

## Lecture 14

# 寄存器分配

徐 辉

xuh@fudan.edu.cn



# 大纲

- 一、寄存器分配问题
- 二、着色算法
- 三、预分配和溢出

# 一、寄存器分配问题

---

# 非SSA指令翻译结果

- 虚拟寄存器只在当前代码块有效
  - 跨代码块可以重新编号
- 单个代码块内的寄存器编号递增

```
BB1:
    %a = alloca i32
    %b = alloca i32
    %r = alloca i32
    store i32 1, i32* %a
    store i32 1, i32* %b
    %a1 = load i32, i32* %a
    %r1 = icmp eq i32 %a1, 0
    br i1 %r1, label %BB2, label %BB3

BB2:
    %a2 = load i32, i32* %a
    %b1 = load i32, i32* %b
    %a3 = add i32 %a2, %b1
    store i32 %a3, i32* %a
    br label %BB2

BB3:
    %a4 = load i32, i32* %a
    %b2 = load i32, i32* %b
    %r2 = add i32 %a4, %b2
    store i32 %r2, i32* %r
```



```
main:
    sub    sp, sp, #16
    mov    %r1, #1
    str    %r1, [sp, #12]
    str    %r1, [sp, #8]
    ldr    %r2, [sp, #12]
    cbnz   %r2, .LBB0_2

.LBB0_1:
    ldr    %r1, [sp, #12]
    ldr    %r2, [sp, #8]
    add    %r3, %r1, %r2
    str    %r3, [sp, #12]
    b      .LBB0_1

.LBB0_2:
    ldr    %r1, [sp, #12]
    ldr    %r2, [sp, #8]
    add    %r3, %r1, %r2
    str    %r3, [sp, #4]
    add    sp, sp, #16
    ret
```

# deSSA指令翻译结果

- 虚拟寄存器可以在多个代码块之间共享
  - 跨代码块不能重新编号
- 代码块之间的寄存器编号递增

```
BB1:
    %a = alloca i32
    %b = alloca i32
    %r = alloca i32
    store i32 1, i32* %a
    store i32 1, i32* %b
    %a1 = load i32, i32* %a
    %b1 = load i32, i32* %b
    %r1 = icmp eq i32 %a1, 0
    br i1 %r1, label %BB2, label %BB3

BB2:
    %a2 = add i32 %a1, %b1
    store i32 %a2, i32* %a
    br label %BB2

BB3:
    %a3 = load i32, i32* %a
    %r2 = add i32 %a3, %b1
    store i32 %r2, i32* %r
    ret void
```



```
main:
    mov     %r1, #1
    str     %r1, [sp, #28]
    str     %r1, [sp, #24]
    ldr     %r2, [sp, #28]
    ldr     %r3, [sp, #24]
    cbnz    %r2, .LBB0_2

.LBB0_1:
    add     %r4, %r2, %r3
    str     %r4, [sp, #28]
    b       .LBB0_1

.LBB0_2:
    ldr     %r5, [sp, #28]
    add     %r6, %r3, %r5
    str     %r6, [sp, #20]
    add     sp, sp, #32
    ret
```

# 寄存器分配问题

- 指令翻译的寄存器需遵循寄存器用法约定
- 如何为其它虚拟寄存器分配实际的物理寄存器？
  - 指令翻译没有限制虚拟寄存器的数量
  - 但物理寄存器的数量是有限的
  - 物理寄存器不足则将数据写入内存（spill），使用时再读取

X86-64寄存器	调用规约	注释
X0-X7	参数1-8	Caller-saved
X0-X1	返回值	Caller-saved
X8	特殊用途：间接调用返回地址	Caller-saved
<b>X9-X15</b>	<b>临时寄存器</b>	<b>Caller-saved</b>
X16-X17	特殊用途：Intra-Procedure-Call	Caller-saved
X18	特殊用途：平台寄存器	Caller-saved
<b>X19-X28</b>	<b>普通寄存器</b>	<b>Callee-saved</b>
X29	栈帧基指针	Caller-saved
X30	返回地址	Caller-saved
SP	栈顶指针	Callee-saved

aarch64寄存器用法约定

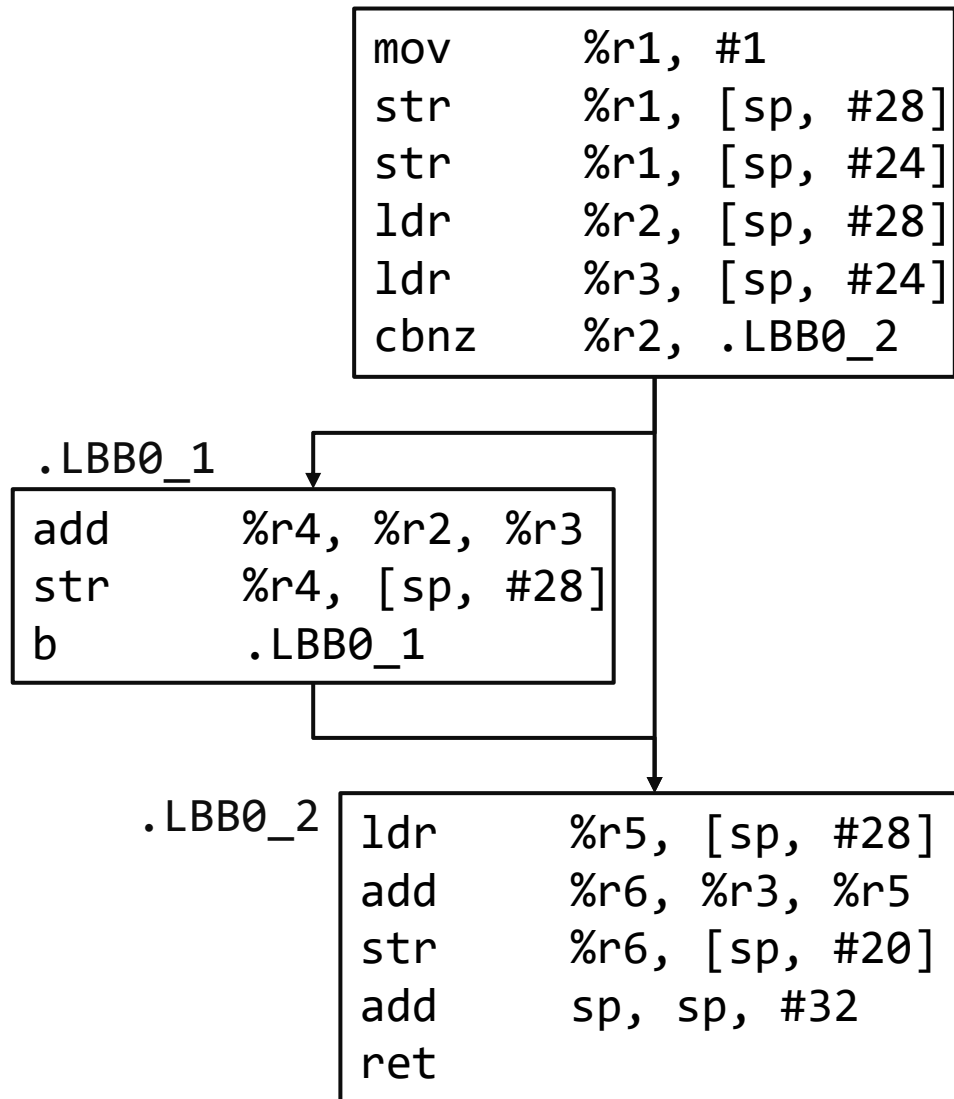
# 活跃性分析（非SSA）

- 如果一个寄存器还会被使用，则在当前节点是活跃的

main:			
	sub	sp, sp, #16	∅
	mov	%r1, #1	∅
	str	%r1, [sp, #12]	%r1
	str	%r1, [sp, #8]	%r1
	ldr	%r2, [sp, #12]	∅
	cbnz	%r2, .LBB0_2	%r2
.LBB0_1:			∅
	ldr	%r1, [sp, #12]	∅
	ldr	%r2, [sp, #8]	%r1
	add	%r3, %r1, %r2	%r1, %r2
	str	%r3, [sp, #12]	%r3
	b	.LBB0_1	∅
.LBB0_2:			∅
	ldr	%r1, [sp, #12]	∅
	ldr	%r2, [sp, #8]	%r1
	add	%r3, %r1, %r2	%r1, %r2
	str	%r3, [sp, #4]	%r3
	add	sp, sp, #16	∅
	ret		∅

# 活跃性分析 (SSA)

- 基于控制流图分析活跃性



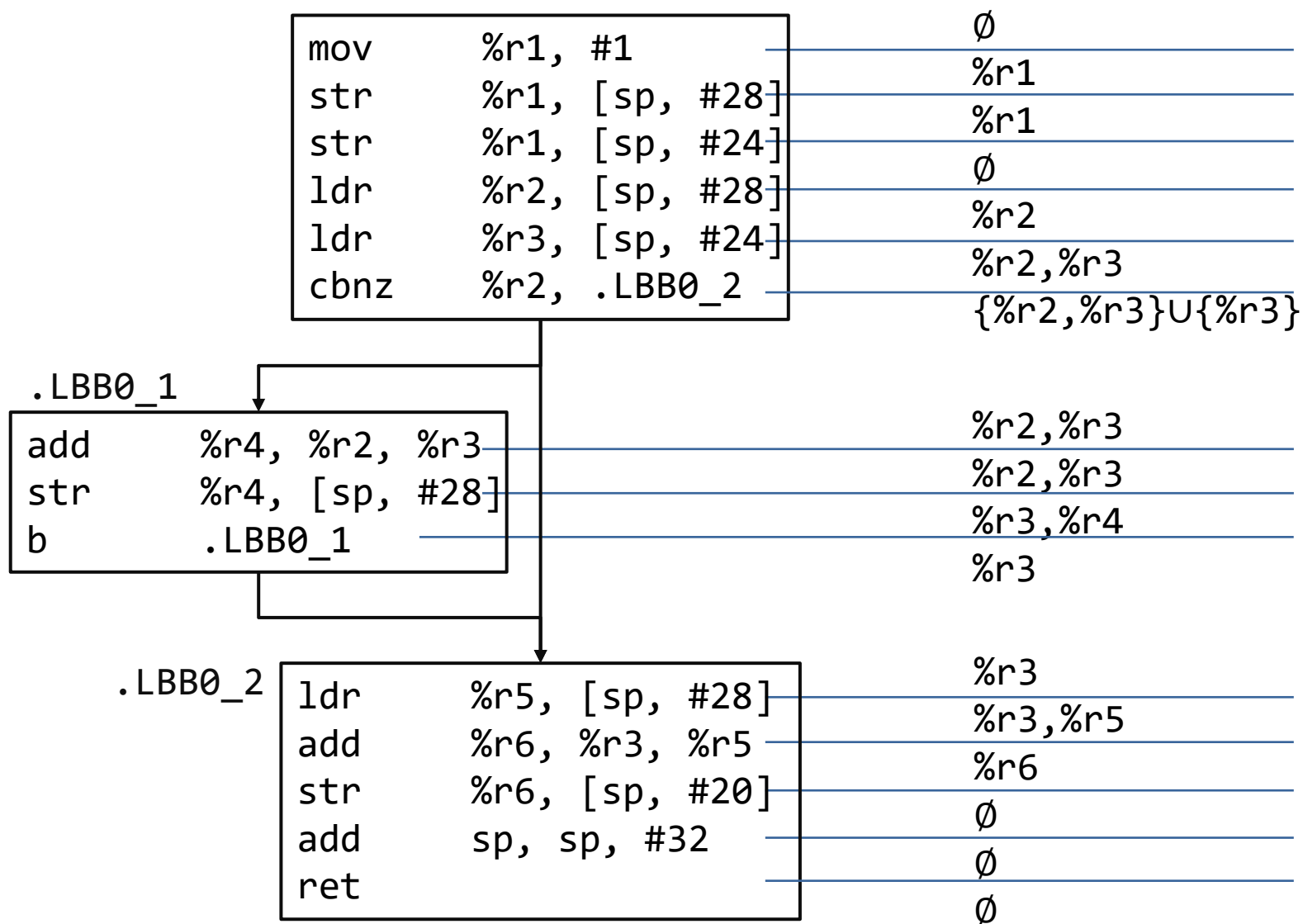
- 反向遍历控制流图:

- 如遇到指令: add r1, r2, r3
  - Gen(n) = {r2, r3}
  - KILL(n) = {r1}
- 如遇到指令: ldr r, [mem]
  - KILL(n) = {r}
- 如遇到指令: str r, [mem]
  - Gen(n) = {r}
- 如遇到指令: op r, label
  - Gen(n) = {r}
- ...
- $IN(n) = (OUT(n) - KILL(n)) \cup Gen(n)$



# 活跃性分析 (SSA)

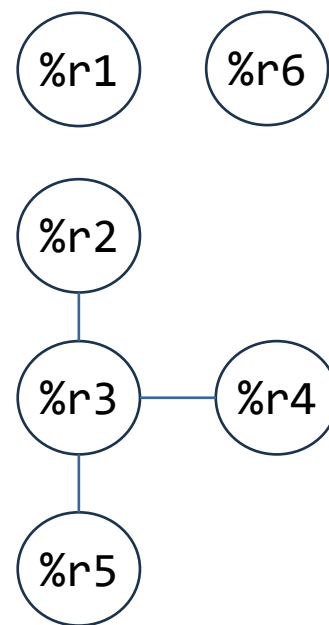
- 基于控制流图分析活跃性



# 干扰图 (Interference Graph)

- 干扰：两个同时活跃的寄存器存在干扰关系
- 干扰图：连接所有存在干扰关系的寄存器节点
- 含义：存在干扰关系的寄存器在某一时刻同时存活，应分配不同的物理寄存器

$\emptyset$	$\emptyset$	%r3
%r1	%r2, %r3	%r3, %r5
%r1	%r3, %r4	%r6
$\emptyset$	%r3	$\emptyset$
%r2		$\emptyset$
%r2, %r3		$\emptyset$



# 翻译结果

w9: %r3

w8: %r1,%r2,%r3,%r4,%r5

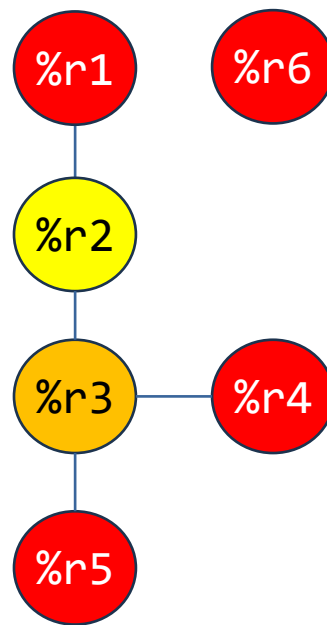
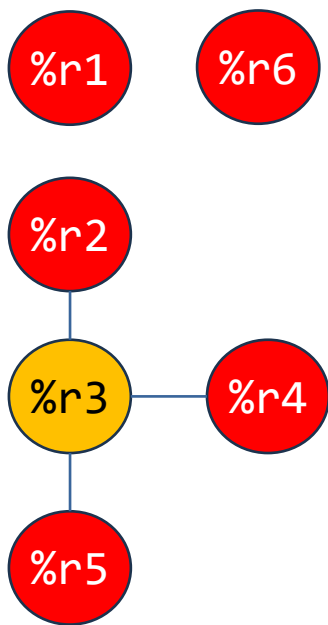
```
main:
    mov     %r1, #1
    str     %r1, [sp, #28]
    str     %r1, [sp, #24]
    ldr     %r2, [sp, #28]
    ldr     %r3, [sp, #24]
    cbnz    %r2, .LBB0_2
.LBB0_1:
    add     %r4, %r2, %r3
    str     %r4, [sp, #28]
    b       .LBB0_1
.LBB0_2:
    ldr     %r5, [sp, #28]
    add     %r6, %r3, %r5
    str     %r6, [sp, #20]
    add     sp, sp, #32
    ret
```



```
main:
    mov     w8, #1
    str     w8, [sp, #28]
    str     w8, [sp, #24]
    ldr     w8, [sp, #28]
    ldr     w9, [sp, #24]
    cbnz    w8, .LBB0_2
.LBB0_1:
    add     w8, w8, w9
    str     w8, [sp, #28]
    b       .LBB0_1
.LBB0_2:
    ldr     w8, [sp, #28]
    add     w8, w9, w8
    str     w8, [sp, #20]
    add     sp, sp, #32
    ret
```

## 寄存器分配=>着色问题 (Graph Coloring)

- 使用不超过K种(X9-X15)颜色, 要求相邻节点颜色均不同
- 当 $K \geq 3$ 时, 该问题是NP完全问题 (Chaitin的证明)



# 基于SAT问题证明

- k-SAT: CNF的每个Clause有不超过k个literals
  - 3SAT是NP-Complete问题
  - 2SAT是多项式复杂度可解
- 如果所有SAT问题可以多项式时间reduce到目标问题, 则说明目标问题的难度至少与SAT相当

Literal:  $x_1, \overline{x_1}, x_2, \overline{x_2}, x_3, \dots$

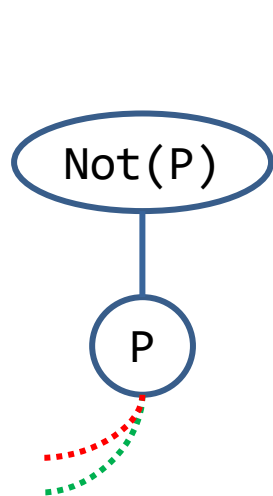
Clause:  $l_1 \vee l_2 \vee l_3$

Conjunctive Normal Form:  $C_1 \wedge C_2 \wedge \dots$

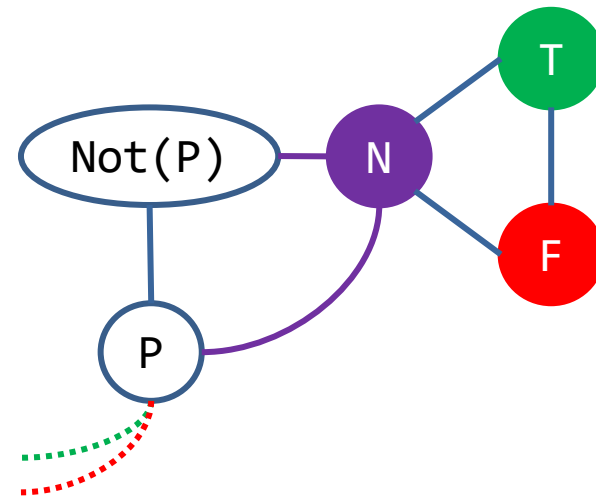
举例:  $(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee x_4) \wedge \dots$

# 3SAT可以reduce到着色问题

- 构造not和or
- and可以用not和or表示：
  - $C_1 \wedge C_2 = \neg(\neg C_1 \vee \neg C_2) \dots$

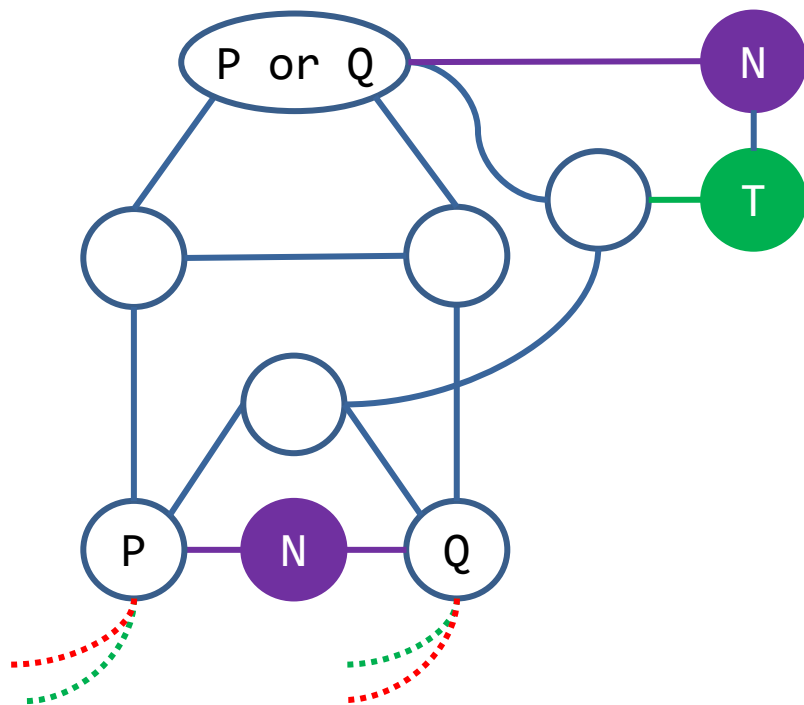


构造not



构造not

# 3SAT可以reduce到着色问题



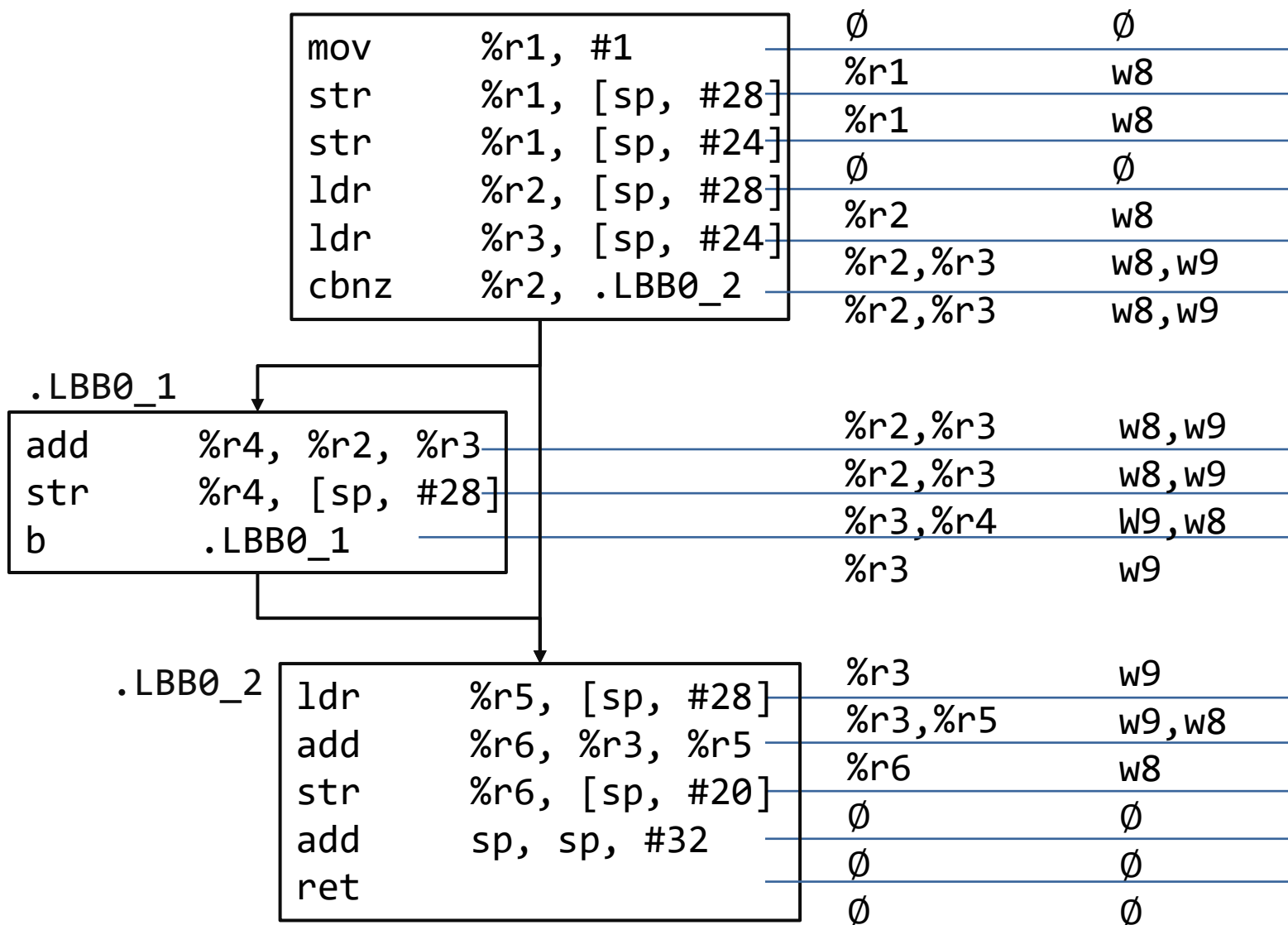
构造or

## 二、着色算法

---



# 线性扫描算法：先到先得



# 贪心法着色

颜色顺序:



- 策略: 根据邻居节点颜色, 为当前节点选取可用的颜色;

贪心算法着色

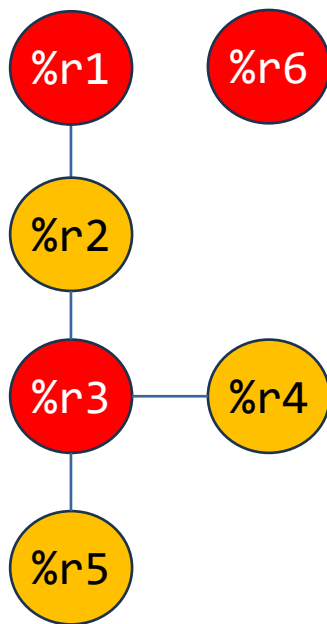
Input:  $G=(V,E)$

Output: Assignment of colors

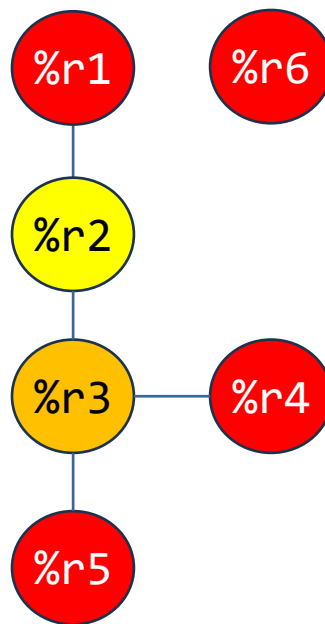
For  $i = 1..n$  do

Let  $c$  be the lowest color not used in  $\text{Neighbor}(v_i)$

Set  $\text{Col}(v_i) = c$



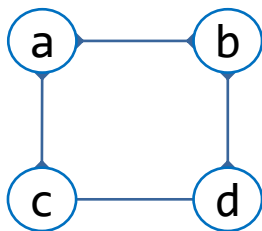
着色顺序是: 1-2-3-4-5-6



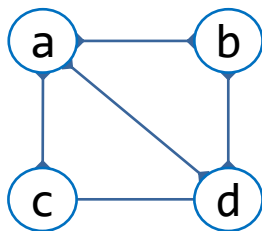
着色顺序是: 1-6-4-3-2-5

# 一类特殊的着色问题：弦图（Chordal Graph）

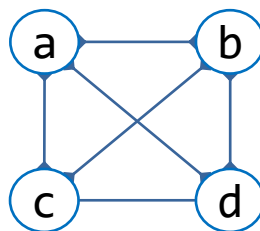
- 任意长度大于3的环都有弦（chord）
- 多项式时间可解
- 静态单赋值形式的干扰图都是弦图



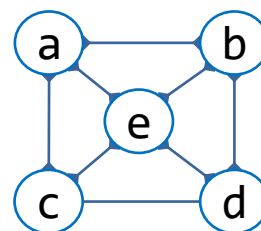
非弦图



弦图

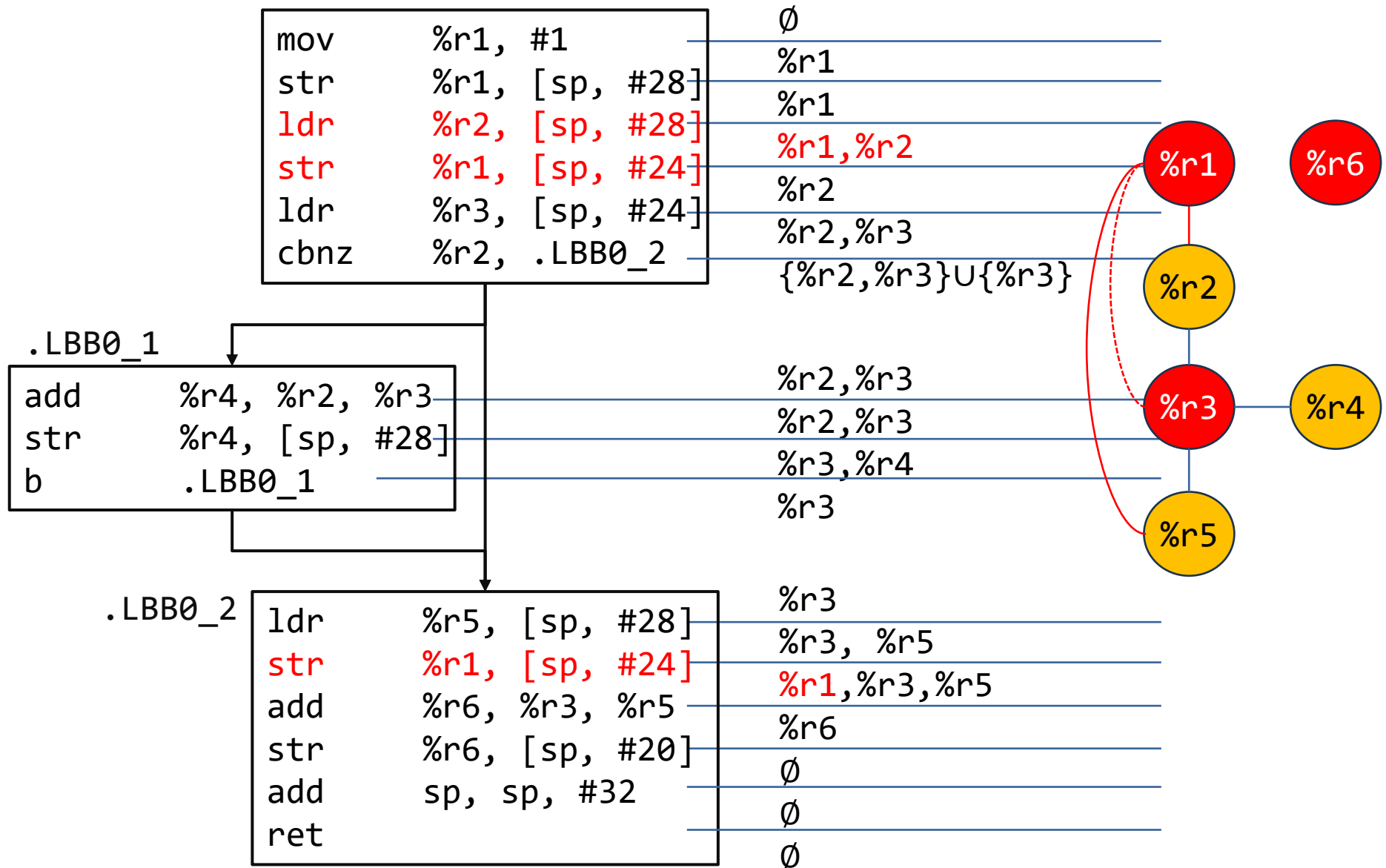


弦图



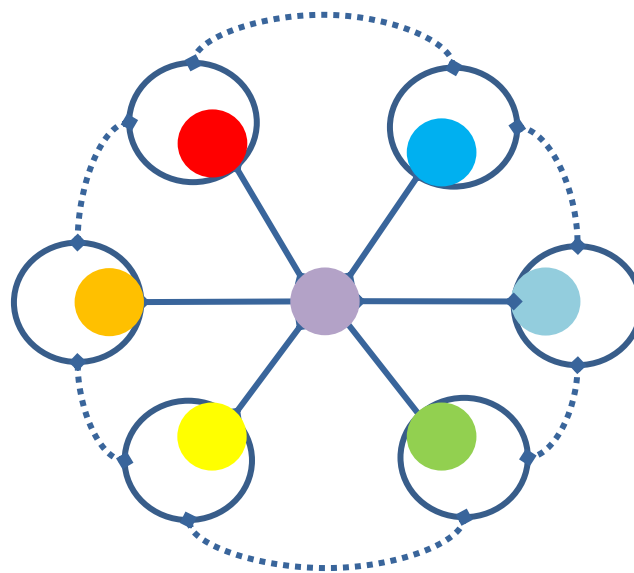
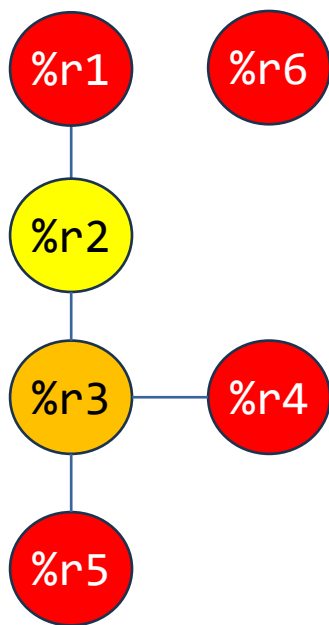
非弦图

# 尝试为SSA构造非弦图？



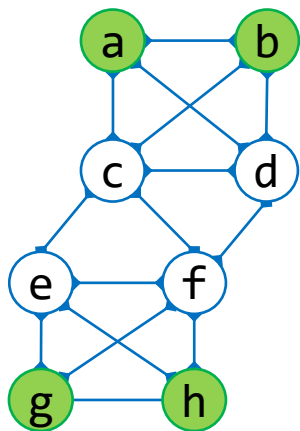
# 着色思路

- 在图上搜索团 (clique)
  - 团：所有节点两两连接
  - 着色所需颜色数与团的大小一致
- 找最大团也是np-hard问题
- 着色顺序不引入非团节点带来的颜色限制即可
  - 单纯消除序列可达到上述目的

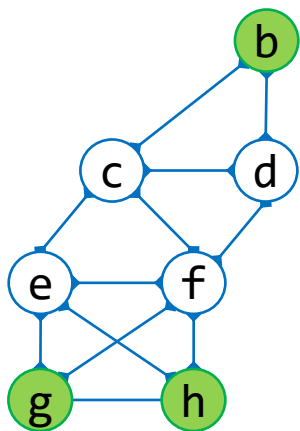


# 单纯消除序列 (Simplicial Elimination Ordering)

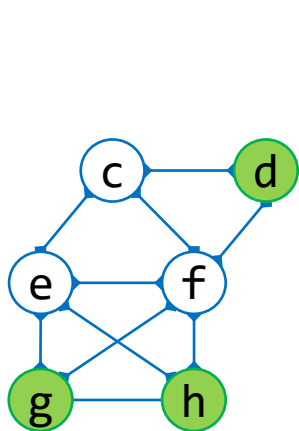
- 单纯点 (simplicial) : 所有邻居组成一个团
- 完美消除序列: 按照该序列消除的每一个点都是单纯点
- 单纯消除序列: 完美消除序列的逆序
- 如果一个图是弦图, 则该图存在完美消除序列



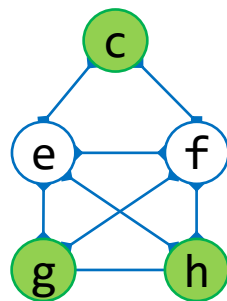
消除a



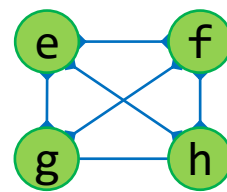
消除b



消除d



消除c



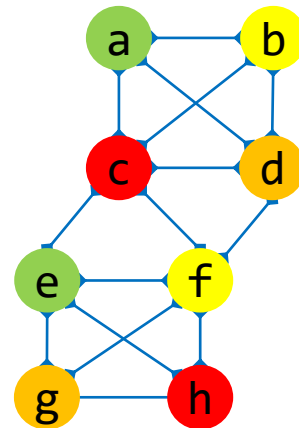
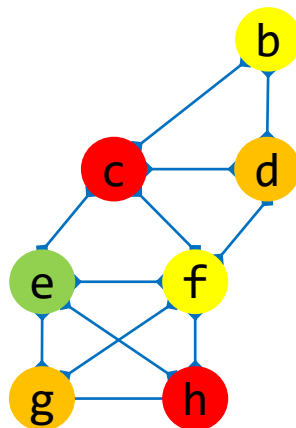
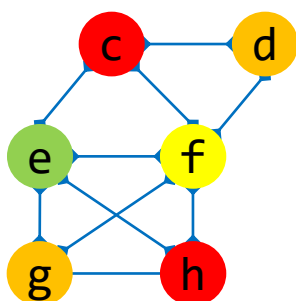
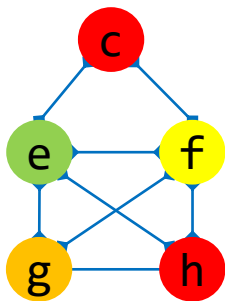
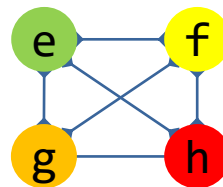
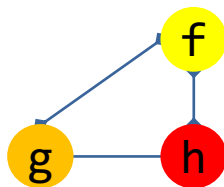
消除e

消除f...

● 单纯点    ○ 非单纯点

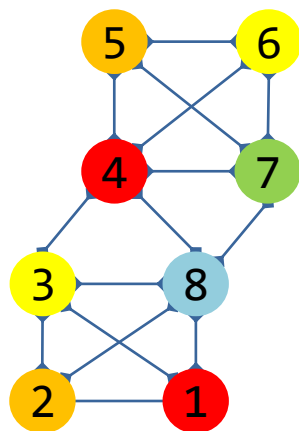
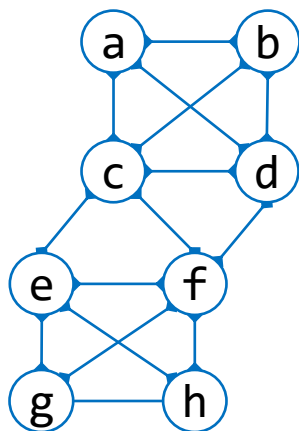
# 基于单纯消除序列着色

- 每次在已着色团的基础上新增一个点，连接该团的所有点



# 如果不遵循非单纯消除序列着色

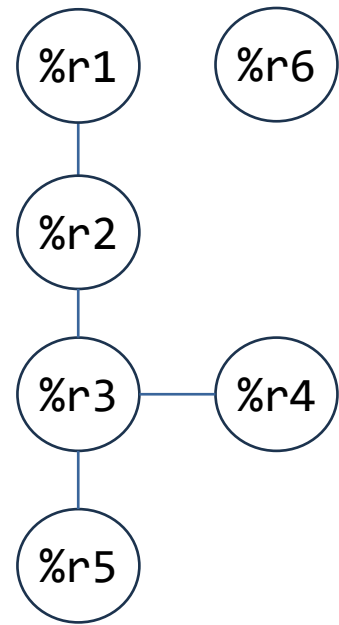
- 所需颜色数可能需要超过最大团大小





# 最大势算法求单纯消除序列

- Maximum Cardinality Search
- 思路：搜索与已着色节点邻居最多的点
  - 维护一个所有点的向量，每次选取值最大的点；
  - 选取一个点后，则其邻居计数加1。



步骤	选取	%r1	%r2	%r3	%r4	%r5	%r6
1	%r1		1	0	0	0	0
2	%r2			1	0	0	0
3	%r3				1	1	0
4	%r4					1	0
5	%r5						0
6	%r6						

# 算法参考

## Maximum Cardinality Search

Input:  $G = (V, E)$

Output: Simplicial elimination ordering  $v_1, \dots, v_n$

For all  $v_i \in V$

$w(v_i) = 0$

Let  $W = V$

For  $i = 1, \dots, n$  do

    Let  $v$  be a node with max weight in  $W$

    Set  $v_i = v$

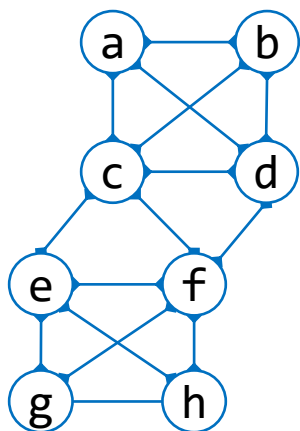
    For all  $u \in W \cap N(v)$

$w(u) = w(u) + 1$

$W = W \setminus \{v\}$

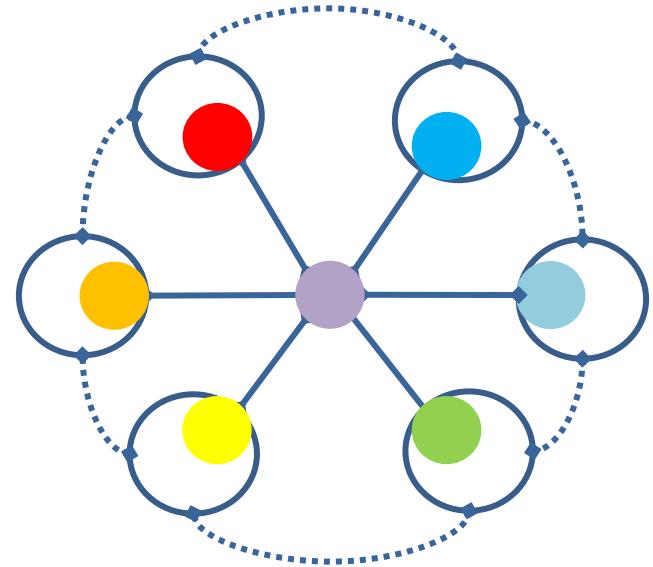
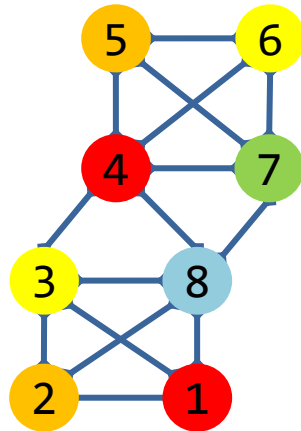
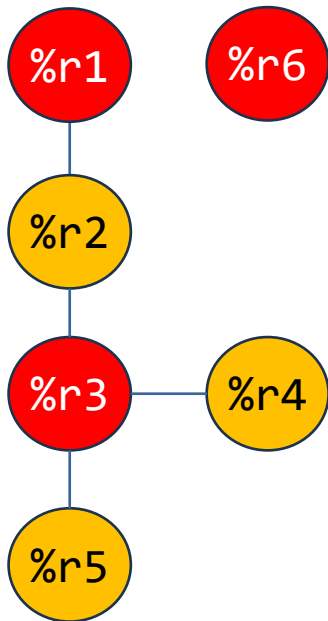
# 练习

- 求下列冲突图的单纯消除序列



步骤	选取	a	b	c	d	e	f	g	h
		0	0	0	0	0	0	0	0
1	a		1	1	1	0	0	0	0
2	b			2	2	0	0	0	0
3	c				3	1	1	0	0
4	d					1	2	0	0
5	f					2		1	1
6	e							2	2
7	g								3
8	h								

思考：为何最大势算法能得到单纯消除序列？



### 三、预分配和溢出

---

# 函数调用需要遵守的寄存器规则

X86-64寄存器	调用规约	注释
X0-X7	参数1-8	Caller-saved
X0-X1	返回值	Caller-saved
X8	特殊用途：间接调用返回地址	Caller-saved
X9-X15	临时寄存器	Caller-saved
X16-X17	特殊用途：Intra-Procedure-Call	Caller-saved
X18	特殊用途：平台寄存器	Caller-saved
X19-X28	普通寄存器	Callee-saved
X29	栈帧基指针	Caller-saved
X30	返回地址	Caller-saved
SP	栈顶指针	Callee-saved

# 函数调用示例

main:

sub sp, sp, #32

str x30, [sp, #16] → 保存返回地址

mov w8, #1

str w8, [sp, #12]

str w8, [sp, #8]

ldr w8, [sp, #12]

ldr w9, [sp, #8]

add w0, w8, w9 → 参数传递

mov w1, w9

bl fnA → 函数调用

str w0, [sp, #8] → 返回值

ldr x30, [sp, #16] → 还原返回地址

add sp, sp, #32

ret

# 临时寄存器的使用：示例

main:

```
sub    sp, sp, #32
str    x30, [sp, #16]
mov    w8, #1
str    w8, [sp, #12]
str    w8, [sp, #8]
ldr    w8, [sp, #12]
ldr    w9, [sp, #8]
add    w0, w8, w9
mov    w1, w9
str    w9, [sp, #4]
str    w8, [sp]
bl     fnA
ldr    w8, [sp]
ldr    w9, [sp, #4]
add    w10, w8, w9
str    w10, [sp, #8]
ldr    x30, [sp, #16]
add    sp, sp, #32
ret
```

→ 将临时寄存器入栈

→ 函数调用

→ 将临时寄存器还原

何时用到普通寄存器：X19-X28？



# 参数过多怎么办？

```
%b2 = call i32 @fnB(  
    i32 %a2, i32 %b1,  
    i32 %a2, i32 %b1,  
    i32 %a2, i32 %b1,  
    i32 %a2, i32 %b1,  
    i32 %a2, i32 %b1 )
```

```
mov     w0, w8  
mov     w1, w9  
mov     w2, w8  
mov     w3, w9  
mov     w4, w8  
mov     w5, w9  
mov     w6, w8  
mov     w7, w9  
mov     x10, sp  
str     w8, [x10]  
mov     x10, sp  
str     w9, [x10, #8]  
bl      fnB
```

# 其它溢出考量因素

- 寄存器不足时应优先溢出哪个虚拟寄存器？
  - 线性统计：代码中出现次数最多的
  - 考虑控制流：代码运行次数最多的
- 目标：最少的溢出次数

# 思考：指令调度可否优化寄存器使用？

- 构造一个程序说明