# 重庆大学编译原理课程实验报告

| 年级、专业、班级 | 21 计卓 1 班 | 姓名 | 李宽宇 |
|---|---|---|---|
| 实验题目 | 编译器设计与实现 | | |
| 实验时间 | 2024.6.6 | 实验地点 | 竹园四栋 |
| 实验成绩 | | 实验性质 | □验证性 □设计性 ■综合性 |

**教师评价：**

□算法/实验过程正确；□源程序/实验内容提交 □程序结构/实验步骤合理；

□实验结果正确； □语法、语义正确； □报告规范；

其他：

评价教师签名:

## 一、实验目的

以系统能力提升为目标，通过实验逐步构建一个将类 C 语言翻译至汇编的编译器，最终生成的汇编代码通过 GCC 的汇编器转化为二进制可执行文件，并在物理机或模拟器上运行。实验内容还包含编译优化部分，帮助深入理解计算机体系结构、掌握性能调优技巧，并培养系统级思维和优化能力。

## 二、实验项目内容

本次实验将实现一个由 SysY (精简版 C 语言，来自 https://compiler.educg.net/) 翻译至 RISC-V 汇编的编译器，生成的汇编通过 GCC 的汇编器翻译至二进制，最终运行在模拟器 qemu-riscv 上

实验至少包含四个部分: 词法和语法分析、语义分析和中间代码生成、以及目标代码生成，每个部分都依赖前一个部分的结果，逐步构建一个完整编译器

**实验一**：词法分析和语法分析，将读取源文件中代码并进行分析，输出一颗语法树

**实验二**：接受一颗语法树，进行语义分析、中间代码生成，输出中间表示 IR (Intermediate Representation)

**实验三**：根据 IR 翻译成为汇编

**实验四(可选)**：IR 和汇编层面的优化

## 三、实验内容实现

### 1、实现哪些内容

实验一（58/58）词法分析和语法分析，将读取源文件中代码并进行分析，输出一颗语法树，具体实现如下：

词法分析：

1.1 任务：对输入的字符流进行处理，再输出词法单元(Token)流。

1.2 思路：通过一个扫描器（scanner）实现——读取输入字符串并生成一系列的 tokens。用有限状态自动机（DFA）来处理输入，在某些转移过程中累计接受到的字符，在适当的时候生成 token。关键是实现 DFA，DFA 状态转移表如下：

表 1 DFA 状态转移表

| 输入 / 当前状态 | space | alpha | '_' | digit | '.' | 操作符 |
|---|---|---|---|---|---|---|
| Empty | Empty | Ident | Ident | IntLiteral | FloatLiteral | op |
| Ident | Empty<br>生成 token | Ident | Ident | Ident | \ | op<br>生成 token |
| op | Empty<br>生成 token | Ident<br>生成 token | Ident<br>生成 token | IntLiteral<br>生成 token | FloatLiteral<br>生成 token | op<br>可能生成 token |
| IntLiteral | Empty<br>生成 token | IntLiteral | \ | IntLiteral | FloatLiteral | op<br>生成 token |
| FloatLiteral | Empty<br>生成 token | \ | \ | FLOATLTR | \ | op<br>生成 token |

1.3 代码与具体实现：

(1)Token 为 TokenType 和 value 的二元组。

```
struct Token {
    TokenType type;
    std::string value;
};
```

其中，value 在 DFA 中累积，TokenType 包含了各种可能的标记类型。每个枚举常量代表一种特定的标记类型，例如 IDENFR 代表标识符，INTLTR

代表整数字面量，FLOATLTR 代表浮点数字面量,各种运算符号和分隔符段落符也都是 token。

```
enum class TokenType{
    IDENFR,     // identifier
    INTLTR,     // int literal
    FLOATLTR,       // float literal
    CONSTTK,        // const
    VOIDTK,     // void
    INTTK,      // int
    FLOATTK,        // float
    IFTK,       // if
    ELSETK,     // else
    WHILETK,        // while
    CONTINUETK,     // continue
    BREAKTK,        // break
    RETURNTK,       // return
    PLUS,       // +
    MINU,       // -
    MULT,       // *
    DIV,        // /
    // ......
```

（2）有限状态自动机（DFA）来处理输入，在某些转移过程中累计接受到的字符，在适当的时候生成 token。定义 DFA 结构体如下：

```
// definition of DFA
struct DFA {
    /**...
    DFA();
    /**...
    ~DFA();
    DFA(const DFA&) = delete;   // copy constructor
    DFA& operator=(const DFA&) = delete;    // assignment
    /**...
    bool next(char input, Token& buf);
    /**...
    void reset();
private:
    State cur_state;    // record current state of the DFA
    std::string cur_str;    // record input characters
};
```

其中，next 函数是 DFA 的状态转移函数，也是实验一重点，实现见(4)，用 cur_state 记录当前状态，cur_str 记录当前累计的字符。

（3）扫描器（Scanner），将字符依次输入 dfa 的 next 函数，当 dfa 发出生成信号时，产生 token，进而将字符串输入转化为 Token 串。其执行函数代码如下：

```cpp
vector<Token> Scanner::run()
{
    // 仿照作业1，但多了删除注释的步骤
    string str = removeComments(fin);
    str += "\n";
    vector<Token> result;
    Token token;
    DFA dfa;
    // 有限自动机
    for (char c : str)
    {
        if (dfa.next(c, token))
        {
            result.push_back(token);
        }
    }
    return result;
}
```

（4）有限状态自动机（DFA）的状态转移函数 next 函数，具体实现如下，其接受参数字符 input 和当前状态，返回布尔值 tkSignal，当其为真时，就代表已经获得了一个完整的 token。此外状态机一共有五种状态，代码如下

```cpp
bool DFA::next(char input, Token &buf)
{
    //      Empty,        // space, \n, \r ...
    //     Ident,        // a keyword or identifier, like 'int' 'a0' 'else
    //     IntLiteral,   // int literal, like '1' '1900', only in decimal
    //     FloatLiteral, // float literal, like '0.1'
    //     op            // operators and '{', '[', '(', ',' ...
    bool tkSignal = false;
```

根据输入的类型和当前状态进行转移，转移表见“思路部分”，首先用 swtich 判断当前状态，在判断输入，下面以 State::IntLiteral 下的转移为例

```cpp
    case State::IntLiteral:
        if (isspace(input))···
        else if (isdigit(input) || (input >= 'a' && input <= 'f') ||
         (input >= 'A' && input <= 'F') || input == 'x' || input == 'X')
        else if (input == '.')···
        else if (opIsChar(input))···
```

如上图，其中包括 4 类合法输入，考虑到不同进制整数，在接受到 a-f A-F, 以及 x 和 X 时，都是合法的数字。

当前状态为 IntLiteral 的情况，也就是说，当前正在解析一个整数字面量（例如：123），如果是空白字符，表示当前数字结束。将 buf（代表当

前 Token 的缓冲区）的类型设置为 INTLTR（整数字面量）。调用 reset()函数复位状态。准备解析下一个 Token。设置 tkSignal 为 true，表示成功解析并生成了一个 Token。终止当前的 switch case。

```
if (isspace(input))
{
    buf.type = TokenType::INTLTR;
    buf.value = cur_str;
    reset();
    tkSignal = true;
    break;
}
```

语法分析：

1.4 任务：将 vector<Token>转成一颗抽象语法树，树上的每个节点都表示源代码中的一种结构。

1.5 思路：算法主要参考了编译原理_中科大(华保健)
https://www.bilibili.com/video/BV16h411X7JY/

```
最朴素的自顶向下分析思想是:
tokens[]; // holding all tokens
i = 0; // 指向第i个token
stack = [S] // S是开始符号
while (stack != [])
    if (stack[top] is a terminal t)
        if (t == tokens[i++]) // 如果匹配成功
            pop();
        else
            backtrack();
    else if (stack[top] is a nonterminal T)
        pop();
        push(the next right hand side of T) // 不符合，尝试下一个右部式
```

我们将定义一系列辅助函数 First_XX，这些函数用于计算某个语法规则的 FIRST 集合。FIRST 集合表示从某个非终结符号出发，可以生成的第一个终结符号集合。递归下降法生成抽象语法树：递归下降解析器是一种自顶向下的解析方法，其中每个非终结符号对应一个函数。我们将从左到右解析输入的 token，逐步构建抽象语法树(AST)。如果匹配成功，则继续解析；如果匹配失败，则回退。

1.6 代码与具体实现：

(1) FIRST 集合求解:采用根据表达式进行递归的方式求解，以 CompUnit 为例，其表达式为：

```
CompUnit -> (Decl | FuncDef) [CompUnit]
```

那么可能的分支有两个 Decl 和 FuncDef，合并这两个分支的 FIRST 集就得到了 Compunit 的 FIRST 集

```
std::unordered_set<frontend::TokenType> Parser::First_Compunit()
{
    auto first_decl = First_Decl();
    auto first_funcdef = First_FuncDef();
    first_funcdef.insert(first_decl.begin(), first_decl.end());
    return first_funcdef;
}
```

当递归到终结符时，返回 tokentype 集合，以 Btype 为例

```
    // BType -> 'int' | 'float'
```

由于 Btype 递归到了 int 和 float，已经确定了其 FIRST 集， 所以其函数如下

```
std::unordered_set<frontend::TokenType> Parser::First_BType()
{
    // BType -> 'int' | 'float'
    return {frontend::TokenType::INTTK, frontend::TokenType::FLOATTK};
}
```

(2) 从根节点 CompUnit 开始构造抽象语法树，以 MulExp 的解析为例：

```
// MulExp -> UnaryExp { ('*' | '/' | '%') UnaryExp }
```

据此，一个乘法表达式 MulExp，它由一个 UnaryExp 表达式开始，后跟零个或多个由乘法运算符 (*)、除法运算符 (/) 或取模运算符 (%) 分隔的 UnaryExp 表达式。代码实现如下

```
bool Parser::parseMulExp(AstNode *parent)
{
    MulExp *result = new MulExp(parent);
    if (!parseUnaryExp(result))
    {
        return false;
    }
    while (CUR_TOKEN_IS(MULT) || CUR_TOKEN_IS(DIV) || CUR_TOKEN_IS(MOD))
    {
        NEW_TERM(result);
        if (!parseUnaryExp(result))
        {
            return false;
        }
    }
    return true;
}
```

**实验二**（58/58）接受一颗语法树，进行语义分析、中间代码生成，输出中间表示 IR (Intermediate Representation) 具体实现如下：

1.7 任务：对输入的字符流进行处理，再输出词法单元(Token)流。需要实现 Analyzer 类，完成 ir::Program::get_ir_program(CompUnit*)；接口，该接口接受一个源程序语法树的根节点 Comp*，对其进行分析，返回分析结果 ir::Program。

1.8 思路：

(1)源程序代码中的各种顺序、结构信息都存储在树中，可以通过深度遍历语法树按源程序的顺序来分析源程序。

(2) 不同作用域中同名变量，为变量名加上与作用域相关的前缀（作用域编号），用一张表来存储这个作用域里所有变量的名称和类型

1.9 具体代码实现

(1) SymbolTable：scope_stack 管理所有作用域，functions 存储所有函数，由于实现的没有类方法，所有函数都是全局函数，

```
struct SymbolTable{
    vector<ScopeInfo> scope_stack;
    map<std::string,ir::Function*> functions;
    // 记录递增的作用域序号,每个作用域ScopeInfo的name需要编号，从这里获取
    int blockId=0;
```

(2) ScopeInfo，包括编号、作用域名称，用域入口根节点,block 为一个作用域整体，作用域映射到符号表入口表

```
// 作用域，作用范围
struct ScopeInfo {
    int cnt;    // 编号
    string name;// 作用域的名称
    Block* block;// 作用域入口根节点,block为一个作用域整体
    map_str_ste table;// table将作用域映射到符号表入口

};
```

(3) 符号表入口，符号表用来存储程序中相关变量信息，包括类型，作用域，访问控制信息。map_str_ste 是 string 类型变量名到 STE 的映射

```
struct STE {
    /*
    struct Operand {
    std::string name;变量名
    Type type;变量类型};
    */
    ir::Operand operand;
    vector<int> dimension;//数组操作数
    string literalVal;  // 把立即数值存下来，变量名直接查
};
```

(4) SymbolTable 中包括以下函数等待实现：

add_scope：输入一个新作用域，将信息记录在 scope_stack 中

exit_scope：退出作用域，弹出信息

```
void frontend::SymbolTable::add_scope()
{
    ScopeInfo scopeInfo;
    scopeInfo.cnt = blockId;                    // 当前作用域编号
    scopeInfo.name = "Block" + to_string(blockId); // 作用域编号
    blockId++;
    scope_stack.push_back(scopeInfo);
}

/**...
void frontend::SymbolTable::exit_scope()
{
    scope_stack.pop_back();
}
```

get_scoped_name：获取作用域名称，为了在不同的作用域中处理相同的名称，将 origin id 更改为具有作用域信息的新 id，在这种情况下，我们有两个变量，名称都为“a”，更改后它们将是“a”和“a_block”

```
string frontend::SymbolTable::get_scoped_name(string id) const
{
    for (int i = scope_stack.size() - 1; i >= 0; --i)
    {
        // scope_stack[i].table :map<string,STE>
        if (scope_stack[i].table.find(id) != scope_stack[i].table.end())
            return id + "_" + scope_stack[i].name; // id+作用域名称
    }
}
```

get_operand：使用输入名称获取 operand

```
Operand frontend::SymbolTable::get_operand(string id) const
{
    for (int i = scope_stack.size() - 1; i >= 0; --i)
    {
        if (scope_stack[i].table.find(id) != scope_stack[i].table.end())
            // scope_stack[i].table :map<string,STE>
            // ->second:STE
            return scope_stack[i].table.find(id)->second.operand;
    }
}
```

get_ste：使用输入的名称获得正确的 ste

```
frontend::STE frontend::SymbolTable::get_ste(string id) const
{
    for (int i = scope_stack.size() - 1; i >= 0; --i)
    {
        if (scope_stack[i].table.find(id) != scope_stack[i].table.end())
            // scope_stack[i].table :map<string,STE>
            // ->second:STE
            return scope_stack[i].table.find(id)->second;
    }
}
```

(5) 实验核心工作，需要实现 Analyzer 类，完成 ir::Program get_ir_program(CompUnit*);接口，该接口接受一个源程序语法树的根节点 CompUnit*，对其进行分析，返回分析结果 ir::Program。下面进行详细说明 get_ir_program 的实现。

symbol_table 添加全局作用域，装载全局函数（IO 库函数名称到对应库函数的映射,装载库函数），

```
ir::Program frontend::Analyzer::get_ir_program(CompUnit *root)
{
    // 符号表添加全局作用域
    symbol_table.add_scope();
    ir::Function *globalFunction = new ir::Function("global", ir::Type::nul
    /*...
    symbol_table.functions["global"] = globalFunction;
    map<string, ir::Function *> libFunctions = *get_lib_funcs();
    for (auto iterator = libFunctions.begin(); iterator != libFunctions.end
        symbol_table.functions[iterator->first] = iterator->second;
```

get_lib_funcs 是库函数的名称和函数指针的 map，实现如下

```
map<string, ir::Function *> *frontend::get_lib_funcs()
{
    static map<string, ir::Function *> lib_funcs = {
        {"getint", new Function("getint", Type::Int)},
        {"getch", new Function("getch", Type::Int)},
        {"getfloat", new Function("getfloat", Type::Float)},
        {"getarray", new Function("getarray", {Operand("arr", Type::IntPtr)
        {"getfarray", new Function("getfarray", {Operand("arr", Type::Float
        {"putint", new Function("putint", {Operand("i", Type::Int)}, Type::
        {"putch", new Function("putch", {Operand("i", Type::Int)}, Type::nu
        {"putfloat", new Function("putfloat", {Operand("f", Type::Float)},
        {"putarray", new Function("putarray", {Operand("n", Type::Int), Ope
        {"putfarray", new Function("putfarray", {Operand("n", Type::Int), O
    };
    return &lib_funcs;
}
```

从 compunit 开始 DFS 遍历 AST,详细见(7)

```
    // DFS遍历AST,从compunit开始
    analyzeCompUnit(root);
```

为全局函数添加 return 中间表示

```
// 修改全局函数的return
ir::Instruction *globalreturn = new ir::Instruction(ir::Operand(),
ir::Operand(), ir::Operand(), ir::Operator::_return);
globalFunction->addInst(globalreturn);
irProgram.addFunction(*globalFunction);
```

scope_stack[0].table 作为全局域的 map<string, STE>，将全局变量从 scope_stack[0].table 转移到 irProgram.globalVal，将多维数组还要展平成 1 维，计算长度

```
// scope_stack[0].table: 全局域的map<string, STE>
for (auto it = symbol_table.scope_stack[0].table.begin();
 it != symbol_table.scope_stack[0].table.end(); it++)
{
    /*...
    // int,float,immediate
    if (it->second.dimension.size() == 0)
        irProgram.globalVal.push_back({{symbol_table.get_scoped_name
        (it->second.operand.name), it->second.operand.type}});
    // array
    else
    {
        int maxlen = 1;
        // 将多维数组展平成1维，计算长度
        for (unsigned int i = 0; i < it->second.dimension.size(); i++
            maxlen = maxlen * it->second.dimension[i];
        // ir::GlobalVal::GlobalVal(ir::Operand va, int len)
        irProgram.globalVal.push_back({{symbol_table.get_scoped_name
        (it->second.operand.name), it->second.operand.type}, maxlen})
    }
}
```

最后返回 irProgram

(6)在语义分析这一步，还需要进行进制转换，语法生成树中的整数可能是二进制、八进制、十六进制，检测数字字符串的开头，使用 stoi 的第 3 个参数即可实现进制转换，具体实现如下，

```
int changeToint(const string &input)
{
    int ans;
    // Check if the input starts with "0x" or "0X" and contains only hexadecimal digits
    if ((input.size() > 2 && input.substr(0, 2) == "0x" || input.substr(0, 2) == "0X"))
    {
        ans = std::stoi(input.substr(2), 0, 16);
    }

    // Check if the input starts with "0b" or "0B" and contains only binary digits
    else if ((input.size() > 2 && input.substr(0, 2) == "0b" || input.substr(0, 2) == "0B"))
    {
        ans = std::stoi(input.substr(2), 0, 2);
    }
    else if (input.size() > 1 && input.substr(0, 1) == "0")
    {
        ans = std::stoi(input.substr(1), 0, 8);
    }
    else
        ans = std::stoi(input);
    return ans;
}
```

(7) 从 compunit 开始 DFS 遍历 AST，就以 CompUnit 为例，语法生成树结构的产生是根据下面的表达式：

```
// CompUnit -> (Decl | FuncDef) [CompUnit]
```

因此这里也是当前节点的判断子节点是否为表达式右部，如果有子节点，第一个（下标 0）分析 Decl ,还要声明全局变量，也有可能是 FuncDef，之后，如果子节点大于 1，说明有可选部分，递归调用自身函数。

```
void frontend::Analyzer::analyzeCompUnit(CompUnit *root)
{
    if (root->children.size() == 0)
    {
        return;
    }
    // 分析 Decl ,还要声明全局变量
    if (Decl *decl = dynamic_cast<Decl *>(root->children[0]))
    {
        vector<ir::Instruction *> decl_insts = analyzeDecl(decl);
        for (auto inst : decl_insts)
            symbol_table.functions["global"]->addInst(inst);
    }
    // 分析 FuncDef
    FuncDef *child;
    ANALYSIS_NODE(child, root, FuncDef, 0);
    // 分析可选部分
    if (root->children.size() > 1)
    {
        CompUnit *child;
        ANALYSIS_NODE(child, root, CompUnit, 1);
    }
}
```

(8) analyzeStmt 是整个语义分析最复杂的分析函数，不仅表达式长，而且涉及类型转换（RETURNTK），IFTK 的 cond 涉及到短路运算，后面问题 6 详细讨论

```
//  Stmt -> LVal '=' Exp ';' | Block | 'if' '(' Cond ')' Stmt
// [ 'else' Stmt ] | 'while' '(' Cond ')' Stmt | 'break' ';'
// | 'continue' ';' | 'return' [Exp] ';' | [Exp] ';'
```

其中 BREAKTK 和 CONTINUETK，由于不知道 stmt 的大小和 i 的位置，这里写成__unuse__， 在 WHILE 中处理（检测到 Operand("break")），并替换为 goto 指令

```
if (term->token.type == TokenType::BREAKTK)
{
    Instruction *breakInstruction = new Instruction(Operand("break", Type::null), Operand(),
                                                    Operand(), Operator::__unuse__);
    instrVec.push_back(breakInstruction);
    return instrVec;
}
```

analyzeStmt 整体实现大致如下

```
vector<ir::Instruction *> frontend::Analyzer::analyzeStmt(Stmt *root)
{
    vector<ir::Instruction *> instrVec;
    // LVal '=' Exp ';'
    if (LVal *lval = dynamic_cast<LVal *>(root->children[0])) ···
    // Block
    ANALYSIS_NODE_WITH_RETURN(root, Block, 0);
    // [Exp] ';' ···
    ANALYSIS_NODE_WITH_RETURN(root, Exp, 0);
    // if while break continue return
    Term *term = dynamic_cast<Term *>(root->children[0]);
    // ';'
    if (term->token.type == TokenType::SEMICN)
        return instrVec;
    // return
    if (term->token.type == TokenType::RETURNTK) ···
    // if
    if (term->token.type == TokenType::IFTK) ···
    // 'while' '(' Cond ')' Stmt
    if (term->token.type == TokenType::WHILETK) ···
    // 'break' ';'
    if (term->token.type == TokenType::BREAKTK) ···
    // 'continue' ';'
    if (term->token.type == TokenType::CONTINUETK) ···
}
```

**实验三**（58/58）根据 IR 翻译成为汇编，并能在 rsic-V 上执行

1.10 任务:根据完成目标代码生成，得到 rsic-V 指令集的汇编，即从实验二的 ir::Program 这一入口，完成初始化全局变量（写到.data），初始化函数（先写到.globl 并标记@function，再在代码段将处理 ir::Function 转成汇编指令）。

1.11 思路：生成的汇编文件应 遵循 riscv ABI 规范，这样我们编译器的生成的汇编才可以使用库函数，正确的被加载，并在执行后正确的返回。从实验二的 ir::Program，完成初始化全局变量，包括整型/浮点型变量/数组，在汇编可以使用 .space .word 等伪指令声明。未初始化的全局或静态变量（包括数组）放入 BSS 段。初始化函数：写函数头，清空栈、跳转表、寄存器，统计操作数，将 calleeSavedReg 函数的 stack 关联起来，存函数参数（优先把参数放到寄存器中，对于超过 8 个参数的情况，这些参数不会被放置在寄存器中，而是直接压入栈中），为函数中的操作数分配栈空间，逐一处理函数中的指令（处理指令时，涉及到寄存器的分配和释

放，这里自行实现），最后是函数返回前还原上下文。可以先实现整数的寄存器和相关指令，再做适当修改就可以实现浮点数的寄存器和相关指令。

1.12 具体代码实现

(1)寄存器枚举类和浮点数枚举类：

```
enum class rvREG { ZERO,RA,SP,GP,TP,T0,T1,T2,S0,S1,A0,A1,A2,S5,S6,S7,
A3,A4,A5,A6,A7,S2,S3,S4,S8,S9,S10,S11,T3,T4,T5,T6 };

enum class rvFREG { F0,F1,F2,F3,F4,F5,F6,F7,FS0,FS1,FA0,FA1,FA2,FA3,
FA4,FA5,FA6,FA7,FS2,FS3,FS4,FS5,FS6,FS7,FS8,FS9,FS10,FS11,FT8,FT9,FT10,
FT11};
```

整数寄存器（rvREG）整数寄存器用于存储和操作整数数据。在 RISC-V 架构中，通常有 32 个整数寄存器：

ZERO: 始终为零的寄存器，用于硬件优化和减少特例处理。

RA: 返回地址寄存器，用于存储函数调用的返回地址。

SP: 栈指针寄存器，用于指向当前栈顶。

GP: 全局指针寄存器，指向全局变量。

TP: 线程指针寄存器，指向线程局部存储。

T0-T2: 临时寄存器，短期保存临时数据。

S0-S1: 保存寄存器，保存跨函数调用的数据。

A0-A7: 函数参数寄存器，用于传递函数参数和返回值。

S2-S11: 更多保存寄存器。

T3-T6: 更多临时寄存器。

浮点寄存器（rvFREG）

浮点寄存器用于存储和操作浮点数数据。在 RISC-V 架构中，同样有 32 个浮点寄存器，每个寄存器的名字和功能如下：

F0-F7: 浮点临时寄存器，短期保存临时浮点数据。

FS0-FS1：浮点保存寄存器，保存跨函数调用的浮点数据。

FA0-FA7：浮点函数参数寄存器，用于传递浮点函数参数和返回值。

FS2-FS11：更多浮点保存寄存器。

FT8-FT11：更多浮点临时寄存器。

(2) 变量寻址，用 map 实现根据变量名栈中查找变量

```
struct StackVarMap
{
  unordered_map<string, int> _table;
};
```

(3) 函数调用过程通常分为以下六步

(4) 调用者将参数存储到被调用的函数可以访问到的位置，op1.name 为函数名，des 为函数返回值，首先将获取指令的所有参数，

```
struct CallInst: public Instruction{
    std::vector<Operand> argumentList;
    CallInst(const Operand& op1, std::vector<Operand> paraList, const Operand& des);
    CallInst(const Operand& op1, const Operand& des);   //无参数情况
    std::string draw() const;
};
```

存入对应的寄存器

```
for (size_t i = 0; i < callinst_ptr->argumentList.size(); i++)
{
    if (i <= 7)
    {
        storeOperand(callinst_ptr->argumentList[i], i);
    }
    else
    {
        storeOperandOnStack(callinst_ptr->argumentList[i], i);
    }
}
```

为调用分配栈空间

```
// Set the stack pointer to reserve space for function call parameters
fout << "\taddi\t" << toString(rv::rvREG::SP) << "," << toString(rv::rvREG::SP)
<< "," << -(stackDistrustedBytes) << "\n";

// Call the function specified by callinst_ptr
fout << "\tcall\t" << callinst_ptr->op1.name << "\n";

// Restore the stack pointer to its original position after the function call
fout << "\taddi\t" << toString(rv::rvREG::SP) << "," << toString(rv::rvREG::SP)
<< "," << stackDistrustedBytes << "\n";
```

此外，如果函数有返回值的话，还要存储 A0 和 FA0 寄存的值，避免丢失

```
if (instr.des.type == ir::Type::Int || instr.des.type == ir::Type::IntPtr)
    saveIntOperand(instr.des, rv::rvREG::A0);
else if (instr.des.type != ir::Type::null)
{
    saveFloatOperand(instr.des, rv::rvFREG::FA0);
}
```

(4) 跳转到被调用函数起始位置；

```
fout << "\tcall\t" << callinst_ptr->op1.name << "\n";
```

(5) 被调用函数获取所需要的局部存储资源，按需保存寄存器(callee saved registers)

```
void backend::Generator::saveCalleeRegisters(int &currentOffset, const ir::Function *function)
{
    auto it = this->calleeSavedReg->_calleeSavedReg.begin();
    while (it != this->calleeSavedReg->_calleeSavedReg.end())
    {
        const auto &reg = *it;
        fout << "\tsw\t" << toString(reg) << "," << currentOffset << "(sp)\n";
        stackVarMap->_table["calleeSavedReg" + toString(reg)] = currentOffset;
        currentOffset += 4;
        ++it;
    }
    auto itf = this->calleeSavedFloatReg->_calleeSavedFloatReg.begin();
    while (itf != this->calleeSavedFloatReg->_calleeSavedFloatReg.end())
    {
        const auto &reg = *itf;
        fout << "\tfsw\t" << toString(reg) << "," << currentOffset << "(sp)\n";
        stackVarMap->_table["calleeSavedFloatReg" + toString(reg)] = currentOffset;
        currentOffset += 4;
        ++itf;
    }
}
```

(6) 执行函数中的指令，生成指令，就是根据操作符去调用相应的指令，用 map 对应指令和函数指针

```
unordered_map<ir::Operator, AddFuncPtr> function_map =
    {
        {ir::Operator::add, &Generator::generate_alu},
        {ir::Operator::sub, &Generator::generate_alu},
        {ir::Operator::mul, &Generator::generate_alu},
        {ir::Operator::div, &Generator::generate_alu},
        {ir::Operator::mod, &Generator::generate_alu},
```

(7) 将返回值存储到调用者能够访问到的位置，恢复之前保存的寄存器(callee saved registers)，释放局部存储资源；返回调用函数的位置（调整栈指针的指向）。

```
void backend::Generator::generate__return(const ir::Instruction &instr)
{
    // 根据函数返回值类型，将结果保存在a0或fa0中
    if (instr.op1.type == ir::Type::IntLiteral)···
    else if (instr.op1.type == ir::Type::Int)···
    else if (instr.op1.type == ir::Type::FloatLiteral)···
    else if (instr.op1.type == ir::Type::Float)···
    // 恢复所有被调用者保存的寄存器
    for (auto it = this->calleeSavedReg->_calleeSavedReg.begin(); it != this->calleeSavedReg->_cal
    {
        string savedRegister = "calleeSavedReg" + toString(*it);
        fout << "\tlw\t" << toString(*it) << "," << stackVarMap->_table[savedRegister] << "(sp)\n"
    }
    for (auto it = this->calleeSavedFloatReg->_calleeSavedFloatReg.begin(); it != this->calleeSave
    {
        string savedRegister = "calleeSavedFloatReg" + toString(*it);
        fout << "\tflw\t" << toString(*it) << "," << stackVarMap->_table[savedRegister] << "(sp)\n
    }
    fout << "\taddi\tsp,sp," << this->stack_size << "\n";
    fout << "\tret\n";
}
```

(8) 寄存器分配相关函数

```
> rv::rvREG backend::Generator::allocLoadintReg(ir::Operand operand)···
  // 保存操作数
> void backend::Generator::saveIntOperand(ir::Operand operand, rv::rvREG int_reg)···
  // 分配一个REG
> rv::rvREG backend::Generator::allocIntReg()···
  // 释放寄存器
> void backend::Generator::freeIntReg(rv::rvREG int_reg)···

  // float
> rv::rvFREG backend::Generator::allocLoadfloatReg(ir::Operand operand)···
  // 保存操作数
> void backend::Generator::saveFloatOperand(ir::Operand operand, rv::rvFREG float_reg)···
  // 分配一个REG
> rv::rvFREG backend::Generator::allocFloatReg()···
  // 释放寄存器
> void backend::Generator::freeFloatReg(rv::rvFREG float_reg)···
```

(9) 将 ir::Instruction 都翻译成汇编，包括 alu 运算逻辑运算，访存与指针，调用返回，goto，移位，空，cvt_f2i cvt_i2f

(10)以 cvt_i2f 为例，首先为指令分配寄存器，汇编指令生成，保存结果，释放寄存器

```
void backend::Generator::generate___cvt_i2f(const ir::Instruction &instr)
{
    rv::rvFREG des = allocFloatReg();
    rv::rvREG intReg = allocLoadintReg(instr.op1);
    fout << "\tfcvt.s.w\t" << toString(des) << "," << toString(intReg) << "\n";
    freeIntReg(intReg);
    saveFloatOperand(instr.des, des);
    freeFloatReg(des);
}
```

**实验四(rank0.20)**：IR 和汇编层面的优化

任务：编译优化是编译器的一个重要部分，旨在改善生成的目标代码的质量和性能。通过应用各种优化技术，可以减少程序的执行时间、减少资源消耗，并提高代码的质量和可维护性。

思路：构造 SSA 形式的中间表示——确定基本块，构建控制流图，插入 Φ 函数，行变量重命名，更新使用处；把寄存器当缓存用，基于数据流分析的寄存器分配算法。

方案：通过短路运算提高性能

**选择以下三个问题回答：**

如何处理数组作为参数的情况，为什么可以这样做？

如何支持短路运算？

在函数调用的过程中，汇编需要如何实现，汇编层次下是怎么控制参数传递的？是怎么操作栈指针的？

2、如何处理数组作为参数的情况，为什么可以这样做？

①语义分析时，数组作为函数参数，会根据参数列表的特征[]，其类型会被存储为 IntPtr 或者 FloatPtr，二维数组还要记录第二个维度的长度，代码如下

```
// FuncFParam -> BType Ident ['[' ']' { '[' Exp ']' }]
ir::Operand frontend::Analyzer::analyzeFuncFParam(FuncFParam *root)
{
    // 获取变量类型
    ir::Type type;
    BType *btype;
    ANALYSIS_NODE_WITH_LeftVal(btype, root, BType, 0, type);
    // 获取变量名
    string name = dynamic_cast<Term *>(root->children[1])->token.value;
    vector<int> dimension = {-1};
    // 数组类型参数
    if (root->children.size() > 2)
    {
        // int数组，类型是IntPtr
        if (type == ir::Type::Int)
            type = ir::Type::IntPtr;
        // Float数组，类型是FloatPtr
        else if (type == ir::Type::Float)
            type = ir::Type::FloatPtr;
        // 二维数组 int a [][exp]
        if (root->children.size() == 7)
        {
            Exp *exp;
            vector<Instruction *> instructions;
            ANALYSIS_NODE_WITH_LeftVal(exp, root, Exp, 6, instructions);
            int value = std::stoi(exp->v);
            dimension.push_back(value);
        }
    }
    ir::Operand param(name, type);
    symbol_table.scope_stack.back().table[name] = {param, dimension};
    ir::Operand funcfParamOperand(symbol_table.get_scoped_name(param.name
    return funcfParamOperand;
}
```

IR->汇编时：数组的起始地址会被传递给函数，不论是整数还是浮点数，前 8 个参数通过 a0 到 a7 寄存器传递，剩下的参数通过栈传递。代码如下：

```
for (size_t i = 0; i < function->ParameterList.size(); ++i)
{
    // 优先把参数放到寄存器中
    // 对于超过8个参数的情况，这些参数不会被放置在寄存器中，而是直接压入栈中。
    if (function->ParameterList[i].type == ir::Type::Int || function->ParameterList[i].type
    == ir::Type::IntLiteral || function->ParameterList[i].type == ir::Type::IntPtr
    || function->ParameterList[i].type == ir::Type::FloatPtr)
    {
        if (intNum < intPremRegs._intPremRegs.size())
        {
            fout << "\tsw\t" << toString(this->intPremRegs._intPremRegs[intNum]) << ", " << currentOffset << "(sp)\n";
        }
        else
        {
            // 超过8个的参数已经在调用者的栈中。
            // 被调用函数需要从栈中读取这些参数，并将它们存储到自己的栈帧中。
            // 先从栈取出来，再转移到另一个位置
            fout << "\tlw\tt1, " << this->stack_size + (i - 8) * 4 << "(sp)\n";
            fout << "\tsw\tt1, " << currentOffset << "(sp)\n";
        }
        intNum++;
    }
```

实际上，如果数组比较小，也可以被分解成单个元素来传递

②在编译的语言中，数组作为参数是以数组名进行传递的，例如 fun(int x[])，其中间表示为类型为 IntPtr 或者 FloatPtr，值是指向其第一个元素的指针，而数组在栈的存储上是连续的，且每个元素大小都是 32bit（单精度浮点数和整数），有了第一个元素的地址就可以加上偏移量找到数组元素

## 3、如何支持短路运算？

通常用于逻辑运算符（如 && 和 ||）的计算。当一个逻辑表达式的结果已经确定时，短路运算会立即停止后续运算，以提高效率和避免不必要的计算。

在语义分析这一步进行，这一步会生成的指令序列

对于或运算，其指令顺序应该如下

```
or instruction
true goto
false goto
true des mov 1
ture end goto
false des
```

初值赋 0，一旦为 true 就要跳转至结束——instructions.size() + 1，代码如下

```
// 初始为0，为真赋值1 ，这是由or的特性决定的
string orName = "0";
// 计算or
Instruction *computeInstruction = new Instruction(ir::Operand(root->v, root->t),
                                                  ir::Operand(lorexp->v, lorexp->t),
                                                  ir::Operand(orName, ir::Type::Int), ir::Operator::_or);
instructions.push_back(computeInstruction);
// 真
Instruction *trueGotoInstruction = new Instruction(ir::Operand(root->v, root->t), ir::Operand(),
                                                   ir::Operand("2", Type::IntLiteral), ir::Operator::_goto);
// 假
Instruction *falseGotoInstruction = new Instruction(ir::Operand(), ir::Operand(),
                                                    ir::Operand("3", Type::IntLiteral), ir::Operator::_goto);
instrVec.push_back(trueGotoInstruction);
instrVec.push_back(falseGotoInstruction);
// true :orName赋为1,有一个真结果为真
Instruction *root_true_assign = new Instruction(ir::Operand("1", Type::IntLiteral), ir::Operand(),
                                                ir::Operand(orName, ir::Type::Int), ir::Operator::mov);
instrVec.push_back(root_true_assign);
// true :跳转至结束
Instruction *true_logic_goto = new Instruction(ir::Operand(), ir::Operand(),
                                               ir::Operand(to_string(instructions.size() + 1), Type::IntLiteral), ir::Operator::_goto);
instrVec.push_back(true_logic_goto);

merge(instrVec, instructions);
CHANGE_NODE(root, orName, Type::Int);
```

对于与运算，其指令顺序应该如下

```
and instruction
true goto
false goto
false des mov 1
false end goto
true des
```

初始为 1，一旦为 false 就要跳转至结束——instructions.size() + 1，代码如下

```
string andName = "1";
// 计算or
Instruction *computeInstruction = new Instruction(ir::Operand(root->v, root->t),
                                                  ir::Operand(landexp->v, landexp->t),
                                                  ir::Operand(andName, ir::Type::Int), ir::Operator::_and);
instructions.push_back(computeInstruction);
// 真
Instruction *trueGotoInstruction = new Instruction(ir::Operand(root->v, root->t), ir::Operand(),
                                                   ir::Operand("4", Type::IntLiteral), ir::Operator::_goto);

// 假
Instruction *falseGotoInstruction = new Instruction(ir::Operand(), ir::Operand(),
                                                    ir::Operand("1", Type::IntLiteral), ir::Operator::_goto);
instrVec.push_back(trueGotoInstruction);
instrVec.push_back(falseGotoInstruction);
// false :andName赋为0,有一个假结果为假
Instruction *root_true_assign = new Instruction(ir::Operand("0", Type::IntLiteral), ir::Operand(),
                                                ir::Operand(andName, ir::Type::Int), ir::Operator::mov);
instrVec.push_back(root_true_assign);
// false :跳转至结束
Instruction *true_logic_goto = new Instruction(ir::Operand(), ir::Operand(),
                                               ir::Operand(to_string(instructions.size() + 1), Type::IntLiteral), ir::Operator::_goto);
instrVec.push_back(true_logic_goto);

merge(instrVec, instructions);
CHANGE_NODE(root, andName, Type::Int);
```

4、在函数调用的过程中，汇编需要如何实现，汇编层次下是怎么控制参数传递的？是怎么操作栈指针的？

将函数地址加载进来，跳转到该地址并将返回地址保存在 ra。前 8 个参数通过寄存器 a0 到 a7 传递,超过 8 个参数的部分通过栈传递（栈顶指针+(i-8)*4 内存地址处依次存取），浮点数存到浮点数的函数参数寄存器，。进入函数时通过加法移动栈指针，结束再移动回去。

举例说明，写了一个调用函数的 demo，

```
const float A = 2;
float float_abs(float x) {
  if (x < 0) return -x;
  return x;
}
int main(){
    float a = float_abs(A);
    putfloat(a);
}
```

其汇编文件头部要记录函数名：

```
A_Block0:
    .float  2.000000
    .text
    .globl  float_abs
    .type   float_abs, @function
```

然后是函数的实现，将栈指针-280，为函数分配空间，然后存储寄存器值到栈中

```
float_abs:
    addi    sp,sp,-280
    sw  s0,0(sp)
```

由于已经将 fa0 存入栈中，从 sp 偏移量 100 取函数参数

```
fsw fa0, 100(sp)
li  s11,0x00000000
fmv.w.x ft11,s11
fsw ft11, 104(sp)
flw ft11, 100(sp)
```

将返回值存储到 fa0

```
fmv.s   fa0,ft9
```

函数执行结束，将 sp 移动回去，ret 伪指令（jr ra）

```
    flw fs11,96(sp)
    addi    sp,sp,280
    ret
```

main 调用这个函数，使用伪指令 call，参数前 8 个参数通过寄存器 a0 到 a7 传递,超过 8 个参数的部分通过栈传递（栈顶指针+(i-8)*4 内存地址处依次存）

```
li  s11,0x40000000
fmv.w.x fa0,s11
addi    sp,sp,0
call    float_abs
addi    sp,sp,0
```

## 四、实验测试

1、测试程序是如何运行的？执行了什么命令？你的汇编是如何变成 RISV 程序并被执行的？

（1）测试程序是如何运行的？ 测试组合了编译和执行和对比结果。

具体而言，测试有三个文件 build.py，run.py，score.py，test.py 可供执行：

build.py:  进入到 build 目录，执行 cmake .. & make

run.py: 运行可执行文件 compiler 编译所有测试用例，打印 compiler 返回值和报错，输出编译结果至 /test/output

score.py: 将 run.py 生成的编译结果与标准结果进行对比并打分

test.py 编译生成 compiler 可执行文件，执行并生成结果，

然而在实际使用中，使用 test.py 最为方便 s0/s1/s2 分别对应实验一的两个部分和实验二，实验三 S 参数并不能正常使用，检查复现没写 S

rv_def.cpp | test.py | rv_def.h | generator.cpp

test > test.py > ...

```python
import sys,json
from build import build_compiler
from run import run_compiler
from score import score_compiler

assert(len(sys.argv) == 2)

step = "-" + sys.argv[1]
if step == "-s0":
    oftype = "tk"
elif step == "-s1":
    oftype = "json"
elif step == "-s2":
    oftype = "ir"
else:
    print("illegal input")
    exit()
```

为了方便测试实验 3，编写了一个 test.sh 编译生成 compiler 可执行文件，执行并生成结果，内容如下：

rv_def.cpp | test.sh | rv_def.h | generator.cpp | semantic.h

test > $ test.sh

```
python3 build.py
python3 run.py S
python3 score.py S
```

（2）执行了什么命令？

①编译：首先进入 /build 若 CMakeList 修改后应执行 cmake 命令 1. cd /build 2. cmake .. 如果一切正常没有报错 执行 make 命令 3. Make。

值得注意的是，根据指导书的提示，需要在 CMakLists.txt 中设置 CMAKE_C_COMPILER 和 CMAKE_CXX_COMPILER 的路径。

```
set(CMAKE_C_COMPILER    "/usr/bin/x86_64-linux-gnu-gcc-7")
set(CMAKE_CXX_COMPILER  "/usr/bin/x86_64-linux-gnu-g++-7")
```

而且 cmake_minimum_required 也需要根据报错信息自行修改，最后 (VERSION 3.10)在我的 WSL2 的 ubuntu 上的 docker 中可以正常运行

②执行：

1. cd /bin 2. compiler <src_filename> [-step] -o <output_filename> [-O1]

-step: 支持以下几种输入 s0: 词法结果 token 串

s1: 语法分析结果语法树，以 json 格式输出

s2: 语义分析结果，以 IR 程序形式输出

e : 执行 IR 测评机，从 xx.sy 读入源文件，重定向 xx.in 作为 IR 程序

的标准输入，并将 IR 的标准输出输出到 <output_filename> 中

S : RISC-v 汇编

(3) 你的汇编是如何变成 RISV 程序并被执行的?

实验三在执行 python3 run.py S 后会生成对应汇编文件，执行 python3 score.py S 就会生成.exe 和输出文件，根据 score.py，首先会执行一条命令：gcc 将汇编文件生成目标代码.o,与静态链接库.a 连接，-o 代表了生成可执行程序

```
cmd = ' '.join(["riscv32-unknown-linux-gnu-gcc", output_dir + file, "sylib-riscv-linux.a", '-o', exec_file])
os.system(cmd)
```

之后执行生成的可执行程序，使用 qemu-riscv32.sh 执行.exe，接收.in 输入，将输出定向到.out 文件。

```
# qemu
cmd = ' '.join(["qemu-riscv32.sh", exec_file])
input_file = testcase_dir + fname + ".in"
if os.path.exists(input_file):
    cmd = ' '.join([cmd, "<", input_file])
cmd = ' '.join([cmd, ">", output_file])
cp = subprocess.run(cmd, shell=True, stderr=subprocess.PIPE, stdout=subprocess.DEVNULL)
with open(output_file, "a") as f:
    f.write("\n" + str(cp.returncode))
```

## 五、实验总结

实验过程中所遇到的问题及解决办法

（1）实验二浮点数，windows 环境下（个人电脑环境问题），导致 16 进制浮点数 C++一直读 0

解决方案：用 ubuntu 拉取 docker 镜像后的环境，就可以正常读取 16 进制浮点数了。

（2）实验三中，浮点数转整数的处理是向下取整而非四舍五入

解决办法 fcvt.w.s 指令后写 rst,保证即可。

（3）在实验三遇到 out 不正确，.s 太长难以定位错误

解决办法：从测试用例中进一步抽取 demo，执行，用 fout 的指针输出每条指令，使得.s 阅读起来更加容易，或者直接修改.s，添加一些输出观察中间变量。

（4）16 进制浮点数难以判断其值

解决方案：写一个 java 脚本

```java
public class Main {
    public static void main(String[] args) {
        // 16进制表示的浮点数字符串
        String hexFloatString = "0x1.7c21fcp+6"; // 这个例子表示3.1415926 * 2^3

        // 使用Double.parseDouble解析16进制浮点数字符串
        double result = Double.parseDouble(hexFloatString);

        // 输出结果
        System.out.println("Hex Float String: " + hexFloatString);
        System.out.println("Double: " + result);
    }

}
```

对实验的建议

（1）实验深入思考后，整体方案不难，难度主要在细节和实验量上，建议多出点实验细节的教程，可以发到 b 站视频里，理论和实践向结合，结合的桥梁不仅需要学生的思考，也需要课程关于实验的帮助。

（2）实验一有作业一和二铺垫，好评。

（3）实验量上，特别是实验三和在期末和夏令营冲突，对学生的关键发展（升学）有很大的阻碍作用，建议如果在大三下开设课程的话，减少实验量，增加学时（也就是减少别的课程时间），或者换一个学期开设，大三下由于毕业实现时间太短，各门课程太密集。

（4） 实验四虽然是可选，但课程提供帮助有限，不太符合成绩来考察课程掌握程度的，建议以后不要计入成绩，加分是支持的，毕竟能反应学生水平。

学到了很多东西，个人感觉是重大最充实，实验设计水平最高的一门课，收获很大，是可以写到简历中项目经历里的高质量实验，真心感谢老师和历届助教和师兄师姐。