Design and Implementation of a TriCore Backend for the LLVM Compiler Framework

Studienarbeit im Fach Informatik

vorgelegt von

Christoph Erhardt

geboren am 14.11.1984 in Kronach

Angefertigt am

Department Informatik
Lehrstuhl für Informatik 4 – Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:

Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat
Dipl.-Inf. Fabian Scheler

Beginn der Arbeit: 01. Dezember 2008
Ende der Arbeit: 01. September 2009

Hiermit versichere ich, dass ich die Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Stellen, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich durch Angabe der Quelle als Entlehnung kenntlich gemacht.

Erlangen, den 01. September 2009 _____

Überblick

Diese Arbeit beschreibt die Entwicklung und Implementierung eines neuen Backends für das LLVM-Compiler-Framework, um das Erzeugen von Maschinencode für die TriCore-Prozessorarchitektur zu ermöglichen.

Die TriCore-Architektur ist eine fortschrittliche Plattform für eingebettete Systeme, welche die Merkmale eines Mikrocontrollers, einer RISC-CPU und eines digitalen Signalprozessors auf sich vereint. Sie findet primär im Automobilbereich und anderen Echtzeitsystemen Verwendung und wird am Lehrstuhl für Verteilte Systeme und Betriebssysteme der Universität Erlangen-Nürnberg für diverse Forschungsprojekte eingesetzt.

Bei LLVM handelt es sich um eine moderne Compiler-Infrastruktur, bei deren Entwurf besonderer Wert auf Modularität und Erweiterbarkeit gelegt wurde. Aus diesem Grund findet LLVM zunehmend Verbreitung in den verschiedensten Projekten – von Codeanalyse-Werkzeugen über Just-in-time-Compiler bis hin zu vollständigen Allzweck-Systemcompilern.

Im Verlauf dieser Arbeit wird zunächst ein technischer Überblick über LLVM und die TriCore-Architektur gegeben. Anschließend werden Entwurf und Implementierung des Compiler-Backends im Detail beschrieben und ein Vergleich mit der bestehenden GCC-Portierung durchgeführt.

Abstract

This thesis describes the development and implementation of a new backend for the LLVM compiler framework that allows the generation of machine code for the TriCore processor architecture.

The TriCore architecture is an advanced platform for embedded systems which unites the features of a microcontroller, a RISC CPU and a digital signal processor. It finds use primarily in the automotive sector and other real-time systems and is employed by the Chair of Distributed Systems and Operating Systems at the University of Erlangen-Nuremberg for various research projects.

LLVM is a modern compiler infrastructure with a particular focus on modularity and extensibility. For this reason, LLVM is finding increasingly widespread use in most diverse projects – from code analysis tools via just-in-time code generators through to full-blown general-purpose system compilers.

In the course of the thesis, initially a technical overview of LLVM and the TriCore architecture is given. Subsequently, the design and implementation of the compiler backend are discussed in detail and a comparison to the existing GCC port is made.

Contents

1 Introduction

Today's world of computing is undergoing a constant, yet silent revolution. Both software and hardware systems – from embedded microcontrollers through to massively parallel high-performance computers – are experiencing an ever-increasing degree of sophistication and complexity. Although it never actually comes to the fore, compiler technology plays a central role in this revolution as the junction between software and hardware. Traditionally, a compiler has to fulfil three main requirements:

1. It has to generate efficient code for the target platform.
2. It should consume only a reasonable amount of time and memory (see Figure 1.1).
3. It must be reliable.

Over the last few years, a fourth feature has been gaining more and more importance: A compiler must be modular, maintainable, and easily extensible.

The GNU Compiler Collection, the de-facto standard system compiler for many Unix and Unix-like operating systems, supports a tremendous amount of frontends (for source languages) and backends (for target CPU architectures). However, due to its rather monolithic design, it is inherently difficult to extend and retarget [3]. LLVM offers a novel approach with an uncompromisingly modular architecture, which makes it less of a traditional compiler and more of a generic and versatile *compiler infrastructure*. As a consequence, LLVM is particularly interesting for research projects such as the Real-Time Systems Compiler (RTSC) developed at the University of Erlangen-Nuremberg, an operating system aware compiler for real-time applications [12].

The goal of this thesis was to implement and describe a new backend that generates assembly code for the TriCore processor architecture and can be employed by both the RTSC and the "vanilla" LLVM compilers. TriCore chips are primarily used in embedded real-time systems, for example in the automotive industry.

1.1 Existing Compilers for the TriCore Platform

The Infineon website lists two major compilers that support code generation for the TriCore architecture [19]: the *TASKING VX-toolset for TriCore*[1] and the *GNU TriCore Development Suite*[2]. While the former is proprietary closed-source software and thus not of peculiar interest for this thesis, the latter is based on the 3.3/3.4 branch of GCC.

[1] `http://tasking.com/products/32_bit/tricore/en/tricore.cfm`

[2] `http://hightec-rt.com/en/products/compiler-debugger/76-tricore-pxros-development-platform.html`

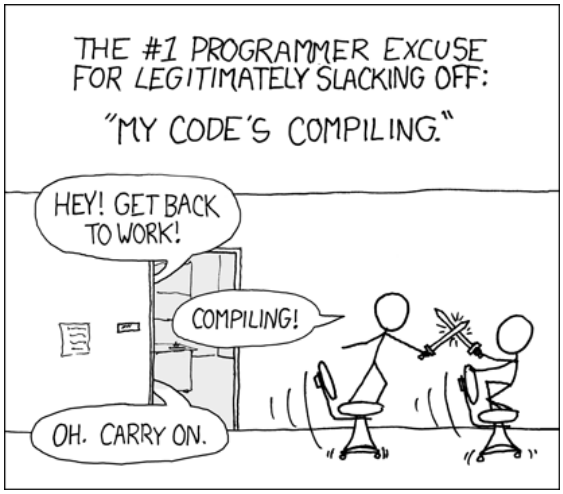

Figure 1.1: The reason why a compiler should not only generate efficient code, but also work efficiently itself [9].

Although the mainline GCC has been available in version 4 since 2005, the TriCore variant is still sticking to the "old" codebase, presumably because the porting effort would by far outweigh the actual benefits [2]. The LLVM backend was compared to this implementation.

1.2 Organization of this Thesis

The Chapters 2 and 3 of this thesis give a tight technical overview of LLVM and the TriCore architecture, highlighting the properties, features, and peculiarities relevant in the context of this work. The design and implementation of the compiler backend are discussed in extensive detail in Chapter 4.

Chapter 5 evaluates the implemented software and compares it to the existing TriCore port of GCC, Chapter 6 concludes the thesis.

2 The LLVM Compiler Infrastructure

The *Low-Level Virtual Machine* is a comprehensive compiler infrastructure designed from scratch to enable aggressive multi-stage optimization. It was started in 2000 by Chris Lattner at the University of Illinois at Urbana-Champaign. LLVM has increasingly gained prominence over the past few years, especially since 2005 when Apple Inc. hired Lattner and became the project's main sponsor. LLVM's code, written in C++, is released under a BSD-style licence, which makes it open-source software [10].

The name *Low-Level Virtual Machine* refers to the compiler's internal representation (IR) language, a RISC-like virtual instruction set. Despite its low-level nature, the representation carries extensive high-level type information, which enables it to perform advanced interprocedural optimizations. Translation of the intermediate code into assembly code takes place either statically at build-time, or dynamically via just-in-time compilation at run-time.

2.1 Application Examples

LLVM is beginning to find widespread use in both academic research and open-/closed-source projects. From version 10.5 on, Mac OS X employs LLVM's JIT capabilities in its OpenGL graphics stack, improving vertex shader performance. The Gallium 3D project, a graphics driver infrastructure for Linux and other operating systems, is currently working on incorporating LLVM for the same purpose. LLVM also plays a role in a number of implementations of the OpenCL framework (e. g., by NVIDIA).

Another interesting example is the IcedTea project, an extension to Sun's OpenJDK. *HotSpot*, the default Java Virtual Machine, only supports x86, x86-64 and SPARC and is very hard to port to other platforms. IcedTea is working on a custom LLVM-based just-in-time compiler called *Shark*. The goal is to make it possible to run JIT-compiled Java programs on all platforms for which LLVM has a JIT backend. The Mono project, a Free Software implementation of the .NET Framework, is also experimenting with LLVM code generation at present.

Apart from its dynamic code generation capabilities for special-purpose applications, LLVM is also increasingly finding use as a regular compiler for "traditional" programming languages such as C or C++. This is aided by the fact that LLVM was designed from the beginning to be used as a drop-in replacement for existing system compilers. In January 2009, the FreeBSD team announced that they were considering taking this step and working towards replacing the default GCC compiler [11].

2.2 Comparison to traditional Compilers

LLVM differs from "traditional" systems such as the GNU Compiler Collection in a number of aspects because it follows a fundamentally different design approach. Whereas GCC is primarily conceived as a part of the GNU toolchain, LLVM is not merely a compiler, but a modular and reusable *compiler infrastructure*. This is strongly reflected in LLVM's architecture, which has a library-based design and defines cleanly separated components. Consequently, on the one hand LLVM is easily extendible, and on the other hand it facilitates third-party applications incorporating specific parts of it[1], for instance the JIT components (see the application examples above).

The need for advanced inter-procedural optimizations often imposes difficult problems on existing compiler architectures: If the generated object files contain assembled machine code, the linker will have only very limited information and thus hardly be able to perform non-trivial optimizations. On the other hand, if the object files contain high-level intermediate code, most optimization passes – even the intra-procedural ones – will have to be deferred until link-time. Thus rebuilding a program will be extremely expensive, as the majority of the work will have to be performed by the linker.
LLVM's "multi-stage" approach tries to find the golden mean by translating each compilation unit into an IR language that is low-level enough to allow optimizations at compile-time while still providing sufficient high-level type information to the linker. At link-time, all input files are combined into a single unit, on which inter-procedural optimizations can be performed before the actual linking process takes place.

In general, LLVM puts a strong focus on efficient compilation. In spite of performing aggressive optimizations in order to yield highly efficient machine code, LLVM has proven to be many times faster than GCC while consuming significantly less memory.

2.3 Basic Architecture

Being a modular and retargetable compiler infrastructure, LLVM features a classic three-tier architecture as shown in Figure 2.1, consisting of a number of language-dependent frontends, one intermediate code optimizer, and a number of target-specific code generators.

The frontend receives a plaintext source code file as input and performs the typical three steps:

1. **Lexical analysis**: The input is broken into tokens. If required by the source language, these tokens are additionally preprocessed.

2. **Syntactic analysis**: The token sequence is parsed in order to identify the structure of the program, building a syntax tree.

3. **Semantic analysis**: The syntax tree is filled with semantic information (e. g., type information) and the symbol table is built.

[1] This is also reflected in the deliberate choice of a BSD-style licence. Generally, the developers' focus on pragmatism rather than politics is often cited as one major advantage (among others) over GCC.

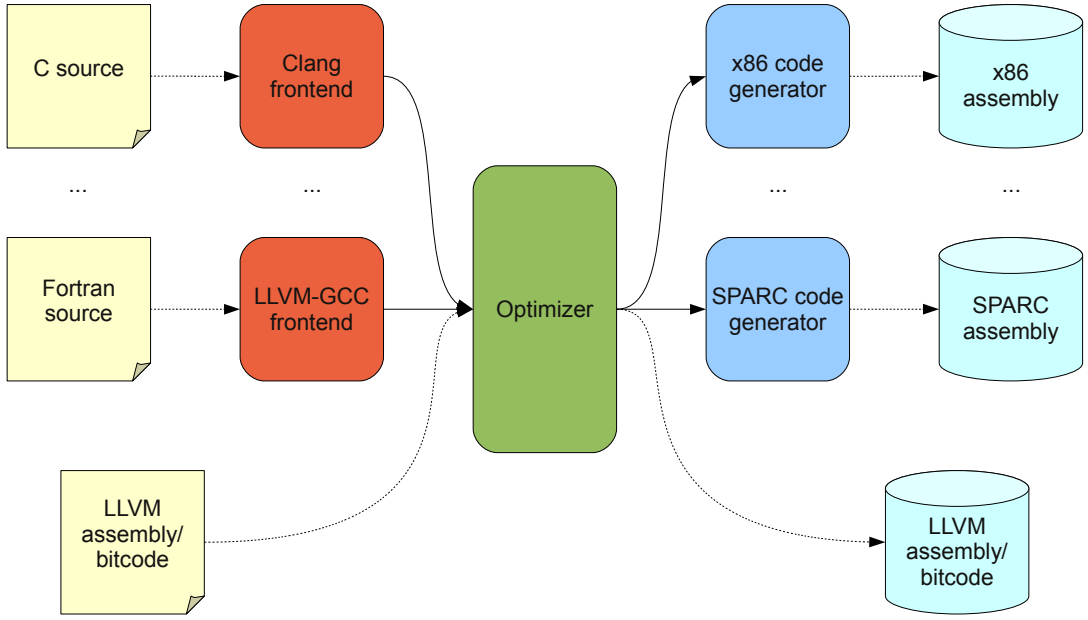

Figure 2.1: Basic LLVM architecture. The optimizer with its analysis and transformation passes is the core of the three-tier composition.

The resulting abstract syntax tree (AST) is still language-dependent and frontend-specific and has to be translated into the compiler's generic internal representation language.

Once the program has been converted into LLVM's intermediate code, this code is passed to the core of the compiler system, which is completely agnostic about the language of the original source code and the corresponding frontend. After performing a number of analyses (control flow, data flow, etc.), the collected information can be used by various optimization passes to apply transformations to the code.

Last of all, the backend corresponding to the selected target platform translates the IR code into the target-specific assembly language. This includes the mapping of virtual instructions to machine instructions and the allocation of variables to registers or to memory.

2.4 Internal Representation

LLVM's intermediate language consists of an instruction set and type system that both are completely independent of the language the source code is written in. This makes it possible to use it as a universal infrastructure for almost any compilation and optimization purpose – from low-level shaders to high-level C++ programs.

The internal type system supports the usual primitive and composite types present in low-level programming languages like C, plus some integer and floating-point vector types. For an overview, see Table 2.1. High-level types that are not supported directly

Data type	Width (in bits)
Integer	1, 8, 16, 32, 64, 128
IEEE-754 floating-point	32, 64, 80, 128
Boolean	target-dependent
Pointer	target-dependent
Integer vector	various
Floating-point vector	various
Array	N/A
Structure	N/A
Function	N/A

Table 2.1: Data types of LLVM's internal representation. Supported are mappings of all primitive and composite types of low-level programming languages like C, and SIMD (*single instruction, multiple data*) vector types.

must be mapped by the frontend. For instance, a C++ class with virtual methods can be represented as a `struct` with a number of function pointers.

It must be noted that – depending on the source language – LLVM programs are not guaranteed to be completely independent of the target machine if the language specification might be interpreted differently on different platforms. For example, the C type `long` may be mapped to the LLVM type `i32` or `i64` by the frontend, according to whether the code is being compiled for a 32-bit or a 64-bit target platform. Consequently, when building an application written in such a language for a different hardware architecture, it is advised to rebuild it from source.

The instruction set is orthogonal and specifies a virtual RISC-like register machine with an unlimited amount of registers. Instructions are mostly in three-address form. All LLVM programs are in *static single assignment* (SSA) form, i. e., for every use of a virtual register there is exactly one reaching definition and that definition strictly dominates the use. This makes data flow analysis significantly easier. The SSA-based optimizer supports three classes of optimizations:

- standard scalar optimizations (e. g., dead code elimination, constant propagation),
- aggressive loop transformations (e. g., unrolling), and
- advanced inter-procedural optimizations (e. g., inlining).

Figure 2.2 shows an LLVM code example for a function that computes the factorial of a given integer. Note the obvious segmentation into basic blocks, the use of explicit `br` instructions instead of implicit fall-through edges, and the SSA `phi` instruction where two reaching definitions from different paths meet.

"LLVM's virtual instruction set is a *first class language* which has a textual, binary, and in-memory representation" [7], so LLVM programs can be stored on disk either in a human-readable assembly-like text form (`.ll`) or as compressed binary code (`.bc`) files. Due to the project's extremely modular architecture, this makes it possible to use the middle-end as a standalone component.

```
define i32 @factorial(i32 %n) nounwind readnone {
entry:
  %cmp = icmp eq i32 %n, 0
  br i1 %cmp, label %return, label %if.else

if.else:
  %sub = add i32 %n, -1
  %cmp.i = icmp eq i32 %sub, 0
  br i1 %cmp.i, label %factorial.exit, label %if.else.i

if.else.i:
  %sub.i = add i32 %n, -2
  %call.i = tail call i32 @factorial(i32 %sub.i) nounwind
  %mul.i = mul i32 %call.i, %sub
  br label %factorial.exit

factorial.exit:
  %call3 = phi i32 [ %mul.i, %if.else.i ], [ 1, %if.else ]
  %mul = mul i32 %call3, %n
  ret i32 %mul

return:
  ret i32 1
}
```

Figure 2.2: LLVM code example. The code of the function `factorial` is in SSA form and computes the factorial of a given integer `n`.

2.5 Frontends

LLVM per se does not include any frontend for a source language. All of its tools require `.ll` or `.bc` files as input. However, there are two official frontends developed by the LLVM team that are available separately: *LLVM-GCC* and *Clang*. In addition, many third-party projects that employ LLVM technology have created frontends of their own. Examples include Java bytecode, Python, Haskell, and many more.

2.5.1 LLVM-GCC

As developing a complete frontend for a programming language is a cumbersome and non-trivial task, the LLVM developers decided to draw on GCC's set of existing frontends. LLVM-GCC is based on Apple's GCC 4.2 tree and implants LLVM into the GCC code base. The glue code translates IR programs from GCC's GIMPLE language into LLVM's language. This approach yields two big advantages:

1. At one stroke, LLVM suddenly supported the entire set of programming languages offered by GCC – including Ada, C, C++, Fortran, Java, Objective-C, and many more.

2. LLVM-GCC can be used as a drop-in replacement for GCC without any adaptations to the source code or build systems of existing applications[2].

The main disadvantage is the fact that the GCC frontend is rather slow, memory-hungry, and carries a lot of legacy code.

2.5.2 Clang

Clang is a relatively new effort to create a modern frontend for C, C++ and Objective-C from scratch. It aims to be fully language-conformant while providing compatibility to GCC's extensions. Clang has a range of benefits over LLVM-GCC:

- Diagnostics (errors/warnings) are much more descriptive and clear. Clang keeps track of all column numbers and for each diagnostic outputs the respective line of code with a marker placed below the responsible expression. Moreover, the original information of `typedef`s is retained, eliminating the problem of literally "drowning" in STL-related error messages.

- Clang adds a huge performance boost to the compilation process. Measurements have shown the frontend to be more than twice as fast as LLVM-GCC and to consume significantly less memory.

- Just like the rest of LLVM, Clang is designed to be modular, extensible and easily reusable (e. g., by source analysis tools or IDEs).

While the C and Objective-C parts are already in a reasonably mature state[3], the C++ part is currently still a work in progress.

2.6 Code Generation

One of the points of criticism at GCC is that retargeting is rather unpleasant and requires to more or less reinvent the wheel. This is where LLVM's modular design pays off once again. It provides a generic code generation framework that includes a set of reusable target-independent algorithms for register allocation, instruction scheduling, etc. This implicates a considerable simplification and unification of the retargeting process. Chapter 4 discusses this process and the code generation framework in detail.

The LLVM 2.5 release bundles code generators for the Alpha, ARM, Cell SPU, Itanium, MIPS, PIC16, PowerPC, SPARC, x86/x86-64, and XCore hardware architectures. The

[2] This is true provided that the application in question does not make use of any builtins / pragmas / inline assembly not (yet) implemented in LLVM.

[3] As of July 2009, Clang is able to compile the complete FreeBSD kernel and most of the userland. The Linux kernel however still has a few build issues.

Alpha, PowerPC and x86/x86-64 targets not only support static compilation, but also JIT code generation. LLVM additionally includes special backends for output of C/C++ source code and .NET CIL code.

2.7 Summary

LLVM is a modern compiler infrastructure with an exceptionally modular design. It puts a main focus on the generation of highly efficient code in a highly efficient way. As it tackles many of the shortcomings of traditional compilers and is fit for a broad range of use cases, it is forseeable that LLVM is going to rise in prominence and popularity over the next few years.

3 The TriCore Processor Architecture

TriCore is a microprocessor architecture developed by Siemens Semiconductors[1] and officially announced in 1997. Contrary to what the name might imply, TriCore is not a multi-core architecture. Instead, it aims at unifying the features of three worlds:

- a real-time microcontroller unit allowing fast context switches and low latencies,
- a powerful digital signal processor (DSP),
- and a superscalar RISC processor.

The TriCore is available as a so-called *intellectual property core*, a reusable building block that can be licenced for use as a component of a system-on-a-chip (SoC) solution. Such a solution would include the microprocessor core along with RAM/ROM memory as well as additional custom application-specific logic (ASIC), all integrated on a single chip [13].

The following chapter gives a brief overview of the basic architecture, as well as its peculiarities.

3.1 Fields of Application

TriCore is designed for a broad range of embedded system applications – primarily automotive control systems. BMW and several suppliers of automotive drivetrain components use chips from Infineon's AUDO family for engine and transmission control, allowing vehicle powertrains to fulfil considerable performance requirements while satisfying strict emission standards [17].

Another concrete example is a telematics and telecommunications platform developed by Infineon and Volkswagen in 2004. This unit is built around a TriCore microprocessor and offers wireless telephony and location-based information services (e. g., navigation) to the driver [15].

Apart from the various applications in the automotive industry, TriCore chips are also finding a use in several other embedded real-time systems. They are especially advantageous for systems that require both controlling functionality and complex digital signal processing, eliminating the need for two separate chips with distinct functionalities.

3.2 Architecture Overview

TriCore is a 32-bit superscalar RISC architecture. It operates on 32-bit words, providing 4 GiB of address space. Memory words larger than one byte are stored in little-endian

[1] since 1999 known as Infineon Technologies

byte order. Instructions have a (RISC-typical) fixed length of 32 bits – with the exception of some special variants that merely take 16 bits for encoding. This is because high code density has been considered a major design goal.

The TriCore architecture exists in two versions: the original *TriCore 1*, and *TriCore 2*, an overhauled release introduced in 2001 which is "essentially backwards compatible" [20] and brings a set of features and improvements mainly for operating system software. The following chapter as well as the implementation of the compiler backend restrict themselves to the *TriCore 1* architecture.

3.3 Registers and Data Types

A TriCore microprocessor is equipped with 32 general-purpose registers, each of them 32 bits wide – divided into 16 regular data registers D0–D15 (for integer/floating-point operands) and 16 address registers A0–A15 (for pointer operands). In comparison to common RISC architectures such as PowerPC or SPARC, this is an unusual design in two ways:

1. Normally the floating-point unit has its own register file, whereas TriCore's FPU accesses the ALU's data register file.

2. Although pointers are most commonly treated just like regular integers, TriCore makes a strict distinction between them, processing them in different register files with different instructions. This implicates that encoding a register operand only requires four bits instead of five. Most short instructions need two registers, so two additional bits can be used for the opcode, allowing four times as many different 16-bit instructions and ultimately providing a substantially higher code density [5].

All general-purpose registers can be freely used with the exception of four address registers: A10 and A11 represent the stack pointer and the return address, respectively; A0, A1, A8 and A9 are reserved for global addresses by convention. D15 and A15 are used as implicit (i. e., not explicitly encoded) operands by some 16-bit instructions.

Two successive 32-bit GPRs, starting with an even-numbered register, can be combined to form an extended 64-bit register (Ex for data registers, Px for address registers). For example E0 comprises D0 and D1; P12 comprises A12 and A13. While all regular arithmetic instructions such as ADD, SUB etc. only process 32-bit registers, some special instructions exist that make use of extended registers.

In addition to the general-purpose registers, there are three "Core Special Function Registers" that are implicitly read or modified by certain instructions: PCXI (previous context information), PSW (processor status word), and PC (program counter). For a complete overview of TriCore's architectural registers, refer to Table 3.1.

The ISA (*instruction set architecture*) supports a number of data types for instruction operands, as shown in Table 3.2. All basic arithmetic instructions operate on 32-bit words. Some variants of MUL (integer multiplication) produce 64-bit results that are put into an extended register. Other than that, computations on operands wider than 32 bits are not supported directly and have to be split into a series of computations on

Data registers:		Address registers:		System registers:
D15	E14	A15	P14	PCXI
D14		A14		PSW
D13	E12	A13	P12	PC
D12		A12		
D11	E10	A11 (ret. addr.)	P10	
D10		A10 (stack ptr.)		
D9	E8	A9 (global addr.)	P8	
D8		A8 (global addr.)		
D7	E6	A7	P5	
D6		A6		
D5	E4	A5	P4	
D4		A4		
D3	E2	A3	P2	
D2		A2		
D1	E0	A1 (global addr.)	P0	
D0		A0 (global addr.)		

Table 3.1: TriCore architectural registers. The TriCore platform has 16 data registers for 32-bit integer/floating-point numbers and 16 address registers for 32-bit pointers.

32-bit words. Characteristically for a RISC architecture, TriCore offers no dedicated instructions for integers smaller than the register width. Such operands have to be sign- or zero-extended to 32 bits upon loading from memory and truncated upon storing back. Booleans are treated as 32-bit integers, with `true` being encoded as 1 and `false` being encoded as 0.

A feature called *packed arithmetic* allows building a vector of four 8-bit words or two 16-bit words in a 32-bit register and performing a set of arithmetic SIMD (*single instruction, multiple data*) instructions on it, thus processing four respectively two words of data simultaneously in a single CPU cycle.

Signed fraction is data type intended specifically for digital signal processing applications. It represents a fixed-point fraction in the range $[-1; 1)$ and can be 16, 32 or 64 bits wide, with the highest-order bit denoting the sign. Real numbers are represented as single-precision (32-bit) IEEE-754 floating-point numbers. Double-precision floating-point numbers are not supported directly.

3.4 Addressing Modes

Being a "load-store" RISC architecture, TriCore's instruction set offers no support for using memory addresses directly as operands. Instead, the contents of a memory cell have to be loaded explicitly to a register and the result has to be stored back explicitly. Load and store instructions exist for 8-, 16- 32- and 64-bit memory words. The 8- and 16-bit load instructions both offer a *signed* variant that sign-extends the contents of the

Data type	Width (in bits)
Integer	32
Address	32
Signed fraction	16, 32, 64
IEEE-754 floating-point	32
packed byte vector	4 x 8
packed half-word vector	2 x 16
Bit String	1–32

Table 3.2: TriCore data types. In addition to the "regular" types supported by all common ALUs, TriCore offers specific types for digital signal processing like signed fractions.

memory location to fit into a 32-bit register, and an *unsigned* variant that zero-extends it.

The TriCore architecture supports the two most commonly used addressing modes, namely absolute addresses (ABS) and three variants of *base register + immediate offset* (BO):

- base + 10-bit offset
- base + 10-bit offset with pre-increment
- base + 10-bit offset with post-increment

Some instructions have an additional variant with a 16-bit offset (BOL).
Beyond that, there are two special addressing modes for "the efficient implementation of typical DSP data structures (circular buffers for filters and bit-reversed indexing for FFTs)" [13].

3.5 Instruction Set

As mentioned above, instructions are encoded either as 32-bit or as 16-bit ("short") words. Execution takes exactly one cycle in most cases, with some exceptions. 32-bit instructions generally have a three-address format, i. e., they have two source operands and one destination operand:[2] $dest \leftarrow src1 \otimes src2$.
Most short instructions are specialized variants of regular instructions, having a two-address format:[3] $accu \leftarrow accu \otimes src2$.

The destination operand is always a register. Source operands can either be registers or immediate values embedded in the instruction word. Regular 32-bit instructions such as ADD are available in the following two formats:

- RR: $reg \leftarrow reg \otimes reg$

[2] $\otimes$ denoting the operation
[3] with *accu* or *src*2 being encoded implicitly, thus saving sufficient bits to fit the opcode into two bytes

- RC: $reg \leftarrow reg \otimes imm$

Immediate operands typically take 9 bits of the instruction word and are sign-extended upon decoding. Immediates in 16-bit instructions are limited to 4 bits. Larger constant values are not supported directly and have to be explicitly loaded into a register.

Unlike common architectures, TriCore does not own a condition code flag register. Typically, these flags would be set either implicitly by any arithmetic operation or by issuing an explicit comparison instruction. Conditional branch instructions would then evaluate the contents of that register. TriCore takes a different approach by merging comparison and branching into a single instruction, e. g., `jeq` (*jump if equal*).

Floating-point numbers can be processed via dedicated instructions offered by an FPU that is available as a separate component. Instead of providing a register file of its own, the FPU operates on the CPU's regular data registers. The instructions offered comprise addition, subtraction, multiplication, division, comparison as well as casts from respectively to integer numbers and signed fractions. When a loss of precision is induced, the result is rounded according to one out of a set of rounding modes that can be configured at runtime. If an FPU is not present, IEEE-754 arithmetic has to be emulated in software.

Being designed specifically for digital signal processing applications, TriCore's ISA contains a number of DSP-oriented instructions, including:

- compound instructions such as MADD (multiply-add) or MSUB (multiply-subtract),
- arithmetic instructions with result saturation,
- bit extraction,
- and SIMD operations on packed data.

3.6 Tasks and Contexts

"A Task is an independent thread of control. There are two types of task: Software-Managed Tasks (SMTs) and Interrupt Service Routines (ISRs)." [13]
Several aspects of the task concept are implemented directly in hardware – a fact that is reflected in the ABI's calling conventions[4]. Each task has a privilege level and owns a so-called *context* that defines its state and consists of the data, address and system registers. The context is subdivided into two parts:

- The **upper context** consists of the registers D8–D15, A10–A15, PCXI, and PSW. It is automatically saved upon calls, interrupts or traps, and restored upon return.
- The **lower context** consists of the registers D0–D7, A2–A7, and PC. It has to be saved and restored explicitly if it is to be preserved across task switches.

Saved contexts are stored as linked lists in on-chip memory. All operations on this list are performed in hardware, guaranteeing complete transparency and low overhead for context loads/stores.

[4] See Chapter 4.5.1.

3.7 Summary

TriCore is a 32-bit platform for high-end embedded systems. While basically following a RISC approach, it offers a range of extended features for digital signal processing purposes and has a few peculiarities due to certain design requirements.

4 Design and Implementation of the Backend

As mentioned in Chapter 2, LLVM provides developers with an extensive generic framework for code generation. It prescribes a fixed directory layout and class hierarchy template. Each backend is located in its own subdirectory within `lib/Target`, where most of the code to be implemented goes. Apart from this, only a handful of the original files in the LLVM source tree have to be modified in order to integrate the new backend with the existing LLVM codebase.

The target code generator is divided into several components, each of which is discussed in a separate section. For a better overview, each section contains a list of the source files that comprise the respective component.

The given class hierarchy specifies a number of abstract base classes with virtual functions and requires the backend developer to implement subclasses for each of them. Most of these classes do not provide any immediate functionality for code generation, but merely give selected information about the characteristics and properties of the target machine. This makes it possible to keep the bulk of the actual algorithms and procedures fully target-independent by accessing all the required target-specific information through the specified interfaces.

A slightly simplified overview of the class hierarchy is shown in Figure 4.1. Pre-existing framework classes are coloured green, red denotes a manually implemented backend class, and yellow marks a class automatically generated by the `tblgen` tool, which is discussed in the following section.

4.1 TableGen

With the above technique, the focus on what the developer has to provide is shifted from imperative code towards extensive descriptive data. As C++ is obviously not the language of choice for that purpose, LLVM brings its own descriptive language for domain-specific modelling, called *TableGen*. TableGen "is specifically designed to allow writing flexible descriptions and for common features of these records to be factored out" [8]. Its syntax is relatively simple and intuitive to read, yet powerful enough to describe complex entities.

TableGen follows a somewhat object-oriented approach. A concrete entity of the target domain (e. g., a register, an instruction, or the like) is modelled as a record of data, defined as an instance of a class. Each TableGen class has a number of attributes and an optional list of template arguments. Inheritance plays a major role and is most often

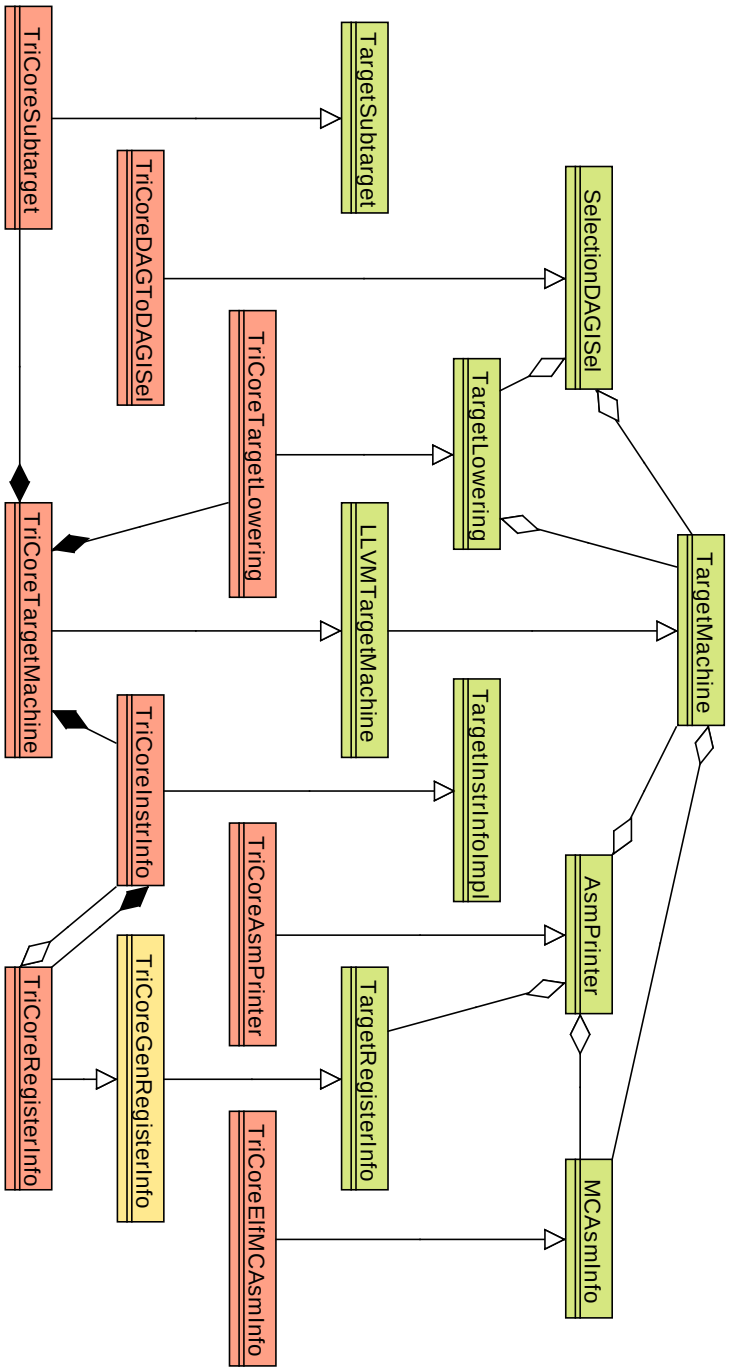

Figure 4.1: Backend class hierarchy. Green classes belong to the code generation framework, red classes are custom. `TriCoreGenRegisterInfo` is auto-generated.

used for factoring out common characteristics. A subclass transitively inherits all fields of its superclass and may add additional attributes.

TableGen descriptions are stored in `.td` files and are processed by the `tblgen` tool, which has a set of backends for numerous domains – mainly components of the code generator, including:

- subtargets,
- register files,
- calling conventions,
- and the instruction set.

For every TableGen backend, certain top-level superclasses have special pre-defined semantics (e. g., the `Register` class for the register description). The declaration of these classes can be found in the file `include/llvm/Target/Target.td`.

The output of `tblgen` is a C++ file that can be compiled along with the regular backend code. This process is integrated transparently with LLVM's build system. Depending on the kind of information represented, the resulting C++ code primarily consists of `enum`s, `struct`s, and arrays. One notable exception is TableGen's instruction selection backend, which produces a plethora of functions containing actual imperative code. For more extensive details about the use of TableGen descriptions in the code generator as well as concrete examples, refer to the corresponding sections later on in this chapter.

4.2 Code Generation Process

In order to understand the structure of the backend, it is necessary to understand how the code generator works. To translate LLVM code into machine code, the backend has to perform a sequence of steps [4], which are described hereafter. Just like all of LLVM's internal procedures, each phase consists of one or more passes. A pass is subclass of the interface `Pass` that represents exactly one analysis or transformation. The entirety of all passes to be executed during one compiler run is governed by the central `PassManager` instance.

Figure 4.2 illustrates the flow of an LLVM program through the code generator and depicts its in-memory representation after each step. An annotated class name indicates that this backend class directly manipulates the code during the respective phase.

4.2.1 Instruction Selection

The first step in the code generation procedure is to transform the LLVM code into a set of *SelectionDAG*s, a directed acyclic graph form with each node representing one instruction. Each definition–use relationship between two instructions is described by an edge from the using node to the defining node, along with information about the type of the affected variable. Additional "chain" edges are inserted for control flow dependencies.

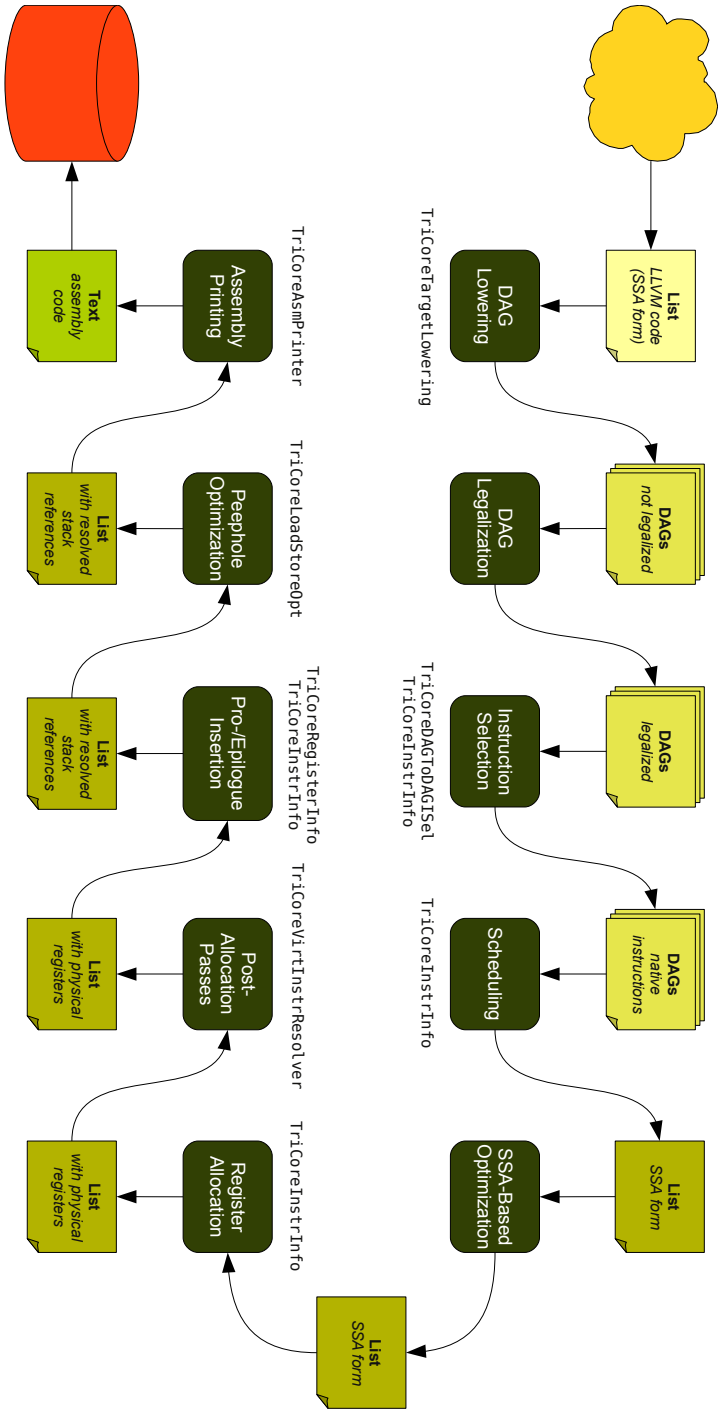

Figure 4.2: Code generation sequence. On the path from LLVM code to assembly code, numerous passes are run through and several data structures are used to represent the intermediate results.

As the target CPU architecture is guaranteed to not support all operations on all types, the DAG has to be legalized, i. e., it has to be converted into a form that only contains supported operations and supported data types. For instance, most of TriCore's integer instructions can only process 32-bit words, thus smaller integer operands have to be extended to `i32`.

The legalized DAG is then passed to the instruction selector, which performs pattern matching on it to build a new DAG whose nodes represent native target instructions. For every pattern of LLVM code, an appropriate pattern of target machine code is selected. Both input and output patterns can range from a single node up to a whole subgraph that comprises several nodes and edges.

4.2.2 Scheduling and Formation

The newly created target DAGs are deconstructed and their instructions are brought into a sequential list form. Every function is represented as an object of the `MachineFunction` class that contains a list of `MachineBasicBlock`s, which on their part include a sequence of `MachineInstr`s. The scheduler is required to determine the order in which the instructions are emitted. The decision is based on a number of constraints, e. g., minimal register pressure [4].

The list is in SSA form, so it does not yet contain fully valid assembly code. With a few exceptions (such as moves to a return value register or the like), all instructions still operate on an unbounded set of virtual registers, and all references to the stack frame address abstract stack slots instead of concrete offsets.

4.2.3 SSA-based Machine Code Optimizations

Before the actual allocation of physical registers takes place, the code generator may perform one or more target-specific low-level optimization passes.

4.2.4 Register Allocation

All references to virtual registers are eliminated from the program. The well-known colouring technique is used to map each virtual register to exactly one physical register. If the amount of live virtual registers exceeds the amount of available physical registers, spill code is generated. As registers may overlap[1], the register allocator has to take register aliasing into account. The elimination of all virtual registers is accompanied by the deconstruction of the SSA form. All previously inserted Φ-pseudo-instructions are simply replaced with copy instructions [4].

4.2.5 Prologue/Epilogue Code Insertion

After the physical registers have been allocated and possible register spills have been generated, it is possible to compute for every function how much space has to be reserved

[1] For instance, `%d3` and `%e2` overlap on the TriCore platform, because `%d3` denotes the upper 32 bits of the 64-bit extended register `%e2`.

for the stack frame. Consequently, the respective prologue and epilogue code sequences can now be emitted and all abstract stack slot references can be mapped to actual memory addresses relative to the frame pointer or stack pointer.

4.2.6 Late Machine Code Optimizations

Any "last-minute" peephole optimizations of the final machine code can be applied during this phase.

4.2.7 Code Emission

Finally, the completed machine code is emitted. For static compilation, the end result is an assembly code file; for JIT compilation, the opcodes of the machine instructions are written into memory.

4.3 General Target Information

Every LLVM target must offer a specified interface through which it can be accessed by the higher-level components. It has to implement a subclass of `TargetMachine`, provide information about existing subtargets, and register itself with the code generator.

4.3.1 Target Machine Characteristics

The class `TriCoreTargetMachine` is the central hub that acts as the primary interface between the backend and the "outside world". It creates and owns a number of objects used throughout the code generator. Many of the backend classes access one or more of these objects via pointers:

- `TargetData` contains extensive details about the low-level data layout, such as byte order (little-endian), pointer width (32 bits), and memory alignment.
- `TargetFrameInfo` describes the stack frame layout. It specifies in which direction the stack grows (downward), which alignment is required for stack frames (8 bytes [16]), and at which offset the area for local variables begins (0 bytes).
- Other classes include `TriCoreInstructionInfo`, `TriCoreTargetLowering`, and `TriCoreRegisterInfo`. Each of them is discussed in detail later in this chapter.

`TriCoreTargetMachine` also contains hook methods for registering the instruction selection pass and optionally a number of additional backend-specific passes with the pass manager.

4.3.2 Subtarget Information

For as good as every CPU architecture, there exists more than one concrete implementation. This may be the case because the architecture has a modular design that allows

some components of the CPU, for instance the FPU, to be optional. Alternatively, more than one vendor may have a licence to produce processors of a particular architecture (e. g., x86). Another possibility is that the specification may have been updated over time, adding new instructions and/or components or changing technical details. An optimal compiler backend would be aware of all chips that implement the respective architecture and their peculiarities.

For this purpose, LLVM provides the concept of *subtargets*. Subtargets implement a common target architecture, but differ from each other in a number of *features*, which are represented either as boolean variables or as enumerations. The backend can make decisions based on these values. For example, when generating code for a chip that includes an FPU, the instruction selector can directly emit floating-point instructions, while it has to map these instructions to library calls if the subtarget has no FPU.

A feature is represented in TableGen as an instance of the class `SubtargetFeature`, which has the feature's name, the name and value of the corresponding variable, a description string, and an optional list of dependencies as attributes:

```
def FeatureFPU : SubtargetFeature<"fpu", "fpu", "true",
                                  "Enable FPU">;
```

A concrete subtarget can then be defined by a name and a list of features that are supported by that processor:

```
def : Proc<"generic", [FeatureFPU]>;
```

The TriCore backend currently only supports this one subtarget, thus the presence of an FPU is always assumed. Thanks to the clean design, it would however be relatively straightforward and unproblematic to extend the target by adding new features or new processors later on.

4.3.3 Target Registration

The `TargetRegistry` is LLVM's mechanism to manage and lookup targets at runtime. Every backend owns a global instance of the `Target` class and offers function hooks to register itself. Upon registration, the `Target` object is filled with references to the relevant backend objects (including `TriCoreTargetMachine`), and the target string is recorded in the registry's lookup list.

It is worth noting that the registration mechanism is somewhat fine-grained: Instruction selector, assembly printer and JIT code emitter (if any) are registered and loaded separately from each other. This is useful as not every component of the backend is actually needed in every application scenario. For instance, when running in JIT mode, loading an assembly code printer would be an unnecessary waste of time and memory.

Files

`TriCore.td`
`TriCore.h`

```
TriCoreTargetMachine.h
TriCoreTargetMachine.cpp
TriCoreSubtarget.h
TriCoreSubtarget.cpp
TargetInfo/TriCoreTargetInfo.cpp
```

4.4 Register Information

As LLVM's code generator tackles most register-related tasks such as register allocation in a target-independent fashion, it needs to be provided with detailed information about the register files belonging to the target architecture. A large part of this information is again characterised in a TableGen input file.

4.4.1 Register Description Table

For the description of physical registers, TableGen pre-declares the specially handled classes `Register` and `RegisterWithSubRegs`. The latter is derived from the former and consequently inherits all of its attributes, but overrides `Register`'s empty list of sub-registers.

The TriCore backend implementation defines the two subclasses `TriCoreReg` and `TriCoreRegWithSubregs` derived from the aforementioned classes. They are declared as follows:

```
class TriCoreReg<bits<4> num, string name> : Register<name> {
  let Namespace = "TriCore";
  field bits<4> number = num;
}

class TriCoreRegWithSubregs<bits<4> num, string name,
                            list<Register> subregs>
        : RegisterWithSubRegs<name, subregs> {
  let Namespace = "TriCore";
  field bits<4> number = num;
}
```

Both classes override the `Namespace` field and carry a four-bit numerical identifier as an additional field[2]. Note the use of template arguments. Figure 4.3 shows a graphical representation of the register type hierarchy in a UML-like notation. It is worth noting that this hierarchy exists purely on a TableGen-specific level and is no longer visible in the resulting C++ code generated by `tblgen`.

[2]This identifier would actually only be needed if the TriCore backend had JIT capabilities. It would be encoded in the corresponding opcode bits of machine instructions.

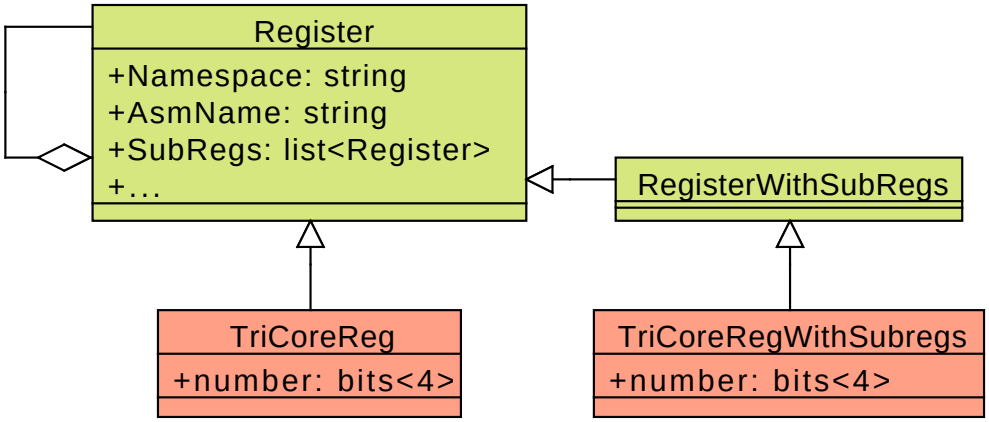

Figure 4.3: Register type hierarchy in TableGen. Pre-existing classes are coloured green, custom classes are marked red.

Each concrete physical register must be defined as an instance of one of the classes declared above. The identifier and name of the register are passed as template arguments. Additionally, a DWARF number for debugging purposes[3] is defined:

```
def D3 : TriCoreReg<3, "d3">, DwarfRegNum<[3]>;
```

Extended 64-bit registers additionally carry a list of the sub-registers they comprise:

```
def E2 : TriCoreRegWithSubregs<2, "e2", [D2, D3]>, DwarfRegNum<[33]>;
```

This information alone however does not suffice. It is necessary to tell the backend for each extended register which sub-register represents the higher 32 bits and which one represents the lower 32 bits. This is achieved by defining two `SubRegSet` records, with record 1 and 2 denominating the lower and higher registers, respectively. The code generator exploits this knowledge for miscellaneous minor optimizations concerning casts (or combinations of shifts and casts) between `i64` and `i32` values.

Last of all, the TableGen file defines a number of *register classes*, each of which is described by a value type, an alignment (for load/store operations), and a set of registers that belong to it. Register classes play an important role in the instruction selection phase: Every instruction that has one or more register operands must define each of these operands to be of exactly one register class, respectively. This information is used in later phases to determine which CPU registers come into question for holding a certain value. It is possible that two distinct register classes own the same physical registers. In the TriCore backend, this is the case for integer and floating-point operands, both of which are located in the data register file (`%d0`–`%d15`).

[3]The DWARF numbers of all registers are specified in the EABI manual [16].

Certain registers must not be seized and thus have to be "protected" from the register allocator. `%a0`, `%a1`, `%a8`, and `%a9` are reserved by convention [16]. Additionally, `%a10` (stack pointer), `%a11` (return address), and in some cases `%a14` (frame pointer) have to be marked as unavailable as they all serve special purposes and are not dispensible under any circumstances.

4.4.2 Non-static Register Information

For some target architectures, some aspects of the target architecture's register set are dependent upon variable factors and have to be determined at runtime. As a consequence, they cannot be generated statically from a TableGen description – although that would be possible for the bulk of them in the case of the TriCore backend. Among them are the following points:

- **Callee-saved registers.** Normally, the ABI specifies a set of registers that a function must save on entry and restore on return if their contents are possibly modified during execution. However, TriCore's built-in task model[4] saves and restores the processor's *upper context* automatically in these situations – so the list of callee-save registers is actually empty in `TriCoreRegisterInfo`.

- **Reserved registers.** Although the set of unavailable registers is already defined in the TableGen file, `TriCoreRegisterInfo` contains a method that marks all non-allocatable register numbers in a bit vector.

- **Frame register.** This is the base register for all memory accesses to stack slots. In most cases, the stack frame has a fixed size, so addresses are calculated relative to the stack pointer `%a10`. A dedicated frame pointer is needed only if the concerning function contains variably sized objects on the stack[5]. The TriCore EABI (Embedded Applications Binary Interface) explicitly does not specify a certain register to be used as the frame pointer [16] – the LLVM backend uses `%a14` for this purpose[6]. In any case, it is advised to pick a register from the *upper context* so that no action is needed to ensure that the frame pointer is preserved across calls.

Last of all, `TriCoreRegisterInfo` contains a few methods that emit code fragments into existing basic blocks. They are called by the `PrologEpilogInserter` pass – at a stage when the LLVM code has been translated into the target machine language and instruction scheduling as well as register allocation have already taken place. The machine code to be modified by those methods is present in doubly-linked list form, allowing easy insertion, removal, and modification of instructions. The following methods are implemented:

- `emitPrologue()` inserts prologue code at the beginning of a function. Thanks to TriCore's context model, this is a trivial task as it is not required to save any

[4] See Chapter 3.6.
[5] Such objects are usually created by a call to the `alloca()` function.
[6] This is the same register that HighTec's TriCore port of GCC uses as the frame pointer.

registers manually. The only thing that has to be done is reserving space for the function's stack frame by decrementing the stack pointer. In addition, if the function needs a frame pointer, the frame register `%a14` is set to the old value of the stack pointer beforehand.

- `emitEpilogue()` is intended to emit instructions to destroy the stack frame and restore all previously saved registers before returning from a function. However, as `%a10` (stack pointer), `%a11` (return address), and `%a14` (frame pointer, if any) are all part of the *upper context*, no epilogue code is needed at all. All cleanup operations are performed implicitly by the `ret` instruction.

- `eliminateCallFramePseudoInstr()` handles the `ADJCALLSTACKDOWN` and `ADJCALLSTACKUP` pseudo-instructions, which are inserted before and after each function call during the *lowering* phase. Both pseudo-instructions have an immediate operand that indicates how many bytes the arguments of the call occupy on the stack. Most commonly, its value is zero as arguments are usually passed in registers[7], so the pseudo-instructions can be simply removed from the basic block. If the calling function has a fixed-size stack frame, space for all actual arguments is included in this size and is consequently already allocated in the prologue of the function, thus no further action is required either.

  The only contrary situation is when the calling function contains variably-sized objects on its stack, thus rendering it impossible to determine a maximum stack frame size at compile-time. In this case, `ADJCALLSTACKDOWN` and `ADJCALLSTACKUP` have to be replaced by a subtraction from and an addition to the stack pointer, respectively.

- `eliminateFrameIndex()` is called for each instruction that references a word of data in a stack slot. All previous passes of the code generator have been addressing stack slots through an abstract frame index and an immediate offset. The purpose of this function is to translate such a reference into a register–offset pair. Depending on whether the machine function that contains the instruction has a fixed or a variable stack frame, either the stack pointer `%a10` or the frame pointer `%a14` is used as the base register. The offset is computed accordingly. Figure 4.4 demonstrates for both cases how a stack slot is addressed.

  If the addressing mode of the affected instruction cannot handle the address because the offset is too large (the offset field has 10 bits for the BO addressing mode and 16 bits for the BOL mode), a sequence of instructions is emitted that explicitly computes the effective address. Interim results are put into an unused address register. If none is available, an already occupied address register is scavenged. For this purpose, LLVM's framework offers a class named `RegScavenger` that takes care of all the details.

[7] TriCore's calling conventions are explained in detail in Chapter 4.5

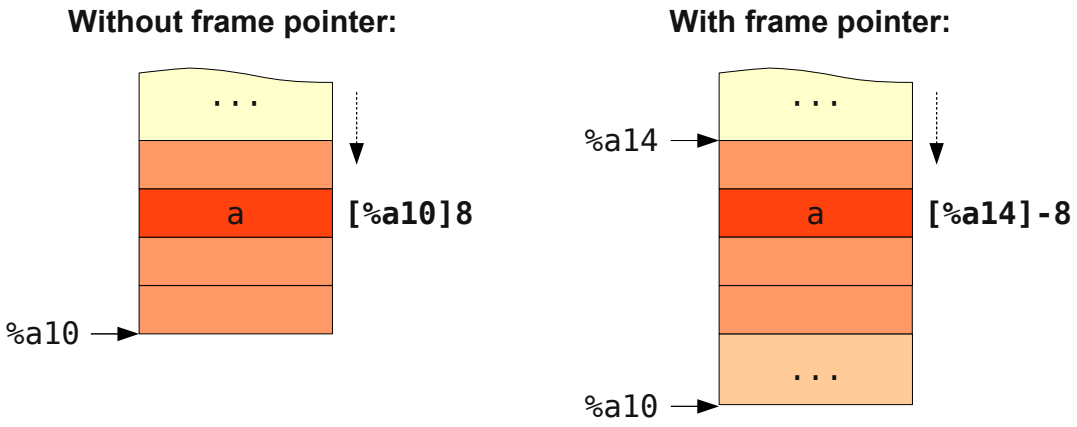

Figure 4.4: Addressing of a variable `a` located on the stack. If the stack frame has a variable size, slots must be addressed relative to the frame pointer.

Files

`TriCoreRegisterInfo.td`
`TriCoreRegisterInfo.h`
`TriCoreRegisterInfo.cpp`

4.5 DAG Lowering

The process of preparing the input program for the instruction selection stage by converting it from list form into a directed acyclic graph is called *lowering*. This is the very first step of the code generation procedure. For every instruction of the LLVM program an `SDNode` (SelectionDAG node) object is created. It contains the following information:

- **Opcode.** This is an integer that identifies the instruction represented by the node.
- **Results (definitions).** While most instructions produce exactly one result, some may either define several values (e. g., operations with side effects, combined division/modulo instructions) or no value at all (e. g., branch instructions). The node object keeps a list of the value types for all of its results.
- **Operands (uses).** Every `SDNode` keeps a record of all other nodes upon which it has a dependency. This involves either a data dependency (i. e., this node uses a value defined by another node) or a control flow dependency (i. e., the instruction represented by another node must be executed before this instruction). Such a relationship to another `SDNode` is represented by an `SDValue` object, which encapsulates a pointer to the pertained node along with the index number of the affected result. Every `SDNode` therefore owns a list of `SDValues` and an associated list of value types. The only difference between data- and control flow ("chain")

dependencies is the fact that for the latter the pseudo-type `MVT::Other` is recorded instead of a regular type (such as `MVT::i32`).

Figure 4.5 shows the DAG that is constructed for the basic block `if.else` of the factorial example function.

The very bulk of the SelectionDAG construction is executed by the target-independent framework classes `SelectionDAGBuild` and `TargetLowering`. The backend class `TriCoreTargetLowering`, a subclass of `TargetLowering`, offers a two-fold functionality. On the one hand, it provides its superclass with target-specific information, including:

- the alignment of machine functions (2 bytes[8]),
- all value types natively supported by the target machine and for each type the register class that has been defined for it,
- and the exact binary representation of boolean values[9] (`true`: 1, `false`: 0).

On the other hand, `TriCoreTargetLowering` handles all cases of instruction nodes that cannot be lowered automatically but require manual intervention. These cases are described below.

4.5.1 Calling Conventions

The way arguments are passed to and return values are received from functions are highly target-specific. The ABI mandates a set of rules that must be strictly satisfied. The TriCore EABI User's Manual [16] specifies the following calling conventions concerning argument passing and return values:

- Up to four arguments of the type `i32` or `f32` can be passed in the registers `%d4`–`%d7`. Integers smaller than 32 bits are extended to this width beforehand.
- 64-bit arguments are put into the extended register pairs (`%d4`, `%d5`) and (`%d6`, `%d7`).
- A maximum of four pointer arguments can be passed in the registers `%a4`–`%a7`.
- If the number of registers does not suffice to accept all arguments, the remaining parameters are pushed onto the stack in reverse order, aligned on a four-byte boundary. If the callee has a variable number of arguments, all non-fixed parameters are also passed via the stack.
- Return values of the type `i32` or `%f32` are returned in `%d2`, smaller integers are extended. 64-bit values are put into the register pair (`%d2`, `%d3`).
- Pointers are returned in `%a2`.

[8] It is actually impossible to break this alignment rule because all instructions are either 16 or 32 bits long.

[9] Some CPU architectures, for instance the Cell SPU, define `true` as -1.

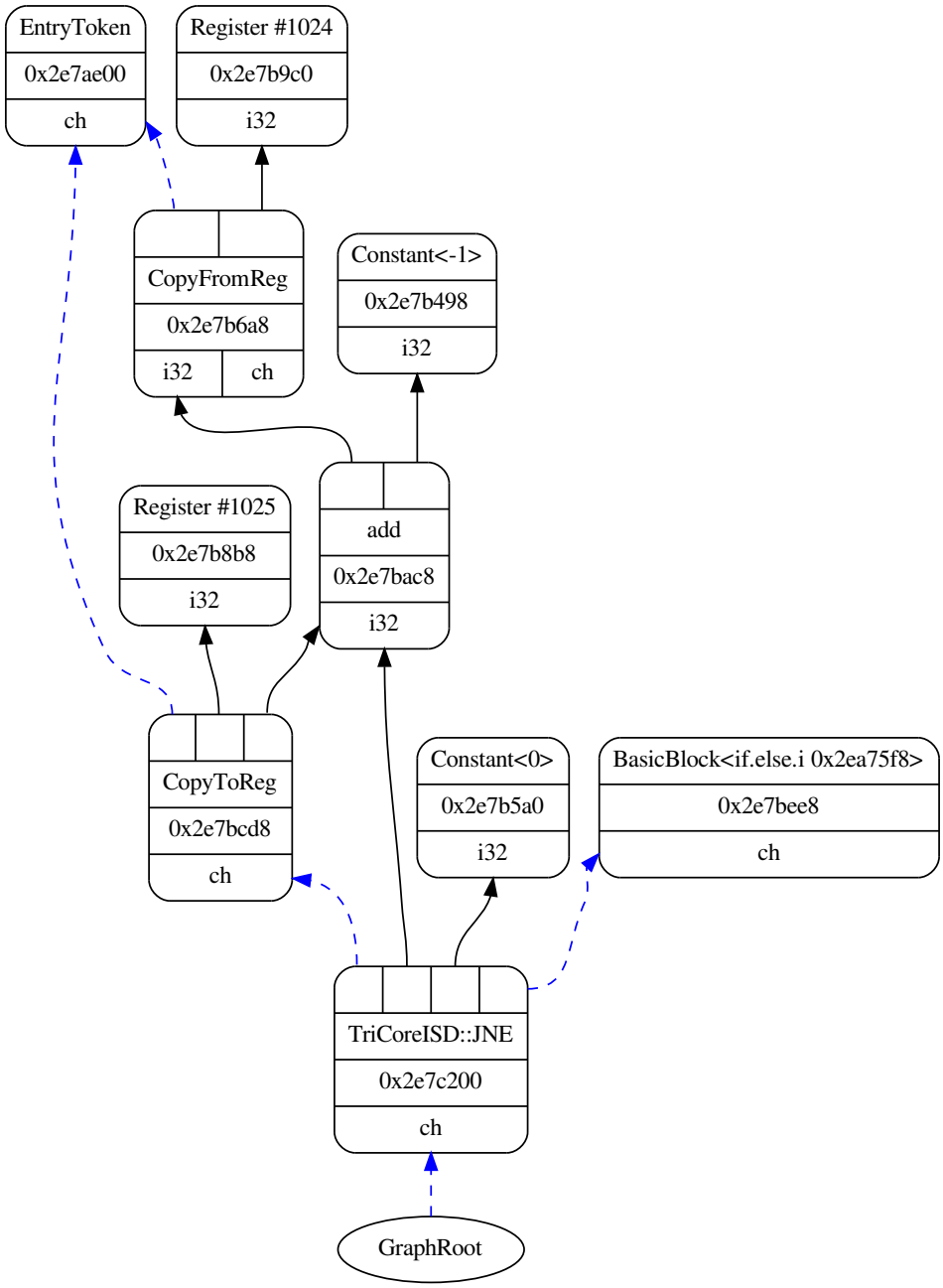

Figure 4.5: SelectionDAG for the basic block `if.else` of the function `factorial()`. Black arrows denote data dependencies while blue dashed arrows represent control flow dependencies or references to other basic blocks. Instruction selection has not yet taken place – the conditional branch was converted to an intermediate `TriCoreISD::JNE` node during the lowering phase because of the comparison requirements discussed below.

The strict distinction between pointers and integers is highly problematic because LLVM's code generator implicitly converts all pointers to integers of the same width (`i32` in the case of TriCore) upon construction of the SelectionDAG. The solution presented later in this section requires some changes in the underlying framework.

Calling conventions are described in TableGen format by a set of conditions and predefined actions such as `CCAssignToReg`, `CCAssignToStack`, etc. The following example shows how an address argument is handled. First, an attempt is made to put the argument into one of the registers `%a4`–`%a7` (in this order). This will fail if all of the designated registers are already occupied by previous function parameters. In this case, the "stack model" calling convention is used instead, which pushes the argument into a four-byte stack slot with an alignment of four bytes.

```
def CC_TriCore_StackModel : CallingConv<[
  CCAssignToStack<4, 4>
]>;

def CC_TriCore_Address : CallingConv<[
  CCAssignToReg<[A4, A5, A6, A7]>,
  CCDelegateTo<CC_TriCore_StackModel>
]>;
```

The description file is processed by `tblgen` in the usual manner. For each definition of a `CallingConv` instance, a function is generated that contains a number of nested `if`-statements. Delegations to another calling convention are expressed as calls to the corresponding function. Whenever a function argument or return value is lowered, the relevant calling convention function is invoked and determines a location for the variable. The actual lowering process is done by the class `TriCoreTargetLowering`, which is consulted every time one of the following situations occurs:

1. The scope of a function is entered. In order to make the formal arguments available to the function, they first have to be moved into virtual registers. For this purpose, `LowerFormalArguments()` is invoked. It determines for each formal argument where it is located, creates a new virtual register of the appropriate register class, and inserts a sequence of moves and/or loads into the DAG. The emitted `SDValues` are connected by "chain" edges.

2. The scope of a function is left. This case is handled by the `LowerReturn()` method. It takes care of moving the return value, which may be split up into several parts, into the corresponding physical registers. Subsequently, a `RET_FLAG` node is emitted that will later be matched with a `ret` instruction during the instruction selection stage.

3. A function is called. This demands that all actual arguments be copied into the right places before the call and that afterwards all return value fragments be transferred back into the designated virtual registers. The whole process must

```
1  define i32 @factorial(i32 %n) nounwind readnone {
2  entry:
3    %cmp = icmp eq i32 %n, 0
4    br i1 %cmp, label %return, label %if.else
5
6  if.else:
7    %sub = add i32 %n, -1
8    %cmp.i = icmp eq i32 %sub, 0
9    br i1 %cmp.i, label %factorial.exit, label %if.else.i
10
11 if.else.i:
12   %sub.i = add i32 %n, -2
13   %call.i = tail call i32 @factorial(i32 %sub.i) nounwind
14   %mul.i = mul i32 %call.i, %sub
15   br label %factorial.exit
16
17 factorial.exit:
18   %call3 = phi i32 [ %mul.i, %if.else.i ], [ 1, %if.else ]
19   %mul = mul i32 %call3, %n
20   ret i32 %mul
21
22 return:
23   ret i32 1
24 }
```

Figure 4.6: LLVM code example with virtual registers affected by calling convention lowering marked red. The following situations have to be handled:

- Line 1: Before the formal argument `%n` can be used, its value must be copied from the physical register `%d4`.
- Line 13: The actual argument `%sub.i` must be moved into the physical register `%d4` before the call. After returning from the function, the return value located in the physical register `%d2` has to be copied into `%call.i`.
- Lines 20 and 23: The return values have to be put into the physical register `%d2`, respectively.

be "enframed" by a `CALLSEQ_BEGIN` and a `CALLSEQ_END` node. These nodes will be transformed into `ADJCALLSTACKDOWN` and `ADJCALLSTACKUP` pseudo-instructions (see Chapter 4.4.2) during the instruction selection stage. The function call is processed by the method `LowerCall()`, which emits an appropriate chain of `SDValue`s.

Figure 4.6 on the next page takes up the code example from Figure 2.2 with annotations demonstrating which of the above three lowering scenarios occurs at which position in the code.

As mentioned above, LLVM's agnosticism regarding pointers initially makes it impossible to comply with the EABI as there is no way to tell whether an integer argument should go into an address register or a data register. Therefore it was necessary to make selective modifications to the framework without breaking compatibility with the existing backends.

The solution is to annotate the `EVT` class, which represents an extended value type, with a flag that indicates whether this is a normal integer (or other type) or actually a pointer. To make this work, several methods of `SelectionDAGLowering` had to be slightly adapted in order to transfer the flag from one node to another. The calling convention functions can then evaluate the annotated flag of the argument type and determine the correct destination location accordingly.

Two other minor customizations were necessary to fulfil the ABI requirements concerning variable arguments and 64-bit values split into two 32-bit parts. The structure `ArgFlagsTy`, which stores certain characteristics of a function parameter, was extended by two additional flags indicating whether the argument is passed as an element of a variable argument list, and whether it is the second part of a split value, respectively. The structure is again accessible by the calling convention functions.

4.5.2 Custom Lowering of Instructions

In addition to the calling conventions, there are a number of instructions and operands that have to be lowered manually due to certain properties and restrictions of the TriCore architecture:

- **Global addresses, jump table indices, and constant pool entries.** These are actually nothing but absolute virtual addresses. During manual lowering, they are split up into a `HI` and a `LO` part, and an `ADD` node is emitted that sums the two parts. The instruction selector will later convert the addition into a machine address used by load, store, or LEA (*load effective address*) instructions.
- **Right-shifts.** One peculiarity of TriCore is the fact that there are no distinct left- and right-shift instructions. Instead, it only has the two instructions `sh` (logical shift) and `sha` (arithmetic shift), with the sign of the second operand determining the direction: If it is positive, the first operand is shifted to the left by this amount; if it is negative, it is shifted to the right by the absolute amount. Consequently, the instruction selector would be unable to match LLVM's `SRL` and `SRA` nodes. `TriCoreTargetLowering` intercepts these nodes and replaces them with the corresponding left-shifts after negating the second operand.

Original	After conversion
$a > const$	$a \geq const + 1$
$a > b$	$b < a$
$a \leq const$	$a < const + 1$
$a \leq b$	$b \geq a$

Table 4.1: Condition code conversion for integers. This is necessary because TriCore has no dedicated instructions for the $>$ and $\leq$ operators.

Operator	Bit 2 ($>$)	Bit 1 ($=$)	Bit 0 ($<$)
$=$	-	1	-
$>$	1	-	-
$\geq$	-	-	0
$<$	-	-	1
$\leq$	0	-	-
$\neq$	-	0	-

Table 4.2: Condition code flags for floating-point numbers. The lower three bits of a data register defined by a `cmp.f` instruction represent the comparison result. A subsequent conditional branch has to examine exactly one of these bits, depending on its comparison operator.

- **Comparisons and conditional branches.** As anticipated in Chapter 3.5, TriCore does not have a condition code register whose contents are set implicitly by certain instructions. Instead, it offers two kinds of comparison operations for integers: explicit instructions that compute a boolean value (1 or 0) as a result and put it into a data register, and combined comparison and conditional branch instructions. The set of conditions is minimal: Only the $<$, $\geq$, $=$, and $\neq$ operators are supported directly. The $\leq$ and $>$ operators have to be emulated as shown in Table 4.1. After the condition code has been converted, the appropriate SelectionDAG node is produced. As the resulting `SDNode` cannot be described by one of LLVM's generic target-independents opcodes, a list of custom DAG opcodes specific to the backend is required from which the respective element is taken.

  For floating-point numbers, things work differently. The `cmp.f` instruction takes two `f32` operands and sets a number of flag bits in the output data register, the lower three bits of which represent the results of *less-than*, *equals*, and *greater-than* comparisons. A conditional branch has to examine exactly one of those bits. Depending on whether the bit is expected to have the value 1 or 0, a `jnz.t` (*jump if bit is non-zero*) or `jz.t` (*jump if bit is zero*) instruction is emitted. Which bit must be evaluated by the instruction and which value it must have can be inferred from Table 4.2.

Files

`TriCoreCallingConv.td`
`TriCoreISelLowering.h`
`TriCoreISelLowering.cpp`

4.6 Instruction Set Specification

The description, handling and transformation of instructions constitutes the core part of the backend. The code generator therefore needs extensive and detailed information about the names, opcodes, operands, behaviour, and possible side effects of the target machine instructions. The description of the ISA is distributed among two separate TableGen files: `TriCoreInstrFormats.td` specifies the abstract structure and format of the instruction set while `TriCoreInstrInfo.td` contains the concrete instruction descriptions.

4.6.1 Instruction Formats

A machine instruction is specified in TableGen by the class `Instruction`, which contains among others the following fields:

- **Output operands.** This contains the value(s) directly defined by the instruction as a result of the computation it performs.
- **Input operands.** This holds all `SDValues` used by the instruction as input arguments.
- **Assembly string.** When the assembly printer emits an instruction, the string stored in this field is printed. The string may contain placeholders for operands.
- **DAG pattern.** This pattern of machine-independent SelectionDAG nodes is matched by the instruction selector to produce an instance of the corresponding target-specific instruction.

As depicted in Figure 4.7, the TriCore backend specifies two subclasses of `Instruction` with different `opcode` fields for 16- and 32-bit instructions, respectively. A third subclass exists for the `ADJCALLSTACKDOWN` and `ADJCALLSTACKUP` pseudo-instructions.

Although all opcodes have a fixed width of either 16 or 32 bits, the actual encoding varies significantly between the diverse concrete instruction formats, with the least-significant bit of the opcode determining the width of the instruction:

- If it is zero, the first byte (*op1*) identifies the (16-bit) instruction and the second byte contains the operand encodings.
- If the value of the bit equals one, the *op1* byte denotes the instruction group. The three subsequent bytes comprise the operands and an additional *op2* bitfield that names the instruction within the group. The actual encoding is highly dependent on the instruction group.

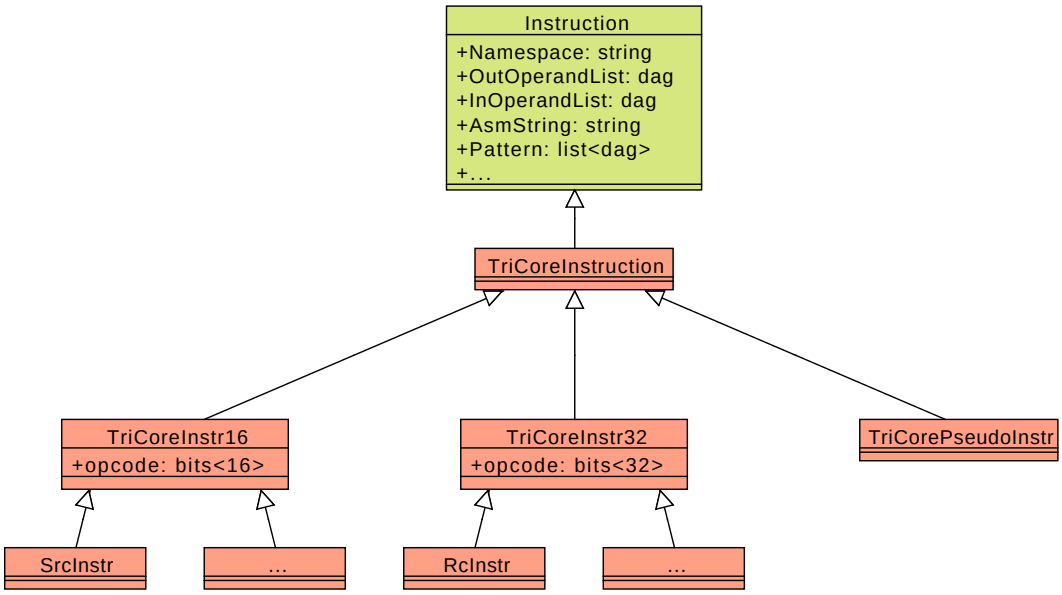

Figure 4.7: Instruction type hierarchy in TableGen. The `Instruction` class is pre-defined whereas all subclasses are target-specific.

Exceptions exist for both cases. Even though the binary encoding is presently only relevant if the target is to support just-in-time compilation – which is not the case for TriCore – it was nonetheless decided to respect it at least partly.

Based on `TriCoreInstr16` and `TriCoreInstr32`, a subclass exists for each instruction format described in the instruction set manual [18]. For example, the subclass for the RC (*register + constant*) opcode format, whose *op2* field spans the seven bits 27–21, is declared as follows:

```
class RcInstr<bits<8> op1, bits<7> op2, dag outs, dag ins,
              string asmString, list<dag> pattern>
        : TriCoreInstr32<outs, ins, asmString, pattern> {
  let opcode{7-0}   = op1;
  let opcode{27-21} = op2;
}
```

The operands of the instruction are not yet encoded in the opcode bits. There is currently no need to do so because the final result of the code generation process is not machine code, but human-readable ASCII assembly code.

4.6.2 Instruction Description Table

Starting from the instruction format classes discussed in the previous section, the TableGen file `TriCoreInstrInfo.td` contains the concrete description of the instructions

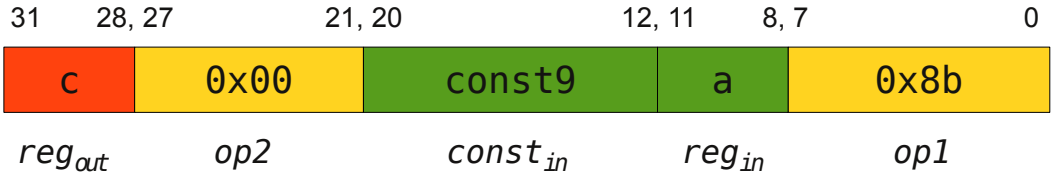

Figure 4.8: Encoding of the `add` instruction in RC form. This instruction adds a data register and a 9-bit signed immediate integer and puts the result in a data register.

of the target machine. The way this works is explained hereafter using the example of the `add` instruction in RC form, which performs the following operation:

$$c \leftarrow a + signExtend(const9)$$

The 9-bit immediate integer *const*9 is sign-extended[10] to 32 bits and then added to the contents of the data register a. The result is put into the data register c. Figure 4.8 shows what the opcode for the instruction looks like.

First of all, it is necessary to define a type for the 9-bit immediate operand, which is actually treated like an `i32` operand, but needs special handling when printed. For this purpose, a custom method must be implemented in the `TriCoreAsmPrinter` (which is discussed below) and TableGen must be told about the name of this method – in the following example, this method is a function template. Again, an operand is described by instance of the pre-defined TableGen class `Operand`:

```
def s9imm : Operand<i32> {
  let PrintMethod = "printSExtConstOperand<9>";
}
```

The instruction selector needs to know when it is possible to select a constant node as 9-bit immediate operand of a machine instruction. This is the case if the value of the node lies within the interval $[-2^8, 2^8)$. The check is carried out by a custom piece of C++ code that examines the operand node `N`, which is a pointer to a `ConstantSDNode`:

```
def immSExt9 : PatLeaf<(i32 imm), [{
  int32_t v = N->getZExtValue();
  return ((v >= -(1 << 8)) && (v < (1 << 8)));
}]>;
```

This definition is translated by `tblgen` into a boolean C++ function that is consulted by the instruction selector whenever a target instruction with a *const9* operand comes into question. Only if `true` is returned (and all other criteria are met), the instruction is selected.

The TriCore ISA offers two variants of some integer arithmetic instructions: one that operates on data registers and one that processes the contents of address registers.

[10] Sign-extension is done by filling the remaining upper bits with the value of the most-significant bit of the immediate number. In contrast, zero-extension would fill these bits with zeros.

Because LLVM's pattern matching mechanism in principle makes its decisions based only on the data types of the operands – which are `i32` in both cases – only one of the two variants could be selected. For the other variant, a lot of moves between data and address registers would be produced. The solution is to add *predicates* to all affected instructions – additional conditions that are checked. For the TriCore target, the predicates `isInteger` and `isPointer` are defined which evaluate the annotated type flag described in Chapter 4.5.1, thus determining whether the result of the `SDNode` in question is a regular integer or a pointer:

```
def isInteger : Predicate<"N.getValueType().isPointer() == false">;
def isPointer : Predicate<"N.getValueType().isPointer()">;
```

Finally, the instruction itself is defined under the name `ADDrc`, using the TableGen classes and definitions discussed above:

```
let Predicates = [isInteger] in
  def ADDrc : RcInstr<0x8b,                          // op1
                      0x00,                          // op2
                      (outs DR:$c),                  // outs
                      (ins DR:$a, s9imm:$const9),    // ins
                      "add\t$c, $a, $const9",        // asmString
                      [(set DR:$c, (add DR:$a, immSExt9:$const9))]>;
```

The output and input operands are tuples in the format `location:$name`, with `location` being either a register class or an instance (pre-defined or custom) of `Operand`. The auto-generation of the assembly printer handles each appearance of `$name` in the assembly string by replacing it with a call to the respective operand printing function.

The last template argument is the instruction selection pattern. It is present in prefix notation and represents the DAG pattern that must be matched in order to select this instruction. All operand names defined by `outs` and `ins` must appear in it. The pattern in the above example should be self-explanatory.

Memory accesses are possible only by using dedicated load and store instructions with the addressing modes presented in Chapter 3.4. The most frequently used addressing mode is BOL (*base register + long immediate offset*). For such addresses, a new subclass of `Operand` has to be defined, once again with a custom print method:

```
def MEMbol : Operand<iPTR> {
  let PrintMethod    = "printMemOperand";
  let MIOperandInfo = (ops AR:$b, i32imm:$off10);
}
```

As TableGen is unable to generate selection code for complex addressing modes, selection of a `MEMbol` operand must be done manually – in the case under consideration by the method `selectADDRbol()` of the class `TriCoreDAGToDAGISel`:

```
def ADDRbol : ComplexPattern<iPTR, 2, "selectADDRbol", [frameindex]>;
```

The two newly defined records can be employed in load and store instructions like other regular operands – `MEMbol` in the input operand list, its counterpart `ADDRbol` in the source pattern.

The previous instruction-based pattern matching approach bears the limitation that the result of a selection is always only a single instruction. It is however possible to define custom expressions that allow matching between arbitrary source patterns and arbitrary target patterns which are used if no single-instruction pattern has matched [4]. For instance, divisions in TriCore assembly code consist of a `dvinit` followed by a sequence of four `dvstep` instructions:

```
def : Pat<(sdiv DR:$a, DR:$b),
        (EXTRACT_SUBREG (DVSTEPrrr (DVSTEPrrr (DVSTEPrrr (DVSTEPrrr
        (DVINITrr DR:$a, DR:$b), DR:$b), DR:$b), DR:$b), DR:$b),
        subreg_even32)>;
```

The concluding `EXTRACT_SUBREG` node is produced because the division result is put into an extended 64-bit data register, with the even register representing the actual result and the odd register containing the remainder.

4.6.3 Non-static Instruction Information

In addition to the static information generated from the instruction description table, the `TriCoreInstrInfo` class contains methods for the analysis, transformation, and insertion of certain target machine instructions. All of these methods are not called until the code is back in list form after the scheduling stage has taken place. They perform the following operations:

- analysis of register moves and stack load/store instructions,
- branch analysis and conversion (e. g., if two unconditional branches directly follow each other, the second one can be removed),
- and insertion of register moves, stack loads/stores, and no-op instructions.

Files

`TriCoreInstrFormats.td`
`TriCoreInstrInfo.td`
`TriCoreInstrInfo.h`
`TriCoreInstrInfo.cpp`

4.7 Instruction Selector

The instruction selection stage is the moment when the program is transformed from LLVM's internal representation into the representation of the target machine. The input and output of the instruction selector are both directed acyclic graphs, with the nodes

of the input DAG representing LLVM instructions and the nodes of the output DAG representing target instructions. The translation is done via pattern matching – the instruction selector searches for known patterns (as specified in the instruction description table) in the first and adds the corresponding instruction patterns to the latter.

Large parts of the selection code are generated automatically by `tblgen`. For the `ADDRC` instruction discussed above, the resulting C++ code looks like this:

```
if ((N.getValueType().isPointer() == false)) {
  SDValue N0 = N.getOperand(0);
  SDValue N1 = N.getOperand(1);
  if (N1.getOpcode() == ISD::Constant) {
    if (Predicate_immSExt9(N1.getNode())) {
      SDNode *Result = Emit_1(N, TriCore::ADDrc, MVT::i32);
      return Result;
    }
  }
}
```

This code fragment is located in the auto-generated function `Select_ISD_ADD_i32()`. Dispatching is done by the function `SelectCode()`, which calls the appropriate sub-function according to the opcode and result type of the currently examined `SDValue`:

```
SDNode *SelectCode(SDValue N) {
  MVT::SimpleValueType NVT = N.getNode()->getValueType(0)
          .getSimpleVT().SimpleTy;
  switch (N.getOpcode()) {
    // Other opcodes...
    case ISD::ADD: {
      switch (NVT) {
      case MVT::i32:
        return Select_ISD_ADD_i32(N);
      default:
        break;
      }
      break;
   }
    // Other opcodes...
  }
  // Error handling...
  return NULL;
}
```

There are, however, a small number of cases that cannot be covered by the automatically created selection functions. They include machine instructions that define multiple result values, complex addressing modes, and others [4]. In the TriCore backend, the following situations are handled manually:

- **Addresses in *base register + immediate offset* form.** This is the most frequently used addressing mode in TriCore programs. This addressing mode is used not only for accesses to stack slots, but also for the combining the `LO` and `HI` parts of global addresses, jump table indices, and constant pool entries. Below are two examples of what the final assembly code will look like:

```
# Load from a stack slot:
ld.w %d0, [%a10]8

# Load from a global address:
movh.a %a2, HI:globalVar
ld.w   %d0, [%a2]LO:globalVar
```

  The selection functions take the root node of an input pattern, which can be either a frame index or an addition of a register and a constant or of a `HI` and a `LO` node. They produce a `MEMbol` (or similar) operand node if all conditions are met. In the two latter cases, base and offset are set accordingly. In the first case, the base is preliminarily set to the frame index and the offset is set to 0 – this will later be corrected during the prologue/epilogue insertion phase by a call to `TriCoreRegisterInfo::eliminateFrameIndex()`.

- **32-bit multiplications that define a 64-bit result.** These instructions are needed for multiplications of 64-bit integers. LLVM's `SMUL_LOHI` (signed) and `UMUL_LOHI` (unsigned) nodes are directly mapped to TriCore's `mul` and `mul.u` instructions, respectively, along with `EXTRACT_SUBREG` nodes that enable extracting the lower or higher 32 bits from the resulting 64-bit extended data register.

- **Logical AND operations.** This is actually not a necessity, but an optimization. TriCore offers dedicated instructions for the extraction of a sequence of n bits (which equals an AND with $2^n - 1$ if the sequence is starting at bit 0). If such a case is detected, an `extr.u` node is emitted – otherwise, the auto-generated instruction selection code is called and a regular `and` instruction is produced. The benefit of `extr.u` is that even for great values of n all operands can still be encoded directly into the instruction whereas for `and`, an immediate right-hand side operand larger than $2^9 - 1$ would not fit into the *const9* field. As a consequence, additional instructions would have to be inserted that load the constant into a data register in 16-bit chunks.

Figure 4.9 depicts the example DAG after the instruction selector has mapped each LLVM instruction to a TriCore machine instruction.

Files

`TriCoreISelDAGToDAG.cpp`

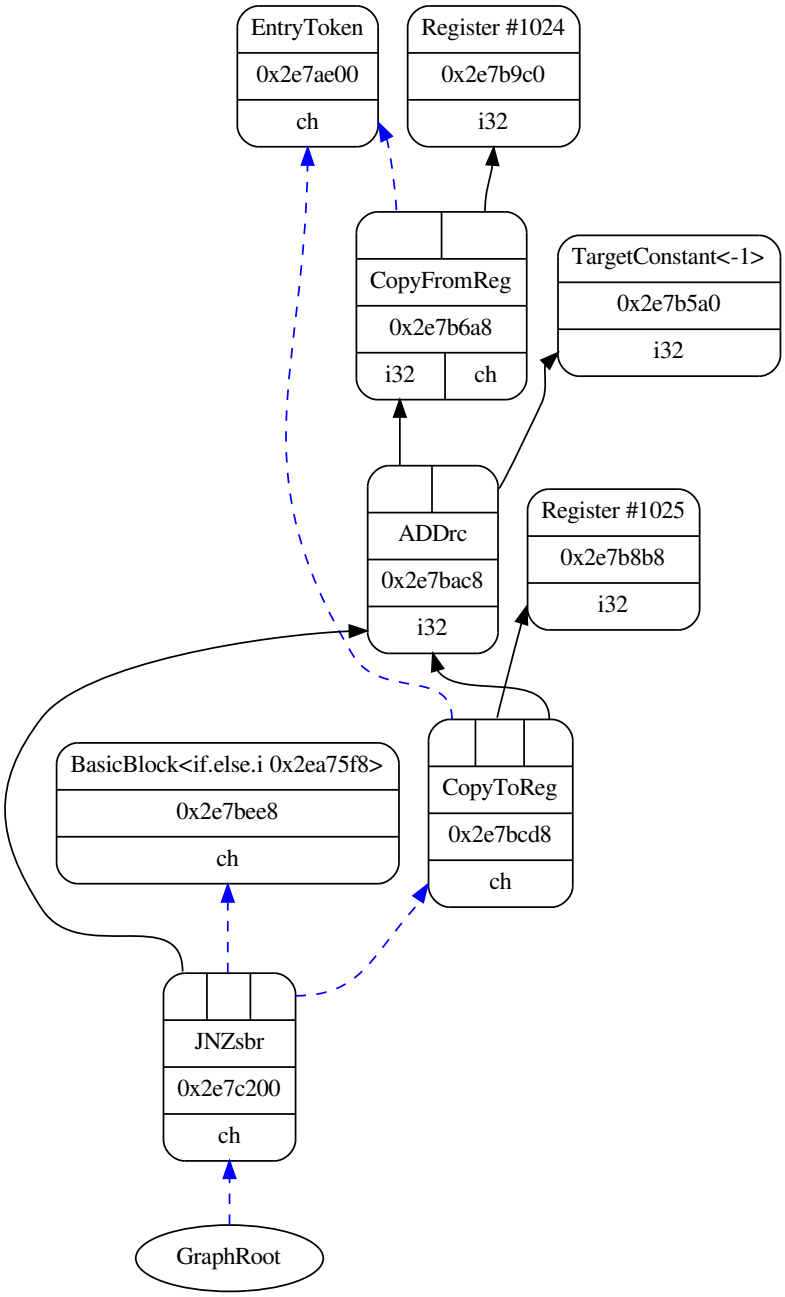

Figure 4.9: DAG for the basic block `if.else` of the function `factorial()` after instruction selection has taken place. The addition and the conditional branch have been transformed into native instructions, whereas the `CopyFromReg` and `CopyToReg` pseudo-instructions are still target-independent. They will be resolved at the latest when the virtual registers are allocated to physical registers.

4.8 Virtual Instruction Resolution Pass

The LLVM code generator contains a table with a mapping from primitive types (without extended value type flags) to register classes. This map is consulted every time a new virtual register is to be created, for example when a value has to be preserved across basic block boundaries. Unfortunately, it has the limitation that only a single entry exists per data type. With integers and pointers both being based on the primitive type `i32`, such newly created virtual registers would either have to be all assigned to the data register file or all to the address register file. As a consequence, in order to enable an instruction to use values located in the wrong register file, a move operation to the correct register file would have to be inserted.

This problem was solved by introducing the new "virtual" register class `IR`, which is superior to the `DR` and `AR` classes and contains all registers from both of them, and record it for the `i32` type in the mapping table. As the instruction selector bases its decisions upon data types, but not upon register classes, its correct functionality remains ensured. The register allocator then assigns the concrete physical registers under consideration of the constraints provided by every machine instruction.

This rather elegant solution however bears one difficulty: Moves between, loads from, and stores to virtual registers need to know exactly which "physical" register classes their operands belong to. This is because different variants exist for different operand types – for instance, `mov` copies a value between two data registers whereas `mov.a` copies a value from a data register to an address register. For this reason, if no concrete machine instructon can be emitted yet because of missing information, a preliminary `VIRTUAL_MOVE`, `VIRTUAL_LOAD`, or `VIRTUAL_STORE` pseudo-instruction is produced. As soon as register allocation has taken place, all required information is available to the `TriCoreVirtInstrResolver` pass, which converts the virtual instructions into the appropriate final CPU instructions.

Files

`TriCoreVirtInstrResolver.cpp`

4.9 Load/Store Peephole Optimizer

Immediately before the final assembly code emission stage, a custom peephole optimization pass is executed if optimizations are enabled for the compiler run. It searches for pairs of successive 32-bit memory accesses and tries to replace each of them with one 64-bit memory access. This is possible if all of the following requirements are met:

- Both instructions are of the same kind.
- The instructions load to / store from neighbouring data registers and address adjacent memory words.

- The data register used by the instruction accessing the smaller memory address has an even number that is smaller by one than the number of the register used by the other instruction.

Below is an example of how two `st.w` instructions can be merged into a single `st.d`:

```
# Before optimization:
st.w [%a10]4, %d9
st.w [%a10]0, %d8

# After optimization:
st.d [%a10]0, %e8
```

A counterexample of situations where the optimization cannot be applied:

```
# (%d5, %d6) is not a valid extended register:
st.w [%a10]4, %d6
st.w [%a10]0, %d5

# (%d4, %d3) is not a valid extended register:
st.w [%a10]4, %d3
st.w [%a10]0, %d4
```

The implementation of the pass makes some simplified assumptions – for example, only successive instructions with identical opcodes are regarded. Thus not all theoretical cases where an optimization would be possible are actually covered, only the most common ones. In order to also handle the rare possible cases – which may never actually occur in the generated code – a deeper and more sophisticated analysis would be necessary.

Files

`TriCoreLoadStoreOptimizer.cpp`

4.10 Assembly Printer

The final phase of the code generation process is the output of the resulting assembly code. Currently, the only supported way is the emission of human-readable assembly instructions into a text file, so it is still inevitable to use a separate assembler and linker. Work is being done though to create an own assembler within LLVM and to provide a framework for direct emission of machine code into object files [1].

The assembly printer is organized as a regular compilation pass and works in a fairly straightforward manner: For every existing function, the `runOnMachineFunction()` method is invoked, which first prints out the header of the function and then processes its basic blocks. For each `MachineBasicBlock`, the label and the instructions it contains are emitted. Large parts thereof are auto-generated from the instruction description table.

The `printInstruction()` method has to be called for every machine instruction. It is created by `tblgen` and prints out the corresponding assembly string specified in the TableGen file. If the string contains operands (denoted by a leading `$` sign), callbacks to the corresponding functions are conducted. All operands with a default type are handled by `printOperand()` – custom operands defined as described in Chapter 4.6.2 (e. g., immediates and complex addresses) are dealt with by the respective appointed print methods.

The output of assembly directives is supported by a comprehensive framework which is configured by implementing a subclass of `MCAsmInfo`. The latter contains a great number of flags describing the characteristics of the targeted assembler tool.

Files

`TriCoreMCAsmInfo.h`
`TriCoreMCAsmInfo.cpp`
`AsmPrinter/TriCoreAsmPrinter.cpp`

4.11 Integration into LLVM

Finally, in order to be able to make use of the new target directly in LLVM's tools, a few changes outside the `Tricore` subdirectory are necessary. The build system must be modified to respect the new target and a unique *target triple* must be defined that identifies the target machine and is consulted to select an appropriate backend. Moreover, to actually support compilation of C programs directly from source, integration with a C frontend is required.

4.11.1 Build System

LLVM includes configuration files for two universal build systems: the GNU Autotools and CMake. While the former constitute the de-facto standard for Unix platforms, support for the latter was introduced to enable compilation also on non-Unix systems like Windows. Another advantage of CMake is that it can generate not only plain Makefiles, but also project files for miscellaneous IDEs. To respect the TriCore target, the subdirectory must be entered into the configuration files of both build systems.

4.11.2 Target Triple Registration

A *target triple* is a string that uniquely identifies a target platform. It consists of three or four parts: achitecture, vendor, operating system, and environment (optional), e. g., `i686-pc-linux-gnu`. The relevant part for determining which backend should be used for code generation is the architecture. All valid architectures are listed in an enumeration, with an explicit mapping between an architecture string and the corresponding `enum` element via `switch`-statements and `if`-cascades. These code portions have to be extended manually to support the TriCore target triple.

Data type	Width (in bits)	Alignment (in bits)
`char`	8	8
`short`	16	16
`int`	32	32
`long`	32	32
`long long`	64	32
`float`	32	32
`double`	64	32
`long double`	64	32
`void *`	32	32

Table 4.3: Widths and alignments of C types on TriCore. These values are typical of a 32-bit platform.

Upon registration of the target machine, the respective enumeration element (i. e., `Triple::tricore` in the case of TriCore) is passed to the `TargetRegistry`. When it comes to selecting the appropriate backend for a given target triple, the lookup procedure tries to find a match among the registered targets.

4.11.3 Integration with the Clang Frontend

For an ideal portable programming language, the frontend would be completely agnostic about the target platform. Due to historical reasons, this is unfortunately not the case for C and C++ because the widths of data types vary from platform to platform. Thus the C/C++ frontend needs to know certain characteristics of each target architecture it supports.

Integrating the TriCore backend with LLVM-GCC would have required a considerable amount of additional work as well as extensive knowledge about the internals of GCC. For this reason, it was decided to support the Clang frontend instead. Following the design philosophy of LLVM's core, Clang is highly modular and easily extensible. In order to integrate the new target, it is necessary to create one additional class `TriCoreTargetInfo` that provides the required width and alignment information about data types and a few other details.

The decision of which `TargetInfo` subclass an instance has to be created is based on the target triple that is passed to the `clang-cc` executable via the command line with the switch `-triple=XXX`.

Files

Contrary to all previous sections, these paths are not relative to the `lib/Target/TriCore` subdirectory, but to the root of the LLVM source tree:

```
CMakeLists.txt
configure
autoconf/configure.ac
```

```
include/llvm/ADT/Triple.h
lib/Support/Triple.cpp
tools/clang/lib/Basic/Targets.cpp
```

4.12 Summary

LLVM's code generation process is organized as a series of passes that perform transformations on the program representation. The central stage is the DAG-based pattern matching instruction selector. Backend developers are provided with an extensive framework that relieves them of a considerable amount of work – in particular, all algorithms that are somewhat complex and possibly error-prone if re-implemented manually are written in a target-independent and reusable way. For instance, the instruction scheduling and register allocation passes require no additional implementation work at all.

The TableGen description language is a powerful tool that aids in representing descriptive information in a compact and formal manner. It is used widely throughout the backend to generate not only static data tables, but also several thousand lines of code for the instruction selector and assembly printer, consequently enormously reducing the risk of implementation bugs.

The great comfort however comes at the price of critical inflexibility if certain features necessitated unconditionally by the backend are not supported, as it was the case with the distinction between integers and pointers. The solution to this problem required several quite invasive modifications in the original code and a few somewhat unorthodox means. Nevertheless, due to its numerous benefits, the code generation framework has to be assessed as a very positive and helpful device.

5 Evaluation

This chapter discusses the state of the implemented backend, its features and shortcomings, and its general maturity. Moreover, it examines how a version of Clang with the TriCore backend integrated compares to the existing TriCore port of GCC.

5.1 General Evaluation

In general, the implementation of the TriCore backend is fairly complete in terms of features. Using the Clang frontend, it is possible to compile arbitrary C programs from source as all required basic capabilities are present:

- **Arithmetical and logical operations on all common data types.** All LLVM instructions on `i32` and `f32` variables, including pointer arithmetic, are supported directly and mapped to the equivalent TriCore instructions. Operands of the types `i8` or `i16` are extended to 32 bits and then processed by the respective `i32` counterpart of the instruction in question. 64-bit integers are split up into two halves and handled by a series of 32-bit operations – for instance, an addition is represented by an `addx` (*add with carry-out*) followed by an `addc` (*add with carry-in*). Operations for which the TriCore ISA offers no hardware support (e. g., `f64` arithmetic) are emulated in software through library calls.

- **Control flow instructions.** The TriCore backend supports conditional and unconditional branches as well as direct and indirect function calls with regard to the calling conventions. Cross-call lifetimes of registers are taken into account.

- **Some advanced optimizations.** Although there is still much room for further improvements, a few advanced compound instructions like `madd` (*multiply and add*) and bit manipulation instructions like `extr.u` can be selected. Moreover, some peephole optimization is done by the `TriCoreLoadStoreOpt` pass.

With the help of a simple Perl script that processes the command line arguments, it is possible to use Clang as a drop-in replacement for the existing TriCore GCC, provided that the appropriate assembler, linker, libraries, and header files are available.

The backend is however still lacking some features that would be desirable for the future:

- **Subtargets.** The backend could be extended to support the TriCore 2 architecture. Subtargets could be added to allow enabling and disabling optional hardware capabilities like the FPU that are not part of every CPU.

- **Debugging support.** While the backend per se is able to emit debugging directives into the assembly code, these directives are not in a format that can be handled by the existing assembler, which expects `.loc` lines instead of debug labels. LLVM is nominally able to generate such `.loc` pseudo-instructions, but currently aborts with a failed assertion due to a type contradiction. This is an issue that would have to be investigated and fixed by the LLVM developers.
- **Selection of SIMD vector instructions.** Although the TriCore architecture offers special SIMD arithmetic instructions for `v2i16` and `v4i8`, currently no support for them is implemented. This was considered a low priority because LLVM cannot auto-vectorize loops at the moment, thus compiling regular C source code will not result in the creation of any vector instructions.
- **DSP-specific CPU features.** As mentioned in the beginning, the TriCore architecture includes several specialized instructions and addressing modes for digital signal processing applications, for example saturated arithmetic or bit-reversed addressing for FFT computations. Many of them are pretty hard to represent in LLVM code and almost impossible in C code, and definitely not portable. If a programmer wishes to make use of these instructions in his application, he can easily resort to inline assembly.

5.2 Comparison to GCC

For every newly developed piece of software, it is desirable to analyse how it compares to established products. For a compiler, there are three relevant criteria that can be measured numerically:

1. Compilation speed
2. Size of the emitted machine code
3. Performance of the generated program

The TriCore LLVM backend was lined up against HighTec's TriCore port of GCC 3.3 mentioned at the beginning. It was built against the latest SVN trunk sources of LLVM and the Clang frontend. The CoreMark benchmark was chosen for the comparison.

5.2.1 The CoreMark Benchmark

CoreMark is a benchmark developed by the Embedded Microprocessor Benchmark Consortium and published in version 1.0 in June 2009. It was conceived as a modern alternative to the popular and widespread Dhrystone benchmark developed in 1984. Although CoreMark is occasionally referred to as "open source software" because it is available in source code form and free of charge, this term is inappropriate due to certain profound licence restrictions [6].

```
2K performance run parameters for coremark.
CoreMark Size    : 666
Total ticks      : 59219179
Total time (secs): 14.0998050
Iterations/Sec   : 28.3691880
Iterations       : 400
Compiler version : GCC4.2.1 Compatible Clang Compiler
Compiler flags   : -O3 -DPERFORMANCE_RUN=1  -Xlinker -T -Xlinker
tricore/tricore.ld
Memory location  : STACK
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0x4983
Correct operation validated. See readme.txt for run and reporting rules.
CoreMark 1.0 : 28.3691880 / GCC4.2.1 Compatible Clang Compiler -O3
-DPERFORMANCE_RUN=1  -Xlinker -T -Xlinker tricore/tricore.ld / STACK
```

Figure 5.1: Output of a CoreMark run. The number of iterations completed per second is used as the benchmark score.

Written in C, CoreMark is designed to be small and easily portable to a multitude of platforms, including embedded microcontrollers. It aims to prevent compilers from "cheating" by optimizing away computations because of constant input data or unused results. Furthermore, no calls to library functions are made during time measurement because they might severely distort the timing results. The Coremark benchmark performs the following actions:

- linked list processing (find, sort, read),
- matrix manipulation,
- state machine operations,
- and CRC computation.

Before the actual benchmarking, a one-second "warm-up" phase is initiated. Based on the number of iterations completed during this interval, CoreMark determines how many iterations must be used for the time measurement run so that it takes at least ten seconds.

The associated benchmarking rules define two sets of fixed seed values (for a "performance run" and a "validation run") that are passed to the program as input parameters. For both runs, the CRC checksums of the results are matched with reference values in order to verify the correctness of the benchmark. The score that is printed out as the final result is computed by dividing the number of completed iterations by the number

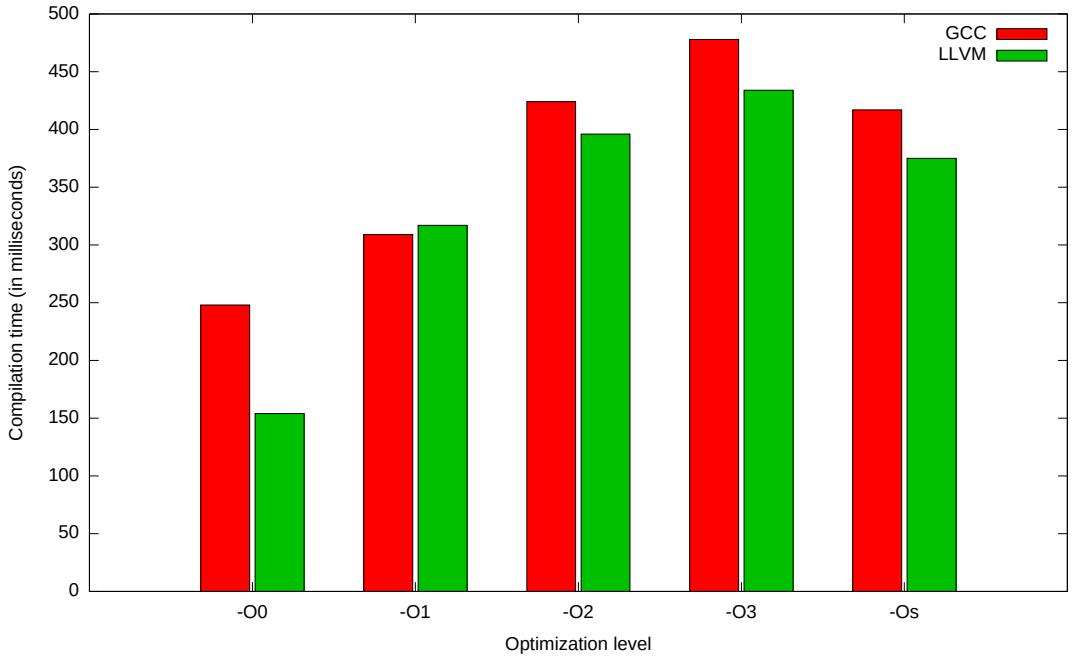

Figure 5.2: Compilation times for the CoreMark application on a Core 2 Quad Q6600 (2.40 GHz). With the exception of `-O1`, LLVM works around 10 % faster than GCC when optimizations are enabled. Without optimizations, compilation takes almost 40 % less time.

of seconds the benchmark took to finish. Figure 5.1 shows the complete output of a CoreMark run.

All tests were performed separately for five different optimization settings: none (`-O0`), performance (`-O1` to `-O3`), and code size (`-Os`).

5.2.2 Compilation Speed

The first test measured the time it took to build all eight CoreMark source files. Only the compilation from C to assembly code was covered, not the subsequent assembling and linking stages. The test was run on a PC with a Core 2 Quad Q6600 processor. Compilation times were measured ten times for each configuration, with the lowest result being taken. The results are shown in Figure 5.2.

The higher the chosen optimization level is, the more additional analysis and transformation passes are run inside the compiler, thus a strong correlation to the build times can be noticed. LLVM obviously manages to live up to its aspirations to work in a highly efficient way. It outperformed GCC in four out of five configurations, equalling only at `-O1`. In average, Clang finished about 10 % earlier than GCC – with optimizations disabled, it was even almost 40 % faster.

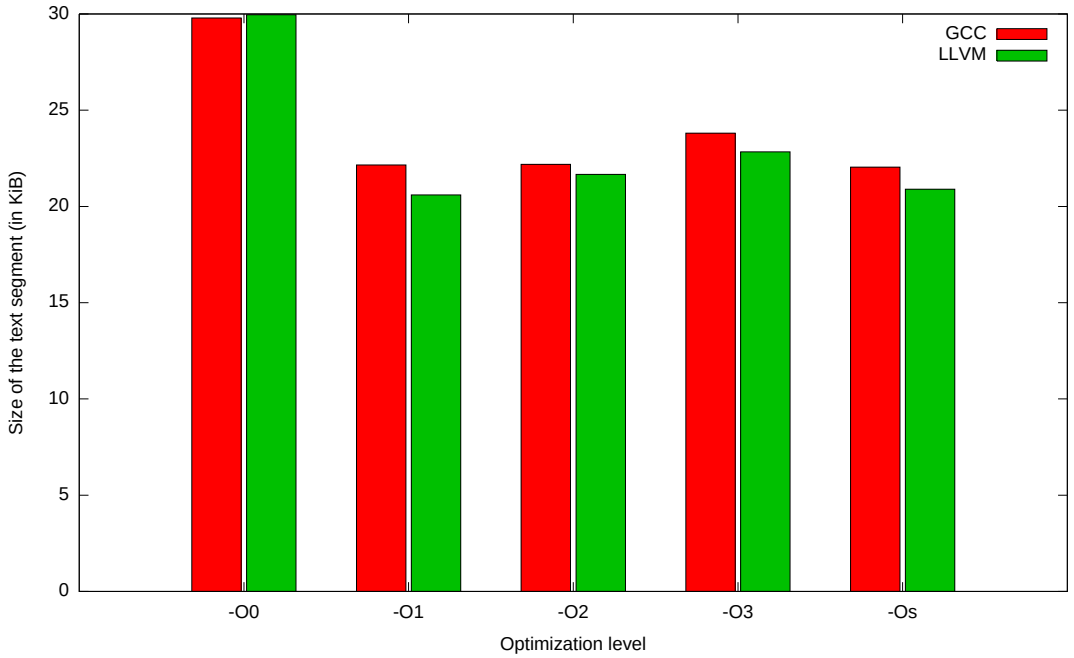

Figure 5.3: Sizes of the text segment in the resulting ELF binaries for the CoreMark benchmark, determined with the aid of the `tricore-size` utility. The code emitted by LLVM has the same or a slightly smaller size (up to 7 %) than the output of GCC.

5.2.3 Code Size

For the second test, the generated ELF binaries were examined with the `tricore-size` tool to compare the text segment sizes. As Figure 5.3 indicates, the size differences were rather small, albeit with a slight advantage of at most 7 % for LLVM. When compiling without optimizations, clearly a primitive register allocation algorithm is used: Local variables are always accessed via the stack and of a lot of spill code is generated, which has a huge effect on the code size. With an increasing degree of optimization, the text segment becomes marginally larger again, mostly due to function inlining.

5.2.4 Code Performance

The third and most important criterion for the comparison was the performance of the generated machine code. For this purpose, the CoreMark benchmark was run on a TriCore TC1796 board clocked at 40 MHz with a connected hardware debugger. The text and data segments were put into the on-chip SPRAM (*scratch-pad RAM*) and LDRAM (*local data RAM*), respectively, to ensure that execution times were not dominated by access latencies of external memory.

All tests were run twice with “performance” seeds and twice with “validation” seeds. As it turned out, the deviation between the results of two runs with the same configuration and the same input data was infinitesimally tiny at best. The difference between two

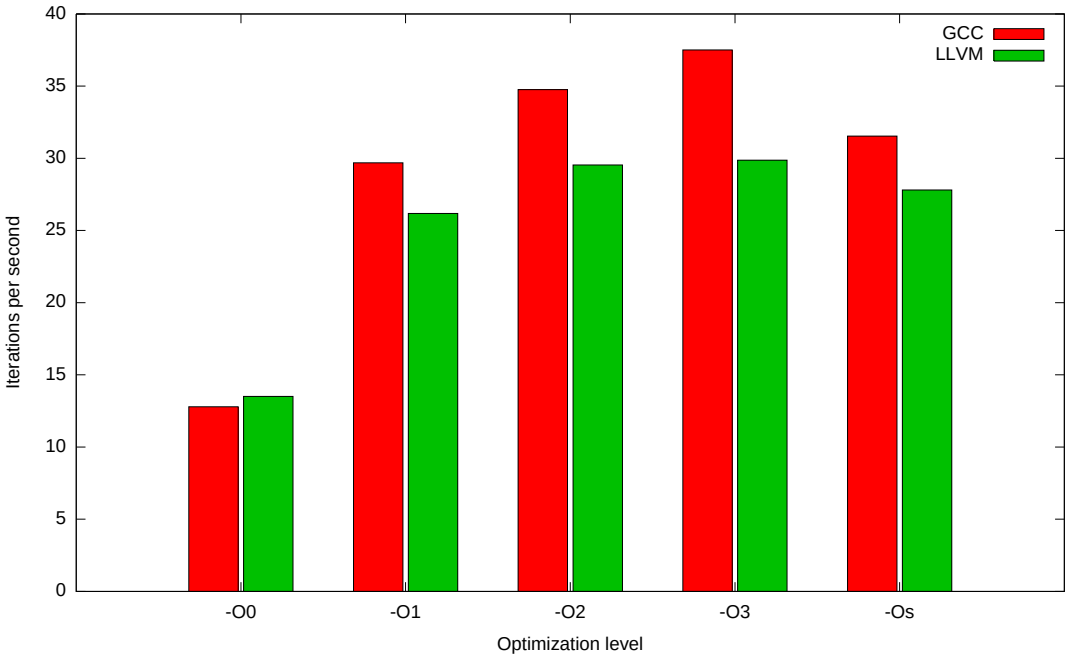

Figure 5.4: CoreMark scores on a TriCore TC1796 microcontroller running at 40 MHz. The TriCore LLVM backend cannot yet fully compete with GCC in terms of code efficiency. Scores are between 12 and 20 % lower – with the exception of `-O0`, where the binary created by LLVM actually performs 6 % faster.

runs with the same configuration, but different input data, was no greater than 0.5 %. This is mainly due to two reasons:

1. The benchmark performs a sufficient number of iterations (at least 200 iterations for all tests in question), consequently virtually eliminating any potential fluctuations.

2. Unlike benchmarks executed on multitasking systems such as PCs, the CoreMark program ran exclusively on the TC1796 hardware, without the possibility of other processes negatively affecting performance.

As Figure 5.4 illustrates, the implemented TriCore backend is not yet fully competitive with GCC when it comes to code performance. Scores are around 12–15 % lower for the "regular" optimization levels. Interestingly, the improvement of `-O3` over `-O2` is negligible for LLVM whereas GCC manages to yield additional speed at `-O3`, increasing the gap to 20 %. Even so, when no optimizations are enabled, the code generated by LLVM performs slightly better than the output of GCC.

As LLVM has proven to be more or less on par with GCC in terms of code efficiency on other platforms, it is clearly the backend that is to blame for the performance loss. On the one hand, this is most probably at least partly caused by the integer/pointer problem described beforehand. While the existing solution works reasonably well, it is

still not refined enough to cover all cases correctly. There are still some situations where the type annotation is not propagated properly or, conversely, an integer is erroneously treated as a pointer. In any event, improvements in this area would require more and deeper modifications to the the LLVM codebase that would go well beyond the scope of this thesis.

On the other hand, there is quite some potential for optimizations in the backend itself. The following is an incomplete list of possible advancements:

- **Improved instruction selection.** The TriCore ISA offers a lot of special and composite instructions that correlate to non-trivial code patterns. For instance, the `abs` (*absolute*) and `cadd` (*conditional add*) instructions would be represented in LLVM code by a case differentiation and consequently cannot be selected directly. Another example are bit manipulation and extraction instructions such as `extr.u`. The code that selects such instructions would have to be written in C++, as it would be too complex to be expressed in the TableGen language in most cases.

- **Better branch analysis and transformation.** The current branch analysis implementation can only handle unconditional branches. Extending it to support conditional branches would be unproblematic, but would require some additional lines of code.

- **Tail call optimization.** If the very last instruction before a `ret` is a `call`, this `call` could be converted into an unconditional branch, eliminating the overhead of implicit context saving [14]. Implementing support for this should be fairly straightforward.

5.3 Summary

While the TriCore backend is relatively complete with respect to features and ready for regular use within a certain scope, the performance of the generated machine code is not yet on par with the code emitted by GCC. This is partly due to the omnipresent integer/pointer problem, but also shows that quite a lot of additional sophisticated refinement is necessary to reach the level of an established compiler.

6 Conclusion

In the scope of this thesis, a new TriCore backend for the LLVM compiler infrastructure was designed, implemented, and evaluated. While many aspects of the design and structure of the backend were dictated by LLVM's comprehensive code generation framework, a fair amount of thought was necessary to address a number of inherent problems, including selective modifications to parts of the framework.

Thanks to LLVM's extremely modular nature, the TriCore backend can be easily incorporated into any software constructed upon the LLVM infrastructure. Integration into the Real-Time Systems Compiler has already been done and was more or less trivial as it necessitated almost no changes to the backend code. On the RTSC side, a subclass of the assembly printer had to be created that prints out the additional information generated by the RTSC extension.

In combination with the Clang frontend, the implemented code generator can compile and optimize arbitrary C programs from source into TriCore assembly code. Adhering to one of LLVM's fundamental paradigms, the compiler is surpassingly fast and produces compact code. However, although its numerous advanced optimization passes make LLVM very competitive in terms of code performance, the comparative measurements have shown that the TriCore backend still requires some additional tuning effort if it is to play in the same league as existing compilers such as the GCC implementation.

6.1 Outlook

If further development is pursued consequently, the TriCore backend in combination with Clang might emerge as a serious alternative to the established compilers for the TriCore architecture. It would be the first TriCore compiler released under a BSD-style licence, which might make it an attractive choice because of its greater liberality compared to the GNU GPL. With LLVM constantly picking up pace and making rapid progress, it will be very exciting to see what the future holds.

Bibliography

[1] Aaron N. Gray et al., *Direct Object Code Emission*, `http://wiki.llvm.org/Direct_Object_Code_Emission`, July 2009.

[2] Aditya Parameswaran et al., *Tricore Port for GCC – An Analysis*, `http://www-cs-students.stanford.edu/~adityagp/acads/tricore-bit-var.pdf`, April 2006.

[3] Alexey Smirnov et al., *GNU C Compiler Internals*, `http://en.wikibooks.org/wiki/GNU_C_Compiler_Internals/Print_version`, February 2007.

[4] Chris Lattner et al., *The LLVM Target-Independent Code Generator*, `http://llvm.org/docs/CodeGenerator.html`, August 2009.

[5] Peter Hoogenboom, *TASKING helps Siemens with 32-bit TriCore architecture design*, Embedded Systems Programming Europe (1997).

[6] Open Source Initiative, *The Open Source Definition*, `http://opensource.org/docs/osd`, August 2009.

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

[8] ———, *TableGen Fundamentals*, `http://llvm.org/docs/TableGenFundamentals.html`, August 2009.

[9] Randall Munroe, *Compiling*, `http://xkcd.com/303/`, August 2007.

[10] University of Illinois/NCSA, *LLVM Release License*, `http://llvm.org/releases/2.5/LICENSE.TXT`, August 2009.

[11] The FreeBSD Project, *FreeBSD Quarterly Status Report*, `http://www.freebsd.org/news/status/report-2009-01-2009-03.html`, January 2009.

[12] Fabian Scheler, Martin Mitzlaff, Wolfgang Schröder-Preikschat, and Horst Schirmeier, *Towards a real-time systems compiler*, Proceedings of the 5th International Workshop on (Intelligent Solutions in Embedded Systems (WISES '07), IEEE Computer Society Press, June 2007, pp. 62–75.

[13] Infineon Technologies, *TriCore 1.3 Architecture Overview Handbook*, 81726 Munich, Germany, May 2002 ed., May 2002.

[14] _____, *TriCore Compiler Writer's Guide*, 81726 Munich, Germany, 2003-12 ed., December 2003.

[15] _____, *Volkswagen and Infineon Find New Ways to Integrate the Telephone Into Cars*, `http://infineon.com/cms/en/corporate/press/news/releases/2004/132095.html`, September 2004.

[16] _____, *TriCore EABI User's Manual*, 81726 Munich, Germany, 2007-02 ed., February 2007.

[17] _____, *Infineon and BMW M GmbH Are Cooperating To Develop A High-Performance Engine Control Unit For The Next BMW M Series*, `http://infineon.com/cms/en/corporate/press/news/releases/2008/INFAIM200802-037.html`, February 2008.

[18] _____, *TriCore 1 Architecture Volume 2: Instruction Set V1.3 & V1.3.1*, 81726 Munich, Germany, 2008-01 ed., January 2008.

[19] _____, *Integrated Compiler Development Environment*, `http://infineon.com/cms/en/product/channel.html?channel=ff80808112ab681d0112ab6b55d607d9`, August 2009.

[20] _____, *TriCore 2*, `http://infineon.com/cms/en/product/channel.html?channel=db3a304312bae05f0112be54d22c00e6`, July 2009.