**1 Introduction**

Compilers are some of the most complex software systems, aimed at performing the parsing, transforming and generating executable machine code for a program.

They have two components, a front-end, that includes all the analysis phases, converts the program into an intermediate memory representation; and a back-end, which generates the executable code.

The design and implementation of back-ends which is modular and easily extensible has been important area of research in the last few years.

The Low-Level Virtual Machine (LLVM), a compiler infrastructure is well-suited to the above requirement as it is written in a way that allows the reusability of classes as often as possible.

It also allows components to be shared across different compilers and hence one compiler gets benefited by the improvements made to other.

**1.1** **Goal of Thesis**

In this thesis I have implemented a new backend based on the LLVM framework that generates assembly code for the REISC architecture.

ReISC (Reduced energy Instruction Set Computer) is an 32-bit architecture with support for 8/16/20/32 data size with variable instruction length.

Having support for secure data, fast interrupt response and parallel operations, Reisc architecture targets at the next generation Ultra Low Power and High Performance applications, such as digital signal processing.

**1.2 Organization of this Thesis**

**2 The LLVM Compiler Infrastructure**

The Low Level Virtual Machine (LLVM) is a compiler framework that was started in 2000 in the University of Illinois by Chris Arthur Lattner. This compiler infrastructure eases out the process of building compilers and is designed for static as well as dynamic compilation.

It is a set of libraries which is independent of both language and target. This type of representation helps to apply common techniques at each stage of compilation. The LLVM representation is expressive and extensible on one hand and low-level on the other hand.

The features that make LLVM stand out from other compilers are its internal architecture, simplicity, understandability, extensibility, stability, reliability and tools like Clang .Some other features supported by LLVM are efficient tail calls, garbage collection, zero-cost exception handling, link-time optimization etc. .All the compilers that are being developed by utilizing this framework get the benefit of all these features for free.

**Comparison to Traditional Compilers:**

**The Three Phase Design:**

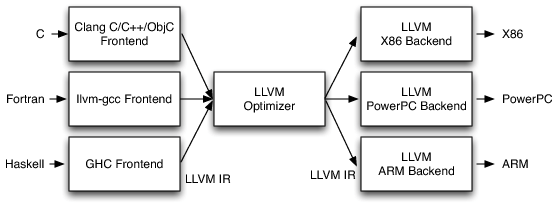
LLVM has a three phase design comprised of front end, optimizer and backend.

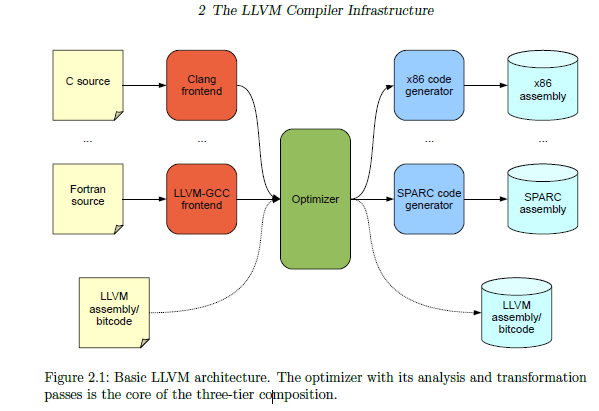


Front end is responsible for parsing and analyzing the source code, transforming the parsed code into an AST and then AST LLVM intermediate representation.

Optimization is an optional phase that performs analysis and optimization on the IR thus improving the code. Optimizer is language and target independent.

The output from the optimizer is then fed as input to the backend also known as code generator that converts the IR to target machine code.





**Key LLVM Design Features:**

**LLVM’s Intermediate Representation:**

LLVM is a Static Single Assignment (SSA) based representation that provides type safety, low-level operations, flexibility, and the capability of representing ‘all’ high-level languages cleanly. It is the common code representation used throughout all phases of the LLVM compilation strategy.

**Type System:**

**Frontends:**

**Retargetable Code Generator:**

### *LLVM Target Description Files*

**Optimization**

**Summary**