



中国科学技术大学
University of Science and Technology of China

代码生成

《编译原理和技术》

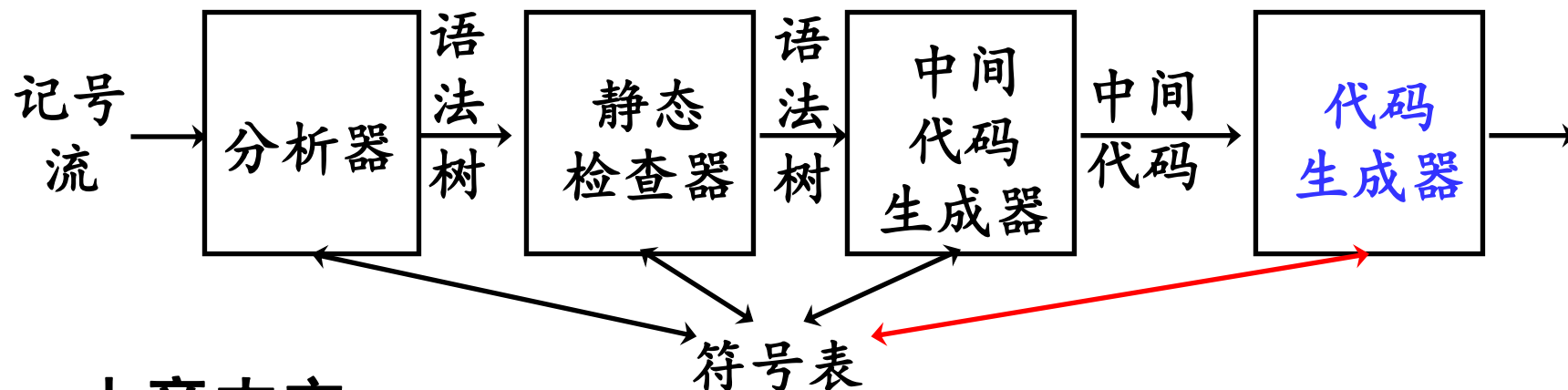
张昱 李诚

0551-63603804, yuzhang@ustc.edu.cn

中国科学技术大学
计算机科学与技术学院



本章内容



本章内容

- 一个简单的代码生成算法，将中间代码IR映射成为可以在目标机器上运行的指令序列
- 涉及目标机器指令选择，寄存器分配和计算次序选择等基本问题



中国科学技术大学
University of Science and Technology of China

1. 代码生成器设计中的问题

- ☐ 目标程序
- ☐ 指令选择
- ☐ 寄存器的分配和指派
- ☐ 计算次序



代码生成器设计中的问题

□ 目标程序(target program)

■ 绝对机器语言程序(absolute machine-language ...)

- 目标程序将装入到内存的固定地方
- 粗略地说，相当于现在的可执行目标模块（第11章介绍）

■ 可重定位目标模块(relocatable object module)

- 代码中含重定位信息，以适应重定位要求



代码生成器设计中的问题

□ 目标程序

■ 可重定位目标模块

.L7:

testl %eax,%eax

je .L3

testl %edx,%edx

je .L7

movl %edx,%eax

jmp .L7

.L3:

leave

ret

可重定位目标模块中，
需要有蓝色部分的重定
位信息



代码生成器设计中的问题

□ 目标程序

■ 绝对机器语言程序

- 目标程序将装入到内存的固定地方
- 粗略地说，相当于现在的可执行目标模块（第11章介绍）

■ 可重定位目标模块

- 代码中含重定位信息，以适应重定位要求
- 允许程序模块分别编译
- 调用其它先前编译好的程序模块



代码生成器设计中的问题

□ 目标程序

■ 绝对机器语言程序

■ 可重定位目标模块

- 代码中含重定位信息，以适应重定位要求
- 允许程序模块分别编译
- 调用其它先前编译好的程序模块

■ 汇编语言程序(assembly-language program)

- 免去编译器重复汇编器的工作
- 从教学角度，增加可读性



代码生成器设计中的问题

□ 指令的选择(instruction selection)

- 目标机器指令系统的性质决定了指令选择的难易程度，指令系统的统一性和完备性是重要的因素
- 指令的速度和机器特点是另一些重要的因素



代码生成器设计中的问题

- 不考虑目标程序的效率和指令的代价，逐条语句地产生代码，常常得到低质量的代码

例：三地址语句 $x = y + z$ (x , y 和 z 都静态分配)

MOV y , $R0$ /* 把 y 装入寄存器 $R0$ */

ADD z , $R0$ /* 把 z 加到 $R0$ 上 */

MOV $R0$, x /* 把 $R0$ 存入 x 中 */



代码生成器设计中的问题

语句序列 $a = b + c$

$d = a + e$

的一种目标代码如下：

MOV **b, R0**

ADD **c, R0**

MOV **R0, a**

MOV **a, R0**

ADD **e, R0**

MOV **R0, d**



代码生成器设计中的问题

语句序列 $a = b + c$

$d = a + e$

的一种目标代码如下：

MOV b, R0

ADD c, R0

MOV R0, a

MOV a, R0

ADD e, R0

MOV R0, d

由于a的值仍然存于寄存器R0中，因此该指令是冗余的。



代码生成器设计中的问题

语句序列 $a = b + c$

$d = a + e$

的一种目标代码如下：

MOV b, R0

ADD c, R0

MOV R0, a

~~MOV a, R0~~

ADD e, R0

MOV R0, d

如果a不再被使用，该指令也可以删除。



代码生成器设计中的问题

- 同一中间表示代码可以由多组指令序列来实现，但不同实现之间的效率差别是很大的。

例：语句 $a = a + 1$ 可以有两种实现方式

MOV	a,	R0
ADD	#1,	R0
MOV	R0,	a

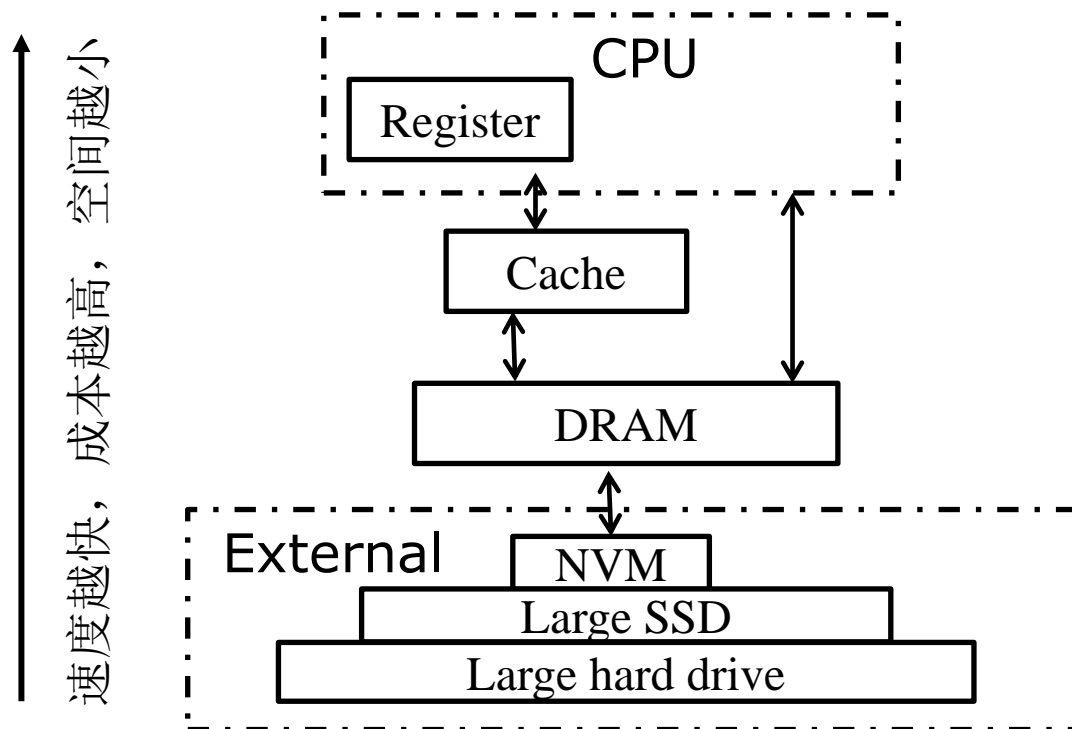
INC a

- 因此，生成高质量代码需要知道指令代价。



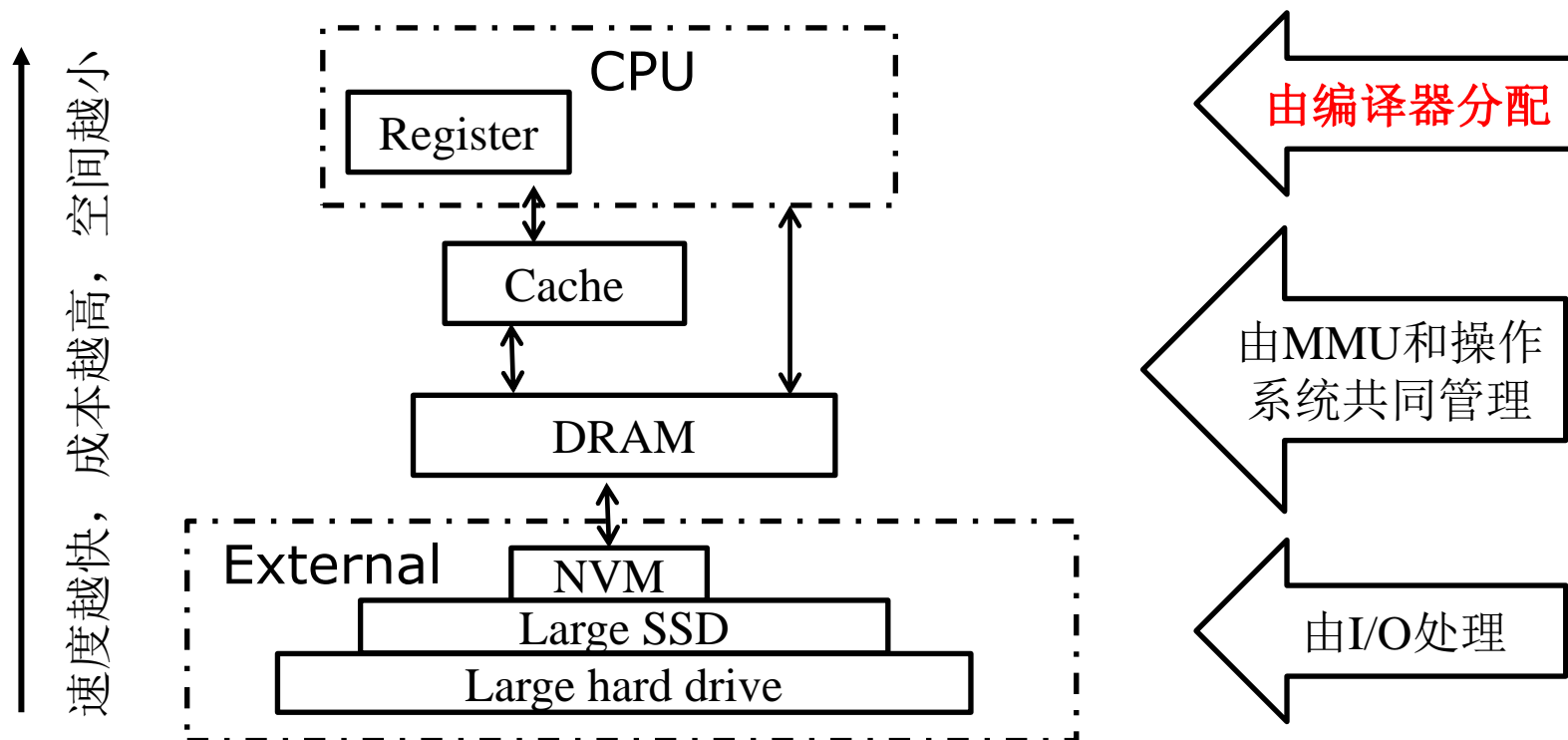
代码生成器设计中的问题

- 除了考虑指令的代价和序列长度外，我们还需要考虑运算对象和结果如何存储的问题。



代码生成器设计中的问题

- 除了考虑指令的代价和序列长度外，我们还需要考虑运算对象和结果如何存储的问题。





代码生成器设计中的问题

- 寄存器的合理使用：运算对象处于寄存器和处于内存相比，指令要短一些，执行也快一些
 - 寄存器分配(register allocation)
 - 选择驻留在寄存器中的一组变量
 - 寄存器指派(register assignment)
 - 挑选变量要驻留的具体寄存器



代码生成器设计中的问题

□ 计算次序的选择(evaluation order)

程序中计算的执行次序会影响目标代码的执行效率

□ 如，对表达式的计算而言，一种计算次序可能会比其它次序需要较少的寄存器来保存中间结果

□ 选择最佳计算次序是一个NP完全问题



中国科学技术大学
University of Science and Technology of China

2. 目标语言

- 目标机器指令集
- 指令代价



目标语言

□ 一个简单目标机器的指令系统

- 字节寻址，四个字节组成一个字
- 有 n 个通用寄存器 $R0, R1, \dots, Rn-1$
- 二地址指令： op 源，目的

MOV {源传到目的}

ADD {源加到目的}

SUB {目的减去源}



目标语言

□ 寻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	$c(\mathbf{R})$	$c + \textit{contents}(\mathbf{R})$	1
间接寄存器	$\ast\mathbf{R}$	$\textit{contents}(\mathbf{R})$	0
间接变址	$\ast c(\mathbf{R})$	$\textit{contents}(c + \textit{contents}(\mathbf{R}))$	1
直接量	$\#c$	c	1



目标语言

□ 例 指令实例

MOV R0, M

MOV 4(R0), M

4(R0)的值: *contents(4 + contents(R0))*

MOV *4(R0), M

4(R0)的值: *contents(contents(4 + contents(R0)))

MOV #1, R0



目标语言

□ 指令的代价(instruction costs)

■ 在上述简单的目标机器上，指令代价简化为

**1 + 指令的源和目的寻址模式(addressing mode)
的附加代价**



目标语言

□ 寻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	$c(\mathbf{R})$	$c + \textit{contents}(\mathbf{R})$	1
间接寄存器	$\ast\mathbf{R}$	$\textit{contents}(\mathbf{R})$	0
间接变址	$\ast c(\mathbf{R})$	$\textit{contents}(c + \textit{contents}(\mathbf{R}))$	1
直接量	$\#c$	c	1



目标语言

□ 指令代价简化为

1 + 指令的源和目的地址模式的附加代价

指令

代价

MOV R0, R1

MOV R5, M

ADD #1, R3

SUB 4(R0), *12(R1)



目标语言

□ 指令代价简化为

1 + 指令的源和目的地址模式的附加代价

指令	代价	
MOV R0, R1	1	寄存器
MOV R5, M	2	寄存器+内存
ADD #1, R3	2	常量+寄存器
SUB 4(R0), *12(R1)	3	变址+间接变址



目标语言

□ 例 $a = b + c$, a 、 b 和 c 都静态分配内存单元

■ 可生成

MOV b, R0

ADD c, R0

MOV R0, a

■ 也可生成

MOV b, a

ADD c, a



目标语言

□ 例 $a = b + c$, a 、 b 和 c 都静态分配内存单元

■ 可生成

MOV b, R0

ADD c, R0

代价= 6

MOV R0, a

■ 也可生成

MOV b, a

ADD c, a

代价= 6



目标语言

□ 例 $a = b + c$, 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, 则可生成

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, 则可生成

ADD $R2, R1$

MOV $R1, a$

代价= 3



3. 代码生成器的输入

- ☐ 中间代码IR的表示
- ☐ 基本块
- ☐ 流图
- ☐ 循环



三地址代码

□ 三地址代码(three-address code)

一般形式: $x = y \text{ op } z$

例 表达式 $x + y * z$ 翻译成的三地址语句序列是

$$t_1 = y * z$$

$$t_2 = x + t_1$$



基本块和流图

□ 基本块

连续的语句序列，控制流从它的开始进入，并从它的末尾离开，没有停止或分支的可能性（末尾除外）

□ 流图(flow graph)

用有向边表示基本块之间的控制流信息，基本块作为结点

(1) $\text{prod} = 0$

(2) $i = 1$

B_1

(3) $t_1 = 4 * i$

(4) $t_2 = a[t_1]$

(5) $t_3 = 4 * i$

(6) $t_4 = b[t_3]$

(7) $t_5 = t_2 * t_4$

(8) $t_6 = \text{prod} + t_5$

(9) $\text{prod} = t_6$

(10) $t_7 = i + 1$

(11) $i = t_7$

(12) if $i \leq 20$ goto (3)

B_2



循环

□ 如果下列条件成立，我们就说流图中的一个结点集合L是一个循环：

- 该集合中所有结点是强连通的。
- 该集合有唯一的入口结点。

□ 不包含其他循环的循环叫做内循环

(1) $\text{prod} = 0$

(2) $i = 1$

B_1

(3) $t_1 = 4 * i$

(4) $t_2 = a[t_1]$

(5) $t_3 = 4 * i$

(6) $t_4 = b[t_3]$

(7) $t_5 = t_2 * t_4$

(8) $t_6 = \text{prod} + t_5$

(9) $\text{prod} = t_6$

(10) $t_7 = i + 1$

(11) $i = t_7$

(12) if $i \leq 20$ goto (3)

B_2



下次引用信息

- 中间代码分析可以记录变量在整个计算过程中被使用的情况，以帮助寄存器的分配和释放
- 名字的引用(use)
 - 三地址码语句*i*为*x*赋值
 - 语句*j*将*x*作为运算对象，且*i*到*j*的控制流路径中无其他对*x*的赋值语句
 - 语句*j*引用了语句*i*计算的*x*值
- 对每一个基本块，反向扫描，对语句 $x = y \text{ op } z$ ，在符号表中记录*x*，*y*，*z*是否活跃或会被下次引用



4.一个简单的代码生成器

- ☐ 寄存器和地址的描述
- ☐ 代码生成算法
- ☐ 寄存器选择函数
- ☐ 为特殊语句产生代码



一个简单的代码生成器

□ 基本考虑：

- 依次考虑基本块的每个语句，为其产生代码
- 假定三地址语句的每种算符都有对应的目标机器算符
- 假定计算结果留在寄存器中尽可能长的时间，除非：
 - 该寄存器要用于其它计算，或者
 - 到基本块结束

为此，在生成代码过程中需要记录一些信息



一个简单的代码生成器

□ 寄存器描述和地址描述

例: 对 $a = b + c$

- 如果寄存器 R_i 含 b , R_j 含 c , 且 b 此后不再活跃
产生 `ADD R_j , R_i` , 结果 a 在 R_i 中
- 如果 R_i 含 b , 但 c 在内存单元, b 仍然不再活跃
产生 `ADD c , R_i` , 或者产生
`MOV c , R_j`
`ADD R_j , R_i`
- 若 c 的值以后还要用, 第二种代码较有吸引力



一个简单的代码生成器

□ 在代码生成过程中，需要跟踪寄存器的内容和名字的地址

■ 寄存器描述记住每个寄存器当前存的是什么，即在任何一点，每个寄存器保存若干个(包括零个)名字的值
例：

```
b = a           // 语句前，R0保存变量a的值
                // 不为该语句产生任何指令
                // 语句后，R0保存变量a和b的值
```



一个简单的代码生成器

- 在代码生成过程中，需要跟踪寄存器的内容和名字的地址
 - 寄存器描述记住每个寄存器当前存的是什么，即在任何一点，每个寄存器保存若干个（包括零个）名字的值
 - 名字（变量）的地址描述记住运行时每个名字的当前值可以在哪个场所找到。这个场所可以是寄存器、栈单元、内存地址、甚至是它们的某个集合
- 例：产生MOV c, R0后，c值可在R0和c的存储单元找到



一个简单的代码生成器

- 在代码生成过程中，需要跟踪寄存器的内容和名字的地址
 - 寄存器描述记住每个寄存器当前存的是什么，即在任何一点，每个寄存器保存若干个（包括零个）名字的值
 - 名字（变量）的地址描述记住运行时每个名字的当前值可以在哪个场所找到。这个场所可以是寄存器、栈单元、内存地址、甚至是它们的某个集合
 - 名字的地址信息存于符号表，另建寄存器描述表
 - 这两个描述在代码生成过程中是变化的



一个简单的代码生成器

□ 寄存器选择函数

■ 函数`getReg`返回保存 $x = y \text{ op } z$ 的 x 值的场所 L

- 如果名字 y 在 R 中，这个 R 不含其它名字的值，并且在执行 $x = y \text{ op } z$ 后 y 不再有下次引用，那么返回这个 R 作为 L
- 否则，如果有的话，返回一个空闲寄存器
- 否则，如果 x 在块中有下次引用，或者 op 是必须用寄存器的算符，那么找一个已被占用的寄存器 R (可能产生`MOV R, M`指令，并修改 M 的描述)
- 否则，如果 x 在基本块中不再引用，或者找不到适当的被占用寄存器，选择 x 的内存单元作为 L



一个简单的代码生成器

□ 代码生成算法

■ 对每个三地址语句 $x = y \text{ op } z$

- 调用函数 *getReg* 决定放 $y \text{ op } z$ 计算结果的场所 L
- 查看 y 的地址描述，确定 y 值当前的一个场所 y' 。如果 y 的值还不在于 L 中，产生指令 $\text{MOV } y', L$
- 产生指令 $\text{op } z', L$ ，其中 z' 是 z 的当前场所之一
- 如果 y 和/或 z 的当前值不再引用，在块的出口也不活跃，并且还在寄存器中，那么修改寄存器描述，使得不再包含 y 和/或 z 的值



一个简单的代码生成器

□ 赋值语句 $d = (a - b) + (a - c) + (a - c)$

■ 编译产生三地址语句序列：

$$t_1 = a - b$$

$$t_2 = a - c$$

$$t_3 = t_1 + t_2$$

$$d = t_3 + t_2$$



一个简单的代码生成器

语 句	生成的代码	寄存器描述	名字的地址描述
		寄存器空	
$t_1 = a - b$			
$t_2 = a - c$			
$t_3 = t_1 + t_2$			
$d = t_3 + t_2$			



一个简单的代码生成器

语 句	生成的代码	寄存器描述	名字的地址描述
		寄存器空	
$t_1 = a - b$	MOV a, R0 SUB b, R0	R0含t_1	t_1在R0中
$t_2 = a - c$			
$t_3 = t_1 + t_2$			
$d = t_3 + t_2$			



一个简单的代码生成器

语 句	生成的代码	寄存器描述	名字的地址描述
		寄存器空	
$t_1 = a - b$	MOV a, R0 SUB b, R0	R0含 t_1	t_1 在R0中
$t_2 = a - c$	MOV a, R1 SUB c, R1	R0含 t_1 R1含 t_2	t_1 在R0中 t_2 在R1中
$t_3 = t_1 + t_2$			
$d = t_3 + t_2$			



一个简单的代码生成器

语 句	生成的代码	寄存器描述	名字的地址描述
		寄存器空	
$t_1 = a - b$	MOV a, R0 SUB b, R0	R0含 t_1	t_1 在R0中
$t_2 = a - c$	MOV a, R1 SUB c, R1	R0含 t_1 R1含 t_2	t_1 在R0中 t_2 在R1中
$t_3 = t_1 + t_2$	ADD R1,R0	R0含 t_3 R1含 t_2	t_3 在R0中 t_2 在R1中
$d = t_3 + t_2$			



一个简单的代码生成器

语 句	生成的代码	寄存器描述	名字的地址描述
		寄存器空	
$t_1 = a - b$	MOV a, R0 SUB b, R0	R0含 t_1	t_1 在R0中
$t_2 = a - c$	MOV a, R1 SUB c, R1	R0含 t_1 R1含 t_2	t_1 在R0中 t_2 在R1中
$t_3 = t_1 + t_2$	ADD R1,R0	R0含 t_3 R1含 t_2	t_3 在R0中 t_2 在R1中
$d = t_3 + t_2$	ADD R1,R0	R0含d	d在R0中
	MOV R0, d		d在R0和内存中



一个简单的代码生成器

□ 前三条指令可以修改，使执行代价降低

修改前

MOV a, R0

SUB b, R0

MOV a, R1

SUB c, R1

. . .

修改后

MOV a, R0

MOV R0, R1

SUB b, R0

SUB c, R1

. . .



一个简单的代码生成器

□ 为特殊语句产生代码

■ 变址和指针语句

□ 变址与指针运算的三地址语句的处理和二元算符的处理相同

语句	i在寄存器Ri中		i在内存Mi中		i在栈中	
	代码	代价	代码	代价	代码	代价
$a = b[i]$	MOV b(Ri), R	2	MOV Mi, R MOV b(R), R	4	MOV Si(Rs), R MOV b(R), R	4
$b[i] = a$	MOV a, b(Ri)	3	MOV Mi, R MOV a, b(R)	5	MOV Si(Rs), R MOV a, b(R)	5



一个简单的代码生成器

□ 为特殊语句产生代码

■ 变址和指针语句

- 变址与指针运算的三地址语句的处理和二元算符的处理相同

■ 条件语句

- 根据寄存器的值是否为下面六个条件之一进行分支：负、零、正、非负、非零和非正
- 用条件码来表示计算的结果或装入寄存器的值是负、零还是正



一个简单的代码生成器

1、根据寄存器的值是否为下面六个条件之一进行分支：负、零、正、非负、非零和非正

□ 例 `if x < y goto z`

- 把`x`减`y`的值存入寄存器`R`
- 如果`R`的值为负，则跳到`z`



一个简单的代码生成器

2、用条件码的例子

□ 例：若 $\text{if } x < y \text{ goto } z$		则： $x = y + w$
的实现是：		$\text{if } x < 0 \text{ goto } z$
$\text{CMP } x, y$		的实现是：
$\text{CJ} < z$		$\text{MOV } y, R0$
		$\text{ADD } w, R0$
		$\text{MOV } R0, x$
		$\text{CJ} < z$



本章小结

