**1 Introduction**

In today’s complex environment, it is very crucial for the devices to feature low power consumption and that to at a low cost. It is said that Internet of Things will bring an era where everything will have chips embedded in them, be it a household appliance, mobile device or industrial equipment. For this vision to get fulfilled, today’s power-needy devices need to be replaced with devices powered by chips that operate on low levels of power. Having this property, Ultra Low Power chip design is in great demand. These chips are at the core of the devices that make up the Internet of things.

Chips can be coded using machine language only but it is extremely difficult for the programmers to write machine language programs. In order to make this task easier for the programmers, chips are programmed using high level programming languages.

But how will the chip function when it does not understand programming languages?

Compilers are a solution to this problem. It takes as input the human readable code written by the programmer and performs the complex task of translating it into machine readable code which can then be understood and executed by the chip.

Chips vary in their architectures and hence accept different versions of machine code. If we want to program a new chip i.e. we want to generate the executable code for a program for a different architecture, the compiler must be redesigned according to the new architecture and the instruction set of that architecture.

* 1. **Goal of Thesis**

The goal of this thesis was to implement a new backend that generates assembly code for the ReISC architecture.

ReISC (Reduced energy Instruction Set Computer) is an embedded architecture meant for low power devices and high performance applications. It has support for secure data, parallel operations and fast interrupt response.

To leverage these features of the architecture, building a compiler is essential.

Ideally, compiler should be completely customized for each target, but on the other hand, they share a lot of commonality and perform very similar tasks. For example, values need to be assigned to registers in each architecture, so the algorithms should be shared wherever possible. There is need of utilizing these common features and writing things specific to an architecture only.

To avoid writing entire compiler i.e. both frontend and backend the high level language frontends already available should be used and backend part should be written.

Writing the backend should involve as low effort as possible. Information redundancy should be minimized and it should be modular, reusable, maintainable and easily extensible.

* 1. **Challenges:**
  2. **Approach**

The GNU Compiler Collection supports a large number of frontends and backends but extending and retargeting it is a very complex task due to its coherent design. Reusability of pieces is not possible and amount of sharing across different compilers is very little.

My approach was to choose LLVM(Low-Level Virtual Machine) since it overcomes these limitations.

LLVM supports the feature of pluggable frontends. It provides the flexibility to write backend only by allowing using already present frontends.

It is written as a set of libraries and hence allows the reusability of classes and sharing of components across different compilers as often as possible. It automates a lot of things in the backend by writing target descriptions in a single location called .td files. Based on this description, plenty of code can be generated by tablegen (An LLVM tool used to generate C++ code) which takes as input .td files and generates .inc files that can be included in other LLVM source files. For ex, instruction set of architecture is described in Instrinfo.td and then TableGen processes this file to generate the “instruction selection algorithm”, which would have been very difficult if written manually.

The LLVM is extremely modular, easily extensible, understandable and reliable.

These remarkable features have been the motivation for developing the backend using LLVM framework.

**1.2 Contributions**

As part of this research, I have implemented the backend for ReISC architecture based on the LLVM framework.

The first main of this thesis is to implement the basic instruction set of ReISC.

The second contribution of this thesis is to generate the machine specific assembly code similar to that generated by GCC.

**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:**

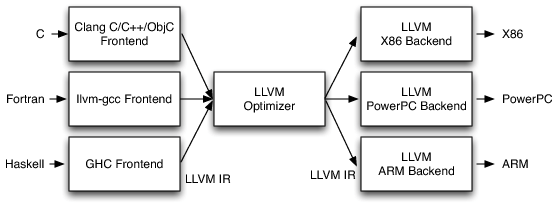
**The Three Phase Design & its Implications:**

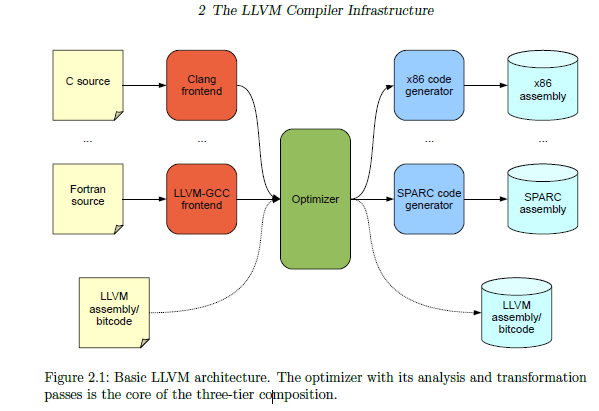
LLVM has a three phase design comprised of front end, optimizer and backend.



Front end is responsible for parsing and analyzing the source code, transforming the parsed code into an AST . AST being language and frontend dependent, is then translated to compiler's generic representation known as 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.

In the compiler back end implementation, areas that involve a lot of effort to be put by language designers are instruction selection, register allocation, and instruction scheduling.





This design has the edge over traditional compilers when there is a need to support a new source language or architecture. Had we been using the traditional compiler design, it would require a whole new compiler to be developed from scratch for each language or architecture.

With this, to support a new source language, only front end part of the compiler needs to be developed, while already existing optimizer and backend for a particular architecture can be reused.

**Key LLVM Design Features:**

**LLVM’s Intermediate Representation:**

Intermediate Representation (IR) is a way of representing the code by LLVM.It is a Static Single Assignment (SSA) based universal representation used in all phases of the LLVM compilation strategy.It provides the flexibility of representing high-level languages in a clean and simple manner.

LLVM IR supports an unlimited number of registers and can be represented in three different forms which are all equivalent : as text which is a human readable form of IR, as bitcode format and as an in memory representation.Files in the LLVM IR are known as modules which consists of:meta-data,global and local variable definitions & function definitions.

**Meta-data may include some sort of special information, provide possibility to attach arbitrary data to the code without a need of changing program behaviour**. Global variables are preceeded by @ whereas local ones are preceeded by % symbol.Labels with a set of instructions in each of them (collectively called a basic block)constitute the function definition with the restriction that the last instruction of every label should either be return instruction or branch instruction.A specific basic block known as the entry block is the place from where the execution of the fuction is started. Functions consists of instructions, which take value type and variable as arguments.

Each block may begin with a sequence of phi instructions that merge incoming values from the block’s predecessors.

The terminator unreachable is a special instruction used to specify that there is function call without a return instruction.

Some features of IR are:

• Low level virtual instruction set;

-extensibility and effectiveness of high-level languages.

• Representation based on Static Single Assignment (SSA)

- Supports instructions like addition, subtraction and branch operations

-Instructions are represented using three adress format

-language independent and target-independent

• Has support for labels

Static Single Assignment (SSA) format is the generic code representation used by LLVM.There should be a single definition of each variable to satisfy the validity condition of SSA form i.e. it is invalid for a variable to be present in two control flow paths.φ-function is used to overcome this issue by returning the value corresponding to the control-flow path being taken.

if (condition) then a := 0 else a := 1

return a

Here the variable a is present in two control-flow paths.

SSA representation of the above example:

if (condition) then a1 := 0 else a2 := 1

a := φ(a1, a2)

return a

If the condition evaluates to true, control flow will take the branch for true and the value returned by the function φ(a1, a2) will be a1.On the other hand,if the condition evaluates to false ,control flow will take the branch for false and the value returned by the function φ(a1, a2) will be a2.

Example:

int foo(int a)

{int c=1,i;

for(i=0;i<=a;i++)

c=c+3;

return c;

}

a function written in the textual LLVM IR is shown in Listing 2.3. This demonstrates the explicit nature of control flow and typing in LLVM IR. Every instruction contains type information for its arguments, and when necessary the result type.

Using LLVM-GCC compiler we can produce LLVM IR from the C code and in Listing3.2 is the result that is already in SSA formThe first line starts with a definition of the function (instruction define). The definition has to include return type (i32 ), name (has to start with @) and entry parameters (pairs of type and name starting with

% ). Inside the body of the function there are 3 labels defined (no indent and ends with :). Entry label is the entry point of the function and will be executed by function call automatically. Entry point has a simple check inside, to make sure that other label code is necessary or if it is better to execute return label and finish function execution. The check is done with the br instruction by controlling the value of %tmp919 and deciding which label to go to .

define i32 @fac(i32 %digit) #0 {

%1 = alloca i32, align 4

%answer = alloca i32, align 4

%i = alloca i32, align 4

store i32 %digit, i32\* %1, align 4

store i32 1, i32\* %answer, align 4

store i32 2, i32\* %i, align 4

br label %2

; <label>:2 ; preds = %10, %0

%3 = load i32\* %i, align 4

%4 = load i32\* %1, align 4

%5 = icmp sle i32 %3, %4

br i1 %5, label %6, label %13

; <label>:6 ; preds = %2

%7 = load i32\* %answer, align 4

%8 = load i32\* %i, align 4

%9 = mul nsw i32 %7, %8

store i32 %9, i32\* %answer, align 4

br label %10

; <label>:10 ; preds = %6

%11 = load i32\* %i, align 4

%12 = add nsw i32 %11, 1

store i32 %12, i32\* %i, align 4

br label %2

; <label>:13 ; preds = %2

%14 = load i32\* %answer, align 4

ret i32 %14

}

Next, look at the body of the loop label, which implements the execution of the for cycle. In the end of that instruction is br (short for ’branch’) which checks if it is time to move into return label, where the final statement is returned out of the function with the ret instruction.

The intermediate representation of a compiler is interesting because it can be a "perfect world" for the compiler optimizer: unlike the front end and back end of the compiler, the optimizer isn't constrained by either a specific source language or a specific target machine. On the other hand, it has to serve both well: it has to be designed to be easy for a front end to generate and be expressive enough to allow important optimizations to be performed for real targets.

**Type System:**

**Frontends:**

**Retargetable Code Generator:**

### *LLVM Target Description Files*

**Optimization**

**Summary**