Implementation of a TriCore Backend in Low Level Vitrual Machine (LLVM)

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
March 29, 2016

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. 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 of 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 3 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 or 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 in when multiplying large numbers or when 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.

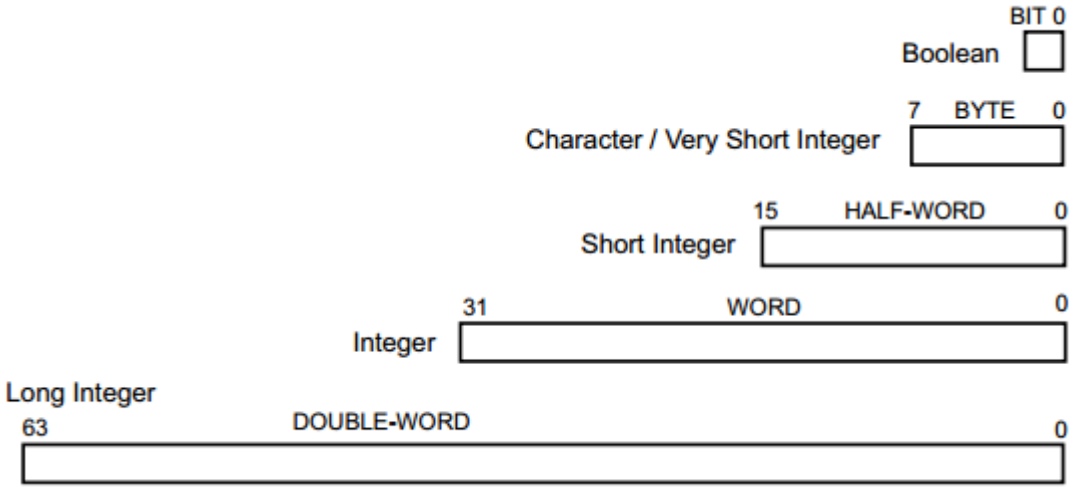

Figure 2.1: Supported data formats in TriCore. Adapted from [2]

Tricore uses little endian byte ordering and byte order is performed either at 1,2 or 4 byte boundaries. The following table lists the alignment information for the fundamental data type 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

Chapter 3

Backend Design

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.