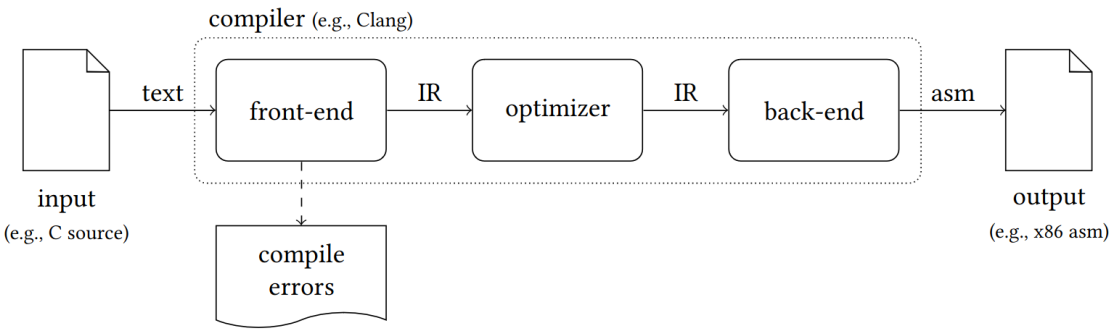
LLVM

- **What is LLVM?**

LLVM (Low Level Virtual Machine) is a compiler infrastructure project that provides a set of modular and reusable compiler and tool-chain technologies. it is now used widely in industry and academia.

The LLVM project includes a variety of components, such as a front-end for different programming languages, a mid-level intermediate representation (IR), optimization passes, code generators, debuggers, and a just-in-time (JIT) compiler. The LLVM IR is a platform-independent representation of program code that can be transformed by optimization passes and ultimately compiled into machine code for specific target architectures.

LLVM is designed to be modular, flexible, and extensible, allowing developers to create custom tool-chains and adapt them to their specific needs. It supports a variety of programming languages, including C, C++, Objective-C, Swift, Rust, and others, and it can target a wide range of hardware platforms, from embedded systems to supercomputers.



- **How to install LLVM**

If you are using any Linux-based operating system then use the following commands, otherwise find it yourself, that how to install in your system.

```
$ sudo apt-get update
$ sudo apt-get install clang llvm
```

- **Test on *HelloWorld* program**

Lets consider a simple hello world program in c.

```
//helloworld.c
#include <stdio.h>
void main()
{
    printf("Hello World\n");
}
```

Now the following command can used for generate LLVM IR from the above C code.

```
$ clang -S -emit-llvm helloworld.c -o helloworld.ll
```

Now the `helloworld.ll` look like this :

```
; ModuleID = 'helloworld.c'
source_filename = "helloworld.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"
@.str = private unnamed_addr constant [13 x i8] c"Hello World\0A\00", align 1
; Function Attrs: noinline nounwind optnone uwtable
define void @main() #0 {
  %1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([13 x i8], [13 x
i8]* @.str, i32 0, i32 0))
  ret void
}
declare i32 @printf(i8*, ...) #1
attributes #0 = { ........ }
attributes #1 = { ........ }
!llvm.module.flags = !{!0}
!llvm.ident = !{!1}
!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 6.0.0-1ubuntu2 (tags/RELEASE_600/final)"}
```

So now the question is that what is this above outcome (`helloworld.ll`), why we need this?

Actually the `helloworld.ll` contain **LLVM IR** code, a mid-level intermediate representation. It is a low-level, platform-independent code representation used by the LLVM compiler infrastructure. It is an intermediate language that serves as an interface between the front-end of a compiler (which translates source code into a higher-level representation) and the back-end (which generates machine code for a specific target architecture).

The LLVM IR is designed to be highly portable and to support a wide variety of optimization techniques. It is also highly modular, which means that different parts of a program can be compiled and optimized independently, making it easier to generate efficient code.

The LLVM IR is similar to assembly language in that it is a textual representation of low-level code. However, it is more structured and abstract than assembly language, and it includes features such as control-flow graphs, data-flow analysis, and type information. This makes it easier for the LLVM compiler to perform sophisticated analyses and optimizations.

One of the key benefits of using LLVM IR is that it allows developers to write code in one language and then compile it for multiple target architectures without having to rewrite the entire program for each target. This can save time and effort, and it can make it easier to develop and maintain high-performance code

To execute the above intermediate code (`helloworld.ll`), you have to generate the executable file by -

```
$ clang helloworld.ll -o helloworld
```

Finally execute by -

```
$ ./helloworld
```

- **Other commands used in LLVM**

```
opt        //optimization
llvm-as    //convert to .ll file to bytecode file
llvm-dis    // convert byte code to .ll code
llvm-link    // linking two .ll file
.....
```

Uses of Optimization Technique with LLVM Pass

Problem 1: Write a program using LLVM pass to display the instructions and it's count in a given C program.

Solution:

Lets take the helloworld program again as victim C program. Now the following code used to solve that problem using LLVM Pass.

```
//instruction-counter.cpp
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/BasicBlock.h"
#include "llvm/IR/Instructions.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
    struct InstructionCounter : public FunctionPass {
    static char ID;
    InstructionCounter() : FunctionPass(ID) {}
    bool runOnFunction(Function &F) override {
      errs() << "Function " << F.getName() << ":\n";
      std::map<std::string, unsigned int> instructionCountMap;
      for (auto& BB : F) {
        for (auto& I : BB) {
          std::string opcodeName = I.getOpcodeName();
          instructionCountMap[opcodeName]++;
        }
      }
      for (auto& countPair : instructionCountMap) {
        errs() << countPair.second << " " << countPair.first << "
instructions\n";
      }
      errs() << "\n";
      return false;
    }
  };
}

char InstructionCounter::ID = 0;
static RegisterPass<InstructionCounter> X("instruction-counter", "Instruction
Counter Pass");
```

To compile the above code you have to execute -

```
$ clang++ -shared -o instruction-counter.so instruction-counter.cpp `llvm-config
--cxxflags --ldflags --libs` -fPIC
```

Now to get the final result -

```
$ opt -load ./instruction-counter.so -instruction-counter < helloworld.ll >
/dev/null
```

Output -

```
Function main:
1 call instructions
1 ret instructions
```

- **Breakdown of the above code**

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/BasicBlock.h"
#include "llvm/IR/Instructions.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;
```

These are the necessary header files and namespaces to include for developing an LLVM pass. They provide access to the LLVM API, including `Function`, `BasicBlock`, `Instruction`, and `raw_ostream`.

```
namespace {
  struct InstructionCounter : public FunctionPass {
    static char ID;
    InstructionCounter() : FunctionPass(ID) {}
    bool runOnFunction(Function &F) override {
      // ...
      return false;
    }
  };
}

char InstructionCounter::ID = 0;
static RegisterPass<InstructionCounter> X("instruction-counter", "Instruction
Counter Pass");
```

This code defines an `InstructionCounter` pass. It is a subclass of `FunctionPass`, which means that it operates on each function in the program. The `runOnFunction` method is the entry point for the pass and is called once for each function in the program.

The `InstructionCounter` pass counts the number of instructions of each type in a function and prints the counts for each function. The `instructionCountMap` variable is a map that stores the count of each instruction type. The pass iterates over each basic block in the function, and for each instruction in the basic block, it increments the count for that instruction type in the

`instructionCountMap` variable.

The `RegisterPass` macro is used to register the `InstructionCounter` pass with LLVM. This allows the pass to be loaded and run on LLVM bitcode files using the `opt` tool.

```
bool runOnFunction(Function &F) override {
  errs() << "Function " << F.getName() << ":\n";
  std::map<std::string, unsigned int> instructionCountMap;
  for (auto& BB : F) {
    for (auto& I : BB) {
      std::string opcodeName = I.getOpcodeName();
      instructionCountMap[opcodeName]++;
    }
  }
  for (auto& countPair : instructionCountMap) {
    errs() << countPair.second << " " << countPair.first << " instructions\n";
  }
  errs() << "\n";
  return false;
}
```

This method is called once for each function in the program. It prints the name of the function, iterates over each basic block in the function, and counts the number of instructions of each type using a `std::map`. Finally, it prints the counts for each instruction type.

```
char InstructionCounter::ID = 0;
```

This line initializes the `ID` field of the `InstructionCounter` pass to 0.

```
static RegisterPass<InstructionCounter> X("instruction-counter", "Instruction
Counter Pass");
```

This line registers the `InstructionCounter` pass with LLVM using the `RegisterPass` macro. The first argument is the pass itself (`InstructionCounter`), and the second argument is a string that describes the pass. The string is used for debugging and informational purposes.

Problem 2: Write a program using LLVM pass to instrument the loops in a given C program.

Solution:

Lets consider the victim C file as follows:

```
//loop.c
#include <stdio.h>
void hook()
{
    printf("Loop here\n");
}
void main()
{
    int i;
    for(i=0; i<10; i++)
    {
```

```
        printf("Value: %d\n", i);
    }
}
```

Now the following code for instrument the loop call with the function `hook()`

```
//loop-instrumenter.cpp
#include "llvm/Analysis/LoopPass.h"
#include "llvm/Analysis/LoopInfo.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/Module.h"
#include "llvm/Pass.h"
#include "llvm/Support/raw_ostream.h"
#include <vector>

using namespace llvm;

namespace {
    struct LoopInstrumenter : public LoopPass {
    static char ID; // Pass identification, replacement for typeid
    LoopInstrumenter() : LoopPass(ID) {}
    bool runOnLoop(Loop *L, LPPassManager &LPM) override {
        // Get the function and module contexts
        Function *F = L->getHeader()->getParent();
        Module *M = F->getParent();

        // Get the loop header basic block
        BasicBlock *HeaderBB = L->getHeader();

        // Get the type of the function that we want to call
        FunctionType *FuncType = FunctionType::get(Type::getVoidTy(M-
>getContext()), false);

        // Get the function that we want to call
        Function *Func = cast<Function>(M->getOrInsertFunction("hook",
FuncType));

        // Create the call instruction and insert it at the beginning of the loop
header
        IRBuilder<> Builder(HeaderBB, HeaderBB->begin());
        Builder.CreateCall(Func);

        // Mark the function as modified
        return true;
    }

    void getAnalysisUsage(AnalysisUsage &AU) const override {
        AU.addRequired<LoopInfoWrapperPass>();
        AU.setPreservesAll();
    }
  };
}

char LoopInstrumenter::ID = 0;
```

```
static RegisterPass<LoopInstrumenter> X("loop-instrumenter", "Instrument loops
with a function call");
```

To compile the above code you have execute

```
$ clang++ -shared -o loop-instrumenter.so loop-instrumenter.cpp `llvm-config --
cxxflags --ldflags --libs` -fPIC
```

Now to get the final result -

```
$ opt -load ./loop-instrumenter.so -loop-instrumenter < loop.ll > inst-loop.ll
```

Finally execute by -

```
$ clang inst-loop.ll -o inst-loop
$ ./inst-loop
```

or

```
$ lli instrument-loop.ll
```

Output -

```
Loop here
Value: 0
Loop here
Value: 1
Loop here
Value: 2
Loop here
Value: 3
Loop here
Value: 4
Loop here
Value: 5
Loop here
Value: 6
Loop here
Value: 7
Loop here
Value: 8
Loop here
Value: 9
Loop here
```

Assignment : Write a program to instrument the loops (before and after) in a particular C program using LLVM pass and the hook functions from another header file.

Example:

Lets consider the following header file contain hook function, that used for loop instrumentation.

```
//hooks.h
void before()
{
    printf("Before Loop\n");
}

void before()
{
    printf("After Loop\n");
}
```

And the following code will be the victim code.

```
//loop.c
#include <stdio.h>
void main()
{
    int i;
    for(i=0; i<5; i++)
    {
        printf("Value: %d\n", i);
    }
}
```

The following output will desire after instrumentation -

```
Before Loop
Value: 0
Value: 1
Value: 2
Value: 3
Value: 4
After Loop
```