北京大学计算机学院

2025 年春季学期　　《编译原理(实验班)》

第七讲　目标代码生成（基础版）

Target Code Generation

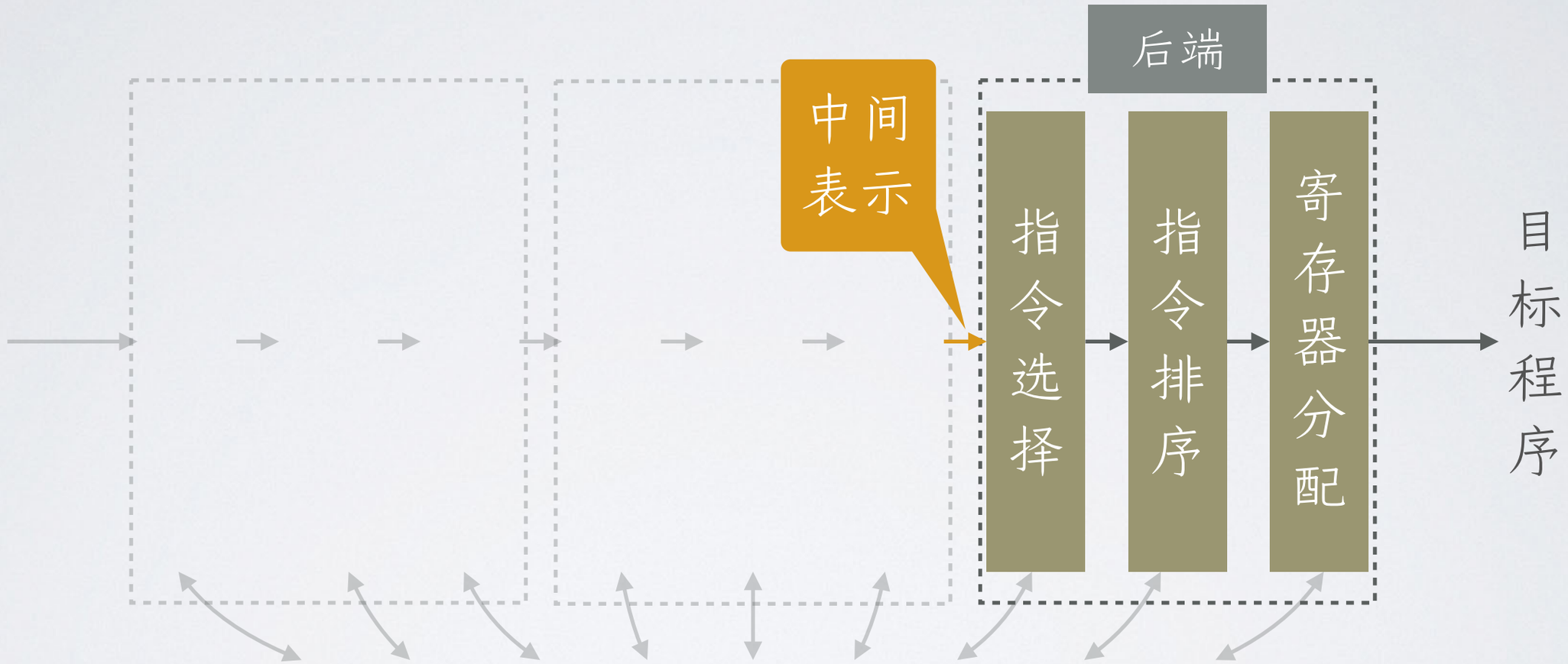
主要内容

- 目标代码生成的基本任务
- 目标机模型
- 指令选择
- 寄存器分配
- 对应龙书章节：第 8 章

主要内容

- **目标代码生成的基本任务**
- 目标机模型
- 指令选择
- 寄存器分配

后端：目标代码生成



目标代码生成的基本任务

- 生成机器代码/虚拟机字节码
  - **能够立即执行/解释的完整程序**，所有符号的地址已确定
  - **可重定位的目标模块**，需进行进一步的链接的加载
  - **汇编程序**，需进行汇编转换成机器代码/虚拟机字节码
- 生成**高质量**的目标代码
  - 目标代码的大小尽量小
  - 目标代码的运行速度尽量快
  - 充分利用目标机的指令系统的特点，特别是寄存器
- 基本任务：
  - **指令选择**
  - 指令排序
  - **寄存器分配**

主要内容

- 目标代码生成的基本任务
- **目标机模型**
- 指令选择
- 寄存器分配
- 局部优化

目标机模型

- 类 RISC 计算机，按字节寻址，以 4 个字节为 1 个字(word)
- 通用寄存器 R1, R2, …, Rn
- 使用如下机器指令，每条指令的长度为 8 字节：
  - LD *dst*, *addr*: 把位置 *addr* 上的值加载到位置 *dst*
  - LD r_1, r_2: 寄存器到寄存器的拷贝
  - ST *x*, *r*: 把寄存器 *r* 中的值保存到位置 *x*
  - *OP dst*, src_1, src_2: 把位置 src_1 和 src_2 中的值运算后将结果放到位置 *dst* 中
    - *OP* 是诸如 ADD 或 SUB 的运算符
  - BR *L*: 控制流转向标号为 *L* 的指令
  - B*cond r*, *L*: 对寄存器 *r* 中的值进行测试，如果为真则转向标号 *L*
    - *cond* 是诸如 LTZ 或 NEZ 的常见测试

目标机的寻址模式

- *contents*(*addr*) 表示 *addr* 所代表的位置中的内容
- *lvalue*(*x*) 表示分配给变量 *x* 的内存位置

位置形式	汇编表示	地址
变量名	x	$lvalue(x)$
数组索引	$a(r)$	$lvalue(a) + contents(r)$
直接常数	$\#M$	M
寄存器	r	r
间接寄存器	$*r$	$contents(r)$
索引	$M(r)$	$M + contents(r)$
间接索引	$*M(r)$	$contents(M + contents(r))$

例子

- `x = y - z`

```
LD  R1, y
LD  R2, z
SUB R1, R1, R2
ST  x, R1
```

- `*p = y`

```
LD R1, p
LD R2, y
ST 0(R1), R2
```

- `b = a[i]`

```
LD  R1, i
MUL R1, R1, 8
LD  R2, a(R1)
ST  b, R2
```

- `if x < y goto L`

```
LD   R1, x
LD   R2, y
SUB  R1, R1, R2
BLTZ R1, Ltgt
```

进行栈式管理的目标代码

- 生成支持栈式存储管理的目标代码
  - 生成过程**调用和返回**的目标代码序列
  - 将 IR 中的**名字**转换成为目标代码中的地址
- 考虑如下简化的调用/返回的三地址代码：
  - call *callee*
  - return
- 过程 *callee* 有如下属性(编译时确定)：
  - *callee*.codeArea：运行时代码区中 *callee* 的第一条指令的地址
  - *callee*.recordSize：*callee* 的一个活动记录的大小

过程的调用和返回

- 在简化场景下只需考虑在活动记录中保存**返回地址**
- 假设寄存器 SP 中维护一个指向栈顶的指针
- **调用指令序列**
  - 调用者　　ST -4(SP), #*here* + 16
    BR *callee*.codeArea
  - 被调用者　　SUB SP, SP, #*callee*.recordSize
- **返回指令序列**
  - 被调用者　　ADD SP, SP, #*callee*.recordSize
    BR *-4(SP)

名字的运行时地址

- 生成中间代码时，可以计算局部程序变量的**相对偏移**
- 寻址模式中的局部变量可以实现为 SP 和其偏移的结合

- **源程序：**

```
int a; int b;
a = b + 1;
```

- **符号表：**

  a 的相对偏移为 0

  b 的相对偏移为 4

- **三地址代码：**

```
a = b + 1;
```

- **目标代码：**

```
LD  R1, 4(SP)
ADD R1, R1, #1
ST  0(SP), R1
```

例子

```
void fact(int k, int *r) {
  if (k == 0) *r = 1;
  else {
    fact(k - 1, r);
    *r = *r * k;
  }
}
```

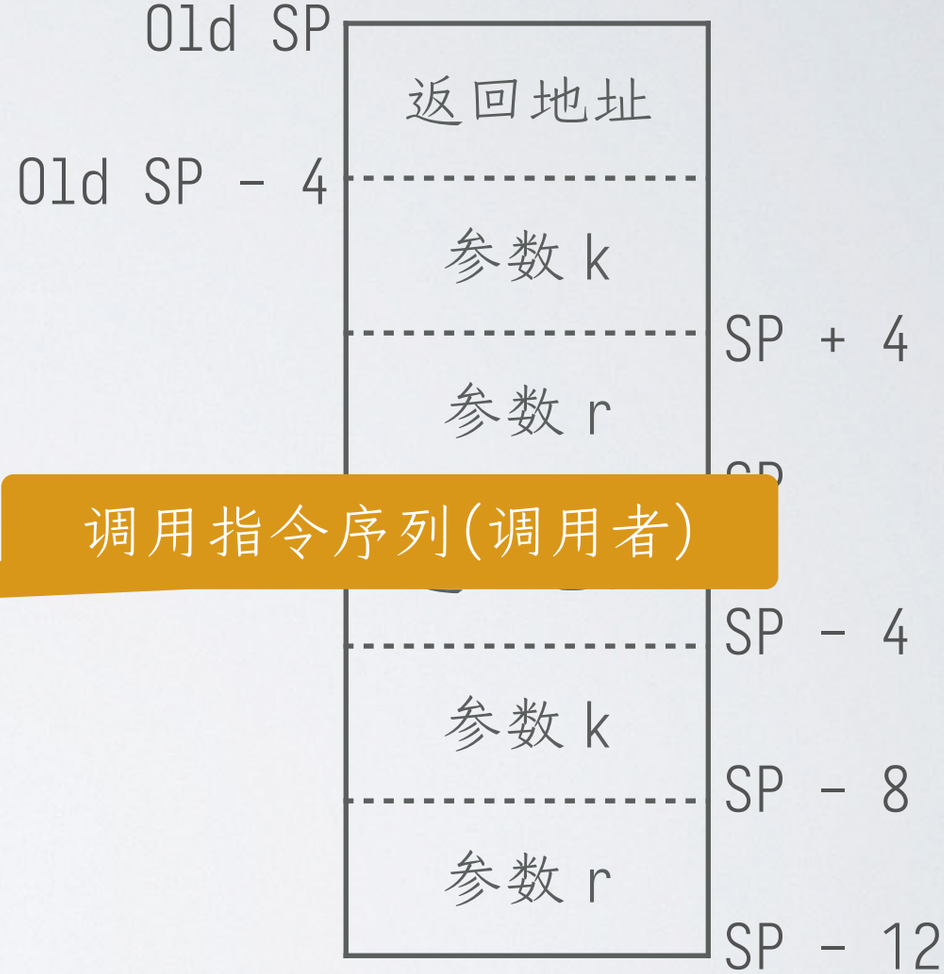
```
fact(k, r):
(1)  if k == 0 goto (3)
(2)  goto (5)
(3)  t1 = 1
(4)  *r = t1
(5)  goto (10)
(6)  t1 = 1
(7)  t2 = k - t1
(8)  call fact(t2, r)
(9)  t3 = *r
(10)  t4 = t3 * k
(11)  *r = t4
(12) return
```

```
200:  SUB  SP, SP, #12
208:  LD   R1, 4(SP)
216:  BEQZ R1, 232
224:  BR   264
232:  LD   R1, #1
240:  LD   R2, 0(SP)
248:  ST   0(R2), R1
256:  BR   376
264:  LD   R1, #1
272:  LD   R2, 4(SP)
280:  SUB  R2, R2, R1
288:  LD   R1, 0(SP)
296:  ST   -8(SP), R2
304:  ST   -12(SP), R1
312:  ST   -4(SP), #328
320:  BR   200
328:  LD   R1, 0(SP)
336:  LD   R1, 0(R1)
344:  LD   R2, 4(SP)
352:  MUL  R2, R1, R2
360:  LD   R1, 0(SP)
368:  ST   0(R1), R2
376:  ADD  SP, SP, #12
384:  BR   *-4(SP)
```

调用指令序列(被调用者)

调用指令序列(调用者)

返回指令序列(被调用者)



主要内容

- 目标代码生成的基本任务
- 目标机模型
- **指令选择**
- 寄存器分配

为单个基本块生成代码

- 依次考虑各个三地址语句，使用寄存器来存放基本块里的值
- **主要问题：最大限度地利用寄存器**
  - **假定有任意多个寄存器可以使用**
  - 避免不必要的加载和保存指令

- 寄存器的使用方法：
  - 执行运算时，把运算分量存放在寄存器中
  - 用来做临时变量
  - 存放在一个基本块中计算而在另一个基本块中使用的**(全局)值**
  - 帮助进行运行时环境管理(比如传递函数参数和返回值)

代码生成算法的基本思想

- 为一个三地址语句生成机器指令时：
  - 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(register descriptor)**
    - 为每个寄存器维护
    - 跟踪哪些变量的当前值存放在该寄存器内
  - **地址描述符(address descriptor)**
    - 为每个程序变量维护
    - 跟踪哪些位置(寄存器、栈中位置等)可以找到该变量的当前值

代码生成的例子（1）

- 考虑下面的三地址代码基本块:

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

- 为方便讨论，假设可以在目标代码中使用程序变量名
- 入口处的寄存器和地址描述符:

R1	R2	R3	R4	R5	R6	R7

a	b	c	d	t	u	v
a	b	c	d			

代码生成的例子（2）

R1	R2	R3	R4	R5	R6	R7

a	b	c	d	t	u	v
a	b	c	d			

```
t = a - b
    LD  R1, a
    LD  R2, b
    SUB R3, R1, R2
```

R1	R2	R3	R4	R5	R6	R7
a	b	t				

a	b	c	d	t	u	v
a,R1	b,R2	c	d	R3		

代码生成算法（1）

- 假设寄存器选择函数 $getReg(i)$ 给三地址语句 i 的每个涉及到内存位置选择寄存器

- **三地址语句 $x = y\ op\ z$:**
  - 调用 $getReg(x = y\ op\ z)$，给 x、y、z 选择寄存器 R_x、R_y、R_z
  - 查 R_y 的寄存器描述符，如果 y 不在 R_y 中则生成指令 LD R_y, y'，其中 y' 是某个存放了 y 的值的内存位置
  - 对 z 做与上一步类似的处理
  - 生成指令 $OP\ R_x, R_y, R_z$，其中 OP 对应 op（比如 ADD 对应 +）
  - 更新寄存器和地址描述符

代码生成算法（2）

- **管理寄存器和地址描述符：**
  - 对于指令 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 x 的位置集合
    - 任何不同于 x 的变量的地址描述符中删除 R
  - 对于指令 $OP\ R_x, R_y, R_z$：
    - R_x 的寄存器描述符：**只包含 x**
    - x 的地址描述符：**只包含 R_x**
    - 任何不同于 x 的变量的地址描述符中删除 R_x

代码生成的例子（3）

R1	R2	R3	R4	R5	R6	R7
a	b	t				

a	b	c	d	t	u	v
a,R1	b,R2	c	d	R3		

```
u = a - c
    LD  R4, c
    SUB R5, R1, R4
```

R1	R2	R3	R4	R5	R6	R7
a	b	t	c	u		

a	b	c	d	t	u	v
a,R1	b,R2	c,R4	d	R3	R5	

代码生成的例子（4）

R1	R2	R3	R4	R5	R6	R7
a	b	t	c	u		

a	b	c	d	t	u	v
a,R1	b,R2	c,R4	d	R3	R5	

```
v = t + u
    ADD R6, R3, R5
```

R1	R2	R3	R4	R5	R6	R7
a	b	t	c	u	v	

a	b	c	d	t	u	v
a,R1	b,R2	c,R4	d	R3	R5	R6

代码生成算法（3）

- **三地址语句 $x = y$:**
  - $getReg(x = y)$ 总是为 x 和 y 选择相同的寄存器
  - 如果 y 不在 R_y 中，那么生成指令 LD R_y, y'，其中 y' 是存放 y 的位置
  - 更新寄存器和地址描述符
- **管理寄存器和地址描述符:**
  - 处理复制语句 $x = y$:
    - 如果生成了 LD 指令，则先按照 LD 的规则处理
    - R_y 的寄存器描述符：把 x 加入变量集合
    - x 的地址描述符：**只包含 R_y**

代码生成的例子（5）

R1	R2	R3	R4	R5	R6	R7
a	b	t	c	u	v	

a	b	c	d	t	u	v
a,R1	b,R2	c,R4	d	R3	R5	R6

```
a = d
    LD  R1, d
```

R1	R2	R3	R4	R5	R6	R7
a,d	b	t	c	u	v	

a	b	c	d	t	u	v
R1	b,R2	c,R4	d,R1	R3	R5	R6

代码生成的例子（6）

R1	R2	R3	R4	R5	R6	R7
a,d	b	t	c	u	v	

a	b	c	d	t	u	v
R1	b,R2	c,R4	d,R1	R3	R5	R6

```
d = v + u
    ADD R7, R6, R5
```

R1	R2	R3	R4	R5	R6	R7
a	b	t	c	u	v	d

a	b	c	d	t	u	v
R1	b,R2	c,R4	R7	R3	R5	R6

代码生成算法（4）

- **基本块的结尾：**
  - 如果变量 x 在出口处**活跃(其值在后续的控制流中会被用到)**，且查 x 的地址描述符发现其不在自己的内存位置上，则生成指令 ST x, R_x
  - 更新寄存器和地址描述符
- **管理寄存器和地址描述符：**
  - 对于指令 ST x, R：
    - 生成这种指令时 R 一定存放了 x 的当前值
    - x 的地址描述符：把 x 自己的内存位置加入位置集合

代码生成的例子（7）

R1	R2	R3	R4	R5	R6	R7
a	b	t	c	u	v	d

a	b	c	d	t	u	v
R1	b,R2	c,R4	R7	R3	R5	R6

```
(exit)
    ST  a, R1
    ST  d, R7
```

假设 a, b, c, d 在出口处活跃

R1	R2	R3	R4	R5	R6	R7
a	b	t	c	u	v	d

a	b	c	d	t	u	v
R1,a	b,R2	c,R4	R7,d	R3	R5	R6

不想维护这些描述符……

- **解决方法：语句结束后立即把值都写回内存位置**

- $x = y\ op\ z$：
  - 调用 $getReg(x = y\ op\ z)$，给 x、y、z 选择寄存器 R_x、R_y、R_z
  - 生成指令 LD R_y, y 和 LD R_z, z
  - 生成指令 $OP\ R_x, R_y, R_z$，其中 OP 对应 op（比如 ADD 对应 +）
  - 生成指令 ST x, R_x

```
t = a - b
    LD  R1, a
    LD  R2, b
    SUB R3, R1, R2
    ST  t, R3
```

```
a = d
    LD  R1, d
    ST  a, R1
```

活跃变量分析

- 变量的**定值(def)**和**使用(use)**
  - 谓词 $def(i,x)$: 语句 i 给变量 x 进行了赋值
  - 谓词 $use(i,x)$: 语句 i 使用了变量 x 的值
- 如果 $def(i,x)$ 且 $use(j,x)$, 并且从语句 i 到语句 j 存在一条路径没有其它的对变量 x 的赋值, 那么 j 使用了 i 处计算的 x
  - 谓词 $live_{out}(i,x)$: 变量 x 在语句 i 后的程序点上**活跃(live)**
- 活跃变量信息的用途: 实现寄存器选择函数 $getReg$
  - 如果一个寄存器只存放了 x 的值, 且 x 在 i 处不活跃, 那么这个寄存器在 i 处可以用于其它目的

活跃变量分析

- 如果 i 的下一条语句是 j, 那么
  - 如果 $use(j,x)$, 那么 $live_{out}(i,x)$
  - 如果 $live_{out}(j,x)$, **且** $\neg def(j,x)$, 那么 $live_{out}(i,x)$
- **例子:** 假设出口处 i, j, a 为活跃变量

	i,j,a 活跃
t_1 = 10 * i	
	i,j,a,t_1 活跃
t_2 = t_1 + j	
	i,j,a,t_2 活跃
t_3 = 8 * t_2	
	i,j,a,t_3 活跃
t_4 = t_3 - 88	
	i,j,a,t_4 活跃
a[t_4] = 0.0	
	i,j,a 活跃
j = j + 1	
	i,j,a 活跃

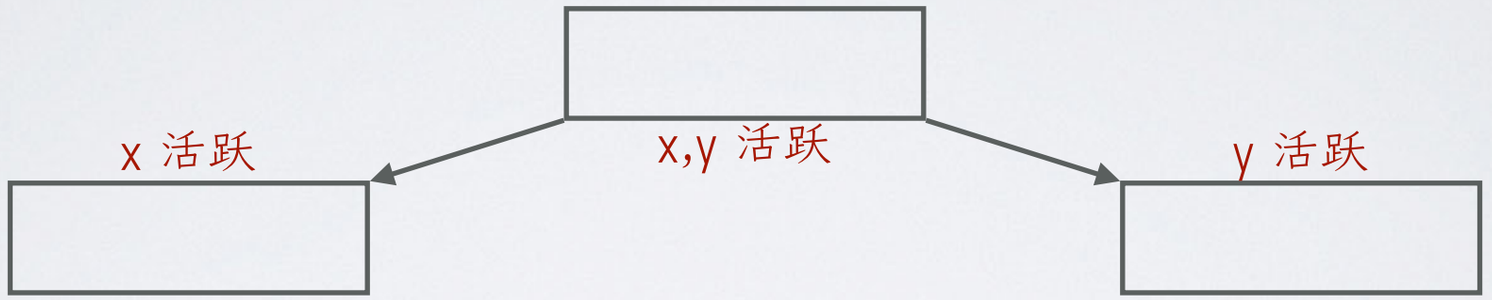
活跃变量分析的算法

- **输入**：基本块 B，假设 B 出口处所有非临时变量都是活跃的
- **输出**：块中各个语句 i 处的变量的活跃性信息
- **算法**：
  - 从 B 的最后一个语句开始**反向扫描**
  - 对于每个形如 $x = y\ op\ z$ 的语句 i，依次做如下处理：
    - 把 x、y、z 当前的活跃性信息关联到语句 i
    - 设置 x 为「不活跃」
    - 设置 y 和 z 为「活跃」
  - **问：最后两步的设置可以交换顺序吗？**

控制流图上的活跃变量分析

- **问题：如何进行跨基本块的活跃变量分析？**

- 基本块出口处的活跃变量由其后继结点的入口活跃变量决定



- 如果一个基本块出口处活跃信息发生变化，就对其重新计算
- 反复进行直到每个基本块的活跃信息都不再变化
- **课程后面将介绍数据流分析**

getReg 函数（1）

- **目标：**减少 LD 和 ST 的指令数目
  - 以 $x = y\ op\ z$ 为例
- **任务：**为运算分量(y 和 z)和结果(x)选择寄存器
- 给运算分量选择寄存器：
  - 如果已经在寄存器中，则选择该寄存器
  - 否则，如果有空闲寄存器，则选择一个空闲寄存器
  - 否则，设 R 是一个候选寄存器，其存放了 v 的值
    - 如果 v 的地址描述符包含其它位置，则可以用 R
    - 如果 v 就是 x 且不为运算分量，则可以用 R
    - 如果 v 在该语句后不是活跃变量，则可以用 R
    - **否则？**

getReg 函数（2）

- 当寄存器不能安全地重复使用时，需要进行**溢出操作(spill)**
- 设 R 是候选寄存器，它存放了变量 v 的值
  - 生成指令 ST v, R，并更新 v 的地址描述符
  - 如果 R 中还存放了别的变量的值，则可能要生成多条 ST 指令
- 给 $x = y\ op\ z$ 的运算结果选择寄存器：
  - 与运算分量的处理基本相同
  - 只存放了 x 的值的寄存器总是可接受的
  - 如果分量 y 在该语句后不是活跃变量，只存放了 y 的值的寄存器也可以接受(对分量 z 类似)
- 处理 $x = y$ 时，先选择 R_y，然后令 $R_x = R_y$

代码生成的第二个例子（1）

- 假设 a, b, c, d 在出口处活跃

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

- 入口处的寄存器**(只有三个)**和地址描述符：

R1	R2	R3

a	b	c	d	t	u	v
a	b	c	d			

代码生成的第二个例子（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 R3, R1, R2
```

活跃变量：a,b,c,d,t

R1	R2	R3
a	b	t

a	b	c	d	t	u	v
a,R1	b,R2	c	d	R3		

代码生成的第二个例子（3）

R1	R2	R3
a	b	t

a	b	c	d	t	u	v
a,R1	b,R2	c	d	R3		

```
u = a - c
    LD  R2, c
    SUB R1, R1, R2
```

活跃变量：b,c,d,t,u

R1	R2	R3
u	c	t

a	b	c	d	t	u	v
a	b	c,R2	d	R3	R1	

代码生成的第二个例子（4）

R1	R2	R3
u	c	t

a	b	c	d	t	u	v
a	b	c,R2	d	R3	R1	

```
v = t + u
    ADD R3, R3, R1
```

活跃变量：b,c,d,u,v

R1	R2	R3
u	c	v

a	b	c	d	t	u	v
a	b	c,R2	d		R1	R3

代码生成的第二个例子（5）

R1	R2	R3
u	c	v

a	b	c	d	t	u	v
a	b	c,R2	d		R1	R3

```
a = d
    LD  R2, d
```

活跃变量：a,b,c,u,v

R1	R2	R3
u	a,d	v

a	b	c	d	t	u	v
R2	b	c	d,R2		R1	R3

代码生成的第二个例子（6）

R1	R2	R3
u	a,d	v

a	b	c	d	t	u	v
R2	b	c	d,R2		R1	R3

```
d = v + u
    ADD R3, R3, R1
```

活跃变量：a,b,c,d

R1	R2	R3
u	a	d

a	b	c	d	t	u	v
R2	b	c	R3		R1	

代码生成的第二个例子（7）

R1	R2	R3
u	a	d

a	b	c	d	t	u	v
R2	b	c	R3		R1	

```
(exit)
    ST  a, R2
    ST  d, R3
```

R1	R2	R3
u	a	d

a	b	c	d	t	u	v
R2,a	b	c	R3,d		R1	

- 问题：只用两个寄存器可以生成不进行溢出操作的代码吗？

主要内容

- 目标代码生成的基本任务
- 目标机模型
- 指令选择
- **寄存器分配**

寄存器分配和指派

- **分配**：哪些值应该放在寄存器中
- **指派**：各个值应该存放在哪个寄存器
- **全局寄存器分配**
  - 前面的代码生成算法在基本块的结尾处会把所有活跃变量的值保存到内存中
  - 可以使一些寄存器在不同基本块中有一致的(全局)指派
  - 比如：循环变量
- **基本方法：**
  - 两个**不同时活跃**的变量可以使用同一个寄存器
  - 可以通过对变量进行**溢出操作**来改变变量的活跃性

一个两趟处理的方法

- 第一趟：
  - 假设有**无穷多个**符号寄存器
  - 用前面的生成算法生成目标代码
    - 可以不生成基本块结尾处的保存指令

- 第二趟：
  - 把物理寄存器指派给符号寄存器
  - 需要符号寄存器的**活跃信息**，构造寄存器**冲突图**
  - 通过**图着色**方法进行寄存器分配

第一趟的例子

a,b,c,d 是程序变量

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

```
LD  R1, a
LD  R2, b
SUB R3, R1, R2
LD  R4, c
SUB R5, R1, R4
ADD R6, R3, R5
LD  R1, d
ADD R7, R6, R5
ST  a, R1
ST  d, R7
```

活跃寄存器

```
{}
{R1}
{R1,R2}
{R1,R3}
{R1,R3,R4}
{R3,R5}
{R5,R6}
{R1,R5,R6}
{R1,R7}
{R7}
{}
```

通过图着色方法进行寄存器分配

- 1971 年 John Cokes 提出：全局寄存器分配可以视为一种**图着色**问题
  - 图着色方法最初在实验性编译程序 IBM 370 PL/I 中使用，很快得到了广泛的推广
- 图着色问题的简单描述
  - 已知一个图 G 和 $m > 0$ 种颜色，是否可以使用这 m 种颜色对 G 的结点进行着色，使得任意两个相邻的结点都具有不同的颜色？
    - ***m*-着色判定问题**：当 $m > 2$ 时是 NP 完全问题
  - 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)

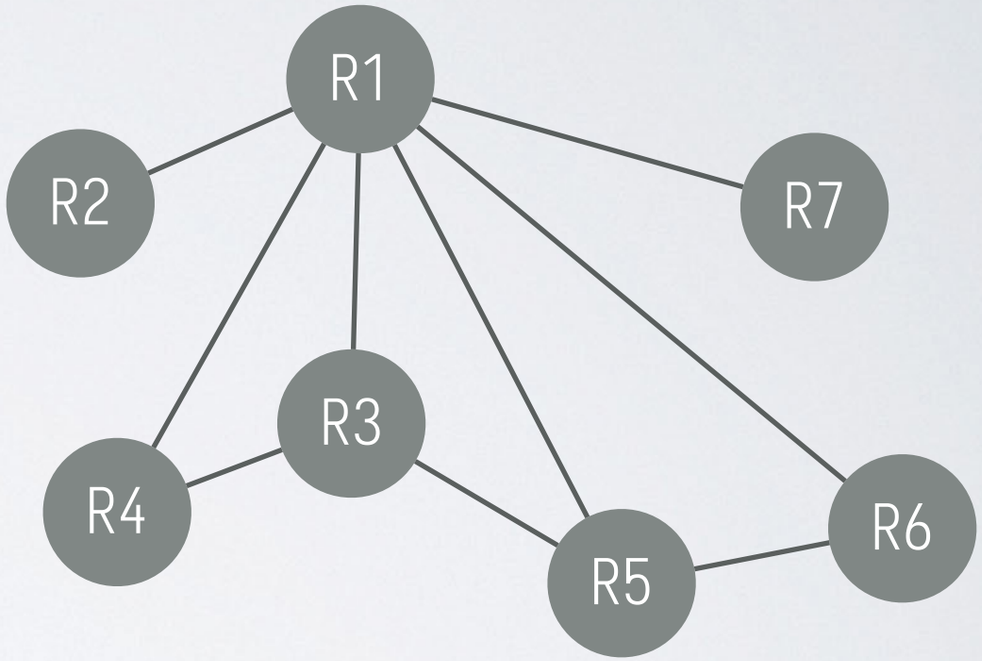
寄存器冲突图

- 构造**寄存器冲突图(register-interference graph)**
  - 结点：在第一趟代码生成中使用的符号寄存器
  - 边：两个符号寄存器不能指派同一个物理寄存器**(相互冲突)**则用边连起来
  - 进行图着色后，相同颜色的结点可以分配同一个物理寄存器

- **冲突：R_1 在 R_2 被定值的地方是活跃的**，则它们相互冲突
  - 也就是说如果存在一个指令 i，使得 $def(i, R_2)$ 且 $live_{out}(i, R_1)$
  - 谓词 $def(i, R)$：指令 i 给寄存器 R 进行了赋值
  - 谓词 $live_{out}(i, R)$：寄存器 R 在指令 i 后的程序点上活跃

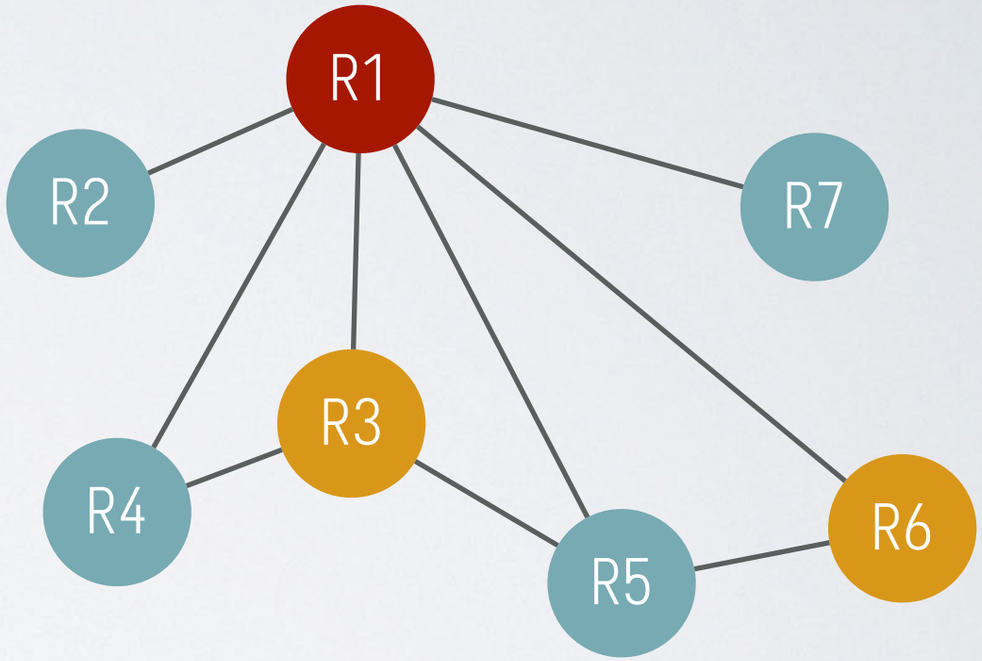
寄存器冲突图的例子（1）

	活跃寄存器
	{}
LD R1, a	{R1}
LD R2, b	{R1,R2}
SUB R3, R1, R2	{R1,R3}
LD R4, c	{R1,R3,R4}
SUB R5, R1, R4	{R3,R5}
ADD R6, R3, R5	{R5,R6}
LD R1, d	{R1,R5,R6}
ADD R7, R6, R5	{R1,R7}
ST a, R1	{R7}
ST d, R7	{}



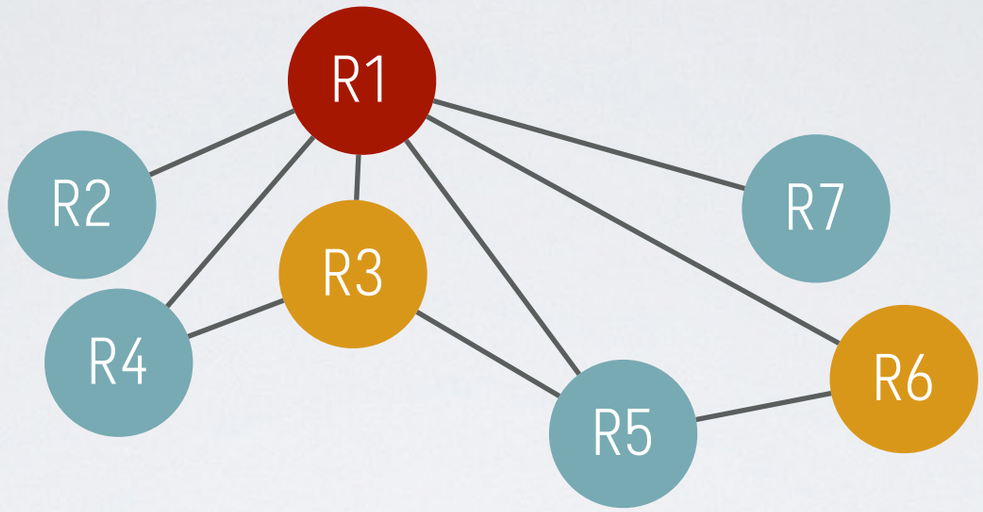
寄存器冲突图的例子（2）

	活跃寄存器
LD R1, a	{}
LD R2, b	{R1}
SUB R3, R1, R2	{R1,R2}
LD R4, c	{R1,R3}
SUB R5, R1, R4	{R1,R3,R4}
ADD R6, R3, R5	{R3,R5}
LD R1, d	{R5,R6}
ADD R7, R6, R5	{R1,R5,R6}
ST a, R1	{R1,R7}
ST d, R7	{R7}
	{}



可以进行 3-着色！

寄存器冲突图的例子（3）



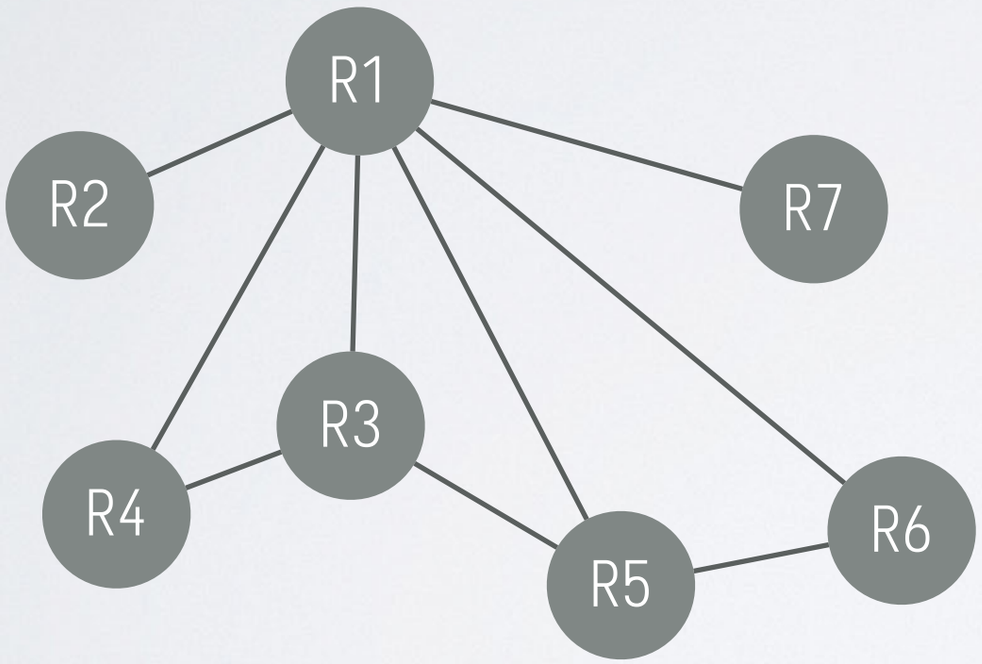

```
LD  R1, a
LD  R2, b
SUB R3, R1, R2
LD  R4, c
SUB R5, R1, R4
ADD R6, R3, R5
LD  R1, d
ADD R7, R6, R5
ST  a, R1
ST  d, R7
```

```
LD  R1, a
LD  R2, b
SUB R3, R1, R2
LD  R2, c
SUB R2, R1, R2
ADD R3, R3, R2
LD  R1, d
ADD R2, R3, R2
ST  a, R1
ST  d, R2
```

图着色的启发式技术

- **如果冲突图中每个结点的度数都 $< m$，则总是可以 m-着色**
  - 每个结点邻居的颜色最多 $m-1$ 种，总能对其着色
- 一个简单的算法：
  - 寻找度数 $< m$ 的结点，从图中删除，并把该结点压到一个栈中
  - 如果所有结点的度数都 $\geq m$：
    - **找到一个溢出结点，不对它着色**
    - 删除该结点
  - 当图为空的时候：
    - 从栈顶依次弹出结点
    - 选择该结点的邻居**没有使用的颜色**进行着色

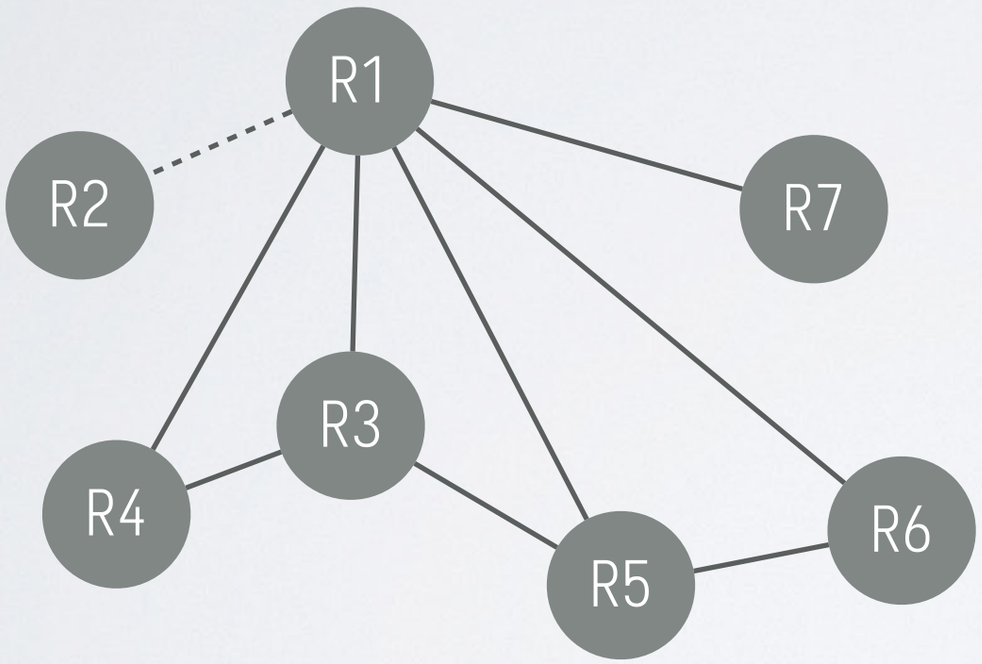
着色算法的例子（1）



$m = 3$

栈

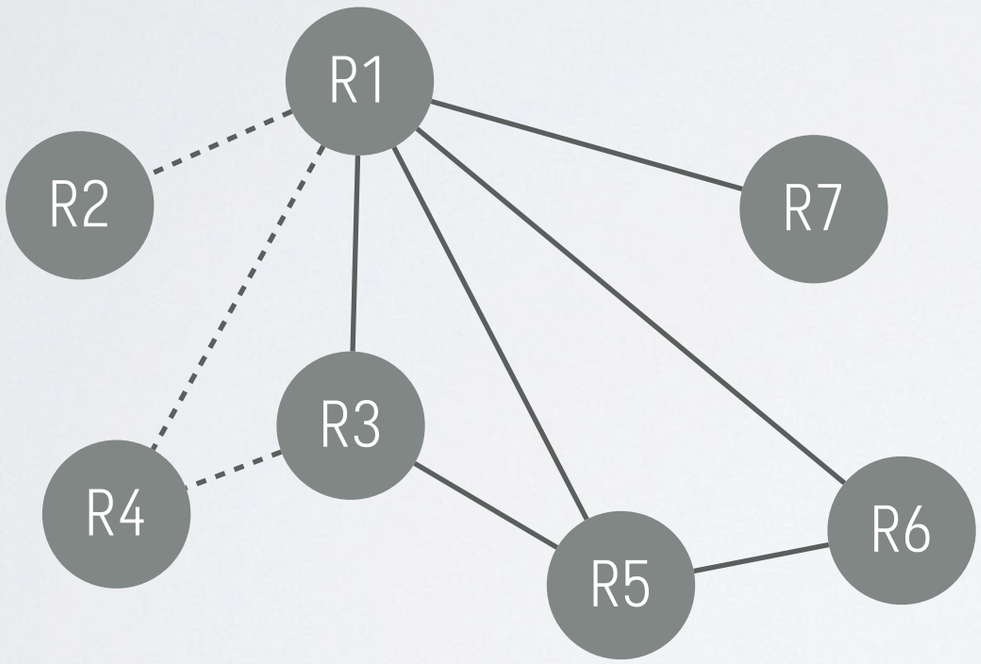
着色算法的例子（2）



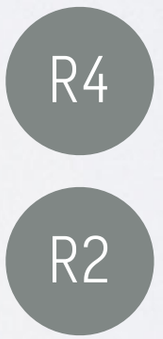
$m = 3$

栈

着色算法的例子（3）

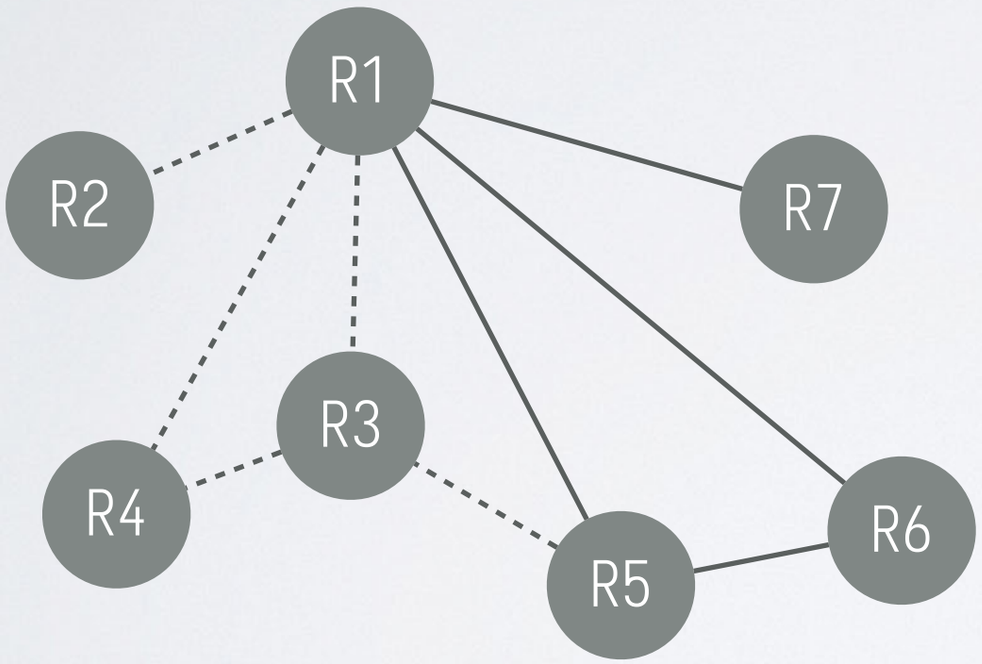


$m = 3$



栈

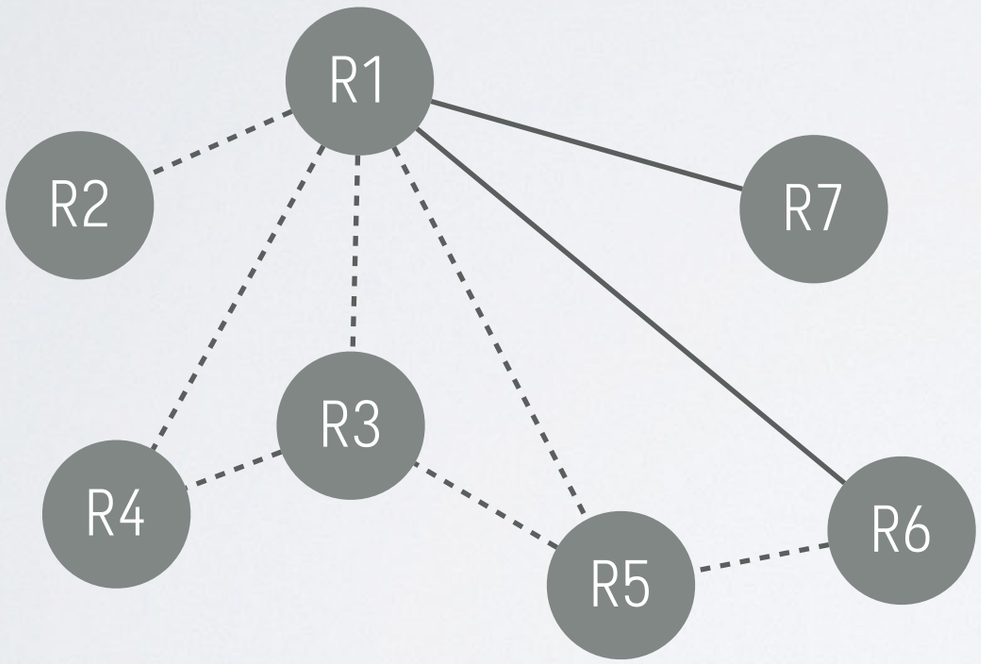
着色算法的例子（4）



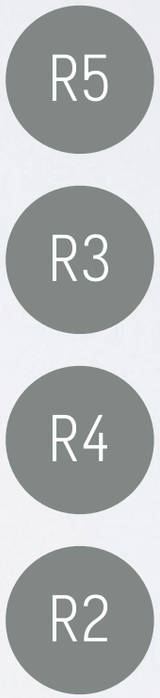
$m = 3$

栈

着色算法的例子（5）

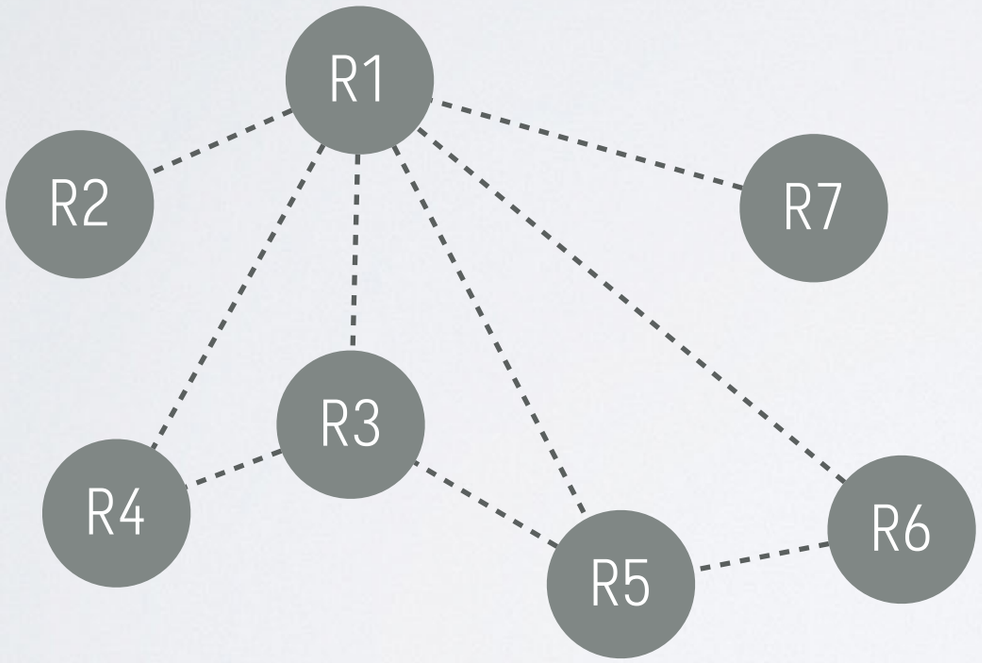


$m = 3$



栈

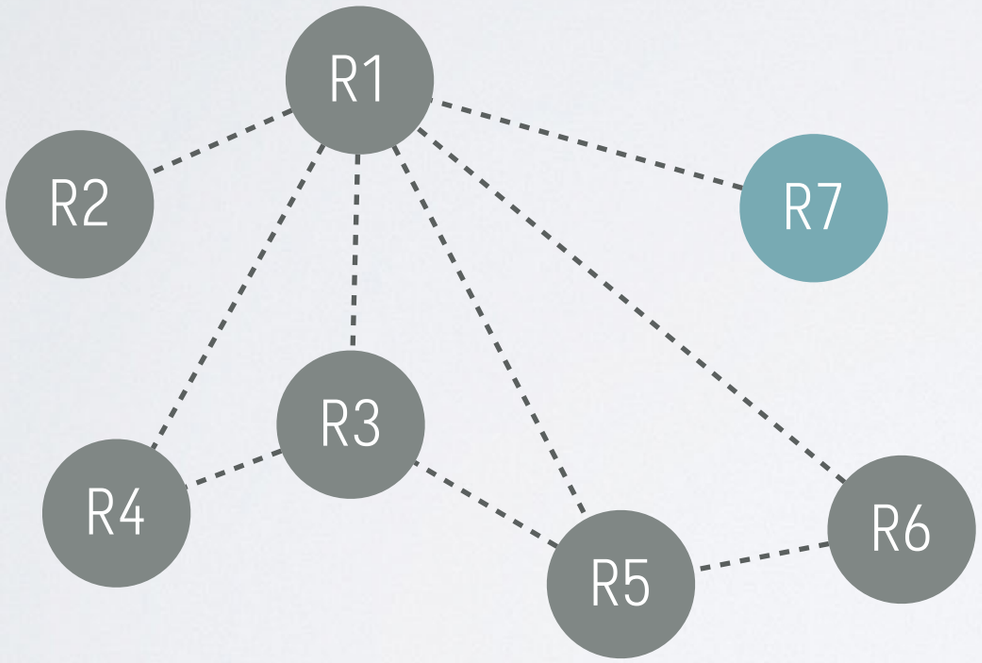
着色算法的例子（6）



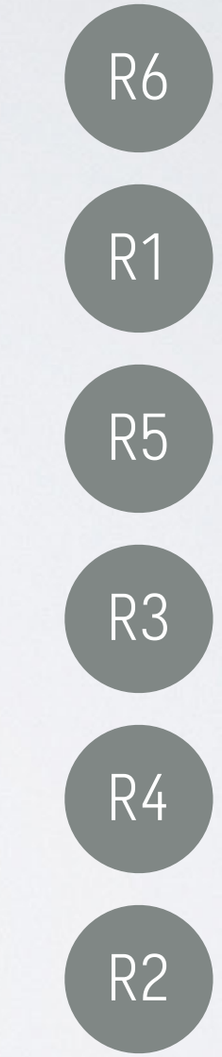
$m = 3$

栈

着色算法的例子（7）

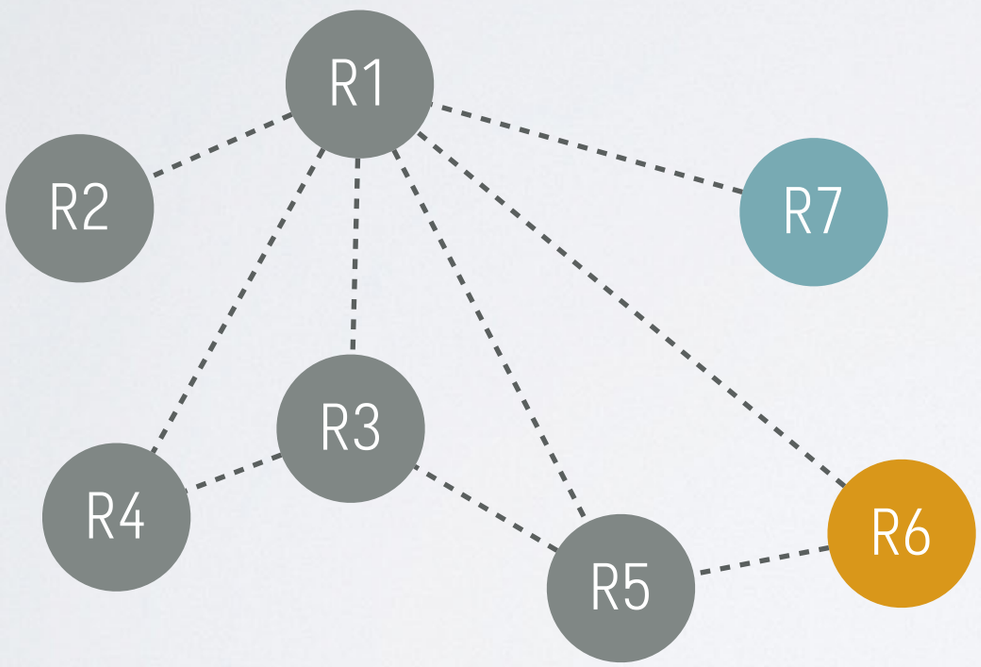


$m = 3$



栈

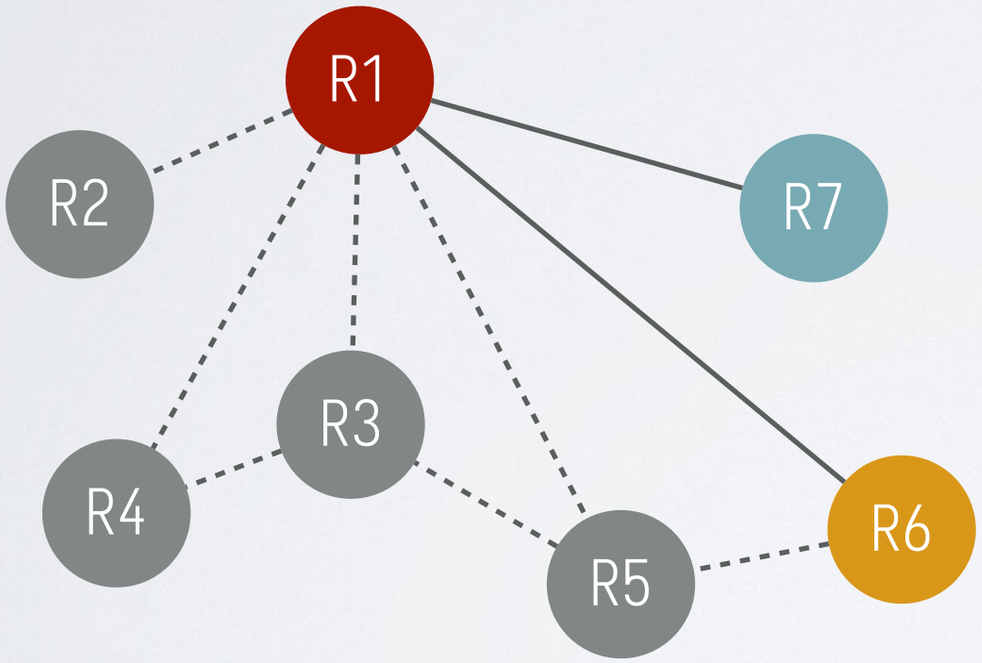
着色算法的例子（8）



$m = 3$

栈

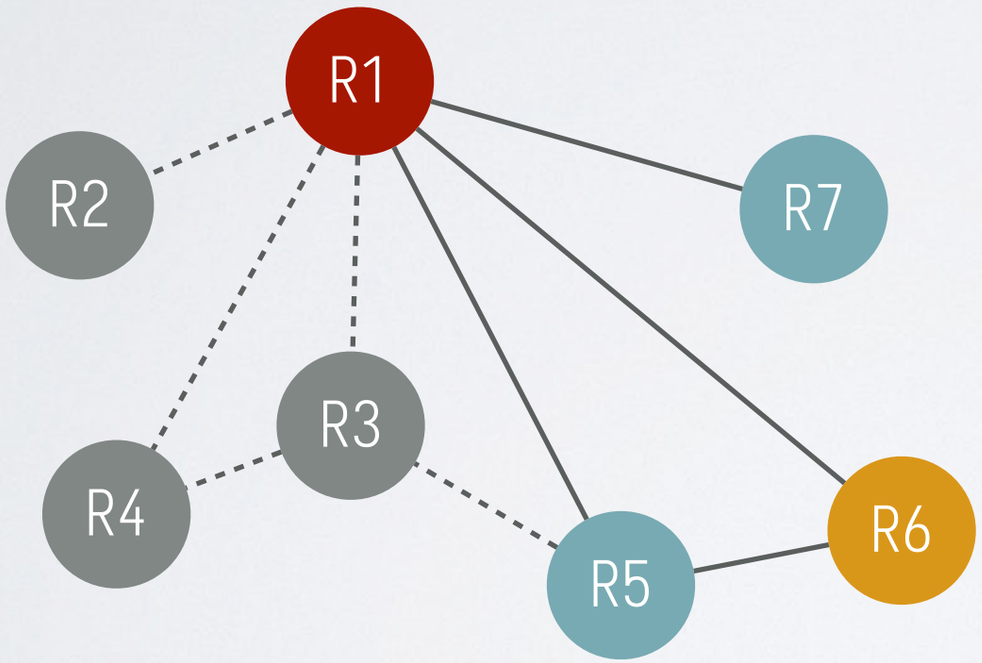
着色算法的例子（9）



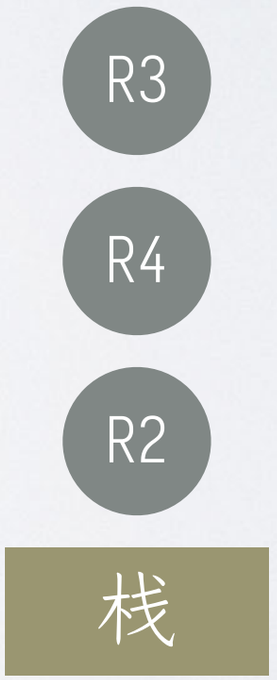
$m = 3$

栈

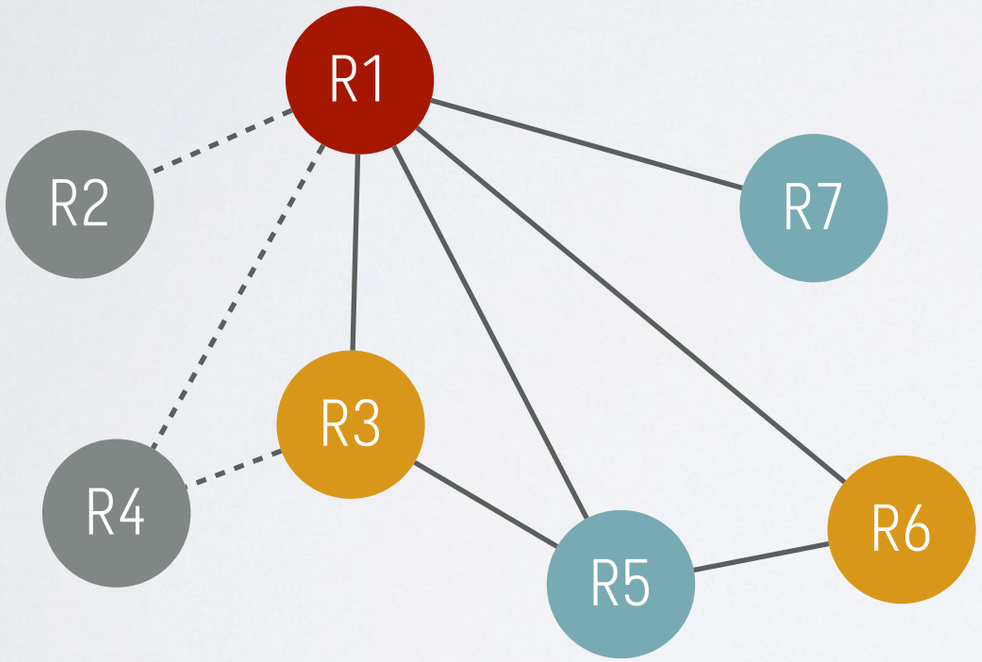
着色算法的例子（10）



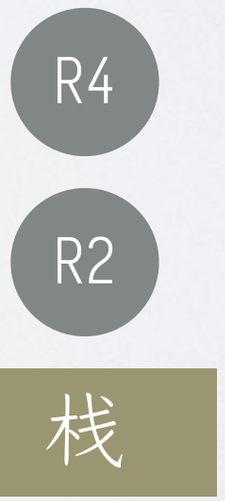
$m = 3$



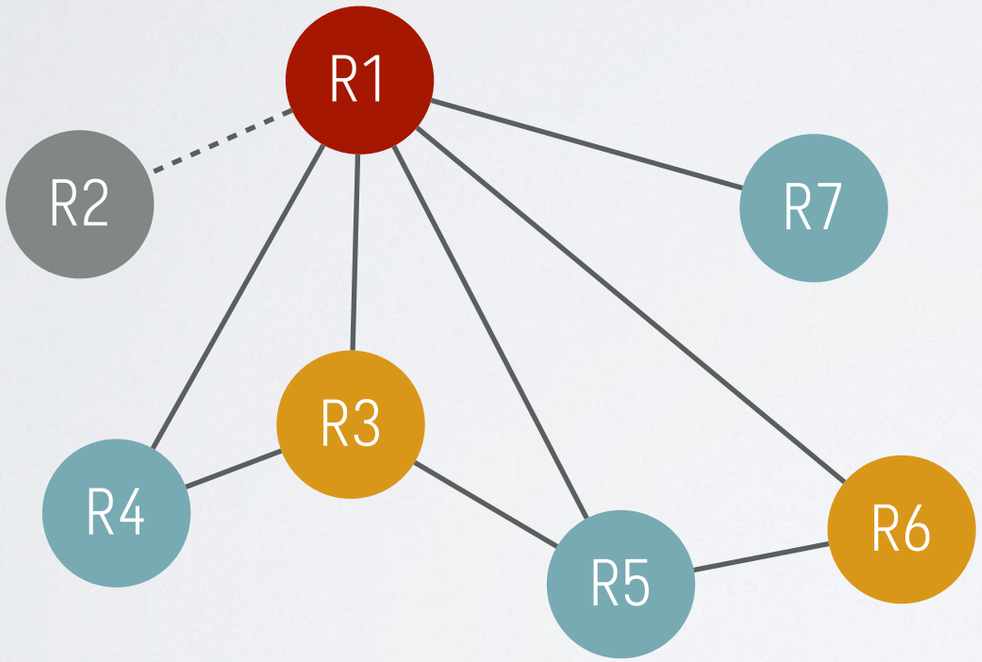
着色算法的例子（11）



$m = 3$



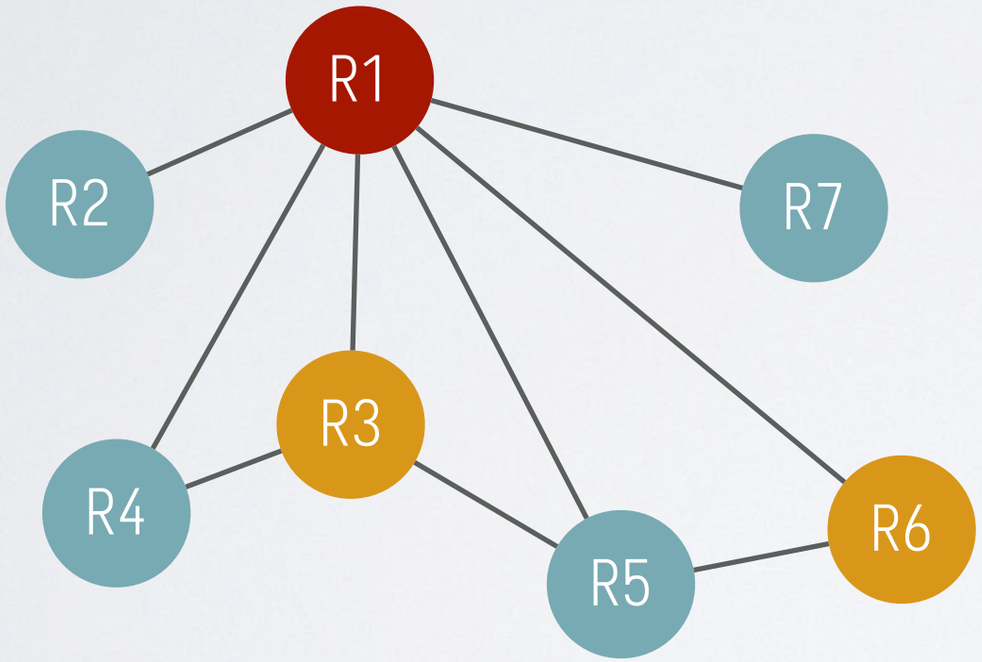
着色算法的例子（12）



$m = 3$

栈

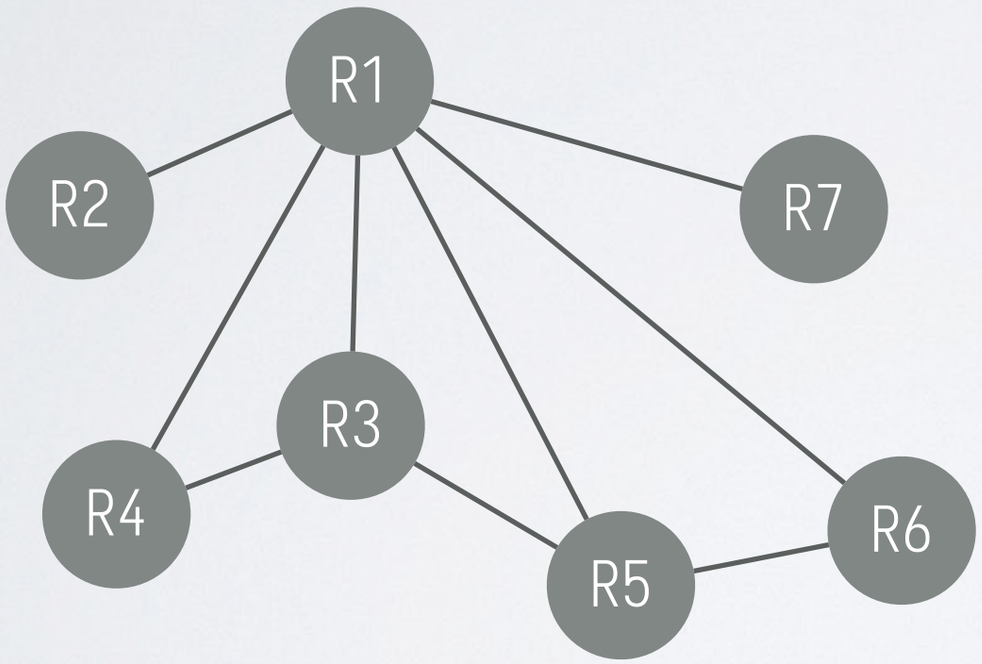
着色算法的例子（13）



$m = 3$

栈

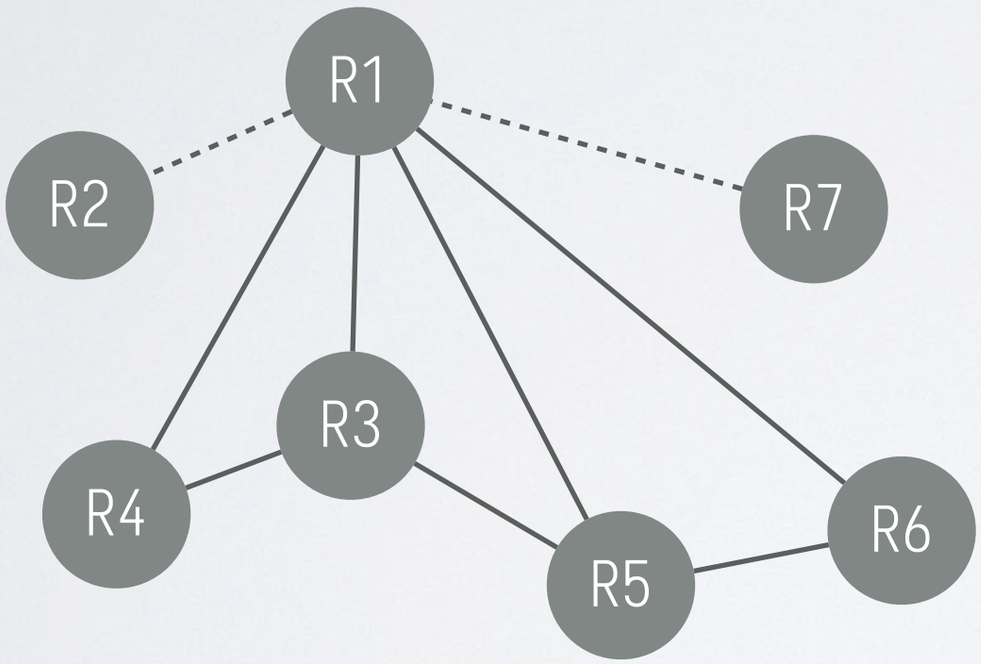
溢出的例子（1）



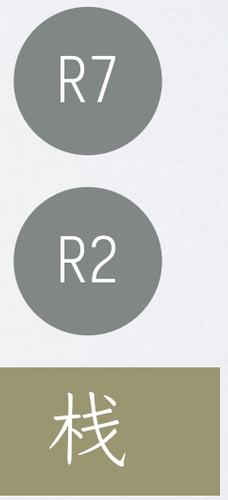
$m = 2$

栈

溢出的例子（2）



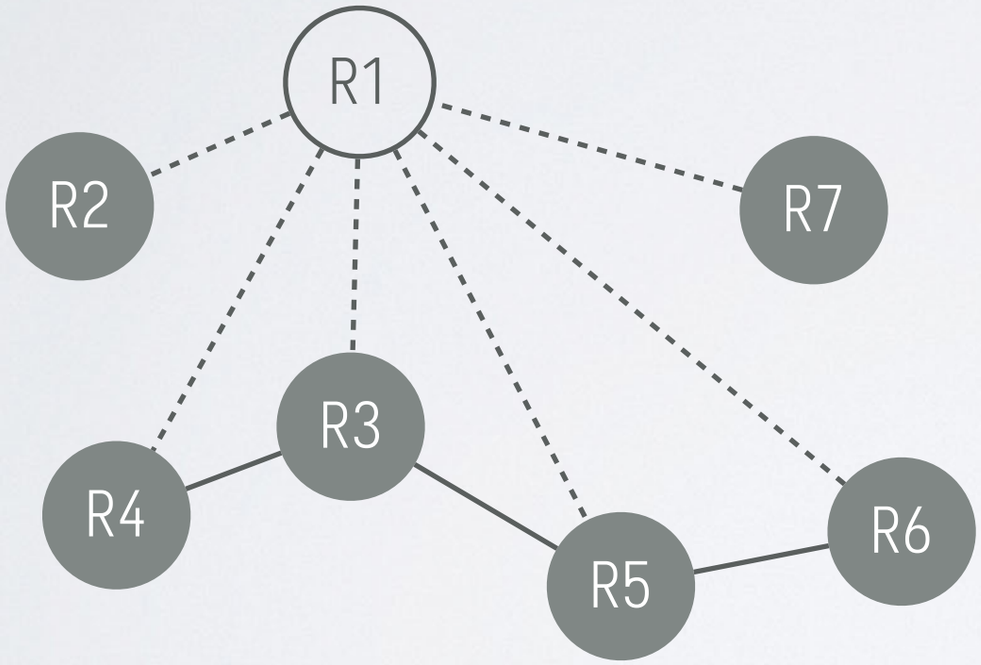
$m = 2$



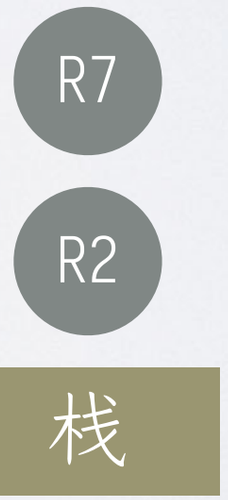
栈

溢出的例子（3）

溢出 R1

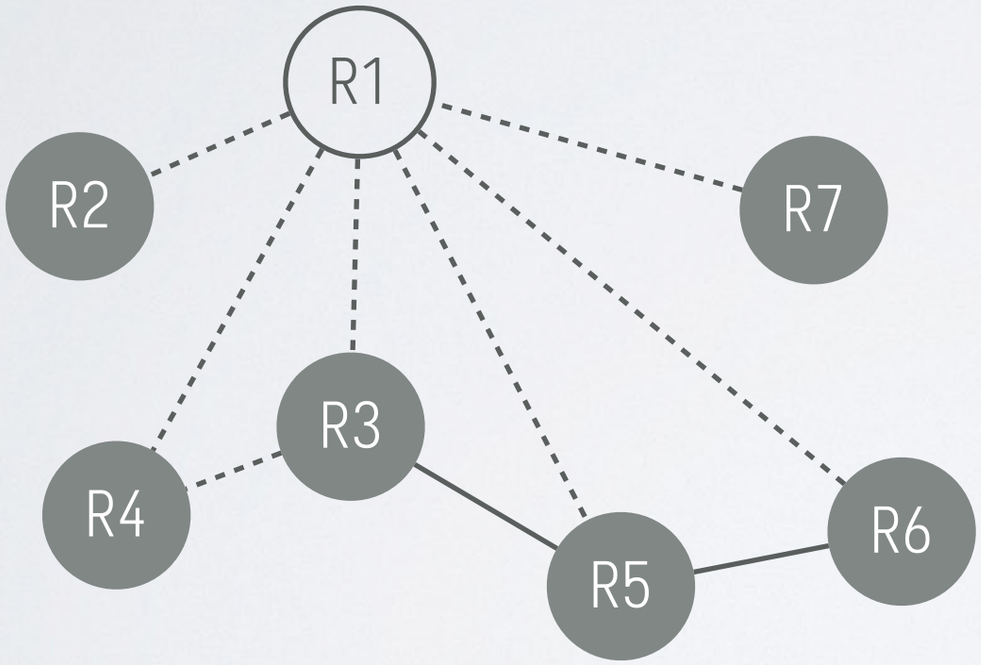


$m = 2$



栈

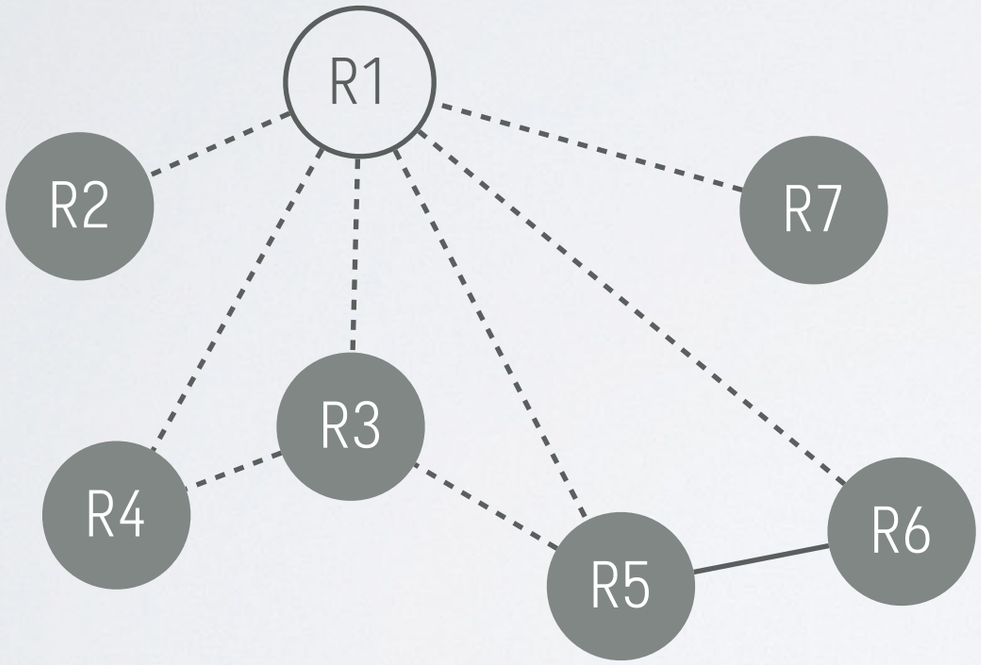
溢出的例子（4）



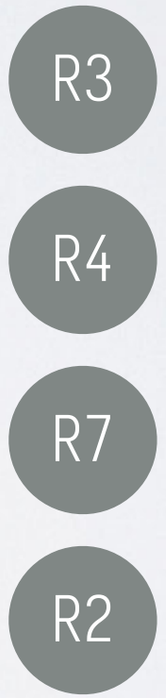
$m = 2$

栈

溢出的例子（5）

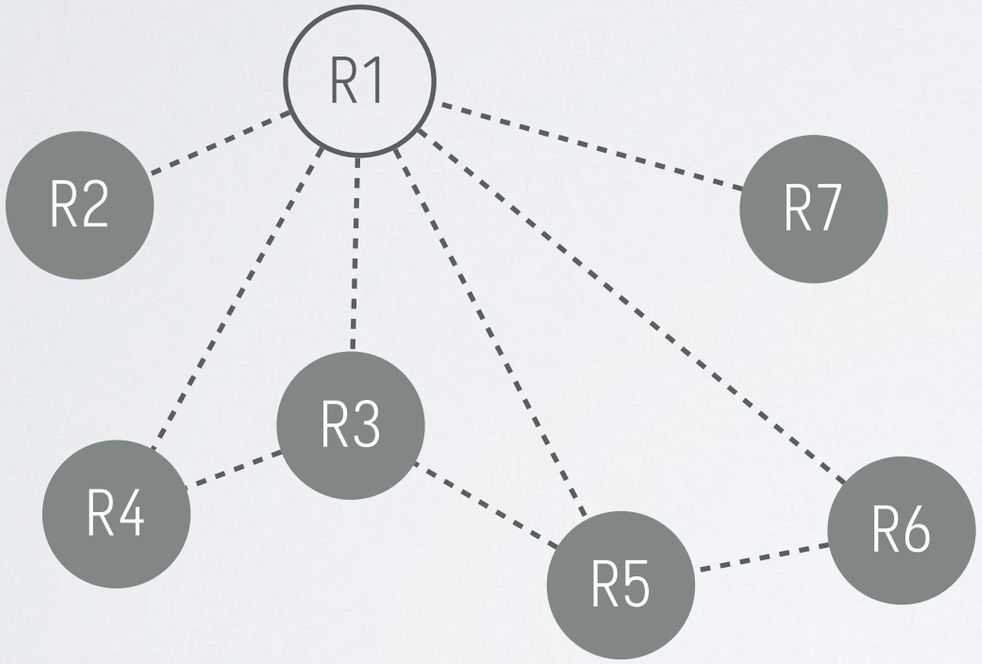


$m = 2$

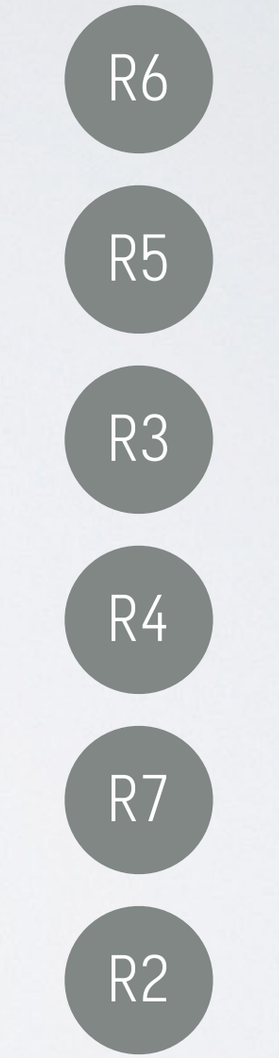


栈

溢出的例子（6）

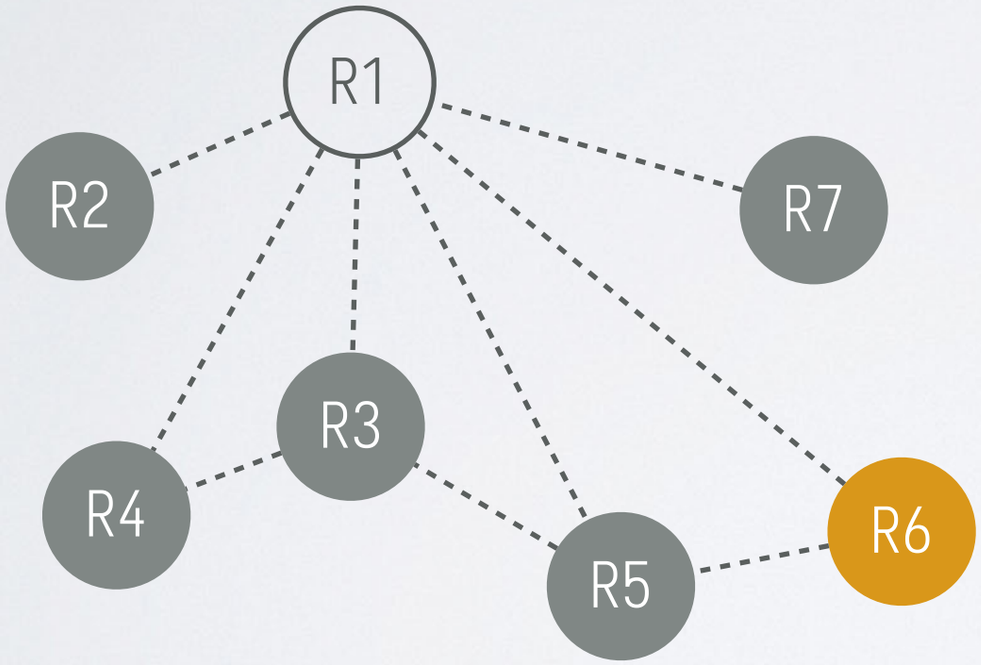


$m = 2$



栈

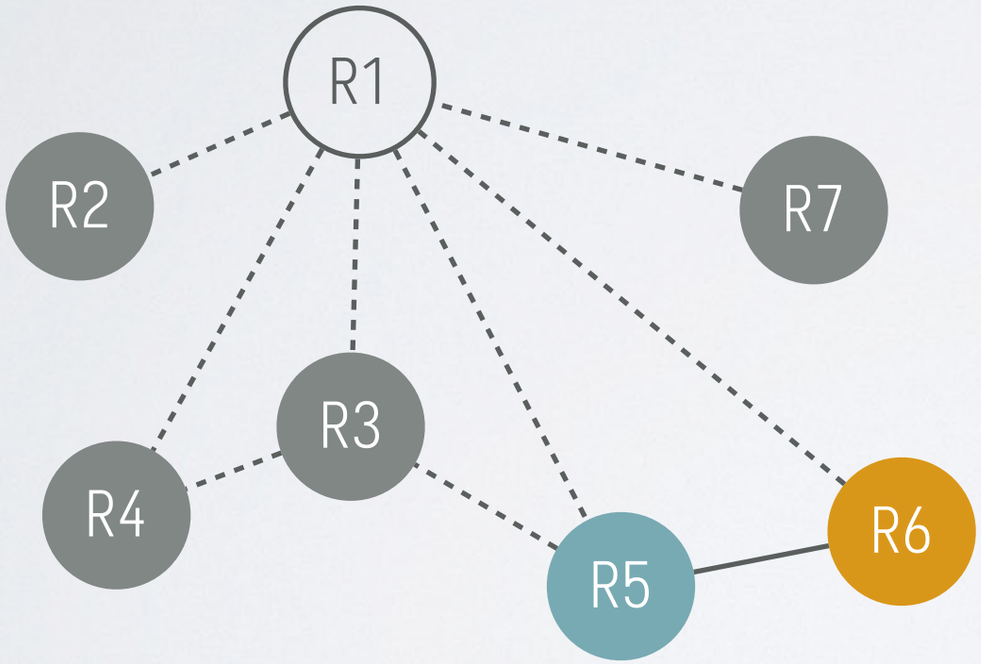
溢出的例子（7）



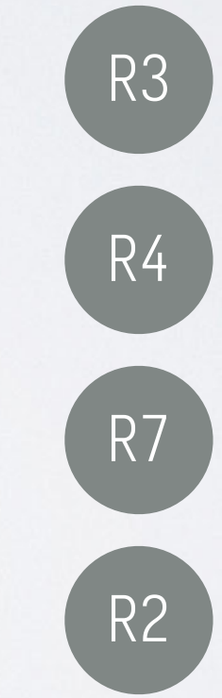
$m = 2$

栈

溢出的例子（8）

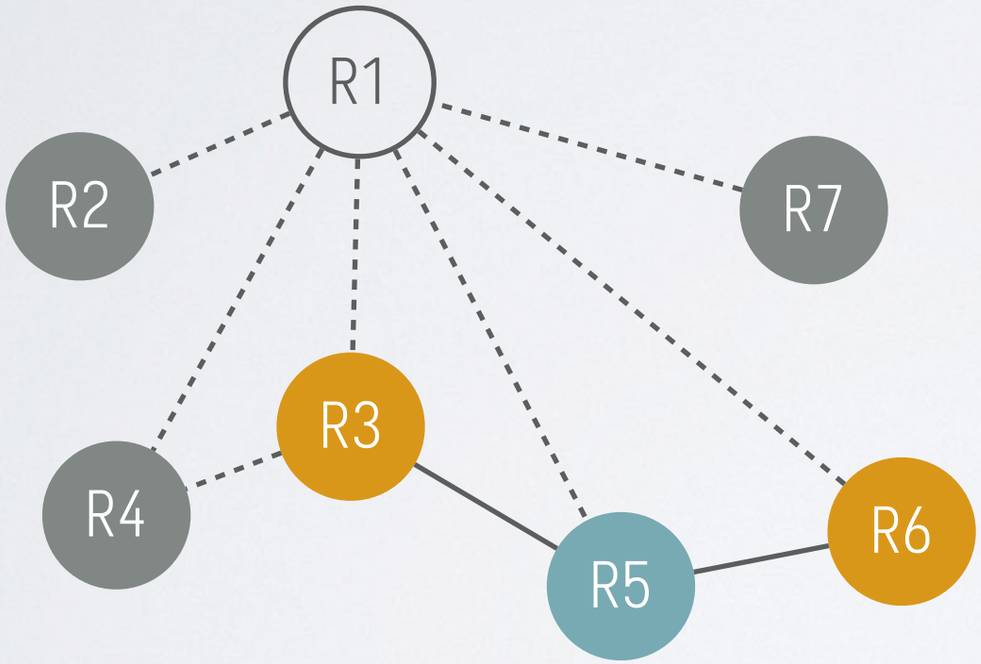


$m = 2$

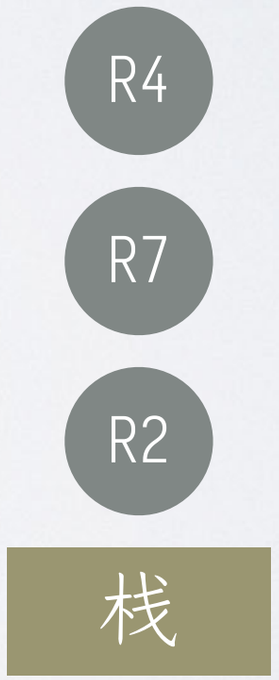


栈

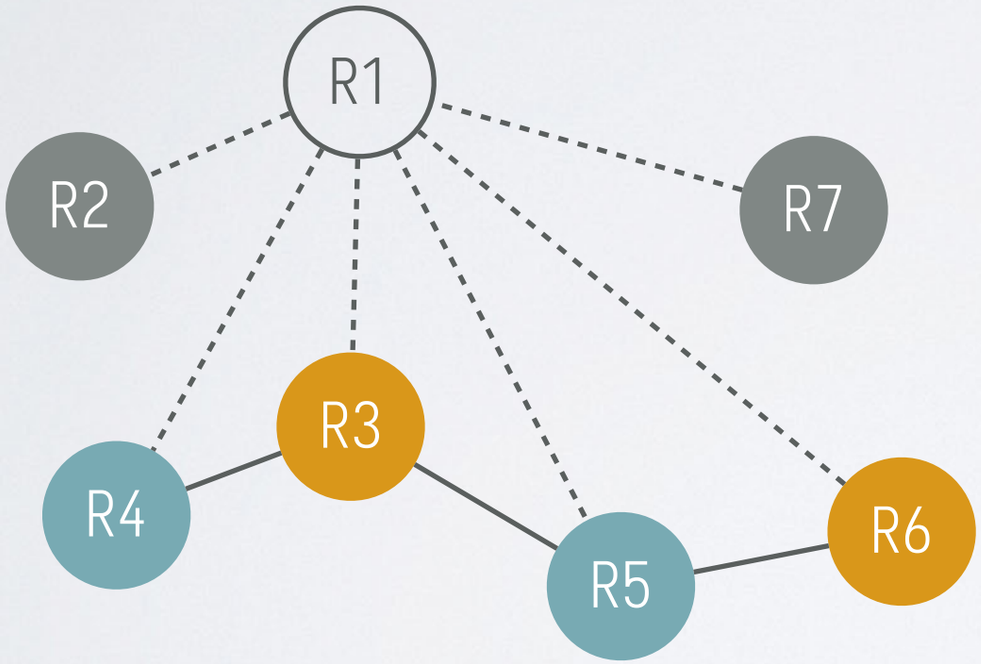
溢出的例子（9）



$m = 2$

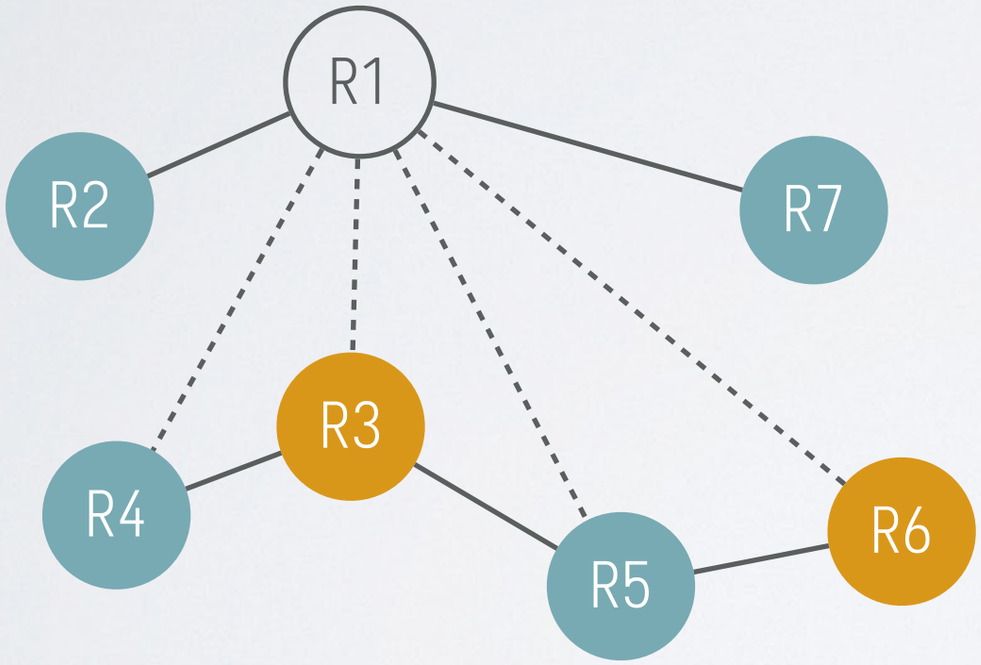


溢出的例子（10）



$m = 2$

溢出的例子（11）



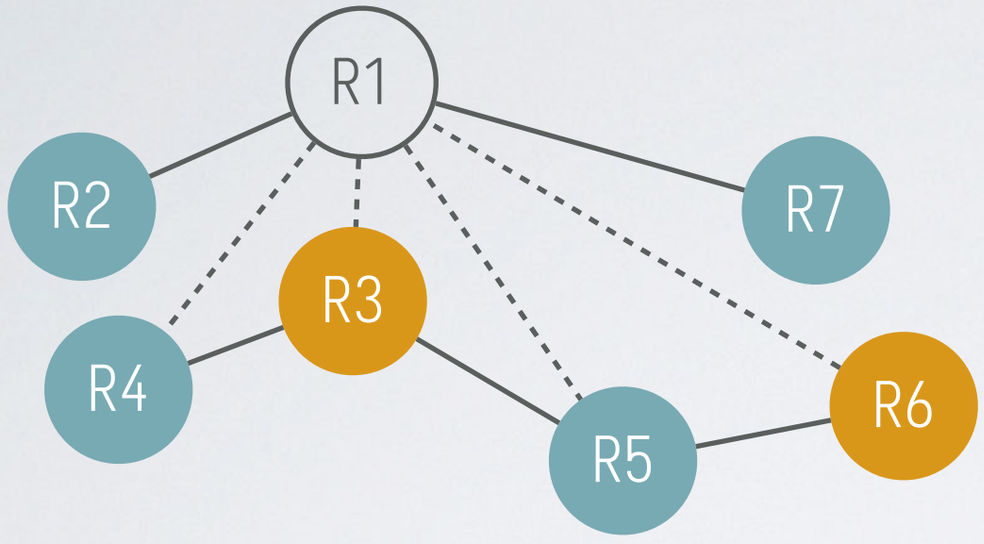
$m = 2$

栈

溢出之后怎么办?

- 溢出结点对应的符号寄存器存放在内存中(比如栈上)
- **问题: 使用这些数据运算的时候仍需要加载到寄存器中**
- 解决方案 1: 预留一些寄存器来进行运算
  - 比较浪费
- **解决方案 2:**
  - 为溢出结点生成代码, 使用时加载到新的符号寄存器中
  - 然后对新的代码**重新**进行活跃性分析和寄存器分配

溢出情况下的代码生成（1）



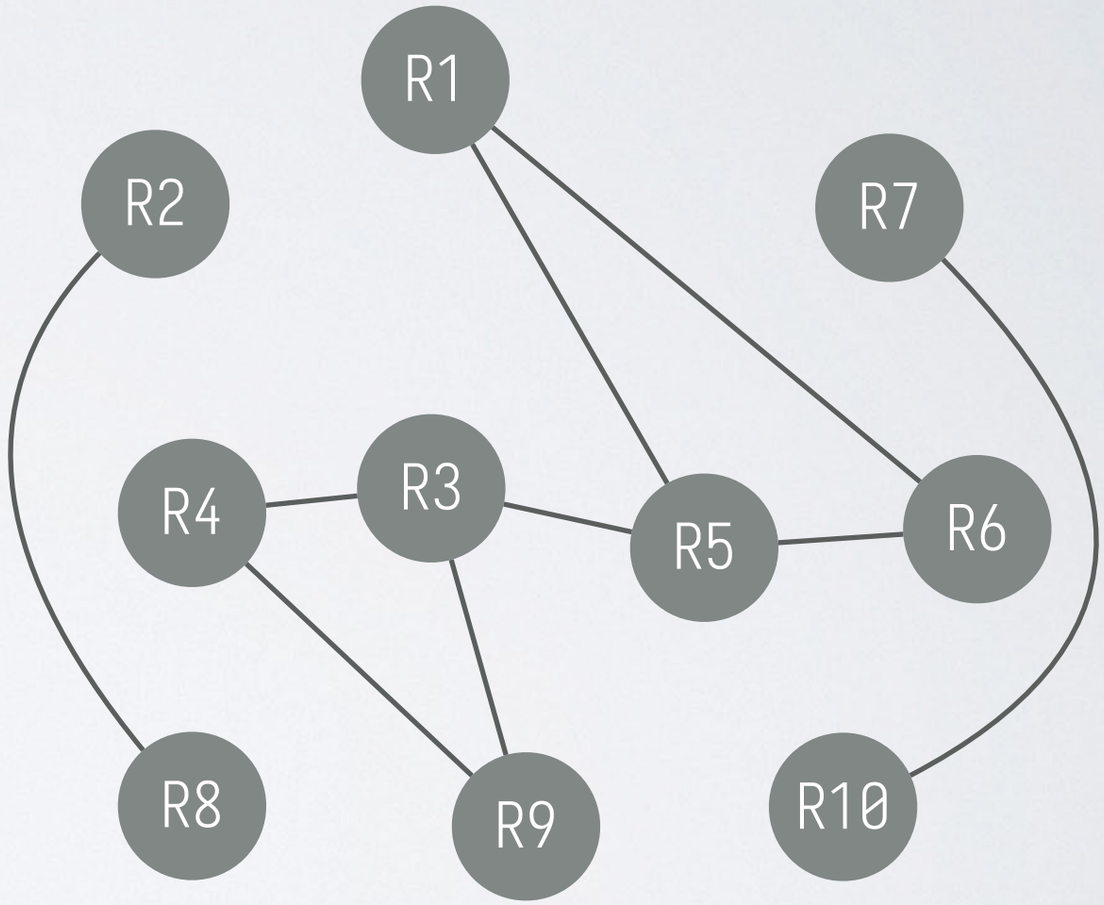

```
LD  R1, a
LD  R2, b
SUB R3, R1, R2
LD  R4, c
SUB R5, R1, R4
ADD R6, R3, R5
LD  R1, d
ADD R7, R6, R5
ST  a, R1
ST  d, R7
```

```
LD  R1, a
ST  4(SP), R1
LD  R2, b
LD  R8, 4(SP)
SUB R3, R8, R2
LD  R4, c
LD  R9, 4(SP)
SUB R5, R9, R4
ADD R6, R3, R5
LD  R1, d
ST  4(SP), R1
ADD R7, R6, R5
LD  R10, 4(SP)
ST  a, R10
ST  d, R7
```

溢出情况下的代码生成（2）

```
LD  R1, a
ST  4(SP), R1
LD  R2, b
LD  R8, 4(SP)
SUB R3, R8, R2
LD  R4, c
LD  R9, 4(SP)
SUB R5, R9, R4
ADD R6, R3, R5
LD  R1, d
ST  4(SP), R1
ADD R7, R6, R5
LD  R10, 4(SP)
ST  a, R10
ST  d, R7
```

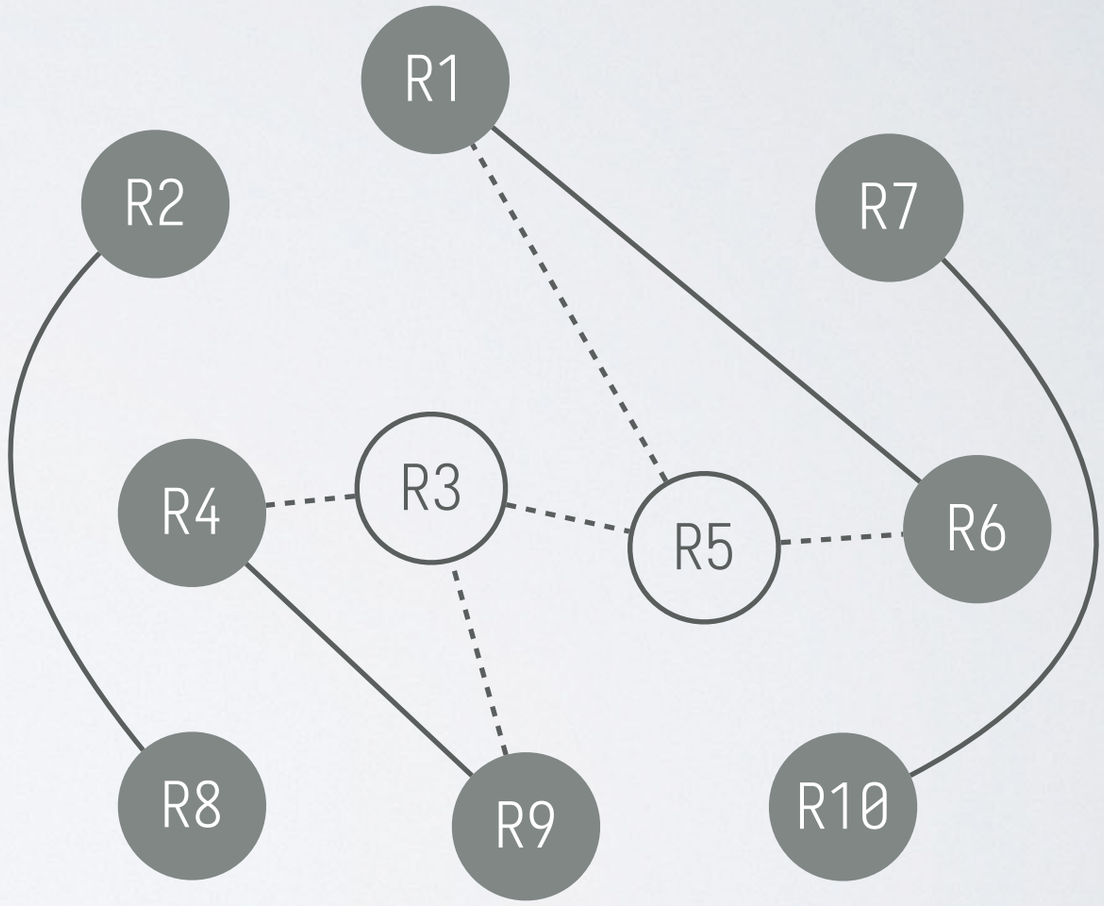
{}
{R1}
{}
{R2}
{R2,R8}
{R3}
{R3,R4}
{R3,R4,R9}
{R3,R5}
{R5,R6}
{R1,R5,R6}
{R5,R6}
{R7}
{R7,R10}
{R7}
{}



溢出情况下的代码生成（3）

```
LD  R1, a          {}
ST  4(SP), R1      {R1}
LD  R2, b          {}
LD  R8, 4(SP)      {R2}
SUB R3, R8, R2     {R2,R8}
LD  R4, c          {R3}
LD  R9, 4(SP)      {R3,R4}
SUB R5, R9, R4     {R3,R4,R9}
ADD R6, R3, R5     {R3,R5}
LD  R1, d          {R5,R6}
ST  4(SP), R1      {R1,R5,R6}
ADD R7, R6, R5     {R5,R6}
LD  R10, 4(SP)     {R7}
ST  a, R10         {R7,R10}
ST  d, R7          {R7}
                   {}
```

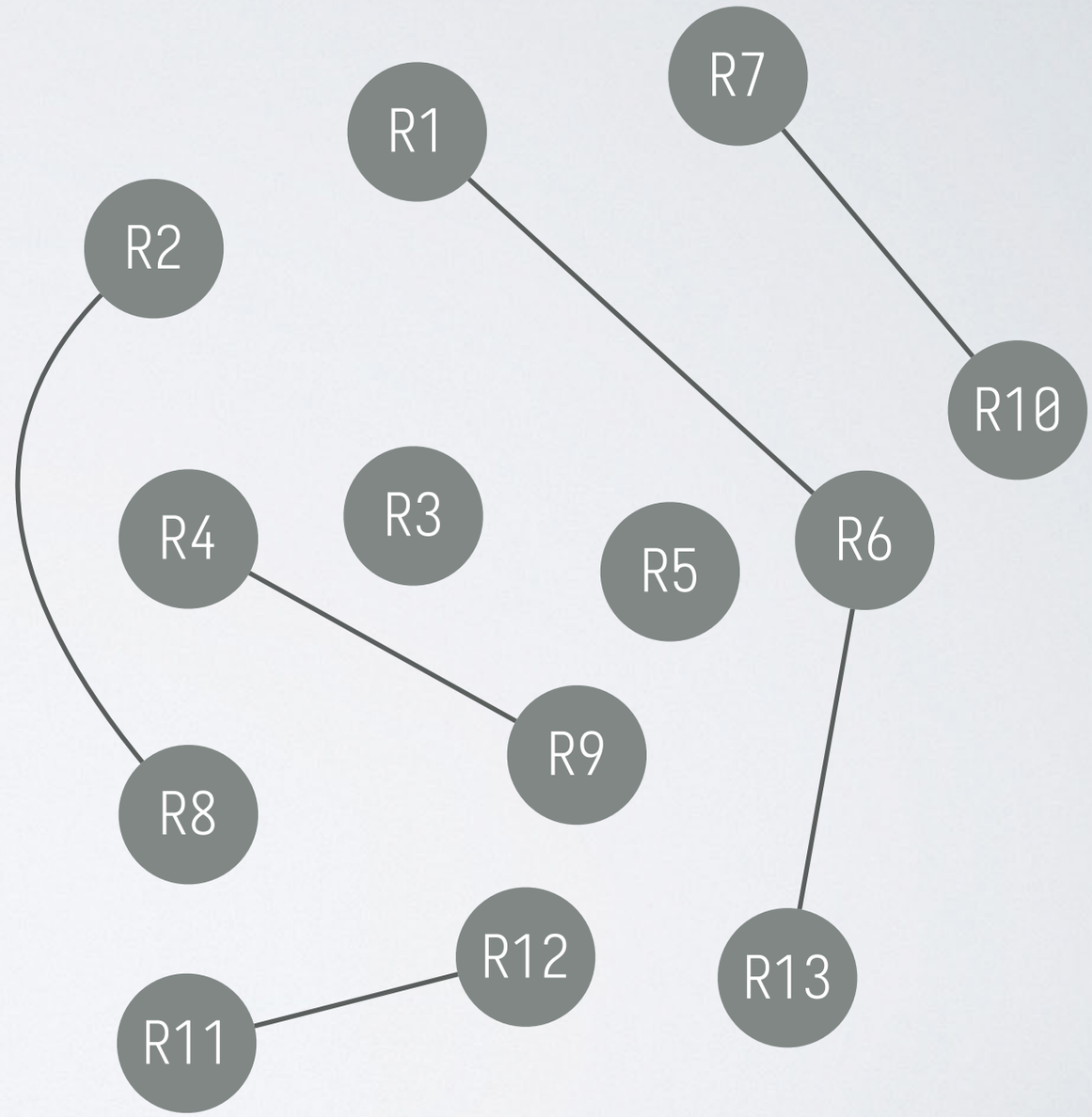
溢出 R3 和 R5



溢出情况下的代码生成（4）

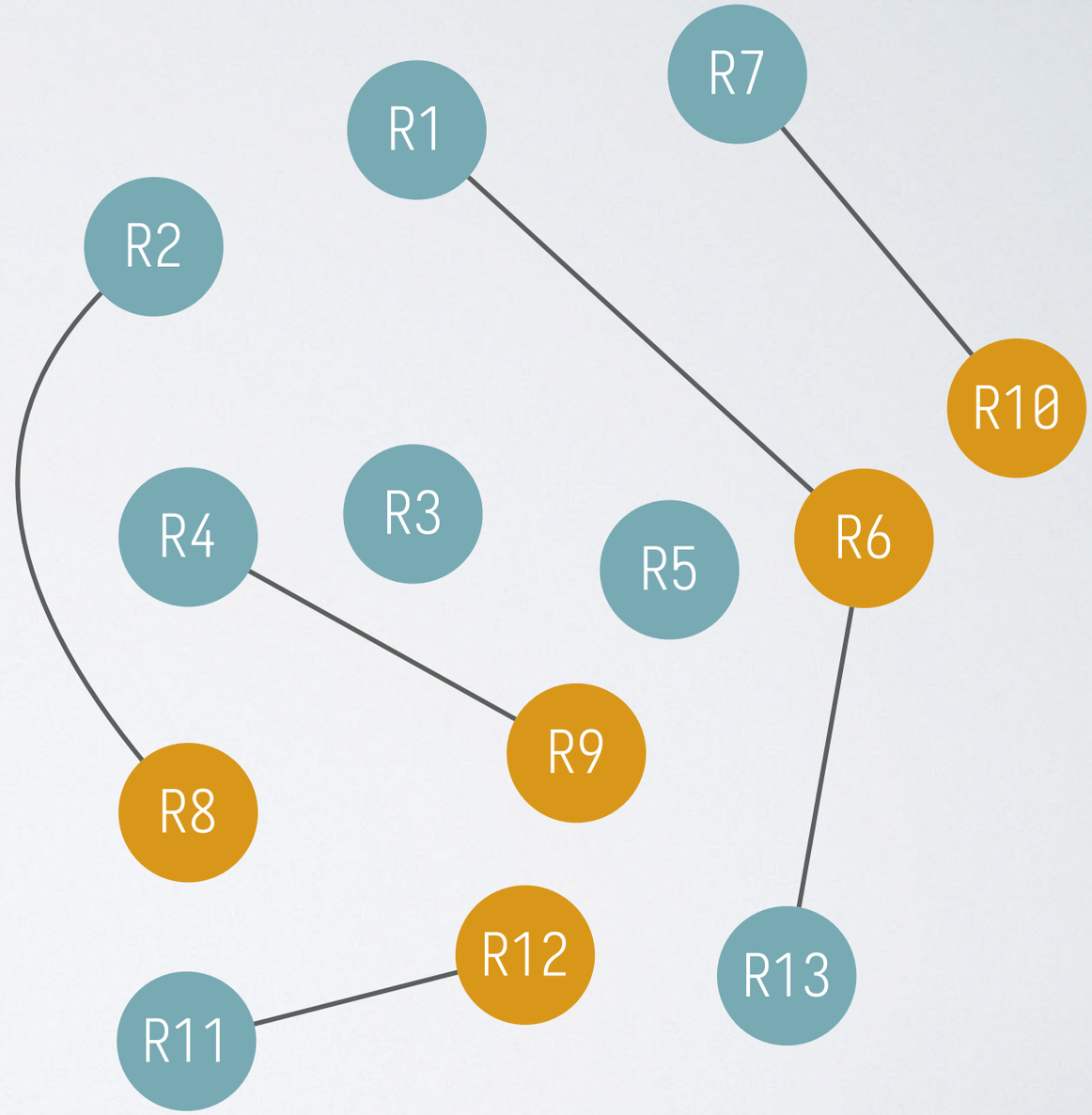
```
LD  R1, a
ST  4(SP), R1
LD  R2, b
LD  R8, 4(SP)
SUB R3, R8, R2
ST  8(SP), R3
LD  R4, c
LD  R9, 4(SP)
SUB R5, R9, R4
ST  12(SP), R5
LD  R11, 8(SP)
LD  R12, 12(SP)
ADD R6, R11, R12
LD  R1, d
ST  4(SP), R1
LD  R13, 12(SP)
ADD R7, R6, R13
LD  R10, 4(SP)
ST  a, R10
ST  d, R7
```

```
{}
{R1}
{}
{R2}
{R2,R8}
{R3}
{}
{R4}
{R4,R9}
{R5}
{}
{R11}
{R11,R12}
{R6}
{R1,R6}
{R6}
{R6,R13}
{R7}
{R7,R10}
{R7}
{}
```



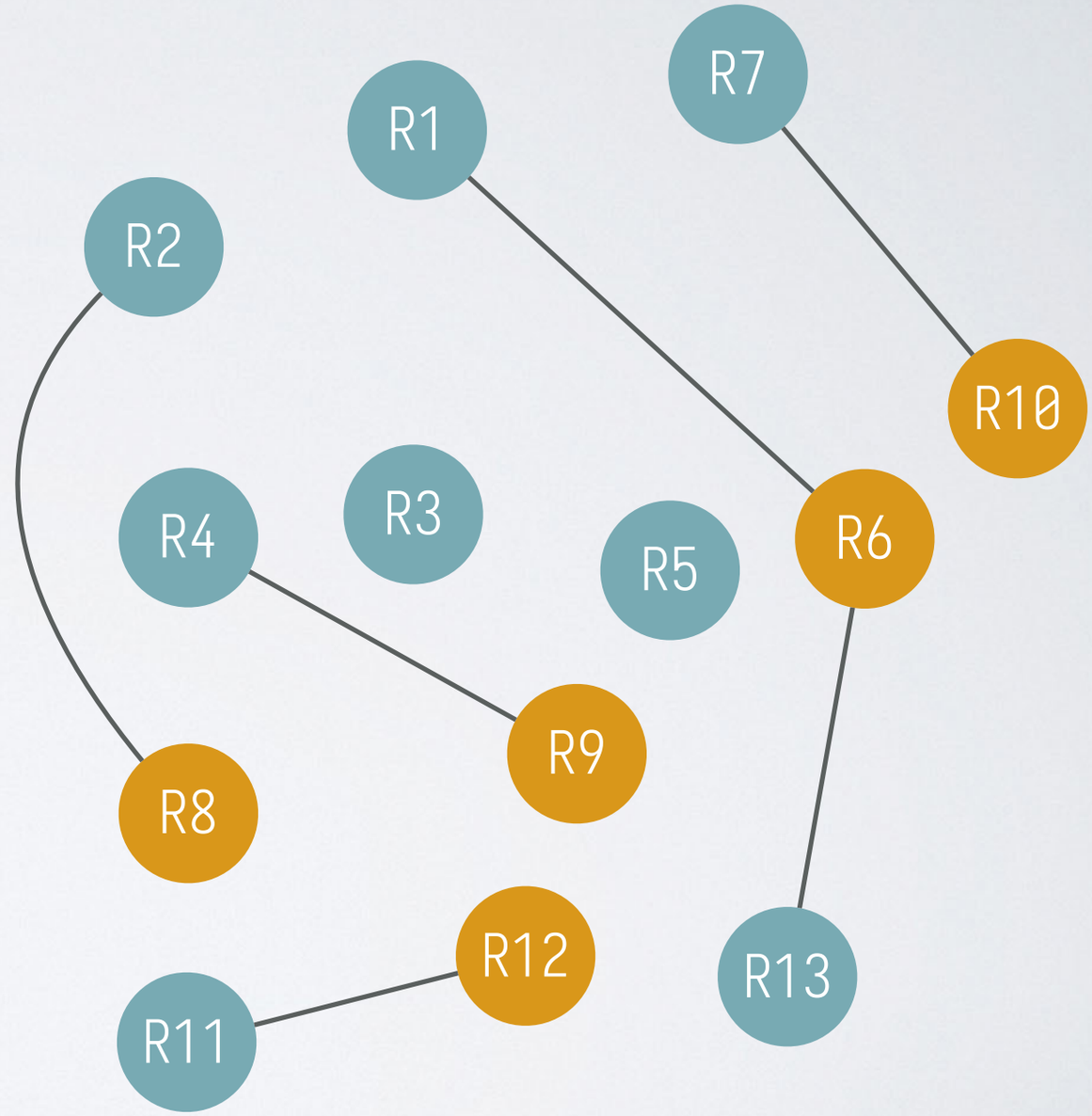
溢出情况下的代码生成（5）

```
LD  R1, a
ST  4(SP), R1
LD  R2, b
LD  R8, 4(SP)
SUB R3, R8, R2
ST  8(SP), R3
LD  R4, c
LD  R9, 4(SP)
SUB R5, R9, R4
ST  12(SP), R5
LD  R11, 8(SP)
LD  R12, 12(SP)
ADD R6, R11, R12
LD  R1, d
ST  4(SP), R1
LD  R13, 12(SP)
ADD R7, R6, R13
LD  R10, 4(SP)
ST  a, R10
ST  d, R7
```

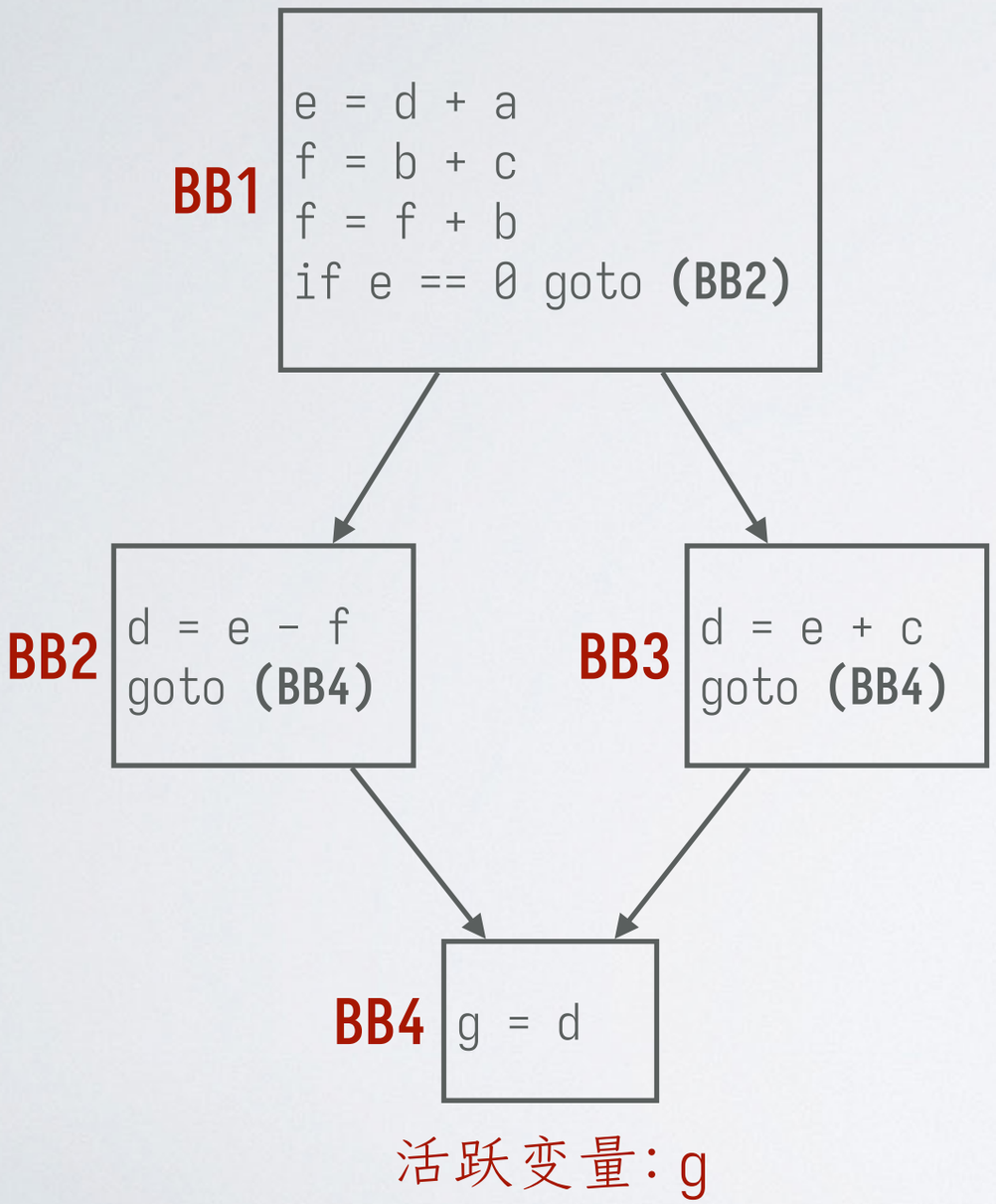


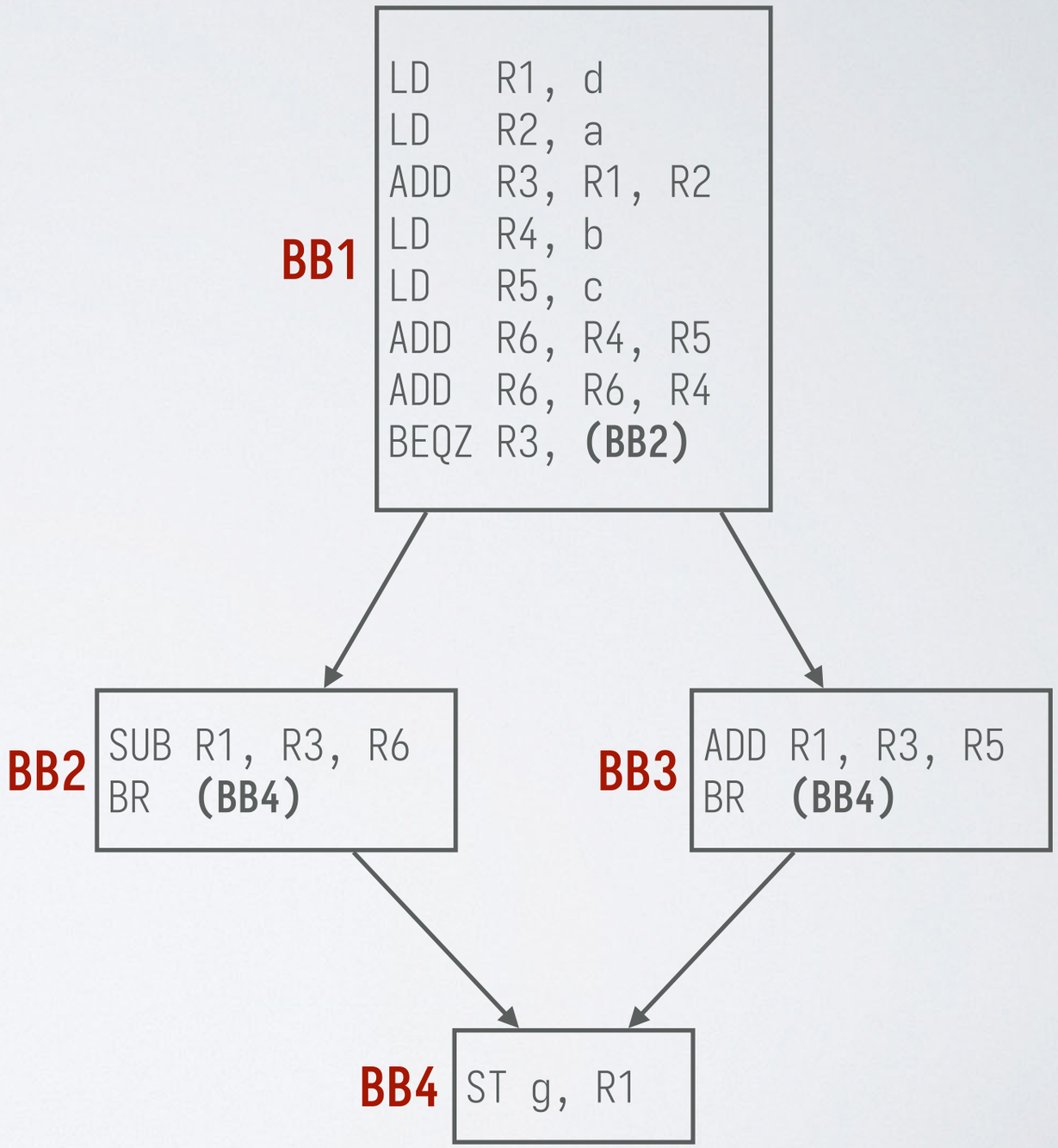
溢出情况下的代码生成（6）

```
LD  R1, a
ST  4(SP), R1
LD  R1, b
LD  R2, 4(SP)
SUB R1, R2, R1
ST  8(SP), R1
LD  R1, c
LD  R2, 4(SP)
SUB R1, R2, R1
ST  12(SP), R1
LD  R1, 8(SP)
LD  R2, 12(SP)
ADD R2, R1, R2
LD  R1, d
ST  4(SP), R1
LD  R1, 12(SP)
ADD R1, R2, R1
LD  R2, 4(SP)
ST  a, R2
ST  d, R1
```

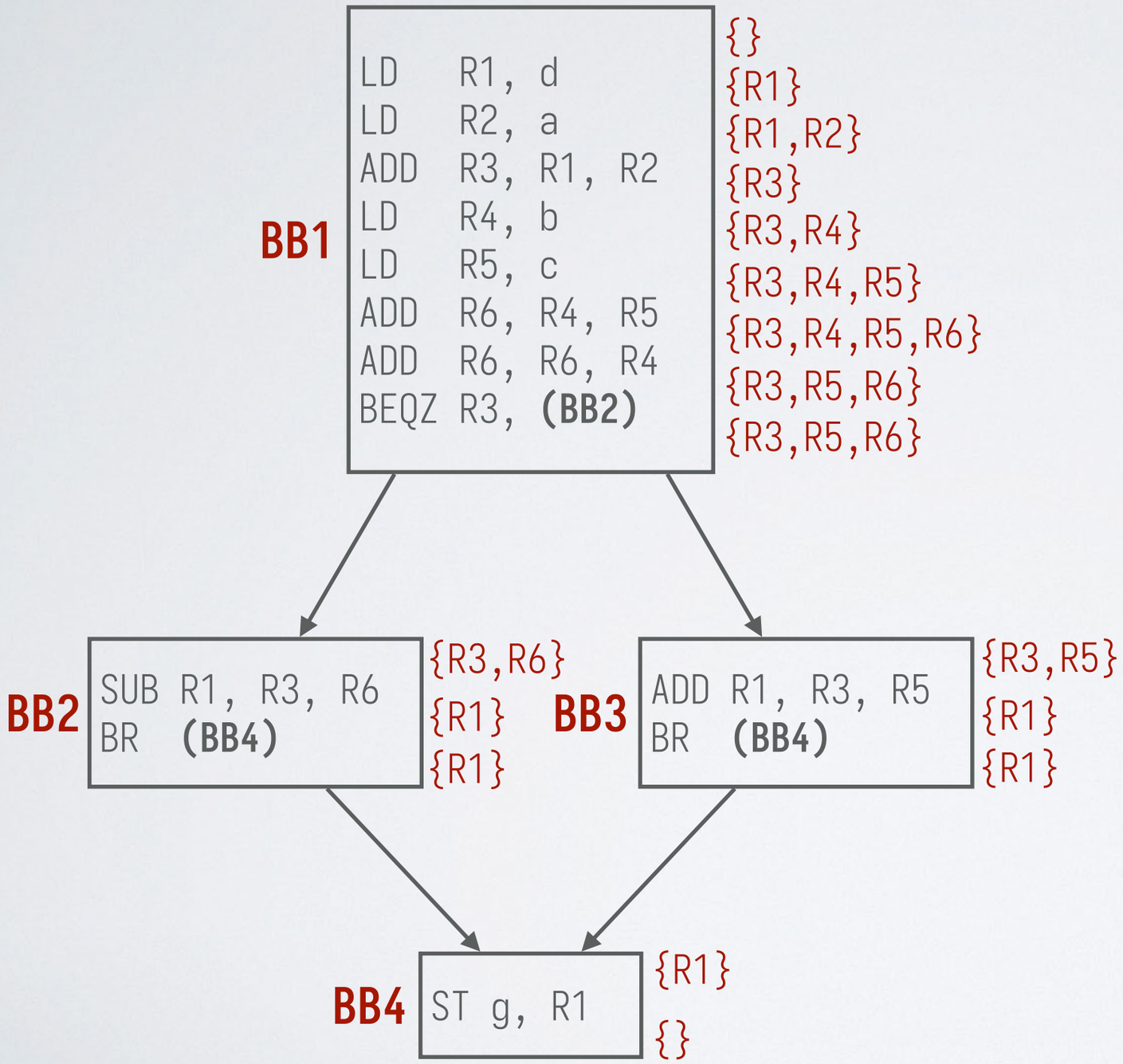


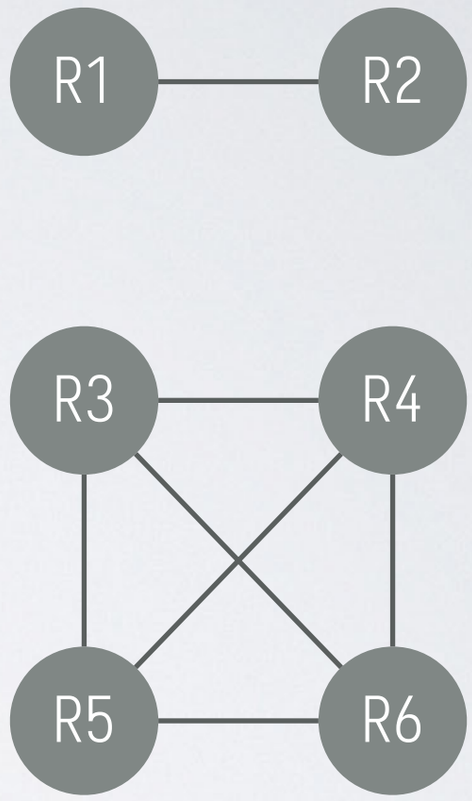
全局寄存器分配的例子（1）





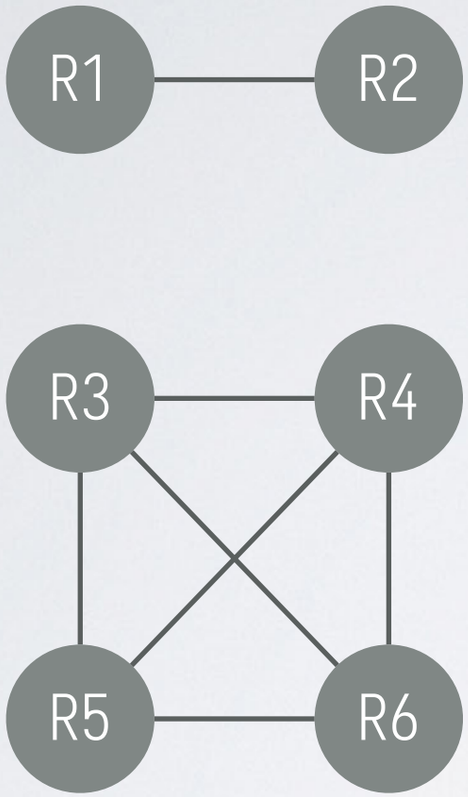
全局寄存器分配的例子（2）





可以 4-着色
不能 3-着色

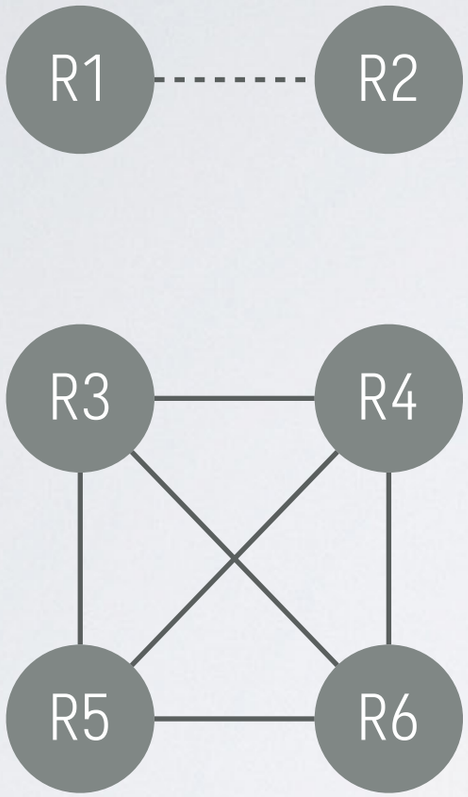
全局寄存器分配的例子（3）



尝试 3-着色

栈

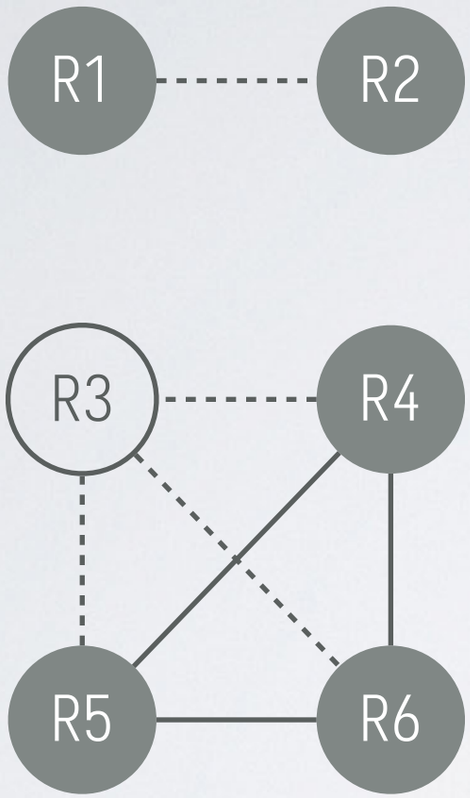
全局寄存器分配的例子（4）



尝试 3-着色

栈

全局寄存器分配的例子（5）

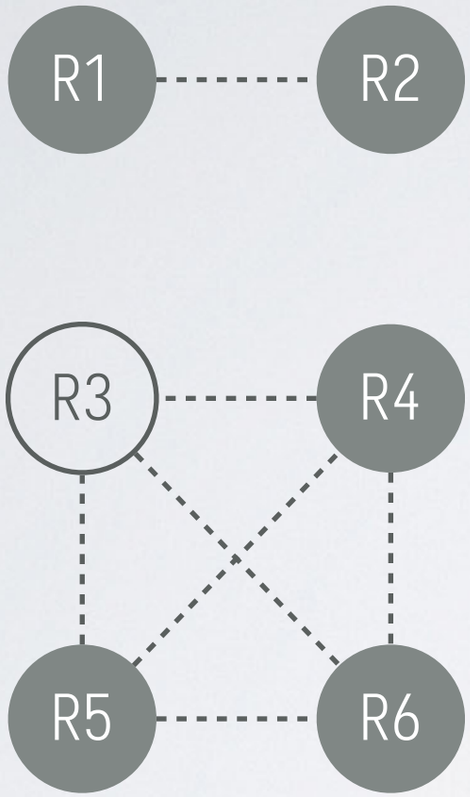


尝试 3-着色

溢出 R3

栈

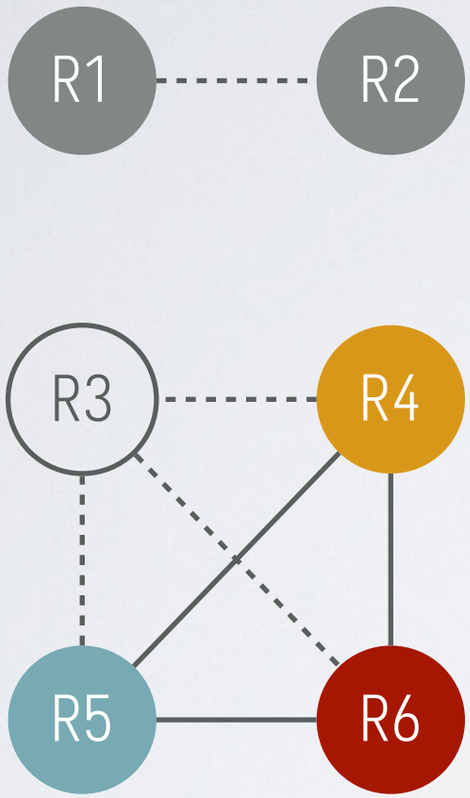
全局寄存器分配的例子（6）



尝试 3-着色

栈

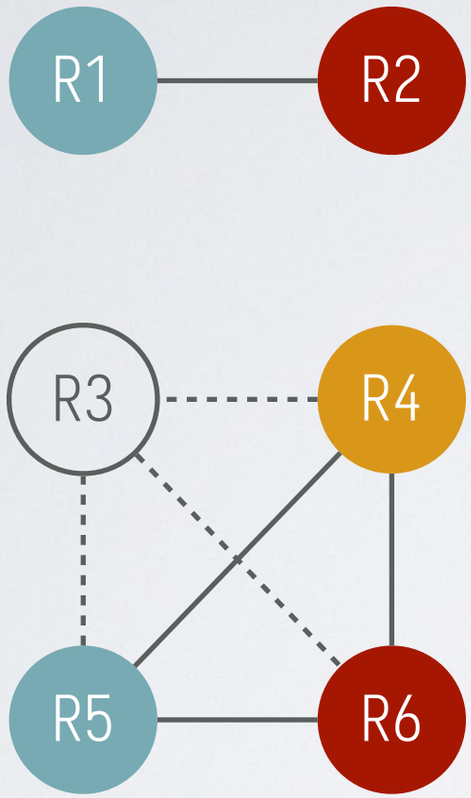
全局寄存器分配的例子（7）



尝试 3-着色

栈

全局寄存器分配的例子（8）



尝试 3-着色

栈

全局寄存器分配的例子（9）

溢出 R3

BB1

```
LD   R1, d
LD   R2, a
ADD  R3, R1, R2
LD   R4, b
LD   R5, c
ADD  R6, R4, R5
ADD  R6, R6, R4
BEQZ R3, (BB2)
```

BB2

```
SUB R1, R3, R6
BR  (BB4)
```

BB3

```
ADD R1, R3, R5
BR  (BB4)
```

BB4

```
ST g, R1
```

BB1

```
LD   R1, d
LD   R2, a
ADD  R3, R1, R2
ST   4(SP), R3
LD   R4, b
LD   R5, c
ADD  R6, R4, R5
ADD  R6, R6, R4
LD   R7, 4(SP)
BEQZ R7, (BB2)
```

BB2

```
LD  R8, 4(SP)
SUB R1, R8, R6
BR  (BB4)
```

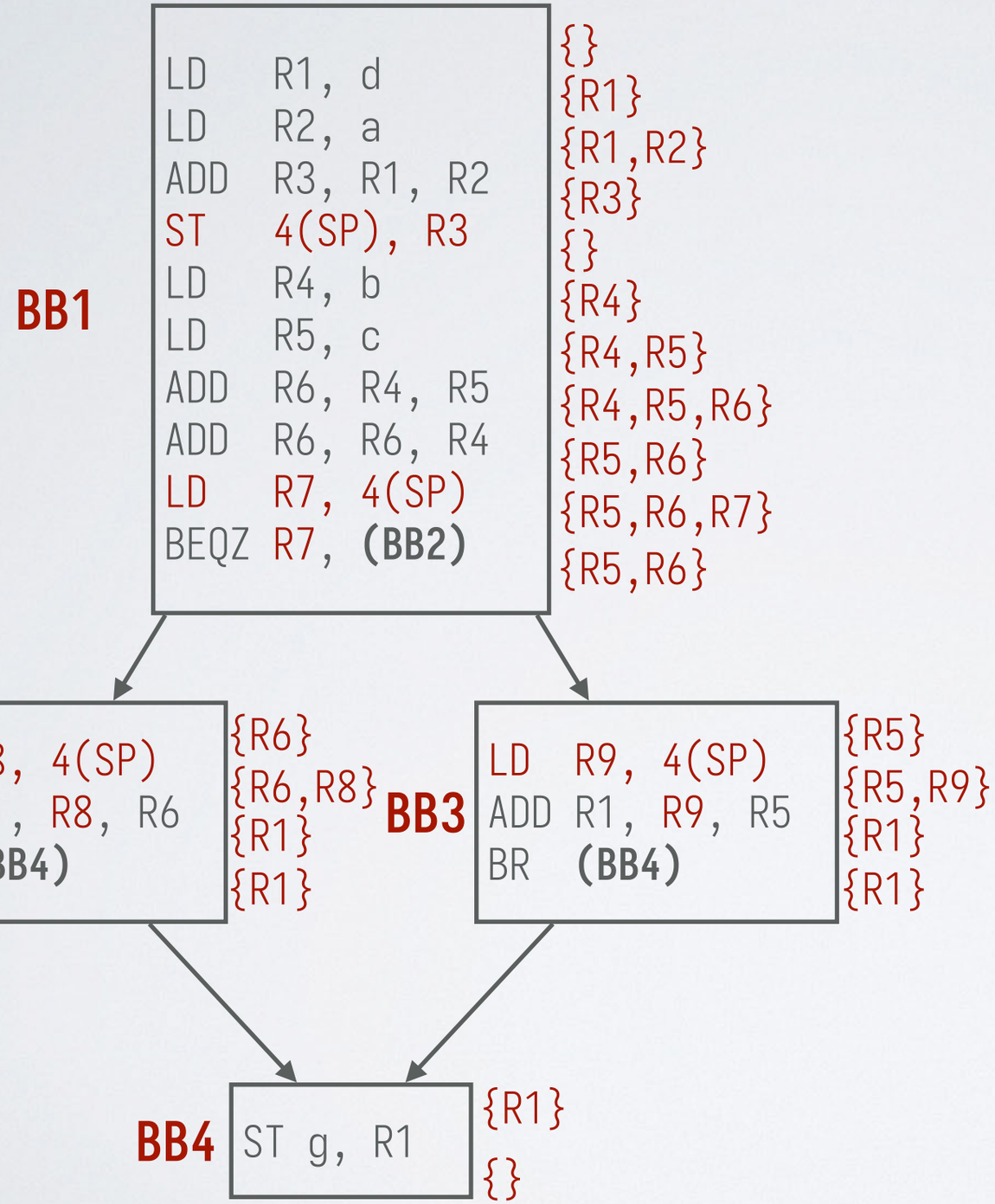
BB3

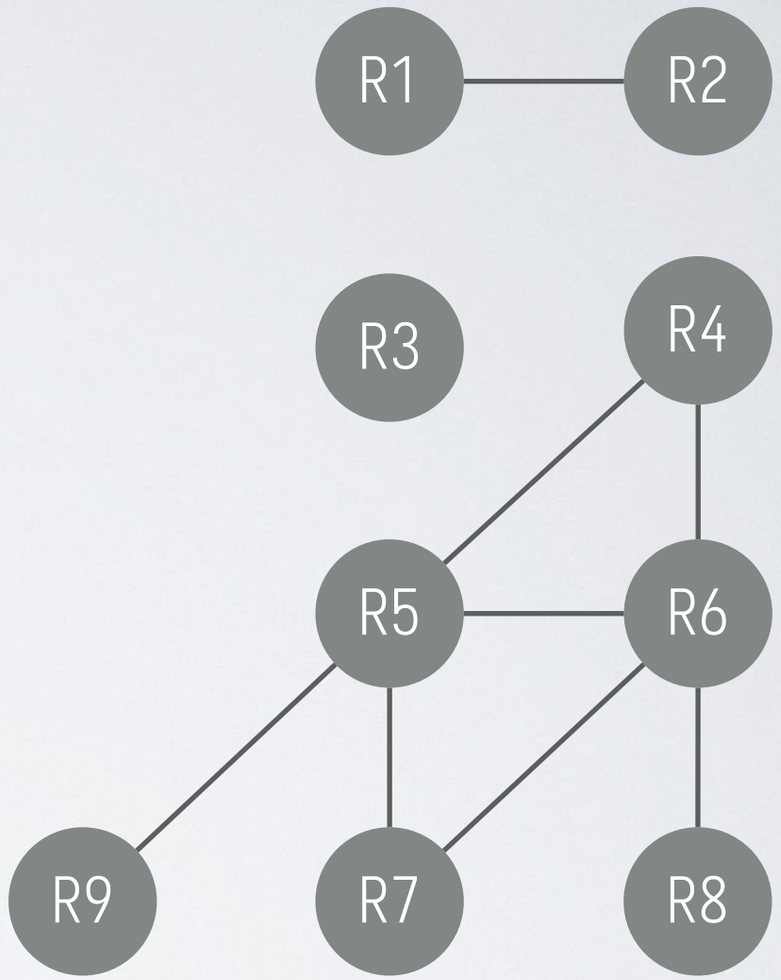
```
LD  R9, 4(SP)
ADD R1, R9, R5
BR  (BB4)
```

BB4

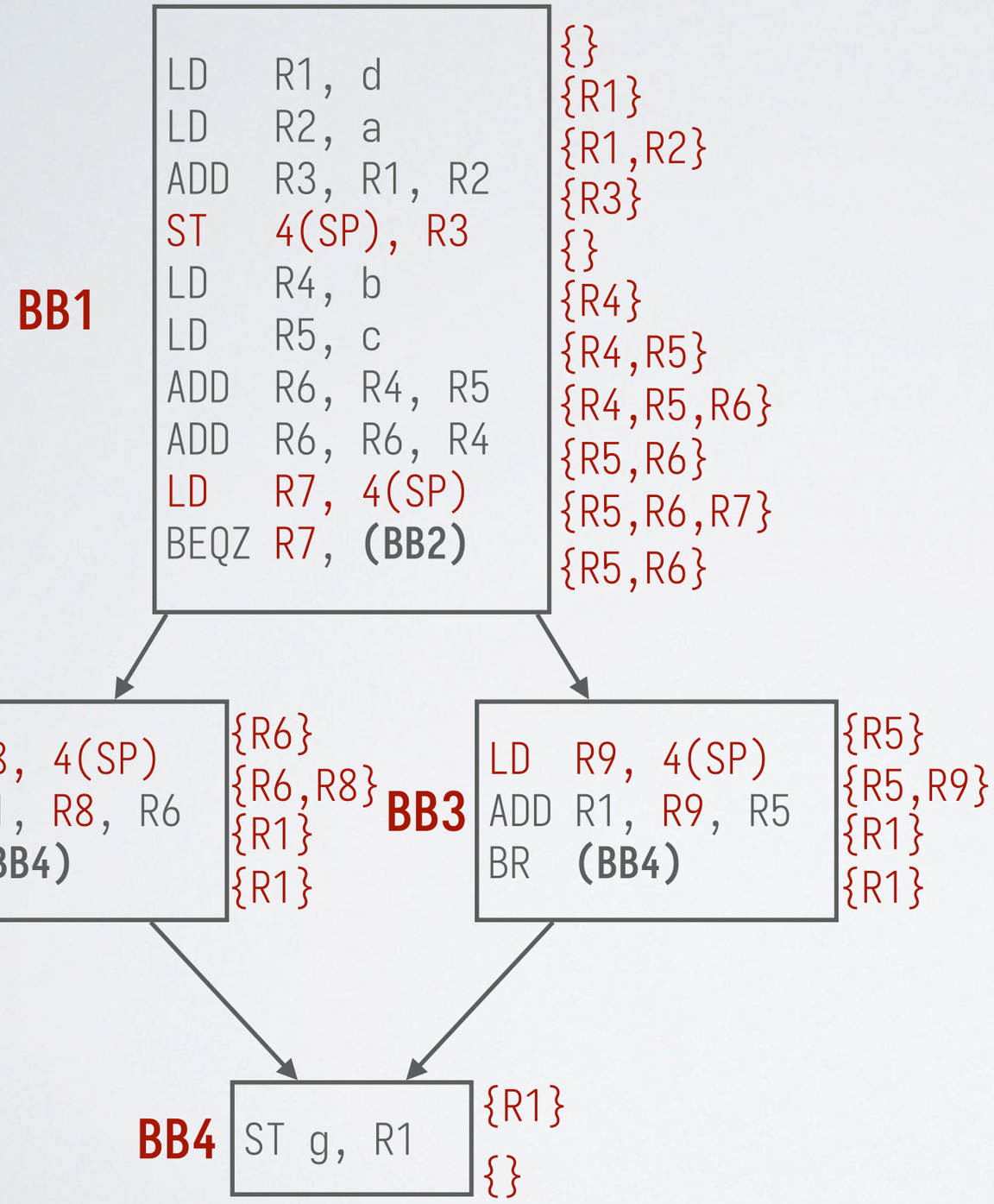
```
ST g, R1
```

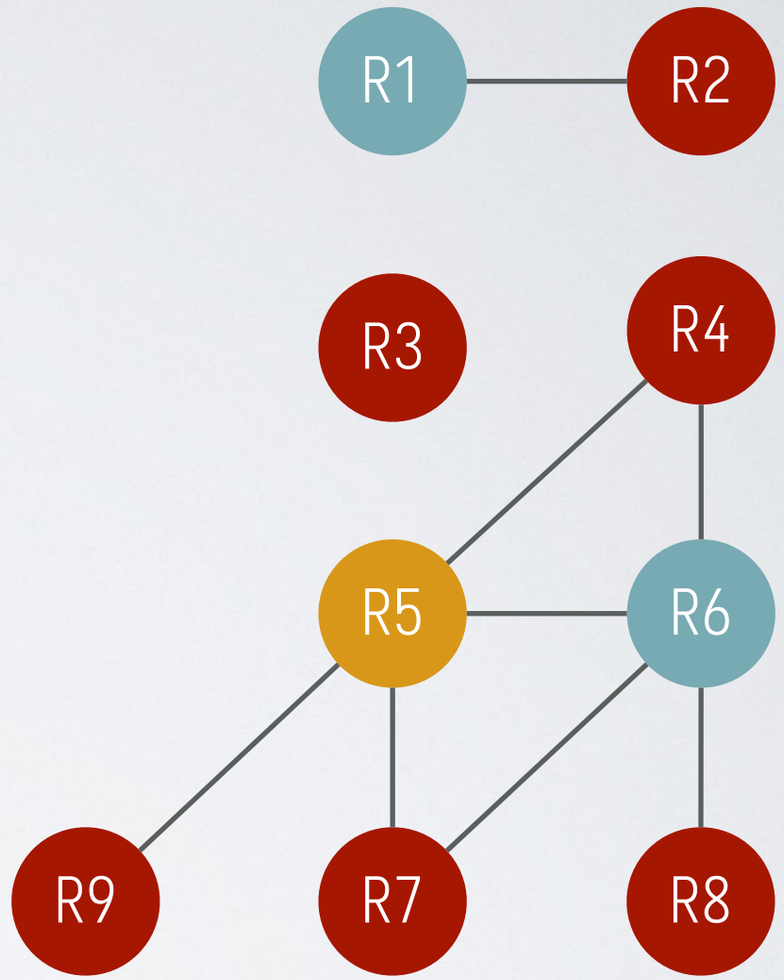
全局寄存器分配的例子（10）





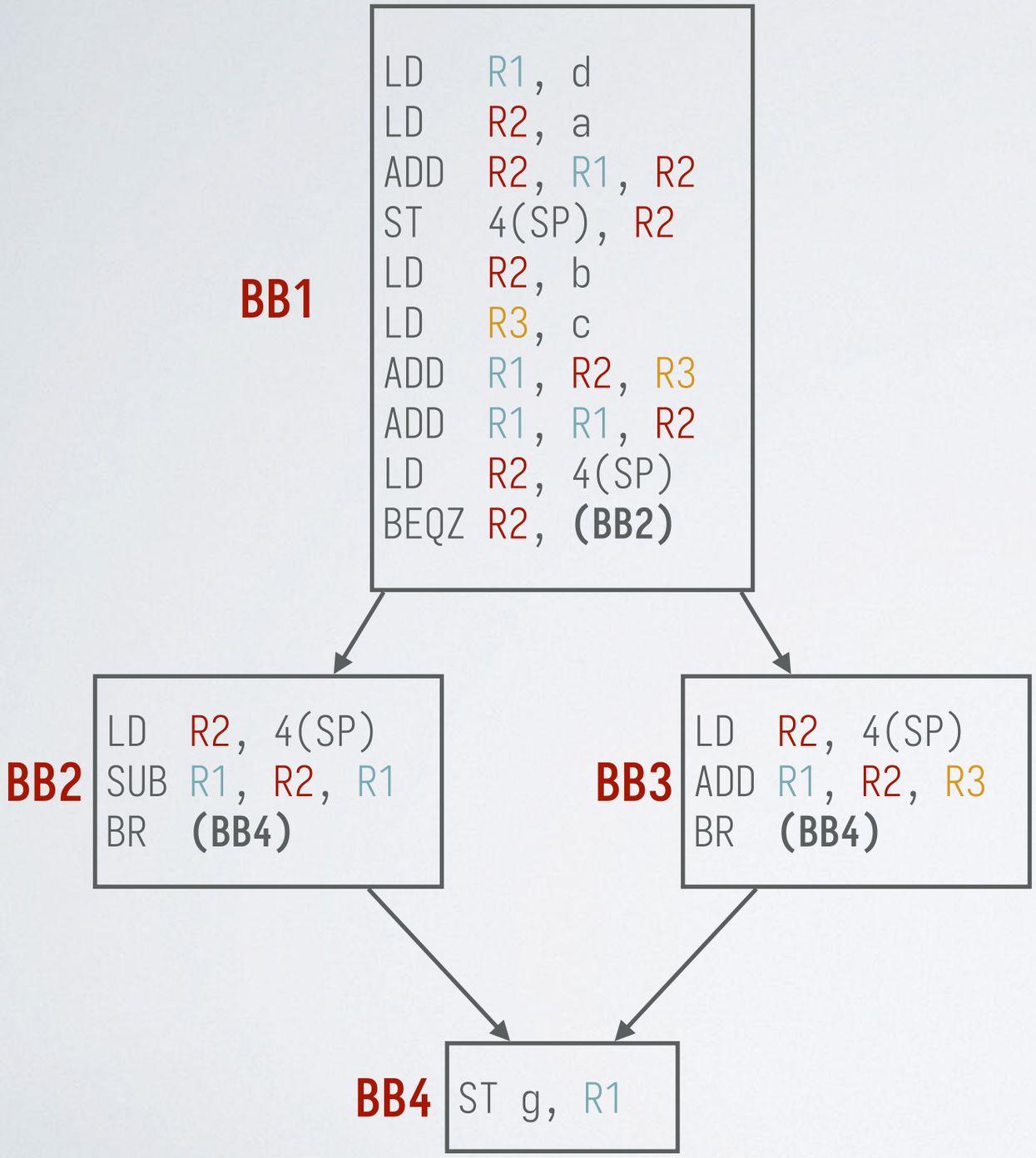
全局寄存器分配的例子（11）

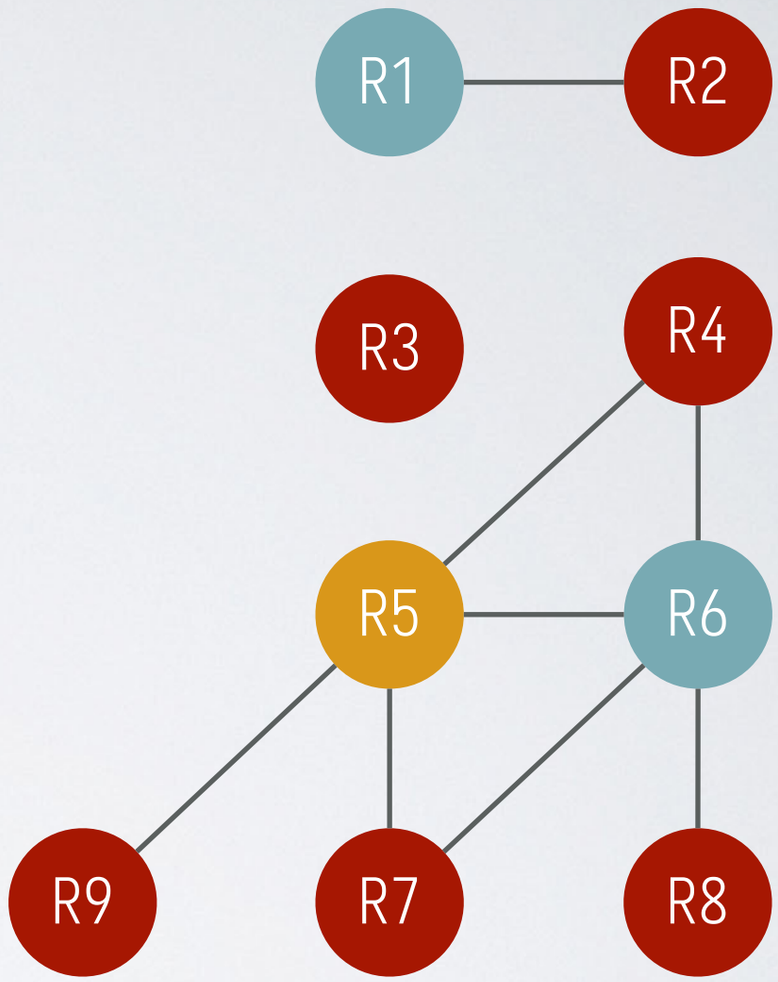




可以 3-着色

全局寄存器分配的例子（12）



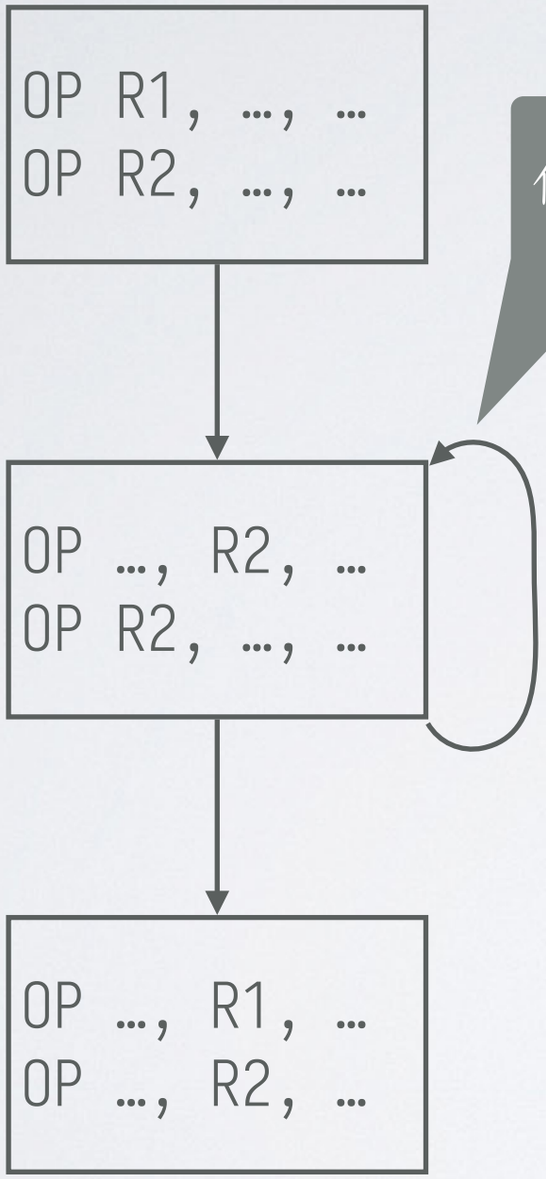


可以 3-着色

如何选择溢出结点？

- 溢出操作会增加额外的加载和保存指令
- 选择溢出结点的原则：避免在循环中引入溢出代码
- **溢出代价(spill cost)**：引入的额外指令的运行时开销
- **问题：如何计算溢出代价？**
  - 无法预计运行时会执行哪个分支，或者循环会执行多少次
  - 体系结构特性(分支预测、缓存等)会影响指令的开销
- **解决方案：使用静态的近似估算**
  - 例如：假设循环会执行 10 次、100 次
  - 选择估算的溢出代价最小的结点

估算溢出代价的例子

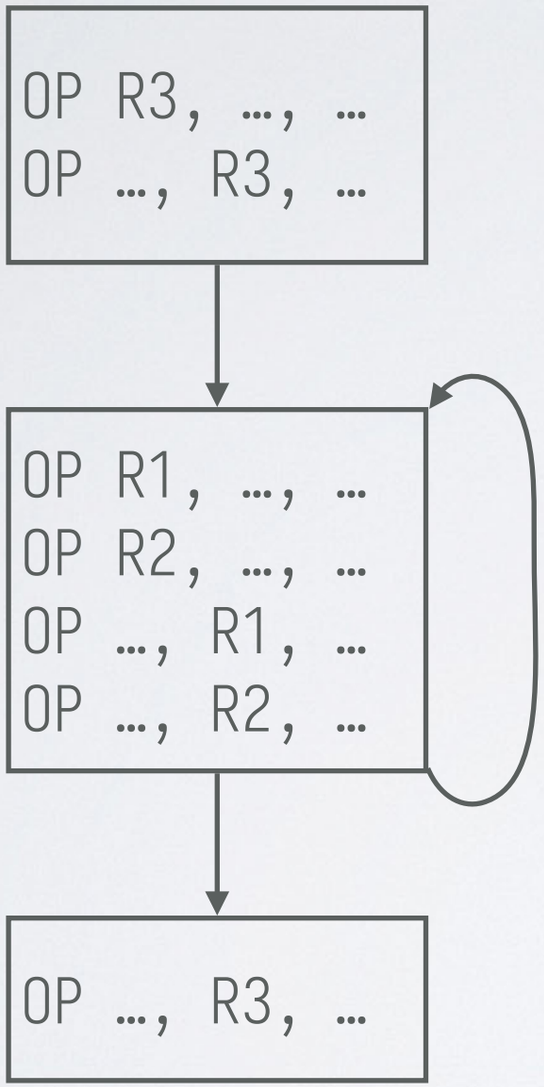


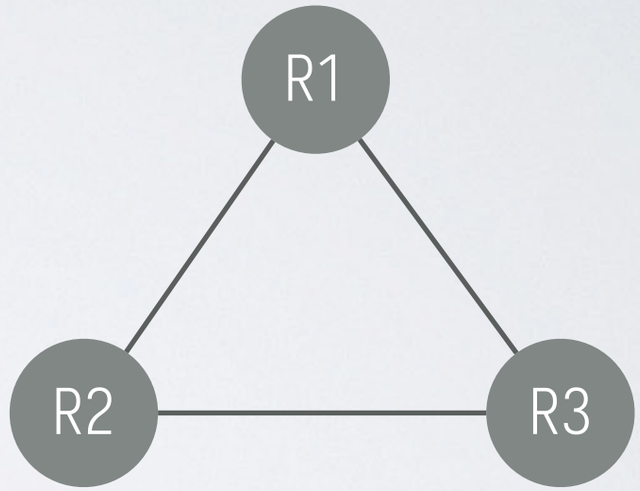
- R1 的溢出代价：
  - $storeCost + loadCost$
- R2 的溢出代价：
  - $11 \cdot storeCost + 11 \cdot loadCost$
- 问：如果只有一个寄存器，应该选择哪个结点溢出？

拆分（splitting）

- 每次对溢出结点的操作都会导致额外的加载/保存指令
- 对一个结点的**活跃范围**进行拆分，降低它在冲突图中的度数，使得拆分后的图可能可以进行着色
- **拆分(split)**的方法：
  - 把结点对应寄存器的值保存到内存中
  - 在拆分的地方把值再加载回来

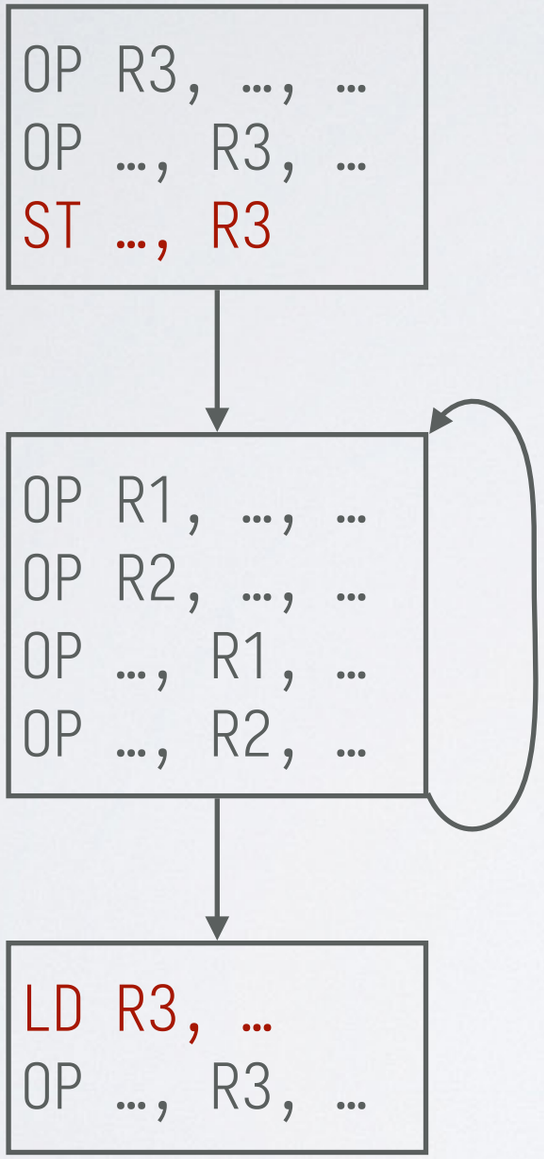
拆分的例子（1）

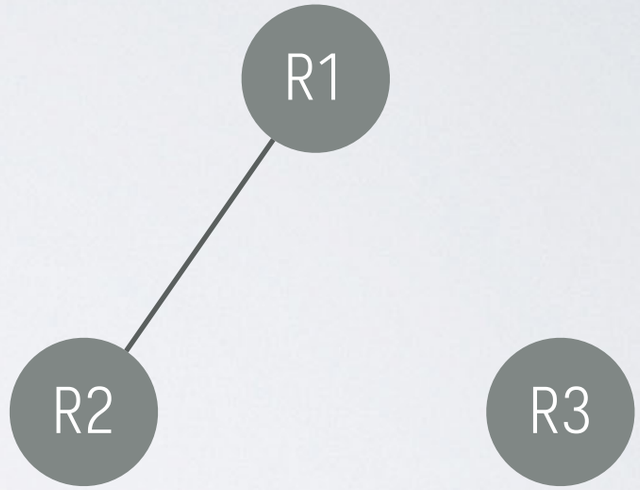




可以 2-着色吗？

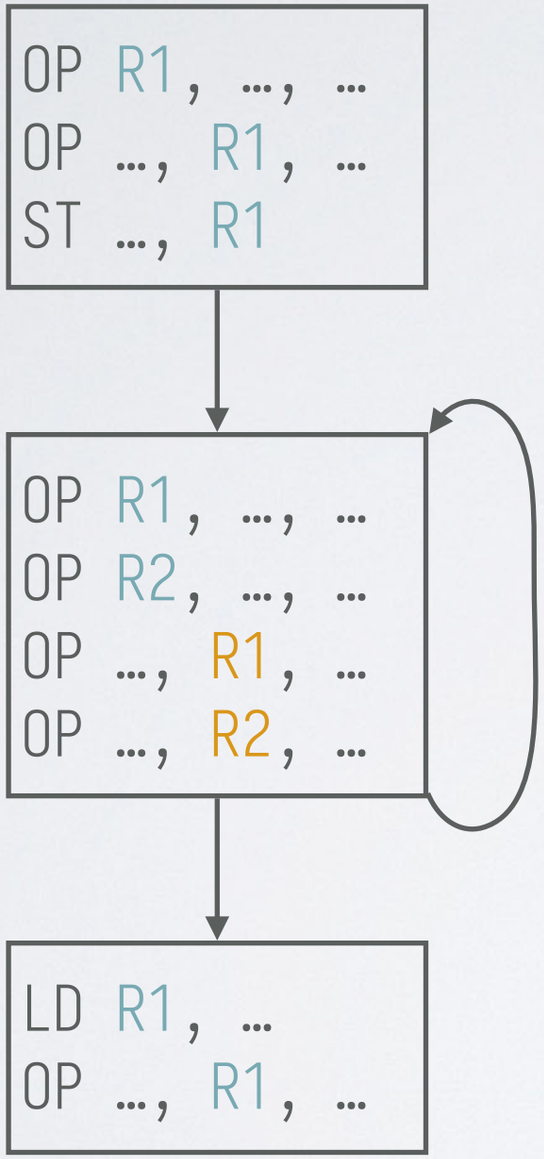
拆分的例子（2）

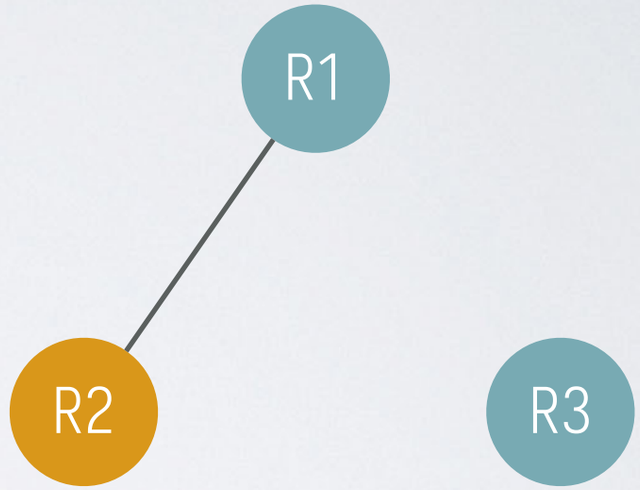




可以 2-着色吗？

拆分的例子（3）





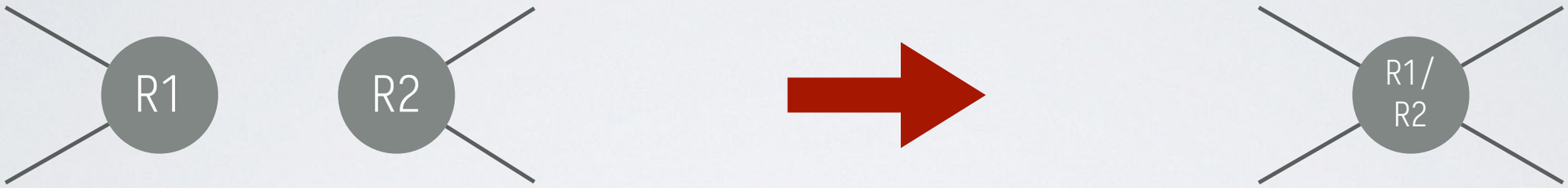
可以 2-着色！

合并（coalescing）

- 代码生成可能会产生大量的寄存器-寄存器拷贝
  - LD R1, R2
  - 如果我们能把 R1 和 R2 分配到同一个物理寄存器中，那么我们就不需要执行这个拷贝
- 思路：如果 R1 和 R2 在冲突图中不相邻的话，那么就可以把它们**合并(coalesce)**成一个符号寄存器

合并（coalescing）

- 问题：合并会增加冲突边的数目，可能导致无法着色



- **解决方案 1：**合并时不要创建高度数（$\geq m$）的结点
- **解决方案 2：**如果 a 的每个邻居 r 都满足下面的条件，才可以把 a 与 b 合并
  - r 与 b 之间有冲突，或者
  - r 的度数比较低（$< m$）

预着色结点（precolored nodes）

- 某些目标机中，有的指令会**隐式使用/定值**固定的物理寄存器
  - 比如 x86 上的 mul 指令
    - 读取 eax；结果写入 eax、edx
  - 比如 RISC-V 上的 call 指令
    - 读取存放参数的寄存器 a0、a1、a2 等
    - 可以视作对调用者保存(caller-saved)寄存器的定值
  - 比如 RISC-V 上的 ret 指令
    - 读取 a0

- 把这些物理寄存器当作**特殊的符号寄存器**处理
- 在着色开始前就加入冲突图中**并着色**
- 着色过程中不要溢出这些特殊结点

预着色结点的例子：以 RISC-V 为例

```
int fact(int k, int *r) {
  if (k == 0) *r = 1;
  else {
    fact(k - 1, r);
    *r = *r * k;
  }
  return 42;
}
```

```
fact(k, r):
(1)  if k == 0 goto (3)
(2)  goto (5)
(3)  t1 = 1
(4)  *r = t1
(5)  goto (10)
(6)  t2 = k - 1
(7)  call fact(t2, r)
(8)  t3 = *r
(9)  t4 = t3 * k
(10)  *r = t4
(11) t5 = 42
(12) return t5
```

```
fact:
    mv   R1, a0
    mv   R2, a1
    beqz R1, if_then
    j    if_else
if_then:
    li   R3, 1
    sw   R3, 0(R2)
    j    if_cont
if_else:
    addi R4, R1, -1
    mv   a0, R4
    mv   a1, R2
    call fact
    lw   R5, 0(R2)
    mul  R6, R5, R1
    sw   R6, 0(R2)
if_cont:
    li   R7, 42
    mv   a0, R7
    ret
```

第一趟：把参数加载到符号寄存器

前 8 个参数：a0 ~ a7
第 9 个参数：0(sp)
第 10 个参数：4(sp)
……

第一趟：在调用函数前按规定传参

第一趟：按规定使用 a0 存储返回值

预着色结点的例子：以 RISC-V 为例

```
int fact(int k, int *r) {
  if (k == 0) *r = 1;
  else {
    fact(k - 1, r);
    *r = *r * k;
  }
  return 42;
}
```

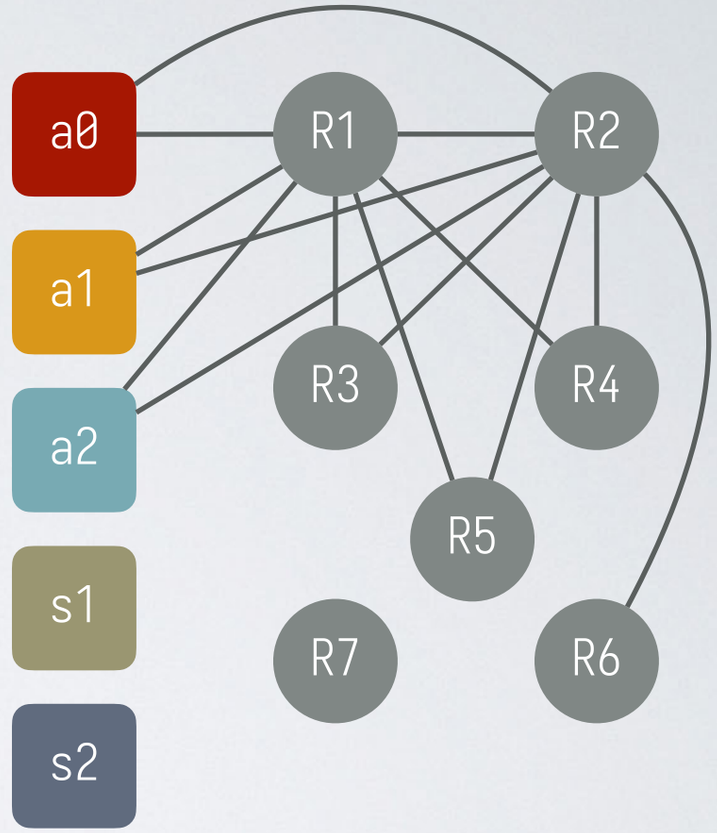
```
fact(k, r):
(1)  if k == 0 goto (3)
(2)  goto (5)
(3)  t1 = 1
(4)  *r = t1
(5)  goto (10)
(6)  t2 = k - 1
(7)  call fact(t2, r)
(8)  t3 = *r
(9)  t4 = t3 * k
(10)  *r = t4
(11) t5 = 42
(12) return t5
```

```
fact:
    mv   R1, a0
    mv   R2, a1
    beqz R1, if_then
    j    if_else
if_then:
    li   R3, 1
    sw   R3, 0(R2)
    j    if_cont
if_else:
    addi R4, R1, -1
    mv   a0, R4
    mv   a1, R2
    call fact
    lw   R5, 0(R2)
    mul  R6, R5, R1
    sw   R6, 0(R2)
if_cont:
    li   R7, 42
    mv   a0, R7
    ret
```

```
{a0,a1}
{R1,a1}
{R1,R2}
{R1,R2}
{R2}
{R2}
{R2,R3}
{}
{R1,R2}
{R1,R2}
{R1,R2,R4}
{R1,R2,a0}
{R1,R2,a0,a1}
{R1,R2}
{R1,R2,R5}
{R2,R6}
{}
{}
{R7}
{a0}
```

call 指令使用 a0 和 a1
并定值所有 caller-saved 寄存器

ret 指令使用 a0



预着色结点的例子：以 RISC-V 为例

```
int fact(int k, int *r) {
  if (k == 0) *r = 1;
  else {
    fact(k - 1, r);
    *r = *r * k;
  }
  return 42;
}
```

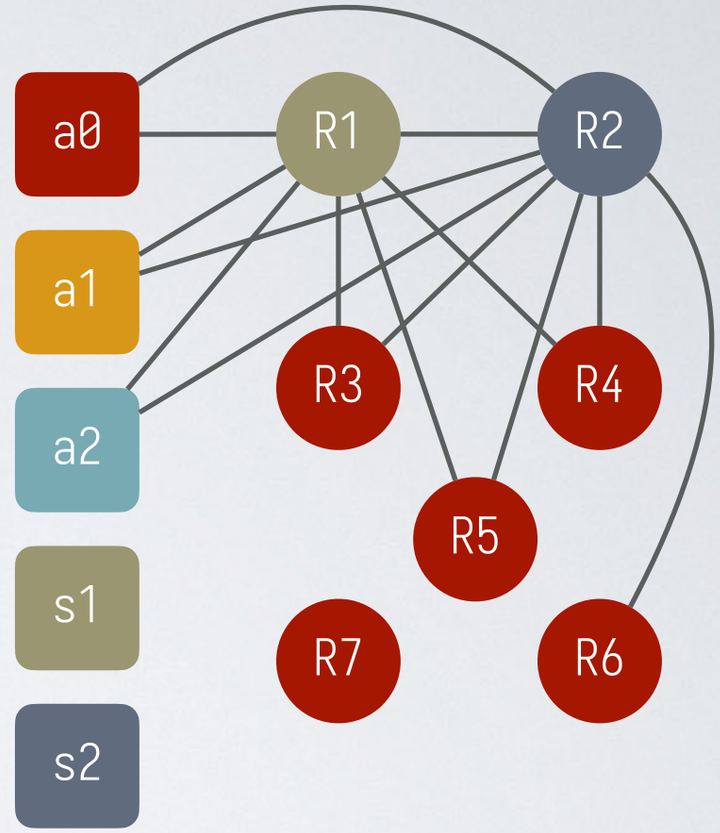
```
fact(k, r):
(1)  if k == 0 goto (3)
(2)  goto (5)
(3)  t1 = 1
(4)  *r = t1
(5)  goto (10)
(6)  t2 = k - 1
(7)  call fact(t2, r)
(8)  t3 = *r
(9)  t4 = t3 * k
(10)  *r = t4
(11) t5 = 42
(12) return t5
```

```
fact:
    mv   R1, a0
    mv   R2, a1
    beqz R1, if_then
    j    if_else
if_then:
    li   R3, 1
    sw   R3, 0(R2)
    j    if_cont
if_else:
    addi R4, R1, -1
    mv   a0, R4
    mv   a1, R2
    call fact
    lw   R5, 0(R2)
    mul  R6, R5, R1
    sw   R6, 0(R2)
if_cont:
    li   R7, 42
    mv   a0, R7
    ret
```

```
{a0,a1}
{R1,a1}
{R1,R2}
{R1,R2}
{R1,R2}
{R1,R2}
{R1,R2,R3}
{R1,R2}
{R1,R2}
{R1,R2}
{R1,R2}
{R1,R2,R4}
{R1,R2,a0}
{R1,R2,a0,a1}
{R1,R2}
{R1,R2,R5}
{R2,R6}
{}
{}
{R7}
{a0}
```

call 指令使用 a0 和 a1
并定值所有 caller-saved 寄存器

ret 指令使用 a0



预着色结点的例子：以 RISC-V 为例

```
int fact(int k, int *r) {
  if (k == 0) *r = 1;
  else {
    fact(k - 1, r);
    *r = *r * k;
  }
  return 42;
}
```

```
fact(k, r):
(1)  if k == 0 goto (3)
(2)  goto (5)
(3)  t1 = 1
(4)  *r = t1
(5)  goto (10)
(6)  t2 = k - 1
(7)  call fact(t2, r)
(8)  t3 = *r
(9)  t4 = t3 * k
(10)  *r = t4
(11) t5 = 42
(12) return t5
```

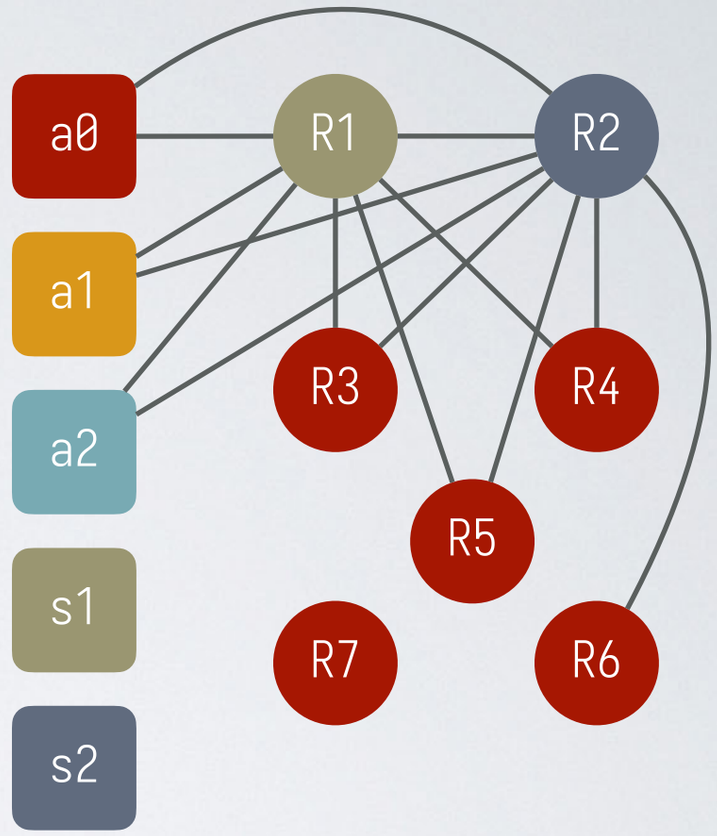
使用了 callee-saved 寄存器，
需要再 patch 一下目标代码

```
fact:
    mv   s1, a0
    mv   s2, a1
    beqz s1, if_then
    j    if_else
if_then:
    li   a0, 1
    sw   a0, 0(s2)
    j    if_cont
if_else:
    addi a0, s1, -1
    mv   a0, a0
    mv   a1, s2
    call fact
    lw   a0, 0(s2)
    mul  a0, a0, s1
    sw   a0, 0(s2)
if_cont:
    li   a0, 42
    mv   a0, a0
    ret
```

```
{a0,a1}
{R1,a1}
{R1,R2}
{R1,R2}
{R1,R2}
{R1,R2}
{R1,R2,R3}
{R1,R2}
{R1,R2}
{R1,R2}
{R1,R2}
{R1,R2,R4}
{R1,R2,a0}
{R1,R2,a0,a1}
{R1,R2}
{R1,R2,R5}
{R2,R6}
{}
{}
{R7}
{a0}
```

call 指令使用 a0 和 a1
并定值所有 caller-saved 寄存器

ret 指令使用 a0



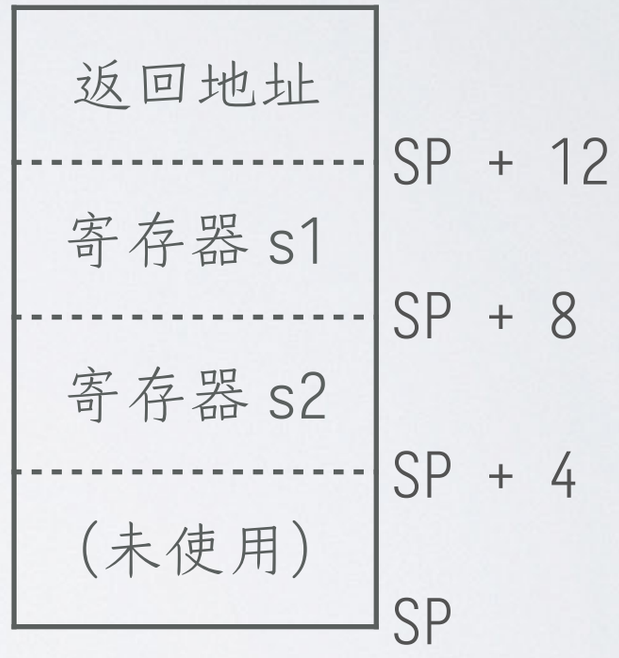
预着色结点的例子：以 RISC-V 为例

```
int fact(int k, int *r) {
  if (k == 0) *r = 1;
  else {
    fact(k - 1, r);
    *r = *r * k;
  }
  return 42;
}
```

```
fact(k, r):
(1)  if k == 0 goto (3)
(2)  goto (5)
(3)  t1 = 1
(4)  *r = t1
(5)  goto (10)
(6)  t2 = k - 1
(7)  call fact(t2, r)
(8)  t3 = *r
(9)  t4 = t3 * k
(10)  *r = t4
(11) t5 = 42
(12) return t5
```

```
fact:
    addi sp, sp, -16
    sw   ra, 12(sp)
    sw   s1, 8(sp)
    sw   s2, 4(sp)
    mv   s1, a0
    mv   s2, a1
    beqz s1, if_then
    j    if_else
if_then:
    li   a0, 1
    sw   a0, 0(s2)
    j    if_cont
if_else:
    addi a0, s1, -1
    mv   a1, s2
    call fact
    lw   a0, 0(s2)
    mul  a0, a0, s1
    sw   a0, 0(s2)
if_cont:
    li   a0, 42
    lw   ra, 12(sp)
    lw   s1, 8(sp)
    lw   s2, 4(sp)
    addi sp, sp, 16
    ret
```

在栈帧中保存 callee-saved 寄存器的值：除了指派中使用到的 s1、s2，还有因为有函数调用导致需要保存的返回地址寄存器 ra



在函数返回前通过之前在栈帧中保存的值恢复 callee-saved寄存器的值

主要内容

- 目标代码生成的基本任务
- 目标机模型
- 指令选择
- 寄存器分配
- **实践演示**

本讲小结

- 目标代码生成的基本任务：指令选择、指令排序、寄存器分配
- 目标机模型
  - 类 RISC 目标机的指令集、寻址模式、栈式存储管理
- 指令选择
  - 三地址代码基本块的指令选择算法和寄存器选择算法
- 寄存器分配
  - 基于图着色的全局寄存器分配算法
- **后续可能讲的专题：**
  - **其它寄存器分配算法、指令排序等**

思考问题

- 目标代码生成和中间代码生成都是从一个语言翻译到另一个语言，两者面临的基本任务和问题有什么不同之处？
- RISC 和 CISC 体系结构对指令选择、寄存器分配等环节的需求有什么不同之处？
- 在进行指令选择前，编译器需要在(机器无关)中间代码上添加哪些信息？
- 如何在图状 IR (比如 AST、DAG)上进行指令选择？
- 目前，越来越多的编译器的寄存器分配环节选择实现更为简洁的算法(比如线性扫描)而非图着色，为什么？