南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

编译原理实验报告

上机大作业报告

谢畅

年级：2021 级

专业：计算机科学与技术

指导教师：王刚

2024 年 1 月 16 日

摘要

Sysy 编译器的总体设计与各模块设计的说明

关键字：编译器 CFG SysY arm 汇编

目录

1 概述与介绍

1.1 总体设计

Sysy 编译器的总体设计遵循经典的编译器构建阶段，包括词法分析、语法分析、语义分析、中间代码生成、代码优化、目标代码生成等关键阶段。这个设计使得编译器能够逐步处理源代码，进行必要的分析和转换，最终生成可执行的目标代码。在整个设计过程中，我们特别注重对 Sysy 语言的特性，如浮点数和数组的支持，以确保编译器能够满足语言规范的要求。

1.2 各模块设计

- **词法分析阶段**:

  我们选择了 lexel 词法分析器作为词法分析阶段的工具，用于将源代码划分为各种词法单元。这是构建编译器的第一步，为后续的语法分析提供了正确的输入。

- **语法分析阶段**:

  在语法分析中，我们利用 yacc 与语法制导翻译构建了一棵语法树。这个树状结构反映了源代码的语法结构，为后续的语义分析和中间代码生成提供了基础。

- **语义分析阶段**:

  语义分析阶段负责检测变量的定义与使用是否合理，以及处理相关函数调用等错误。我们还在这个阶段完成了对隐式类型转换的处理，确保 Sysy 语言的类型系统能够得到正确的实现。

- **中间代码生成阶段**:

  在这一阶段，我们通过自顶向下遍历语法树，将语法树节点转化为中间代码的相关指令，并将其保存在一个新的数据结构（如 unit 树）中。这个中间表示将为后续的代码优化和目标代码生成提供一个抽象的桥梁。

- **目标代码生成阶段**:

  目标代码生成是编译器的最后一步，它需要再次遍历中间代码生成阶段得到的 unit 树，将其中的每个指令转化为一个或多个目标代码。在这个过程中，我们引入了 genmachinecode 成员函数，负责将中间代码翻译成目标代码。此外，我们在这个阶段进行寄存器分配，以最小化使用的寄存器数量，从而提高生成的目标代码的效率。

- **代码优化：中间代码的 cfg 简化**

  在代码优化阶段，删除一些无用的基本块，比如不可达的基本块，以及基本块的合并。同时进行了寄存器分配优化的尝试。

整个编译器的设计思路是保证每个阶段都能正确处理 Sysy 语言的语法和语义，同时在目标代码生成阶段进行必要的优化，以获得高效的目标代码。

代码链接

2 完成的相关工作

2.1 词法分析

完成了浮点数的识别、完成了其他终结符的相关识别。如下图，识别了 10 进制、16 进制的浮点数常量

```
1
2 DECIMAL_FLOATCONST (([0-9]*\.[0-9]+)|([0-9]+\.))|[0-9]+)([eE][+-]?[0-9]+)?
3 HEX_FLOATCONST 0[xX](([0-9a-fA-F]*\.[0-9a-fA-F]+)|[0-9a-fA-F]+\.)|[0-9a-fA-F]+)([pP][+-]?[0-9]+)?
```

2.2 语法分析

2.2.1 符号表的搜索

```
1  SymbolEntry* SymbolTable::lookup(std::string name)
2  {
3      SymbolTable*cur=identifiers;
4      while(cur!=nullptr){
5          if(cur->symbolTable.find(name)!=cur->symbolTable.end()){
6              return cur->symbolTable[name];
7          }
8          cur=cur->prev;
9      }
10     return nullptr;
11 }
12
```

完善符号表的相关搜索，如上设计了相关代码。

2.2.2 数据类型、函数返回值类型

2.2.3 表达式 (算术运算:+、-、*、/、%，逻辑运算与关系运算)

实现了算术运算:+、-、*、/、%, 逻辑运算 (>,<,>=,<=,!=), 关系运算 (&&,||、!)。

主要是要注意优先级，比如加法减法、乘法除法等，还有逻辑运算与关系运算。注意三者也有关系，首先进行算数运算、再是逻辑运算 (>,<.>=,<=,!=,==等)，然后最后是关系运算 (&&,||,!) 等。

```
1  Exp → AddExp
2  AddExp → MulExp | AddExp ('+' | '−') MulExp
3  MulExp → UnaryExp | MulExp ('∗' | '/' | '\%') UnaryExp
4  RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp
5  Cond → OrExp
6  OrExp → AndExp | OrExp '||' AndExp
7  AndExp → EqExp | AndExp '\&\&' EqExp
8  EqExp → RelExp | EqExp ('==' | '!=') RelExp
```

2.2.4 输入输出 (连接了 sysy 运行时库)

这里主要是在词法分析中修改，将特定的库函数名置入符号表中。

2.2.5 函数的识别与调用

在这里要注意函数的参数情况，形参若有数组，那么第一个维度是 []，没有表达式；在函数调用时正常识别即可。主要是要与队友实现的变量定义、数组结合，将函数形参转化为一个变量定义结点。在此阶段主要是要在语法分析中得到一颗语法树，各节点对应不同属性。浮点数的识别与支持，相关运算等

隐式类型转换在语义分析阶段，这里实现的略微粗糙，注意要区别调用函数的 lval，必须得到其 returnType，运算的中间临时变量的类型，需要根据两个操作数的类型进行判断，如果有浮点数，那么要转成浮点数。在这里把小的类型转为大的类型，即 bool 型转为整型，整型转浮点数。

2.3 语义分析/类型检查

2.3.1 函数相关

检查未声明函数、及函数形参与实参类型、数目是否匹配

如下面代码所示，首先进行参数数目的匹配，然后是参数类型的匹配。在这里设计隐式类型转换，所以每个参数类型不同不会报错，而是在 2.2.3 节进行隐式类型转换。

```
 if(Rparams->params.size()!=paramsType.size()){
    fprintf(stderr, "FunctionCallExpr: function %s, param size not match\n",
    this->symbolEntry->toStr().c_str());
    assert(Rparams->params.size()==paramsType.size());
}
for(int i=0;i<(int)paramsType.size();i++){
    Type*Rparams_type=Rparams->params[i]->getSymPtr()->getType();
    if(Rparams_type->isFunc()){
        FunctionType*func=dynamic_cast<FunctionType*>(
        Rparams->params[i]->getSymPtr()->getType());
        Rparams_type=func->getRetType();
    }
```

2.3.2 return 的检查

检查 return 语句操作数与函数声明的返回值类型是否匹配。

这里在语法分析的语法制导中添加，即添加一个 flag，如果检测到有 return 那么 return 的元素的类型与该函数 se 的返回值类型一致，那么正确；否则，如果返回值不匹配，或者无 return 且返回值不是 void，那么报错即可。

2.3.3 隐式类型转换完善

如浮点数、整型、Bool 三者的转换，以及数组初始化的语句里面的类型转换 (浮点数、整型)，添加了 typecast 节点。

主要是在 typecheck 添加，比如 initvalList(即 int a[2]={} 的 {})，也需要检测 {} 中的每个值与原数组的元素的类型是否一致，不一致需要隐式类型转换。

```

void::VarDef::typeCheck()
{
    // TODO: 类型与初值相结合的检查
    id->typeCheck();
    if(dim) dim->typeCheck();
    if(init){
        if(!dim){
```

```
            //单个变量
            dynamic_cast<InitVal*>(init)->setDeclType(
            this->getID()->getSymPtr()->getType());
        }
        else{
            //数组
            dynamic_cast<InitValList*>(init)->setDeclType(
            this->getID()->getSymPtr()->getType());
        }
        init->typeCheck();
    }
}
void InitValList::typeCheck()
{
    if(seq){
        dynamic_cast<SeqInitVal*>(seq)->setDeclType(this->getDeclType());
        seq->typeCheck();
    }
}
void FunctionCall::typecheck(){
    ...
    if(paramsType[i]!=Rparams_type){
        //给定实参是函数的返回值，需要检查是否可以转换
        //float/int/bool 转换
        //实参类型转为形参类型 有八种情况 实际由于形参不可能为 bool，所以减少两种
        //由于常量可以直接转换，所以首先判断是否为常量  int/float 直接转换
        if(paramsType[i]->isPtr()||paramsType[i]->isArray()){
            continue;//不检查指针
        }
        if(Rparams->params[i]->getSymPtr()->isConstant()){
            //常量转换
            if(paramsType[i]==TypeSystem::floatType){
                //实参为 int，形参为 float
                double value=dynamic_cast<ConstantSymbolEntry*>(
                Rparams->params[i]->getSymPtr())->getValue();
                SymbolEntry *se = new FloatConstantSymbolEntry(
                TypeSystem::floatType, value);
                Rparams->params[i] = new Constant(se);
                continue;
```

```
48              }
49              else if(paramsType[i]==TypeSystem::intType){
50                  //实参为 float, 形参为 int
51                  int value=dynamic_cast<FloatConstantSymbolEntry*>(
52                  Rparams->params[i]->getSymPtr())->getValue();
53                  SymbolEntry *se = new ConstantSymbolEntry(TypeSystem::intType, value);
54                  Rparams->params[i] = new Constant(se);
55                  continue;
56              }
57          }
58      }....
59  }
```

2.4 中间代码生成

2.4.1 赋值语句

```
1  //翻译成 store 与 load 的组合
2  void AssignStmt::genCode()
3  {
4      lval->setAssign();
5      lval->genCode();
6      Operand *addr;
7      if(lval->getSymPtr()->getType()->isArray()||lval->getSymPtr()->getType()->isPtr()){
8          addr=lval->getOperand();//注意设置 assign(), 标识左值 不用再 load, 直接取指针即可
9      }
10     else{
11         addr = dynamic_cast<IdentifierSymbolEntry*>(lval->getSymPtr())->getAddr();
12     }
13     Operand *src = expr->getOperand();
14     new StoreInstruction(addr, src, bb);
15 }
```

主要区别左值。左值不需 load，而右值需要先 load(因为只有地址，要获取值)，而直接将值 store 进左边的地址即可。上述代码的 lval 是左值，而右值是 expr，我们只需要得到其地址，也就是用 getOperand 即可。

2.4.2 控制流语句

if-else、while 语句、return、break、continue 等语句

难点主要是回填技术。即一开始相关控制流语句不知道跳转的位置，在后来生成了跳转的基本块后，才能知晓。而 break、continue 是需要加无条件跳转指令。

```
cond->setCond();
cond->genCode();
//回填技术，回填真/假跳转的基本块
backPatch(cond->trueList(), then_bb,true);
backPatch(cond->falseList(), end_bb,false);
```

同时，注意最后要加一条无条件跳转语句，调至 ifstmt/whilestmt 后面的 endblock，但是这里首先检测是否最后有无条件跳转，如果有，说明可能是 break 等的影响，那么不用再加无条件跳转了。

在这里以如下代码详细讲述：

- 首先我们要保存此指令至 whilestmtStack 栈中，是因为 break 与 continue 会用到，我们 break 与 continue 是与最内层的 while 进行匹配。
- 然后需要进行 cond_bb、then_bb、end_bb 基本块的生成
- 注意这里要进行回填，我们首先对 cond 进行 setcond，然后 gencode，最后回填 cond 的 true_list，在 end_bb 生成后，也会回填 cond 的 false_list。
- 在 while 的 stmt 结束后，要进行无条件跳转至 end_bb，这里判断了一下 then_bb 的最后一条指令是否为无条件跳转指令，如果是那么不用生成。

```
void WhileStmt::genCode()
{
    whileStmtStack.push(this);
    Function *func;
    BasicBlock *cond_bb,*then_bb, *end_bb,*bb;
    bb=builder->getInsertBB();
    func = builder->getInsertBB()->getParent();
    cond_bb=new BasicBlock(func);

    this->setCondBB(cond_bb);
    //从当前基本块跳转到 cond_bb
    then_bb=new BasicBlock(func);
    ...
    else{

```

```
16          new UncondBrInstruction(cond_bb, bb);
17          builder->setInsertBB(cond_bb);
18          cond->setCond();
19          cond->genCode();
20          backPatch(cond->trueList(), then_bb,true);
21      }
22
23      builder->setInsertBB(then_bb);
24      Stmt->genCode();
25      then_bb = builder->getInsertBB();
26      end_bb=new BasicBlock(func);
27      //回填 end_bb 保证 end_bb 是最后生成的基本块
28      for(auto i:breaklist){
29          dynamic_cast<UncondBrInstruction *>(i)->setBranch(end_bb);
30      }
31      backPatch(cond->falseList(), end_bb,false);
32      //循环结束后跳转到 cond_bb
33      //如果 then_bb 的最后一条指令不是无条件跳转指令，则生成一条无条件跳转指令
34      if(!then_bb->rbegin()->isUncond())
35          new UncondBrInstruction(cond_bb, then_bb);
36
37      builder->setInsertBB(end_bb);
38      // then_bb->addSucc(cond_bb);
39      // cond_bb->addPred(then_bb);
40
41      whileStmtStack.pop();
42  }
```

2.4.3 输入输出

输入输出主要是在前部生成一个声明，也就是库函数的声明。

而为了方便起见，我们会在 unit 中保存需要声明的库函数的符号表项，但是这个需要去重，所以采用 set 去重即可；然后在符号表项增加一个 gen_declcode 的代码，用以生成库函数的声明

```
1  void Unit::output() const
2  {
3      for(auto se:lib_func_list){
4          se->gen_declcode();
```

```
    }
    ...
}
...

//库函数产生声明的代码
void IdentifierSymbolEntry::gen_declcode()
{

    if(isLibFunc()){
        fprintf(yyout, "declare %s @%s(",
                dynamic_cast<FunctionType *>(type)->getRetType()->toStr().c_str(),
                name.c_str());
        //输出参数类型
        for(auto i:dynamic_cast<FunctionType *>(type)->getParamsType()){
            fprintf(yyout, "%s", i->toStr().c_str());
            if(i!=dynamic_cast<FunctionType *>(type)->getParamsType().back()){
                fprintf(yyout, ", ");
            }
        }
        fprintf(yyout, ")\n");
    }
}
```

2.4.4 函数

涉及函数声明 (有形参时的相关代码生成)、函数调用

对于函数声明，关键在于参数的处理，因为其他函数体等都是调用的各个子节点的处理函数。

- 首先我们在语法分析阶段拥有保存的参数列表。在这里需要对每个参数生成标识，标识其是第几个参数
- 然后我们要判断形参是否为数组, 如果是那么需要更改这些形参的类型, 因为我们语法分析阶段，符号表项没有正确设置，这里结合了队友对数组的处理。
- 最后更改完参数的 se 后，我们要一开始分配空间，将参数存储在栈中，也就是 store 指令。

- 注意在这里生成参数的标识符，是为了目标代码对参数数量的标识，因为中间代码对参数没有数量限制，无论多少个都可以用 alloca 处理；而目标代码中，只有前 4 个参数可以由 r0-r3/s0-s3 接收，其他保存在栈中。

```
void FunctionFParam::genCode(){
    Function *func = builder->getInsertBB()->getParent();
    BasicBlock *entry = func->getEntry();
    std::vector<Operand*>param_list;
    int paramNo=0;
    for(auto i:params){
        IdentifierSymbolEntry *se = dynamic_cast<IdentifierSymbolEntry*>(
        i->getID()->getSymPtr());
        se->setParamNo(paramNo++);
        //区别数组/普通类型
        ConstDim*dim=i->getdim();
        if(se->getdim()){
            //改为指向数组的指针类型
            // 为了避免每次使用都 new arraytype，这里一次性 new 了
            if(dim->elements){
                Type **list = new Type*[dim->dim + 1];
                //这里函数参数为数组，是 int a[][2] 形式

                for(int d=1;d<=dim->dim-1;d++){
                    list[d] = new ArrayType(se->getType(), d,
                    {dim->dim_values.end()-d,dim->dim_values.end()});
                }
                //0 维就是普通类型
                list[0] = se->getType();
                //标识是函数参数的数组
                list[dim->dim]=nullptr;
                se->setArrayType(list);
                // se 改为指向数组的指针类型  [3 x [4 x [5 x i32]]]* %b
                se->setType(new PointerType(list[dim->dim-1]));
            }
            else{
                //说明直接设置为指针
                Type **list = new Type*[dim->dim + 1];
                list[0] = se->getType();
                //标识是函数参数的数组
```

```
36                  list[1]=nullptr;
37                  se->setArrayType(list);
38                  se->setType(new PointerType(se->getType()));
39              }
40          }
41          Operand*param_src=new Operand(se);
42          param_list.push_back(param_src);
43          Instruction *alloca;
44          Operand *addr;
45          SymbolEntry *addr_se;
46          Type *type;
47          type = new PointerType(se->getType());
48          //分配空间，生成指针
49          addr_se = new TemporarySymbolEntry(type, SymbolTable::getLabel());
50          addr = new Operand(addr_se);
51          alloca = new AllocaInstruction(addr, se);
52          entry->insertFront(alloca);
53          se->setAddr(addr);
54          //用形参给其赋值
55          //todo  数组
56          Instruction *store = new StoreInstruction(addr, param_src);
57          entry->insertBack(store);
58      }
59      func->setParams(param_list);
60  }
```

函数声明的关键在于处理参数，即之前的参数要当作一个变量定义，首先把这些参数 store 到本地 alloca 的局部栈中，在目标代码中也要进行处理。

对于函数调用，我们首先对保存的实参列表的相关表达式进行生成中间代码，因为实参可以是 Expr；在之后，生成 call 指令即可

2.4.5 完善构造前驱后继流图

在最后 FunctionDef::genCode 中，我们遍历一个函数每个基本块，在这里做了一个优化: 如果这个基本块的所有指令中，ret 之后还有指令，可以直接删去后续指令。每个基本块中，检测最后一条指令: 如果是无条件跳转，增加一个控制流关系；如果是有条件跳转，需要增加两个控制流关系。

2.4.6 lval：访问参数数组的元素

在队友的基础上，对函数形参存在数组的情况，进行数组的完善、同时支持使用函数形参数组

在函数形参中，若存在数组，那么这个其实是指针，我们在栈中首先 alloc，然后得到的是指针的指针，用到这个数组时，需要 load 得到指针，然后 gep，此时第一条 gep 只存在第一个偏移量，不存在第二个偏移量，与不是参数的数组不一样；这是因此形参数组没有第一个维度的 exp 导致的。

对于数组是参数的情况，使用有如下的中间代码:

```
%t2 = load [3 x i32]*, [3 x i32]** %t1, align 4
%t3 = getelementptr inbounds [3 x i32], [3 x i32]* %t2, i32 2
%t4 = getelementptr inbounds [3 x i32], [3 x i32]* %t3, i32
    0, i32 4
```

这是因为形参是数组时，我们不知道第一个维度，所以传的其实是其他维度的指针，而第一个维度相当于是 getelementptr 的第一个偏移的参数。所以对函数数组进行使用时，我们第一条 gep 指令是只有第一个偏移量，没有第二个偏移量的。

如下代码所示:

```
int ifFirst = 1;//用其区别第一条 gep 指令
int total_dim=se->getdim();
if(!se->getArrayType(se->getdim())){
    //函数参数的形参数组
    // 首先加载数组指针 也就是开辟的空间
    dst = new Operand(new TemporarySymbolEntry(se->getType(), SymbolTable::getLabel()));
    new LoadInstruction(dst, prev_addr, bb);
    //开始循环
    prev_addr=dst;
    for(d=_dim->dim;d>0;d--){
        Type *dst_type = new PointerType(se->getArrayType(total_dim-1));

        SymbolEntry *__addr_se = new TemporarySymbolEntry(dst_type, SymbolTable::getLabe
        Operand *__addr = new Operand(__addr_se);

        ExprNode *expr = dimExprList[_dim->dim-d];
        expr->genCode();
      TemporarySymbolEntry(TypeSystem::intType, SymbolTable::getLabel());

```

```
        new getelementptrInstruction(__addr, prev_addr, expr->getOperand(),
        ifFirst ,bb, ifFirst);
        ifFirst=0;
        prev_addr = __addr;
        total_dim-=1;
    }
    //getArrayType(0) 可以获取数组基类型
        if(_dim->dim==se->getdim()&&!this->isAssign()){
                dst = new Operand(new TemporarySymbolEntry(se->getArrayType(0),
                SymbolTable::getLabel()));
                new LoadInstruction(dst, prev_addr, bb);
        }
        else
            dst=prev_addr;
}
```

2.4.7 完善 typecast

对 typecast 节点进行完善，覆盖了所有隐式转换对应的中间代码生成，增加 intfloatcastInstruction 等指令。

对于 float 与 Bool，可以先把 bool 转成 int，再将 Int 转为浮点数。

```
 else if(castType==ITF){
    //%9 = sitofp i32 %6 to float
    expr->genCode();
    Operand *src = expr->getOperand();
    new INTFLOATcastInstruction(INTFLOATcastInstruction::I2F, dst,src, bb);
}
else if(castType==FTI){
    //%10 = fptosi float %5 to i32
    expr->genCode();
    Operand *src = expr->getOperand();
    new INTFLOATcastInstruction(INTFLOATcastInstruction::F2I, dst,src, bb);
}
else if(castType==BTF){
    //先把 bool 转为 int，再把 int 转为 float
    expr->genCode();
    Operand *src = expr->getOperand();
    Operand*temp=new Operand(new TemporarySymbolEntry(TypeSystem::intType,
```

```
18      SymbolTable::getLabel()));
19      new ZextInstruction(temp, src, bb);
20      new INTFLOATcastInstruction(INTFLOATcastInstruction::I2F,dst, temp, bb);
21  }
```

2.4.8 浮点数

浮点数与整型比较类似，首先对于浮点数与整型的运算，我们在之前完善了typecast，所以最终只用关注浮点数与浮点数的运算即可。

- 浮点数在 llvm ir 的值的表示。经过查阅资料发现，llvm 要转成 16 进制的 HexStr 形式。有如下代码

```
1   // llvm 的 16 进制 float 格式
2   std::string Double2HexStr(double val)
3   {
4       union HEX
5       {
6           double num;
7           unsigned char hex_num[8];
8       } ptr;
9       ptr.num = val;
10
11      char hex_str[16] = {0};
12      for (int i = 0; i < (int)sizeof(double); i++)
13          snprintf(hex_str + 2 * i, 8, "%02X", ptr.hex_num[8 - i - 1]);
14      if (('0' <= hex_str[8] && hex_str[8] <= '9' && hex_str[8] % 2 == 1) ||
15          ('A' <= hex_str[8] && hex_str[8] <= 'F' && hex_str[8] % 2 == 0))
16          hex_str[8] -= 1;
17      return "0x" + std::string(hex_str, hex_str + 9) + std::string(7, '0');
18
19  }
20
```

- 对运算指令进行修改，增加 fadd、fmul、fsub；同时 cmp 进行修改，增加 one、oeq

```
1       //binary 中
2       if(operands[0]->getType()->isInt()){
```

```
        switch (opcode)
        {
        case ADD:
            op = "add";
            break;
        ...}
    }
    else{
        switch (opcode)
        {
        case ADD:
            op = "fadd";
            break;
        ...}
    }
    //cmp 中
    if(operands[1]->getType()->isInt()){
        switch (opcode)
        {
        case E:
            op = "eq";
            break;
        }
    }
    else{
        switch (opcode)
        {
        case E:
            op = "oeq";
            break;
        }
    }
```

2.5 目标代码生成

2.5.1 函数

完成函数定义以及函数调用的翻译。

函数定义在这里不需要翻译，在中间代码时我们将函数定义转为了各个小结

点的 instruction，因此这里只需要关注 call 指令。如下所示：

- 调用之前要把前 4 个参数移至 r0,r1,r2,r3 中；多余 4 个的情况压入栈中。在函数定义要先 push、同时局部变量会占据栈的空间，将栈 sp 的位置进行移动；在函数的最后，要恢复相关保存的寄存器、同时 bx lr 跳回原位置。
- 同时，注意函数调用之前参数的置值的位置，应该是从右往左处理参数，也就是大于 4 个的参数情况，最右边的参数最先入栈。
- 其次，在函数定义开始要 Push 保存的寄存器，在这里保存的寄存器只有最后寄存器分配之后才能得到，所以在最后 machinefunction::output 里面再 push 即可，这里不用产生 stack 指令。而产生 stack 指令的地方，是调用函数时，参数超过 4 个的情况。注意函数调用完毕要恢复栈指针 sp 的位置 (可能之前调用函数，将参数入栈)。
- 注意，函数在使用的参数若超过 4 个，那么一开始不用从 r0-r3 中储存，而这些参数已经存在栈中，但是偏移量会改变，这是因为函数一开始 Push 保存寄存器引起的，我们原来设定的超过 4 个以上的参数的偏移量是从 0 开始的，那么现在的起始值是 0+ 保存寄存器的 size*4，这样就可以正确使用超过 4 个以上的参数了。

有如下代码示意：

```
for(int i=(int)operands.size()-1;i>=1;i--){
    //r0-r3 四个  浮点数 s0-s3todo
     auto src = genMachineOperand(operands[i]);

    if(i<=4){
        //生成的是真实寄存器 0-3 编号
        auto dst = genMachineReg(i-1,src->getValType());
        //立即数 需要加一条到寄存器   浮点数 todo
        if(src->isImm()&&src->getValType()==TypeSystem::intType){

            cur_inst = new LoadMInstruction(cur_block, dst, src);
            cur_block->InsertInst(cur_inst);
        }
        else{
             if(src->isImm()&&src->getValType()==TypeSystem::floatType){
                //ldr r10, =1069547520   vmov s31,r4
                 auto internal_reg = genMachineVReg();
```

```
                cur_block->InsertInst(new LoadMInstruction(cur_block, internal_reg, src));
                src = new MachineOperand(*internal_reg);
             }
            //不是立即数  ldr r10, [fp, #-12] 在 call 之前会做   mov r0,r10     mov
            cur_inst = new MovMInstruction(cur_block,
            dst->getValType()->isFloat() ? MovMInstruction::VMOV :
            MovMInstruction::MOV, dst, src);
             cur_block->InsertInst(cur_inst);
        }
    }
    else{
        //压栈  压完栈需要恢复现场
        //ldr r10, [fp, #-12]   str r10, [sp, #4]
        if(src->isImm())
            src=cur_block->insertLoadImm(src);
        //压栈
        cur_inst = new StackMInstructon(cur_block, StackMInstructon::PUSH, src);
        cur_block->InsertInst(cur_inst);
    }

}
```

2.5.2 完成 Retinstruction

在这里，我们 return 不仅首先要对 r0/s0 等寄存器赋值，而且要恢复保存的寄存器以及 sp、同时要 bx lr 跳转至正确的位置。在我们对 retinstruction 进行 Genmachinecode 时，我们并不知道这个 pop 的寄存器是什么，因此在这里为了方便处理，加一个跳转指令，所有的 return 都调至一个 Lend，在 machinefunction::output 时再补充 Lend 里面的 pop{保存寄存器}、mov sp,fp，以及 bx lr，这样就解决了上述问题。

```
void RetInstruction::genMachineCode(AsmBuilder* builder)
{
    auto cur_block = builder->getBlock();
    if(operands.size()){
        //有返回值
        auto src = genMachineOperand(operands[0]);
        //立即数 需要加一条到寄存器
```

```
        if(src->isImm()){

            src = cur_block->insertLoadImm(src);
        }
        //mov r0, src   也可能是浮点数 所以用 val_type 决定
        auto dst = new MachineOperand(MachineOperand::REG, 0, src->getValType());
        //浮点数的 mov 也要更改
        if(dst->getValType()->isFloat()){
            auto cur_inst = new MovMInstruction(cur_block, MovMInstruction::VMOV,
            dst, src);
            cur_block->InsertInst(cur_inst);
        }
        else{
            auto cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
            dst, src);
            cur_block->InsertInst(cur_inst);
        }
    }
    //跳转到 pop、bx 的 L+ 函数名 _end
    std::string name=".L"+builder->getFunction()->getFuncName()+"_end";
    auto cur_inst = new BranchMInstruction(cur_block,
    BranchMInstruction::B, new MachineOperand(name));
    cur_block->InsertInst(cur_inst);
}
```

2.5.3 完成浮点数的目标代码生成

1. 浮点数的全局变量中，要将将一个浮点数的二进制表示转换为相应的无符号整数。

   在使用浮点数的立即数也要进行相关转换：加载浮点数常量、整数到浮点转换、存储浮点数到内存

```
union FloatIntUnion {
    float floatValue;
    unsigned intValue;
};
unsigned getMvalue(){FloatIntUnion unionValue;unionValue.
   floatValue = value;return unionValue.intValue;}
```

```
7
8       ;关于浮点数全局变量
9       .global a
10      .align 4
11      .size a, 4
12  a:
13      .word    1067869798
14
15      ;关于浮点立即数的使用
16      ldr r4, =1067869798
17      vmov s16, r4
18      vstr.32 s16, [fp, #-4]
19
20      //关于浮点数的转换 即Intfloatinstruction的genmachineCode
21
22  case I2F:
23          //sitofp
24          //vmov s31, r10
25          //vcvt.f32.s32 s31, s31
26          //vmov s0, s31
27          ....
28          break;
29  case F2I:
30          //fptosi
31          //vcvt.s32.f32 s30, s31
32          //vmov r10, s30
33          ...
34          break;
```

2. 更改相关 printreg 代码，因为浮点数对应的寄存器是 s 寄存器。同时需要在 machineoperand 增加 valtype，区分浮点数和整型。

3. 支持了 vmov、vldr.32、vstr.f32、vcvt.s32.f32、vcvt.f32.s32 等指令。

   注意我们每次对之前的 operand 生成 machinoperand 时,都要判断其背后的临时变量/立即数等等的类型，如果是浮点数，那么要传给 machineoperand 的 valtype 属性，将其置值，同时 mov、ldr、str 背后 output 时，要检测类型，如果是整型直接用 mov、str、ldr 即可，如果是浮点数是需要用 vmov、vstr.f32、vldr.f32 等指令的。

4. 同时，注意函数参数有数组的情况，是用 ldr 将 r0 导入本地局部变量栈中，无论是整型数组还是浮点数数组，因为地址是整型，所以不能用 vldr.s32。而要用 ldr 10,[fp,#offset]

```
    //在 genmachineOperand(Operand*ope) 中
else if(id_se->isParam()){

    //只有 0-3 生成  r0,r1,r2,r3  之后的参数在栈中
    int paramNo = id_se->getParamNo();
    int flag=0;
    if(id_se->getType()->isFloat()||id_se->getType()->isInt()){
        flag=1;
    }
    if (paramNo >= 0 && paramNo <= 3){
        if(flag==1)
            mope = new MachineOperand(MachineOperand::REG, paramNo,
            id_se->getType());
        else    //无论是整型数组还是浮点数数组对应的都是 Int，地址是 int
            mope = new MachineOperand(MachineOperand::REG, paramNo,
            TypeSystem::intType);
    }
}
```

5. 注意，浮点数存在一种对齐的情况。如果仅仅是满足 4 字节对齐，就会出现我们前面打印浮点数的错误问题，所以在函数一开始 sub 栈 sp 指针时，要进行检测，如果是 putarray/getarray 等，要再多加 4Bytes，也就是 8 字节对齐。

2.5.4 相关错误的修改

注意全局变量的地址与使用的代码距离不能过大。

我们可以每隔 300 行，就再定义一个全局变量的地址.LTORG 其中包括 addr_n_name，那么使用的时候不会因为地址距离过远而报错。

2.6 代码优化

代码优化：Mem2reg 尝试

- 循环遍历函数列表

- 计算支配树（Dominance Tree）：

  函数的目标是计算给定函数 func 的支配树。支配树是指在控制流图中，某个节点支配另一个节点的情况。这是为了在后续的步骤中更容易处理基本块的支配关系。

- 计算支配前沿（Dominance Frontier）：

  ComputeDomFrontier 函数用于计算给定函数 func 的支配前沿。支配前沿是指在支配树上, 某个节点的支配边界。支配前沿的计算有助于在插入 PHI 节点时确定合适的位置。

- 插入 PHI 节点：

  InsertPhi 函数的是在适当的位置插入 PHI 节点。PHI 节点是用于处理支配边界处的变量定义的一种特殊类型的指令。帮助在支配节点之间传递变量的值，以确保在执行时具有正确的值。

- 重命名（Rename）：

  Rename 函数负责对函数中的变量进行重命名, 是为了确保在插入 PHI 节点之后，变量的引用和定义都得到正确的处理，以适应新的支配关系。

在中间代码阶段进行一些小的优化

比如删除了基本块中位于 ret 之后的所有代码，同时控制流语句最后只有一条无条件跳转指令，避免 break 等影响，避免有多条无条件跳转指令。