Lecture 4.2

寄存器分配

徐 辉
xuh@fudan.edu.cn

大纲

一、寄存器分配

二、着色问题

三、着色算法

四、更多考量

何时引入的寄存器？

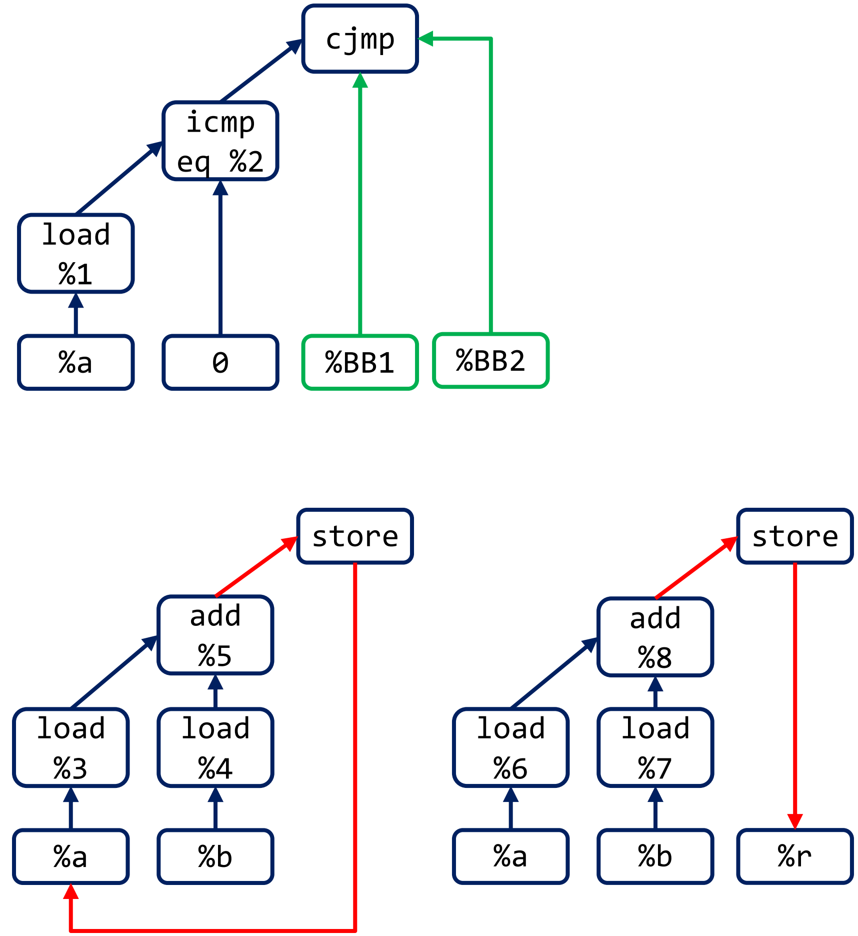
- 线性IR中引入虚拟寄存器
- 编号单调递增

```
fn foo(a:int, b:int)->int {
    if(a==0)
        a = a + b;
    let r:int = a + b;
    return r;
}
```

```
define fn i32 foo(i32 %a, i32 %b){
%BB0:
    %a = stackalloc i32;
    %b = stackalloc i32;
    %r = stackalloc i32;
    %1 = load i32, %a;
    %2 = icmp eq i32 %1, 0;
    cjmp i1 %2, %BB1, %BB2;
%BB1:
    %3 = load i32, %a;
    %4 = load i32, %b;
    %5 = add i32 %3, %4;
    store i32 %5, %a;
    jmp %BB2;
%BB2:
    %6 = load i32, %a;
    %7 = load i32, %b;
    %8 = add i32 %6, %7;
    store i32 %8, %r;
    %9 = load i32, %r;
    ret %9;
}
```

指令翻译按照代码块进行

```
define fn i32 foo(i32 %a, i32 %b){
%BB0:
    %a = stackalloc i32;
    %b = stackalloc i32;
    %r = stackalloc i32;
    %1 = load i32, %a;
    %2 = icmp eq i32 %1, 0;
    cjmp i1 %2, %BB1, %BB2;
%BB1:
    %3 = load i32, %a;
    %4 = load i32, %b;
    %5 = add i32 %3, %4;
    store i32 %5, %a;
    jmp %BB2;
%BB2:
    %6 = load i32, %a;
    %7 = load i32, %b;
    %8 = add i32 %6, %7;
    store i32 %8, %r;
    %9 = load i32, %r;
    ret %9;
}
```



指令翻译结果

- 单个代码块内的寄存器编号递增
- 跨代码块重新编号

```
define fn i32 foo(i32 %a, i32 %b){
%BB0:
    %a = stackalloc i32;
    %b = stackalloc i32;
    %r = stackalloc i32;
    %1 = load i32, %a;
    %2 = icmp eq i32 %1, 0;
    cjmp i1 %2, %BB1, %BB2;
%BB1:
    %3 = load i32, %a;
    %4 = load i32, %b;
    %5 = add i32 %3, %4;
    store i32 %5, %a;
    jmp %BB2;
%BB2:
    %6 = load i32, %a;
    %7 = load i32, %b;
    %8 = add i32 %6, %7;
    store i32 %8, %r;
    %9 = load i32, %r;
    ret %9;
}
```

⇨

```
%BB0:
    MOV %RSP, %RBP
    MOV %EDI, -0x4(%RBP)
    MOV %ESI, -0x8(%RBP)
    MOV %EDX, -0xc(%RBP)
    MOV -0x4(%RBP), %r1
    CMP %r1, 0
    JNZ .BB2
%BB1:
    MOV -0x4(%RBP), %r1
    MOV -0x8(%RBP), %r2
    ADD %r1, %r2
    MOV %r2, -0x4(%RBP)
%BB2:
    MOV -0x4(%RBP), %r1
    MOV -0x8(%RBP), %r2
    ADD %1, %r2
    MOV %2, -0xc(%RBP)
    MOV -0xc(%RBP), %EAX
    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寄存器用法约定

X86-64寄存器	调用规约	注释	用途
%RAX	返回值	Caller-saved	
%RDI	参数1	Caller-saved	
%RSI	参数2	Caller-saved	
%RDX	参数3	Caller-saved	
%RCX	参数4	Caller-saved	
%R8	参数5	Caller-saved	
%R9	参数6	Caller-saved	
%R10-%R11		Caller-saved	
%RBP		Callee-saved	函数栈帧基地址
%RSP		Callee-saved	栈顶地址
%RBX		Callee-saved	
%R12-%R15		Callee-saved	

X86-64寄存器用法约定

IR指令模式	运算数	汇编指令	寄存器预分配	结果

`mul(%1,%2)`	i64	`MOV %1, %r1`
		`MOV %2, %r2`
		`MUL %r2`
			`MOV %1, %RAX`
			`MOV %2, %r2`
			`MUL %r2`
				高位：%RDX
				低位：%RAX

`div(%1,%2)`	i64	`MOV %1, %r1`
		`MOV %2, %r2`
		`DIV %r2`
			`MOV %1, %RAX`
			`MOV %2, %r2`
			`DIV %r2`
				商：%RAX
				余数：%RDX

...				

大纲

一、寄存器分配

二、着色问题

三、着色算法

四、更多考量

活跃性（Liveness）分析

- 如果一个寄存器还会被使用，则在当前节点是活跃的

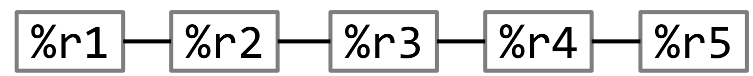
```
MOV -a(%rbp), %r1
MOV -b(%rbp), %r2
MOV %r1, %r3
ADD %r2, %r3
MOV %r2, %r4
ADD %r3, %r4
MOV %r3, %r5
ADD %r4, %r5
MOV %r5, -r(%rbp)
```

∅
%r1
%r1,%r2
%r2,%r3
%r2,%r3
%r3,%r4
%r3,%r4
%r4,%r5
%r5

干扰图（Interference Graph）

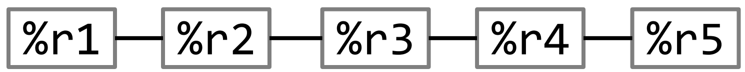
- 干扰：两个同时活跃的寄存器存在干扰关系
- 干扰图：连接所有存在干扰关系的寄存器节点
- 含义：存在干扰关系的寄存器在某一时刻同时存活，应分配不同的物理寄存器

```
                              ∅
MOV -a(%rbp), %r1
                              %r1
MOV -b(%rbp), %r2
                              %r1,%r2
MOV %r1, %r3
                              %r2,%r3
ADD %r2, %r3
                              %r2,%r3
MOV %r2, %r4
                              %r3,%r4
ADD %r3, %r4
                              %r3,%r4
MOV %r3, %r5
                              %r4,%r5
ADD %r4, %r5
                              %r5
MOV %r5, -r(%rbp)
```



着色问题（Graph Coloring）

- 寄存器分配问题转换为着色问题
- 使用不超过K种颜色为冲突图着色，要求相邻节点颜色均不同
- 当K≥3时，该问题是NP完全问题（Chaitin的证明）



G.J. Chaitin, *et al*, Register allocation via coloring. *Computer languages*, 1981.

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: $x_1, \overline{x_1}, x_2, \overline{x_2}, x_3, \ldots$

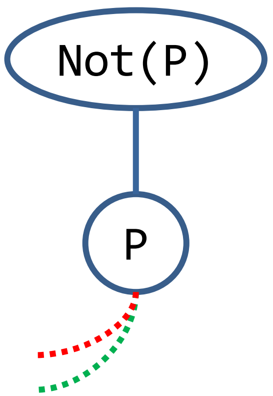
Clause: $l_1 \vee l_2 \vee l_3$

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

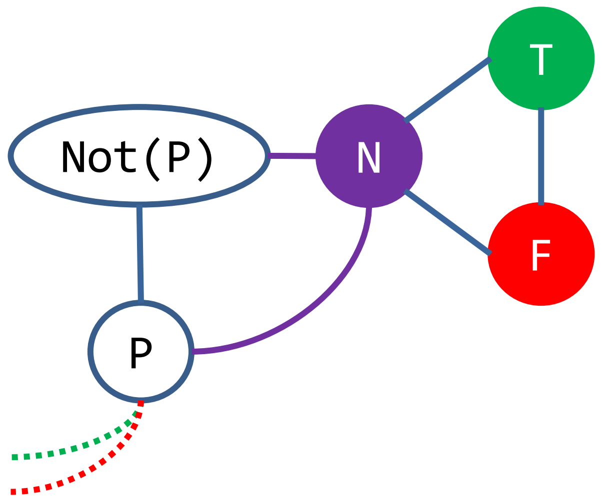
举例: $(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee x_4) \wedge \cdots$

3SAT可以reduce到着色问题

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



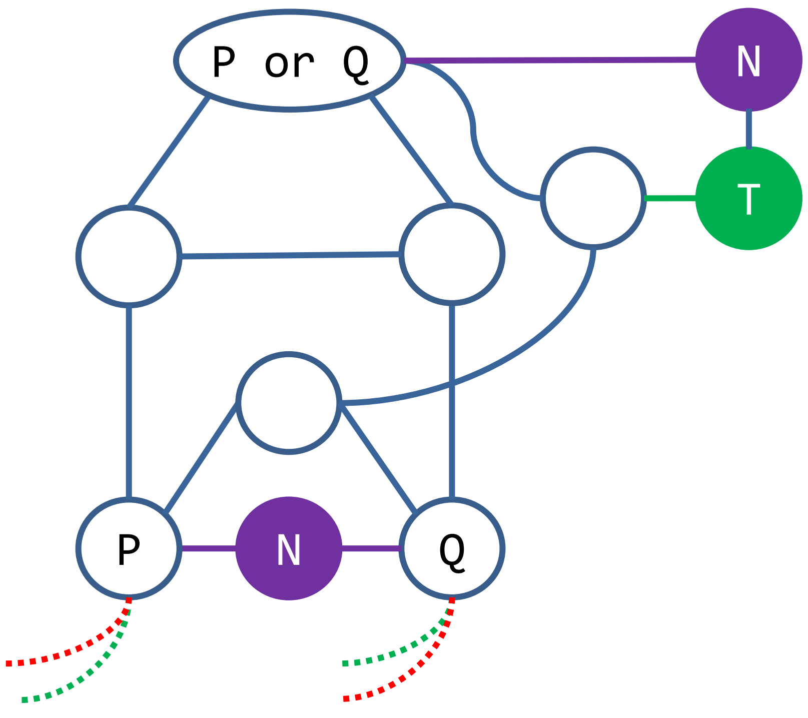
构造not门



构造not门

Albert R. Meyer和他的学生，MIT

3SAT可以reduce到着色问题



构造or门

大纲

一、寄存器分配

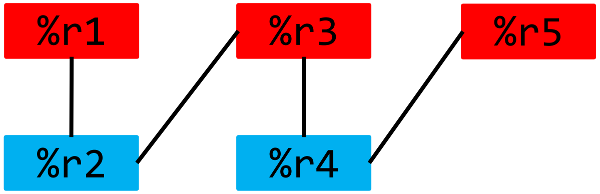
二、着色问题

三、着色算法

四、更多考量

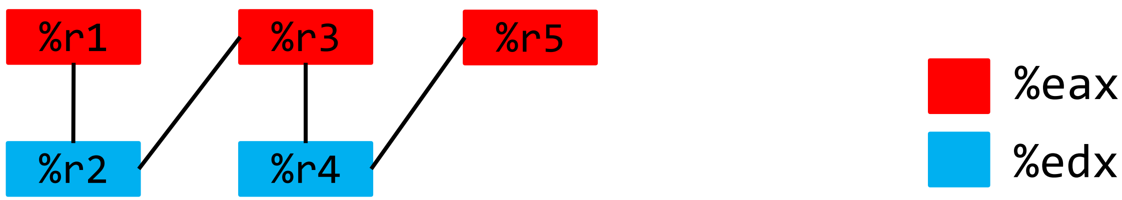
贪心法着色

- 策略：根据邻居节点颜色，为当前节点选取可用的颜色；
- 假设变量的着色顺序是%r1、%r2、%r3、%r4、%r5




```
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 Neighbor(vi)
  Set Col(vi) = c
```

寄存器分配后的程序



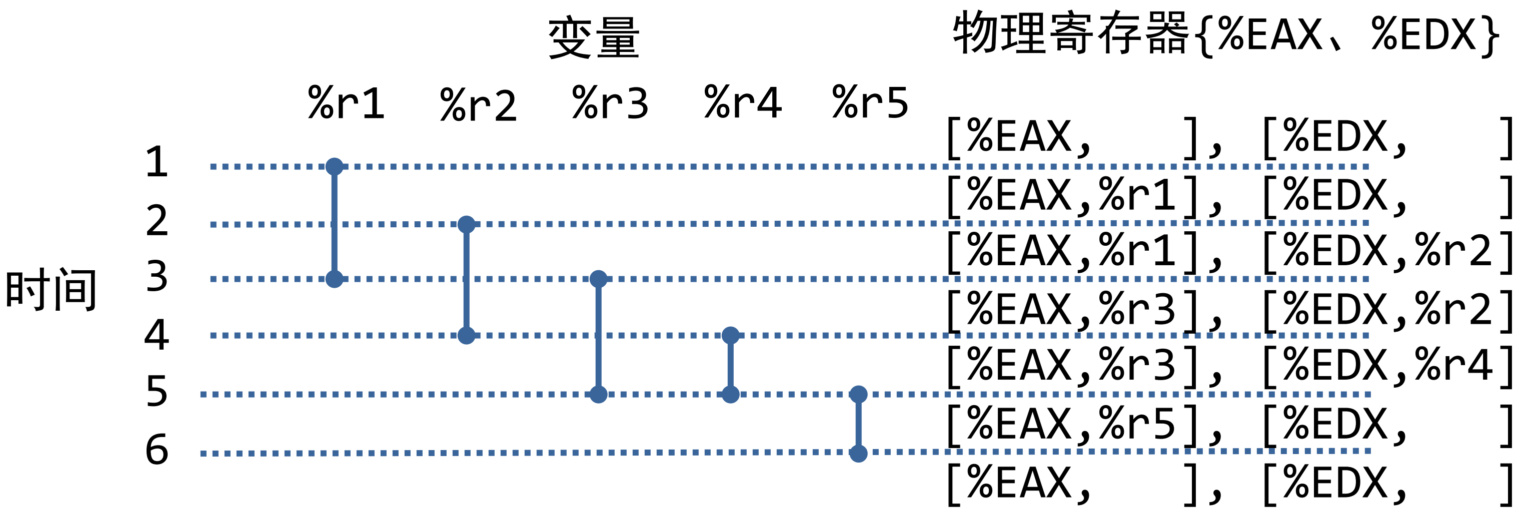

```
MOV -a(%rbp), %r1
MOV -b(%rbp), %r2
MOV %r1, %r3
ADD %r2, %r3
MOV %r2, %r4
ADD %r3, %r4
MOV %r3, %r5
ADD %r4, %r5
MOV %r5, -r(%rbp)
```

⇨

MOV -a(%rbp), %eax
MOV -b(%rbp), %edx
~~MOV %eax, %eax~~
ADD %edx, %eax
~~MOV %edx, %edx~~
ADD %eax, %edx
~~MOV %eax, %eax~~
ADD %edx, %eax
MOV %eax, -r(%rbp)

Linear Scan

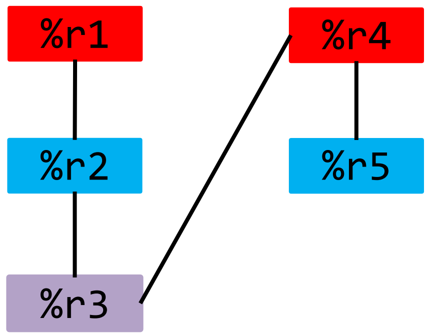
- 先到先得，不考虑全局因素



M. Poletto, and V. Sarkar, Linear scan register allocation. *TOPLAS*, 1999
O. Traub, *et al*. Quality and speed in linear-scan register allocation. *ACM SIGPLAN Notices*, *1998*.

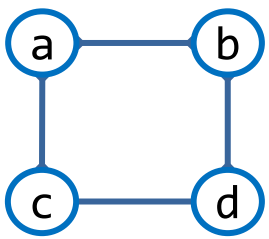
贪心着色算法有时不能求到最优解

- 假设着色顺序是%r1-%r4-%r2-%r3-%r5
- 需要使用3种颜色

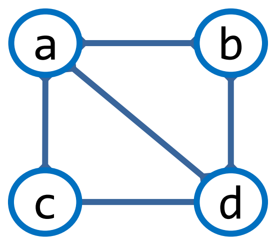


一类特殊的着色问题：弦图chordal graph

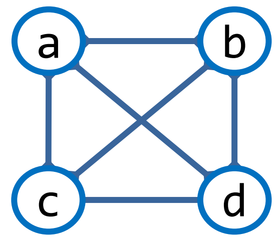
- 任意长度大于3的环都有弦（chord）
- 多项式时间可解



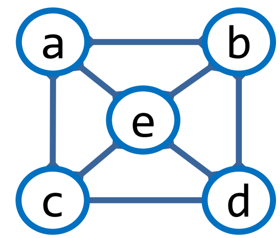
非弦图



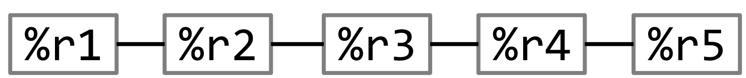
弦图



弦图

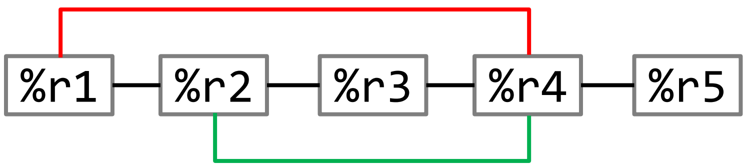


非弦图



尝试构造非弦图？

- 静态单赋值形式的干扰图都是chordal graph



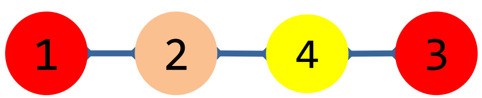
如果%r1和%r4冲突，则%r1一定和%r1和%r4之间的活跃变量冲突

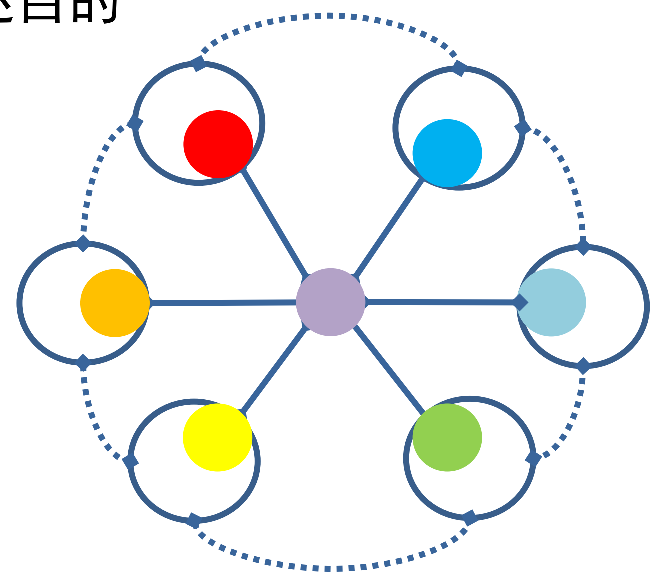
```
MOV -a(%rbp), %r1
MOV -b(%rbp), %r2
MOV %r1, %r3
ADD %r2, %r3
MOV %r2, %r4
ADD %r3, %r4
MOV %r1, XXX
MOV %r3, %r5
ADD %r4, %r5
MOV %r5, -r(%rbp)
```

∅
%r1
%r1,%r2
%1,%r2,%r3
%1,%r2,%r3
%1,%r3,%r4
%1,%r3,%r4
%r3,%r4
%r4,%r5
%r5

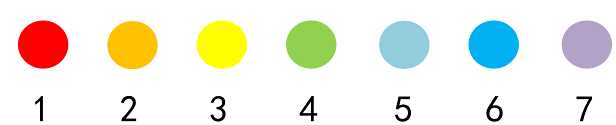
着色思路

- 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



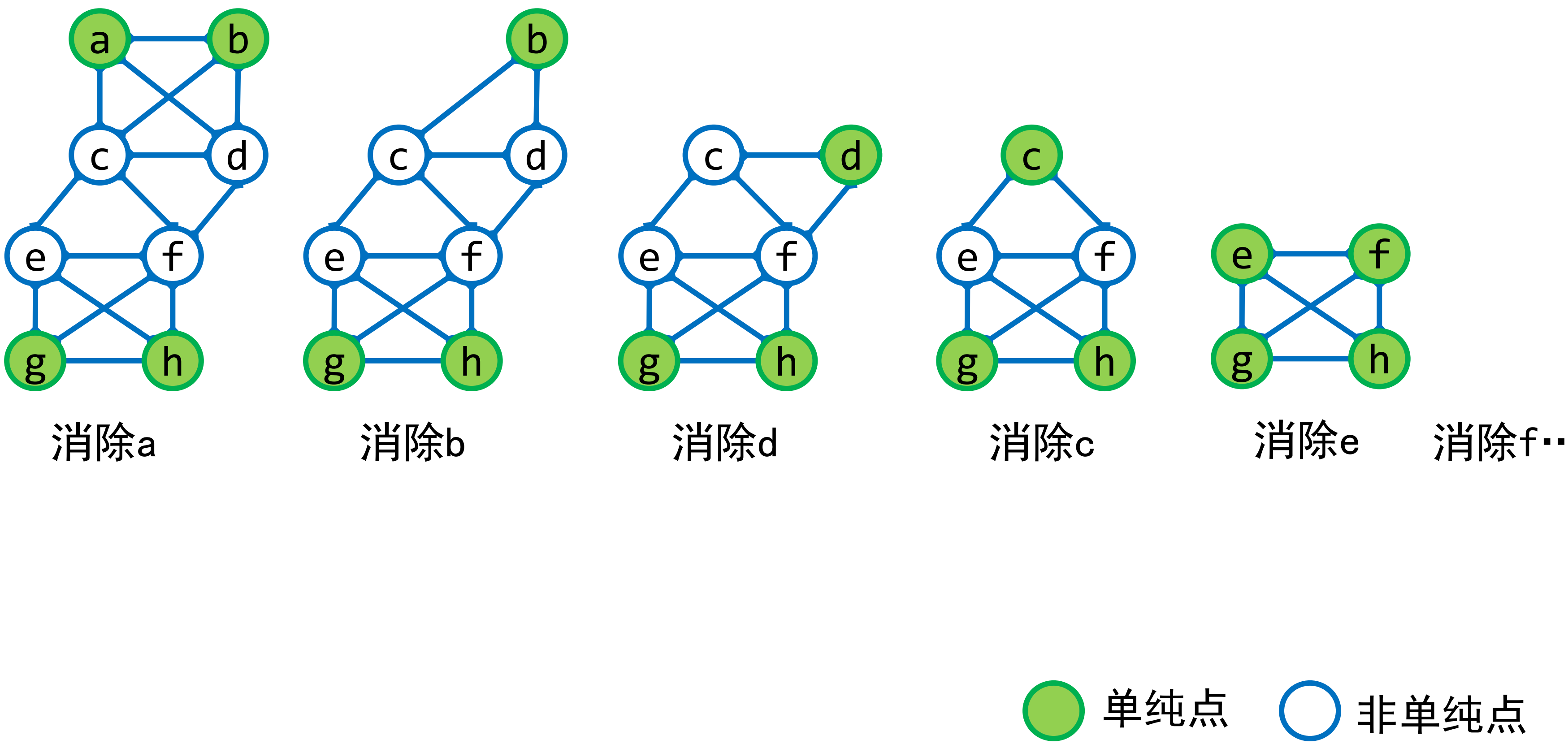


着色顺序：



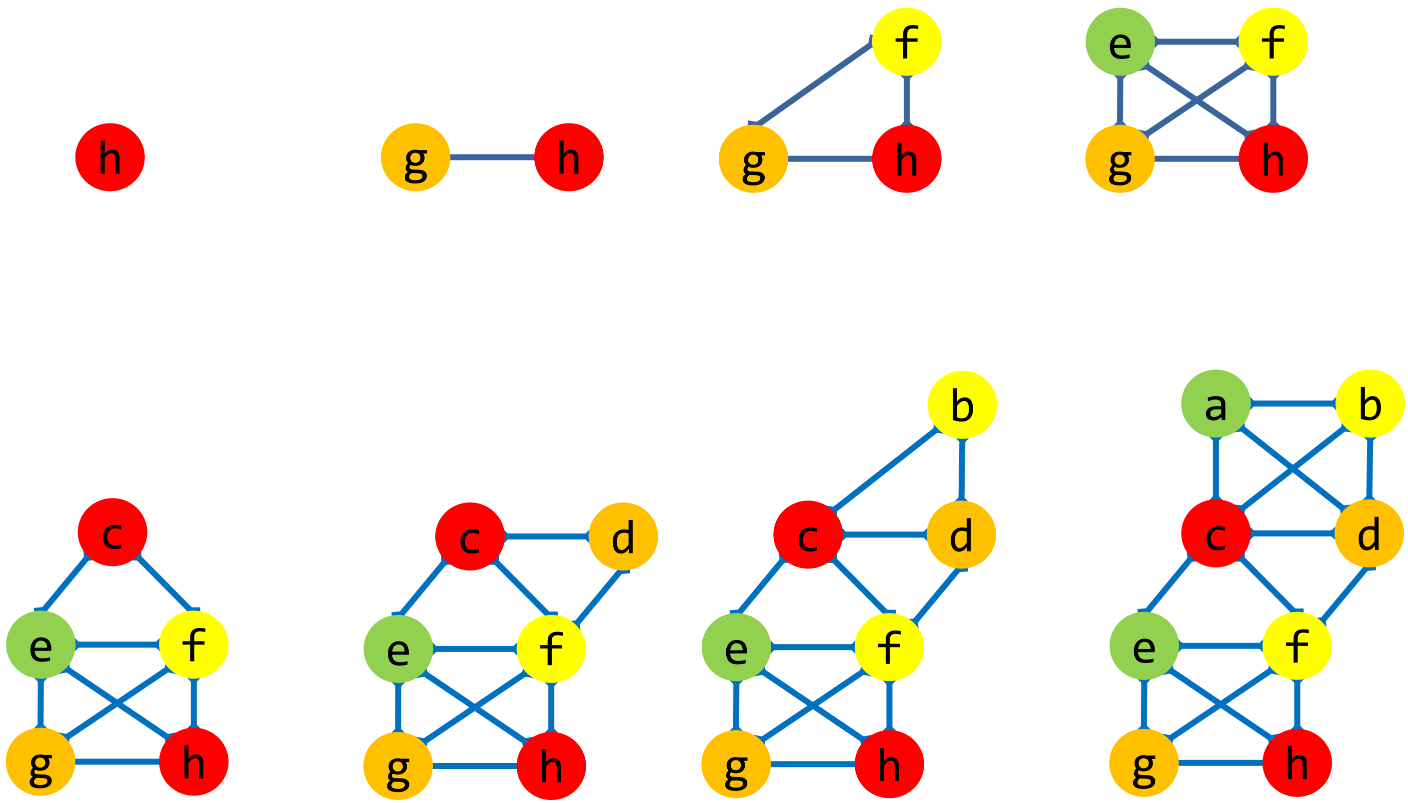
单纯消除序列simplicial elimination ordering

- **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

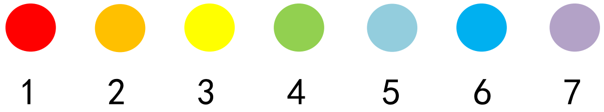


基于单纯消除序列着色

- 每次在已着色团的基础上新增一个点，连接该团的所有点

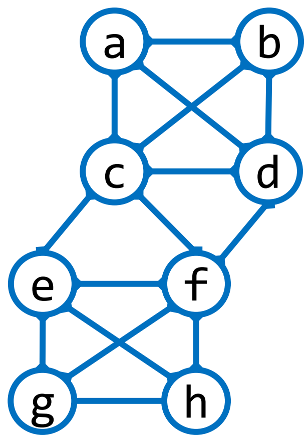


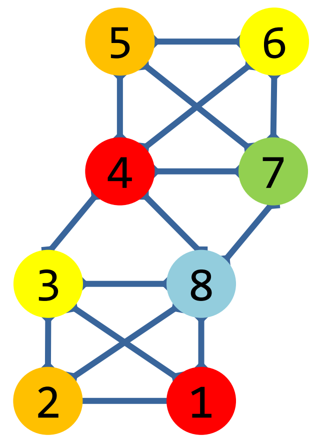
着色顺序：



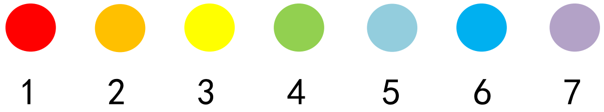
如果不遵循非单纯消除序列着色

- 所需颜色数可能需要超过最大团大小





着色顺序：



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。

%r1 — %r2 — %r3 — %r4 — %r5

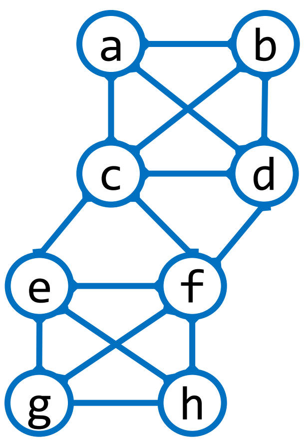
步骤	选取	%r1	%r2	%r3	%r4	%r5
		0	0	0	0	0
1	%r1		1	0	0	0
2	%r2			1	0	0
3	%r3				1	0
4	%r4					1
5	%r5					

算法参考

```
Maximum Cardinality Search
Input: G = (V, E)
Output: Simplicial elimination ordering v_1, …, v_n
For all v_i ∈ V
  w(v_i) = 0
Let W = V
For i = 1,..,n do
  Let v be a node with max weight in W
  Set v_i = v
  For all u ∈ W ∩ N(v)
    w(u) = w(u) + 1
  W = W\{v}
```

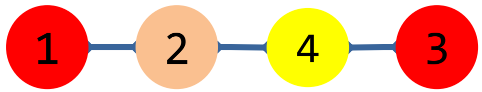
练习

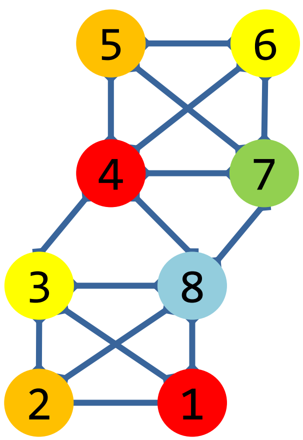
- 求下列冲突图的单纯消除序列

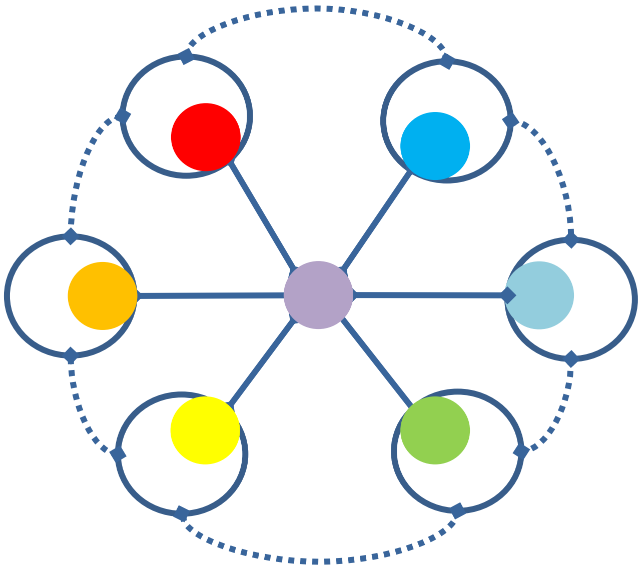


步骤	选取	a	b	c	d	e	f	g	h
		0	0	0	0	0	0	0	0
1	a		1	1	1	0	0	0	0
2	b			2	2	0	0	0	0
3	c				3	1	1	0	0
4	d					1	2	0	0
5	f					2		1	1
6	e							2	2
7	g								3
8	h								

思考1：为什么能得到单纯消除序列？



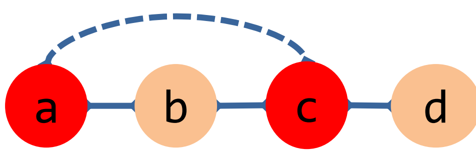


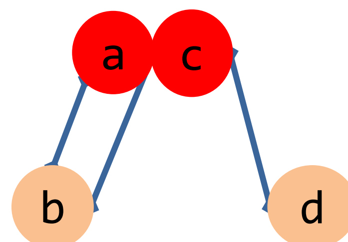


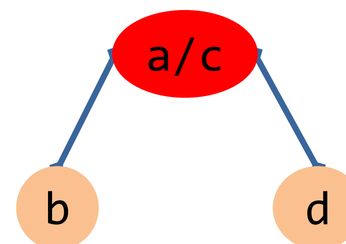
着色顺序：

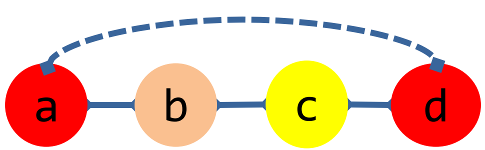
基于Coalesce的方法

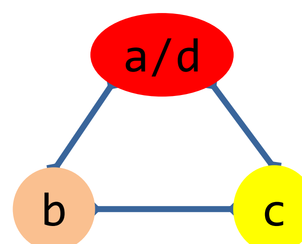
- 两个虚拟寄存器可以使用同一个物理寄存器
- 如何判断是否可以coalesce?
  - 不增加所需颜色K的数量
  - Briggs的方法：合并节点的边数≥K的邻居少于K个
  - George的方法
  - ...











大纲

一、寄存器分配

二、着色问题

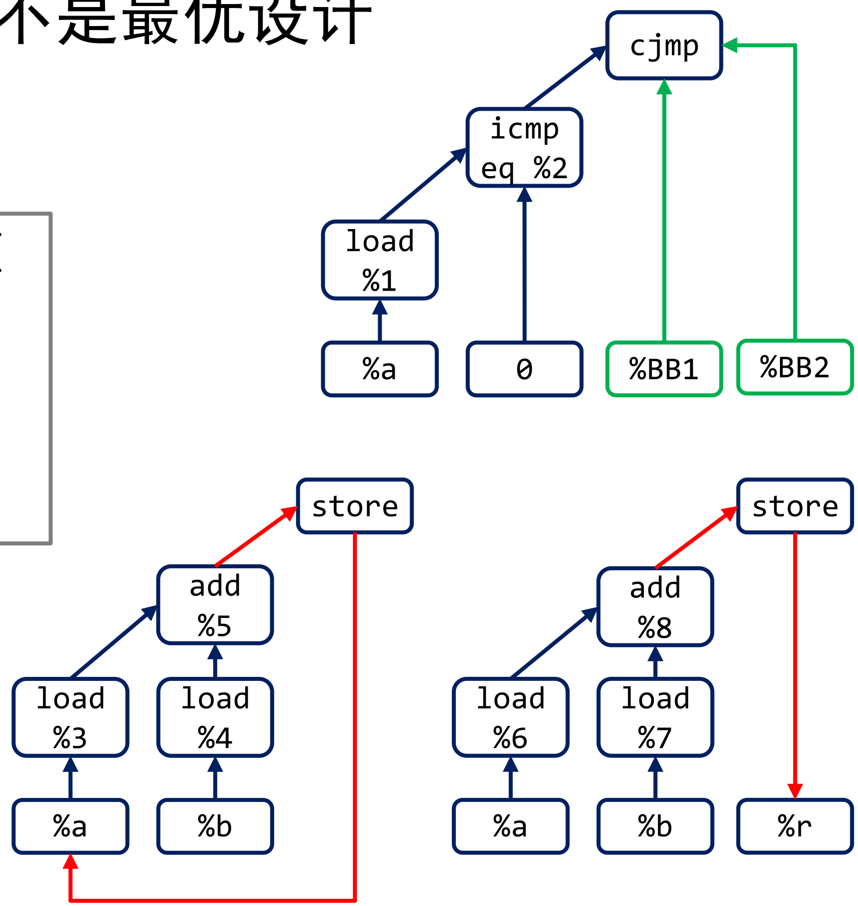
三、着色算法

四、更多考量

跨代码块的问题

- 目前的中间代码设计无法复用跨代码块的临时结果
- 代码块结束前临时结果都spill到内存中，下次使用时重新load
- 方便编译器实现，但不是最优设计
  - 可以先分配，后优化

```
fn foo(a:int, b:int)->int {
    if(a==0)
        a = a + b;
    let r:int = a + b;
    return r;
}
```



改进编译过程

- 虚拟寄存器编号可以跨代码块直接使用，无需重复加载
  - 静态单赋值形式（下节课会详细介绍SSA）
- 不在中间代码引入虚拟寄存器，直接基于内存计算
  - 为每个变量分配寄存器
  - 以函数为对象通过数据流分析算法提取变量活跃信息

```
fn foo(a:int, b:int)->int {
    if(a==0)
        a = a + b;
    let r:int = a + b;
    return r;
}
```

IR=>SSA

- SSA：未重新赋值前无需重复load
- 跨代码块活跃性分析

```
fn foo(a:int, b:int)->int {
    if(a==0)
        a = a + b;
    let r:int = a + b;
    return r;
}
```

```
%BB0:
    %a = stackalloc i32;
    %b = stackalloc i32;
    %r = stackalloc i32;
    store %-1, %a;
    store %-2, %b;
    %1 = load i32, %a;
    %2 = icmp eq i32 %1, 0;
    cjmp i1 %2, %BB1, %BB2;
%BB1:
    %3 = load i32, %a;
    %4 = load i32, %b;
    %5 = add i32 %3, %4;
    store i32 %5, %a;
    jmp %BB2;
%BB2:
    %6 = load i32, %a;
    %7 = load i32, %b;
    %8 = add i32 %6, %7;
    store i32 %8, %r;
    %9 = load i32, %r;
    ret %9;
```

```
%BB0:
    %a = stackalloc i32;
    %b = stackalloc i32;
    %r = stackalloc i32;
    store %-1, %a;
    store %-2, %b;
    %1 = load i32, %a;
    %2 = icmp eq i32 %1, 0;
    cjmp i1 %2, %BB1, %BB2;
%BB1:
    %4 = load i32, %b;
    %5 = add i32 %1, %4;
    store i32 %5, %a;
    jmp %BB2;
%BB2:
    %6 = phi(%1:%BB0,%5:%BB1)
    %7 = load i32, %b;
    %8 = add i32 %6, %7;
    store i32 %8, %r;
    %9 = load i32, %r;
    ret %9;
```

分析方法

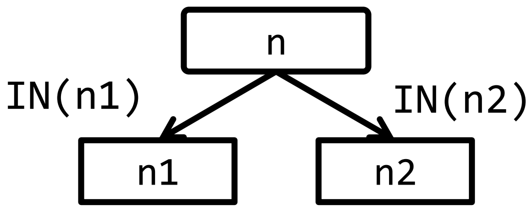
```
%BB0:
%a = stackalloc i32;
%b = stackalloc i32;
%r = stackalloc i32;
store %-1, %a;
store %-2, %b;
%1 = load i32, %a;
%2 = icmp eq i32 %1, 0;
cjmp i1 %2, %BB1, %BB2;
```

```
%BB1:
%4 = load i32, %b;
%5 = add i32 %1, %4;
store i32 %5, %a;
jmp %BB2;
```

```
%BB2:
%6 = phi(%1:%BB0,%5:%BB1)
%7 = load i32, %b;
%8 = add i32 %6, %7;
store i32 %8, %r;
%9 = load i32, %r;
ret %9;
```

- 为每条指令分配一个编号
  - BB0-1: %a = stackalloc i32
  - BB0-2: %b = stackalloc i32
  - ...
- 结果存储:
  - IN(n): 节点的入向属性集合
  - OUT(n): 节点的出向属性集合
- 反向遍历控制流图:
  - 如遇到指令: x = load a
    - $\mathrm{Gen(n)} = \{\mathrm{a}\}$
    - $\mathrm{KILL(n)} = \{x\}$
  - 如遇到指令: x = add a, b;
    - $\mathrm{Gen(n)} = \{\mathrm{a}, \mathrm{b}\}$
    - $\mathrm{KILL(n)} = \{x\}$
  - …
  - $\mathrm{IN(n)} = (\mathrm{OUT(n)} - \mathrm{KILL(n)}) \cup \mathrm{Gen(n)}$

控制流和环的处理



$$\mathrm{OUT}(n) = \bigcup_{n' \in \mathrm{successor}(n)} \mathrm{IN}(n')$$

```
For (each node n):
    IN[n] = OUT[n] = Ø
Repeat:
    For(each node n):
        For(each n’s successor p)
            OUT[n] = OUT[n] ∪ IN[p]
        IN(n)=(OUT[n]-KILL(n)) ∪ Gen(…)
Until IN[n] and OUT[n] stops changing for all n
```

单调递增，上界为所有变量

循环一定会终止

应用

%BB0:

```
%a = stackalloc i32;
%b = stackalloc i32;
%r = stackalloc i32;
store %-1, %a;
store %-2, %b;
%1 = load i32, %a;
%2 = icmp eq i32 %1, 0;
cjmp i1 %2, %BB1, %BB2;
```

%BB1:

```
%4 = load i32, %b;
%5 = add i32 %1, %4;
store i32 %5, %a;
jmp %BB2;
```

%BB2:

```
%6 = phi(%1:%BB0,%5:%BB1)
%7 = load i32, %b;
%8 = add i32 %6, %7;
store i32 %8, %r;
%9 = load i32, %r;
ret %9;
```

n	IN[n]	OUT[n]
BB0-1	--	
BB0-2		
BB0-3		
BB0-4		
BB0-5		
BB0-6		
BB0-7		
BB0-8		{%1}
BB1-1		
BB1-2		
BB1-3		
BB1-4		{%5}
BB2-1	{%1,%5}	{%6}
BB2-2	{%6}	{%6,%7}
BB2-3	{%6,%7}	{%8}
BB2-4	{%8}	Ø
BB2-5	Ø	{%9}
BB2-6	{%9}	Ø

应用

```
%BB0:
%a = stackalloc i32;
%b = stackalloc i32;
%r = stackalloc i32;
store %-1, %a;
store %-2, %b;
%1 = load i32, %a;
%2 = icmp eq i32 %1, 0;
cjmp i1 %2, %BB1, %BB2;
```

```
%BB1:
%4 = load i32, %b;
%5 = add i32 %1, %4;
store i32 %5, %a;
jmp %BB2;
```

```
%BB2:
%6 = phi(%1:%BB0,%5:%BB1)
%7 = load i32, %b;
%8 = add i32 %6, %7;
store i32 %8, %r;
%9 = load i32, %r;
ret %9;
```

n	IN[n]	OUT[n]
BB0-1	{%-1,%-2}	{%-1,%-2}
BB0-2	{%-1,%-2}	{%-1,%-2}
BB0-3	{%-1,%-2}	{%-1,%-2}
BB0-4	{%-1,%-2}	{%-2}
BB0-5	{%-2}	∅
BB0-6	∅	{%1}
BB0-7	{%1}	{%1,%2}
BB0-8	{%1,%2}	{%1}
BB1-1	{%1}	{%1,%4}
BB1-2	{%1,%4}	{%5}
BB1-3	{%5}	{%5}
BB1-4	{%5}	{%5}
BB2-1	{%1,%5}	{%6}
BB2-2	{%6}	{%6,%7}
BB2-3	{%6,%7}	{%8}
BB2-4	{%8}	∅
BB2-5	∅	{%9}
BB2-6	{%9}	∅

其它考量因素

- 寄存器不足时应优先指派（或spill）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spill次数

小结

- 寄存器分配问题:
  - 预分配寄存器
  - 干扰图=>着色问题
  - spill
- 着色问题求解:
  - 线性分配
  - 贪心法着色
  - 基于单纯消除序列方法求解

思考2：指令调度可否优化寄存器使用？

- 构造一个程序说明