CS 201 - Project 1 Report

Celvin Lizama Pena

January 19, 2022

1 Introduction

The purpose of this project is to get the development environment set up, to get familiar with Clang and LLVM, and to experience how to create CFGs. This report will give details about the experiments conducted as well as show the inputs, outputs, and commands used to run the experiments. The list of experiments conducted are as follows:

- Source (.c) to binary (executable
- Source (.c) to objective (.o)
- Source (.c) to machine assembly(.s)
- Source(.c) to LLVM bitcode (.bc)
- Source(.c) to LLVM IR(.ll)
- LLVM bitcode (.bc) to LLVM IR (.ll)
- LLVM IR (.ll) to LLVM bitcode (.bc)
- LLVM IR to machine assembly (.s)
- Interpret the LLVM IR
- Generate the CFG of test.c

Note that the aforementioned experiments are run on the program **test.c**, which is a simple program written in C that writes "Hello World!" to the console.

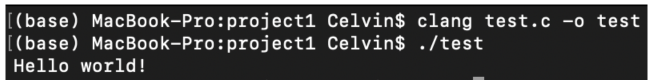
2 Environment Setup

The following screenshot is supplied as evidence that both Clang and LLVM have been successfully installed on my machine.

```
[(base) MacBook-Pro:project1 Celvin$ clang --version
Homebrew clang version 13.0.0
Target: x86_64-apple-darwin21.1.0
Thread model: posix
InstalledDir: /usr/local/opt/llvm/bin
[(base) MacBook-Pro:project1 Celvin$ opt -version
Homebrew LLVM version 13.0.0
  Optimized build.
  Default target: x86_64-apple-darwin21.1.0
  Host CPU: westmere
```

3 Source to binary (executable)

Compiling the program from source to binary is a simple process. This can be done using the command `clang test.c -o test` where `test.c` is the name of source file, and the flag `-o` names the executable file `test`. Then using the command `./test` will run the executable file `test`. Upon running the executable, `Hello world!` will be printed to the console.

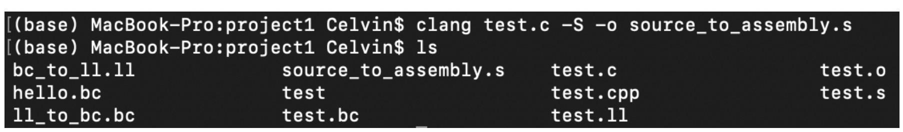


4 Source (.c) to objective (.o)

The command to achieve this is `clang test.c -c`. This will generate the desired `objective` file. The `-c` flag will run the preprocessor, parser, and type checking stages as well as the LLVM generation and optimization stages and the target-specific code generation, which produces an `assembly` file. Finally, the assembler is run, which generates the targeted `.o` object file.

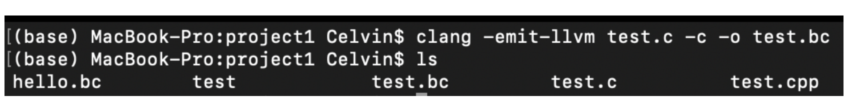
5 Source (.c) to machine assembly(.s)

The command is the same as above but uses the `-S` flag, which will run the preprocessor, parser, and type checking stages as well as the LLVM generation and optimization stages and the target-specific code generation, which produces an `assembly` file. The complete command used in this experiment is `clang test.c -S -o source_to_a`



6 Source(.c) to LLVM bitcode (.bc)

Translating the source file into LLVM bitcode allows us to use the LLVM tools on the bitcode file. To do the translation, the `-emit-llvm` option can be used along with `-c` flag to emit an LLVM `.bc` file. The complete command used in this project is `clang -emit-llvm test.c -c -o test.bc`, which name the bitcode file `test.bc`. The following screenshot confirms that the command creates the `test.bc` file as expected.



7 Source(.c) to LLVM IR(.ll)

The command is similar to the command to convert from a `source` file to `bitcode` file. Instead of using the `-c` flag, you use the `-S` flag. The full command used is `clang -emit-llvm test.c -S -o test.ll`. The following screenshot confirms that the command creates the `test.ll` file as expected.

```
[(base) MacBook-Pro:project1 Celvin$ clang -emit-llvm test.c -S -o test.ll
[(base) MacBook-Pro:project1 Celvin$ ls
hello.bc        test            test.bc         test.c          test.cpp        test.ll
```

8 LLVM bitcode (.bc) to LLVM IR (.ll)

The same command can also be used to translate from bitcode to IR. The command works by translating the first file to either bitcode or IR, depending on the flag used (`-c` for bitcode, `-S` for IR). For this experiment, since the translation if from bitcode to IR, the following command was issued: `clang -emit-llvm test.bc -S bc_to_ll.ll`. The following screenshot verifies that the IR file was indeed created.

```
[(base) MacBook-Pro:project1 Celvin$ clang -emit-llvm test.bc -S -o bc_to_ll.ll
[(base) MacBook-Pro:project1 Celvin$ ls
bc_to_ll.ll     test            test.c          test.ll
hello.bc        test.bc         test.cpp        test.s
```

9 LLVM IR (.ll) to LLVM bitcode (.bc)

Same as above, except the first file is `test.ll` and the flag is `-c`, as shown in below.

```
[(base) MacBook-Pro:project1 Celvin$ clang -emit-llvm test.ll -c -o ll_to_bc.bc
[(base) MacBook-Pro:project1 Celvin$ ls
bc_to_ll.ll     ll_to_bc.bc     test.bc         test.cpp        test.o
hello.bc        test            test.c          test.ll         test.s
```

10 LLVM IR to machine assembly (.s)

In order to convert the LLVM IR created in Section 5, `llc` command, which compiles LLVM source inputs into assembly language, which can then be passed through an assembler and linker to generate an executable. The command used in this experiment was `llc test.ll -o test.s`. The following screenshot verifies that the command generate the desired `test.s` file, which contains the assembly language code.

```
[(base) MacBook-Pro:project1 Celvin$ llc test.ll -o test.s
[(base) MacBook-Pro:project1 Celvin$ ls
hello.bc        test.bc         test.cpp        test.s
test            test.c          test.ll
```

11 Interpret the LLVM IR

Now that we have the LLVM IR file, we can use `lli` to directly execute programs in LLVC IR or bitcode formats. To interpret the LLVC IR file, we use the command `lli test.ll`, where `test.ll` is the LLVC IR file. This will print `Hello world!` to console without needing to run an executable file. Notice that the same can be done with the `bitcode` file created in Section 4. The following screenshot confirms this:

```
[(base) MacBook-Pro:project1 Celvin$ lli test.ll
Hello world!
[(base) MacBook-Pro:project1 Celvin$ lli test.bc
Hello world!
```

12 Generate the CFG of test.c

Now that we know how to generate the LLVM bitcode, we can we `opt` in order to generate a `dot` file containing the CFG of `test.c`. The command used is `opt -dot-cfg -enable-new-pm=0 test.bc`, which takes the LLVM bitcode file `test.bc` and writes the CFG to `.main.dot`. Finally, we use `graphviz` to convert the CFG into a PDF. The command for this is `dot -Tpdf .main.dot -o cfg.pdf`.

```
[(base) MacBook-Pro:project1 Celvin$ opt -dot-cfg -enable-new-pm=0 test.bc
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.

Writing '.main.dot'...
[(base) MacBook-Pro:project1 Celvin$ dot -Tpdf .main.dot -o cfg.pdf
[(base) MacBook-Pro:project1 Celvin$ ls
a.out                ll_to_bc.bc            test.c       test.s
bc_to_ll.ll          source_to_assembly.s   test.cpp
cfg.pdf              test                   test.ll
hello.bc             test.bc                test.o
```

The created CFG is simple, as the `test.c` program simply outputs "Hello world!".