重庆大学编译原理课程实验报告

年级、专业、班级	2022 级弘深计算机拔尖班	姓名	鲁永卓
实验题目	编译器设计与实现		
实验时间	2024.6.15	实验地点	DS1402
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. 实现了实验一全部功能和实验二、实验三的绝大部分功能，即词法分析和语法分析并构建语法树、对语法树进行语义分析并生成中间代码、根据中间代码翻译成 RISC v 程序，仅对浮点数支持不完全。最终结果为实验一 100 分，实验二 98 分，实验三 98 分。

2. CMakelist 中通过以下代码使用 IR 库：
使用静态库链接的方法：

```
# --------------------- from lib ---------------------
# link libxx.a
# u should rename libxx-x86-win.a or libxx-x86-linux.a to libxx.a
according to ur own platform
link_directories(./lib)
# --------------------- from lib ---------------------
```

使用源代码的方法(我不用这个，就注释了)：

```
# --------------------- from src ---------------------
# aux_source_directory(./src/ir IR_SRC)
# add_library(IR ${IR_SRC})
# aux_source_directory(./src/tools TOOLS_SRC)
# add_library(Tools ${TOOLS_SRC})
# --------------------- from src ---------------------
```

3. IR 中使用 ir::Program.globalVal 记录全局变量的名称、类型和长度（常量长度设为 0）。创建 global 函数，在这个函数中加入全局变量的初始化语句，并在 main 函数的开头调用 global 函数。
以下是 analyzeCompunit 函数中对全局变量的记录，因为只有 Compunit 节点导出的 Decl 是且一定是全局变量的声明。

```
    if(MATCH_CHILD_TYPE(DECL, 0)){
        GET_CHILD_PTR(decl, Decl, 0);
        analyzeDecl(decl, buffer.functions.back().InstVec);
        // 记录全局变量
        for(size_t i = 0; i < decl->n.size(); ++i){
            auto type = decl->t;
            if(decl->size[i] > 0) type = (type == Type::Int) ?
Type::IntPtr : Type::FloatPtr; // 如果是数组，类型为指针
            buffer.globalVal.push_back(ir::GlobalVal(Operand(decl->n[i],
type), decl->size[i]));
        }
    }
```

后端的好处大概是只需要在汇编程序开头的 data 字段声明即可，值的初始化实际上跟普通变量使用同一个逻辑，而声明所需的全部信息已经记录在 ir::Program.globalVal 中了。ir::Program.globalVal 会传递到后端。
后端对全局变量的初始化在上面说过了，只要 global 函数正确实现就不需要额外操作。代码如下：

```
    fout << ".data\n";
    for (const auto& gv : program.globalVal) {
        if (gv.maxlen > 0) {
            // array
            fout << gv.val.name << ": .space " << gv.maxlen * 4 << "\n";
        } else {
            fout << gv.val.name << ": .word 0\n";
        }
    }
```

调用全局变量时由于其地址和局部变量存在栈中不同，需要先判断变量是否是全局变量，是则使用 la 指令获取地址。至于判断，检查 ir::Program.globalVal 这个 vector。示例代码如下：

```
    bool isGlobalArray = false;
    for (const auto& gv : program.globalVal) {
        if (gv.val.name == arg.name && gv.maxlen > 0) {
            isGlobalArray = true;
            break;
        }
    }
    if (isGlobalArray) {
        fout << "  la t0, " << arg.name << "   # get global array
address\n";
        fout << "  addi sp, sp, -4\n";
        fout << "  sw t0, 0(sp)        # push array address\n";
```

4. 函数调用的过程在 IR 中即为 call 指令。在 ir_executor.cpp 中我们可以找到 call 指令的解析逻辑，其处理流程如下：
    1. 提取函数名。
    2. 检查是否为库函数，如果是则执行库函数的调用。然后退出。
    3. 在 program->functions 中查找目标函数，创建新的执行上下文。
    4. 类型检查，验证返回值类型和参数类型是否匹配。
    5. 参数传递，将实参值存入新上下文的内存映射。
    6. 当前上下文入栈，切换到新上下文执行。

5. 在 IR 生成时：

analyzeFuncFParam 是分析每个形参的函数。分析时不能只记录参数名和类型，还需要把数组参数除第一维外的每一维大小记录下来，以便后续使用时(LVal)能够正确计算下标偏移。第一次写的时候偷懒没记维度，结果写到 LVal 的时候又得回来改。

```
    std::string name =
symbol_table.get_scoped_name(ident->token.value);
    auto type = btype->t;
    vector<int> dims = {};
    if(root->children.size() > 2){
        type = type == Type::Int ? Type::IntPtr : Type::FloatPtr; // 数
组
        dims.push_back(1);
        for(int i = 5; i < root->children.size(); i += 3){
            if(MATCH_CHILD_TYPE(EXP, i)){
                GET_CHILD_PTR(exp, Exp, i);
                analyzeExp(exp, buffer.InstVec);
                assert(exp->is_computable && exp->value >= 0 && "Exp
must be positive integer");
                dims.push_back(exp->value);
```

```
            }
            else{
                assert(0 && "FuncFParam error: expected Exp");
            }
        }
    }
    buffer.ParameterList.push_back(Operand(name, type));
    symbol_table.scope_stack.back().table[name] = STE(Operand(name,
type), dims);
```

analyzeLVal 中的使用如下，因为要生成 IR，并且每次进行计算都得弄个临时变量，所以长了点。

```
    // 数组变量
    // 计算线性偏移
    string offsetVar;
    for(size_t k = 0, index = 2; k < dims.size(); ++k, index += 3){
        GET_CHILD_PTR(exp, Exp, index);
        analyzeExp(exp, buffer);
        // 计算本维度的 stride
        int stride = 1;
        for(size_t j = dims.size() - 1; j > k; --j) stride *= dims[j];
        // idx_k * stride
        string part = getTmpName();
        if(stride == 1){
            buffer.push_back(new Instruction(Operand(exp->v, exp->t),
Operand(), Operand(part, Type::Int), Operator::def));
        }
        else{
            Operand t1, t2;
            if(exp->t == Type::IntLiteral){
                t1 = Operand(getTmpName(), Type::Int);
                buffer.push_back(new Instruction(Operand(exp->v,
Type::IntLiteral), Operand(), t1, Operator::def));
            }
            else if(exp->t == Type::Int) t1 = Operand(exp->v,
Type::Int);
            else assert(0 && "LVal error: Exp type must be Int or
IntLiteral");
            t2 = Operand(getTmpName(), Type::Int);
            buffer.push_back(new
Instruction(Operand(std::to_string(stride), Type::IntLiteral),
Operand(), t2, Operator::def));
            buffer.push_back(new Instruction(t1, t2, Operand(part,
Type::Int), Operator::mul));
        }
        // 累加到 offsetVar
        if(k == 0) offsetVar = part;
        else{
            string sum = getTmpName();
            buffer.push_back(new Instruction(Operand(offsetVar,
Type::Int), Operand(part, Type::Int), Operand(sum, Type::Int),
Operator::add));
            offsetVar = sum;
        }
    }
```

在汇编生成时：

首先明确数组的传递均为地址传递，值为数组首地址。

传参时是翻译 IR 的 call 指令。首先判断数组类型是全局数组、当前函数内定义的数组还是当前函数的数组参数，判断方式为：若 program.globalVal 中存在且 maxlen>0，则为全局数组；若函数内有 alloc 指令分配该数组，则为当前函数内定义的数组；前两类均不是则为当前函数的数组参数（这其实可以等同于普通指针变量处理）。全局数组通过 la 获取地址；当前函数内定义的数组则计算当前栈指针与数组在栈中的偏移量相加；当前函数的数组参数就直接取其值。存入传参的寄存器(a0~a7，若更多则使用栈传递)即可完成传参。
使用时，由于参数值即为数组首地址，因此加偏移量即可得到数组元素的地址。
为什么可以这样做：地址对了就对了嘛。

6. 处理短路：
    1. 先把 result 置为短路结果（or 为 1，and 为 0）。
    2. 计算第一项表达式的结果 res1，并生成一个条件跳转。若满足短路条件（or 为 res1!=0，and 为 res1==0），则跳转目标为整个表达式末尾；否则跳到第二项表达式计算的开头。
    3. 计算第二项表达式的结果 res2，并赋给 result。
    4. 此时在整个表达式末尾。把 result 赋给 root->v。

以 and 为例，代码如下：

```
    // 如果有 '&& next'
    if (root->children.size() > 1) {
        // 为整个 and 的结果准备一个临时变量，初始化为 0（短路结果）
        std::string result = getTmpName();
        buffer.push_back(new Instruction(
            Operand("0", Type::IntLiteral), Operand(), Operand(result,
Type::Int), Operator::def
        ));

        // 如果 lhs == 0，则短路到 end
        std::string notVar = getTmpName();
        buffer.push_back(new Instruction(
            Operand(lhs->v, Type::Int), Operand(), Operand(notVar,
Type::Int), Operator::_not
        ));
        int jmpPos = (int)buffer.size();
        auto shortJump = new Instruction(
            Operand(notVar, Type::Int), Operand(), Operand("",
Type::IntLiteral), Operator::_goto
        );
        buffer.push_back(shortJump);

        // 2. eval rhs
        GET_CHILD_PTR(rhs, LAndExp, 2);
        analyzeLAndExp(rhs, buffer);
        // 将 rhs 的值覆盖 result
        buffer.push_back(new Instruction(
            Operand(rhs->v, Type::Int), Operand(), Operand(result,
Type::Int), Operator::mov
        ));

        // 3. patch shortJump 跳到 end
        int endPos = (int)buffer.size();
        shortJump->des = Operand(std::to_string(endPos - jmpPos),
Type::IntLiteral);
```

```
        // 4. 写回 root
        root->v = result;
        root->t = Type::Int;
        root->is_computable = false;
    }
```

8. 每个函数都会生成一个标签，调用时 jal ra function_name 即可。
传参：
    1. 使用寄存器 a0~a7。
    2. 若参数超过 8 个则使用栈传递：调用方将第 9 个及之后的参数压至当前栈顶（虽然 sp 是-4），跳转后被调用方会先分配一个 frame 的栈空间(sp-=frame)，那么被调用方计算参数地址的方法如下：

```
    int callerArgOffset = (i - 8) * 4;
    int totalOffset = frame + callerArgOffset;
```

栈指针控制策略：
    1. 固定栈帧大小：每个函数分配固定 2044 字节栈空间。
    2. 栈向下增长：sp 减小表示分配空间，增大表示释放空间。
    3. 栈空间管理：调用方负责清理栈上的参数。

四、实验测试

测试程序 test.py 先编译项目得到 compiler，然后运行 compiler 对每个测试用例进行编译得到输出结果，再与标准输出进行比较，若一致则该测试点通过。其中实验三的-S 选项还需将汇编代码通过 docker 内置的 RISC v 编译器和模拟器进行编译运行。

执行命令：
1. 创建并运行容器
    docker run -it -v D:\MyRepos\Compilers\lab3:/cg frankd35/demo:v3
2. 进入 test 目录
    cd cg/test
3. 进行测试打分，test.py 包括编译、运行和打分
    python3 test.py [s1/s2]

其中第三步是 lab1 和 lab2 用的，lab3 的-S 选项在 test.py 里没写。
以及 build.py 每次都要删除 bin 和 build 再重新生成，对于一个急着看 bug 修好没的人来说太慢了，因此双开命令行，一个在 build 目录下 make，一个在 test 目录下运行 python3 run.py S && echo && python3 score.py S。另外写了个脚本把 compiler 编译、IR 生成、汇编生成、汇编编译运行打包一键运行。
其他用到的命令：
4. 编译单个测试用例
    (在 bin 目录下) ./compiler test.sy -S -o test.s
5. 将汇编代码编译运行，写下面那问里了

汇编变成 RISC v 程序并执行的方法：使用 docker 内置的 RISC v 编译器和模拟器。
(在 bin 目录下)

```
riscv32-unknown-linux-gnu-gcc test.S sylib-riscv-linux.a -o test.exe
qemu-riscv32.sh test.exe < test.in > test.out
```

五、实验总结

遇到问题：

首先最大的问题是在每个实验开始时，看完指导书好像理解了什么，好像又什么都没理解，写起代码来还是大脑一片空白。lab1 比较简单，还能硬啃指导书和框架写出来。lab2 和 lab3 实在想不明白，只能去 GitHub 上学习。好在这个实验在入门后难度会越来越小，写着写着就发现已经差不多理解了，思路也越来越清晰。

Debug 时的问题非常非常多，基本上是对着测试点一个个调试的，Git 的提交记录也可见一斑，写的时候完全想不到有这么多种可能的情况。这么多问题并不能一一记录（当然我也不能全都想起来），选取两个印象最深的问题列下。

1. 在 73_int_io 这个测试点中，源程序非常简单，实现了忽略其他字符的整数输入输出。但是汇编程序运行后的输出里出现许多乱码。一开始想的是检查 while 和 if 嵌套且带 continue 和 break 的跳转出问题，写了几个简单测试发现并不是；又想是不是取模运算出问题，通过打印中间值检查，发现也没问题。灵光一现把 i 的中间值一并打印，发现 i 的值异常，但是源程序对 i 的修改很明显不会导致这个结果，因此推测是其他变量的赋值导致 i 的改变，也就是某个变量的地址与 i 重叠。排查下来的确是 b[2]与 i 的值始终相同，它们的地址重叠。最终发现 bug 原因是 add_operand 实现中手滑写反了顺序。

```
    next_offset += static_cast<int>(size);
    _table[op] = next_offset;
```

应该是

```
   _table[op] = next_offset;
   next_offset += static_cast<int>(size);
```

2. 在 64_calculator 这个测试点中，汇编代码编译运行后总是段错误。最开始以为是数组太大了导致访存出问题，改成只有大数组的简单测试却正常运行。经过很久的排查后发现是分配的栈空间不够，在这些简单程序中每个函数分配 1024 字节都不够确实不好想象，并且之前出过一次同样的问题，当时以为开大到 1024 完全够用了。这个问题的正确做法应该是动态分配栈空间，但是固定大小实在方便。改成 2048 字节后又发现 RISC v 不支持大于 2047 的立即数，再改成 2044 发现还有别的地方会用到 2047 以上，再把这个地方修了就好了。

建议：

1. 加强入门引导，也许要指导到一两个函数的实现。指导书看完真的云里雾里。
2. 指导书里的 IR 定义语焉不详，例如 load 和 store 只写了一维数组的情况，二维数组的偏移得靠猜（即使很符合直觉和常识）。
3. 框架似乎有点小问题。能想起来的一个，IR 中浮点数的大小比较结果要求用浮点数存（这个要求是从框架里的 assert 得知的），但是逻辑表达式的结果很明显是 int(bool)类型，难道是汇编需要这样实现？