LLVM Backend Development for ReISC Architecture

Student Name: Pooja Gupta

IIIT-D MTech CSE
July, 2015

Under the Supervision of Dr. Apala Guha

Indraprastha Institute of Information Technology Delhi
July, 2015

LLVM Backend Development for ReISC Architecture

Student Name: Pooja Gupta

IIIT-D MTech CSE

July, 2015

Indraprastha Institute of Information Technology
New Delhi

Submitted in partial fulfillment of the requirements
for the Degree ofMaster of Technology in Computer Science and Engineering

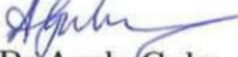
©2015 Indraprastha Institute of Information Technology (IIITD), New Delhi
All rights reserved

Certificate

This is to certify that the thesis titled **"LLVM Backend Development for ReISC Architecture"** being submitted by Pooja Gupta to the Indraprastha Institute of Information Technology Delhi, for the award of the *Master of Technology in Computer Science and Engineering*, is an original research work carried out by her . In my opinion, the thesis has reached the standards fulfilling the requirements of the regulations relating to the degree.

The results contained in this thesis have not been submitted in part or full to any other university or institute for the award of any degree/diploma.

July, 2015



Dr.Apala Guha
Department of Computer Science and Engineering
Indraprastha Institute of Information Technology Delhi
New Delhi, 110020

Acknowledgement

I would like to take the opportunity to thank those people who guided me and supported me during this period of my study. Without their valuable contributions, this work would not have been possible.
I am highly thankful to my M.Tech. supervisor, **Dr. Apala Guha**, for her guidance, advice and support throughout the thesis work. This thesis could not have been completed without her supervision and support.
I would also like to thank **Mr. Himal Prasad Ghimiray** and **Mr. Surinder Pal Singh** for their untired support and whole hearted help during my work.
I also thank IIIT Delhi and STMicroelectronics for providing an enthusiastic and peaceful atmosphere, well facilitated workplace to carry on research uninterrupted.
Last but not the least I would like to thank the continuous support and encouragement provided by my parents, who kept me motivated and cheered in the most difficult of times. . .

Pooja Gupta
MT13011

Abstract

In today's complex environment, it is very crucial for the devices to feature low power consumption at a low cost. Internet of Things is predicted to bring an era where everything will have chips embedded in them, be it a household appliance, mobile device or industrial equipment. To fulfill this vision, todays power-needy devices need to be replaced by less power consuming devices. Ultra Low Power chip design has this property which increases its demand.
ReISC (Reduced energy Instruction Set Computer) is an embedded architecture meant for low power devices with high performance applications. It provides support for secure data, parallel operations and fast interrupt response. To leverage these features of this architecture, building a compiler is essential.
This work describes the design and implementation of a new backend for the ReISC architecture based on Low Level Virtual Machine (LLVM) compiler infrastructure. This thesis contains the detailed discussion of translation of the code in LLVM intermediate representation to ReISC assembly code. It also includes the comparison between assembly code generated by the LLVM back-end and code generated by the ReISC toolchain.
The analysis of results implies that the LLVM backend generated code is in close proximity to toolchain generated code.

Contents

1 Introduction **1**

1.1 Goal of Thesis . . . 1

1.2 Approach . . . 2

1.3 Contributions . . . 3

1.4 Organization of Thesis . . . 3

2 The LLVM and the ReISC **4**

2.1 The LLVM Compiler Infrastructure . . . 4

2.1.1 The three phase design and its implications . . . 4

2.1.2 LLVM's Intermediate Representation . . . 6

2.1.3 Type System . . . 7

2.1.4 TableGen . . . 9

2.2 The ReISC Architecture . . . 10

2.2.1 Addressing Modes . . . 11

2.2.2 Code Generation . . . 12

3 Instruction Selection **13**

3.1 DAG Lowering . . . 13

3.2 DAG Legalization . . . 14

3.2.1 Type Legalization . . . 14

3.2.2 DAG Legalization . . . 15

3.3 ReISC Base Instructions . . . 16

3.4 Pattern Matching . . . 19

4 Register Allocation **21**

4.1 ReISC Registers . . . 22

4.2 Handling ReISC architecture constraints . . . 23

4.2.1 Calling Convention Constraints . . . 23

4.2.2 Handling two address instructions . . . 23

4.2.3 Handling Memory/Register Instructions . . . 23

4.3 Register Allocation in ReISC 24

5 CALL/RETURN HANDLING **25**

5.1 Call/Return mechanism of ReISC 25

5.2 Prologue/ epilogue code insertion in ReISC 27

6 CODE EMISSION **29**

6.1 Assembly printing in ReISC 29

6.1.1 Handling operand specifications 29

6.1.2 Handling instruction mnemonics 30

7 Results **31**

8 Conclusion **52**

List of Figures

2.1 Major components of a three-phase compiler. 5
2.2 The LLVM compiler infrastructure. 5
2.3 The LLVM type system: Primitive types. 8
2.4 The LLVM type system: Derived types. 9
2.5 The ReISC architecture. 10
2.6 Transformation steps of code generation process. 12

3.1 Instruction selection process. 14
3.2 The ReISC base instruction set: Move operations. 17
3.3 The ReISC base instruction set: Arithmetic operations. 17
3.4 The ReISC base instruction set: Shift operations. 18
3.5 The ReISC base instruction set: Logic operations. 19
3.6 The ReISC base instruction set: Branch operations. 19

4.1 Standard Registers with special functionality. 22

5.1 Division of standard registers based on their usage. 26
5.2 Stack layout. 27

Chapter 1

Introduction

In today's complex environment, it is very crucial for the devices to feature low power consumption 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, todays 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 should share a commonality and perform 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 frontend already available should be used and backend part should be written. The goal should be to minimize the effort in writing the backend and decrease information redundancy. Moreover the system should be modular, reusable, maintainable and easily extensible.

1.2 Approach

The GNU Compiler Collection supports a large number of frontend and backend 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.

In this approach LLVM(Low-Level Virtual Machine) was chosen as it overcomes these limitations. LLVM supports the feature of pluggable frontends. It provides the flexibility to write backend by allowing the use of already present frontends.

LLVM is written as a set of libraries and therefore it allows the reusability of classes and sharing of components across different compilers as often as possible. It automates a lot of things at 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 example, instruction set of architecture is described in Instrinfo.td and then TableGen processes this file to generate the instruction selection algorithm, this would have been 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 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 instruction selection phase of the code generation process.

Chapter 4 describes the register allocation phase.

Chapter 5 contains the description of the call/return handling procedure.

Chapter 6 explains the code emission process, the final phase of code generation process.

Chapter 7 highlights the results generated by the LLVM backend and its comparison with ReISC tool chain results.

Chapter 8 discusses about the conclusion.

Chapter 2

The LLVM and the ReISC

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.

2.1.1 The three phase design and its implications

LLVM has a three phase design comprised of front end, optimizer and backend as shown in Figure 2.1:

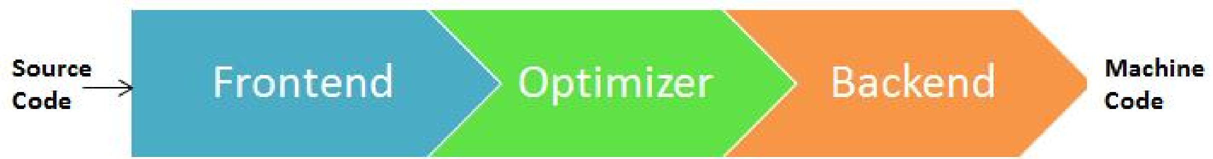


Figure 2.1: Major components of a three-phase compiler.

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 intermediate representation 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.

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. Figure 2.2 shows the LLVM compiler structure.

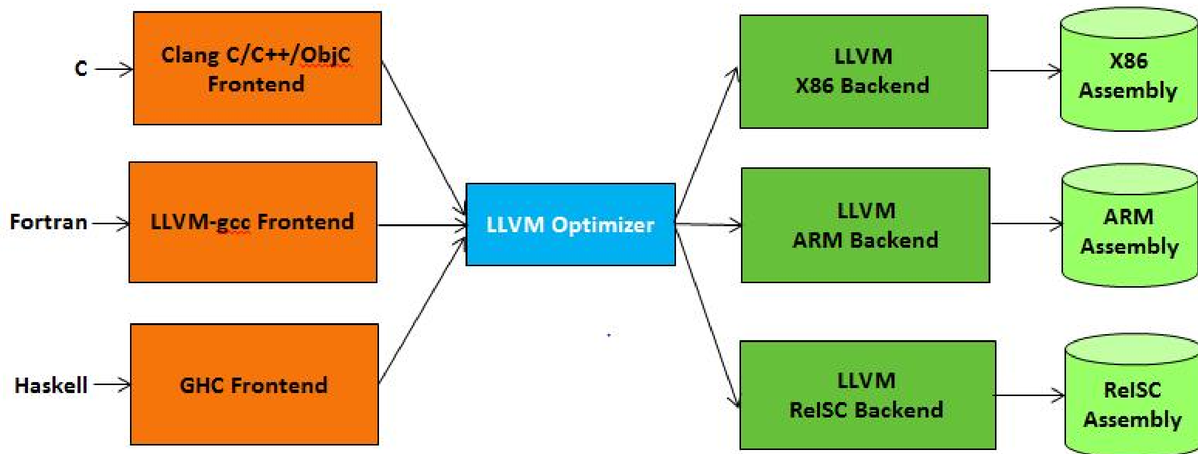


Figure 2.2: The LLVM compiler infrastructure.

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.

2.1.2 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 consist 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 blocks 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)

This 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.

2.1.3 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. Figure 2.3 shows the primitive types in LLVM system.

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

Figure 2.3: The LLVM type system: Primitive types.

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. Figure 2.4 shows the derived types in LLVM system.

Type	Syntax	Overview
Function	i32 (i32)	Function taking an i32, returning an i32
Pointer	<type> *	Used to specify memory locations
Array	[<# elements> x <element type>]	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

Figure 2.4: The LLVM type system: Derived types.

2.1.4 TableGen

TableGen is the descriptive language used by LLVM to describe several machine aspects used in 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 ReISCRegisterInfo.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.

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. Architechture ReISC is shown in Figure 2.5 .

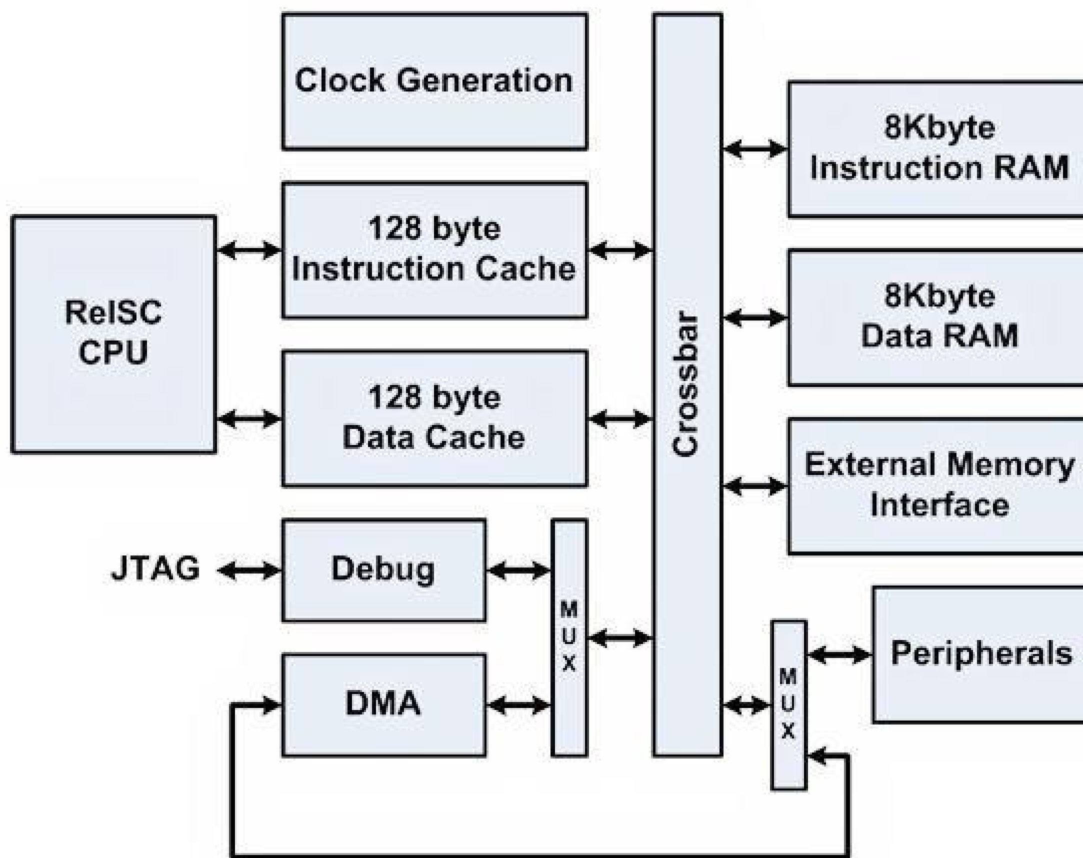


Figure 2.5: The ReISC architecture.

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 bytes wide cache act as an interface to the SoCs on-chip instruction and 128 bytes 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.

2.2.1 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 following ...

- 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

2.2.2 Code Generation

The code generation process is comprised of various passes that analyze and transform the LLVM intermediate representation (IR) into assembly code. The IR changes after each pass and gets more similar to the target instructions. The following diagram illustrates the steps of transformation of intermediate representation to assembly code. Figure 2.6 shows the transformation steps of code generation process.

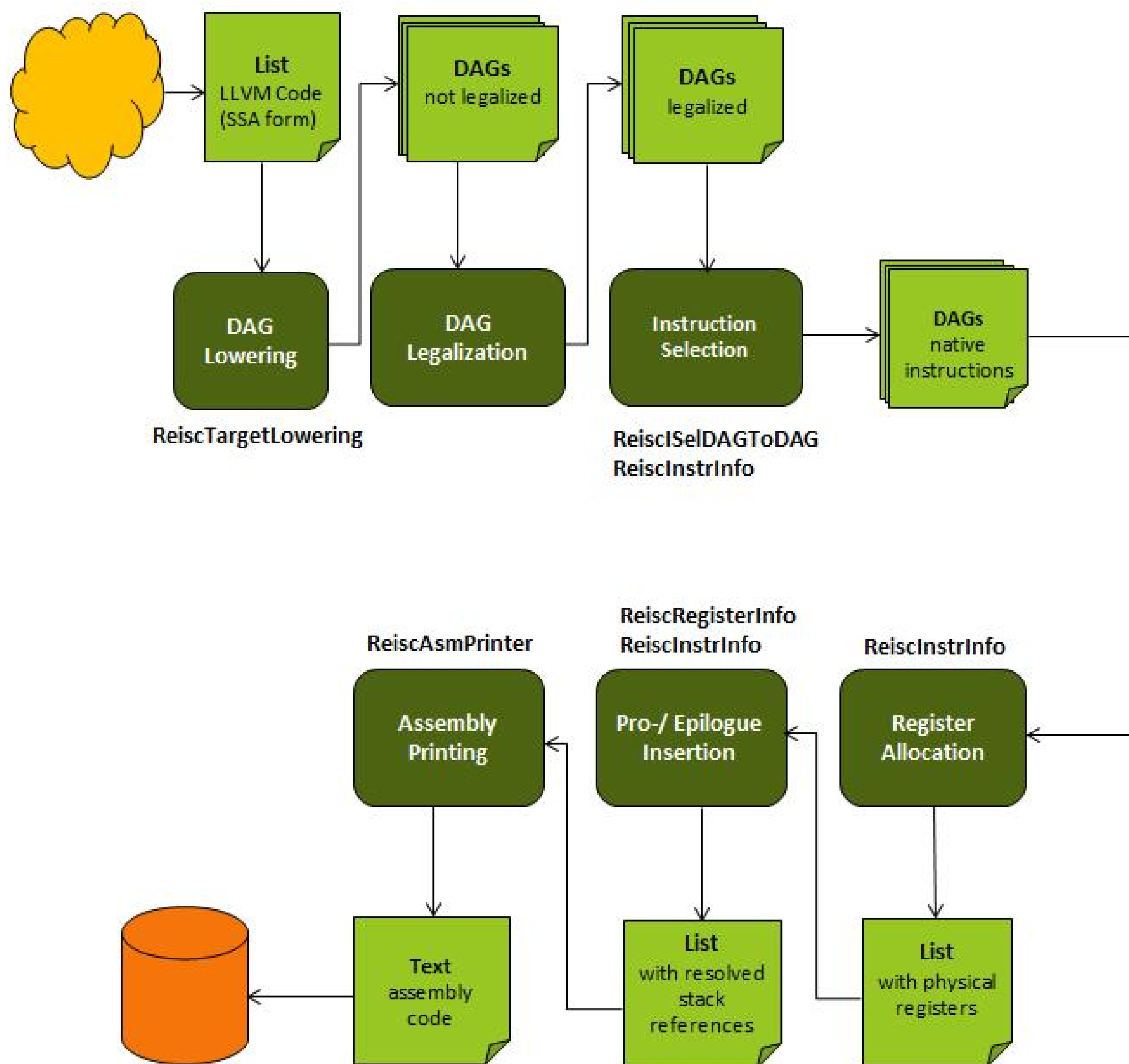


Figure 2.6: Transformation steps of code generation process.

Each step of the code generation process is explained in detail in the upcoming chapters.

Chapter 3

Instruction Selection

In this chapter of thesis work, we describe the process of instruction selection in which the instructions in LLVMs intermediate representation are mapped to the corresponding target specific instructions.

The transformation of the LLVM code into a set of DAGs is the first step of the instruction selection phase. DAG refers to a directed acyclic graph representation where nodes denote instructions and the edges denote definitionuse relationship among them. Target architecture do not support all operations on all types, hence the DAG needs to be transformed so that it contains supported data types and operations only. This transformation process is called Dag legalization.

The DAG nodes are then transformed into nodes that represent the target specific instructions. By the end of instruction selection phase, all the nodes in the DAG will be target specific.

3.1 DAG Lowering

The code generation process starts with intermediate representation. This language and target independent intermediate representation needs to be converted into a representation that can run on some certain architecture. DAG lowering performs this conversion and lowers the code in intermediate representation to a DAG representation whose nodes represent target independent instructions that can be worked upon by the target specific instruction selector.

DAG lowering converts intermediate representation to DAG with nodes having target indepen-

dent instructions, however during this transformation process, DAG nodes representing a few special operations such as in ReISC architecture call and ret i.e. how to pass arguments during a function call and how to return once the function call is over, already need to be handled according to the specifications of ReISC. These DAG nodes are converted into target specific nodes at this stage. The matching and replacement of all other remaining nodes is performed at the time of instruction selection. Instruction selection process is shown in Figure 3.1 .

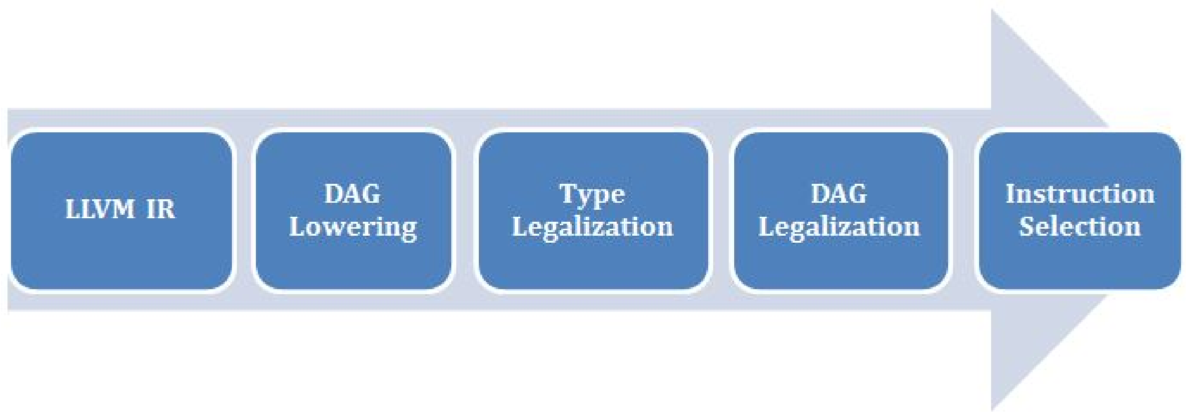


Figure 3.1: Instruction selection process.

As a result of this phase, the DAG may contain both target specific and target independent nodes.

3.2 DAG Legalization

DAG Legalization phase of code generation process transforms the DAG to eliminate any instructions that are unsupported by the target architecture and are not prepared for instruction selection. There are two phases in DAG legalization: DAGs Type Legalization and DAGs Legalization.

3.2.1 Type Legalization

The purpose of type legalization phase is to make sure that instruction selection only needs to operate on legal types i.e. the types that are supported by the target natively. For example, an operation with boolean operands is illegal in target that supports i32 types only. Targets contain the explicit declaration of the supported types by defining the register classes associated

with each type. So, the illegal types are must to be identified and handled.

The illegal types can be transformed into legal ones by following techniques:

Promote: This technique converts small type to larger type supported by the target.

LLVM intermediate representation supports all i1, i8, i16 and i32 data types whereas ReISC architecture does not support i1 (Boolean) data type, so this is promoted.

Expand: This technique splits larger type hence converting it into smaller one.

ReISC architecture does not support long long data type which is 64 bit. To handle this long long type is expanded.

All these transformations should be made in such a way that the final code retains its behaviors.

3.2.2 DAG Legalization

There are some operations which are not supported by the target for a given type. So, the purpose of DAG legalization phase is to convert such operations into operations that are natively supported by the target by changing the data type of operands, by using a set of other supported operations having similar effect or by using a specialized function.

The unsupported operations can be transformed into supported ones by following techniques:

Promote: This technique converts an operation that is not supported by the target natively for a given type to a larger type that is supported.

Expand: This technique splits an operation that is not supported by the target natively into a set of operations that are supported and have the similar effect.

In ReISC, expansion operation is used to handle the conditional instructions.

Custom: This technique is used for the operations that are not supported by the target natively and for which promotion and expansion does not work. For such operations, a special function is written that simulates its behavior.

In ReISC, custom is used to handle the branch conditions. A specialized function is written and called in this case.

3.3 ReISC Base Instructions

There are three categories of instructions in the instruction set of ReISC architecture: BASE instructions, LONG instructions and DSP instructions out of which the base instruction set falls within the scope of this thesis.
The BASE instructions consist of the following groups of operations ...

- Move operations
- Arithmetic operations
- Shift operations
- Logic operations
- Branch operations
- Special operations

In ReISC, instructions follow two-address format, i.e., src1 (dest) src1 OP src2, where one of the source operands works as destination as well in order to improve the code density. 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.

Move operations

Move operations consist of five instructions: load, store, mov, mpush, mpop. Figure 3.2 shows Move operations of the ReISC base instruction set.

Move Operations	Overview
LOAD	Moves data into a ReISC standard register from memory location
STORE	Moves data into memory location from a ReISC standard register
MOV	Moves data into one of the ReISC standard registers from another ReISC standard register or immediate
MPUSH	Moves data into memory locations from a group of ReISC standard registers
MPOP	Moves data into a group of ReISC standard registers from memory locations

Figure 3.2: The ReISC base instruction set: Move operations.

Arithmetic operations

Arithmetic operations consist of six instructions: uext, sext, add, sub, div, and mul. Figure 3.3 lists Arithmetic operations of the ReISC base instruction set.

Arithmetic Operations	Overview
UEXT	Stores the result obtained from extending an unsigned data value in ReISC standard register to unsigned 32-bit value, into another standard register
SEXT	Stores the result obtained from extending a signed data value in ReISC standard register to signed 32-bit value, into another standard register
ADD	Addition of two operands
SUB	Subtraction of two operands
MUL	Multiplication of two integer operands
DIV	Division of on operand by another one

Figure 3.3: The ReISC base instruction set: Arithmetic operations.

In add, sub, div, and mul instructions, one operand is a register and the other could be a register, immediate or memory location.

Shift operations

Shift operations consist of five instructions: srl, sra, sla, rot, and rotc. In Figure 3.4 Shift operations of the ReISC base instruction set are shown.

Shift Operations	Overview
SRL	Logical right-shift of one operand by an amount specified by the immediate field or register value
SRA	Arithmetic right-shift of one operand by an amount specified by the immediate field or register value
SLA	Logic left-shift of one operand by an amount specified by the immediate field or register value
ROT	Logical right-rotation of one operand by an amount specified by the immediate field or register value
ROTC	Logical rotation with carry flag, of one operand by 1 bits

Figure 3.4: The ReISC base instruction set: Shift operations.

Rot instruction performs logical left-rotation of one operand by an amount specified by the immediate field or register value if the immediate or register value is negative. In case of rotc instruction, the dir field of instruction encoding specifies direction of rotation.

Logic operations

Logic operations consist of four instructions: and, or, xor, and cmp.

One of the operands in and, or and xor instruction can be a register, immediate or memory location. Figure 3.5 shows Logic operations of the ReISC base instruction set.

Logic Operations	Overview
AND	Bitwise logical AND operation of two operands
OR	Bitwise logical OR operation of two operands
XOR	Bitwise logical XOR operation of two operands
CMP	Compares two operands where one operand can be a register, immediate or memory location

Figure 3.5: The ReISC base instruction set: Logic operations.

Branch operations

Branch operations consist of six instructions: jpd, jpi, jpr, jlr, jli, and rfe. Branch operations of the ReISC base instruction set are shown in Figure 3.6 .

Branch Operations	Overview
JPD	Performs pc-relative conditional (or unconditional) branch
JPI	Performs absolute (immediate) conditional (or unconditional) branch
JPR	Performs absolute (register) conditional (or unconditional) branch
JLR	Store return address in link register R14 & uses absolute address in register as the call target
JLI	Store return address in link register R14 & uses immediate address as the call target
RFE	Used to return from trap, exception or interrupt

Figure 3.6: The ReISC base instruction set: Branch operations.

3.4 Pattern Matching

Pattern Matching is the most important phase of the code generation process. It aims at converting the legalized DAG to a new DAG whose nodes contain ReISC specific instructions by matching the abstract, target-independent nodes to concrete, ReISC specific nodes.

All the special cases such as the operations and types that are not supported by our architecture have already been handled in the previous steps and we are left with the simple cases to be

pattern matched.

Pattern matching is performed with the help of a table which contains all the patterns generated from instruction definitions written in the instruction description table of the ReISC architecture. Now, a one to one matching is performed using this table where for each target independent pattern of legalized DAG, an appropriate ReISC specific pattern is selected. After this step, the DAGs that are generated contain nodes with reisc specific instructions in each of them.

Chapter 4

Register Allocation

In this chapter of thesis work, we describe the process of deconstruction of static single assignment form of intermediate representation in which virtual registers are eliminated by mapping them to physical registers.

The instruction selector generates code which is in static single assignment form according to which there is infinite number of registers available. The code generator invokes a register allocator, which performs the task of replacing infinite number of virtual registers by physical registers. Since targets register bank have a limited number of physical registers, spill code is produced if the number of available physical registers is less than those of live virtual registers, wherein some virtual registers are assigned to memory locations known as the spill slots.

In some cases, physical registers might already be used in code fragments even before the process of register allocation. This happens because in some architectures there are instructions that require a specific register like register used to return value. In such cases, register allocator handles all other virtual registers and maps them to physical registers.

The static single assignment representation of instructions may contain phi instructions (explained in previous section). While deconstructing this representation, the phi instructions need to be replaced with general instructions and to achieve this copy instruction is used in place of phi instructions.

4.1 ReISC Registers

The ReISC's register bank contains sixteen standard registers of 32 bit each, a program counter register and four instruction result flags Carry, Zero, Negative and Overflow. Only standard register set is used for implementing the base instruction set of the reisc architecture.

The standard register set

There are 16 registers, R0 to R15 in the standard register set of ReISC target. The width of each register is 32-bit and can hold 8-bit, 16-bit, 20-bit and 32-bit values. If the width of operands is less than that of the registers like 8 bit or 16 bit, only the lower bits of the 32 bit destination register are altered whereas the higher bits remain unaffected.

In case of operands having width 20-bit, which is also less than the width of the registers, destination register is updated in a manner different from the previous one. Here after execution, the result having 20-bit value is padded with 12 bit zeros in the higher bits and then this 32 bit value is copied to the destination register. Other than being used as source or destination operands, some of the standard registers have special functionality as indicated in the Figure 4.1:

Standard Registers	Alias Name	Special Functionality
R14	Link register	For a function call
R15	Stack pointer	For MPUSH and MPOP instructions

Figure 4.1: Standard Registers with special functionality.

The standard register R14 is referred by the name Link Register which is used to store the value of program counter during a procedure call, such as jump and link instruction and to jump again to the caller function by retrieving the value of program counter stored in it. The standard register R15 is referred by the name Stack Pointer, which is the stack pointer used for the MPUSH and MPOP instruction.

When the special functionality of these registers is not being used, they serve as normal 32 bit registers for example, standard register R14 (link register) is used as a normal register in case

of non-leaf functions since it does not require any jump or return to be made.

4.2 Handling ReISC architecture constraints

4.2.1 Calling Convention Constraints

The ReISC architecture has 16 standard registers out of which two registers R14 and r15 are reserved to perform the dedicated functionality as a link register and stack pointer register. Other remaining 14 registers are freely available for normal use, even reserved registers R14 and R15 can be used as normal standard registers when the special functionality of these registers is not in use.

Out of 14 standard registers available freely, the ReISC instruction set has some constraints on their use. The registers in which the arguments should be passed and the register which holds the return value is fixed. Standard Registers R0, R1, R2 and R3 are used to pass arguments and the return value is held in register R0. These registers will be preassigned and due to this there are code fragments in which physical registers are used even before the process of register allocation. In such cases, register allocator handles the remaining virtual registers.

4.2.2 Handling two address instructions

The LLVM machine code instructions follow three address format. However, ReISC architecture follow two address format i.e. one of the source registers acts as destination register as well. So to produce correct code, the three address instructions must be converted into two address instructions. To achieve this conversion, the operand could have been deleted from the machine instruction but this will be problematic. So, the operand is not deleted. Instead the two operands hold the very same register after the register allocation phase.

4.2.3 Handling Memory/Register Instructions

ReISC architecture belongs to memory/register architecture where one of the operands needs to be in the register and the other one is read directly from a memory location.

To achieve this all the instructions from the beginning of basic block are scanned and a list of

occupied registers is created along with the information of register being dirty or not. Then a load or move instruction is generated to put the value of one operand into a free register.
Now if the register is freely available, it can be easily allocated to the operand. If the register is not available, the value of a used register will be spilled wherein the value will be written to the stack if the register is live and dirty. After the basic block ends, dirty and live registers are written back so that the value previously present in the register doesnt get lost.

4.3 Register Allocation in ReISC

The register allocation process follows the concept of live variable analysis in which the variables which are present in move instructions and which are live at same time instance are determined. An interference graph is constructed based on this information whose vertices represent a unique variable. The vertices which are live at the same time instance are connected by interference edges and vertices which are present in move instructions are connected by preference edges.
Now we have to color the nodes of the interference graph and this problem can be reduced to K-coloring graph problem where K represents the number of available physical registers of ReISC architecture. To handle the calling convention constraint, some vertices are colored at the beginning. Two vertices connected via an interference edge are never assigned the same color whereas the vertices connected via a preference edge are assigned the same color whenever possible.
If the interference graph cannot be colored following the above mentioned technique, some variables are assigned to memory locations (spilled). After this, K-coloring technique is applied again on the remaining variables. This process of coloring and spilling goes on recursively till the graph is colored completely i.e. remaining variables in the graph are colored and registers are allocated to them.

Chapter 5

CALL/RETURN HANDLING

This chapter of thesis work describes a special handling mechanism that is required for the function call and return to happen smoothly and efficiently.

Whenever a function call is encountered, the actual arguments are copied before calling the function. The function call starts with function prologue which performs the reservation of callee saved registers and stack frame by decrementing the stack pointer. When the scope of a function is entered, the location of each formal argument is determined, they are moved into the virtual registers and a sequence of move and/or load instructions is inserted into the DAG. While returning from the function, the return value is copied into a virtual register.

When the scope of a function is left, the return value which is present in a virtual register is copied in the corresponding physical register and during the instruction selection phase, matching with the return instruction is performed. The function call ends with function epilogue which destroys the reserved stack frame and restores all saved registers before returning from a function.

5.1 Call/Return mechanism of ReISC

In ReISC architecture standard registers are of fixed size of 32 bit each whereas variables sizes vary from 1 byte for character data type to 4 bytes for integer data type and varying size for structure and union depending on their data members. So, the registers are assigned to the variables according to their size. The variable with size less than or equal to the size of standard registers is assigned one register and the variables with size greater than standard registers size

are assigned more than one registers in little endian order.

Out of the 14 standard registers available freely, they have been divided into two sections based on their usage: callee changed registers and callee saved registers as shown in Figure 5.1 .

Standard Registers	Class
R0	Callee changed registers: could be changed by called function
R1	
R2	
R3	
R4	Callee saved registers: unchanged by called function
R5	
R6	
R7	
R8	
R9	
R10	
R11	
R12	
R13	

Figure 5.1: Division of standard registers based on their usage.

Callee changed registers can be clobbered by the called function so the value already present in these registers must be saved before they are used. After the function call finishes the original value that was present in these registers before the function call is restored and then the control returns to the caller.

Callee saved registers cannot be clobbered by the called function. The value already present in these registers must be saved before their use and restored to its original state by the called function.

For a function with fixed argument list, all the arguments are allocated on callee changed registers. If the number of arguments is less than the callee changed registers available, allocation

becomes very easy and registers are allocated to these arguments in left to right order in increasing manner i.e. from R0 to R3. If the number of arguments increases the number of callee changed registers, arguments are allocated on these registers in increasing order until their availability and the remaining arguments are pushed on the stack.
The result of the function call is also allocated in callee changed registers.

5.2 Prologue/ epilogue code insertion in ReISC

When a function is called, function prologue comes into picture. It creates an area in the stack which holds the values of all the registers that are clobbered by the called function known as saved register area as shown in Figure 5.2 . Link register R14 is pushed onto the stack if function is a non-leaf function. Value of frame pointer is computed from the value of stack pointer and saved in register R4. Another area called function frame area is created whose size is determined by decrementing the stack pointer by the size of frame. This area contains all the automatic variables of the called function.

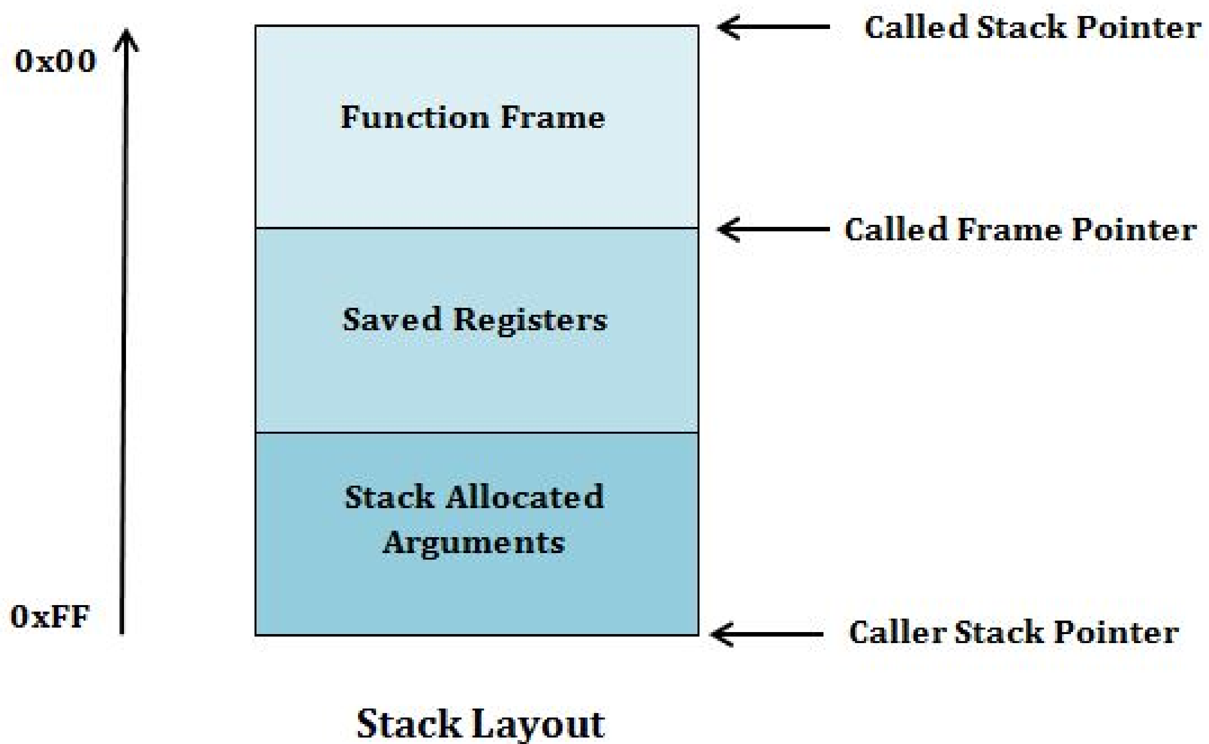


Figure 5.2: Stack layout.

When a function call finishes, function epilogue comes into picture. It restores the stack pointer

to its original value by incrementing it by the size of function frame. If the function is a non-leaf function, the link register R14 must have been saved into the stack during prologue insertion. Now, the link register is popped out of the stack. Callee saved registers that are stored in the saved register area of stack are popped out of the stack.

Chapter 6

CODE EMISSION

This chapter of thesis work describes the final phase of the code generation process which performs the emission of completed assembly code.

In this phase, for each function first the function header is emitted and then the basic blocks are processed where each machine instruction is processed and the corresponding assembly string is printed out. There are respective appointed print methods which handle the printing of different operands such as immediate and memory operands.

6.1 Assembly printing in ReISC

6.1.1 Handling operand specifications

The ReISC architecture has a particular specification for each kind of operands described below:

Register Specification: All the 16 standard registers are denoted by integer numbers that range from 0 to 15 and registers are preceded by % sign.

Immediate operand specification: Immediate operands are represented in hexadecimal notation and are preceded by # sign.

Memory operand specification: In case of memory operands, [] is used as the addressing mode bracket.

6.1.2 Handling instruction mnemonics

Different architectures have different instruction assembler mnemonic for same instruction. So, the instruction printer should have the information of how each instruction is represented in architecture. To handle this issue, all the assembly strings for a target are defined by adding them to the instruction definitions and this string is then used by the assembly printer to emit corresponding target specific instruction mnemonic.

While emitting the assembly code, first of all the initialization are done. The constant values that have been spilled to memory are printed out next after which the jump tables that the current function uses are printed out. Then the current functions label is emitted.

Next is the code for the function where label of each basic block is emitted, each instruction of basic block is processed and its assembly code is generated with the help of dedicated print methods for the target architecture.

At last when the assembly printer has finished processing each function, global variables and constants are emitted.

Chapter 7

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 foo()
{
    return 4;
}
```

LLVM Output

```
foo:
    .frame    %sp,0,%r14
    .mask     0x00000000,0
    .set      noreorder
    .set      nomacro
# BB#0:
    movw      %r0, #0x4
    jpra      %r14
    .set      macro
    .set      reorder
```

```
    .end        foo
$tmp0:
    .size       foo, ($tmp0)-foo
```

Toolchain Output

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

In this test case, the only operation performed in the fuction foo() is to return an interger value 4. In LLVM output, value 4 is moved to register R0 since R0 is the return register.Toolchain produces the similar assembly code.

Test Case 2

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

LLVM Output

```
foo:
    .frame    %sp,8,%r14
    .mask     0x00000000,0
    .set      noreorder
    .set      nomacro
# BB#0:
    addw      %sp, #0xfffffffffffffff8
    movw      %r0, #0x5
    stw       4[%sp], %r0
    addw      %sp, #0x8
```

```
    jpra     %r14
    .set     macro
    .set     reorder
    .end     foo
$tmp0:
    .size    foo, ($tmp0)-foo
```

Toolchain Output

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

In this test case, in fuction foo() first value 5 is assigned to integer variable a and then same value is returned. In LLVM output, value 5 is moved to register R0 (movw %r0, #0x5)since R0 is the return register and then the value of register R0 which is 5 is stored at apposition respective to stack pointer(stw 4[%sp],%r0) for int a=5. Toolchain produces the similar assembly code.

Test Case 3

```
int main()
{
    int a ;
    int b ;
    int c = a + b;
    return (c);
}
```

LLVM Output

```
main:
    .frame   %sp,16,%r14
    .mask    0x00000000,0
```

```
        .set        noreorder
        .set        nomacro
# BB#0:
        addw        %sp, #0xfffffffffffffff0
        movw        %r0, #0x0
        stw         12[%sp],%r0
        ldw         %r1, 4[%sp]
        ldw         %r0, 8[%sp]
        addw        %r0, %r1
        stw         0[%sp],%r0
        addw        %sp, #0x10
        jpra        %r14
        .set        macro
        .set        reorder
        .end        main
$tmp0:
        .size       main, ($tmp0)-main
```

Toolchain Output

```
00000000 <main>:
0:    44 f3           mpushw   %r4,     %r4
2:    4f ab           movw     %r4,     %r15
4:    f8 27 0c 00     subw     %r15,    #0xC
8:    00 00
a:    14 87 fc ff     ldw      %r1,     -4[%r4]
e:    0c af           movw     %r0,     #-8
10:   04 13           addw     %r0,     %r4
12:   00 8b           ldw      %r0,     [%r0]
14:   01 13           addw     %r0,     %r1
16:   04 97 f4 ff     stw      -12[%r4],     %r0
1a:   04 87 f4 ff     ldw      %r0,     -12[%r4]
```

```
1e:    f8 17 0c 00    addw    %r15,    #0xC
22:    00 00
24:    44 f7          mpopw   %r4,     %r4
26:    0e ed          jpra    %r14
```

In LLVM output for this test case, the value of a is copied in register R1 and value of b is copied in register R0 (ldw %r1, 4[%sp] and ldw %r0, 8[%sp]).
Next addw instruction performs the addition of these two variables a and b. The final result is in register R0 which is then moved to a position respective to stack pointer. After this the control returns.

Test Case 4

```
int test()
{
    int a = 5;
    int b = 2;
    int e, f;
    e = a * b;
    f = (a << 2);
    return (e+f);
}
```

LLVM Output

```
test:
    .frame    %sp,16,%r14
    .mask     0x00000000,0
    .set      noreorder
    .set      nomacro
# BB#0:
    addw      %sp, #0xfffffffffffffff0
    movw      %r0, #0x5
```

```
	stw	12[%sp], %r0
	movw	%r0, #0x2
	stw	8[%sp], %r0
	ldw	%r0, 12[%sp]
	slaw	%r0, #0x1
	stw	4[%sp], %r0
	ldw	%r1, 12[%sp]
	slaw	%r1, #0x2
	stw	0[%sp], %r1
	ldw	%r0, 4[%sp]
	addw	%r0, %r1
	addw	%sp, #0x10
	jpra	%r14
	.set	macro
	.set	reorder
	.end
$tmp0:
	.size	test, ($tmp0)-test
```

Toolchain Output

```
00000000 <test>:
0:	44 f3		mpushw	%r4,	%r4
2:	4f ab		movw	%r4,	%r15
4:	f2 27		subw	%r15,	#16
6:	08 af 05 00	movw	%r0,	#0x5
a:	00 00
c:	04 97 fc ff	stw	-4[%r4],	%r0
10:	05 af		movw	%r0,	#2
12:	04 97 f8 ff	stw	-8[%r4],	%r0
16:	14 87 fc ff	ldw	%r1,	-4[%r4]
1a:	0c af		movw	%r0,	#-8
```

```
1c:   04 13         addw    %r0,    %r4
1e:   00 8b         ldw     %r0,    [%r0]
20:   01 43         mulw    %r0,    %r1
22:   04 97 f4 ff   stw     -12[%r4],    %r0
26:   04 87 fc ff   ldw     %r0,    -4[%r4]
2a:   01 c7         slaw    %r0,    #2
2c:   04 97 f0 ff   stw     -16[%r4],    %r0
30:   14 87 f4 ff   ldw     %r1,    -12[%r4]
34:   0d af         movw    %r0,    #-16
36:   04 13         addw    %r0,    %r4
38:   00 8b         ldw     %r0,    [%r0]
3a:   01 13         addw    %r0,    %r1
3c:   f2 17         addw    %r15,    #16
3e:   44 f7         mpopw   %r4,    %r4
40:   0e ed         jpra    %r14
```

In LLVM outputs basic block BB0_0, first four movw and stw instructions are used to initialize the varaibles a and b with values 5 and 2 respectively. The value of a is loaded in R0 (ldw %r0, 12[%sp]) and then multiplied by 2 using slaw instruction since left shift by 1 is equivalent to multiplication by 2.This value is stored at a position respective to stack pointer which is variable e. Again the value of a is loaded in R1 (ldw %r1, 12[%sp]) and then left shift by 2 is performed using slaw instruction (slaw %r1, #0x2) and this result is stored at a position respective to stack pointer which is variable f.

Next the value of e is loaded in R0 and added with the value in register R1 i.e. f after which control returns from the function.

Test Case 5

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

```
    return b;
}
```

LLVM Output

```
test:
    .frame    %sp,8,%r14
    .mask     0x00000000,0
    .set      noreorder
    .set      nomacro
# BB#0:
    addw      %sp, #0xfffffffffffffff8
    movw      %r0, #0xb
    stw       4[%sp], %r0
    movw      %r0, #0x90
    stw       4[%sp], %r0
    addw      %sp, #0x8
    jpra      %r14
    .set      macro
    .set      reorder
    .end      test
$tmp0:
    .size     test, ($tmp0)-test
```

Toolchain Output

```
00000000 <test>:
0:    44 f3          mpushw    %r4,    %r4
2:    4f ab          movw      %r4,    %r15
4:    f4 27          subw      %r15,   #4
6:    08 af 0b 00    movw      %r0,    #0xB
a:    00 00
c:    04 97 fc ff    stw       -4[%r4],    %r0
```

```
10:   04 87 fc ff    ldw    %r0,    -4[%r4]
14:   06 17          addw   %r0,    #1
16:   08 47 0c 00    mulw   %r0,    #0xC
1a:   00 00
1c:   04 97 fc ff    stw    -4[%r4],    %r0
20:   04 87 fc ff    ldw    %r0,    -4[%r4]
24:   f4 17          addw   %r15,   #4
26:   44 f7          mpopw  %r4,    %r4
28:   0e ed          jpra   %r14
```

In LLVM output for this test case, first two instructions movw and stw are used to initialize the variable b with value 11 which is 0xb in hexadecimal representation. Next the arithmetic expression ((b+1)*12) is evaluated which is equal to 144 in decimal representation and 0x90 in hexadecimal representation.

The next move instruction copies the final result (0x90) to register R0 after which control returns from the function.

Test Case 6

```
int test()
{
    unsigned int a = 0;
    int b = 1;
    if (a == 0)
    {
     a++;
    }
    return (a);
}
```

LLVM Output

test:

```
        .frame      %sp,8,%r14
        .mask       0x00000000,0
        .set        noreorder
        .set        nomacro
# BB#0:
        addw        %sp, #0xfffffffffffffff8
        movw        %r0, #0x0
        stw         4[%sp], %r0
        movw        %r0, #0x1
        stw         0[%sp], %r0
        ldw         %r0, 4[%sp]
        cmpw        %r0, #0x0
        jpdne       $BB0_2
        jpda        $BB0_1
$BB0_1:
        ldw         %r0, 4[%sp]
        addw        %r0, #0x1
        stw         4[%sp], %r0
$BB0_2:
        ldw         %r0, 4[%sp]
        addw        %sp, #0x8
        jpra        %r14
        .set        macro
        .set        reorder
        .end        test
$tmp0:
        .size       test, ($tmp0)-test
```

Toolchain Output

```
00000000 <test>:
0:      44 f3           mpushw      %r4,      %r4
```

```
2:   4f ab         movw    %r4,   %r15
4:   f3 27         subw    %r15,  #8
6:   07 af         movw    %r0,   #0
8:   04 97 fc ff   stw     -4[%r4],  %r0
c:   06 af         movw    %r0,   #1
e:   04 97 f8 ff   stw     -8[%r4],  %r0
12:  04 87 fc ff   ldw     %r0,   -4[%r4]
16:  07 a7         cmpw    %r0,   #0
18:  06 d2         jpdne   24<test+0x24>
1a:  04 87 fc ff   ldw     %r0,   -4[%r4]
1e:  06 17         addw    %r0,   #1
20:  04 97 fc ff   stw     -4[%r4],  %r0
24:  04 87 fc ff   ldw     %r0,   4[%r4]
28:  f3 17         addw    %r15,  #8
2a:  44 f7         mpopw   %r4,   %r4
2c:  0e ed         jpra    %r14
```

In the LLVM outputs basic block BB0_0, first four movw and stw instructions are used to initialize the varaibles a and b. The value of a is loaded in R0 (ldw %r0, 4[%sp]). Then the value of R0 (i.e a)is compared with 0. If this condition is not satisfied (i.e. a!=0), control goes to basic block $BB0_2 as indicated by instruction jpdne $BB0_2 where value of a is loaded in register R0 (by ldw instruction) and then control returns.

If the above condition is satisfied (i.e. a==0) control goes to basic block $BB0_1. In BB0_1, value of a is loaded in register R0 (by ldw instruction), increment by 1 is performed by addw instruction and incremented value is stored(by stw)instruction.

Next, control goes to BB0_2 where value of a is loaded in register R0 (by ldw instruction) and then control returns.

Test Case 7

```
int test()
{
```

```
    int h = 7;
    int i = 8;
    if (i <h)
    {
     i++;
    }
    return (i);
}
```

LLVM Output

```
test:
    .frame      %sp,8,%r14
    .mask       0x00000000,0
    .set        noreorder
    .set        nomacro
# BB#0:
    addw        %sp, #0xfffffffffffffff8
    movw        %r0, #0x7
    stw         4[%sp], %r0
    movw        %r0, #0x8
    stw         0[%sp], %r0
    ldw         %r1, 0[%sp]
    ldw         %r0, 4[%sp]
    cmpw        %r1, %r0
    jpdge       $BB0_2
    jpda        $BB0_1
$BB0_1:
    ldw         %r0, 0[%sp]
    addw        %r0, #0x1
    stw         0[%sp], %r0
$BB0_2:
```

```
        ldw       %r0, 0[%sp]
        addw      %sp, #0x8
        jpra      %r14
        .set      macro
        .set      reorder
        .end      test
$tmp0:
        .size     test, ($tmp0)-test
```

Toolchain Output

```
00000000 <test>:
0:   44 f3          mpushw   %r4,     %r4
2:   4f ab          movw     %r4,     %r15
4:   f3 27          subw     %r15,    #8
6:   08 af 07 00    movw     %r0,     #0x7
a:   00 00
c:   04 97 f8 ff    stw      -8[%r4], %r0
10:  03 af          movw     %r0,     #8
12:  04 97 fc ff    stw      -4[%r4], %r0
16:  14 87 fc ff    ldw      %r1,     -4[%r4]
1a:  04 87 f8 ff    ldw      %r0,     -8[%r4]
1e:  10 a3          cmpw     %r1,     %r0
20:  06 d8          jpdge    2c<test+0x2c>
22:  04 87 fc ff    ldw      %r0,     -4[%r4]
26:  06 17          addw     %r0,     #1
28:  04 97 fc ff    stw      -4[%r4], %r0
2c:  04 87 fc ff    ldw      %r0,     -4[%r4]
30:  f3 17          addw     %r15,    #8
32:  44 f7          mpopw    %r4,     %r4
34:  0e ed          jpra     %r14
```

In the LLVM outputs basic block BB0_0, first four movw and stw instructions are used to initialize the varaibles h and i. The value of i is copied in R1 and value if h in R0(ldw %r1, 0[%sp] and ldw %r0, 4[%sp]). Then the value of R1 and R0 (i.e i and h)is compared (cmpw %r1,%r0). If this condition is satisfied (i.e. i≥h), control goes to basic block $BB0_2 as indicated by instruction jpdge $BB0_2.

In basic block BB0_2, value of i is loaded in register R0 (by ldw instruction) and then control returns. If the above condition is not satisfied (i.e. i<h) control goes to basic block $BB0_1. In BB0_1, value of i is loaded in register R0 (by ldw instruction), increment by 1 is performed by addw instruction and incremented value is stored(by stw)instruction.

Next, control goes to BB0_2 where value of i is loaded in register R0 (by ldw instruction) and then control returns.

Test Case 8

```
int test()
{
    int a=0;
    int i = 0;
    while (i <7)
    {
     a++;
    }
    return a;
}
```

LLVM Output

```
test:
    .frame      %sp,8,%r14
    .mask      0x00000000,0
    .set       noreorder
    .set       nomacro
# BB#0:
```

```
        addw    %sp, #0xfffffffffffffff8
        movw    %r0, #0x0
        stw     4[%sp], %r0
        stw     0[%sp], %r0
$BB0_1:
        ldw     %r0, 0[%sp]
        cmpw    %r0, #0x7
        jpdge   $BB0_3
        jpda    $BB0_2
$BB0_2:
        ldw     %r0, 4[%sp]
        addw    %r0, #0x1
        stw     4[%sp], %r0
        jpda    $BB0_1
$BB0_3:
        ldw     %r0, 4[%sp]
        addw    %sp, #0x8
        jpra    %r14
        .set    macro
        .set    reorder
        .end    test
$tmp0:
        .size   test, ($tmp0)-test
```

Toolchain Output

```
00000000 <test>:
0:   44 f3         mpushw   %r4,    %r4
2:   4f ab         movw     %r4,    %r15
4:   f3 27         subw     %r15,   #8
6:   07 af         movw     %r0,    #0
8:   04 97 fc ff   stw      -4[%r4],   %r0
```

```
c:    07 af          movw    %r0,    #0
e:    04 97 f8 ff     stw     -8[%r4],    %r0
12:   06 dc          jpda    1e <test+0x1e>
14:   04 87 fc ff     ldw     %r0,    -4[%r4]
18:   06 17          addw    %r0,    #1
1a:   04 97 fc ff     stw     -4[%r4],    %r0
1e:   04 87 f8 ff     ldw     %r0,    -8[%r4]
22:   08 a7 07 00     cmpw    %r0,    #0x7
26:   00 00
28:   f6 db          jpdlt   14 <test+0x14>
2a:   04 87 fc ff     ldw     %r0,    -4[%r4]
2e:   f3 17          addw    %r15,    #8
30:   44 f7          mpopw   %r4,    %r4
32:   0e ed          jpra    %r14
```

In the LLVM outputs basic block BB0_0, value 0 is moved in register R0. The value of R0 is then stored at 2 respective positions to stack pointer (first two stw instructions) which initializes the variables a and i to 0.

In basic block BB0_1, value of R0 is compared with 6 since in the while loop condition is i<7. If this condition is satisfied (i.e. i>6), control goes to basic block BB3 as indicated by instruction jpdlt $BB0_3. In basic block BB0_3, value of a is loaded in register R0 (by ldw instruction) and then control returns.

If the above condition is not satisfied (i.e. i $\leq$6) control goes to basic block $BB0_2. In BB0_2, value of a is loaded in register R0 (by ldw instruction), increment by 1 is performed by addw instruction and incremented value is stored(by stw)instruction.

Next, control jumps to BB0_1 where again the condition is checked and this process continues until condition check fails.

Test Case 9

```
int sum(int x1, int x2,int x3)
{
```

```
    int sum =x3-x2+x1;
    return sum;
}
```

LLVM Output

```
sum:
    .frame    %sp,16,%r14
    .mask     0x00000000,0
    .set      noreorder
    .set      nomacro
# BB#0:
    addw      %sp, #0xfffffffffffffff0
    stw       12[%sp], %r0
    stw       8[%sp], %r1
    stw       4[%sp], %r2
    ldw       %r0, 8[%sp]
    subw      %r2, %r0
    ldw       %r0, 12[%sp]
    addw      %r2, %r0
    stw       0[%sp], %r2
    movw      %r0, %r2
    addw      %sp, #0x10
    jpra      %r14
    .set      macro
    .set      reorder
    .end      sum
$tmp0:
    .size     sum, ($tmp0)-sum
```

Toolchain Output

```
00000000 <sum>:
```

```
0:   44 f3        mpushw   %r4,   %r4
2:   4f ab        movw     %r4,   %r15
4:   f2 27        subw     %r15,  #16
6:   04 97 f8 ff  stw      -8[%r4],   %r0
a:   14 97 f4 ff  stw      -12[%r4],  %r1
e:   24 97 f0 ff  stw      -16[%r4],  %r2
12:  14 87 f0 ff  ldw      %r1,   -16[%r4]
16:  08 af f4 ff  movw     %r0,   #0xFFFFFFF4
1a:  ff ff
1c:  04 13        addw     %r0,   %r4
1e:  10 2b        subw     %r1,   [%r0]
20:  0c af        movw     %r0,   #-8
22:  04 13        addw     %r0,   %r4
24:  00 8b        ldw      %r0,   [%r0]
26:  01 13        addw     %r0,   %r1
28:  04 97 fc ff  stw      -4[%r4],   %r0
2c:  04 87 fc ff  ldw      %r0,   -4[%r4]
30:  f2 17        addw     %r15,  #16
32:  44 f7        mpopw    %r4,   %r4
34:  0e ed        jpra     %r14
```

In LLVM outputs basic block BB#0, the incoming arguments x1, x2, x3 which are there in registers R0, R1 and R2 respectively are stored as indicated by the first 3 store (stw) instructions. The value of x2 is loaded in register R0 (ldw %r0, 8[%sp]) which is then subtracted from the value of x3 present in register R2(subw %r2, %r0).

After that, value of x1 is loaded in register R0 (ldw %r0, 12[%sp]) which is added to the value present in register R2(result of x2-x1). The final result is in register R2 which is moved to register R0 (movw %r0, %r2) since R0 is the return register and then control returns from the function.

Test Case 10

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

LLVM Output

```
foo:
        .frame      %sp,8,%r14
        .mask       0x00000000,0
        .set        noreorder
        .set        nomacro
# BB#0:
        addw        %sp, #0xfffffffffffffff8
        stw         4[%sp], %r0
        stw         0[%sp], %r1
        movw        %r0, #0x0
        stw         4[%sp], %r0
$BB0_1:
        ldw         %r0, 4[%sp]
        cmpw        %r0, #0x5
        jpdge       $BB0_4
        jpda        $BB0_2
$BB0_2:
        ldw         %r0, 0[%sp]
        addw        %r0, #0x1
        stw         0[%sp], %r0
# BB#3:
        ldw         %r0, 4[%sp]
        addw        %r0, #0x1
```

```
    stw      4[%sp], %r0
    jpda     $BB0_1
$BB0_4:
    ldw      %r0, 0[%sp]
    addw     %sp, #0x8
    jpra     %r14
    .set     macro
    .set     reorder
    .end     foo
$tmp0:
    .size    foo, ($tmp0)-foo
```

Toolchain Output

```
00000000 <foo>:
0:   44 f3          mpushw   %r4,    %r4
2:   4f ab          movw     %r4,    %r15
4:   f8 27 0c 00    subw     %r15,   #0xC
8:   00 00
a:   04 97 f8 ff    stw      -8[%r4],    %r0
e:   14 97 f4 ff    stw      -12[%r4],   %r1
12:  07 af          movw     %r0,    #0
14:  04 97 fc ff    stw      -4[%r4],    %r0
18:  0b dc          jpda     2e <foo+0x2e>
1a:  04 87 f4 ff    ldw      %r0,    -12[%r4]
1e:  06 17          addw     %r0,    #1
20:  04 97 f4 ff    stw      -12[%r4],   %r0
24:  04 87 fc ff    ldw      %r0,    -4[%r4]
28:  06 17          addw     %r0,    #1
2a:  04 97 fc ff    stw      -4[%r4],    %r0
2e:  04 87 fc ff    ldw      %r0,    -4[%r4]
32:  08 a7 05 00    cmpw     %r0,    #0x5
```

```
36:   00 00
38:   f1 db          jpdlt    1a <foo+0x1a>
3a:   04 87 f4 ff    ldw      %r0,    -12[%r4]
3e:   f8 17 0c 00    addw     %r15,   #0xC
42:   00 00
44:   44 f7          mpopw    %r4,    %r4
46:   0e ed          jpra     %r14
```

In the LLVM outputs Basic block BB0_0, the arguments i and a are stored as indicated by the two stw operations, then value 0 is moved in register R0.Next stw instruction says to store the value of R0 i.e. 0 at the place where I was stored first since in for loop I is initialized with value 0.

In next basic block BB0_1, value of R0 is compared with 4 since in the for loop condition is i<5. If this condition is satisfied (i.e. i>4), control goes to basic block 4 BB4 as indicated by instruction jpdge $BB0_4.

In basic block BB0_4, value of a is loaded in register R0 (by ldw instruction) and then control returns. If the above condition is not satisfied (i.e. $i \leq 4$) control goes to basic block $BB0_2.

In BB0_2, value of a is loaded in register R0 (by ldw instruction), increment by 1 is performed by addw instruction and incremented value is stored(by stw)instruction.

Next, value of I is loaded in register R0 (ldw instruction), increment by 1 is performed since in for loop increment condition says i++. This incremented value is stored (by stw instruction) and control jumps to BB0_1 where again the condition is checked and this process continues until condition check fails.

On comparing the assembly code generated by LLVM backend and Reisc toolchain, we have found that there is around 95% similarity in the results.

Chapter 8

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, it'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.

Bibliography

[1] The LLVM Instruction Set and Compilation Strategy.
http://llvm.org/pubs/2002-08-09-LLVMCompilationStrategy.pdf

[2] The LLVM Target-Independent Code Generator.
http://llvm.org/docs/CodeGenerator.html

[3] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization, Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.
http://llvm.cs.uiuc.edu

[4] TableGen Fundamentals.
http://llvm.org/docs/TableGen/index.html

[5] LLVM Language Reference Manual.
http://llvm.org/docs/LangRef.html

[6] The Architecture of Open Source Applications.
http://www.aosabook.org/en/llvm.html

[7] Writing an LLVM backend.
http://llvm.org/docs/WritingAnLLVMBackend.html/target-registration

[8] Writing an LLVM Compiler Backend.
https://root.cern.ch/svn/root/vendors/llvm/docs/WritingAnLLVMBackend.rst

[9] A deeper look into LLVM code generator.
http://eli.thegreenplace.net/2013/02/25/a-deeper-look-into-the-llvm-code-generator-part-1

[10] Life of an instruction in LLVM.

http://eli.thegreenplace.net/2012/11/24/life-of-an-instruction-in-llvm

[11] Creating an LLVM Backend for CPU0 architecture.

http://jonathan2251.github.io/lbd/

[12] ReISC ISA Document : version 4.3.3

[13] ReISC ABI Document : version 4.2