CS 201
Project 3: Liveness Analysis
Report

Huy Tran & Rajat M. Jain

Algorithm's Summary

The approach to perform Liveness Analysis on a Basic Block is as follow:

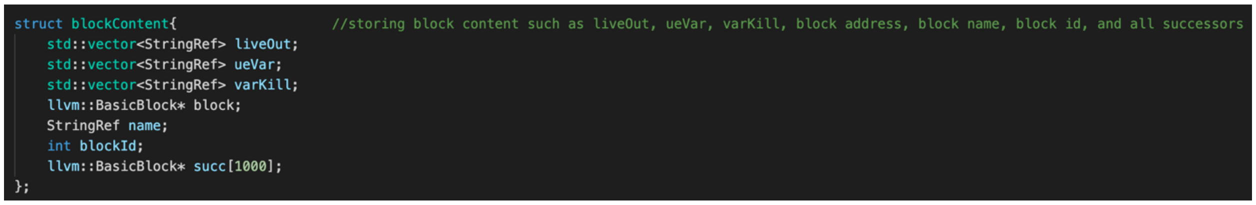
1. Our first task is to iterate through all the basic blocks and fetch each basic block in the CFG.
2. From each basic block, we not only store the basic block but we also store it's name and blockId for tracking purposes.
3. Now that we receive information about the basic block, our next task to iterate through the instructions executed in the basic block.
4. We then store the 2 operands used in the instruction.
5. This is followed by checking the nature of our instruction which could be load, store, addition, subtraction, multiplication, or division.
6. For these instructions we add the operands on the right side of the '=' to our Upward Exposed Variables and the variable on the left of '=' to our Variable Kill Set.
7. After constructing our UEVar, VarKill we can proceed to compute the LiveOut based on the formula:

$$\text{LIVEOUT}(N) = \cup_{X \in succ(N)} (\text{LIVEOUT}(X) - \text{VARKILL}(X) \cup \text{UEVAR}(X))$$

8. To compute the LiveOut we iterate through all the blocks we saved for our reference and use the successors of the block to apply the formula.
9. Finally, we print out by iterating through the structure we created to keep track of the UEVar, VarKill and LiveOut of each block.

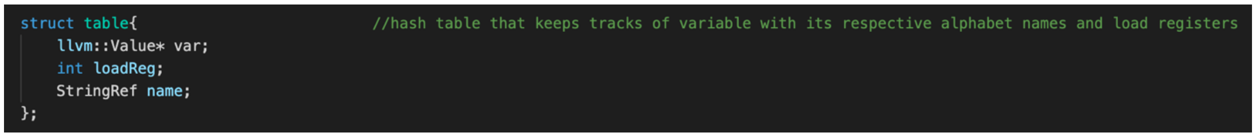
Data Structures used:

1. blockContent : A structure defined to store the BasicBlock itself and important variables like liveOut, ueVar and varKill in the form of StringRef. We also give each block a blockId and a list of its successor basic blocks.



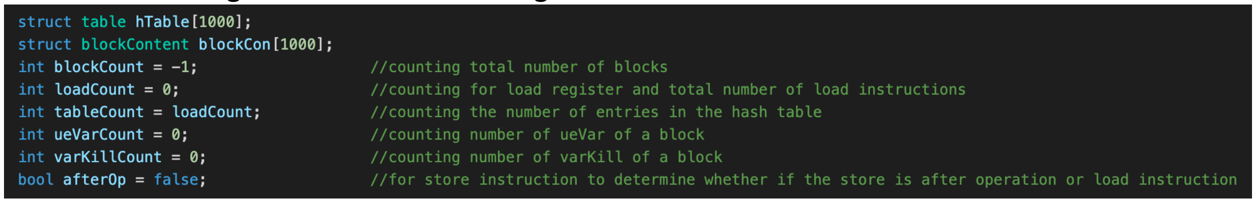
```
struct blockContent{           //storing block content such as liveOut, ueVar, varKill, block address, block name, block id, and all successors
    std::vector<StringRef> liveOut;
    std::vector<StringRef> ueVar;
    std::vector<StringRef> varKill;
    llvm::BasicBlock* block;
    StringRef name;
    int blockId;
    llvm::BasicBlock* succ[1000];
};
```

2. table: This is a hash table used to keep track of the variable with its respective loading register used for operations.



```
struct table{                  //hash table that keeps tracks of variable with its respective alphabet names and load registers
    llvm::Value* var;
    int loadReg;
    StringRef name;
};
```

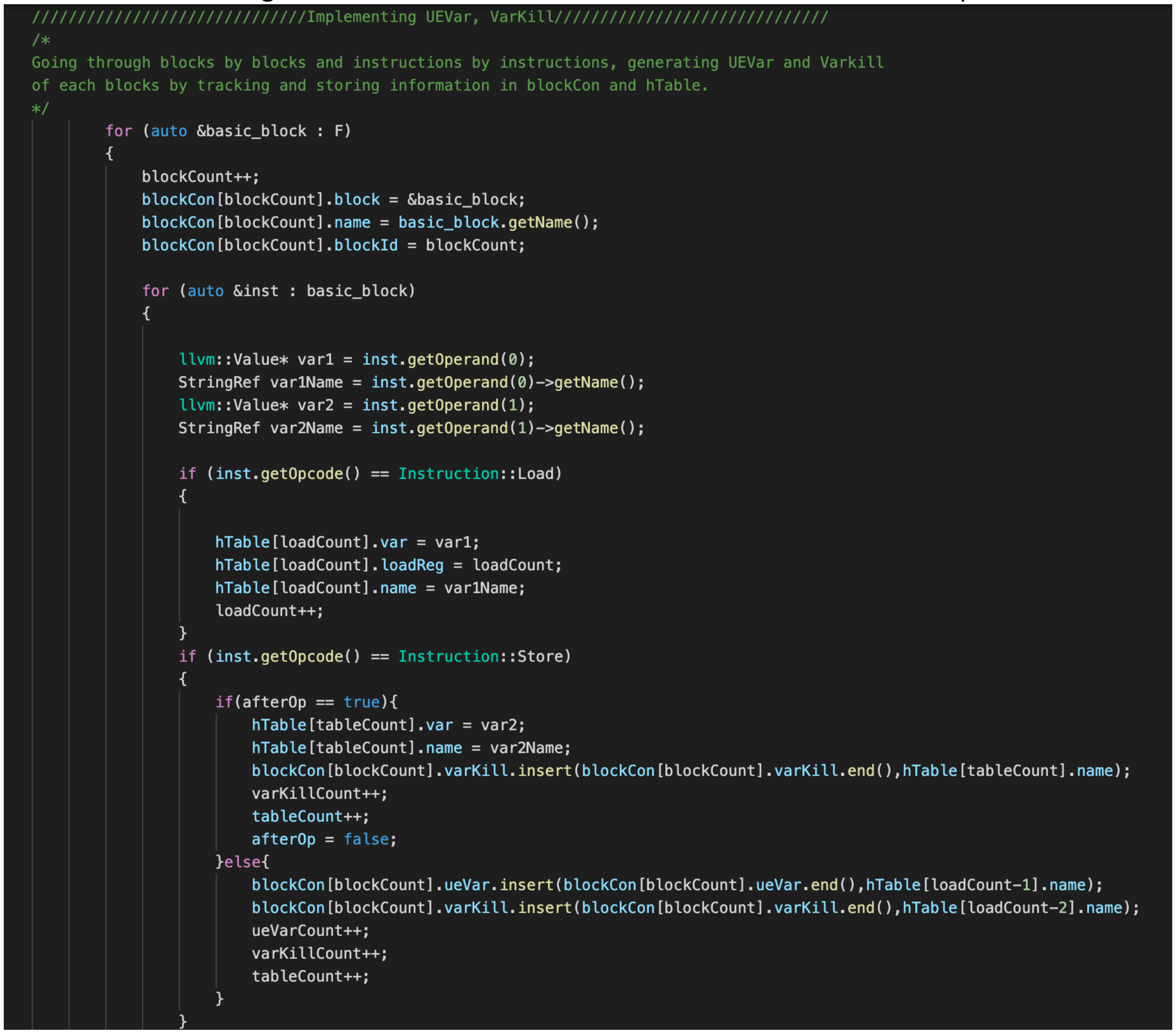
3. Indexing and tracking variables: We define a few more variables to assist our algorithm in indexing and condition checking.



```
struct table hTable[1000];
struct blockContent blockCon[1000];
int blockCount = -1;                    //counting total number of blocks
int loadCount = 0;                      //counting for load register and total number of load instructions
int tableCount = loadCount;             //counting the number of entries in the hash table
int ueVarCount = 0;                     //counting number of ueVar of a block
int varKillCount = 0;                   //counting number of varKill of a block
bool afterOp = false;                   //for store instruction to determine whether if the store is after operation or load instruction
```

Explanation:

1. Implementation of computing UEVar and VarKill: We loop through the basic blocks and first increment the blockCount and then use blockCon of type blockContent to store the basic block itself and it's name and blockId which is the blockCount. The inner loop then iterates through all the instructions in the basic block and fetches the operands for us.

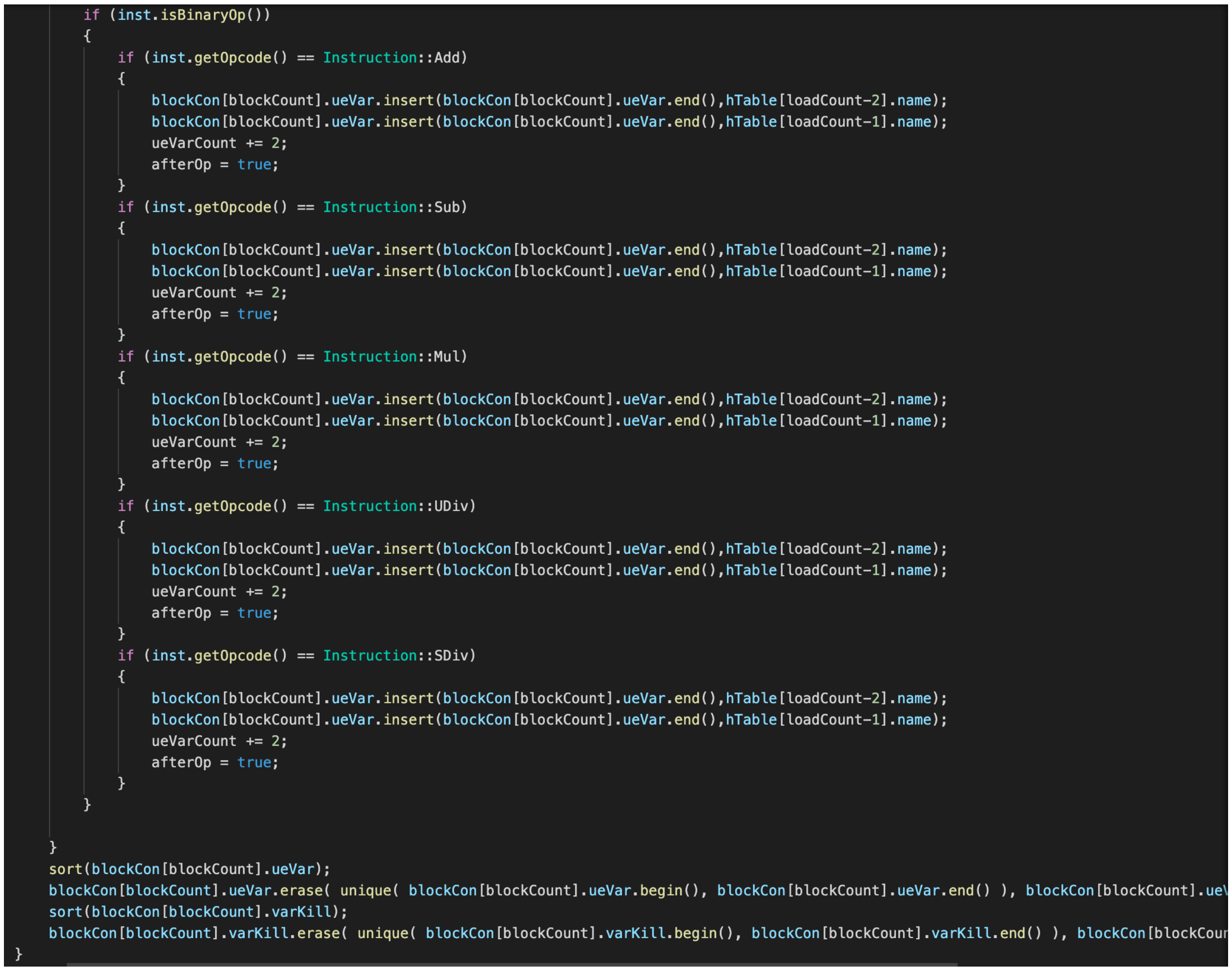


```
////////////////////////////Implementing UEVar, VarKill////////////////////////////
/*
Going through blocks by blocks and instructions by instructions, generating UEVar and Varkill
of each blocks by tracking and storing information in blockCon and hTable.
*/
        for (auto &basic_block : F)
        {
            blockCount++;
            blockCon[blockCount].block = &basic_block;
            blockCon[blockCount].name = basic_block.getName();
            blockCon[blockCount].blockId = blockCount;

            for (auto &inst : basic_block)
            {

                llvm::Value* var1 = inst.getOperand(0);
                StringRef var1Name = inst.getOperand(0)->getName();
                llvm::Value* var2 = inst.getOperand(1);
                StringRef var2Name = inst.getOperand(1)->getName();

                if (inst.getOpcode() == Instruction::Load)
                {

                    hTable[loadCount].var = var1;
                    hTable[loadCount].loadReg = loadCount;
                    hTable[loadCount].name = var1Name;
                    loadCount++;
                }
                if (inst.getOpcode() == Instruction::Store)
                {
                    if(afterOp == true){
                        hTable[tableCount].var = var2;
                        hTable[tableCount].name = var2Name;
                        blockCon[blockCount].varKill.insert(blockCon[blockCount].varKill.end(),hTable[tableCount].name);
                        varKillCount++;
                        tableCount++;
                        afterOp = false;
                    }else{
                        blockCon[blockCount].ueVar.insert(blockCon[blockCount].ueVar.end(),hTable[loadCount-1].name);
                        blockCon[blockCount].varKill.insert(blockCon[blockCount].varKill.end(),hTable[loadCount-2].name);
                        ueVarCount++;
                        varKillCount++;
                        tableCount++;
                    }
                }
```

The statements in the block when the instruction is a load instruction is to store variable address, name and loading register in hash table. Loading register will always start %0 and go on regard

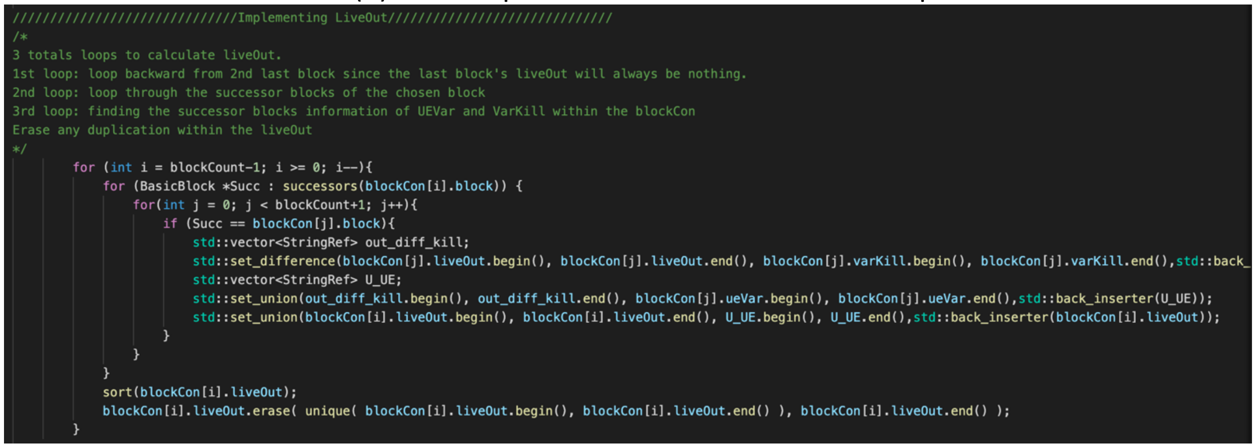
to the number of load instruction. We want to keep track of the load register because operation instruction only use these registers as operands.
The if statement in the block when the instruction is a store instruction is for storing after operations. For example, c = a + b where first we load a and b, after that we add a and b, and finally the result which is c will be stored, else statement is for a special condition where for example, a = b which is load and then store.

We then insert the UEVar to their respective basic block by accessing blockCon[blockCount] when the instruction is a binary operation.



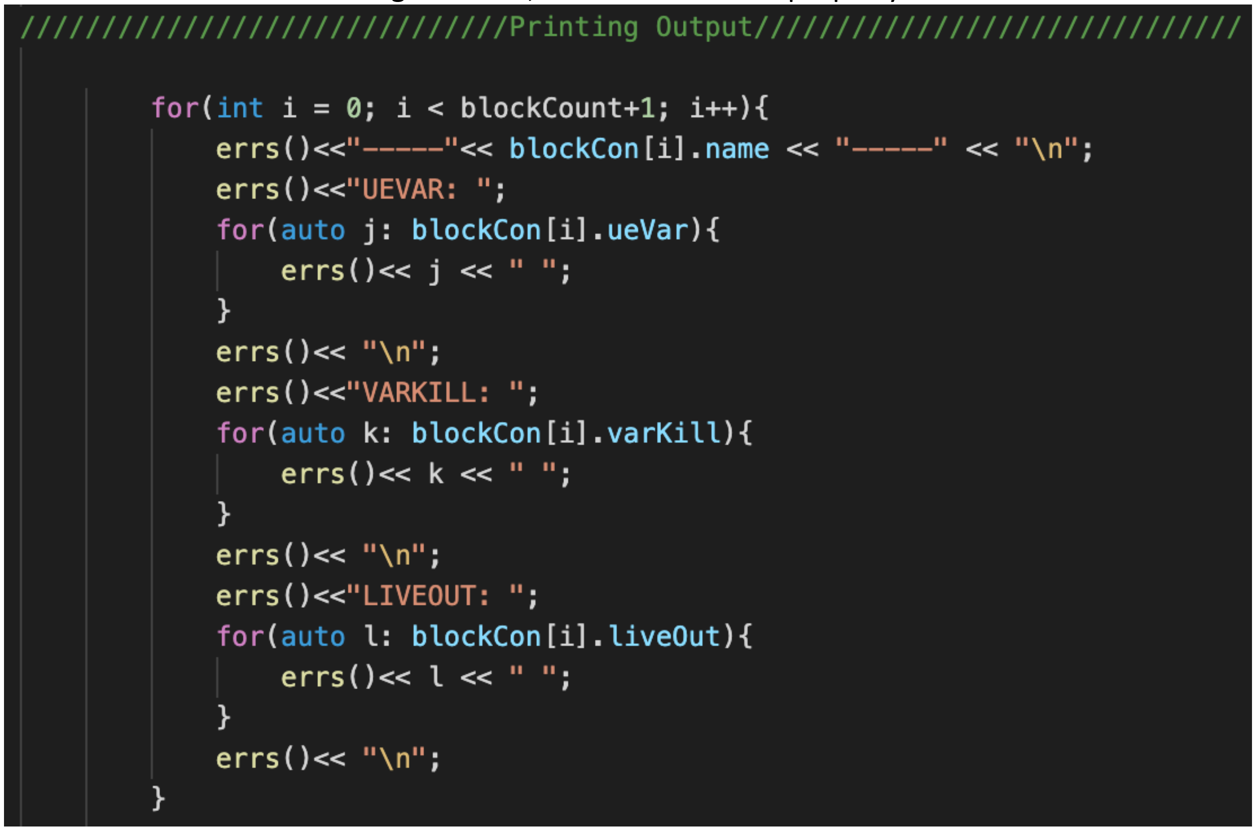
```
        if (inst.isBinaryOp())
        {
            if (inst.getOpcode() == Instruction::Add)
            {
                blockCon[blockCount].ueVar.insert(blockCon[blockCount].ueVar.end(),hTable[loadCount-2].name);
                blockCon[blockCount].ueVar.insert(blockCon[blockCount].ueVar.end(),hTable[loadCount-1].name);
                ueVarCount += 2;
                afterOp = true;
            }
            if (inst.getOpcode() == Instruction::Sub)
            {
                blockCon[blockCount].ueVar.insert(blockCon[blockCount].ueVar.end(),hTable[loadCount-2].name);
                blockCon[blockCount].ueVar.insert(blockCon[blockCount].ueVar.end(),hTable[loadCount-1].name);
                ueVarCount += 2;
                afterOp = true;
            }
            if (inst.getOpcode() == Instruction::Mul)
            {
                blockCon[blockCount].ueVar.insert(blockCon[blockCount].ueVar.end(),hTable[loadCount-2].name);
                blockCon[blockCount].ueVar.insert(blockCon[blockCount].ueVar.end(),hTable[loadCount-1].name);
                ueVarCount += 2;
                afterOp = true;
            }
            if (inst.getOpcode() == Instruction::UDiv)
            {
                blockCon[blockCount].ueVar.insert(blockCon[blockCount].ueVar.end(),hTable[loadCount-2].name);
                blockCon[blockCount].ueVar.insert(blockCon[blockCount].ueVar.end(),hTable[loadCount-1].name);
                ueVarCount += 2;
                afterOp = true;
            }
            if (inst.getOpcode() == Instruction::SDiv)
            {
                blockCon[blockCount].ueVar.insert(blockCon[blockCount].ueVar.end(),hTable[loadCount-2].name);
                blockCon[blockCount].ueVar.insert(blockCon[blockCount].ueVar.end(),hTable[loadCount-1].name);
                ueVarCount += 2;
                afterOp = true;
            }
        }

    }
    sort(blockCon[blockCount].ueVar);
    blockCon[blockCount].ueVar.erase( unique( blockCon[blockCount].ueVar.begin(), blockCon[blockCount].ueVar.end() ), blockCon[blockCount].ue
    sort(blockCon[blockCount].varKill);
    blockCon[blockCount].varKill.erase( unique( blockCon[blockCount].varKill.begin(), blockCon[blockCount].varKill.end() ), blockCon[blockCoun
}
```

2. Impementation of LiveOut: To compute the LiveOut of a block we follow the designated formula by first taking the difference between the LiveOut(X) and VarKill(X) and then union it with UEVar(X). We keep a track of the successors to implement the formula.



```
////////////////////////////////Implementing LiveOut////////////////////////////////
/*
3 totals loops to calculate liveOut.
1st loop: loop backward from 2nd last block since the last block's liveOut will always be nothing.
2nd loop: loop through the successor blocks of the chosen block
3rd loop: finding the successor blocks information of UEVar and VarKill within the blockCon
Erase any duplication within the liveOut
*/
        for (int i = blockCount-1; i >= 0; i--){
            for (BasicBlock *Succ : successors(blockCon[i].block)) {
                for(int j = 0; j < blockCount+1; j++){
                    if (Succ == blockCon[j].block){
                        std::vector<StringRef> out_diff_kill;
                        std::set_difference(blockCon[j].liveOut.begin(), blockCon[j].liveOut.end(), blockCon[j].varKill.begin(), blockCon[j].varKill.end(),std::back_
                        std::vector<StringRef> U_UE;
                        std::set_union(out_diff_kill.begin(), out_diff_kill.end(), blockCon[j].ueVar.begin(), blockCon[j].ueVar.end(),std::back_inserter(U_UE));
                        std::set_union(blockCon[i].liveOut.begin(), blockCon[i].liveOut.end(), U_UE.begin(), U_UE.end(),std::back_inserter(blockCon[i].liveOut));
                    }
                }
            }
            sort(blockCon[i].liveOut);
            blockCon[i].liveOut.erase( unique( blockCon[i].liveOut.begin(), blockCon[i].liveOut.end() ), blockCon[i].liveOut.end() );
        }
```

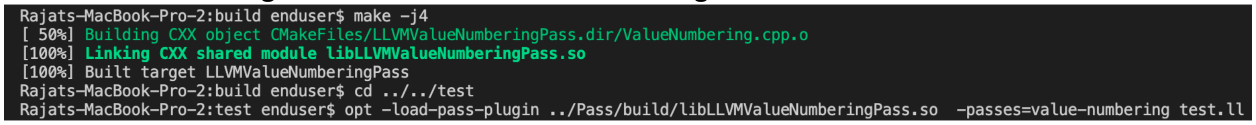
3. Printing the output: Ultimately we use errs() to print the output by iterating through blockCon and accessing its ueVar, varKill and liveOut property.



```
//////////////////////////////Printing Output//////////////////////////////

        for(int i = 0; i < blockCount+1; i++){
            errs()<<"-----"<< blockCon[i].name << "-----" << "\n";
            errs()<<"UEVAR: ";
            for(auto j: blockCon[i].ueVar){
                errs()<< j << " ";
            }
            errs()<< "\n";
            errs()<<"VARKILL: ";
            for(auto k: blockCon[i].varKill){
                errs()<< k << " ";
            }
            errs()<< "\n";
            errs()<<"LIVEOUT: ";
            for(auto l: blockCon[i].liveOut){
                errs()<< l << " ";
            }
            errs()<< "\n";
        }
```

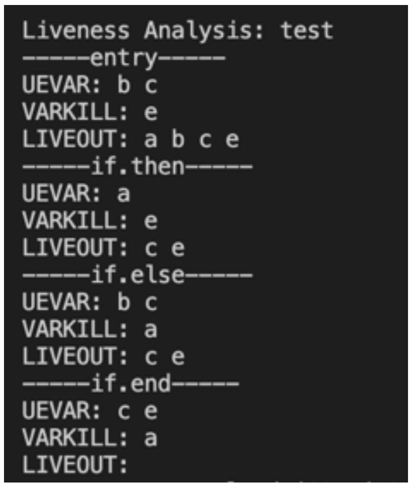
Commands:

The following are the commands to run the algorithm on test.c



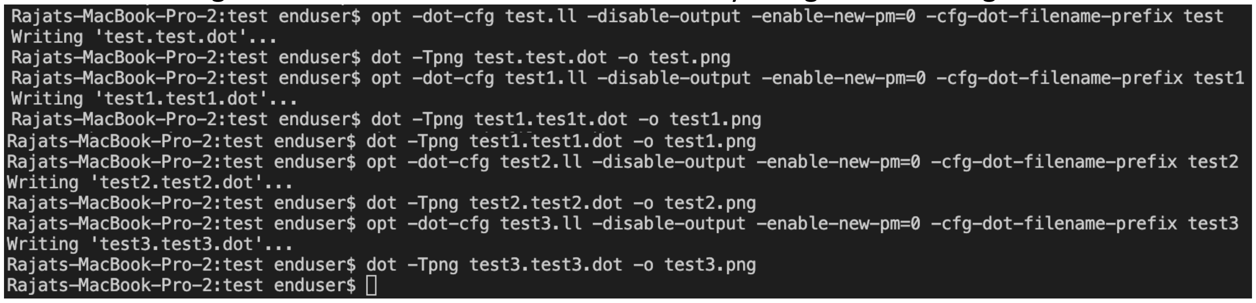
```
Rajats-MacBook-Pro-2:build enduser$ make -j4
[ 50%] Building CXX object CMakeFiles/LLVMValueNumberingPass.dir/ValueNumbering.cpp.o
[100%] Linking CXX shared module libLLVMValueNumberingPass.so
[100%] Built target LLVMValueNumberingPass
Rajats-MacBook-Pro-2:build enduser$ cd ../../test
Rajats-MacBook-Pro-2:test enduser$ opt -load-pass-plugin ../Pass/build/libLLVMValueNumberingPass.so  -passes=value-numbering test.ll
```

Output:



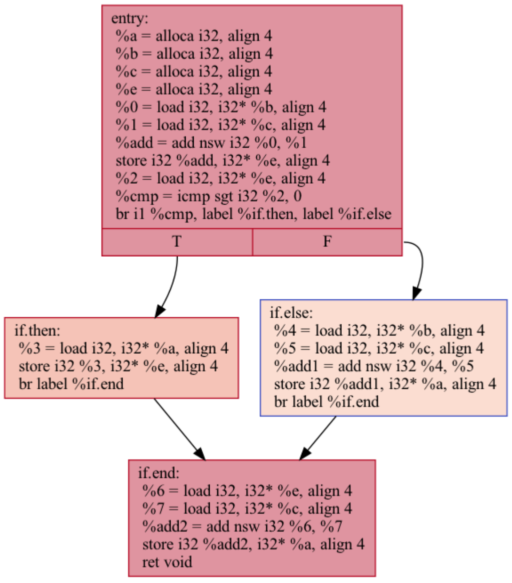
```
Liveness Analysis: test
-----entry-----
UEVAR: b c
VARKILL: e
LIVEOUT: a b c e
-----if.then-----
UEVAR: a
VARKILL: e
LIVEOUT: c e
-----if.else-----
UEVAR: b c
VARKILL: a
LIVEOUT: c e
-----if.end-----
UEVAR: c e
VARKILL: a
LIVEOUT:
```

We also generate the CFG for better reference by using the following commands:



```
Rajats-MacBook-Pro-2:test enduser$ opt -dot-cfg test.ll -disable-output -enable-new-pm=0 -cfg-dot-filename-prefix test
Writing 'test.test.dot'...
Rajats-MacBook-Pro-2:test enduser$ dot -Tpng test.test.dot -o test.png
Rajats-MacBook-Pro-2:test enduser$ opt -dot-cfg test1.ll -disable-output -enable-new-pm=0 -cfg-dot-filename-prefix test1
Writing 'test1.test1.dot'...
Rajats-MacBook-Pro-2:test enduser$ dot -Tpng test1.tes1t.dot -o test1.png
Rajats-MacBook-Pro-2:test enduser$ dot -Tpng test1.test1.dot -o test1.png
Rajats-MacBook-Pro-2:test enduser$ opt -dot-cfg test2.ll -disable-output -enable-new-pm=0 -cfg-dot-filename-prefix test2
Writing 'test2.test2.dot'...
Rajats-MacBook-Pro-2:test enduser$ dot -Tpng test2.test2.dot -o test2.png
Rajats-MacBook-Pro-2:test enduser$ opt -dot-cfg test3.ll -disable-output -enable-new-pm=0 -cfg-dot-filename-prefix test3
Writing 'test3.test3.dot'...
Rajats-MacBook-Pro-2:test enduser$ dot -Tpng test3.test3.dot -o test3.png
Rajats-MacBook-Pro-2:test enduser$ 
```

CFG for test.c:



```
entry:
 %a = alloca i32, align 4
 %b = alloca i32, align 4
 %c = alloca i32, align 4
 %e = alloca i32, align 4
 %0 = load i32, i32* %b, align 4
 %1 = load i32, i32* %c, align 4
 %add = add nsw i32 %0, %1
 store i32 %add, i32* %e, align 4
 %2 = load i32, i32* %e, align 4
 %cmp = icmp sgt i32 %2, 0
 br i1 %cmp, label %if.then, label %if.else
```

```
if.then:
 %3 = load i32, i32* %a, align 4
 store i32 %3, i32* %e, align 4
 br label %if.end
```

```
if.else:
 %4 = load i32, i32* %b, align 4
 %5 = load i32, i32* %c, align 4
 %add1 = add nsw i32 %4, %5
 store i32 %add1, i32* %a, align 4
 br label %if.end
```

```
if.end:
 %6 = load i32, i32* %e, align 4
 %7 = load i32, i32* %c, align 4
 %add2 = add nsw i32 %6, %7
 store i32 %add2, i32* %a, align 4
 ret void
```

CFG for 'test' function