Building an LLVM backend for TriCore architecture

Kumail Ahmed

Supervisors:
Prof. Dr. Wolfgang Kunz & M.Sc 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
April 14, 2016

Contents

1 Introduction **1**
1.1 Thesis aim 1
1.2 Thesis outline 2

2 TriCore Architecture **3**
2.1 Register description 3
2.2 Supported data types 5
2.3 Data formats, and alignment 5
2.4 Instruction Set 6

3 LLVM Structure **9**
3.1 LLVM Design 9
3.1.1 Frontend 10
3.1.2 Backend 10
3.2 LLVM Intermediate Representation 11
3.3 Target-Independent Code Generator 12

4 Backend Design **13**

Bibliography **14**

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. Currently, Tricore is the only single-core 32-bit architecture that is optimized for real-time embedded systems. 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 are also 16-bit instructions for more frequent usage. 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[1].

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	Address Registers	System Registers
D15 (Implicit Data)	A15 (Implicit Base Address)	PC
D14	A14	PCXI
D13	A13	PSW
D12	A12	
D11	A11 (Return Address)	
D10	A10 (Stack Return)	
D9	A9 (Global Address Register)	
D8	A8 (Global Address Register)	
D7	A7	
D6	A5	
D5	A5	
D4	A4	
D3	A3	
D2	A2	
D1	A1 (Global Address Register)	
D0	A0 (Global Address Register)	

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[2].

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 conditonal jumps and logical instructions.
2. Bit String : produced using logical, and shift instrucitons.
3. Byte : an 8-bit value
4. Signed Fraction : comes in three varients 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 unsinged integers are sign-extended or zero-extended when loaded from memory to a register.
7. IEEE-754 Single precision Floating-point number

Hardare support for IEEE-754 floating point numbers and long long integers in provided wit the basic TriCore ISA. Hence, a specific coprocessor implementation is required in order to extend the 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.

```
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.

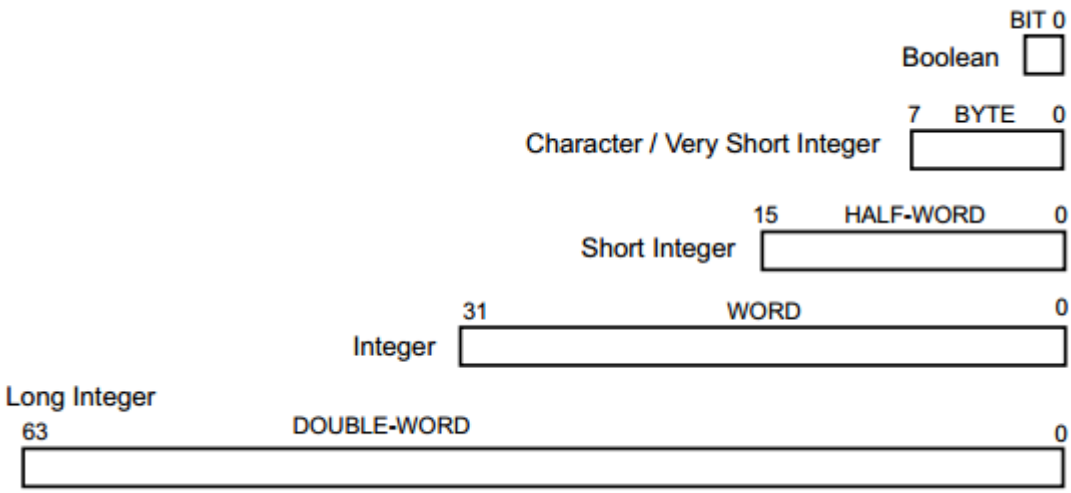

Figure 2.1: Supported data formats in TriCore. Adapted from [2]

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 elimination 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 shifting instruction For Arthimatic and logical shifting. The performing 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.

Chapter 3

LLVM Structure

LLVM is a "collection of modular and reusable compiler and tool-chain technologies"[3]. 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. An example of a proprietary compiler is HP aC++.

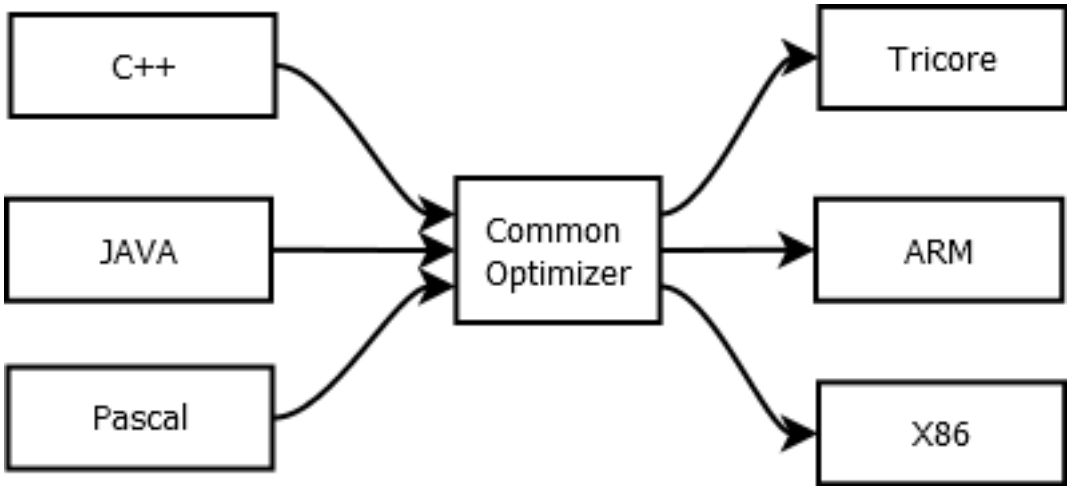

Figure 3.1: LLVM retargetability

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

LLVM 3.7 has code generation 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.

Chapter 4

Backend Design

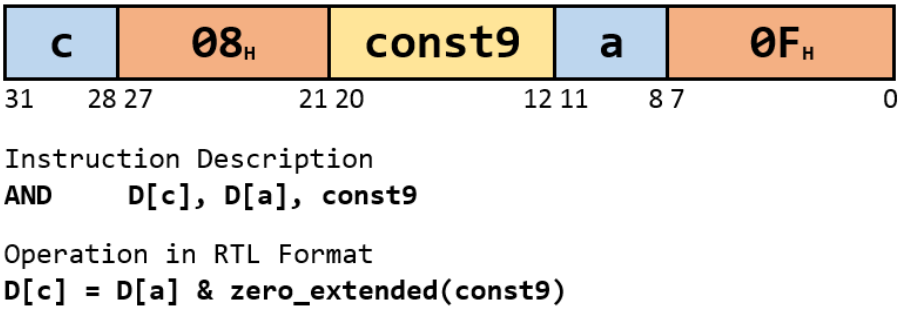

Figure 4.1: Instruction format for AND instruction

```
def : Pat<(i64 (sextloadi8 addr:$offset)),
       (INSERT_SUBREG (i64 (IMPLICIT_DEF)),
               (SHArc (EXTRACT_SUBREG (INSERT_SUBREG (i64
                  (IMPLICIT_DEF)) ,
                   (EXTRrrpw (LDBbo addr:$offset),
                        (i32 0), (i32 8)), subreg_even), subreg_even)
               , (i32 -31)), subreg_odd)>;
```

Listing 4.1: Pattern selection example

Bibliography

(1) *TriCore Architecture Volume1: Core Architecture V1.3 & V1.3.1*, Infineon Technologies: TriCore Design Group, Bristol, UK, Jan. 2008.

(2) *TriCore Architecture Manual*, Infineon Technologies: TriCore Design Group, Bristol, UK, Mar. 2007.

(3) *The LLVM Compiler Infrastructure*, May 2016, `http://llvm.org/`.