

# 第9章 目标代码生成



*LI Wensheng, SCS, BUPT*

知识点：基本块、程序流图

下次引用信息

代码生成算法

# § 9 目标代码生成

- 9.1 目标代码生成概述
- 9.2 基本块与流图
- 9.3 下次引用信息
- 9.4 一个简单的代码生成程序

# 9.1 目标代码生成概述

## ■ 目标代码生成程序的任务

- ◆ 将前端产生的源程序的中间代码表示转换为等价的目标代码。

## ■ 对目标代码生成程序的要求：

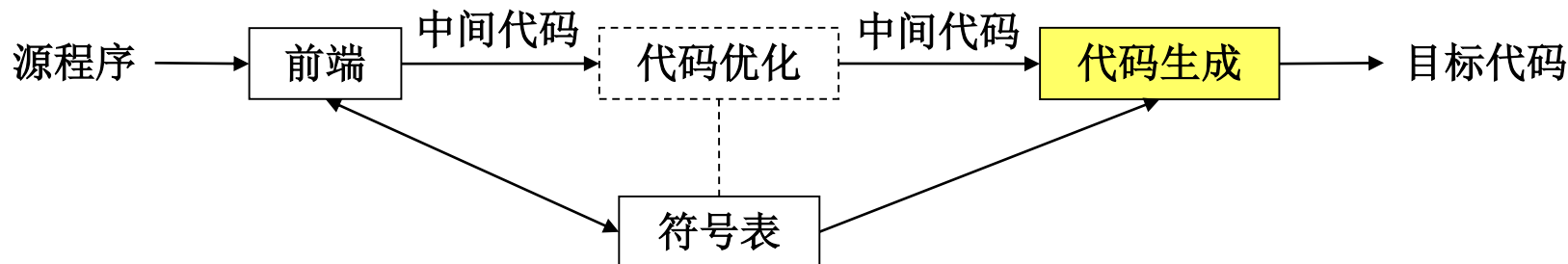
- ◆ 正确
- ◆ 高质量
  - 1. 有效地利用目标机器的资源
  - 2. 占用空间少，运行效率高
- ◆ 产生最优化代码问题是不可判定的，实践中能够产生好的（虽不是最优的）代码的启发式技术就很令人满意了

## ■ 本节内容：

- ◆ 代码生成程序的位置
- ◆ 代码生成程序设计的相关问题

# 9.1.1 代码生成程序的位置

## ■ 代码生成程序在编译程序中的位置



## ■ 代码生成程序的输入

- ◆ 中间代码：经过语法分析/语义检查之后得到的、正确的
  - 假定：前期工作结果正确、可信
  - 中间代码足够详细、必要的类型转换符已正确插入、明显的语义错误已经发现、且正确恢复
- ◆ 符号表
  - 记录了与名字有关的信息
  - 决定中间表示中的名字所代表的数据对象的运行地址

# 代码生成程序的位置(续)

- 代码生成程序的输出：与源程序等价的目标代码
- 目标代码的形式
  - ◆ 绝对地址的机器语言程序
    - 可把目标代码放在内存中固定的地方、立即执行
  - ◆ 可重定位的机器语言程序
    - **.obj (DOS)、.o (UNIX)**
    - 开发灵活，允许各子模块单独编译
    - 由连接装配程序将它们连接在一起，生成可执行文件
  - ◆ 汇编语言程序

## 9.1.2 代码生成程序设计的相关问题

- 代码生成程序的具体细节依赖于目标机器和操作系统
- 代码生成程序设计时需要考虑的问题
  - ◆ 存储管理
  - ◆ 指令选择
  - ◆ 寄存器分配
  - ◆ 计算次序的选择

# 存储管理

- 从名字到存储单元的转换由前端和代码生成程序共同完成
- 符号表中的信息
  - ◆ 在处理声明语句时填入
  - ◆ “类型” 决定了它的域宽
  - ◆ “地址” 确定该名字在过程的数据区域中的相对位置
  - ◆ 上述信息用于确定中间代码中的名字对应的数据对象在运行时的地址
- 三地址代码中的名字
  - ◆ 指向该名字在符号表中位置的指针

# 例如：三地址代码与机器语言代码的对应

	四元式	地址	长度
	...		
→ 100	( , , , )	n	12
101	( , , , )	n+12	8
102	( , , , )	n+20	16
103	( , , , )	n+36	4
	...		

	机器语言代码
0	...
n	四元式100 的机器码
n+12	四元式101 的机器码
n+20	四元式102 的机器码
n+36	四元式103 的机器码
n+40	...

## ■ 对于四元式j: goto i

- ◆  $i < j$  四元式i的地址已有，可以直接生成机器指令
- ◆  $i > j$  将四元式j的地址记入与i相关的链表中，等待回填



# 指令选择

- 机器指令系统的性质决定了指令选择的难易程度

- ◆ 一致性
- ◆ 完整性
- ◆ 指令的执行速度
- ◆ 机器的特点

- 对每一类三地址语句，可以设计它的代码框架

如  $x:=y+z$  的代码框架

MOV R<sub>0</sub>, y

ADD R<sub>0</sub>, z

MOV x, R<sub>0</sub>

$a:=b+c$

$d:=a+e$

MOV R<sub>0</sub>, b

ADD R<sub>0</sub>, c

MOV a, R<sub>0</sub>

MOV R<sub>0</sub>, a

ADD R<sub>0</sub>, e

MOV d, R<sub>0</sub>

$a:=a+1$

INC a

MOV R<sub>0</sub>, a

ADD R<sub>0</sub>, #1

MOV a, R<sub>0</sub>

# 寄存器分配

- 充分利用寄存器可以生成好的代码
- 寄存器使用的两个问题
  - ◆ 哪些变量要放在寄存器中
  - ◆ 指定的变量放在哪个寄存器中
- 寄存器指派的困难
  - ◆ 可用寄存器
    - 专用寄存器
    - 通用寄存器
    - 寄存器对
  - ◆ 把寄存器指派给相应的变量
    - 变量需要什么样的寄存器
    - 操作需要什么样的寄存器
  - ◆ 选择最优的寄存器指派方案是一个NP完全问题

# 计算次序的选择

- 计算次序影响目标代码的效率
- 选择最佳计算次序是一个NP完全问题
- 代码生成程序的设计原则
  - ◆ 能够正确地生成代码
  - ◆ 易于实现、便于测试和维护
- 只介绍一个简单的代码生成算法
  - ◆ 主要考虑寄存器的有效使用

## 9.2 基本块与流图

### ■ 基本块

- ◆ 具有原子性的一组连续语句序列。
- ◆ 控制从第一条语句（入口语句）流入，从最后一条语句（出口语句）流出，中途没有停止或分支

### ■ 如：

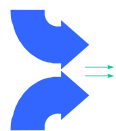
$t_1 := a * a$   
 $t_2 := b * b$   
 $t_3 := t_1 + t_2$

### 基本块：

$t_1 := a * a$   
 $t_2 := a * b$   
 $t_3 := 2 * t_2$   
 $t_4 := t_1 + t_3$   
 $t_5 := b * b$   
 $t_6 := t_4 + t_5$

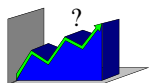
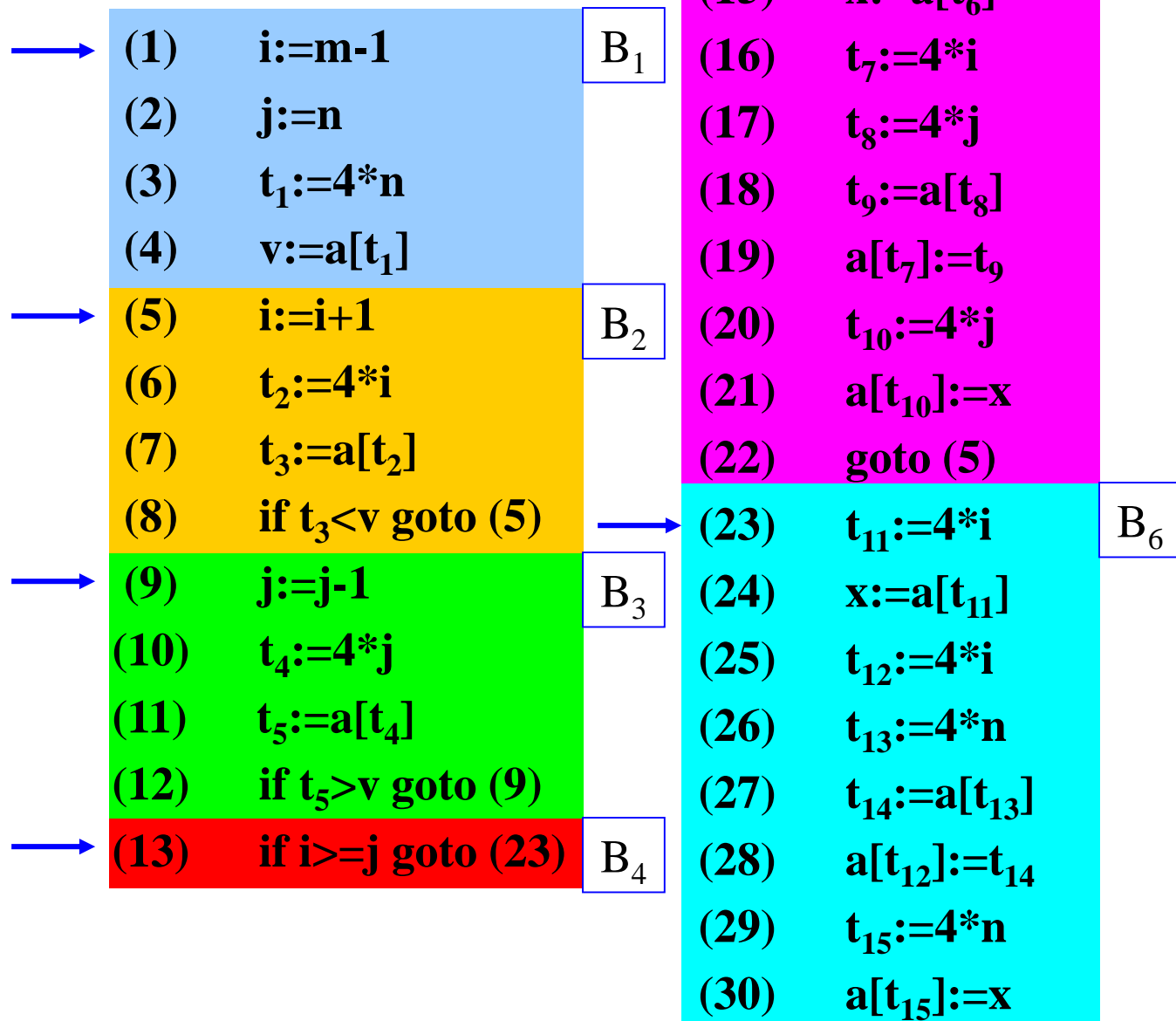
# 基本块的划分方法

- 确定入口语句：
  - ◆ 三地址代码的第一条语句；
  - ◆ **goto**语句转移到的目标语句；
  - ◆ 紧跟在**goto**语句后面的语句。
- 确定基本块：与每一个入口语句相应的基本块
  - ◆ 从一个入口语句（**含该语句**）到下一个入口语句（**不含**）之间的语句序列；
  - ◆ 从一个入口语句（**含该语句**）到停止语句（**含该语句**）之间的语句序列。

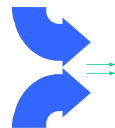


# 举例

## 基本块划分:

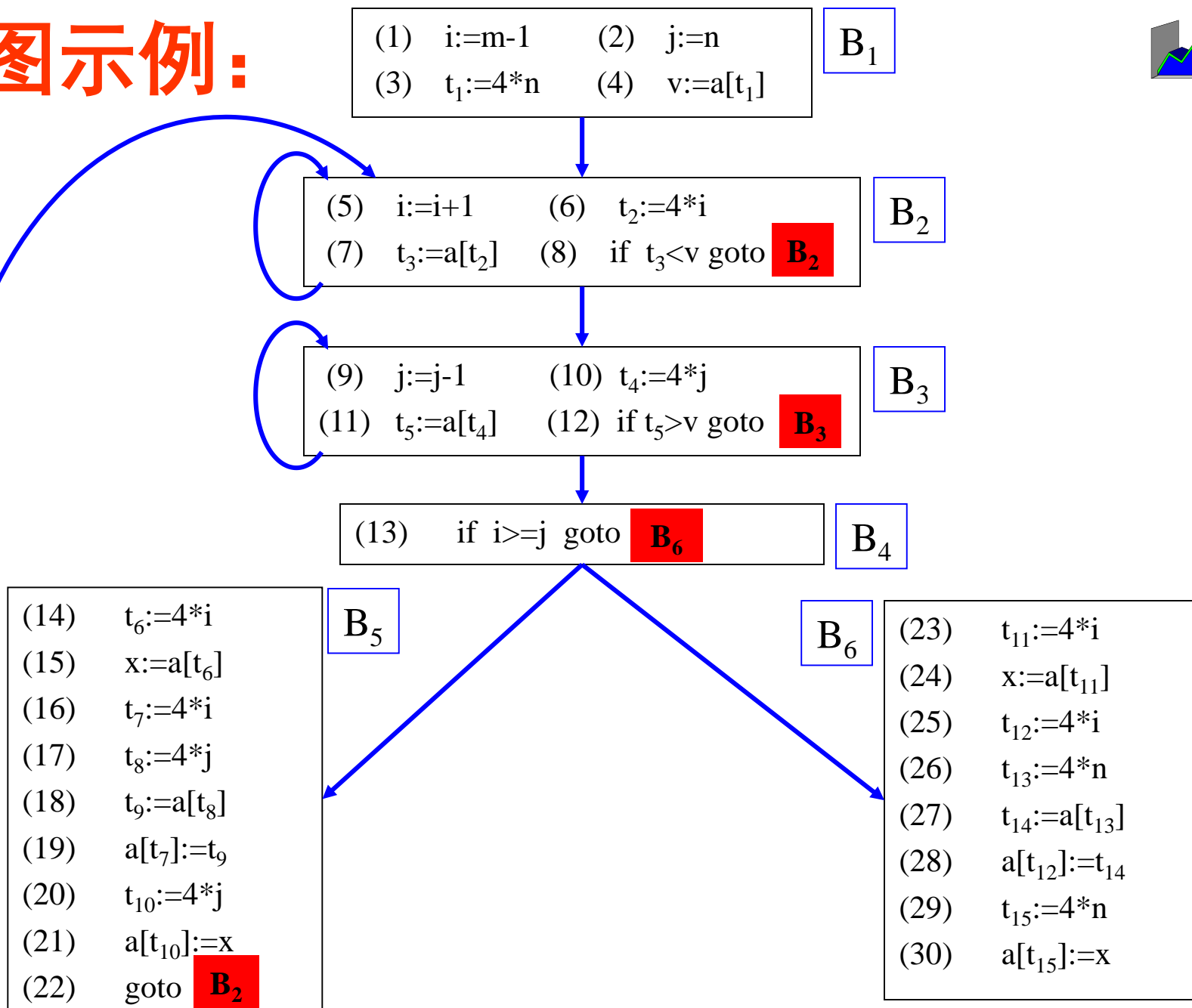
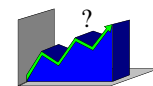


# 流图



- 定义：把控制信息加到基本块集合中，形成程序的有向图，称为**流图**（控制流图）。
- 构造：
  - ◆ 流图的**结点**是基本块
  - ◆ 如果一个结点基本块的入口语句是程序的第一条语句，则称此基本块结点为流图的**首结点**。
  - ◆ 如果在某个执行序列中，基本块 $B_2$ 紧跟在基本块 $B_1$ 之后执行，则从 $B_1$ 到 $B_2$ 有一条有向边， $B_1$ 是 $B_2$ 的**前驱**， $B_2$ 是 $B_1$ 的**后继**。即如果：
    - $B_1$ 的最后一条语句不是无条件转移语句，并且在程序的语句序列中， $B_2$ 紧跟在 $B_1$ 之后。
    - 有一个条件/无条件转移语句从 $B_1$ 的最后一条语句转移到 $B_2$ 的第一条语句；
- 转移语句指向块而不是指向三地址语句
  - ◆ 因为基本块变换后，语句会发生变化
- 循环的定义： ①强连通； ②唯一入口

# 流图示例:





## 9.3 下次引用信息

- 在把三地址代码转换成为目标代码时，遇到的一个重要问题：

如何充分利用寄存器？

- 基本思路：

- ◆ 在一个基本块范围内考虑
- ◆ 把在基本块内还要被引用的变量的值尽可能保存在寄存器中
- ◆ 把在基本块内不再被引用的变量所占用的寄存器尽早地释放

- 如：翻译语句  $x := y \text{ op } z$

- ◆  $x$ 、 $y$ 、 $z$ 在基本块中是否还会被引用？
- ◆ 在哪些三地址语句中被引用？

# 活跃变量

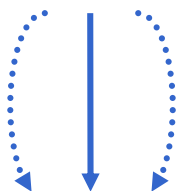
- 考虑变量  $x$  和程序点  $p$
- 分析  $x$  在点  $p$  上的值是否会在流图中的某条从点  $p$  出发的路径中使用。
  - ◆ 是，则  $x$  在  $p$  上是活跃的；
  - ◆ 否则， $x$  在  $p$  上是死的。
- 基于流图进行活跃变量分析，可以实现基本块的存储分配，即只需为活跃变量分配寄存器即可。
- 活跃信息用于代码优化时的全局数据流分析。

# 下次引用

三地址语句序列：

i:  $x := 1$

语句i对变量x定值



没有对变量x定值的其他语句

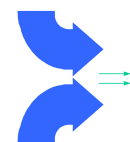
j:  $y := x \text{ op } z$

语句j引用x在语句i处定的值

**语句 j 是三地址语句 i 中 x 的下次引用信息。**

## ■ 假定

- 讨论在一个基本块内的引用信息
- 所有的变量在基本块出口处都是活跃的
- 三地址语句的结构中，记录语句中出现的每个名字的下次引用信息和活跃信息
- 符号表中含有记录下次引用信息和活跃信息的域



# 算法

输入：组成基本块的三地址语句序列。

输出：基本块中各变量的下次引用信息。

方法：

1. 把基本块中各变量在符号表中的下次引用信息域置为“**无下次引用**”、活跃信息域置为“**活跃**”。
2. 从基本块出口到入口**由后向前**依次处理各语句，对每个三地址语句 **i**:  $x := y \text{ op } z$ ，依次执行下述步骤：
  - ①把当前符号表中变量 **x** 的下次引用信息和活跃信息附加到语句 **i** 上；
  - ②把符号表中 **x** 的下次引用信息置为“无下次引用”，活跃信息置为“非活跃”；
  - ③把当前符号表中变量**y和z**的下次引用信息和活跃信息附加到语句 **i** 上；
  - ④把符号表中**y和z**的下次引用信息均置为 '**i**'，活跃信息均置为“活跃”。

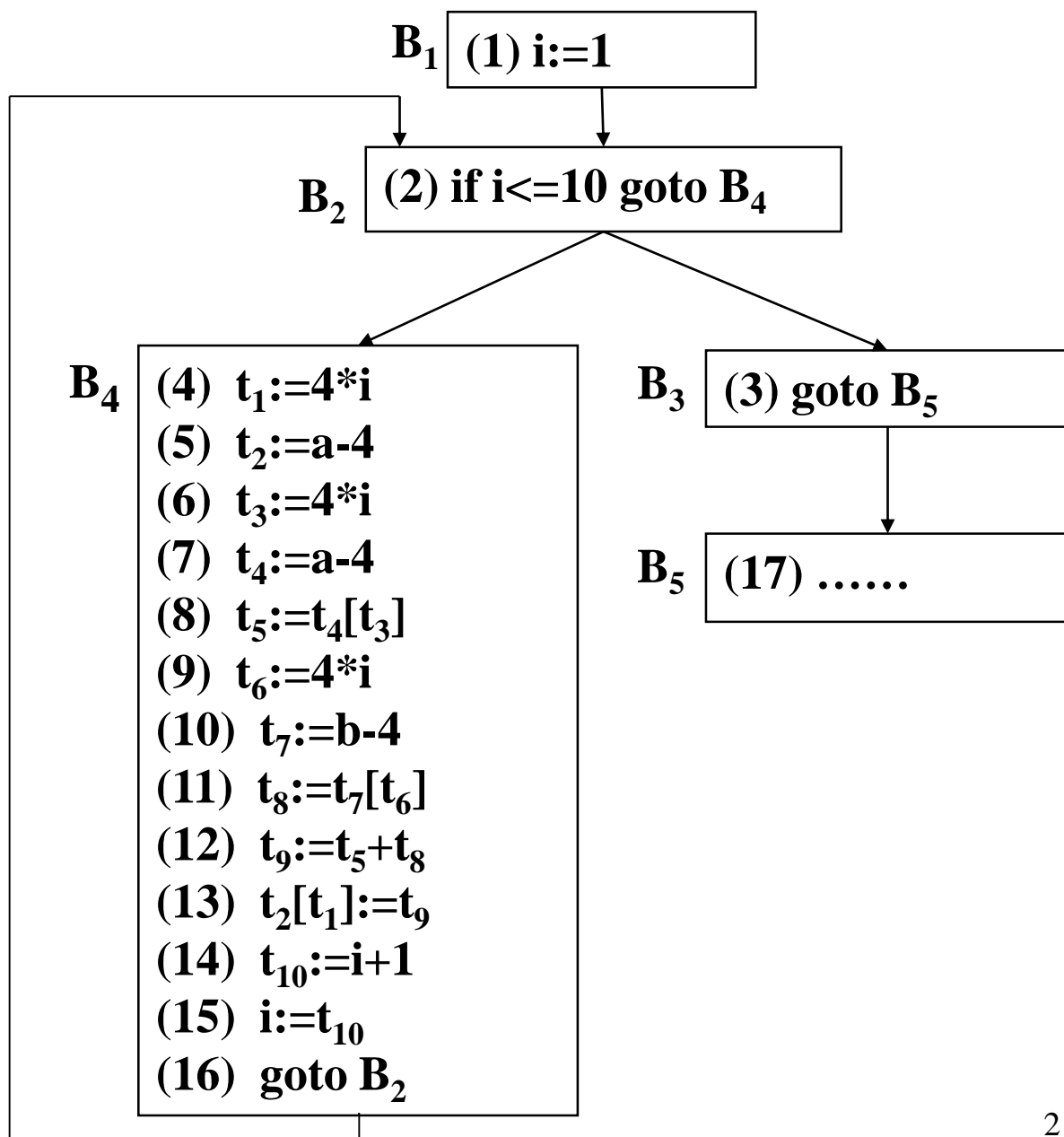
①②  
③④  
不能  
颠倒

# 举例

## ■ 程序片断：

```
i:=1;  
while (i<=10) do  
begin  
    a[i]:=a[i]+b[i];  
    i:=i+1  
end;
```

## ■ 程序的控制流图：



# 计算 $B_4$ 中变量的下次引用信息

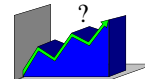
## ■ 初始化符号表:

$B_4$

(4)	$t_1 := 4 * i$
(5)	$t_2 := a - 4$
(6)	$t_3 := 4 * i$
(7)	$t_4 := a - 4$
(8)	$t_5 := t_4[t_3]$
(9)	$t_6 := 4 * i$
(10)	$t_7 := b - 4$
(11)	$t_8 := t_7[t_6]$
(12)	$t_9 := t_5 + t_8$
(13)	$t_2[t_1] := t_9$
(14)	$t_{10} := i + 1$
(15)	$i := t_{10}$
(16)	goto $B_2$

变量	下次	活跃
----	----	----

i	无	活
a	无	活
b	无	活
$t_1$	无	活
$t_2$	无	活
$t_3$	无	活
$t_4$	无	活
$t_5$	无	活
$t_6$	无	活
$t_7$	无	活
$t_8$	无	活
$t_9$	无	活
$t_{10}$	无	活



# 从出口到入口依次检查每条语句

B<sub>4</sub>

(4) $t_1 := 4 * i$	$t_1$ (13) 活	$i$ (6) 活	
(5) $t_2 := a - 4$	$t_2$ (13) 活	$a$ (7) 活	
(6) $t_3 := 4 * i$	$t_3$ (8) 活	$i$ (9) 活	
(7) $t_4 := a - 4$	$t_4$ (8) 活	$a$ 无	活
(8) $t_5 := t_4[t_3]$	$t_5$ (12) 活	$t_4$ 无	活 $t_3$ 无 活
(9) $t_6 := 4 * i$	$t_6$ (11) 活	$i$ (14) 活	
(10) $t_7 := b - 4$	$t_7$ (11) 活	$b$ 无	活
(11) $t_8 := t_7[t_6]$	$t_8$ (12) 活	$t_7$ 无	活 $t_6$ 无 活
(12) $t_9 := t_5 + t_8$	$t_9$ (13) 活	$t_5$ 无	活 $t_8$ 无 活
(13) $t_2[t_1] := t_9$	$t_2$ 无	活 $t_1$ 无	活 $t_9$ 无 活
(14) $t_{10} := i + 1$	$t_{10}$ (15) 活	$i$ 无	非活
(15) $i := t_{10}$	$i$ 无	活 $t_{10}$ 无	活
(16) goto B <sub>2</sub>			

变量 下次 活跃

$i$	(4)	活
$a$	(5)	活
$b$	(10)	活
$t_1$	无	非活
$t_2$	无	非活
$t_3$	无	非活
$t_4$	无	非活
$t_5$	无	非活
$t_6$	无	非活
$t_7$	无	非活
$t_8$	无	非活
$t_9$	无	非活
$t_{10}$	无	非活

## 9.4 一个简单的代码生成程序

- 依次处理基本块中的每条三地址语句
- 考虑在基本块内充分利用寄存器的问題
  - ◆ 尽可能让变量的值保存在寄存器中，只有在下面两种情况下存储它们
    - 如果此寄存器要用于其它计算
    - 已到达基本块出口
  - ◆ 后续的目标代码尽可能引用变量在寄存器中的值，而不访问主存。
- 在基本块之间如何充分利用寄存器的困难：
  - ◆ 一个基本块可能有多个后继，而每个后继又可能有多个前驱，因而后继基本块不易判断变量的值是否存放在寄存器中，以及存放在那个寄存器中
- 在基本块之间如何充分利用寄存器的问題比较复杂，简单起见，在离开基本块时，把有关变量在寄存器中的当前值存放到内存单元中去。
- 代码生成时需考察许多情形，如下次引用信息、活跃信息、当前值的存放位置等，在不同的情况下生成的代码也不同。



## 9.4.1 目标机器描述

- 设计代码生成程序的必要条件：熟悉目标机器

- 一般信息

- ◆ 编址方式：

- 按字节编址
    - 每个字有4个字节

- ◆ 寄存器：

- $n$ 个通用寄存器： $R_0$ 、 $R_1$ 、 $R_{n-1}$

- ◆ 指令形式：

- OP DEST, SRC

其中 OP: MOV、ADD、SUB

SRC: 源操作数

DEST: 目的操作数

# 操作数寻址方式

地址形式	汇编方式	地址	附加开销
立即寻址	#c	常数c	1
直接寻址	M	M	1
间接寻址	@M	contents(M)	1
寄存器寻址	R	R	0
寄存器间接寻址	@R	contents(R)	0
变址寻址	c[R]	c+contents(R)	1
间接变址寻址	@c(R)	contents(c+contents(R))	1
基址寻址	[BR][R]	ccontents(BR)+contents(R)	0

## ■ 指令开销

= 指令所占用存储单元字数

= 1 + DEST寻址方式附加开销 + SRC寻址方式附加开销

# 指令开销举例

- **MOV  $R_0, R_1$** 
  - ◆ 将寄存器 $R_1$ 的内容复制到 $R_0$ 中
  - ◆ 开销：1
- **MOV  $R_5, M$** 
  - ◆ 将存储单元 $M$ 中的内容存入寄存器 $R_5$ 中
  - ◆ 开销：2
- **ADD  $R_3, \#1$** 
  - ◆ 将寄存器 $R_3$ 的内容增加1
  - ◆ 开销：2
- **SUB @12[ $R_1$ ], 4[ $R_0$ ]**
  - ◆ 将地址为 $(\text{contents}(12 + \text{contents}(R_1)))$ 的单元中的值减去 $\text{contents}(4 + \text{contents}(R_0))$ ，结果仍存放到地址为 $(\text{contents}(12 + \text{contents}(R_1)))$ 的单元中。
  - ◆ 开销：3

# 例：三地址语句 $a:=b+c$ 的代码生成

(1) MOV  $R_0, b$

ADD  $R_0, c$

指令开销为6

MOV  $a, R_0$

(2) MOV  $a, b$

ADD  $a, c$

指令开销为6

(3) 假定 $R_1$ 和 $R_2$ 中分别包含 $b$ 和 $c$ 的值， $b$ 没有下次引用：

ADD  $R_1, R_2$

MOV  $a, R_1$

指令开销为3

(4) 假定 $R_0$ 、 $R_1$ 和 $R_2$ 中分别存放了 $a$ 、 $b$ 和 $c$ 的地址：

MOV  $@R_0, @R_1$

ADD  $@R_0, @R_2$

指令开销为2

## 9.4.2 代码生成算法

### ■ 基本思路：

- ◆ 以基本块为单位，依次把三地址语句转换为目标语言语句
- ◆ 根据名字的下次引用信息，在基本块范围内，充分利用寄存器
  - 尽可能让变量的值保存在寄存器中
  - 后续的代码尽可能引用变量在寄存器中的值
  - 离开基本块时，把有关变量在寄存器中的值送到它的存储单元中

**MOV M, R**

# 数据结构

## ■ 寄存器描述符

- ◆ 记录每个寄存器当前保存的是哪些名字的值。
- ◆ 开始时，寄存器描述符指示所有的寄存器均为空。
- ◆ 代码生成过程中，每个寄存器在任一给定时刻可保留0个或多个名字的值。

## ■ 地址描述符

- ◆ 记录一个名字的当前值的存放位置，可能是：
  - 一个寄存器
  - 一个栈单元
  - 一个存储单元
  - 或这些地址的一个集合
- ◆ 这些信息用来确定对一个名字的寻址方式，可以存放在符号表中。

# 函数getreg(s)

- 输入：三地址语句  $x:=y \text{ op } z$
- 输出：存放 $x$ 值的地址 $L$ （ $L$ 或者是寄存器，或者是存储单元）
- 数据结构：寄存器描述符、名字的地址描述符
- 算法

```
switch 参数语句 {  
    case 形如 $x:=y \text{ op } z$ 的赋值语句：  
    case 形如 $x:= \text{op } y$  的赋值语句：  
        查看名字  $y$  的地址描述符;  
        if ( $y$ 的值存放在寄存器 $R$ 中) {  
            查看 $R$ 的寄存器描述符;  
            if ( $R$ 中仅有名字 $y$ 的值) {  
                查看名字 $y$ 的下次引用信息和活跃信息;  
                if (名字 $y$ 无下次引用, 且非活跃) return  $R$ ;  
            }  
        }  
}
```

# 函数getreg(s) (续)

```
else if (存在空闲寄存器R) return R;
else {
    查看名字x的下次引用信息;
    if (x有下次引用 || op 需要使用寄存器) {
        选择一个已被占用的寄存器R;
        for (R寄存器描述符中记录的每一个名字n)
            if (名字n的值仅在寄存器R中) {
                outcode('MOV' Mn, R);
                更新名字n的地址描述符为Mn; //Mn表示名字n的存储单元地址
            };
        return R;
    };
    else return Mx; //Mx表示名字x的存储单元地址
}
```



# 代码生成算法

输入：基本块的三地址语句 输出：基本块的目标代码

方法：

- (1) for (基本块中的每一条三地址语句) {
- (2)   switch 当前处理的三地址语句 {
- (3)     case 形如  $x:=y \text{ op } z$  的赋值语句:
- (4)        $L=\text{getreg}(i: x:=y \text{ op } z)$ ;
- (5)       查看名字  $y$  的地址描述符, 若  $y$  的值同时存放在存储器和寄存器中,  
      那么选择寄存器作为  $y$  值的当前存放位置  $y'$ ;
- (6)       if ( $y' \neq L$ ) outcode('MOV'  $L, y'$ );
- (7)       else 将  $L$  从  $y$  的地址描述符中删除( $y$  不在  $L$  中);
- (8)       查看名字  $z$  的地址描述符, 若  $z$  的值同时存放在存储器和寄存器中,  
      那么选择寄存器作为  $z$  值的当前存放位置  $z'$ ;
- (9)       outcode(op  $L, z'$ );
- (10)      更新  $x$  的地址描述符以记录  $x$  的值仅在  $L$  中;
- (11)      if ( $L$  是寄存器) 更新  $L$  的寄存器描述符以记录  $L$  中只有  $x$  的值;
- (12)      查看  $y/z$  的下次引用信息和活跃信息, 以及  $y/z$  的地址描述符;
- (13)      if ( $y/z$  没有下次引用, 在块出口处非活跃, 且当前值在寄存器  $R$  中) {
- (14)         从  $y/z$  的地址描述符中删除寄存器  $R$ ;
- (15)         从  $R$  的寄存器描述符中删除名字  $y/z$  ( $R$  不再包含  $y/z$  的值); }
- (16)    break;

# 代码生成算法（续1）

- (17) case 形如 $x:=op\ y$ 的赋值语句:
- (18)  $L=getreg(i: x:=op\ y);$
- (19) 查看名字  $y$  的地址描述符, 若 $y$ 的值同时存放在存储器和寄存器中, 那么选择寄存器作为 $y$ 值的当前存放位置 $y'$ ;
- (20) if ( $y'!=L$ ) outcode('MOV'  $L, y'$ );
- (21) else 将 $L$ 从  $y$  的地址描述符中删除( $y$ 不在 $L$ 中);
- (22) outcode( $op\ L$ );
- (23) 更新 $x$ 的地址描述符以记录 $x$ 的值仅在 $L$ 中;
- (24) if ( $L$ 是寄存器) 更新 $L$ 的寄存器描述符以记录 $L$ 中只有 $x$ 的值;
- (25) 查看  $y$  的下次引用信息和活跃信息, 以及 $y$ 的地址描述符;
- (26) if ( $y$ 没有下次引用, 在块出口处非活跃, 且当前值在寄存器 $R$ 中) {
- (27)     从  $y$  的地址描述符中删除寄存器 $R$ ;
- (28)     从 $R$ 的寄存器描述符中删除名字 $y$  ( $R$ 不再包含 $y$ 的值); }
- (29) break;

# 代码生成算法（续2）

```
(30) case 形如  $x:=y$  的赋值语句:
(31)   查看名字 $y$ 的地址描述符;
(32)   if ( $y$ 的值在寄存器 $R$ 中) {
(33)     在 $R$ 的寄存器描述符中增加名字 $x$ ;
(34)     更新名字 $x$ 的地址描述符以记录 $x$ 的值仅在 $R$ 中; }
(35)   else {
(36)      $L = \text{getreg}(i: x:=y)$ ;
(37)     if ( $L$ 是寄存器) {
(38)        $\text{outcode}('MOV' L, y')$ ; //  $y'$ 为 $y$ 值的当前存放位置
(39)       更新 $L$ 的寄存器描述符为名字 $x$ 和 $y$ ;
(40)       更新名字 $x$ 的地址描述符为 $L$ ;
(41)        $y$ 的地址描述符中增加寄存器 $L$ ;
(42)     }
(43)     else { // 此时,  $L$ 是名字 $x$ 的存储单元地址 $M_x$ 
(44)        $\text{outcode}('MOV' L, y')$ ; //  $y'$ 为 $y$ 值的当前存放位置
(45)       更新名字 $x$ 的地址描述符为 $M_x$ ;
(46)     }
(47)   } // end of if-else
(48)   break;
```

# 代码生成算法（续3）

```
(49) } // end of switch
(50) } // end of for, 基本块中的所有语句已经处理完毕
(51) for (在出口处活跃的每一个变量 x) {
(52) 查看x的地址描述符;
(53) if (x值的存放位置只有寄存器R)
(54)     outcode('MOV' Mx, R); // 将 x 的值存入它的内存单元中;
(55) } // end of for
```

- 在没有进行数据流分析的情况下，需要假定用户定义的所有变量在基本块出口处都是活跃变量
  - ◆ 在计算下次引用信息的算法中，第一步要把活跃信息域置为“活跃”

# 示例：

- 考虑赋值语句  $x := a + b * c - d$

- 三地址语句序列：

$t := b * c$

$u := a + t$

$v := u - d$

$x := v$

- 假定在基本块的出口， $x$ 是活跃的
- 有两个寄存器 $R_0$ 和 $R_1$

# 翻译过程

三地址语句	目标代码	寄存器描述器	地址描述器
		寄存器全空	a:Ma b:Mb c:Mc d:Md
t:=b*c	MOV R <sub>0</sub> , b MUL R <sub>0</sub> , c	R <sub>0</sub> : t	t: R <sub>0</sub>
u:=a+t	MOV R <sub>1</sub> , a ADD R <sub>1</sub> , R <sub>0</sub>	R <sub>0</sub> : t R <sub>1</sub> : u	t: R <sub>0</sub> u: R <sub>1</sub>
v:=u-d	SUB R <sub>1</sub> , d	R <sub>0</sub> : t R <sub>1</sub> : v	t: R <sub>0</sub> u: v: R <sub>1</sub>
x:=v		R <sub>1</sub> : v, x	x: R <sub>1</sub>
	MOV M <sub>x</sub> , R <sub>1</sub>		x: R <sub>1</sub> , M <sub>x</sub>

## 9.4.3 其他常用语句的代码生成

- 涉及变址的赋值语句
- 涉及指针的赋值语句
- 转移语句

# 1. 涉及变址的赋值语句

- 两种语句形式： $a:=b[i]$  和  $a[i]:=b$
- 假定数组采用静态存储分配
  - ◆ 基址已知，分别用  $a$  和  $b$  表示
  - ◆ 变量  $i$  存放的位置不同，生成的目标代码也不同
- 假定调用  $L:=\text{getreg}(a:=b[i])$  及  $L:=\text{getreg}(a[i]:=b)$  返回的是寄存器地址

$a:=b[i]$ 的代码生成过程如下。

- (1)  $L:=\text{getreg}(a:=b[i]);$
- (2) 查看名字  $i$  的地址描述符;
- (3) if (  $i$  的值在寄存器  $R_i$  中 )
- (4)      $\text{outcode}('MOV' \ L, b[R_i]);$
- (5) else if (  $i$  的值在内存单元  $M_i$  中 ) {
- (6)      $\text{outcode}('MOV' \ L, M_i);$
- (7)      $\text{outcode}('MOV' \ L, b[L]);$
- (8)     }
- (9) else if (  $i$  的值在栈单元  $d_i[SP]$  中 ) {
- (10)      $\text{outcode}('MOV' \ L, d_i[SP]);$
- (11)      $\text{outcode}('MOV' \ L, b[L]);$
- (12) }

$a[i]:=b$ 的代码生成过程如下。

- (1)  $L:=\text{getreg}(a[i]:=b);$
- (2) 查看名字  $i$  的地址描述符;
- (3) if (  $i$  的值在寄存器  $R_i$  中 )
- (4)      $\text{outcode}('MOV' \ a[R_i], b);$
- (5) else if (  $i$  的值在内存单元  $M_i$  中 ) {
- (6)      $\text{outcode}('MOV' \ L, M_i);$
- (7)      $\text{outcode}('MOV' \ a[L], b);$
- (8)     }
- (9) else if (  $i$  的值在栈单元  $d_i[SP]$  中 ) {
- (10)      $\text{outcode}('MOV' \ L, d_i[SP]);$
- (11)      $\text{outcode}('MOV' \ a[L], b);$
- (12) }



## 2. 涉及指针的赋值语句

- 两种语句形式： $a:=*p$  和  $*p:=a$
- 假定采用静态存储分配
  - ◆ 指针变量  $p$  存放的位置不同，生成的目标代码也不同
- 假定调用  $L:=\text{getreg}(a:=*p)$  及  $L:=\text{getreg}(*p:=a)$  返回的是寄存器地址

$a:=*p$ 的代码生成过程如下。

- (1)  $L:=\text{getreg}(a:=*p);$
- (2) 查看名字  $p$  的地址描述符;
- (3) if (  $p$  的值在寄存器  $R_p$  中 )
- (4)      $\text{outcode}('MOV' \ L, @R_p);$
- (5) else if (  $p$  的值在内存单元  $M_p$  中 ) {
- (6)      $\text{outcode}('MOV' \ L, M_p);$
- (7)      $\text{outcode}('MOV' \ L, @L);$
- (8)     }
- (9) else if (  $p$  的值在栈单元  $d_p[SP]$  中 ) {
- (10)      $\text{outcode}('MOV' \ L, d_p[SP]);$
- (11)      $\text{outcode}('MOV' \ L, @L);$
- (12) }

$*p:=a$ 的代码生成过程如下。

- (1)  $L:=\text{getreg}(*p:=a);$
- (2) 查看名字  $p$  的地址描述符;
- (3) if (  $p$  的值在寄存器  $R_p$  中 )
- (4)      $\text{outcode}('MOV' \ @R_p, a);$
- (5) else if (  $p$  的值在内存单元  $M_p$  中 ) {
- (6)      $\text{outcode}('MOV' \ L, M_p);$
- (7)      $\text{outcode}('MOV' \ @L, a);$
- (8)     }
- (9) else if (  $p$  的值在栈单元  $d_p[SP]$  中 ) {
- (10)      $\text{outcode}('MOV' \ L, d_p[SP]);$
- (11)      $\text{outcode}('MOV' \ @L, a);$
- (12) }

# 3. 转移语句

- 两种形式：
  - ◆ goto L
  - ◆ if E goto L
- 假设L所标识的三地址语句的目标代码首地址为L'。
- 对于goto L，生成的目标代码为 **JMP L'**
  - ◆ 如果在处理该 goto语句时，地址L'已经存在，则直接产生完整的目标指令即可；
  - ◆ 否则，需要先生成没有目标地址的JMP指令，待L'确定后再回填。

# 转移语句（续）

## ■ 对于 if E goto L，两种实现方式

- ◆ 当目标寄存器的值满足以下几个条件之一时产生转移：  
结果为负、为零、为正、非负、非零或非正。

- E的结果送入寄存器R

- 判断R的值为正、负、还是零

- E为真，则转移到 L

if a<b goto L:  
a-b ==>R  
CJ< L

- ◆ 利用条件码指示计算结果或存入寄存器的值为正、负、还是零。

- 如：if a<b goto L

CMP a, b  
CJ< L

## ■ 对于如下的语句序列

x:=a-b

if x<0 goto L

MOV R<sub>0</sub>, a  
SUB R<sub>0</sub>, b  
MOV x, R<sub>0</sub>  
CJ< L

# 小结

## ■ 设计代码生成程序时要考虑的问题

- ◆ 输入、输出
- ◆ 存储管理、寄存器分配
- ◆ 目标机器相关问题（指令、寄存器、编址方式、寻址能力、寻址模式等）
- ◆ 指令选择、计算顺序选择

## ■ 基本块和流图

- ◆ 基本块：具有原子性的语句序列
- ◆ 基本块的划分：入口语句的确定
- ◆ 流图：有向图，结点：基本块，边：控制流

# 小结（续）

- 下次引用信息
  - ◆ 作用
  - ◆ 计算方法
- 代码生成程序
  - ◆ 寄存器描述器
  - ◆ 地址描述器
  - ◆ 寄存器分配函数**getreg**
  - ◆ 代码生成算法