基本块形成**流图(flow graph)**的结点，结点之间的边指明可能的执行次序

4.1 基本块

- 把三地址指令序列划分为基本块（算法）：
 - **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**：
 - 首先，**确定首指令**：某个基本块的第一个指令；
 - 然后，**确定基本块**：从首指令开始，直到下一个**首指令**(不含)或者中间程序的**结尾指令**之间的所有指令

4.1 基本块

- **选择首指令的规则：**

- 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

4.1 基本块

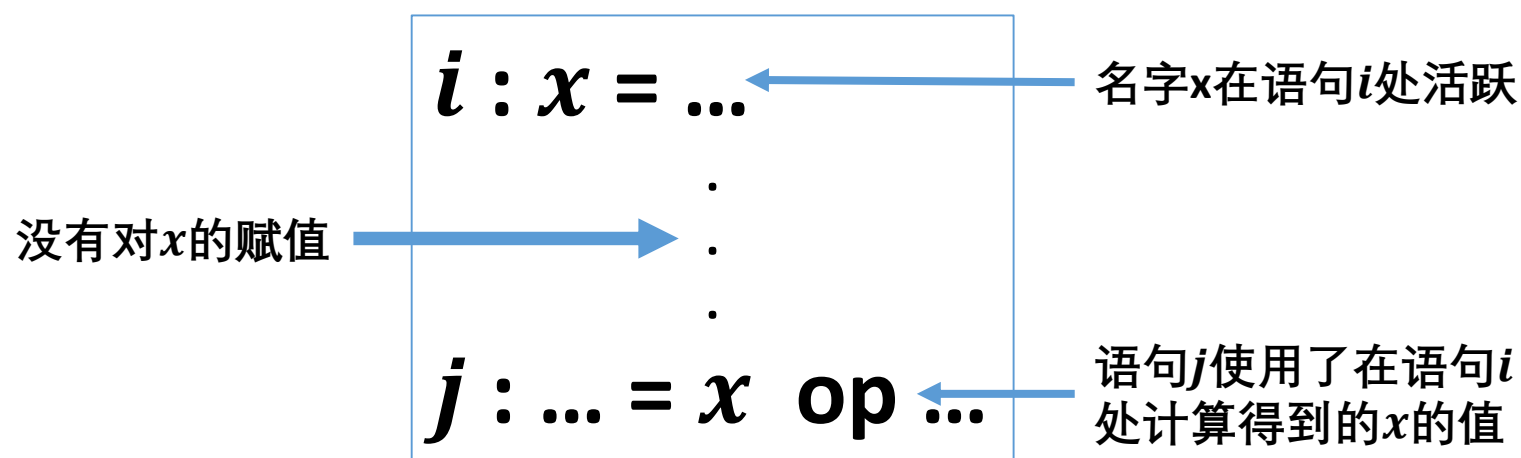
```
1)  i = 1
2)  j = 1
3)  t1 = 10 * i
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

确定首指令：

- 第一个三地址指令：
 - 指令1
- 转移目标指令：
 - 指令3
 - 指令2
 - 指令13
- 转移指令的下一条指令：
 - 指令10
 - 指令12

4.2 后续使用信息

- 名字的后继使用信息可用于优化
- 三地址语句中对名字的使用：



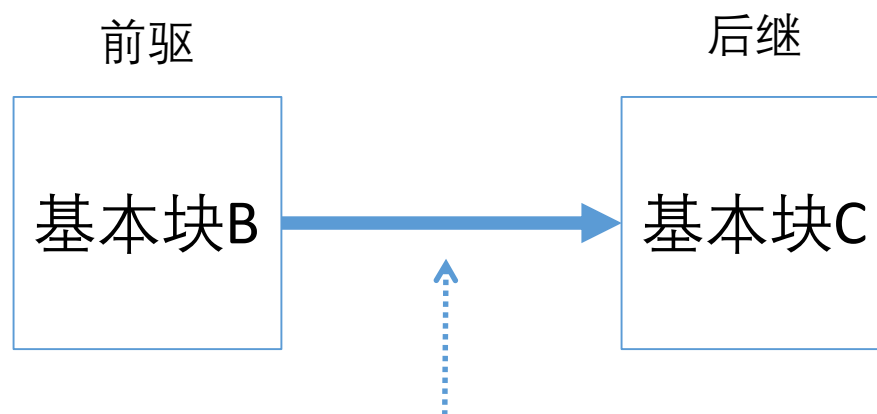
如果名字*x*的值在某点之后还要被引用，则*x*在该点是活跃的

4.2 后续使用信息

- **确定基本块中活跃与后续使用信息(算法)**

- **输入**：基本块B，假设在开始的时候符号表显示B中的所有非临时变量都是活跃的
- **输出**：对于B的每个语句 $i: x = y + z$ ，将 x 、 y 及 z 的活跃性信息及后续使用信息关联到 i
- **方法**：从B的最后一个语句开始，反向扫描到B的开始处，对于每个语句 $i: x = y + z$ ：
 1. 把符号表中 x 、 y 及 z 的当前后续使用和活跃性信息与语句 i 关联起来
 2. 在符号表中，设置 x 为“不活跃”和“无后续使用”
 3. 在符号表中，设置 y 和 z 为“活跃”，把它们的下一次使用设置为语句 i

4.3 流图



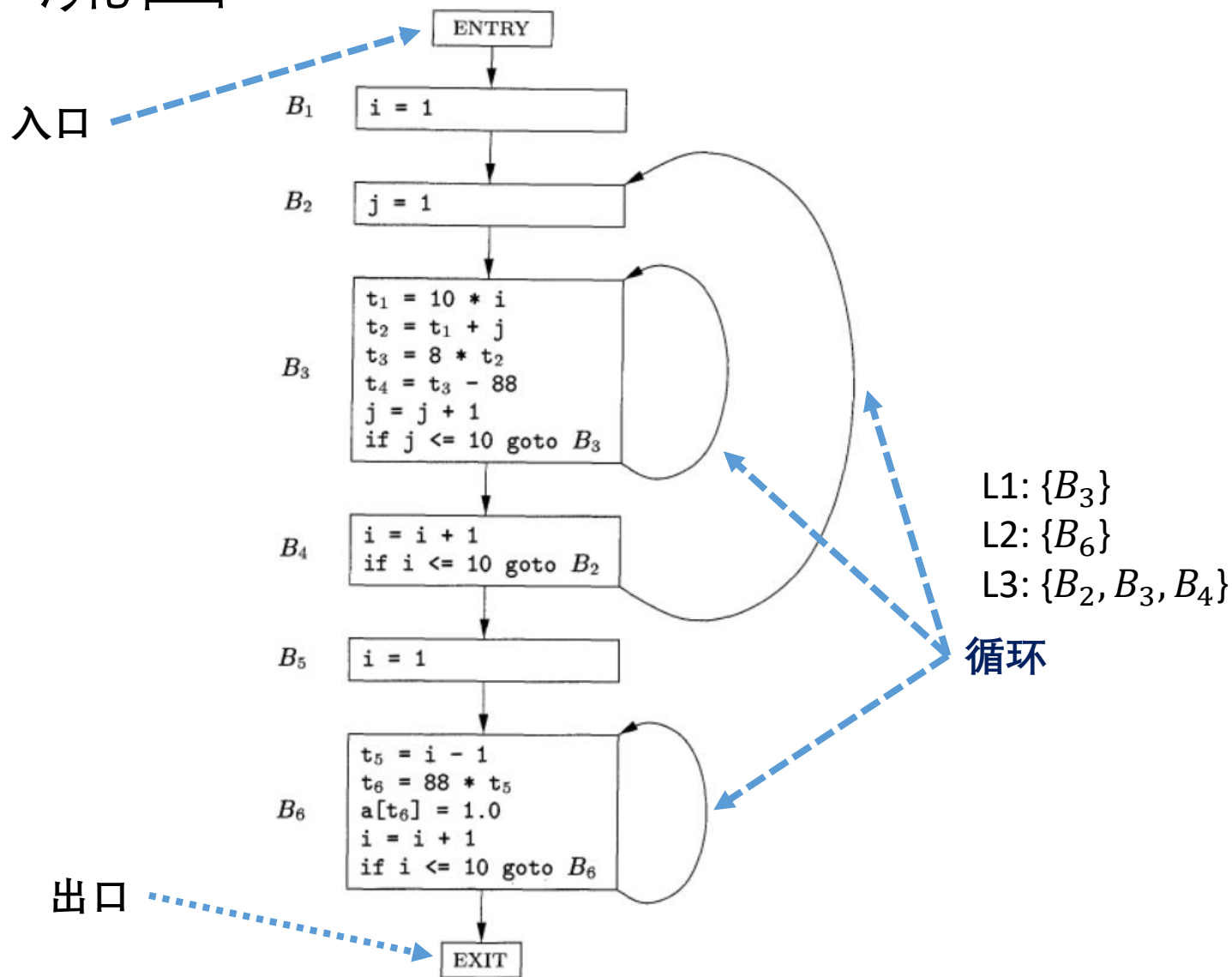
**C的第一个指令可能紧跟在B的最后一个指令之后执行，
两种情况：**

- ✓ **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吗，或者**
- ✓ **C紧跟在B之后，且B的结尾不存在无条件跳转语句**

4.3 流图

- 流图入口结点
 - 从入口到第一个可执行结点有一条边
- 流图出口结点
 - 从任何包含了可能是程序的最后执行指令的基本块到出口有一条边
 - 如果程序的最后指令不是无条件转移指令，则相应的基本块是出口结点的一个前驱
 - 任何包含跳转到程序之外的跳转指令的基本块是出口结点的前驱

4.3 流程图



4.3 流图

- 循环L的入口结点E
 - 是唯一的，前驱可能在L之外的结点
 - 从流图入口到L中任何结点必经E
 - E不是流图入口结点
 - L中每个结点都能够到达E，且路径全部在L中

第五节 基本块的优化

5.1 基本块的DAG表示

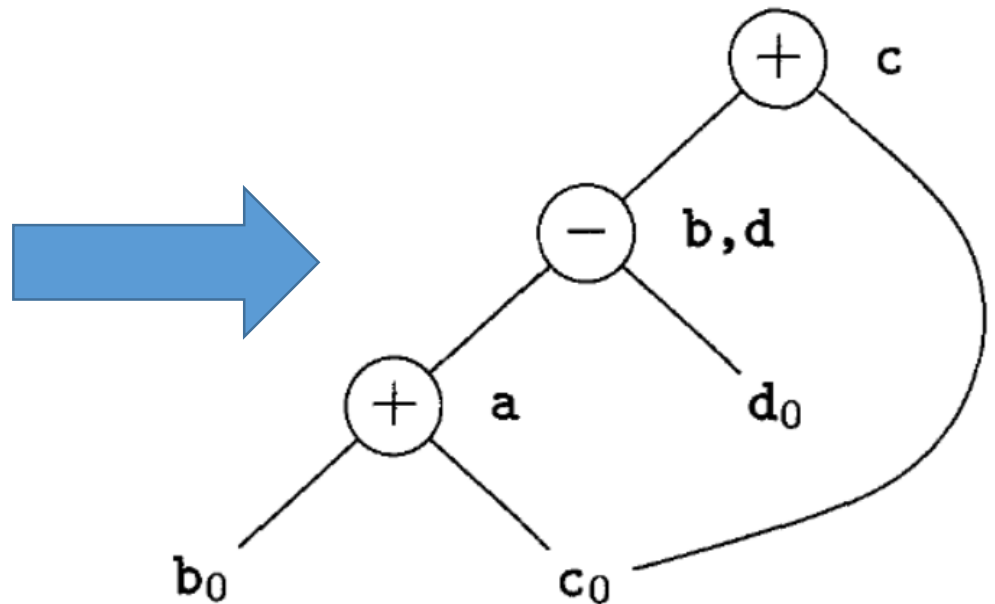
- 为基本块构造**DAG(有向无环图)**：

- 块中每个变量 v 有结点，代表初始值 v_0
- 每个语句 s 有一个结点 N ，其标号是 s 的运算符
- N 的子结点 i 对应于其他语句 t_i ，语句 t_i 在 s 之前对 s 的某个运算分量 p_i 进行定值，且是最晚的
- **N 关联一组变量 x_i ，表示语句 s 是在此基本块内最晚对 x_i 定值的语句**
- **输出结点**：相关变量在基本块出口处活跃

5.2 寻址局部公共子表达式

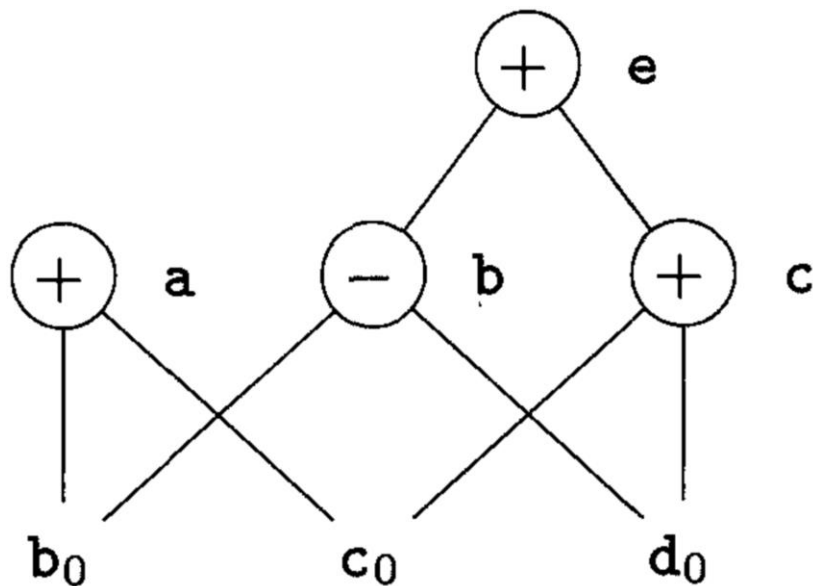
- 建立某个结点M之前，首先检查是否存在一个结点N，它和M具有相同的运算符和子结点（顺序也相同）
- 如果存在，则不需要生成新的结点，用N代表

1. $a = b + c$
2. $b = a - d$
3. $c = b + c$
4. $d = a - d$



5.3 消除死代码

- 在DAG上消除没有附加活跃变量的根结点



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

5.4 使用代数恒等式

- 代数恒等式表示基本块的另一类重要的优化方法

消除
计算
步骤

$$\begin{aligned}x + 0 &= 0 + x = x & x - 0 &= x \\x \times 1 &= 1 \times x = x & x/1 &= x\end{aligned}$$

强度
消减

$$\begin{aligned}x^2 &= x \times x \\2 \times x &= x + x \\x/2 &= x \times 0.5\end{aligned}$$

常量
合并

$$2 \times 3.14 = 6.28$$

其他：条件表达式和算术表达式、结合律、交换律

5.5 数组引用的表示

- 考虑三地址指令序列

1. $x = a[i]$

2. $a[j] = y$

3. $z = a[i]$

如果把 $a[i]$ 看作一个公共子表达式，可能会把

$$z = a[i]$$

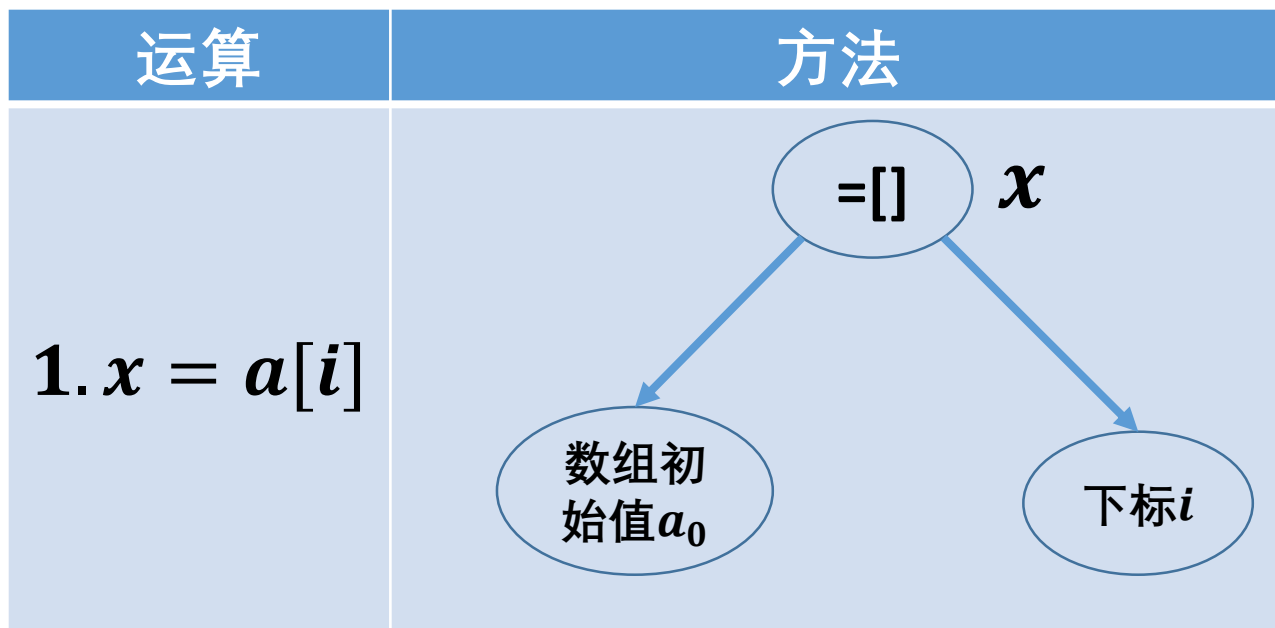
优化为

$$z = x$$

但是， $a[j] = y$ 可能会改变 $a[i]$ 的值，因此优化不合法

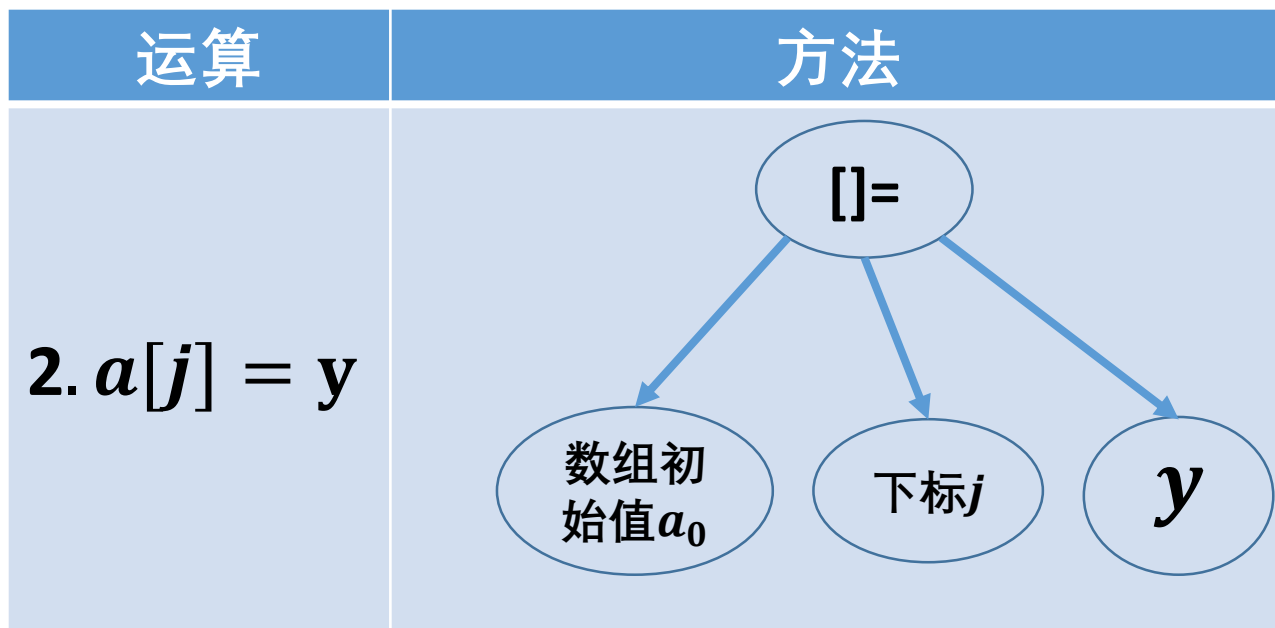
5.5 数组引用的表示

- 在有向无环图中，表示数组访问的正确方法如下：



5.5 数组引用的表示

- 在有向无环图中，表示数组访问的正确方法如下：



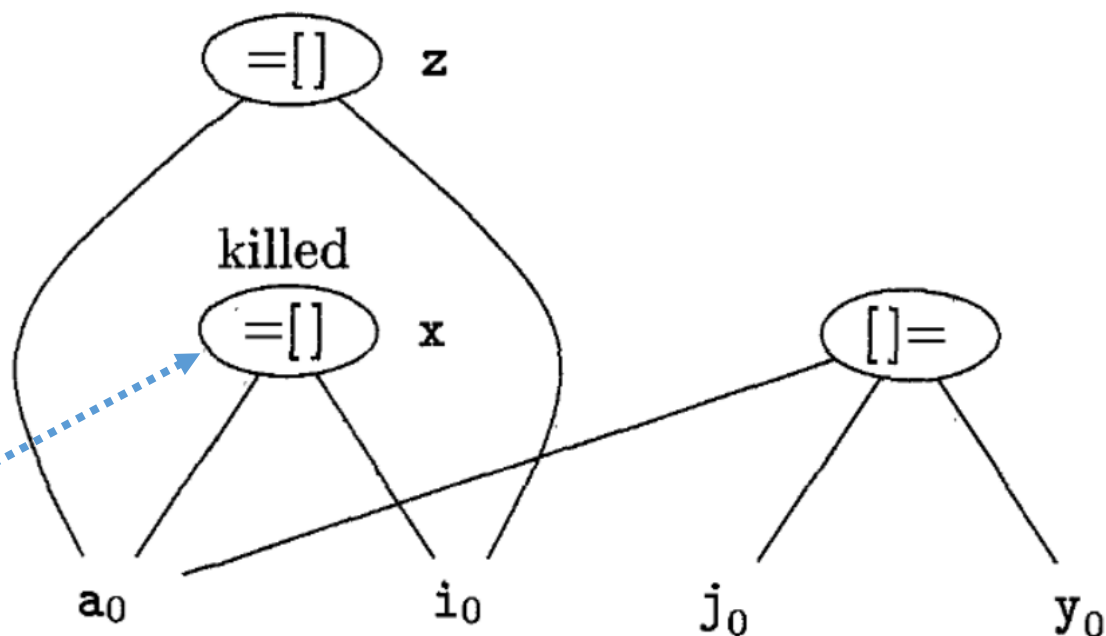
5.5 数组引用的表示

- 考虑三地址指令序列

1. $x = a[i]$

2. $a[j] = y$

3. $z = a[i]$

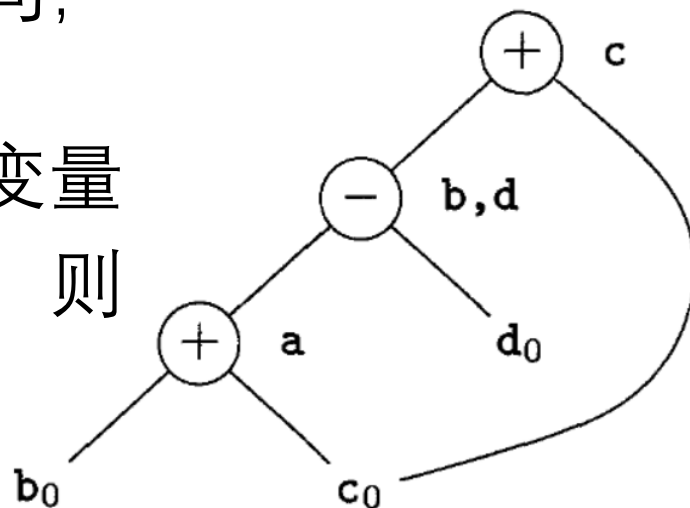


被杀死后，不能作为公共子表达式

有向无环图

5.6 从DAG到基本块的重组

- 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



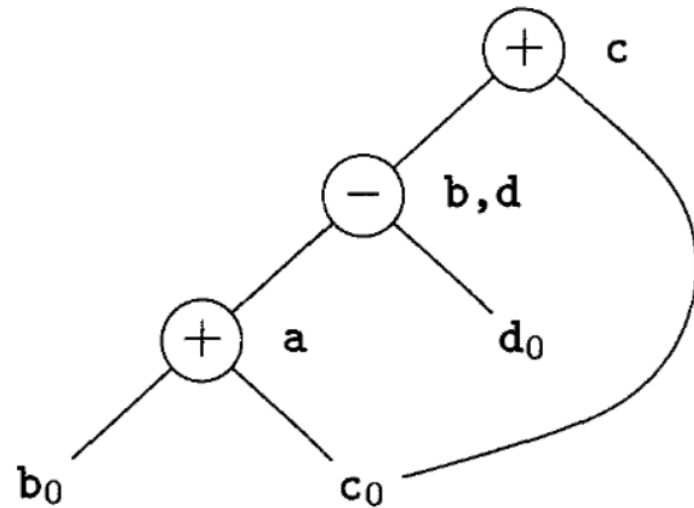
1. $a = b + c$
2. $b = a - d$
3. $c = b + c$
4. $d = a - d$

如果b在基本块出口不活跃

1. $a = b + c$
2. $d = a - d$
3. $c = d + c$

5.6 从DAG到基本块的重组

- 如果b和d都在基本块出口处活跃



1. $a = b + c$

2. $b = a - d$

3. $c = b + c$

4. $d = a - d$

如果b在基本块出口不活跃

1. $a = b + c$

2. $d = a - d$

3. $b = d$

4. $c = d + c$

5.6 从DAG到基本块的重组

- 重组时应该注意求值的顺序
 - 指令顺序必须遵守DAG中结点的顺序
 - 数组**赋值**必须跟在所有**原来在它之前的赋值/求值**操作之后
 - 数组元素**求值**必须跟在所有**原来在它之前的赋值**指令之后
 - **变量使用**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
 - **过程调用**或者**指针间接赋值**必须跟在**原来在它之前的任何变量求值**之后

第六节 一个简单的代码生成器

6 一个简单的代码生成器

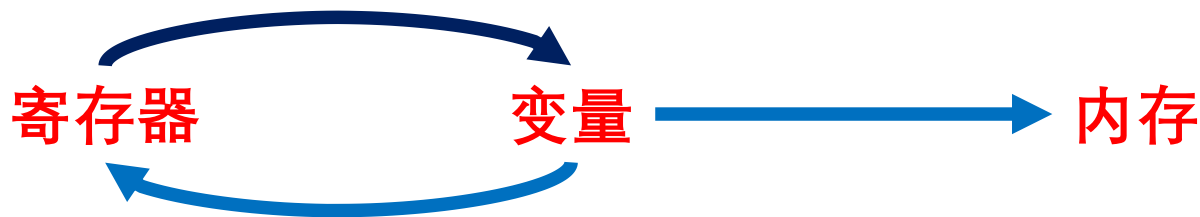
- 如何最大限度地利用寄存器资源？
- 寄存器的四种主要使用方法
 - 执行运算时，运算分量必须放在寄存器中
 - 用于存放临时变量
 - 用于存放全局的值，值被跨块使用时
 - 用于运行时存储管理

6 一个简单的代码生成器

- 尽可能把值存入寄存器，减少寄存器/内存之间的数据交换
- 为三地址指令生成机器指令时
 - 仅当运算分量不在寄存器中时，才从内存载入
 - 仅当寄存器中的值没用时，才覆盖它

6 一个简单的代码生成器

- 有必要维护值与位置之间的联系信息
 - **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)



6 一个简单的代码生成器

- 代码生成算法框架
 - 遍历三地址指令
 - 利用`getReg(I)`函数选择合适的寄存器
 - 根据可用寄存器资源生成机器指令
- 函数`getReg(I)`
 - 根据寄存器描述符和地址描述符、数据流信息，为指令`I`选择最佳的寄存器
 - 机器指令质量依赖于寄存器的选取算法

6 一个简单的代码生成器

- 运算的机器指令

考虑三地址指令： $x = y + z$

➤调用 $\text{getReg}(x=y+z)$ ，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

$\text{LD } R_y, y'$ (y' 表示存放 y 值的当前位置)

➤类似地，确定是否生成： $\text{LD } R_z, z'$

➤最后，生成指令： $\text{ADD } R_x, R_y, R_z$

6 一个简单的代码生成器

- 复制语句的机器指令

考虑三地址指令： **$x=y$**

➤ **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 **$LD Ry, y$**

➤ 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 **$ST x, Rx$**

6 一个简单的代码生成器

- 修改寄存器和地址描述符的规则

规则1：指令 **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
- 从其他变量的地址描述符中删除**R**

6 一个简单的代码生成器

- 修改寄存器和地址描述符的规则

规则2 : ST x , R

- 修改 x 的地址描述符，包含自己的内存位置

规则3 : 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 （不含 x 的内存位置）
- 从其他变量的地址描述符中删除 R_x

6 一个简单的代码生成器

- 修改寄存器和地址描述符的规则

规则4：处理 $x=y$ 时

- 如果生成**LD Ry y**，按照**规则1**处理，然后
- 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**

6 一个简单的代码生成器

- 把如下基本块翻译成机器代码

1. $t = a - b$

2. $u = a - c$

3. $v = t + u$

4. $a = d$

5. $d = v + u$

其中a、b、c、d在出口处活跃，t、u、v是局部临时变量

6 一个简单的代码生成器

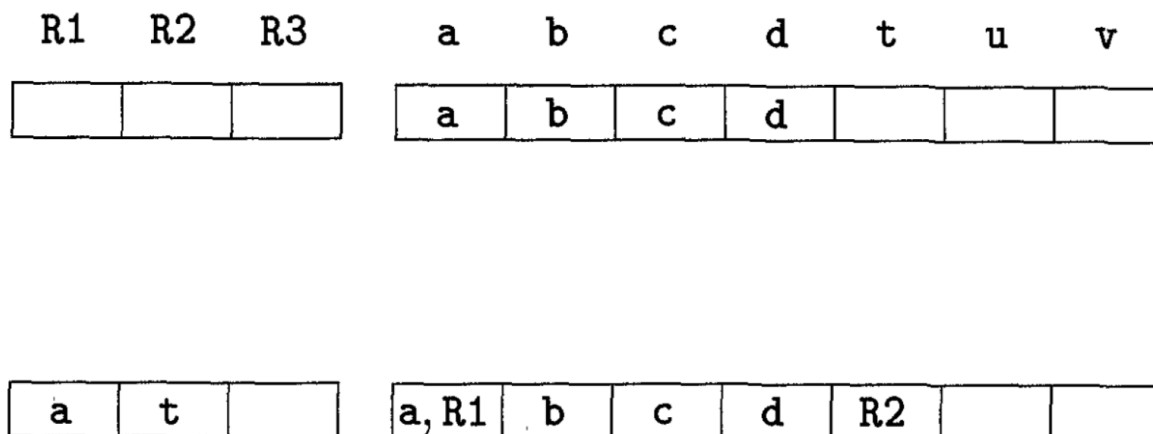
处理第一条三地址指令

$t = a - b$

LD R1, a

LD R2, b

SUB R2, R1, R2



b的值在基本块内不再使用，所以可以使用R2来存放t的值

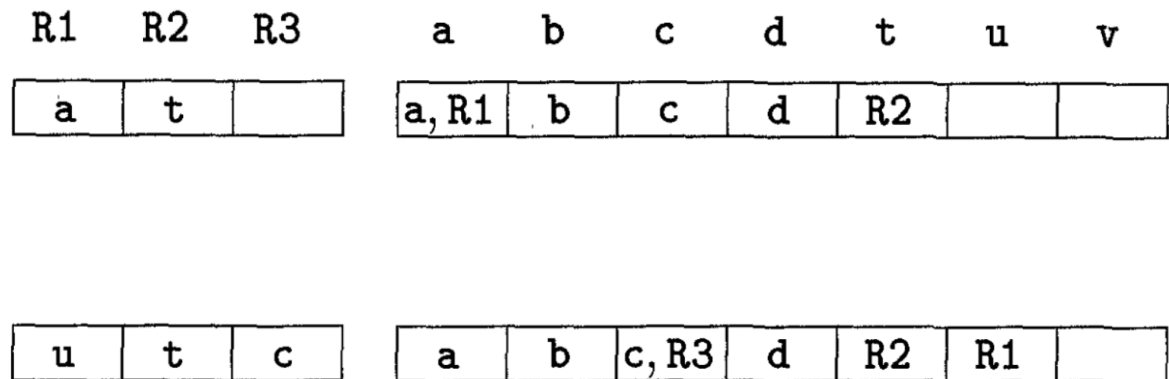
6 一个简单的代码生成器

处理第二条三地址指令

$u = a - c$

LD R3, c

SUB R1, R1, R3



a的值在基本块内不再使用，所以可以使用R1来存放u的值
a的地址描述符被修改了，a不在R1中，但仍在内存中

6 一个简单的代码生成器

处理第三条三地址指令

$v = t + u$

ADD R3, R2, R1

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

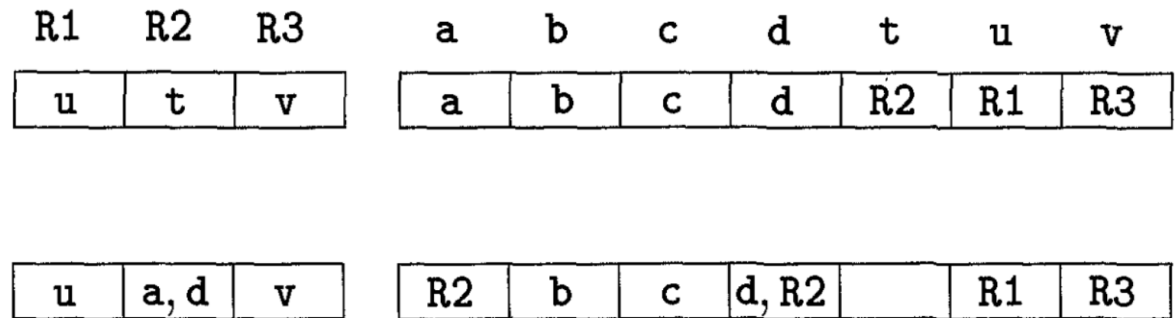
c的值在基本块内不再使用，所以可以使用R3来存放v的值

6 一个简单的代码生成器

处理第四条三地址指令

a = d

LD R2, d



t不再使用

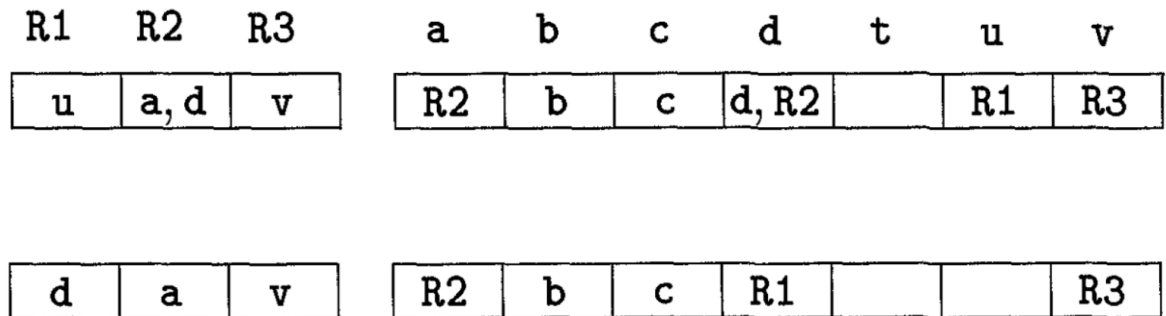
把a加入到R2的描述符中是处理复制语句的结果，不是指令自动处理的

6 一个简单的代码生成器

处理第五条三地址指令

$d = v + u$

ADD R1, R3, R1



↑
u不再使用

6 一个简单的代码生成器

收尾

exit

ST a, R2

ST d, R1

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

d	a	v	a, R2	b	c	d, R1			R3
---	---	---	-------	---	---	-------	--	--	----

将在出口处活跃的变量a和d的值保存回它们的内存位置

a、b、c、d在出口处活跃

第七节 窥孔优化

7 窥孔优化

- 使用一个滑动窗口(窥孔)来检查**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

7 窥孔优化

- 同一个基本块内多余的LD/ST指令

- **LD R0, a**

- **ST a, R0**

可以删除其中的保存指令

7 窥孔优化

- 级联跳转代码

1. `if debug==1 goto L1`

2. `goto L2`



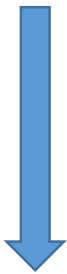
1. `if debug!=1 goto L2`

如果已知debug一定是0，那么替换成为goto L2

7 窥孔优化

- 控制流优化

```
goto L1
...
L1 : goto L2
```



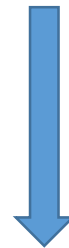
```
goto L2
...
L1 : goto L2
```

```
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 :
```

7 窥孔优化

- 应用代数恒等式进行优化
 - 消除 $x = x + 0$ 、 $x = x * 1$ 等
 - 使用 $x * x$ 替换 x^2
- 使用机器特有指令：INC，DEC等

小结

- 目标机器指令、寻址方式
- 基本块和流图的概念
- 基本块的局部优化
- 代码生成，维护寄存器和地址描述符
- 窥孔优化