1 INTRODUCTION

In today's complex environment, it is very crucial for the devices to feature low power consumption and that to at low costs. 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.

1.1 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.

1.2 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.

1.3 Contributions

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

1.4 Organization of this Thesis

This thesis report is organized as follows:
Chapters 2 of this thesis give an overview of LLVM and the ReISC architecture.
Chapter 3 contains the description of design of the compiler backend.
Chapter 4 describes the step by step process of creating backend for ReISC.
Chapter 5 highlights the results generated by the LLVM backend and comparison with gcc results.
Chapter 6 discusses about the conclusion and future work.

2 THE LLVM COMPILER AND REISC ARCHITECTURE

2.1 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.

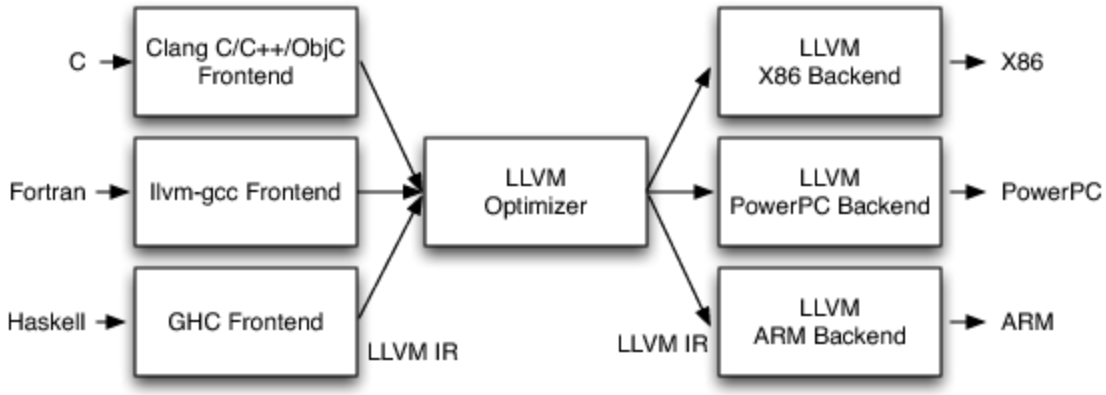
The features that make LLVM stand out from other compilers are its internal architecture, simplicity, understandability, extensibility, stability, reliability and tools like Clang .Some other features supported by LLVM are efficient tail calls, garbage collection, zero-cost exception handling, link-time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

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.



The LLVM Compiler Infrastructure

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.

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 bit code 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 behavior. Global variables are preceded by @ whereas local ones are preceded 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 function is started. Functions consist 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 instruction is 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

• 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.

The intermediate representation is ideal for the compiler optimizer since on one hand it is both language and target independent and on the other hand, it has to designed such that the front end can easily generate code for it as well as is expressive enough to allow important optimizations to be performed.

Type System:

Unlike most RISC instruction sets, LLVM follows strict type representation with a simple type system (e.g., i32 is a 32-bit integer, i32** is a pointer to pointer to 32-bit integer). Every virtual register and memory location has a specified type. On the other hand, LLVM IR is a low-level, expressive and extensible language.

According to LLVM type representation; there is an associated type for each memory location and SSA value as well as type rules for all operations. The high level type system is the unique feature of LLVM, which helps LLVM optimizer and compiler in producing optimal code and performing high-level transformation on low-level code. It eliminates the need to perform extra analyses before the transformation.

In addition, errors in optimizations can be detected if there is a type mismatch. In LLVM instructions, there are some restrictions on the operands to preserve type correctness.

For example, both operands of the add instruction should be of arithmetic (i.e., integral or floating-point) type, and result is also a value of same type.

The type system used in LLVM falls in two categories: the primitive types and the derived types.

Primitive types: The primitive types are the fundamental building blocks of the LLVM system. They are independent of the source language.

TYPE	SYNTAX	OVERVIEW
Void	Void	does not represent any value and has no size
Integer	iN where N is the number of bits integer will occupy	specifies an arbitrary bit width for the integer type desired
Label	label	represents code labels
Float	float	32-bit floating point value

Derived types: The derived types are complex types that are made up of primitive types and other derived types. They provide the ability to represent arrays, vectors, pointers and functions and are independent of the source language.

TYPE	SYNTAX	OVERVIEW
Function	`i32 (i32)`	function taking an `i32`, returning an `i32`
Pointer	`<type> *`	used to specify memory locations
Array	[<# elements> x <elementtype>]	The number of elements is a constant integer value; `element type` may be any type with a size.
Structure	{ [type], [type] }	represent a collection of data members together in memory

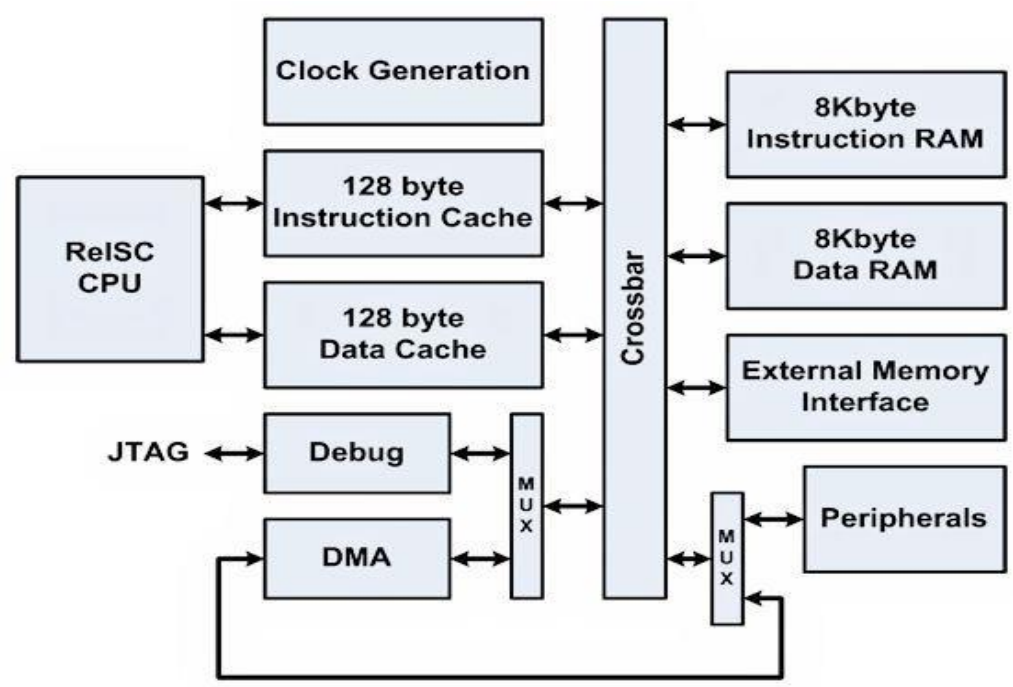
Code Generation:

LLVM is designed as a set of reusable libraries which facilitates sharing of components across different compilers. Its modular design eases out the code generation process to a great extent. The complete code generation framework is described in detail in Chapter 3.

2.2 The ReISC Architecture

ReISC (Reduced Energy Instruction Set Computer) is a 32-bit architecture developed by STMicroelectronics. It supports variable-length instructions (16, 32, or 48 bits), variable data size (8/16/20/32 bits), data security, quick interrupt response and parallel operations. Variable-length instructions are the main feature of ReISC instructions that helps to achieve high code density.

ReISC targets the next generation Ultra Low Power devices and High Performance applications, such as digital signal processing, media processing, biomedical devices, wireless sensors etc.



ReISC core has an enhanced RISC architecture with a 3- stages pipeline. It supports concurrent instruction fetch and memory access. The instruction and data cache both are present in this processor architecture, however these plug-ins are optional. Around 50% of the total energy utilization is consumed by the memory subsystem, so focusing on saving energy in memory access cycles can help in saving substantial amount of energy.

In case of ReISC, the small 128 bits wide cache act as an interface to the SoC's on-chip instruction and 128 bits wide data SRAMs, hence minimizing the energy required for accessing memory. The caches are not faster than the local on-chip memories and don't contribute in performance improvement, but in case of a cache hit, memory cycle need not to be run on the larger memory arrays, which saves energy.

The register file and the ALU are optimized to work with different data sizes with a granularity of a single instruction. Long and small integers, pointers, and char data types can live together in the pipeline and in the register file, saving power while keeping low-end 32bit processors

performance. Short relative jumps are supported at no code-space expense, while optimized Long Branch instruction can jump directly into the whole address space. The ReISC roadmap includes multi-core SoCs and many DSP extensions to the instruction set

Consuming a bare minimum amount of power, this type of technology will certainly enable many new implantable devices that must operate at extremely low powers and squeeze every bit of juice out of their batteries or energy-harvesting means.

Addressing Modes

ReISC Core belongs to memory/register architecture, rather than the prevailing register/register architecture adopted by most of the commercial RISC microprocessors. This allows reducing the energy for inter-instruction data transfer, and to obtain a more compact instruction size.

The ReISC architecture supports multiple addressing modes, including the followings:

- Immediate addressing mode
- Register addressing mode
- Absolute addressing mode
- Displacement addressing mode
- Indirect addressing mode
- Auto-increment addressing mode
- Self-update addressing mode
- Index addressing mode

Instruction Set

In ReISC, instructions are encoded either as 48 bit, 32-bit or 16-bit words. Instructions follow two-address format, i.e., src1 (dest) ← src1 OP src2, where one of the source operands works as destination as well. The destination operand is always a register. Source operands can either be registers or immediate values embedded in the instruction word.

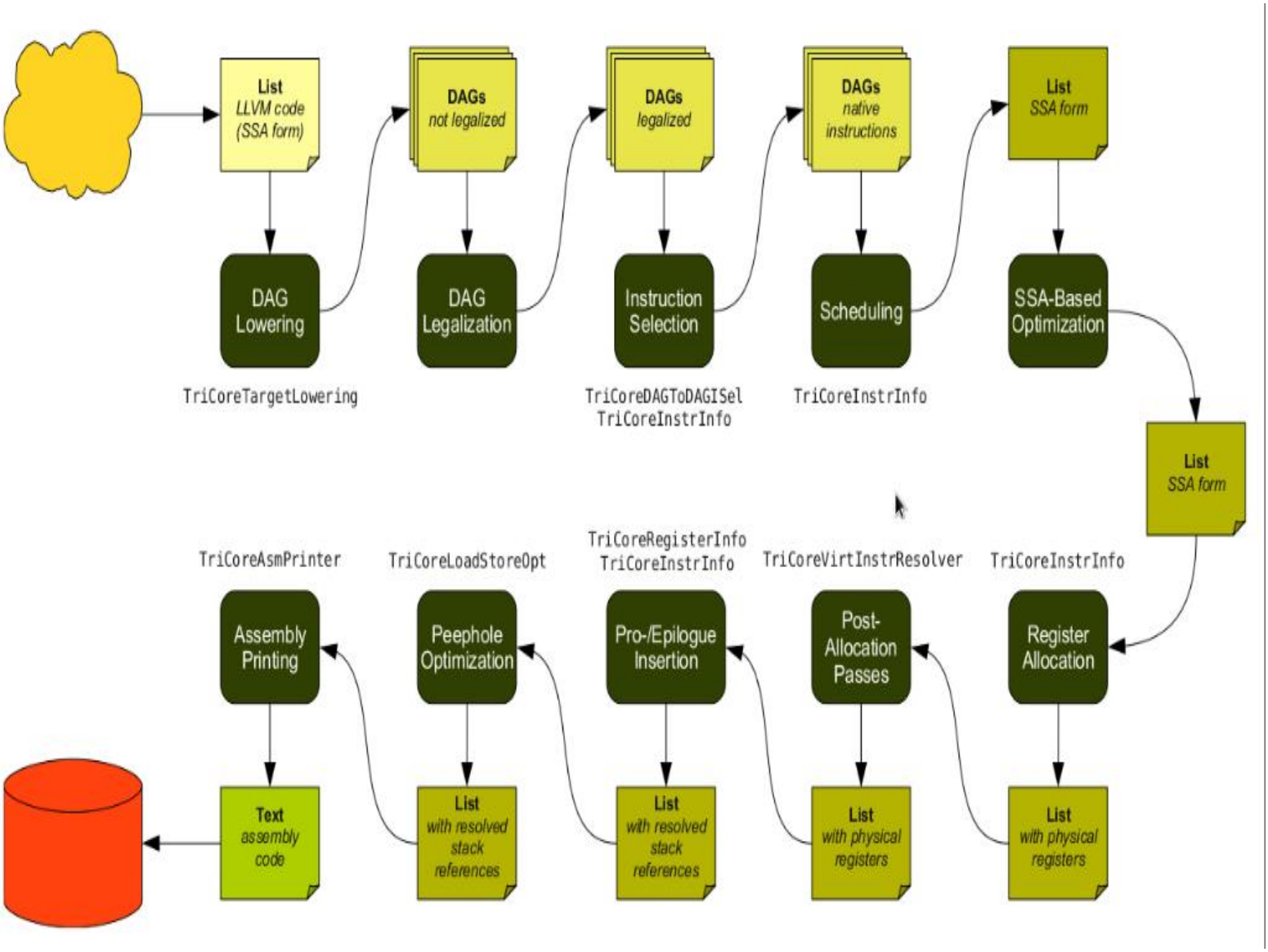
Immediate operands take 4, 16 or 32 bits of the instruction word depending on the instruction word size and immediate operand value. Immediate operand is limited to 4 bits in 16-bit instructions, 16 bits in 32-bit instructions and 32 bits in 48-bit instructions.

ReISC do not have a condition code flag register. Typically architectures having flag register follow the approach of setting its value either implicitly by any arithmetic operation or by issuing an explicit comparison instruction. Contents of this register are then assessed by the conditional branch instructions.

ReISC's approach is quite different from this. It merges compare and branch instructions into a single instruction, ex. JPDEQ (jump if equal).

3 DESIGN OF THE COMPILER BACKEND

The backend is comprised of various passes that analyze and transform the LLVM intermediate representation (IR) into assembly code. There are several steps involved in transforming the LLVM IR into target assembly code. The IR changes after each pass and gets more similar to the target instructions. The following diagram illustrates the steps of transformation LLVM IR to assembly code.



1 Instruction Selection

The transformation of the LLVM code transformed into a set of Selection DAGs is the first step of the code generation process. Selection DAG is a directed acyclic graph representation where nodes denote instructions and the edges denote definition–use relationship among them. There are separate DAGs for each basic block. Additional "chain" edges are inserted for control flow dependencies.

After construction of selection DAGs, they are legalized so that it contains operations and data types supported by the target architecture only. The DAG after legalization is passed to the instruction selector, which transforms it into a new DAG with nodes denoting actual target instructions.

By the end of this phase, the DAG has all of its LLVM IR nodes converted to target-machine nodes, that is, nodes that represent machine instructions rather than LLVM instructions.

2 Scheduling and Formation

The instructions of the legalized DAGs are then converted into a list format.

Being in SSA form, the instructions in list operate on an infinite number of virtual registers and are not completely valid assembly code.

The scheduler is required to determine the order in which the instructions are emitted. To determine instruction ordering inside basic blocks, a three-address representation is required. Preregister Allocation(RA) Scheduling, the first instance of Instruction Scheduling, performs this task of ordering the instructions which are then transformed to the three-address representation.

3 SSA-based Machine Code Optimizations

Some target-specific low-level optimizations may be performed by the code generator before physical registers are allocated.

4 Register Allocation

In this phase, virtual registers are eliminated by mapping them to physical registers.

Spill code is produced if the number of available physical registers is less than those of live virtual registers.

5 Prologue/Epilogue Code Insertion

After the mapping of registers have been performed, prologue and epilogue code can be emitted.

Prologue performs the reservation of stack frame by decrementing the stack pointer.

Epilogue destroys the reserved stack frame and restores all saved registers before returning from a function.

6 Late Machine Code Optimizations

This phase includes peephole optimizations that are performed on the final target code .

7 Code Emission

This is the final phase of the transformation process which performs the emission of completed assembly code.

TableGen

TableGen is the descriptive language used by LLVM to describe several machine aspects used in several compiler stages. Had this concept not been there, the programmer would have to write code that reflects the same target characteristics in different files. It causes information redundancy in the code despite the extra effort. So if there is a change in any one of the aspects, programmer would need to change various parts of the code.

The TableGen concept has reduced this complexity of writing and maintaining the backend code to a great extent. It can describe complex entities effectively although it has a very simple syntax. TableGen is a declarative programming language used to describe files that act as a central repository of target specific information. The approach is to describe machine aspects in a single location, for example, the machine register description in <Target>RegisterInfo.td which are then processed by the TableGen tool with a specific goal, for example, generate the pattern-matching instruction selection algorithm.

TableGen descriptions are stored in .td files. For every TableGen backend, certain top-level superclasses have special pre-defined semantics (e. g., the Register class for the register description). The declaration of these classes is present in the file include/llvm/Target/Target.td. The TableGen generates a C++ file as output which can then be included and compiled along with the regular code base. This process is integrated transparently with LLVM's build system.

Here is an overview of TableGen files used for code generation:

ReiscOther.td:

The <Target>.td file (for example, X86.td) defines features supported by ISA features and processor families. For example, Reisc.td defines the compare feature as:

```
def FeatureCmp        : SubtargetFeature<"cmp", "HasCmp", "true",
                                         "Enable 'cmp' instructions.">;
```

A processor type with the features it supports can be defined as:

```
def : Proc<"reisc32",  [FeatureReisc32]>;
```

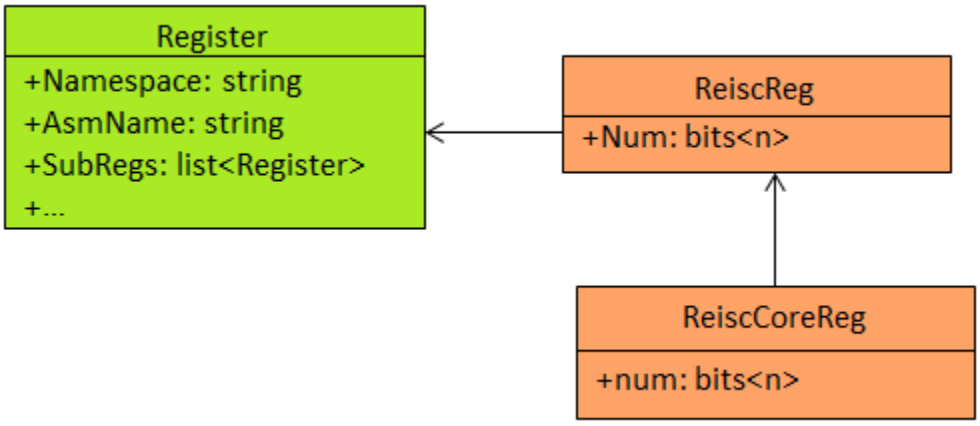
The <Target>.td file is the top level file for target-specific information and includes all other .td files. The Reisc.td file has part of the information that TableGen uses to generate the X86GenSubtargetInfo.inc file, but it is not limited to it. In fact, there is no one to one mapping between a .td file and an .inc file. So, TableGen always compiles all .td files.

Register Description Table (ReiscRegisterInfo.td)

The ReiscRegisterInfo.td file contains the definition of registers and register classes. There are two subclasses in ReISC backend, ReiscReg derived from Register class and ReiscCoreReg derived from ReiscReg class declared as follows:

```
class ReiscReg<string n> : Register<n> {
  field bits<x> Num;
  let Namespace = "Reisc";
}
class ReiscCoreReg<bits<x> num, string n> : ReiscReg<n> {
  let Num = num;
}
```

Where x denotes the number of bits required to represent registers in ReISC architecture.



Register Type Hierarchy

Register definitions are then written as follows:

```
def  X   : ReiscCoreReg<0,  "X">,    DwarfRegNum<[0]>;
```

where X denotes any register of ReISC architecture.

ReiscGenRegisterInfo.inc is generated by the tablegen tool using this .td file which contains the enumeration of ReISC architecture.

Instruction Formats (ReiscInstrFormats.td)

ReiscInstrFormats.td contains the description of format of instructions. Directly or indirectly, instruction format is a subclass of the Instruction class and contains the following fields:

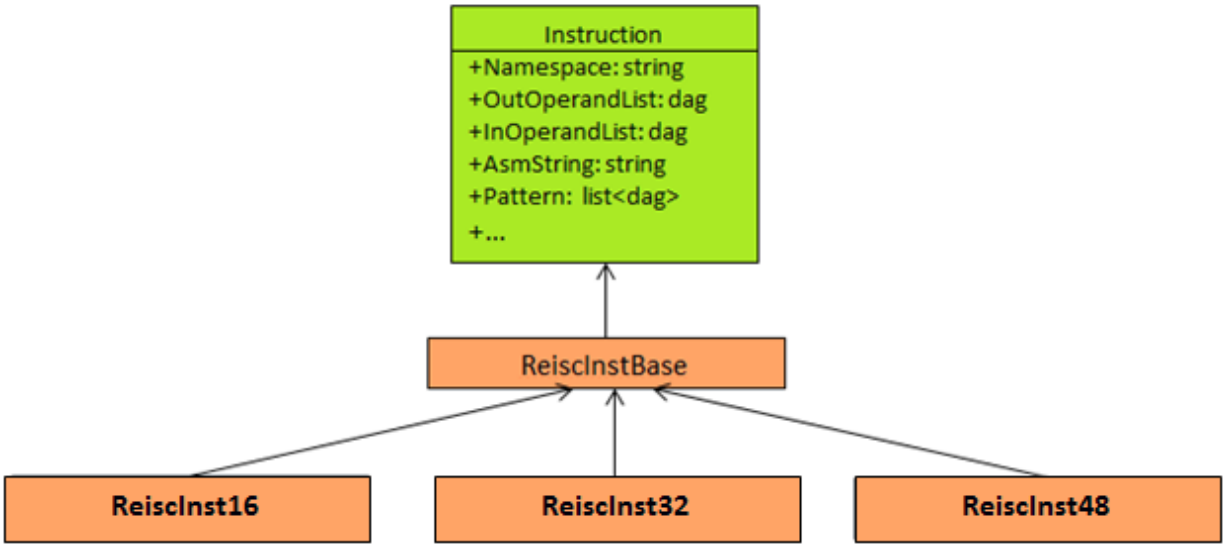
Output operands: This contains the value that represent the outcome of an instruction.

Input operands: This contains all SDValues used by the instruction as input arguments.

Assembly string: This field represents the string that gets printed when an instruction is being emitted.

DAG pattern: During instruction selection this pattern DAG nodes is used to perform pattern matching and replaces the nodes with corresponding target-specific instructions on a successful match.

The ReISC backend has different Instruction subclasses for instructions of different size and further subclasses for each instruction format.



Instruction Type Hierarchy

Instruction Description Table (ReiscInstrInfo.td)

ReiscInstrInfo.td contains the description and definition of instructions of the architecture. The TableGen tool processes this definition which is then used in the instruction selection phase of code generation.

Calling Convention (ReiscCallingConvention.td)

<Target>CallingConv.td contains the information of how arguments are passed to and return values are received from functions. All the rules of ReISC's ABI are specified in ReiscCallingConv.td through conditions such as CCIfType and actions such as CCAssignToReg, CCAssignToStack etc.
ReiscGenCallingCovn.inc is generated when ReiscCallingConv.td is processed by TableGen tool. ReiscGenCallingCovn.inc file contains functions with nested if-statements generated for each definition of ReiscCallingConv.td file. Whenever a function argument or return value is lowered, the relevant calling convention function is invoked and determines a location for the variable.

4 CREATING ReISC BACKEND

In this chapter of thesis work, the step by step process of ReISC backend creation has been described.

4.1 ReISC backend machine ID and relocation records

To create a new backend, some files in llvm root directory need to be modified to include information like the name and ID of target machine and relocation records. The following files are modified to add ReISCbackend:
/llvm/config-ix.cmake
llvm/CMakeLists.txt
/llvm/include/llvm/ADT/Triple.h
llvm/include/llvm/MC/MCExpr.h
/llvm/include/llvm/Object/ELFObjectFile.h
/llvm/include/llvm/Support/ELF.h
/llvm/lib/MC/MCELFStreamer.cpp
/llvm/lib/MC/MCExpr.cpp
/llvm/lib/object/ELF.cpp
/llvm/include/llvm/Support/ELFRelocs/Reisc.def
/llvm/lib/Support/Triple.cpp

4.2 Creating the Initial ReISC .td Files

As discussed in the previous section, LLVM use target description files (.td files) to describe several machine aspects like register set, instruction set, and calling conventions. On compilation, the tablegen tool generates a C++ file with .inc extension using these .td files which are included in other backend files.

4.3 Target Registration

The target needs to be registered with the TargetRegistry, which enables other LLVM tools to lookup and use the target at runtime. A global Target object which represents the target during registration should be declared for each target. Then, in the TargetInfo library, the target should define that object and use the RegisterTarget template to register the target.
For example, the file TargetInfo/ReiscTargetInfo.cpp register TheReiscelTarget for little endian, as follows.

```
extern "C" void LLVMInitializeReiscTargetInfo() {

  RegisterTarget<Triple::reisc,

        /*HasJIT=*/true> X(TheReiscTarget, "reisc", "Reisc");
```

Files
Reisc.h
ReiscTargetMachine.h
ReiscTargetMachine.cpp
TargetInfo/ReiscTargetInfo.cpp
MCTargetDesc/ReiscMCTargetDesc.h
MCTargetDesc/ReiscMCTargetDesc.cpp

By the end of this step, the Target Registration for Reisc backend is done, hence it can be recognized by the backend compiler command llc.

4.4 Defining Target Machine Structure

Every LLVM target must offer a specified interface through which it can be accessed by the higher-level components. It has to implement a subclass of TargetMachine, provide information about existing subtargets, and register itself with the code generator.
The class ReiscTargetMachine provide methods to enable accesses to the target-specific information such as the register files, instructions, frames, etc. via the get**Info methods (getInstrInfo, getRegisterInfo, getFrameInfo, etc.)

Functions: RegisterTargetMachine() is used to register a target-specific machine.

For reisc : **RegisterTargetMachine**<**ReiscelTargetMachine**> **T**(**TheReiscTarget**);

Some other files used for this purpose are:
ReiscSubTarrget.h &ReiscSubTarrget.cpp: To support the several variations of a given chip set, LLVM has the concept of subtargets. Subtargets implement a common target architecture, but have different features, which are represented either as boolean variables or as enumerations. These files are used to inform the code generation process about available variations of the chip i.e the processor information and ABI informormation.
Currently the Reiscbackend supports only one subtarget Reisc32.
ReiscFrameLowering.h &ReiscFrameLowering.cpp: These files contain frame lowering information such as direction and amount of growth of stack.

ReiscISelLowering.h & ReiscISelLowering.cpp :contains code for converting a DAG to use types and operations that are natively supported by the target and converting unsupported types and operations to supported ones by using Promote, Expand, Custom, or Legal options.

4.5 Adding ReiscDAGtoDAGISel Class (Instruction Selection)

ReiscDAGToDAGISel class is a subclass of SelectionDAGISel class which performs instruction selection by implementing the Select() method. An SDNode parameter is the input to this method and the output is an SDNode value that represents a physical instruction. The Select() method uses the code generated from TableGen patterns to perform this matching of physical instructions and if no match is found, an error occurs. The match is performed as follows:

TableGen generates the SelectCode() method and the MatcherTable using the instruction definitions in ReiscInstrInfo.td file. This method and table are present in the ReiscGenDAGISel.inc file in the build directory. The MatcherTable contains the mapping of ISD and ReiscISD nodes to physical-instruction nodes.

The SelectCode() method is called by Select() method which in turn calls SelectCodeCommon() method. The SelectCodeCommon() uses the target matcher table and performs the matching of nodes.

4.6 Adding ASM Printer

The working of assembly printer is as follows:

AsmPrinter is a machine function pass that invokes the runOnMachineFunction()
method for each function and emits the function header first and then basic blocks are processed, where one MI instruction is sent at a time to the EmitInstruction() method for further processing. This method is overloaded by the AsmPrinter subclass of the target (ReiscAsmPrinter.cpp in our case).

EmitInstruction() method converts MI instruction into an MCInst instance through the MCInstLowering interface which is subclassed by each target (ReiscMCInstLowering.cpp in our case).

The MCStreamer class processes MCInst instructions and prints assembly language via MCAsmStreamer subclass.

The printInstruction() method has to be called for every machine instruction and
prints out the corresponding assembly string specified in the TableGen file.

All operands with a default type are handled by printOperand() – custom operands (e. g., immediates and complex addresses) are dealt with by the respective appointed
print methods.

Files

ReiscMCAsmInfo.h
ReiscMCAsmInfo.cpp
ReiscAsmPrinter.h
AsmPrinter/ReiscAsmPrinter.cpp
ReiscInstPrinter.cpp
ReiscMCInstLowering.h
ReiscMCInstLowering.cpp

At this point we have a frame work for ReISC target machine which supports a few instructions like load, store and is able to compile llvm intermediate code into ReISC assembly code.

4.7 Adding Remaining Instructions and Control Flow statements

The ReiscInstrInfo.td file is modified in this step to support remaining instructions of the base instruction set of architecture like arithmetic, logic and shift instructions.
ReiscInstrInfo.td and ReiscISelLowering.cpp file is modified to support the control flow statements, like "while" ,"if else" and "for" loop statements. These files contain methods that transforms the control flow statements of llvm IR into Reisc instructions.

4.8 Adding Function Call Support

Passing arguments is a very important aspect of function call. Arguments can be passed either all in stack or in the registers reserved for function arguments and then in stack if the number of arguments is more than the number of reserved registers.
These rules are specified in REISC ABI and are added in **ReiscISelLowering.cpp** file in function static bool CC_ReiscGccABI(...).
The ReiscTargetLowering class performs the actual lowering and is called if:
1. **The scope of a function is entered**: LowerFormalArguments() method is called which determines the location of each formal argument, moves it into virtual registers and and inserts a sequence of moves and/or loads into the DAG.

2. **The scope of a function is left**: The LowerReturn() method copies the return value in the corresponding physical register and during the instruction selection phase,matching with the return instruction is performed.

3. **A function is called**: The LowerCall() method processes the function call, copies the actual arguments before calling the function and return value into the virtual register while returning from the function.

Files:

ReiscISelLowering.h
ReiscISelLowering.cpp
ReiscInstrInfo.td
ReiscMCInstLower.cpp

5 RESULTS

In this chapter of thesis work ,we are generating the assembly code using the LLVM backend developed and gcc .Assembly code generated by both the methods is then compared to find out the amount of similarity between them.

Test Case 1

```
int main()
{
  return 0;
 }
```

Reisc Toolchain Output with -O3 :

```
00000000 <main>:
   0:   07 af       movw            %r0,    #0
   2:   0e ed               jpra            %r14
```

LLVM output with -O3 :

```
main:
        .frame  %sp,0,%r14
        .mask   0x00000000,0
        .set    noreorder
        .set    nomacro
# BB#0:
        movw    %r0, #0
        jpra    %r14
        .set    macro
        .set    reorder
        .end    main
$tmp0:
        .size   main, ($tmp0)-main
```

Test Case 2

```
int foo()
{
 return 4;
 }
```

Toolchain Output with -O3:

```
00000000 <foo>:
  0:    04 af       movw            %r0,    #4
  2:    0e ed               jpra            %r14
```

LLVM output with -O3 :

```
foo:
        .frame  %sp,0,%r14
```

```
        .mask   0x00000000,0
        .set    noreorder
        .set    nomacro
    # BB#0:
        movw    %r0, #4
        jpra    %r14
        .set    macro
        .set    reorder
        .end    foo
    $tmp0:
        .size   foo, ($tmp0)-foo
```

Test Case 3

```
int foo()
{
 int a=5;
 return a; }
```

Toolchain Output with -O3:

```
    00000000 <foo>:
  0:    08 af 05 00         movw        %r0,    #0x5
  4:    00 00
  6:    0e ed               jpra        %r14
```

LLVM output with -O3 :

```
     foo:
        .frame  %sp,0,%r14
        .mask   0x00000000,0
        .set    noreorder
        .set    nomacro
     # BB#0:
        movw    %r0, #5
        jpra    %r14
        .set    macro
        .set    reorder
        .end    foo
    $tmp0:
        .size   foo, ($tmp0)-foo
```

Test Case 4

```
int main()
{
  int a = 5;
  int b = 2
    int c = a + b;
     int d = b + 1;
```

```
            return (c+d);
          }
```

Reisc Toolchain Output with -O3 :

```
          00000000 <main>:
   0:   08 af 0a 00     movw            %r0,    #0xA
   4:   00 00
   6:   0e ed           jpra            %r14
```

LLVM output with -O3 :

```
      main:
         .frame  %sp,0,%r14
         .mask   0x00000000,0
         .set    noreorder
         .set    nomacro
# BB#0:
        movw    %r0, #10
        jpra    %r14
        .set    macro
        .set    reorder
        .end    main
$tmp0:
        .size       main, ($tmp0)-main
```

Test Case 5

```
          int test()
          {
           int a = 5;
            int b = 2;
           int c, d, e, f, g, h, i;

             c = a + b;
            d = a - b;
             e = a * b;
             f = (a << 2);
             g = (a >> 2);
           h = (1 << a);
           i = (0x80 >> a);

            return (c+d+e+f+g+h+i);
             }
```

Reisc Toolchain Output with -O3 :

```
        00000000 <test>:
   0:   08 af 4d 00     movw            %r0,    #0x4D
   4:   00 00
   6:   0e ed           jpra            %r14
```

LLVM output with -O3 :

```
      test:
         .frame    %sp,0,%r14
         .mask     0x00000000,0
         .set      noreorder
```

```
        .set    nomacro
# BB#0:
        movw    %r0, #77
        jpra    %r14
        .set    macro
        .set    reorder
        .end    test
$tmp0:
        .size   test, ($tmp0)-test
```

Test Case 6

```
int test()
  {
   int b = 11;
   b = (b+1)*12;

   return b;
  }
```

Reisc Toolchain Output with -O3 :

```
00000000 <test>:
   0:   08 af 90 00     movw            %r0,    #0x90
   4:   00 00
   6:   0e ed           jpra            %r14
```

LLVM output with -O3 :

```
    test:
        .frame  %sp,0,%r14
        .mask   0x00000000,0
        .set    noreorder
        .set    nomacro
# BB#0:
        movw    %r0, #144
        jpra    %r14
        .set    macro
        .set    reorder
        .end    test
$tmp0:
        .size   test, ($tmp0)-test
```

Test Case 7

```
 int test()
   {
     int a = 5;
     int b = 3;
    int c = 0, d = 0, e = 0;

    c = (a & b);
    d = (a | b);
```

```
    e = (a ^ b);
    b = !a;

    return (c+d+e+b);
   }
```

Reisc Toolchain Output with -O3 :

```
      00000000 <test>:
   0:   08 af 0e 00     movw            %r0,    #0xE
   4:   00 00
   6:   0e ed           jpra            %r14
```

LLVM output with -O3 :

```
     test:
          .frame    %sp,0,%r14
          .mask     0x00000000,0
          .set      noreorder
          .set      nomacro
# BB#0:
          movw      %r0, #14
          jpra      %r14
          .set      macro
          .set      reorder
          .end      test
$tmp0:
          .size     test, ($tmp0)-test
```

Test Case 8

```
 int test()
   {
      int a = 5;
      int b = 3;
      int c, d, e, f, g, h;

      c = (a == b);
      d = (a != b);
      e = (a < b);
     f = (a <= b);
      g = (a > b);
     h = (a >= b);

      return (c+d+e+f+g+h);
   }
```

Reisc Toolchain Output with -O3 :

```
       00000000 <test>:
   0:   08 af 03 00     movw            %r0,    #0x3
   4:   00 00
   6:   0e ed           jpra            %r14
```

LLVM output with -O3 :

```
       test:
```

```
        .frame  %sp,0,%r14
        .mask   0x00000000,0
        .set    noreorder
        .set    nomacro
# BB#0:
        movw    %r0, #3
        jpra    %r14
        .set    macro
        .set    reorder
        .end    test
$tmp0:
        .size   test, ($tmp0)-test
```

Test Case 9

```
 int test()
 {
    int a = 3, b = 1;
    int d = 0, e = 0, f = 0;

    d = (a < 1);
     e = (b < 2);
     f = d + e;

     return (f);
  }
```

Reisc Toolchain Output with -O3 :

```
     00000000 <test>:
   0:   06 af           movw            %r0,    #1
   2:   0e ed           jpra            %r14
```

LLVM output with -O3 :

```
     test:
        .frame  %sp,0,%r14
        .mask   0x00000000,0
        .set    noreorder
        .set    nomacro
# BB#0:
        movw    %r0, #1
        jpra    %r14
        .set    macro
        .set    reorder
        .end    test
$tmp0:
        .size   test, ($tmp0)-test
```

Test Case 10

```
int test()
        {
          unsigned int a = 0;
          int b = 1;
          int c = 2;
          int d = 3;
          int e = 4;
          int f = 5;
          int g = 6;
            int h = 7;
            int i = 8;
            int j = 9;

            if (a == 0) {
              a++;
            }
            if (b != 0) {
              b++;
            }
            if (c > 0) {
              c++;
            }
            if (d >= 0) {
              d++;
            }
            if (e < 0) {
              e++;
            }
            if (f <= 0) {
              f++;
              }
              if (g <= 1) {
                g++;
              }
              if (h >= 1) {
                h++;
              }
              if (i < h) {
                i++;
              }
              if (a != b) {
                j++;
              }

               return (a+b+c+d+e+f+g+h+i+j);
        }
```

Reisc Toolchain Output with -O3 :

```
    00000000 <test>:
   0:   08 af 33 00     movw            %r0,    #0x33
   4:   00 00
   6:   0e ed           jpra            %r14
```

LLVM output with -O3 :

```
    test:
        .frame  %sp,0,%r14
        .mask   0x00000000,0
        .set    noreorder
        .set    nomacro
# BB#0:
        movw    %r0, #51
        jpra    %r14
        .set    macro
        .set    reorder
        .end    test
$tmp0:
        .size   test, ($tmp0)-test
```

Test Case 11

```
int main()
{
  int a=0;
  int b = 5;
  int i = 0;

  for (i = 0; i == 3; i++) {
    a = a + i;
  }
for (i = 0; i != 3; i++) {
    a = a + i;
  }
  for (i = 0; i > 3; i++) {
    a = a + i;
  }
  for (i = 0; i > 3; i++) {
    a = a + i;
  }
  for (i = 0; i == b; i++) {
    a++;
  }
  for (i = 0; i != b; i++) {
    a++;
  }
  for (i = 0; i < b; i++) {
    a++;
  }
  for (i = 7; i > b; i--) {
    a--;
  }
```

```
for (i = 0; i <= b; i++) {
  a++;
}
 return a;
}
```

Reisc Toolchain Output with -O3 :

LLVM output with -O3 :

Test Case

```
Int main()
{
  int a=0;
    int i = 0;

 while (i < 7) {
        a++;
        i++;
 if (a >= 4)
      continue;
 else if (a == 3) {
       break;
    }
  }
return a;

}
```

Reisc Toolchain Output with -O3 :

LLVM output with -O3 :

Test Case

```
int main()
{
  Int a=0;
  switch (a) {
   case 1:
   a = a+1;
  break;
  case 2:
 a = a+2;
  break;
 default:
  a = a+8;
 }
  return a;

 }
```

Reisc Toolchain Output with -O3 :

LLVM output with -O3 :

Test Case

```
int test()
{
  int a = 1,b = -2,c = 3;

  if (a == 0) {
   a++;
```

```
        }
       if (b == 0) {
           a = a + 3;
            b++;
        }
     else if (b < 0) {
           a = a + b;
           b--;
       }
     if (c > 0) {
           a = a + c;
          c++;
       }

      return a;
   }
```

Reisc Toolchain Output with -O3 :

LLVM output with -O3 :

Test Case

```
long test_shift()
 {
     long a = 4;
     long b = 0x12;
     long c;
      long d;

     c = (b >> a);
     d = (b << a);

       return (c+d);
 }
```

Reisc Toolchain Output with -O3 :

LLVM output with -O3 :

Test Case

```
int foo()
          {
           int i,a=0;
           for(i=0;i<5;i++)
                 a++;
           return a;
          }
```

Reisc Toolchain Output with -O3 :

```
00000000 <foo>:
0: 08 af 05 00     movw            %r0,    #0x5
4: 00 00
6: 0e ed           jpra            %r14
```

LLVM output with -O3 :

```
foo:
   .frame  %sp,0,%r14
   .mask   0x00000000,0
   .set    noreorder
   .set    nomacro
```

```
   # BB#0:
      movw    %r0, #5
      jpra    %r14
      .set    macro
      .set    reorder
      .end    foo
   $tmp0:
      .size   foo, ($tmp0)-foo
```

Test Case

```
int sum(int x1, int x2,int x3)
     {
      int sum =x3-x2+x1;
        return sum;

     }
```

Reisc Toolchain Output with -O3 :

```
      Disassembly of section .text:

    00000000 <sum>:
   0: 21 23           subw            %r2,    %r1
   2: 02 13           addw            %r0,    %r2
   4: 0e ed           jpra            %r14
```

LLVM output with -O3 :

```
  sum:
      .frame  %sp,0,%r14
      .mask   0x00000000,0
      .set    noreorder
      .set    nomacro
# BB#0:
      subw    %r0, %r1
      addw    %r0, %r2
      jpra    %r14
      .set    macro
      .set    reorder
      .end    sum
$tmp0:
      .size   sum, ($tmp0)-sum
```

Test Case

```
int foo(int i,int a)
          {

           for(i=0;i<5;i++)
                 a++;
           return a;
          }
```

Reisc Toolchain Output with -O3 :

```
   00000000 <foo>:
   0: 08 af 05 00     movw            %r0,    #0x5
   4: 00 00
   6: 01 13           addw            %r0,    %r1
   8: 0e ed           jpra            %r14
```

LLVM output with -O3 :

```
      foo:
      .frame  %sp,0,%r14
      .mask   0x00000000,0
      .set    noreorder
      .set    nomacro
# BB#0:
      addw    %r1, #5
      movw    %r0, %r1
      jpra    %r14
      .set    macro
      .set    reorder
      .end    foo
$tmp0:
      .size   foo, ($tmp0)-foo
```

6 CONCLUSION AND FUTURE WORK

Conclusion

This thesis describes the design, implementation and evaluation of a new back-end, based on Low Level Virtual Machine compiler infrastructure for the ReISC architecture.
ReISC is a 32-bit embedded architecture meant for ultra-low power devices. Having an enhanced RISC architecture, ReISC's instruction set has many extensions for digital signal processing operations. Some of the novel concepts of LLVM have been described such as the three phase design, LLVM intermediate representation and the type system.
The implemented backend can transform LLVM IR, generated from the source program by using the clang front end, into ReISC assembly code. The comparative measurements have shown that the assembly code generated by the LLVM back-end is in close proximity to the code generated by the GCC.

Future Work