

## Lecture 10

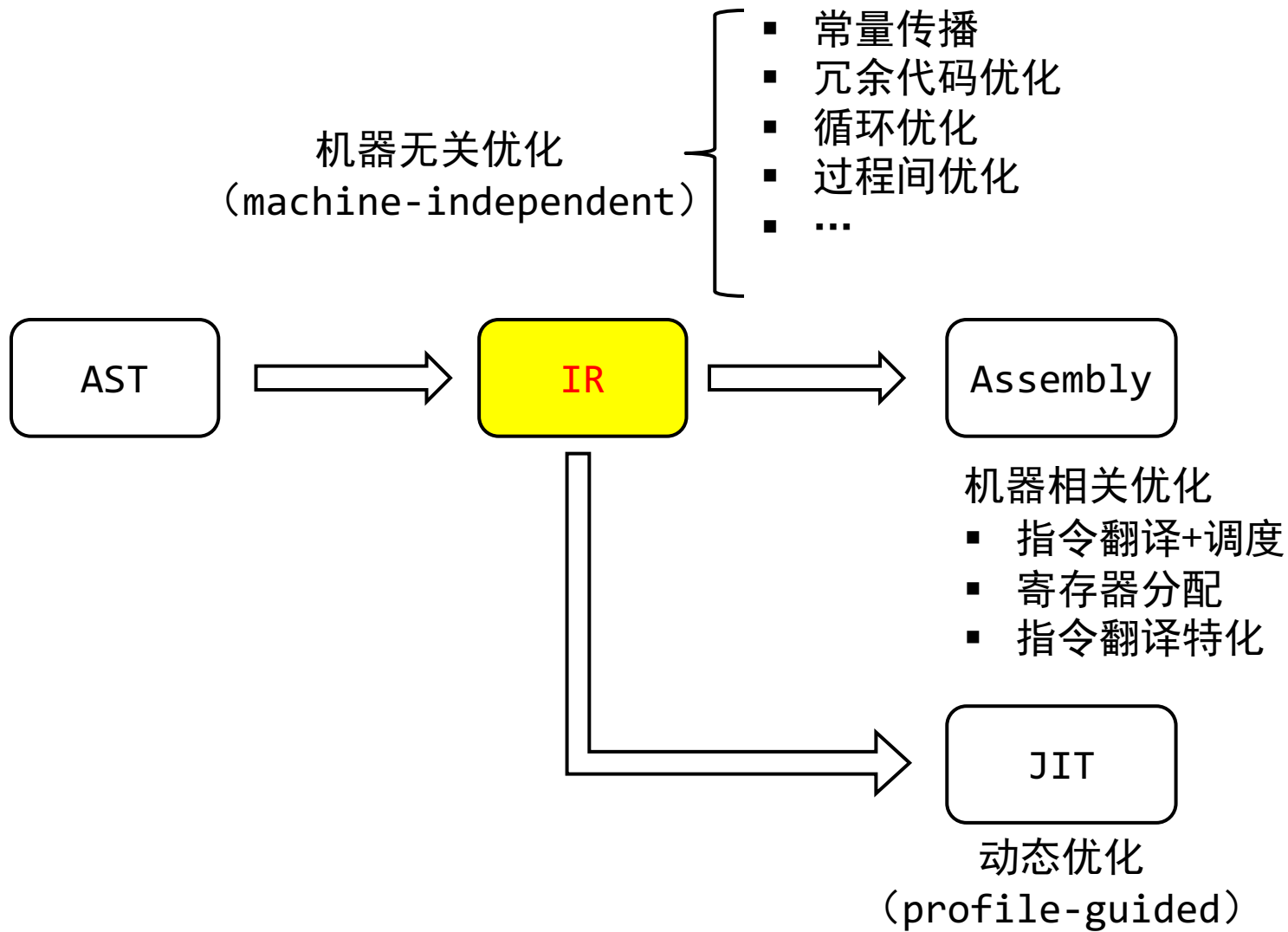
# 基于IR的优化

徐 辉

xuh@fudan.edu.cn



# 优化策略



# IR优化

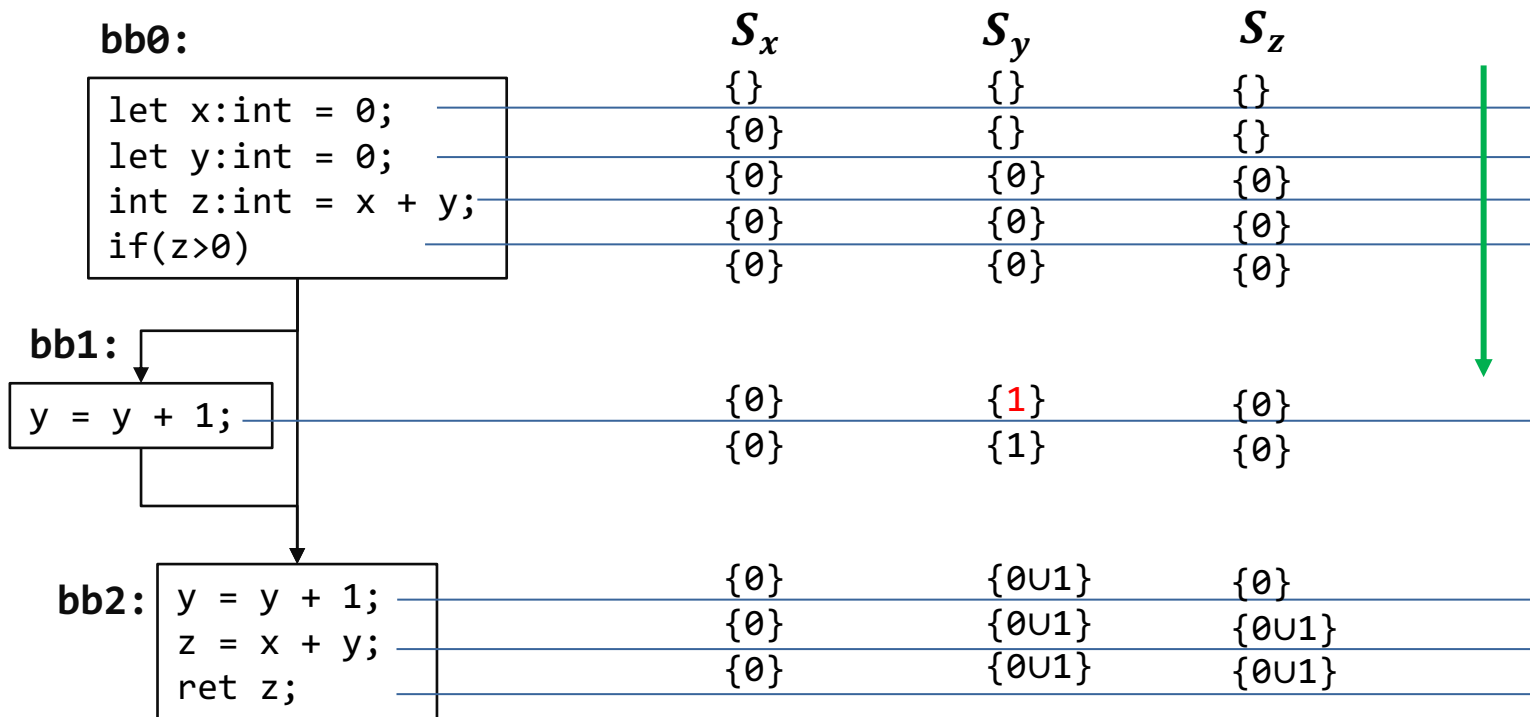
- ❖ 一、基于常量传播的优化
- ❖ 二、面向冗余代码的优化
- ❖ 三、循环优化
- ❖ 四、过程间优化

# 一、基于常量传播的优化

---

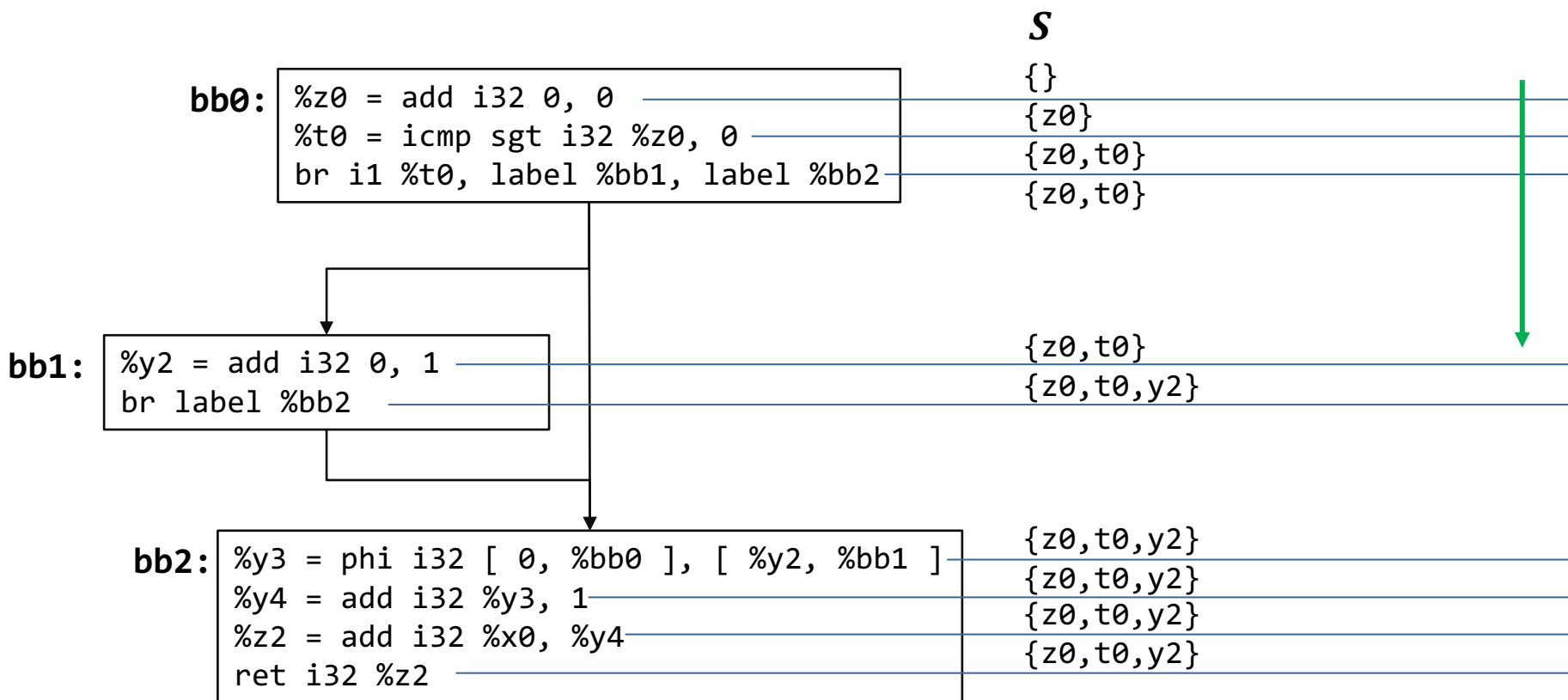
# 常量分析问题

- 分析变量x在特定程序节点p是否为常量



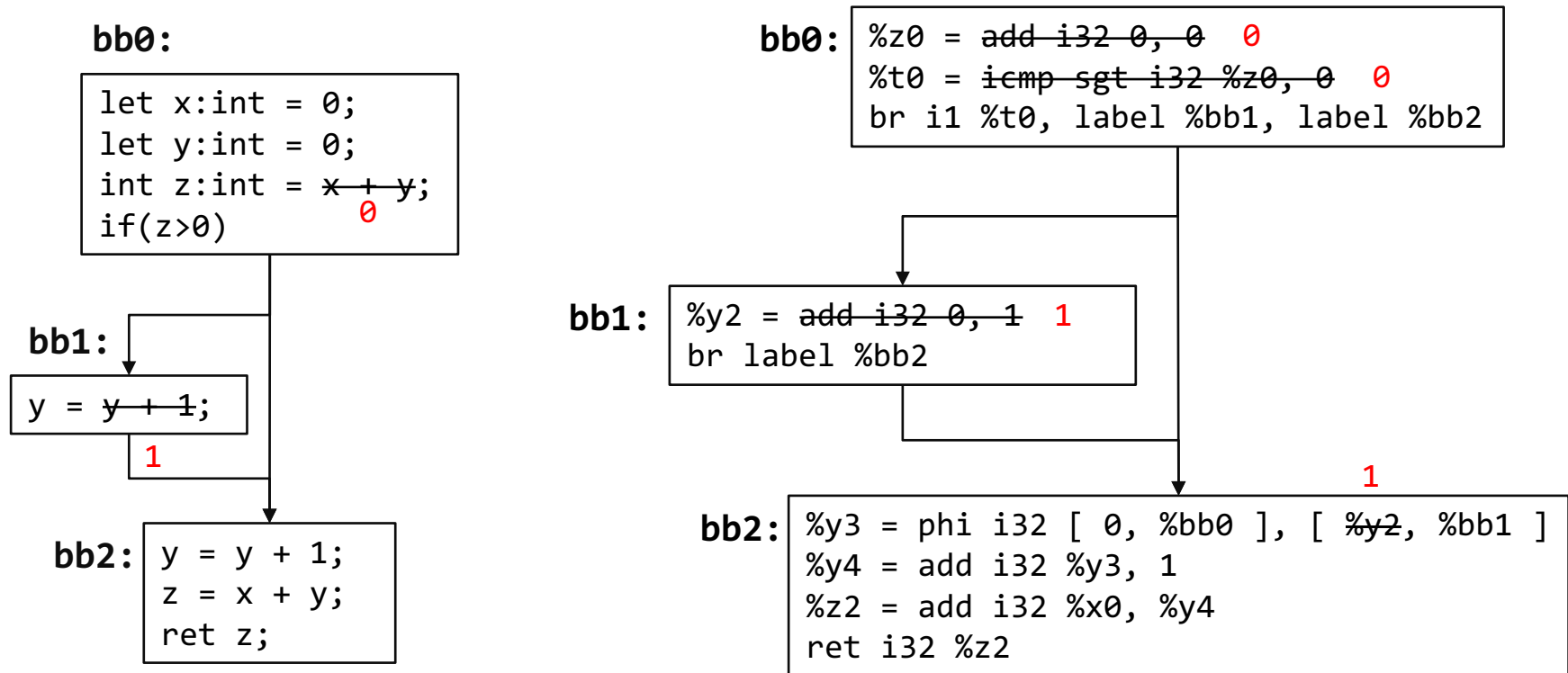
# 基于SSA的常量分析

- 分析哪些寄存器内容为常量

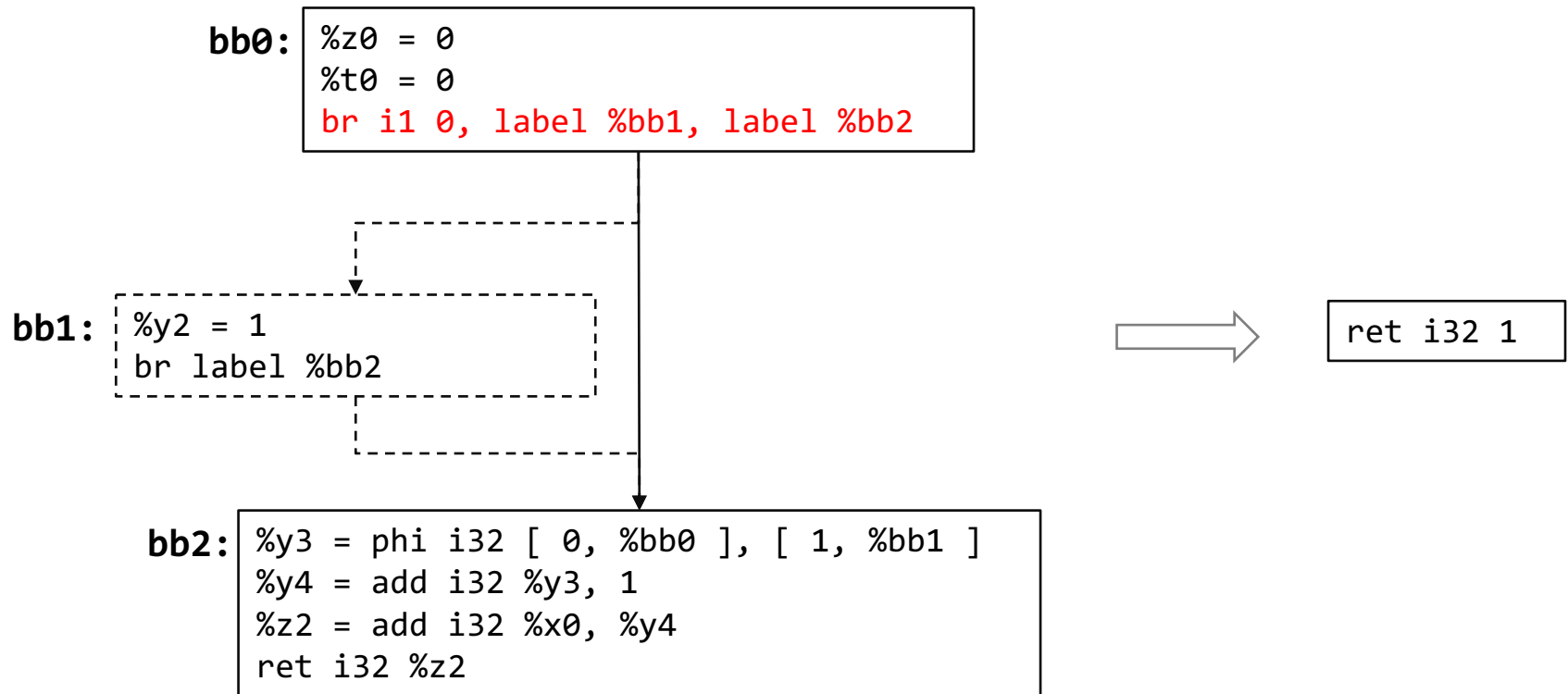


# 主要思想：在编译时完成常量相关的计算

- 常量传播：识别常量并将相应的变量替换为常量
- 常量折叠：编译时完成对常量表达式的计算



# 继续对代码进行优化





# 指令合并

- 两条二元运算指令可以合并的条件：
  - 指令1：一个运算数为常量，另一个为变量
  - 指令2：一个运算数为常量，另一个为指令1的运算结果

```
y = x + 1;  
...  
y = y + 2;  
...  
z = y + 3;
```



```
y = x + 1;  
...  
y = x + 3;  
...  
z = x + 6;
```

```
%y0 = add i32 %x0, 1  
...  
%y1 = add i32 %y0, 2  
...  
%z0 = add i32 %y1, 3
```



```
%y0 = add i32 %x0, 1  
...  
%y1 = add i32 %x0, 3  
...  
%z0 = add i32 %x1, 6
```

## 二、冗余代码优化

---

# 全局值编号 (Global Value Numbering)

- 相同的运算（运算符、运算数）只算一次即可

```
%y0 = add i32 %x0, 1  
...  
%y1 = add i32 %x0, 1  
...  
%z0 = add i32 %x0, 1
```

```
%y0 = add i32 %x0, 1  
...  
%y1 = %y0  
...  
%z0 = %y0
```

- 非LLVM IR; 必须以指令开头
- 直接替换USE(%y1)为USE(%y0)

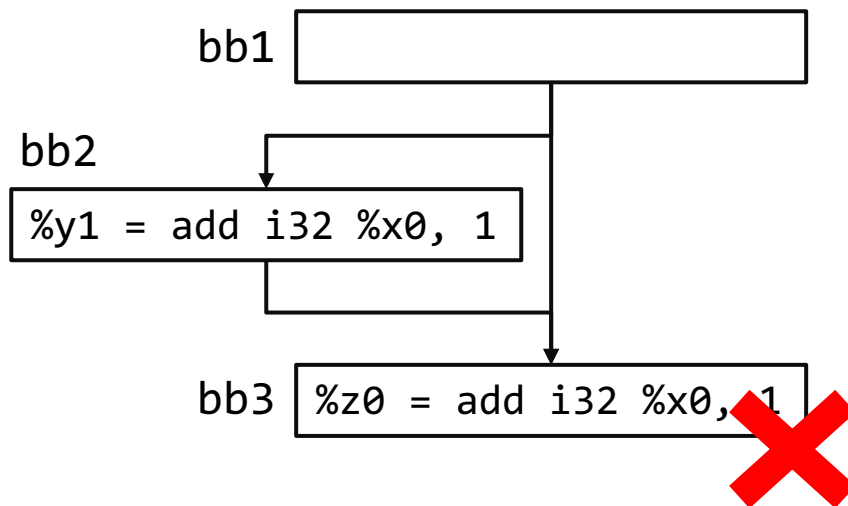
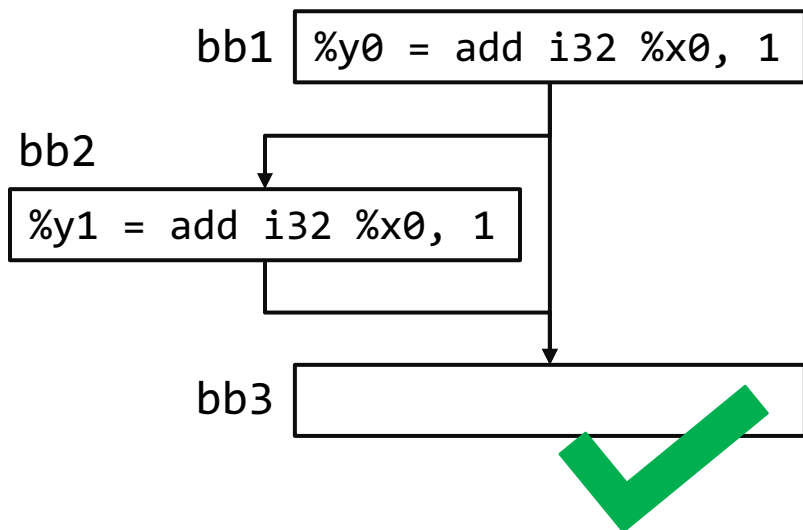


```
%y0 = add i32 %x0, 1  
...  
%x0 = bitcast i32 %y0 to i32  
...  
%y0 = bitcast i32 %y0 to i32
```

等价LLVM IR

# GVN: 公共表达式（可用表达式）

- 该表达式在存在支配关系的两条指令中重复出现

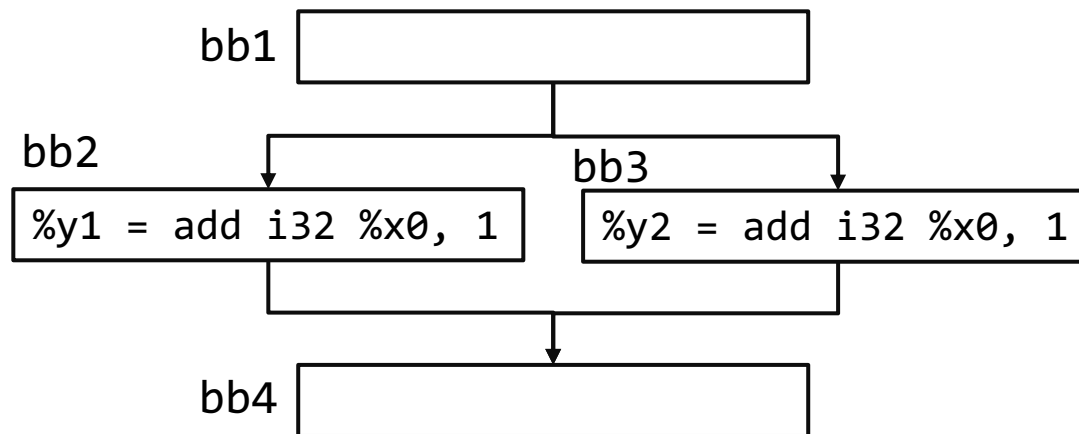


弱公共子表达式

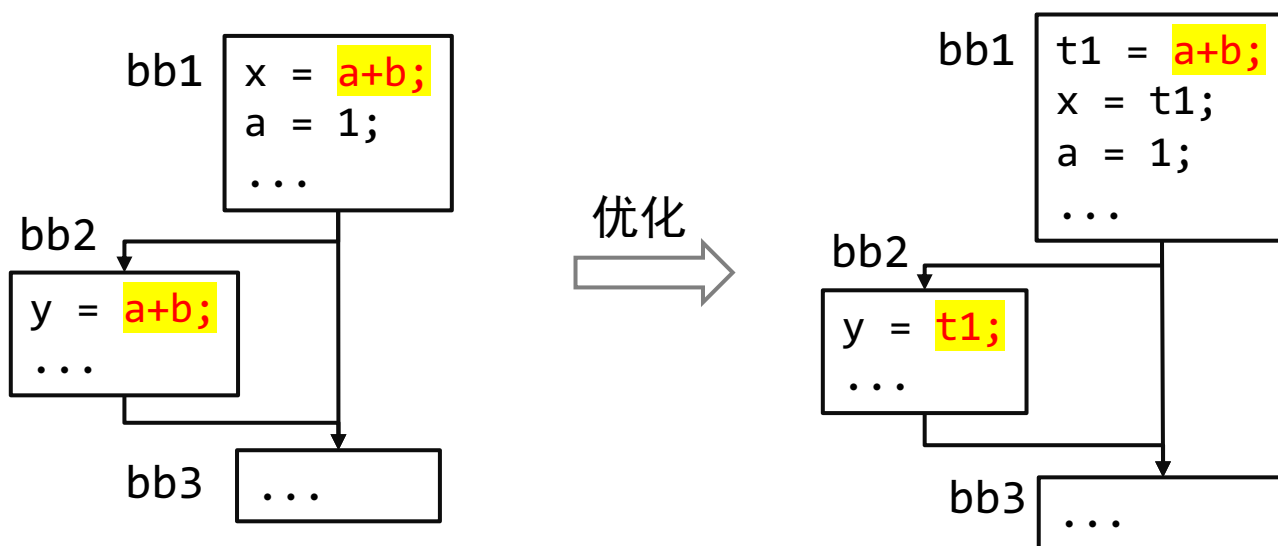
逆向数据流分析=>代码提升

# GVN: 繁忙表达式

- 不同代码分支中都存在的表达式
- 可以优化代码体积



# 基于非SSA形式做可用表达式分析？



- 正向遍历控制流图

- 如遇到指令:  $x = a + b$ 
  - $Gen(n) = \{ \langle a + b \rangle \}$
  - $KILL(n) = \{ \langle \varepsilon \rangle : \text{表达式} \varepsilon \text{ 包含 } x \}$
- ...
- $OUT(n) = (IN(n) - KILL(n)) \cup Gen(n)$

# 死代码

- 无用代码块：代码块不可达（条件语句恒真或恒假）
- 无用计算：缺少use的def
- 无用参数：IR中没有use/store该参数
- 无用局部变量：IR中没有load该局部变量

## 三、循环优化

---



# 循环中的不变代码

- 出现位置：循环条件、循环体中都可能出现

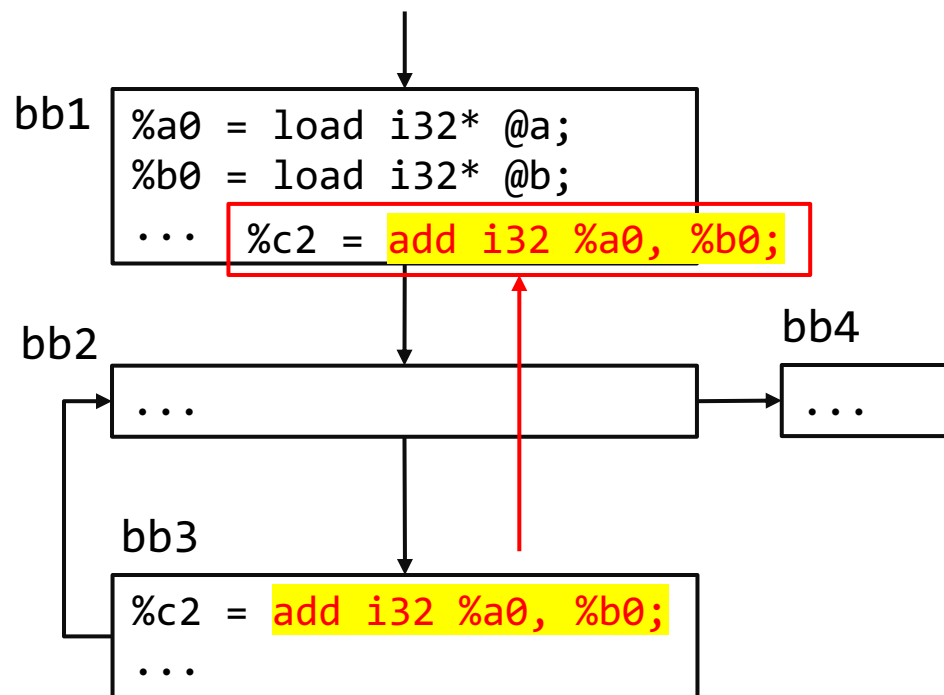
```
let a = ...;
let b = ...;
let s:list = ...;
for i in 1..100 {
  let t = (a + b)*i;
  s.push(t);
}
```

```
let a = ...;
let b = ...;
let s:list = ...;
for i in 1..s.len() {
  let t = (a + b)*i;
  s[i] = t;
}
```

```
let a = ...;
let b = ...;
let s:list = ...;
for i in 1..100 {
  let t = foo();
  s.push(t);
}
```

```
let a = ...;
let b = ...;
let s:list = ...;
for i in 1..s.len() {
  let t = s.pop();
  s[i] = t;
}
```

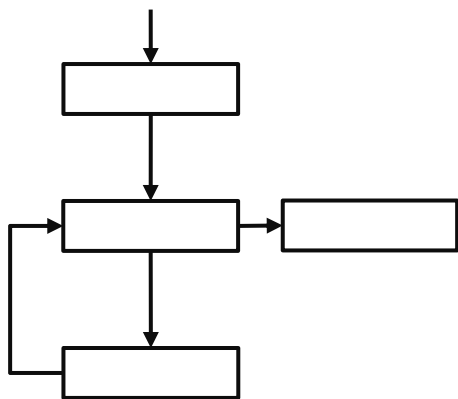
# 循环不变代码



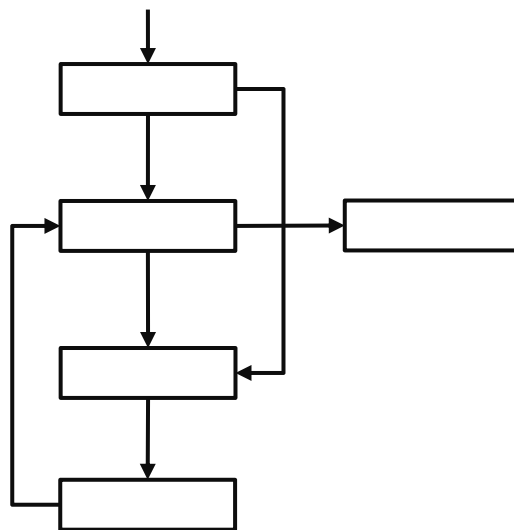
- 检测循环不变代码
  - 操作数定义自循环外部
  - 如何检测循环?
- 前移到循环外部
  - 支配节点

# 自然循环natural loop

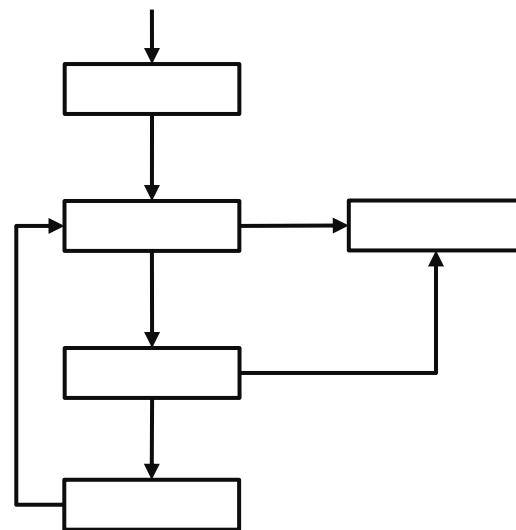
- 一个循环是自然循环的条件：
  - 有唯一的入口（支配所有节点）
  - 返回入口节点的返回边
- 一般正常的控制流语句形成的环： while、if-else、 for
  - goto语句会造成非自然循环



自然循环



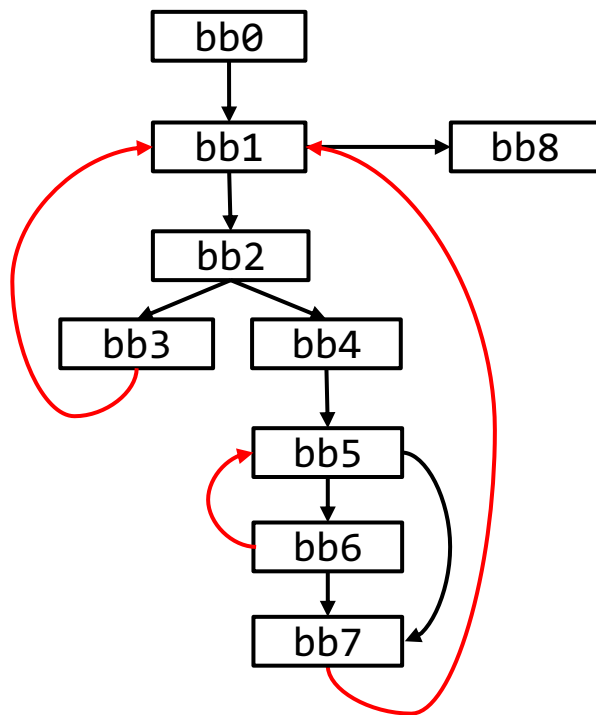
非自然循环



自然循环

# 自然循环的性质

- 两个自然循环之间不相交：相切、嵌套、分离
- 两个首节点相同的自然循环：嵌套、相切
- 自然循环标识：每条返回边对应一个自然循环



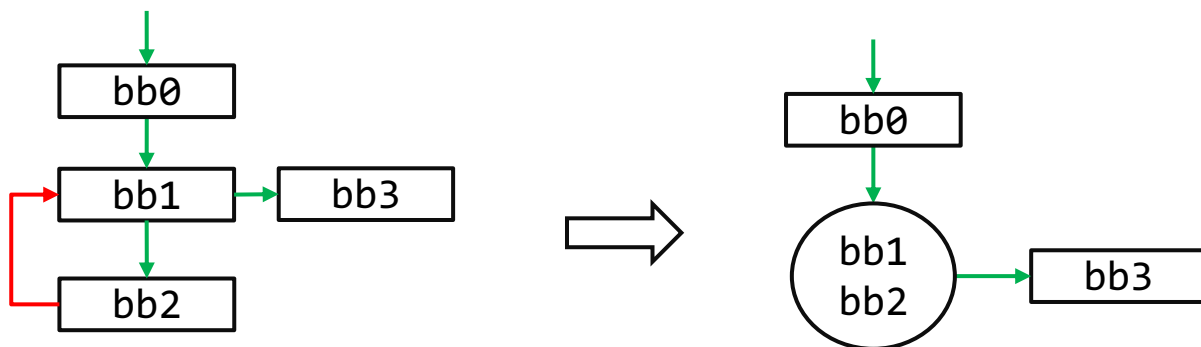
bb3->bb1 : 1-2-3


bb6->bb5 : 5-6

bb7->bb1 : 1-2-4-5-6-7

# 可规约控制流图：Reducible CFG

- 可规约CFG的所有循环都是自然循环
- 边可以分为前进边和返回边两个不交集=>可以缩环

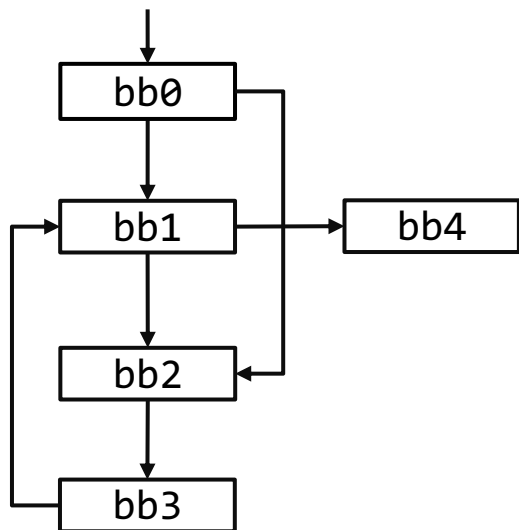


入边： 

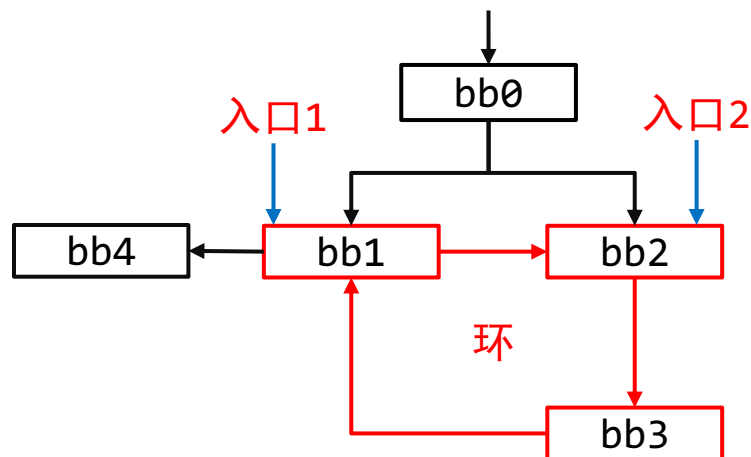
出边： 

# 不可规约控制流图

- 无法确定循环入口和返回边



非自然循环



同态图  
isomorphic

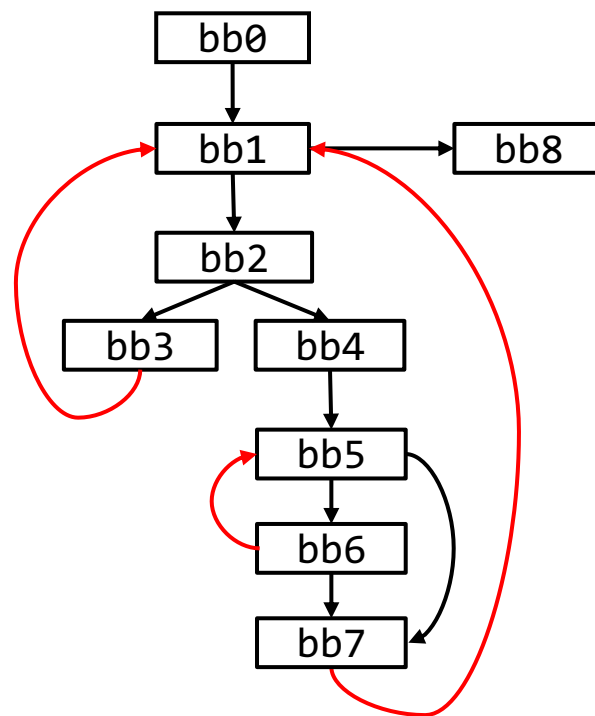
# 自然循环检测算法

- 基本思路：

- 1) 遍历CFG=>支配关系矩阵M1
- 2) 比对图邻接表M2=>检测返回边
- 3) 识别每一条回边对应的环

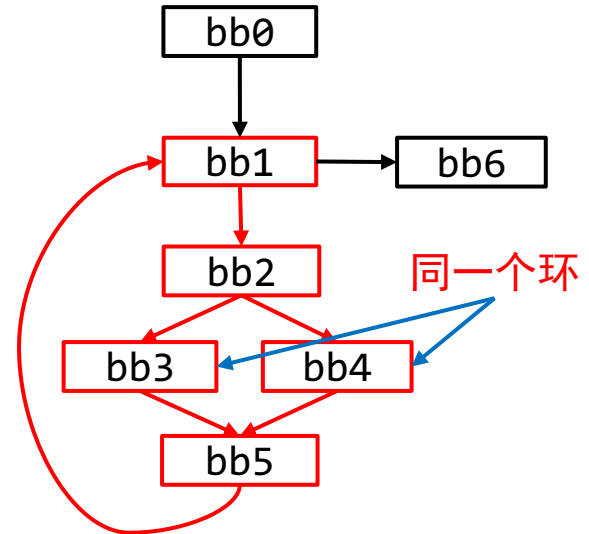
- DFS检测环

- 如果栈中已存在节点=>返回边
- 若干栈顶元素组成环
- 相同返回边的环需要合并



# DFS检测环

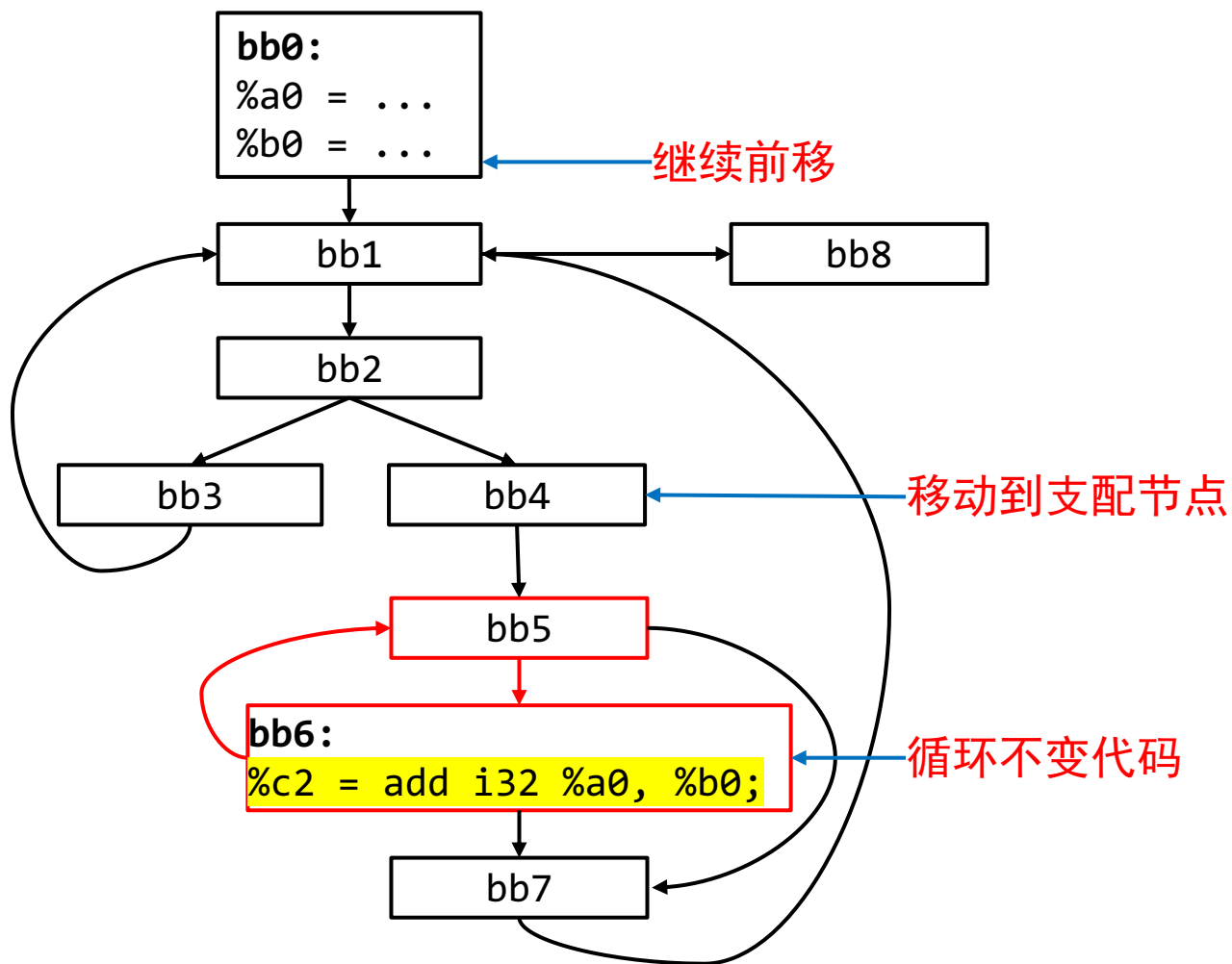
```
stack s;  
DFSVisit(v) {  
    s.push(v);  
    for each w in OUT(v) {  
        if s.contains(w) { //找到回边  
            AddLoopback(w,v);  
        } else {  
            DFSVisit(w);  
        }  
    }  
}  
AddLoopback(v,w) {  
    new = CreateLoop(top n items of s untill w);  
    old = Findloop(v, w)  
    merge(old,new)  
}
```





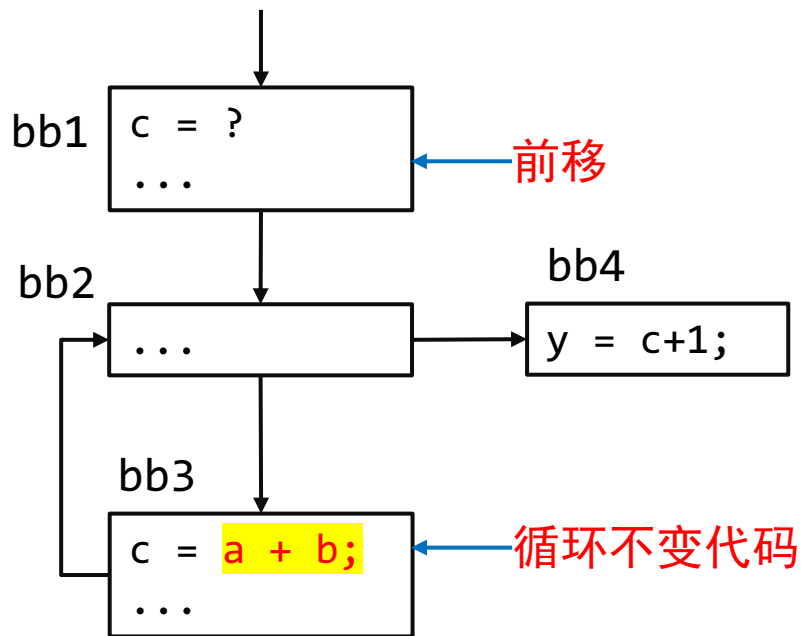
# 前移位置

- 单层循环：前移到最近的支配节点
- 多层循环：前移至不能移动为止

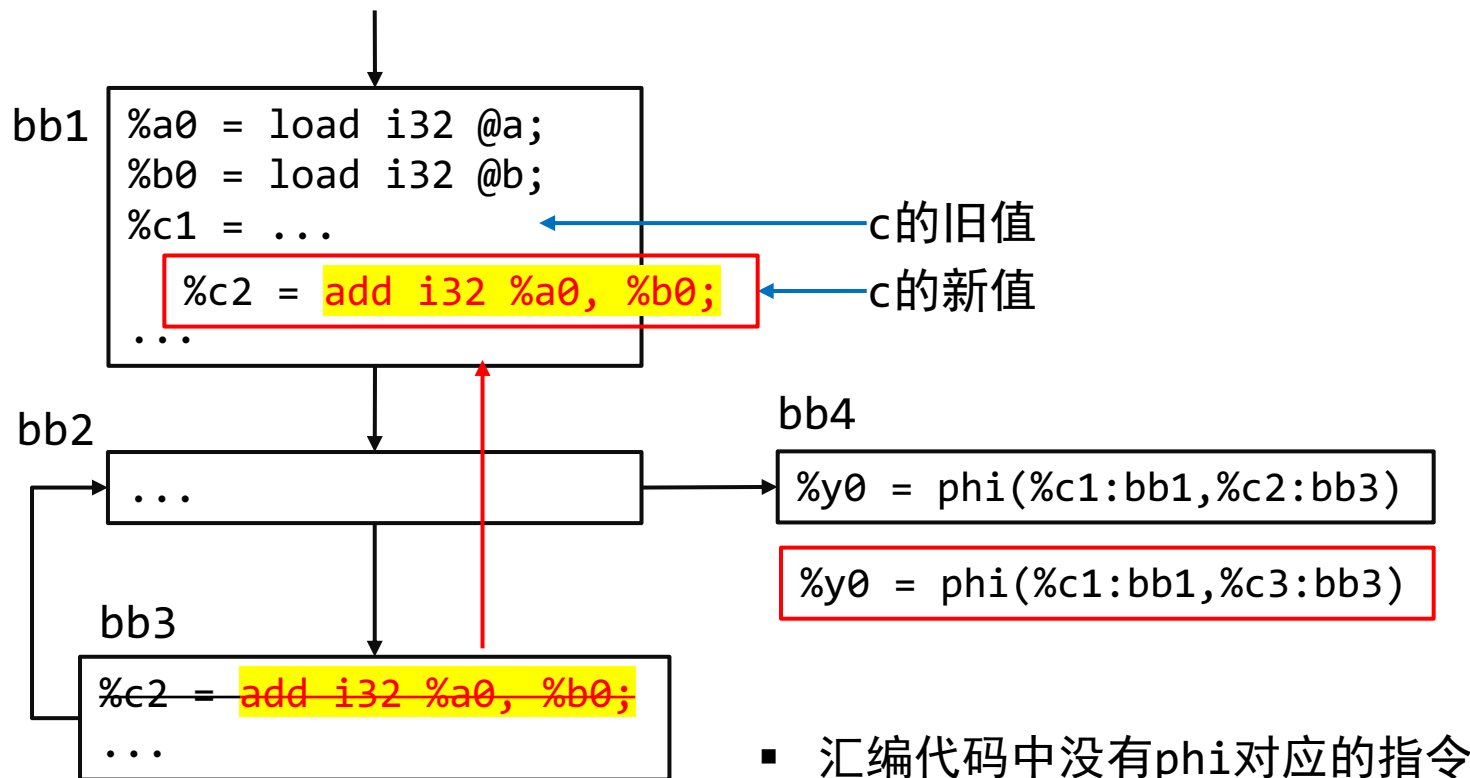


# 可能会有副作用？

- 如未进入循环，会错误修改x的值



# SSA形式会有副作用吗？



- 汇编代码中没有phi对应的指令
- 翻译汇编前用store替代phi

# 归纳变量

- 变量 $x$ 的值每轮循环增加固定值，则称 $x$ 为归纳变量
  - 基本归纳变量 $x$
  - 依赖归纳变量 $y = ax + b$ ， $a$ 和 $b$ 为常量

```
for i in 1..100 {  
  y = 10 * i + 1;  
  s[i] = y;  
}
```



```
t1 = 1;  
for i in 1..100 {  
  t1 = t1 + 10;  
  s[i] = t1;  
}
```

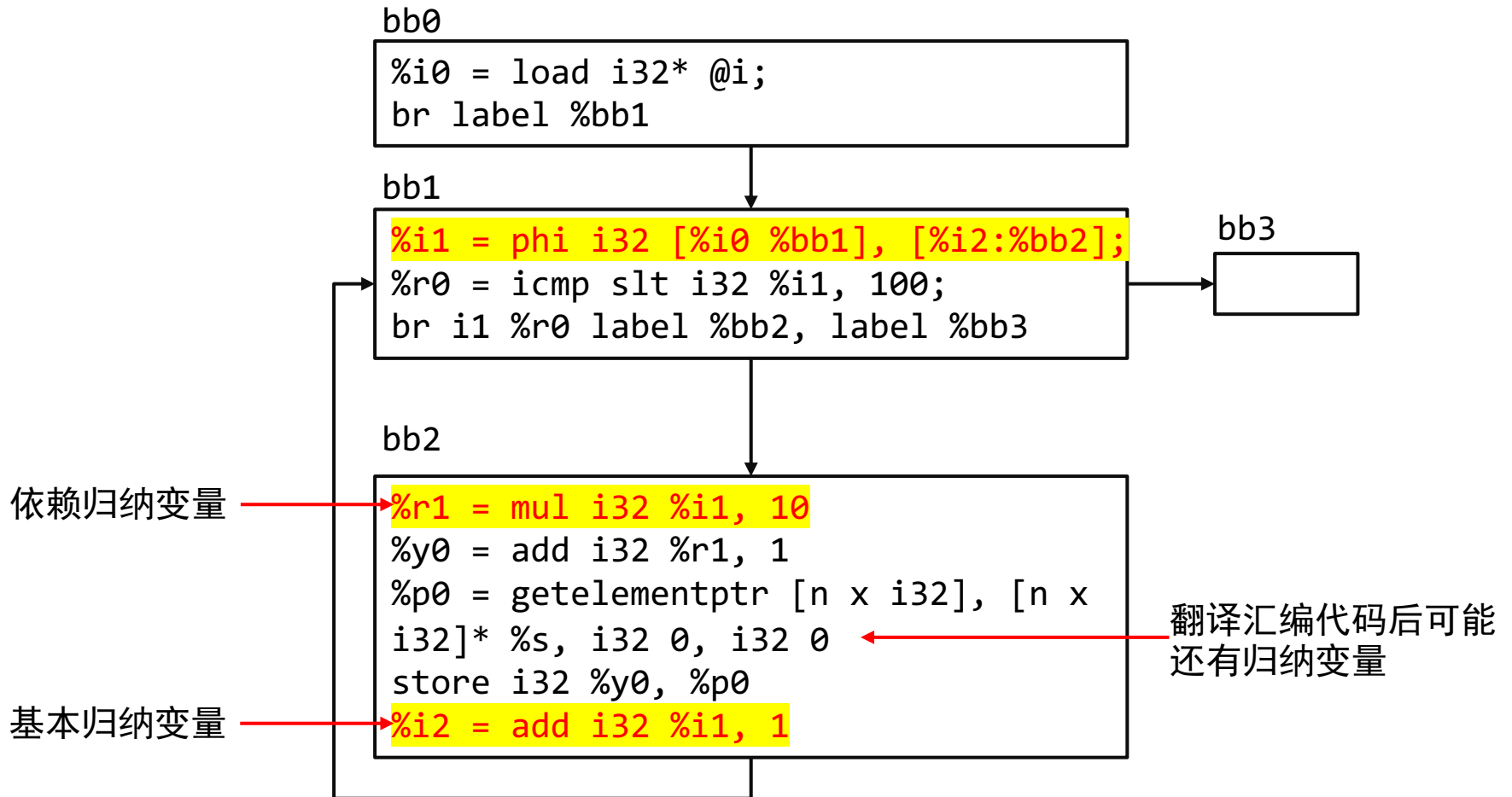


```
let i:int = 1;  
while(i<100) {  
  y = 10 * i + 1;  
  s[i] = y;  
  i = i + 1;  
}
```



```
let i:int = 1;  
let t1 = 1;  
while(i<100) {  
  y = t1 + 10;  
  s[i] = y;  
  i = i + 1;  
}
```

# 基于IR识别归纳变量



# 标量替换: Scala Replacement

- 使用标量替换循环内部的频繁内存读写操作
- 在IR层自动替换 $R[i][j]$ 的难点?  $R[i][j]$ 和 $i$ 可能是alias

```
for i in 0..rowA {  
  for j in 0..colB {  
    for k in 0..colA {  
       $R[i][j]$  =  $R[i][j]$  + A[i][k]*B[k][j];  
    }  
  }  
}
```

每次从内存或cache读写 $R[i][j]$ 会影响性能



```
for i in 0..rowA {  
  for j in 0..colB {  
     $t = R[i][j]$ ;  
    for k in 0..colA {  
       $t$  =  $t$  + A[i][k]*B[k][j];  
    }  
     $R[i][j] = t$ ;  
  }  
}
```

使用临时变量替换, 可直接使用寄存器中的值

# 降低分支预测的代价

- Loop unswitching: 外提（减少）循环内条件判断
- Loop unroll: 将循环体复制多遍

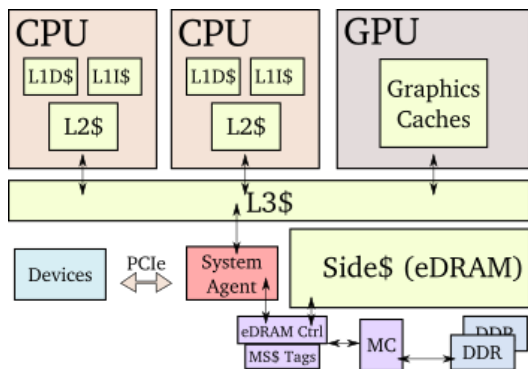
```
void testbrpred(int* a, int len, int x){
    unsigned long long cycle = rdtsc();
    while(len>-1){
        len-=1;
        if(a[len]>x) ;
        else ;
    }
    unsigned long long cycl = rdtsc()- cycle;
    printf("x = %d, cycles = %d\n", x, cycl);
}
```

```
int main(int argc, char** argv){
    int a[1000];
    srand(time(NULL));
    for(int i = 0; i < 1000; i++) a[i] = rand()%1000;
    testbrpred(a,1000,100);
    testbrpred(a,1000,300);
    testbrpred(a,1000,500);
    testbrpred(a,1000,700);
    testbrpred(a,1000,900);
}
```

```
x = 100, cycles = 23630
x = 300, cycles = 47175
x = 500, cycles = 63744
x = 700, cycles = 49642
x = 900, cycles = 26301
```

# Cache

- Cache访问速度优于内存访问速度
- 最小单位是cache line
- 通过降低cache miss提升代码性能



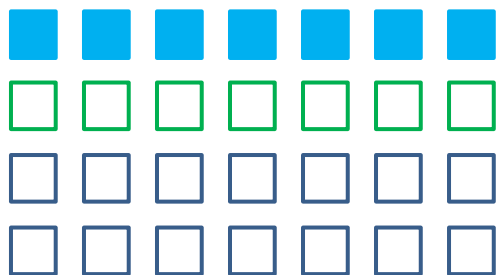
index	valid	tag	data
001	0x...		64 B
002	0x...		64 B
003	0x...		64 B
...	0x...		64 B

cache	size	line	speed
L1	32 KB + 32 KB	64 B	4-5 cycles
L2	256 KB	64 B	12 cycles
L3	up to 2 MB	64 B	30-50 cycles

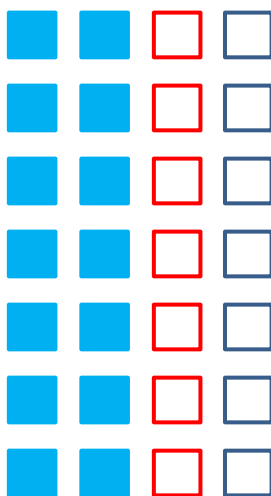


# 矩阵乘法：循环分块

```
for i in 0..rowA {  
  for j in 0..colB {  
    for k in 0..colA {  
      R[i][j] = R[i][j] + A[i][k]*B[k][j];  
    }  
  }  
}
```



A[rowA][colA]



B[rowB][colB]

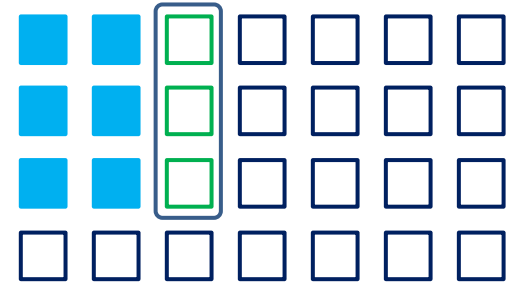
- 假设：

- 已经算完 $R[0][0]..R[0][j]$
- $B[][0]..B[][j]$ 已经在cache中

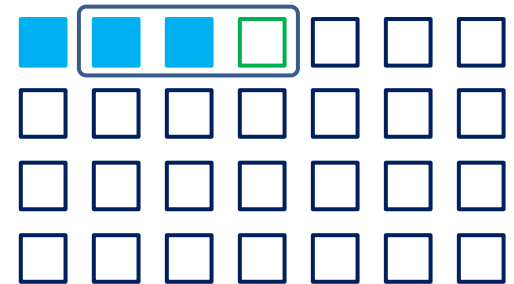
- 先算 $R[0][j+1]$ 还是 $R[1][0]$ ?

# 循环交换

```
for i in 1..m-2 {  
  for j in 0..n-1 {  
    R[i][j] = A[i-1][j] + A[i][j] + A[i+1][j];  
  }  
}
```



```
for j in 0..n-1 {  
  for i in 1..m-2 {  
    R[i][j] = A[i-1][j] + A[i][j] + A[i+1][j];  
  }  
}
```



# 循环合并和拆分

```
for i in 0..n-1 {  
    R1[i] = A[i] + B[i];  
}  
for i in 0..n-1 {  
    R2[i] = A[i] + B[i];  
}
```

合并  
fusion

```
for i in 0..n-1 {  
    R1[i] = A[i] + B[i];  
    R2[i] = A[i] + B[i];  
}
```

```
for i in 0..n-1 {  
    R1[i] = A[i] + B[i];  
    R2[i] = C[i] + D[i];  
}
```

拆分  
distribution

```
for i in 0..n-1 {  
    R1[i] = A[i] + B[i];  
}  
for i in 0..n-1 {  
    R2[i] = C[i] + D[i];  
}
```

可能对寄存器分配有利：减少冲突关系

## 四、过程间优化

---

# 函数优化策略

常用优化技术  
(内联)

partial evaluation  
(program specialization)

compile-time  
execution

参数均未知

部分参数已知

全部参数已知

```
fn foo(n:int, r:int) -> int {  
    if (n == 0) {  
        ret r;  
    } else {  
        ret factorial(n-1, n*r);  
    }  
}  
  
fn foo(x:int) -> int {  
    foo(x, x+1);  
    foo(x, 1);  
    foo(0, 0);  
}
```

# 内联


- 优点：
  - 避免函数调用规约带来的运行时开销
  - 便于代码优化，如partial evaluation
- 缺点：
  - 代码复制可能会增大代码体积
  - 函数体太大不利于寄存器分配
- 给定bugget上限，选取最优的内联函数组合
  - 转化为背包问题（Knapsack problem）

$$\max \sum_{i=1}^n v_i x_i \quad s.t. \sum_{i=1}^n w_i x_i \leq thres \text{ and } x_i \in \{0,1\}$$


# 尾递归函数

- return前的最后一条语句调用自己

```
fn factorial(n:int) -> int {  
  if (n == 0) {  
    ret 1;  
  }  
  else {  
    ret n * factorial(n-1);  
  }  
}
```



```
fn factorial(n:int, r:int) -> int {  
  if (n == 0) {  
    ret r;  
  }  
  else {  
    ret factorial(n-1, n*r);  
  }  
}
```



```
fn i32 factorial(%n0:i32, %r0:i32) {  
%bb0:  
  %n = alloca i32  
  %r = alloca i32  
  store i32 %n0, %n  
  store i32 %r0, %r  
  br label %bb1  
%bb1  
  %n1 = load i32, i32* %n  
  %r0 = icmp eq i32 %n1, 0;  
  br i1 %r0 label %bb2, label %bb3  
%bb2:  
  %r1 = load i32, i32* %r  
  ret i32 %r1;  
%bb3:  
  %n2 = load i32, i32* %n  
  %r2 = load i32, i32* %r  
  %t0 = sub i32 %n2, 1;  
  %t1 = mul i32 %n2, %r2;  
  %t2 = call i32 @factorial(%t0, %t1);  
  ret %t2;  
}
```

# 尾递归消除

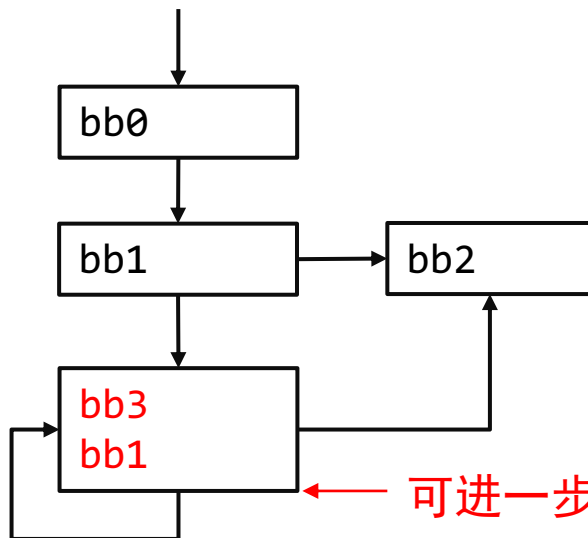
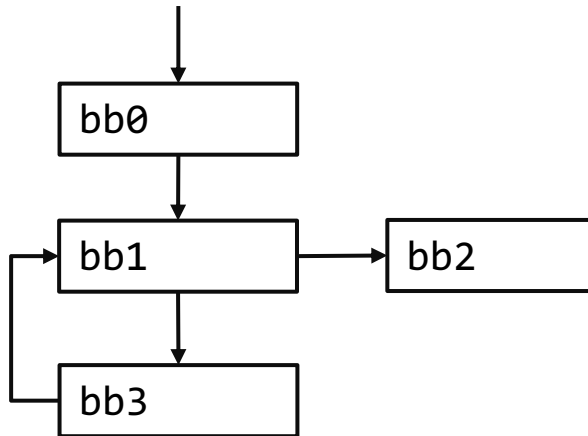
```
fn i32 factorial(%n0:i32, %r0:i32) {  
  %bb0:  
    %n = alloca i32  
    %r = alloca i32  
    store i32 %n0, %n  
    store i32 %r0, %r  
    br label %bb1  
  %bb1  
    %n1 = load i32, i32* %n  
    %r0 = icmp eq i32 %n1, 0;  
    br i1 %r0 label %bb2, label %bb3  
  %bb2:  
    %r1 = load i32, i32* %r  
    ret i32 %r1;  
  %bb3:  
    %n2 = load i32, i32* %n  
    %r2 = load i32, i32* %r  
    %t0 = sub i32 %n2, 1;  
    %t1 = mul i32 %n2, %r2;  
    %t2 = call i32 @factorial(%t0, %t1);  
    ret %t2;  
}
```



```
fn i32 factorial(%n0:i32, %r0:i32) {  
  %bb0:  
    %n = alloca i32  
    %r = alloca i32  
    store i32 %n0, %n  
    store i32 %r0, %r  
    br label %bb1  
  %bb1  
    %n1 = load i32, i32* %n  
    %r0 = icmp eq i32 %n1, 0;  
    br i1 %r0 label %bb2, label %bb3  
  %bb2:  
    %r1 = load i32, i32* %r  
    ret i32 %r1;  
  %bb3:  
    %n2 = load i32, i32* %n  
    %r2 = load i32, i32* %r  
    %t0 = sub i32 %n2, 1;  
    %t1 = mul i32 %n2, %r2;  
    store i32 %t0, %n  
    store i32 %t1, %r  
    br %bb1  
}
```



# 尾递归优化



```
fn i32 factorial(%n0:i32, %r0:i32) {
%bb0:
    %n = alloca i32
    %r = alloca i32
    store i32 %n0, %n
    store i32 %r0, %r
    br label %bb1
%bb1:
    %n1 = load i32, i32* %n
    %r0 = icmp eq i32 %n1, 0;
    br i1 %r0 label %bb2, label %bb3
%bb2:
    %r1 = load i32, i32* %r
    ret i32 %r1;
%bb3:
    %n2 = load i32, i32* %n
    %r2 = load i32, i32* %r
    %t0 = sub i32 %n2, 1;
    %t1 = mul i32 %n2, %r2;
    store i32 %t0, %n
    store i32 %t0, %r
    %n3 = load i32, i32* %n
    %r2 = icmp eq i32 %n3, 0;
    br i1 %r0 label %bb2, label %bb3
}
```

# Sibling Call优化

- Caller和callee函数签名相同并且是tail call
- 栈帧结构复用（汇编代码优化）

```
fn foo(a:int) -> int{  
  if (a < 0) {  
    return a;  
  }  
  let b:int = a - 1;  
  ret bar(b);  
}
```

```
fn bar(b:int) -> int{  
  let c:int = b - 1;  
  ret c;  
}
```