编译原理研讨课 PR003 实验报告

小组成员：

杜旭蕾 2022K8009929003　王锦如 2022K8009929022

目录

一.实验任务

由实验 2 生成的 IR 生成 riscv 汇编代码。

二.实验思路

整体思路是先生成一个中间表示 MIR，其结构 llvmIR 相似，都是 program→func→block→instruction，其 instruction 是 MIR 指令格式，生成时只进行虚拟寄存器的分配。MIR 生成完成之后，通过 instruction 的一对一映射为 riscv 汇编指令，并同时进行寄存器分配。

我们小组写代码的思路是先从最基础的测试用例出发将这两部分的内容一点一点完善，但是在这个过程中，由于从一开始就没有添加浮点指令的支持，使得通过 test32.cact 非常困难，最后经过很多调试，我们只能做到让 32.cact 生成能跑起来的 riscv 汇编，但是跑出来的结果是 3 个 error 和 1 个 ok，和标准的 gcc 跑出来的结果不符。这个问题我们改了一天，没有改出来，因此放弃测试 32.cact。代码可以通过其它所有测试。

本次实验的代码结构如下：

```
/cact
|—— /build
     |—— build.sh             # 生成 compiler 的脚本
     |—— libcact_rt.a         # 链接库，包函 print_int 等
     |—— output.ll            # 生成的 IR 文件
     |—— output.mir           # 生成的 MIR 文件
     |—— output.s             # 生成的 riscv 汇编文件
|—— /include
     |—— Analysis.h           # 实验 2 使用：生成 IR 的代码的头文件
     |—— MIR.h                # 生成 MIR 代码的头文件
     |—— StackFrame.h         # 生成 MIR 代码时进行栈管理
     |—— SymbolTable.h        # 实验 2 使用：符号表管理
|—— /src
     |—— Analysis.cpp         # 实验 2 使用：生成 IR
     |—— generateMachineIR.cpp #生成 MIR
     |—— generateRISCV.cpp    # 生成 riscv 汇编，但是由于头文件 include 问题，全部放到了 main.cpp 中
     |—— main.cpp             # mian 函数和生成 riscv 汇编需要的函数
```

三.MIR 的代码实现

1. MIR 的数据结构

我们实现的 MIR 的数据结构和 llvm::Ir 类似，具有如下数据结构。首先最底层是 Minstruction，其有一个 opcode、3 个操作数和用于函数变量初始化的数组 ExtraDatum 和判断这个 Minstruction 是不是含有浮点操作数的 isfloat 组成。其对应的是 llvm::Instruction 数据结构，在每一个 block 里，遍历每一条 llvm::Instruction 可以生成对应的 1 条或多条 Minstruction，把这些 Minstruction 顺序存入 MIRBlock 中。

```
struct ExtraDatum {
    std::string name;
    bool isFloat;

    ExtraDatum(const std::string &name, bool isFloat)
        : name(name), isFloat(isFloat) {}
};

struct MInstruction {
    MOpcode op;
    std::string dst;
    std::string src1;
    std::string src2;
    std::vector<ExtraDatum> extraData;
    bool isfloat;

    MInstruction(MOpcode op,
                 const std::string &dst = "",
                 const std::string &src1 = "",
                 const std::string &src2 = "",
                 const std::vector<ExtraDatum> &extraData = {},
                 bool isfloat = false)
        : op(op), dst(dst), src1(src1), src2(src2), extraData(extraData), isfloat(isfloat) {}
};
```

MIRBlock 的结构如下所示，其主要包含块名和块中的 Minstruction 序列。

```
struct MIRBlock {
    std::string label;
    std::vector<MInstruction> instructions;

    void addInstruction(const MInstruction& inst) {
        instructions.push_back(inst);
    }
};
```

MIRFunction 的结构如下，主要包括函数名和块序列，以及函数是否返回浮点数的标志位。最上层的 MIRProgram 结构由一个函数序列和全局变量和外部函数声明序列和静态浮点数序列构成，结构如下。

```
struct MIRFunction {
```

```
    std::string name;
    bool returnFloat; // 是否返回浮点数
    std::vector<MIRBlock> blocks;
};

struct MIRfloat {
    std::string symbol;  // 符号名
    double value;        // 浮点数值
};

struct MIRProgram {
    std::vector<MIRFunction> functions;
    std::vector<MInstruction> globalInstructions; // 保存全局变量和外部函数声明
    std::vector<MIRfloat> staticFloats; //保存静态浮点数
};
```

除了以上数据结构之外，本次实验使用以下 MIR 的指令类型。

```
enum class MOpcode {
    // 函数与标签
    LABEL,
    EXTERN_FUNC,
    VAR_GLOBAL,

    // 数据传输
    LOAD_INT,   STORE_INT,     // int: lw / sw
    LOAD_FLOAT, STORE_FLOAT,   // float: flw / fsw
    LOAD_CHAR,  STORE_CHAR,    // char: lb / sb

    // 算术运算
    ADD_INT, SUB_INT, MUL_INT, DIV_INT,
    ADD_FLOAT, SUB_FLOAT, MUL_FLOAT, DIV_FLOAT,

    // 比较操作（整数）
    ICMP_EQ_INT, ICMP_NE_INT,
    ICMP_LT_INT, ICMP_LE_INT,
    ICMP_GT_INT, ICMP_GE_INT,

    // 比较操作（浮点）
    ICMP_EQ_FLOAT, ICMP_NE_FLOAT,
    ICMP_LT_FLOAT, ICMP_LE_FLOAT,
    ICMP_GT_FLOAT, ICMP_GE_FLOAT,

    // 控制流
    CALL,
    RET,
    JMP,     // 无条件跳转
    BR,      // 条件跳转

    // 条件跳转伪指令
    BEQ, BNE, BGT, BLT, BGE, BLE,
```

```
    // 其他伪操作
    MOV_INT,         // int 类型寄存器赋值 -> mv
    MOV_FLOAT,       // float 类型寄存器赋值 -> fmv.s
    MOV_CHAR,        // char 类型（本质是 int8） -> mv + 截断（可选）
    GETPTR,          // 获取地址（GEP 等）
    LA ,             // Load Address（加载地址）
    LI ,             // Load Immediate（加载立即数）
    STATIC_FLOAT,    // 静态浮点数
    SREM             // 取模（整数）
};
```

2. 函数栈帧的实现思路

函数栈帧在进入一个函数时插入栈中，函数栈帧和存放函数栈帧的栈的结构如下。其中，函数栈帧主要由函数栈帧的大小、函数中需要存入栈中的局部变量表和函数需要保存的寄存器表组成。存放函数栈帧的栈就是由一组函数栈帧构成，支持插入和删除操作。

```
struct StackFrame {
    int frameSize = 0;
    std::map<Variable, int> localVarOffsets;
    std::vector<Register> calleeSavedRegs;

    // 栈帧信息打印
    void debugPrint() const;
};

class Stack {
    private:
        std::vector<StackFrame> frames;

    public:
        void push(const StackFrame& frame) {
            frames.push_back(frame);
        }

        void pop() {
            if (frames.empty()) throw std::runtime_error("Stack underflow");
            frames.pop_back();
        }

        StackFrame& top() {
            if (frames.empty()) throw std::runtime_error("Stack is empty");
            return frames.back();
        }

        const StackFrame& top() const {
            if (frames.empty()) throw std::runtime_error("Stack is empty");
            return frames.back();
        }
```

```
        bool empty() const {
            return frames.empty();
        }

        size_t size() const {
            return frames.size();
        }
    };
```

在函数栈帧生成时，需要使用以下函数获取函数局部变量的大小，加上函数需要保存的寄存器大小之后，就得到了函数栈帧的大小。其核心思想就是遍历函数的所有块找到所有的 alloca 语句，获取 alloca 语句生成的局部变量的大小，然后将其存储到函数栈的 size 字段中。

```
int calculateLocalVarSize(const llvm::Function &func, StackFrame &frame) {
    const llvm::DataLayout &DL = func.getParent()->getDataLayout();

    int offset = 0;

    for (const auto &bb : func) {
        for (const auto &inst : bb) {
            if (const llvm::AllocaInst *alloca = llvm::dyn_cast<llvm::AllocaInst>(&inst)) {
                llvm::Type *ty = alloca->getAllocatedType();

                // 使用 DataLayout 获得准确大小
                uint64_t size = DL.getTypeAllocSize(ty);

                // 4 字节对齐
                if (size % 4 != 0)
                    size = ((size + 3) / 4) * 4;

                // 存储偏移，建议用 Value* 作为 key
                Variable var{ alloca->getName().str() };
                frame.localVarOffsets[var] = offset;

                offset += size;
            }
        }
    }
    offset += static_cast<int>(frame.calleeSavedRegs.size()) * 4;
    frame.frameSize = offset;
    return offset;
}
```

在进入函数块时使用 insertPrologue 函数设置栈偏移 SP 的值，并且将所有的 calleeSavedRegs 中的寄存器入栈; 在退出函数时需要使用 insertEpilogue 函数将保存的所有寄存器从栈中加载回去，然后再回复栈指针。这两个函数的代码如下。

```
void insertPrologue(const StackFrame &frame, std::vector<MInstruction> &mir) {
    // sp = sp - frameSize
    mir.emplace_back(MOpcode::ADD_INT, SP.name, SP.name, std::to_string(-frame.frameSize));

    int offset = frame.frameSize - 4;
    for (const auto &reg : frame.calleeSavedRegs) {
        if(reg.name[0] == 'f') {
            // 如果是浮点寄存器，使用 STORE_FLOAT
            mir.emplace_back(MOpcode::STORE_FLOAT, reg.name, std::to_string(offset), SP.name);
        } else {
            mir.emplace_back(MOpcode::STORE_INT, reg.name, std::to_string(offset), SP.name);
        }

        offset -= 4;
    }
}

void insertEpilogue(const StackFrame &frame, std::vector<MInstruction> &mir) {
    int offset = frame.frameSize - 4;
    for (const auto &reg : frame.calleeSavedRegs) {
        // LOAD_INT reg <- [sp + offset]
        if(reg.name[0] == 'f') {
            mir.emplace_back(MOpcode::LOAD_FLOAT, reg.name, std::to_string(offset), SP.name);
        } else {
            mir.emplace_back(MOpcode::LOAD_INT, reg.name, std::to_string(offset), SP.name);
        }
        offset -= 4;
    }

    // sp = sp + frameSize
    mir.emplace_back(MOpcode::ADD_INT, SP.name, SP.name, std::to_string(frame.frameSize));
}
```

3. 最外层 llvm::Module 的处理

在处理最外层 llvm::Module 的时候最重要的就是将所有的全局数据存储到 MIRProgram 的 globalInstructions 中去。globalInstructions 的每一项都是使用 Minstruction 结构存储的，其中 Mopcode 选择 VAR_GLOBAL，在第一个参数 dst 中存入全局变量名称，src1 中存变量类型，src2 中存变量总大小，如果是数组则 extraData 字段存入数组的所有初始值，关键代码段如标黄所示，完整代码如下。

```
MIRProgram generateMIRFromLLVM(const llvm::Module &module) {
    MIRProgram program;

    //collectFloatConstants(module, program.staticFloats);

    // 1. 处理全局变量
    for (const auto &globalVar : module.globals()) {
        std::string name = globalVar.getName().str();
        std::string sizeStr = "4"; // 默认字节数
```

```
        std::vector<ExtraDatum> initialValues;
        std::string typeHint = "int"; // 默认为整数类型
        bool isFloat = false;
        if (globalVar.hasInitializer()) {
            const llvm::Constant *init = globalVar.getInitializer();
            llvm::errs() << "Global var: " << name << " has initializer: ";
            init->print(llvm::errs());
            llvm::errs() << "\n";
            llvm::errs() << "Initializer type: " << *init->getType() << "\n";

            // 整数常量
            if (auto *CI = llvm::dyn_cast<llvm::ConstantInt>(init)) {
                llvm::SmallString<16> strBuf;
                CI->getValue().toString(strBuf, 10, true, false);
                initialValues.push_back({std::string(strBuf.begin(), strBuf.end()), isFloat});
                typeHint = "int";
                sizeStr = "4";
            }

            else if (auto *CFP = llvm::dyn_cast<llvm::ConstantFP>(init)) {
                llvm::APFloat apf = CFP->getValueAPF();
                std::string strVal;
                isFloat = CFP->getType()->isFloatTy();  // ←判断是否是 float 类型

                llvm::SmallVector<char, 32> strBuf;

                if (isFloat) {
                    // 使用 float 精度（IEEE single）转换为字符串
                    llvm::APFloat apf_f(apf); // 拷贝
                    bool losesInfo;
                        apf_f.convert(llvm::APFloat::IEEEsingle(), llvm::APFloat::rmNearestTiesToEven,
&losesInfo);
                    apf_f.toString(strBuf, 6);  // 保留 6 位小数，够 float 精度
                    typeHint = "float";
                    sizeStr = "4";
                } else {
                    // double 类型，直接保留精度输出
                    apf.toString(strBuf, 17);  // double 精度最大是 17 位十进制有效数字
                    typeHint = "double";
                    sizeStr = "8";
                }

                strVal = std::string(strBuf.begin(), strBuf.end());

                initialValues.push_back({strVal, isFloat});
            }


            // 整数数组（如 .word 0,1,2,...）
            else if (auto *CA = llvm::dyn_cast<llvm::ConstantArray>(init)) {
                bool isFloatArray = false;
                for (unsigned i = 0; i < CA->getNumOperands(); ++i) {
```

```
                    const llvm::Constant *elem = CA->getOperand(i);

                    if (auto *elemInt = llvm::dyn_cast<llvm::ConstantInt>(elem)) {
                        llvm::SmallString<16> strBuf;
                        elemInt->getValue().toString(strBuf, 10, true, false);
                        initialValues.push_back({std::string(strBuf.begin(), strBuf.end()), isFloat});
                        typeHint = "int_array";
                    } else if (auto *elemFloat = llvm::dyn_cast<llvm::ConstantFP>(elem)) {
                        isFloat = true;
                        llvm::SmallVector<char, 16> strBuf;
                        elemFloat->getValueAPF().toString(strBuf, 6);
                        initialValues.push_back({std::string(strBuf.begin(), strBuf.end()), isFloat});
                        typeHint = "float_array";
                        isFloatArray = true;
                    }
                }
                sizeStr = std::to_string((isFloatArray ? 4 : 4) * initialValues.size());
            }
            else if (auto *CDA = llvm::dyn_cast<llvm::ConstantDataArray>(init)) {
                if (CDA->isString()) {
                    // 是字符串常量，比如 "hello"
                    std::string str = CDA->getAsCString().str();
                    for (char ch : str) {
                        initialValues.push_back({std::to_string(static_cast<int>(ch)), isFloat});
                    }
                    typeHint = "char_array";
                    sizeStr = std::to_string(str.size());
                } else if (CDA->getElementType()->isIntegerTy(32)) {
                    for (unsigned i = 0; i < CDA->getNumElements(); ++i) {
                                initialValues.push_back({std::to_string(CDA->getElementAsInteger(i)),
isFloat});
                    }
                    typeHint = "int_array";
                    sizeStr = std::to_string(4 * CDA->getNumElements());
                } else if (CDA->getElementType()->isFloatTy()) {
                    isFloat = true;
                    for (unsigned i = 0; i < CDA->getNumElements(); ++i) {
                        float val = CDA->getElementAsFloat(i);
                        initialValues.push_back({std::to_string(val), isFloat});
                    }
                    typeHint = "float_array";
                    sizeStr = std::to_string(4 * CDA->getNumElements());
                }
            }

        } else {
            if(globalVar.getType()->isFloatingPointTy()) {
                isFloat = true; // 如果没有初始值，默认设置为浮点数
            }
            initialValues.push_back({"0", isFloat}); // 如果没有初始值，默认设置为 0
        }
```

```
        // 将 typeHint 填入 src1，用于标识类型（例如 "int", "float"）
        program.globalInstructions.emplace_back(
            MOpcode::VAR_GLOBAL,
            name,
            typeHint,     // src1: 类型标识
            sizeStr,      // src2: 大小
            initialValues,
            isFloat       // 是否为浮点数
        );
    }
```

在此之后，还需要处理函数的声明和定义，函数声明需要将 Mopcode 字段设置为 EXTERN_FUNC，然后将这句 Minstruction 存入 globalInstructions 中；如果是函数定义动画通过 processFunction 函数继续处理。

```
    // 2. 处理函数声明和定义
    for (const auto &func : module) {
        if (func.isDeclaration()) {
            // 外部函数声明
            std::string name = func.getName().str();
            std::string retType = typeToString(func.getReturnType());
            std::string argTypes = "";
            for (auto &arg : func.args()) {
                if (!argTypes.empty()) argTypes += ",";
                argTypes += typeToString(arg.getType());
            }
            program.globalInstructions.emplace_back(
                MOpcode::EXTERN_FUNC,
                name,
                retType,
                argTypes
            );
        } else {
            // 函数定义
            MIRFunction mirFunc = processFunction(func,program);
            program.functions.push_back(std::move(mirFunc));
        }
    }

    return program;
}
```

4. 函数的处理

函数块的处理函数有之前处理 llvm::Module 的函数调用，处理这个 module 中的每一个 llvm::Function。其首先需要构造一个函数栈帧，使用的事实前文中的 calculateLocalVarSize 函数，这个函数返回函数声明的局部变量的大小。getCalleeSavedRegisters 这个函数如果是 main 函数则只保存 ra 寄存器，如果不是 main 函数则保存所有应该由被调用函数保存的寄

存器。构造栈帧的代码如下。

```
MIRFunction processFunction(const llvm::Function &func,MIRProgram &program) {
    MIRFunction mirFunc;
    mirFunc.name = func.getName().str();
    mirFunc.returnFloat = func.getReturnType()->isFloatingPointTy(); // 检查返回类型是否为浮点数
    std::unordered_map<const llvm::Value*, std::string> valueToRegister;
    allocateRegisters(func, valueToRegister);

    // 1. 构造栈帧
    StackFrame frame;

    frame.calleeSavedRegs = getCalleeSavedRegisters(func);
    int localVarSize = calculateLocalVarSize(func,frame);
    frame.frameSize = localVarSize;

    frame.debugPrint();

    int argIndex = 0;
    for (const llvm::Argument &arg : func.args()) {
        std::string argReg;
        if(arg.getType()->isFloatingPointTy()) {
            argReg = "fa" + std::to_string(argIndex++);
        } else {
            argReg = "a" + std::to_string(argIndex++);
        }
        valueToRegister[&arg] = argReg;
    }
```

在构造完栈帧之后就可以创建函数入口块了，这个入口块主要存放 insertPrologue 函数生成的设置 sp 寄存器和将函数参数保存到栈中，之后就可以处理函数体的每一个基本块，其命名方式按照函数名称+基本块原有的名称构成，之后就可以遍历这个块的每条 IR 指令，调用 processInstruction 函数进行处理。在处理完成后就可以创建函数的结束块，如果这个函数没有 return 语句则调用 insertEpilogue 函数，如果有 return 语句那么久放到 return 语句处理的位置来操作。代码如下。

```
    // 2. 创建函数入口块，并插入 Prologue
    MIRBlock entryBlock;
    entryBlock.label = mirFunc.name + "_enterpoint";
    insertPrologue(frame, entryBlock.instructions);
    mirFunc.blocks.push_back(std::move(entryBlock));

    static int bbCounter = 0;
    // 3. 处理函数体的每个基本块
    for (const auto &bb : func) {
        MIRBlock mirBlock;
        std::string bbName = bb.getName().str();
        bbName = mirFunc.name + "_" + bbName;
        mirBlock.label = bbName;
```

```
        for (const auto &inst : bb) {
            processInstruction(inst, mirBlock,
                valueToRegister, frame,program);
        }
        mirFunc.blocks.push_back(std::move(mirBlock));
    }

    // 4. 创建函数结束块，插入 Epilogue 和 Ret
    bool hasReturn = false;
    // 遍历基本块和指令，检查是否存在 ReturnInst
    for (const auto &bb : func) {
        for (const auto &inst : bb) {
            if (llvm::isa<llvm::ReturnInst>(inst)) {
                hasReturn = true;
                break;
            }
        }
        if (hasReturn) break;
    }

    if (!hasReturn) {
        MIRBlock exitBlock;
        exitBlock.label = mirFunc.name + "_exit";
        insertEpilogue(frame, exitBlock.instructions);
        exitBlock.instructions.emplace_back(MOpcode::RET);
        mirFunc.blocks.push_back(std::move(exitBlock));
    }

    return mirFunc;
}
```

5. IR 指令处理

IR 指令主要分为 8 类，使用以下函数进行分类讨论。

```
void processInstruction(const llvm::Instruction &inst,
    MIRBlock &mirBlock,
    std::unordered_map<const llvm::Value*, std::string> &valueToRegister,
    const StackFrame &frame,MIRProgram &program){
        if (const auto *bin = llvm::dyn_cast<llvm::BinaryOperator>(&inst)) {
            handleBinaryOp(bin, mirBlock.instructions, valueToRegister, frame,program);
        } else if (const auto *ret = llvm::dyn_cast<llvm::ReturnInst>(&inst)) {
            handleReturnInst(ret, mirBlock, valueToRegister, frame,program);
        }
        else if (const auto *call = llvm::dyn_cast<llvm::CallInst>(&inst)) {
            handleCallInst(call, mirBlock.instructions, valueToRegister, frame,program);
        }
        else if (const auto *br = llvm::dyn_cast<llvm::BranchInst>(&inst)) {
                                                    handleBranchInst(br,        *inst.getFunction(),
mirBlock.instructions,frame,valueToRegister,program);
        }
        else if (const auto *icmp = llvm::dyn_cast<llvm::ICmpInst>(&inst)) {
            handleICmpInst(icmp, mirBlock.instructions, valueToRegister, frame,program);
```

```
        }
        else if (const auto *load = llvm::dyn_cast<llvm::LoadInst>(&inst)) {
            handleLoadInst(load, mirBlock.instructions, valueToRegister, frame,program);
        }
        else if (const auto *store = llvm::dyn_cast<llvm::StoreInst>(&inst)) {
            handleStoreInst(store, mirBlock.instructions, valueToRegister, frame,program);
        }
        else if (const auto *gep = llvm::dyn_cast<llvm::GetElementPtrInst>(&inst)) {
            handleGetElementPtrInst(gep, mirBlock.instructions, valueToRegister, frame,program);
        }
}
```

除了以上函数之外，还有一个关键的寄存器分配函数 getOperandStr，代码如下。其作用是对每一个 IR 指令的操作数进行处理：如果是常量整数则分配一个寄存器然后使用 LI 指令进行加载;如果是常量浮点数则需要将其命名并存入 MIRProgram 的 staticFloats 字段中，之后在生成 riscv 汇编的时候要生成进 data 段，然后使用 LA 加载出浮点常量的地址，再使用 LOAD_FLOAT 指令将这个常量 load 出来，并返回 load 的目标寄存器，寄存器中即为浮点数的值；如果操作数在寄存器映射表 valueToRegister 中可以查到，则返回寄存器映射表中的寄存器；如果常量是栈偏移则返回形如 4（SP）这样的字符串。具体代码如下。

```
std::string getOperandStr(
    const llvm::Value *val,
    std::unordered_map<const llvm::Value*, std::string> &valueToRegister,
    const StackFrame &frame,
    std::vector<MInstruction> &mir,
    MIRProgram &program
) {

    // 1. 如果是常量整数
    if (const auto *constInt = llvm::dyn_cast<llvm::ConstantInt>(val)) {
        int vregCounter = valueToRegister.size();
        std::string tmpReg = "v" + std::to_string(vregCounter + TEMPReg.size());
        TEMPReg.push(tmpReg); // 将临时寄存器加入栈
        mir.emplace_back(MOpcode::LI, tmpReg, std::to_string(constInt->getSExtValue())); // 生成寄存器
赋值指令
        valueToRegister[val] = tmpReg; // 记录寄存器映射
        return tmpReg;
    }

    // 2. 如果是常量浮点数
    if (const auto *constFP = llvm::dyn_cast<llvm::ConstantFP>(val)) {
        float valDouble = constFP->getValueAPF().convertToFloat();

        // 1. 从浮点常量表里找对应符号
        std::string symbolName;
        // 先查找是否已经存在
        for (const auto &f : program.staticFloats) {
            if (f.value == valDouble) {
```

```
            symbolName = f.symbol;
            break;
        }
    }
    if (symbolName.empty()) {
        // 没找到才添加
        symbolName = "float_const" + std::to_string(program.staticFloats.size());
        program.staticFloats.push_back({symbolName, valDouble});
    }


    int vregCounter = valueToRegister.size();
    std::string tmpReg = "v" + std::to_string(vregCounter + TEMPReg.size());
    TEMPReg.push(tmpReg); // 将临时寄存器加入栈

    mir.emplace_back(MOpcode::LA, tmpReg, symbolName); // 生成加载浮点常量的指令

    std::string floatReg = "v" + std::to_string(vregCounter + TEMPReg.size() + 1);
    TEMPReg.push(floatReg); // 将临时寄存器加入栈

    mir.emplace_back(MOpcode::LOAD_FLOAT, floatReg, "0", tmpReg,
        std::vector<ExtraDatum>(), // 没有初始值
        true // 标记为浮点数
    ); // 生成加载浮点数的指令

    valueToRegister[val] = floatReg;

    return floatReg;
}

if (const auto *globalVar = llvm::dyn_cast<llvm::GlobalVariable>(val)) {
    // 判断是否已处理过
    int vregCounter = valueToRegister.size();
    std::string addrReg = "v" + std::to_string(vregCounter + TEMPReg.size());
    TEMPReg.push(addrReg); // 将临时寄存器加入栈

    llvm::Type *ty = globalVar->getValueType();
    bool isFloat = ty->isFloatTy() || ty->isDoubleTy();

    // 先加载地址
    mir.emplace_back(MOpcode::LA, addrReg, globalVar->getName().str());
    valueToRegister[globalVar] = addrReg;

    return valueToRegister.at(globalVar);
}

// 3. 寄存器映射
auto it = valueToRegister.find(val);
if (it != valueToRegister.end()) {
    return it->second;
}
```

```
    // 4. 栈偏移
    if (val->hasName()) {
        Variable var{val->getName().str()};
        auto offsetIt = frame.localVarOffsets.find(var);
        if (offsetIt != frame.localVarOffsets.end()) {
            // 只返回栈偏移地址字符串，比如 "12(sp)"
            return std::to_string(offsetIt->second) + "(" + SP.name + ")";
        }
    }
    return "tmp" + std::to_string(reinterpret_cast<uintptr_t>(val));
}
```

还有一个关键是 valueToRegister 这个变量名-寄存器映射表，它具有如下结构：std::unordered_map<const llvm::Value*, std::string> &valueToRegister，其作用是将常量名与寄存器名映射起来，可以存入、删除和读取。

分指令处理函数时还会使用下面这个辅助函数，用来判断操作数是否是存在栈里的，如果是则返回 true，并且将 baseReg 和 offset 设置为 sp 和栈偏移。

```
bool parseStackOffset(const std::string &addrStr, std::string &offset, std::string &baseReg) {
    static const std::regex pattern(R"((\d+)\(\(\w+)\))");
    std::smatch match;
    if (std::regex_match(addrStr, match, pattern)) {
        offset = match[1].str();
        baseReg = match[2].str();
        return true;
    }
    return false;
}
```

接下来分函数介绍如何处理这 8 种指令：

① 运算指令 BinaryOp

函数 handleBinaryOp 的主要功能是将 LLVMIR 中的二元运算转换为中间表示形式，包括加减乘除四种运算。

函数开始时，先从传入的 LLVM 二元操作指令 bin 中提取出左右操作数 lhs 和 rhs，以及结果类型 ty。通过检查类型的 LLVM 类型系统，判断当前操作是整数、字符还是浮点数（float），并据此决定应该使用哪种对应的 MIR 操作码，具体代码如下。

```
void handleBinaryOp(const llvm::BinaryOperator *bin,
    std::vector<MInstruction> &mir,
    std::unordered_map<const llvm::Value*, std::string> &valueToRegister,
    const StackFrame &frame,MIRProgram &program) {

    using namespace llvm;

    const Value *lhs = bin->getOperand(0);
```

```
    const Value *rhs = bin->getOperand(1);
    const Type *ty = bin->getType();

    MOpcode opcode;

    // 判断类型是 int / float / char（这里只区分 int32, float, int8）
    bool isInt = ty->isIntegerTy(32);
    bool isChar = ty->isIntegerTy(8);
    bool isFloat = ty->isFloatTy();

    switch (bin->getOpcode()) {
        case Instruction::Add:
            opcode = isFloat ? MOpcode::ADD_FLOAT :
                    isInt   ? MOpcode::ADD_INT :
                              MOpcode::ADD_INT;
            break;
        case Instruction::Sub:
            opcode = isFloat ? MOpcode::SUB_FLOAT :
                    isInt   ? MOpcode::SUB_INT :
                              MOpcode::SUB_INT;
            break;
        case Instruction::Mul:
            opcode = isFloat ? MOpcode::MUL_FLOAT :
                    isInt   ? MOpcode::MUL_INT :
                              MOpcode::MUL_INT;
            break;
        case Instruction::SDiv:
        case Instruction::UDiv:
            opcode = isFloat ? MOpcode::DIV_FLOAT :
                    isInt   ? MOpcode::DIV_INT :
                              MOpcode::DIV_INT;
            break;
        case Instruction::SRem:
            opcode =  MOpcode::SREM ;
            break;
        default:
            std::cerr << "[MIR] 未支持的 BinaryOperator 类型: " << bin->getOpcodeName() << "\n";
            return;
    }
```

接下来通过 getOperandStr 函数获取操作数和结果的字符串表示，这些表示可能是寄存器名、栈偏移表达式或符号地址。如果发现左右操作数之一是全局变量，则生成相应的加载指令，把变量从内存中读入一个临时虚拟寄存器，并更新 src1 或 src2 的实际使用寄存器。

```
    std::string dst  = getOperandStr(bin, valueToRegister, frame, mir,program);
    std::string src1 = getOperandStr(lhs, valueToRegister, frame, mir,program);
    std::string src2 = getOperandStr(rhs, valueToRegister, frame, mir,program);

    if (llvm::isa<llvm::GlobalVariable>(lhs)) {
```

```
        std::cout << "Global variable detected in lhs: " << lhs->getName().str() << std::endl;
        std::string loadedLhs = "v" + std::to_string(valueToRegister.size() + TEMPReg.size());
        TEMPReg.push(loadedLhs); // 将临时寄存器加入栈
        if(lhs->getType()->getPointerElementType()->isIntegerTy(32)) {
            mir.emplace_back(MOpcode::LOAD_INT, loadedLhs, "0", src1);
        } else if(lhs->getType()->getPointerElementType()->isFloatTy()) {
            mir.emplace_back(MOpcode::LOAD_FLOAT, loadedLhs, "0", src1,
                std::vector<ExtraDatum>(),
                true);
        } else if(lhs->getType()->getPointerElementType()->isIntegerTy(8)) {
            mir.emplace_back(MOpcode::LOAD_CHAR, loadedLhs, "0", src1);
        } else {
            mir.emplace_back(MOpcode::LOAD_INT, loadedLhs, "0", src1);
        }
        //valueToRegister[lhs] = loadedLhs; // 更新映射
        valueToRegister.erase(lhs); // 删除原有映射
        src1 = loadedLhs;
    }

    if (llvm::isa<llvm::GlobalVariable>(rhs)) {
        std::string loadedRhs = "v" + std::to_string(valueToRegister.size() + TEMPReg.size());
        TEMPReg.push(loadedRhs);
        if(rhs->getType()->getPointerElementType()->isIntegerTy(32)) {
            mir.emplace_back(MOpcode::LOAD_INT, loadedRhs, "0", src2);
        } else if(rhs->getType()->getPointerElementType()->isFloatTy()) {
            mir.emplace_back(MOpcode::LOAD_FLOAT, loadedRhs, "0", src2,
                std::vector<ExtraDatum>(),
                true);
        } else if(rhs->getType()->getPointerElementType()->isIntegerTy(8)) {
            mir.emplace_back(MOpcode::LOAD_CHAR, loadedRhs, "0", src2);
        } else {
            mir.emplace_back(MOpcode::LOAD_INT, loadedRhs, "0", src2);
        }
        //valueToRegister[rhs] = loadedRhs; // 更新映射
        valueToRegister.erase(rhs); // 删除原有映射
        src2 = loadedRhs;
    }
```

之后通过 parseStackOffset 检查 src1 和 src2 是否表示的是栈上的变量。如果是，就生成一条加载指令从内存中读取到临时寄存器，再将 src1 或 src2 更新为这个新寄存器。

```
    llvm::Type* type1 = lhs->getType()->isPointerTy() ? lhs->getType()->getPointerElementType() : lhs->getType();
    llvm::Type* type2 = rhs->getType()->isPointerTy() ? rhs->getType()->getPointerElementType() : rhs->getType();

    std::string typeStr1;
    llvm::raw_string_ostream rso1(typeStr1);
    type1->print(rso1);
    rso1.flush();
```

```
std::string typeStr2;
llvm::raw_string_ostream rso2(typeStr2);
type2->print(rso2);
rso2.flush();

std::cout << "type of src1: " << typeStr1 << std::endl;
std::cout << "type of src2: " << typeStr2 << std::endl;

std::string offset, baseReg;

// 处理 src1
if (parseStackOffset(src1, offset, baseReg)) {
    int vregCounter = valueToRegister.size();
    std::string tmpReg1 = "v" + std::to_string(vregCounter + TEMPReg.size());
    if(lhs->getType()->getPointerElementType()->isIntegerTy(32)) {
        mir.emplace_back(MOpcode::LOAD_INT, tmpReg1, offset, baseReg);
    }else if(lhs->getType()->getPointerElementType()->isFloatTy()) {
        mir.emplace_back(MOpcode::LOAD_FLOAT, tmpReg1, offset, baseReg,
            std::vector<ExtraDatum>(), // 没有初始值
            true);
    } else if(lhs->getType()->getPointerElementType()->isIntegerTy(8)) {
        // 如果是 8 位整型，加载后可能需要扩展
        mir.emplace_back(MOpcode::LOAD_CHAR, tmpReg1, offset, baseReg);
        // TODO: 如果后续操作是 32 位整数加法，可能需要生成扩展指令
    } else {
        // 默认当整数处理
        mir.emplace_back(MOpcode::LOAD_INT, tmpReg1, offset, baseReg);
    }

    src1 = tmpReg1;
    //valueToRegister[lhs] = tmpReg1;
}
offset = "0";
baseReg = "0";
// 处理 src2
if (parseStackOffset(src2, offset, baseReg)) {
    int vregCounter = valueToRegister.size();
    std::string tmpReg2 = "v" + std::to_string(vregCounter + TEMPReg.size());
    if(rhs->getType()->getPointerElementType()->isIntegerTy(32)) {
        mir.emplace_back(MOpcode::LOAD_INT, tmpReg2, offset, baseReg);
    }else if(rhs->getType()->getPointerElementType()->isFloatTy()) {
        mir.emplace_back(MOpcode::LOAD_FLOAT, tmpReg2, offset, baseReg,
            std::vector<ExtraDatum>(), // 没有初始值
            true);
    } else if(rhs->getType()->getPointerElementType()->isIntegerTy(8)) {
        mir.emplace_back(MOpcode::LOAD_CHAR, tmpReg2, offset, baseReg);
        // TODO: 如果后续操作是 32 位整数加法，可能需要生成扩展指令
    } else {
        mir.emplace_back(MOpcode::LOAD_INT, tmpReg2, offset, baseReg);
    }
    src2 = tmpReg2;
```

```
        //valueToRegister[rhs] = tmpReg2;
    }
```

最后在运算目标 dst 的处理部分，同样检查它是否为栈地址。如果 dst 是一个内存地址，不能直接作为指令目标寄存器使用，因此需要先创建一个临时寄存器作为结果保存位置，然后生成实际的二元操作指令，再将计算结果通过 STORE_INT 或 STORE_FLOAT 等指令存回内存。这个过程结束后，函数会提前返回。如果目标 dst 是一个合法的寄存器或中间值，直接生成一条对应的 MIR 二元操作指令，并更新 valueToRegister 映射表，记录该指令对应的目标寄存器。

```
    // 处理 dst
    bool dstIsMemory = false;
    std::string dstOffset, dstBaseReg;
    if (parseStackOffset(dst, dstOffset, dstBaseReg)) {
        // dst 是内存地址，不能直接作为运算结果寄存器
        dstIsMemory = true;
        int vregCounter = valueToRegister.size();
        std::string tmpDstReg = "v" + std::to_string(vregCounter + TEMPReg.size());
        TEMPReg.push(tmpDstReg); // 将临时寄存器加入栈

        mir.emplace_back(opcode, tmpDstReg, src1, src2);

        // 将结果存回内存地址 dst
        if (ty->isIntegerTy(32)) {
            mir.emplace_back(MOpcode::STORE_INT, tmpDstReg, dstOffset, dstBaseReg);
        } else if (ty->isFloatTy()) {
            mir.emplace_back(MOpcode::STORE_FLOAT, tmpDstReg, dstOffset, dstBaseReg,
                std::vector<ExtraDatum>(), // 没有初始值
                true);
        } else {
            mir.emplace_back(MOpcode::STORE_INT, tmpDstReg, dstOffset, dstBaseReg);
        }

        // 注册临时寄存器映射（如果后续需要）
        //valueToRegister[bin] = tmpDstReg;

        // 处理结束，直接 return
        return;
    }

    mir.emplace_back(opcode, dst, src1, src2,
        std::vector<ExtraDatum>(), // 没有额外数据
        isFloat // 是否为浮点数
    );

    // 记得更新寄存器映射
    valueToRegister[bin] = dst;
}
```

② 返回指令 ReturnInst

handleReturnInst 函数的作用是将 LLVM 中的 ReturnInst 指令转换为中间表示中的返回指令，最终用于生成汇编代码。在处理时，函数首先获取返回值指针，如果没有返回值，说明函数是 void 类型，此时它会直接插入函数尾部清理代码并生成 RET 指令后返回。若存在返回值，则先通过 getOperandStr 获取对应的 MIR 寄存器或栈偏移地址字符串。

```
void handleReturnInst(const llvm::ReturnInst *retInst, MIRBlock &mirBlock,
                      std::unordered_map<const llvm::Value*, std::string> &valueToRegister,
                      const StackFrame &frame,MIRProgram &program) {
    const llvm::Value *retVal = retInst->getReturnValue();
    // 创建函数结束块，插入 Epilogue 和 Ret
    std::cout << "[MIR] 处理 Return 指令: " << (retVal ? retVal->getName().str() : "void") << "\n";
    if (!retVal) {
        insertEpilogue(frame, mirBlock.instructions);
        mirBlock.instructions.emplace_back(MOpcode::RET);
        return; // 如果没有返回值，直接返回
    }

    std::string src = getOperandStr(retVal, valueToRegister, frame, mirBlock.instructions,program);
```

接着，检查返回值是否是全局变量，如果是，就生成相应的 LOAD 指令将全局值加载到一个新的虚拟寄存器中。加载完成后更新 src，指向新生成的寄存器。

```
    if (llvm::isa<llvm::GlobalVariable>(retVal)) {
        std::string tempReg = "v" + std::to_string(valueToRegister.size() + TEMPReg.size());
        TEMPReg.push(tempReg); // 将临时寄存器加入栈
        if(retVal->getType()->getPointerElementType()->isIntegerTy(32)) {
            mirBlock.instructions.emplace_back(MOpcode::LOAD_INT, tempReg, "0", src);
        } else if(retVal->getType()->getPointerElementType()->isFloatTy()) {
            mirBlock.instructions.emplace_back(MOpcode::LOAD_FLOAT, tempReg, "0", src,
                std::vector<ExtraDatum>(), // 没有初始值
                true);
        } else if(retVal->getType()->getPointerElementType()->isIntegerTy(8)) {
            mirBlock.instructions.emplace_back(MOpcode::LOAD_CHAR, tempReg, "0", src);
        } else {
            mirBlock.instructions.emplace_back(MOpcode::LOAD_INT, tempReg, "0", src);
        }
        //valueToRegister[retVal] = tempReg; // 更新映射
        valueToRegister.erase(retVal); // 删除原有映射
        src = tempReg;
    }
}
```

随后判断 src 是否是栈地址，如果是，会将这个地址的值加载到一个新的临时寄存器中，并更新映射表以及 src 的值，以保证参与返回的值是在寄存器中的。

```
    llvm::Type *retType = retVal->getType();
    std::string dstOffset, dstBaseReg;
    if (parseStackOffset(src, dstOffset, dstBaseReg)) {
        // 如果 src 是栈偏移地址格式，先加载到临时寄存器
        int vregCounter = valueToRegister.size();
        std::string tmpReg = "v" + std::to_string(vregCounter+ TEMPReg.size());
        TEMPReg.push(tmpReg); // 将临时寄存器加入栈

        if (retType->isIntegerTy(32)) {
            mirBlock.instructions.emplace_back(MOpcode::LOAD_INT, tmpReg, dstOffset, dstBaseReg);
        } else if (retType->isFloatTy()) {
            mirBlock.instructions.emplace_back(MOpcode::LOAD_FLOAT, tmpReg, dstOffset, dstBaseReg,
                std::vector<ExtraDatum>(), // 没有初始值
                true);
        } else if (retType->isIntegerTy(8)) {
            mirBlock.instructions.emplace_back(MOpcode::LOAD_CHAR, tmpReg, dstOffset, dstBaseReg);
        } else {
            mirBlock.instructions.emplace_back(MOpcode::LOAD_INT, tmpReg, dstOffset, dstBaseReg);
        }
        valueToRegister[retVal] = tmpReg; // 更新寄存器映射
        src = tmpReg; // 更新 src 为临时寄存器
    }
```

当返回值已经就绪，则根据返回值的类型，决定使用哪种寄存器传回调用者：32 位整数和字符会被写入到 a0，浮点数会被写入到 fa0。MOV 指令的类型也会根据数据类型选择对应的 MOV_INT、MOV_CHAR 或 MOV_FLOAT。最后，函数插入函数清理代码，再插入 RET 指令，表示函数结束并将控制权交还给调用者。

```
    if (retType->isIntegerTy(32)) {
        mirBlock.instructions.emplace_back(MOpcode::MOV_INT, "a0", src);
    } else if (retType->isFloatTy()) {
        mirBlock.instructions.emplace_back(MOpcode::MOV_FLOAT, "fa0", src,"",
            std::vector<ExtraDatum>(), // 没有初始值
            true); // 标记为浮点数
    } else if (retType->isIntegerTy(8)) {
        // char 类型
        mirBlock.instructions.emplace_back(MOpcode::MOV_CHAR, "a0", src);
    } else {
        // 默认当整型处理
        mirBlock.instructions.emplace_back(MOpcode::MOV_INT, "a0", src);
    }
    insertEpilogue(frame, mirBlock.instructions);
    mirBlock.instructions.emplace_back(MOpcode::RET);
}
```

③ 存储指令 StoreInst

handleStoreInst 函数的作用是将 LLVM 中的 StoreInst 指令转换为对应的 MIR 存储指令。首先，它提取出要被存储的值以及存储的目标地址，然后调用 getOperandStr 获取待存值的操作数字符串。之后检测这个字符串是否是栈偏移格式。如果是，则根据值的类型生成一条对应的 LOAD 指令，将值从栈中加载到一个新的虚拟寄存器中，然后将这个寄存器作为新的待存值。如果原始的值字符串本身就是一个寄存器名，则将其作为基址寄存器，偏移设为 0，表示值已经在寄存器中，无需加载。

```
void handleStoreInst (
    const llvm::StoreInst *store,
    std::vector<MInstruction> &mir,
    std::unordered_map<const llvm::Value*, std::string> &valueToRegister,
    const StackFrame &frame,MIRProgram &program)
{
    const llvm::Value *val = store->getValueOperand();
    const llvm::Value *ptr = store->getPointerOperand();

    std::string valStr = getOperandStr(val, valueToRegister, frame, mir,program);

    std::string offset_val = "0";
    std::string baseReg_val = "0";
    const llvm::Type *valType = val->getType();
    if(parseStackOffset(valStr, offset_val, baseReg_val)) {
        if(valType->isIntegerTy(32)){
            // 如果是 32 位整数
            valStr = "v" + std::to_string(valueToRegister.size()+ TEMPReg.size());
            mir.emplace_back(MOpcode::LOAD_INT, valStr, offset_val, baseReg_val);
        } else if(valType->isFloatTy()) {
            // 如果是浮点数
            valStr = "v" + std::to_string(valueToRegister.size()+ TEMPReg.size());
            mir.emplace_back(MOpcode::LOAD_FLOAT, valStr, offset_val, baseReg_val,
                std::vector<ExtraDatum>(), // 没有初始值
                true); // 标记为浮点数
        } else if(valType->isIntegerTy(8)) {
            // 如果是 8 位整数（char）
            valStr = "v" + std::to_string(valueToRegister.size()+ TEMPReg.size());
            mir.emplace_back(MOpcode::LOAD_CHAR, valStr, offset_val, baseReg_val);
        } else {
            // 默认当整型处理
            valStr = "v" + std::to_string(valueToRegister.size()+ TEMPReg.size());
            mir.emplace_back(MOpcode::LOAD_INT, valStr, offset_val, baseReg_val);
        }
        TEMPReg.push(valStr); // 将临时寄存器加入栈
        //valueToRegister[val] = valStr; // 更新寄存器映射

    } else {
        // valStr 就是寄存器名
        offset_val = "0"; // 偏移为 0
        baseReg_val = valStr; // 寄存器名作为基址
    }
```

在此之后，通过同样的 getOperandStr 获取存储目标地址，并判断其是否为栈偏移格式。如果是，便拆解出偏移量和基址寄存器；如果不是，则视其为纯寄存器地址，偏移默认为 0。

```
    std::string addrStr = getOperandStr(ptr, valueToRegister, frame, mir,program); // 可能是寄存器或栈偏移

    std::string offset = "0";
    std::string baseReg = addrStr;

    // 解析栈偏移格式
    if (parseStackOffset(addrStr, offset, baseReg)) {
        // 解析成功，offset 和 baseReg 已拆分
    } else {
        // addrStr 就是寄存器名，offset 保持为0
    }
    const llvm::Type *ty = val->getType();
```

完成地址处理后根据待存值的类型选择正确的存储操作码，并生成对应的 STORE 指令，将值写入目标地址。这些指令也根据类型细分为整数、浮点数和字符三类，以确保写入操作与数据类型匹配。最终，这一系列的转换指令被加入到 MIR 指令流中，表示 LLVMIR 中的 store 指令已经成功转换为平台中间表示形式。

```
    if (ty->isIntegerTy(32)) {
        mir.emplace_back(MOpcode::STORE_INT, valStr, offset, baseReg);
    } else if (ty->isFloatTy()) {
        mir.emplace_back(MOpcode::STORE_FLOAT, valStr, offset, baseReg,
            std::vector<ExtraDatum>(), // 没有初始值
            true); // 标记为浮点数
    } else if (ty->isIntegerTy(8)) {
        mir.emplace_back(MOpcode::STORE_CHAR, valStr, offset, baseReg);
    } else {
        mir.emplace_back(MOpcode::STORE_INT, valStr, offset, baseReg);
    }
    // 更新寄存器映射
    //valueToRegister[store] = addrStr; // 假设 store 的地址是
}
```

④ 分支指令 BranchInst

handleBranchInst 函数的主要功能是处理 LLVM 中的分支指令，将其转换为中间表示 MIR 的跳转指令。对于无条件跳转的情况，函数直接生成跳转到目标基本块的 JMP 指令，并使用函数名和基本块名组合生成唯一的标签名称。对于条件跳转的情况，函数首先获取条件值对应的寄存器，然后根据条件值的真假分别生成跳转到两个不同基本块的标签名称。通

过生成 BNE 指令实现条件跳转逻辑，当条件寄存器不等于零时跳转到 true 分支，否则通过后续的 JMP 指令跳转到 false 分支。如果遇到未知的分支类型，就输出错误信息提示不支持该分支类型。完整代码如下。

```
void handleBranchInst(const llvm::BranchInst *br,
                      const llvm::Function &func,
                      std::vector<MInstruction> &mir,
                      const StackFrame &frame,
                       std::unordered_map<const llvm::Value*, std::string> &valueToRegister,MIRProgram
&program) {
    using namespace llvm;

    if (br->isUnconditional()) {
        // 无条件跳转
        const BasicBlock *target = br->getSuccessor(0);
                      std::string    label    =    br->getFunction()->getName().str()    +    "_"    +
br->getSuccessor(0)->getName().str();
        mir.emplace_back(MOpcode::JMP, label);
    } else if (br->isConditional()) {
        const Value *cond = br->getCondition();
        std::string condReg = getOperandStr(cond, valueToRegister, frame, mir,program);

        const BasicBlock *trueBB = br->getSuccessor(0);
        const BasicBlock *falseBB = br->getSuccessor(1);

        std::string trueLabel = br->getFunction()->getName().str() + "_" + trueBB->getName().str();
        std::string falseLabel = br->getFunction()->getName().str() + "_" + falseBB->getName().str();



        // condReg ≠ 0 跳转 trueLabel，否则跳 falseLabel
        // 可以用 BNE condReg, z, trueLabel
        mir.emplace_back(MOpcode::BNE, trueLabel, condReg, "x0");
        mir.emplace_back(MOpcode::JMP, falseLabel);

    } else {
        std::cerr << "[Error] handleBranchInst: 未知分支类型！\n";
    }
}
```

⑤ 比较指令 IcmpInst

handleICmpInst 函数处理 LLVM 中的整数比较指令和浮点数比较指令，将其转换为中间表示 MIR 的比较指令。首先获取比较指令的两个操作数，并分别处理左操作数和右操作数，检查它们是否为全局变量或栈偏移地址。如果是全局变量，函数会生成对应的加载指令，将值加载到临时寄存器；如果是栈偏移地址，则解析偏移量并生成相应的加载指令，确保操作数最终都存储在寄存器中。

```
void handleICmpInst(const llvm::ICmpInst *icmp,
                    std::vector<MInstruction> &mir,
```

```
                std::unordered_map<const llvm::Value*, std::string> &valueToRegister,
                const StackFrame &frame,MIRProgram &program) {
using namespace llvm;

const Value *lhs = icmp->getOperand(0);
const Value *rhs = icmp->getOperand(1);

std::string lhsReg = getOperandStr(lhs, valueToRegister, frame, mir,program);

// 如果是全局变量，可能需要加载到寄存器
if (const GlobalVariable *gv = llvm::dyn_cast<GlobalVariable>(lhs)) {
    std::string tempReg = "v" + std::to_string(valueToRegister.size() + TEMPReg.size());
    TEMPReg.push(tempReg); // 将临时寄存器加入栈
    if (gv->getType()->getPointerElementType()->isIntegerTy (32)) {
        mir.emplace_back(MOpcode::LOAD_INT, tempReg, "0", lhsReg);
    } else if (gv->getType()->getPointerElementType()->isFloatTy()) {
        mir.emplace_back(MOpcode::LOAD_FLOAT, tempReg, "0", lhsReg,
            std::vector<ExtraDatum>(), // 没有初始值
            true); // 标记为浮点数
    } else if (gv->getType()->getPointerElementType()->isIntegerTy(8)) {
        mir.emplace_back(MOpcode::LOAD_CHAR, tempReg, "0", lhsReg);
    } else {
        mir.emplace_back(MOpcode::LOAD_INT, tempReg, "0", lhsReg);
    }
    valueToRegister.erase(lhs);
    lhsReg = tempReg; // 更新为临时寄存器
    //valueToRegister[lhs] = tempReg; // 更新寄存器映射
}
// 如果是栈偏移地址格式，处理成寄存器
std::string offset, baseReg;
if (parseStackOffset(lhsReg, offset, baseReg)) {
    int vregCounter = valueToRegister.size();
    std::string tmpReg = "v" + std::to_string(vregCounter + TEMPReg.size());
    TEMPReg.push(tmpReg); // 将临时寄存器加入栈
    if (lhs->getType()->getPointerElementType()->isIntegerTy(32)) {
        mir.emplace_back(MOpcode::LOAD_INT, tmpReg, offset, baseReg);
    } else if (lhs->getType()->getPointerElementType()->isFloatTy()) {
        mir.emplace_back(MOpcode::LOAD_FLOAT, tmpReg, offset, baseReg,
            std::vector<ExtraDatum>(), // 没有初始值
            true); // 标记为浮点数
    } else if (lhs->getType()->getPointerElementType()->isIntegerTy(8)) {
        mir.emplace_back(MOpcode::LOAD_CHAR, tmpReg, offset, baseReg);
    } else {
        mir.emplace_back(MOpcode::LOAD_INT, tmpReg, offset, baseReg);
    }
    lhsReg = tmpReg; // 更新为临时寄存器
    //valueToRegister[lhs] = tmpReg; // 更新寄存器映射
}

std::string rhsReg = getOperandStr(rhs, valueToRegister, frame, mir,program);
// 如果是全局变量，可能需要加载到寄存器
```

```
    if (const GlobalVariable *gv = llvm::dyn_cast<GlobalVariable>(rhs)) {
        std::string tempReg = "v" + std::to_string(valueToRegister.size() + TEMPReg.size());
        TEMPReg.push(tempReg); // 将临时寄存器加入栈
        if (gv->getType()->getPointerElementType()->isIntegerTy(32)) {
            mir.emplace_back(MOpcode::LOAD_INT, tempReg, "0", rhsReg);
        } else if (gv->getType()->getPointerElementType()->isFloatTy()) {
            mir.emplace_back(MOpcode::LOAD_FLOAT, tempReg, "0", rhsReg,
                std::vector<ExtraDatum>(), // 没有初始值
                true); // 标记为浮点数
        } else if (gv->getType()->getPointerElementType()->isIntegerTy(8)) {
            mir.emplace_back(MOpcode::LOAD_CHAR, tempReg, "0", rhsReg);
        } else {
            mir.emplace_back(MOpcode::LOAD_INT, tempReg, "0", rhsReg);
        }
        valueToRegister.erase(rhs);
        rhsReg = tempReg; // 更新为临时寄存器
        //valueToRegister[rhs] = tempReg; // 更新寄存器映射
    }
    // 如果是栈偏移地址格式，处理成寄存器
    if (parseStackOffset(rhsReg, offset, baseReg)) {
        int vregCounter = valueToRegister.size();
        std::string tmpReg = "v" + std::to_string(vregCounter + TEMPReg.size());
        TEMPReg.push(tmpReg); // 将临时寄存器加入栈
        if (rhs->getType()->getPointerElementType()->isIntegerTy(32)) {
            mir.emplace_back(MOpcode::LOAD_INT, tmpReg, offset, baseReg);
        } else if (rhs->getType()->getPointerElementType()->isFloatTy()) {
            mir.emplace_back(MOpcode::LOAD_FLOAT, tmpReg, offset, baseReg,
                std::vector<ExtraDatum>(), // 没有初始值
                true); // 标记为浮点数
        } else if (rhs->getType()->getPointerElementType()->isIntegerTy(8)) {
            mir.emplace_back(MOpcode::LOAD_CHAR, tmpReg, offset, baseReg);
        } else {
            mir.emplace_back(MOpcode::LOAD_INT, tmpReg, offset, baseReg);
        }
        rhsReg = tmpReg; // 更新为临时寄存器
        //valueToRegister[rhs] = tmpReg; // 更新寄存器映射
    }
```

在处理完操作数后，函数分配一个新的目标寄存器用于存储比较结果。接着，函数根据比较指令的类型和具体的 op，选择对应的 MIR 比较操作码。如果是浮点数比较，则使用浮点数比较操作码；如果是整数比较，则使用整数比较操作码。最后，函数生成对应的 MIR 比较指令，并将结果寄存器与比较指令关联起来，存储在 valueToRegister 映射中，以便后续指令可以引用该比较结果。

```
    std::string dstReg = "v" + std::to_string(valueToRegister.size() + TEMPReg.size());
    TEMPReg.push(dstReg);
    MOpcode op;
```

```
    // 判断是否是浮点数比较
    bool isFloat = lhs->getType()->isFloatingPointTy();

    auto pred = icmp->getPredicate();
    if (isFloat) {
        switch (pred) {
            case CmpInst::FCMP_OEQ: op = MOpcode::ICMP_EQ_FLOAT; break;
            case CmpInst::FCMP_ONE: op = MOpcode::ICMP_NE_FLOAT; break;
            case CmpInst::FCMP_OLT: op = MOpcode::ICMP_LT_FLOAT; break;
            case CmpInst::FCMP_OLE: op = MOpcode::ICMP_LE_FLOAT; break;
            case CmpInst::FCMP_OGT: op = MOpcode::ICMP_GT_FLOAT; break;
            case CmpInst::FCMP_OGE: op = MOpcode::ICMP_GE_FLOAT; break;
            default:
                std::cerr << "[Error] 未支持的浮点比较类型：" << pred << "\n";
                return;
        }
    } else {
        switch (pred) {
            case CmpInst::ICMP_EQ: op = MOpcode::ICMP_EQ_INT; break;
            case CmpInst::ICMP_NE: op = MOpcode::ICMP_NE_INT; break;
            case CmpInst::ICMP_SLT: op = MOpcode::ICMP_LT_INT; break;
            case CmpInst::ICMP_SLE: op = MOpcode::ICMP_LE_INT; break;
            case CmpInst::ICMP_SGT: op = MOpcode::ICMP_GT_INT; break;
            case CmpInst::ICMP_SGE: op = MOpcode::ICMP_GE_INT; break;
            default:
                std::cerr << "[Error] 未支持的整数比较类型：" << pred << "\n";
                return;
        }
    }

    mir.emplace_back(op, dstReg, lhsReg, rhsReg,
                     std::vector<ExtraDatum>(), // 没有初始值
                     isFloat); // 是否是浮点数比较
    valueToRegister[icmp] = dstReg;
}
```

⑥ 函数调用指令 CallInst

handleCallInst 函数的主要功能是处理 LLVM 中的 CallInst 指令，将其转换为中间表示 MIR 的函数调用指令。函数首先获取被调用的函数指针，检查是否为直接函数调用，如果不是则报错并返回。接着，函数处理调用参数，将每个参数转换为寄存器字符串，并检查参数是否为全局变量或栈偏移地址，如果是则生成相应的加载指令到临时寄存器，并更新参数寄存器映射。对于每个参数，判断其类型是否为浮点数，并将参数寄存器包装成 ExtraDatum 结构体以便后续处理。随后，函数检查被调用函数的返回类型，如果是 void 类型则生成无返回值的调用指令；否则分配一个新的寄存器用于存储返回值，并生成带返回值的调用指令。最后，函数将返回值寄存器与 CallInst 指令关联起来，存储在 valueToRegister 映射中，确保

后续指令可以正确引用该返回值。完整代码如下。

```
void handleCallInst(const llvm::CallInst *call,
                    std::vector<MInstruction> &mir,
                    std::unordered_map<const llvm::Value*, std::string> &valueToRegister,
                    const StackFrame &frame,MIRProgram &program) {

    using namespace llvm;

    // 1. 获取调用的函数指针
    const Function *calledFunc = call->getCalledFunction();
    if (!calledFunc) {
        std::cerr << "[Error] handleCallInst: 不是直接调用函数指针的 CallInst，暂不支持！\n";
        return;
    }

    std::string funcName = calledFunc->getName().str();

    // 2. 处理参数，转换成寄存器字符串
    std::vector<ExtraDatum> argRegs;
    for (unsigned i = 0; i < call->getNumArgOperands(); ++i) {
        bool isFloat = false;
        const Value *argVal = call->getArgOperand(i);
        std::string argReg = getOperandStr(argVal, valueToRegister, frame, mir,program);
        // 如果是全局变量，可能需要加载到寄存器
        if (const GlobalVariable *gv = llvm::dyn_cast<GlobalVariable>(argVal)) {
            std::string tempReg = "v" + std::to_string(valueToRegister.size() + TEMPReg.size());
            TEMPReg.push(tempReg); // 将临时寄存器加入栈
            if (gv->getType()->getPointerElementType()->isIntegerTy(32)) {
                mir.emplace_back(MOpcode::LOAD_INT, tempReg, "0", argReg);
            } else if (gv->getType()->getPointerElementType()->isFloatTy()) {
                mir.emplace_back(MOpcode::LOAD_FLOAT, tempReg, "0", argReg,
                    std::vector<ExtraDatum>(), // 没有初始值
                    true); // 标记为浮点数
                isFloat = true; // 标记为浮点数
            } else if (gv->getType()->getPointerElementType()->isIntegerTy(8)) {
                mir.emplace_back(MOpcode::LOAD_CHAR, tempReg, "0", argReg);
            } else {
                mir.emplace_back(MOpcode::LOAD_INT, tempReg, "0", argReg);
            }
            argReg = tempReg; // 更新为临时寄存器
            valueToRegister.erase(argVal);
            //alueToRegister[argVal] = tempReg; // 更新寄存器映射
        }
        //如果是栈偏移地址格式，处理成寄存器
        std::string offset, baseReg;
        if (parseStackOffset(argReg, offset, baseReg)) {
            int vregCounter = valueToRegister.size();
            std::string tmpReg = "v" + std::to_string(vregCounter + TEMPReg.size());
            TEMPReg.push(tmpReg); // 将临时寄存器加入栈
            if (argVal->getType()->getPointerElementType()->isIntegerTy(32)) {
                mir.emplace_back(MOpcode::LOAD_INT, tmpReg, offset, baseReg);
```

```
            } else if (argVal->getType()->getPointerElementType()->isFloatTy()) {
                mir.emplace_back(MOpcode::LOAD_FLOAT, tmpReg, offset, baseReg,
                    std::vector<ExtraDatum>(), // 没有初始值
                    true); // 标记为浮点数
                isFloat = true; // 标记为浮点数
            } else if (argVal->getType()->getPointerElementType()->isIntegerTy(8)) {
                mir.emplace_back(MOpcode::LOAD_CHAR, tmpReg, offset, baseReg);
            } else {
                mir.emplace_back(MOpcode::LOAD_INT, tmpReg, offset, baseReg);
            }
            argReg = tmpReg; // 更新为临时寄存器
            valueToRegister[argVal] = tmpReg; // 更新寄存器映射
        }
        argRegs.push_back(ExtraDatum(argReg, isFloat)); // 包装成 ExtraDatum
    }

    // 3. 判断是否有返回值
    Type *retType = calledFunc->getReturnType();

    if (retType->isVoidTy()) {
        // 无返回值调用
        mir.emplace_back(MOpcode::CALL, "",funcName,"", argRegs);
    } else {
        // 有返回值调用，分配寄存器保存结果
        int vregCounter = valueToRegister.size() + TEMPReg.size();
        std::string retReg = "v" + std::to_string(vregCounter);
        TEMPReg.push(retReg);

        // argRegs 是所有参数的寄存器名列表
        // 把参数列表转换为单个字符串（例如逗号分隔），或者直接传给 extra
        mir.emplace_back(MOpcode::CALL, retReg, funcName, "", argRegs);

        // 保存返回值寄存器映射
        valueToRegister[call] = retReg;
    }
}
```

⑦ 加载指令 LoadInst

handleLoadInst 函数的主要功能是处理 LLVM 中的 LoadInst 指令，将其转换为中间表示 MIR 的加载指令。函数首先获取 load 指令的指针操作数，并检查该指针是否为 GetElementPtr 指令，如果是则调用 handleGetElementPtrInst_forLoad 函数处理 GEP 指令并计算地址寄存器。如果指针是全局变量，则生成对应的加载指令并将结果存入临时寄存器。handleGetElementPtrInst_forLoad 函数是 handleGetElementPtrInst 函数的变体，其处理逻辑基本一致但是参数不同。

```
void handleLoadInst(const llvm::LoadInst *load,
                    std::vector<MInstruction> &mir,
```

```
                std::unordered_map<const llvm::Value*, std::string> &valueToRegister,
                const StackFrame &frame,MIRProgram &program) {
    using namespace llvm;

    const Value *ptr = load->getPointerOperand();
    std::string addrReg;

    // 假设 load 是 const llvm::LoadInst* 类型
    load->print(llvm::outs());
    llvm::outs() << "\n";

    std::cout << "[DEBUG] ptr = ";
    ptr->print(llvm::outs());
    llvm::outs() << "\n";

    bool isGlobal = false;
    // 如果 ptr 是 GEP，先调用 handleGetElementPtrInst，计算地址寄存器
    if (const llvm::ConstantExpr *ce = llvm::dyn_cast<llvm::ConstantExpr>(ptr)) {
        if (ce->getOpcode() == llvm::Instruction::GetElementPtr) {
            llvm::Type *gepElemType = nullptr;

            llvm::Value *ptrOperand = ce->getOperand(0);
            llvm::Type *ptrOperandType = ptrOperand->getType();

            if (auto pt = llvm::dyn_cast<llvm::PointerType>(ptrOperandType)) {
                gepElemType = pt->getElementType();  // 这里才是正确的指针元素类型
            } else {
                std::cerr << "[Error] ptrOperand type is not PointerType.\n";
                // 返回或处理错误
            }

            llvm::SmallVector<llvm::Value *, 8> idxs;
            for (auto it = ce->op_begin() + 1; it != ce->op_end(); ++it) {
                idxs.push_back(it->get());
            }

            llvm::GetElementPtrInst *gepInst = llvm::GetElementPtrInst::Create(
                gepElemType,
                ptrOperand,
                idxs,
                "tmp_gep",
                (llvm::Instruction *)nullptr
            );

            if (gepInst) {
                std::cout << "[DEBUG] 处理 GEP 指令，计算地址寄存器\n";
                valueToRegister[gepInst]    =    "v"    +    std::to_string(valueToRegister.size()    +
TEMPReg.size());
                TEMPReg.push(valueToRegister[gepInst]);
                std:: cout<<"[DEBUG]gepinst 内容"<<gepInst->getName().str()<<std::endl;
                gepInst->print(llvm::outs());
                llvm::outs() << "\n";
```

```
            const llvm::DataLayout &DL = load->getModule()->getDataLayout();

            handleGetElementPtrInst_forLoad(gepInst, mir, valueToRegister, frame, DL,program);


            std::cout << "[DEBUG] GEP 计算完成，地址寄存器: " << valueToRegister[gepInst] << "\n";
            addrReg = valueToRegister[gepInst];
        } else {
            std::cerr << "[Error] Failed to create temporary GEP instruction.\n";
        }
    }
}
else {
    // 否则直接从 valueToRegister 中获取寄存器或操作数字符串
    addrReg = getOperandStr(ptr, valueToRegister, frame, mir,program);
    //如果是全局变量，可能需要加载到寄存器
    if (const GlobalVariable *gv = llvm::dyn_cast<GlobalVariable>(ptr)) {
        isGlobal = true;
        std::string tempReg = "v" + std::to_string(valueToRegister.size() + TEMPReg.size());
        TEMPReg.push(tempReg); // 将临时寄存器加入栈
        if (gv->getType()->getPointerElementType()->isIntegerTy(32)) {
            mir.emplace_back(MOpcode::LOAD_INT, tempReg, "0", addrReg);
        } else if (gv->getType()->getPointerElementType()->isFloatTy()) {
            mir.emplace_back(MOpcode::LOAD_FLOAT, tempReg, "0", addrReg,
                std::vector<ExtraDatum>(), // 没有初始值
                true); // 标记为浮点数
        } else if (gv->getType()->getPointerElementType()->isIntegerTy(8)) {
            mir.emplace_back(MOpcode::LOAD_CHAR, tempReg, "0", addrReg);
        } else {
            mir.emplace_back(MOpcode::LOAD_INT, tempReg, "0", addrReg);
        }
        //valueToRegister[ptr] = tempReg; // 更新寄存器映射
        valueToRegister.erase(ptr);
        addrReg = tempReg; // 更新 addrReg 为临时寄存器
    }
}
```

函数随后根据加载值的类型分配目标寄存器，并判断地址寄存器是否为栈偏移形式。如果是栈偏移形式，则直接使用偏移和基址寄存器生成对应的加载指令；如果不是栈偏移形式且不是全局变量，则以 0 为偏移量生成加载指令。对于全局变量，函数会直接返回而不生成额外的加载指令。最后将目标寄存器与 load 指令关联起来，存储在 valueToRegister 映射中。

```
const Type *valType = load->getType();

// 分配用于加载值的寄存器
std::string vreg = "v" + std::to_string(valueToRegister.size() + TEMPReg.size());
TEMPReg.push(vreg);
```

```
    std::string offset_val, baseReg_val;

    // 判断 addrReg 是否是类似 "4(sp)" 这种偏移形式
    if (parseStackOffset(addrReg, offset_val, baseReg_val)) {
        // 是栈偏移形式，直接用偏移+基址加载
        if (valType->isIntegerTy(32)) {
            mir.emplace_back(MOpcode::LOAD_INT, vreg, offset_val, baseReg_val);
        } else if (valType->isFloatTy()) {
            mir.emplace_back(MOpcode::LOAD_FLOAT, vreg, offset_val, baseReg_val,
                std::vector<ExtraDatum>(), // 没有初始值
                true); // 标记为浮点数
        } else if (valType->isIntegerTy(8)) {
            mir.emplace_back(MOpcode::LOAD_CHAR, vreg, offset_val, baseReg_val);
        } else {
            mir.emplace_back(MOpcode::LOAD_INT, vreg, offset_val, baseReg_val);
        }
    } else if (!isGlobal) {
        // 不是偏移形式，addrReg 本身是寄存器名，偏移 0 加载
        if (valType->isIntegerTy(32)) {
            mir.emplace_back(MOpcode::LOAD_INT, vreg, "0", addrReg);
        } else if (valType->isFloatTy()) {
            mir.emplace_back(MOpcode::LOAD_FLOAT, vreg, "0", addrReg,
                std::vector<ExtraDatum>(), // 没有初始值
                true); // 标记为浮点数
        } else if (valType->isIntegerTy(8)) {
            mir.emplace_back(MOpcode::LOAD_CHAR, vreg, "0", addrReg);
        } else {
            mir.emplace_back(MOpcode::LOAD_INT, vreg, "0", addrReg);
        }
    } else {
        valueToRegister[load] = addrReg;
        return; // 如果是全局变量，直接返回
    }

    valueToRegister[load] = vreg;

}
```

⑧ 地址指针获取指令 GetElementPtrInst

handleGetElementPtrInst 函数用于将 LLVM 中的 GEP 指令翻译为 MIR 形式，用于计算结构体或数组元素在内存中的偏移地址。

函数一开始提取了 GEP 的基地址操作数，并将其转换为 MIR 层的寄存器或地址表达形式，作为偏移计算的起点。随后，通过 LLVM 的 DataLayout 获取类型大小信息，并初始化一个 totalOffsetReg，用于累加最终的内存偏移值，这个寄存器最初被赋值为常数 0。

```
void handleGetElementPtrInst(const llvm::GetElementPtrInst *gep,
                             std::vector<MInstruction> &mir,
                             std::unordered_map<const llvm::Value*, std::string> &valueToRegister,
```

```
                          const StackFrame &frame,MIRProgram &program) {
    using namespace llvm;

    const Value *ptrOperand = gep->getPointerOperand();
    std::string baseReg = getOperandStr(ptrOperand, valueToRegister, frame, mir,program);
    std::cout << "[MIR] 处理 GEP，基址: " << baseReg << std::endl;

    const DataLayout &DL = gep->getModule()->getDataLayout();
    Type *curType = gep->getSourceElementType();

    std::string totalOffsetReg = "v" + std::to_string(valueToRegister.size() + TEMPReg.size());
    TEMPReg.push(totalOffsetReg); // 将临时寄存器加入栈
    mir.emplace_back(MOpcode::LI, totalOffsetReg, "0");
```

在此之后跳过 GEP 的第一个索引，因为第一个索引通常表示的是数组中哪一个元素，一般是零，不影响偏移计算。进入主循环后逐一处理 GEP 指令中的每一个索引值，并判断当前类型是否为数组类型。如果是数组类型，先计算该类型元素的大小，然后将索引值乘以元素大小，得到当前层级的偏移量，接着将其加到 totalOffsetReg 中，从而累计总偏移。每处理完一层索引，类型信息也会随之更新，进入下一个元素类型，为下一次偏移计算做准备。

```
    // 遍历所有索引
    auto idxIter = gep->idx_begin();
    std::cout << "[MIR] 处理 GEP，初始类型: " << typeToString(curType) << std::endl;
    std::cout << "[MIR] idxIter 初始位置: " << idxIter << std::endl;
    ++idxIter;

    for (; idxIter != gep->idx_end(); ++idxIter) {
        const Value *indexVal = idxIter->get();
        std::string indexReg = getOperandStr(indexVal, valueToRegister, frame, mir,program);

        if (curType->isArrayTy()) {
            auto *arrayTy = dyn_cast<ArrayType>(curType);
            Type *elemTy = arrayTy->getElementType();
            uint64_t elemSize = DL.getTypeAllocSize(elemTy);
             std::cout << "[MIR] 处理 GEP 数组元素，元素类型: " << typeToString(elemTy) << ", 大小: " <<
elemSize << std::endl;

            // size 寄存器
            std::string sizeReg = "v" + std::to_string(valueToRegister.size() + TEMPReg.size());
            TEMPReg.push(sizeReg); // 将临时寄存器加入栈
            mir.emplace_back(MOpcode::LI, sizeReg, std::to_string(elemSize));

            // offset = index * elemSize
            std::string subOffsetReg = "v" + std::to_string(valueToRegister.size() + TEMPReg.size());
            TEMPReg.push(subOffsetReg); // 将临时寄存器加入栈
            mir.emplace_back(MOpcode::MUL_INT, subOffsetReg, indexReg, sizeReg);

            // totalOffset += subOffset
                  std::string  newTotalOffsetReg  =  "v"  +  std::to_string(valueToRegister.size()  +
```

```
TEMPReg.size());
            TEMPReg.push(newTotalOffsetReg); // 将临时寄存器加入栈
            mir.emplace_back(MOpcode::ADD_INT, newTotalOffsetReg, totalOffsetReg, subOffsetReg);
            totalOffsetReg = newTotalOffsetReg;

            curType = elemTy; // 进入下一层类型
        }else{
            //处理非数组，是 int 类型或者 float 或者 char
            break;
        }
    }
```

当索引处理完成后，函数将基地址和总偏移相加，就得到最终地址。如果初始基址是栈偏移格式，则会先使用 ADD_INT 指令把该地址转换成纯寄存器形式，然后再与总偏移合并。最终结果被保存在一个新的虚拟寄存器中，并将该寄存器映射到当前的 GEP 指令节点，使得后续其他指令可以引用这个地址。

```
    // 最后：result = base + totalOffset
    std::string offsetStr, baseStr;
    std::string resolvedBaseReg = baseReg;
    if (parseStackOffset(baseReg, offsetStr, baseStr)) {
        // 是 4(sp) 形式，先 ADD_INT 计算基址
        std::string tempReg = "v" + std::to_string(valueToRegister.size() + TEMPReg.size());
        TEMPReg.push(tempReg); // 将临时寄存器加入栈
        mir.emplace_back(MOpcode::ADD_INT, tempReg, baseStr, offsetStr);
        resolvedBaseReg = tempReg;
    }

    std::string resultReg = "v" + std::to_string(valueToRegister.size() + TEMPReg.size());
    TEMPReg.push(resultReg); // 将临时寄存器加入栈
    mir.emplace_back(MOpcode::ADD_INT, resultReg, resolvedBaseReg, totalOffsetReg);

    valueToRegister[gep] = resultReg;
}
```

四.riscv 汇编的代码实现

1. 总体思路

本部分的核心目标是将中间表示（MIR）转换为对应的 RISC-V 汇编代码。转换过程中，需完成虚拟寄存器到物理寄存器的映射，并根据操作类型生成相应指令文本。为此，我们设计了整型和浮点两类寄存器池，结合映射器实现虚拟寄存器的动态分配与释放，确保寄存器资源合理利用。同时，为减少不必要的寄存器保留，还实现了对虚拟寄存器后续使用情况的静态分析，从而在合适的时机回收已不再使用的寄存器，提高生成代码的效率与质量。

2. 寄存器池的实现思路（RegisterPool 与 FloatRegisterPool）

为了支持将中间表示（MIR）中使用的虚拟寄存器（如 v1、v2 等）映射为 RISC-V 指令中使用的真实物理寄存器，我们设计了两个寄存器池类 RegisterPool 和 FloatRegisterPool，分别负责整型寄存器和浮点寄存器的分配与管理。

RegisterPool 维护一个包含所有可用整型寄存器的向量 allRegs，如 s1 到 s11，并使用 used 集合来跟踪当前已被占用的寄存器。当需要分配寄存器时，系统从 allRegs 中依次查找一个未被使用的寄存器返回；若没有空闲寄存器，则抛出运行时错误。类似地，FloatRegisterPool 实现了对浮点寄存器 fs1 到 fs11 的管理，供浮点数相关操作使用。

具体代码实现如下:

```
class RegisterPool {
    std::vector<std::string> allRegs{"s1", "s2", "s3", "s4", "s5", "s6", "s7", "s8", "s9", "s10",
"s11"}; // 可用的物理寄存器列表
    std::unordered_set<std::string> used; // 当前已分配出去的寄存器集合

public:
    std::string allocate() { // 从 allRegs 中找出未被使用的寄存器，并将其标记为已使用
        for (const auto &reg : allRegs) {
            if (used.find(reg) == used.end()) {
                used.insert(reg);
                return reg;
            }
        }
        throw std::runtime_error("No available registers"); // 若所有寄存器都被占用，则抛出异常（防止寄存
器溢出错误）
    }

    void release(const std::string &reg) { // 释放寄存器的使用权（从 used 中删除）
        used.erase(reg);
    }

};

// 管理浮点寄存器池 与 RegisterPool 类似
class FloatRegisterPool {
    std::vector<std::string> allFloatRegs{
        "fs1", "fs2", "fs3", "fs4", "fs5", "fs6", "fs7", "fs8", "fs9", "fs10", "fs11"};
    std::unordered_set<std::string> usedFloat;
public:
    std::string allocateFloat() {
        for (const auto &reg : allFloatRegs) {
            if (usedFloat.find(reg) == usedFloat.end()) {
                usedFloat.insert(reg);
                return reg;
            }
        }
        throw std::runtime_error("No available float registers");
```

```
    }
    void releaseFloat(const std::string &reg) {
        usedFloat.erase(reg);
    }
};
```

3. 寄存器的分配（RegisterAllocator 和 FloatRegisterAllocator）

在 RISC-V 汇编生成阶段，我们通过 RegisterAllocator 和 FloatRegisterAllocator 两个类将 MIR 中的虚拟寄存器映射为真实寄存器。其核心逻辑在于判断一个操作数是否是虚拟寄存器，如果是则分配一个真实寄存器，并记录映射关系，便于后续指令使用。

```
std::string mapVReg(const std::string &vreg) {
    if (vreg.empty() || vreg[0] != 'v') return vreg;
    if (vregMap.count(vreg)) return vregMap[vreg];
    std::string realReg = regPool.allocate();
    vregMap[vreg] = realReg;
    return realReg;
}
```

该函数首先检查当前虚拟寄存器是否已被分配过真实寄存器，若没有，则从寄存器池中获取一个空闲寄存器，并将其与虚拟寄存器绑定。在寄存器释放后映射表也会清除对应条目。

浮点数分配逻辑完全类似，只是操作对象变为浮点寄存器池和 fsX 类型寄存器。

具体代码实现如下：

```
class RegisterAllocator {
    RegisterPool &regPool;
    std::unordered_map<std::string, std::string> &vregMap;

public:
    RegisterAllocator(RegisterPool &pool, std::unordered_map<std::string, std::string> &map)
        : regPool(pool), vregMap(map) {}

        std::string mapVReg(const std::string &vreg) {
            if (vreg.empty() || vreg[0] != 'v') {
                // 非虚拟寄存器，直接返回原值
                return vreg;
            }

            // 是虚拟寄存器，检查是否已映射
            if (vregMap.count(vreg)) {
                return vregMap[vreg];
            }

            // 分配真实寄存器并映射
            std::string realReg = regPool.allocate();
            vregMap[vreg] = realReg;
            return realReg;
        }
```

```
    void releaseVReg(const std::string &vreg) {
        printf("释放寄存器 %s\n", vreg.c_str());
        if (vregMap.count(vreg)) {
            regPool.release(vregMap[vreg]);
            vregMap.erase(vreg);
        }
    }
};
class FloatRegisterAllocator {
    FloatRegisterPool &regPool;
    std::unordered_map<std::string, std::string> &vregMap;
public:
      FloatRegisterAllocator(FloatRegisterPool  &pool,  std::unordered_map<std::string,  std::string>
&map)
        : regPool(pool), vregMap(map) {}

    std::string mapVReg(const std::string &vreg) {
        if (vreg.empty() || vreg[0] != 'v') {
            // 非虚拟寄存器，直接返回原值
            return vreg;
        }
        // 是虚拟寄存器，检查是否已映射
        if (vregMap.count(vreg)) {
            return vregMap[vreg];
        }
        // 分配真实寄存器并映射
        std::string realReg = regPool.allocateFloat();
        vregMap[vreg] = realReg;
        return realReg;
    }
    void releaseVReg(const std::string &vreg) {
        printf("释放浮点寄存器 %s\n", vreg.c_str());
        if (vregMap.count(vreg)) {
            regPool.releaseFloat(vregMap[vreg]);
            vregMap.erase(vreg);
        }
    }
};
```

4. 寄存器的回收（willBeUsedInFutureBlocks）

willBeUsedInFutureBlocks 函数的主要作用是分析某个虚拟寄存器在后续基本块中是否仍会被使用，从而决定是否可以将其映射的物理寄存器释放。

该函数以当前所在的基本块名与指令索引为起点，依次扫描当前块的后续指令和程序中其后的基本块，判断是否存在使用该 vreg 的情况。若后续存在任何一条指令的 src1、src2、或额外数据字段使用了该寄存器，函数即返回 true；否则认为其生命周期已结束，可以释放。

首先，函数开始时定义了一个布尔变量 inCurrentBlock，用于标记是否已经遍历到当前

基本块。在遍历所有 MIRBlock 时，程序首先寻找当前基本块 currentBlockLabel，一旦找到，则设置 inCurrentBlock=true，并从该基本块中当前指令之后的位置开始检查目标虚拟寄存器 vreg 是否仍会被使用。这一阶段只关注当前块内、当前指令之后的部分。

```
    bool inCurrentBlock = false;
    for (const auto &block : blocks) {
        if (block.label == currentBlockLabel) {
            inCurrentBlock = true;
            for (size_t i = currentInstrIndex + 1; i < block.instructions.size(); ++i) {
                const auto &instr = block.instructions[i];
                if (instr.dst == vreg && (instr.op == MOpcode::LI || instr.op == MOpcode::LA || instr.op
== MOpcode::CALL))
                    return false;
                if (instr.src1 == vreg || instr.src2 == vreg)
                    return true;
                for (const auto &extra : instr.extraData) {
                    if (extra.name == vreg)
                        return true;
                }
            }
        }
```

接着，如果当前块中没有发现该寄存器的后续用途，程序将继续向后遍历后续的所有基本块。这一阶段假设：若该虚拟寄存器在后续的基本块中还会被用到，则必须保留；若发现它会在后续块中被覆盖（通过立即数、地址或调用的写入），则可以提前判断其不再被使用。

```
         else if (inCurrentBlock) {
            // 之后的所有 block 全部遍历
            for (const auto &instr : block.instructions) {
                if (instr.dst == vreg && (instr.op == MOpcode::LI || instr.op == MOpcode::LA || instr.op
== MOpcode::CALL))

                    return false;
                if (instr.src1 == vreg || instr.src2 == vreg){
                    printf("vreg %s will be used in future blocks\n", vreg.c_str());
                    return true;
                }
                for (const auto &extra : instr.extraData) {
                    if (extra.name == vreg)
                        return true;
                }
            }
        }
    }
```

5. 由 MIR 生成 riscv 汇编的核心函数（generateRISCV）

① 全局变量的声明

首先，该函数会遍历 MIR 中的全局变量与外部函数，并将它们翻译为对应的.data

或.extern 汇编指令。在第一次遇到全局变量时，它会插入.data 段的声明。根据变量的类型（如 int、float、double 等），决定采用.word、.float 还是.double 等指令进行初始化值的写入。如果是数组类型，则多个值被连续输出。对于外部函数，使用.extern 直接标识函数名，便于链接器处理函数调用。

```
    for (const auto &global : mir.globalInstructions) {
        switch (global.op) {
            case MOpcode::VAR_GLOBAL: {
                if (!hasDataSection) {
                    out << ".data\n";
                    hasDataSection = true;
                }
                const std::string &name = global.dst;
                const std::string &type = global.src1;
                const std::string &size = global.src2;
                std::vector<std::string> values;
                for(const auto &extra : global.extraData) {
                   values.push_back(extra.name);
                }

                out << ".size " << name << ", " << size << "\n";
                out << name << ":\n";

                std::string directive;
                if (type == "int" || type == "int_array") {
                    directive = ".word";
                } else if (type == "float" || type == "float_array") {
                    directive = ".float";
                } else if (type == "double") {
                    directive = ".double";
                } else {
                    directive = ".word"; // 默认情况
                }

                for (const auto &val : values) {
                    out << "  " << directive << " " << val << "\n";
                }

                break;
            }
            case MOpcode::EXTERN_FUNC:
                out << ".extern " << global.dst << "\n";
                break;
            default:
                break;
        }
    }
```

② 静态浮点数声明

接着，该函数开始处理静态浮点数常量的声明。这些静态浮点常量保存在 mir.staticFloats 中，处理时会给每个静态浮点数分配一个符号名，并使用.float 汇编指令写入.data 段，同时为每个静态常量加上.size 信息。写完数据段后，进入.text 段准备生成函数指令。

```
    for (const auto &flt : mir.staticFloats) {
        if(!hasDataSection) {
            out << ".data\n";
            hasDataSection = true;
        }
        const std::string &name = flt.symbol;
        const double value = flt.value;
        out << ".size " << name << ", 4\n"; // 假设静态浮点数大小为 4 字节
        out << name << ":\n";
        out << "  .float " << value << "\n";
    }
    out << "\n.text\n";
```

③ 函数内指令生成

然后，进入 MIR 中每个函数的指令生成阶段。函数名将被声明为全局.globl 标签，每个基本块则以其 label 为开头输出。此过程中创建了整数寄存器和浮点寄存器的分配器 RegisterAllocator 和 FloatRegisterAllocator，分别处理虚拟寄存器与物理寄存器之间的映射。指令遍历时会首先判断是否为浮点操作，并根据不同情况从正确的寄存器池分配物理寄存器。每条 MIR 指令根据其类型（如算术操作、跳转、调用、加载/存储等）翻译为对应的 RISC-V 汇编语句。

```
    for (const auto &flt : mir.staticFloats) {
        if(!hasDataSection) {
            out << ".data\n";
            hasDataSection = true;
        }
        const std::string &name = flt.symbol;
        const double value = flt.value;
        out << ".size " << name << ", 4\n"; // 假设静态浮点数大小为 4 字节
        out << name << ":\n";
        out << "  .float " << value << "\n";
    }
    out << "\n.text\n";

    // 2. Emit functions
    for (const auto &func : mir.functions) {
        out << ".globl " << func.name << "\n";
        out << func.name << ":\n";

        RegisterPool regPool;
        FloatRegisterPool floatRegPool;
        std::unordered_map<std::string, std::string> vregMap;
        std::unordered_map<std::string, std::string> floatVregMap;
```

```
        RegisterAllocator regAlloc(regPool, vregMap);
        FloatRegisterAllocator floatRegAlloc(floatRegPool, floatVregMap);

        for (const auto &block : func.blocks) {
            out << block.label << ":\n";

            const auto &instructions = block.instructions;
            for (size_t i = 0; i < instructions.size(); ++i) {
                const auto &instr = instructions[i];
                std::string dst, src1, src2;
                bool isfloat = false;

                if(instr.op == MOpcode::ADD_FLOAT || instr.op == MOpcode::SUB_FLOAT ||
                    instr.op == MOpcode::MUL_FLOAT || instr.op == MOpcode::DIV_FLOAT ||
                    instr.op == MOpcode::MOV_FLOAT || instr.op == MOpcode::STORE_FLOAT ||
                    instr.op == MOpcode::LOAD_FLOAT || instr.op == MOpcode::ICMP_EQ_FLOAT ||
                    instr.op == MOpcode::ICMP_NE_FLOAT || instr.op == MOpcode::ICMP_LT_FLOAT ||
                    instr.op == MOpcode::ICMP_LE_FLOAT || instr.op == MOpcode::ICMP_GT_FLOAT ||
                    instr.op == MOpcode::ICMP_GE_FLOAT
                 ) {
                    isfloat = true;
                }
                if(instr.isfloat){
                    isfloat = true; // 如果指令标记为浮点数，则设置 isfloat 为 true
                }
                if(isfloat && ( instr.op == MOpcode::STORE_FLOAT ||instr.op == MOpcode::LOAD_FLOAT )){
                    dst = floatRegAlloc.mapVReg(instr.dst);
                    src1 = instr.src1.empty() ? "" : regAlloc.mapVReg(instr.src1);
                    src2 = instr.src2.empty() ? "" : regAlloc.mapVReg(instr.src2);

                }else if(isfloat) {
                    dst = floatRegAlloc.mapVReg(instr.dst);
                    src1 = instr.src1.empty() ? "" : floatRegAlloc.mapVReg(instr.src1);
                    src2 = instr.src2.empty() ? "" : floatRegAlloc.mapVReg(instr.src2);

                }else {
                    dst = regAlloc.mapVReg(instr.dst);
                    src1 = instr.src1.empty() ? "" : regAlloc.mapVReg(instr.src1);
                    src2 = instr.src2.empty() ? "" : regAlloc.mapVReg(instr.src2);
                }

                switch (instr.op) {
                    case MOpcode::ADD_INT:
                        out << "  add " << dst << ", " << src1 << ", " << src2 << "\n";
                        break;
                    case MOpcode::SUB_FLOAT:
                        out << "  fsub.s " << dst << ", " << src1 << ", " << src2 << "\n";
                        break;
                    case MOpcode::ADD_FLOAT:
                        out << "  fadd.s " << dst << ", " << src1 << ", " << src2 << "\n";
                        break;
                    case MOpcode::SUB_INT:
                        out << "  sub " << dst << ", " << src1 << ", " << src2 << "\n";
```

```
                    break;

                case MOpcode::LOAD_INT:
                    out << "  lw " << dst << ", " << src1 << "(" << src2 << ")\n";
                    break;
                case MOpcode::STORE_INT:
                    out << "  sw " << dst << ", " << src1 << "(" << src2 << ")\n";
                    if (!willBeUsedInFutureBlocks(func.blocks, block.label, i, instr.dst))
                        regAlloc.releaseVReg(instr.dst);
                    break;
                case MOpcode::LOAD_FLOAT:
                    out << "  flw " << dst << ", " << src1 << "(" << src2 << ")\n";
                    break;
                case MOpcode::STORE_FLOAT:
                    out << "  fsw " << dst << ", " << src1 << "(" << src2 << ")\n";
                    if (!willBeUsedInFutureBlocks(func.blocks, block.label, i, instr.dst))
                        floatRegAlloc.releaseVReg(instr.dst);
                    break;
                case MOpcode::LOAD_CHAR:
                    out << "  lb " << dst << ", " << src1 << "(" << src2 << ")\n";
                    break;
                case MOpcode::STORE_CHAR:
                    out << "  sb " << dst << ", " << src1 << "(" << src2 << ")\n";
                    if (!willBeUsedInFutureBlocks(func.blocks, block.label, i, instr.dst))
                        regAlloc.releaseVReg(instr.dst);
                    break;
                case MOpcode::LI:
                    out << "  li " << dst << ", " << instr.src1 << "\n";
                    break;
                case MOpcode::LA:
                    out << "  la " << dst << ", " << instr.src1 << "\n";
                    break;
                case MOpcode::RET:
                    out << "  ret\n";
                    break;
                case MOpcode::MOV_INT:
                    out << "  mv " << dst << ", " << src1 << "\n";
                    break;
                case MOpcode::MOV_FLOAT:
                    out << "  fmv.s " << dst << ", " << src1 << "\n";
                    break;
                case MOpcode::MOV_CHAR:
                    out << "  mv " << dst << ", " << src1 << "\n";
                    out << "  andi " << dst << ", " << dst << ", 0xFF\n"; // 截断为8位
                    break;
                case MOpcode::MUL_INT:
                    out << "  mul " << dst << ", " << src1 << ", " << src2 << "\n";
                    break;
                case MOpcode::DIV_INT:
                    out << "  div " << dst << ", " << src1 << ", " << src2 << "\n";
                    break;
                case MOpcode::DIV_FLOAT:
                    out << "  fdiv.s " << dst << ", " << src1 << ", " << src2 << "\n";
```

```
            break;
        case MOpcode::MUL_FLOAT:
            out << "  fmul.s " << dst << ", " << src1 << ", " << src2 << "\n";
            break;
        case MOpcode::SREM:
            out << "  rem " << dst << ", " << src1 << ", " << src2 << "\n";
            break;

        case MOpcode::CALL: {
            // src1: 返回值寄存器（如果无返回值则为空）
            // src2: 函数名
            // extra: 参数寄存器列表（std::vector<std::string>）

            // 先将参数寄存器装载到约定的 a0~a7 寄存器
            // 一般 RISC-V 前 8 个整型参数用 a0-a7 传递
            const auto &argRegs = instr.extraData; // 假设 extra 是存储参数寄存器名的 vector

            // 参数个数最多 8 个，超过部分一般在栈上（这里暂不处理）
            for (size_t i = 0; i < argRegs.size() && i < 8; ++i) {
                if(argRegs[i].isFloat) {
                    // 如果是浮点参数，使用浮点寄存器
                    std::string argRegs_f = floatRegAlloc.mapVReg(argRegs[i].name);
                    out << "  fmv.s fa" << i << ", " << argRegs_f << "\n";
                } else {
                    // 整型参数使用整数寄存器
                    std::string argRegs_i = regAlloc.mapVReg(argRegs[i].name);
                    out << "  mv a" << i << ", " << argRegs_i << "\n";
                }
            }

            // 调用函数
            out << "  call " << src1 << "\n";

            // 返回值写回目标寄存器
            // 如果有返回寄存器名
            if (!dst.empty()) {
                out << "  mv " << dst << ", a0\n";
            }
            break;
        }
        case MOpcode::JMP:
            out << "  j " << instr.dst << "\n";
            break;
        case MOpcode::BEQ:
            out << "  beq " << src1 << ", " << src2 << ", " << instr.dst << "\n";
            break;
        case MOpcode::BNE:
            out << "  bne " << src1 << ", " << src2 << ", " << instr.dst << "\n";
            break;
        case MOpcode::BGT:
            out << "  bgt " << src1 << ", " << src2 << ", " << instr.dst << "\n";
            break;
```

```
                case MOpcode::BLT:
                    out << "  blt " << src1 << ", " << src2 << ", " << instr.dst << "\n";
                    break;
                case MOpcode::BGE:
                    out << "  bge " << src1 << ", " << src2 << ", " << instr.dst << "\n";
                    break;
                case MOpcode::BLE:
                    out << "  ble " << src1 << ", " << src2 << ", " << instr.dst << "\n";
                    break;
                // === ICMP（整数或浮点比较） ===
                case MOpcode::ICMP_EQ_INT:
                    out << "  sub " << dst << ", " << src1 << ", " << src2 << "\n";
                    out << "  seqz " << dst << ", " << dst << "\n"; // 相等：减法后为0，seqz 判断
                    break;
                case MOpcode::ICMP_NE_INT:
                    out << "  sub " << dst << ", " << src1 << ", " << src2 << "\n";
                    out << "  snez " << dst << ", " << dst << "\n"; // 不等
                    break;
                case MOpcode::ICMP_LT_INT:
                    out << "  slt " << dst << ", " << src1 << ", " << src2 << "\n";
                    break;
                case MOpcode::ICMP_LE_INT:
                    out << "  sgt " << dst << ", " << src1 << ", " << src2 << "\n";
                    out << "  xori " << dst << ", " << dst << ", 1\n"; // dst = !(src1 > src2)
                    break;
                case MOpcode::ICMP_GT_INT:
                    out << "  sgt " << dst << ", " << src1 << ", " << src2 << "\n";
                    break;
                case MOpcode::ICMP_GE_INT:
                    out << "  slt " << dst << ", " << src1 << ", " << src2 << "\n";
                    out << "  xori " << dst << ", " << dst << ", 1\n"; // dst = !(src1 < src2)
                    break;

                // === 浮点数比较（使用浮点比较伪指令）===
                case MOpcode::ICMP_EQ_FLOAT:
                    out << "  feq.s " << dst << ", " << src1 << ", " << src2 << "\n";
                    break;
                case MOpcode::ICMP_NE_FLOAT:
                    out << "  feq.s " << dst << ", " << src1 << ", " << src2 << "\n";
                    out << "  xori " << dst << ", " << dst << ", 1\n"; // 取反
                    break;
                case MOpcode::ICMP_LT_FLOAT:
                    out << "  flt.s " << dst << ", " << src1 << ", " << src2 << "\n";
                    break;
                case MOpcode::ICMP_LE_FLOAT:
                    out << "  fle.s " << dst << ", " << src1 << ", " << src2 << "\n";
                    break;
                case MOpcode::ICMP_GT_FLOAT:
                    out << "  flt.s " << dst << ", " << src2 << ", " << src1 << "\n"; // 交换顺序
                    break;
                case MOpcode::ICMP_GE_FLOAT:
                    out << "  fle.s " << dst << ", " << src2 << ", " << src1 << "\n"; // 交换顺序
```

```
                    break;

                default:
                    out << "  # unknown opcode\n";
                    break;
            }
```

④ 寄存器释放策略

最后，在每条指令执行完后，会进行一次寄存器释放策略的判断，以避免不必要的寄存器占用，提高寄存器利用率。释放策略通过调用 willBeUsedInFutureBlocks 函数来判断某个虚拟寄存器是否还会在未来使用（包括当前 block 的剩余部分以及后续 block）。若不会再使用，则对应的物理寄存器可立即释放，以供后续指令分配使用。

```
            if (!willBeUsedInFutureBlocks(func.blocks, block.label, i, instr.src1)) {
                floatRegAlloc.releaseVReg(instr.src1);
                regAlloc.releaseVReg(instr.src1);
            }

            if (!willBeUsedInFutureBlocks(func.blocks, block.label, i, instr.src2)){
                floatRegAlloc.releaseVReg(instr.src2);
                regAlloc.releaseVReg(instr.src2);
            }
```

五.实验结果

以 test09.cact 为例，以下为 test09 的代码。

```
int defn(){
    return 4;
}

int main(){
    int a=0;
    a = defn();
    return a;
}
```

实验 2 生成的结果如下。

```
; ModuleID = 'main'
source_filename = "main"

declare void @print_int(i32)

declare void @print_float(float)

declare void @print_char(i8)
```

```
declare i32 @get_int()

declare float @get_float()

declare i8 @get_char()

define i32 @defn() {
entry:
  ret i32 4
}

define i32 @main() {
entry:
  %a = alloca i32, align 4
  store i32 0, i32* %a, align 4
  %a_elem_ptr = getelementptr inbounds i32, i32* %a, i32 0
  %calltmp = call i32 @defn()
  store i32 %calltmp, i32* %a_elem_ptr, align 4
  %a_var = load i32, i32* %a, align 4
  ret i32 %a_var
}
```

本次实验生成的 MIR 代码如下。

```
EXTERN_FUNC print_int, void, int
EXTERN_FUNC print_float, void, float
EXTERN_FUNC print_char, void, char
EXTERN_FUNC get_int, int
EXTERN_FUNC get_float, float
EXTERN_FUNC get_char, char

LABEL defn
LABEL defn_enterpoint
ADD_INT sp, sp, -96
STORE_INT ra, 92, sp
STORE_INT s0, 88, sp
STORE_INT s1, 84, sp
STORE_INT s2, 80, sp
STORE_INT s3, 76, sp
STORE_INT s4, 72, sp
STORE_INT s5, 68, sp
STORE_INT s6, 64, sp
STORE_INT s7, 60, sp
STORE_INT s8, 56, sp
STORE_INT s9, 52, sp
STORE_INT s10, 48, sp
STORE_INT s11, 44, sp
STORE_FLOAT fs1, 40, sp
STORE_FLOAT fs2, 36, sp
STORE_FLOAT fs3, 32, sp
STORE_FLOAT fs4, 28, sp
```

```
STORE_FLOAT fs5, 24, sp
STORE_FLOAT fs6, 20, sp
STORE_FLOAT fs7, 16, sp
STORE_FLOAT fs8, 12, sp
STORE_FLOAT fs9, 8, sp
STORE_FLOAT fs10, 4, sp
STORE_FLOAT fs11, 0, sp
LABEL defn_entry
LI v0, 4
MOV_INT a0, v0
LOAD_INT ra, 92, sp
LOAD_INT s0, 88, sp
LOAD_INT s1, 84, sp
LOAD_INT s2, 80, sp
LOAD_INT s3, 76, sp
LOAD_INT s4, 72, sp
LOAD_INT s5, 68, sp
LOAD_INT s6, 64, sp
LOAD_INT s7, 60, sp
LOAD_INT s8, 56, sp
LOAD_INT s9, 52, sp
LOAD_INT s10, 48, sp
LOAD_INT s11, 44, sp
LOAD_FLOAT fs1, 40, sp
LOAD_FLOAT fs2, 36, sp
LOAD_FLOAT fs3, 32, sp
LOAD_FLOAT fs4, 28, sp
LOAD_FLOAT fs5, 24, sp
LOAD_FLOAT fs6, 20, sp
LOAD_FLOAT fs7, 16, sp
LOAD_FLOAT fs8, 12, sp
LOAD_FLOAT fs9, 8, sp
LOAD_FLOAT fs10, 4, sp
LOAD_FLOAT fs11, 0, sp
ADD_INT sp, sp, 96
RET

LABEL main
LABEL main_enterpoint
ADD_INT sp, sp, -8
STORE_INT ra, 4, sp
LABEL main_entry
LI v4, 0
STORE_INT v4, 0, sp
LI v6, 0
ADD_INT v7, sp, 0
ADD_INT v8, v7, v6
CALL v9, defn
STORE_INT v9, 0, v8
LOAD_INT v10, 0, sp
MOV_INT a0, v10
LOAD_INT ra, 4, sp
ADD_INT sp, sp, 8
```

```
RET
```

生成的 riscv 汇编结果如下。

```
.extern print_int
.extern print_float
.extern print_char
.extern get_int
.extern get_float
.extern get_char

.text
.globl defn
defn:
defn_enterpoint:
  add sp, sp, -96
  sw ra, 92(sp)
  sw s0, 88(sp)
  sw s1, 84(sp)
  sw s2, 80(sp)
  sw s3, 76(sp)
  sw s4, 72(sp)
  sw s5, 68(sp)
  sw s6, 64(sp)
  sw s7, 60(sp)
  sw s8, 56(sp)
  sw s9, 52(sp)
  sw s10, 48(sp)
  sw s11, 44(sp)
  fsw fs1, 40(sp)
  fsw fs2, 36(sp)
  fsw fs3, 32(sp)
  fsw fs4, 28(sp)
  fsw fs5, 24(sp)
  fsw fs6, 20(sp)
  fsw fs7, 16(sp)
  fsw fs8, 12(sp)
  fsw fs9, 8(sp)
  fsw fs10, 4(sp)
  fsw fs11, 0(sp)
defn_entry:
  li s1, 4
  mv a0, s1
  lw ra, 92(sp)
  lw s0, 88(sp)
  lw s1, 84(sp)
  lw s2, 80(sp)
  lw s3, 76(sp)
  lw s4, 72(sp)
  lw s5, 68(sp)
  lw s6, 64(sp)
  lw s7, 60(sp)
```

```
  lw s8, 56(sp)
  lw s9, 52(sp)
  lw s10, 48(sp)
  lw s11, 44(sp)
  flw fs1, 40(sp)
  flw fs2, 36(sp)
  flw fs3, 32(sp)
  flw fs4, 28(sp)
  flw fs5, 24(sp)
  flw fs6, 20(sp)
  flw fs7, 16(sp)
  flw fs8, 12(sp)
  flw fs9, 8(sp)
  flw fs10, 4(sp)
  flw fs11, 0(sp)
  add sp, sp, 96
  ret
.globl main
main:
main_enterpoint:
  add sp, sp, -8
  sw ra, 4(sp)
main_entry:
  li s1, 0
  sw s1, 0(sp)
  li s1, 0
  add s2, sp, 0
  add s3, s2, s1
  call defn
  mv s1, a0
  sw s1, 0(s3)
  lw s1, 0(sp)
  mv a0, s1
  lw ra, 4(sp)
  add sp, sp, 8
  ret
```

实验使用以下脚本来运行结果。

```
echo "[*] 正在编译..."
riscv32-unknown-linux-gnu-gcc -static -o test_static output.s ./libcact_rt.a
if [ $? -ne 0 ]; then
    echo "[!] 编译失败"
    exit 1
fi

echo "[*] 正在运行程序..."
spike pk ./test_static
exit_code=$?

echo "[*] 程序退出码：$exit_code"
```

实验运行结果如下，结果正确。

```
root@2e8dba519866:~/cact/build# ./run_riscv.sh
[*] 正在编译...
[*] 正在运行程序...
[*] 程序退出码: 4
```

六.实验总结

通过本次实验，我们系统完成了从 LLVM IR 到 RISC-V 汇编代码的转换流程，深入理解了指令选择、寄存器分配与函数栈管理的协同工作机制。实验初期，我们以基础测试用例为起点，逐步构建起两级转换框架：首先生成虚拟寄存器形式的 MIR，采用四层结构（Program→Function→Block→Instruction）映射 LLVM IR 的逻辑层次；继而通过一对一指令翻译与动态寄存器分配，完成到 RISC-V 汇编的转换。这一过程中，我们实现了对算术运算、内存访问、控制流跳转和函数调用等关键语义的完整支持，为后续编译器优化奠定了坚实基础。

在技术实现层面，我们突破了多项关键挑战。针对函数栈管理，设计栈帧动态计算机制：通过扫描函数内 alloca 指令确定局部变量空间，结合 callee-saved 寄存器数量生成自适应栈帧，并在 prologue/epilogue 中插入寄存器保存与恢复指令，确保函数调用的上下文完整性。面对寄存器资源竞争，采用双池分离策略——整型寄存器池（s0-s11）与浮点寄存器池（fs0-fs11）独立管理，并通过 willBeUsedInFutureBlocks() 函数实时释放闲置寄存器，显著提升寄存器利用率。对于全局数据存储，通过识别 int/float/array 等类型特征，在.data 段生成带初始值的.word 或.float 声明，并建立静态浮点常量池避免冗余存储。

本次实验通过亲手实现从 IR 到汇编的完整转换，增强了我们对目标代码生成阶段实现细节的理解，使我们了解了寄存器分配策略和栈帧设计对性能和程序运行的重要作用，以及中间表示（MIR）的结构设计对翻译流程的清晰度的影响。