Project Part - 03 - Liveness Analysis (LLVM) - Yeshwanth Kallati (862186252), Gowtham Tumati (862186477)

The primary objective of the project is to perform the Liveness analysis and to implement the data-flow analysis pass. The main idea of the implementation behind the code is we maintain three hashmaps, namely " `map<BasicBlock*, set<llvm::StringRef>> UpwardExposed;` " , " `map<BasicBlock*, set<llvm::StringRef>> VariableKill;` " , " `map<BasicBlock*, set<llvm::StringRef>> LOut;` ".

Code Walk through -

we start by the declaration of the hash maps for Upward Exposed Variables, VarKill, and LiveOut. In the next step the I/O are handled.
An output file with the 'filename.out' is created to allow the write operations of the program output to be written to this file.

```
ofstream createOutput(Function &F){
                         string filename = F.getParent()
->getSourceFileName();
                         int index = filename.find_last_of(".");
                         string sourcefile
=filename.substr(0,index);
                         //string c = ".out";
                         //file.append(c);
                         ofstream pathOut;
                         string outputfile = sourcefile + ".out";

                         pathOut.open(outputfile);
                         return pathOut;
                 }
```

We use the "`computeUEandVarKill()"` function to calculate the upward exposed variables and VarKill.

```
auto computeUpwardExposedandVarKill(Function &F,set<StringRef>
UpwardExposedVariable, set<StringRef> VarKill, BasicBlock &BB )
```

Input - This function takes the UEVAR and VarKill hashmaps along with the basic block as input and calculates the UEVar and VarKill for the basic blocks.

So, we in accordance with the load operation code, we perform a basic check and if the op code is 31, we modify the UEVar hashmap with respective values,
else if the op code is 32, we perform the appropriate insertion of values in the VarKill hashmap.

```
if (inst.getOpcode() == 31){
    UpwardExposed.insert(std::pair<BasicBlock*,
std::set<llvm::StringRef>>(&BB,UpwardExposedVariable));
}
```

```
if (inst.getOpcode() == 32) {
    VariableKill.insert(std::pair<BasicBlock*,
std::set<llvm::StringRef>>(&BB,VarKill));
}
```

And, after completing the calculation of UEVar and VarKill, we proceed to the computation of the LiveOut function. The `computeLiveOut` function primarily utilizes the UEVar and VarKill hashmaps. The main idea behind this function is that it utilizes the worklist approach to compute the LiveOut. We maintain a worklist and using this, we calculate the output and update it to the `resultSet` HashMap.

```
auto computeLOut(Function &F, map<BasicBlock*,
set<llvm::StringRef>> UpwardExposed, map<BasicBlock*,
set<llvm::StringRef>> VariableKill){
```

We loop over with variable starting from index 0 and all the up to `NSucc`, and for each iteration, we get the basic block pointer and also obtain successors LiveOut/VarKill/UEVar. We compute the values and update the results to the resultset Map.

```
for (int i = 0, NSucc = TInst->getNumSuccessors(); i < NSucc; i+
+) {
                                        BasicBlock* succ =
TInst->getSuccessor(i);// get BB pointer
                                        // get successor's
LIVEOUT/VARKILL/UEVAR
                                        set<llvm::StringRef>
LIVEOUT = LO.find(succ)->second;
                                        set<llvm::StringRef>
VarKill = VK.find(succ)->second;
                                        set<llvm::StringRef>
UEVar = UE.find(succ)->second;
                                        set<llvm::StringRef>
subtrSet (LIVEOUT);
                                        for
(set<llvm::StringRef>::iterator setIt = VarKill.begin(); setIt !
= VarKill.end(); setIt++) {

subtrSet.erase(*setIt);
                                        }

std::set_union(UEVar.begin(), UEVar.end(), subtrSet.begin(),

subtrSet.end(), std::inserter(resultSet, resultSet.begin()));
```

```
                                }
```

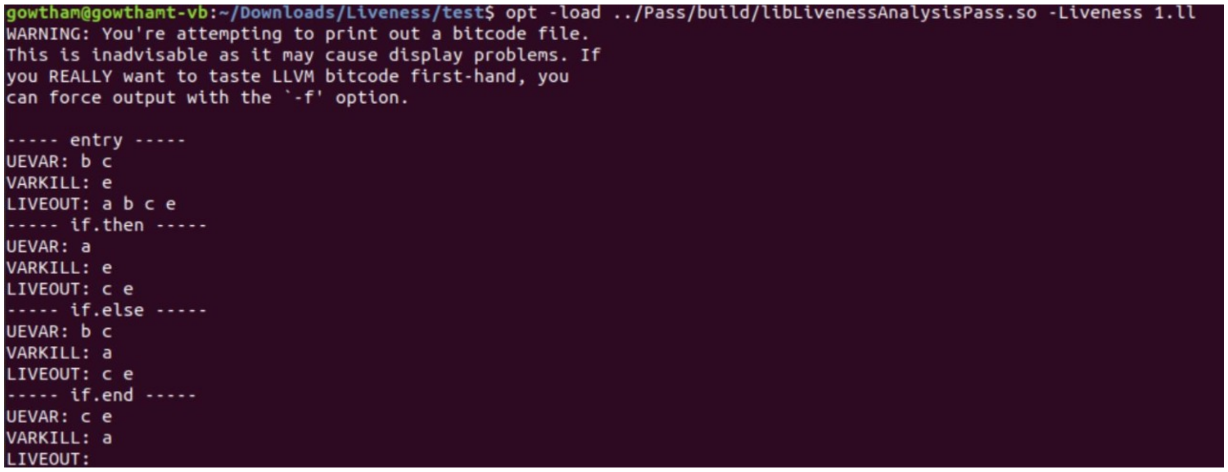
Towards the end of the loop, we check if the resultset has changed, and if changed, we add the predecessor to the worklist.

```
if (resultSet != originLIVEOUT) {// if changed, add predecessor
to worklist
                                        for (auto predIt =
pred_begin(tmp), predEnd = pred_end(tmp);
                                                        predIt !
= predEnd; predIt++) {
                                                BasicBlock* pred
= *predIt;

workList.push_back(pred);
                                        }
                                }
```

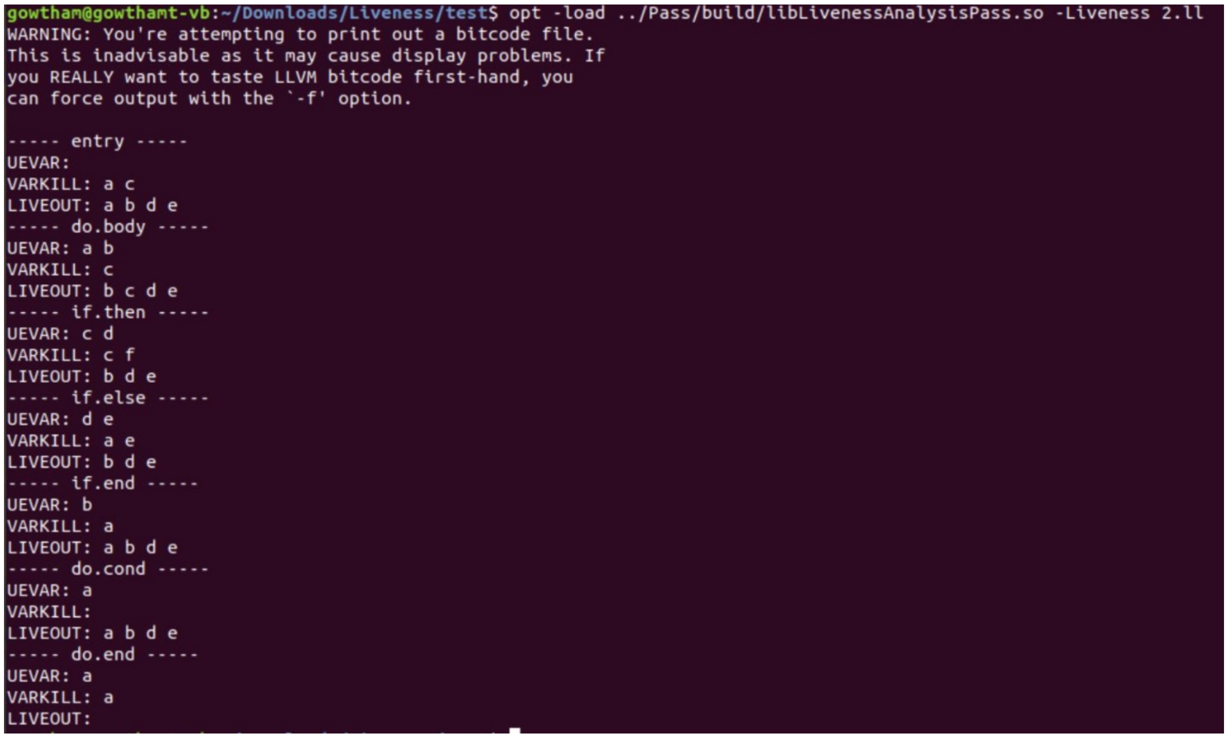
Now, the UEVar, VarKill, and LiveOut functions are computed and with the `printResult()` function, we direct the output as a human readable format (more intuitive) to the output file.

So, these are the outputs generated on “1.c” and “2.c” to the output files “test.out” and “test2.out” respectively.



```
gowtham@gowthamt-vb:~/Downloads/Liveness/test$ opt -load ../Pass/build/libLivenessAnalysisPass.so -Liveness 1.ll
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the `-f' option.

----- 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:
```



```
gowtham@gowthamt-vb:~/Downloads/Liveness/test$ opt -load ../Pass/build/libLivenessAnalysisPass.so -Liveness 2.ll
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the `-f' option.

----- entry -----
UEVAR:
VARKILL: a c
LIVEOUT: a b d e
----- do.body -----
UEVAR: a b
VARKILL: c
LIVEOUT: b c d e
----- if.then -----
UEVAR: c d
VARKILL: c f
LIVEOUT: b d e
----- if.else -----
UEVAR: d e
VARKILL: a e
LIVEOUT: b d e
----- if.end -----
UEVAR: b
VARKILL: a
LIVEOUT: a b d e
----- do.cond -----
UEVAR: a
VARKILL:
LIVEOUT: a b d e
----- do.end -----
UEVAR: a
VARKILL: a
LIVEOUT:
```