# CS6383 - Introduction to Compiler Engineering

## Mini-Assignment #1

**Submitted By :**

1. **Girish Thatte – CS22MTECH11005**
2. **Raghvendra Gupta – CS22MTECH11009**

---

## 1. LLVM Directory Structure

a. **llvm/cmake** – This contains the cmake utility files which in turn generates system build files.

b. **llvm/include** – It contains all the header files which are exported from the LLVM library. It includes three subdirectories, one is 'llvm' which contains the LLVM-specific header files, and subdirectories for LLVM, second is 'Support' which contains generic support libraries provided with LLVM, and the other is 'Config' that contains header files configured by cmake.

c. **llvm/lib** – This folder contains LLVM source files. This folder has some more subfolders like '/IR' which contains core LLVM source files which have implementations for BasicBlock & Instruction classes used in Passes, '/Analysis' having files for Call Graphs, '/Bitcode' having code for reading & writing bitcode, '/Transforms' which contains source code files used in transforming one IR to another IR after applying the 'opt' tool

d. **llvm/tools** – It contains executables built out of the libraries present in llvm/lib. Some important tools built are : opt, llvm-as, llvm-dis, llvm-link, lli, llc, etc.

   1. **llvm-as** : The LLVM assembler tool transforms human readable LLVM assembly to LLVM bitcode. (.ll → .bc)
   2. **llvm-dis** : The LLVM disassembler tool transforms LLVM bitcode to human readable LLVM assembly. (.bc → .ll)
   3. **llvm-link** : Links multiple LLVM modules into a single program.
   4. **llc** : It is basically an LLVM backend compiler, which translates LLVM bitcode(.bc) to a native code assembly file(.s).
   5. **opt** – It reads LLVM bitcode, applies a series of LLVM to LLVM transformations (which are specified on the command line), and outputs the resultant bitcode. 'opt -help' is a good way to get a list of the program transformations available in LLVM.

---

## 2. Study on LLVM–IR

Following programs are used for study of LLVM-IR code generated using clang.

1. Fibonacci Series
2. Palindrome Checker
3. Ternary Search
4. Merge Sort
5. Matrix Multiplication

To generate the LLVM-IR for the input program, we use below command:

**./clang++ -emit-llvm <path/to/source/code> -S -o <path/to/store/generated/IR>**

e.g. for the Fibonacci Series program (fib.cpp), following is the example command:

*./clang++ -emit-llvm "/home/cs22mtech11005/Assignment1/fib.cpp" -S -o "/home/cs22mtech11005/Assignment1/fib.ll"*

- **Fibonacci Series:**

```
; ModuleID = '/home/cs22mtech11005/Assignment1/fib.cpp'
source_filename = "/home/cs22mtech11005/Assignment1/fib.cpp"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
```

The 1st line in the above snippet is a comment which represents the ModuleID which defines an LLVM module for 'fib.cpp' file. Every LLVM program (which is technically called a module). This LLVM module is a high–level data structure that consists of functions, global variables, external function prototypes, and symbol table entries. The 2nd line represents the entire path to high-level source code file name.
The 3rd line represents how the data is stored on the underlying architecture in the memory. 'e' represents that the data is stored in Little Endian format. If 'E' was written instead of 'e' then it would have meant that data is stored in Big Endian format. n:8:16:32:64 represents the size of integers in bits that can be represented on the target CPU architecture. S128 tells us that each entry represented by a stack pointer is 128-bit. i64:64-f80 tells us that the integer type alignment is of 64-bit size & 64 bit aligned and f80 tells us that the 80-bit variant of long double is supported.
The 4th line containing "target triple" represents the information about the target host. The string is hyphen-separated(–) with 4 components, Architecture-Vendor-OS-Version x86_64 is the underlying architecture, OS is linux, and gnu is the version of OS.

```
define dso_local noundef i32 @main() #5 {
entry:
 %retval = alloca i32, align 4
 %n = alloca i32, align 4
 store i32 0, i32* %retval, align 4
 %call = call noundef nonnull align 8 dereferenceable(16) %"class.std::basic_istream"*
@_ZNSirsERi(%"class.std::basic_istream"* noundef nonnull align 8 dereferenceable(16) @_ZSt3cin,
i32* noundef nonnull align 4 dereferenceable(4) %n)
 %call1 = call noundef nonnull align 8 dereferenceable(8) %"class.std::basic_ostream"*
@_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc(%"class.std::basic_ostream"* noundef
nonnull align 8 dereferenceable(8) @_ZSt4cout, i8* noundef getelementptr inbounds ([10 x i8], [10
x i8]* @.str, i64 0, i64 0))
 %0 = load i32, i32* %n, align 4
 %call2 = call noundef i32 @_Z3fibi(i32 noundef %0)
 %call3 = call noundef nonnull align 8 dereferenceable(8) %"class.std::basic_ostream"*
@_ZNSolsEi(%"class.std::basic_ostream"* noundef nonnull align 8 dereferenceable(8) %call1, i32
noundef %call2)
 ret i32 0
}
```

define dso_local noundef i32 @main() #5 — The entity names starting with '@' signify that entity is either a global variable or function declaration/definition. In the above snippet, '@main' indicates the IR format name for our C++ main() function. The 'dso_local' is a 'Runtime Preemption Specifier' that tells us the scope of the function is local. The noundef attribute, tells that when a function call argument or the return value will not contain uninitialized bits. 'i32' is the return type of the main function which tells that main() function returns a 32-bit integer value. The '#5' means to use the attributes named #5 for the function. These attribute meanings are represented at the end of the LLVM IR file and they will be applied to our main() function. E.g. See below :

```
attributes #5 = { mustprogress noinline norecurse optnone uwtable
"frame-pointer"="all" "min-legal-vector-width"="0" "no-trapping-math"="true"
"stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
```

Now, we can see that there is only 1 Basic Block(BB) with the label 'entry'. The IR name for local variables starts with '%'. The local variables assigned memory with 'alloca' keyword, are allocated memory on the stack. So, here %retval & %n are two stack-allocated local variables.

%retval = alloca i32, align 4 – This line allocates a 32 bit integer on the stack. %retval is the name of a pointer to this location on the stack. The align 4 ensures that the address will be a multiple of 4.

%0 = load i32, i32* %n, align 4 – This loads the 32-bit integer into the %0 register allocated variable pointed to by %n which is 32-bit value. The align 4 ensures that the address will be a multiple of 4.
%call2 = call noundef i32 @_Z3fibi(i32 noundef %0) – This calls the 'fib' function called in our high-level source code, 'i32' tells that the return value is 32-bit integer value. The 'fib' function takes one 32-bit integer as a parameter.

ret i32 0 – This instruction indicates the termination of the BB. Some termination instructions for BB are ret, br, callbr, switch, etc. It tells us that we return from the function with a 32-bit integer return value of 0.

```
define dso_local noundef i32 @_Z3fibi(i32 noundef %n) #4 {
entry:
 %retval = alloca i32, align 4
 %n.addr = alloca i32, align 4
 store i32 %n, i32* %n.addr, align 4
 %0 = load i32, i32* %n.addr, align 4
 %cmp = icmp sle i32 %0, 1
 br i1 %cmp, label %if.then, label %if.end

if.then:                                          ; preds = %entry
 %1 = load i32, i32* %n.addr, align 4
 store i32 %1, i32* %retval, align 4
 br label %return
```

```
if.end:                                           ; preds = %entry
  %2 = load i32, i32* %n.addr, align 4
  %sub = sub nsw i32 %2, 1
  %call = call noundef i32 @_Z3fibi(i32 noundef %sub)
  %3 = load i32, i32* %n.addr, align 4
  %sub1 = sub nsw i32 %3, 2
  %call2 = call noundef i32 @_Z3fibi(i32 noundef %sub1)
  %add = add nsw i32 %call, %call2
  store i32 %add, i32* %retval, align 4
  br label %return

return:                                           ; preds = %if.end, %if.then
  %4 = load i32, i32* %retval, align 4
  ret i32 %4
}
```

define dso_local noundef i32 @_Z3fibi(i32 noundef %n) #4 — The entity names starting with '@' signify that entity is either a global variable or function declaration/definition. In the above snippet, '@_Z3fibi' indicates the IR format name for the fib() function in C++ program. The 'dso_local' is a 'Runtime Preemption Specifier' that tells us the scope of the function is local. The noundef attribute, tells that when a function call argument or the return value will not contain uninitialized bits. 'i32' is the return type of the fib() function which tells that fib() function returns a 32-bit integer value. The '#4' means to use the attributes named #4 for the function. These attribute meanings are represented at the end of the LLVM IR file and they will be applied to our fib() function. This function takes one 32-bit integer as a parameter '%n' which is a name to the stack location.

There are 4 basic blocks(BB) in the "fib" function represented in LLVM IR format. These BB are ones with labels : *%entry, %if.then, %if.end, %return*. The ;preds = %entry indicates that the predecessor BB for the BB with labels %if.then, %if.end. We can reach the BB '%return' from 2 predecessor BB with labels '%if.then' and '%if.end'.

%cmp = icmp sle i32 %0, 1 – This represents the compare instruction, it compares two values one represented by local variable %0 and other one is constant value of 1.

br i1 %cmp, label %if.then, label %if.end – This is a conditional branch instruction which if evaluates to true, then goes to BB with label '%if.then', else goes to BB with label '%if.end'.

%1 = load i32, i32* %n.addr, align 4 – This loads the 32-bit integer value pointed by stack pointer '%n.addr' into register-allocated variable %1 and does 4 byte alignment.

store i32 %1, i32* %retval, align 4 – This sets the 32 bit integer pointed to by %retval to the 32 bit value stored in local variable %1 and does 4 byte alignment.

br label %return – This is break instruction which breaks out to the next BB labeled '%return'. This marks the end of BB labeled '%if.end'.

%sub = sub nsw i32 %2, 1 – This is an arithmetic instruction which does subtraction of a 32-bit integer in %2 local variable value by 1. 'nsw' stands for 'no signed wrap' which tells that the resultant value of the sub is undefined if unsigned and/or signed overflow if it occurs.

%call = call noundef i32 @_Z3fibi(i32 noundef %sub) – This is the recursive call to fib() function with the updated parameter value %sub. This tells that the return value is a 32-bit integer value and passes the value represented by %sub as the parameter.

%3 = load i32, i32* %n.addr, align 4 – This loads the 32-bit integer value pointed by stack pointer '%n.addr' into register-allocated variable %3 and does 4 byte alignment.

%sub1 = sub nsw i32 %3, 2 – This is an arithmetic instruction which does subtraction of a 32-bit integer in %3 local variable value by 2. 'nsw' stands for 'no signed wrap' which tells that the resultant value of the sub is undefined if unsigned and/or signed overflow if it occurs.

%call2 = call noundef i32 @_Z3fibi(i32 noundef %sub1) – This is the recursive call to fib() function with the updated parameter value %sub1. This tells that the return value is a 32-bit integer value and passes the value represented by %sub1 as the parameter.

%add = add nsw i32 %call, %call2 – This is an arithmetic add instruction which adds two 32-bit integers whose values are obtained from the recursive function calls and stored in %call and %call2 local variables. 'nsw' stands for 'no signed wrap' which tells that the resultant value of the sub is undefined if unsigned and/or signed overflow if it occurs.

store i32 %add, i32* %retval, align 4 – This sets the 32 bit integer pointed to by %retval to the 32 bit value stored in local variable %add and does 4 byte alignment.

br label %return – This is break instruction which breaks out to the next BB labeled '%return'. This marks the end of BB labeled '%if.end'.

%4 = load i32, i32* %retval, align 4 – This loads the 32-bit integer value pointed by '%retval' into register-allocated variable %4 and does 4 byte alignment.

ret i32 %4 – This instruction indicates the termination of the BB. It tells us that we return from the function with a 32-bit integer return value of 0.

❖ **Palindrome Check**:

```
; ModuleID = '/home/cs22mtech11005/Assignment1/palindrome.cpp'
source_filename = "/home/cs22mtech11005/Assignment1/palindrome.cpp"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
```

The 1st line in the above snippet is a comment which represents the ModuleID which defines an LLVM module for 'palindrome.cpp' file. Every LLVM program (which is technically called a module). This LLVM module is a high–level data structure that consists of functions, global variables, external function prototypes, and symbol table entries. The 2nd line represents entire path to high-level source code file name. The 3rd line represents how the data is stored on the underlying architecture in the memory. 'e' represents that the data is stored in Little Endian format. If 'E' was written instead of 'e' then it would have meant that data is stored in Big Endian format. n:8:16:32:64 represents the size of integers in bits that can be represented on the target CPU architecture. S128 tells us that each entry represented by a stack pointer is 128-bit. i64:64-f80 tells us that the integer type alignment is of 64-bit size & 64 bit aligned and f80 tells us that the 80-bit variant of long double is supported.
The 4th line containing "target triple" represents the information about the target host. The string is hyphen-separated(–) with 4 components, Architecture-Vendor-OS-Version x86_64 is the underlying architecture, OS is linux, and gnu is the version of OS.

```
define dso_local noundef zeroext i1 @_Z12isPalindromeiPc(i32 noundef %n, i8* noundef %str) #4 {
entry:
 %retval = alloca i1, align 1
 %n.addr = alloca i32, align 4
 %str.addr = alloca i8*, align 8
 %i = alloca i32, align 4
 %j = alloca i32, align 4
 store i32 %n, i32* %n.addr, align 4
 store i8* %str, i8** %str.addr, align 8
 %0 = load i32, i32* %n.addr, align 4
 %1 = zext i32 %0 to i64
 store i32 0, i32* %i, align 4
 %2 = load i32, i32* %n.addr, align 4
 %sub = sub nsw i32 %2, 1
 store i32 %sub, i32* %j, align 4
 br label %while.cond

while.cond:                                       ; preds = %if.end, %entry
 %3 = load i32, i32* %i, align 4
 %4 = load i32, i32* %j, align 4
 %cmp = icmp slt i32 %3, %4
 br i1 %cmp, label %while.body, label %while.end
```

```
while.body:                                       ; preds = %while.cond
  %5 = load i8*, i8** %str.addr, align 8
  %6 = load i32, i32* %i, align 4
  %idxprom = sext i32 %6 to i64
  %arrayidx = getelementptr inbounds i8, i8* %5, i64 %idxprom
  %7 = load i8, i8* %arrayidx, align 1
  %conv = sext i8 %7 to i32
  %8 = load i8*, i8** %str.addr, align 8
  %9 = load i32, i32* %j, align 4
  %idxprom1 = sext i32 %9 to i64
  %arrayidx2 = getelementptr inbounds i8, i8* %8, i64 %idxprom1
  %10 = load i8, i8* %arrayidx2, align 1
  %conv3 = sext i8 %10 to i32
  %cmp4 = icmp ne i32 %conv, %conv3
  br i1 %cmp4, label %if.then, label %if.end

if.then:                                          ; preds = %while.body
  store i1 false, i1* %retval, align 1
  br label %return

if.end:                                           ; preds = %while.body
  %11 = load i32, i32* %i, align 4
  %inc = add nsw i32 %11, 1
  store i32 %inc, i32* %i, align 4
  %12 = load i32, i32* %j, align 4
  %dec = add nsw i32 %12, -1
  store i32 %dec, i32* %j, align 4
  br label %while.cond, !llvm.loop !4

while.end:                                        ; preds = %while.cond
  store i1 true, i1* %retval, align 1
  br label %return

return:                                           ; preds = %while.end, %if.then
  %13 = load i1, i1* %retval, align 1
  ret i1 %13
}
```

define dso_local noundef zeroext i1 @_Z12isPalindromeiPc(i32 noundef %n, i8* noundef %str) #4 — The entity names starting with '@' signify that entity is either a global variable or function declaration/definition. In the above snippet, '@_Z12isPalindromeiPc' indicates the IR format name for the fib() function in C++ program. The 'dso_local' is a 'Runtime Preemption Specifier' that tells us the scope of the function is local. The noundef attribute, tells that when a function call argument or the return value will not contain uninitialized bits. 'zeroext' indicates to the code generator that the parameter or return value should be

zero-extended. ‘i32’ is the return type of the fib() function which tells that fib() function returns a 32-bit integer value. The ‘#4’ means to use the attributes named #4 for the function. These attribute meanings are represented at the end of the LLVM IR file and they will be applied to our fib() function. This function takes one 32-bit integer as a parameter ‘%n’ which is a name to the stack location.

There are 7 basic blocks(BB) in the “fib” function represented in LLVM IR format. These BB are ones with labels : *%entry, %while.cond, %while.body, %if.then, %if.end, %while.end %return*. The ;preds = %entry, %if.end are the predecessor BB for the BB with labels %while.cond. We can reach the BB ‘%return’ from 2 predecessor BB with labels ‘%if.then’ and ‘%while.end’.

%i = alloca i32, align 4 – This line allocates a 32 bit integer on the stack. %i is the name of a pointer to this location on the stack. The align 4 ensures that the address will be a multiple of 4.

%j = alloca i32, align 4 – This line allocates a 32 bit integer on the stack. %j is the name of a pointer to this location on the stack. The align 4 ensures that the address will be a multiple of 4.

%cmp = icmp slt i32 %3, %4 – This represents the compare instruction, it compares two values one represented by local variable %3 and other one is constant value of %4.

%arrayidx = getelementptr inbounds i8, i8* %5, i64 %idxprom – The ‘getelementptr’ instruction is used to get the address of a subelement some DS. It only does address calculation and does not access memory.

store i8* %str, i8** %str.addr, align 8 –This stores the address of the array ‘str’ which is obtained using dereferencing into the ‘%str’.

%conv3 = sext i8 %10 to i32 – sext keyword indicates that the local variable %10 should be sign-extended to 32-bit integer value.

%cmp4 = icmp ne i32 %conv, %conv3 – This represents the compare instruction, it compares two values one represented by local variable %conv and other one is constant value of %conv3.

br i1 %cmp4, label %if.then, label %if.end – This is break instruction which breaks out to the BB labeled ‘%if.then’ if condition evaluates to true otherwise to BB labeled ‘%if.end‘ if condition evaluates to false. This marks the end of BB labeled ‘%while.body’.

%dec = add nsw i32 %12, -1 – This is an arithmetic add instruction which adds two 32-bit integers whose value is stored in local variable %12 and constant value -1. ‘nsw’ stands for ‘no signed wrap’ which tells that the resultant value of the sub is undefined if unsigned and/or signed overflow if it occurs. This instruction is in Static Single Assignment(SSA) form.

---

# 3. Assembly language

## a. Fibonacci Series :

```
# %bb.0:                                # %entry
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    movabsq $_ZStL8__ioinit, %rdi
    callq   _ZNSt8ios_base4InitC1Ev
    movabsq $_ZNSt8ios_base4InitD1Ev, %rdi
    movabsq $_ZStL8__ioinit, %rsi
    movabsq $__dso_handle, %rdx
    callq   __cxa_atexit
    popq    %rbp
    .cfi_def_cfa %rsp, 8
    retq
```

movq (%rsp), %rsp – This moves a 64-bit value where (%rsp) is called source or src and %rsp is called the destination or the dst. It basically looks up in a register with name %rsp and gets its value from that register and goes to the memory of that value and then assigns it to %rsp.
pushq %rbp – The 'rbp' is the frame pointer on x86_64. This instruction saves the address of the previous stack frame. It subtracts 8 bytes from the stack pointer.
callq _ZNSt8ios_base4InitC1Ev – It pushes a 64-bit return address.
movabsq $_ZNSt8ios_base4InitD1Ev, %rdi – It basically means that the machine encoding will contain a 64-bit value which will either be an immediate constant, or an absolute memory address
popq %rbp – It retrieves the value at current stack top (%rsp) and increments stack pointer by 8 bytes.
retq – It returns (ret) by popping a 64-bit word address from the stack.

```
# %bb.1:                                # %if.then
    movl    -8(%rbp), %eax
    movl    %eax, -4(%rbp)
    jmp .LBB1_3
```

movl –8(%rbp), %eax – It adds the value in register %rbp as base to the offset represented by constant i.e.–8 to produce a memory reference. It then moves the contents of memory location into register %eax.
movl %eax, –4(%rbp) – It moves the contents of present in the register %eax and moves it to the memory reference pointed by the value in register %rbp as base which is added to the offset represented by constant i.e. –8.
jmp .LBB1_3 – It performs an unconditional jump to the basic block with label .LBB1_3.

---

# 4. a) Going all the way from preprocessing stage to binary

We have used clang as a complete tool chain to convert the high level language program all the way from preprocessing step to the binary stage. We tried with a program which calculates the Area of Circle where the value of 'PI' is defined as a macro using '#define' construct. The steps which we followed in order are as follows:

1. **Preprocessing :** This step performs the actions of C language preprocessor directive for e.g. expanding the macros and expanding the included library files.
   e.g. for the Area of Circle program (areacircle.cpp), following is the example command:

*./clang++ -E /home/cs22mtech11009/Assignment1/areacircle.cpp -o /home/cs22mtech11009/Assignment1/areacircle_pre.cpp*

2. **Parsing :** This parses the source code and generates the Abstract Syntax Tree (AST).

3. **Intermediate Representations :** This step converts the source level IR (AST) to optimiser specific IR. (LLVM in our case).
   e.g. for the Area of Circle program (areacircle.cpp), following is the example command:

*./clang++ -emit-llvm -S /home/cs22mtech11009/Assignment1/areacircle.cpp -o /home/cs22mtech11009/Assignment1/areacircle.ll*

4. **Compiler Backend :** This converts the optimized IR to architecture specific assembly code.
   e.g. for the Area of Circle program (areacircle.cpp), following is the example command:

*./clang++ -S /home/cs22mtech11009/Assignment1/areacircle.ll -o /home/cs22mtech11009/Assignment1/areacircle.s*

5. **Assembler :** This step finally converts the target-specific assembly code to target-specific machine code object files.
   e.g. for the Area of Circle program (areacircle.cpp), following is the example command:

*./clang++ -c /home/cs22mtech11009/Assignment1/areacircle.s -o /home/cs22mtech11009/Assignment1/areacircle.o*

The snippets below show the output we obtained after every step and how the code of 'areacircle.cpp' gets transformed.

## Preprocessing

```
namespace std __attribute__ ((__visibility__ ("default")))
{
# 60 "/usr/lib/gcc/x86_64-linux-gnu/11/../../../../include/c++/11/iostream" 3
 extern istream cin;
 extern ostream cout;
 extern ostream cerr;
 extern ostream clog;
```

```
 extern wistream wcin;
 extern wostream wcout;
 extern wostream wcerr;
 extern wostream wclog;

 static ios_base::Init __ioinit;
}
# 2 "/home/cs22mtech11009/Assignment1/areacircle.cpp" 2
using namespace std;
int main(){
   int radius, area;
   cin>>radius;
   area = 3.14*radius*radius;
   cout<<area;
   return 0;
}
```

## Intermediate Representations

```
; Function Attrs: mustprogress noinline norecurse optnone uwtable
define dso_local noundef i32 @main() #4 {
entry:
 %retval = alloca i32, align 4
 %radius = alloca i32, align 4
 %area = alloca i32, align 4
 store i32 0, i32* %retval, align 4
 %call = call noundef nonnull align 8 dereferenceable(16) %"class.std::basic_istream"*
@_ZNSirsERi(%"class.std::basic_istream"* noundef nonnull align 8 dereferenceable(16) @_ZSt3cin, i32*
noundef nonnull align 4 dereferenceable(4) %radius)
 %0 = load i32, i32* %radius, align 4
 %conv = sitofp i32 %0 to double
 %mul = fmul double 3.140000e+00, %conv
 %1 = load i32, i32* %radius, align 4
 %conv1 = sitofp i32 %1 to double
 %mul2 = fmul double %mul, %conv1
 %conv3 = fptosi double %mul2 to i32
 store i32 %conv3, i32* %area, align 4
 %2 = load i32, i32* %area, align 4
 %call4 = call noundef nonnull align 8 dereferenceable(8) %"class.std::basic_ostream"*
@_ZNSolsEi(%"class.std::basic_ostream"* noundef nonnull align 8 dereferenceable(8) @_ZSt4cout, i32
noundef %2)
 ret i32 0
}
```

## Compiler Backend

```
main:                                   # @main
   .cfi_startproc
# %bb.0:                                # %entry
   pushq   %rbp
   .cfi_def_cfa_offset 16
   .cfi_offset %rbp, -16
   movq    %rsp, %rbp
   .cfi_def_cfa_register %rbp
   subq    $16, %rsp
   movl    $0, -4(%rbp)
   movabsq $_ZSt3cin, %rdi
   leaq    -8(%rbp), %rsi
   callq   _ZNSirsERi
   cvtsi2sdl   -8(%rbp), %xmm1
   movsd   .LCPI1_0(%rip), %xmm0           # xmm0 = mem[0],zero
   mulsd   %xmm1, %xmm0
   cvtsi2sdl   -8(%rbp), %xmm1
   mulsd   %xmm1, %xmm0
   cvttsd2si   %xmm0, %eax
   movl    %eax, -12(%rbp)
   movl    -12(%rbp), %esi
   movabsq $_ZSt4cout, %rdi
   callq   _ZNSolsEi
   xorl    %eax, %eax
   addq    $16, %rsp
   popq    %rbp
   .cfi_def_cfa %rsp, 8
   retq
.Lfunc_end1:
   .size   main, .Lfunc_end1-main
   .cfi_endproc
                                        # -- End function
   .section    .text.startup,"ax",@progbits
   .p2align    4, 0x90                         # -- Begin function _GLOBAL__sub_I_areacircle.cpp
   .type   _GLOBAL__sub_I_areacircle.cpp,@function
_GLOBAL__sub_I_areacircle.cpp:          # @_GLOBAL__sub_I_areacircle.cpp
   .cfi_startproc
```

We repeated the same sequence of steps for the ‘Palindrome Check’ program (palindrome.cpp) as well. We are giving all the intermediate generated files in the submission ZIP file.

---

# 4. b) Applying trivial optimizations Dead Code Elimination and Constant Propagation

For applying trivial optimization like Dead Code Elimination and Constant Propagation, we have used two programs **deadcode.cpp** and **constprop.cpp** respectively. These optimizations can be realized by using llvm/opt tool and **-dce** flag is used for Dead Code Elimination and -**sccp** flag for Sparse Conditional Constant Propagation. The commands which we used are mentioned below:

### Dead Code Elimination commands for deadcode.cpp

./opt -mem2reg -dce /home/cs22mtech11009/Assignment1/deadcode.ll -S -o /home/cs22mtech11009/Assignment1/deadcode_opt.ll -enable-new-pm=0

### Constant Propagation commands for constprop.cpp

./opt -mem2reg -sccp /home/cs22mtech11009/Assignment1/constprop.ll -S -o /home/cs22mtech11009/Assignment1/constprop_opt.ll -enable-new-pm=0

The commands for generating the CFG and Dom Tree for both these programs are mentioned below:

### CFG Generation Commands

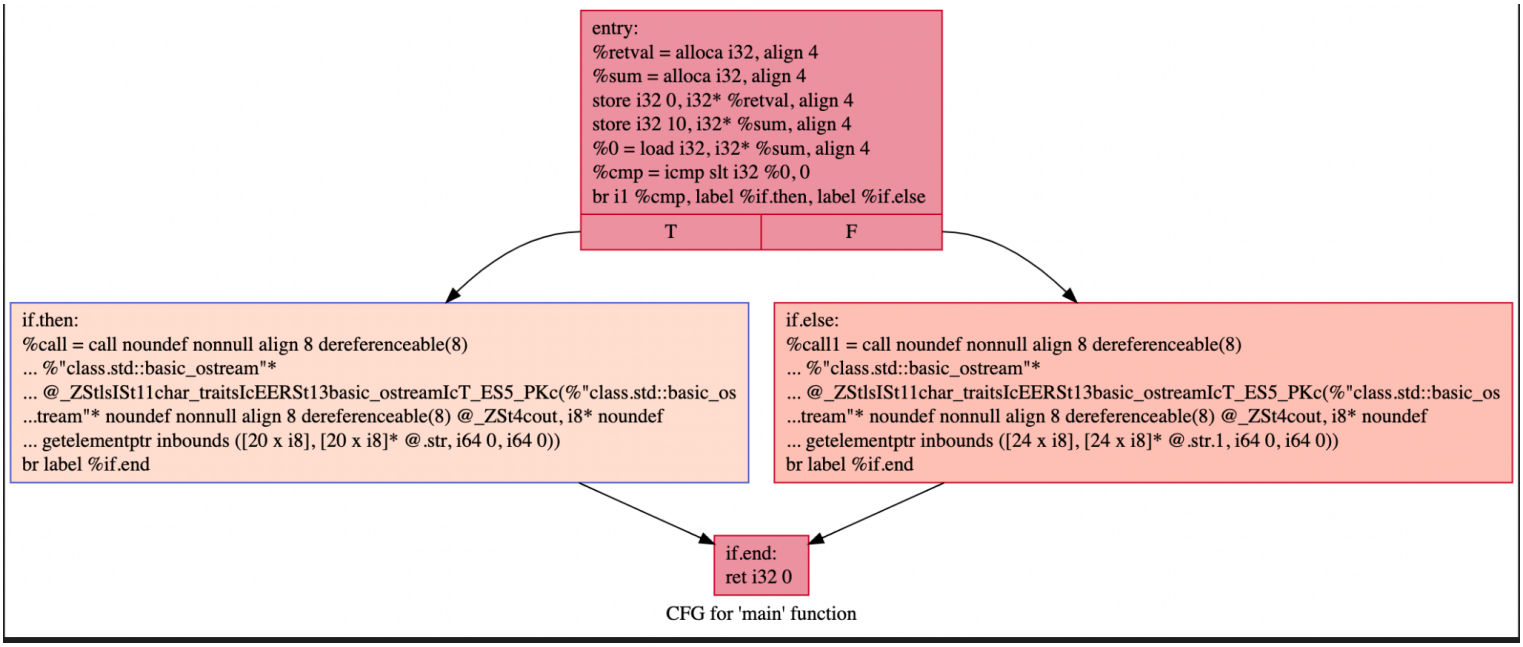
./opt -dot-cfg /home/cs22mtech11009/Assignment1/deadcode.ll -enable-new-pm=0
./opt -dot-cfg /home/cs22mtech11009/Assignment1/constprop.ll -enable-new-pm=0

### DOM Tree Generation Commands

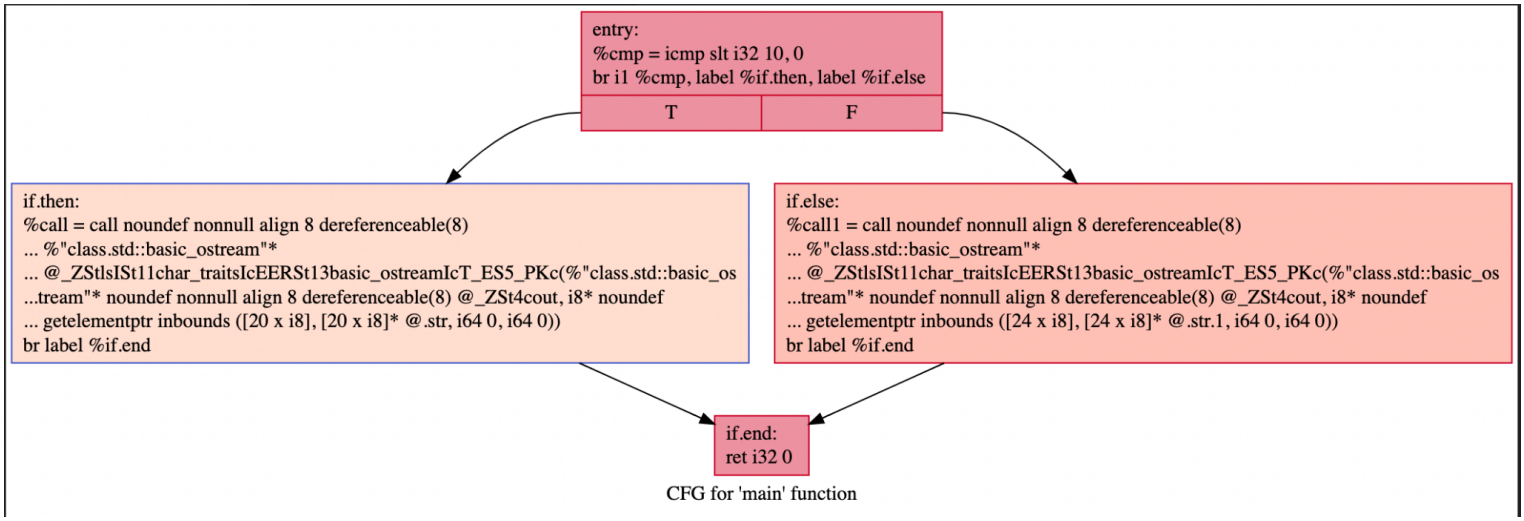
./opt -dot-dom /home/cs22mtech11009/Assignment1/deadcode.ll -enable-new-pm=0
./opt -dot-dom /home/cs22mtech11009/Assignment1/constprop.ll -enable-new-pm=0

For analyzing how the IR gets transformed after applying these optimization techniques, we need to observe the CFG and Dom Tree structure for both of these programs prior and after the optimization. We have hereby included the snippets of both aforementioned structures for analysis.

## CFG for Deadcode.cpp (Unoptimized)

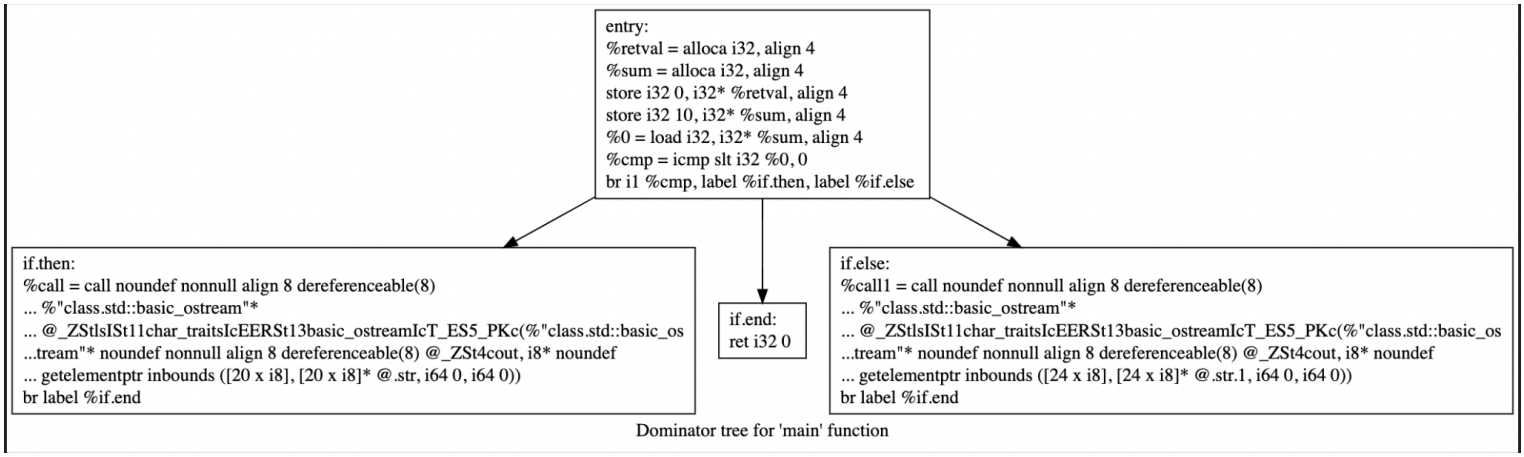


# CFG for Deadcode.cpp (Optimized)



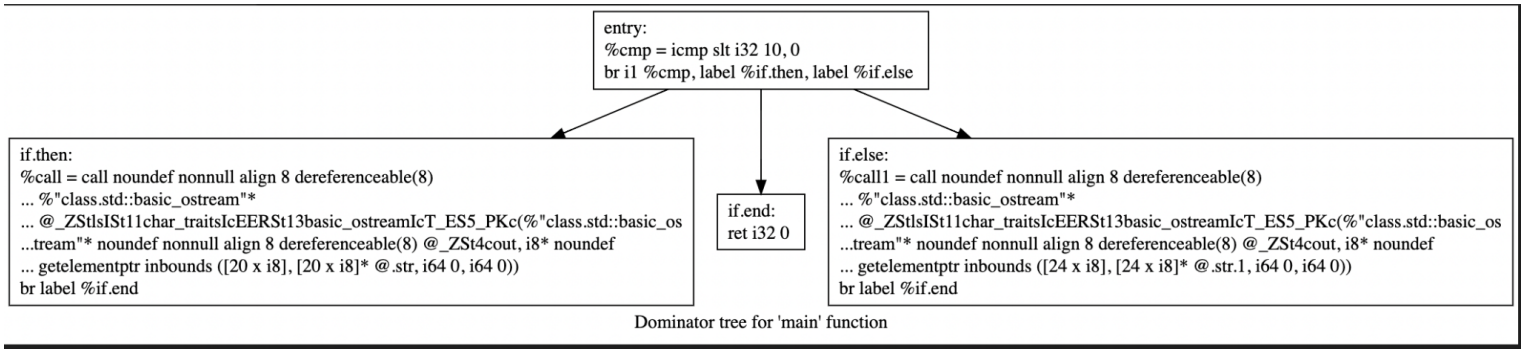
CFG for 'main' function

# Dom Tree for Deadcode.cpp (Unoptimized)



Dominator tree for 'main' function

# Dom Tree for Deadcode.cpp (Optimized)



Dominator tree for 'main' function

## CFG for Constprop.cpp (Unoptimized)

```
entry:
%retval = alloca i32, align 4
%x = alloca i32, align 4
%b = alloca i32, align 4
store i32 0, i32* %retval, align 4
store i32 10, i32* %x, align 4
%0 = load i32, i32* %x, align 4
%div = sdiv i32 %0, 2
store i32 %div, i32* %b, align 4
%call = call noundef nonnull align 8 dereferenceable(8)
... %"class.std::basic_ostream"*
... @_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc(%"class.std::basic_os
...tream"* noundef nonnull align 8 dereferenceable(8) @_ZSt4cout, i8* noundef
... getelementptr inbounds ([11 x i8], [11 x i8]* @.str, i64 0, i64 0))
%1 = load i32, i32* %b, align 4
%call1 = call noundef nonnull align 8 dereferenceable(8)
... %"class.std::basic_ostream"* @_ZNSolsEi(%"class.std::basic_ostream"* noundef
... nonnull align 8 dereferenceable(8) %call, i32 noundef %1)
ret i32 0
```

CFG for 'main' function

## CFG for Constprop.cpp (Optimized)

```
entry:
%call = call noundef nonnull align 8 dereferenceable(8)
... %"class.std::basic_ostream"*
... @_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc(%"class.std::basic_os
...tream"* noundef nonnull align 8 dereferenceable(8) @_ZSt4cout, i8* noundef
... getelementptr inbounds ([11 x i8], [11 x i8]* @.str, i64 0, i64 0))
%call1 = call noundef nonnull align 8 dereferenceable(8)
... %"class.std::basic_ostream"* @_ZNSolsEi(%"class.std::basic_ostream"* noundef
... nonnull align 8 dereferenceable(8) %call, i32 noundef 5)
ret i32 0
```

CFG for 'main' function

## DOM Tree for Constprop.cpp (Unoptimized)

```
entry:
%retval = alloca i32, align 4
%x = alloca i32, align 4
%b = alloca i32, align 4
store i32 0, i32* %retval, align 4
store i32 10, i32* %x, align 4
%0 = load i32, i32* %x, align 4
%div = sdiv i32 %0, 2
store i32 %div, i32* %b, align 4
%call = call noundef nonnull align 8 dereferenceable(8)
... %"class.std::basic_ostream"*
... @_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc(%"class.std::basic_os
...tream"* noundef nonnull align 8 dereferenceable(8) @_ZSt4cout, i8* noundef
... getelementptr inbounds ([11 x i8], [11 x i8]* @.str, i64 0, i64 0))
%1 = load i32, i32* %b, align 4
%call1 = call noundef nonnull align 8 dereferenceable(8)
... %"class.std::basic_ostream"* @_ZNSolsEi(%"class.std::basic_ostream"* noundef
... nonnull align 8 dereferenceable(8) %call, i32 noundef %1)
ret i32 0
```

Dominator tree for 'main' function

## DOM Tree for Constprop.cpp (Optimized)

```
entry:
%call = call noundef nonnull align 8 dereferenceable(8)
... %"class.std::basic_ostream"*
... @_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc(%"class.std::basic_os
...tream"* noundef nonnull align 8 dereferenceable(8) @_ZSt4cout, i8* noundef
... getelementptr inbounds ([11 x i8], [11 x i8]* @.str, i64 0, i64 0))
%call1 = call noundef nonnull align 8 dereferenceable(8)
... %"class.std::basic_ostream"* @_ZNSolsEi(%"class.std::basic_ostream"* noundef
... nonnull align 8 dereferenceable(8) %call, i32 noundef 5)
ret i32 0
```

Dominator tree for 'main' function

It is clearly visible after observing the Dom Tree and CFG for optimized and unoptimized code that it has reduced the instructions by a considerable amount. We have provided the .dot files for CFG as well as the Dom tree of the specified programs in our zip file.

---

## 4. c) Explore OPT tools like llvm-as, llvm-dis, llvm-link, llc, opt

**llvm/tools** – It contains executables built out of the libraries present in llvm/lib. Some important tools built are : opt, llvm-as, llvm-dis, llvm-link, lli, llc, etc.

a. **llvm-as** – The LLVM assembler tool transforms human readable LLVM assembly to the LLVM bitcode. (.ll → .bc)
b. **llvm-dis** – The LLVM disassembler tool transforms LLVM bitcode to human readable LLVM assembly. (.bc → .ll)
c. **llvm-link** – Links multiple LLVM modules into a single program.
d. **llc** – It is basically an LLVM backend compiler, which translates LLVM bitcode(.bc) to a native code assembly file(.s).
e. **opt** – It reads LLVM bitcode, and then applies a series of LLVM to LLVM transformations (which are given out on the command line), and which outputs the resultant bitcode.