中国人民大学
RENMIN UNIVERSITY OF CHINA

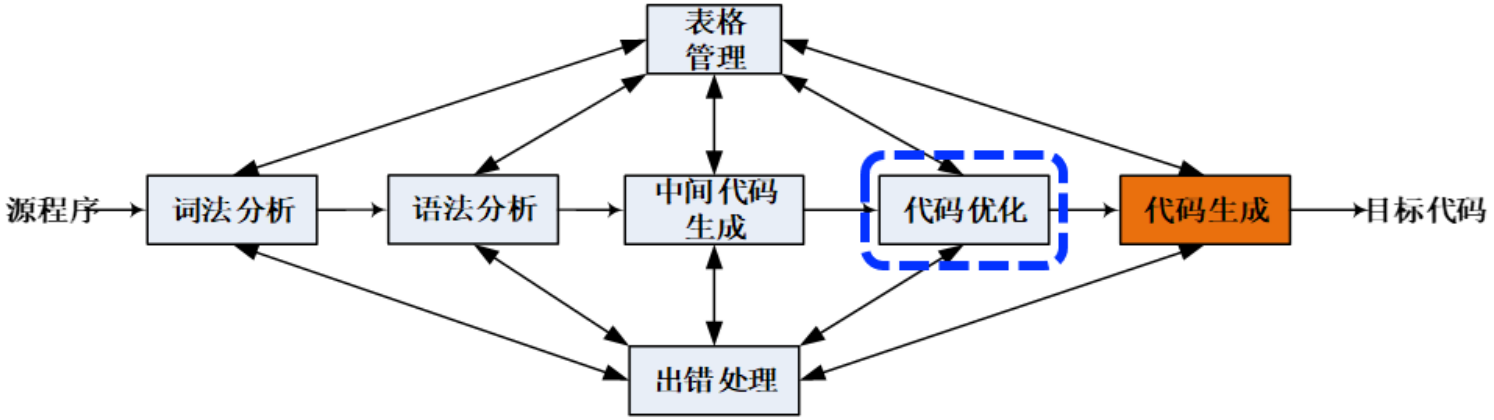
编译原理

--目标代码生成与优化

刘爽
中国人民大学信息学院

目标代码生成

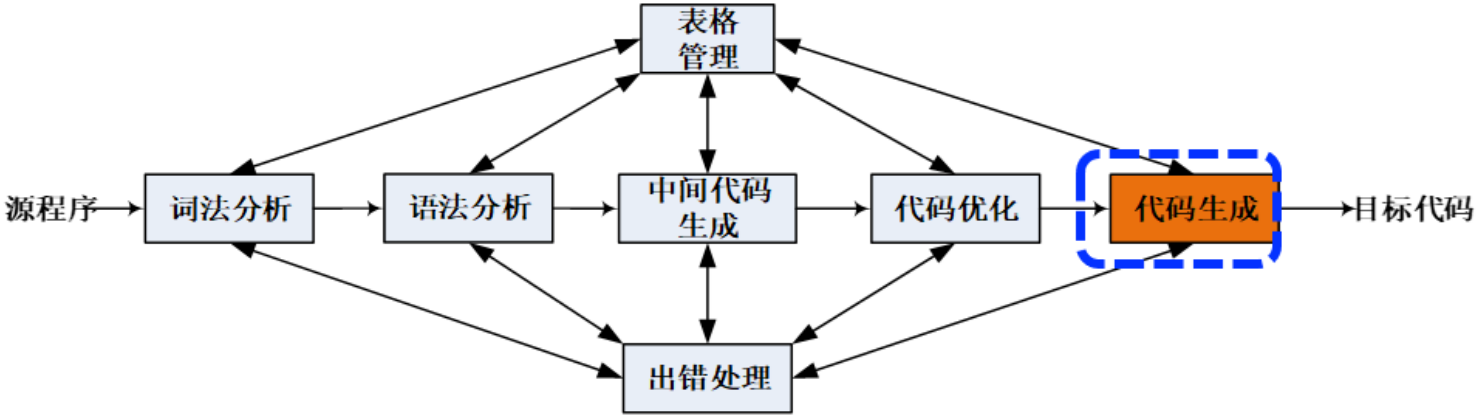
- 编译过程

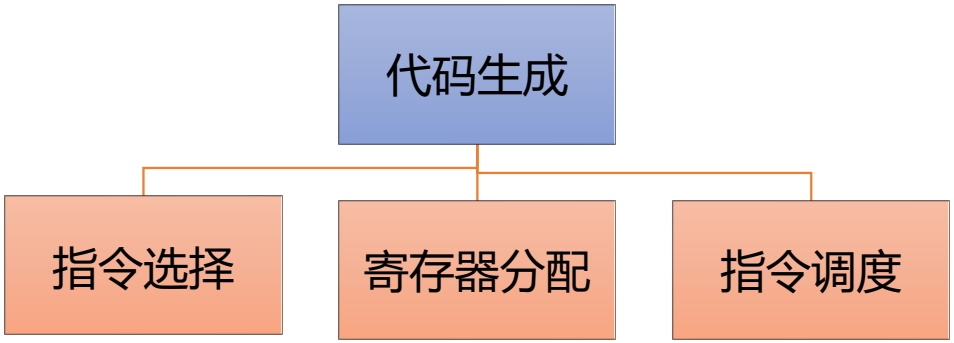


  - 将中间代码翻译成目标代码
  - 与目标机器相关

目标代码生成涉及的问题

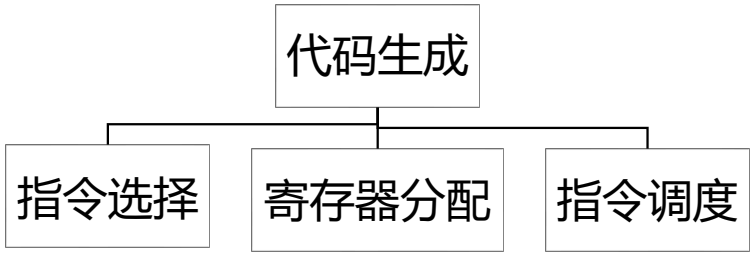
- 编译过程





- 将中间代码翻译成目标代码
- 与目标机器相关

目标代码生成涉及的问题



- 指令选择
  - 将中间代码翻译成等价的目标机指令集指令序列，实现中间代码到目标机指令的映射
- 寄存器分配
  - 对于load/store架构的目标机是必须要考虑的问题
- 指令调度
  - 通过重排指令，减少流水线中的停顿，最大化指令级并行

内容

- 微体系结构简介
- RISC-V架构简介
- 指令选择
- 寄存器分配
- 指令调度

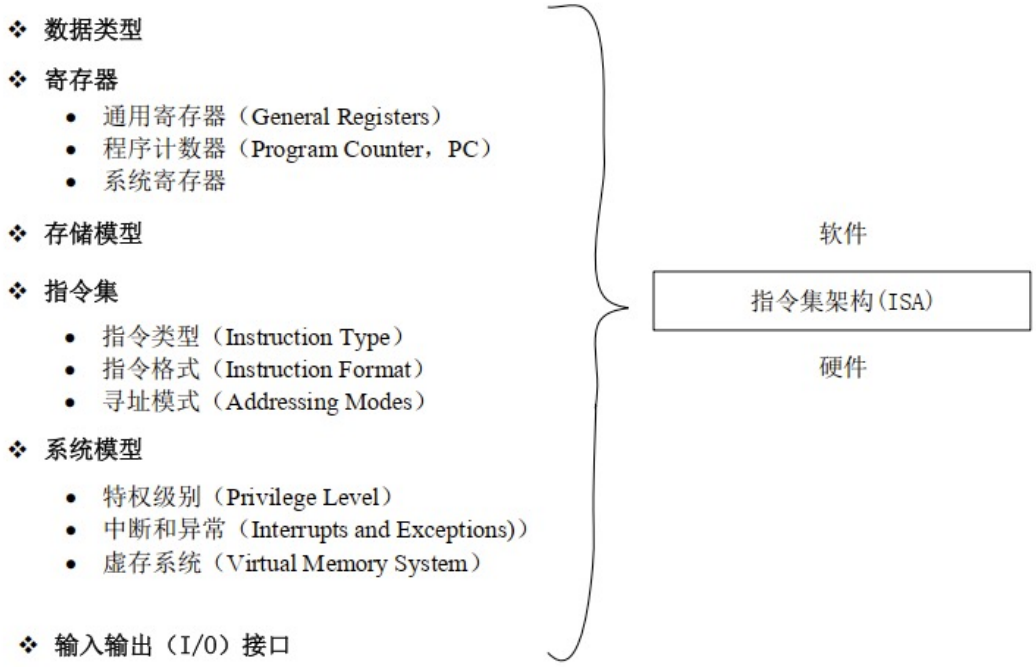
一、微处理器体系结构简介

- 代码生成器需要感知目标机的相关信息
  - 微处理器的指令集架构（Instruction Set Architecture，ISA）
  - 针对处理器的微架构（Microarchitecture）进行优化设计

- ISA是程序员看到的处理器的属性，包括概念性结构和功能特性
- 微架构则指处理器的硬件组织和结构
  - 流水线（Pipeline）
  - 存储层次
  - 多核多线程等
- 同一种指令集架构可以有多种微架构实现

1. ISA

- 一个抽象层，该抽象层在多个方面定义了架构规范



- 构成了处理器底层硬件与运行于其上的软件之间的桥梁与接口
- 软件开发人员和硬件设计人员必须共同遵循这些规范，从而使软件无须做任何修改就可以运行在任何一款遵循同一ISA规范的处理器上

1. ISA

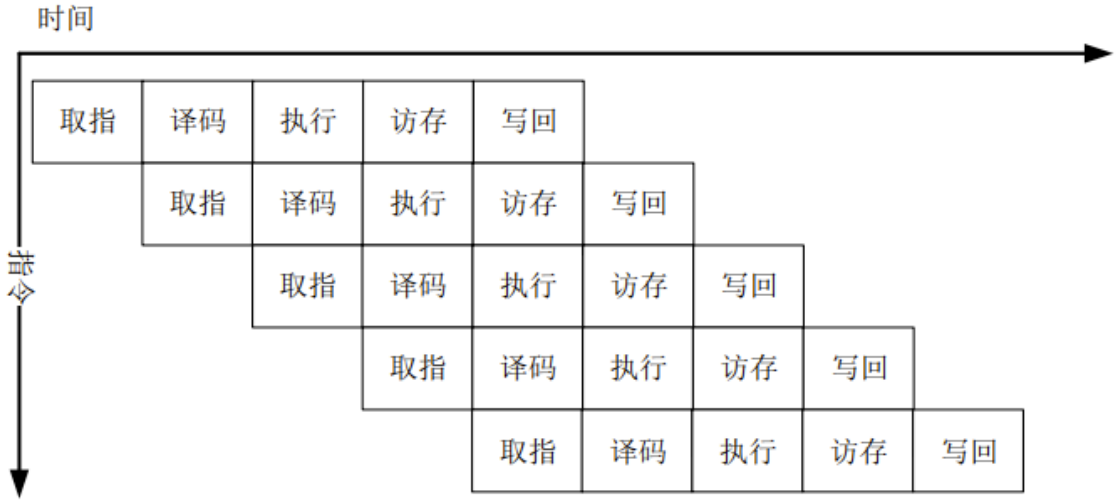
- 通用寄存器体系结构
  - 指令操作数字段编码的要么是寄存器编号，要么是存储地址，少数操作数字段可以是立即数。
  - 寄存器-存储器型
    - ISA中除存取（load/store）指令外，其他很多指令也可以访问存储器
    - x86架构；
  - 载入-存储（load-store）型
    - 只有存取（load/store）指令才能访问存储器
    - 例如RISC-V，ARM架构等

1. ISA

- 按照指令集复杂度划分
  - 复杂指令集（Complex Instruction Set Computer，CISC）
    - 指令长度可变，指令类型复杂，不仅包含处理器常用的指令，还包含了许多不常用的特殊指令
    - 典型代表是x86架构
  - 精简指令集（Reduced Instruction Set Computer，RISC）
    - 指令长度固定，指令格式统一，指令类型相对简单，只包含处理器常用的指令，而对于不常用的操作，则通过执行多条常用指令的方式来达到同样的效果
    - 典型代表是RISC-V、ARM

2. 流水线

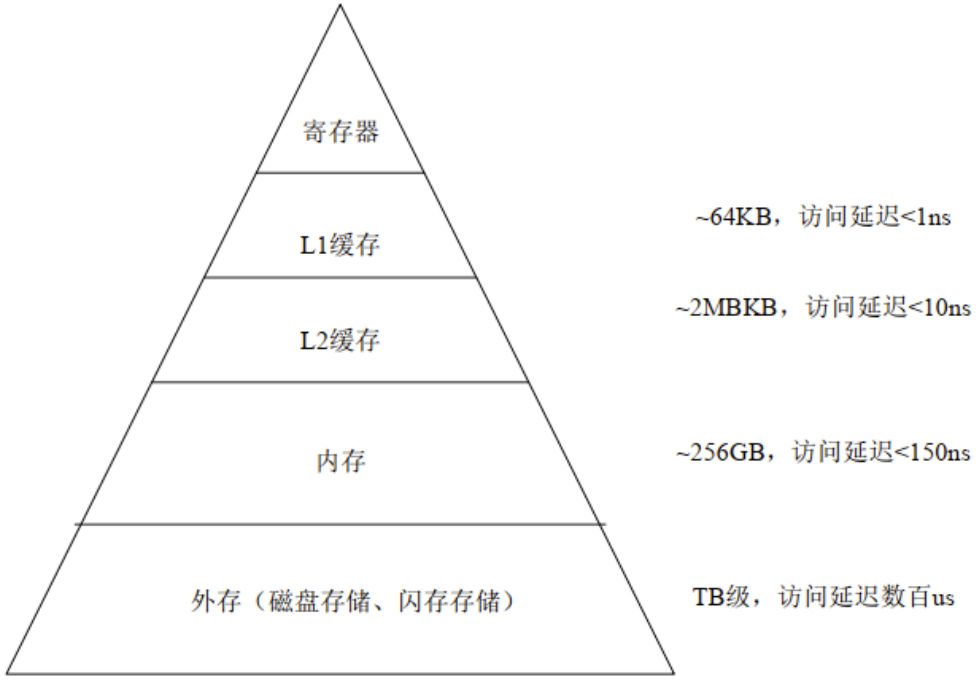
- 采用流水线技术来使多条指令重叠执行，从而实现指令级并行（Instruction Level Parallelism，ILP）
- 通过多个功能部件，使每个时钟周期可以处理多条指令的流水线设计，称为超标量流水线



5级流水线示意图

3. 存储层次

- 多级存储层次



多级存储架构

- 存储层次分为寄存器，多级高速缓存（Cache），内存，外存（磁盘）。
- 位于顶层的寄存器访问速度最快，通常仅为数百皮秒（ps）
- 存储层次越往下，其存储容量越大，访问速度也逐渐降低

二、RISC-V架构

- 精简指令集架构
- 模块化
  - 具有一个最小的指令集，通过模块化的方式实现其他指令的扩展
  - 整数乘除扩展M、浮点单/双精度扩展F/D、向量扩展V
- 针对32位处理器的最小指令集
  - RV32I：只包含47条指令
- 针对64位处理器的最小指令集
  - RV64I：在RV32I的基础上，增加了双字（double word）和长整型（long）指令的支持，并且将所有寄存器扩展到64位。

二、RISC-V架构

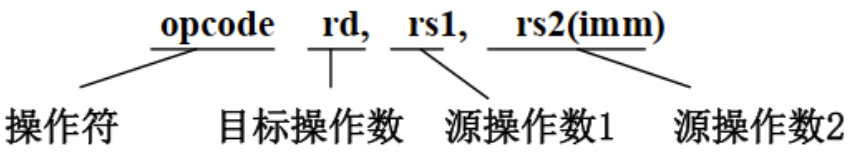
- 32个32位通用寄存器，其中0号寄存器作为常零寄存器

寄存器名	ABI名称	说明
x0	zero	恒为0
x1	ra	链接寄存器，保存函数返回地址
x2	sp	栈指针，指向栈的地址
x3	gp	全局寄存器，用于链接器松弛优化
x4	tp	线程寄存器，用于线程变量空间指针（基地址）
x5~x7	t0~t6	临时寄存器
x8, x9	s0,s1	保存寄存器，如果函数调用使用这些寄存器，需要保存在栈里。s0可用作帧指针（FP）
x10~x17	a0~a7	函数参数寄存器
x18~x27	s2~s11	保存寄存器，如果函数调用使用这些寄存器，需要保存在栈里
X28~x31	t3~t6	临时寄存器

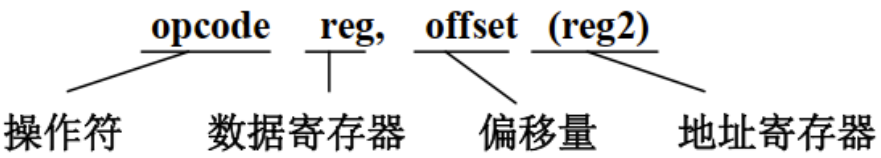
- 独立的寄存器PC
- 一组控制和状态寄存器（CSR）

二、RISC-V架构

- RV32I指令集
  - 运算（算术运算、逻辑运算、移位运算）指令、存取指令、控制传输指令和其他指令。
    - 运算指令最多包含3个操作数：目标操作数（寄存器）rd、源操作数rs1和源操作数rs2。rs1是寄存器，rs2是寄存器或者立即数



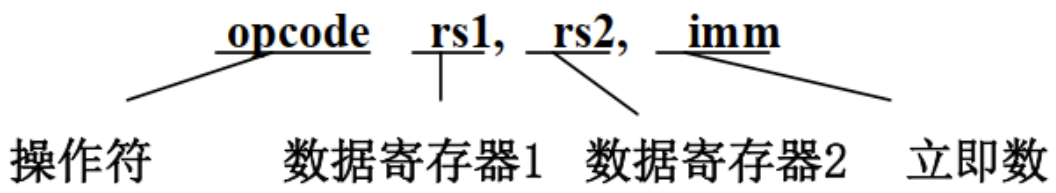
    - 存取指令将数据读出（load）或者保存（store）到内存中。一般情况下，存取指令有2个操作数，数据寄存器reg和内存地址寄存器reg2。



二、RISC-V架构

- RV32I指令集
  - 运算（算术运算、逻辑运算、移位运算）指令、存取指令、控制传输指令和其他指令。

  - 控制传输指令包含有条件分支指令和无条件跳转指令。有条件分支指令包含3个操作数，处理器比较数据寄存器rs1和rs2中的值，如果满足操作符指定的条件，则程序指针指向当前程序计数器（PC）加上立即数imm后所得数值的位置。



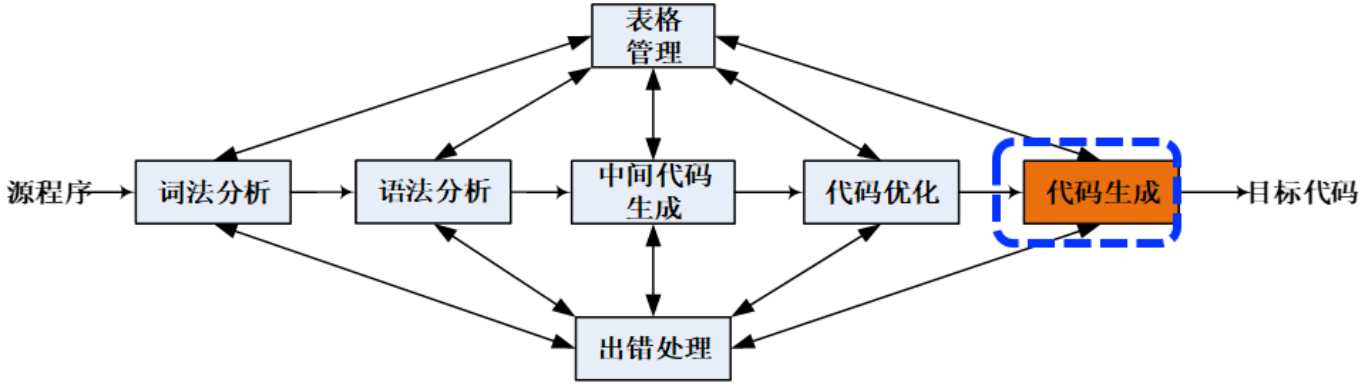
  - 无条件跳转指令包含两条：直接跳转和间接跳转。

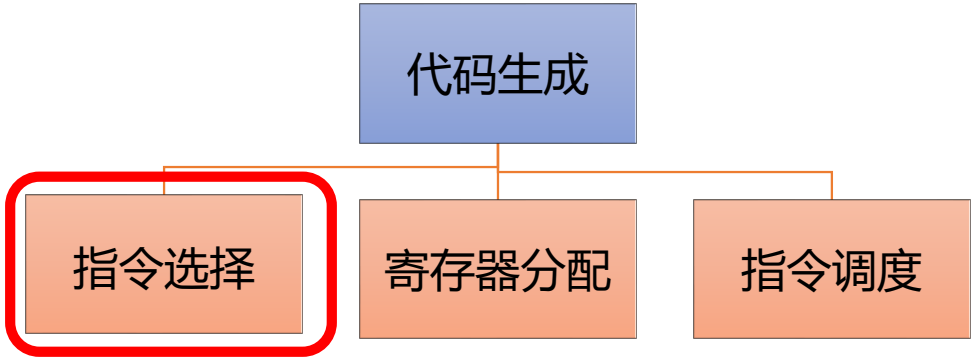
二、RISC-V架构

- 寻址方式是处理器执行指令获取数据地址，或者下一条指令地址的方式
- RISC-V支持四种寻址方式：
  - 立即数寻址
  - 寄存器寻址
  - 寄存器间接寻址
  - 程序计数相对寻址

三、指令选择

- 编译过程

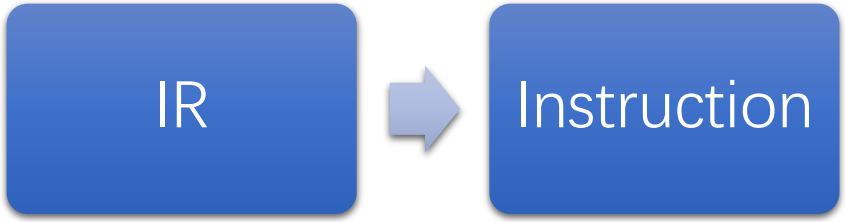




指令选择将中间代码翻译成等价的目标机指令集(ISA)指令序列，实现中间代码到目标机ISA的映射。

1. 什么是指令选择

- 将中间表示(IR)翻译成等价的目标机指令集指令序列的过程
  - 编译器前端和优化器都运行在代码的IR形式上
  - IR代码必须翻译成目标机指令序列，才能在目标机上执行
  - 指令选择实现IR到目标机指令的映射



1.1 指令选择的目标

- 生成代价(即成本)最小的指令序列
  - 执行时间最短
  - 代码长度最短
  - 能耗最低
  - 其他目标
    - 如果处理器上有空闲功能单元，选择生成该功能单元的指令，则该指令的执行实际是“免费”的（与指令调度有部分重叠）

1.2 指令选择的复杂性

- 指令选择的搜索空间巨大
  - 目标机ISA有大量备选指令序列可达到同一语义效果
    - x86上设置eax寄存器为0 ***mov eax,0*** ***xor eax,eax*** ***sub eax,eax*** ***imul eax,0***
    - RV32IM上计算r=r*2 ***add x5,x5,x5*** ***sll x5,x5,#1*** ***muli x5,x5,#2***
    - RISC机器：每个IR操作对应1~2条ISA指令
    - CISC机器：可能需要将几个IR操作汇聚为一条ISA指令
  - 灵活的寻址模式使搜索空间变大
    - 一条指令可以同时完成寄存器算术运算和地址计算
  - 某些ISA对特定操作增加了附加约束

1.3 指令选择的基本思想

- 模式匹配(Pattern-Match)进行指令选择
  - 基于模式匹配技术选择与一段IR匹配的指令
  - 直到获得覆盖全部IR的指令序列
- 指令选择方法包括
  - 基于树模式匹配的指令选择
  - 基于窥孔匹配的指令选择
- 指令选择阶段假设有足够多的符号寄存器
  - 到寄存器分配阶段再考虑符号寄存器到物理寄存器的分配

2. 低层次的中间表示

- 回顾：中间表示
  - 高层次中间表示(HIR)
    - 靠近源语言，更多上下文信息用于进行高层次优化
  - 中层次中间表示 (MIR)
    - 中端编译优化
  - 低层次中间表示(LIR)
    - 靠近机器，用于进行低层次优化
    - 更容易映射为目标机指令
- 针对不同的中间语表示，有不同的指令选择方法
- 在指令选择之前，可以将中间代码转换为更底层的表示

2. 低层次的中间表示

- LLVM在指令选择之前，将LLVM IR转换为DAG
  - 与目标机无关的低级IR
  - DAG节点表示指令或者一个操作数
  - 边描述了指令间存在数据依赖关系，每个基本块对应一个DAG
  - 基于DAG采用模式匹配进行指令选择

2.1 LIR示例

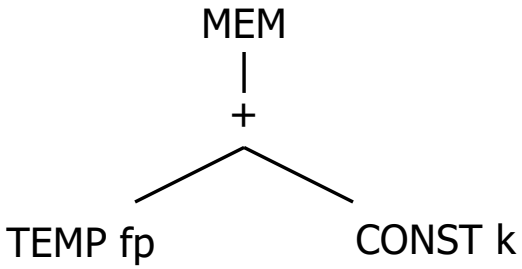
- 一个化简的LIR示例
  - 支持的表达式
    - **CONST**(i) – 整型常量i
    - **TEMP**(t) – 临时变量t (符号寄存器)
    - **BINOP**(op, e1, e2) – e1 op e2操作
    - **MEM**(e) – 地址e处的内存内容
    - **CALL**(f, args) – 调用函数f，参数列表为args

2.1 LIR示例

- 支持的语句
  - **MOVE**(TEMP t, e) – 将e存储到临时变量t中
  - **MOVE**(MEM(e1), e2) – 将e2存储到e1指示的内存中
  - **EXP**(e) – 执行表达式e，忽略结果
  - **SEQ**(s1, s2) – 先执行s1，再执行s2
  - **NAME**(n) – 汇编指令的标号n
  - **JUMP**(e) – 跳转到e处, e可以是一个NAME标号或更复杂的情况 (如switch)
  - **CJUMP**(op, e1, e2, t, f) – 执行e1 op e2; 若结果为真则跳转到标号t, 否则跳转到标号f
  - **LABEL**(n) – 确定标号n在代码处的位置

2.1 LIR示例

- 栈上，栈帧指针fp+偏移k处的变量
  - 树IR



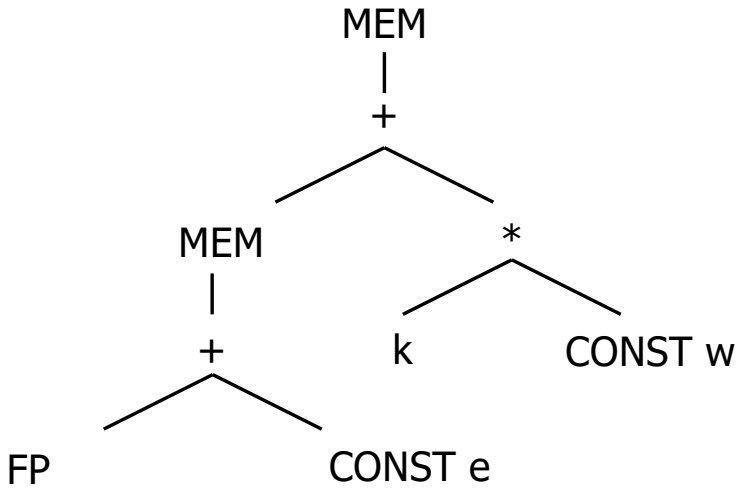
  - 线性IR

MEM(BINOP(PLUS, TEMP fp, CONST k))

- 一个类RISC机器上的低层次中间表示
  - 3地址码，寄存器-寄存器指令 + load/store指令
    - r1 ← r2 op r3

2.1 LIR示例

- 栈上数组e下标为k的元素e[k]，存储每个元素需要w个字节
  - 树IR

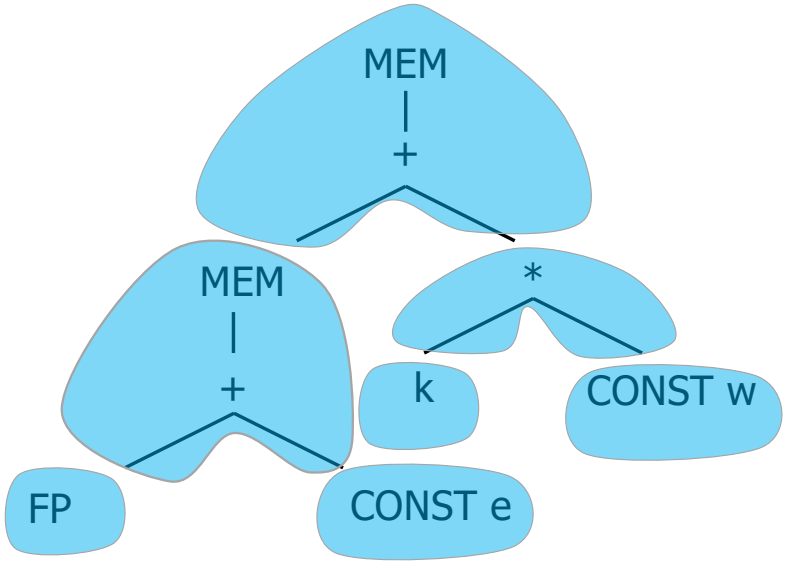


  - 线性IR

MEM(BINOP(PLUS, MEM(BINOP(PLUS, TEMP fp, CONST e)), BINOP(MUL, k, CONST w)))

3.1 树模式匹配方法

- IR树
  - IR树中的一段树枝，称之为树型 (tree pattern)
  - 树型可以看成是树中的一个节点及其子节点

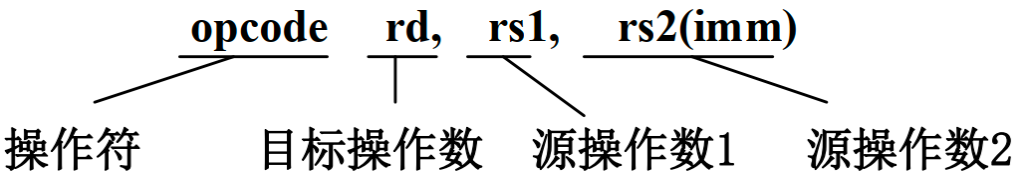


3.1 树模式匹配方法

- 操作树
  - 为了方便进行模式匹配，我们把目标机指令也表示为树，称之为操作树或瓦片 (tile)
- 基于树模式匹配的指令选择
  - 给定一个IR树和一组操作树，将操作树平铺(tiling)到IR树上
    - 用操作树的最小集合来平铺一个IR树，并且这组操作树不重叠

3.2 操作树

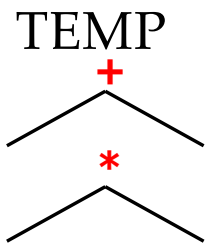
- 为将指令选择转换为树模式匹配问题，目标机指令表示为操作树
- RV32IM算术运算指令



① ri

② add rd, rs1, rs2

③ mul rd, rs1, rs2



temp节点作为寄存器ri使用

3.2 操作树

- 立即数指令

④ addi rd,rs1,imm

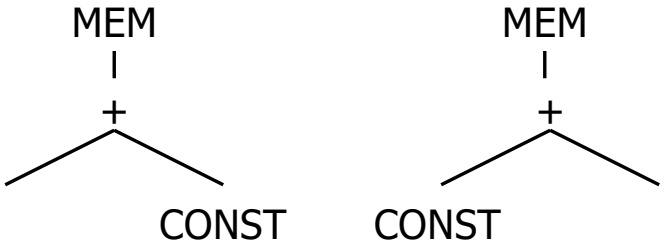
⑤ muli rd, rs1,imm

3.2 操作树

- 访存指令(包含寻址)

⑥ lw r1, offset(r2)

r1 = MEM(r2+offset)

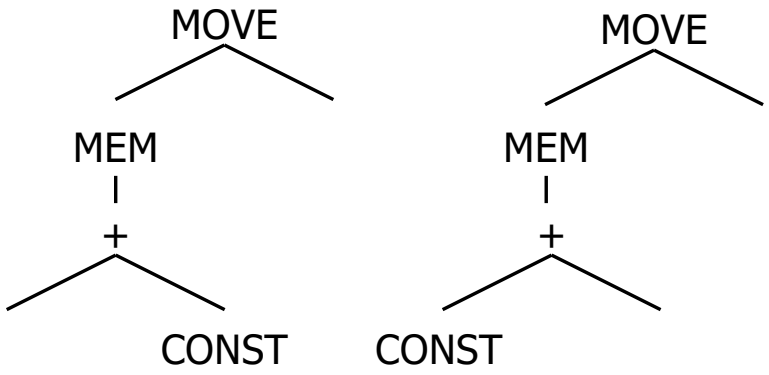


3.2 操作树

- 访存指令(包含寻址)

⑦ sw r1, offset(r2)

- MEM[r2+offset] = r1



3.3 平铺方案

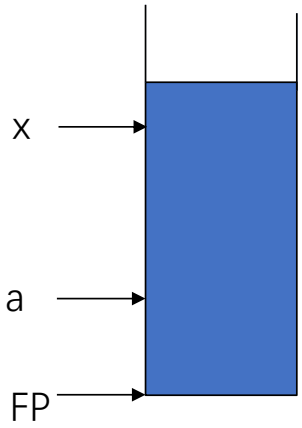
- 平铺方案是一组<node,op>对
  - node是IR树中的一个节点
  - op是一个操作树 (瓦片)
  - 一个<node,op>对表示op对应的目标机指令可以覆盖node为根的子树

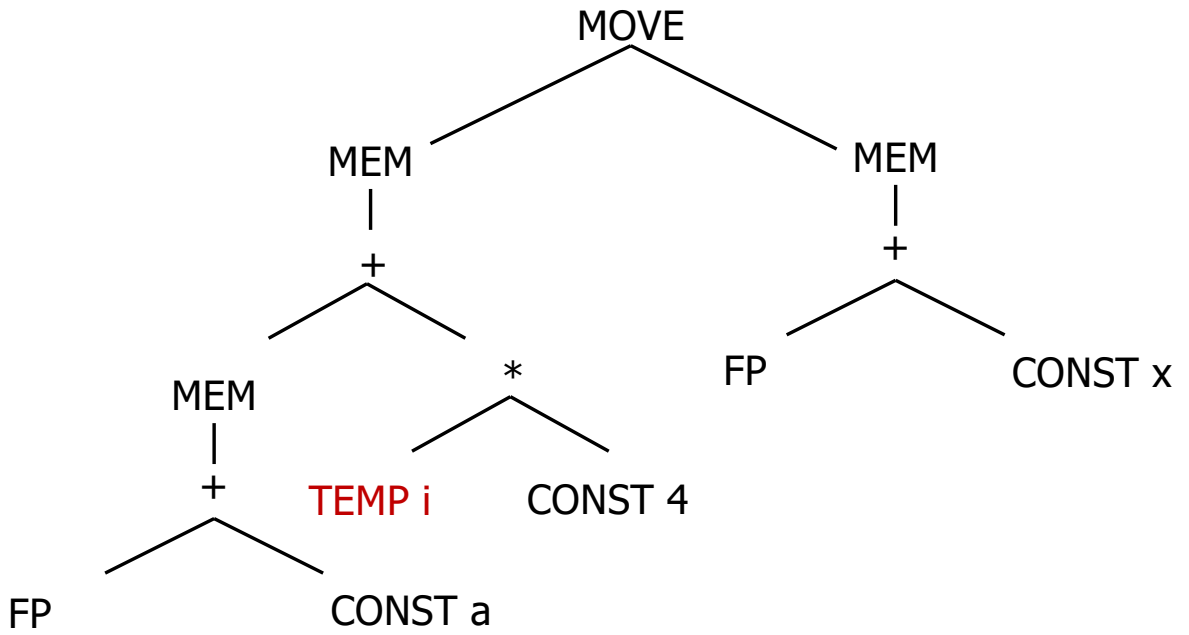
3.3 平铺方案

- 如果一个平铺方案覆盖了一个IR树中的每一个node，并且每个op树(瓦片)都与其邻居op树相连接，则该平铺方案实现对该IR树的覆盖
- 对于一个<node,op>对，如果其node被平铺方案中另一个op树的一个叶节点涵盖，或者该node是IR树的根节点，那么称该op树与其邻居op树相连接(两个瓦片相连接)
  - 当两个op树相连接时，两者公共节点的存储类别必须一致，否则无法将正确的值从较低的树传到较高的树
    - 如都存储于寄存器中

3.3 平铺方案

- 例子: a[i]=x
  - 假设a是栈上的数组，i是寄存器变量，x是栈上的变量（a实际是指向数组的指针的栈帧偏移，x实际是x的栈帧偏移）

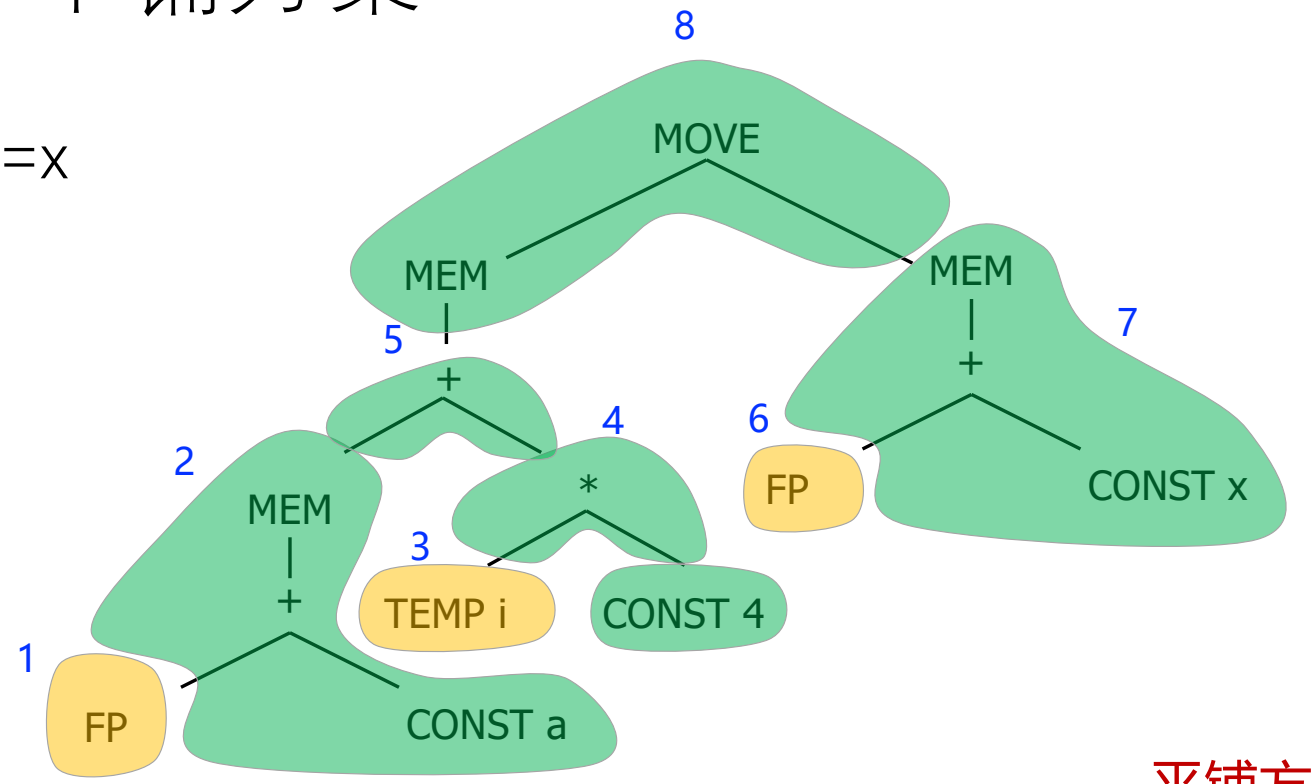




IR树

3.3 平铺方案

- a[i]=x



平铺方案

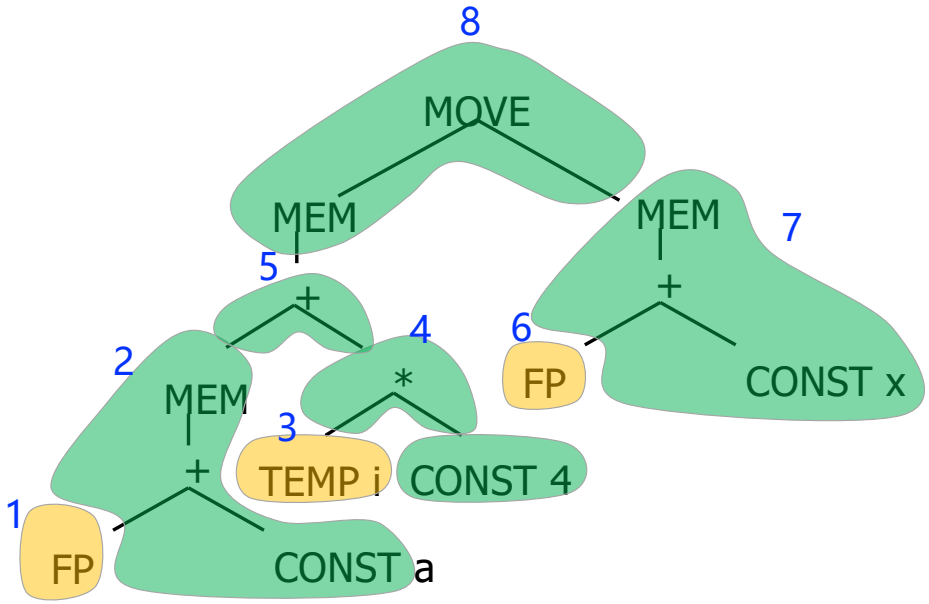
3.4 代码生成

- 基于IR树的平铺方案，进行代码生成
  - 后根次序自底向上遍历瓦片
  - 按瓦片顺序，基于重写规则，生成目标机指令序列
  - 使用相同寄存器名将相连瓦片在重叠位置连接在一起
- 重写规则
  - 一条重写规则由瓦片，代码模板和相关成本组成

规则	树型	成本	代码模板
(1)	Reg → +(Reg1, Reg2)	1	add r1, r2, r3
(2)	Reg → M (+(Reg1, Num2))	2	lw r1, offset(r2)

3.4 代码生成

- 1、3、6子树不对应任何指令，是已经含有正确值的寄存器
- 生成指令序列 (通过相同寄存器名连接瓦片)




```
2    lw    r1, a(fp)   //读取a的地址
4    muli  r2, ri , 4  //临时变量i在ri
5    add   r1, r1, r2  //计算a[i]地址
8    lw    r2, x(fp)
9    sw    r2, (r1)
```

3.5 如何找到平铺方案

- IR树的每个节点可能有多个匹配的瓦片，可以生成多个平铺方案
- 关键是如何快速找到一个良好的平铺方案，满足指令选择的目标 （如程序执行时间最短）
- 从两个层次看这个问题
  - 通过树遍历，找到所有的平铺方案
  - 将瓦片与成本关联，通过累加成本，找到在每个节点处最低成本的匹配

3.5 如何找到平铺方案

- Tile(node n)算法
  - 为IR树中以n为根节点的子树找到平铺方案
  - Num(n)标注与节点n匹配的所有瓦片的集合(即与节点n匹配的所有重写规则的集合)
  - 假设IR树中每个节点至多有两个子节点，且一个重写规则的右侧至多有一个操作
  - 按后跟次序遍历IR树，确保标注节点n之前先标注其子节点

3.5 如何找到平铺方案

- Tile(node n)算法

```
Num(n) = ∅;  //按后根次序遍历IR树，Num集合初始为空
//按照后跟顺序遍历IR树
if n has two children then //二元操作
    Tile(left(n))    //递归遍历左子树
    Tile(right(n))   //递归遍历右子树
    for each rule r that implements n //对能够实现节点n指定的操作的每个规则r
        if  (left(r) ∈ Label(left(n)) && right(r) ∈ Label(right(n)) )
            Num(n) = Num(n) ∪ {r} //如果左右子树均匹配，则将r加入到Num(n)
else if n has one child then //一元操作原理同二元操作
    Tile(left(n))
    for each rule r that implements n
        if  (left(r) ∈ Label(left(n)))
            Label(n) = Label(n) ∪ {r}
else //n是叶子节点
    Label(n) = { all rules that implement n }
```

3.5 如何找到平铺方案

- Tile算法可以从整个操作树集合(重写规则集合)中找到所有可能的平铺方案
- 如何快速找到最低成本的平铺方案?
  - 从所有匹配的平铺方案中找到最低成本匹配（低效）
  - 利用动态规划方法找到最低成本匹配（高效）

3.5 如何找到平铺方案

- 利用动态规划方法找到最低成本匹配
  - 在Tile算法的基础上考虑每个瓦片(重写规则)的成本
  - 在自底向上遍历中，计算每个节点的成本，选择最低成本的匹配
    - 成本=与n匹配的规则的成本 + n的所有子树的成本之和
    - 要考虑操作数的存放位置（如寄存器，内存或常量立即数）
  - 通过累加成本并在每个节点处选择最低成本的匹配，保证生成局部最优，全局低成本的平铺方案

4. 基于窥孔匹配的指令选择方法

- 窥孔优化(Peephole Optimization)
  - 编译器使用滑动窗口(窥孔)在代码上移动
  - 每次仅考察窥孔中的指令序列(一小段相邻指令序列)
  - 寻找可以改进的特定模式
  - 识别出一个模式时，使用更好的指令序列重写该模式
- 快速高效：有限的模式集合+有限的关注区域

4.1 窥孔优化例子

- e.g 调用store指令后调用load指令, 或调用push调用pop

original	improved
mov [ebp-8],eax mov eax,[ebp-8]	mov [ebp-8],eax
push eax pop eax	---

4.1 窥孔优化例子

- 简单的代数恒等式

original	improved
add eax,0	---
add eax,1	inc eax
mul eax,2	add eax,eax
mul eax,4	shl eax,2

4.1 窥孔优化例子

- 跳转指令

original

```
        jmp  here

here:  jmp there
```

improved

```
jmp there
```

4.2 窥孔匹配实现

- 早期实现
  - 一组有限的手工编码模式
  - 窗口很小（通常只有2、3条指令）
  - 通过穷举搜索进行模式匹配
  - 有限的模式+小窗口，保证了算法的高效运行
- 现代实现
  - 日益复杂的指令集驱动更系统化的处理方法
  - 处理过程划分为三个任务：展开、简化和匹配
  - 使用符号解释(symbolic interpretation)和简化的系统化应用

4.2 窥孔优化实现

- 展开
  - 识别IR形式的输入代码，重写为一系列底层IR(LLIR, Low-Level IR)操作
  - 该LLIR需表示原IR操作的所有直接影响
    - 如设置add操作的条件码
  - 基于模板进行重写，各操作逐一展开，无需考虑上下文

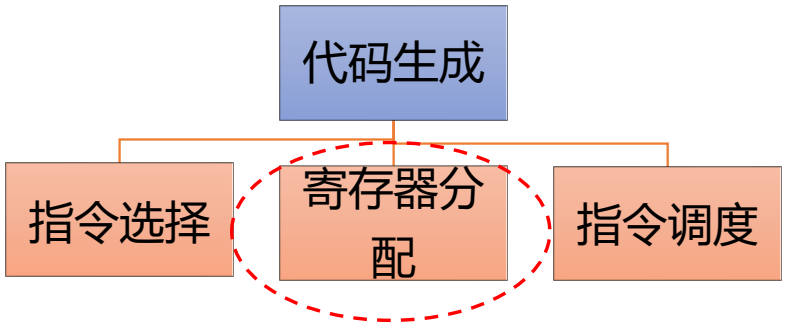
4.2 窥孔优化实现

- 简化
  - 通过一个小滑动窗口对LLIR进行一遍处理
  - 以系统化方式进行局部优化
    - 前向替换
    - 代数化简（x+0 → x）
    - 常量传播
    - 死代码消除
  - 简化程序是处理过程的核心

4.3 窥孔优化实现

- 匹配
  - 对着模式库比较简化过的LLIR，寻找能够以最低成本保留住LLIR中所有效应的模式
    - 保证正确性的效应必须保留
    - 可以有新的不影响正确性的效应
  - 输出目标机指令(通常是汇编码)

目标代码生成涉及的问题



- 指令选择
  - 将中间代码翻译成等价的目标机指令集指令序列，实现中间代码到目标机指令的映射
- 寄存器分配
  - 对于load/store架构的目标机是必须要考虑的问题
- 指令调度
  - 通过重排指令，减少流水线中的停顿，最大化指令级并行

内容

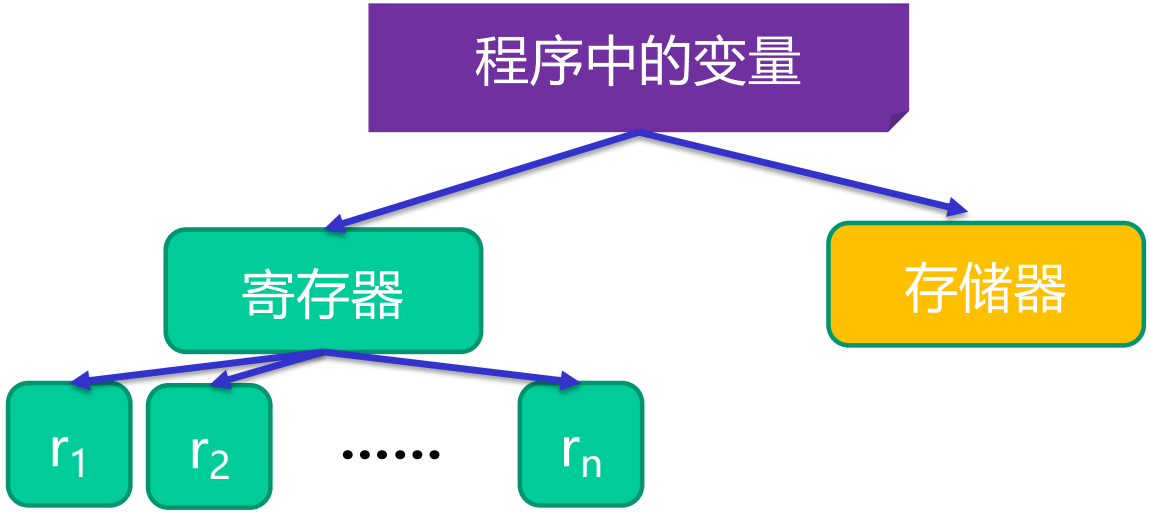
- 微体系结构简介
- RISC-V架构简介
- 指令选择
- 寄存器分配
- 指令调度

四、寄存器分配

- Load-store架构只有存取（load/store）指令才能访问存储器，其他指令的操作数存放在寄存器中，只有少部分指令允许立即数
- 多级存储，寄存器访问速度最快，容量最小
- 寄存器分配
  - 高效、合理地使用有限的寄存器资源
  - 编译器中最重要的优化之一
  - 实验表明采用较优寄存器分配算法生成的代码比将所有变量都保存于存储器中的代码快2.5倍

1.1 什么是寄存器分配

- 将程序中的变量分配到寄存器或者存储器的过程，寄存器分配确定
  - 寄存器分配：哪些变量保存在寄存器中，哪些变量保存在存储器中
  - 寄存器指派：对于保存在寄存器中的变量：具体放在哪个寄存器



1.2 寄存器分配的原则

- 程序变量数目 vs. 寄存器数目
  - 程序中有大量的变量，变量数一般大于寄存器数
- 溢出
  - 将一个变量从寄存器中送回到存储器中的动作叫做寄存器溢出（Spill）
  - 溢出意味着插入一条写指令，并且下次使用该变量时，需要再为该变量分配寄存器，并且通过读指令将值从存储器中读入到指派的寄存器
  - 尽可能减少溢出

1.2 寄存器分配的原则

- 寄存器分配的原则
  - 尽可能地将较多的变量保存在寄存器中
  - 尽可能地将频繁使用的变量保存在寄存器中

  （将那些使用较少的变量存放在存储器中）

- 一个好的编译器应设计出尽可能满足这两个标准的寄存器分配算法

1.3 寄存器分配方法

- 基于使用计数的寄存器分配
  - 20世纪80年代之前使用，如银河-I编译器
- 基于图着色的寄存器分配
  - 突破性进展，20世纪80年代开始流行，如GCC，LLVM
- 基于线性扫描的寄存器分配
  - 1999年提出，如LLVM
- 基于SSA的寄存器分配
  - 2005年提出

2. 基于使用计数的寄存器分配

- 使用计数（Usage Count）
  - 在程序代码中，变量的使用次数

t1 = a – b *t2 = a * c* *t3 = t1 + t2* *a = d + c* *d = t2 + t3*	a, 2 b, 1 c, 2 d, 1 t1, 1 t2, 2 t3, 1

2.1 基于使用计数的局部寄存器分配

- 基于使用计数的局部寄存器分配

1. 以基本块为单位，根据需要依次分配寄存器，需要一个，分配一个
2. 遇到一次使用，引用计数便减1
3. 当计数为0时，寄存器便可再次分配给其它变量
4. 当寄存器不够时，根据启发式溢出一个寄存器

2.1 基于使用计数的局部寄存器分配

- 使用计数（Usage Count）
  - 在程序代码中，变量使用次数

$t1 = a - b$	a, 2
$t2 = a * c$	b, 1
$t3 = t1 + t2$	c, 2
$a = d + c$	d, 1
$d = t2 + t3$	t1, 1
	t2, 2
	t3, 1

$t1^{r2} \leftarrow a^{r1} - b^{r2}$

$t2^{r1} \leftarrow a^{r1} * c^{r3}$

$t3^{r2} \leftarrow t1^{r2} + t2^{r1}$

$a \leftarrow d^{?} + c^{r3}$

没有空闲的寄存器分配
为了给d分配寄存器，不得不进行溢出？

- 假设可供分配的寄存器有3个：r1、r2和r3
- t1、t2、t3都是临时变量，在存储中无副本
- a、b、c是程序变量，在存储器中有副本，且在基本块出口是活跃的

2.1 基于使用计数的局部寄存器分配

- 如何溢出?
  - 根据启发式溢出一个寄存器。
    - 溢出使用计数最小的，因为使用次数较小意味着可能使用不频繁
    - 溢出已经有副本在存储器中的。这样，可以少一条store指令，从而减少spill代码。

$t1^{r2} \leftarrow a^{r1} - b^{r2}$

$t2^{r1} \leftarrow a^{r1} * c^{r3}$

$t3^{r2} \leftarrow t1^{r2} + t2^{r1}$

$a \leftarrow d^{?} + c^{r3}$

$t1^{r2} \leftarrow a^{r1} - b^{r2}$

~~$t2^{r1}$~~ $\leftarrow a^{r1} * c^{r3}$

$t3^{r2} \leftarrow t1^{r2} + t2^{r1}$

$a^{r1} \leftarrow d^{r1} + c^{r3}$

$d^{r2} \leftarrow t2^{r3} + t3^{r2}$

```
ld    r1  a
ld    r2  b
sub   r2  r1,r2
ld    r3  c
mult  r1  r1,r3
add   r2  r2,r1
st    t2  r1
ld    r1  d
add   r1  r1,r3
ld    r3  t2
add   r2  r3,r2
st    a   r1
st    d   r2
```

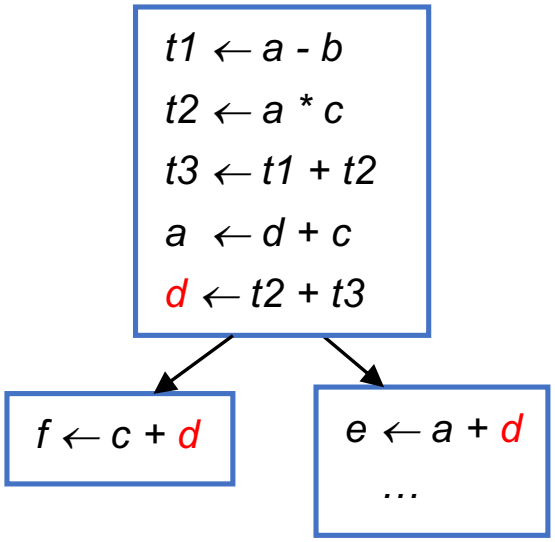
spill code

只能从t2和t3中选择一个寄存器溢出，t2、t3的剩余计数相同，都为1，且都无存储器副本

我们任意选择溢出t2占用的寄存器r1，这样t2的值需要store到存储器中

2.1 基于使用计数的局部寄存器分配

- 基于使用计数的局部寄存器分配
- 优点
  - 简单，易于实现
- 缺点
  - 使用计数只考虑单个基本块
  - 不能真正反映变量的执行时使用情况，e.g 循环内使用的变量
  - 局限在基本块内，导致生成了一些多余的spill代码



2.2 基于使用计数的全局寄存器分配

- 跨基本块边界，在一个程序段的范围统一考虑寄存器分配问题
- 全局性地评估各种变量分配寄存器所得到的好处
  - 不只考虑引用次数一个因素，也考虑变量在基本块的入口和出口的活跃情况
  - 针对大部分程序90%的执行时间花在循环上的情况，优先分配循环中的变量，给循环中的变量加以适当的权重。

- 这种方法几乎是上个世纪80年代之前所有编译器通用的寄存器分配方法
  - 银河1巨型机编译器

2.2 基于使用计数的全局寄存器分配

- 全局性地评估各种变量分配寄存器所得到的好处

$$benefit(v, \mathrm{B}) = use(v, \mathrm{B}) + 2 \times liveout(v, \mathrm{B})$$

基本块B中为变量v分配寄存器获得的收益

- $use(v,B)$:在基本块B中，定值v之前对v的引用次数
- 如果v在基本块B出口是活跃的，并且在B中被定值，则$liveout(v, \mathrm{B})$为1，否则为0

- 对于循环L中的变量v，分配寄存器而得到的收益是循环内的所有基本块的好处之和

$$benefit(v, \mathrm{L}) = \sum_{B \in L} \big(use(v, \mathrm{B}) + 2 \times liveout(v, \mathrm{B})\big)$$

2.2 基于使用计数的全局寄存器分配

- 全局性地评估各种变量分配寄存器所得到的好处

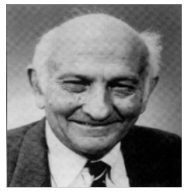
$$benefit(v, \mathrm{B}) = use(v, \mathrm{B}) + 2 \times liveout(v, \mathrm{B})$$

$$benefit(v, \mathrm{L}) = \sum_{B \in L} \left(use(v, \mathrm{B}) + 2 \times liveout(v, \mathrm{B}) \right)$$

1. 计算了每个变量的收益之后，全局寄存器分配使用R个寄存器进行全局分配，留若干个寄存器用于其余变量
2. 分配从最内层循环开始，由内向外给前R个获益值最高的变量分配寄存器
3. 在循环之外的代码，则根据使用次数和活跃情况来计算收益，然后根据收益进行分配寄存器和溢出寄存器

3. 基于图着色的寄存器分配

- 1971年IBM的John Cocke提出



xJohn Cocke
1987

JOHN COCKE DL

United States – 1987

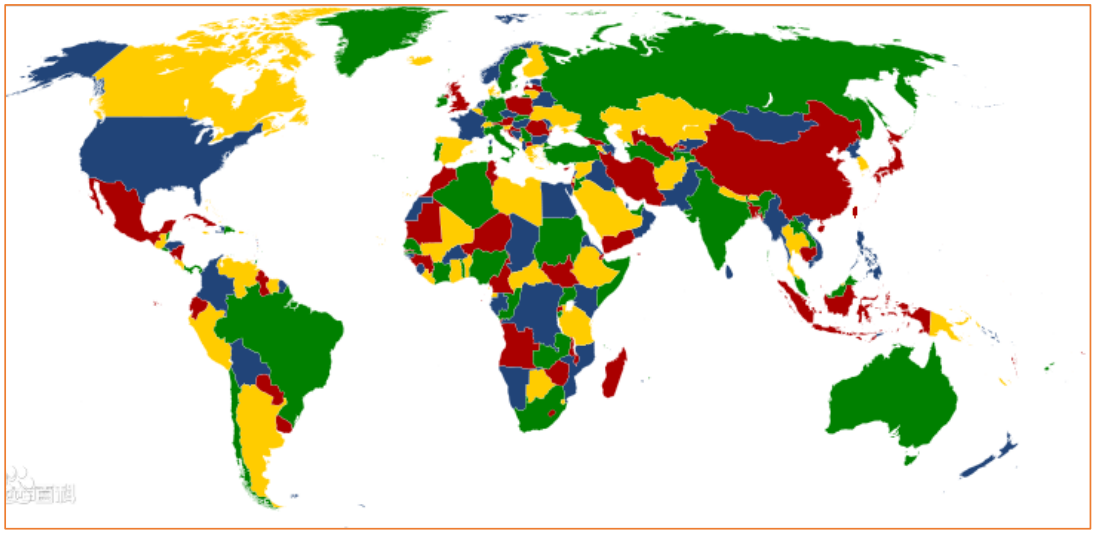
CITATION

For significant contributions in the design and theory of compilers, the architecture of large systems and the development of reduced instruction set computers (RISC); for discovering and systematizing many fundamental transformations now used in optimizing compilers including reduction of operator strength, elimination of common subexpressions, register allocation, constant propagation, and dead code elimination.

- 1981年IBM的Chaitin首次实现
- 随后开始流行，衍生出基于Chaitin算法的多种改进算法
  - 1994年Briggs的改进算法
  - 1996年George的改进算法

3.1 图着色问题

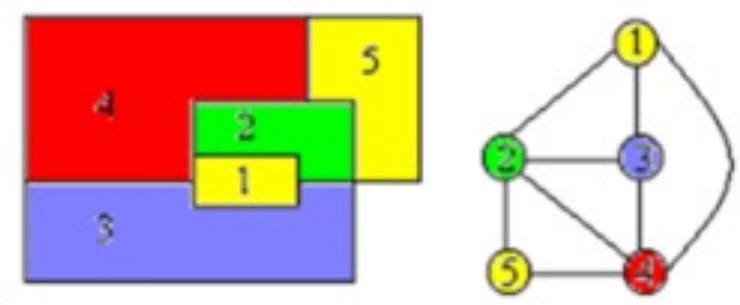
- 问题：什么是四色定理?



- 用不同的颜色（4种）给地图着色，相邻国家不能着同一颜色

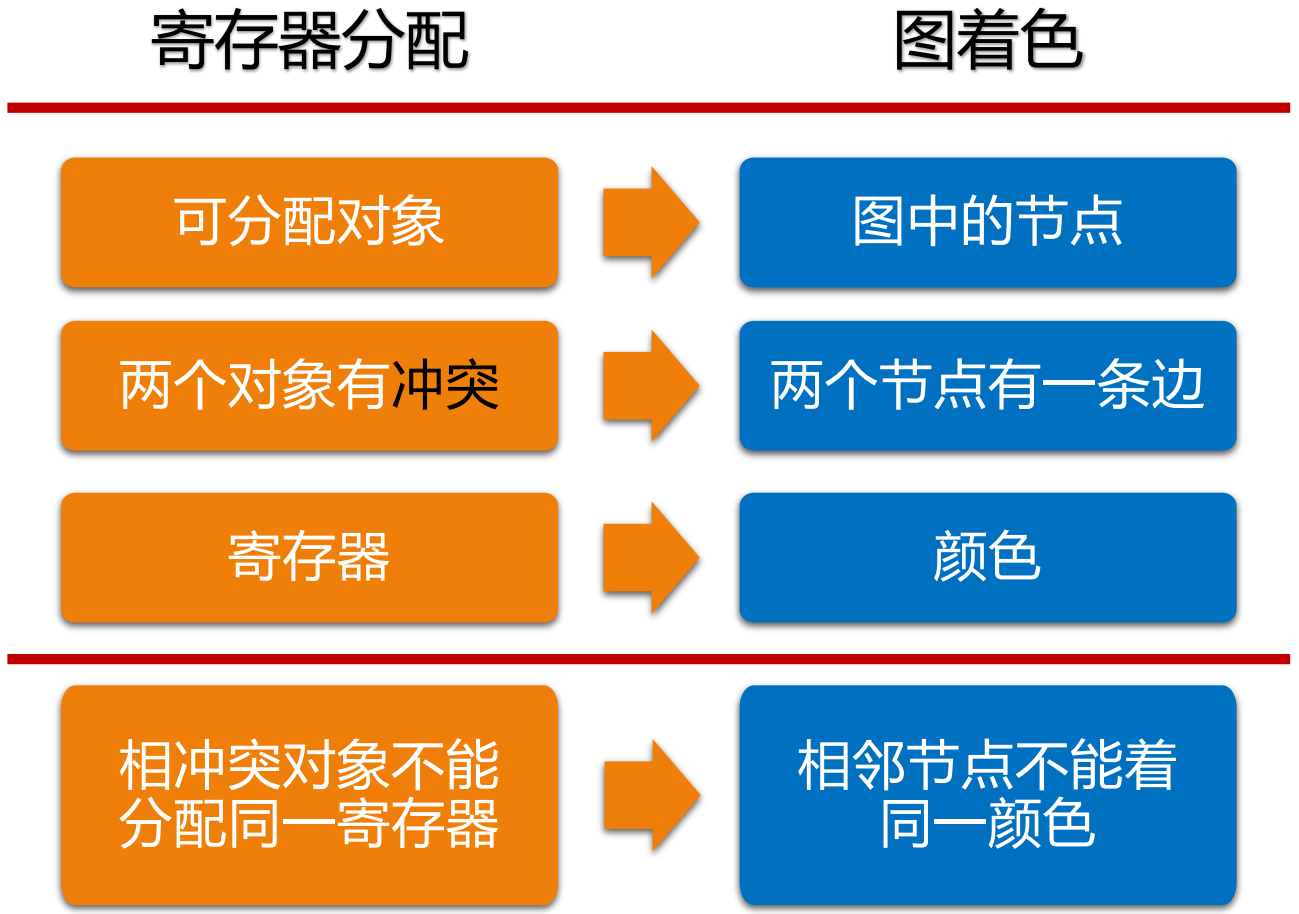
3.1 图着色问题

- 用不同的颜色给无向图中的节点着色
- 相邻节点（有一条边相连）不能着同一颜色
  - 不相邻的节点可以着相同的颜色



- 一个图是**k-可着色(k-colorable)** = 可以用**k**种颜色着色

3.2 寄存器分配与图着色的关联



3.3 可分配对象

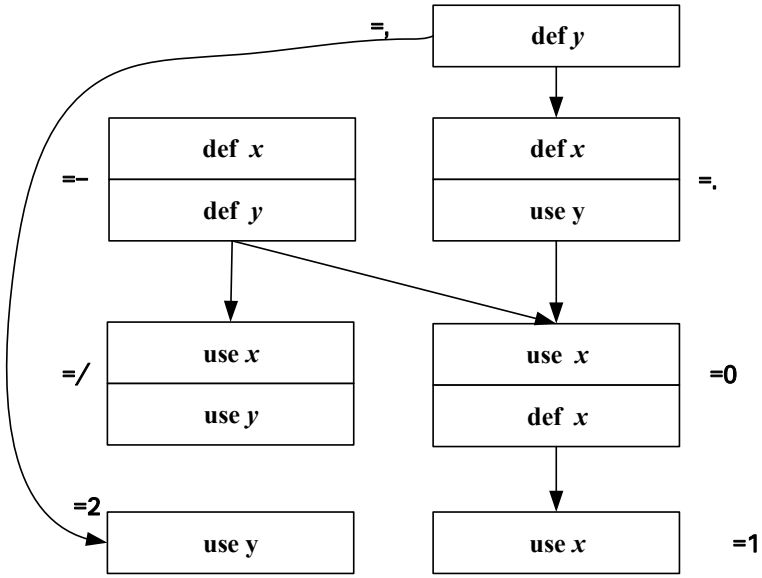
- 可分配对象
  - 变量，临时变量
    - 程序中有许多变量，同一个变量可以用于不同的目的被重复地使用

```
for(i=0;i<N;i++)
{…}

for(i=0;i<M;i++)
{…}
```

    - 将这种变量分隔开来，减少与其它变量的冲突
  - 网（web）

3.3 可分配对象

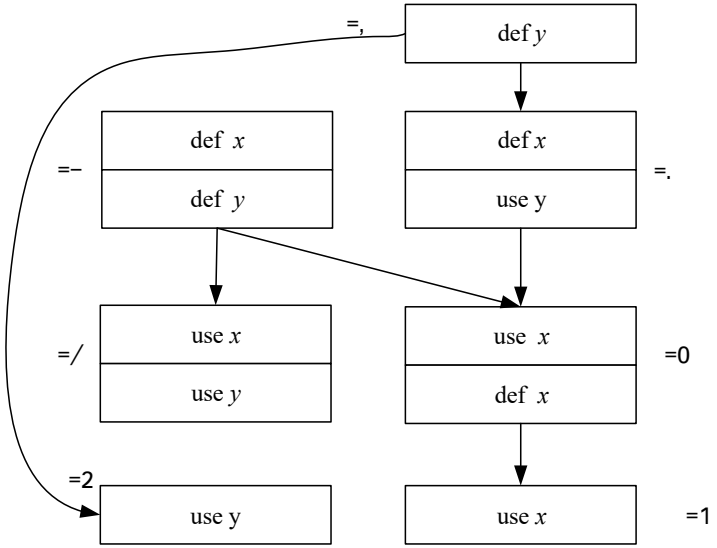
- 网（web）
  - 如果两个DU链有公共的使用，则这两个DU链相交
  - 一个变量的网是该变量的各个定值-使用（DU）链中相交的那些DU链组成的最大并集

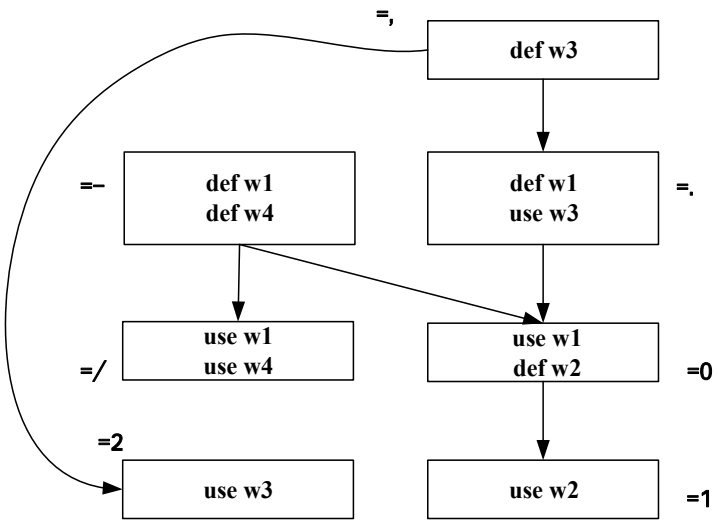


网	成员
w1	在B2、B3中的x定值、在B4、B5中x的使用
w2	在B5中的x定值、在B6中x的使用
w3	在B1中y的定值，在B3、B7中y的使用
w4	在B2中y的定值，在B4中y的使用

3.3 可分配对象

- 为每个网分配一个符号寄存器





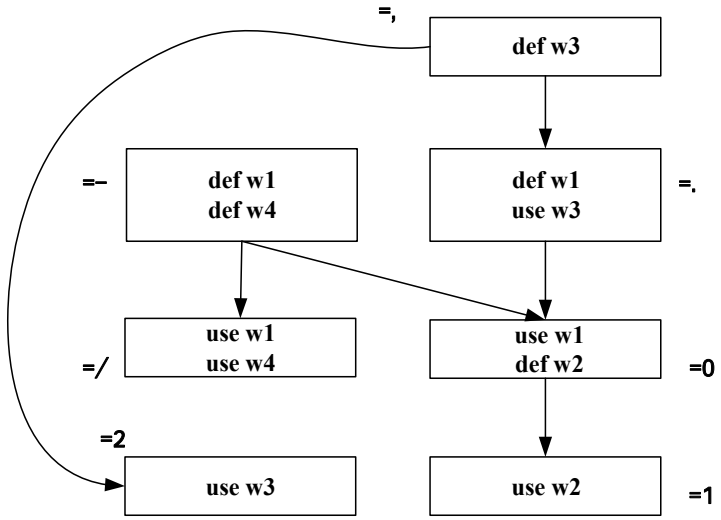
x: $(def_{B2} \mid use_{B4}, use_{B5})$
$(def_{B3} \mid use_{B5})$
$(def_{B5} \mid use_{B6})$

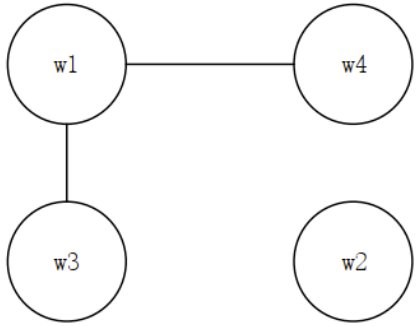
y: $(def_{B1} \mid use_{B3}, use_{B7})$
$(def_{B2} \mid use_{B4})$

3.4 冲突图

➢一个冲突图（Interference Graph，IG）是一个无向图*IG(V,E)*

- V = { v | v是程序中的可分配对象 (allocatable object)}
- E = { e = < v1 ,v2 >| v1∈V ,v2∈V , v1和v2之间有一条边，则表示v1和v2在程序的某个点 (program point) 同时活跃}





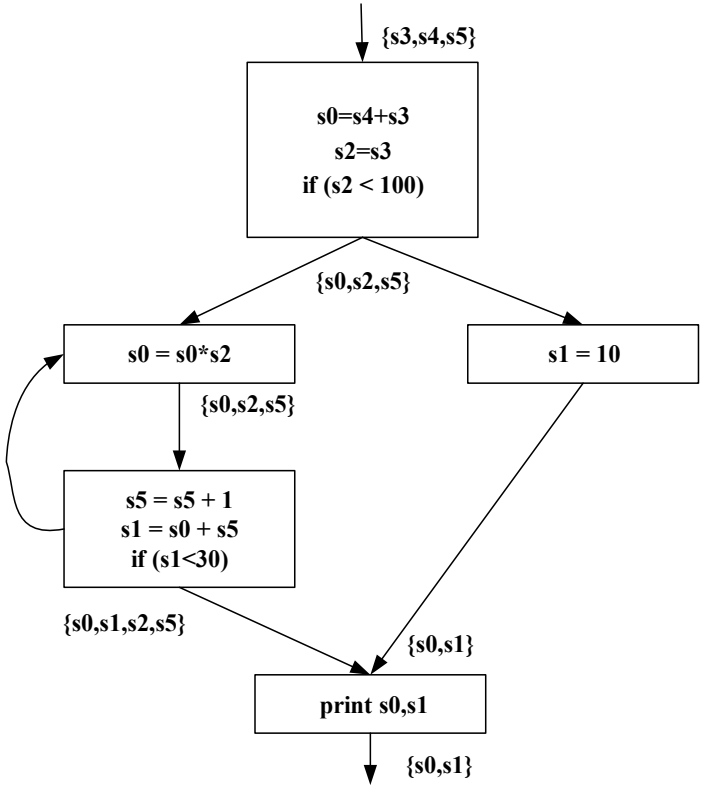
IG

3.4.1 冲突图的构建

- 方法：基于活跃变量分析构建冲突图
  - 活跃变量分析：如果一个变量v在点p开始的某条路径上被使用，那么变量v在点p是活跃 (live)的；否则，变量v在点p是死变量
- 通过活跃变量分析，获得程序中每一个点的活跃变量集合

3.4.1 冲突图的构建

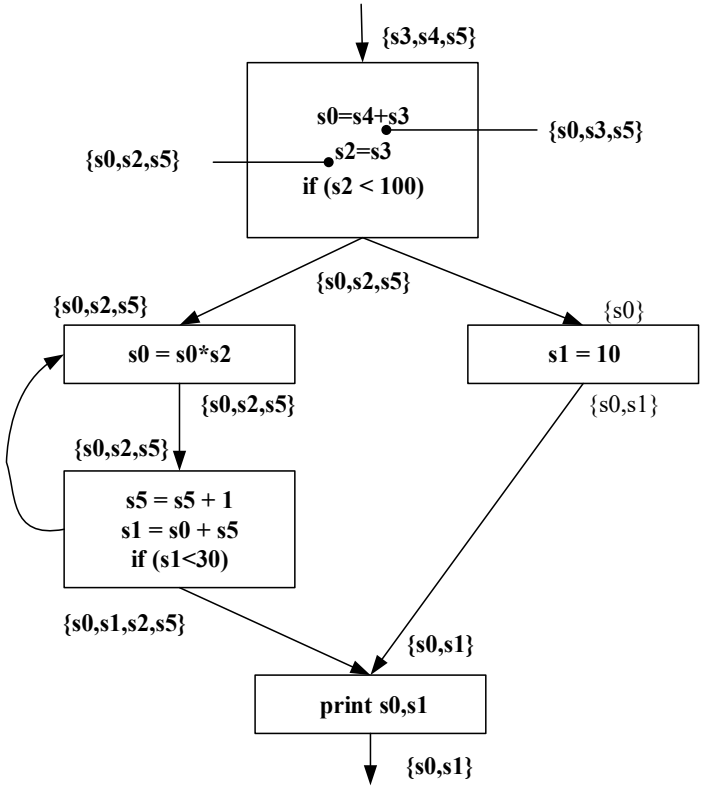
- 例子：考虑如下控制流图，有s0~s5共6个符号寄存器



确定可分配对象

3.4.1 冲突图的构建

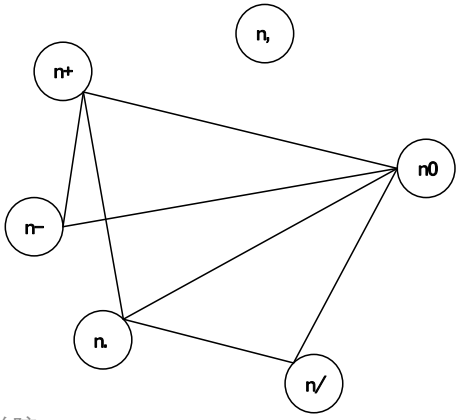
- 例子：考虑如下控制流图，有s0~s5共6个符号寄存器



确定可分配对象

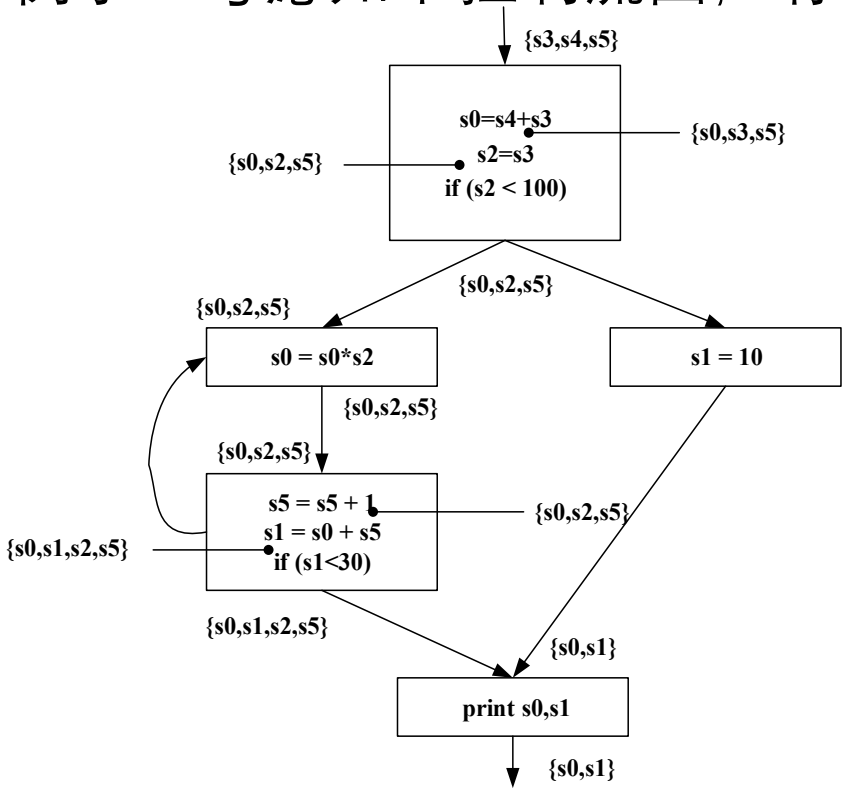
↓

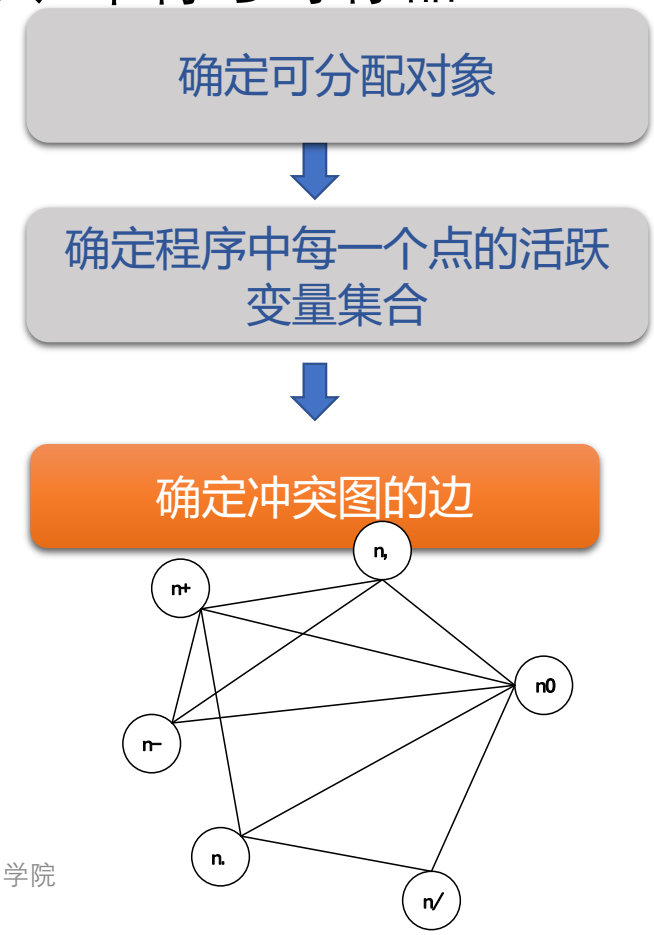
确定程序中每一个点的活跃变量集合



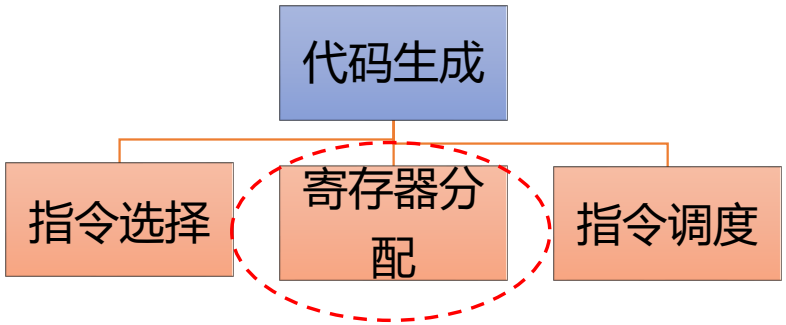
3.4.1 冲突图的构建

- 例子：考虑如下控制流图，有s0~s5共6个符号寄存器





目标代码生成涉及的问题

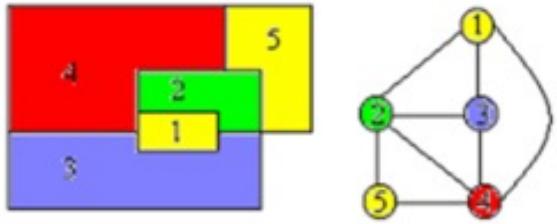


- 指令选择
  - 将中间代码翻译成等价的目标机指令集指令序列，实现中间代码到目标机指令的映射
- 寄存器分配
  - 对于load/store架构的目标机是必须要考虑的问题
- 指令调度
  - 通过重排指令，减少流水线中的停顿，最大化指令级并行

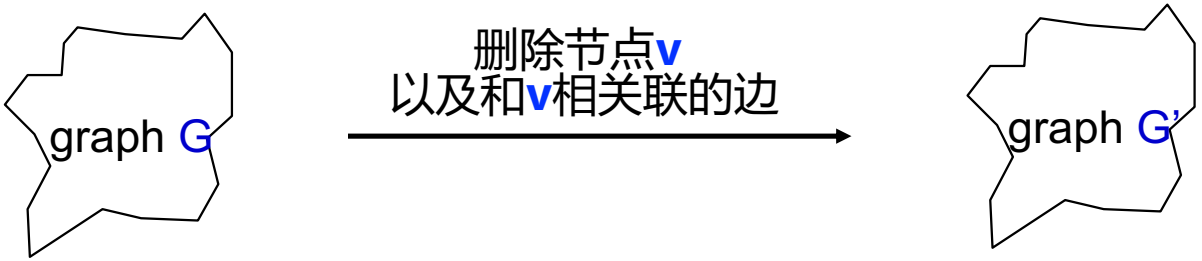
4. 基于图着色的寄存器分配

- 为程序中的可分配对象建立**冲突图**
- 将颜色看作寄存器，尝试为这个冲突图找到一种用k种颜色着色的方案，其中k是寄存器的数目

4.1 度< k定理



- Kempe做四色定理证明时提出
- 度<k定理
  - 定义Degree(v) = 和v相关联的边的数目
  - 设节点v是G中的一个节点，且满足degree(v) < k



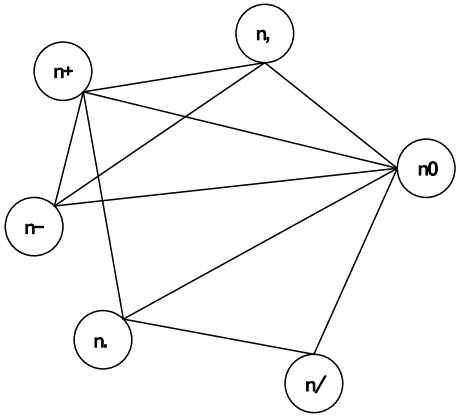
  - **如果G是k-可着色的，当且仅当G ′是k-可着色的**

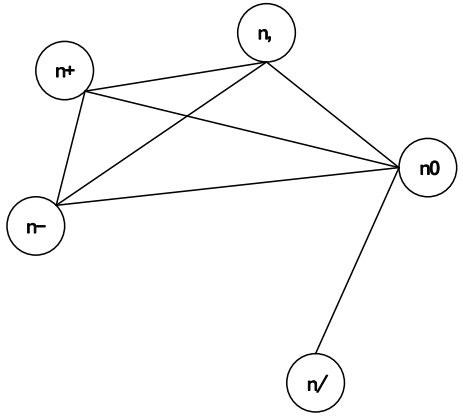
4.1 通过化简着色

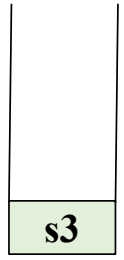
- 第一步：化简冲突图 (simplify)
  - 1. 从冲突图G中选出一个节点**v**满足degree(**v**) < k
  - 2. 把节点**v**以及和**v**相关联的边从冲突图G中删除，并将**v**压栈
  - 3. 重复1和2直到冲突图中只剩下一个节点

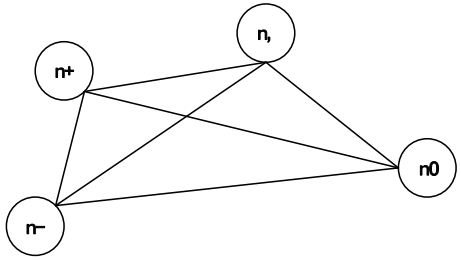
4.2 通过化简着色

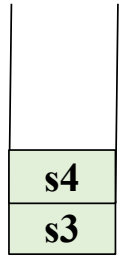
- 例子：考虑如下冲突图的着色，k=4
  - degree(s3)<4，删除s3
  - degree(s4)<4，删除s4





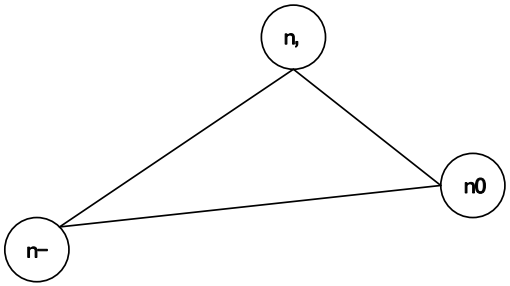


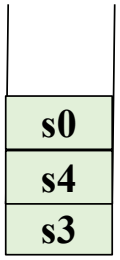


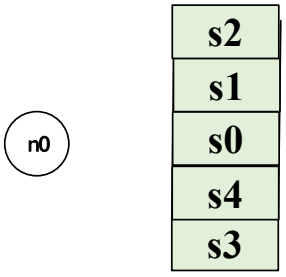


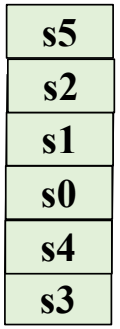
4.2 通过化简着色

- 例子：考虑如下冲突图的着色，k=4
  - degree(s0)<4，删除s0
  - 同理，删除S1、S2，剩余s5
  - 最后删除s5







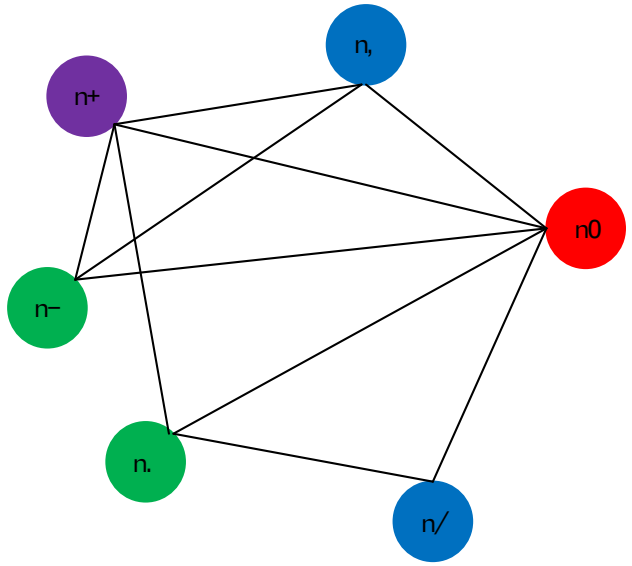


4.2 通过化简着色

- 第二步：选择颜色 (select)
  - 1.从最后一个进栈的节点开始
  - 2. 每次弹出一个节点，将其重新放回到冲突图中，并为其选择一个和已着色邻居节点不同的颜色

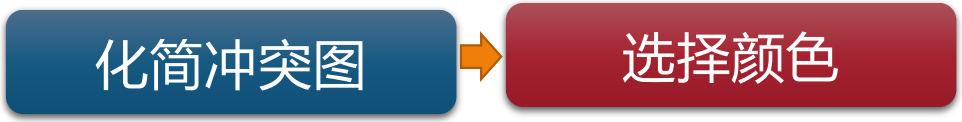
4.2 通过化简着色

- 例子：考虑如下冲突图的着色， k=4
  - 将节点依次弹栈，放回到冲突图中，给节点选择一个颜色（与邻居节点不同的颜色）



4.2 通过化简着色

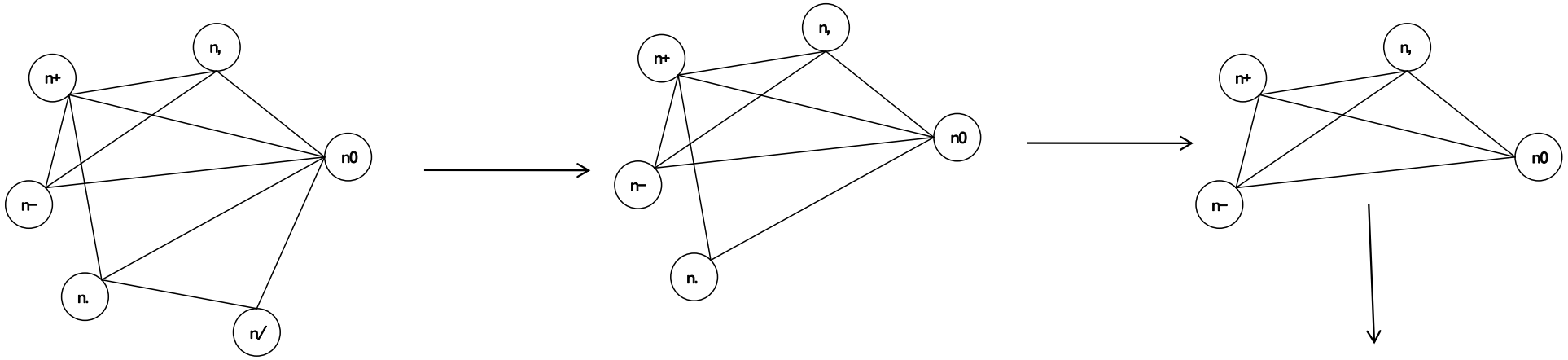
- 对于k-可着色图，通过化简着色包括2个步骤



- 如果一个冲突图是k-可着色的，则存在一个使用不超过k种颜色的寄存器分配方案
- 对于寄存器分配问题，不能保证一个冲突图一定是k-可着色的
- 如果冲突图不是k-可着色的，怎么办？

4.2 通过化简着色

- 假设k=3，考虑冲突图的着色
  - 所有节点的度数都大于等于3



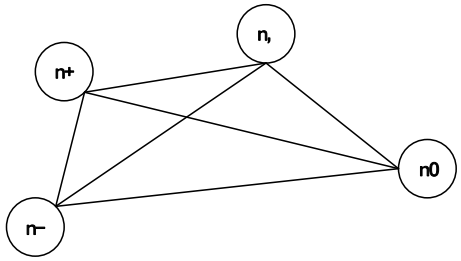
无法化简

？

如何减少冲突?

4.3 溢出

- 例子：考虑如下冲突图的着色，k=3
  - 在G中删掉一个节点成为G′， G′可能成为可着色的



  - 通过溢出一个节点，减少冲突边，使得寄存器分配可以继续

4.3 溢出

- 溢出的选择
  - 简单的启发信息：度最高的节点
  - 溢出收益评估

$$benefit(v,\mathrm{B}) = use(v,\mathrm{B}) + 2\times liveout(v,\mathrm{B})$$

$$benefit(v,\mathrm{L}) = \sum_{B\in L}\left(use(v,\mathrm{B}) + 2\times liveout(v,\mathrm{B})\right)$$

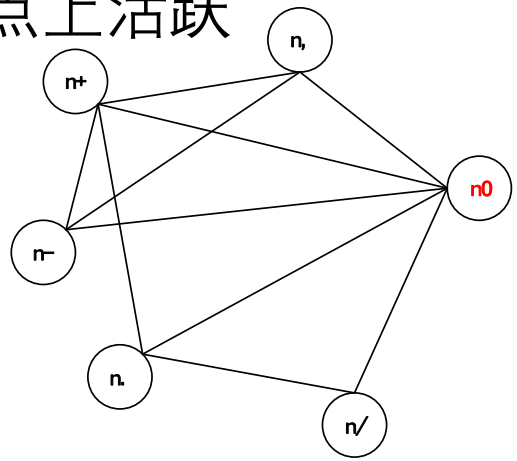
  - 溢出代价评估

$$\cos t(w) = defwt\cdot\sum_{def\in w}10^{depth(def)} + usewt\cdot\sum_{use\in w}10^{depth(use)} - copywt\cdot\sum_{copy\in w}10^{depth(copy)}$$

    - def、use和copy分别是网w的各个定值、使用和寄存器复制
    - defwt、usewt和copywt是给指令类型指定的权重
    - depth表示循环的嵌套深度

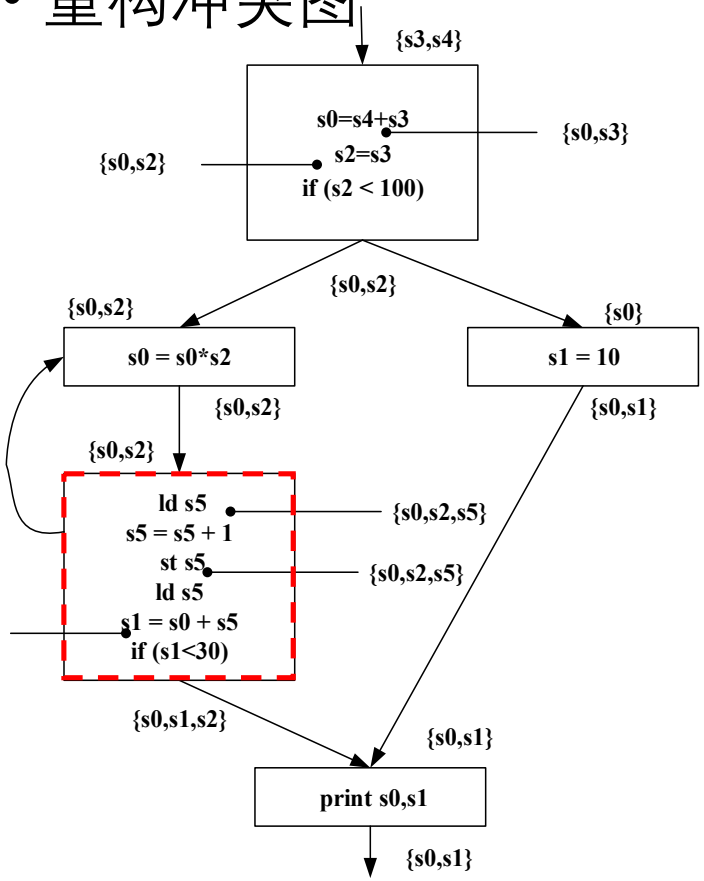
4.3 溢出

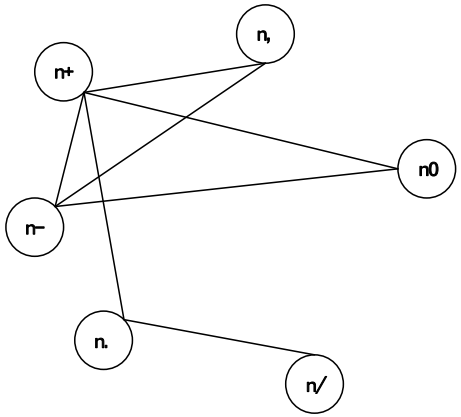
- 选择溢出度数最高的s5溢出
- s5的活跃区间发生变化，s5仅在下列程序点上活跃
  - 在load **s5** 和下一条指令(s5的使用)之间
  - 在store s5和前一条指令(s5的定值)之间
- 溢出后需重构冲突图



4.3 溢出

- 重构冲突图





- 溢出的作用是减少图中的冲突边，以减少寄存器的压力
- 溢出代价很大，是寄存器分配需要极力避免的

可以的化简顺序：S4、S3、S5、S1、S0、S2

是否还有其它方法可以减少寄存器的压力，从而避免溢出呢?

4.4 合并

- 考虑如下复制指令，x赋值给y后不再使用x
  - x和y不同时活跃 (没有冲突边)
  - x和y可以使用同一个寄存器
  - 从而可以删除复制指令

$x \leftarrow \ldots$
$\ldots$
$y \leftarrow x$（之后不再使用x）
$\ldots$
$\ldots \leftarrow y$

- 合并(coalescing):寻找可以删除的复制指令并合并其节点的过程
  - 如果x和y不冲突，且由一条复制指令相关联，则可将x和y合并成一个节点，使x和y使用同一个寄存器(减少了冲突图的节点数)

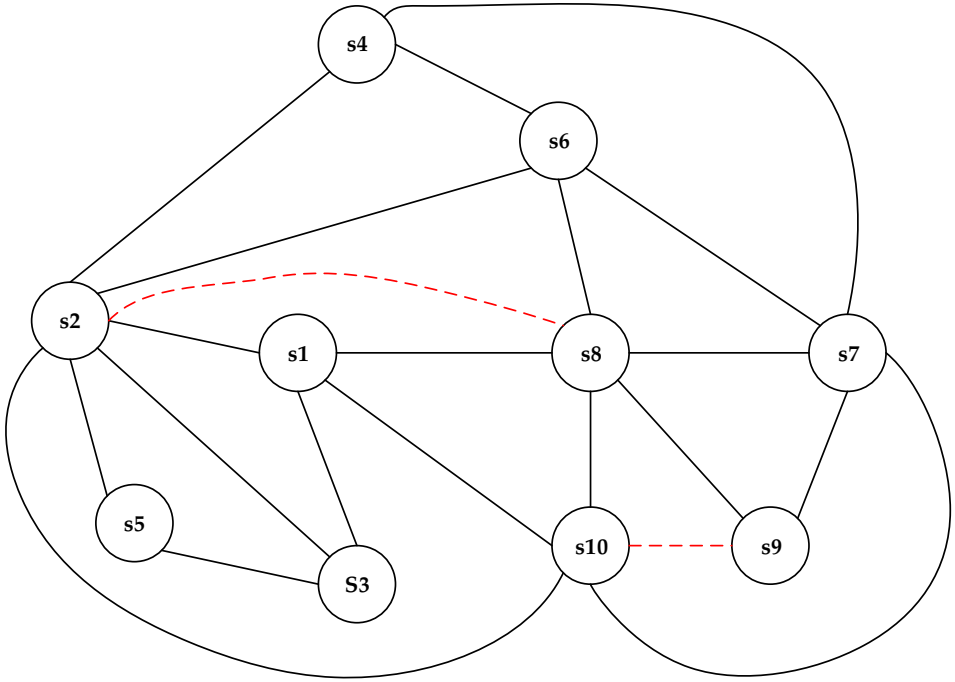
4.4 合并

- 合并
  - 通过减少节点数减少寄存器分配的压力
  - 活跃区间不会发生变化，无需重构冲突图
  - 合并后的新节点的边是原来两个节点的边的并集
- 溢出
  - 通过减少冲突边减少寄存器分配的压力
  - 活跃区间发生变化，重构冲突图

4.4 合并

- 例子：考虑如下代码，合并复制相关的s9-s10

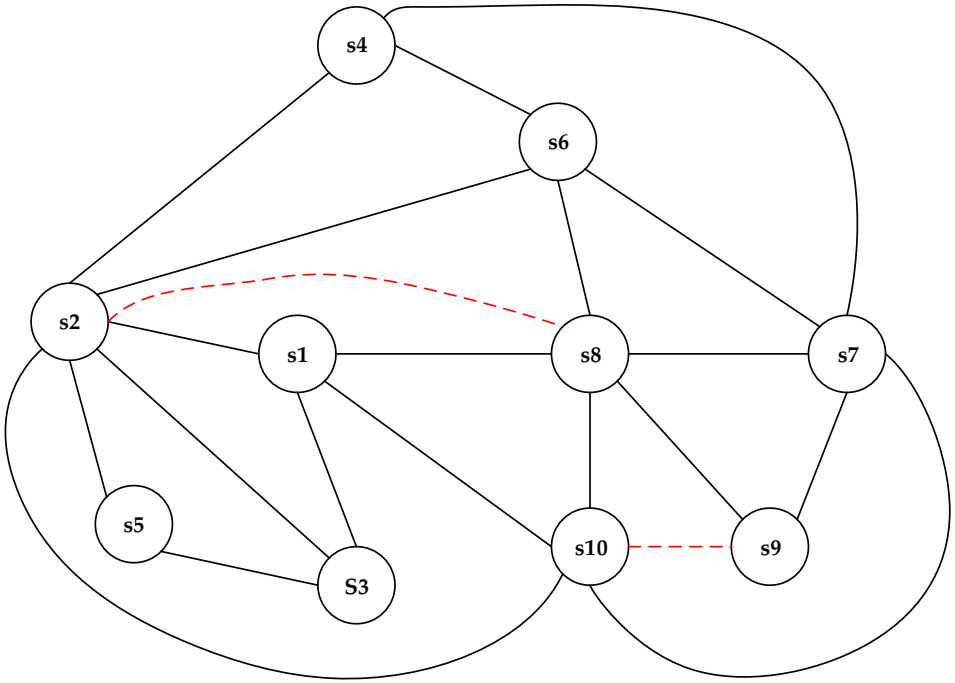
```
入口活跃: s1  s2
        s3 =mem[s2+12]
        s4 =s1 - 1
        s4 =s3 + s5
        s6 =mem[s2+8]
        s7 =mem[s2+16]
        s8 =mem[s4]
        s9 =s6 + 8
        s10 =s9
        s1 =s7 + 4
        s2 =s8
出口活跃: s10  s1  s2
```

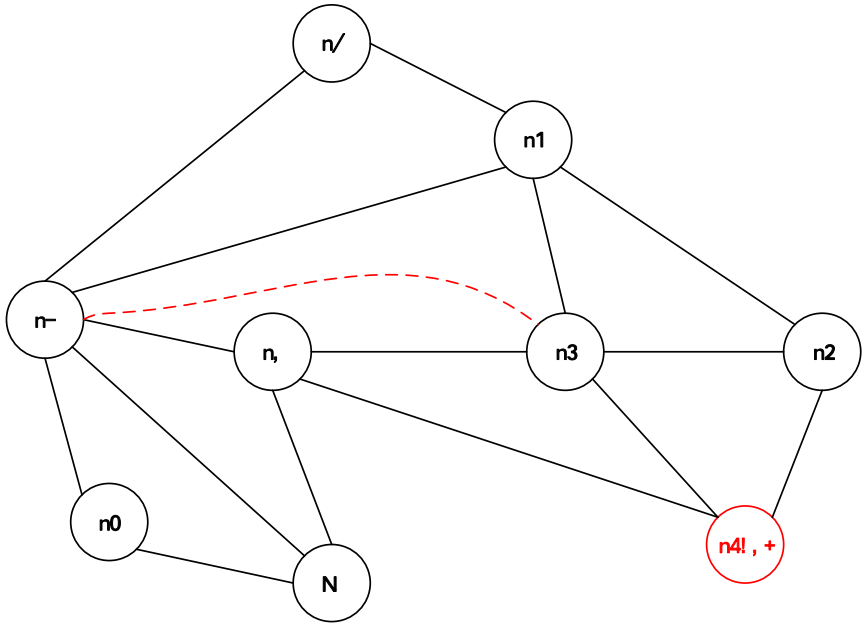


虚线表示传送指令

4.4 合并

- 例子：考虑如下代码，合并复制相关的s9-s10





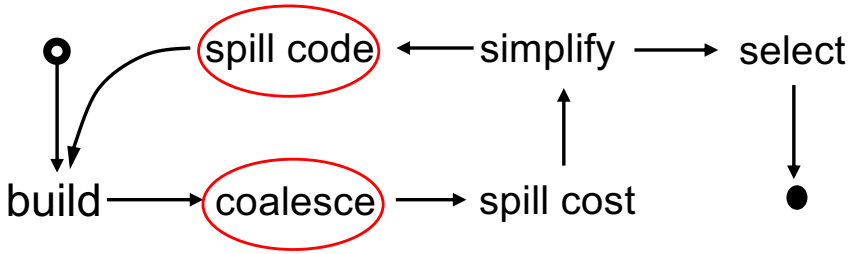
合并s9-s10

4.5 Chaitin算法

- 1. 确定可分配对象
- 2. 构建冲突图 (build)
- 3. 合并复制相关且无冲突边的节点 (coalesce)
- 4. 评估每个节点的溢出代价 (spill cost)
- 5. 化简冲突图，将度小于k的低度数节点删除并入栈 (simplify)
- 6. 如果不再有低度数节点，则根据4选择一个节点溢出，插入溢出代码 (spill code)，重复2-5直至所有节点入栈
- 7. 将节点依次弹栈，为其选择一个颜色 (select)

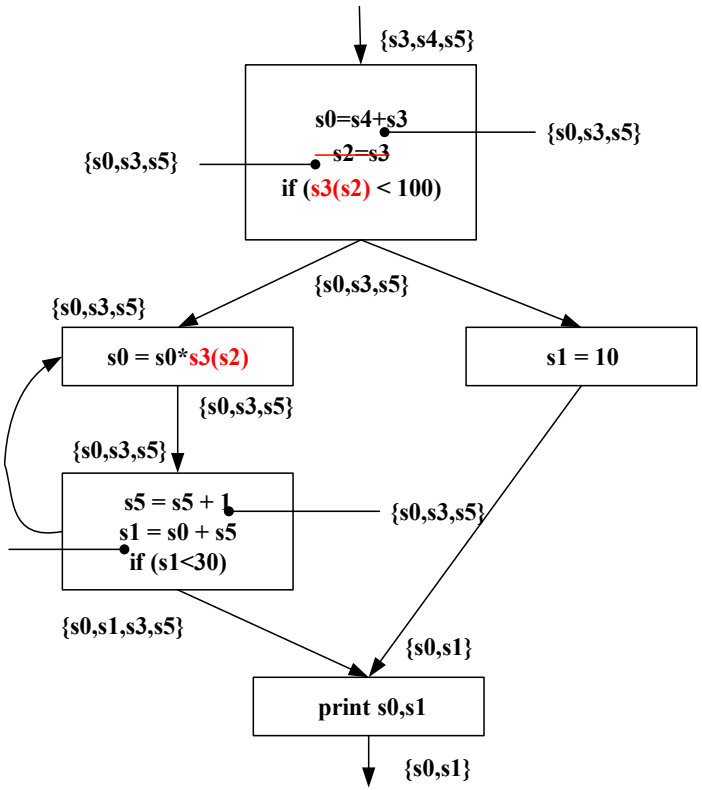
4.5 Chaitin算法

- 构建冲突图、合并节点、化简冲突图、溢出节点的迭代过程

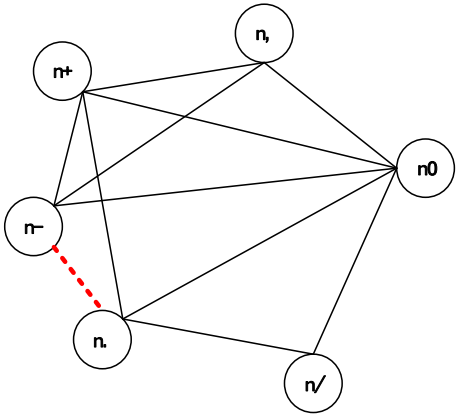


- 奠定了图着色寄存器分配算法的基础
- 后续改进算法主要在合并和溢出部分

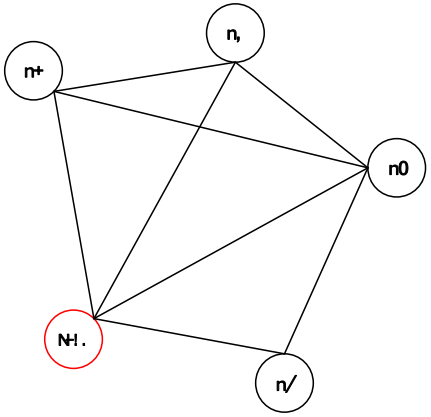
4.5 Chaitin算法



1. 构建冲突图

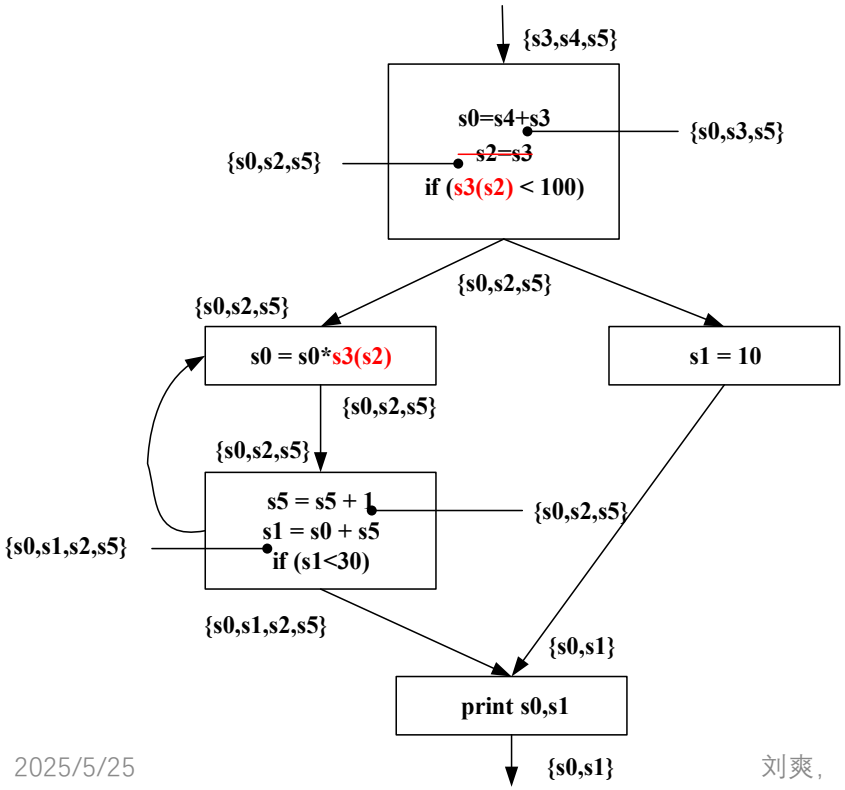


2. 合并s2和s3



4.5 Chaitin算法

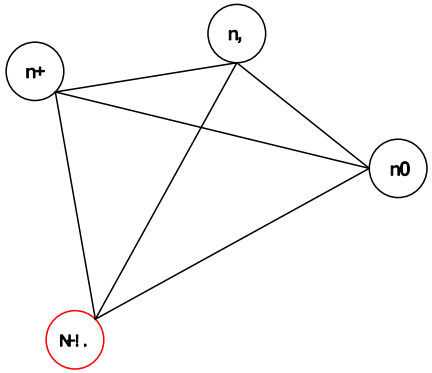
- K=3（寄存器数为3）



3. 计算溢出代价

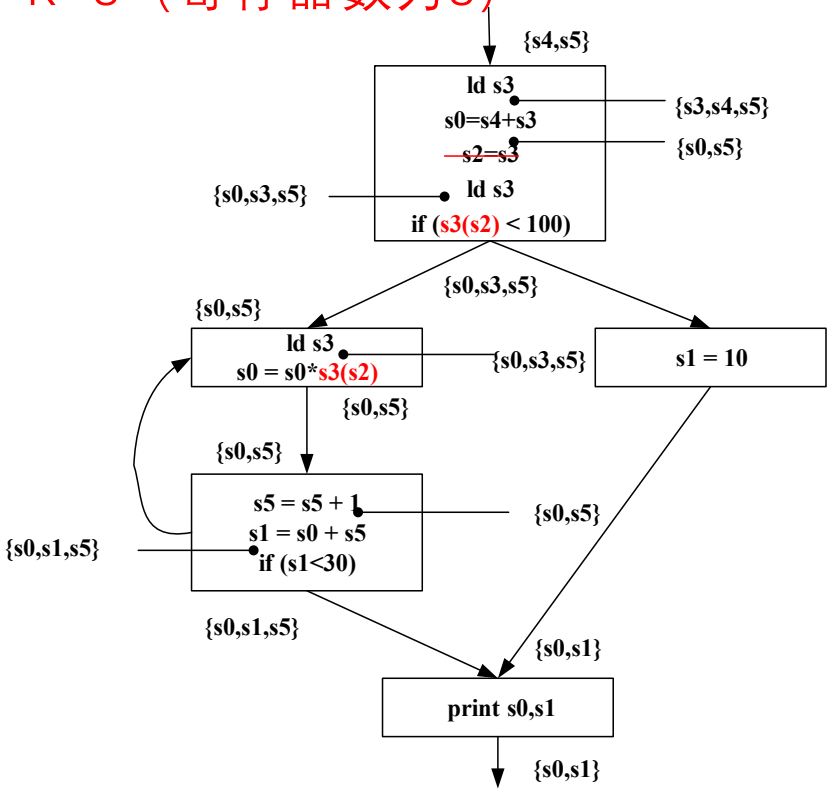
$$\cos t(w) = defwt \cdot \sum_{def \in w} 10^{depth(def)} + usewt \cdot \sum_{use \in w} 10^{depth(use)} - copywt \cdot \sum_{copy \in w} 10^{depth(copy)}$$

4. s4度数为2，化简，直到节点度都大于等于3

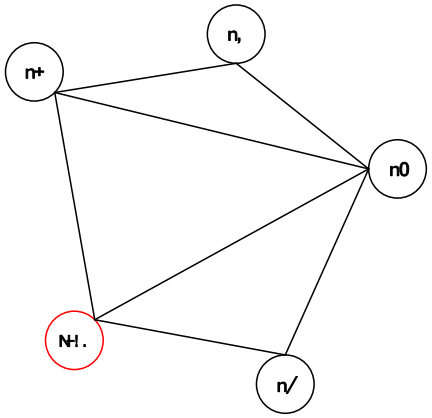


4.5 Chaitin算法

- K=3（寄存器数为3）

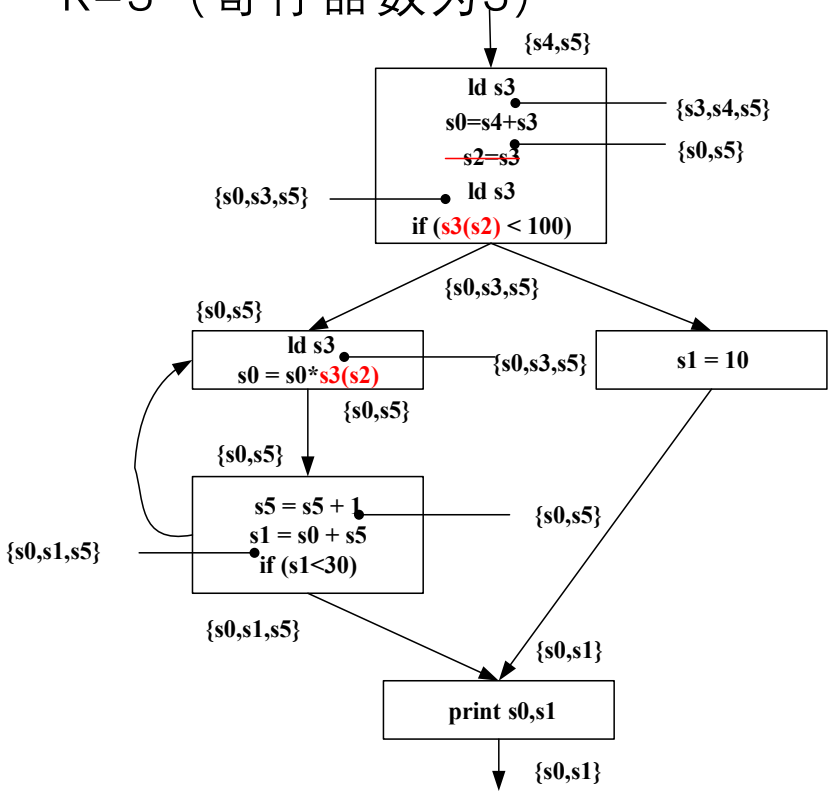


5. 选择溢出代价最小的s2&s3溢出，重构冲突图，计算溢出代价

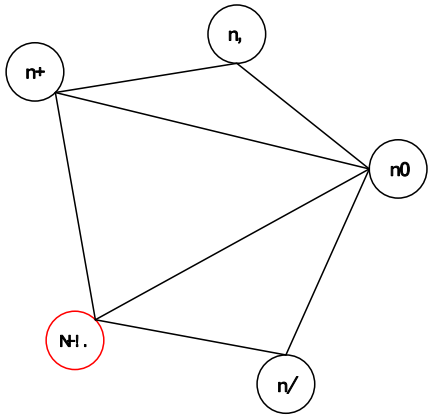


4.5 Chaitin算法

- K=3（寄存器数为3）

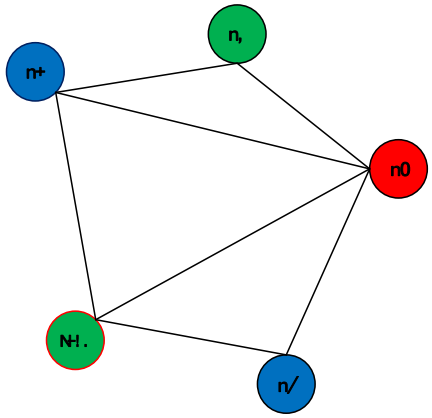


5. 化简



化简s4,s1,s2&3,s0,s5

6. 指派颜色



4.6 Briggs改进算法

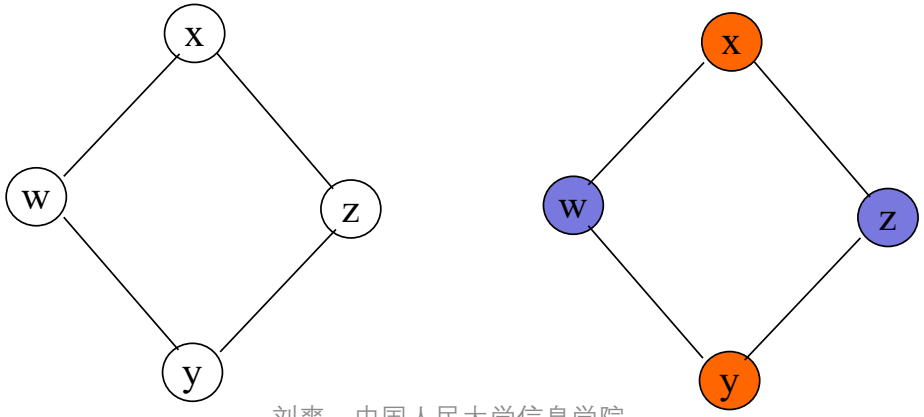
- Chaitin算法可能存在的问题
  - 合并策略太激进
  - 无条件合并无冲突边相连的传送相关的节点，可能导致合并后节点的度数增高，进而导致原来是k-可着色的图变成不能用k种颜色着色，从而不得不进行溢出
- Briggs的改进
  - 保守合并：对合并增加了额外的限制条件
  - 乐观着色：将实际的溢出推迟到选择阶段

4.6 Briggs改进算法

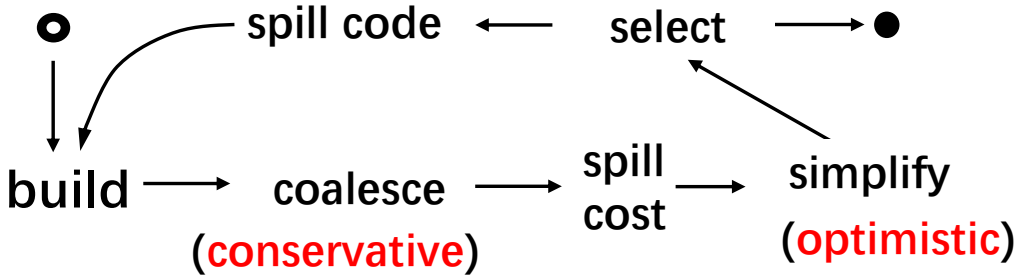
- 保守合并：限定节点a和b可以被合并的条件
  - a和b是传送相关的
  - 合并后的新节点**a&b**的高度数(度≥k)邻居节点数小于**k**
- 保守合并不会改变原图的色数

4.6 Briggs改进算法

- 乐观着色：将溢出推迟到选择阶段
  - 化简到不再有低度数节点时，选择一个溢出代价最小的节点，乐观地将其压栈，并继续化简
  - 选择阶段，如果不能为其指派颜色，才发生实际溢出
- 考虑到一个节点的两个邻居节点有可能使用相同的颜色

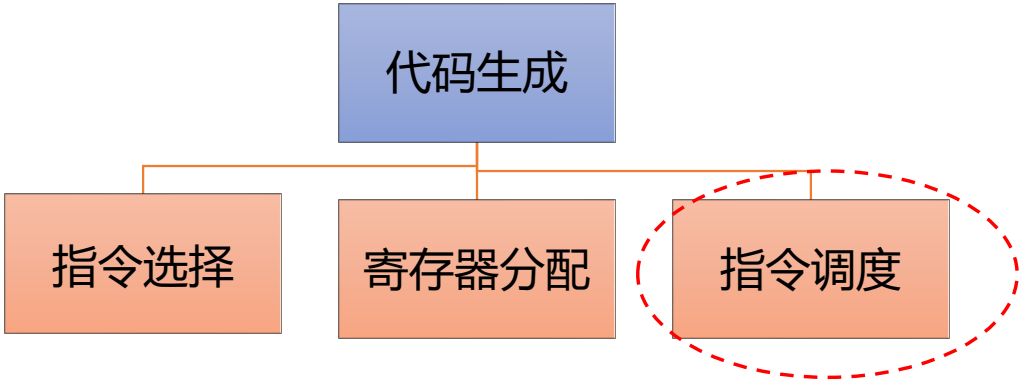


4.6 Briggs改进算法



- 相比Chaitin算法更加高效：有效减少了溢出和重构冲突图的代价

目标代码生成涉及的问题



- 指令选择
  - 将中间代码翻译成等价的目标机指令集指令序列，实现中间代码到目标机指令的映射
- 寄存器分配
  - 对于load/store架构的目标机是必须要考虑的问题
- 指令调度
  - 通过重排指令，减少流水线中的停顿，最大化指令级并行

内容

- 微体系结构简介
- RISC-V架构简介
- 指令选择
- 寄存器分配
- 指令调度

五、指令调度

- 指令序列中各操作的执行顺序影响总执行时间
- 指令调度通过重排指令序列中各操作的执行顺序，试图减少总执行时间

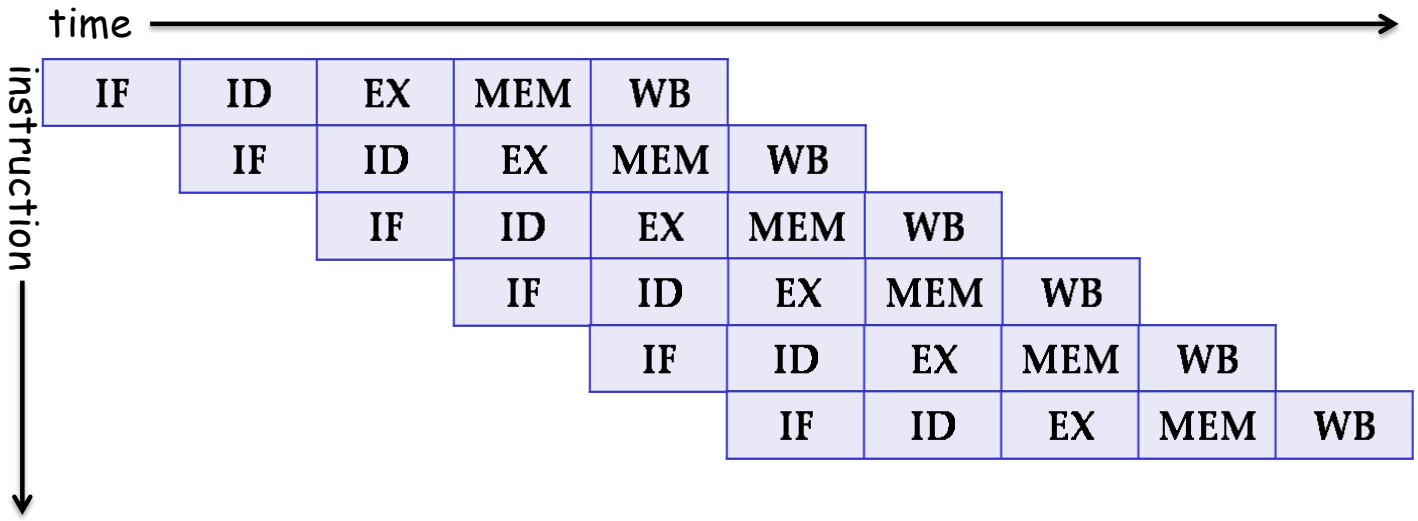
- 指令调度的目标
  - 最大化指令级并行
  - 减少流水线中的气泡(bubbles)
- 指令调度需要考虑的问题非常多，一个最优的指令调度是一个NP完全问题

1. 指令级并行

- 现代处理器有许多技术提高性能，隐藏访存延迟
  - 超标量 (Superscalar)
    - 重复设置多个功能部件
  - 多发射 (Multiple issue)
    - 每个时钟周期流出多条指令
  - 投机执行 (Speculative execution)
    - CPU分支预测器(Branch predictors)
    - 投机取指令 (Speculative loads)
  - 深流水线 (Deep pipelines)
- 上述技术都可以提高指令级并行

1.1流水线

Stage 1	Stage 2	Stage 3	Stage 4	Stage 5
Instruction fetch	Instruction decode	Execute	Memory access	Register write-back
IF	ID/RF	EX	MEM	WB



如何调度一个程序的指令，才能提高指令级并行的能力？

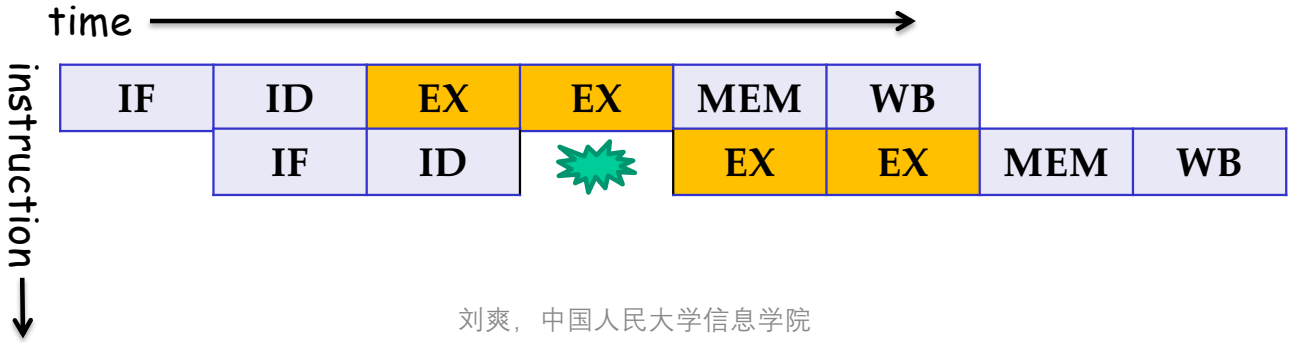
1.1 指令调度的限制

- 结构冲突 (Structural hazards)
- 数据冲突 (Data hazards)
- 控制冲突 (Control hazards)

1.2 指令调度的限制

- 结构冲突 (Structural hazards)
  - 资源约束
  - 没有足够的资源来开发并行性
    - 如寄存器资源、功能部件等

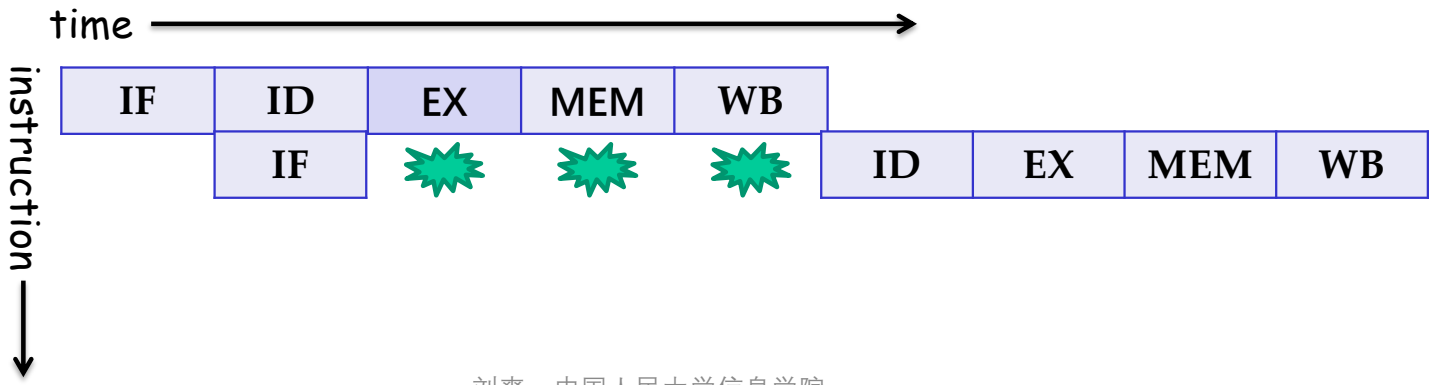
```
mul $r1,$r2,$r3   // Suppose multiplies take two cycles
mul $r4,$r5,$r6
```



1.2 指令调度的限制

- 数据冲突 (Data hazards)
  - 数据依赖关系
  - 指令依赖于前序指令尚未计算出或写回的结果

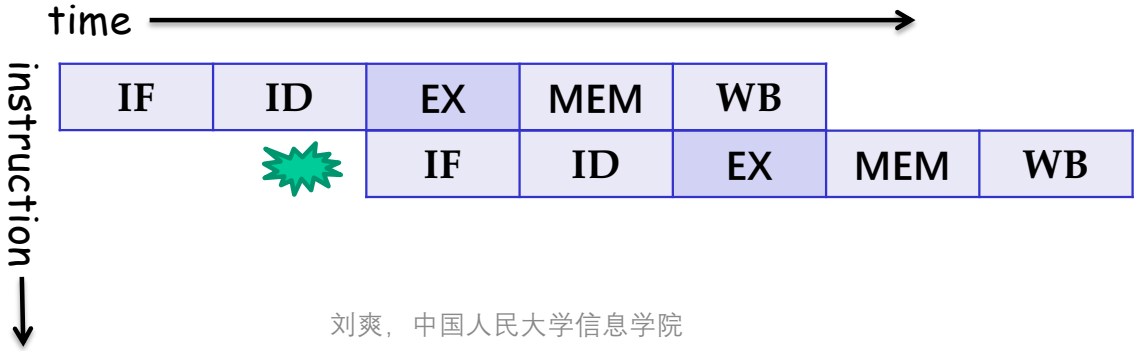
```
mul $r1, $r2, $r3
mul $r4, $r1, $r6
```



1.2 指令调度的限制

- 控制冲突 (Control hazards)
  - 控制依赖关系
  - 分支和跳转指令修改程序计数器
  - 影响到底哪条指令应该发射到流水线上

```
bz $r1, label
add $r2, $r3, $r4
```



1.3 硬件如何解决指令调度的限制

- 结构冲突
  - 复制多个功能部件 (expensive)
  - 更深的流水
- 控制冲突
  - 硬件分支预测 (branch prediction)
  - 运行时投机执行 (runtime speculation)
- 数据冲突
  - 数据定向技术 (data forwarding)
  - 运行时投机执行

2. 依赖

- 在程序执行过程中，如果指令A必须在指令B之前执行，则称B依赖于A
  - 依赖是程序执行顺序上的约束

```
S1:        x1 =a+b
S2:        x2 = x1 *a
S3:        if (x2 > c) goto L1
S4:        x3 =x1/x2
S5:        goto L2
S6:   L1:  x3 =x2
S7:   L2:  x2 = ...
```

- S1给变量x1赋值，S2使用x1的新值，那么S1必须先于S2执行，S2数据依赖于S1
- 执行S4还是S6，取决于S3比较指令的结果。因此，S4、S6控制依赖于S3

2. 依赖

- 根据约束发生的来源

  - 控制依赖：如果约束是由程序的控制流引起

  - 数据依赖：如果约束是由程序的数据流引起，由程序的定值-使用关系导致

```
S1:          x1 =a+b
S2:          x2 = x1 *a
S3:          if (x2 > c) goto L1
S4:          x3 =x1/x2
S5:          goto L2
S6:    L1:   x3 =x2
S7:    L2:   x2 = ...
```

- S1给变量x1赋值，S2使用x1的新值，那么S1必须先于S2执行，S2数据依赖于S1

- 执行S4还是S6，取决于S3比较指令的结果。因此，S4、S6控制依赖于S3

2.1 控制依赖

- 如果一条语句S2执行依赖于S1的输出，那么S2控制依赖于S1，记作$S1\delta^{c}S2$

```
S1:          a ← b + c
S2:          if (a > 10) goto L1
S3:          d ← b * e
S4:          e ← d + 1
S5:  L1:     e ← d / 2
```

$S2\ \delta^{c}\ S3$

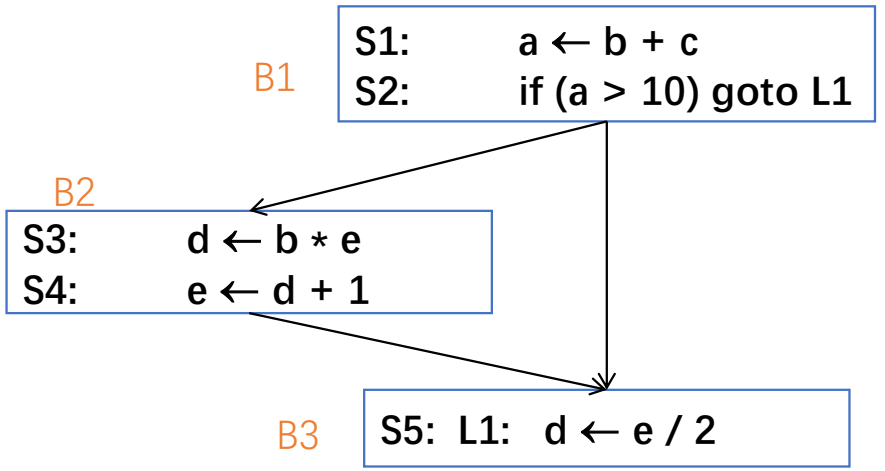
$S2\ \delta^{c}\ S4$

$S2\ \delta^{c}\ S5$??

2.1 控制依赖

- 当基本块B2的执行与否依赖于基本块B1的输出，我们就称基本块B2控制依赖于基本块B1，记作B1δ^cB2

```
S1:        a ← b + c
S2:        if (a > 10) goto L1
S3:        d ← b * e
S4:        e ← d + 1
S5:  L1:   e ← d / 2
```



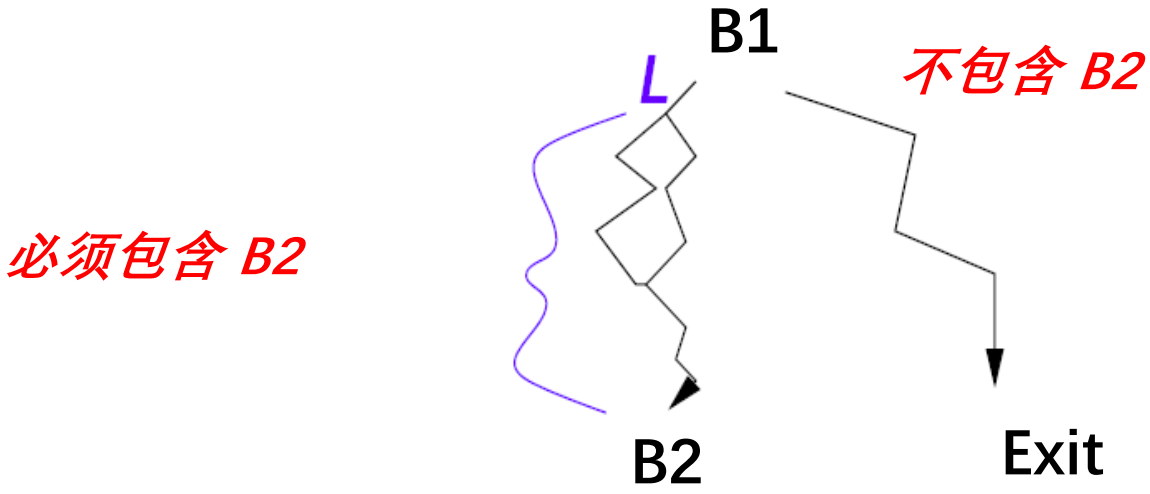
B1 δ^c B2　　B1 δ^c B3 ??

2.1 控制依赖

- 如果基本块B2控制依赖于基本块B1，当且仅当

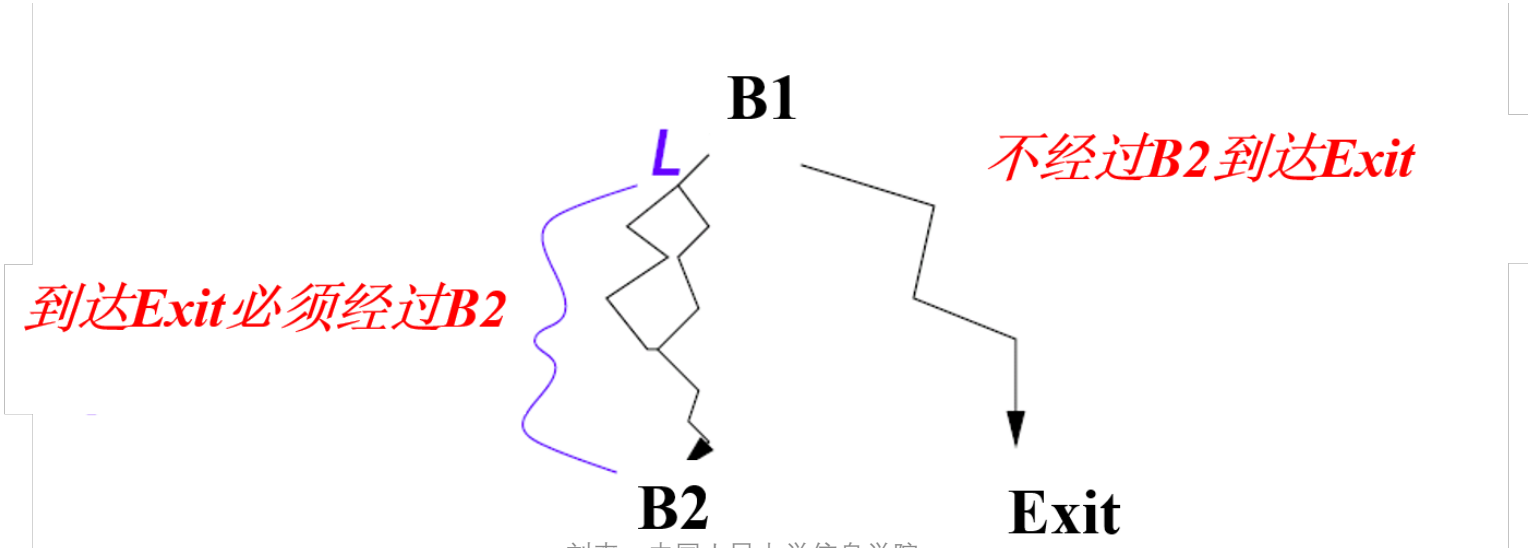
  B2是B1的某些后继的后必经节点，但

  B2不是B1的所有后继的后必经节点



2.1 控制依赖

- 基本块B2控制依赖于基本块B1当且仅当从B1到B2有一条非空路径，并且除了B1外，这一条路径上其他所有节点到达出口节点时都必须经过B2
  - 如果B2控制依赖于B1，那么存在两条路径：一条从B1到出口的路径不包括B2，另外一条则必然经过B2



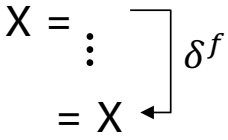
2.2 数据依赖

- 数据依赖
  - 读、写同一存储地址而引起的依赖
- 语句T依赖于语句S是指存在S的一个实例S'和T的一个实例T'以及一个存储单元M，满足
  - S′和T′都访问M（读或写）
  - 当程序串行执行时， S'是在T'之前执行
  - 在同一次执行中，在S'执行结束与T'开始执行前没有对存储单元M的写操作

2.2 数据依赖

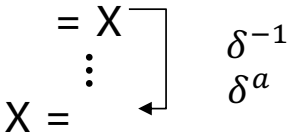
- 数据依赖分类

流依赖



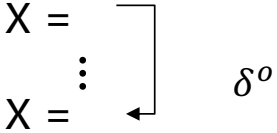
先写后读

反向依赖



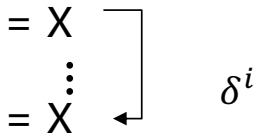
先读后写

输出依赖



先写后写

输入依赖



先读后读

读操作不改变存储状态，并不约束语句的执行顺序，也称为假依赖

2.3 依赖图

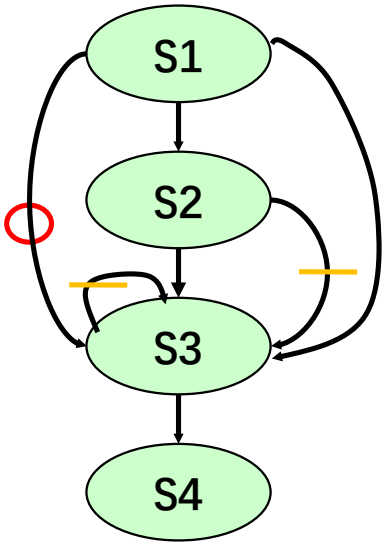
- 使用数据依赖图 G=(N,E)表示数据依赖关系
  - 节点: 指令
  - 有向边(S→T)表示S和T之间存在依赖
    - S必须在T之前执行
  - 在一个基本块内，依赖图是有向无环图

2.3 数据依赖图

- 数据依赖图
  - 节点表示语句
  - 有向边(S→T)表示一个数据依赖
    - S是依赖源，T是依赖槽

S_1: A = 0
S_2: B = A
S_3: A = B + A
S_4: C = A

S_1 δ^f S_2
S_1 δ^f S_3
S_1 δ^0 S_3
S_2 δ^{-1} S_3
S_3 δ^{-1} S_3
S_3 δ^f S_4
S_2 δ^f S_3



（边上“○”表示输出依赖，“—”表示反向依赖，其余是流依赖）

127

3. 指令调度的先决条件

- 资源预约表 (Resource reservation table)
  - 每条指令需要使用的资源
- 机器描述 (Machine description)
  - 对机器可用资源的描述
- 需要调度的程序
  - 需要构建依赖图

指令调度和目标机器紧密相关

3.1 资源预约表

- 在指令调度时，使用资源预约表检查是否存在资源冲突
  - 如果两条指令S1和S2在同一时刻需要相同的资源，则S1和S2冲突

add:

Time	read src1 opnd	read src2 opnd	ALU stage 0	ALU stage 1	MULTIPLIER stage 0	MULTIPLIER stage 1	MULTIPLIER stage 2	MULTIPLIER stage 3	write result bus
0	X	X							
1			X						
2				X					
3									X

mul:

Time	read src1 opnd	read src2 opnd	ALU stage 0	ALU stage 1	MULTIPLIER stage 0	MULTIPLIER stage 1	MULTIPLIER stage 2	MULTIPLIER stage 3	write result bus
0	X	X							
1					X				
2						X			
3							X		
4								X	
5									X

latency(add) = 4, latency(mul) = 6
加法操作不能只落后2拍跟着乘法操作

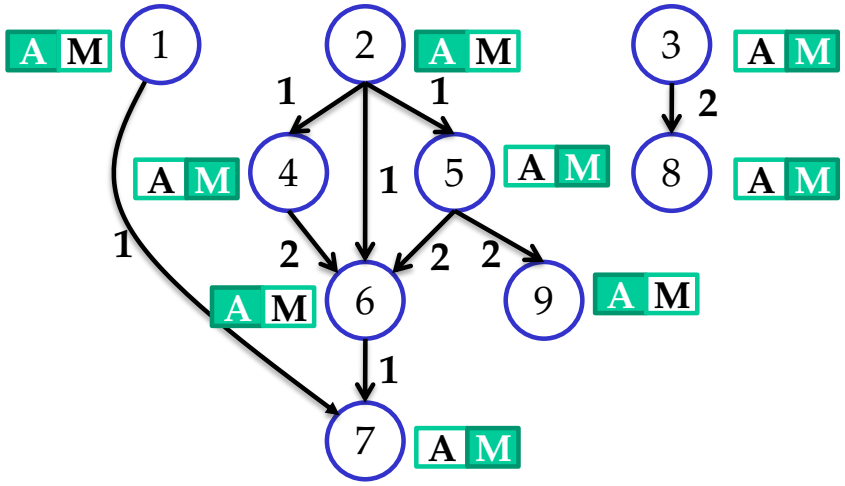
3.2 带延迟的数据依赖图

- 使用数据依赖图 G=(N,E)表示数据依赖关系
  - 节点: 指令
    - 节点集合N中的每一个节点n有一个资源预约表RT_n
  - 边(s1,s2)表示S1和S2之间存在依赖
    - S1必须在S2之前执行
    - 用延迟d_e标记每一条边，表示S1流出后，S2至少需要延迟d_e拍后才能流出

3.2 带延迟的数据依赖图

- 例子

1	addi t2, t1, 1
2	addi sp, sp,12
3	sw t0, 0(t6)
4	lw t3, -4(sp)
5	lw t4, -8(sp)
6	addi sp, sp, 8
7	sw t2, 0(sp)
8	lw t5, 0(t6)
9	addi t4, t4, 1



假设：

latency (ALU op) = 1

latency (MEM op) = 2

4. 表调度

- 一种贪婪的启发式方法
- 一种局部调度，以基本块为单位，调度基本块中的指令序列（指令序列中没有分支）
- 能够发现合理的调度，容易修改以适应计算机体系结构的改变
  - 20世纪70年代末开始成为指令调度的主要范式

4.1 基本思想

构建依赖图G

候选调度集合 ← G中所有根节点 (没有入边的节点)

while(当候选调度集合 ≠ ∅)

从候选调度集合中选择一条指令*s*

调度*s*

//将s从候选调度集合中删除

Candidates ← Candidates – {*s*}

//候选调度集合 =候选调度集合并上“曝露”的节点

Candidates ← Candidates ∪ “exposed” nodes

根据启发信息按序选择指令

将前驱都被调度了的节点加入候选调度集合

4.2 启发信息

- 表调度算法不会回溯
  - 对依赖图中的每个节点只进行一次调度
- 根据优先级选择下一个进行调度的节点
- 给节点置优先级时，需考虑下面问题
  - 关键路径
  - 节点后继数
  - 资源需求
  - 是否与后继有互锁

4.2 启发信息

- 关键路径
  - 是依赖图中最长的路径
  - 用来作为优先级函数的度量可以是节点的高度，即从节点开始的最长路径的长度
- 节点后继数
  - 后继越多，表示该节点调度完成后，曝露的可被调度的节点越多，下一次调度的候选更多，更具灵活性

4.2 启发信息

- 资源需求
  - 如可用的寄存器，使用较多寄存器的节点具有较高优先级，调度该节点可以减少寄存器压力
- 是否与后继有互锁
  - 是否和有向无环图中其后继互锁(硬件上的互锁)，调度该节点，使得与其互锁的后继可以被调度执行

4.3 表调度算法

构建数据依赖图G

为每一条指令置优先级

创建包含符合下面条件的指令列表，在列表中，指令按优先级排序

- 指令的所有前驱已经被调度
- 所有延迟都已经满足

循环，直到所有指令都被调度

从列表中删除具有最高优先级的指令

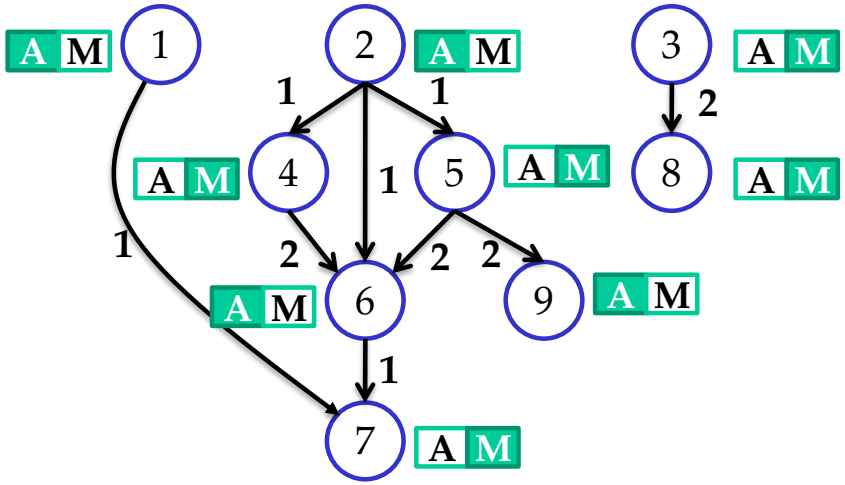
调度该指令，将指令s放在满足其前驱和资源约束的指令槽中

将新的符合调度条件的指令加入到列表（列表需要重新按优先级排序）

4.4 贪婪的表调度实例

- 构建依赖图

1	addi t2, t1, 1
2	addi sp, sp,12
3	sw t0, 0(t6)
4	lw t3, -4(sp)
5	lw t4, -8(sp)
6	addi sp, sp, 8
7	sw 0(sp), t2
8	lw 0(t6), t5
9	addi t4, t4, 1

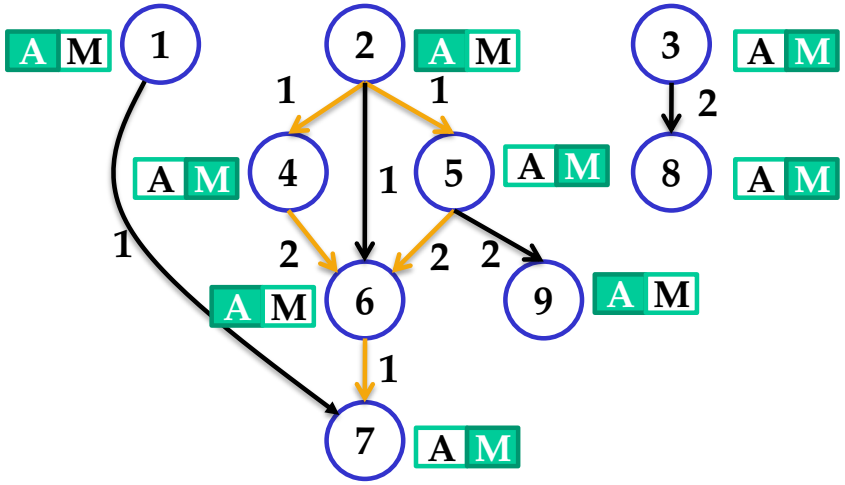


假设：

latency (ALU op) = 1

latency (MEM op) = 2

4.4 贪婪的表调度实例

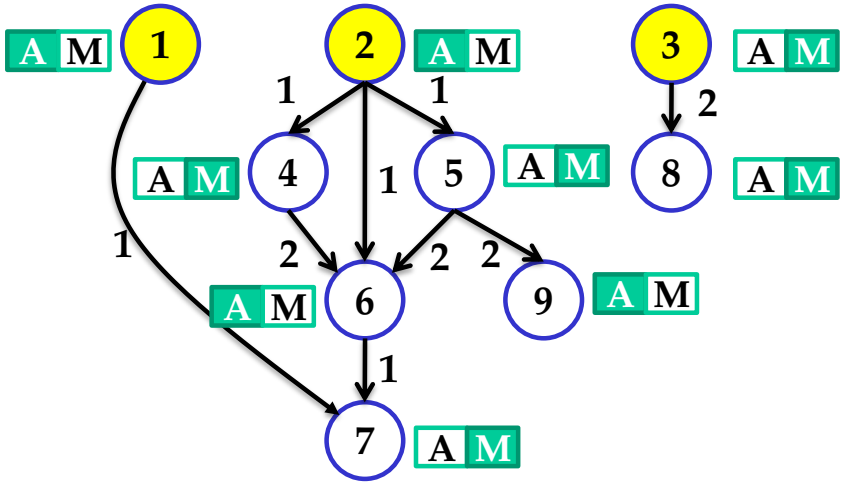


Critical Path: 2-4-6-7
2-5-6-7

Candidates list: 2→3→1
Schedule :

A	2		
M	3		

4.4 贪婪的表调度实例

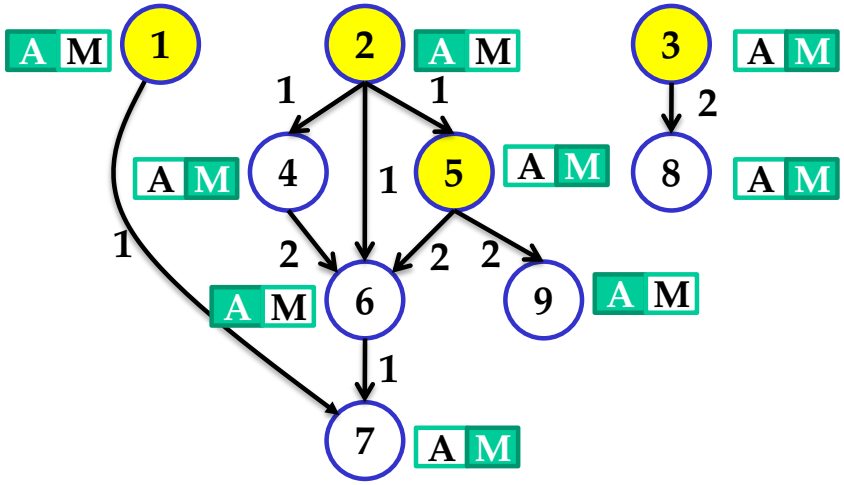


Critical Path: 2-4-6-7
2-5-6-7

Candidates list: 5→4→8
Schedule:

A	2	1	
M	3		

4.4 贪婪的表调度实例

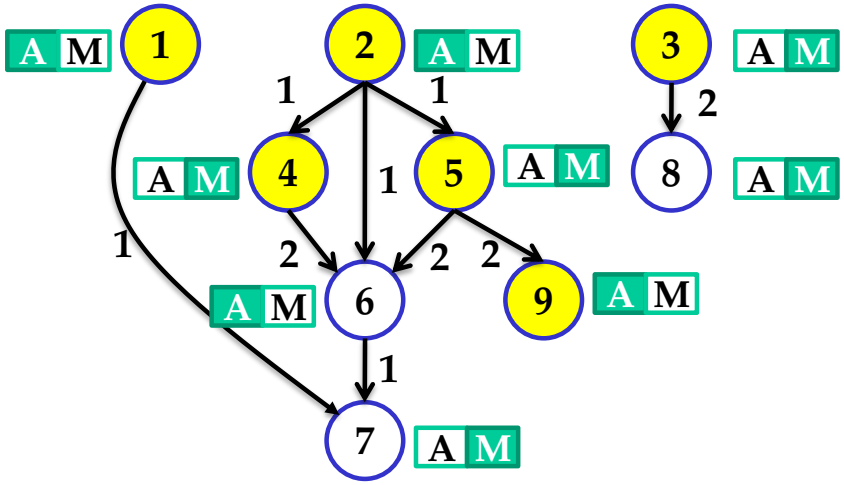


Critical Path: 2-4-6-7
2-5-6-7

Candidates list: 4→8→9
Schedule:

A	2	1			
M	3		5		

4.4 贪婪的表调度实例

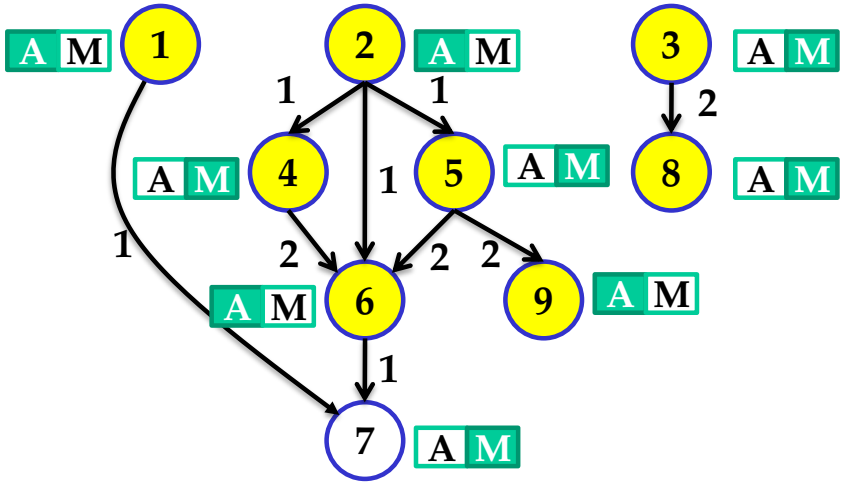


Critical Path: 2-4-6-7
2-5-6-7

Candidates list: 8→6
Schedule:

A	2	1			9		
M	3		5		4		

4.4 贪婪的表调度实例



Critical Path: 2-4-6-7
2-5-6-7

Candidates list: 7
Schedule:

A	2	1			9		6			
M	3		5		4		8		7	

10 cycles

4.5 表调度复杂度

- 指令数的二次方
  - 构建依赖图是$O(n^2)$
  - 每一次调度可能需要检查每一条指令 O(n)
  - 实践中: 接近于线性

5. 基于trace的全局调度

- 一个基本块包含的指令条数可能不多，无法挖掘出足够的指令调度
  - 通过循环展开、循环合并等优化技术扩大基本块
  - 实现跨基本块的调度，即全局调度
- 跨基本块的调度，编译器不仅需要考虑数据依赖，还要考虑控制依赖
- 轨迹（Trace）调度是一种常用的全局调度方法
  - Fisher，1981

5.1 基本块执行概率

- 为控制流图中的节点（基本块）或者边标记执行概率
  - 边的执行频率精度高于结点执行频率
- 执行概率的获得
  - 静态分支预测
  - 预先插桩→执行获取执行测试→计算执行概率
- 静态分支预测

```
P = malloc(N);
if (p == NULL)
{
   //错误处理
}
```

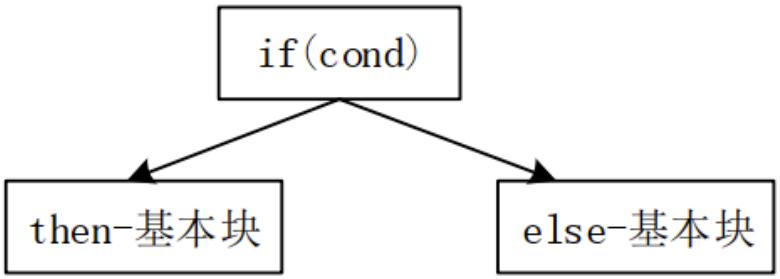
p!= NULL 概率远远大于p == NULL

```
for(i=0;i<N;i++)
{
    ….
}
```

大部分情况下，循环会执行多次

5.1 基本块执行概率

- 静态分支预测



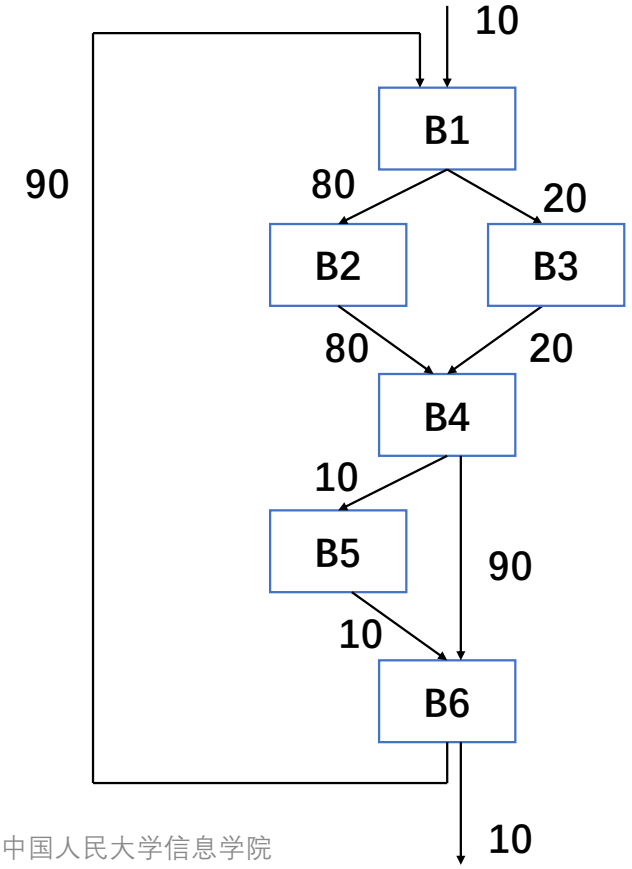
	条件	执行概率
指针	!= NULL	85
操作符	> 0	79
	!=	71
	浮点比较	90
分支后继	函数调用	71
	循环	95

分支预测启发信息

5.2 加权的CFG

- 节点（基本块）或者边标记执行概率的控制流图

```
B1:   x=x+3
      y= x+y
      if( x< 0) goto B3

B2:   z=x+z
      x=y
      goto B4

B3:   z=x-y
      y=x

B4:   z=z*x
      if(z>0) goto B6

B5:   x=z
      z=z-y

B6:  y=z
      if(z>x) goto B1
…….
```



5.3 轨迹

- 轨迹
  - 由控制流图中基本块构成的无环路径
- 轨迹调度
  - 将一个轨迹看作是一个基本块进行调度
  - 在轨迹的入口和出口插入必要的补偿代码

- 轨迹调度目标是使程序中频繁执行的轨迹执行得更快

```
i = 0
标记所有基本块为未访问
while (还有未访问的基本块) {
    seed = 未访问基本块中执行频率最高的基本块
    trace[i] += seed;
    标记此基本块已被访问
    current = seed
    /* 向前扩展轨迹*/
    while (1) {
        next = best_successor_of(current)
        if (next == 0)  break;// 当前轨迹已经不能进一步扩展
        trace[i] += next
        标记找到的最佳后继next已被访问
        current = next
    }
    i++;  //寻找下一条轨迹
}
```

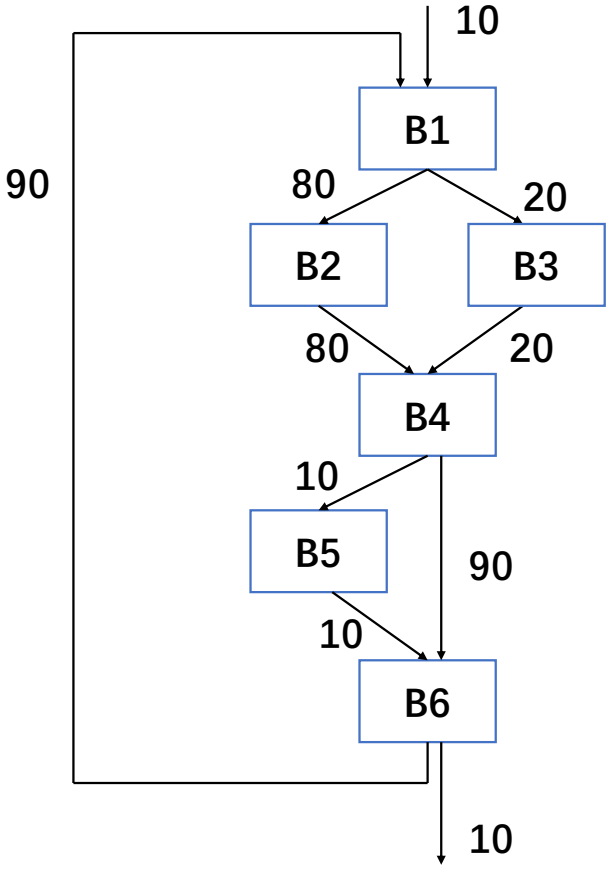
构建轨迹的关键就是寻找当前节点的最佳后继，扩展轨迹。当没有找到最佳后继节点时，轨迹就不再扩展

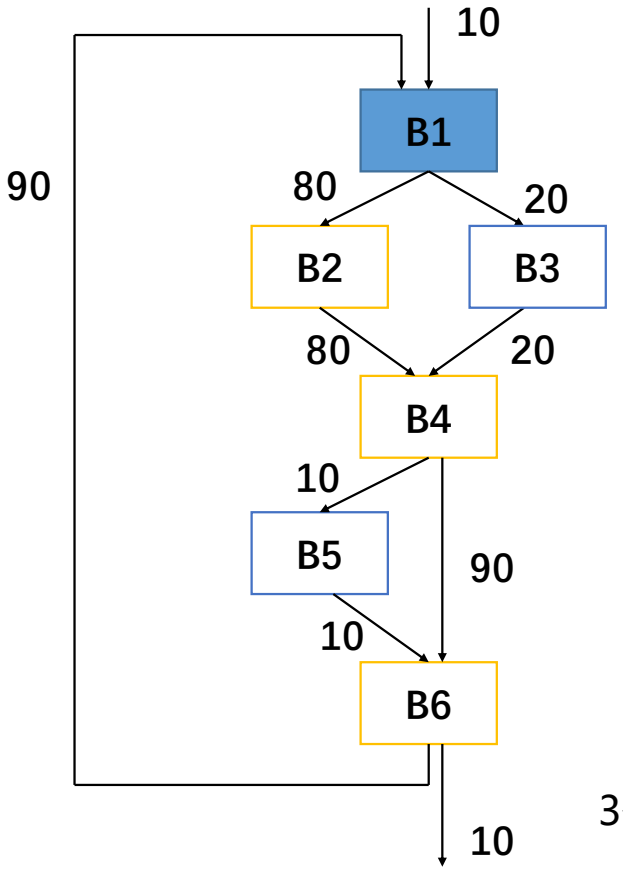
5.3.1 构建轨迹

```
best_successor_of(BB){
    e = 离开基本块BB的具有最高执行概率的边;
    if (e是向后边)  //轨迹是无环路径
        return 0;

    if (probability(e) <= 阈值)
        return 0;

d = 边e的终点基本块;
    if (d is visited) then  //一个基本块只能出现在一条轨迹中
        return 0;

    return d;
}
```

当节点没有执行概率大于设置的阈值的后继时，轨迹就不再扩展

5.3.2 轨迹示例





阈值：75

① Seed = B1

②B1最佳后继 B2
轨迹：B1-B2

③ B2最佳后继 B4
轨迹：B1-B2-B4

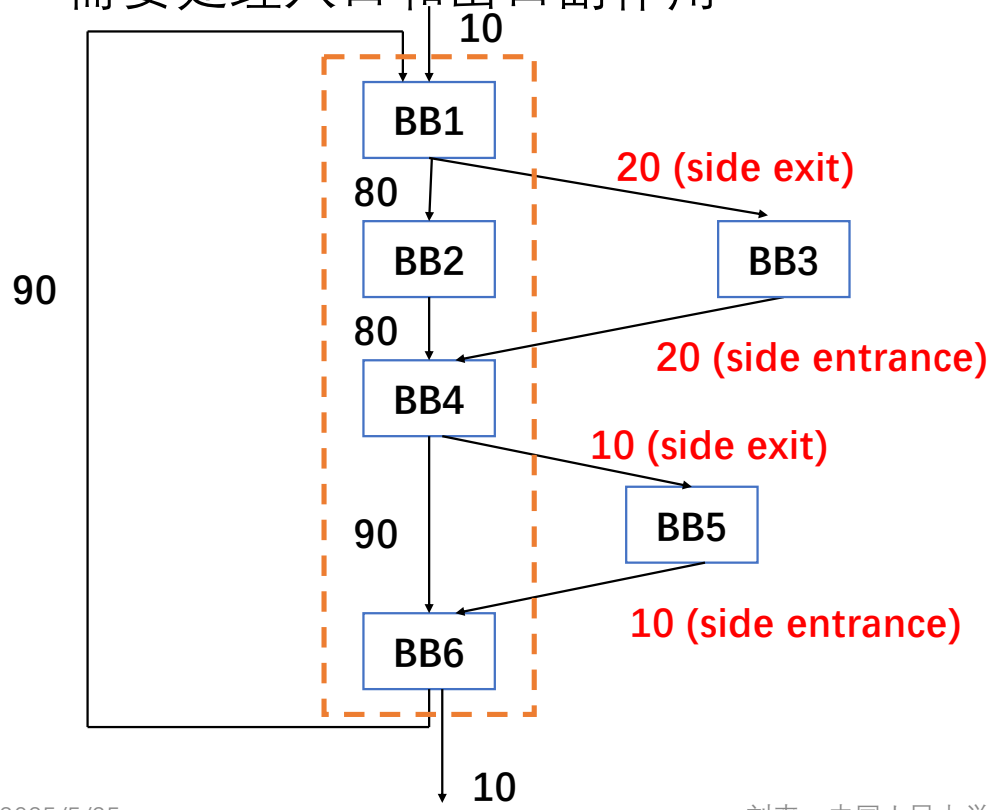
④B4最佳后继B6
轨迹：B1-B2-B4-B6

⑤B6执行频率>阈值的边是条向后边
轨迹：B1-B2-B4-B6

3条轨迹：{B1,B2,B4,B6}，{B3}以及{B5}

5.4 基于轨迹的调度

- 将轨迹看作是基本块调度
  - 需要处理入口和出口副作用



- 轨迹并不是完全顺序执行的指令序列
  - ✓ 有分支可以在中间离开轨迹,e.g B4跳转到B5
  - ✓ 也可以分支跳转到一条轨迹中间,e.g B3跳转到B4

5.4 基于轨迹的调度

- 将轨迹看作是基本块调度
  - 需要处理入口和出口副作用

- 语句y=x+y跨基本块被移动到了B2。由于B1→B3有一条边能够离开轨迹，如果实际运行是B1→B3，那么将导致语句y=x+y没有执行。因此，我们需要在B3插入补偿语句y=x+y

- B6中的语句y=z提前到B4中执行，由于存在路径B4->B5，B5->B6，此时也需要在B5尾部插入补偿代码

```
B1: x=x+3
    y= x+y
    if(x< 0) goto B3

B2: z=x+z
    x=y
    goto B4

B3: z=x-y
    y=x

B4: z=z*x
    if(z>0) goto B6

B5: x=z
    z=z-y

B6:  y=z
    if(z>x) goto B1
    .....
```

```
B1: x=x+3

    if(x < 0) goto B3

B2: y= x+y
    z=x+z
    x=y
    goto B4

B3: y= x+y //补偿代码
    z=x-y
    y=x

B4: z=z*x
    y=z
    If(z>0) goto B6

B5: x=z
    z=z-y
    y=z//补偿代码
B6:
    if(z>x) goto B1
    .....
```