并行编译与优化
Parallel Compiler and Optimization

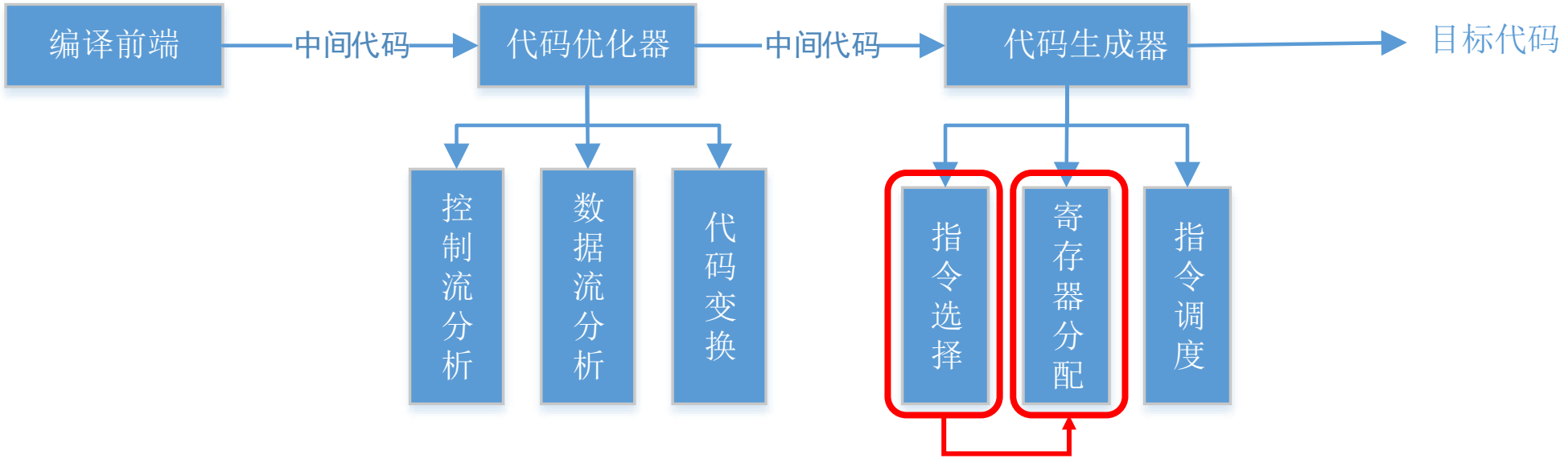
计算机研究所编译系统室

Lecture 10: Register Allocation Part1

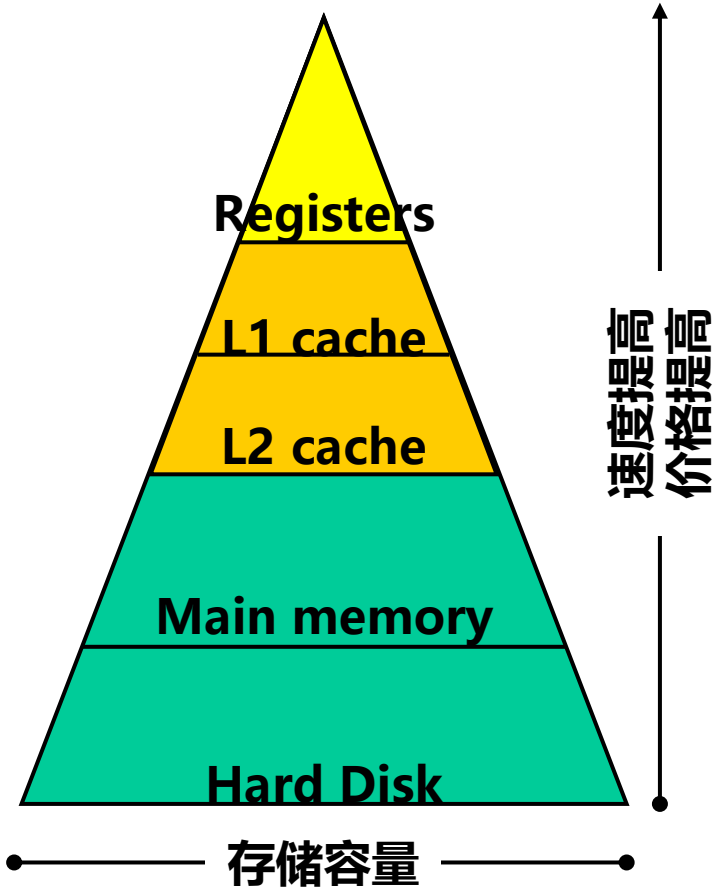
第十课：寄存器分配（一）

回顾

- 编译中端：分析+变换
- 编译后端：指令选择
  - 目标：将IR翻译成等价的、成本(如执行时间)最小的目标指令序列
- 该指令序列是最终的目标指令序列吗？



从存储层次看寄存器分配



- 现代计算机体系结构采用多级存储层次
- 寄存器位于最顶层，是计算机大部分指令的操作对象
- 程序中的数据必须从**存储器**传送到**寄存器**参与运算
- **编译器**决定数据在存储器和寄存器之间的移动

从存储层次看寄存器分配

	访问延迟	容量
寄存器	< 1 cycle	256~8000B
缓存	~3 cycles	256KB~64MB
存储器	20~100 cycles	512MB~64GB

- 寄存器访问速度最快，容量最小
- 寄存器分配
  - 目标：高效、合理地使用有限的寄存器资源
  - 编译器中最重要的优化之一：实验表明采用较优寄存器分配算法生成的代码比将所有变量都保存于存储器中的代码快2.5倍

学习内容

COMP
CCRG Open MP

10.1 寄存器分配概述

10.2 基于使用计数的寄存器分配方法

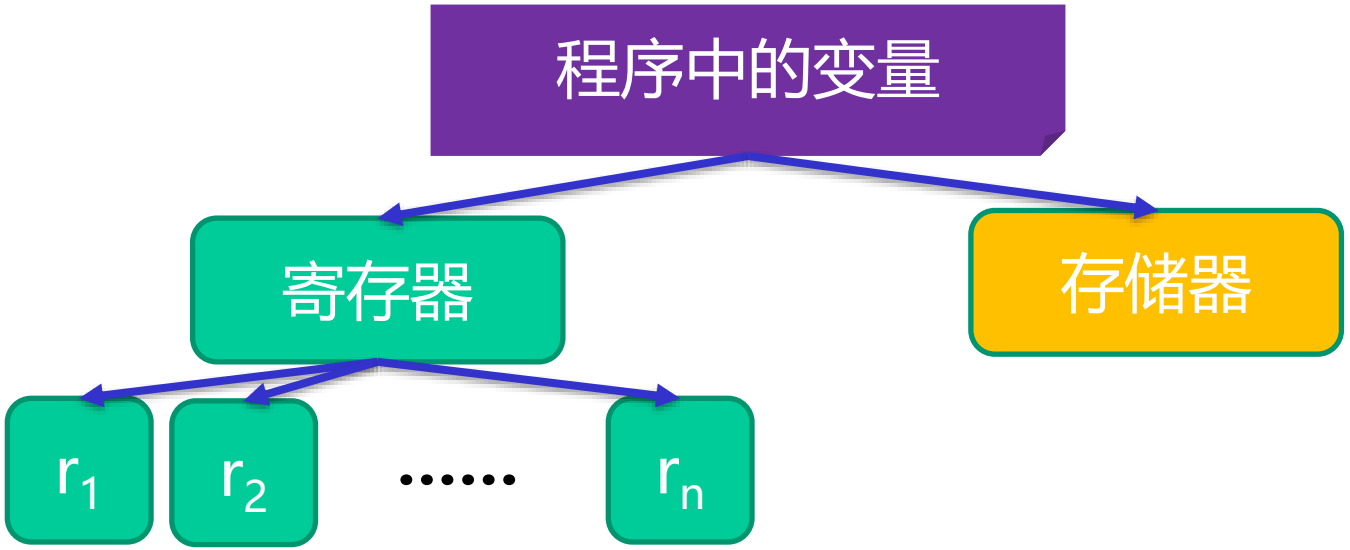
10.3 基于图着色的寄存器分配方法

学习目标

- 掌握基于使用计数的寄存器分配方法和基于图着色的寄存器分配方法
- 理解寄存器分配的基本概念和基本原则

10.1.1 寄存器分配的基本概念

- 将程序中的变量分配到寄存器或者存储器的过程
- 寄存器分配确定
  - 哪些变量保存在寄存器中，哪些变量保存在存储器中（寄存器分配）
  - 对于保存在寄存器中的变量：具体放在哪个寄存器（寄存器指派）



10.1.2 寄存器分配的基本原则

- 程序变量数目 vs. 寄存器数目
  - 程序中有大量的变量，变量数一般大于寄存器数
- 寄存器分配原则一: 尽可能将更多的变量保存在寄存器中
  - 不能让一个变量占用寄存器的时间比实际需要的时间长

10.1.2 寄存器分配的基本原则

- 溢出(spill)
  - 将一个变量存放到存储器而不是寄存器
  - 插入溢出代码(spill code)
    - 每次定值该变量后，需要插入一条写指令，将变量的值写入存储器中
    - 每次使用该变量前，需要再为该变量分配寄存器，并且通过读指令将值从存储器读入到指派的寄存器中
- 寄存器分配原则二: 尽可能将频繁使用的变量保存在寄存器中
  - 将那些使用较少的变量存放在存储器中
  - 尽可能减少溢出和读写指令次数

10.1.3 寄存器分配的主要方法

- 基于使用计数的寄存器分配方法
  - 20世纪80年代之前使用
- 基于图着色的寄存器分配方法
  - 突破性进展，20世纪80年代开始流行，如GCC

- 基于线性扫描的寄存器分配方法
  - 1999年提出，如LLVM

- 基于SSA的寄存器分配方法
  - 2005年提出

学习内容

COMP
CCRG Open MP

10.1 寄存器分配概述

10.2 基于使用计数的寄存器分配方法

10.3 基于图着色的寄存器分配方法

10.2.1 基于使用计数的局部分配方法

- **使用计数**: 变量的使用的次数
- 基于使用计数的局部分配方法
  - 以基本块为单位，根据需要依次分配寄存器(需要一个则分配一个)
  - 遇到变量的一次使用，使用计数减1
  - 当使用计数为0时，寄存器便可再次分配给其它变量使用
  - 当寄存器不够时，根据启发式信息溢出一个变量
    - 溢出使用计数最小的
    - 溢出已经有副本在存储器中

10.2.1 基于使用计数的局部分配方法

示例

- 如下代码共有7个变量，假设有3个可供分配的寄存器: r1, r2, r3,
- t1, t2, t3是临时变量
- a, b, c, d是程序变量，在存储器中有副本，且在基本块出口处活跃

$t1 \leftarrow a - b$

$t2 \leftarrow a * c$

$t3 \leftarrow t1 + t2$

$a \leftarrow d + c$

$d \leftarrow t2 + t3$

a, 2
b, 1
c, 2
d, 1
t1, 1
t2, 2
t3, 1

$t1^{r2} \leftarrow a^{r1} - b^{r2}$

$t2^{r1} \leftarrow a^{r1} * c^{r3}$

$t3^{r2} \leftarrow t1^{r2} + t2^{r1}$

$a \leftarrow d^{?} + c^{r3}$

没有可用寄存器！
为了将变量d存放在寄存器中，应该溢出哪个变量?

10.2.1 基于使用计数的局部分配方法

示例

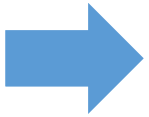
$t1^{r2} \leftarrow a^{r1} - b^{r2}$

~~$t2^{r1}$~~ $\leftarrow a^{r1} * c^{r3}$

$t3^{r2} \leftarrow t1^{r2} + t2^{r1}$

$a^{r1} \leftarrow d^{r1} + c^{r3}$

$d^{r2} \leftarrow t2^{r3} + t3^{r2}$



```
ld      r1  a
ld      r2  b
sub     r2  r1, r2
ld      r3  c
mult    r1  r1, r3
st      t2  r1
ld      r1  t2
add     r2  r2, r1
ld      r1  d
add     r1  r1, r3
ld      r3  t2
add     r2  r3, r2
st      a   r1
st      d   r2
```

溢出代码

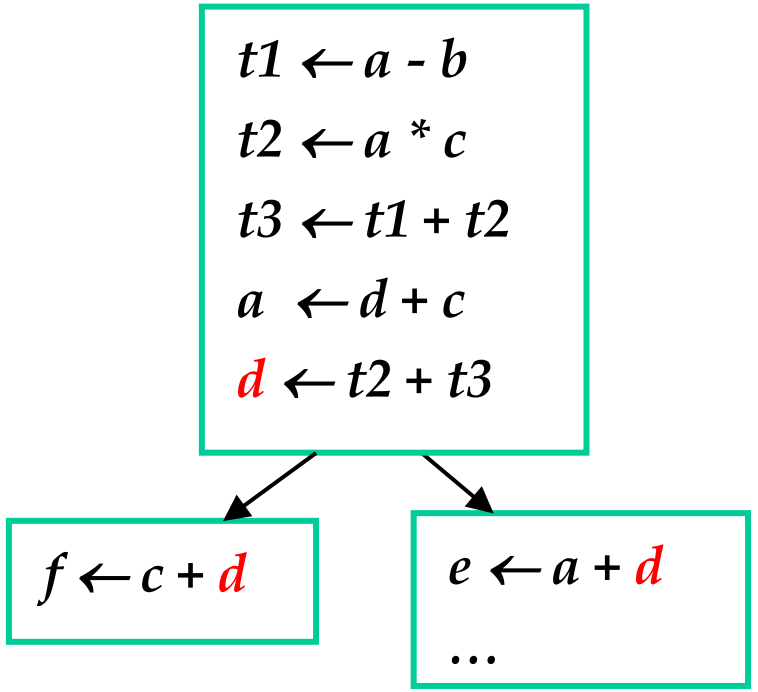
t2和t3的剩余计数都为1，都无存储器副本，下一个使用点也都相同，因此任意选择溢出t2

a, b, c, d在基本块出口处活跃

a和d的值改变，b和c的值未改变

10.2.1 基于使用计数的局部分配方法

- 优点
  - 简单，易于实现
- 缺点
  - 使用计数只考虑单个基本块
  - 不能真正反应变量的频繁使用情况
    - 循环内使用的变量 vs 循环外使用的变量
  - 局限在基本块内，导致生成冗余的读写指令



10.2.2 基于使用计数的全局分配方法

- 跨越基本块边界，全局性评估变量分配寄存器所得到的收益
- 不仅考虑使用计数，也考虑变量在基本块边界的活跃情况
- 优先分配循环中的变量，给循环中的变量加以适当权重

10.2.2 基于使用计数的全局分配方法

- 基本块B中的变量v因分配寄存器获得的收益

$$benefit(v, B) = use(v, B) + 2 \cdot live(v, B)$$

$use(v, B)$: 在基本块B中，定值v之前对v的使用计数

$live(v, B)$: 如果v在基本块B的出口处活跃，且在B中被定值，则$live(v, B)$为1，否则为0

- 循环L中的变量v因分配寄存器获得的收益是循环内所有基本块的收益之和

$$benefit(v, L) = \sum_{B \in L} (use(v, B) + 2 \times liveout(v, B))$$

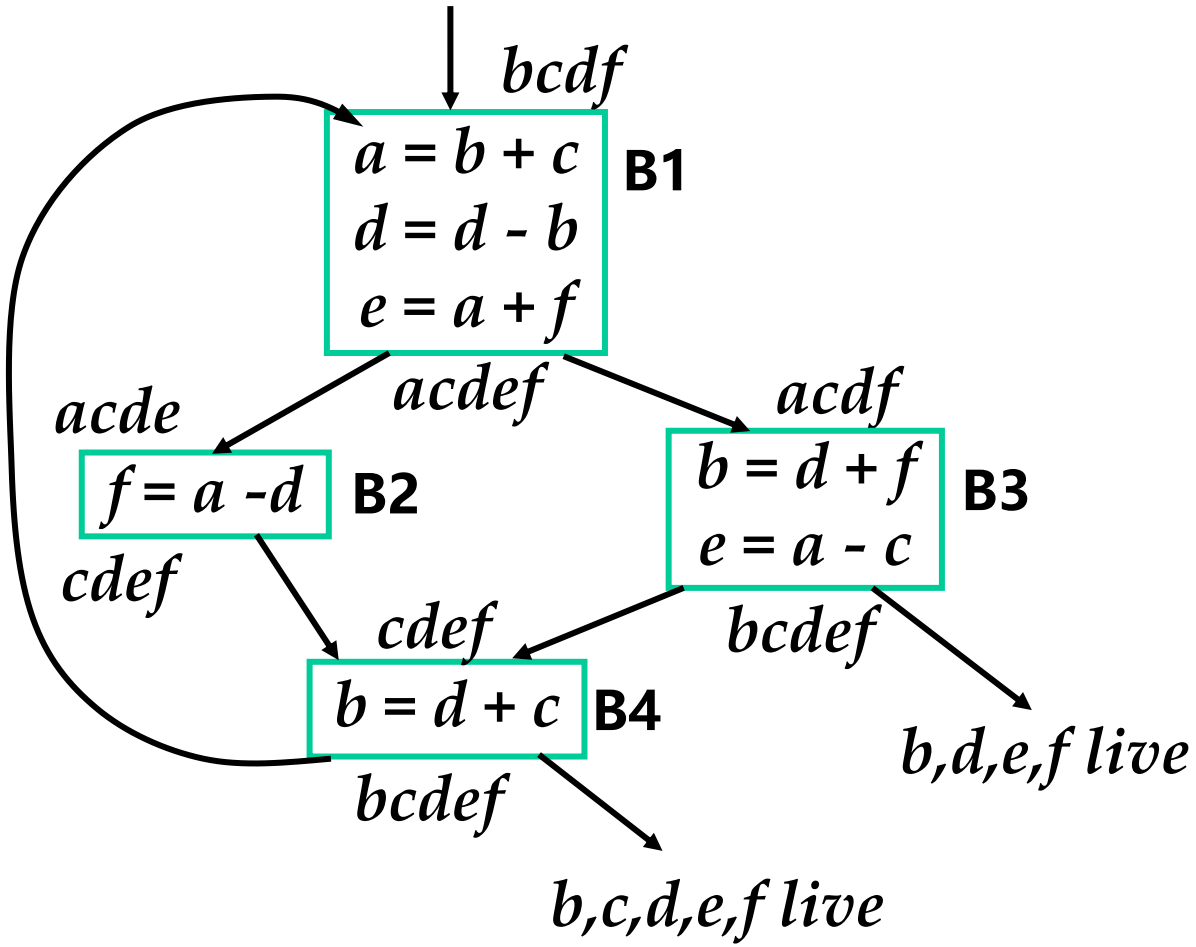
10.2.2 基于使用计数的全局分配方法

- 基于使用计数的全局分配方法
  - 使用R个寄存器用于循环中的变量，留若干寄存器用于其余变量
  - 从最内层循环开始，由内向外给前R个收益值最高的变量分配寄存器
  - 对于循环外的变量，根据使用计数和活跃情况来计算收益，根据收益进行寄存器分配和溢出

10.2.2 基于使用计数的全局分配方法

示例

$$benefit(v,L) = \sum_{B \in L} (use(v,B) + 2 \times liveout(v,B))$$



variable	a	b	c	d	e	f
benefit	4					

use(a,B1)=0, use(a,B2)=1
use(a,B3)=1, use(a,B4)=0

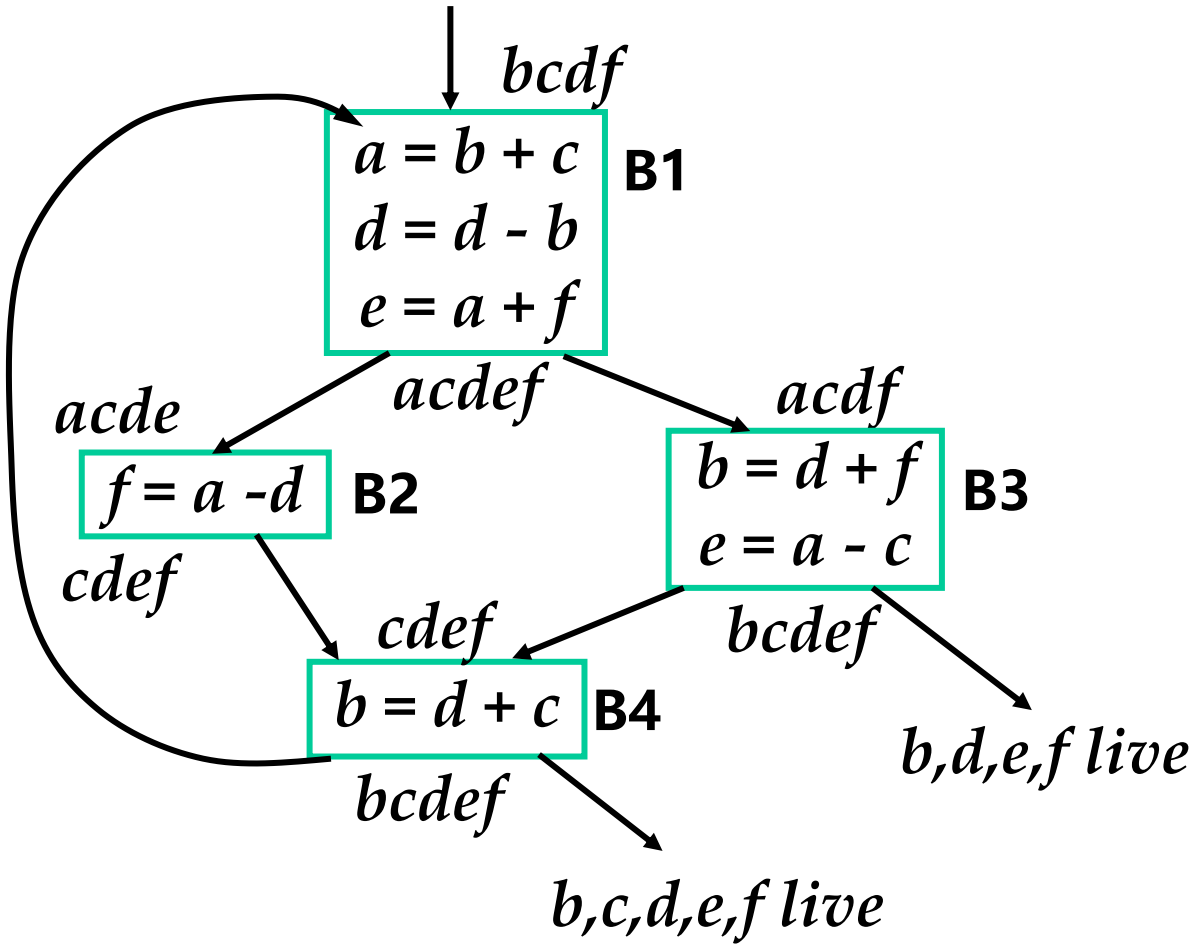
live(a,B1)=1, live(a,B2)=0
live(a,B3)=0, live(a,B4)=0

benefit(a,L) = 2+2*1 = 4

10.2.2 基于使用计数的全局分配方法

示例

$$benefit(v, L) = \sum_{B \in L} (use(v, B) + 2 \times liveout(v, B))$$



variable	a	b	c	d	e	f
benefit	4	6	3	6	4	4

假设有4个可供分配的寄存器r0, r1, r2, r3, 其中前3个用于循环:

- r0: a
- r1: b
- r2: d

10.2.2 基于使用计数的全局分配方法

- 优点
  - 实现简单，且有实际有效
- 缺点
  - 只全局分配部分寄存器
  - “全局”只是跨越循环内的基本块边界，而不是所有基本块
- 是图着色寄存器分配方法使用前应用最普遍的方法

银河-I巨型机

学习内容

10.1 寄存器分配概述

10.2 基于使用计数的寄存器分配方法

10.3 基于图着色的寄存器分配方法

图着色寄存器分配

- 全局分配算法，实际效果好
- 全局寄存器分配 ➡ 图着色问题
- 1971年IBM的John Cocke提出
  - 色数大于三时，图着色问题是NP-完全问题
  - 受限于当时的计算机速度和内存容量，十年来没有实质性进展

John Cocke
1987

JOHN COCKE DL 1987年图灵奖得主

United States – 1987

CITATION

For significant contributions in the design and theory of compilers, the architecture of large systems and the development of reduced instruction set computers (RISC); for discovering and systematizing many fundamental transformations now used in optimizing compilers including reduction of operator strength, elimination of common subexpressions, register allocation, constant propagation, and dead code elimination.

图着色寄存器分配

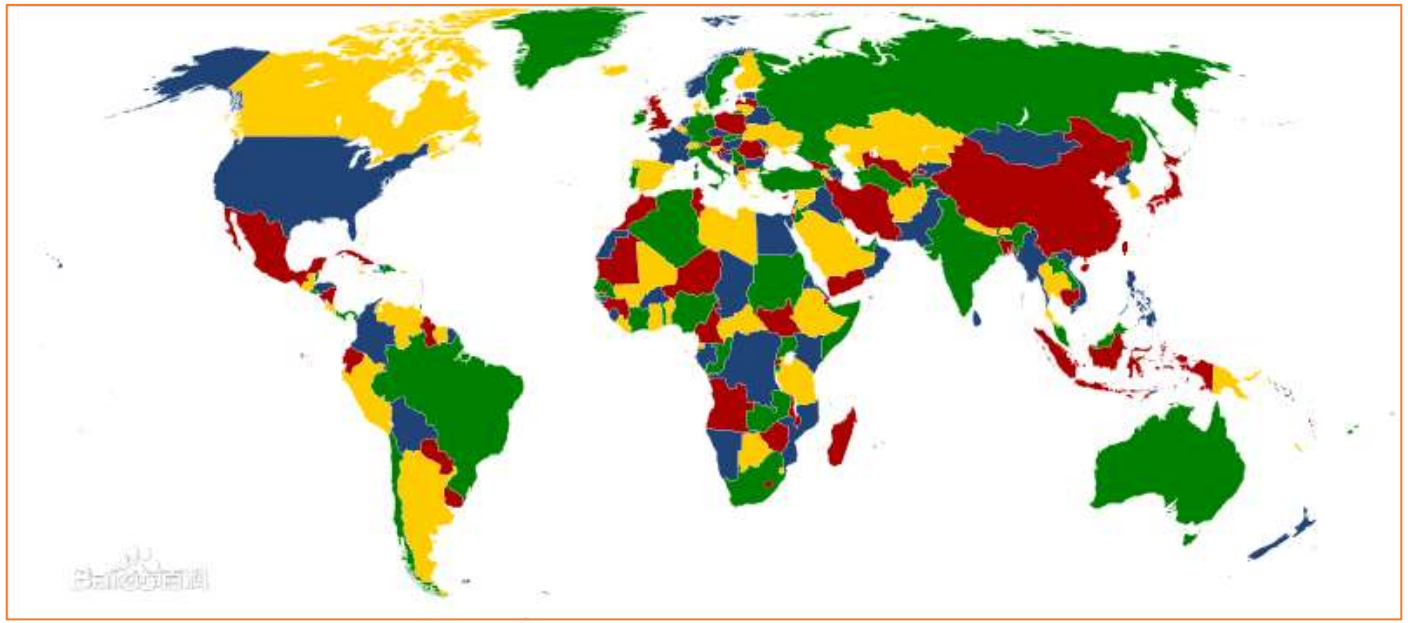
- 突破: 1981年IBM的Chaitin在IBM 370的PL/1编译器首次实现基于图着色的寄存器分配算法
- 随后开始流行，衍生出基于Chaitin算法的多种改进算法
  - 1994年Briggs的改进算法
  - 1996年George的改进算法

敢于求变，勇做破局者

敢于创新，争做开创者

10.3.1 图着色与寄存器分配

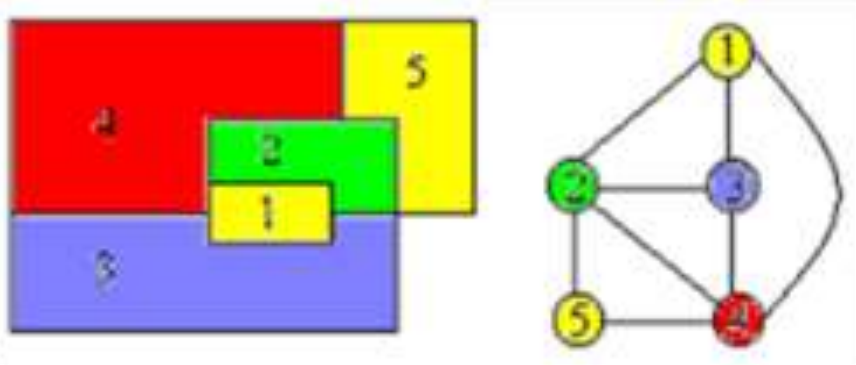
- 什么是四色定理?



- 用不同的颜色（4种）给地图着色，相邻国家不能着同一颜色

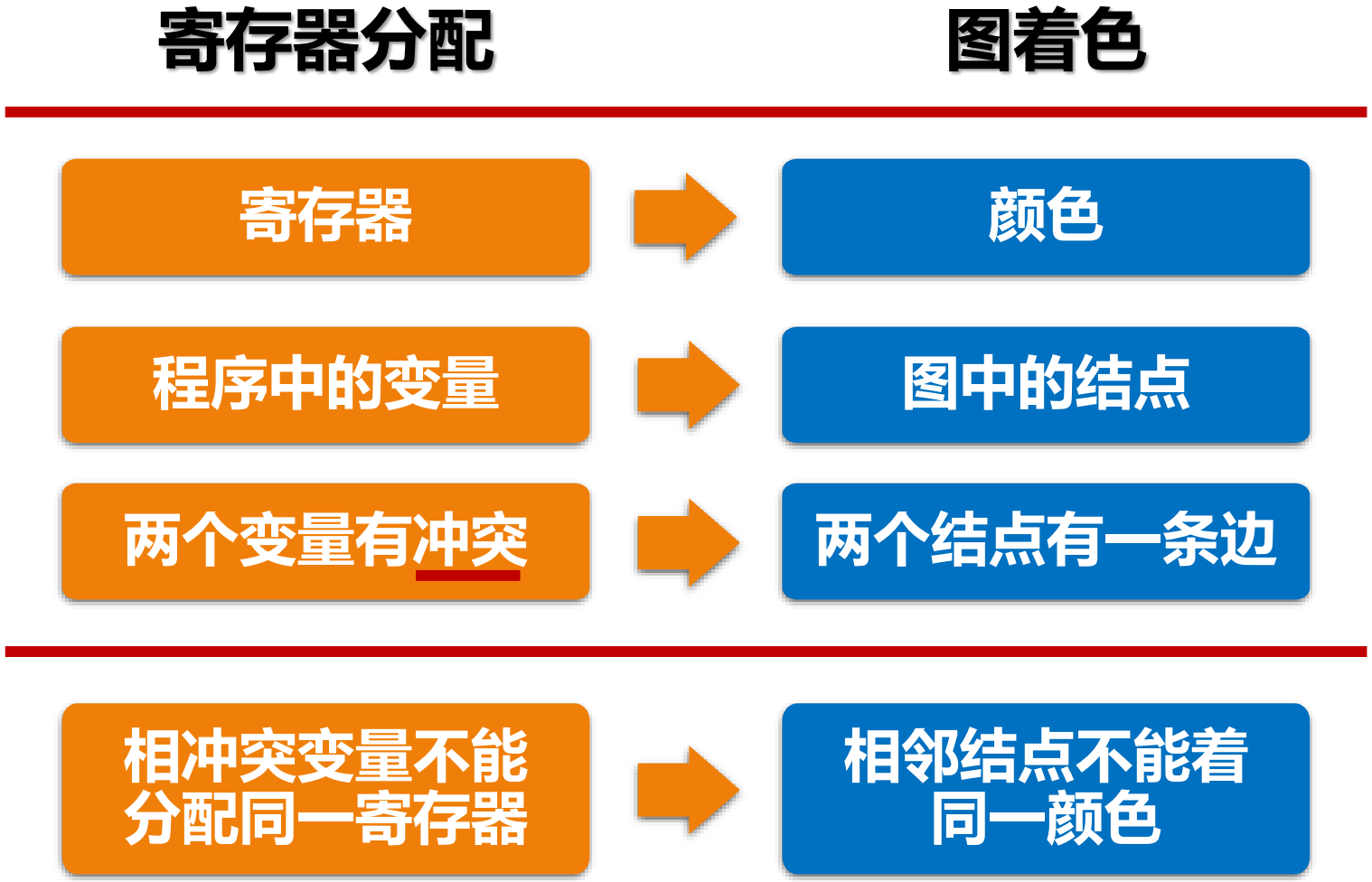
10.3.1 图着色与寄存器分配

- 用不同的颜色给无向图中的结点着色
- 相邻结点（有一条边相连）不能着同一颜色
- 不相邻的结点可以着相同的颜色



- 一个图是k-可着色(k-colorable) = 可以用k种颜色着色

10.3.1 图着色与寄存器分配



如何定义两个变量有冲突?

示例

- 考虑如下代码，有a~e共6个变量，r1~r4共4个寄存器，假设a和e在使用后不再活跃，如何进行寄存器分配?

$$a \leftarrow c + d$$
$$e \leftarrow a + b$$
$$f \leftarrow e - c$$

- 在某个点同时活跃的变量存在冲突，不能分配同一寄存器
  - 如{c,d}, {a,b}, {e,c}
- 在任何点都不同时活跃的变量不存在冲突，可以分配同一寄存器
  - 如{a, e, f}
- 方案: a, e, f分配同一个寄存器r1

$$a^{r1} \leftarrow c^{r2} + d^{r3}$$
$$e^{r1} \leftarrow a^{r1} + b^{r4}$$
$$f^{r1} \leftarrow e^{r1} - c^{r2}$$

10.3.1 图着色与寄存器分配

- 构建冲突图(interference graph)
- 将颜色看作寄存器，尝试为冲突图找到一种用k种颜色着色的方案，其中k是寄存器的数目
- 冲突图的结点是程序中的可分配对象
  - 变量，临时变量
  - 网(web): 将重复使用的变量按不同用途分割开来，形成网
    - 如循环控制变量

10.3.2 冲突图

- 一个冲突图是一个无向图G(V,E):
  - V = { v | v是程序中的可分配对象 (allocatable object)}
  - E = { e = < v1 ,v2 >| v1∈V ,v2∈V , v1和v2之间有一条边，则表示v1和v2在程序的某个点同时活跃}

一个基本块内的点：
- 语句之间
- 第一条语句之前
- 最后一条语句之后

10.3.2 冲突图

- 冲突图的结点
  - 可分配对象：理解为程序中的变量
- 冲突图的边
  - 两个结点之间有一条边：两个变量存在冲突，在程序的某个点同时活跃，不能分配同一寄存器
  - 两个结点之间没有边：两个变量不存在冲突，在程序的任何一个点都不同时活跃，可以分配同一寄存器

10.3.3 构建冲突图

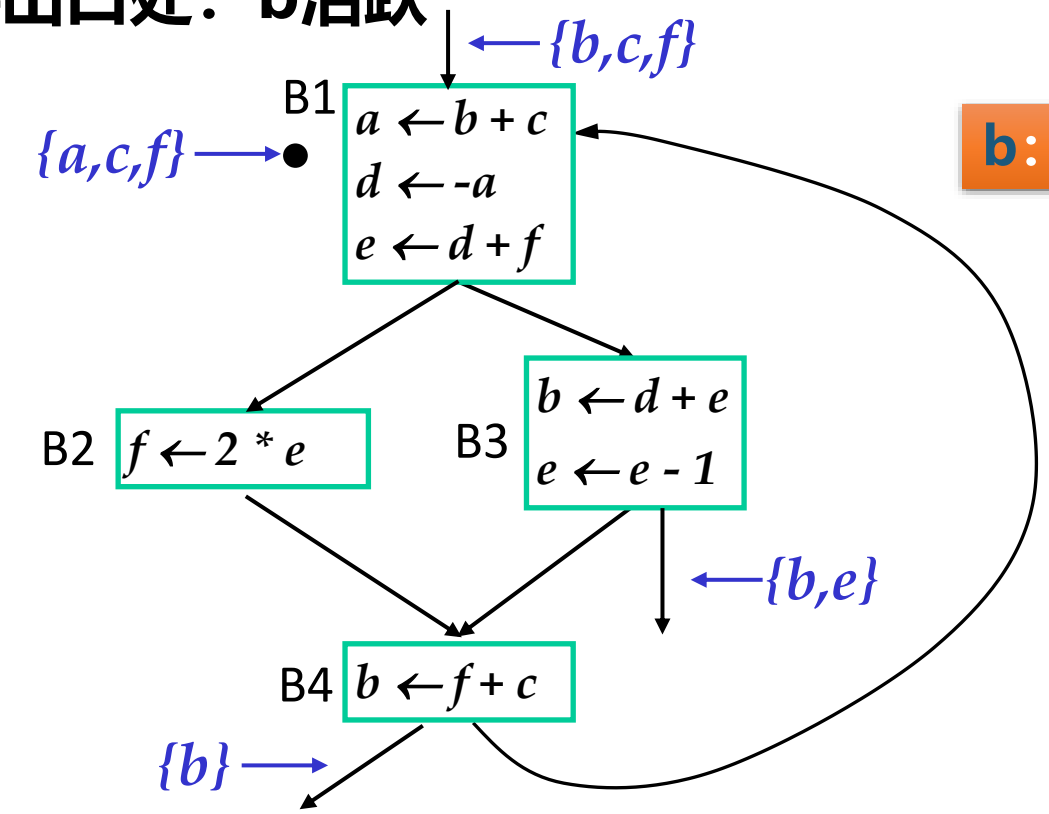
- 方法：基于活跃变量分析构建冲突图

> 活跃变量分析：一个变量v在点p开始的某条路径上被使用，那么变量v在点p是活跃的；否则，变量v在点p是死变量

- 通过活跃变量分析，获得程序中每个点的活跃变量集合

10.3.3 构建冲突图

- 示例：考虑如下控制流图，有a~e共6个变量
  - B1入口处：b,c,f活跃
  - B3出口处：b,e活跃
  - B4出口处：b活跃



确定程序中每一个点的活跃变量集合

a：被B1的S2使用，活跃

b：被B3的S1、B4的S1的定值杀死，不活跃

c：被B4的S1使用，活跃

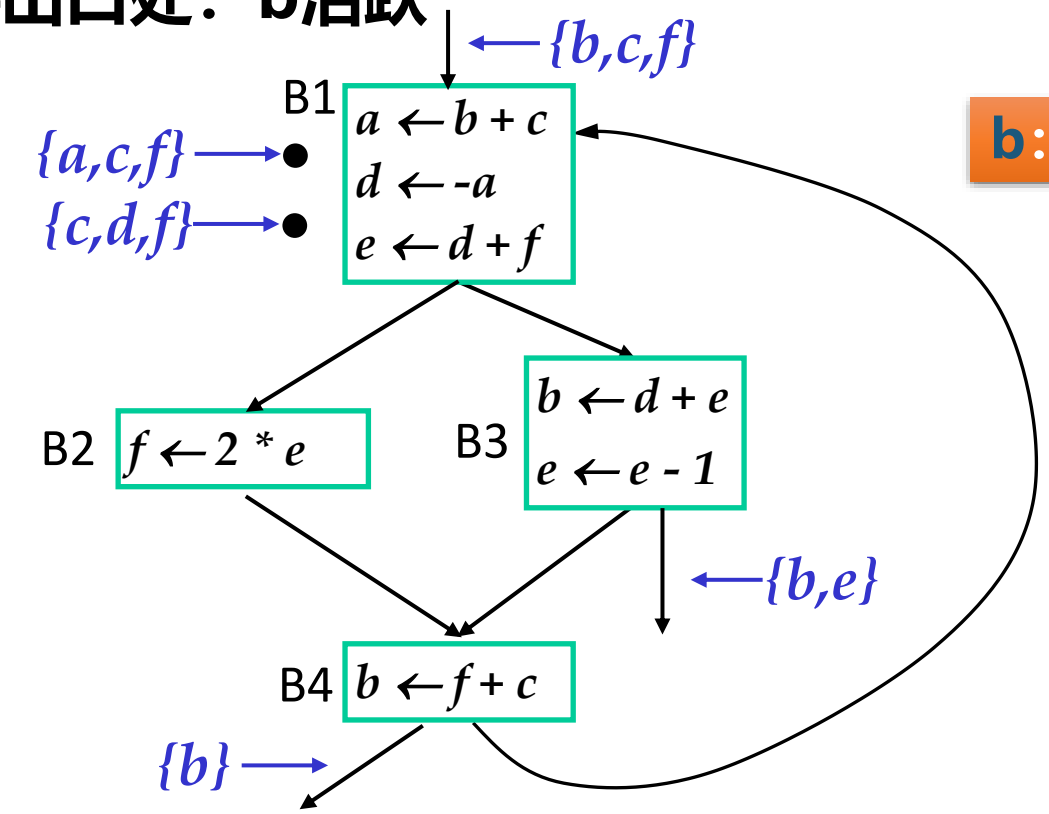
d：被B1的S2的定值杀死，不活跃

e：被B1的S3的定值杀死，不活跃

f：被B1的S3使用，活跃

10.3.3 构建冲突图

- 示例：考虑如下控制流图，有a~e共6个变量
  - B1入口处：b,c,f活跃
  - B3出口处：b,e活跃
  - B4出口处：b活跃



确定程序中每一个点的活跃变量集合

a：被B1的S1的定值杀死，不活跃

b：被B3的S1、B4的S1的定值杀死，不活跃

c：被B4的S1使用，活跃

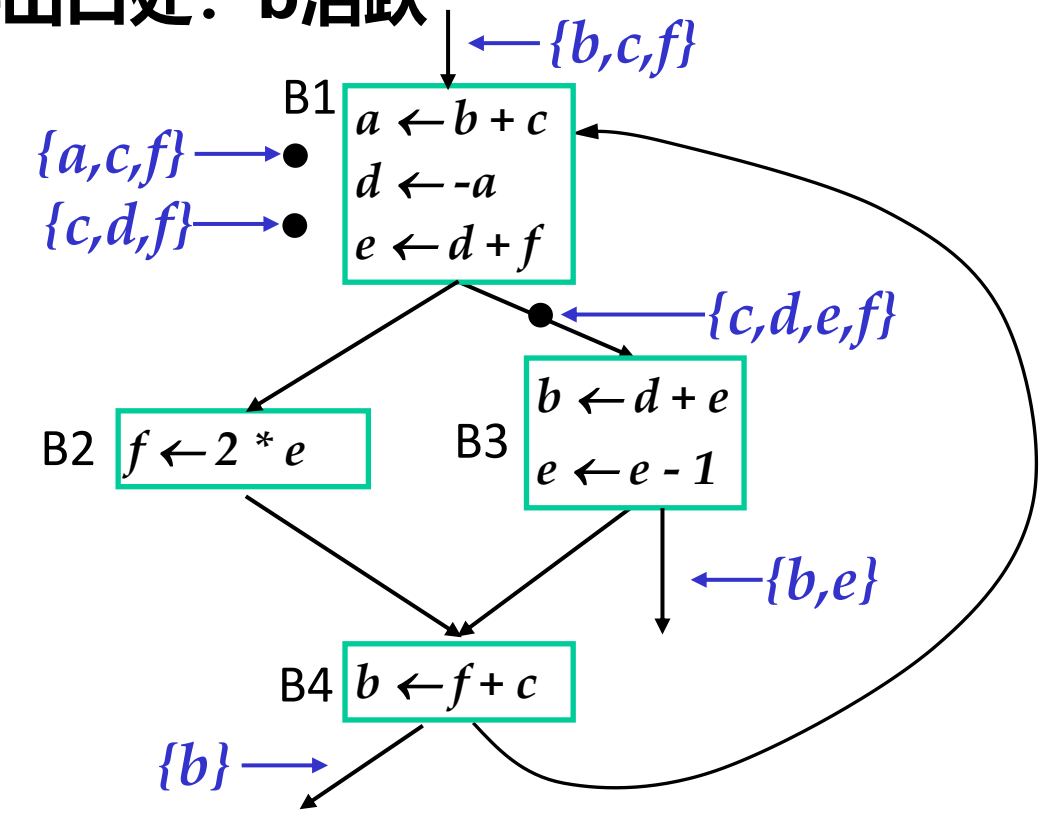
d：被B1的S3使用，活跃

e：被B1的S3的定值杀死，不活跃

f：被B1的S3使用，活跃

10.3.3 构建冲突图

- 示例：考虑如下控制流图，有a~e共6个变量
  - B1入口处：b,c,f活跃
  - B3出口处：b,e活跃
  - B4出口处：b活跃



确定程序中每一个点的活跃变量集合

a：被B1的S1的定值杀死，不活跃

b：被B3的S1的定值杀死，不活跃

c：被B4的S1使用，活跃

d：被B3的S1使用，活跃

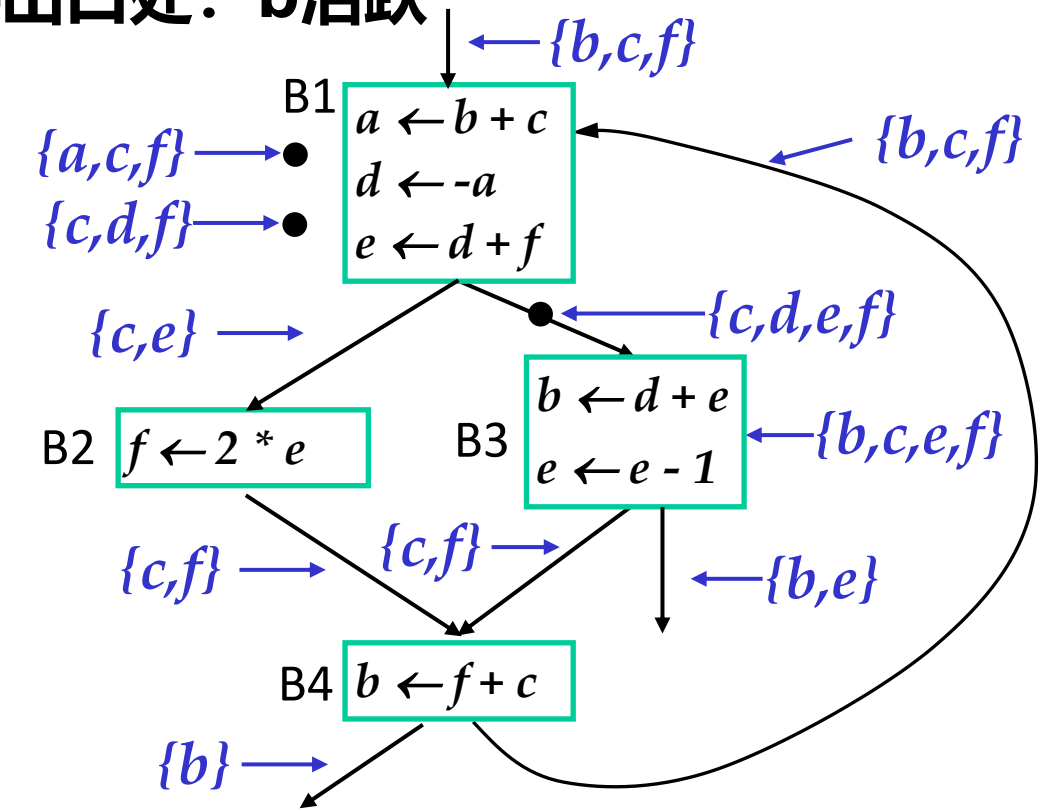
e：被B3的S1使用，活跃

f：被B4的S1使用，活跃

10.3.3 构建冲突图

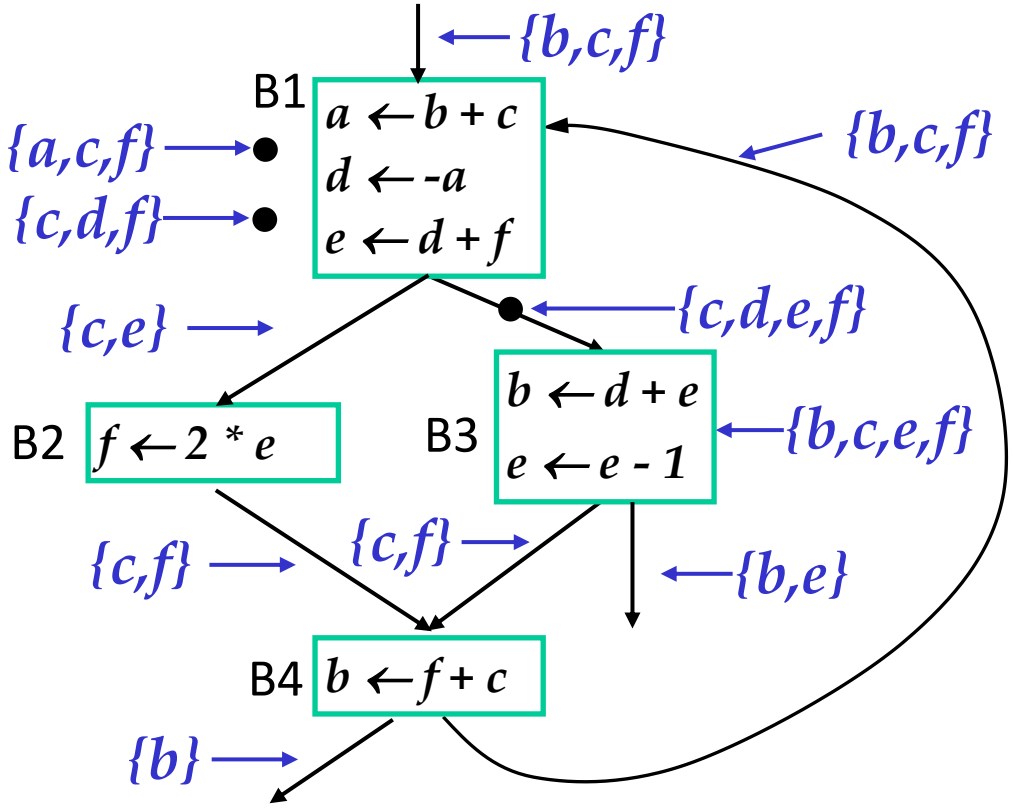
- 示例：考虑如下控制流图，有a~e共6个变量
  - B1入口处：b,c,f活跃
  - B3出口处：b,e活跃
  - B4出口处：b活跃

确定程序中每一个点的活跃变量集合



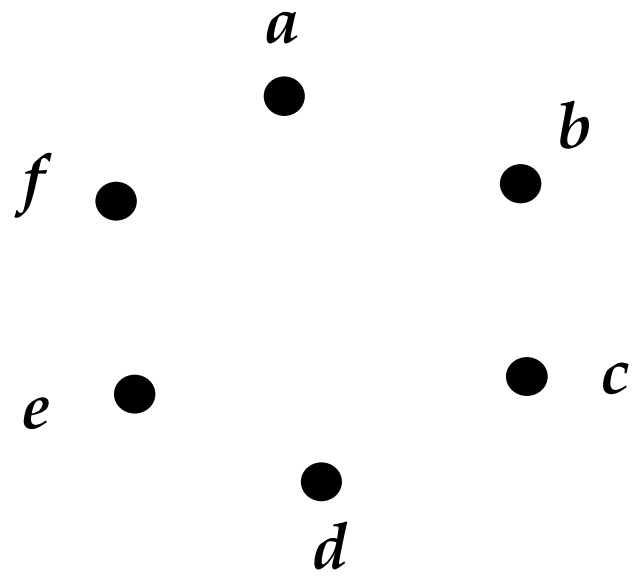
10.3.3 构建冲突图

将程序中的变量作为冲突图的结点



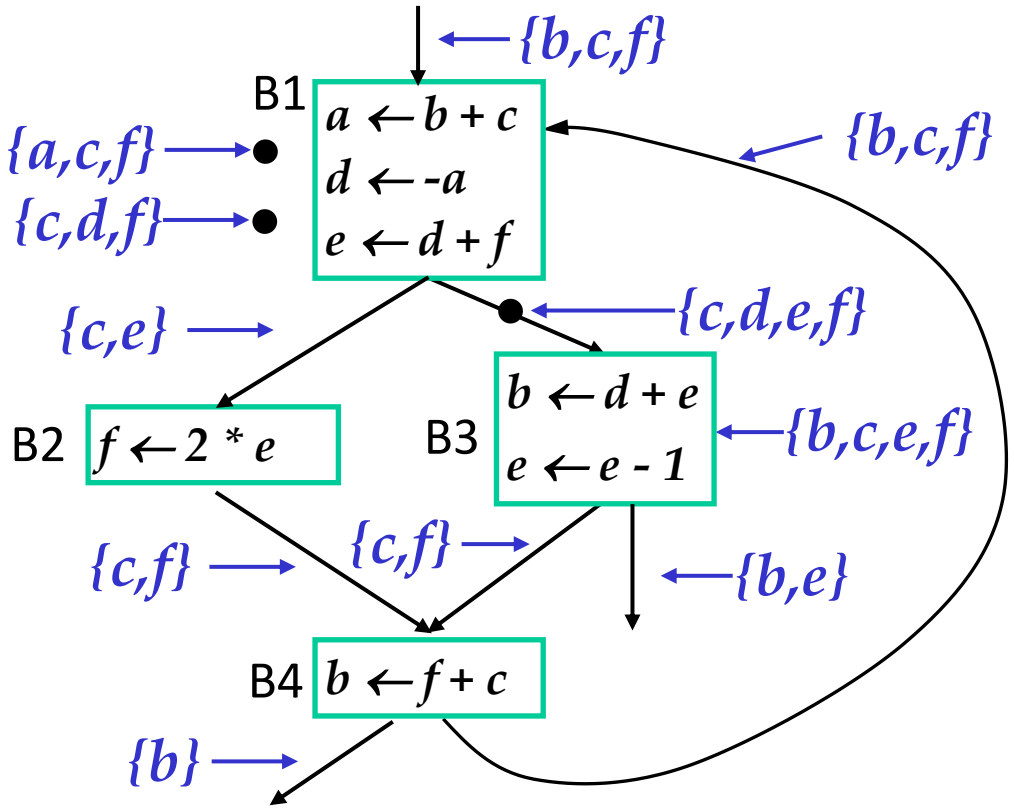
确定程序中每一个点的活跃变量集合

确定冲突图的结点



10.3.3 构建冲突图

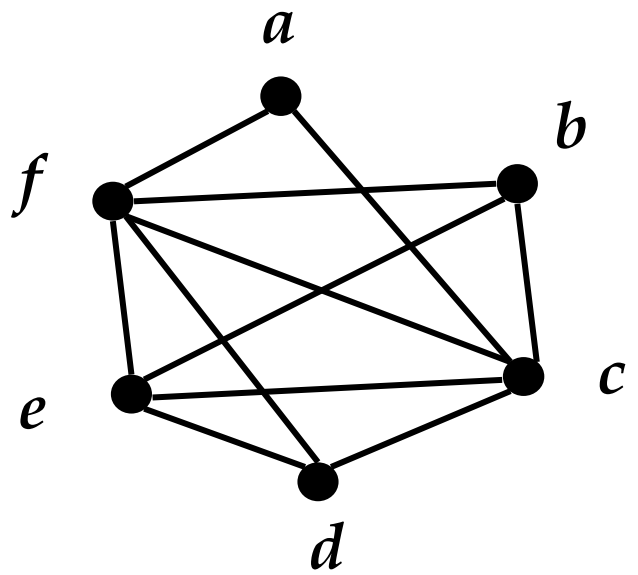
- 两个变量在程序的某个点同时活跃，则对应两个结点之间有一条边相连(表示存在冲突)



确定程序中每一个点的活跃变量集合

确定冲突图的结点

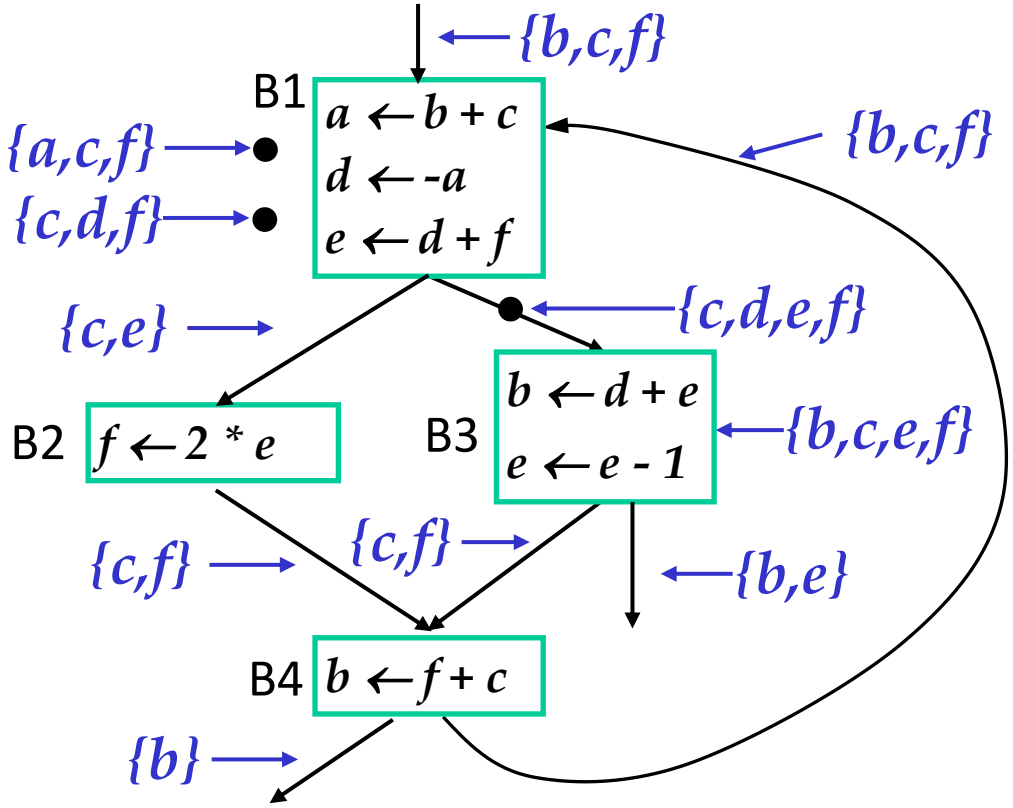
确定冲突图的边



10.3.3 构建冲突图

- 两个变量在程序的任何一个点都不同时活跃，则对应两个结点之间没有边(表示不存在冲突)

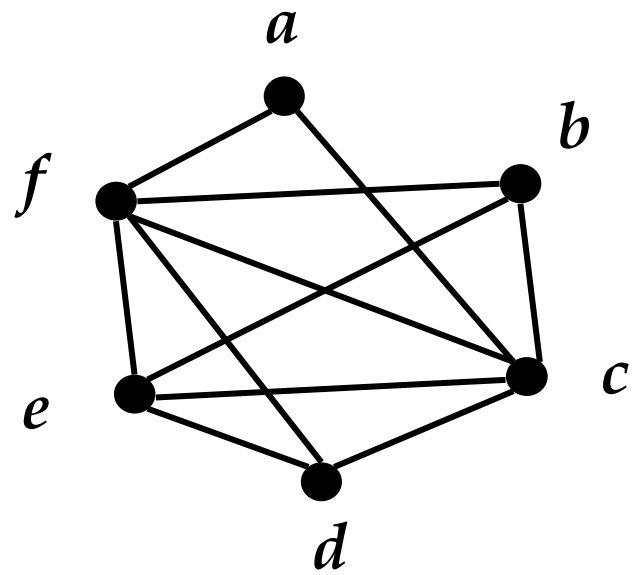
{a,b}, {a,d}, {a,e}, {b,d}之间无边



确定程序中每一个点的活跃变量集合

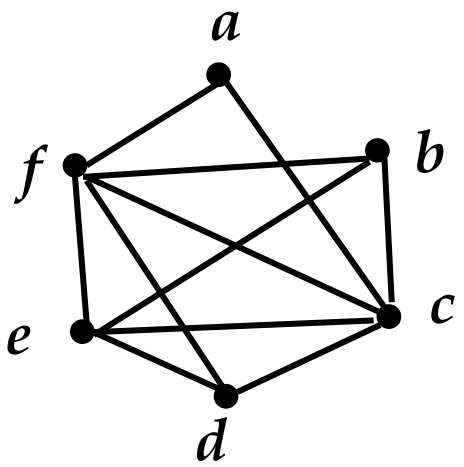
确定冲突图的结点

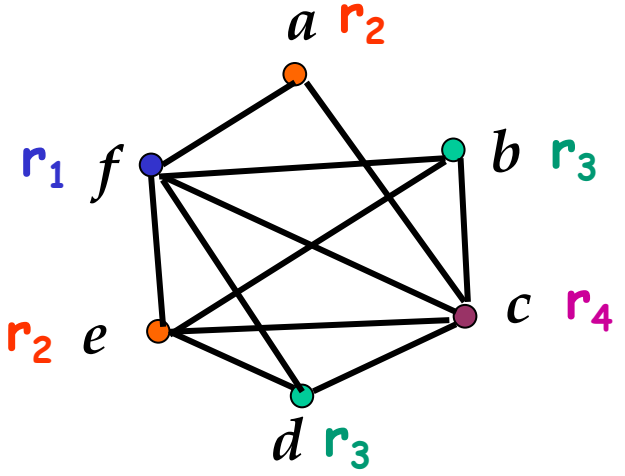
确定冲突图的边



练习

- 假设颜色数(寄存器数)=4，给出一种冲突图着色方案(寄存器分配方案)
  - 相邻结点不能着同一颜色，不相邻的结点可以着相同的颜色



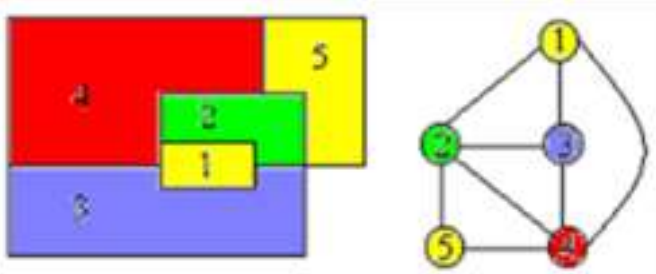


4-可着色

- 如果一个冲突图是k-可着色的，那么存在一个可行的寄存器分配方案，其所需寄存器数不超过k

10.3.4 通过化简着色

度<k定理



Kempe做四色定理证明时提出

- 定义Degree(v) = 和v相关联的边的数目
- 设结点v是G中的一个结点，且满足degree(v) < k

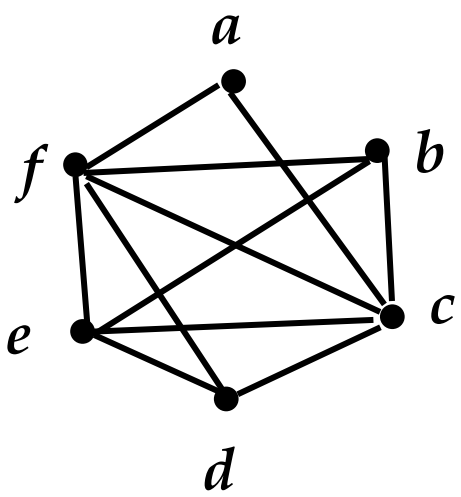
- 如果G是k-可着色的，当且仅当G'是k-可着色的

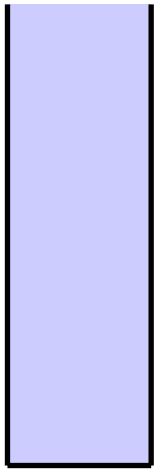
10.3.4 通过化简着色

- 第一步: 化简冲突图 (simplify)
  - ① 从冲突图G中选出一个结点v满足degree(v) < k
  - ② 把结点v以及和v相关联的边从冲突图G中删除，并将v压栈
  - ③ 重复①和②直到冲突图中只剩下一个结点

10.3.4 通过化简着色

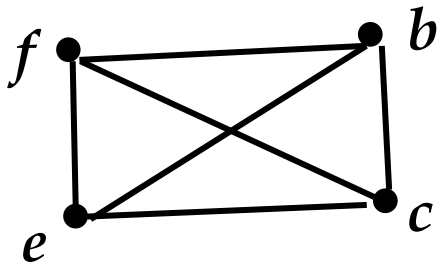
- 示例: 考虑如下冲突图的着色, k=4
  - degree(a)<4，删除a
  - degree(d)<4，删除d

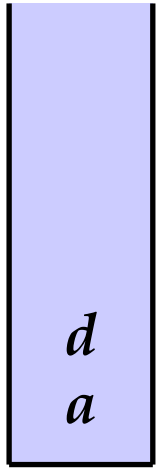




10.3.4 通过化简着色

- 示例: 考虑如下冲突图的着色, k=4
  - 到此所有结点的度数均小于4，因此可以依次删除: b, c, e, f





stack

10.3.4 通过化简着色

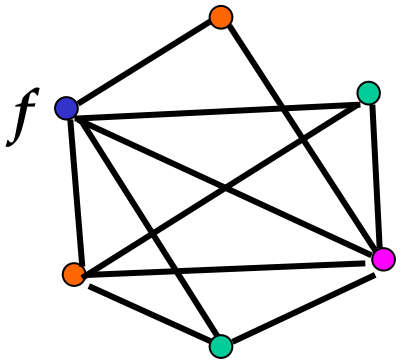
- 第二步: 选择颜色 (select)

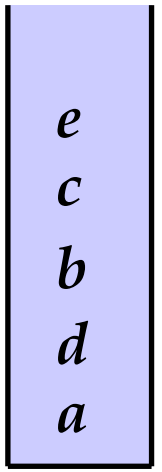
  ①从最后一个进栈的结点开始弹栈

  ②每弹出一个结点，将其重新放回到冲突图中，并为其选择一个与已着色邻居结点不同的颜色

10.3.4 通过化简着色

- 示例: 考虑如下冲突图的着色, k=4
  - 将结点依次弹栈，放回到冲突图中，为结点选择一个与已着色邻居结点不同的颜色

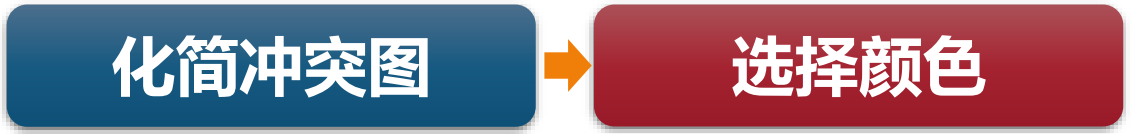




stack

10.3.4 通过化简着色

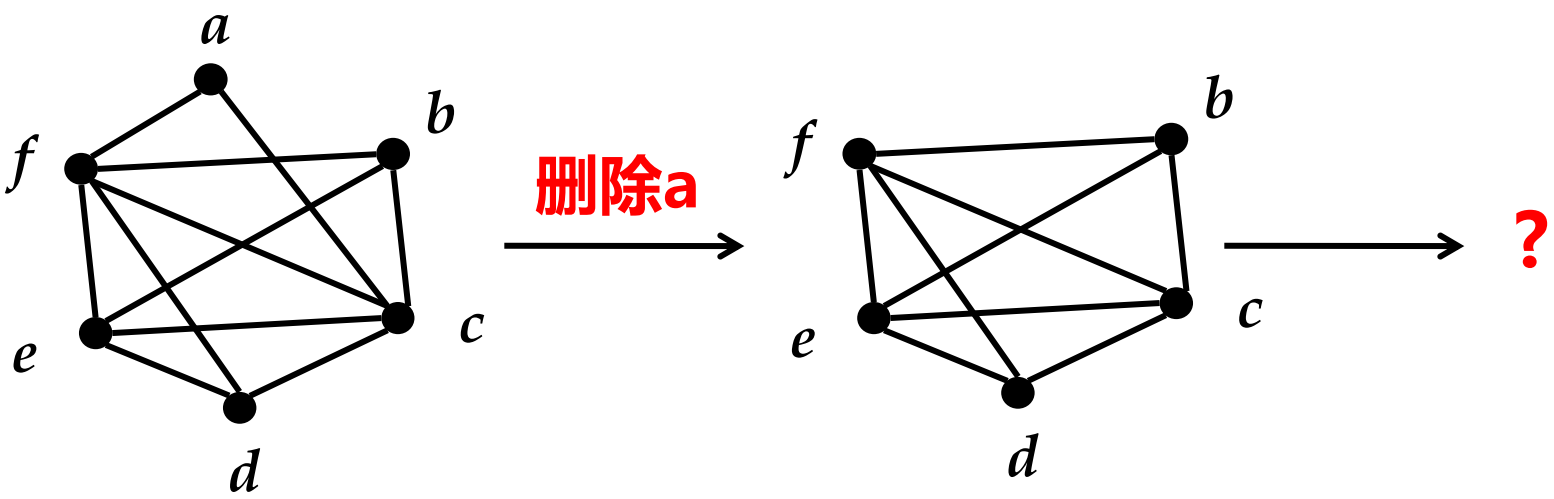
- 对于k-可着色图，通过化简着色包括2个步骤



- 如果一个冲突图是k-可着色的，则存在一个使用不超过k种颜色的着色方案
- 对于寄存器分配问题，不能保证冲突图一定是k-可着色的
- 如果冲突图不是k-可着色的，怎么办?

10.3.4 通过化简着色

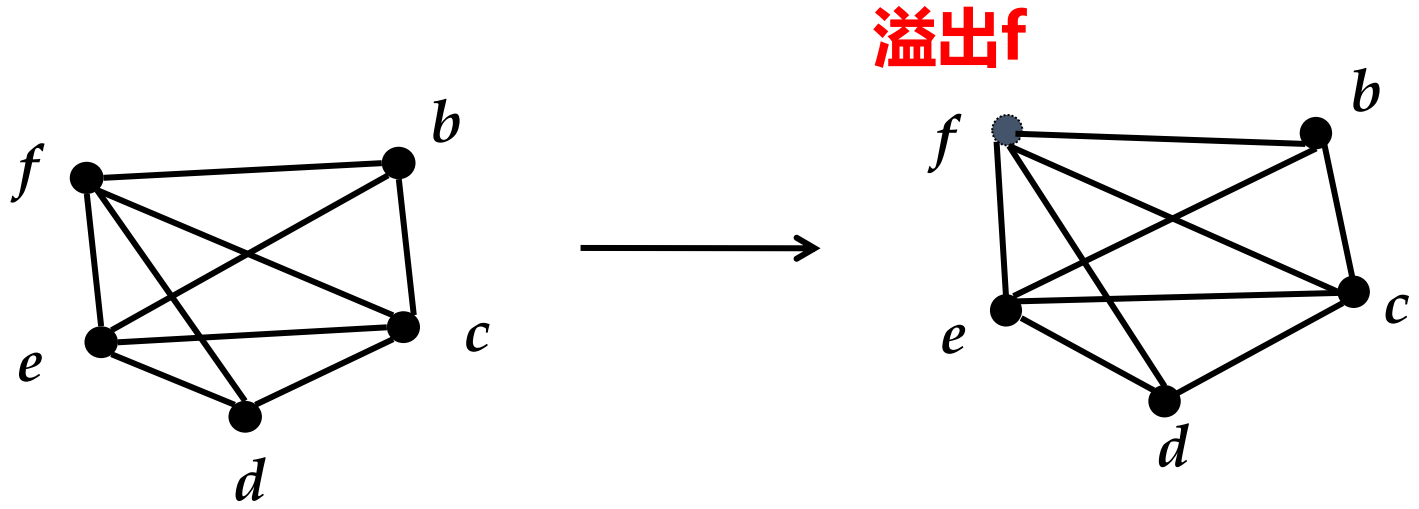
- 示例: 考虑如下冲突图的着色, k=3
  - 化简后，剩余所有结点的度数都大于等于3



如何减少冲突?

10.3.5 溢出

- 示例: 考虑如下冲突图的着色, k=3
  - 在G中删掉一个结点成为G′， G′可能成为可着色的



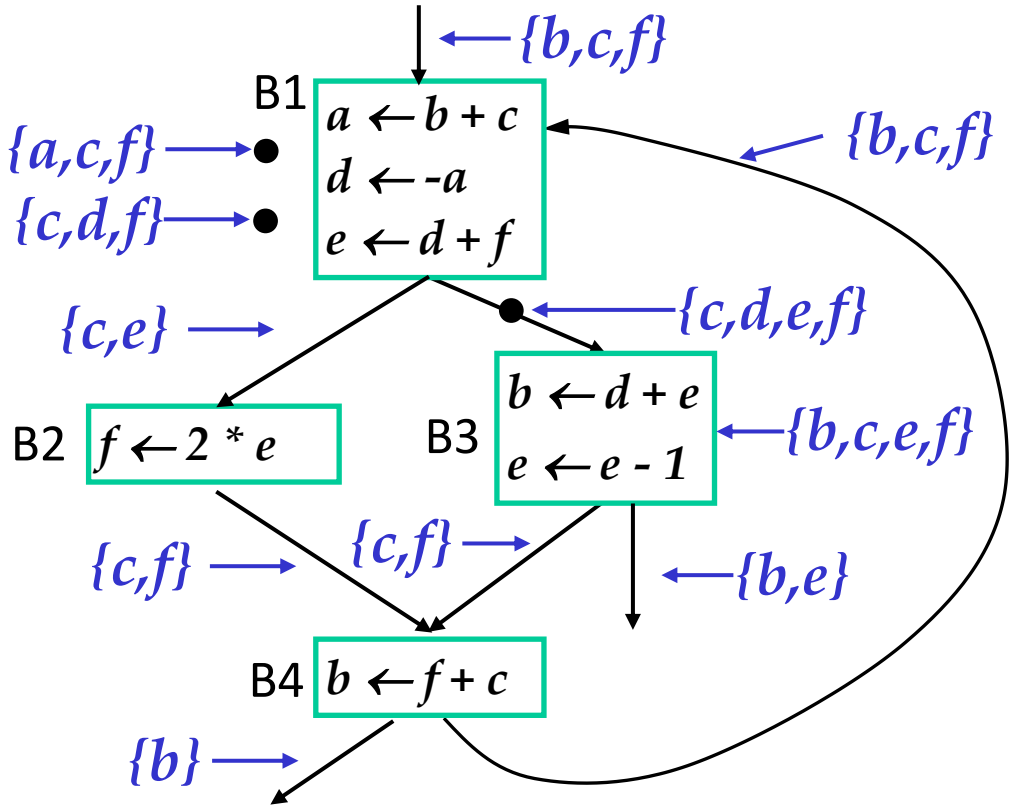
  - 通过溢出一个结点，减少冲突边，使得寄存器分配可以继续

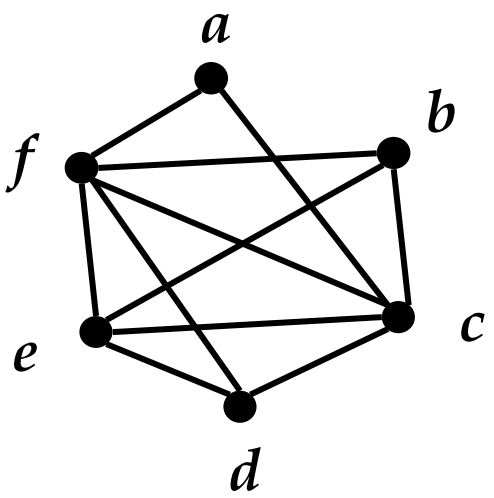
10.3.5 溢出

- **溢出(spill)：选择一个变量，在程序执行期间将它存储在存储器中而不是寄存器中**
- **溢出代码(spill code)：产生溢出则编译器需要插入访存指令**
  - **在f的每一个使用语句之前，插入**
    - **f = load f_{addr}**
  - **在f的每一个定值语句之后，插入**
    - **store f, f_{addr}**

10.3.5 溢出

- 溢出f前每个点的活跃变量集合和冲突图



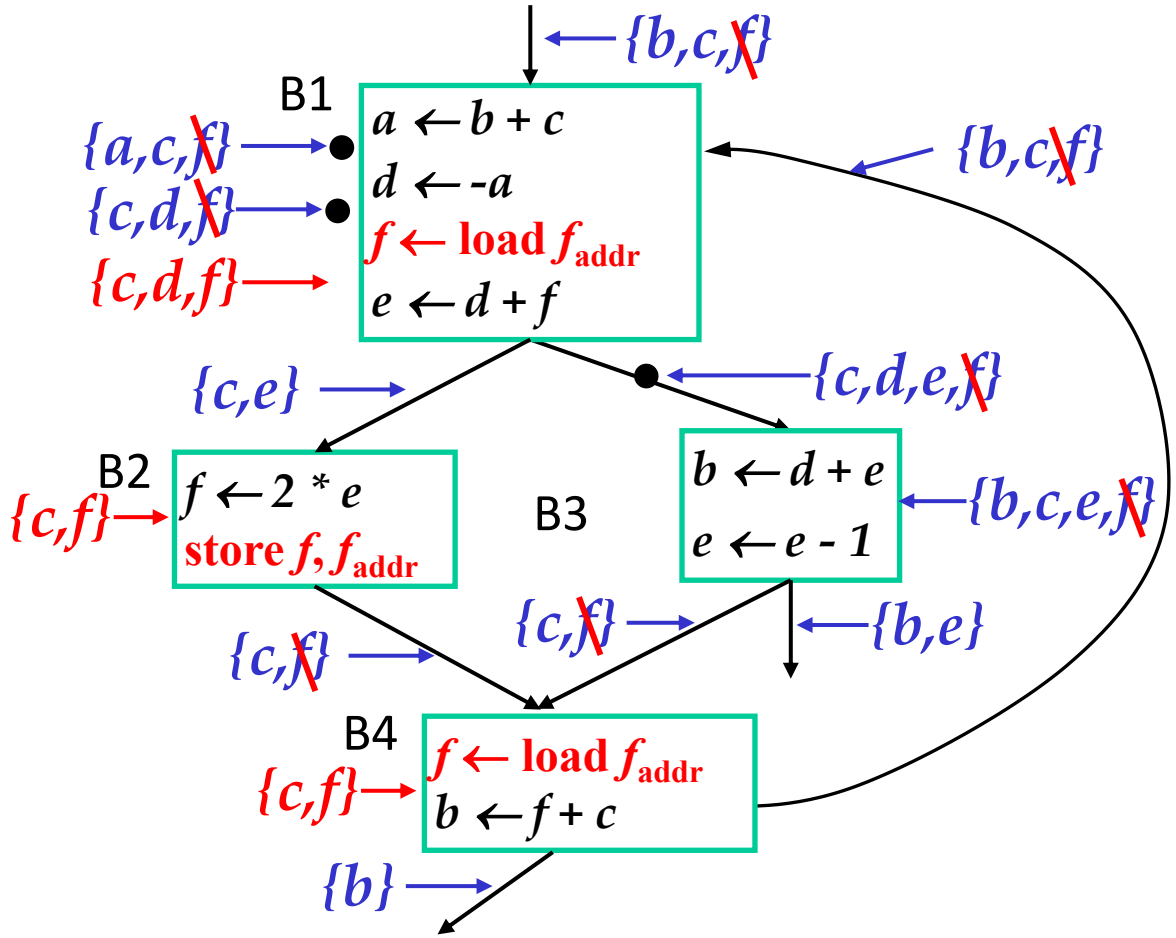


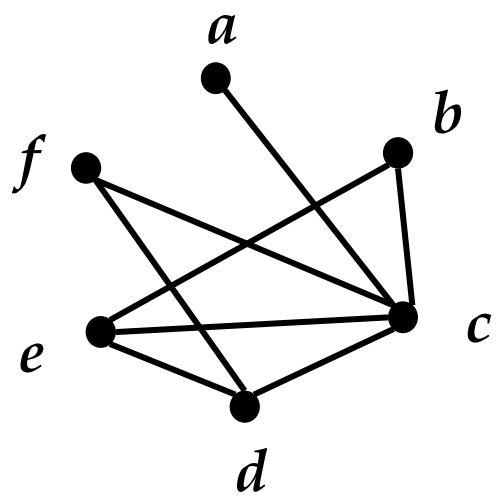
10.3.5 溢出

- 溢出f后需插入溢出代码
- f的活跃信息发生变化，f仅在下列程序点上活跃
  - 在load f_{addr} 和下一条指令(f的使用)之间
  - 在store f, f_{addr} 和前一条指令(f的定值)之间
- 溢出后需重构冲突图

10.3.5 溢出

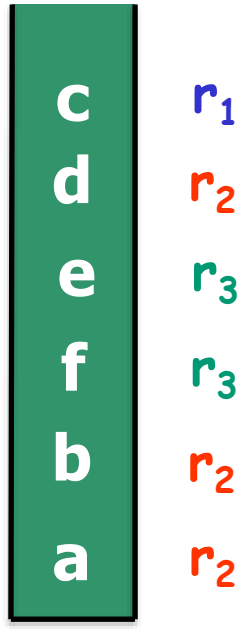
溢出后每个点的活跃变量集合和冲突图



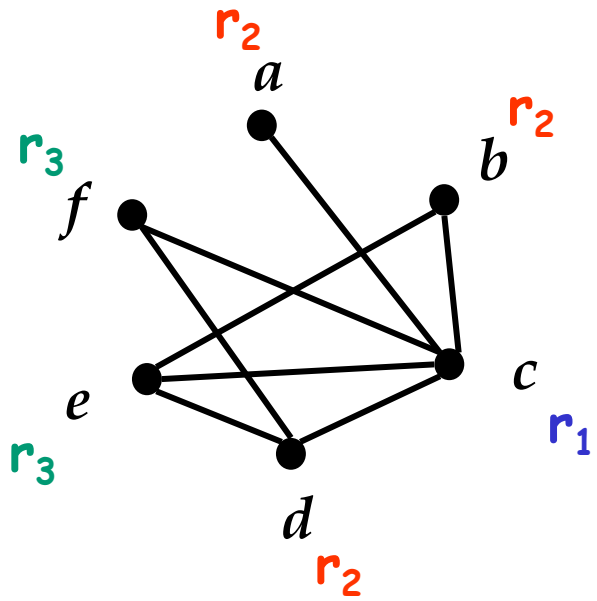


10.3.5 溢出

- 通过化简做寄存器分配 (k=3)



stack



3-可着色

10.3.5 溢出

- 溢出代价很高，要尽可能避免溢出
- 当不得不溢出时，选择哪个结点更经济?
- 使用启发式方法评估溢出代价，选择溢出一个结点
  - 冲突最多的结点 (度数最高的结点)
  - 收益最少的结点 (例如选择定值和使用少的临时变量)
  - 溢出代价较小的结点 (例如避免在内层循环中进行溢出)

10.3.6 合并

- 考虑如下传送指令，x赋值给y后不再使用x
  - x和y不同时活跃 (没有冲突边)
  - x和y可以使用同一个寄存器
  - 从而可以删除传送指令

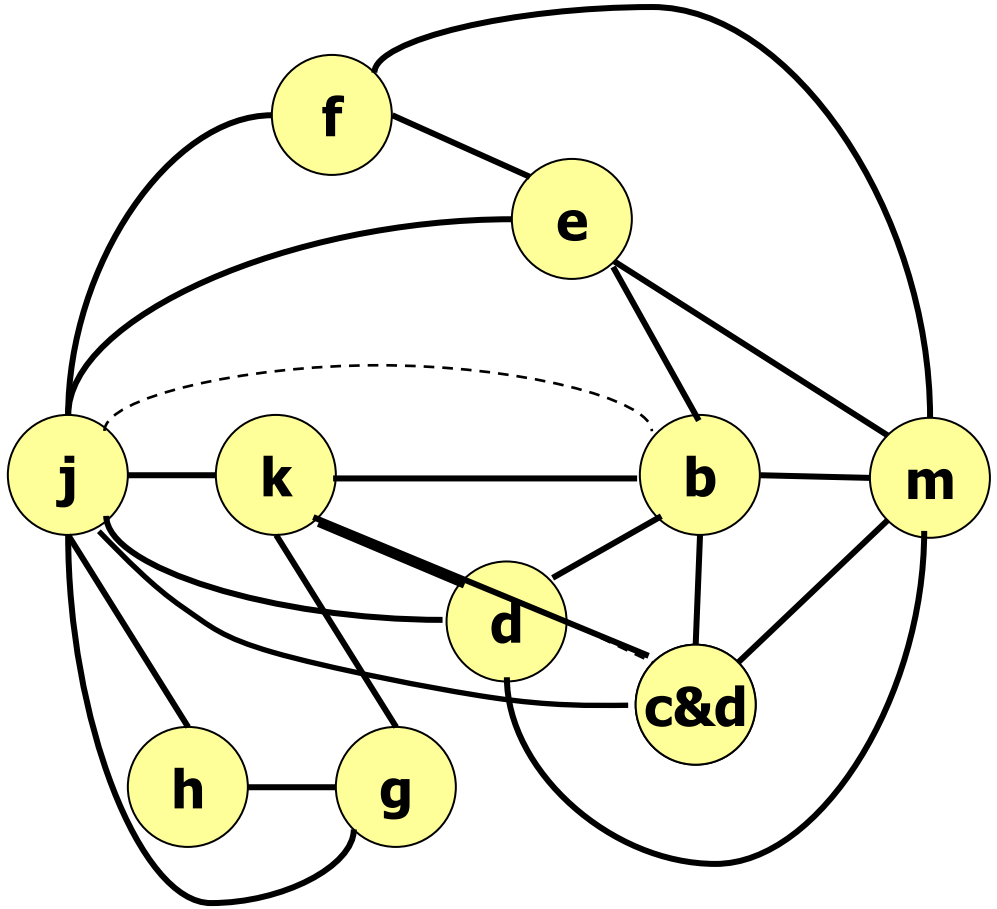
$x \leftarrow \ldots$
$\ldots$
$y \leftarrow x$（之后不再使用x）
$\ldots$
$\ldots \leftarrow y$

- 合并(coalescing): 寻找可以删除的传送指令并合并结点的过程
  - 如果x和y不冲突，并且由一条传送指令相关联，则可将x和y合并成一个结点，使x和y使用同一个寄存器(从而减少冲突图的结点数)

10.3.6 合并

- 示例：考虑如下代码，合并传送相关的c-d

```
LIVE-IN: k  j
        g ← mem[j+12]
        h ← k - 1
        f ← g + h
        e ← mem[j+8]
        m ← mem[j+16]
        b ← mem[f]
        c ← e + 8
        d ← c
        k ← m + 4
        j ← b
LIVE-OUT: d  k  j
```



虚线表示传送指令

合并与溢出的对比

- 合并
  - 通过减少结点数减少寄存器分配的压力
  - 活跃变量信息不会发生变化，无需重构冲突图
  - 合并后的新结点的边是原来两个结点的边的并集

- 溢出
  - 通过减少冲突边减少寄存器分配的压力
  - 活跃变量信息会发生变化，需要重构冲突图

10.3.7 Chaitin图着色分配算法

① 确定可分配对象(renumber)

② 构建冲突图(build)

③ 合并传送相关且无冲突边的结点(coalesce)

④ 启发式方法评估每个结点的溢出代价(spill cost)

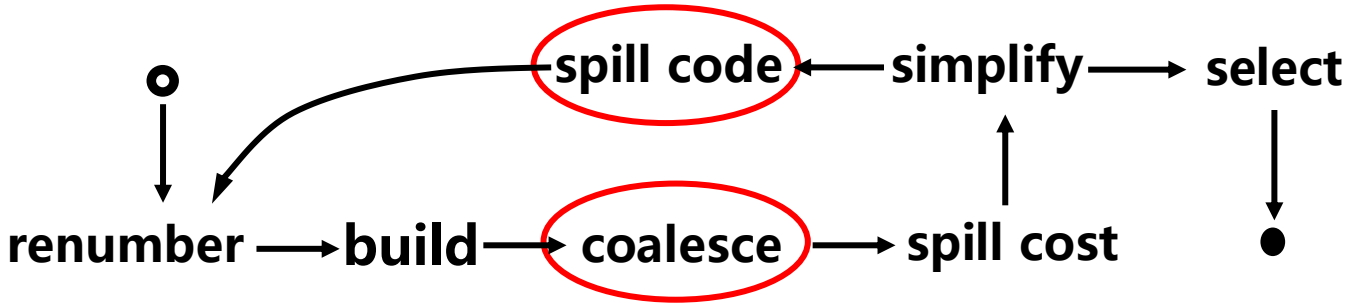
⑤ 化简冲突图，将度小于k的低度数结点删除并入栈(simplify)

⑥ 如果不再有低度数结点，则根据③选择一个结点溢出，插入溢出代码(spill code)，重复①-⑤直至所有结点入栈

⑦ 将结点依次弹栈，为其选择一个颜色(select)

10.3.7 Chaitin图着色分配算法

- 构建冲突图、合并结点、化简冲突图、溢出结点的迭代过程



- Chaitin算法法奠定了图着色寄存器分配算法的基础
- 改进算法主要在合并和溢出部分

本讲小结

- 寄存器分配是编译器必须具备的优化
- 寄存器分配概的原则
  - 尽可能将更多的变量保存在寄存器中
  - 尽可能将频繁使用的变量保存在寄存器中
- 基于使用计数的局部和全局寄存器分配方法
  - 实现简单，实际有效
  - “全局”只是跨越循环内的基本块边界
- 基于图着色的全局寄存器分配方法
  - 将寄存器分配问题转化为图着色问题
  - Chaitin算法: 构建冲突图，合并结点，评估溢出代价，化简冲突图，溢出结点，选择颜色

作业

- 对示例代码运用Chaitin算法进行寄存器分配，假设寄存器数为4 (k=4)
  - 给出合并，化简，选择颜色的具体过程和最终的寄存器分配方案
  - 如需溢出，溢出度数最高的结点

```
LIVE-IN: k  j
        g ← mem[j+12]
        h ← k - 1
        f ← g + h
        e ← mem[j+8]
        m ← mem[j+16]
        b ← mem[f]
        c ← e + 8
        d ← c
        k ← m + 4
        j ← b
LIVE-OUT: d  k  j
```

参考资料

COMP CCRG Open MP

- 《高级编译器设计与实现》(鲸书) 第16章
- 《现代编译原理C语言描述》(虎书) 第11章
- 论文
  - Richard A. Freiburghouse. Register Allocation via Usage Counts，CACM，vol.17, No.11, Nov. 1974
  - G.J. Chaitin. et. al, Register allocation via coloring. *Computer Languages*, 1981
  - P. Briggs, et al. Improvement to graph coloring register allocation. *Transactions on Programming Languages and Systems*,1994
  - L. George and Andrew W. Appel. Iterated register coalescing. *Transactions on Programming Languages and Systems (TOPLAS)*, 1996
  - A. B. Kempe, On the geographical problem of four colours. *American Journal of Mathematics.*