第11章 代码生成

重点： 代码生成器设计中的问题, 目标语言，一个简单的代码生成器, 寄存器的分配和指派

难点： 寄存器的分配和指派

第11章 代码生成

11.1 代码生成器设计中的问题

11.2 目标语言

11.3 一个简单的代码生成器

11.4 窥孔优化

11.5 寄存器分配与指派

11.6 本章小结

第11章 代码生成

- 代码生成是编译的最后一个阶段，由代码生成器完成。其任务是把中间代码转换为等价的、具有较高质量的目标代码，以充分利用目标机器的资源。当然，代码生成器本身也必须具有较高的运行效率。
- 目标代码可以是绝对地址的机器代码，或相对地址的机器代码，也可以是汇编代码。
- 本章用微型机的汇编指令来表示目标代码。

11.1 代码生成器设计中的问题

- 虽然代码生成器的具体实现依赖于目标机器的体系结构、指令系统和操作系统，但存储管理、指令选择、寄存器分配和计算顺序等问题却是设计各种代码生成器都要考虑的问题，本节讨论这类共性问题。

11.1.1 代码生成器的输入

- 代码生成器的输入包括中间代码和符号表信息，符号表信息主要用来确定中间代码中的变量所代表的数据对象的运行时地址。
- 假设在代码生成前，编译器的前端已经将源程序扫描、分析和翻译成为足够详细的中间代码，其中变量的值已经可以表示为目标机器能够直接操作的量(位、整数、实数、指针等)；
- 已经完成了必要的类型检查；
- 在需要的地方已经插入了类型转换符；明显的语义错误(如试图把浮点数作为数组下标)也都已经被检测出来了。

11.1.2 目标代码的形式

- 代码生成器的输出是目标代码。目标代码的形式主要有如下3种：
  - 绝对机器语言代码。所有地址均已定位，可以立即被执行。适于小程序的编译，因为它们可以迅速地被执行。
  - 可重定位的机器语言代码。允许分别将子程序编译成一组可重定位模块，再由连接装配器将它们和某些运行程序连接起来，转换成能执行的机器语言程序。好处是比较灵活，并能利用已有的程序资源，代价是增加了连接和装配的开销。
  - 汇编语言代码。生成汇编语言代码后还需要经过汇编程序汇编成可执行的机器语言代码，但其好处是简化了代码生成过程并增加了可读性。

11.1.3 指令选择

- 所谓指令选择是指寻找一个合适的机器指令序列来实现给定的中间代码。
- 目标机器指令系统的性质决定了指令选择的难易程度
  - 指令系统的一致性和完备性是两个重要的因素
  - 特殊机器指令的使用和指令速度是另一些重要的因素

11.1.3 指令选择

- 若不考虑目标程序的效率，指令的选择将非常简单：
  - 如：三地址语句x := y + z翻译成如下代码序列：（x，y和z都是静态分配）
  - **MOV y, R0 /* 把y装入寄存器R0 */**
  - **ADD z, R0 /* z加到R0上 */**
  - **MOV R0, x /* 把R0存入x中 */**
- 逐个语句地产生代码，常常得到低质量的代码

11.1.3 指令选择

语句序列 **a := b + c**

d := a + e

的代码如下

```
MOV     b,   R0
ADD     c,   R0
MOV     R0,  a     -- 若a不再使用，第三条也多余
MOV     a,   R0    -- 多余的指令
ADD     e,   R0
MOV     R0,  d
```

11.1.3 指令选择

- 如果目标机器有加l指令(INC)，则a :＝a+1的最有效实现是:

  INC a

而不是

MOV a, R0

ADD #1, R0

MOV R0, a

11.1.4 寄存器分配

- 将运算对象放在寄存器中的指令通常要比将运算对象放在内存中的指令快且短，因此，要想生成高质量的目标代码，必须充分使用目标机器的寄存器，寄存器的使用包括：
  - 寄存器分配：为程序的某一点选择驻留在寄存器的一组变量
  - 寄存器指派：确定变量将要驻留的具体寄存器

11.1.4 寄存器分配

- 选择最优的寄存器指派方案是一个NP完全问题，如果考虑到目标机器的硬件和(或)操作系统对寄存器的使用约束，该问题还会进一步复杂。有关寄存器分配和指派的策略将在11.5节再进行详细讨论。

11.1.5 计算顺序选择

- 计算执行的顺序同样会影响目标代码的效率。后面将会看到，某些计算顺序比其它顺序需要较少的寄存器来保存中间结果，因而其目标代码的效率也要高。
- 选择最佳计算顺序也是一个NP完全问题。为简单起见，只讨论如何按给定的三地址码的顺序生成目标代码。

11.2 目标语言

11.2.1 目标机模型

选择可作为几种微机代表的寄存器机器

- 字节寻址，四字节组成一个字，具有n个通用寄存器R0, R1, …, Rn-1。
- 二地址指令： op 源，目的

  MOV {将源移到目的中}

  ADD {将源加到目的中}

  SUB {在目的中减去源}

11.2 目标语言

寻址模式和它们的汇编语言形式及相关开销

寻址模式	汇编形式	地址	增加的开销
绝对地址	M	M	1
寄存器	R	R	0
下标	*c*(R)	*c* + *contents*(R)	1
间址寄存器	*R	*contents*(R)	0
间址下标	**c*(R)	*contents*(*c*+*contents*(*R*))	1
直接量	#*c*	*c*	1

11.2 目标语言

指令实例

MOV R0, M

MOV 4(R0), M

***contents*(4 + *contents*(R0))**

MOV *4(R0), M

***contents*(*contents*(4+*contents*(R0)))**

MOV #1, R0

11.2 目标语言

11.2.2 程序和指令的开销

指令开销:= 1+源和目的寻址模式的附加开销

指令	开销
MOV R0，R1	1
MOV R5，M	2
ADD #1，R3	2
SUB 4(R0), *12(R1)	3

程序和指令的开销

a := b + c， **a、b和c都静态分配内存单元**

MOV b, R0

ADD c, R0 **开销= 6**

MOV R0, a

程序和指令的开销

a := b + c， **a、b和c都静态分配内存单元**

MOV b, R0

ADD c, R0 **开销= 6**

MOV R0, a

MOV b, a

ADD c, a **开销= 6**

程序和指令的开销

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

MOV *R1, *R0

ADD *R2, *R0　　开销= 2

程序和指令的开销

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

MOV *R1, *R0

ADD *R2, *R0　　开销= 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

ADD R2, R1

MOV R1, a　　开销= 3

11.2.3 变量的运行时刻地址

- 存储分配策略以及过程的活动记录中局部数据的布局决定了如何访问变量所对应的内存位置
- 前面假设三地址码中的变量实际上是一个指向符号表表项的指针，在代码生成阶段，变量必须被替换为运行时的内存地址
- 例11.1 考虑三地址码x:=0，假设处理完过程的声明部分之后，x在符号表中的相对地址为12

11.2.3 变量的运行时刻地址

- 如果x被分配在一个地址从static开始的静态内存区域中，则x的运行时刻地址为static+12。如果静态区从地址100开始， x:=0的目标代码为：

  MOV #0, 112。

- 如果采用栈式存储分配策略，则只有等到运行时刻才能知道一个过程的活动记录位置。此时， x:=0的目标代码为：

  MOV #0, 12(SP)。

11.3 一个简单的代码生成器

- 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
  - 到达基本块末尾
- 后续的目标代码也要尽可能地引用保存在寄存器中的变量值

11.3.1 后续引用信息

- 为了在代码生成过程中能充分合理地使用寄存器，应把基本块中还会再被引用的变量的值尽量保留在寄存器中，而把基本块内不会再被引用的变量所占用的寄存器及早释放。为此，对于每个形如a:=b op c的三地址码，需要知道变量a、b和c在基本块内是否还会再被引用以及会在哪里被引用，这些信息称为后续引用信息。

11.3.1 后续引用信息

- 如果在一个基本块中，语句i定义了x，语句j要引用x的值，且从i到j之间没有x的其它定义，则称i中x的定义能够到达j。从j所能到达的每一个x的引用点都称为i点定义的变量x的后续引用信息，所有这样的j所组成的引用链则称为变量x的后续引用信息链。

后续引用信息的计算

(1) 初始时，将基本块中各变量的符号表表项的后续引用信息域置为“无后续引用”，并根据该变量在基本块的出口是否活跃，将其活跃信息域置为“活跃”或“不活跃”；

(2) 从基本块出口向入口反向扫描，并对每个形如**i:a:=b op c**的三地址码依次执行如下操作：

① 将符号表中**a**的后续引用信息和活跃信息附加到**i**上

② 将符号表中**a**的后续引用信息和活跃信息分别置为“无后续引用”和“不活跃”

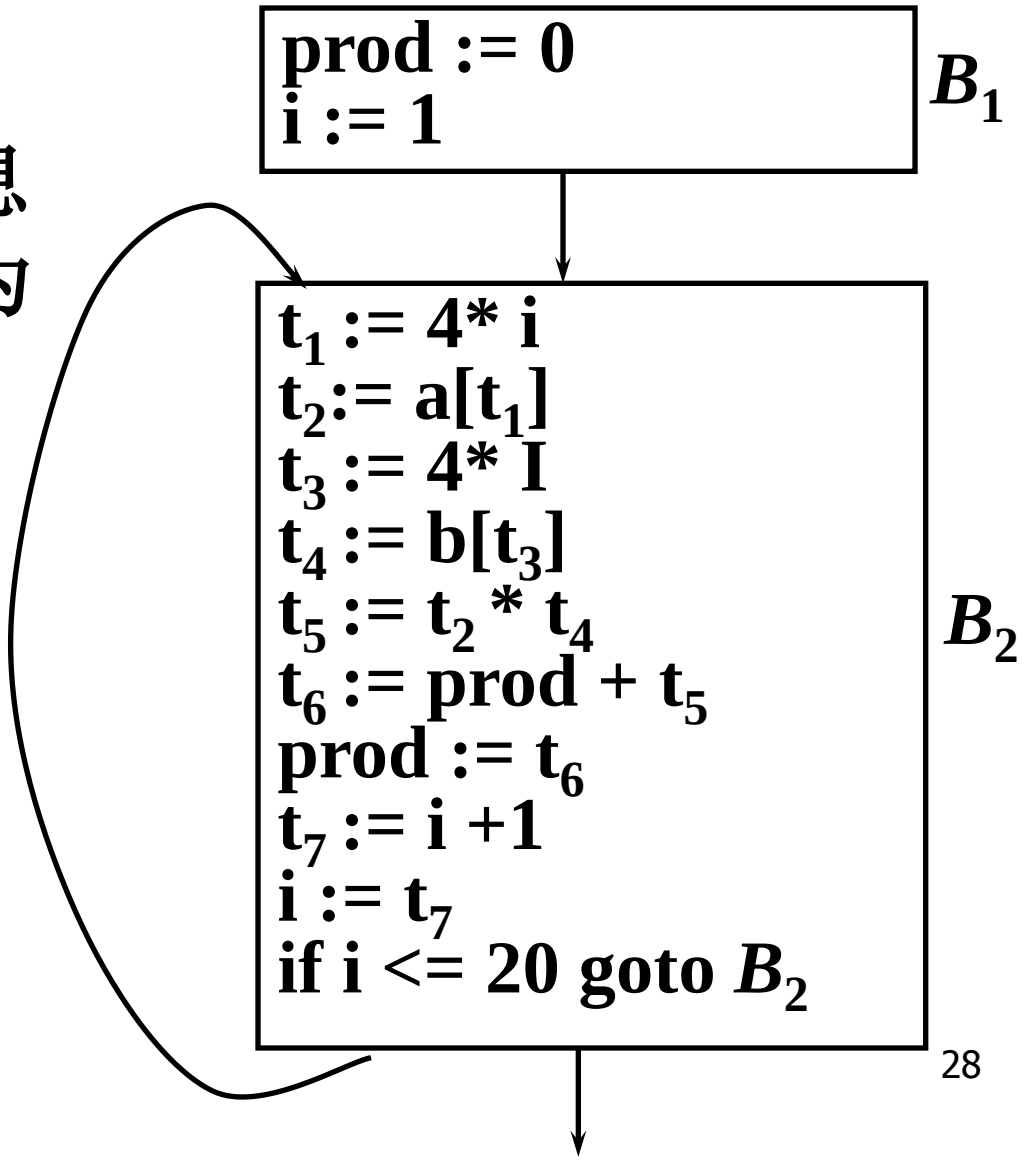
③ 将符号表中**b**和**c**的后续引用信息和活跃信息附加到**i**上

④ 将符号表中变量**b**和**c**的后续引用信息均置为**i**，活跃信息均置为“活跃”

注意，上述次序不能颠倒，因为**b**和**c**也可能就是**a**。此外，因为过程调用可能带来副作用，所以在划分基本块时将过程调用也作为基本块的入口。

11.3 一个简单的代码生成器

在没有收集全局信息前，暂且以基本块为单位来生成代码



11.3.2 寄存器描述符与地址描述符

例：对a := b + c

- 如果寄存器Ri含b，Rj含c，且b此后不再活跃
  - 产生ADD Rj, Ri，结果a在Ri中
- 如果Ri含b，但c在内存单元，b仍然不再活跃
  - 产生ADD c, Ri，或者
  - MOV c, Rj
    ADD Rj, Ri

若c的值以后还要用,第二种代码比较有吸引力

11.3.2 寄存器描述符与地址描述符

在代码生成过程中，需要跟踪寄存器的内容和名字的地址

- **寄存器描述符**记录每个寄存器当前存的是什么
  - 在任何一点，每个寄存器保存若干个(包括零个)名字的值
- **名字的地址描述符**记录运行时每个名字的当前值存放的一个或多个位置
  - 该位置可能是寄存器、栈单元、内存地址或者是它们的某个集合
  - 这些信息可以存放在符号表中
- 这两个描述符在代码生成过程中是变化的。

11.3.3 代码生成算法

对每个三地址语句i: x := y *op* z

- 调用函数*getreg*(*i:x:= y op z*)确定可用于保存y *op* z的计算结果的位置L
- 查看y的地址描述符以确定y值当前的一个位置y′。如果y值当前既在内存单元中又在寄存器中，则选择寄存器作为y'。如果y的值还不在L中，则生成指令MOV y′，L
- 生成指令*op* z′，L，其中z′是z的当前位置之一
- 如果y和/或z的当前值没有后续引用，在块的出口也不活跃，并且还在寄存器中，则修改寄存器描述符以表示在执行了x:=y op z 之后，这些寄存器分别不再包含y和(或)z的值。

11.3.3 代码生成算法

寄存器选择函数*getreg*

函数*getreg*返回保存x := y *op* z的x值的位置L

- 如果变量y在R中,且R不含其它变量的值,并且在执行x:=y *op* z后y不会再被引用,则返回R作为L。
- 否则，返回一个空闲寄存器，如果有的话
- 否则，如果x在块中还会再被引用，或者*op*是必须使用寄存器的算符，则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的地址描述符)
- 否则，如果x在基本块中不会再被引用，或找不到适当的被占用寄存器，则选择x的内存单元作为L。

11.3.3 代码生成算法

赋值语句d := (a – b) + (a –c) + (a – c)

编译产生的三地址码序列为：

$t_1 := a - b$

$t_2 := a - c$

$t_3 := t_1 + t_2$

$d := t_3 + t_2$

d在基本块的出口是活跃的。

11.3.3 代码生成算法

语 句	生成的代码	寄存器描述符	名字地址描述符
		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 MOV R0, d	R0含d	d在R0中 d在R0和内存中

11.3.3 代码生成算法

前三条指令可以修改，使执行代价降低

```
MOV a, R0
SUB b, R0
MOV a, R1
SUB c, R1
. . .
```

```
MOV a, R0
MOV R0，R1
SUB b, R0
SUB c, R1
. . .
```

11.3.4 常用三地址码的代码生成

(1)复制： a:=b

- 如果b的当前值在寄存器R中，则不必生成代码，只要将a添加到R的寄存器描述符中，并把a的地址描述符置为R即可。
- 如果b在基本块中不会再被引用且在基本块的出口也不活跃，则还要从R的寄存器描述符中删除b，并从b的地址描述符中删除R。
- 但若b的当前值只在内存单元中，如果只是简单地将a的地址描述符置为b的内存地址，那么，若不对a的值采取保护措施，a的值将会为b的再次定义所影响。此时，生成一条形如MOV b, R的指令会较为稳妥。

11.3.4 常用三地址码的代码生成

(2) 一元运算： a:= op b

- 与二元运算的处理类似。

(3) 数组元素引用： a:=b[i]。

- 假设a在基本块中还会再被引用，而且寄存器R是可用的，则将a保留在寄存器R中。于是，如果i的当前值不在寄存器中，则生成如下指令序列：

  MOV i, R

  MOV b(R), R　　开销=4

- 如果i的当前值在寄存器R*i*中，则生成如下指令：

  MOV b(R*i*), R　　开销=2

11.3.4 常用三地址码的代码生成

⑷ 数组元素赋值：a[i]:=b

- 假设a是静态分配的。如果i的当前值不在寄存器中，则生成如下指令序列：

  MOV i, R

  MOV b, a(R)　　开销=5

- 如果i的当前值在寄存器R*i*中，则生成如下指令：

  MOV b, a(R*i*)　　开销=3

11.3.4 常用三地址码的代码生成

(5) 指针引用： a:=*p

- 同样假设a在基本块中还会再被引用，而且寄存器R是可用的，则将a保留在寄存器R中。于是，如果p的当前值不在寄存器中，则生成如下指令：

  MOV p, R

  MOV *R, R　　开销=3

- 如果p的当前值在寄存器R*i*中，则可生成如下指令：

  MOV *R*i*, a　　开销=2

11.3.4 常用三地址码的代码生成

(6) 指针赋值：*p:=a

- 假设a是静态分配的。如果p的当前值不在寄存器中，则生成如下指令：

  MOV p, R

  MOV a, *R　　开销=4

- 如果p的当前值在寄存器R*i*中，则可生成如下指令：

  MOV a, *R　　开销=2

11.3.4 常用三地址码的代码生成

⑺ 无条件转移：goto L

- 假设L为三地址语句的序号，则生成指令JMP L'。
- 其中，L'为序号为L的三地址语句的目标代码首址。

11.3.4 常用三地址码的代码生成

(8) 条件转移： if a rop b goto L

- 同样假设L为三地址语句的序号，则生成如下的指令序列：

  CMP a, b

  CJrop L'

- 其中，L'的含义与(7)中相同，CMP为比较指令，Cjrop为条件码跳转指令，CMP根据rop取>、<或=而将条件码分别置为正、负或零。如果a和/或b的当前值在寄存器中，则在生成目标代码时应尽量使用寄存器寻址模式。

11.4 窥孔优化

- 窥孔优化是一种简单有效的局部优化方法，它通过检查目标指令中称为窥孔的短序列，用更小更短的指令序列进行等价代替，以此来提高目标代码的质量。
- 窥孔是放在目标程序上的一个移动的小窗口。孔中的代码不需要是连续的。
- 该技术的特点是每次优化后的结果可能又为进一步的优化带来了机会。所以有时会对目标代码重复进行多遍优化。下面介绍几种典型的窥孔优化技术。

11.4 窥孔优化

- **11.4.1 冗余指令消除**
- 如果遇到如下的指令序列：

  (1) **MOV R0, a**

  (2) **MOV a, R0** **(11.1)**

  则可以删除指令(2)。但是，如果(2)带有标号，通常是不能删除的。

11.4 窥孔优化

- **11.4.2** 不可达代码消除
- 删除紧跟在无条件跳转指令后的无标号指令称为不可达代码删除。

11.4 窥孔优化

- **11.4.3 强度削弱**
- 强度削弱是指在目标机器上用时间开销小的等价操作代替时间开销大的操作。例如用x*x实现x2要比调用一个指数过程快很多。用移位操作实现乘以2或除以2的定点运算要更快一些。用乘法实现(近似)浮点除法也可能会更快一些。

11.4 窥孔优化

- **11.4.4 特殊机器指令的使用**
- 为了提高效率，目标机器有时会使用一些硬件指令来实现某些特定的操作。例如，有一些机器具有自动加1和自动减1的寻址模式。这些模式的运用可以大大提高参数传递过程中压栈和出栈的代码质量，它们还可以用在形如i:=i+1的语句的代码中。

11.4 窥孔优化

- **11.4.5 其他处理**
- 也可以利用其他一些途经进行窥孔优化。例如，通过删除那些不必要的连续转移；利用代数恒等性质删除形如**x := x+0**或**x := x*1**的代码的代数化简。

11.5 寄存器分配与指派

- 由于只涉及寄存器运算对象的指令要比那些涉及内存运算对象的指令短且快，因此有效地利用寄存器非常重要。
- 寄存器分配的任务是为程序的某一点选择应该驻留在寄存器中的一组变量
- 寄存器指派则负责挑出变量将要驻留的具体寄存器。

11.5.1 全局寄存器分配

- 由于程序的大多数时间都花在内层循环上，因此一种比较自然的全局寄存器分配方法是在整个循环中将经常引用的值保存在固定的寄存器中。
- 假设已经利用第10章的技术找出了流图中的循环结构，而且知道基本块中计算出的哪些值要在该块外被引用，则有一种简单的全局寄存器分配策略，那就是分配固定数目的寄存器来保存每个内部循环中最活跃的值。
- 不同循环选中的值也会有所不同。
- 未被分配的寄存器可用于存放11.4节讨论的基本块的局部值。
- 该方法的缺点是：固定的寄存器数目对全局寄存器分配来说可能不够用，但实现简单。

11.5.2 引用计数

- 如果*x*在寄存器中，则对*x*的每次引用都将节省一个单元的开销。于是可以采用一种简单的方法来确定执行循环L时将变量*x*保存在寄存器中所节省的开销。
- 计算循环*L*中为*x*分配寄存器所节省开销的近似公式：

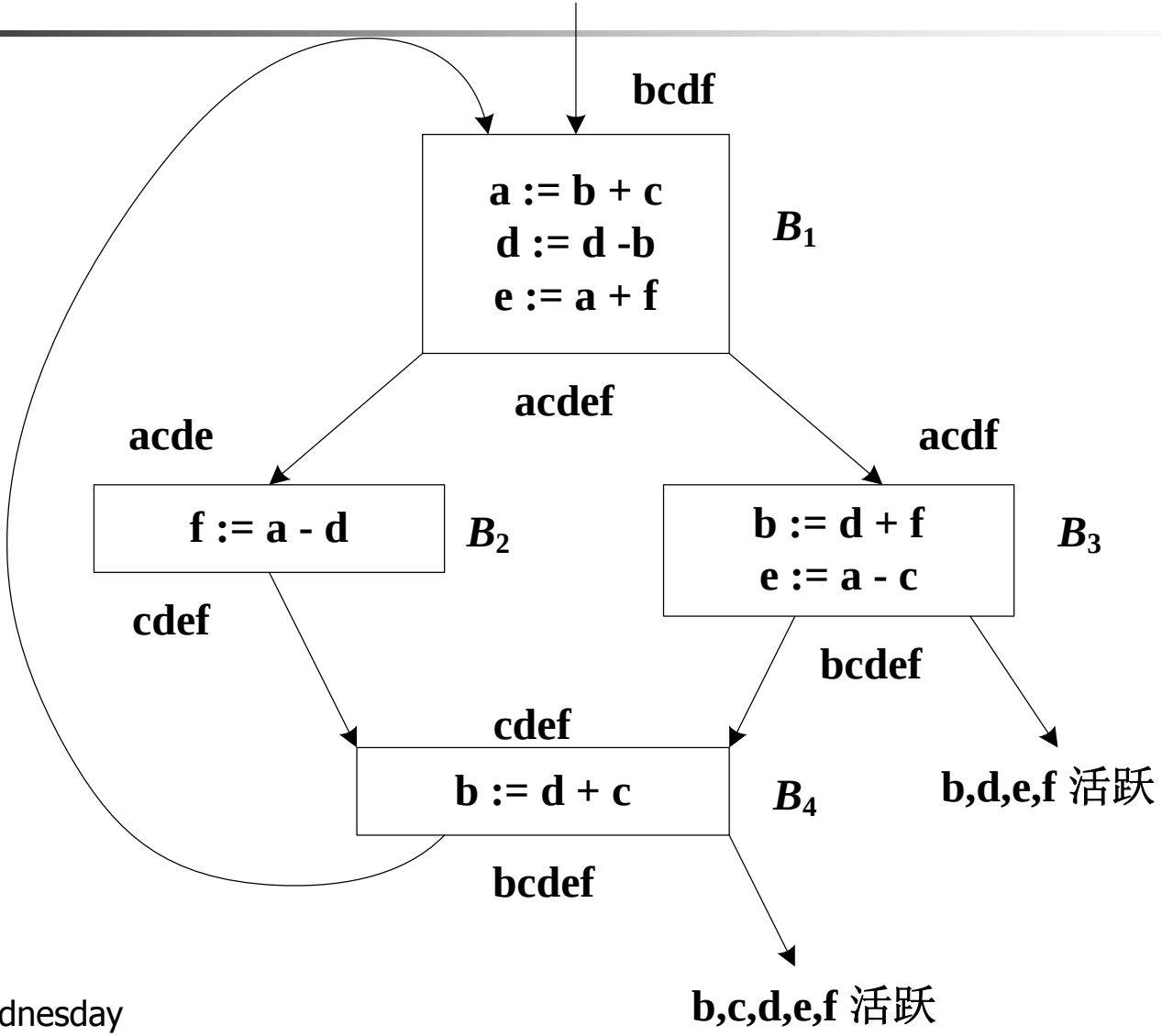
$$\sum_{L\text{中的块}B}(use(x,B)+2*live(x,B)) \qquad \textbf{(11.5)}$$

- 其中，***use*(*x*, *B*)**是定义*x*之前，块*B*中对*x*的引用次数。如果*x*在*B*的出口处是活跃的，而且*B*中含有对x的赋值，则***live*(*x*, *B*)**为1，否则***live*(*x*, *B*)**为0。注意，(11.5)是个近似公式。这是因为循环中的块的迭代次数可能是不一样的，而且我们还假设循环会迭代许多次。

11.5.2 引用计数

- 例11.3 考虑图11.2中内层循环中的基本块，块中的跳转语句均已删除。假设用寄存器R0、R1和R2来保存循环中的值。为方便起见，图中还列出了在每个块入口和出口活跃的变量。e和f在B_1的末尾都是活跃的，但只有e在B_2的入口是活跃的，只有f在B_3的入口是活跃的。一般地，块末尾活跃的变量是其后继块入口活跃变量的并集。

例11.3


bcdf
a := b + c
d := d -b
e := a + f
B_1
acdef
acde
acdf
f := a - d
B_2
b := d + f
e := a - c
B_3
cdef
bcdef
cdef
b := d + c
B_4
b,d,e,f 活跃
bcdef
b,c,d,e,f 活跃

例11.3

- 首先计算x=a时(11.5)的值，由于a在B_1的出口是活跃的，B_1中还含有对a的赋值，而且a在B_2、B_3和B_4的出口不活跃，因此

$$\sum_{L\text{中的}B} 2^{2*live(a,B)+use(a,B)}$$

因为a是在任何引用之前于B_1中定义的，所以$use(a, B_1)=0$。同理，$use(a, B_2)=use(a, B_3)=1$，而且，$use(a, B_4)=0$，于是，

$$\sum_{L\text{中的}B} use(a,B)$$

综上，x=a时(11.5)式的值为4。亦即，将a存入某个全局寄存器可以节省4个单元的开销。由于x=b、c、d、e和f时(11.5)式的值分别为6、3、6、4和4，因此可以将a、b和d放入寄存器R0、R1和R2。使用R0存放e或f而不是a可以节省相同的开销。

例11.3

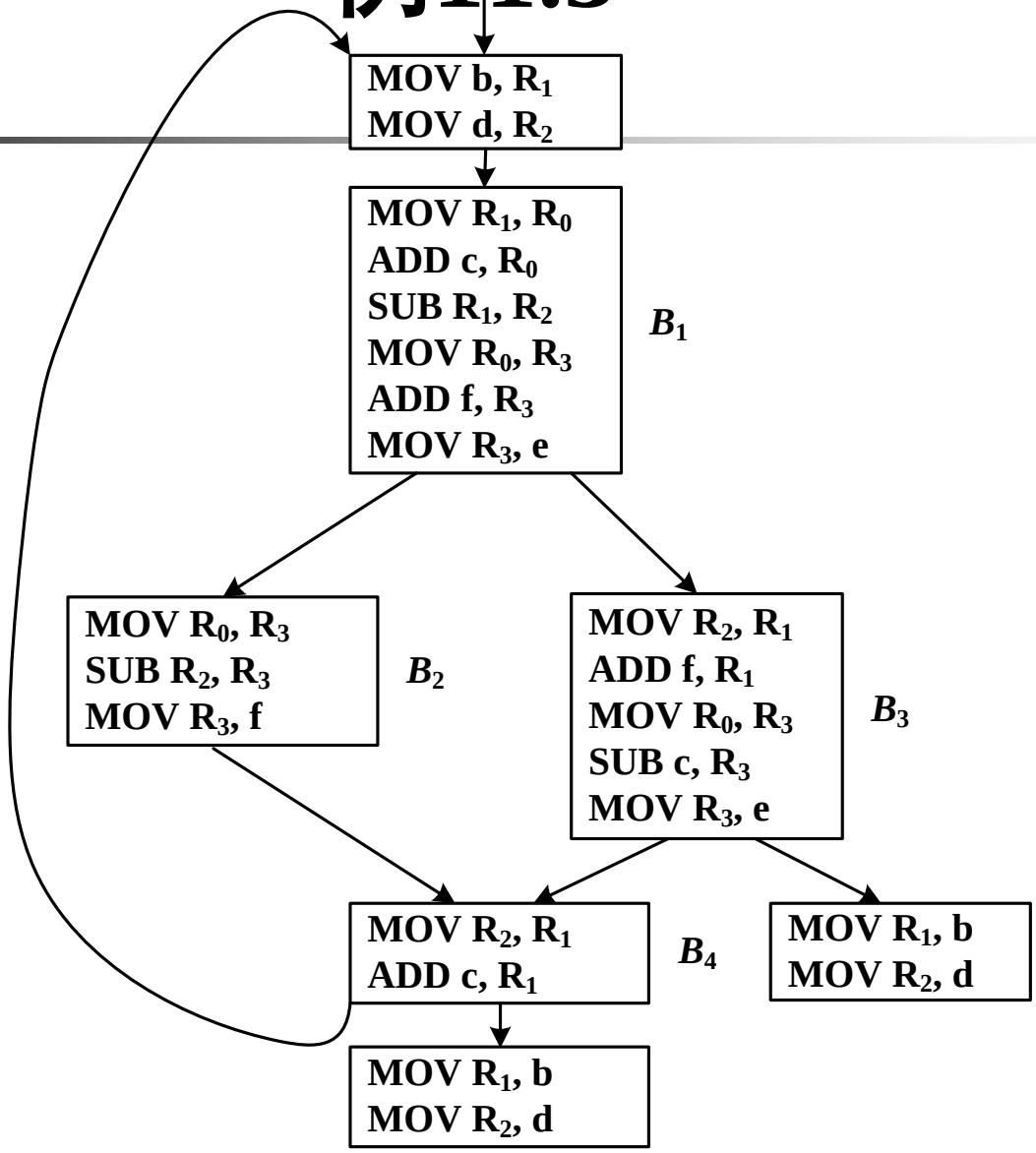


图11.3 使用全局寄存器分配的代码序列

11.5.3 外层循环的寄存器指派

- 为内层循环分配了寄存器并生成代码之后，可以将同样的方法应用到外层循环上。
- 如果外层循环L_1包含内层循环L_2，则在L_2中分配了寄存器的变量不必再在L_2-L_1中分配寄存器。
- 如果变量x是在循环L_1中而不是在L_2中分配了寄存器，则必须在L_2的入口处保存x并在离开L_2进入块L_1-L_2之前装载x。

11.5.3 外层循环的寄存器指派

- 如果在L_2而不是L_1中为x分配了寄存器，则必须在L_2的入口装载x，并在L_2的出口保存x。
- 如果计算时需要寄存器但所有可用的寄存器均被占用，则必须将某个正被使用的寄存器中的内容存放(溢出)到内存中以释放一个寄存器。
- 图染色法是一种简单的用于寄存器分配和寄存器溢出管理的系统技术。

本章小结

1. 目标代码的生成需要尽力开发利用机器提供的资源，特别是根据开销选用恰当的指令和寄存器，以提高其执行效率；
2. 稀缺资源寄存器的有效利用涉及到后续引用问题，寄存器描述符用来记录每个寄存器当前的内容；地址描述符记录运行时存放变量当前值的一个或多个位置，用来确定对变量的存取方式；
3. 使用引用计数能够良好地实现寄存器的分配和指派；
4. 不同形式的三地址码对应不同的目标代码，且具有不同的执行代价；
5. 不可达和冗余指令删除、控制流优化、强度削弱、代数化简、特殊指令使用等都是有效的窥孔优化方法；