Ben Smith
Email: bxs566@case.edu
Course: CSDS 337 - Compiler Design
Instructor: Dr. Vipin Chaudhary

Homework - PG3
ID: 3559750
Term: Spring 2024
Due Date: 20^{th} March, 2024

Number of hours delay for this Problem Set: 14
Cumulative number of hours delay so far: 40

I discussed this homework with: Jackson Schuetzle

More information is available in the README.md file.

An Intro To LLVM

LLVM is a powerful framework that allows you to generate and even execute code. In this assignment, we will look at the basics of programming with LLVM. This is by no means a complete guide, but our hope is that you get an understanding on how LLVM works and can use this as a base for future LLVM-related projects.

Assignment

Your assignment is to modify main.cpp to generate code for 6 optimized functions. Each stage of the assignment will get progressively harder (though sometimes getting started can be the hardest). Please see the guide section of the README first for a quick guide about LLVM and for common FAQs. WINDOWS_SETUP instructions are provided, though the VM will have all the packages needed by default. A runDocker.sh script is provided for building with docker if that is preferred. You may need to 'chmod +x run.sh' and other scripts first before running them with './run.sh'.

Problem 1

Before we can do anything, an LLVM context and module needs to be establed. This module should be called 'sampleMod'. Have this module be printed (printing will print LLVM assembly) to 'sampleMod.ll' and have its bitcode be written to 'sampleMod.bc'. Note that the printed '.ll' file is text, and will show you the exact code you are generating.

Problem 2

Create a function named 'simple'. It takes no parameters, and returns a 32-bit integer. Have it return 0. Use 'llvm::verifyFunction(*FUNC_HERE);' after you are done building a function to check for errors.

Problem 3

Create a function named 'add' that takes two 32-bit integers and returns the sum of them (a 32-bit integer).

Problem 4

Create a function named 'addIntFloat' that has the first parameter be a 32-bit integer and the second parameter be a float. The function should return a float. Note that there are different types of casts to float, you can assume the input integer is signed.

Problem 5
Adding things together in a linear fashion is fun, but what about temporary variables and control flow? Create a function called ‘conditional‘, it will take a boolean input (1-bit integer) and output a 32-bit integer. Allocate a mutable variable stored on the stack in the entry block. If the input parameter is true, store a ‘3‘ to the variable, else store a ‘5‘. Using only one add instruction in the entire function, return the value of the stored variable added with ‘11‘. The function should return ‘14‘ if the parameter is true or ‘16‘ if it is false. **DO NOT OPTIMIZE THIS FUNCTION YOURSELF.** The point of this part is to make sure you understand control flow and mutable stack variables.

Problem 6
Do the last part again, except name the function ‘oneTwoPhi‘ and do it with phi nodes instead.

Problem 7
Do the last part again, except name the function ‘selection‘ and do it with the ‘select‘ instruction instead.

Problem 8
Add an LLVM legacy function pass manager ‘llvm::legacy::FunctionPassManager‘. Add the ‘llvm::createPromoteMemoryToRegisterPass()‘, ‘llvm::createReassociatePass()‘, ‘llvm::createGVNPass()‘, and ‘llvm::createCFGSimplificationPass()‘ passes and run it on the 6 functions you created earlier. What was the effect of these passes on each of the 6 functions?

Solution: The passes did not change the functions without any control flow. In the functions with control flow, the passes optimized the functions by replacing any conditional statements, conditional breaks, and their corresponding blocks with a single select statement. This greatly reduced the complexity of the functions.

Problem 9
By now, you should have a decent base knowledge of generating code with LLVM. Note that the below items are completely optional, but they can give you ideas on what you can do:
* Make hello world in LLVM.
* Make a function’s arguments mutable local to the function by storing them into stack variables.
* Declare the ‘malloc‘ and ‘free‘ functions and call them in different functions to play with heap memory.
* Make a while or for loop with LLVM.
* Play around with struct types.
* Create a basic calculator interpreter program.