实验报告：编译器后端实现——RISC-V 代码生成器

组别：12 组
成员：吴亚晨　祝田田　全荟霖

一、实验目的

本实验的主要目的是实现一个完整的 RISC-V 代码生成器，能够将中间代码（IR）转换为 RISC-V 汇编代码。通过该实验，深入理解编译器后端的各个组成部分，包括寄存器分配、指令选择、代码优化和目标代码生成等关键步骤。

二、实验步骤

1. 整体架构设计与模块职责划分

本次实验的目标是将自定义的中间表示转换为合法的 RISC-V 汇编程序。代码生成器的核心流程由 RiscVCodeGen.cpp 实现，而具体的汇编指令、操作数、寄存器等实体由 RiscVInstr.cpp 提供支持。整个流程大致分为以下几个子模块：

（1）指令对象表示与打印（RiscVInstr 系列）

所有 RISC-V 汇编指令都继承自基类 RiscVInstr，并按类型细分为：

RTypeInstr: 三寄存器算术指令（如 add, sub, slt）；
ITypeInstr: 两寄存器一立即数指令（如 addi, lw, jalr）；
STypeInstr: 存储类指令（如 sw, sb）；
BTypeInstr: 条件跳转（如 beq, blt）；
UTypeInstr: 只用高 20 位的指令（如 lui）；
TypeInstr: 非条件跳转（如 jal, j）；
LabelInstr: 标签伪指令（如 loop_start:）；

每类指令根据其格式通过构造函数添加合适的 Operand 类型（如寄存器、立即数、标签等）；

所有指令重载了 toString() 方法用于打印为标准 RISC-V 格式。

（2）寄存器与栈分配模块（RegisterAllocator）

统一管理整数和浮点寄存器池的分配和释放；
当寄存器不足时，支持自动将虚拟寄存器映射到栈上的偏移位置；
为中间代码的每个虚拟寄存器分配唯一的物理位置；
支持函数参数和局部变量在栈帧中的分配。

（3）代码生成主流程（RiscVCodeGen::generateFunction()）

每个函数翻译为一个指令列表：

1. 生成入口标签；
2. 分配函数参数寄存器/栈位置；
3. 遍历基本块及其内指令，调用 generateInstr() 生成对应汇编；
4. 追加返回语句与函数结尾处理（如释放栈帧、返回）；
5. 输出最终指令列表为合法汇编文本。

（4）指令翻译策略（generateInstr()）

每条 MIR 指令通过类型分发策略处理，例如：

add, sub, mul → R 型指令；
load, store → I 型/S 型指令；
ret, br → J 型/B 型指令；

const → li, lui+addi 等加载指令；

特别处理函数调用、跳转、返回、比较、加载立即数等复杂场景。

2. 操作数与指令建模细节

（1）操作数 Operand 类设计

操作数是所有汇编指令的基本构成单位，涵盖立即数、寄存器、标签、地址偏移等多种形式。为支持这些功能，Operand 类采用如下枚举和数据结构：

```
enum OperandType {
    IMM,       // 立即数
    REG,       // 寄存器（x0 ~ x31, f0 ~ f31）
    LABEL,     // 标签，如 loop_start
    STACK_SLOT // 栈槽，表示为 offset(sp)
};
```

成员变量设计：

OperandType type：标明具体类型；

int immVal：若为立即数，记录其数值；

std::string regName：寄存器名（如 x5, a0, s1）；

std::string label：标签名称；

int offset：相对 sp 偏移值（用于栈槽）；

操作数的打印由 Operand::toString() 统一完成，自动区分格式：

IMM → 42

REG → a0

LABEL → loop_end

STACK_SLOT → -8(sp)

这种建模确保了操作数能独立构造、复用并在最终指令打印中格式正确。

（2）指令 RiscVInstr 继承体系

本实验构造了一个以 RiscVInstr 为基类的多态结构，各类具体指令继承并实现自己的打印逻辑。主要派生类如下：

RTypeInstr（R 型）：如 add a0, a1, a2，包含 3 个寄存器操作数，适用于 add, sub, mul, slt 等指令；

ITypeInstr（I 型）：如 addi a0, a1, 4，用于立即数操作、加载、函数调用等；

STypeInstr（S 型）：如 sw a0, 0(sp)，用于存储类操作；

BTypeInstr（B 型）：如 beq a0, a1, label，条件跳转；

UTypeInstr（U 型）：如 lui a0, 4096，高 20 位加载；

JTypeInstr（J 型）：如 jal ra, func_label，非条件跳转；

LabelInstr（伪指令）：如 func_entry: 打印标签定义；

DirectiveInstr（伪指令）：如 .text, .globl main，汇编指令区和全局标记声明。

每类指令构造时传入必要的操作数，并统一调用 toString() 输出格式。

3. 指令生成逻辑与策略分发

在完成指令和操作数的建模后，核心任务是将中间表示正确翻译为对应的 RISC-V 指令序列。本实验在 RiscVCodeGen.cpp 中实现了指令生成的主流程。

（1）指令分发机制

RiscVCodeGen 类中的主函数 gen() 负责遍历中间代码列表（MIR），并根据操作类型（如 ADD, SUB, LOAD, STORE, CALL, RET, BR 等）调用对应的处理函数。每类操作指令均有一个对应的

genXXX() 方法，如：

genAdd()：生成 add 或 addi 指令；

genLoad()：根据是否为局部变量或全局变量选择使用 lw 或 la；

genCall()：生成 jal 指令，同时处理参数传递；

genBr()：处理条件跳转，如 beq, bne, blt 等；

genRet()：生成返回值 mv a0, t0 与 ret。

这样通过分发机制，将逻辑清晰地分离并易于维护。

（2）立即数与地址加载策略

针对 RISC-V 指令集对立即数和地址的限制（如 12 位立即数上限），本实验实现了以下策略：

若立即数超出范围，则采用 lui 加 addi 组合加载；

若加载全局变量地址，使用 la 指令；

栈变量访问使用偏移形式，如 lw t0, -4(sp)；

支持浮点寄存器的 load/store（如 flw, fsw）与整数指令分开处理。

（3）标签与跳转处理

对分支和函数跳转的指令，会自动生成并插入对应的标签（如 label1:），跳转目标通过 Operand 类型 LABEL 表示。跳转指令统一采用 jal 或条件跳转 beq, bne, blt 等构造，确保控制流正确。

4.代码细节展示

（1）RegisterAllocator 类负责寄存器的分配和管理，为每个虚拟寄存器分配一个物理寄存器，并处理寄存器溢出的情况。

```
class RegisterAllocator {
public:
    RegisterAllocator();
    void reset();
    Register allocateIntReg();
    Register allocateFloatReg();
    void freeReg(Register reg);
    int allocateStack(int size);

private:
    std::map<Register, bool> intRegUsed;
    std::map<Register, bool> floatRegUsed;
    int stackOffset;
    std::set<Register> availableIntRegs = {Register::T0, Register::T1, Register::T2, Register::T3, Register::T4, Register::T5, Register::T6, Register::S2, Register::S3};
    std::set<Register> availableFloatRegs = {Register::FT0, Register::FT1, Register::FT2, Register::FT3, Register::FS2, Register::FS3};
};
```

（2）RiscVCodeGen 类 负责生成具体的 RISC-V 汇编代码，包括函数序言、基本块代码、指令选择等。

```
class RiscVCodeGen {
```

```
public:
    RiscVCodeGen(Manager* mgr);
    void generateCode();
    void outputAssembly(const std::string& filename);
    void outputAssembly(std::ostream& stream);

private:
    void generateFunction(Function* func);
    void generateFunctionPrologue(Function* func);
    void generateFunctionEpilogue(Function* func);
    void generateBasicBlock(BasicBlock* bb);
    void generateInstruction(Instr* instr);
    void generateAlu(INSTR::Alu* alu);
    void generateIcmp(INSTR::Icmp* icmp);
    void generateLoad(INSTR::Load* load);
    void generateStore(INSTR::Store* store);
    void generateReturn(INSTR::Return* ret);

    VirtualRegInfo& getOrCreateVReg(Value* value);
    Register getPhysicalReg(Value* value);
    void emitInstr(std::shared_ptr<RiscVInstr> instr);
    void emitLabel(const std::string& label);
    bool isFloatType(Type* type);
    int getTypeSize(Type* type);
    RiscV::InstrType getLoadInstr(Type* type);
    RiscV::InstrType getStoreInstr(Type* type);
    void loadConstant(Register dest, int value);
    void loadFromMemory(Register dest, Register base, int offset, Type* type);
    void calculateStackFrame(Function* func);

    Manager* manager;
    Function* currentFunction;
    int currentFunctionStackSize;
    RegisterAllocator regAlloc;
    std::unordered_map<Value*, VirtualRegInfo> valueToVReg;
    int nextLabelId;
    std::vector<std::shared_ptr<RiscVInstr>> instructions;
};
```

（3）指令生成示例

以下是一个生成 add 指令的示例：

```
void RiscVCodeGen::generateAlu(INSTR::Alu* alu) {
    Register dest = getPhysicalReg(alu);
    Register src1 = getPhysicalReg(alu->getRVal1());
```

```
    Register src2 = getPhysicalReg(alu->getRVal2());
    emitInstr(std::make_shared<RTypeInstr>(InstrType::ADD, dest, src1, src2));
}
```

（4）寄存器分配示例

以下是一个分配整数寄存器的示例：

```
Register RegisterAllocator::allocateIntReg() {
    for (auto reg : availableIntRegs) {
        if (!intRegUsed[reg]) {
            intRegUsed[reg] = true;
            return reg;
        }
    }
    return Register::ZERO; // 无可用寄存器，需要溢出
}
```

5．测试与验证

编写测试用例，验证生成的 RISC-V 汇编代码是否正确。例如，测试一个简单的加法操作：

```
void testAddition() {
    Instr* addInstr = new INSTR::Alu(INSTR::Alu::Op::ADD, Script::create_int32_type(),
funcParams[0], funcParams[1], returnReg);
    RiscVCodeGen codeGen(manager);
    codeGen.generateInstruction(addInstr);
    codeGen.outputAssembly("output.s");
}
```

生成的汇编代码类似于：

```
    add a0, a1, a2
```

三、实验总结

本次实验通过构建一个完整的 RISC-V 汇编生成器，初步实现了从中间表示（MIR）到目标代码的翻译过程。整个流程涵盖了操作数与指令建模、语义映射策略设计、指令生成分发、跳转控制逻辑管理等多个关键环节，体现了编译器后端的基本框架与实现思路。

实验过程中，我深入理解了 RISC-V 指令集结构、立即数处理限制、寄存器调用约定等底层机制，并结合中间代码的语义逐步完成了从抽象语义到具体机器指令的映射。通过多个样例程序的测试与调试，验证了生成器的功能正确性，也锻炼了调试与排错的能力。

总体而言，本实验不仅加深了我对编译器后端原理的理解，也让我体会到将抽象语义映射到具体硬件指令的复杂性与系统性，对我今后进一步学习系统软件和程序设计语言实现具有重要意义。