Building an LLVM backend for TriCore architecture

Kumail Ahmed

Supervisors:
Prof. Dr.-Ing. Wolfgang Kunz & M.Sc M. Ammar Ben Khadra

This thesis is presented as part of the requirements for the conferral of the degree:

M.Sc Electrical and Computer Engineering

University of Kaiserslautern
Department of Electrical and Computer Engineering

March 2016

Declaration

I, Kumail Ahmed, declare that this thesis submitted in fulfilment of the requirements for the conferral of the degree M.Sc Electrical and Computer Engineering, from the University of Kaiserslautern, is wholly my own work unless otherwise referenced or acknowledged. This document has not been submitted for qualifications at any other academic institution.

Kumail Ahmed
May 17, 2016

Abstract

A compiler is a program that converts high-level code into machine instructions for a specific architecture . One of the key component of a compiler is the backend that is responsible for analysis, optimization, and code-generation. It is the duty of the backend to transform intermediate representation into target-specific assembly.

This thesis details the complete process of designing and implementing a backend for TriCore architecture in Low-Level Virtual Machine (LLVM). It explains the different phases that are part of the back-end design. Furthermore, a thorough - but meaningful - explanation is also given for a significant tool called "tblgen" that plays a vital role in describing TriCore specific instructions.

Acknowledgments

My sincere gratitude to my supervisor M.Sc. M. Ammar Ben Khadra for his intial idea, valuable feedback, and support during the entire period of this thesis.

I am really thankful to Prof. Dr.-Ing. Wolfgang Kunz for his support by providing me an opportunity to work in the chair of Electronic Design Automation at the University of Kaiserslatuern.

I am most grateful to my parents for their belief in me and their constant prayers. My heartfelt thanks are extended to all my friends who kept me motivated throughout the thesis work.

Lastly, I must acknowledge the resourcefulness of the LLVM documentation and its forum that was always filled with interesting ideas and alternates whenever I was lost in the process.

Contents

Chapter 1

Introduction

The computational power of processors is doubling every eighteen months. The expanse of embedded systems spreads over all domains of applications - aerospace, medical, automotive, etc. With the ever increasing complexity of hardware and software, the focus on speed, performance is becoming more important. This has lead to development of new processor architectures that can cope with these ever-growing needs.

Compilers play an integral role at the point where the hardware meets the software. They have become a well established research domain for hardware research. Essentially, the job of a compiler is to convert a high-level programming source code into a target language in a reliable fashion. Compiler architecture is divided into three parts:

1. Front End : The job of the front end is to analyse the structure of the high-level source code and build an intermediate representation (IR) of the code. Checks for syntax and semantic errors are also performed in this phase.

2. Middle End : Performs optimizations on the intermediate representation.

3. Back End : The backend is responsible for generating architecture specific code from the intermediate representation provided by the middle end.

1.1 Thesis aim

The goal of this thesis is to develop a backend for a TriCore architecture using Low Level Virtual Machine (LLVM 3.7). LLVM provides a modular compiler infrastructure that provides this frontend/backend interface.

As mentioned before, the responsibility of the backend is to convert the target agnostic IR representation into system-dependent representation, and generate assembly code as a result. This conversion requires a lot of features that are mostly hidden from an application programmer. Some of these features include calling convention layout, memory layout, register allocation, instruction selection.

1.2 Thesis outline

This thesis is divided into five chapters. The first chapter is an introductory overview of the work. The second chapter gives an introduction about the TriCore architecture. Chapter 3 gives an introduction to the LLVM compiler infrastructure. Chapter 4 provides a description of the backend implementation. The thesis concludes with a conclusion and discusses some future works.

Chapter 2

TriCore Architecture

Infineon started the first generation of TriCore microprocessor in 1999 under the trademark AUDO (AUtomotive Unified-ProcessOr). Since 1999, the company has advanced the TriCore technology, and currently the 4th generation TriCore chip is sold under the trademark of AUDO MAX. TriCore is the first single-core 32-bit architecture that is optimized for real-time embedded systems [1]. TriCore unifies real-time responsiveness, computational power of a DSP, and high performance implementation of the RISC load-store architecture into a single core.

TriCore provides simplified instruction fetching as the entire architecture is represented in a 32-bit instruction format. In addition to these 32-bit instructions, there exists 16-bit version for more frequently used instructions. These instruction can be used to reduce code size, memory overhead, system requirement, and power cost.

The real-time capability of the TriCore is defined by the fast context switching time and low latency. The interrupt latency is minimized by avoiding long multi-cycle instructions.This makes TriCore a wise choice for in a real-time application. TriCore also contains multiply-accumulate units that speed up DSP calculations.

This chapter describes the key components of the TriCore ISA that are essential in the understanding of the backend design.

2.1 Register description

TriCore consists of following registers:

1. 32 General Purpose Registers (GPRs)
2. Program Counter (PC)
3. Previous Context Information Register (PCXI)
4. Program Status Word (PSW)

The PC, PCXI, and PSW registers play an important role in storing and restoring of task context[2].

The 32 GPRs are divided into two types, i.e the so-called Address registers and Data registers. The Address registers are used for pointer arthimatics, while data registers are used for integral/floating type calculation. This peculiar Address/Data distinction creates a problem that would be discussed in chapter 4 in the section of calling convention implementation. The following table shows the registers and their special functions:

Data Registers
D15 (Implicit Data)
D14
D13
D12
D11
D10
D9
D8
D7
D6
D5
D4
D3
D2
D1
D0

Address Registers
A15 (Implicit Base Address)
A14
A13
A12
A11 (Return Address)
A10 (Stack Return)
A9 (Global Address Register)
A8 (Global Address Register)
A7
A5
A5
A4
A3
A2
A1 (Global Address Register)
A0 (Global Address Register)

System Registers
PC
PCXI
PSW

Table 2.1: TriCore registers

D15 and A15 are the implicit registers that are normally used by 16-bit instructions. The registers A0, A1, A8, and A9 are designated as global registers, and they are neither saved nor restored between function calls. By convention A0 and A1 register are reserved for compiler use and A8 and A9 are reserved for application usages. A11 holds the return address from jump and call instructions.

Finally, the two distinct colors show the two respective task context. Register A10-A15, D8-D15, PSW, and PCXI belong to the upper context, while registers A2-A7, and D0-D7 belong to the lower context. The upper context is automatically restored using the `RET` instruction. The lower context is not preserved automatically[3].

Moreover these GPRs can also combine in an "odd-even" pair to form a 64-bit register. There are no intrinsic real 64-bit registers in TriCore, hence for performing calculations that require 64-bit manipulation, an "extended register" is created by the "odd-even" combination. E0 is defined as [D1-D0], E2 is defined as [D3-D2], and so on. By convention, extended registers for the address type are named as P[0], P[2], and so on. Extended registers are used when multiplying large numbers or passing 64-bit arguments as a formal argument. More about this would be discussed in the calling convention section.

2.2 Supported data types

The TriCore Instruction set supports the following data types:

1. Boolean : mostly used in conditional jumps and logical instructions.
2. Bit String : produced using logical, and shift instructions.
3. Byte : an 8-bit value
4. Signed Fraction : comes in three variants 16-bit, 32-bit, and 64-bit. Mostly used in DSP instructions.
5. Address : a pointer value.
6. Signed and unsigned integers : a 32-bit value that can either be zero- or sign-extended. short signed and unsigned integers are sign-extended or zero-extended when loaded from memory to a register.
7. IEEE-754 Single precision Floating-point number

Specific coprocessors implementation are required for floating point numbers and long long integers manipulation, as hardware support for such operations is not provided with the basic TriCore ISA.

The address is always a 32-bit unsigned value that points to a memory address. In C parlance, it is simply a pointer variable. Hence, if the following code is executed, the `ptr` variable would always be stored in an address type register, and always be a positive integer.

```c
int a = 10;
int *ptr = &a; // ptr holds the address of variable a
```

Listing 2.1: Pointer example

2.3 Data formats, and alignment

The 32-bit TriCore registers can be loaded as a byte, a half-word, a word or as a double-word. The particular load/store instructions define whether the value is loaded/stored as a sign extended number or a zero extended number. For example, calling LD.W loads a word with sign extension and LD.WU loads a word with zero extension.

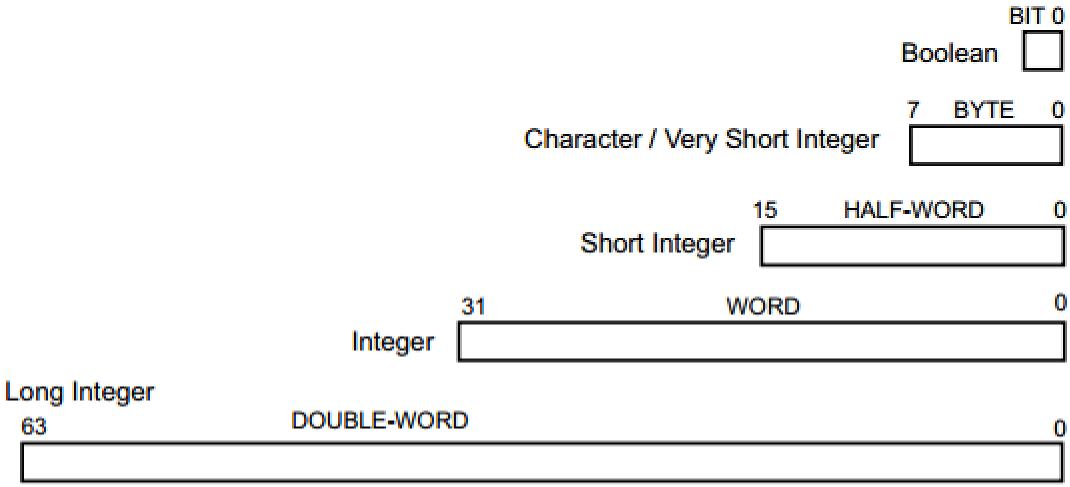

Figure 2.1: Supported data formats in TriCore. Adapted from [3]

TriCore uses little endian byte ordering. Byte ordering is performed either at 1,2 or 4 byte boundaries. The following table lists the alignment information for the primitive data types in C.

Data Type	**Size**	**Alignment**
char	1	1
short	2	2
int	4	4
long	4	4
long long	8	4
float	4	4

Table 2.2: Alignment information for primitive data types

2.4 Instruction Set

TriCore instruction set is divided into two formats: 16-bit opcode format and 32-bit opcode format. These formats are then further divided into different types depending on the type of data type modifiers. In total, there are 12 sub-formats for the 16-bit opcodes and 25 sub-formats for the 32-bit instructions. These two type formats can be distinguished on the basis of the zeroth bit. The zeroth bit for a 16-bit format is always zero, and always one for 32-bit format.

For example, one of the frequent occurring sub-format for a 32-bit opcode is called RC. The letter "R" means register, and "C" means a constant. Hence instructions such as a register and constant ADD would use an instruction that is of the type RC. Similarly, a register-register ADD would use a type RR sub-format. This distinction allows grouping common instructions together when writing them in the LLVM backend.

The 16-bit opcode format mostly supplements common instructions in the ISA. for example, eight variants exist for the ADD instruction in TriCore. These variants are characterised according to the range of constant value they can take as an input, the type of register they point to, and the opcode format. Consider the following example,

```
int a = 10;
a = a + 10;
```

Listing 2.2: Addition example

Assuming that constant folding is turned off, the backend can either generate a 32-bit instruction or a 16-bit instruction. Furthermore, it can also chose whether to put the result in an implicit register (i.e. D15) or any other data register. Considering these nuances would allow in the generation of a more efficient code in terms of size and speed.

TriCore provides several addressing mode for loading and storing data to and from memory. The most commonly used mode are :

1. Base + Short Offset (Format BO)
2. Base + Long Offset (Format BOL)
3. Pre/Post Increment (Format BO)

The addressing modes define the effective address for a load/store instruction, and then update the value of the base pointer. In this thesis, the first two modes are mostly used. Just like in the case of ADD instruction, Load and store instructions can also be divided into several types. Furthermore, TriCore defines a whole range of different loads and store depending on the primitive data type.

Opcode format for a jump instruction is characterised with an initial "B". Hence a jump that compares a register and a constant would be of the type BRC. On the same line, a register-register compare is of the type BRR. Generic jumps are in 32-bit opcode format, while jumps that compare with zero comes in 16-bit opcode format. Comparing to other RISC architectures like ARM, the jump instruction in TriCore performs comparison and jump in the same instruction. Hence, there is no need to use the common status register. In most cases the result for a jump instruction in written in the implicit register, but instruction also exist of for writing them into a register of a different type.

TriCore offers only a single shift instruction right and left shifts. To perform a left shift a positive shift value is given, while for a right shift, the value is passed in negative. The shift counter is a 6-bit signed number, hence allowing values from -32 to +31, allowing 31 bits shift to the left and 32 bits shift to the right.

2.5 Calling Convention

Calling convention define the process of how a sub-routine is called by the caller, and how the callee returns a value back to the caller. Calling convention for TriCore are defined in the TriCore EABI manual [3]. The calling convention schema between the caller and the callee is as follows:

- Non-pointer arguments are passed in register `D[8]` through `D[15]`.
- Pointer arguments are passed in register `A[4]` through `A[7]`.
- In case the registers are already filled, extra arguments are passed on the stack at the lower end of the caller's stack frame. In the callee, these arguments are place at zero offset from the stack pointer.
- 64-bit arguments are passed in register `E[4]` and `E[6]`.
- 32-bit values from the callee are passed to register `D[2]`, and values smaller than 32-bit are sign extended before being placed in register `D[2]`.
- Pointer values are passed to register `A[2]`.

Chapter 3

LLVM Structure

LLVM is a "collection of modular and reusable compiler and tool-chain technologies"[4]. More specifically, it a compiler infra-structure that provides a front-end (parser and lexer), optimizations, and a back-end (that converts the IR representation into machine code). LLVM also provides a range of optimizations including compile-time, link-time, and run-time optimizations. The project started at the university of Illinois by Chris Lattner, and has now grown into a huge open-source project including a vast range of tool-chain technologies such as JIT systems, debuggers, optimizers, interpreter, etc.

3.1 LLVM Design

A brief introduction of compiler design was given in chapter 1. A classical compiler is divided into a front-end, optimizer (middle-end), and a back-end. In the context of LLVM, a significant benefit of such a design is retargetability. LLVM uses the same optimizer for any arbitrary high-level language that has to be translated into machine code.

This technique considerably reduces the amount of code that has to be written when writing a new compiler from scratch. Hence, without retargetability, a compiler that support M programming languages, and N back-ends, M*N compilers ought to be written. LLVM reduces this job to M+N.

The three-phase design also offers encapsulation between the different parts of the compiler. This leads to another major benefit that different open-source communities can focus on specific parts of the tool-chain. The expertise required for designing a front-end are different from implementing a backend. This allows in the development of efficient optimizations in a smaller amount of time. In the case of proprietary compilers the efficiency is defined by the amount of budget that a company invests and the target market of the compiler designer. Many proprietary compilers exist including HP's aC++ and Intel's ICC among the most popular.

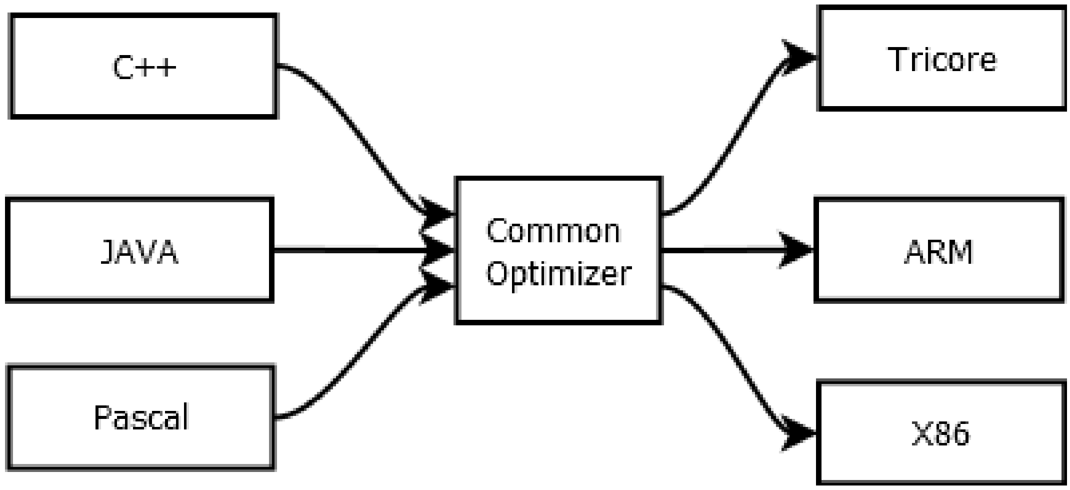

Figure 3.1: LLVM retargetability

3.1.1 Frontend

LLVM was initially written as a code generator for GCC. As the project evolved, new frontends starts to emerge. Currently, LLVM supports a wide variety of high-level languages including C, C++, Pascal, Ada, Fortan, Objective-C, and Swift. The most poupular frontend used with LLVM is called clang. It is also developed as an opensource project and support byte code generation for C, C++, Objective-C, and Objectice-C++. clang has the following advantages:

1. More readable diagnostic messages
2. Compatible with GCC, and lower memory usage.
3. Supports a wide variety of clients including services like static code analysis, and code refactoring.

Another known frontend was llvm-gcc. It was a modified version of the original GCC compiler that uses LLVM as its backend and had support for C and Onjective-C language. This project is now deprecated with DragonEgg that is attached with GCC 4.5 and above as plugin. The DragonEgg extension also support languages like Ada and Fortran, although results are poor for Ada on newer GCC verison. Compared to clang, the diagnostic information from DragonEgg is poor, and only compiles a "reasonable amount" of Objectice-C++ code.

3.1.2 Backend

As mentioned before, the goal of the backend is to convert LLVM intermediate representation (IR) into target-specific machine code. Backend architecture includes three basic operations. Firstly, is the analysis phase that gathers information from the LLVM IR. Control flow graphs and data flow graphs are built during this phase. Next comes the optimization phase that deals with making the IP representation faster to process while not changing the functionality of a program in any case. Some common optimizations include dead code elimination, constant propagation, and loop transformations. Lastly, the optimzed IR is passed to the code-generator that creates a target-specific machine output.

LLVM 3.7 has back-end support for several architectures including AArch64, AMDGPU, ARM, BGP, Hexagon, MSP430, Mips, NVPTX, PowerPC, Space, SystemZ, and XCore. The latest release also has support for AVR, but it is in an experimental stage. Backend targets can be accessed from `llvm/lib/Target` folder.

3.2 LLVM Intermediate Representation

The most valuable aspect of LLVM is its intermediate representation(IR). LLVM intermediate representation is a type of human-readable assembly language yet powerful. It is designed to be a generic IR that on one hand can represent low-level machine information, and on the other hand encapsulate high-level language ideas. While the front-end and back-end are restricted by the constraints of the source language and target architecture respectively, an IR provides complete freedom to the optimizer both in terms of expressiveness and performance.

```
define void @foo(i32 %d) #0 {
entry:
  %d.addr = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  store i32 %d, i32* %d.addr, align 4
  store i32 230, i32* %a, align 4
  store i32 210, i32* %b, align 4
  %0 = load i32, i32* %a, align 4
  %1 = load i32, i32* %d.addr, align 4
  %add = add nsw i32 %0, %1
  store i32 %add, i32* %c, align 4
  ret void
  }

// C code for this IR
void foo(int d) {
  int a=230,b=210,c;
  c = a + d;
```

```
}
```

Listing 3.1: LLVM intermediate representation example

As can be seen from the listing, LLVM IR is a similar to a RISC instruction set. It had instruction for loading and storing from and to the the memory. All other operations are performed in virtual registers (%0, %1, %add). LLVM has a static single assignment (SSA) form, hence it requires that each virtual register is assigned exactly once. All instruction are in the three address form, where the left-hand operand defines the destination register, while on the right-hand side there are exactly two operands.

3.3 Target-Independent Code Generator

The target-independent code generator contains reusable LLVM components that provides the framework for generating machine code for specific target platforms. There are six major components of this framework:

1. `Abstract target description` defines a target at an abstract level. It contain information about the instructions and registers that a architecture might support.

2. `Classes for code generation` are abstract classes implemented in `/llvm/lib/CodeGen`. These are abstract classes that must be implemented for a specific target architecture.

3. `Classes for MC layer` define textual information that is inserted into a generated assembly. This information include block labels, constrain information, header and footer for assembly, etc.

4. `Classes for target-independent code-generation algorithms` define the different phases of the backend including register allocation, instruction selection, stack representation, etc.

5. Implementation of abstract target description for a specific target. Any concrete architecture (i.e. TriCore) must inherit from this abstract description.

6. Target independent implementation of Just-in-Time compiler (optional).

In the development of this thesis work, the MC layer classes were implemented both for assembly generation and object file generation. The exact details of the implementation are presented in the next chapter.

3.4 TableGen tool

Before writing a backend, the first prerequisite is to extract the ISA and ABI information present in TriCore documentation and write it for LLVM. The mechanism that LLVM uses to express this information is written into target description files. For writing this target description files LLVM uses a tool called TableGen.

TableGen is a domain specific language that is used for developing human readable record for instruction and register description files for a specific target. Tblgen greatly reduces the amount of work that has to be done in describing instructions by allowing code reuse. A backend engineer will describe the target that is to be implemented in .td files that will be translated into C++ code by LLVM. Henceforth, generated C++ can be used in describing the overall process of instruction selection and register allocation. Secondly, description files allows code maintainability and reliability due to its modular design.

Although TableGen has a defined semantic, not much information is available apart from [5]. The best source to understand its functionality is to read implementation performed in other target architectures.

TableGen operates by performing two distinct passes. A pass in llvm is defined as a piece of code that performs optimizations and transformations. The first pass is an expansion of a the templates defined by the back-end engineer. This expansion pass is target-independent, and is predominantly a huge switch-case structure inside the `build/lib/Target/TriCore` folder in the LLVM hierarchy. The second pass adds target dependent information and is packaged inside various .inc files for a specific backend.

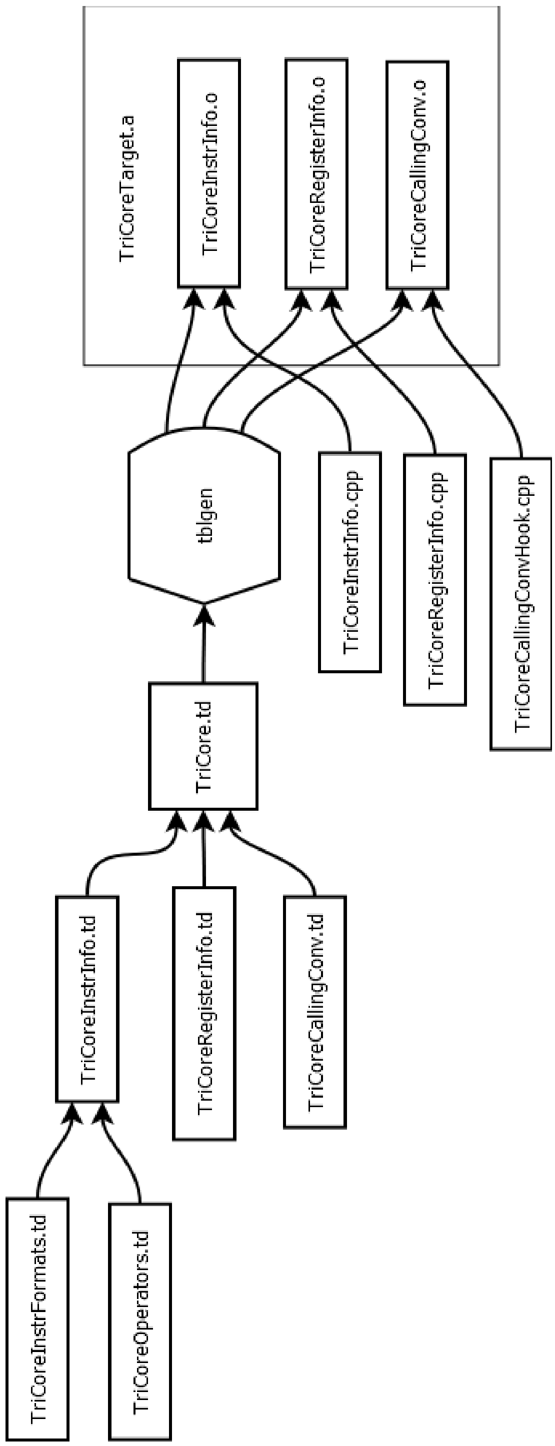

Figure 3.2: Class hierarchy for the table generator file systems. As a convention the name of the file starts with the name of the architecture. The files are grouped together according to their common functionality. In the fist pass, tblgen expands the macros. In the second pass target-specific information is incorporated with these exapnded marcos.

Chapter 4

Backend Design

In essence, the backend constitutes a number of passes that converts the LLVM intermediate representation into machine code. As mentioned in chapter 3, every backend has to inherit from a common set of abstract classes to perform target-specific code emission. A thorough discussion is given in [6]. At first, LLVM IR is passed to the backend. As the code generation process progresses forward, the IR is converted into TriCore-specific representation. In total, there are seven steps that are performed by LLVM. The diagram below gives an overview of the steps to output either an assembly or an object file as a final product.

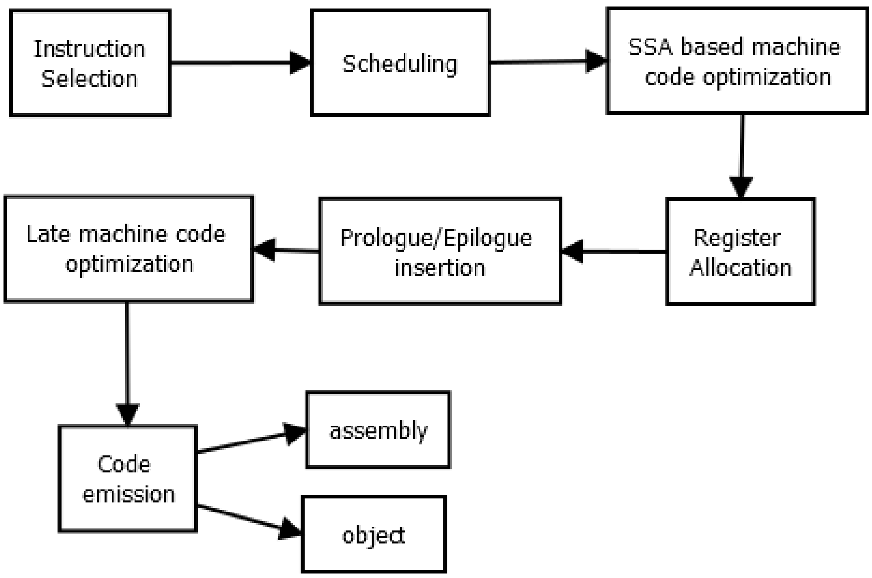

Figure 4.1: Steps in code generation

Each phase is called a pass in LLVM. The phases that are marked as `optimization` are crucial for increasing the code quality, while other passes are called `superpasses` that are made of up many smaller passes defined inside the LLVM framework. These superpasses are the most essential to be implemented in order to get a working back-end.

In this chapter, a detailed overview is given for the above mentioned steps and related information about the essential prerequisites.

4.1 Tabelgen Files

4.1.1 Instruction Pattern

Before we dive into the LLVM's code generation process, it is necessary to understand how an instruction is described in TableGen. TableGen essentially attaches a set of input and output DAGs (Direct Acyclic Graph) either to a LLVM defined pattern or a user-defined pattern. A DAG is a collection of vertexes and edges where an end node `Z` can never loop back to a start node `A`.

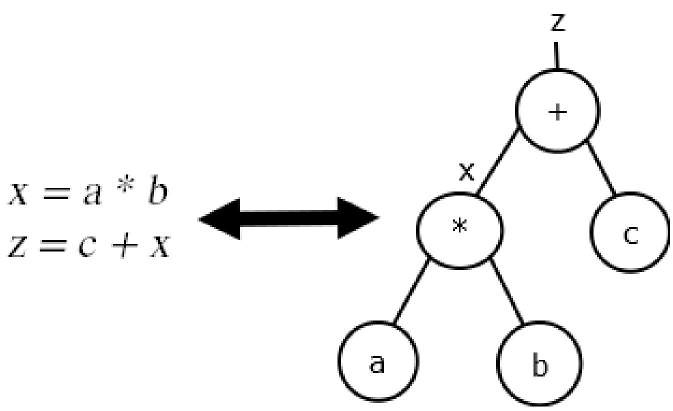

Figure 4.2: Example of a DAG

Similarly, TableGen uses DAGs to represent an instruction. For example, consider the pattern selection defined below for an `add` instruction in TriCore. It contains two different DAG, i.e. an output DAG defined by the keyword `outs`, and an input DAG defined by the `ins` keyword. Next the assembly string is supplied, that is printed once at the code emission stage. Lastly comes in the matching pattern defined inside the [ ] brackets. The pattern defined below assigns the output of the `add` node to the register $d.

```
def ADDrr : RR<0x0B, 0x00, (outs DataRegs:$d),
        (ins DataRegs:$s1, DataRegs:$s2),
        "add $d, $s1, $s2",
        [(set i32:$d, (add i32:$s1, i32:$s2))]>;
```

Listing 4.1: Pattern selection example

4.1.2 Register Classes

The `add` instruction mentioned above also defines an inherited register class. TriCore register information is present in register classes. The register class defines the data type, name, and optional dwarf information for a given register. As mentioned in chapter 2, TriCore has two different types of registers, i.e. the address registers and the data registers. Each type of register inherits from its respective class as described in the hierarchy in figure 4.3.

`TriCoreRegWithSubregs` class is used to define registers that hold 64 bit data values. As there is no true 64 bit register available in TriCore, hence two 32-bit registers have to be combined together. The process of doing this in TableGen is governed by the idea of sub-registers. A sub-register can either be the lower byte or the higher byte in the context of 64 bits. These sub-registers are later assigned to an extended registers by assigning individual sub-registers to an extended register.

```
let SubRegIndices = [subreg_even, subreg_odd] in {
  def E0 : TriCoreRegWithSubregs<0, "e0", [D0,D1] >, DwarfRegNum<[32]>;
  }
```

Listing 4.2: 64-bit register definition

In the example above, Extended register E0 is made up of two sub-registers, namely `subreg_even` and `subreg_odd`. `subreg_odd` register in this case is D1, while the even register is D0. Furthermore, D0 is the MSB (Most Significant Byte) while D1 defines the LSB (Least Significant Byte) for TriCore. The DwarfRegNum comes from the TriCore EABI information giving a unique debug code to each register.

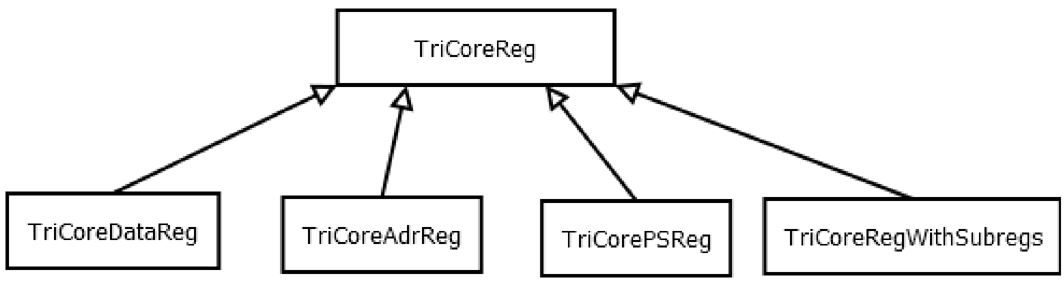

Figure 4.3: Register class hierarchy

4.1.3 Instruction Format

Continuing with the ADD instruction example above, The initial `RR<` represents the instruction format this instruction belongs to. An instruction format allows the representation of appropriate byte codes for a specific instruction, the output and input DAGS, the assembly string, and the pattern that has to be matched. For example consider an AND instruction.

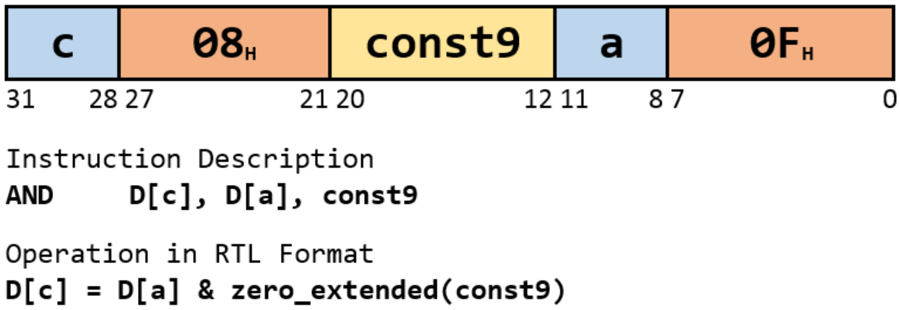

Figure 4.4: Instruction format for AND instruction

An instruction format is a direct representation of TriCore instruction type in TableGen language. This AND instruction takes in a data registers and a constant value and outputs the result into a third register. Furthermore, it defines two op-codes that have to be assigned in order to perform a bit-wise AND operation. An example in TableGen for such an instruction format is as follows:

```
class RC<bits<8> op1, bits<7> op2, dag outs, dag ins, string asmstr,
                list<dag> pattern> : T32<outs, ins, asmstr, pattern> {
  bits<4> s1;
  bits<4> d;
  bits<9> const9;

  let Inst{7-0} = op1;
  let Inst{11-8} = s1;
  let Inst{20-12} = const9;
  let Inst{27-21} = op2;
  let Inst{31-28} = d;
}
```

Listing 4.3: Instruction format for a RC-type instruction

Defining a class for each specific instruction format provides encapsulation and inheritance capabilities. This allows in writing more legible and efficient code for other instructions that inherit similar properties.

4.1.4 Representing Constants

Many instructions in TriCore take in constants as parameters. A constant is either a signed or a zero type value. A singed number, as its name suggest, can have both negative and positive value, while a zero type integer holds any value greater or equal to zero. In TableGen parlance, these properties are supplied using constraints. The instruction described in figure 4.4 takes a 9-bit constant as a parameter. Such an operator constraint is written as:

```
def immZExt9 : ImmLeaf<i32, [{return Imm == (Imm & 0x1ff);}]>;
def immSExt4 : PatLeaf<(imm), [{ return isInt<4>(N->getSExtValue()); }]>;
```

Listing 4.4: constant constraints

A 9-bit zero constant can have a value $0 \leq Imm \leq 2^9$. For a signed constant, the most significant byte is used to describe the sign. Hence, the range for a 4-bit number is $-2^3 \leq Imm \leq 2^3 - 1$.

4.1.5 Summary

An overall hierarchy of TableGen files is present in figure 3.2. The organization of TableGen files is as follows

- `TriCoreInstrInfo.td`,
  `TriCoreInstrFormats.td`: Contains information related to TriCore instructions and their formats
- `TriCoreOperators.td`: Contains information related to TriCore addressing and constant operator, parameter constraints, and user-defined TriCore nodes.
- `TriCoreRegisterInfo.td`: Contains information related to TriCore register description including type information, alignment, and debug information.
- `TriCoreCallingConv.td`: TriCore EABI information regarding how function is called, register allocation for formal parameters and return parameters.
- `TriCore.td`: Contains information about TriCore machine.

4.2 Code Generation Process

4.2.1 Instruction selection

During the instruction selection process an in-memory IR is converted into a TriCore SelectionDAG. The LLVM intermediate representation is converted in a tree of DAGs (Direct Acyclic Graphs). An example of selection DAG tree is presented in figure 4.5.

Each DAG graph represent a particular function inside the LLVM IR, while a node in a DAG graph represent a single instruction. A wealth of information

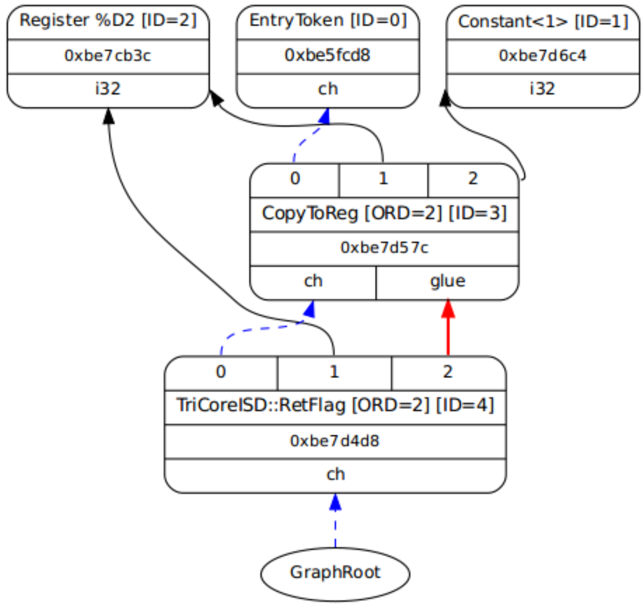

Figure 4.5: instruction selection DAG graph

is attached with each node. The black arrow show the data-dependency between instructions. Each node in graph either supply some values or read them from other nodes. A node is not always a TriCore specific data type. A node can be

- a concrete TriCore type that can either be an address register or data register. It also constains the type information related to a concrete type that can be i32, i64, or a pointer depending on the DAG process.
- a `other` type representing a chain value (marked ch in figure 4.5). These are abstract data type that are not allocated any memory in TriCore, but are used for showing a dependency between nodes.
- a `glue` type representing glue nodes, marked as glues in the figure. This again is an example of an abstract data type that is used for scheduling purposes.

Each selection DAG starts with an `EntryToken` that signifies the first operation in a Basic Block. This node has a value type of `Other`, hence allowing chaining to happen between nodes. At the end of the graph, there is also a chain node that marks the last instruction.

The blue arrow (called chain) represent the order in which the blocks are scheduled, i.e. which block will execute first and so on. In the example above, `CopytoReg` instruction will be executed before the return instruction at the graph root. The red arrow is called `glue` and represents that two nodes are glued together, i.e. two nodes must always be scheduled in the respective aforementioned order, and no other nodes can be scheduled between them.

As can be seen from figure 4.5, LLVM tries to add as much target-specific information as possible, but still the SelectionDAG remains target-agnostic for most of the part. `EntryToken`, `Constant`, and `CopyToReg` are target-agnostic nodes, while `Register %D2` and `TriCoreISD::Retflag` are examples of TriCore specific nodes. This step allows LLVM to apply the tree-based instruction selection algorithm during the instruction selection stage.

Selection DAG nodes have a defined semantics attached with each node. The semantics can be read from the LLVM documentation. Some of the nodes that can be seen in figure 4.5 are:

- `Register`: This can either be a physical register supported by TriCore or a virtual register generated by LLVM.
- `CopytoReg`: This node is used to copy a value from either a concrete type or abstract type. In return it chains itself with another node that accepts a `other` type value. One example of such a node is allocating variables on the stack. In the above example, a return value must be copied to register %D2 before the return instruction is executed. CopytoReg node ensures the correct flow of execution be chaining %D2 with TriCoreISD::Retflag.
- `CopyFromReg`: This node complements `CopytoReg` node. This node is often produced when copying values that are outside the scope of the current DAG instance. For example, consider a main() function calling another arbitrary function that accept some formal arguments. These formal arguments are not directly visible to this arbitrary function, and hence need to be copied from the DAG instance of main().

As mentioned before, instruction selection process is a `superpass`. Hence, it contain many smaller passes that are detailed in section 4.4.

4.2.2 Scheduling

The SelectionDAG does not really imply any optimized ordering between different DAG nodes. The first stage of instruction scheduling is called pre-register allocation scheduling. The scheduling phase takes in the SelectionDAG supplied from the instruction selection stage and tries to perform instruction ordering. Instruction ordering is performed with the idea of maximising instruction parallelism. After the instruction order is finalized, the SelectionDAG is erased and converted into LLVM `MachineInstr` instances.

This step helps in reducing the total register usage or the number of instructions that are generated in the the first phase. Scheduling happens by reducing the number of live registers to as low as possible. Consider the following IR:

```
%0 = add i32 1, i32 0
%1 = add i32 2, i32 0
store i32 %0, i32* %2, align 4
store i32 %b, i32* %2, align 4
```

Listing 4.5: without instruction re-ordering

In the above example, instruction re-order is yet not performed, hence virtual registers %0, %1, and %2 are live till the end of this basic block. LLVM must allocate 3 physical registers to each virtual register. Consider the same example with instruction re-ordering:

```
%0 = add i32 1, i32 0
store i32 %0, i32* %2, align 4
%1 = add i32 2, i32 0
store i32 %b, i32* %2, align 4
```

Listing 4.6: with instruction re-ordering

In the listing above, %0 is marked "dead" after the execution of the second instruction. This mean that the same physical register that LLVM allocated in the first instruction can be re-allocated for virtual register %1. As a result, only 2 physical register are used in the process.

Another example of pre-register allocation scheduling is the minimization of TriCore instructions. For example, TriCore's MADD instruction is a combination of a multiply and addition. A MADD instruction can reduce two instructions into a single instruction, and hence reduce total number of clock cycles.

4.2.3 SSA-based Machine Code Optimization

This is an optimization phase that is used to increase the efficiency of the generated code. SSA enforces that each register is only allocated once. Some example of SSA-based optimizations are constant folding, constant sub-expression elimination, constant propagation, and dead code elimination.

4.2.4 Register allocation

LLVM can allocate an infinite number of virtual register in its intermediate representation. These register are to be mapped accordingly to TriCore specific registers. This happens during this stage. LLVM tries to map all pointers to address registers, and data types to data registers. In case, register allocation is not possible, a spill is generated and reported.

Moreover, the second part of the scheduling, called post-register allocation scheduling happens here. As complete register allocation information is available and all virtual registers are removed, instruction re-ordering can be improved further.

4.2.5 Prologue/Epilogue insertion

Prologue and epilogue are similar to headers and footers for a function call. These are added to functions, in order for proper stack allocation. Details about this stage is given further in this report.

4.2.6 Late machine code optimization

Some final optimizations are performed in this stage. This include peephole optimization. Consider an example where a multiplication is to be performed by 4. One way of accomplishing this is to directly use the `mul` instruction, but such an instruction is really slow, hence reduces code efficiency. A much more efficient way would be to perform a 2 bit right shift. This would allow a further performance increase. Note that, as register allocation is already performed, peephole optimization can only occur if the alternate instruction uses the same number and type of registers.

4.2.7 Code emission

This stage finally emits the TriCore specific assembly files. At this stage LLVM converts the `MachineInstr` into `MCInst` instances. This new form is most suitable for either printing out the assembly file (.s) or object file (.o). All consideration were kept in writing the correct opcodes for each instruction in TriCore, the object file generated in this thesis is ready to be linked together with other object files.

4.3 Target Machine Description

4.3.1 DataLayout String

The DataLayout String provides the overall information about memory layout, data type alignment, and size of pointer in TriCore. It also defines the endianess of an architecture.

```
target datalayout =
"e-m:e-p:32:32-i1:8:32-i8:8:32-i16:16:32-i64:32-f32:32-f64:32-a:0:32-n32"
```

A small "e" at the start defines that the architecture is little endian. Next it defines the size of pointer that is 32 bit. Later on, is the description of different data types in TriCore, e.g. a char (i8) has a size of 8 bits and an alignment of 32 bits as mentioned in chapter 2. At the end of the string, "n" defines the alignment requirement for the stack that is 32 bits.

4.4 Instruction Selection Details

Instruction selection is a process of converting LLVM IR into Selection DAG Nodes (SDNodes in LLVM parlance). It is an example of a superpass, that is made up of three smaller passes. The passes involved in this phase include target lowering, combing DAGs, and finally legalization. Essentially, LLVM uses the TableGen instruction patterns to match the LLVM IR instruction to TriCore specific instructions. The legalization phase allows in removing target agnostic such as changing memory alignment, and type size including others, so that instructions and data can fit together in TriCore assembly.

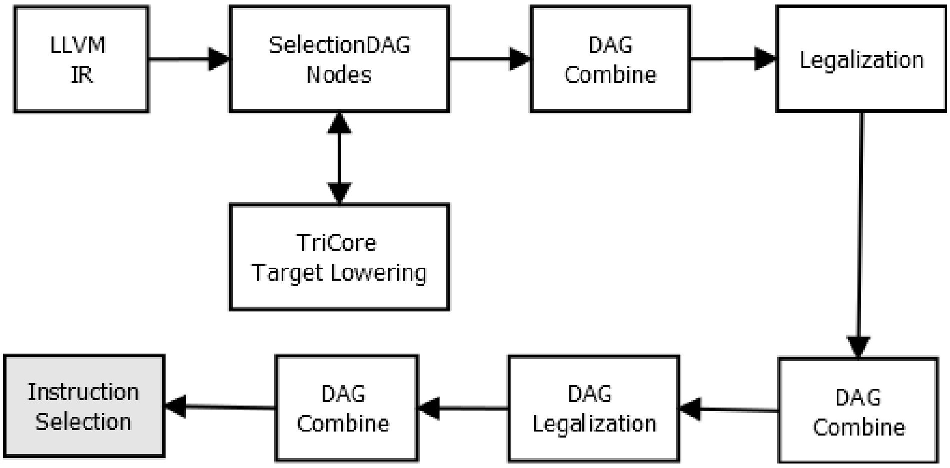

Figure 4.6: LLVM passes during instruction selection

4.4.1 Target Lowering

During the creation of a selection DAG graph, each LLVM IR instruction is lowered to TriCore specific instructions. There are many instructions that are common between different backends, e.g. addition and subtraction. Such nodes are automatically lowered into TriCore-specific instruction using Target Lowering functions defined in LLVM `TargetLowering` class. For other instructions, custom lowering routines need to be implemented. These lowering routines are written in `TriCoreISelLowering` class that inherits from LLVM's `TargetLowering` class. The

`TriCoreISelLowering` class overloads functions from the later, and provides an implementation for representing each node in TriCore-specific machine language.

To implement custom lowering we need to provide LLVM with the nodes that have to be lowered and also provide the respective operation it has to perform in such a case. listing 4.7 details the outline of the code required to imitate custom lowering. LLVM needs to know the name of the node that has to be lowered, and the associated operation that has to be performed for that related node. For example, the branch node in LLVM is called `ISD::BR_CC`. `setOperationAction` takes in three parameter, i.e. LLVM node name, machine value type, and the type of lowering that has to be performed. Whenever a branch node is read from the LLVM IR, TriCore runs the the associated function `LowerBR_CC(Op, DAG)` for this node. Similar operations are performed for all other custom lowering nodes include shift instructions, lowering global variables, ternary operators, and calling convention.

```
TriCoreTargetLowering::TriCoreTargetLowering(TriCoreTargetMachine
    &TriCoreTM)
    : TargetLowering(TriCoreTM), Subtarget(*TriCoreTM.getSubtargetImpl()) {
      .....
      // Nodes that require custom lowering
      setOperationAction(ISD::GlobalAddress, MVT::i32, Custom);
      setOperationAction(ISD::BR_CC, MVT::i32, Custom);
      setOperationAction(ISD::BR_CC, MVT::i64, Custom);
      setOperationAction(ISD::SELECT_CC, MVT::i32, Custom);
      ....
      }

SDValue TriCoreTargetLowering::LowerOperation(SDValue Op, SelectionDAG
    &DAG) const {
   switch (Op.getOpcode()) {
  default:              llvm_unreachable("Unimplemented operand");
  case ISD::GlobalAddress:    return LowerGlobalAddress(Op, DAG);
  case ISD::BR_CC:            return LowerBR_CC(Op, DAG);
  case ISD::SELECT_CC:        return LowerSELECT_CC(Op, DAG);
  case ISD::SETCC:            return LowerSETCC(Op, DAG);
 ...
  }
}
```

Listing 4.7: TriCore custom target lowering

4.4.1.1 Lowering Branches

Branch instructions in TriCore are represented by the type of branching that has to be performed followed by the branching operators and a displacement address where it has to jump in case of a successful jump.

```
JGE D[a], const4, disp15
JLT D[a], D[b], disp15
JNE D[a], const4, disp15
JEQ D[a], const4, disp15
```

Listing 4.8: Example branch instructions in TriCore

There are two noticeable peculiarities in TriCore in the context of implementing a branch. Firstly, TriCore does not have a status register where it can save the result of the comparison. Rather, it can use any possible data register for saving a comparison result. For instance, consider the following DAG for a branch instruction:

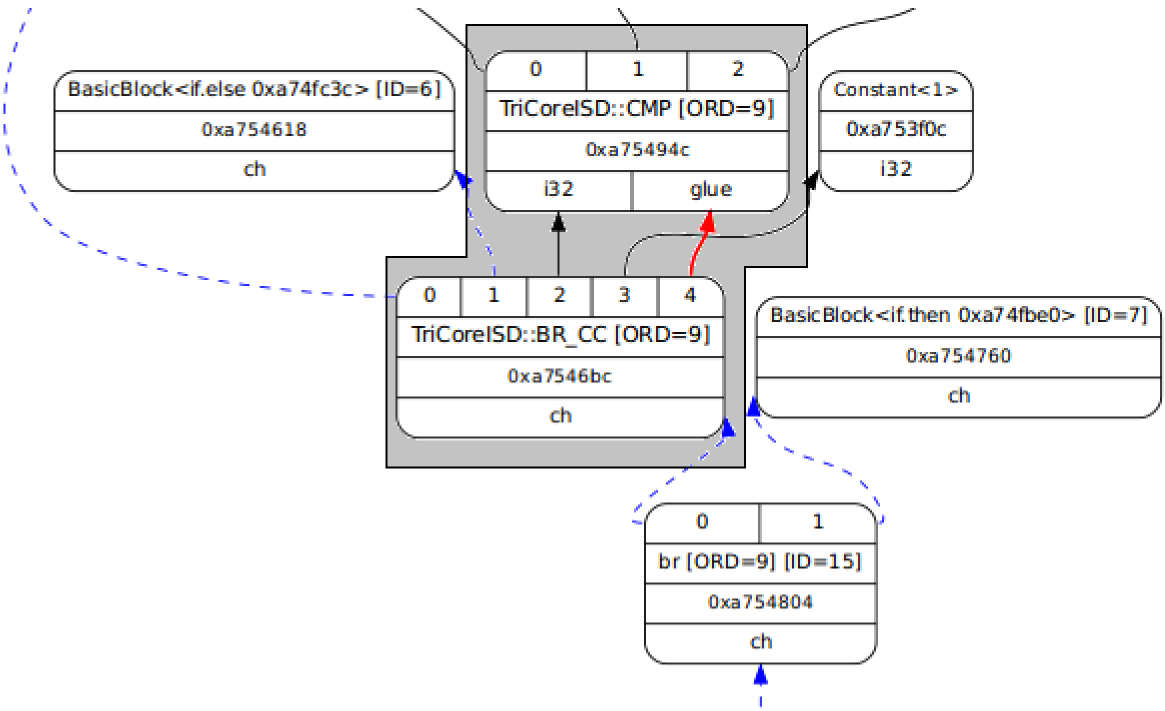

Figure 4.7: Selection DAG Node for a branch in TriCore. The grayed-area shows that the user-defined nodes added in order perform correct branching.

New user-defined nodes were created for answering this peculiarity. Whenever a jump is need to be performed, LLVM lowers the `br` instruction into a `TriCoreISD::CMP` and a `TriCoreISD::BR_CC` node. `TriCoreISD::CMP` performs a comparison for a set of given values that are passed as arguments. Here comes the second peculiarity; unlike many other ISAs, e.g. ARM, TriCore does not have

Original Condition	Adapted Condition
$a > const$	$a \geq const + 1$
$a > b$	$b < a$
$a \leq const$	$a < const + 1$
$a \leq b$	$b \geq a$

Table 4.1: Conversion of predicates not supported by TriCore

predicate conditions such as greater than ($>$) or less than equal ($\leq$). Hence, these conditions need to be adapted using the set of instructions that TriCore already have. The adaption is performed according to table 4.1.

`TriCoreISD::CMP` and `TriCoreISD::BR_CC` are connected with a `glue` node. This makes sure that these two nodes are always scheduled together. `TriCoreISD::BR_CC` is used in TableGen as shown in code listing below.

```
multiclass JUMP_16<bits<8> op1_sb, bits<8> op1_sbr,
                        string asmstring, PatLeaf PF>
  {
  def sbr: SBR<op1_sbr, (outs),
             (ins jmptarget:$disp4, DataRegs:$s1),
             !strconcat(asmstring, " $s1, $disp4"),
             [(TriCorebrcc bb:$disp4, DataRegs:$s1, PF)]>;
  }

  let isBranch = 1, isTerminator = 1 in {
      // Conditional branches
    defm JNZ : JUMP_16<0xEE, 0xF6, "jnz", TriCore_COND_NE>;
    defm JZ : JUMP_16<0x6E, 0x76, "jz", TriCore_COND_EQ>;
  } // isBranch, isTerminator
```

Points of consideration:

- `multiclass` is a TableGen keyword. It allows creation of multiple instructions records for a common instruction. In the above case, `JNZ` and `JZ` instructions inherit from `JUMP_16` class. `JUMP_16` takes in a opcode, an assembly string, and a `PatLeaf` as input arguments.

- `PatLeaf` is a TableGen keyword. It is similar to a `C++ #define` marco. In this example, `PatLeaf` is used for supplying predicates such as `TriCore_COND_NE` and `TriCore_COND_EQ`.

- `isBranch` and `isTerminator` properties allow instruction records to get these properties. These properties allow LLVM to chose the branch instructions from the pool of all TriCore instructions.

TriCore also provide explicit instructions for multiple predicate statements. These instructions include `and.lt`, `and.ge`, `or.lt`, and `or.ge`. They also have there unsigned versions that is shown by the suffix `.u`. These instructions allow in generating a smaller amout of code, and hence reduce the size of the output file.

Finally, 64-bit versions of comparisons are also implemented in the course of the thesis. For comparing a 64-bit register, the value is broken down into two separate registers and comparison is performed on each register separately. As shown in listing 4.9, the splitting takes place during the creation of `TriCoreISD::CMP` node. The lower byte is extracted by masking in with a 32-bit value, while the higher bit is results by performing a 32-bit right shift. From this point forward, the process is the same as 32-bit comparison.

```
ConstantSDNode *C = cast<ConstantSDNode>(RHS);
int64_t immVal = C->getSExtValue();
int32_t lowerByte = immVal & 0xffffffff;
int32_t HigherByte = (immVal >> 32);

RHSlo = DAG.getConstant(lowerByte, dl, MVT::i32);
RHShi = DAG.getConstant(HigherByte, dl, MVT::i32);
```

Listing 4.9: spiliting a 64-bit value before emitting a `TriCoreISD::CMP` node

4.4.1.2 Lowering Shifts

Shift instruction is also custom lowered for TriCore architecture. TriCore does not have separate instructions for right and left shifts. It rather uses the same instruction for performing both. For a logical shift, TriCore supplies `SH` instruction, while for an arithmetic shift `SHA` instruction is used. The distinction between a right and a left shift is made by the signed-ness attached with the amount of shift that has to be performed. To perform a left shift, the shift amount is taken as a positive value, while for a right shift it is converted into a negative value. This routine is implemented in `TriCoreTargetLowering::LowerShifts` function.

Shifts can be performed on 32-bit integer values and 64-bit long integers. A new user-defined node was created to take care for both of these cases. For performing 64-bit shifts, rather than programming it directly in `TriCoreTargetLowering.cpp`, pattern were written in TableGen. This allowed in reducing the line of code that has to written. A pattern in TableGen has the following syntax:

```
def :  Pat< (DAG to find in LLVM IR), (DAG to be replaced with) >
```

An example of 64-bit shifting in given in listing 4.10. The pattern implements a left shift for a 64-bit number. Whenever it finds a `shl` instruction in the LLVM IR, it replaces it with this pattern.

```
def : Pat<(shl ExtRegs:$src, (i32 imm0_31:$amt)),
         (INSERT_SUBREG( i64 (INSERT_SUBREG (i64 (IMPLICIT_DEF)),
                        (DEXTRrrpw (EXTRACT_SUBREG ExtRegs:$src,
                           subreg_odd),
                            (EXTRACT_SUBREG ExtRegs:$src, subreg_even),
                                        (i32 imm0_31:$amt)), subreg_odd)),
                                (SHrc (EXTRACT_SUBREG ExtRegs:$src,
                                   subreg_even),
                                          ( i32 imm0_31:$amt)),
                                             subreg_even)>;
```

Listing 4.10: TableGen pattern for performing a 64-bit shift

Points of consideration:

- `INSERT_SUBREG` keyword is used in the context of sub-registers. Extended registers in TriCore are made up of two sub-register. The first `INSERT_SUBREG` is used to write value to `subreg_even`, while the second `INSERT_SUBREG` writes to `subreg_odd`.
- `EXTRACT_SUBREG` keyword extract the even and odd parts of the extended register.
- `IMPLICIT_DEF` holds the current extended register that is in use.
- `DEXTRrrpw` and `imm0_31` are TriCore specific instruction and `PatLeaf` respectively. An explanation about these terms is given later.

`imm0_31` is a constraint that makes sure that the amount of shift being passed is inside the allowed range. TableGen is again used to write such a constraint (see listing 4.11). Constraints are essentially `PatLeaf` types. They taken a value and apply an assertion to it. If the assertion is true, this mean that a value value is passed. Hence `true` is generated as an output. An example of using a constraint is given in listing 4.10.

```
def imm0_31 : PatLeaf<(imm),
[{
  uint64_t val = N->getZExtValue();
  return val > 0 && val < 32;
```

```
}]>;
```

Listing 4.11: creating a pattern constraint

4.4.1.3 Lowering Ternary Operator

A ternary Operator in C/C++ is type of if-else condition. It is represented as

`test ? expression1 : expression2`

If the `test` returns a true, than `expression1` else `expression2` is executed. A ternary operator is modeled using the `SELECT_CC` node in LLVM. In LLVM IR, this node is identified with a `select` instruction. The ternary operator is different from the `BR_CC` node, because it does not have a jump address attached during in the LLVM IR. To perform a `select`, a custom insertor is used that can create a control flow for the respective branches that would be taken. The result for a `SELECT_CC` node are again created from a `TriCoreISD::CMP` node. The only difference in this case is that, basic blocks are added during the lowering stage.

```
let usesCustomInserter = 1 in {
  def Select8 : Pseudo<(outs DataRegs:$dst),
  (ins DataRegs:$src, DataRegs:$src2, i32imm:$cc, DataRegs:$src1 ),
  "# Select8 PSEUDO",
  [(set DataRegs:$dst,
              (TriCoreselectcc DataRegs:$src, DataRegs:$src2, imm:$cc,
                 DataRegs:$src1))]>;
}
```

Listing 4.12: Custom inserter pseudo-instruction for generating a ternary operator.

Listing 4.12 mentions the instruction record for generating a ternary operator. Such an operator is modeled on the same pattern of a branch condition. Point of consideration is the above listing are:

- `usesCustomInserter` is a TableGen keyword. Adding this property to the instruction record tells LLVM that this instruction would be exapanded during the lowering stage.
- `Pseudo` is a TableGen keyword. As the name suggests, it a instruction that has no legal meaning in TriCore assembly. This instruction must be morphed into a legal set of instructions that are already present in TriCore. Making this node `Pseudo` gives LLVM the ability to intercept such instruction at a later stage. A `Pseudo`instruction would in a general case are emitted into a more that a single instruction in the final code emission.

- `TriCoreselectcc` is a user-defined node that takes as input two data registers, the target condition, and condition flag generated from `TriCoreISD::CMP` node.

LLVM defines a virtual function named `EmitInstrWithCustomInserter` that is inherited by `TriCoreISelLowering` class. This function is executed, when a custom inserter instruction is intercept in the LLVM IR. In the course of this thesis, only ternary operator was implemented using a custom inserter.

4.4.2 Lowering Pseudo-instructions

Pseudo-instructions provide a mechanism to lower into more than one TriCore instruction. It is noted that this is not the only way to perform multiple instruction lowering as it can also be formed during DAG-to-DAG selection stage (see section 4.4.6). An example is adding 64-bit long integers with carry.

```
let Defs = [PSW] in {
def ADDi64C : Pseudo<(outs ExtRegs:$d),
        (ins ExtRegs:$s1, imml_32_h_32:$const64),
        "ADDi64C Pseudo",
        [(set ExtRegs:$d, (add ExtRegs:$s1, imml_32_h_32:$const64)),
         (implicit PSW)]>;
}
```

Listing 4.13: Example of a pseudo-instruction

A 64-bit register addition is performed by calling two instructions. Firstly, `ADDC` instruction is called that adds the lower bytes of the two extended registers being passed and also produce a carry flag. This carry flag is saved in TriCore's `PSW` register. The next instruction called is `ADDX` that adds the carry flag present in `PSW` register together with the higher byte of the extended registers. As a result, two instructions are outputted in the end. Implementation of pseduo-instructions in written in inside the inherited function `expandPostRAPseudo` in `TriCoreInstrInfo.cpp`. In the course of this thesis, many 64-bit related functions were implemented as pseduo-instructions. Such functions include logical operators as OR, AND, XOR and arithmetic operations such as subraction.

4.4.3 DAG Combine

The DAG combine pass optimizes the creation of SelectionDAG be introducing different yet similar node in a way that is more resource friendly by either reducing reducing the amount of DAG nodes that are to be generated. An example of DAG combination is constant folding. A DAG that adds 0 to a register value would be folded, and `add` DAG node would not be emitted. These sort of optimizations are mentioned in the file `DAGCombiner.cpp` that is included inside `TriCoreISelLowering.cpp`. As seen in 4.1 the DAG Combine pass executes twice; this is performed so that all suboptimal DAG nodes can be removed from the selectionDAG graph and as a final result allow in producing more precise and efficient code.

4.4.4 Legalization

Legalization pass deals with type legalization. Instruction selection can only take place on types that are legal for the TriCore architecture. For example, TriCore does not have a real 64-bit register, rather it combines two 32-bit reigster to form an extended register. Hence a value that is of type 64-bit is split into two 32-bit values. The legalization information comes from the register record written in `TriCoreRegisterIndo.td`. Each register class takes in size information to which different registers belong. for example, each register that belongs to a `DataRegs` class has a type of i32 attached with it (see listing 4.14). This information is passed during the initialization phase of `TriCoreIselLowering`.

```
def DataRegs : RegisterClass<"TriCore", [i32], 32, (add
      // Implict Data
      D15,
      // Mostly Used
      D2, D3,
      D4, D5, D6, D7,
      D8, D9, D10, D11,
      D12, D13, D14,
      // Others - Compiler Speicific
      D0, D1)>;
```

Listing 4.14: Register class for data registers

4.4.5 DAG legalization

DAG legalization is similar to type legalization that is mentioned above. This pass removes all operations that are illegal in TriCore architecture. For example, TriCore uses the same instruction for right or left shift as mentioned above. Hence, DAG nodes related to left and right shift are illegal in the context of TriCore. These nodes are lowered into TriCore-specific node during this process. DAG legalization is performed used the `setOperationAction` mentioned in listing 4.7. Other forms of DAG legalization are `Promote` and `Expand`.

4.4.6 DAG-to-DAG selection

DAG-to-DAG selection pass converts target agnostic nodes into target specific nodes by using pattern matching. For example, in figure 4.5, `CopytoReg` is a target agnostic node that is converted into a load instruction in TriCore. Pattern matching is performed by implementing the `Select` method in `TriCoreISelDAGToDAG.cpp`. `Select` method is at the crux of pattern matching. It can match instruction either by using a pattern generated using TableGen or, in case of a complex instruction, such a pattern can also be written in C++. An example of a complex instruction pattern is a pattern where an instruction emits more than or single TriCore instruction; another example of such a pattern is in a case where we set external predicate flags which can not be performed just be using TableGen.

```
SDNode *TriCoreDAGToDAGISel::Select(SDNode *N) {
  switch (N->getOpcode()) {
      case ISD::Constant:
       return SelectConstant(N);
      case ISD::FrameIndex:
          ....
      case ISD::STORE:
          ....
   }
  SDNode *ResNode = SelectCode(N);
  return ResNode;
}
```

Listing 4.15: Outline of `Select` method for pattern matching during DAG-to-DAG selection

As seen in listing 4.15, `Select` method takes in a `SDNode*`, then performs some operations on that node, and finally return it back. Essentially, `Select` is intended to call `SelectCode(N)` method that is generated through TableGen. During the process of generation, TableGen creates a map of LLVM `ISD` node and `TriCoreISD` with some instruction that is present in TriCore. This mapping information comes from `TriCoreInstrInfo.td` that contains relationships between individual TriCore instructions and LLVM.

For some cases, simply calling TableGen generated function does not work, and hence c++ routines have to be written in order to cope with thses cases. Once such example is presented in the implementation of `SelectConstant(N)` method.

4.4.6.1 SelectConstant method for pattern matching

As an example case where C++ implementation was expressive and allowed emission of complex instruction is in the implementation of `SelectConstant` method. In case `ISD::Constant` is intercepted during instruction selection, the TableGen generated map is bypassed and `SelectConstant` method is executed. This method implements an instruction named `imask` in TriCore ISA. This is a fairly complex instruction (see figure 4.8), and would require a pattern to be written in TableGen that contains many of predicate statements in order to implement it. Hence, a C++ implementation is given for it.

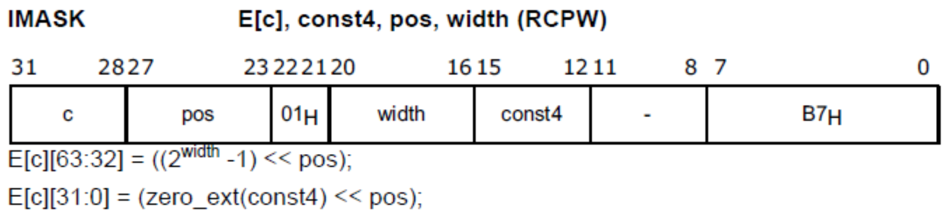

Figure 4.8: `imask` instruction details in TriCore. Taken from [7]

As `imask` does not work for all 64-bit values, so in most cases two instructions would be emitted for a 64-bit number. These two instructions are `MOVH` and `ADDI`. `MOVH` instruction sets the most significant byte of the extended register, and does not effect the lower byte, while `ADDI` adds a 32-bit constant value to the lower significant byte. Hence, generating a 64-bit number as a result. Furthermore, writing C++ implementation allows possibility for more optimization. For example, if a constant is `0xffffffff00000000`, there is no need to emit an `ADDI` instruction because the lower byte equals zero.

4.5 Calling Convention

Implementing calling convention is challenging in LLVM because of its internal nature of handling physical register. As mentioned in section 2.5, TriCore makes a distinction between address registers and data registers. Non-pointer data types are stored in data registers while pointer values are stored in address registers. Consider the following listing:

```
int a = 3 // The value is stored in data registers %D[n].
int* ptr_a = &a // The value is stored in address registers %A[n].
```

Listing 4.16: Physical register allocation for pointer and non-pointer types.

At first sight, this inherent register differentiation in TriCore seems impossible to be implemented simply using the TableGen files. Hence, changes were made in

LLVM code generation source.

```
def RetCC_TriCore : CallingConv<[
  CCIfType<[i8, i16], CCPromoteToType<i32>>, //line 1
  CCIfType<[i32], CCAssignToReg<[D2]>>,
  CCIfType<[i64], CCAssignToReg<[E2]>>
]>;

def CC_TriCore : CallingConv<[
  // ------  Further implementation in TriCoreCallingConventionHook.cpp
  CCIfType<[i8, i16], CCPromoteToType<i32>>,
  CCIfType<[i64], CCAssignToReg<[E4, E6]>>, //line 2

  CCIfType<[i32], CCAssignToStack<4, 4>>,
  CCIfType<[i64], CCAssignToStack<8, 4>>
]>;
```

Listing 4.17: TriCore calling convention implementation in TableGen

Initial outline of TriCore convention is presented in `TriCoreCallingConv.td` mentioned in listing 4.17. Points of consideration:

- `RetCC_TriCore` defines the convention for passing values back from the callee.
- `CCIfType` allows an action to be taken for data type given as a parameter. For example, according to TriCore calling convention, 8-bit and 16 bits must be sign-extended into 32 bit type before they are passed back into a register. `line 1` in the listing above mentions this idea in TableGen semantics. Whenever an i8 or i16 type is encountered it is promoted to an i32 type using the keyword `CCPromoteToType`.
- `CC_TriCore` defined the convention for passing values from the caller to the callee.
- `CCAssignToReg` assigns a caller's formal parameter to a pyhiscal register. For example, an i64 type (long int) is assigned to either register `E[4]` or `E[6]`.
- `CCAssignToStack` is used to define the size and offset for a data type. For example, i32 types are allocated a space of 4 bytes and also an offset of 4 bytes whenever the values are placed on the stack.

`TriCoreCallingConv.td` does not contain any method for handling pointer data types, as LLVM lowers i32** type into i32, hence there is no way to distinguish between a pointer type and an integer type. To answer this problem, another class was implemented that handles pointer arguments as well. The prime goal of this class named `TriCoreCallingConventionHook` is to differentiate between a pointer and a data type. An instance of this class is instantiated during the process of lowering formal arguments. This instance keeps track of all the arguments that are allocated to data or address registers. This distinction is performed using LLVM `MachineFunction` instance that holds the type definition of a variable. It can be noted that i64 type is assigned to a register using TableGen semantic as pointers are always 32-bit in TriCore. Hence, assignment is performed using the normal TableGen semantics. The information as to whether formal argument is present in an address register must also be passed inside LLVM DAG-to-DAG selector. This is necessary because TriCore has specific instructions that only operate on data registers and other instructions that only operator on address registers. The mechanism for transferring this information was to create an overloaded implementation of `getCopyFromReg` function. Several minor changes were made all along the selectionDAG creation process, so that correct TriCore instructions can be selected based on the register type. Furthermore, the concept of `Predicate` pattern was used to convey this information to the instruction records written in TableGen. Listing 4.18 presents an example of a store instruction that is generated when a value from an address register is to be saved in memory. A predicate is used to make sure if the operand is a data register or an address register. In case, of an address register, the `isPointer` predicate returns true. Hence, the appropriate TriCore instruction, `ST.A` is emitted. In case of a non-pointer data type, TriCore generates the instruciton `ST.D`.

```
def isPointer : Predicate<"isPointer() == true">;
...
let Predicates = [isPointer] in
      def STAbo : BO<0x89, 0x26, (outs), (ins AddrRegs:$d, memsrc:$memri),
      "st.a $memri, $d",
      [(store i32:$d, addr:$memri)]>;
```

Listing 4.18: Using `Predicate` patterns in TableGen

A similar address/data register distinction must also be made when returning value back from a callee to the caller. In TriCore, pointer addresses are stored in register A[2], while data type smaller than i32 are stored in register D[2]. This distinction was achieved by reading the type of the return value and hacking TableGen for pointer types. This hacking is performed in `LowerReturn` function, that bypasses TableGen assignment in case the return type of a function is a pointer.

4.6 Frame Lowering

Frame lowering deals with emitting prologue and epilogue sections for function calls. The prologue contains code that deals with setting up stack memory and registers for incoming arguments. It appears at the start of function. On the contrary, epilogue is added at the end of a function call, and restore the stack pointer back to its original value. In TriCore there is no need to add an epilogue because the `ret` instruction implicitly restores the upper context register values.

Function prologue is generated inside the `emitPrologue` function that is defined in the file `TriCoreFrameLowering.cpp`. It performs the following steps:

1. Compute the size of stack that needs to be designated for the function call.

2. Emit `MOVAA` instruction that copies the stack register (`A[10]`) into register `A[14]`.

3. In case the stack size is less than `0xffff` then perform step (a) else (b).

   (a) Subtract the amount of computed stack space needed from register `A[14]`.

   (b) Emit a set of move instructions that can handle stack sizes greater than `0xffff`.

4. Erase `ADJCALLSTACKUP` and `ADJCALLSTACKDOWN` pesudo-instructions that are added during the instruction selection process. This erasure is performed in the function `eliminateCallFramePseudoInstr`.

TriCore addressing is performed using TableGen `ComplexPattern`. A `ComplexPattern` allows implementation of a custom C++ function that is be called when an address node is emitted during the instruction selection process. Inside the custom C++ function, the target independent `ISD::FRAME` node is converted into a TriCore specific register+offset value that is emitting in the final assembly.

4.7 Code emission

This is the last stage of the back-end code generation process that takes place after all the virtual registers are allocated physical registers. As mentioned in listing 4.4.1.1, each TableGen instruction record has an assembly string attached along with its definition. These assembly strings are translated into assembly representation in this stage. The intial pass that runs in this process is the `AsmPrinter` pass. This pass calls the `TriCoreAsmPrinter` instances to call `EmitInstruction` function for each machine instruction. In case an assembly output is intended, the machine instruction instance is passed to an assembly streamer (`MCAsmStreamer`) or an object streamer (`MCObjectStreamer`). The process of printing out the instruction takes place in the `TriCoreInstPrinter`. This pass contain functions to output assembly related symbols for related assembly keyword such as register names and instruction name.

In case of a pseudo-instruction, an assembly expression is generated that is lowered into machine instructions in the `TriCoreMCInstLower`. An example of an assembly expression is the `load` instruction. Every such instruction generated a register+offset value. These values are combined into an expression later must be lowered before being printed.

Object file emission is implemented in `TriCoreMCCodeEmitter`. For instructions that are encoded at inside the instruction records, the object code emission works by reading the opcode and register parameters of the instruction. The binary-code is shifted likewise for each instruction as generated by TableGen.

In places, where an instruction is represented in terms of labels, relocation and fix-ups are needed with such values. Adding a relocation to an expression alerts LLVM that a future fix-up is needed. These fixup allow emitting the correct byte pattern during object emission. For example, in TriCore global address are fetched as 32-bit values. In order to enforce the correct byte-ordering two fix-ups, namely `fixup_tricore_mov_lo16_pcrel` and `fixup_tricore_mov_hi16_pcrel` are added to the respective bytes for a global address. These fix-ups are applied before the object code emission stage.

4.8 Extension in front-end

Changes were implemented into the front-end to include TriCore specific information related type sizes and type alignment. The most powerful advantage of the three-phase compiler design comes into play here. Due to "loose coupling" between the front-end and the back-end, different front-ends can be extended to the TriCore backend. In this thesis, the front-end of choice was `Clang`. The front-end information was integrated into LLVM in `TriCoreTargetInfo`. The related target information for the front-end, such as size and alignment, is implemented in `tools/Clang/lib/Basic/Targets.cpp`.

To choose the correct target at the time of code generation, `-target` flags needs `tricore-unknown-linux-gnu` triple in order to incorporate the correct type information for the machine. This triple is encoded in `llvm/Support/Triple.cpp` and `include/llvm/ADT/Triple.h`

Chapter 5

Conclusion

This chapter summarizes the goals achieved during the implementation of the thesis. It also contains the current short comings and concluding remarks related with future work.

5.1 Achieved Goals

A fairly working TriCore backend has been designed and implemented from scratch. The following goals are achieved:

- **Implementation of arithmetic operations**. These operations are work with data sizes including `i8`, `i16`, `32`, and `i64`. In cases, where is was possible, the 16-bit variants of the respective arthimatic operations were also implemented in order to reduce the absolute size of the generated file.
- **Implementation of logical operations**. All logical instructions were implemented to be supported for all data types. Moreover, instruction such as `NAND` was also implemented that cobine two logical operations together.
- **Implementation of shift operations**. Right and left shift requires special treatment in TriCore as discussed in the previous chapter. These instructions work for all data types. Furthermore, shifts also take into account the signedness of a value, that is taken care of using the `sext` and `zext` instructions.
- **Implementation of conditional jumps**. The peculiar nature of TriCore jumps, as mentioned in chapter 4, is functional and works for most data types. Furthermore, instructions that combine logical and jump arguments together were also implemented. An example of such an instruction is `AND.GE.U` that compared the logical `AND` of two values and then check whether it is greater or equal to a certain value. Such instruction reduce the total amount of branching

blocks that have to emitting in case logical and jumps were to be allowed in multiple instruction. Furthermore, support for ternary operator is also available.

- **Implementation of calling convention**. The current calling convention work for pointer and data types. It encompasses all the details that are mentioned in TriCore EABI manual as discussed in chapter 2 and chapter 4.

- **Implementation of instructions related to extended registers**. Instructions specific for extended-registers such as `imask`, `dextr`, etc. are also implemented. These instructions allow better pattern-matching and reduce the total number of instructions generated. Moreover, `dextr` allows type-casting for different variable types.

- **Optimization of move instructions**. Using a combination of tblgen instruction records and custom C++ routines during DAG-to-DAG selection, total number of move instructions have been reduced.

- **Global variables and arrays**. The backend allows lowering global variable data type and array types for all primitives.

- **Emission of Object code**. It is possible to view the object code dump using the gcc `objdump` command that allows a possibility of viewing the individual opcodes emitted.

5.2 Evaluation

Two criteria for measure the performance of a compiler are size and time for generating code. Comparisons were made with EDV's tricore-gcc compiler. All instructions mentioned in chapter 4 were verified individually and also in specific test patterns. Three different programs were evaluated:

- **Sorting**: A set of sorting algorithms that include qsort, selection sort, and insertion sort.

- **Factorial.c**: A program used to calculate the factorial of a number.

- **LinkedList.c**: A program to emulated the LinkedList data structure in C.

Tested were conducted on Windows and Linux on Intel Core i7. LLVM time has been broken down separately for the clang front-end and the back-end. As can be seen from the graph, the backend in some cases perform better than GCC. It was also observed that it in few cases, tricore-gcc opted to generate 32-bit format

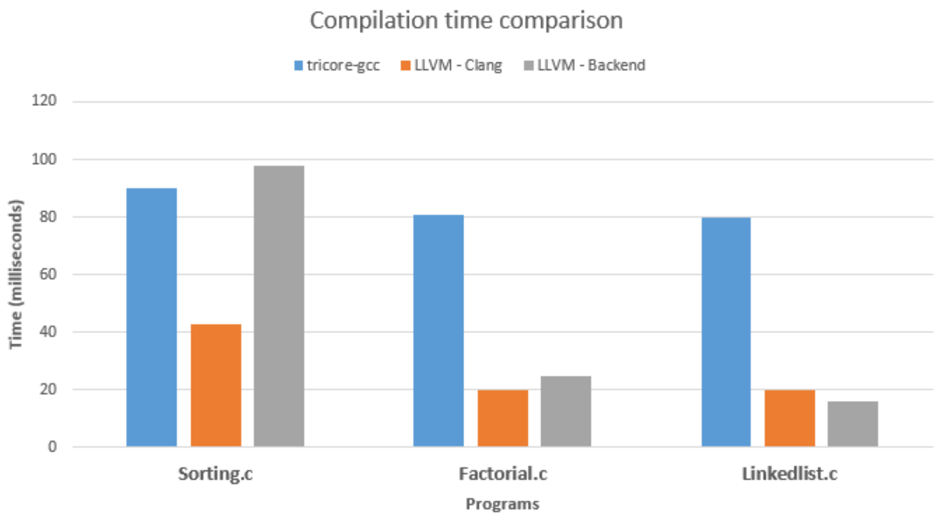

Figure 5.1: Compilation time comparison for tricore-gcc vs LLVM

instructions as compared to the smaller 16-bit formal instructions. Although the gains are not very obvious for small code, this difference might play a big role when the program size is big. Individual time for difference phases for tricore-gcc can not be measured as `-time` was not implemented.

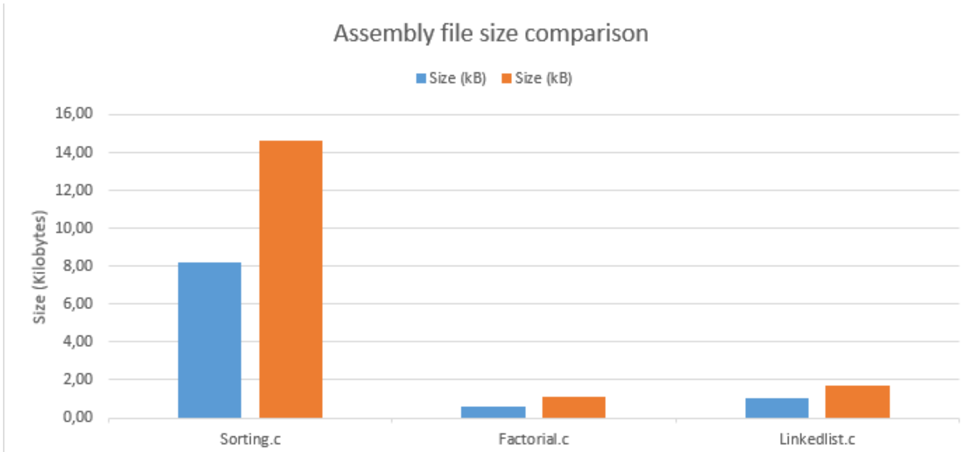

Figure 5.2: Sizes of generated assembly

The file size comparison shows that tricore-gcc produces more compact code compared to LLVM implementation. This is because more instructions are implemented for the former compared to later, hence allowing a wider choice of instruction selection.

This implementation and a similar implementation mentioned in [6] have a large pool of instructions that are common between both. The later implementation also has support for floating point numbers that this thesis lacks. As the source code for [6] was not available, it can not be definitely said to what extent 64-bit instructions are implemented. The current thesis implements a range of instructions related with extended registers as mentioned in Chapter 4. The current thesis also has support for compound data types such as `struct`, global variables, including others.

5.3 Future Work

TriCore implementation can be extended in a number of ways in the future.

- **Complete implementation of floating points**. Initial work regarding register alias has been written in the tblgen description and front-end related information is also implemented, but instruction record details need to be completed.
- **Writing a disassembler**. A disassembler is a program that translates machine-code into assembly code. As it is possible to generate object files and assembly files, it would be a fairly useful to write a disassembler for this project.
- **Implementation of DSP instructions**. It is difficult to write DSP-specific code in C hence a proper conversion from C to LLVM was not achieved.
- **Implementation of vector**. TriCore has support for SIMD vectors, but it was not possible to generate LLVM instructions that might match such types. As this was not an essential part of the thesis, it can be implemented in the future.

Bibliography

[1] *Tricore 1.3 32-bit unified processor core - ipcores*, Infineon Technologies AG, Munich, Germany, May 2002.

[2] *Tricore architecture volume1: Core architecture v1.3 & v1.3.1*, Infineon Technologies: TriCore Design Group, Bristol, UK, Jan. 2008.

[3] *Tricore architecture manual*, Infineon Technologies: TriCore Design Group, Bristol, UK, Mar. 2007.

[4] (May 2016). The llvm compiler infrastructure, [Online]. Available: `http://llvm.org/`.

[5] (May 2016). Tablegen fundamentals, [Online]. Available: `http://llvm.org/docs/TableGen/index.html`.

[6] C. Erhardt, *Design and Implementation of a TriCore Backend for the LLVM Compiler Framework.*

[7] *Tricore architecture volume 2: Instruction set, v1.3 & v1.3.1*, Infineon Technologies: TriCore Design Group, Bristol, UK, Jan. 2008.