编译原理研讨课CACT实验PR003报告

成员组成

第16小组：

彭睿思 2020K8009915018

严哲虞 2020K8009907030

游昆霖 2020K8009926006

赵若雯 2020K8009929031

任务说明

PR001和PR002已经完成了对.cact程序的词法分析、语法分析和语义分析，在本次实验中我们在前两个实验的基础上完成：

1. 中间代码的设计和生成
2. 从中间代码生成RISCV汇编代码
3. 代码的优化

实验过程

设计思路

1. **无优化中间代码生成**
   - 语法文件（cact.g4）：为了方便在语义分析中生成中间代码，需要对语法文件进行修改，添加部分节点属性以传递中间代码相关信息（例如初始值字符串的拼接，break和continue语句对应label的传递、cond条件短路的label传递等），同时也添加了空节点以执行中间代码生成的语义动作（如lab和go节点）
   - 类型定义（cact_type.h）：添加了中间代码运算类型（IR_op_t）、语句类型（IR_code_t）和操作对象类型（IR_temp_t）的定义。考虑到原始文件中的变量名可能出现在不同作用域同名，为便于变量的区分和汇编的识别，统一对全局变量、局部变量和临时变量采用新的命名方式，即前缀加上序号。
   - 符号表（SymbolTable.h）：为变量和函数形参的符号表项中额外记录了中间代码对象名，从而可以在语义分析中直接根据变量节点使用其对象。
   - 语义分析（SemanticAnalysis.h/cpp）：在类中管理label、全局/局部/临时变量的生成以保证不重复。在遍历语法分析树时，根据相应的语义动作生成中间代码语句。需要考虑的问题主要有：
     - 控制流的跳转：

       对于if-else语句和while循环，需要借助g4文件添加的lab和go节点针对具体结构来生成标签和跳转语句；对于break和continue，可以通过逐级传递其跳转位置来获得跳转的目标位置；
     - 条件短路的处理：

       由于条件短路要求在计算部分条件即可确定cond真假时，结束条件判断；我们采用递归传递true_label和false_label进行实现。在lAndExp和lOrExp时，根据其逻辑传递标签，在eqExp和relExp中，根据对应的操作生成条件跳转的中间语句。
     - 变量的分配、初始化与使用：

       变量分配可以便于在生成汇编阶段确定函数栈帧的大小；初始化依赖对初始值字符串的拼接，考虑到多维数组可能出现部分初始化的情形，对于空白位置使用占位符填充；中间代码生成中变量的使用可以根据变量表中记录的中间代码对象IR_name实现。
     - 迭代计算的处理：

       在语法分析树的相应节点，会根据其操作符获得对应的中间代码运算，同时结果使用临时变量存储，以供上级表达式使用。在栈式分配中，临时变量同样会开辟相应的空间进行存储。
     - 函数的定义和调用：

       在函数定义部分，会将每个形参加入变量表，并为其分配对应的中间代码对象名IR_name，便于在汇编生成时计算栈帧和后续使用，上述动作都通过IR_FUNC_BEGIN实现。在函数调用部分，需要通过IR_PARAM传递参数，并通过IR_CALL最终调用函数，如果函数有返回值，则IR_CALL语句的arg1会设置为存储返回值的变量名字。
     - 子数组和数组元素的处理：

对于左值含中括号的情形，主要基于基址和偏移的计算实现。因此，该对象通过两个变量复合实现，一个变量记录基址，一个变量计算字节偏移；且两个变量根据该对象为数组还是元素使用不同的中缀进行连接。

- 关于中间代码的详细设计

  我们将一条中间代码定义为如下结构体：

```
typedef struct{
    IR_op_t IRop;
    cact_basety_t basety;
    std::string result;
    std::string arg1;
    std::string arg2;
}IR_code_t;
```

IRop表示该指令的操作码，arg1，arg2和result分别表示两个操作数和计算得到的结果的名字，basety表示结果的类型，方便后续转换成对应类型的RISCV汇编指令。

有关操作码类型IR_op_t的定义，首先考虑源代码中的表达式，生成对应中间代码的操作码需要与表达式操作类型一致，因此可以定义出IR_ASSIGN、IR_ADD~IR_BGE操作码。其次考虑函数的定义、调用和传参过程，在中间代码中采用IR_FUNC_BEGIN和IR_FUNC_END标记所定义的函数，函数被定义在两者之间。以IR_PARAM为操作码的语句标记了传入函数的形参和形参类型，IR_CALL表示发起调用的动作，以IR_RETURN为操作码的语句包含了函数的返回值和返回值类型。最后还要考虑变量的声明，在生成的中间代码中，我们用IR_G_ALLOC和IR_L_ALLOC分别用来表示源代码中对全局变量和局部变量的声明。完整的IR_op_t结构定义如下：

```
typedef enum{
    IR_UNKNOWN=0,
    IR_LABEL,
    IR_FUNC_BEGIN,
    IR_FUNC_END,
    IR_PARAM,
    IR_CALL,
    IR_RETURN,
    IR_ASSIGN,
    IR_ADD,
    IR_SUB,
    IR_MUL,
    IR_DIV,
    IR_SLL,
    IR_SRA,
    IR_MOD,
    IR_AND,
    IR_OR,
    IR_NEG,
    IR_NOT,
    IR_BEQ,
    IR_BNE,
    IR_BLT,
    IR_BGT,
    IR_BLE,
    IR_BGE,
    IR_J,
    IR_G_ALLOC,     //全局变量声明
    IR_L_ALLOC,     //局部变量声明
}IR_op_t;
```

在生成中间代码时，需要用符号标记操作数的属性，例如操作数是否为立即数或操作数是否为全局变量。在我们的中间代码设计中，使用不同的前缀来区别操作数的各种属性，各个前缀的定义如下：

```
#ifdef IR_gen
#define TEMP_PREFIX         '%'
#define ADDR_INFIX          '>'     //表示子数组相对变量的偏移
#define ITEM_INFIX          '<'     //表示子元素相对变量的偏移
#define ARRAY_PLACEHOLDER   '$'
#define IMM_PREFIX          '#'
#define LABEL_PREFIX        'L'
#define GVAR_PREFIX         '@'
#define LVAR_PREFIX         '^'
#endif
```

2. **汇编代码生成**

汇编代码需要根据中间代码翻译得到相应的汇编代码。需要考虑的问题主要有：

- 浮点数、全局变量和局部数组初始值的处理：

  在生成汇编时，浮点数需要通过标号进行加载，因此在rodata段要设置对应标号和值。全局变量根据是否初始化可以分为bss段和data段，并需要将值写入对应位置；在局部数组初始化时，对于有初始值的数组，也通过统一标号进行加载，以减少反复的读取基址操作。上述值都写到Const_Stack中，并在最后统一输出到汇编文件。

- 栈帧的计算：

由于在函数入口就需要开辟栈帧，但是栈帧的大小在遍历该函数的所有中间代码才能计算得到，因此生成汇编需要扫描两遍。在第一遍扫描时会计算函数的栈帧，并将每个非全局变量在栈中的位置进行记录；在第二遍扫描时，会根据相应中间代码生成汇编语句，并利用变量在栈中的位置进行存取操作。

- 不同类型变量的处理：

  在生成汇编代码时，同一个中间代码操作可能根据类别得到不同的汇编指令，如浮点和整型等，需要通过中间代码的basety进一步确定选择的汇编指令。

- 子数组和数组元素的处理：

  在生成汇编时，如果操作对象为两个变量的复合，需要在生成汇编时加以特殊考虑。

3. **局部优化**

局部优化主要基于中间代码IR缓冲区和流图基本块完成。

基于IR缓冲区的优化方式：

- 控制流优化：针对龙书上提到的三种情形进行优化，通过遍历和修改IR缓冲区进行实现。需要计算Label的位置和使用位置，根据label前后的语句和使用位置的语句判断如何优化和删改。
- 代数化简和强度削弱：针对等价语句进行优化，同样基于遍历和修改IR缓冲区进行实现。主要考虑了乘除法操作对象为2的幂次情形，将对应语句修改为移位运算，以减少指令执行代价。

基于流图基本块的优化方式：

- 基本块的切分与合并：基于label的位置和使用位置，可以确定首指令及其前驱指令。根据首指令位置，可以将基本块进行顺序切分和保存，基于首指令的前驱指令，可以确定每个基本块的前驱和后继。将IR缓冲区的中间代码划分到不同基本块内。
- 死代码消除：删除后续不使用的死代码，基于对基本块的活跃变量分析实现。首先需要遍历基本块的IR缓冲区为每个基本块计算def和use集合，并反复遍历每个基本块计算live_in和live_out直至不变。在基本块内，基于live_out进行逆序遍历，并维护活跃集live，即可将后续不使用的语句进行删除。
- 公共子表达式优化：对重复计算的语句，使用之前的计算结果替代。该优化主要在基本块内完成，通过遍历基本块内的IR缓冲区实现。由于仅局限在基本块内进行计算，不需要基于到达-定值进行迭代计算，对于一条计算语句，只需要向前遍历找到相同计算的语句，并保证在这两条语句中间，计算的结果和操作对象不被重新定值即可。在完成语句替代后，同时管理replace_map，使得后续语句在应用该句结果时，可以直接应用之前计算的结果，从而允许完成多轮的公共子表达式替换。

基于汇编代码缓冲区的窥孔优化：

在汇编代码中进行简单的遍历，仿照中间代码划分基本块的方式，考虑一条load指令所在的基本块前有无与其对应的store指令，如有，则将该条load指令替换为move，以减少访存开销。

代码实现

中间代码生成

1. g4文件

在stmt的语法规则的控制流语句的定义中增加lab和go，用于产生中间代码里的标签和跳转语句：

```
stmt
    locals[
        cact_basety_t ret_type,
        //IR
        //长度非空时，enter时跳进到in_label，exit跳出到out_label
        std::string in_label,
        std::string out_label,
        //因为不知道break和continue层级，在每个stmt/block->stmt/block推导时传递
        std::string break_label,
        std::string continue_label,
    ]
    : lVal ASSIGN exp SEMICOLON
    | (exp)? SEMICOLON
    | block
    | IF LeftParen cond RightParen lab stmt (go lab ELSE stmt)? lab
    | lab WHILE LeftParen cond RightParen lab stmt go lab
    | (BREAK | CONTINUE | RETURN exp?) SEMICOLON
    ;
```

为了保证语法分析的正确性，lab和go的语法规则都是生成空串。此外，对两者定义in_label和out_label属性，在翻译成中间代码时会分别产生跳转语句和代码标签。以if控制流语句为例，对于if语句，条件判断为真则跳转到if后的stmt，因此需要在该stmt前添加标签lab；对于if-else语句，条件判断为真跳转到if后的stmt，为假时跳转到else后的stmt，因此需要在这两个stmt前都要加上lab标签。与此同时，在第一个stmt结束之后要跳转出if语句，所以还要加上go。while控制语句添加标签和跳转语句的原理也类似。.g4文件中lab和go的语法规则和属性值定义如下：

```
//空节点，用于label的跳入和跳出
lab
    locals[
        std::string in_label,
    ]
    :
    ;

go
    locals[
        std::string out_label,
    ]
    :
```

```
    ;
```

2. SemanticAnalysis.h

   在SemanticAnalysis.h头文件中添加对中间代码初始化，用IRC_array存储所有生成的中间代码，label_cnt用来标记代码标签的序号。在生成中间代码时，将变量和常量的定义分成三种：局部、全局和临时变量，分别由Lvar_array、Gvar_array和Temp_array来存储和管理。我们用前缀来区分变量属于哪一种变量：局部、全局和临时变量的命名前缀分别为`LVAR_PREFIX(%)`、`GVAR_PREFIX(^)`和`TEMP_PREFIX(@)`。结合前缀和该变量在array中存储的位置，可对变量进行命名，例如：%1代表序号为1的临时变量。

```
#ifdef IR_gen
std::vector <IR_code_t> IRC_array;
size_t label_cnt=0;

size_t len_from_type(cact_type_t type){
    if(type.arrdims.size()==0){
        return 1;
    }
    else{
        size_t product = 1;
        for(int dim: type.arrdims){
            product *= dim;
        }
        return product;
    }
}

std::vector<IR_temp_t> Temp_array;
std::string newTemp(cact_basety_t basety, bool is_const=false, size_t length=1){
    std::string temp_name = TEMP_PREFIX + std::to_string(Temp_array.size());
    Temp_array.push_back((IR_temp_t){.basety=basety,.is_const=is_const,.length=length});
    return temp_name;
}

std::vector<IR_temp_t> Gvar_array;
std::string newGvar(cact_basety_t basety, bool is_const=false, size_t length=1){
    std::string gvar_name = GVAR_PREFIX + std::to_string(Gvar_array.size());
    Gvar_array.push_back((IR_temp_t){.basety=basety,.is_const=is_const,.length=length});
    return gvar_name;
}

std::vector<IR_temp_t> Lvar_array;
std::string newLvar(cact_basety_t basety, bool is_const=false, size_t length=1){
    std::string lvar_name = LVAR_PREFIX + std::to_string(Lvar_array.size());
    Lvar_array.push_back((IR_temp_t){.basety=basety,.is_const=is_const,.length=length});
    return lvar_name;
}

std::string newLabel(){
    std::string label_name = LABEL_PREFIX + std::to_string(label_cnt);
    label_cnt++;
    return label_name;
}

//缺省参数，可以从左到右写参数，省略后面的部分
void addIRC(IR_op_t IRop,cact_basety_t basety=BTY_UNKNOWN,std::string result="",std::string
arg1="",std::string arg2=""){
    IRC_array.push_back((IR_code_t){IRop,basety,result,arg1,arg2});
}
```

   如上代码所示，我们用newTemp、newLvar和newGvar函数将变量保存在其对应类型的数组中。new_label函数表示生成一个新的代码标签，并将表示标签计数的变量label_cnt加1。addIRC函数用于生成一条中间代码指令，并把这条新生成的指令放入IRC_array中。

   len_from_type函数用来计算变量的长度，单个变量（type.arrdims.size()=0）如`a`，`b`的长度为1。若变量为形如`c[2][2]`的多维数组，则需要计算整个数组的长度，即元素个数。该函数计算出的变量长度也需要记录在Temp_array/Lvar_array/Gvar_array中，以供后续在生成汇编代码时计算函数栈帧大小。

3. SemanticAnalysis.cpp

   - 变量/常量的定义
     - exitConst_Var_Def函数

       在离开ConstDef或VarDef节点时，首先判断所定义的变量是否为全局变量，如果是，则生成对应的中间代码中IRop为G_ALLOC，否则为L_ALLOC。利用变量类型和len_from_type计算变量长度后，使用newGvar或newLvar将该变量添加到array中存储，并得到变量名。将变量名写入符号表中，以便在后续的中间代码生成中直接使用。相关代码如下：

```
bool is_global = cur_scope == symbol_table.root_scope;
IR_op_t IRop;
size_t len;
std::string IR_name;
if(is_global){
    IRop = IR_G_ALLOC;
    len = len_from_type(ctx->type);
```

```
    IR_name = newGvar(ctx->basety, is_const, len);
}
else{
    IRop = IR_L_ALLOC;
    len = len_from_type(ctx->type);
    IR_name = newLvar(ctx->basety, is_const, len);
}

symbol_table.var_table[(name_scope_t){name,cur_scope}] = (var_symbol_item_t){.type=ctx-
>type,.IR_name=IR_name};
```

如果在检查constDef或varDef节点时发现了constInitVal子节点，说明在声明该变量时还同时对它进行了初始化，因此在获得初始值后再加上立即数前缀IMM_PREFIX便可生成一条声明该变量的中间代码：

```
std::string initval = IMM_PREFIX + ctx->constInitVal()->value_list;
addIRC( IRop, ctx->basety,IR_name,initval);
```

- exitConstInitVal函数

  该函数在上一个实验中实现了对多维数组列举和嵌套初始化定义检查的支持。在生成多维数组初始化的中间代码时，如果在定义中出现元素的缺失，需要用占位符ARRAY_PLACEHOLDER(
)来填充。例如比如嵌套定义$a[3][2] = ,1$，那么对于最里面花括号，就要填充成2个元素，对于外层的花括号，要由三个拼接，并且每个都
  ,, 1,,。我们用hold_len表示每一个嵌套层需被填充的长度，len表示实际定义的元素数量，如果hold_len>len，则需要增加占位符$。

- 函数的定义
  - enterFuncDef和exitFuncDef函数

    进入FuncDef节点后，此时正在进行一个函数的定义，因此需要生成一条IR_FUNC_BEGIN的中间代码。离开FuncDef节点说明函数的定义已经结束，需要生成一条IR_FUNC_END的中间代码

  - enterBlock函数

    如果block是函数在定义时的block，则需要在符号表中加入函数定义时用到的形参名字IR_name，方便后续生成中间代码中对变量名的直接使用。

    ```
    std::string IR_name = newLvar(basety);//使用缺省参数 isconst=false,len=1
    (*ctx->fparam_list_ptr)[i].IR_name = IR_name;
    var_symbol_item_t item = (var_symbol_item_t){.type=fparam.type,.IR_name=IR_name};
    ```

  - exitStmt_bcr函数

    在函数的定义中，最后一句可能是return返回语句，因此如果stmt_bcr的语法规则中生成了RETURN节点，会生成一条IRop为IR_RETURN的中间代码语句，并将ret_type和RETURN token后exp节点的result_name属性添加到该代码语句中。

- 函数的调用
  - exitUnaryExp函数

    `unaryExp : Ident LeftParen (funcRParams)? RightParen` 定义了函数调用的语法规则，所以在离开exitUnaryExp节点和检查传参过程中各变量的类型和数量是否与被调用函数的形参列表一致后，若检查无问题，则会为每一个参数生成IRop为IR_PARAM的中间代码，并将该参数的基本类型basety和变量名exp_name填入到代码中。

    完成IR_PARAM传参语句后，生成IR_CALL中间代码。此外还需要注意的是，如果函数的返回值不是void类型，我们还需要再分配一个临时变量来存储该函数调用的返回值：

    ```
    if(basety==BTY_VOID){
        addIRC( IR_CALL,
                BTY_VOID,
                func_name);
    }
    else{
        ctx->result_name = newTemp(basety);
        addIRC( IR_L_ALLOC,
                basety,
                ctx->result_name);
        addIRC( IR_CALL,
                basety,
                func_name,
                ctx->result_name);
    }
    ```

- 控制流语句的翻译

  .cact源码中的控制流语句以if/if-else/while为主，这里以while语句为例阐述生成相应中间代码的过程。

  - enterStmt_while函数

    根据语法规则 `stmt : lab WHILE LeftParen cond RightParen lab stmt go lab`，我们需要在中间代码中生成3个lab对应的代码标签，分别为start_label，block_start和end_label。如果cond为真，控制需要跳转到block_start的代码位置，如果为假，则会跳出while循环，控制转移到end_label的代码位置。因此 `ctx->cond()->true_label` 和 `ctx->cond()->false_label` 分别赋为block_start和end_label。此外，如果stmt执行完毕，需要返回while中的条件语句进行是否进行下一轮循环的判断，因此stmt后的go会被翻译成跳转到start_label处代码的跳转指令。如果stmt是一个break语句，则结束

while循环，stmt节点的break_label属性被赋为end_label；如果stmt是一个continue语句，则控制直接转换到while的条件语句处，stmt节点的continue_label属性被赋值为start_label

```
void SemanticAnalysis::enterStmt_while(CACTParser::Stmt_whileContext *ctx){
    #ifdef IR_gen
    std::string start_label = newLabel();
    std::string end_label = newLabel();
    std::string blk_start = newLabel();
    ctx->cond()->true_label = blk_start;
    ctx->cond()->false_label = end_label;
    ctx->lab()[0]->in_label = start_label;
    ctx->lab()[1]->in_label = blk_start;
    ctx->lab()[2]->in_label = end_label;
    ctx->go()->out_label = start_label;
    //沿子节点传递
    ctx->stmt()->break_label = end_label;
    ctx->stmt()->continue_label = start_label;
    #endif
}
```

- enterLAndExp/enterLOrExp/enterRelExp函数

  对于控制流中的条件语句cond，其条件表达式的操作符可能为&&、||或==，分别对应着LAndExp、LOrExp和RelExp的情况，这里以enterLAndExp函数为例说明代码翻译的过程：

  `lAndExp : lAndExp1 AND lab eqExp` 语法规则定义了&&表达式。对于 `lAndExp1`，如果表达式为真，则控制需要跳转到eqExp前的lab代码标签继续进行条件判断，若eqExp条件判断也为真，则整个 `lAndExp` 表达式为真，控制流跳转到lAndExp的true_label处，即ctx->true_label。若eqExp条件判断为假，整个 `lAndExp` 表达式为假，控制流跳转到lAndExp的false_label处，即ctx->false_label。如果lAndExp1表达式为假，则整个 `lAndExp` 表达式为假，控制跳转到lAndExp的false_label处，即ctx->false_label。代码实现逻辑如下：

```
void SemanticAnalysis::enterLAndExp(CACTParser::LAndExpContext *ctx){
    #ifdef IR_gen
    if(ctx->lAndExp()!=nullptr){ //lAndExp AND eqExp
        //部分为true不需要特别处理，继续向下即可，不必和OR一样加额外标签
        std::string and_label = newLabel();
        ctx->lab()->in_label = and_label;
        ctx->lAndExp()->true_label = and_label;
        ctx->lAndExp()->false_label = ctx->false_label;
        ctx->eqExp()->true_label = ctx->true_label;
        ctx->eqExp()->false_label = ctx->false_label;
        ctx->eqExp()->has_label = true;
    }
    else{//eqExp
        ctx->eqExp()->true_label = ctx->true_label;
        ctx->eqExp()->false_label = ctx->false_label;
        ctx->eqExp()->has_label = true;
    }
    #endif
}
```

- 数组左值和子数组地址的获得
  - exitLVal函数

    1）如果表达式的操作类型为OP_ITEM，则要求获得数组的左值，所以需要根据数组的维度、给定的index和元素类型来计算目标位置相对数组基地址的偏移量。具体代码逻辑如下所示：

```
#ifdef IR_gen
//单位元素的字节偏移，语义分析不出现PTR，只有IR传参的时候可能使用
int base_size;
switch(iter_type.basety){
    case BTY_BOOL:
        base_size = 1; break;
    case BTY_INT: case BTY_FLOAT:
        base_size = 4; break;
    case BTY_DOUBLE:
        base_size = 8; break;
    default:
        break;
}

//为了充分利用公共子表达式对维度的计算，每个计算结果都用新的变量名存储，防止result和arg相同
std::string index_offset; //当前维度的字节偏移
std::string old_offset;//之前累加
std::string byte_offset;//本轮后累加
for(int i=0;i<layer;i++){
    int subproduct = 1;
    for(int j=i+1;j<dim_size;j++){
        subproduct *= iter_type.arrdims[j];
```

```
    }
    std::string times = IMM_PREFIX + std::to_string(subproduct*base_size);
    index_offset = newTemp(BTY_INT);
    addIRC(IR_L_ALLOC,BTY_INT,index_offset);
    addIRC(IR_MUL,BTY_INT,index_offset,exp_list[i]->result_name,times);
    if(i==0){
        byte_offset = index_offset;
    }
    else{
        old_offset = byte_offset;
        byte_offset = newTemp(BTY_INT);
        addIRC(IR_L_ALLOC,BTY_INT,byte_offset);
        addIRC(IR_ADD,BTY_INT,byte_offset,old_offset,index_offset);
    }
}


#endif
```

2）如果表达式的操作类型为OP_ARRAY，则要求获得子数组的地址，子数组的地址相对于数组基地址的偏移的计算与上述情况相同，但在生成中间代码时要用不同的前缀区分这两种情况：

```
#ifdef IR_gen
char infix = (op==OP_ARRAY) ? ADDR_INFIX : ITEM_INFIX;
ctx->result_name = IR_name + infix + byte_offset;
#endif
```

4. cact_types.h

为了简化代码和方便使用，在`cact_types.h`中`TypeUtils`类增添定义了表达式操作类型cact_op_t与中间代码操作码IR_op_t以及IR_op_t与生成的中间代码串的转换：

```
#ifdef IR_gen
std::map <cact_op_t, IR_op_t> op_to_IRop{
    {OP_NEG, IR_NEG},
    {OP_NOT, IR_NOT},
    {OP_ADD, IR_ADD},
    {OP_SUB, IR_SUB},
    {OP_MUL, IR_MUL},
    {OP_DIV, IR_DIV},
    {OP_MOD, IR_MOD},
    {OP_LEQ, IR_BLE},
    {OP_GEQ, IR_BGE},
    {OP_LT,  IR_BLT},
    {OP_GT,  IR_BGT},
    {OP_EQ,  IR_BEQ},
    {OP_NEQ, IR_BNE},
};

std::map <IR_op_t, std::string> IRop_to_str{
    {IR_LABEL,      "Label"},
    {IR_FUNC_BEGIN, "Func Begin"},
    {IR_FUNC_END,   "Func End"},
    {IR_PARAM,      "Param"},
    {IR_CALL,       "Call"},
    {IR_RETURN,     "Return"},
    {IR_ASSIGN,     "ASSIGN"},
    {IR_ADD,        "ADD"},
    {IR_SUB,        "SUB"},
    {IR_MUL,        "MUL"},
    {IR_DIV,        "DIV"},
    {IR_SLL,        "SLL"},
    {IR_SRA,        "SRA"},
    {IR_MOD,        "MOD"},
    {IR_AND,        "AND"},
    {IR_OR,         "OR"},
    {IR_NEG,        "NEG"},
    {IR_NOT,        "NOT"},
    {IR_BEQ,        "BEQ"},
    {IR_BNE,        "BNE"},
    {IR_BLT,        "BLT"},
    {IR_BGT,        "BGT"},
    {IR_BLE,        "BLE"},
    {IR_BGE,        "BGE"},
    {IR_J,          "J"},
    {IR_G_ALLOC,    "G_Alloc"},//全局变量声明
    {IR_L_ALLOC,    "L_Alloc"},//局部变量声明
};
#endif
```

汇编代码生成

在`RiscvGen.h`中定义一个`RiscvGen`类，用于处理由中间代码生成汇编代码的过程。main函数中会实例化一个`RiscvGen`类的对象，通过执行它的`Gen_All`成员函数来生成所有汇编指令并将其写入文件。

在`RiscvGen.cpp`中给出了`RiscvGen`类各成员函数的具体定义。

Gen_All

Gen_All函数先向ASM_array中添加文件名等开头信息，接着声明两个变量，rescan用于标记当前是第一次还是第二次扫描，`rescan_loc`，用于记录重新开始扫描的位置。

随后遍历存放中间代码的数组`ir_optim.IRC_array`，进行两次扫描。

在第一遍扫描时，即rescan是false时，对于全局变量、临时变量和函数的参数，都要计算它们所占的空间，并更新栈帧的大小。遇到IR_FUNC_END表示第一遍扫描完成，此时将rescan置为true并更新循环变量i，使得在下一个iteration中会从IR_FUNC_BEGIN开始进行第二遍扫描。

```
case IR_FUNC_END:
    //对栈帧向上对齐到16倍数处理
    frame_size = up_to_align(frame_size,16);
    //从Func_Begin开始重新扫描
    rescan = true;
    //考虑后续执行i++，跳回到前一个位置
    i = rescan_loc-1;
    break;
```

在第二遍扫描时，根据中间代码的类型调用对应的函数处理，生成中间代码。最后，把`ASM_array`中的汇编指令和`Const_Stack`中的常量部分写入文件。

stack_initval

stack_initval函数用于解析初始值字符串，根据基本类型生成相应的汇编指令，并将其写入`Const_Stack`。

首先处理初始值字符串，按逗号拆分成子串，并去除前缀。

```
    //将初始值字符串拆分
    std::vector<std::string> initval_array;
    //去除前缀
    initval_str = initval_str.substr(1);
    int loc; //第一个逗号位置
    for(int i=0;i<len;i++){
        loc = initval_str.find(',');
        if(i==len-1 && loc==-1){
            //允许最后一个没有，后缀
            initval_array.push_back(initval_str);
        }
        else{
            initval_array.push_back(initval_str.substr(0,loc));
            initval_str = initval_str.substr(loc+1);
        }
    }
```

然后遍历子串，根据基本类型`basety`的不同，生成相应的汇编指令。在此过程中，变量`zero_size`用于统计空白位置的大小，每次将其写入Const_Stack后清零。

var_to_reg

var_to_reg函数用于将变量从栈上转移到寄存器中。

首先根据变量名确定变量的类型。通过变量名的前缀，可以判断变量是否为全局变量、局部变量、临时变量和立即数。

```
    bool is_gvar = var_name[0] == GVAR_PREFIX;
    bool is_lvar = var_name[0] == LVAR_PREFIX;
    bool is_temp = var_name[0] == TEMP_PREFIX;
    bool is_imm = var_name[0] == IMM_PREFIX;
```

接着通查找变量名中是否存在`ADDR_INFIX`或`ITEM_INFIX`，它们标识了子数组/子元素相对相对变量的偏移。如果存在，就递归调用`var_to_reg`函数将临时变量转移到`s3`寄存器中。然后分别考虑全局变量、局部变量、临时变量、立即数的情况，生成对应的汇编指令并填入ASM_array。在此过程中只使用s3、s4寄存器计算地址，不会和其他冲突。

reg_to_var

该函数将值从寄存器转移到栈上。

先确定变量的类型，通过前缀判断变量是否为全局变量、局部变量、临时变量。然后在变量名中查找`ITEM_INFIX`，如果找到则将变量转移到s5寄存器中。接着分别考虑全局变量、局部变量、临时变量的情况，根据变量的类型用fsd、fsw、sw、sb等指令将值存入栈中。

Gen_FuncBegin

该函数在第二遍扫描到函数开头时被调用，用于生成函数开头必要的汇编指令。

它会先生成开头的信息，然后生成函数名对应的label，再根据第一遍扫描时统计的栈帧大小，生成开辟栈帧的指令。随后，它通过函数名在符号表中查找函数的参数列表，将函数形参从寄存器转移到栈上，当前只考虑a0-a7和fa0-fa7。对于整数参数（a0-a7），使用`s`寄存器和`lw`、`lb`指令；整数参数（a0-a7），使用`s`寄存器和`lw`、`lb`指令。最后检查函数参数的数量，如果整数参数数量`icnt`或浮点数参数数量`fcnt`超过等于8，则输出错误信息并退出程序。

Gen_FuncEnd

该函数在第二遍扫描到函数结尾时被调用，用于生成函数开头必要的汇编指令，以恢复寄存器和栈帧，并在函数结尾处进行返回操作。同时，该函数还定义了函数的size，以便后续进行链接和代码优化。

Gen_G_Alloc

该函数用于生成分配和初始化全局变量的汇编代码，会在第二遍扫描到全局变量时被调用。

首先，根据全局变量的名称获取其在全局变量数组中的索引`order`和堆栈标签`stack_label`。获取索引需要从变量名中去除前缀，并转为数字，这一过程通过函数`num_from_name`实现，使代码更简洁。

随后，根据全局变量的属性判断其所属的分区（`.rodata`、`.data`或`.bss`），并将相应的信息添加到`Const_Stack`中。如果全局变量需要初始化，则调用`stack_initval`函数来解析初始值字符串并生成相应的汇编代码。否则，添加指令`.zero`来分配适当大小的空间。

Gen_L_Alloc

Gen_L_Alloc函数在第二遍扫描到局部变量时被调用，生成分配和初始化局部变量的汇编代码。

首先检查变量是否为局部变量且是否需要进行初始化。如果不是局部变量或不需要初始化，则直接返回。

随后调用`num_from_name`获取变量的索引`order`，再获取长度`len`和偏移量`offset`。然后在`.rodata`段声明常量，分别考虑len>1和len<=1两种情况，对数组均使用标号初始化，对浮点的单个常量也需要给出标号。

最后，生成将局部变量从标号转移到栈上的汇编代码。对于长度大于1的数组，使用寄存器`s2`存储标号的地址，并根据每次转移的字节数选择相应的指令进行转移，在转移8/4/2/1字节时分别采用不同的后缀d/w/h/b。每次采用能转移尽可能多字节的指令，避免不必要的多次转移。对于长度为1的变量，根据基本类型选择相应的指令将初始值存储到栈上。

局部优化

在IRoptim.h/cpp 基于中间代码进行了局部优化，在RiscvGen.h/cpp的asm_optim方法中基于汇编代码暂存区进行了窥孔优化

IRoptim.h提供了基本块和IR优化方法的类定义，同时在类中定义了公共变量，供成员函数及外部函数使用。

IRoptim.cpp提供了上述两个类中成员函数的具体实现，以下对主要的函数进行说明：

- IROptim::get_head_label 获取label的位置和使用位置

  遍历IR缓冲区，对标签语句，记录其位置；对跳转语句，将其位置记录到对应label的使用列表中；将FUNC_begin、FUNC_END下一条、CALL下一条作为首指令，且在head_map其前驱指令。

  遍历IR缓冲区后，对label进行遍历，对于被使用的label，将其记录为首指令，并在head_map中记录其前驱指令。

- IROptim::split/merge 基本块的切分和合并

  split方法调用get_head_label方法获取head_map，则head_map中的每个key是首指令的位置，对应的value是该首指令的前驱指令。根据首指令位置可以进行基本块切分，并将IR填到对应基本块中，记录其起止位置；根据首指令前驱指令的位置和每个基本块对应IR缓冲区的起止位置可以确定每个基本块的前驱和后继基本块，将指向这些基本块的指针记录到指针数组pre_list和next_list中。

  merge方法按照基本块的位置顺序提取其缓冲区内容进行了合并。

- IROptim::controlflow 控制流优化方法封装

  对龙书中三种控制流优化的情形进行了实现，并通过延迟修改的方式允许在单次循环中修改多个位置，减少循环次数。

  调用get_head_label方法得到每个label的位置及使用列表，然后对label进行遍历：考虑label处的指令，跳转到label的指令进行优化。如果需要进行修改，则设置flow_change为true以该次遍历结束后开始下一轮遍历，针对需要删除的指令，将位置记录在erase_set中，针对某位置需要插入的指令（可能为多条），将其位置及数组记录在insert_map中，并在每轮遍历结束后调用update_irc方法，根据erase_set和insert_map更新IR缓冲区。同时，如果将无条件跳转更新为条件跳转，需要在后面加上标签，以方便进一步的控制流优化

  以下例子展示了对控制流的优化：

  ```
      goto L1
  L1: goto L2
  L2: ifcond L3
  L4:
  ```

  第一次优化

  ```
      goto L2
  L1: ifcond L3
  L5: goto L4
  L2: ifcond L3
  L4:
  ```

  第二次优化

```
    ifcond L3
L6: goto L4
L1: ifcond L3
L5: goto L4
L2: ifcond L3
L4:
```

通过优化，虽然中间代码的数量可能有所增加，但实际执行的跳转次数会小于等于未优化情况。

- IROptim::algebrasimplify 代数化简和强度削弱方法封装

  对IR缓冲区进行遍历，对操作数为2的幂次立即数的乘除法代码，替换为移位运算。

- BasicBlock::init_def_use 获取基本块的def和use

  对基本块的IR缓冲区进行遍历，将语句使用且之前未定值的变量加入use集，将定值且之前未使用的变量加入def集。考虑到中间代码的语句对象可能是两个变量的复合，也可能是立即数等，通过get_var进行变量的提取。同时如果定值的对象是一个数组元素，则应当将数组加入定值表，将偏移加入use表。另外，由于要求加入def的变量未被使用，加入use的变量未被定值，使用add_use和add_def封装操作，仅在该变量未出现在def/use时才可加入use/def。

- BasicBlock::dead_code 删除基本块的死代码

  根据得到的基本块live_out，对基本块的IR缓冲区进行逆序遍历，并管理活跃集，记录当前IR后的活跃变量。如果该句dst不在活跃集中，则删除该IR；否则从活跃集删除dst，再加入src变量。对dst和src是两个变量复合的考虑同init_def_use方法。

- IROptim::deadcode 死代码优化方法封装

  首先调用每个基本块的init_def_use方法初始化其def和use集。根据活跃变量控制流方程迭代计算每个基本块的live_in和live_out。最后调用每个基本块的dead_code方法消除死代码。

- BasicBlock::dag 公共子表达式优化

  计算公共子表达式的过程仅针对计算语句，且要求src均不为两个变量的复合。在查找某个语句IRC能否进行公共子表达式优化时，会先尝试将src按照replace_map进行替换，然后向前查找，如果有语句对src定值，则查找结束，不可优化；如果有语句pre_IRC的运算、src均和该句一致，且result不覆盖src，则可将IRC改为将IRC.result定值为pre_IRC.result，并将其加入replace_map。

  需要注意的是，在前向搜索的过程中，需要记录每个语句的定值，需要考虑该定值覆盖pre_IRC定值的情形。同时，除非是成功发生替换的语句，其余含定值的语句需要从replace_map擦除该定值。

  针对上述注意点的例子可见过程思考，以下展示优化例子：

```
MUL | %2 | ^1 | #8
MUL | %3 | ^2 | #2
ADD | %4 | %2 | %3
MUL | %5 | ^1 | #8
MUL | %6 | ^2 | #2
ADD | %7 | %5 | %6
ADD | @1<%4 | @1<%7 | #1
```

  如果只对IR更改，不管理replace_map可以得到以下结果：

```
MUL     | %2 | ^1 | #8
MUL     | %3 | ^2 | #2
ADD     | %4 | %2 | %3
ASSIGN  | %5 | %2
ASSIGN  | %6 | %3
ADD     | %7 | %5 | %6
ADD     | @1<%4 | @1<%7 | #1
```

  通过管理replace_map，我们可以将进一步得到如下结果：

```
MUL     | %2 | ^1 | #8
MUL     | %3 | ^2 | #2
ADD     | %4 | %2 | %3
ASSIGN  | %5 | %2
ASSIGN  | %6 | %3
ASSIGN  | %7 | %4
ADD     | @1<%4 | @1<%4 | #1
```

  该结果可以利用死代码消除方法进一步得到：

```
MUL     | %2 | ^1 | #8
MUL     | %3 | ^2 | #2
ADD     | %4 | %2 | %3
ADD     | @1<%4 | @1<%4 | #1
```

  在matrix测试样例中，处理不同数组有某些相同下标时，该方法就可以很好的消除重复计算的部分。在语义分析部分也保证了每个维度计算的结果使用不同变量记录，以使其能充分利用公共子表达式优化。

- IROptim::dag 公共子表达式优化封装

  遍历每个基本块，对其进行公共子表达式优化。

RiscvGen.h/cpp 对load和store指令的窥孔优化

定义了get_vec方法，用于从一行汇编代码字符串中提取操作符和操作对象；blk_end方法用于在某汇编代码为基本块起始标志时终止向前查找。

在asm_optim方法中，将会对汇编代码缓冲区进行遍历，对于一条load指令，将会向前查找与其匹配的store指令，如果碰到某前序指令使blk_end()返回值为true，即搜索结束。查找与其匹配的store指令也结束。

这个过程中也同样需要管理def表，用于记录被修改的寄存器，以免错误用到寄存器已经被覆盖的值。

过程思考

遇到的问题和debug思路

- 访存报错解决

  对tim_sort测试时出现`User load segfault @ 0x0000000000000000`错误，应该是内存访问越界。通过print定义调试错误，

  ```
  if( iter < tail ){
      print_int(4);
      if( nums[iter] <= nums[iter + 1] ){
          print_int(5);
          while( iter < tail && nums[iter] <= nums[iter + 1] ){
              iter = iter + 1;
          }
      } else {
          print_int(6);
  ```

  出错位置只打印至4，因此猜测是数组访存错误。注意到，这里的nums是指针，而在其他样例中对数组的访问并未出现问题，猜测问题出现在关于指针的访问。结合出错地方应当在访问nums[0]元素，问题可能出现为传参时并未将地址正确传递到nums变量在栈上的对应位置，导致从栈上提取地址时提取到了0；或者对栈中值的使用不正确，导致最终访问到0地址。

  观察报错时的寄存器，发现用于存储元素地址（指针加上偏移）的寄存器s4的值是正确的，和objdump查看得到的.data段地址一致，因此可以判断错误发现在load操作，查看源码发现，在考虑指针情况最终的load操作时，将s4笔误写为了s3。

  经过上述修改，上述位置的错误消失，同样的user load segfault错误发生在调用binary_insert函数时，结合报错时的寄存器信息，a0为正确的全局地址，预期load地址的寄存器s4最后8位为fffffb27，即十进制的-1025+4，其中-1025为array的第一个元素，4为字节偏移。因此错误发生在对指针进行了重复的寻址操作。

  首先查看对栈该位置的写入和读出，发现值不同，因此可能错误发生在中途对该位置进行了修改。查看代码发现`sw  s7, -24(s6)`使用了相同的偏移。查看源码发现，在负责将寄存器值转移到栈上的函数reg_to_var中，遗漏考虑了局部变量为指针的情形，补充对该情况的考虑即可通过所有测试。

- 值传递和引用传递

  使用`for(auto blk: blk_list)`遍历基本块计算live_in和live_out时，发现尽管循环内修改了blk的公共变量，但在下一轮循环中，原先已更改值恢复原值。该错误是因为上述循环语句使用了值传递，更改均发生在临时变量，而不会改变原值。正确的写法应当为`for(auto &blk: blk_list)`，通过引用传递改变其真值。

- 死代码优化问题解决

  针对数组变量和结尾基本块的特殊处理。在进行死代码优化时，发现大量有效代码被删除导致结果出错，通过优化前后中间代码比对，需要进行以下特殊处理：

  - 对数组元素的定值：可以将记录offset的变量加入use，将记录基址的变量加入def，但在消除死代码时不能因为数组的定值就将其删除。
  - 对函数结尾基本块，需要将所有全局变量加入live_out，这是因为所有全局变量后续都可能使用，需要视为活跃，以免函数中的定值无效。

  同时，在计算每个基本块的live_in和live_out时，最初采用是先将其清空在按照控制流方程计算的方式，导致了错误。这是因为一个基本块的前驱同样有可能是自己，如果将live_in清空，就无法正确的计算得到live_out。考虑到迭代改变的过程不会让每个基本块的live_in和live_out失去变量，因此无需进行清空。

- 公共子表达式优化问题解决

  ```
  MUL %1 | ^1 | #4
  ASSIGN ^1 | #1
  MUL %3 | ^1 | #4
  ```

  在向前传播过程中看到src被定值则终止可以防止最后语句被优化为`ASSIGN %3 | %1`

  ```
  MUL %1 | ^1 | #4
  ADD %1 | %2 | #1
  MUL %3 | ^1 | #4
  ```

  通过def_set的管理，可以在最后替换的时候检查，避免最后语句被优化为`ASSIGN %3 | %1`

  ```
  MUL %1 | ^1 | #4
  MUL %3 | ^1 | #4
  ASSIGN %3 | #1
  ASSIGN %4 | %3
  ```

  通过replace表的管理，可以避免最后语句被修改为`ASSIGN %4 | %1`

优化结果分析

- 优化性能表

Method	Samples	time	kInsts	kCycles
GCC O0	fib_subseq	64	1471	2723

Method	Samples	time	kInsts	kCycles
	matrix	389	4593	16271
	tim_sort	167	4411	7044
O0	fib_subseq	73	1898	3097
	matrix	177	5562	7560
	tim_sort	195	5925	8190
O1	fib_subseq	66	1850	2781
	matrix	120	4438	5197
	tim_sort	176	5861	7428

> GCC生成的matrix汇编代码在评测时出现float accuray not met的问题，因此上述评测结果偏大。

由于O0采用栈式分配，我们的临时变量也都需要先存到栈上在进行使用，导致了大量的额外load和store操作，因此O0的性能弱于gcc的O0。

通过O1的局部优化，我们的代码性能相比O0有了显著提高，尤其是公共子表达式的优化在matrix测试样例中取得了显著成功，最终用时只有O0的2/3；

后续本组将继续完善窥孔优化和寄存器分配等策略，以求减少无效的load和store操作，进一步提升编译器性能。

总结

实验结果总结

本实验中，我们修改了之前实现的语义分析部分，使得在进行语义分析的过程中可以生成中间代码。随后，我们的编译器会根据中间代码生成汇编代码。另外，我们在此基础上实现了-O1局部优化。至此，我们的代码已经成为一个完整的编译器，并具备简单的优化功能。它能够识别CACT语言格式的源文件，生成RISCV64GC平台下的汇编代码。汇编代码通过spike软件模拟运行，生成了正确的执行结果。

分成员总结

游昆霖：

在本次实验中，我主要负责了代码的设计、实现和优化。在编写实验中，我对中间代码的设计、汇编代码的生成和局部优化实现都有了比较深刻的认识。在中间代码的设计中，我主要考虑了怎么能够更好的使一条语句和其中能够表达更充分的信息，以减少生成汇编过程的耦合性，方便进行代码的生成优化；在汇编代码生成过程中，我对RISCV指令集有了更深刻的了解，包括如何根据不同类型的变量选择对应的指令，如何用更少的指令高效实现功能；在局部优化的实现方面，我主要基于龙书的指导和我们的中间代码设计进行了优化，对于如何高效的利用计算结果和优化中间代码有了更深刻的了解，也对课内所学知识有了更深刻的认识。

彭睿思：

在本次实验中，我主要负责程序功能性和正确性的调试。由于本次实验复杂度高，出错的可能位置也较多。我们在充分检查代码逻辑的基础上，结合测试样例的结果进行了错误的定位和修正。通过print、gdb等方法，我们可以较好的将错误定位到具体的代码语句，并进而检查中间代码和汇编代码，依次查缺补漏，从而较好的完善了整体实验。

严哲虞：

本次实验的代码量较大，但模块划分比较清晰。难点在于设计中间代码，需要考虑的情况比较复杂，并且要对中间代码和目标汇编代码之间的对应关系有清晰的规划。通过本次实验，我对编译器生成汇编代码的过程有了更清晰的认识，也巩固了对RISC-V汇编以及ABI的认识，对于之前并不十分了解的伪指令也有了一定的认识，并进一步加深了对编译器工作方式的理解。

赵若雯：中间代码和汇编代码的生成是编译器开发的重要环节，这个过程需要深入了解语法、语义分析过程和RISCV指令集的指令结构。在设计中间代码时，需要合理地设计数据、代码结构以便后续的优化；在生成汇编代码时也需要仔细考虑寄存器的分配和代码布局的问题。在本次实验中，通过分析和生成不同情况下的中间代码表示，我更深入地了解了编译器后端的工作原理，也巩固了一些C++开发语言的知识。总体来说，通过研讨课上的三次实验，我体会到了构建一个完整的编译器的基本流程和工程，同时也加深了我对课内相关知识的理解。