中山大學
SUN YAT-SEN UNIVERSITY

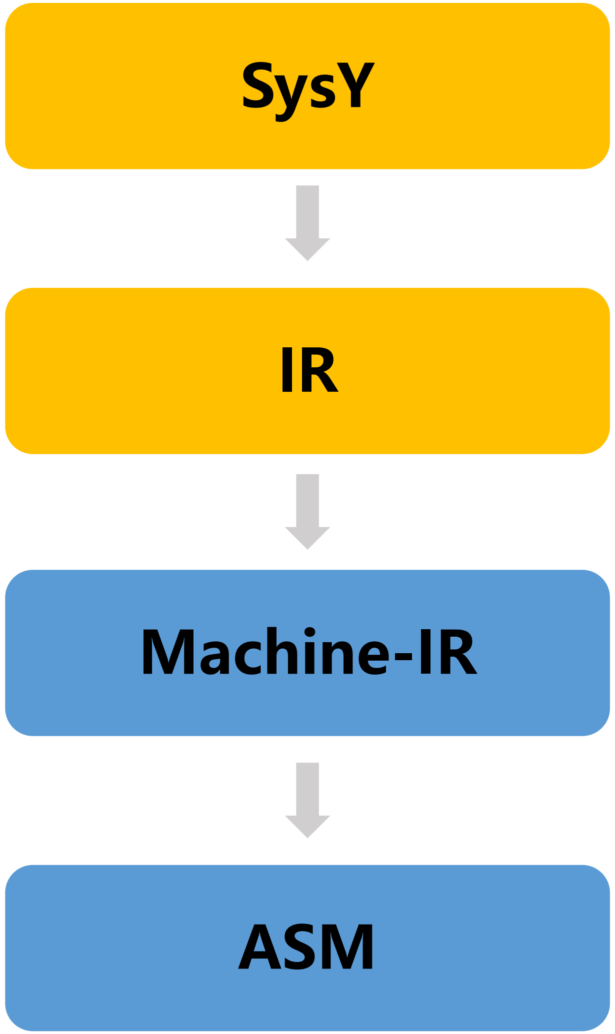
现代编译器构建流程
以Yat-CC为例

2024/6/20

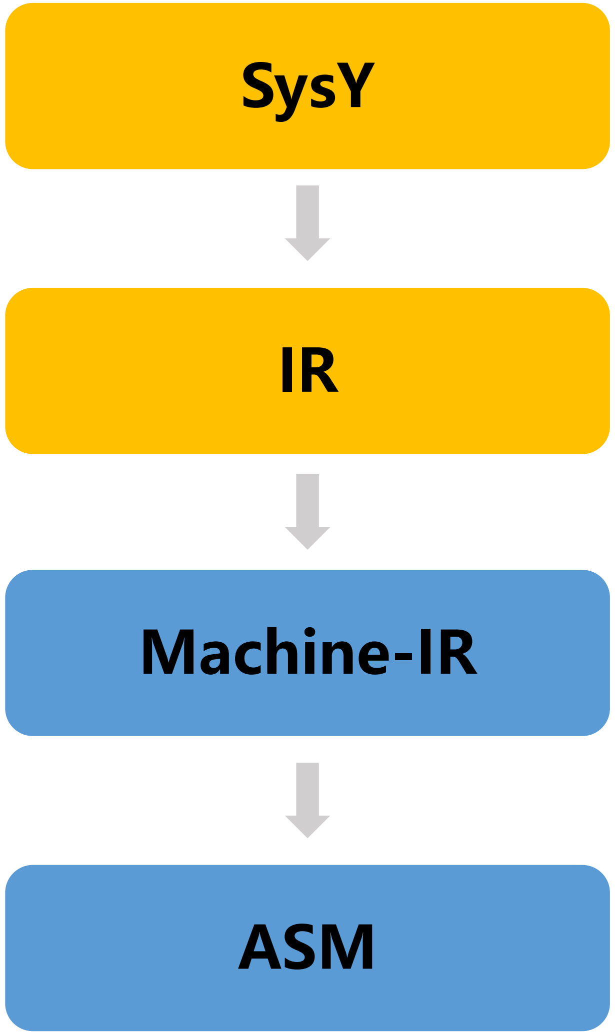
助教：黄瀚、潘文轩、郑中淳

目录

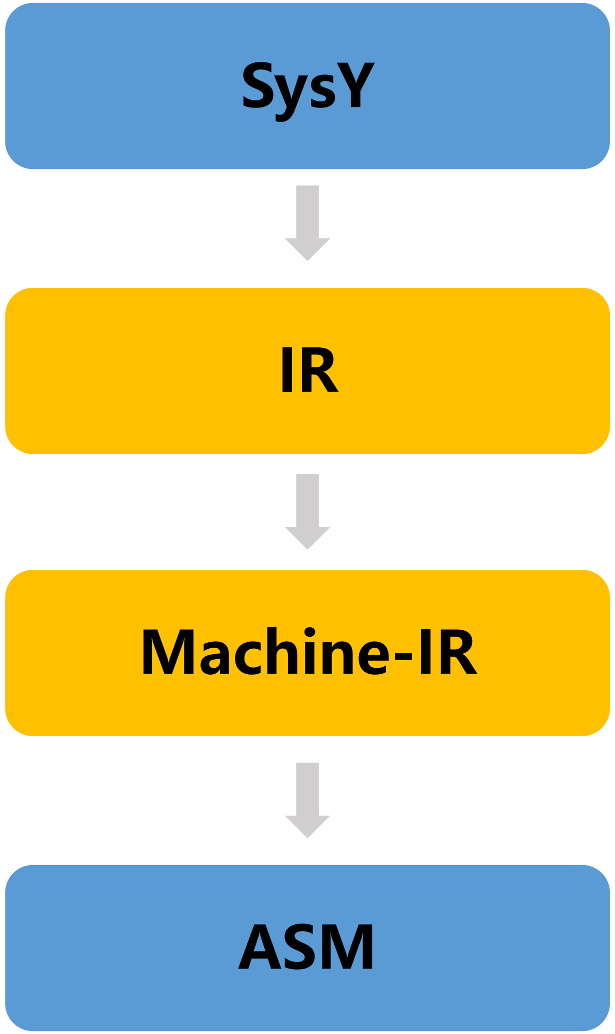
- 前端
  - 使用ANTLR，将SysY语言分析成IR
  - 使用的IR与LLVM等价，可以直接使用llvm后端执行



- 前端
  - Antlr能够自动生成语法分析器和词法分析器
  - 可以直接完成从源代码到IR的转换
  - 首先将源代码变为抽象语法树
  - 使用Vistor或者Listener遍历抽象语法树生成IR



- 中端
  - SSA转换 (Mem2Reg)
  - 死代码消除（Dead Code Elimination)
  - 公共子表达式消除（Common Subexpression Elimination)
  - 常量传播（Constant Propagation)
  - 常量折叠 (Constant Folding)
  - 函数内联（Function Inline)
  - 指令重排（Reassociation)
  - 循环优化（Loop Optimization)
  - 控制流简化（CFG Simplification)



- 优化介绍
  - **Mem2Reg**
  - LLVM O0生成的代码不是严格的SSA形式，alloca的变量会被多次赋值
  - Mem2Reg可以消除这部分局部变量，使得IR的SSA形式更加严格
  - Mem2Reg无法处理局部数组
  - 分为两个主要阶段，插入Phi指令与变量重命名

```
define i32 @max(i32 %a, i32 %b) #0 {
entry:
  %retval = alloca i32, align 4
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %1 = load i32, i32* %b.addr, align 4
  %cmp = icmp sgt i32 %0, %1
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  %2 = load i32, i32* %a.addr, align 4
  store i32 %2, i32* %retval, align 4
  br label %return

if.else:                                          ; preds = %entry
  %3 = load i32, i32* %b.addr, align 4
  store i32 %3, i32* %retval, align 4
  br label %return

return:                                           ; preds = %if.else, %if.then
  %4 = load i32, i32* %retval, align 4
  ret i32 %4
}
```

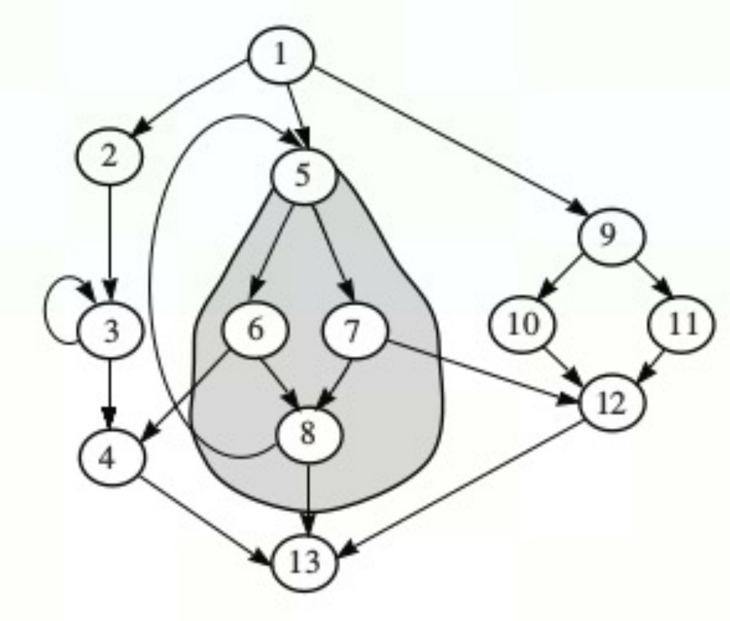
```
define i32 @max(i32 %a, i32 %b) {
entry:
  %0 = icmp sgt i32 %a, %b
  br i1 %0, label %if.then, label %if.else

if.then:
  br label %return

if.else:
  br label %return

return:
  %retval = phi i32 [%a, %if.then], [%b, %if.else]
  ret i32 %retval
}
```

- 优化介绍
  - **插入Phi指令**
  - 每个Alloca的变量，可能会有多次Store，每次Store视为对变量的一次新定义
  - 只需要在支配边界插入Phi指令（在该块，可能会有一个变量的多个定义）
  - 直观解释支配边界就是所有最近不能被 x 严格支配的节点的集合
  - 该图中，节点5是{5、6、7、8}的支配节点，{5、4、13、12}是5的支配边界
  - 有些变量并不会经过这些支配边界（liveInBlock），不需要插入Phi指令

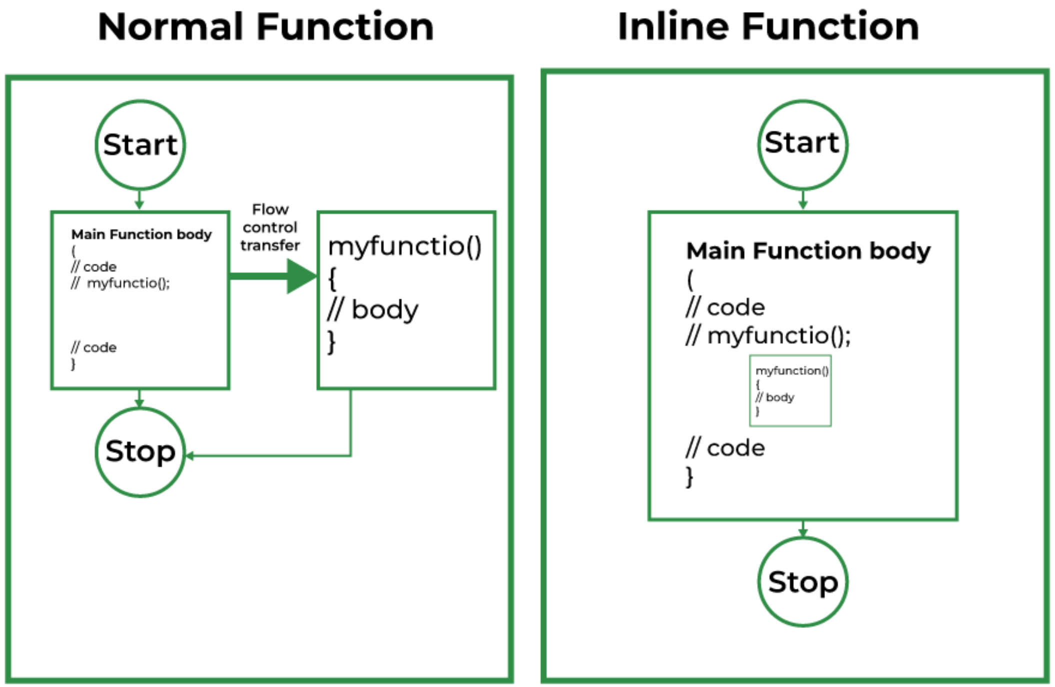


- 优化介绍
  - **变量重命名**
  - 原本使用的操作数，是load得到的值，现在需要将load指令替换为store指令存储的值，或者是phi指令得到的值。

Algorithm 3.3: Renaming algorithm for second phase of SSA construction

```
▷ rename variable definitions and uses to have one definition per variable name
foreach v : Variable do
    v.reachingDef ← ⊥
foreach BB: basic Block in depth-first search preorder traversal of the dom. tree do
    foreach i : instruction in linear code sequence of BB do
        foreach v : variable used by non-φ-function i do
            updateReachingDef(v, i)
            replace this use of v by v.reachingDef in i
        foreach v : variable defined by i (may be a φ-function) do
            updateReachingDef(v, i)
            create fresh variable v′
            replace this definition of v by v′ in i
            v′.reachingDef ← v.reachingDef
            v.reachingDef ← v′
    foreach φ: φ-function in a successor of BB do
        foreach v : variable used by φ do
            updateReachingDef(v, φ)
            replace this use of v by v.reachingDef in φ
```

- 优化介绍
  - **函数内联**
  - 如果在函数调用图中，一个函数不在任何环中，则将该函数进行内联（拓扑排序）
  - 对内联函数的每个调用，复制一次函数，将函数参数替换为call指令的参数，使用复制的函数替换call指令
  - 尽可能地进行函数内联，以获得更多的优化机会，但可能导致代码体积变大，损失指令缓存的时间

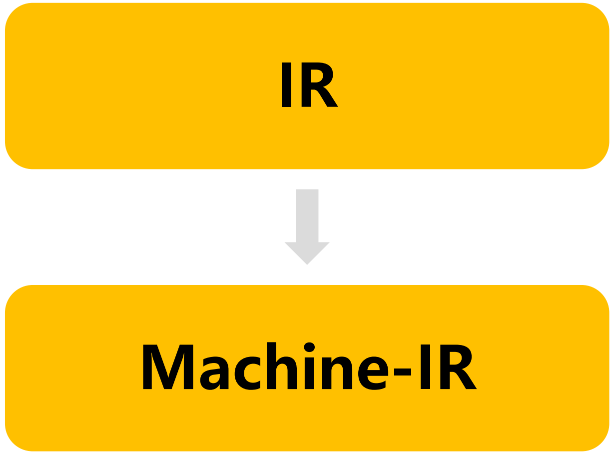


目录

- **Machine IR：**

  一种介于LLVM IR和汇编中间的IR

```
.L3_28:
    movw vr50, :lower16:tape
    movt vr50, :upper16:tape
    add vr51, vr10, #0
    ldr vr52, [vr50, vr51, lsl #2]
    movw vr53, :lower16:output_length
    movt vr53, :upper16:output_length
    ldr vr54, [vr53]
    movw vr55, :lower16:output
    movt vr55, :upper16:output
    add vr56, vr54, #0
    str vr52, [vr55, vr56, lsl #2]
    movw vr57, :lower16:output_length
    movt vr57, :upper16:output_length
    ldr vr58, [vr57]
    add vr59, vr58, #1
    movw vr60, :lower16:output_length
    movt vr60, :upper16:output_length
    str vr59, [vr60]
    mov vr100, vr63
    b .L3_35
.L3_29:
```

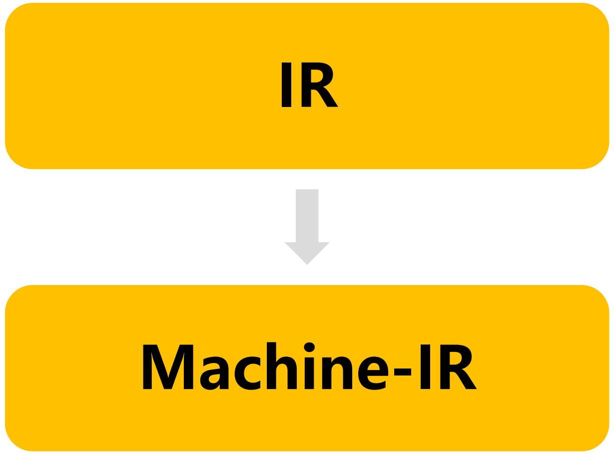


中山大學 SUN YAT-SEN UNIVERSITY

- **Machine IR：**

  Machine IR的两个特点：

1. 拥有与汇编相似的形式（除寄存器分配）
2. 更贴近底层：优化更加直观简单



目录

- **Instruction Selection**

类似实验三，对每一种语句从LLVM IR变成Machine IR

A ▾ Options ▾ Filters ▾ Control Flow Graph

```
@mod = dso_local local_unnamed_addr constant i32 998244353, align
@maxlen = dso_local local_unnamed_addr constant i32 10005, align 4
@.str = private unnamed_addr constant [3 x i8] c"%d\00", align 1
@d = dso_local local_unnamed_addr global i32 0, align 4

define dso_local noundef signext i32 @main() local_unnamed_addr {
entry:
  %x = alloca i32, align 4
  call void @llvm.lifetime.start.p0(i64 4, ptr nonnull %x) #4
  %call = call signext i32 (ptr, ...) @__isoc99_scanf(ptr noundef
  %0 = load i32, ptr %x, align 4
  %div = sdiv i32 %0, 20
  store i32 %div, ptr %x, align 4
  %call1 = call signext i32 (ptr, ...) @printf(ptr noundef nonnull
  call void @llvm.lifetime.end.p0(i64 4, ptr nonnull %x) #4
  ret i32 0
}

declare void @llvm.lifetime.start.p0(i64 immarg, ptr nocapture) #1

declare noundef signext i32 @__isoc99_scanf(ptr nocapture noundef

declare noundef signext i32 @printf(ptr nocapture noundef readonly

declare void @llvm.lifetime.end.p0(i64 immarg, ptr nocapture) #1

declare void @llvm.dbg.assign(metadata, metadata, metadata, metada

!40 = distinct !DIAssignID()
!49 = distinct !DIAssignID()
```

```
.L3_28:
    movw vr50, :lower16:tape
    movt vr50, :upper16:tape
    add vr51, vr10, #0
    ldr vr52, [vr50, vr51, lsl #2]
    movw vr53, :lower16:output_length
    movt vr53, :upper16:output_length
    ldr vr54, [vr53]
    movw vr55, :lower16:output
    movt vr55, :upper16:output
    add vr56, vr54, #0
    str vr52, [vr55, vr56, lsl #2]
    movw vr57, :lower16:output_length
    movt vr57, :upper16:output_length
    ldr vr58, [vr57]
    add vr59, vr58, #1
    movw vr60, :lower16:output_length
    movt vr60, :upper16:output_length
    str vr59, [vr60]
    mov vr100, vr63
    b .L3_35
.L3_29:
```

- 加减乘除

算数运算直接翻译即可保证正确性

```
%add = add nsw i32 %0, 20
%div  =  sdiv i32 %add, 20
```

```
li        vr0,20
addi      vr1,sp,8
addiw     vr1,vr1,20
divw      vr1,vr1,vr0
```

- **Load/Store**

Load / Store这种指令，对于全局变量以及局部变量有不同的翻译方法：

- **全局变量：** 需要做一个高位/低位的 Load / Store
- **局部变量：** 直接Store即可

```
store i32 %div, ptr %x, align 4
store i32 %div, ptr @d, align 4
```

→

```
auipc   vr1, %pcrel_hi(d)
sw      vr0, %pcrel_lo(.Lpcrel_hi1)(vr1)
sw      vr0, 12(sp)
```

- **GEP**

GEP指令：翻译为一个对%index变量的累加指令

```
%arrayidx9 = getelementptr inbounds [2005 x i32], ptr %a, i64 0, i64 %indvars.iv21
%3 = load i32, ptr %arrayidx9, align 4
```

→

```
lw      vr2, 0(vr0)
mul     vr2, vr2, vr1
sw      vr2, 0(vr0)
addi    vr0, vr0, 4
```

目录

- **除法优化**

实验四中，对于除法指令（除以一个常数）的优化没有乘法指令那么简单

- 乘16可以通过强度削减变成位移，但是除16则不行

```
x /= 16;
```

→

```
%0 = load i32, ptr %x, align 4
%div = sdiv i32 %0, 16
```

- 除法优化

对于这种除法指令的优化，就需要在后端上做

```
li      vr0,16
addi    vr1,sp,8
divw    vr1,vr1,vr0
```

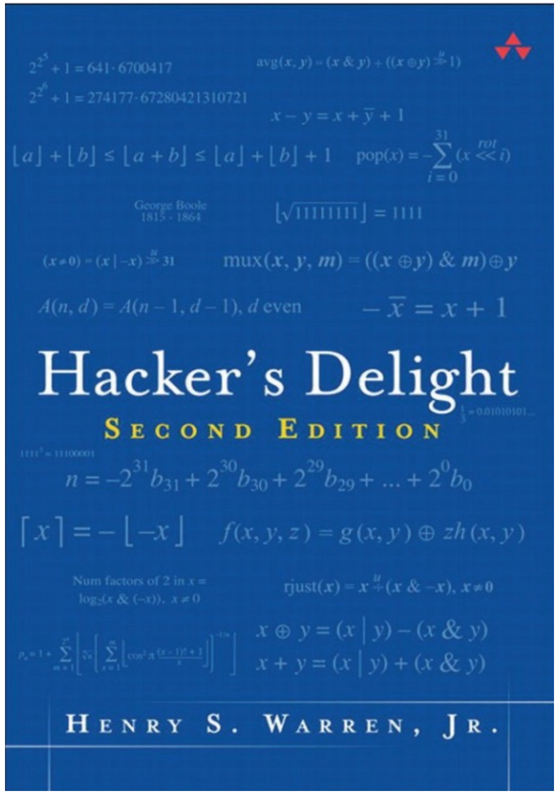
→

```
srai   vr1, vr0, 31
srli   vr1, vr1, 28
add    vr0, vr0, vr1
srai   vr0, vr0, 4
```

- 参考资料

关于除法优化以及其它一些后端在汇编层级的优化，大家可以参考：

《Hacker’s Delight》



Chapter 10. Integer Division By Constants

On many computers, division is very time consuming and is to be avoided when possible. A value of 20 or more elementary *add* times is not uncommon, and the execution time is usually the same large value even when the operands are small. This chapter gives some methods for avoiding the *divide* instruction when the divisor is a constant.

10–1 Signed Division by a Known Power of 2

Apparently, many people have made the mistake of assuming that a *shift right signed* of k positions divides a number by 2^k, using the usual truncating form of division [GLS2]. It’s a little more complicated than that. The code shown below computes $q = n \div 2^k$, for $1 \le k \le 31$ [Hop].

```
shrsi  t,n,k-1       Form the integer
shri   t,t,32-k      2**k - 1 if n < 0, else 0.
add    t,n,t         Add it to n,
shrsi  q,t,k         and shift right (signed).
```

It is branch free. It simplifies to three instructions in the common case of division by 2 ($k = 1$). It does, however, rely on the machine’s being able to shift by a large amount in a short time. The case $k = 31$ does not make too much sense, because the number 2^{31} is not representable in the machine. Nevertheless, the code does produce the correct result in that case (which is $q = -1$ if $n = -2^{31}$ and $q = 0$ for all other n).

To divide by -2^k, the above code can be followed by a *negate* instruction. There does not seem to be any better way to do it.

The more straightforward code for dividing by 2^k is

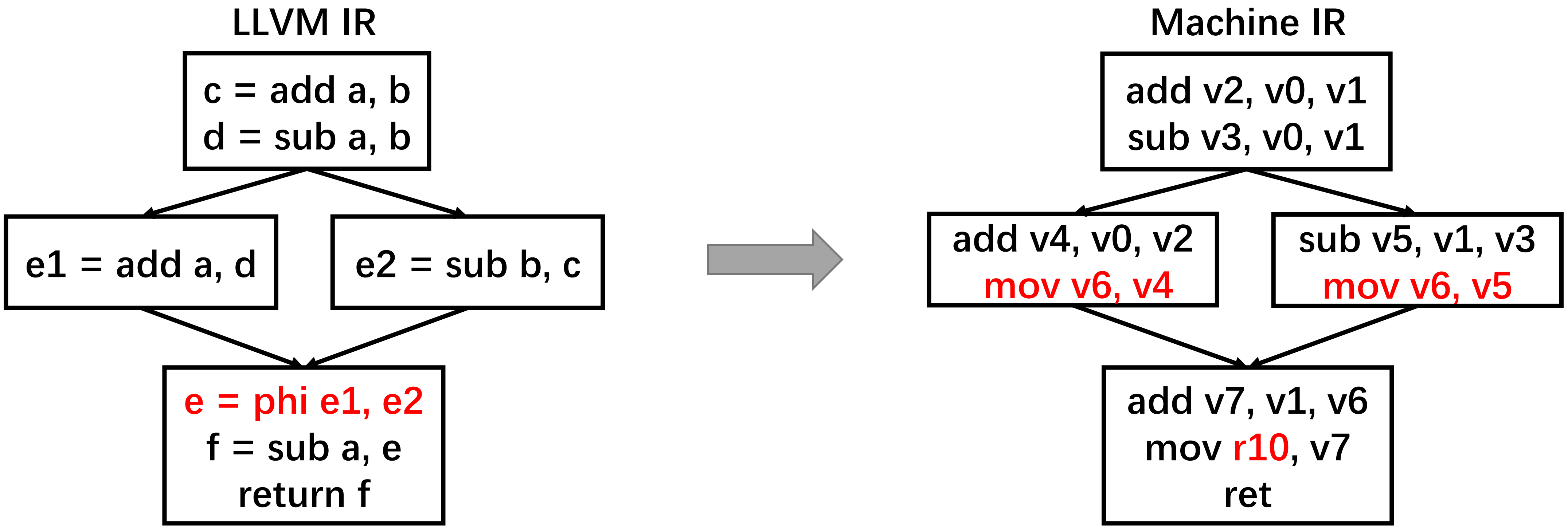
```
       bge    n,label       Branch if n >= 0.
       addi   n,n,2**k-1    Add 2**k - 1 to n,
label  shrsi  n,n,k         and shift right (signed).
```

This would be preferable on a machine with slow shifts and fast branches.

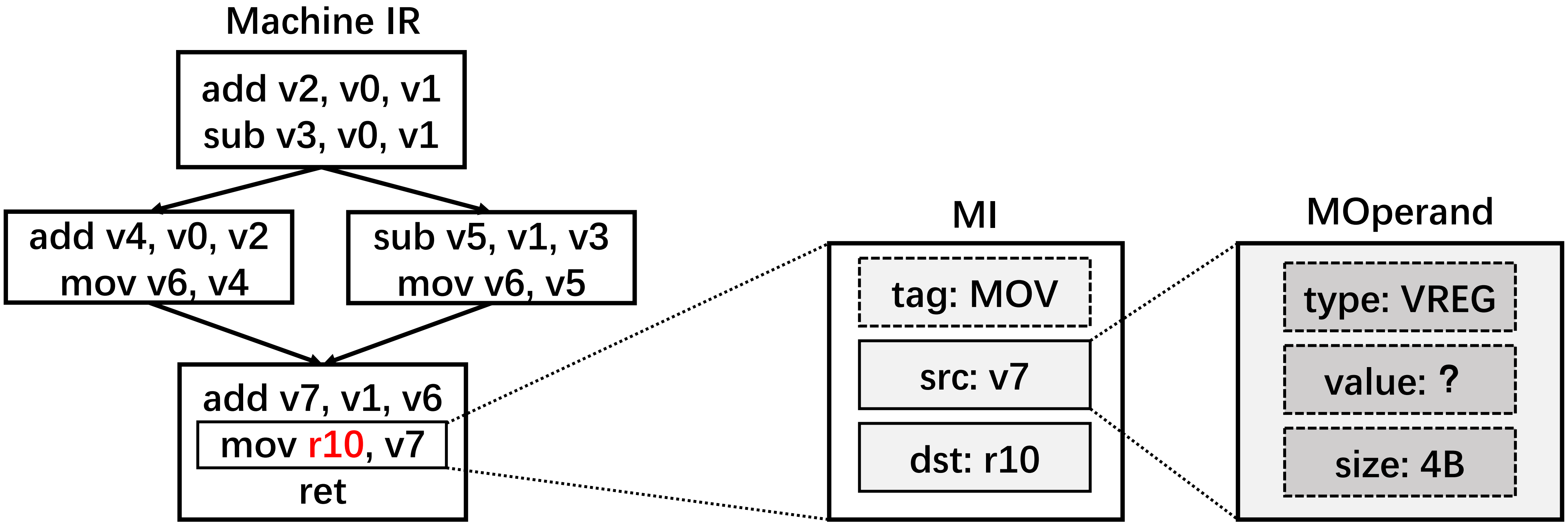
PowerPC has an unusual device for speeding up division by a power of 2 [GGS]. The *shift right signed* instructions set the machine’s carry bit if the number being shifted is negative and one or more 1-bits are shifted out. That machine also has an instruction

目录

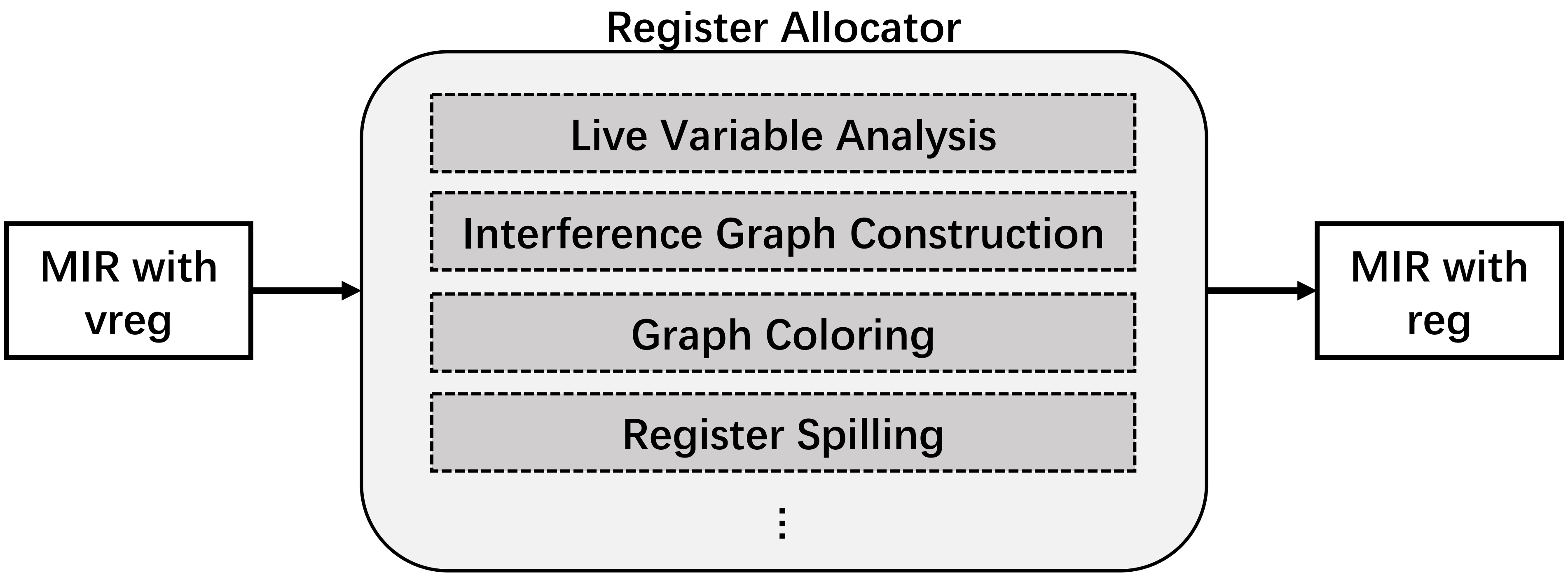
- **背景介绍：Machine IR**
  - 比LLVM IR更低级的IR，用于表示汇编指令
  - 符合特定语言规范 (machine specific IR)



- **背景介绍：Machine IR**
  - Machine IR由MI组成，其中MOperand为MI的操作数



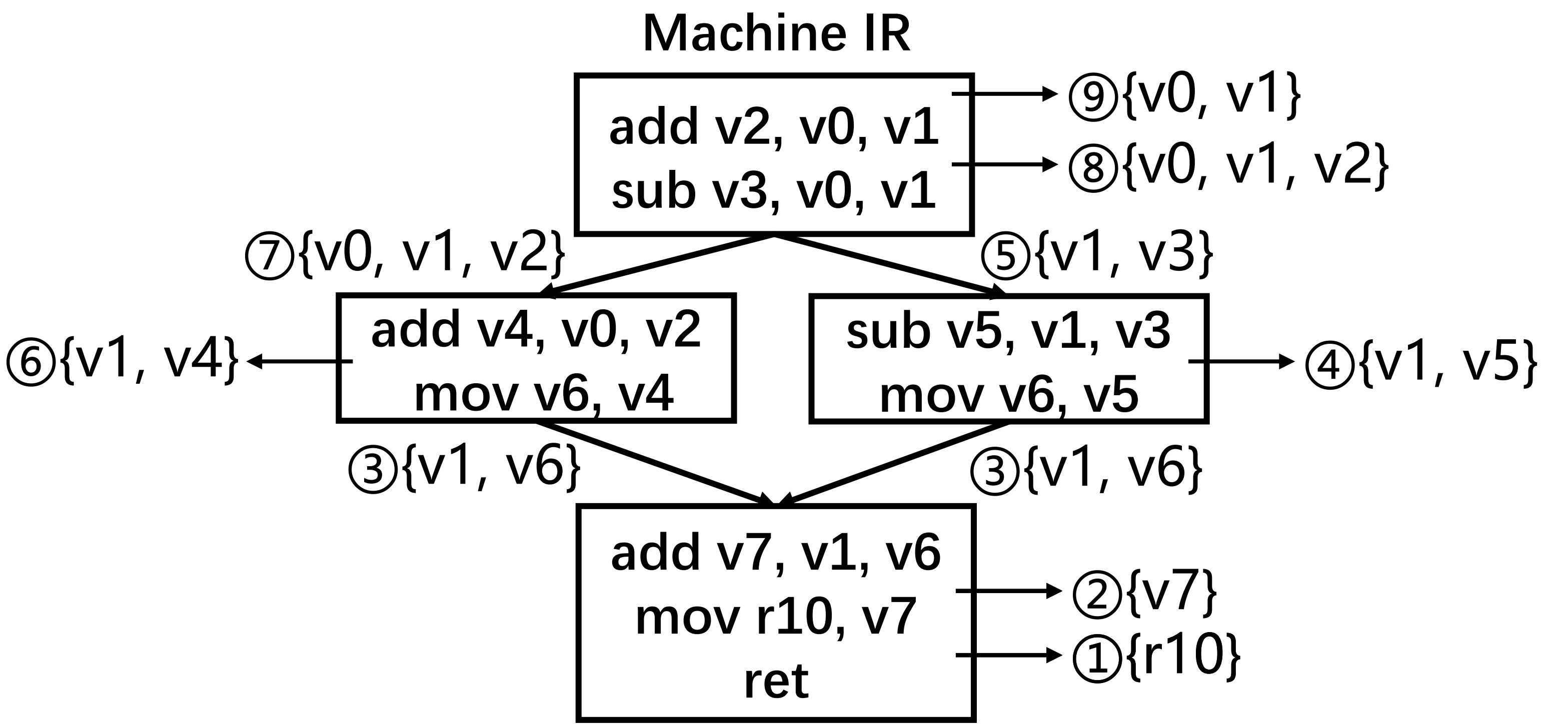
- 总体任务：将MachineIR中的虚拟寄存器映射到物理寄存器
  - 难点：保持程序语义的前提下复用有限数量的物理寄存器



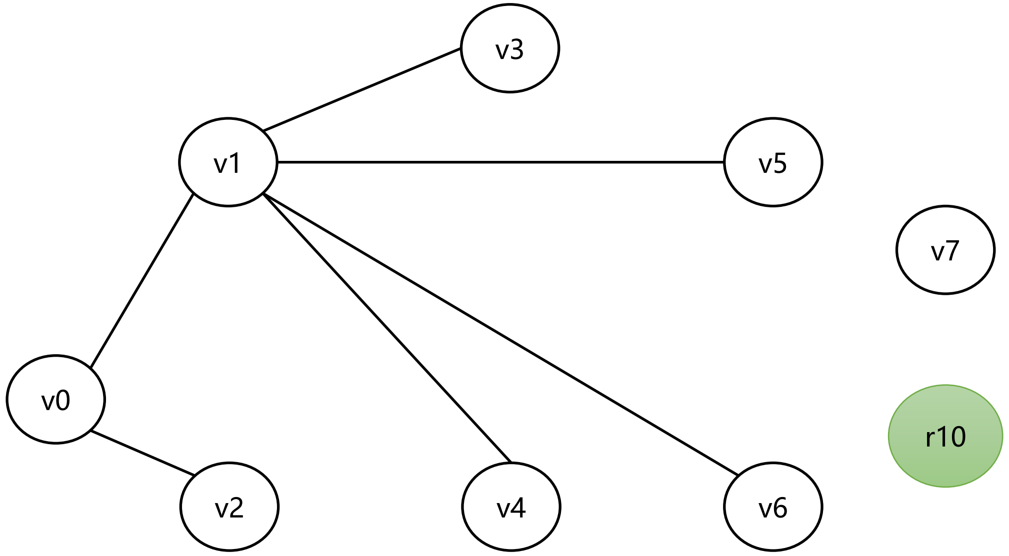
目录

- **常用寄存器分配算法**
  - 图着色算法：代码质量高，分配速度慢
  - 线性扫描：分配速度快 (LLVM默认分配算法)
  - 基于约束的分配算法
  - ……
- **图着色算法**
  - 干涉：寄存器v0、v1在某一时刻处于活跃状态，无法复用一个物理寄存器
  - 干涉图：表示寄存器之间的干涉关系，顶点为寄存器，边表示寄存器间存在干涉
  - 图着色问题：使用k种颜色为所有顶点着色，要求相邻顶点颜色不同

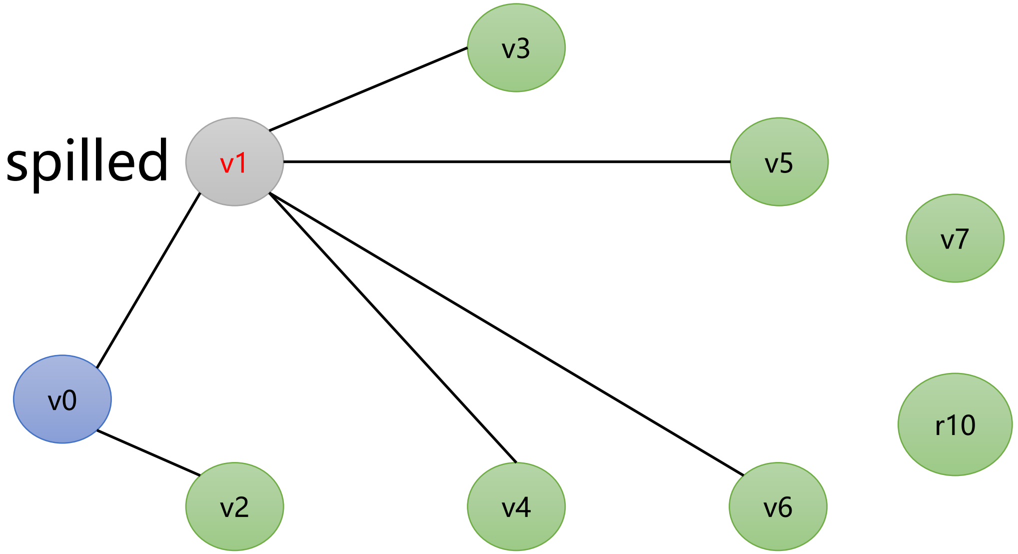
- 活跃变量分析
  - 从后往前维护
  - 对于每条指令，从活跃变量集中删除该指令的def，并添加该指令的use



- 干涉图构建
  - 将所有活跃变量集中的寄存器两两连接
  - 优化空间：冗余移动删除、寄存器合并



- 图着色：假设有2个物理寄存器
  - 找到可染色的顶点vi；若不存在，则选择一个顶点vi溢出至栈中
  - 为vi分配与邻居不同的颜色后，从图中删除vi，并更新干涉图
  - 重复上述步骤，直至所有虚拟寄存器均完成分配



- 寄存器溢出：假设虚拟寄存器v1溢出
  - 定义后将v1保存至栈中
  - 使用前将v1从栈中加载

```
mul v1, r6, r6
...
add r7, r6, v1
sub r6, v6, v1
```

v1 spilled →

```
mul r7, r6, r6
sw r7, imm(sp)      store after def
...
lw r7, imm(sp)      load before use
add r7, r6, r7
sub r6, v6, r7
```

- **图着色算法流程总结**
  - 活跃变量分析
  - 干涉图构建
  - 图着色
  - 寄存器溢出

- **优化空间**
  - 干涉图构建效率
  - 顶点着色顺序：若v1先着色结果如何？
  - 寄存器溢出后load/store消除

感谢

请批评和指正