HelloWorld.cpp is in llvm-tutor-main\HelloWorld
The test files and the shell script files are in llvm-tutor-main\build

Date: 1/30/2024
Objective: Install llvm-tutor-main, install test files, and run demo from TA's slides
Tasks Completed: Install llvm-tutor-main and test files
Challenges Faced:

1) ./test.sh didn't work in WSL.

Solutions/Workarounds:

1) dos2unix test.sh (command for wsl, shell file to work)

Code Snippets:

```
export LLVM_DIR=/lib/llvm-17/
cd build
cmake -DLT_LLVM_INSTALL_DIR=$LLVM_DIR ..
make
clang -c -emit-llvm -fno-discard-value-names -O0 demo.cpp -o demo.bc
opt -load-pass-plugin ./lib/libHelloWorld.so -passes=hello-world -disable-output demo.bc
./demo.sh
```

Testing: Ran demo file to see function names and its arguments
Learnings: Introduced to HelloWorld's visitor function.
Next Steps: Finish project

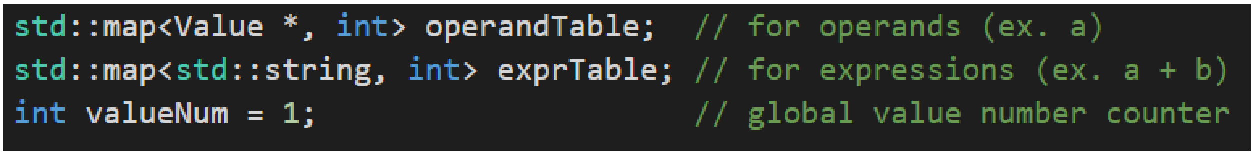
Date: 2/2/2024
Objective: Finish project
Tasks Completed: Finished project
Challenges Faced:

1) Issue with hashTable. Originally I had only one hashtable for both operands and expressions. However, there was conflict between the Value* addresses for operands and the string expressions.
2) There was confusion on memory address occurrences for load and store in relation to what value number they should have.
3) Given how I implemented updateOperandTable, there were multiple instances of the same key with different value numbers, when I really wanted to update the previous value stored there.
4) I had a condition to check if the expression already existed in the table in the visitor function. It was strange extraneous code.

Solutions/Workarounds:

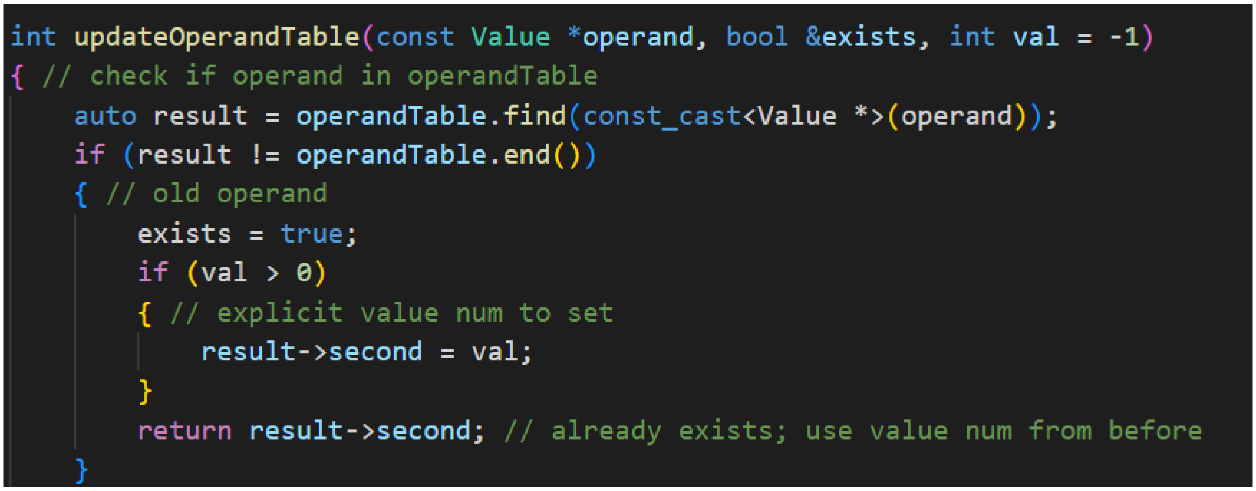
1) Instead of just one hashTable, I made an operandTable<Value*,int> and exprTable<string,int>. Both tables share the same global value number.
2) I printed out addresses for load &inst and inst.getOperand(0) and store inst.getOperand(0) and inst.getOperand(1). Then I was able to see that placing inst.getOperand(0) in the operandTable for both load and store before placing their respective destination addresses in the operandTable.
3) I added a parameter "val" in updateOperandTable to allow specific assignment of a value number to the operand passed in. This removed the uncertainty of multiple value numbering mapped to a single operand.
4) In the end, I simplified everything by adding a parameter "exists" to update and determine if the key exists already in exprTable.

Code Snippets:
Definition of hash tables and global value number

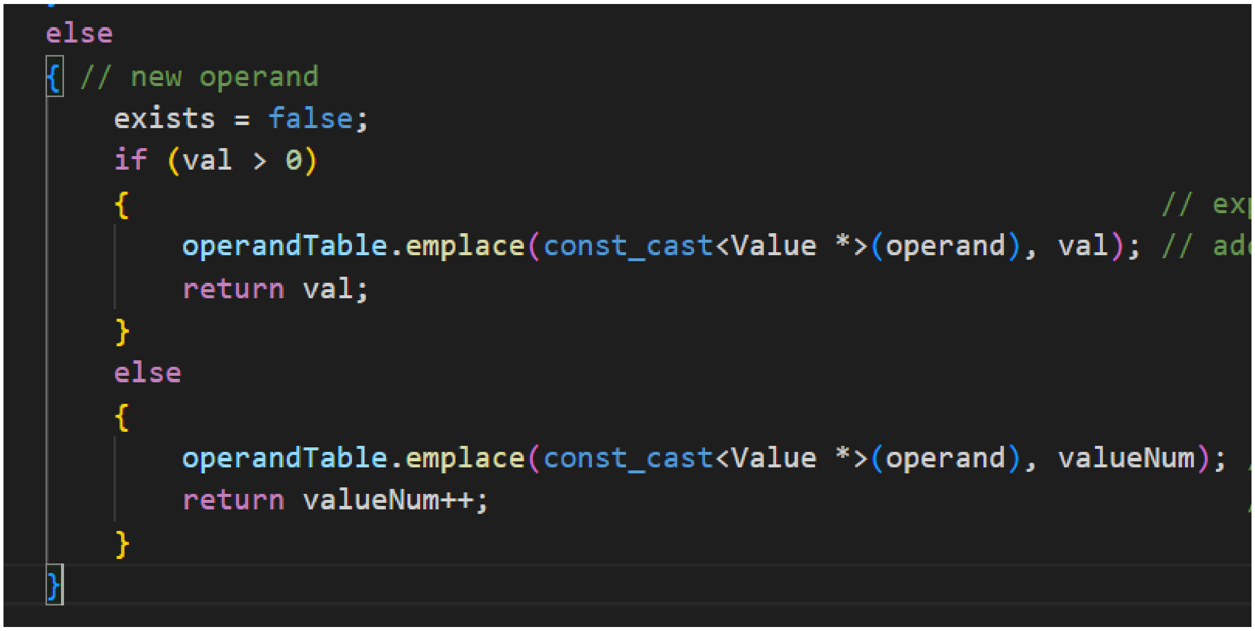


```
std::map<Value *, int> operandTable;  // for operands (ex. a)
std::map<std::string, int> exprTable; // for expressions (ex. a + b)
int valueNum = 1;                     // global value number counter
```

Function to update operandTable with optional option to specify value number with val

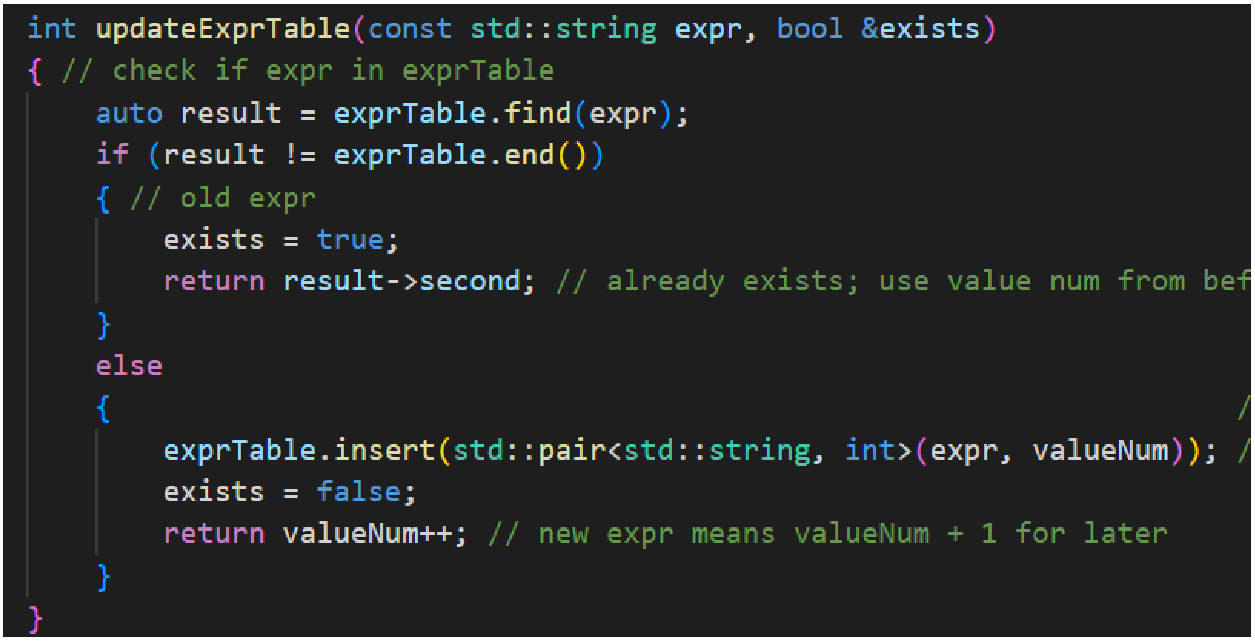


```
int updateOperandTable(const Value *operand, bool &exists, int val = -1)
{ // check if operand in operandTable
    auto result = operandTable.find(const_cast<Value *>(operand));
    if (result != operandTable.end())
    { // old operand
        exists = true;
        if (val > 0)
        { // explicit value num to set
            result->second = val;
        }
        return result->second; // already exists; use value num from before
    }
```



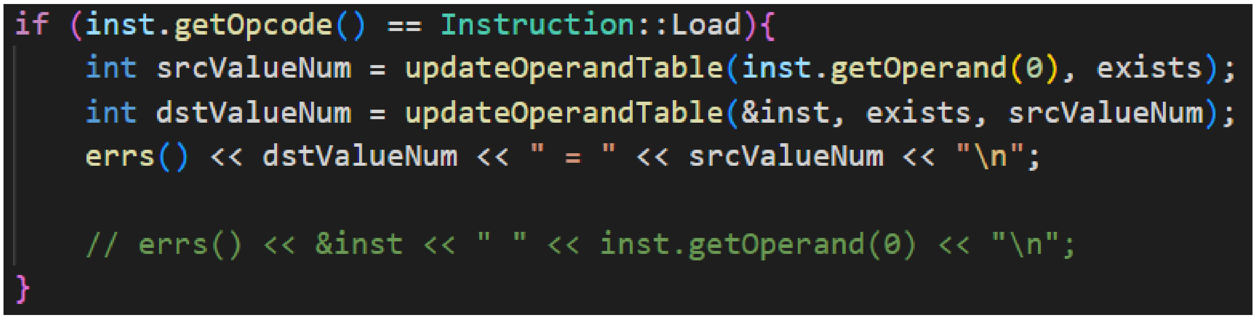
```
    else
    { // new operand
        exists = false;
        if (val > 0)
        {                                                                  // ex
            operandTable.emplace(const_cast<Value *>(operand), val); // ad
            return val;
        }
        else
        {
            operandTable.emplace(const_cast<Value *>(operand), valueNum); /
            return valueNum++;                                            /
        }
    }
```

Function to update exprTable with the same global value number as operandTable.



```
int updateExprTable(const std::string expr, bool &exists)
{ // check if expr in exprTable
    auto result = exprTable.find(expr);
    if (result != exprTable.end())
    { // old expr
        exists = true;
        return result->second; // already exists; use value num from bef
    }
    else
    {                                                                     /
        exprTable.insert(std::pair<std::string, int>(expr, valueNum)); /
        exists = false;
        return valueNum++; // new expr means valueNum + 1 for later
    }
}
```

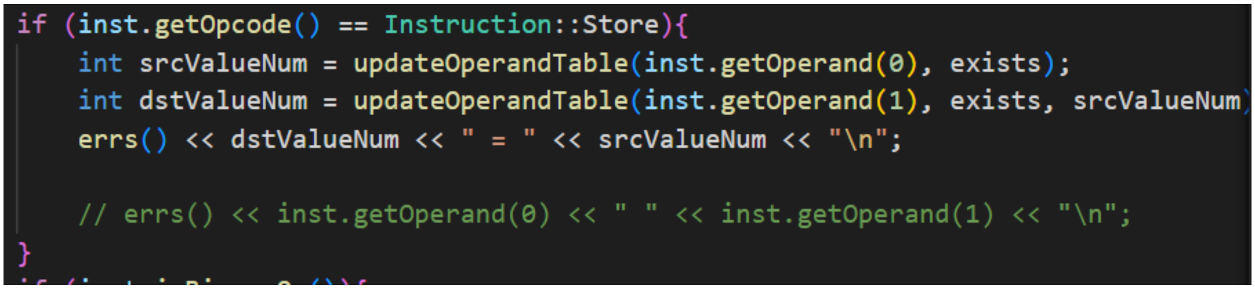
visitor() Load handler, place source operand into operandTable then destination with same value number



```
if (inst.getOpcode() == Instruction::Load){
    int srcValueNum = updateOperandTable(inst.getOperand(0), exists);
    int dstValueNum = updateOperandTable(&inst, exists, srcValueNum);
    errs() << dstValueNum << " = " << srcValueNum << "\n";

    // errs() << &inst << " " << inst.getOperand(0) << "\n";
}
```

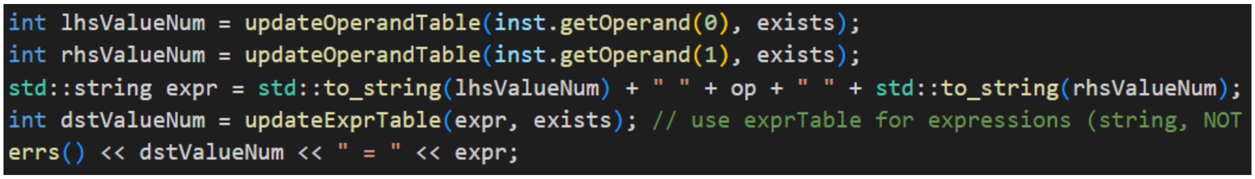
visitor() Store handler, place source operand into operandTable then destination with same value number



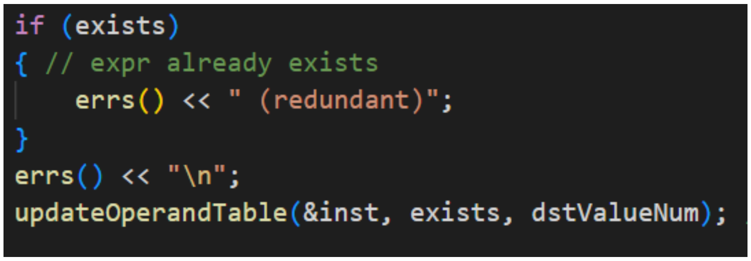
```
if (inst.getOpcode() == Instruction::Store){
    int srcValueNum = updateOperandTable(inst.getOperand(0), exists);
    int dstValueNum = updateOperandTable(inst.getOperand(1), exists, srcValueNum)
    errs() << dstValueNum << " = " << srcValueNum << "\n";

    // errs() << inst.getOperand(0) << " " << inst.getOperand(1) << "\n";
}
```

visitor() Expression handling, place expr into exprTable. If it's already there, state that the current expression is redundant.



```
int lhsValueNum = updateOperandTable(inst.getOperand(0), exists);
int rhsValueNum = updateOperandTable(inst.getOperand(1), exists);
std::string expr = std::to_string(lhsValueNum) + " " + op + " " + std::to_string(rhsValueNum);
int dstValueNum = updateExprTable(expr, exists); // use exprTable for expressions (string, NOT
errs() << dstValueNum << " = " << expr;
```



```
if (exists)
{ // expr already exists
    errs() << " (redundant)";
}
errs() << "\n";
updateOperandTable(&inst, exists, dstValueNum);
```

Testing: Ran ./test.sh and the output matches exactly as given in the project description. Also created testDiv.c using the "/" operator.

Learnings: Learned about llvm passes and how to handle instructions in basic blocks in a function. I saw how particular instructions are structured and how to access their operands, and particularly how these address values relate to other instruction types (load and store). I learned about the process of value numbering and how the addresses of type Value* should be stored and what value numbers particular addresses should point to.

Next Steps: Finish project