实验要求

1. 对lab3 生成的 IR 进行优化。为下一阶段编写优化方案提供必要的基础知识：SSA 格式的 IR ，以及优化 Pass 的概念。

思考题

1. 请简述概念：支配性、严格支配性、直接支配性、支配边界。

   支配性：d是n的支配顶点（$d \in Dom(n)$）指若从初始结点起，每条到达n的路径都要经过d

   严格支配性：当且仅当$a \in Dom(n) - \{n\}$，a严格支配n

   直接支配性：严格支配集合中距离n最近的节点，记做$IDom(n)$

   支配边界：若对于节点n的支配边界$DF(n)$中任意节点m满足两条性质：1. n支配m的一个前驱节点；2.n并不严格支配m

2. `phi`节点是SSA的关键特征，请简述`phi`节点的概念，以及引入`phi`节点的理由。

   `phi`节点指，如果在基本块B定义了y，则在$DF(B)$包含的每个节点起始处插入 $y \leftarrow \phi(y, y)$

   理由：SSA需要满足：1.过程中每个定义需要有唯一名字；2.每个使用处都引用了一个定义

   因此需要插入`phi`节点将汇合处不同路径上静态单赋值形式名调和为一个名字

3. 观察下面给出的`cminus`程序对应的 LLVM IR，与**开启**`Mem2Reg`生成的LLVM IR对比，每条`load`，`store`指令发生了变化吗？变化或者没变化的原因是什么？请分类解释。

```c
int globvar;
int func(int x){
    if(x > 0){
        x = 0;
    }
    return x;
}
int main(void){
    int arr[10];
    int b;
    globvar = 1;
    arr[5] = 999;
    b = 2333;
    func(b);
    func(globvar);
    return 0;
}
```

   before `Mem2Reg`:

```
@globVar = global i32 zeroinitializer
declare void @neg_idx_except()
define i32 @func(i32 %arg0) {
label_entry:
  %op1 = alloca i32
  store i32 %arg0, i32* %op1
  %op2 = load i32, i32* %op1
```

```
  %op3 = icmp sgt i32 %op2, 0
  %op4 = zext i1 %op3 to i32
  %op5 = icmp ne i32 %op4, 0
  br i1 %op5, label %label6, label %label7
label6:                                           ; preds =
%label_entry
  store i32 0, i32* %op1
  br label %label7
label7:                                           ; preds =
%label_entry, %label6
  %op8 = load i32, i32* %op1
  ret i32 %op8
}
define i32 @main() {
label_entry:
  %op0 = alloca [10 x i32]
  %op1 = alloca i32
  store i32 1, i32* @globVar
  %op2 = icmp slt i32 5, 0
  br i1 %op2, label %label3, label %label4
label3:                                           ; preds =
%label_entry
  call void @neg_idx_except()
  ret i32 0
label4:                                           ; preds =
%label_entry
  %op5 = getelementptr [10 x i32], [10 x i32]* %op0, i32 0, i32 5
  store i32 999, i32* %op5
  store i32 2333, i32* %op1
  %op6 = load i32, i32* %op1
  %op7 = call i32 @func(i32 %op6)
  %op8 = load i32, i32* @globVar
  %op9 = call i32 @func(i32 %op8)
  ret i32 0
}
```

After `Mem2Reg`:

```
@globVar = global i32 zeroinitializer
declare void @neg_idx_except()
define i32 @func(i32 %arg0) {
label_entry:
  %op3 = icmp sgt i32 %arg0, 0
  %op4 = zext i1 %op3 to i32
  %op5 = icmp ne i32 %op4, 0
  br i1 %op5, label %label6, label %label7
label6:                                           ; preds =
%label_entry
  br label %label7
label7:                                           ; preds =
%label_entry, %label6
  %op9 = phi i32 [ %arg0, %label_entry ], [ 0, %label6 ]
  ret i32 %op9
}
define i32 @main() {
```

```
label_entry:
  %op0 = alloca [10 x i32]
  store i32 1, i32* @globVar
  %op2 = icmp slt i32 5, 0
  br i1 %op2, label %label3, label %label4
label3:                                           ; preds =
%label_entry
  call void @neg_idx_except()
  ret i32 0
label4:                                           ; preds =
%label_entry
  %op5 = getelementptr [10 x i32], [10 x i32]* %op0, i32 0, i32 5
  store i32 999, i32* %op5
  %op7 = call i32 @func(i32 2333)
  %op8 = load i32, i32* @globVar
  %op9 = call i32 @func(i32 %op8)
  ret i32 0
}
```

1. 删去 `store i32 %arg0, i32* %op1`,`%op2 = load i32, i32* %op1`,将`%op3 = icmp sgt i32 %op2, 0`中的`%op2`用`%arg0`代替。
2. 删去`label6`中`store i32 0, i32* %op1`与`label7`中`%op8 = load i32, i32* %op1`,在`label7`中添加`%op9 = phi i32 [ %arg0, %label_entry ], [ 0, %label6 ]`
3. `store i32 1, i32* @globVar`未发生变化
4. 删去`store i32 2333, i32* %op1`,`%op6 = load i32, i32* %op1`，将`%op7 = call i32 @func(i32 %op6)`中`%op6`替换为`2333`
5. `%op8 = load i32, i32* @globVar`未发生变化

对于*1,2,4*，由于`%op1`与`2333`是局部变量，会在函数`generate_phi()`中被加入`global_live_var_name`,同时在对应支配边界插入`phi`指令。在`fun`函数中，`%op1`出现在块`label_entry`,`label6`,`label7`中，而$DF(label6) = label7$,所以会在`label7`插入`phi(%op1)`函数,`2333`出现在`label4`中，而$DF(label4) = \emptyset$,所以不会插入`phi`函数。

在函数`rename()`中对于`phi`，会将左值压入栈（`var_val_stack`）用作后续替换；对于`store`，会将右值压栈（`var_val_stack`）取代左值；对于`load`指令，如果在`var_val_stack`中找到变量，则将已经记录的左值取代该变量。然后清除冗余的`load`与`store`指令

对于*3,5*，由于`globVar`是全局变量,不会被处理，因为其不在任何一个块内被定义，不能根据块的支配关系定义`phi`函数。程序代码体现为`if (!IS_GLOBAL_VARIABLE(l_val) && !IS_GEP_INSTR(l_val))`。

4. 指出放置phi节点的代码，并解释是如何使用支配树的信息的。（需要给出代码中的成员变量或成员函数名称）

   从支配树获取支配边界信息，并在对应位置插入 phi 指令:

```
   std::map<std::pair<BasicBlock *, Value *>, bool> bb_has_var_phi; //
bb has phi for var
    for (auto var : global_live_var_name) {
        std::vector<BasicBlock *> work_list;
        work_list.assign(live_var_2blocks[var].begin(),
live_var_2blocks[var].end());
        for (int i = 0; i < work_list.size(); i++) {
            auto bb = work_list[i];
```

```
            for (auto bb_dominance_frontier_bb : dominators_->get_dominance_frontier(bb)) {
                if (bb_has_var_phi.find({bb_dominance_frontier_bb, var}) == bb_has_var_phi.end()) {
                    auto phi =
                        PhiInst::create_phi(var->get_type()->get_pointer_element_type(), bb_dominance_frontier_bb);
                    phi->set_lval(var);
                    bb_dominance_frontier_bb->add_instr_begin(phi);
                    work_list.push_back(bb_dominance_frontier_bb);
                    bb_has_var_phi[{bb_dominance_frontier_bb, var}] = true;
                }
            }
        }
    }
```

为 lval 对应的 phi 指令参数补充完整

```
    for (auto succ_bb : bb->get_succ_basic_blocks()) {
        for (auto &instr1 : succ_bb->get_instructions()) {
            auto instr = &instr1;
            if (instr->is_phi()) {
                auto l_val = static_cast<PhiInst *>(instr)->get_lval();
                if (var_val_stack.find(l_val) != var_val_stack.end()) {
                    assert(var_val_stack[l_val].size() != 0);
                    static_cast<PhiInst *>(instr)->add_phi_pair_operand(var_val_stack[l_val].back(), bb);
                }
            }
        }
    }
```

`work-list`存储将要访问的块，初始化为为变量`a`被计算的`block`集合，第一层循环遍历`work-list`所有块；

第二层循环`for (auto bb_dominance_frontier_bb : dominators_->get_dominance_frontier(bb))`,其中`bb_dominance_frontier_bb`存储块`B`对应的支配边界集合$DF(B)$,算法会将`phi`插入所有支配边界的块（`create_phi`），同时将这些块加入`work-list`

以`fun`函数`%op9 = phi i32 [ %arg0, %label_entry ], [ 0, %label6 ]`为例：

在`fun`函数中，对于`%op1`，它会将`worlist`初始化为`label_entry`,`label6`,`label7`，$DF(label\_entry) = \emptyset$，然后从`worlist`移除`label_entry`,而$DF(label6) = label7$,所以会在`label7`插入`phi(%op1)`函数，然后从`worlist`移除`label_entry`,并将`label7`加入`worlist`，而$DF(label7) = \emptyset$

5. 算法是如何选择`value`(变量最新的值)来替换`load`指令的？（描述清楚对应变量与维护该变量的位置）

在函数`generate_phi()`中首先遍历块中的`load`函数,找到所有不是全局变量的全局名字，将其加入集合`global_live_var_name`,同时将他们所属的块存入`live_var_2blocks`。然后对于所有找到的活跃变量在对应支配边界插入`phi`指令。

在函数`rename()`中对于`phi`，会将左值与`phi`参数记录在字典栈（`var_val_stack`）用作后续替换；对于`store`，会将变量与`value`(变量最新的值)压入对应字典栈（`var_val_stack`），字典栈最顶端永远存储最新的值；对于`load`指令，如果在`var_val_stack`中找到变量，则`pop`出字典栈（`var_val_stack`）对应栈的顶端元素，然后用`replace_all_use_with`用取代该变量。然后清除冗余的`load`与`store`指令,冗余指令存储在`wait_delete`中。

代码阅读总结

此次实验有什么收获

了解SSA优化的基本思想与基础概念：支配性，`phi`节点等

了解了部分优化思路：如何利用UD链删除冗余的`store`,`load`指令等