机器无关代码优化
Part4：Mem2Reg

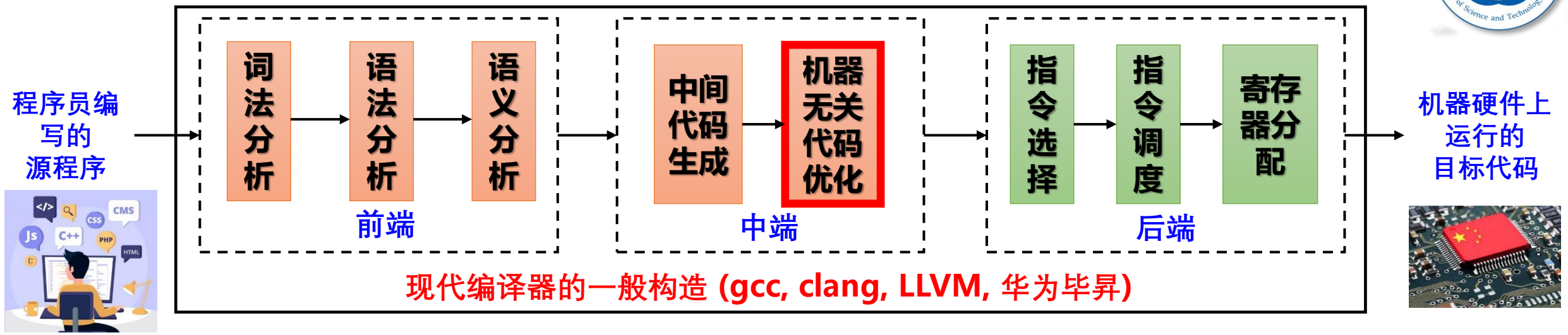
徐 伟

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

计算机科学与技术学院

2025年10月30日

本节提纲



- **Dominators**
- **Mem2Reg**

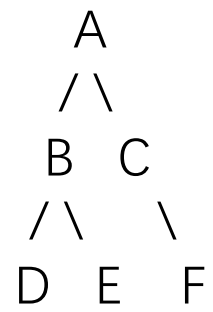
Dominators

❑逆后序遍历和先序遍历

❑先序遍历（Pre-order Traversal）

- **先序遍历的顺序是：**
  - 访问根节点
  - 遍历左子树
  - 遍历右子树

遍历顺序： 根 -> 左 -> 右



先序遍历的结果是：**A -> B -> D -> E -> C -> F**

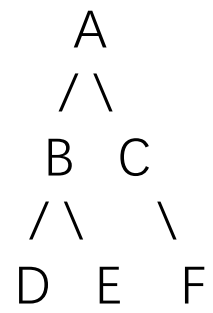
Dominators

❑逆后序遍历和先序遍历

❑逆后序遍历（Reverse Post-order Traversal）

- **后序遍历的顺序：**
  - 遍历左子树
  - 遍历右子树
  - 访问根节点
- **逆后续遍历的顺序**
  - 访问根节点
  - 遍历右子树
  - 遍历左子树

遍历顺序： 根 -> 右 -> 左



后序遍历的结果是： **D -> E -> B -> F -> C -> A**
逆后续遍历的结果是： **A -> C -> F -> B -> E -> D**

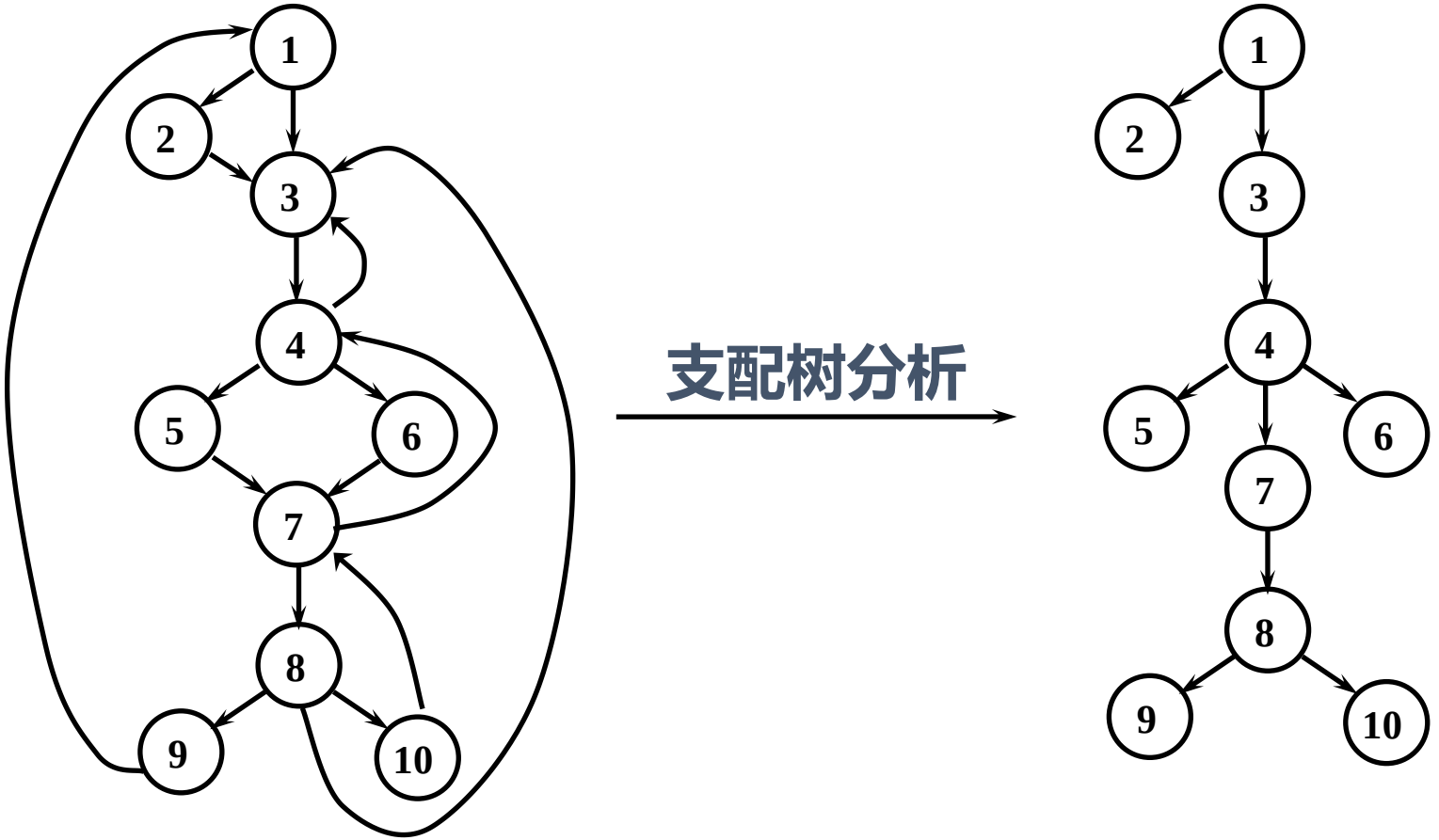
Dominators

- **逆后序遍历和先序遍历应用场景**
  - **先序遍历**
    - 用于构建树的复制
    - 用于序列化二叉树
    - 用于深度优先搜索（DFS）
  - **逆后续遍历**
    - 在图的深度优先搜索中，逆后序遍历可以用于拓扑排序（将图的节点按依赖关系排序）
    - 在某些算法中，逆后序遍历可以更好地处理依赖关系

- **先序遍历和逆后序遍历的主要区别在于子树的访问顺序不同，但它们都以根节点为起点**

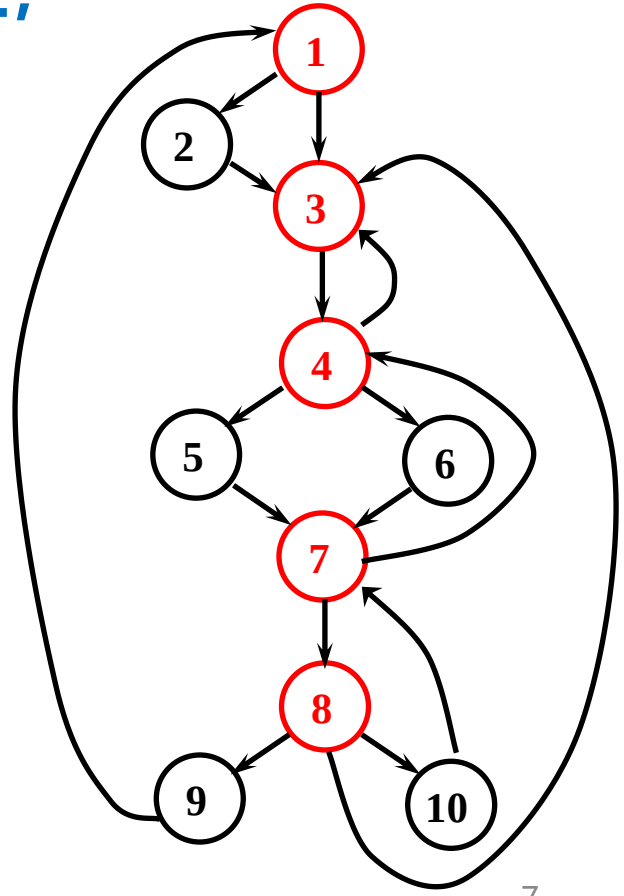
Dominators

- 什么是支配树?
  - 给出函数中基本块节点间的直接支配关系的树形数据结构



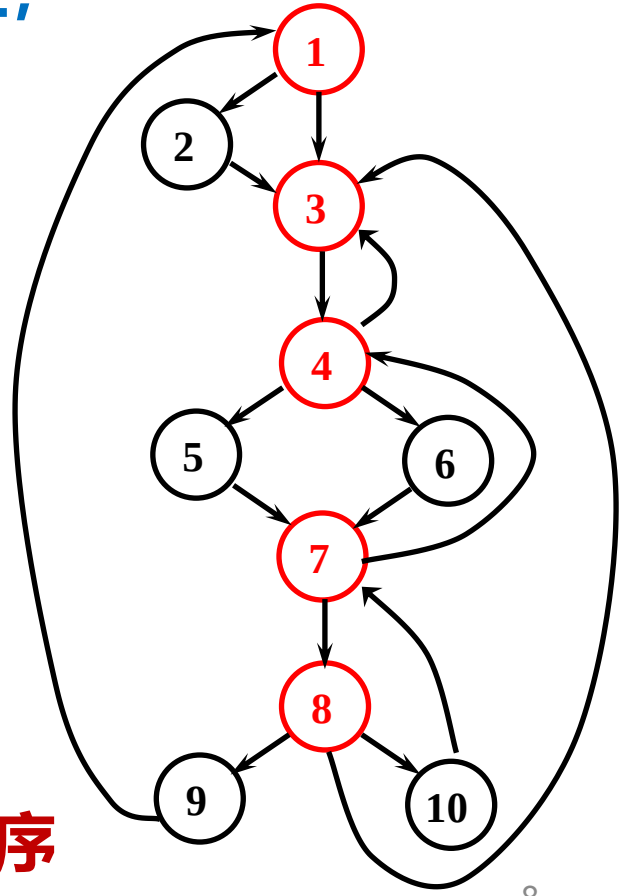
Dominators

- 什么是支配树?
  - 给出函数中基本块节点间的直接支配关系的树形数据结构
  - 支配：如果节点 n 位于从 CFG 入口节点到 b 的每条路径上，则称节点 n 支配 b，记作 n∈Dom(b)、n dom b
    - 如右图 1、3、4、7、8 均为 8 的支配节点



Dominators

- 什么是支配树?
  - 给出函数中基本块节点间的直接支配关系的树形数据结构
  - 支配：如果节点 n 位于从 CFG 入口节点到 b 的每条路径上，则称节点 n 支配 b，记作 n∈Dom(b)
    - 如右图 1、3、4、7、8 均为 8 的支配节点



注意到如果节点 m, n 都支配节点 b，那么：

a) m 支配 n

b) n 支配 m

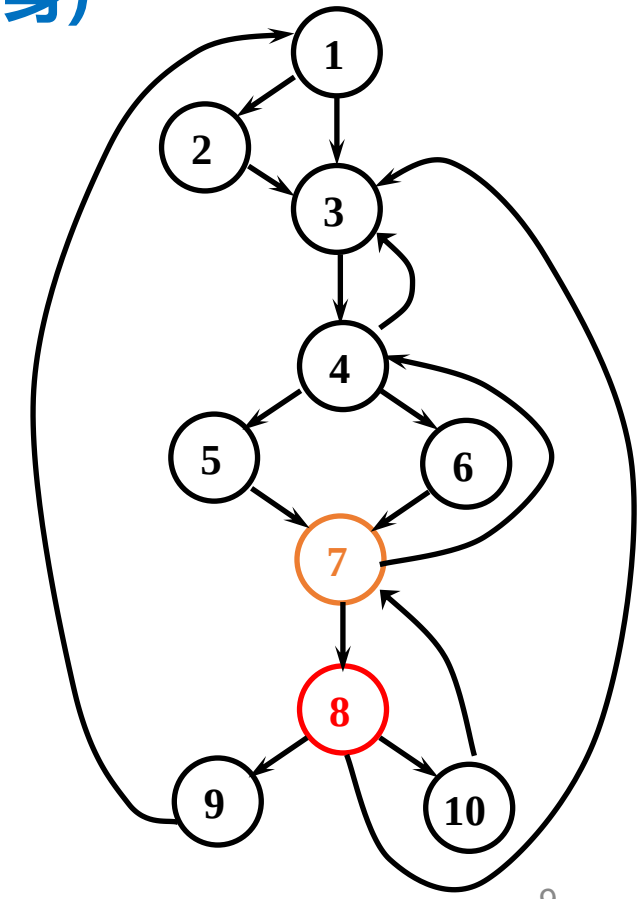
至少有一条成立

也就是说，所有 b 的支配节点均可以按照偏序关系排序

这个偏序关系就是节点间的支配关系，**支配关系一定满足逆后序**

Dominators

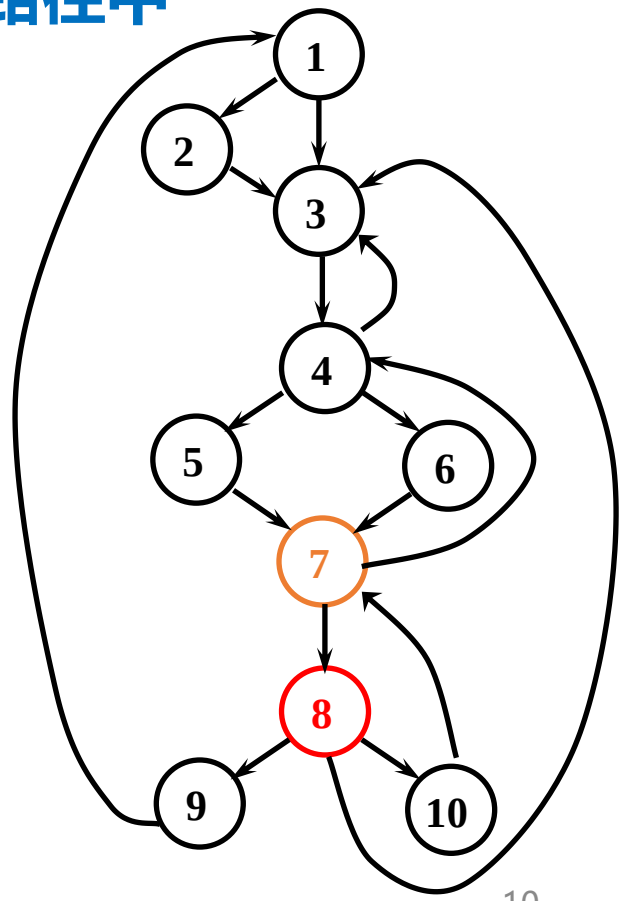
- 什么是支配树?
  - 给出函数中基本块节点间的直接支配关系的树形数据结构
  - 直接支配：如果 n 直接支配 b，且所有 b 的支配节点(除自身)都是 n 的支配节点，则称 n 直接支配 b
    记作 IDom(b) = n、　　n idom b
    - 例如图中 8 的直接支配节点为 7；7 的直接支配节点为 4



Dominators

- 什么是支配树?
  - 给出函数中基本块节点间的直接支配关系的树形数据结构
  - 直接支配:从入口结点到达b的任何路径（不含b）中，它是路径中最后一个支配n的节点

    记作 IDom(b) = n
    - 例如图中 8 的直接支配节点为 7；7 的直接支配节点为 4
    - 后面所说的顺序（如果没有指定）默认为逆后序



Dominators

- 支配的性质
  - 自反性：每个节点支配自身
  - 传递性：如果 a dom b 且 b dom c，则 a dom c
  - 反对称性：如果 a dom b 且 b dom a，则 a=b

Dominators

❑寻找支配结点算法

- **计算流图中各个结点的所有支配结点**
  - $p_1, p_2, \ldots p_k$是n的所有前驱，且d!=n，那么d dom n当且仅当d dom p_i $(1 \le i \le k)$

- **一个结点的支配结点集合是它的所有前驱的支配结点集合的交集，再加上它自己**

- **前向数据流分析问题**

Dominators

❑前向与后向数据流分析对比

特征	前向数据流分析	后向数据流分析
方向	入口 → 出口	出口 → 入口
典型问题	到达定义、可用表达式	活跃变量分析
方程核心	**OUT[B] = gen_B ∪ (IN[B] - kill_B)**	**IN[B] = gen_B ∪ (OUT[B] - kill_B)**
初始值	入口节点初始化（如 IN[Entry] = ∅）	出口节点初始化（如 OUT[Exit] = ∅）
收敛条件	所有 OUT[B] 不再变化	所有 IN[B] 不再变化

Dominators

- 寻找支配结点算法
  - 求解数据流方程，得到各结点对应的支配结点集合
  - D(n) = OUT[n]

支配结点	
域	The power set of N
方向	Forward
传递函数	$f_B(x) = x \cup \{B\}$
边界条件	$OUT[ENTRY] = \{ENTRY\}$
交汇运算∧	∩
方程式	$OUT[B] = f_B(IN[B])$ $IN[B] = \wedge_{p \subset pred(B)} OUT[P]$
初始化设置	$OUT[B] = N$

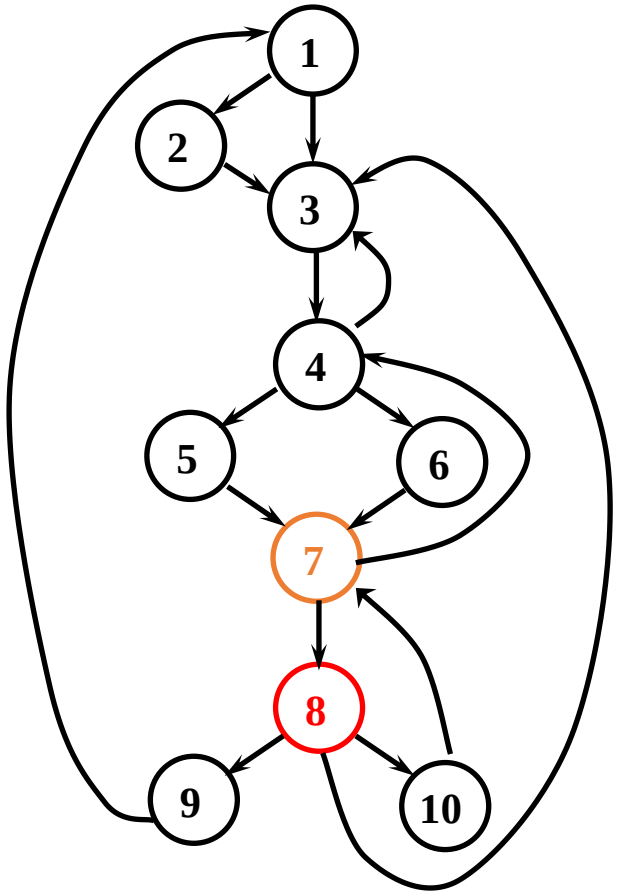
Dominators

❑寻找支配结点算法

$OUT[ENTRY] = \{ENTRY\}$
for(除ENTRY外的每个BB) $OUT[B] = N$
while(某个OUT值发生了改变)
    for(除ENTRY之外的每个BB){
        $IN[B] = \wedge_{p \subset pred(B)} OUT[P]$
        $OUT[B] = f_B(IN[B])$
    }

$$D(1) = \{1\}$$
$$D(2) = \{2\} \cup D(1) = \{1,2\}$$
$$D(3) = \{3\} \cup (\{1\} \cap \{1, 2\} \cap \{1, 2, \cdots, 10\} \cap \{1, 2, \cdots, 10\}) = \{1, 3\}$$

$$D(4) = \{4\} \cup (D(3) \cap D(7)) = \{4\} \cup (\{1,3\} \cap \{1,2,\ldots,10\}) = \{1,3,4\}$$
$$D(5) = \{5\} \cup D(4) = \{5\} \cup \{1,3,4\} = \{1,3,4,5\}$$
$$D(6) = \{6\} \cup D(4) = \{6\} \cup \{1,3,4\} = \{1,3,4,6\}$$
$$D(7) = \{7\} \cup (D(5) \cap D(6) \cap D(10))$$
$$= \{7\} \cup (\{1,3,4,5\} \cap \{1,3,4,6\} \cap \{1,2,\ldots,10\}) = \{1,3,4,7\}$$
$$D(8) = \{8\} \cup D(7) = \{8\} \cup \{1,3,4,7\} = \{1,3,4,7,8\}$$
$$D(9) = \{9\} \cup D(8) = \{9\} \cup \{1,3,4,7,8\} = \{1,3,4,7,8,9\}$$
$$D(10) = \{10\} \cup D(8) = \{10\} \cup \{1,3,4,7,8\} = \{1,3,4,7,8,10\}$$



Dominators

- 如何得到一棵支配树?

```
for all nodes, b /* initialize the dominators array */
    doms[b] ← Undefined
doms[start_node] ← start_node
Changed ← true
while (Changed)
    Changed ← false
    for all nodes, b, in reverse postorder (except start_node)
        new_idom ← first (processed) predecessor of b /* (pick one) */
        for all other predecessors, p, of b
            if doms[p] ≠ Undefined /* i.e., if doms[p] already calculated */
                new_idom ← intersect(p, new_idom)
        if doms[b] ≠ new_idom
            doms[b] ← new_idom
            Changed ← true
```

初始化

Dominators

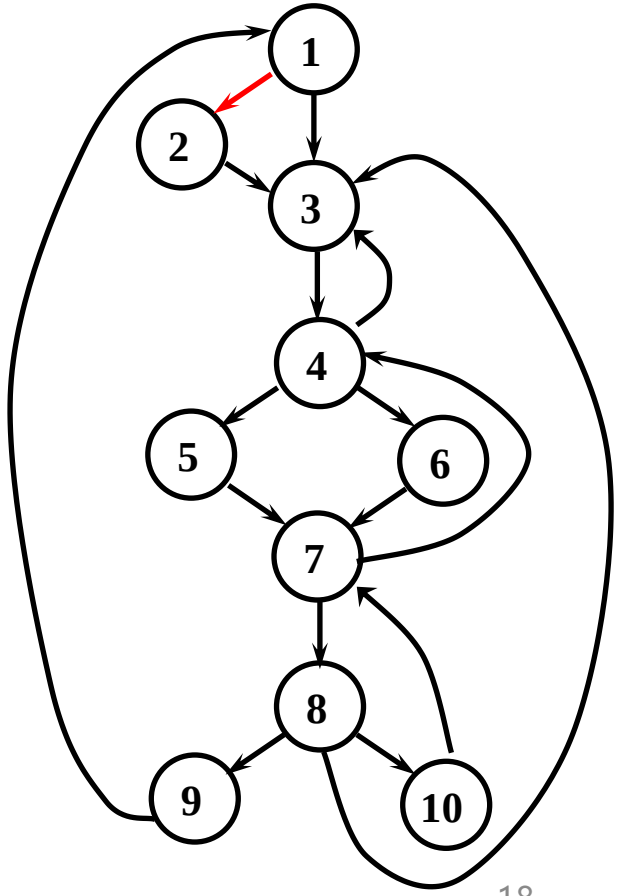
如何得到一棵支配树?

```
for all nodes, b /* initialize the dominators array */
    doms[b] ← Undefined
doms[start_node] ← start_node
Changed ← true
while (Changed)
    Changed ← false
    for all nodes, b, in reverse postorder (except start_node)
        new_idom ← first (processed) predecessor of b /* (pick one) */
        for all other predecessors, p, of b
            if doms[p] ≠ Undefined /* i.e., if doms[p] already calculated */
                new_idom ← intersect(p, new_idom)
        if doms[b] ≠ new_idom
            doms[b] ← new_idom
            Changed ← true
```

按照逆后序遍历节点，记当前节点为b
将 b 的 (逆后序) 最小前驱与其余前驱(直接支配节点非空)
的共同最大支配节点记作 (暂时的) 直接支配节点
如果任一直接支配节点有改变，则迭代计算新的支配节点集合

Dominators

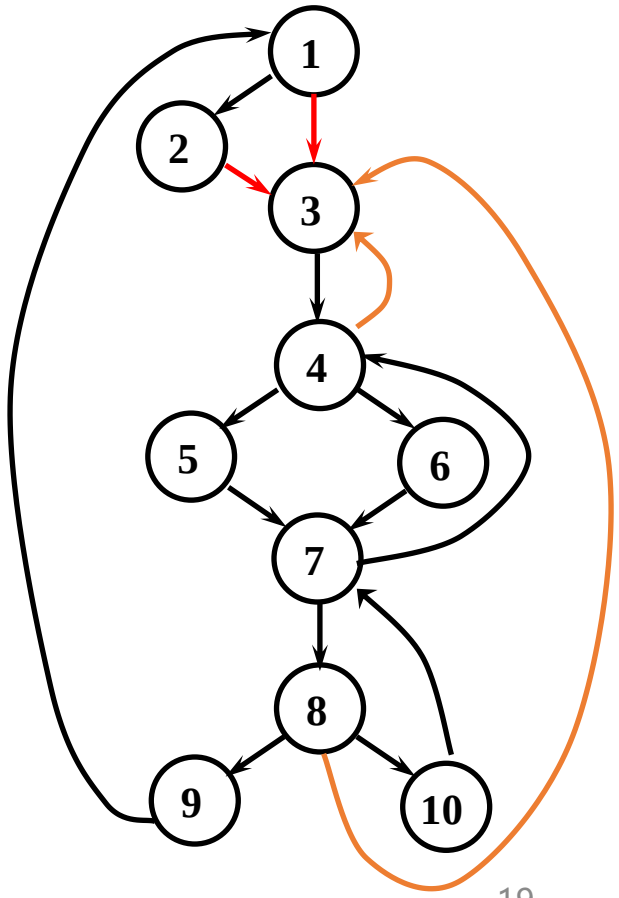
序号	直接支配节点
1	1
2	1
3	Undef
4	Undef
5	Undef
6	Undef
7	Undef
8	Undef
9	Undef
10	Undef



Dominators

序号	直接支配节点
1	1
2	1
3	?
4	Undef
5	Undef
6	Undef
7	Undef
8	Undef

节点 3 有四个前驱节点：1、2、4、8
new_idom = 1



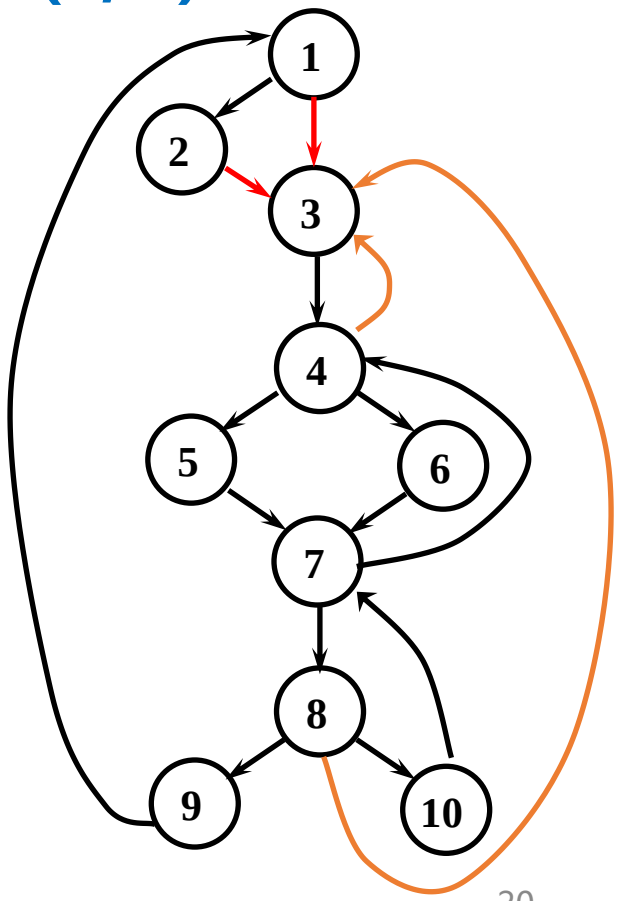

```
for all nodes, b, in reverse postorder (except start_node)
    new_idom ← first (processed) predecessor of b /* (pick one) */
    for all other predecessors, p, of b
        if doms[p] ≠ Undefined /* i.e., if doms[p] already calculated */
            new_idom ← intersect(p, new_idom)
    if doms[b] ≠ new_idom
        doms[b] ← new_idom
        Changed ← true
```

Dominators

序号	直接支配节点
1	1
2	1
3	?
4	Undef
5	Undef
6	Undef
7	Undef
8	Undef

节点 3 有四个前驱节点：1、2、4、8
p = 2
new_idom = intersect (2, 1)
= 1

其余两个节点(4、8)
doms 尚未定义

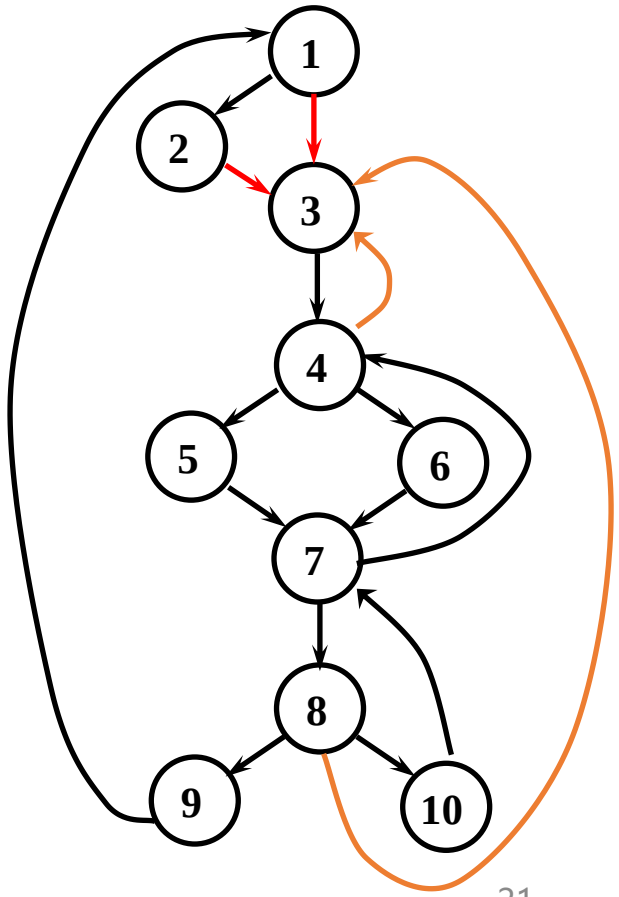



```
for all nodes, b, in reverse postorder (except start_node)
    new_idom ← first (processed) predecessor of b /* (pick one) */
    for all other predecessors, p, of b
        if doms[p] ≠ Undefined /* i.e., if doms[p] already calculated */
            new_idom ← intersect(p, new_idom)
    if doms[b] ≠ new_idom
        doms[b] ← new_idom
        Changed ← true
```

Dominators

序号	直接支配节点
1	1
2	1
3	1
4	Undef
5	Undef
6	Undef
7	Undef
8	Undef

节点 3 有四个前驱节点：1、2、4、8
doms[3] = 1

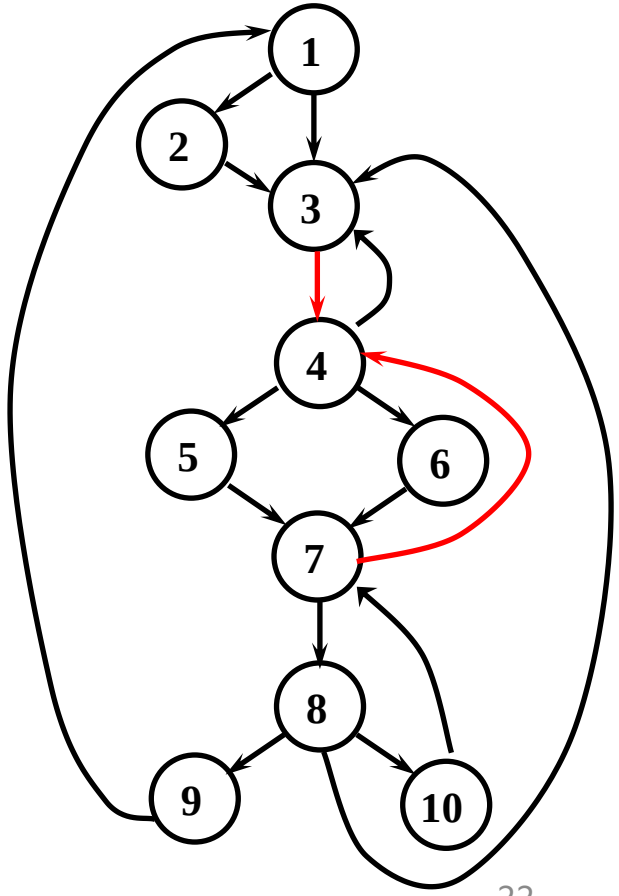



```
for all nodes, b, in reverse postorder (except start_node)
    new_idom ← first (processed) predecessor of b /* (pick one) */
    for all other predecessors, p, of b
        if doms[p] ≠ Undefined /* i.e., if doms[p] already calculated */
            new_idom ← intersect(p, new_idom)
    if doms[b] ≠ new_idom
        doms[b] ← new_idom
        Changed ← true
```

Dominators

序号	直接支配节点
1-3	1
4	?
5	Undef
6	Undef
7	Undef
8	Undef
9	Undef
10	Undef

节点 4 有两个前驱节点：3、7
new_idom = 3

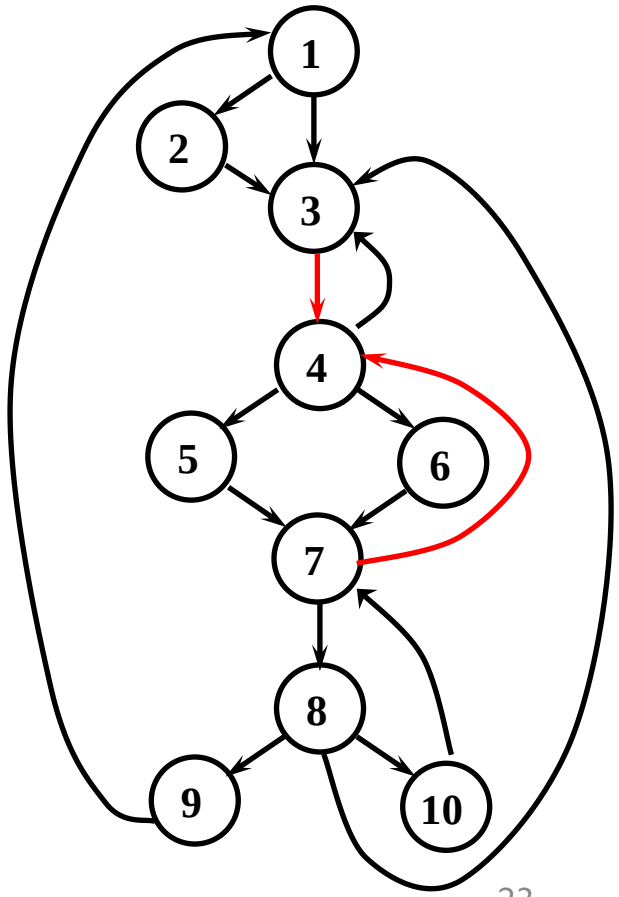



```
for all nodes, b, in reverse postorder (except start_node)
    new_idom ← first (processed) predecessor of b /* (pick one) */
    for all other predecessors, p, of b
        if doms[p] ≠ Undefined /* i.e., if doms[p] already calculated */
            new_idom ← intersect(p, new_idom)
    if doms[b] ≠ new_idom
        doms[b] ← new_idom
        Changed ← true
```

Dominators

序号	直接支配节点
1-3	1
4	?
5	Undef
6	Undef
7	Undef
8	Undef
9	Undef
10	Undef

节点 4 有两个前驱节点：3、7
new_idom = 3
p = $\bar{7}$
doms[7] 尚未定义

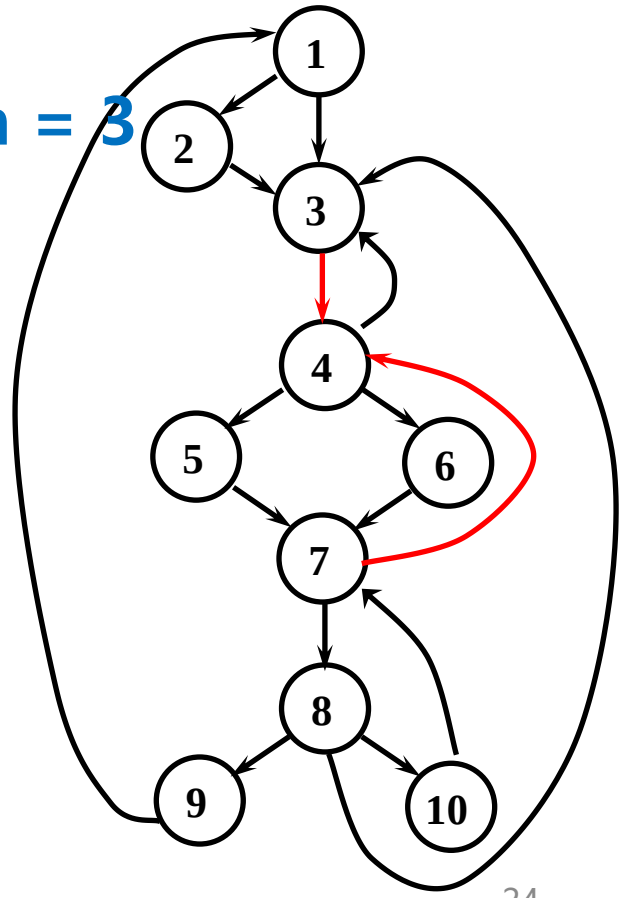



```
for all nodes, b, in reverse postorder (except start_node)
    new_idom ← first (processed) predecessor of b /* (pick one) */
    for all other predecessors, p, of b
        if doms[p] ≠ Undefined /* i.e., if doms[p] already calculated */
            new_idom ← intersect(p, new_idom)
    if doms[b] ≠ new_idom
        doms[b] ← new_idom
        Changed ← true
```

Dominators

序号	直接支配节点
1-3	1
4	3
5	Undef
6	Undef
7	Undef
8	Undef
9	Undef
10	Undef

节点 4 有两个前驱节点：3、7
new_idom = 3
p = 7
doms[7] 尚未定义
doms[4] = new_idom = 3

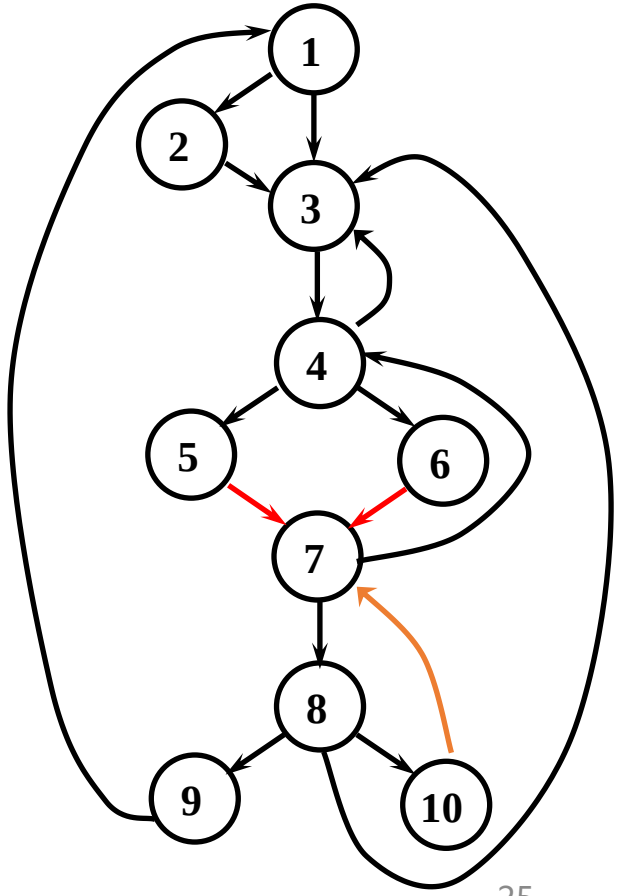



```
for all nodes, b, in reverse postorder (except start_node)
    new_idom ← first (processed) predecessor of b /* (pick one) */
    for all other predecessors, p, of b
        if doms[p] ≠ Undefined /* i.e., if doms[p] already calculated */
            new_idom ← intersect(p, new_idom)
    if doms[b] ≠ new_idom
        doms[b] ← new_idom
        Changed ← true
```

Dominators

序号	直接支配节点
1-3	1
4	3
5	4
6	4
7	?
8	Undef
9	Undef
10	Undef

节点 7 有三个前驱节点：5、6、10
new_idom = 5



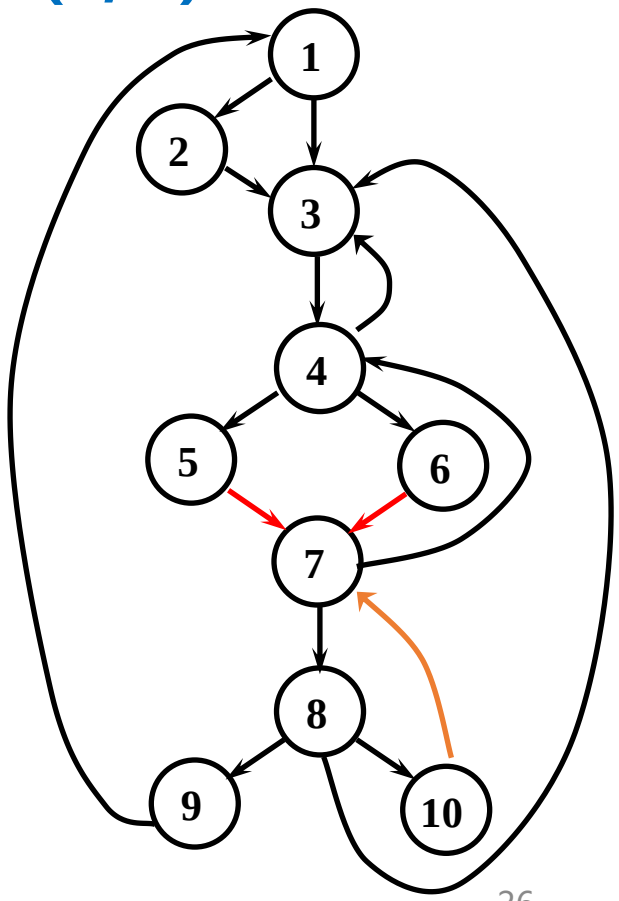

```
for all nodes, b, in reverse postorder (except start_node)
    new_idom ← first (processed) predecessor of b /* (pick one) */
    for all other predecessors, p, of b
        if doms[p] ≠ Undefined /* i.e., if doms[p] already calculated */
            new_idom ← intersect(p, new_idom)
    if doms[b] ≠ new_idom
        doms[b] ← new_idom
        Changed ← true
```

Dominators

序号	直接支配节点
1-3	1
4	3
5	4
6	4
7	?
8	Undef
9	Undef
10	Undef

节点 7 有三个前驱节点：5、6、10
p = 6
new_idom = intersect (5, 6)
= 4

doms[10] 尚未定义

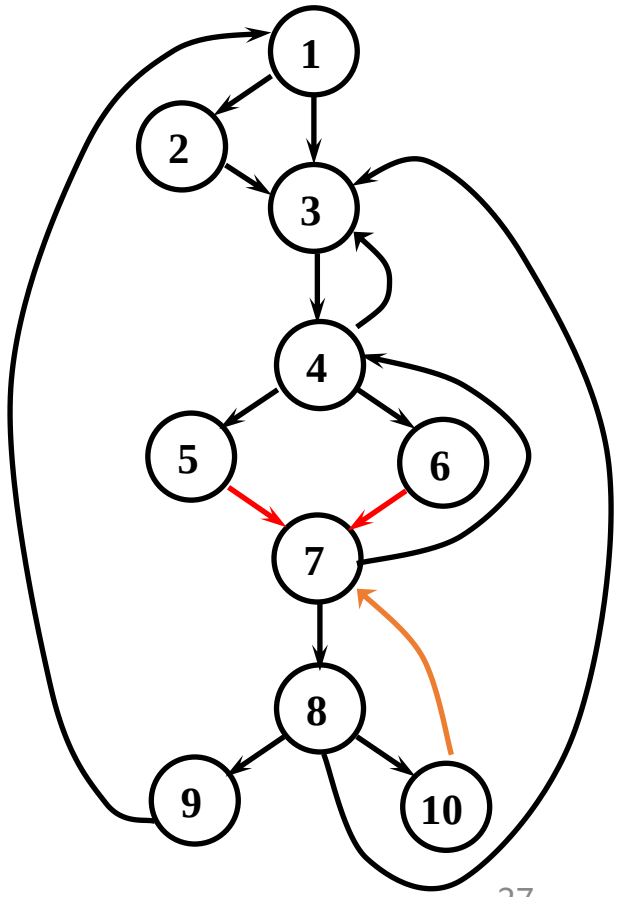



```
for all nodes, b, in reverse postorder (except start_node)
    new_idom ← first (processed) predecessor of b /* (pick one) */
    for all other predecessors, p, of b
        if doms[p] ≠ Undefined /* i.e., if doms[p] already calculated */
            new_idom ← intersect(p, new_idom)
    if doms[b] ≠ new_idom
        doms[b] ← new_idom
        Changed ← true
```

Dominators

序号	直接支配节点
1-3	1
4	3
5	4
6	4
7	4
8	Undef
9	Undef
10	Undef

节点 7 有三个前驱节点：5、6、10
doms[7] = 4

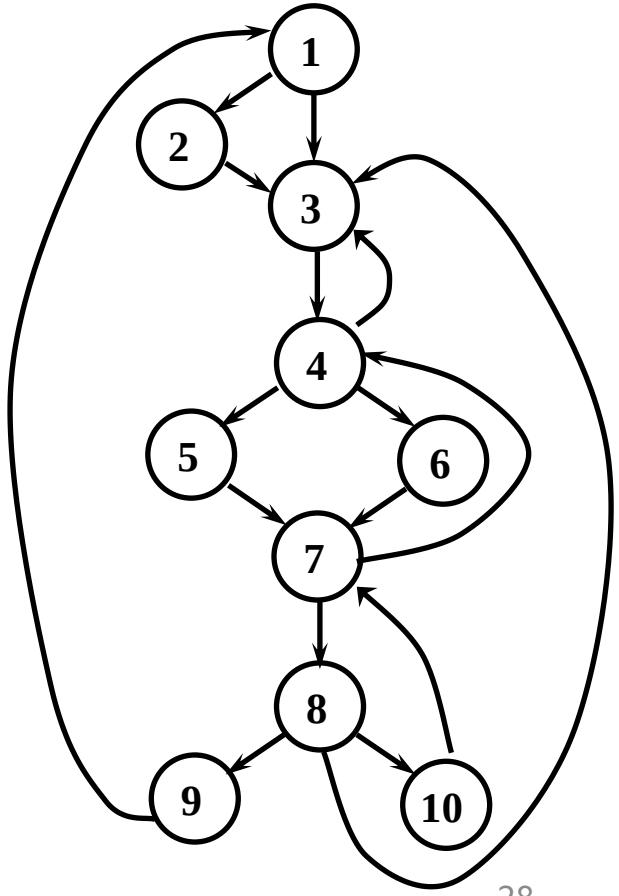



```
for all nodes, b, in reverse postorder (except start_node)
    new_idom ← first (processed) predecessor of b /* (pick one) */
    for all other predecessors, p, of b
        if doms[p] ≠ Undefined /* i.e., if doms[p] already calculated */
            new_idom ← intersect(p, new_idom)
    if doms[b] ≠ new_idom
        doms[b] ← new_idom
        Changed ← true
```

Dominators

序号	直接支配节点
1	1
2	1
3	1
4	3
5	4
6	4
7	4
8	7
9	8
10	8

在逆后序遍历过程中发现doms被改变，进入下一次遍历

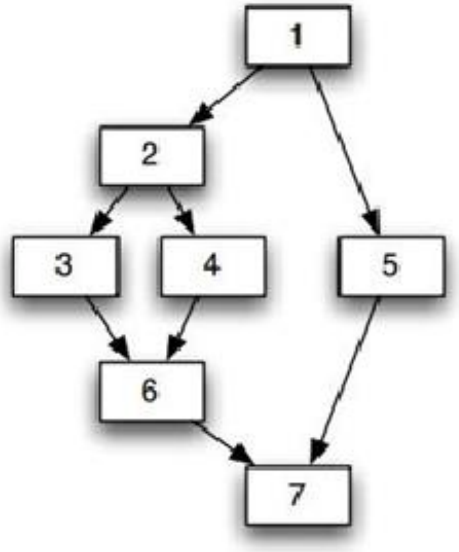


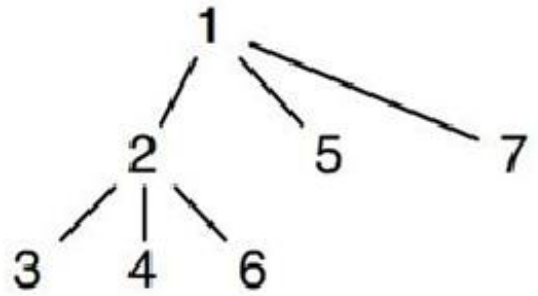
Dominators

- 至此，已经得到了直接支配节点的集合(idom_)
- 据此可以得到函数的支配树与支配边界

Dominators

- 支配树
  - 对于除entry外每一个结点u，从idom(u)向u连边，便构成了一个有n个结点，n-1条边的有向图
  - 支配关系一定不会构成循环，也就是这些边一定不构成环
  - 得到的图事实上是一棵树，称这颗树为原图的支配树





Dominators

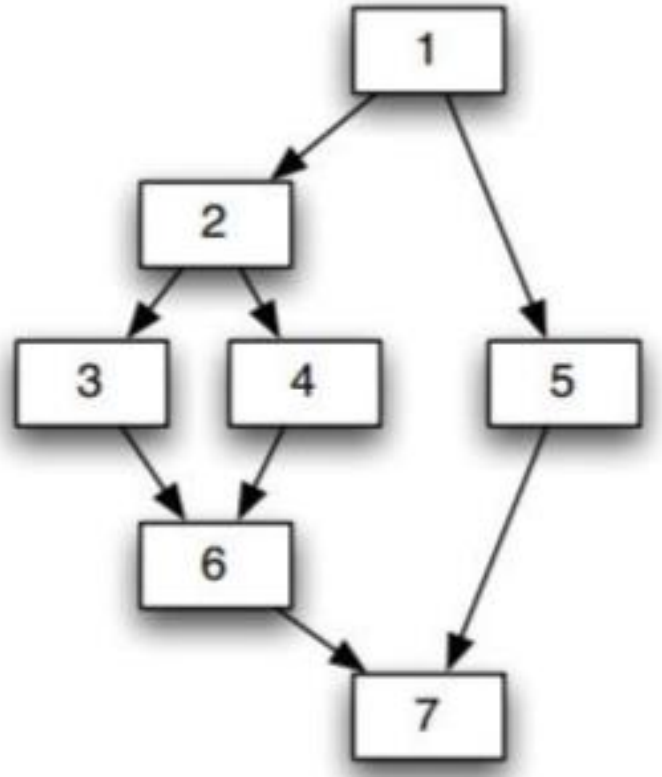
- 严格支配（Strictly Dominate）

- 在有向图中，节点 d 严格支配节点 n，记作 d sdom n，是指满足以下两个条件：
  - d 支配 n，即从起点到 n 的所有路径都经过 d
  - d 不等于 n
- 换句话说，严格支配关系排除了节点自身支配自己的情况

- 严格支配 sdom，即 d≠m ，且 d dom m，则称 d sdom m
- 直接支配 idom，在节点 n 的严格支配集 sdom(n) 中，离 n 最近的节点称为 n 的直接支配点

Dominators

- 支配边界（Dominance Frontier）
  - 支配边界是一系列节点的集合，记为 W
  - 某节点 x 的支配边界应满足如下条件，x 是 w 的前驱节点，但 x 不是 w 的严格支配节点（即不可以自己支配自己）

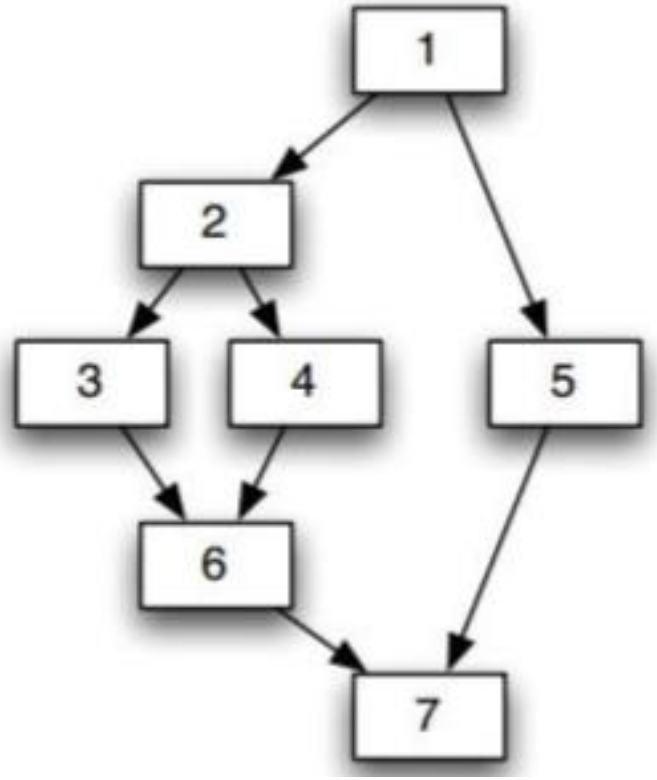


	1	2	3	4	5	6	7
Sdom	NULL	1	1，2	1，2	1	1，2	1
支配边界							

Dominators

- 如何得到支配边界?

```
//计算CFG中每个节点的支配边界
for (a, b) in CFG edges do:
        x <- a;
        while x dose not strictly dominate b do:
                DF(x) = (DF(x) ∪ b);
                x = immediate dominator(x);
```



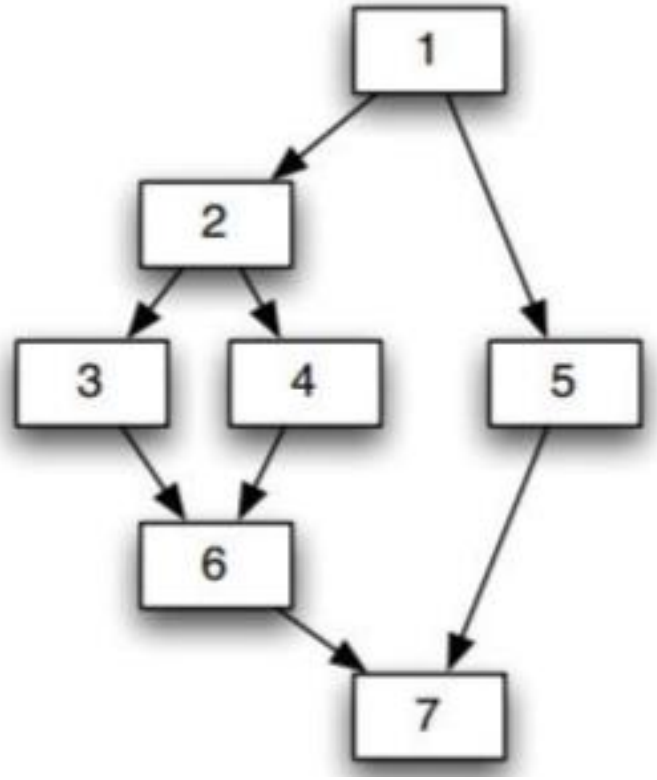
	1	2	3	4	5	6	7
Sdom	NULL	1	1，2	1，2	1	1，2	1
支配边界	NULL						

1->2 严格支配 DF[1] = NULL
1->5 严格支配 DF[1] = NULL

Dominators

❑如何得到支配边界?

```
//计算CFG中每个节点的支配边界
for (a, b) in CFG edges do:
        x <- a;
        while x dose not strictly dominate b do:
                DF(x) = (DF(x) ∪ b);
                x = immediate dominator(x);
```



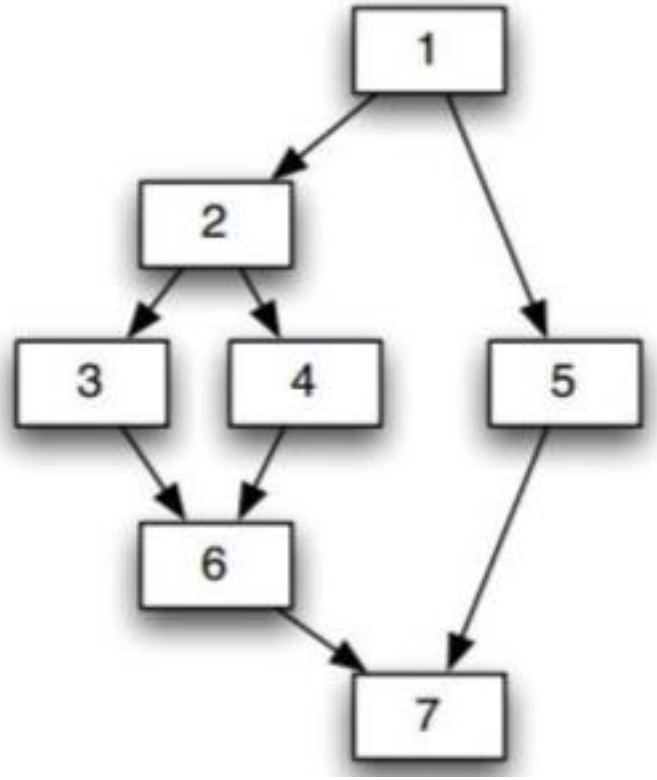
	1	2	3	4	5	6	7
Sdom	NULL	1	1，2	1，2	1	1，2	1
支配边界	NULL	NULL					

2->3 严格支配 DF[2] = NULL
2->4 严格支配 DF[2] = NULL

Dominators

- 如何得到支配边界?

```
//计算CFG中每个节点的支配边界
for (a, b) in CFG edges do:
        x <- a;
        while x dose not strictly dominate b do:
                DF(x) = (DF(x) ∪ b);
                x = immediate dominator(x);
```



	1	2	3	4	5	6	7
Sdom	NULL	1	1，2	1，2	1	1，2	1
支配边界	NULL	NULL	6	6			

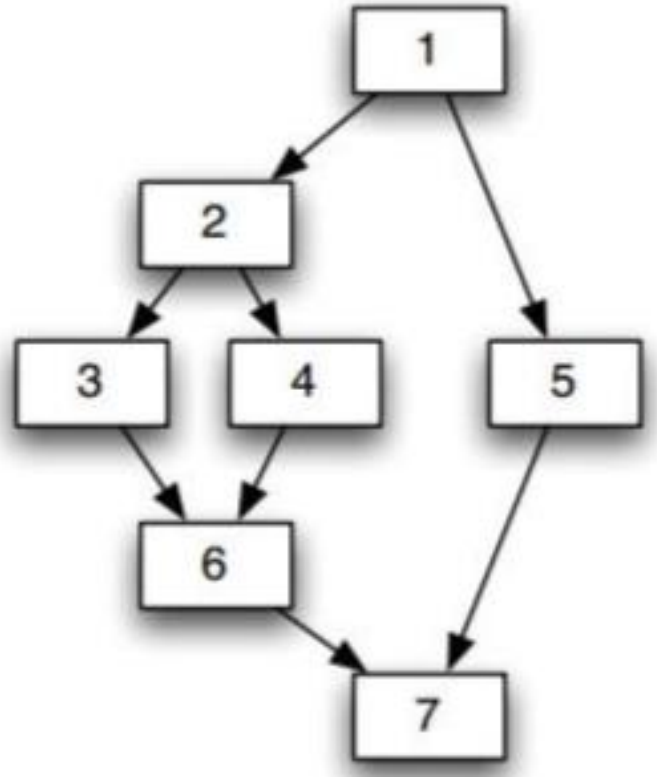
3->6 存在支配边界 DF[3] = 6
x = idom(3) = 2，严格支配

4->6 存在支配边界 DF[4] = 6
x = idom(3) = 2，严格支配

Dominators

- 如何得到支配边界?

```
//计算CFG中每个节点的支配边界
for (a, b) in CFG edges do:
        x <- a;
        while x dose not strictly dominate b do:
                DF(x) = (DF(x) ∪ b);
                x = immediate dominator(x);
```



	1	2	3	4	5	6	7
Sdom	NULL	1	1，2	1，2	1	1，2	1
支配边界	NULL	7	6	6		7	

6->7 存在支配边界 DF[6] = 7

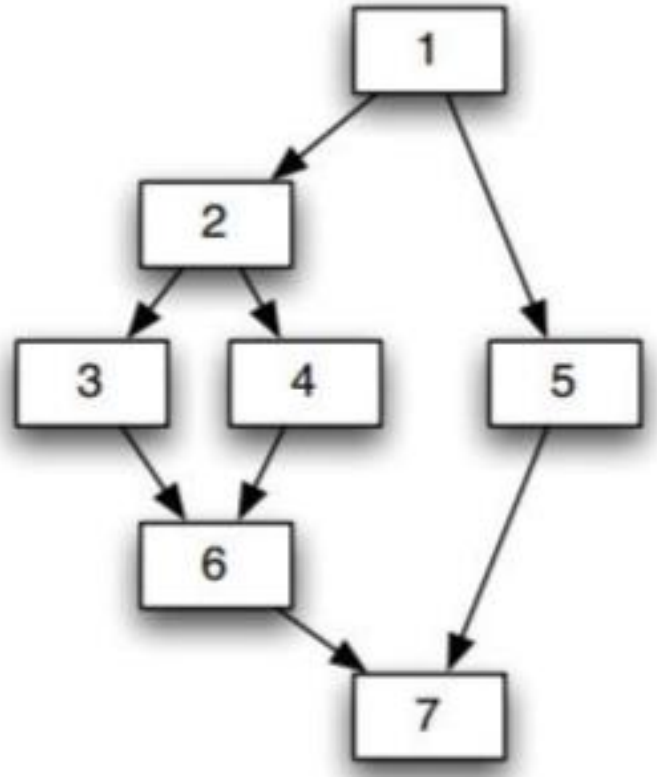
x = idom(6) = 2，更新DF[2] = 7

x = idom(2) = 1，严格支配

Dominators

- 如何得到支配边界?

```
//计算CFG中每个节点的支配边界
for (a, b) in CFG edges do:
        x <- a;
        while x dose not strictly dominate b do:
                DF(x) = (DF(x) ∪ b);
                x = immediate dominator(x);
```



	1	2	3	4	5	6	7
Sdom	NULL	1	1，2	1，2	1	1，2	1
支配边界	NULL	7	6	6	7	7	NULL

5->7 存在支配边界 DF[5] = 7
x = idom(5) = 1，严格支配

DF[7] = NULL

Dominators

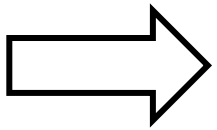
- 至此，已经得到了直接支配节点的集合(idom_)、函数支配树与支配边界
- 下面继续介绍Mem2Reg

Mem2Reg

- LightIR (LLVM) 引入了alloca指令来表示栈变量
- 前端可以简单地将每个局部变量映射到一个alloca指令，并通过load/store操作读写相应的内存变量，简化了前端设计难度

```
int main(void) {
    int a;
    a = 1 + 1;
    return a;
}
```

前端生成的IR代码



```
define i32 @main() {
label_entry:
    %op0 = alloca i32
    %op1 = add i32 1, 1
    store i32 %op1, i32* %op0
    %op2 = load i32, i32* %op0
    ret i32 %op2
}
```

Mem2Reg

- 该IR存在的问题
  - 不必要的alloca/load/store指令 -> 访存多，性能差
  - 不是严格的SSA形式 -> 不利于后续中间代码优化

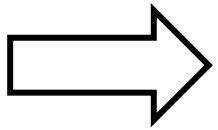
```
define i32 @main() {
label_entry:
    %op0 = alloca i32
    %op1 = add i32 1, 1
    store i32 %op1, i32* %op0
    %op2 = load i32, i32* %op0
    ret i32 %op2
}
```

Mem2Reg

- Mem2Reg：自动将内存变量提升为寄存器变量
  - 删除不必要的访存指令
  - 使得IR符合SSA形式

```
define i32 @main() {
label_entry:
    %op0 = alloca i32
    %op1 = add i32 1, 1
    store i32 %op1, i32* %op0
    %op2 = load i32, i32* %op0
    ret i32 %op2
}
```

Mem2Reg



```
define i32 @main() {
label_entry:
    %op1 = add i32 1, 1
    ret i32 %op1
}
```

Mem2Reg

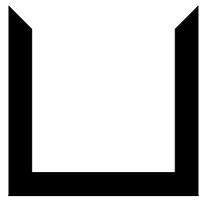
- **Mem2Reg基本思想：对内存变量使用基于栈的到达定义分析**
  - **按行扫描该基本块中的所有指令**
    - 对alloca指令，建立相关变量的栈；
    - 对store指令，将需store的值入栈；
    - 对load指令，使用栈顶元素替换目标寄存器的所有使用；

Mem2Reg

- Mem2Reg过程：对内存变量使用基于栈的到达定义分析
  - 按行扫描该基本块中的所有指令
    - 对alloca指令，建立相关变量的栈；

扫描到当前位置

```
define i32 @main() {
label_entry:
  %op0 = alloca i32
  %op1 = add i32 1, 2
  store i32 %op1, i32* %op0
  %op2 = load i32, i32* %op0
  %op3 = mul i32 %op2, 4
  store i32 %op3, i32* %op0
  ret i32 0
}
```



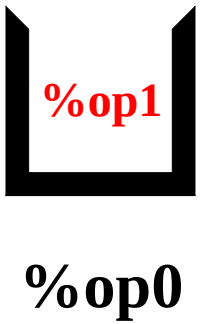
%op0

Mem2Reg

- Mem2Reg过程：对内存变量使用基于栈的到达定义分析
  - 按行扫描该基本块中的所有指令
    - 对store指令，将需store的值入栈；

扫描到当前位置

```
define i32 @main() {
label_entry:
  %op0 = alloca i32
  %op1 = add i32 1, 2
  store i32 %op1, i32* %op0
  %op2 = load i32, i32* %op0
  %op3 = mul i32 %op2, 4
  store i32 %op3, i32* %op0
  ret i32 0
}
```

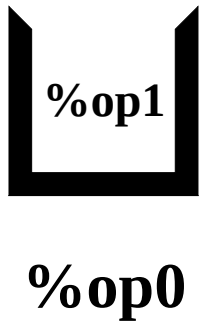


Mem2Reg

- Mem2Reg过程：对内存变量使用基于栈的到达定义分析
  - 按行扫描该基本块中的所有指令
    - 对load指令，使用栈顶元素替换目标寄存器的所有使用；

扫描到当前位置

```
define i32 @main() {
label_entry:
  %op0 = alloca i32
  %op1 = add i32 1, 2
  store i32 %op1, i32* %op0
  %op2 = load i32, i32* %op0
  %op3 = mul i32 %op1, 4
  store i32 %op3, i32* %op0
  ret i32 0
}
```

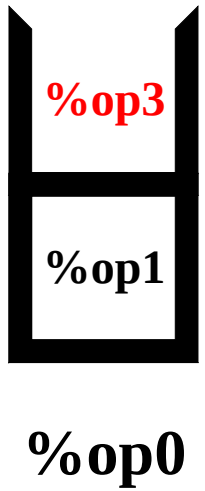


Mem2Reg

- Mem2Reg过程：对内存变量使用基于栈的到达定义分析
  - 按行扫描该基本块中的所有指令
    - 对store指令，将需store的值入栈；

扫描到当前位置

```
define i32 @main() {
label_entry:
  %op0 = alloca i32
  %op1 = add i32 1, 2
  store i32 %op1, i32* %op0
  %op2 = load i32, i32* %op0
  %op3 = mul i32 %op1, 4
  store i32 %op3, i32* %op0
  ret i32 0
}
```

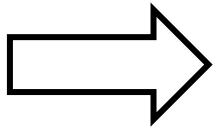


Mem2Reg

- Mem2Reg过程：对内存变量使用基于栈的到达定义分析
  - 按行扫描该基本块中的所有指令
  - 扫描结束后，删除所有alloca/load/store指令；

```
define i32 @main() {
label_entry:
  %op0 = alloca i32
  %op1 = add i32 1, 2
  store i32 %op1, i32* %op0
  %op2 = load i32, i32* %op0
  %op3 = mul i32 %op2, 4
  store i32 %op3, i32* %op0
  ret i32 0
}
```

删除已经处理完毕的内存操作

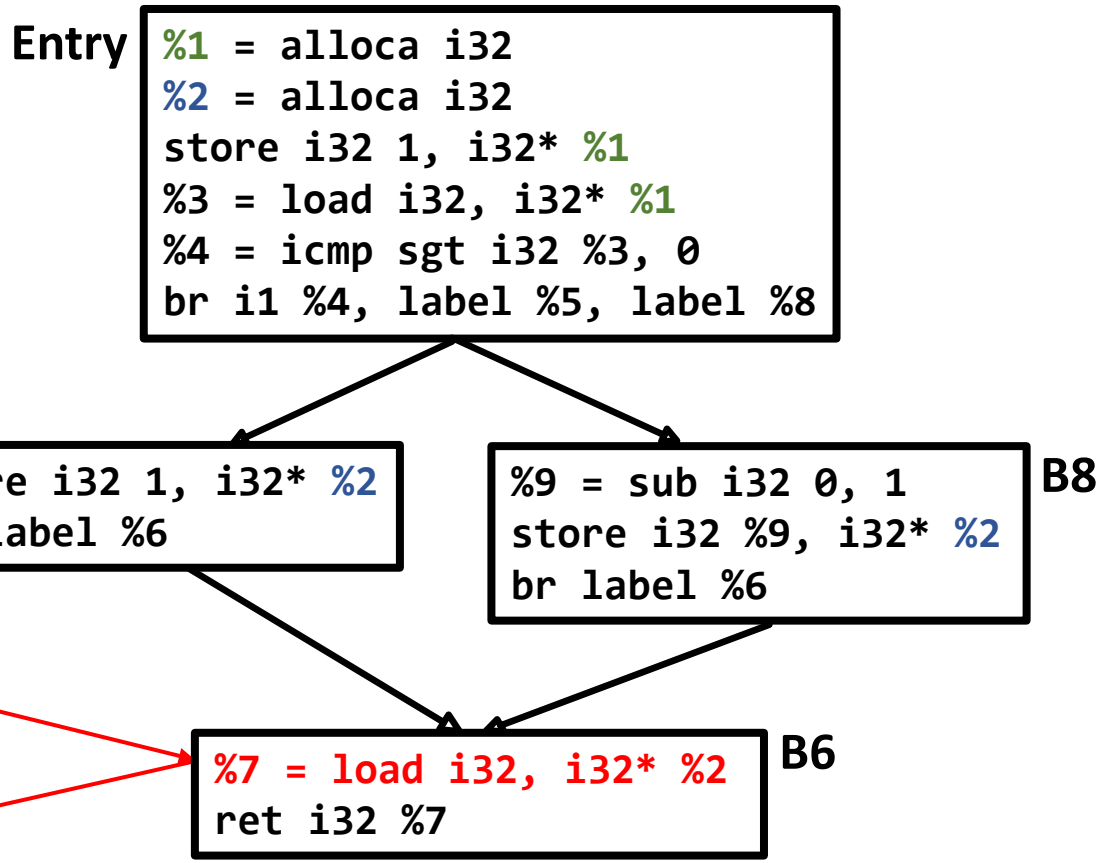


```
define i32 @main() {
label_entry:
    %op1 = add i32 1, 2
    %op3 = mul i32 %op1, 4
    ret i32 0
}
```

Mem2Reg

❑考虑更一般的IR情况

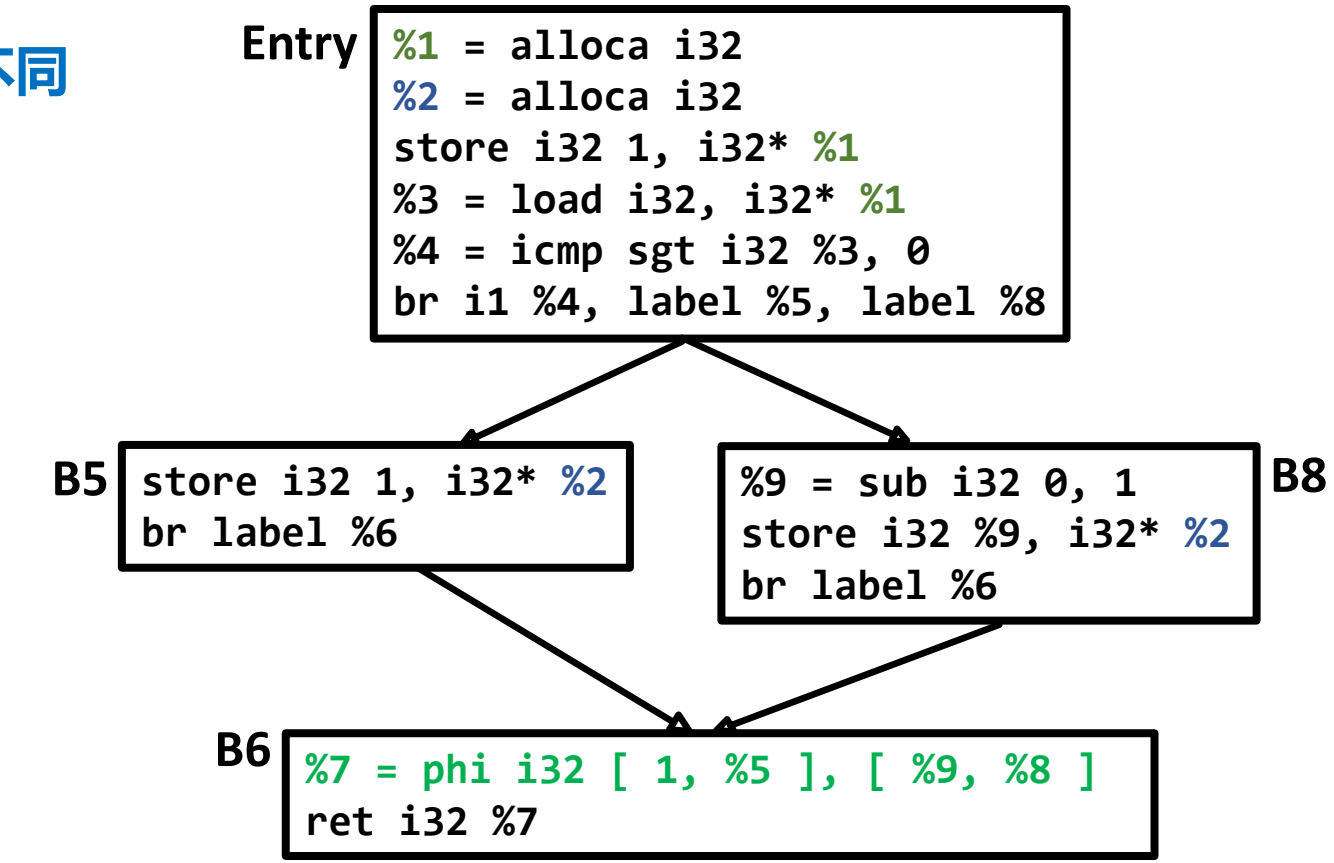
```
int main() {
    int cond;
    int x;
    cond = 1;
    if (cond > 0)
        x = 1;
    else
        x = -1;
    return x;
}
```



%7的值即可能取1，也可能取%9，此时有多个定义，基于栈的到达定义分析无法处理，怎么办?

Mem2Reg

- 引入phi函数
  - 在程序的控制流图中的分支节点处，合并不同路径上变量的定义。
  - 当从5号基本块进入6号基本块，%7取1；
  - 当从8号基本块进入6号基本块，%7取%9；



Mem2Reg

❑Mem2Reg的一般过程：

1. **插入phi函数 （基于支配树分析）**
   - 通过插入phi函数解决交汇块定义冲突问题
   - 仅插入phi函数，不填写相关参数
2. **变量重命名 （基于栈的到达定值分析）**
   - 到达定值分析，消除内存变量涉及的load指令依赖
   - 回填phi函数
3. **删除冗余指令**
   - 删除内存变量涉及的alloca/load/store指令

Mem2Reg

❑Mem2Reg的一般过程：

1. **插入phi函数 （基于支配边界分析）**
   - 通过插入phi函数解决交汇块定义冲突问题
   - 仅插入phi函数，不填写相关参数
2. **变量重命名 （基于栈的到达定义分析）**
   - 到达定值分析，消除内存变量涉及的load指令依赖
   - 回填phi函数
3. **删除冗余指令**
   - 删除内存变量涉及的alloca/load/store指令

Mem2Reg

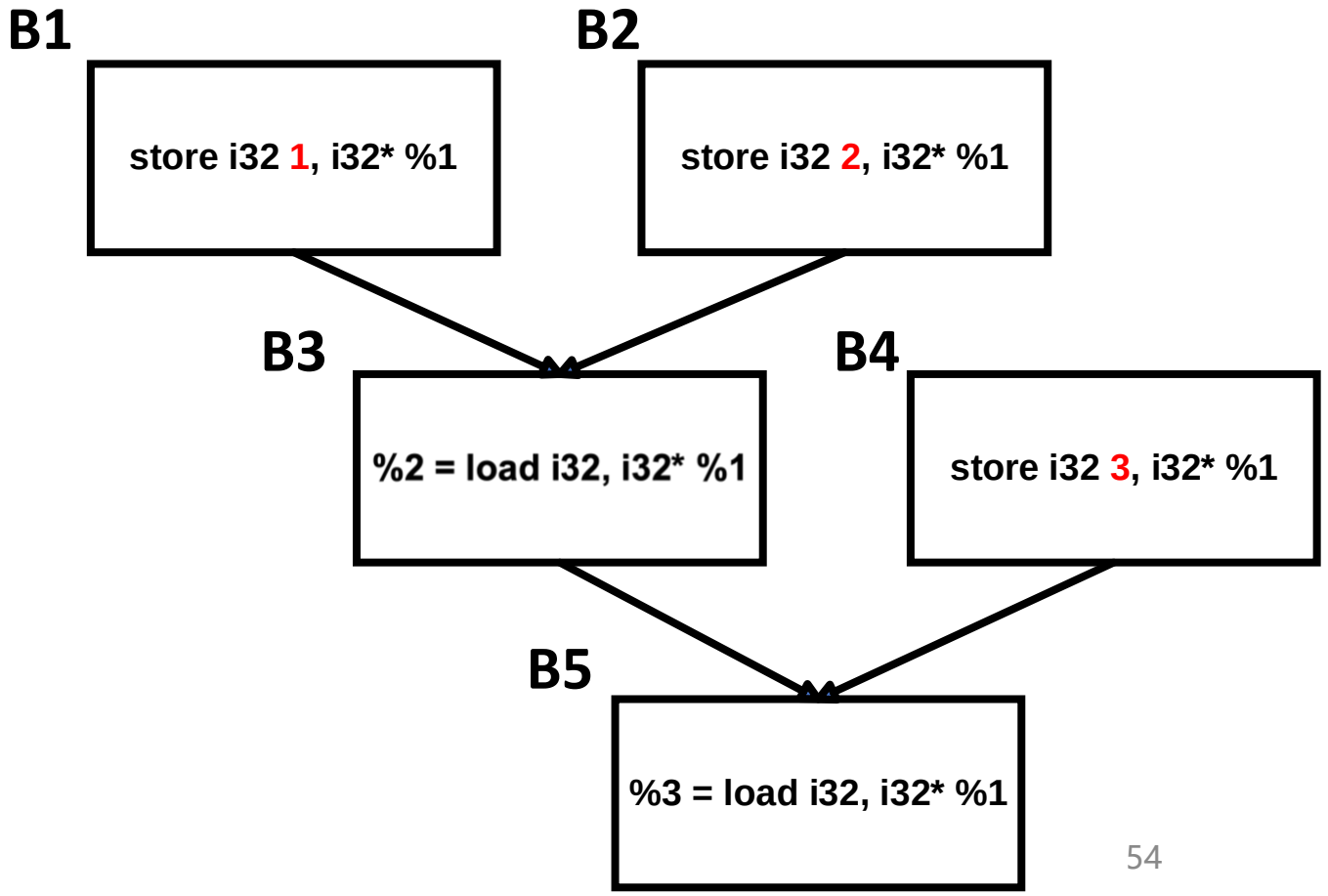
- 插入phi函数
  - 哪些内存变量需要插入?
  - 在哪插入?

Mem2Reg

- 插入phi函数
  - 哪些内存变量需要插入？ -> 跨多个基本块的内存变量
  - 在哪插入？

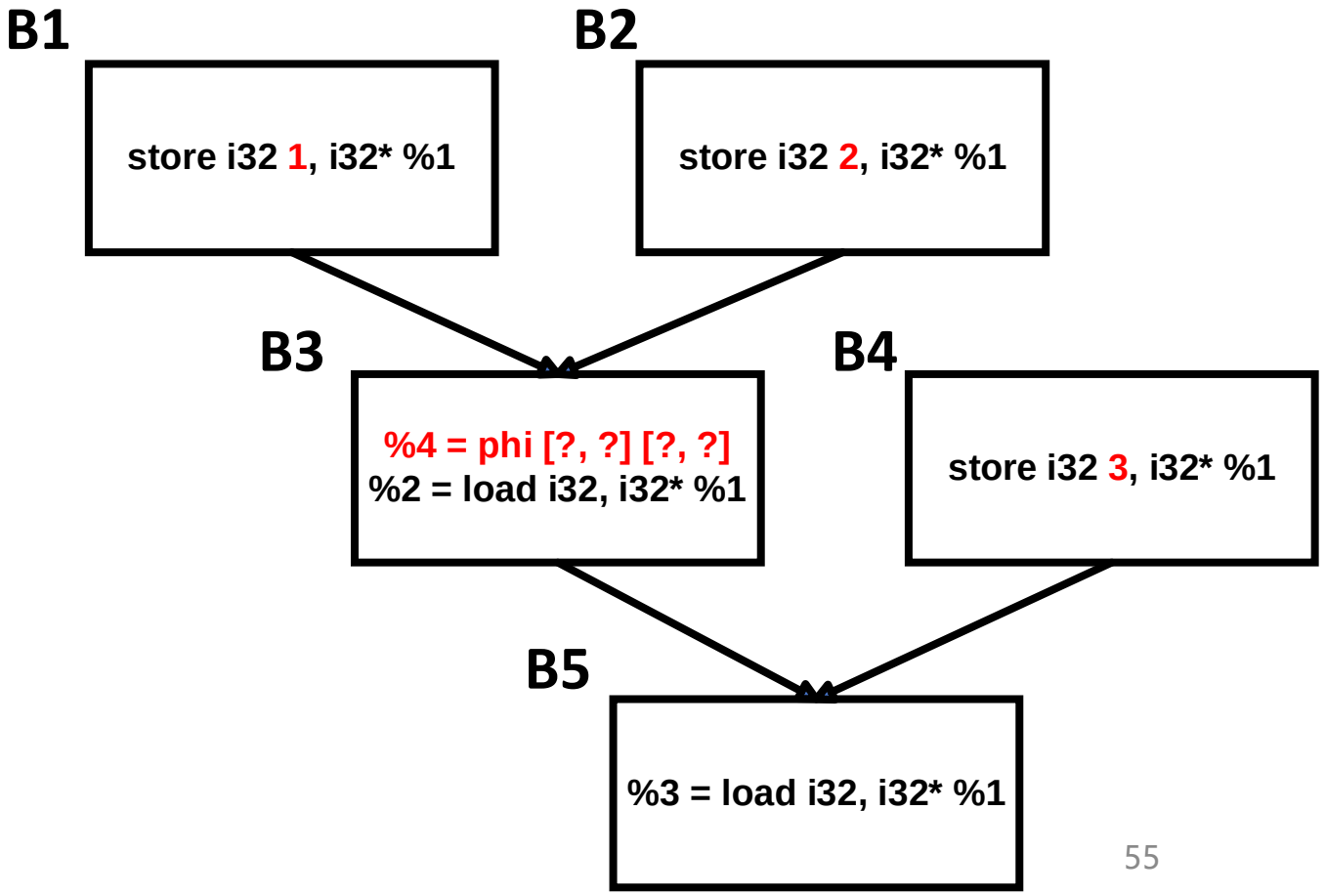
Mem2Reg

- 插入phi函数
  - 哪些内存变量需要插入？ -> 跨多个基本块的内存变量
  - 在哪插入？



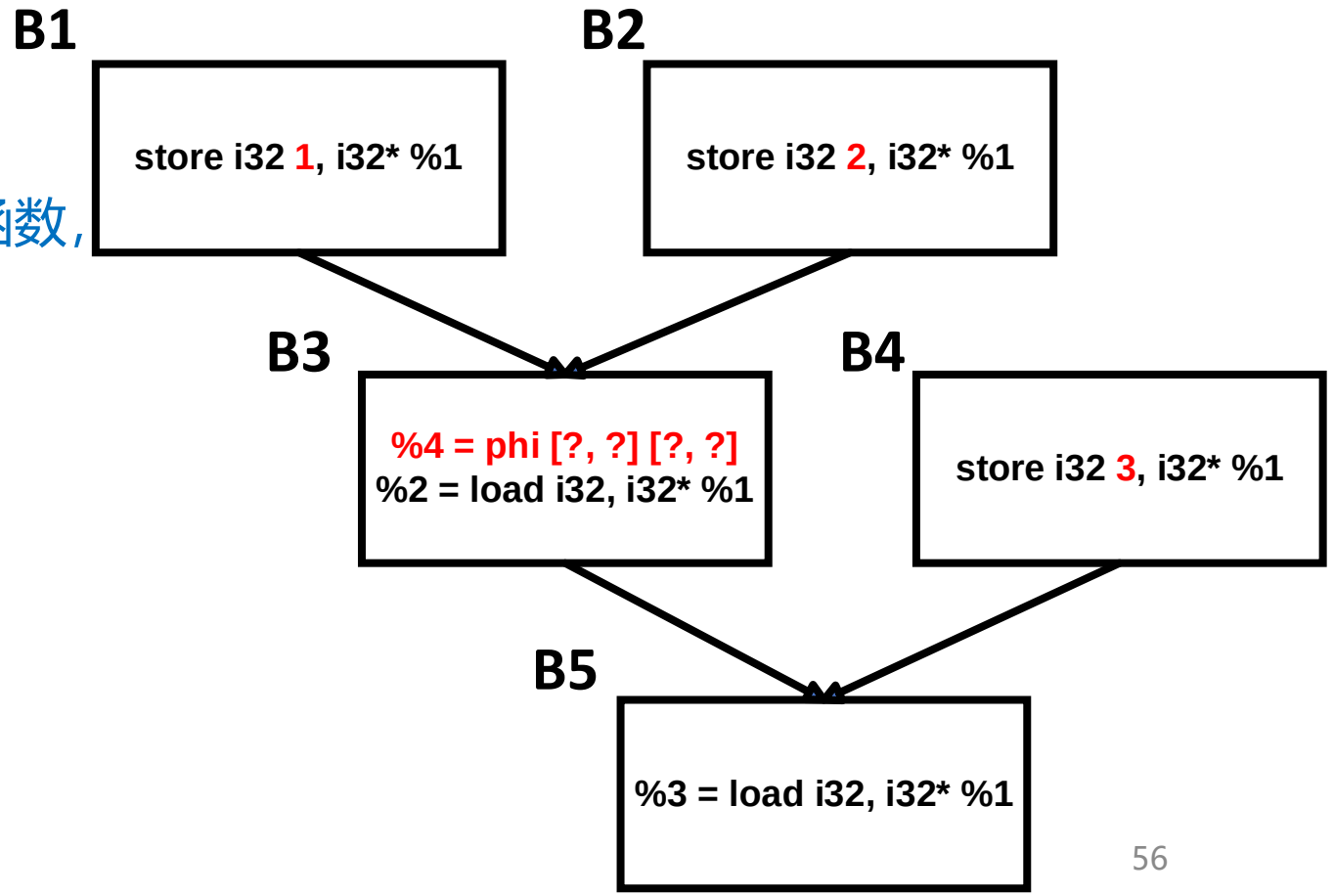
Mem2Reg

- 插入phi函数
  - 哪些内存变量需要插入？ -> 跨多个基本块的内存变量
  - 在哪插入？
    - 在B3中插入phi函数



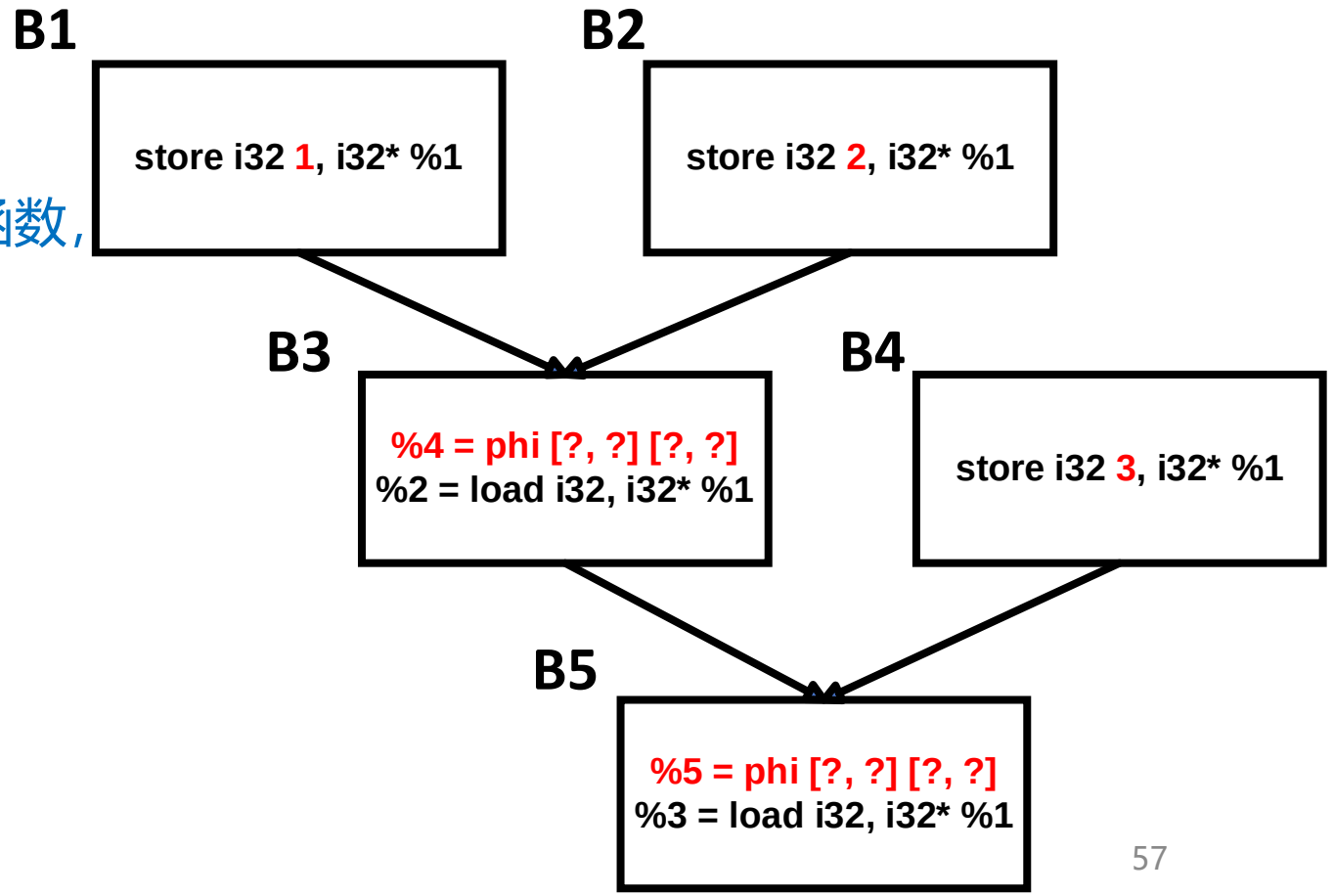
Mem2Reg

- 插入phi函数
  - 哪些内存变量需要插入？-> 跨多个基本块的内存变量
  - 在哪插入？
    - 在B3中插入phi函数
    - B3插入的phi函数本身也是对%1的定义，也要在其支配边界DF(B3) = {B5}插入phi函数，所以在B5中插入phi函数



Mem2Reg

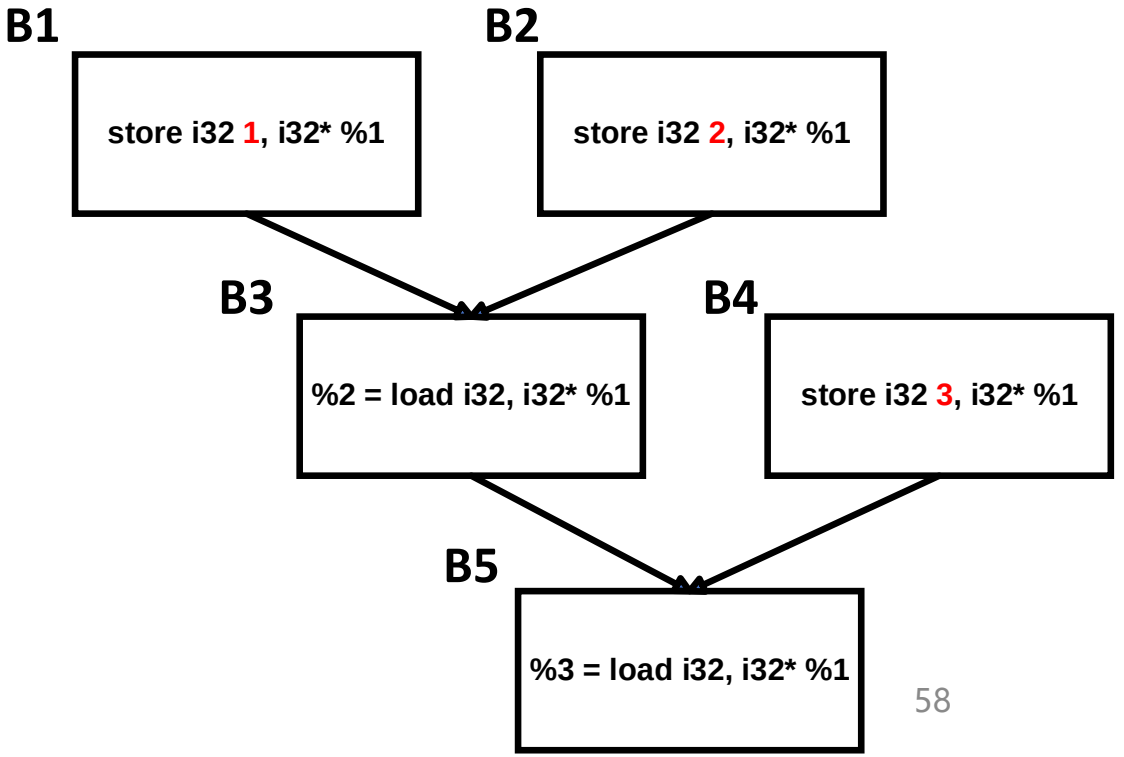
- 插入phi函数
  - 哪些内存变量需要插入？ -> 跨多个基本块的内存变量
  - 在哪插入？
    - 在B3中插入phi函数
    - B3插入的phi函数本身也是对%1的定义，也要在其支配边界DF(B3) = {B5}插入phi函数，所以在B5中插入phi函数



Mem2Reg

❑插入phi函数

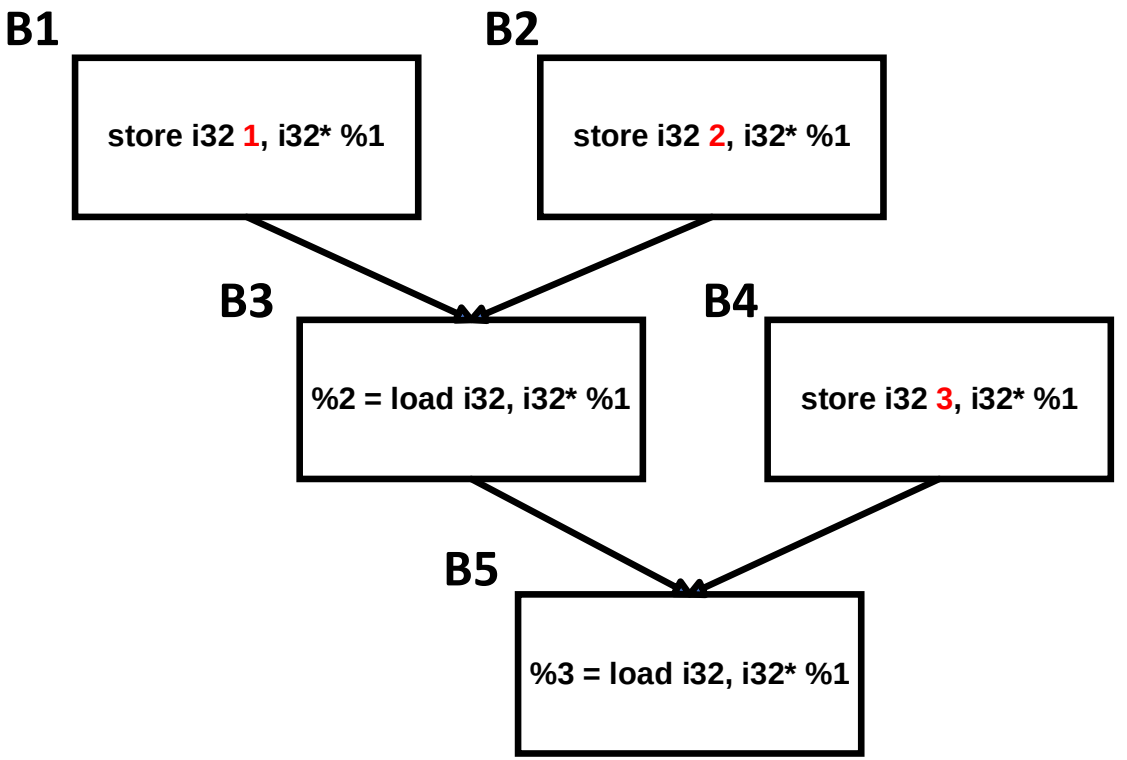
- **哪些内存变量需要插入？-> 跨多个基本块的内存变量**
- **在哪插入？ -> 被定义变量所在的支配边界**
  - 严格支配：如果 n 支配 x，且 $n \neq x$，则称 n 严格支配 x.
  - 支配边界 $DF(n) = \{x \mid n$支配x的前驱节点，n不严格支配$x\}$.



Mem2Reg

❑插入phi函数

- **哪些内存变量需要插入？-> 跨多个基本块的内存变量**
- **在哪插入？ -> 被定义变量所在的支配边界**
  - 支配边界 $DF(n) = \{x \mid n$支配x的前驱节点，n不严格支配$x\}$.
- **%1在B1, B2, B4中被定义，其支配边界为：**
  - **DF(B1) = {B3}**
  - **DF(B2) = {B3}**
  - **DF(B4) = {B5}**



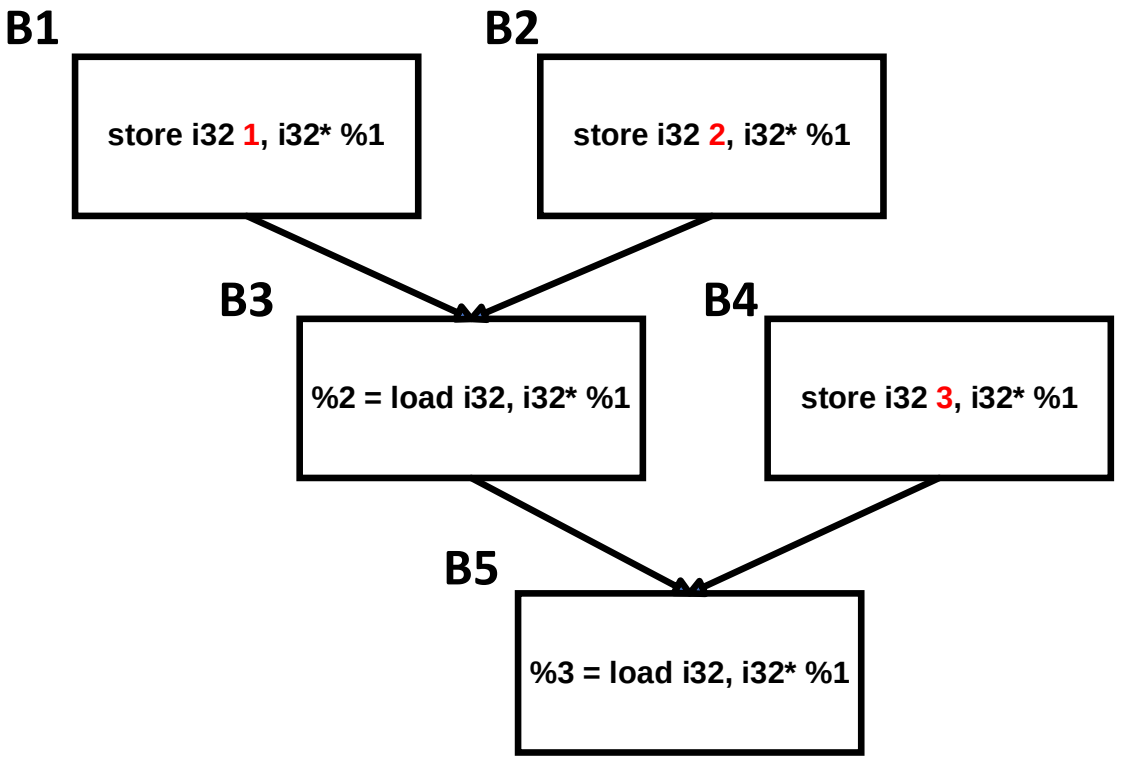
Mem2Reg

❑插入phi函数

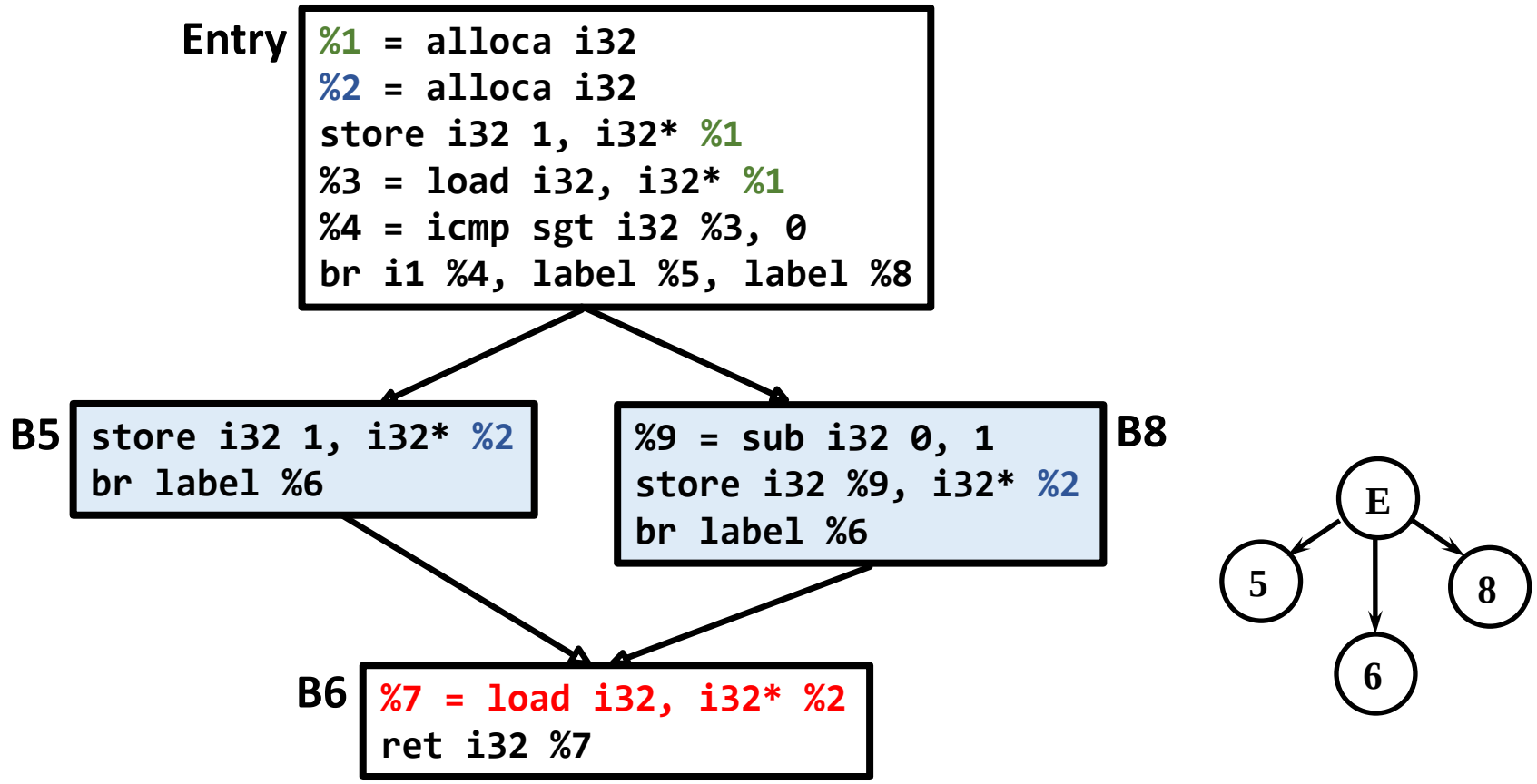
- ■ 哪些内存变量需要插入？-> 跨多个基本块的内存变量
- ■ 在哪插入？ -> 被定义变量所在的支配边界
  - ➢ 支配边界 $DF(n) = \{x \mid n$支配x的前驱节点，n不严格支配$x\}$.
- ■ %1在B1, B2, B4中被定义，其支配边界为：
  - ➢ DF(B1) = {B3}
  - ➢ DF(B2) = {B3}
  - ➢ DF(B4) = {B5}

在支配边界中，存在相关变量的多个定义

- ➢ 如B3: %2取1还是2
- ➢ 如B5: %3取1、2还是3

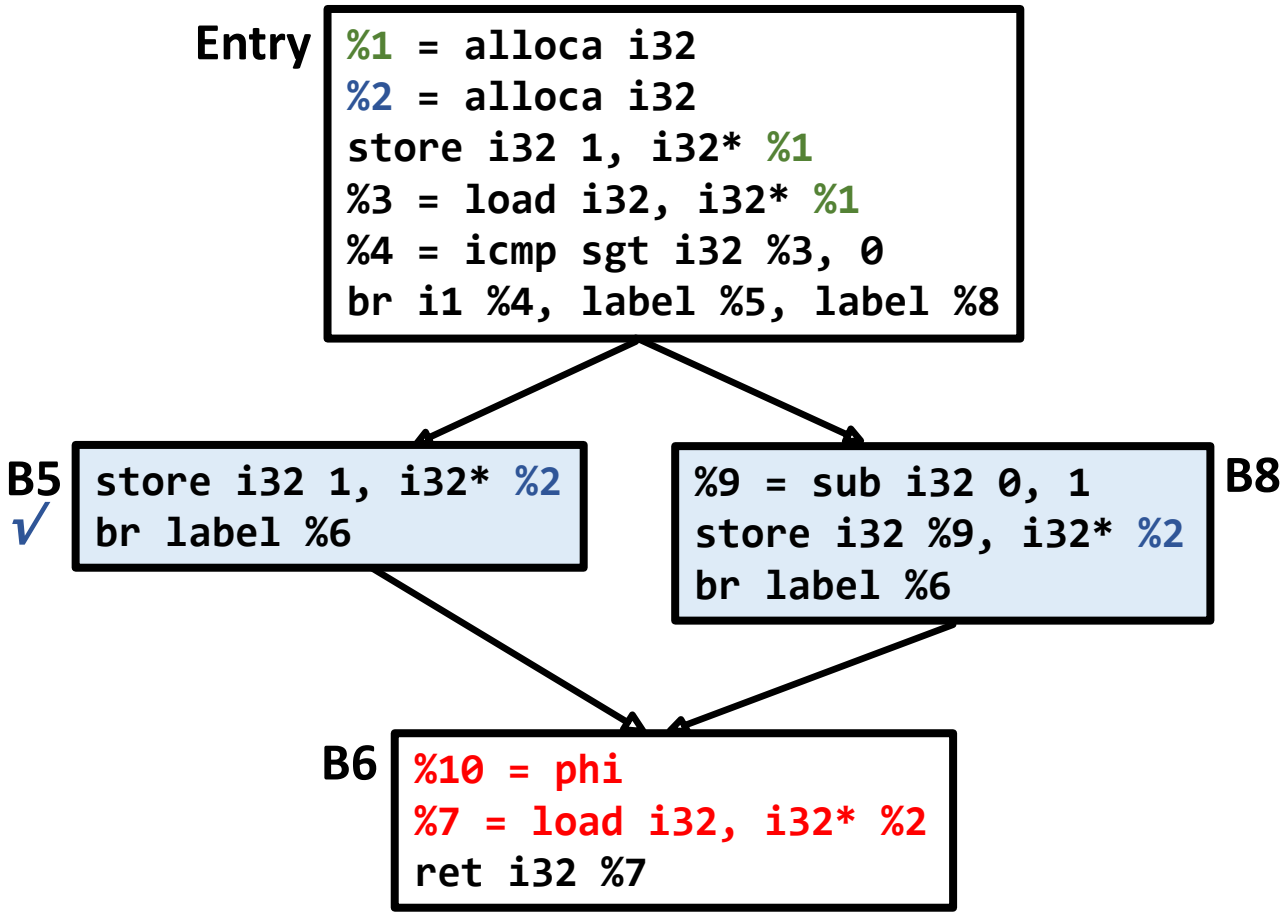


Mem2Reg



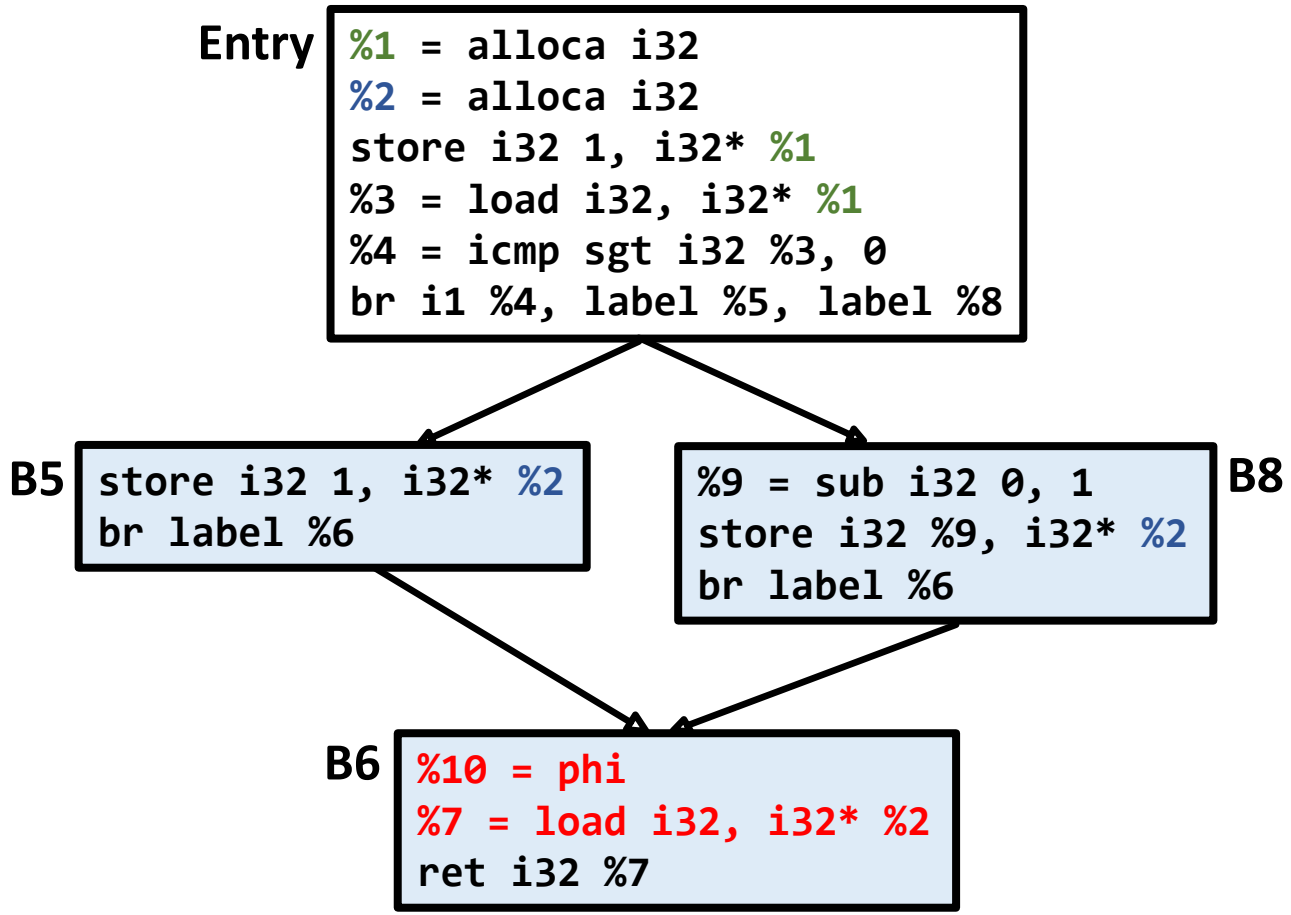
Step1 找到对%2进行定义的基本块

Mem2Reg



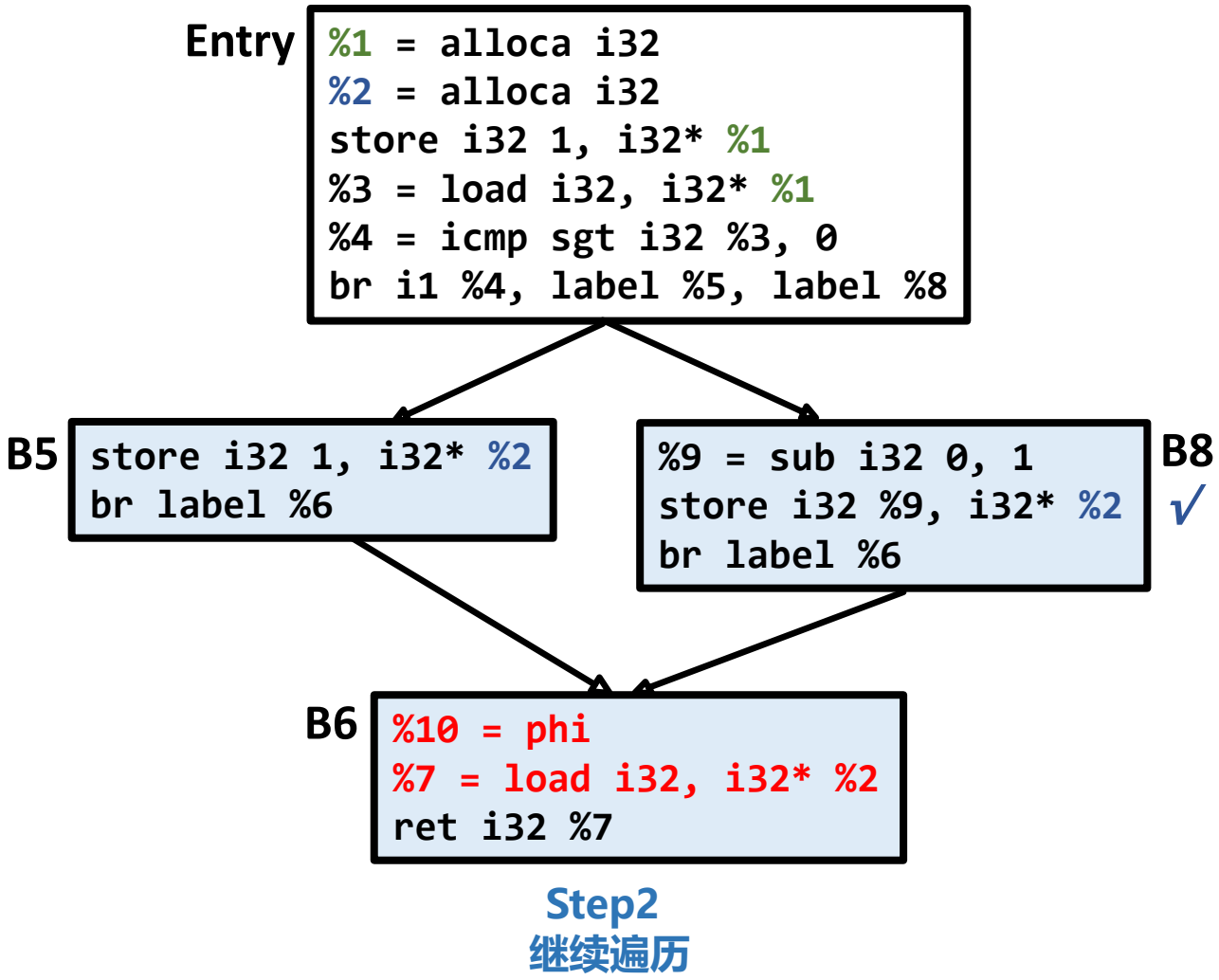
Step2
遍历每个定值基本基本块B，
在DF(B)插入phi函数

Mem2Reg

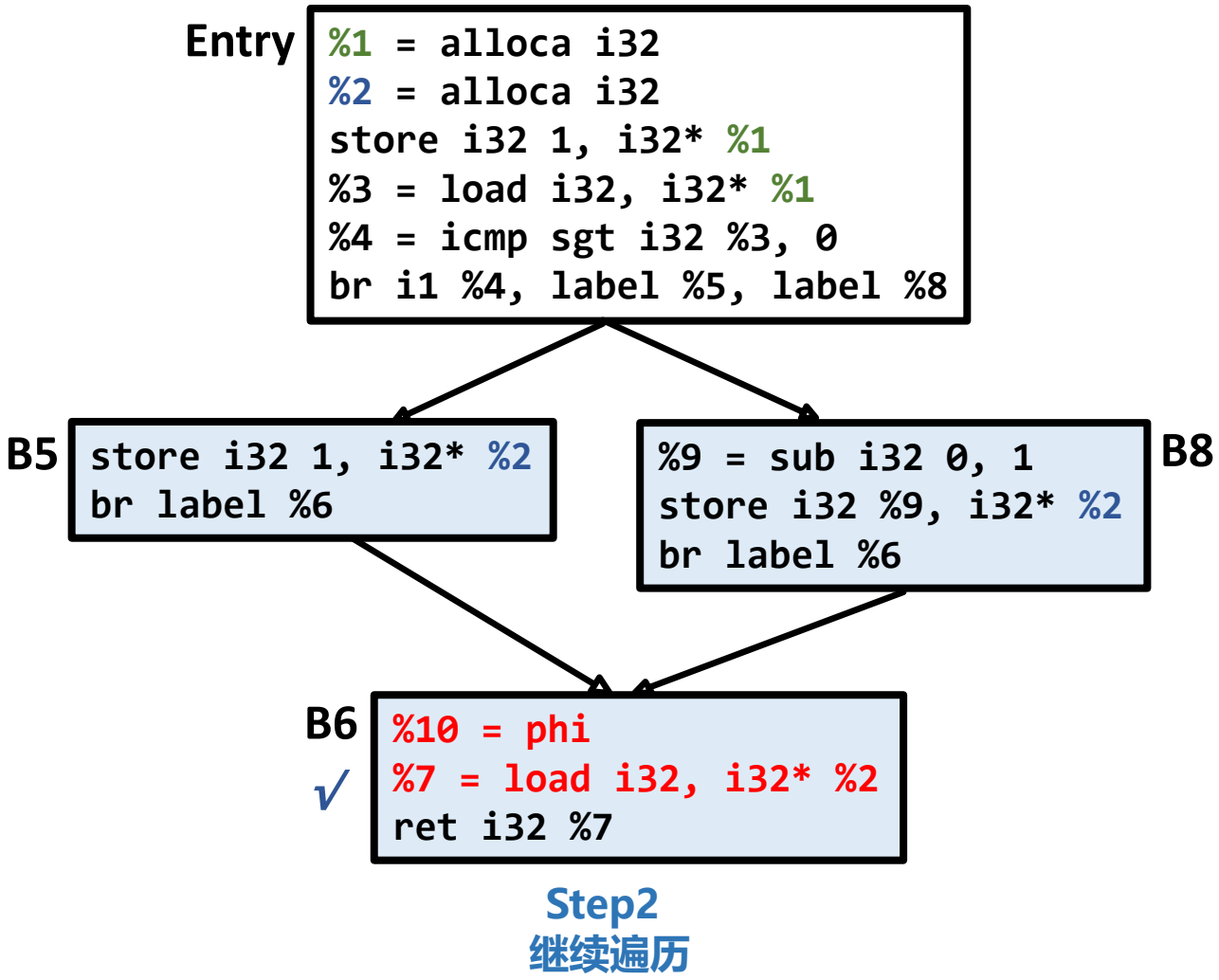


Step2
插入phi后，
DF(8)也是对%2的定值基本块

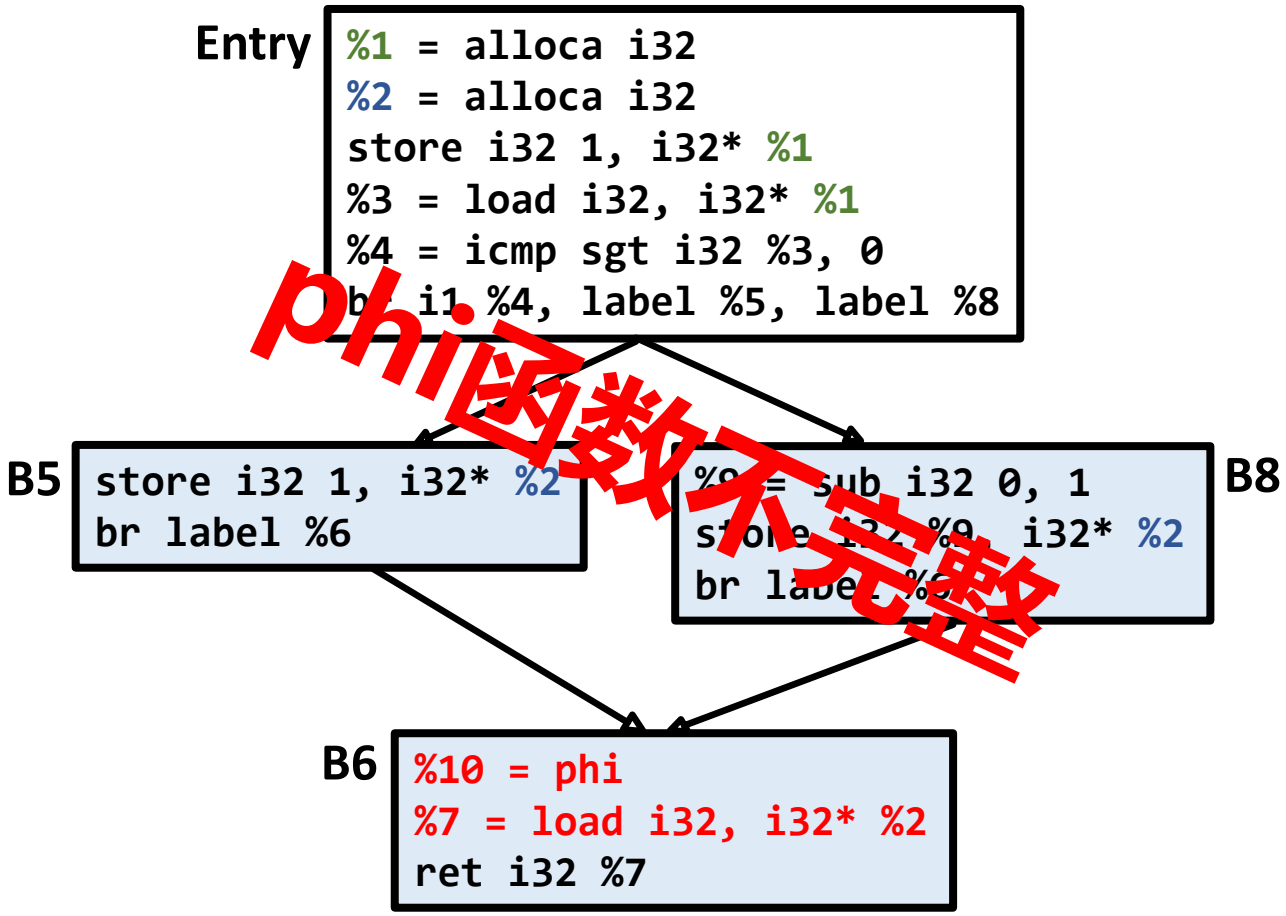
Mem2Reg


Entry
%1 = alloca i32
%2 = alloca i32
store i32 1, i32* %1
%3 = load i32, i32* %1
%4 = icmp sgt i32 %3, 0
br i1 %4, label %5, label %8
B5
store i32 1, i32* %2
br label %6
%9 = sub i32 0, 1
store i32 %9, i32* %2
br label %6
B8
√
B6
%10 = phi
%7 = load i32, i32* %2
ret i32 %7
Step2
继续遍历

Mem2Reg


Entry
%1 = alloca i32
%2 = alloca i32
store i32 1, i32* %1
%3 = load i32, i32* %1
%4 = icmp sgt i32 %3, 0
br i1 %4, label %5, label %8
B5
store i32 1, i32* %2
br label %6
%9 = sub i32 0, 1
store i32 %9, i32* %2
br label %6
B8
B6
%10 = phi
%7 = load i32, i32* %2
ret i32 %7
Step2
继续遍历

Mem2Reg



此时phi函数参数是无法填写的
因为编译器缺少
相关到达定义信息

Mem2Reg

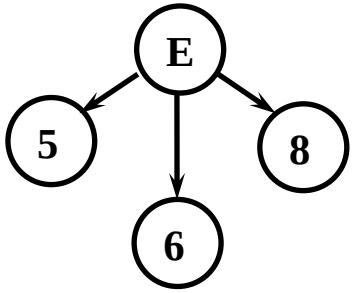
❑Mem2Reg的一般过程：

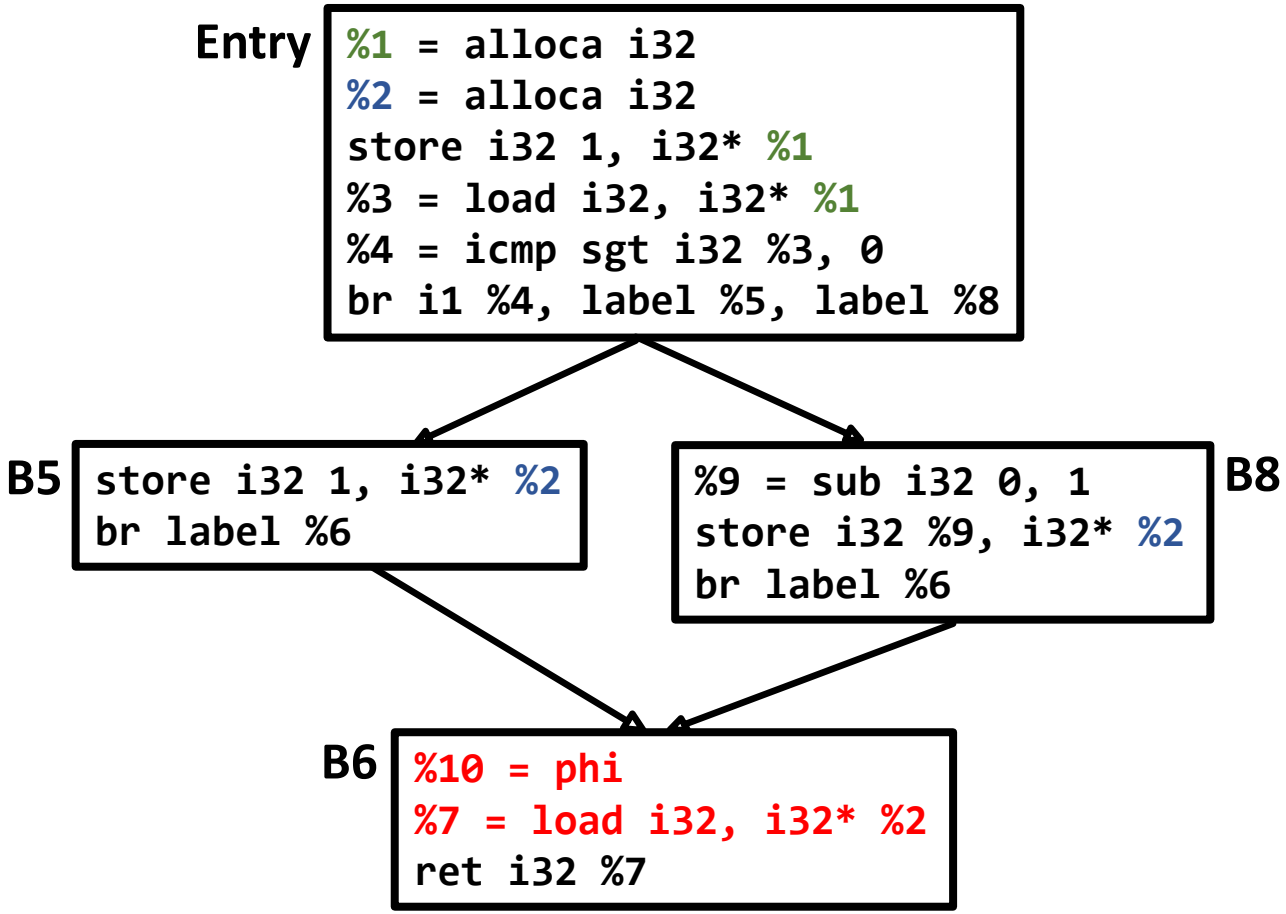
1. **插入phi函数（基于控制流分析）**
   - 通过插入phi函数解决交汇块定义冲突问题
   - 仅插入phi函数，不填写相关参数
2. **变量重命名（基于栈的到达定值分析）**
   - 到达定值分析，消除内存变量涉及的load指令依赖
   - 回填phi函数
3. **删除冗余指令**
   - 删除内存变量涉及的alloca/load/store指令

Mem2Reg

- **变量重命名**
  - **使用基于栈的到达定义分析（基于DFS遍历每条指令）**
    - 对alloca指令，建立相关变量的栈；
    - 对store指令，将需store的值入栈；
    - 对load指令，使用栈顶元素替换目标寄存器的所有使用；
    - **对于phi函数，将phi函数的寄存器入栈；**
  - **回填phi函数参数**

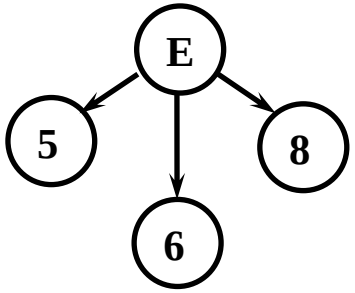
Mem2Reg





Step3 按照支配树的DFS顺序遍历
进行到达定值分析
Entry->B5->B6->B8

Mem2Reg



Entry ✓

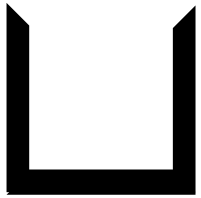
```
%1 = alloca i32
%2 = alloca i32
store i32 1, i32* %1
%3 = load i32, i32* %1
%4 = icmp sgt i32 %3, 0
br i1 %4, label %5, label %8
```

B5

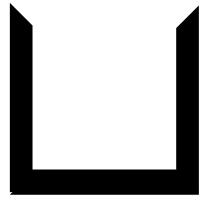
```
store i32 1, i32* %2
br label %6
```

B8

```
%9 = sub i32 0, 1
store i32 %9, i32* %2
br label %6
```



%1



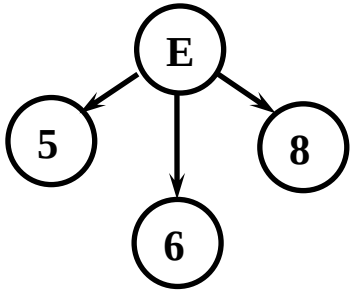
%2

B6

```
%10 = phi
%7 = load i32, i32* %2
ret i32 %7
```

Step3 按照支配树的DFS顺序遍历
进行到达定值分析
alloca指令：建立分析栈

Mem2Reg



Entry ✓

```
%1 = alloca i32
%2 = alloca i32
store i32 1, i32* %1
%3 = load i32, i32* %1
%4 = icmp sgt i32 %3, 0
br i1 %4, label %5, label %8
```

B5

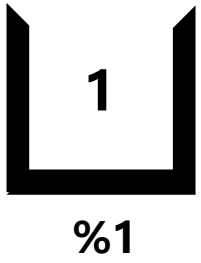
```
store i32 1, i32* %2
br label %6
```

```
%9 = sub i32 0, 1
store i32 %9, i32* %2
br label %6
```

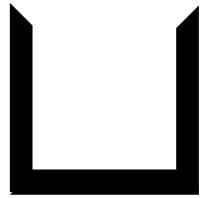
B8

B6

```
%10 = phi
%7 = load i32, i32* %2
ret i32 %7
```



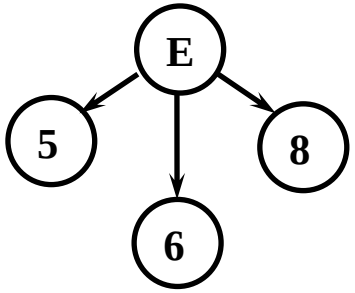
%1



%2

Step3 按照支配树的DFS顺序遍历进行到达定值分析
store指令：push最新定值

Mem2Reg



Entry ✓

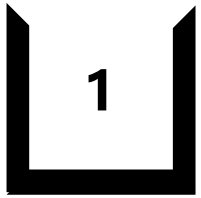
```
%1 = alloca i32
%2 = alloca i32
store i32 1, i32* %1
%3 = load i32, i32* %1
%4 = icmp sgt i32 1, 0
br i1 %4, label %5, label %8
```

B5

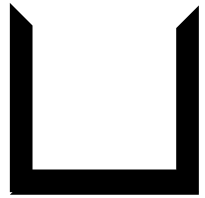
```
store i32 1, i32* %2
br label %6
```

B8

```
%9 = sub i32 0, 1
store i32 %9, i32* %2
br label %6
```



%1



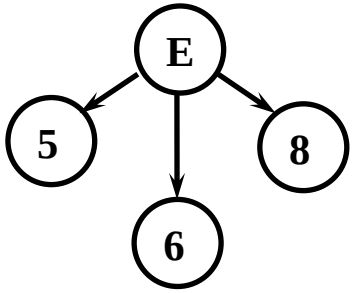
%2

B6

```
%10 = phi
%7 = load i32, i32* %2
ret i32 %7
```

Step3 按照支配树的DFS顺序遍历
进行到达定值分析
load指令：替换成栈顶定值

Mem2Reg



Entry

```
%1 = alloca i32
%2 = alloca i32
store i32 1, i32* %1
%3 = load i32, i32* %1
%4 = icmp sgt i32 1, 0
br i1 %4, label %5, label %8
```

B5 ✓

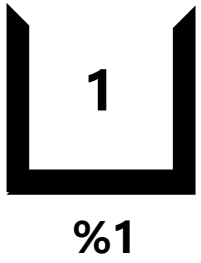
```
store i32 1, i32* %2
br label %6
```

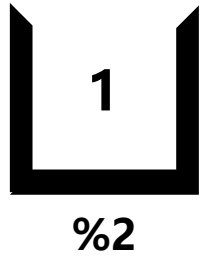
```
%9 = sub i32 0, 1
store i32 %9, i32* %2
br label %6
```

B8

B6

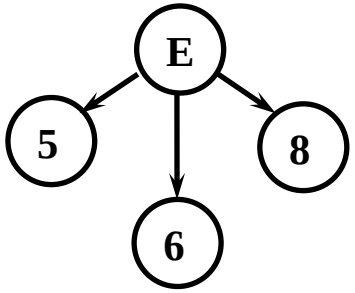
```
%10 = phi
%7 = load i32, i32* %2
ret i32 %7
```





Step3 按照支配树的DFS顺序遍历
进行到达定值分析
store指令：push最新定值

Mem2Reg



Entry

```
%1 = alloca i32
%2 = alloca i32
store i32 1, i32* %1
%3 = load i32, i32* %1
%4 = icmp sgt i32 1, 0
br i1 %4, label %5, label %8
```

B5 ✓

```
store i32 1, i32* %2
br label %6
```

```
%9 = sub i32 0, 1
store i32 %9, i32* %2
br label %6
```

B8

1

%1

1

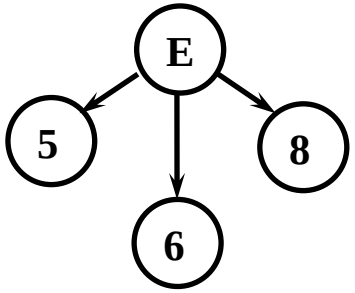
%2

B6

```
%10 = phi [1,%5]
%7 = load i32, i32* %2
ret i32 %7
```

Step4 补全后继BB的phi参数
来自B5的定值

Mem2Reg



Entry

```
%1 = alloca i32
%2 = alloca i32
store i32 1, i32* %1
%3 = load i32, i32* %1
%4 = icmp sgt i32 1, 0
br i1 %4, label %5, label %8
```

B5 ✓

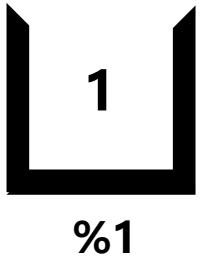
```
store i32 1, i32* %2
br label %6
```

```
%9 = sub i32 0, 1
store i32 %9, i32* %2
br label %6
```

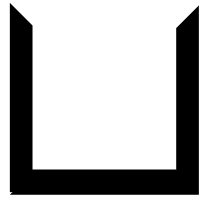
B8

B6

```
%10 = phi [1,%5]
%7 = load i32, i32* %2
ret i32 %7
```



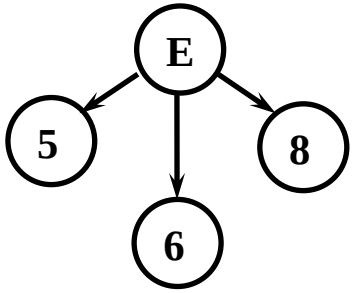
%1



%2

Step5 弹出栈中相关定值
支配树上B5无后继

Mem2Reg



Entry

```
%1 = alloca i32
%2 = alloca i32
store i32 1, i32* %1
%3 = load i32, i32* %1
%4 = icmp sgt i32 1, 0
br i1 %4, label %5, label %8
```

B5

```
store i32 1, i32* %2
br label %6
```

```
%9 = sub i32 0, 1
store i32 %9, i32* %2
br label %6
```

B8

1

%1

%10

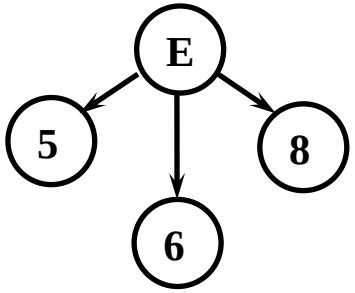
%2

B6 ✓

```
%10 = phi [1,%5]
%7 = load i32, i32* %2
ret i32 %7
```

Step3 按照支配树的DFS顺序遍历
进行到达定值分析
phi指令：push最新定值

Mem2Reg



Entry

```
%1 = alloca i32
%2 = alloca i32
store i32 1, i32* %1
%3 = load i32, i32* %1
%4 = icmp sgt i32 1, 0
br i1 %4, label %5, label %8
```

B5

```
store i32 1, i32* %2
br label %6
```

```
%9 = sub i32 0, 1
store i32 %9, i32* %2
br label %6
```

B8

1

%1

%10

%2

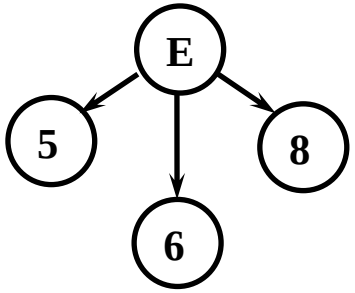
B6

✓

```
%10 = phi [1,%5]
%7 = load i32, i32* %2
ret i32 %10
```

Step3 按照支配树的DFS顺序遍历
进行到达定值分析
load指令：替换成栈顶定值

Mem2Reg



Entry

```
%1 = alloca i32
%2 = alloca i32
store i32 1, i32* %1
%3 = load i32, i32* %1
%4 = icmp sgt i32 1, 0
br i1 %4, label %5, label %8
```

✓

B5

```
store i32 1, i32* %2
br label %6
```

```
%9 = sub i32 0, 1
store i32 %9, i32* %2
br label %6
```

B8

1

%1

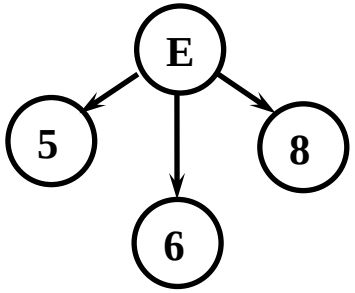
%2

B6

```
%10 = phi [1,%5]
%7 = load i32, i32* %2
ret i32 %10
```

Step5 弹出栈中相关定值
支配树上B6无后继

Mem2Reg



Entry

```
%1 = alloca i32
%2 = alloca i32
store i32 1, i32* %1
%3 = load i32, i32* %1
%4 = icmp sgt i32 1, 0
br i1 %4, label %5, label %8
```

B5

```
store i32 1, i32* %2
br label %6
```

```
%9 = sub i32 0, 1
store i32 %9, i32* %2
br label %6
```

B8 ✓

B6

```
%10 = phi [1,%5]
%7 = load i32, i32* %2
ret i32 %10
```

1

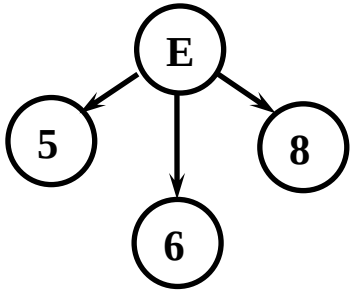
%1

%9

%2

Step3 按照支配树的DFS顺序遍历
进行到达定值分析
store指令：push最新定值

Mem2Reg



Entry

```
%1 = alloca i32
%2 = alloca i32
store i32 1, i32* %1
%3 = load i32, i32* %1
%4 = icmp sgt i32 1, 0
br i1 %4, label %5, label %8
```

B5

```
store i32 1, i32* %2
br label %6
```

B8 ✓

```
%9 = sub i32 0, 1
store i32 %9, i32* %2
br label %6
```

1

%1

%9

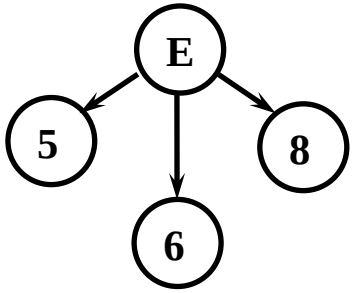
%2

B6

```
%10 = phi [1,%5] [%9,%8]
%7 = load i32, i32* %2
ret i32 %10
```

Step4 补全后继BB的phi参数
来自B8的定值

Mem2Reg



Entry

```
%1 = alloca i32
%2 = alloca i32
store i32 1, i32* %1
%3 = load i32, i32* %1
%4 = icmp sgt i32 1, 0
br i1 %4, label %5, label %8
```

B5

```
store i32 1, i32* %2
br label %6
```

```
%9 = sub i32 0, 1
store i32 %9, i32* %2
br label %6
```

B8 ✓

1

%1

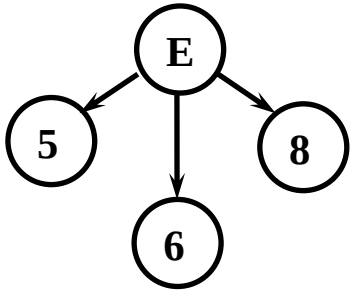
%2

B6

```
%10 = phi [1,%5] [%9,%8]
%7 = load i32, i32* %2
ret i32 %10
```

Step5 弹出栈中相关定值
支配树上B8无后继

Mem2Reg



Entry ✓

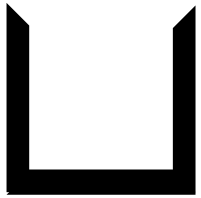
```
%1 = alloca i32
%2 = alloca i32
store i32 1, i32* %1
%3 = load i32, i32* %1
%4 = icmp sgt i32 1, 0
br i1 %4, label %5, label %8
```

B5

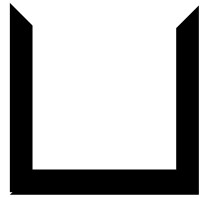
```
store i32 1, i32* %2
br label %6
```

B8

```
%9 = sub i32 0, 1
store i32 %9, i32* %2
br label %6
```



%1



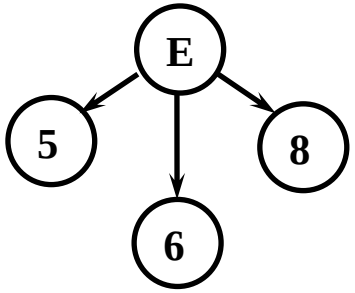
%2

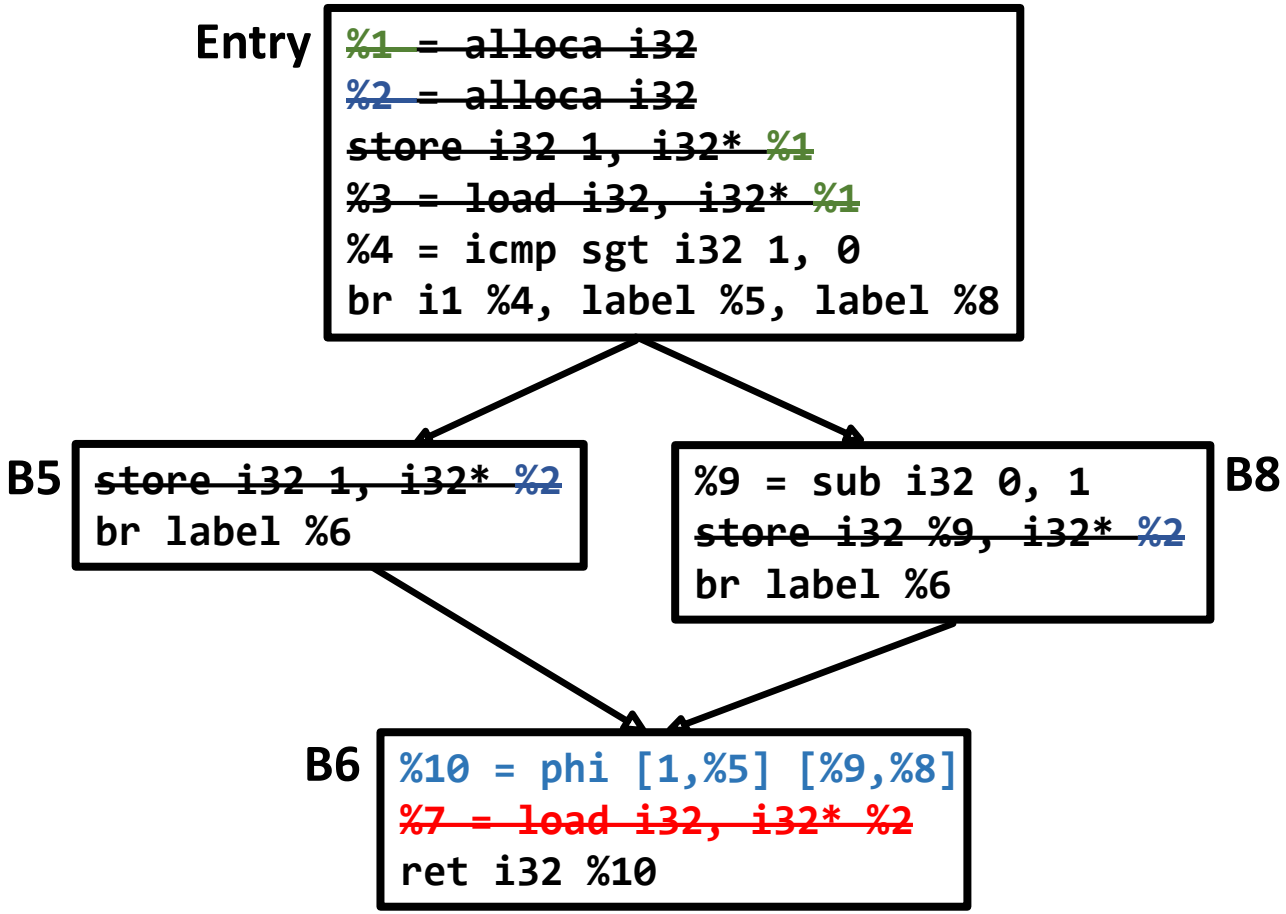
B6

```
%10 = phi [1,%5] [%9,%8]
%7 = load i32, i32* %2
ret i32 %10
```

Step5 弹出栈中相关定值
支配树上Entry的后继已经遍历完

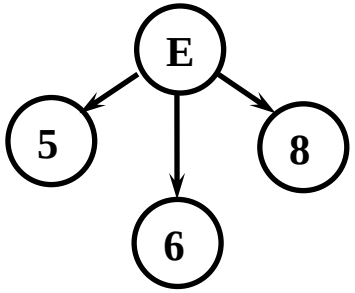
Mem2Reg

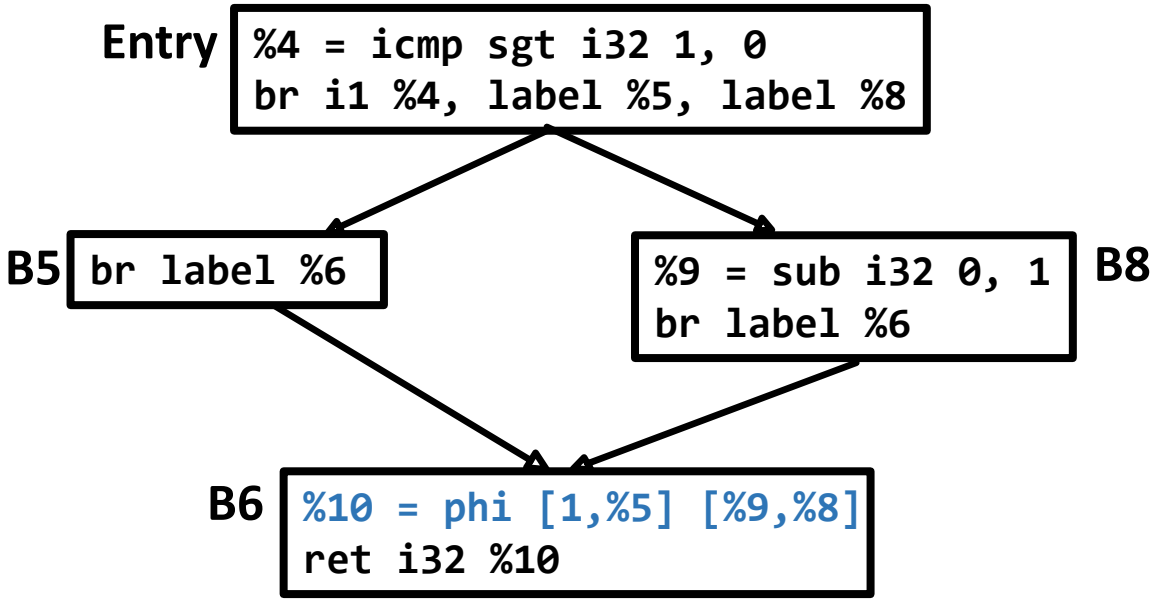




Step6 删除各块中的冗余指令
alloca/load/store

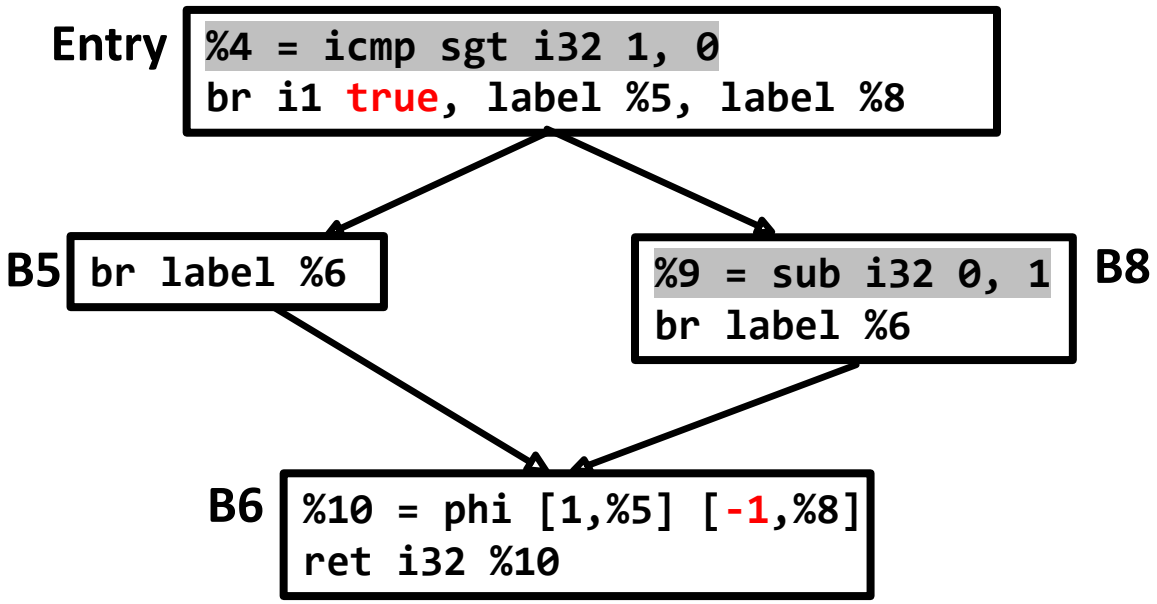
Mem2Reg





最终结果！！

What is more



常量传播

What is more

```
ret i32 1
```

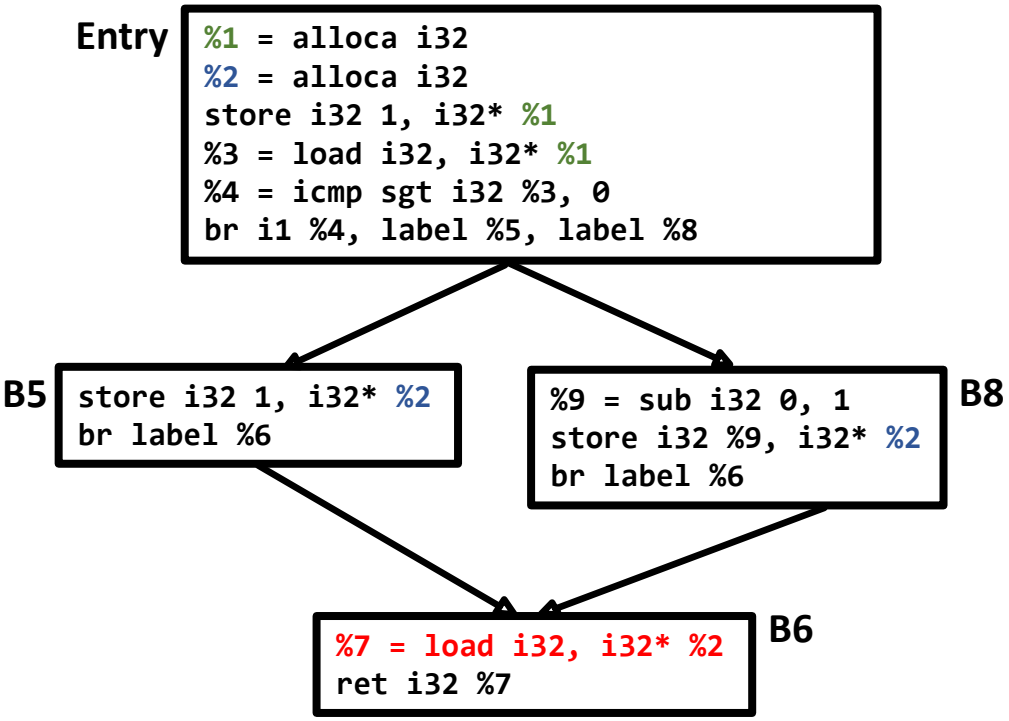
死代码删除+控制流化简

What is more

```
int main() {
    int cond;
    int x;
    cond = 1;
    if (cond > 0)
        x = 1;
    else
        x = -1;
    return x;
}
```

cminus 源程序

前端翻译



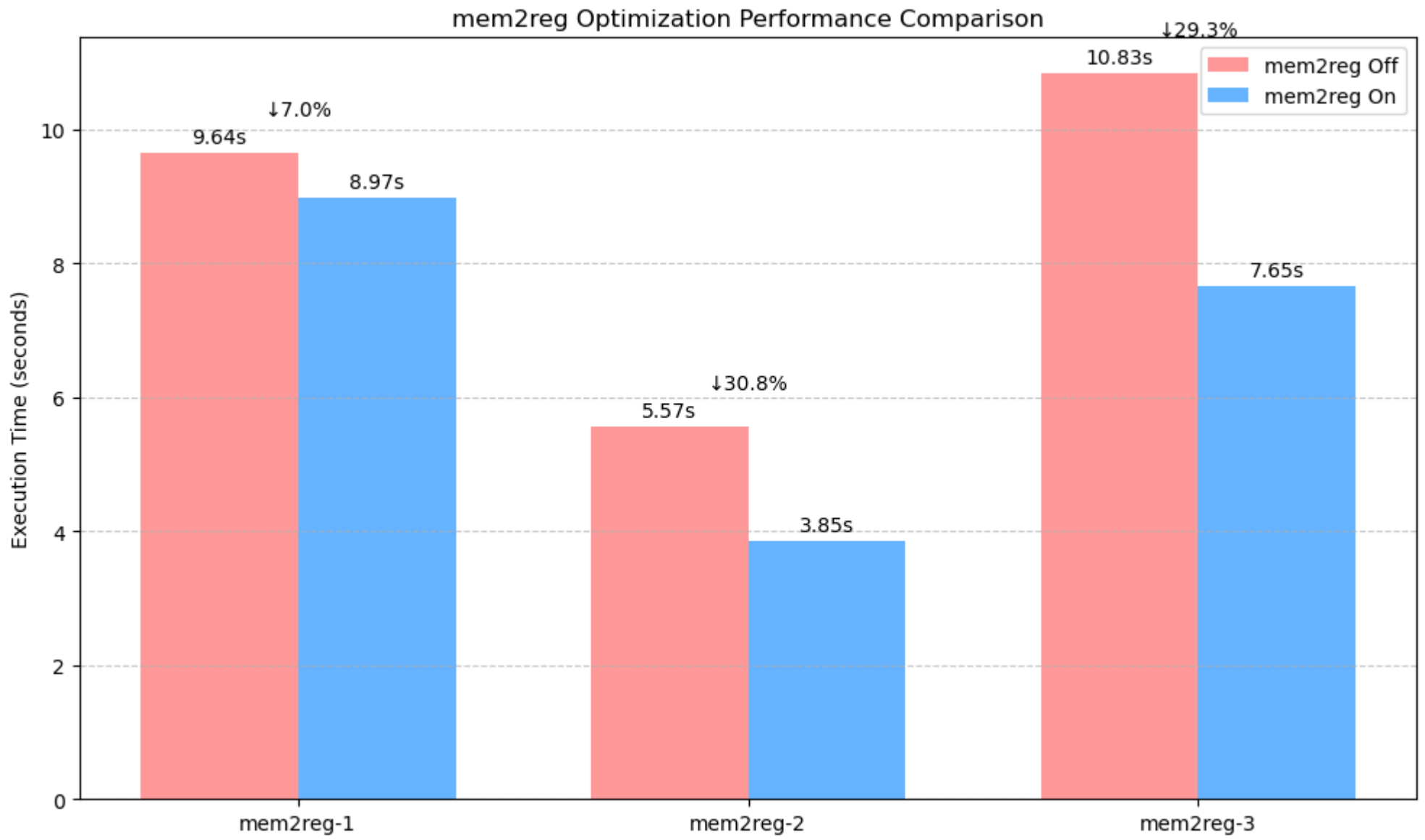
初步翻译得到的IR

优化pass

```
ret i32 1
```

完全优化后的IR

What is more


mem2reg Optimization Performance Comparison
Execution Time (seconds)
mem2reg Off
mem2reg On
9.64s
8.97s
↓7.0%
5.57s
3.85s
↓30.8%
10.83s
7.65s
↓29.3%
mem2reg-1
mem2reg-2
mem2reg-3
0
2
4
6
8
10

一起努力
打造国产基础软硬件体系！

徐 伟

国家高性能计算中心(合肥)、信息与计算机国家级实验教学示范中心

计算机科学与技术学院

2025年10月30日