北京大学信息科学技术学院

2022年春季学期《编译技术》

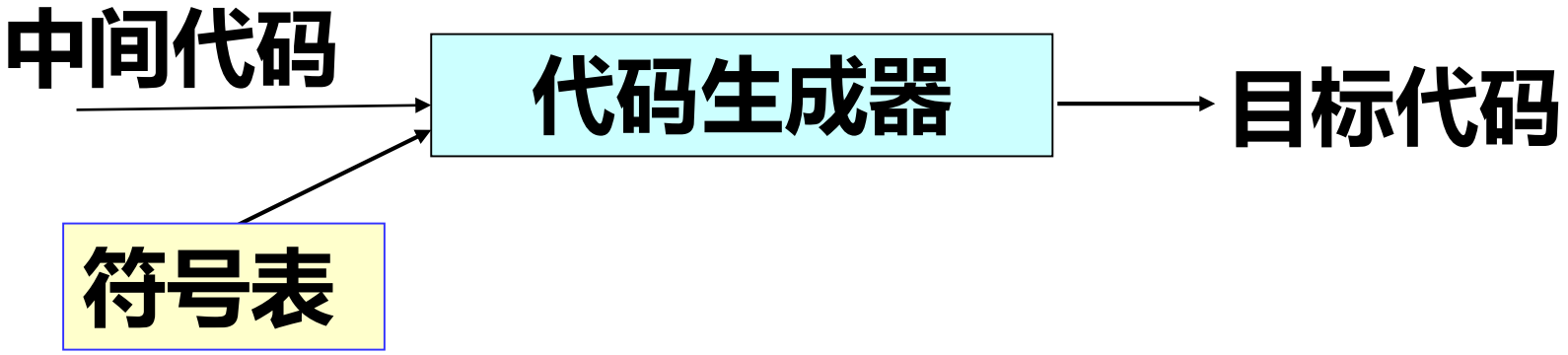
第八章　目标代码生成

Target Code Generation

主要内容

- 目标代码生成概述
- 目标计算机模型
- 基本块和流图
- 基本块的优化
- 目标代码生成方法
- 窥孔优化
- 寄存器分配

目标代码生成概述



- 目标代码一般有以下三种形式：
  - 能够立即执行的机器语言代码，所有地址已经定位
  - 待装配的机器语言模块。执行时，由连接装配程序把它们和某些运行程序连接起来，转换成能执行的机器语言代码；
  - 汇编语言代码。尚须经过汇编程序汇编，转换成可执行的机器语言代码。

代码生成主要解决的问题

- 代码生成主要考虑的问题：
  - 如何使生成的目标代码执行的更快/更短
  - 如何充分利用计算机的寄存器，减少目标代码中访问内存单元的次数
- 基本任务
  - 指令选择
  - 寄存器分配
  - 指令排序

目标机器

- 通用寄存器 $R_0, R_1, \ldots, R_{n-1}$
- 使用三地址机器模型，主要指令：
  - **LD dst, addr；** 把地址addr中的内容加载到dst所指寄存器。addr：内存地址/寄存器
  - **ST x, r；** 把寄存器r中的内容保存到x中。
  - **OP dst, src1, src2；** 把src1和scr2中的值运算后将结果存放到dst中。dst, src1, src2 都是寄存器。
  - **BR L；** 控制流转向标号L的指令
  - **Bcond r, L；** 对r中的值进行测试，如果为真则转向L。

目标机器的地址方式

- 有关地址方式及它们的汇编语言形式如下表所示：

地址方式	汇编	地址
变量	x	value(x)
数组	a(Ri)	a + contents(R_i)
常量（直接地址）	M	M
寄存器方式	R_i	R_i
间接寄存器	*R_i	contents(R_i)
索引方式	c(R_i)	c + contents(R_i)
间接索引方式	*c(R_i)	contents (c + contents(Ri))

contents(a)表示由a所代表的寄存器或存储单元的内容

例子

- x=y-z
  - LD R1, y //R1=y
  - LD R2, z //R2=z
  - SUB R1, R1, R2 //R1=R1-R2
  - ST x, R1 //x=R1
- b=a[i]
  - LD R1, I //R1=i
  - MUL R1, R1, 8 //R1=R1*8
  - LD R2, a(R1) //R2=contents(a+contents(R1))
  - ST b, R2 //b = R2

程序及指令的代价

- 程序的代价有不同的度量方式
  - 最短编译时间、目标程序大小、运行时间、能耗
  - 无法判定一个目标程序是否最优

- 假设:相同指令的代价(指令长度)与运算分量寻址模式相关
  - **LD R0, R1**；代价为1
  - **LD R1, *100(R2)**；代价为2

基本块和流图

- 基本块 (Basic Block)
  - 控制流只能从第一个指令进入
  - 除基本块的最后一个指令外，控制流不会跳转
- 流图 (即控制流图, Control Flow Graph)
  - 表示各基本块（可能发生的）执行次序的有向图
  - 流图的结点代表基本块、有向边代表基本块之间的执行次序
- 流图可以作为优化的基础
  - 它指出了基本块之间的控制流
  - 可以根据流图了解到一个值是否会被使用等信息

示例

```
1)  i = 1
2)  j = 1
3)  t1 = 10 * i
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

```
for i from 1 to 10 do
  for j from 1 to 10 do
     a[i,j]=0.0

for i from 1 to 10 do
     a[i,i]=1.0
```

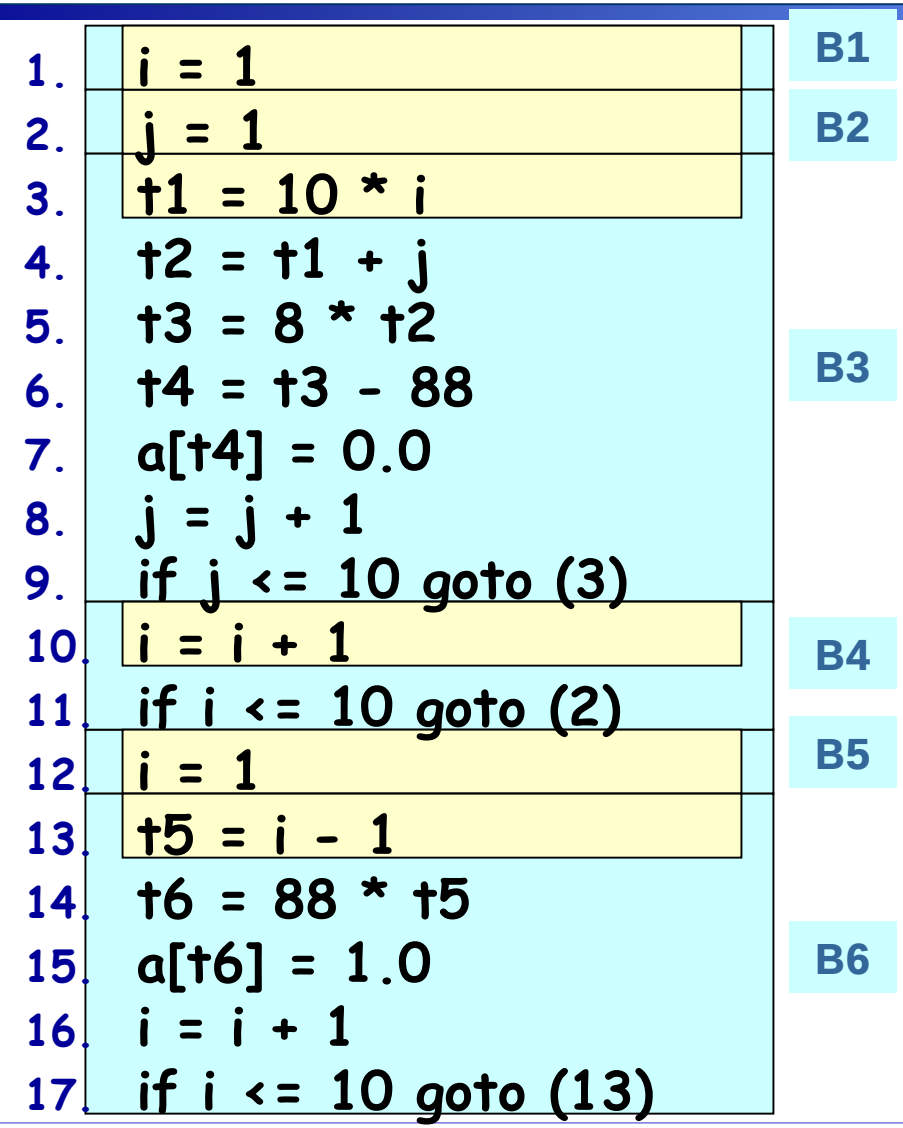
划分基本块的算法

- 输入：三地址指令序列
- 输出：基本块的列表
- 方法：
  - 确定首指令（leader, 基本块的第一个指令）
    - 第一个三地址指令
    - 任意一个条件或无条件转移指令的目标指令
    - 紧跟在一个条件/无条件转移指令之后的指令
  - 确定基本块
    - 每个首指令对应于一个基本块：从首指令开始到下一个首指令

基本块划分算法

```
leaders = {1}                    // start of program
for i = 1 to |n|                 // all instructions
    if instr(i) is a branch
        leaders = leaders ∪ targets of instr(i) ∪ instr(i+1)
worklist = leaders
While worklist not empty
    x = first instruction in worklist
    worklist = worklist - {x}
    block(x) = {x}
    for i = x + 1; i <= |n| && i not in leaders; i++
        block(x) = block(x) ∪ {i}
```

基本块的例子



leader

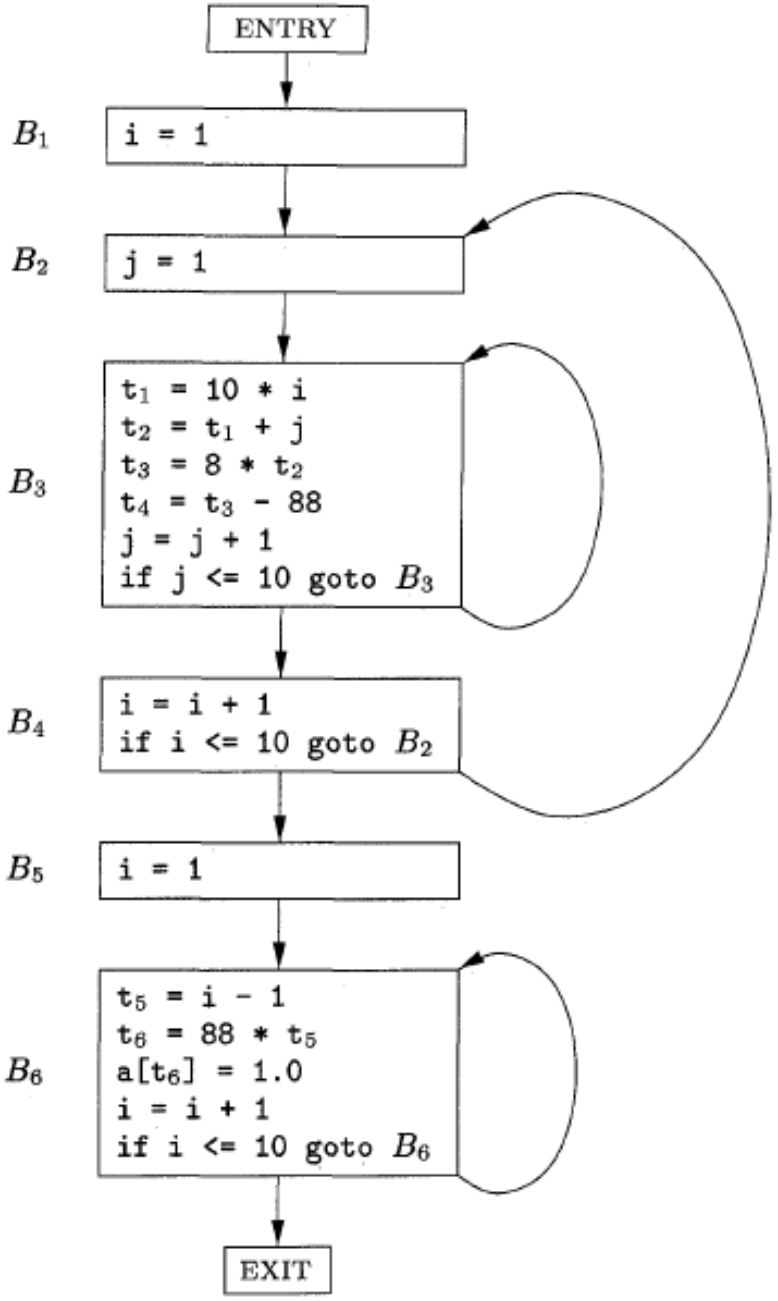
基本块

流图的构造

- 流图的顶点是基本块
- 两个顶点B和C之间存在一条有向边 *iff* 基本块C的第一个指令可能在B的最后一个指令之后执行。
  - B的结尾指令是一条跳转到C的开头的条件/无条件语句
  - 在原来的序列中，C紧跟在B之后，且B的结尾不是无条件跳转语句
  - 称B是C的前驱(predecessor)，C是B的后继(successor)
- 入口(ENTRY)，出口(EXIT)结点
  - 流图中额外添加的顶点，不对应中间代码（基本块）
  - 入口到第一条指令有一条边
  - 从任何可能最后执行的基本块到出口有一条边

控制流图

```
1.  i = 1                          B1
2.  j = 1                          B2
3.  t1 = 10 * i
4.  t2 = t1 + j
5.  t3 = 8 * t2
6.  t4 = t3 - 88                   B3
7.  a[t4] = 0.0
8.  j = j + 1
9.  if j <= 10 goto (3)
10. i = i + 1                      B4
11. if i <= 10 goto (2)
12. i = 1                          B5
13. t5 = i - 1
14. t6 = 88 * t5
15. a[t6] = 1.0                    B6
16. i = i + 1
17. if i <= 10 goto (13)
```

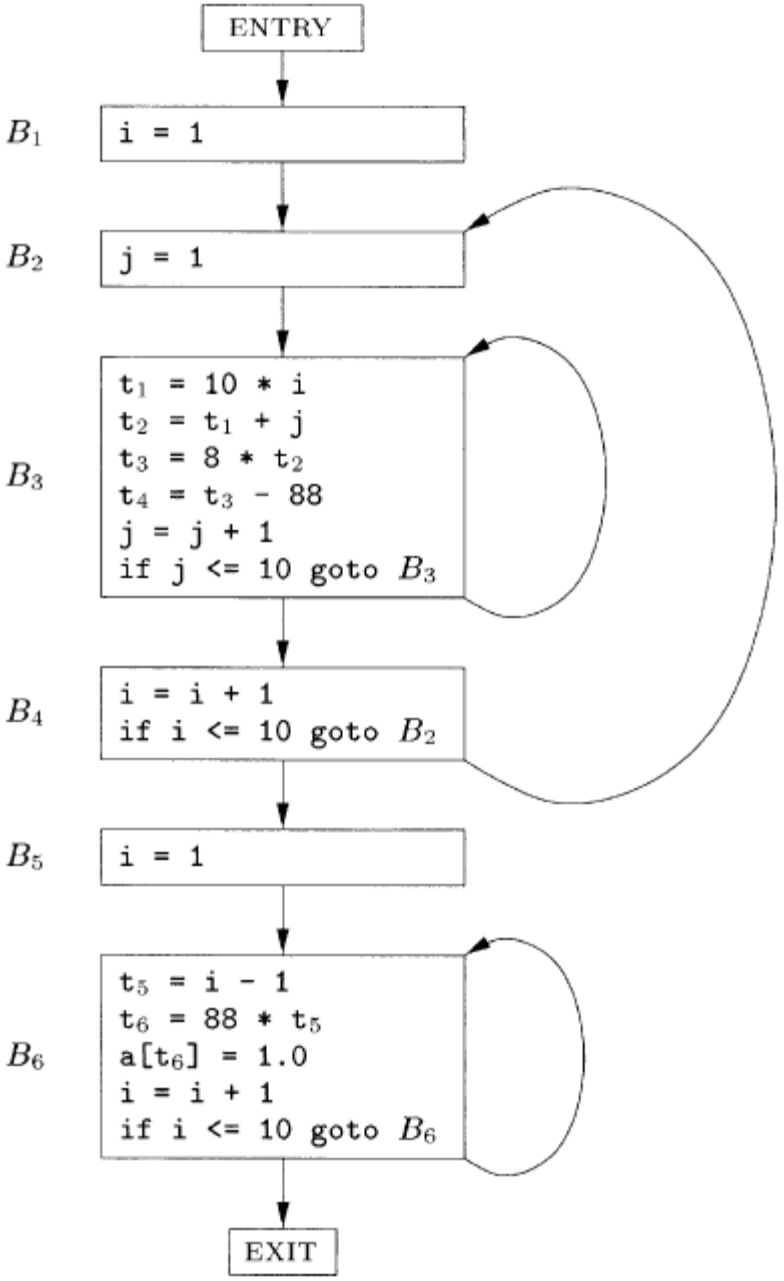


循环（loop）

- 程序的大部分运行时间花费在循环上
  - 因此循环是编译器需要识别的重点
- 循环的定义
  - 循环L是一个结点集合
  - 存在一个循环入口（loop entry）结点。是唯一的、其前驱可以在循环L之外的结点，到达其余结点的路径必然先经过这个入口结点；
  - 其余结点都存在到达入口结点的非空路径，且路径都在L中。

循环的例子

- 循环
  - {B3}
  - {B6}
  - {B2,B3,B4}
    - B2为入口结点
    - B1，B5，B6不在循环内的理由
      - 到达B1可不经过B2
      - B5，B6没有到达B2的结点



后续使用信息

- 变量值的使用 （def-use chain）
  - 三地址语句i向x赋值、如果j的运算分量为x，且从i开始有一条路径到达j，且路径上没有对x赋值，那么j就使用了i处计算得到的x的值。
  - 即：x在语句i后的程序点上活跃(live)
    - 即在程序执行完语句i的时刻，x中存放的值将被后面的语句使用
    - 不活跃是指变量中存放的值不会被使用，而不是变量不会被使用；
- 后续使用信息的用途：代码生成
  - 如果x在i处不活跃，且x占用了一个寄存器，可以把这个寄存器用于其它目的。

确定基本块中的活跃性、后续使用

- **输入**：基本块B，开始时B的所有非临时变量都是活跃的；
- **输出**：各个语句i上的变量的活跃性、后续使用信息
- **算法**：
  - 从B的最后一个语句开始反向扫描
  - 对于每个语句 i：x=y+z。
    - 把x, y, z 的当前活跃性信息/使用信息关联到语句i
    - 设置x为“不活跃”、“无后续使用”
    - 设置y和z为“活跃”，它们的使用为语句i

例子

- 假设 i, j, a 不是临时变量。它们在出口处活跃，其余变量不活跃

```
3)   t1 = 10 * i
4)   t2 = t1 + j
5)   t3 = 8 * t2
6)   t4 = t3 - 88
7)   a[t4] = 0.0
8)   j = j + 1
```

i,j,a live, i used at 3
i,j,a, t1 live, t1,j used at 4
i,j,a, t2 live, t2 used at 5
i,j,a, t3 live, t3 used at 6
i,j,a, t4 live, t4 used at 7
i,j,a live, j used at 8
i,j,a live

基本块的优化

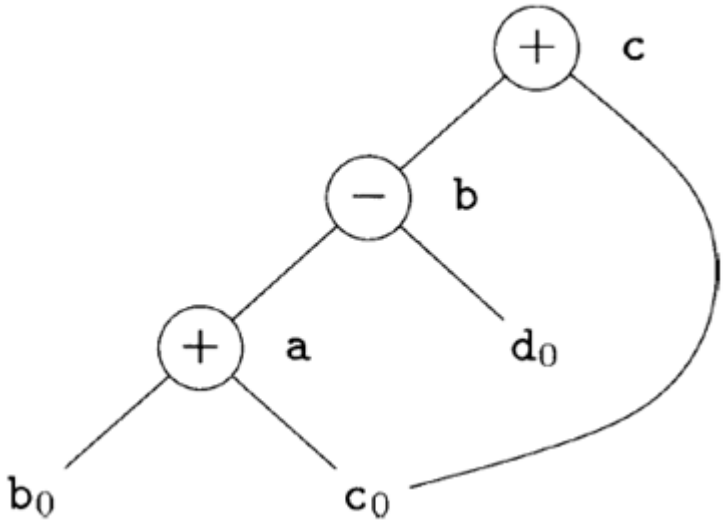
- 针对基本块的优化可以有很好的效果
  - 基本块中各个指令要么都执行，要么都不执行
- 基本块可以用**DAG**表示
  - 每个变量对应于DAG的一个结点，代表其初始值；
  - 每个语句s对应一个结点N，代表计算得到的值
    - N的子结点对应（其运算分量当前值的）其它语句
    - 结点N的标号是s的运算符
    - N和一组变量关联，表示s是在此基本块内最晚对它们定值的语句
    - **输出结点：**结点对应的变量在基本块出口处活跃
- 从DAG可以知道各个变量最后值和初始值的关系

DAG图的构造

- 为基本块中出现的每个变量建立结点（表示初始值），各变量和相应结点关联
- 顺序扫描各个三地址指令，进行如下处理：

1. 如果指令为**x=y op z**
   - 为这个指令建立结点N，标号为op；
   - N的子结点为y、z当前关联的结点；
   - 令x和N关联；
2. 如果指令为**x=y**；
   - 不建立新结点；
   - 设y关联到N，那么x现在也关联到N
3. 扫描结束后，对于所有在出口处活跃的变量x，将x所关联的结点设置为**输出结点**

例子



- 指令序列
  - a=b+c
  - b=a-d
  - c=b+c
- 过程：
  - 结点b0，c0和d0对应于b，c和d的初始值；它们和相应结点关联；
  - a=b+c：构造第一个加法结点，a与之关联
  - b=a-d：构造减法结点，b与之关联
  - c=b+c：构造第二个加法结点，c与之关联（注意第一个子结点对应于减法结点）

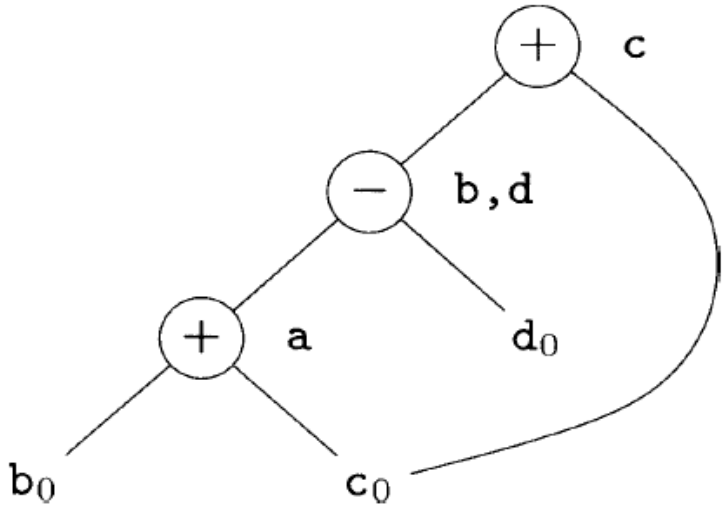
DAG的作用

- DAG图描述了基本块运行时各变量的值（和初始值）之间的关系

- 以DAG为基础，可以对代码进行变换和优化
  - 消除局部公共子表达式
  - 消除死代码
  - 对语句重新排序
  - 重新排序运算分量的顺序

局部公共子表达式
(local common subexpression)

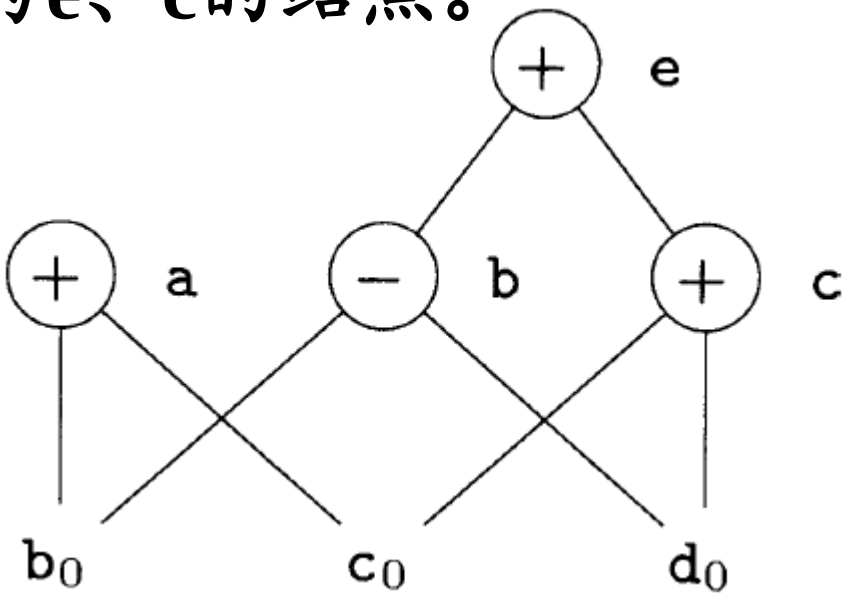
- 局部公共子表达式的发现
  - 建立某个结点M之前，首先检查是否存在一个结点N，它和M具有相同的运算符和子结点（顺序也相同）。
  - 如果存在，则不需要生成新的结点，用N代表M；
- 例如：
  - a=b+c
  - b=a-d
  - c=b+c
  - d=a-d



- 注意：两个b+c并不是公共子表达式

消除死代码(dead code elimination)

- 消除死代码: 在DAG图上消除没有附加活跃变量的根结点（没有父结点的结点）
- 例：如果图中a和b活跃，但c和e不是活跃变量，则可以删除标号为e、c的结点。



代数恒等式 (algebraic identity)

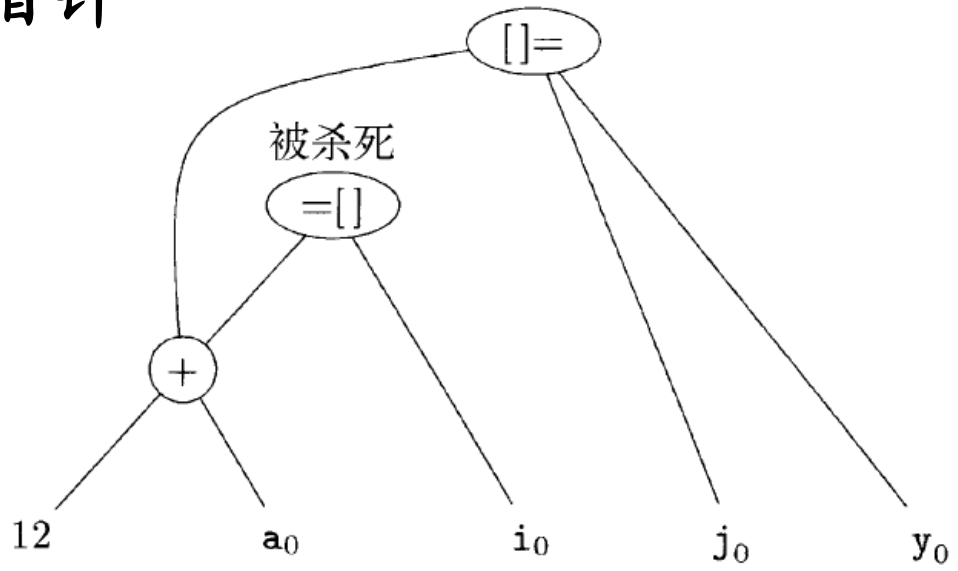
- 消除计算步骤
  - x+0=0+x=x　　x-0=x
  - x*1=1*x=x　　x/1=x
- 降低计算强度
  - x^2=x*x　　2*x=x+x
- 常量合并
  - 2*3.14 可以用 6.28 替换
- 实现这些优化时，只需要在DAG图上寻找特定的模式

数组引用的注意事项

- a[j]可能改变a[i]的值，因此不能和普通的运算符一样构造相应的结点
  - x=a[i]
  - a[j]=y
  - z=a[i]
- 从数组取值的运算x=a[i]对应于=[]的结点，x作为这个结点的标号之一；
- 对数组赋值的运算a[j]=y对应于[]=的结点；没有关联的变量、且杀死所有依赖于a的变量；

数组引用的DAG的例子

- 设a是数组，b是指针
  - b=12+a
  - x=b[i]
  - b[j]=y



- 注意：a是被杀死的结点的孙结点
- 一个结点被杀死，意味着它不能被重复使用
  - 考虑：如果后面有指令m=b[i]？

指针赋值/过程调用

- 通过指针进行取值/赋值：x=*p; *q=y。最保守的估计：
  - x使用了任意变量，因此无法消除死代码
  - 而*q=y对任意变量赋值，因此杀死了全部其他结点

- 可以通过（全局/局部）指针分析部分解决这个问题；
- 过程调用也类似，必须安全（保守）地加以假设
  - 使用了可访问范围内的所有变量
  - 修改了可访问范围内的所有变量

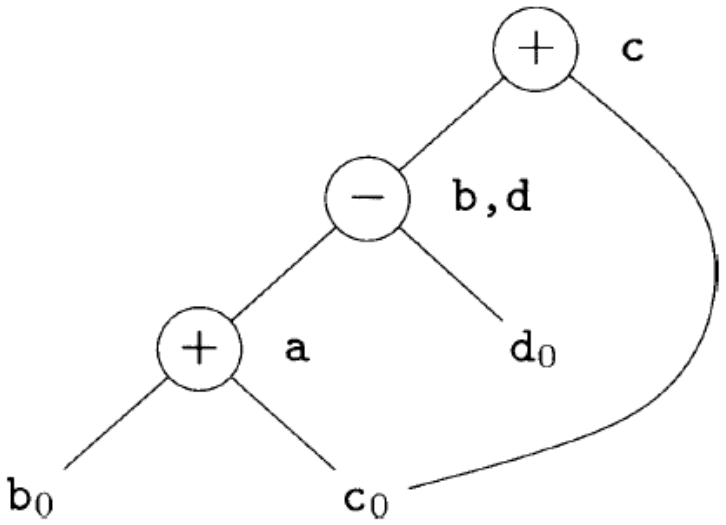
从DAG到基本块

- 重构的方法
  - 每个结点构造一个三地址语句，计算对应的值
  - 如果结点有多个关联的变量，则需要用复制语句进行赋值
  - 结果应该尽量赋给一个活跃的变量

重组基本块的例子

- 根据DAG构造时要考虑结点的顺序
  - 这里假设b和d在出口处都活跃

- a=b+c
- d=a-d
- b=d
- c=d+c



重组的规则

- 重组时应该注意求值的顺序
  - 指令的顺序必须遵守DAG中结点的顺序
  - 对数组的赋值必须跟在所有原来在它之前的赋值/求值操作之后
  - 对数组元素的求值必须跟在所有原来在它之前的赋值指令之后
  - 对变量的使用必须跟在所有原来在它之前的过程调用和指针间接赋值之后
  - 任何过程调用或者指针间接赋值必须跟在原来在它之前的变量求值之后
- 需要保证：如果两个指令之间可能相互影响，那么他们的顺序就不应该改变。

主要内容

- 目标代码生成概述
- 目标计算机模型
- 基本块和流图
- 基本块的优化
- 目标代码生成方法
- 窥孔优化
- 寄存器分配

目标代码生成器

- 根据三地址指令序列生成机器指令
  - 假设：每个三地址指令只有一个对应的机器指令
  - 有一组寄存器用于计算基本块内部的值
- 主要的目标是尽量减少加载和保存指令，即最大限度利用寄存器
- 寄存器的使用方法
  - 执行运算时，运算分量必须放在寄存器中
  - 用于临时变量
  - 存放全局的值
  - 进行运行时刻管理

算法的基本思想和数据结构

- 依次考虑各三地址指令，尽可能把值保留在寄存器中，以减少寄存器/内存之间的数据交换
- 为一个三地址指令生成机器指令时
  - 只有当运算分量不在寄存器中时，才从内存载入
  - 尽量保证只有当寄存器中的值不被使用时，才把它覆盖掉
- 数据结构：记录各个值对应的位置
  - 寄存器描述符：跟踪各个寄存器都存放了哪些变量的当前值
  - 地址描述符：某个变量的当前值存放在哪个或哪些位置(包括内存位置和寄存器）上

代码生成算法（1）

- 寄存器选择函数：**getReg(I)**
  - 根据寄存器描述符和地址描述符、数据流信息，为三地址指令I选择最佳的寄存器
  - 得到的机器指令的质量依赖于getReg函数选取寄存器的算法

- 代码生成算法会依次处理每个三地址指令

代码生成算法（2）

- x=y+z
  - 调用getReg(x=y+z)，为x,y,z选择寄存器Rx,Ry,Rz
  - 查Ry的寄存器描述符，如果y不在Ry中则生成指令：LD Ry y’　　(y’表示存放y值的当前位置)
  - 类似地，确定是否生成 LD Rz,z’
  - 生成指令 ADD Rx, Ry, Rz
- 复制语句：x=y
  - getReg(x=y) 总是为x和y选择相同的寄存器
  - 如果y不在Ry中，生成机器指令 LD Ry, y
- 基本块的结束
  - 如果变量x在出口处活跃，且x不在内存中，则生成指令 ST x, Rx

代码生成算法（3）

- 代码生成的同时更新寄存器和地址描述符
- 处理普通指令时生成 **LD R, x**
  - R的寄存器描述符：只包含x
  - x的地址描述符：R作为新位置加入到x的位置集合中
  - 从任何不同于x的变量的地址描述符中删除R
- **ST x, R**
  - 修改x的地址描述符，包含自己的内存位置

代码生成算法（4）

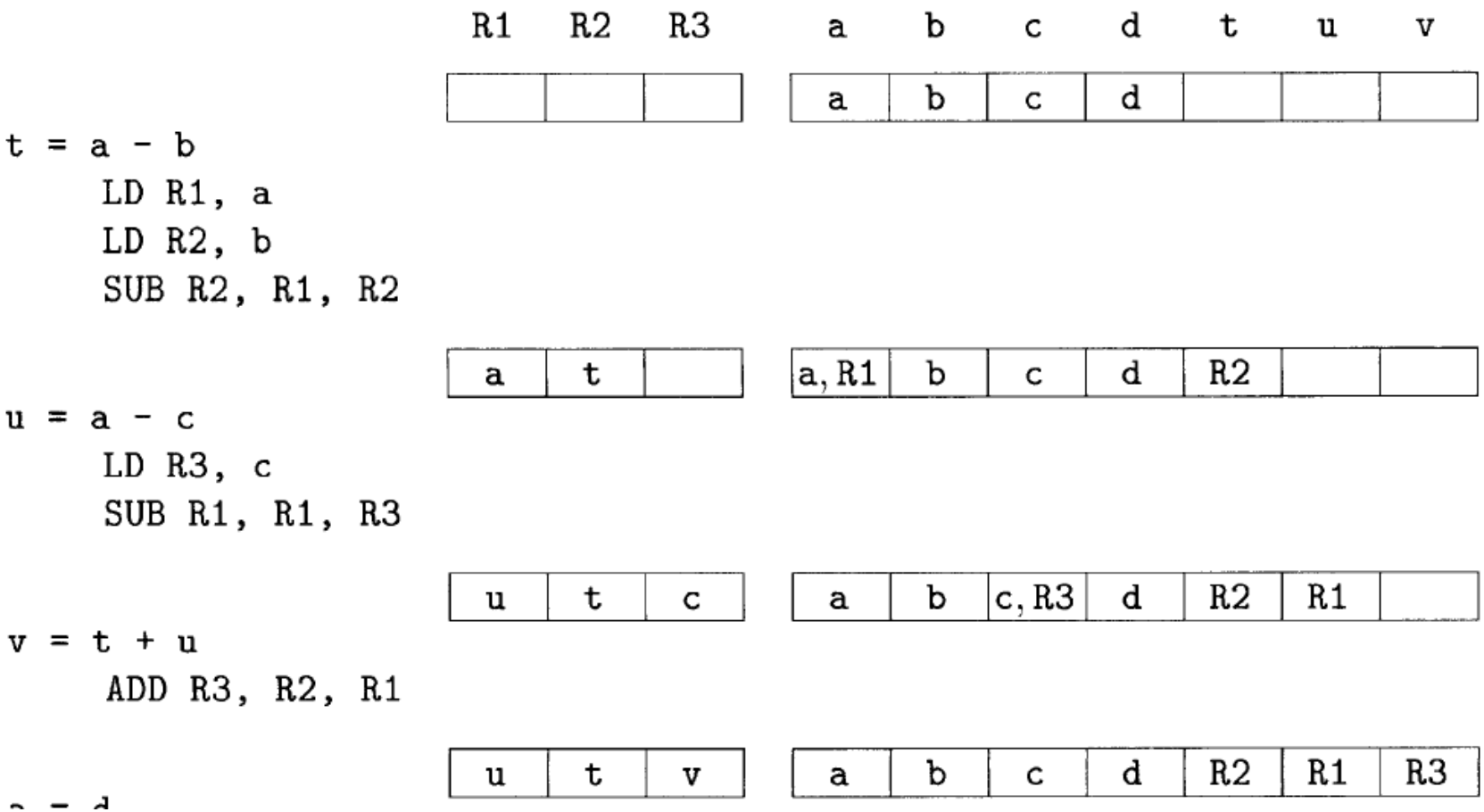
- ADD Rx, Ry, Rz
  - Rx的寄存器描述符只包含 x
  - x 的地址描述符只包含Rx (不包含x的内存位置!)
  - 从任何不同于x的变量的地址描述符中删除Rx。
- 处理x=y时，
  - 如果生成LD Ry y，按照LD规则处理；
  - 把x加入到Ry的寄存器描述符中（Ry存放了x和y的当前值）；
  - 修改x的地址描述符，使它只包含Ry。

例子（1）

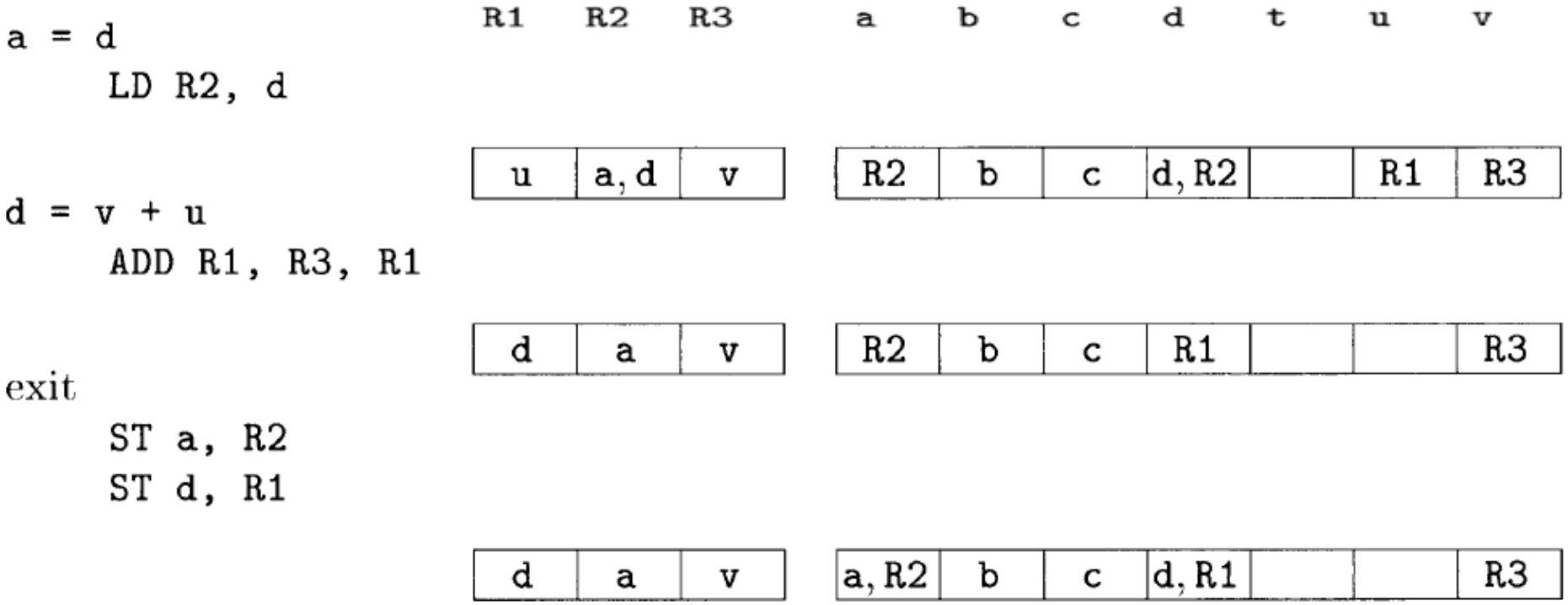
- a、b、c、d在出口处活跃
- t、u、v是局部临时变量

```
t = a - b
u = a – c
v = t + u
a = d
d = v + u
```

例子（2）


R1 R2 R3
a b c d t u v
a b c d
t = a - b
LD R1, a
LD R2, b
SUB R2, R1, R2
a t
a, R1 b c d R2
u = a - c
LD R3, c
SUB R1, R1, R3
u t c
a b c, R3 d R2 R1
v = t + u
ADD R3, R2, R1
u t v
a b c d R2 R1 R3

例子（3）


R1 R2 R3 a b c d t u v
a = d
LD R2, d
u | a,d | v
R2 | b | c | d,R2 | | R1 | R3
d = v + u
ADD R1, R3, R1
d | a | v
R2 | b | c | R1 | | | R3
exit
ST a, R2
ST d, R1
d | a | v
a,R2 | b | c | d,R1 | | | R3

getReg函数（1）

- getReg的目标：减少LD/ST指令
  - 以 x = y + z 为例
- 任务：为运算分量(y,z)和结果(x)分配寄存器
- 为运算分量分配寄存器
  - 如果已经在某个寄存器中，不需要进行处理，选择这个寄存器
  - 如果不在寄存器中，且有空闲寄存器，选择一个空闲寄存器
  - 如果不在寄存器中，且没有空闲寄存器？

getReg函数（2）

- 为运算分量(y, z)分配寄存器
  - …如果不在寄存器中，且没有空闲寄存器
  - 寻找一个寄存器R，且R的寄存器描述符表示v在R中
    - 如果v的地址描述符表明还可以在别的地方找到v √
    - v就是x（即结果），且x不是运算分量之一 √
    - 如果v在此之后不会被使用 √
    - 否则：生成保存指令ST v R（溢出）；并修改v的地址描述符；如果R中存放了多个变量的值，那么需要生成多条ST指令

getReg函数（3）

- 为x选择寄存器Rx的方法基本上和上面要把y从内存LD时一样；但是
  - 只存放x的值的寄存器总是可接受的；
  - 如果y在指令I之后不再使用，且Ry仅仅保存了y的值，那么Ry同时也可以作为Rx；

- 处理x=y时，总是让Rx=Ry。

主要内容

- 目标代码生成概述
- 目标计算机模型
- 基本块和流图
- 基本块的优化
- 目标代码生成方法
- **窥孔优化**
- **寄存器分配**

窥孔优化 (peephole optimization)

- 使用一个滑动窗口（窥孔, peephole）来检查目标指令，在窥孔内实现优化
  - 冗余指令消除
  - 控制流优化
  - 代数简化
  - 机器特有指令的使用

冗余指令

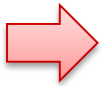
- 多余的LD/ST指令
  - LD R0 a
  - ST a R0
  - 且没有指令跳转到第二条指令处

- 跳转代码
  - if debug==1 goto L1；goto L2
  - if debug!=1 goto L2;
  - 如果已知debug一定是0，那么替换成为goto L2

控制流优化

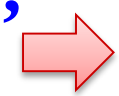
- 级联跳转

```
goto   L1; … …;
L1:  goto L2
```



```
goto L2; … …;
goto L2
```

```
if a<b goto L1; … … ;
L1: goto L2
```



```
if a<b goto L2; … … ;
L1: goto L2
```

代数化简/强度消减

- 应用代数恒等式进行优化
  - 消除x=x+0　x=x*1
  - 用 x*x 替换 x^2
- 使用机器特有指令
  - INC，DEC，…
  - 代替：x=x+1　x=x-1

寄存器分配和指派

- 目的：有效利用寄存器
- 简单的基本方法：把特定类型的值分配给特定的寄存器
  - 数组基地址指派给一组寄存器
  - 算术计算分配给一组寄存器
  - 栈顶指针分配一个寄存器
  - ……
- 缺点：寄存器的使用效率较低

全局寄存器分配

- 在循环中频繁使用的值存放在固定寄存器
- 分配固定多个寄存器来存放内部循环中最活跃的值
- 可以通过使用计数的方法来估算把一个变量放到寄存器中会带来多大好处，然后根据这个估算来分配寄存器

寄存器分配的问题

- 有很多的值可以存放到寄存器中
  - 编译器生成的临时变量的值
  - 局部变量中保存的值
  - 大的常数
  - 数组元素或对象域中的值

- 但是寄存器的个数是有限的
  - 通常只有16个integer和16个floating point寄存器
  - 而且有些寄存器还被指定用于特定目的
    - 例如rsp, rbp

寄存器分配的基本方法(1)

- 为提高程序性能，最理想的方式是把每一个临时变量都放到寄存器中
- 但是，我们并没有足够的寄存器
- 解决思路：
  - 当一个临时变量不再活跃（dead）时，可以重复使用它的寄存器
  - 两个同时活跃的临时变量不能使用同一个寄存器

寄存器分配的基本方法(2)

- 当一个临时变量不再活跃（dead）时，可以重复使用它的寄存器
- 例子：

```
a = c + d
e = a + b
f = e - 1
```

(假设a和e在使用之后不在活跃)

- 临时变量a, e, f可以使用同一个寄存器

```
r1 = c + d
r1 = r1 + b
r1= r1 - 1
```

寄存器分配的基本方法(3)

- 两个同时活跃的临时变量不能使用同一个寄存器
- 例子：

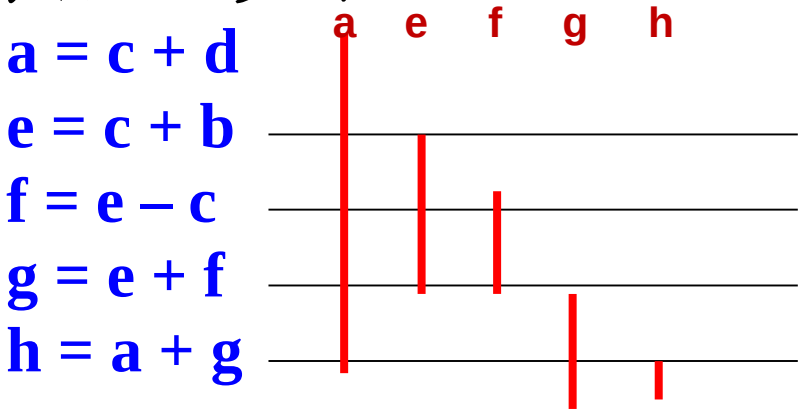
```
a = c + d
e = a + b
f = e - a
```

- 临时变量e和a不可以使用同一个寄存器

```
r1 = c + d
r2 = r1 + b
r1= r2 - r1
```

寄存器分配的基本方法(4)

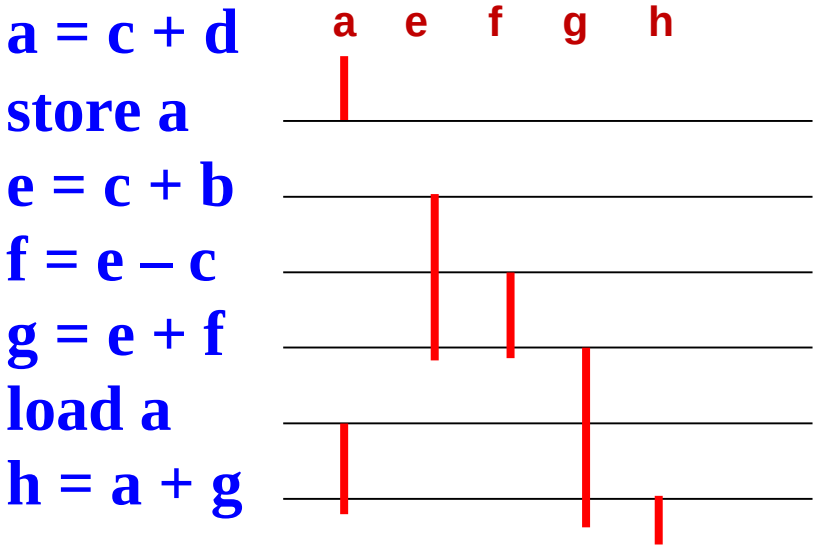
- 如果寄存器不够用怎么办？



2个寄存器
无法分配！

(假设在最后只有g和h是活跃的)

- 可以通过把变量保存到内存来拆分活跃区间



图着色法的寄存器分配方法

- 1971年John Cokes提出：全局寄存器分配可以视为一种图着色问题

- 图着色问题的简单描述
  - 已知一个图G和m>0种颜色，在只准使用这m种颜色对G的结点着色的情况下，是否能使图中任何相邻的两个结点都具有不同的颜色呢？这个问题称为m-着色判定问题
  - m-着色的最优化问题是求可对图G着色的最小整数m。这个整数称为图G的色数(Chromatic Number)

图着色的寄存器分配

- 基本思想：
  - 构造以变量(活跃范围)为顶点的干扰图 (interference graph)
  - 图的边：两个结点之间不能使用同一个寄存器（称为相互干扰）则用边连起来
- 采用图论中的图着色算法对干扰图G进行着色
  - 如果两个顶点之间存在一条边，则它们不能使用相同的颜色
  - 利用图着色的算法给干扰图着色之后，就可以给相同颜色的顶点分配同一个寄存器
  - 着色所需不同颜色的种类就是寄存器分配所需的个数

图着色的寄存器分配

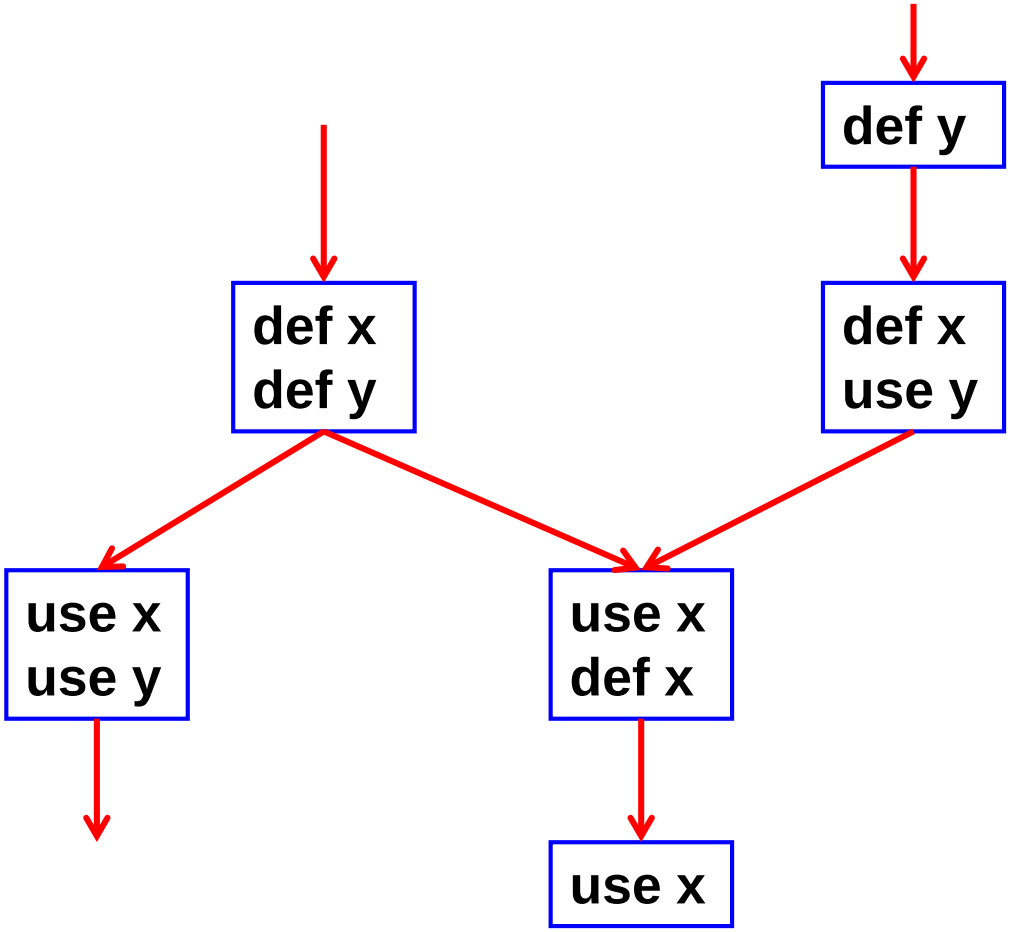
- 基本步骤
  - 确定每个值的活跃范围（web）
  - 确定重叠的活跃范围（干扰）
  - 计算把每个web保存到寄存器中的代价 (溢出代价)
  - 确定哪些web会分配到寄存器（分配）
  - 如有需要，对web进行拆分（溢出和拆分）
  - 给web指派实际的寄存器（指派）
  - 生成包含溢出的目标代码（代码生成）

Web的构造

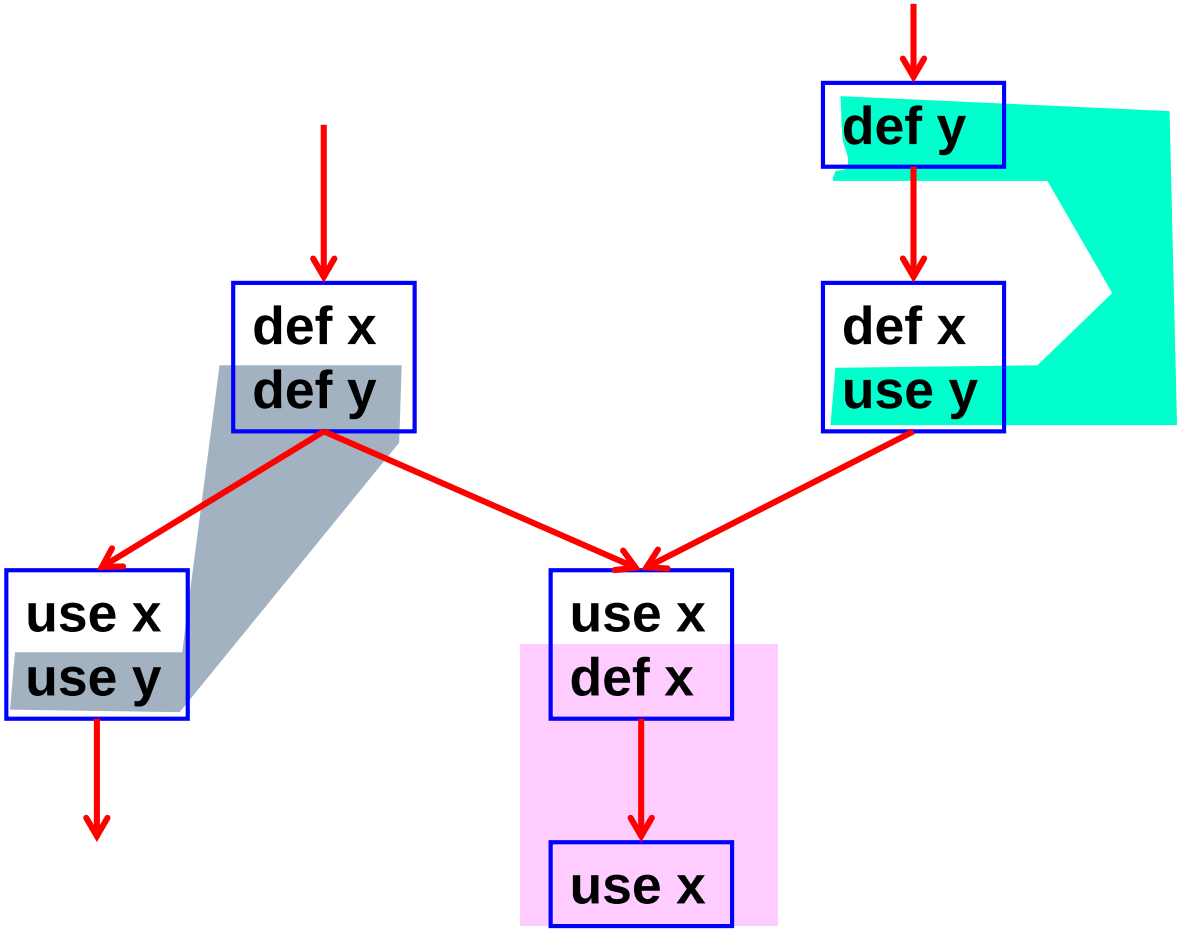
- 构造web的基础是def-use chain（DU链）
  - DU链可以把值的定义连接到所有可达的使用

- 把def和use放到同一个web中的条件：
  - 一个def和所有可达的use都必须要放到同一个web中
  - 到达一个use的所有def都必须要放到同一个web中

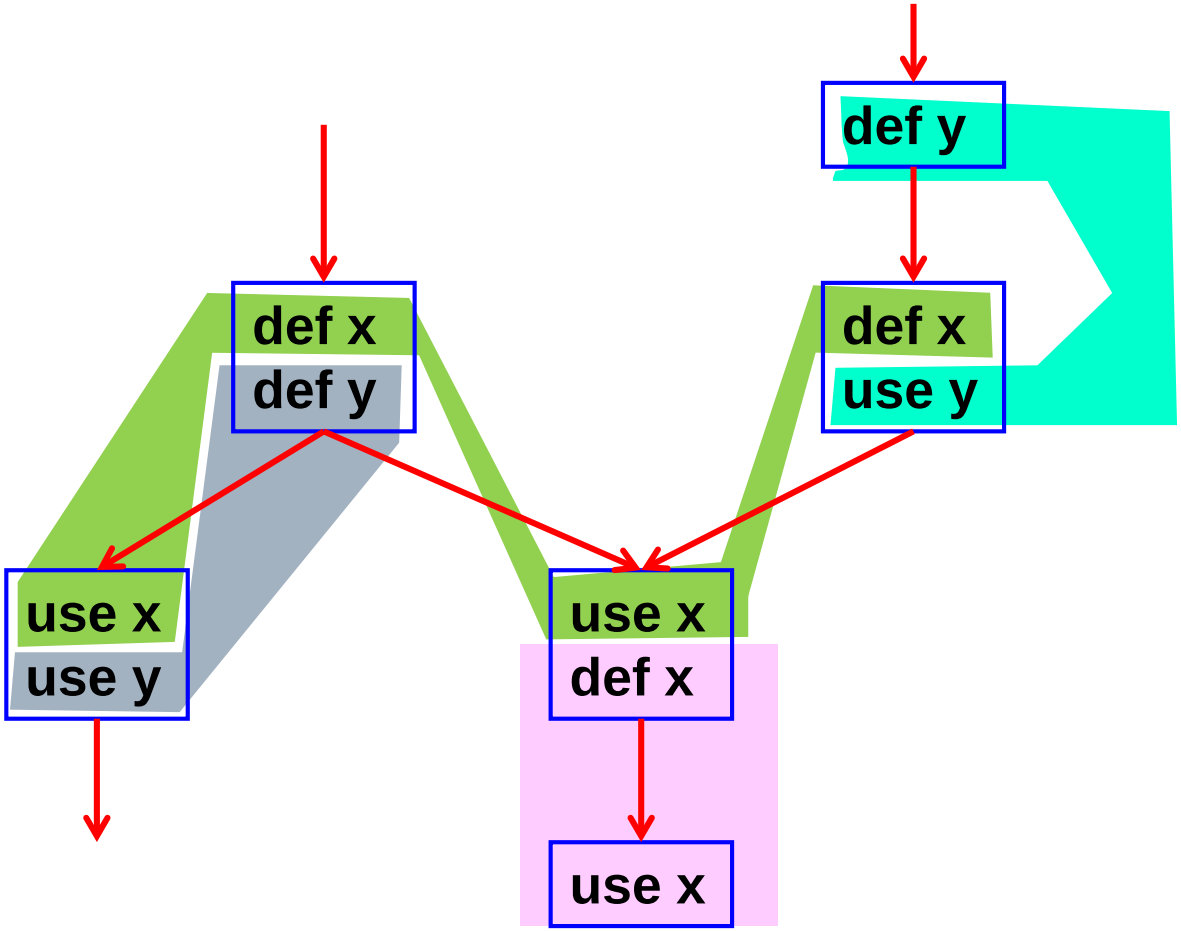
Web 示例



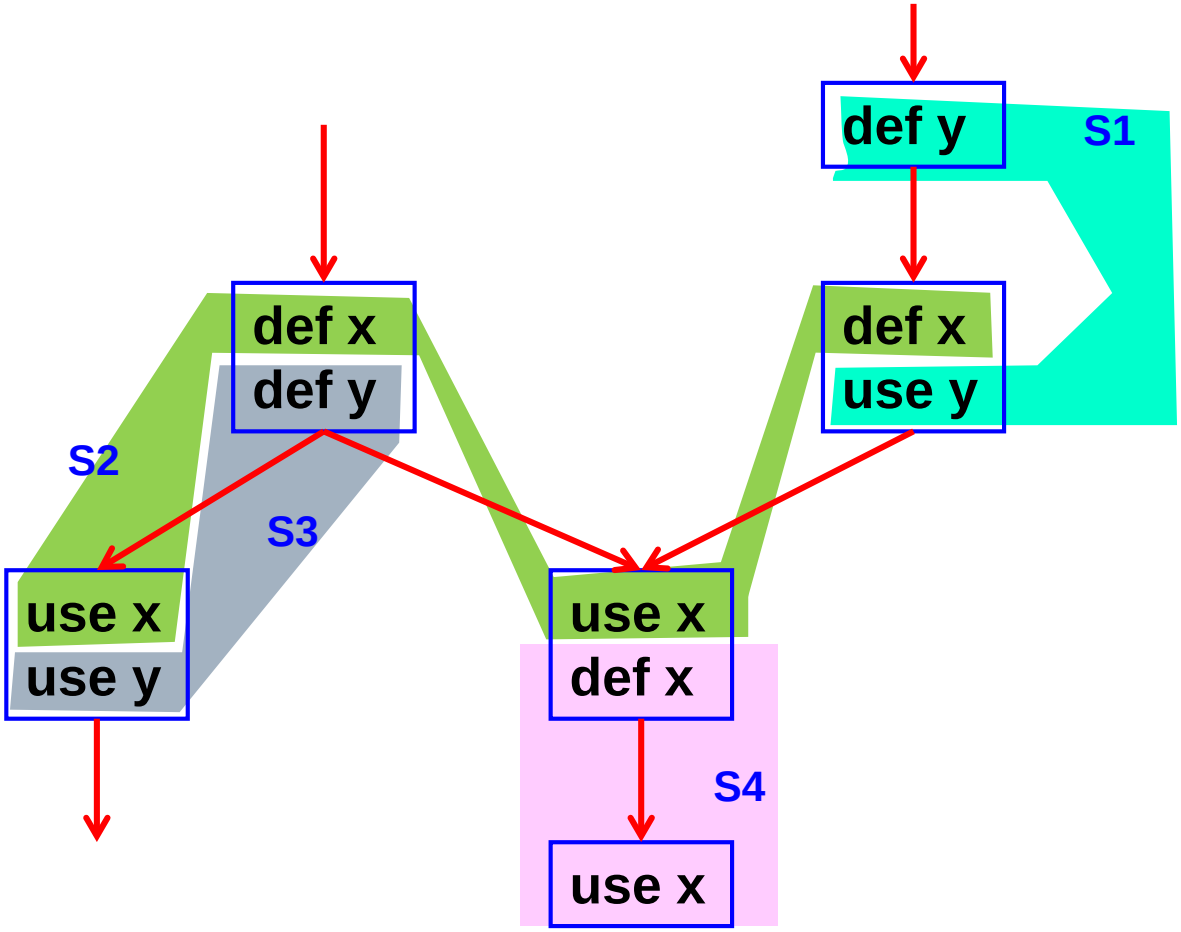
Web 示例



Web 示例



Web 示例



Web的使用

- Web是寄存器分配的单元

- 如果为一个web分配了某个寄存器R，那么
  - 该web中所有def的计算结果都放到R中
  - 该web中所有use都会从R中读取

- 如果为一个web分配了某个内存地址M
  - 该web中所有def的计算结果都放到M中
  - 该web中所有use都会从M中读取

凸集和活跃范围

- 凸集（convex set）的定义：一个集合S，任取A, B∈S，如果任意C位于从A到B的路径上，都有C∈S，则S为凸集。

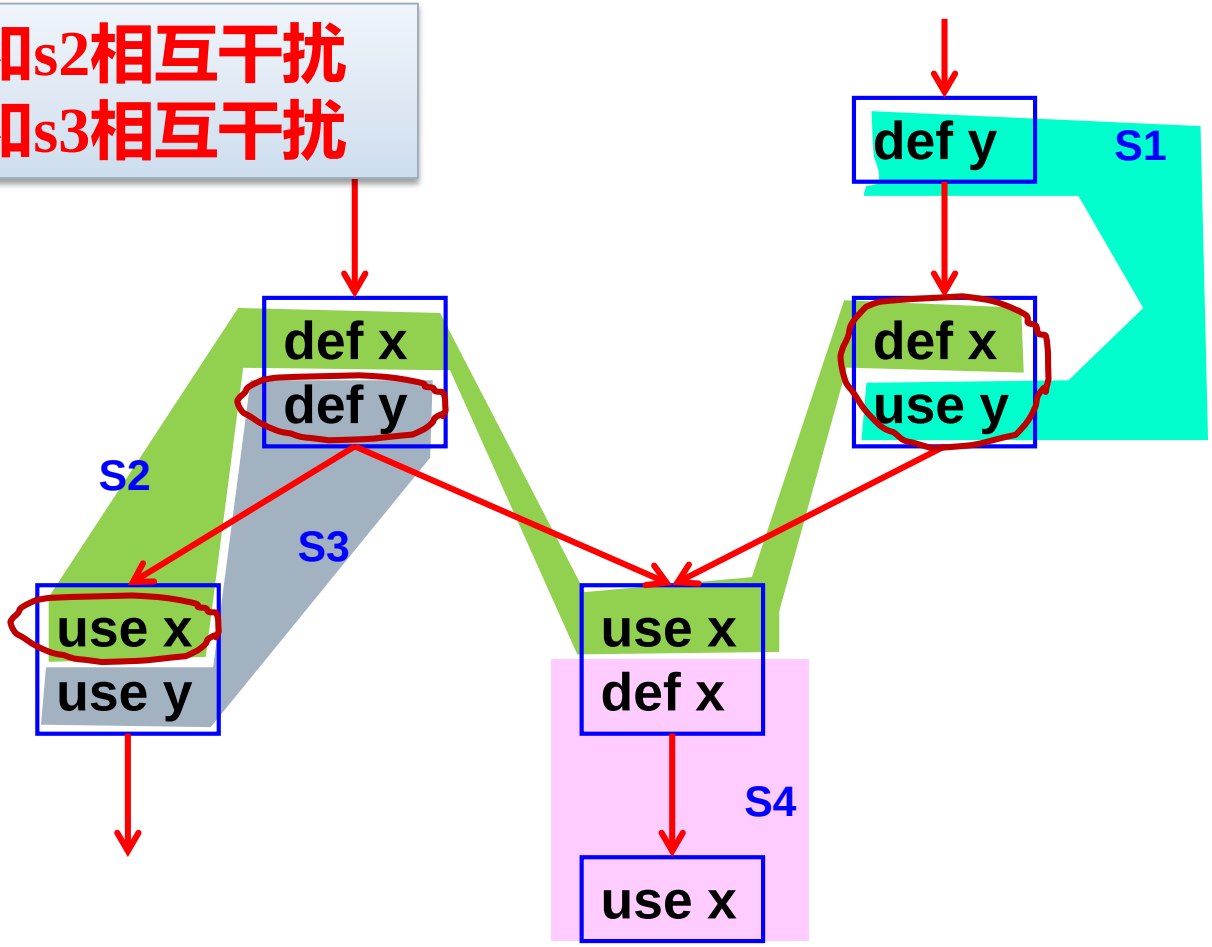
- Web的活跃范围（live range）：
  - 包含一个web中的所有def和use的指令构成的最小凸集。
  - 即：一个web中的值的活跃范围

干扰关系（interference）

- 如果两个web的活跃范围之间有重叠，则认为他们之间 相互干扰（interfere）
  - 如果两个web相互干扰，则不能给它们对应的值分配相同的寄存器（或者相同的内存地址）
  - 如果两个web不相互干扰，则可以把它们的值保存到相同的寄存器（或内存地址）中

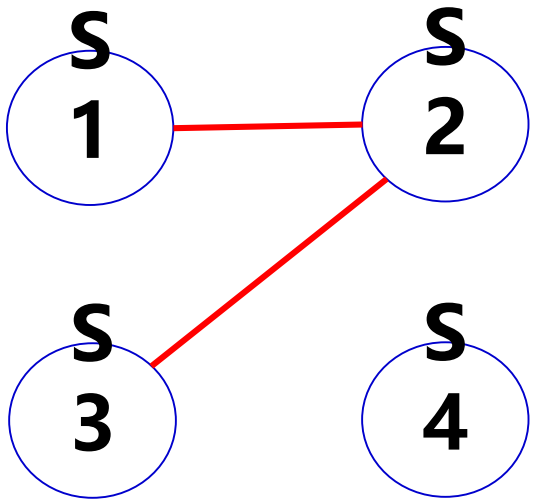
干扰的例子

Web s1和s2相互干扰
Web s2和s3相互干扰



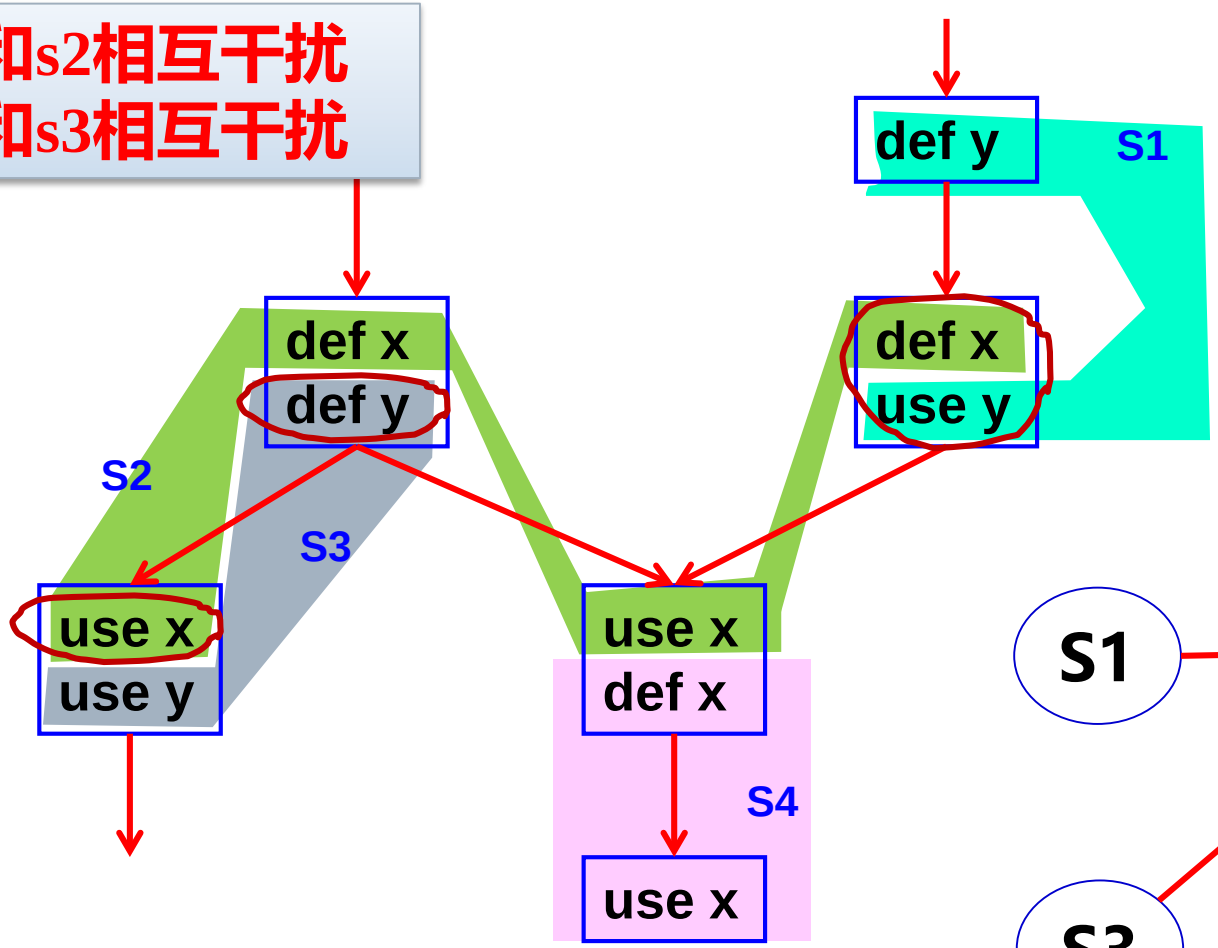
干扰图

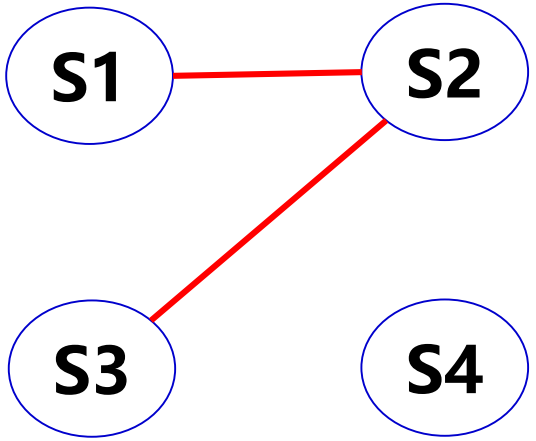
- 用干扰图（interference graph）来表示web和它们之间的干扰关系
  - 每个结点代表一个web
  - 每条边代表两个结点（web）之间相互干扰



干扰图的例子

Web s1和s2相互干扰
Web s2和s3相互干扰



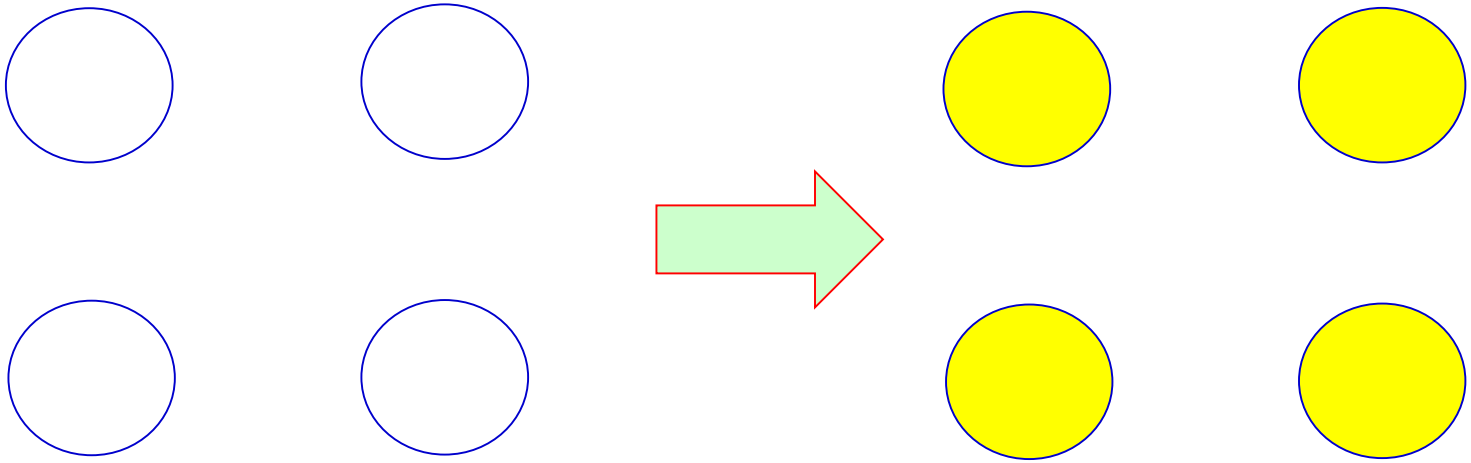


干扰图

基于干扰图的着色分配寄存器

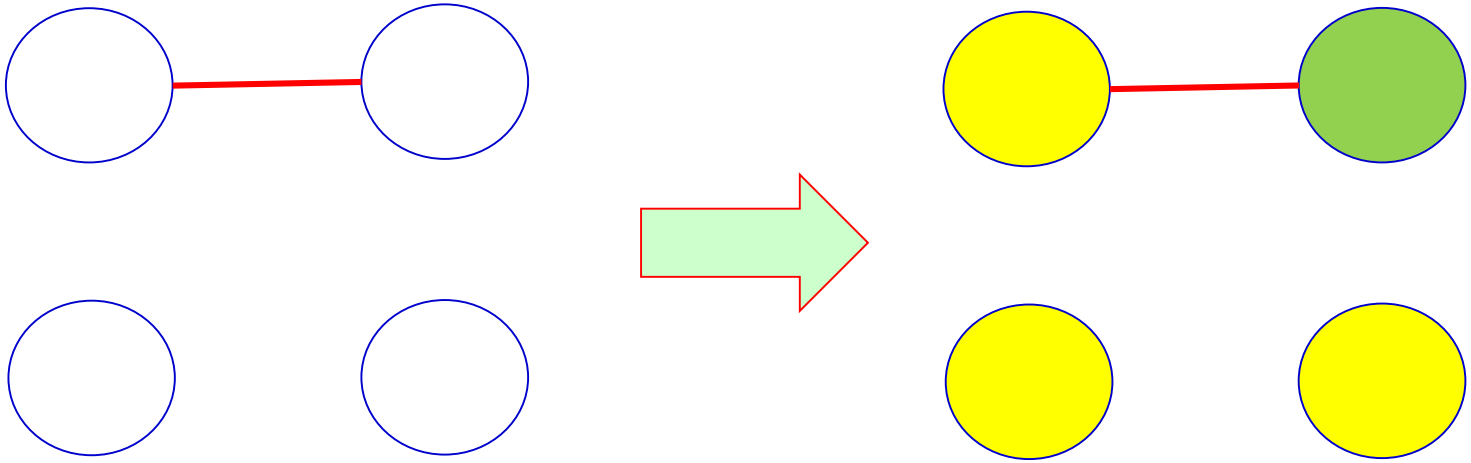
- 为干扰图中的每个结点指派一种颜色
- 两个直接相连的结点必须指派不同的颜色
- 着色完成之后，为每种颜色对应的结点（web）中的值都分配相同的寄存器

图着色的例子



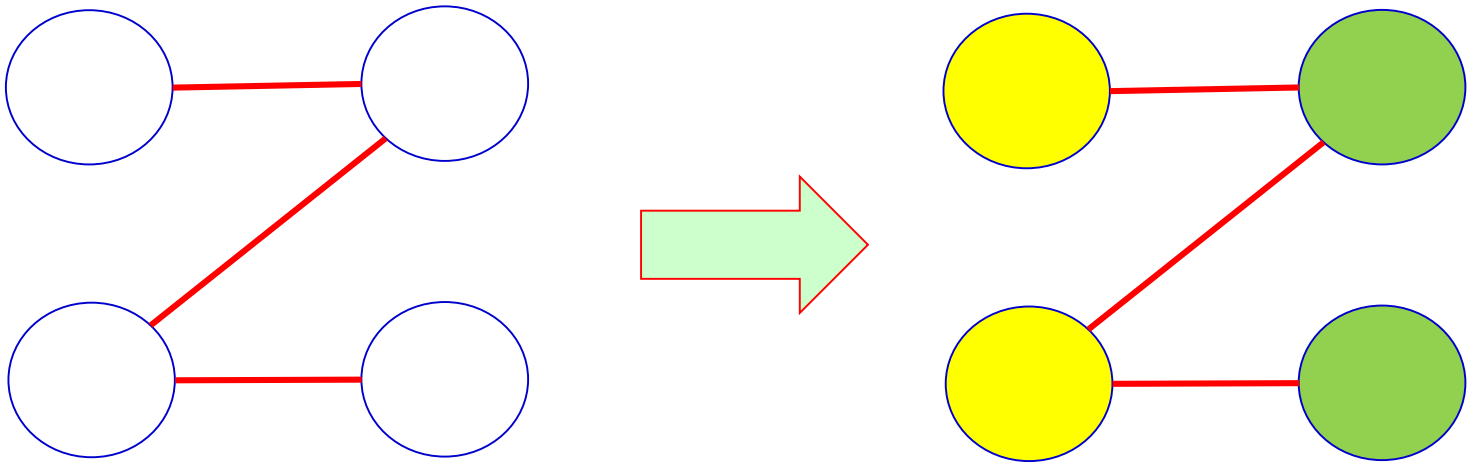
- 1种颜色

图着色的例子



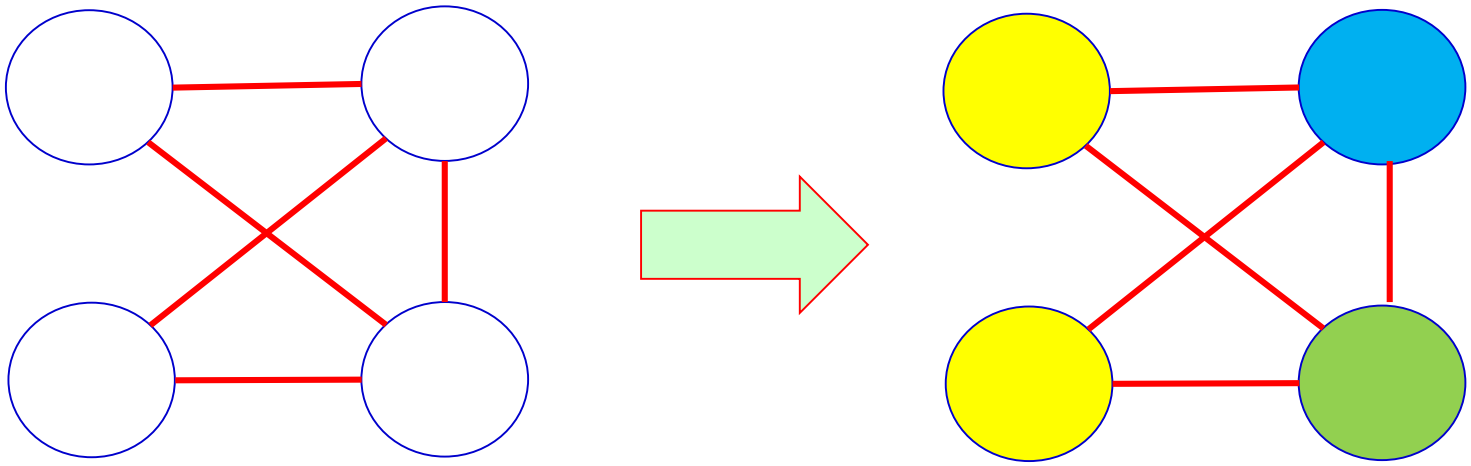
- 2种颜色

图着色的例子



- 2种颜色

图着色的例子



- 3种颜色

图着色法的启发式规则

- 对干扰图使用N种颜色进行着色

- 如果干扰图中每个结点的度数都＜N
  - 该图总是可以被着色（在对所有其他结点着色之后，至少存在一种颜色对这个结点着色）

- 如果有结点的度数＞N
  - 也可能可以使用N种颜色着色

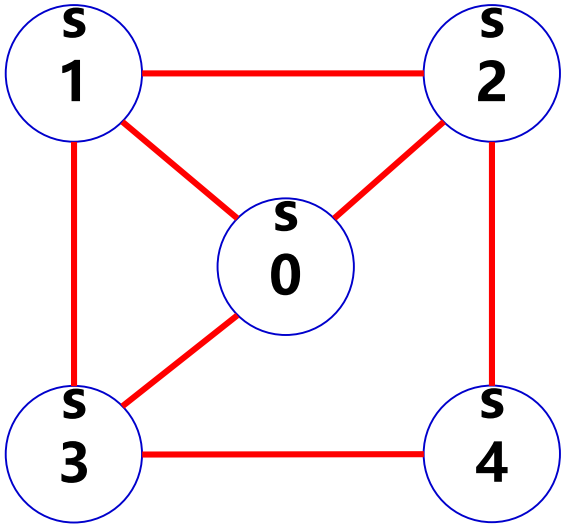
图着色的基本算法

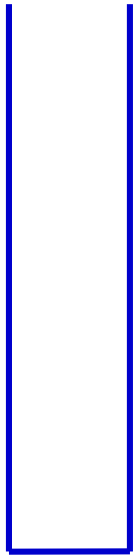
- 查找图的N-可着色方案
  - 最早由Kempe提出[1879]

1. 寻找度数＜N的结点，从图中删除
   - 把该结点压到一个栈中
2. 如果所有结点的度数都＞=N
   - 找到一个溢出（spill）结点（不对它着色）
   - 删除该结点
3. 当图为空的时候
   - 从栈顶依次弹出结点
   - 选择它的邻居都没有使用的颜色进行着色。

着色过程示例

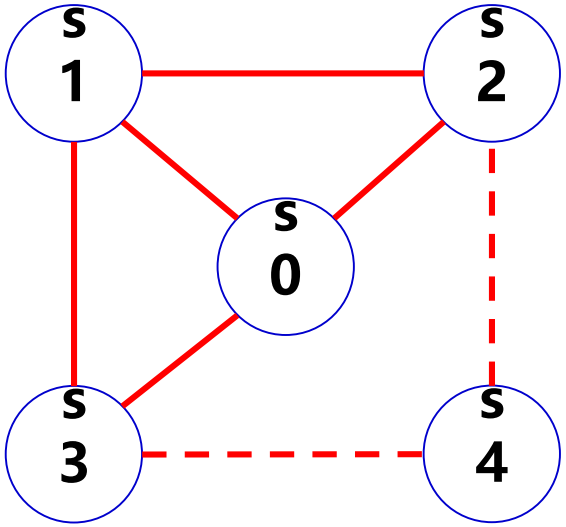
N=3

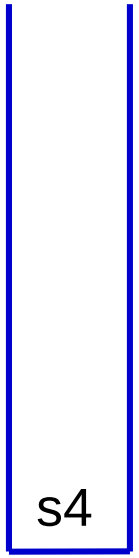




着色过程示例

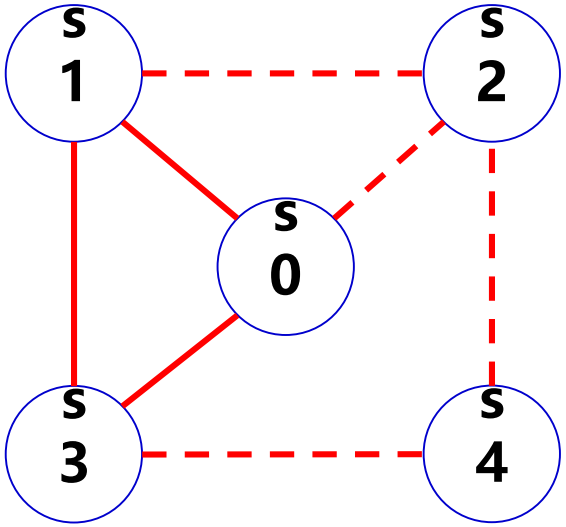
N=3

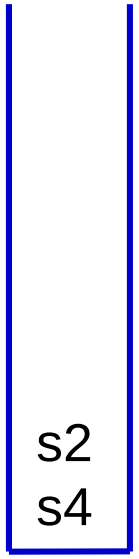




着色过程示例

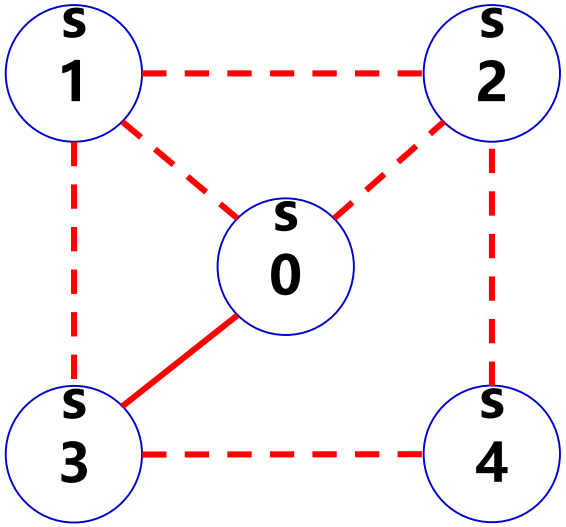
N=3

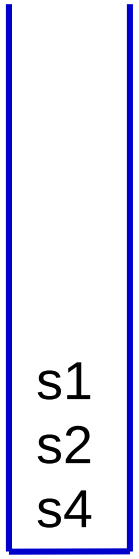




着色过程示例

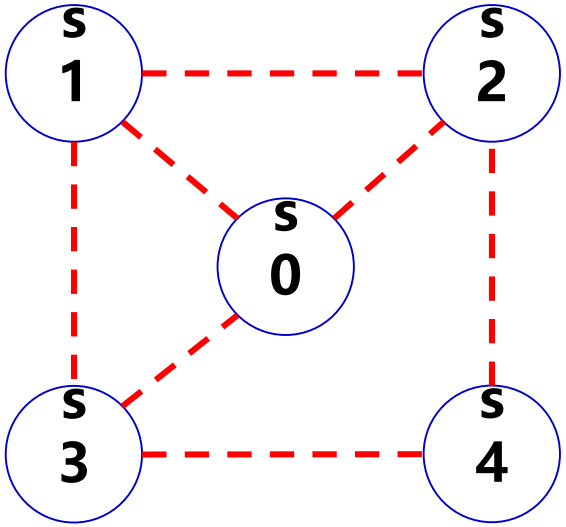
N=3

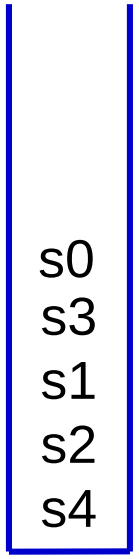




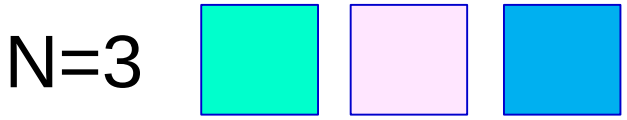
着色过程示例

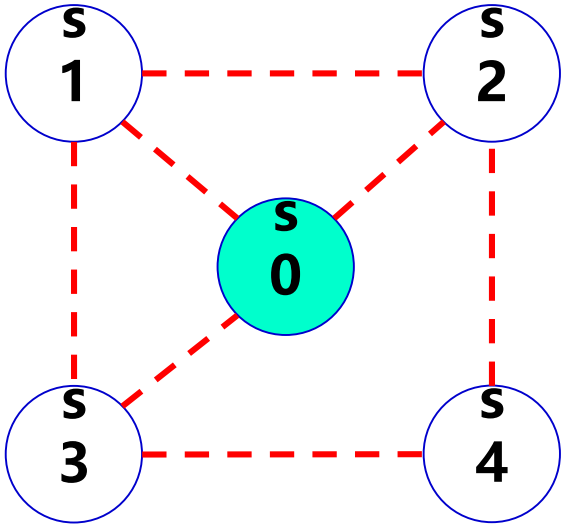
N=3

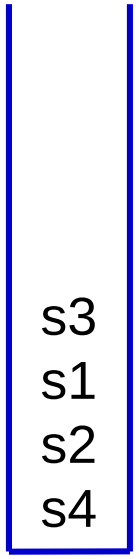




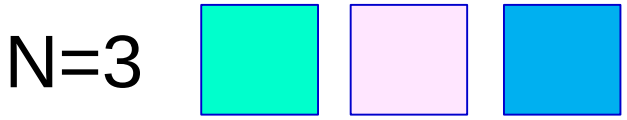
着色过程示例

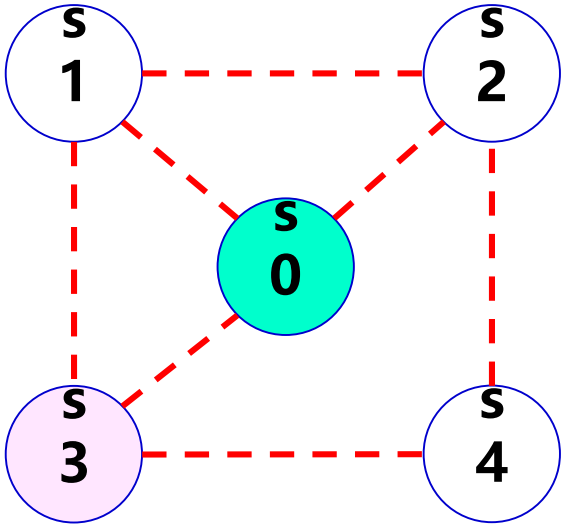


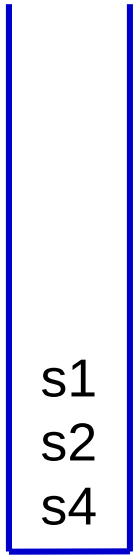




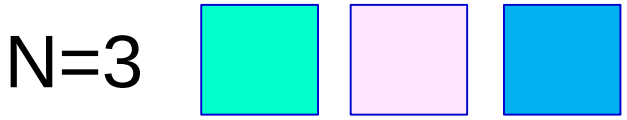
着色过程示例

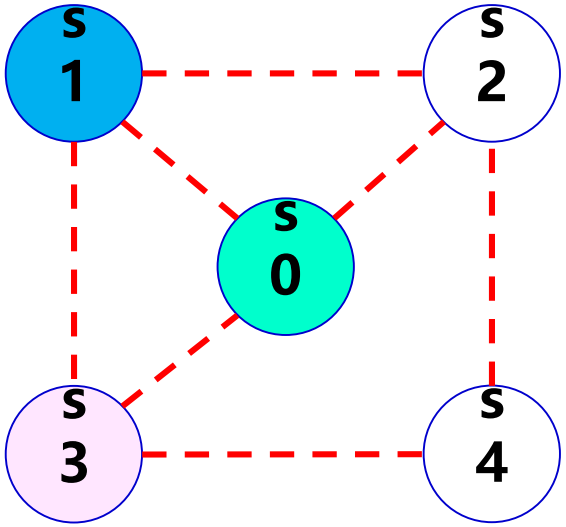


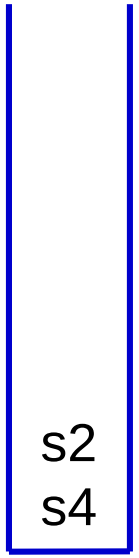




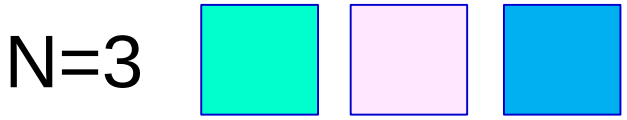
着色过程示例

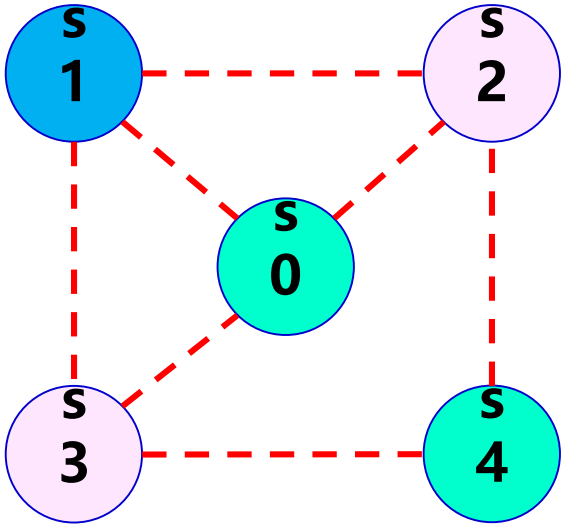


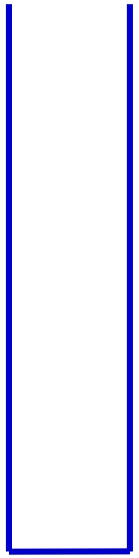




着色过程示例



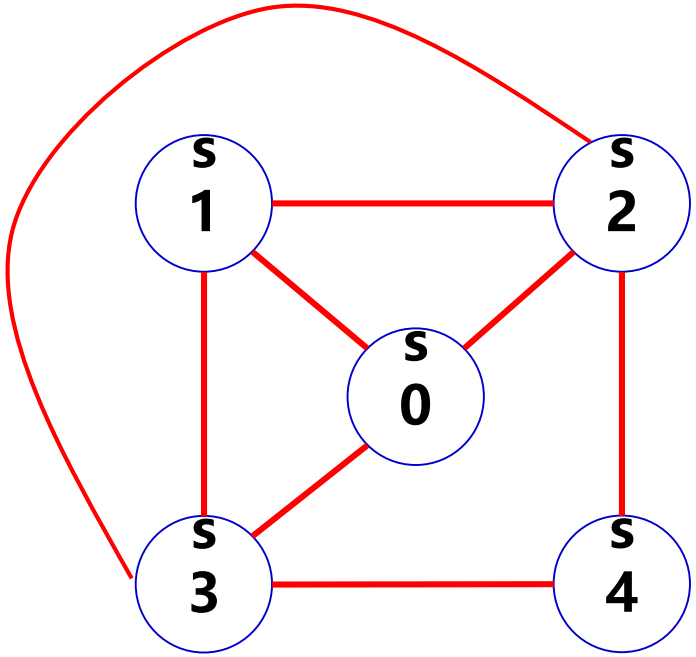


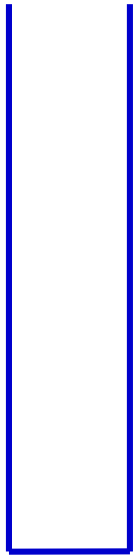


着色成功！

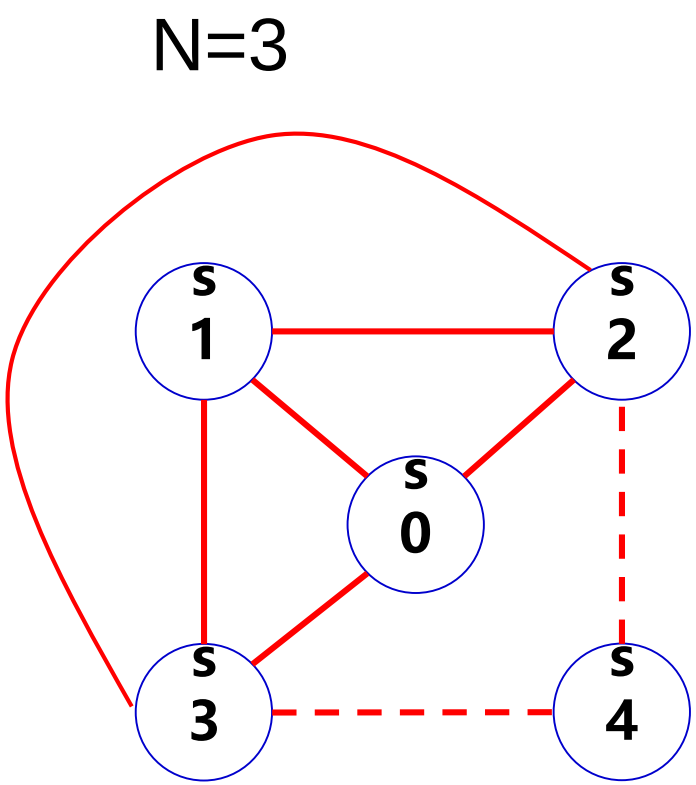
另一个图着色的例子

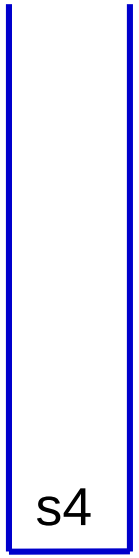
N=3



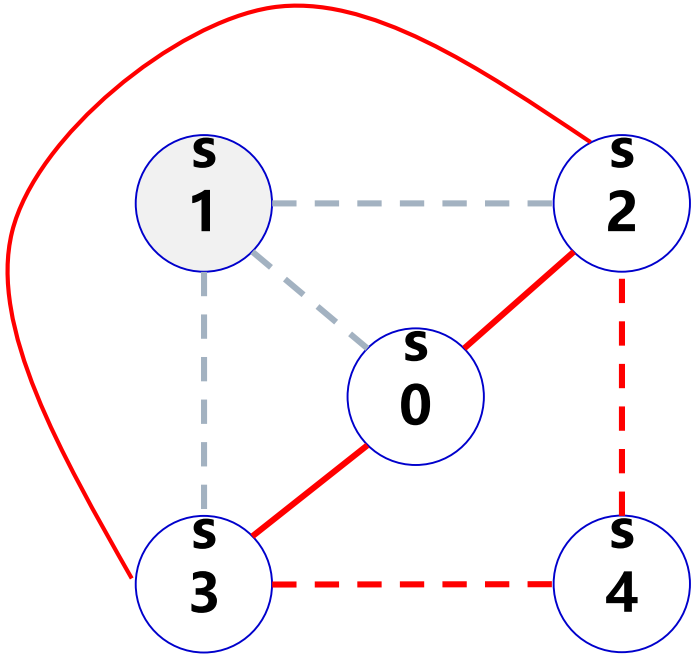


另一个图着色的例子

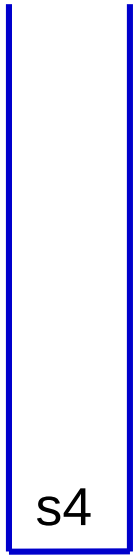




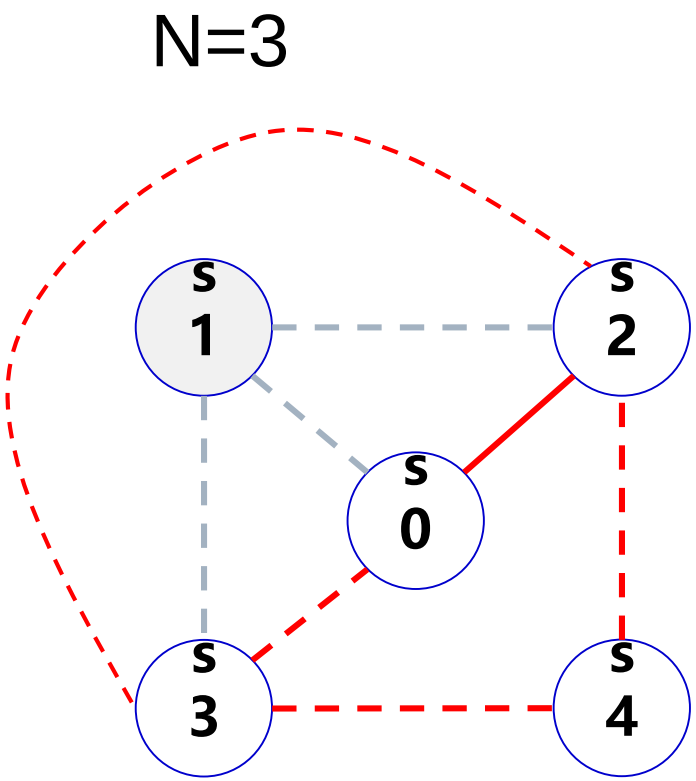
另一个图着色的例子

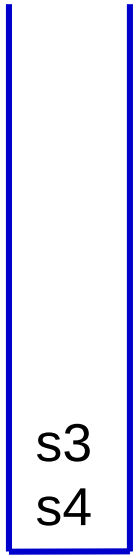


溢出s1

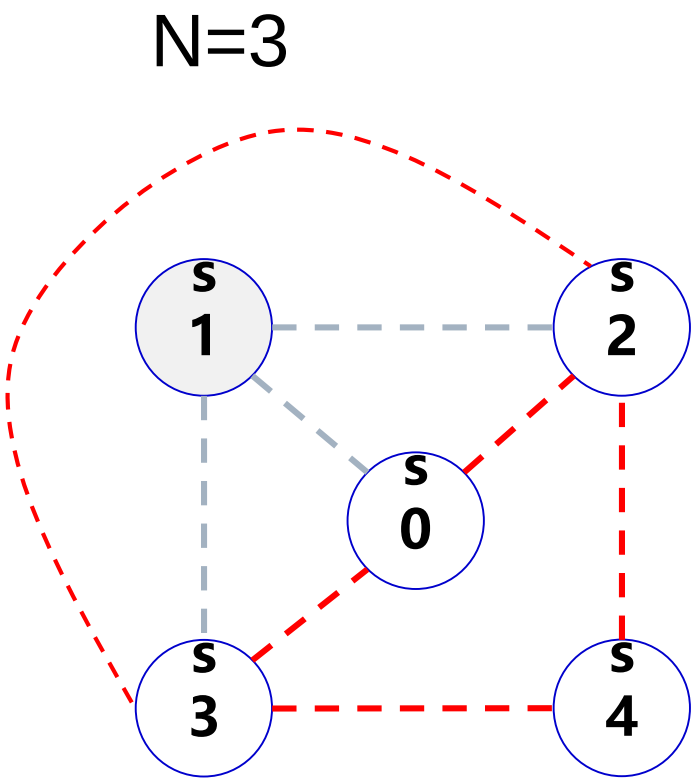


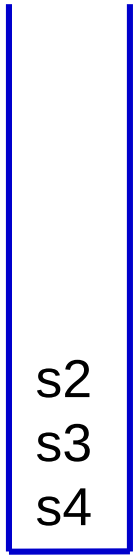
另一个图着色的例子



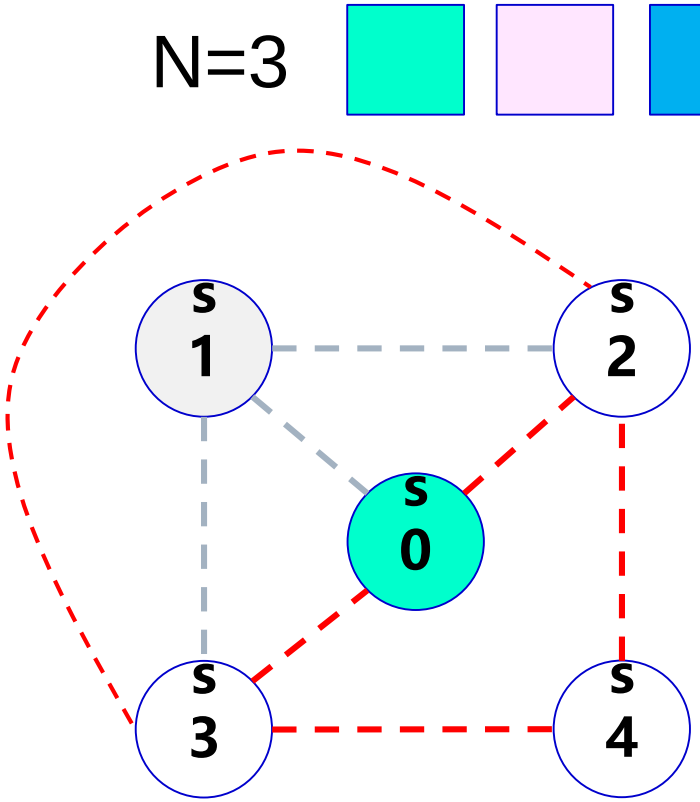


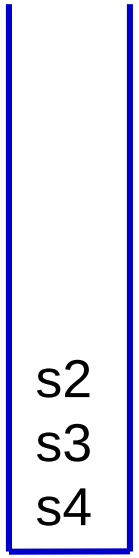
另一个图着色的例子



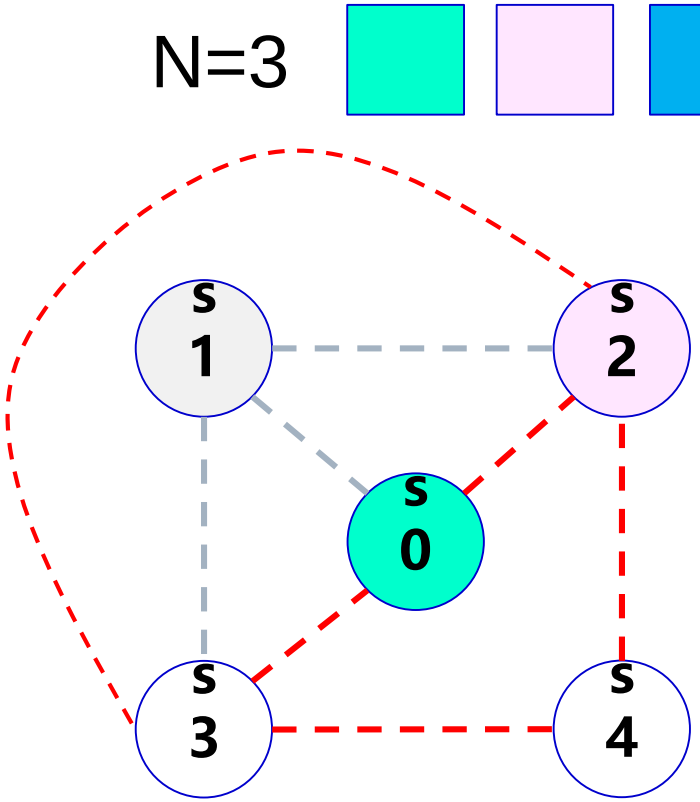


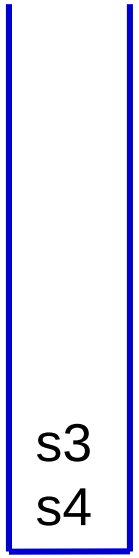
另一个图着色的例子



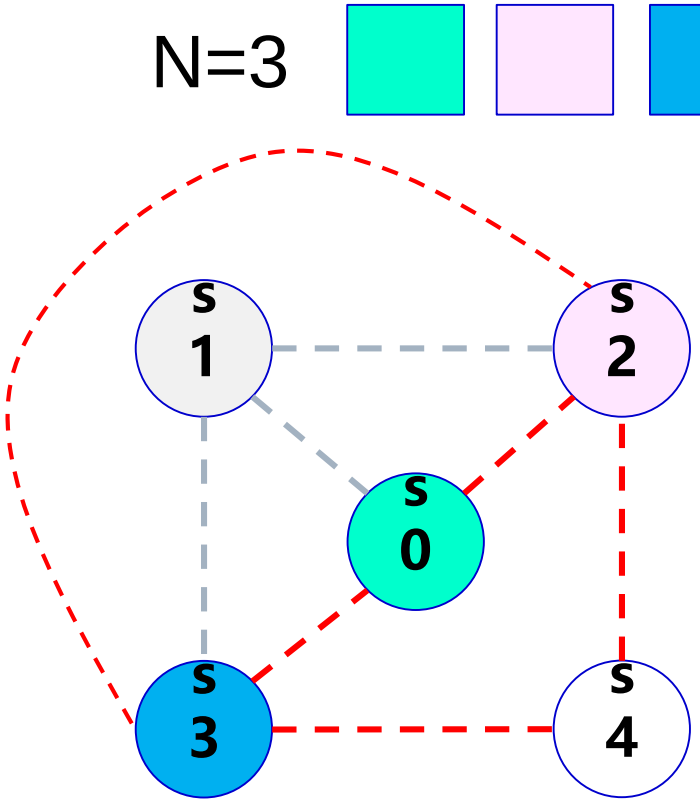


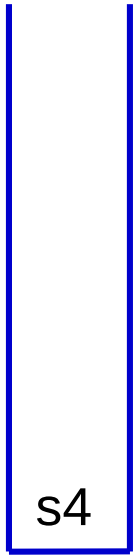
另一个图着色的例子



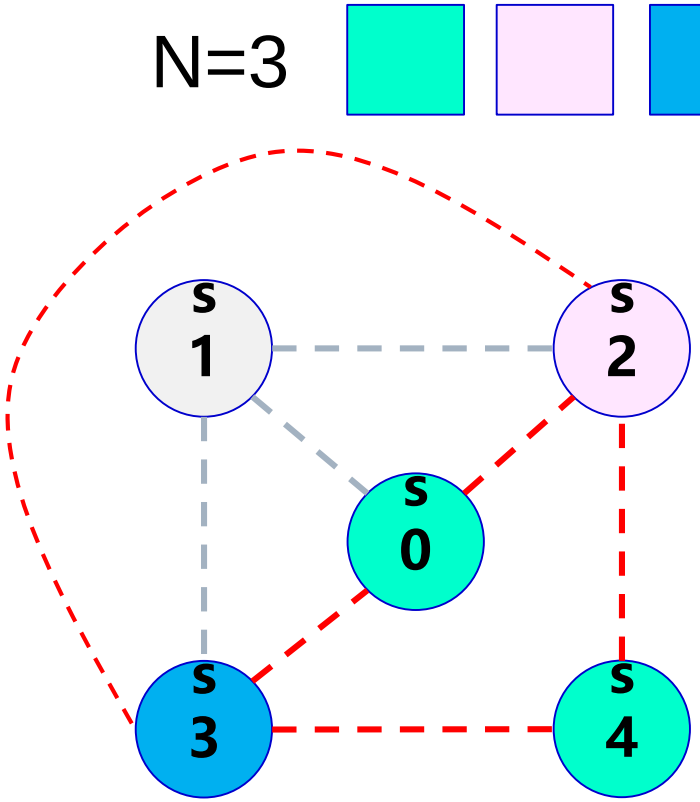


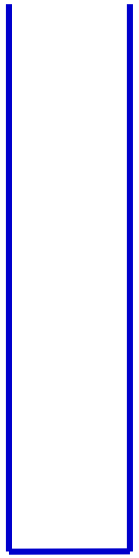
另一个图着色的例子





另一个图着色的例子





接下来该怎么办?

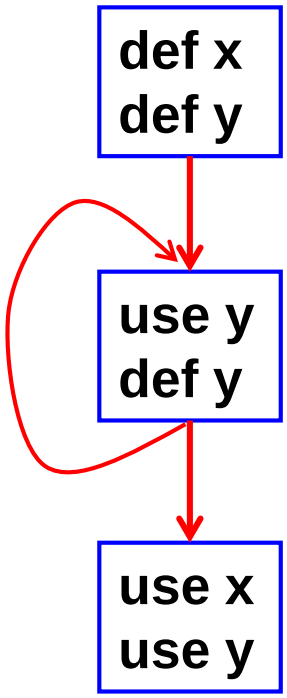
选择溢出的web结点

- 在所有结点的度数都>=N的时候，需要选择一个web把它的值保存到内存中，把它拆分为多个web
  - 需要重新进行着色
- 如何选择要溢出的web结点？
  - 选择一个度数>=N的结点
  - 它的溢出代码(spill cost)应该最小
- 溢出代价是什么？
  - 把web中的值放到内存中所需的额外的load和store指令的个数

如何计算溢出代价

- 理想的溢出代价：额外的load和store指令的动态开销（事实上无法计算）
  - 无法预计运行时会采取哪个分支或者循环会执行多少次
  - 每次执行时可能代价并不相同
- 解决方案：可以使用静态的近似估算
  - 例如，可以假设一个循环会执行10或100次
  - 选择溢出代价最低的web

溢出代价的例子



- x的溢出代价：

  storeCost + loadCost

- y的溢出代价：

  11*storeCost + 11*loadCost

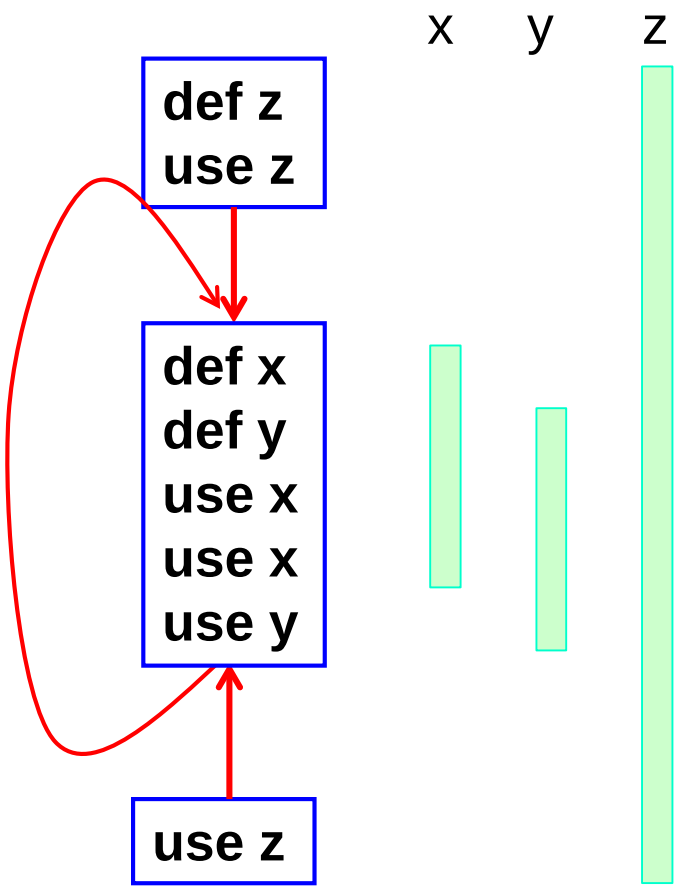
- 如果只有1个寄存器，应该选择哪个变量溢出？

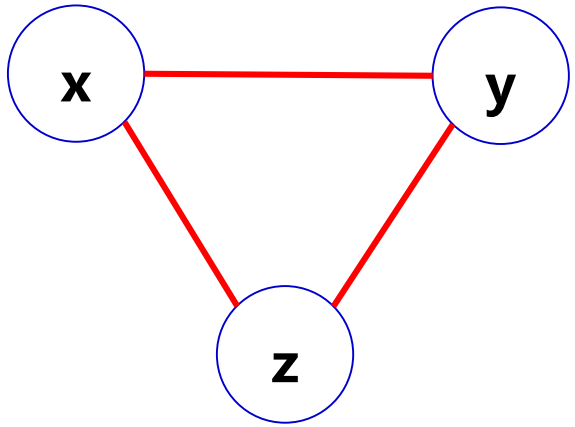
Web拆分（splitting）

- 对一个web进行拆分，得到多个活跃范围更小的web，可以降低它在干扰图的度数，从而使得拆分后的图可能可以着色

- 拆分的方法：
  - 把web中的值保存到内存中
  - 在web拆分的地方把值再加载回来

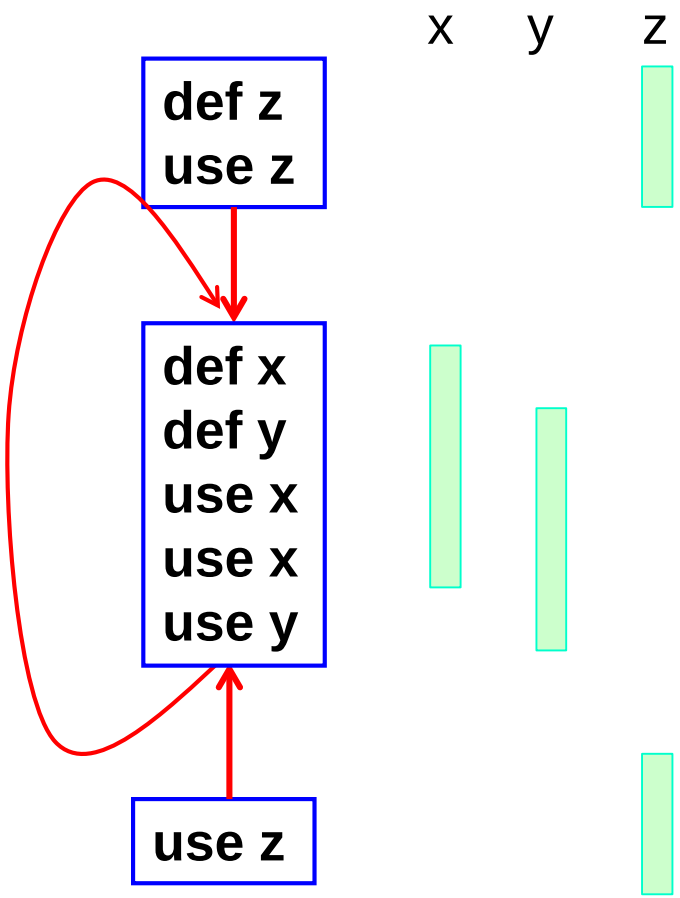
拆分的例子

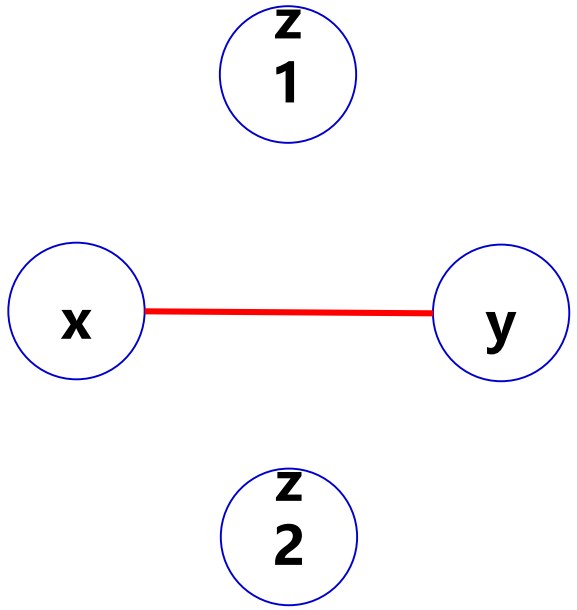




2-可着色吗?

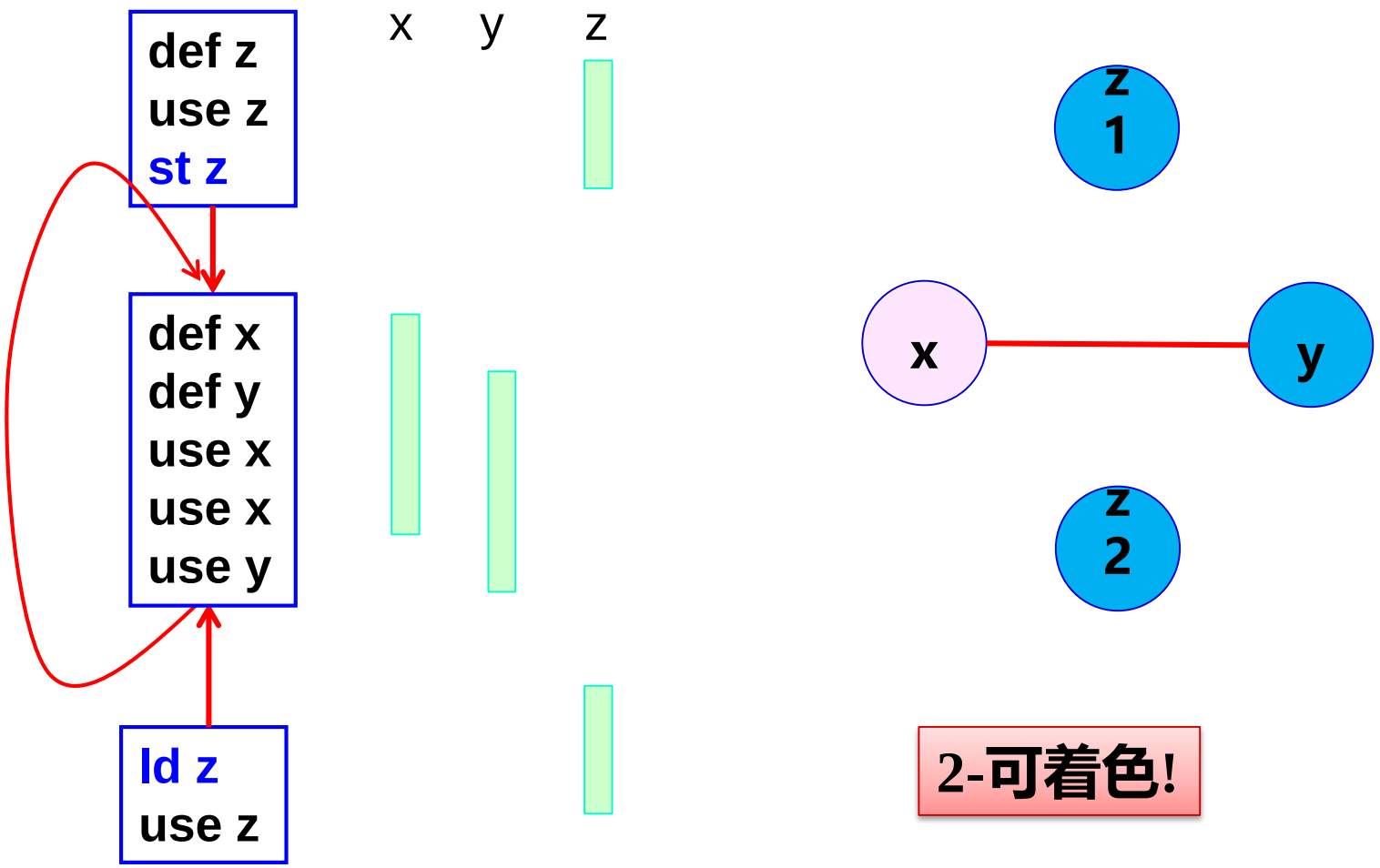
拆分的例子





2-可着色吗?

拆分的例子


def z
use z
st z
def x
def y
use x
use x
use y
ld z
use z
x
y
z
z
1
x
y
z
2
2-可着色!

预着色的结点(Precolored Nodes)

- 有些变量会被预先分配到固定的寄存器中
  - **Eg: call on x86/pentium**
    - **Defines (trashes) caller-save registers eax, ecx, edx**
- 把这些寄存器当做特殊的临时变量来处理
- 一开始就加到图中并着色

预着色的结点的处理

- 化简干扰图的过程中不能删除预先着色的结点
- 预先着色的结点作为着色过程的开始
- 如果干扰图化简为只包含已着色的结点，那么就可以按照前面的算法把删除的结点添回来并进行着色

本章小结

- 掌握目标代码生成的基本过程与方法
  - 简单的代码生成算法
  - 简单的寄存器分配算法
- 掌握基于图着色的寄存器分配算法
- 了解基本块的优化和窥孔优化的技术

作业

- Ex. 8.4.1
- Ex. 8.6.1, 8.6.3, 8.6.4, 8.6.5（都只做第2小题）
- Ex. 8.8.1
- 补充 8-1：使用图着色的寄存器分配法重复练习 8.6.4和8.6.5 (第2小题)
  - 提示：采用龙书第8.8.4节中讲的算法，先按照任意多寄存器分配，然后对所有分配的寄存器进行活跃分析，建立干扰图，通过图着色的方法进行二次分配。