

编译技术



胡春明

hucm@buaa.edu.cn

2019.9-2019.12

第十五章 目标代码生成

面向目标体系结构的代码生成和优化技术



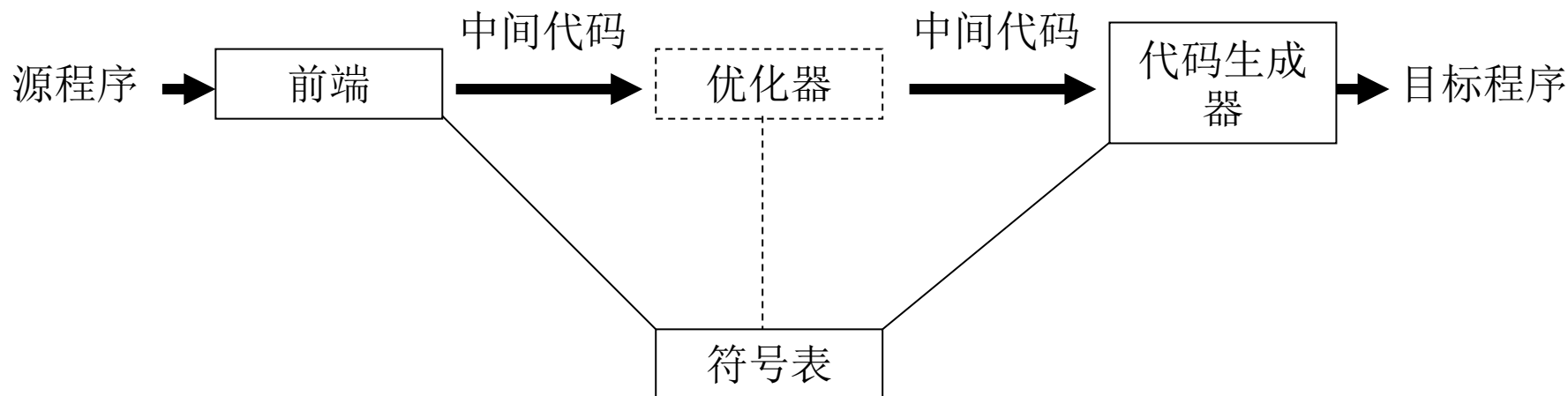
编译过程是指将**高级语言程序**翻译为等价的**目标程序**的过程。

习惯上是将编译过程划分为5个基本阶段：

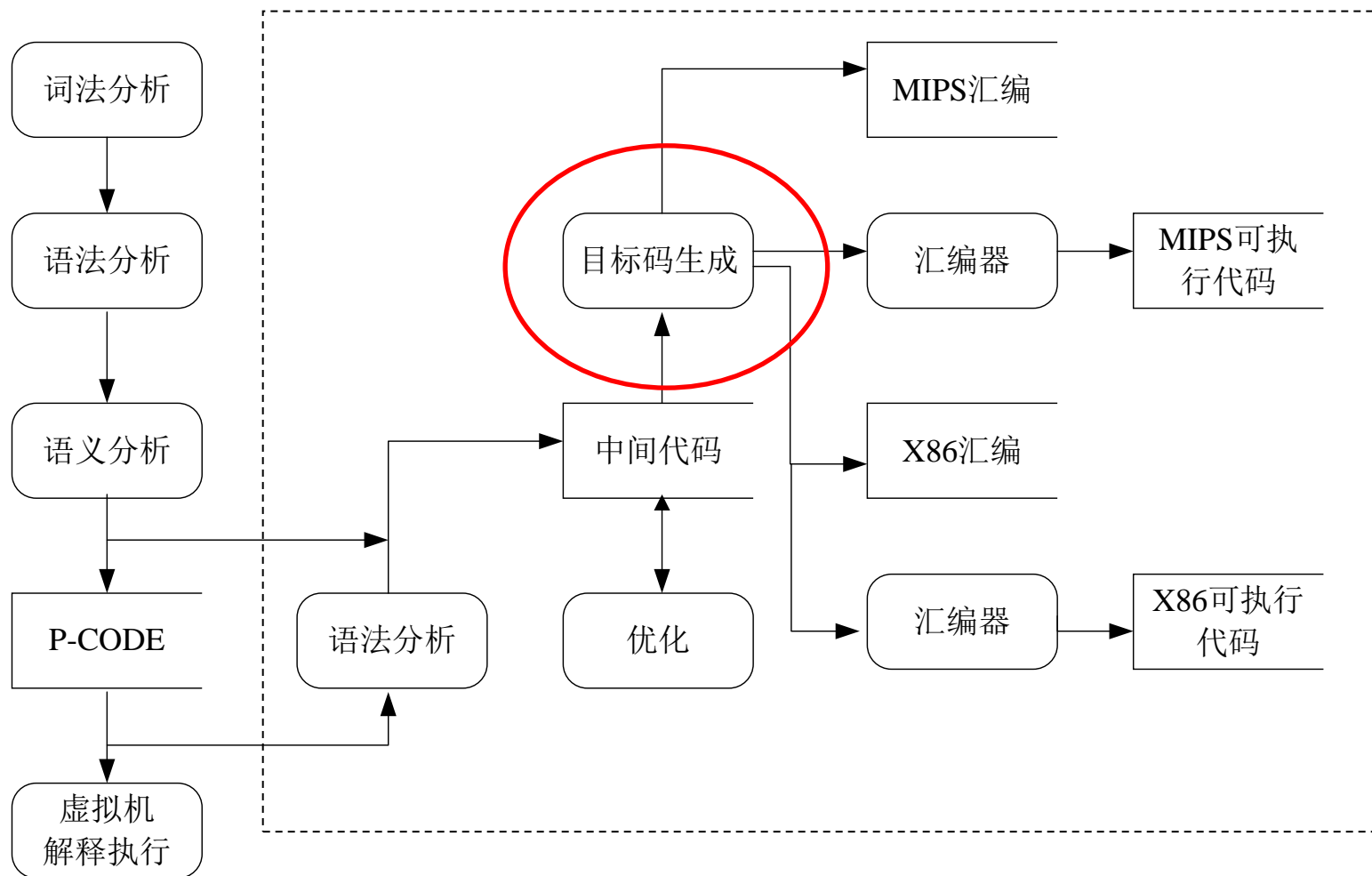


目标代码生成：任务

代码生成器在编译系统中的位置



教学编译器架构



代码生成器的输入

- 源程序的中间表示
 - 线性表示（波兰式）
 - 三地址码（四元式）
 - 栈式中间代码（P-CODE/Java Bytecode）
 - 图形表示
- 符号表信息

代码生成器对输入的要求

- 编译器前端已经将源程序扫描、分析和翻译成足够详细的**中间表示**
- 中间语言中的**标识符**表示为目标机器能够直接操作的变量（位、整数、浮点数、指针等）
- 完成了必要的类型检查，类型转换/检测操作已经加入到中间语言的必要位置
- **完成语法和必要的语义检查**，代码生成器可以认为输入中没有与语法或语义错误

目标程序的种类

- **汇编语言**

- 生成宏汇编代码，再由汇编程序进行编译，连接，从而生成最终代码（.S/.ASM文件）

- **包含绝对地址的机器语言**

- 执行时必须被载入到地址空间中（相对）固定的位置
- EXE (MS-WIN)、COM (MS-WIN)、A.OUT (Linux)

- **可重定位的机器语言**

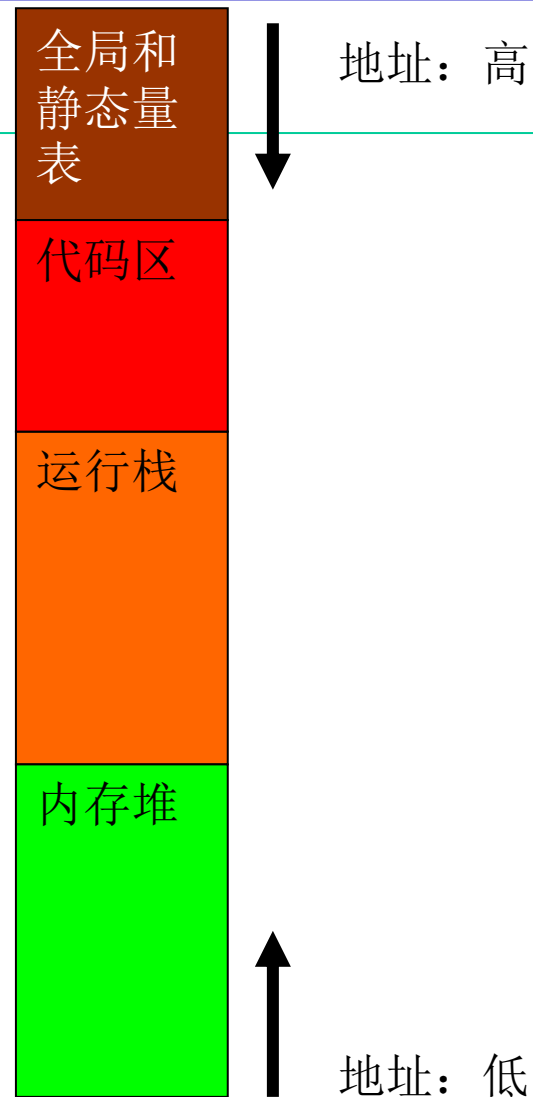
- 一组可重定位的模块/子程序可以用连接器装配后生成最终的目标程序（.obj/.o文件组）
- 可动态加载的模块/子程序（DLL/.SO动态连接库）

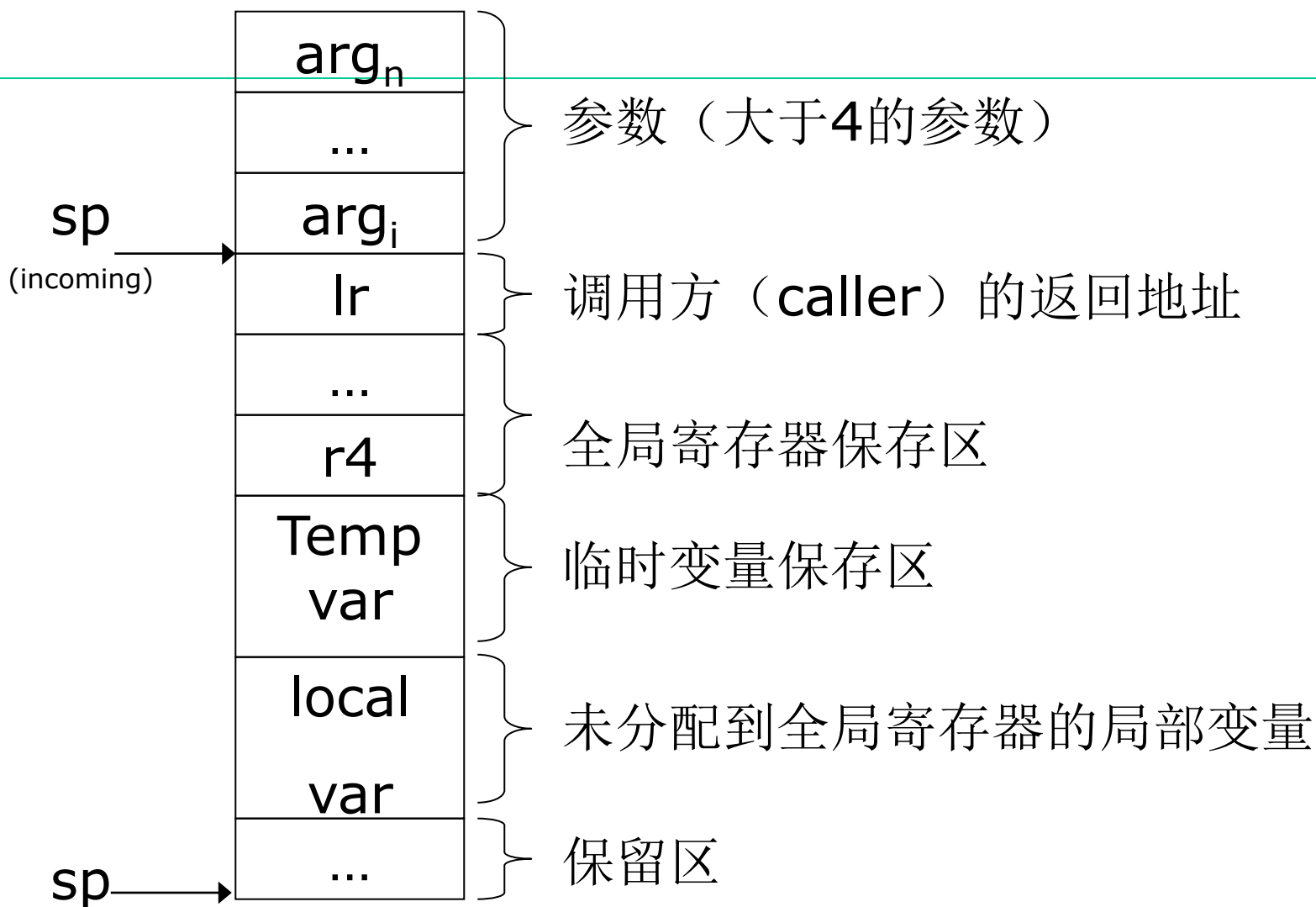
面向特定的目标体系结构生成目标代码

- **目标体系结构**可以是：
 - 某种微处理器，如X86、MIPS、ARM等
 - 某种虚拟机或运行时系统，如Java虚拟机、C#运行时系统、P-code虚拟机等
- **虚拟机：**
 - P-code栈式虚拟机
 - 虚拟机的代码需要**解释器**解释或者**即时编译器**编译后才能运行

运行栈结构与地址空间

- **代码区**
 - 存放目标代码
- **静态数据区**
 - 全局变量
 - 静态变量
 - 部分常量，例如字符串
- **动态内存区**
 - 也被称为内存堆Heap
 - 程序员管理：C、C++
 - 自动管理（内存垃圾收集器）：Java、Ada
- **程序运行栈**
 - 活动记录
 - 函数调用的上下文现场
 - 由调用方保存的一些临时寄存器
 - 被调用方保存的一些全局寄存器



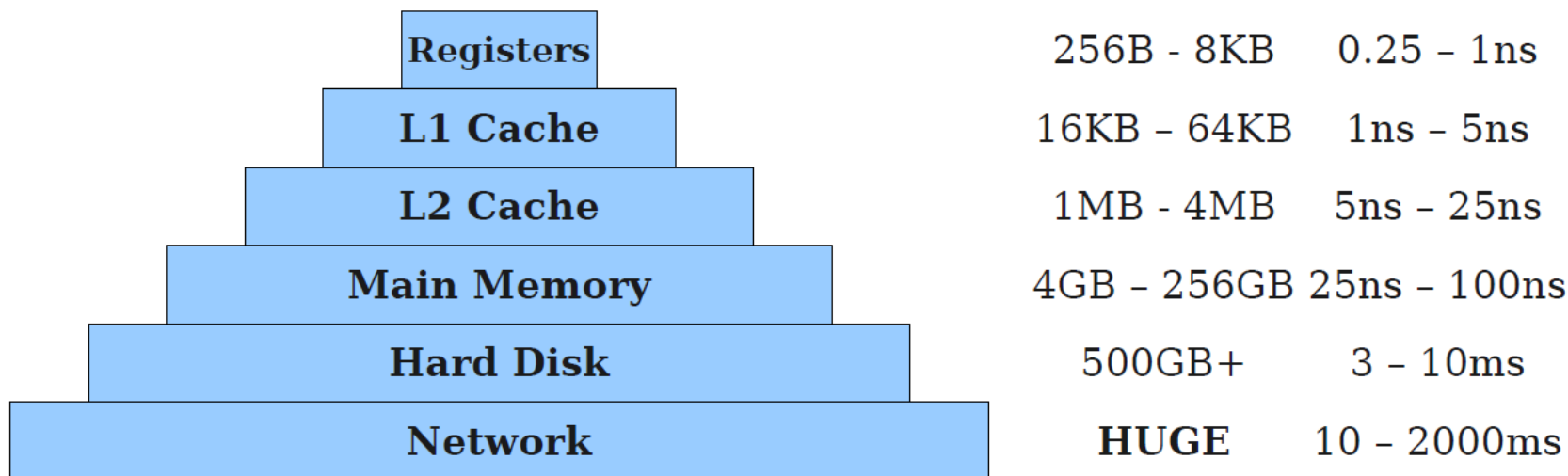


面向微处理器体系结构的代码生成技术

- 主要内容：
 1. 目标代码地址空间的划分，目标体系结构上存贮单元（如寄存器和内存单元）的分配和指派
 2. 从中间代码（或者源代码）到目标代码转换过程中所进行的指令选择
 3. 面向目标体系结构的优化

面向微处理器体系结构的代码生成技术

- 技术挑战：
 1. 在内存找到目标，并最大限度的利用目标体系结构特点（存储层次、缓存、指令架构）
 2. 这一过程对程序员“透明”



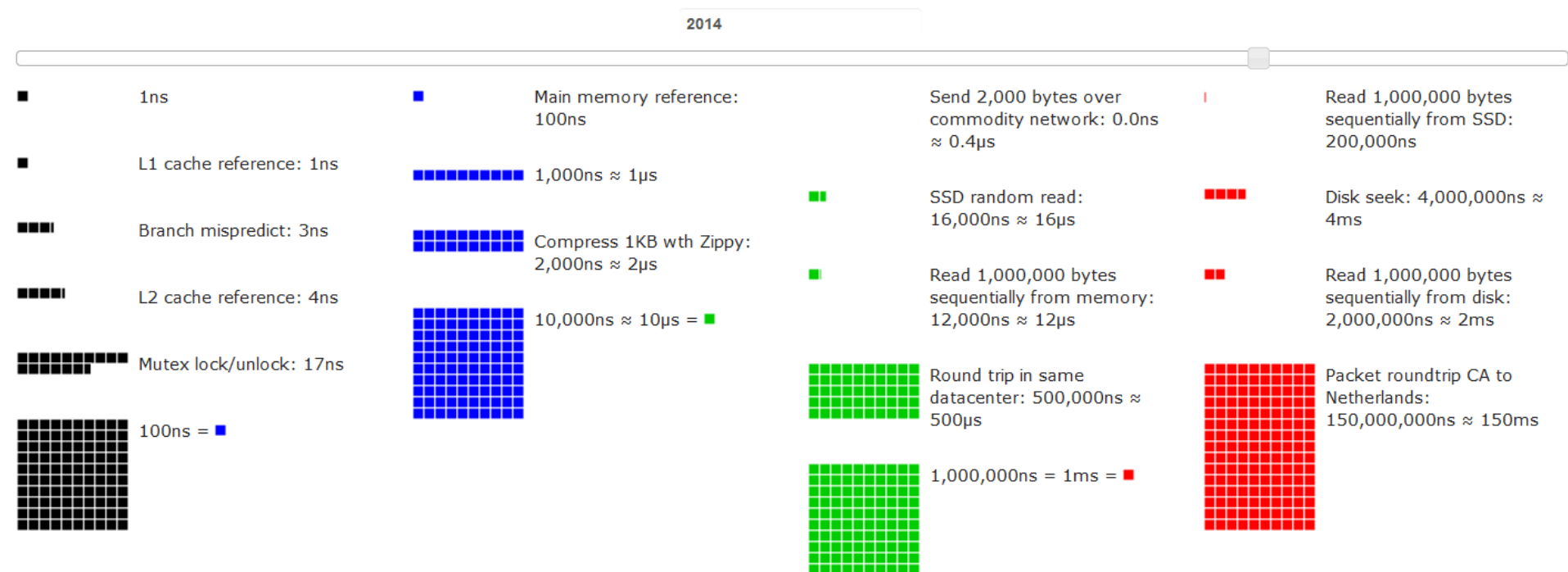
存储器层次架构

存储器层次架构

Latency Numbers Every Programmer Should Know

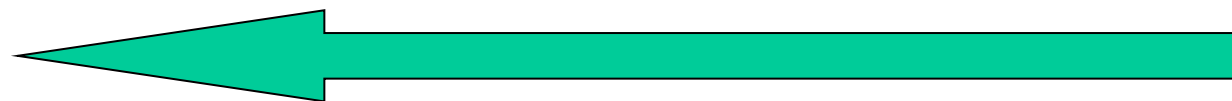
- http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html

Latency Numbers Every Programmer Should Know



存储器层次架构

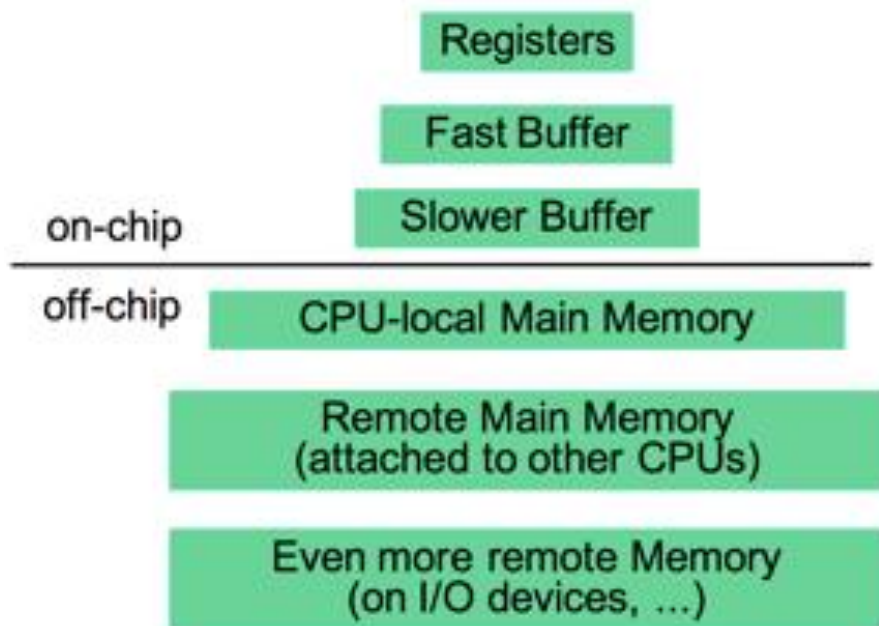
寄存器、缓存、内存、硬盘的存储访问特性



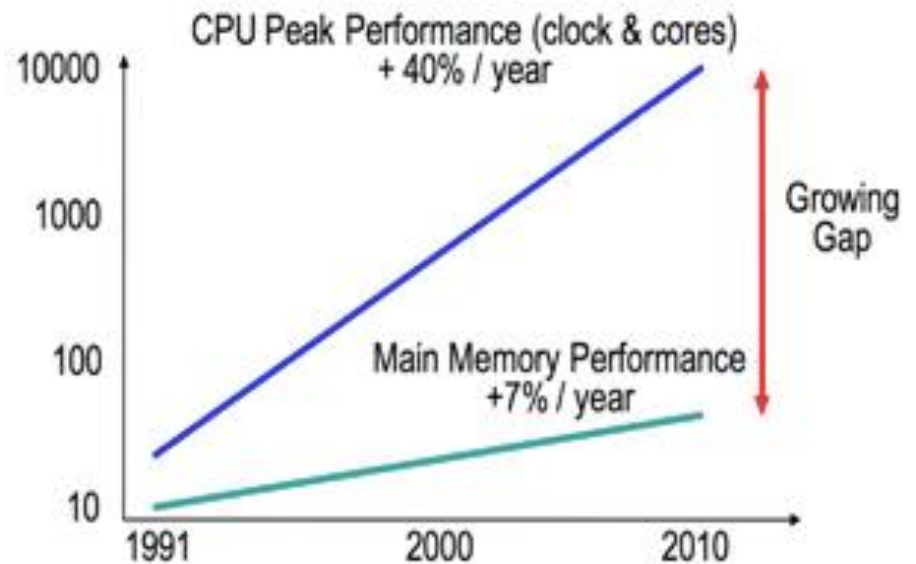
Level	1	2	3	4
Called	Registers	Cache	Main memory	Disk storage
Typical size	< 1 KB	< 16 MB	< 16 GB	> 100 GB
Implementation technology	Custom memory with multiple ports, CMOS	On-chip or off-chip CMOS SRAM	CMOS DRAM	Magnetic disk
Access time (in ns)	0.25 -0.5	0.5 to 25	80-250	5,000,000
Bandwidth (in MB/sec)	20,000-100,000	5,000-10,000	1000-5000	20-150
Managed by	Compiler	Hardware	Operating system	Operating system/operator
Backed by	Cache	Main memory	Disk	CD or Tape

寄存器的访问速度基本可以保证处理器在每个时钟周期内访问到需要的数据

存储器层次架构



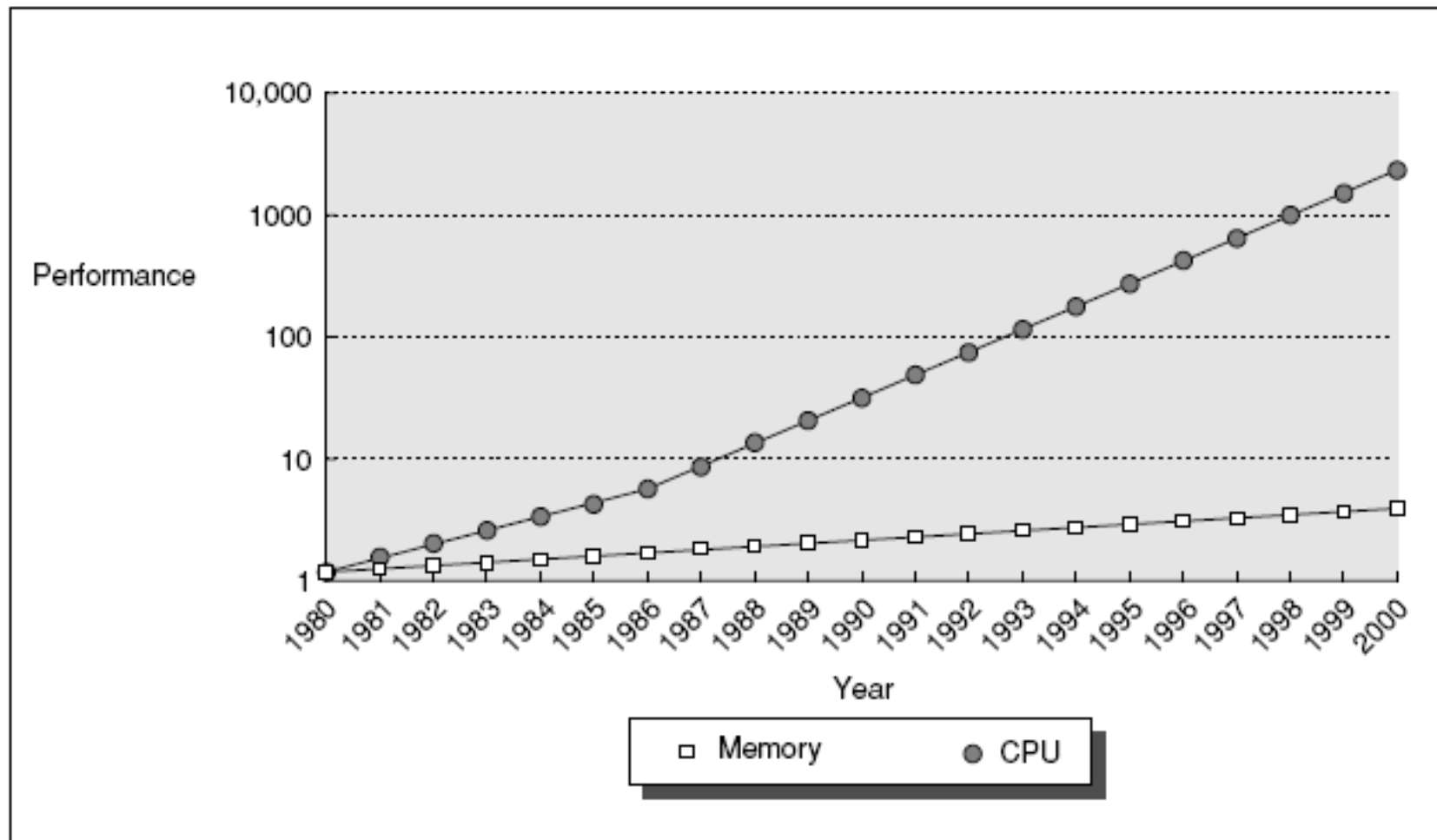
[The Memory Hierachy]



Access latency to main memory today up to 300 cycles

[The Memory Wall]

1980 ~ 2000年，CPU性能和内存访问性能的提高



策略：尽可能多的访问寄存器，利用缓存

- **但是，寄存器的数量极其有限**
 - 32位X86微处理器上有8个通用寄存器
 - Xscal/ARM, MIPS: 大约16~32个通用寄存器
 - 需要“聪明”的分配策略
- **对缓存的利用，对于大型数据结构有用**
 - 缓存的管理单位是：缓存行。（数十或上百字节）
 - 每次从内存载入的是一组地址连续的数据，而不仅仅是被访问数据

现代微处理器体系结构特点

12.1 现代微处理器体系结构简介

指令集 (Instruction Set)

流水线和指令级并行

Pipeline and Instruction Level Parallelism

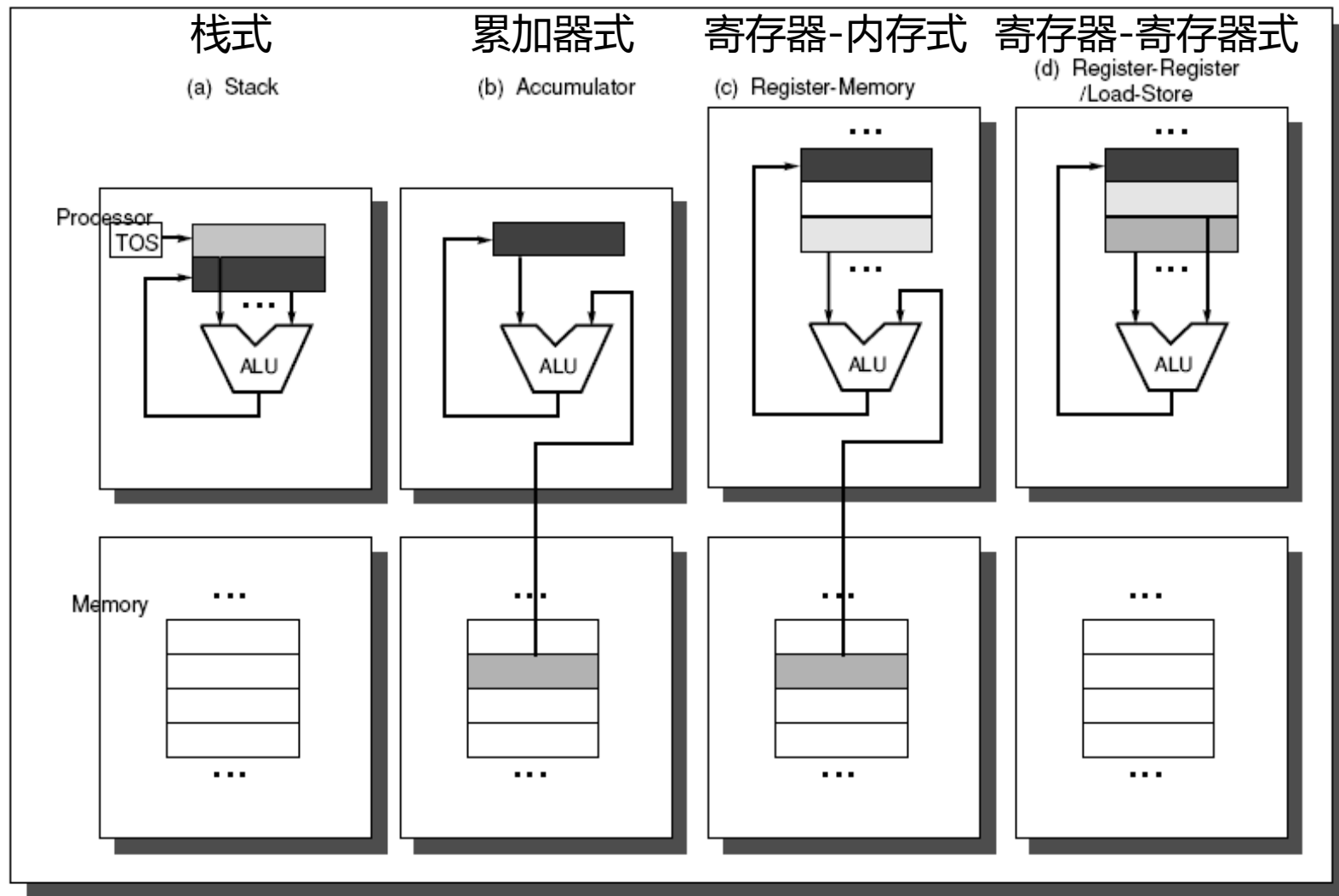
存储结构和I/O

Memory Hierarchy and I/O Systems

多处理器和线程级并行

Multiprocessor and Thread Level Parallelism

指令集架构

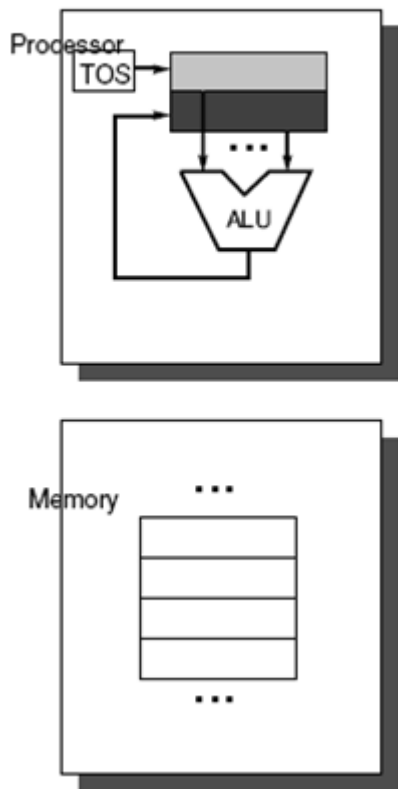


不同：算术逻辑单元ALU对存储单元的访问方式不同

(1) 栈式指令集架构

栈式

(a) Stack



类似于P-code和Java虚拟机

$C=A+B$

代码:

PUSH A

PUSH B

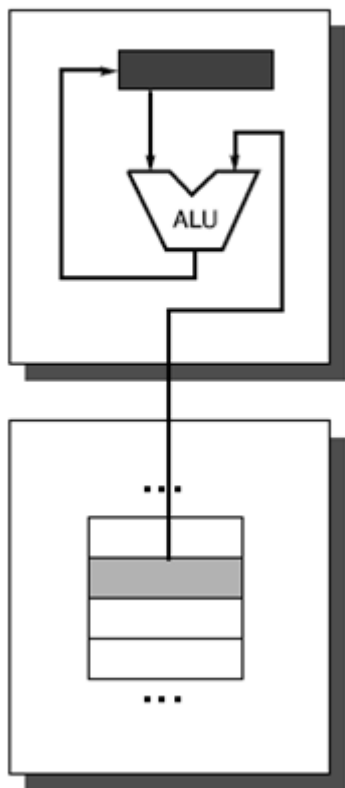
ADD

POP C

(2) 累加器式指令集架构

累加器式

(b) Accumulator



$C=A+B$

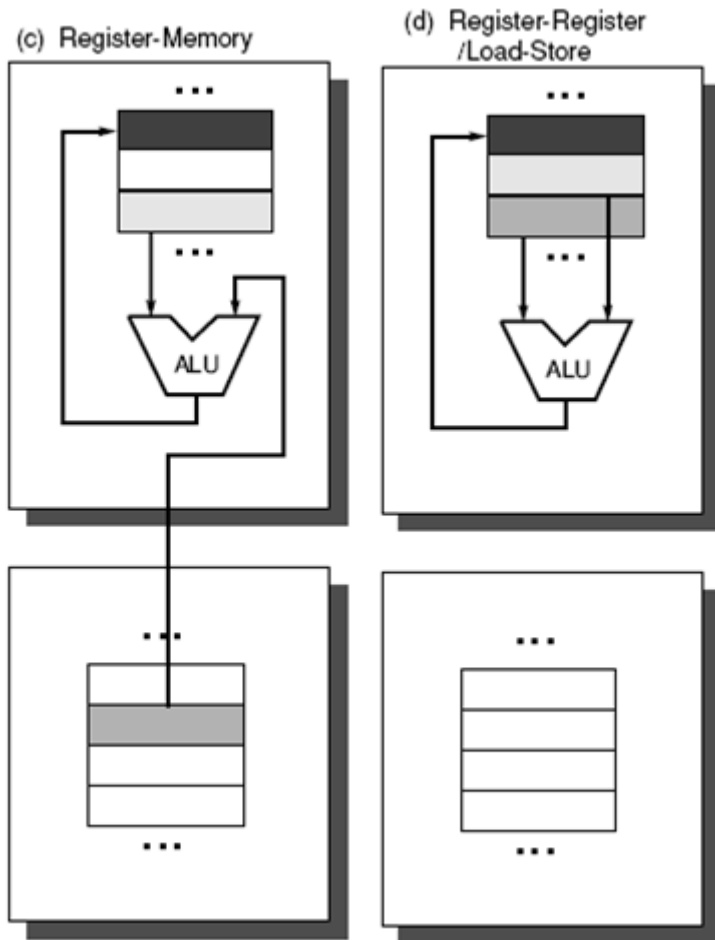
代码:

```
LOAD A
ADD B
STORE C
```

代码短了，开销不一定小
直接访问内存

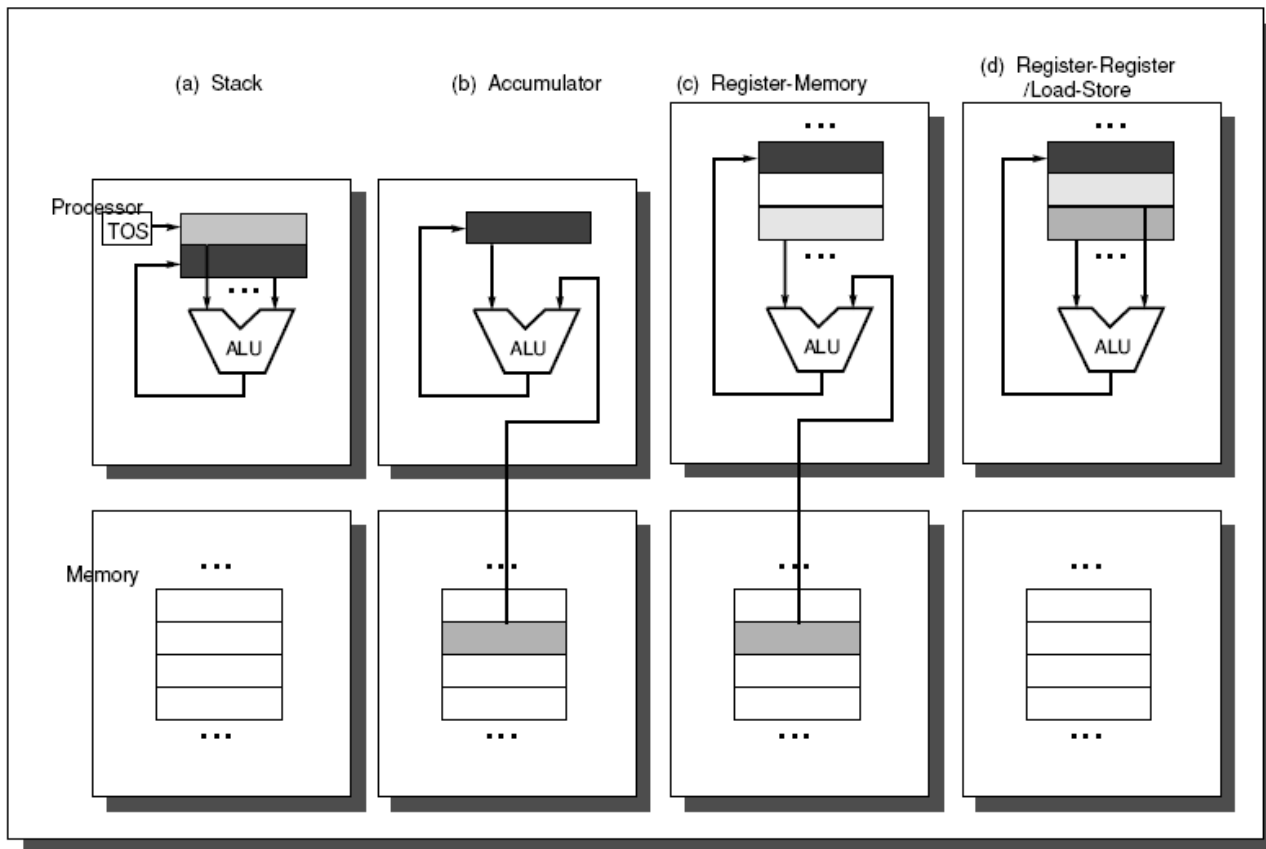
寄存器架构

寄存器-内存式 寄存器-寄存器式



- 寄存器-内存指令集架构
- 寄存器-寄存器指令架构

- **区别：** 是否可以直接内存寻址
- **共性：** 内部有多个寄存器可以直接作为ALU指令的任一操作数
- 优点：
 - 减少内存访问
 - 减少指令数



Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

$$D = (A * B) + (B * C)$$

- 栈式架构

PUSH A

PUSH B

MUL

PUSH B

PUSH C

MUL

ADD

POP D

8条指令， 5条内存访问

- 寄存器-寄存器架构

LOAD R1, A

LOAD R2, B

LOAD R3, C

MUL R1, R2, R4

MUL R2, R3, R5

ADD R4, R5, R5

STORE R5, D

7条指令， 4条存储访问

- **寄存器-内存指令集架构处理器：**

Complex Instruction Set Computers (CISC)

- Intel X86架构处理器
- 曾经十分辉煌，现已退出历史舞台的DEC VAX

- **寄存器-寄存器指令集架构处理器：**

Reduced Instruction Set Computer (RISC)

- Alpha、ARM、MIPS、PowerPC、SPARC等

寄存器分配 (Register Allocation)

- 寄存器通常分为
 - **通用寄存器**
 - X86: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, etc
 - ARM: R0~R15, etc
 - **专用寄存器**
 - X86: 浮点寄存器栈, etc
- 通用寄存器
 - 保留寄存器
 - 例如, X86的ESP栈指针寄存器, ARM的返回寄存器LR
 - 调用方保存的寄存器——**临时寄存器**
 - X86: EAX, ECX, EDX
 - ARM: R0~R3, R12, LR
 - 被调用方保存的寄存器——**全局寄存器**
 - X86: EBX, ESI, EDI, EBP (ESP为运行栈寄存器, 不参与寄存器分配)
 - ARM: R4~R11

- **寄存器特点：**
 - 访问快
 - 有些寄存器和指令执行有关系
 - 数量少，分配策略
- **寄存器分配的目标：**
 - 将变量与寄存器之间建立对应关系
 - 从程序优化的角度来说，我们希望所有指令的执行都仅在寄存器中完成

- **全局寄存器分配：**

- “**全局**” 相对于 “基本块” 而言，不是 “程序全局”
- 全局寄存器分配的对象主要是函数的局部变量，包括函数入口参数。

- **分配原则** **寄存器专属于线程！**

- 优先分配给跨基本块仍然活跃的变量，尤其是循环体内最活跃的变量
- 局部变量参与全局寄存器分配
 - 为了线程安全，全局变量/静态变量一般不参与全局寄存器分配。

如果全局/静态量参与寄存器分配?

- ◆ 全局变量和静态变量一般不参与全局寄存器分配，即使他们在某个循环体中被多次访问 ... WHY?
- ◆ 如果发生线程切换:
 - ◆ 当前的寄存器状态将作为线程现场被保存
 - ◆ 切入线程将恢复其此前保存的寄存器状态
 - ◆ 这就导致了其他线程无法得到该寄存器在此前线程中的值，程序运行可能会发生不可预知的错误。

寄存器分配：基线

```
a = b + c;
d = a;
c = a + d;
```

Param N	fp + 4N
...	...
Param 1	fp + 4
Stored fp	fp + 0
Stored ra	fp - 4
a	fp - 8
b	fp - 12
c	fp - 16
d	fp - 20

```
lw    $t0, -12(fp)
lw    $t1, -16(fp)
add   $t2, $t0, $t1
sw    $t2, -8(fp)

lw    $t0, -8(fp)
sw    $t0, -20(fp)

lw    $t0, -8(fp)
lw    $t1, -20(fp)
add   $t2, $t0, $t1
sw    $t2, -16(fp)
```

- **改进目标:**

- 尽可能映射更多的变量到寄存器
- 减少内存读写次数

- **要解决的问题:**

- 把哪个变量放到哪个寄存器?
- 如果寄存器用完了, 如何替换?

- **本质问题是：**

- 当活跃变量数量超过寄存器个数时，如何做出取舍

- **类比CPU调度：**

- 优化指标：公平性 vs 性能
- Round Robin (轮转) / **Weighted RR (加权轮转)**
- FIFO (先来先服务)
- LRU (最近使用优先)
- Bin Packing (装箱/背包)

权重是什么?

寄存器分配：引用计数

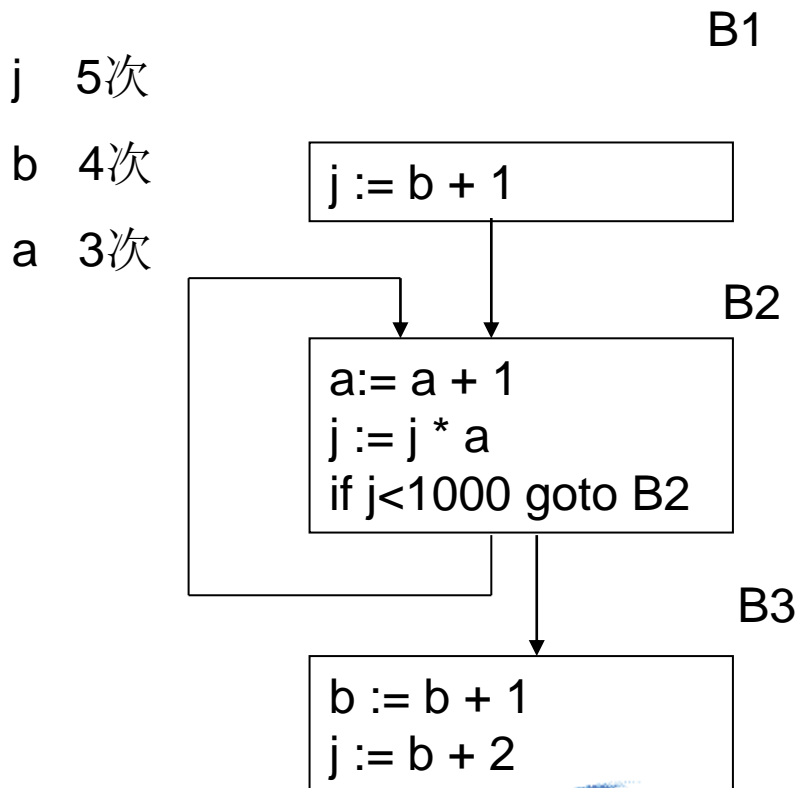
• 引用计数

- 通过统计变量在函数内被引用的次数，并根据被引用的特点赋予不同的权重，最终为每个变量计算出一个唯一的权值
- 根据权值的大小排序，将全局寄存器依次分配给权值最大的变量

引用计数

- **原则：** 如果一个局部变量被访问的次数较多，那么它获得全局寄存器的机会也较大
- **注意：** 出现在循环，尤其是内层嵌套循环中的变量的被访问次数应该得到一定的加权

3个局部变量，2个全局寄存器可供分配，谁将获得寄存器？



引用计数

- **分配算法**：如果有 N 个全局寄存器可供分配，则前 N 个变量拥有全局寄存器，其余变量在程序运行栈（活动记录）分配存贮单元
- **问题**：不再使用的变量不能及时释放寄存器
 - 如变量 a 在前期大量使用，后端程序中不适用了
 - 还是需要对“变量是否有用”的细粒度追踪

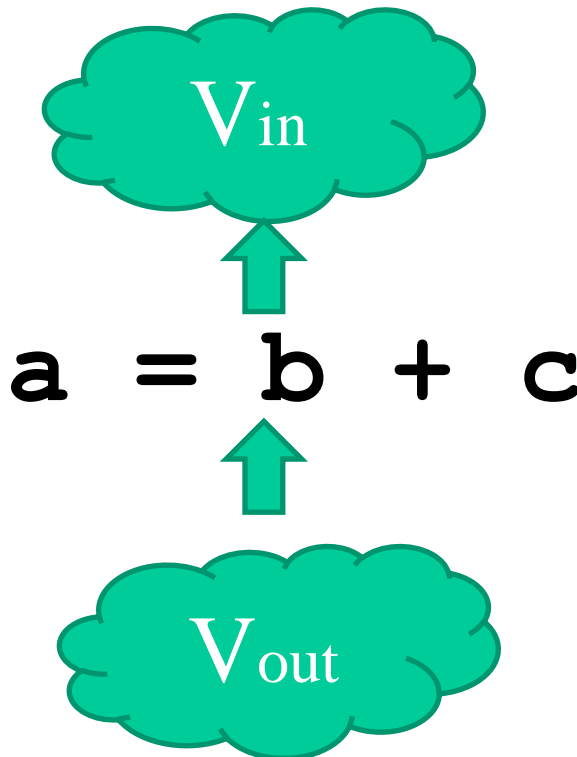
$$\text{out}[S] = \text{gen}[S] \cup (\text{in}[S] - \text{kill}[S])$$

$$\text{in}[S] = \text{use}[S] \cup (\text{out}[S] - \text{def}[S])$$

引用变量会产生新的数据流

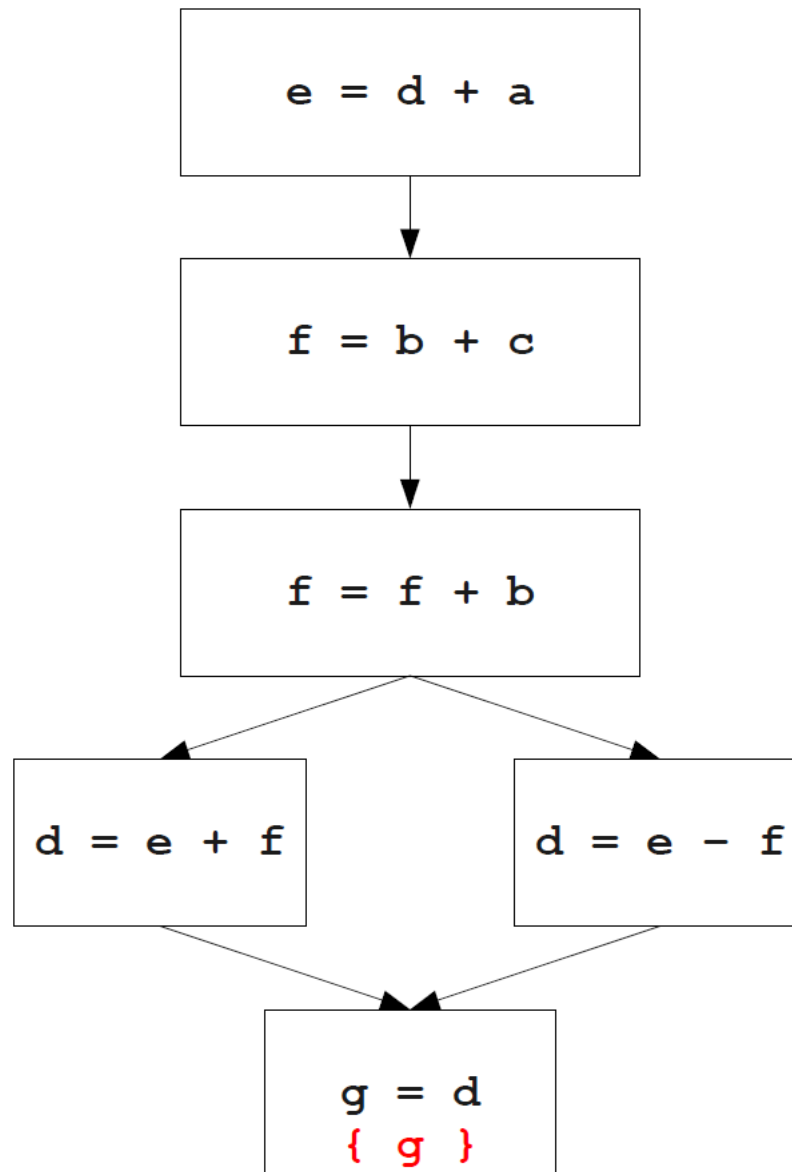
赋值会删除数据流

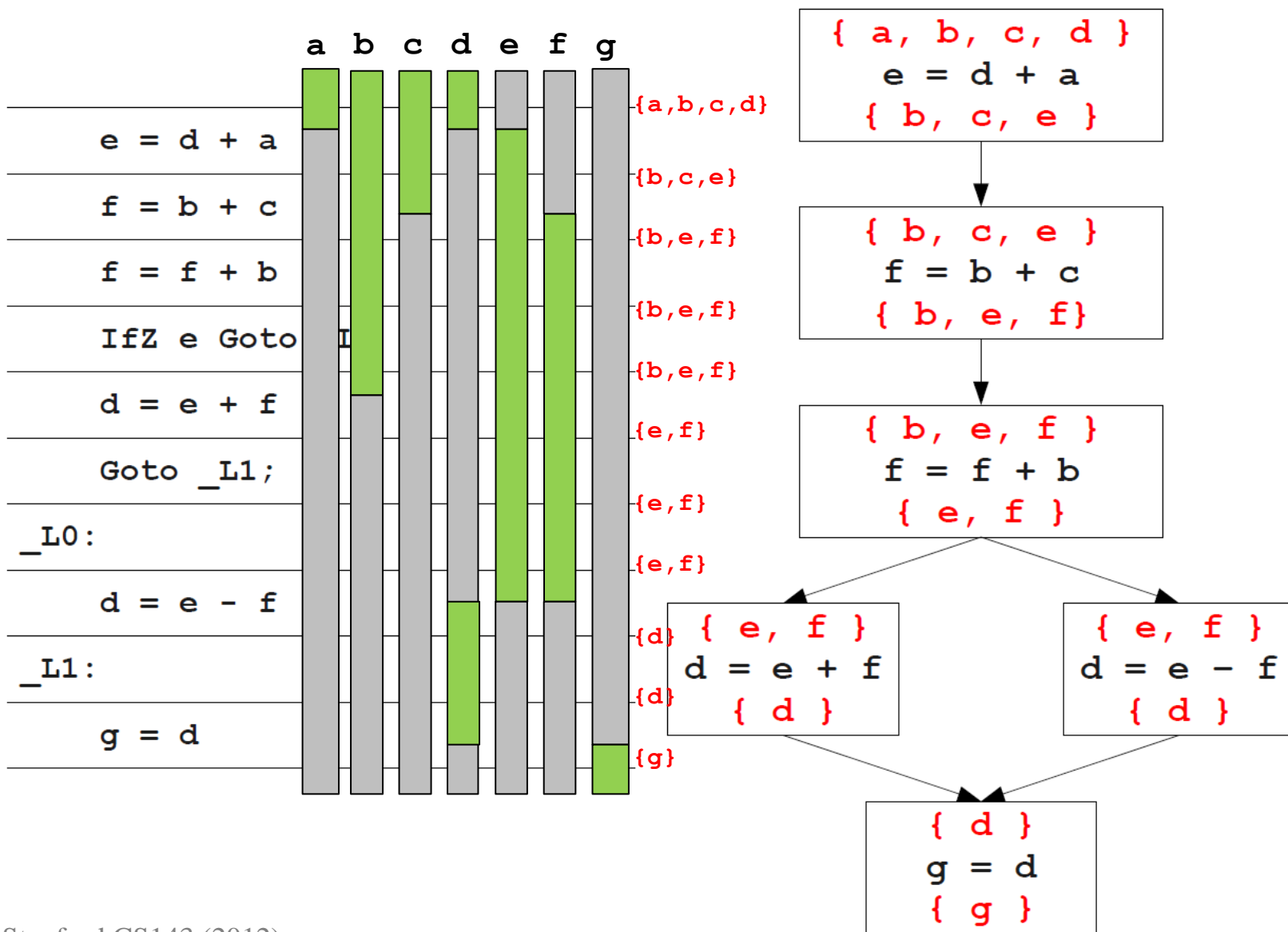
$$(L - \{a\}) \cup \{b, c\}$$



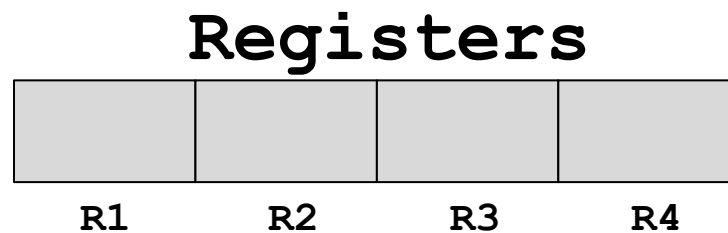
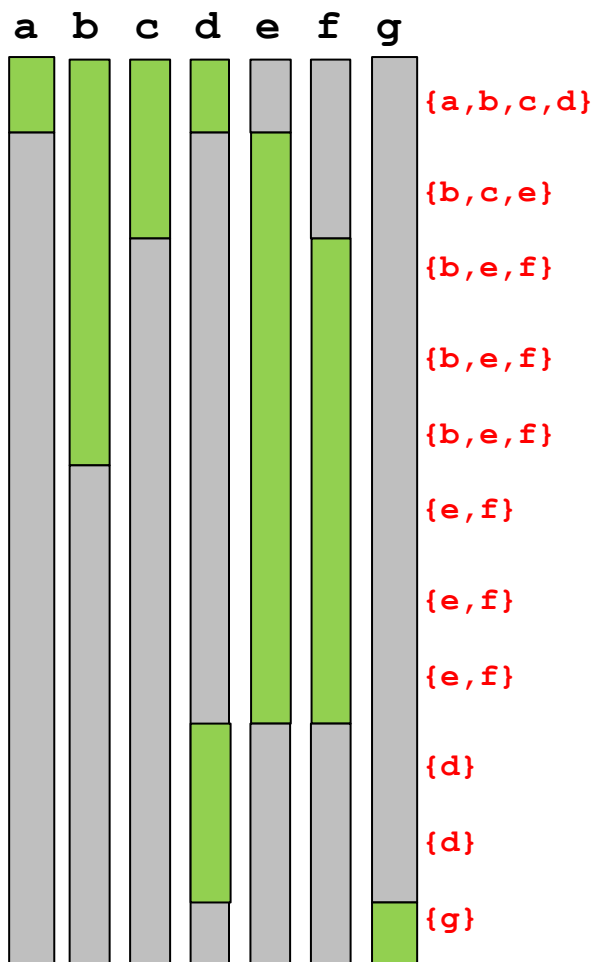
寄存器分配：线性扫描

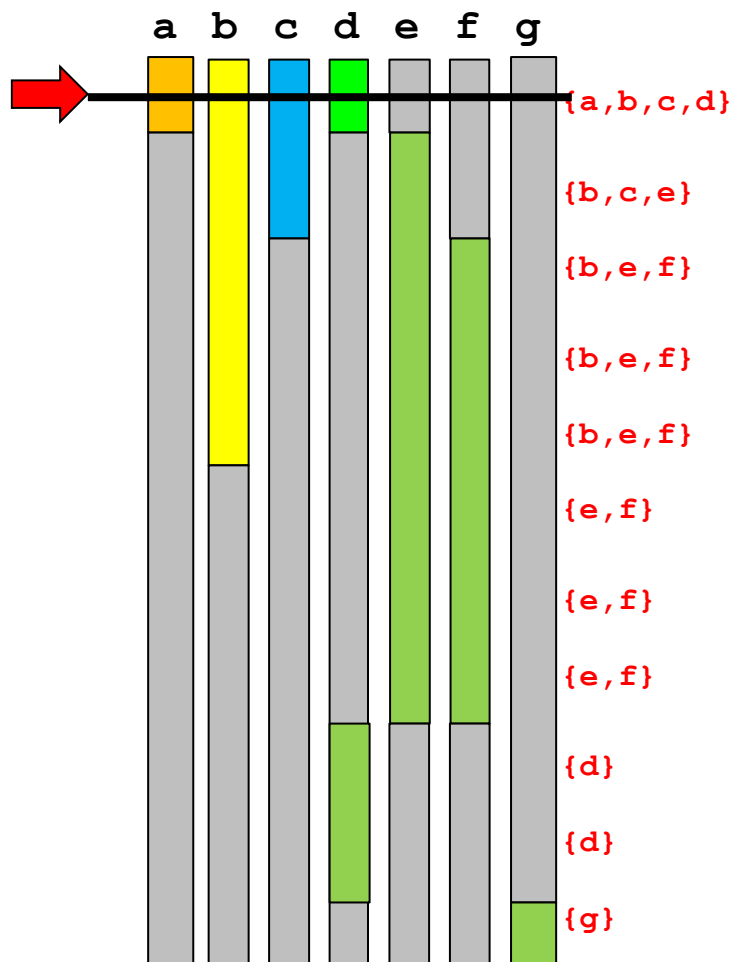
```
e = d + a
f = b + c
f = f + b
IfZ e Goto _L0
d = e + f
Goto _L1;
_L0:
d = e - f
_L1:
g = d
```



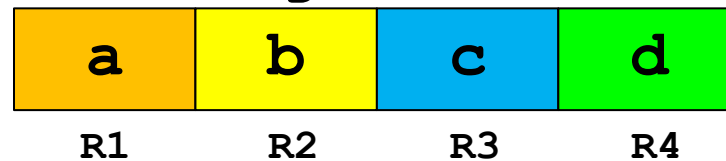


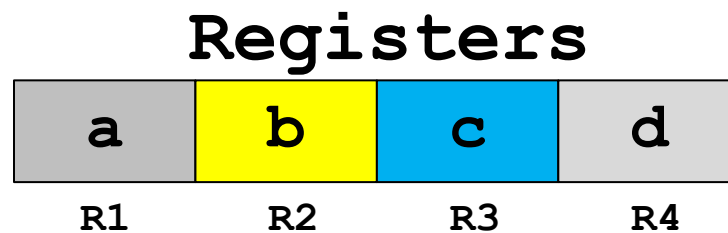
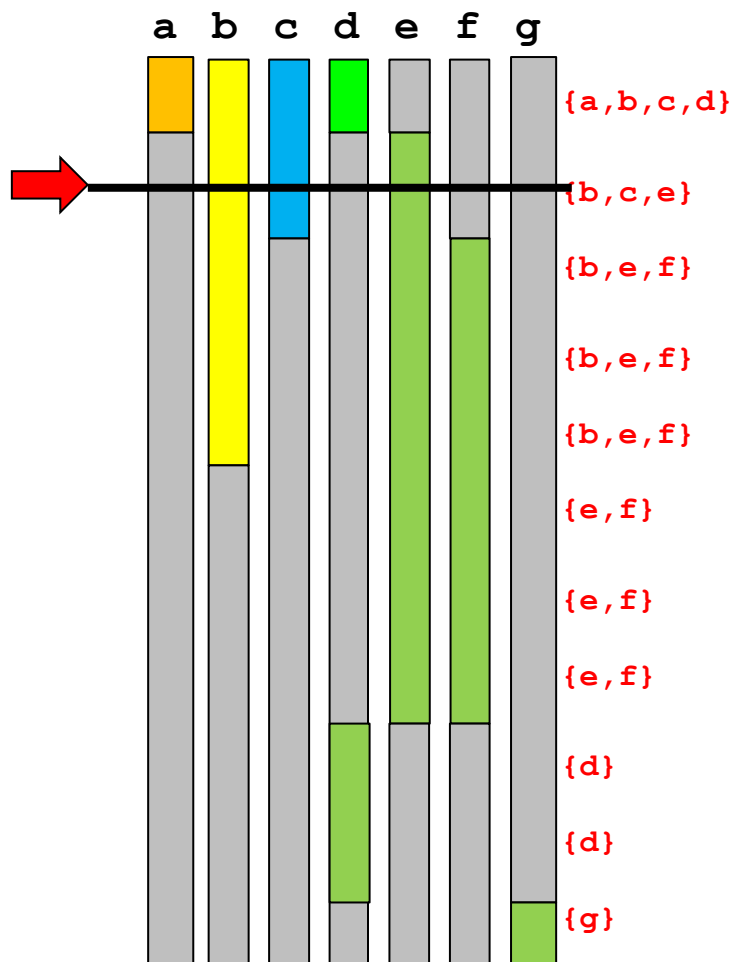
Source: Stanford CS143 (2012)

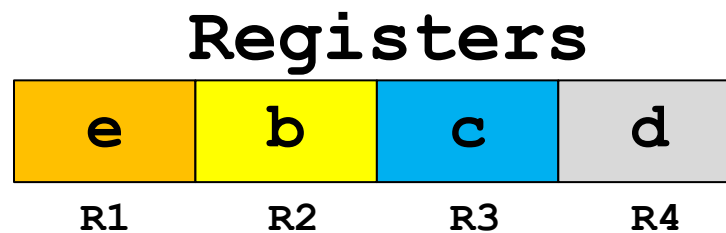
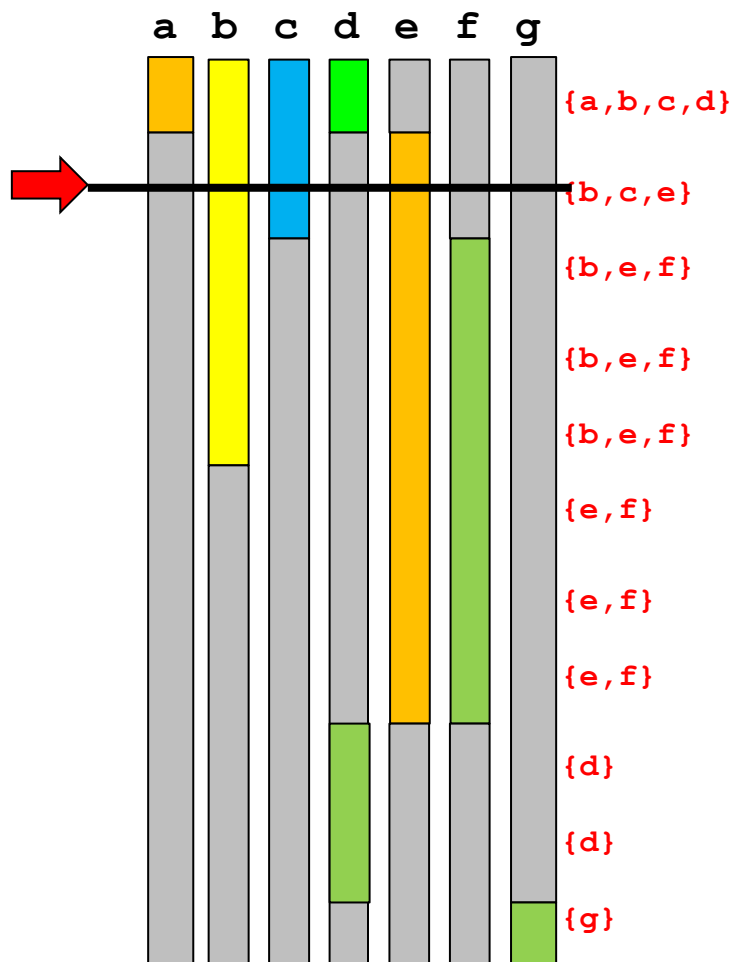


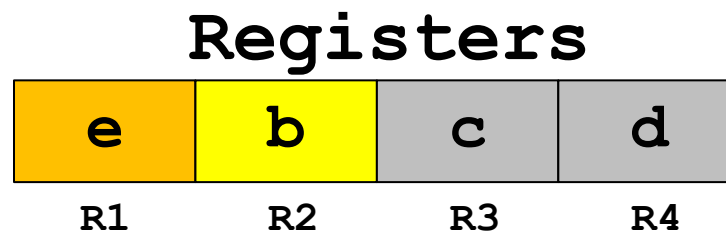
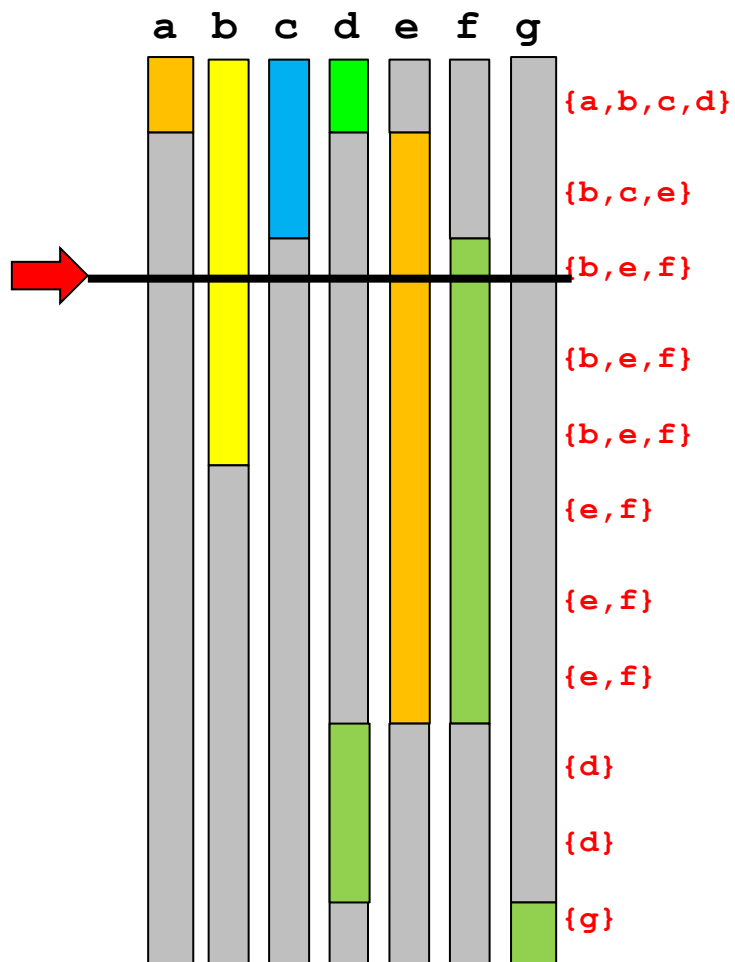


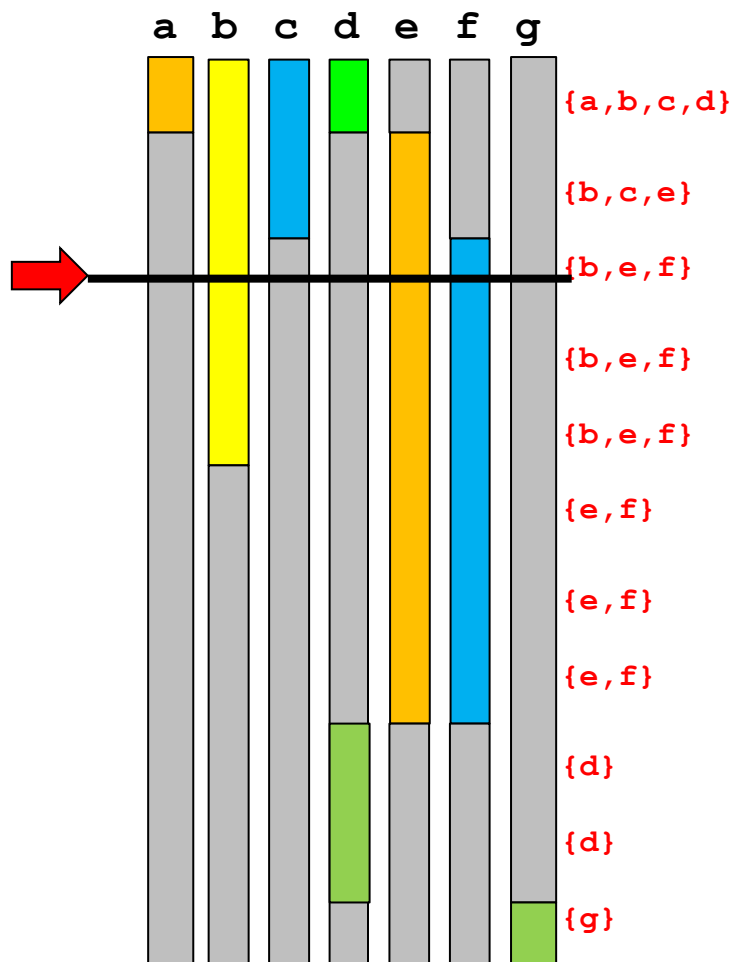
Registers





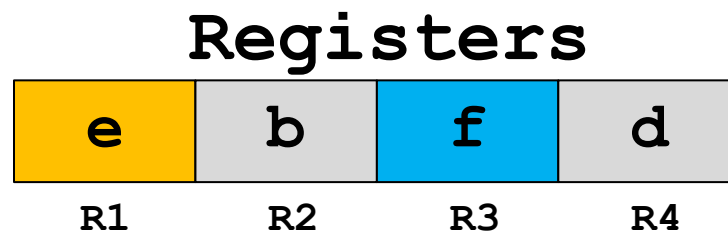
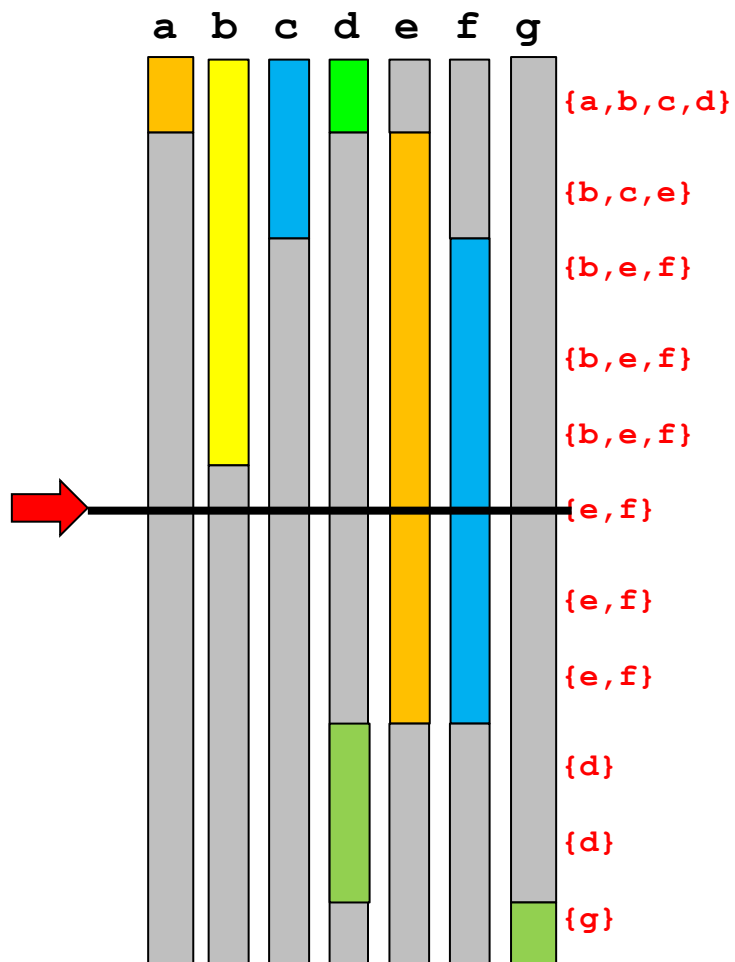


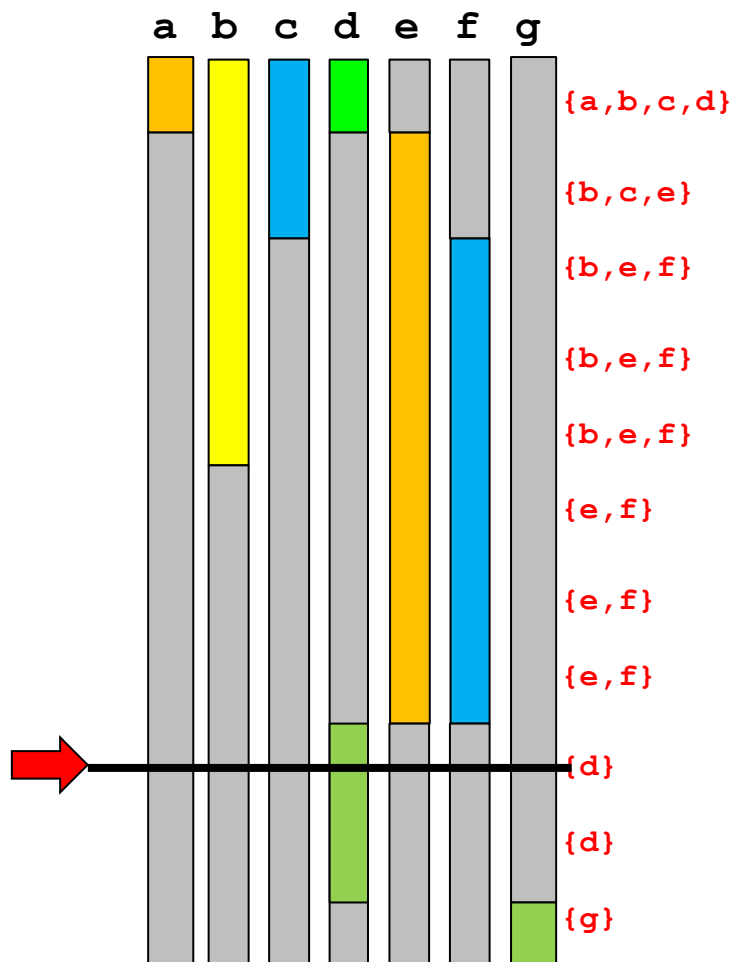




Registers

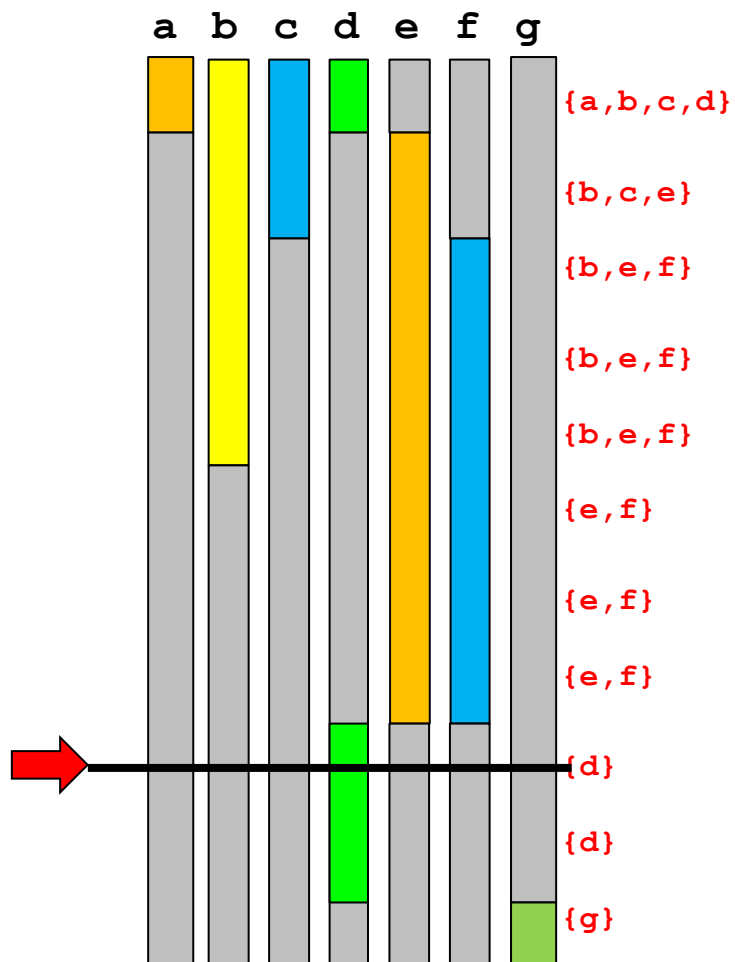
e	b	f	d
R1	R2	R3	R4





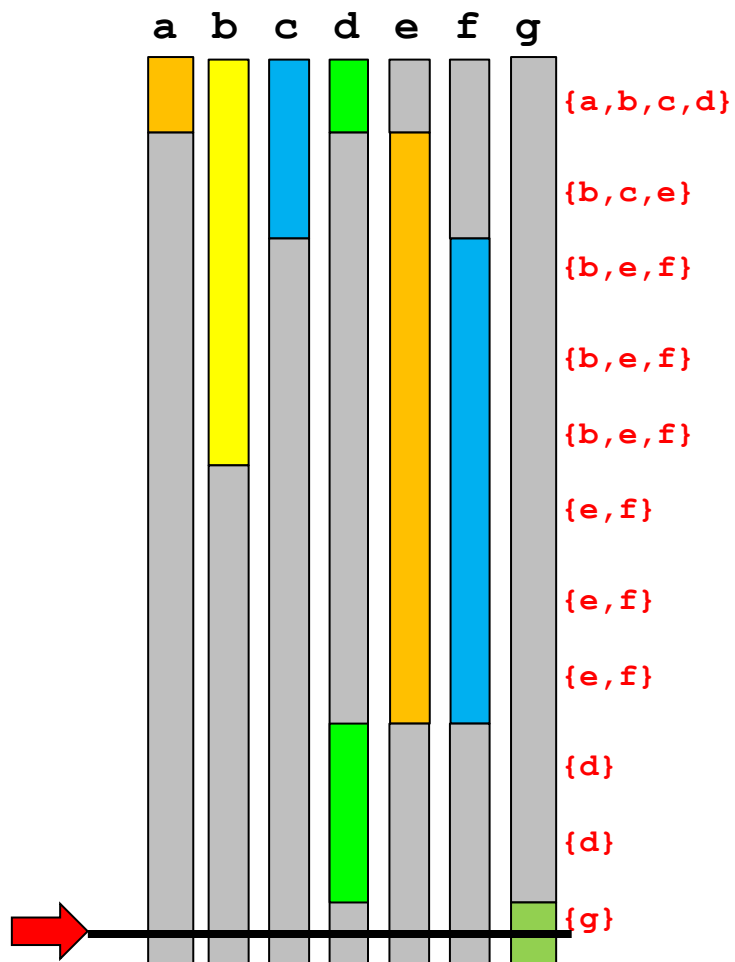
Registers

e	b	f	d
R1	R2	R3	R4



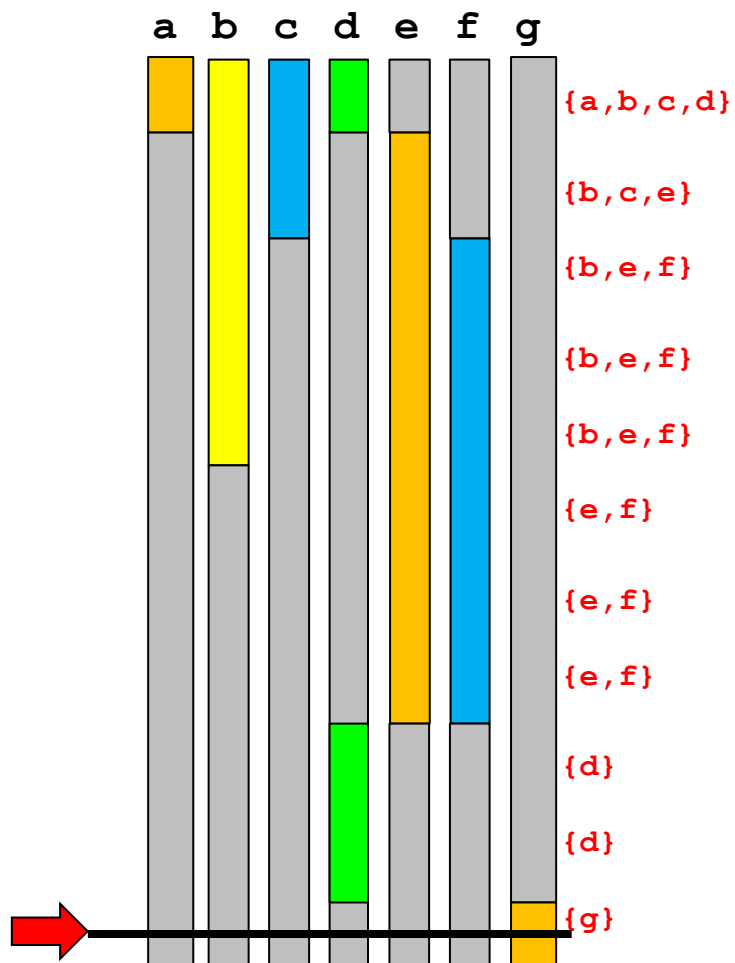
Registers

e	b	f	d
R1	R2	R3	R4



Registers

e	b	f	d
R1	R2	R3	R4

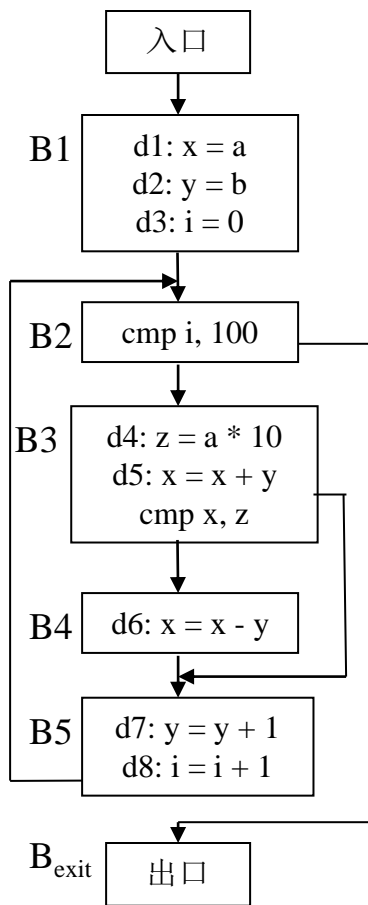


Registers

g	b	f	d
R1	R2	R3	R4

寄存器分配：图着色算法

流图



def[B]	use[B]	in[B]	out[B]	in[B]	out[B]	in[B]	out[B]
x, y, i	a, b	a, b	a,x,y,i	a, b	a,x,y,i	a, b	a,x,y,i
∅	i	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i
z	a, x, y	a,x,y,i	x, y, i	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i
∅	x, y	x, y, i	y, i	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i
∅	y, i	y, i	∅	a,x,y,i	a,x,y,i	a,x,y,i	a,x,y,i
		∅	∅	∅	∅	∅	∅

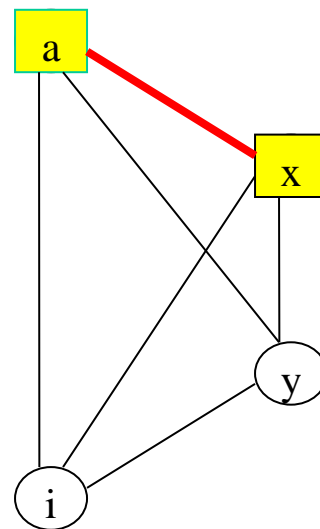
冲突图

	in[B]	out[B]
B1	a, b	a,x,y,i
B2	a,x,y,i	a,x,y,i
B3	a,x,y,i	a,x,y,i
B4	a,x,y,i	a,x,y,i
B5	a,x,y,i	a,x,y,i
B _{exit}	∅	∅

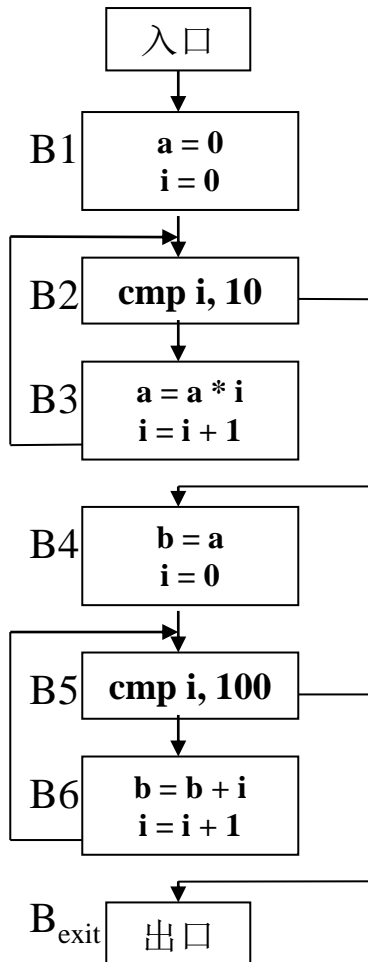
节点a: 待分配全局寄存器的变量a

节点x: 待分配全局寄存器的变量x

边a-x: 变量 a 在变量 x 定义（赋值）处是活跃的



- 变量的定义-使用链 (Define-Use链) , 变量的某一定义点, 以及所有可能使用该定义点所定义变量值的使用点所组成的一个链



变量a: L1 {<B1, 1>, <B3, 1>, <B4, 1>}
L2 {<B3, 1>, <B3, 1>, <B4, 1>}

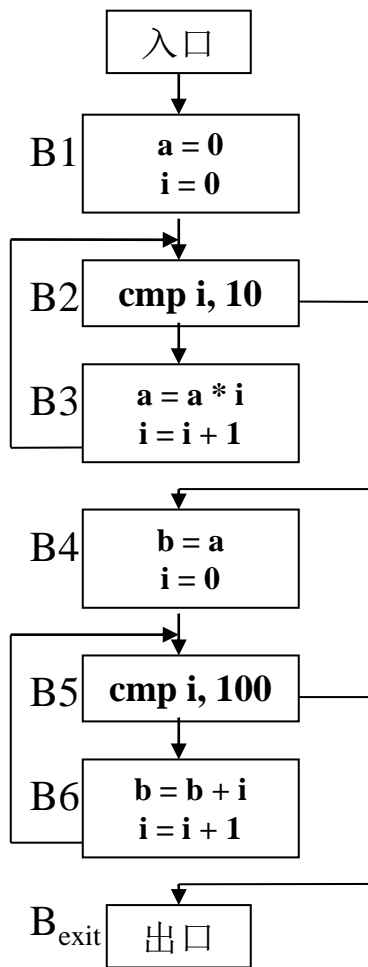
变量b: L3 {<B4, 1>, <B6, 1>}
L4 {<B6, 1>, <B6, 1>}

变量i: L5 {<B1, 2>, <B2, 1>, <B3, 1>, <B3, 2>}
L6 {<B3, 2>, <B2, 1>, <B3, 1>, <B3, 2>}
L7 {<B4, 2>, <B5, 1>, <B6, 1>, <B6, 2>}
L8 {<B6, 2>, <B5, 1>, <B6, 1>, <B6, 2>}

可以发现: L5、L6和L7、L8是没有关系的。
后面的网, 可以发现同一个变量的定义使用链分裂了,
是两个网

(a)

- 同一变量的多个定义-使用链，如果它们拥有某个同样的使用点，则合并为同一个网



变量a: L1 {<B1, 1>, <B3, 1>, <B4, 1>}

L2 {<B3, 1>, <B3, 1>, <B4, 1>}

变量b: L3 {<B4, 1>, <B6, 1>}

L4 {<B6, 1>, <B6, 1>}

变量i: L5 {<B1, 2>, <B2, 1>, <B3, 1>, <B3, 2>}

L6 {<B3, 2>, <B2, 1>, <B3, 1>, <B3, 2>}

L7 {<B4, 2>, <B5, 1>, <B6, 1>, <B6, 2>}

L8 {<B6, 2>, <B5, 1>, <B6, 1>, <B6, 2>}

变量a: W1 { L1 {<B1, 1>, <B3, 1>, <B4, 1>}, L2 {<B3, 1>, <B3, 1>, <B4, 1>}}

变量b: W2 { L3 {<B4, 1>, <B6, 1>}, L4 {<B6, 1>, <B6, 1>}}

变量i: W3 { L5 {<B1, 2>, <B2, 1>, <B3, 1>, <B3, 2>}, L6 {<B3, 2>, <B2, 1>, <B3, 1>, <B3, 2>}}

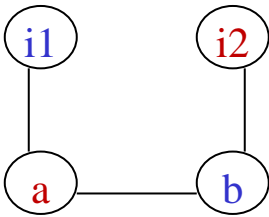
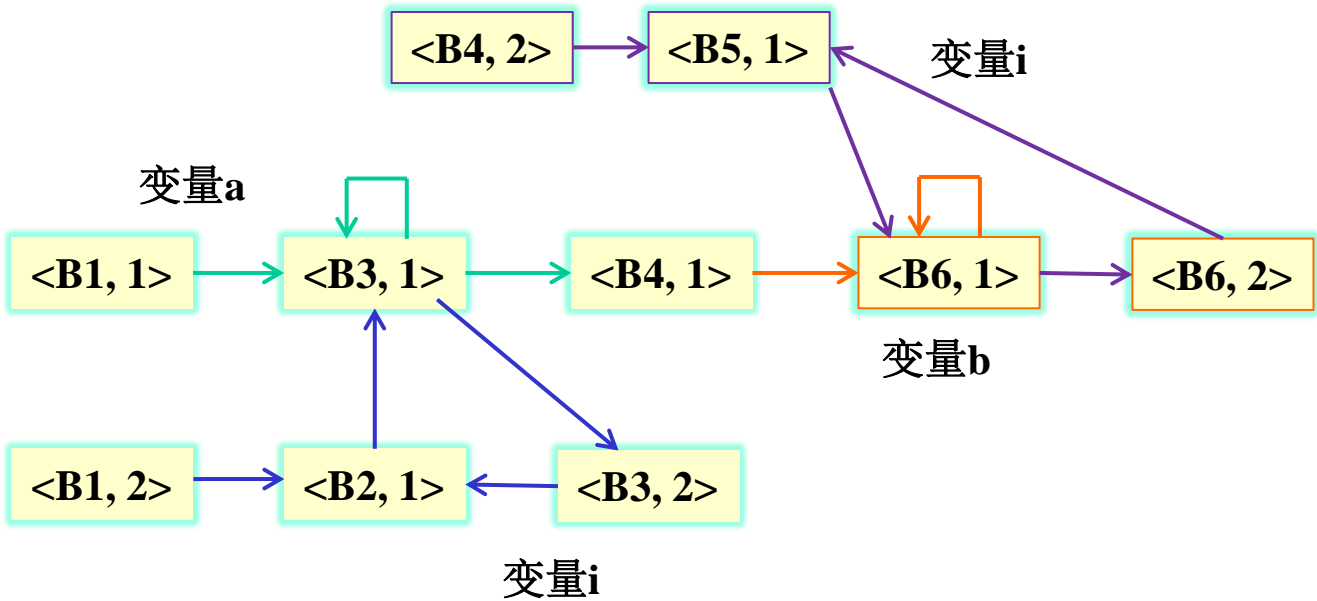
W4 { L7 {<B4, 2>, <B5, 1>, <B6, 1>, <B6, 2>}, L8 {<B6, 2>, <B5, 1>, <B6, 1>, <B6, 2>}}

变量a: W1 { L1 {<B1, 1>, <B3, 1>, <B4, 1>}, L2 {<B3, 1>, <B3, 1>, <B4, 1>}}

变量b: W2 { L3 {<B4, 1>, <B6, 1>}, L4 {<B6, 1>, <B6, 1>}}

变量i: W3 { L5 {<B1, 2>, <B2, 1>, <B3, 1>, <B3, 2>}, L6 {<B3, 2>, <B2, 1>, <B3, 1>, <B3, 2>}}

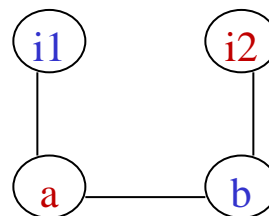
W4 { L7 {<B4, 2>, <B5, 1>, <B6, 1>, <B6, 2>}, L8 {<B6, 2>, <B5, 1>, <B6, 1>, <B6, 2>}}



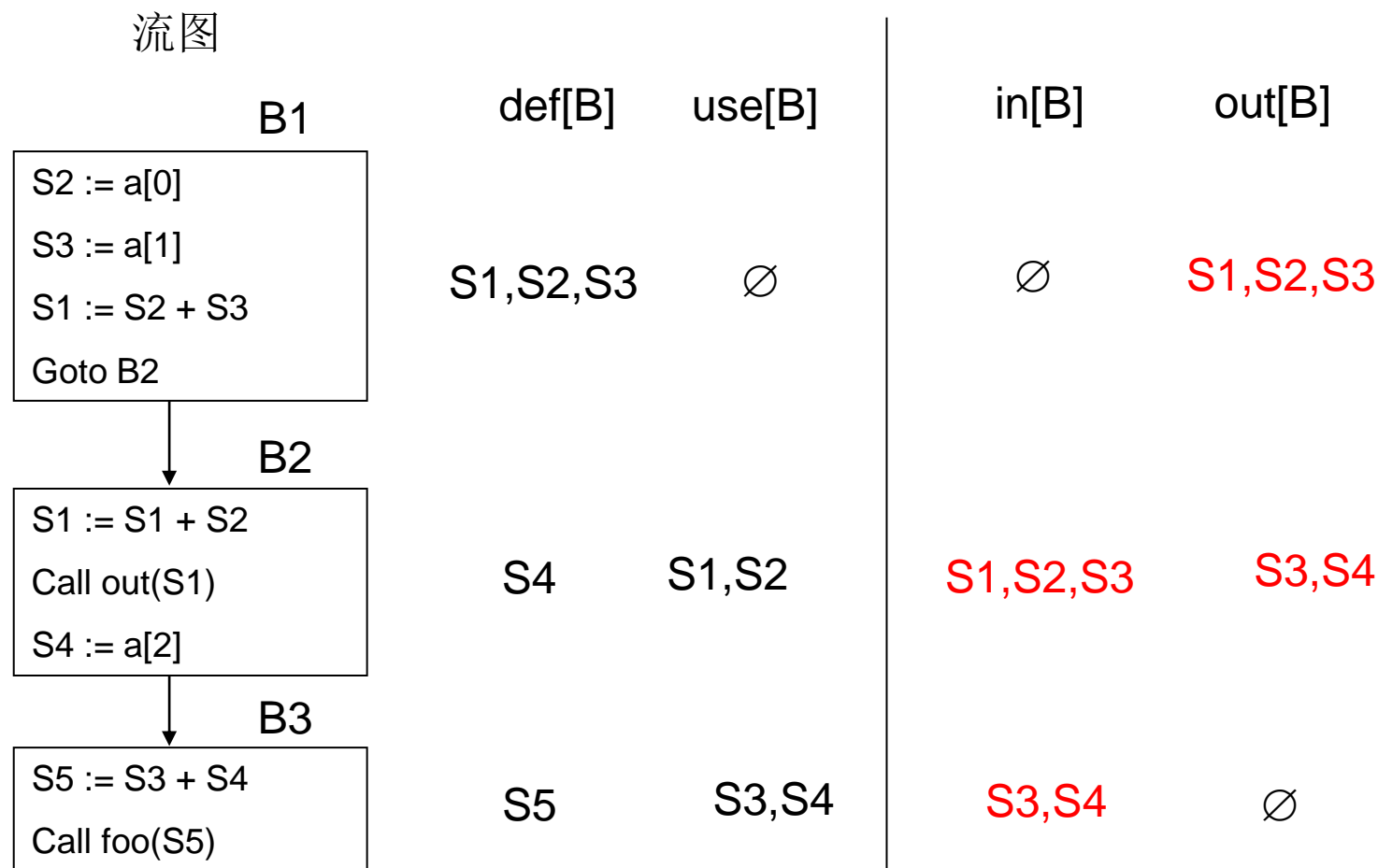
图着色算法

- 算法目的:

- 给定冲突图，给出寄存器分配方案
- 基本思想：如果可供分配 k 个全局寄存器，就尝试用 k 种颜色给冲突图着色
- （原则：两个冲突变量不能着相同颜色）



• 首先构造冲突图：



• 首先构造冲突图：

基本块入口处
的活跃变量

流图

冲突图

B1

```
S2 := a[0]
S3 := a[1]
S1 := S2 + S3
Goto B2
```

B2

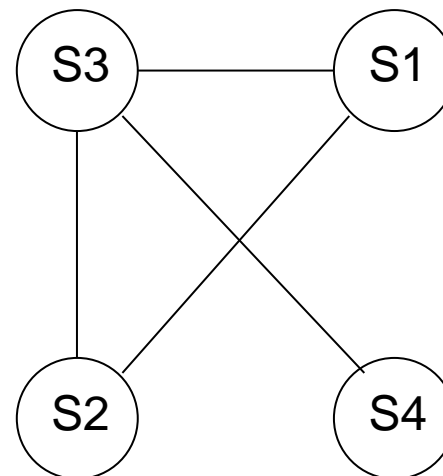
```
S1 := S1 + S2
Call out(S1)
S4 := a[2]
```

B3

```
S5 := S3 + S4
Call foo(S5)
```

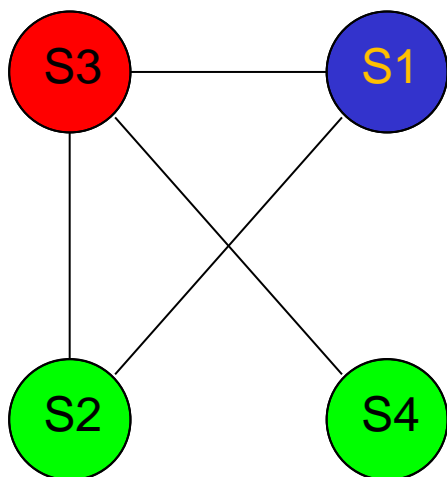
S1, S2, S3

S3, S4

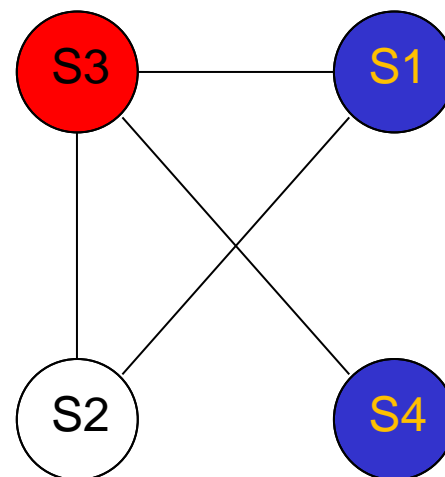


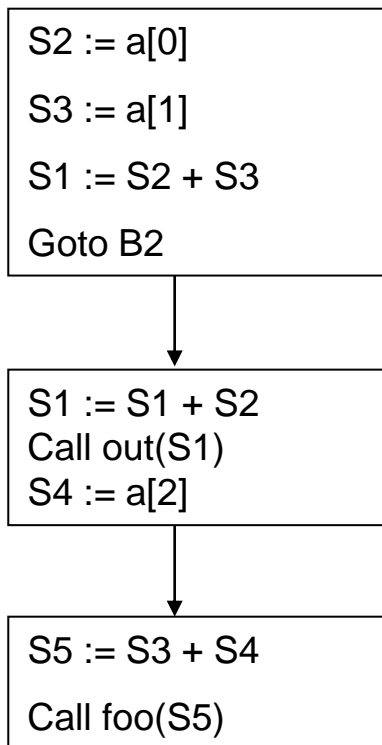
- 步骤2：尝试用k种颜色给该冲突图着色

假设1： $k=3$, R_0, R_1, R_2

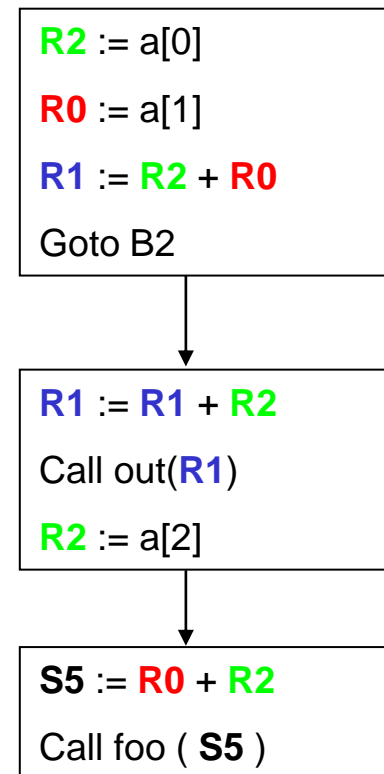
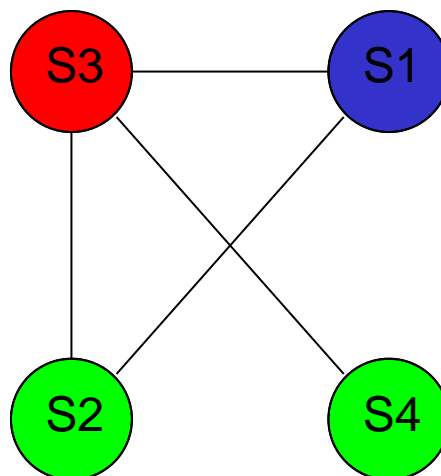


假设2： $k=2$, R_0, R_1





假设1: $k=3$, $R0, R1, R2$

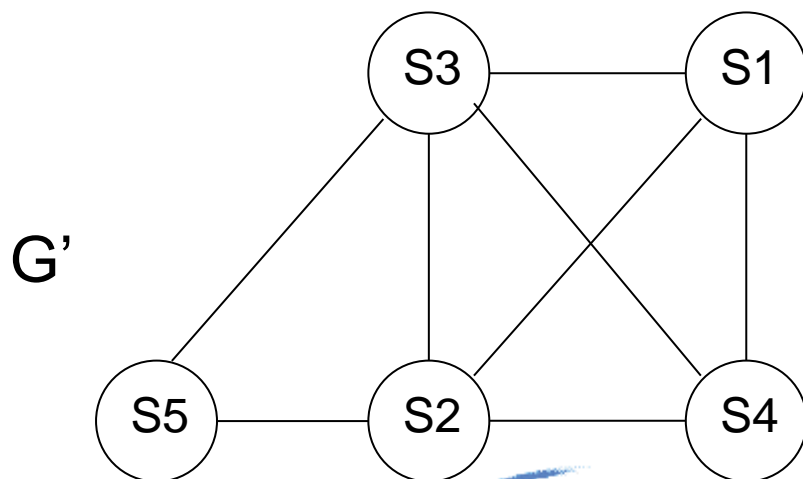
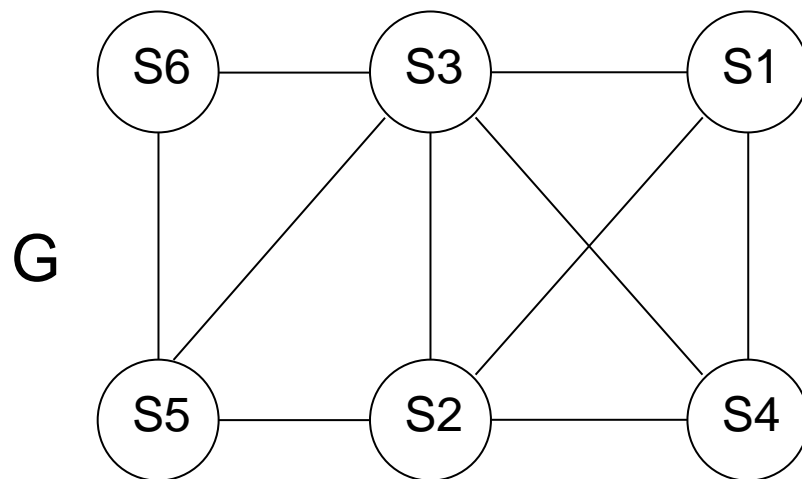
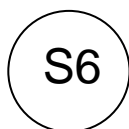


一种启发式图着色算法：Chaitin-Briggs算法

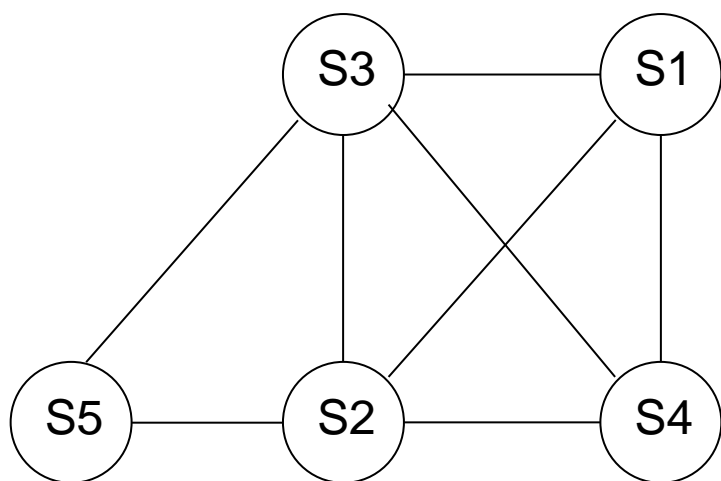
- 冲突图G
 - 寄存器数目为K
 - 假设 $K=3$

■ **步骤1**、找到第一个连接边数目小于K的结点，将它从图G中移走，形成图G'

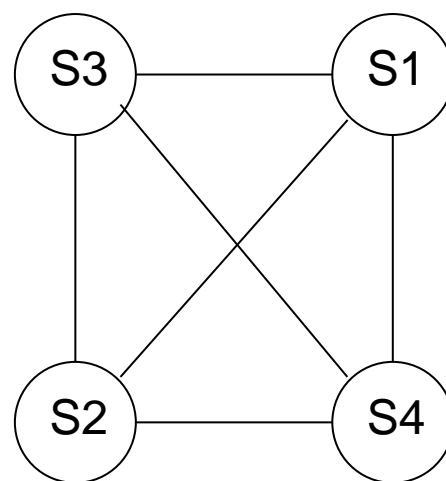
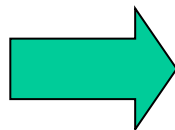
已移走节点



■ 步骤2、重复步骤1，直到无法再从 G' 中移走结点

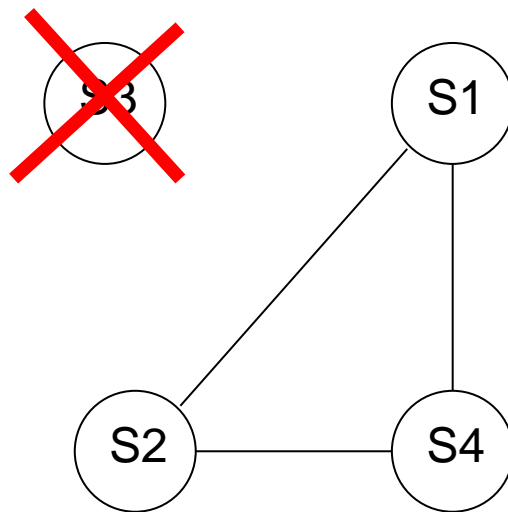
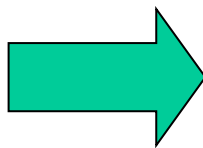
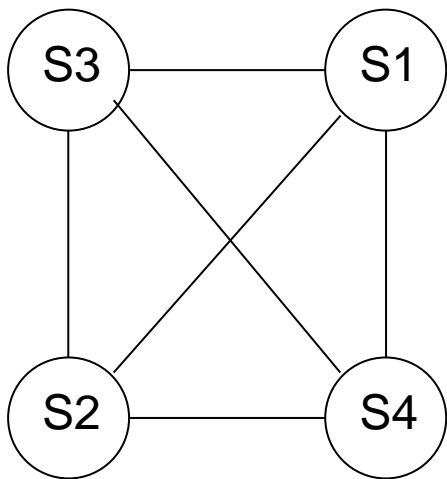



已移走节点 






已移走节点  

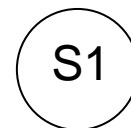
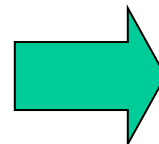
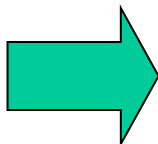
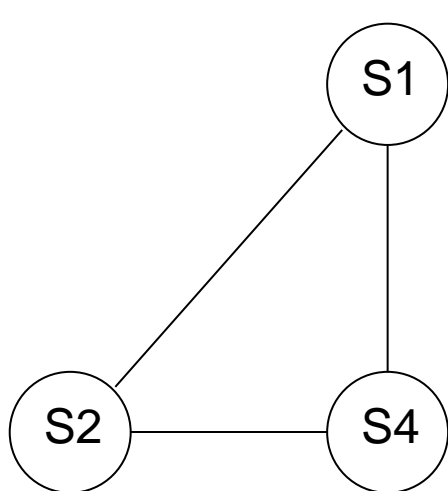
- **步骤3**、在图中选取**适当**的结点，将它记录为“不分配全局寄存器”的结点，并从图中移走



已移走节点  

已移走节点   

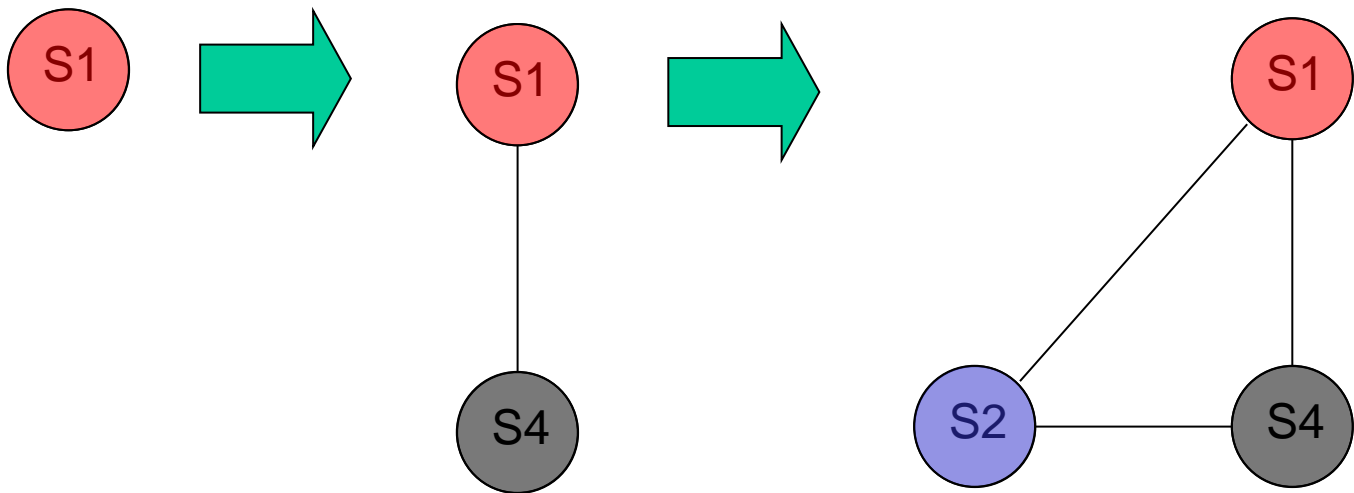
■ 步骤4、重复上述步骤，直到图中仅剩余1个结点



已移走 S6 S5 ~~S3~~

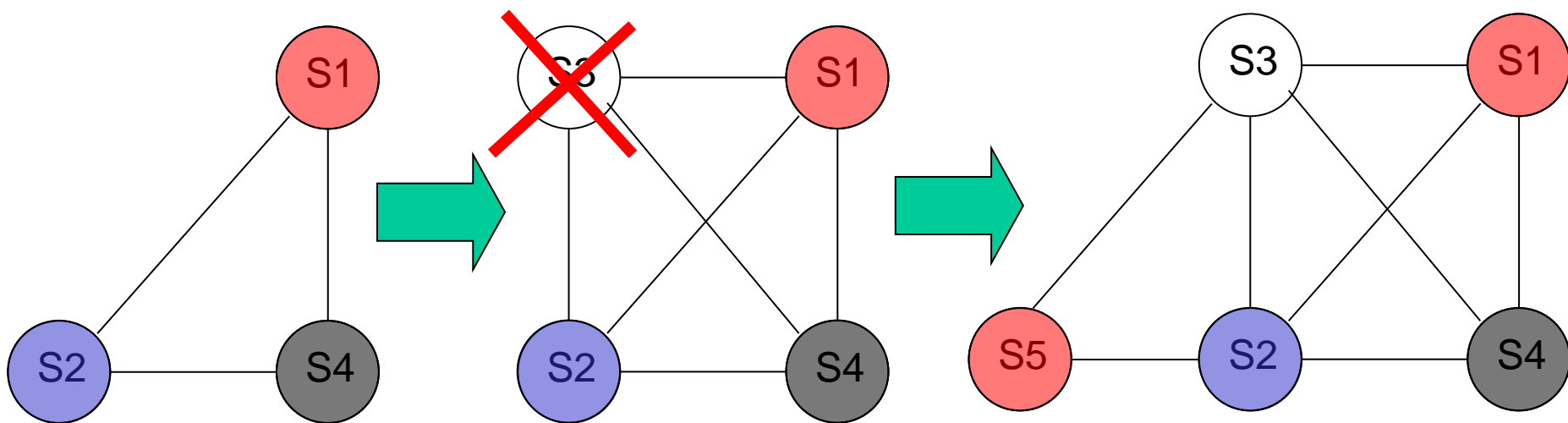
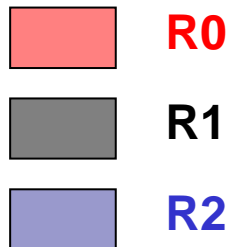
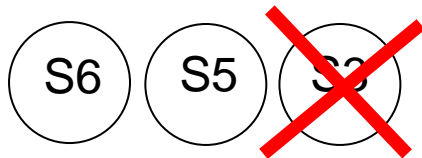
已移走 S6 S5 ~~S3~~ S2 S4

■ 步骤5 按照结点移走的反向顺序将点和边添加回去，并分配颜色



■ 步骤5 按照结点移走的反向顺序将点和边添加回去，并分配颜色

移走顺序



■ 步骤5 按照结点移走的反向顺序将点和边添加回去，并分配颜色

移走顺序

S6

R0

R1

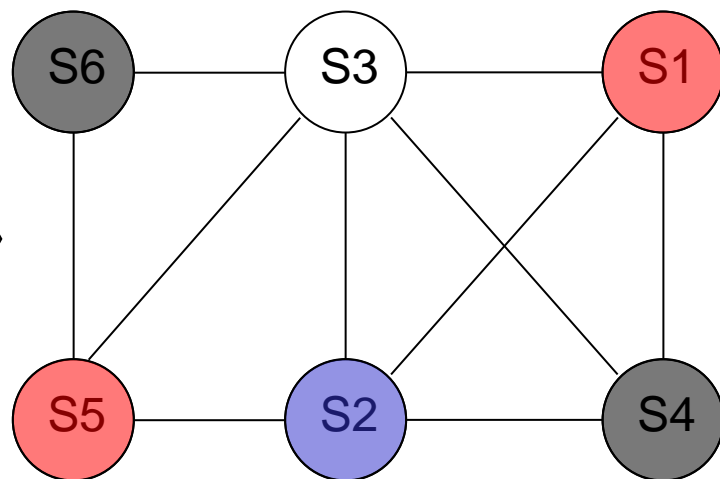
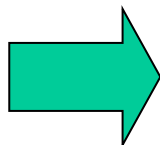
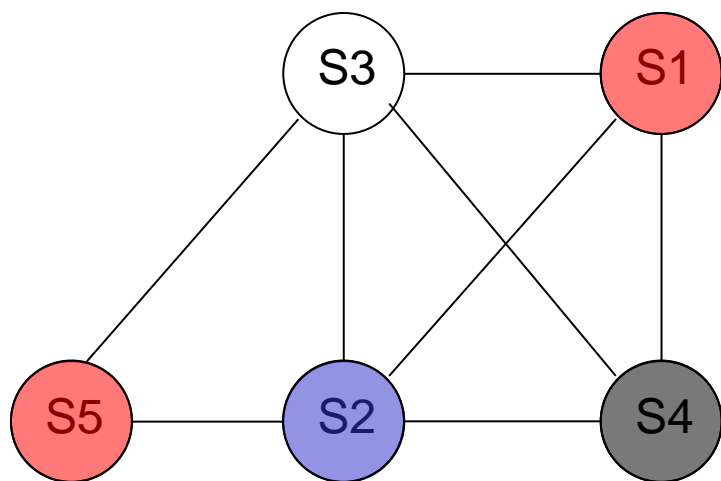
R2

R0: S1, S5

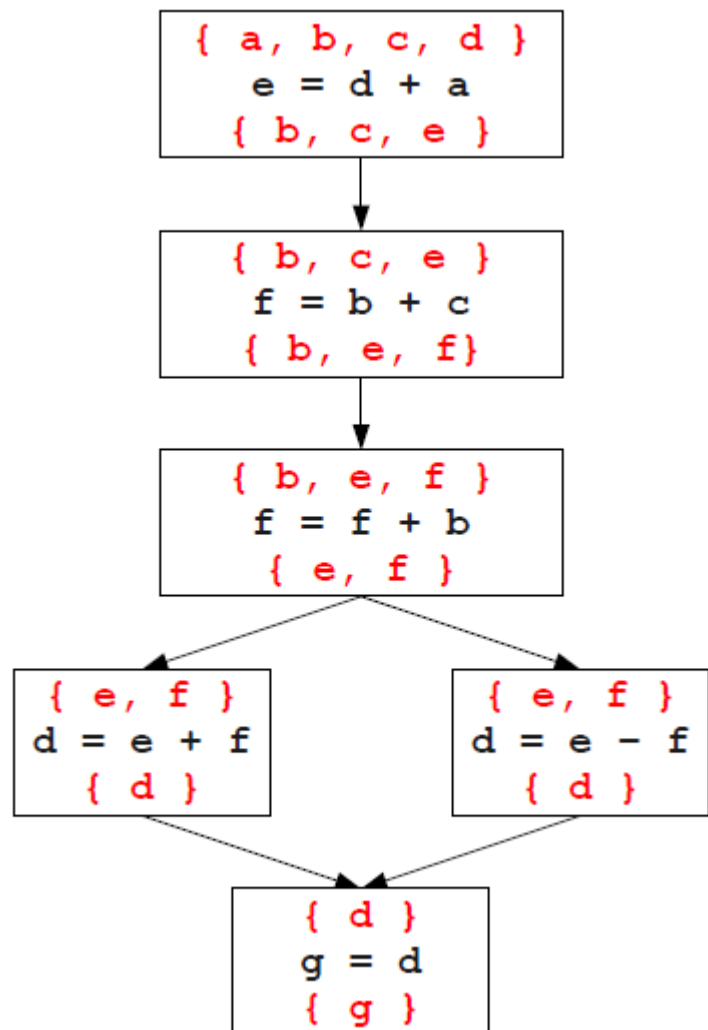
R1: S4, S6

R2: S2

变量 S3 不分配寄存器



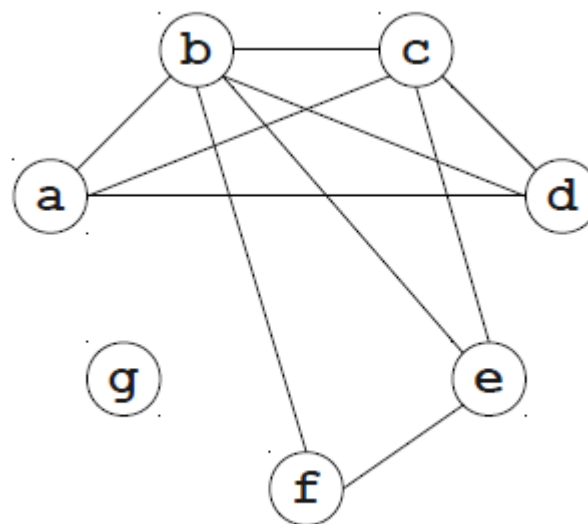
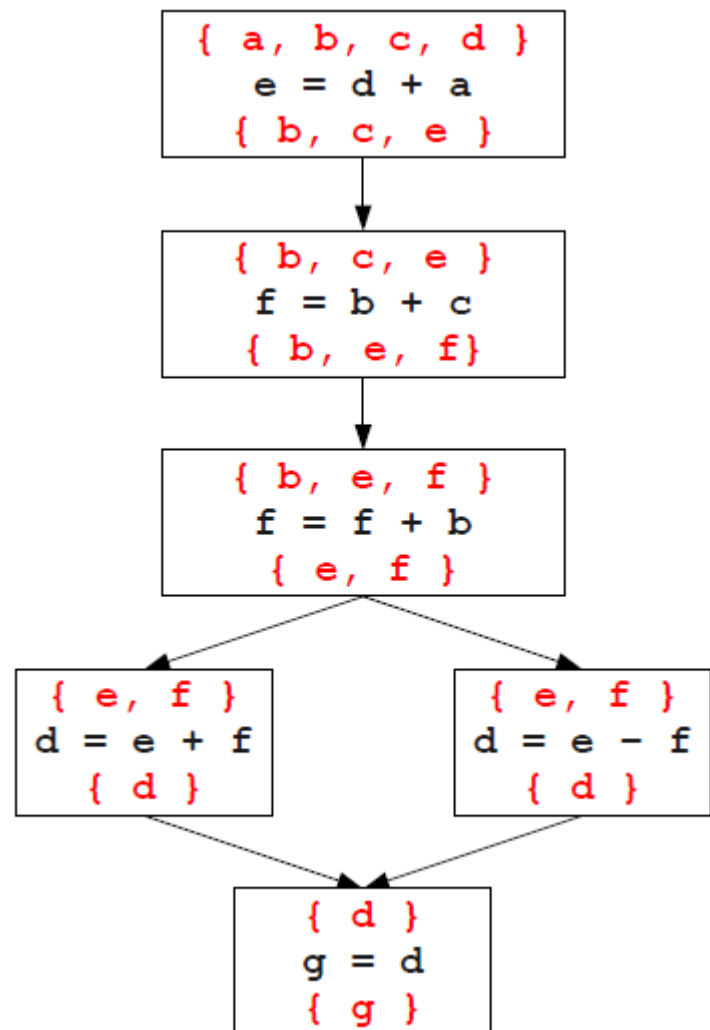
■ 一个完整的例子：



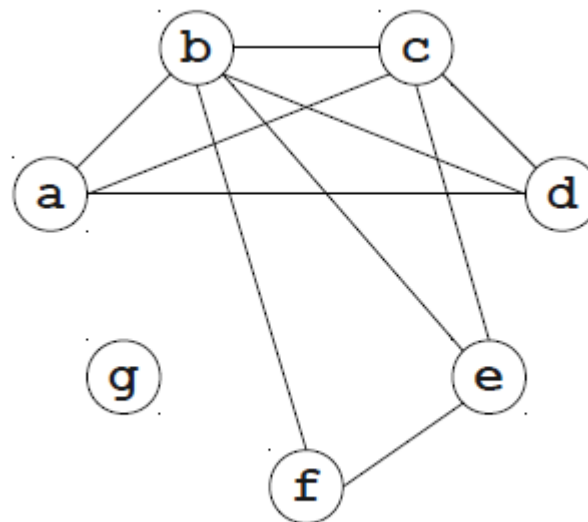
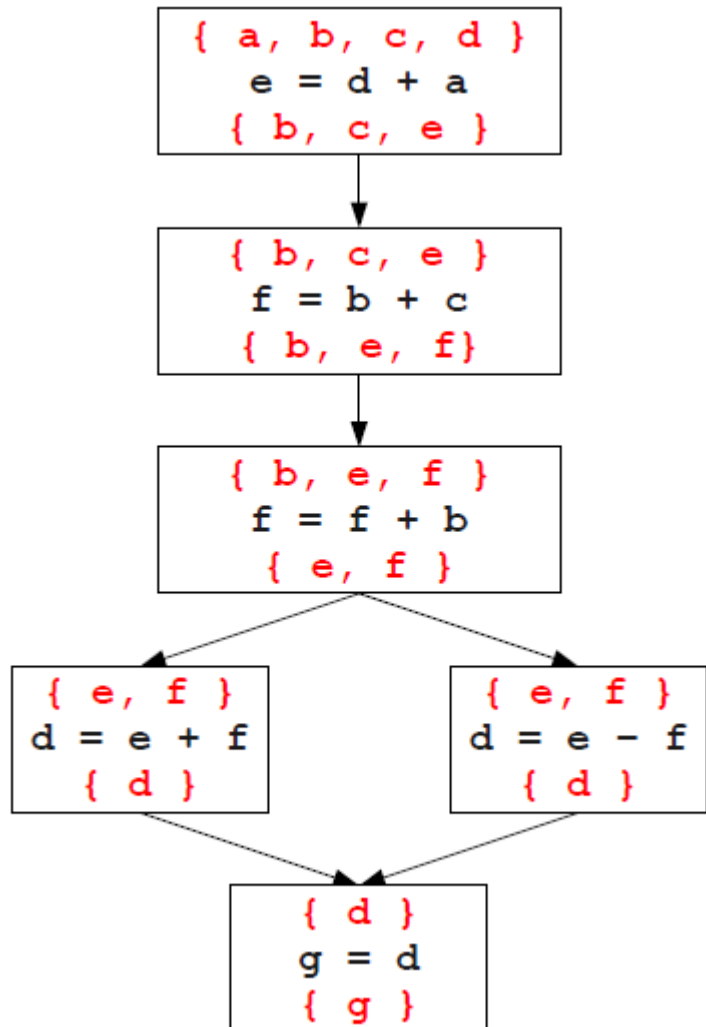
Registers



■ 一个完整的例子：

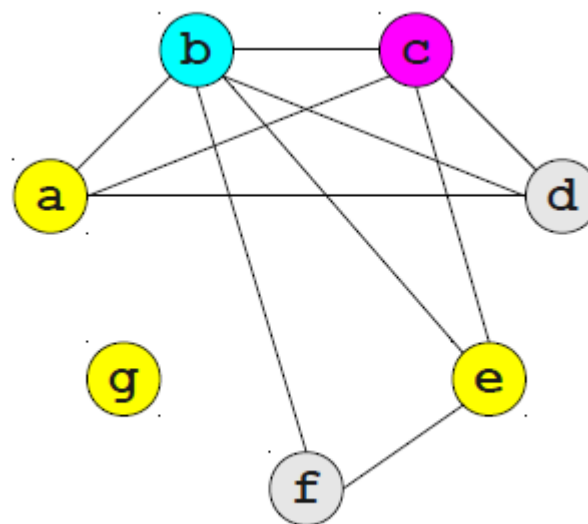
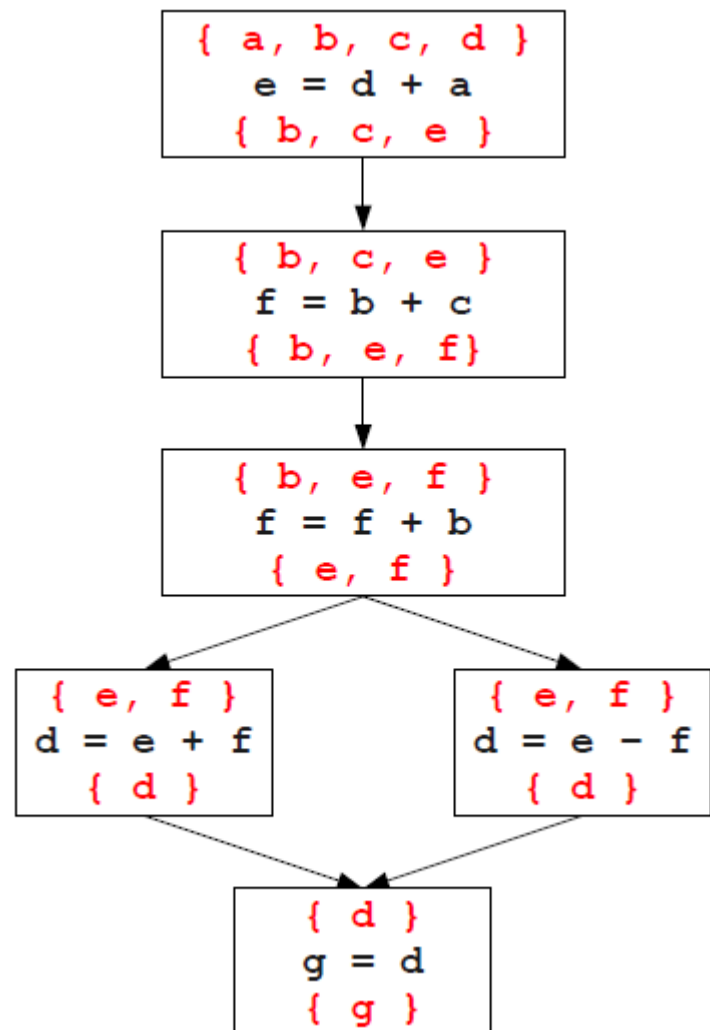


■ 一个完整的例子：

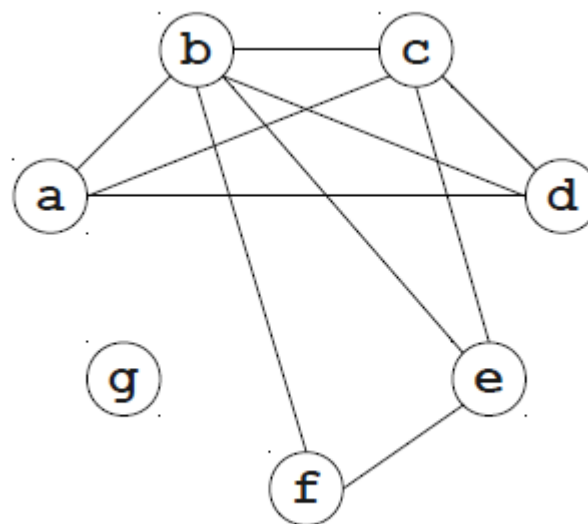
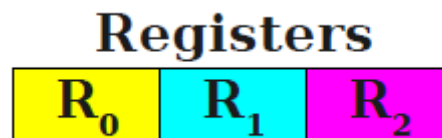
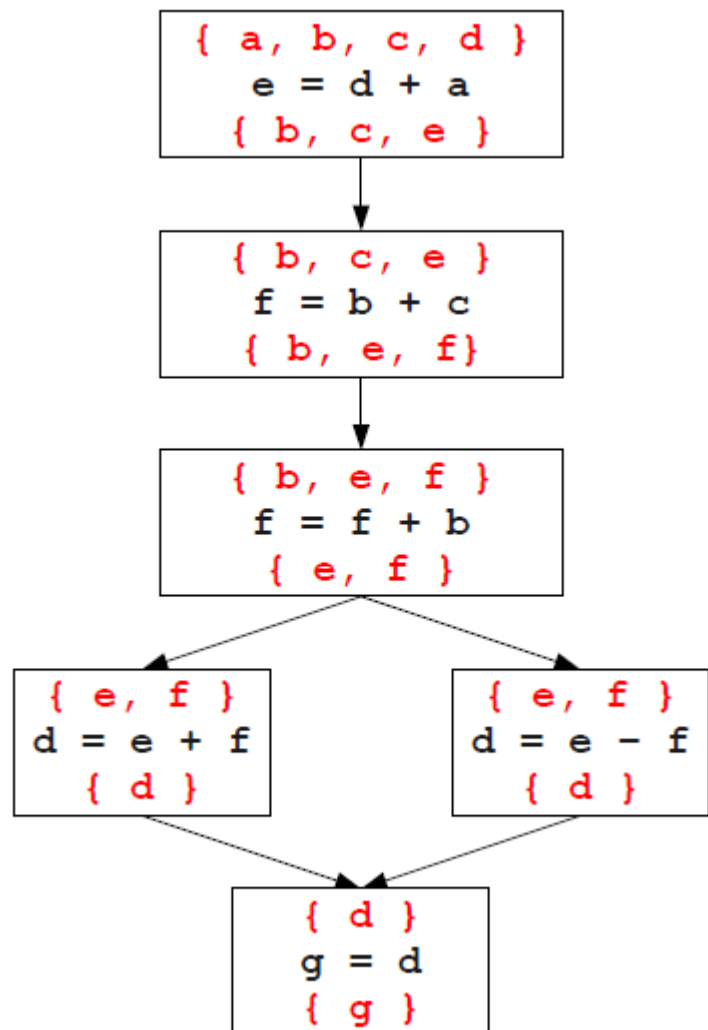


节点移除： g, f, e, d, c, b, a

■ 一个完整的例子：

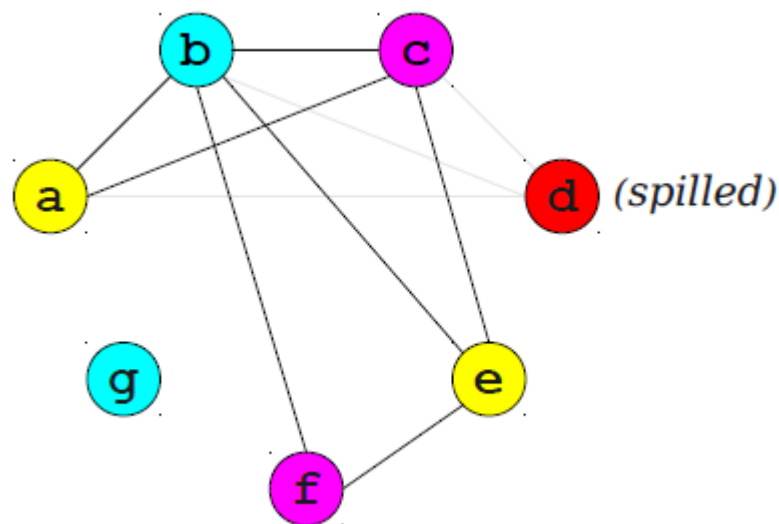
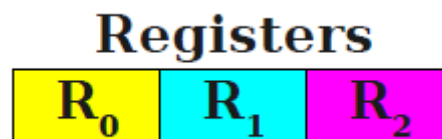
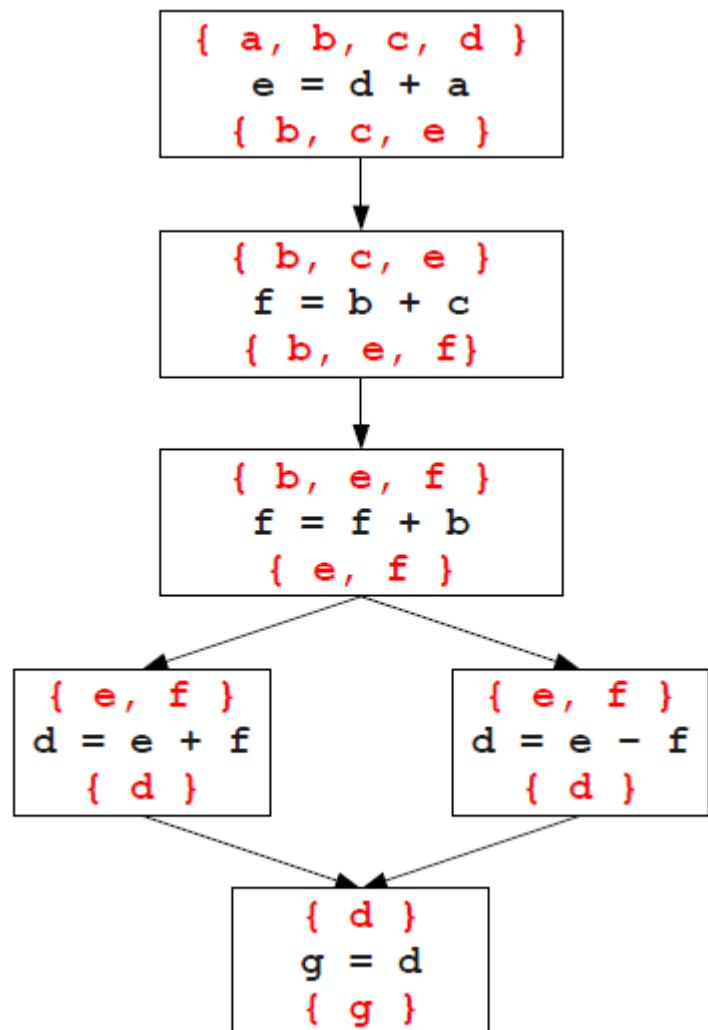


■ 一个完整的例子：

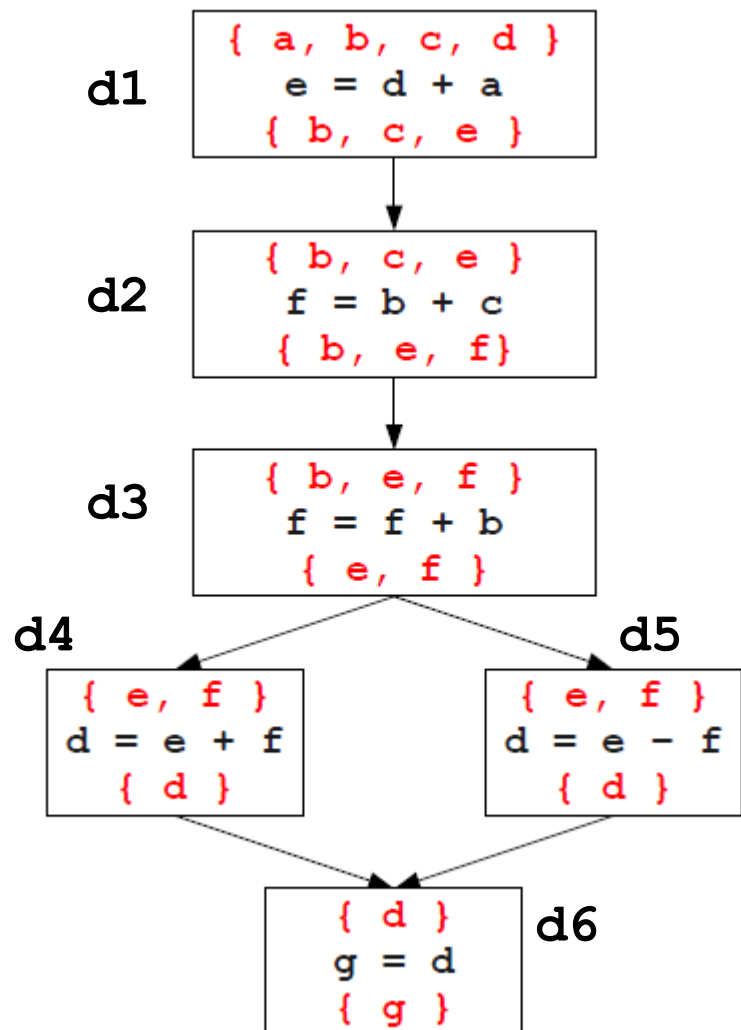


节点移除： g, f, e, d, c, b, a

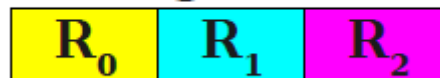
■ 一个完整的例子：



■ 一个完整的例子：



Registers



变量 d 的 define-use 链：

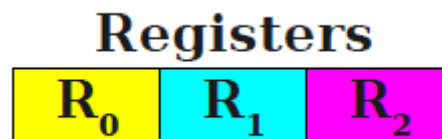
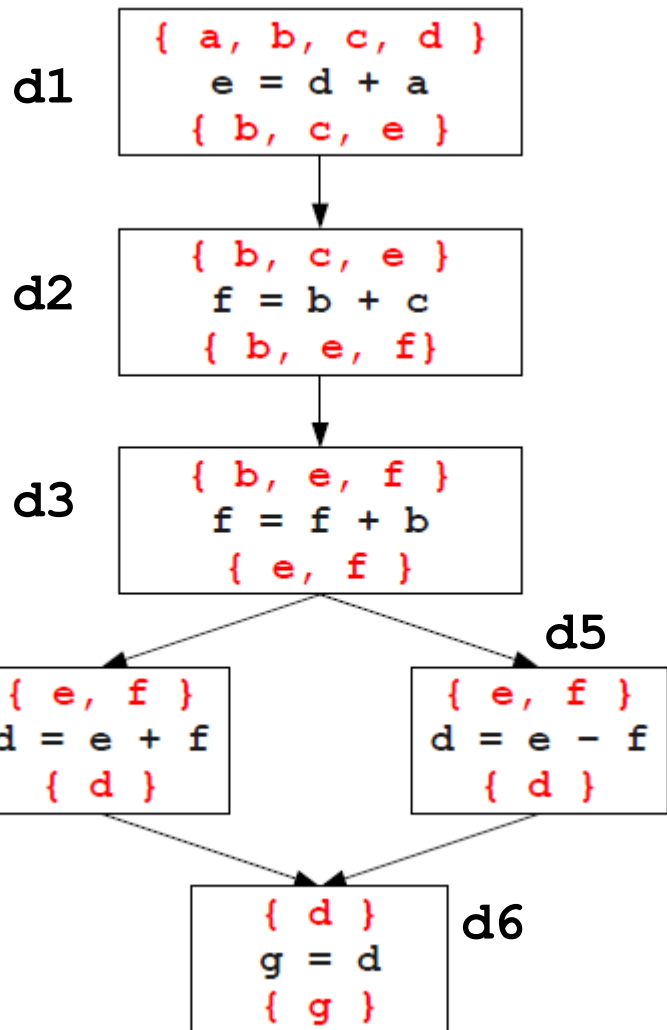
{?, $d1$ }

{ $d4$, $d6$ }

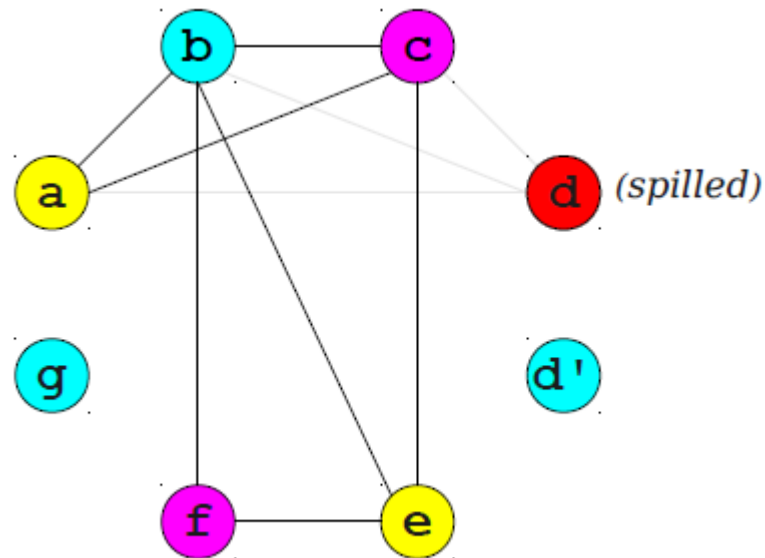
{ $d5$, $d6$ }

可以将 d 改写为 d 和 d'

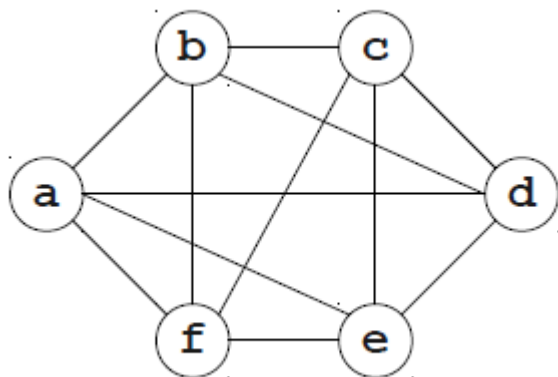
■ 一个完整的例子：



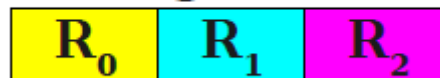
变量 d 改写为 d 和 d'



- 给定一个顺序，染色的过程可能产生不同的结果

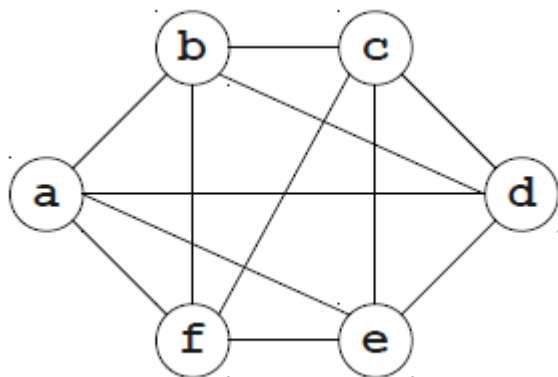


Registers

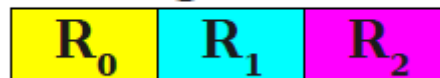


顺序: {**a**, **c**, b, e, f, d}

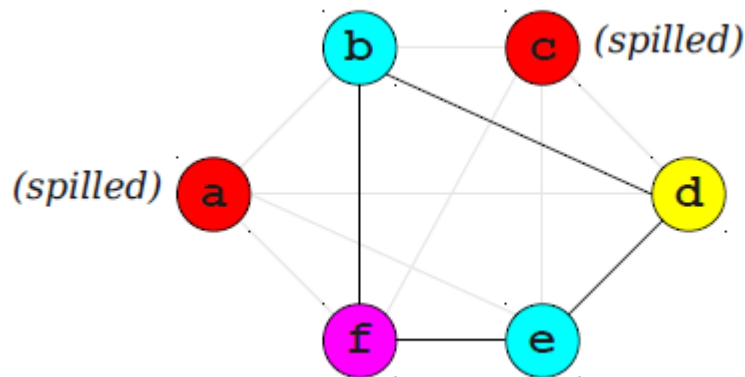
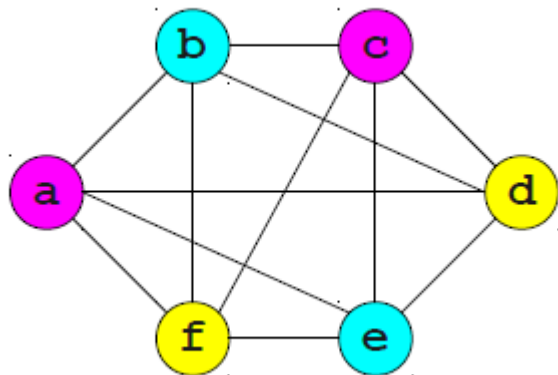
给定一个顺序，染色的过程可能产生不同的结果



Registers



顺序: {**a**, **c**, b, e, f, d}



寄存器分配：临时寄存器分配

临时寄存器分配

- 为什么在代码生成过程中，需要对临时寄存器进行管理？
 - 因为生成某些指令时，必须使用指定寄存器
 - 临时寄存器中保存有此前的计算中间结果
- 以X86为例，生成代码时可用的临时寄存器
 - EAX, ECX, EDX等

临时寄存器分配

- 临时寄存器的生存范围
 - 不超越基本块
 - 不跨越函数调用
- 临时寄存器的管理方法
 - 寄存器池

全局寄存器分配结果：

a	EBX
b	ESI
c	EDI

临时变量在运行栈上的保存地址：

t3	ESP+10H
t2	ESP+0CH
t1	ESP+08H

寄存器池：

t1 := - c

t1	EAX
	EDX

mov EAX, EDI

neg EAX

t2 := t1 - b

t1	EAX
t2	EDX

mov EDX, EAX

sub EDX, ESI

t3 := t2 + t2

t3	EAX
t2	EDX

mov [ESP+08H], EAX

mov EAX, EDX

add EAX, EAX

a := t3

t3	EAX
t2	EDX

mov EBX, EAX

例:

- (1) $t1 = -c$
- (2) $t2 = t1 - b$
- (3) $t3 = t2 + t2$
- (4) $a = t3$

a
b
c

EBX
ESI
EDI

t3
t2
t1

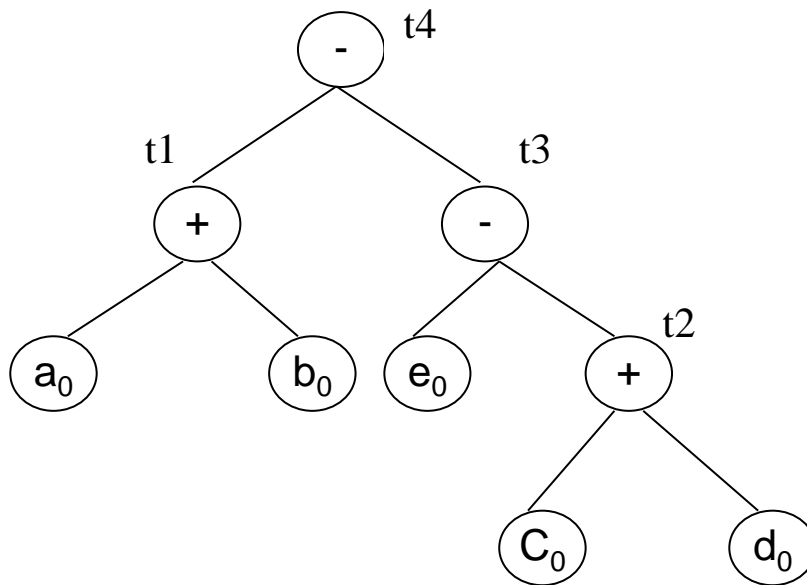
ESP+10H
ESP+0CH
ESP+08H

逐句转换:

- (1) `mov ECX, EDI`
`neg ECX`
`mov [ESP+08H], ECX`
- (2) `mov ECX, [ESP+08H]`
`sub ECX, ESI`
`mov [ESP+0CH], ECX`
- (3) `mov ECX, [ESP+0CH]`
`add ECX, [ESP+0CH]`
`mov [ESP+10H], ECX`
- (4) `mov EBX, [ESP+10H]`

逐句转换+临时寄存器池:

- (1) `mov EAX, EDI`
`neg EAX`
- (2) `mov EDX, EAX`
`sub EDX, ESI`
- (3) `mov [ESP+08H], EAX`
`mov EAX, EDX`
`add EAX, EAX`
- (4) `mov EBX, EAX`



从DAG图重新导出中间代码

从“寄存器池”角度看：

- 局部变量a, b, c, d, e 均不占用寄存器（使用内存）
- 仅有两个寄存器 **eax**, **edx** 可供t1,t2,t3,t4使用

(1) $t1 = a + b$
 $t2 = c + d$
 $t3 = e - t2$
 $t4 = t1 - t3$

(2) $t2 = c + d$
 $t3 = e - t2$
 $t1 = a + b$
 $t4 = t1 - t3$

```
(1) t1 = a + b
    t2 = c + d
    t3 = e - t2
    t4 = t1 - t3
```

```
mov eax, a      ; t1 = a + b
add eax, b
mov edx, c      ; t2 = c + d
add edx, d
mov [ESP+08H], eax ; t3 = e - t2
mov eax, e
sub eax, edx
mov [ESP+0CH], edx ; t4 = t1 - t3
mov edx, [ESP+08H]
sub edx, eax
```

从“寄存器池”角度看：

- 局部变量a, b, c, d, e 均不占用寄存器（使用内存）
- 仅有两个寄存器 **eax, edx** 可供t1,t2,t3,t4使用
- **[ESP+08H], [ESP+0CH] 均为临时变量在运行栈上的临时保存单元地址**

(1) $t1 = a + b$

$t2 = c + d$

$t3 = e - t2$

$t4 = t1 - t3$

mov eax, a ; $t1 = a + b$

add eax, b

mov edx, c ; $t2 = c + d$

add edx, d

mov [ESP+08H], eax

; $t3 = e - t2$

mov eax, e

sub eax, edx

mov [ESP+0CH], edx

; $t4 = t1 - t3$

mov edx, [ESP+08H]

sub edx, eax

(2) $t2 = c + d$

$t3 = e - t2$

$t1 = a + b$

$t4 = t1 - t3$

mov eax, c ; $t2 = c + d$

add eax, d

mov edx, e ; $t3 = e - t2$

sub edx, eax

mov [ESP+0CH], eax

; $t1 = a + b$

mov eax, a

add eax, b

mov [ESP+08H], eax

; $t4 = t1 - t3$

sub eax, edx

临时寄存器池：基本思想 FIFO

- **进入基本块：**清空临时寄存器池
- **全局变量、局部变量使用临时寄存器：**向临时寄存器池申请
- **申请处理：**
 - 有空闲寄存器：分配申请，做标识
 - 没有空闲寄存器：（启发式）**选取一个在即将生成代码中不会被使用的寄存器写回相应的内存空间**，标识该寄存器被新的变量占用，返回该寄存器
- **退出基本块（或函数调用发生前）：**将寄存器池中的值写回内存，清空临时寄存器池

作业： p381

第十五章 1,4,5,6

谢谢!