# Chapter 1

# Introduction to LLVM

## Introduction

In this chapter, we will go through the high level architecture of LLVM and its components in brief. The goal of this chapter is to familiarise users with LLVM toolchain and key concepts like AST, IR, LibTooling etc. The takeaway would be to use the llvm/clang framework for compilation, static analysis, viewing AST, and dump LLVM IR. High level understanding of important libraries and their use.

Compilers are the backbone of today's high-tech tools and products. Creating a compiler from scratch is not a trivial job. It takes a huge amount of time and effort to write a compiler for a language and convert the source code into machine code. Contemporary software devices demand the highest possible speed of execution. And for that the compiler needs to optimize the source code during compilation in the highest possible manner.

Compilation broadly includes various stages such as preprocessing, Lexing, Semantic Analysis, Object Files Creation, Linking and Executable/Library creation. During this process, the compiler also needs to handle syntax & semantic errors in source code and notify it to the user. And along with this also do optimization to generate optimized executable binary. Writing a production grade compiler end-to-end is not a trivial task. It's a lot of effort and has a higher cost. There are very good compilers available today but they have limitations such as

* Traditional compilers are built on very old technology and lack support for cross-file optimization and JIT.
* They are monolithic compilers and hence consist of non-reusable components.
* Very difficult to understand and maintain.
* Limited hooks for optimization

## Birth of LLVM

LLVM - Low Level Virtual Machine! Don't go by the name. It is not a java-like virtual machine. It is basically a modular and reusable collection of libraries which helps convert a source code into a machine code providing granular level control over various compilation stages. LLVM achieves this with the introduction of Intermediate Representation (IR). LLVM IR is in itself a first class language with a type system. It’s quite low level which makes it source language agnostic. And at the same time abstract away the target architecture making it target agnostic. One can even write programs in IR language itself which can be compiled into various target machines.

We will learn this in detail but let us get a hang of high level architecture of LLVM.

## LLVM architecture – Bird Eye View

LLVM can be broadly classified into three main components: frontend, middle-end and backend.



Figure 1.1 LLVMArchitecture

Frontend -- Source language parsing tools such as Clang, llgo, flang, Rust, and so on. Frontend converts the source language into LLVM IR

Middle-end -- LLVM optimizer, takes IR from the frontend and applies various optimization on IR resulting in optimized IR

Backend -- Target dependent machine code generation This takes an optimized IR and generates target dependent machine code

We will introduce you to each component in brief and its associated libraries in the next few pages.

## Evolution and Strength of LLVM

Before going into the component details, it’s worth taking a look at how LLVM solves the difficult problem of compiler construction in today’s fast paced technology period.

### Evolution

As we delve into the fascinating journey of LLVM, we'll explore its development milestones, key features, and the impact it has had on the programming and compiler design landscape.

**Early Development 2000: Project Inception** LLVM was started as a research project at the University of Illinois at Urbana-Champaign by Vikram Adve and Chris Lattner.

**2003: First Public Release** The first version of LLVM was released, laying the foundation for a robust intermediate representation (IR) and a modular compiler architecture.

**2006: Clang Development Begins** Development of Clang, a new C/C++/Objective-C frontend for LLVM, was initiated. Clang aimed to provide faster compilation and better diagnostics compared to GCC.

**2009: Clang Maturity** Clang became mature enough to compile large projects, including LLVM itself, and started to be used in production environments.

**2010: Integration into Xcode** Apple integrated LLVM into its Xcode development environment, replacing GCC and marking a significant endorsement of LLVM’s capabilities.

**2011: LLVM Becomes a Brand** The name "LLVM" stopped being an acronym and became an umbrella term for the entire suite of tools and technologies under the LLVM project.

**2011: LLVM Compiler Infrastructure** The LLVM Compiler Infrastructure was established as a comprehensive framework for a variety of language frontends, supporting extensive community contributions.

**2014: LLVM 3.5 Release** This release introduced numerous enhancements, including better code generation, new optimization passes, and support for more architectures.

**2020s: Continued Evolution** LLVM has maintained a steady release cycle, continuously improving performance, usability, and support for emerging hardware architectures. It’s used in major projects such asAndroid, Firefox, and Visual Studio Code.

### Key Components in LLVM

Understanding the core components of LLVM is crucial to appreciating its versatility and power. In this chapter, we will break down these components and examine how they interact to form a robust framework for compiler development.

**LLVM Core Libraries:** Support for various optimization passes, code generation, and machine-level code manipulation.

**Clang:** A frontend for C, C++, and Objective-C, leveraging LLVM for backend processing.

**LLVM IR:** A powerful intermediate representation used for optimization and code generation.

**LLDB:** A high-performance debugger built using LLVM libraries.

**libc++:** A C++ standard library implementation designed for high performance.

### Strength of LLVM

LLVM addresses several traditional compiler issues with its innovative design and architecture. Some key issues that LLVM helps to resolve are as follows:

* **Modularity and Reusability Traditional Issue:** Traditional compilers are often monolithic and tightly coupled, making it difficult to reuse components or extend functionality.

**LLVM Solution:** LLVM’s modular design allows developers to use and extend individual components, such as the front-end, optimizer, or back-end, independently. This promotes reusability and flexibility.

* **Persistence Intermediate Representation (IR) Traditional Issue:** Different phases of compilation in traditional compilers often use different intermediate representations, leading to complexity and inefficiency.

**LLVM Solution:** LLVM uses a single, language-agnostic Intermediate Representation (IR) throughout the compilation process. This IR can be saved to disk and later used for optimization and code generation as per requirement. Which enables the support for JIT. This unified IR simplifies the implementation of optimizations and transformations.

* **Uniform, whole program compilation and optimization Traditional Issue:** Traditional compilers perform optimizations at specific stages, which can miss opportunities for cross-phase optimizations.

**LLVM Solution:** LLVM’s IR allows for continuous and aggressive optimizations at different stages: compile-time, link-time, runtime, and idle-time. This results in more efficient and optimized code. The unified IR generated by different language frontends can be compiled and optimized together.

* **Portability Traditional Issue:** Writing code for multiple architectures often requires significant changes and maintenance in traditional compilers.

**LLVM Solution:** LLVM’s IR serves as a portable assembly language. By targeting the IR, code can be easily retargeted to different architectures with minimal changes, enhancing portability. The LLVM IR is agnostic to runtime models such asJVM, .Net, python. It handles the various critical hardware aspects such as type checking, exception handling, memory management in an agnostic way.

* **Code Generation Traditional Issue:** Code generation in traditional compilers is often tightly integrated with the front-end, making it difficult to support multiple languages.

**LLVM Solution:** LLVM decouples the front-end from the back-end. Different front-ends can generate the same IR, which can then be compiled to various target architectures, enabling support for multiple languages.

* **Just-In-Time (JIT) Compilation Traditional Issue:** Traditional compilers are typically designed for ahead-of-time (AOT) compilation and may not support runtime optimizations.

**LLVM Solution:** LLVM provides robust JIT compilation support, allowing code to be compiled and optimized at runtime. This enables dynamic languages and runtime optimizations based on actual execution patterns. This is also called profile guided optimization. Optimization based on profile data and usage patterns.

* **Extensibility and Research Traditional Issue:** Traditional compilers can be challenging to extend for new research and experimental features.

**LLVM Solution:** LLVM’s modular architecture and open-source nature make it an excellent platform for compiler research and experimentation. Developers can easily extend and modify LLVM to explore new ideas.

LLVM's innovative design addresses many of the limitations of traditional compilers by promoting modularity, reusability, portability, and advanced optimizations. Its impact on the field of compiler technology is significant, providing a flexible and powerful infrastructure for a wide range of applications. This is how LLVM supports lifelong analysis of programs.

## Frontend – Clang

Though many frontends available for LLVM as of today such as Clang, Flang, Rust, Swift, and so on. We will briefly discuss the most mature and widely used frontend Clang in this chapter. Clang is a LLVM frontend for C family language such as C/C++/Obj-C. It’s a drop in replacement for GCC. It has evolved over the years into a production grade compiler with more and more industry adoption.

One of the distinguishing features of clang is its expressive diagnostics, modular architecture and reusable components. Diagnostic messages such as errors and warnings are very verbose and easy to understand. Users do not need to look at the code to understand the issue. The location of the exact point of error is captured at character level. Which makes error resolution fast and easy.

*Example:*

> // example.c

> #include <stdio.h>

>

> void printArrayElement(int \*arr, int size, int index) {

> printf("Element at index %d is %d\n", index, arr[index]);

> }

>

> int main() {

> int arr[5] = {1, 2, 3, 4, 5};

> printArrayElement(arr, 5, 10); // Out-of-bounds access

> return 0;

> }

*Output*:

> $ clang -fsyntax-only example.c

> example.c:5:38: warning: array index 10 is past the end of the array (which contains 5 elements) [-Warray-bounds]

> printf("Element at index %d is %d\n", index, arr[index]);

> ^ ~~~~~~~

> example.c:10:12: note: array 'arr' declared here

> int arr[5] = {1, 2, 3, 4, 5};

> ^

> 1 warning generated.

Clang can be used as a compiler to build programs. Or it can be used as a library to implement custom compiler tools such as code checker, syntax highlighter, IDE integration, and so on.

### Important libraries in Clang

To harness the full potential of LLVM, it is essential to understand the pivotal libraries that form its backbone. This chapter will guide you through the most significant libraries within LLVM, detailing their functionalities and how they contribute to the overall system.

**libbasic** -- It handles the source file information such as location, filesystem, diagnostics, and so on

**liblex** -- Preprocessing library (macro expansion, pragma, and so on)

**libparse** -- Parsing of source code

**libsema** -- Semantic Analysis, AST generation

**libcodegen** -- Convert the AST into LLVM IR (it is called lowering)

**libanalysis** -- Static Analysis of Source code.

**clang** – is a driver program which invokes various libs depending on the options specified.

We will go through the details on each library in later chapters.  
Let us get familiar with the compilation stages first.

### Pre-Processing

To invoke the Clang pre-processor and see the expanded output, you can use the clang command with the -E flag:

$clang -E llvm-demo.c

output:

<TODO>

This command preprocesses the file example.c and outputs the result, including all macro expansions and file inclusions.

### AST generation

Use the Clang compiler with the -Xclang -ast-dump options to generate and display the AST

$clang -Xclang -ast-dump -fsyntax-only llvm-demo.c

output:

<TODO>

### Syntax Checking

Clang uses -fsyntax-only option to perform syntax checks without generating any output files. This is useful for quickly identifying syntax errors in your code. You can use additional flags to enhance the diagnostics provided by Clang. For example: -Wall: Enables all the commonly used warning messages. -Wextra: Enables additional warning messages not included by -Wall.

$clang -fsyntax-only -Wall -Wextra llvm-demo.c

### IR generation

To generate LLVM IR from source file, use options -S -emit-llvm

$clang -S -emit-llvm llvm-demo.c -o llvm-demo.ll

## Abstract Syntax Tree

Most language front ends will convert the source code into its in memory representation. It is called Abstract Syntax Tree (AST). AST of any language is highly coupled with source language syntax. As it's used for a lot of source analysis tools.

Clang's AST is very granular. It captures all the details to the level of single parenthesis. AST can be used for developing a variety of source analysis and refactoring tools.

Let us see how a AST looks like:

To dump the AST for a source file we use the flag -ast-dump to clang the compiler. When I say pass option to clang compiler use option -Xclang before that option. Similarly, for other component specific options.

> $clang -Xclang -ast-dump -fsyntax-only llvm-demo.c

> `-FunctionDecl 0x56360af2e4d8 <llvm-demo.c:3:1, line:6:1> line:3:5 main 'int ()'

> `-CompoundStmt 0x56360af2e700 <col:12, line:6:1>

> |-CallExpr 0x56360af2e678 <line:4:5, col:47> 'int'

> | |-ImplicitCastExpr 0x56360af2e660 <col:5> 'int (\*)(const char \*, ...)' <FunctionToPointerDecay>

> | | `-DeclRefExpr 0x56360af2e580 <col:5> 'int (const char \*, ...)' Function 0x56360aeeda98 'printf' 'int (const char \*, ...)'

> | `-ImplicitCastExpr 0x56360af2e6b8 <col:12> 'const char \*' <NoOp>

> > | `-ImplicitCastExpr 0x56360af2e6a0 <col:12> 'char \*' <ArrayToPointerDecay>

> | `-StringLiteral 0x56360af2e5e0 <col:12> 'char[33]' lvalue "Welcome to the LLVM universe!!!\n"

> `-ReturnStmt 0x56360af2e6f0 <line:5:5, col:12>

> `-IntegerLiteral 0x56360af2e6d0 <col:12> 'int' 0

**-Xclang -ast-dump**

Here, -Xclang represents clang front-end and ast-dump option is for the clang frontend. Similarly additional options can be given to individual components such as pre-processor, semantic analyser, optimizer. For example, -Xpreprocessor is a pre-processor, -Xanalyzer is a static analyser and so on.

**-fsyntax-only**

This option tells Clang to only check the syntax of the code without generating any output files. It's useful for quickly verifying that your code is syntactically correct.

In the above example, I have omitted the AST for the stdio.h header file. As the C/C++ source files generally have #include, after pre-processing the all the included header are merged with the corresponding source file. So clang will dump AST for all the source code irrespective of whether it’s from the header of actual user source code. AST for the header files is mostly not required so we can add a filter option to clang to dump AST for only matching functions.

The AST generated above is self-explanatory enough. Every line of source code is captured in AST with minute details. We will go through the specific details in AST in later chapters.

## LibTooling

LibTooling is a powerful library within the LLVM ecosystem designed to facilitate the creation of standalone tools that interact with C, C++, and Objective-C code. These tools leverage Clang’s robust parsing, analysis, and transformation capabilities to perform tasks such as static analysis, refactoring, and code generation. This chapter explores the fundamental concepts, setup, and practical applications of LibTooling.

LibTooling offers a structured way to build tools that can parse and manipulate source code. It builds upon Clang’s core libraries to provide a flexible framework for developing custom tooling without requiring deep integration into the Clang compiler itself.

### Key features

Abstract Syntax Tree (AST) Manipulation -- Access and modify the AST of programs.

* Frontend Actions -- Define actions to be performed on the source code.
* Integration with Compilation Database -- Understand and use the environment in which code is compiled.
* Extensibility -- Easily extend and customize for specific needs.

*Example:*

The clang-check which we introduced in the previous chapter is the best example of a tool built using libTooling libraries.

The main component of libTooling is header file *tooling.h* which provides,

* CommonOptionsParser -- This class helps parse common command-line options, including the compilation database and source file paths.
* ClangTool -- This class represents a tool that can run over a set of source files using a specified FrontendAction.
* FrontendAction -- This is an abstract base class for actions that can be performed on the source code during the frontend phase of compilation.
* CompilationDatabase -- This class provides access to the compilation commands used to build the source files, allowing tools to understand the build environment.
* runToolOnCode -- This function allows running a FrontendAction on a code snippet provided in memory, useful for unit testing or quick checks.

We will build a custom tool in this book using the LibTooling framework for a practical understanding.

## Static Analysis using Clang

The Clang Static Analyser is a powerful tool for performing static analysis on C, C++, and Objective-C programs. It helps detect bugs and security vulnerabilities by analysing the source code without actually executing it. Here are some key features and benefits of using the Clang Static Analyser:

### Key Features

In this section, we will explore the standout functionalities that make Clang an indispensable tool for developers aiming to enhance code quality and reliability.

* Path-Sensitive Analysis -- It performs path-sensitive analysis based on symbolic execution, which means it can reason about different execution paths in the code.
* Inter-Procedural Analysis -- The analyser can analyse across function boundaries, providing a more comprehensive view of potential issues.
* Custom Checkers -- You can define custom checkers to detect specific types of bugs or coding patterns.
* Integration with Build Systems -- It can be integrated with build systems using tools like scan-build to automatically analyse code during the build process.
* IDE Integration -- Many Integrated Development Environments (IDEs) support the Clang Static Analyser, allowing developers to run analysis directly within the IDE

We have seen examples of how to invoke static analysers in clang in previous sections. We will again go through the static analysis tool design and check implementation in later chapters. Also we will build custom checks for the existing tool.

## LLVM IR

This is the main ingredient in the LLVM framework on which everything is based on. Typically, the language frontend will lower the source code into LLVM IR. LLVM IR can be represented in 3 ways:

* Text
* in-memory IR
* bit code

We can convert the text to another format and vice versa.

// llvm-demo.c

#include <stdio.h>

int main() {

printf("Welcome to the LLVM universe!!!\n");

return 0;

}

### Source code to ir bit code

> $clang -emit-llvm -c llvm-demo.c -o llvm-demo.bc

### Example IR:

LLVM bc code to human readable ***.ll*** file

> $llvm-dis llvm-demo.bc -o llvm-demo.ll

> ; ModuleID = 'llvm-demo.bc'

> source\_filename = "llvm-demo.c"

> target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-i128:128-f80:128-n8:16:32:64-S128"

> target triple = "x86\_64> -unknown-linux-gnu"

> @.str = private unnamed\_addr constant [33 x i8] c"Welcome to the LLVM universe!!!\0A\00", align 1

> ; Function Attrs: noinline nounwind optnone uwtable

> define dso\_local i32 @main() #0 {

> %1 = alloca i32, align 4

> store i32 0, ptr %1, align 4

> > %2 = call i32 (ptr, ...) @printf(ptr noundef @.str)

> ret i32 0

> }

> declare i32 @printf(ptr noundef, ...) #1

> attributes #0 = { noinline nounwind 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"="+cmov,+cx8,+fxsr,+mmx,+sse,+sse2,+x87" > "tune-cpu"="generic" }

> attributes #1 = { "frame-pointer"="all" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" > "target-features"="+cmov,+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }

> !llvm.module.flags = !{!0, !1, !2, !3, !4}

> !llvm.ident = !{!5}

> !0 = !{i32 1, !"wchar\_size", i32 4}

> !1 = !{i32 8, !"PIC Level", i32 2}

> !2 = !{i32 7, !"PIE Level", i32 2}

> !3 = !{i32 7, !"uwtable", i32 2}

> !4 = !{i32 7, !"frame-pointer", i32 2}

> !5 = !{!"clang version 18.1.8"}

A separate chapter is dedicated to understanding the LLVM IR language. This is just the introduction to get yourself accustomed.

## LLVM Optimizer

It is the most important part of the LLVM framework. LLVM optimizer is part of the middle-end. It takes the compiler generated IR as input and applies various optimizations and generates the optimized IR. The optimizer is modular, meaning it consists of multiple passes that can be run in different orders to achieve the desired optimizations.

> $opt -O2 llvm-demo.bc -o llvm-demo-optimized.bc

### Key Components of LLVM Optimizer

* Analysis Passes – These passes gather information about the code without modifying it. Examples include alias analysis, dependence analysis, and loop analysis.
* Transform Passes – These passes modify the code to improve its performance. Examples include dead code elimination, loop unrolling, and function inlining.
* Utility Passes – These passes perform tasks such as verifying the correctness of the IR or printing information about the code.

LLVM optimizer supports optimization labels from O0 to O3, Os and Oz

O0 -- No optimization

Oz -- aggressive optimization

we can even specify the optimization passes

> $opt -O2 -inline -simplifycfg -o llvm.bc llvm.bc

Here the – simplify CGF option tells optimizer to run only to simplify CFG of function.

## Link Time Optimization

Link Time Optimization (LTO) in LLVM is a powerful technique that allows the compiler to perform optimizations across multiple translation units (source files) at the linking stage. This can lead to better performance and smaller code size by enabling inter-procedural optimizations that are not possible when each source file is compiled separately.

### Working of LTO

This section will explore how LTO works, detailing the process and the significant benefits it brings to code optimization and performance.

**Compilation to Bitcode:** Each source file is compiled into LLVM Intermediate Representation (IR) bitcode instead of directly generating machine code.

**Linking with Bitcode:** During the linking stage, the linker collects all the bitcode files and passes them to the link-time optimizer (libLTO).

**Intermodular Optimizations:** The link-time optimizer performs optimizations across the collected bitcode files, such as function inlining, dead code elimination, and constant propagation.

**Code Generation:** Finally, the optimized bitcode is converted into machine code and linked into the final executable.

To enable LTO with Clang, you can use the -flto flag during compilation and linking. For example:

> $clang -flto -O2 -c file1.c -o file1.o

> $clang -flto -O2 -c file2.c -o file2.o

> $clang -flto -O2 file1.o file2.o -o my\_program$

### ThinLTO

ThinLTO is a scalable and incremental variant of LTO. It divides the optimization process into two phases: a fast, serial thin link step and a parallel backend optimization step3. This makes it more efficient for large projects and allows for faster incremental builds.

Example of ThinLTO To use ThinLTO, you can add the -flto=thin option:

> $clang -flto=thin -O2 -c file1.c -o file1.o

> $clang -flto=thin -O2 -c file2.c -o file2.o

> $clang -flto=thin -O2 file1.o file2.o -o my\_program

### Benefits of LTO

**Improved Performance:** By optimizing across multiple files, LTO can eliminate redundant code and improve runtime performance.

**Reduced Code Size:** LTO can remove unused functions and data, leading to smaller executables.

**Better Optimization:** LTO enables more aggressive optimizations that are not possible when files are compiled separately.

### LLVM Link

'llvm-link' is a utility to link multiple .bc files into a combined .bc file. This combined IR can be given for further processing. This utility does not perform any optimizations.

> $llvm-link file1.bc file2.bc -o combined.bc

## LLVM Code Generation

This is the LLVM backend part and final step in compilation where the optimized LLVM IR is translated into machine code. Major LLVM libraries involved in doing this are:

***llc*** – It’s a LLVM system compiler. Just like clang takes C/C++ source files as input and generates the IR. llc takes the bit code IR as input and converts it into target assembly code

> $llc llvm-demo.bc -o llvm-demo.s

We can even explicitly specify the target architecture for which to generate the assembly code.

> $llc llvm-demo.bc -march=arm -o llvm-demo-arm.s

After the target specific assembly files are created, the host system assembly can generate the object files from assembly files. Like on most Unix like system 'as' tool is used

> $as llvm-demo.s -o llvm-demo.o

Once the object file is created, LLVM can use the system linker or LLVm linker 'lld' to convert them into executable/library files. lld is a modern linker developed by the LLVM project.

It is designed to be a drop-in replacement for system linkers and offers several advantages:

### **Performance:** lld is significantly faster than traditional linkers like GNU ld. It can handle large programs more efficiently

* **Compatibility:** It supports various formats such as ELF (Unix), PE/COFF (Windows), Mach-O (macOS), and WebAssembly.
* **Features:** lld includes features useful for toolchain developers, such as link-time optimization (LTO) and support for LLVM bit code.
* **Size:** It is smaller and more lightweight compared to traditional linkers

## Understanding Target Triple

In LLVM, a target triple is a string that describes the target architecture, vendor, operating system, and often the ABI (Application Binary Interface) for which the code is being compiled. It provides a standardized way to specify the target for the compiler and linker.

Structure of a Target Triple A typical target triple looks like this:

<arch>-<vendor>-<sys>-<abi>

<arch>: The architecture (e.g., x86\_64, arm, aarch64).

<vendor>: The vendor (e.g., pc, unknown).

<sys>: The operating system (e.g., linux, win32, darwin).

<abi>: The ABI, which is optional and may be omitted or specify details such as gnueabi for GNU EABI on ARM.

Determining the Target Triple to determine the default target triple for your system using Clang, you can run:

> clang -v

> Target: x86\_64-unknown-linux-gnu

Specifying a Target Triple You can specify a target triple when compiling code with Clang:

> clang -target x86\_64-pc-linux-gnu -c llvm-demo.c -o llvm-demo.o

The target triple is used throughout the LLVM toolchain to ensure that the generated code is compatible with the intended architecture and environment. It's a critical part of cross-compilation and ensuring that code runs correctly on different platforms.

## MLIR (Multi-Level Intermediate Representation)

It is a sub-project of the LLVM Compiler Infrastructure project. It aims to build a reusable and extensible compiler infrastructure that can handle multiple levels of abstraction. Here are some key points about MLIR:

### Key Features of MLIR

* **Multi-Level Intermediate Representation:** MLIR allows for multiple levels of intermediate representation, enabling progressive lowering from high-level abstractions to lower-level machine code.
* **Extensibility:** MLIR supports the definition of custom dialects, which can be used to represent domain-specific operations and data structures.
* **Optimization:** MLIR provides a framework for performing optimizations at various levels, including high-level transformations and low-level code generation.
* **Heterogeneous Hardware Support:** MLIR is designed to optimize code for a variety of computing platforms, such as CPUs, GPUs, TPUs, FPGAs, and more.
* **Integration with Existing Compilers:** MLIR aims to connect existing compilers together, reducing software fragmentation and the cost of building domain-specific compilers.

### Example Use Cases

**Machine Learning:** MLIR can represent dataflow graphs, perform graph optimizations, and lower them to target-specific code for accelerators such as GPUs and TPUs.

**High-Performance Computing:** MLIR supports high-performance loop optimizations, such as loop fusion and tiling, which are essential for HPC applications.

**Hardware Synthesis:** MLIR can be used for hardware synthesis tools, enabling the generation of hardware-specific code from high-level descriptions

### Block Diagram

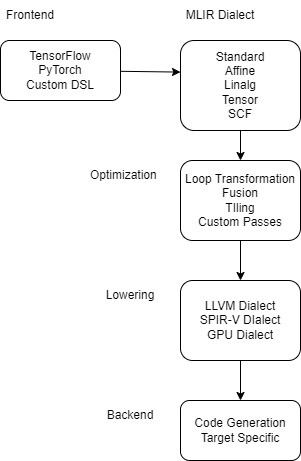


Figure 1.2 MLIR Block Diagram

MLIR is the latest framework which is actively developed.

Here is a glimpse of a use case which explains the power of MLIR.

Imagine you're working on a machine learning compiler. You can represent high-level tensor operations in a custom dialect, perform optimizations specific to machine learning at this high level, then progressively lower these operations through intermediate representations until you finally generate highly optimized machine code. MLIR's greatest strength is its adaptability—it provides a versatile and modular framework that can be customized and extended to meet the needs of a wide range of domains and applications.

Here are some active repositories related to MLIR on GitHub:

**ncnn:** A high-performance neural network inference framework optimized for mobile platforms.

**IREE:** A retargetable MLIR-based machine learning compiler and runtime toolkit.

**CirCT:** LLVM's Circuit IR Compilers and Tools. Torch-MLIR: Provides first-class support from the PyTorch ecosystem to the MLIR ecosystem.

**Concrete:** A TFHE compiler that converts Python programs into fully homomorphic encryption (FHE) equivalent Python.

**BladeDISC:** An end-to-end dynamic shape compiler project for machine learning workloads.

**MegCC:** A lightweight, high-performance deep learning model compiler.

**Heir:** A compiler for homomorphic encryption.

MLIR Extensions by Intel: A collection of MLIR dialects and tools for Intel devices. These repositories showcase various applications and extensions of MLIR across different domains.

A separate chapter is dedicated to MLIR which will go in-depth on concepts and implementations.

## Tools based on LLVM/Clang

LLVM/Clang framework is built using reusable components. As an example, as seen in the initial Frontend chapter, several libs are involved to compile a program from pre-processing to code generation. These libraries can be used independently or along with some other libs to build a required tool.

### clang-check

*clang-check* is a command-line tool that uses Clang's LibTooling framework to perform basic error checking and Abstract Syntax Tree (AST) dumping on C and C++ code. It's particularly useful for developers who want to perform static analysis and syntax checking on their code. It’s a built-in tool in the LLVM framework.

Example:

> $clang-check --analyze clang-check-demo.cpp --

> clang-check-demo.cpp:3:2: warning: Undefined or garbage value returned to caller [core.uninitialized.UndefReturn]

> 3 | return ptr;

> | ^~~~~~~~~~

> $clang-check --ast-print clang-check-demo.cpp --

> TranslationUnitDecl 0x6286a71eb9c8 <<invalid sloc>> <invalid sloc>

> |-TypedefDecl 0x6286a71ec238 <<invalid sloc>> <invalid sloc> implicit \_\_int128\_t '\_\_int128'

> | `-BuiltinType 0x6286a71ebf90 '\_\_int128'

> |-TypedefDecl 0x6286a71ec2a8 <<invalid sloc>> <invalid sloc> implicit \_\_uint128\_t 'unsigned \_\_int128'

> | `-BuiltinType 0x6286a71ebfb0 'unsigned \_\_int128'

> |-TypedefDecl 0x6286a71ec620 <<invalid sloc>> <invalid sloc> implicit \_\_NSConstantString '\_\_NSConstantString\_tag'

> | `-RecordType 0x6286a71ec390 '\_\_NSConstantString\_tag'

> | `-CXXRecord 0x6286a71ec300 '\_\_NSConstantString\_tag'

> |-TypedefDecl 0x6286a71ec6c8 <<invalid sloc>> <invalid sloc> implicit \_\_builtin\_ms\_va\_list 'char \*'

> | `-PointerType 0x6286a71ec680 'char \*'

> | `-BuiltinType 0x6286a71eba70 'char'

> |-TypedefDecl 0x6286a7235220 <<invalid sloc>> <invalid sloc> implicit \_\_builtin\_va\_list '\_\_va\_list\_tag[1]'

> | `-ConstantArrayType 0x6286a72351c0 '\_\_va\_list\_tag[1]' 1

> | `-RecordType 0x6286a71ec7b0 '\_\_va\_list\_tag'

> | `-CXXRecord 0x6286a71ec720 '\_\_va\_list\_tag'

> |-FunctionDecl 0x6286a7235308 </home/hemantab/workspace/sf\_llvm/Chapters/Chapter\_1/clang-check-demo.cpp:1:1, line:4:1> line:1:6 used getPtr 'int \*()'

> | `-CompoundStmt 0x6286a72354d8 <col:15, line:4:1>

> | |-DeclStmt 0x6286a7235478 <line:2:2, col:10>

> | | `-VarDecl 0x6286a7235410 <col:2, col:7> col:7 used ptr 'int \*'

> | `-ReturnStmt 0x6286a72354c8 <line:3:2, col:9>

> | `-ImplicitCastExpr 0x6286a72354b0 <col:9> 'int \*' <LValueToRValue>

> | `-DeclRefExpr 0x6286a7235490 <col:9> 'int \*' lvalue Var 0x6286a7235410 'ptr' 'int \*'

> `-FunctionDecl 0x6286a7235558 <line:7:1, line:10:1> line:7:5 main 'int ()'

> `-CompoundStmt 0x6286a72357d0 <col:12, line:10:1>

> |-DeclStmt 0x6286a7235788 <line:8:2, col:21>

> | `-VarDecl 0x6286a7235648 <col:2, col:20> col:7 ptr 'int \*' cinit

> | `-CallExpr 0x6286a7235768 <col:13, col:20> 'int \*'

> | `-ImplicitCastExpr 0x6286a7235750 <col:13> 'int \*(\*)()' <FunctionToPointerDecay>

> | `-DeclRefExpr 0x6286a72356f8 <col:13> 'int \*()' lvalue Function 0x6286a7235308 'getPtr' 'int \*()'

> `-ReturnStmt 0x6286a72357c0 <line:9:2, col:9>

> `-IntegerLiteral 0x6286a72357a0 <col:9> 'int' 0

When you run this command, you may encounter error, such as:

> Error while trying to load a compilation database:

> Could not auto-detect compilation database for file "clang-check-demo.cpp"

*clang-check* is tool built using libTooling framework. It is not a compiler. It is mainly used for static analysis and refactoring. So to properly parse source files it needs compilation parameters such as includes, macro definitions, and so on. This info is captured by the compilation database (compile\_commands.json). Most of the build systems support dumping compilation databases during the build process. This compilation database can then be used by clang based tools to do code analysis.

By-default clang-check will check for the compilation database to properly compile the file or we can specify the build parameters in the same command at the end after --. Trivial cpp file, like the above example, does not need any extra build info. Hence we just appended -- and specified nothing.

*clang-check* can be used for -

Syntax Checking -- Detects syntax errors in your code.

AST Dumping -- Outputs the Abstract Syntax Tree of the code, which can be helpful for understanding code structure.

Fixit-Hints -- Can apply fix-it hints suggested by Clang to automatically correct certain issues.

IDE Integration -- Suitable for use with IDEs or editors to provide real-time feedback on code quality.

### Populer LLVM based tools

Clang-Tidy -- A linting tool that provides a wide range of checks to improve code quality and catch bugs.

ClangFormat -- A tool for automatic code formatting to ensure consistent coding style.

Clang-Fixit -- Automatically applies fix-it hints to code, helping to correct common errors.

Clang-Plugin -- Allows developers to create plugins that can run additional actions on the Abstract Syntax Tree (AST) during compilation.

Clang-Tooling -- Provides infrastructure for writing standalone tools that need syntactic and semantic information about a program.

LLVM IR -- The intermediate representation used by LLVM, which allows for language-agnostic optimizations and code generation.

LLD -- The LLVM linker, which is used to link object files into executables and libraries.

LLDB -- A debugger that leverages LLVM's infrastructure to provide powerful debugging capabilities.

LLVM-MCA -- A tool for static performance analysis of LLVM IR code.

Sanitizers -- Tools such as AddressSanitizer, MemorySanitizer, and UndefinedBehaviorSanitizer, which help detect various kinds of bugs and vulnerabilities.

Polly -- A loop and data-locality optimizer for LLVM.

MLIR -- Multi-Level Intermediate Representation, which extends LLVM to support multiple levels of abstraction and domain-specific optimizations.

These tools are widely used in the industry to improve code quality, enforce coding standards, and automate various development tasks.

We will touch base upon each tool in the upcoming chapters as and when they are needed.

So to summarize, let us take a closer look at the LLVM architecture diagram. We have introduced the frontend, middle-end and backend libraries. The below diagram shows which library fits into which component.

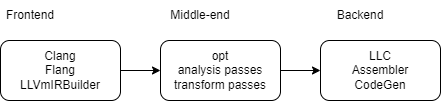


Figure 1.3 LLVM Block Diagram with details

Complete workflow example:

#### Step 1: Compilation to IR (.bc file):

> $clang -emit-llvm -c llvm-demo.c -o llvm-demo.bc

#### Step 2: Optimization

> $opt -O2 llvm-demo.bc -o llvm-demo-opt.bc

#### Step 3: Machine code generation (.s or assembly file)

> $llc fllvm-demo-opt.bc -o llvm-demo-opt.s

#### Step 4: Object file creation

> $as llvm-demo-opt.s -o llvm-demo-opt.o

#### Step 5: Executable / Library creation

> On Linux

> $ld.lld -o llvm-demo llvm-demo-opt.o llvm-demo-1-opt.o

> On Windows

> $lld-link -o llvm-demo.exe llvm-demo-opt.obj llvm-demo-1-opt.obj

Here, Step 1 forms Frontend Step 2 forms Middle-end Step 3,4 & 5 forms Backend

All these steps are orchestrated by the clang driver program *clang*. Which internally invokes all the stages of compilation under the hood. Clang driver is developed to be a drop in replacement for gcc hence all the gcc options are applicable to clang also. However, we can specify extra options to individual component in clang command using -X

Example:

> clang -Xclang -analyzer-checker=core -analyzer-output=text source.c

Here, -Xclang is an actual clang compiler and we are passing options 'analyser-checker=core -analyser-output=text' to the compiler to run all core checkers on source.c file. Similarly, there are following options available to fine tune the compilation process

> -analyzer <arg> Pass <arg> to the static analyzer

> -Xrch\_device <arg> Pass <arg> to the CUDA/HIP device compilation

> -Xarch\_host <arg> Pass <arg> to the CUDA/HIP host compilation

> -Xassembler <arg> Pass <arg> to the assembler

> -Xclang=<arg> Alias for -Xclang

> -Xclang <arg> Pass <arg> to clang -cc1

> -Xcuda-fatbinary <arg> Pass <arg> to fatbinary invocation

> -Xcuda-ptxas <arg> Pass <arg> to the ptxas assembler

> -Xlinker <arg> Pass <arg> to the linker

> -Xoffload-linker<triple> <arg>

> -Xopenmp-target=<triple> <arg>

> -Xopenmp-target <arg> Pass <arg> to the target offloading toolchain.

> -Xpreprocessor <arg> Pass <arg> to the preprocessor

LLVM JIT Compiler/Interpreter

The LLVM lli is a powerful interpreter for LLVM bitcode, allowing developers to execute LLVM bitcode directly without the need to compile it into machine code first. This can be incredibly useful for testing, debugging, and rapid prototyping.

### **Overview of LLI**

LLVM lli is part of the LLVM project and provides an execution environment for LLVM bitcode. It leverages the Just-In-Time (JIT) compilation capabilities of LLVM to translate bitcode into executable machine code on the fly. This ensures that the interpreted code runs efficiently while maintaining the flexibility of an interpreter.

### **Key Features of LLI**

* **JIT Compilation**: lli uses LLVM's JIT compiler to execute bitcode efficiently, translating it into native machine code during runtime.
* **Flexibility**: Developers can run, test, and debug LLVM bitcode without needing to produce a binary executable.
* **Integration**: lli seamlessly integrates with the LLVM toolchain, allowing for easy testing of code generated by other LLVM tools.

### **Using LLI**

To use lli, you need LLVM bitcode, which can be generated using the clang compiler or other LLVM-based tools. Here is a basic workflow for using lli:

We have already generated the out demo bit code file llvm-demo.bc in previous examples, lets see how we can execute it directly using lli

$clang -emit-llvm -c example.c -o example.bc

output:

Welcome to the LLVM universe!!!

LLVM lli is a versatile and powerful tool in the LLVM ecosystem. It provides a convenient way to execute LLVM bitcode directly, making it invaluable for rapid prototyping, debugging, and educational purposes. By leveraging LLVM's JIT compilation capabilities, lli ensures that the executed code is both efficient and flexible.

Conclusion

Now this sets the warmup exercise for LLVM framework understanding. In the later chapters, we will deep dive into the details of each component. Build amazing use cases on LLVM framework. Until now, you did not need to build the LLVM to experiment with these commands. But to build serious tools, default LLVM binaries would not suffice, we have to build LLVM ourselves. Which we will see in the next chapter.

In this chapter, we have used the LLVM pre-build libraries downloaded from the LLVM website. We should be able to use clang comfortably. In the next chapter we will build the LLVM from sources and customize it according to user use cases.