代码优化及目标代码生成

代码优化概述

* 代码优化的任务

对中间代码及目标代码进行等价变换，使变换后的代码质量更高。

* 代码优化的要求

- 等价变换
- 提高代码的运行速度
- 减少代码占用的空间

* 代码优化的原则

- 等价原则、有效原则、合算原则

中间代码优化种类

* **与机器有关的优化**
  - 结合目标计算机的特点进行优化。
  - 主要有寄存器优化、指令优化、处理机优化等

* **与机器无关的优化**
  - 不针对目标计算机的特点进行优化
  - 主要有删除公共子表达式、复写传播、代码外提、删除无用代码等

中间代码优化种类

* 中间代码优化
  - 局部优化(基本块优化)

    在基本块内进行的优化，主要技术有：常数合并与传播、删除公共子表达式、复制传播、削弱计算强度、改变计算次序等
  - 循环优化

    在循环语句所生成的中间代码序列上进行的优化，主要技术有：循环展开、代码外提、削弱计算强度、删除归纳变量等
  - 全局优化

    在非线性程序段上（包含多个基本块）进行的优化

目标代码优化种类

* **目标代码优化**

  窥孔优化：在目标代码上进行局部改进的优化

  删除冗余指令、控制流优化、代数化简等

局部优化（基本块优化）

基本块是一段连续的顺序执行的语句序列，只有一个入口和一个出口。

如果一段程序中含有中断或者分支结构，那么可以按照下面的算法划分为多个基本块。

1、确定基本块的入口语句。规则如下：
   a）代码序列的第1条语句；
   b）能由条件转移语句或者无条件转移语句转移到的语句是入口语句；
   c）紧随条件转移语句后面的语句是入口语句。
2、依据1中得到的每一条入口语句，确定其基本块。它由该入口语句与下一条入口语句（不包含该入口语句）之间的语句序列构成；或者由该入口语句与下一条转移语句（包含该转移语句）之间的语句序列构成；或者由该入口语句与一条停语句（包含该停语句）之间的语句序列构成。
3、删除那些未出现在任何基本块中的语句。

基本块划分

有如下中间代码序列，请划分基本块。

（1）查找入口语句：

(1)、(4)、(8)、(6)

（2）划分基本块

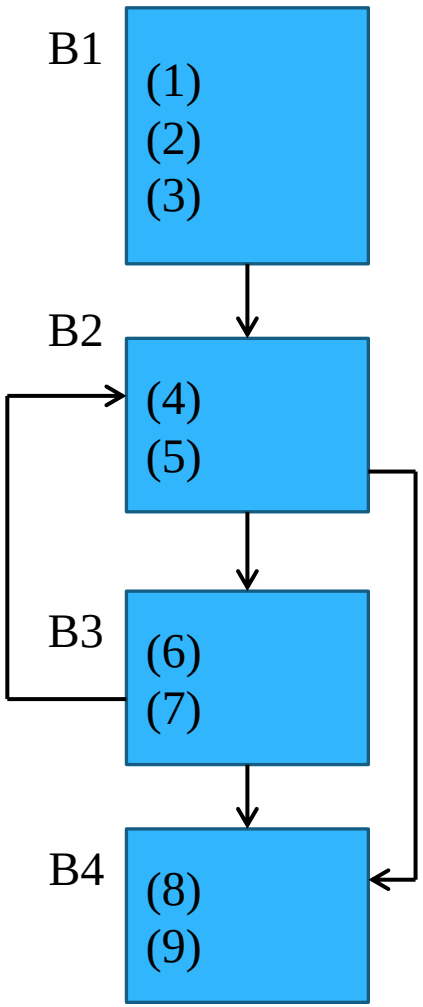
B1: (1) ~ (3)

B2: (4) ~ (5)

B3: (6) ~ (7)

B4: (8) ~ (9)

（3）连接各基本块



流图

```
(1)      read (C)
(2)      A:= 0
(3)      B:= 1
(4) L1: A:=A + B
(5)      if  B>= C  goto  L2
(6)      B:=B+1
(7)      goto  L1
(8) L2:  write (A)
(9)      halt
```

基本块优化

- **删除公共子表达式**：在一个基本块中，当第一次对表达式E求值后，如果E的运算对象都没有改变，就再次对E求值，则除E的第一次求值外，其他的都是冗余的公共表达式。
- **复写传播**：在对变量赋值后a := b。如果在该赋值语句后的语句序列中，a和b的值都没有变化，则对a的引用可以用b来代替。
- **删除无用代码**：
  - 删除不可到达代码
  - 删除死代码。
- **代数恒等变换**：
  - 合并已知量：将在编译时可计算出值的表达式用其值代替。
  - 常数传播：用在编译时已知的变量值代替程序正文中对这些变量的引用。
  - 削弱计算强度：对表达式中的求值计算用代数上等价的形式替换。
  - 改变计算次序：如果两个语句互不依赖，则可以改变计算次序。

基本块优化-删除公共子表达式

B_4

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

B_4'

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

基本块优化-复写传播

在复制语句a:= b 之后，尽可能用b代替a

(1) t_1:=4*i	(1) t_1:=4*i	(1) t_1:=4*i	(1) t_1:=4*i
(2) t_2:=a-4	(2) t_2:=a-4	(2) t_2:=a-4	(2) t_2:=a-4
(3') t_3:= t_1	(3') t_3:= t_1		
(4') t_4:= t_2	(4') t_4:= t_2		
(5) t_5:=t_4[t_3]	(5') t_5:=t_2[t_1]	(5') t_5:=t_2[t_1]	(5') t_5:=t_2[t_1]
(6') t_6:= t_1	(6') t_6:= t_1		
(7) t_7:=b-4	(7) t_7:=b-4	(7) t_7:=b-4	(7) t_7:=b-4
(8) t_8:=t_7[t_6]	(8') t_8:=t_7[t_1]	(8') t_8:=t_7[t_1]	(8') t_8:=t_7[t_1]
(9) t_9:=t_5+t_8	(9) t_9:=t_5+t_8	(9) t_9:=t_5+t_8	(9) t_9:=t_5+t_8
(10) t_2[t_1]:=t_9	(10) t_2[t_1]:=t_9	(10) t_2[t_1]:=t_9	(10) t_2[t_1]:=t_9
(11) t_{10}:=i+1	(11) t_{10}:=i+1	(11) t_{10}:=i+1	
(12) i:=t_{10}	(12) i:=t_{10}	(12) i:=t_{10}	(12') i:= i+1
(13) goto B_2	(13) goto B_2	(13) goto B_2	(13) goto B_2

基本块优化-删除无用代码

- **不可到达代码**：无论输入任何数据都不可能被执行的代码。
- **死变量**：仅仅被定值，但从未被引用的变量。
- **死代码**：可以被执行，但对于执行结果没有任何作用的代码。
- **死块**：控制流不可到达的块。
  - 如果一个基本块是在某一条件为真时才被执行，经数据流分析后得知该条件恒定为假，则该基本块为死块。
  - 如果一个基本块是在某一条件为假时才被执行，经数据流分析后得知该条件恒定为真，则该基本块也是死块。
  - 当某一个死块被删除后，如果有后续基本块也成为无控制转入的块，则该基本块也是死块。

代数恒等变换

* **合并已知量**：将在编译时可计算出结果的表达式用其计算结果替代。

如 a = 3*4+y; 可用 a = 12+y; 替换

* **代数化简**：利用运算符与操作数的组合特性，对基本块进行化简。

如 i = - (-i)

* **强度削弱**：用较快的运算来替换较慢的运算。

如x = y ** 2; 可用 x = y*y;替换

* **变量的重新组合**：利用交换律、结合律、分配律等规则对表达式进行改写，并充分利用在基本块中已定值的变量。

如a = b +c; 可替换为： a = b + c;

e = b + d + c; e = a + d;//将c 和d交换后再替换

改变计算次序

在一个基本块中，如果两个语句是互不依赖的，那么交换这两个语句的执行顺序并不会影响基本块的执行结果。好处在于在寄存器分配过程中可能会减少读取内存单元的次数。

如： $t_1:=b+c$

$t_2:=x+y$

如果x、y均不为t_1，b、c均不为t_2，则交换这两个语句的位置不影响基本块的执行结果。

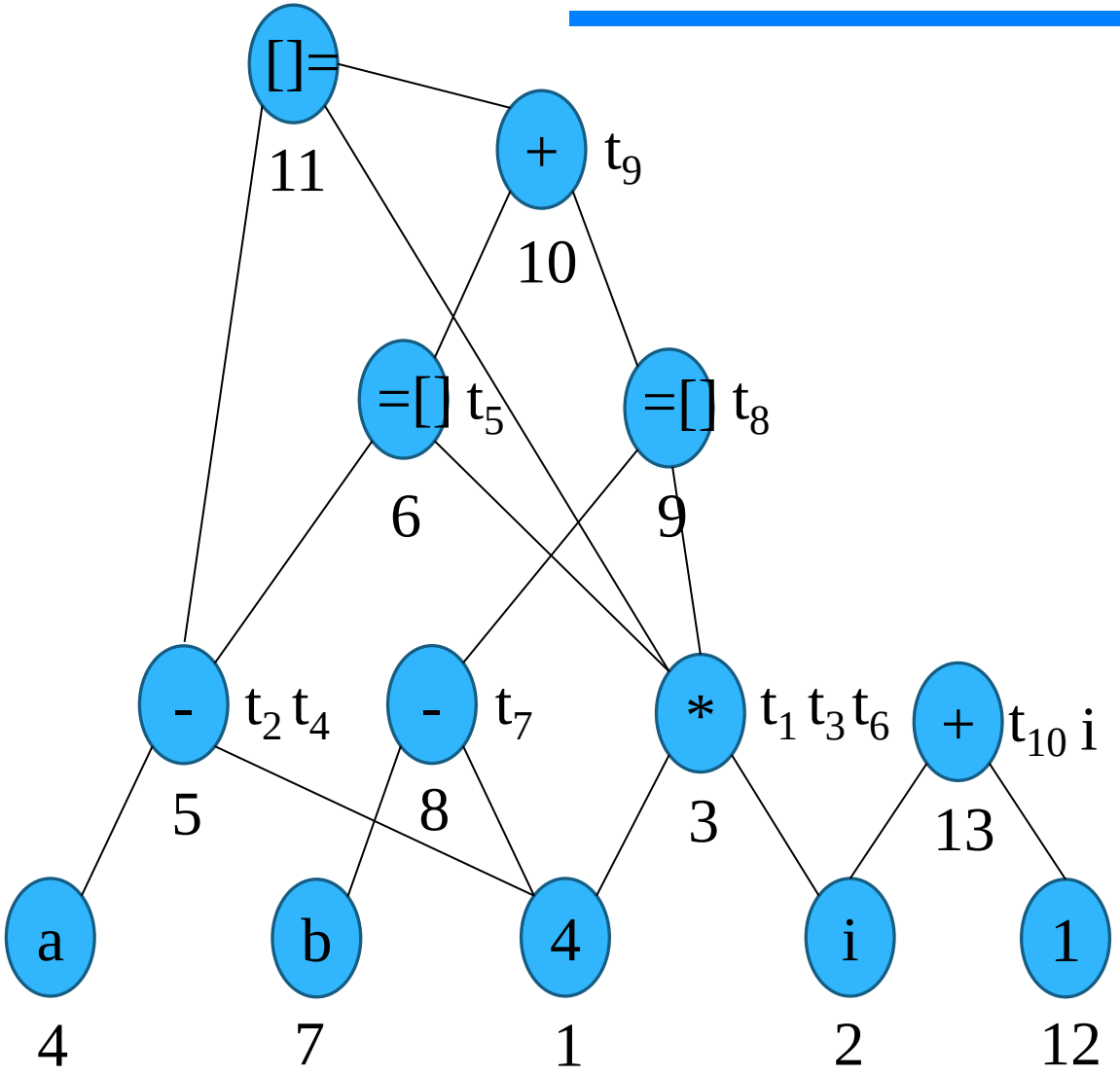
DAG在局部优化中的应用

* DAG是实现基本块等价变换的一种有效的数据结构。

一个基本块的DAG是一种在其结点上带有下述标记的有向非循环图：

- 图的叶结点由变量名或常量标记。
  - 根据作用到一个名字上的算符，可以决定需要的是名字的左值还是右值。
  - 大多数叶结点代表右值（叶结点代表名字的初始值），因此，通常将其标识符加上脚标0，以区别于指示名字的当前值的标识符。
- 图的内部结点由一个运算符号标记，每个内部结点均代表应用其运算符对其子结点所代表的值进行运算的结果。
- 图中每个结点都有一个标识符表，其中可有零个或多个标识符。这些标识符都具有该结点所代表的值。

DAG



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

基本块的DAG构造

输入：一个基本块。

输出：该基本块的DAG，其中包括如下的信息：

- 每个结点都有一个标记，叶结点的标记是一个名字或者常数，内部结点的标记是一个运算符号。
- 在每个结点上有一个附加的标识符表，表中可以有零或多个名字。

算法用到的主要数据结构：

- 保存DAG的数据结构（如数组、链表等），其中存储各结点的信息以及结点之间的关系。
- 保存结点附加信息的数据结构，需要记录结点的编号、标记、以及与结点相关的名字列表或常数。

基本块的DAG构造方法

* 开始时，DAG为空，对基本块中的每一条语句依次执行如下步骤：

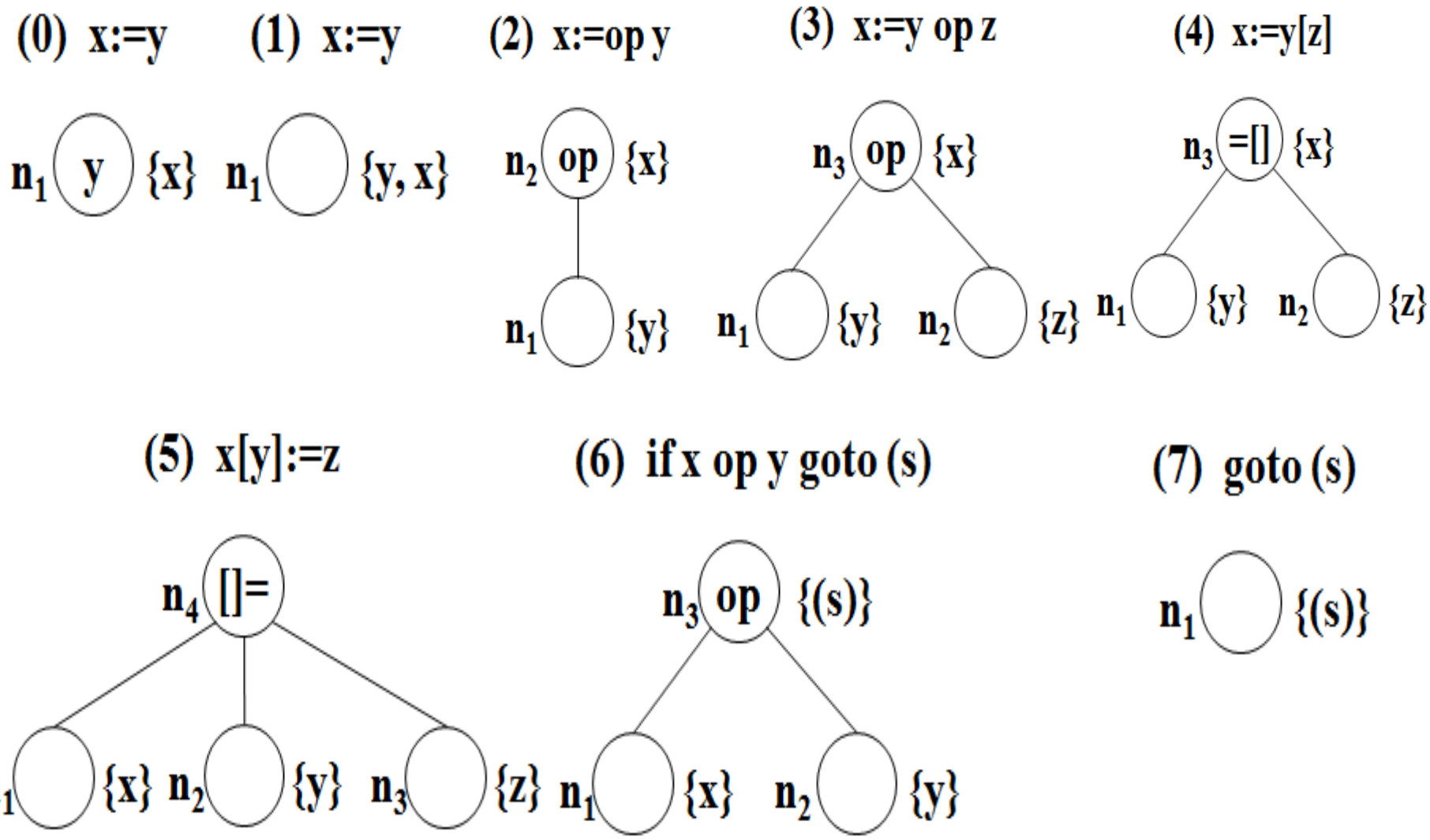
(1) 对于形如X:=Y的中间代码语句，查找是否存在一个节点Y，若不存在，则创建节点Y，并在附加标识符表中增加标识符X;

(2) 对于形如X:= op Y的中间代码语句，查找是否存在一个节点Y，若不存在，则创建节点Y，接着查找是否存在一个节点op，其子节点为Y，若不存在，则创建节点op，并将op和Y连接，同时，在附加标识符表中增加标识符X;

(3) 对于形如X:=Y op Z的中间代码语句，查找是否存在节点Y和Z，若不存在，则创建节点Y和(或)Z；接着查找是否存在一个节点op，其左子节点为Y，右子节点为Z，若不存在，则创建节点op，并将op和Y、Z连接，同时在附加标识符表中增加标识符X;

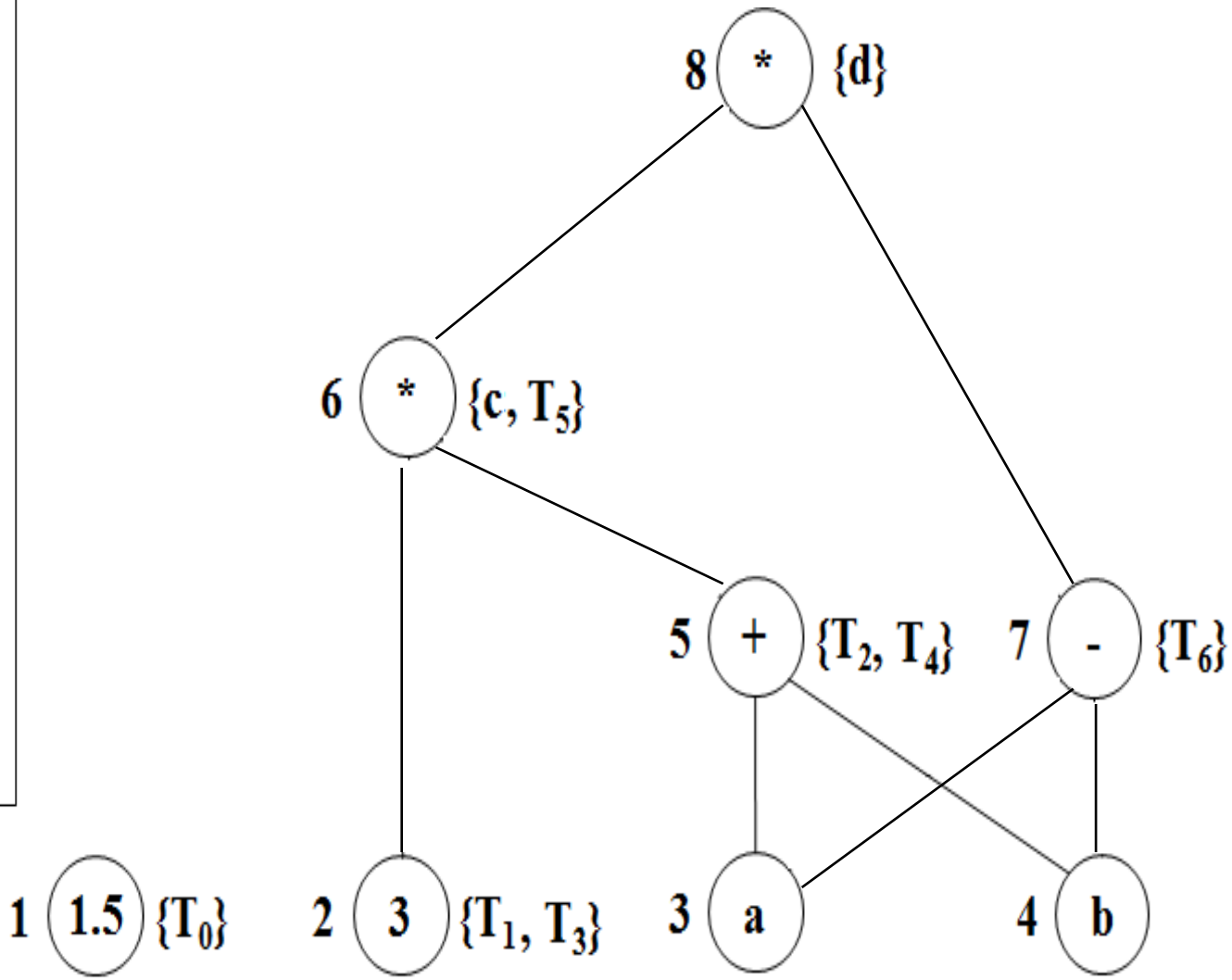
(4) 由于X的当前值是刚建立或已找到的节点的值，因此，对于事先附加在其他节点上的X需要删除掉。

常用三地址语句的DAG



算法示例

(1) T_0:=1.5
(2) T_1:=2*T_0
(3) T_2:=a+b
(4) c:=T_1*T_2
(5) d:=c
(6) T_3:=2*T_0
(7) T_4:=a+b
(8) T_5:=T_3*T_4
(9) T_6:=a-b
(10) d:=T_5*T_6



DAG的应用

* 在构造DAG的过程中，可以获得代码优化所需的一些信息：

(1) 首先，可以检测出公共子表达式。

(2) 其次，可以确定出哪些名字的值在前驱块中计算而在本块内被引用。
即，dag中叶子结点对应的名字。

(3) 再次，可以确定出哪些名字的值在本块中计算而可以在后继块中被引用。
即，在dag构造的结尾仍存在于结点的标识符表中的那些名字。

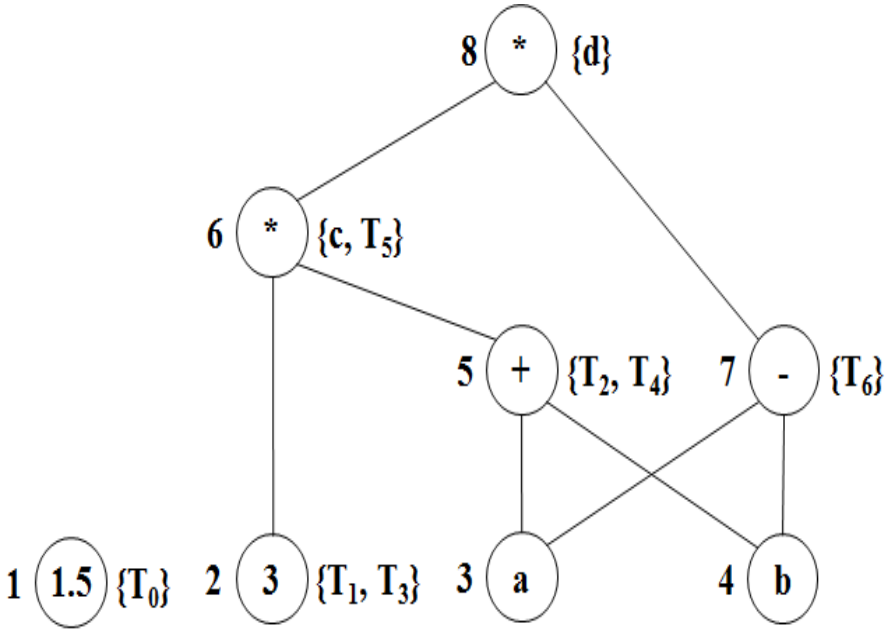
因此，可以将DAG应用在基本块的优化中

(1) 简化基本块

(2) 重拍基本块的计算顺序

简化基本块

(1) T_0:=1.5
(2) T_1:=3
(3) T_3:=3
(4) T_2:=a+b
(5) T_4:=T_2
(6) c:=3*T_2
(7) T_5:=c
(8) T_6:=a-b
(9) d:=c*T_6



(1) T_2:=a+b
(2) c:=3*T_2
(3) T_6:=a-b
(4) d:=c*T_6

重排基本块的计算顺序

```
s:=a+b
t:=c+d
u:=e-t
v:=s-u
```

```
(1)  MOV R0, a
(2)  ADD  R0, b
(3)  MOV R1, c
(4)  ADD  R1, d
(5)  MOV s,  R0
(6)  MOV R0, e
(7)  SUB  R0, R1
(8)  MOV R1, s
(9)  SUB  R1, R0
(10) MOV v,  R1
```

```
t:=c+d
u:=e-t
s:=a+b
v:=s-u
```

```
(1)  MOV R0, c
(2)  ADD  R0, d
(3)  MOV R1, e
(4)  SUB  R1, R0
(5)  MOV R0, a
(6)  ADD  R0, b
(7)  SUB  R0, R1
(8)  MOV  v,  R0
```

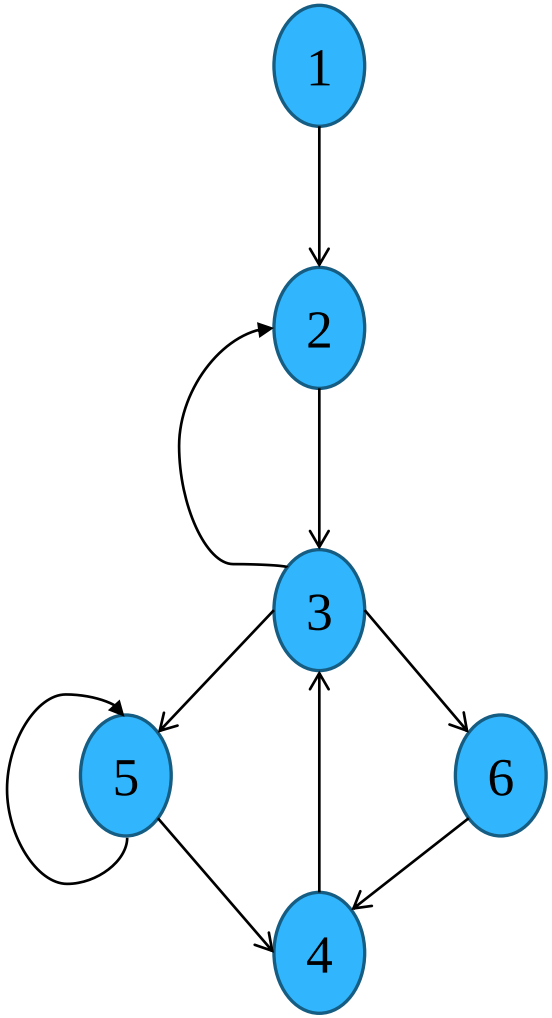
循环优化概述

* 循环就是程序中那些可能被反复执行的代码序列。因为循环中代码可能要反复执行，所以，进行代码优化时的重点是考虑循环的代码优化，这对提高目标代码的效率将起更大的作用。高级语言的循环语句(for语句、while语句、repeat语句等)

* 如何确定循环?

* 循环必备的两个基本性质：一是具有唯一的头节点；二是具有强连通性。

循环查找



具有强联通性

{2, 3}	非循环	入口：2, 3
{5}	循环	入口：5
{3, 4, 5, 6}	循环	入口：3
{2, 3, 4, 5, 6}	循环	入口：2
{3, 4, 5}	非循环	入口：3, 4

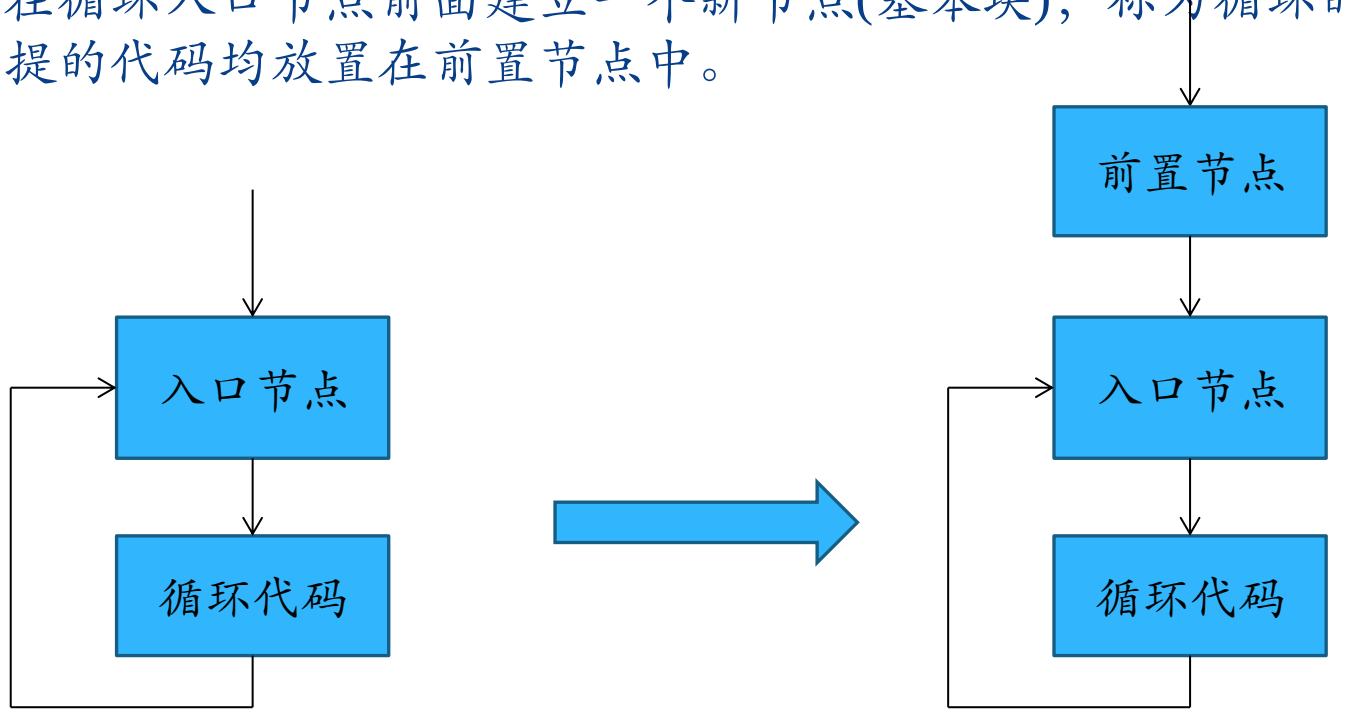
循环优化-代码外提

必经节点：控制流图中，如果起始节点S_0到达某一个节点S_n的所有有向边路径都经过某节点S_d，则称节点S_d是S_n的必经节点。

不变运算：指无论循环的代码被执行多少次，那些结果均不改变的运算。

代码外提：将不变运算的代码放到循环外执行。

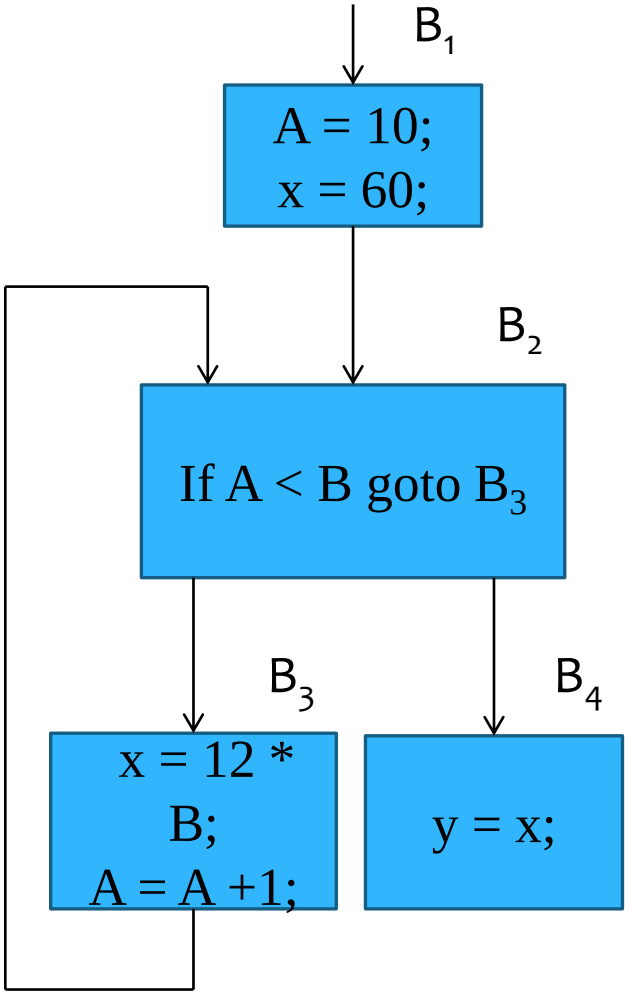
具体操作：在循环入口节点前面建立一个新节点(基本块)，称为循环的前置节点。从循环中外提的代码均放置在前置节点中。



循环优化-代码外提

* 例：

```
A = 10;
x = 60;
while A < B do
    begin
        x = 12 * B;
        A ++;
    end
y = x;
```



存在循环{B_2, B_3}

循环优化–代码外提

* 例：

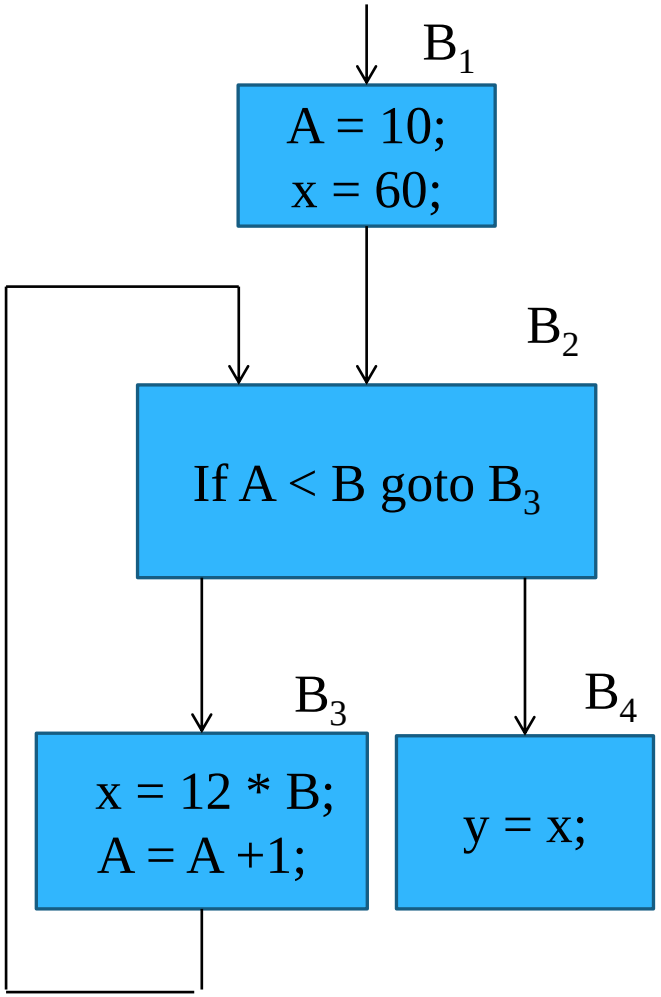
在循环中，B_3中的语句

x = 12 * B;

由于B在循环中没有定值，无论循环多少次，12 * B的值都不改变。

属于不变运算。

代码外提！

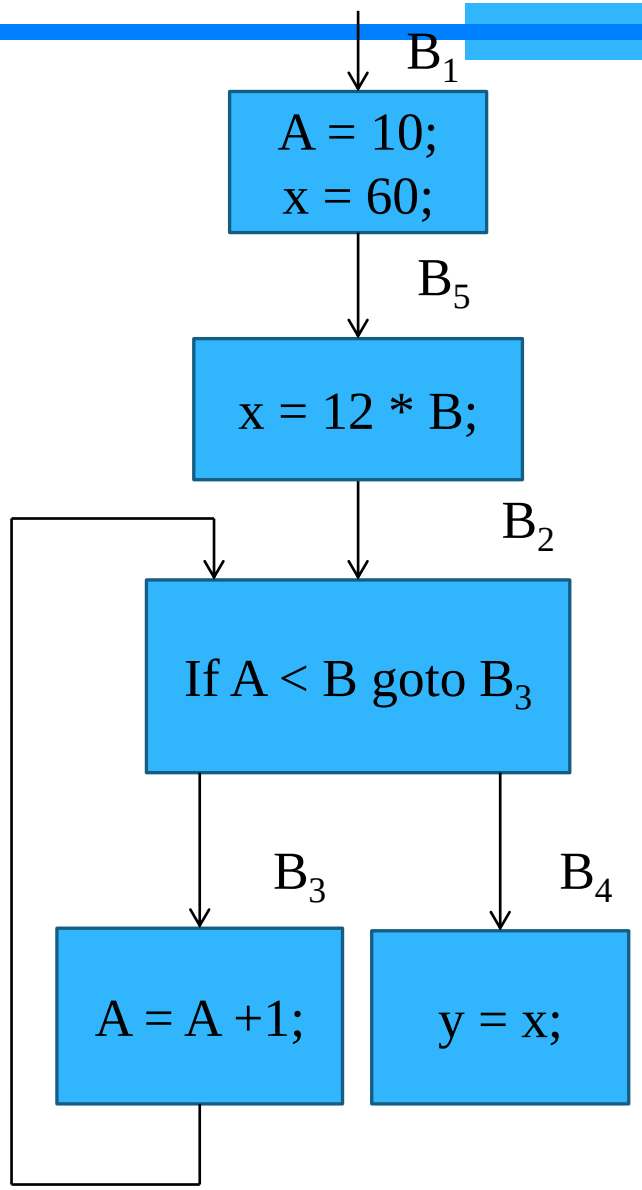


循环优化–代码外提

检查代码外提后的运行结果

假设在其他基本块中定义B=20; 代码外提后，对循环{B2, B3}中各语句运行结果都没有变化。

代码外提是有效的。



循环优化–代码外提

检查代码外提后的运行结果

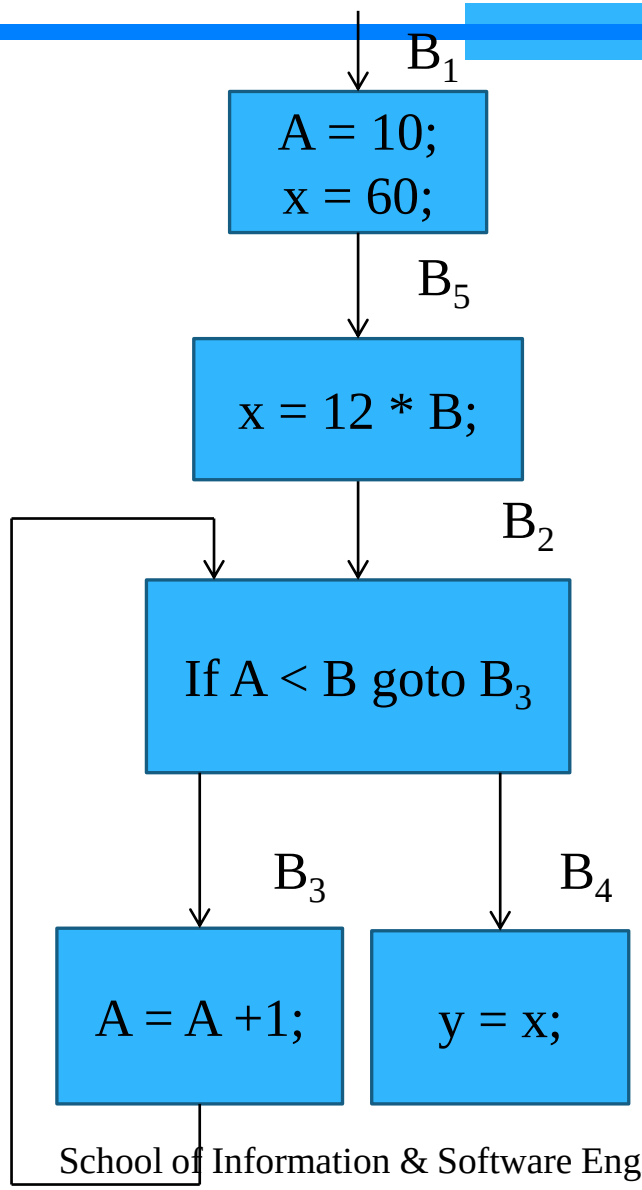
但存在另外一种情况：

假设在其他基本块中定义B = 8;

代码外提前，B3不会被执行。因此在B4中，y = x = 60;

代码外提后，x = 12 * B在进入循环前执行1次，因此在B4中y=x=96。

运算结果不一致，代码外提无效!



循环优化–代码外提

* 代码外提的条件

(1) 进行代码外提的不变运算，其所在的节点必须是循环所有出口的必经节点，

(2) 把形如X = Y op Z的不变运算外提时，要求循环中的其他地方不能对X再定值，而且对X的所有引用值均为该不变运算中确定的X的值。

循环优化–代码外提

不变运算查找算法

输入：循环L; L中的所有变量引用点的信息。

输出：查找、标识“不变运算”后的循环L;

方法：

(1) 依次查看L中各基本块的每条语句，如果其中的每个运算对象为常数或定值点在L外，则将该语句标记为“不变运算”；

(2) 依次查看每条未被标记为“不变运算”的语句，如果其运算对象为常数或定值点在L外，或只有一个到达一定值点且该点上的代码已标记为“不变运算”，则把该被查看的语句标记为“不变运算”。

(3) 重复执行步骤(2), 直到没有新的语句被标记为“不变运算”。

循环优化–代码外提

代码外提算法

输入：循环L。

输出：外提后的循环L′;

方法：

(1) 求出循环L中的所有不变运算;

(2) 对(1)求出的每一个不变运算，形如 S： A = B op C 或者 A = op B 或者 A= B，检查是否满足如下条件之一：

1）同时满足三点：a) S所在的节点是L的所有出口节点的必经节点；b)A在L中其它地方未再定值；c) L中所有A的引用点只有S中A的定值才能到达；

2) X在离开循环后是不活跃的，且条件1) 的b)和c)成立。

3) 依次把符合上述条件之一的不变运算外提到循环的前置节点中。若某不变运算S的运算对象是在L中定值的，那么，只有当这些定值语句都提到前置节点中后，才可把S外提。

循环优化-强度削弱

强度削弱：将程序中强度高的运算使用强度低的运算替代，以便使程序运行时间缩短。

一般情况下，如果循环L中存在 I = I ± C的语句，且在L中存在T = K * I ± D的语句，这种情况下，T呈线性变化。因此可以用± 运算代替 *运算。

$$T = K * I \pm D$$

$$T' = K * (I \pm C) \pm D$$

$$T' = K*I \pm K*C \pm D$$

$$T' = K * I \pm M$$

因此T'也是线性增长的，可以用T' = K' T' + D' 来替换。

循环优化-删除归纳变量

基本归纳变量与归纳变量

如果循环中变量I仅有唯一的I = I ± C的形式的赋值，其中C为循环不变量，则称I为循环中的基本归纳变量。

如果I是循环中的一个基本归纳变量，变量J在循环中的定值总可化为I的同一线性函数的形式：J= C_1*I±C_2，其中C_1，C_2是循环不变量，则称J是归纳变量，并称J与I同族。

如果循环中有两个或者两个以上同族的归纳变量，则可以只用其中的一个来代替基本归纳变量进行循环的控制，而去掉其余的归纳变量，这种方法称为删除归纳变量。

```
     i := n-1
S5:  if i<1 goto s1
     j := 1
s4:  if j>i goto s2
     t1 := j-1
     t2 := 4*t1
     t3 := A[t2]    ;A[j]

     t4 := j+1
     t5 := t4-1
     t6 := 4*t5
     t7 := A[t6]    ;A[j+1]

     if t3<=t7 goto s3
```

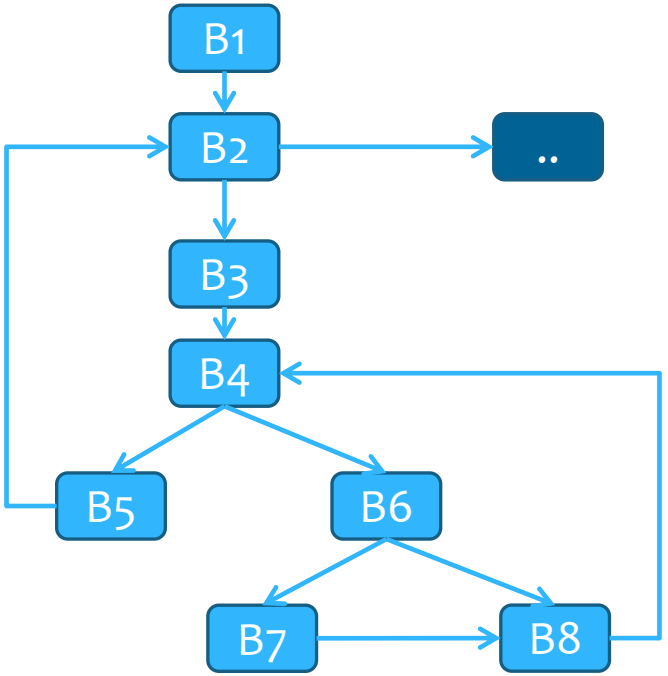
```
     t8 :=j-1
     t9 := 4*t8
     temp := A[t9]   ;A[j]
     t10 := j+1
     t11:= t10-1
     t12 := 4*t11
     t13 := A[t12]   ;A[j+1]
     t14 := j-1
     t15 := 4*t14
     A[t15] := t13 ;A[j]:=A[j+1]
     t16 := j+1
     t17 := t16-1
     t18 := 4*t17
     A[t18]:=temp   ;A[j+1]:=temp
s3:  j := j+1
     goto S4
S2:  i := i-1
     goto s5
s1:
```

```
FOR i := n-1 DOWNTO 1 DO
  FOR j := 1 TO i DO
    IF A[j]> A[j+1] THEN BEGIN
           temp := A[j];
           A[j] := A[j+1];
           A[j+1] := temp
    END
```

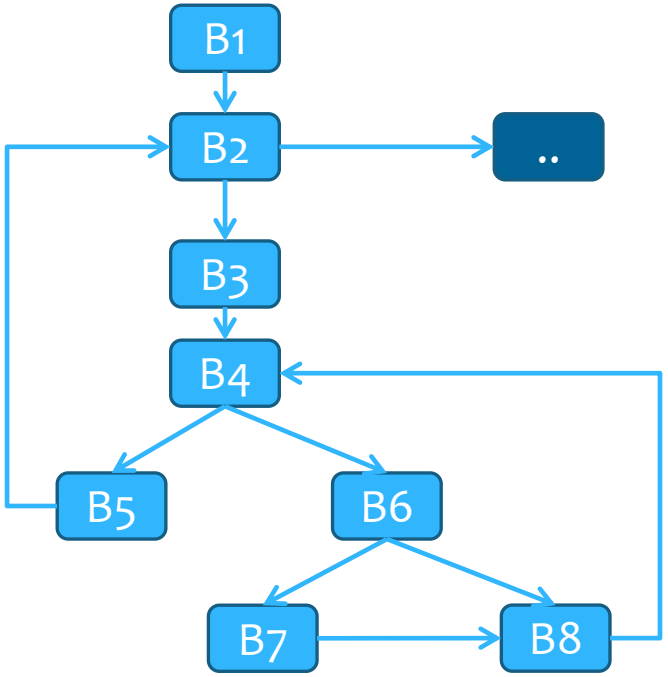
```
     i := n-1
S5:  if i<1 goto s1
     j := 1
s4:  if j>i goto s2
     t1 := j-1
     t2 := 4*t1
     t3 := A[t2]    ;A[j]

     t4 := j+1
     t5 := t4-1
     t6 := 4*t5
     t7 := A[t6]    ;A[j+1]

     if t3<t7 goto s3

t12:= 4*j

A[t9]:=t13

A[t12]:=temp

t6 := 4*j

     t8 :=j-1
     t9 := 4*t8
     temp := A[t9]   ;A[j]
     t10 := j+1
     t11:= t10-1
     t12 := 4*t11
     t13 := A[t12]   ;A[j+1]
     t14 := j-1
     t15 := 4*t14
     A[t15] := t13   ;A[j]:=A[j+1]
     t16 := j+1
     t17 := t16-1
     t18 := 4*t17
     A[t18]:=temp    ;A[j+1]:=temp
s3:  j := j+1
     goto S4
S2:  i := i-1
     goto s5
s1:
```

```
B1: i := n-1
B2: if i<1 goto out
B3: j := 1
B4: if j>i goto B5
B6: t1 := j-1
    t2 := 4*t1
    t3 := A[t2]      ;A[j]
    t6 := 4*j
    t7 := A[t6]      ;A[j+1]
    if t3<=t7 goto B8
```

```
B7: t8 := j-1
    t9 := 4*t8
    temp := A[t9]    ;temp:=A[j]
    t12 := 4*j
    t13 := A[t12]    ;A[j+1]
    A[t9]:= t13      ;A[j]:=A[j+1]
    A[t12]:=temp     ;A[j+1]:=temp
B8: j := j+1
    goto B4
B5: i := i-1
    goto B2
out:
```



```
B1: i := n-1
B2: if i<1 goto out
B3: j := 1
B4: if j>i goto B5
B6: t1 := j-1
    t2 := 4*t1
    t3 := A[t2]        ;A[j]
    t6 := 4*j
    t7 := A[t6]        ;A[j+1]
    if t3<=t7 goto B8
B7:
                       ;temp:=A[j]
                       ;A[j+1]
    A[t2]:= t7         ;A[j]:=A[j+1]
    A[t6]:= t3         ;A[j+1]:=temp
B8: j := j+1
    goto B4
B5: i := i-1
    goto B2
out:
```

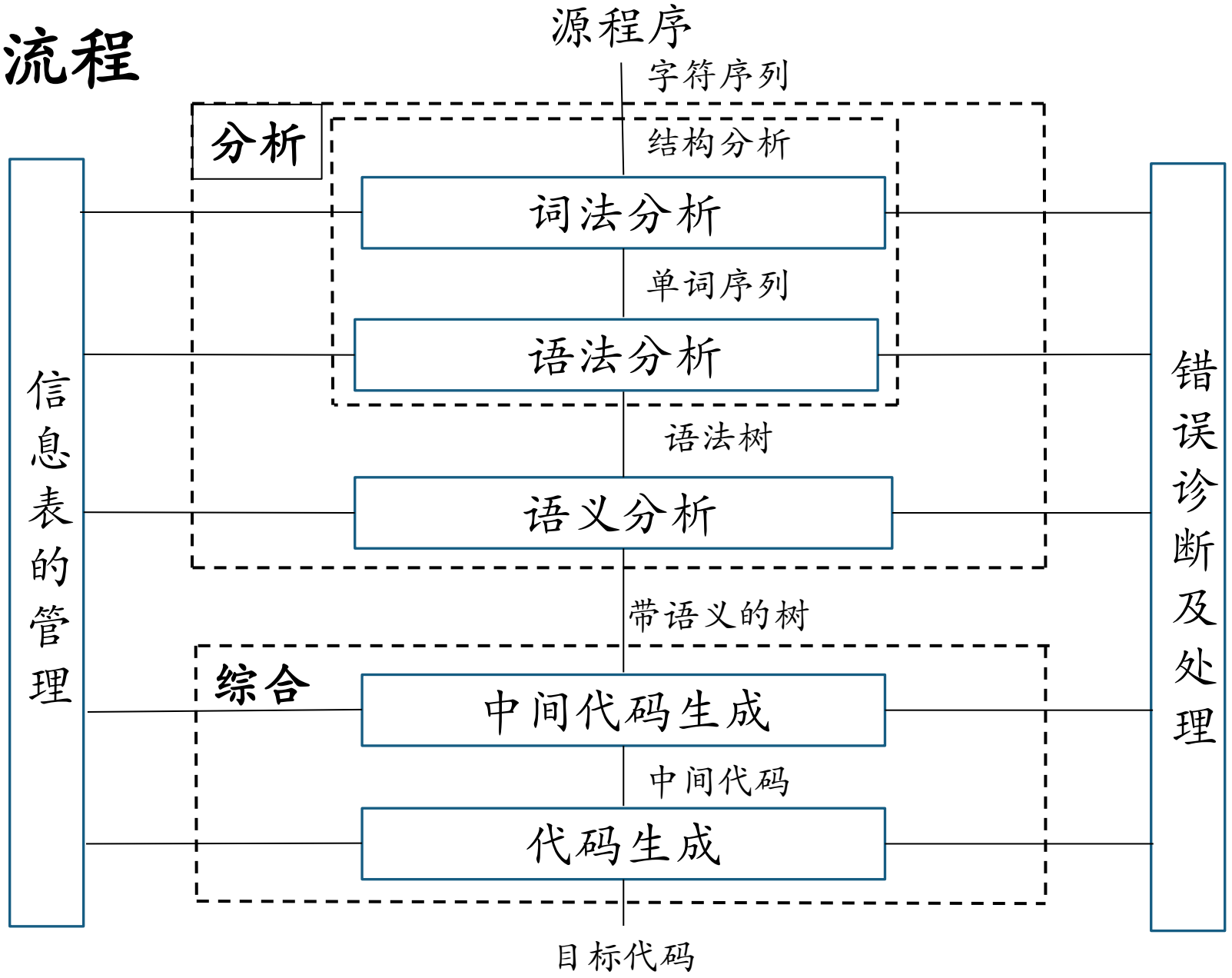


目标代码生成的任务及要求

* 目标代码生成的任务
  - 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
  - 1、正确
  - 2、高质量—占用空间少，运行效率高

编译流程


源程序
字符序列
分析
结构分析
词法分析
单词序列
语法分析
语法树
语义分析
带语义的树
综合
中间代码生成
中间代码
代码生成
目标代码
信息表的管理
错误诊断及处理

目标代码生成的输入

* **中间代码**：经过语法分析/语义检查之后得到的中间表示
  - 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

目标代码生成的输出

* 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

目标代码生成的相关问题

* 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

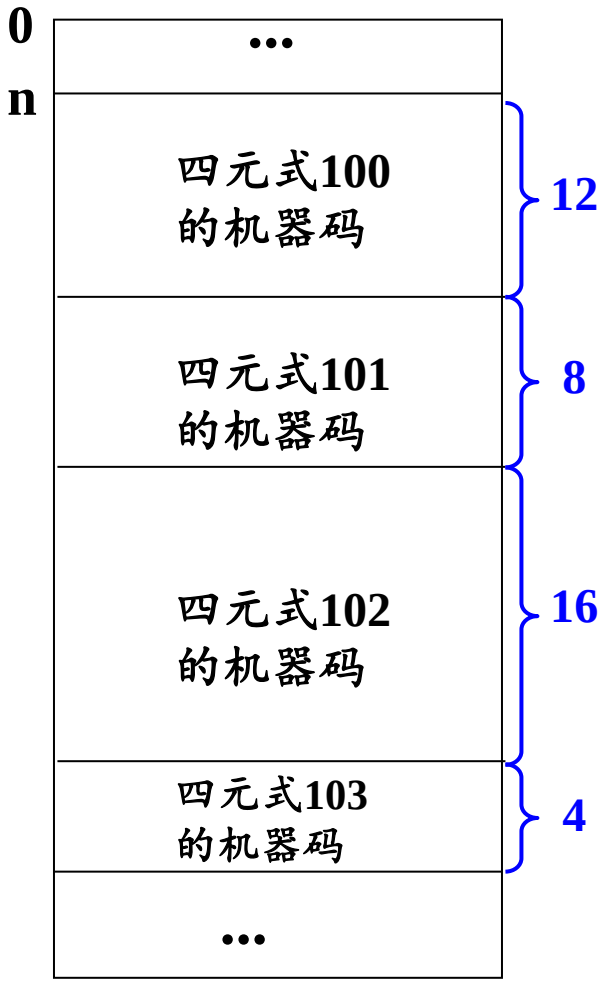
* 三地址代码中的名字
  - 指向该名字在符号表中位置的指针

存储管理

中间代码

	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



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  R0, y
ADD  R0, z
MOV  x, R0
```

a:=b+c
d:=a+e

```
MOV  R0, b
ADD  R0, c
MOV  a, R0
MOV  R0, a
ADD  R0, e
MOV  d, R0
```

a:=a+1

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

```
INC  a
```

...

寄存器分配

- 选出要使用寄存器的变量
  - 局部范围内
  - 在程序的某一点上
- 寄存器指派
  - 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

计算次序的选择

* 计算次序影响目标代码的效率
  - RISC体系结构的一种通用的流水线限制是：从内存中取出存入寄存器的值在随后的几个周期内是不能用的。
  - 在这几个周期期间，可以调出不依赖于该寄存器值的指令来执行，如果找不到这样的指令，则这些周期就会被浪费。
  - 所以，对于具有流水线限制的体系结构，选择合适的计算次序是必需的。
  - 有些计算顺序可以用较少的寄存器来保留中间结果

* 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

目标机器模型–寄存器

* 以8086微处理器作为目标机，以汇编语言作为目标代码形式为例
* 8个通用寄存器，均为16位

AX: Accumulator, 累加寄存器，也称为累加器;
BX: Base, 基地址寄存器;
CX: Count, 计数器寄存器;
DX: Data, 数据寄存器;

数据寄存器
可拆分成两个独立的8位寄存器使用

BP: Base Pointer, 基指针寄存器;
SP: Stack Pointer, 堆栈指针寄存器

指针寄存器
不可拆分
用于寻址操作

SI: Source Index, 源变址寄存器;
DI: Destination Index, 目的变址寄存器

变址寄存器
不可拆分
用于寻址操作

目标机器模型–寄存器

* 以8086微处理器作为目标机，以汇编语言作为目标代码形式为例
* 其他寄存器

  CS: Code 代码段寄存器;

  IP: Index Pointer, 指令指针寄存器;

  指示CPU当前要读取的指令的地址

  SS: Stack Segment, 堆栈段寄存器;

  DS: Data Segment, 数据段寄存器;

  ES: Extra Segment, 附加段寄存器;

  FLAG: 标志寄存器。

  按位起作用

* https://blog.csdn.net/weixin_40913261/article/details/90762210

目标机器模型–指令

以8086微处理器作为目标机，以汇编语言作为目标代码形式为例

指令	含义	备注
MOV R_d, R_s/M	表示将R_s/M中的内容送到R_d中	
MOV R_d/M, R_s	表示将R_s中的内容送到R_d/M中	
MOV R_d/M, imm	表示将imm送到R_d/M中	
ADD R_d, R_s/M	表示对R_s/M，R_d中的内容求和，并将结果送到R_d中	其他逻辑运算类似
ADD R_d/M, R_s/imm	表示对R_d/M, R_s/imm中的内容求和，并将结果送到R_d/M中	
CMP A, B	根据A和B的比较结果设置条件标志位	
JMP	无条件转移	
JZ, JE	全0，或相等则转移	Z=1(全0)，(A)=(B)
JNZ, JNE	不为0，或不相等则转移	Z=0(非全0), (A)≠(B)
CALL	调用子程序	
RET	子程序的返回指令	

目标机器模型–指令

以8086微处理器作为目标机，以汇编语言作为目标代码形式为例

指令寻址方式

类 型	指令形式	意义(设 op 是二目运算符)
直接地址型	op R_i, M	(R_i) op (M) ⇒ R_i
寄存器型	op R_i, R_j	(R_i) op (R_j) ⇒ R_i
变址型	op R_i, c(R_j)	(R_i) op $((R_j)+c)$ ⇒ R_i
间接型	op R_i, *M	(R_i) op ((M)) ⇒ R_i
	op R_i, * R_j	(R_i) op $((R_j))$ ⇒ R_i
	op R_i, *c(R_j)	(R_i) op $((R_j)+c)$ ⇒ R_i

如果op是一目运算符，则“op R_i，M”的意义为：op(M) R_i，其余类型可类推。

简单的代码生成器（基本块内）

例： T4 = A + B – (E – (C + D))代码生成

共生成10条代码

T1:= A+B	MOV A,R0
	ADD B,R0
T2:=C+D	MOV C,R1
	ADD D,R1
T3:=E-T2	MOV R0,T1
	MOV E, R0
	SUB R1,R0
T4:=T1-T3	MOV T1,R1
	SUB R0,R1
	MOV R1, T4

简单的代码生成器（基本块内）

例： T4 = A + B – (E – (C + D))代码生成

共生成8条代码

	MOV A,R0
	ADD B,R0
T1:= A+B	MOV C,R1
	ADD D,R1
T2:=C+D	MOV R0,T1
	MOV E, R0
T3:=E-T2	SUB R1,R0
	MOV T1,R1
	SUB R0,R1
T4:=T1-T3	MOV R1, T4

```
MOV  C,R0
ADD  D,R0
MOV  E,R1
SUB  R0,R1
MOV  A,R0
ADD  B, R0
SUB  R1,R0
MOV  R0,T4
```

在一个基本块范围内考虑如何充分利用寄存器

- 尽可能地让变量的值保留在寄存器中
- 尽可能引用变量在寄存器中的值

待用信息与活跃信息

* 若在一个基本块中，变量A在四元式i中被定值，在i后面的四元式j中如果要引用A的值，且从i到j之间没有其他对A的定值点，则称j是四元式i中对变量A的待用信息，并且A在i处是活跃的。如果A被多次引用则可构成待用信息链与活跃信息链。

* 待用信息有助于把基本块内还要被引用的变量值尽可能地保存在寄存器中，同时，把基本块内不再被引用的变量所占用的寄存器尽早释放。

* 可从基本块的出口由后向前扫描，对每个变量建立相应的待用信息链和活跃变量信息链。

计算待用信息

* 用符号对(×，×)表示变量的待用信息和活跃信息。

其中:

- i表示待用信息(即下一个引用点)，y表示活跃，∧表示非待用或非活跃;
- 在符号表中，(×，×)→(×，×)表示在算法执行过程中后面的符号对将替代前面的符号对。

计算待用信息的算法

在符号表中增加“待用信息”栏和“活跃信息”栏

(1)初始化。把各基本块的符号表中的“待用信息”和“活跃信息”栏置初值。即把“待用信息”栏置为“非待用”，把“活跃信息”栏按照在基本块出口处是否为活跃置为“活跃”或“非活跃”。

计算待用信息的算法

(2)按照以下步骤从基本块出口到入口由后向前依次处理每个四元式(形式为A = B op C 或 A = op B 或A = B):

- 把符号表中变量A的代用信息和活跃信息附加到四元式i上。
- 把符号表中变量A的待用信息栏和活跃信息栏分别置为“非待用”和“非活跃”。
- 把符号表中变量B和变量C(如果存在)的待用信息和活跃信息附加到四元式i上。
- 把符号表中变量B和C的待用信息栏置为“i”，活跃信息栏置为“活跃”。

例：基本块的中间代码序列为：

假设W是基本块出口之后的活跃变量。

变量名	待用信息		活跃信息	
	初值	待用信息链	初值	活跃信息链
A				
B				
C				
T				
U				
V				
W				

(1)T = A – B　　(2)U = A – C

(3)V = T + U　　(4)W = V + U

初始化

变量名	待用信息		活跃信息	
	初值	待用信息链	初值	活跃信息链
A	∧		∧	
B	∧		∧	
C	∧		∧	
T	∧		∧	
U	∧		∧	
V	∧		∧	
W	∧		Y	

(1)T = A – B　　(2)U = A – C

(3)V = T + U　　(4)W = V + U

处理代码（4）

变量名	待用信息		活跃信息	
	初值	待用信息链	初值	活跃信息链
A	∧		∧	
B	∧		∧	
C	∧		∧	
T	∧		∧	
U	∧	4	∧	Y
V	∧	4	∧	Y
W	∧	∧	Y	∧

(1)T = A – B　　(2)U = A – C

(3)V = T + U　　(4)W (∧,Y) = V(∧, ∧) + U (∧, ∧)

处理代码（3）

变量名	待用信息					活跃信息				
	初值	待用信息链				初值	活跃信息链			
A	∧					∧				
B	∧					∧				
C	∧					∧				
T	∧		3			∧		Y		
U	∧	4	3			∧	Y	Y		
V	∧	4	∧			∧	Y	∧		
W	∧	∧				Y	∧			

(1)T = A – B　　(2)U = A – C

(3)V(4,Y) = T (∧, ∧) + U (4,Y)　　(4)W (∧,Y) = V(∧, ∧) + U (∧, ∧)

处理代码（2）

变量名	待用信息				活跃信息			
	初值	待用信息链			初值	活跃信息链		
A	∧			2	∧			Y
B	∧				∧			
C	∧			2	∧			Y
T	∧		3		∧		Y	
U	∧	4	3	∧	∧	Y	Y	∧
V	∧	4	∧		∧	Y	∧	
W	∧	∧			Y	∧		

(1)T = A – B

(2)U (3,Y) = A (∧, ∧) – C (∧, ∧)

(3)V (4,Y) = T (∧, ∧) + U (4,Y)

(4)W (∧,Y) = V (∧, ∧) + U (∧, ∧)

处理代码（1）

变量名	待用信息					活跃信息				
	初值	待用信息链				初值	活跃信息链			
A	∧			2	1	∧			Y	Y
B	∧				1	∧				Y
C	∧			2		∧			Y	
T	∧		3		∧	∧		Y		∧
U	∧	4	3	∧		∧	Y	Y	∧	
V	∧	4	∧			∧	Y	∧		
W	∧	∧				Y	∧			

(1)T (3,Y) = A (2,Y) – B (∧, ∧)

(2)U (3,Y) = A (∧, ∧) – C (∧, ∧)

(3)V (4,Y) = T (∧, ∧) + U (4,Y)

(4)W (∧,Y) = V (∧, ∧) + U (∧, ∧)

寄存器分配

为了生成更有效的目标代码，需要考虑的一个问题就是如何更有效地利用寄存器。每生成一条目标代码时，如果其运算对象的值在寄存器中，那么，总是将该寄存器作为操作数地址，使得生成的目标代码执行速度较快。为此，应该尽可能把各变量的现行值保存在寄存器中，且把基本块不再引用的变量所占用的寄存器及早释放出来。

寄存器分配是将程序中的数量几乎无限的虚拟寄存器映射到数量有限的物理寄存器。

原则：用得最多的变量应该放在寄存器中

循环中最活跃，循环外最活跃

寄存器分配

寄存器分配算法有：图着色算法、线性扫描算法、整数线性规划算法、PBQP算法、Multi-Flow Commodities算法、基于SSA的寄存器分配。

LLVM没有全部支持上述寄存器分配算法，LLVM中支持的寄存器分配算法有4种：Basic Register Allocator、Fast Register Allocator、PBQP Register Allocator、Greedy Register Allocator

寄存器分配-图着色算法

* 基本概念

生存期：变量的定义和使用的期限。如果两个变量的生存期交叠，则发生了冲突，不能使用同一个寄存器。

图着色：将生存期看成顶点，而将冲突定义为边。给每个顶点一个颜色，相邻的顶点不能使用同样的颜色。

K-着色：能用K种颜色着色，如果一个冲突图中有k-着色则表示最多需要K个寄存器。

寄存器分配-图着色算法

* 处理流程

首先选择目标机器指令，处理时假设有无穷多个符号化的寄存器。

然后将物理寄存器指派给符号化寄存器，寻找到一个溢出代价最小的指派方法。

构造寄存器冲突图，图中结点是符号化寄存器，对于任意两个结点，如果一个节点在另一个节点被定值的地方是活跃的，那么这两个结点之间就有一条边。

尝试用K种颜色对寄存器冲突图进行着色。K为可指派的寄存器的个数。

寄存器分配-图着色算法

* NP-完全问题

无法在确定的时间内为一个冲突图进行着色。

k-coloring a graph Kempe [1879]

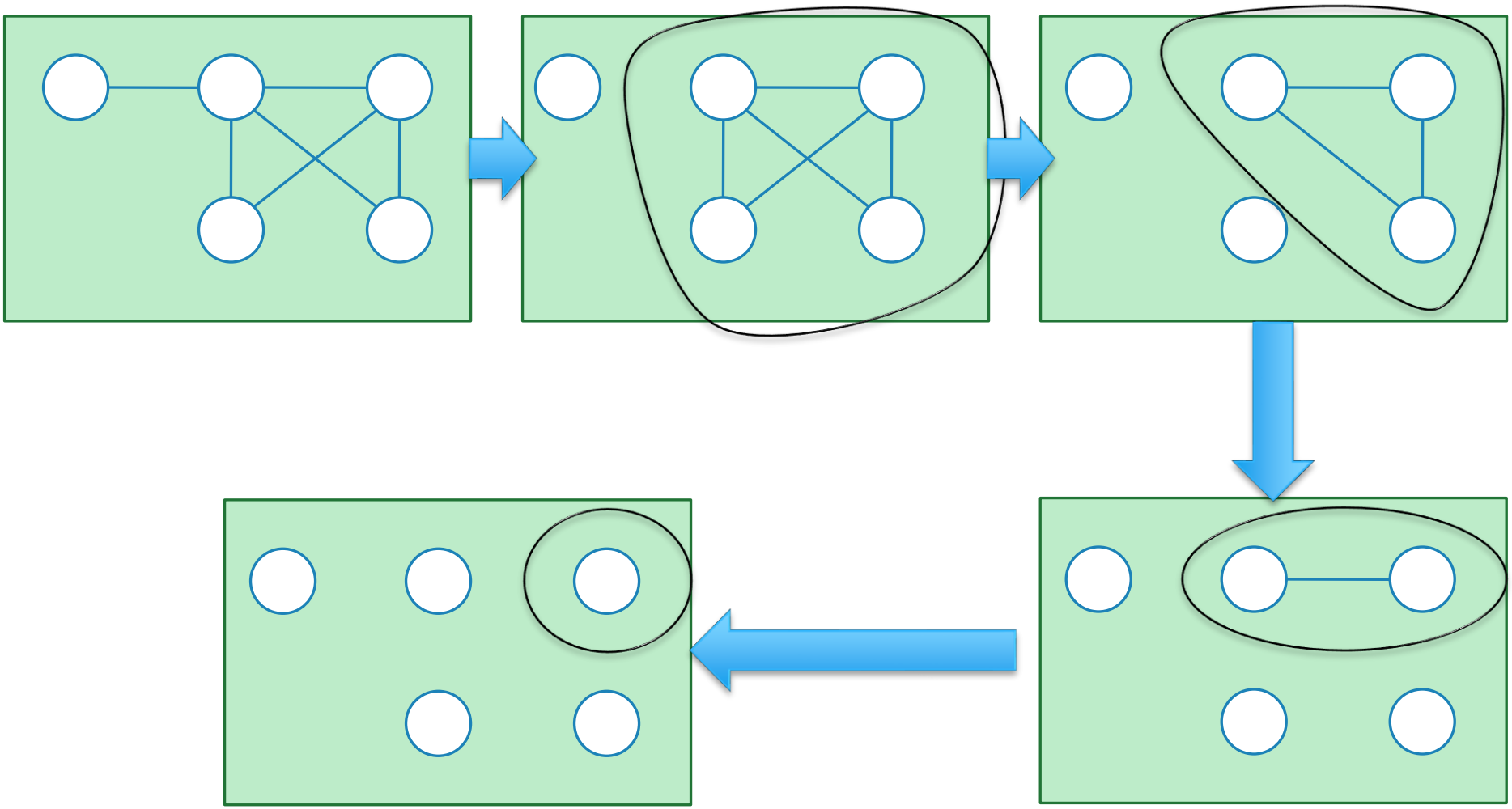
递归地使用以下三个步骤:

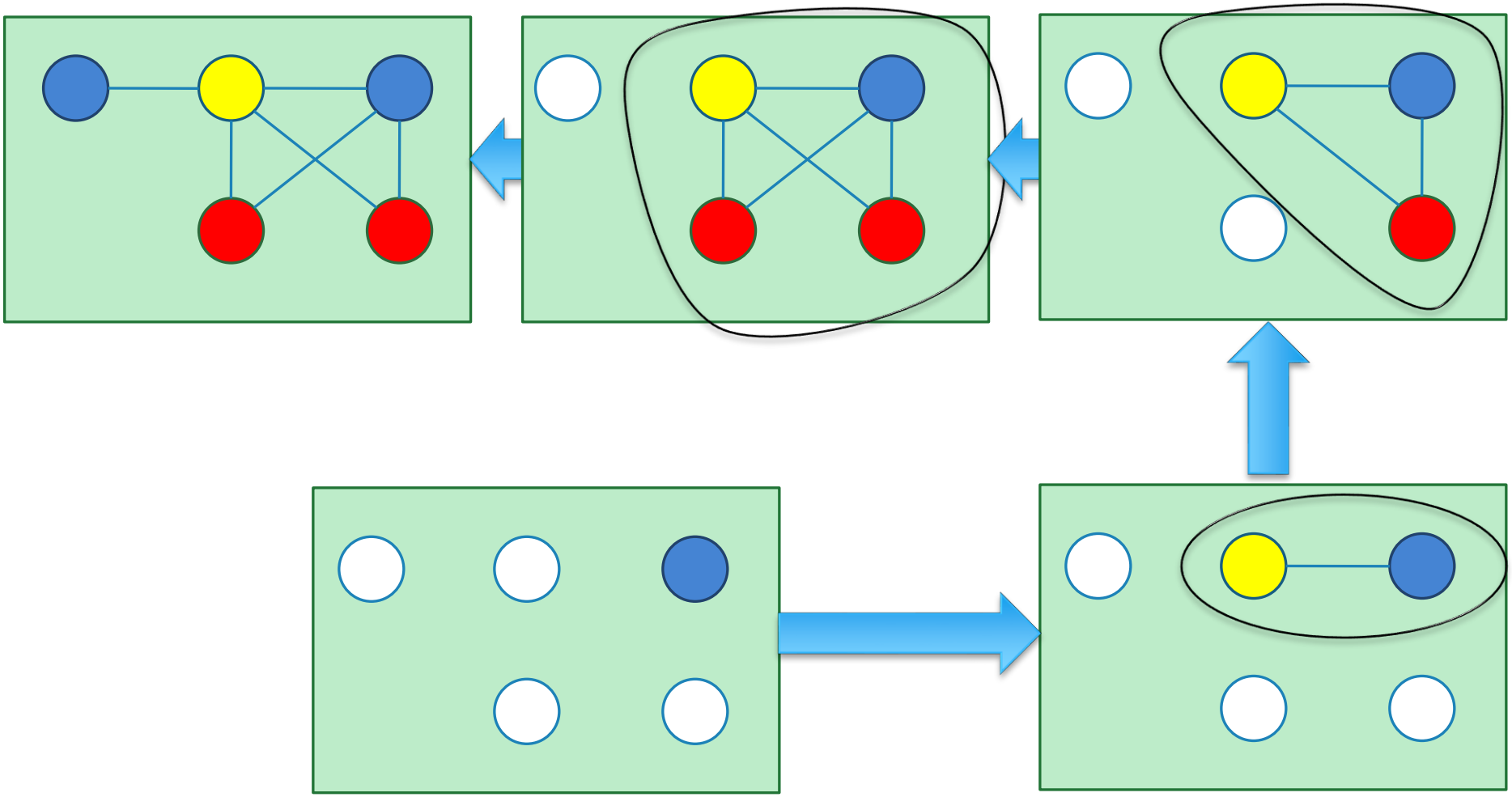
Step 1: 找到一个邻居结点小于K的节点，将它及其所有的边临时性地删除。

Step 2: 对剩下的子图重复步骤1，直到剩下一个空图或者图中所有结点都至少有K个邻居。

Step 3: 对于步骤2的第一种情况：根据前面两个步骤中被删除的相反顺序对结点进行着色。从而得到一个原图的K着色方案。对于步骤2的第二种情况，可以通过引入保存和重新加载寄存器的代码，将某个结点溢出。

3个可用寄存器





寄存器分配-线性扫描

* **变量生存周期**

  变量从被赋值到最后一次被引用的时间

  如果两个变量的生存周期并不交叠，则这两个变量可以共享同一个寄存器。

寄存器分配-线性扫描

* 线性扫描算法

Step 1: 生存期分析。计算每一个变量的生存期。

a{1-10}

b{5-20}

c{15-40}

d{9-50}

e{25-30}

寄存器分配-线性扫描

* 线性扫描算法

Step 1: 生存期分析。计算每一个变量的生存期。

Step 2: 根据开始时间将所有生存期排序并写入表Live_intervals中。

Live_intervals

a{1-10}

b{5-20}

d{9-50}

c{15-40}

e{25-30}

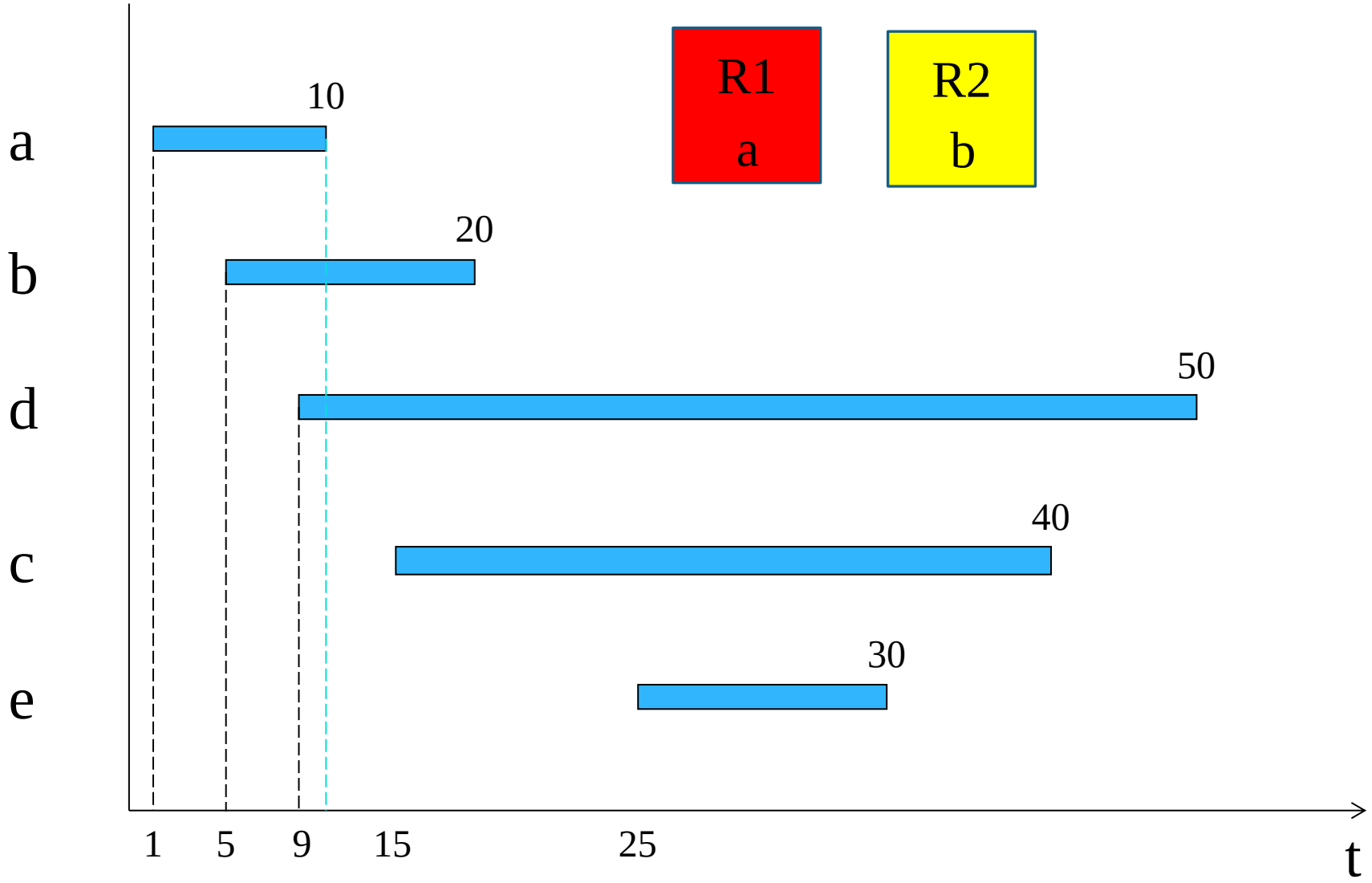
寄存器分配-线性扫描

* 线性扫描算法

Step 1: 生存期分析。计算每一个变量的生存期。

Step 2: 根据开始时间将所有生存期排序并写入表Live_intervals中。

Step 3: 根据结束时间将所有有效生存期排序并写入表Active中。

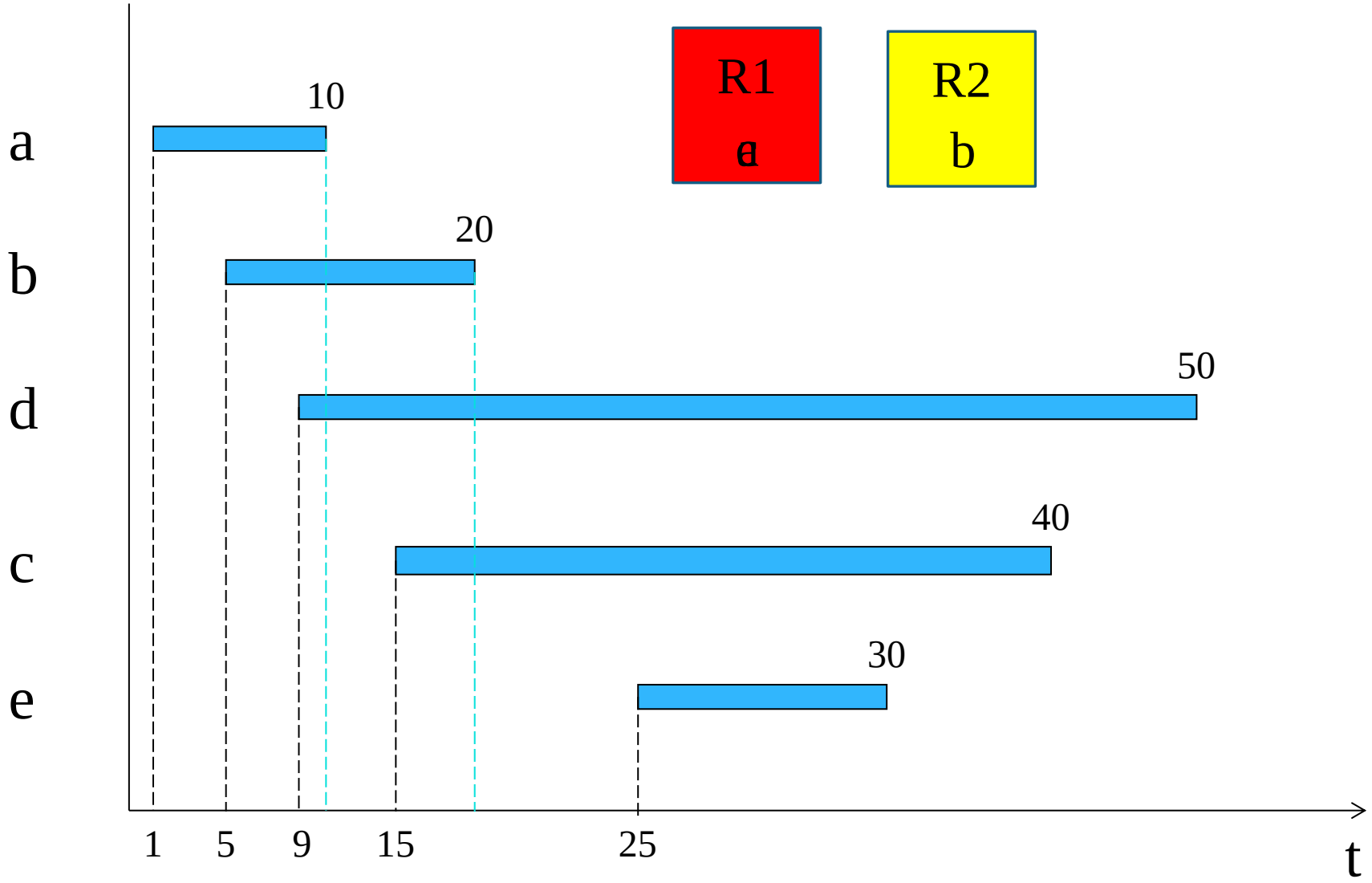


0<t<1: active = {}

1<t<5: active = {a}

5<t<9: active = {a, b}

9<t<10: active = {a, b, d}

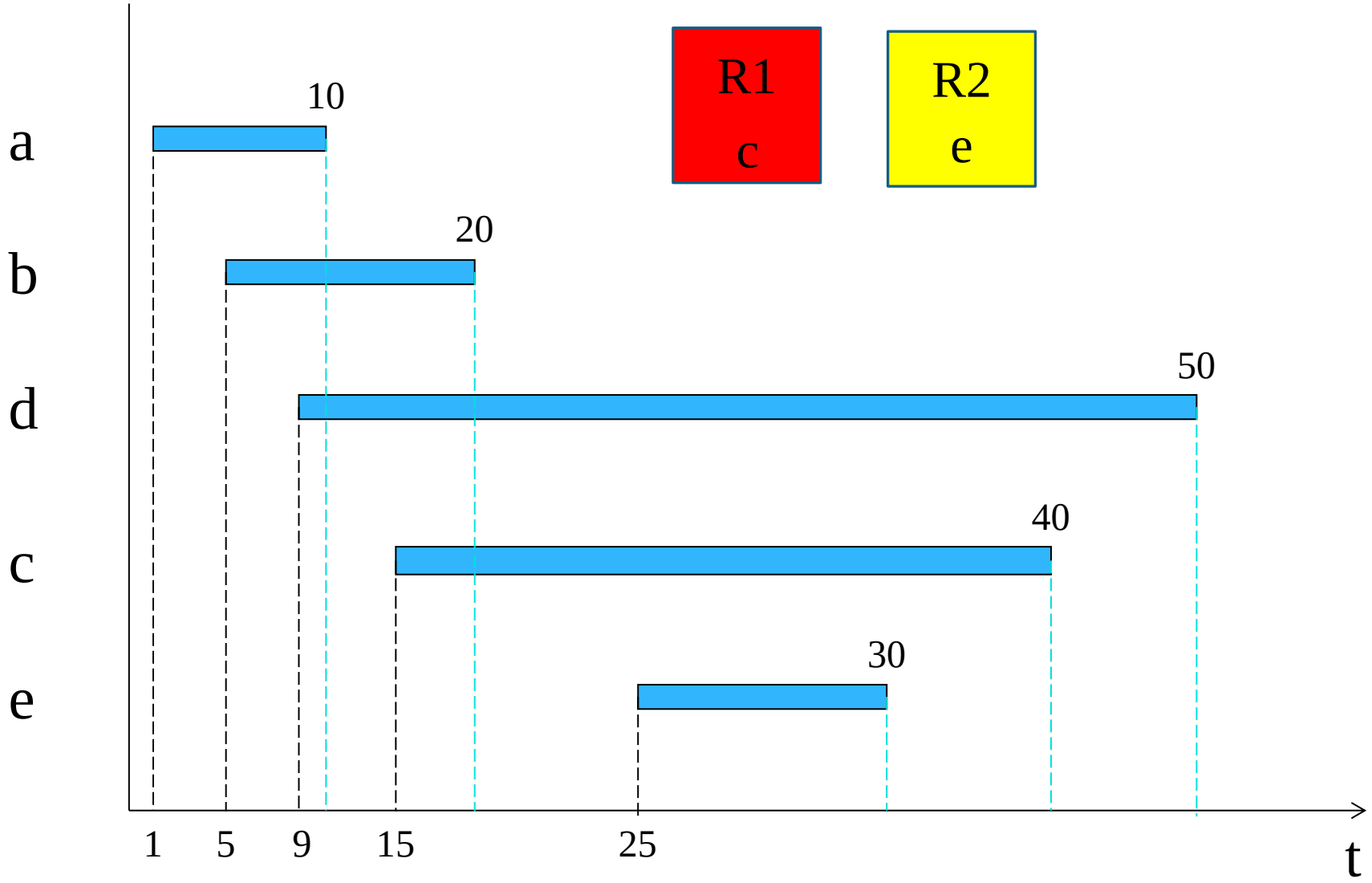


9<t<10: active = {a, b}; Spill = {d}

10<t<15: active = {b}; Spill = {d}

15<t<20: active = {b, c}; Spill = {d}

20<t<25: active = {c}; Spill = {d}



25<t<30: active = {c, e}; Spill = {d}

30<t<40: active = {e}; Spill = {d}

40<t<50: active = {}; Spill = {d}

50<t: active = {}; Spill = {}

寄存器描述和地址描述

为在代码生成中更有效地进行寄存器分配，需要随时掌握各寄存器的占用情况。为此，在代码生成过程中，需要使用寄存器描述数组RVALUE和变量地址描述数组AVALUE。

RVALUE动态地记录着每一个寄存器的分配信息。

RVALUE[Ri] = {A, B}表示寄存器Ri被A和B占用。

AVALUE动态地记录各变量现行值的存放位置。

AVALUE[X] = {Ri, Rj}表示变量X存放在寄存器Ri和Rj中。

寄存器分配算法GETREG

对于中间代码i A = B op C来说

1) 如果B的当前值在寄存器Ri中，且RVALUE[Ri] = {B}, 同时，满足如下两种情况之一:

   (a) 如果B和A是同一标识符，或者

   (b) 在中间代码i的附加信息中，B的待用信息和活跃信息分别为“非待用”和“非活跃”，意味着B的现行值在执行中间代码A = B+C后不会再被使用，则选取Ri为所需的寄存器，并返回。

2) 如果有空闲的寄存器，则从中选取一个Ri为所需的寄存器，并返回。

3) 从已分配的寄存器中选取一个Ri为所需的寄存器。假设变量X占用Ri, 则需满足以下条件之一，就可对寄存器所含的变量以及变量在内存中的情况做出调整:

   (a) 变量X的现行值，存放在内存单元中，可以从内存中读取X;

   (b) 在基本块中，X不会被引用或者要在最远的将来才会被引用;

代码生成算法–例子

假设只有R0和R1是可用寄存器。

中间代码	目标代码	RVALUE	AVALUE
T = A - B	LD R0, A SUB R0, B	R0中包含T	T在R0中
U = A - C	LD R1, A SUB R1, C	R1中包含U	T在R0中 U在R1中
V = T + U	ADD R0, R1	R0中包含V R1中包含U	V在R0中 U在R1中
W = V + U	ADD R0, R1	R0中包含W	W在R0中

代码生成算法

对每个中间代码i : A = B op C，依次执行下述步骤。

(1)以中间代码i : A = B op C为参数，调用函数过程GETREG(i : A = B op C)。当从GETREG返回时，我们得到一个寄存器R，它将用作存放A现行值的寄存器。

(2)利用地址描述数组AVALUE[B]和AVALUE[C]，确定出变量B和C现行值的存放位置B′和C′。如果其现行值在寄存器中，则把寄存器取作B′和C′。

(3)如果B′≠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}，以表示变量A的现行值只在R中并且R中的值只代表A的现行值。

(5)如果B和C的现行值在基本块中不再被引用，它们也不是基本块出口之后的活跃变量(由该中间代码i上的附加信息知道)，并且其现行值在某寄存器中，则删除AVALUE[Rk]中的B或C以及AVALUE[B]中的Rk，使该寄存器不再为B或C所占用。

活跃变量

为那些经常需要使用的变量指派单独的寄存器，将剩余寄存器用于存放基本块中局部变量的值。

如何评价变量在循环中是否被经常使用?

指令执行代价：各变量在循环内需要访问内存单元的次数+1。

op Ri, Ri	1
op Ri, M	2
op Ri, *Ri	2
op Ri, *Mi	3

指令代价计算公式

$$S = \left(\sum_{B \in W} \text{Refer}(X, B) + 2 * \text{Live}(X, B) \right) * A - 4$$

其中：

函数Refer(X, B)记录的是基本块B中对X定值前引用X的次数；

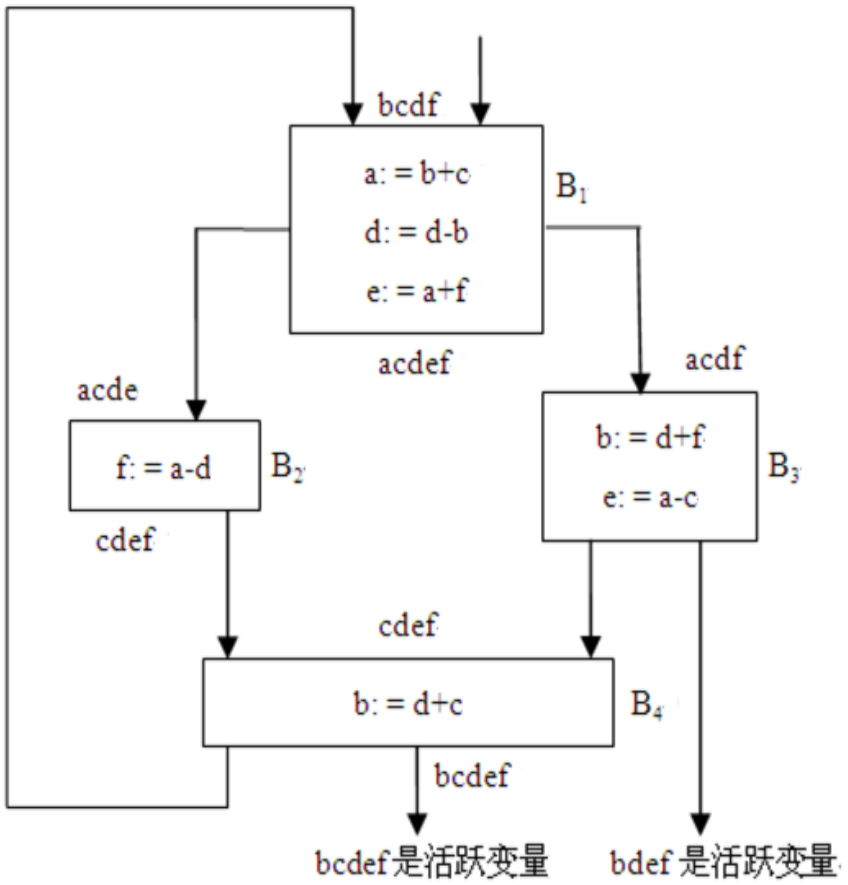
函数Live(X, B)记录的是基本块B中被定值的X在B的出口之后是否活跃，若X是活跃变量，则返回1，否则返回0。

A是变量执行概率。简单起见，通常设置为1。

通常为S值较大的变量固定分配寄存器，以节省内存访问时间。

寄存器分配–例子

	Refer				Live			
	B1	B2	B3	B4	B1	B2	B3	B4
a	0	1	1	0	1	0	0	0
b	2	0	0	0	0	0	1	1
c	1	0	1	1	0	0	0	0
d	1	1	1	1	1	0	0	0
e	0	0	0	0	1	0	1	0
f	1	0	1	0	0	1	0	0



寄存器分配–例子

	Refer				Live			
	B1	B2	B3	B4	B1	B2	B3	B4
a	0	1	1	0	1	0	0	0
b	2	0	0	0	0	0	1	1
c	1	0	1	1	0	0	0	0
d	1	1	1	1	1	0	0	0
e	0	0	0	0	1	0	1	0
f	1	0	1	0	0	1	0	0

	S
a	=(0+1+1+0+2*1+2*0+2*0+2*0)*1-4=0
b	=(2+0+0+0+2*0+2*0+2*1+2*1)*1-4=2
c	=(1+0+1+1+2*0+2*0+2*0+2*0)*1-4=-1
d	=(1+1+1+1+2*1+2*0+2*0+2*0)*1-4=2
e	=(0+0+0+0+2*1+2*0+2*1+2*0)*1-4=0
f	=(1+0+1+0+2*0+2*1+2*0+2*0)*1-4=0

按照S的大小，将R0分配给D，R1分配给B，R2分配给A、E、F中任一个。

其余变量从剩余的寄存器中分配。

调整后的代码生成算法

(1) 循环中的目标代码，凡是涉及到已固定分配寄存器的变量，就用分配给它的寄存器来表示。

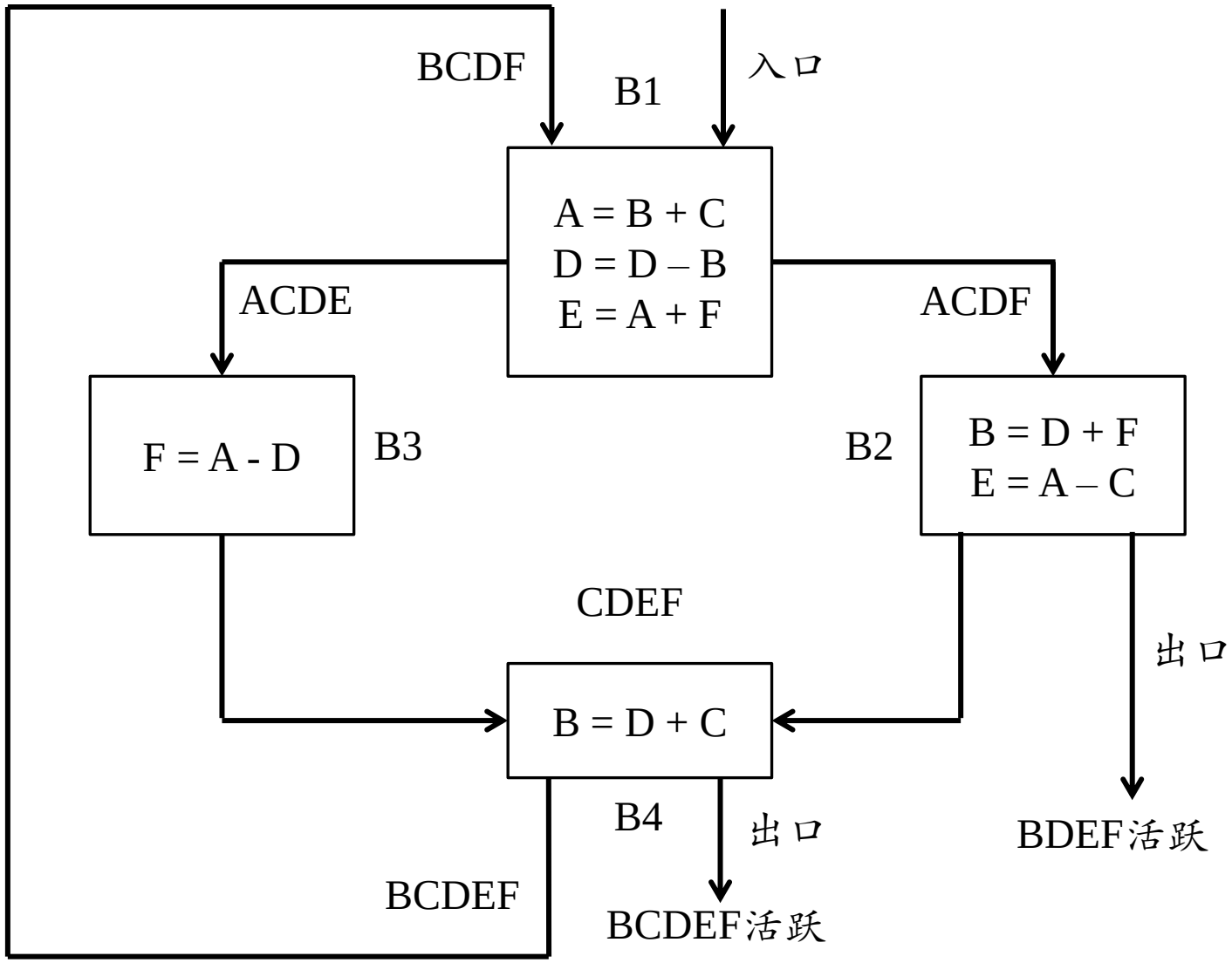
(2) 如果某些已经分配寄存器的变量在循环入口前是活跃的，那么在循环入口之前，要生成把它们的值分别取到相应寄存器中的目标代码。

(3) 如果某些已经分配寄存器的变量在循环出口之后是活跃的，那么在循环出口之后，要生成把它们的值存放到内存单元的目标代码。

(4) 在循环中每个基本块的出口，对未固定分配寄存器的变量，仍然按照以前的算法生成目标代码，并把它们在寄存器中的值存放到内存单元中。

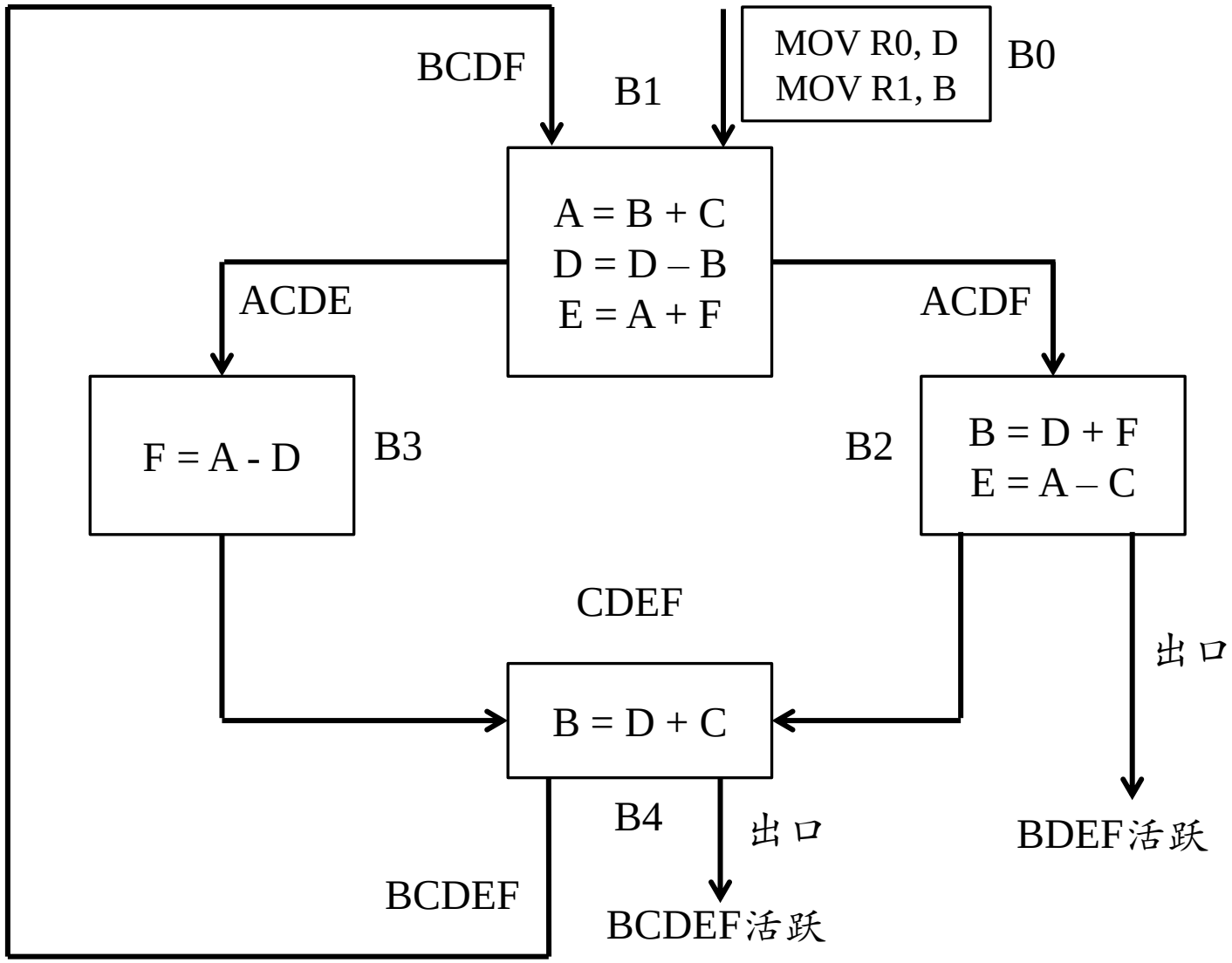
代码生成–例子

R0分配给D
R1分配给B
R2分配给A



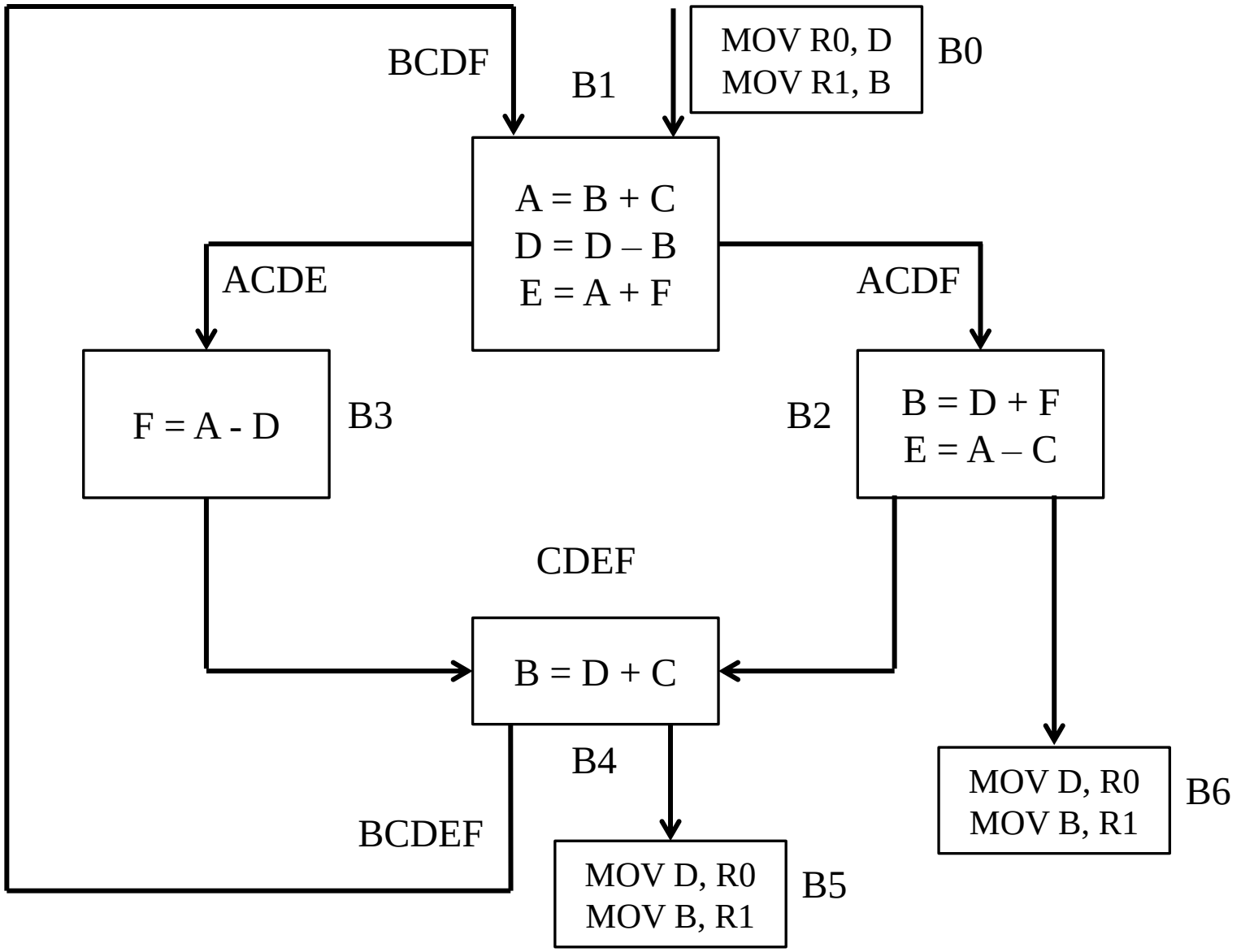
代码生成–添加入口代码B0

R0分配给D
R1分配给B
R2分配给A



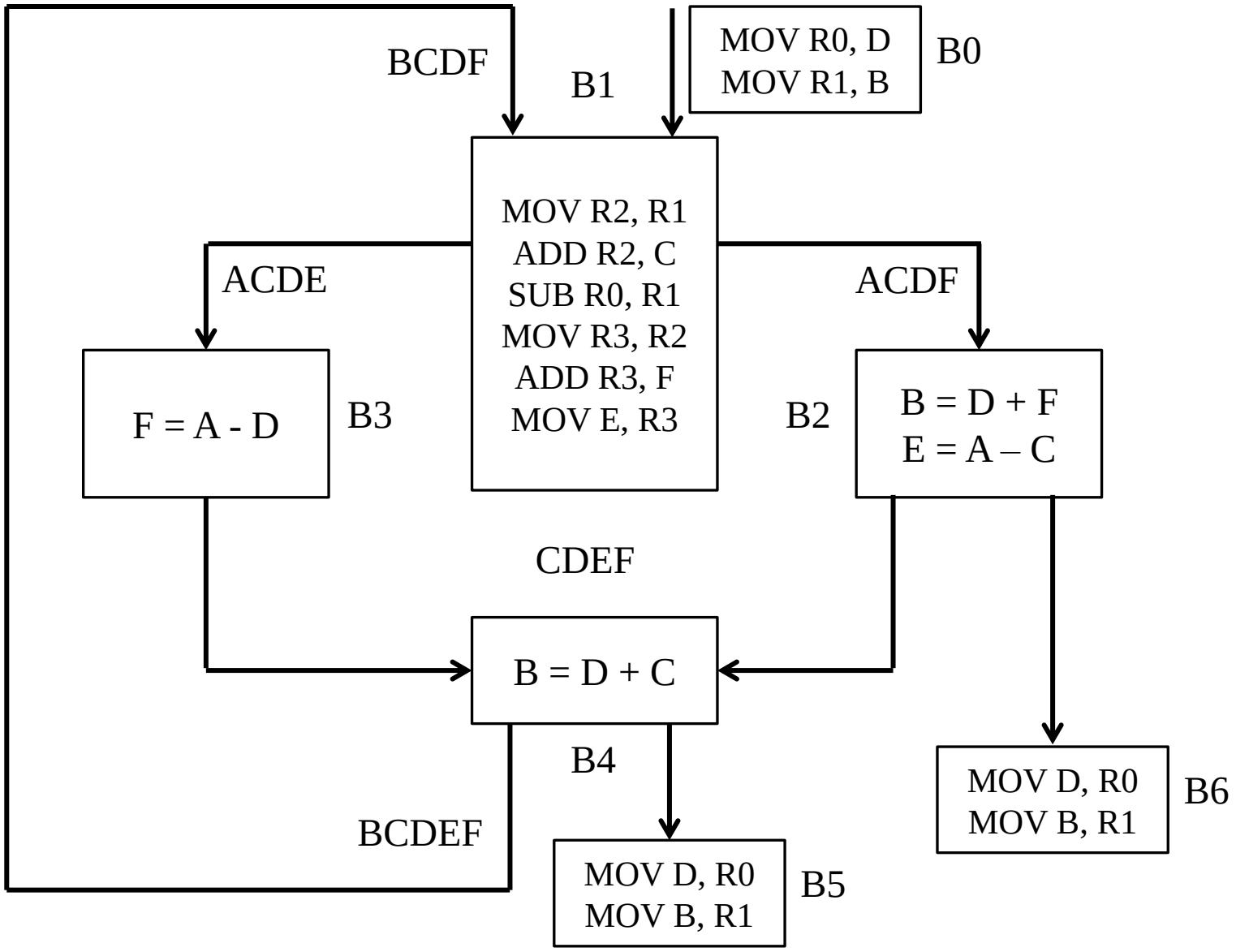
代码生成–添加出口代码B5、B6

R0分配给D
R1分配给B
R2分配给A



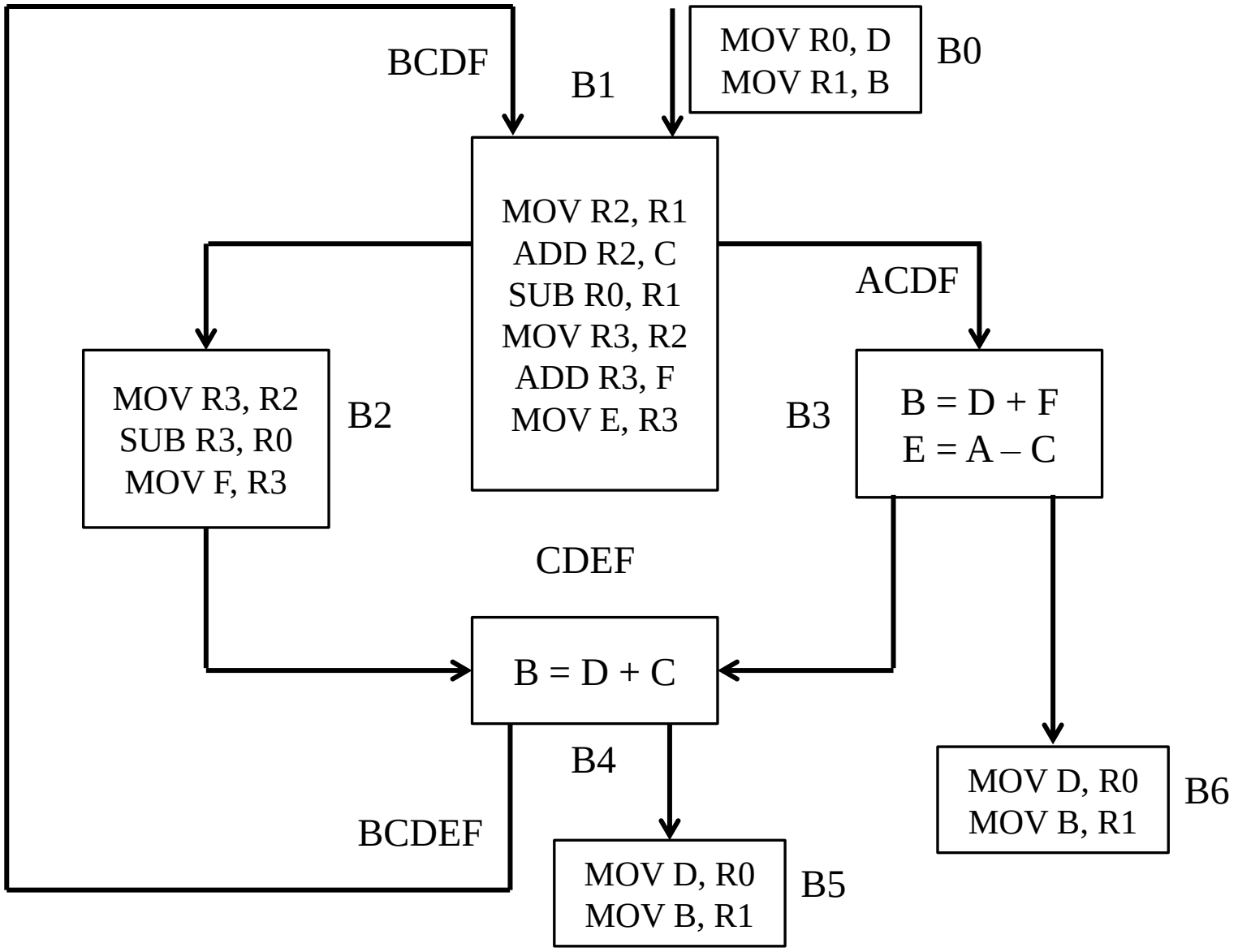
代码生成-生成B1

R0分配给D
R1分配给B
R2分配给A



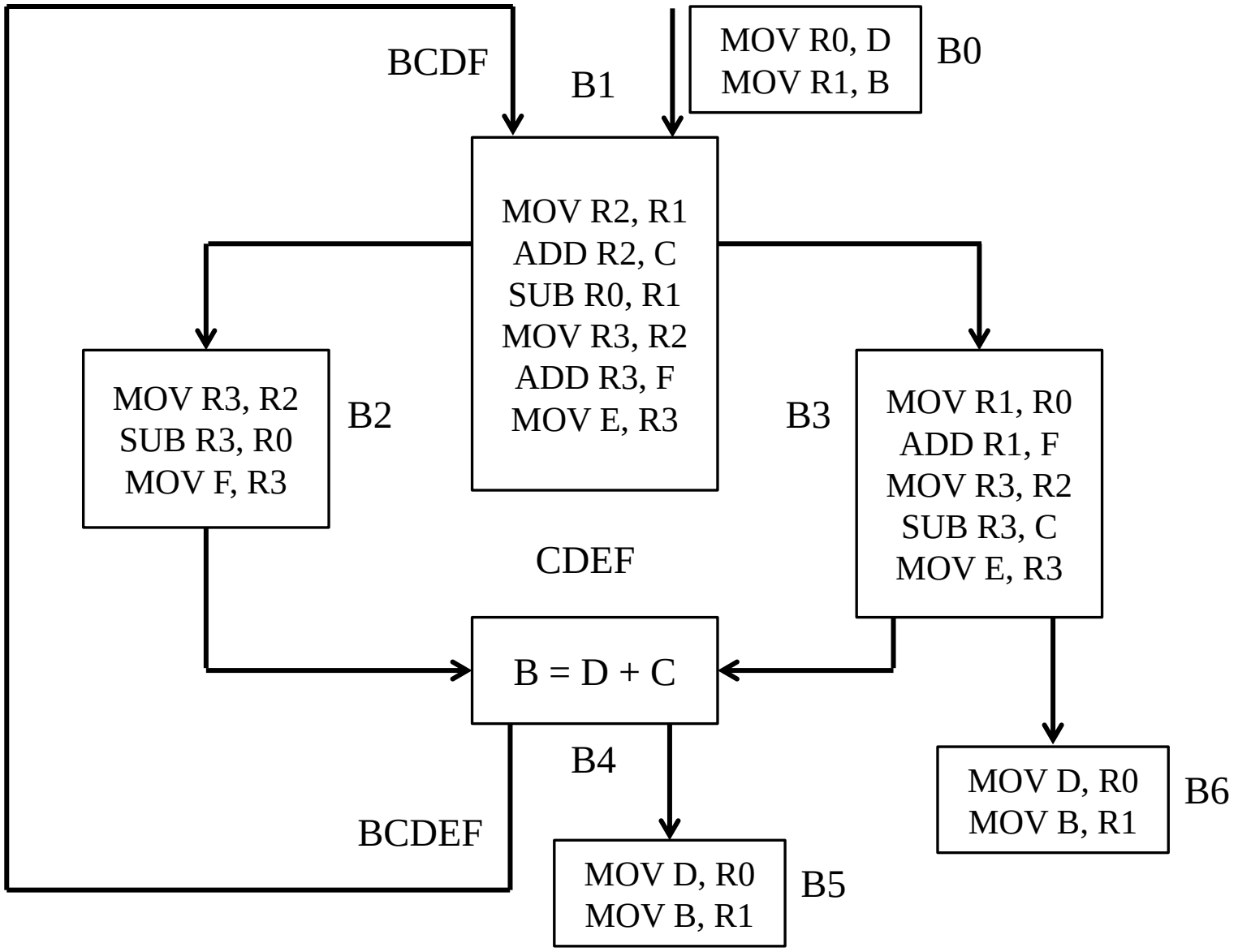
代码生成-生成B2

R0分配给D
R1分配给B
R2分配给A



代码生成-生成B3

R0分配给D
R1分配给B
R2分配给A



代码生成-生成B4

R0分配给D
R1分配给B
R2分配给A