Compilers 2 Mini-Assignment 4: Report

Taha Adeel Mohammed: CS20BTECH110xx
Vikhyath: CS20BTEC110xx
Pranav K Nayak: ES20BTECH11035

Q2.

The choice of five programs is used to illustrate concepts like looping, object-orientedness, recursion, control flow with conditionals, and basic arithmetic computations. The five programs are:

i. bubble_sort.cpp, which implements a bubble sort,
ii. class.cpp, which implements basic class and object declaration,
iii. fibonacci.cpp, which calculates the fibonacci sequence and is used to demonstrate recursion.
iv. get_area.cpp, which demonstrates basic control flow through conditionals, and
v. sum.cpp, which demonstrates basic arithmetic.

The IRs for all five of these follow the format defined in the LLVM Language Reference. They consist of ‘Modules’, and since each one is a standalone program on its own, none of the Modules are linked.

```
; ModuleID = 'bubble_sort.cpp'
```

```
; ModuleID = 'class.cpp'
```

```
; ModuleID = 'fibonacci.cpp'
```

```
; ModuleID = 'get_area.cpp'
```

```
; ModuleID = 'sum.cpp'
```

Further perusal of the IR shows us examples of how LLVM defines the scope of its variables, from local variables being prefixed by a ‘%’ character:

```
%"class.std::ios_base::Init" = type { i8 }
%"class.std::basic_ostream" = type { i32 (...)**, %"class.std::basic_ios" }
%"class.std::basic_ios" = type { %"class.std::ios_base", %"class.std::basic_ostream"*, i8, i8, %"class.std::basic_st
%"class.std::ios_base" = type { i32 (...)**, i64, i64, i32, i32, i32, %"struct.std::ios_base::_Callback_list"*, %"st
%"struct.std::ios_base::_Callback_list" = type { %"struct.std::ios_base::_Callback_list"*, void (i32, %"class.std::i
%"struct.std::ios_base::_Words" = type { i8*, i64 }
```

To global ones being prefixed by an ‘@’ character:

```
@_ZStL8__ioinit = internal global %"class.std::ios_base::Init" zeroinitializer, align 1
@__dso_handle = external hidden global i8
@__const.main.arr = private unnamed_addr constant [10 x i32] [i32 10, i32 7, i32 5, i32 9, i32 2, i32 1, i32 8, i32 6, i32 4, i32 3], align 16
@_ZSt4cout = external dso_local global %"class.std::basic_ostream", align 8
@.str = private unnamed_addr constant [2 x i8] c"\00", align 1
@llvm.global_ctors = appending global [1 x { i32, void ()*, i8* }] [{ i32, void ()*, i8* } { i32 65535, void ()* @_GLOBAL__sub_I_bubble_sort.cpp, i8* null }]
```

We also see examples of comments, which are delimited by a ‘;’ and proceed to the end of the line:

```
; Function Attrs: nofree nounwind
```

```
; preds = %1, %3
```

```
; preds = %2, %14, %10, %6, %4
```

We see the use of unnamed temporaries, which are named sequentially, as follows:

```
%1 = tail call nonnull align 8 dereferenceable(8)
%2 = tail call nonnull align 8 dereferenceable(8)
%3 = tail call nonnull align 8 dereferenceable(8)
%4 = tail call nonnull align 8 dereferenceable(8)
%5 = tail call nonnull align 8 dereferenceable(8)
```

How LLVM represents user-defined functions can be seen below:

```
define dso_local float @_Z8get_areafi(float %0, i32 %1) local_unnamed_addr #3 {
```

With the ‘define’ keyword being used along with return type as well as the function name, which is mangled, and parameter types, all of them being a part of the function signature.

We see how LLVM represents control flow in the form of a switch statement as:

```
switch i32 %1, label %14 [
  i32 1, label %4
  i32 2, label %4
  i32 3, label %6
  i32 4, label %15
  i32 5, label %10
]
```

These and more features can be observed by going through the .ll files in the Q2 directory.

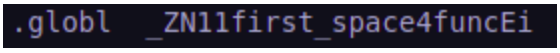
Q3.

For this question, five different programs were written to test how the C++ compiler mangles names. C++ mangles names according to the IA64 C++ ABI Specification, which states that all mangled names have a general structure as follows:

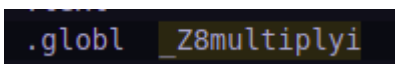
1. All names are mangled by prefixing “_Z” to the encoding of its name.
2. Encodings must be of the form:
    a. <function_name> <bare_function_type>
    b. <data_name>
    c. <special_name>
3. More details can be found here: https://itanium-cxx-abi.github.io/cxx-abi/abi.html#mangling

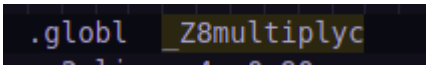
Some examples of mangled names are as follows:

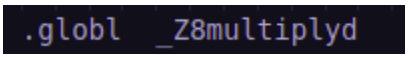
1. Name in source code: “first_space”



2. An example of how overloaded function names are mangled:



,




In the source code, the function multiply was overloaded by changing the type of its input parameter, with one having an int, another having a double, and the last having a char. We can see this play out in the last character of the mangled function names, each corresponding to the type of its input.

More examples of name mangling can be seen by going through the .s files in the Q3 directory.

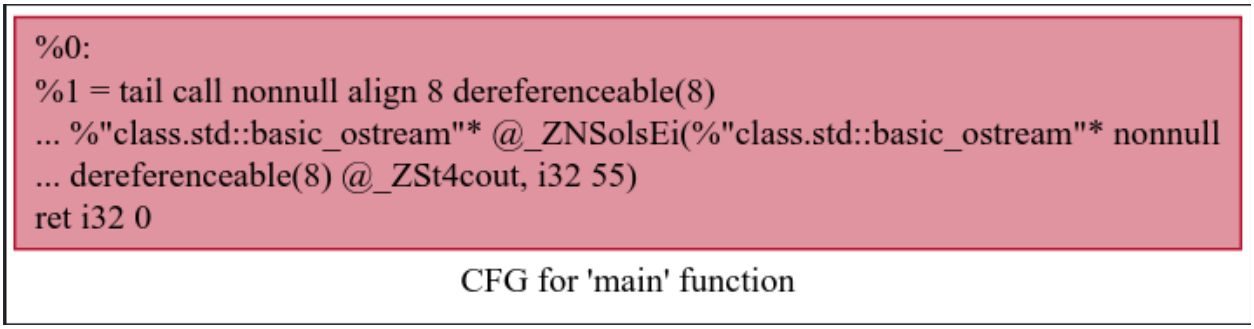
Q4.

a) Running the bash script with “Q4 a” as its arguments, performs the compilation stage-wise for the source codes in the Q4/a directory.
   - First, we run the preprocessing phase using the -E flag with clang to generate the .i file with the preprocessed code. In the examples, we can see that the preprocessor expands upon the macros, includes the source code from the header files and substitutes macros appropriately apart from removing comments.
   -
   -
   - Next, we generate the LLVM IR from the preprocessed code. The output IR can be seen in their corresponding .bc (bitcode IR) and .ll (human readable IR) files.
   - Then, using llc, we generate the native assembly code(.s file) from the bitcode LLVM IR file (.bc file).
   - Finally, we compile the native assembly code into the native binary, which we can execute.

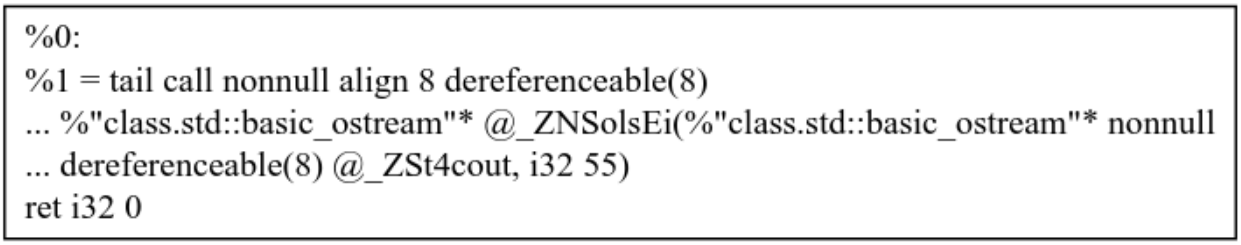
   We run the above stages again with clang optimizations enabled. We can see that the generated LLVM IR and assembly code are significantly smaller with the optimizations. The optimization unrolls the loops, performs dead code elimination, reorders the assembly instructions for optimization, inlines functions for speed, etc to generate optimized code. For example, to add the first 1 million numbers, the unoptimized code takes 3 msecs, while the optimized code takes less than 1msec.

b) for optimized code with constant propagation: The main functions are almost empty because only few variables are being used in the code using print statements.

So, the state of the rest of variables is being totally ignored as they are not utilized anywhere.

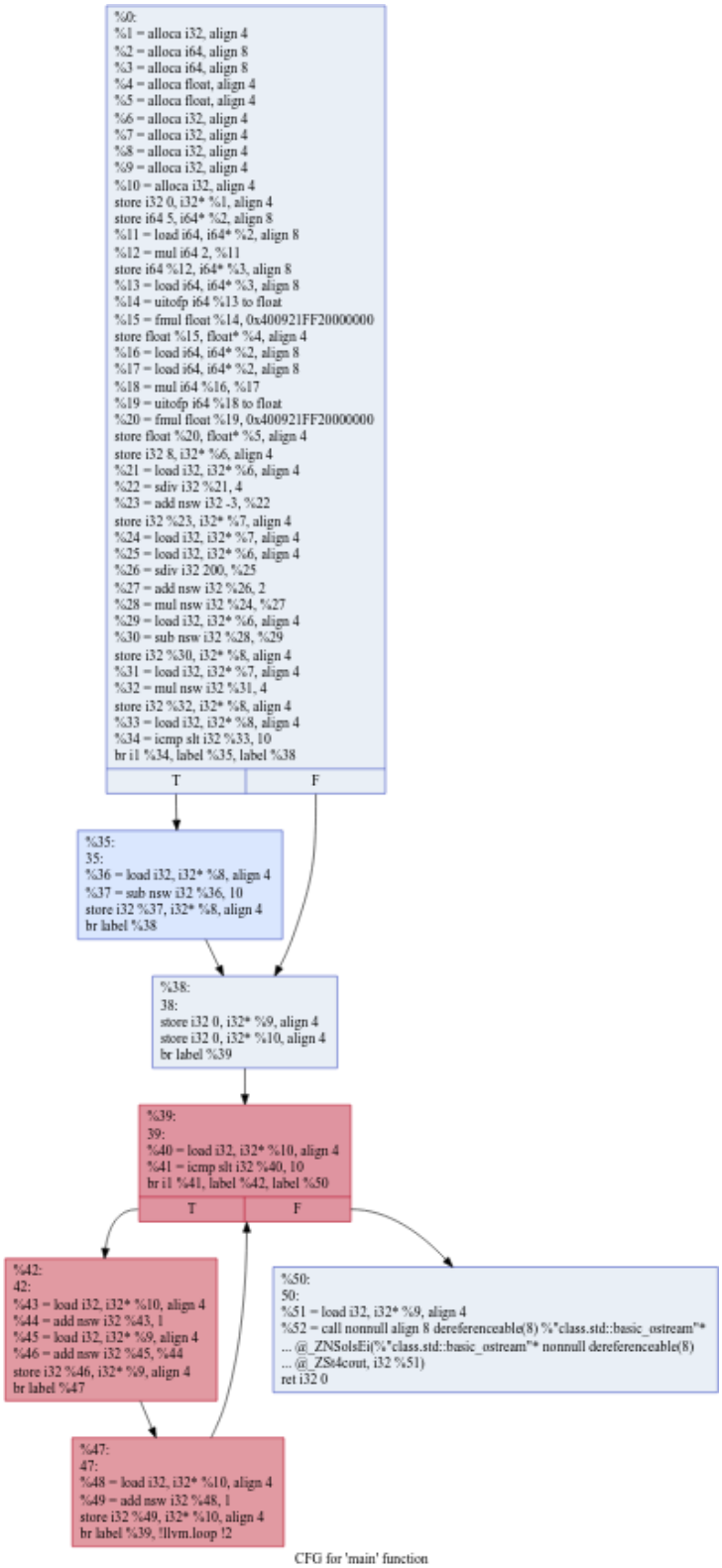


CFG for 'main' function

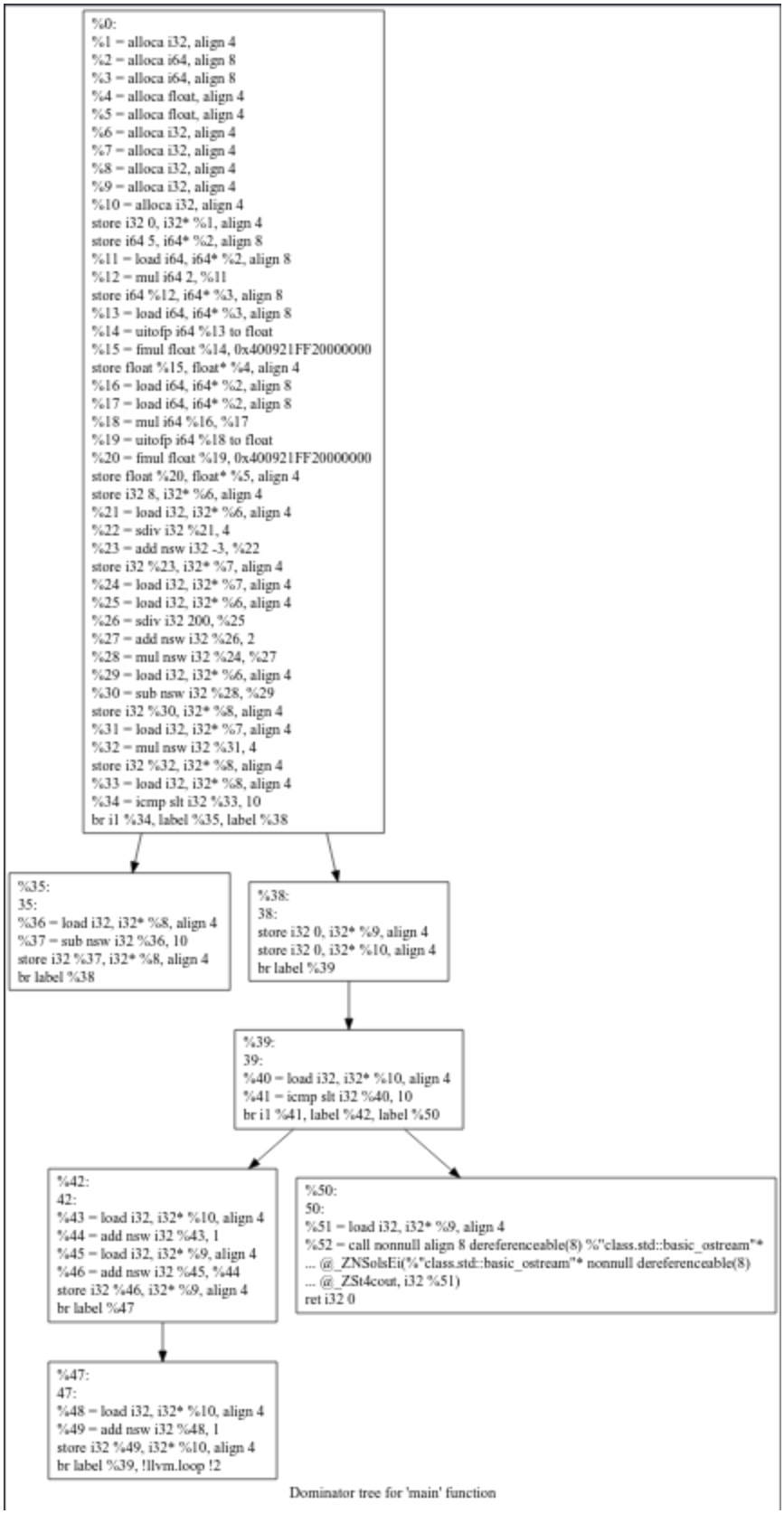


Dominator tree for 'main' function

for unoptimized code with constant propagation.

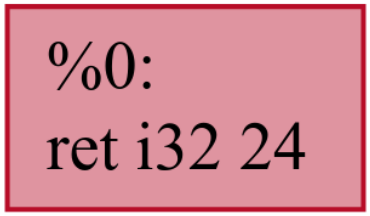


CFG for 'main' function

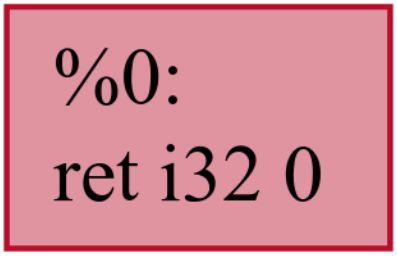


Dominator tree for 'main' function

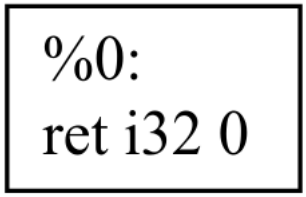
for optimized code with dead code elimination: Here we can see the compiler completely ignores code that is unreachable (code after a valid return statement) and unused variables (variables initialized but not utilized anywhere in the code). And as the rest of the values are constant and can be computed, the compiler also employs constant propagation which is evident from the return values across CFG's and DOM.



CFG for '_Z3foov' function



CFG for 'main' function

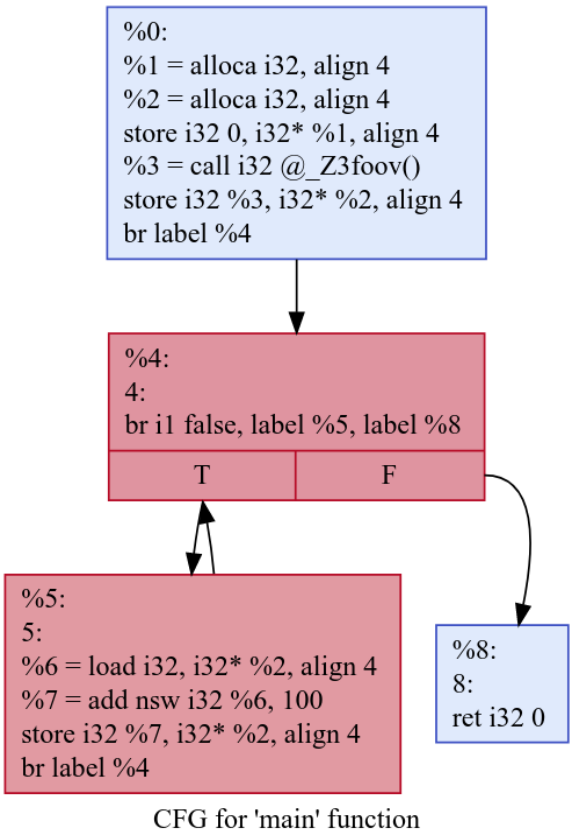


Dominator tree for 'main' function

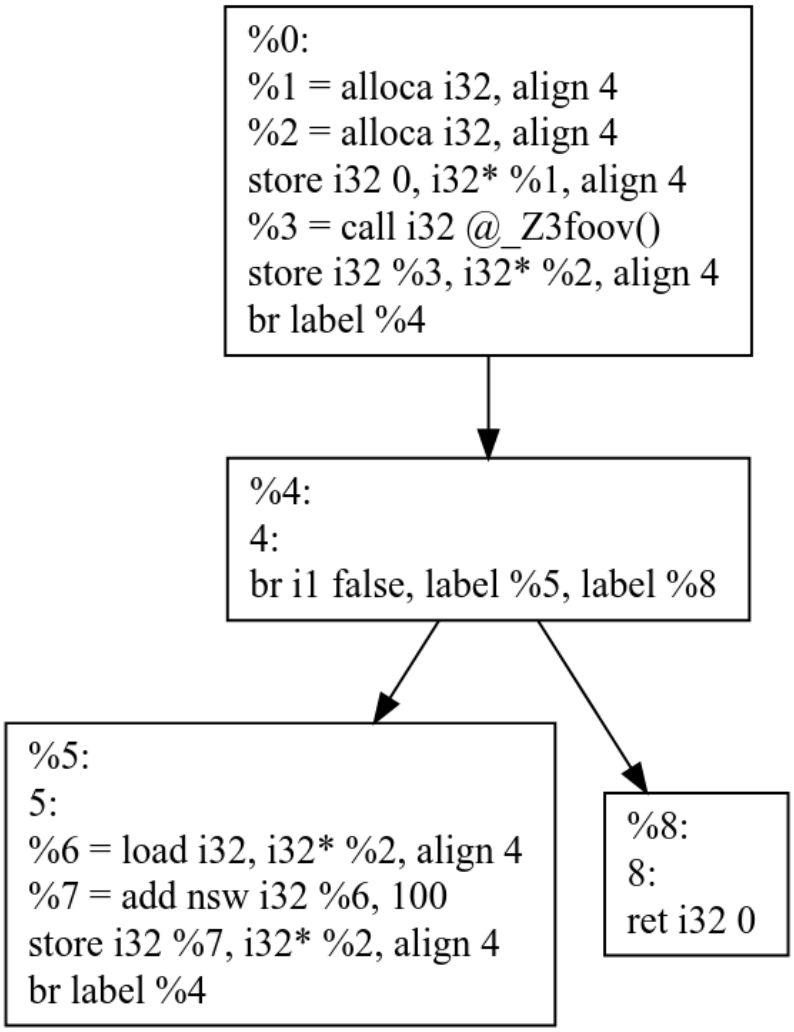
for unoptimized code with dead code elimination:

```
%0:
%1 = alloca i32, align 4
%2 = alloca i8, align 1
%3 = alloca i32, align 4
store i32 24, i32* %1, align 4
store i8 0, i8* %2, align 1
%4 = load i32, i32* %1, align 4
%5 = srem i32 %4, 4
store i32 %5, i32* %3, align 4
%6 = load i32, i32* %1, align 4
%7 = load i32, i32* %3, align 4
%8 = add nsw i32 %6, %7
ret i32 %8
```

CFG for '_Z3foov' function



CFG for 'main' function



Dominator tree for 'main' function

(c) Explore the relevant tools in LLVM - llvm-as, llvm-dis, llc, lli, etc. with different examples.

We use the different llvm tools on the test source files in Q4/c.

- **Llvm-as :** llvm-as is the assembler provided by LLVM. We can generate the bitcode file from LLVM IR using this tool.
- **Llvm-dis :** llvm-dis is the disassembler provided by LLVM. This tool can be used to generate the human readable LLVM IR from the bitcode file.
- **Llc :** llc is the static compiler provided by LLVM. It helps us generate the assembly files from the c source files directly
- **Lli :** lli allows us to execute the bitwise LLVM IR code using Just in Time Compilation (JIT).