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）

作品设计和开发文档

队伍名称：GPT 5.0

作　　者：罗钰浩、郭闰航、钟震、王南钦

填写日期：2024 年 8 月 7 日

目 录

摘 要

本文档详细介绍了一个用于 SySY 语言编译的编译系统的设计与实现。SySY 语言是 C 语言的一个子集，广泛用于教学和竞赛中。本编译系统的设计目标是实现一个高效、稳定且易于扩展的编译器，能够将 SySY 代码转换为 RISC-V 目标汇编代码。

所实现的编译器是一个针对 SySY 语言子集的编译器，它以 SySY 语言源代码作为输入，以 RISC-V指令集作为输出。它以 Flex 和 Bison++作为词法分析与语法分析工具，通过遍历 AST 生成对应的中间代码，再通过 Mem2reg 过程生成 SSA 形式的 IR 以进行活跃性分析，Phi 函数插入，死代码消除等优化，最后再转换成接近汇编语言的 MIR，通过寄存器分配和一系列的操作，最终生成 RISC-V 汇编代码。

关键词：SySY　　中间代码优化　　RISC-V　　编译系统

第一章 项目目的：实现对 SySY 语言的编译器

1.1 SySY 语言简介

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。部分 SySY 运行时库函数的参数类型会超出 SySY 支持的数据类型，如可以为字符串。SySY 编译器需要能处理这种情况，将 SySY 程序中这样的参数正确地传递给 SySY 运行时库。有关在 SysY 程序中可以使用哪些库函数，请参见 SySY 运行时库文档。

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（!、&&、||），非 0 表示真、0 表示假，而关系运算或逻辑运算的结果用 1 表示真、0 表示假。算符的优先级和结合性以及计算规则(含逻辑运算的“短路计算”)与 C 语言一致。

主函数的定义，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。

1.2 实现平台与运行平台简介

在 Linux 环境下进行编译，采用 RISC-V 架构，所有文件采用 utf-8 编码。利用

Flex 实现词法分析；利用 Bison++进行语法分析。

1.3 最终目的

通过实现一个可以把类似 C 语言的源代码转变为汇编代码的编译器，更好地理解编译的过程，锻炼我们的编程能力，合作能力。通过使用 GitLab 平台在线管理代码，我们能够更好地协同工作，并保持代码的版本控制。这种合作方式不仅促进了个体技能的提升，还能够加速整个项目的进展。

第二章 需求分析及实现方法

2.1 需求分析

给出 Sysy 语言的词法和语法定义，并根据对应的语法定义写出一些属性文法和语法制导。根据词法和语法的定义，构造一个编译程序，它主要可以完成如下功能：

（1）读入某个已经编辑好的 SySY 源程序文件，通过词法分析器，生成二元组，同时检查词法错误；

（2）语法分析器将产生的二元组作为输入，进行语法分析，同时检查语法错误；

（3）在语法分析同时，利用属性文法和语法制导技术，产生具体的语意动作，并对符号表进行操作；

（4）语义动作产生整个源程序的四元式序列；

（5）将产生的四元式序列连同符号表一起输出，作为编译程序的最终输出结果；

（6）对最后的代码优化和目标代码生成要有所考虑，必须留有一定的接口供以后扩展；

（7）增大程序的可移植性，努力做到整个系统方便移植。

2.1.1 引言

本需求分析报告介绍了 SySY 编译器的基本功能、用户界面和设计目标，旨在帮助用户初步了解该编译器的特点和用途。

2.1.2 编写目的

编写此报告的目的是为了定义 SySY 编译器的基本需求和范围，明确开发方向，并为开发团队和相关利益方提供清晰的指导。

2.1.3 技术风险

SySY 编译器的性能和稳定性可能会受到所选编程语言和编译优化算法的影响。需要在不同平台上支持目标代码生成，这可能引入一些平台特定的技术挑战。

2.1.4 文档约定

正文风格：宋体、小四
一级标题：宋体、四号
二级标题：宋体、四号

2.1.5 预期读者和阅读建议

开发人员：仔细阅读功能需求、设计规范和系统架构，以便进行编译器的设计和实现。

测试人员：重点关注功能需求、性能要求和测试用例，以确保编译器的质量和稳定性。

项目管理人员：了解项目的范围、进度和风险，以便进行项目规划和监控。

2.1.6 综合描述

SySy 编译器是一个用于将高级编程语言代码编译成中间代码和目标代码的工具。它的设计目标是提供高性能、可扩展性和可移植性的编译器框架。

2.1.7 产品的功能

我们的 Sysy 编译器具有以下主要功能：支持 SySy 语言，他是一个 C 语言的子集，具有多维数组，条件语句等多种 C 语法；提供多种编译优化选项，包括代码优化、内存优化和性能优化；可以生成类 LLVM 的中间表示代码（LLVM IR），用于进一步的优化和目标代码生成；支持 RISC-V 平台的目标代码生成。

2.1.8 假设和约束(依赖)

需要在支持 SySY 编译器的操作系统上运行，如 Linux、Windows 等;开发人员需要熟悉 SySY 编译器的架构和 API，以进行扩展和定制;编译器的性能和稳定性可能受到所选编程语言和编译优化算法的影响;这个示例涵盖了 SySY 编译器的基本需求和范围，以及与项目相关的风险、文档约定、预期读者和用户类。具体的需求和功能可以根据项目的实际情况进行进一步细化和规划。

2.2 实现方法

2.2.1 具体实现方法

利用 Flex 进行词法分析、Bison++进行语法分析、语义分析与中间代码产生、优化、目标代码生成。

2.2.2 小组合作方式

1. 使用了 GitLab 作为代码托管平台。每隔一段时间进行一次组会分享学习、发进度。
2. 团队协作流程上：设定固定的代码提交规范，确保代码库的整洁和一致性，同时制定明确的任务分配机制，以确保每个成员都明确自己的责任和期限。
3. 代码规范上：定期进行代码审查，确保代码质量，促进知识共享；使用 GitLab 的 Pull Request 功能进行代码审查，以便于追踪和讨论代码变更。
4. 定期组织技术培训和学习活动，促进团队成员间的知识共享和技能提升。
5. 在 GitLab 上采用分支管理策略，例如使用 feature 分支进行新功能开发，develop 分支用于日常开发，master 分支用于发布稳定版本。

第三章 项目设计概览

3.1 前端

前端主要通过 Yacc 工具链中的 Flex 和 Bison 完成词法分析和语法分析后，使用 codegen 遍历 AST 生成类 LLVM IR。前端的具体流程如图 3-1-1 所示。

图 3-1-1

3.2 中端

3.2.1 中端结构设计

中端 IR 使用类似于 Dancing Links 的数据结构维护中端优化所需的 Use-Def 关系 User 通过 uselist 找到 Use-Def 关系，从而找到被使用者；Value 通过 userlist 找到 Use-Def 关系，从而找到使用者。中端 IR 的基本数据结构关系如图 3-2-1 所示

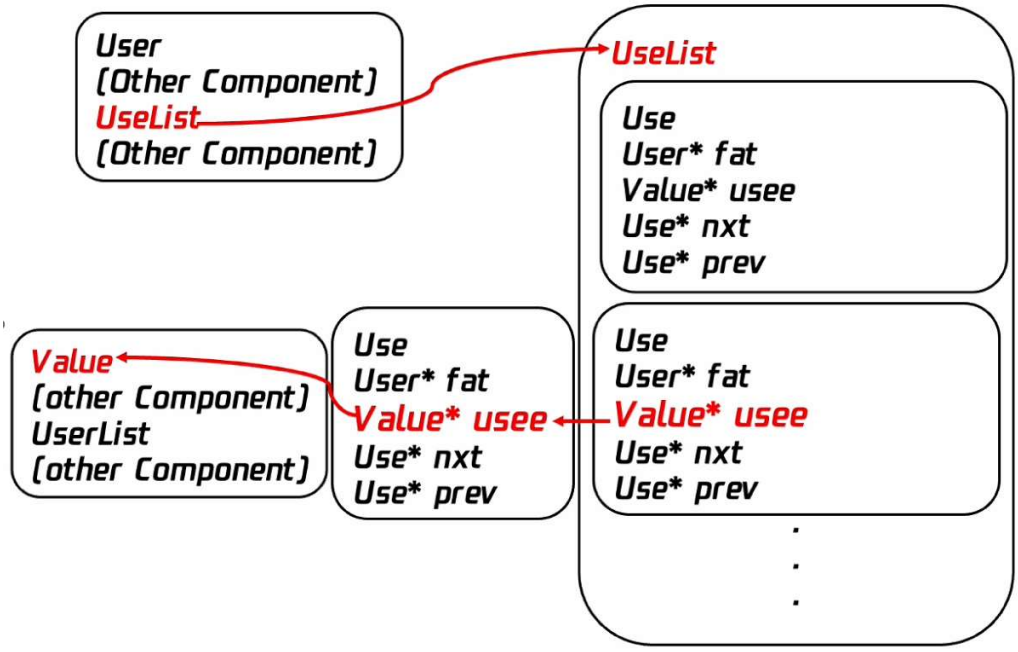


图 3-2-1

3.2.2 中端优化设计

分析类 Pass

- AliasAnalysis
- CondProbAnalysis
- DomiantAnalysis
- LoopAnalysis
- SideEffect

常规 Pass

- Mem2Reg
- ConstProp
- DCE
- CSE
- DeadArgsElimination
- DSE
- GepCombine
- GepEvaluate
- InstructionSimplify
- Reassociate
- PRE

控制流 Pass

- BlockMerge
- Simplifycfg
- CondMerge
- DealCriticalEdge
- Inline
- TailRecurseElimination

全局变量 Pass

- Global2Local
- StoreOnlyGlobalElimination
- Local2Global

循环 Pass

- LoopSimplify

- LCSSA
- LoopRotate
- LoopDeletion
- Licm
- LoopUnroll

并行 Pass

3.3 后端

3.3.1 整体结构设计

后端整体结构设计如下图 3-3-1 所示，主要分为四个部分：寄存器实现，指令、基本块、翻译结构和翻译流程。其中寄存器实现，指令、基本块、翻译结构主要用于实现后端代码及翻译的结构支持，翻译流程主要体现中端代码到目标代码的转换过程。

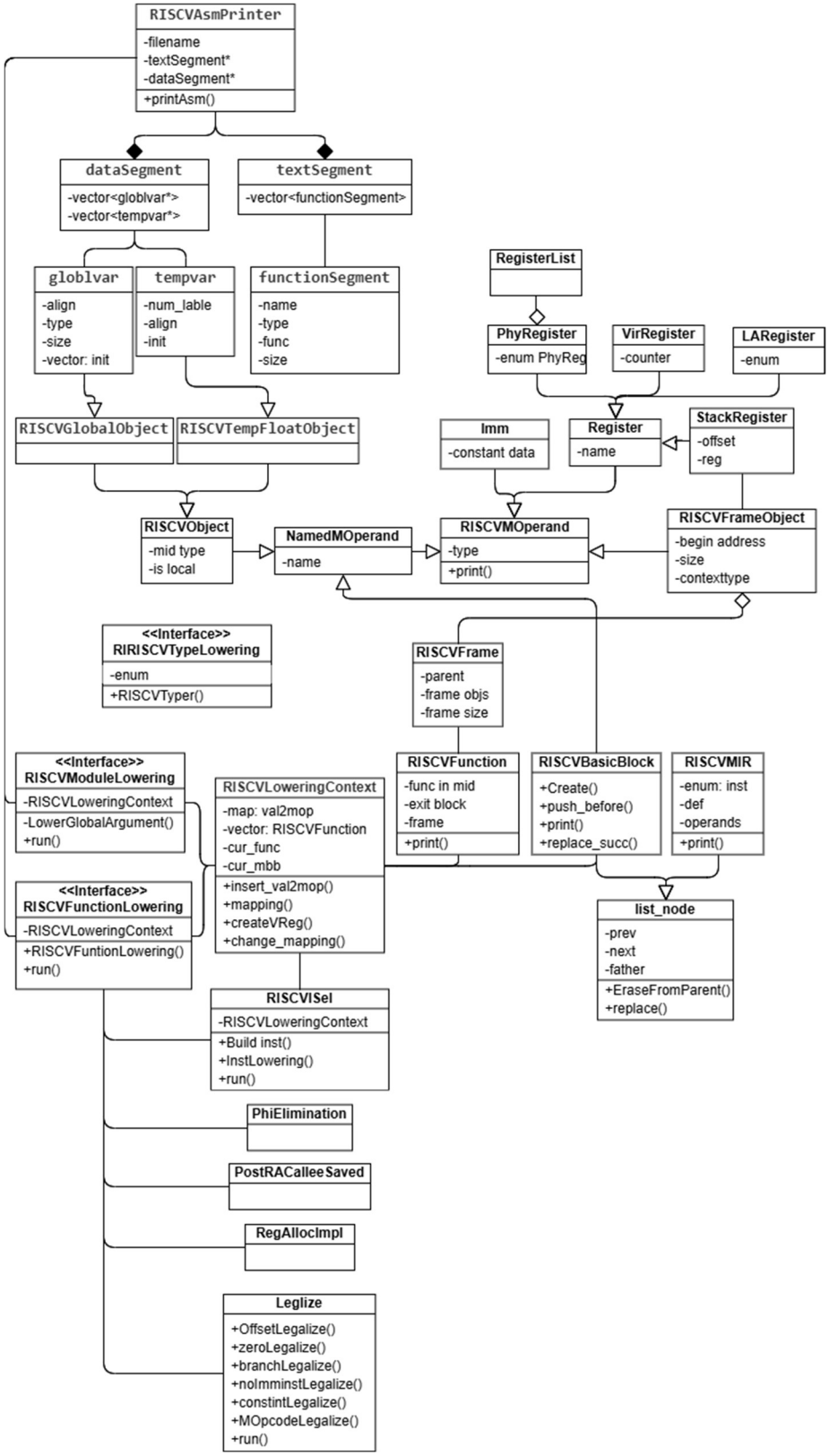

RISCVAsmPrinter
-filename
-textSegment*
-dataSegment*
+printAsm()
dataSegment
-vector<globlvar*>
-vector<tempvar*>
textSegment
-vector<functionSegment>
globlvar
-align
-type
-size
-vector: init
tempvar
-num_lable
-align
-init
functionSegment
-name
-type
-func
-size
RegisterList
PhyRegister
-enum PhyReg
VirRegister
-counter
LARegister
-enum
RISCVGlobalObject
RISCVTempFloatObject
Imm
-constant data
Register
-name
StackRegister
-offset
-reg
RISCVObject
-mid type
-is local
NamedMOperand
-name
RISCVMOperand
-type
+print()
RISCVFrameObject
-begin address
-size
-contexttype
<<Interface>>
RIRISCVTypeLowering
-enum
+RISCVTyper()
RISCVFrame
-parent
-frame objs
-frame size
<<Interface>>
RISCVModuleLowering
-RISCVLoweringContext
-LowerGlobalArgument()
+run()
RISCVLoweringContext
-map: val2mop
-vector: RISCVFunction
-cur_func
-cur_mbb
+insert_val2mop()
+mapping()
+createVReg()
+change_mapping()
RISCVFunction
-func in mid
-exit block
-frame
+print()
RISCVBasicBlock
+Create()
+push_before()
+print()
+replace_succ()
RISCVMIR
-enum: inst
-def
-operands
+print()
<<Interface>>
RISCVFunctionLowering
-RISCVLoweringContext
+RISCVFuntionLowering()
+run()
list_node
-prev
-next
-father
+EraseFromParent()
+replace()
RISCVISel
-RISCVLoweringContext
+Build inst()
+InstLowering()
+run()
PhiElimination
PostRACalleeSaved
RegAllocImpl
Leglize
+OffsetLegalize()
+zeroLegalize()
+branchLegalize()
+noImminstLegalize()
+constintLegalize()
+MOpcodeLegalize()
+run()

图 3-3-1

3.3.2 寄存器实现设计

下图为寄存器设计结构图 3-3-2，其中最重要也是最基本的类为 RISCVMOperand ，后端所有可以操作的对象都直接或间接继承自该类，与 Linux 中“Everything is file”，中端“Everything is value”类似，在后端中 Everything is MOperand。

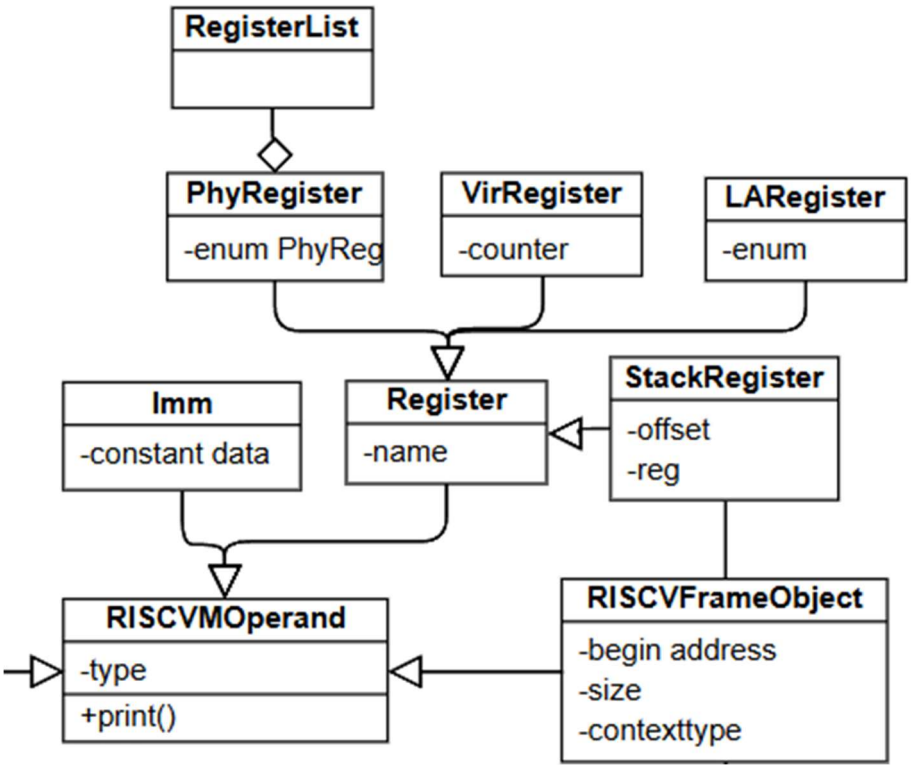


图 3-3-2

为了应对 RISCV 目标代码中操作数的不同需求，设计了以下一种立即数和四种寄存器。

Imm 类，其作用是保存直接以立即数形式存在的数据，包括整形常量和 32 位浮点常量。当然我们对于浮点常量的加载有着特殊的规则，这点将在第四章中详细讲解。

Register 类，作为所有种类寄存器的父类存在，给予他们统一的属性。

PhyRegister 类为物理寄存器，表现所有形如 a0，fa0，s0，sp 等形式的寄存器，他们在内存中唯一存在；VirRegister 类为虚拟寄存器，我们假定有无限多的寄存器可以使用，使用计数器进行计数，其表现形式为%1，%2。LARegister 类服务于加载标签地址时使用，形如%hi(lable)，%lo(lable)(reg)，通过两条指令分别加载地址的高位与低位实现真正的加载地址；StackRegister 类为栈帧寻址所用寄存器，形如 offste(reg)，其中 offset 为相对于基址，也即是 reg 的偏移值。

3.3.3 指令，基本块，函数实现设计

下图为指令、基本块、函数实现设计图 3-3-3。RISCVMIR、RISCVBasicBlock、RISCVFunction，均继承了一个统一的链表结构 list_node，实现 RISCVFunction->RISCVBasicBlock->RISCVMIR 的存储查询链。

每个 RISCVFunction 中均存在一个栈帧结构 RISCVFrame，用于保存该函数在运行时的数据；该栈帧中存在多个栈中对象 RISCVFrameObject，在栈帧中统一管理。

RISCVBasicBlock 中除了常规基本块外，还有单独设计了 entry 块以及 exit 块。

RISCVMIR 中保存一条 RISCV 指令所有需要的信息：操作码、源寄存器、目的寄存器。

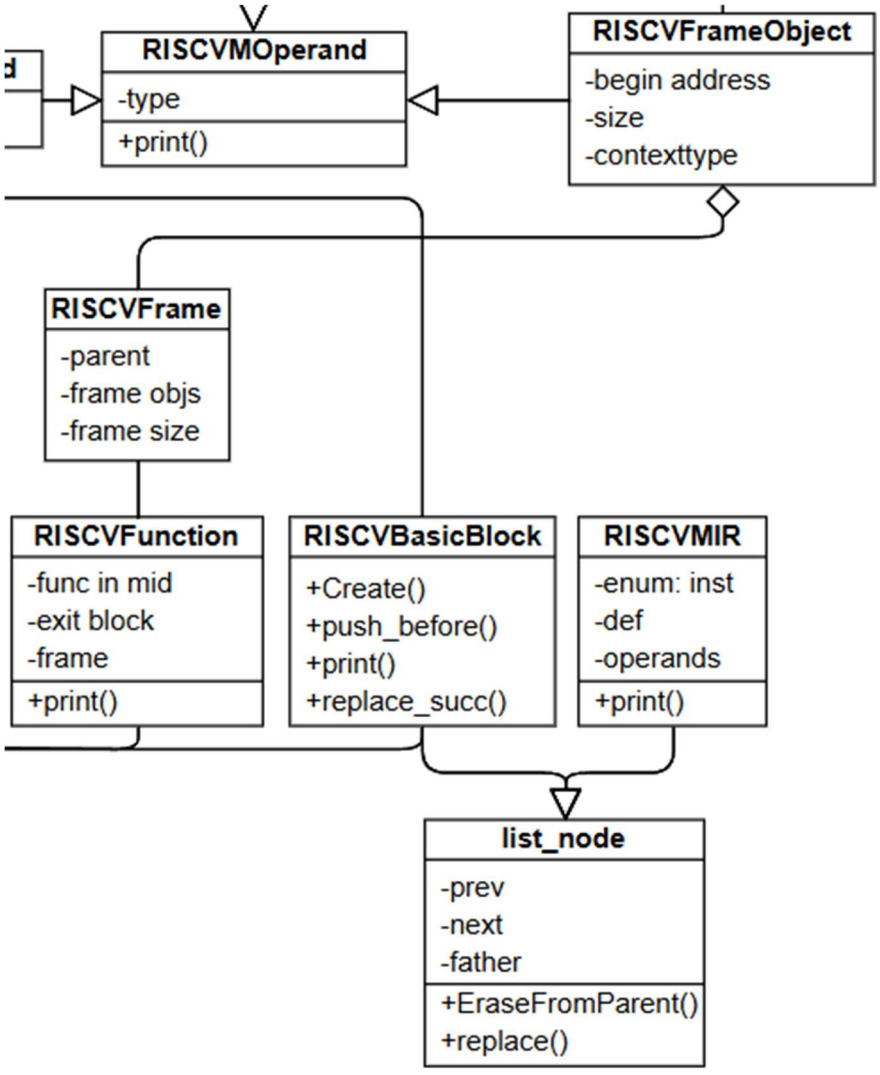


图 3-3-3

3.3.4 翻译结构设计

下图为翻译结构设计图 3-3-4，主要设计并实现了 RISCVAsmPrinter 类。该类的作用是：整合所有代码的信息，完成代码、数据分段，打印等工作。

RISCVAsmPrinter 类由 dataSegment 类和 textSegment 类聚合而成，dataSegment 类保存了全局变量、浮点常量等数据；textSegment 类主要保存了函数的代码与指令。Globvar 与 tempvar 类为实现数据保存的具体实现。FunctionSegment 则保存了所有函数的信息。

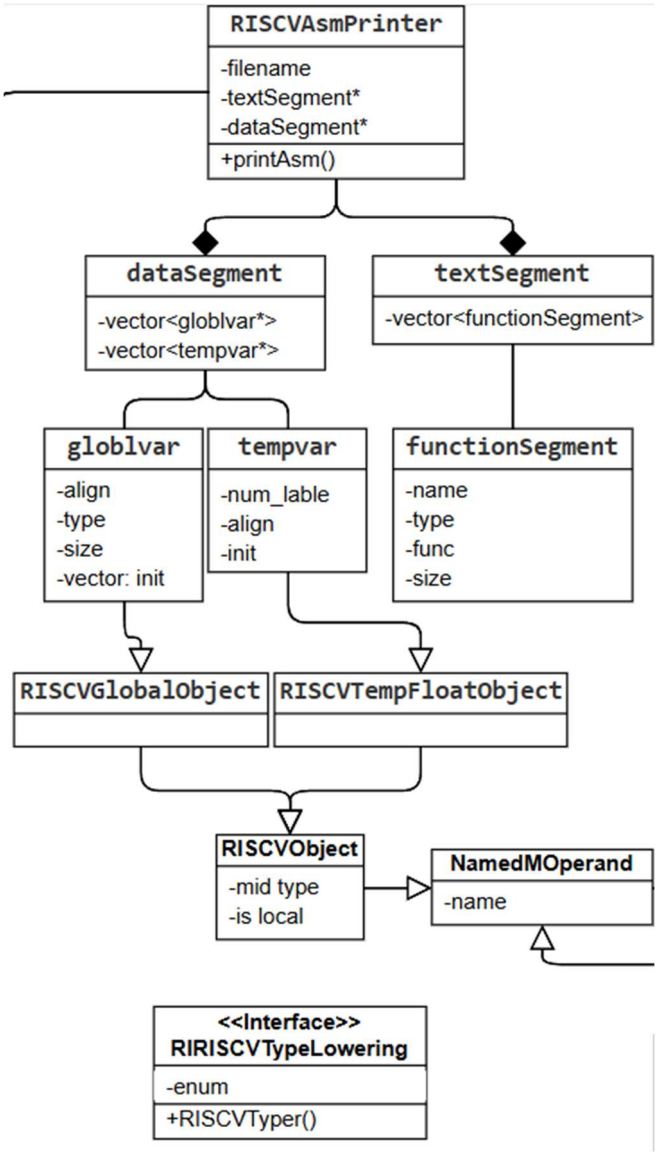


图 3-3-4

3.3.5 翻译流程设计

下图为翻译流程设计图 3-3-5，通过 RISCVModuleLowering 这个接口开始真正的翻译工作。RISCVFunctionLowering 则是以函数为单位进行翻译，包括指令选择，寄存器分配，创建栈帧，合法化等工作。RISCVLoweringContext 类则表示当前翻译工作中正在翻译的函数或基本块或指令的上下文。

指令选择的作用是将一条来自中端的代码翻译成为后端目标指令；寄存器分配的作用为指令中所有虚拟寄存器分配物理寄存器；栈帧生成作用为每个函数生成确定的栈帧；合法化是将翻译过程中产生的不合法指令转换为合法指令。

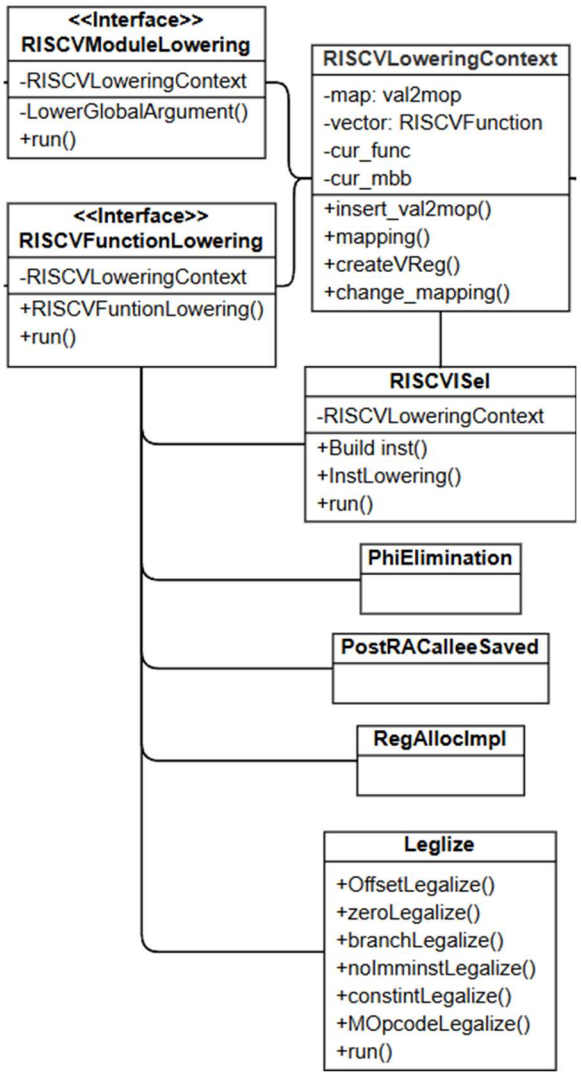


图 3-3-5

至此我们完成后端所有规定流程，得到了正确格式以及保证了正确性的目标汇编代码。

第四章 实现过程

4.1 基础测试用代码样例

代码 4-1 基础测试用代码样例

```
// test if-{if-else}
int if_ifElse_() {
    int a;
    a = 5;
    int b;
    b = 10;
    if(a == 5){
        if (b == 10)
            a = 25;
        else
            a = a + 15;
    }
    return (a);
}

int main(){
    return (if_ifElse_());
}
```

4.2 AST 结构说明

AST 提供了一种抽象的、与具体编程语言无关的表示形式，它捕捉了程序的结构和语义信息，方便编译器和解释器进行进一步的分析和处理。在编译过程中，AST 可用于进行语法检查、类型检查、代码优化、代码生成等操作。在实验中，语法树的主要目的是为了方便构建中间代码。实验中采用多叉树+基类指针的方式存储 AST 节点信息，通过递归调用 codegen（针对 CompUnit 和 FuncDef）、GetInst

（针对函数体中的语句）、GetOperand（针对表达式），即可完成中间代码生成。图 4-5-1 以 CompUnit 为例，有代表性的展示了 AST 的结构。

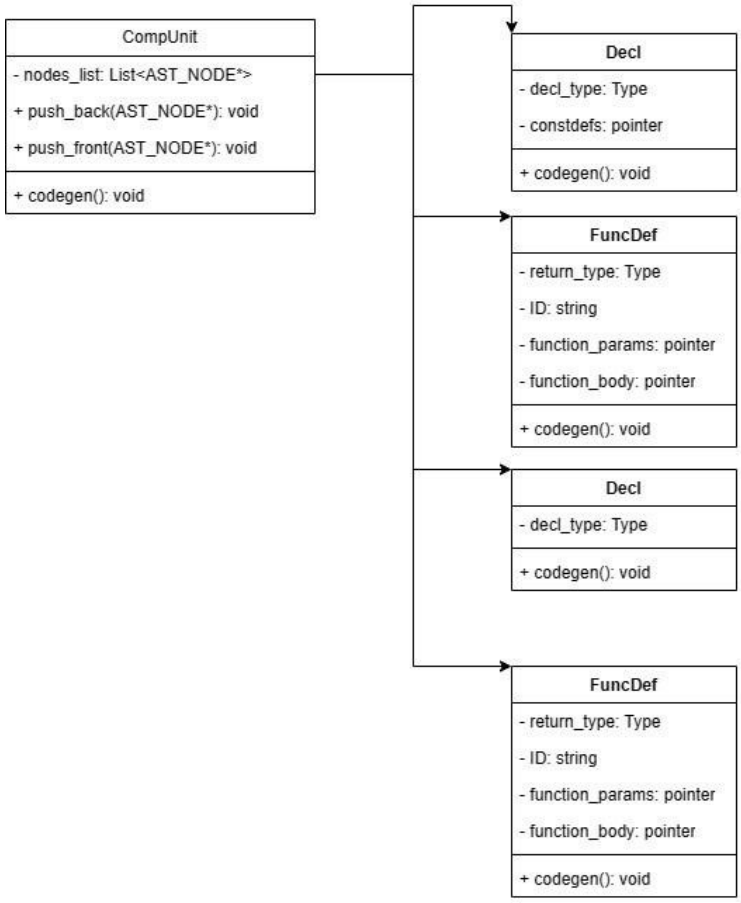


图 4-2-1 以 CompUnit 为例

4.3 符号表

4.3.1 符号表说明

符号表是一种数据结构。符号表由名字与属性域构成，在符号表中，程序源代码中的每个标识符都和它的声明或使用信息绑定在一起，比如其数据类型、作用域以及内存地址。在编译程序工作的过程中需要不断收集、记录和使用源程序中一些语法符号的类型和特征等相关信息。

4.3.2 符号表设计

符号表由一个(Key 为 name,Value 为指向存储中间代码中 value 指针类型的栈的指针)的哈希表和层级记录器 rec 组成。通过层级记录器，完成多层嵌套下的作用域问题，即作用域开始 rec 开始记录有合法添加的栈的指针，层级结束后 rec 通过记录的指针访问对应的栈，将其一一删除。

代码 4-3-1 符号表设计

```
#pragma once
#include "BaseCFG.hpp"
#include <map>
#include <stack>
#include <string>
class SymbolTable
{
    protected:
    using recoder=std::stack<Value*>*;
    std::map<std::string,std::unique_ptr<std::stack<Value*>>> mp;
    std::vector<std::vector<recoder>> rec;
    public:
    void layer_increase(){
        /*添加一层，用于记录这一层所有增加过的变量栈的指针*/
        rec.push_back(std::vector<recoder>());
    }
    void layer_decrease(){
        /*这一层结束之后，通过访问栈的指针，
将变量名与中间代码对应的 Value 解除绑定*/
        for(auto &i:rec.back())
            while(!i->empty())i->pop();
        rec.pop_back();
    }
    Value* GetValueByName(std::string name){
        auto &i=mp[name];
        assert(i!=nullptr&&!i->empty());
        return i->top();
    }
    void Register(std::string name,Value* val){
        auto &i=mp[name];
```

```
        if(i==nullptr)i.reset(new std::stack<Value*>());
        if(!rec.empty())rec.back().push_back(i.get());
        i->push(val);
    }
};
```

4.4 中间代码生成

4.4.1 中间代码说明

在进行了语法分析和语义分析阶段的工作之后，有的编译程序将源程序变成一种内部表示形式，这种内部表示形式叫做中间语言或中间表示或中间代码。所谓“中间代码”是一种结构简单、含义明确的记号系统，这种记号系统复杂性介于源程序语言和机器语言之间，容易将它翻译成目标代码。另外，还可以在中间代码一级进行与机器无关的优化。产生中间代码的过程叫中间代码生成。

定义一种语言除了要求定义语法外，还要求定义语义，即对语言的各种语法单位赋予具体的意义。语义分析的任务是首先对每种语法单位进行静态的语义审查，然后分析其含义，并用另一种语言形式，即比源语言更加接近于目标语言的一种中间代码来进行描述这种语言。因此，中间代码就显得十分重要，它关系着整个程序语言的正确编译与否，同时也是进行下一步编译的重要先决条件。

4.4.2 生成类 SSA 的中间代码测试用例代码生成的中间代码

中间代码的生成就是对抽象语法树的遍历过程，在遍历过程中需要的变量属性都从符号表中读取。

中间代码生成最大程度上参照了 llvm 的处理方式，生成的是类 SSA 形式的 IR，这使得下一步的优化更容易，并且前端最终生成的前端代码可使用 llvm 工具链进行检验，极大简化了后期测试。

为使生成类 SSA 的中间代码，依照 llvm 创建了 Alloca、Store、Load 等指令。包括 Alloca：声明变量，使用 Alloca 指令在栈帧上分配局部变量，得到一个指向该变量的指针；Store：把虚拟寄存器的值写主存；Load：把值读出为 SSA value，使每个变量都只被赋值一次；Binary：双操作数操作（单操作数操作可以在 ir 阶段变为双操作数操作）；GEP： 完成指针内存偏移的指令；FPTSI/SITFP：完成类型转换；br：branch 语句；Call/Ret 语句等指令。

4.4.3 生成中间代码的函数

生成中间代码主要通过遍历 AST 递归调用 codegen（针对 CompUnit,FuncDef 等）、GetInst（针对一系列 statement）和 GetOperand（针对一系列表达式），并在执行过程中直接完成基本块划分。

4.4.4 中间代码的设计

我们通过遍历 AST 得到的结果应该是一个 TAC 链式结构，通过分析和研究 LLVM 的内存设计，我们实现了主要的数据结构，包括但不局限于 Type、Value、User、Instruction、BasicBlock、Function 的数据结构。

在 LLVM 当中几乎所有的类都是 Value，在我们的数据结构中它用来承载基础的信息。如果一个 Value 被使用了那么使用它的数据结构我们称为 User，同时注意，User 很可能也会在其他地方作为 Value 使用，所以他需要继承于 Value 类。

同时根据 LLVM 的 Use-Def 设计，我们还需要建立一个 Use 类来管理 Value 和 User 的这种关系，它就相当于一条边连接 Value 和 User。

具体的内存形式如下图 4-4-1 所示：

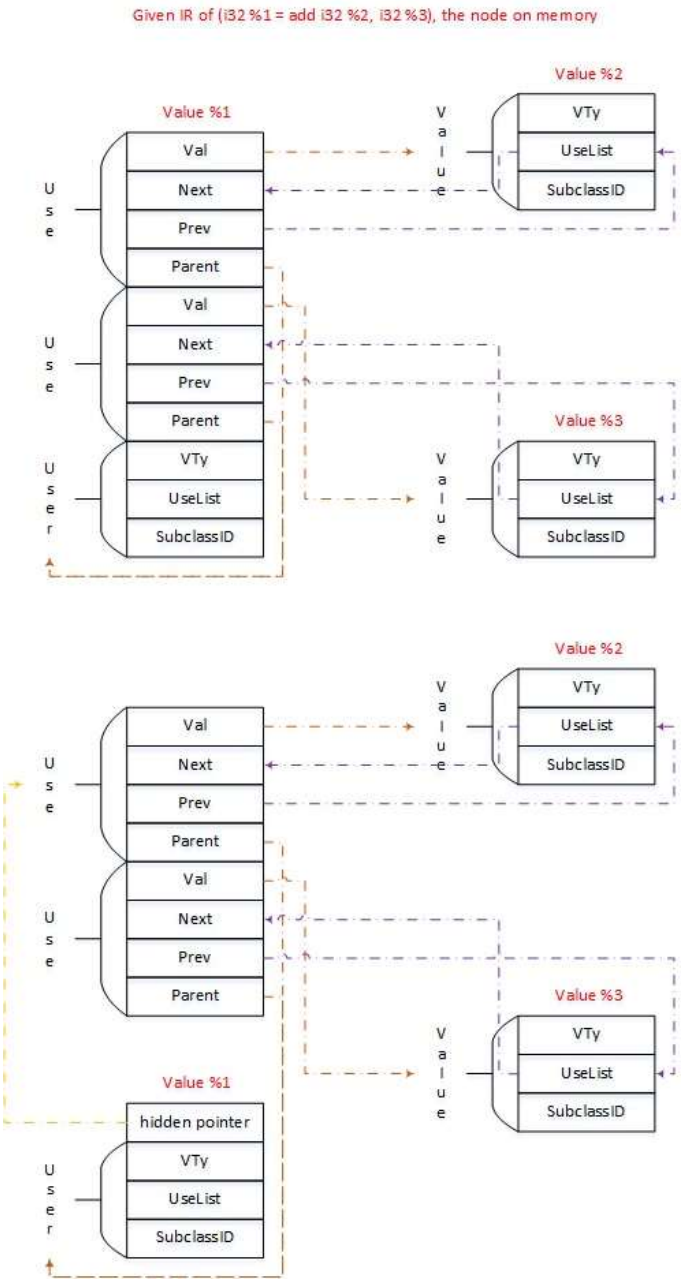


图 4-4-1 Use-Def 的内存关系

通过上图我们可以看到一个 Instruction 包含了多个 Use、一个 User、并通过 Use 连接了多个 Value 对象。

4.4.5 中间代码重要接口及其实现示例

介绍完主要的数据结构，代码 4-9 以基本的 Value、Use、User 和 AllocaInst 为例，介绍重要接口及其实现。注意代码中只展示了重要接口。

代码 4-4-1 中间代码的重要接口及其实现

```
//注意代码中只展示了重要接口。
class Use
{
    ......
    /// @brief 使用者
    User* fat=nullptr;
    /// @brief 被使用者
    Value* usee=nullptr;
    /// @brief 下一个 Use
    Use* nxt=nullptr;
    /// @brief 管理这个 Use 的指针
    Use** prev=nullptr;
    public:
    /// @brief User 删除 def-use 关系
    void RemoveFromUserList(User* is_valid);
    ......
};
class Value
{
    ......
    UserList userlist;
    protected:
    public:
    /// @brief 为一个 Value 添加 User
    void add_user(Use* __data);
    ......
};
class User:public Value
{
    ......
    using UsePtr=std::unique_ptr<Use>;
    protected:
    /// @brief 存储 Use 指针的 vector
```

```
    std::vector<UsePtr> uselist;
    /// @brief 为一个 User 以新增 Use 的方式添加用到的 Value
    void add_use(Value* __data);  ......};
```

4.5 中间代码优化

4.5.1 Mem2Reg

SSA 是中间表示（IR）的属性，它要求每个变量只分配一次，并且每个变量在使用之前定义。

LVM 在前端生成的是内存形式的 SSA 也就是说当中 Mem2reg 整体流程:

1．LLVM 假设程序中所有的局部变量都在栈上，并且通过 Alloca 指令在函数的 Entry BB 进行声明，并且只出现在 Entry BB 里。但不同的前端可能会有特殊情况，但是就我们的目的机器而言，基本都是这样的。所以我们暂时接受这个假设。

2．对步骤 1 中声明的每个 ALLOC 指令，LLVM 都会检查它是否 promotable（检查是否可以转化成 SSA 形式，从而删掉这个 Alloca）。如果满足下面的 2 个条件，就是可 promotable：这个 ALLOC 出来的临时变量不在任何 volatile 指定中使用，这个变量是直接通过 LOAD 或者 STORE 进行访问的。比如，不会被取址（eg：当 AllocaInst 作为 StoreInst 的第一个操作数时，存储指令正在尝试将 AllocaInst 所分配的内存的地址存储到某个位置，即取址）

3. 对于步骤 2 中选出来的可 promotable 的那些局部变量，扫描一次当前函数，看下哪些基本块在使用这些 ALLOC，哪些基本块定义了这个 ALLOC（这里的使用是指 LOAD，定义是通过 STORE 语句）

4. 在这个过程中，可以执行一些基本的优化，比如：未使用的 ALLOC 变量，比如声明或定义的但没使用变量；如果只有一个定义基本块（只有一个 STORE 语句，且只存在一个基本块中），那么被这个定义基本块所支配的所有 LOAD 都要被替换为 STORE 语句中的那个右值；如果某个 ALLOC 出来的局部变量的读或者写都只存在一个基本块中，那么我们就没必要去遍历所有的 CFG 的，因为这个 store 语句支配了这些 LOAD，所以可以用 STORE 的右值直接替换使用 LOAD 指令的那些值。

5.构建支配树。

6.利用支配信息，来插入 phi 函数，具体做法是：对于每一个经过上述步骤筛选后的 ALLOC 临时变量，找到一组基本块，这些基本块只使用了这个临时变量，而没有定义它，这组基本块也可能不存在；通过一种线性时间复杂度的算法来求得 Phi 节点的插入点；将 Phi 节点插入到上一步找到的那些插入点，并且在 BasicBlock 的开头。

7.一旦所有 Phi 节点插入到位后，开始重命名 PASS。

以上板块为 SSA 化的重难点、需要对数据结构和算法有较高的理解。

首先我们遍历每个 function 初始的 BasicBlock 的指令 Inst，对于每个 AllocaInst，检查他是否可以 Promote：

随后开始进行一些轻度的优化（即进行步骤四）：对于未使用的 ALLOCA 变量，比如声明或定义的但没使用变量，直接删除这个 ALLOCA

如果只有一个定义基本块（只有一个 STORE 语句，且只存在一个基本块中），那么被这个定义基本块所支配的所有 LOAD 都要被替换为 STORE 语句中的那个右值。

如果某个 ALLOC 出来的局部变量的读或者写都只存在一个基本块中，那么我们就没必要去遍历所有的 CFG 的，因为这个 STORE 语句支配了这些 LOAD，所以可以用 STORE 的右值直接替换使用 LOAD 指令的那些值。

4.5.2 CSE

公共子表达式删除优化通过识别和消除程序中重复计算的相同子表达式来优化代码，从而减少不必要的计算，提升程序的执行效率。这个 Pass 采用数据流分析以及可用表达式分析的技术，利用集合Gens和Kills，使用数据流方程来表示这种可用性。 它采用下述数据流方程来计算出关于可用表达式的最大不动点：

$$in[n] = \bigcap_{p \in pred[n]} out[p]$$

$$out[n] = gen[n] \cup (in[n] - kill[n])$$

在代码实现上，主要目标是通过分析和优化消除函数中的公共子表达式，从而提高代码的运行效率。整个过程包括计算活跃变量信息，迭代直到信息稳定，然后

针对每个基本块执行优化，并删除不再需要的指令。

4.5.3 Reassociate

采用代码重组的主要目的是通过重新排列算术表达式中的操作顺序来减少浮点计算中的误差和提高编译器生成代码的效率，在这个 Pass 中我们将某些类型的算术运算重新关联，以减少计算错误并利用可能存在的常数合并或中间结果的重复使用，具体来说，它实现了下面的任务：

1. 通过代码重组重新排列加法和乘法运算，识别加法和乘法运算，并尝试重新排列这些运算以减少误差。例如，将(a + b) + c 重新排列为 a + (b + c)

2. 识别和合并常数，该传递会识别并合并常数表达式。例如，将 a + 2 + 3 重新排列并合并为 a + 5

3. 消除冗余运算：通过重新排列运算，该传递可以识别并消除一些冗余运算。例如，将(a * 1) + b 重新排列为 a + b, 去掉无效的乘法运算

在具体实现中通过将所有个 BasicBlock 组成 RPO 顺序进行遍历，对每一条指令进行 Rank 值标号，对于特殊的值比如常量或标号为 0，方便后续的判断。

通过代码的重组，我们可以简化很多冗余的指令。

4.5.4 LICM

循环不变量提取 Loop-Invariant Code Motion(LICM) 是一个重要优化技术，他的主要目的是通过将循环不变的代码从循环体中移出，从而减少循环中的重复计算，提高程序的执行效率。对于一些在循环中但不会变化的指令来说，如果将他放在循环中，每一次执行都会为他单独分配一个寄存器，将这些代码提取到循环之外，进而减少循环体中的指令数量，从而减少循环执行时的开销，提高整体程序的执行效率。这种优化对于深度嵌套的循环特别有效，因为它能显著减少循环中的重复计算。考虑一个简单的例子：

代码 4-5-1 LICM 示例

```
for (int i = 0; i < n; i++) {
    int x = a + b;
    arr[i] = x * i;
}
```

```
//经过 LICM 变为
int x = a + b;
for (int i = 0; i < n; i++) {
    arr[i] = x * i;
}
```

我们在具体的代码实现使用了两个函数来进行不变量提取：hoist 和 sink，即代码下沉和代码上提

4.5.5 Gloabl2Local

Global2Local 优化旨在将全局变量转化为局部变量，以减少全局变量的使用，从而提高程序的局部性，减少访问延迟并且潜在地提升性能。这个 Pass 的核心思想是将那些仅在单个函数中使用的全局变量转化为该函数的局部变量。它包含的工作包括：

- 检测递归函数，将全局变量转化为递归函数内的变量会出现问题
- 删除未使用的全局变量
- 如果全局变量仅在一个函数中使用，则将其转化为该函数的局部变量。
- 如果全局变量是数组类型，且在非递归函数中使用，则进行特殊处理。
- 需要为每一个全局变量在局部创建一个 alloca 指令，并替换相应的 UserList

4.6 RISC-V 目标代码生成

任务：把优化后的中间代码转换成 RISC-V 汇编代码，汇编器可以将汇编代码转会为机器码被 CPU 处理，遍历中间代码逐条翻译成汇编代码。我们期望的输入是中间代码，输出是 RISC-V 汇编指令代码。

4.6.1 RISC-V 目标代码说明

目标代码生成的是 RISC-V 汇编指令，RISC-V 是一个基于精简指令集（RISC）原则的开源指令集。其特殊之处在于它使用模块化的 ISA（指令集），而指令集分为基本部分和扩展部分，其中扩展部分又分为标准扩展和非标准扩展。“I” 基本整数集，其中包含整数的基本计算、Load/Store 和控制流，所有的硬件实现都必须

包含这一部分;“M”标准整数乘除法扩展集,增加了整数寄存器中的乘除法指令;“A” 标准操作原子扩展集，增加对储存器的原子读、写、修改和处理器间的同步;“F” 标准单精度浮点扩展集，增加了浮点寄存器、计算指令、L/S 指令;“D”标准双精度扩展集，扩展双精度浮点寄存器，双精度计算指令、L/S 指令。I+M+F+A+D 被缩写为“G”,共同组成通用的标量指令。本次基本实现了 RV64G 的大部分指令。

4.6.2 后端结构介绍

后端的工作主要是从中端代码到后端目标代码，为实现相应的功能需要对应的结构进行支持。本节将介绍在后端实现中比较重要的结构：RISCVModuleLowering 、 AsmPrinter 、 RISCVFunction 、 RISCVBasicBlock 、 RISCVMIR 、 RISCVMOperand 等。而一些细节的结构如 RISCVFrame 、RISCVFrameObject、PostRACalleeSavedLegalizer 等，将在后续的篇章中进行详细介绍。

RISCVModuleLowering

RISCVModuleLowering 类是作为中端代码翻译到目标代码的入口作用，在其中会调用 LowerGlobalArgument(), RISCVFunctionLowering.run()等函数，进行真正的翻译工作。

RISCVFunctionLowering

该类的作用主要是实现以函数为单位的翻译工作，包括处理库函数、指令选择入口、寄存器分配入口、函数栈帧生成入口、指令合法化入口。

RISCVAsmPrinter

RISCVAsmPrinter 该类的主要作用是作为对来自中端的代码获取特定信息：全局变量、临时变量相关信息，并作为打印目标代码的入口。在该类中就已经实现了汇编代码规定的不同段（text 段、data 段、bss 段）分类保存，中其中成员

textSegment* text 保存了 text 段所需信息，包括具体的函数与指令；成员 dataSegment* data 便保存了临时变量和全局变量，按照变量是否存在初始值存放于 data 段和 bss 段。

同时需要注意的是，我们加载全局变量和浮点临时变量的方式有别于普通变量。普通变量仅需要从栈帧中对应位置取出即可，但全局变量和浮点临时变量需要首先加载其地址，再从该地址中加载出真正的数据。当然写入数据时也是相同的步骤，首先得到地址再写入数据。为实现地址的加载需要 lui 指令，其作用是加载地址的高 20 位到特定寄存器中。为实现全局变量和临时变量的正确保存与加载，引入了 dataSegment 类，对于以上特殊变量进行相应的合法化操作。

RISCVFunciton

该类的实现基于已经实现的 mylist 结构，将一个 RISCVFunction 与多个 RISCVBasicBlock 对应起来，关于链表内部的修改操作已经再 mylist 这个通用的结构中实现。

该类的主要功能是保存一个 function 中所有的信息，包括其中有哪些基本块，特殊的 exit 块的处理，栈帧的维护等。同时根据 RISCVFunction 这个结构完成了记录调用该函数需要保存参数到栈帧的变量，以及被调用寄存器（Callee Register）的保存与恢复。

RISCVBasicBlock

RISCVBasicBlock 类其实现方式与 RISCVFunction 类似，均采用 mylist 结构来实现基本块与每一条指令的对应。

RISCVMIR

该类的主要作用是实现指令的功能，包括指令的操作符、左右操作数，同时指令集的定义也在该类中。该类仅提供了简单的创建、修改左右操作数以及获取该指令不同信息的函数。

RISCVMOperand

该类是后端所有数据共有的类（如 RISCVFunction、RISCVMIR、Register 等），与中端“ everything is value ”类似可以说 “ everything is RISCVMOperand ”。其子类有 NamedMOperand、RISCVObject、RISCVGlobalObject、RISCVTempFloatObject、RISCVFrameObject、Register、Imm 等。

其中 NamedMOperand 是有名操作数，RISCVMIR、RISCVBasicBlock、RISCVFunction 等就是此类；RISCVGlobalObject 用于保存全部变量和指向函数的指针等信息；RISCVTempFloatObject 用于保存浮点临时变量；RISCVFrameObject 用于保存栈帧中的变量的指针；Register 主要用于实现虚拟寄存器 VirRegister 和物理寄存器 PhyRegister。Imm 用于保存立即数。

在后端翻译工作中实际可以处理的所有对象都是 RISCVMOperand。

寄存器的实现

为了满足后端不同寄存器的需求，这里引入了不同种类的寄存器以应对不同需求。

物理寄存器 PhyRegister， 主要是实现了普通类型的物理寄存器，每个物理寄存器均全局唯一。根据 RISCV 提供的寄存器，分为通用寄存器和浮点寄存器，其中浮点寄存器仅用于浮点数的保存与运算。其中通用寄存器根据规定有不同用途：zero 寄存器硬编码恒为 0，ra 寄存器用于保存函数调用的返回地址，sp 寄存器用于保存堆栈指针，gp 寄存器用于保存全局指针，tp 寄存器用于保存线程指针，t0-t2 为临时寄存器，s0 寄存器是帧指针，s1 为保存寄存器，a0-a1 为函数参数或作为函数返回值，a2-a7 为函数参数，s2-s11 为保存寄存器，t3-t6 为临时寄存器。这些寄存器又分为 Caller 寄存器和 Callee 寄存器，分别由调用者和被调用者保存。

除了以上普通形式的寄存器还有用于加载地址，栈帧寻址的特殊寄存器，我们实现为 LARegister 与 StackRegister。其中 LARegister 形如%hi(lable), %lo(lable)，主要用作于加载地址的特殊寄存器；StackRegister 形如 12(sp), 24(s0), -4(t0)等，其作用是在栈帧中进行相对寻址，取出对应的数据。

4.6.3 指令选择

指令选择部分是后端工作的第一部分，将来自中端的代码进行初步的转化。指令选择部分采用了模式匹配加宏展开的形式实现，主要完成 IR 指令到 RISCV ISA 对应指令的转换，并实现 phi 函数的消除退出 SSA 形式。

常规指令的指令选择

对于普通的指令，我们可以进行一对一的翻译，仅需要将中端 IR 指令中的操作码对应到后端恰当的操作码、对所有的操作数均以虚拟寄存器表现。但考虑到一种中端 IR 的操作码可能会对应多种后端的操作码，还需要添加足够的判断条件以生成正确的后端目标代码。由于针对不同的代码均需要不同的逻辑完成，篇幅相对较长，此处仅以 store 指令为例进行指令选择。

针对于 Store 指令的指令选择，主要做了如下几个工作:通过中端 IR 的操作码与该 IR 的操作数的类型进行判断，得到实际后端代码的操作码，如在此处中端 store 对应后端的操作码就有 sw, sd, fsw, fsd。同时还需考虑操作数是否为立即数，针对于立即数有不同的合法化处理，这点在代码合法化部分再详细介绍。

CondInst 的特殊处理

CondInst 即条件跳转指令，由于中端 IR 的条件跳转指令结构与后端条件跳转指令结构不同，这里进行了特殊处理，通过两条中端指令翻译为三条后端指令。我们需要将中端指令中的条件判断指令和跳转指令分开解析，将条件部分与判断为真的跳转合并为一条真正的后端指令，再生成一条无条件跳转语句实现判断为假时的跳转。

CallInst 的特殊处理

函数的调用需要从中端的不受限制的虚拟寄存器，到完全遵守调用规约的转变——使用物理寄存器，并将其余的参数按照调用规约放入栈上对应的位置。使用拷贝指令 mv 将作为参数的虚拟寄存器拷贝到物理寄存器，不仅可以避免参数与某个物理寄存器的“绑死”问题，同时将如何分配参数的寄存器重新交会了寄

存器分配算法的手中。在寄存器分配中消除冗余的 mv 指令。

RetInst 的特殊处理

在进行指令选择时对 RetInst 的翻译较为简单，在实际的后端代码中仅被翻译为一条“ret ”伪指令。但仅这样翻译就丢失了大部分原指令所包含的信息，所以我们对 Ret 指令进行特殊处理，保留其所有信息为了后续进行寄存器分配时提供足够的 def 和 use 信息。同时 ret 作为一个函数的出口，我们将会在一个函数中的每一个 ret 指令前插入一个 exit 块作为函数的退出工作。

GetElementPtrInst 的特殊处理

GEP 指令主要用来计算一个变量，如数组中某个元素其在栈帧中所处的位置的指针。我们需要根据数组的类型、维度进行判断，得到其针对数组起始位置的偏移值。

Phi 函数消除

关于中端使用的 phi 函数的理解：每一个基本块前的所有 phi 函数在进入该基本块时会被“同时执行”。phi 函数表达的语义是在基本块之间进行跳转时产生数据的流动的方式。

后端需要将基本块前的所有 phi 语句转化为正确的拷贝语句序列。

设一个基本块为 succ，考虑 succ 中所有来源为某一前驱 pred 的所有 phi 函数，即仅考虑两个基本块之间的数据流动关系。由 phi 函数的定义可知，phi 函数对于同一个前驱对应的值有且仅有一个。将寄存器作为图的节点，基于数据流动关系建立有向边（从目的寄存器指向源寄存器），则可以构成有向图 G。图 G 中的节点有且最多只有一个出度，由基环树的性质可知，图 G 为内向基环树森林。同时可以发现，设图 G 中除去环的部分的一个拓扑序为 P，按拓扑序 P 执行对应的拷贝语句，是一个正确的拷贝顺序。剩余的环上的拷贝通过可以引入一个临时寄存器解决。

对于 Pred 到 Succ 的拷贝：

1. 建立有向图

2. 拓扑序访问，并创建对应的拷贝指令

3. 对于剩下的所有环，使用一个临时寄存器将环解开完成拷贝

4.6.4 寄存器分配

对于中端的所有虚拟寄存器，都需要置换成真实的寄存器，但是基于 SSA 形式的中端代码假定有无限个寄存器可以用于存放临时变量，寄存器分配的任务就是将大量的临时变量分配到计算机实际具有的少量机器寄存器中，同时在可能的情况下，给一条 MOVE 指令的源地址和目标地址分配同一个寄存器，以便后续我们能删除这条 MOVE 指令

通过活跃性分析的结果，我们可以构建一个冲突图，冲突图的每一个节点代表一个临时变量。每一条边指出边两边的临时变量不能分配到同一个寄存器，然后我们利用算法对这个冲突图进行着色，如果当一个节点没有颜色可以进行着色时，说明我们此时需要将这个节点进行溢出，图着色的算法由四个主要的处理阶段组成：构造、简化、溢出和选择：

1. 构造：构造冲突图。利用数据流分析方法，计算在每个程序点同时活跃的临时变量集合。由该集合中的每一对临时变量形成--条边，并将这些边加入到冲突图中，对程序中的每一点重复这一处理过程

2. 简化： 用一个简单的启发式对图着色。假设图 G 有一个结点 m,它的邻结点个数少于 K,其中 K 是机器寄存器的个数。令 G'为 G- {m}，即 G'是从图 G 中去掉结点 m 后得到的图。若 G'能够用 K 色着色，那么 G 也可以。因为当将 m 添加到已着色的图 G'时，m 的邻结点至多使用了 K-1 种颜色，所以总是能找到一种颜色作为 m 的颜色。这自然地导出了一种基于栈(或递归)的图着色算法:这个算法重复地删除度数小于 K 的结点(并将它压入栈中)。每简化掉--个结点都会减少其他结点的度数，从而产生更多的简化机会

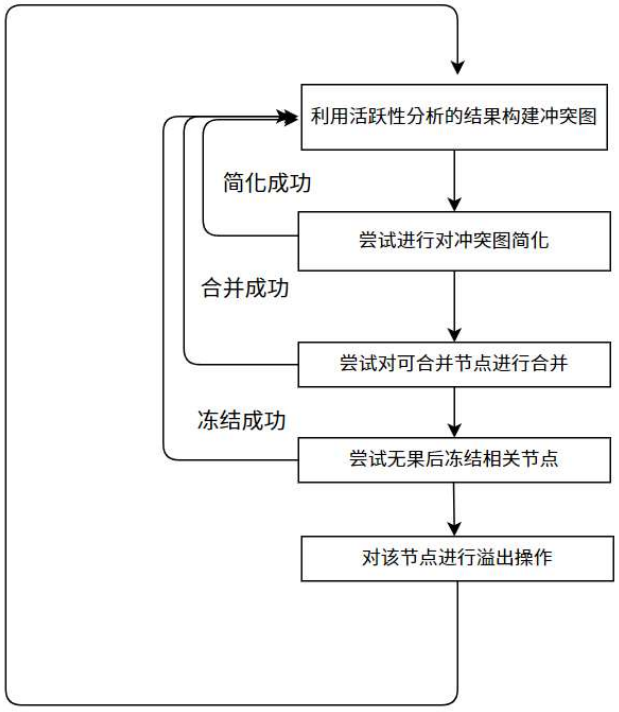
3. 溢出：假设在简化过程的某一点图 G 只包含高度数结点，即度≥K 的结点。这时简化阶段使用的启发式算法已不起作用，于是我们标记某个结点是需要溢溢出(spill): 假设在简化过程的某一点图 G 只包含高度数(significant degree)结点，即度≥K 的结点。这时简化阶段使用的启发式算法已不起作用，于是我们标记某个结点是需要溢出的结点。我们对这个溢出的效果做出乐观的估计，即寄希

望于这个被溢出的结点将来不会与余留在图中的其他结点发生冲突。因此可以将这个选中的结点从图中删除并压入栈中，然后继续进行简化处理。

4. 选择：将颜色指派给图中的结点。我们从一个空的图开始，通过重复地将栈顶结点添加到图中来重建原来的冲突图。当我们往图中添加一个结点时，一定会有一种它可使用的颜色，因为在简化阶段将这个结点移出的前提是只要图中剩余的结点可以成功着色，这个结点就总是有可能分配到一种颜色。

5. 重新开始：如果选择阶段不能为某个(或某些)结点找到颜色，则必须对程序进行改写，使得在每次使用这些结点之前将它从存储器中读出，在每次对这些结点定值之后将它存回到存储器中。这样，一个被溢出的临时变量会转变成几个具有较小活跃范围的新的临时变量。这些新临时变量可能会与图中的其他临时变量发生冲突，因此对改写后的程序还要再重复用该算法进行一-次寄存器分配。这种处理过程将反复迭代，直到没有溢出而简化成功为止。

代码流程图：



4.6.5 函数栈帧生成

每一个函数都会有其对应的一个栈帧，但是这个栈帧里有什么内容只有等到寄存器分配后才能得知：一是在寄存器分配时需要溢出到栈帧的内容；二是在调用函数时大于 8 个的参数需要被保存在栈帧中进行参数传递；三是每个栈帧的固定部分，包括 ra, s0 寄存器和局部变量；四是对 Callee 寄存器的保存。

栈帧生成

由于只有在寄存器分配完成后函数的栈帧才会被真正确定，在此之前使用的 stackRegister 的 offset 均为 0，在寄存器分配后才根据栈帧的内容确定 offset 的具体的值。

一个 RISCVFrame 的结构如下，主要保存了一个关于 RISCVFrameObject 的 vector, 包含了该栈帧的所有内容。通过 spill()函数为寄存器分配和 Callee 寄存器的保存提供接口。

在寄存器完成分配后我们才通过函数 GenerateFrame()真正的进行栈帧的分配，其工作包括计算该栈帧的大小以及每一个 FrameObject 的开始与结束地址，对所有 StackRegister 的 offset 进行更新。同时需要注意的是 RISCV 的栈帧需要以 16 字节进行对齐。

在生成栈帧后才生成创建于销毁栈帧对应的指令。对于创建栈帧的指令全都插入到该函数的第一个基本块的前面，对于销毁栈帧的指令全部放入到一个 exit 块中，当在打印指令时遇到 ret 指令则打印该 exit 块的所有内容。同时由于 StackRegister（形如 12（sp））这类寄存器其 offset 能够表示的值有限，我们还需要对其进行合法化处理。

保存 Caller Saved Register

在后端中，call 语句理解为 def 所有的 caller saved 寄存器，use 作为参数的寄存器即可表达所有的 caller saved 寄存器都会在函数调用后被修改。此时除了作为返回地址的 ra 寄存器，所有 call 前后使用到的 caller saved 寄存器都会在寄存器分配前后通过 spill 的方式保存。

```
{all caller saved registers} = call func (a0,a1,a2)
```

保存 Callee Saved Regsiter

遍历函数使用到的 callee saved register，在函数退出前后对进行保存。

4.6.6 指令合法化

在经过指令选择后我们得到了大体上正确的后端代码，但在某些细节上还并非合法，我们还需要针对不同的情况进行特定的合法化。

如 stackRegister(形如(12)sp)的 offset 值的大小超出了其能表示的范围，branch 语句中不能有立即数的出现，支持立即数的普通指令中立即数不得超出其表示范围（12 位立即数）等形况均为非法指令。

针对 stackRegister 的 offset 值超出表示范围的情况做出如下处理，将 offset 的值与基址寄存器的值加放入一个固定物理寄存器（此处为 t0）中，原指令中根据 s0 或 sp 进行偏移则变为了根据 t0 进行偏移。

如果为 StackRegister 出现在了不支持的位置，如作为 store 指令中的源寄存器和 mv 指令的操作数等情况，需要提前将 stackRegister 中的内容提前加载到固定的物理寄存器（此处为 t0）中，再使用 t0 进行替换。

如果为支持立即数的指令中的立即数超过了其表示范围，也需要进行合法化。一般来说使用 li 指令进行提前加载立即数即可，此处考虑到加载效率进行了进一步的优化，其主要思想是尽量使 li 加载的数为 1 的逻辑左移任意位，再通过 addi 指令进行精确的加载。

参考文献

[1] 周尔强 周帆 韩蒙 陈文宇，编译技术

[2] [美] Andrew W.Appel　Maia GinsBurg，现代编译原理

[3] [美] Alfred V.Aho　Monica-S.Lam 等，编译原理

[4] [美] David Patterson　Andrew Waterman 等，RISC-V 手册

[5] Lam S K, Pitrou A, Seibert S. Numba: A llvm-based python jit compiler[C]//Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC. 2015: 1-6.

[6] Rus S, He G, Alias C, et al. Region array SSA[C]//Proceedings of the 15th international conference on Parallel architectures and compilation techniques. 2006: 43-52.

[7] Appel A W. Modern compiler implementation in C[M]. Cambridge university press, 2004.

[8] Bacon D F, Graham S L, Sharp O J. Compiler transformations for high-performance computing[J]. ACM Computing Surveys (CSUR), 1994, 26(4): 345-420.

[9] Johnson S C. Yacc: Yet another compiler-compiler[M]. Murray Hill, NJ: Bell Laboratories,1975.

[10] Hennessy J L, Patterson D A. A new golden age for computer architecture[J]. Communications of the ACM, 2019, 62(2): 48-60.

[11] Cytron R, Ferrante J, Rosen B K, et al. Efficiently computing static single assignment form and the control dependence graph[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1991, 13(4): 451-490.

[12] Namjoshi K S. Witnessing an SSA transformation[C]//VeriSure Workshop, CAV. 2014.

[13] Zhao P, Amaral J N. To inline or not to inline? Enhanced inlining decisions[C]//Languages and Compilers for Parallel Computing: 16th International Workshop, LCPC 2003, College Station, TX, USA, October 2-4, 2003. Revised Papers 16. Springer Berlin Heidelberg, 2004: 405-419.

[14] Ferstl W, Klahn T, Schweikert W, et al. Inline analysis in microreaction technology: a suitable tool for process screening and optimization[J]. Chemical Engineering & Technology: Industrial Chemistry-Plant Equipment-Process Engineering-Biotechnology, 2007, 30(3): 370-378.