机器无关代码优化
Part5：数据流与活跃变量分析

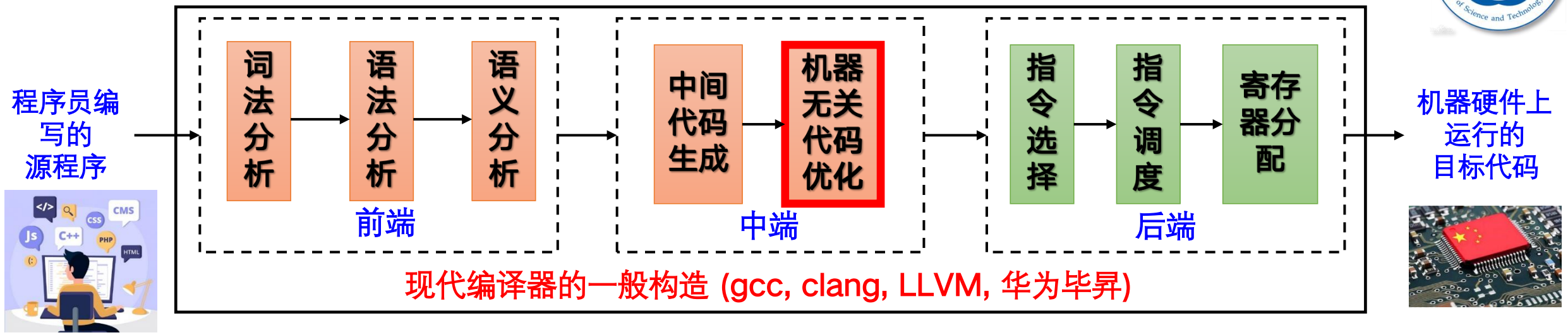
徐 伟

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

计算机科学与技术学院

2025年5月8日

本节提纲



- 活跃变量定义及应用
- 活跃变量分析算法
- 示例驱动的分析流程

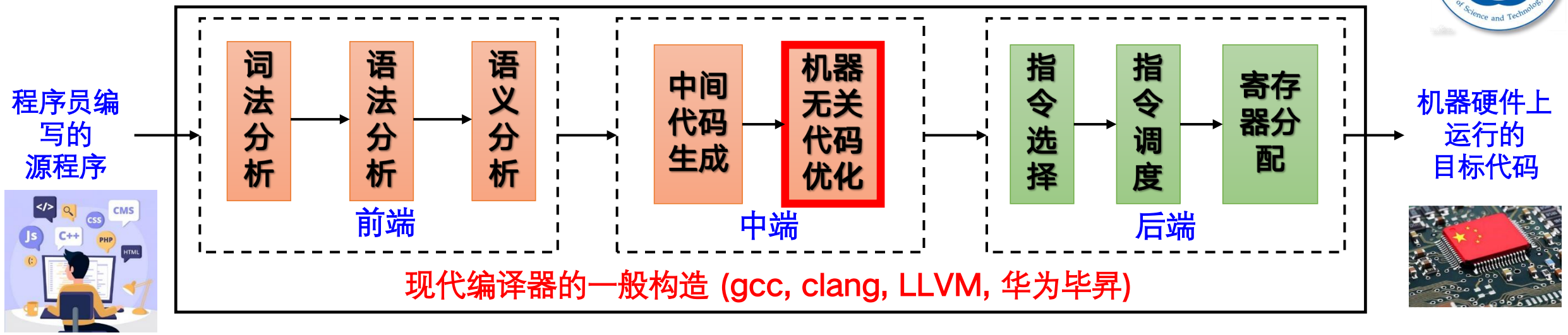
活跃变量

- 定义：
  - 对于变量x和程序点p，如果x的值在*p*点开始的某条执行路径上被引用，则说x在*p*点活跃（live），否则称x在*p*点已经死亡（dead）

活跃变量分析的应用

- **为基本块分配寄存器**
  - 如果所有寄存器都被占用，且还需要申请一个寄存器，则应该考虑使用已经存放死亡值的寄存器
  - 如果一个值在基本块结尾处是死的，就不必在结尾处保存这个值了

本节提纲



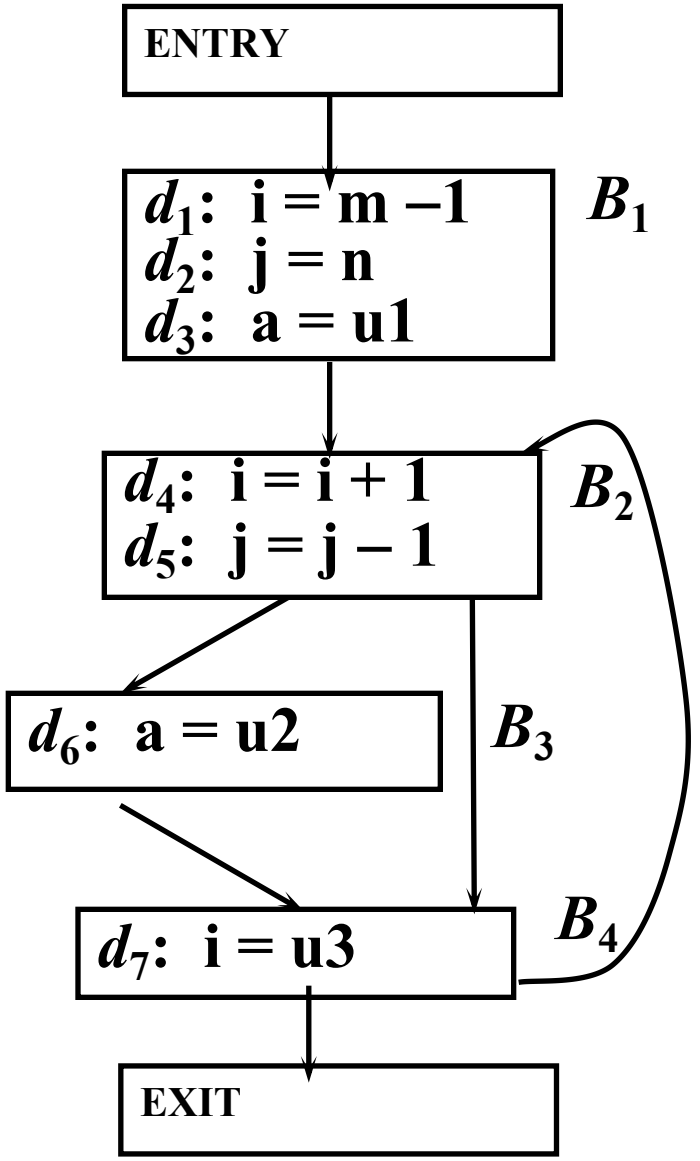
- 活跃变量定义及应用
- **活跃变量分析算法**
- **示例驱动的分析流程**

活跃变量分析

- 相关定义
  - IN[B]：块B开始点的活跃变量集合
  - OUT[B]：块B结束点的活跃变量集合
  - use_B：块B中有引用，且在引用前在B中没有被定值的变量集
  - def_B：块B中有定值，且该定值前在B中没有被引用的变量集

use与def的计算

- 例
  - $use[B_1] = \{ m, n, u1 \}$
  - $def[B_1] = \{i, j, a\}$
  - $\boldsymbol{use[B_2] = \{ i, j \}}$
  - $\boldsymbol{def[B_2] = \{\}}$
  - $use[B_3] = \{ u2\}$
  - $def[B_3] = \{a\}$
  - $use[B_4] = \{ u3 \}$
  - $def[B_4] = \{i\}$



求解方程组

- **活跃变量分析逆向数据流等式**
  - IN [EXIT] = ∅
    - 边界条件：程序出口处没有活跃变量
  - **OUT[*B*] = ∪ $_{S是B的后继}$ IN [*S*]**
  - **IN [*B*] = use_B ∪ (OUT [*B*] – def_B)**
    - 入口处活跃：1) 在B中重定值之前被使用；2)离开时活跃且没有在B中被定值
- **和到达－定值等式之间的联系与区别**
  - **都以集合并算符作为它们的汇合算符**
  - **信息流动方向相反，IN和OUT的作用相互交换**
  - ***use*和*def*分别取代*gen*和*kill***
  - **仍然需要最小解**

活跃变量的迭代计算算法

输入：流图G，其中每个基本块B的use和def都已计算

输出：IN[B]和OUT[B]

IN[EXIT] = Ø;

for (除了EXIT以外的每个块B) IN[B] = Ø;

while (某个IN值出现变化) {

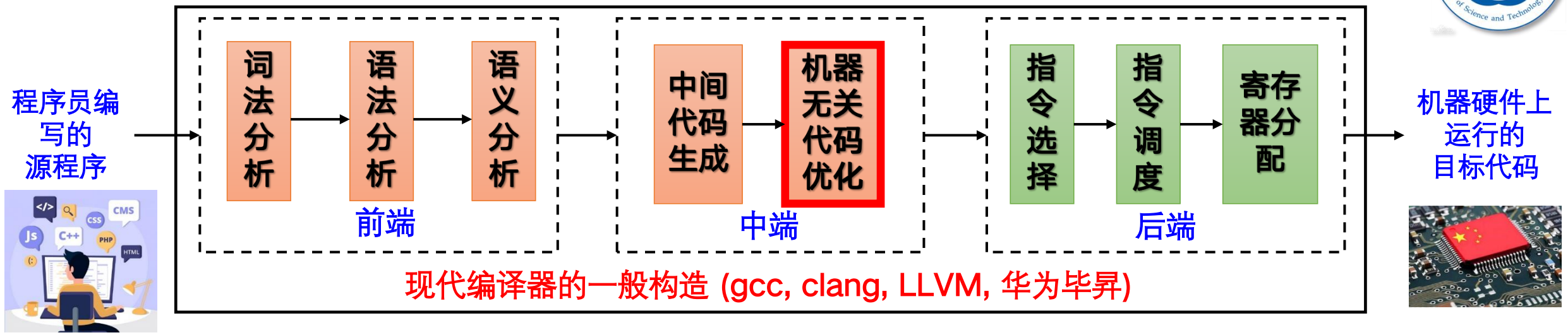
for (除了EXIT以外的每个块B) {

$$\mathrm{OUT}[B] = \cup_{S\text{是}B\text{的后继}} \mathrm{IN}[S]$$

$$\mathrm{IN}[B] = use_B \cup (\mathrm{OUT}[B] - def_B);$$

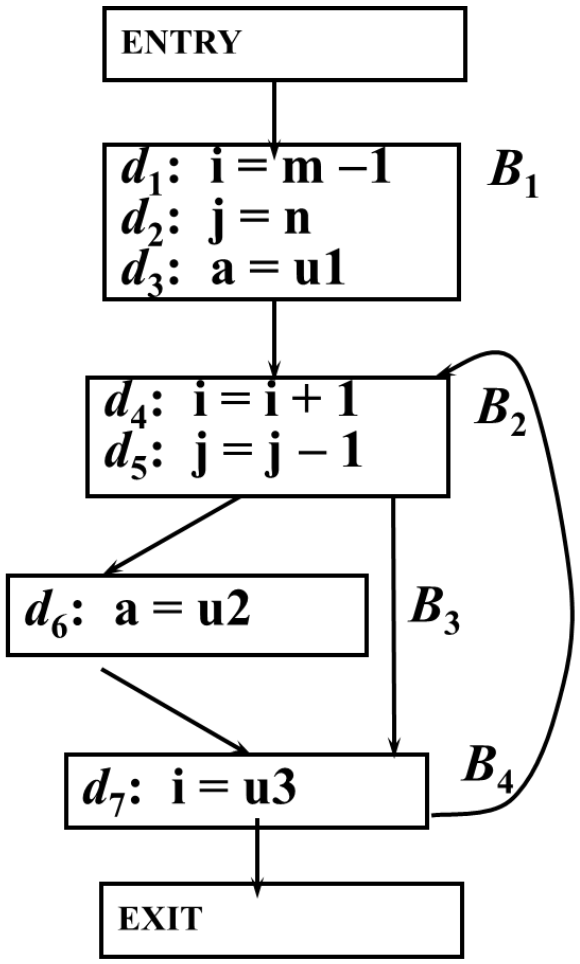
}}

本节提纲



- 活跃变量定义及应用
- **活跃变量分析算法**
- 示例驱动的分析流程

活跃变量分析-举例1



$use[B_1] = \{ m, n, u1 \}$
$def[B_1] = \{i, j, a\}$
$use[B_2] = \{ i, j \}$
$def[B_2] = \{\}$
$use[B_3] = \{ u2\}$
$def[B_3] = \{a\}$
$use[B_4] = \{ u3 \}$
$def[B_4] = \{i\}$

IN[EXIT] = Ø;

for (除了EXIT以外的每个块B) IN[B] = Ø;

while (某个IN值出现变化) {

for (除了EXIT以外的每个块B) {

$OUT[B] = \cup_{S是B的后继} IN[S]$

$IN[B] = use_B \cup (OUT[B] - def_B)$;

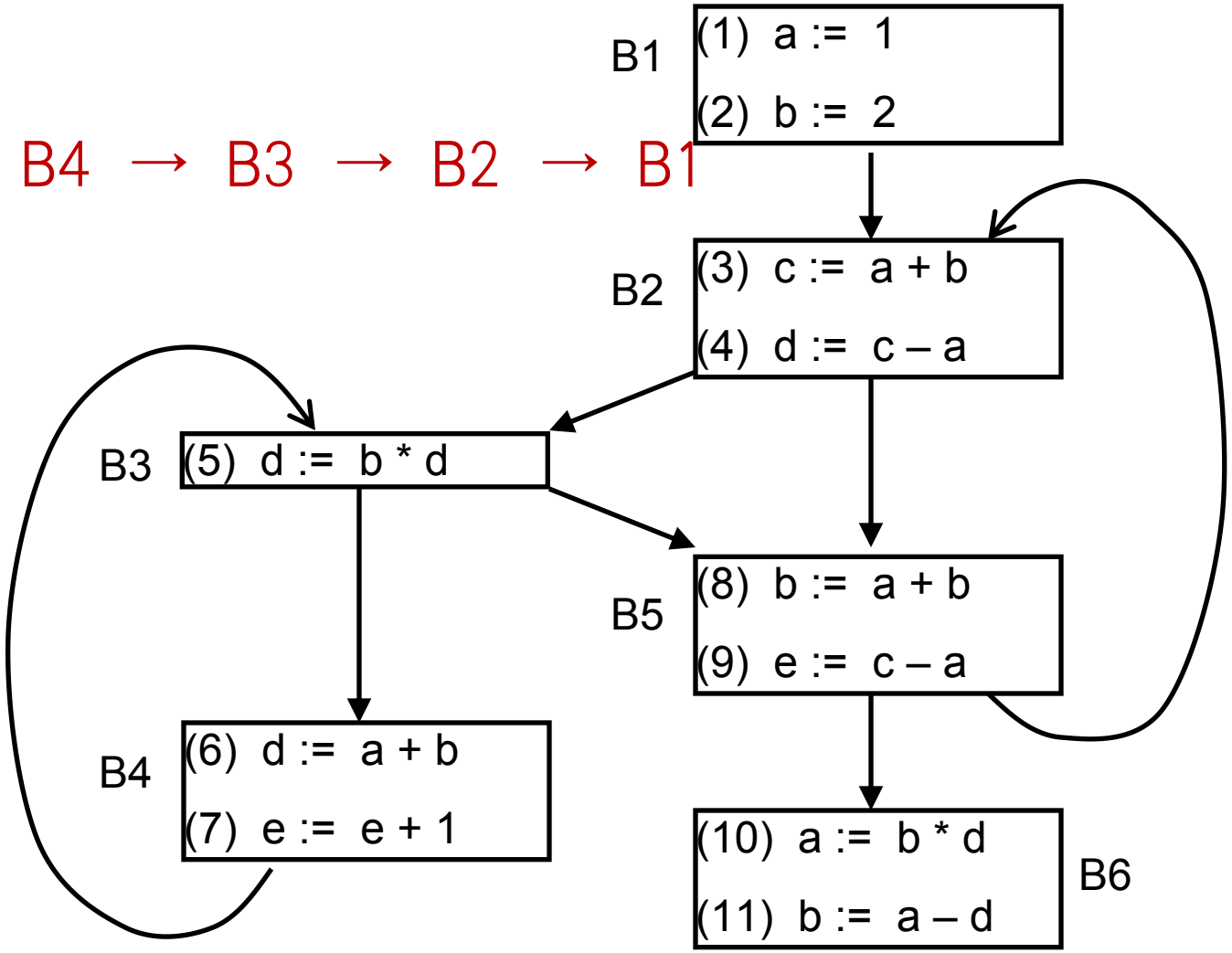
}}

	$OUT[B]^1$	$IN[B]^1$	$OUT[B]^2$	$IN[B]^2$	$OUT[B]^3$	$IN[B]^3$
B_4		u3	i，j，u2，u3	j，u2，u3	i，j，u2，u3	j，u2，u3
B_3	u3	u2，u3	j，u2，u3	j，u2，u3	j，u2，u3	j，u2，u3
B_2	u2，u3	i，j，u2，u3	j，u2，u3	i，j，u2，u3	j，u2，u3	i，j，u2，u3
B_1	i，j，u2，u3	m，n，u1，u2，u3	i，j，u2，u3	m，n，u1，u2，u3	i，j，u2，u3	m，n，u1，u2，u3

活跃变量分析-举例2

计算次序

* B6 → B5 → B4 → B3 → B2 → B1



基本块出口活跃变量

- 各基本块USE和DEF如下，

  USE[B1] = { } ; DEF[B1] = { a, b }

  USE[B2] = { a, b } ; DEF[B2] = { c, d }

  USE[B3] = { b, d } ; DEF[B3] = { }

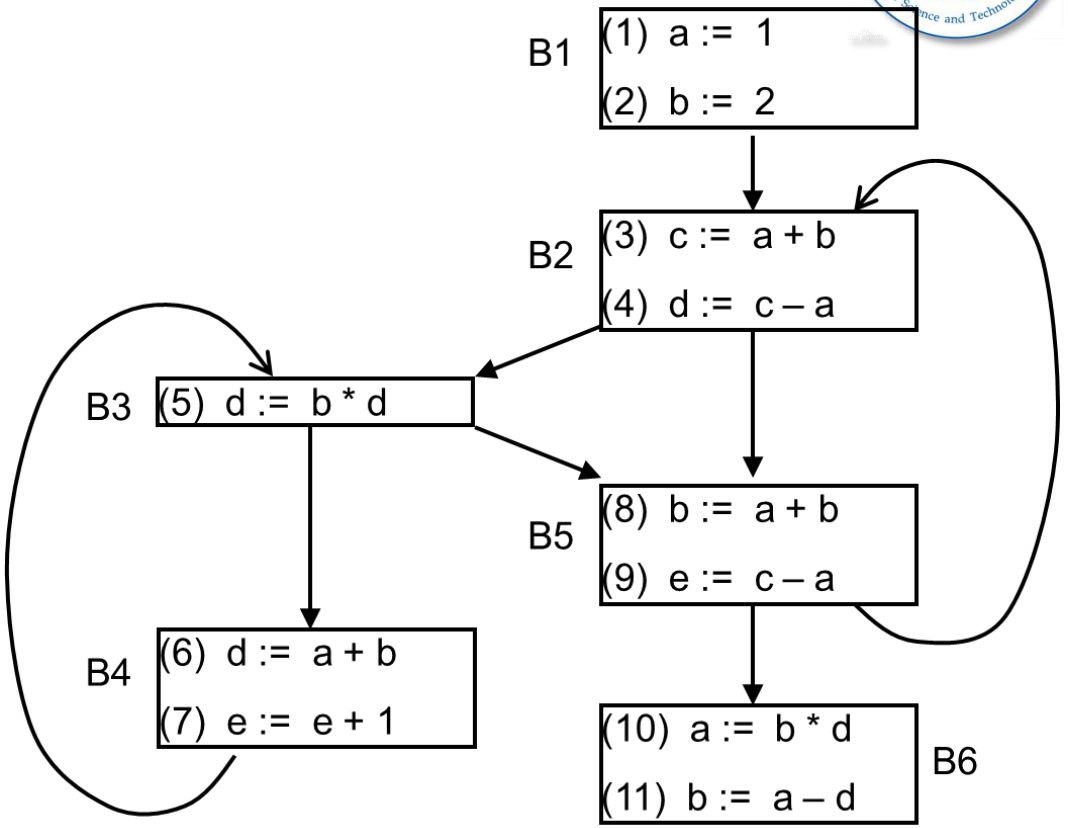
  USE[B4] = { a, b, e } ; DEF[B4] = { d }

  USE[B5] = { a, b, c } ; DEF[B5] = { e }

  USE[B6] = { b, d } ; DEF[B6] = { a }

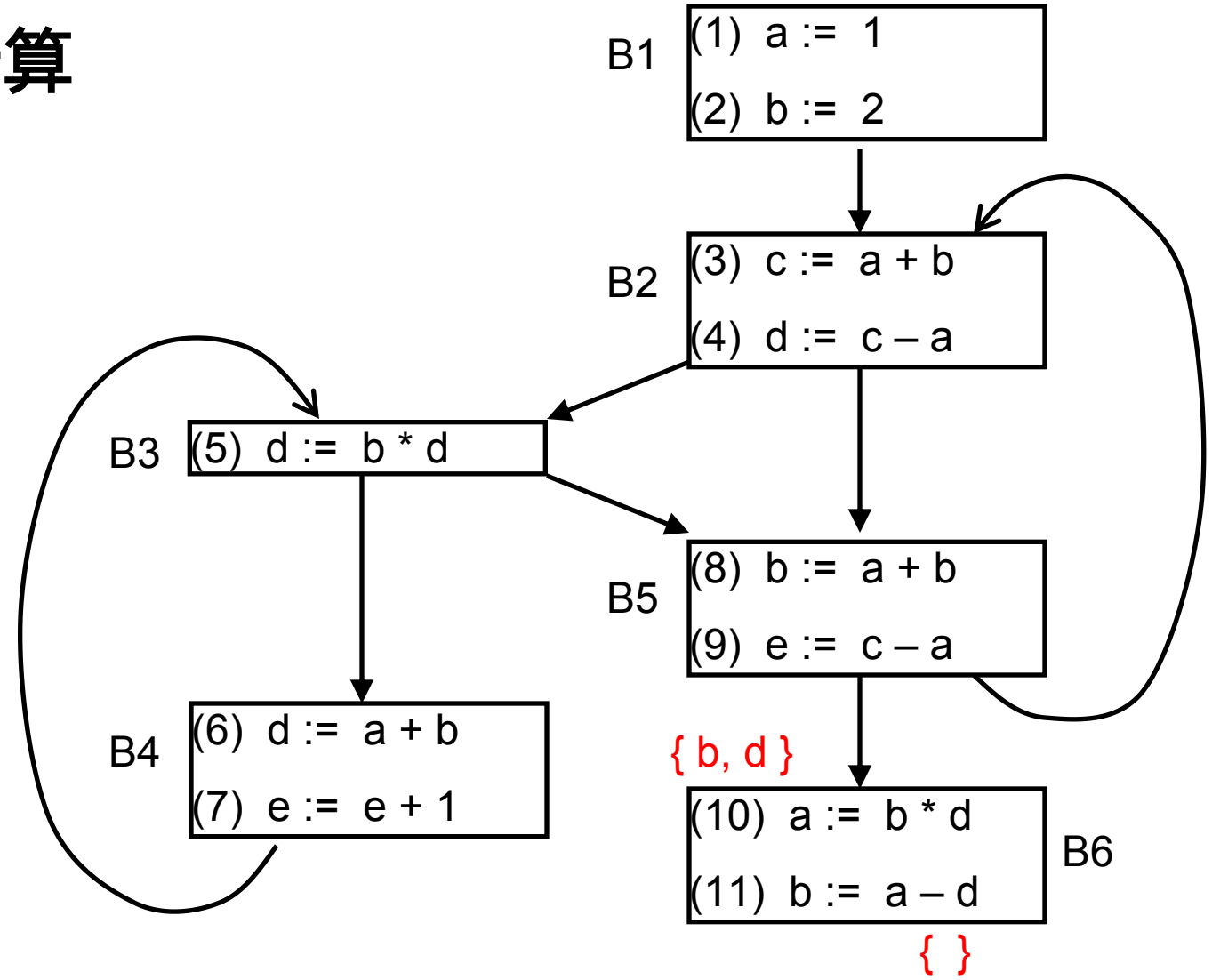
- 初始值，all B, IN[B] = { },

  OUT[B6]={ }//出口块



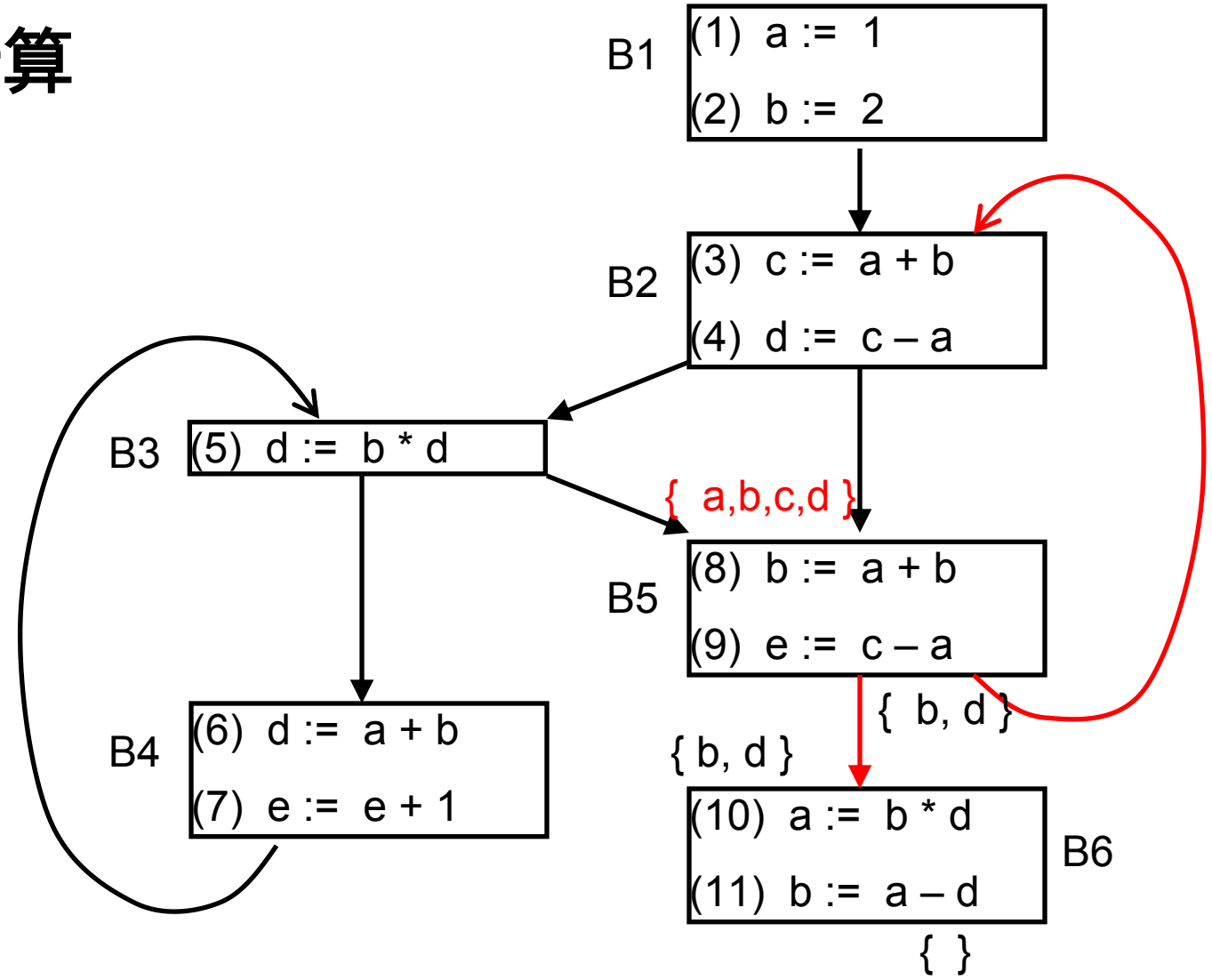
基本块出口活跃变量

- 第一次迭代计算



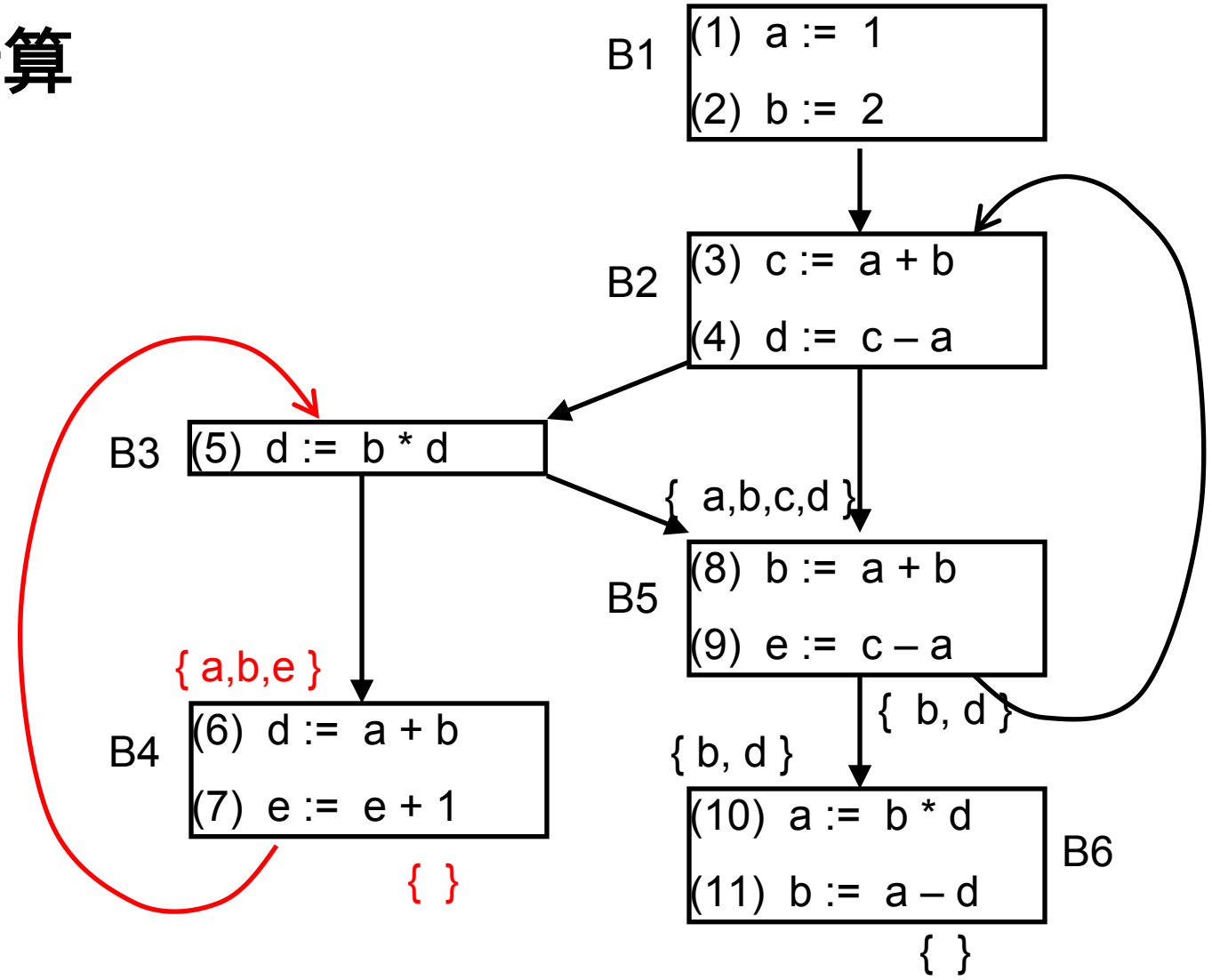
基本块出口活跃变量

- 第一次迭代计算



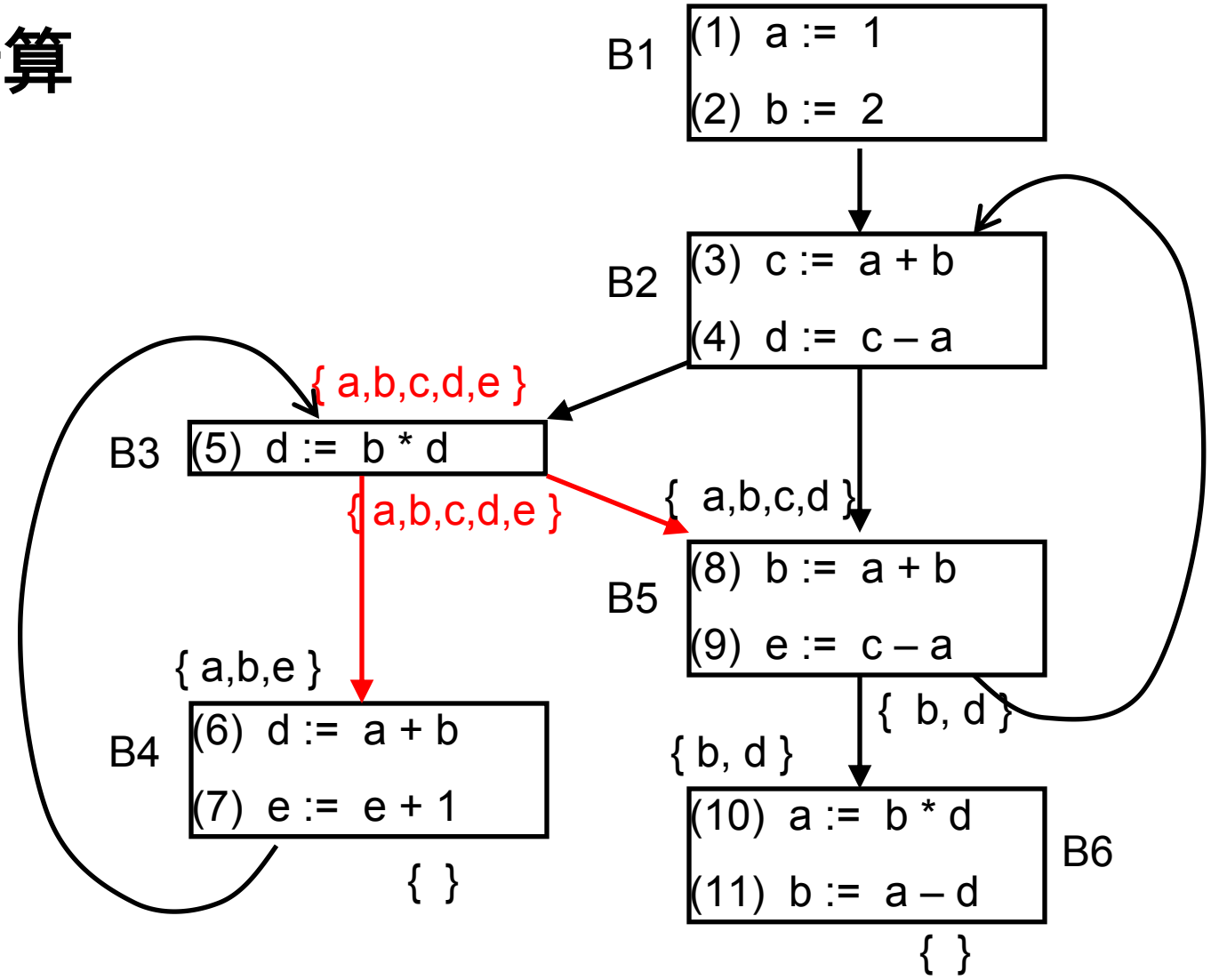
基本块出口活跃变量

- 第一次迭代计算



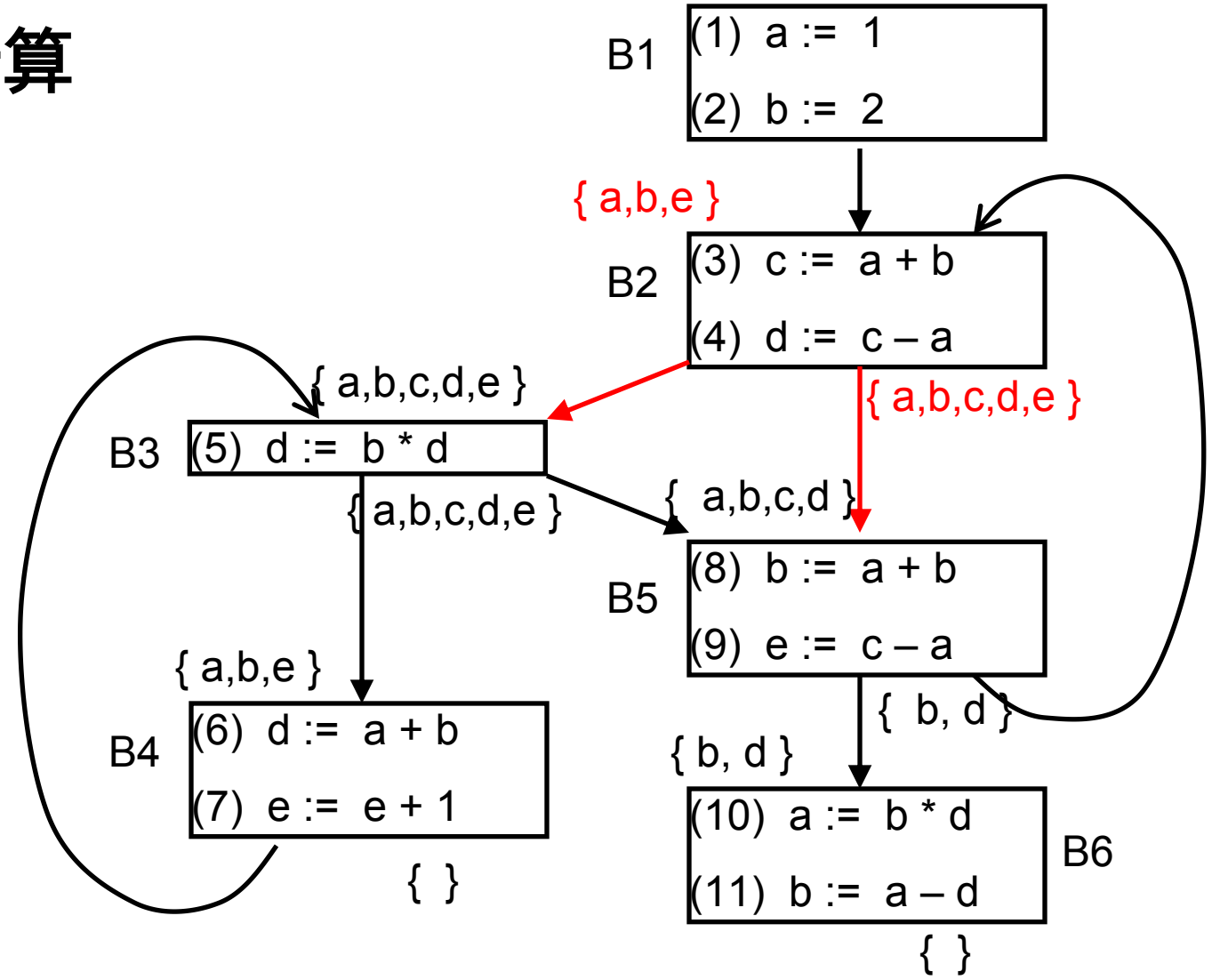
基本块出口活跃变量

- 第一次迭代计算



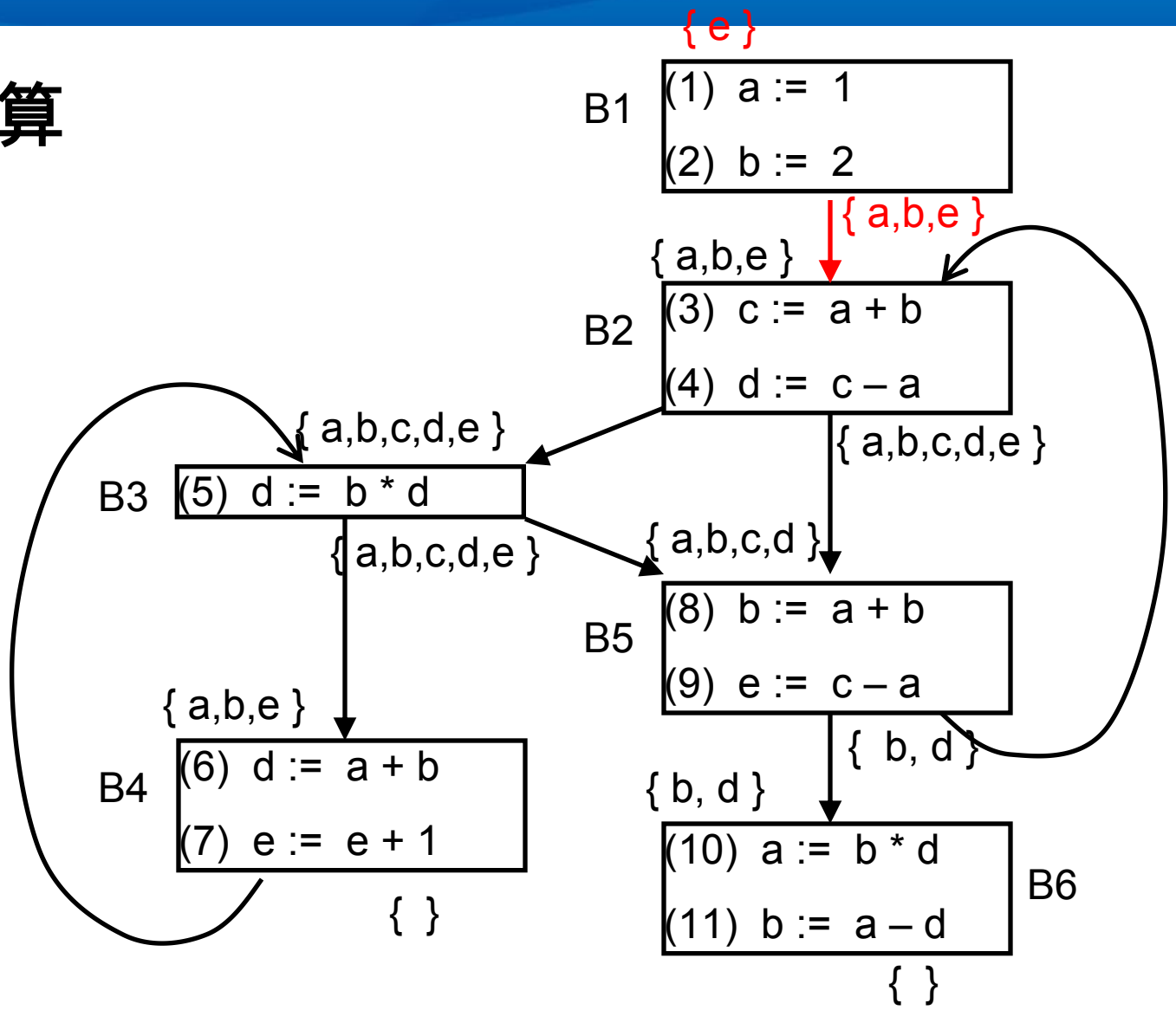
基本块出口活跃变量

- 第一次迭代计算



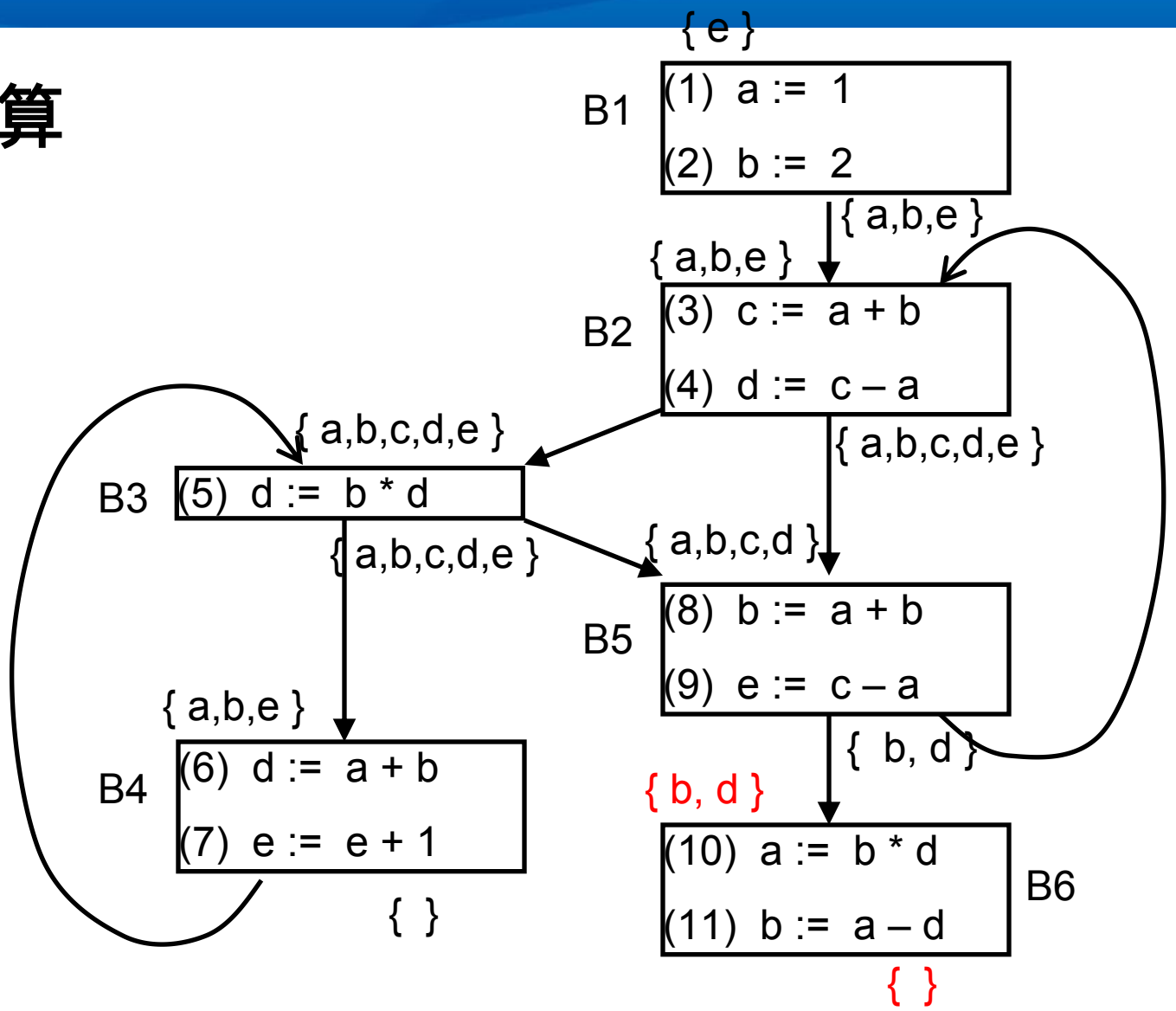
基本块出口活跃变量

- 第一次迭代计算



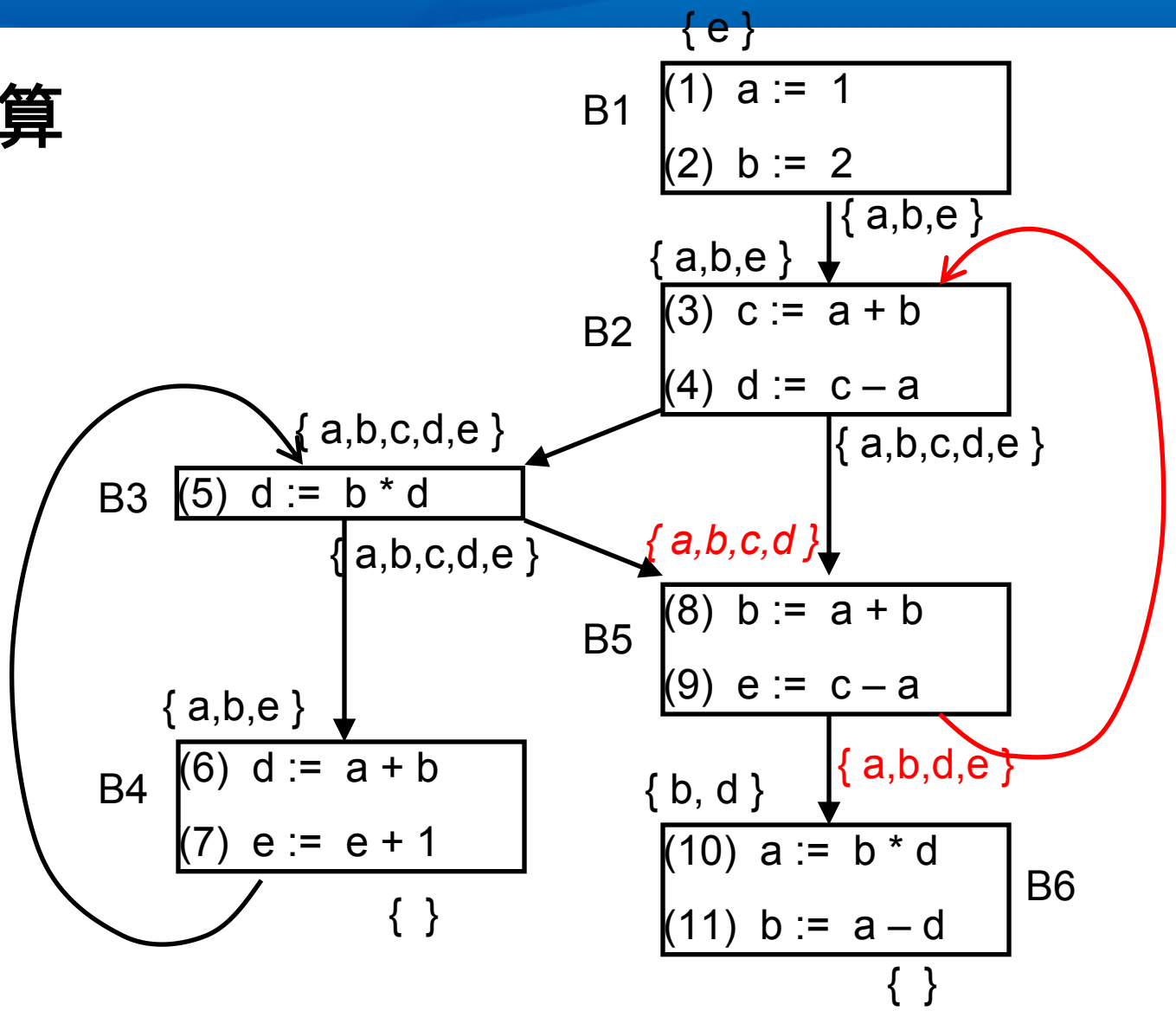
基本块出口活跃变量

- 第二次迭代计算



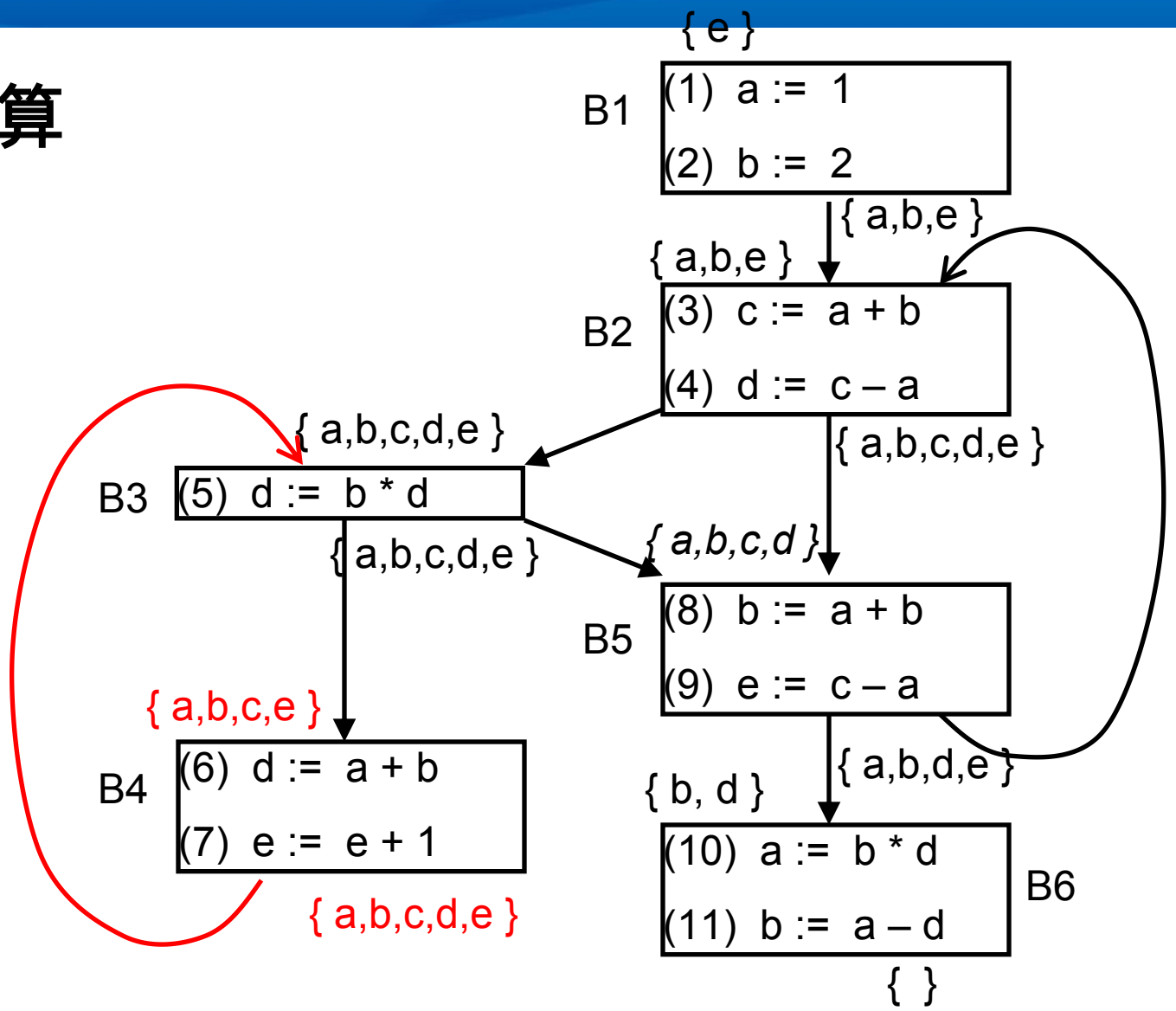
基本块出口活跃变量

- 第二次迭代计算



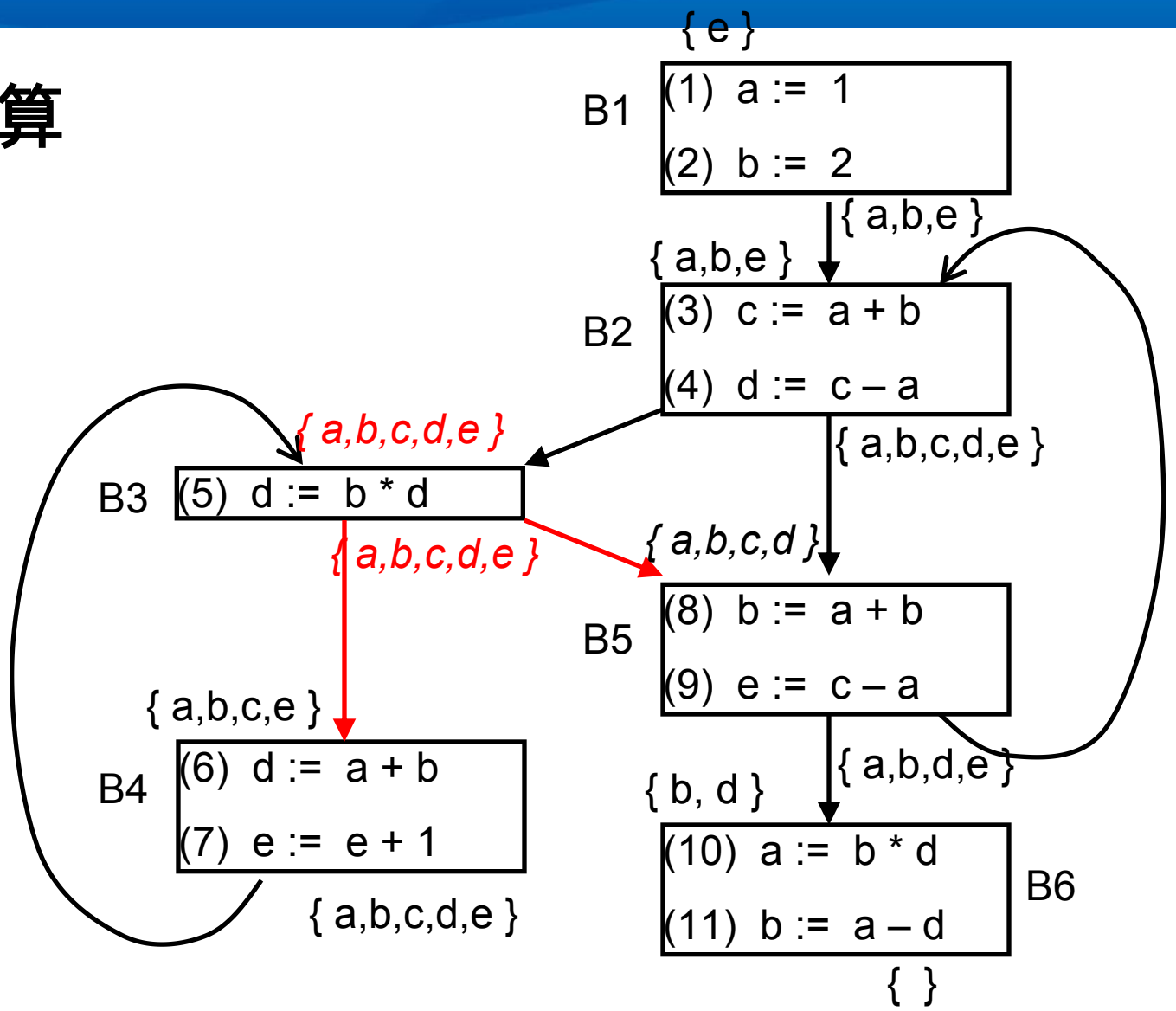
基本块出口活跃变量

- 第二次迭代计算



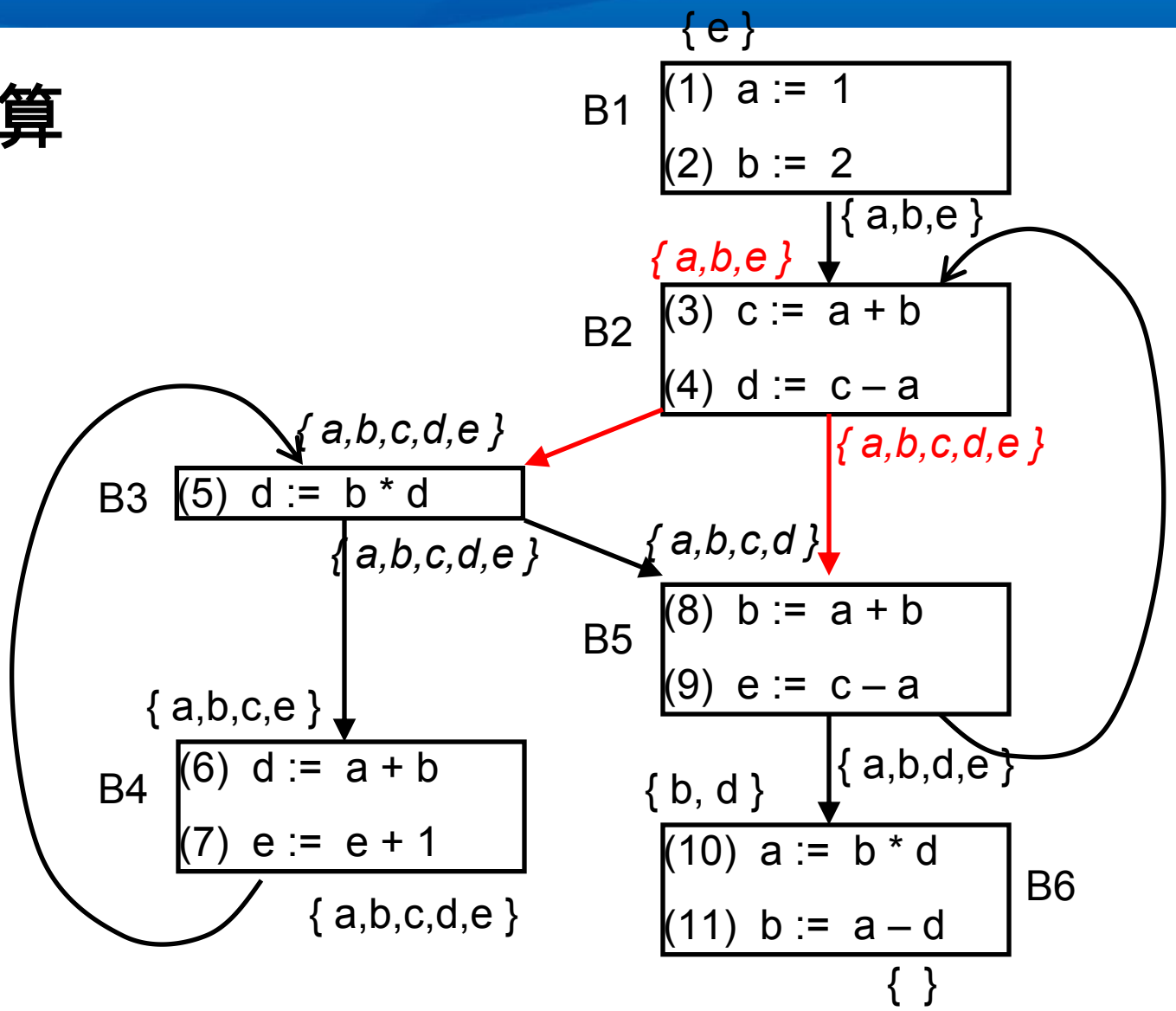
基本块出口活跃变量

- 第二次迭代计算



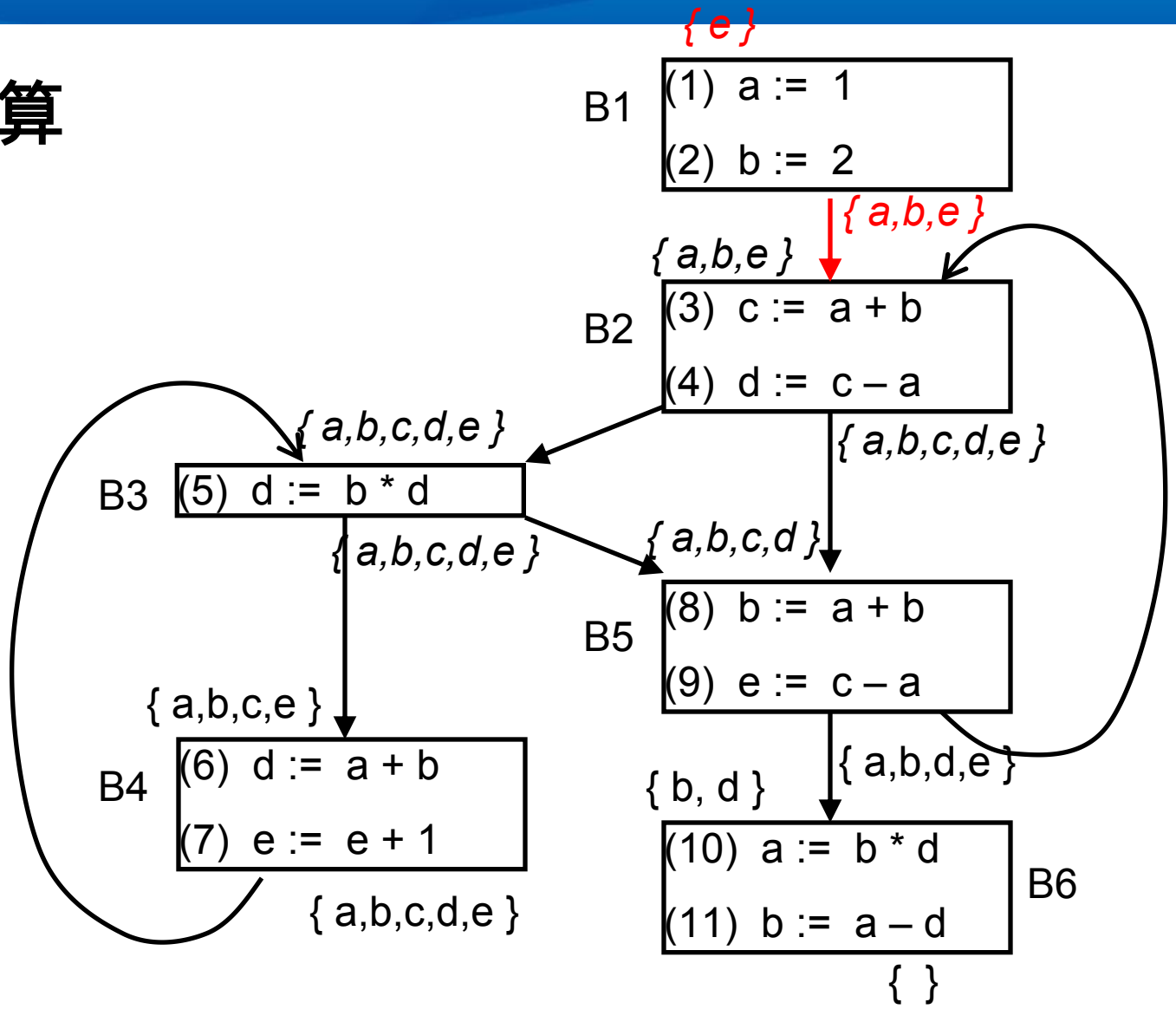
基本块出口活跃变量

- 第二次迭代计算



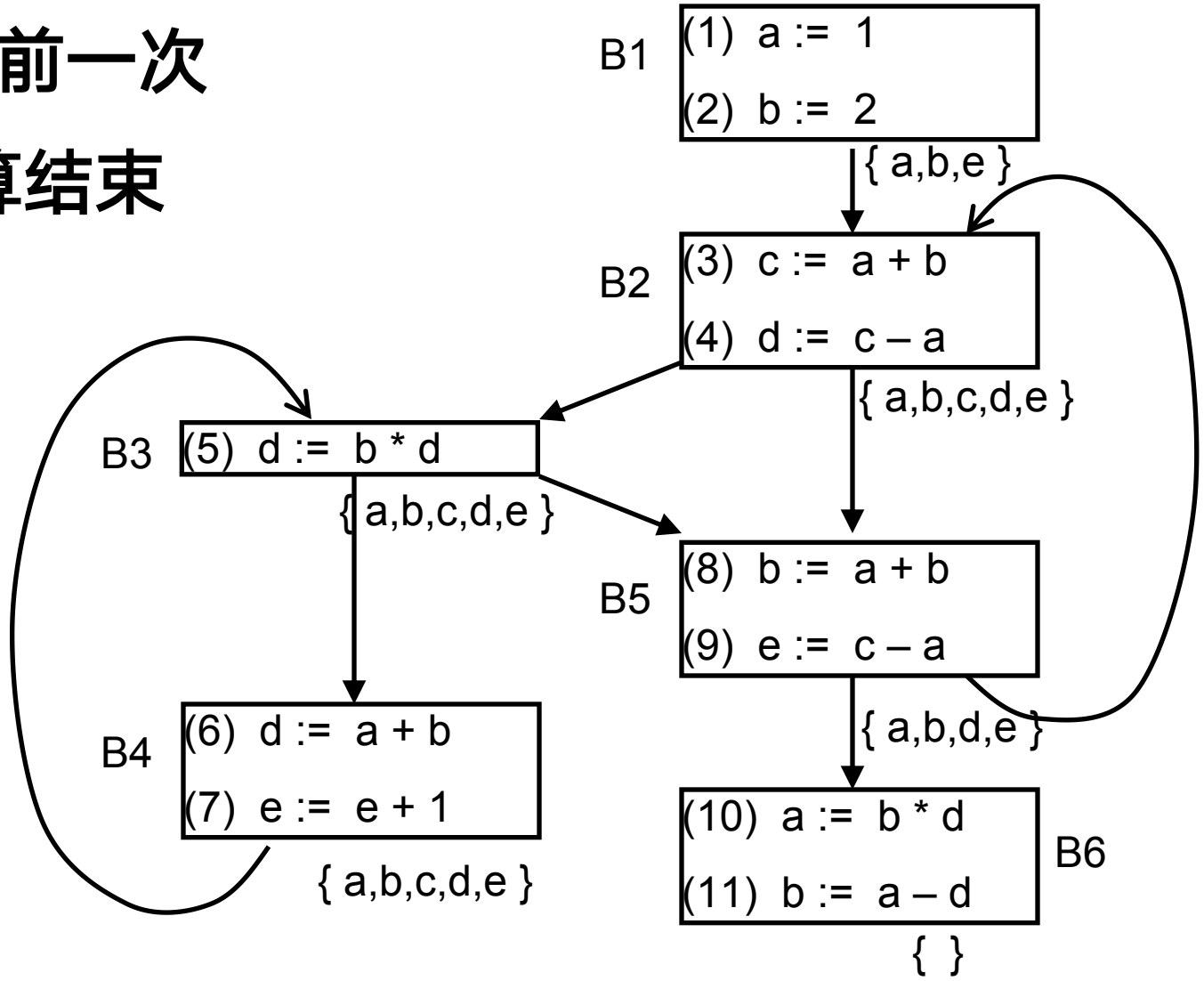
基本块出口活跃变量

- 第二次迭代计算



基本块出口活跃变量

- 第三次迭代与前一次结果一样，计算结束

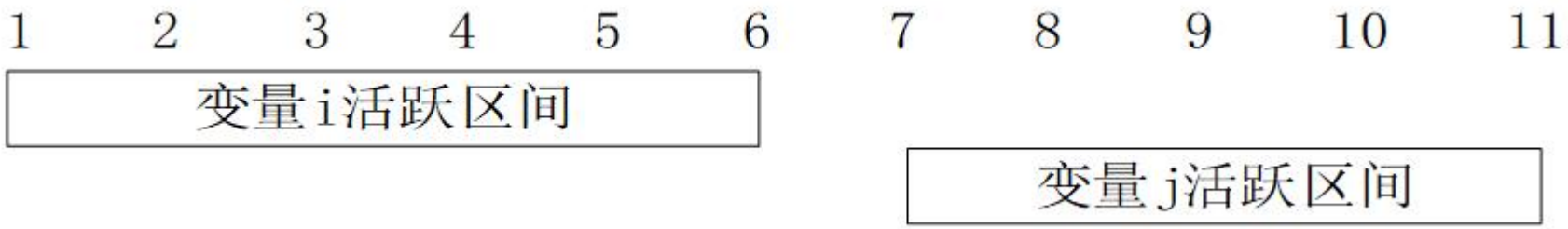


活跃变量区间

- 计算活跃区间来定位变量在程序执行中的生命周期，即变量在哪个时间段需要被寄存器保存

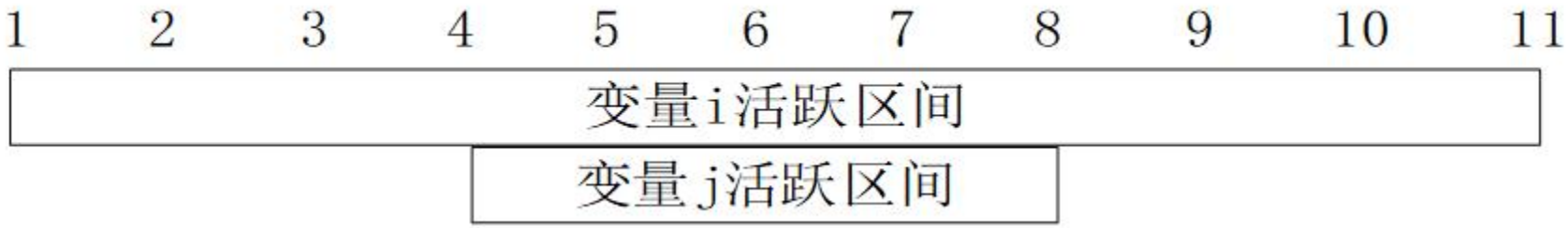
- 对于变量x首次被定义的点称为start，最后被引用的点称为end，区间[start, end]称为x的活跃区间

活跃变量区间



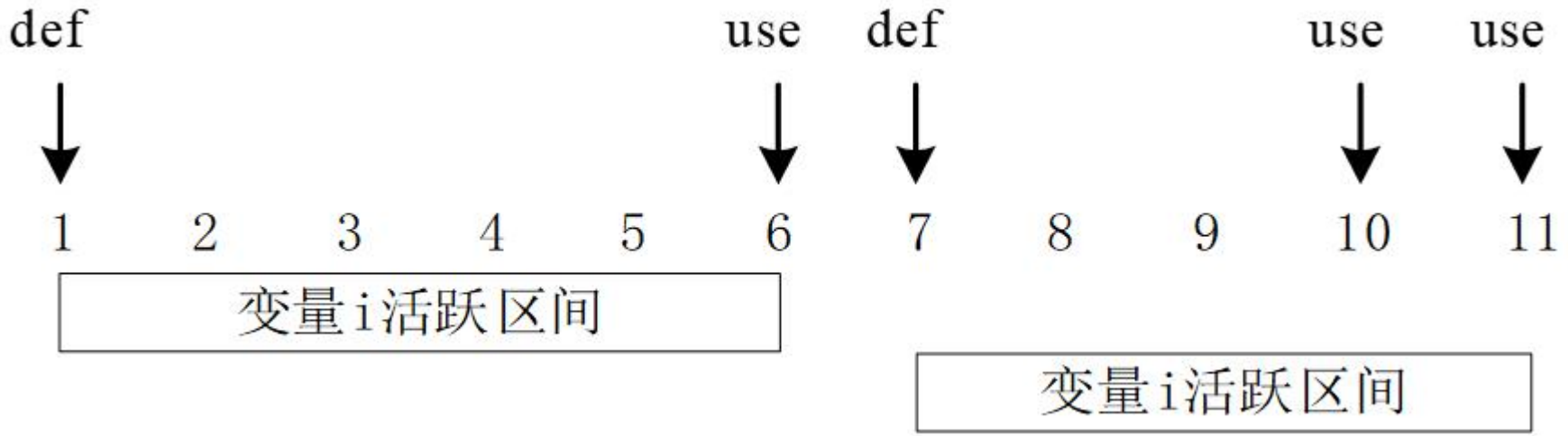
- 变量i的活跃区间是[1,6]，j的活跃区间是[7,11]
- 两个活跃区间并无交集，可以通过一个寄存器完成寄存器分配

活跃变量区间



- i的活跃区间是[1,11]，j的活跃区间是[4,8]
- 活跃区间之间存在交集相互干涉
- 需要两个寄存器才能完成寄存器分配

活跃变量区间



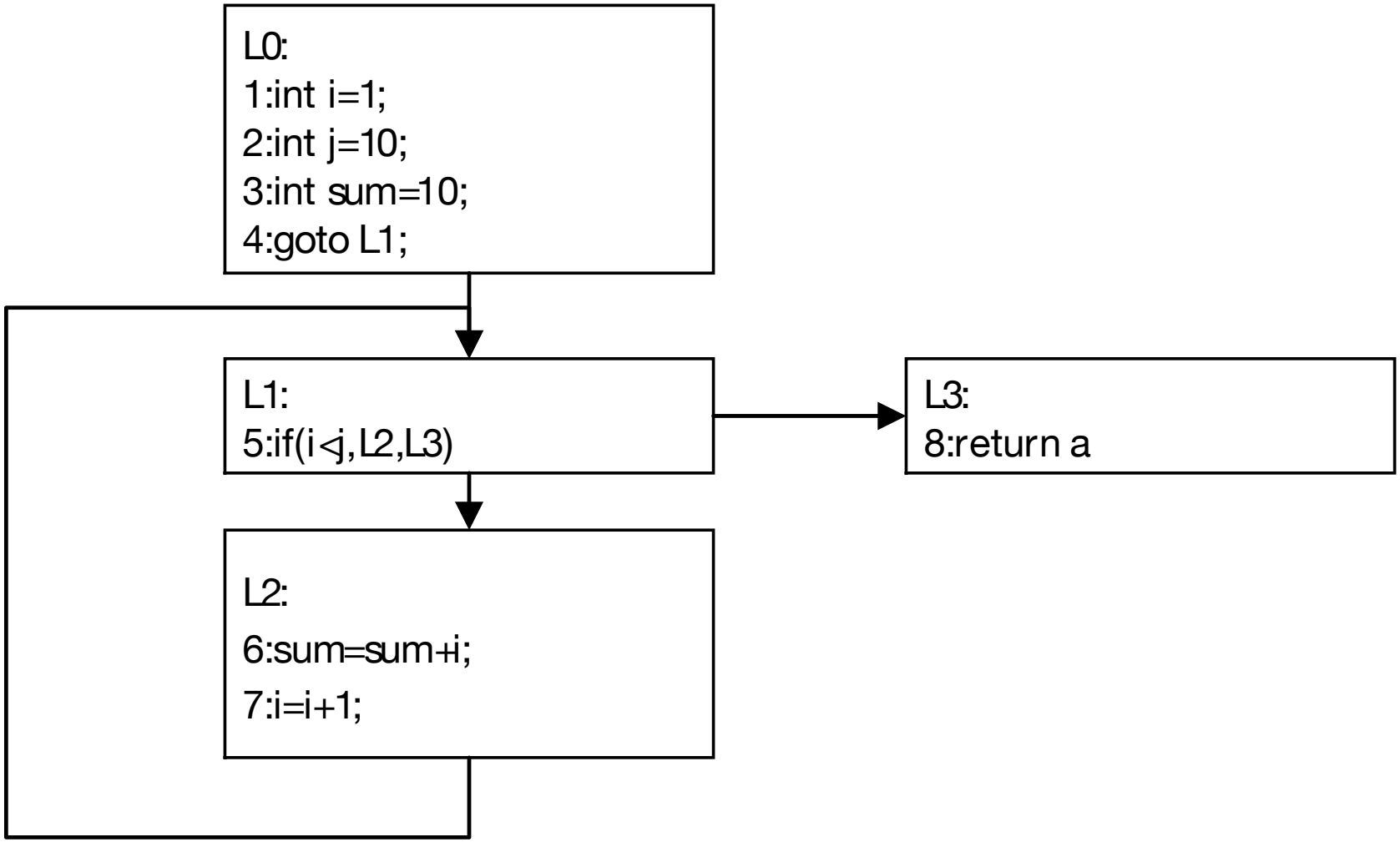
- [start, end]是x的活跃区间，但x并非在[start, end]区间中处处活跃
- 采用更精确的活跃区间，可以有效提高寄存器分配的效果

伪线性序

- L1有两个后继基本块L2和L3

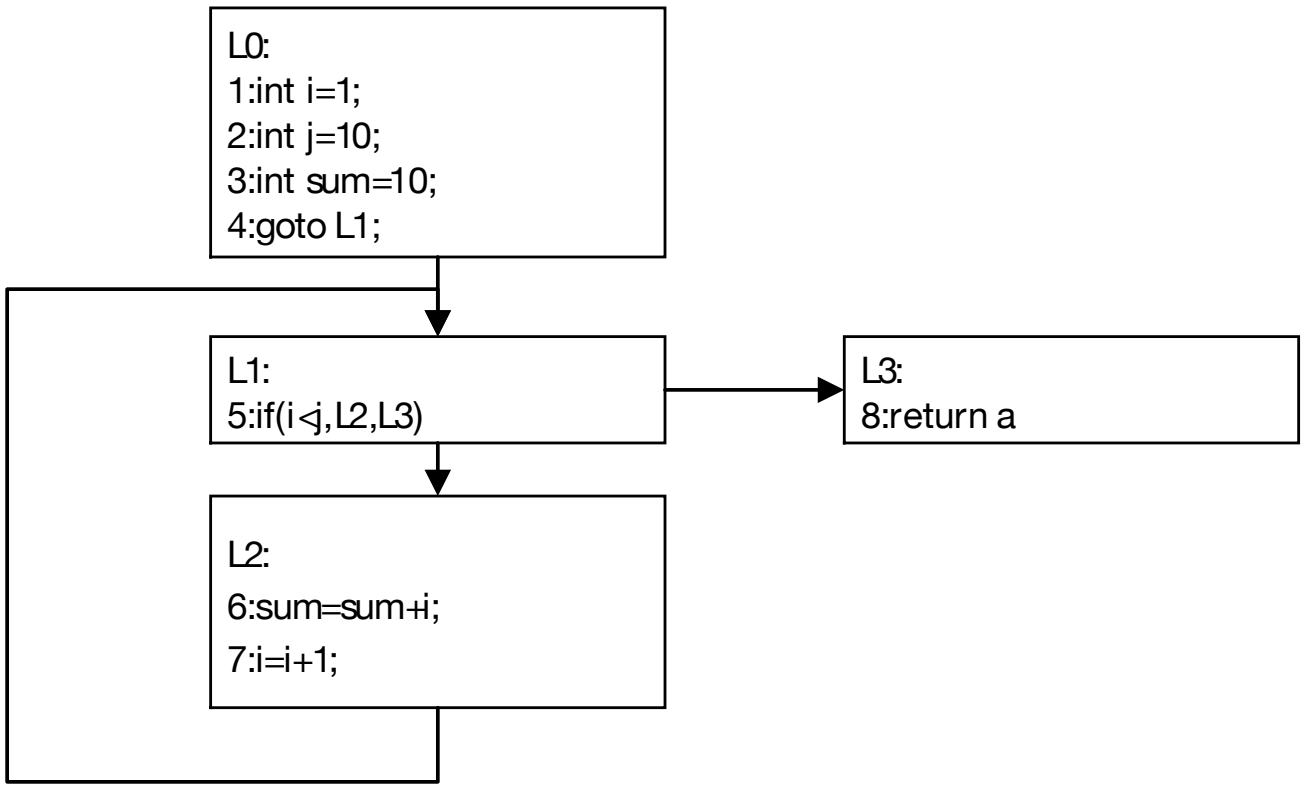
- 运行时存在L1→L2和L1→L3两种流向

- 只用一个线性序无法完整表示程序运行过程



伪线性序

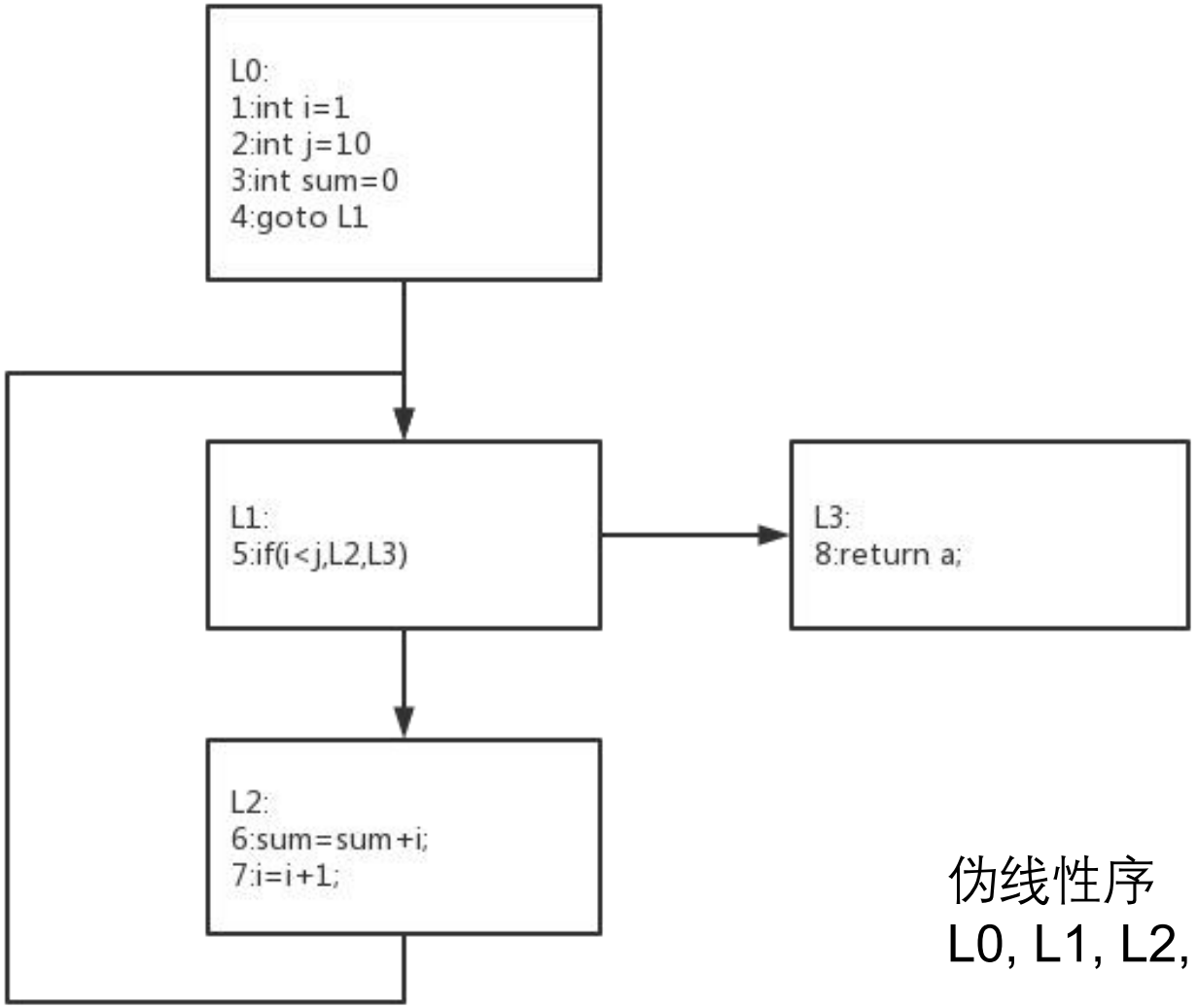
- **伪线性序算法**
  - 从Entry块按照执行顺序排布
  - 如当前块为循环头，如何排布？



L0->L1->?

伪线性序

❑计算活跃区间的前提条件—基本块BB的执行的线性序



伪线性序
L0, L1, L2, L3

思路：
1、从entry块开始向后扫描
2、循环，先扫描循环头，再优先排列循环体内的BB
3、其他BB，按执行顺序排列
4、若存在还未遍历的BB，回退，返回未遍历BB的入口，再进行排列

伪线性序

- 根据伪线性序，获得代码行号

- 活跃变量分析->每个变量初始的活跃范围
  - 以OUT[B]或IN[B]进行计算

活跃区间计算小技巧

- 活跃区间的计算将直接影响寄存器分配效果
- 为了得到更好的分配效果，将函数的区间大小设置为指令个数的2倍
- 比如函数有10条指令，函数的区间就是[0,20]
- 每条指令所在的行号是当前指令数量的2倍。比如第5条指令的对应的值是10。

活跃区间计算小技巧

- 将函数区间进行放大两倍处理有什么好处？
- 不采用两倍示例

```
0 store i32 4, i32* @a                    ;全局变量a赋值4
1 %op0 = load i32, i32* @a                ;将a值取出存入变量%op0
2 %op1 = icmp sgt i32 %op0, 0             ;将%op0与0比较，真则赋值为1，否则为0
3 %op2 = zext i1 %op1 to i32              ;将bool变量%op1转换为int变量%op2
4 %op3 = icmp ne i32 %op2, 0              ;将%op2与0比较，真则%op3赋值为1，否则为0
5 br i1 %op3, label %label4, label %label5
```

活跃区间

op0，i32，main：
{[1,2]}
op1，i32，main：
{[2,3]}
op2，.i32，main：
{[3,4]}
op3，i32，main：
{[4,5]}

	1	2	3	4	5
op0	■	■			
op1		■	■		
op2			■	■	
op3				■	■

需要2个寄存器
才能完成分配

活跃区间计算小技巧

- 将函数区间进行放大两倍处理有什么好处？
- 采用两倍示例

```
0       store i32 4, i32* @a                    ;全局变量a赋值4
2       %op0 = load i32, i32* @a                ;将a值取出存入变量%op0
4       %op1 = icmp sgt i32 %op0, 0             ;将%op0与0比较，真%op1赋值为1，否则为0
6       %op2 = zext i1 %op1 to i32              ;将bool变量%op1转换为int变量%op2
8       %op3 = icmp ne i32 %op2, 0              ;将%op2与0比较，真%op3赋值为1，否则为0
10      br i1 %op3, label %label4, label %label5
```

活跃区间

```
op0，i32，main：{[3,5)}//实际计算，活跃区间为{[3,4]}
op1，i32，main：{[5,7)}//实际计算，活跃区间为{[5,6]}
op2，i32，main：{[7,9)}//实际计算，活跃区间为{[7,8]}
op3，i32，main：{[9,11)}          //实际计算，活跃区
```

活跃区间
左端点：赋值语句下一行
右端点：使用语句当前行

让活跃区间更加精细化

活跃区间计算小技巧

- 将函数区间进行放大两倍处理有什么好处？
- 采用两倍示例

```
0       store i32 4, i32* @a                    ;全局变量a赋值4
2       %op0 = load i32, i32* @a                ;将a值取出存入变量%op0
4       %op1 = icmp sgt i32 %op0, 0             ;将%op0与0比较，真%op1赋值为1，否则为0
6       %op2 = zext i1 %op1 to i32              ;将bool变量%op1转换为int变量%op2
8       %op3 = icmp ne i32 %op2, 0              ;将%op2与0比较，真%op3赋值为1，否则为0
10      br i1 %op3, label %label4, label %label5
```

活跃区间

```
op0，i32，main：{[3,5)}//实际计算，活跃区间为{[3,4]}
op1，i32，main：{[5,7)}//实际计算，活跃区间为{[5,6]}
op2，i32，main：{[7,9)}//实际计算，活跃区间为{[7,8]}
op3，i32，main：{[9,11)}          //实际计算，活跃区
```

	0	1	2	3	4	5	6	7	8	9	10	11
op0				■	■							
op1						■	■					
op2								■	■			
op3										■	■	

需要1个寄存器就能完成分配

活跃区间计算小技巧

- 活跃区间的左端点总是赋值点的行号若不加1

```
0 %op0 = alloca i32;
2 %op1 = alloca i32;
4 %op2 = alloca i32;
6 store i32 1, i32* %op0
8 store i32 2, i32* %op1
10 %op3 = load i32, i32* %op0
12 %op4 = load i32, i32* %op1;
14 %op5= add i32, %op3, %op4
16 store i32 %op5, %op2
```

活跃区间

```
%op3, i32, main{[10,15)},
%op4, i32, main{[12,15)}
%op5, i32, main{[14,17]}
```

寄存器分配方案

```
R0 <— %op3
R1 <— %op4
R2 <— %op5
```

需要3个寄存器才能完成分配

活跃区间计算小技巧

- 活跃区间的左端点总是赋值点的行号加1

```
0 %op0 = alloca i32;
2 %op1 = alloca i32;
4 %op2 = alloca i32;
6 store i32 1, i32* %op0
8 store i32 2, i32* %op1
10 %op3 = load i32, i32* %op0
12 %op4 = load i32, i32* %op1;
14 %op5= add i32, %op3,%op4
16 store i32 %op5, %op2
```

活跃区间

```
%op3, i32, main{[11,15)},
%op4, i32, main{[13,15)}
%op5, i32, main{[15,17]}
```

寄存器分配方案

```
R0 <— %op3
R1 <— %op4
R0 <— %op5
```

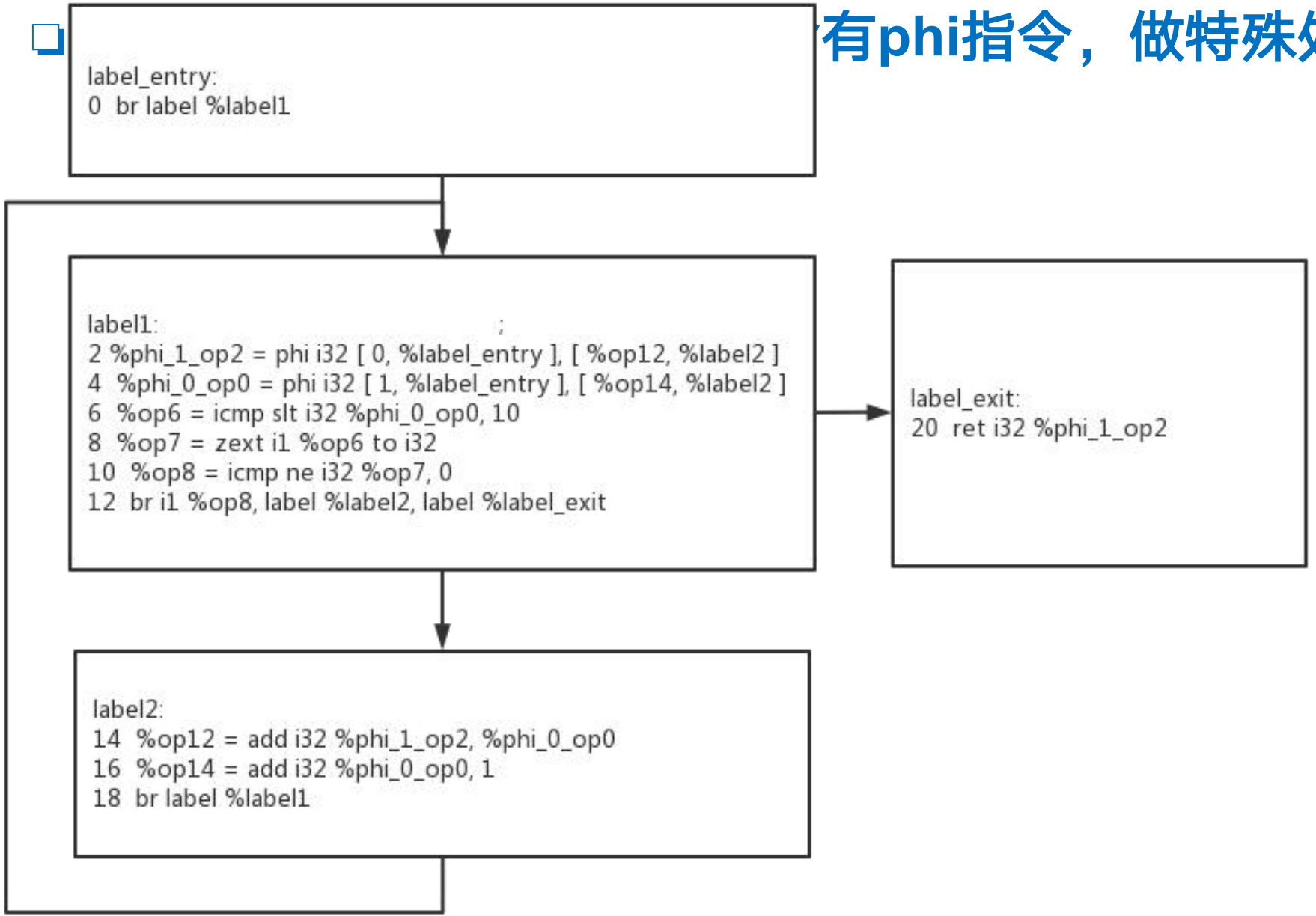
需要2个寄存器就能完成分配

活跃区间计算小技巧

- 注意：mem2reg后的IR，若含有phi函数，做特殊处理
  - Phi函数定义过程拆解到对应前驱基本块最后

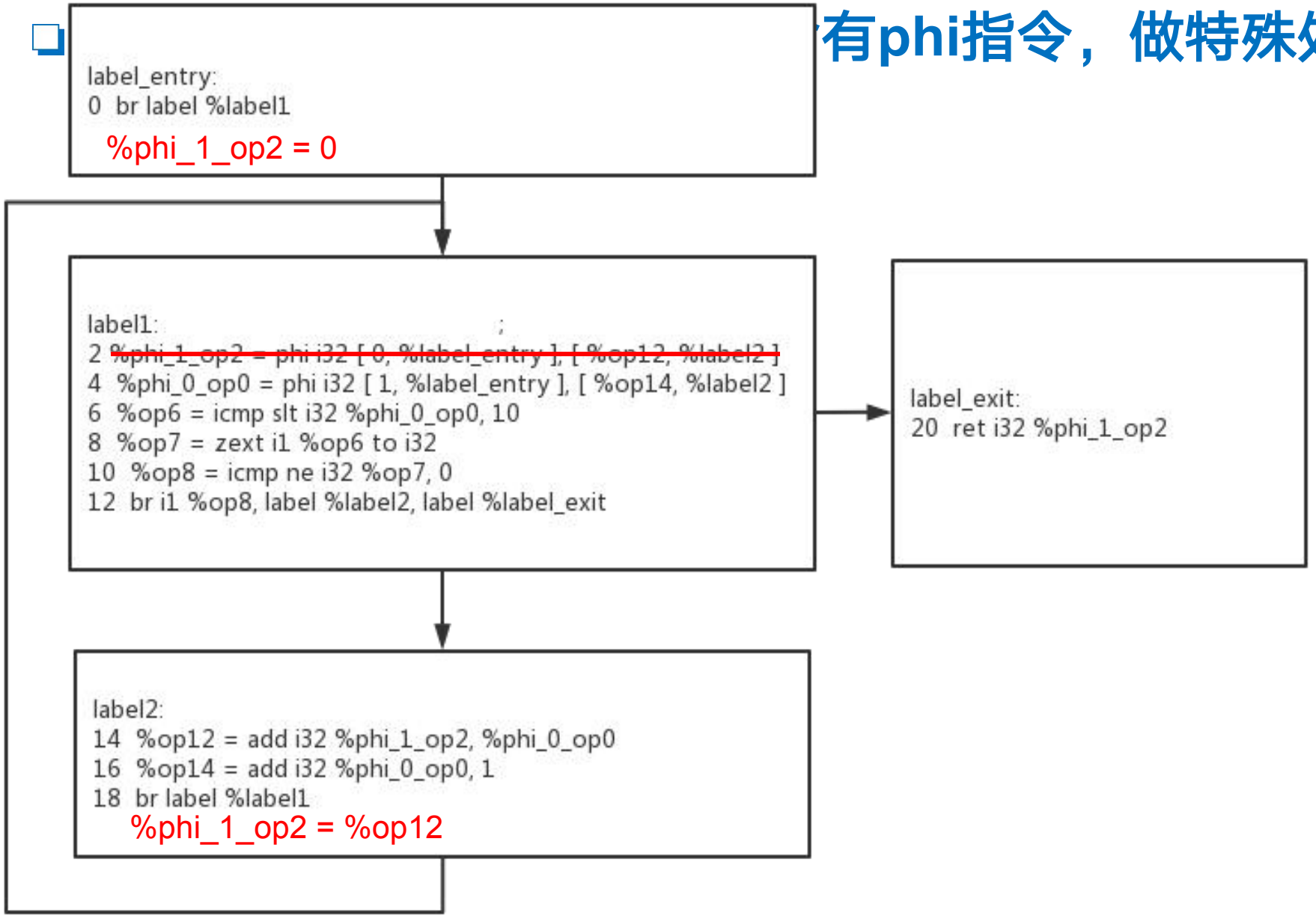
活跃区间计算小技巧

- 有phi指令，做特殊处理



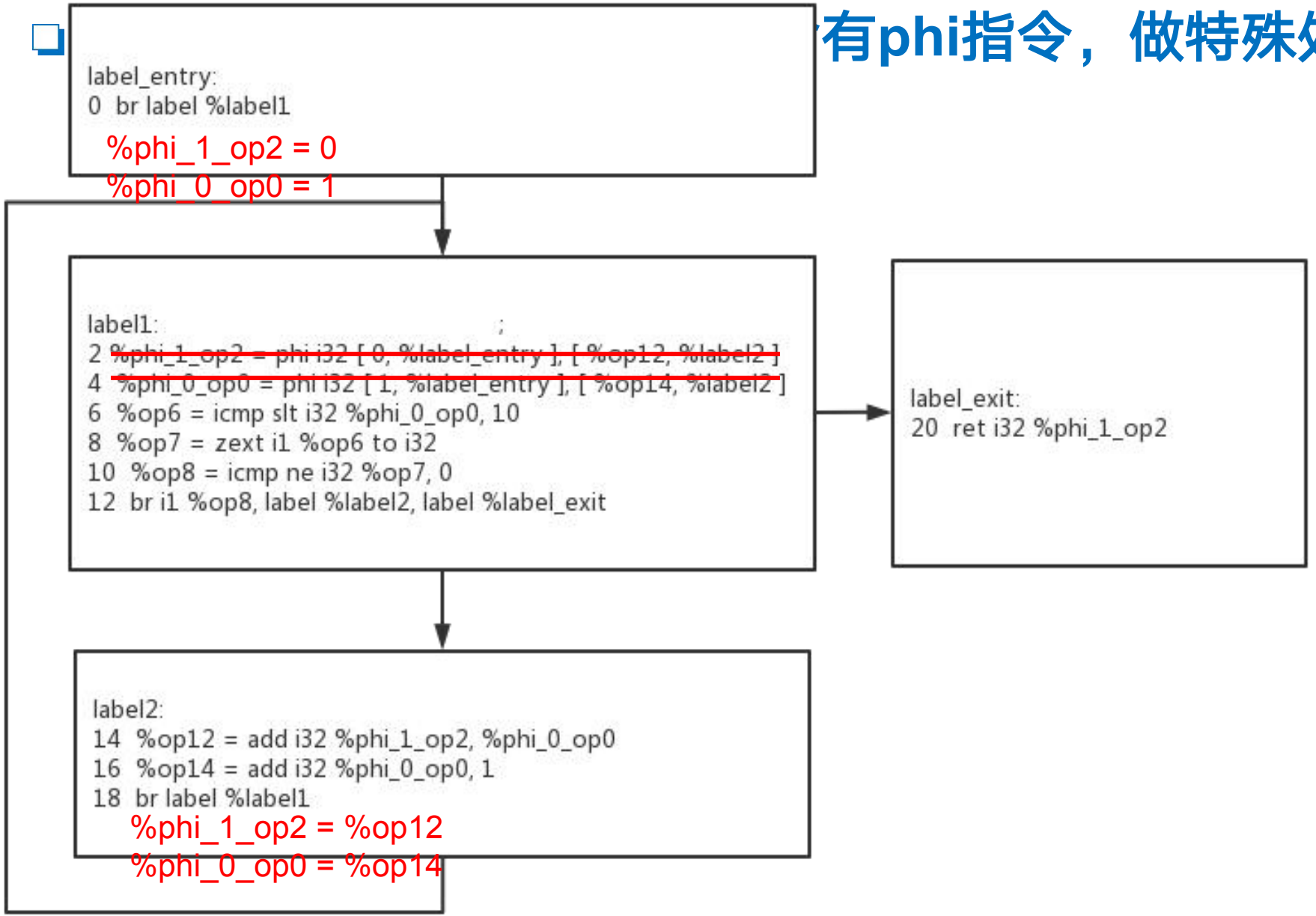
活跃区间计算小技巧

- 有phi指令，做特殊处理



活跃区间计算小技巧

- 有phi指令，做特殊处理



活跃区间计算小技巧

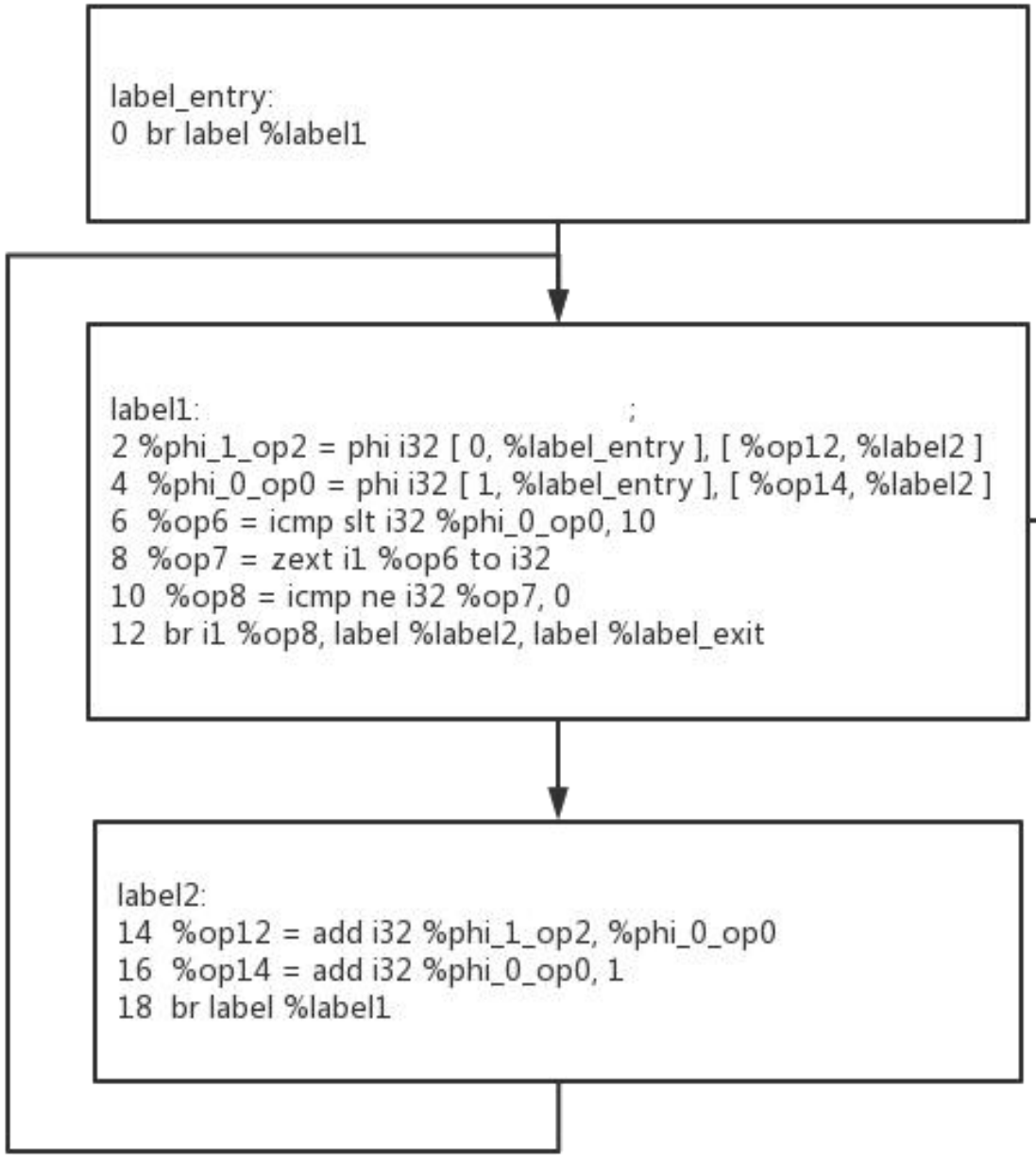
- 注意：mem2reg后的IR，若含有phi函数，做特殊处理
  - Phi函数定义过程拆解到对应前驱基本块最后

- 后续介绍更一般性的活跃区间计算方法

活跃区间算法

- 初始化变量活跃区间
- 伪线性序倒序的顺序扫描基本块，基本块内部倒序方式扫描指令
- 一般变量的活跃区间计算
- Phi函数的活跃区间计算

活跃区间算法

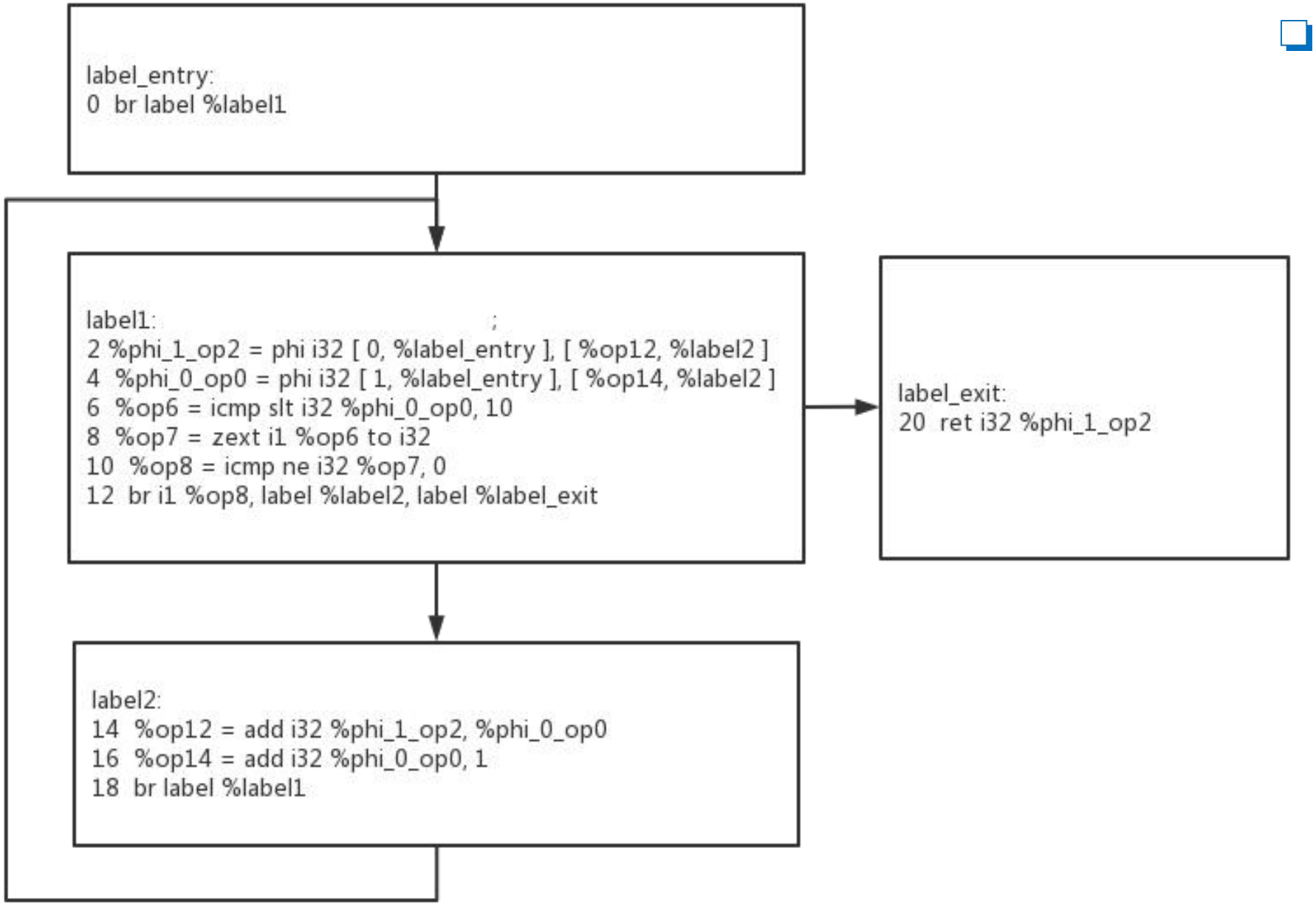


❑初始化变量活跃区间

- BB出口活跃变量的活跃区间初始为BB的开始到BB结束
- 同一变量的活跃区间合并

基本块	label_exit	Label1	Label2	label_entry
OUT[]	ø	%phi_0_op0, %phi_1_op2	%op12、%op14	ø

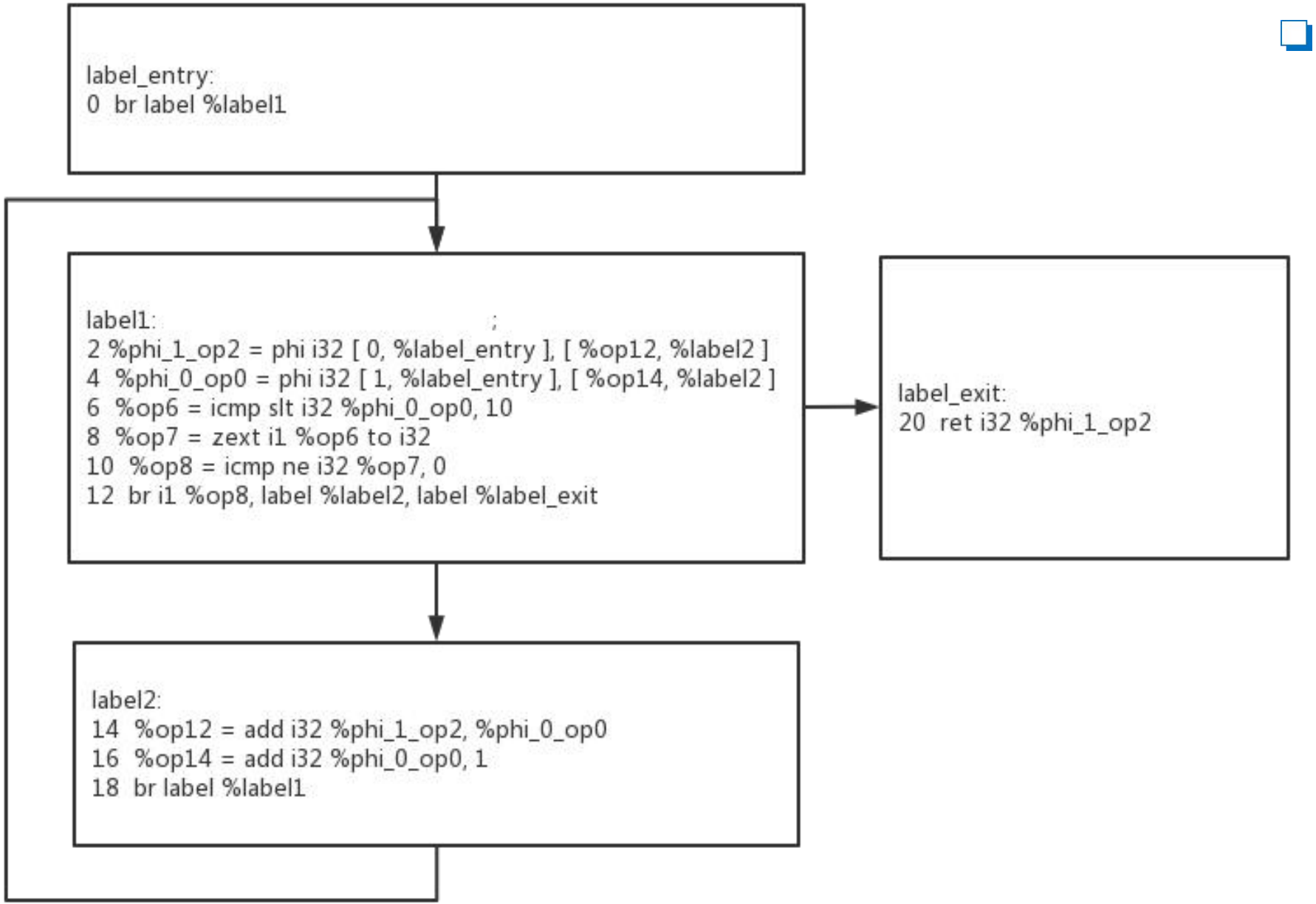
活跃区间算法



❑一般变量的活跃区间

- **指令包括变量的定义**
  - 左端点为当前指令行号+1
- **指令包括变量的使用**
  - 右端点为最后一次使用的指令行号

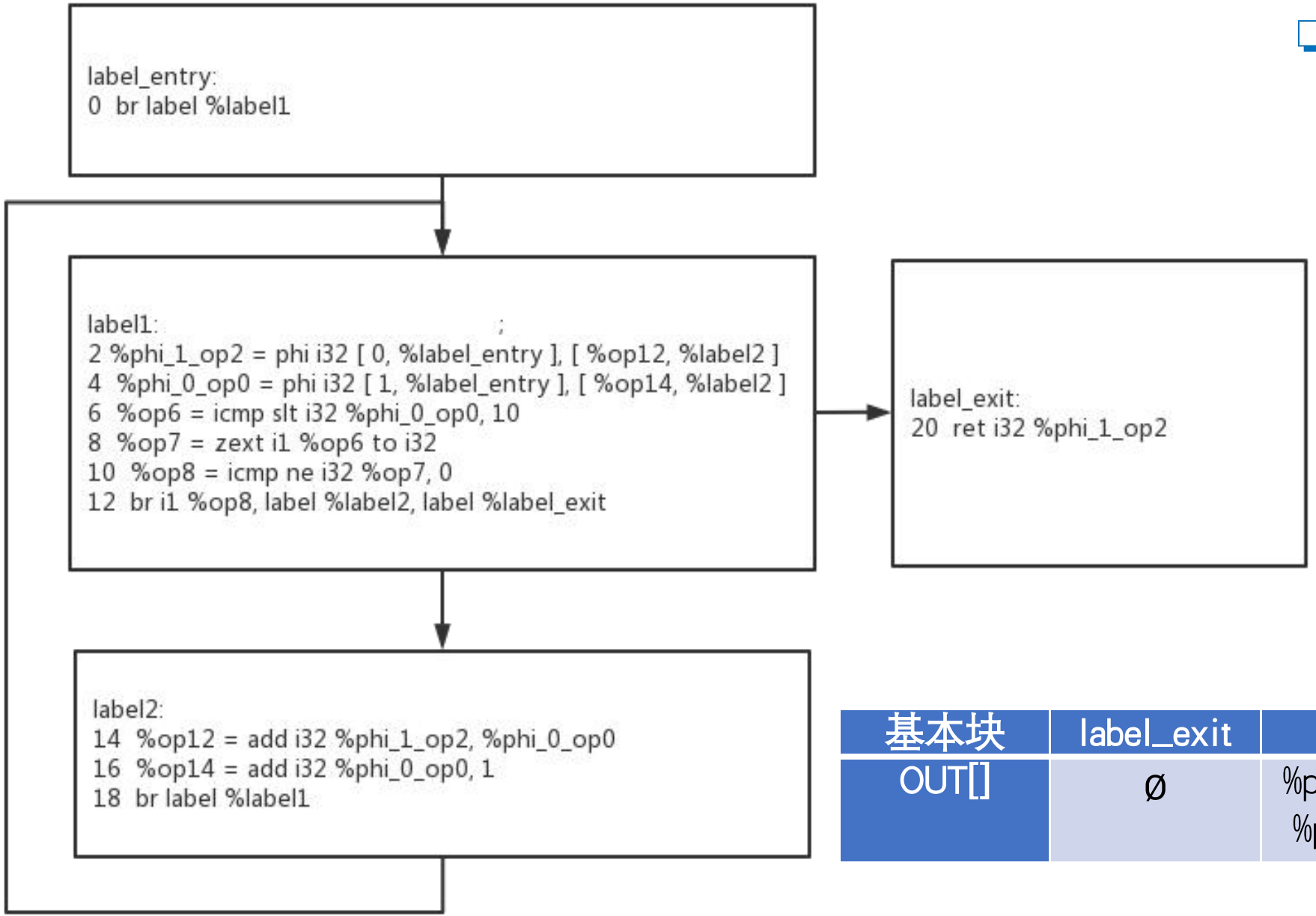
活跃区间算法



❑Phi函数的活跃区间

- **指令包括phi函数定义**
  - 左端点为当前BB开始行号
- **指令包括phi函数使用**
  - 右端点为最后一次使用的指令行号

活跃区间算法



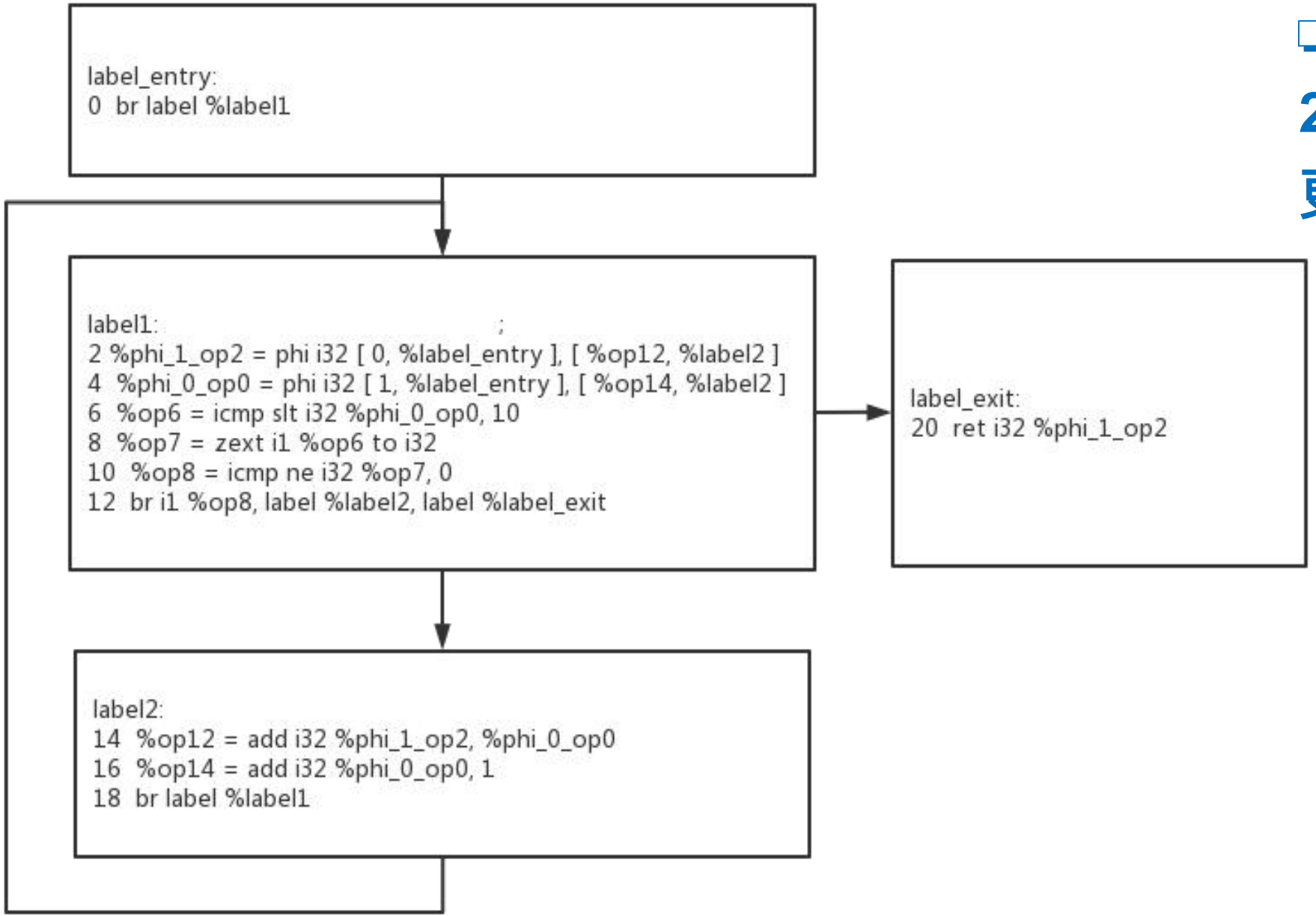
❑初始化

- ■ BB出口活跃变量的活跃区间初始为BB的开始到BB结束

	start	end
op6		
op7		
op8		
op12	14	19
op14	14	19
phi_0_op0	2	13
phi_1_op2	2	13

基本块	label_exit	Label1	Label2	label_entry
OUT[]	ø	%phi_0_op0，%phi_1_op2	%op12、%op14	ø

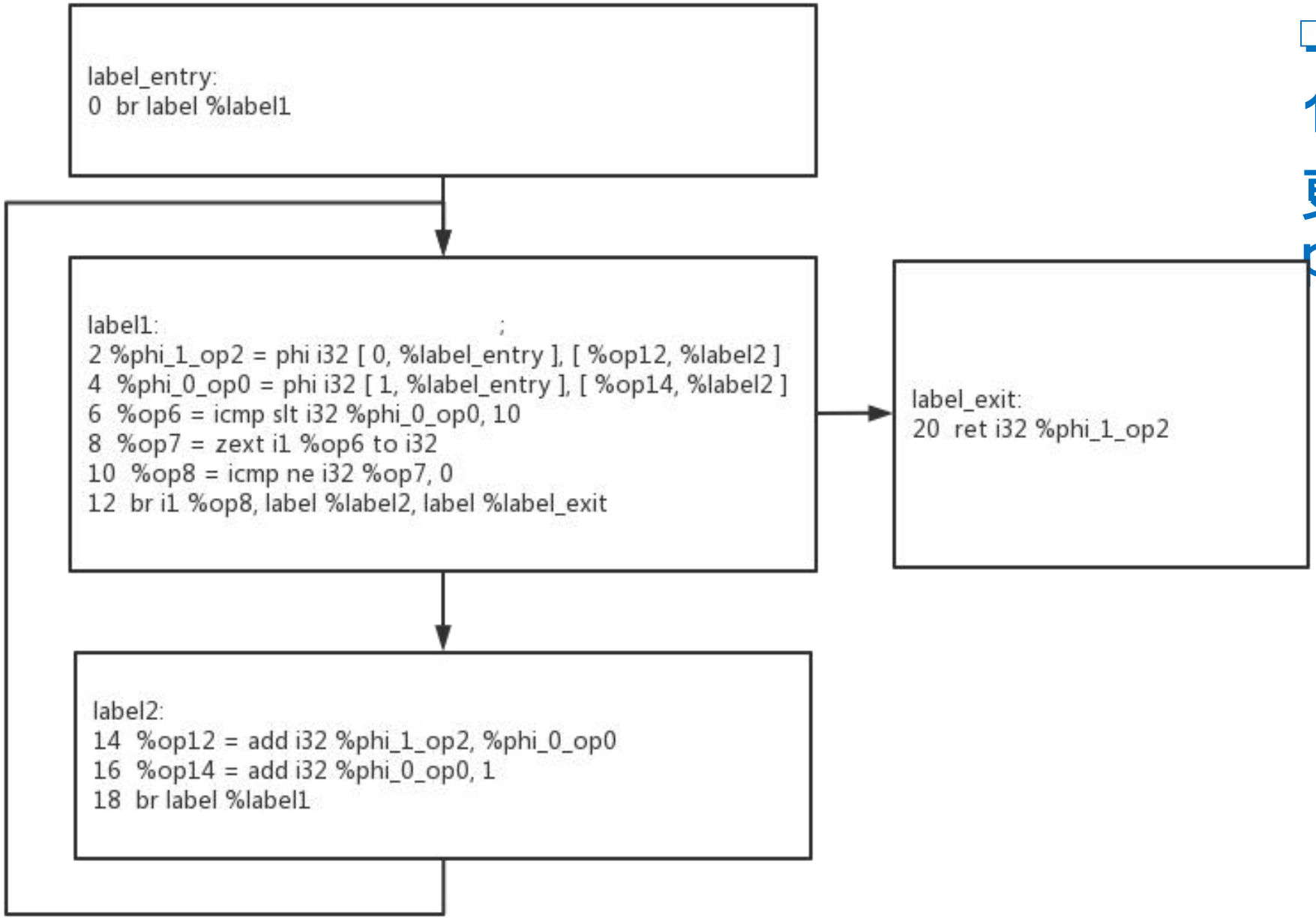
活跃区间算法



- 逆序扫描
- 20行
- 更新phi_1_op2右端点

	start	end
op6		
op7		
op8		
op12	14	19
op14	14	19
phi_0_op0	2	13
phi_1_op2	2	20

活跃区间算法



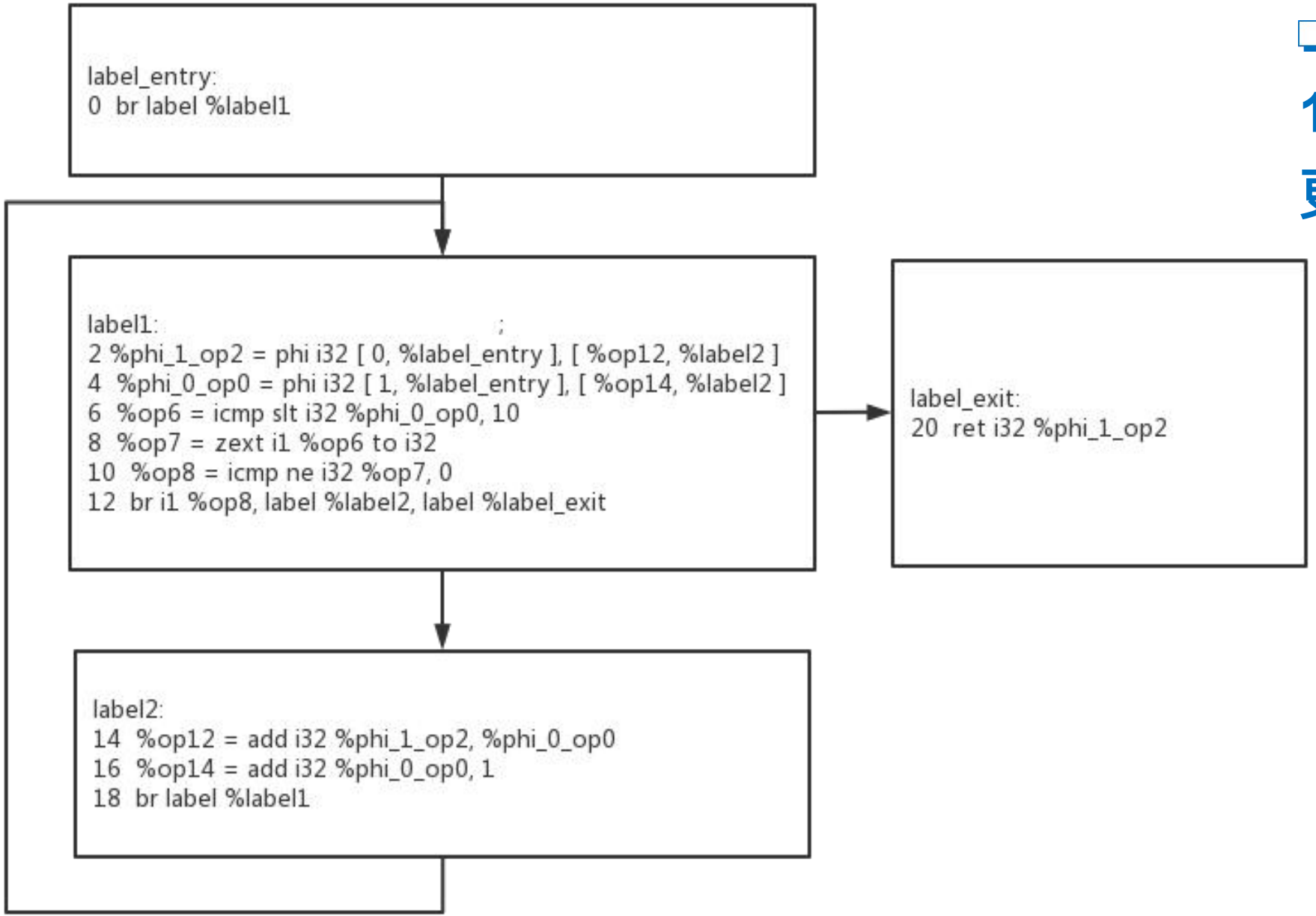
- 逆序扫描

16行、14行

更新op14、op12、phi_0_op0

	start	end
op6		
op7		
op8		
op12	15	19
op14	17	19
phi_0_op0	2	17
phi_1_op2	2	20

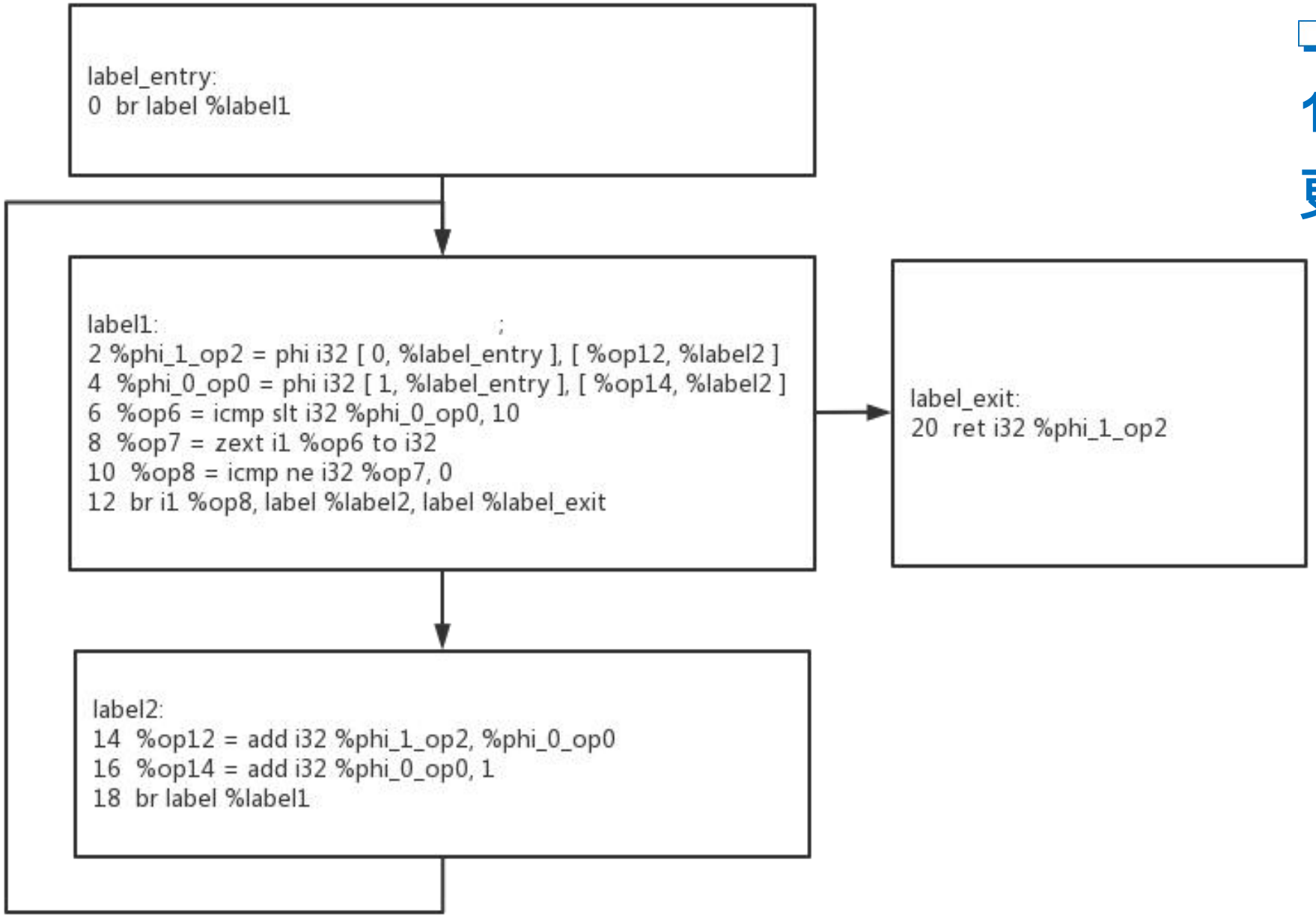
活跃区间算法



- 逆序扫描
- 12行
- 更新op8

	start	end
op6		
op7		
op8		12
op12	15	19
op14	17	19
phi_0_op0	2	17
phi_1_op2	2	20

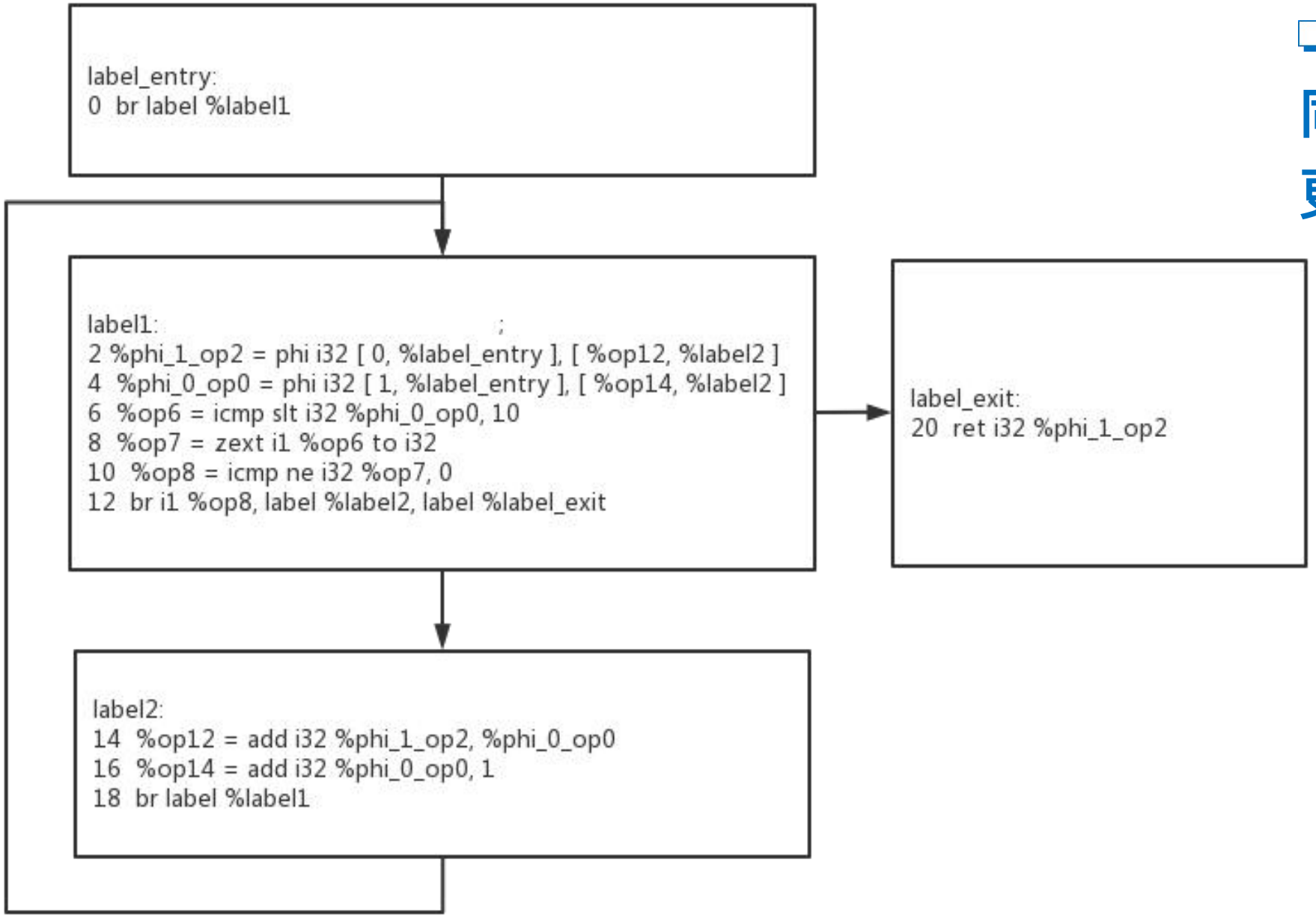
活跃区间算法



- 逆序扫描
- 10行
- 更新op8

	start	end
op6		
op7		
op8	11	12
op12	15	19
op14	17	19
phi_0_op0	2	17
phi_1_op2	2	20

活跃区间算法



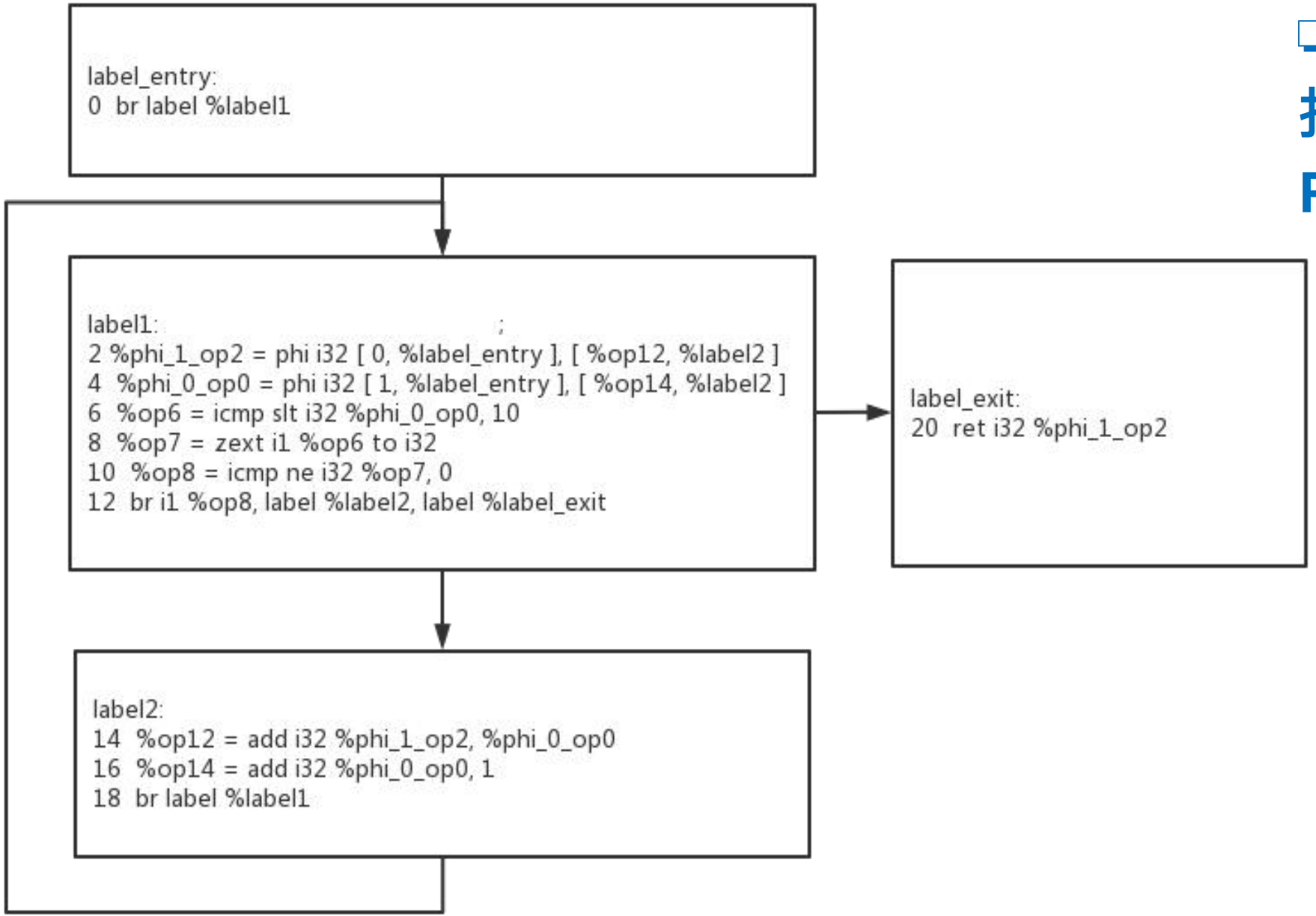
- 逆序扫描

同理扫描8行、6行

更新op7、op6

	start	end
op6	7	8
op7	9	10
op8	11	12
op12	15	19
op14	17	19
phi_0_op0	2	17
phi_1_op2	2	20

活跃区间算法



- 逆序扫描
- 扫描4行、2行
- Phi函数区间无变化

	start	end
op6	7	8
op7	9	10
op8	11	12
op12	15	19
op14	17	19
phi_0_op0	2	17
phi_1_op2	2	20

一起努力
打造国产基础软硬件体系！

徐 伟

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

计算机科学与技术学院

2025年5月8日