中国科学技术大学

学士学位论文

中国科学技术大学

编译竞赛特等奖代码的分析与总结

作者姓名： 吉志远

学科专业： 计算接科学与技术

导师姓名： 徐伟 高级实验师

完成时间： 二〇二一年五月十五日

University of Science and Technology of China

A dissertation for bachelor's degree

Analysis and summary of the compiler contest grand prize source code

Author: Ji Zhiyuan

Speciality: Comupter Science and Technology

Supervisor: Senior Laboratory Technican Xu Wei

Finished time: May 15, 2021

致　谢

在毕业设计的学习过程中，我有幸得到了这些人的帮助，首先我要感谢我的老师，他们指导我的学习和方向，感谢徐伟导师的悉心教导和核心帮助，感谢卢建良班主任老师的引领和指导；然后我要感谢我的父母，是他们给予我重要的支持；感谢我的同学和朋友，他们是我日常生活中最浓墨重彩的点缀和陪伴；最后要感谢学校提供的各种平台和环境，感谢校 linux 和 vlab 社团的技术和贡献，是他们提供了我的毕设研究环境。没有他们不可能有本文的产生，再次表示诚挚的谢意！

目 录

中文内容摘要

为了更深入的学习编译原理，并了解编译器的设计和实现，本文就 2020 年首届全国大学生计算机系统能力大赛编译系统设计赛的特等奖项目“燃烧我的编译器”的源代码进行了阅读。

该项目代码量大，设计巧妙，表述简洁清晰，实现了编译优化领域的大量经典优化算法，是一个类 LLVM 实现方式的简单 C 语言子集编译器设计的模范，同时本项目不失创新，在图染色的寄存器分配算法和循环多线程部分有很大自主性创意，这些亮点值得分析和研究。

本文就项目中实现的各种优化分析算法如控制流图简化，半剪枝法构造静态单赋值形式，Tarjen 算法计算强连通分量实现循环查找，改进的启发式图染色法分配寄存器和其他创新部分进行了阅读和分析，并辅佐以实例来更好的解释说明，是对比赛代码的详细分析和总结。

关键词：编译；编译器设计；编译原理；编译优化

Abstract

In order to learn more about compile principles used in compiler design and implementation, in this paper the writer reads the source code of project ”Flamming My Compiler”, the grand prize project in the compiler system design competition of the First National Computer Systems Competitiveness Competition in 2020.

This project has a amount of source code, and the design of project is simple with concise and clear construction. It implements a large number of classical optimization algorithms in the field of compilation and optimization, and is a model of simple compiler design for a subset of C programming language like LLVM. At the same time, the project is innovative and has a lot of independent creativity, for example, in the graph coloring algorithm of register allocation and the original multithreading framework, which both are worth analyzing and studying.

In this paper, the writer analyze various optimization analysis algorithms implemented in the project such as control flow graph simplification, semi-pruning method to construct static single assignment forms, Tarjen's algorithm to compute strongly connected components to achieve loop find, improved heuristic graph staining method to allocate registers and other innovative parts, supplemented with examples to better explain and illustrate. This article is a detailed analysis and summary of the competition code.

Key Words: Compile; Compiler Design; Compile Principles; Compile Optimization

第一章　绪论

第一节　编译器设计竞赛简介

一、发起背景

编译原理一直以来都是计算机科学的重点研究方向和热门话题，为检验人才培养成效，计算机类教指委和系统能力培养专家组于 2020 年共同发起首届“全国大学生计算机系统能力大赛编译系统设计赛（华为毕昇杯）”。该赛事面向全国高校本科生，以鼓励学生设计、实现一个综合性的编译系统，展示面向特定目标平台的编译器构造与编译优化的能力为目标，提升参赛者在计算机系统设计、分析、优化、应用方面的能力，是我国高校编译系统领域唯一的学科竞赛。

编译系统设计赛要求各参赛队综合运用各种知识（包括但不局限于编译技术、操作系统、计算机体系结构等），构思并实现一个综合性的编译系统，以展示面向特定目标平台的编译器构造与编译优化的能力。本文就着重对这次比赛由来自中国科学技术大学的队伍“燃烧我的编译器”队的特等奖项目进行了分析和总结，挖掘和研究了项目中的亮点之处。

二、竞赛评分标准

大赛要求各参赛队综合运用前文提及的各种计算机系统能力知识，设计一个综合性的编译系统，各参赛队可自行决定编译器体系结构、前端与后端设计、代码优化等细节，同时需要兼顾编译器的迁移能力，纠错能力，拓展能力。因此，本次比赛对编译器的设计提出了较高要求，较为接近一个现代编译器的简化设计。

参赛队伍的项目需要通过各个基本的功能测试用例和性能基准测试用例，在通过功能测试的前提下，记录每个基准测试在目标硬件平台上的执行时间作为评价依据。

三、编译器面向语言 SysY 简介

SysY 语言是本次大赛要实现的编程语言，是 C 语言的一个子集。每个 SysY 程序的源码存储在一个扩展名为 sy 的文件中。该文件中有且仅有一个名为 main 的主函数定义，还可以包含若干全局变量声明、常量声明和其他函数定义。SysY

语言支持 int 类型和元素为 int 类型且按行优先存储的多维数组类型，其中 int 型整数为 32 位有符号数；const 修饰符用于声明常量。

SysY 语言本身没有提供输入/输出 (I/O) 的语言构造，I/O 是以运行时库方式提供，库函数可以在 SysY 程序中的函数内调用。

函数：函数可以带参数也可以不带参数，参数的类型可以是 int 或者数组类型；函数可以返回 int 类型的值，或者不返回值 (即声明为 void 类型)。当参数为 int 时，按值传递；而参数为数组类型时，实际传递的是数组的起始地址，并且形参只有第一维的长度可以空缺。函数体由若干变量声明和语句组成。

变量/常量声明：可以在一个变量/常量声明语句中声明多个变量或常量，声明时可以带初始化表达式。所有变量/常量要求先定义再使用。在函数外声明的为全局变量/常量，在函数内声明的为局部变量/常量。

语句：语句包括赋值语句、表达式语句 (表达式可以为空)、语句块、if 语句、while 语句、break 语句、continue 语句、return 语句。语句块中可以包含若干变量声明和语句。

表达式：支持基本的算术运算、关系运算和逻辑运算，真假的表示和界定，关系或逻辑运算的结果，算符的优先级和和结合性以及计算规则 (含逻辑运算的“短路计算”) 均与 C 语言一致。

第二节　论文和项目简要介绍

一、项目简介

来自中国科学技术大学的队伍设计的项目“燃烧我的编译器”（下称，本项目）在本次比赛中取得了特等奖的最好成绩。该项目已经在 Github 上开源，网址：https://github.com/mlzeng/CSC2020-USTC-FlammingMyCompiler/。

项目设计上采用类 LLVM 的三段式编译器架构但别出新意，同时又有很多创新点，且易于阅读学习，开发时支持详细的自动化开发功能验证文件，并能生成详细的汇编代码注释。此外，本项目的拓展性较好，得益于使用了与机器或特定语言无关的中间表示形式 IR 来组织源代码解析的结果，可以方便的设计成 C 或其它语言的编译器，并利于迁移到其他常见架构平台。

最关键的是，本项目具有极强的优化能力，作为一个学习用的编译器，在针对 SysY 语言的大多数比赛测试用例时，在目标机树莓派 4B 上拥有超过 GCC -O3 的优化能力。除了简单编译器的经典优化如死代码消除，循环不变量外提，

函数内联等优化，本项目还充分利用了树莓派 ARM Cortex A7 的 4 核处理器，设计并独创了创新多线程框架，可以更好的降低寄存器切换的开销，使得项目性能在决赛中脱颖而出。

二、项目架构

典型的三段式编译器架构如图 1.1：

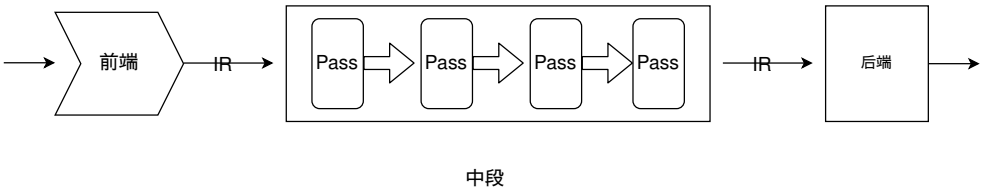


图 1.1　编译器的三段式设计

而本项目对中端做了进一步的拓展，并细分了后端表示中虚拟指令的“底层程度”，下面对各个层次的架构进行简要说明。

首先，前端读取 SysY 源代码，使用 Flex 进行词法分析，生成 token 流，在这一步中，编译器主要是识别特殊符号，字符串，关键字和标识符等。然后进行语法分析，将 token 流的形式组织成抽象语法树 AST。最后前端需要采用访问者模式遍历抽象语法树，生成与机器无关的中间表示，即生成 IR，严格意义上的 IR 与特定的编程语言和机器架构实现都无关，这样便于拓展和移植编译器项目。

但是为降低从 AST 得到 IR 的难度，即尽量使得 IR 描述能力接近高级编程语言。同时，后端又希望 IR 可以方便的转化为机器码，即希望 IR 能够方便的翻译到底层汇编语言。为了联系这些看起来非常难同时实现的要求，计算机科学领域的其他实现已经告诉我们了答案：分层。本项目采用了三层 IR 的设计（High IR、Middle IR、Low IR），使用 HIR 弥补通用 IR 和 AST 的差距，使用 LIR 缩小 IR 和 asm 的差距。第二章中会简要介绍本项目的三层 IR 机制。

项目的后端负责 IR 到机器码的生成，结合 ARM 汇编的指令，负责寄存器分配和硬件资源的进一步利用，与中间代码优化的低层 IR Pass 相配合共同完成指令融合、调度和选择等优化。后端部分包含三个层次的虚拟指令，上层指令会选择最小代价智能翻译成一系列下层指令, 在第四章会简要介绍。

整个编译过程可以描述为如下的流程。

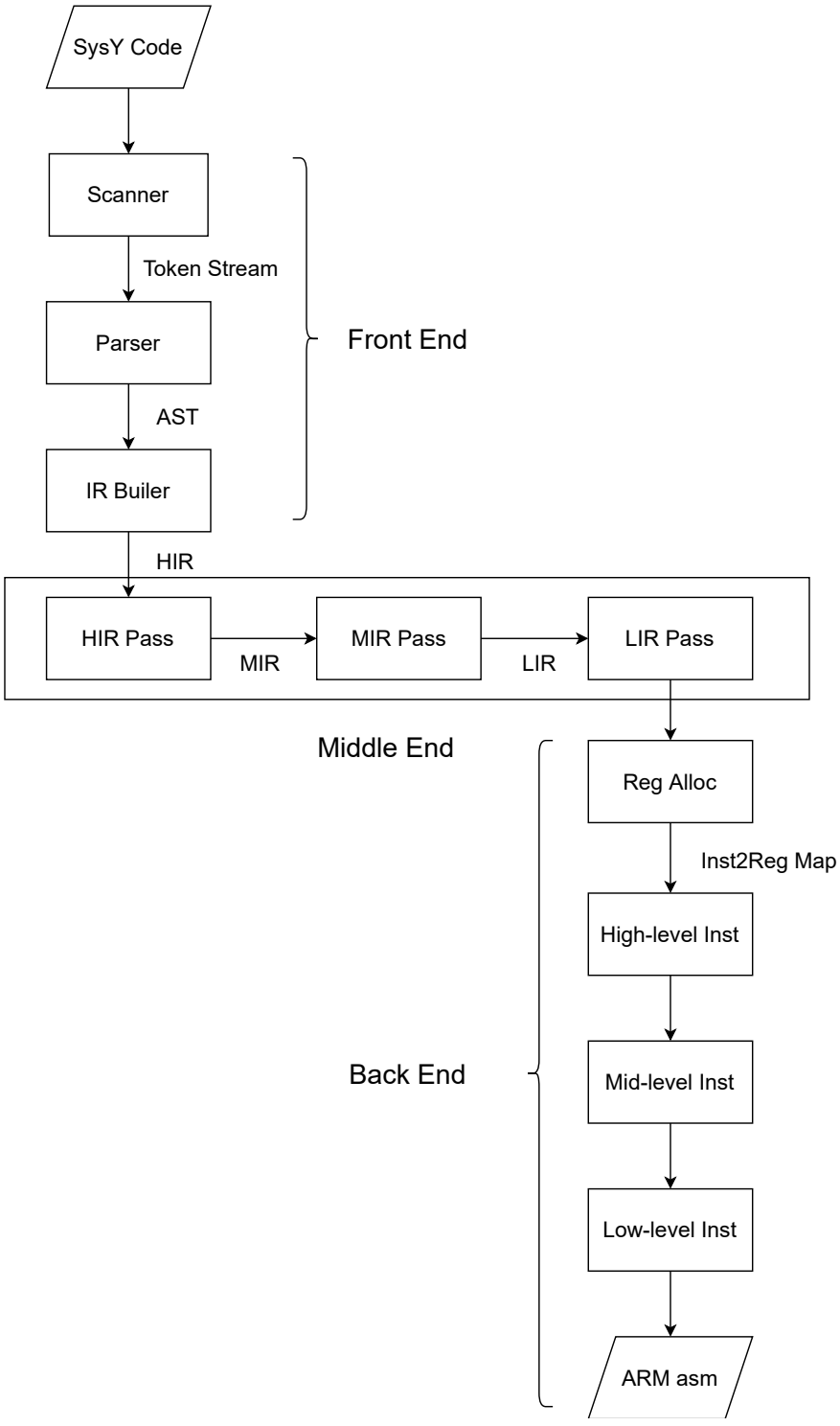


图 1.2　本项目的执行流程示意图

三、论文组织

本文的剩余内容主要对上方列举的编译器中端的 IR Pass 中挑选几个具有代表意义的优化进行着重的理论说明和举例分析，同时兼顾后端中重要的组成部分寄存器分配算法。

第二章将介绍前端产生的 IR 组织方式，简要介绍由解析器生成的 AST 形式

如何，怎样转换到中间表示，及中间表示的三层组织，并会对 HIR 的结构作一个简要介绍，将其与 AST 进行对比来说明 HIR 为从 AST 到真正 IR 所做的铺垫。第三章将介绍中端 IR 的 Pass 概念，即对如何对 IR 做优化，并举了简化控制流程图的例子来具体说明。第四章会介绍本项目中用到的 IR 的重要表示形式：静态单赋值形式及其构造，会介绍图论中的重要概念——支配理论。第五章介绍基于循环的优化，并举例说明之前构造的 SSA 对优化的设计有什么好处和简化。第六章介绍后端部分，主要是一些指令的底层化和寄存器分配算法。最后一章是对全文的总结。

第二章　编译器前端

第一节　抽象语法树

一、AST 简介

通常而言，编译器前端的主要流程是：先读取源代码文件中的字符流，然后通过词法分析生成 token 流，再通过语法分析生成抽象语法树（AST），最后将 AST 转换为与编程语言无关的 IR 送往中段，进行各种通用优化。

AST 是程序结构的一种抽象表示，以树的形式表现编程语言的语法结构，树上的每个节点都表示源代码中的一种结构，如本项目中的类 sysyParaser 定义了以下结构性 AST 节点：

表 2.1　源码结构的抽象语法树

AST node	描述
CompUnit	编译单元，即源文件
FuncDef	函数声明，对应唯一函数名
Block	大括号作用域
SelectStmt	if 语句的语法结构
LoopStmt	while 语句的语法结构

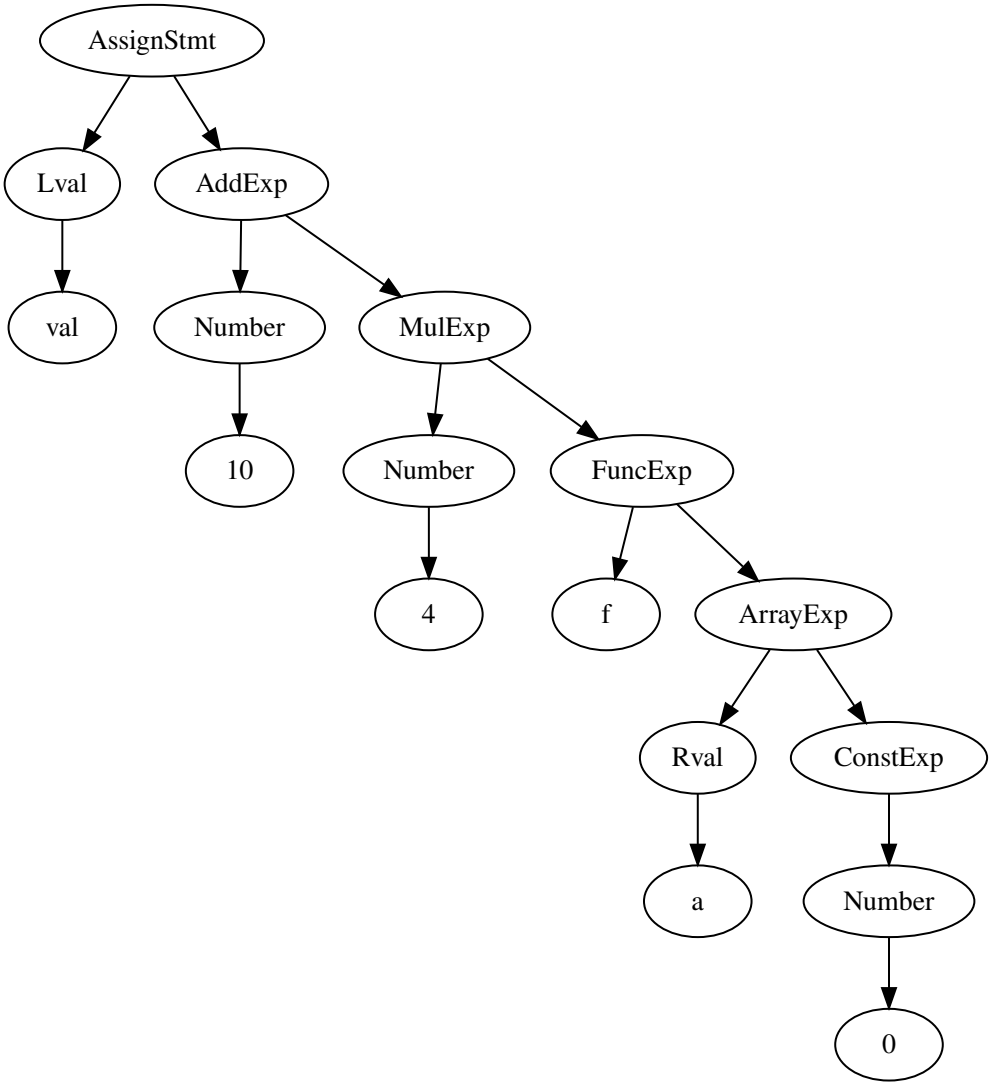
这些结构都会直接翻译到 HIR。对于指令和操作数对应的节点，这里以声明和赋值语句为例：

表 2.2　声明和赋值语句的抽象语法树

AST node	描述
VarDecl	声明语句，如 int a
VarDef:int	声明的作用域内唯一的变量名
AssignStmt	赋值语句，如 a = 1
Lval	左值，查找作用域最近的左值变量的名称
AddExp	加法的优先级最低
MulExp	乘法优先级较高
ArrayExp	数组取元素优先级高于算数运算
FuncExp	函数作用符优先级最高
Number	立即数

二、AST 节点生成举例

指令 var = 10 + 4 ∗ f(a[0]) 经过词法分析和语法分析后得到的 AST 表示如图:



var = 10+4*f(a[0])

抽象语法树保留了源程序的结构性信息，适合转换为 IR。

第二节　生成中间表示

一、中间表示简介

抽象语法树可以转换为更容易进行分析变换的中间形式 IR, 好的 IR 设计应该满足如下特点:

1. 容易从 AST 转换得来

2. 利于进行各种通用优化

3. 容易转换到各种平台的机器码

为了对应以上三个需求，本项目提出了三层 IR 的设计，即利于从 AST 得到，保留结构性信息的高层 IR，利于优化的中层 IR，和直接对应到比赛测试平台目标机器底层架构的底层 IR，如图 2.1。

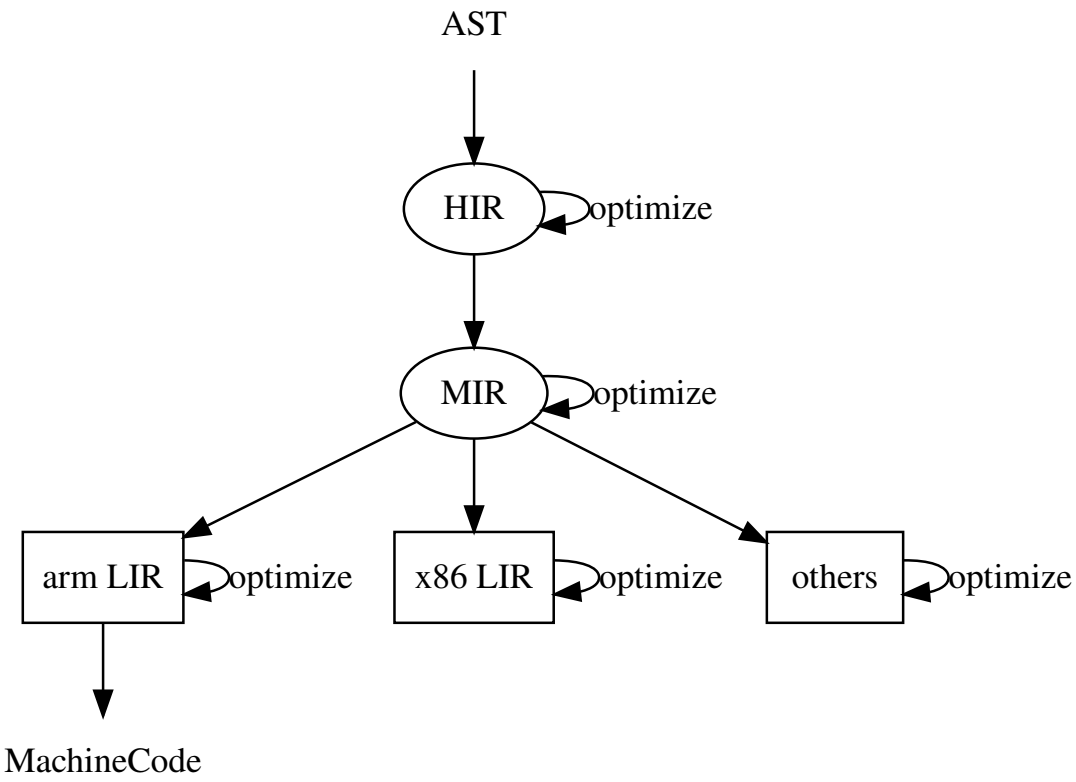


图 2.1　三层 IR 的示意图

高层 IR 保留结构信息，利于从 AST 变换，中层 IR 即通常意义上与机器和语言无关的中间表示，利于优化，底层 IR 与特定架构相结合，进一步提高优化效果，生成目标机器码。

1. 高层 IR

设计上保留了源代码 if while 等结构的信息，方便结构级变换。在该层的主要分析和优化有：

AccumulatePattern：累加变量外提

HighIRsimplyCFG：高层 IR 上的流程图简化

LoopMerge：while 循环合并

MergeCond：嵌套 if 条件块合并

2. 中层 IR

中层 IR 设计上接近 LLVM IR，适合于各类通用优化，是本文论述的重点，在该层的主要分析和优化有：

ActiveVars 活跃变量分析

BBCommonSubExper 块内公共子表达式消除

BBConstPropagation 块内常量传播

BranchMerge 分支合并

CondSimplify 条件块简化合并

ConstFlod 常量折叠

ConstLoopExpansion 循环展开

DeadCodeEliminate 死代码删除

Dominators 支配树分析

FunctionInline 简易函数内联

IRCheck 检测 IR 的数据结构是否有不一致

LoopFind 循环查找

LoopInvariant 循环不变量外提

Multithreading 循环多线程化

PowerArray 将特殊形式数组的访问替换为计算

ReachDefinition 到达定值分析

RefactorParlins 交换操作数位置

SimplifyCFG 基本块合并与删除

Mem2Reg 半剪枝算法构造 SSA 形式 IR

3. 低层 IR

如上方图 2.1, LIR 设计上贴近硬件架构, 舍弃了移植性但是进一步利用 ARM 架构的特征提高了性能，与后端相配合共同完成指令融合、调度和选择等优化。

InstructionSchedule: 指令调度软流水

LowerIR：将中层 IR 翻译到低层 IR 并在上面做一系列相关优化，如针对数组寻址利用 ARM 架构的加法和位移“融合”指令来减少指令数，针对高中层的求余指令翻译到 ARM 架构的实际运算指令，消除不被用到的操作数等等。

RegisterAllocation: 寄存器分配

二、访问者模式生成 IR

访问者模式是一种行为设计模式，能够将算法操作（访问者）与其被作用的将对象（被访问者）分离，适用于对象本身具有复杂结构且对象相对固定但需要频繁增加对于对象的操作的场景。在编译器设计中访问者对应于对象 IRBuilder，被访问者即由词法和语法解析器生成的 AST，IRBuilder 通过 visit 方法访问 AST，AST 对象通过 accept 方法接受访问。

在编译器前端完成词法分析将 SysY 源代码程序转换 token 流，再完成语法分析将 token 流构建为 AST 之后，需要对 AST 做多次遍历，如建立函数和全局变量列表、合法性检查、转换至 IR。

下面介绍对 IRBuilder 的伪代码。

算法 2.1 访问模块生成 IR

Data: syntax tree visitor of a Module
Result: Build Hight IR

```
initialization;
Create TyInt32;
Create TyInt1/Bool;
Create TyInt32Ptr;
Create Void;
while List<DeclDef> not empty do
    visit DeclDef;
end
```

当模块中的 DeclDef 子语法树节点在接受访问时，除了不创建 Void，32 位整数，32 位整数指针，布尔变量这四个全局唯一的类型外，其他代码相同，即对其语法树的子节点，深度优先遍历，直到叶节点。以函数语法节点的遍历为例：在函数的 IR 建立过程中，编译器分析参数类型和参数列表，函数返回值，变量名称何其作用域，根据 AST 的语法结构创建 HIR 指令序列。

三、HIR 生成示例

如以下代码文件:

```
int main() {
  int b = 10;
  int c;
```

```
    if (b>0){
     c = b+4;
    }
    else {
     c = 0;
    }
    return c;
}
```

整个文件经过词法分析和语法分析后，得到抽象语法树表示如图 2.2:

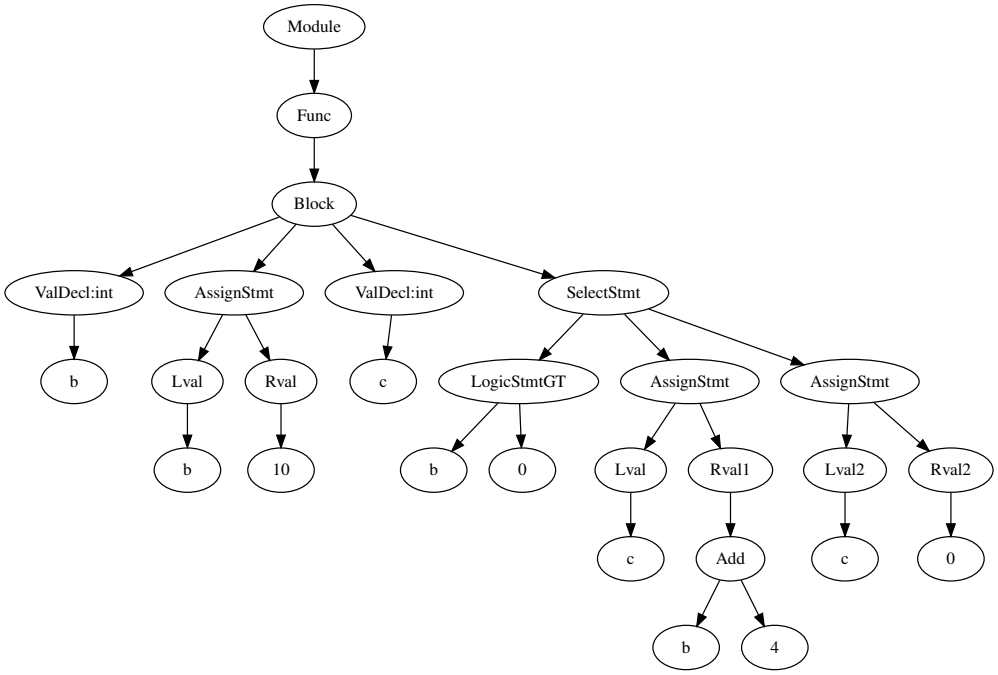


图 2.2　抽象语法树的形式

AST 的根节点 Module 即整个编译文件，由于该文件内仅定义了一个函数 main，所以根节点只有一个子节点 Func，在 High IR 层级，因为需要保留程序的结构性信息，也为了降低从 AST 得到 HIR 的难度，所以 HIR 定义的并不是基本块，项目定义了保留结构性信息的 Base Block，If Block 或 While Block，在经过 HIR 到 MIR 的转换时再生成 Basic Block。因而经过 IRBuilder 之后生成的 HIR 图 2.3中仅有顺序执行的 Block。

HIR 会保留 if 或者 while 等控制流结构，虽然方便从语法树中的 SelectStmt 部分直接得来，但是无法体现出控制流的动态流动。事实上，第二个 Block 会在转换到 MIR 时对三个部分进行拆分。

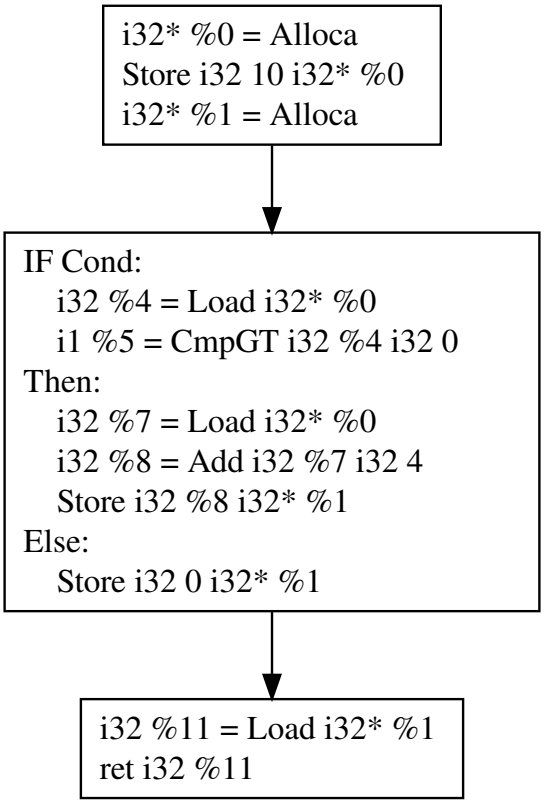


图 2.3　HIR 示例

其中源代码变量的 define 和 use 与虚拟寄存器的对应如表 2.3

表 2.3　虚拟寄存器和源代码变量对照表

virtual register	define or use
%0	变量 b 的声明
%1	变量 c 的声明
%4	if 的条件使用了变量 b
%5	if 的结果布尔值的声明
%7	add 的操作数使用了变量 b
%8	add 的结果的声明
%11	返回语句的参数使用了变量 c

四、中间表示组织

理解中间形式的组织和表示方式是理解整个项目甚至是以 LLVM 为代表的现代编译器设计理论的基础和关键，IR 按照如下层级关系进行组织：

1. Module : 编译的单元，通常是整个文件

2.Function : 源码中的某个函数，通常是为完成某种功能的封装

3.Basic Block : 函数中一个只能在结尾处跳转，在入口处进入的指令集合

4.Instruction : 基本块中的某条最基本的指令，是 IR 执行的最小单位

同时，为了方便后续编译器的优化，编译器会花费大量的篇幅用于索引和遍历函数，基本块，甚至是指令的操作数，因此为这些数据结构设置 C++ 风格的“迭代器”（Iterators）是有必要的，也是符合 IR 的组织形式的，它们的迭代器及其作用如下:

1. Module::iterator 对编译的单元，迭代其内部的所有函数

2. Function::iterator 对某个函数，遍历构成其的基本块集合

3. BasicBlock::iterator 对某个基本块，顺序遍历其内部的指令

这四种迭代器的设计是对 IR 的层级架构的直观表达。本项目的输入文件是一个 sy 文件，对应一个 Module，一个文件内可以定义几个函数，这对应若干个该 Module 内的 Function，一个函数内乍看起来是一定数目的指令，但是直接对控制流复杂的函数进行分析并不方便。基本块是只能从其第一条指令进入，从最后一条指令结束执行，中间不能发生跳转的指令集和，基本块在循环识别，数据流分析等优化中都有重要作用，因此我们采用包含若干条指令的基本块作为函数迭代的结果。而对基本块的遍历就是其中的顺序流指令。

第三章　编译器中端优化

第一节　中端优化处理简介

一、Pass 简介

顾名思义，Pass 就是“遍历一遍 IR”的意思。在本项目的设计和实现中，基于类 Pass 的派生类有两个，它们的名称和功能如下：

1. Analysis，只遍历和分析 IR，不对 IR 进行转换，用于得出一些结论，方便后续变换，如支配树分析，活跃变量分析等。

2. Transform，遍历和转换 IR，根据一些优化算法对 IR 做优化，如控制流图简化，循环不变量外提，死代码消除等。

所有的 Pass 类都要重写虚函数 run，用于表示这个 Pass 应该对 IR 做哪些分析或者变换，编译器需要哪些 Pass，就将其添加到一个列表中，再统一检查和执行，这种设计可以降低编译器的耦合性，方便了编译器的拓展和裁剪。后文介绍每个 Pass 时，都会对其 run 方法着重介绍。

二、本项目对 Pass 的管理

具体项目实现上，编译器设计了一个管理 Pass 对象的类 PassManager。

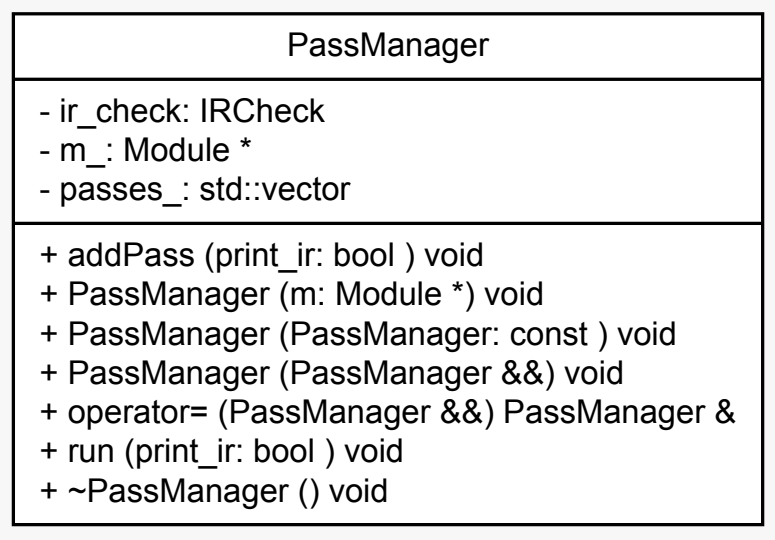


图 3.1　PassManager 类成员变量和函数示意图

图中的数据结构和成员函数的解释说明如下：

1. ir_check 是函数指针，指向 IR 合法性检查的函数对象

2. m_ 是 Module 类指针，是对要处理的编译文件的 IR 数据结构

3.passes_ 是一个向量对象，存储要进行的 Pass，通过 addPass 方法向其增加需要的 Pass。

PassManager 的 run 方法会顺序检查和调用 passes_ 中的所有优化，具体的控制流程图如下：

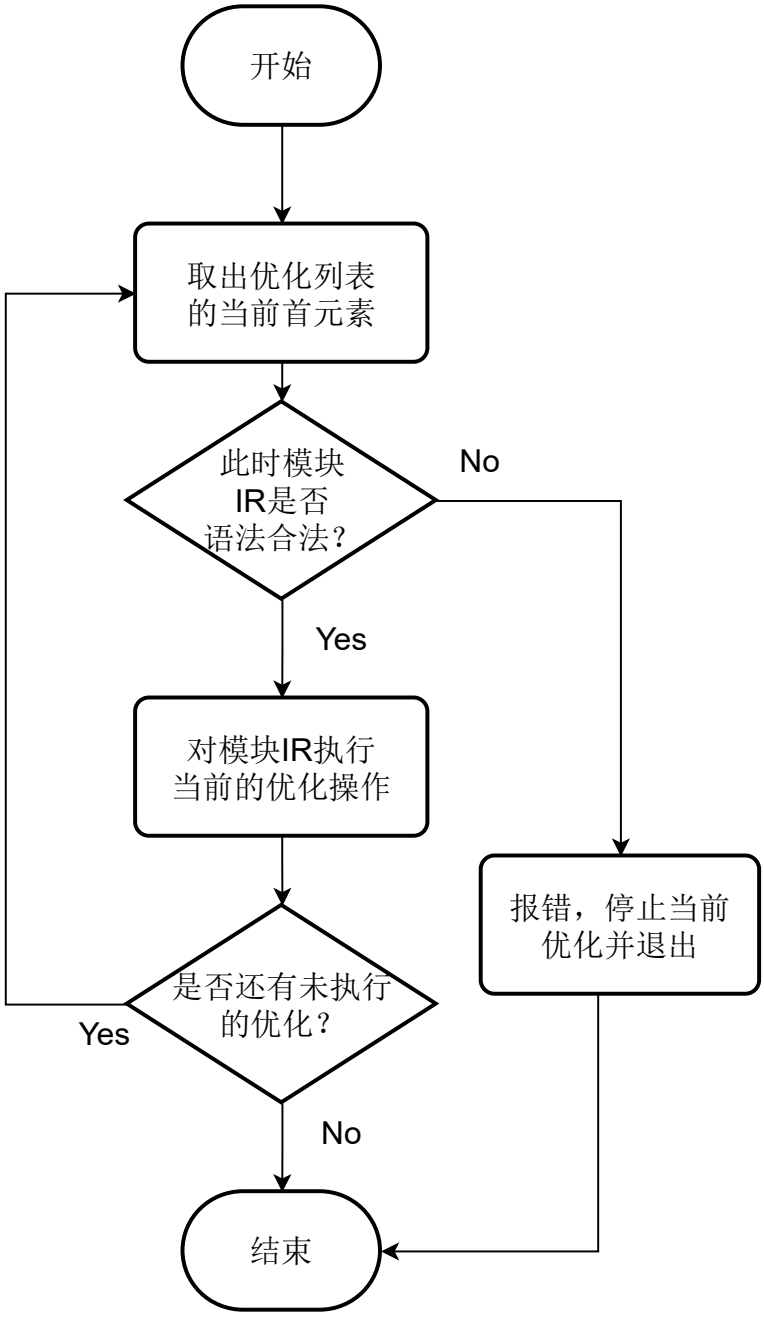


图 3.2　编译器使用 PassManager 管理每个 IR Pass 的流程图

第二节　控制流程图简化

一、控制流程图的定义

控制流程图（control-flow graph）简称 CFG，是编译器静态分析和代码优化的核心技术与重要环节。

上一章定义了基本块, 即满足只能从入口指令进入该基本块, 只能从末尾指令离开该基本块两个条件的指令序列。控制流程图的是结点即为基本块。

而控制流程图中的边对应了基本块之间的跳转关系，例如，基本块 B1 到基本块 B2 之间有一条边当且仅当 B1 的末尾有跳转到 B2 的分支指令，此时称 B1 为 B2 的前驱，同样地，称 B2 为 B1 的后继。

在一个基本块内部，平均是 4 到 6 条指令，它们往往彼此数据相关，完成对变量的操作如逻辑判断或算术运算等；同时基本块通过彼此之前跳转的逻辑关系，可以实现循环，条件分支等程序控制逻辑，进而组成函数，可以认为控制流程图是整个 IR 从源代码到实际指令承上启下的表示。

因此，对基本块构成的控制流程图的简化会直接通过影响函数执行基本块的数量来影响编译结果性能的好坏。

二、简化控制流程图算法

控制流程图简化的主要目的是消除不可达或者冗余的基本块:

1. 若其他 Pass 生成了某个无前驱的基本块，那么该基本块即为不可达基本块，可以将其删除

2. 若某个基本块仅可从其前驱跳转过来，且其前驱也只能跳转到该基本块, 那么这条跳转语句显然是多余的，可以将该基本块和前驱合并

具体而言，该优化 Pass 会对 IR 中的所有函数顺序执行一些优化，首先是处理上一个 Pass 产生的冗余流程图:

1. RemoveNoPredecessorBB: 如果某 basic block 不是 entryBB 且无前驱，说明控制流不会到达该基本块，标记该 bb 为待删除，对其后继的所有 succbb，在它们的前驱列表中移除 bb；对于后继 bb 的所有指令，如果是 ϕ 指令，因为这条指令少了个待选择的基本块，修改这些 ϕ 指令的参数。

2. EliminateSinglePredecessorPhi: 如果某个 ϕ 指令（ϕ 指令将在下文介绍，这里只需直到 ϕ 指令 define 的对象和 ϕ 指令的参数列表有关）的参数仅有一个，则这个 ϕ 指令可能的值就确定为那个唯一的参数，将该 ϕ 指令的 use 对象替换为

ϕ 指令的参数。

3. MergeSinglePredecessorBB: 对每个基本块，如果其仅有一个前驱基本块 prebb，若 prebb 的最后一条指令 br 满足仅有一个跳转分支，可以删除 br，把 bb 的指令挪到 bb 的前驱中，更新 bb 后继的前驱，更新对 bb 的所有使用为指向 prebb，完成前前驱基本块和其前驱的合并。

然后处理可能由以上转换导致的无效流程图节点:

4. RemoveSelfLoop: 如果前驱基本块仅有一个而且这个前驱基本块就是自身，标记为删除。selfloop 不会直接出现在 IR builder 的结果中，只会由其他 IR 转换导致。

5. RemoveNoPredecessorBB: 如果某个基本块不可达，即从入口基本块无法访问，将其标记为删除。同样的，NoPredecessorBB 不会直接出现在 IR builder 的结果中，只会由其他 IR 转换导致。

该 Pass 的 run 方法伪代码描述如 3.1。

算法 3.1 SimplifyCFG Pass Run Method

Data: IR of a compiler module
Result: module with simplified CFG

```
for func in module.getFuns() do
    RemoveNoPredecessorBB();
    EliminateSinglePredecessorPhi();
    MergeSinglePredecessorBB();
    EliminateSingleUnCondBrBB();
    RemoveSelfLoopBB();
    RemoveNoPredecessorBB();
    EliminateSinglePredecessorPhi();
end
```

run 方法内首先尝试对上个 Pass 产生的冗余控制流进行消除，这说明该 Pass 不应该直接用于刚刚经过前端分析而生成的 IR，否则如上所述该优化 Pass 只能消除一些空函数或者无条件跳转等冗余节点。但是在后续的各种优化和变形中，程序的 IR 形式很可能会发生改变，甚至产生包含冗余或者根本不可达的无效控制流。

因此，我们可以得出该 Pass 在整个 PassManager 中的 passes 列表中应该是多次出现的，具体而言，如果执行了 Transform 类的 Pass，那么应该在后续安排

一次控制流程图的简化。

例如，经过死代码删除，这个基本块只剩下了跳转指令，那么可以删掉这些跳转指令，把该基本块的前驱的跳转目标地址设置为该基本块的后继基本块；再例如，经过常量传播，某些条件分支指令的条件被计算出了逻辑真假，而程序又不可能跳转到逻辑为假的分支目标基本块，就也可以删掉这个无用的基本块；更为常见的，一旦修改了某个基本块的分支语句，就必须检查该基本块的后继中的 ϕ 指令（将在第三节介绍）参数列表。

三、应用举例

为了说明优化 Pass 对代码执行效率的重要作用，这里以比赛中的测试用例 if.sy 为例子说明，该测试用例的代码如下：

```
int main() {
  int a = 5;
  int b = 10;
  if(a == 5)
    if (b == 10)
      a = 25;
    else
      a = a + 15;
  return (a);
}
```

经过前端解析和一定优化后得到的控制流程图如图 3.3。

在常量折叠和常量传播 Pass 中，编译器会计算出可以在编译时确定的指令结果，如第一个基本块的结尾处分支指令，可以在编译时判断出结果为真，因此可以删去跳转到结果为假时的分支。同理，之后的基本块的分支目标地址也可以在常量传播中得到编译时确定。

经过控制流程图的简化，删去原流程图中的不可达基本块和无效分支。就可以使得规模包含 5 个基本块，15 条指令的图 3.3优化为仅有一条 return 指令的图 3.4。

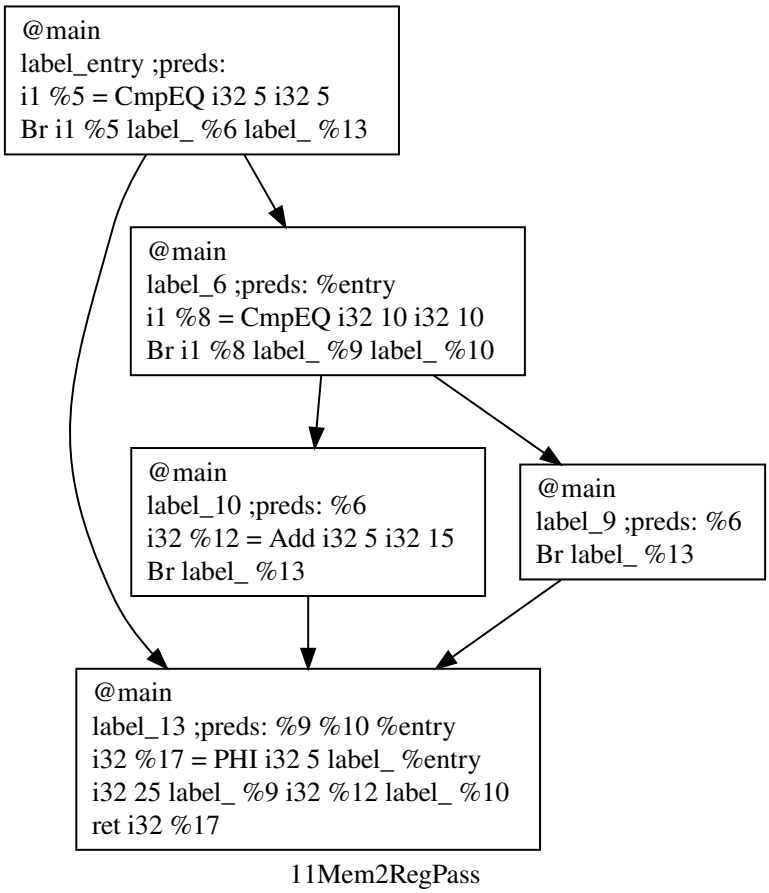


图 3.3　简化控制流之前

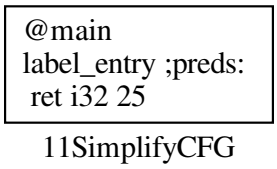


图 3.4　简化控制流之后

第四章　静态单赋值形式及其构造

第一节　从内存模型到寄存器模型

一、SSA 简介

静态单赋值形式（Static Single Assignment (SSA) form ）是一种 IR 的实现方式，SSA 由 IBM 实验室最早提出[1]，用于编程语言和编译器设计研究中。静态单赋值要求每个变量只能被赋值一次。在 SSA 中，UD 链（use-definechain，赋值代表 define，使用变量代表 use）十分明确，即每个虚拟变量都有唯一的一个 define 和在此 define 之后的 use。

SSA 形式对较多 IR 的分析，IR 优化算法实现都有很大程度上的简化。例如，有如下的源代码程序:

```
y = 1
y = 2
x = y
```

容易观察到，第一条关于 y 的赋值语句是不必须的，它会被第二条赋值语句覆盖掉，但是我们的编译器并没有这么聪明的观察能力，基于源代码的冗余赋值语句消除需要对变量 x 和 y 进行到达定值分析，判断出 $y = 1$ 处的定值因为被 $y = 2$ 处的定值覆盖而无法到达 $x = y$ 后，编译器才能删除第一条赋值语句。

如何基于 SSA 进行到达定值分析? 用 SSA 形式描述上述代码，使用“%”加一个唯一的数字来描述某个变量，得到的结果如下:

```
%1 = 1
%2 = 2
%3 = %2
```

此时不聪明的编译器也会发现，整个程序中都没有对%1 的 use，这样编译器就可以简单的删去该条语句。因此，SSA 形式下使用只能赋值一次的虚拟变量简化了到达定值分析。

二、Memory 模型 SSA 的介绍

在编译器项目的前端，代码是通过 Load / Store 体系来组织的，这种 SSA 的组织方法被称为基于内存的静态单赋值表示（Memory-Based SSA）。

具体而言，每声明一个新变量，就通过 Store 将其存储在临时的栈帧上，当函数退出时，栈帧内的局部变量会被自动回收。当使用一个变量时，通过 Load 将其值从栈内地址读取，并赋值给一个新的变量名，之所以要给这个 Load 的变量取一个新名字，是为了满足 SSA 的单赋值要求。

给上一节的代码文件加上声明语句，使其能符合 SysY 的语法检查：

```
int y;
y = 1;
y = 2;
int x;
x = y;
```

那么生成的 Memory SSA 为：

```
%1 = Alloca     # Allocate stack address %1
Store %1, 1     # Store 1 into %1
Store %1, 2     # Store 2 into %1
%2 = Alloca     # Allocate stack address %2
%3 = Load %1    # Load %2 into %3
Store %3, %2    # Store %3 into %2
```

这样的好处是不需要关心后端的具体实现，不牵扯到寄存器的分配问题，但是缺点同样非常明显，因为不考虑使用寄存器，所以每次使用变量之前都需要 Load 一次，这使得代码非常冗长且性能较低，例如在 Pentium 4 架构中，基于栈式的局部变量分配的程序相比基于寄存器的局部变量分配会需要 32.3% 的额外执行时间[2]。显然这并不适合作为竞赛用的体系架构。

三、Register 模型 SSA 的介绍

基于寄存器的 SSA 是指，假定计算机有无穷多个虚拟寄存器，之前用“%”加一个唯一的数字来描述一个单赋值虚拟变量，现在寄存器模型的 SSA 形式下可以理解为一个序号唯一的虚拟寄存器。

此次比赛的目标平台面向 32 位 ARMv7，有较多个通用寄存器，因而采用寄存器模式的 SSA 可以有较好的效果，相比寄存于内存的 SSA 可以消除冗余 Load 或 Store 的次数，提高程序性能。

但是用虚拟寄存器表示 SSA 时，会遇到某些麻烦。考虑用 IR 表示一个循环的情况，那么迭代变量的值会被重复赋值，这违反了 SSA 的单赋值要求。

注意到迭代变量的赋值可以被划分为两种情况，第一种，刚刚经过初始化，第二种，自上次循环递增而来。在控制流程图中，这表现为迭代变量所在的基本块有两个前驱：初始化前驱和更新前驱，为了满足静态单赋值的要求，我们可以引入 ϕ 指令:

ϕ 指令用于根据当前基本块的执行期间的实际前驱从 ϕ 指令的参数列表中选择一个值。有了 ϕ 指令，就可以解决基于虚拟寄存器的 SSA 实现多前驱的问题，如循环体基本块 Loop Block 有两个前驱，且这两个前驱都对代码中的变量 x 进行了赋值，一个是初始值 0，一个是更新值 $x+1$，CFG 表示如下:

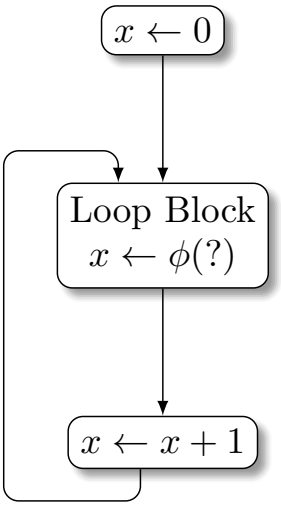


图 4.1　非 SSA 形式的 CFG

但是这显然违反了 SSA 的要求：对每个变量只能赋值一次。为了在不违反 SSA 的前提下实现循环，我们需要一种方式来重命名图中的 x。

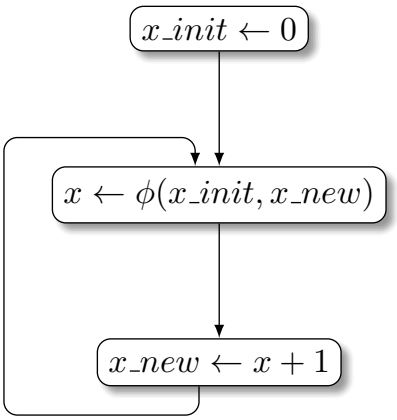


图 4.2　SSA 形式的 CFG

ϕ 指令表示，x 的值取决于该指令的运行时前驱，即如果上个执行的基本块是循环初始化基本块，那么赋给 x 的值就是 x_init，即 0；如果上个执行的基本块是循环中的更新变量基本块，那么赋值给 x 的就是 x_new。

四、phi 指令的插入位置

从上方的例子中可以意识到，只有某个基本块的前驱数大于等于 2 时，才是有必要使用到 ϕ 指令的，但是对于每个变量都需要一条 ϕ 指令吗，这个答案是否定的，考虑图 4.3的例子（为了形象直观，这里的虚拟寄存器是采用变量加数字的形式描述，实际编译器项目中采用全局唯一的数字描述）。

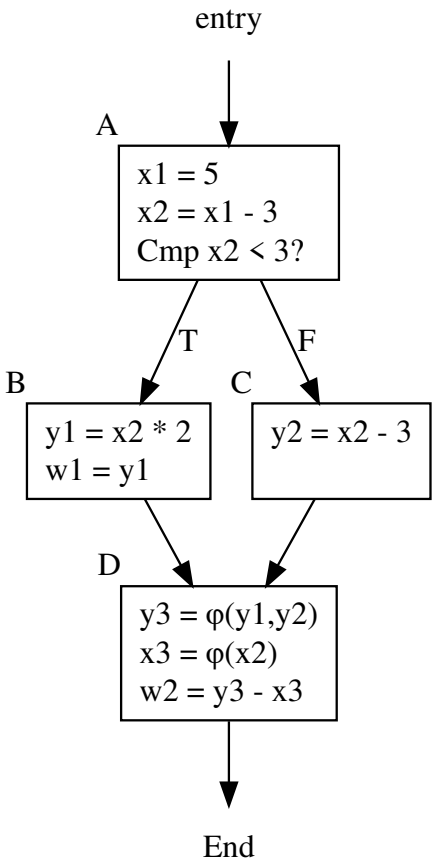


图 4.3　An SSA CFG Example

由该图可见，y 在 D 中的使用，可以被指定为 y1 亦或是 y2，这得根据它流程的来源来决定，所以插入一条关于 y 的 ϕ 指令；但是对于 x 或者 w，x2 或 w1 在 D 中就是它们的唯一版本，即关于 x 或者 w 的 ϕ 指令都只会有一个参数，因而 w 或 x 没有需要插入 ϕ 指令的问题，$x3 = \phi(x2)$ 是可以删去的。

给一个任意的控制流，如何确定哪些变量需要在哪里设置 ϕ 指令？这是一

个寄存器模型 SSA 生成中必须面对的问题，支配边界（dominance frontiers）可以有效的解决这一困难。

第二节　支配关系分析

一、什么是支配关系

支配（Dominance）是图论中的概念：控制流图的一个节点 d 支配节点 n，当且仅当从 entry 节点到节点 n 的每一条路径均要经过节点 d，写作 d dom n)。

一些来相关概念：

1. 说一个节点 d 严格控制节点 n，当且仅当 d 控制 n 而不等于 n。

2. 节点 n 的立即支配节点的定义是，严格支配节点 n，却不支配任何严格支配节点 n 的其他节点。

3. 一个节点 d 的支配边界是一个点集，其中任意节点 n 均满足，d 能严格支配所有节点 u（(u,v) 是图中的一条有向边），却不能严格支配 n。支配边界就是 d 支配能力的极限。

4. 一个支配树是一棵树，它的子节点是被其立即支配的所有节点，如图 4.4。

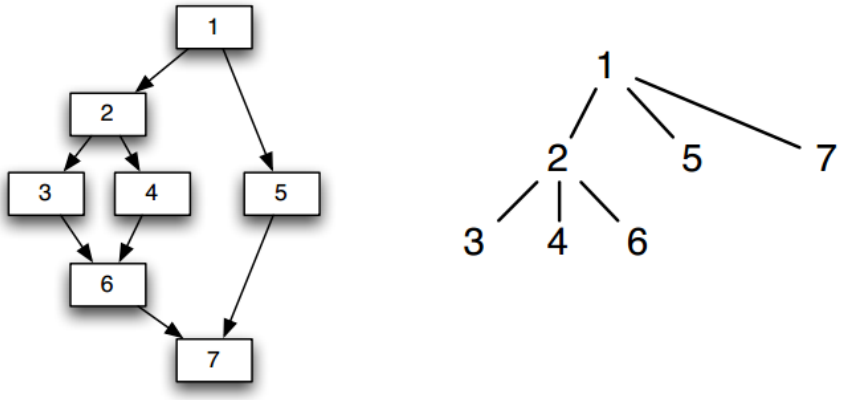


图 4.4　某个 CFG 和对应的支配树

注：根据上述 4 条定义，知道了立即支配关系，连接立即支配关系下的父子节点就得到了支配树。在图 4.4中，1 支配图中所有节点，因为 1 是全图的入口节点。2 支配且立即支配 3 和 4，同时由于 6 的可达路径只能为 1-2-3-6 或 1-2-4-6，因此 6 被 2 支配但不被 3 或 4 支配，进而，2 立即支配节点 6，同理可得 1 立即支配 5 和 7。

二、支配关系分析的算法

1. 支配关系计算

目前求解支配树效率最高的算法是 Lengauer-Tarjan algorithm[3]，该算法有接近线性的复杂度，但是实现较为复杂，教学领域介绍的更多是数据流迭代求解的算法[4]：支配 n 的节点就是 n 和支配 m 的节点的并集，其中 m 为 n 的前驱。

$$\mathrm{DOM}(n_0) = \{n_0\}$$

$$\mathrm{DOM}(n) = \{n\} \cup \left(\bigcap_{m \in preds(n)} \mathrm{DOM}(m) \right)$$

本项目对该算法的实现伪代码可以描述为：

算法 4.1 The Iterative Dominator Algorithm

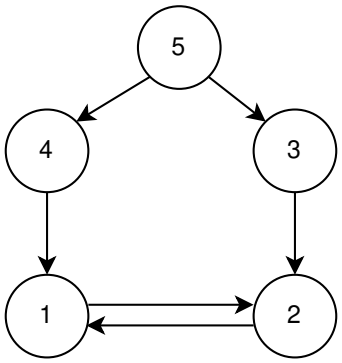
```
for each n in nodes do
    DOM[n] ← {1 ··· n}
end
Changed ← True
while Changed do
    Changed ← False
    for each n in reversedPostOrder(nodes) do
        new_set ← {n} ∪ ( ∩_{m∈preds(n)} DOM[m] )
        if new_set ≠ DOM[n]) then
            Changed ← True
            DOM[n] ← new_set
        end
    end
end
```

该算法会初始化 CFG 中所有节点的支配节点为全集，然后开始按照逆后序（reverse post order）按照数据流方程更新每个节点的支配集合，直到每个节点的支配集合都不发生变化时，结束循环。

在求迭代数据流分析问题，对于该类依赖于节点前驱结果的前向数据流问题，可以对求值过程进行适当排序，使算法经过很少的迭代就能收敛，即在对当前节点 n 进行求值之前，应该尽量让当前节点的所有前驱节点 m 在本次迭代前

就已经完成求值，否则会增加一次迭代过程，降低算法的效率。

这个“适当的排序”就是逆后序排序，下面对比逆后序排序和不按照逆后序排序进行遍历达到不动点需要的迭代次数，以下图为例：



按照 1-2-3-4-5 的次序计算节点的支配关系，需要进行四次迭代：

表 4.1　求图中的节点的支配关系

节点	Initial	1st Pass	2nd Pass	3rd Pass	4th Pass
1	Ω	{1,2,4,5}	{1,4,5}	{1,5}	{1,5}
2	Ω	{1,2,3,5}	{2,3,5}	{2,5}	{2,5}
3	Ω	{3,5}	{3,5}	{3,5}	{3,5}
4	Ω	{4,5}	{4,5}	{4,5}	{4,5}
5	Ω	{5}	{5}	{5}	{5}

采用逆后序遍历次序只需要进行三次迭代：

表 4.2　按照图的逆后序遍历的节点序列求支配关系

节点	Initial	1st Pass	2nd Pass	3rd Pass
5	Ω	{5}	{5}	{5}
4	Ω	{5,4}	{5,4}	{5,4}
3	Ω	{5,3}	{5,3}	{5,3}
2	Ω	{5,3,2}	{5,2}	{5,2}
1	Ω	{5,1}	{5,1}	{5,1}

逆后序排序类似有向图中的拓扑排序，如果用有向图中的边表示优先级次序，头节点的优先级小于尾节点，拓扑排序要求排序结果按照优先级从高到低的顺序排列，但如果图中有环，就不可能得到一个拓扑排序结果。由于 CFG 中可能有环，因此逆后序排序弱化地要求尽可能多的遍历某个节点的前驱节点后再

遍历该节点，这样可以减小迭代到不动点的次数。

2. 支配边界计算

根据支配边界的定义：Y 是 X 的支配边界，当且仅当 X 支配 Y 的一个前驱结点同时 X 并不严格支配 Y。求解支配边界的算法可以用伪代码描述如下[5]：

算法 4.2 The Dominance-Frontier Algorithm

```
1  for all b in blocks do
2      if b.predecessors.size() ≥2 then
3          for all p in b.predecessors do
4              runner ← p
5              while runner ≠doms[b] do
6                  add b to runner's dominace frontier set
7                  runner = doms[runner]
8              end
9          end
10     end
11 end
```

该算法的描述如下：遍历所有节点，如果一个 b 节点的 pre 节点的数量少于 2，那么跳入下一次循环。b 节点处只有一条可达路径，不需要 ϕ 指令。如果大于 2，则逐个遍历节点 X 的 pre 节点，这个 pre 节点不是当前节点的直接支配节点，那么这个 pre 节点的支配边界里就要加入 X 节点；然后将 pre 节点作为当前节点，继续向前看其 pre 节点是否是它的直接支配节点，一直递归到前驱节点等于当前借点，或者是到达入口节点。这样，就计算出了所有 BB 的支配边界。

按照支配的概念，每个 define 必须支配对应的 use（否则定值对于使用就是不可达的），所以如果达到了某个定值的支配边界基本块，就必须考虑其它路径是否有对相同变量的定值，由于在编译期间无法确定会采用哪一条分支，所以需要放置 ϕ 指令来描述这种可能。

三、插入 phi 指令

上一节已经提到，支配边界是确定生成 SSA 时在何处插入必须的 ϕ 指令的解决方案。

不妨先不在乎开销，为了实现寄存器模型的 SSA，最简单的方法就是，每当 CFG 中的节点 x 包含一个变量 a 的定义时，所有 x 的支配边界中的节点 n 都应

该插入一条关于 a 的 ϕ 指令。这种方法被称为：最小静态单赋值形式。

然而，有些 ϕ 函数可能会变成无用的代码，最小化 SSA 并不一定生产最少数量的 ϕ 函数。为减少 ϕ 函数的数量，基于以下观察：ϕ 函数的变量只在 ϕ 函数所在基本块中被需要时才是必须的，即该变量在此基本块中活跃（live），因此，在插入 ϕ 函数的阶段，使用活跃变量信息来决定 ϕ 函数是否需要，只保留那些必须的 ϕ 函数，以此方法建构剪枝的 SSA。

如果为了减少分析开销和实现复杂度，可以采取折中的办法，使用半剪枝的 SSA，在建构时，去掉不跨越基本块的变量名，即如果某个变量只在局部使用，就不用给它设计 ϕ 函数。

三者的伪代码描述如下：

算法 4.3 Inserting ϕ nodes (minimal SSA)

```
1 for all t in nodes do
2     S ← {n | t ∈ Defs(n)}
3     Comupters Dominance-Frontier of n
4     for all n in Dominance-Frontier do
5         Insert a φ node for t at n
6     end
7 end
```

在最小 SSA 中，即为上文所提及的最简单的设置 ϕ 指令的想法：在所有的有多个前驱的基本块开头对所有变量设置 ϕ 指令。

算法 4.4 Inserting ϕ nodes (minimal SSA)

```
1 for all t in nodes do
2     if t Lives global then
3         S ← {n | t ∈ Defs(n)}
4         Comupters Dominance-Frontier of n
5         for all n in Dominance-Frontier do
6             Insert a φ node for t at n
7         end
8     end
9 end
```

半剪枝的 SSA 只计算“全局”变量；“全局”变量是指那些在此基本块中没被赋值而被使用的，即其 liveness 穿过了 Basic Block 的边界。

算法 4.5 Inserting ϕ nodes (pruned SSA)

```
1  for all t in nodes do
2      if t Lives global then
3          S ← {n | t ∈ Defs(n)}
4          Comupters Dominance-Frontier of n
5          for all n in Dominance-Frontier do
6              if t Lives-in at n then
7                  Insert a φ node for t at n
8              end
9          end
10     end
11 end
```

剪枝的 SSA 不仅只计算“全局”变量，而且还需要检查在该插入关于变量 t 的 ϕ 指令的节点 n 是否需要变量 t，即 t 的定值能否到达 n。

三者的对比如图 4.5

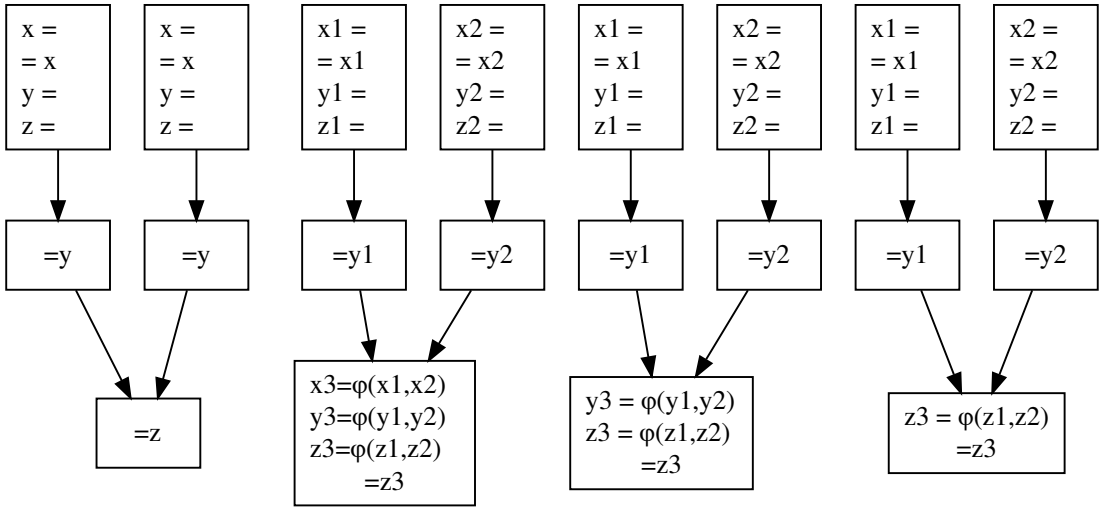


图 4.5　源流程图，最小 SSA，半剪枝 SSA，全剪枝的 SSA 示意图对比

注：x、y、z 为变量名，在赋值号左侧出现的变量名表示该条指令对变量进行定值，在赋值号右侧出现的变量名表示该条指令需要使用这个变量的值

第五章　循环优化

第一节　循环优化简介

循环是程序复杂度的主要来源，也是所有编程语言里必须支持的基本语句功能，对循环的优化通常是对程序性能影响最大的改进方式。

由于循环语句通常都是如下形式:

$$\text{for}(i = \text{start}; i < \text{end}; i = i + step)$$

因此为了描述方便，本节术语约定如下:

1. 用“迭代变量”代指循环中的变量 i,

2. 用“循环左边界”代指迭代变量的初值 i,

3. 用术语“循环右边界”代指迭代变量的结束判断条件,

4. 用“迭代步长”代指每次循环中更新 i 的变量 step,

5. 用 n 代指循环执行的次数 $(end - start)/step$。

编译原理中对于循环的常见优化有:

1. 循环不变量外提（loop-invariant code motion），把与循环迭代次数无关的计算语句移动到循环外。如循环体中的语句 $invariant = constA * constB$

2. 循环展开（loop unrolling），把循环步长增大，把本来在循环中依次执行的若干条指令在一次循环中执行，这样可以提高并发性，减少分支预测的次数，从而在硬件级别提高程序的性能。

3. 删除归纳变量（inductive variable elimination），迭代变量及迭代变量的线性组合即归纳变量，对于复制语句 j=t，如果在归纳变量 j 的所有引用点都可以用对 t 的引用代替对 j 的引用，在循环的出口处不活跃，则可以删除复制语句 j=t，这可以通过复写传播优化实现。

下面着重介绍 SSA 形式下本项目的循环不变量外提算法。

一、循环查找

要实现上述的循环优化算法，第一步应是识别控制流程图中的循环结构。控制流程图中的循环具有以下特征，一，有唯一的入口节点，记为 header，二，至少有一条回边，即从某个 header 支配的节点 n 通向 header 的路径，这两个条件是控制流程图中有循环的充要条件。

定义，假定 CFG 中存在一条回边 n header，循环由 n, header 和其他所有不需要经过 header 就能到达 n 的节点组成。

事实上，控制流程图中的一个强连通分量就是一个循环。Tarjen 算法[6] 是基于深度优先查找，对有向图的所有强连通分量顶点集的划分算法，可以使用该经典图论算法来进行循环查找，算法的伪代码描述如 5.1。

算法 5.1 访问模块生成 IR

```
Data: graph G = (V, E)
Result: set of strongly connected components (sets of vertices)
index = 0;
stack = emptry;
for v in nodes do
    if v.index is undefined then
        strongconnect(v);
    end
end
```

Tarjen 算法的核心在 strongconnect 函数，该函数的算法流程是，接受图中某一节点为参数，初始化该店的 index 为当前的时间戳，由于此时还未发现该节点的后代节点，初始化该节点在图中能达到的最早时间戳 lowlink 为其自己的时间戳。

从该点开始进行深度优先搜索，使用相同的时间戳发现和标记后代节点的序号 index。将后代节点按照被访问的顺序入栈，在深度优先搜索的回溯过程返回至一个父节点时，由于已经获得了子节点的可达性信息，可以由下而上的更新父节点的 lowlink 为父子节点分别 lowlink 的最小值。

如果该节点的 lowlink 小于其 index，说明该节点在控制流程图中可达被更早发现的节点，该类节点是某条由自身指向祖先节点的回边的头节点，也即该节点应该是循环体内的节点。

在整个深度优先搜索过程结束后，判断参数节点 v 的 lowlink 和 index 值，如果二者相同，就说明该节点是某个强连通分量的根节点，也即该循环的 header，则在它之前出堆栈且还不属于其他强连通分量的节点构成了该节点所在的强连通分量，将栈中循环体中的元素逐个弹出直至发现 v。

注意到 Tarjen 算法会对所有经过某次深度优先搜索后未被发现的节点开始一次搜索，所以 Tarjen 算法结束时，一定可以得到整个图的全部强连通分量连接

情况。

算法 5.2 Tajen 算法的 strongconnect 函数伪代码

```
1  v.index = index //当前最小 index
2  v.lowlink = index //初始化为当前可达的自己的 index
3  index = index + 1
4  S.push(v)
5  for (v,w) in E do
6      if w.index is undefined then
7          //后继未访问过，深度优先访问
8          strongconnect(w)
9          v.lowlink = min(v.lowlink, w.lowlink)
10         //更新该节点可达的最小 index 值
11     else
12         if w in S then
13             //w 已在栈 S 中，则其也在当前的强连通分量中
14             v.lowlink = min(v.lowlink, w.index)
15         end
16     end
17 end
18 //深度优先搜索结束
19 if v.lowlink = v.index then
20     // 若 v 是根则出栈，并得到一个强连通分量
21     start a new strongly connected component
22     while not w==v do
23         w = S.pop()
24         add w to current strongly connected component
25     end
26     output the current strongly connected component
27 end
```

以图 5.1为例，说明循环发现的过程，假定从入口节点 entry 开始深度优先遍历，事实上，由于 Tarjen 算法会检查图中所有未被发现的节点，所以从任何一个图中节点开始遍历的结果相同。

表 5.1记录了深度优先搜索的过程中发现各个节点的时间戳 index 和从该节点回溯时确定的该节点在图中可达的最小时间戳 lowlink 值。

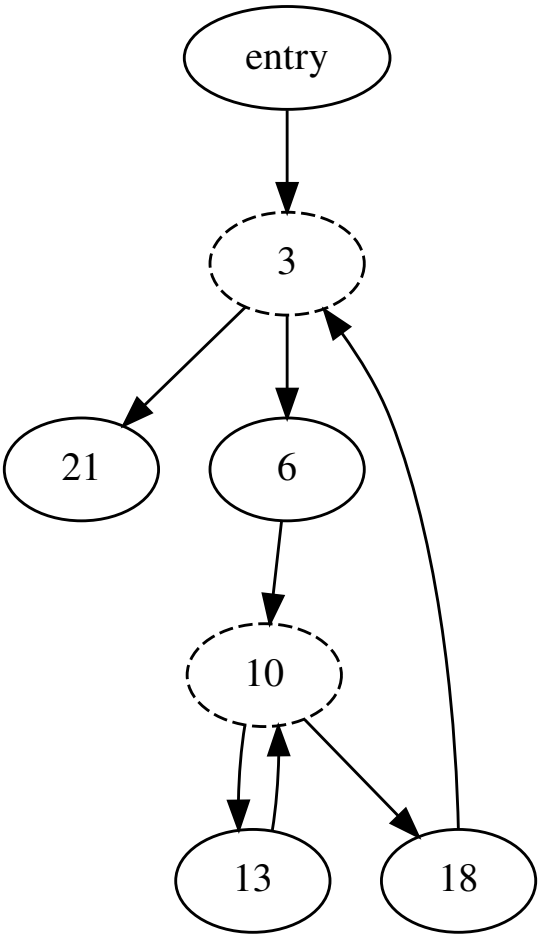


图 5.1　程序控制流程图

表中只描述了发现循环体 {10,13,18} 的过程：在节点 10 的回溯过程中，弹出在 10 之前入栈的节点 13 和 18，由于 10 的 index 与 lowlink 相同，它们构成了一个循环。对于 3 号节点，同理。对于 21 号节点，虽然 21 号节点也会被当作一个强连通分量，但是其规模为 1，无法构成循环。

表 5.1　深度优先遍历发现内层循环的过程

节点	low link	index
entry	0	0
3	1	1
6	2	2
10	3	3
13	4	3
18	5	3

第二节　循环不变量外提

经过了循环查找，如果循环的某个节点中有条指令的所有源操作数为不变量，那么这条指令可以移动到循环外。具体而言，外提的不变量形式如下：

1. 把常量标记为不变量

2. 把只在循环外定值的变量标记为不变量

3. 把所有源操作数为上述两个条件定义的不变量的指令目的操作数定义为不变量

不变量外提算法是一个数据流分析的过程，按照上述次序标记循环中的不变量指令，迭代到不动点，最后的结果中，不变量指令和循环次数无关，因此可以移动到循环之外。

而外提的具体实现比较简单：只需要在循环的进入前新加一个基本块 Loop-Invariant Area，把不变量指令提出到这个新的基本块中，然后调整该基本块的前驱后继关系，并把后续基本块中用到这个不变量指令的 ϕ 指令参数进行修改，重新使得 IR 合法。

该 Pass 类的 run 方法如下：

算法 5.3　Loop Invariant Motion

```
Result: Mark and Move Loop Invariant
Changed = True
Invariant= empty
while Changed do
    for all e in Instructions do
        mark Const as Invariant;
        mark Reached Value as Invariant;
        if all of e.operations is Invariant then
            mark e.value as Invariant;
        end
    end
    if IR_new equals to IR then
        Changed = False;
    end
end
do Loop Invariant Motion;
```

值得注意的是，如果循环的条件不满足，但是我们进行了外提，那么对于源

代码级的程序会产生多余的变量赋值，考虑循环内外都对某个变量进行了赋值，而循环又因为条件为假不会进入的情况，如果直接进行外提，编译器会错误的产生一个新赋值，这将使得循环后用到这个赋值的地方出错，那么，我们是不是应该把循环的条件部分也外提一次，即在基本块 LoopInvariant Area 前设置一个 if 条件基本块呢？比如这个案例：

```
int n = 10;
int i = 1;
int step = 0;
while (i > n){  // FALSE condition, cannot execute!
int step = 4*i;
// code
}
k = step;
// use of variable k
```

如果我们外提时没有添加 if，那么编译器会错误的产生一个新赋值 step = 4，这将使得下方的 k 出错！

```
int n = 10;
int i = 1;
int step = 0;
// if (i>n)
    int step = 4*i;
while (i > n){
// code
}
k = step;   // ERROR HERE
// use of variable k
```

事实上，这种情况是不会发生的，因为对于 SSA 形式，每个变量只有一次被赋值的机会，出现在循环内的赋值就意味着不会再出现对此变量的其他赋值语句，经过外提后只有用得上或者用不上两种情况，而不会产生上述的覆盖出错。

所以来自源代码级的外提导致的错误赋值的情况没有必要在 SSA 形式下进行考虑，验证一下，由上述代码生成控制流程图 5.2：

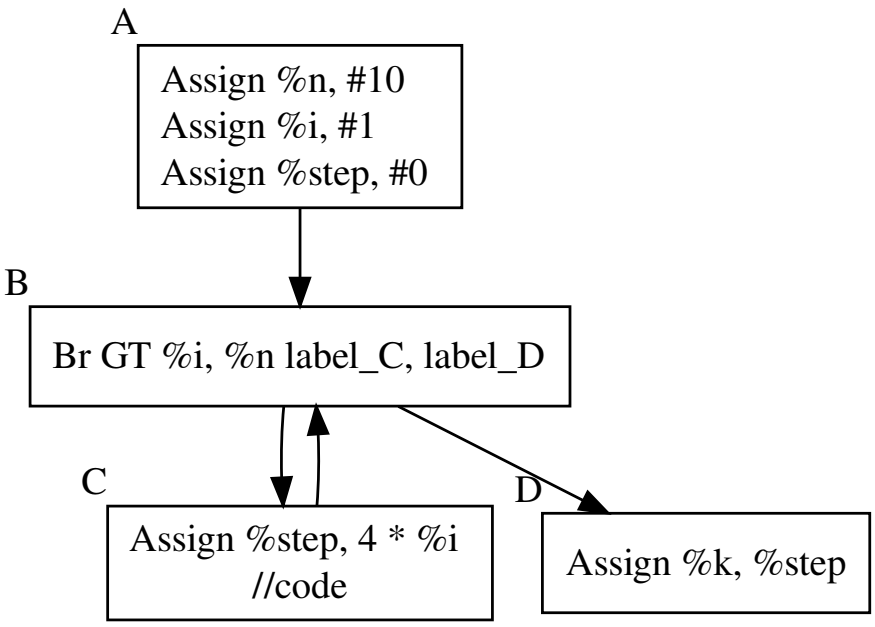


图 5.2　直接从源代码得到的控制流程图

注意，现在是 SSA 的形式，我们需要满足 SSA 的要求，即每个变量只能被赋值一次。检查整个控制流程图，可以看到虚拟变量%step 在基本块 A 和 C 中被赋值了两次，这违反了 SSA 的定义，所以这并不是合法的 SSA 控制流程图。

为了符合单赋值的要求，把基本块 C 中的虚拟变量%step 重命名为%step_new，在有多个对 step 进行定值的基本块 B 中插入一条关于 step 的 ϕ 指令，并将基本块 D 中对%step 的使用修改为%step_phi，得到图 5.3。

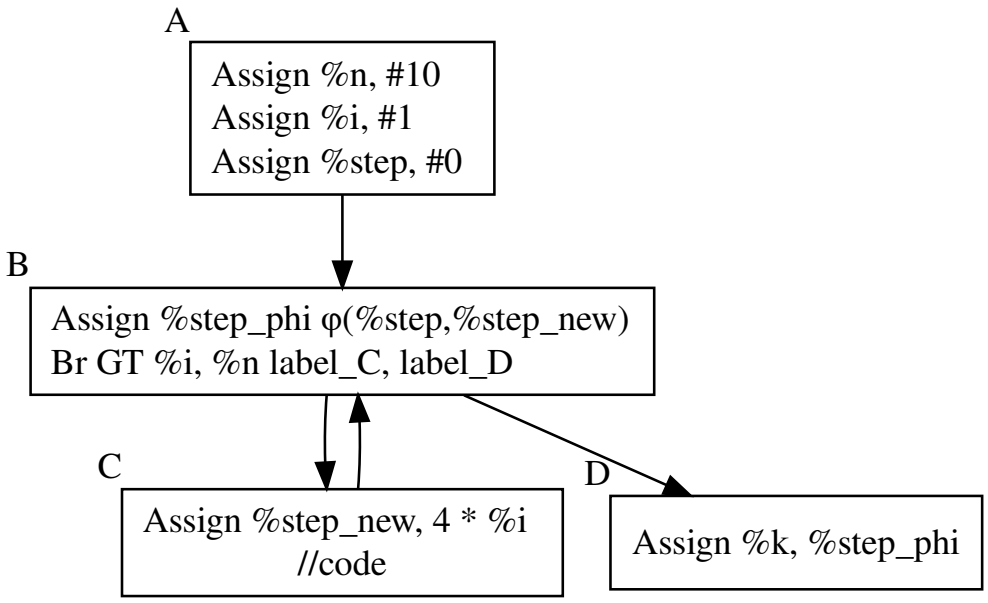


图 5.3　符合静态单赋值要求的控制流程图

这样一来就是正确的 SSA 形式，进行代码外提之后，得到图 5.4。

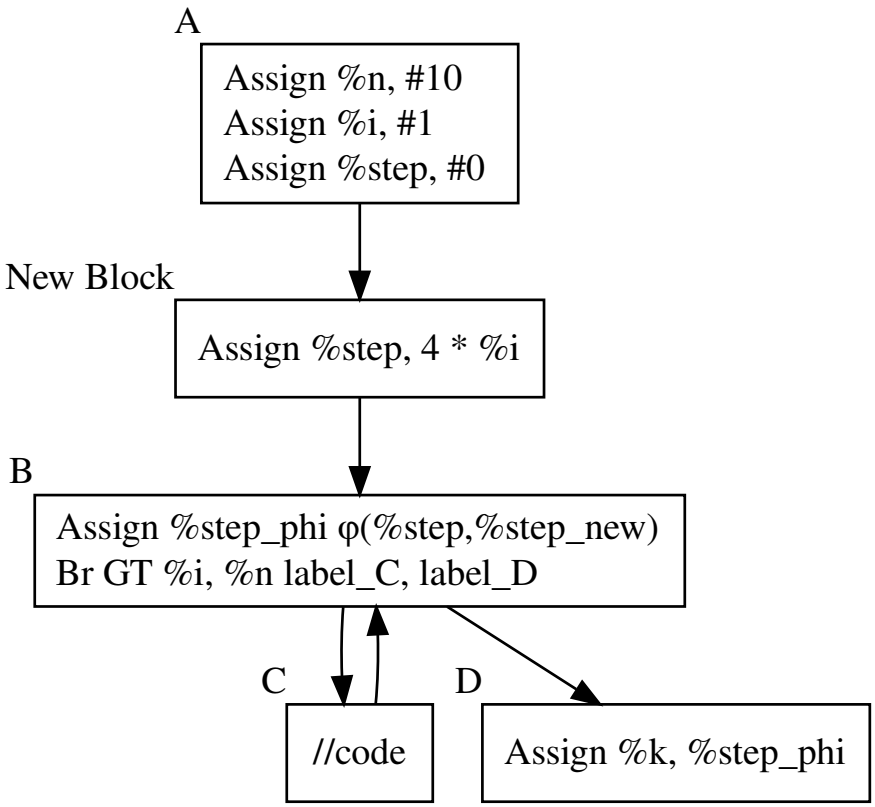


图 5.4　进行代码外提后的控制流程图

可以发现基本块 D 中 step 的值实际上取决于虚拟变量%step_phi，也即实际的运行时控制流决定了 step 的值。所以说 SSA 形式上变量外提时，是不需要添加 if 条件的。

第三节　循环多线程化

一、简介

本项目对于最简单的并行计算情况，利用了树莓派 ARM EABI 架构的 4 核处理器，开发了创新多线程框架，与经典框架 OpenMP 对比，其优点有：

1. 共享栈空间：不需要将被并行化的区域拆分出来变成函数

2. 框架更加易于实现：不需要保存上下文和维护各种信息

3. 更便于代码变换：只需在 IR 前后就地插入 Call MTStart 和 MTStart End

4. 更高的运行性能：栈上的资源仍然可以直接通过栈指针加偏移访问

本项目的多线程框架类似于 GPU 的设计思想，仅针对不需要线程通信的情况，即每个线程之间互相不会产生干扰，彼此读写属于自己的区域，这也是上文所述线程之间可以共享栈中的局部变量，而不需要互斥锁等同步机制的原因。

举例来说，对规模为 100 的数组 A 的每个元素进行平方运算，由于这个问题可以划分为 4 个独立进行互不干扰的子问题，编译器可以通过系统调用创建 4 个线程并行独立地共同完成这个问题：一号线程读写前 25 个元素，二号线程读写 26 到 50 个元素，以此类推，从而充分利用树莓派的 4 核处理器，进一步提高机器码性能。

二、多线程框架代码实现

下面介绍如何在 ARM EABI Linux 的系统上通过汇编码创建共享栈空间的子线程。

首先，多线程必然需要系统调用的支持，Linux 在 arm/EABI 的 system call 约定是 R7 作为系统调用号寄存器，使用寄存器 R1 到 R6 传递参数，R0 作为返回值。其次，要注意区分 Windows 系统和 Linux 对线程的不同设计，Linux 系统的线程设计可以表述为，线程只不过是共享虚拟地址空间和文件描述符表的进程，定义上线程之间共享除寄存器、栈之外的所有资源。操作系统和底层硬件天然地保证了线程不会共享寄存器，TLS 不是必须的，本项目又使得线程之间共享栈空间，更关键的是，由于共享栈空间，使得栈指针寄存器共享，所以局部变量也被共享，切换的开销大大缩小。

clone 跟 fork 都是创建子进程（线程），区别在于 clone 能更详细的定制子进程和父进程共享的内容：如虚拟内存地址空间，文件描述符表，因而本项目使用 clone 系统调用，设置参数 clone_flags 为父子线程共享栈空间，不必共享其他资源。

clone 对父进程返回所创建的子进程的 pid，如果失败，返回-1，这正是程序区分父子线程的关键。

```
sub r2, r2, \#1
cmp r2, \#0
beq .finish
mov r7, \#120
mov r0, \#273
mov r1, sp
swi \#0          ; system call
cmp r0, \#0
bne .mtstart
```

说明：

前三条指令表示利用 r2 统计已经创建的线程数：r2 事先初始化为 4，每循环一次减一，当 r2 等于 0 时，创建子线程完毕，

第四条指令为设置系统调用参数号为 120，即 clone 的系统调用号。

第五条指令表示设置系统调用的首个参数 clone_flags，273 的十六进制即 0x0111，在 Linux 系统代码中表示共享虚拟存储的子进程，显然由于 SysY 中无无文件读写等功能，所以只需共享最基本的虚存空间即可。

第六条指令中，由于系统调用 clone 的参数 r1 为子线程栈地址，通常框架下子线程拥有自己独立的栈空间来存储自己的局部变量，并且防止误写父线程的局部变量，但是该项目的多线程框架下无需担心此问题，所以只需要把子线程的栈指针设置为父线程的栈指针 sp 即可。

第七条 swi #0 是 arm EABI 的软中断指令，类似 x86 的 int 0x80。从这条指令之后，父子线程会开始并行执行，为了区分父子线程，利用上文提到的 clone 对父子线程返回的值 r0 不同：对子线程返回 0，子线程会顺序执行，离开子线程创建的汇编代码区域；对父线程返回子线程 pid，Linux 下的创建成功 pid 是一个非负整数，因而必然会导致父线程回到第一条指令，对 r2 减一，判断所有子线程是否已经创建完毕。

第六章　后端

第一节　底层指令变换

一、ARM 架构简介

ARM 是一个精简指令集（RISC）处理器架构家族统称，广泛应用在低功耗的嵌入式领域，同时在手机处理器市场上也占有绝对优势，最近的 Apple M1 芯片也是 ARM 架构应用于电脑的成功尝试。

树莓派也是基于 ARM 架构的单片机电脑，基于 Linux 系统，在计算机和编程教育方面有着广泛的应用，根据树莓派基金会统计，截至 2015 年 2 月，已经售出超过 500 万台树莓派，这使得树莓派成为最畅销的英国“电脑”。同时，树莓派在简易服务器搭建，并行计算，机器人控制，甚至线上服务后端等各种领域都有着广泛的创意应用。

回到 ARM 架构上，Raspberry Pi 4 Model B 是 arm/EABI 架构，有着一个 1.5 GHz 64/32-bit 4 核 ARM Cortex-A72 处理器，这里比赛中要求采用 32 位模式。ARM Cortex-A72 支持 3 路乱序超标量流水，有较多的通用寄存器，它们的作用如下

表 6.1　ARM v8 架构的寄存器作用说明

ARM REG	Description
R0	General Purpose
R1-R5	General Purpose
R6-R10	General Purpose
R11	Frame Pointer
R12	Intra Procedural Call
R13	Stack Pointer
R14	Link Register
R15	Program Counter
CPSR	Current Program State Register/Flags

在 IR 级别的指令，与机器平台无关，因而可以设计实际机器并不支持的求余指令，减少变量数目，方便分析变换。而在翻译为汇编指令的时候，再用乘法除法和加法替代。经过寄存器分配后，可以使用通用寄存器和低频临时寄存器

r12 r14，进一步向底层翻译时，可用高频临时寄存器 r11，最终生成汇编码时，使用物理指，直接对应到 ARM 指令。

二、底层指令转换

1. 融合比较和分支指令

在 IR 层级，编译器无法获得机器平台的信息，所以条件分支被设计成通用的先计算比较结果，再根据结果真假分支。了解到 ARM v8 支持根据比较结果直接分支时，在比较结果不会被其他地方用到时，编译器可以融合比较和分支指令：融合前

i1 %5 = CmpLT i32 %4 i32 100

Br i1 %5 <label> %6 <label> %21

如果%5 不会在除了分支指令外的指令中用到，就可以进行这一变换

融合后:

Br LT i32 %4 i32 100 <label> %6 <label> %21

这样可以省掉虚拟变量%5，并减少寄存器的使用。

2. 融合加法和乘法指令

ARM 架构的另一项指令集特征是；可以将移位和旋转操作融合为一条求地址的指令，此处移位操作是地址对齐因数，旋转操作表示从基址增加这个偏移量，举例来说，对于如下 SysY 指令:

a = a + (j « 2);

ARM 架构可以用一条指令实现:

ADD Ra, Ra, Rj, LSL #2

这在取数组中变量的地址时很常用。

第二节　寄存器分配

一、简介

本项目的寄存器分配算法使用图染色算法实现，虽然图染色算法不是目前最优秀的的寄存器分配算法，而且也有复杂度较低且更易于实现的线性扫描算

法，但是由于其经典性和不错的性能，所以本项目借鉴了线性扫描的思想，使用图染色的方法来进行寄存器分配。

算法的基本思想：

为了实现 SSA 中虚拟变量的 define 和 use 两个操作，需要保证某个变量在 use 时其 Value 可以被访问到，即存储其 Value 的寄存器不应该被其他虚拟变量的 Value 覆盖，那么，我们有两种实现方式：

第一种，可以把虚拟变量的值存在栈上，即“不分配”，这种实现方式比较简单，优点是不需要考虑寄存器或 CPU 等目标机器平台特性，如 JVM，便于移植，同时代码量较小，但是缺点是性能显然会由于频繁的读取存储器而较差。

第二种，把虚拟变量的值存在寄存器上，那么在这个变量的整个存活期被分配的寄存器都不应该被其他赋值覆盖。

二、图染色算法

寄存器分配算法的问题描述：

输入：一个有 N 个节点，k 种可选颜色的无向图，节点代表某虚拟变量，颜色数代表可用的寄存器数

输出：所有变量的一个寄存器分配方案

图的 k 染色问题是一个 NP 难问题，算法的基本思想如下：

1. 如果 max degree of nodes < k 那么总是能染色：把最大度数的节点删去，其他节点各分配一种颜色，至少会留下一种颜色给这个度数最大的节点

2. 如果 max degree of nodes ≥ k 也是有可能 k 染色的：最简单的例子：星型图。我们只需把这些度数超过 k 的节点从图中删去（同时删去其邻边），判断新的图能否 k-1 染色

3. 如果删去所有度数大于 k 的节点后余下的图是一个完全图，且每个节点的度数大于 k 减去删除的节点数，那么这个图是不可以 k 染色的，我们需要选择抛出的节点，将其存入内存，通过 Load 操作访问，让其不参与寄存器分配

算法流程具体如下：

1. 把所有度数小于 N 的节点压栈（从图中暂时删除）同时删去邻边

2. 当图中所有节点度数大于 N 时，用启发式函数找到最应该被抛出的那个节点，删去此节点

3. 重复上述两个步骤直到图为空

4. 逐个弹出栈中元素，它们都是可染色的

5. 对于那些被删除的节点（暂时 spill）两种解决方案：Chaitin 的算法认为这种节点只能抛出，Briggs 改进算法认为可以通过 split 分裂一个 live range 为多个来实现重新染色[7]，这里本项目采用简单的抛出。

三、算法实现和案例

1. 对于基本块的 OUT 集合中的虚拟变量，它们的存活期是整个基本块，所以它们的 remains 初始化为 1，也即整个基本块中它们存在的寄存器都处于占用状态

2. 自前向后的对于每条指令，余下的基本块部分如果该指令用到了某个虚拟变量，就可以把它的 remains 减一，模拟线性扫描算法中某个变量存活期所跨越的指令数，当 remains 减至 0 时，就说明后边儿不会再使用此变量了，该寄存器得以释放

我们需要两个数据结构：用于记录变量冲突图的 Interface Graph 和模拟变量存活期的 remians，下面将结合图 6.1来说明这 remains 数据结构的作用：

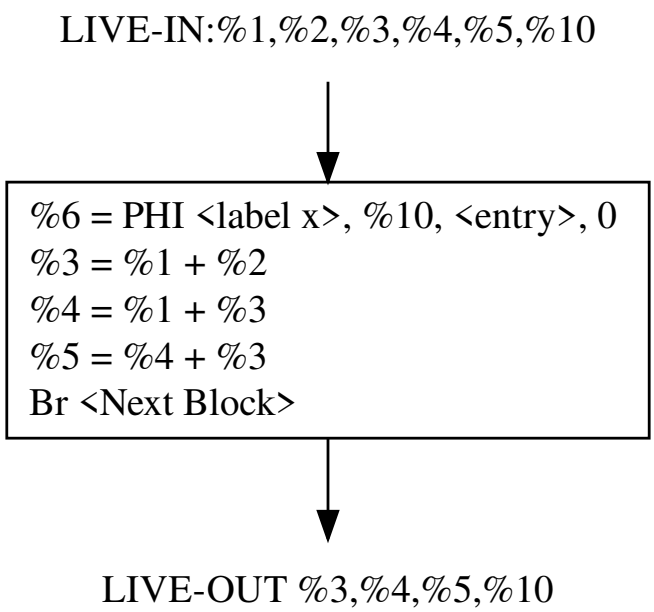


图 6.1　在块内执行寄存器分配的示例

注：LIVE-IN 表示经过数据流分析后，在该基本块需要的变量，LIVE-OUT 表示数据流分析中，在该基本块之后仍然被需要的变量。

本项目中寄存器分配在基本块级别进行，所以第一步是进行数据流分析，得到从前驱进入该基本块的变量和被该基本块后继需要的变量。对于活跃进入该

基本块的变量，本项目中假定它们在之前的基本块内寄存器分配时被分配了寄存器，在出口处不活跃的变量，只需活跃到最后一次使用它的指令的地方即可，否则，应该在基本块结束时确保它们已经被分配到寄存器中。按照上述规定，该案例的所有变量的活跃期如图 6.2

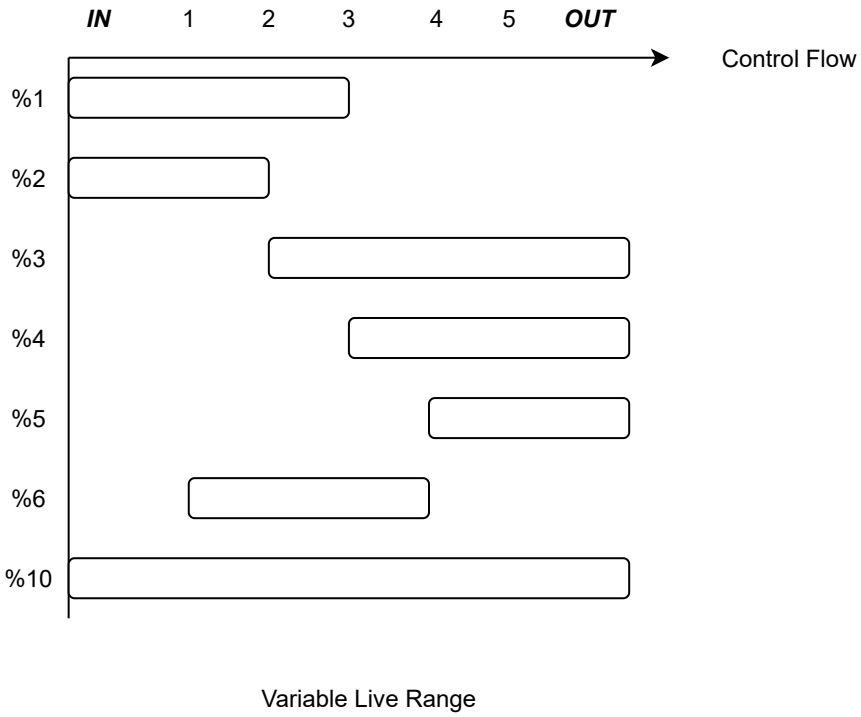


图 6.2　图 6.1中的虚拟变量关于指令次序的存活期

注：横轴代表了程序的指令执行次序，纵轴代表了虚拟变量，在程序执行到任意时刻作一条平行于纵轴的直线，穿过的变量即此刻互相冲突的变量，冲突的变量任何时刻都不应该被分配到同一寄存器上。

变量%1 和%2：由于%1 在出口处不活跃，所以只需要从活跃到最后一条使用%1 的指令，即第三条指令处；变量%2 与之类似。

变量%3,%4 和%5，这些变量都在出口处活跃，所以他们的活跃期应该是自赋值指令起，到基本块的。

变量%6 属于基本块内定义的变量，而且不在出口处活跃，所以只需要从定义处活跃到最后一次使用%6 的指令，即第四条指令处.

变量%10 同时在入口处和出口处活跃，因而在寄存器充分的情况下可以将其一直保留在寄存器中，否则，至少需要一对存取操作。

由此，自上而下的分析每条指令，该指令的左值应该与此时所有活跃的变量冲突，构建得到冲突图，如图 6.3。

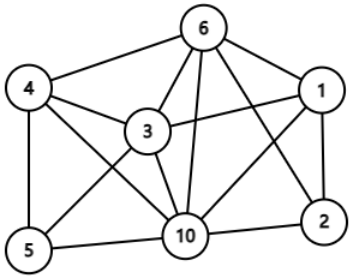


图 6.3　冲突图

按照算法，逐个把度数最小的节点入栈，这些度数小于 k 的节点一定可以 k 染色（k 是可用颜色数，现假定为 4 种颜色：红黄蓝绿）。节点 5 的度数为 3，将其入栈，得到图 6.4。

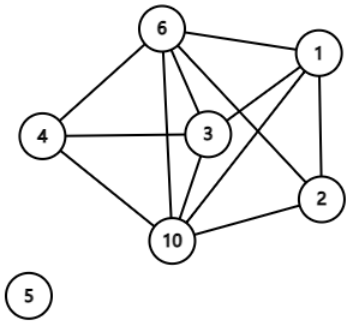


图 6.4　删去节点 5

当删去 5 的邻边后，节点 4 成为度数最小的节点之一，删去节点 4，此时的冲突图如图 6.5:

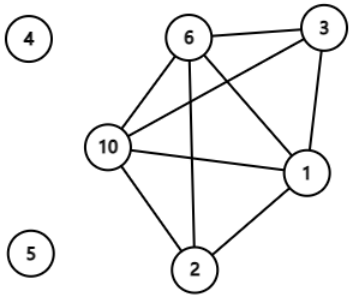


图 6.5　删去节点 4

同理，删去节点 3 得到图 6.6，此时的冲突图是一个 4-完全图。

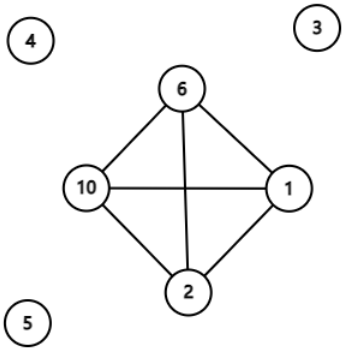


图 6.6　删去节点 3

因为我们有 4 个可用寄存器，因此这些节点都可以被染色，不妨按照红黄蓝绿的次序给节点 1，2，6，10 分别染色，依次弹出栈中元素 3，4，5。对之前弹出的节点进行染色，节点 3 与节点 6，10 冲突，因此可选颜色为红色或黄色，不妨假定为红色；节点 4 与 3，6，10 冲突，只能为黄色；节点 5 与 3，4，10 冲突，只能为蓝色，至此一种 4 染色的方案就完成了。

回到图 4.6 中的完全图，如果可用寄存器数目 k 是 3 呢？那么此时无论如何都不可能对该 4-完全图进行 3 染色，只能选择合适的节点抛出。

第七章　总结

在本文中，简要的对编译器中的中间表示的组织方式，经典优化和由 IR 向机器码的翻译及与指令集架构密切相关的优化等话题做了简要的介绍和举例说明。在笔者进行毕业设计的两个月里，深感自己曾经在课堂上所学过的编译原理知识只是入门性质的介绍，距离实际的现代编译原理和高性能编译器设计还有着很大的差距，因此笔者参考了很多国内外研究生课堂的课件和教材，同时笔者也复习了图论中的很多经典算法，发现计算机科学与离散数学有着千丝万缕的密切联系，最后还认识到，现代的程序员不仅要理解编译器，还要理解计算机的体系结构，只有这样才能写出充分利用硬件性能的好程序。

由于篇幅时间有限，而且笔者才学疏浅，因此文中存在说明不当或者有误之处在所难免，但是本文可以作为后续的本科编译原理教学实验内容的一个简单参考文档，或者作为一个编译器项目的学习笔记，可以为简单但功能俱全且性能优异的编译器设计提供参考思路。

再次对我的指导老师徐伟老师表示诚挚的感谢，徐老师在毕业设计期间为我们提供了很多的专业指导与参考资料。不仅在学术上如此，徐老师还关心帮助学生的生活和学业方向，是位非常好的老师。

参 考 文 献

[1] ROSEN B K, WEGMAN M N, ZADECK F K. Global value numbers and redundant computations[C/OL]//POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA: Association for Computing Machinery, 1988: 12–27. https://doi.org/10.1145/73560.73562.

[2] SHI Y, CASEY K, ERTL M A, et al. Virtual machine showdown: Stack versus registers[J/OL]. ACM Trans. Archit. Code Optim., 2008, 4(4). https://doi.org/10.1145/1328195.1328197.

[3] LENGAUER T, TARJAN R E. A fast algorithm for finding dominators in a flowgraph[J/OL]. ACM Trans. Program. Lang. Syst., 1979, 1(1): 121–141. https://doi.org/10.1145/357062.357071.

[4] DE SUTTER B, VAN PUT L, DE BOSSCHERE K. A practical interprocedural dominance algorithm[J/OL]. ACM Trans. Program. Lang. Syst., 2007, 29(4): 19–es. https://doi.org/10.1145/1255450.1255452.

[5] CYTRON R, FERRANTE J, ROSEN B K, et al. Efficiently computing static single assignment form and the control dependence graph[J/OL]. ACM Trans. Program. Lang. Syst., 1991, 13(4): 451–490. https://doi.org/10.1145/115372.115320.

[6] TARJAN R. Depth-first search and linear graph algorithms[J/OL]. SIAM Journal on Computing, 1972, 1(2): 146-160. https://doi.org/10.1137/0201010.

[7] BRIGGS P, COOPER K D, TORCZON L. Improvements to graph coloring register allocation[J/OL]. ACM Trans. Program. Lang. Syst., 1994, 16(3): 428–455. https://doi.org/10.1145/177492.177575.