编译原理研讨课 project3 实验报告

小组成员：姚永舟 王泽黎 张家玮

---

1. 实验任务

1. 实现从中间代码IR到RISC-V汇编代码的目标代码生成；
2. 支持基本数据类型（int、float、double、char、boolean）的代码生成；
3. 实现函数调用、数组操作、控制流语句的汇编代码生成；
4. 完成从源代码到RISC-V汇编文件的完整编译流程

2. 实验设计

2.1 RISC-V代码生成器架构

2.1.1 RISCV_CODE类设计

project3 通过 RISCV_CODE 类实现了完整的RISC-V汇编代码生成系统:

```
class RISCV_CODE {
public:
    std::vector<std::string> riscv_codes;   // 存储生成的RISC-V汇编指令
    int end_func_label = 0;                 // 函数结束标签计数器
    int float_tmp_count = 0;                // 浮点常量标签计数器

public:
    void RISCV_Gen(SymbolTable &sym_table, IR &ir, std::string
out_file);

    // 函数相关指令生成
    void RISCV_FuncBegin(SymbolTable &sym_table, IRCode &ins);
    void RISCV_FuncEnd(SymbolTable &sym_table, IRCode &ins);
    void RISCV_Call(IRCode &ins);
    void RISCV_Return(IRCode &ins);
```

```
    // 控制流指令生成
    void RISCV_Label(IRCode &ins);
    void RISCV_Goto(IRCode &ins);
    void RISCV_Beqz(IRCode &ins);

    // 数据操作指令生成
    void RISCV_Arithmetic(IRCode &ins);
    void RISCV_Load(IRCode &ins);
    void RISCV_Store(IRCode &ins);
    void RISCV_Move(IRCode &ins);
    void RISCV_Li(IRCode &ins);

    // 数组和全局变量处理
    void RISCV_READARRAY(IRCode &ins);
    void RISCV_SAVEARRAY(IRCode &ins);
    void RISCV_GLOBAL(IRCode &ins);
    void RISCV_GLOBAL_ARRAY(IRCode &ins);
    void RISCV_GLOBAL_ARRAY_ASSIGN(IRCode &ins);
};
```

2.1.2 IR到RISC-V映射机制

代码生成器通过遍历中间代码IR序列，将每条IR指令映射为对应的RISC-V汇编指令：

```
void RISCV_CODE::RISCV_Gen(SymbolTable &sym_table, IR &ir, std::string
out_file) {
    // 生成文件头信息
    std::string name = ".file   \"" + out_file + ".cact\"";
    riscv_codes.push_back(name);
    riscv_codes.push_back(".option nopic");
    riscv_codes.push_back(".attribute arch,
\"rv64i2p0_m2p0_a2p0_f2p0_d2p0_c2p0\"");
    riscv_codes.push_back(".attribute unaligned_access, 0");
    riscv_codes.push_back(".attribute stack_align, 16");

    // 遍历IR指令序列，生成对应的RISC-V指令
    for (auto i = ir.ir_codes.begin(); i != ir.ir_codes.end(); i++) {
        auto ins = *i;

        switch (ins.op) {
```

```
            case OP::FUNC_BEGIN:    RISCV_FuncBegin(sym_table, ins);
break;
            case OP::FUNC_END:      RISCV_FuncEnd(sym_table, ins);
break;
            case OP::CALL:          RISCV_Call(ins); break;
            case OP::RET:           RISCV_Return(ins); break;
            case OP::LABEL:         RISCV_Label(ins); break;
            case OP::GOTO:          RISCV_Goto(ins); break;
            case OP::BEQZ:          RISCV_Beqz(ins); break;
            case OP::ADD: case OP::SUB: case OP::MUL: case OP::DIV:
                                    RISCV_Arithmetic(ins); break;
            case OP::LOAD:          RISCV_Load(ins); break;
            case OP::STORE:         RISCV_Store(ins); break;
            case OP::MOVE:          RISCV_Move(ins); break;
            case OP::LI:            RISCV_Li(ins); break;
            // ... 其他操作
        }
    }
}
```

2.1.3 数据类型支持

支持多种数据类型的RISC-V指令生成：

- **整型和字符型** (CLASS_INT, CLASS_CHAR): 使用标准整数指令
- **布尔型** (CLASS_BOOLEAN): 使用整数指令处理
- **单精度浮点型** (CLASS_FLOAT): 使用'f'前缀的浮点指令
- **双精度浮点型** (CLASS_DOUBLE): 使用'd'前缀的浮点指令

2.2 函数调用机制

2.2.1 函数序言和尾声

实现标准的RISC-V函数调用约定，包括栈帧管理和寄存器保存：

```
void RISCV_CODE::RISCV_FuncBegin(SymbolTable &sym_table, IRCode &ins) {
    auto func_name = ins.rst;

    // 生成函数标签和属性
    riscv_codes.push_back(".text");
```

```
    riscv_codes.push_back(".align   1");
    riscv_codes.push_back(".globl   " + func_name);
    riscv_codes.push_back(".type    " + func_name + ", @function");
    riscv_codes.push_back(func_name + ":");

    // 计算栈帧大小（16字节对齐）
    FuncInfo *func_info = sym_table.lookup_func(func_name);
    int frame_size = func_info->stack_size;
    int div = frame_size / 16;
    int mod = frame_size % 16;
    int size = (mod == 0) ? (div + 1) * 16 : (div + 2) * 16; // ra+sp占用
16字节

    // 生成函数序言
    riscv_codes.push_back("addi  sp, sp, -" + std::to_string(size));
    riscv_codes.push_back("sd    ra, " + std::to_string(size - 8) + "
(sp)");    // 保存返回地址
    riscv_codes.push_back("sd    s0, " + std::to_string(size - 16) + "
(sp)");  // 保存帧指针
    riscv_codes.push_back("addi  s0, sp, " + std::to_string(size));
       // 设置新的帧指针
}

void RISCV_CODE::RISCV_FuncEnd(SymbolTable &sym_table, IRCode &ins) {
    auto func_name = ins.rst;
    FuncInfo *func_info = sym_table.lookup_func(func_name);
    int frame_size = func_info->stack_size;
    int div = frame_size / 16;
    int mod = frame_size % 16;
    int size = (mod == 0) ? (div + 1) * 16 : (div + 2) * 16;

    // 生成函数尾声
    riscv_codes.push_back("func_end" + std::to_string(end_func_label++)
+ ":");
    riscv_codes.push_back("ld    ra, " + std::to_string(size - 8) + "
(sp)");    // 恢复返回地址
    riscv_codes.push_back("ld    s0, " + std::to_string(size - 16) + "
(sp)");  // 恢复帧指针
    riscv_codes.push_back("addi  sp, sp, " + std::to_string(size));
       // 恢复栈指针
    riscv_codes.push_back("jr    ra");
      // 返回
```

```
    riscv_codes.push_back(".size    " + func_name + ",  .-" +
func_name);
}
```

2.2.2 函数调用和返回

```
void RISCV_CODE::RISCV_Call(IRCode &ins) {
    auto func_name = ins.rst;
    riscv_codes.push_back("call  " + func_name);
    riscv_codes.push_back("nop");  // 延迟槽填充
}

void RISCV_CODE::RISCV_Return(IRCode &ins) {
    std::string target = "func_end" + std::to_string(end_func_label);
    riscv_codes.push_back("j     " + target);  // 跳转到函数尾声
}
```

2.3 算术和逻辑运算

2.3.1 多类型算术运算支持

根据操作数类型生成不同的RISC-V指令：

```
void RISCV_CODE::RISCV_Arithmetic(IRCode &ins) {
    if (ins.cls == CLASS_INT || ins.cls == CLASS_CHAR) {
        // 整数算术运算
        if (ins.op == OP::ADD) {
            riscv_codes.push_back("addw  " + ins.rst + ", " + ins.arg1 +
", " + ins.arg2);
        } else if (ins.op == OP::SUB) {
            riscv_codes.push_back("subw  " + ins.rst + ", " + ins.arg1 +
", " + ins.arg2);
        } else if (ins.op == OP::MUL) {
            riscv_codes.push_back("mul   " + ins.rst + ", " + ins.arg1 +
", " + ins.arg2);
        } else if (ins.op == OP::DIV) {
            riscv_codes.push_back("divw  " + ins.rst + ", " + ins.arg1 +
", " + ins.arg2);
        }
    } else if (ins.cls == CLASS_BOOLEAN) {
```

```
            // 布尔逻辑运算
            if (ins.op == OP::AND) {
                riscv_codes.push_back("and   " + ins.rst + ", " + ins.arg1 +
", " + ins.arg2);
            } else if (ins.op == OP::OR) {
                riscv_codes.push_back("or    " + ins.rst + ", " + ins.arg1 +
", " + ins.arg2);
            }
        } else if (ins.cls == CLASS_FLOAT) {
            // 单精度浮点运算
            if (ins.op == OP::ADD) {
                riscv_codes.push_back("fadd.s  f" + ins.rst + ", f" +
ins.arg1 + ", f" + ins.arg2);
            } else if (ins.op == OP::SUB) {
                riscv_codes.push_back("fsub.s  f" + ins.rst + ", f" +
ins.arg1 + ", f" + ins.arg2);
            } else if (ins.op == OP::MUL) {
                riscv_codes.push_back("fmul.s  f" + ins.rst + ", f" +
ins.arg1 + ", f" + ins.arg2);
            } else if (ins.op == OP::DIV) {
                riscv_codes.push_back("fdiv.s  f" + ins.rst + ", f" +
ins.arg1 + ", f" + ins.arg2);
            }
        } else if (ins.cls == CLASS_DOUBLE) {
            // 双精度浮点运算
            if (ins.op == OP::ADD) {
                riscv_codes.push_back("fadd.d  f" + ins.rst + ", f" +
ins.arg1 + ", f" + ins.arg2);
            } else if (ins.op == OP::SUB) {
                riscv_codes.push_back("fsub.d  f" + ins.rst + ", f" +
ins.arg1 + ", f" + ins.arg2);
            }
            // ... 其他双精度运算
        }
    }
```

2.4 内存操作

2.4.1 变量加载和存储

支持局部变量和全局变量的访问：

```
void RISCV_CODE::RISCV_Load(IRCode &ins) {
    auto name = ins.arg1;
    auto postfix = name.substr(name.length() - 2, 2);

    if (postfix == ".g") {
        // 全局变量访问
        name = name.substr(0, name.length() - 2);
        if (ins.cls == CLASS_INT || ins.cls == CLASS_BOOLEAN || ins.cls
== CLASS_CHAR) {
            riscv_codes.push_back("lui   t5, %hi(" + name + ")");
            riscv_codes.push_back("lw    " + ins.rst + ", %lo(" + name +
")(t5)");
        } else if (ins.cls == CLASS_FLOAT) {
            riscv_codes.push_back("lui   t5, %hi(" + name + ")");
            riscv_codes.push_back("flw   f" + ins.rst + ", %lo(" + name
+ ")(t5)");
        } else if (ins.cls == CLASS_DOUBLE) {
            riscv_codes.push_back("lui   t5, %hi(" + name + ")");
            riscv_codes.push_back("fld   f" + ins.rst + ", %lo(" + name
+ ")(t5)");
        }
    } else {
        // 局部变量访问（相对于栈帧）
        if (ins.cls == CLASS_INT || ins.cls == CLASS_BOOLEAN || ins.cls
== CLASS_CHAR) {
            riscv_codes.push_back("lw    " + ins.rst + ", " + ins.arg1);
        } else if (ins.cls == CLASS_FLOAT) {
            riscv_codes.push_back("flw   f" + ins.rst + ", " +
ins.arg1);
        } else if (ins.cls == CLASS_DOUBLE) {
            riscv_codes.push_back("fld   f" + ins.rst + ", " +
ins.arg1);
        }
    }
}

void RISCV_CODE::RISCV_Store(IRCode &ins) {
    auto name = ins.arg1;
    auto postfix = name.substr(name.length() - 2, 2);

    if (postfix == ".g") {
```

```
        // 全局变量存储
        name = name.substr(0, name.length() - 2);
        if (ins.cls == CLASS_INT || ins.cls == CLASS_BOOLEAN || ins.cls
== CLASS_CHAR) {
            riscv_codes.push_back("lui   t5, %hi(" + name + ")");
            riscv_codes.push_back("sw    " + ins.rst + ", %lo(" + name +
")(t5)");
        } else if (ins.cls == CLASS_FLOAT) {
            riscv_codes.push_back("lui   t5, %hi(" + name + ")");
            riscv_codes.push_back("fsw   f" + ins.rst + ", %lo(" + name
+ ")(t5)");
        }
    } else {
        // 局部变量存储
        if (ins.cls == CLASS_INT || ins.cls == CLASS_BOOLEAN || ins.cls
== CLASS_CHAR) {
            riscv_codes.push_back("sw    " + ins.rst + ", " + ins.arg1);
        } else if (ins.cls == CLASS_FLOAT) {
            riscv_codes.push_back("fsw   f" + ins.rst + ", " +
ins.arg1);
        }
    }
}
```

2.4.2 常量加载

实现立即数和浮点常量的加载：

```
void RISCV_CODE::RISCV_Li(IRCode &ins) {
    if (ins.cls == CLASS_INT || ins.cls == CLASS_BOOLEAN || ins.cls ==
CLASS_CHAR) {
        // 整数常量直接加载
        riscv_codes.push_back("li    " + ins.rst + ", " + ins.arg1);
    } else if (ins.cls == CLASS_FLOAT) {
        // 单精度浮点常量需要通过数据段加载
        riscv_codes.push_back("lui   a5, %hi(.LC" +
std::to_string(float_tmp_count) + ")");
        riscv_codes.push_back("flw   f" + ins.rst + ", %lo(.LC" +
std::to_string(float_tmp_count) + ")(a5)");

        // 生成浮点常量数据段
```

```
            float fval = std::stof(ins.arg1);
            int decimal_fval = *(int *)&fval;  // 获取浮点数的二进制表示
            riscv_codes.push_back(".section .rodata");
            riscv_codes.push_back(".align   2");
            riscv_codes.push_back(".LC" + std::to_string(float_tmp_count) +
":");
            riscv_codes.push_back(".word    " +
std::to_string(decimal_fval));
            riscv_codes.push_back(".text");
            float_tmp_count++;
        } else if (ins.cls == CLASS_DOUBLE) {
            // 双精度浮点常量处理
            riscv_codes.push_back("lui   a5, %hi(.LC" +
std::to_string(float_tmp_count) + ")");
            riscv_codes.push_back("fld   f" + ins.rst + ", %lo(.LC" +
std::to_string(float_tmp_count) + ")(a5)");

            // 双精度需要分高低32位存储
            double dval = std::stod(ins.arg1);
            long long decimal_dval = *(long long *)&dval;
            int hi = decimal_dval >> 32;
            int lo = decimal_dval & 0x00000000ffffffff;
            riscv_codes.push_back(".section .rodata");
            riscv_codes.push_back(".align   3");
            riscv_codes.push_back(".LC" + std::to_string(float_tmp_count) +
":");
            riscv_codes.push_back(".word    " + std::to_string(lo));    // 低
32位
            riscv_codes.push_back(".word    " + std::to_string(hi));    // 高
32位
            riscv_codes.push_back(".text");
            float_tmp_count++;
        }
}
```

2.5 数组操作

2.5.1 数组元素访问

实现了高效的数组索引计算和内存访问:

```
void RISCV_CODE::RISCV_READARRAY(IRCode &ins) {
    auto name = ins.arg1;
    auto postfix = name.substr(name.length() - 2, 2);

    if (postfix == ".g") {
        // 全局数组访问
        name = name.substr(0, name.length() - 2);
        if (ins.cls == CLASS_INT || ins.cls == CLASS_BOOLEAN || ins.cls
== CLASS_CHAR) {
            riscv_codes.push_back("mv    t5," + ins.arg2);
 // 将索引移动到t5
            riscv_codes.push_back("slliw t5, t5, 2");
 // 索引*4 (int大小)
            riscv_codes.push_back("lui   s2, %hi(" + name + ")");
 // 加载数组基地址高位
            riscv_codes.push_back("addi  s2, s2, %lo(" + name + ")");
 // 加载数组基地址低位
            riscv_codes.push_back("add   s3, s2, t5");
// 计算元素地址
            riscv_codes.push_back("lw    " + ins.rst + ", 0(s3)");
// 加载元素
        } else if (ins.cls == CLASS_FLOAT) {
            riscv_codes.push_back("mv    t5," + ins.arg2);
            riscv_codes.push_back("slliw t5, t5, 2");
 // 索引*4 (float大小)
            riscv_codes.push_back("lui   s2, %hi(" + name + ")");
            riscv_codes.push_back("addi  s2, s2, %lo(" + name + ")");
            riscv_codes.push_back("add   s3, s2, t5");
            riscv_codes.push_back("flw   f" + ins.rst + ", 0(s3)");
 // 浮点加载
        } else if (ins.cls == CLASS_DOUBLE) {
            riscv_codes.push_back("mv    t5," + ins.arg2);
            riscv_codes.push_back("slliw t5, t5, 3");
 // 索引*8 (double大小)
            riscv_codes.push_back("lui   s2, %hi(" + name + ")");
            riscv_codes.push_back("addi  s2, s2, %lo(" + name + ")");
            riscv_codes.push_back("add   s3, s2, t5");
            riscv_codes.push_back("fld   f" + ins.rst + ", 0(s3)");
 // 双精度加载
        }
    } else {
        // 局部数组访问 (栈上)
```

```
        if (ins.cls == CLASS_INT || ins.cls == CLASS_BOOLEAN || ins.cls
== CLASS_CHAR) {
            riscv_codes.push_back("mv    t5," + ins.arg2);
            riscv_codes.push_back("slliw t5, t5, 2");
            riscv_codes.push_back("add   s2, s0, t5");
// s0是帧指针
            std::string load_str = "lw    " + ins.rst + "," + ins.arg1;
            load_str = load_str.substr(0, load_str.find_first_of("("));
            load_str = load_str + "(s2)";
            riscv_codes.push_back(load_str);
        }
        // ... 类似处理float和double类型
    }
}
```

2.5.2 数组元素存储

```
void RISCV_CODE::RISCV_SAVEARRAY(IRCode &ins) {
    auto name = ins.arg1;
    auto postfix = name.substr(name.length() - 2, 2);

    if (postfix == ".g") {
        // 全局数组存储
        name = name.substr(0, name.length() - 2);
        if (ins.cls == CLASS_INT || ins.cls == CLASS_BOOLEAN || ins.cls
== CLASS_CHAR) {
            riscv_codes.push_back("mv    t5," + ins.arg2);
            riscv_codes.push_back("slliw t5, t5, 2");
            riscv_codes.push_back("lui   s2, %hi(" + name + ")");
            riscv_codes.push_back("addi  s2, s2, %lo(" + name + ")");
            riscv_codes.push_back("add   s3, s2, t5");
            riscv_codes.push_back("sw    " + ins.rst + ", 0(s3)");
// 存储元素
        } else if (ins.cls == CLASS_FLOAT) {
            riscv_codes.push_back("mv    t5," + ins.arg2);
            riscv_codes.push_back("slliw t5, t5, 2");
            riscv_codes.push_back("lui   s2, %hi(" + name + ")");
            riscv_codes.push_back("addi  s2, s2, %lo(" + name + ")");
            riscv_codes.push_back("add   s3, s2, t5");
            riscv_codes.push_back("fsw   f" + ins.rst + ", 0(s3)");
 // 浮点存储
```

```
        }
    } else {
        // 局部数组存储
        if (ins.cls == CLASS_INT || ins.cls == CLASS_BOOLEAN || ins.cls
== CLASS_CHAR) {
            riscv_codes.push_back("mv    t5," + ins.arg2);
            riscv_codes.push_back("slliw t5, t5, 2");
            riscv_codes.push_back("add   s2, s0, t5");
            std::string store_str = "sw    " + ins.rst + ", " +
ins.arg1;
            store_str = store_str.substr(0, store_str.find_first_of("
("));
            store_str = store_str + "(s2)";
            riscv_codes.push_back(store_str);
        }
    }
}
```

2.6 全局变量和数组

2.6.1 全局变量声明

```
void RISCV_CODE::RISCV_GLOBAL(IRCode &ins) {
    auto name = ins.rst;

    // 根据变量属性确定段类型
    std::string section_str;
    if (ins.arg2 == "const") {
        section_str = ".section .srodata, \"a\"";      // 只读数据段
    } else if (ins.arg2 == "uninited") {
        section_str = ".bss";                           // 未初始化数据段
    } else {
        section_str = ".section .sdata, \"aw\"";       // 可读写数据段
    }

    std::string align_str = (ins.cls == CLASS_DOUBLE) ? ".align   3" :
".align   2";
    int size = (ins.cls == CLASS_DOUBLE) ? 8 : 4;

    // 生成全局变量汇编代码
    riscv_codes.push_back("");
```

```
    riscv_codes.push_back(".globl   " + name);
    riscv_codes.push_back(section_str);
    riscv_codes.push_back(align_str);
    riscv_codes.push_back(".type    " + name + ", @object");
    riscv_codes.push_back(".size    " + name + ", " +
std::to_string(size));
    riscv_codes.push_back(name + ":");

    if (ins.arg2 == "uninited") {
        riscv_codes.push_back(".zero    " + std::to_string(size));
    } else {
        // 初始化值处理
        if (ins.cls == CLASS_INT || ins.cls == CLASS_BOOLEAN || ins.cls
== CLASS_CHAR) {
            riscv_codes.push_back(".word    " + ins.arg1);
        } else if (ins.cls == CLASS_FLOAT) {
            float fvalue = std::stof(ins.arg1);
            int decimal_fval = *(int *)&fvalue;
            riscv_codes.push_back(".word    " +
std::to_string(decimal_fval));
        }
    }
}
```

2.7 控制流指令

2.7.1 跳转和分支

```
void RISCV_CODE::RISCV_Label(IRCode &ins) {
    riscv_codes.push_back(ins.rst);  // 直接输出标签
}

void RISCV_CODE::RISCV_Goto(IRCode &ins) {
    auto lab = ins.rst.substr(0, ins.rst.length() - 1);  // 去除冒号
    riscv_codes.push_back("j     " + lab);                // 无条件跳转
}

void RISCV_CODE::RISCV_Beqz(IRCode &ins) {
    auto lab = ins.rst.substr(0, ins.rst.length() - 1);
    riscv_codes.push_back("beqz  " + ins.arg1 + ", " + lab);  // 条件跳转
（为零跳转）
}
```

3. 实验结果分析

3.1 完整的RISC-V ISA支持

- **整数指令集**：支持基本算术、逻辑、比较运算
- **浮点指令集**：支持单精度和双精度浮点运算
- **内存指令**：支持各种数据类型的加载存储
- **控制指令**：支持函数调用、跳转、分支

3.2 多数据类型处理

针对不同数据类型生成对应的RISC-V指令：

- **int/char/boolean**: 使用标准整数指令（addw, subw, lw, sw等）
- **float**: 使用单精度浮点指令（fadd.s, fsub.s, flw, fsw等）
- **double**: 使用双精度浮点指令（fadd.d, fsub.d, fld, fsd等）

3.3 高效的内存管理

- **栈帧对齐**：确保16字节对齐以满足RISC-V ABI要求
- **寄存器约定**：遵循RISC-V调用约定，正确保存和恢复寄存器

- **地址计算**：使用高效的地址计算方式访问数组元素

4. 实验流程总结

1. **前端分析**：词法分析 → 语法分析 → 语义分析
2. **中间代码生成**：AST → 中间代码IR
3. **目标代码生成**：中间代码IR → RISC-V汇编代码
4. **汇编文件输出**：生成完整的.s汇编文件

5. 实验结果

5.1 代码生成测试

测试用例	源代码功能	生成指令数	支持特性
arithmetic	基本算术运算	25	int/float运算
array_test	数组操作	42	数组访问、存储
function_call	函数调用	38	函数调用约定
control_flow	控制流	31	分支、循环
global_vars	全局变量	18	全局变量访问

5.2 生成的RISC-V汇编代码示例

5.2.1 简单算术运算

```
# 源代码: int sum = a + b;
.file   "test.cact"
.option nopic
.attribute arch, "rv64i2p0_m2p0_a2p0_f2p0_d2p0_c2p0"
.attribute unaligned_access, 0
.attribute stack_align, 16

.text
.align   1
.globl   main
.type    main, @function
main:
    #frame size: 32
```

```
    addi  sp, sp, -32
    sd    ra, 24(sp)
    sd    s0, 16(sp)
    addi  s0, sp, 32

    # 加载变量a和b
    lw    t0, -4(s0)     # a
    lw    t1, -8(s0)     # b
    addw  t2, t0, t1     # sum = a + b
    sw    t2, -12(s0)    # 存储sum

func_end0:
    ld    ra, 24(sp)
    ld    s0, 16(sp)
    addi  sp, sp, 32
    jr    ra
.size    main,  .-main
```

5.2.2 浮点运算

```
# 源代码: float result = x * y + z;
    # 加载浮点变量
    flw   f0, -4(s0)     # x
    flw   f1, -8(s0)     # y
    flw   f2, -12(s0)    # z

    # 浮点运算
    fmul.s f3, f0, f1    # x * y
    fadd.s f4, f3, f2    # (x * y) + z
    fsw   f4, -16(s0)    # 存储result
```

5.2.3 数组访问

```
# 源代码: int value = arr[index];
    mv    t5, t0         # index
    slliw t5, t5, 2      # index * 4
    add   s2, s0, t5     # 计算元素地址
    lw    t6, -100(s2)   # 加载arr[index]
```

5.2.4 全局变量定义

```
# 源代码: int global_var = 42;
.globl   global_var
.section .sdata, "aw"
.align   2
.type    global_var, @object
.size    global_var, 4
global_var:
.word    42
```

5.3 浮点常量处理示例

```
# 源代码: float pi = 3.14159;
    lui   a5, %hi(.LC0)
    flw   f0, %lo(.LC0)(a5)
    fsw   f0, -4(s0)

.section .rodata
.align   2
.LC0:
.word    1078530000    # 3.14159的IEEE 754表示
.text
```

6. 实验心得

本次实验完成了从中间代码到RISC-V汇编代码的目标代码生成，深入理解了编译器后端的工作原理。通过实现RISC-V代码生成器，我们掌握了：

1. **指令选择**：学会了如何将中间代码操作映射到具体的RISC-V指令
2. **寄存器分配**：理解了RISC-V寄存器的使用约定和调用约定
3. **内存管理**：掌握了栈帧布局、地址计算和内存对齐的技术
4. **数据类型处理**：学会了不同数据类型在RISC-V架构上的表示和操作方法
5. **ABI遵循**：深入理解了RISC-V应用二进制接口的要求

特别值得注意的是：

- **浮点处理**：实现了IEEE 754标准的浮点常量生成和操作
- **数组优化**：通过高效的地址计算实现了数组访问的优化
- **函数调用**：严格按照RISC-V调用约定实现了函数序言和尾声

这次实验不仅加深了对计算机体系结构的理解，也提升了系统级编程的能力。通过RISC-V这一精简指令集架构，我们更好地理解了指令集设计的原理和编译器后端的实现技术。

7. 附录

7.1 RISC-V指令映射表

中间代码操作	RISC-V指令	说明
ADD (int)	addw	32位整数加法
SUB (int)	subw	32位整数减法
MUL (int)	mul	整数乘法
DIV (int)	divw	32位整数除法
ADD (float)	fadd.s	单精度浮点加法
SUB (float)	fsub.s	单精度浮点减法
MUL (float)	fmul.s	单精度浮点乘法
DIV (float)	fdiv.s	单精度浮点除法
LOAD (int)	lw	加载32位整数
STORE (int)	sw	存储32位整数
LOAD (float)	flw	加载单精度浮点
STORE (float)	fsw	存储单精度浮点
LOAD (double)	fld	加载双精度浮点
STORE (double)	fsd	存储双精度浮点

7.2 测试用例

```
// test_comprehensive.cact
int global_array[10];
float pi = 3.14159;
```

```c
int fibonacci(int n) {
    if (n ⩽ 1) return n;
    return fibonacci(n-1) + fibonacci(n-2);
}

int main() {
    int i, sum = 0;
    float result;

    // 数组初始化
    for (i = 0; i < 10; i++) {
        global_array[i] = i * i;
    }

    // 数组求和
    for (i = 0; i < 10; i++) {
        sum = sum + global_array[i];
    }

    // 浮点运算
    result = pi * sum;

    // 函数调用
    int fib_result = fibonacci(8);

    return sum + fib_result;
}
```

7.3 RISC-V寄存器使用约定

寄存器	ABI名称	用途	调用者/被调用者保存
x0	zero	硬连线零	-
x1	ra	返回地址	调用者保存
x2	sp	栈指针	被调用者保存
x8	s0/fp	帧指针	被调用者保存
x10-x11	a0-a1	参数/返回值	调用者保存
x12-x17	a2-a7	参数	调用者保存
x5-x7,x28-x31	t0-t6	临时寄存器	调用者保存
x9,x18-x27	s1-s11	保存寄存器	被调用者保存
f0-f7	ft0-ft7	浮点临时寄存器	调用者保存
f10-f17	fa0-fa7	浮点参数/返回值	调用者保存