CS 201 Compiler Construction - Project 3: Liveness Analysis
Group Hengshuo

Hengshuo Zhang
hzhan402@ucr.edu

Zhaoze Sun
zsun114@ucr.edu

Demo video link
https://drive.google.com/file/d/1pAfRW4fVmWf0MMob1_CM1WiuU1GAxHy8/view?usp=sharing

The major data structures used in Liveness Analysis are as follows:

1. Basic Block: A basic block is a sequence of code that has a single entry point and a single exit point. It is the basic unit of a program's control flow graph (CFG).

2. Control Flow Graph: A CFG is a directed graph that represents the control flow of a program. Each node in the graph represents a basic block, and the edges represent the control flow between the basic blocks.

3. Data Flow Equations: The liveness analysis uses data flow equations to compute the live variables in a program. The data flow equations are used to propagate the liveness information from one basic block to another in the CFG.

The algorithm used for Liveness Analysis is as follows:

1. For each basic block in the CFG, initialize the UEVar, VarKill, and LiveOut sets to empty.

2. Traverse each instruction in the basic block, and if the instruction is a load, add the variable to the UEVar set if it is not already in the VarKill or UEVar set. If the instruction is a store, add the variable to the VarKill set and add the variables used in the instruction to the UEVar set.

3. Compute the LiveOut set for each basic block by taking the union of the LiveIn sets of its successor blocks.

4. Iterate through all the basic blocks until there is no change in the LiveOut sets. At each iteration, update the LiveOut set of each basic block by taking the union of the LiveIn sets of its successor blocks.

5. Compute the LiveIn set for each basic block by subtracting the VarKill set from the LiveOut set and taking the union of the UEVar set.

6. Repeat steps 4 and 5 until there is no change in the LiveIn sets.

7. The UEVar set for a basic block represents the variables that are used before they are defined in the block. The VarKill set represents the variables that are defined in the block. The LiveIn set represents the variables that are live on entry to the block, and the LiveOut set represents the variables that are live on exit from the block.

The pseudocode of algorithm

- Compute ***UEVAR(N)*** and ***VARKILL(N)***

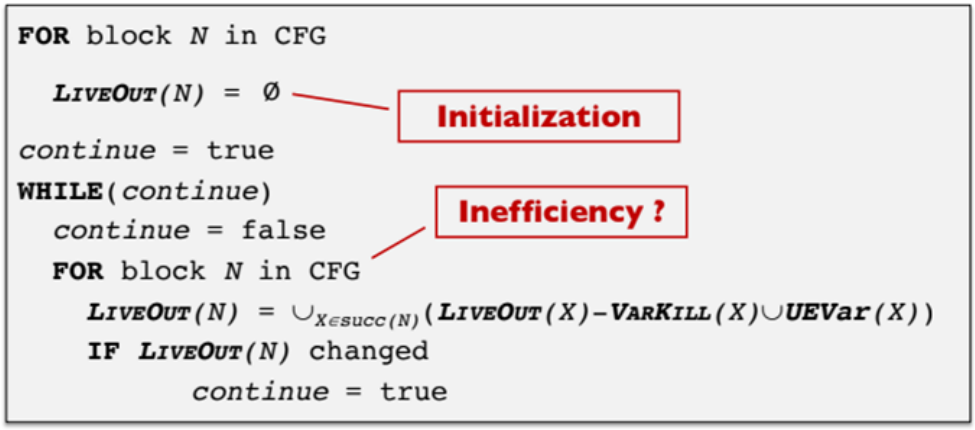
```
assume block N has operations o1, o2, …, ok
VarKill(N) = Ø
UEVar(N) = Ø
FOR i = 1 to k
    assume oi is "x ← y op z"
    IF y ∉ VarKill(N)
      UEVar(N) = UEVar(N) ∪ y
    IF z ∉ VarKill(N)
      UEVar(N) = UEVar(N) ∪ z
    VarKill(N) = VarKill(N) ∪ x
```

- **Complexity**: O(k) (assuming set member check takes constant time)
- **Direction**: Forward (backward is possible)

- **Algorithm**
  - STEP 1: Gather "summary sets"
    - Compute ***UEVAR(N)*** for each block ***N***
    - Compute ***VARKILL(N)*** for each block ***N***
  - STEP 2: Compute ***LIVEOUT(N)***
    - Evaluate equations iteratively over the CFG
    - Terminate when all ***LIVEOUT(N)*** sets stop changing.

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

- Basic Iterative Algorithm



Code Details Explained

Load instruction:

This part of a liveness analysis algorithm that analyzes a basic block of LLVM instructions to determine the set of variables that are used before they are defined (UEVAR set).

```
for (auto &inst: basic_block) {
    if (inst.getOpcode() == Instruction::Load) {
        // Get the name of the variable being loaded from memory.
        string var = string(inst.getOperand(0)->getName());

        // If the variable being loaded is not already in the VARKILL or UEVAR sets, add it to the UEVAR set.
        if (count(VarKill.begin(), VarKill.end(), var) == 0 &&
            count(UEVar.begin(), UEVar.end(), var) == 0) {
            UEVar.push_back(var);
        }
    }
```

The for loop iterates over each instruction in the basic block, and the if statement checks whether the current instruction is a load instruction. If it is, then the name of the variable being loaded from memory is extracted from the instruction's operand using the getName() function. The code then checks whether the variable being loaded is already present in the VARKILL set or the UEVAR set using the count() function. If the variable is not in either set, it is added to the UEVAR set by pushing it onto the end of the vector using the push_back() function. By the end of the loop, the UEVAR set will contain all the variables that are used before they are defined in the basic block.

Store instruction:

The main goal here is to update the UEVar and VarKill sets for the basic block.

```
}
if (inst.getOpcode() == Instruction::Store) {

    string var1, var2;

    // If the instruction is a "store" instruction,
    // get the names of the variables being stored (if any) from the operands.
    if (isa<ConstantInt>(inst.getOperand(0))) { // If the first operand is a constant, ignore this
        var1 = "";
    } else {
        for (int i = 0; i <
                    2; i++) { // Check whether each operand is a constant integer or a user-defined variable.
            Value *operand = inst.getOperand(i);
            if (isa<ConstantInt>(operand)) {
                if (i == 0) {
                    var1 = "";
                } else {
                    var2 = "";
                }
            } else {
                string operandName = string(dyn_cast<User>(operand)->getOperand(0)->getName());
                if (i == 0) {
                    var1 = operandName;
                } else {
                    var2 = operandName;
                }
            }
        }
```

First, the code extracts the names of the variables being stored (if any) from the operands of the instruction. If either operand is a constant integer, it is ignored. If either operand is a user-defined variable, its name is extracted and stored in the corresponding variable (var1 or var2).

```
            if (isa<ConstantInt>(
                    dyn_cast<User>(dyn_cast<User>(inst.getOperand(0))->getOperand(0))->getOperand(0)) ||
                isa<ConstantInt>(
                        dyn_cast<User>(dyn_cast<User>(inst.getOperand(0))->getOperand(1))->getOperand(0))) {
                var1 = "";
                var2 = "";
            }
        }
```

The variable being stored is then extracted from the second operand of the instruction, and it is added to the VarKill set if it is not already there.

```
            // If the variable is not in VarKill and not in UEVar, add it to UEVar.
            for (string var: {var1, var2}) {
                if (find(VarKill.begin(), VarKill.end(), var) == VarKill.end() &&
                    find(UEVar.begin(), UEVar.end(), var) == UEVar.end()) {
                    UEVar.emplace_back(var);
                }
            }

            // Add varName to VarKill.
            if (find(VarKill.begin(), VarKill.end(), varName) == VarKill.end()) {
                VarKill.emplace_back(varName);
            }
        }
    }
    // storing the results of the liveness analysis for a basic block
    (*blockUEVar)[blockName] = UEVar;
    (*blockVarKill)[blockName] = VarKill;
}
```

Next, the variables stored in var1 and var2 are checked to see if they are in VarKill or UEVar. If they are not, they are added to the UEVar set. Finally, the UEVar and VarKill sets for the current basic block are updated with the updated sets, so that they can be used in the liveness analysis for subsequent basic blocks.

Overall, this code block helps to update the liveness analysis sets for a basic block when the instruction is a "store" instruction, by adding the variables being stored to the VarKill set and the UEVar set as appropriate.

Computing the LiveOut:

```
// Initialize the liveOut vector for each basic block to an empty vector.
for (auto &basic_block: F) {
    (*blockLiveOut)[string(basic_block.getName())] = vector < string > {};
}
```

1. Initialize the LiveOut vector for each basic block to an empty vector.

```
vector <string> liveOut, liveOutTemp, unionSuccessor;

auto blockLiveOutIt = blockLiveOut->find(string(basic_block.getName()));
liveOut = blockLiveOutIt->second;

vector <string> varKill = (*blockVarKill)[string(basic_block.getName())];
vector <string> ueVar = (*blockUEVar)[string(basic_block.getName())];
```

2. Initialize liveOut, unionSuccessor, varKill, and ueVar for the current basic block.

```
// Compute the union of live variables across all successors
for (BasicBlock *Succ: successors(&basic_block)) {
    vector <string> diffTemp, unionTemp;

    auto blockLiveOutSuccIt = blockLiveOut->find(string(Succ->getName()));
    vector <string> liveOutSucc = blockLiveOutSuccIt->second;
    vector <string> varKillSucc = (*blockVarKill)[string(Succ->getName())];
    vector <string> ueVarSucc = (*blockUEVar)[string(Succ->getName())];

    set_difference(liveOutSucc.begin(), liveOutSucc.end(), varKillSucc.begin(), varKillSucc.end(),
                   std::back_inserter(diffTemp));

    set_union(diffTemp.begin(), diffTemp.end(), ueVarSucc.begin(), ueVarSucc.end(),
              std::inserter(unionTemp, unionTemp.begin()));
    unionSuccessor.insert(unionSuccessor.end(), unionTemp.begin(), unionTemp.end());
}
```

3. This code block is computing the union of the live variables across all successors of the current basic block. The loop iterates over all the successors of the current basic block using the successors function. For each successor, it retrieves the ‘liveOut’ set of the successor block and the ‘varKill’ and ‘ueVar’ sets associated with it. Then it calculates the difference between the ‘liveOut’ set and the ‘varKil’l set using the ‘set_difference’ function. The resulting set is stored in ‘diffTemp’. The code then calculates the union of ‘diffTemp’ and ‘ueVarSucc’ using the ‘set_union’ function. The resulting set is stored in ‘unionTemp’. Finally, the code inserts the elements of ‘unionTemp’ into the ‘unionSuccessor’ vector using the insert function. At the end of the loop, the ‘unionSuccessor’ vector contains the union of the live variables across all successors of the current basic block.

```
                // Sort and remove duplicates from the union of live variables across all successors
                std::sort(unionSuccessor.begin(), unionSuccessor.end());
                unionSuccessor.erase(std::unique(unionSuccessor.begin(), unionSuccessor.end()),
unionSuccessor.end());

                // Update liveOut if it's different from unionSuccessor.
                if (liveOut != unionSuccessor) {
                    changed = true;
                    blockLiveOutIt->second = unionSuccessor;
                }
```

4. The next following steps are to check whether the liveOut set of the current basic block is different from the unionSuccessor set. If they are different, then the liveOut set needs to be updated to the unionSuccessor set, and the loop needs to be repeated to check whether the liveOut sets of all basic blocks have stabilized. Updating liveOut is to sort the unionSuccessor vector. Then we are going to remove duplicates which means reorders the elements so that duplicates are all in the back, and returns an iterator pointing to the start of the range containing the unique elements. Finally we are going to check whether liveOut and unionSuccessor are different. If they are different, changed is set to true, indicating that at least one basic block's liveOut set has been updated.

```
// declares a lambda function that sorts the vector and prints out each element separated by spaces
auto print_vector = [](const vector <string> &vec) {
    for (const auto &elem: vec) {
        errs() << elem << " ";
    }
};
```

5. Declares a lambda function that sorts the vector and prints out each element separated by spaces.

Print:

```
// print out-put by format
for (auto &basic_block: F) {
    const auto &blockName = string(basic_block.getName());
    vector <string> UEVarTemp = (*blockUEVar)[blockName];
    std::sort(UEVarTemp.begin(), UEVarTemp.end());
    errs() << "----- " << blockName << " -----\n";
    errs() << "UEVAR: ";
    print_vector(UEVarTemp);
    errs() << "\n";

    vector <string> varKillTemp = (*blockVarKill)[blockName];
    std::sort(varKillTemp.begin(), varKillTemp.end());
    errs() << "VARKILL: ";
    print_vector(varKillTemp);
    errs() << "\n";

    vector <string> liveOutTemp = (*blockLiveOut)[blockName];
    std::sort(liveOutTemp.begin(), liveOutTemp.end());
    errs() << "LIVEOUT: ";
    print_vector(liveOutTemp);
    errs() << "\n";
}
```

Print out-put by format which is mentioned in the project3 requirement.

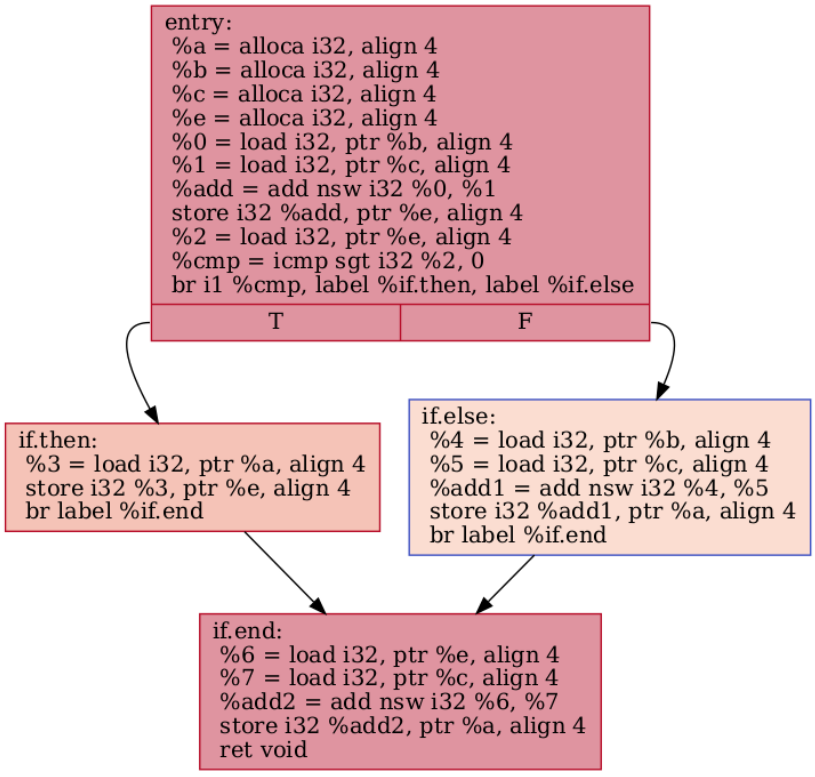
Demo Test:

```
// generate test.ll file for test.c using following command.

$ clang -S -fno-discard-value-names -emit-llvm test.c -o test.ll

// After generating test.ll, run the following command to test the LLVM Pass.

$ opt -load-pass-plugin ../Pass/build/libLLVMValueNumberingPass.so
-passes=value-numbering test.ll

// Generate .test.out file from test.ll file

$ opt -dot-cfg test.ll -disable-output -enable-new-pm=0

// Generate the cfg file and save it in .pdf file format

$ dot -Tpdf .test.dot -o test.pdf
```

—Output and CFG in the next few pages—

Test Case 1:

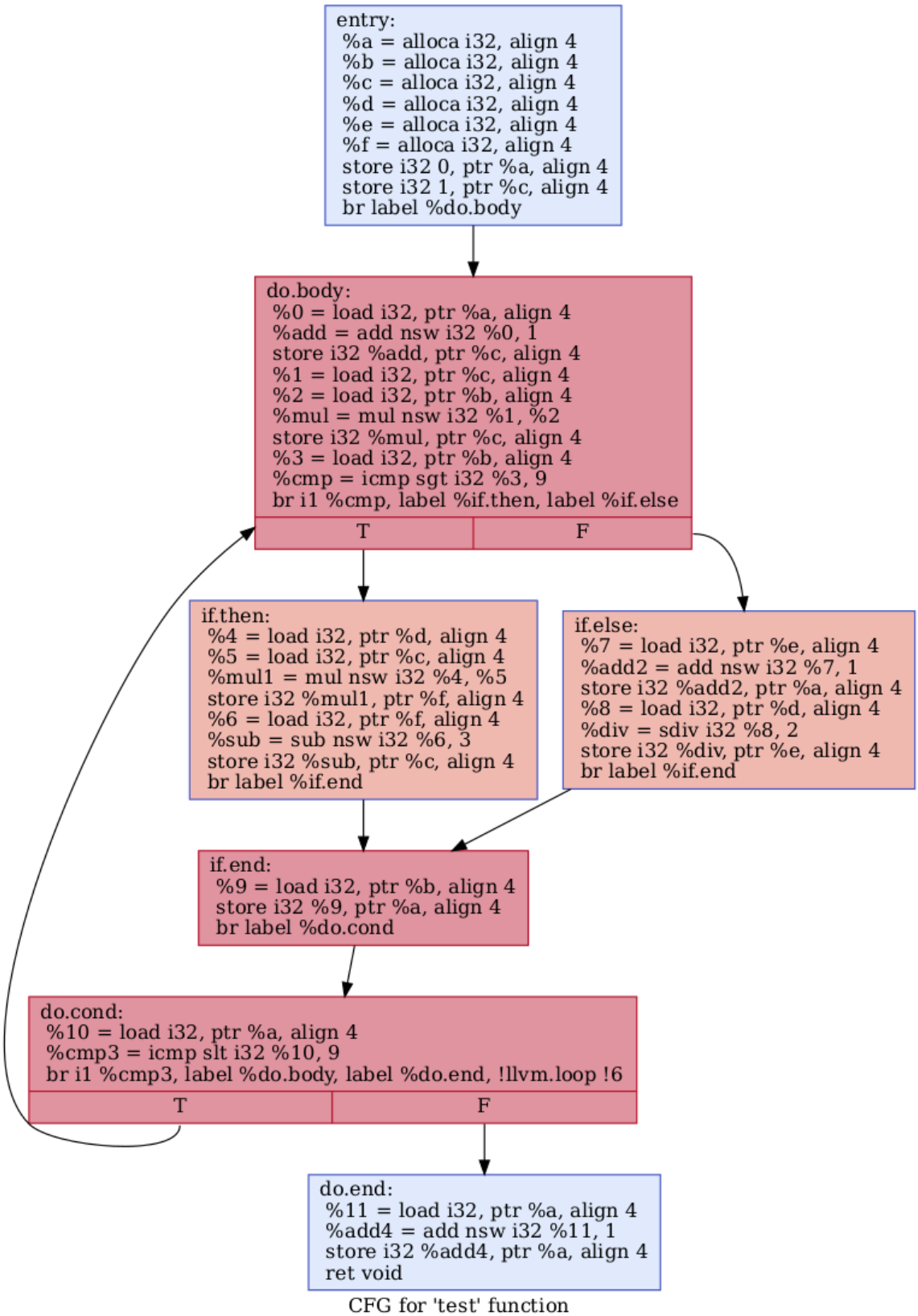
```
hengshuo@hengshuo-vm:~/data-flow-analyzer-hengshuo/test$ opt -load-pass-plugin
 ../Pass/build/libLLVMValueNumberingPass.so  -passes=value-numbering test1.ll
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the `-f' option.

LivenessAnalysis: 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:
```



CFG for 'test' function

Test Case 2:

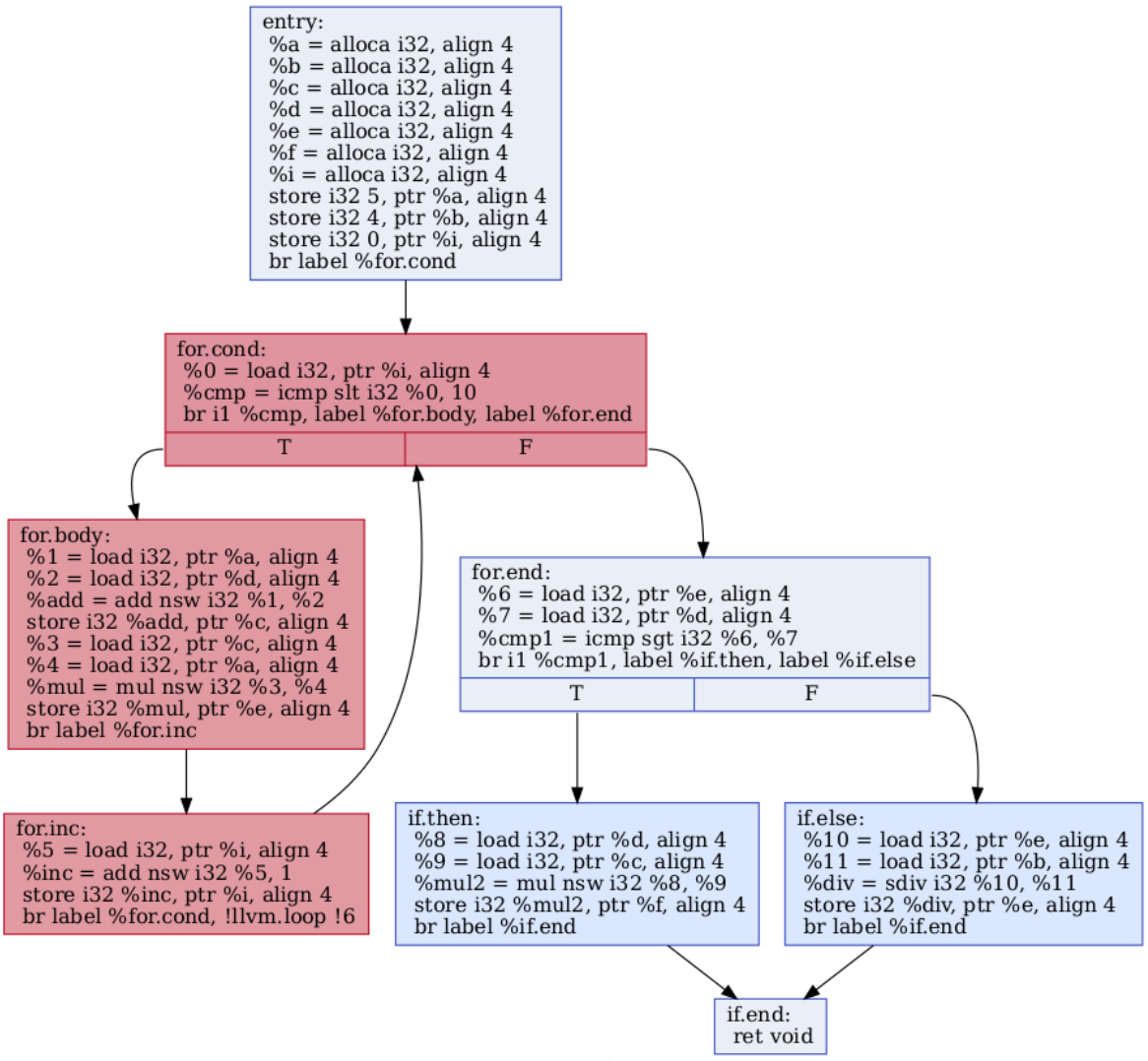
```
hengshuo@hengshuo-vm:~/data-flow-analyzer-hengshuo/test$ opt -load-pass-plugin
 ../Pass/build/libLLVMValueNumberingPass.so  -passes=value-numbering test2.ll
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the `-f' option.

LivenessAnalysis: test
----- 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:
```



CFG for 'test' function

Test Case 3:

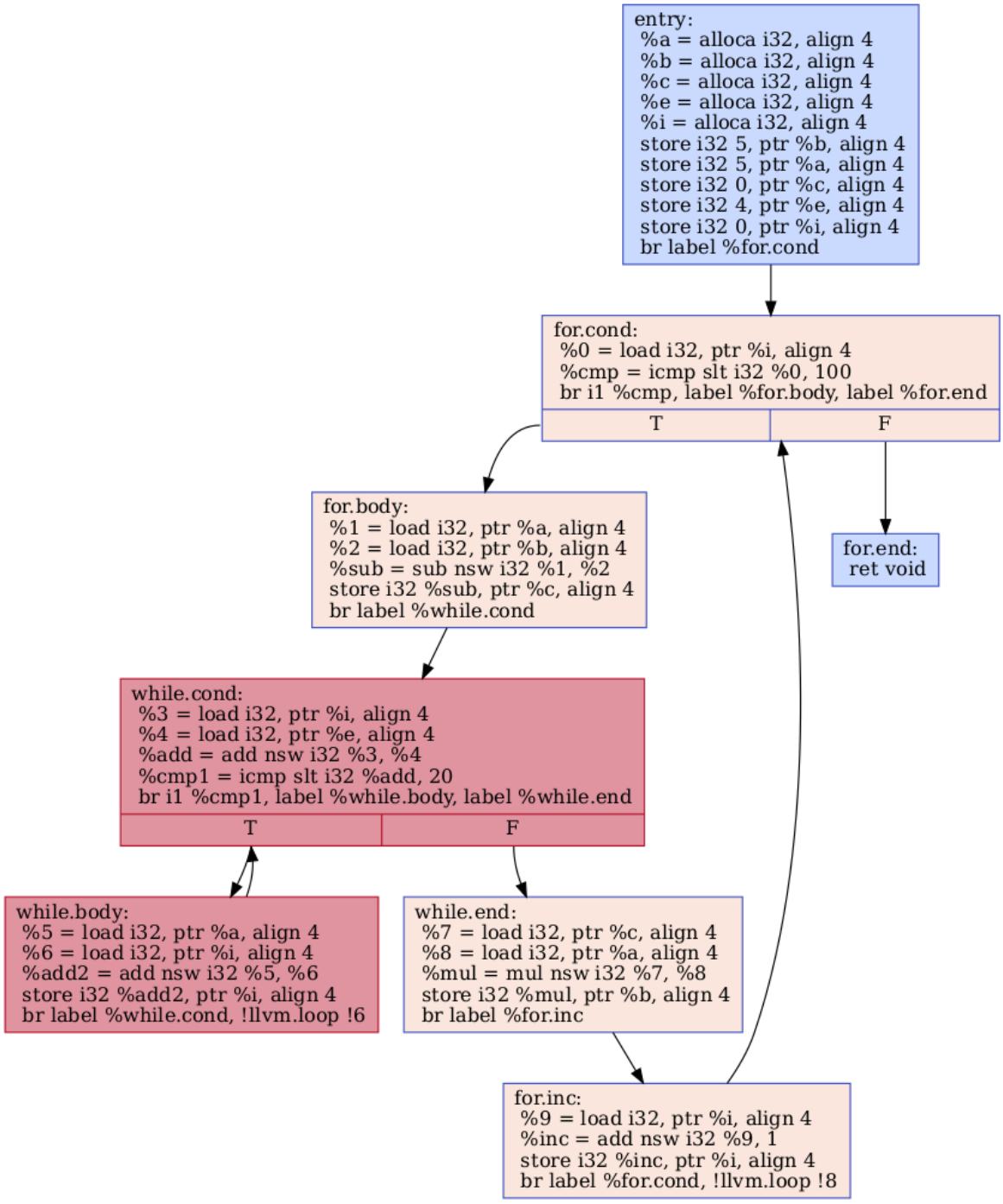
```
hengshuo@hengshuo-vm:~/data-flow-analyzer-hengshuo/test$ opt -load-pass-plugin
 ../Pass/build/libLLVMValueNumberingPass.so  -passes=value-numbering test3.ll
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the `-f' option.

LivenessAnalysis: test
----- entry -----
UEVAR:
VARKILL: a b i
LIVEOUT:  a b c d e i
----- for.cond -----
UEVAR: i
VARKILL:
LIVEOUT:  a b c d e i
----- for.body -----
UEVAR:  a d
VARKILL: c e
LIVEOUT:  a b c d e i
----- for.inc -----
UEVAR:  i
VARKILL: i
LIVEOUT:  a b c d e i
----- for.end -----
UEVAR: d e
VARKILL:
LIVEOUT:  b c d e
----- if.then -----
UEVAR:  c d
VARKILL: f
LIVEOUT:
----- if.else -----
UEVAR:  b e
VARKILL: e
LIVEOUT:
----- if.end -----
UEVAR:
VARKILL:
LIVEOUT:
```



CFG for 'test' function

Test Case 4:

```
hengshuo@hengshuo-vm:~/data-flow-analyzer-hengshuo/test$ opt -load-pass-plugin
 ../Pass/build/libLLVMValueNumberingPass.so  -passes=value-numbering test4.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.

LivenessAnalysis: test
----- entry -----
UEVAR:
VARKILL: a b c e i
LIVEOUT:  a b e i
----- for.cond -----
UEVAR: i
VARKILL:
LIVEOUT:  a b e i
----- for.body -----
UEVAR:  a b
VARKILL: c
LIVEOUT:  a c e i
----- while.cond -----
UEVAR: e i
VARKILL:
LIVEOUT:  a c e i
----- while.body -----
UEVAR:  a i
VARKILL: i
LIVEOUT:  a c e i
----- while.end -----
UEVAR:  a c
VARKILL: b
LIVEOUT:  a b e i
----- for.inc -----
UEVAR:  i
VARKILL: i
LIVEOUT:  a b e i
----- for.end -----
UEVAR:
VARKILL:
LIVEOUT:
```



CFG for 'test' function