



山东大学
SHANDONG UNIVERSITY

编译原理

第十一章 目标代码生成

授 课 教 师 : 余仲星
手 机 : 15866821709 (微信同号)
邮 箱 : zhongxing.yu@sdu.edu.cn

第十一章 目标代码生成

- 11.1 基本问题
- 11.2 目标机器模型
- 11.3 一个简单的代码生成器
 - 11.3.1 待用信息
 - 11.3.2 寄存器描述和地址描述
 - 11.3.3 代码生成算法
- 11.4 寄存器分配
- 11.5 DAG的目标代码
- 11.6 窥孔优化

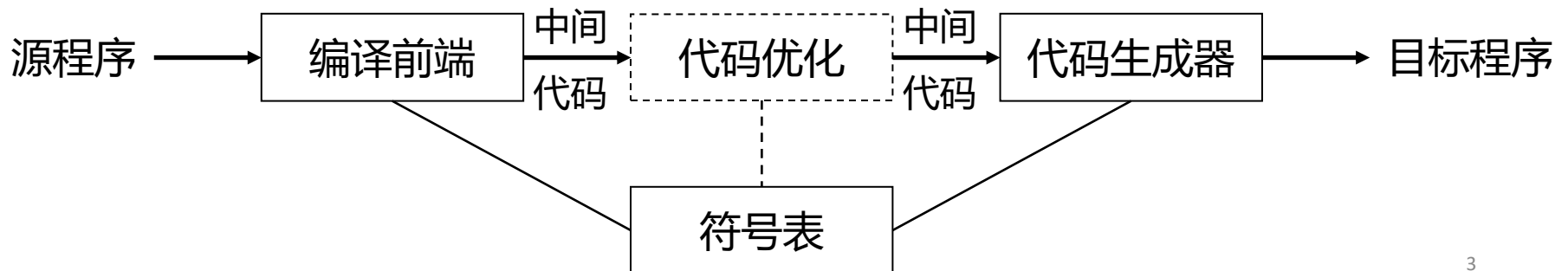
第十一章 目标代码生成

□ **目标代码生成：**以源程序的中间代码作为输入，产生等价的目标程序作为输出；目标代码有三种形式

- 能够立即执行的机器语言代码，所有地址均已定位。
- 待装配的机器语言模块。
- 汇编语言代码，需经过汇编程序汇编，转换为可执行的机器语言代码。

□ **代码生成要着重考虑两个问题：**

- 如何使生成的目标代码较短；
- 如何充分利用寄存器，减少目标代码中访问存储单元的次数。



第十一章 目标代码生成

□ 11.1 基本问题

□ 11.2 目标机器模型

□ 11.3 一个简单的代码生成器

- 11.3.1 待用信息
- 11.3.2 寄存器描述和地址描述
- 11.3.3 代码生成算法

□ 11.4 寄存器分配

□ 11.5 DAG的目标代码

□ 11.6 窥孔优化

11.1 基本问题

□ 代码生成器的输入

- 中间语言的选择，本章采用三地址码，但其中许多技术可以用于其它中间表示
- 代码生成器利用符号表中的信息，决定中间代码中名字所指示的数据对象的运行时地址，它是可再定位地址或绝对地址。
- 假定已做过类型检查，输入没有错误。

□ 目标程序

- 本章采用汇编语言作为目标语言。

□ 指令选择

- 指令集的一致性和完全性是重要因素；
- 生成代码的质量取决于它的速度和大小。

11.1 基本问题

□ 寄存器分配

- 在寄存器分配期间，为程序的某一点选择驻留在寄存器中的一组变量；
- 在随后的寄存器指派阶段，挑出变量将要驻留的具体寄存器。

□ 计算顺序选择

- 计算完成的顺序会影响目标代码都有效性。

第十一章 目标代码生成

□ 11.1 基本问题

□ 11.2 目标机器模型

□ 11.3 一个简单的代码生成器

- 11.3.1 待用信息
- 11.3.2 寄存器描述和地址描述
- 11.3.3 代码生成算法

□ 11.4 寄存器分配

□ 11.5 DAG的目标代码

□ 11.6 窥孔优化

11.2 目标机器模型

□ 基本指令

- 直接地址型: $op\ R_i, M$ // 单目: $op\ (M) \Rightarrow R_i$, 双目: $(R_i)\ op\ (M) \Rightarrow R_i$
- 寄存器型: $op\ R_i, R_j$ // 单目: $op\ (R_j) \Rightarrow R_i$, 双目: $(R_i)\ op\ (R_j) \Rightarrow R_i$
- 变址型: $op\ R_i, c(R_j)$
 // 单目: $op\ (c + (R_j)) \Rightarrow R_i$, 双目: $(R_i)\ op\ (c + (R_j)) \Rightarrow R_i$
- 间接型: $op\ R_i, *M$ // 单目: $op\ ((M)) \Rightarrow R_i$, 双目: $(R_i)\ op\ ((M)) \Rightarrow R_i$
 $op\ R_i, *R_j$ // 单目: $op\ ((R_j)) \Rightarrow R_i$, 双目: $(R_i)\ op\ ((R_j)) \Rightarrow R_i$
 $op\ R_i, *c(R_j)$
 // 单目: $op\ ((c + (R_j))) \Rightarrow R_i$, 双目: $(R_i)\ op\ ((c + (R_j))) \Rightarrow R_i$
- 运算符 (操作码) op 包括常见的运算, 如果ADD、SUB、MUL、DIV等。
- 立即数前面加#。

11.2 目标机器模型

□ 存取指令

- $LD\ R_i, B$ // 把 B 单元的内容加载到寄存器 R_i , 即 $(B) \Rightarrow R_i$
- $ST\ R_i, B$ // 把寄存器 R_i 的内容存储到 B 单元, 即 $(R_i) \Rightarrow B$

11.2 目标机器模型

□ 比较与跳转指令

- $J X$ // 无条件跳转到 X 单元
- $CMP A, B$ // 把 A 单元和 B 单元的内容比较, 根据比较情况把机器内部特征寄存器 CT 置成相应状态, 0,1,2分别表示小于、等于、大于。
- $J < X$ // 如果 $CT = 0$ 则跳转到 X 单元
- $J \leq X$ // 如果 $CT = 0$ 或 $CT = 1$ 则跳转到 X 单元
- $J = X$ // 如果 $CT = 1$ 则跳转到 X 单元
- $J \neq X$ // 如果 $CT \neq 1$ 则跳转到 X 单元
- $J > X$ // 如果 $CT = 2$ 则跳转到 X 单元
- $J \geq X$ // 如果 $CT = 1$ 或 $CT = 2$ 则跳转到 X 单元

第十一章 目标代码生成

- 11.1 基本问题
- 11.2 目标机器模型
- 11.3 一个简单的代码生成器
 - 11.3.1 待用信息
 - 11.3.2 寄存器描述和地址描述
 - 11.3.3 代码生成算法
- 11.4 寄存器分配
- 11.5 DAG的目标代码
- 11.6 窥孔优化

11.3 一个简单的代码生成器

【例11.1】 $A = (B + C) * D + E$ ，中间代码如下

$$T_1 = B + C$$

$$T_2 = T_1 * D$$

$$A = T_2 + E$$

对 $x = y + z$ ，可以简单翻译为：

LD R, y

ADD R, z

ST R, x

目标代码

(1) *LD R, B*

(2) *ADD R, C*

(3) *ST R, T₁*

(4) *LD R, T₁*

(5) *MUL R, D*

(6) *ST R, T₂*

(7) *LD R, T₂*

(8) *ADD R, E*

(9) *ST R, A*

11.3 一个简单的代码生成器

目标代码

- (1) *LD R, B*
- (2) *ADD R, C*
- (3) *ST R, T₁*
- (4) *LD R, T₁*
- (5) *MUL R, D*
- (6) *ST R, T₂*
- (7) *LD R, T₂*
- (8) *ADD R, E*
- (9) *ST R, A*

□ 问题

- 指令 (4) (7) 存储后马上取出, 寄存器内容未变化, 因此是多余的;
- 临时变量 T_1 和 T_2 是生成中间代码引入到, 基本块后不活跃, 因此 (3) (6) 多余。

优化后的目标代码

- (1) *LD R, B*
- (2) *ADD R, C*
- (3) *MUL R, D*
- (4) *ADD R, E*
- (5) *ST R, A*

第十一章 目标代码生成

- 11.1 基本问题
- 11.2 目标机器模型
- 11.3 一个简单的代码生成器
 - 11.3.1 待用信息
 - 11.3.2 寄存器描述和地址描述
 - 11.3.3 代码生成算法
- 11.4 寄存器分配
- 11.5 DAG的目标代码
- 11.6 窥孔优化

11.3.1 待用信息

□ 一个基本块范围内考虑如何充分利用寄存器的问題

- 当生成计算某变量值的目标代码时，尽可能的将该变量的值保存在寄存器中，直到该寄存器必须用来存放别的变量值，或达到基本块的出口；
- 后续的目标代码尽可能的引用保存在寄存器的值，而不访问主存。

□ 具体做法

- 把还要引用的变量值尽可能保存在寄存器中；
- 把基本块内不再被引用的变量所占用的寄存器及早释放。

□ 为此，当翻译 $A = B \text{ op } C$ 时，需要知道：

- A, B, C 是否还会在基本块内被引用，即活跃信息；
- 在哪些中间代码中被引用，即待用信息。

11.3.1 待用信息

□ 获得待用信息和活跃信息的基本思路

- 从基本块的出口从后向前扫描每个中间代码，对每个变量建立相应的待用信息链和活跃变量信息链；
- 如果没有进行过数据流分析，且临时变量不可跨基本块使用，则把基本块中所有临时变量均看作基本块出口之后的非活跃变量，而把所有非临时变量看作基本块出口之后的活跃变量；
- 如果某些临时变量可以跨基本块引用，那么也把它们看作基本块出口之后的活跃变量。

□ 符号

- 符号 (x, x) 表示待用及活跃信息；
- 待用信息 i 表示下一个引用点；
- 活跃信息用 Y 表示活跃。

获取待用信息的算法

□ 为每个变量建立待用和活跃信息链，每条中间代码的变量可附加该信息

(1) 初始化，把基本块中各变量的符号表中，待用信息栏填为“非待用”；并根据变量在基本块出口之后是不是活跃，填写活跃信息栏。

(2) 从基本块出口到基本块入口由后向前依次处理各个中间代码，对每个中间代码

(i) $A = B \text{ op } C$

- ① 把符号表中 A 的待用信息和活跃信息附加到中间代码(i)上；
- ② 把符号表中 A 的待用信息和活跃信息分别置为“非待用”和“非活跃”；
- ③ 把符号表中 B 和 C 的待用信息和活跃信息附加到中间代码(i)上；
- ④ 把符号表中 B 和 C 的待用信息置为 i ，活跃信息置为“活跃”。

【例11.2】考察基本块，其中 W 是出口活跃变量，计算待用信息和活跃信息。

$$(1) T = A - B$$

$$(2) U = A - C$$

$$(3) V = T + U$$

$$(4) W = V + U$$

变量名	待用信息及活跃信息
T	$(-, -) \rightarrow (3, Y) \rightarrow (-, -)$
A	$(-, -) \rightarrow (2, Y) \rightarrow (1, Y)$
B	$(-, -) \rightarrow (1, Y)$
C	$(-, -) \rightarrow (2, Y)$
U	$(-, -) \rightarrow (4, Y) \rightarrow (3, Y) \rightarrow (-, -)$
V	$(-, -) \rightarrow (4, Y) \rightarrow (-, -)$
W	$(-, Y) \rightarrow (-, -)$

序号	中间代码	左值	左操作数	右操作数
1	$T = A - B$	$(3, Y)$	$(2, Y)$	$(-, -)$
2	$U = A - C$	$(3, Y)$	$(-, -)$	$(-, -)$
3	$V = T + U$	$(4, Y)$	$(-, -)$	$(4, Y)$
4	$W = V + U$	$(-, Y)$	$(-, -)$	$(-, -)$

第十一章 目标代码生成

- 11.1 基本问题
- 11.2 目标机器模型
- 11.3 一个简单的代码生成器
 - 11.3.1 待用信息
 - 11.3.2 寄存器描述和地址描述
 - 11.3.3 代码生成算法
- 11.4 寄存器分配
- 11.5 DAG的目标代码
- 11.6 窥孔优化

11.3.2 寄存器描述和地址描述

□ 寄存器信息

- 空闲？分配给某个变量？分配给几个变量（复写时出现这种情况）？
- 建立一个编译时用的寄存器描述数组 *Rvalue*，动态地记录各寄存器的上述信息

□ 变量信息

- 如果变量存在寄存器中，自然希望使用寄存器中的值，而不是主存中的值。
- 建立一个变量地址描述数组 *Avalue*，动态地记录各变量现行值的存放位置：是寄存器中，还是某主存单元，还是既在寄存器又在主存单元。

第十一章 目标代码生成

- 11.1 基本问题
- 11.2 目标机器模型
- 11.3 一个简单的代码生成器
 - 11.3.1 待用信息
 - 11.3.2 寄存器描述和地址描述
 - 11.3.3 代码生成算法
- 11.4 寄存器分配
- 11.5 DAG的目标代码
- 11.6 窥孔优化

□ 获得存放A的寄存器 $getR(i)$ ，设第 i 个四元式为 $A = B \text{ op } C$

(1) 如果B存放在某个寄存器 R_i ， $Rvalue[R_i]$ 只包含B，同时，或者B与A是同一标识符，或者中间代码 i 中B的信息为 $(-, -)$ （即后面不再引用），则选取 R_i 为所需寄存器，转(4)。

(2) 如果(1)失败，若有空闲寄存器 R_i ，选择其作为所需寄存器，转(4)。

(3) 若(2)也失败，需从已分配寄存器中选择 R_i ：占用该 R_i 的变量也保存在主存中，或者在最远的地方被引用（即待用信息值最大）。

(4) 对 $Rvalue[R_i]$ 中的每个变量 M ，如果 $M \neq A$ ，或者 $M = A = C \neq B \wedge B \notin Rvalue[R_i]$ ，则：

① 如果 $M \notin Avalue[M]$ ，生成目标代码 $ST\ R_i, M$ ；

② 如果 $M = B$ ，或者 $M = C \wedge B \in Rvalue[R_i]$ ，则令 $Avalue[M] = \{M, R\}$ ，否则令 $Avalue[M] = \{M\}$ ；

③ 删除 $Rvalue[R_i]$ 中的 M ；

④ 给出 R ，返回。

基本块代码生成算法

□ 对每个中间代码：(i) $A = B \text{ op } C$

(1) $R = \text{getR}(i)$ 。

(2) 根据 $Avalue[B]$ 和 $Avalue[C]$ ，确定变量 B 和 C 的存放位置 B' 和 C' ；如果现行值在寄存器，则把寄存器取做 B' 和 C' 。

(3) 如果 $B' \neq R$ ，则生成目标代码 $LD\ R, B'$ ； $op\ R, C'$ ；否则生成目标代码 $op\ R, C'$ 。如果 B' 或 C' 为 R ，则删除 $Avalue[B]$ 或 $Avalue[C]$ 中的 R 。

(4) 令 $Avalue[A] = \{R\}$ ，并令 $Rvalue[R] = \{A\}$ 。

(5) 如果 B 和 C 在基本块不再引用，在基本块出口后不再活跃，且现行值在某个寄存器 R_i ，则删除 $Rvalue[R_i]$ 中的 B 或 C ，以及 $Avalue[B]$ 或 $Avalue[C]$ 中的 R_i 。

【例11.3】基本块生成目标代码，假设有寄存器 R_0, R_1 。

(1) $T = A - B$

(2) $U = A - C$

(3) $V = T + U$

(4) $W = V + U$

序号	中间代码	左值	左操作数	右操作数
1	$T = A - B$	$(3, Y)$	$(2, Y)$	$(-, -)$
2	$U = A - C$	$(3, Y)$	$(-, -)$	$(-, -)$
3	$V = T + U$	$(4, Y)$	$(-, -)$	$(4, Y)$
4	$W = V + U$	$(-, Y)$	$(-, -)$	$(-, -)$

中间代码	目标代码	Rvalue	Avalue
$T = A - B$	$LD\ R_0, A$ $SUB\ R_0, B$	$Rv(R_0) = \{T\}$	$Av(T) = \{R_0\}$
$U = A - C$	$LD\ R_1, A$ $SUB\ R_1, C$	$Rv(R_0) = \{T\}$ $Rv(R_1) = \{U\}$	$Av(T) = \{R_0\}$ $Av(U) = \{R_1\}$
$V = T + U$	$ADD\ R_0, R_1$	$Rv(R_0) = \{V\}$ $Rv(R_1) = \{U\}$	$Av(V) = \{R_0\}$ $Av(U) = \{R_1\}$
$W = V + U$	$ADD\ R_0, R_1$	$Rv(R_0) = \{W\}$	$Av(W) = \{R_0\}$

各中间代码对应的目标代码

□ $A = B \text{ op } C$

$LD\ R_i, B$

$op\ R_i, C$

- 其中 R_i 是新分配给 A 的寄存器;
- 如果 B 和/或 C 的现行值在寄存器中, 则目标中 B 和/或 C 用寄存器表示。但如果 C 在 R_i 中, 则 C 要用其主存单元表示;
- 如果 B 的现行值在 R_i 中, 则不生成第一条目标代码。

□ $A = op\ B$

$LD\ R_i, B$

$op\ R_i, R_i$

- 其中 R_i 是新分配给 A 的寄存器;
- 如果 B 的现行值在 R_i 中, 则不生成第一条目标代码。

各中间代码对应的目标代码

□ $A = B$

$LD\ R_i, B$

- 其中 R_i 是新分配给 A 的寄存器;
- 如果 B 的现行值在 R_i 中, 则不生成目标代码。

□ $A = B[I]$

$LD\ R_j, I$

$LD\ R_i, B(R_j)$

- 其中 R_i 是新分配给 A 的寄存器;
- 如果 I 的现行值在某个 R_j 中, 则不生成第一条目标代码, 否则 R_j 是分配给 I 的寄存器。

各中间代码对应的目标代码

□ $A[I] = B$

$LD\ R_i, B$

$LD\ R_j, I$

$ST\ R_i, A(R_j)$

- 如果 B 的现行值在 R_i 中，则不生成第一条目标代码；
- 如果 I 的现行值在某个 R_j 中，则不生成第二条目标代码，否则 R_j 是分配给 I 的寄存器。

□ $goto\ X$

$J\ X'$

- X' 是标号为 X 的中间代码的目标代码的首地址。

各中间代码对应的目标代码

□ *if A θ B goto X*

LD R_i, A

CMP R_i, B

j θ X'

- X' 是标号为 X 的中间代码的目标代码的首地址;
- 如果 A 的现行值在 R_i 中, 则不生成第一条目标代码;
- 如果 B 的现行值在某个 R_k 中, 则目标代码中的 B 就是 R_k ;
- θ 指 $<, \leq, =, \neq, >, \geq$ 。

□ *A =* p*

*LD R_i, * p*

- 其中 R_i 是新分配给 $* p$ 的寄存器。

各中间代码对应的目标代码

□ $*p = A$

$LD R_i, A$

$ST R_i, *p$

- 其中 R_i 是新分配给 A 的寄存器;
- 如果 A 的现行值在 R_i 中, 则不生成第一条目标代码。

□ 如果寄存器中的某变量在基本块出口之后是活跃的, 则需要用 ST 指令将其存储到主存单元中

- 利用 $Rvalue$ 确定哪些变量的现行值在寄存器中;
- 利用 $Avalue$ 确定哪些变量的现行值不在主存中;
- 利用活跃变量信息确定哪些变量是活跃的。

第十一章 目标代码生成

- 11.1 基本问题
- 11.2 目标机器模型
- 11.3 一个简单的代码生成器
 - 11.3.1 待用信息
 - 11.3.2 寄存器描述和地址描述
 - 11.3.3 代码生成算法
- 11.4 寄存器分配
- 11.5 DAG的目标代码
- 11.6 窥孔优化

11.4 寄存器分配

□ 基本思想

- 在循环中，寄存器不是平均分配，而是从可用寄存器中分出几个，固定分配给几个变量单独使用。

□ 执行代价：每条指令的执行代价 = 每条指令访问主存单元次数 + 1

- $op\ R_i, R_j$ 执行代价为1
- $op\ R_i, M$ 执行代价为2
- $op\ R_i, * R_j$ 执行代价为2
- $op\ R_i, * M$ 执行代价为3

□ 可以计算：如果循环中把某个固定寄存器分配给该变量，执行代价能节省多少

- 根据计算结果，把可用的几个寄存器，固定分配给节省执行代价多的几个变量

11.4 寄存器分配

□ 固定分配寄存器，相对于原简单代码生成算法，节省的执行代价计算如下

- 原代码生成算法中，仅当变量在基本块中被定值时，其值才存放在寄存器中。因此固定分配寄存器后，在**该变量被定值前**，**每引用一次**，就减少一次主存访问，**执行代价就减少1**。
- 原代码生成算法中，如果在基本块中被定值且在基本块出口之后是活跃的，那么出基本块时要把它存储到主存中。固定分配后，**出基本块时**无需再转存到主存，因此**执行代价节省2**。

11.4 寄存器分配

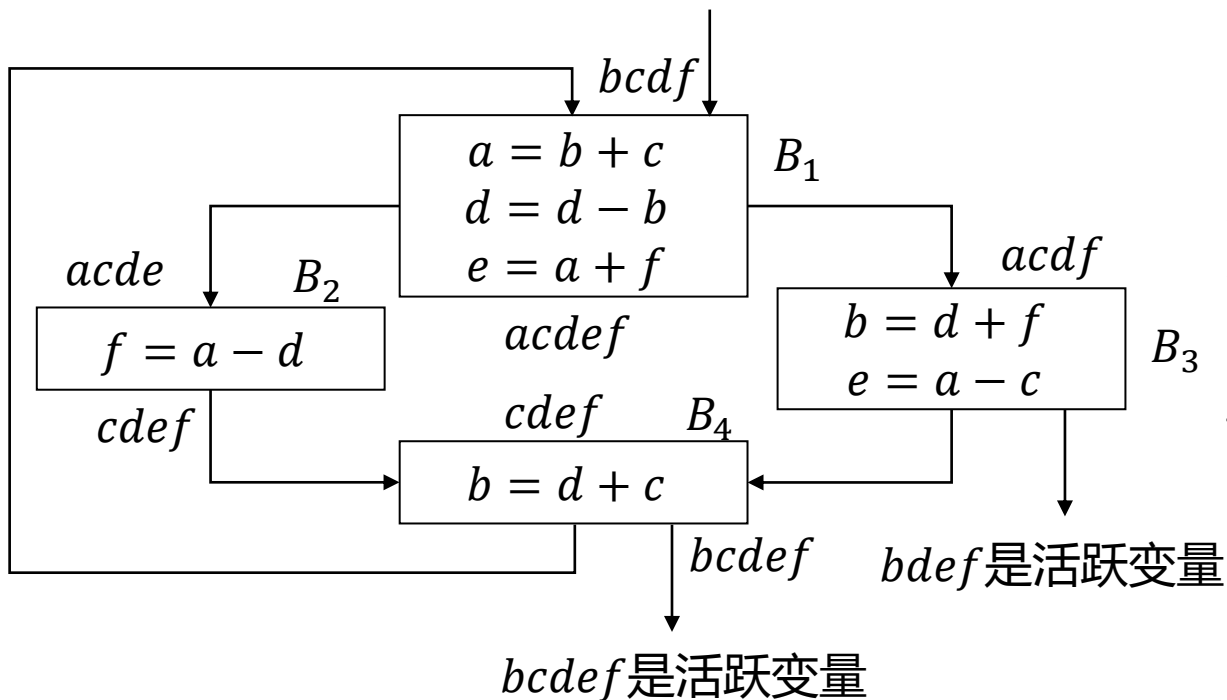
□ 循环 L 中代价节省公式: $\sum_{B \in L} [Use(M, B) + 2Live(M, B)]$

- $Use(M, B)$: 基本块 B 中对 M 定值前引用 M 的次数。
- $Live(M, B) = \begin{cases} 1, & \text{如果} M \text{在基本块} B \text{中被定值且在} B \text{的出口后是活跃的} \\ 0, & \text{其它情况} \end{cases}$

□ 忽略的因素

- 如果 M 在循环入口前是活跃的, 循环入口需要取到固定寄存器, 执行代价加2; 如果 B 是循环出口基本块, C 是循环外 B 的后继基本块, 如果 C 入口前 M 活跃, 则出口时需要将 M 存入主存, 执行代价加2; 但这两处只执行一次, 相对循环次数可以忽略。
- 循环一次, 各基本块不一定都执行到, 该因素也忽略。

【例11.4】某程序的最内层循环，假定 R_0, R_1, R_2 固定分配给3个变量使用。



$$\begin{aligned} Use(c, B_1) &= 1, Use(c, B_2) = 0 \\ Use(c, B_3) &= 1, Use(c, B_4) = 1 \\ Live(c, B_1) &= 0, Live(c, B_2) = 0 \\ Live(c, B_3) &= 0, Live(c, B_4) = 0 \end{aligned}$$

$$\sum_c \dots = 1 + 1 + 1 + 2 \times 0 = 3$$

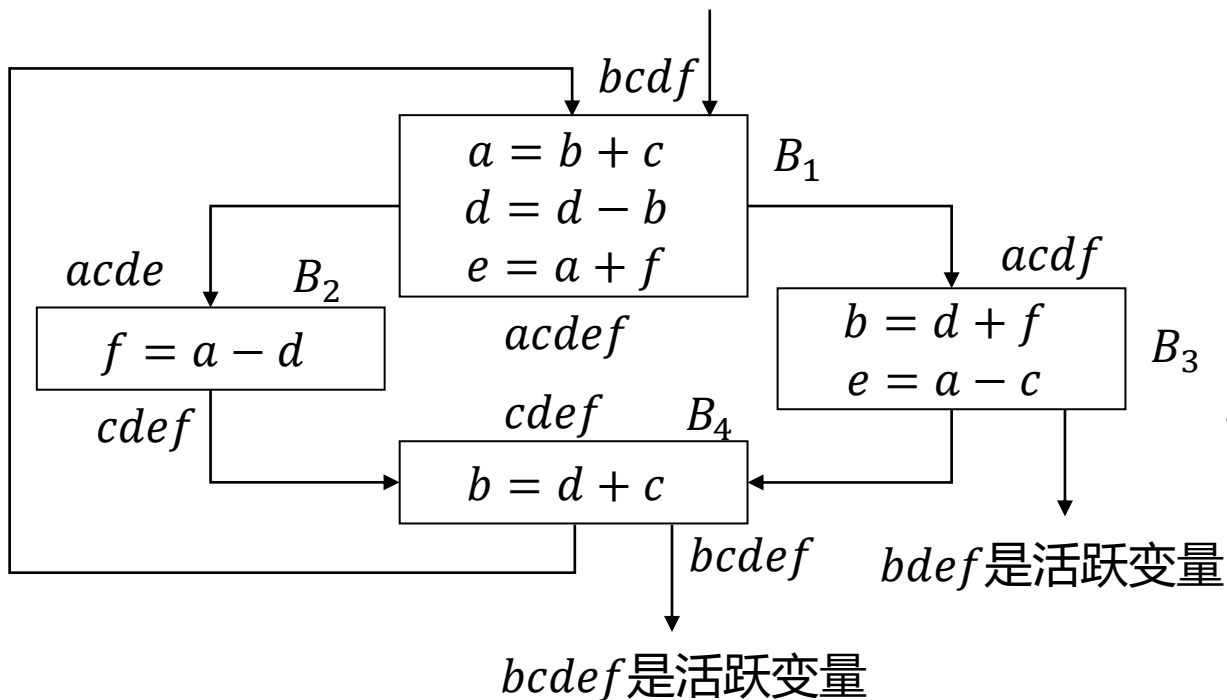
$$\begin{aligned} Use(a, B_1) &= 0, Use(a, B_2) = 1 \\ Use(a, B_3) &= 1, Use(a, B_4) = 0 \\ Live(a, B_1) &= 1, Live(a, B_2) = 0 \\ Live(a, B_3) &= 0, Live(a, B_4) = 0 \end{aligned}$$

$$\sum_a \dots = 1 + 1 + 2 \times 1 = 4$$

$$\begin{aligned} Use(b, B_1) &= 2, Use(b, B_2) = 0 \\ Use(b, B_3) &= 0, Use(b, B_4) = 0 \\ Live(b, B_1) &= 0, Live(b, B_2) = 0 \\ Live(b, B_3) &= 1, Live(b, B_4) = 1 \end{aligned}$$

$$\sum_b \dots = 2 + 2 \times (1 + 1) = 6$$

【例11.4】某程序的最内层循环，假定 R_0, R_1, R_2 固定分配给3个变量使用。



$$Use(f, B_1) = 1, Use(f, B_2) = 0$$

$$Use(f, B_3) = 1, Use(f, B_4) = 0$$

$$Live(f, B_1) = 0, Live(f, B_2) = 1$$

$$Live(f, B_3) = 0, Live(f, B_4) = 0$$

$$\sum_f \dots = 1 + 1 + 2 \times 1 = 4$$

$$Use(d, B_1) = 1, Use(d, B_2) = 1$$

$$Use(d, B_3) = 1, Use(d, B_4) = 1$$

$$Live(d, B_1) = 1, Live(d, B_2) = 0$$

$$Live(d, B_3) = 0, Live(d, B_4) = 0$$

$$B_3 \sum_d \dots = 1 \times 4 + 2 \times 1 = 6$$

$$Use(e, B_1) = 0, Use(e, B_2) = 0$$

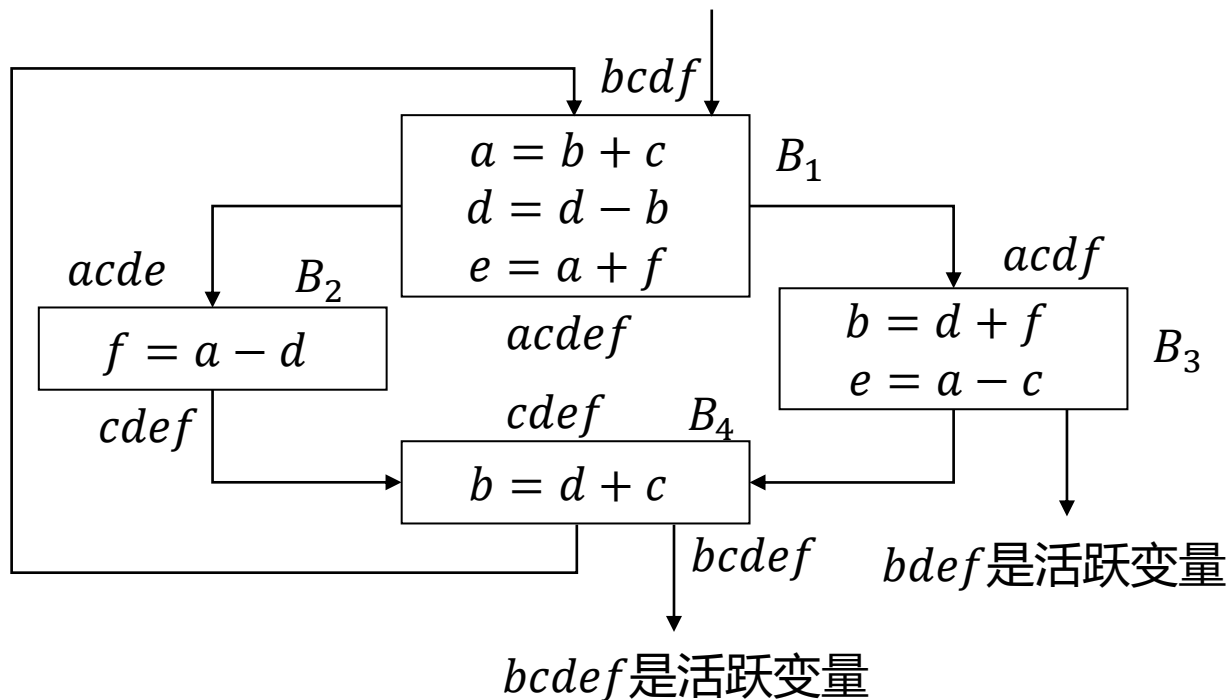
$$Use(e, B_3) = 0, Use(e, B_4) = 0$$

$$Live(e, B_1) = 1, Live(e, B_2) = 0$$

$$Live(e, B_3) = 1, Live(e, B_4) = 0$$

$$\sum_e \dots = 0 + 2 \times (1 + 1) = 4$$

【例11.4】某程序的最内层循环，假定 R_0, R_1, R_2 固定分配给3个变量使用。



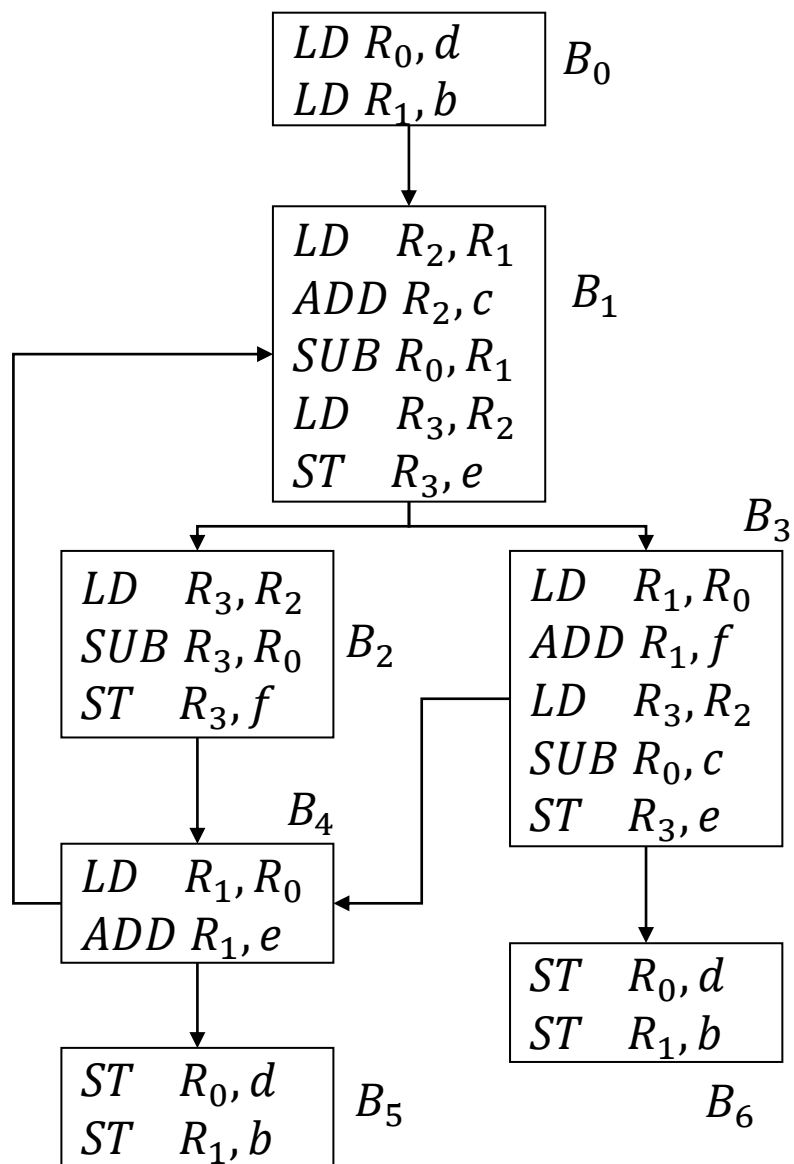
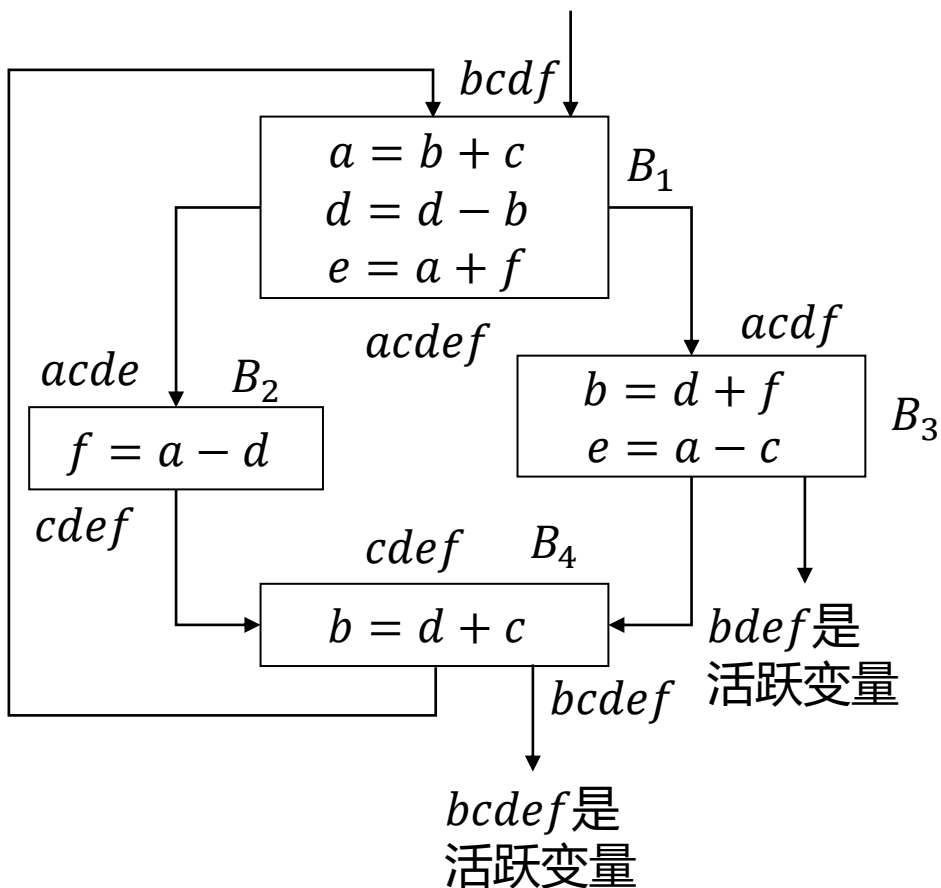
$$\Sigma_a \dots = 4, \Sigma_b \dots = 6, \Sigma_c \dots = 3, \Sigma_d \dots = 6, \Sigma_e \dots = 4, \Sigma_f \dots = 4$$

可以选择 b, d ，以及 a, e, f 中的一个。

寄存器分配目标代码生成

- 循环中，如果涉及已固定分配寄存器的变量，则采用分配给的寄存器
 - 但对 $A = B \text{ op } C$ ，如果 $A = C \neq B$ ，且寄存器 R 固定分配给 A ，但 B 的值不在 R 中，那么当 $M \notin Avalue[C]$ 时，先生成目标代码 $ST\ R, C$ ，再认为 C 在主存中生成 $A = B \text{ op } C$ 的目标代码。
- 如果其中某变量在循环入口之前是活跃的，那么循环入口之前要生成把它们值分别取到相应寄存器的目标代码。
- 如果其中某变量在循环出口之后是活跃的，那么循环出口后面，要分别生成代码，把它们在寄存器中的值放入主存单元。
- 在循环中每个基本块的出口，对未固定分配到寄存器的变量，仍按以前算法生成目标代码，把它们在寄存器的值存入主存单元。

【例11.5】 R_0, R_1, R_2 固定分配给 d, b, a 使用。



第十一章 目标代码生成

- 11.1 基本问题
- 11.2 目标机器模型
- 11.3 一个简单的代码生成器
 - 11.3.1 待用信息
 - 11.3.2 寄存器描述和地址描述
 - 11.3.3 代码生成算法
- 11.4 寄存器分配
- 11.5 DAG的目标代码
- 11.6 窥孔优化

基本思想

□ 计算 $A = B \text{ op } C$ 时，如果计算完右边左对象 B ，紧接着计算 A ，就可以及时利用寄存器中的信息。

➤ 考虑先算DAG的右子树，再算左子树，最后算父结点。

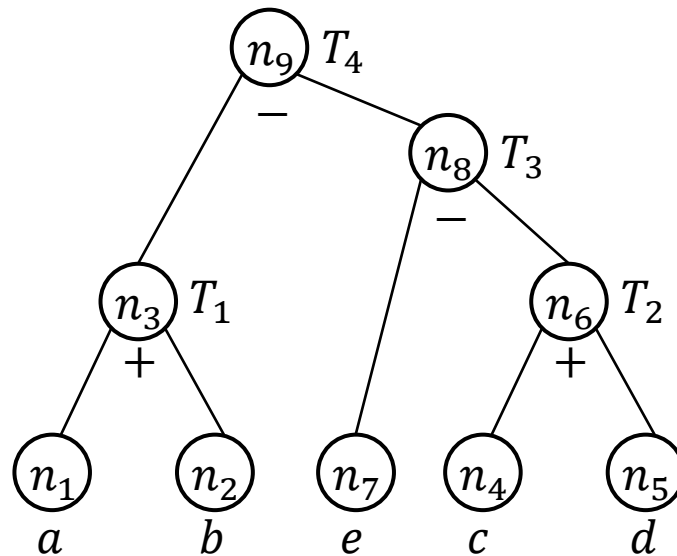
【例11.5】利用DAG调整语句顺序

$$T_1 = a + b$$

$$T_2 = c + d$$

$$T_3 = e - T_2$$

$$T_4 = T_1 - T_3$$



(1) $T_2 = c + d$
(2) $T_3 = e - T_2$
(3) $T_1 = a + b$
(4) $T_4 = T_1 - T_3$

基本思想

□ 原程序 (假设2个寄存器)

$$T_1 = a + b$$

$$T_2 = c + d$$

$$T_3 = e - T_2$$

$$T_4 = T_1 - T_3$$

□ 调序后 (假设2个寄存器)

$$T_2 = c + d$$

$$T_3 = e - T_2$$

$$T_1 = a + b$$

$$T_4 = T_1 - T_3$$

(1) *LD R₀, a*

(2) *ADD R₀, b*

(3) *LD R₁, c*

(4) *ADD R₁, d*

(5) *ST R₀, T₁*

(6) *LD R₀, e*

(7) *SUB R₀, R₁*

(8) *LD R₁, T₁*

(9) *SUB R₁, R₀*

(10) *ST R₁, T₄*

(1) *LD R₀, c*

(2) *ADD R₀, d*

(3) *LD R₁, e*

(4) *SUB R₁, R₀*

(5) *LD R₀, a*

(6) *ADD R₀, b*

(7) *SUB R₀, R₁*

(8) *ST R₀, T₄*

DAG结点计算顺序

□ 设DAG有 N 个内部结点，线性表 $T[N]$ 记录计算顺序，初始为空值。

$i = N;$ // 从后往前填充

while 存在未列入 T 的内部结点 {

 选取一个未列入 T 但其父结点均列入 T ，或者没有父结点的内部结点 n ;

$T[i--] = n$;

 while n 的最左子结点 m 不为叶结点，且其全部父结点均已列入 T {

$T[i--] = m; n = m$;

 }

}

□ 最后的 $T[1], T[2], \dots, T[N]$ 即为结点计算顺序。

□ 如果叶结点上有附加信息，可以先计算。

【例11.6】利用DAG调整语句顺序

$$(1) T_1 = a + b$$

$$(2) T_2 = a - b$$

$$(3) F = T_1 * T_2$$

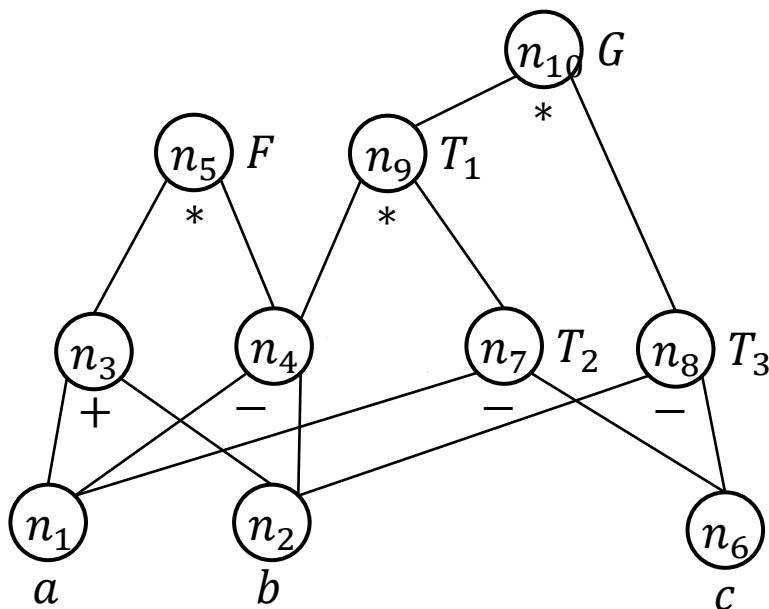
$$(4) T_1 = a - b$$

$$(5) T_2 = a - c$$

$$(6) T_3 = b - c$$

$$(7) T_1 = T_1 * T_2$$

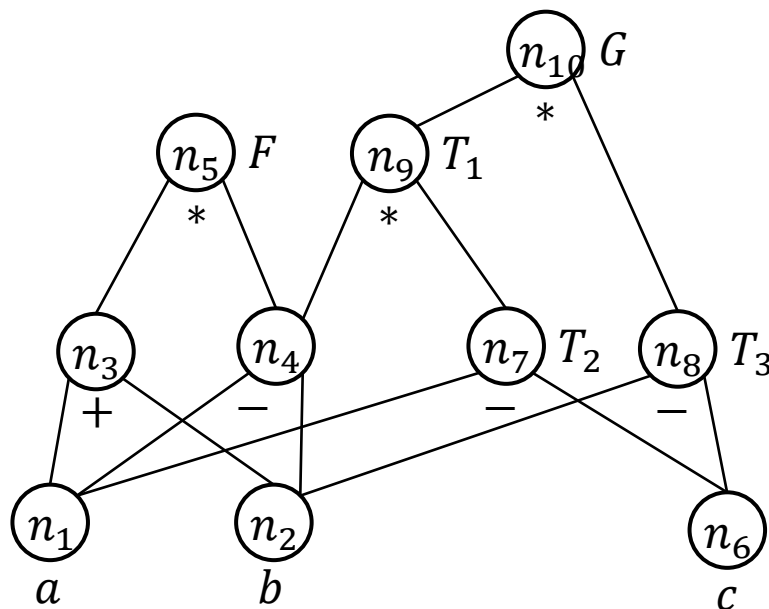
$$(8) G = T_1 * T_3$$



【例11.6】利用DAG调整语句顺序

(1) $T_1 = a + b$
(2) $T_2 = a - b$
(3) $F = T_1 * T_2$
(4) $T_1 = a - b$
(5) $T_2 = a - c$
(6) $T_3 = b - c$
(7) $T_1 = T_1 * T_2$
(8) $G = T_1 * T_3$

1	2	3	4	5	6	7
n_8	n_7	n_4	n_9	n_{10}	n_3	n_5



(1) $T_3 = b - c$
(2) $T_2 = a - c$
(3) $S_1 = a - b$
(4) $T_1 = S_1 * T_2$
(5) $G = T_1 * T_3$
(6) $S_2 = a + b$
(7) $F = S_2 * S_1$

基本块的DAG优化算法

1. 构造运算变量结点

0型

1型

2型

对每个四元式:

(0) $A = B$

(1) $A = \theta B$

(2) $A = B \theta C$

$n = 1$

(下Y右N)

2. 合并已知量

a. 1型B是否为常数

b. 2型B、C是否为常数

c. 1型合并已知量

d. 2型合并已知量

3. 删除公共子表达式

a. 1型

b. 2型

4. 附加A, 删除无用赋值

4、附加A，删除无用赋值

- 若 $Node(A)$ 无定义把A附加到结点 n
- 若 $Node(A)$ 已定义，则把A从原 $Node(A)$ 删除，再把A附加到新结点 n ，以下情况不删除：
 - A是叶结点标记
 - 原 $Node(A)$ 有前驱，且该结点只有A一个附加信息。

重新生成四元式前，把有前驱但没有附加信息的内部结点，生成新的临时变量附加上去。

第十一章 目标代码生成

- 11.1 基本问题
- 11.2 目标机器模型
- 11.3 一个简单的代码生成器
 - 11.3.1 待用信息
 - 11.3.2 寄存器描述和地址描述
 - 11.3.3 代码生成算法
- 11.4 寄存器分配
- 11.5 DAG的目标代码
- 11.6 窥孔优化

11.6 窥孔优化

- ❑ 窥孔优化(peephole optimization): 通过考察一小段目标指令 (称为窥孔), 把这些指令替换为更短和更快的一段指令, 从而提高目标代码质量。
 - 窥孔是目标程序中的一个可移动的小窗口。
 - 窥孔代码不一定是相邻的, 尽管有的实现有这样的要求。
 - 窥孔优化的一个特点是, 优化后所产生的结果可能会给后面的优化提供进一步的机会, 为了最大优化效果, 有时需对目标代码进行若干遍处理。

❑ 冗余存取

(1) $ST\ R_0, A$

(2) $LD\ R_0, A$

- (2)可删除;
- 如果(2)带有标号, 则不能保证(2)一定紧接着(1)执行, 此时不能删除;
- 如果(1)(2)在同一个基本块, 这种变换一定是安全的。

11.6 窥孔优化

□ 不可达代码

- 无条件转移指令之后的无标号指令应该删除;
- 这种操作可以重复。

➤ c语言代码

```
#define debug 0  
.....  
if (debug) {打印调试信息}
```

➤ 中间代码

```
if debug = 1 goto L1  
goto L2  
L1: 打印调试信息  
L2: ... ..
```

➤ 初步优化后

```
if debug ≠ 1 goto L2  
L1: 打印调试信息  
L2: ... ..
```

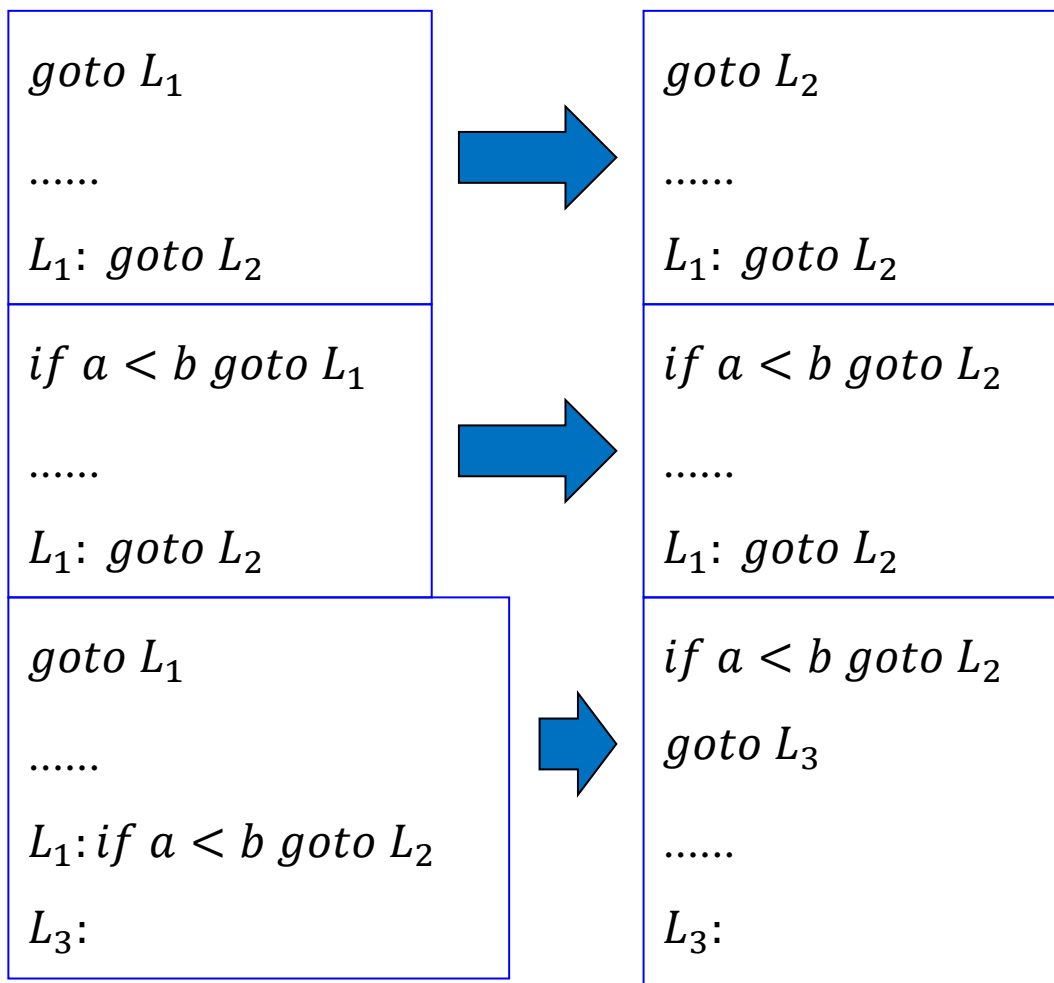
➤ 现在条件 $0 \neq 1$ 恒真，相当于

```
goto L2  
打印调试信息 // 可删  
L2: ... ..
```


11.6 窥孔优化

□ 控制流优化

➤ 中间代码生成算法可能会产生不必要的连续跳转。



➤ 如果没有别的语句跳转到 L_1 ，且 L_1 紧跟在一个无条件跳转语句之后，则可删除。

➤ 同上处理。

➤ 替换后指令条数相同，但后者可能跳过无条件跳转，而前者总要执行无条件跳转。

11.6 窥孔优化

□ 强度削弱

- $MUL\ R, \#2$ 可替换为: $\ll R, \#1$
- $MUL\ R, \#4$ 可替换为: $\ll R, \#2$

□ 删除无用操作

- $ADD\ R, 0$
- $MUL\ R, \#1$