目标代码生成

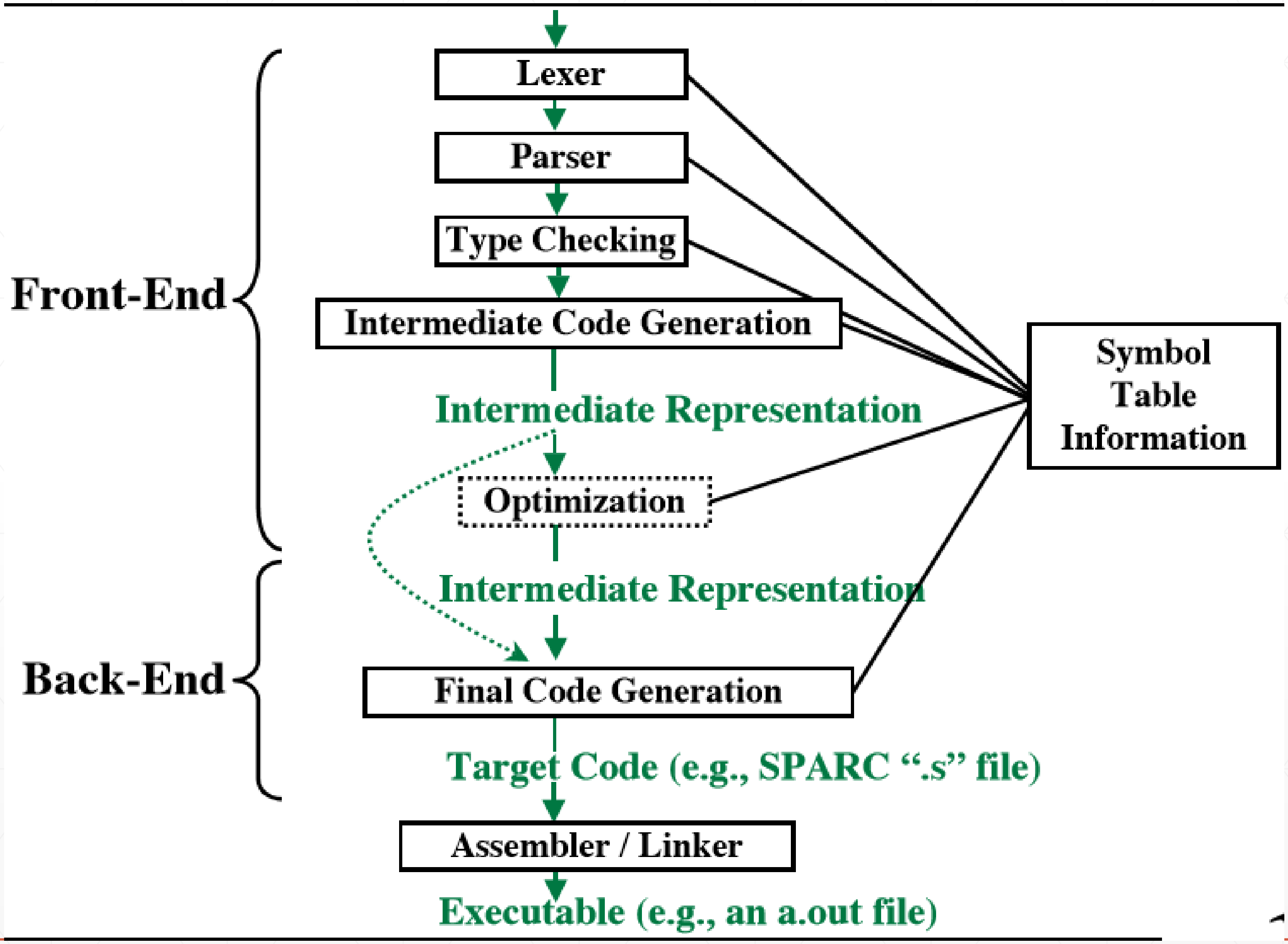
北京理工大学　计算机学院

Thanks

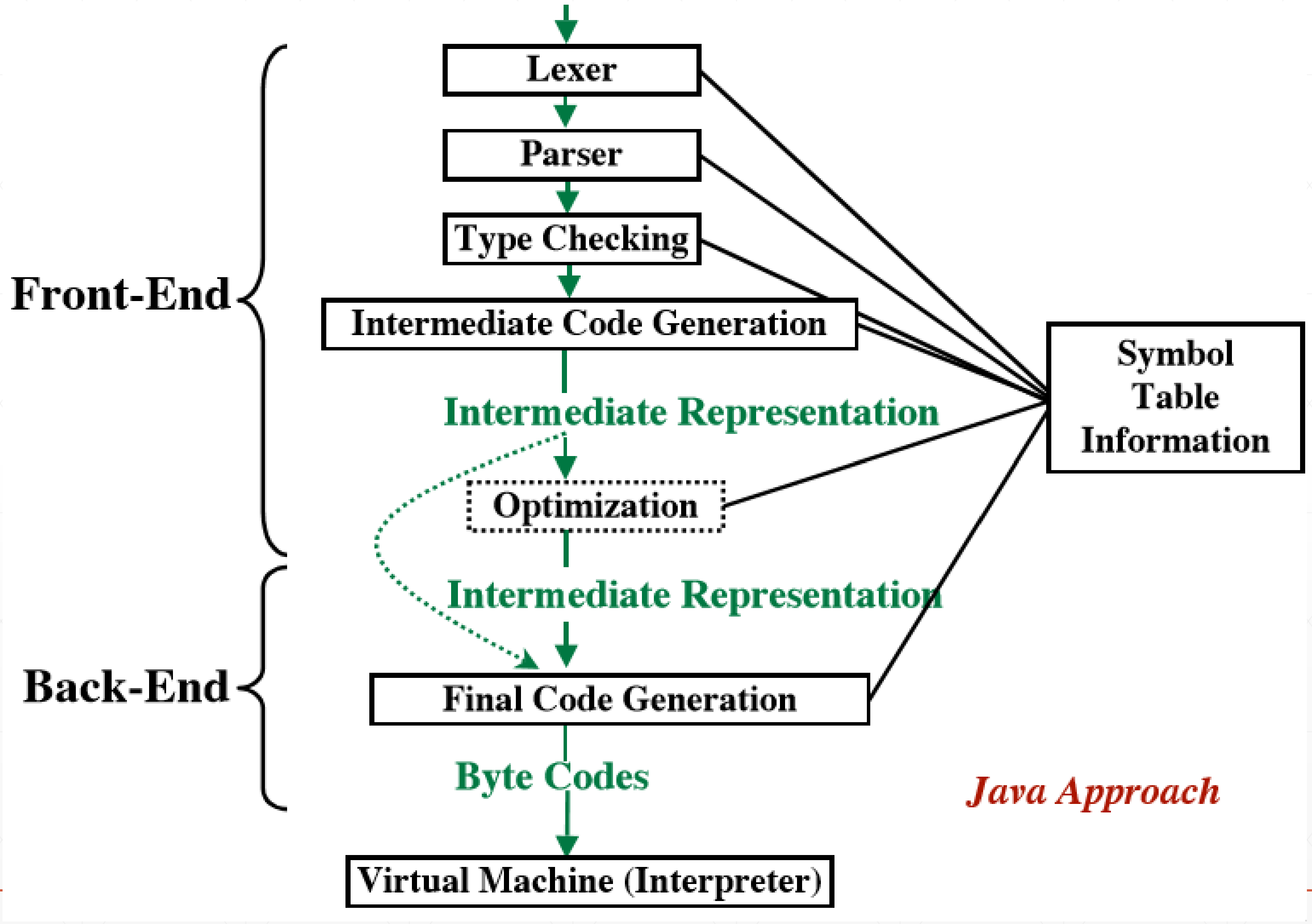
- Prof. David August, Princeton University
- Keith D. Cooper, Ken Kennedy & Linda Torczon, Rice University
- Harry H. Porter, 2006, COMP36512
- Mohamed Zahran, G22, Compiler Construction , NYU
- Alexander Krolik, COMP 520: Compiler Design

内容

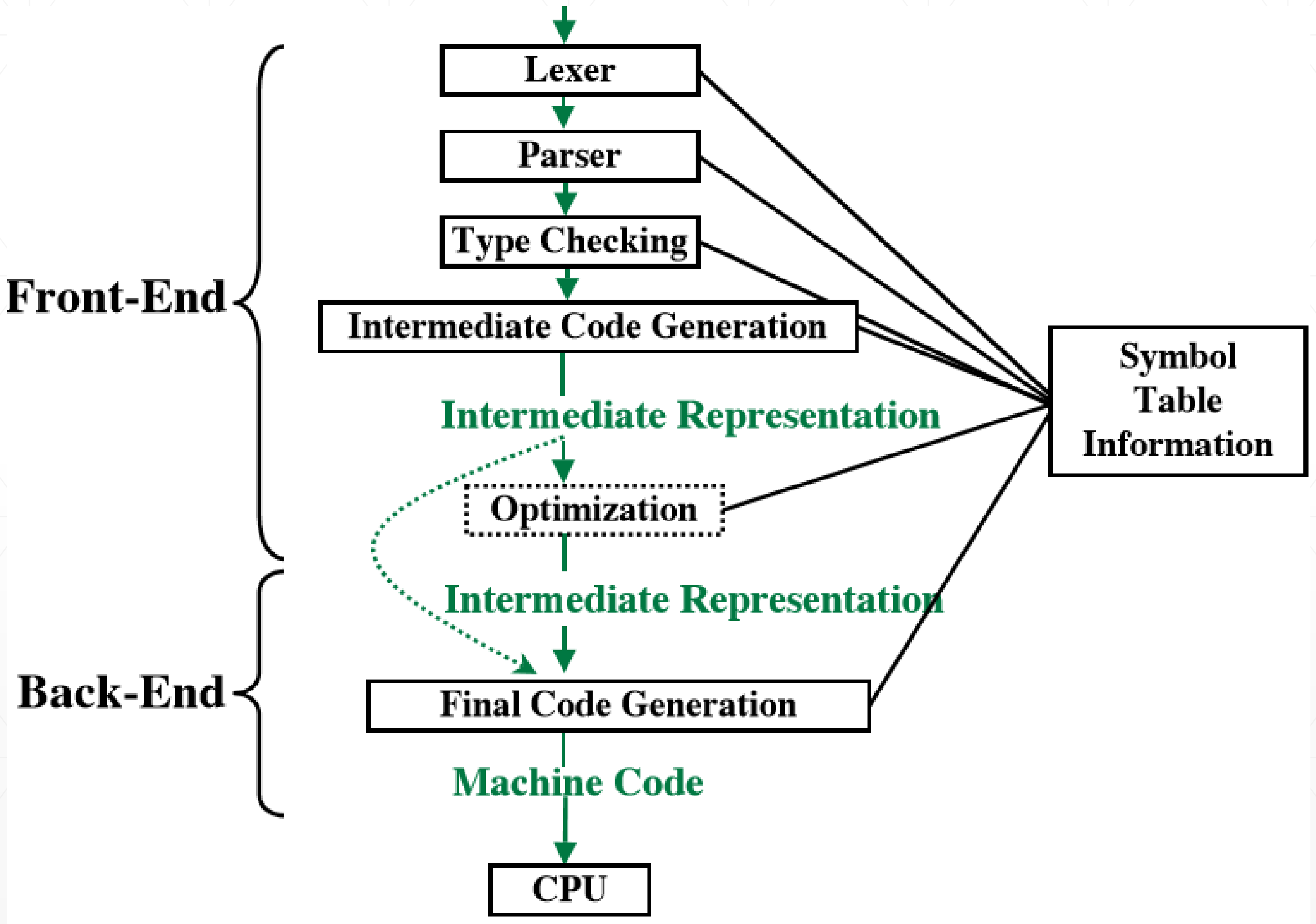
概览


Front-End
Back-End
Lexer
Parser
Type Checking
Intermediate Code Generation
Intermediate Representation
Optimization
Intermediate Representation
Final Code Generation
Target Code (e.g., SPARC ".s" file)
Assembler / Linker
Executable (e.g., an a.out file)
Symbol Table Information

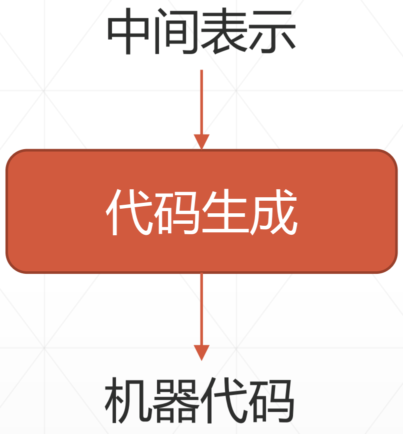
概览


Front-End
Back-End
Lexer
Parser
Type Checking
Intermediate Code Generation
Intermediate Representation
Optimization
Intermediate Representation
Final Code Generation
Byte Codes
Virtual Machine (Interpreter)
Symbol Table Information
Java Approach

概览


Front-End
Back-End
Lexer
Parser
Type Checking
Intermediate Code Generation
Intermediate Representation
Optimization
Intermediate Representation
Final Code Generation
Machine Code
CPU
Symbol Table Information

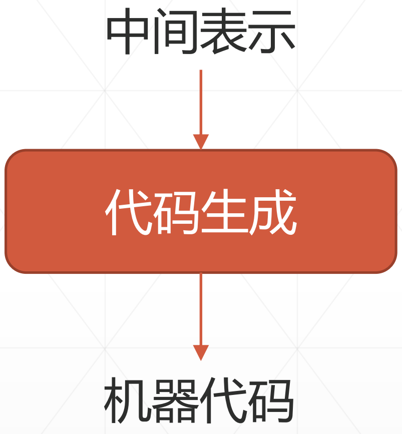
概览

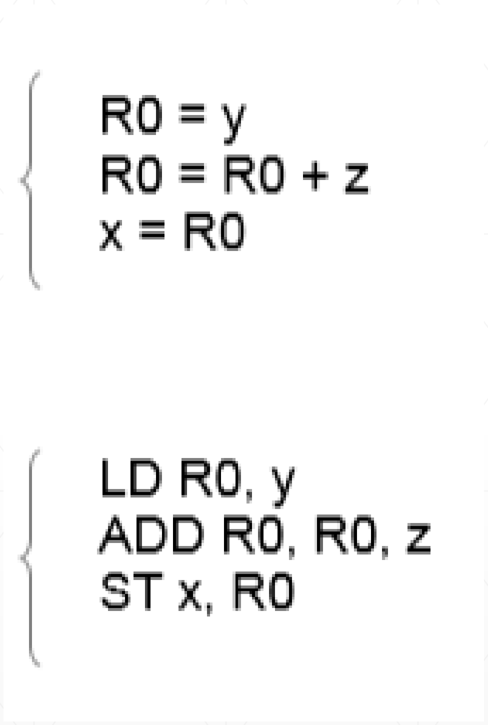


- 3地址指令（四元式、三元式和间接三元式）
- 逆波兰式
- 图形化的表示（语法树 / DAG等）

- RISC（寄存器比较多，三地址形式，寻址模式简单）
- CISC（寄存器比较少，量地址形式，多变的寻址模式，指令边长，不同类的寄存器）
- 基于堆栈的机器

概览





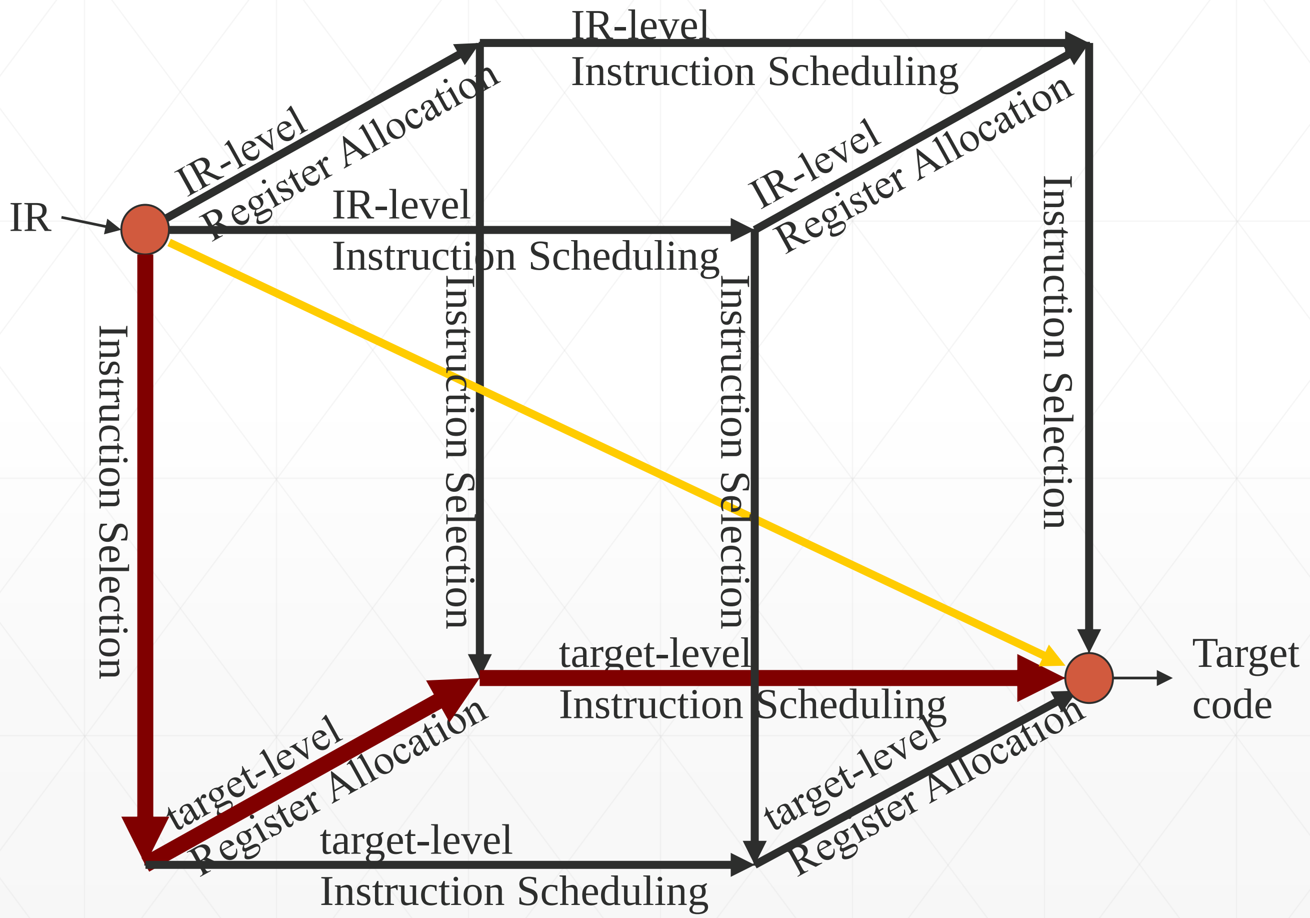
```
R0 = y
R0 = R0 + z
x = R0
```

```
LD R0, y
ADD R0, R0, z
ST x, R0
```

概览

- 从中间代码到目标代码包括
  - 指令选择/Instruction selection
  - 寄存器分配/Register allocation
  - 指令调度/Instruction scheduling
- 生成最优目标代码的问题通常是NP难
  - 近似算法
  - 启发式算法
  - 保守估计

概览


IR
IR-level Register Allocation
IR-level Instruction Scheduling
IR-level Instruction Scheduling
IR-level Register Allocation
Instruction Selection
Instruction Selection
Instruction Selection
Instruction Selection
target-level Instruction Scheduling
target-level Register Allocation
target-level Instruction Scheduling
target-level Register Allocation
Target code

概览：指令选择

- 指令选择的复杂度依赖于
  - IR的层次：
    - 低层次IR有助于生成高效的代码
  - 指令集本身的特点
    - 指令集的一致性和完备性
    - 浮点数需要特定的寄存器
  - 期望生成的目标代码质量
    - INC a / LD R0, a; ADD R0, R0, #1; ST R0, a;

概览：寄存器分配

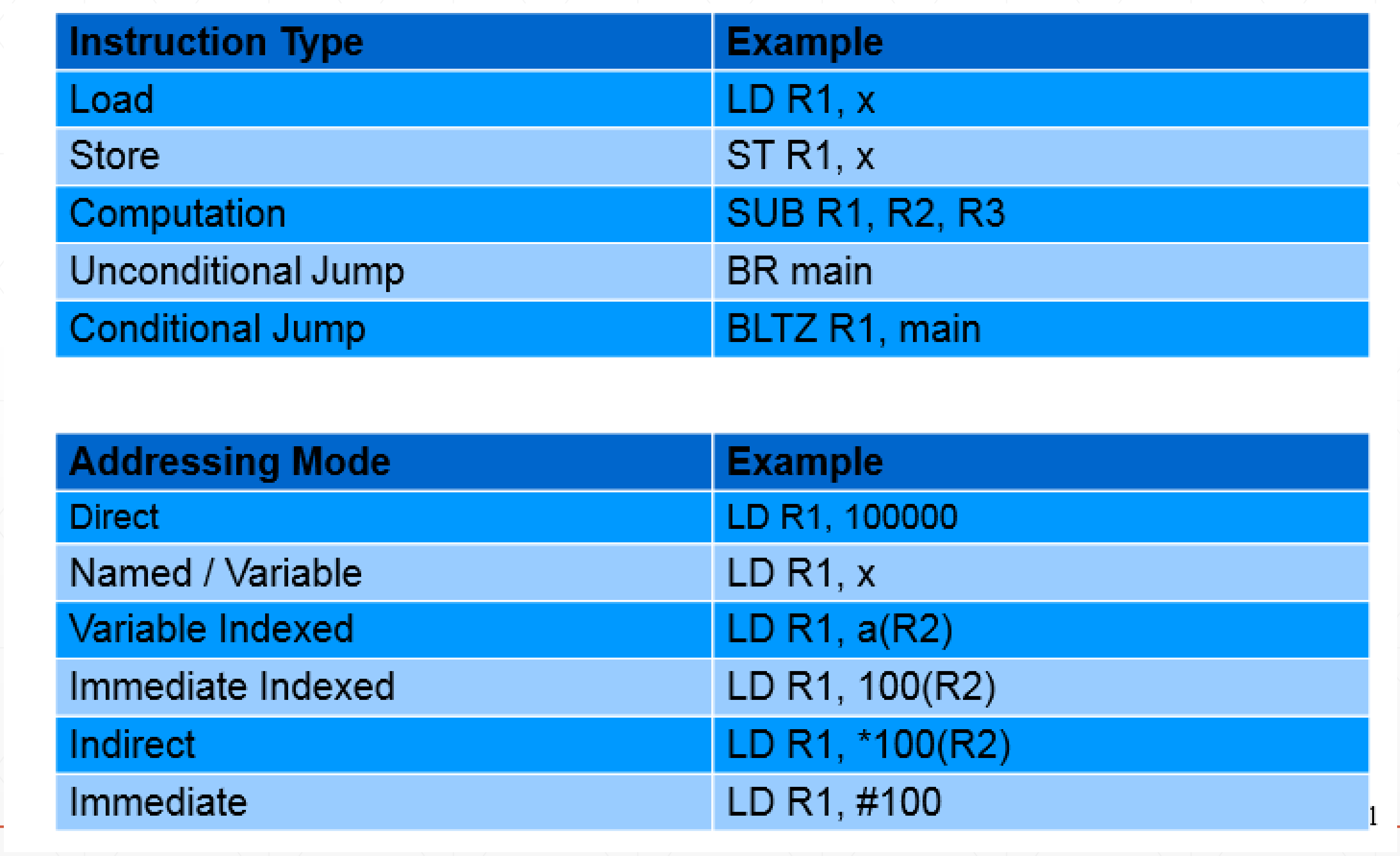
- 寄存器分配主要涉及
  - 将哪些变量放到寄存器中
  - 分配那个寄存器给一个变量
- 找到最优分配是一个NP-Complete问题
  - 寄存器的组合使用（双精度计算）
  - 为特定指令保留特定寄存器

概览：指令调度

- 指令的顺序影响执行效率
- 选择最优的顺序是NP-complete

目标机和调用惯例

- 一个简化的目标机

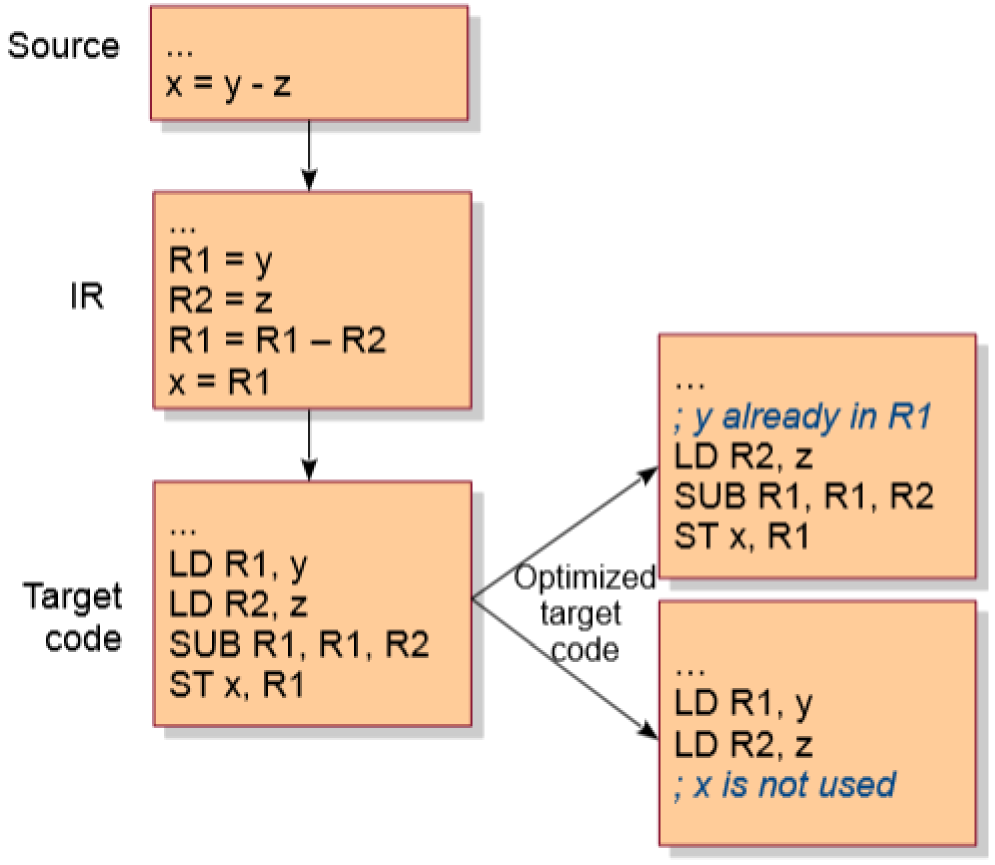


Instruction Type	Example
Load	LD R1, x
Store	ST R1, x
Computation	SUB R1, R2, R3
Unconditional Jump	BR main
Conditional Jump	BLTZ R1, main

Addressing Mode	Example
Direct	LD R1, 100000
Named / Variable	LD R1, x
Variable Indexed	LD R1, a(R2)
Immediate Indexed	LD R1, 100(R2)
Indirect	LD R1, *100(R2)
Immediate	LD R1, #100

目标机和调用惯例

- 代码生成示例



```
...
x = y - z
```

```
...
R1 = y
R2 = z
R1 = R1 – R2
x = R1
```

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

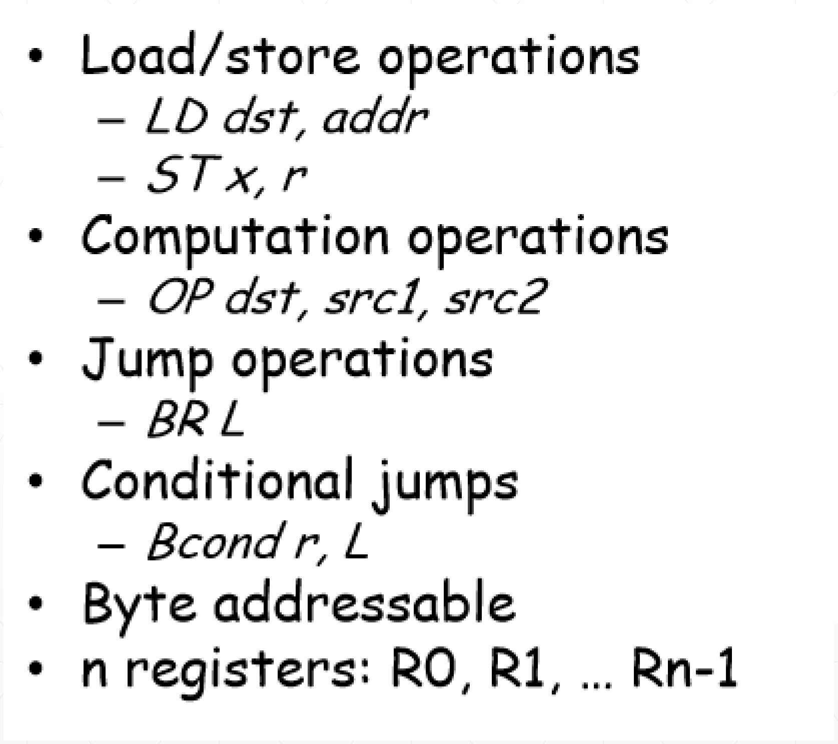
```
...
; y already in R1
LD R2, z
SUB R1, R1, R2
ST x, R1
```

```
...
LD R1, y
LD R2, z
; x is not used
```

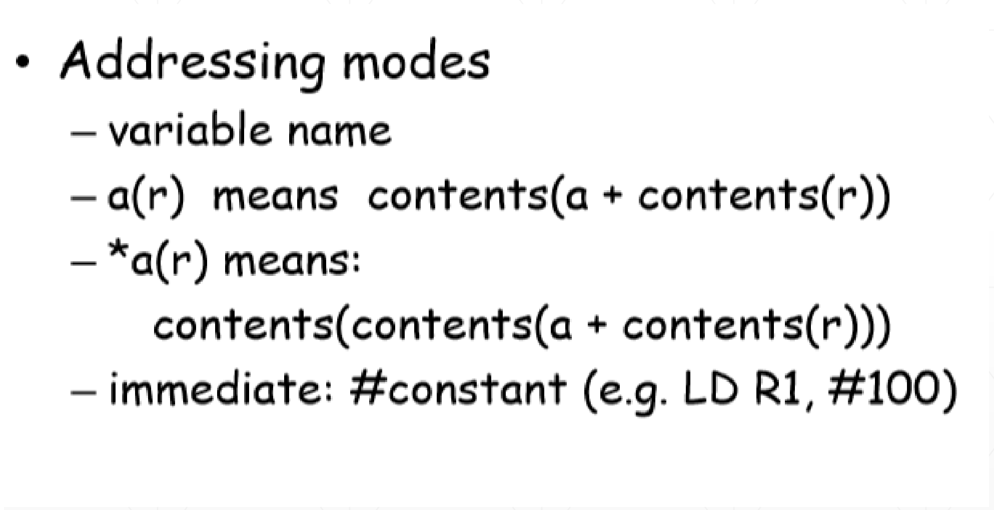
Optimization and Code Generation are often run together multiple-times.

目标机和调用惯例

- Simple target machine



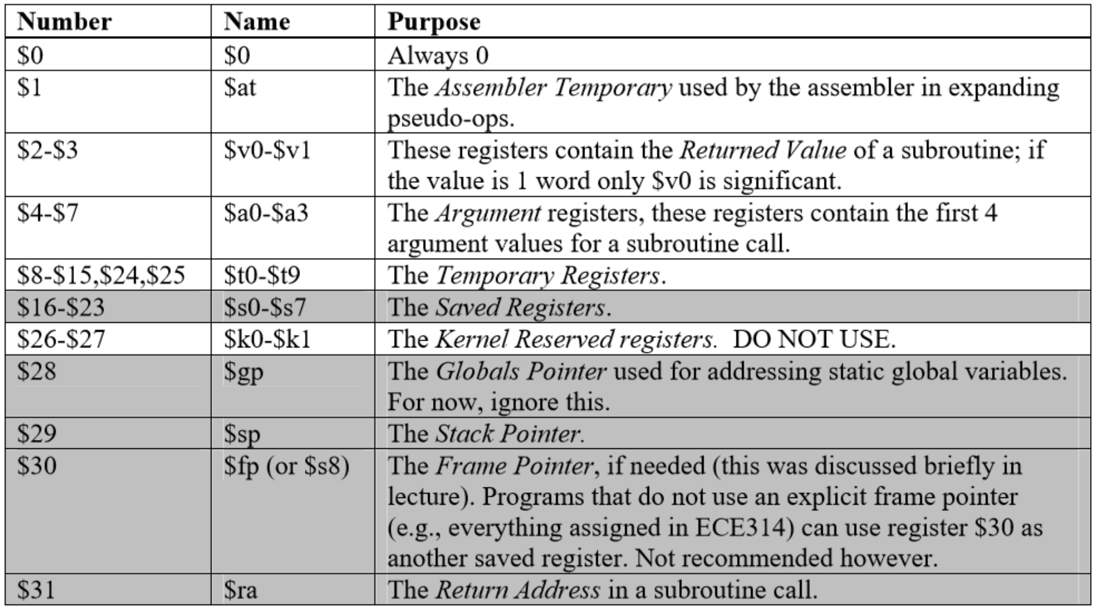
- Load/store operations
  - *LD dst, addr*
  - *ST x, r*
- Computation operations
  - *OP dst, src1, src2*
- Jump operations
  - *BR L*
- Conditional jumps
  - *Bcond r, L*
- Byte addressable
- n registers: R0, R1, ... Rn-1



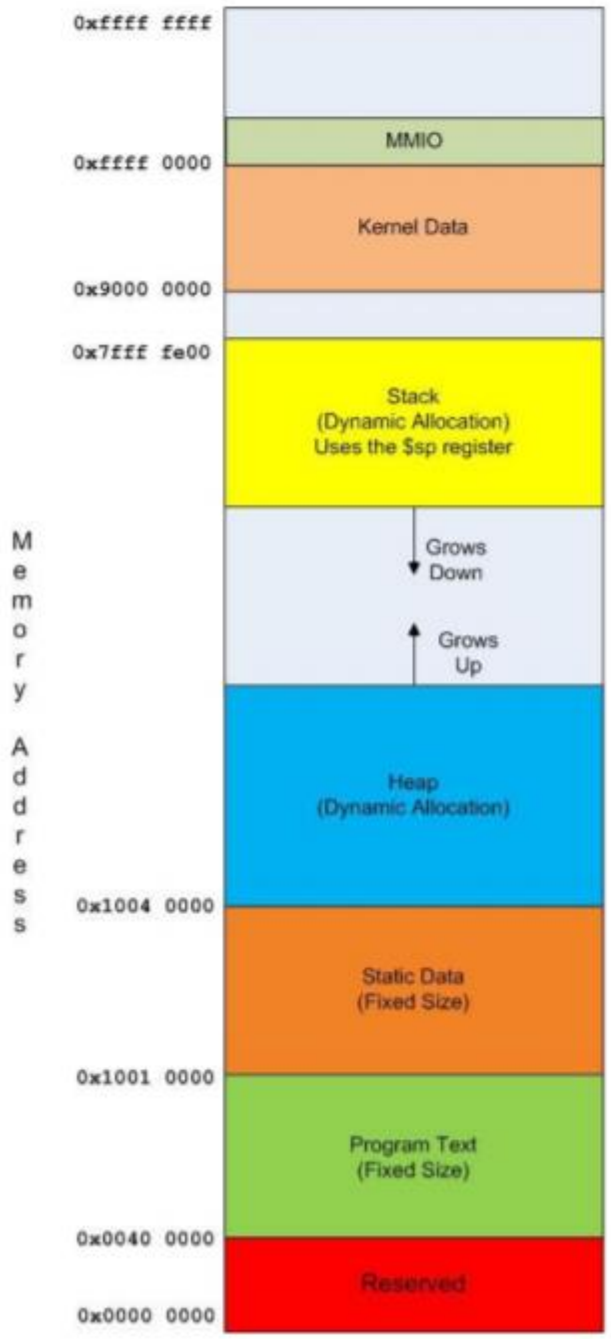
- Addressing modes
  - variable name
  - a(r) means contents(a + contents(r))
  - *a(r) means:
    contents(contents(a + contents(r)))
  - immediate: #constant (e.g. LD R1, #100)

目标机和调用惯例

- MIPS 内存布局和寄存器使用

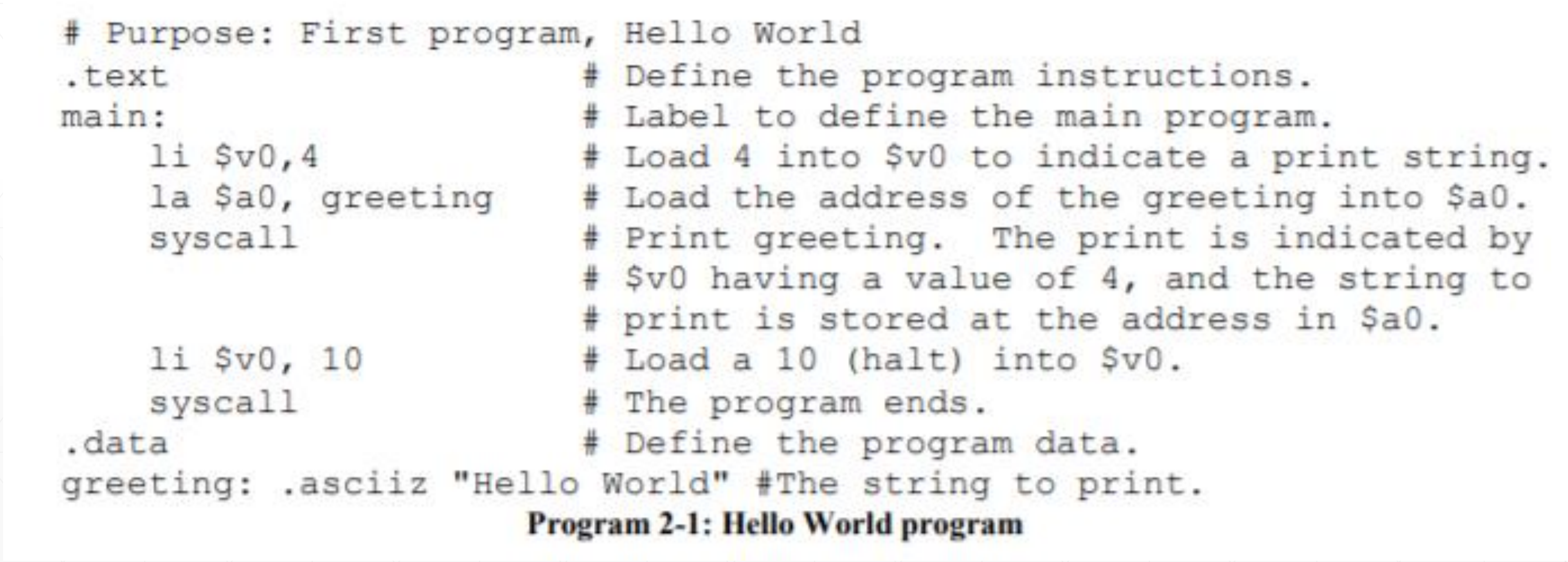


Number	Name	Purpose
$0	$0	Always 0
$1	$at	The *Assembler Temporary* used by the assembler in expanding pseudo-ops.
$2-$3	$v0-$v1	These registers contain the *Returned Value* of a subroutine; if the value is 1 word only $v0 is significant.
$4-$7	$a0-$a3	The *Argument* registers, these registers contain the first 4 argument values for a subroutine call.
$8-$15,$24,$25	$t0-$t9	The *Temporary Registers*.
$16-$23	$s0-$s7	The *Saved Registers*.
$26-$27	$k0-$k1	The *Kernel Reserved registers*. DO NOT USE.
$28	$gp	The *Globals Pointer* used for addressing static global variables. For now, ignore this.
$29	$sp	The *Stack Pointer*.
$30	$fp (or $s8)	The *Frame Pointer*, if needed (this was discussed briefly in lecture). Programs that do not use an explicit frame pointer (e.g., everything assigned in ECE314) can use register $30 as another saved register. Not recommended however.
$31	$ra	The *Return Address* in a subroutine call.



目标机和调用惯例

- MIPS （MARS模拟器）示例代码
  - 输出Hello World

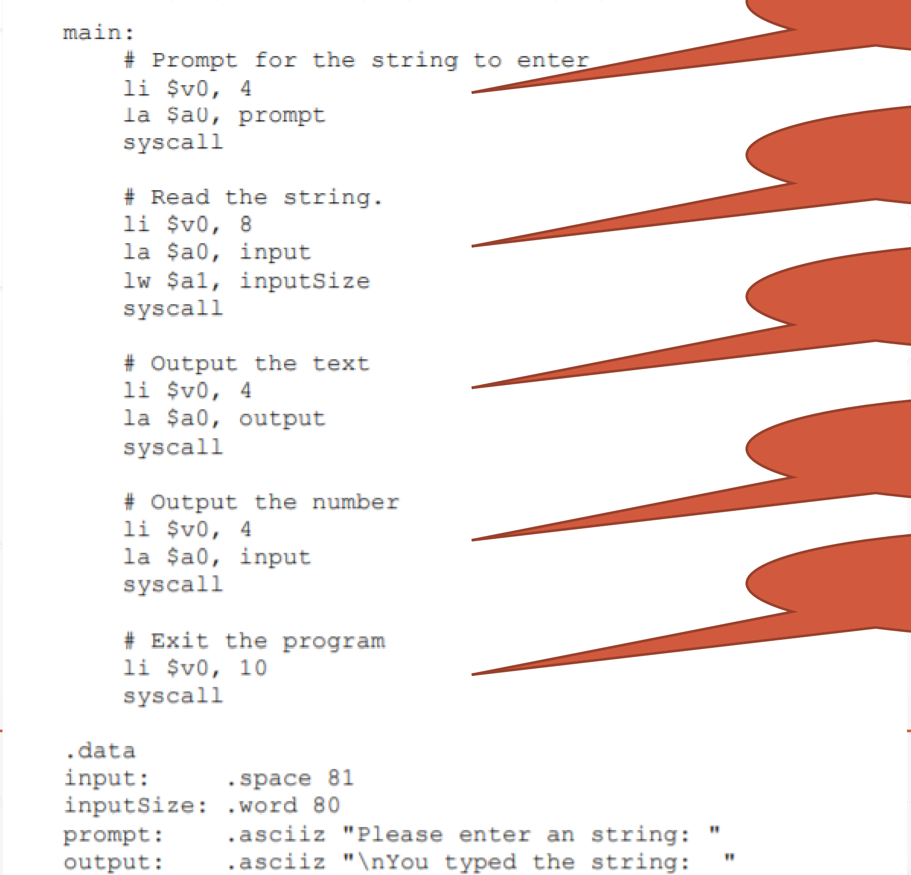


```
# Purpose: First program, Hello World
.text                       # Define the program instructions.
main:                       # Label to define the main program.
    li $v0,4                # Load 4 into $v0 to indicate a print string.
    la $a0, greeting        # Load the address of the greeting into $a0.
    syscall                 # Print greeting.  The print is indicated by
                            # $v0 having a value of 4, and the string to
                            # print is stored at the address in $a0.
    li $v0, 10              # Load a 10 (halt) into $v0.
    syscall                 # The program ends.
.data                       # Define the program data.
greeting: .asciiz "Hello World" #The string to print.
```

Program 2-1: Hello World program

目标机和调用惯例

- MIPS（MARS模拟器）示例代码
  - 读入字符串并输出



```
main:
     # Prompt for the string to enter
     li $v0, 4
     la $a0, prompt
     syscall

     # Read the string.
     li $v0, 8
     la $a0, input
     lw $a1, inputSize
     syscall

     # Output the text
     li $v0, 4
     la $a0, output
     syscall

     # Output the number
     li $v0, 4
     la $a0, input
     syscall

     # Exit the program
     li $v0, 10
     syscall

.data
input:      .space 81
inputSize:  .word 80
prompt:     .asciiz "Please enter an string: "
output:     .asciiz "\nYou typed the string:  "
```

输出提示信息

读输入字符串

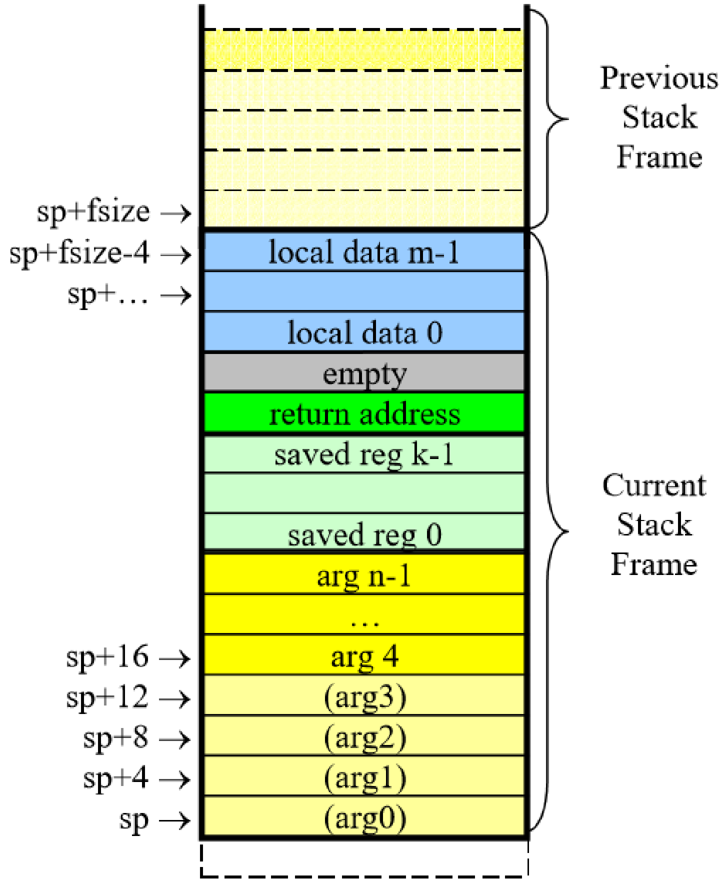
输出提示信息

输出提示信息

终止程序运行

目标机和调用惯例

- MIPS活动记录



Note that in this figure the stack is growing in a *downward* direction. I.e., the *top* of the stack is at the *bottom* of the picture.

一个简单的代码生成器

- 针对每个基本块完成代码生成
- 假设只有唯一的目标机指令选择方案
- 基本过程
  - 遍历基本块中的每个三地址指令
  - 确定需要将哪些操作对象加载到寄存器
  - 生成从内存加载到寄存器的指令
  - 生成计算指令
  - 生成必要的写回指令（从寄存器到内存）

一个简单的代码生成器

- 翻译过程中的辅助信息
  - 哪些变量的值保存在寄存器里面？以及是哪个寄存器？
  - 与变量相关联的存储位置是否保存有最新的值？
- 翻译过程中的辅助数据结构
  - **寄存器描述符:** 用于记录每个寄存器当前存储的变量
  - **地址描述符:** ：用于记录每个程序变量最新值的存储位置（内存或寄存器）

一个简单的代码生成器

- 假设有足够多的寄存器（暂时不考虑寄存器分配的问题）
- *getReg(I)* 函数
  - **Input:** 三地址指令I
  - **Output:** 指令I中操作数对应的寄存器
  - 该函数可以访问所有的**寄存器描述符**和**变量描述符**

一个简单的代码生成器

- 对于三地址指令：$x=y+z$
  - 使用$getReg(x = y + z)$为x、y和z选择寄存器，假设对应的寄存器为R_x、R_y和R_z
  - 如果y不在R_y中（根据R_y的寄存器描述符判断），生成指令：$LD\ R_y, Addr(y)$
  - 如果z不在R_z中（根据R_z的寄存器描述符判断），生成指令：$LD\ R_z, Addr(z)$
  - 生成指令：$ADD\ R_x, R_y, R_z$
  - 将R_x从其他变量（除x以外）的描述符中删除
  - 基本块结束时，如果x的最新值不在内存里，则生成：$ST\ Addr(x), R_x$

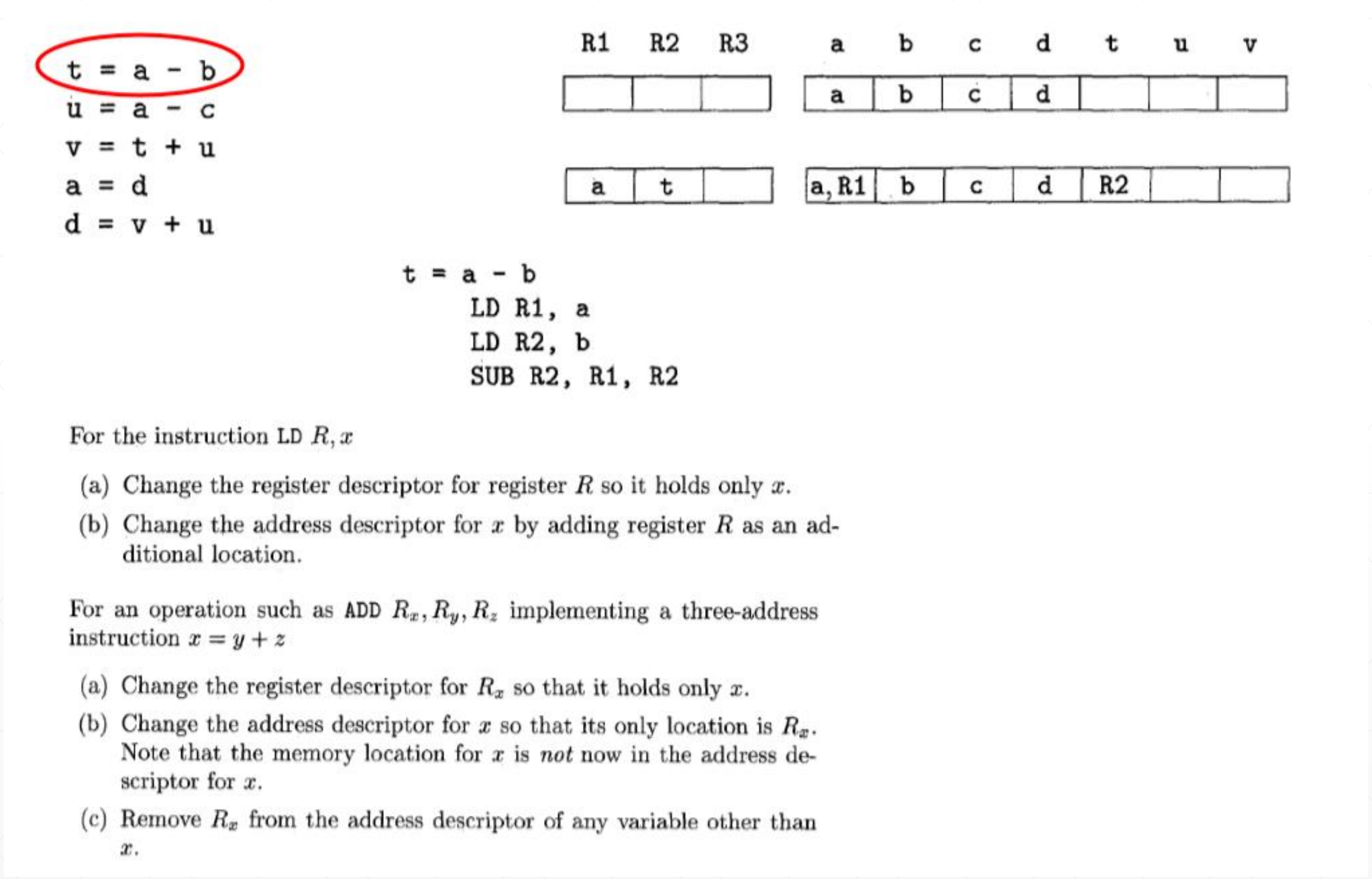
一个简单的代码生成器

- 寄存器和地址描述符管理
  - 针对load指令： ***LD R, Addr(x)***
    - 将寄存器***R***的描述符设置为持有***x***
    - 将变量***x***的变量描述符中增加***R***，表示值也在寄存器***R***中
  - 针对store指令： ***ST Addr(x), R***
    - 将变量***x***的变量描述符中增加对应的内存位置，表示值也在内存中

一个简单的代码生成器

- 寄存器和地址描述符管理
  - 针对三地址代码： $x = y$,
    - getReg始终为x和y分配相同的寄存器
    - 生成加载y的指令： $LD\ R_y, Addr(y)$
    - 将x添加到寄存器R_y的描述符中
    - 修改x的描述符使得该描述符只包含R_y

一个简单的代码生成器



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

R1	R2	R3
a	t	

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

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

For the instruction LD R, x

(a) Change the register descriptor for register R so it holds only x.

(b) Change the address descriptor for x by adding register R as an additional location.

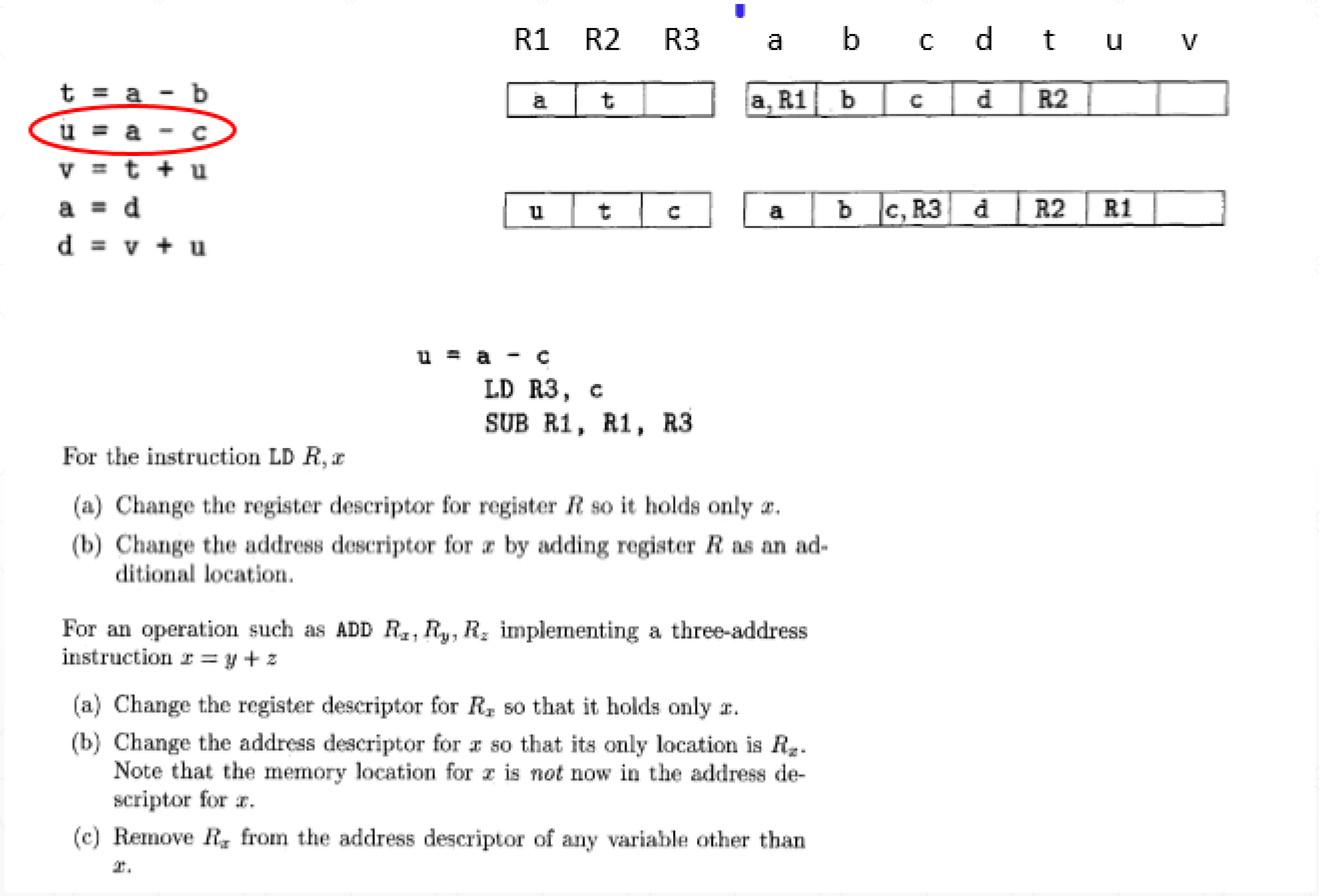
For an operation such as ADD R_x, R_y, R_z implementing a three-address instruction $x = y + z$

(a) Change the register descriptor for R_x so that it holds only x.

(b) Change the address descriptor for x so that its only location is R_x. Note that the memory location for x is *not* now in the address descriptor for x.

(c) Remove R_x from the address descriptor of any variable other than x.

一个简单的代码生成器



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

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

```
u = a - c
    LD R3, c
    SUB R1, R1, R3
```

For the instruction LD R, x

(a) Change the register descriptor for register R so it holds only x.

(b) Change the address descriptor for x by adding register R as an additional location.

For an operation such as ADD R_x, R_y, R_z implementing a three-address instruction $x = y + z$

(a) Change the register descriptor for R_x so that it holds only x.

(b) Change the address descriptor for x so that its only location is R_x. Note that the memory location for x is *not* now in the address descriptor for x.

(c) Remove R_x from the address descriptor of any variable other than x.

一个简单的代码生成器

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

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

```
v = t + u
    ADD R3, R2, R1
```

For an operation such as ADD R_x, R_y, R_z implementing a three-address instruction $x = y + z$

(a) Change the register descriptor for R_x so that it holds only x.

(b) Change the address descriptor for x so that its only location is R_x. Note that the memory location for x is *not* now in the address descriptor for x.

(c) Remove R_x from the address descriptor of any variable other than x.

一个简单的代码生成器

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

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

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

For an operation such as ADD R_x, R_y, R_z implementing a three-address instruction $x = y + z$

(a) Change the register descriptor for R_x so that it holds only x.

(b) Change the address descriptor for x so that its only location is R_x. Note that the memory location for x is *not* now in the address descriptor for x.

(c) Remove R_x from the address descriptor of any variable other than x.

一个简单的代码生成器

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

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

exit

```
    ST a, R2
    ST d, R1
```

Ending the basic block: For each variable whose memory location is not up to date generate *ST x, R* (R is the register where x exists at end of the block)

For the instruction ST x, R, change the address descriptor for x to include its own memory location.

一个简单的代码生成器

- getReg函数实现机制
  - 如果变量*x*已经在寄存器中，直接返回对应的寄存器
  - 如果*y*不在寄存器，且有空闲的寄存器，则返回一个空闲寄存器
  - 否则选择一个已经占用的寄存器，根据寄存器描述符将对应的变量写回内存，然后返回对应的寄存器。

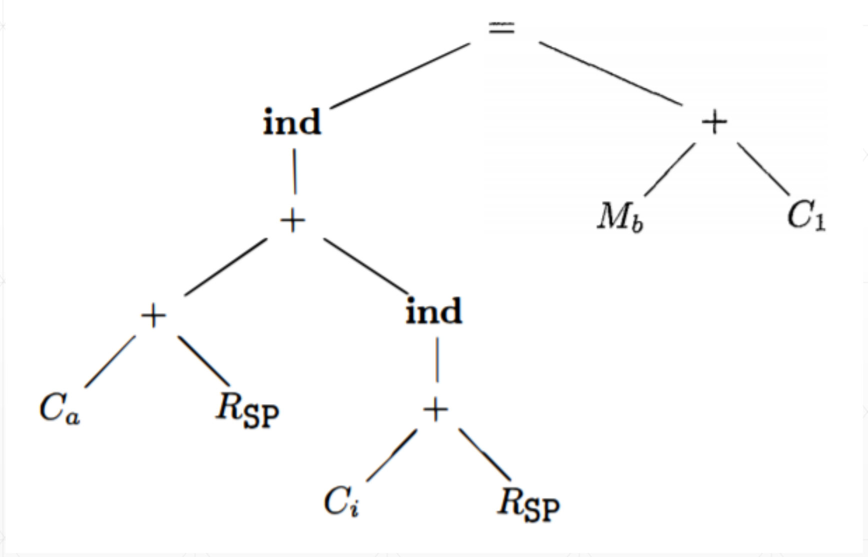
Spill

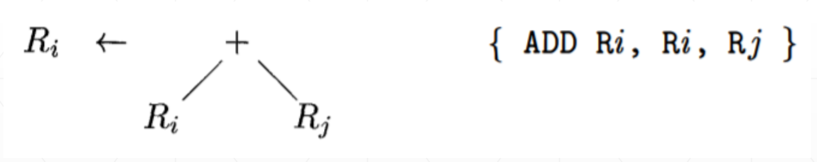
指令选择

- 很多编译器使用树状 IR 表示
  - 抽象语法树： Abstract syntax trees
  - 表达式的DAG或者数状表示
- 指令选择就是找到一个机器指令集合实现IR树中的操作
- 每个机器指令可以表示为一个IR树部分-tree pattern
- 指令选择的目标即是为IR树找到一个不重叠的模式覆盖

指令选择

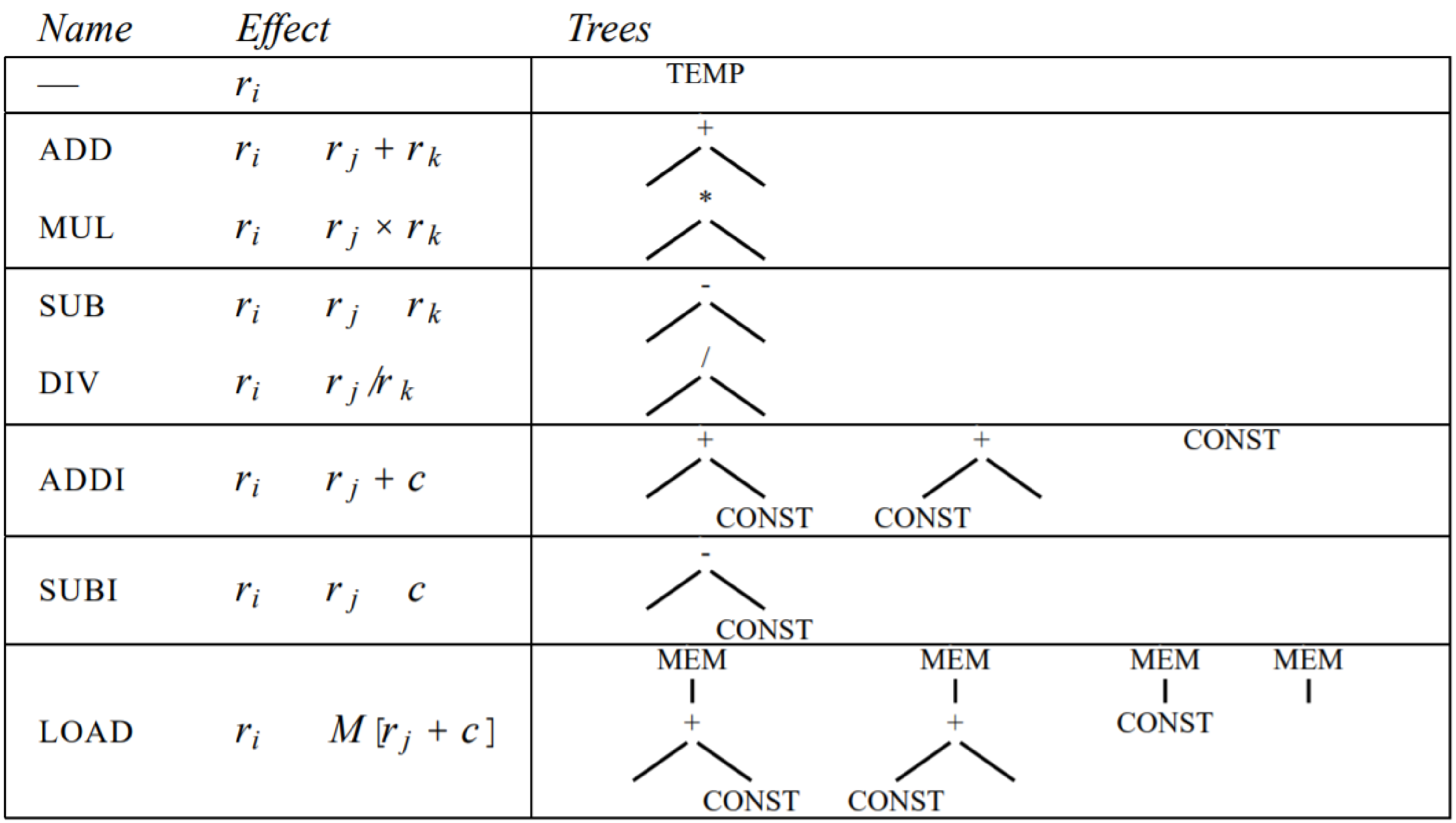
- 基于树的翻译方案





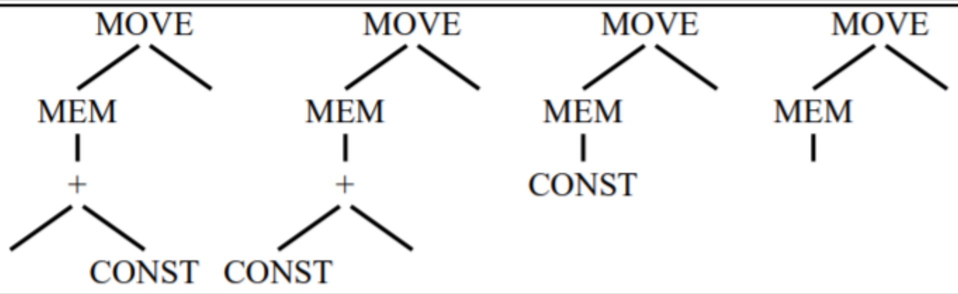
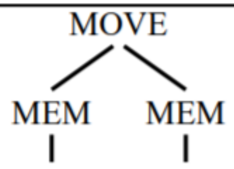
{ ADD Ri, Ri, Rj }

指令选择: Tree-pattern匹配



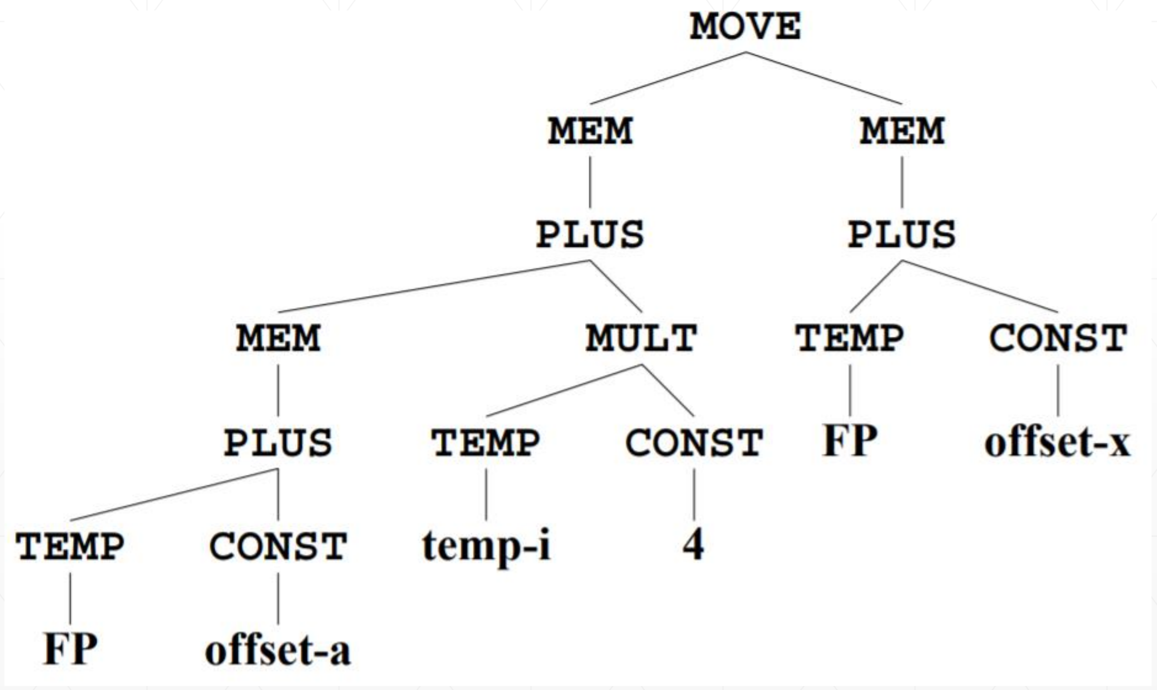
Name	*Effect*	*Trees*
—	r_i	TEMP
ADD	$r_i \quad r_j + r_k$	+
MUL	$r_i \quad r_j \times r_k$	*
SUB	$r_i \quad r_j \quad r_k$	-
DIV	$r_i \quad r_j / r_k$	/
ADDI	$r_i \quad r_j + c$	+ CONST; + CONST; CONST
SUBI	$r_i \quad r_j \quad c$	- CONST
LOAD	$r_i \quad M[r_j + c]$	MEM + CONST; MEM + CONST; MEM CONST; MEM

指令选择: Tree-pattern匹配

STORE	$M[r_j + c]$	r_i	 MOVE MEM + CONST MOVE MEM + CONST MOVE MEM CONST MOVE MEM
MOVEM	$M[r_j]$	$M[r_i]$	

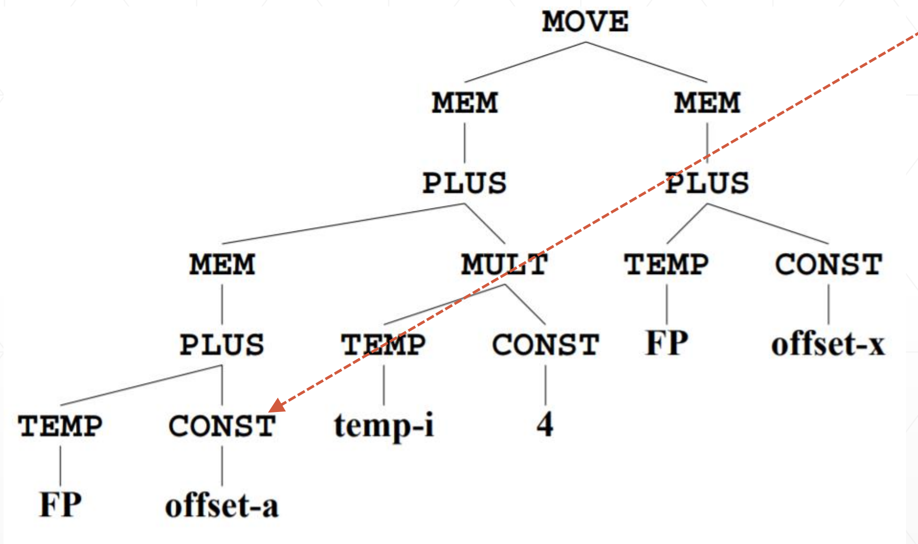
指令选择: Tree-pattern 匹配

- 假设*i*在寄存器中，*x*在内存中（活动记录中），则*a[i] := x* 表示为:



指令选择: Tree-pattern 匹配

- 单个节点匹配

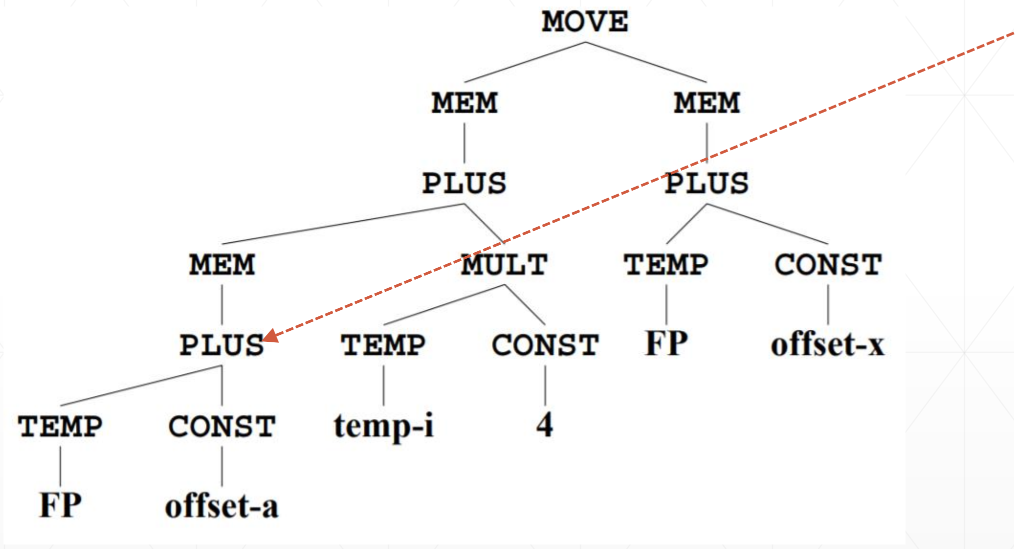


```
ADDI   r1 = r0 + offset_a
ADD    r2 = r1 + FP
LOAD   r3 = M[r2 + 0]

ADDI   r4 = r0 + 4
MUL    r5 = r4 * r_i

ADD    r6 = r3 + r5

ADDI   r7 = r0 + offset_x
ADD    r8 = r7 + FP
LOAD   r9 = M[r8 + 0]

STORE M[r6 + 0] = r9
```

9 registers, 10 instructions

指令选择: Tree-pattern 匹配

- 单个节点匹配

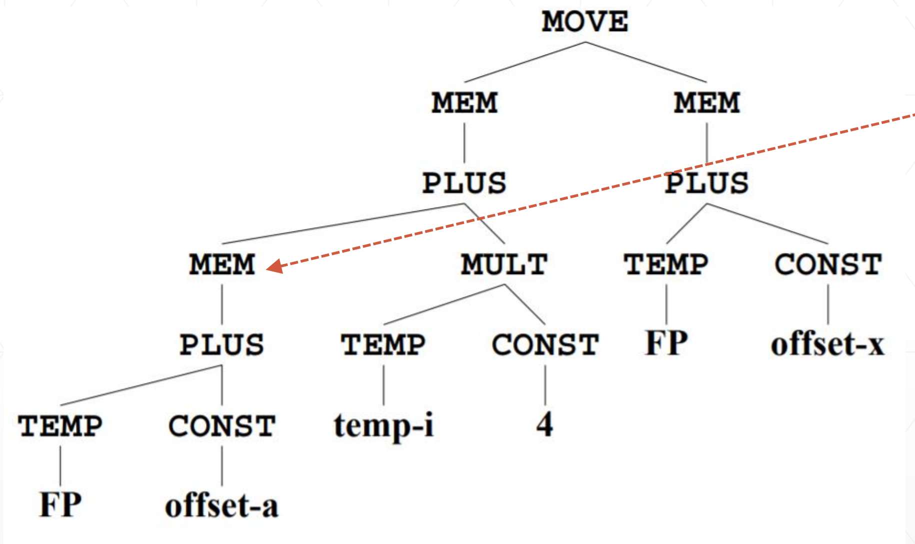


```
ADDI  r1 = r0 + offset_a
ADD   r2 = r1 + FP
LOAD  r3 = M[r2 + 0]

ADDI  r4 = r0 + 4
MUL   r5 = r4 * r_i

ADD   r6 = r3 + r5

ADDI  r7 = r0 + offset_x
ADD   r8 = r7 + FP
LOAD  r9 = M[r8 + 0]

STORE M[r6 + 0] = r9
```

9 registers, 10 instructions

指令选择: Tree-pattern 匹配

- 单个节点匹配

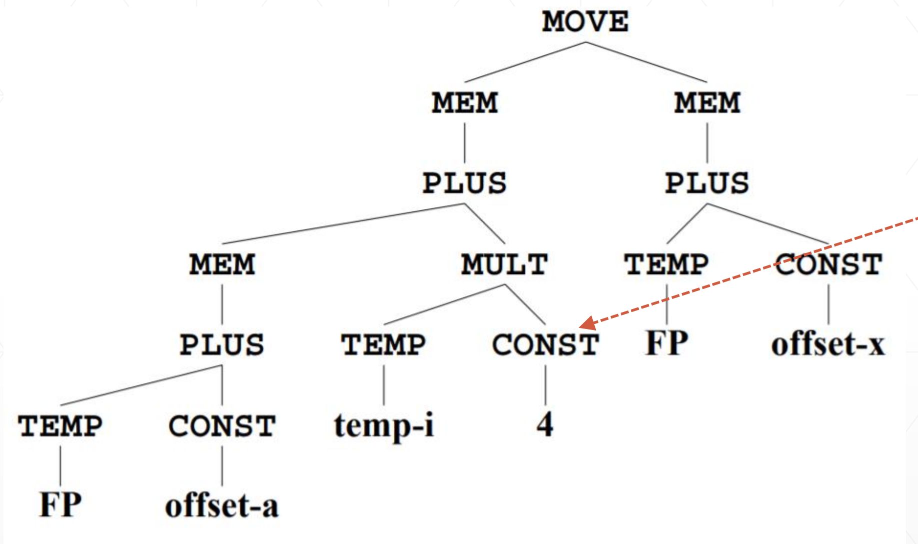


```
ADDI   r1 = r0 + offset_a
ADD    r2 = r1 + FP
LOAD   r3 = M[r2 + 0]

ADDI   r4 = r0 + 4
MUL    r5 = r4 * r_i

ADD    r6 = r3 + r5

ADDI   r7 = r0 + offset_x
ADD    r8 = r7 + FP
LOAD   r9 = M[r8 + 0]

STORE M[r6 + 0] = r9
```

9 registers, 10 instructions

指令选择: Tree-pattern 匹配

- 单个节点匹配

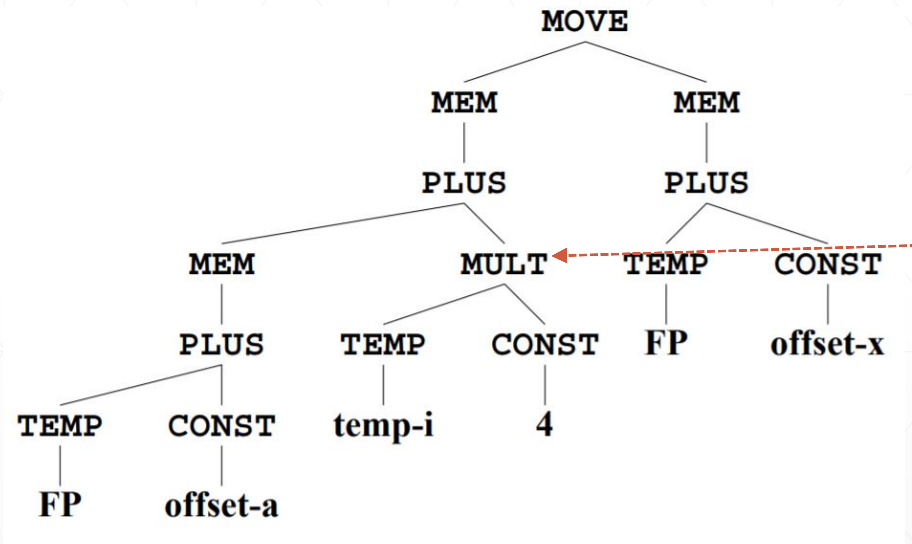


```
ADDI   r1 = r0 + offset_a
ADD    r2 = r1 + FP
LOAD   r3 = M[r2 + 0]

ADDI   r4 = r0 + 4
MUL    r5 = r4 * r_i

ADD    r6 = r3 + r5

ADDI   r7 = r0 + offset_x
ADD    r8 = r7 + FP
LOAD   r9 = M[r8 + 0]

STORE M[r6 + 0] = r9
```

9 registers, 10 instructions

指令选择: Tree-pattern 匹配

- 单个节点匹配

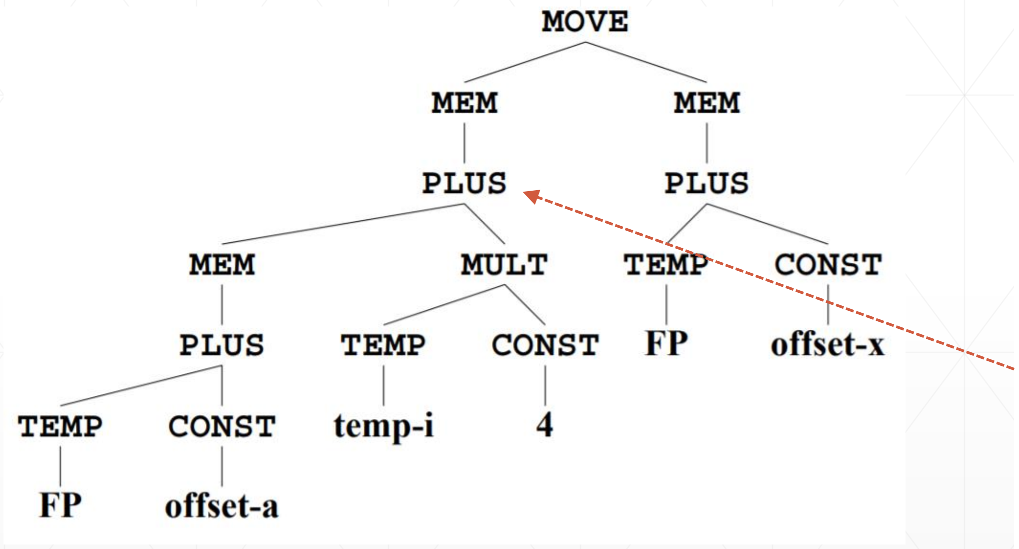


```
ADDI   r1 = r0 + offset_a
ADD    r2 = r1 + FP
LOAD   r3 = M[r2 + 0]

ADDI   r4 = r0 + 4
MUL    r5 = r4 * r_i

ADD    r6 = r3 + r5

ADDI   r7 = r0 + offset_x
ADD    r8 = r7 + FP
LOAD   r9 = M[r8 + 0]

STORE M[r6 + 0] = r9
```

9 registers, 10 instructions

指令选择: Tree-pattern 匹配

- 单个节点匹配

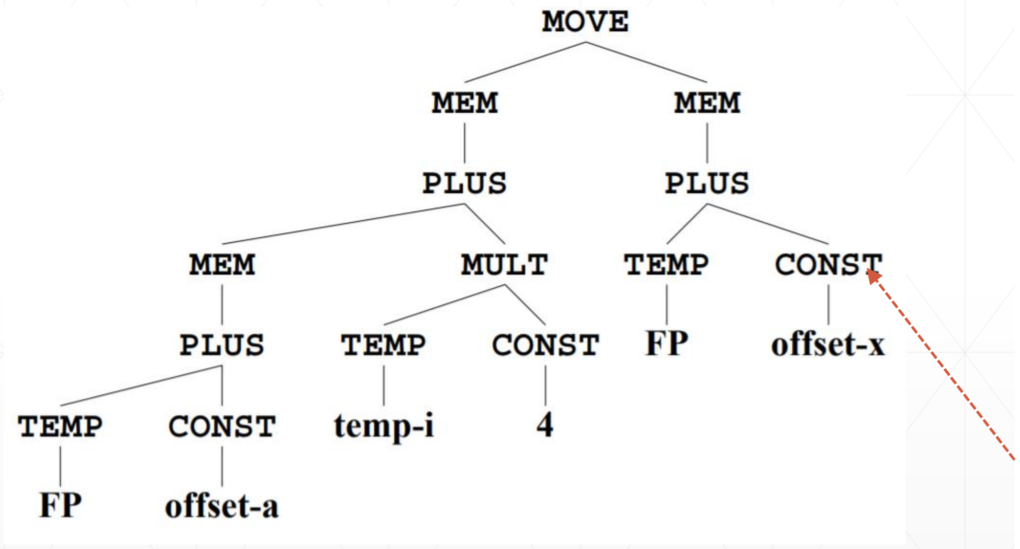


```
ADDI   r1 = r0 + offset_a
ADD    r2 = r1 + FP
LOAD   r3 = M[r2 + 0]

ADDI   r4 = r0 + 4
MUL    r5 = r4 * r_i

ADD    r6 = r3 + r5

ADDI   r7 = r0 + offset_x
ADD    r8 = r7 + FP
LOAD   r9 = M[r8 + 0]

STORE M[r6 + 0] = r9
```

9 registers, 10 instructions

指令选择: Tree-pattern 匹配

- 单个节点匹配

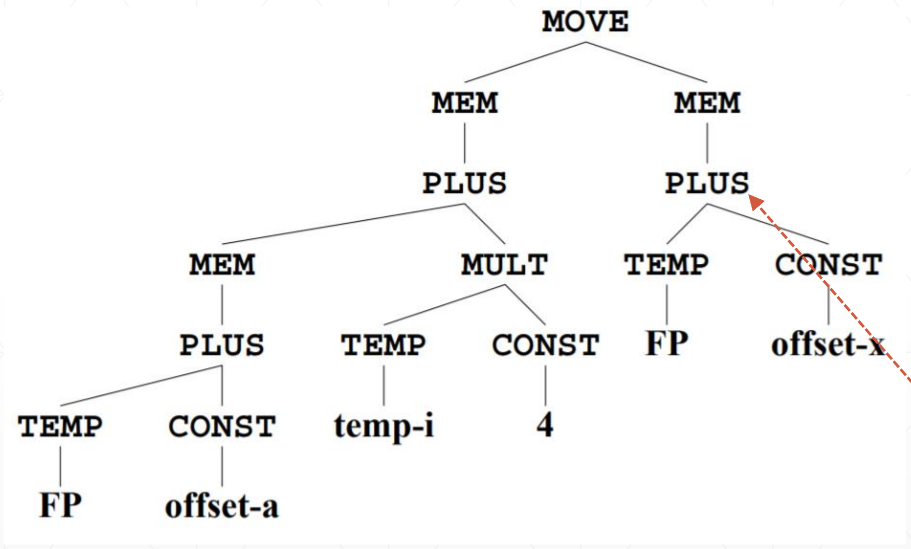


```
ADDI  r1 = r0 + offset_a
ADD   r2 = r1 + FP
LOAD  r3 = M[r2 + 0]

ADDI  r4 = r0 + 4
MUL   r5 = r4 * r_i

ADD   r6 = r3 + r5

ADDI  r7 = r0 + offset_x
ADD   r8 = r7 + FP
LOAD  r9 = M[r8 + 0]

STORE M[r6 + 0] = r9
```

9 registers, 10 instructions

指令选择: Tree-pattern 匹配

- 单个节点匹配

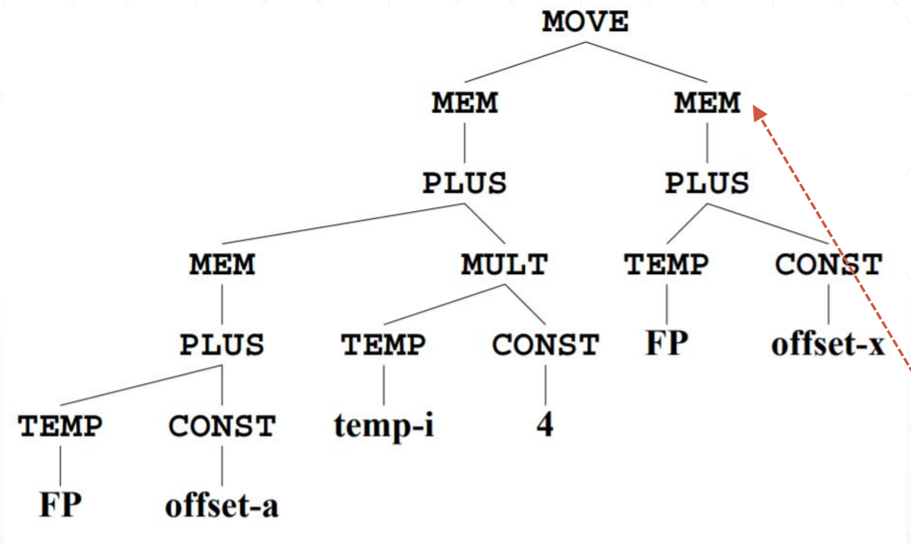


```
ADDI   r1 = r0 + offset_a
ADD    r2 = r1 + FP
LOAD   r3 = M[r2 + 0]

ADDI   r4 = r0 + 4
MUL    r5 = r4 * r_i

ADD    r6 = r3 + r5

ADDI   r7 = r0 + offset_x
ADD    r8 = r7 + FP
LOAD   r9 = M[r8 + 0]

STORE M[r6 + 0] = r9
```

9 registers, 10 instructions

指令选择: Tree-pattern 匹配

- 单个节点匹配

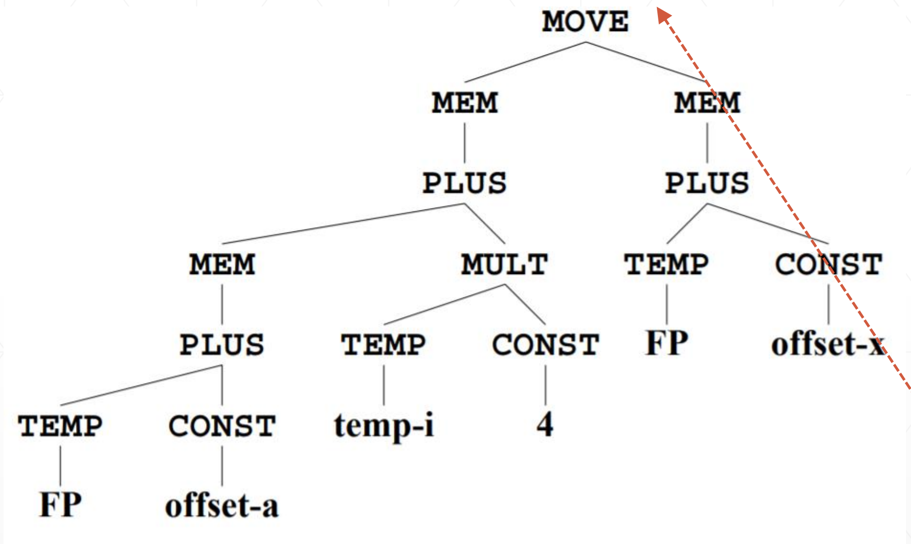


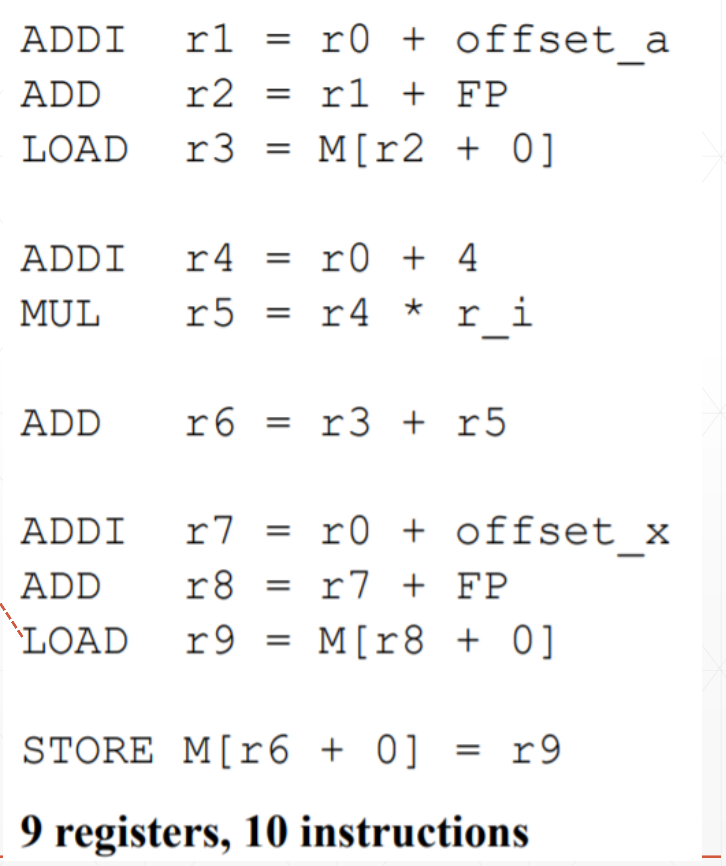
```
ADDI   r1 = r0 + offset_a
ADD    r2 = r1 + FP
LOAD   r3 = M[r2 + 0]

ADDI   r4 = r0 + 4
MUL    r5 = r4 * r_i

ADD    r6 = r3 + r5

ADDI   r7 = r0 + offset_x
ADD    r8 = r7 + FP
LOAD   r9 = M[r8 + 0]

STORE M[r6 + 0] = r9
```

9 registers, 10 instructions

指令选择: Tree-pattern 匹配

- 单个节点匹配

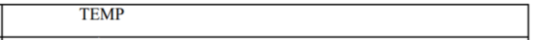
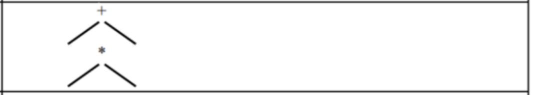
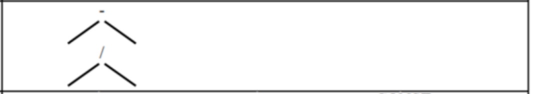
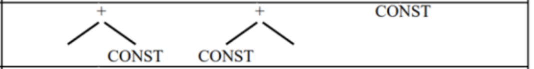
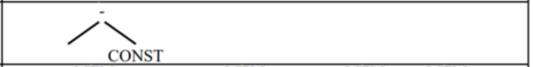
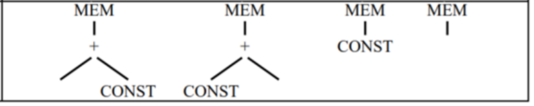


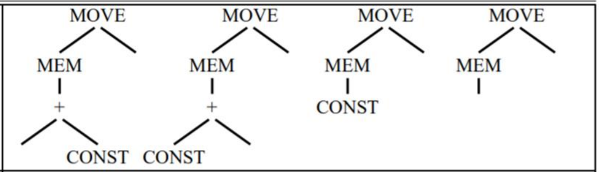
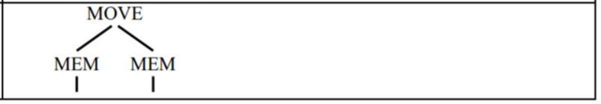


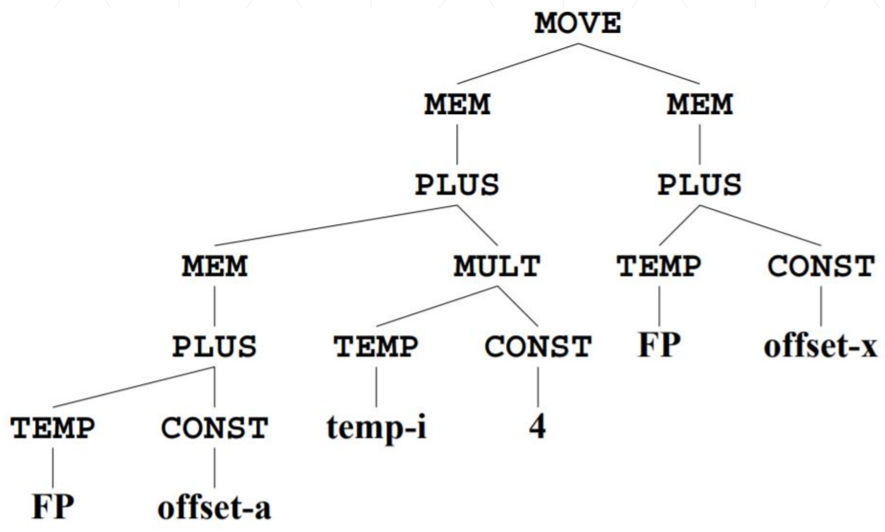
```
ADDI  r1 = r0 + offset_a
ADD   r2 = r1 + FP
LOAD  r3 = M[r2 + 0]

ADDI  r4 = r0 + 4
MUL   r5 = r4 * r_i

ADD   r6 = r3 + r5

ADDI  r7 = r0 + offset_x
ADD   r8 = r7 + FP
LOAD  r9 = M[r8 + 0]

STORE M[r6 + 0] = r9
```

9 registers, 10 instructions

指令选择: Tree-pattern匹配

Name	*Effect*		*Trees*
—	r_i		
ADD MUL	r_i r_i	$r_j + r_k$ $r_j \times r_k$	
SUB DIV	r_i r_i	$r_j \quad r_k$ r_j / r_k	
ADDI	r_i	$r_j + c$	
SUBI	r_i	$r_j \quad c$	
LOAD	r_i	$M[r_j + c]$	

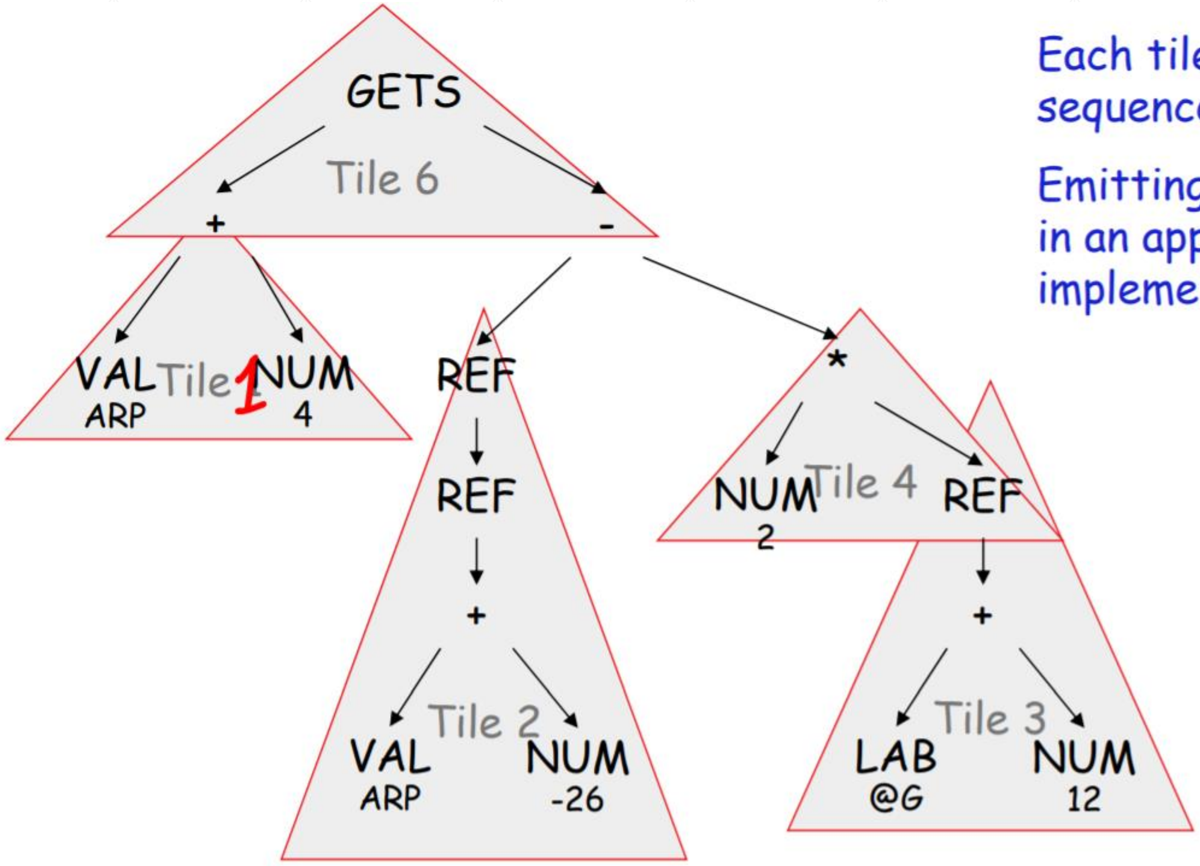
STORE	$M[r_j + c]$	r_i	
MOVEM	$M[r_j]$	$M[r_i]$	



```
LOAD  r3 = M[FP + offset_a]

ADDI  r4 = r0 + 4
MUL   r5 = r4 * r_i

ADD   r6 = r3 + r5

ADD   r8 = FP + offset_x
MOVEM M[r6] = M[r8]
```

5 registers, 6 instructions

指令选择

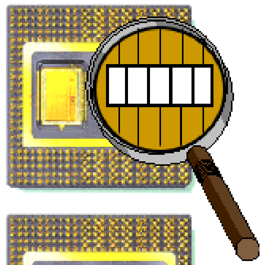
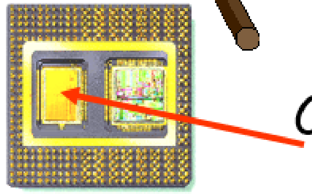
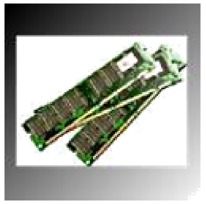


Each tile corresponds to a sequence of operations

Emitting those operations in an appropriate order implements the tree.

寄存器分配

- 计算机中的存储层次

	Registers	1 cycle	256-8000 bytes
	Cache	3 cycles	256k-1M
	Main memory	20-100 cycles	32M-1G
	Disk	0.5-5M cycles	4G-1T

寄存器分配

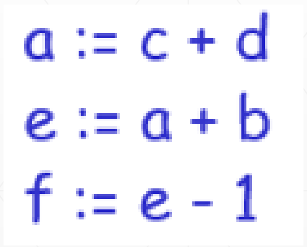
- 存储层次管理
  - **程序**视图中只有内存和磁盘
  - **程序员**负责将数据从磁盘移到内存（文件读写）
  - **CPU中硬件**负责内存和Cache之间的数据移动
  - **编译器**负责内存和寄存器之间的数据移动

寄存器分配

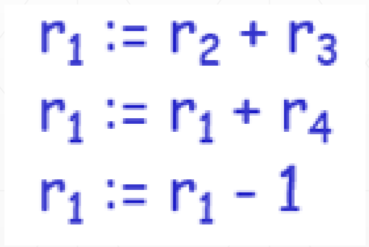
- 寄存器分配问题
  - 通常中间代码会有大量的临时变量，简化了中间代码生成和优化
  - 但是为最终代码生成带来了困难
- 解决思路
  - 重写中间代码使得每个时刻使用的临时变量比可用寄存器数少
  - 将更多的临时变量分配到同一个寄存器
  - 不改变程序的行为

寄存器分配

- 寄存器分配实例
  - 假设*a*和*e*在使用后死亡
  - 变量*a*对应的寄存器在第二个语句之后会被重用
  - 变量*e*对应的寄存器在第三个语句之后会被重用
  - 可以将*a*、*e*和*f*分配给同一个寄存器*r1*



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



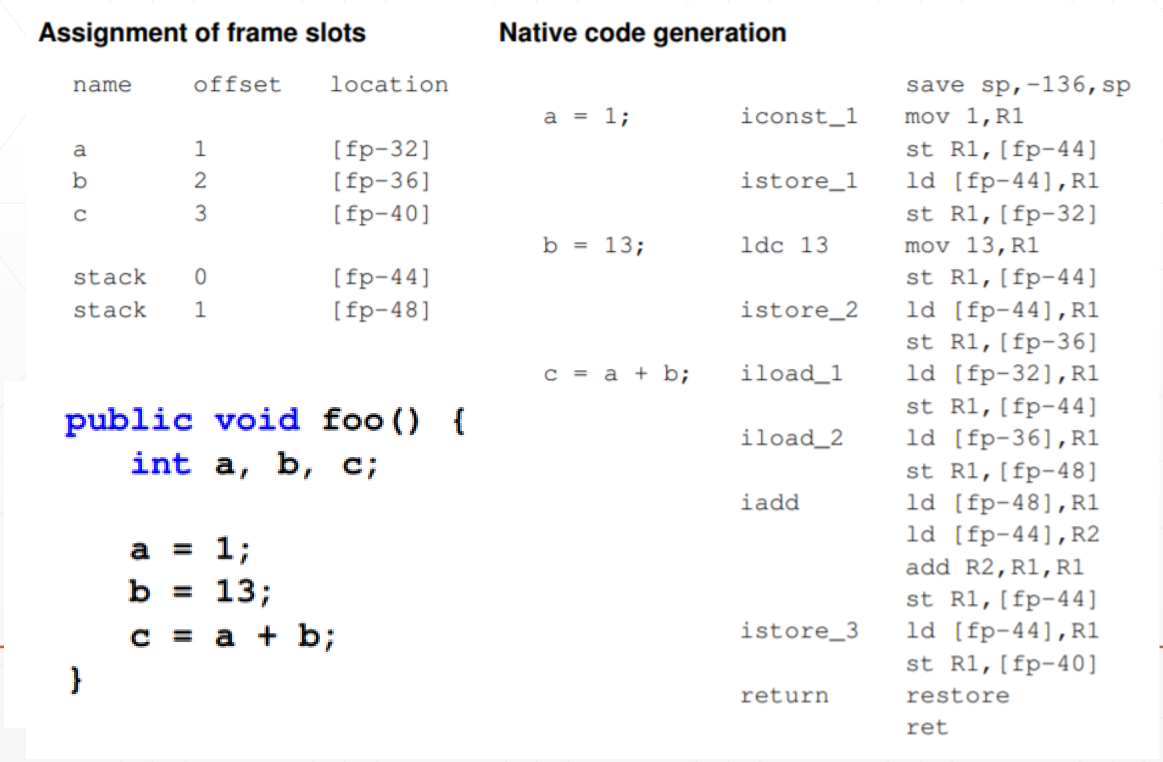
$r_1 := r_2 + r_3$

$r_1 := r_1 + r_4$

$r_1 := r_1 - 1$

寄存器分配: Naïve Allocation

- Naïve Allocation = No Allocation
  - 每次使用时从内存加载，使用完写回
  - 生成大量的访存操作指令



Assignment of frame slots

```
name      offset    location

a         1         [fp-32]
b         2         [fp-36]
c         3         [fp-40]

stack     0         [fp-44]
stack     1         [fp-48]
```

```java
public void foo() {
    int a, b, c;

    a = 1;
    b = 13;
    c = a + b;
}
```

Native code generation

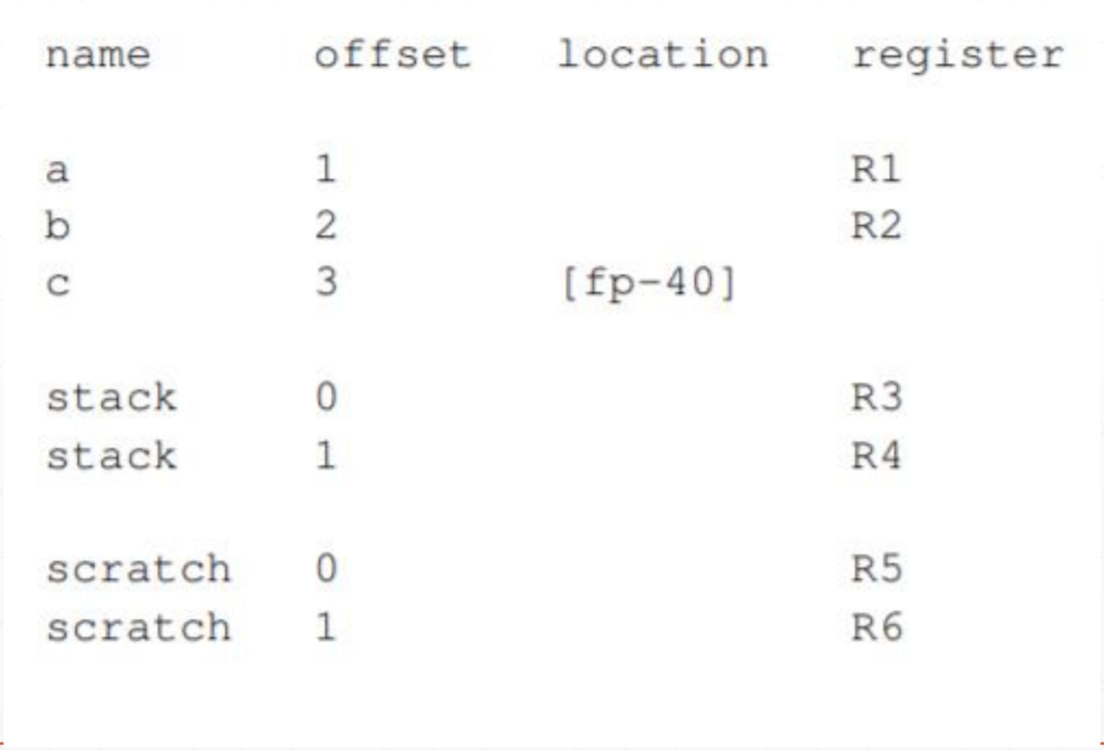
```
                               save sp,-136,sp
a = 1;         iconst_1        mov 1,R1
                               st R1,[fp-44]
               istore_1        ld [fp-44],R1
                               st R1,[fp-32]
b = 13;        ldc 13          mov 13,R1
                               st R1,[fp-44]
               istore_2        ld [fp-44],R1
                               st R1,[fp-36]
c = a + b;     iload_1         ld [fp-32],R1
                               st R1,[fp-44]
               iload_2         ld [fp-36],R1
                               st R1,[fp-48]
               iadd            ld [fp-48],R1
                               ld [fp-44],R2
                               add R2,R1,R1
                               st R1,[fp-44]
               istore_3        ld [fp-44],R1
                               st R1,[fp-40]
               return          restore
                               ret
```

寄存器分配: 固定寄存器分配

- 将m个寄存器分配给m个局部变量
- 将n个寄存器分配给n个堆栈位置
- 分配k个临时计算的寄存器
- 将其他的保留在内存中

寄存器分配: Fixed Register Allocation

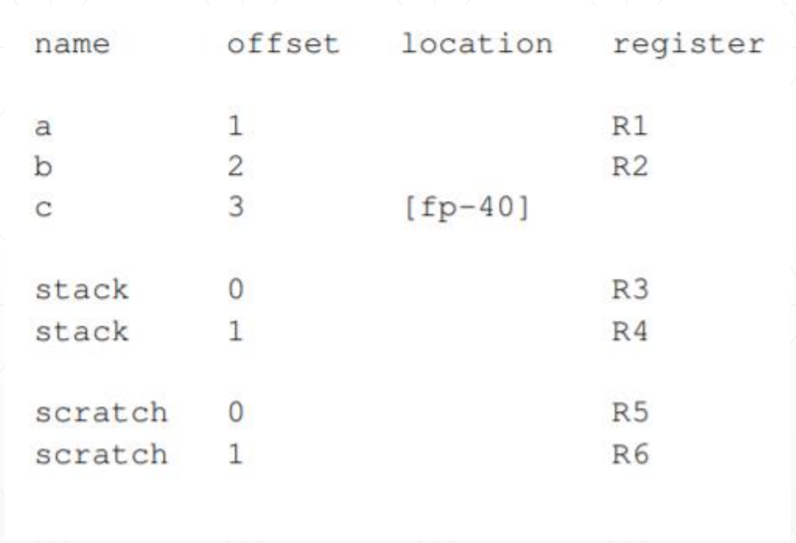
- 假设有6个寄存器（m=n=k=2），寄存器分配方案如下



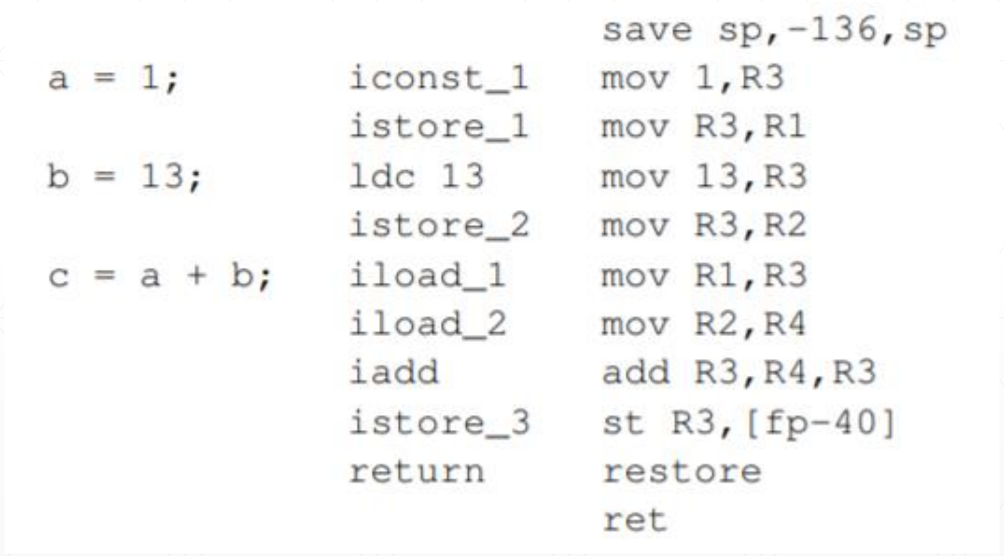
name	offset	location	register
a	1		R1
b	2		R2
c	3	[fp-40]	
stack	0		R3
stack	1		R4
scratch	0		R5
scratch	1		R6

寄存器分配: 固定寄存器分配

- 相对于No Allocation的改进方案



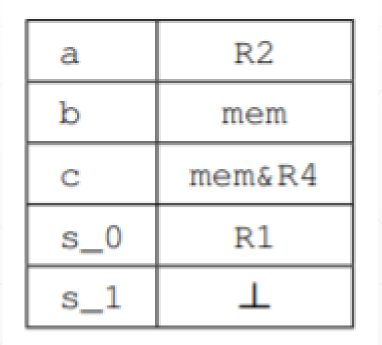
name	offset	location	register
a	1		R1
b	2		R2
c	3	[fp-40]	
stack	0		R3
stack	1		R4
scratch	0		R5
scratch	1		R6



```
                                   save sp,-136,sp
a = 1;           iconst_1          mov 1,R3
                 istore_1          mov R3,R1
b = 13;          ldc 13            mov 13,R3
                 istore_2          mov R3,R2
c = a + b;       iload_1           mov R1,R3
                 iload_2           mov R2,R4
                 iadd              add R3,R4,R3
                 istore_3          st R3,[fp-40]
                 return            restore
                                   ret
```

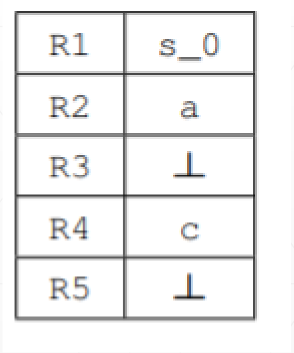
寄存器分配: 局部寄存器分配（基本块）

- 在基本块开始时加载，结束时写回
- 基本块内部按需分配
- 根据每个指令更新寄存器和变量描述符
  - 每个描述符可能的取值{⊥, mem, Ri, mem & Ri}



a	R2
b	mem
c	mem&R4
s_0	R1
s_1	⊥

变量描述符

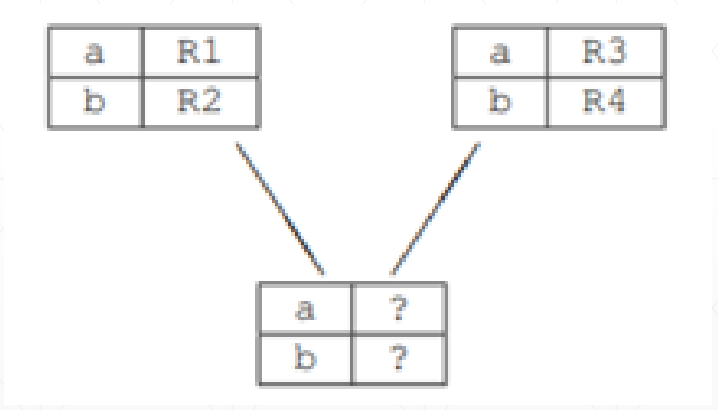


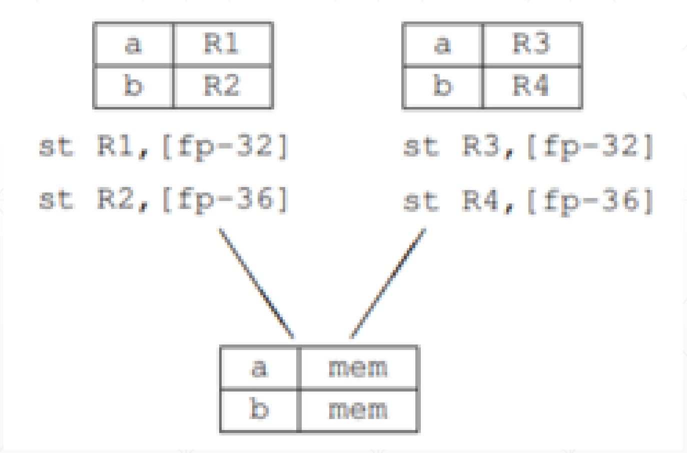
R1	s_0
R2	a
R3	⊥
R4	c
R5	⊥

寄存器描述符

寄存器分配: 局部寄存器分配（基本块）

- 基本块汇合寄存器怎么处理?
  - 每个基本块结束的时候写回

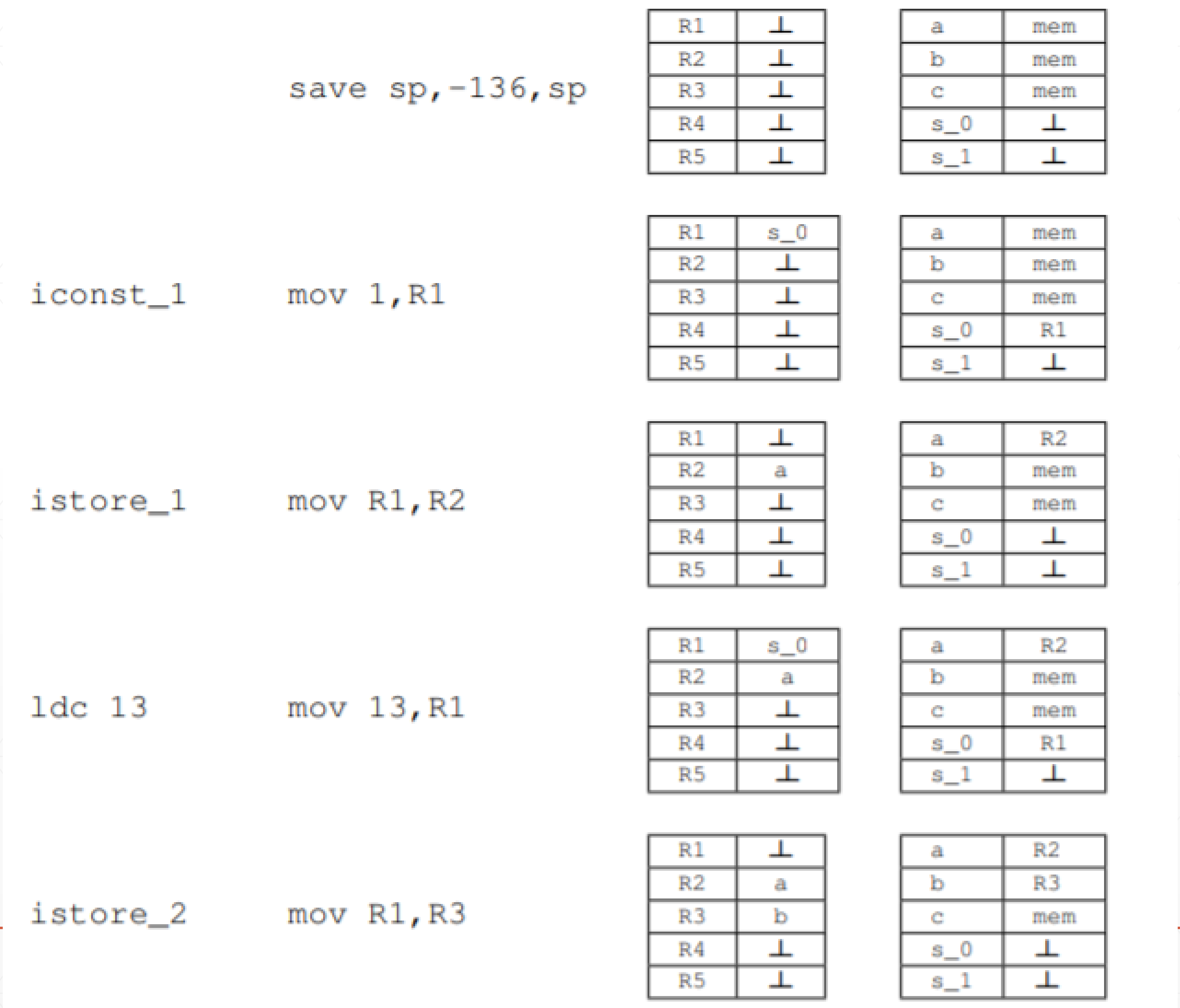




```
st R1,[fp-32]
st R2,[fp-36]
```

```
st R3,[fp-32]
st R4,[fp-36]
```

寄存器分配: 局部寄存器分配（基本块）



```
                save sp,-136,sp
```

R1	⊥	a	mem
R2	⊥	b	mem
R3	⊥	c	mem
R4	⊥	s_0	⊥
R5	⊥	s_1	⊥

```
iconst_1        mov 1,R1
```

R1	s_0	a	mem
R2	⊥	b	mem
R3	⊥	c	mem
R4	⊥	s_0	R1
R5	⊥	s_1	⊥

```
istore_1        mov R1,R2
```

R1	⊥	a	R2
R2	a	b	mem
R3	⊥	c	mem
R4	⊥	s_0	⊥
R5	⊥	s_1	⊥

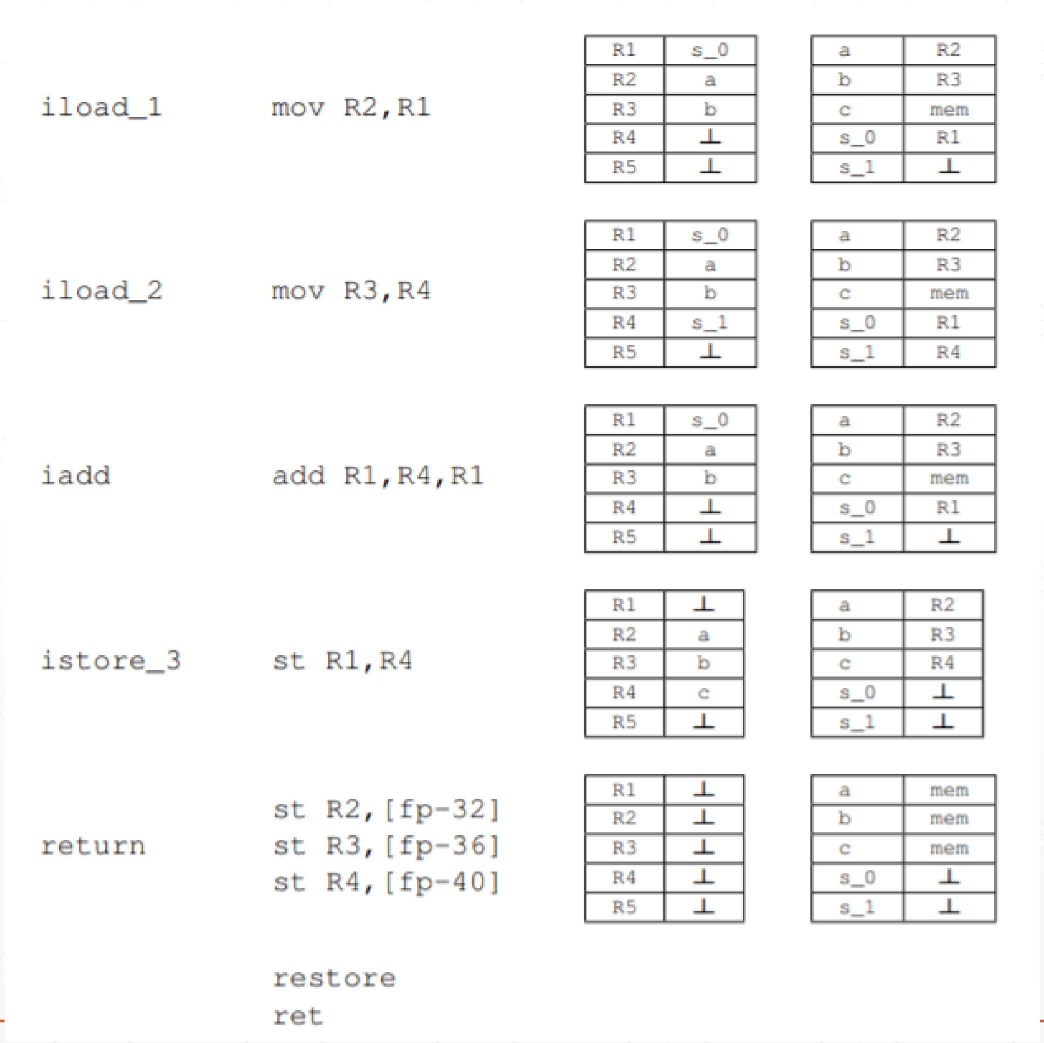
```
ldc 13          mov 13,R1
```

R1	s_0	a	R2
R2	a	b	mem
R3	⊥	c	mem
R4	⊥	s_0	R1
R5	⊥	s_1	⊥

```
istore_2        mov R1,R3
```

R1	⊥	a	R2
R2	a	b	R3
R3	b	c	mem
R4	⊥	s_0	⊥
R5	⊥	s_1	⊥

寄存器分配: 局部寄存器分配（基本块）



```
iload_1        mov R2,R1
```

R1	s_0
R2	a
R3	b
R4	⊥
R5	⊥

a	R2
b	R3
c	mem
s_0	R1
s_1	⊥

```
iload_2        mov R3,R4
```

R1	s_0
R2	a
R3	b
R4	s_1
R5	⊥

a	R2
b	R3
c	mem
s_0	R1
s_1	R4

```
iadd           add R1,R4,R1
```

R1	s_0
R2	a
R3	b
R4	⊥
R5	⊥

a	R2
b	R3
c	mem
s_0	R1
s_1	⊥

```
istore_3       st R1,R4
```

R1	⊥
R2	a
R3	b
R4	c
R5	⊥

a	R2
b	R3
c	R4
s_0	⊥
s_1	⊥

```
               st R2,[fp-32]
return         st R3,[fp-36]
               st R4,[fp-40]
```

R1	⊥
R2	⊥
R3	⊥
R4	⊥
R5	⊥

a	mem
b	mem
c	mem
s_0	⊥
s_1	⊥

```
               restore
               ret
```

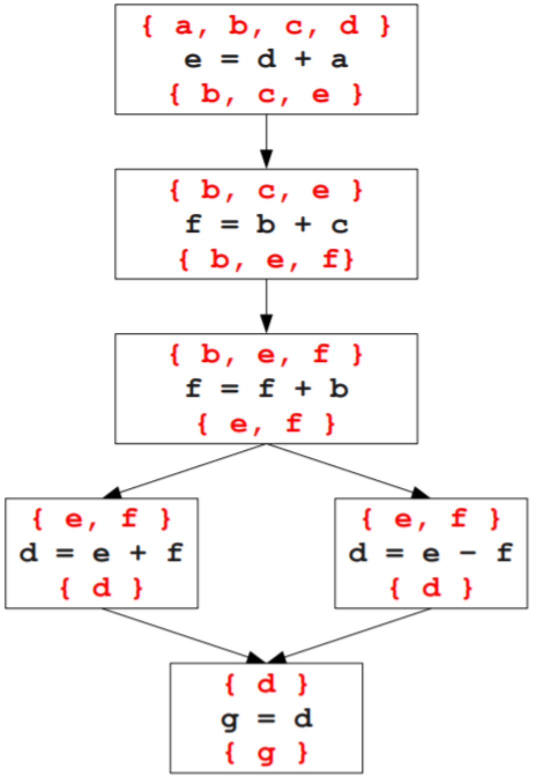
寄存器分配：线性扫描

- Live Ranges and Live Intervals
  - 一个变量在程序中某个点是活跃的是指该变量再次被赋值之前会被读
  - 一个变量的Live range是指该变量活跃的程序点集
  - 一个变量的Live interval是指该变量在中间代码层包含live range的最小区间
    - 是中间代码上定义的，不是CFG上
    - 没有live range精确，但是易于计算和使用

寄存器分配：线性扫描

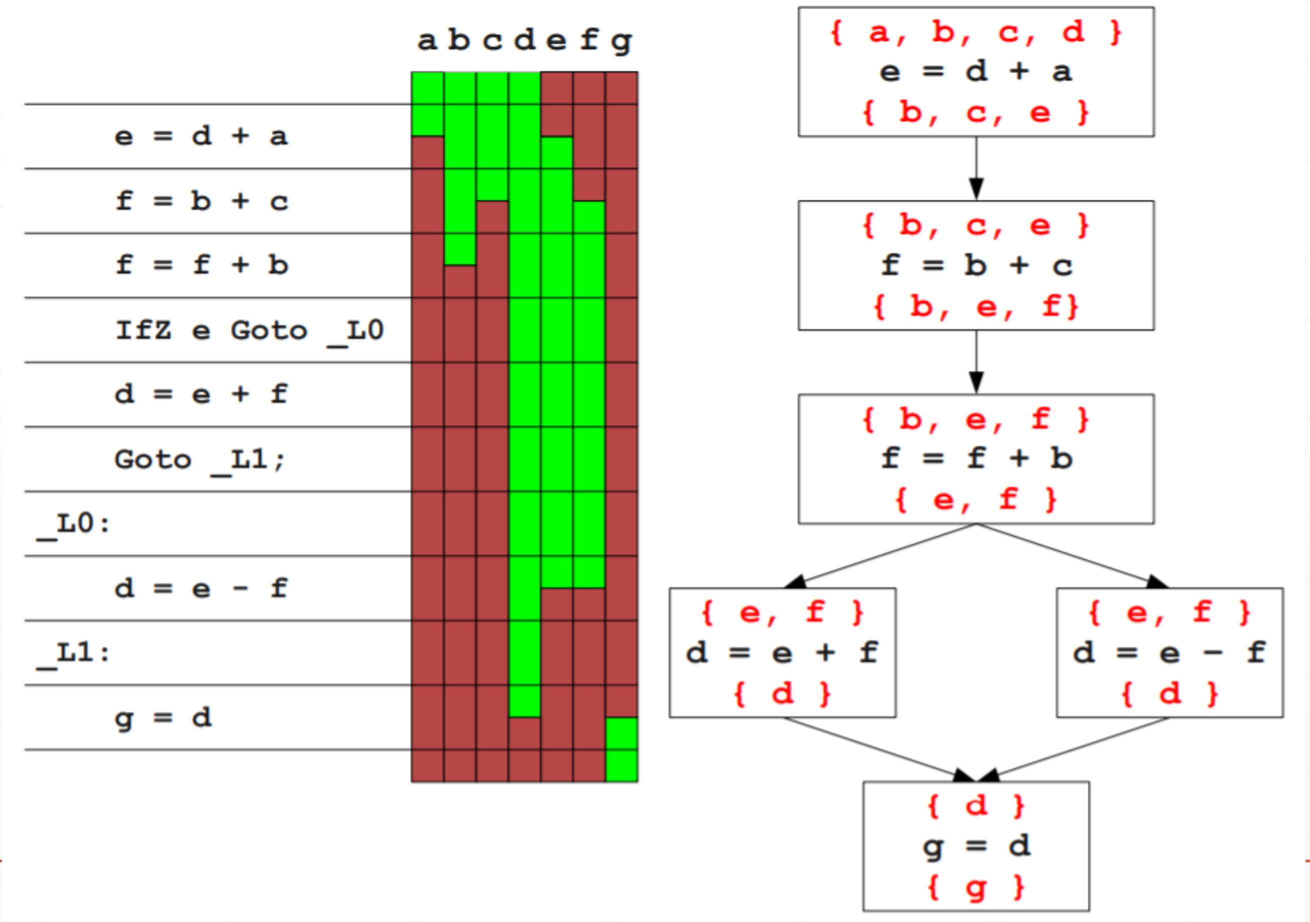
- Live Ranges and Live Intervals

```
    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
```



寄存器分配：线性扫描

- Live Ranges and Live Intervals

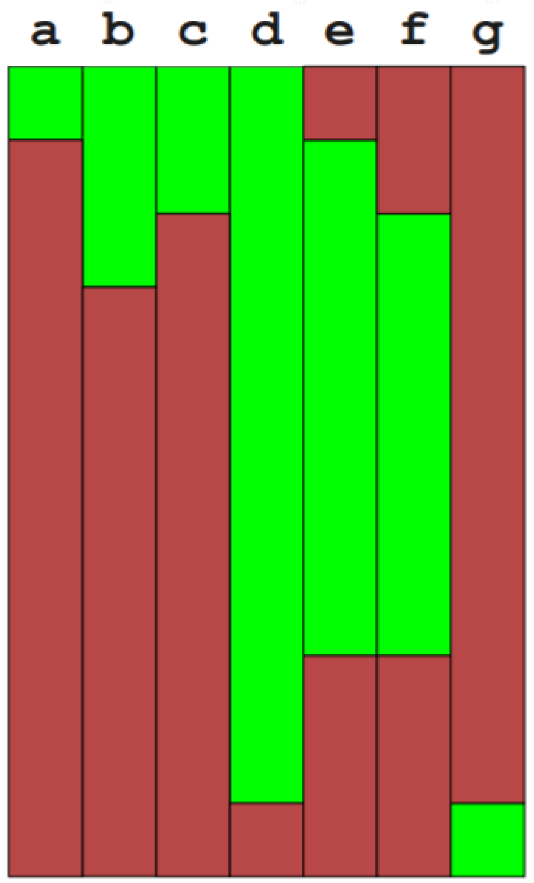


```
    e = d + a
    f = b + c
    f = f + b
    IfZ e Goto _L0
    d = e + f
    Goto _L1;
_L0:
    d = e - f
_L1:
    g = d
```

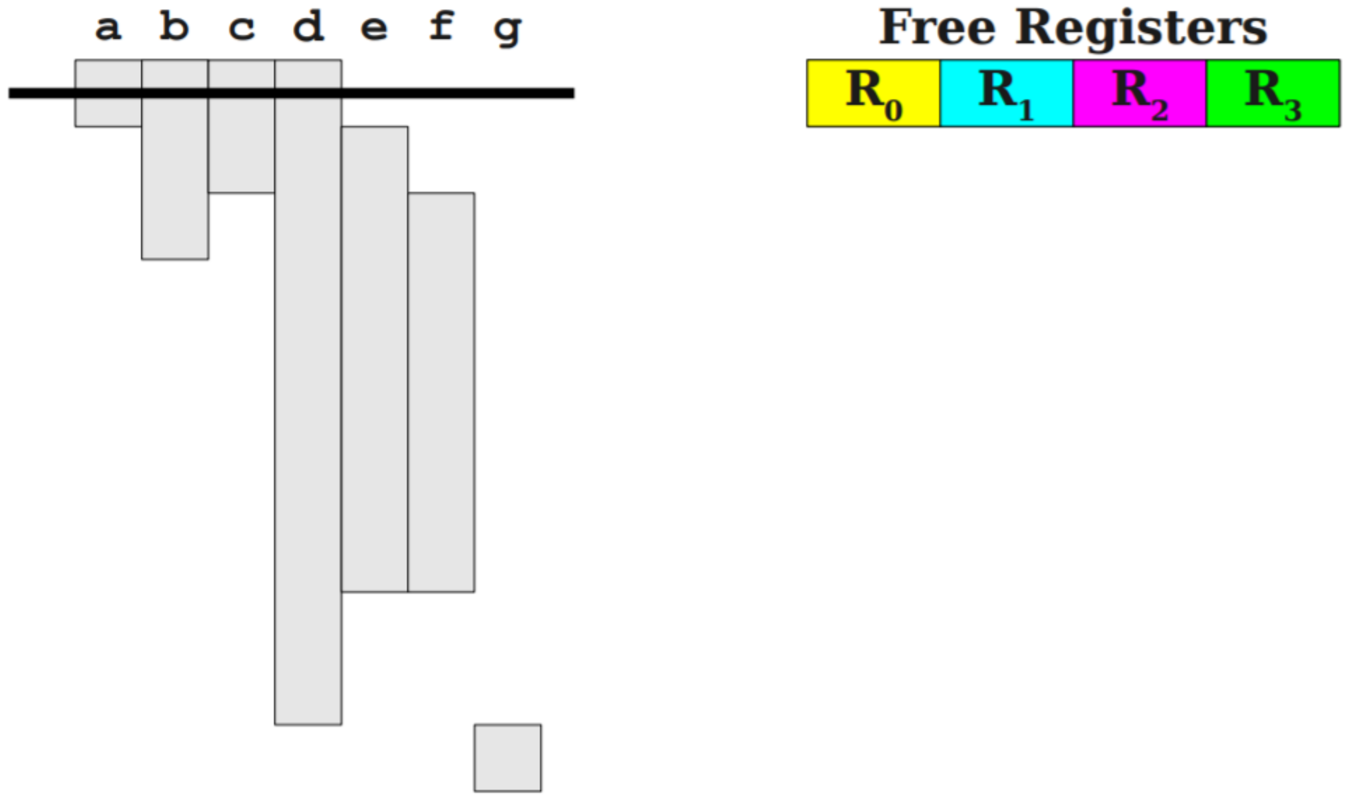
```
{ a, b, c, d }
e = d + a
{ b, c, e }

{ b, c, e }
f = b + c
{ b, e, f}

{ b, e, f }
f = f + b
{ e, f }

{ e, f }
d = e + f
{ d }

{ e, f }
d = e - f
{ d }

{ d }
g = d
{ g }
```

寄存器分配：线性扫描

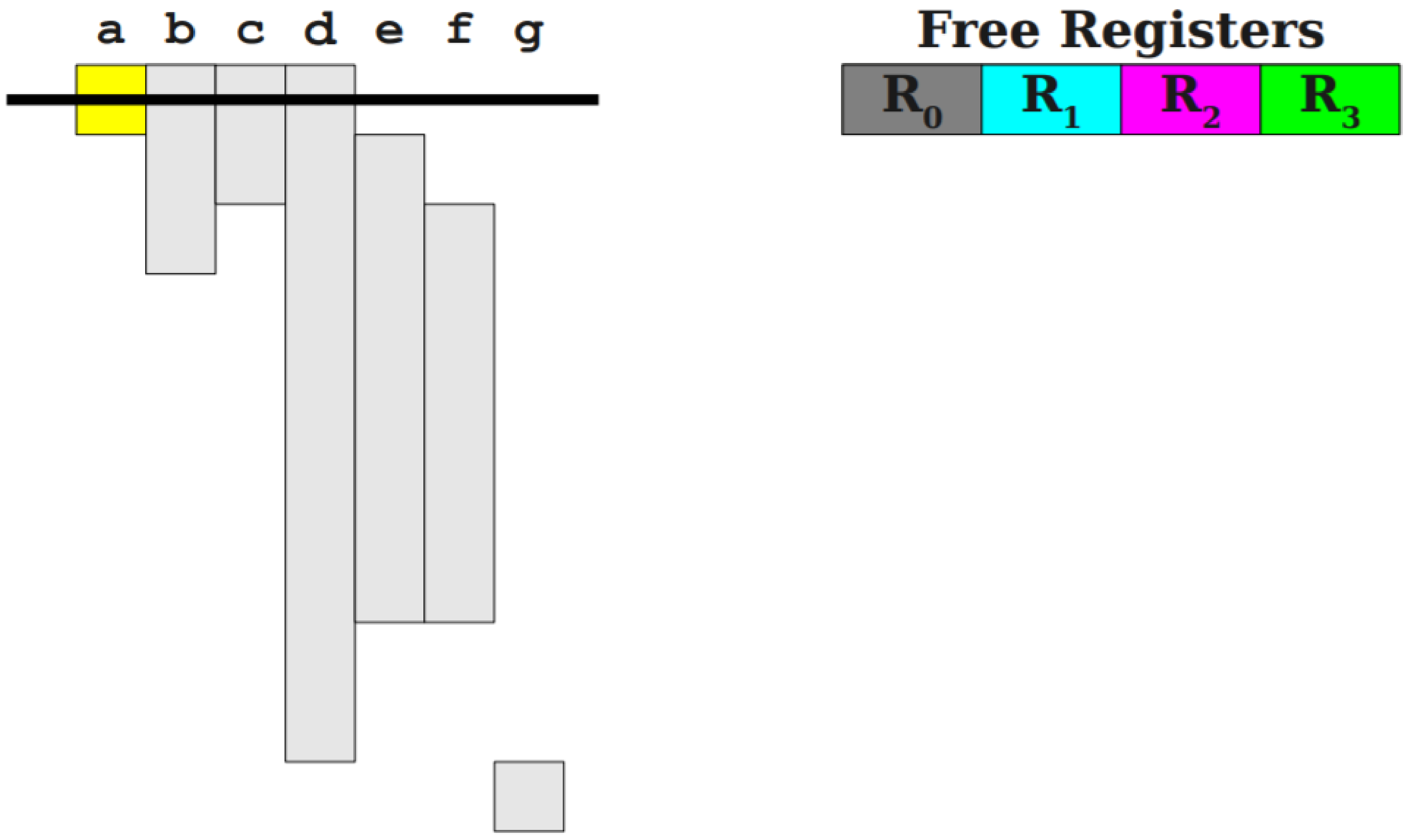
- 已知所有变量的活跃区间，可以使用简单的贪心算法完成寄存器分配
- 基本思路
  - 记录每个程序点上哪个寄存器是空闲的
  - 当遇到一个新的活跃区间的开始时，为该区间对应的变量分配一个寄存器
  - 当一个活跃区间结束时，释放对应的寄存器
  - 有可能会出现寄存器不够用的情况



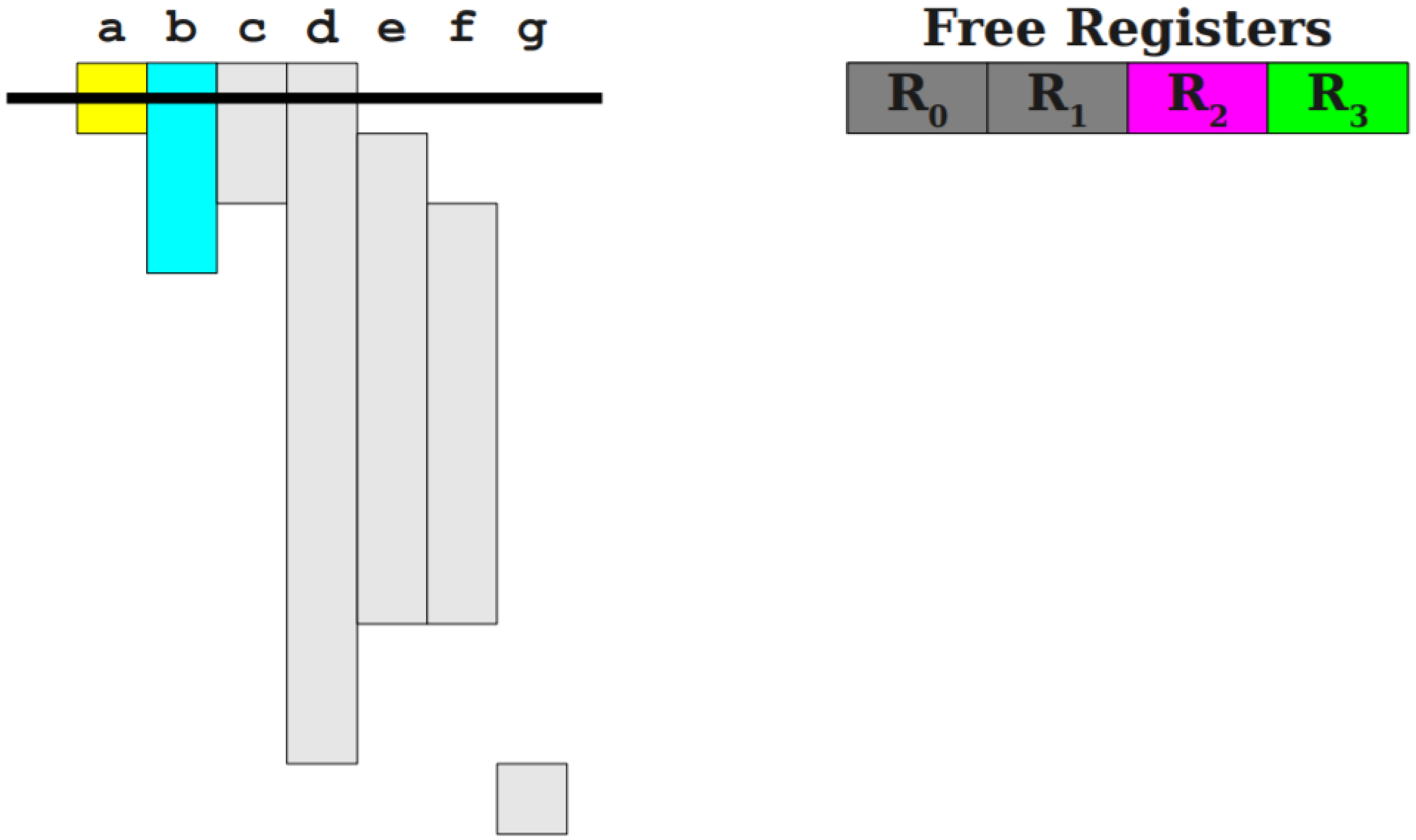
寄存器分配：线性扫描



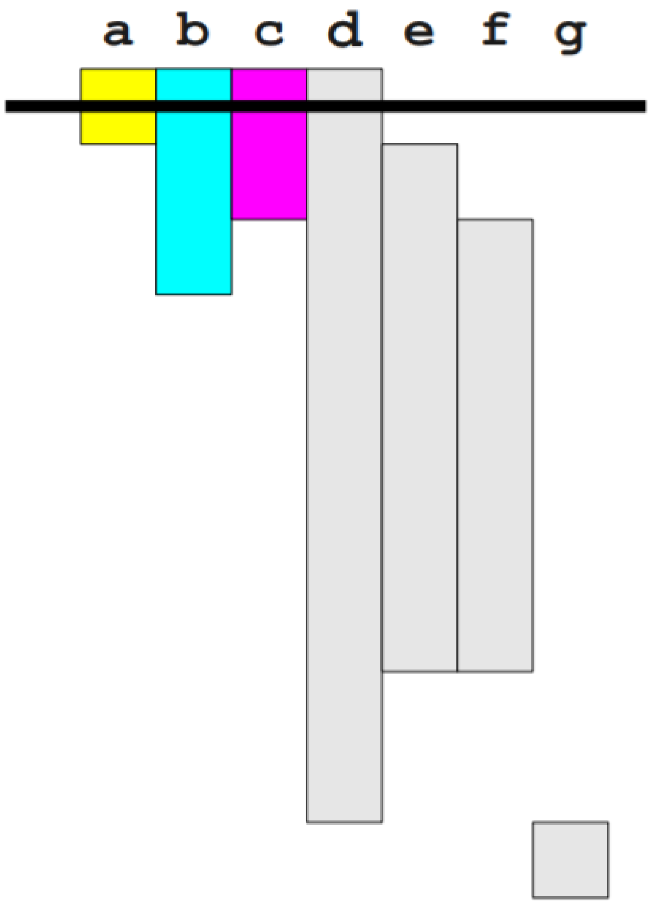
寄存器分配：线性扫描

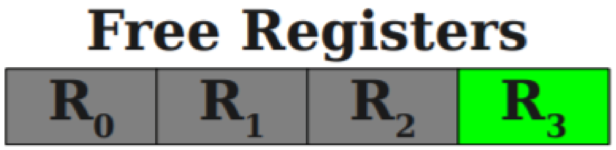


寄存器分配：线性扫描

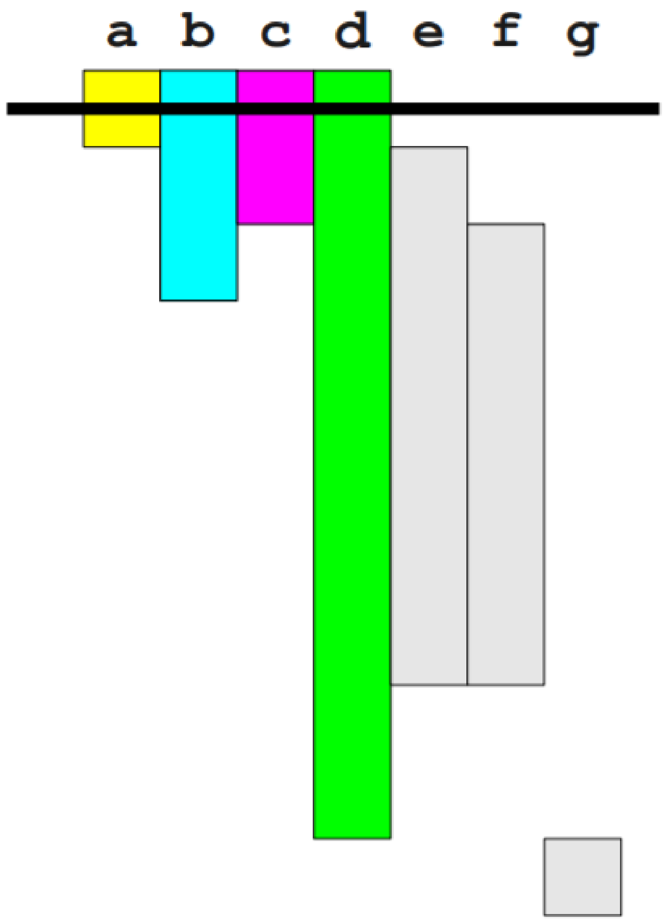


寄存器分配：线性扫描





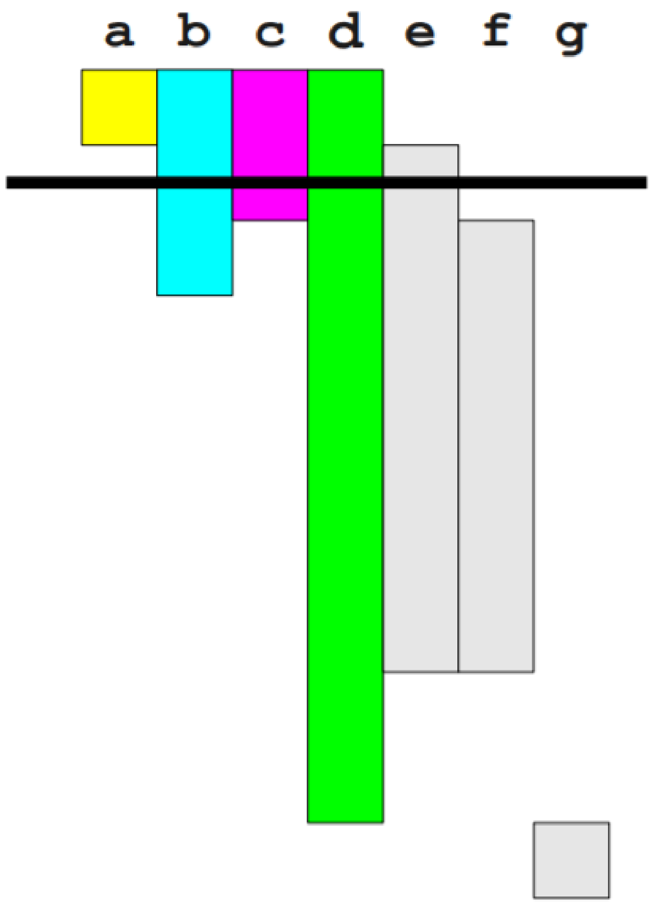
寄存器分配：线性扫描

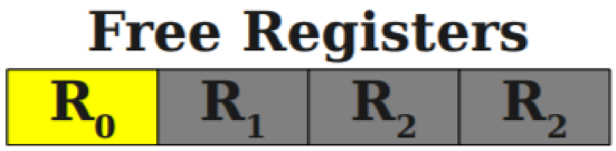


Free Registers

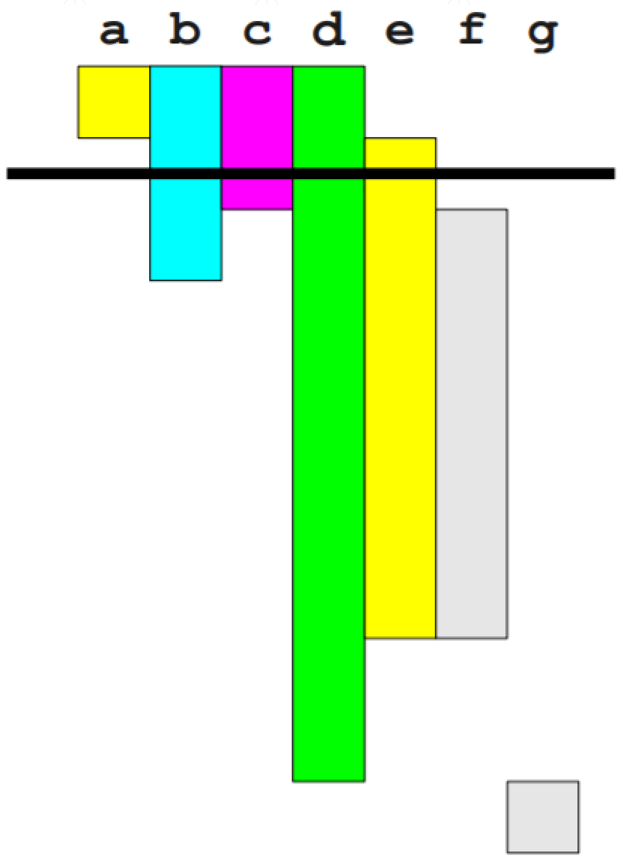
R_0	R_1	R_2	R_2

寄存器分配：线性扫描





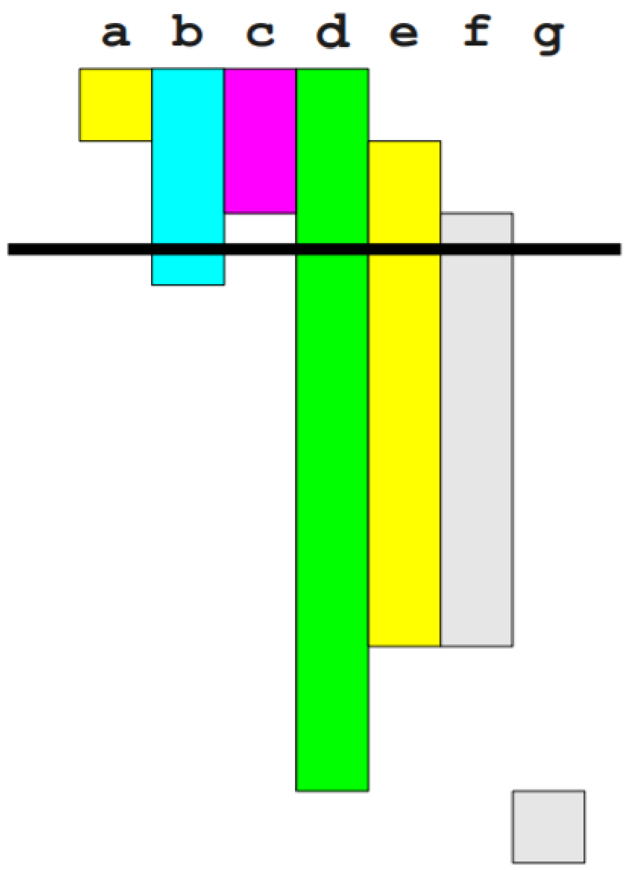
寄存器分配：线性扫描

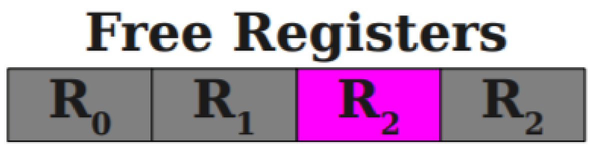


Free Registers

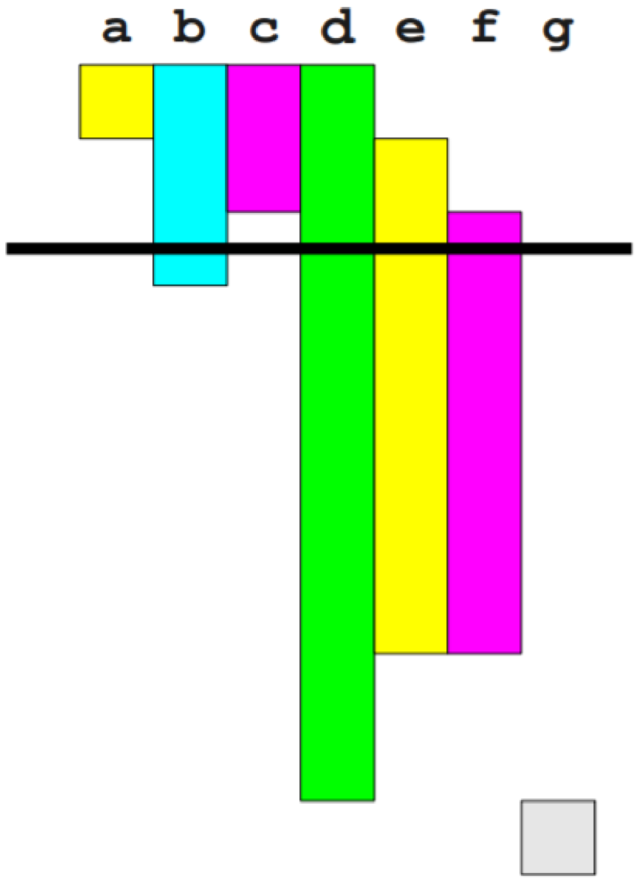
R_0	R_1	R_2	R_2

寄存器分配：线性扫描





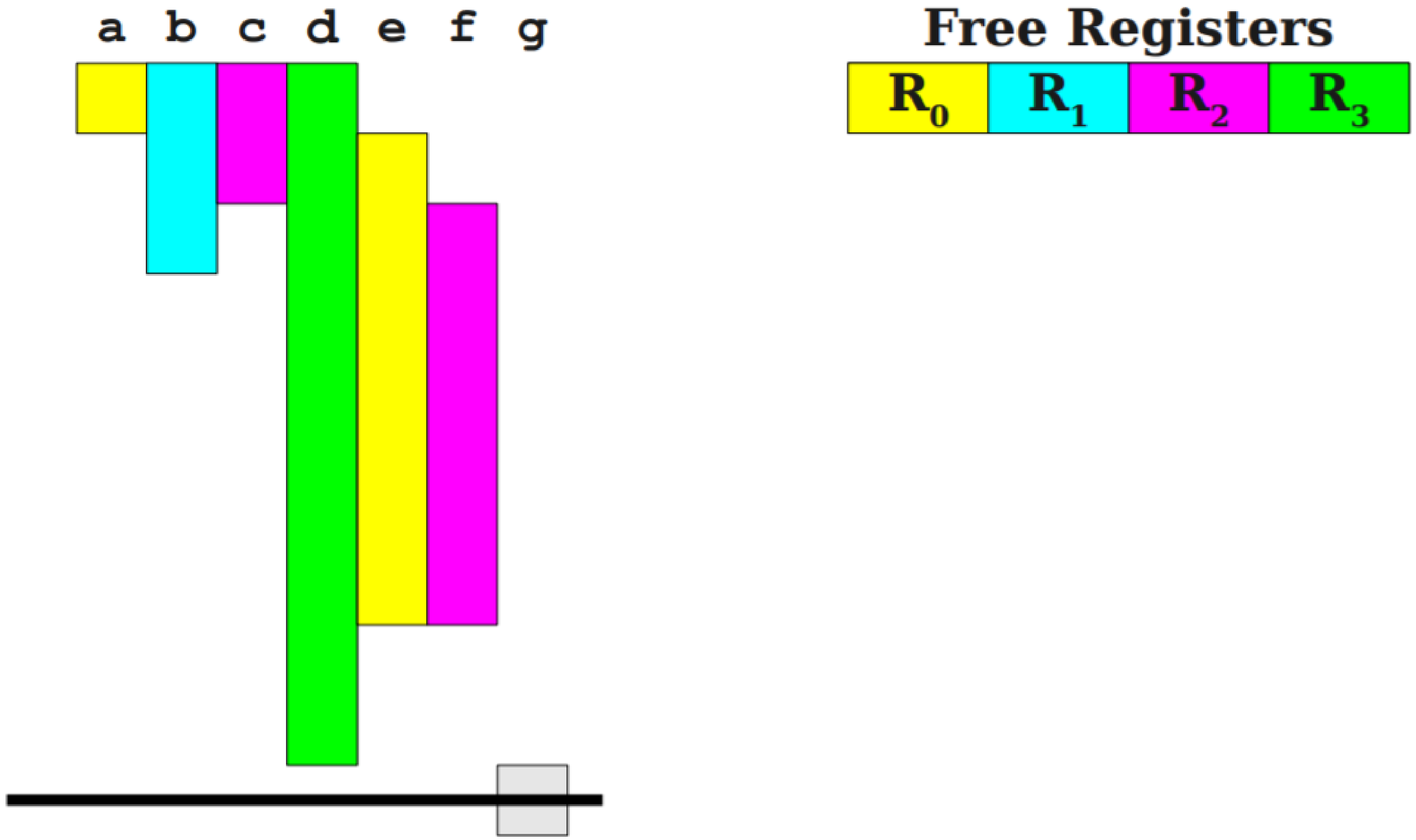
寄存器分配：线性扫描



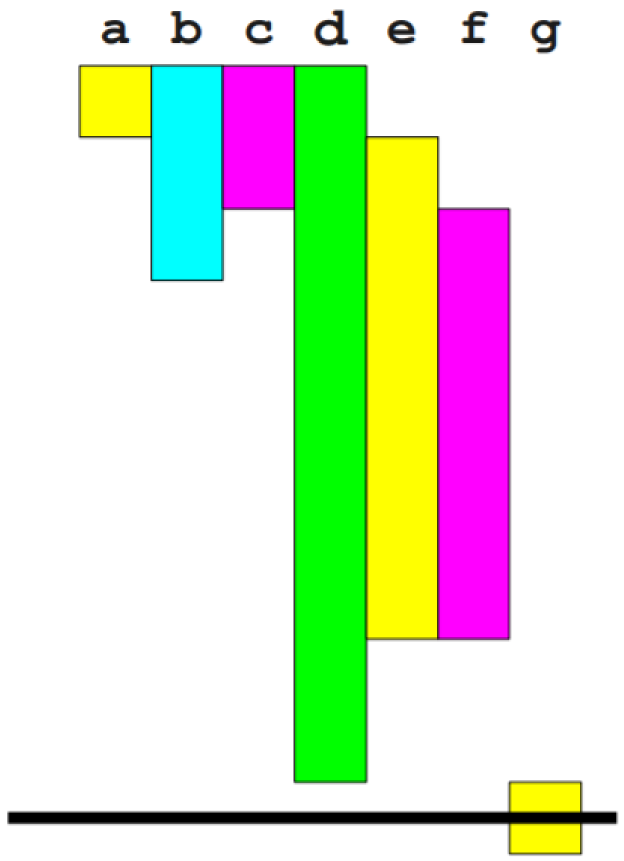
Free Registers

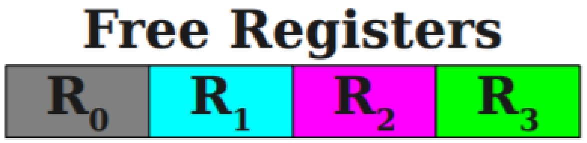
R_0	R_1	R_2	R_2

寄存器分配：线性扫描

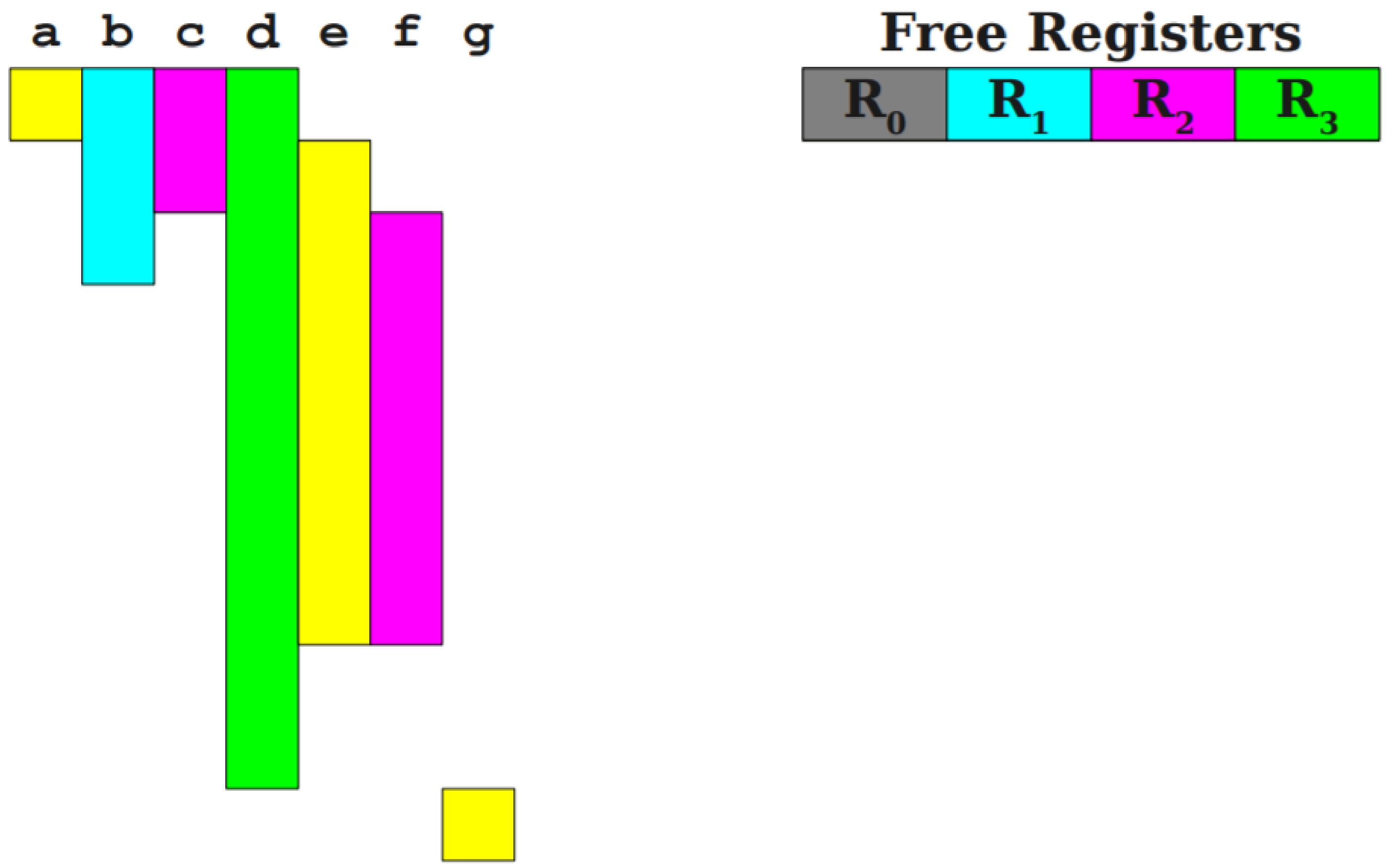


寄存器分配：线性扫描

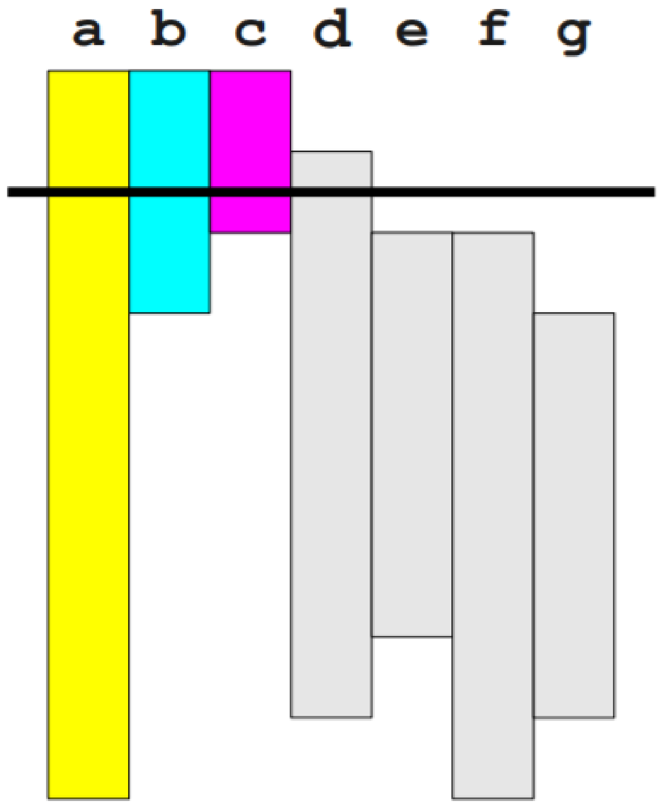


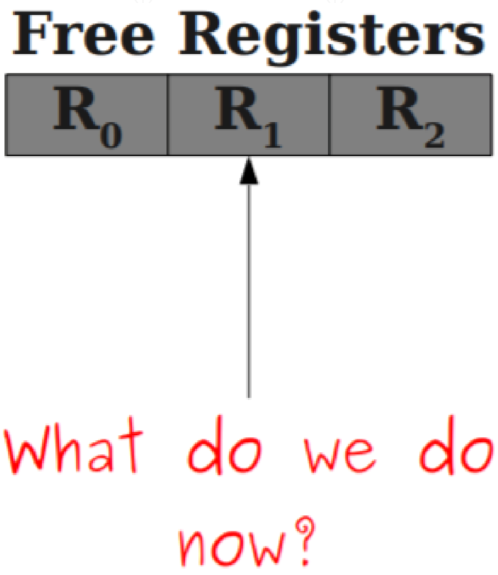


寄存器分配：线性扫描



寄存器分配：线性扫描

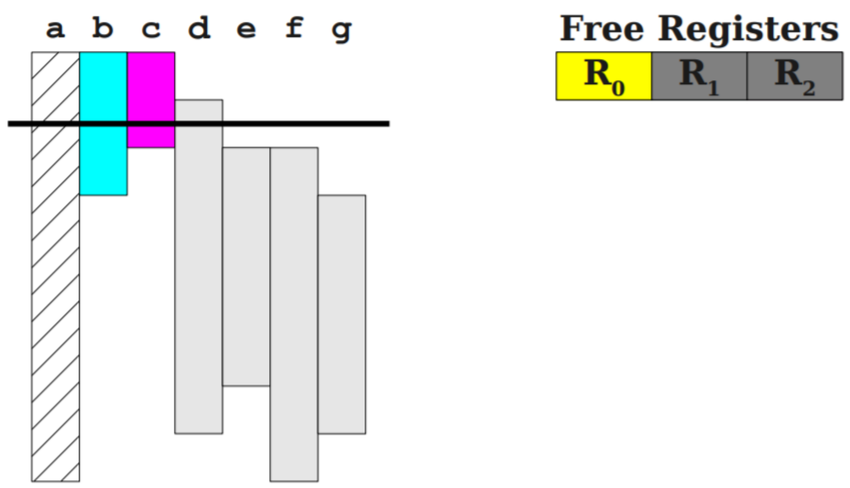


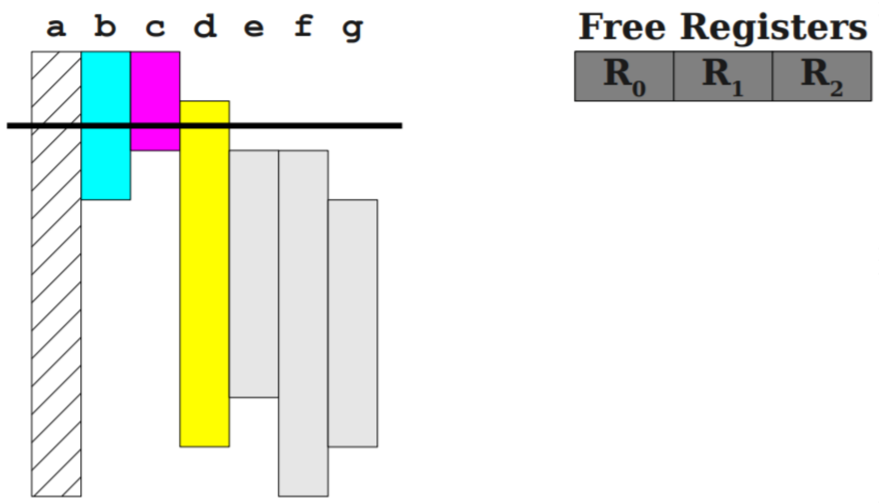


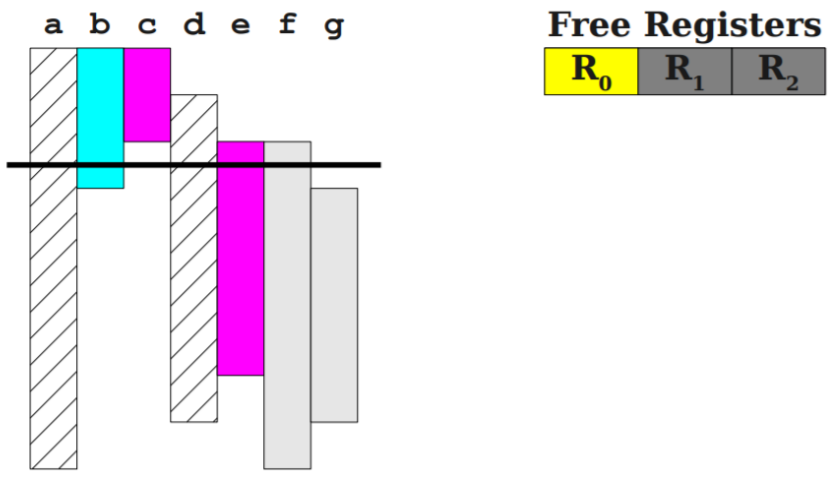
寄存器分配：线性扫描

- 寄存器Spilling
  - 如果找不到一个可用寄存器给变量v，需要将一个变量spill到内存
  - 一个变量spilled之后，存储在内存而非寄存器中
  - 基本过程
    - 选择寄存器作为spill的目标并将其内容写回内存
    - 将变量v从内存load到寄存器
    - 对变量v进行操作
    - 将变量v的值从寄存器写回内存
    - 重新load寄存器原来的变量的值

寄存器分配：线性扫描

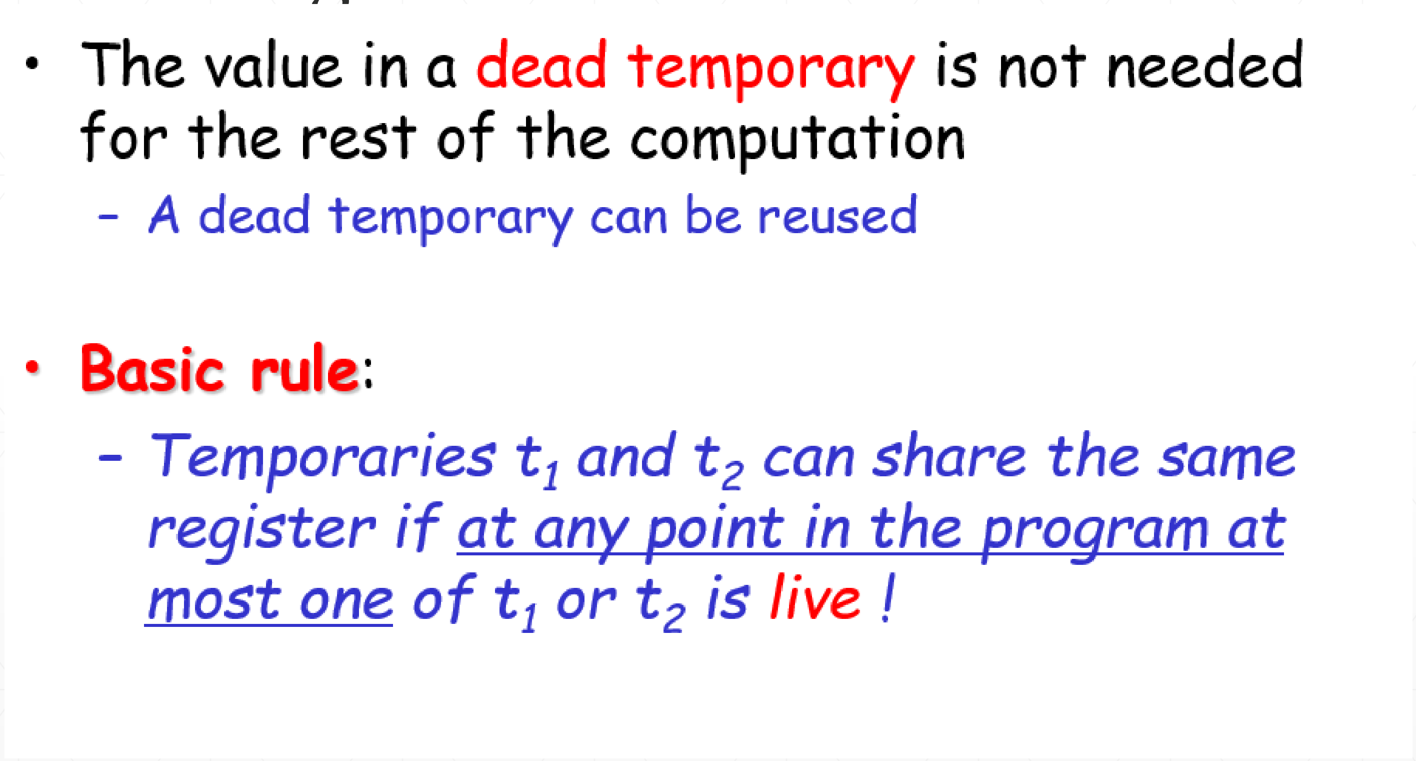

a b c d e f g
Free Registers
R_0 R_1 R_2


a b c d e f g
Free Registers
R_0 R_1 R_2


a b c d e f g
Free Registers
R_0 R_1 R_2

寄存器分配：图着色

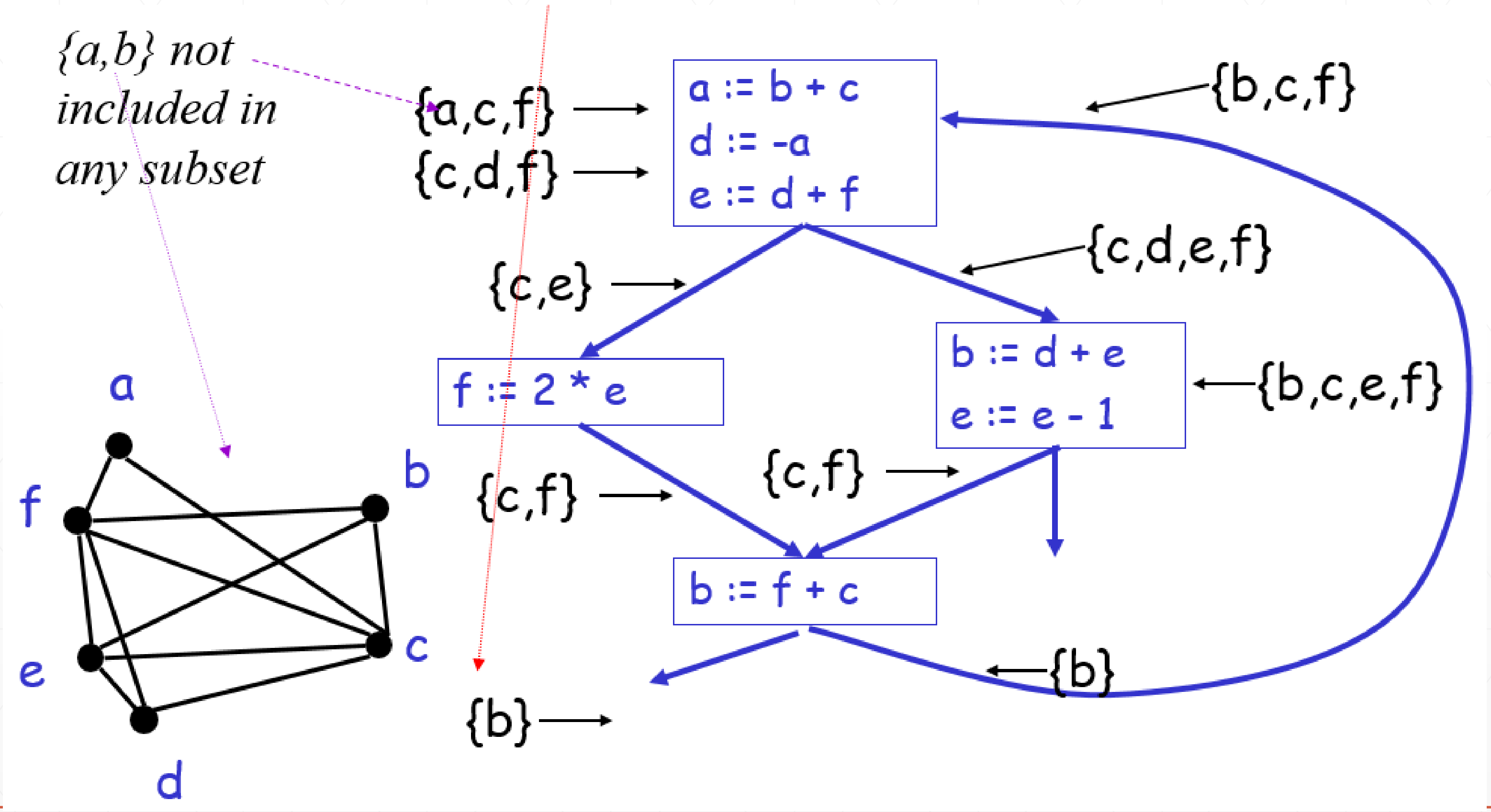
- Basic Register Allocation Idea



- The value in a dead temporary is not needed for the rest of the computation
  - A dead temporary can be reused
- **Basic rule**:
  - *Temporaries t_1 and t_2 can share the same register if at any point in the program at most one of t_1 or t_2 is live !*

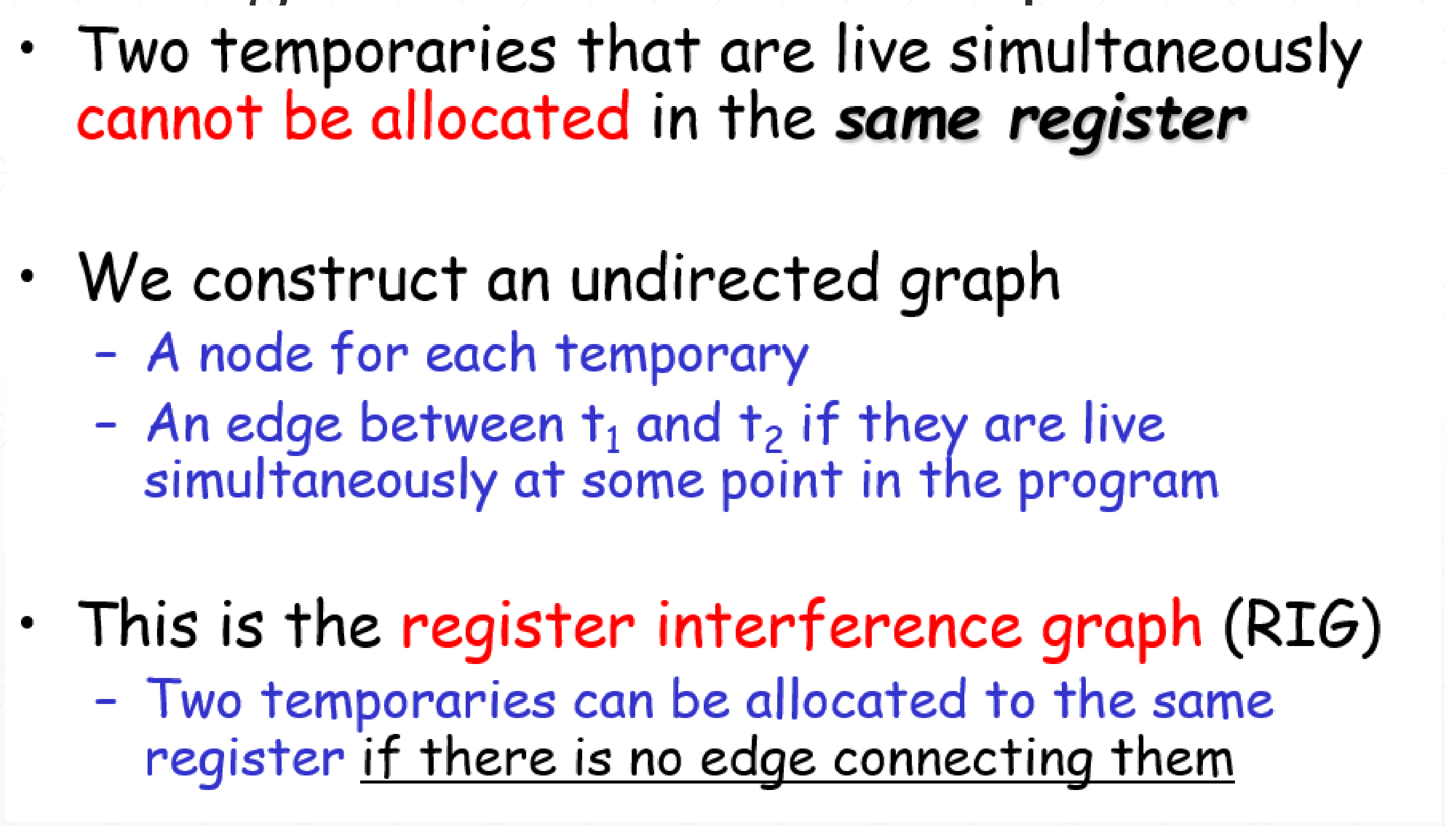
寄存器分配：图着色

- Compute live variables for each point:



寄存器分配：图着色

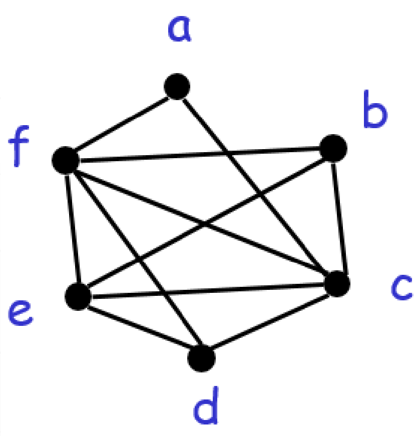
- The Register Interference Graph

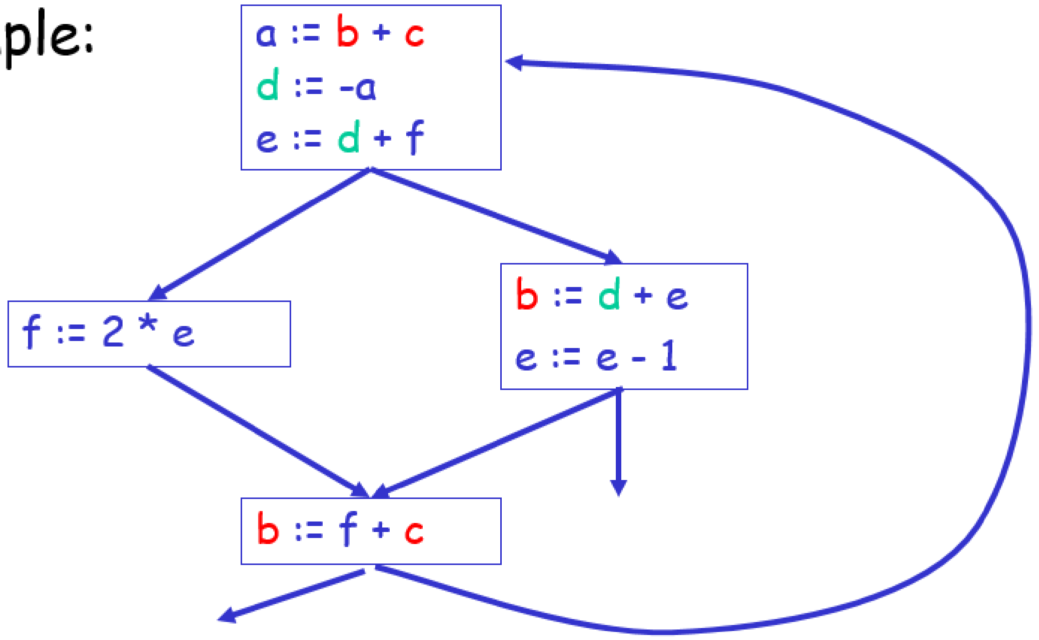


- Two temporaries that are live simultaneously cannot be allocated in the ***same register***
- We construct an undirected graph
  - A node for each temporary
  - An edge between t_1 and t_2 if they are live simultaneously at some point in the program
- This is the register interference graph (RIG)
  - Two temporaries can be allocated to the same register if there is no edge connecting them

寄存器分配：图着色

- Register Interference Graph
  - For our example:

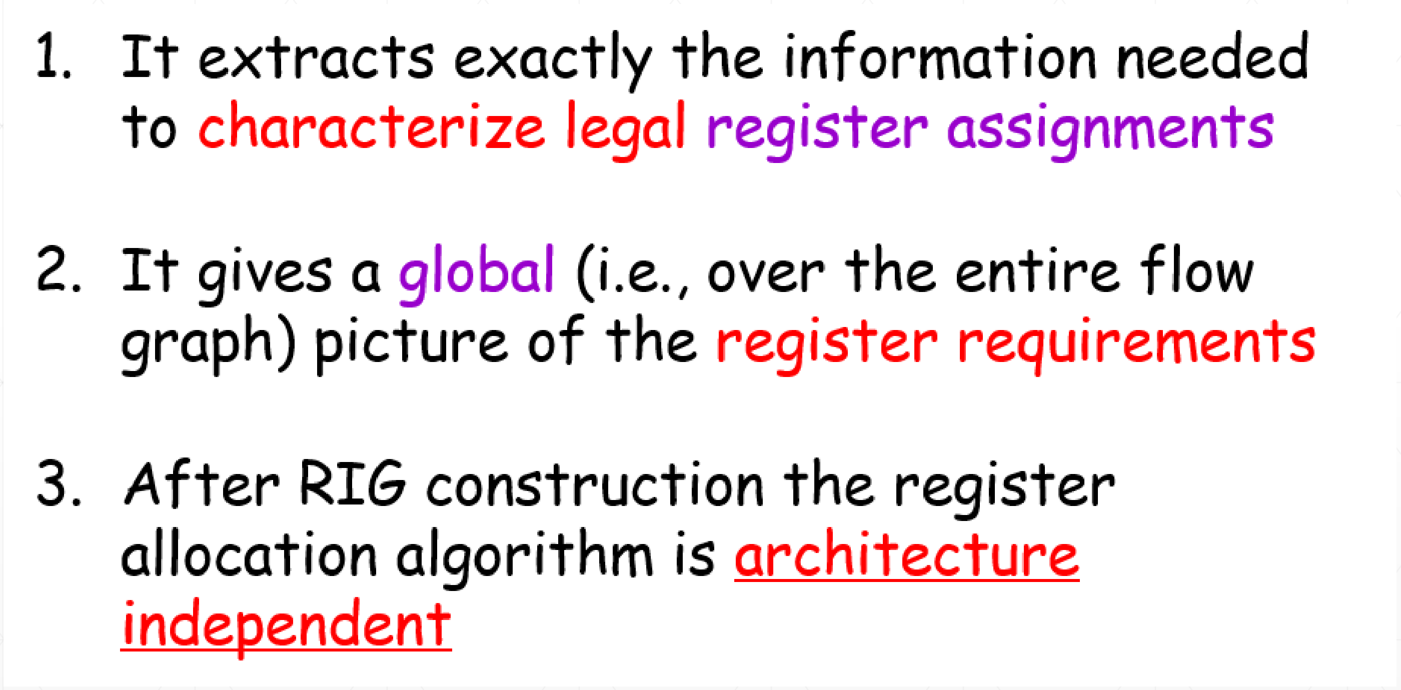




  - E.g., b and c cannot be in the same register
  - E.g., b and d can be in the same register

寄存器分配：图着色

- Register Interference Graph



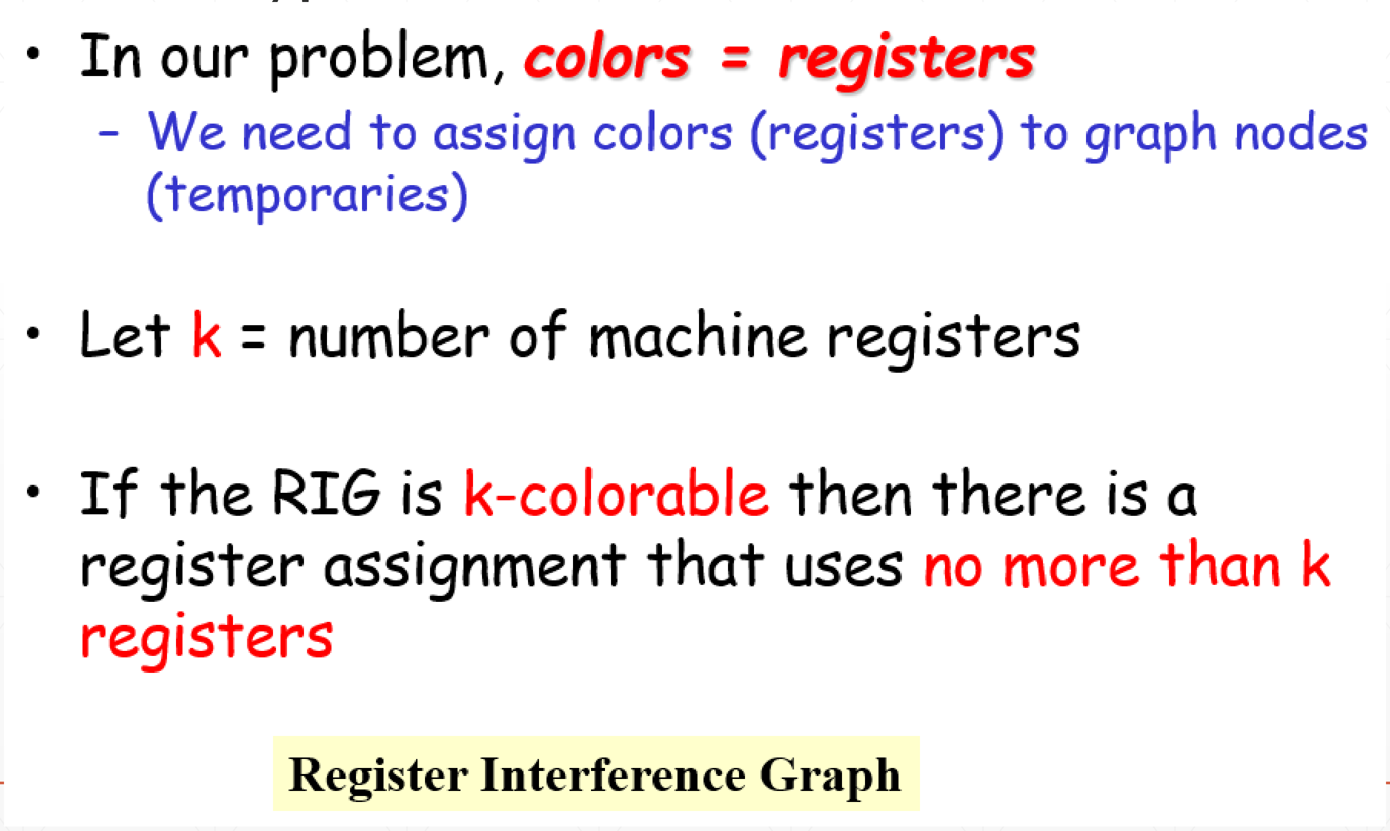
1. It extracts exactly the information needed to characterize legal register assignments
2. It gives a global (i.e., over the entire flow graph) picture of the register requirements
3. After RIG construction the register allocation algorithm is architecture independent

寄存器分配：图着色

- Graph Coloring
  - A coloring of a graph is an assignment of colors to nodes, such that nodes connected by an edge have different colors
  - A graph is k-colorable if it has a coloring with k colors

寄存器分配：图着色

- Register Allocation Through Graph Coloring

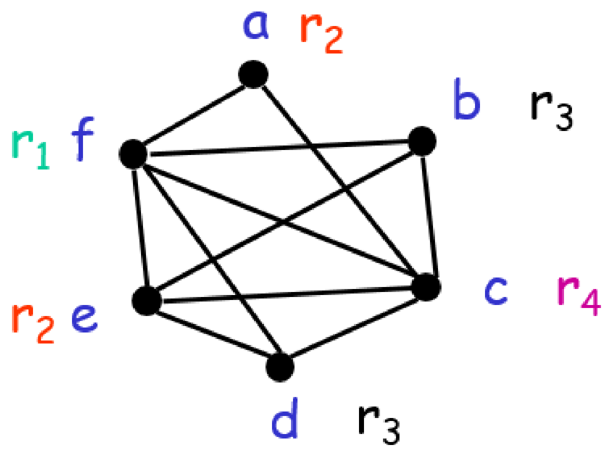


- In our problem, ***colors = registers***
  - We need to assign colors (registers) to graph nodes (temporaries)
- Let k = number of machine registers
- If the RIG is k-colorable then there is a register assignment that uses no more than k registers

Register Interference Graph

寄存器分配：图着色

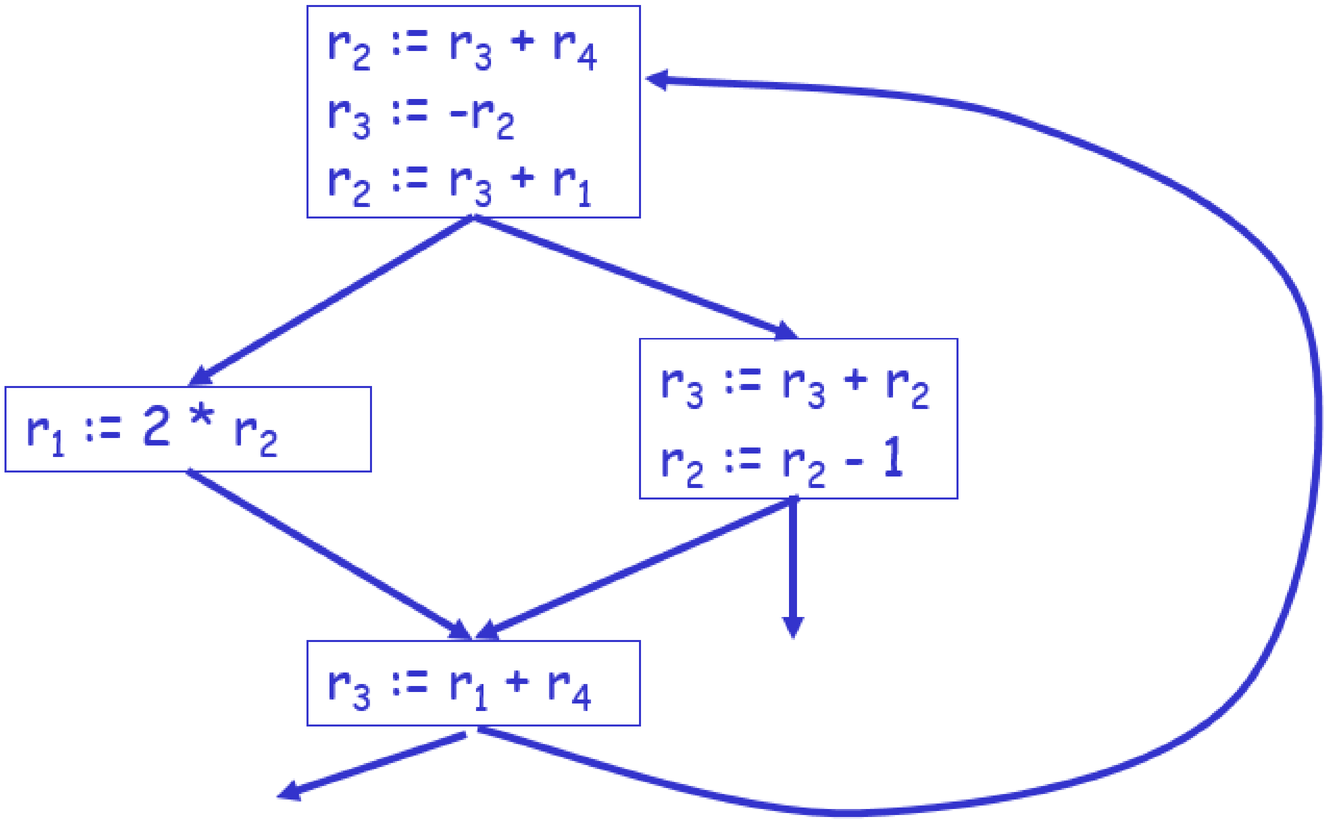
- Graph Coloring
  - Consider the example RIG



  - There is no coloring with less than 4 colors
  - There are 4-colorings of this graph

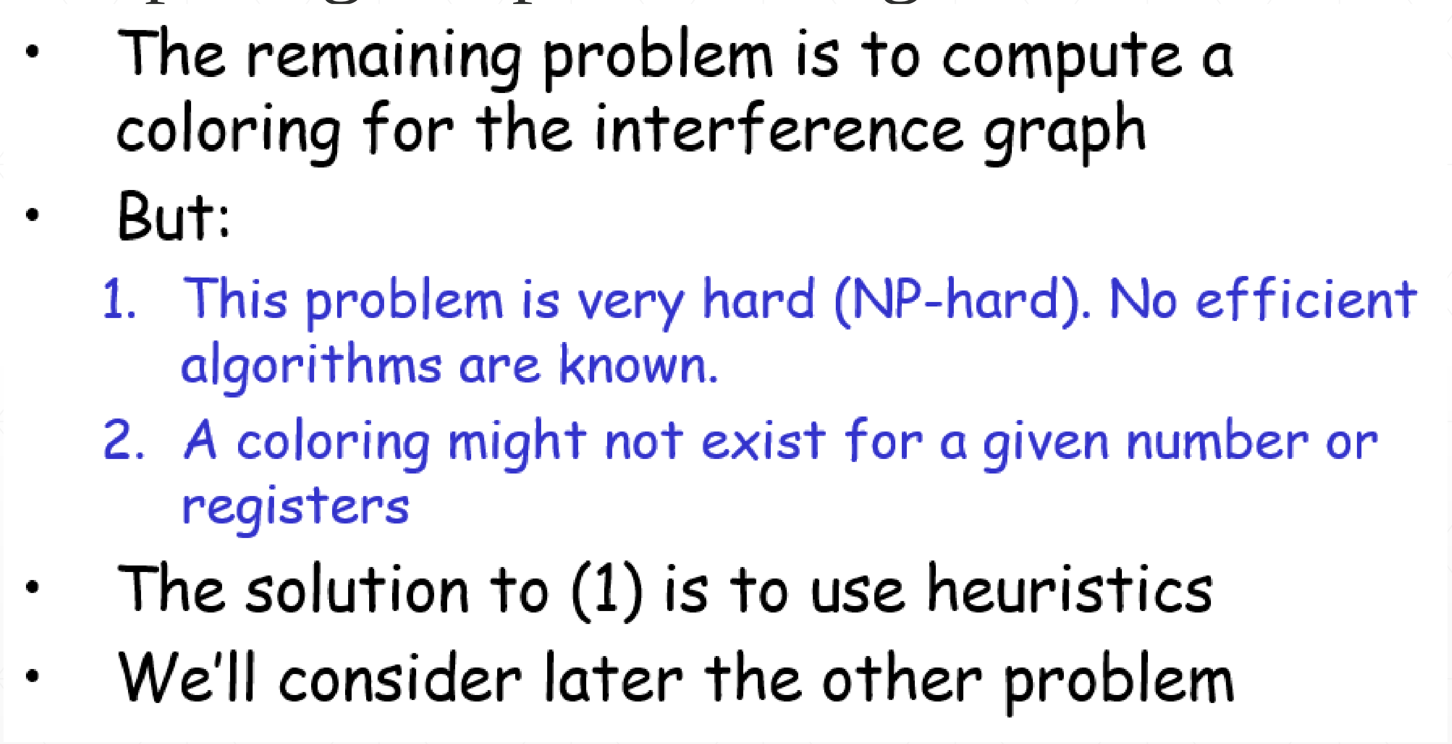
寄存器分配：图着色

- Under this coloring the code becomes



寄存器分配：图着色

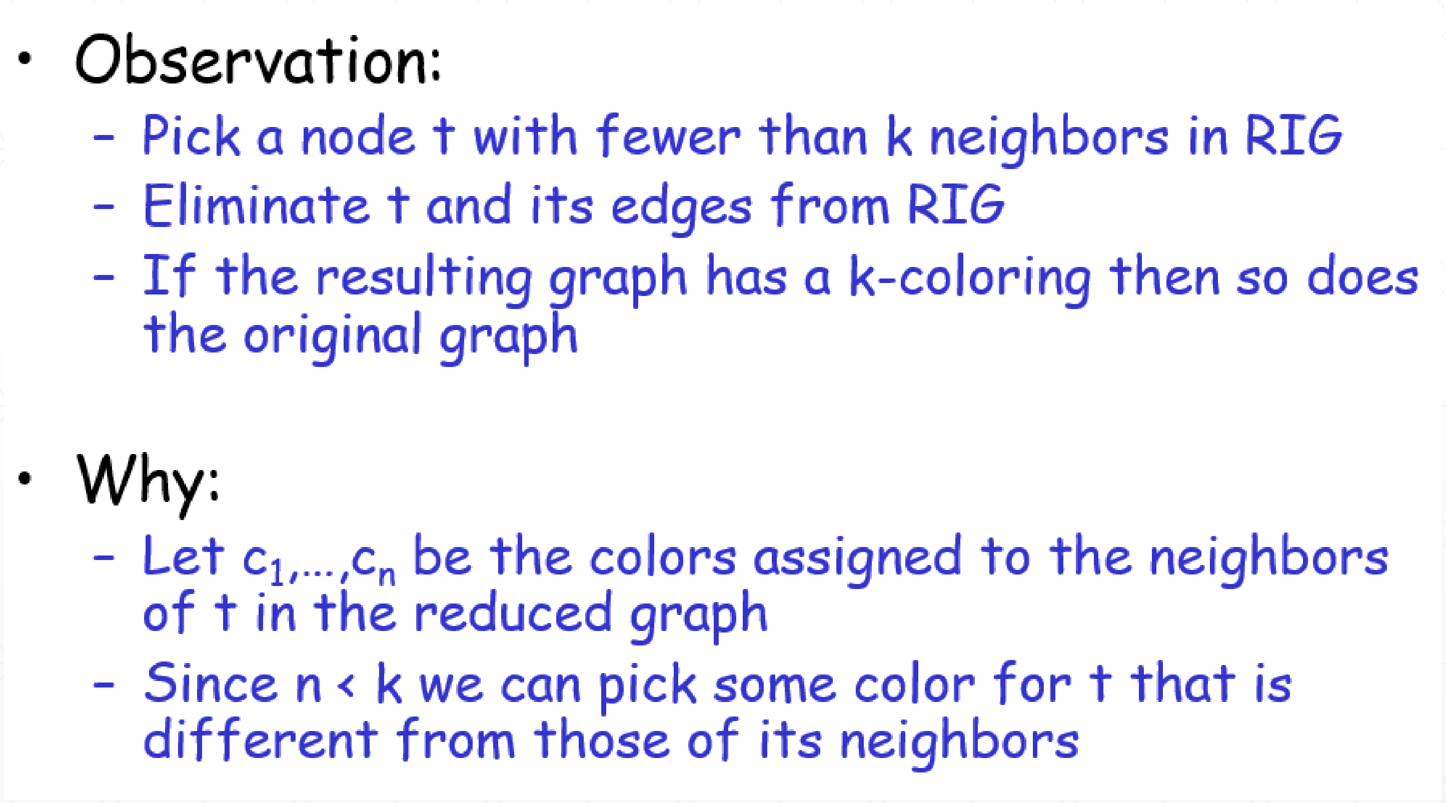
- Computing Graph Colorings



- The remaining problem is to compute a coloring for the interference graph
- But:
  1. This problem is very hard (NP-hard). No efficient algorithms are known.
  2. A coloring might not exist for a given number or registers
- The solution to (1) is to use heuristics
- We'll consider later the other problem

寄存器分配：图着色

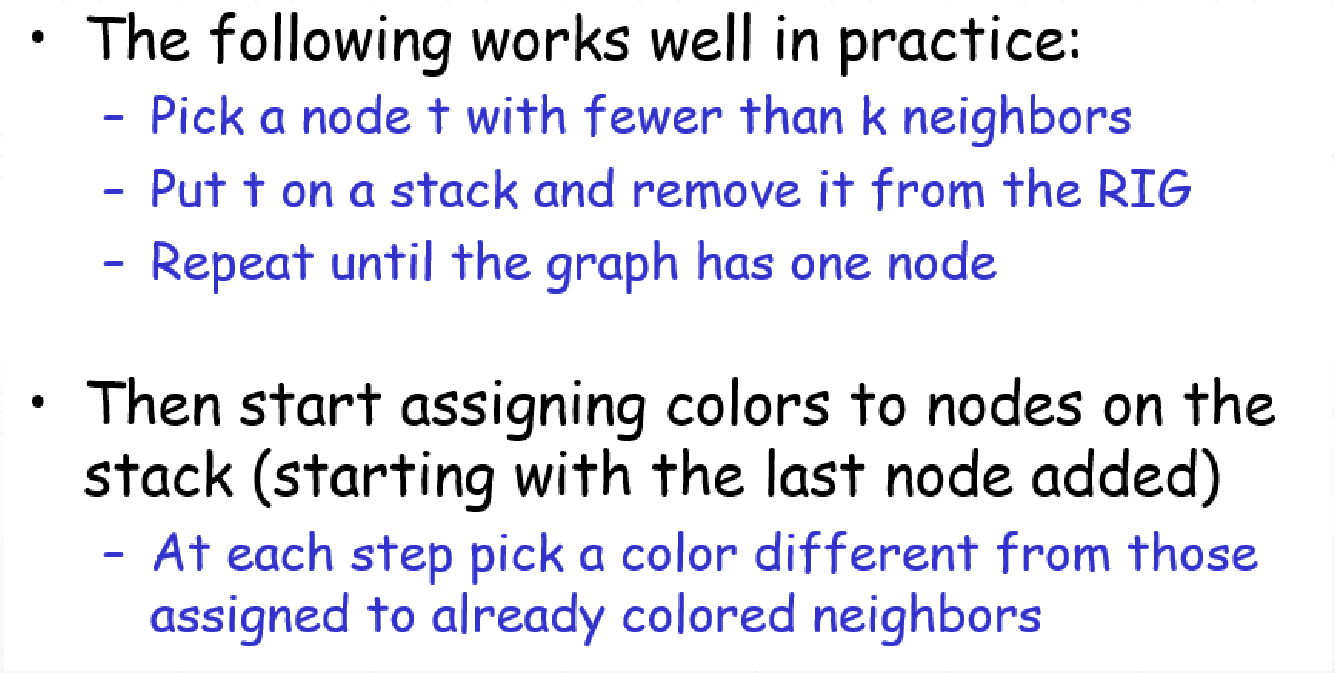
- Graph Coloring Heuristic



- Observation:
  - Pick a node t with fewer than k neighbors in RIG
  - Eliminate t and its edges from RIG
  - If the resulting graph has a k-coloring then so does the original graph
- Why:
  - Let $c_1,\ldots,c_n$ be the colors assigned to the neighbors of t in the reduced graph
  - Since n < k we can pick some color for t that is different from those of its neighbors

寄存器分配：图着色

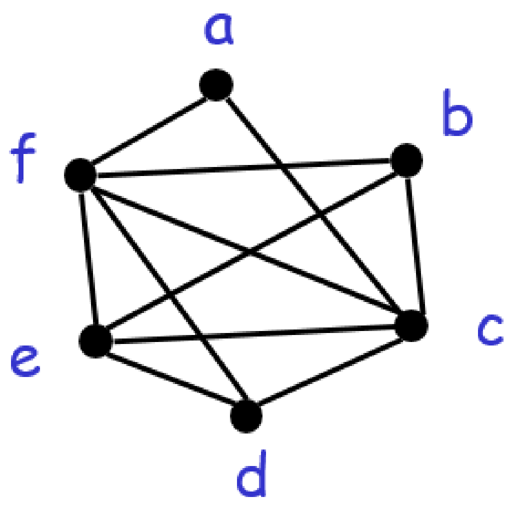
- Graph Coloring Heuristic



- The following works well in practice:
  - Pick a node t with fewer than k neighbors
  - Put t on a stack and remove it from the RIG
  - Repeat until the graph has one node
- Then start assigning colors to nodes on the stack (starting with the last node added)
  - At each step pick a color different from those assigned to already colored neighbors

寄存器分配：图着色

- Graph Coloring Example
  - Start with the RIG and with k = 4:

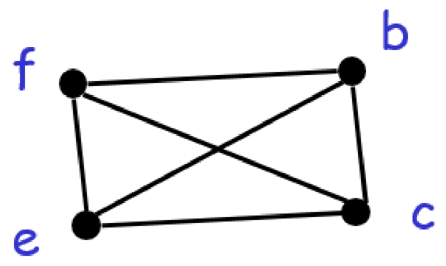


Stack: {}

  - Remove a and then d

寄存器分配：图着色

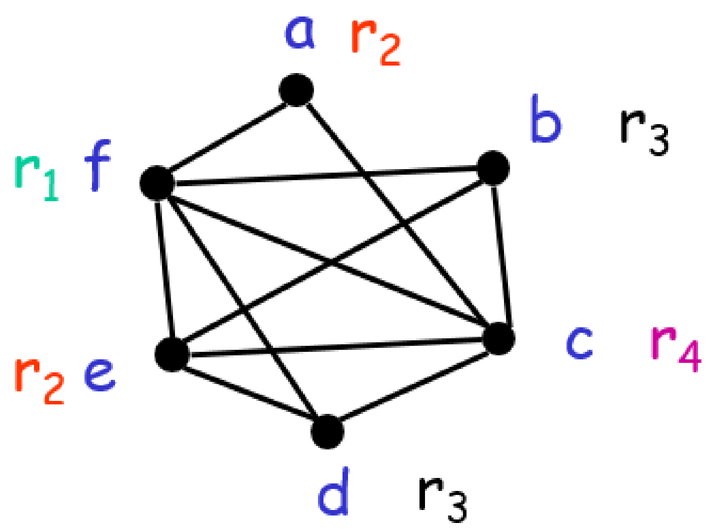
- Graph Coloring Example
  - Now all nodes have fewer than 4 neighbors and can be removed: c, b, e, f



Stack: {d, a}

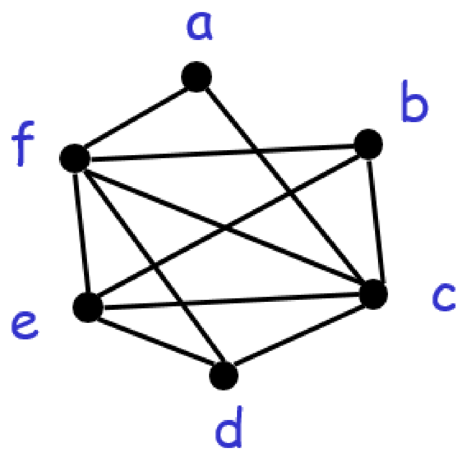
寄存器分配：图着色

- Graph Coloring Example
  - Start assigning colors to: f, e, b, c, d, a



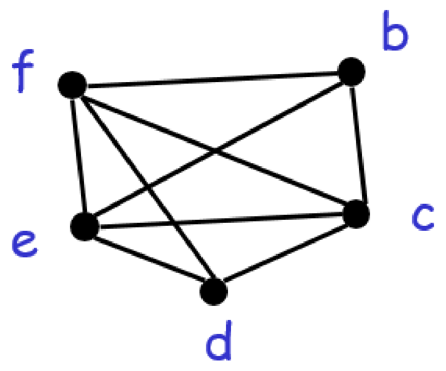
寄存器分配：图着色

- What if the Heuristic Fails?
  - What if during simplification we get to a state where all nodes have k or more neighbors ?
  - Example: try to find a 3-coloring of the RIG:



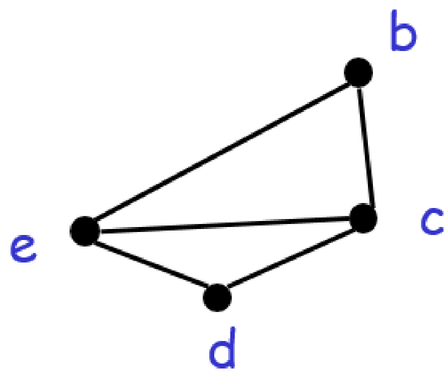
寄存器分配：图着色

- What if the Heuristic Fails?
  - Remove a and get stuck (as shown below)
  - Pick a node as a candidate for spilling
    - A spilled temporary "lives" in memory
  - Assume that f is picked as a candidate



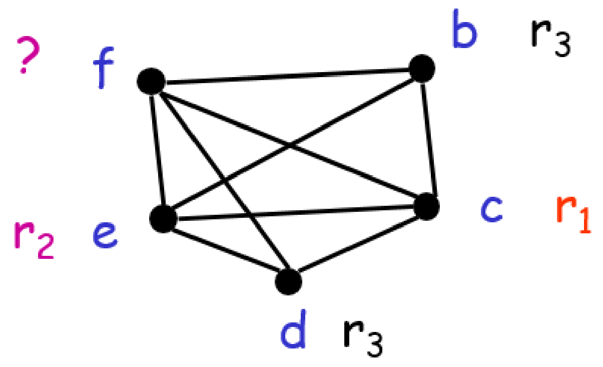
寄存器分配：图着色

- What if the Heuristic Fails?
  - Remove f and continue the simplification
    - Simplification now succeeds: b, d, e, c



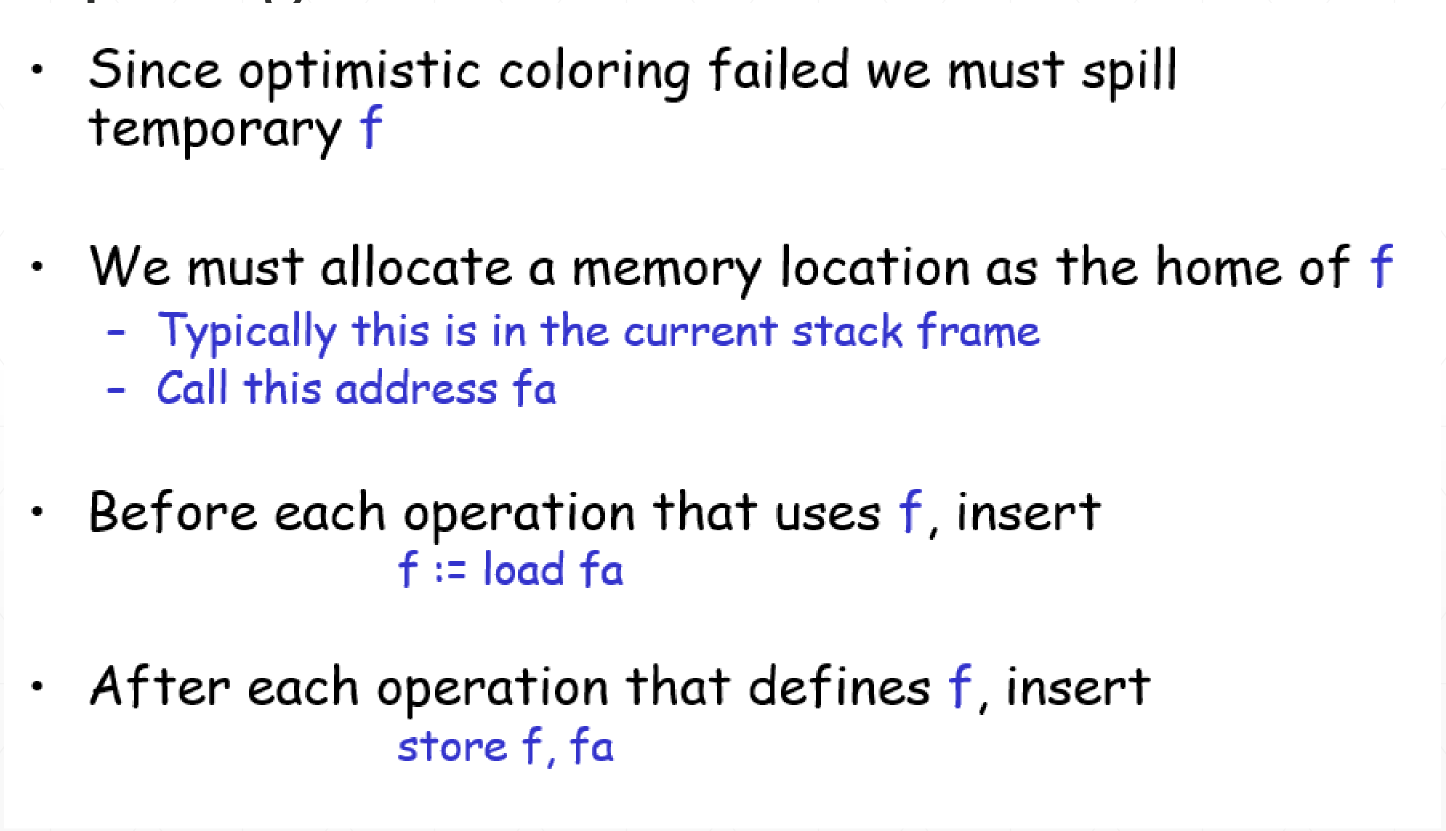
寄存器分配：图着色

- What if the Heuristic Fails?
  - On the assignment phase we get to the point when we have to assign a color to f
  - We hope that among the 4 neighbors of f we use less than 3 colors ⇒ <u>optimistic coloring</u>



寄存器分配：图着色

- Spilling



- Since optimistic coloring failed we must spill temporary f

- We must allocate a memory location as the home of f
  - Typically this is in the current stack frame
  - Call this address fa

- Before each operation that uses f, insert

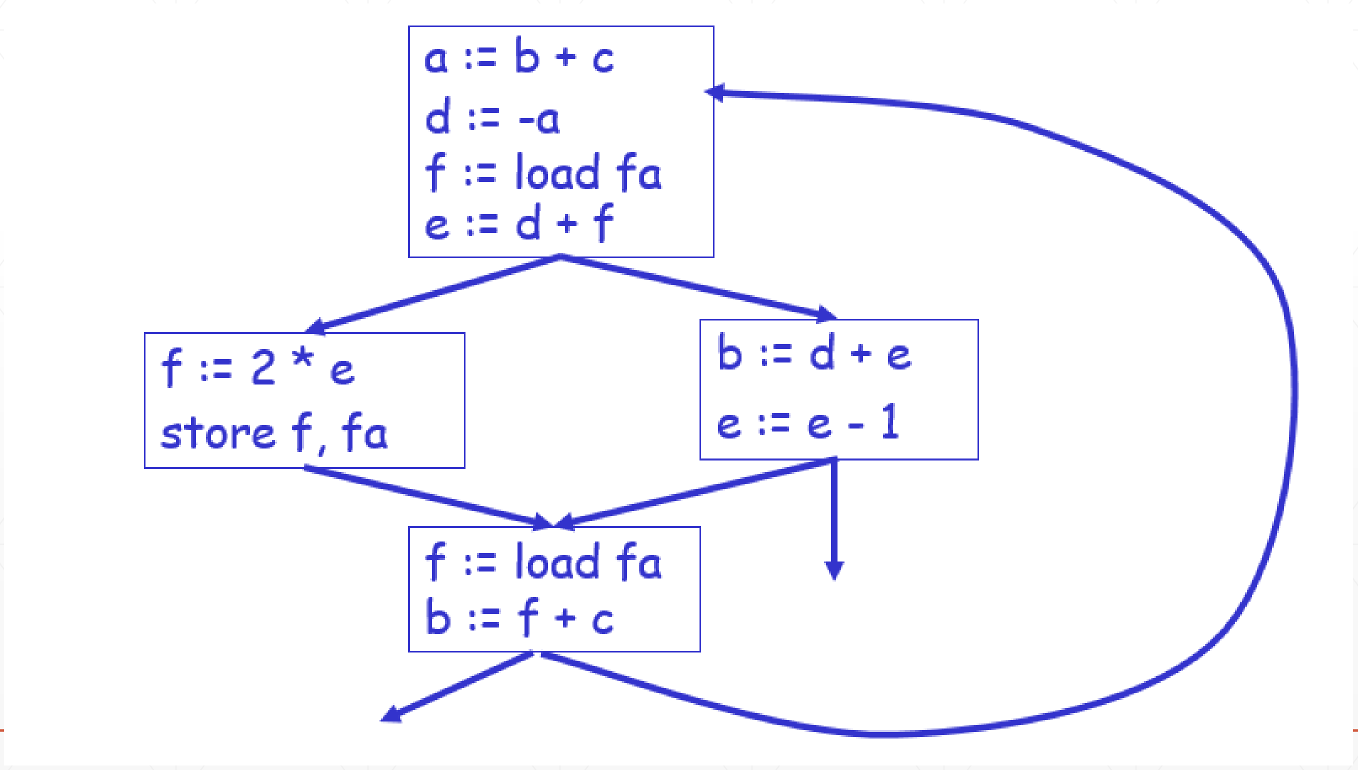
```
              f := load fa
```

- After each operation that defines f, insert

```
              store f, fa
```

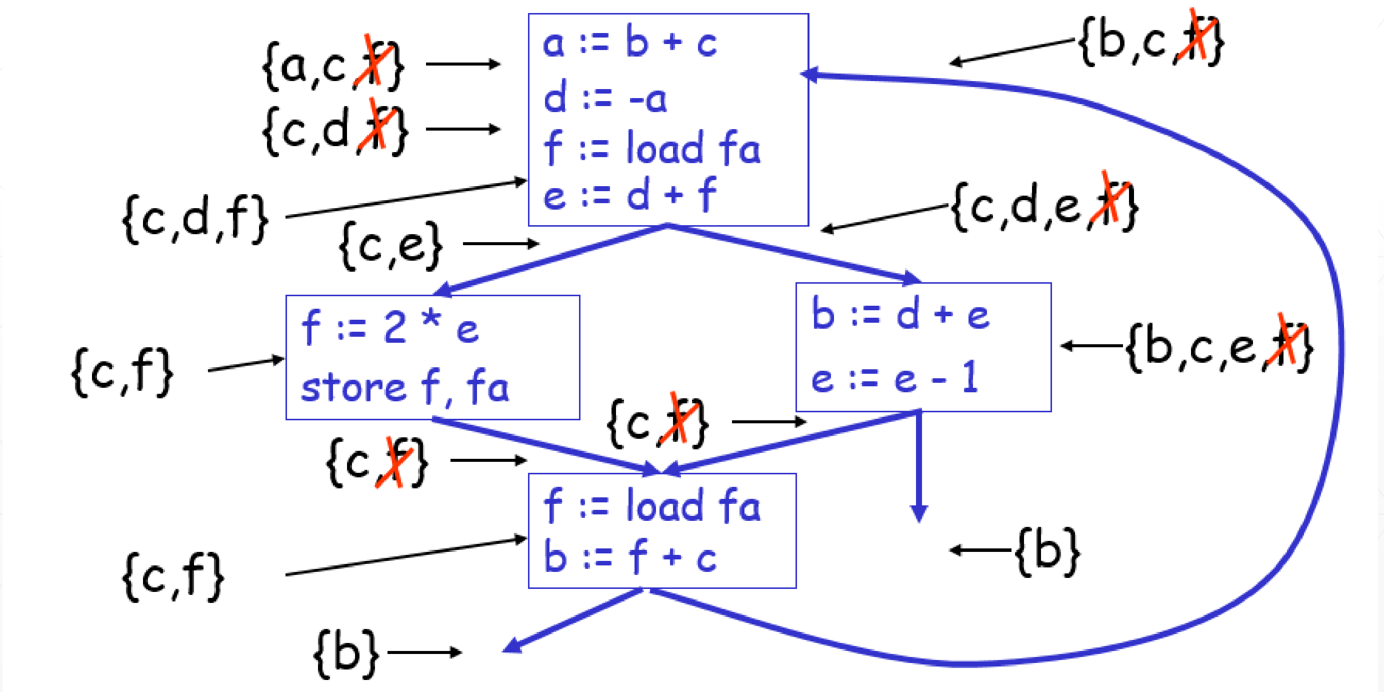
寄存器分配：图着色

- Spilling举例
  - Spilling f 之后新的代码



寄存器分配：图着色

- 重新计算活跃信息
  - Spilling之后的活跃信息

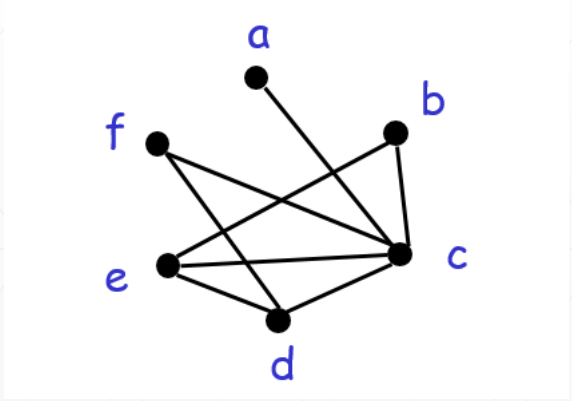


寄存器分配：图着色

- 重新计算活跃信息
  - 新的活跃信息与以前的比较相近
  - f是活跃的
    - f := load fa 和下一条指令之间
    - store f, fa和前一条指令之间
  - Spilling减少了f的live range
  - 因此减少了冲突
  - RIG中f的相邻节点也就减少了

寄存器分配：图着色

- Spilling之后重新计算RIG
  - 主要的改变就是移除那些spilled节点的边
  - 新的IRG中f只与c和d冲突
  - 新的IRG是3-colorable



寄存器分配：图着色

- Spilling
  - 在找到着色方案之前可能需要多次Spilling
  - 重点在于选择对哪个变量做spilling
  - 可能的启发式信息
    - 对冲突较多的临时变量进行spill
    - 对定义和使用较少的变量进行spill
    - 避免对内存循环中的变量做spill
  - 还有很多其他的启发式信息