Lisa Chen
862186585
Project 2: Local Value Numbering(LLVM)

Algorithm/Code Summary

To ensure the variables work, I had done the following changes to the terminal commands:

- `clang -S -fno-discard-value-names -emit-llvm test5.c -o test.ll`
- `opt -instnamer -load ../Pass/build/libLLVMValueNumberingPass.so -ValueNumbering test.ll`

This allows for getName() to work and to keep the original variables to match the operand labels (ex: a is %a).

1. General
    a. For simplicity purposes, all pointers are "treated the same" as the variables themselves. To do this, for example, "a.addr" will be added to the table as "a". Also, lookups for "a.addr" will be referenced as "a" as well.

```
37    string removeDotAddr(string word) {
38        size_t dotPos;
39        if ((dotPos = word.find_first_of(".")) != string::npos) {
40            return word.substr(0, dotPos);
41        }
42        return word;
43    }
```

    b. For constant values, they do not have names. Thus, there is a check to see if an operand has a name; if not, it is a constant and saved as an operand. For example, "10" will be referenced as "i32 10".

    The code below is just to get the operand itself when the check for having a name fails.

```
45     string getStrOper(Value* val) {
46         string operStr;
47         raw_string_ostream rs(operStr);
48         rs << *val;
49         // errs() << "stringstream " << operStr << "\n";
50         return operStr;
51     }
```

    c. There is an integer currentHash representing the next hash number to use. This is used and incremented as you add new hash key-value pairs.
    d. The hash table ws crated using unordered_map using string keys (for the operands/operand names) and integers (starting from 1 like in the lectures).

```
27    int getHashValue(unordered_map<string, int> hashmap, string key) {
28        if(hashmap.find(key) != hashmap.end()) {
29            return hashmap.find(key)->second;
30        }
31        else {
32            // errs() << "Could not find key in hashmap: " << key;
33            return -1;
34        }
35    }
```

Lisa Chen
862186585
Project 2: Local Value Numbering(LLVM)

e. The hash table is printed for verification purposes.

```
164                 errs() << "Keys\n";
165                 for (const auto& kv : hashmap) {
166                     errs() << kv.first << " " << kv.second <<  "\n";
167                 }
168                 errs() << "\n";
```

2. Load
    a. When the operation is a load, you will need to some operand's address to a given operand. That operand you are trying to load into the given operand will be looked up in the hash table to see if its value has been used before.
        i. Although there is a check to see if the value is already in the hash table, it should be always the case since the value it is loading should have already been seen in a store (no operands are used unless there is a value stored in the operands first).

```
66                  if(inst.getOpcode() == Instruction::Load){
67                      errs() << "This is Load"<<"\n";
68
69                      //load %var should be new, so add to hashmap with value it is pointing to
70                      string loadFromOp = removeDotAddr(inst.getOperand(0)->getName());
71                      errs() << "Load From Op: " << loadFromOp << "\n";
72                      int value = getHashValue(hashmap, loadFromOp);
73                      if (value == -1) {
74                          value = currentHash++;
75                      }
76                      hashmap[instName] = value;
77                  }
```

3. Store
    a. When the operation is a load, you will store an operand (constant or variable's value) into an operand. That operand's value you are trying to store will be looked up in the hash table to see if its value has been used before.
        i. Ex: store i32 %add2, i32* %e, align 4 //example of storing an operand value (%add2) to an operand (%e)
        ii. Ex: store i32 10, i32* %a, align 4 //example of storing a constant (100 to an operand (%a)
        iii. If the operand you are trying to store cannot be found, both the operand that you storing the value into and the operand you are trying to store will be added to the hash table with a new hash value (operand names being the keys).
            1. This scenario is possible for storing an operand's value to another operand if it is the first store calls in the class, for example (i.e. store i32 %a, i32* %a.addr, align 4).
            2. In the example, store i32 %add2, i32* %e, align 4, %e and %add2 will be assign as new keys pointing to currentHash (the next new hash number). Then, currentHash is incremented).
        iv. If the operand's value is found in the hash table (operand's value being the key), the hash value is returned. The operand you are storing the

value will be added to the hash table as the key with this hash value as the value.

1. In the example, store i32 %add2, i32* %e, align 4, %e will be assign as a new key pointing the value in which %add2 key points to in the hash table. So, if %add2 points to 1, %e also points to 1.\

```
79  if(inst.getOpcode() == Instruction::Store){
80      errs() << "This is Store"<<"\n";
81
82      string storeFrom = inst.getOperand(0)->getName();
83      //if constant, has no name
84      if (storeFrom == "") {
85          storeFrom= getStrOper(inst.getOperand(0));
86      }
87
88      //storeTo is always a variable (never a constant)
89      string storeTo = removeDotAddr(inst.getOperand(1)->getName());
90
91      int value = getHashValue(hashmap, storeFrom);
92      if (value == -1) {
93          value = currentHash++;
94      }
95      hashmap[storeTo] = value;
96      hashmap[storeFrom] = value;
97  }
```

4. Binary Operators
    a. If it is a binary operator (+, -, *, /), there's a check for which operator it is and then sets the opcodeString to the corresponding "+", "-", "*", "/".

```
105     errs() << "Op Code:" << inst.getOpcodeName()<<"\n";
106     if(inst.getOpcode() == Instruction::Add){
107         errs() << "This is Addition"<<"\n";
108         opcodeString = "+";
109     }
110     if(inst.getOpcode() == Instruction::Sub){
111         errs() << "This is Subtraction"<<"\n";
112         opcodeString = "-";
113     }
114     if(inst.getOpcode() == Instruction::Mul){
115         errs() << "This is Multiplication"<<"\n";
116         opcodeString = "*";
117     }
118     if(inst.getOpcode() == Instruction::UDiv || inst.getOpcode() == Instruction::SDiv){
119         errs() << "This is Division"<<"\n";
120         opcodeString = "/";
121     }
```

    b. After this check, the two operands in the operation are grabbed.
    c. There's a check to see if the operands' values are in the hash table. It is possible for an operand to not be in the hash table if the operand is a constant that has never been used.
        i. If it is a constant that has never been used, the constant is added as a key pointing to currentHash (the next new hash number). Then, currentHash is incremented.
        ii. Otherwise, it is a constant that has been used before or a operand with a value (which should be in the hash table already since all operands are initialized with a value stored before performing a binary operation).

Lisa Chen
862186585
Project 2: Local Value Numbering(LLVM)

```
123                 //chedk if either operands are constants (constants have no names)
124                 op1 = removeDotAddr(inst.getOperand(0)->getName());
125                 if (op1 == "") {
126                     op1 = getStrOper(inst.getOperand(0));
127                 }
128
129                 op2 = removeDotAddr(inst.getOperand(1)->getName());
130                 if (op2 == "") {
131                     op2 = getStrOper(inst.getOperand(1));
132                 }
133
134                 //check values in hashmap already
135                 int value1 = getHashValue(hashmap, op1);
136                 int value2 = getHashValue(hashmap, op2);
137                 //not  already in hash map, add to it
138                 if(value1 == -1) {
139                     value1 = currentHash;
140                     hashmap[op1] = currentHash++;
141                 }
142                 if(value2 == -1) {
143                     value2 = currentHash;
144                     hashmap[op2] = currentHash++;
145                 }
```

d. The expression is made using the operands' names and the opcodeString. For example, "%add1 = add nsw i32 %2, %3" will have "(%2's value) + (%3's value)". This expression (in the example, would be "(%2's value) + (%3's value)") will be checked if it is in the hash table already.
    i. If the expression is in the hash table, it is considered redundant and a message is printed out. In addition, the value for this expression is set to %add1 (in the example) or whichever operand the binary computation will be stored (variable instName below).
    ii. If the expression is not in the hash table, the expression is added as a key with currentHash as the value to the hash table. Then, currentHash is incremented.

```
147                 opString = to_string(value1) + opcodeString + to_string(value2);
148                 int opStrValue = getHashValue(hashmap, opString);
149                 errs() << "OpString: " << opString << "\n";
150
151                 //not already in hash map, add to it
152                 if(opStrValue == -1) {
153                     hashmap[opString] = currentHash;
154                     hashmap[instName] = currentHash++;
155                 }
156                 else {
157                     hashmap[instName] = opStrValue;
158                     errs() << "\n\tRedundancy found: " << inst << "\n\n";
159                 }
```

Test Cases

1. test.c Redundancies
    a. %add2 = add nsw i32 %tmp4, %tmp5
        i. The loads are as follows:
        %tmp4 = load i32, i32* %a.addr, align 4
        %tmp5 = load i32, i32* %b.addr, align 4
        ii. Thus, this add corresponds to a + b.

Lisa Chen
862186585
Project 2: Local Value Numbering(LLVM)

iii. You can see later that this is stored to %e:
store i32 %add2, i32* %e, align 4
iv. Thus, this instruction corresponds to “int e = a + b;”, which should be redundant because of “int c = a + b;” and no reassignments of a or b prior to use.

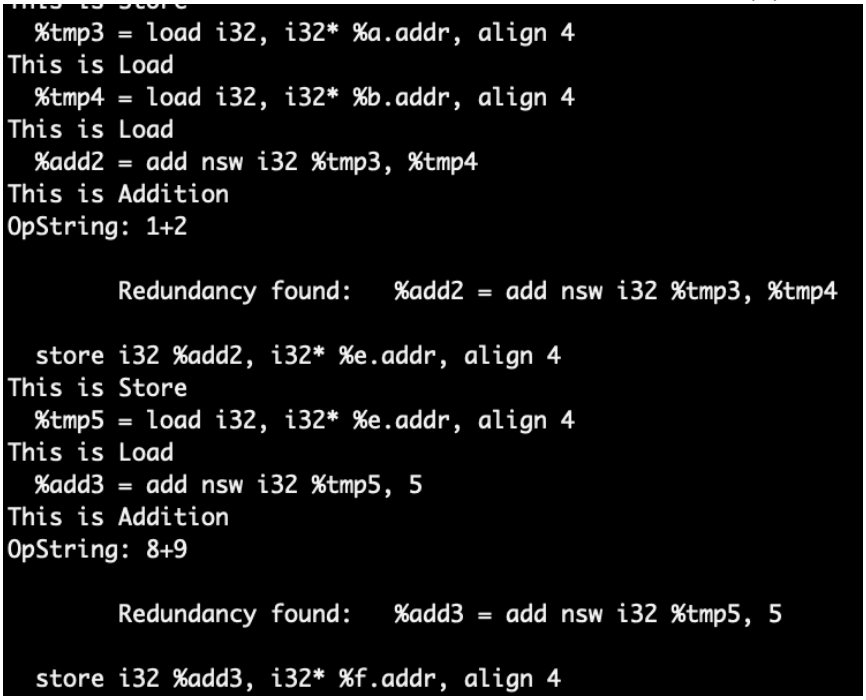
b. %add3 = add nsw i32 %tmp6, %tmp7
i. The loads are as follows:
%tmp6 = load i32, i32* %b.addr, align 4
%tmp7 = load i32, i32* %e, align 4
ii. Thus, this add corresponds to b + e. There’s only one “b+e” in test.c, which is “int f = b + e;”. When going through the code, this also means “f = b + a + b” where “b + a + b” is also the same as “b + c”; thus, this is redundant since we have “int d = b + c;”. Thus, this redundancy is correct.

```
  %tmp4 = load i32, i32* %a.addr, align 4
This is Load
  %tmp5 = load i32, i32* %b.addr, align 4
This is Load
  %add2 = add nsw i32 %tmp4, %tmp5
This is Addition
OpString: 1+2

        Redundancy found:   %add2 = add nsw i32 %tmp4, %tmp5

  store i32 %add2, i32* %e, align 4
This is Store
  %tmp6 = load i32, i32* %b.addr, align 4
This is Load
  %tmp7 = load i32, i32* %e, align 4
This is Load
  %add3 = add nsw i32 %tmp6, %tmp7
This is Addition
OpString: 2+3

        Redundancy found:   %add3 = add nsw i32 %tmp6, %tmp7

  store i32 %add3, i32* %f, align 4
```

2. test1.c Redundancies
a. %add2 = add nsw i32 %tmp3, %tmp4
i. The loads are as follows:
%tmp3 = load i32, i32* %a.addr, align 4
%tmp4 = load i32, i32* %b.addr, align 4
ii. Thus, this add corresponds to a+b.
iii. You can see this later stored as e:
store i32 %add2, i32* %e.addr, align 4
iv. Thus, this instruction corresponds to “e = a + b;”, which should be redundant since we have this earlier operation: “c = a + b;”.
b. %add3 = add nsw i32 %tmp5, 5

Lisa Chen
862186585
Project 2: Local Value Numbering(LLVM)

i. The load is as follows:
%tmp5 = load i32, i32* %e.addr, align 4
ii. Thus, this corresponds to “e+5”. In test1.c, there is only one “e+5”, which is “f = e + 5;”. “e” was assigned to “a+b”, which was redundant with “c=a+b” (as seen in redundancy in part 2.a above). Thus, we have “f=c+5” as well. Since we have calculated “d = c + 5;”, this is a redundancy.



3. test2.c Redundancies
   a. %add1 = add nsw i32 %tmp2, %tmp3
      i. The loads are as follows:
      %tmp2 = load i32, i32* %x.addr, align 4
      %tmp3 = load i32, i32* %y.addr, align 4
      ii. Thus, this add corresponds to x+y.
      iii. You can see this is later stored to a:
      store i32 %add1, i32* %a.addr, align 4
      iv. Thus, this instruction corresponds to “a = x + y;” which should be redundant since we have this earlier operation: “b = x + y;”.

   b. %add2 = add nsw i32 %tmp4, %tmp5
      i. The loads are as follows:
      %tmp4 = load i32, i32* %x.addr, align 4
      %tmp5 = load i32, i32* %y.addr, align 4
      ii. Thus, this add corresponds to x+y.
      iii. You can see this is later stored to c:
      store i32 %add2, i32* %c.addr, align 4
      iv. Thus, this instruction corresponds to “c= x + y;” which should be redundant since we have this earlier operation: “b = x + y;”.

Lisa Chen
862186585
Project 2: Local Value Numbering(LLVM)

```
  %tmp2 = load i32, i32* %x.addr, align 4
This is Load
  %tmp3 = load i32, i32* %y.addr, align 4
This is Load
  %add1 = add nsw i32 %tmp2, %tmp3
This is Addition
OpString: 4+5

        Redundancy found:   %add1 = add nsw i32 %tmp2, %tmp3

  store i32 %add1, i32* %a.addr, align 4
This is Store
  store i32 17, i32* %a.addr, align 4
This is Store
  %tmp4 = load i32, i32* %x.addr, align 4
This is Load
  %tmp5 = load i32, i32* %y.addr, align 4
This is Load
  %add2 = add nsw i32 %tmp4, %tmp5
This is Addition
OpString: 4+5

        Redundancy found:   %add2 = add nsw i32 %tmp4, %tmp5

  store i32 %add2, i32* %c.addr, align 4
```

4. test3.c Redundancies
    a. %add1 = add nsw i32 %tmp3, %tmp4
        i. The loads are as follows:
        %tmp3 = load i32, i32* %x.addr, align 4
        %tmp4 = load i32, i32* %z.addr, align 4
        ii. Thus, this corresponds to “x+z”. In test3.c, there is only one “x+z” used, which is “c = x + z;”. Earlier in test3.c, z is assigned y, so “c=x+y” is also true. Since this operation “a = x + y;” occurs prior to “c = x + z;” with no reassignment to x, y, or z, this is redundant as expected.

```
  %tmp3 = load i32, i32* %x.addr, align 4
This is Load
  %tmp4 = load i32, i32* %z.addr, align 4
This is Load
  %add1 = add nsw i32 %tmp3, %tmp4
This is Addition
OpString: 4+5

        Redundancy found:   %add1 = add nsw i32 %tmp3, %tmp4

  store i32 %add1, i32* %c.addr, align 4
```

5. test4.c Redundancies
    a. %sub2 = sub nsw i32 %tmp6, %tmp7
        i. The loads are as follows:
        %tmp6 = load i32, i32* %a.addr, align 4
        %tmp7 = load i32, i32* %d.addr, align 4
        ii. Thus, this corresponds to a – d.

iii. You can see that this is later stored as d:
store i32 %sub2, i32* %d.addr, align 4
iv. Thus, this instruction corresponds to “d = a - d;”. This is verified redundant as “b = a - d;” was called prior to this “d = a - d;” operation with no reassignments of a or d.

b. There are no other redundancies as the expression “b+c” is not redundant given b is reassigned prior to “b+c” second use.

```
  %tmp6 = load i32, i32* %a.addr, align 4
This is Load
  %tmp7 = load i32, i32* %d.addr, align 4
This is Load
  %sub2 = sub nsw i32 %tmp6, %tmp7
This is Subtraction
OpString: 5-4

        Redundancy found:   %sub2 = sub nsw i32 %tmp6, %tmp7

  store i32 %sub2, i32* %d.addr, align 4
```

6. test5.c Redundancies
   a. There are no redundancies in this code.
   b. Although there was a trick at “e = d * c;” in hopes of finding a coding bug matching this to “a = b * c;” because “d = b;” was called between these two operations, these two are NOT necessarily equal and therefore NOT redundant. This is because c is reassigned using “c = a + b;” prior to “e = d * c;”.
7. test6.c Redundancies
   a. %mul2 = mul nsw i32 %tmp5, %tmp6
      i. The loads are as follows:
      %tmp5 = load i32, i32* %i.addr, align 4
      %tmp6 = load i32, i32* %j.addr, align 4
      ii. Thus, this corresponds to i * j.
      iii. You can see that this is later stored as t3:
      store i32 %mul2, i32* %t3, align 4
      iv. Thus, this instruction corresponds to “int t3 = i * j;”, which is redundant as “int t1 = i * j;” is called prior to this instruction with no reassignment of i or j.

```
This is Store
  %tmp5 = load i32, i32* %i.addr, align 4
This is Load
  %tmp6 = load i32, i32* %j.addr, align 4
This is Load
  %mul2 = mul nsw i32 %tmp5, %tmp6
This is Multiplication
OpString: 4*5

        Redundancy found:   %mul2 = mul nsw i32 %tmp5, %tmp6

  store i32 %mul2, i32* %t3, align 4
```