



中国科学技术大学
University of Science and Technology of China

代码生成

《编译原理和技术》

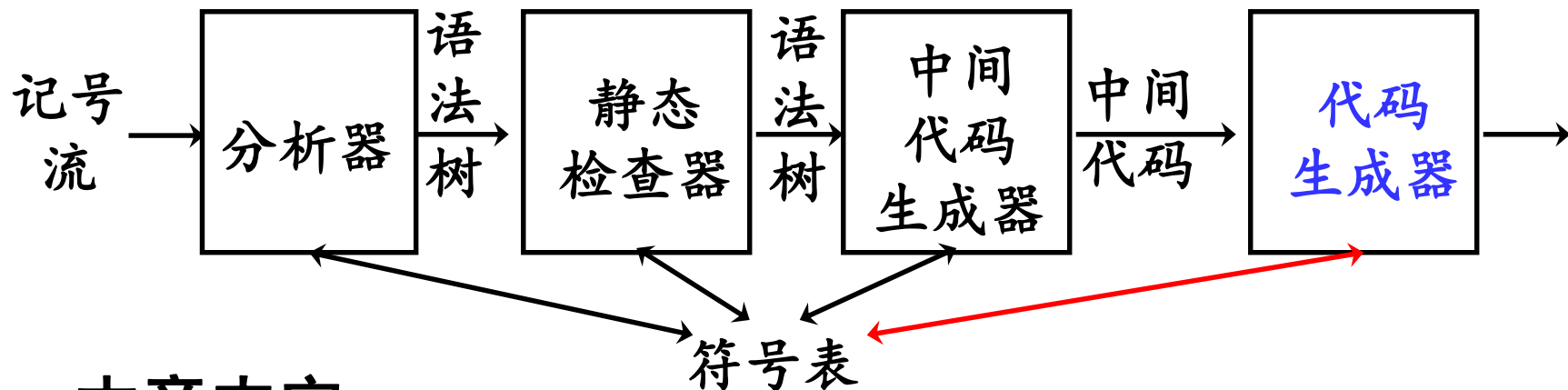
张昱

0551-63603804, yuzhang@ustc.edu.cn

中国科学技术大学
计算机科学与技术学院



本章内容



本章内容

- 代码生成：中间代码IR → 目标机器指令序列
- 涉及目标机器指令选择、寄存器分配和计算次序选择等基本问题
- LLVM中的代码生成



1. 代码生成器设计中的问题

- ☐ 目标程序
- ☐ 指令选择
- ☐ 寄存器的分配和指派
- ☐ 计算次序



代码生成器设计中的问题

□ 目标程序(target program)

■ 绝对机器语言程序(absolute machine-language ...)

□ 目标程序将装入到内存的**固定**地方

□ 粗略地说，相当于现在的可执行目标模块

■ 可重定位目标模块(relocatable object module)

□ 代码中含重定位信息，以适应重定位要求



代码生成器设计中的问题

□ 目标程序

■ 可重定位目标模块

.L7:

`testl %eax,%eax`

`je .L3`

`testl %edx,%edx`

`je .L7`

`movl %edx,%eax`

`jmp .L7`

.L3:

`leave`

`ret`

可重定位目标模块中，
需要有蓝色部分的重定
位信息



代码生成器设计中的问题

□ 目标程序

■ 绝对机器语言程序

- 目标程序将装入到内存的固定地方
- 粗略地说，相当于现在的可执行目标模块

■ 可重定位目标模块(relocatable object module)

- 代码中含重定位信息，以适应重定位要求
- 允许对程序模块**分别编译**
- 调用其它先前编译好的程序模块



代码生成器设计中的问题

□ 目标程序

■ 绝对机器语言程序

■ 可重定位目标模块

- 代码中含重定位信息，以适应重定位要求

- 允许对程序模块分别编译

- 调用其它先前编译好的程序模块

■ 汇编语言程序(assembly-language program)

- 生成汇编程序，可以避免编译器重复汇编器的工作

- 从教学角度，增加可读性



□ 指令的选择(instruction selection)

- 目标机器指令系统的性质决定指令选择的难易程度，指令系统的**统一性**和**完备性**是重要因素
- 指令的**速度**和**机器特点**是另一些重要的因素



代码生成器设计中的问题

□ 代码生成机制

逐条语句地产生代码，常常会得到**低质量**的代码

例：三地址语句 $x = y + z$ (假设 x 、 y 和 z 都是静态分配)

MOV y , R0 /* 把 y 装入寄存器 R0 */

ADD z , R0 /* 把 z 加到 R0 上 */

MOV R0, x /* 把 R0 存入 x 中 */



代码生成器设计中的问题

语句序列

a = b + c

d = **a** + e

的一种目标代码如下：

```
MOV    b,    R0
ADD    c,    R0
MOV    R0,    a
MOV    a,    R0
ADD    e,    R0
MOV    R0,    d
```

X86-32位汇编

```
movl   b, %edx
movl   c, %eax
addl   %edx, %eax
movl   %eax, a
movl   a, %edx
movl   c, %eax
addl   %edx, %eax
movl   %eax, d
```

```
int a,b,c,d;
void f() {
    a = b+c;
    d = a+c;
}
```

```
.text
.comm  a,4,4
.comm  b,4,4
.comm  c,4,4
.comm  d,4,4
```

声明为未初始化的通用内存区域
符号, 长度, 对齐



代码生成器设计中的问题

arm-32位汇编

```
ldr    r3, .L2
ldr    r2, [r3]
ldr    r3, .L2+4
ldr    r3, [r3]
add    r3, r2, r3
ldr    r2, .L2+8
str    r3, [r2]
ldr    r3, .L2+8
ldr    r2, [r3]
ldr    r3, .L2+4
ldr    r3, [r3]
add    r3, r2, r3
ldr    r2, .L2+12
str    r3, [r2]
```

arm-32位汇编

```
.L2:
.word  b
.word  c
.word  a
.word  d
```

X86-32位汇编

```
movl   b, %edx
movl   c, %eax
addl   %edx, %eax
movl   %eax, a
movl   a, %edx
movl   c, %eax
addl   %edx, %eax
movl   %eax, d
```

```
int a,b,c,d;
void f() {
    a = b+c;
    d = a+c;
}
```

```
.text
.comm  a,4,4
.comm  b,4,4
.comm  c,4,4
.comm  d,4,4
```

声明为未初始化的通用内存区域
符号, 长度, 对齐



代码生成器设计中的问题

语句序列 $a = b + c$

$d = a + e$

的一种目标代码如下：

MOV b, R0

ADD c, R0

MOV R0, a

MOV a, R0

ADD e, R0

MOV R0, d

由于a的值仍然存于寄存器R0中，因此该指令是冗余的。



代码生成器设计中的问题

语句序列 $a = b + c$

$d = a + e$

的一种目标代码如下：

MOV b, R0

ADD c, R0

MOV R0, a

~~MOV a, R0~~

ADD e, R0

MOV R0, d

如果a不再被使用，该指令也可以删除。



代码生成器设计中的问题

□ 代码生成机制

- 同一中间表示代码可以实现为多组指令序列
不同实现之间的**效率差别**是很大的
- 例：语句 $a = a + 1$ 可以有两种实现方式

MOV	a,	R0
ADD	#1,	R0
MOV	R0,	a

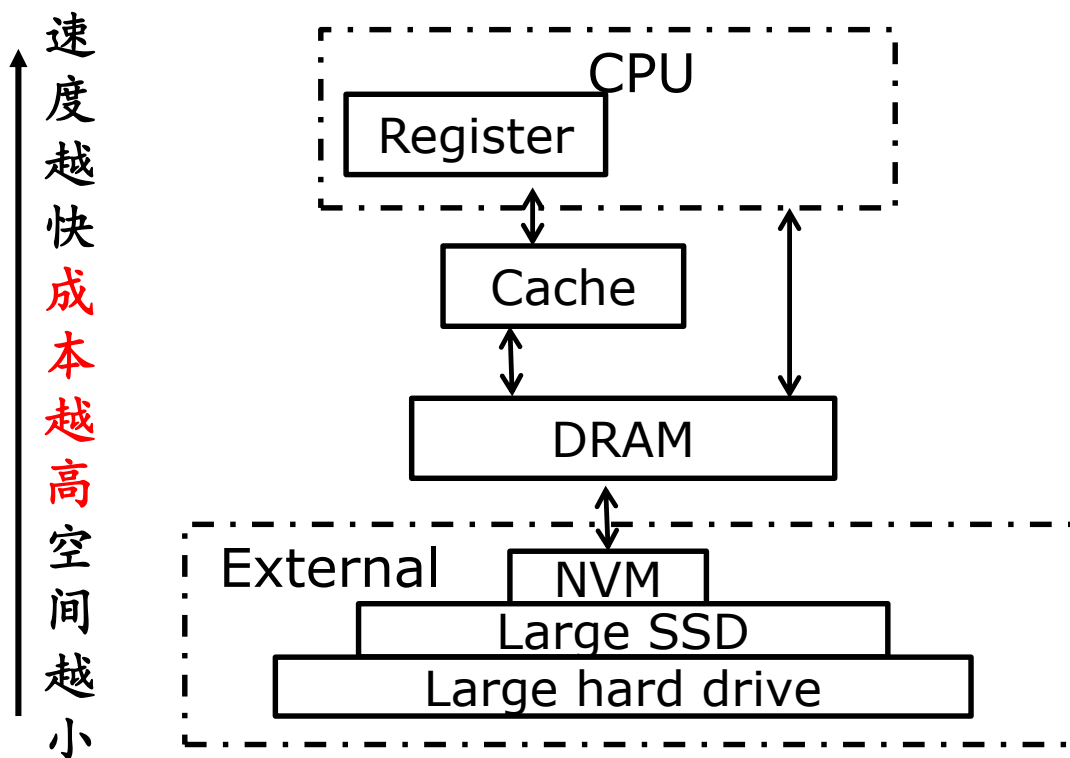
INC a

- 因此，生成高质量代码需要知道**指令代价**。



□ 代码生成机制

考虑 指令的代价和序列长度、**运算对象和结果如何存储**

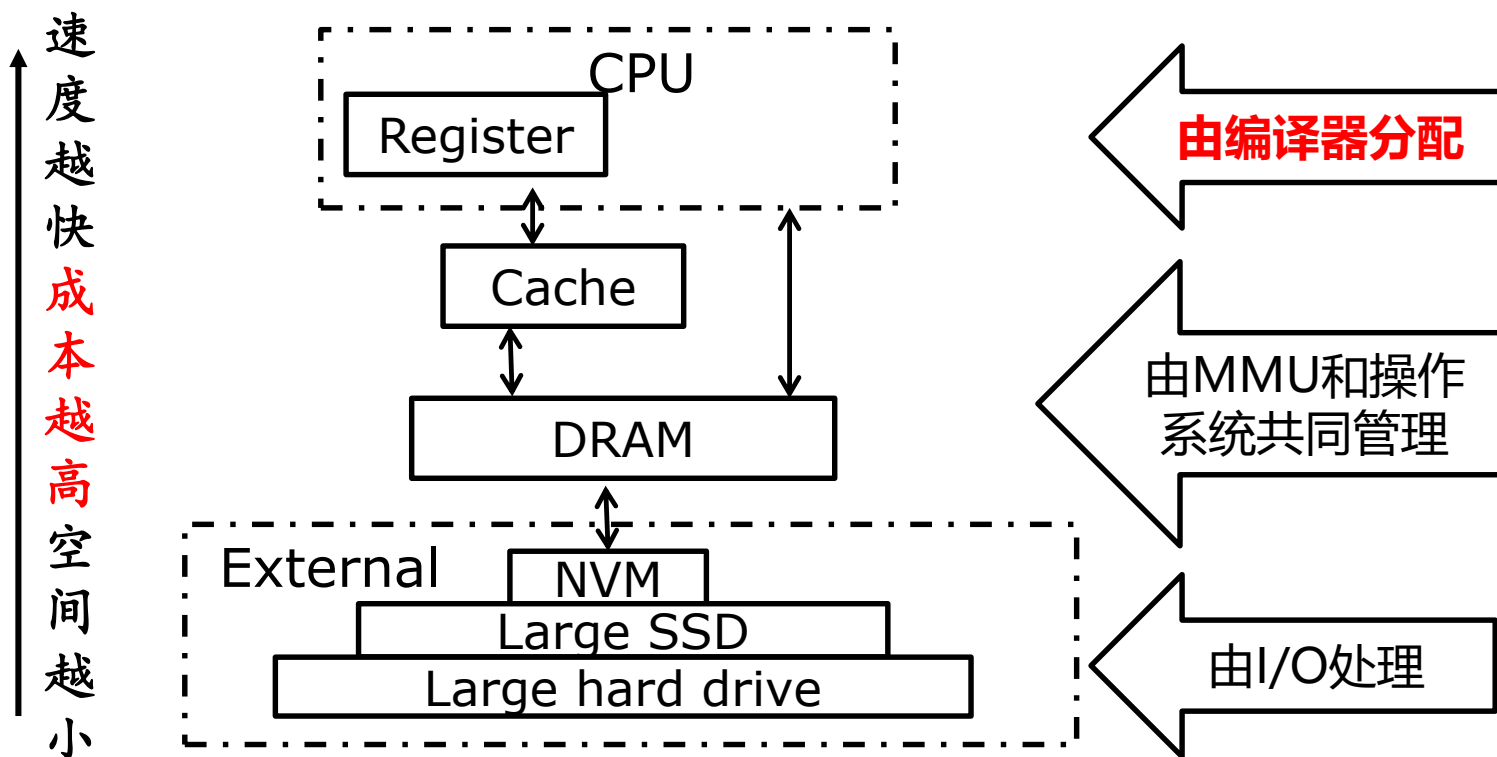




代码生成器设计中的问题

□ 代码生成机制

考虑 指令的代价和序列长度、**运算对象和结果如何存储**





代码生成器设计中的问题

□ 寄存器的合理使用

相比操作置于内存的运算对象，操作寄存器型操作数的指令要短一些，执行也快一些

■ 寄存器分配(register allocation)

- 选择驻留在寄存器中的一组变量

■ 寄存器指派(register assignment)

- 挑选变量要驻留的具体寄存器



□ 计算次序的选择(evaluation order)

- 计算的**执行次序**会影响目标代码的执行效率
- 如，对表达式而言，一种计算次序可能会比其它次序需要较少的寄存器来保存中间结果
- 选择最佳计算次序是一个NP完全问题



2. 目标语言

- 目标机器指令集
- 指令代价
- LLVM中的目标机器描述:
tabgen



目标语言

□ 一个简单目标机器的指令系统

- 字节寻址，四个字节组成一个字
- 有 n 个通用寄存器 $R0, R1, \dots, Rn-1$
- 二地址指令： **op** 源， 目的

MOV {源传到目的}

ADD {源加到目的}

SUB {目的减去源}



目标语言

□ 例 指令实例

MOV R0, M

MOV 4(R0), M

4(R0)的值: *contents(4 + contents(R0))*

MOV *4(R0), M

4(R0)的值: *contents(contents(4 + contents(R0)))

MOV #1, R0



目标语言

□ 指令的代价(instruction costs)

在上述简单的目标机器上，指令代价简化为

**1 + 指令的源和目的寻址模式(addressing mode)的
附加代价**



目标语言

□ 寻址模式和它们的汇编语言形式及附加代价

模式	形式	地址	附加代价
绝对地址	M	M	1
寄存器	R	R	0
变址	$c(R)$	$c + contents(R)$	1
间接寄存器	*R	$contents(R)$	0
间接变址	* $c(R)$	$contents(c + contents(R))$	1
直接量	# c	c	1



目标语言

□ 指令代价简化为

1 + 指令的源和目的地址模式的附加代价

指令	代价
MOV R0, R1	
MOV R5, M	
ADD #1, R3	
SUB 4(R0), *12(R1)	



目标语言

□ 指令代价简化为

1 + 指令的源和目的地址模式的附加代价

指令	代价	
MOV R0, R1	1	寄存器
MOV R5, M	2	寄存器+内存
ADD #1, R3	2	常量+寄存器
SUB 4(R0), *12(R1)	3	变址+间接变址



目标语言

□ 例 $a = b + c$, a 、 b 和 c 都静态分配内存单元

■ 可生成

MOV b, R0

ADD c, R0

MOV R0, a

■ 也可生成

MOV b, a

ADD c, a



目标语言

□ 例 $a = b + c$, a 、 b 和 c 都静态分配内存单元

■ 可生成

MOV b, R0

ADD c, R0

代价= 6

MOV R0, a

■ 也可生成

MOV b, a

ADD c, a

代价= 6



目标语言

□ 例 $a = b + c$, a 、 b 和 c 都静态分配内存单元

■ 若 $R0$, $R1$ 和 $R2$ 分别含 a , b 和 c 的地址, 则可生成

`MOV *R1, *R0`

`ADD *R2, *R0` 代价= 2

■ 若 $R1$ 和 $R2$ 分别含 b 和 c 的值, 并且 b 的值在这个赋值后不再需要, 则可生成

`ADD R2, R1`

`MOV R1, a` 代价= 3



□ 目标机器

寄存器、寄存器类、指令集、调用约定(calling convention)

□ TableGen

- C++风格的语法: TableGen编程指南

- LLVM中已定义的不同类型的后端

 - RegisterInfo, InstrInfo, AsmWriter...

- 通过提取不同架构的相同信息, 避免冗余开发

- TableGen后端生成C++的.inc文件

利用 llvm-tblgen工具处理.td文件, 生成描述后端的.inc文件

利用tblgen描述寄存器

□ 在X86RegisterInfo.td文件中定义了X86Reg抽象类

```
class X86Reg<string n, bits<16> Enc, list<Register> subregs = []> : Register<n> {  
  let Namespace = "X86";  
  let HWEncoding = Enc;  
  let SubRegs = subregs;  
}
```

设置命名空间为X86

在Target.td中定义的寄存器抽象记录

□ 再将X86Reg作为父类定义具体的寄存器

```
let SubRegIndices = [sub_16bit, sub_16bit_hi], CoveredBySubRegs = 1 in {  
def EAX : X86Reg<"eax", 0, [AX, HAX]>, DwarfRegNum<[-2, 0, 0]>;  
def EDX : X86Reg<"edx", 2, [DX, HDX]>, DwarfRegNum<[-2, 2, 2]>;  
def ECX : X86Reg<"ecx", 1, [CX, HCX]>, DwarfRegNum<[-2, 1, 1]>;  
def EBX : X86Reg<"ebx", 3, [BX, HBX]>, DwarfRegNum<[-2, 3, 3]>;  
def ESI : X86Reg<"esi", 6, [SI, HSI]>, DwarfRegNum<[-2, 6, 6]>;  
def EDI : X86Reg<"edi", 7, [DI, HDI]>, DwarfRegNum<[-2, 7, 7]>;  
def EBP : X86Reg<"ebp", 5, [BP, HBP]>, DwarfRegNum<[-2, 4, 5]>;  
def ESP : X86Reg<"esp", 4, [SP, HSP]>, DwarfRegNum<[-2, 5, 4]>;  
def EIP : X86Reg<"eip", 0, [IP, HIP]>, DwarfRegNum<[-2, 8, 8]>;  
}
```



利用tblgen描述寄存器类

- 在X86RegisterInfo.td文件中，除了定义寄存器之外，还定义许多寄存器类(Register class)

32位通用寄存器
(general-purpose
registers)类

```
def GR32 : RegisterClass<"X86", [i32], 32,  
    (add EAX, ECX, EDX, ESI, EDI, EBX, EBP, ESP,  
     R8D, R9D, R10D, R11D, R14D, R15D, R12D, R13D)>;
```

在Target.td中定义的
寄存器类抽象记录



利用tblgen描述指令集

□ 在X86InstrFormat.td中定义了所有指令的超类：

```
class X86Inst<bits<8> opcode, Format f, ImmType i, dag outs, dag ins,  
| | | | | string AsmStr, Domain d = GenericDomain>  
: Instruction {
```

□ 对于不同类型的指令，再定义不同的抽象类：

```
class I<bits<8> o, Format f, dag outs, dag ins, string asm,  
| | | list<dag> pattern, Domain d = GenericDomain>  
: X86Inst<o, f, NoImm, outs, ins, asm, d> {
```

□ 在X86InstrArithmetic.td文件中描述算术指令，如

```
def LEA16r : I<0x8D, MRMSrcMem,  
| | | | | (outs GR16:$dst), (ins anymem:$src),  
| | | | | "lea{w}\t{$src|$dst}, {$dst|$src}", []>, OpSize16;
```




llvm-tblgen X86.td -gen-register-info

其部分输出：

**GR32寄存器类
变量在.inc文件
中的表示 →**

```
// GR32 Register Class...
const MCPhysReg GR32[] = {
    X86::EAX, X86::ECX, X86::EDX, X86::ESI,
    X86::EDI, X86::EBX, X86::EBP, X86::ESP,
    X86::R8D, X86::R9D, X86::R10D, X86::R11D,
    X86::R14D, X86::R15D, X86::R12D, X86::R13D,
};
```

```
namespace llvm {

class MCRegisterClass;
extern const MCRegisterClass X86MCRegisterClasses[];

namespace X86 {
enum {
    NoRegister,
    AH = 1,
    AL = 2,
    AX = 3,
    BH = 4,
    BL = 5,
    BP = 6,
```

**各个寄存器
在.inc中的表
示：枚举类型**



LLVM中的指令代价

□ 在Target/XXX/XXXTransformationInfo.cpp中规定XXX架构中指令代价，以X86为例

保存指令代价的数据结构

```
/// Cost Table Entry  
struct CostTblEntry {  
    int ISD;  
    MVT::SimpleValueType Type;  
    unsigned Cost;  
};
```

ISD:
SelectionDAG结点
Type:
目标机器值类型

```
static const CostTblEntry SLMCostTable[] = {  
    { ISD::MUL,    MVT::v4i32, 11 }, // pmulld  
    { ISD::MUL,    MVT::v8i16, 2  }, // pmullw  
    { ISD::MUL,    MVT::v16i8, 14 }, // extend/pmullw/trunc sequence.  
    { ISD::FMUL,   MVT::f64,    2  }, // mulsd  
    { ISD::FMUL,   MVT::v2f64,  4  }, // mulpd  
    { ISD::FMUL,   MVT::v4f32,  2  }, // mulps  
    { ISD::FDIV,   MVT::f32,    17 }, // divss  
    { ISD::FDIV,   MVT::v4f32, 39 }, // divps  
    { ISD::FDIV,   MVT::f64,    32 }, // divsd  
    { ISD::FDIV,   MVT::v2f64, 69 }, // divpd  
    { ISD::FADD,   MVT::v2f64,  2  }, // addpd  
    { ISD::FSUB,   MVT::v2f64,  2  }, // subpd  
    // v2i64/v4i64 mul is custom lowered as a series of long:  
    // multiplies(3), shifts(3) and adds(2)  
    // slm muldq version throughput is 2 and addq throughput 4  
    // thus: 3X2 (muldq throughput) + 3X1 (shift throughput) +  
    //        3X4 (addq throughput) = 17  
    { ISD::MUL,    MVT::v2i64, 17 },
```

部分指令代价的计算是有规律的



3. 代码生成器的输入

- ☐ 中间代码IR
- ☐ 基本块与流图
- ☐ 循环 9.6节



一般形式: $x = y \text{ op } z$

□ 程序举例

```
prod = 0;  
i = 1;  
do {  
    prod = prod + a[i] * b[i];  
    i = i + 1;  
} while (i <= 20);
```

第*i*个元素的
类型为int

(1) prod = 0

(2) i = 1

(3) $t_1 = 4 * i$

(4) $t_2 = a[t_1]$

(5) $t_3 = 4 * i$

(6) $t_4 = b[t_3]$

(7) $t_5 = t_2 * t_4$

(8) $t_6 = \text{prod} + t_5$

(9) prod = t_6

(10) $t_7 = i + 1$

(11) i = t_7

(12) if i <= 20 goto (3)

元素的地址要转
换成按字节寻址



基本块和流图

(1) $\text{prod} = 0$

(2) $i = 1$

(3) $t_1 = 4 * i$

(4) $t_2 = a[t_1]$

(5) $t_3 = 4 * i$

(6) $t_4 = b[t_3]$

(7) $t_5 = t_2 * t_4$

(8) $t_6 = \text{prod} + t_5$

(9) $\text{prod} = t_6$

(10) $t_7 = i + 1$

(11) $i = t_7$

(12) if $i \leq 20$ goto (3)

(1) $\text{prod} = 0$

(2) $i = 1$

B_1

(3) $t_1 = 4 * i$

(4) $t_2 = a[t_1]$

(5) $t_3 = 4 * i$

(6) $t_4 = b[t_3]$

(7) $t_5 = t_2 * t_4$

(8) $t_6 = \text{prod} + t_5$

(9) $\text{prod} = t_6$

(10) $t_7 = i + 1$

(11) $i = t_7$

(12) if $i \leq 20$ goto (3)

B_2



流图(变换成 SSA 格式)

(1) $\text{prod} = 0$
(2) $i_1 = 1$
(3) $i_3 = \phi(i_1, i_2)$
(4) $t_1 = 4 * i_3$
(5) $t_2 = a[t_1]$
(6) $t_3 = 4 * i_3$
(7) $t_4 = b[t_3]$
(8) $t_5 = t_2 * t_4$
(9) $t_6 = \text{prod} + t_5$
(10) $\text{prod} = t_6$
(11) $t_7 = i_3 + 1$
(12) $i_2 = t_7$
(13) if $i_2 \leq 20$ goto (3)

(1) $\text{prod} = 0$

(2) $i_1 = 1$

B_1

(3) $i_3 = \phi(i_1, i_2)$

(4) $t_1 = 4 * i_3$

(5) $t_2 = a[t_1]$

(6) $t_3 = 4 * i_3$

(7) $t_4 = b[t_3]$

(8) $t_5 = t_2 * t_4$

(9) $t_6 = \text{prod} + t_5$

(10) $\text{prod} = t_6$

(11) $t_7 = i_3 + 1$

(12) $i_2 = t_7$

(13) if $i_2 \leq 20$ goto (3)

B_2

利用流图，可快速找到 B_2 的前驱基本块，按控制流逆向找到最近对 i 的定值



流图上的程序点和路径

□ 流图上的(程序)点

- 基本块中，两个相邻的语句之间为程序的一个点
- 基本块的开始点和结束点

□ 流图上的路径

- 点序列 p_1, p_2, \dots, p_n ，对1和 $n - 1$ 间的每个 i ，满足
 - (1) p_i 是先于一个语句的点， p_{i+1} 是同一基本块中位于该语句后的点，或者
 - (2) p_i 是某基本块的结束点， p_{i+1} 是后继块的开始点



流图上的路径

□ 流图(flow graph)

举例

(1, 2, 3, 4, 9)

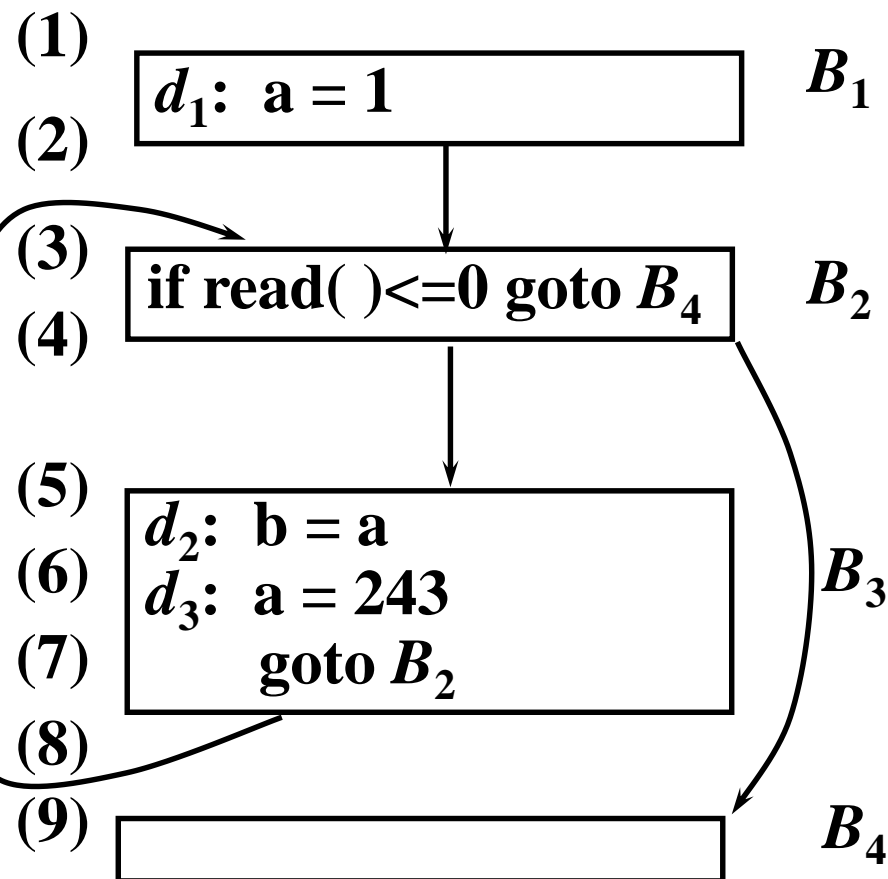
(1, 2, 3, 4, 5, 6, 7, 8, 3, 4, 9)

(1, 2, 3, 4, 5, 6, 7, 8,
3, 4, 5, 6, 7, 8, 3, 4, 9)

(1, 2, 3, 4, 5, 6, 7, 8,
3, 4, 5, 6, 7, 8,
3, 4, 5, 6, 7, 8, ...)

■ 路径长度无限

■ 路径数无限





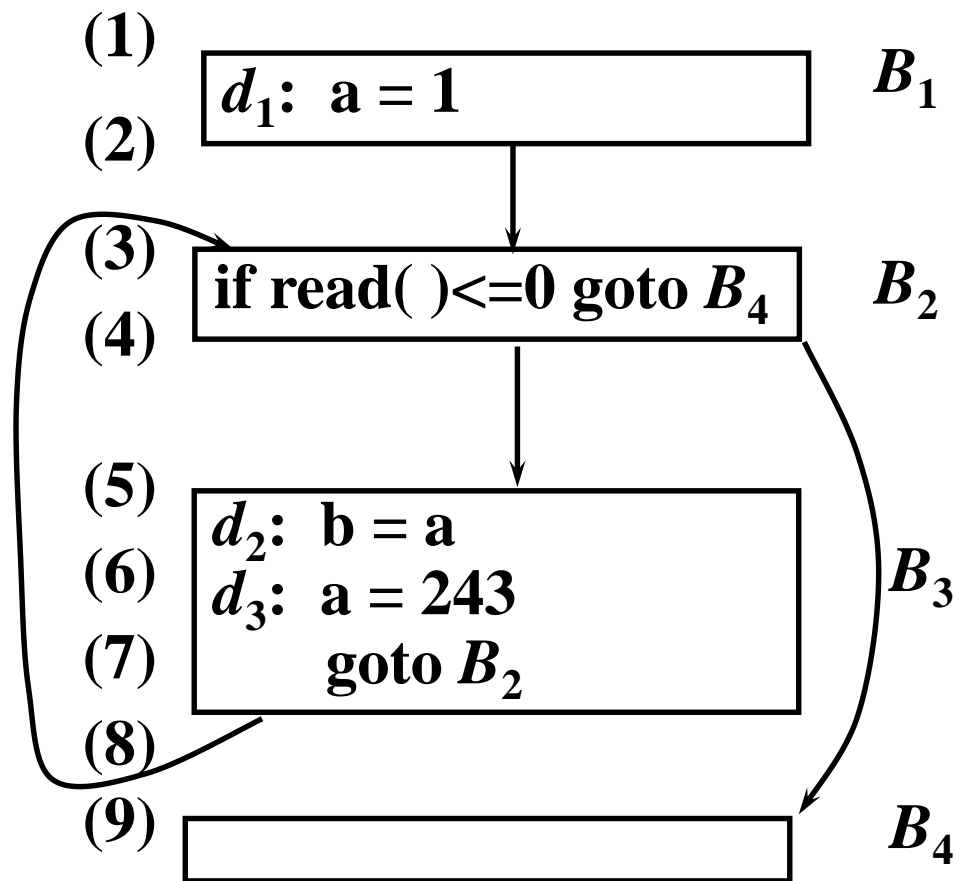
□ 循环

流图中的一个结点集合 L 是一个循环，如果：

- 该集合中所有结点是强连通的
- 该集合有唯一的入口结点


□ 内循环

不包含其他循环的循环





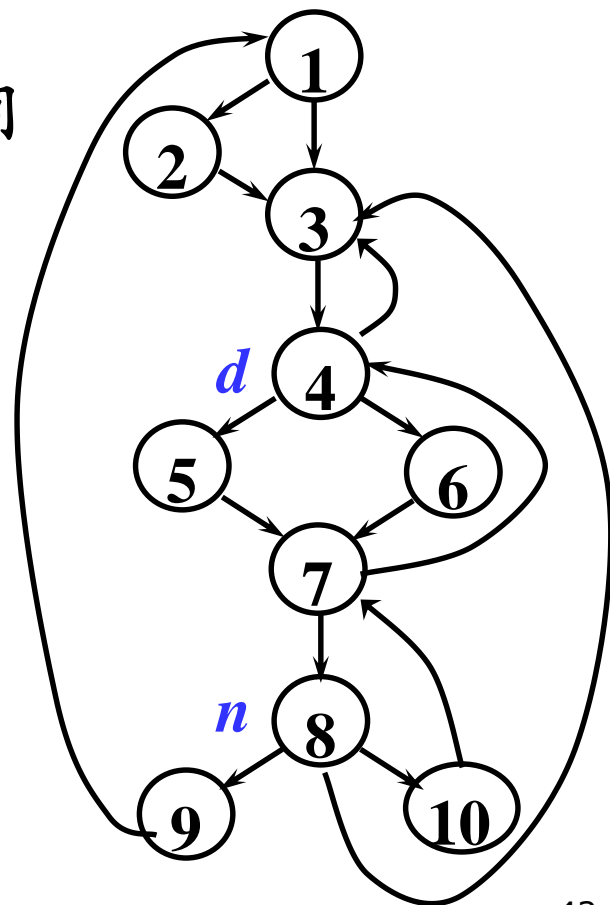
□ 识别循环并对循环专门处理的重要性

- 程序执行的大部分时间消耗在循环上，改进循环性能
的优化会对程序执行产生显著影响
 - 循环也会影响程序分析的运行时间
- 
- A diagram showing a loop structure. It consists of a circular node labeled '1' with a vertical line extending downwards from its center. A curved arrow starts from the top of this vertical line, loops around to the left, and points back into the top of the circular node, representing a self-loop or a cycle in a control flow graph.

□ 支配结点

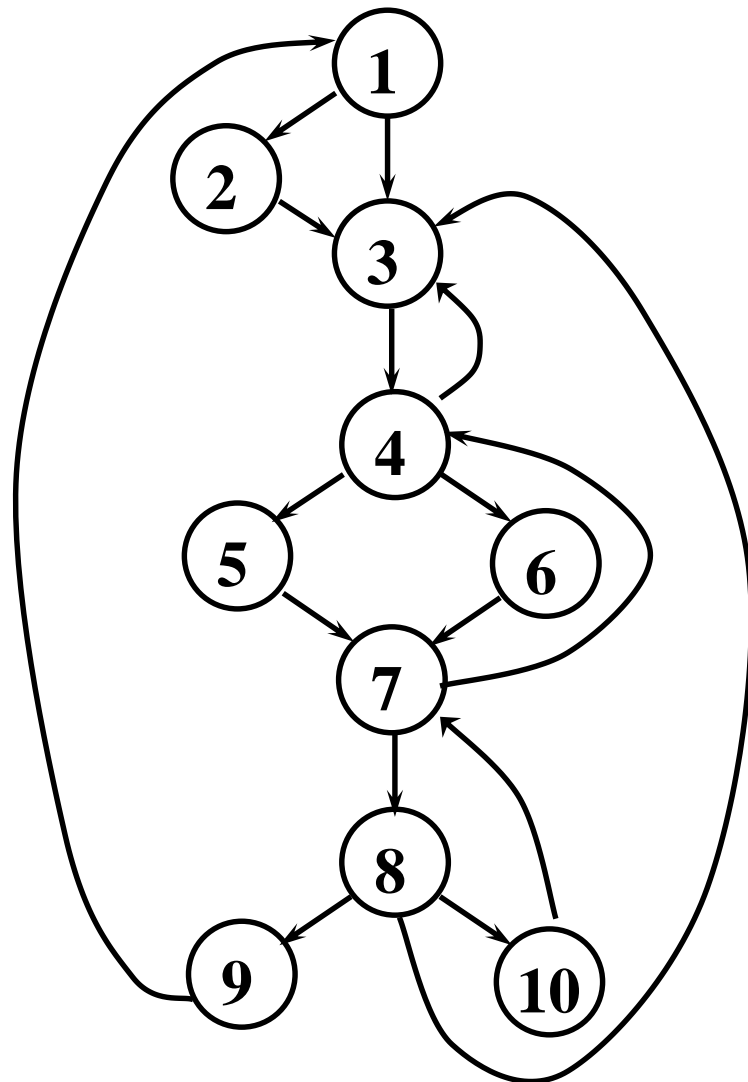
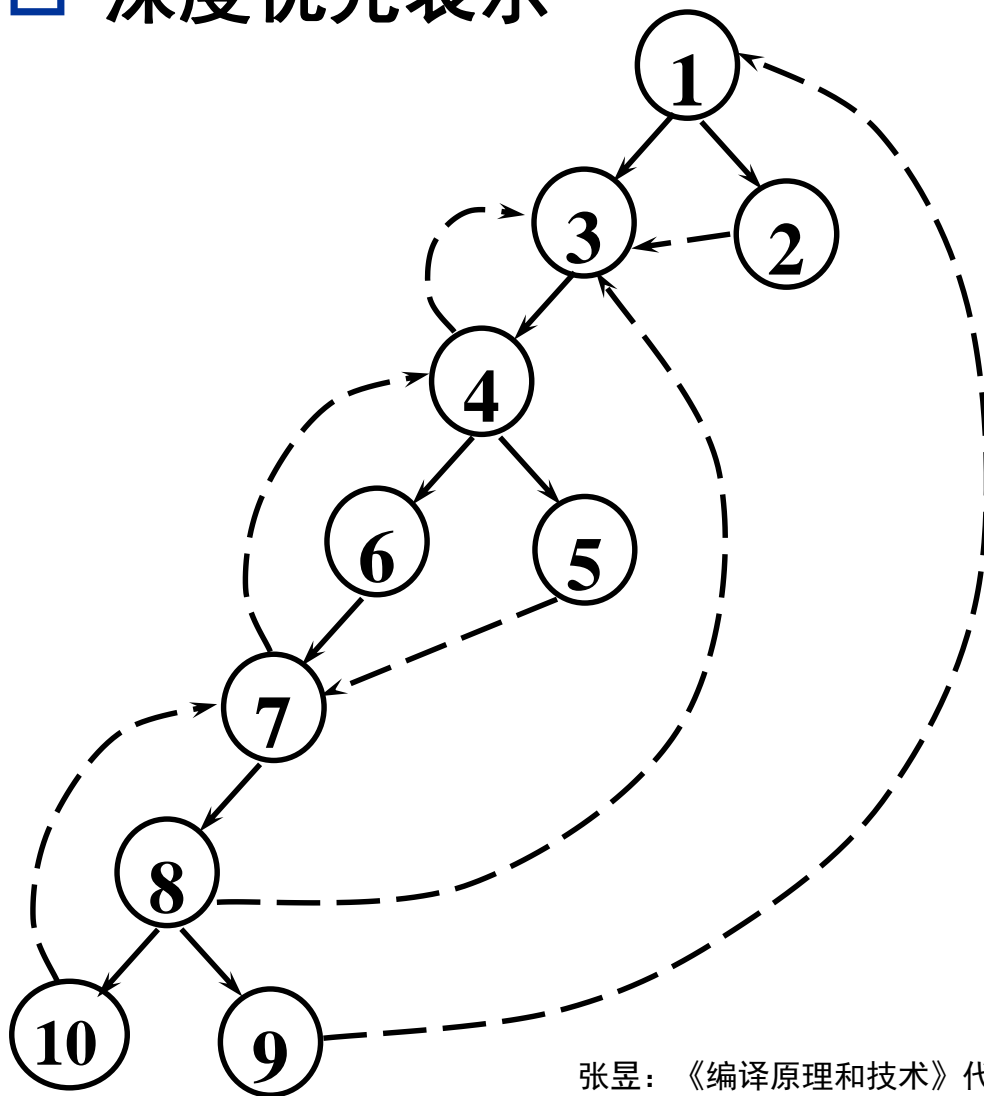
d 是 n 的支配结点($d \text{ dom } n$): 若从初始结点起, 每条到达 n 的路径都要经过 d

- 结点是它本身的支配结点
- 循环的入口是循环中所有结点的支配结点





□ 深度优先表示





流图中的边的分类

□ 深度优先表示

- **前进边**(深度优先生成树的边, 以及存在边 $m \rightarrow n$ 且 n 在树中是 m 的真后代)

- $m \rightarrow n$ 是 **后撤边**(retreating edge)

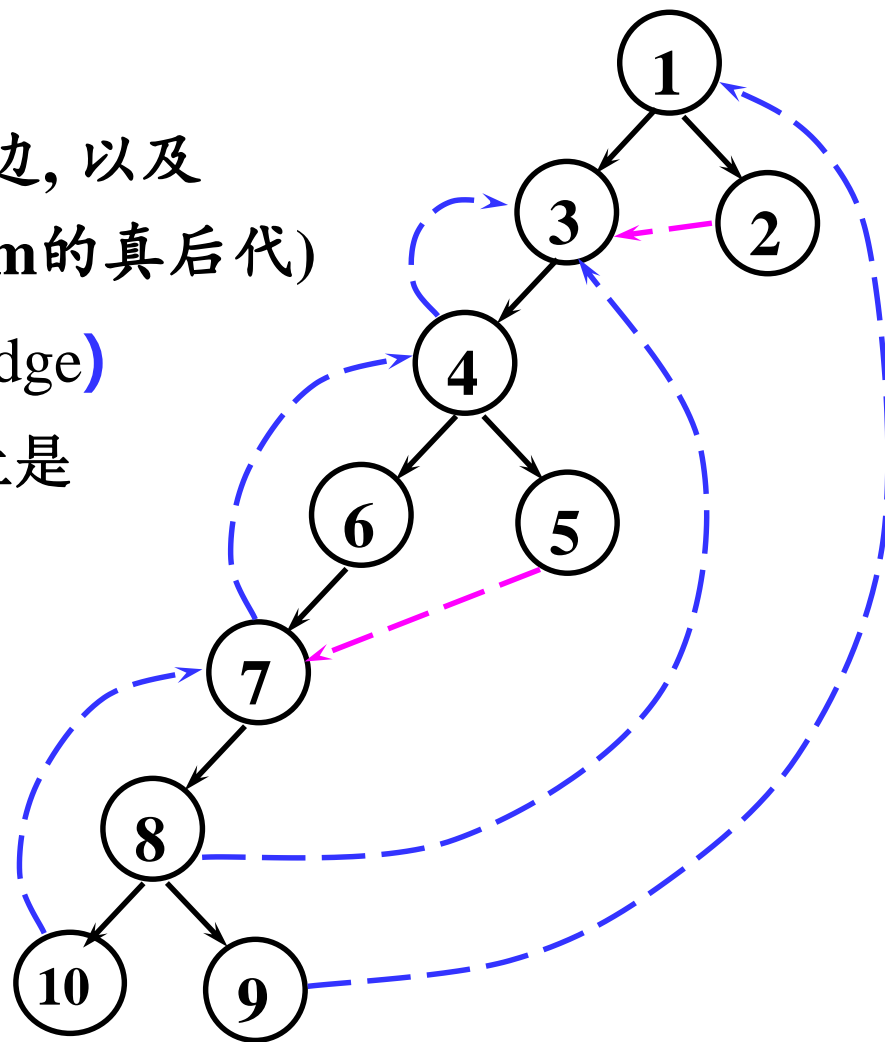
- 如果 n 在深度优先生成树上是 m 的祖先

- $4 \rightarrow 3$ 、 $7 \rightarrow 4$ 、 $10 \rightarrow 7$ 、 $8 \rightarrow 3$ 和 $9 \rightarrow 1$

- $m \rightarrow n$ 是 **交叉边**(cross edge)

- 如果 n 和 m 在深度优先生成树上互不为对方的祖先

- $2 \rightarrow 3$ 和 $5 \rightarrow 7$





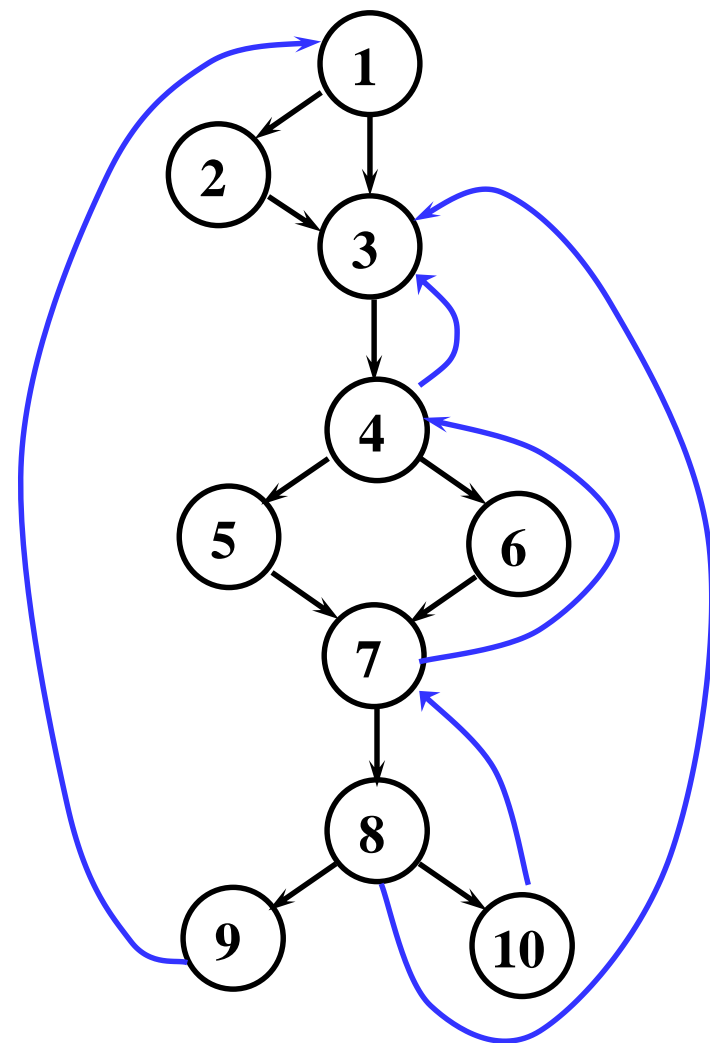
回边和可归约性

□ 回边

如果有 $a \text{ dom } b$ ，那么边 $b \rightarrow a$ 叫做**回边**(back edge)

□ 可归约性

一个流图称为**可归约的**(reducible), 如果
在它任何深度优先生成树上,
所有的后撤边都是回边。



回边和可归约性

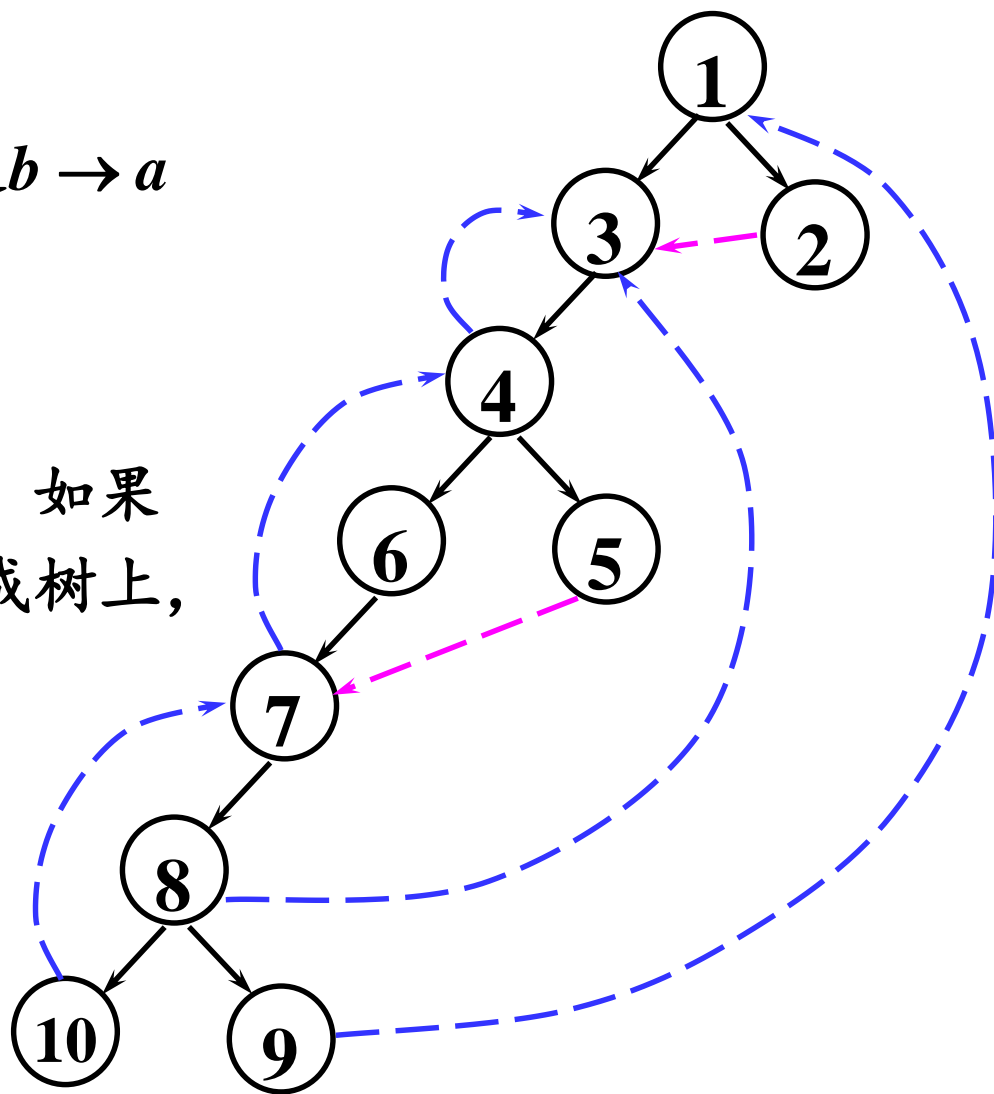
□ 回边

如果有 $a \text{ dom } b$ ，那么边 $b \rightarrow a$ 叫做**回边**

□ 可归约性

一个流图称为**可归约的**，如果在它的任何深度优先生成树上，**所有的后撤边都是回边**。

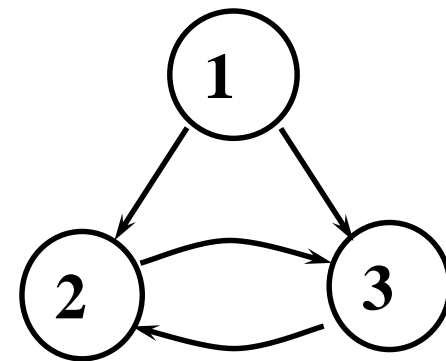
- 如果把一个流图中所有回边删掉后，剩余的图无环





不可归约流图

- 开始结点是1
- $2 \rightarrow 3$ 和 $3 \rightarrow 2$ 都不是回边
- 该图不是无环的
- 从结点2和3两处都能进入由它们构成的环





流图的深度

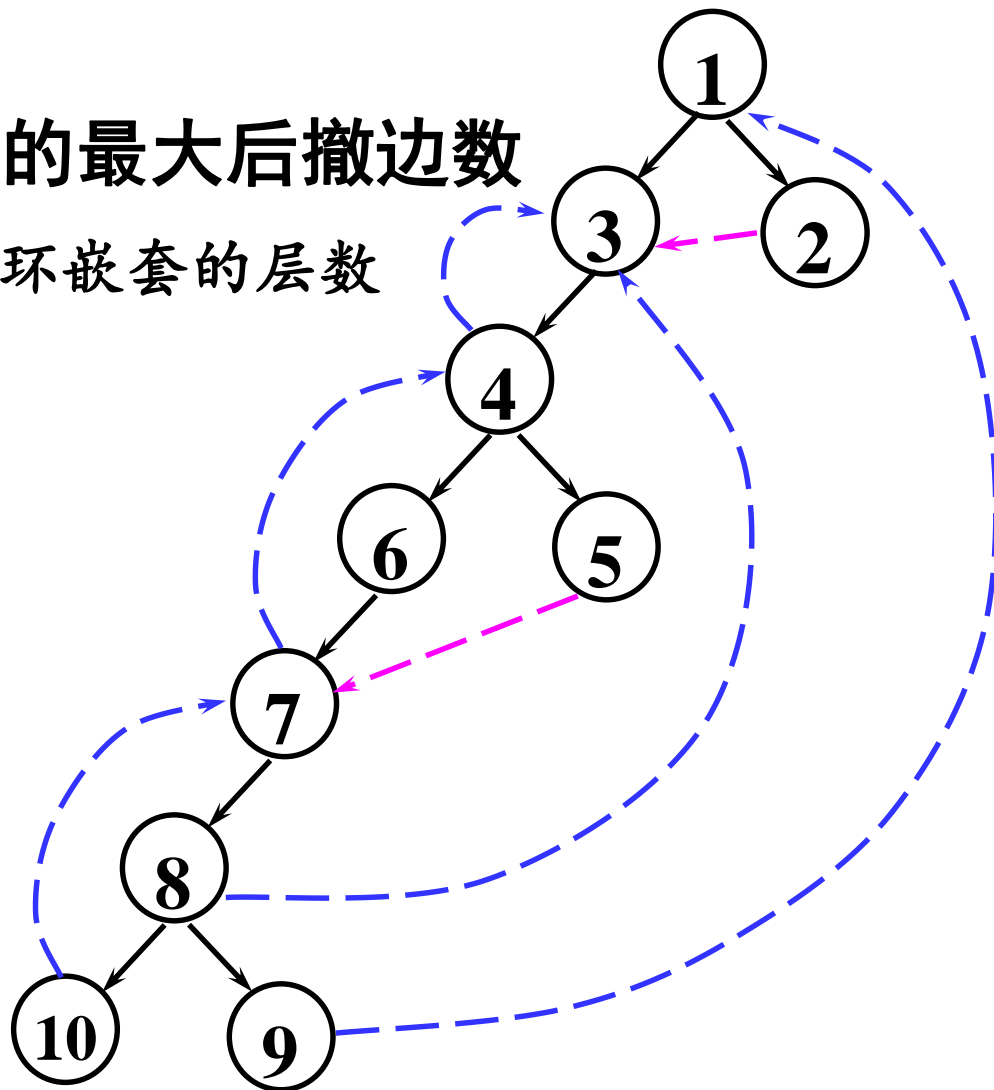
□ 流图的深度

任何可能无环路径上的最大后撤边数

■ 深度不大于流图中循环嵌套的层数

■ 该例深度为3

$10 \rightarrow 7 \rightarrow 4 \rightarrow 3$





自然循环

□ 自然循环的性质

- 有惟一入口结点(**首结点**), 它支配该循环中的所有结点
 - 至少存在一条**回边**进入该循环的首结点
- 是流图的强连通分量(SCC)中的一种类型

□ 回边 $n \rightarrow d$ 确定的自然循环

- d 加上不经过 d 能到达 n 的所有结点
- 结点 d 是该循环的首结点

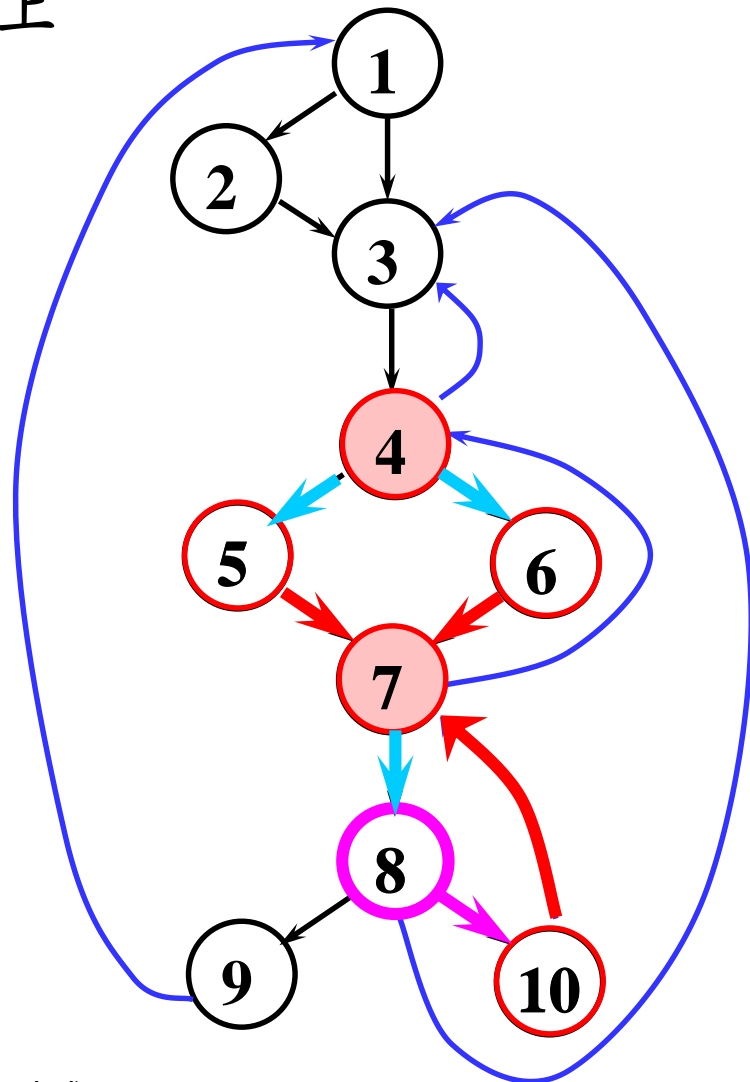
构造算法: 1) $loop$ 初值是 $\{n, d\}$, 标记 d 为“已访问”
2) 从结点 n 开始, 完成对流图 G 的逆向流图的深度优先搜索, 把搜索过程中访问的所有结点都加入 $loop$



自然循环

回边 $n \rightarrow d$ 确定的自然循环是 d 加上
不经过 d 能到达 n 的所有结点

- 回边 $10 \rightarrow 7$
循环 $\{7, 8, 10\}$
- 回边 $7 \rightarrow 4$
循环 $\{4, 5, 6, 7, 8, 10\}$





自然循环

回边 $n \rightarrow d$ 确定的自然循环是 d 加上不经过 d 能到达 n 的所有结点

■ 回边 $10 \rightarrow 7$

循环 $\{7, 8, 10\}$

■ 回边 $7 \rightarrow 4$

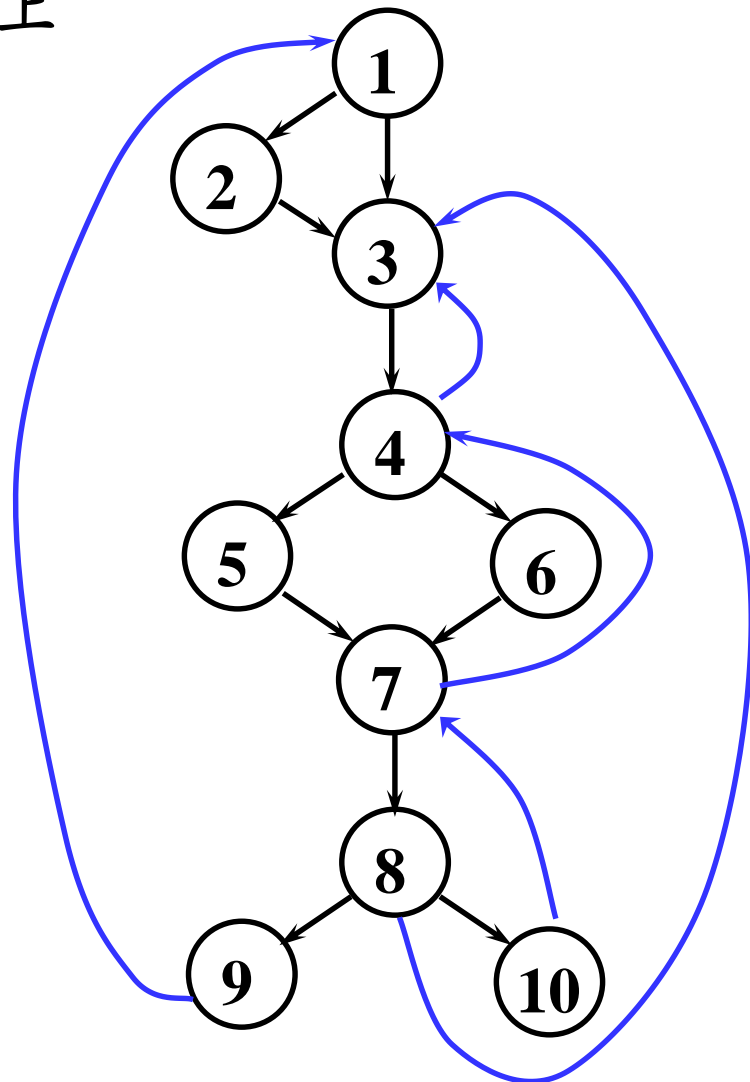
循环 $\{4, 5, 6, 7, 8, 10\}$

■ 回边 $4 \rightarrow 3$ 和 $8 \rightarrow 3$

循环 $\{3, 4, 5, 6, 7, 8, 10\}$

■ 回边 $9 \rightarrow 1$

循环 $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$



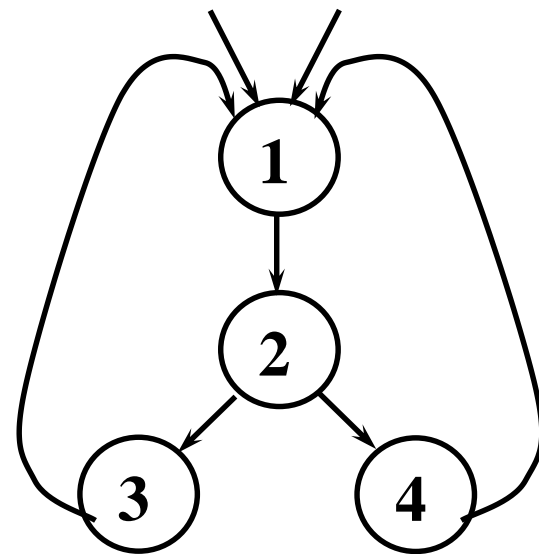


内循环

□ 内循环

若一个循环的结点集合是另一个循环的结点集合的子集

- 两个循环有相同的首结点，但并非一个结点集是另一个的子集，则看成一个循环





下次引用信息

如果一个变量的值当前存放在寄存器中且之后不再被使用，该寄存器就可以被分配给其他变量

变量**接下来被使用的信息** => **帮助寄存器的分配和释放**

□ **名字的引用(use)**: 假设三地址码语句*i*对 *x* 赋值，语句*j*将 *x* 作为运算对象，且*i*到*j*的控制流路径中无其他对*x*的赋值语句，则称语句*j*引用了语句*i*计算的*x*值

□ **计算基本块B中下次引用信息的方法**

从B中的最后一个语句开始，反向扫描到B的开始处，对每个语句*i*: ***x*** = ***y*** op ***z***，在符号表中：

■ 设置***x***为**不活跃**和**无下次引用**

■ 设置***y***、***z***为**活跃**，并把它们的**下次引用**设置为语句*i*



4. 一个简单的代码生成器

- ☐ 寄存器和地址的描述
- ☐ 代码生成算法
- ☐ 寄存器选择函数
- ☐ 为特殊语句产生代码



□ 基本思想

- 依次考虑基本块的每个语句，为其产生代码
 - 跟踪记录哪个值存放在哪个寄存器中
- 假定三地址语句的每种算符都有对应的目标机器算符
- 假定计算结果尽可能长地保留在寄存器中，除非：
 - 该寄存器要用于其它计算，或者
 - 到基本块结束

□ 代码生成中的主要问题

如何最大限度地利用寄存器



□ 寄存器描述符(descriptor)和地址描述符

例: 对 $a = b + c$

- 如果寄存器 R_i 含 b , R_j 含 c , 且 b 此后不再活跃
产生 `ADD R_j , R_i` , 结果 a 在 R_i 中
- 如果 R_i 含 b , 但 c 在内存单元, b 仍然不再活跃
产生 `ADD c , R_i` , 或者产生
`MOV c , R_j`
`ADD R_j , R_i`
- 若 c 的值以后还要用, 第二种代码较有吸引力



一个简单的代码生成器

□ 在代码生成过程中，需要跟踪

寄存器的内容和名字的地址

- 寄存器描述符记住每个寄存器当前存的是什么，即在任
何一点，每个寄存器保存若干个(包括零个)名字的值
例：

```
b = a           // 语句前，R0保存变量a的值
                // 不为该语句产生任何指令
                // 语句后，R0保存变量a和b的值
```



一个简单的代码生成器

□ 在代码生成过程中，需要跟踪

寄存器的内容和名字的地址

- 寄存器描述符记住每个寄存器当前存的是什么，即在任何一点，每个寄存器保存若干个（包括零个）名字的值
 - 名字（变量）的地址描述符记住运行时每个名字的当前值可以在哪个场所找到。这个场所可以是寄存器、栈单元、内存地址、甚至是它们的某个集合
- 例：产生MOV c, R0后，c值可在R0和c的存储单元找到



一个简单的代码生成器

□ 在代码生成过程中，需要跟踪

寄存器的内容和名字的地址

- 寄存器描述符记住每个寄存器当前存的是什么，即在任
何一点，每个寄存器保存若干个（包括零个）名字的值
- 名字（变量）的地址描述符记住运行时每个名字的当前
值可以在哪个场所找到。这个场所可以是寄存器、栈单
元、内存地址、甚至是它们的某个集合
- 例：产生MOV c, R0后，c值可在R0和c的存储单元找到
- 名字的地址信息存于符号表，另建寄存器描述表
- 这两个描述在代码生成过程中是变化的



□ 寄存器选择函数

■ 函数 $getReg(I)$ 返回保存 $I: x = y \ op \ z$ 的 x 值的场所 L

- 如果名字 y 在 R 中，这个 R 不含其它名字的值，并且在执行 $x = y \ op \ z$ 后 y 不再有下次引用，那么返回这个 R 作为 L
- 否则，如果有的话，返回一个空闲寄存器
- 否则，如果 x 在块中有下次引用，或者 op 是必须用寄存器的算符，那么找一个已被占用的寄存器 R (可能产生 $MOV \ R, M$ 指令，并修改 M 的描述)
- 否则，如果 x 在基本块中不再引用，或者找不到适当的被占用寄存器，选择 x 的内存单元作为 L



□ 代码生成算法

■ 对每个三地址语句 $x = y \text{ op } z$

- 调用函数 *getReg* 决定放 $y \text{ op } z$ 计算结果的场所 L
- 查看 y 的地址描述，确定 y 值当前的一个场所 y' 。如果 y 的值还不在于 L 中，产生指令 $\text{MOV } y', L$
- 产生指令 $\text{op } z', L$ ，其中 z' 是 z 的当前场所之一
- 如果 y 和/或 z 的当前值不再引用，在块的出口也不活跃，并且还在寄存器中，那么修改寄存器描述，使得不再包含 y 和/或 z 的值



一个简单的代码生成器

□ 赋值语句 $d = (a - b) + (a - c) + (a - c)$

■ 编译产生三地址语句序列：

$$t_1 = a - b$$

$$t_2 = a - c$$

$$t_3 = t_1 + t_2$$

$$d = t_3 + t_2$$



一个简单的代码生成器

语 句	生成的代码	寄存器描述	名字的地址描述
		寄存器空	
$t_1 = a - b$			
$t_2 = a - c$			
$t_3 = t_1 + t_2$			
$d = t_3 + t_2$			



一个简单的代码生成器

语 句	生成的代码	寄存器描述	名字的地址描述
		寄存器空	
$t_1 = a - b$	MOV a, R0 SUB b, R0	R0含t_1	t_1在R0中
$t_2 = a - c$			
$t_3 = t_1 + t_2$			
$d = t_3 + t_2$			



一个简单的代码生成器

语 句	生成的代码	寄存器描述	名字的地址描述
		寄存器空	
$t_1 = a - b$	MOV a, R0 SUB b, R0	R0含t_1	t_1在R0中
$t_2 = a - c$	MOV a, R1 SUB c, R1	R0含t_1 R1含t_2	t_1在R0中 t_2在R1中
$t_3 = t_1 + t_2$			
$d = t_3 + t_2$			



一个简单的代码生成器

语 句	生成的代码	寄存器描述	名字的地址描述
		寄存器空	
$t_1 = a - b$	MOV a, R0 SUB b, R0	R0含 t_1	t_1 在R0中
$t_2 = a - c$	MOV a, R1 SUB c, R1	R0含 t_1 R1含 t_2	t_1 在R0中 t_2 在R1中
$t_3 = t_1 + t_2$	ADD R1,R0	R0含 t_3 R1含 t_2	t_3 在R0中 t_2 在R1中
$d = t_3 + t_2$			



一个简单的代码生成器

语 句	生成的代码	寄存器描述	名字的地址描述
		寄存器空	
$t_1 = a - b$	MOV a, R0 SUB b, R0	R0含 t_1	t_1 在R0中
$t_2 = a - c$	MOV a, R1 SUB c, R1	R0含 t_1 R1含 t_2	t_1 在R0中 t_2 在R1中
$t_3 = t_1 + t_2$	ADD R1,R0	R0含 t_3 R1含 t_2	t_3 在R0中 t_2 在R1中
$d = t_3 + t_2$	ADD R1,R0	R0含d	d在R0中
	MOV R0, d		d在R0和内存中



一个简单的代码生成器

□ 前三条指令可以修改，使执行代价降低

修改前

MOV a, R0

SUB b, R0

MOV a, R1

SUB c, R1

...

修改后

MOV a, R0

MOV R0, R1

SUB b, R0

SUB c, R1

...



一个简单的代码生成器

□ 为特殊语句产生代码

■ 变址和指针语句

变址与指针运算的三地址语句的处理和二元算符的处理相同

语句	i在寄存器Ri中		i在内存Mi中		i在栈中	
	代码	代价	代码	代价	代码	代价
$a = b[i]$	MOV b(Ri), R	2	MOV Mi, R MOV b(R), R	4	MOV Si(Rs), R MOV b(R), R	4
$b[i] = a$	MOV a, b(Ri)	3	MOV Mi, R MOV a, b(R)	5	MOV Si(Rs), R MOV a, b(R)	5



一个简单的代码生成器

□ 为特殊语句产生代码

- 变址和指针语句

- 条件语句

- 根据寄存器的值是否为下面六个条件之一进行分支

- 负、零、正、非负、非零和非正

例, `if x < y goto z`

- 把 `x` 减 `y` 的值存入寄存器 `R`

- 如果 `R` 的值为负, 则跳到 `z`



一个简单的代码生成器

□ 为特殊语句产生代码

- 变址和指针语句
- 条件语句

□ 用**条件码**表示计算结果或装入寄存器的值是负，零还是正
例：若if $x < y$ goto z

- **CMP** x, y
- **CJ** < z

```
int a, b, c;
int main(){
    a = b + 4;
    if ( a < b )
        c = a;
    else
        c = b;
}
```

```
movl    b, %eax
addl    $4, %eax
movl    %eax, a
movl    a, %edx
movl    b, %eax
cmpl    %eax, %edx
jge     .L2
```

16位程序状态字寄存器PSW

CF(进位标志位)

ZF零标志位

SF符号标志位

OF溢出标志位

PF奇偶标志

AF辅助进位标志:

SF=OF, >=跳转



5. 寄存器分配算法

- ☐ 线性扫描算法
- ☐ 图着色算法
- ☐ LLVM中的寄存器分配



线性扫描算法

给定一个函数中变量的**活跃区间**，该算法将线性扫描所有活跃区间，并以**贪心方式**将寄存器分配给变量。

□ 术语

- **活跃区间**live interval: 假设 IR 的指令按数字编号，变量 v 的活跃区间就是 v 被使用的第一条指令的编号 i 以及 v 最后一次被使用的指令编号 j 构成的区间 $[i, j]$
- **激活表**active list: 表示已经分配了寄存器的各活跃区间的表，表中各活跃区间按照结束位置递增的顺序排列

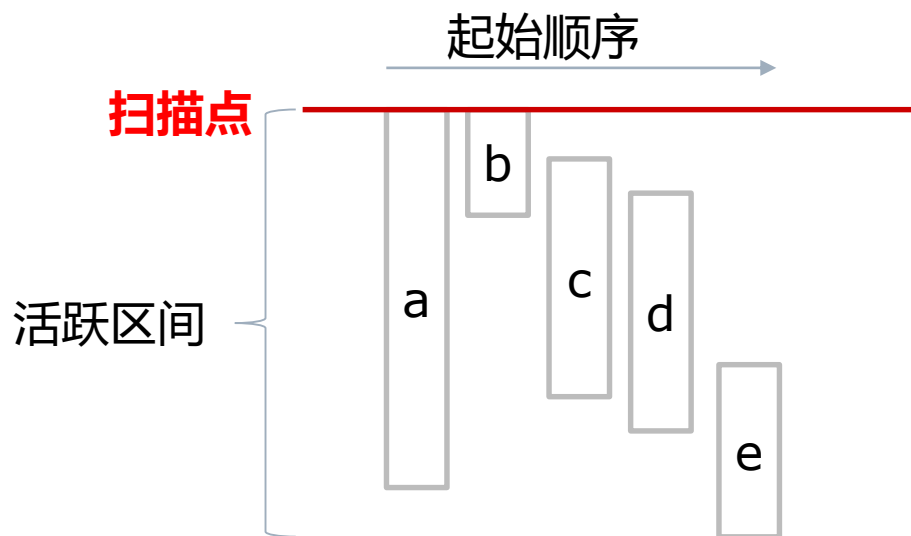
[TOPLAS1999] Linear Scan Register Allocation



线性扫描算法

□ 算法

- 将所有活跃区间按照起始位置先后排序
- 线性扫描所有活跃区间，为变量分配寄存器
- 当没有空闲寄存器可分配时，**溢出结束位置距当前程序点最远**的活跃区间对应的变量



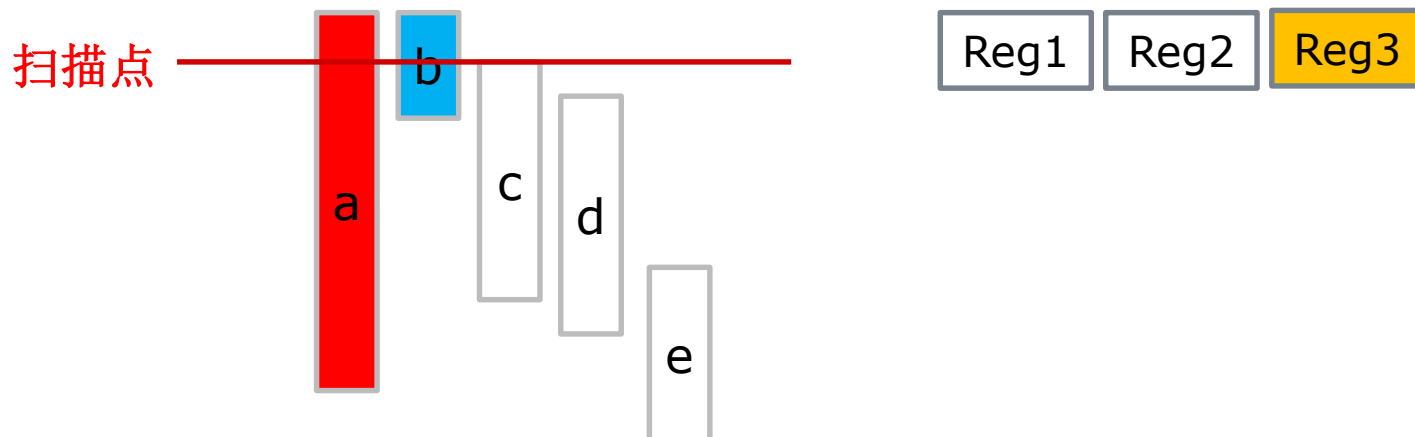
Free Registers



- 为变量a分配寄存器1
- 为变量b分配寄存器2



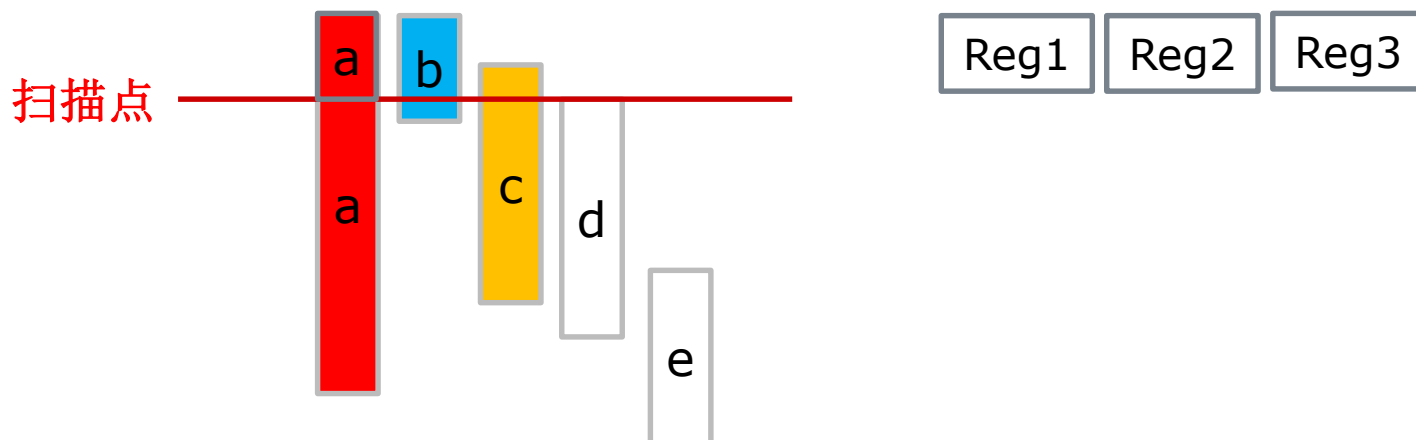
线性扫描算法



■ 为变量c分配寄存器3



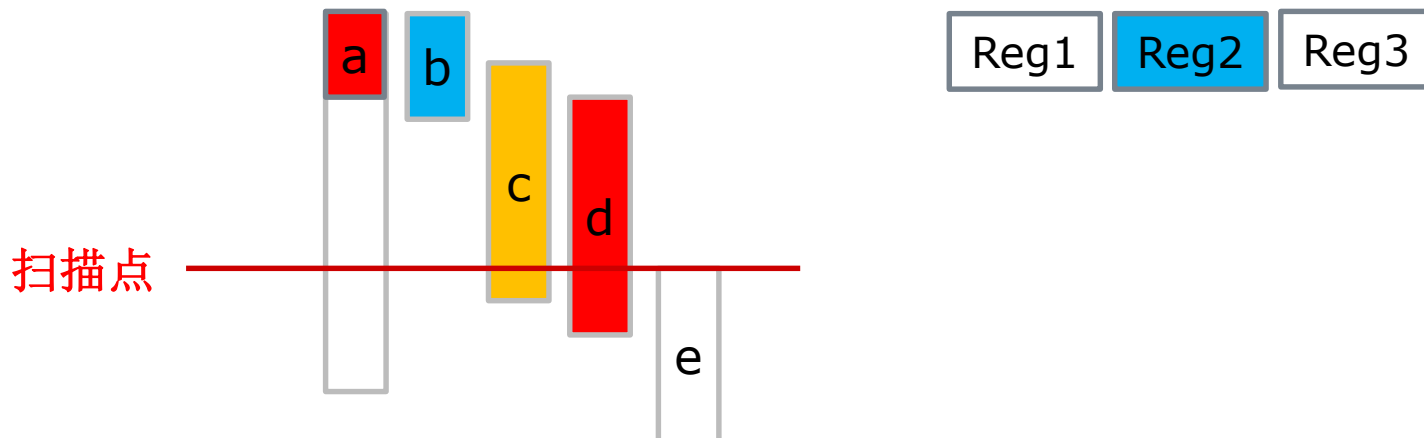
线性扫描算法



- 无空闲寄存器，溢出距离当前程序点最远的变量a，为变量d分配寄存器1



线性扫描算法



- 变量b 活跃区间结束，寄存器2恢复空闲状态
- 为变量e分配寄存器2



线性扫描算法

□ 算法

LINEARSCANREGISTERALLOCATION

```
active  $\leftarrow$  {}  
foreach live interval i, in order of increasing start point  
    EXPIREOLDINTERVALS(i)  
    if length(active) = R then  
        SPILLATINTERVAL(i)  
    else  
        register[i]  $\leftarrow$  a register removed from pool of free registers  
        add i to active, sorted by increasing end point
```

EXPIREOLDINTERVALS(*i*)

```
foreach interval j in active, in order of increasing end point  
    if endpoint[j]  $\geq$  startpoint[i] then  
        return  
    remove j from active  
    add register[j] to pool of free registers
```

SPILLATINTERVAL(*i*)

```
spill  $\leftarrow$  last interval in active  
if endpoint[spill] > endpoint[i] then  
    register[i]  $\leftarrow$  register[spill]  
    location[spill]  $\leftarrow$  new stack location  
    remove spill from active  
    add i to active, sorted by increasing end point  
else  
    location[i]  $\leftarrow$  new stack location
```

局限性：活跃区间是粗粒度的
假设一个变量只在某个程序开头和结尾被使用，则此变量的活跃区间会是整个程序运行区间



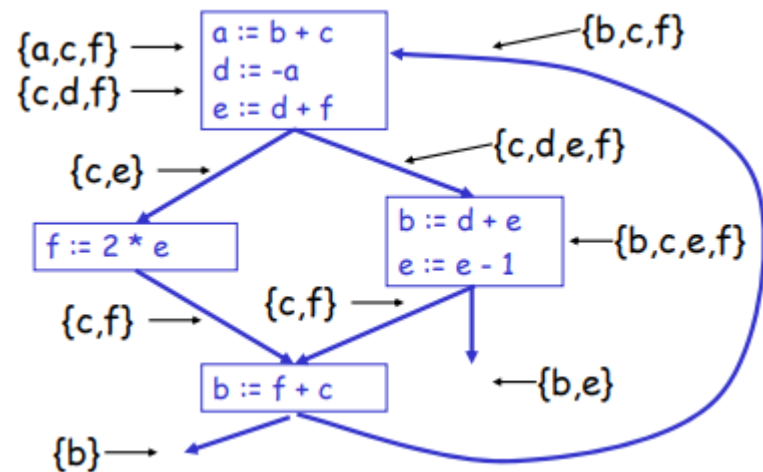
图着色算法

□ 图着色算法

- 变量被赋予不同的节点，在同一个block内同时活跃的变量之间连边，表示不能被分配同一个寄存器
- 对构造出的图进行k着色，k为空闲寄存器的个数
- 按照着色结果对变量进行寄存器赋值

□ 实现：可参考[这里](#)

- 计算每个程序点的活跃变量



图着色算法

□ 图着色算法

□ 实现：可参考[这里](#)

■ 计算每个程序点的活跃变量

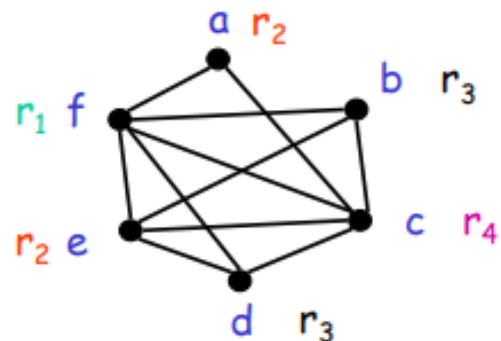
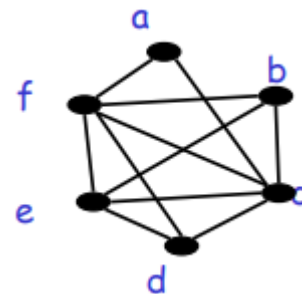
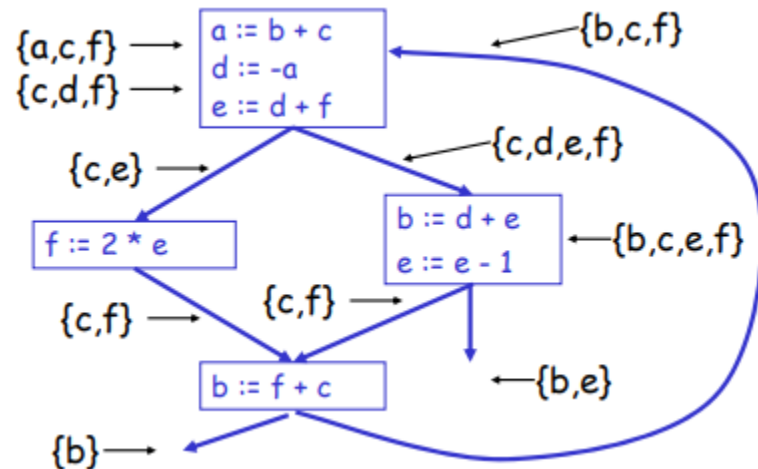
■ 构造寄存器干涉图

(RIG, register interference graph)

□ 顶点：(临时)变量

□ 边(t1,t2)：t1和t2同时活跃

■ 运用图着色算法给每个顶点分配颜色（此处为寄存器）



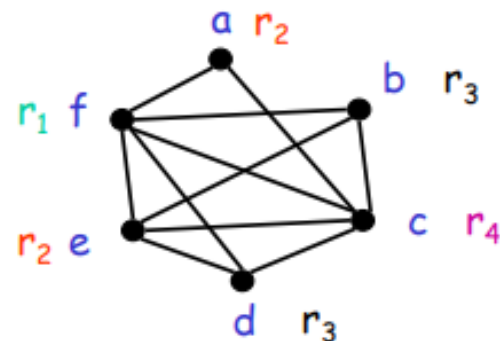
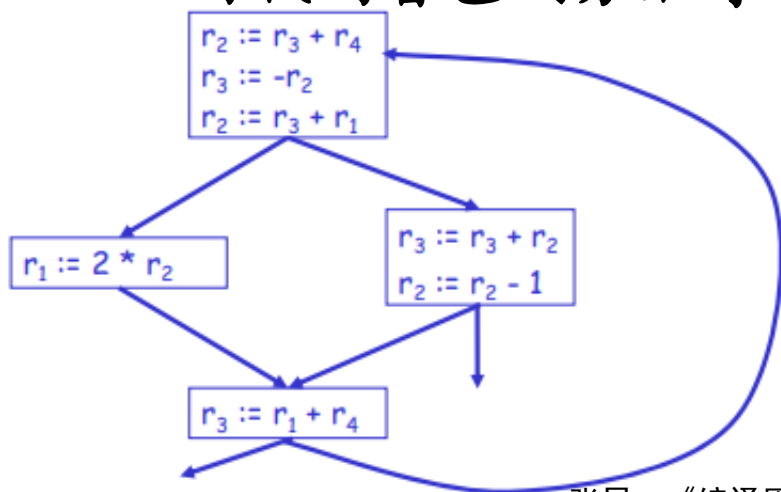
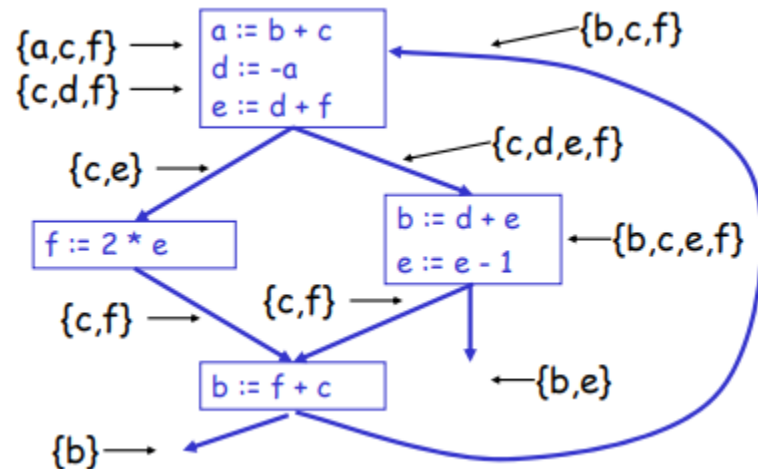


图着色算法

□ 图着色算法

□ 实现：可参考[这里](#)

- 计算每个程序点的活跃变量
- 构造寄存器干涉图
- 运用图着色算法给每个顶点分配颜色（此处为寄存器）
- 对代码着色（分配寄存器后）





□ 挑战

- 实际的目标平台上，寄存器总是偏少的
- graph-coloring思路本身只解决一个问题：
 当有 k 种颜色（ k 个可用寄存器）的时候，我们的程序是否可以**不溢出 (spill)** 就完成着色(寄存器分配)，如果是的话，这个分配是怎样的？
- 它不能解决更重要的**spill问题**



LLVM中的寄存器分配算法

- **Basic**: 线性扫描算法的改进，使用启发式的顺序对寄存器进行生存期赋值
- **Fast**: 顺序扫描每条指令，对其中的变量进行寄存器分配，当没有寄存器可以分配时，选择**溢出代价***最小的寄存器进行溢出操作
- **Greedy**: 线性扫描算法的改进，Basic分配器的高度优化的实现，合并了全局生存期分割，努力最小化溢出代码的成本
- **PBQP**: 基于分区布尔二次编程（PBQP）的寄存器分配器。其工作原理是构造一个表示寄存器分配问题的PBQP问题，使用PBQP求解器解决该问题，并将该解决方案映射回寄存器分配



本章小结

