

编译原理研讨课实验PR003实验报告

[author]: 高梓源 官奕琳 [Stu.Num]: 2019K8009929026 2019K8009907026

编译原理研讨课实验PR003实验报告

任务说明

成员组成

实验设计

设计思路

整体框架叙述

IR设计思路

中间代码结构设计

IRContainers

IRCode

IROperands

目标代码结构设计

寄存器管理思路

优化设计思路

指令降级与替换

常量折叠

基本块和控制流图

死代码删除

寄存器管理

实验实现

基本IR翻译过程

SSA风格的IR实现

目标代码构建过程

优化实现

常量折叠

死代码删除

寄存器指派

其它

运行方法

测试结果

总结

实验结果总结

分成员总结

任务说明

在前两次实验的基础上，加入IR和目标代码的设计，在语法分析过程中逐渐构建，以正确地将源CACT程序翻译成为.S汇编文件，进而由gcc代劳翻译为目标RISC-V二进制代码。

成员组成

- 高梓源，2019K8009929026
- 官奕琳，2019K8009907026

实验设计

设计思路

整体框架叙述

根据C++面向对象的特点，类的结构也可以理解为项目层次结构，每一层都具有公开和私有的属性/方法，对外的展现是接口，与其他层应保持隔离性。这样的层次划分目的是每层进行修改和优化都不会影响其他层功能的实现，因此可以相互独立地开发。

本项目的层次结构如下：

| 层次结构 | 作用 |
|-----------------------|--------------------|
| Register, SymbolTable | 寄存器管理逻辑，符号表 |
| TargetCode | 目标代码框架 |
| IROperands | 中间代码操作数 |
| IRCode | 中间代码 |
| IRContainers | 中间代码容器，即函数与程序 |
| IRGenerator | 作为中间层，语法树到中间代码的生成器 |
| semanticAnalysis | 语义分析 |
| main | 入口函数，参数检测 |

表述不严格准确，因为不相邻的类也可能有直接使用的相邻关系。

此外保持层次结构分明的好处是能够聚合操作从而简化代码，比如IRGenerator本质上是中间层，主要作用是：

1. 将IRContainers中的IRFunction, IRProgram进行实例化以管理；
2. 作为semanticAnalysis与上述两者之间的桥梁，将具有公共特征的操作提取，例如enterFunction，会统一在IRProgram中增加此function，将currentIRFunction指针指向当前function等，将操作提取。

此外为方便使用，建立了ErrorMsg和Tools类，提供静态方法和属性可直接调用，分别负责报错信息和工具函数。

IR设计思路

中间代码结构设计

采用自上而下叙述：

IRContainers

作为中间代码的容器，将存放中间代码及所需变量。考虑语句位置存在两种，一种是应用程序级（全局变量定义语句），一种是函数级，因此容器也设置两种：IRFunction与IRProgram，后者包含前者，且为一对多关系。

- IRFunction
 - 会包含2种IR变量，但存储为三种list：参数列表、人工定义变量列表、临时变量列表
 - 会包含Function内部的标签label列表
 - 会包含中间代码
 - 会为变量/标签的命名提供累加器
 - 会包含function符号表
 - 会包含栈大小信息
- IRProgram
 - 包含全局变量列表
 - 记录IRFunction列表
 - 将IRFunction转化成中间代码操作数之后记录，以便IRCall指令调用
 - 记录必要的立即数（float, double, array），并为他们的命名提供累加器

IRCode

中间代码类。一般中间代码由四个部分组成：operation, result, arg1, arg2，即为操作类型与至多三个操作数。中间代码所需支持的方法很少：构造函数，打印，生成目标代码。但是中间代码种类非常之多，本项目支持的操作有以下：

| | | |
|---|---|------------|
| 1 | ADD, SUB, NEG, MUL, DIV, MOD, NOT, OR, AND, SEQ, SNE, SLT, SGT, SLEQ, SGEQ, | // 运算操作 |
| 2 | BEQZ, GOTO, ADD_LABEL, | // 跳转操作 |
| 3 | PHI, REPLACE, | // SSA风格 |
| 4 | ASSIGN, FETCH_ARRAY_ELEM, ASSIGN_ARRAY_ELEM, | // 赋值与数组操作 |
| 5 | ADD_PARAM, GET_PARAM, CALL, RETURN, GET_RETURN, | // 函数调用操作 |

总共5类28种。各种处理方式都有优劣：

- 1类：只设置一类而后使用C版本的switch case虽然可能略微简单，但是每个函数将相当冗长，需要支持28种case；雪上加霜的是以上操作还不涉及操作数类型，一旦考虑操作数类型，到目标代码的翻译过程更加复杂。
- 29类：为28种操作建立28个类，遇到操作数类型需要单独考虑的在类内进行switch case，算是一种折中方式，估计是单层来看最好的解决办法。
- 1+28×操作数种类数类：面向对象贯彻到底的做法，必要时将以上28类作为中间类，中间类提供打印方法，子类根据操作数类型提供声称目标代码的方法，代码量仍然相当大。

本项目采用第三种做法，但并不认为这种思路很好。

IROperands

设置五种IROperands: label, value, symbolVar, symbolFunc, tempVar, 分别表示标签、立即数、用户定义变量、函数、临时变量。采用面向父类的设计, IROperand以虚函数支持底层五种操作数的所有方法, 使得IRCode所接触到的操作数都是多态的IROperands。

value需要表示的立即数由很多种, 各种数据类型的数组/非数组立即数, 因此采用一个vector来存储每个元素, 外加一个bool类型判断是否是数组。

之所以把变量拆分成symbolVar和tempVar, 是因为符号表中只记录用户定义变量信息, 而临时变量是在IR生成过程中编译器添加的, 因此symbolVar指向符号表即可获取充足的信息, 而tempVar通过设置相关属性来模拟符号表中存储。

symbolFunc也是存储符号表信息, 注意它并不能存储IRFunction的信息, 因为由此前层次结构叙述, 这层在IRFunction之上, 绝不能逆向调用, 而为了获取充足的IRFunction信息, 可以的做法是在function符号表中增加相应的属性, 由IRFunction传递上去, 由symbolFunc接取。

目标代码结构设计

TargetCode并没有采用IRCode相同的构建方法, 太废篇幅, 而是采用1类的设计建立Code类。每条目标代码实际上组成元素最多有: 汇编指令, 目标寄存器, 两个源寄存器, 源与目标寄存器是否是浮点类型的判断, 立即数, 标签string, 指导信息。对应:

```
1  class Code {
2  public:
3      ASMOperation op;
4      FloatPointType rdFloatPointType;
5      FloatPointType srcFloatPointType;
6      Register *rd;
7      Register *rs1;
8      Register *rs2;
9      int offset;
10     std::string label;
11     std::string directives;
12 }
```

本项目需要用到的汇编指令有:

```
1  ADD, SUB, ADDI, NEG, MUL, REM, DIV, SLL, SLR, SLLI, SLRI,
2  NOT, OR, AND, XOR, SEQZ, SNEZ, SLTZ, SGTZ, SLT, FEQ, FLT, FLE,
3  BEQZ, JR, LABEL, CALL, RET,
4  LLA, LI, LB, LW, LD, FLW, FLD, SB, SW, SD, FSW, FSD,
5  MV(多种),
6  DIRECTIVE
```

打印输出时直接根据汇编指令类型输出指令string即可。

此外建立TargetCodes作为中间层, 存储目标代码列表, 并且管理通用/浮点两组寄存器, 添加具体类型的目标代码、对寄存器进行请求和释放也通过此类实现。

寄存器管理思路

本项目的寄存器管理是细粒度的，完全可以用作模拟器的寄存器管理，有部分设计可能对于编译器冗余。

关于寄存器首先设计了Register类作为单个寄存器的表征，记录信息如下：

```
1 class Register {
2 private:
3     RegisterType registerType;           // 通用/浮点
4     FloatPointType floatPointType;       // 若为浮点，浮点类型Single/Double
5     int number;                           // 寄存器序号，非ABI
6     std::string abiAliasName;             // ABI名称
7     long long value;                     // 寄存器中值（编译器冗余）
8     bool allocated;                       // 是否已被分配
9     bool occupied;                       // 是否已被指派
10    bool tmpStored;                       // 是否暂存中
11    int tmpStoreOffset;                   // 暂存位置，栈内偏移
12 }
```

本项目分配与指派的概念与编译原理课上所教授有所不同，分配只临时调用寄存器资源，指派只应用于变量绑定某寄存器。

而后需要建立两组寄存器：通用/浮点，通过RegisterType来判断，采用面向父类，但使用过程中并不面向父类。

寄存器组需要存储的信息如下：

```
1 class GeneralPurposeRegisters : public Registers {
2 private:
3     std::vector<std::string> registerClass;           // ABI下寄存器前缀名
4     std::unordered_map<std::string, int> registerClassNum; // 前缀名中寄存器数量
5     std::unordered_map<std::string, int> registerAllocBitmap; // 分配Bitmap
6     std::unordered_map<std::string, int> registerOccupiedBitmap; // 指派Bitmap
7     std::unordered_map<std::string, Register *> registerList; // ABI名称对应Register列表
8 }
```

寄存器需要支持的操作如下：

- 寻找下一个空闲未分配的寄存器
- 寻找下一个空闲未指派的寄存器
- 寻找特定寄存器
- 释放分配寄存器
- 释放指派寄存器

优化设计思路

以下四个部分对应优化级别为：-00，-01，-02，-03，后面的优化级别也会完成所有此前级别的优化。

指令降级与替换

主要体现是对于编译阶段能确定的，有常量参与的运算进行指令优化。

比如理论上都位于数据区的data，若为bool或int，且范围在2048内将直接使用`addi rd, zero, imm`替代，超过2048使用`li rd, imm`替代。原先的做法是都采用`.equ label, value`的方式指代，或存储在数据区，现在的做法更贴近gcc。

再者如果编译器检测出常数乘除法，若常数与2的幂相差1以内，则会使用至多两条指令替代，第一条为移位指令，第二条为加减。

常量折叠

函数主要位于IRContainers: void IRFunction::constFolding()

此处设计的常量折叠主要涉及以下几种情况：

- 单操作数赋值语句：如ASSIGN, NEG, NOT
 - 如果arg1为常量，例如`res = const`，那么直到下一次res被定值之前，将所有res的引用替换为const
 - 不仅可以实现指令降级和替换，还可能使得代码能进一步被常量折叠，例如原本后续存在代码`a = 1 + res`；当res被替换为常量后，则该代码也可以进行常量折叠
- 双操作数赋值语句：
 - arg1, arg2均为常量：如ADD, SUB, SEQ, SNE, etc
 - 例如`a = const1 + const2`；
 - 计算出`const1 + const2`，将计算出的结果设置为一个新的IRValue，不妨名为new_value
 - 将该code修改为`a = new_value`；
 - 扫描后续代码，直到下一次a被定值前，将所有a的引用替换为new_value
 - arg1, arg2只有一个为常量：此处仅考虑了ADD, SUB, MUL操作
 - 传统意义上的常量折叠操作
 - 记该代码为code
 - 如果code为ADD/SUB操作：假设`res = a + const1`；
 - 扫描后续代码new_code，直到下一次res被定值前——如果new_code也为ADD/SUB操作，且new_code的一个操作数为res，即new_code形如：`new_res = res - const2`；
 - 替换new_code为`new_res = a + new_value`；其中new_value的值等于`const1 - const2`
 - 相当于对用code对res进行了替换，常量折叠
 - 如果code为MUL操作：假设`res = a * const1`；
 - 扫描后续代码new_code，直到下一次res被定值前——如果new_code也为MUL操作，且new_code的一个操作数为res，即new_code形如：`new_res = res * const2`；
 - 替换new_code为`new_res = a * new_value`；其中new_value的值等于`const1 * const2`

值得一提的是，这一部分常量折叠仅仅是对基本块内的代码进行折叠，而且常量折叠部分并不会进行代码删除的操作，因为此处还没有进行活跃变量分析，删除代码可能导致错误：

比如 `a = 1 + 2; constFolding()` 方法会将代码变成 `a = 3;`，且会将在本基本块内，所有 `a` 在下一次定值之前的引用改为 3，但由于不知道基本块出口活跃变量，`a` 可能是本基本块出口的活跃变量，但其余基本块对 `a` 的引用并没有被改变为 3，所以如果删除代码可能会导致错误。

基本块和控制流图

函数主要位于 `IRContainers::void IRFunction::basicBlockDivision()`。

根据理论课定义划分基本块：

基本块入口语句可能是：
程序的第一个语句
跳转的目标语句
条件跳转的下一条语句
基本块的结束语句可能是
停机语句
跳转语句
跳转语句的前一条语句（词法序）

算法上本项目只需使用统计基本块入口，定义相关数据结构，本函数主要目的就是生成他们：

```
1  class IRFunction {
2      std::vector<int> entrances;                // 基本块入口下标
3      std::vector<std::vector<IRCode *>> basicBlocks; // 基本块组
4      std::vector<std::vector<int>> controlFlow;    // 控制流
5      std::vector<int> cycleNum;                  // 基本块循环嵌套数
6  };
```

针对中间代码的优化基本体现在 `IRFunction` 中，在生成目标代码之前进行。

核心是 `entrances` 的生成，按照算法扫描中间代码，针对跳转语句和目标进行处理即可：

```
1  entrances.clear();
2  entrances.push_back(0);
3  for (int i = 0; i < codes.size(); i++) {
4      if (codes[i]->getOperation() == IROperation::ADD_LABEL) {
5          entrances.push_back(i);
6      }
7      if (codes[i]->getOperation() == IROperation::BEQZ || codes[i]-
>getOperation() == IROperation::GOTO) {
8          entrances.push_back(i + 1);
9      }
10 }
11 entrances.push_back(codes.size());
```

可以认为入口下标之间的区域（左闭右开）都是基本块，于是将 `code` 分块 `push` 进入 `basicBlocks` 即可：

```

1   for (int i = 0; i < entrances.size() - 1; i++) {
2       basicBlocks.emplace_back(codes.begin() + entrances[i], codes.begin() +
entrances[i + 1]);
3   }
4   entrances.pop_back();

```

控制流，本项目controlFlow的定义是每个基本块可以到达的基本块下标，因此扫描基本块时只需要考虑跳转语句，然后遍历label语句寻找跳转目标所在基本块即可。

基本块循环嵌套数的应用主要为后续图着色+计数法的寄存器分配进行准备。根据理论课定义，循环的发现主要依赖于回边，因此若controlFlow跳转目标的基本块下标小于本基本块，则该基本块位于一个循环中，循环数++。

死代码删除

主要函数集中在IRContainers中

```
void IRFunction::delDeadCode(); -- 进行死代码删除
```

```
void IRFunction::liveVarAnalysis(); -- 对每个基本块进行活跃变量分析
```

```
void IRFunction::usedefVarsAnalysis(); -- 对每个基本块的use变量和def变量进行分析
```

■ 活跃变量分析

- 对每个基本块的use和def变量进行分析
 - 对每个基本块从后往前扫描，对每一个IRCode
 - 将该code中定值的量从use中删除，加入def
 - 将该code中引用的量从def中删除，加入use
 - 注意对同一个code而言，操作顺序为先处理定值变量，再处理引用变量，例如`x = x + 1`；应该先将x从use中删除，加入def，再从def中删除，加入use
- 分析outVars和inVars：
 - 每一个block的outVars为该block的所有后继blocks的inVars的并集
 - 根据 $inVars = useVars + (outVars - defVars)$ 确定每一个block新的inVars
 - 比较新的inVars和之前的inVars，直到每一个block的两次inVars都保持不变才能结束循环

■ 死代码删除

■ 基本块内优化

- 将该block的所有outVars设置为alive
- 扫描基本块中的每一个IRCode，记为code
- 对于每一个code
 - 如果定值变量为不活跃，则删除code
 - 否则，将定值变量设置为not alive，再将引用变量设置为alive，注意顺序不应该颠倒
 - 不同的IRCode的定值变量和引用变量情况不同，需要一一判别，例如`GETRETURN res`相当于对res的定值，而`AddParam arg1, arg2`，相当于对arg1的引用

- 值得一提的是，上述活跃变量分析算法只是保守估计了每一个基本块的outVars，可能存在的一种情况是，如果基本块间无循环，block A有且仅有一个后继block B，在block B中存在 $t1 = t2 + 1$ ；这样的代码，而t1不活跃，该代码被删除，且block B中不存在对t2的其余引用，实际上t2可以被设置为not alive；但活跃变量分析算法会将t2当作use vars存储，所以t2会在B的inVars中，进而存在于A的outVars中，导致代码冗余。所以，如果我们发现基本块之间不存在循环，可以对代码的所有基本块从后向前扫描，由算法+控制流决定每一个基本块的outVars（这种情况下t2不会存在于A的outVars中），使得代码更加优化

寄存器管理

最初的寄存器细粒度设计是为了指派，但是初期将其作为分配使用后也可以避免寄存器交叠使用的错误，于是都采用相似的管理方式。

分配：

分配和指派都采用两层约束，Register类本身有被分配和指派的判断信息，在Registers中使用Bitmap作为Cache。Bitmap按照不同的ABI前缀划分，具体初始化形式为：

```
1 registerAllocBitmap = {{ "t", 0 }, { "a", 0 }, { "s", 0 }, { "tp", 0 }, { "gp", 0 }, { "sp", 0 }, { "ra", 0 }, { "zero", 0 } };
```

这是RISC-V常量寄存器的初始化，在此前定义了寄存器分配ABI前缀的选择顺序，这里沿用，虽然是unordered_map，但使用中严格遵循顺序。根据寄存器用途，t系列寄存器随便使用，因此优先度最高，其余寄存器ABI要求中都存在特殊用途，因此优先度靠后。

因此在寄存器分配时需要严格维护两套分配判断信息。注意以下数据结构存储实际Register *，索引方法是使用ABI别名：

```
1 std::unordered_map<std::string, Register *> generalPurposeRegisterList;
```

- 在寻找下一个可用寄存器操作中，使用Bitmap，寻找第一个为0的偏移，生成寄存器ABI名称后返回Register *，并且直接设置Bitmap和Register *中的分配信息。
- 而在寻找特定寄存器时，直接利用ABI名称索引到对应的Register *后判断是否可分配，设置Bitmap和Register *中的相关判断后返回。

指派：

思路基本同上，指派算法见实验实现。

实验实现

基本IR翻译过程

SSA风格的IR实现

根据SSA的定义：每个变量在中间代码中只能出现在左值一次（不算初始化）。因此在一部分IROperand中加入被赋值的判断信息，语义分析时扫描到左值已被赋值的IROperand，则新建一个临时变量替代它，替代动作使用IRReplace表示，将新的临时变量和原变量之间相互映射，在此后对原变量的所有引用将转移到新的临时变量上。

此外，若在循环语句中对外层变量进行赋值，那么就需要使用IRPhi操作。在语义分析中使用vector统计循环内部被修改的所有变量，传递到外层后加入new tempVar = IRPhi(vector<IROperand *> rawVars)，进行一个总结，实际上是新建一个映射，方便后续使用时明确该使用的变量最新值，不必困扰与分支语句的多可能性。

SSA风格并不需要体现在目标代码中，但tempVar的复杂性会影响部分代码的翻译工作。

目标代码构建过程

目标代码构建主要在IRCode.cpp中，通过每种IRCode的genTargetCode函数实现，函数会对操作数的类型进行判断，而后选择最合适的目标代码生成方式，传送到targetCode中，比如IRAdd：

```
1 void IRAddI::genTargetCode(TargetCodes *t) {
2     bool hasFreeRegister;
3     Register *resultReg;
4     // 结果变量是否绑定寄存器
5     if (result->getBindRegister()) {
6         resultReg = result->getTargetBindRegister();
7     }
8     else {
9         resultReg = t->getNextFreeRegister(true, false, FloatPointType::NONE,
hasFreeRegister);
10    }
11    // 若某一方为立即数，使用ADDI指令
12    if (arg1->getOperandType() == OperandType::VALUE && arg2->getOperandType() !=
OperandType::VALUE) {
13        Register *arg2Reg = arg2->load(t, true);
14        t->addCodeAddi(resultReg, arg2Reg, stoi(arg1->getValue()));
15        t->setRegisterFree(arg2Reg);
16    }
17    else if (arg1->getOperandType() != OperandType::VALUE && arg2->
getOperandType() == OperandType::VALUE) {
18        Register *arg1Reg = arg1->load(t, true);
19        t->addCodeAddi(resultReg, arg1Reg, stoi(arg2->getValue()));
20        t->setRegisterFree(arg1Reg);
21    }
22    // 若两方都为立即数，进行折叠
23    else if (arg1->getOperandType() == OperandType::VALUE && arg2->
getOperandType() == OperandType::VALUE) {
24        Register *zero = t->tryGetCertainRegister(true, "zero", hasFreeRegister);
25        t->addCodeAddi(resultReg, zero, stoi(arg1->getValue()) + stoi(arg2->
getValue()));
26        t->setRegisterFree(zero);
27    }
28    else {
29        // 否则正常ADD
30        Register *arg1Reg = arg1->load(t, true);
```

```

31     Register *arg2Reg = arg2->load(t, true);
32     t->addCodeAdd(resultReg, arg1Reg, arg2Reg, FloatPointType::NONE);
33     t->setRegisterFree(arg1Reg);
34     t->setRegisterFree(arg2Reg);
35 }
36 result->storeFrom(t, resultReg);
37 t->setRegisterFree(resultReg);
38 }

```

为了方便生成，我们为每个操作数内部提供了load/store方法，最初是严格按照load/store结构，将会从内存中（全局/只读数据区/栈）将操作数加载出来。后面提供了寄存器指派的方式，可能操作数本身被指派到某个寄存器中，则使用时直接将寄存器返回即可。

提供load/store方法的操作数有立即数/符号变量/临时变量，情况极其复杂。操作数存的内容有不同种类，数据类型/是否是数组，需要考虑。是否是函数变量或者全局变量也需要考虑。因此这部分算法不复杂，但实现极为复杂。

优化实现

常量折叠

- 双操作数赋值语句中，arg1和arg2中只有一个为常量的情况

```

1  else if (arg1->getOperandType() == OperandType::VALUE || arg2->getOperandType()
2  == OperandType::VALUE) {
3      if(op != IROperation::ADD && op != IROperation::SUB && op !=
4      IROperation::MUL)
5          continue;
6
7      IRoperand* imm_arg = arg1->getOperandType() == OperandType::VALUE ? code-
8      >getArg1() : code->getArg2();
9      IRoperand* var_arg = arg1->getOperandType() == OperandType::VALUE ? code-
10     >getArg2() : code->getArg1();
11
12     for(int j = i + 1; j < block.size(); j++){
13         IRCode *new_code = block[j];
14         IRoperation new_op = new_code->getOperation();
15
16         if(new_code->getArg1() == res && new_code->getArg2()->getOperandType() ==
17         OperandType::VALUE){
18             // new_code: arg1 is Var, arg2 is immVal
19             if(op == IROperation::ADD){
20                 IRValue* new_value = immAddSub(imm_arg, new_code->getArg2(),
21                 new_op);
22
23                 switch(var_arg->getMetaDataType()){
24                     case MetaDataType::INT:
25                         codes[entrances[bnum] + j] = new IRAddI(new_code-
26                         >getResult(), var_arg, new_value);
27                     break;
28                     case MetaDataType::FLOAT:

```

```

21         codes[entrances[bnum] + j] = new IRAddF(new_code-
>getResult(), var_arg, new_value);
22         break;
23     case MetaDataType::DOUBLE:
24         codes[entrances[bnum] + j] = new IRAddD(new_code-
>getResult(), var_arg, new_value);
25         break;
26     }
27
28     } else if (op == IROperation::SUB) {
29         if(arg2->getOperandType() == OperandType::VALUE){ // origin code:
arg1 is var
30             IRValue* new_value = immAddSub(imm_arg, new_code->getArg2(),
new_op == IROperation::ADD ? IROperation::SUB : IROperation::ADD);
31             switch(var_arg->getMetaDataType()){
32                 case MetaDataType::INT:
33                     codes[entrances[bnum] + j] = new IRSubI(new_code-
>getResult(), var_arg, new_value);
34                     break;
35                 case MetaDataType::FLOAT:
36                     codes[entrances[bnum] + j] = new IRSubF(new_code-
>getResult(), var_arg, new_value);
37                     break;
38                 case MetaDataType::DOUBLE:
39                     codes[entrances[bnum] + j] = new IRSubD(new_code-
>getResult(), var_arg, new_value);
40                     break;
41             }
42         } else { // origin code: arg2 is var
43             IRValue* new_value = immAddSub(imm_arg, new_code->getArg2(),
new_op);
44             switch(var_arg->getMetaDataType()){
45                 case MetaDataType::INT:
46                     codes[entrances[bnum] + j] = new IRSubI(new_code-
>getResult(), new_value, var_arg);
47                     break;
48                 case MetaDataType::FLOAT:
49                     codes[entrances[bnum] + j] = new IRSubF(new_code-
>getResult(), new_value, var_arg);
50                     break;
51                 case MetaDataType::DOUBLE:
52                     codes[entrances[bnum] + j] = new IRSubD(new_code-
>getResult(), new_value, var_arg);
53                     break;
54             }
55         }
56     } else if (op == IROperation::MUL) {
57         if(new_op == IROperation::MUL) {
58             IRValue* new_value = immMul(imm_arg, new_code->getArg2());
59             new_code->setArg1(var_arg);
60             new_code->setArg2(new_value);
61         }

```

```

62     }
63     } else if (new_code->getArg1()->getOperandType() == OperandType::VALUE &&
new_code->getArg2() == res) {
64         // new_code: arg2 is Var, arg1 is immVal
65         if (op == IR0peration::ADD) {
66             IRValue* new_value = immAddSub(new_code->getArg1(), imm_arg,
new_op);
67             new_code->setArg1(new_value);
68             new_code->setArg2(var_arg);
69         } else if (op == IR0peration::SUB) {
70             if(arg2->getOperandType() == OperandType::VALUE){ // origin code:
arg1 is var
71                 new_code->setArg2(var_arg);
72                 IRValue* new_value = immAddSub(new_code->getArg1(), imm_arg,
new_op == IR0peration::ADD ? IR0peration::SUB : IR0peration::ADD);
73                 new_code->setArg1(new_value);
74             } else { // origin code: arg2 is var
75                 IRValue* new_value = immAddSub(new_code->getArg1(), imm_arg,
new_op);
76                 switch(var_arg->getMetaDataType()){
77                     case MetaDataType::INT:
78                         if(new_op == IR0peration::ADD)
79                             codes[entrances[bnum] + j] = new IRSubI(new_code->
getResult(), new_value, var_arg);
80                         else
81                             codes[entrances[bnum] + j] = new IRAddI(new_code->
getResult(), new_value, var_arg);
82                         break;
83                     case MetaDataType::FLOAT:
84                         if(new_op == IR0peration::ADD)
85                             codes[entrances[bnum] + j] = new IRSubF(new_code->
getResult(), new_value, var_arg);
86                         else
87                             codes[entrances[bnum] + j] = new IRAddF(new_code->
getResult(), new_value, var_arg);
88                         break;
89                     case MetaDataType::DOUBLE:
90                         if(new_op == IR0peration::ADD)
91                             codes[entrances[bnum] + j] = new IRSubD(new_code->
getResult(), new_value, var_arg);
92                         else
93                             codes[entrances[bnum] + j] = new IRAddD(new_code->
getResult(), new_value, var_arg);
94                         break;
95                 }
96             }
97         } else if (op == IR0peration::MUL) {
98             if(new_op == IR0peration::MUL) {
99                 IRValue* new_value = immMul(imm_arg, new_code->getArg1());
100                 new_code->setArg1(new_value);
101                 new_code->setArg2(var_arg);
102             }

```

```

103     }
104 }
105 }

```

- 这个情况较为复杂，需要分析code和new_code的操作，++/+-/--/--都可能对应不同的情况，同时，在这四种不同的大情况下，还需要考虑操作数的顺序，例如究竟是var - const，还是const - var，这都可能产生不同情况折叠的code

例如：原始code为a = 1 - b；扫描得到的新code为c = a - 2；那么新code要替换为c = -1 - b；

如果新code为c = 2 - a；那么新code要被替换为c = 1 + b；情况繁琐，需将各种情况考虑全面及正确

死代码删除

- 活跃变量分析

```

1 void IRFunction::liveVarAnalysis() {
2     bool changed;
3     // use & def vars 分析
4     usedefVarsAnalysis();
5     inVars = std::vector<std::vector<IROperand*>>(basicBlocks.size(),
std::vector<IROperand*>());
6     outVars = std::vector<std::vector<IROperand*>>(basicBlocks.size(),
std::vector<IROperand*>());
7
8     do{
9         changed = false;
10        for(int i = basicBlocks.size() - 1; i >= 0; i--){ // i for block
number
11            auto out = useVars[i];
12            std::vector<IROperand*> newin;
13            std::vector<IROperand*> newOut;
14            // update out vars
15            vector<int> ctrlflow = controlFlow[i];
16            for(int back : ctrlflow){
17                if (back >= inVars.size()) {
18                    continue;
19                }
20                for(auto var : inVars[back])
21                    addOperandToVec(newOut, var);
22            }
23
24            // IN = use U (out - def)
25            // add OUT variables
26            for(auto & var : newOut)
27                addOperandToVec(newin, var);
28            // delete DEF variables
29            for(auto & var : defVars[i])
30                delOperandInVec(newin, var);
31            // add USE variables
32            for(auto & var : useVars[i])

```

```

33         addOperandToVec(newin, var);
34
35         if(!cmpTwoInVars(inVars[i], newin)){
36             // update inVar
37             changed = true;
38             inVars[i].clear();
39             for(auto & var : newin)
40                 inVars[i].push_back(var);
41         }
42         if(!cmpTwoInVars(outVars[i], newOut)){
43             // update inVar
44             changed = true;
45             outVars[i].clear();
46             for(auto & var : newOut)
47                 outVars[i].push_back(var);
48         }
49     }
50     } while (changed);
51 }

```

寄存器指派

本项目采用图着色+计数法的思路完成寄存器指派。因为图着色法的假说是基于无穷多寄存器，因此不能保证收敛在RISC-V能够支持的数目之内，因此仍然需要判断那些变量组最适合放在寄存器中。

完成此前的基本块划分和控制流生成之后，进行变量活跃区间计算，结果直接存储在IROperand中，而不是IRFunction。活跃区间指的是定值-执行流后续的所有首次定值前引用之间的区间，单位是语句。求解过程需要维护definitions和uses两个vector，代表定值点和引用点。而后将引用点和前方最近的定值点连线即可。需要注意控制流中的定值点会在目标基本块起始位置，随后在目标基本块中的引用即可得到定值点位置。

活跃区间计算完成后需要计算变量之间冲突信息，这部分较为简单，从需要指派寄存器的变量取出活跃区间信息，两两之间相互比较，得到冲突表。

利用冲突表可以画寄存器冲突图，冲突表中存在冲突的两个寄存器之间有连线。

冲突图形成后即可运行图着色算法，本项目采用的算法描述如下：

将当前未着色的点按度数降序排列。
 将第一个点染成一个未被使用的颜色。
 顺次遍历接下来的点，若当前点和所有与第一个点颜色相同的点不相邻，则将该点染成与第一个点相同的颜色。
 若仍有未着色的点，则回到步骤1, 否则结束。

通过图着色算法，我们可以得到二维变量组，每一维是之间没有冲突的变量组，这些变量能够被分到同一个寄存器中不引发灾难。

但图着色算法不能保证收敛在RISC-V能够支持的寄存器数目之内，因此再度进行计数，对所有变量进行估值，对每层循环内的变量按10倍估值，得到变量使用频率的排序。针对排序内在寄存器指派最大值内的变量指派寄存器。

为了体现ABI的一致性，针对寄存器指派后的优化，需要在call其他function的时候将本函数所使用到的caller saved寄存器暂存，call结束后将除a0寄存器的其他caller saved寄存器加载，a0寄存器在与结果变量交互后若已被指派则进行加载。在function进入的时候对于已分配的callee saved寄存器进行暂存，在函数返回时全部加载。

其它

运行方法

在项目根目录下运行`./build.sh`，将会编译生成编译器，而后运行`./run_tests.sh -O[0-2]`，将会对`tests/samples_codegen/`目录下的CACT测试样例编译，并且在同目录下生成中间代码文件`.ir`，汇编文件`.S`，目标代码，反汇编`.obj`，并且会使用spike测试，输出结果到`.out`。因此总共1个文件变成6个文件。

测试结果

在三种优化下均能够正确运行，且优化结果能够明显看出，但优化后执行效率根据具体情况而定。比如若函数定义很多变量的同时很频繁地调用其他函数，在`-O3`的优化开启时，将会引发大量ABI要求下的load/store开销，结果性能可能不如不开优化。

给出`-O2`优化的结果：

■ 源代码：

```
1  int main()
2  {
3      int n = 0, result;
4      if(n > 1)
5          result = 999 * 2000;
6      else {
7          result = 1 + 2 * 8 + 100 - 9999 + 10000/5;
8          n = result + 1000;
9      }
10     return 0;
11 }
```

■ 优化后IR:

```
1  ----- Codes -----
2  main:
3      return 0;
```

■ 优化后汇编：

```
1  .text
2  .align 1
3  .globl main
4  .type main, @function
5  main:
6      mv a0, zero
7      ret
```

总结

实验结果总结

本编译器项目是已经是一个非常庞大完整的项目，兼有前后端的完整编译链。而且加入了优化框架，尽管优化仍然还有很大的空间，但框架已然搭建完成，剩余所需的仅是应用算法。

而且在进行优化的过程中，我们会发现很多情况下人主观能够发现的优化依靠代码很难实现。有一部分在后续加入函数间数据流分析后能得到缓解，有些还需依赖更加智能的分析方法。

分成员总结

■ 高梓源

通过这一学期的实验，我完全掌握了编译各个阶段的设计和算法，实现了自己的编译器，并且进行了适当的优化，实现了理论课所学到的知识。通过编译原理实验，也让我面向对象编程的思想掌握和应用的更加纯然，对于RISC-V架构和指令集有了更广泛的了解，对于以后从事软硬件交互的研究工作有很大帮助。

■ 官奕琳

通过本学期的学习，我了解了编译的各个阶段如何组织运行，发现了编译的许多值得探究和品味之处——例如中间代码设计需要在统一和详细之间做权衡，如果设计统一，那么可以更简化优化的算法；但设计详细可以使得目标代码生成时拥有更多信息；优化时不仅需要用演绎的思维去探索算法应该如何推演，还要根据实际情况和诸多细节进行完善，相比起前两部分来说，编译优化更有挑战性，也更有趣味。