

## Lecture 5.2

# 常用代码优化技术

徐 辉

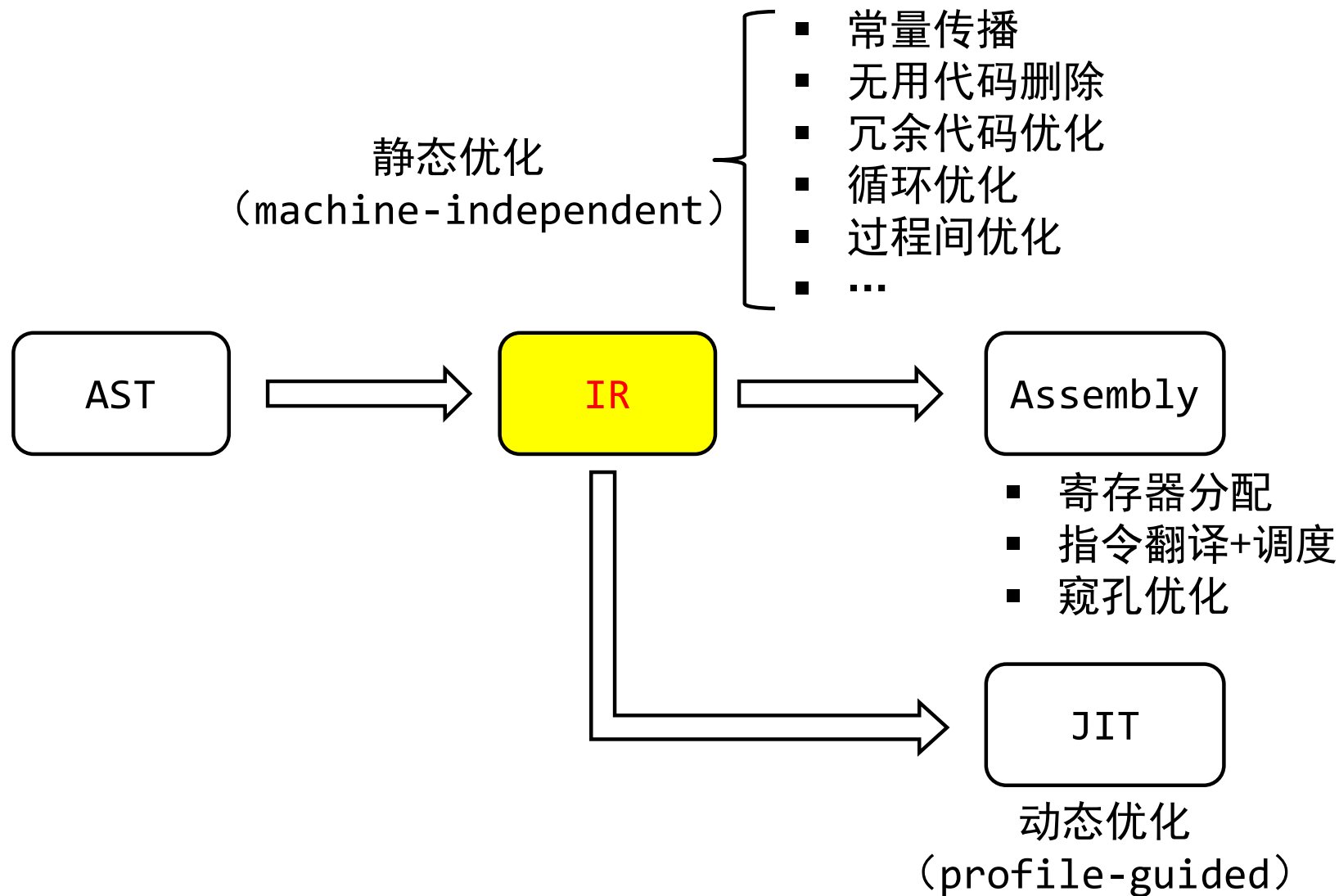
xuh@fudan.edu.cn



# 大纲

- 一、冗余代码优化
- 二、循环优化
- 三、过程间优化

# 优化策略



# 无用代码

- 无用类型声明：代码中没有声明新类型相关的对象
- 无用参数：IR中没有load该参数（局部变量）
- 无用局部变量：IR中没有load该局部变量
- 无用全局变量：IR中没有load该全局变量
- 无用计算：计算的值def缺少对应的use
- 无用代码块：代码块不可达
  - 条件跳转语句对应的条件谓词（predicate）为恒真或恒假
  - match-case间接跳转地址问题

# 无用代码块检测：谓词predicate分析

```
fn foo (x:int) {  
  if(x != -1) {  
    ...  
  }  
  else {  
    ...  
  }  
}
```

```
foo(1)
```

```
fn foo (x:int) {  
  if(x*x= -1) {  
    ...  
  }  
  else {  
    ...  
  }  
}
```

```
fn foo (x:int) {  
  while (x>0) {  
    if(x != -1) {  
      ...  
    }  
    else {  
      ...  
    }  
  }  
}
```

# match-case类似

```
match(x*x){  
  -1: => { x = 0; }  
  0:  => { x = 1; }  
  _:  => { x = -1; }  
}
```

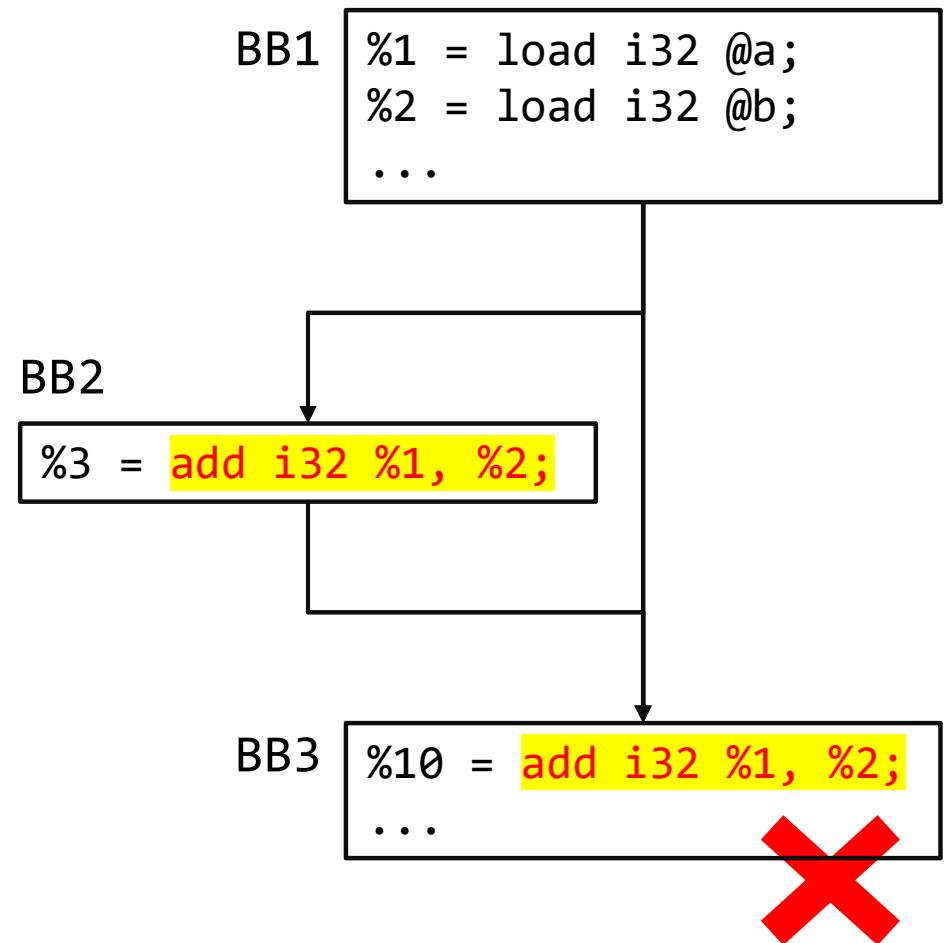
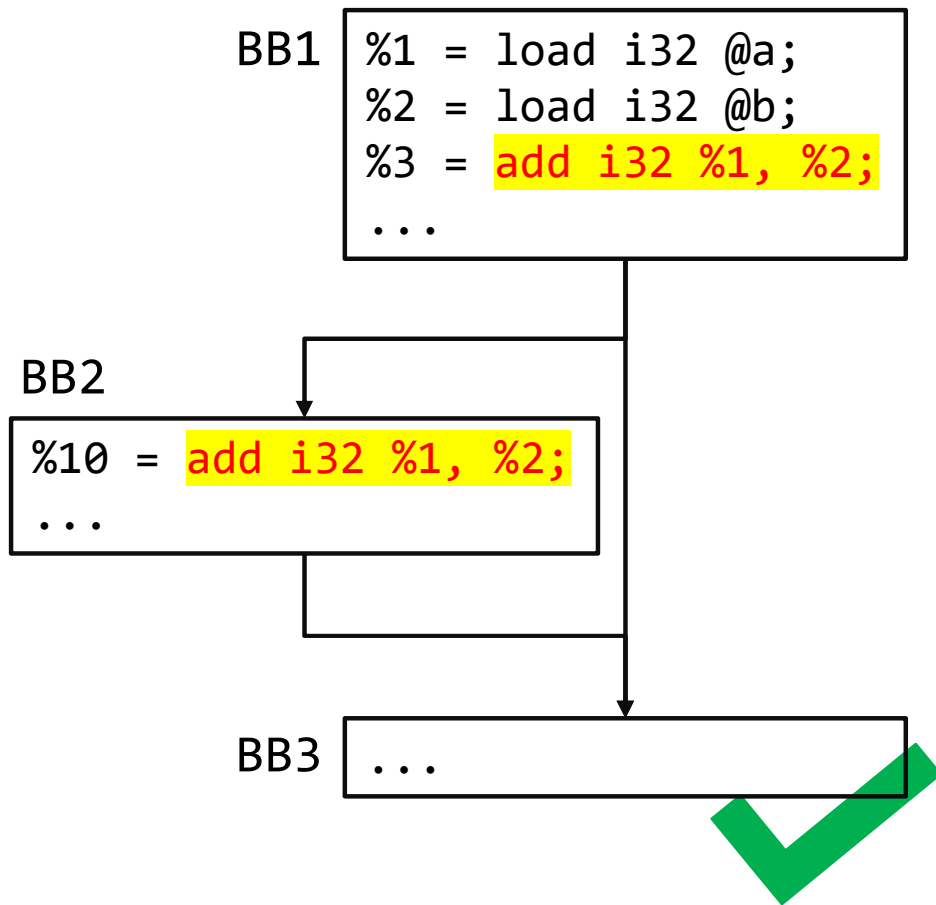


```
%0 = load i32 %x;  
%1 = mul i32 %1, %1;  
match i32 %1, %BB3 [  
  i32 -1, %BB1  
  i32 0, %BB2  
]  
%BB1:  
  store i32 0, %x;  
  jmp %BB4  
%BB2:  
  store i32 1, %x;  
  jmp %BB4  
%BB3:  
  store i32 -1, %x;  
  jmp %BB4  
%BB4:  
  ...
```

# 冗余代码优化

- 公共子表达式：
  - 同一个表达式出现多次，且存在支配关系
- 部分冗余代码删除
  - 部分路径存在公共子表达式
- 代码提升
  - 同一个表达式在不同路径中出现多次，不存在先后关系

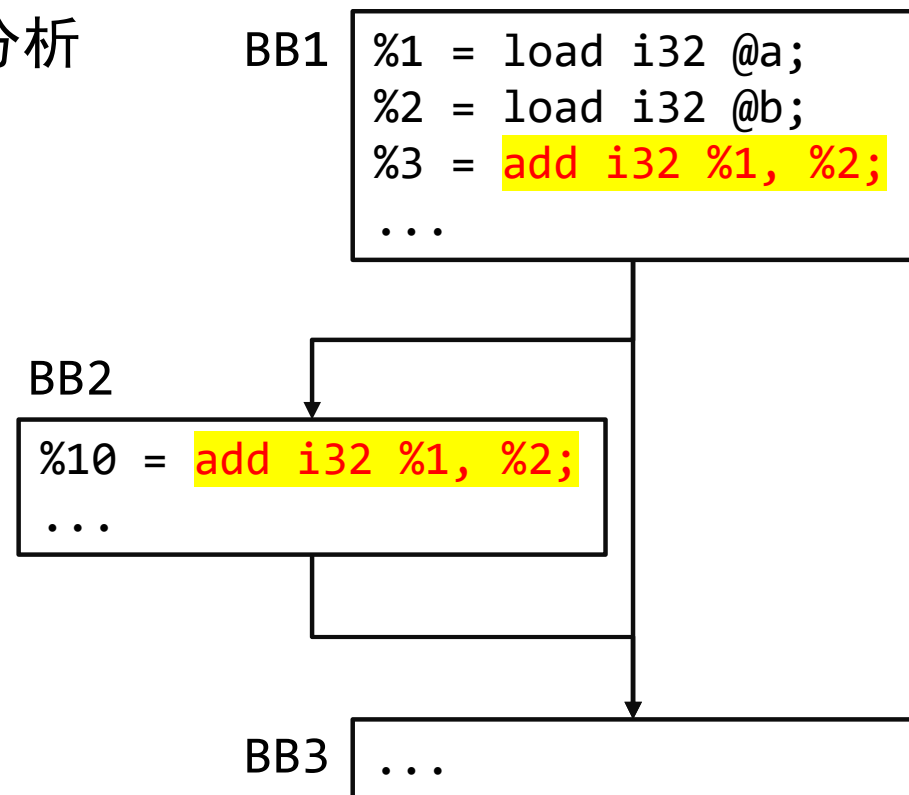
# 公共子表达式：SSA



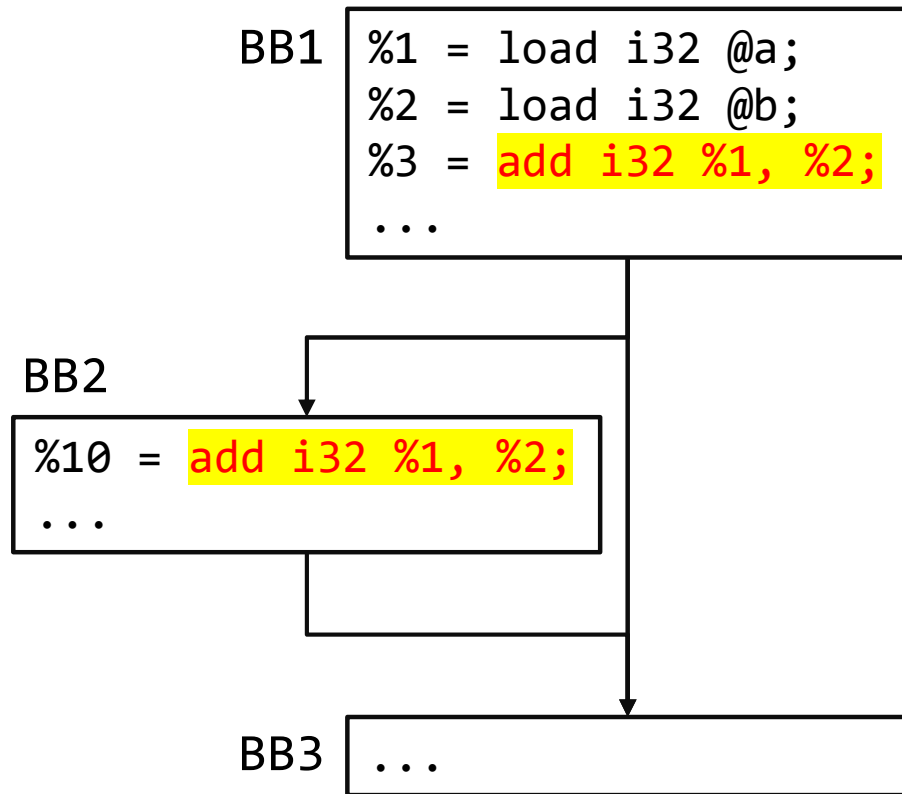


# 可用表达式分析

- 该表达式在存在支配关系的两条指令中重复出现
- 且其操作数都未被重新赋值
  - SSA形式一定未被重新赋值
- 可用表达式 vs 可达性分析

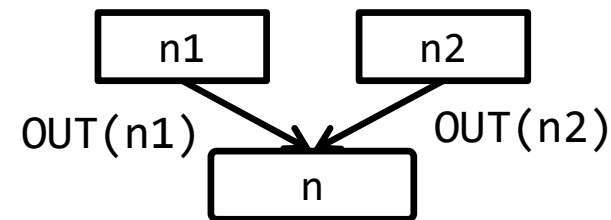


# 可用表达式分析：SSA形式



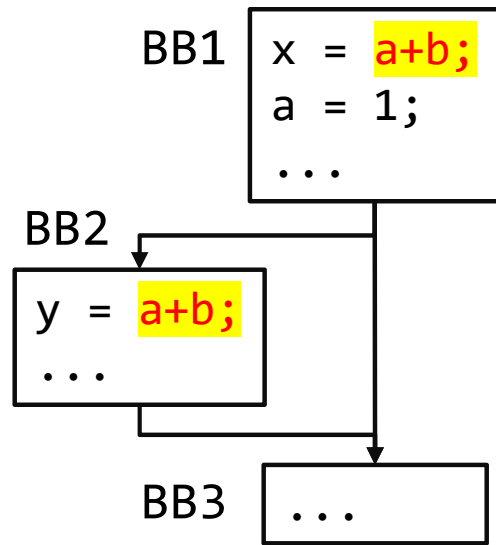
## 正向遍历控制流图

- 如遇到指令:  $x = \text{add } i32 \text{ } \%1, \%2$ 
  - $\text{Gen}(n) = \{ \langle \text{add } i32, \%1 \%2 \rangle \}$
- ...
- $\text{OUT}(n) = \text{IN}(n) \cup \text{Gen}(n)$



$$\text{IN}(n) = \bigcap_{n' \in \text{predecessor}(n)} \text{OUT}(n')$$

# 可用表达式分析：非SSA形式

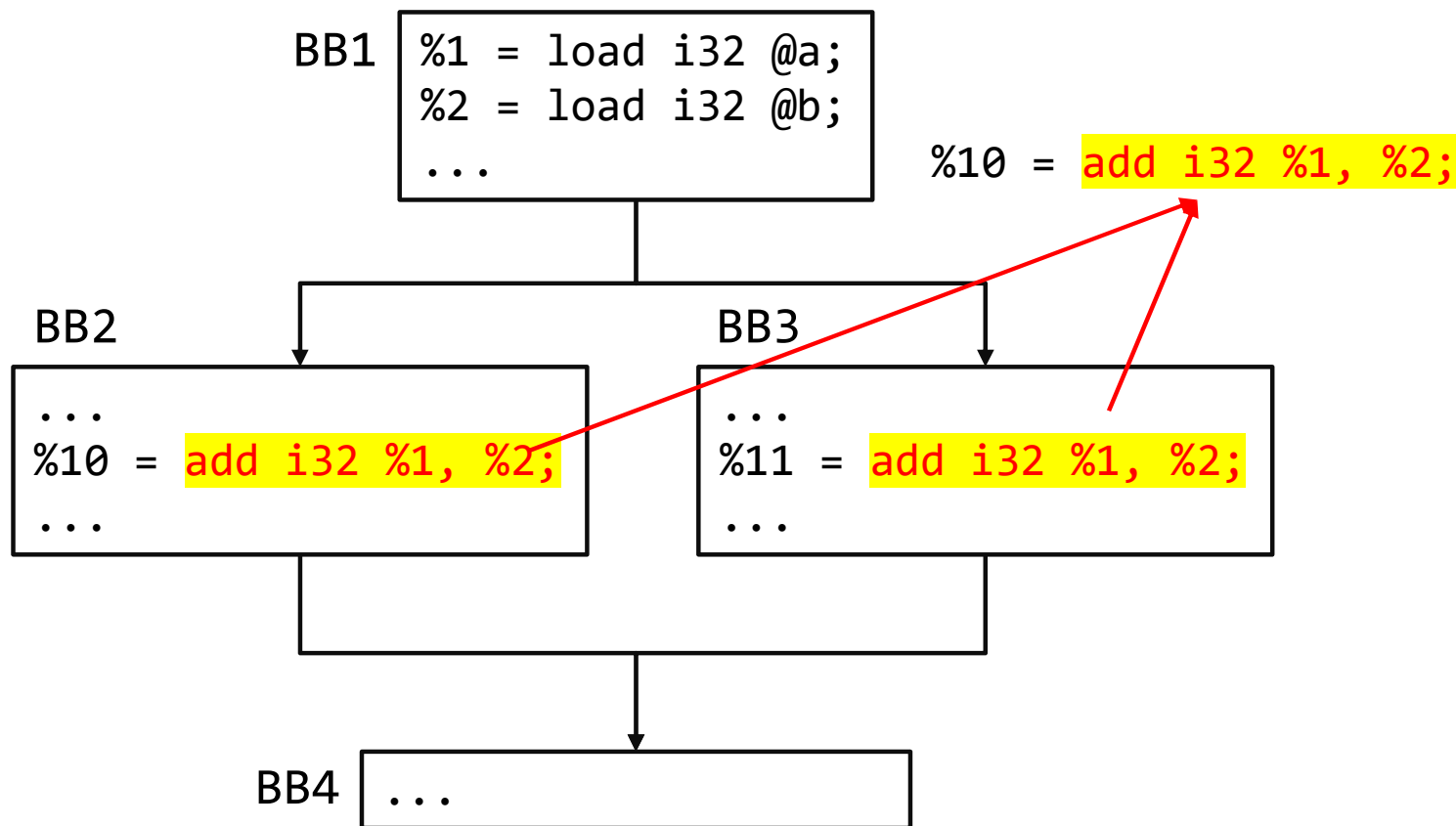


- 正向遍历控制流图

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

# 代码提升：繁忙表达式分析

- 多个代码块中都存在的公共表达式，其操作数被重新赋值
- 分析方法：后序数据流分析
- 不能提升代码运行性能，可以优化代码体积



# 数据流分析方法小节

- 可达性分析
- 活跃变量分析
- 可用表达式分析
- 繁忙表达式分析

$$\boxed{\phantom{x}} [n] = ( \boxed{\phantom{x}} [n] - \text{KILL}[n] ) \cup \text{GEN}[n]$$

$$\boxed{\phantom{x}} [n] = \boxed{\phantom{x}} \boxed{\phantom{x}} [n']$$
$$n' \in \boxed{\phantom{x}} (n)$$

---

$\boxed{\phantom{x}}$	= IN or OUT	$\boxed{\phantom{x}}$	= U (may) or n (must)
$\boxed{\phantom{x}}$		$\boxed{\phantom{x}}$	= predecessors or successors

# 大纲

- 一、冗余代码优化
- 二、循环优化
- 三、过程间优化

# 循环优化

- 循环不变代码
  - 同一个表达式在循环体内被多次执行
- 归纳变量优化
  - 数组下标计算
- 条件语句优化
- 其它优化

# 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<int> = ...;  
for i in 1..100{  
    let t = (a + b)*i;  
    s.push(t);  
}
```

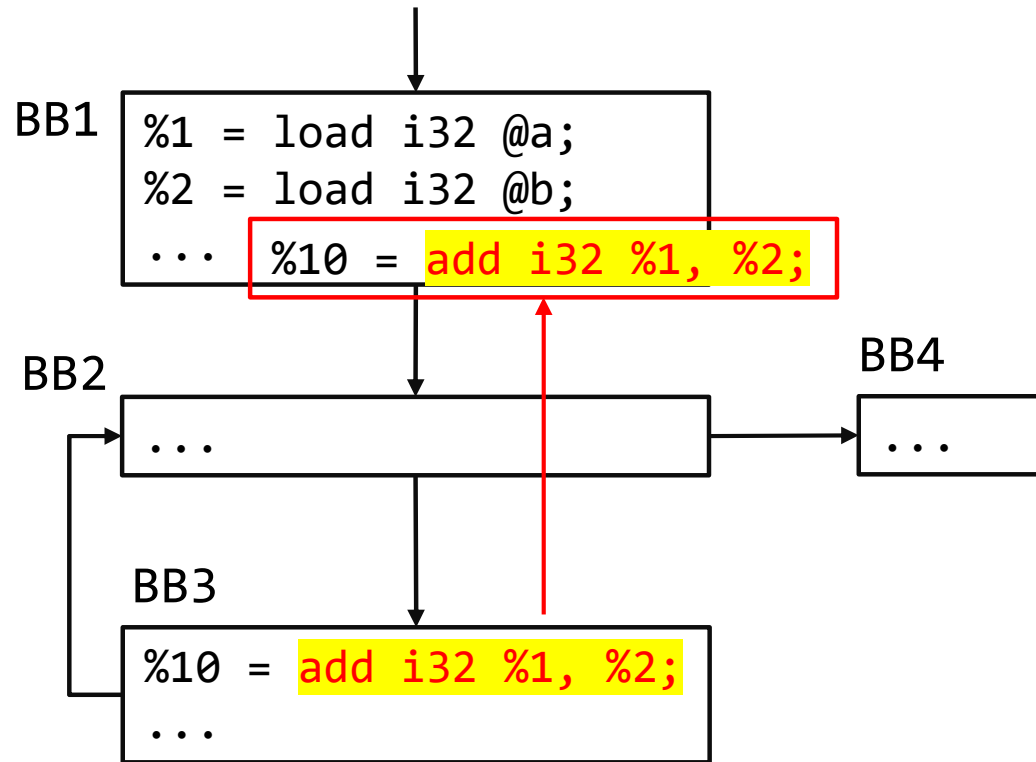
```
let a = ...;  
let b = ...;  
let s:list<int> = ...;  
for i in 1..s.len(){  
    let t = (a + b)*i;  
    s[i] = t;  
}
```

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

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



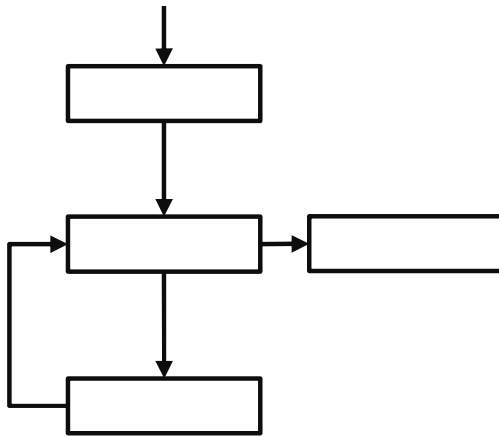
# 循环不变代码



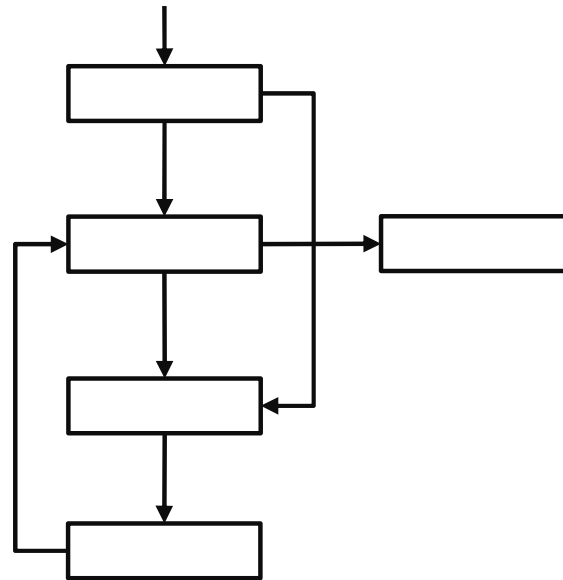
- 检测循环不变代码
  - 操作数定义自循环外部
  - 如何检测循环?
- 前移到循环外部
  - 支配节点

# 自然循环natural loop

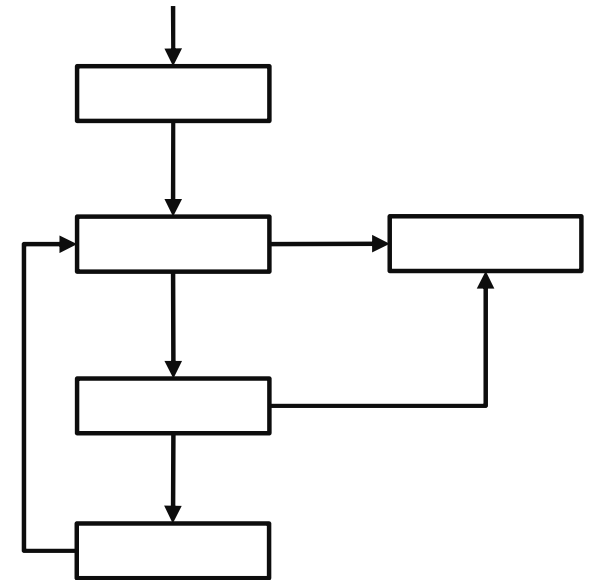
- 一个循环是可规约环的条件：
  - 有唯一的入口
  - 返回入口节点的回边
    - $B \rightarrow A$ 为回边的条件：A支配B
- 一般正常的控制流语句形成的环：while、for、if-else
  - goto语句会造成非自然循环



自然循环

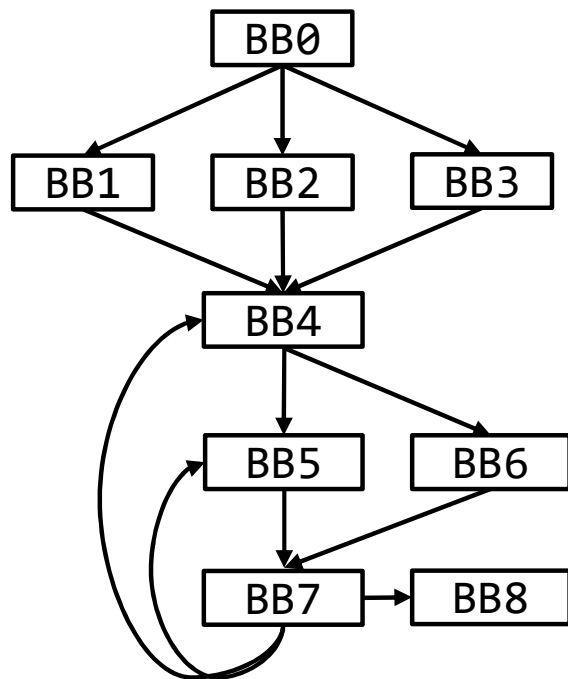


非自然循环



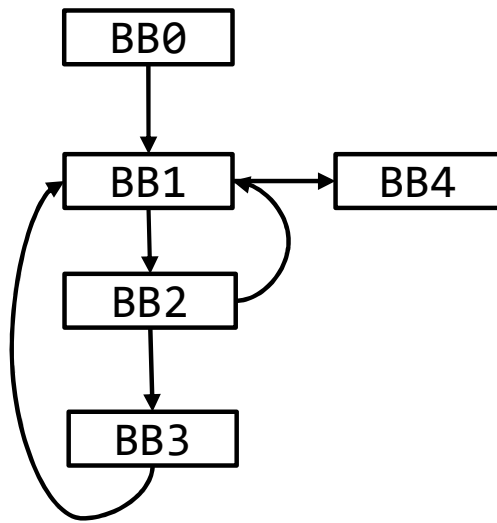
自然循环

下图是否包含非自然循环？

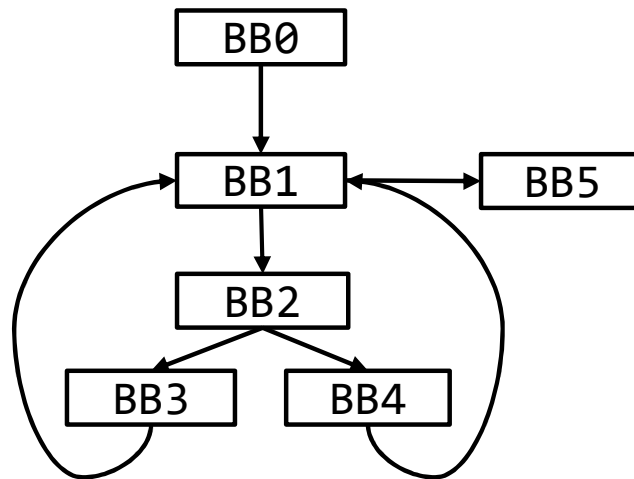


# 自然循环识别

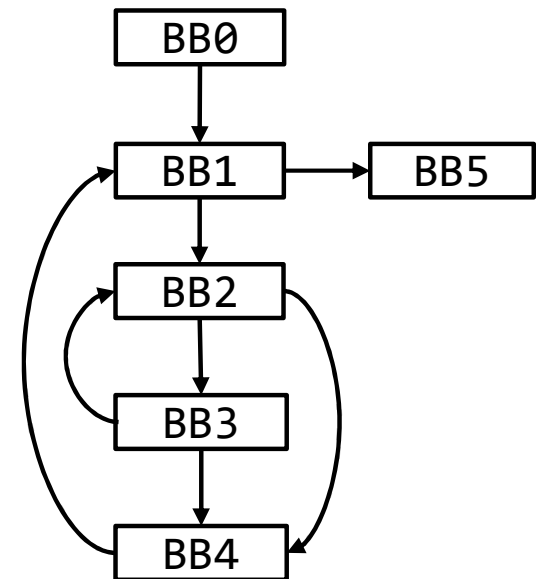
- 每条回边对应一个自然循环



BB2->BB1: 1-2  
BB3->BB1: 1-2-3



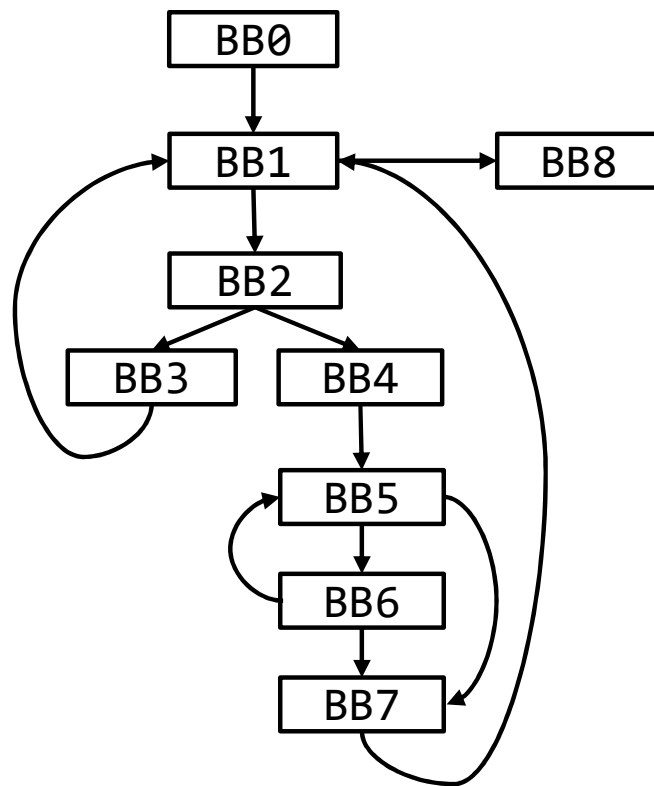
BB3->BB1: 1-2-3  
BB4->BB1: 1-2-4



BB3->BB2: 2-3  
BB4->BB1: 1-2-3-4

# 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



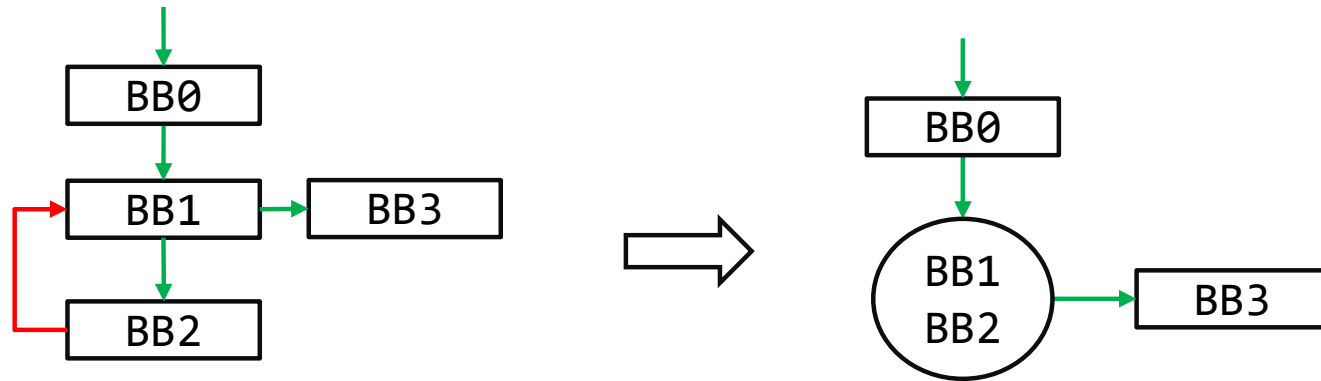
BB3 → BB1: 1-2-3



BB6 → BB5: 5-6

BB7 → BB1: 1-2-4-5-6-7

# 可规约控制流图：Reducible CFG

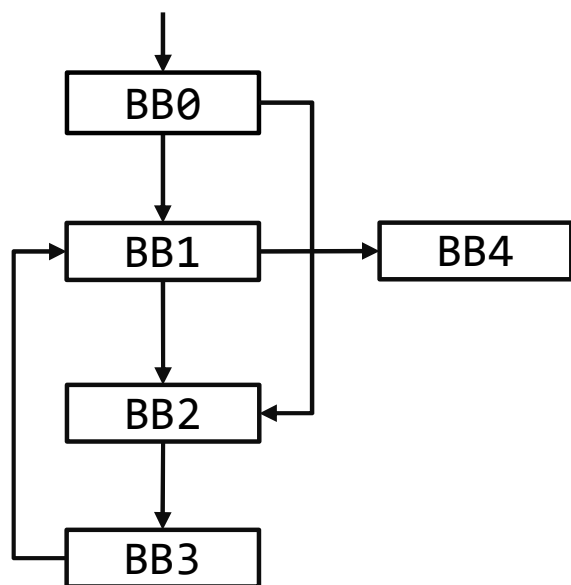
- 可规约CFG的边可以分为前进边和回边两个不交集
- 可以缩环



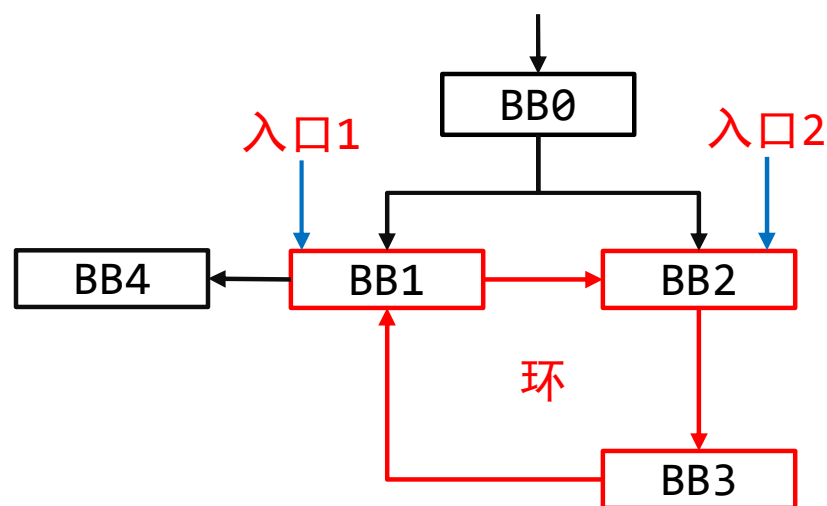
入边：   
出边： 

# 不可规约控制流图

- 难以确定回边
  - BB3 -> BB1? 但BB1不支配BB3



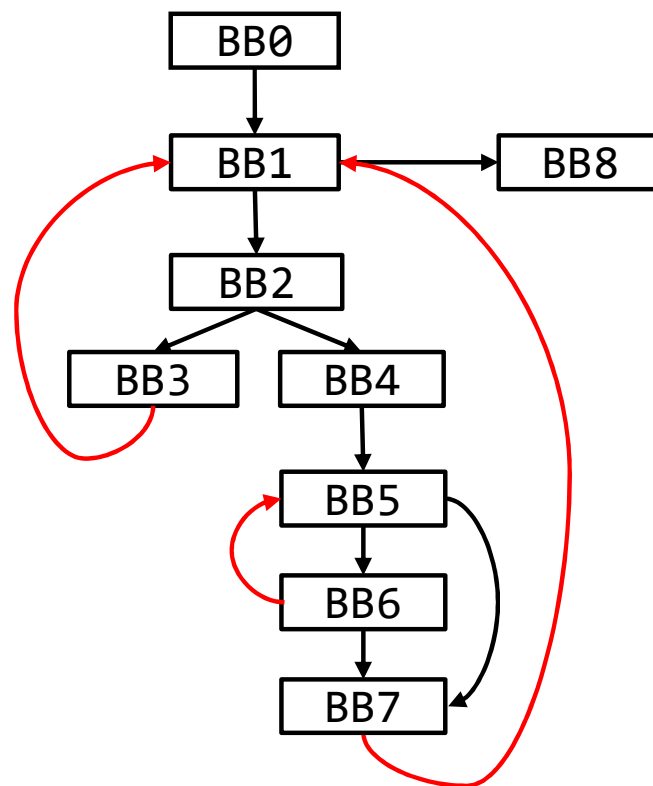
非自然循环



同态图  
isomorphic

# 自然循环检测算法

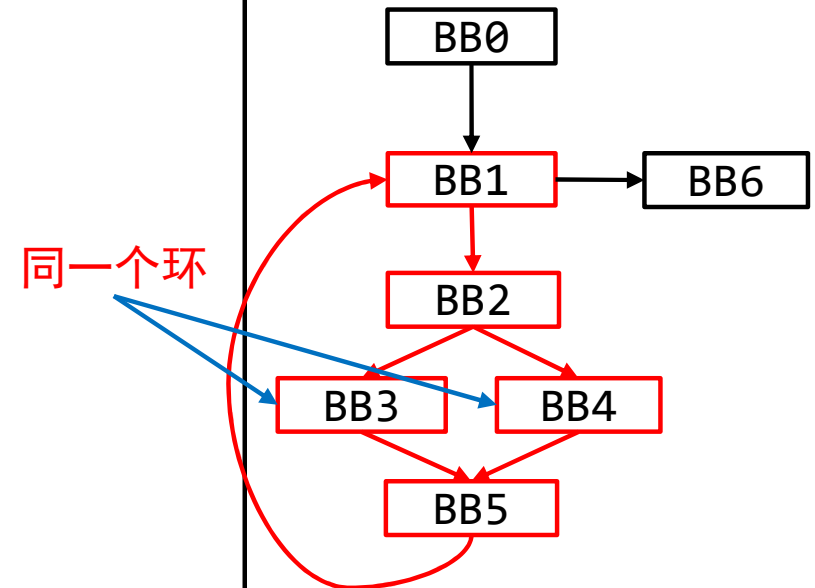
- 基本思路：
  - 1) 遍历CFG=>支配关系矩阵M1
  - 2) 比对图邻接表M2=>检测回边
  - 3) 识别每一条回边对应的环
- DFS
  - 如果栈中已存在节点=>回边
  - 若干栈顶元素组成环
  - 相同回边的环需要merge





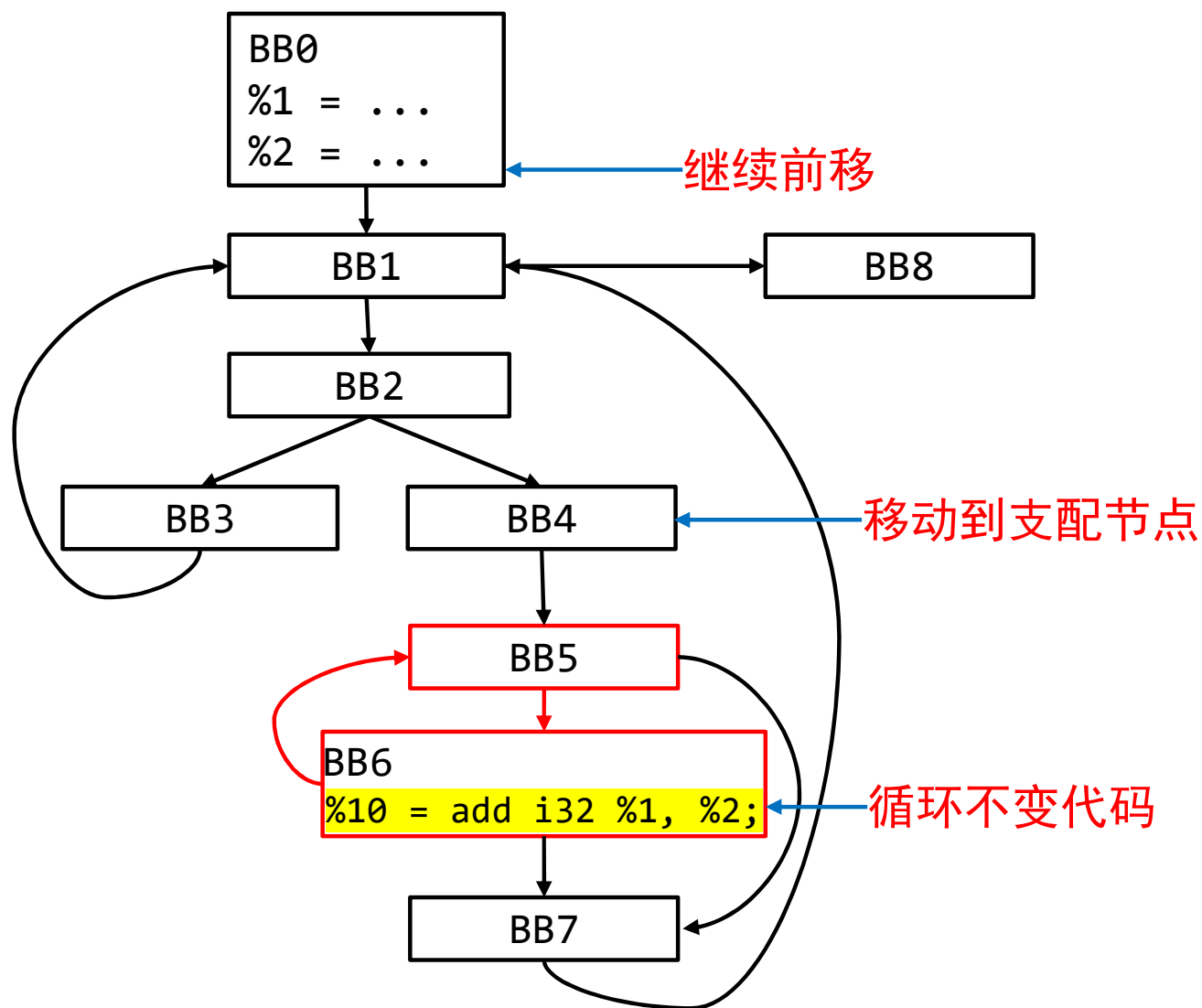
# DFS检测环

```
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)  
}
```



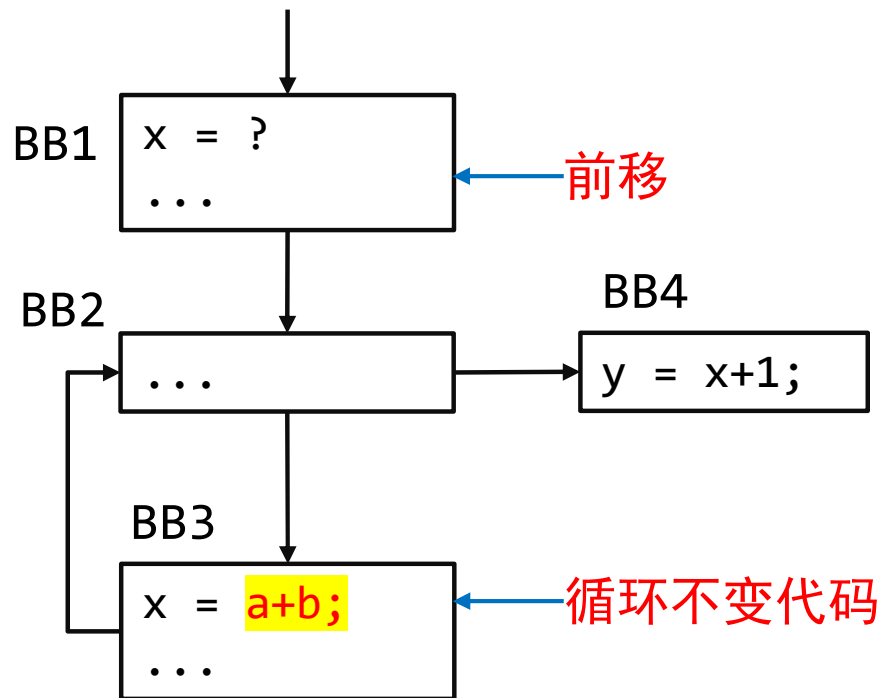
# 前移位置

- 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



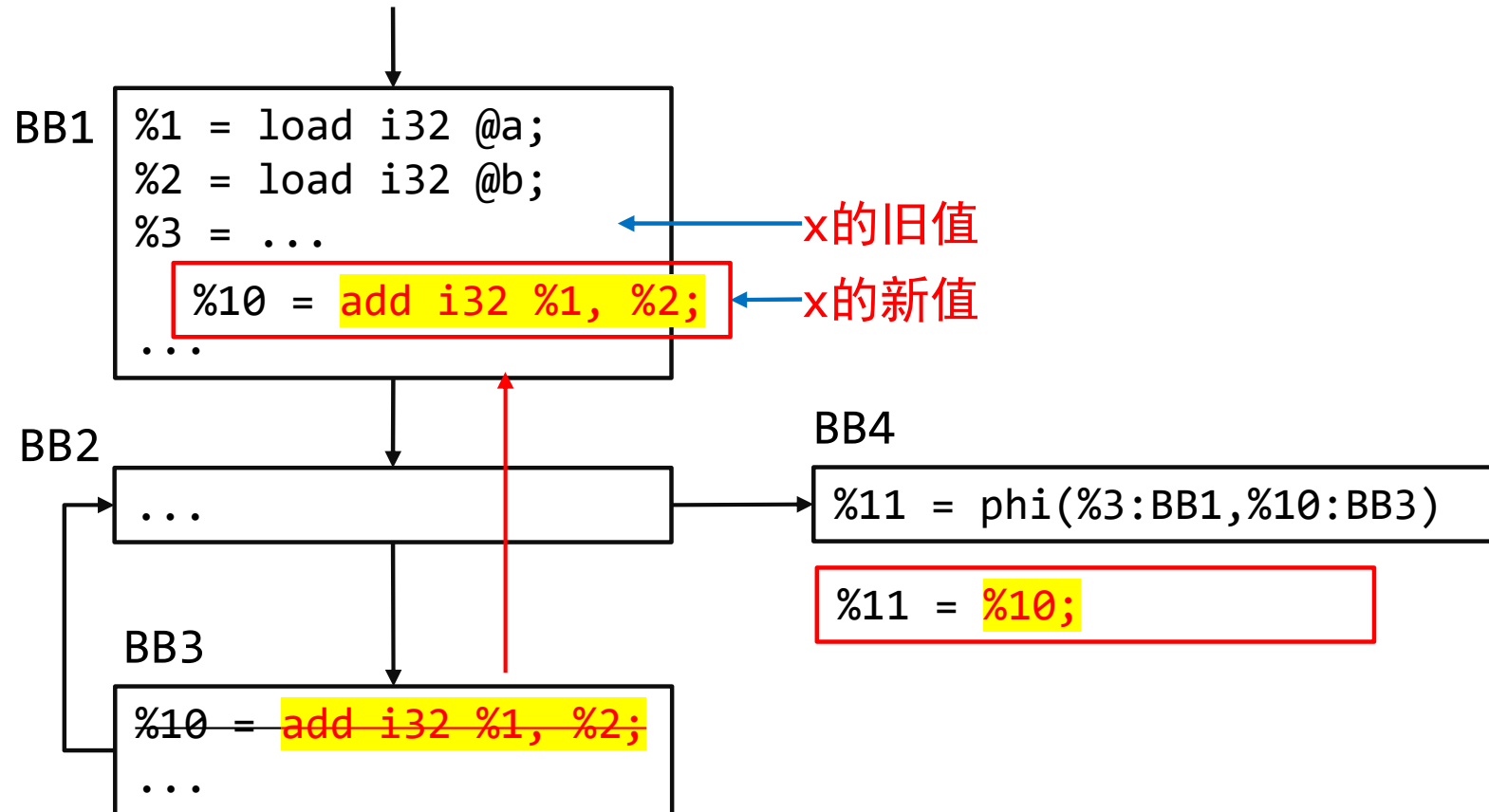
# 可能会有side effect

- 如未进入循环，会错误修改x的值



# 消除side effect

- 如未进入循环



# 归纳变量

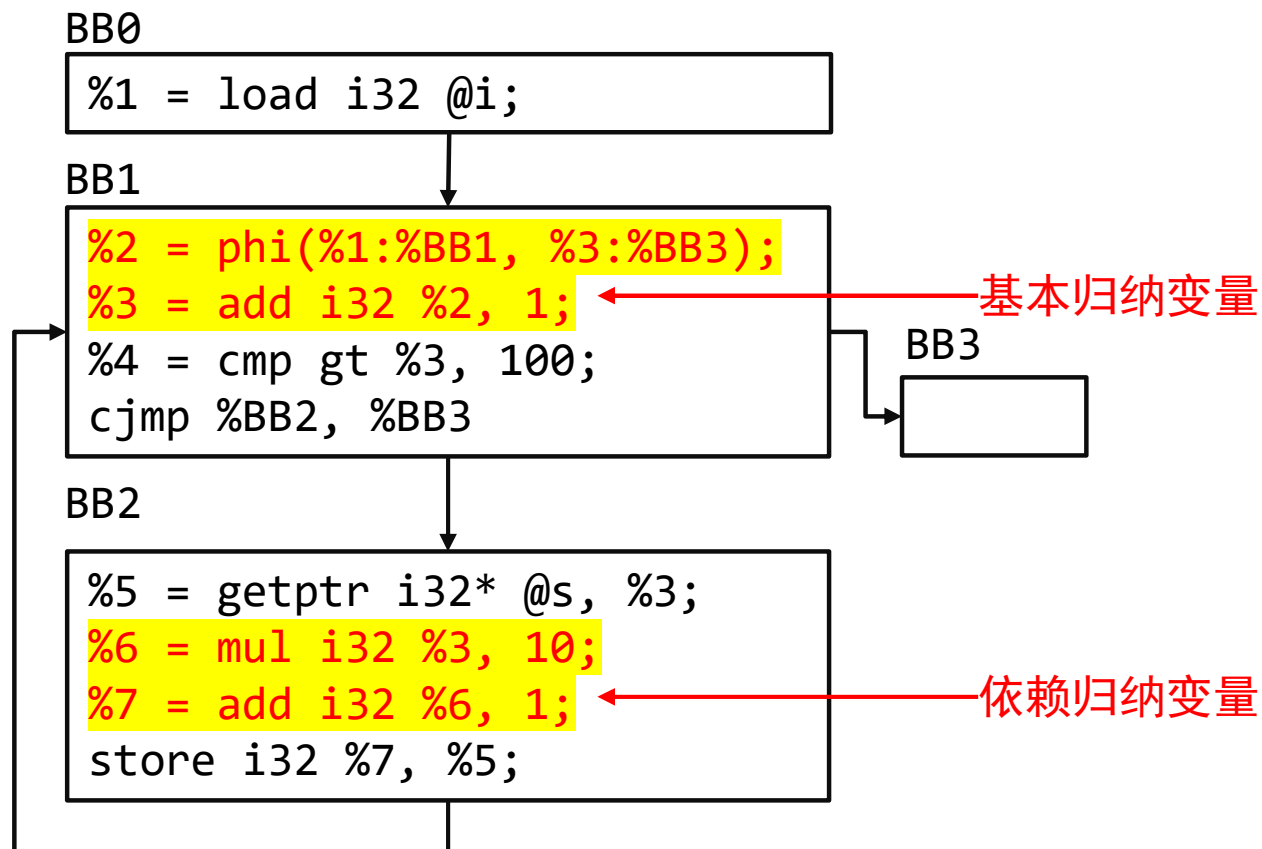
- 变量 $x$ 的值每次增加固定值，则称 $x$ 为归纳变量
  - 基本归纳变量 $x$
  - 依赖归纳变量 $y = ax + b$ ,  $a$ 和 $b$ 为常量

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



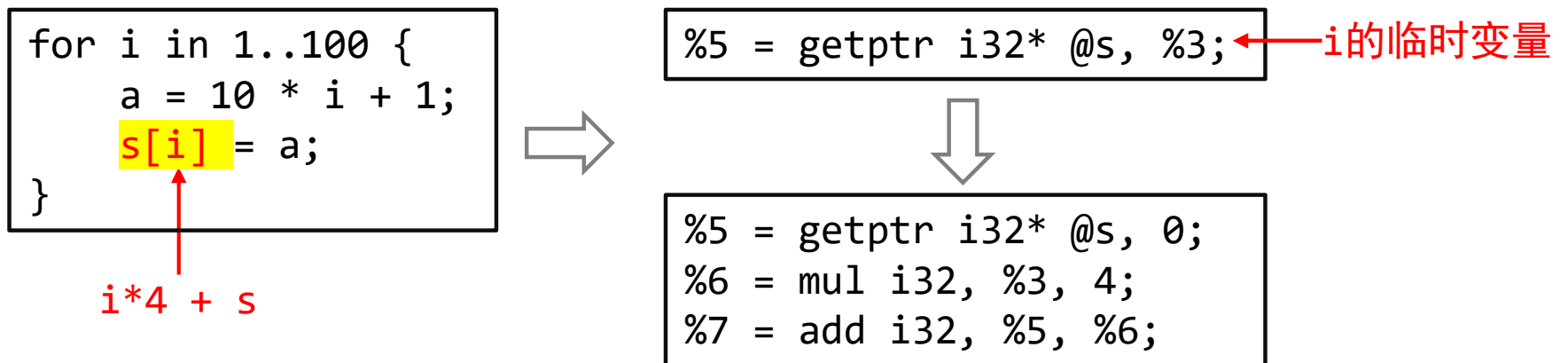
```
t1 = 1;  
for i in 1..100 {  
    t1 = t1 + 10;  
    s[i] = t1;  
}
```

# 基于IR识别归纳变量



# 数组下标中的归纳变量

- IR中不一定可见



# 分支预测的代价

- Loop unswitching: 外提（减少）循环内条件判断
  - 减少分支跳转次数

```
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
```



# 如何测量运行开销

- 一般的CPU都提供Time Stamp Counter (TSC)功能
  - 记录上次reset以来的CPU cycles
  - X86-64读取TSC的指令：RDTSC/RDTSCP
    - 结果保存在EDX:EAX

```
static inline uint64_t rdtscp(void)
{
    uint64_t tsc;
    __asm__ __volatile__(
        "rdtscp;"
        "shl $32, %%rdx;"
        "or %%rdx, %%rax"
        : "=a"(tsc)
        :
        : "%rcx", "%rdx");

    return tsc;
}
```

# 其它循环优化

- Loop unroll: 将循环体复制多遍
- Loop fusion: 将多个循环体合成一个
- Loop distribution: 将一个循环体拆成多个

# 大纲

- 一、冗余代码优化
- 二、循环优化
- 三、过程间优化

# 函数优化策略

常用优化技术

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

```
fn foo(x:int) -> int {  
    foo(x, x+1);  
    foo(x, 1);  
    foo(0, x);  
}
```

# 过程间优化

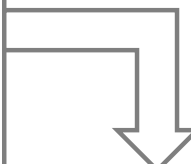
- 一般在link time进行，可以前移到中间代码环节
- 主要优化方法：
  - 函数内联
  - 尾递归优化

# 内联

- 在函数调用处使用函数体替换
- 提升运行效率
- Partial evaluation/Compile-time execution
  - 常量传播=>优化

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

```
fn foo(x:int) -> int {  
  let a = foo(x, x+1);  
  let b = foo(x, 1);  
  let c = foo(0, x);  
}
```



```
fn foo(x:int) -> int {  
  let a = {if (x == 0) 1 else x};  
  let b = {if (x == 0) 1 else 0};  
  let c = x;  
}
```

# 内联问题建模

- 内联的好处：利于优化，提升代码运行效率
- 内联是有开销的，代码复制可能会增大代码体积
- 给定bugget上限，选取最优的内联函数组合
- 背包问题（Knapsack problem）

# 0-1背包问题

- 假设有 $n$ 个选项 $x_1 \dots x_n$
- 选择 $x_i$ 的开销是 $w_i$ ，收益是 $v_i$
- 给定开销上限，如何选择可以获得最大收益？

$$\max \sum_{i=1}^n v_i x_i$$

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



# 递归函数

- 递归函数内联问题（内联 vs 宏展开）
- 尾递归函数可以优化
  - 特点：return前的最后一条语句调用自己

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



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

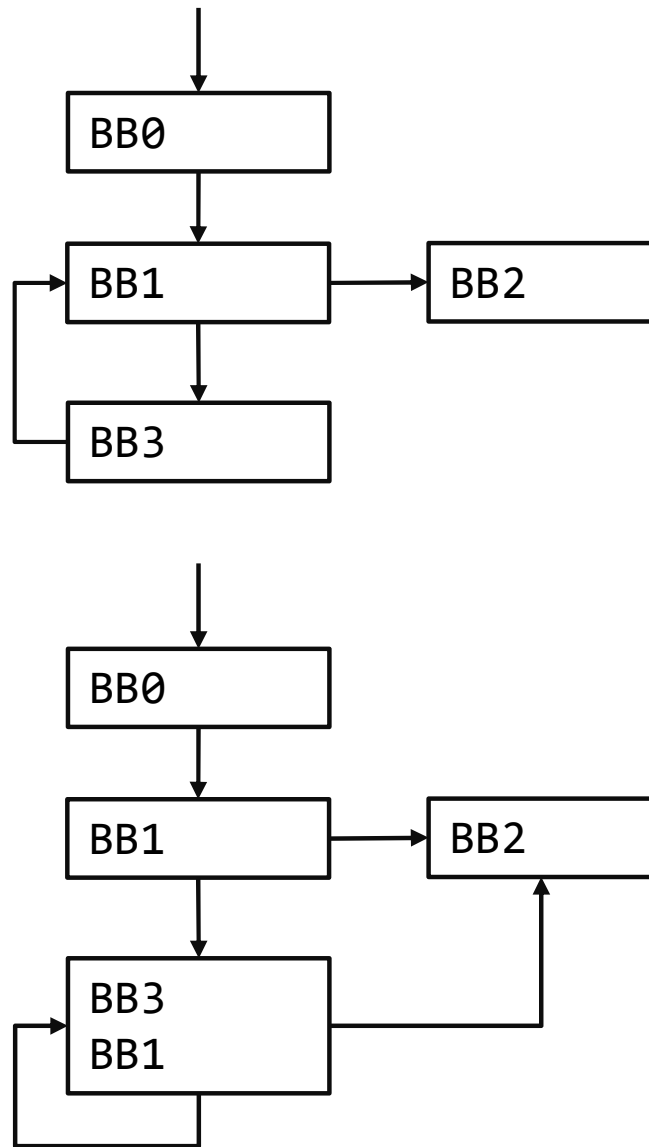


# 尾递归消除

```
fn factorial(n:int, r:int) {  
  %BB0:  
    stackalloc i32, @n;  
    stackalloc i32, @r;  
    store %-1, @n;  
    store %-2, @r;  
    %1 = load i32 @n;  
    %2 = load @r;  
    %3 = cmp eg %1, 0;  
    cjmp %2 %BB1, %BB2;  
  %BB1:  
    ret %2;  
  %BB2:  
    %4 = sub i32 %1, -1;  
    %5 = mul i32 %1, %2;  
    %6 = call factorial(%4, %5);  
    ret %6;  
}
```

```
fn factorial(n:int, r:int) {  
  %BB0:  
    stackalloc i32, @n;  
    stackalloc i32, @r;  
    store %-1, @n;  
    store %-2, @r;  
    jmp %BB1;  
  %BB1  
    %1 = load i32 @n;  
    %2 = load @r;  
    %3 = cmp eg %1, 0;  
    cjmp %2 %BB2, %BB3;  
  %BB2:  
    ret %2;  
  %BB3:  
    %4 = sub i32 %1, -1;  
    %5 = mul i32 %1, %2;  
    store %4 @n;  
    store %5 @r;  
    cjmp %BB1;  
}
```

# 尾递归优化



```
fn factorial(n:int, r:int) {
%BB0:
    stackalloc i32, @n;
    stackalloc i32, @r;
    store %-1, @n;
    store %-2, @r;
%BB1:
    %1 = load i32 @n
    %2 = load @r
    %3 = cmp eg %1, 0;
    cjmp %2 %BB2, %BB3
%BB2:
    ret %2;
%BB3:
    %4 = sub i32 %1, -1;
    %5 = mul i32 %1, %2;
    %3 = cmp eg %4, 0;
    cjmp %2 %BB2, %BB3
}
```

# Sibling Call优化

- Sibling call
  - 调用函数A和被调用函数B签名相同
  - tail call

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

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

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

# 总结

- 冗余代码优化
  - 无用代码、冗余代码
  - 数据流分析
- 循环优化
  - 循环不变代码
    - 循环检测
  - 归纳变量
  - Loop unswitch/unroll/fusion/distribution
- 过程间优化
  - 函数内联
  - 尾递归优化