中国科学院大学
《编译原理（研讨课）》实验报告

姓名 吴修齐 学号 2022K8009922026
姓名 吴喆 学号 2022K8009929033
姓名 卢柯圳 学号 2022K8009929023
实验项目编号 prj23

一、 实验任务说明

- 基于生成的 LLVM IR 实现目标代码生成
- 掌握从 IR 到汇编的翻译过程，由抽象语法树 AST 生成中间表示 IR
- 编译 CACT 源码到 riscv64gc 汇编文件 (.S)
- 可以 MIR 阶段添加各种编译优化
- 可以扩展 CACT 语言特性

二、实验过程

实验设计

本次实验需要在遍历语法树的过程中，生成中间表示；而后根据中间表示，进行内存地址的分配，并生成目标代码。

设计中间表示的形式：三地址代码、操作数

- 设计三地址代码：基类、方法，以及各类操作的扩展类
- 设计操作数：基类、方法，对全局变量、局部变量、常量、标号等进行抽象

生成中间表示，完成语法制导翻译：

- 实现常量、局部变量、全局变量的定义、初始化
- 实现函数的调用、参数的传递、返回值的传递
- 实现整数、浮点数的运算
- 实现向量的整数、浮点数运算
- 实现 if、else、while

生成目标代码：

- 生成全局变量
- 为局部变量分配内存空间
- 设计变量从内存加载到寄存器、从寄存器存回内存的机制
- 翻译三地址代码到汇编代码

在实验过程中，我们一开始使用了助教提供的中间表示生成的代码，在一开始的编码过程中我们发现由于不熟悉这些代码，调试过程产生了一定的障碍，因此我们先根据先前对实验二代码思路的记忆和本地还保存的设计思路和少量代码，与同学进行了充分而又富有建设性的积极讨论，对中间表示的生成代码进行了重构，这个重构的过程也部分借鉴了助教提供的代码设计，同时又与我们组实验一中已经写好的 cact 代码相匹配，基于我们新构建的中间表示生成代码，我们继续完成了目标代码的生成任务，具体实现见下一部分。

实验实现

符号表

符号表设计：

```
class SymbolInfo {
private:
    std::string name;
    IROperand* operand;

public:
    std::string getName();
    void setOp(IROperand* op);
    IROperand* getOp();

    virtual DataType getDataType() = 0;
    virtual int getArraySize() = 0;
    virtual SymbolType getSymbolType() = 0;

    SymbolInfo(const std::string& name);
};
```

这是符号表的基础类，所有符号（常量、变量、数组、函数）都继承自它。
作用是记录符号名，定义类型接口。

CONST 符号子类：

```
class ConstSymbolInfo : public SymbolInfo {
private:
    DataType dataType;

public:
    virtual DataType getDataType() { return dataType; }
    virtual int getArraySize() { return -1; }
    virtual SymbolType getSymbolType() { return SymbolType::CONST; }

    ConstSymbolInfo(const std::string& name, DataType dataType);
};
```

它表示常量符号，记录类型和具体值。

函数符号子类：

```
class FuncSymbolInfo : public SymbolInfo {
private:
    DataType returnType;
    std::vector<SymbolInfo*> paramList;
    BlockInfo* blockInfo;

public:
    virtual DataType getDataType() { return returnType; }
    virtual int getArraySize() { return paramList.size(); }
    virtual SymbolType getSymbolType() { return SymbolType::FUNC; }
    std::vector<SymbolInfo*> getparamList() { return paramList; }
    int getparamNum() { return paramList.size(); }

    SymbolInfo* addParamVar(const std::string& name, DataType dataType);
    SymbolInfo* addParamArray(const std::string& name, DataType dataType);

    FuncSymbolInfo(const std::string& name, DataType returnType);
};
```

主要是表示函数符号，记录返回类型、参数列表、参数个数、函数体块。

语句块 BlockInfo：

```
class BlockInfo {
private:
    BlockInfo* parentBlock;
    FuncSymbolInfo* belongTo;
    std::map<std::string, SymbolInfo*> symbolTable;
    std::map<std::string, FuncSymbolInfo*> funcTable;
    std::vector<BlockInfo*> blockTable;

public:
    BlockInfo* getParentBlock() { return parentBlock; }
    SymbolInfo* lookUpSymbol(std::string symbolName);
    FuncSymbolInfo* lookUpFunc(std::string symbolName);

    ConstSymbolInfo* addNewConst(const std::string& name, DataType dataType);
    VarSymbolInfo* addNewVar(const std::string& name, DataType dataType);
    ConstArraySymbolInfo* addNewConstArray(const std::string& name, DataType dataType, int arraySize);
    VarArraySymbolInfo* addNewVarArray(const std::string& name, DataType dataType, int arraySize);

    FuncSymbolInfo* addNewFunc(const std::string& name, DataType returnType);

    BlockInfo* addNewBlock();
    BlockInfo* addNewBlock(FuncSymbolInfo* belongTo);

    BlockInfo(BlockInfo* parentBlock);
    BlockInfo(BlockInfo* parentBlock, FuncSymbolInfo* belongTo, const std::vector<SymbolInfo*>& paramList);
};
```

作用：语句块，支持嵌套、符号查找、符号添加。

符号查找：

```
SymbolInfo * BlockInfo::lookUpSymbol(std::string symbolName) {
    if (symbolTable.count(symbolName)) {
        return symbolTable[symbolName];
    }
    return parentBlock ? parentBlock->lookUpSymbol(symbolName) : nullptr;
}
```

作用：支持递归查找符号，实现作用域嵌套。

添加新符号：

```
ConstSymbolInfo * BlockInfo::addNewConst(const std::string & name, DataType type) {
    DUP_CHECK(symbolTable, name);
    auto * sym = new ConstSymbolInfo(name, type);
    symbolTable[name] = sym;
    return sym;
}
```

作用：添加新常量符号，防止重复定义。

其他类对应的函数也与此类似，按照类型编写代码即可，这里不再赘述。

语义分析和类型检查

关于类型的问题，由于 CACT 不允许任何形式的类型转换，因此类型问题常见于赋值、运算、传参、返回值等各个方面，这就需要我们在对应的符号表中进行查询某些变量的类型以判断是否发生错误。

定义错误也是急需我们进行修正的问题，如重复定义，未定义等待，此外还有一些条件表达式也需要考虑（判断条件必须为布尔类型）。

由于我们在符号表阶段已经在语句块中实现了嵌套结构，因此在该部分可以将遍历语法树后获得的信息向上级传递进行判断，从而实现报错。

由于语义分析的内容较多较杂，这里以语义分析和类型检查举例说明：

语义分析和类型检查：

```
void SemanticAnalysis::enterConstDefBasic(CACTParser::ConstDefBasicContext * ctx) {
    std::string name = ctx->Ident()->getText();

    ctx->thisSymbolInfo = currentBlock->addNewConst(name, currentDataType);
}
void SemanticAnalysis::exitConstDefBasic(CACTParser::ConstDefBasicContext * ctx) {
    std::string name = ctx->Ident()->getText();
    IRValue * initVal = ctx->constExp()->result;

    if (ctx->constExp()->dataType != currentDataType) {
        throw std::runtime_error("inital value data type not match.");
    }

    if (currentFunc) {
        IRVariable* irVar = irGen->newVar(name, currentDataType);
        irGen->assignBasic(currentDataType, irVar, initVal);
        ctx->thisSymbolInfo->setOp(irVar);

    } else {
        initVal->setName(name);
        ctx->thisSymbolInfo->setOp(initVal);
    }
}
```

```
void SemanticAnalysis::enterConstDefArray(CACTParser::ConstDefArrayContext * ctx) {
    std::string name = ctx->Ident()->getText();
    int arraySize = std::stoi(ctx->IntConst()->getText());

    ctx->thisSymbolInfo = currentBlock->addNewConstArray(name, currentDataType, arraySize);
}
void SemanticAnalysis::exitConstDefArray(CACTParser::ConstDefArrayContext * ctx) {
    std::string name = ctx->Ident()->getText();
    int arraySize = std::stoi(ctx->IntConst()->getText());
    IRValue * initVal = ctx->constArrExp()->result;

    if (ctx->constArrExp() && ctx->constArrExp()->result) {
        initVal = ctx->constArrExp()->result;

        if (ctx->constArrExp()->dataType != currentDataType) {
            throw std::runtime_error("inital value data type not match.");
        }

        if (ctx->constArrExp()->arraySize > arraySize) {
            throw std::runtime_error("inital value array size not match.");
        }
    } else {
        initVal = irGen->newValue(currentDataType);
    }
    initVal->fillValue(arraySize);

    if (currentFunc) {
        IRVariable * irVar = irGen->newVar(name, currentDataType, arraySize);
        irGen->assignArray(currentDataType, arraySize, irVar, initVal);
        ctx->thisSymbolInfo->setOp(irVar);

    } else {
        initVal->setName(name);
        ctx->thisSymbolInfo->setOp(initVal);
    }
}
```

功能：语义分析阶段，处理常量定义，区分数组和基本类型，添加符号。

中间表示的实现

三地址代码

在我们的设计中，中间表示的三地址代码基本形式如下：

```
class IRCode {
private:
    IROperation operation;
    IROperand * result;
    IROperand * arg1;
    IROperand * arg2;

public:
    IRCode(IROperation new_op, IROperand * new_result, IROperand * new_arg1, IROperand * new_arg2);

    IROperation get_op();
    IROperand * get_a1();
    IROperand * get_a2();
    IROperand * get_ans();

    virtual void print() = 0;
    virtual void genTargetCode(TargetCodeList * t) = 0;
};
```

这是三地址代码基类，所有 IR 指令都继承自它。

可以看出，在设计中，三地址均为指向参数项的指针，这样的操作使得我们不用再考虑重名问题，且可以直接访问变量。此外，这样的写法使得可以从一个基类拓展到每种具体的三地址操作，方便了我们之后的操作。各个基类需要重写 print 函数（用于输出中间表示）和 genTargetCode 函数（用于生成目标代码）。

举例说明：

IRAddInt：

```
#define DECLARE_BINARY_OP_CLASS(name, op_type) \
class IR_##name : public IRCode { \
public: \
    IR_##name(IROperand * result, IROperand * arg1, IROperand * arg2); \
    virtual void print(); \
    void genTargetCode(TargetCodeList * t); \
};

DECLARE_BINARY_OP_CLASS(int_add, ADD_INT)
```

宏展开后为：

```
class IR_int_add : public IRCode {
public:
    IR_int_add(IROperand * result, IROperand * arg1, IROperand * arg2);
    virtual void print();
    void genTargetCode(TargetCodeList * t);
};
```

实现：

```
IR_int_add::IR_int_add(IROperand * result, IROperand * arg1, IROperand * arg2)
    : IRCode(ADD_INT, result, arg1, arg2) { }

void IR_int_add::print() {
    // 输出三地址代码格式
}

void IR_int_add::genTargetCode(TargetCodeList * t) {
    // 生成 RISC-V 汇编加法指令
}
```

作用：整型加法 IR 指令，负责 print 和目标代码生成。

操作数

中间表示的操作数是设计中最困难的地方。我们既要保证操作数有较为统一的接口，以方便目标代码的生成，又要保证能够满足全局变量、局部变量、常量、标号等等形式各样的操作数的实际需要。

为了迎合目标代码的特点，方便目标代码的生成，我们将操作数抽象成了三类：标号、变量、值。

• 标号（IRLabel）：用于表示跳转目标位置的标签。

• 变量（IRVariable）：用于表示栈中的局部变量。其记录了数据类型、数组长度，以及 相对于帧指针的内存偏移量。

• 值（IRValue）：用于表示数据段中的全局变量、浮点常量、整数立即数。其记录了初始值、数据类型、数组长度，以及它的名称（作为目标代码中的标号）。

这里的变量（IRVariable）和值（IRValue）同时抽象了单个变量和数组。在存储意义上，单个变量和数组的区别仅仅是在长度上，其他方面并没有太大差别。

上述所说的“常量”，指的是形如 123、233.322 这样的常量数字，而不是带有 const 修饰符的“变量”。

操作数抽象的具体实现：

基类：IROperand

```
class IROperand {
public:
    virtual ~IROperand() {}
    virtual bool isVariable() = 0;
    virtual void setName(std::string) = 0;
    virtual std::string getName() = 0;
    virtual int getSize() = 0;
    virtual std::string getImme() = 0;
    virtual void alloc(TargetCodeList* t) = 0;
    virtual void loadTo(TargetCodeList* t, std::string reg) = 0;
    virtual void storeFrom(TargetCodeList* t, std::string reg) = 0;
    virtual void loadAddrTo(TargetCodeList* t, std::string reg) = 0;
};
```

标签：IRLabel

```
class IRLabel : public IROperand {
private:
    std::string name;
public:
    IRLabel(std::string newName);
    // ...实现了所有虚函数
};
```

变量：IRVariable

```
class IRVariable : public IROperand {
private:
    std::string name;
    DataType dataType;
    int length;
    int memOffset;
public:
    IRVariable(std::string newName, DataType dt, int len);
    IRVariable(std::string newName, DataType dt);
    // ...实现了所有虚函数
};
```

值：IRValue

```
class IRValue : public IROperand {
private:
    std::string name;
    DataType dataType;
    bool isVar;
    std::vector<std::string> values;
public:
    IRValue(std::string newName, DataType newDataType);
    IRValue(int newVal);
    // ...实现了所有虚函数
};
```

中间表示的生成

变量的定义与初始化

变量定义与初始化的过程，实际上就是根据上面的抽象产生操作数的过程。这是我们首先需要关注的问题，它是后面一切运算的基石。

对于全局变量，我们需要产生 IRValue，其本身也就记下了初值。

对于局部变量，我们需要产生一个 IRVariable 来表示这个变量本身，同时产生一个 IRValue 来记录这个变量的初值。此外，我们需要生成一条“复制”的中间代码，将 IRValue 的值赋给 IRVariable。

下面是不带 const 修饰符的变量定义的例子。我们一定会在 constExp 初值这里产生一个 IRValue：如果是全局变量，则直接把这个 IRValue 当作这个变量；如果是局部变量，则创建一个 IRVariable，然后添加复制代码。

exitVarDefBasic：

```
void SemanticAnalysis::exitVarDefBasic(CACTParser::VarDefBasicContext * ctx) {
    std::string name = ctx->Ident()->getText();
    IRValue * initVal;
    if (ctx->constExp() && ctx->constExp()->result) {
        initVal = ctx->constExp()->result;
        if (ctx->constExp()->dataType != currentDataType) {
            throw std::runtime_error("inital value data type not match.");
        }
    } else {
        initVal = irGen->newValue(currentDataType, "");
    }
    if (currentFunc) {
        IRVariable * irVar = irGen->newVar(name, currentDataType);
        irGen->assignBasic(currentDataType, irVar, initVal);
        ctx->thisSymbolInfo->setOp(irVar);
    } else {
        initVal->setName(name);
        initVal->setVariable(true);
        ctx->thisSymbolInfo->setOp(initVal);
    }
}
```

这里判断当前是否在函数内（局部变量）或全局（全局变量），并分别生成 IRVariable 或 IRValue。

局部变量赋初值时调用 irGen->assignBasic 生成复制 IRCode。

assignBasic：

```
void IRGenerator::assignBasic(DataType datatype, IROperand * d, IROperand * s) {
    switch (datatype) {
        case INT:
        case BOOL:
            addCode(new IR_w_Copy(d, s));
            break;
        case FLOAT:
            addCode(new IR_f_Copy(d, s));
            break;
        case DOUBLE:
            addCode(new IR_d_Copy(d, s));
            break;
        default:
            break;
    }
}
```

根据数据类型，生成不同类型的复制 IRCode（如 IR_w_Copy、IR_f_Copy、IR_d_Copy）。

函数的调用、参数与返回值的传递

RISC-V 中，函数参数与返回值都是通过寄存器传递的，但在我们的设计中，变量只与内存地址有关系。为此，我们设计了下面五种中间代码：

- PARAM：调用者传递变量作为参数（将变量的值放入寄存器）
- GET_PARAM：被调用者接收参数存入变量（将寄存器的值放入变量）
- RETURN：被调用者传递变量作为返回值（将变量的值放入寄存器）
- GET_RETURN：调用者接收返回值存入变量（将寄存器的值放入变量）
- CALL：调用某个函数

参数传递相关：

```
#define DECLARE_PARAM_CLASS(name, type) \
class IR_##name : public IRCode { \
public: \
    IR_##name(IROperand * operand); \
    virtual void print(); \
    void genTargetCode(TargetCodeList * t); \
};

DECLARE_PARAM_CLASS(para_w, PARAM_W)
DECLARE_PARAM_CLASS(para_f, PARAM_F)
DECLARE_PARAM_CLASS(para_d, PARAM_D)
```

这会生成 IR_para_w, IR_para_f, IR_para_d 三个类，分别对应整型、浮点型、双精度型参数传递。

参数接收相关：

```
#define DECLARE_GET_PARAM_CLASS(name, type) \
class IR_##name : public IRCode { \
public: \
    IR_##name(IROperand * operand); \
    virtual void print(); \
    void genTargetCode(TargetCodeList * t); \
};

DECLARE_GET_PARAM_CLASS(para_w_get, GET_PARAM_W)
DECLARE_GET_PARAM_CLASS(para_f_get, GET_PARAM_F)
DECLARE_GET_PARAM_CLASS(para_d_get, GET_PARAM_D)
```

这会生成 IR_para_w_get, IR_para_f_get, IR_para_d_get 三个类，分别对应参数接收。

函数调用相关：

```
class IR_call : public IRCode {
public:
    IR_call(IROperand * new_arg1);
    virtual void print();
    void genTargetCode(TargetCodeList * t);
};
```

返回值相关：

```
#define DECLARE_RETURN_CLASS(name, type) \
class IR_##name : public IRCode { \
public: \
    IR_##name(IROperand * operand); \
    virtual void print(); \
    void genTargetCode(TargetCodeList * t); \
};

DECLARE_RETURN_CLASS(w_return, RETURN_W)
DECLARE_RETURN_CLASS(f_return, RETURN_F)
DECLARE_RETURN_CLASS(d_return, RETURN_D)
```

这会生成 IR_w_return, IR_f_return, IR_d_return 三个类，分别对应不同类型的返回值。

返回值接收相关：

```
#define DECLARE_RETURN_GET_CLASS(name, type) \
class IR_##name : public IRCode { \
public: \
    IR_##name(IROperand * operand); \
    virtual void print(); \
    void genTargetCode(TargetCodeList * t); \
};

DECLARE_RETURN_GET_CLASS(w_return_get, GET_RETURN_W)
DECLARE_RETURN_GET_CLASS(f_return_get, GET_RETURN_F)
DECLARE_RETURN_GET_CLASS(d_return_get, GET_RETURN_D)
```

这会生成 IR_w_return_get, IR_f_return_get, IR_d_return_get 三个类。

实现文件举例：

```
IR_para_w::IR_para_w(IROperand * operand) : IRCode(PARAM_W, nullptr, operand, nullptr)
void IR_para_w::print() { ... }
void IR_para_w::genTargetCode(TargetCodeList * t) { ... }
```

实现文件包括构造、打印和目标代码生成。

整数、浮点数的运算

语法分析器已经帮我们把复杂的表达式拆解成一个一个的二元表达式对。我们的实现思路是：

- 对常量，创建一个 IRValue 作为其输出结果；
- 对运算，创建一个 IRVariable 临时变量作为其输出结果，插入三地址代码对子节点的结果进行运算；
- 对赋值，将右边表达式的输出结果赋给左边表达式，这里不会添加新的临时变量。

```
void SemanticAnalysis::exitAddExpAddExp(CACTParser::AddExpAddExpContext * ctx) {
    ctx->isArray = ctx->addExp()->isArray;
    ctx->arraySize = ctx->addExp()->arraySize;
    ctx->dataType = ctx->addExp()->dataType;
    ctx->result = irGen->newTemp(ctx->dataType);    // 创建临时变量

    std::string add_op = ctx->addOp()->getText();
    if (add_op == "+") {
        if (ctx->dataType == INT) {
            irGen->addCode(new IR_int_add(ctx->result, ctx->addExp()->result, ctx->mulExp()->result));
        } else if (ctx->dataType == FLOAT) {
            irGen->addCode(new IR_float_add(ctx->result, ctx->addExp()->result, ctx->mulExp()->result));
        } else if (ctx->dataType == DOUBLE) {
            irGen->addCode(new IR_double_add(ctx->result, ctx->addExp()->result, ctx->mulExp()->result))
        }
    } else if (add_op == "-") {
        if (ctx->dataType == INT) {
            irGen->addCode(new IR_int_sub(ctx->result, ctx->addExp()->result, ctx->mulExp()->result));
        } else if (ctx->dataType == FLOAT) {
            irGen->addCode(new IR_float_sub(ctx->result, ctx->addExp()->result, ctx->mulExp()->result));
        } else if (ctx->dataType == DOUBLE) {
            irGen->addCode(new IR_double_sub(ctx->result, ctx->addExp()->result, ctx->mulExp()->result))
        }
    }
}
```

这里根据表达式类型创建临时变量（IRVariable），并插入对应的 IRCode（如 IR_int_add、IR_float_add）。

```
IRVariable * IRGenerator::newTemp(DataType dataType) {
    std::string name = ".Temp" + std::to_string(tempCount++);
    auto * temp = new IRVariable(name, dataType);
    if (currentIRFunc) {
        currentIRFunc->localVariables.push_back(temp);
    }
    return temp;
}

void IRGenerator::addCode(IRCode * newCode) {
    if (currentIRFunc)
        currentIRFunc->codes.push_back(newCode);
}
```

newTemp 用于创建临时变量（IRVariable）。
addCode 用于将新生成的 IRCode 插入当前函数的 IRCode 列表。

向量的运算

向量运算是 CACT 的独有特性。对于向量运算基本上有两种办法：
向量直接展开：对于我们已有的设计是比较困难的：需要考虑如何让树状的代码产生多遍，如何传递下标，如何分配使用临时变量等等。
使用循环：问题则简单了许多。循环只需要在开头和结尾各添加几条语句，传递一个固定的下标变量即可，具体过程如下：

- 在处理赋值语句的左值时，如果变量是数组且没有下标，则认为是向量运算。此时，创建下标临时变量 index，值设为数组最后一个元素的下标；创建循环开始标号；创建一条语句，下标变量减少一个元素的大小。
- 在处理右值时（变量），断言其是数组且没有下标，采用 index 作为下标，创建语句 将其对应值读取到临时变量，以普通变量继续参与运算。
- 在赋值语句结束时：创建语句，将等号右侧表达式的输出变量写回到的目标数组的

index 下标；创建语句，如果下标大于零则跳回循环开始。

核心逻辑

```
void IRGenerator::startArrOp(DataType datatype, int len) {
    int cellSize = SizeOfDataType(datatype);
    auto * ir_cs = newInt(cellSize);
    auto * ir_len = newInt(len);
    arrRepeatVar = newTemp(INT);
    arrRepeatLabel = newLabel();
    addCode(new IR_int_mul(arrRepeatVar, ir_cs, ir_len));
    addCode(new IR_lab(arrRepeatLabel));
    addCode(new IR_int_sub(arrRepeatVar, arrRepeatVar, ir_cs));
}
```

该函数用于初始化数组循环，创建下标变量和循环标签，并插入相关 IRCode。

```
void IRGenerator::endArrOp() {
    if (arrRepeatVar) {
        addCode(new IR_greater_than_zero_goto_if(arrRepeatLabel, arrRepeatVar));
        arrRepeatLabel = nullptr;
        arrRepeatVar = nullptr;
    }
}
```

该函数用于结束数组循环，插入条件跳转 IRCode，实现循环控制。

```
void IRGenerator::assignArray(DataType datatype, int len, IROperand * d, IROperand * s) {
    startArrOp(datatype, len);
    IROperand * getA = getArrRepeatVar();
    IROperand * wzh = newTemp(datatype);
    switch (datatype) {
        case INT:
            addCode(new IR_w_indexed_copyfrom(wzh, s, getA));
            addCode(new IR_w_indexed_copy(d, wzh, getA));
            endArrOp();
            return;
        case FLOAT:
            addCode(new IR_f_indexed_copyfrom(wzh, s, getA));
            addCode(new IR_f_indexed_copy(wzh, wzh, getA));
            endArrOp();
            return;
        case DOUBLE:
            addCode(new IR_d_indexed_copyfrom(wzh, s, getA));
            addCode(new IR_d_indexed_copy(d, wzh, getA));
            endArrOp();
            return;
        default:
            endArrOp();
            break;
    }
}
```

该函数用于实现数组赋值，利用循环和下标变量，插入数组元素的复制 IRCode。

if-else、while 控制流

控制流的核心逻辑在 enterStmtCtrlIf、enterStmtCtrlWhile，以及 IR_lab、IR_jump 和条件跳转相关类。通过 IRLabel 和 IR_jump、条件跳转 IRCode 实现控制流。

```
void SemanticAnalysis::enterStmtCtrlIf(CACTParser::StmtCtrlIfContext * ctx) {
    irGen->addCode(ctx->codeBefore);
    if (!ctx->flowNext) {
        ctx->flowEnd = irGen->newLabel();
        ctx->flowNext = ctx->flowEnd;
    }
    ctx->cond()->flowTrue = irGen->newLabel();
    ctx->cond()->flowFalse = ctx->flowNext;
    ctx->stmt()->codeBefore.push_back(new IR_lab(ctx->cond()->flowTrue));
    ctx->stmt()->flowNext = ctx->flowNext;
}
```

这里通过生成 IRLabel（标签）和插入 IR_lab（标签指令）实现 if 控制流。

```
void SemanticAnalysis::enterStmtCtrlWhile(CACTParser::StmtCtrlWhileContext * ctx) {
    irGen->addCode(ctx->codeBefore);
    IRLabel * flowBegin = irGen->newLabel();
    irGen->addCode(new IR_lab(flowBegin));
    if (!ctx->flowNext) {
        ctx->flowEnd = irGen->newLabel();
        ctx->flowNext = ctx->flowEnd;
    }
    ctx->cond()->flowTrue = irGen->newLabel();
    ctx->cond()->flowFalse = ctx->flowNext;
    ctx->stmt()->codeBefore.push_back(new IR_lab(ctx->cond()->flowTrue));
    ctx->stmt()->flowNext = flowBegin;
    irGen->enterLoop(flowBegin, ctx->flowNext);
}
```

这里通过生成循环开始/结束标签和插入 IR_lab 实现 while 控制流。

IRCode.h 的条件跳转相关类

```
#define DECLARE_CONDITIONAL_JUMP_CLASS(name, op_type) \
class IR_##name : public IRCode { \
public: \
    IR_##name(IROperand * result, IROperand * arg1, IROperand * arg2); \
    virtual void print(); \
    void genTargetCode(TargetCodeList * t); \
};

DECLARE_CONDITIONAL_JUMP_CLASS(w_equal_goto_if, IF_EQUAL_GOTO_W)
DECLARE_CONDITIONAL_JUMP_CLASS(f_equal_goto_if, IF_EQUAL_GOTO_F)
DECLARE_CONDITIONAL_JUMP_CLASS(d_equal_goto_if, IF_EQUAL_GOTO_D)
DECLARE_CONDITIONAL_JUMP_CLASS(w_greater_than_goto_if, IF_GREATER_THAN_GOTO_W)
// ... 还有更多条件跳转相关类
```

这些类用于实现各种条件跳转 IRCode（如 if、while 的条件判断和跳转）。

IR_jump 和 IR_lab

```
class IR_lab : public IRCode {
public:
    IR_lab(IROperand * new_arg1);
    virtual void print();
    void genTargetCode(TargetCodeList * t);
};

class IR_jump : public IRCode {
public:
    IR_jump(IROperand * new_result);
    virtual void print();
    void genTargetCode(TargetCodeList * t);
};
```

用于实现无条件跳转和标签。

生成目标代码

全局变量的分配

IRValue::alloc 生成全局变量目标代码：

```
void IRValue::alloc(TargetCodeList* t) override {
    switch (dataType) {
        case BOOL:
        case INT:
        case FLOAT:
        case DOUBLE:
        default:
            // 这里会生成 .data/.rodata/.text 等段的汇编代码
    }
}
```

该函数负责为全局变量生成 RISC-V 汇编的段声明、对齐、类型、大小等指令。

局部变量的分配

IRFunction::targetGen 分配局部变量栈空间：

```
void IRFunction::generatetg(TargetCodeList * t) {
    int stack_off = -16;

    for (auto var : localVariables) {
        int align = var->getAlign();

        stack_off -= var->getSize();
        stack_off &= ~((1 << align) - 1);

        var->setMemOff(stack_off);
    }

    stack_off >>= 4;
    stack_off <<= 4;

    frameSize = -stack_off;

    t->add(std::string("\t.text"));
    t->add(std::string("\t.align\t1"));
    t->add(std::string("\t.globl\t") + functionName);
    t->add(std::string("\t.type\t") + functionName + std::string(", @function"));
    t->add(functionName + std::string(":"));

    t->add(std::string("\taddi\tsp, sp, ") + std::to_string(-frameSize));
    t->add(std::string("\tsd\tra, ") + std::to_string(frameSize - 8) + std::string("(sp)"));
    t->add(std::string("\tsd\ts0, ") + std::to_string(frameSize - 16) + std::string("(sp)"));
    t->add(std::string("\taddi\ts0, sp, ") + std::to_string(frameSize));

    for (auto oneCode : codes) {
        oneCode->genTargetCode(t);
    }

    t->add(std::string("\tld\tra, ") + std::to_string(frameSize - 8) + std::string("(sp)"));
    t->add(std::string("\tld\ts0, ") + std::to_string(frameSize - 16) + std::string("(sp)"));
    t->add(std::string("\taddi\tsp, sp, ") + std::to_string(frameSize));
    t->add(std::string("\tjr\tra"));
}
```

该函数负责为每个局部变量分配栈空间，并生成函数头、尾的汇编代码。

加法指令生成

IRAddInt::genTargetCode 生成加法指令：

```
void IR_int_add::genTargetCode(TargetCodeList * t) {
    get_a1()->loadTo(t, "t5");
    get_a2()->loadTo(t, "t6");
    t->add(std::string("\tadd\tt4, t5, t6"));
    get_ans()->storeFrom(t, "t4");
}
```

该函数负责将三地址加法指令翻译为 RISC-V 汇编加法指令。
这种“随用随取”的寄存器分配方式，直接将变量加载到临时寄存器，运算后再存回，简化了寄存器管理。

三、总结

实验结果总结

对本次实验我们组所输出的 RISC-V 64GC 汇编代码使用 riscv64-linux-gnu-gcc 进行编译链接，在模拟器中运行得到的可执行文件，得到的大部分结果与期望一致，基本实现了了从 CACT 到 RISC-V 64GC 的生成。本次实验结果对我们而言也是一个遗憾，因为时间限制和个人能力的不足，我们最终没能实现 pass 优化，同时有四个测试样例没能通过，相信假以时日对编译原理进行进一步深入学习后我们可以进一步完善我们的代码最终完全完成实验。

分成员总结

吴修齐：

在本次编译实验中，我主要负责了中间代码生成的核心构建和目标代码的生成，设计并实现了三地址代码的结构及操作数体系，完成了语法制导翻译的核心逻辑，将语法分析树有效转化为中间表示。在此基础上，我对三地址代码到汇编代码的转换过程进行了设计和维护，并最终成功实现了目标代码的生成模块，为整个编译器后端奠定了坚实基础。这次实验让我深刻理解了编译器后端工作的复杂性与系统性，锻炼了我的整体思维，对编译原理的核心环节有了更透彻的认识，也提升了解决复杂工程问题的能力。

吴喆：

我在本次实验中主要负责协助目标代码生成，同时对语法制导翻译过程进行优化。我维护了语法制导翻译过程中向量（数组）的整数和浮点数运算处理，确保运算指令能正确、高效地生成，解决了数据类型处理的关键细节。同时，我积极参与了目标代码生成的实现，并在实验后期承担了实验报告的主要撰写工作，负责整理、描述实验原理和结果。通过实验，我深入理解了编译器在处理不同数据类型和复杂数据结构时的原理。协助目标代码生成让我对指令选择和寄存器分配有了更直观的认识，加深了我对后端代码生成的理解。

卢柯圳：

我的工作重点在于目标代码生成的维护和优化。我主要负责优化了变量从内存加载到寄存器的机制，通过改进寄存器分配策略或访存指令的选择，提升生成代码的执行效率。同时，我也参与了目标代码生成环节的实现，并在后期负责了实验报告的部分撰写工作。编译后端的编写和中间代码生成过程中间涉及大量的参数传递和函数调用，非常容易遗漏，这使得 debug 的过程十分冗杂，但是在这一过程中也显著提升了我的 debug 能力，同时对计算机体系结构的存储层次和生成汇编代码的原理有了进一步的认识。