
# UE19CS351 - Compiler Design

Session: Jan 2022 – May 2022

## MANUAL

Semester: VI

Course Anchor : Prof. Preet Kanwal

Teaching Assistant : Anirudh H M

# LLVM DOCUMENTATION

## TABLE OF CONTENTS

# 1. INTERMEDIATE CODE GENERATION

- The structure of a compiler is such that after lexical, syntax and semantic analysis of the source code, an intermediate representation is generated and optimized, from which target machine code is created based on the desired instruction set. **(Refer Figure 1)**
- The LLVM Project, in a nutshell, lays significant emphasis on a very important stage, i.e, Intermediate Code Generation.

  *Why is Intermediate Code Generation so important?*
  -> Without Intermediate Code Generation, we would need to convert source language directly into target language which leads to very high dependency between the front-end and the back-end portions of the compiler.
  - ❖ For example, for **m** source (programming) languages and **n** target (assembly) languages, **m*n** compilers would be needed for every source-target language conversion. However, with Intermediate Code Generation, only **m+n** compilers would be required.

  -> Thus, the introduction of an Intermediate Code Generation Layer is seen to eliminate the dependence between the front-end and back-end of the compiler.

- Intermediate Representation (IR) can be:-
  1) An Actual Language (**LLVM IR**)
  2) Internal Data Structures that are shared by phases of a compiler (**Quadruples, Triples, Indirect Triples**)
- The choice and design of an Intermediate Representation varies from compiler to compiler. **LLVM uses an actual language for its intermediate representation.**

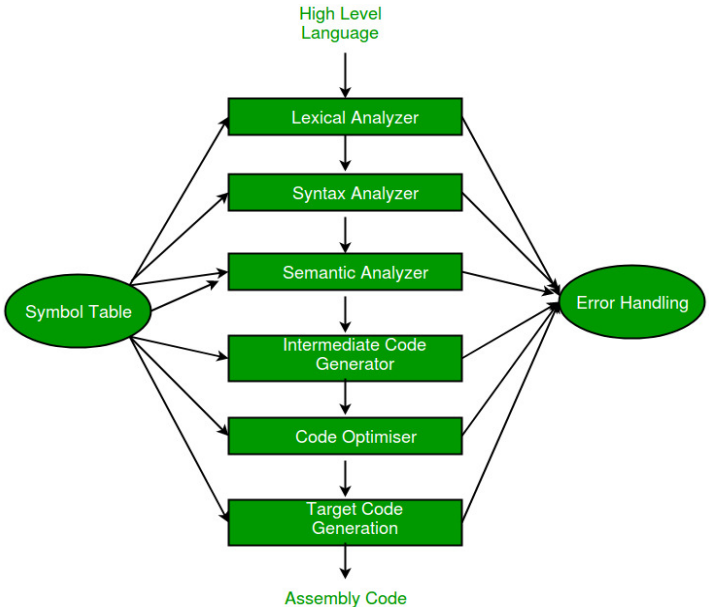
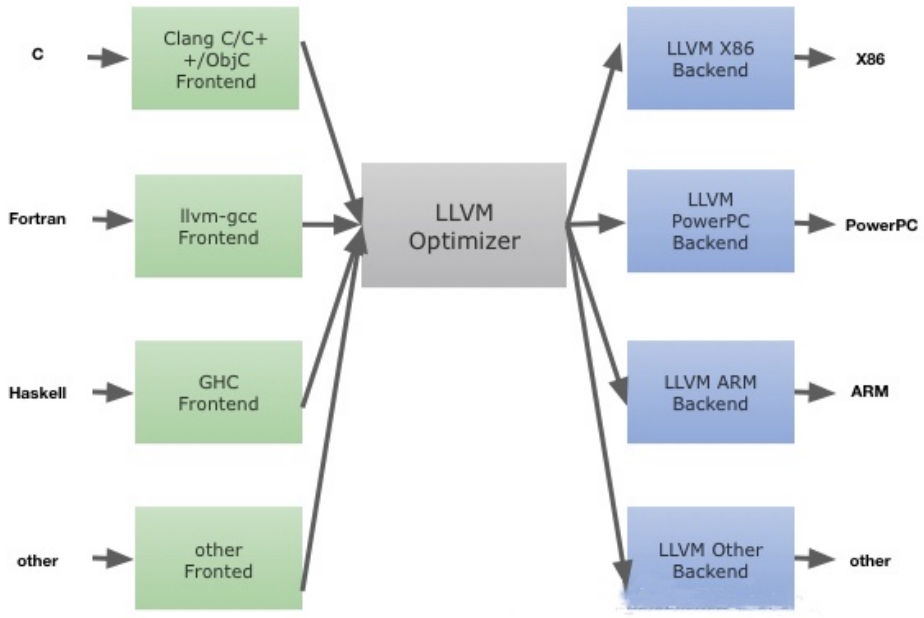



**Figure 1:- Phases Of A Compiler**

# 2. ABOUT LLVM

- LLVM was started as a research project at UIUC by **Chris Lattner**. LLVM was designed as a compiler framework to support lifelong program analysis and transformations.
- LLVM is a set of compiler and toolchain technologies, which can be used to develop a front end for any programming language and a back end for any instruction set architecture. LLVM is designed around a language-independent intermediate representation (IR) that serves as a portable, high-level assembly language that can be optimized with a variety of transformations over multiple passes.
- LLVM is written in C++ and is designed for compile-time, link-time, run-time, and "idle-time" optimization.
- Originally implemented for C and C++, the language-agnostic design of LLVM has since spawned a wide variety of front ends: languages with compilers that use LLVM (or which do not directly use LLVM but can generate compiled programs as LLVM IR) include ActionScript, Ada, C#, Common Lisp, PicoLisp, Crystal, CUDA, D, Delphi, Dylan, Forth, Fortran, Free Basic, Free Pascal, Graphical G, Halide, Haskell, Java bytecode, Julia, Kotlin, Lua, Objective-C, OpenCL, PostgreSQL's SQL and PLpgSQL, Ruby, Rust, Scala, Swift, XC, Xojo and Zig.
- LLVM offers intermediate code generation for programming languages such as C,C++,Fortran,Haskell,Julia,Objective-C,Rust and Swift using **various front-ends**.
    - For example, compilers like **Clang** use LLVM by supporting and parsing C, C++, Objective-C code and translating it into a representation suitable for LLVM.
- On the backend, LLVM IR supports a number of instruction sets including **ARM**, **RISC**, **x86** amongst others.
- **Basically, LLVM acts as a bridge that connects the front-end analysis to the back-end architecture-specific target code generation by itself providing transformations and optimizations over the Intermediate Representation.**



# 3. LLVM ARCHITECTURE

➢ LLVM basically splits a compiler into 3 separate modules with functions as below:-

| MODULE | FUNCTION |
|---|---|
| 1) **Front-End** | Parsing Programming Language |
| 2) **Middle-End** | Analyzing and Optimizing LLVM IR |
| 3) **Back-End** | Producing machine code for programs |

➢ The workflow of source code via LLVM Architecture is demonstrated using Figure 2

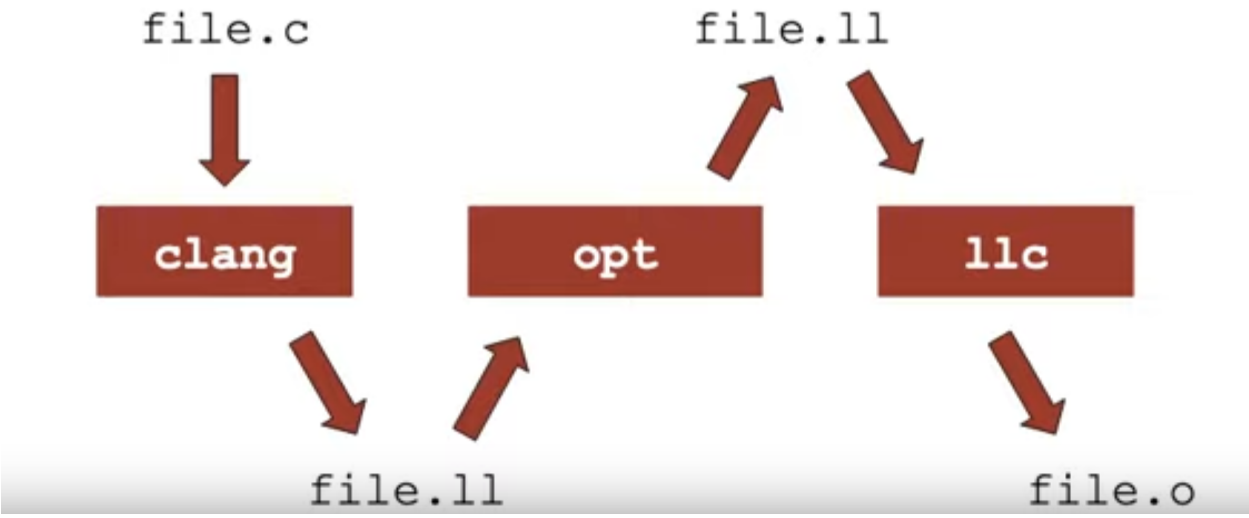



**Figure 2:- Source Code Workflow Using LLVM Architecture**

**NOTE**:-

- Clang is a **front-end** compiler for C,C++ and Objective C (C Family Of Languages) programming languages.
- Opt stands for Optimizer. This is the **middle-end** module that further optimizes and transforms the Intermediate Representation.
- LLC stands for LLVM Static Compiler. It is the **back-end** module that compiles IR into assembly language for specified architecture.

# 4. ADVANTAGES OF LLVM

## 4.1 REUSE OF CODE

★ By translating other languages into LLVM IR, one can leverage the transformations and optimizations already developed by LLVM. This helps in reuse of code across languages and compilers, enabling all languages to benefit from a single compiler.

★ Nowadays, modern applications have components written in multiple different languages. Having a common IR helps tremendously in quickly and optimally compiling the entire application. (Refer Figure 6)

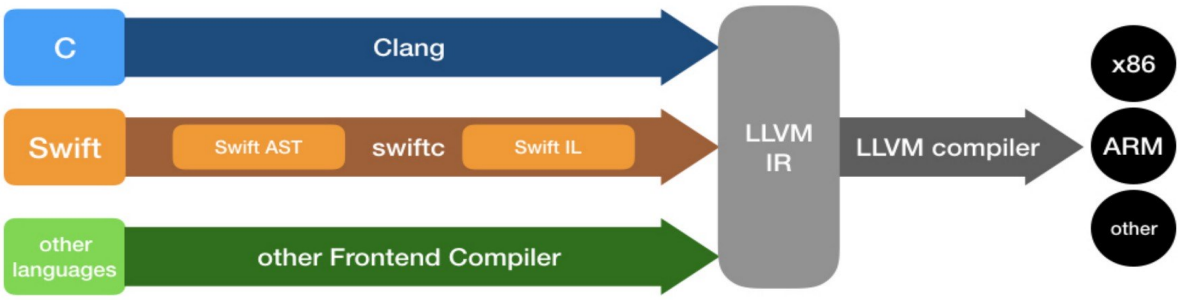



**Figure 6:- Advantages Of Using LLVM**

*Unlike Clang, why does Swift have another layer of abstraction before the LLVM IR?*
-> A lot of important tasks like memory allocation in high level languages like Swift are left in the hands of the compiler. A lot of abstraction would be lost in case a language like Swift is being compiled. (i.e Swift -> LLVM IR -> Assembly).
-> To avoid this, Swift (developed by Chris Lattner, the creator of LLVM) came up with another intermediate layer between Swift and LLVM IR and named it SIL (Swift Intermediate Language). As SIL is of a much higher level, it can understand Higher Level details of the program much better than LLVM IR.

## 4.2 INTEGRATING NEW OPTIMIZATIONS & EASE OF USAGE

★ Many compilers like GCC are **monolithic** in nature, that is, they have their own specific set of transformations and optimizations unique to their own architecture. Because of this, integrating newer transformations and optimizations is quite tedious.

★ The monolithic compiler acts as a black box during the entire process of converting source language into target machine code.

★ LLVM, unlike monolithic GCC, is a set of libraries and tools as well as a standardized intermediate representation that supports the development of other compilers on top of it while also generating target assembly code based on the specific instruction set architectures. Thus, integrating newer transformations and optimizations is easier. Apart from this, LLVM IR is richer, more expressive and much more flexible than GCC IR (GIMPLE)

★ A custom optimization can be integrated in the compilation process by adding a LLVM Optimization Pass. After registering the pass, the custom pass will be executed.

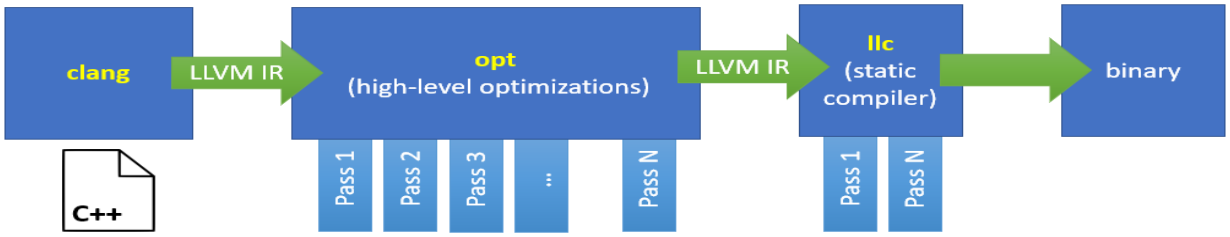



**Figure 7:- Adding Custom Optimizations**

★ LLVM offers flags to control the level of optimization (pre-defined set of passes) during compilation.
★ LLVM also helps in easy mapping of errors and indexing compiler output. This is done by preserving the **unoptimized parsed syntax**, which is then used by third party tools to trace back the transformations to the source code.
★ LLVM's architecture-neutral design makes it easier to support hardware of all kinds, present and future. It also provides a library-based architecture, so that the compiler interoperates with other tools (IDE's) that interact with source code, unlike GCC.

# 5. ADDITIONAL READING

- **LLVM: A Compilation Framework For Lifelong Program Analysis & Transformation**
  Chris Lattner, Vikram Adve, CGO, 2004.
  LLVM: A Compilation Framework For Lifelong Program Analysis & Transformation | IEEE Conference Publication | IEEE Xplore
- Students are encouraged to contribute towards the open source project.
  - The open source project repository The LLVM Compiler Infrastructure
  - The guide for open source contribution Contributing to LLVM - Documentation