Alexandru Agape, BSc

Accelerating the Development of an LLVM Compiler Backend

MASTER'S THESIS

to achieve the university degree of
Diplom-Ingenieur

Master's degree programme:
Information and Computer Engineering

submitted to
Graz University of Technology

Supervisor:
Dipl.-Ing. Dr. techn. Michael Krisper, BSc
Institute of Technical Informatics, Graz University of Technology
Area of Cognitive Products, Pro2Future GmbH

Advisor:
Vignesh Manjunath, M.Tech.
Area of Cognitive Products, Pro2Future GmbH

Graz, May 2025

Acknowledgements

I want to express my gratitude to my advisor, Vignesh, for his invaluable guidance, support and encouragement during the implementation of this thesis. His expertise and insights have been instrumental in shaping this work.

I am also grateful to Michael Krisper for his invaluable guidance throughout the process of writing this thesis. His support and advice were crucial in helping me navigate the challenges of structuring and documenting my work effectively. I would also like to thank Mr. Krisper for teaching the course on "*Design Patterns*" in a way that inspired my passion for clean code and architecture design. His approach to the subject has had a lasting impact on how I approach software development and problem-solving, and it remains my favorite course from my Master studies.

I would also like to thank TU Graz for providing an excellent study environment that fostered learning, collaboration, and personal growth throughout my academic journey.

Special thanks to the LLVM community for their extensive documentation and open-source contributions, which served as a foundation for this work.

Finally, I want to thank my family, friends, colleagues, and especially my partner Teodora, for their unwavering support and patience. A special shoutout to all of you who had to hear me say, for the better part of two years, "I am 6 months away from finishing my Master's!" Your encouragement and understanding have been a constant source of motivation.

Graz, May 2025
Alexandru Agape

Abstract

Developing custom backends for the LLVM compiler infrastructure is a complex and time-intensive process, requiring a deep understanding of both the target architecture and LLVM's internal structure. This thesis addresses these challenges by introducing the **LLVM Backend Development Kit** (LBDK) and the **LLVM Development Environment** (LDE) tools designed to simplify and accelerate backend development. The LBDK provides a reusable backend template and a suite of scripts that automate the creation of the LDE. These scripts integrate the template backend into LLVM's build system and streamline the development workflow, making the process more efficient and manageable.

To demonstrate the framework's effectiveness, a proof-of-concept backend was implemented for the TriCore TC1.6 architecture, showcasing how the LBDK reduces the complexity of backend creation. The evaluation highlights significant improvements in developer efficiency, enabling a focus on target-specific implementation rather than boilerplate setup. Additionally, this thesis contributes a detailed developer guide, bridging gaps in existing resources by offering a clear, structured, and practical approach to LLVM backend development.

Contents

1 Introduction **1**
- 1.1 Context and Domain 1
- 1.2 Motivation 2
- 1.3 Goal 3
- 1.4 Repository 4
- 1.5 Structure of the Thesis 4

2 Background and Related Work **5**
- 2.1 Compiler Construction 5
- 2.2 Tooling and Templates 22
- 2.3 TriCore Architecture 25

3 Architecture and Design **27**
- 3.1 LLVM Backend Development Kit 27
- 3.2 LLVM Development Environment 32

4 Implementation **35**
- 4.1 LLVM Backend Development Kit 35
- 4.2 LLVM Development Environment 42
- 4.3 Process of developing an LLVM Backend 46

5 Contribution: Developer Guide **51**
- 5.1 How to use the framework 51
- 5.2 TriCore LLVM Backend - Proof of concept 52

6 Evaluation **63**
- 6.1 Evaluation of Workflow Streamlining for Backend Development 63
- 6.2 Compilation results for the TriCore LLVM Backend 65

7 Conclusion and Future Work **73**

Bibliography **77**

CHAPTER 1
Introduction

The Universe is under no obligation to make sense to you.

Neil deGrasse Tyson

Compilers have become an essential tool for developers since the 1950s when Grace Murray Hopper developed the FORTRAN compiler. It allowed a programmer to use a problem-oriented source language. The LLVM project is a widely used compiler infrastructure that provides a set of tools and libraries for constructing compilers. It is a versatile framework that allows developers to create frontends for any programming language, as well as custom backends for various hardware targets. However, developing a custom backend within LLVM is a complex task that requires a deep understanding of the target architecture, as well as the LLVM framework itself. The process of creating a custom backend involves writing a significant amount of boilerplate code, which can be time-consuming and often laborious. This thesis aims to address this issue by providing a reusable template that can speed up the initial setup of an LLVM development environment. The template includes a set of scripts that automate the process of creating a new backend within LLVM, allowing developers to focus on the specifics of the target architecture. A proof of concept is provided in this thesis to demonstrate the effectiveness of the template, This consists of an example where the template is employed to develop a minimal custom backend for the TriCore TC1.6 architecture that can produce valid assembly code for this architecture.

1.1 Context and Domain

This thesis was written at the Institute of Technical Informatics at Graz University of Technology and Pro2Future GmbH in the context of the CompEAS-BSW project (Compositional Embedded Automotive Software — Basic SoftWare) (FFG Contract No. 911655). The goal of the project was to research and develop compositional automotive software that is maintainable, correct, and compliant with all requirements.

No specific domain was targeted in this thesis; however, for the use-case, a TriCore-CPU was chosen because it is the prevalent choice in the automotive industry, which was the domain of the overall project.

1.2 Motivation

Writing a custom LLVM backend for any CPU architecture is intricate and time-intensive. It demands a profound understanding of the target architecture, knowledge of compiler design principles, as well as a deep understanding of LLVM's architecture and APIs. In general, developing an LLVM backend spans several months to years of work, depending on the complexity of the target and the level of experience of the developers, as well as the size of the team.

Understanding the target architecture, getting comfortable with LLVM's architecture and APIs, and implementing the necessary components for a functional backend contribute to the duration, reflecting the complexity of the process.

Some of the main challenges that make writing an LLVM backend difficult include:

- **Deep familiarity with the target architecture**: Developers must understand low-level details of their CPU architecture, like instruction formats, calling conventions, or register usage.
- **Knowledge of general compiler design principles**: This includes reading about frontends, intermediate representations, optimization phases, and code generation strategies.
- **Good understanding of LLVM internals and APIs**: The LLVM codebase is large, modular, and highly complex, introducing a steep learning curve for new developers aiming to build solutions on top of it.
- **Lack of practical documentation**: The available documentation is often either too high-level to be practically useful or too low-level, overwhelming developers with intricate code details and lacking clear guidance.
- **Reliance on existing backends as references**: The official LLVM documentation [1] encourages copying from existing backends, but doing so effectively requires prior experience to know what parts of the code to keep, remove, or adapt.

These challenges illustrate the motivation behind this thesis: to simplify and support the LLVM backend development process by addressing some key difficulties faced by developers.

1.3 Goal

The primary goal of this thesis is to lower the barrier to entry for developers aiming to implement custom LLVM backends. This is achieved by streamlining the initial setup process, reducing boilerplate effort, and providing a structured development guide. To support this goal, we introduce a templated backend implementation, called *Skeleton*, along with a suite of scripts that generate an **LLVM Development Environment (LDE)**. The **LDE** helps developers get started more quickly, enabling them to focus on implementing target-specific logic rather than spending time configuring the build system or reverse-engineering existing backends, which is the currently recommended way in the documentation [1].

In addition, the thesis provides a dedicated developer guide that walks through the backend creation process step by step. This guide is designed to be practical and hands-on, offering insight into how to use the provided tools to implement a functional LLVM backend from scratch.

1.3.1 Research Questions and Hypothesis

Research Questions

1. How can a modular and reusable LLVM backend template be created?
2. Which tools are needed to accelerate the initial setup process and streamline the development process of the backend?
3. How can the initial knowledge gap and learning curve for the developers be reduced?

Hypotheses

1. A modular and reusable LLVM backend template can be created by leveraging LLVM's modular design, making it applicable for any CPU architecture.
2. The tools provided in this thesis will accelerate the initial setup process and streamline the backend development process by automating common tasks and reducing boilerplate code.
3. A dedicated developer guide will significantly improve the accessibility of the backend development process, making it easier for new developers to create custom LLVM backends.

1.4 Repository

The source code that accompanies this thesis can be found in the following repositories:

- **LLVM Backend Development Kit**
  https://github.com/agape94/llvm-backend-template/

- **TriCore LLVM Backend**
  https://github.com/agape94/llvm-tricore-backend

1.5 Structure of the Thesis

The remainder of this thesis is structured as follows:

- **Chapter 2** presents the **background and related work** relevant to this thesis. It covers key topics such as general compiler design and considerations, LLVM as a modern framework to build compilers, as well as an overview of the TriCore TC1.6 architecture, which serves as the target for our proof of concept.

- **Chapter 3** discusses the **architecture** and **design**. It explains the main design of the solution proposed in this thesis.

- **Chapter 4** explains the practical contributions of this thesis by showing and discussing the **implementation**.

- **Chapter 5** represents one of the **contributions** of this thesis. It serves as a **guide for developers** who want to develop their own backend.

- **Chapter 6 evaluates** the usability of our framework and also presents an evaluation of our proof of concept.

- **Chapter 7** elaborates on the open issues and **future work**, as well as some **final thoughts** and takeaway messages.

CHAPTER 2

Background and Related Work

If I have seen further it is by standing on the shoulders of giants.

Isaac Newton

This chapter provides the background knowledge relevant to this thesis and outlines the related work associated with the various topics discussed. This includes the topics of compiler technology with a focus on LLVM (more specifically CodeGen, TableGen and the general backend creation process), as well as an introduction to the TriCore TC1.6 architecture. Additionally, the state of the art concerning code generation tools is discussed, followed by a short discussion where we explain the decision to implement a custom code generation solution instead of using an existing one.

2.1 Compiler Construction

Programming languages serve as notations for describing computations to both humans and machines. They are essential to modern life, as every piece of software operating on a computer is created using one. However, before a program can actually run, it must be converted into a format that the computer can understand. This task is handled by specialized software known as *compilers*. In other words, a compiler is a program that takes code written in one language (the *source* language) and transforms it into a corresponding program written in another language (the *target* language). [2, 3]. For example, `GCC` compiles C or C++ programs into executable machine code, typically targeting architectures like X86. The term *to compile* was first used in the context of programming in the early 1950s, notably with Grace Hopper's development of the **A-0** system in 1952, which is considered one of the first tools to translate symbolic mathematical code into machine code, laying the groundwork for modern compilers [4, 5, 2].

2

2.1.1 Typical compiler architecture

To convert source code into assembly code, a compiler follows a series of stages, as illustrated in Figure 2.1. The core parts of a compiler include the **Frontend**, the **Optimizer**, and the **Backend**, which are connected through the **Intermediate Representation** (IR). The IR is designed to be simple and consistent, making it easier to perform optimizations, conduct analyses, and generate efficient machine code. It typically combines characteristics of the high-level source language with elements of the lower-level assembly language. [2, 3].

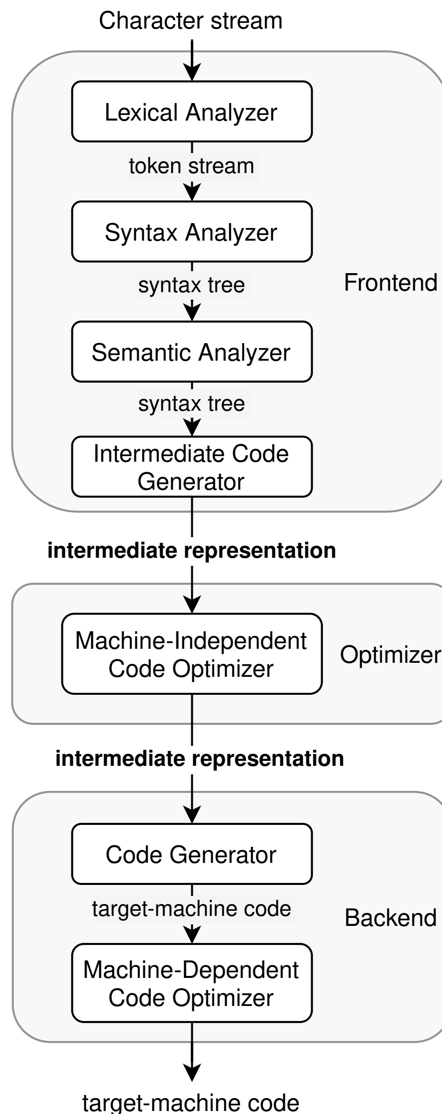

Figure 2.1: Structure of a generic compiler. This figure was redrawn from [2].

1. **Frontend**: The Frontend of the compiler reads the program written in the source language and translates it into the Intermediate Representation (IR) [2].
2. **Machine-Independent Code Optimizer**: High-level language constructs often introduce substantial run-time overhead if they are translated into machine-code. This component of the compiler eliminates these inefficiencies and prepares the code for the final stage, which is **Target-Machine Code Generation** [2, 3].
3. **Backend**: The final stage of the compilation is the **Target-Machine Code Generation**, which is performed by the Backend. The Backend takes as input the IR and outputs target-machine code [2, 3].

2.1.2 LLVM Compiler Infrastructure

The components of a compiler should be stand-alone and reusable, however, in practice, this is rarely the case [6]. For example, it is difficult to reuse certain parts of the GCC compiler, because the components are tightly coupled in the way they interact with each other, this being one of the motivations to develop new solutions [6].

Low Level Virtual Machine (LLVM) is a suite of modular and reusable compiler and toolchain technologies, originally introduced by Chris Lattner in 2002 [7]. Despite what its name suggests, LLVM has little to do with traditional virtual machines. Although "LLVM" initially was an acronym, it has since evolved into the brand name of the overall project. LLVM began in December 2000 as a group of modular libraries designed to work together through clearly defined interfaces. At the time, many open-source programming language tools were created as large, single-purpose programs, which made it difficult to reuse individual parts. For instance, reusing a parser from a static compiler like GCC for tasks such as static analysis or code refactoring was very challenging [7, 6]. LLVM addressed this limitation by separating the frontend and backend components of the compiler, connecting them through LLVM's **Intermediate Representation** (IR), a virtual instruction set resembling RISC architectures [8].

2.1.2.1 LLVM Architecture

LLVM is based on the traditional three-stage compiler design, which involves three main components mentioned in Section 2.1.1: the frontend, the optimizer, and the backend, as shown in Figure 2.2. The frontend takes in the source code and translates it into the LLVM Internal Representation (IR), which is then processed by the optimizer to apply various optimizations. The modified IR code is then passed to the backend to generate target-specific assembly or machine code that can be flashed and executed on the desired architecture [7, 6].

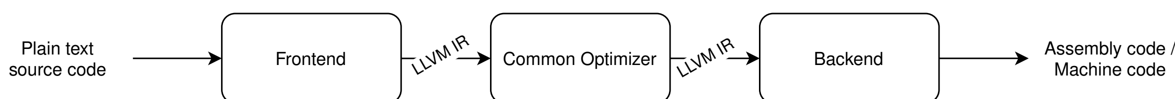

Figure 2.2: Basic LLVM Architecture based on a three-stage design. The figure was adapted from [6].

Although LLVM is recognized for its unique features and valuable tools—such as the Clang compiler for C, C++, and Objective-C, which offers several advantages over GCC—its most distinguishing characteristic is its **internal architecture** [6]. To divide the compiler into multiple tools, LLVM enforces a strict separation of its components into independent libraries that are dynamically linked. This design is supported by LLVM's use of C++ with object-oriented programming principles and a modular pass interface, enabling easy integration of transformations and optimizations throughout the compilation process [9].

Thanks to the clear separation of its core components, with the code optimizer positioned centrally, LLVM achieves one of its greatest advantages: retargetability. As a modular and retargetable compiler framework, LLVM adopts a traditional three-layered architecture, consisting of multiple language-specific frontends, a centralized intermediate code optimizer, and a range of target-specific code generators. The LLVM IR is the "glue that holds the system together" [7], as it enables the decoupling between the components, but also ensures compatibility between them. Because the frontend and backend are not tightly coupled, this gives the compiler developers the possibility to combine any frontend with any backend with little to no development effort. As Figure 2.3 shows, if a compiler decides to support another programming language, a new frontend has to be developed for it, however, the backend and the optimizer can be reused [6].

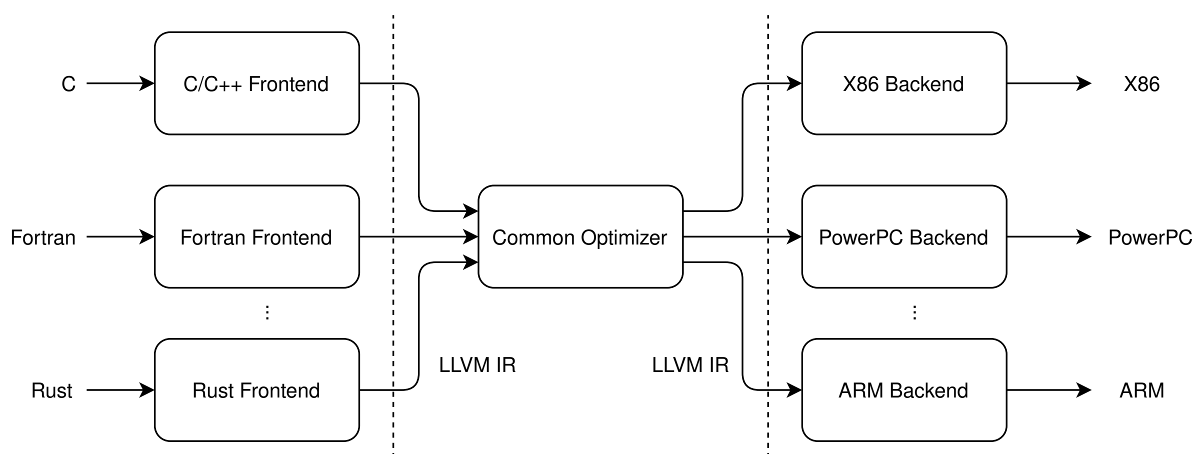

Figure 2.3: Retargetable LLVM Architecture. The figure was adapted from [6].

Another big advantage of LLVM's design, which stems directly from retargetability, is that it makes it easier for a wider range of programmers to contribute to its development. Instead of working with just one source language and one target, LLVM supports many, which helps attract more people to use and improve it. This is especially important for an open-source project because having a larger community leads to faster improvements and more new features. This is one reason why open-source compilers like GCC, usually produce better-optimized machine code than more limited compilers like FreePASCAL [6, 8].

2.1.2.2 TableGen

TableGen is a domain-specific language used in LLVM to define and generate structured data for various compiler components, such as instruction sets, register descriptions, and calling conventions. It allows developers to write declarative specifications, which LLVM then processes to generate C++ code, reducing redundancy and improving maintainability [10, 6]. LLVM uses TableGen in several key areas:

- Target Descriptions: Defines instructions, registers, and assembly syntax for different architectures.
- Instruction Selection: Helps map high-level IR operations to machine instructions.
- Calling Conventions: Specifies function argument passing rules.

The *TableGen files* (`.td` files) are processed by the `tblgen` utility, which generates C++ code used throughout LLVM, making it easier to support multiple targets and optimize compiler infrastructure efficiently [11]. Because TableGen is used to define the particularities of the target architecture, it is important to assess its usability. For this purpose, Schlamecher et al. presented in 2024 an experience report for extending Clang/LLVM with custom instructions using TableGen [12]. The authors present an automated approach for integrating custom instructions into the Clang/LLVM toolchain using `CoreDSL`, a domain-specific language. Their method generates the necessary TableGen and C++ code from CoreDSL descriptions, significantly reducing manual effort and minimizing invasive changes to the compiler. Focusing on RISC-V as a case study, they demonstrate seamless support for custom instructions across the compiler, linker, and debugger. While centered on RISC-V, the approach is generalizable to other architectures, offering a practical solution for extending LLVM-based toolchains.

2.1.3 Compiler Frontend

As stated previously, the Compiler Frontend is responsible for reading and translating the *source* language into the intermediate representation. As Figure 2.1 shows, the Frontend consists of the following components:

- The Lexical Analyzer, also called the Scanner, reads the plain text of the program and groups individual characters into complete tokens [2, 3].
- The Syntax Analyzer, or Parser, takes the tokens and organizes them into full statements and expressions. It uses a grammar that defines the formal rules for how language elements fit together. The parser produces an **abstract syntax tree (AST)**, which captures the program's structure and also tracks where each part came from in the original source file [2, 3].
- The Semantic Analyzer examines the AST to gather more meaning about the program. After this step, the AST is transformed into an **Intermediate Representation (IR)**, a simplified form of assembly code [2, 3].

Compiler frontends have been the subject of substantial research throughout the years. In the next few paragraphs we highlight a few representative examples.

In 2014, Bokan et al. [13] adapted the GCC frontend for embedded processors, using the RTCC compiler library. Their approach compiles C code to intermediate representation (IR), then hands it off to RTCC for further compilation. The study shows that adapting GCC for embedded systems requires deeper front-end changes to address the constraints of embedded and DSP environments, contributing to open-source front-end development for specialized compilers.

Nițu R. et al. [14] analyzed third-party development tools to identify key requirements for improving the D compiler library. They found that symbol resolution, scope retrieval, and error handling were essential features. These were integrated into the D reference compiler's frontend, replacing custom logic in external tools, with no performance penalty for either the compiler or the tools using its standardized interface.

Sana et al. [15] introduced *GLE* (Global, Local, and Expression-level) parsing, a three-phase technique designed to improve syntax error messages for students. By separating code analysis into functional structures, control structures, and expressions, GLE parsing provides clearer feedback. In a study with 51 undergraduates, GLE significantly helped students identify syntax errors, though it didn't reduce the time taken to fix them.

In the context of LLVM, Osmialowski [16] provided an overview of Flang, an open-source Fortran frontend for LLVM developed by PGI® and announced in May 2017. Flang aims to become a fully integrated component of the LLVM ecosystem, similar to Clang. The paper presents Flang's internal workings and source code structure, providing insights for developers interested in extending the frontend.

Reedy [17] discusses the development of a new LLVM-based backend for the Scala compiler, aiming to compile Scala code into optimized native executables. By using LLVM as an intermediate representation, the backend can utilize LLVM's program transformations, analyses, and both ahead-of-time and just-in-time compilation. The paper

outlines the backend's design and the challenges involved in building a fully-geatured Scala backend.

2.1.4 Optimizer

The Optimizer is a component responsible for improving the Intermediate Representation (IR) of a program to make it more efficient, without changing its behavior. It performs target-independent optimizations, many of which have been the subject of substantial research. In this section, we highlight a few representative examples from the literature. Aho et al. [2] identified two types of optimizations that can be performed by the optimizer: *Local code* optimizations and *Global* optimizations.

Local optimizations are code improvements that are performed within a basic block (such as function bodies), obtaining substantial improvements in the runtime of the code. A few examples of such optimizations are presented below:

1. Eliminate *local common subexpressions*: The instructions that compute a value that has already been computed are eliminated [2].
2. Eliminate *dead code*, representing instructions that compute values that are never used [2].
3. Reorder statements that don't depend on one another as it may result in faster execution or it may reduce the time a temporary value needs to be saved in a register [2].
4. Simplify computations by reordering operands of three-address instructions using algebraic laws [2].

In LLVM, the Optimizer applies various improvements to the LLVM Intermediate Representation (IR). The IR acts as the shared interface between the frontend and backend components developed within the LLVM project. It is designed to be *abstract* enough for high-level frontends to generate it easily, while also being *simple* enough to be translated into assembly or machine code for specific CPU architectures [18]. Frontends are responsible for producing the IR, and backends for consuming it. Most of LLVM's target-independent optimizations are performed at the IR level [9].

The LLVM IR language is an assembly-like language that uses the *single static assignment* format, meaning that any variable is assigned to once. This is a very important feature of the language, as it allows for certain optimizations that would have been cumbersome to implement otherwise. It is also important to note that LLVM IR is not a stable language, meaning that backwards compatibility is not guaranteed between different versions of LLVM IR [19].

With this context in place, the following paragraphs discuss notable contributions from the literature that focus on optimizer design and implementation strategies.

In 2021, Khaldi et al. [20] introduced `DPC++` to extend LLVM IR for efficient matrix operations, specifically targeting `Intel® AMX` (Advanced Matrix Extensions). They propose IR modifications to leverage AMX's capabilities, improving performance in matrix-intensive tasks. A case study within the `DPC++` programming model demonstrates these extensions, contributing to better hardware utilization and potential performance gains in high-performance computing.

Junod et al. [21] introduced OLLVM, a software obfuscation tool built on the LLVM framework to improve software security by complicating reverse-engineering. It works on LLVM's Intermediate Representation (IR) and offers a language-agnostic, architecture-independent solution for code obfuscation. The tool replaces simple instructions with complex or uncommon alternatives to hinder static analysis without changing functionality. Though intended as a prototype, OLLVM demonstrates the potential of LLVM IR-level manipulation, applicable across different languages and architectures.

Jingu et al. [22] proposed an automatic loop parallelization method at the LLVM Intermediate Representation (IR) level in 2018. They developed an LLVM compiler pass that generates parallel code from IR directives. Using the PolyBench/C benchmark suite, the parallelized programs achieved performance improvements of up to 3.45x while maintaining correctness. This work demonstrates the potential of IR-level parallelization, particularly for legacy or proprietary systems where source code access is limited.

In 2023, Yintong et al. [23] studied the factors affecting the effectiveness of LLVM's Intermediate Representation (IR) for binary code similarity detection. They analyzed the impact of different compilers, optimization levels, architectures, and obfuscation strategies. Their findings show that LLVM IR optimization works best with GCC or Clang at moderate optimization levels (e.g., `-O1`, `-O2`), while certain architectures (e.g., `MIPS32`) and obfuscation techniques hinder optimization performance.

Karacalı et al. explore a compiler-level optimization technique in LLVM [24], proposing the replacement of certain multiplication operations with equivalent addition sequences in the LLVM IR. This aims to improve execution efficiency on architectures where addition is cheaper than multiplication. The authors implemented an IR pass that detects and transforms multiplications into additions, and evaluated it on `x86_64`. Results showed measurable performance gains in specific cases, validating the approach.

2.1.5 Compiler Backend

The backend, which translates the IR into target-specific assembly and machine code, is a major focus of this thesis. This is due to the development of a framework to accelerate backend creation, upon which the new backend for the TriCore architecture is based. For this reason, we examine the backend in greater detail in the following sections.

The requirements imposed on the backend (or code generator) are very restrictive. The target program must be semantically equivalent to the source program and also

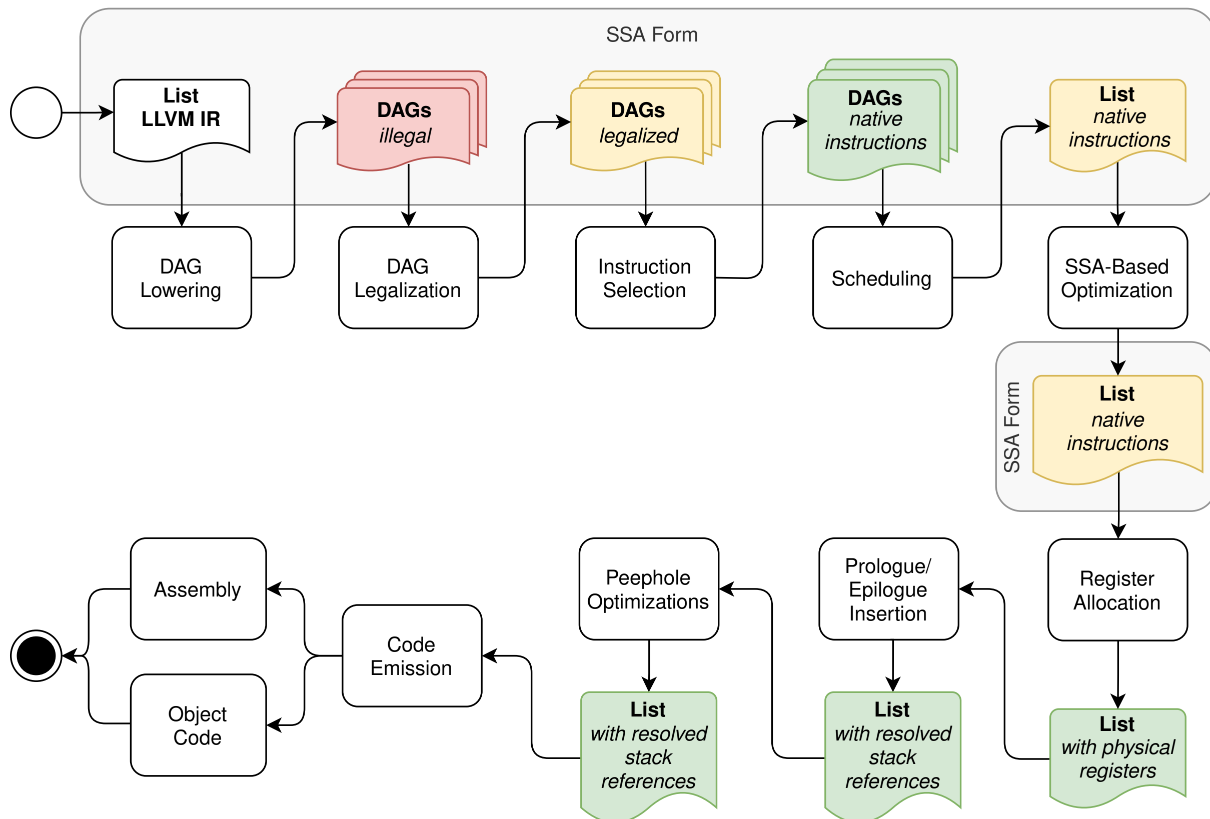

Figure 2.4: LLVM Code Generation Process. This figure was redrawn from [25].

make efficient use of the available resources on the target machine. The challenge is that the problem of generating an optimal target program for a given source program is undecidable [2]. In practice, heuristic techniques are often used to generate good, but not necessarily optimal, code. [2].

In the context of LLVM, code generation involves mapping target-independent virtual instructions from LLVM IR to machine instructions, as well as allocating variables to registers or memory. Transforming LLVM IR into target-specific assembly or machine code requires several steps. Figure 2.4 illustrates how the *LLVM Code Generator* translates target-independent LLVM IR into target-specific assembly or object code [6]. LLVM's Code Generation Process is explained in detail in this chapter.

Understanding how the LLVM Code Generator works is necessary to better understand the structure of an LLVM backend. The LLVM Code Generator translates LLVM IR code into target machine code by executing multiple LLVM passes.

As Figure 2.4 shows, the steps involved in code generation are:

1. DAG Lowering: The process of converting the LLVM IR code into a *Directed Acyclic Graph* (DAG).
2. DAG Legalization: The DAG is transformed such that all data types and operations present in the DAG are compatible with the CPU architecture.
3. Instruction Selection: As one of the most important steps in Code Generation, Instruction Selection replaces all operations in the DAG with machine instructions.
4. Scheduling: Determines the execution order based on the constraints of the CPU architecture.
5. SSA-Based Machine Code Optimizations: Certain optimizations that may take place before allocating registers.
6. Register Allocation: The process of replacing the virtual registers used by LLVM IR with physical registers available for the target. What cannot be saved in a register is usually pushed on the stack.
7. Prologue/Epilogue Insertion: The process of calculating the total space needed for each function's stack frame.
8. Peephole Optimizations: Certain optimizations that may take place, such as removing unnecessary instructions.
9. Code Emission: The final step of the Code Generation process is where assembly code or object code (based on the LLVM IR code provided at the start) is produced.

In the next sections we will explain each step in detail.

2.1.5.1 SelectionDAG

Before we go through the Code Generation Process, we need to briefly explain the data structure that enables Instruction Selection, the `SelectionDAG`. The `SelectionDAG` (Directed Acyclic Graph) is an important data structure used in LLVM's code generation. It offers an abstraction for representing code, which supports Instruction Selection through automated methods, such as dynamic programming-based pattern matching selectors. Additionally, it is well suited for other stages of code generation, like Instruction Scheduling [26, 6]. The `SelectionDAG` consists of nodes, represented by instances of the `SDNode` class, which define operations and their operands. Each `SDNode` has an **Opcode** that indicates the operation it performs, **Operands** that are involved in that operation, and it can produce multiple values, which are represented by the `SDValue` class. Every value also has an associated `Machine Value Type(MVT)` that specifies its data type. SelectionDAGs represent both **data flow** (integer or floating-point values) and **control flow dependencies** (chain edges of type `MVT::Other`), which enforce ordering for operations that produce side effects, such as memory access or function calls. Nodes that have side effects consume and produce a **token chain**, which maintains the execution order. Every SelectionDAG has an **Entry node** (`ISD::EntryToken`) and a **Root node**, which is the final operation with side effects, such as a return statement[26].

2.1.5.2 SelectionDAG Lowering

The first step of the code generator process is the *DAG Lowering* pass, an essential preparatory step for the Instruction Selection, which comes later. In this phase, the input program in the form of LLVM IR instructions, is converted into a Directed Acyclic Graph (DAG). This transformation facilitates the mapping of high-level, target-independent IR to low-level, target-specific machine instructions [26, 11].

2.1.5.3 SelectionDAG Legalization

The `SelectionDAG Legalization` process transforms the DAG ensures that all data types and operations present in the DAG are compatible with the target architecture. This process is done in two phases: *Type legalization* and *Operation legalization*. At the end of the DAG Legalization phase, the DAG should only contain types and operations natively supported by the target [1].

- **Type Legalization**: The purpose of this process is to convert all values in the DAG into types that are compatible with the target architecture. There are two primary methods for handling unsupported scalar types:
  - **Promoting** – This involves converting smaller types into larger ones. For example, a target might require that `i1`, `i8`, or `i16` values be promoted to `i32`.
  - **Expanding** – This involves splitting larger types into smaller ones. For instance, `i64` values can be expanded into two `i32` values.

  These transformations may include adding sign or zero extensions as necessary to ensure the final code behaves the same as the original input [26]. A target implementation informs the legalizer about supported types (and the corresponding register classes) by using the `addRegisterClass` method in its `TargetLowering` constructor.

- **Operation Legalization**: This phase ensures that the DAG contains only operations that the CPU architecture supports. Since different targets may have specific limitations, such as certain operations not being available for every data type (e.g., `X86` lacks byte conditional moves and `PowerPC` cannot sign-extend loads from 16-bit memory [26]), the legalization phase handles this in three main ways:
  - **Expansion** – Replaces unsupported operations with equivalent sequences of supported ones.
  - **Promotion** – Changes a type to a larger one that can support the operation.
  - **Custom legalization** – Uses a target-specific method to handle certain operations.

  Targets define which operations are unsupported and how to handle them (expansion, promotion, or custom) using the `setOperationAction` method in their `TargetLowering` constructor [26].

2.1.5.4 Instruction Selection

Instruction Selection is the stage where LLVM IR is converted into machine-specific instructions. It starts with a legal `SelectionDAG`, which comes from the DAG Legalization phase, and builds a new DAG that represents native instructions. This is done by matching patterns (which are predefined in the `TableGen` files) against the input DAG and generating the corresponding instructions in the output. Much of this matching logic is automatically created by `tblgen` [26].

The **TableGen-based instruction selector** uses `.td` files to define instruction patterns and automatically generates the code required to match these patterns for a given target. This approach offers several advantages, :

The **TableGen-based instruction selector** uses `.td` files to define instruction patterns and automatically generates the code required to match these patterns for a given target. This approach offers several benefits, as outlined in the LLVM documentation [26]:

- Pattern validation – It checks if instructions are valid during compilation.
- Operand constraints – It handles arbitrary constraints on operands, like matching specific bit patterns for immediates.
- Pattern identities – It recognizes standard identities, such as commutative operations (e.g. addition), eliminatig the need for special handling.
- Type inference – It infers types for pattern nodes without explicit specification.
- Pattern fragments – Reusable pieces of patterns (built-in or defined by the target) that get in-lined into instruction patterns during compilation. These fragments simplify pattern definitions by abstracting common operations. For instance, since `SelectionDAG` does not have a native `not` operation, it is represented as `(xor x, -1)`. Targets can create similar shorthand pattern fragments to streamline instruction selection.
- Arbitrary instruction patterns – Targets can use the `Pat` class to define complex patterns that may map to one or more machine instructions, not just single instructions.
- Complex operand handling – Patterns can describe complex operands.
- Even though it is highly automated, the system allows targets to write custom C++ code to match special cases if the situation calls for it.

While the system has many advantages, there are certain limitations, especially regarding `SelectionDAG` nodes that define multiple values (such as `LOAD` or `CALL`). In these cases, custom C++ pattern matching code is necessary. Despite the limitations, the instruction selector is still very important because it handles most binary and logical operations automatically in typical instruction sets [26, 11]. The majority of the Instruction Selection phase happens automatically, based on the instruction patterns defined in the `.td` files, however, in more complex cases, custom C++ code (for instance in `SkeletonISelDAGToDAG.cpp` file) is often necessary [11].

Because the Instruction Selection Phase is one of the most important steps in the Code Generation process, considerable research and development efforts have been dedicated to enhancing its speed and efficiency.
In their 2008 paper “Near-Optimal Instruction Selection on DAGs”, Koes and Goldstein introduce NOLTIS, a linear-time algorithm for the NP-complete problem of instruction selection on DAGs [27]. This is especially valuable in embedded systems, where efficient code generation is essential. NOLTIS produces near-optimal results, reducing code size by an average of 5.1% compared to traditional tree-based methods.

Root et al.'s 2024 work, “Fast Instruction Selection for Fast Digital Signal Processing” [28], presents Pitchfork, a system that improves instruction selection for fixed-point DSP code. DSPs have unique operations like rounding and saturation, which general-purpose compilers often handle poorly. Pitchfork introduces a portable fixed-point IR (FPIR), lifting integer operations into FPIR and lowering them into optimized, target-specific instructions. This integration with LLVM overcomes its DSP limitations, achieving up to 2.44× speedup on Hexagon HVX with comparable or better compile times.

In their 2024 paper “Efficiently Synthesizing Lowest Cost Rewrite Rules for Instruction Selection” [29], Daly et al. present an automated method for generating low-cost rewrite rules using Satisfiability Modulo Theories (SMT). The approach finds functionally equivalent program pairs and synthesizes rewrite rules, introducing two key optimizations: unique rule generation to avoid duplicates and cost minimization to favor efficient rules. Evaluations across multiple ISAs showed up to 768× speed-up in synthesis and a significant reduction in redundant or high-cost rules.

2.1.5.5 SelectionDAG Scheduling

The Scheduling phase takes the target instruction DAG generated in the selection phase and determines the execution order, factoring in machine constraints like minimizing register pressure or improving instruction latencies. After the order is decided, the DAG is **converted into a list** of `MachineInstr`s, and the `SelectionDAG` is discarded. While scheduling is technically a separate phase from Instruction Selection, they are closely connected in practice, as both rely on the `SelectionDAG` [26].

2.1.5.6 SSA-based Machine Code Optimizations

Before allocating physical registers, the code generator may execute several target-specific low-level optimization passes to improve efficiency and performance [25].

2.1.5.7 Register Allocation

The **Register Allocation** problem involves “*mapping a program P_v, which can use an unbounded number of virtual registers, to a program P_p that contains a finite number of physical registers*” [26]. Every target architecture has its own set of physical registers.

If there aren't enough physical registers to store all the virtual registers, some will need to be saved in memory. These are known as "*spilled virtuals*" [26].

LLVM provides several register allocation algorithm. To assess their performance, Xavier et. al conducted an empirical evaluation of four register allocation strategies within the LLVM compiler framework: Fast, Basic, Greedy, and PBQP (Partitioned Boolean Quadratic Programming) [30]. The study assesses each allocator's performance across various benchmarks, focusing on execution time, compilation time, and generated code size. The findings indicate that the Greedy allocator offers a favorable balance between compilation speed and runtime performance, making it suitable for general-purpose applications. In contrast, the PBQP allocator, despite its potential for producing more optimized code, incurs higher compilation times, which may limit its practicality in time-sensitive scenarios.

In their 2024 paper "*Fully Integrated Quantum Method for Classical Register Allocation in LLVM*", Chichereau, Vialle, and Carribault [31] present a hybrid quantum-classical approach to register allocation in LLVM. They model the problem as a Quadratic Unconstrained Binary Optimization (QUBO) task and apply quantum annealing to find near-optimal solutions. Their implementation shows that quantum techniques can be integrated into traditional compiler infrastructures, potentially improving allocation efficiency and code performance. This marks a promising step toward applying quantum computing to compiler optimization.

2.1.5.8 Prologue/Epilogue Insertion

Once the physical registers have been allocated and any required register spills have been processed, the total space needed for each function's stack frame can be calculated [11, 26]. The prologue and epilogue code sequences can now be generated based on this. In this context, prologue and epilogue refer to:

- The prologue is the code executed at the beginning of a function. It typically sets up the stack frame, saves callee-saved registers, and allocates space for local variables and spilled registers.
- The epilogue is the code executed at the end of a function. It restores saved registers, deallocates the stack frame, and performs the final return to the caller [11, 26].

After generating the prologue and epilogue, all abstract references to stack slots (used during earlier compilation stages) can be resolved into actual memory addresses. These addresses are computed relative to the frame pointer or stack pointer, ensuring proper access to local variables and spilled data during execution.

2.1.5.9 Late Machine Code Optimizations

Peephole optimizations are applied just before code emission to eliminate redundant instructions, streamline instruction sequences, and improve runtime efficiency [26].

Common examples include removing unnecessary move instructions, folding constant computations into a single instruction and replacing multiplications by powers of two with shift operations.

2.1.5.10 Code Emission

The Code Emission phase in code generation handles the creation of the final Assembly or Object file code. This is done by lowering the higher-level abstractions, such as `MachineFunction` or `MachineInstr`, to the abstractions used by the `MCLayer`, like `MCInst` or `MCStreamer`. Since the `MClayer` operates at the object file level, it does not work with high-level constructs like functions or global variables. Rather, it "*processes labels, directives, and instructions*" [26] (the fundamental components of assembly and object files). A key class in this process is MCStreamer, which provides an abstraction for emitting assembly directives and machine instructions. It defines methods like `switchSection`, `EmitLabel`, and `EmitSymbolAttribute`, which map directly to assembly-level instructions. Depending on its implementation, `MCStreamer` can output different formats, such as a `.s` assembly file or an ELF `.o` object file, effectively serving as an assembler API [26].

2.1.6 LLVM Backend Implementations

In previous sections, we examined the LLVM Code Generation Process in detail, which outlines the steps a typical LLVM compiler backend must follow to convert LLVM IR code into machine code or assembly specific to the target architecture. In this section we will discuss in more detail a few backend implementations, highlighting important details and challenges faced by the authors.

1. "*Design and Implementation of a TriCore Backend for the LLVM Compiler Framework*" [25] by *Christoph Erhardt*. This thesis was chosen because our implementation of the TriCore Backend was based on this previous work.
2. "*Tutorial: Creating an LLVM Backend for the Cpu0 Architecture*" [32] by *Chen Chung-Shu*. This work was chosen because it represents an extensive (700+ pages) tutorial on writing LLVM backends. It is the result of almost 13 years of work done by several contributors, and is cited by many papers in the literature.
3. "*Adding microMIPS backend to the LLVM Compiler Infrastructure*" [33] by *Jozef Kolek, Zoran Jovanović, Nenad Šljivić and Dragan Narančić*. This work explores the extensibility of LLVM backends. The authors explain the process of extending an existing LLVM backend (MIPS) to add support for an architecture that is part of the same family of architectures.

These research papers are discussed in detail in the following sections.

2.1.6.1 LLVM Backend for the TriCore Architecture

In his Master's thesis, "*Design and Implementation of a TriCore Backend for the LLVM Compiler Framework*" from 2009 [25], Erhardt developed the first open-source LLVM backend for the TriCore Architecture. TriCore is a hybrid architecture designed for embedded systems that integrates features from microcontrollers, RISC processors and DSPs (Digital Signal Processors). The goal of this project was to demonstrate how LLVM's backend infrastructure could be adapted to support a non-mainstream, specialized architecture such as TriCore, which is commonly used in safety-critical environments such as automotive control systems. Erhardt describes in detail the process of developing a full LLVM backend for TriCore based on the LLVM framework, while also discussing any particularities or challenges faced during development. The most important aspects are listed below:

- **Distinction between pointers and integers**: TriCore makes a clear distinction between integer and pointer arguments (as detailed above), meaning that there are separate registers for addresses (`A0-A15` and for data (`D0-D15`. This is problematic because LLVM's code generator implicitly converts all pointers to integers of the same width (`i32` in the case of TriCore) [25] when constructing the `SelectionDAG`. The solution involved "quite invasive modifications" [25] to the LLVM source code and some unconventional workarounds.
- **Custom Lowering of Instructions**: Because of the unique characteristics and limitations of the architecture, Erhardt had to handle several instructions and operands manually: certain addressing modes, shift instructions and comparison handling.
- **Integration of the target with LLVM**: Erhardt underlined that in order to register the new target with LLVM and CLANG, several changes to the source code were necessary. Our work automates this point so that the developer can focus only on the backend related topics.

The TriCore LLVM backend developed within this thesis was compared to the existing TriCore GCC compiler. The results show that Erhardt's backend produces code that is comparable in size with GCC, while achieving 12-20% higher code efficiency.

Our Proof of Concept which is discussed in Chapter 5.2 was based on this work. Initially, the goal of this thesis was to update Erhardt's TriCore backend to the latest LLVM release. However, due to significant changes in LLVM APIs between versions 9 and 17, this was not feasible. Thus, we attempted to build upon this work and rewrite the backend for the latest LLVM release. We reused certain parts such as the Register, Instructions and Calling Conventions definitions, and updated algorithms such as Instruction Selection algorithm, Instruction Lowering or Register Allocation.

2.1.6.2 LLVM Backend for the Cpu0 architecture

Chen Chung-Shu's work titled "*Tutorial: Creating an LLVM Toolchain for the Cpu0 Architecture*" from 2024 [32] represents an extensive (700+ pages long), hands-on tutorial

designed to teach developers how to implement a backend for a custom CPU architecture using the LLVM compiler infrastructure. `Cpu0` is a fictional RISC architecture that was designed for educational purposes. This decision was made so that the LLVM code is as simple and easy to understand as possible for developers that want to learn how to write an LLVM backend.

The tutorial walks through all essential components of a functioning LLVM backend: from setting up the `TargetMachine`, defining instruction formats, registers, and calling conventions, to implementing instruction selection, DAG (Directed Acyclic Graph) lowering, and emitting assembly code. Each section introduces relevant LLVM source files and configuration mechanisms, including `.td` TableGen files and C++ classes spread across subdirectories like `Target`, `AsmPrinter`, and `InstPrinter`. A major strength of this tutorial lies its progressive structure—starting from minimal working code and incrementally adding complexity. This allows readers to compile and test their work at every stage.

One of the key challenges highlighted throughout the tutorial is the significant learning curve of the LLVM framework, warning that the developer "*may easily get stuck debugging*" their backend [32, Chapter 2] when something does not function properly.

The authors also dedicate an entire chapter [32, Chapter 3] to explaining the structure of the backend, noting that approximately "*5000 lines of code*" [32, p. 83] are introduced in the process. Much of this code consists of boilerplate required for setting up any new backend. This thesis aims to streamline and automate the generation of such repetitive code to reduce manual effort and potential errors. On the same page, the authors note that they copied "*almost all the code from MIPS*", following the guidance provided in the official LLVM Documentation for Writing Backends [1]. However, this approach is prone to errors, as each backend has its own unique characteristics, requirements and solutions tailored to the respective architecture. It can be difficult for developers to determine which parts of the copied code need to be modified or removed to correctly support their CPU architecture. Providing a template backend can significantly speed up this process, especially for those just beginning backend development.

Chung-Shu's work reveals both the flexibility and intricacy of LLVM's compiler infrastructure, making clear that while it is powerful and flexible, it demands a thorough understanding of LLVM internals. This tutorial remains a valuable reference for anyone interested in custom hardware support within LLVM, as it is cited in multiple research papers in the literature.

2.1.6.3 Adding microMIPS backend to the LLVM compiler infrastructure

In their work titled "*Adding microMIPS Backend to the LLVM Compiler Infrastructure*" [33], Kolek et. al describe the process and technical considerations involved in extending the LLVM compiler infrastructure with support for `microMIPS`, a compact variant of the `MIPS` instruction set architecture. microMIPS was designed by MIPS Technologies to improve code density, which is particularly beneficial for embedded systems where

memory space is limited. Despite its practical relevance, microMIPS was not natively supported in LLVM at the time, and this project aimed to extend the existing backend to also offer support for it.

The authors first explain the architectural distinctions between traditional MIPS and microMIPS. microMIPS includes a mix of 16- and 32-bit instructions, requiring different encoding and decoding logic compared to regular MIPS. Additionally, it has unique constraints and instruction redefinitions that complicate direct reuse of the existing MIPS LLVM backend. These differences meant that supporting microMIPS required non-trivial modifications across multiple components of the LLVM backend, especially within TableGen instruction descriptions and code emission stages.

A major challenge discussed in the paper was how to integrate microMIPS support into the existing LLVM MIPS backend without duplicating large portions of the code. The team opted to extend the MIPS backend by introducing microMIPS-specific flags, instruction encodings, and conditionals while reusing as much of the existing logic as possible. They had to carefully manage differences in instruction semantics, register usage, and ABI compliance, particularly to ensure interoperability with existing MIPS code and runtime environments.

Another area of focus was the assembler and disassembler, as microMIPS requires custom handling of compressed instructions and alternate instruction formats. The team extended LLVM's `AsmPrinter` and disassembler modules accordingly, making use of LLVM's TableGen system to avoid redundancy while maintaining correctness across multiple instruction encodings.

The result was a working microMIPS backend integrated into LLVM, tested with standard compiler benchmarks and capable of producing compact, valid microMIPS binaries. The project demonstrates how LLVM can be extended to support novel or specialized ISAs, but also underlines the complexity of maintaining architectural consistency when extending an existing backend.

Overall, this work is a valuable case study that shows existing backends can be extended to support other (related) targets.

2.2 Tooling and Templates

The main contribution of this thesis is the proposal of a new approach for developing LLVM backends. This includes a template LLVM backend that incorporates all essential boilerplate code required for an LLVM backend, which is used as a base for creating a new custom backend. Instead of writing all the code from scratch or copying code from other projects, this new approach uses a template and code generation tools to speed up the development process. Additionally, it is supported by a suite of helper scripts designed to streamline the development workflow. In this section, we will explore other development tools and code generator solutions that influenced our approach.

2.2.1 Code Generators

Code generation is a well-established area in software development, with numerous projects and tools available for automating the creation of code. These tools often help developers by generating boilerplate code, enabling faster development cycles and reduce human errors. Several popular code generator frameworks have been developed over the years, each with their unique features and design choices. In the next few sections we briefly discuss some of the most popular code generation tools, followed by a section where we explain why we chose to implement a custom solution. Nevertheless, in the development of our solution, we took inspiration from these existing tools.

2.2.2 T4 Code Generation

T4 (Text Template Transformation Toolkit) is a code generation tool integrated into the Microsoft Visual Studio IDE (Integrated Development Environment). T4[1] allows developers to define templates that generate code based on specific input, such as data models or configurations. It is widely used in .NET development, where it helps generate data access code automatically. One of the key benefits of T4 is its flexibility, as it allows developers to automate the creation of text files, including source code, as well as complex data structures. Because T4 is so well-established as it's deeply integrated with the .NET programming language and also with Microsoft Visual Studio IDE, it was the subject of a substantial amount of research in the literature.

Naimi et al., in their study "A DSL-based Approach for Code Generation and Navigation Process Management in a Single Page Application" [34], used T4 to automate SPA code generation from a Domain-Specific Language (DSL). Their DSL defines the SPA's structure and navigation, which is then transformed via T4 templates into platform-specific architecture and base code, including components, routing modules, and directory structure.

In their 2017 study "Comparison of the expressiveness and performance of template-based code generation tools" [35], Luhunu and Syriani evaluated nine tools, including Microsoft's T4. They found T4 expressive in handling metamodel patterns like navigation and recursion via C# functions, but lacking native polymorphism support, often requiring code duplication. Performance-wise, T4 was competitive for small models (under 1,000 objects) but slowed down with larger models due to XML navigation overhead.

[1] T4 Official Documentation: https://learn.microsoft.com/en-us/visualstudio/modeling/code-generation-and-t4-text-templates?view=vs-2022

2.2.3 Entity Framework

Entity Framework (EF)[2] is a library developed by Microsoft for the .NET platform, that enables developers to access databases. It is designed as an *Object-Relational Mapper* (ORM). ORMs work by mapping between two worlds: the relational database and the object-oriented software world of classes and code. EFs main advantage is that it allows developers to interact with databases using .NET objects, reducing the need for manual SQL queries [36]. It abstracts the underlying database schema through the Entity Data Model (EDM), which represents entities and their relationships in a conceptual model. EF Core, the cross-platform and open-source evolution of EF [36], introduces enhancements such as improved performance, support for asynchronous operations, and compatibility with multiple database providers like SQL Server, PostgreSQL, and MySQL.
It also offers features like change tracking, lazy loading, and migrations for schema evolution, making it suitable for modern application development [36].

In their 2025 study "Harnessing the Power of Entity Framework Core for Scalable Database Solutions", Jayaraman and Siddharth examine EF Core's features, performance strategies, and best practices for building scalable database systems. Their experiments show that EF Core handles diverse database operations efficiently, making it well-suited for applications requiring fast queries, high throughput, and low latency [37].

2.2.4 Motivation for a Custom Code Generator solution

In this thesis, we developed an LLVM backend template that serves as the foundation for an automated code generation system. The system takes the template and together with a specified LLVM release, generates the **LLVM Development Environment**. In this environment, the template is fully integrated into the LLVM source tree and build system. The system is implemented using **Python Invoke tasks**[3], which provide a structured way to execute and organize the various steps involved in generating a backend.

We chose not to rely on existing code generation tools such as T4 or Entity Framework because they are primarily designed for generating high-level application or data-access code withing managed environments like .NET. In contrast, our use case involves manipulating a large and complex C++ codebase like LLVM, where adaptions are done based on context-aware logic or pattern matching. We chose to implement the system using Python for several reasons. First, Python offers excellent readability and maintainability, which lowers the barrier to contribution, while also keeping the code friendly for new developers. Second, Python's rich standard library and ecosystem (including packages for file manipulation, templating, and parsing) make it well-suited for tasks involving code generation and file transformations.

[2] EntityFramework Official Documentation: https://learn.microsoft.com/en-us/ef/core/
[3] Python Invoke Documentation: https://docs.pyinvoke.org/en/stable/concepts/invoking-tasks.html

Additionally, Python's platform independence ensures that the tool can be run on most development environments with minimal setup, requiring only the Python interpreter and a small number of external packages. This aligns with our goal of reducing the initial work required for new backend developers. We opted to implement the scripts as Python Invoke tasks because they can be easily executed from anywhere within the development environment, making them highly flexible. The tasks also support short-form commands, enabling users to quickly perform common actions with minimal typing. This design choice streamlines workflows and enhances productivity by providing developers with a simple and efficient way to interact with the system throughout the development process.

2.3 TriCore Architecture

In 1997 Siemens Semiconductors officially introduced the TriCore microprocessor architecture. Despite its name, TriCore was not designed as a multi-core architecture. Rather, it combines three different domains by integrating three processors on a single chip [25]:

- a real-time microcontroller unit for fast context switching and low latencies,
- a high-performance digital signal processor (DSP),
- a superscalar RISC processor.

2.3.1 Instruction Set Architecture

The TriCore Instruction Set Architecture (ISA) features "*a uniform 32-bit address space with optional virtual addressing and memory-mapped I/O*" [38], supporting various implementations from scalar to superscalar and enabling multiprocessing compatibility. This flexibility allows for cost-performance trade-offs at different implementation levels [38].

The architecture includes "*both 16-bit and 32-bit instruction formats*" [38], with all instructions available in 32-bit form. Frequently used instructions are also available in a 16-bit subset, minimizing code size, memory usage, and power consumption.

Real-time performance is optimized through minimizing interrupt delay and enabling fast context switching. The architecture minimizes latency by avoiding long multi-cycle instructions and incorporating a hardware-supported interrupt scheme [39].

2.3.2 Register set

The TriCore register set consists of 32 GPRs (General Purpose Registers), two 32-bit registers with program status information (PCXI - Previous Context Information and PSW - Program Status Word) and a PC (Program Counter). "*The 32 general-purpose registers are divided into sixteen 32-bit data registers (D0 to D15) and sixteen 32-bit*

address registers (A0 to A15)" [39]. Four of these GPRs have special roles: D15 acts as an implicit data register, A10 is used as the stack pointer (SP), A11 serves as the return address register, and A15 functions as the implicit base address register. For greater flexibility regarding the data types, any two consecutive data registers, starting with the even index, can be concatenated to form eight extended registers (E0 to E14), to support 64-bit values [39].

2.3.3 Research on TriCore Microcontrollers

The TriCore Architecture, developed by Infineon, has gained significant attention in the embedded systems research community due to its high performance, low power consumption and applicability in a wide range of fields, including automotive, industrial or communication systems. Several studies have explored the use of the TriCore MCU, and in this section we will highlight some of them.

Oka et al. [40] conducted in 2014 a security analysis of the widely used Infineon TriCore processor architecture (specifically the TC1797 microcontroller) within the context of electronic control units (ECUs) in vehicles. Their study, carried out in a simulated environment under controlled conditions, demonstrated that it was possible to manipulate the program flow and execute arbitrary code.

Binder et al. [41] studied how amplification timing anomalies affect the TriCore superscalar architecture, which can compromise worst-case execution time (WCET) analysis in real-time systems. They extended a canonical pipeline model to capture dual-pipeline features like asynchronous store buffers and structural hazards, using model checking to detect anomalies and evaluate computational complexity.

CHAPTER 3

Architecture and Design

> *The brain weighs only three pounds, yet it is the most complex object in the solar system.*
>
> Michio Kaku

The design of any software tool plays a crucial role in determining its usability, efficiency, and maintainability. Initially, this project aimed to create an LLVM backend for the TriCore TC1.6 architecture, which was discussed in Chapter 2.3. However, during development, it became evident that the initial setup phase of the LLVM environment presents a significant challenge. The complexity of boilerplate code, the need for deep knowledge of LLVM's architecture, and the lack of structured guidance create a steep learning curve for new developers.

To address this topic, the project evolved into designing and implementing a framework that streamlines the creation of the *LLVM Development Environment* by automating boilerplate generation and integrating the new target seamlessly with LLVM, CLANG (the C, C++, and Objective-C frontend), and LLD (the LLVM linker). This chapter details the design decisions we made to achieve this goal, outlining the architecture and key components.

3.1 LLVM Backend Development Kit

To streamline the development of a new LLVM backend, the **LLVM Backend Development Kit** (LBDK) is introduced in this thesis. It consists of a template LLVM backend, called **Skeleton**, and a suite of convenient scripts that have the following purposes: *setting up* the **LLVM Development Environment** (LDE) and *adding* a new target to an existing LDE. The LLVM Development Environment is produced by the LBDK, as shown in Figure 3.1. It includes the necessary scripts, configurations, and dependencies required to develop, build, and test a custom LLVM backend.

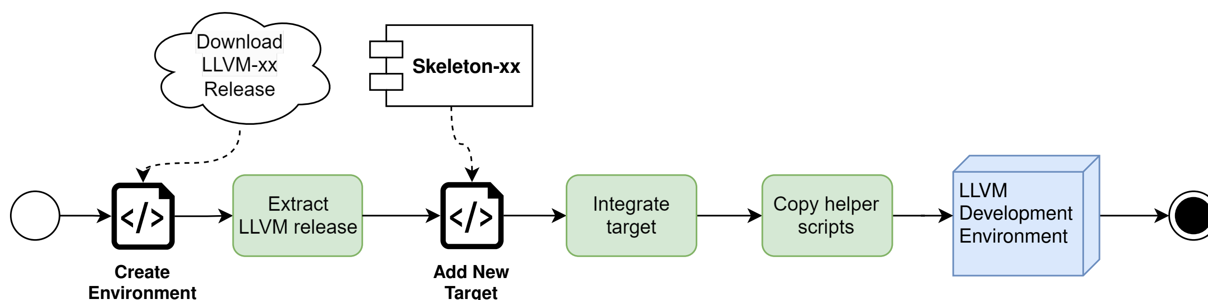

Figure 3.1: Overview of the generation process for the LLVM Development Environment

3.1.1 Structure of the LLVM Backend Development Kit

As Figure 3.1 shows, the LLVM Backend Development Kit consists of the following components:

- **LLVM Backend Template — Skeleton**: The template LLVM backend that serves as the base for developing custom backends. It is a bare-bones LLVM backend, containing only the essential components required for a successful build.
- **Convenient Scripts** – A set of scripts that facilitates the development process.
    1. **Create environment**: This script will create the *LLVM Development Environment*.
    2. **Add new target**: This script can be used to add a new custom target into an existing *LLVM Development Environment*.

3.1.2 How to create the LLVM Development Environment

As shown in Figure 3.1, the process of creating the LDE starts by executing the *Create New Environment* script (`invoke setup_new_environment`). The process involves multiple steps, which are performed by the two scripts, part of the LBDK. The steps to create the LDE are explained below:

1. **Download the LLVM release** – The first step involves downloading the specified release from the official GitHub repository[1] of the LLVM project. The release package includes all the necessary tools and libraries to develop a new compiler for the target, from the frontend to the backend itself.
2. **Extract the LLVM sources** – Once the LLVM release package is downloaded, the next step is to extract the LLVM source code to a specified directory. This directory will be the root directory of the environment.
3. **Copy the Skeleton backend template** – The process of adding a target to the environment begins now, and this is done by calling the *Add new target* script (`invoke add_new_target`). The Skeleton backend template folder is copied to

[1] https://github.com/llvm/llvm-project

the correct location within the LLVM source tree. This folder is then renamed to the desired target name (along with all occurrences of the word “Skeleton” in every file).

4. **Integrate the backend** – After placing the new target's folder in the correct location, the new backend is integrated into the LLVM, Clang, and LLD build systems, also done by the *Add new target* script. This step requires modifying LLVM source files and generating new target-specific files, ensuring they are placed in the correct locations. This is the most important phase of the environment creation process, as it sets up the environment for the development of the new custom backend.
5. **Copy the convenience scripts** – Finally, after the target is integrated, the *Create new environment* script (`invoke setup_new_environment`) copies the helper scripts (*Build* and *Compile Sample Code*) into the new environment. These scripts are meant to be used during the whole development process of the backend. They make it easier for developers to build and test their implementation.

Once the creation process is finished, the LDE can be found at the location specified by the user.

3.1.3 LLVM Backend Template — Skeleton

As outlined in Section 3.1.1, the LLVM Backend Development Kit includes an LLVM Backend Template called Skeleton, which serves as the core component of this thesis. Its primary goal is to provide a stable and user-friendly foundation for developing custom LLVM backends.

The LLVM Code Generator defines several abstract base classes with virtual functions, requiring backend developers to implement specific subclasses. While most of these classes do not directly contribute to code generation, they provide important details about the target machine's characteristics and properties. This approach ensures that the core algorithms and processes remain largely target-independent, as they access all necessary target-specific details through well-defined interfaces.

A simplified version of this class hierarchy of the Skeleton is presented in Figure 3.2. In the diagram, pre-existing framework classes are highlighted in red, manually implemented backend classes in white, and classes automatically generated by the TableGen tool have a dotted outline.

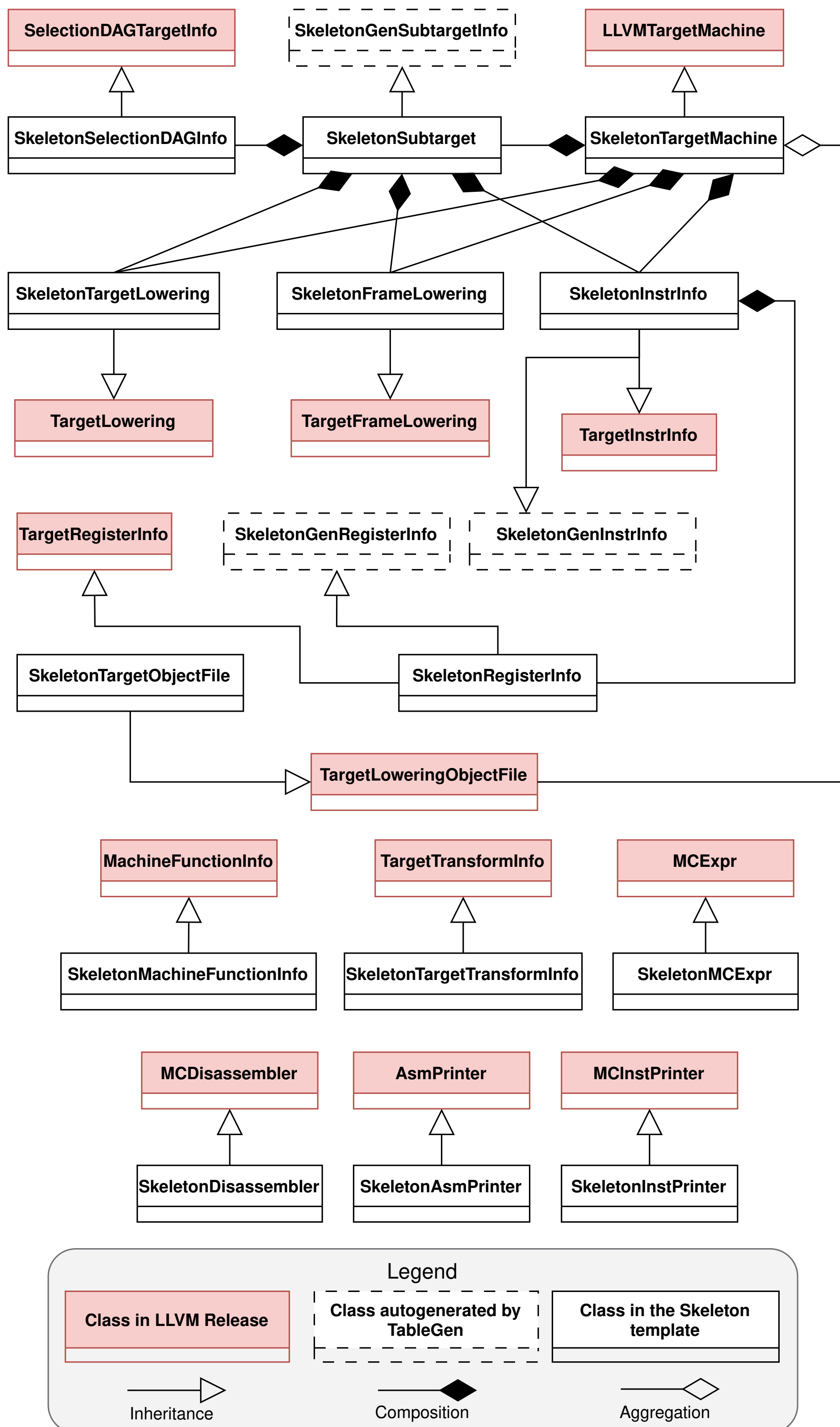

Figure 3.2: Skeleton class hierarchy. Classes in solid white are the classes from the Skeleton itself, classes in red are from the LLVM project, and the ones with the dotted outline are the classes auto-generated by `tblgen`

Classes that are part of the Skeleton backend are discussed below:

- **SkeletonTargetMachine** – `SkeletonTargetMachine` represents the target machine for the Skeleton backend. It manages various components such as subtarget information, instruction info, frame lowering, and target lowering. This class provides target-specific information for code generation and optimization. It interacts with `SkeletonSubtarget`, `SkeletonInstrInfo`, `SkeletonFrameLowering` and `SkeletonTargetLowering` to gather and manage target-specific details. Additionally, it aggregates `TargetLoweringObjectFile` to handle object file-specific lowering.
- **SkeletonSubtarget** – The `SkeletonSubtarget` class provides subtarget-specific information for the Skeleton target. It manages instruction info, frame lowering, register info, target lowering, and selection DAG info. This class interacts with `SkeletonInstrInfo`, `SkeletonFrameLowering`, `SkeletonTargetLowering`, and `SkeletonSelectionDAGInfo` to provide detailed subtarget configurations.
- **SkeletonFrameLowering** – The `SkeletonFrameLowering` class handles the lowering of the stack frame for the Skeleton target. It manages stack frame layout and callee-saved registers. This class interacts with `SkeletonSubtarget` to apply frame-lowering strategies specific to the Skeleton target.
- **SkeletonTargetLowering** – The `SkeletonTargetLowering` class handles the lowering of LLVM IR to machine instructions for the Skeleton target. It provides custom lowering hooks and target-specific operations. This class interacts with `SkeletonSubtarget` to implement target-specific lowering techniques.
- **SkeletonMachineFunctionInfo** – The `SkeletonMachineFunctionInfo` class contains private Skeleton target-specific information for each machine function. It manages the return register, the global base register, and the frame index. This class interacts with `SkeletonTargetMachine` to store and retrieve function-specific information.
- **SkeletonTargetObjectFile** – The `SkeletonTargetObjectFile` class handles target-specific object file information for the Skeleton target. It manages section and symbol handling. This class interacts with `SkeletonTargetMachine` to provide object file-specific lowering.
- **SkeletonAsmPrinter** – The `SkeletonAsmPrinter` class converts the internal representation of machine-dependent LLVM code to the Skeleton assembly language. It manages the printing of operands and instructions for inline assembly expressions. This class interacts with `SkeletonMCInstLower` to lower machine instructions to assembly instructions. It also interacts with `SkeletonInstrInfo` and `SkeletonSubtarget` to gather necessary information for printing.
- **SkeletonDisassembler** – The `SkeletonDisassembler` class disassembles machine code for the Skeleton target. It provides methods for decoding instructions. This class interacts with `SkeletonInstrInfo` and `SkeletonSubtarget` to decode and interpret machine instructions.

- **SkeletonMCInstLower** – The `SkeletonMCInstLower` class lowers a machine instruction into an `MCInst` for the Skeleton target. It manages the lowering of operands. This class interacts with `SkeletonAsmPrinter` to convert machine instructions to assembly instructions.
- **SkeletonMCExpr** – The `SkeletonMCExpr` class represents target-specific expressions for the Skeleton target in the MC layer. It interacts with `SkeletonMCInstLower` and `SkeletonAsmPrinter` to handle expressions during instruction lowering and printing.
- **SkeletonInstrInfo** – The `SkeletonInstrInfo` class provides instruction information for the Skeleton target. This class interacts with `SkeletonSubtarget` and `SkeletonRegisterInfo` to supply detailed instruction and register information.
- **SkeletonRegisterInfo** – The `SkeletonRegisterInfo` class provides register information for the Skeleton target. It manages register allocation and usage. This class interacts with `SkeletonInstrInfo` to supply register details for instruction handling.
- **SkeletonSelectionDAGInfo** – The `SkeletonSelectionDAGInfo` class provides selection DAG information for the Skeleton target. It manages target-specific DAG operations. This class interacts with `SkeletonSubtarget` to implement DAG operations specific to the Skeleton target.
- **SkeletonInstPrinter** – The `SkeletonInstPrinter` class prints machine instructions for the Skeleton target. It manages the printing of operands and mnemonics. This class interacts with `SkeletonAsmPrinter` to provide detailed printing of instructions and operands.

3.2 LLVM Development Environment

This environment provides a structured workspace for developers to implement and test custom LLVM backends efficiently. It includes the LLVM release package, which contains a backend template, along with a suite of helper scripts designed to automate common development tasks such as building and testing. The structure of the LDE is shown in Figure 3.3.

3.2.1 LLVM Backend Template — Skeleton

At its core, the LDE simplifies the process of extending LLVM by providing a pre-integrated backend template, called *Skeleton*, which serves as a foundation for implementing a custom backend for a target architecture.

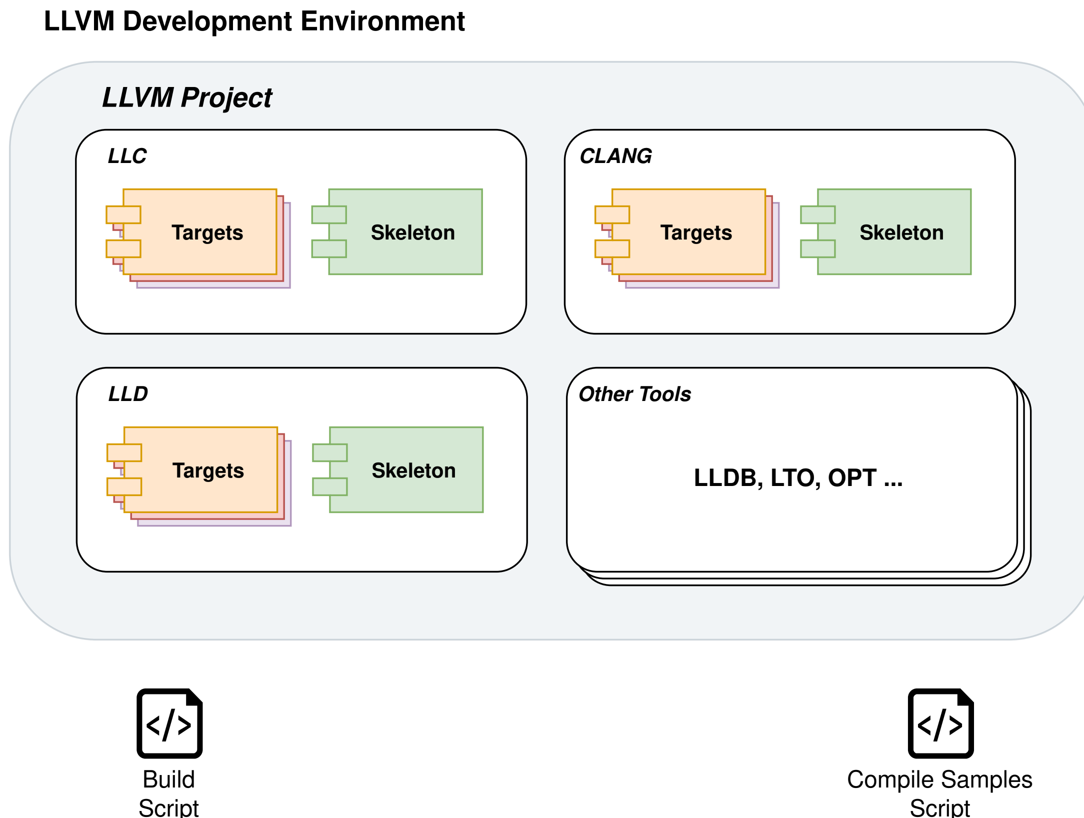

Figure 3.3: LLVM Development Environment

This template is already integrated with key LLVM components, including:

- **CLANG** – The C/C++ frontend, responsible for translating high-level source code into LLVM Intermediate Representation (IR). Integrating the backend with CLANG ensures that developers can compile C-based source code directly for their custom target, making it easier to test the entire compilation pipeline from source to machine code.
- **LLC** – The LLVM static compiler, which lowers LLVM IR to target-specific assembly code or object file. LLC is using the custom backend implementation to execute the code generation process. It produces target-specific assembly code or object files, depending on the configuration.
- **LLD** – The LLVM linker, responsible for linking compiled object files into executable binaries. LLD is included because linking is an essential step in generating fully functional programs, and ensuring compatibility with LLD allows the backend to produce complete executables without relying on external linkers.

3.2.2 Convenience Scripts

In addition to the backend template already integrated, the LDE includes two convenience scripts: *Build environment* and *Compile sample code*.

3.2.2.1 Build environment

The **Build environment** script is a very important component of the LDE (created by the *Create New Environment* script). It handles the setup and initiation of the build process for the new compiler. This process ensures that the custom target compiler is built and ready for use. The task follows a sequence of steps to complete the build:

1. Create the build directory: Depending on the selected build type, the script creates an appropriate build directory.
2. Configuration: The script prepares the necessary configuration based on the build type, target selection, and system settings.
3. Build process: The build process is then executed, ensuring the compilation and linking steps are carried out properly.

3.2.2.2 Compile sample code

To be able to test that the compiler works, several sample C code files are provided within the LDE. They are collected from the *Learn C* Github Repository[2]. This collection of code files was selected because it covers a wide range of C scenarios, addressing most of the key specifications of the C language. The *Compile sample code* script uses the custom target's compiler to compile each C code sample individually, saving the log output for each file. A test is considered passed if assembly code is generated, although the correctness of the generated assembly is not verified, leaving this to the developer.

3.2.3 Conclusion

By bundling these components into a ready-to-use package, the LDE significantly reduces initial setup time and complexity, allowing developers to focus on implementing and refining their target-specific backend features. Furthermore, the LDE helps the backend developer by supporting the testing of their backend implementation, facilitating the use of the *Test-Driven Development* methodology.

[2] https://github.com/abhayanigam/Learn_C

CHAPTER 4

Implementation

If you can't explain it simply, you don't understand it well enough.

Albert Einstein

The Implementation chapter showcases the technical details behind the core solutions presented in this thesis:

- LLVM Backend Development Kit;
- LLVM Development Environment, which is generated by the kit.

Finally, we provide a high-level overview of the process of developing an LLVM backend using our solution, highlighting the steps that we streamlined.

4.1 LLVM Backend Development Kit

As discussed in previous chapters, the LLVM Backend Development Kit (LBDK) is a tool designed to simplify the process of creating and testing custom LLVM backends. The LBDK includes a Backend Template called **Skeleton** and a suite of convenience scripts which help in generating the LLVM Development Environment (LDE):

- Create new environment
- Add target to environment

The scripts and their functionality were discussed in detail in Chapter 3.2. The goal of the LBDK is to automatically set up a new LLVM Development Environment, which contains a full release of the *LLVM Project*[1], and a new target (a copy of the Skeleton template) that is already integrated with LLVM, Clang, and LLD. This setup eliminates the need for developers to write boilerplate code, enabling them to focus on implementing the target-specific components of the backend.

[1] https://llvm.org

4.1.1 LLVM Backend Template - Skeleton

The Skeleton template backend represents the core component of the LBDK because it streamlines the initial setup of the backend development process. Its implementation was started by following the advice from the LLVM documentation on writing an LLVM backend, where we are advised to "copy existing examples" from existing backends, and "change the file names for your target" [1]. Instead of adapting the implementation for a new target, we gradually removed all target-specific code and definitions until only a minimal backend remained. All classes and methods that still remain contain only stub implementations, intended to be extended later for any custom target. In chapter 3.1.3 we discussed the class structure of the Skeleton backend and explained in detail the responsibilities of each class and the relationships between them. The Skeleton was based on the existing `Lanai` backend, chosen for its relatively small codebase compared to other backends like `RISC-V`, `ARM`, or `X86`. Other viable alternatives included `MSP430` and `MIPS`; however, this choice did not significantly impact the final result, as all LLVM backends share the same overall structure.

4.1.1.1 Multiple LLVM versions

In practice, the kit includes multiple backend versions, each corresponding to a supported LLVM version. This is done to increase flexibility for the users of the framework. We need to include multiple versions of the template because LLVM doesn't guarantee API stability between major releases, as the LLVM Developer Policy [19] (C API Changes section) states. At the time of writing, the LBDK supports the following LLVM Major Releases: *LLVM 17*, *LLVM 18*, *LLVM 19*, *LLVM 20*. This means that, for every new release of LLVM, a corresponding version of the Skeleton has to be created to support to new version. We only need to support the major release versions of LLVM. The reason is that the LLVM Developer Policy[19], in the C API Changes section states that they guarantee Release Stability: *"We won't break the C API on the release branch with patches that go on that branch, with the exception that we will fix an unintentional C API break that will keep the release consistent with both the previous and next release."*. In other words, the release branch should be relatively stable in general, meaning that we should not need a new version of the Skeleton for patch releases.

4.1.1.2 Update Skeleton to new LLVM version

Maintainability is not a concern for the Skeleton due to its small size and reduced dependency on LLVM's APIs. However, in some cases, adaptations may be required to support newer LLVM versions. Any necessary modifications between LLVM versions are documented in the project's `CHANGELOG.md` file, found at the root of our repository.

To add support for a new LLVM release we need to follow the steps below:

1. **Add the new LLVM version string**
   In the `lib/variables.py` file, the `SupportedLLVMVersions` enum needs to be updated as shown in the following listing:

```
class SupportedLLVMVersions(Enum):
    # We are supporting the latest major releases of LLVM
    LLVM_17 = "17.0.6"
    LLVM_18 = "18.1.8"
    LLVM_19 = "19.1.7"
    LLVM_20 = "20.1.0"
    LLVM_XX = "XX.xx.x"

```

2. **Create a new Skeleton template for the new LLVM Release**
   Copy the most recent Skeleton template, `Skeleton_20` in this example, and rename it to `Skeleton_XX`, where `XX` represents the LLVM major version number. This is important because the script selects the appropriate template based on the LLVM version specified as a parameter.
3. **Create a new environment**
   This is done by running the commands shown below:

```
$ inv se -t Skeleton -l XX -p {PATH}
```

4. **Build the environment**
   We first need to change to the environment's directory and then run the following command:

```
$ inv build
```

5. **Fix build errors and copy the changes**
   The goal is to achieve a successful build, which requires fixing any build errors that may arise. After resolving an error, it is essential to apply the fix to the folder created in Step 2. Once all build errors have been addressed, support for the new LLVM version is successfully added.

4.1.2 Convenience scripts

In Chapter 3, we discussed the design and architecture of the LBDK and saw that, in addition to the Skeleton, it also includes two convenience scripts: *Create new environment* and *Add new target* to an existing environment. For reasons discussed in this chapter, all scripts provided in this thesis are implemented as **Python Invoke** tasks.

4

4.1.2.1 Python Invoke

Before delving into the particularities of the scripts, we should first explain the decision to implement them as *Python Invoke Tasks*. According to the documentation, "***Invoke** is a Python library designed for managing shell-oriented subprocesses and organizing executable Python code into CLI-invokable tasks*" [2]. The primary goal of **invoke** is to provide a powerful and clean feature set for task execution and automation. Developers can define tasks in a `tasks.py` file, run shell commands and handle common patterns with minimal boilerplate. The main benefits of using invoke are:

- **Flexibility**: As part of the system's design, Python Invoke was chosen because it offers a simple and clean approach to defining and executing Python scripts from any subdirectory within the directory that contains `tasks.py`. This allows developers to efficiently manage the build process and test their backend implementation without ensuring the scripts are discoverable in the system path. For example, consider the *Print Message* Invoke task, defined in Listing 4.1. This task prints a message to the system console.

```python
# Contents of tasks.py

@task(aliases=['p'],
help={"message": "The message to print to the console"})
def print_message(ctx, message="Hello World! I am an Invoke Task!")
    :
"""
Prints a message to the system console
"""
    print (message)

```

Listing 4.1: Python Build Invoke Task

We now have the possibility to invoke the task from anywhere within the directory that contains `tasks.py`, by calling the command shown in Listing 4.2. As Listing 4.2 illustrates, the invoke command has a short form for the name of the task as well as for the arguments. This makes the command easy to remember and very convenient to use regularly.

```
$ invoke print_message --message="Hello World!"
# or simply
$ inv p -m "Hello World!"

```

Listing 4.2: Example of a command to invoke an Invoke task. This Listing also shows that invoke also accepts a short form for the command.

- **Useful help messages**: Another reason for choosing Python Invoke is its ability to clearly explain the arguments of a script and provide an understanding of what the script does. This feature is particularly valuable for users, as it improves the

[2] https://www.pyinvoke.org/

transparency and usability of the scripts, making it easier for them to understand how to interact with and utilize the system effectively. Considering the task defined in Listing 4.1, Listing 4.3 shows how Invoke generates a help message that explains the required arguments and their respective meanings.

```
$ invoke --help print_message
Usage: inv[oke] [--core-opts] p [--options] [other tasks here ...]

Docstring:
    Prints a message to the system console

Options:
    -m, --message     The message to print to the console


```

Listing 4.3: Example of help message for an invoke task

In conclusion, using Python invoke tasks allowed us to define complex operations that can be called with easy-to-remember commands such as the one shown in Listing 4.2.

4.1.2.2 Create new environment

The **Create new environment** Invoke task is implemented in the `tasks.py` file that can be found at the root of the LBDK repository.

As mentioned in Section 4.1.2.1, all scripts provided in this thesis are available as Invoke tasks. To learn what the script does and what arguments it expects, we can run the `help` command for the `setup_new_environment`. The `setup_new_environment` script expects the following arguments:

- `arch` – This argument represents the bit width of the CPU architecture, and it can have one of the following values: *8, 16, 32, 64*.
- `endianness` – This argument specifies of the CPU architecture is *little* or *big* endian.
- `llvm-version` – This argument specifies which LLVM major version the compiler will be based on, and it can take one of the following values: *17, 18, 19, 20*, representing the supported Major Releases of LLVM, also listed in 3.1.3.
- `path` – The location on disk where the LDE will be placed.
- `target-name` – The name of the CPU architecture.

The arguments `arch` and `endianness` are crucial for integrating the new backend into `LLVM`, `CLANG`, and `LLD` and are passed to the *Add new target* script which does the integration. Listing 4.4 provides an example of how to write the command to invoke the task.

```
$ invoke setup_new_environment --path=~/Temp --target_name=MyTarget --
    arch=32 --endianness=little --llvm_version=19
or the shorter version:
$ inv se -p ~/temp -t MyTarget -a 32 -e little -l 19
```

Listing 4.4: Invoke command example for the `setup_new_environment` task

As mentioned in Chapter 3.1.2, this task also copies the helper scripts needed to build and test the compiler. Additionally, it creates two configuration files that assist in the build process:

- `official_targets_to_build.ini`
- `experimental_targets_to_build.ini`

These files contain lists of targets that should be included in the build without requiring manual specification when running the build script. Each target name must be written on a separate line in the appropriate file, depending on whether the target has an official LLVM backend. The target for which the environment is created is automatically added to `experimental_targets_to_build.ini`. This approach allows users to define the targets to be built just once, simplifying the build command and reducing the need for repeated manual input. An example of configuring the build for `X86` and `RISCV` by adding them to the `official_targets_to_build.ini` file is shown in the Listing 4.5.

```
# Place here the official LLVM targets you want to build
RISCV
X86
```

Listing 4.5: Example for configuring the build for `X86` and `RISCV` by adding them to the `official_targets_to_build.ini` file

4.1.2.3 Add new target

The `add_target_to_environment` task integrates the new custom target into the *LLVM*, *CLANG*, and *LLD* build systems. This process requires several modifications to the LLVM source files and various *CMake* files. Some of these changes depend on the target's `bit-width` and `endianness`. To learn what the script does and what arguments it expects, we can run the `help` command for the `add_new_target`. The `add_new_target` script expects the following arguments:

- `target_name` – The name of the CPU architecture.
- `path` – The location on disk where the LDE will be placed.
- `arch` – This argument represents the bit width of the CPU architecture, and it can have one of the following values: *8, 16, 32, 64*.
- `endianness` – This argument specifies of the CPU architecture is *little* or *big* endian.
- `llvm_version` – This argument specifies which LLVM major version the compiler will be based on, and it can take one of the following values: *17, 18, 19, 20*, representing the supported Major Releases of LLVM, also listed in 3.1.3.

The arguments `arch` and `endianness` are crucial for integrating the new backend into `LLVM`, `CLANG`, and `LLD`. In some cases, this information determines specific changes that need to be made; the changes are explained in detail in Chapter 4.2.1.

Listing 4.6 provides an example of how to write the command to invoke the task.

```
$ invoke add_new_target --target_name=MyTarget --path=~/Temp --arch=32
    --endianness=little --llvm_version=19
or the shorter version:
$ inv at -t MyTarget -p ~/temp -a 32 -e little -l 19
```

Listing 4.6: Invoke command example for the `add_new_target` task

The modifications that need to be made are specified in *YAML* files and can be found inside the `lib/register_target/configs/llvm_XX`[3] folder within the LBDK. We identified two types of modifications:

- **String** – This represents the majority of the modifications necessary. They are applied by searching for *pattern*s in the file and adding the *content* before or after the *pattern*. Below is an example of a *String* modification applied in one of the `CMakeLists.txt` files found in CLANG:

```
modifications:
    - file_path: clang/lib/Basic/CMakeLists.txt
      changes:
      - type: string
        content: >2
            Targets/<Target>.cpp
        pattern: add_clang_library
        relative_location: after

```

- **New file** – Certain files specific to the target have to be generated and placed in the right location. The *content* of these files is defined in the *YAML* file and may have to be edited later, depending on the specific features of the target. Below is an example of a *New file* modification, where we generate the `ELF Relocations`[4] specific to the target:

```
modifications:
    - file_path: llvm/include/llvm/BinaryFormat/ELFRelocs/<Target>.
    def
      changes:
      - type: new_file
        content: >2
            #ifndef ELF_RELOC
            #error "ELF_RELOC must be defined"
            #endif
            ELF_RELOC(R_<TARGET>_NONE,        0)

```

[3] `XX` is the LLVM Major Release version the environment is based on, and the version of the Skeleton we used. For more details please refer to Chapter 4.1.1

[4] ELF relocations are metadata entries in an ELF file that adjust addresses in the binary during linking or loading to ensure symbols and references point to their correct locations in memory.

Note that the YAML format that we defined allows for multiple changes for any file. The *Add new target* script reads all the YAML configuration files and processes each modification sequentially. One important aspect is that if a pattern cannot be found in the specified file, an error is raised and the execution stops immediately.

4.2 LLVM Development Environment

4.2.1 Integrating of the Skeleton

In this section we will explain the modifications that are needed to integrate a target into `LLVM`, `CLANG` and `LLD`. As mentioned previously, all the modifications listed in this section are performed by the *Add new target* script, which is part of the LDE. The changes necessary to integrate the target in LLVM can be seen in Table 4.1, for CLANG in Table 4.2 and for LLD in Table 4.3. As the tables show, we have to apply several modifications to multiple source files from the LLVM project. The majority of modifications seen in the tables have to be applied for every target, regardless of `bit-width` or `endianness`, except for those highlighted in light yellow. The *<Target>* string is a placeholder for the actual name of the target, and has to be done only in the case of creating target-specific files.

File Path	Type	Number of Modifications
llvm/include/llvm/Object/ELFObjectFile.h	String	1
llvm/include/llvm/BinaryFormat/ELF.h	String	2
llvm/include/llvm/BinaryFormat/ELFRelocs/ *<Target>*.def	New File	-
llvm/include/llvm/TargetParser/Triple.h	String	1
llvm/lib/BinaryFormat/ELF.cpp	String	2
llvm/lib/Object/ELF.cpp	String	2
llvm/lib/Object/RelocationResolver.cpp	String	2
llvm/lib/ObjectYAML/ELFYAML.cpp	String	2
llvm/lib/TargetParser/Triple.cpp	String	9

Table 4.1: Modifications needed to integrate a custom backend into LLVM. Highlighted in yellow are the modifications that depend on the `bit-width` or `endianness`. The *<Target>* string is a placeholder for the name of the target we want to integrate, and will be replaced before generating the new file.

File Path	**Type**	**Number of Modifications**
clang/lib/Basic/CMakeLists.txt	String	1
clang/lib/Basic/Targets.cpp	String	2
clang/lib/Basic/Targets/<*Target*>.cpp	New File	-
clang/lib/Basic/Targets/<*Target*>.h	New File	-
clang/lib/CodeGen/CMakeLists.txt	String	1
clang/lib/CodeGen/CodeGenModule.cpp	String	1
clang/lib/CodeGen/TargetInfo.h	String	1
clang/lib/CodeGen/Targets/<*Target*>.cpp	New File	-
clang/lib/Driver/Driver.cpp	String	2
clang/lib/Driver/ToolChains/Clang.cpp	String	2
clang/lib/Driver/ToolChains/Clang.h	String	1
clang/lib/Driver/ToolChains/CommonArgs.cpp	String	2
clang/lib/Driver/ToolChains/<*Target*>.h	New File	-

Table 4.2: Modifications needed to integrate a custom backend into CLANG. `<Target>` is a placeholder for the name of the target we want to integrate, and will be replaced before generating the new file.

File Path	**Type**	**Number of Modifications**
lld/ELF/Arch/<Target>.cpp	New File	-
lld/ELF/CMakeLists.txt	String	1
lld/ELF/Driver.cpp	String	1
lld/ELF/InputFiles.cpp	String	1
lld/ELF/ScriptParser.cpp	String	1
lld/ELF/Target.cpp	String	1
lld/ELF/Target.h	String	1

Table 4.3: Modifications needed to integrate a custom backend into LLD. Highlighted in yellow are the modifications that depend on the `bit-width` or `endianness`. The <*Target*> string is a placeholder for the name of the target we want to integrate, and will be replaced before generating the new file.

To see the contents that are added and the patterns that the *Add new target* script is searching in each file, please refer to the configuration files in the repository of this project.

4.2.1.1 Initial implementation

At the start of this project, the modifications were applied using Git patches, however, this approach had multiple downsides:

- **Performance** — To apply patches, we needed a Git repository with all LLVM project files indexed. Given the LLVM release consists of over 150,000 files, indexing took anywhere from 10 to 15 minutes, depending on machine specifications.
- **Reliability** — The primary drawback of using Git patches was their extreme sensitivity to any changes in a file. Even minor alterations, such as changes in line numbers, would render a patch inapplicable. Maintaining this approach would have been highly challenging, as manually editing and understanding a patch file is significantly harder than modifying a YAML file.
- **Cross-Platform Compatibility** — Git patches are affected by differences in line endings between operating systems. Windows uses `CRLF`, while Linux and macOS use `LF`. These differences could cause patches to fail or introduce unintended changes. The new implementation completely avoids this issue, ensuring consistency across platforms.

4

4.2.2 Build environment

The `build_environment` Invoke task is an integral part of the LDE created by the *Create environment* task. Executing the `help` command as described in Section 4.1.2.1, shows that the build task expects the following arguments:

- `debug` – Whether to build in *Debug* mode. The default build type is *Release*.
- `parallel_link_jobs` – During development, it was observed that the linking phase of the compilation requires large amounts of memory. Setting this argument to higher values may result in a system crash. This argument is set to *2* by default but can be adjusted depending on the system's specifications.
- `official_targets` – Which official LLVM backends to include in the build. This argument is optional, and if it is not specified, the list of official targets to build is read from the `official_targets_to_build.ini` file in the root of the LDE.
- `experimental_targets` – Targets that are not part of the LLVM release are considered *Experimental* until they are fully implemented. This means that the new target we are creating has to be in the *Experimental* targets list. This argument is optional, and if it is not specified, the list of experimental targets to build is read from the `experimental_targets_to_build.ini` file in the root of the LDE.
- `llc_only` – Whether to include CLANG and LLD in the build process or not. By default it is set to *False*.

The build task performs the following steps:

1. **Create the build directory** – Based on the build type, the script creates a *Release* or *Debug* build directory
2. **CMake** – After creating the build directory, the script generates the *CMake* command based on the build type, the official and/or experimental targets list and the number of parallel link jobs. The `CMake` configuration is then executed to set up the build.
3. **Build** – The actual build process then begins, executed by *Ninja*. Ninja is a fast, lightweight build system designed for efficiency, making it well-suited for large projects such as LLVM, where incremental builds and parallel execution are crucial for managing the extensive compilation process.

Listing 4.7 provides an example of how to write the command to invoke the task.

```
1 $ invoke build_environment --debug --parallel_link_jobs=2 --
    official_targets="RISCV;X86" --experimental_targets="MyTarget" --
    llc_only
2 or the shorter version:
3 $ inv b -p 2 -o "RISCV;X86" -e "MyTarget -l
```

Listing 4.7: Invoke command example for the `build_environment` task

4.2.3 Compile samples

The last component of the LLVM Development Environment is the `compile_samples` Invoke task. As already discussed in Chapter 3.2.2.2, this task uses the compiler built previously to compile a suite of C code samples. This is necessary to ensure that the compiler is able to handle arbitrary code in the C language.

Executing the `help` command as described in Section 4.1.2.1, shows that the *compile_samples* task expects the following arguments:

- `debug` – Whether to use the *Debug* build of the compiler. By default the task uses the *Release* build.
- `filter` – Use this argument to filter which C code files are compiled. For example, if the filter is "`test_*`", only the file names that start with "`test_`" will be considered.
- `optimization` – Which `optimization` level to use when compiling the files.
- `output` – Location on disk where to save intermediate results and log files that resulted from the test run.

- Compilation control arguments – The following arguments can be used to control the compilation process when we only want to test one part of the compilation process. Please note that only one of the following arguments can be *true*:
  - `c_to_llvm_ir` – This parameter will stop the compilation after the input C code has been converted to LLVM IR code. In other words, the compilation process is stopped after CLANG is executed.
  - `c_to_assembly` – We can set this argument to *true* when we want to test the whole compilation chain (from C code to target assembly code).
  - `llvmir_to_assembly` – We can set this argument to *true* to only run LLC. Please note that this assumes that the code files were already converted to LLVM IR.

All intermediate results will be saved in the `output` folder. Log files will capture the output from `clang`, `llc`, and `lld`. These intermediate results are crucial for backend development, providing invaluable insights into potential compilation issues.

Listing 4.8 provides an example of how to write the command to invoke the task.

```
1 $ invoke compile_samples --debug --filter="test_*" --optimization=1 --
    output=~/Temp --c-to-assembly
2 or the shorter version:
3 $ inv cs -d -f "test_*" -o 1 -u "~/Temp" -c
```

Listing 4.8: Invoke command example for the `compile_samples` task

4.3 Process of developing an LLVM Backend

Before delving into the proof of concept and the advantages of leveraging the LLVM Development Environment for backend development, it is essential to first outline the process of creating an LLVM backend from scratch. The LLVM Project offers a comprehensive guide in its official documentation, titled "Writing an LLVM Backend" [1], which details the necessary steps for developing a custom backend. Because the LLVM backend is largely responsible for Code Generation (translating LLVM IR into Machine instructions), the process of implementing a backend involves implementing all the modules discussed in Chapter 2.1.5. Figure 4.1 illustrates a high-level overview of the modules required to implement a fully functional LLVM backend. The steps that are highlighted in green are already implemented in the LDE.

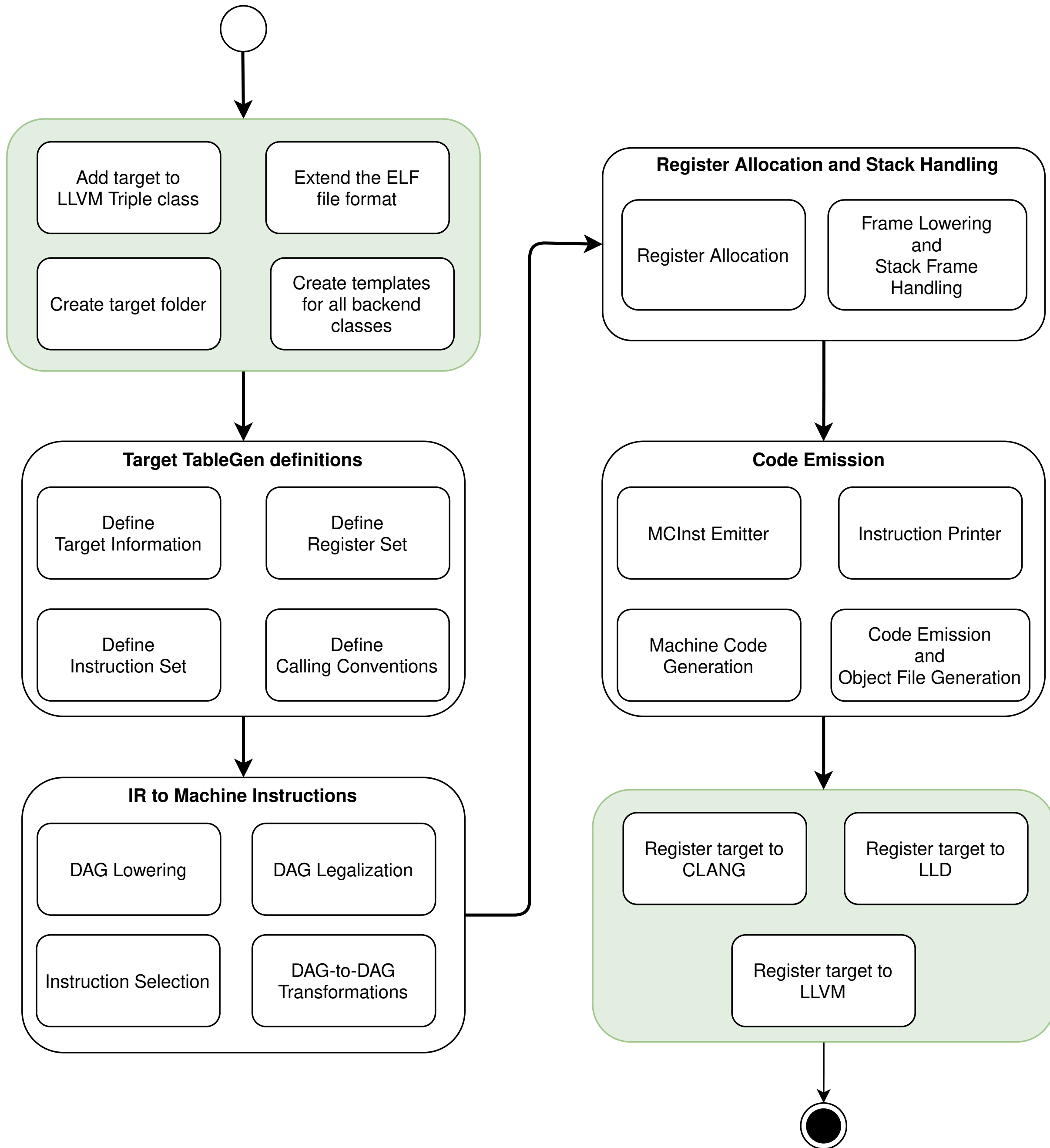

Figure 4.1: Steps necessary to write an LLVM Backend. Highlighted in green are the steps that are already done by the LLVM Development Environment.

4

All the modules observed in Figure 4.1 are briefly explained below:

1. **Target setup**
   a) Create Target Folder – The folder with the same name as the target has to be placed in `llvm/lib/Targets`
   b) Extend the ELF file format – Add support for the target-specific relocations to the libraries and tools that handle ELF object files. [11]
   c) Add target to LLVM Triple class – This tells LLVM about the new CPU architecture and can be referenced by the LLVM APIs.
   d) Register target into LLVM, CLANG and LLD – These steps are crucial for having a complete compiler that is capable of taking arbitrary high-level code (in this case C, C++ or Objective-C) and produce the assembly, object files or executable for the program at the other side. This involves multiple changes to the LLVM, CLANG and LLD source files. As stated in the previous section, the LLVM Development Environment already has the new custom backend integrated in these LLVM tools, simplifying the LLVM Backend creation process.
2. **Target TableGen definitions**
   a) Define Register Set – Define the register set of the CPU architecture in the TableGen format
   b) Define Instruction Set – Define the instruction formats and instruction set of the CPU architecture in the TableGen format
   c) Define Calling Conventions – Define the specific calling conventions of the CPU architecture in the TableGen format
3. **LLVM IR to Machine Instructions**
   a) Implement *DAG Lowering* – Based on the LLVM IR code, the Code Generator creates corresponding `SelectionDAG`s. The `DAG`s offer a structured representation of the instructions. This step is a preparatory step for Instruction Selection.
   b) Implement *DAG Legalization* – After `DAG` Lowering, some operations may still use unsupported types or instructions for the target architecture. `DAG` Legalization ensures that all nodes conform to what the hardware supports.
   c) Implement *Instruction Selection Algorithm* – The target-independent nodes in the `DAG` are converted to target-specific nodes. These nodes represent the actual **machine instructions**.

4. **Register Allocation and Stack Handling**
   a) Implement *Register Allocation Algorithm* – As discussed in previous chapters, LLVM IR is in Single Static Assignment (SSA) form. This means that the instructions use an infinite number of virtual registers. The Register Allocation Algorithm replaces all virtual registers with the physical registers the CPU architecture has, and when all registers are occupied, the variables are saved in memory.
   b) Implement *Frame Lowering and Stack Frame Handling* – This is the process of managing function stack frames during code generation. It ensures that local variables, function arguments, return addresses, and saved registers are correctly allocated and restored on the stack. **Frame Lowering** refers to transforming high-level function calls into concrete stack operations specific to the target architecture.
5. **Code Generation and Emission**
   a) Implement *`MCInst` Emitter* – This module is responsible for translating machine instructions from LLVM's internal representation to `MCInst`, which is suitable for assembly printing and binary encoding (necessary for object code emission).
   b) Implement *Instruction Printer* – The Instruction Printer module is responsible for printing the assembly representation of an `MCInst` instruction. It takes an `MCInst` (Machine Code Instruction) and converts it into human-readable assembly syntax according to the target architecture's assembly format.
   c) Implement *Machine Code Generation* – This module is responsible for translating `MCInst` objects into its respective binary encoding, according to the CPU architecture specifications.
   d) Implement *Code Emission and Object File Generation* – This module takes the assembly and binary encoding for each instruction and produces the final assembly code or object file for the respective translation unit (source or header file). It ensures proper formatting, applies relocations, and generates the necessary sections for the target architecture.

CHAPTER 5

Contribution: Developer Guide

Programming is a skill best acquired by practice and example rather than from books.

Alan Turing

This chapter represents one of the key contributions of this thesis, as it serves as a developer guide for writing an LLVM backend. In this chapter we guide developers through the process of utilizing the **LLVM Backend Development Kit** and the **LLVM Development Environment** to create their own LLVM backends. Specifically, we demonstrate this by implementing a backend for the TriCore TC 1.6 architecture from scratch, using the LDE. By working through this guide, developers will gain hands-on experience in setting up their environment, developing and debugging their own LLVM backends.

5.1 How to use the framework

In this section we provide a high-level overview of how the frameworks provided in this thesis are helping the developers develop their backends. All the components discussed in detail in previous chapters, when put together, form a useful tool not only for LLVM Backend Development, but also for other components related to LLVM such as LLVM Frontends, stand-alone LLVM passes and so on. Figure 5.1 shows the general workflow of developing LLVM Backend using the LBDK and by extension the LDE. Figure 5.1 illustrates the iterative process that a backend developer follows when implementing a custom backend. The process begins with setting up the environment using the *Create environment* script, as described in Chapter 4.1.2.2. Next, the actual backend implementation takes place, following the steps outlined in Chapter 4.3.

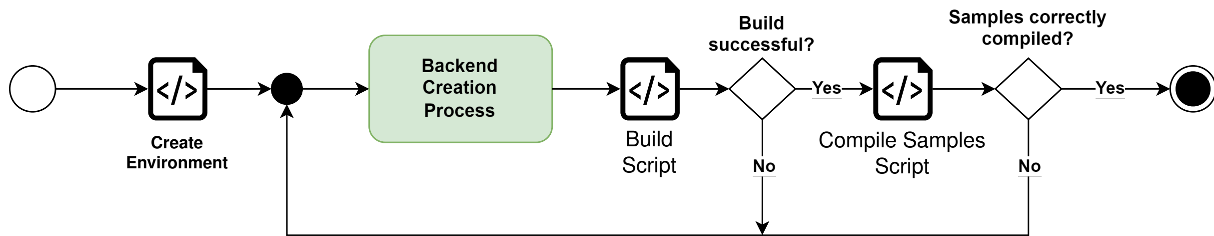

Figure 5.1: How to use the LDE to develop any LLVM Backend.

Once the *Backend Creation Process* is completed, the environment is built by running the *Build* script, as explained in Chapter 4.2.2. If the build fails, the errors must be addressed before proceeding. Otherwise, the next step is to test the compiler by compiling C code samples, as detailed in Chapter 4.2.3. At this stage, the generated assembly code is analyzed to verify its correctness. If the compilation results are correct, the development process is complete. If not, further modifications to the implementation are necessary.

5.2 TriCore LLVM Backend - Proof of concept

In the following sections, we are going to showcase the usefulness of the LDE by implementing a custom LLVM backend from scratch. As a proof of concept, we will implement an LLVM backend for the *Infineon TriCore TC1.6* architecture, explained in Chapter 2.3, and describe each step in detail. We chose TriCore as the target architecture due to its widespread use in both the automotive industry and academic settings. Despite its popularity, there is currently no open-source compiler available for TriCore, which poses a limitation for researchers, students, and developers. By providing an open-source alternative, this project aims to fill that gap and support educational and research efforts that rely on the TriCore platform. For the backend implementation, we followed the steps outlined in Section 5.1. The implementation is based on Christoph Erhardt's LLVM TriCore Backend [25], which was modified and adapted to newer versions of LLVM.

However, it is important to note that the goal of this proof of concept was to provide a practical usage example for LDE users rather than to develop a fully featured compiler for the TriCore CPU architecture. The following sections present a high-level overview of the implementation, which consists of approximately 6200 lines of code. For more details on certain parts of the implementation, please refer to the repository of the project.

5.2.1 Creating the environment

The first step towards implementing the TriCore LLVM Backend was to create the LLVM Development Environment for this project. For that, we executed the command in Listing 5.1, following the example shown in Chapter '4.1.2.2.

```
$ inv se -p ~/tricore -t TriCore -a 32 -e little -l 17
```

Listing 5.1: Command to create the LLVM Development Environment for the TriCore architecture

Note that when the project was started, `LLVM-17.0.6` was the latest LLVM release available.

This step created the LDE in the `tricore` folder in the `home` directory. We then executed an initial build of the environment using the `build_environment` script by executing the command shown in Listing 5.2.

```
# for Release builds
$ inv b
# or for Debug builds
$ inv b -d
```

Listing 5.2: Build command for the TriCore environment

This works because, as discussed in Chapter 3.1.2, TriCore is automatically added to the `experimental_targets_to_build.ini` file, and we don't have to specify it anymore.

5.2.2 Target TableGen definitions

For the implementation of the backend we followed the steps outlined in Chapter 4.3 - *Process of developing an LLVM Backend*. All the definitions we will discuss in this section will be processed by the `tblgen` tool and will generate C++ classes and methods, which we will use later in the C++ implementation of the backend.

In this step we had to define the target information, register set, instruction set and calling conventions. This information can be found in the TriCore documentation [39][38] and can be directly defined using the `TableGen` language.

5.2.2.1 Define the Register Set

As mentioned in Chapter 2.3, the TriCore TC1.6 architecture includes thirty-two general-purpose registers, consisting of sixteen data registers and sixteen address registers. To define the register set in TableGen, we have to extend the file `TriCoreRegisterInfo.td`, we must first create the register definition, as shown in Listing 5.3. In this listing, the `TriCoreReg` class extends the `Register` class, indicating to `tblgen` that `TriCoreReg` represents a register definition. There are also specialized versions of `TriCoreReg`, including `TriCoreDataReg`, which is shown in Listing 5.3.

```
class TriCoreReg<string n> : Register<n> {
    field bits<16> Num;
    let Namespace = "TRICORE";
    let HWEncoding = Num;
}

// General Purpose Data Registers
class TriCoreDataReg<bits<16> num, string n> : TriCoreReg<n> {
  let Num = num;
}
```

Listing 5.3: TriCore register definition in Tablegen

Next, we define the data registers `D0-D15`, as shown in Listing 5.4. The `DwarfRegNum` class (a TableGen class) represents the `DWARF`[1] register number, which is used for debugging, stack unwinding, and similar purposes. Each register is defined using multiple inheritance, allowing it to inherit properties from both `TriCoreDataReg` and `DwarfRegNum` at the same time. Similarly, we define the address registers and the program status registers by defining the `TriCoreAddrReg` and `TriCorePSReg` classes as shown in Listing 5.3 and then the registers `A0-A15`, `PSW`, `PCXI`, `PC` and `FCX` as shown in Listing 5.4.

```
def D0 : TriCoreDataReg<0, "d0">, DwarfRegNum<[0]>;
def D1 : TriCoreDataReg<1, "d1">, DwarfRegNum<[1]>;
...
def D15 : TriCoreDataReg<15, "d15">, DwarfRegNum<[15]>;
```

Listing 5.4: Define TriCore Data Registers

The TriCore architecture also defines **Extended** data and address registers, which are pairs of two consecutive registers put together. For example, registers `D0` and `D1` will form the extended register `E0`, as shown in Listing 5.5. Here we extend the TableGen class `RegisterWithSubregs` that helps us define a register with sub-registers.

```
class TriCoreRegWithSubregs<bits<16> num, string n, list<Register>
   subregs> : RegisterWithSubRegs<n, subregs> {
  field bits<16> Num;
  let Num = num;
  let Namespace = "TRICORE";
}

def E0:TriCoreRegWithSubregs<0, "e0", [D0,D1]>, DwarfRegNum<[32]>;
...
def E14:TriCoreRegWithSubregs<14, "e14", [D14,D15]>, DwarfRegNum<[39]>;
```

Listing 5.5: TriCore Extended Registers

Finally, we define the `RegisterClass`es, which represent groups of registers, as shown in Listing 5.6. These register classes will later be referenced in the backend's C++ code. The order in which registers appear in the list determines their priority during the Register Allocation phase of Code Generation.

[1] The `DWARF` register number for each register is defined in the TriCore EABI[42]

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

Listing 5.6: Defining register classes in TableGen

5.2.2.2 Define the Instruction Set

The instruction set definition for an LLVM backend is divided into two main parts: instruction formats (which represent the abstract structure of the instruction) and concrete instruction definitions. For TriCore, the instruction formats are specified in `TriCoreInstrFormats.td`, while the actual instruction definitions are found in `TriCoreInstrInfo.td`. To add new instructions, we need to extend the implementations in both of these files. The first step is to implement the instruction formats, which define the binary encoding of opcodes and specify the number of operands for each instruction type. All opcode formats can be found in the TriCore Architecture Manual [38], Chapter 1.2. Listing 5.7 shows the generic instruction format for every TriCore instruction. It specifies that any TriCore instruction definition has to provide a list of input and output operands, the assembly string and a pattern which will be used in the **Instruction Selection** phase of **Code Generation**. This generic format has to be extended further, as shown in Listing 5.8, which defines a special format for all 32-bit instructions. It adds the field that will hold the actual instruction opcode, which is represented by a 32-bit integer. However, this format is still too generic, and has to be extended by special formats that define the binary format of the opcode, as the `ABS` format shown in Listing 5.9. It constructs the opcode of the instruction by setting each bit to the value defined in the TriCore architecture manual. This approach should be applied for all instruction formats defined in the TriCore ISA.

```
class InstTriCore<dag outs, dag ins, string asmstr, list<dag> pattern>
    : Instruction {
  let Namespace = "TRICORE";
  /// outs and ins are inherited from the instruction class.
  dag OutOperandList = outs;
  dag InOperandList  = ins;
  let AsmString    = asmstr;
  let Pattern = pattern;
}
```

Listing 5.7: Generic instruction format for a TriCore instruction. Any TriCore instruction will be based on this generic definition and will have input and output parameter lists, a string that represents the assembly string for this instruction and a TableGen pattern (which will be used in the Instruction Selection phase of the Code Generation

5

```
class T32<dag outs, dag ins, string asmstr, list<dag> pattern>
    : InstTriCore<outs, ins, asmstr, pattern> {
  field bits<32> Inst;
  let Size = 4;
  field bits<32> SoftFail = 0;
}
```

Listing 5.8: Generic instruction format for a 32-bit TriCore instruction. In addition to the generic format, this defines the actual instruction opcode, Inst, which is a 32-bit wide integer.)

```
class ABS<bits<8> op1, bits<2> op2, dag outs, dag ins, string asmstr,
                  list<dag> pattern> : T32<outs, ins, asmstr, pattern> {

  bits<18> off18 = 0;
  bits<4> s1_d = 0;

  let Inst{31-28} = off18{9-6};
  let Inst{27-26} = op2;
  let Inst{25-22} = off18{13-10};
  let Inst{21-16} = off18{5-0};
  let Inst{15-12} = off18{17-14};
  let Inst{11-8} = s1_d;
  let Inst{7-0} = op1;
  let DecoderMethod = "DecodeABSInstruction";
}
```

Listing 5.9: Specialized instruction format for the ABS type of TriCore instructions. This format, which is based on the T32 format, defines the binary format of the instruction by setting every bit to the value defined in the documentation [38]

Next, we begin defining the instructions based on the formats we previously defined. Listing 5.10 shows the definition of the `ADDrr` and `ADD_Arr` instructions, which represent the addition of two data registers and two address registers, respectively.

```
// Arithmetic Instructions
let isCommutable = 1 in {

 let AddedComplexity = 6 in
 def ADDrr : RR<0x0B, 0x00, (outs DataRegs:$d),
   (ins DataRegs:$s1, DataRegs:$s2),
   "add $d, $s1, $s2",
   [(set i32:$d, (add i32:$s1, i32:$s2))]>;

 def ADD_Arr : RR<0x01, 0x01, (outs AddrRegs:$d),
   (ins AddrRegs:$s1, AddrRegs:$s2),
   "add.a $d, $s1, $s2",
   [(set AddrRegs:$d, (add AddrRegs:$s1, AddrRegs:$s2))]>;
} // let isCommutable = 1
```

Listing 5.10: `ADDrr` and `ADD_Arr` instructions definitions

In Listing 5.10, `isCommutable` is a flag that indicates the definitions represent commutative operations. This means that the operands of these instructions can be swapped

without changing the result of the operation; this can be important for the Instruction Scheduling or Register Allocation phases of Code Generation, where the order of operands may be fixed.

By setting `isCommutable` to 1, it signals that the operands can be rearranged freely, which is important for optimizations like instruction scheduling and register allocation.

In the listing, we can observe how the following components are defined for each instruction:

- Input and Output Parameters:
  - For the `ADDrr` instruction, the output is a data register `((outs DataRegs:$d))`, and the inputs are two data registers `((ins DataRegs:$s1, DataRegs:$s2))`.
  - For the `ADD_Arr` instruction, the output is an address register `(outs AddrRegs:$d)`, and the inputs are two address registers `(ins AddrRegs:$s1, AddrRegs:$s2)`.
- Assembly String:
  - The assembly string for `ADDrr` is `"add $d, $s1, $s2"`, and for `ADD_Arr`, it is `"add.a $d, $s1, $s2"`. These strings define the assembly syntax used when the instruction is assembled.
- Instruction Selection Pattern:
  - For `ADDrr`, the pattern is `[(set i32:$d, (add i32:$s1, i32:$s2))]`, which tells the backend how to translate the operands into the corresponding machine level operation.
  - `[(set AddrRegs:$d, (add AddrRegs:$s1, AddrRegs:$s2))]` is the pattern for the `ADD_Arr`, similarly describing how to handle the addition of two address registers.

  The pattern is composed of two parts, for example `set i32:$d` and `add i32:$s1, i32:$s2)`. The second part represents the operation that is performed by the instruction (in this case `add`), and the first part specifies where the result of the operation will be placed.

We applied the same approach to all the instructions defined in the TriCore ISA. For additional examples and details, please refer to the TriCore LLVM Backend repository, which is linked in this thesis.

5.2.2.3 Define the Calling Conventions

Similar to the register set and instruction set, the calling conventions are also defined using the TableGen language. For TriCore, the calling conventions are placed in the `TriCoreCallingConv.td` file. Due to the TriCore architecture's clear distinction between pointer arguments and non-pointer arguments [42], the pointer arguments are handled manually in the `TriCoreISelLowering` C++ class. As a result, the calling conventions defined is `TriCoreCallingConv.td` only take care of promoting arguments and passing the non-pointer arguments either in registers or on the stack, as shown in Listing 5.11.

5

```
// TriCore C Calling convention.
def CC_TriCore : CallingConv<[
    // Promote i8/i16 arguments to i32.
    CCIfType<[i8, i16], CCPromoteToType<i32>>,
    CCIfType<[i32], CCAssignToReg<[D4, D5, D6, D7]>>,

    CCIfType<[i64], CCAssignToReg<[E4, E6]>>,

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

Listing 5.11: Calling convetion definitions for TriCore

As specified by TriCore calling convention documentation [42], the 8- and 16-bit integer arguments are promoted to 32-bits, and the first four integer arguments are passed in register `D4-D7`, while the first two 64-bit arguments are passed in registers `E4, E6`. All other arguments are passed on the stack. As stated previously, pointer arguments are handled manually in the C++ code.

5.2.2.4 Define the TriCore target

Finally, we need to define the **target class** in TableGen, which will bring all the definitions we made previously together. This target class consolidates the register set, instruction set, calling conventions, and other related components. The target class definition for TriCore can be found in the `TriCore.td` file.

5.2.3 LLVM IR to Machine Instructions

The following sections will give a high-level overview of the DAG Lowering and Legalization, Instruction Selection and DAG-to-DAG Transformations, indicating the C++ files that have to be extended.

5.2.3.1 DAG Lowering and DAG Legalization

As discussed in Chapter 2.1.5, DAG Lowering is the process of creating the `Selection DAG` from the LLVM IR code that is provided as input to the backend and DAG Legalization is the process of transforming the illegal nodes (unsupported by the target) in the DAG with legal nodes. The TriCore LLVM Backend implements target-specific lowering and legalization hooks for the calling conventions because it needs to handle pointer arguments and place them in address registers.

In addition to the calling conventions, several instructions and operands must be manually lowered due to specific limitations and properties of the TriCore architecture [25]:

- **Global addresses, jump table indices, and constant pool entries** are essentially absolute virtual addresses. During manual lowering, these addresses are split into two parts: the HI (high) and LO (low) components. An ADD instruction is then used to combine these parts. "*The instruction selector will later convert the addition into a machine address used by load, store, or LEA (load effective address) instructions*" [25]. This approach ensures that the addresses are handled correctly and converted into a format compatible with the TriCore architecture.
- A unique feature of TriCore is the absence of separate **left-shift** 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*" [25]. As a result, the instruction selector in TriCore cannot directly map LLVM's **SRL** (logical right shift) and **SRA** (arithmetic right shift) nodes. To address this, `TriCoreTargetLowering` intercepts these nodes and replaces them with the corresponding left-shifts after negating the second operand [25].

The implementation of the DAG Lowering hooks can be found in the `TriCoreISelLowering` class.

5.2.3.2 Instruction Selection and DAG-To-DAG Transformations

In Chapter 2.1.5, we discussed that Instruction Selection is the process of translating LLVM IR code into target-specific machine instructions. This is when the patterns defined in the TableGen files are used to search for nodes in the DAG and replace them with target-specific instruction nodes. This process is mostly automated by the code generated by the `tblgen` tool, however, some manual handling is necessary in special cases. These include "*machine instructions that define multiple result values, complex addressing modes and others*" [25]. In the TriCore backend, the following situations are handled manually:

- Addresses in base register + immediate offset form.
- 32-bit multiplications which produce a 64-bit result.

These cases are handled in the `TriCoreISelDAGToDAG.cpp` file.

5.2.4 Register Allocation and Stack Handling

A large part of the Register Allocation and Stack Handling phases is handled by the LLVM target-independent framework, based on the register definitions we provided in the `TriCoreRegisterInfo.td`. However, there are several instances where custom handling is necessary to accommodate the specific requirements of the TriCore architecture. These customizations are implemented in the `TriCoreRegisterInfo` class and other related components.

5.2.4.1 Register Allocation

The register allocation process assigns physical registers to the virtual registers that are used in LLVM IR code. For the TriCore backend, several methods in the `TriCoreRegisterInfo` class are implemented for guiding this process:

1. **Reserved Registers** – The `getReservedRegs` method specifies which registers are reserved and cannot be used for general-purpose allocation. For the TriCore architecture, registers such as `PC` (Program Counter), `A10` (Stack Pointer), `A14` (Frame Pointer), and `PSW` (Program Status Word) are reserved because they serve specific purposes in the architecture. These registers are marked as reserved to ensure the register allocator does not assign them to other uses.
2. **Callee-Saved Registers** – The `getCalleeSavedRegs` method defines which registers must be preserved across function calls. For the TriCore backend, this is defined as an array of physical registers. These registers are saved and restored by the callee function if they are used.

5.2.4.2 Stack handling

Stack handling involves managing the function's stack frame, including allocating space for local variables, spilling registers, and handling function calls. The TriCore backend handles this process in the following ways.

1. **Frame Index Elimination** – During code generation, frame indices (stack slot placeholders) are replaced with actual offsets relative to the base pointer. This is done by the `eliminateFrameIndex` method, which chooses between the frame pointer (`A14`) and stack pointer (`A10`) based on function requirements. It computes the stack offset and updates the instruction. If the offset can't be encoded directly, helper instructions like `ADDrc` are inserted.
2. **Frame register selection** – The `getFrameRegister` method determines which register is used as the frame pointer. If the function has a frame pointer, `A14` is used; otherwise, `A10` (Stack Pointer) is used.
3. **Stack Slot Offsets** – The `MachineFrameInfo` class is used to manage stack slots and their offsets. The `eliminateFrameIndex` method retrieves the offset of a stack slot using `getObjectOffset` and updates the instruction accordingly.

5.2.5 Code Emission

The **Code Emission Phase** is the final step in the LLVM compilation pipeline for the TriCore backend. This phase is responsible for converting the machine instructions generated during the earlier phases into the final output format, which can be either human-readable assembly code or binary object files. The TriCore backend makes sure that the emitted code respects the specifications (binary encoding, assembly strings, operands, etc) of the TriCore architecture.

The Code Emission Phase is divided into two main parts: *Assembly Emission* and *Object Code Emission*. Below, we explain each part in detail and specify in which files the implementations can be found.

5.2.5.1 Assembly code emission

The assembly code emission process generates assembly code for the TriCore architecture based on the machine instructions generated in earlier phases. This step represents an intermediary step before generating object files, the assembly code being useful for debugging and analysis of the code.

Assembly code emission is handled by the `TriCoreAsmPrinter` class. This class is responsible for formatting and printing machine instructions in the correct assembly syntax for the TriCore architecture. The structure is defined in the TableGen files we discussed earlier. The implementation is handled in several methods of this class:

- **`emitInstruction`**: This method is called for each machine instruction in the body of a function. It converts the machine instruction into an intermediate representation (`MCInst`) and sends it to the output stream. The `LowerToMCInst` method is called to translate the `MachineInstr` into an `MCInst`, which is a lower-level representation of the instruction that will be used later for instruction printing.
- **`printOperand`**: This method is responsible for formatting individual operands of an instruction. It handles registers, immediate values, and memory operands, ensuring they are printed correctly.
- **`emitFunctionBody`**: This method emits the body of a function, iterating over all the machine instructions and calling `emitInstruction` for each one.

Relevant Files:

- `TriCoreAsmPrinter.cpp`: Implements the `TriCoreAsmPrinter` class, which handles the emission of assembly code.
- `TriCoreInstPrinter.cpp`: Provides helper methods for formatting instructions and operands in assembly syntax.
- `TriCoreInstrInfo.td`: Defines the instruction set and their assembly syntax, which is used by the `AsmPrinter`.

5.2.5.2 Object Code Emission

The object code emission process generates binary object files for the TriCore architecture. These files are in the ELF format and are used as input for the linker to produce the final executable. Object code emission is handled by two classes:

- **`TriCoreMCCodeEmitter`**: Encodes machine instructions into their binary representation. The encoding details for each instruction are defined in the `TriCoreInstrFormats.td` file.
- **`TriCoreELFObjectWriter`**: Handles the creation of ELF object files and manages ELF relocations. Relocations are used to resolve addresses and symbols during the linking phase. The `getRelocType` method determines the type of relocation we need to apply.

CHAPTER 6

Evaluation

The absence of evidence is not the evidence of absence.

Carl Sagan

In the Evaluation chapter we will assess the effectiveness and impact of the framework presented in this thesis. This chapter is divided into two sections. First, we examine how the framework streamlines the backend development process by automating the initial setup stage as well as building and testing the new compiler. By offering a structured starting point as well as a guide for implementing LLVM backends, we enable developers to focus on the implementation of the target-specific features.

The second section of this chapter presents an evaluation of the proof of concept developed using the framework. Specifically, we compiled a suite of C samples to assess the capability of our backend to successfully compile them. The results of this compilation test will be presented in detail, and any shortcomings will be discussed.

6.1 Evaluation of Workflow Streamlining for Backend Development

In this section we evaluate how the proposed **LLVM Backend Development Kit**, and implicitly the **LLVM Development Environment**, streamline the development workflow for new LLVM Backend developers.

Before the introduction of the **LLVM Backend Development Kit**, the process of developing a new LLVM backend was both time-consuming and error-prone. Developers were required to get familiar with LLVM's internal architecture and build system, often spending significant effort just to configure a minimally functional environment. The official documentation from the LLVM [1] development team offers little guidance, raising the barrier to entry for new developers, as it advises copying code from other backends and adapt it to fit the new target's requirements.

This process is, however, very often error-prone. Choosing this approach may result in developers spending a lot of time debugging their implementation.

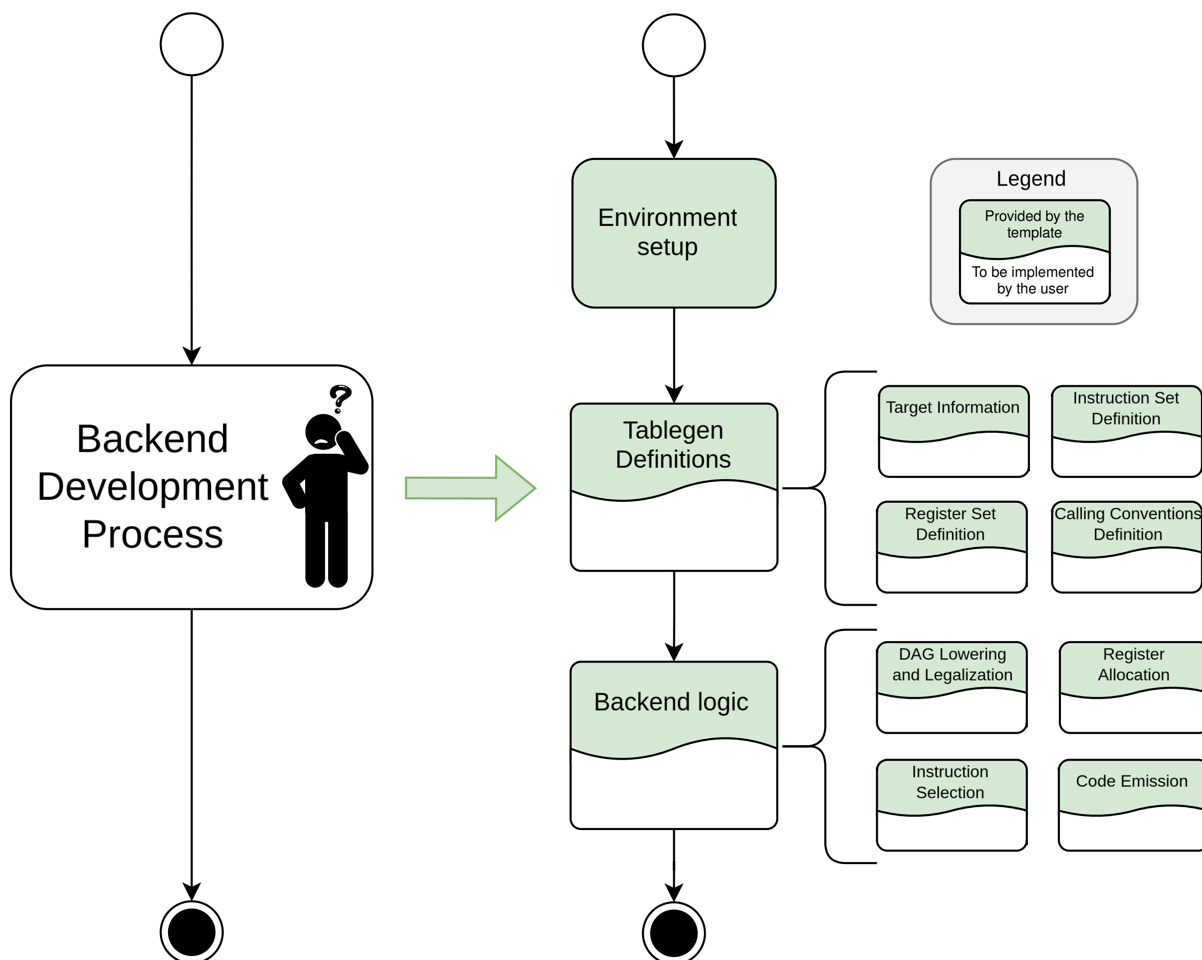

Figure 6.1: How the solution proposed in this thesis streamlines the LLVM backend development process. Portions colored in green are components provided by the template and the ones in white indicate the target-specific details that the developer has to implement.

6

Figure 6.1 illustrates a high-level comparison of the traditional backend development process (left) with the streamlined workflow enabled by the LBDK (right). In the traditional approach, developers start with little guidance and must manually construct the entire environment. This involves downloading the LLVM release, manually creating the structure of the backend, integrating the new backend in LLVM's build system, registering it to an existing frontend (such as CLANG), before they can meaningfully begin implementing the actual backend logic.

By contrast, the LBDK automates the environment setup stage entirely. It generates a ready-to-use **LLVM Development Environment** in which the new backend is already integrated in the build system. As shows in Figure 6.1, the portions colored in green represent components provided by the environment, while the white areas

indicate the sections left for the developer to implement. This clear separation reduces ambiguity and allows developers to focus solely on target-specific implementation details. This thesis also provides a developer guide which significantly reduces the barrier to entry and points developers in the right direction during the implementation of the backend. The template includes stub implementations for all core classes of the backend. This means that developers can immediately begin filling in target-specific logic without needing to reverse-engineer the structure from existing backends. This option is still available, but it enables developers to concentrate solely on individual classes or functions, without being overwhelmed by unfamiliar code.

Another significant benefit of using the LDE is represented by the **build** and **compile samples** scripts that are included. The generated backend is syntactically complete and buildable out of the box. This enables developers to make use of incremental builds from the very beginning, checking that their implementation builds correctly at every step. This shortens feedback loops and prevents integration errors. Additionally, developers can verify whether their compiler generates any assembly code at each step. While the script doesn't validate the correctness of the generated assembly code, it still provides early insights into any changes made whenever a build is run.

Overall, the LBDK, and subsequently the LDE, significantly improves the backend development process (explained in 5 and Chapter 2.1.5) by automating the environment setup and the build process, shifting developer's focus from writing boilerplate code and understanding LLVM's internal architecture to implementing the functionality of the backend, thus improving efficiency throughout the development cycle.

6.2 Compilation results for the TriCore LLVM Backend

To evaluate the capabilities of the proof-of-concept backend we implemented, we selected 20 sample C programs from Daniel Traussnig's collection [43], originally used in the validation chapter of his thesis. Given that our TriCore backend is not yet feature-complete, we intentionally chose simpler samples to verify that the compiler can successfully generate valid assembly code in basic scenarios—such as simple loops, if statements, or return statements.

The code samples used in this evaluation are listed in Table 6.1. As shown, our compiler, with optimizations disabled, successfully generated valid assembly for **9 out of 20** samples, while the remaining **11** failed during compilation. When compiling with Level 1 optimization, the compiler successfully generates valid assembly code for **14 out of 20** samples. This improvement is likely due to CLANG's optimization passes, which eliminate instructions that have no effect on the overall program behavior—a common pattern in simple or minimal code samples.

In the following sections, we take a closer look at the test cases we compiled. For the failed cases, we discuss possible causes of failure. For three of the successful cases, we compare the original C code with the generated assembly to assess the correctness of the output.

ID	File Name	Description	Result
1	callback	Function call with a function pointer as parameter	Fail
2	clb_switch	Switch statement in callback function	Fail
3	funccall	A normal function call	Fail
4	goto	Uses goto statements	Pass
5	high_callstack	Nested function calls	Pass
6	if	Simple if-else statement	Pass
7	if_func	If-else with function calls	Fail
8	indir	Function call through a static local function pointer	Fail
9	loop_cont	Loop with continue statement	Pass
10	loop_dyn	Loop without static knowledge about the number of iterations	Pass
11	loop_func	Function call inside a loop	Fail
12	loop_funcp	Iteration over function pointer array	Fail
13	loopbreak	For loop that stops with a break statement	Pass
14	loop_static	Loop with static number of iterations	Pass
15	many_funcs	Several consecutive and nested function calls	Pass
16	mult_calls	Several consecutive function calls	Fail
17	mult_ret	Several return statements at different positions	Pass
18	mult_same_call	Same function gets called twice consecutively	Fail
19	recursion	Function which calls itself	Fail
20	switch_5	A switch statement with 5 cases	Fail

Table 6.1: Collection of sample C code files (source files along with their descriptions are taken from Daniel Traussnig's work [43]) that were compiled using the TriCore LLVM backend (with optimizations disabled). Out of 20 samples, 9 successfully compiled valid assembly code, while 11 failed during compilation (assembly code was not generated for these files)

6.2.1 Failed tests cases

Upon analyzing the logs of the failed cases, we observed that they can be grouped based on the cause of the errors:

- Test cases involving function pointers failed because this feature is currently not supported by the backend, this being a known issue. The test cases affected are:
  - `callback`
  - `clb_switch`
  - `indir`
  - `loop_funcp`

- The implementation of function calls with parameters is currently not working properly. In particular, register allocation fails for function calls, which may be due to incomplete or incorrect support for the calling convention, or another underlying issue. The error that the compiler prints to the console is "*Bad machine code: Using an undefined physical register*". This issue requires further investigation to determine the root cause. The test cases that fall under this category are:
  - **`funccall`**
  - **`if_func`**
  - **`loop_func`**
  - **`mult_calls`**
  - **`mult_same`**
  - **`recursion`**
- The final test case failed during the instruction selection stage of code generation. The compiler reported an error indicating that it was unable to select a `br_jt` (JumpTable) instruction from the LLVM IR. According to the LLVM documentation, `switch` instructions may be lowered in various ways, including a sequence of conditional branches, an indirect branch via a lookup table, or a jump table. The selection depends on the characteristics of the target architecture and the nature of the switch statement.
  In this specific case, it appears that LLVM attempted to lower the switch statement into a jump table. However, because our backend does not yet implement support for jump tables or indirect branching via lookup tables, instruction selection failed. Supporting this kind of lowering may require handling of jump table construction and addressing. The test case affected by this problem is:
  - **`switch_5`**

The fact that several test cases fail with similar errors suggests that they may share a common root cause. If this is the case, addressing the underlying issue could potentially resolve multiple failures at once. However, this remains speculative and would require further investigation to accurately identify and confirm the root cause.

6

6.2.2 Passed tests cases

From our experiments, we observed that **9 out of the 20** test cases produced valid assembly code when compiler optimizations were disabled. The passing test cases are listed in Table 6.1. To evaluate the correctness of the generated assembly, we compare the original C code with the corresponding assembly output for the following three test cases:

- `many_funcs` – This test case demonstrates that our backend can handle function calls with no parameters, which is one of the current implementation's limitations.
- `mult_ret` – This test case demonstrates that our custom backend is capable of compiling if statements and handling function returns based on conditional branches.
- `loop_cont` – This test case is interesting because it shows that our compiler fully support `for` loops.

With this selection of passing tests, we demonstrate that our compiler successfully handles basic C code and can be utilized in certain simple scenarios. This establishes a foundation for future developments and further research to expand its capabilities. In the following sections, we will compare the C code with the corresponding assembly code to verify that the assembly code is functionally equivalent to the C code. Figures 6.2, 6.3 and 6.4 show a side-by-side comparison between the original C code on the left and the corresponding TriCore assembly code on the right, for `many_funcs`, `mult_ret` and `loop_cont` test cases respectively. For each section in the C code, the corresponding section in the TriCore assembly is highlighted and explained in detail by the green boxes on the right. The Figures show that the TriCore assembly code produced by our compiler is functionally equivalent to the original C code we supplied as input.

6.2.2.1 Test case `many_funcs`

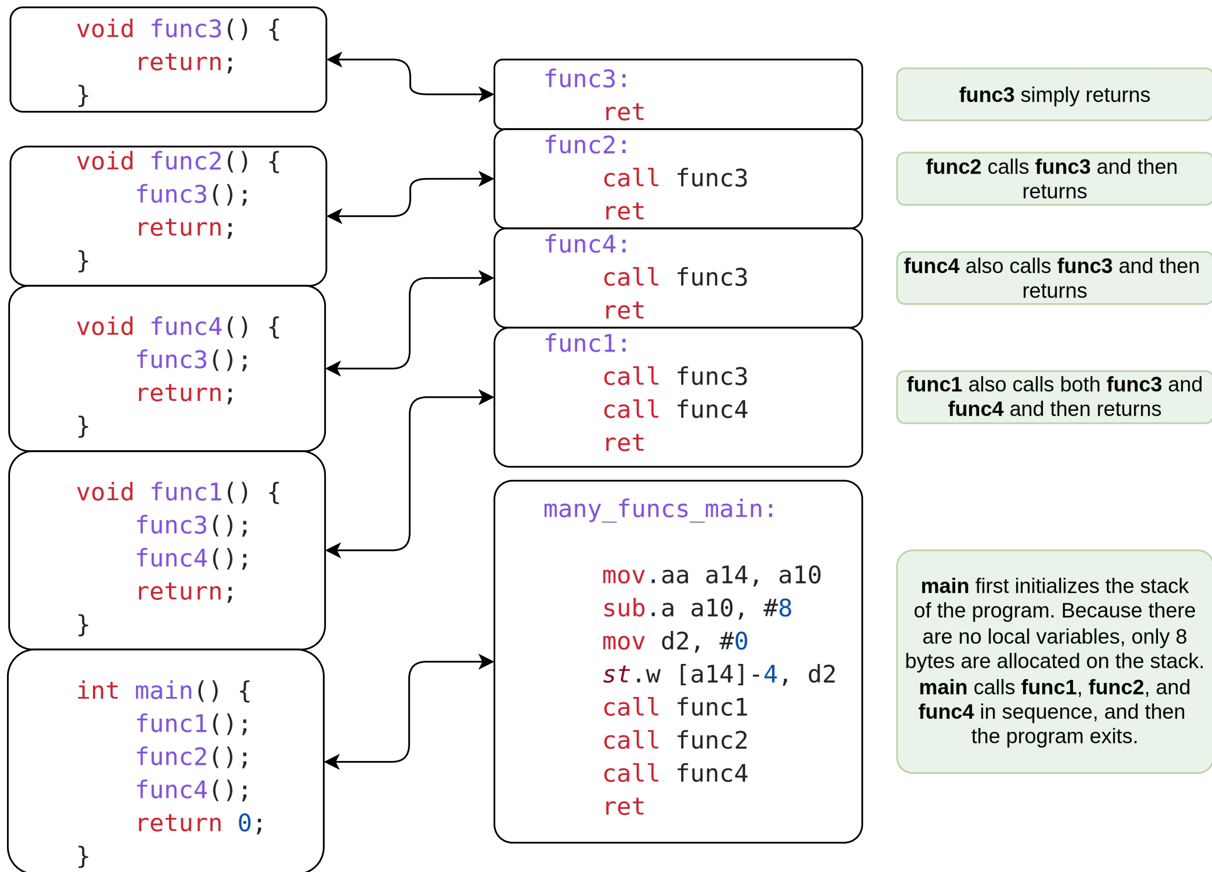

Figure 6.2: Assembly code generated for the `many_funcs` test case

6.2.2.2 Test case `mult_ret`

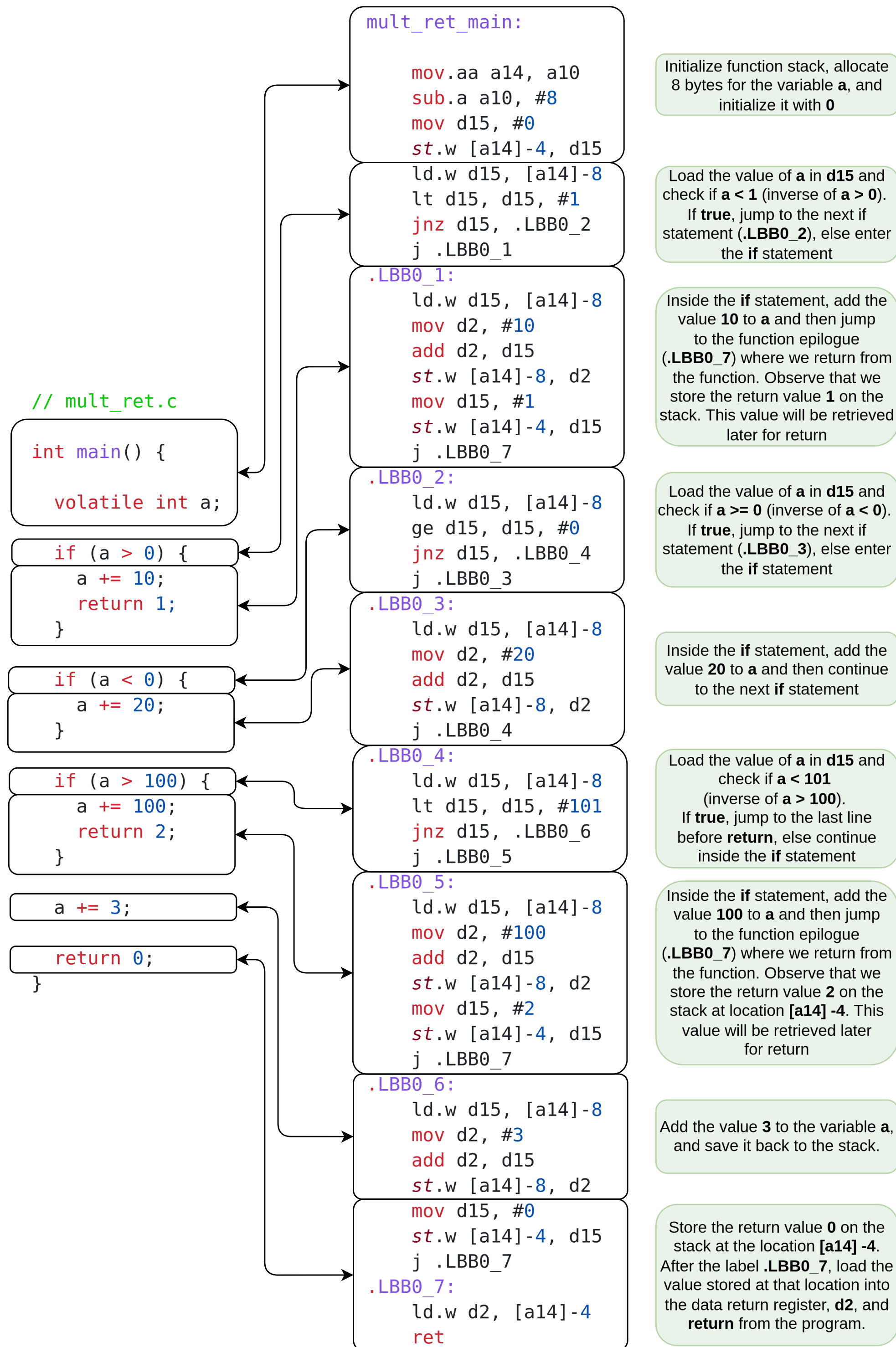

Figure 6.3: Assembly code generated for the `mult_ret` test case

6.2.2.3 Test case `loop_cont`

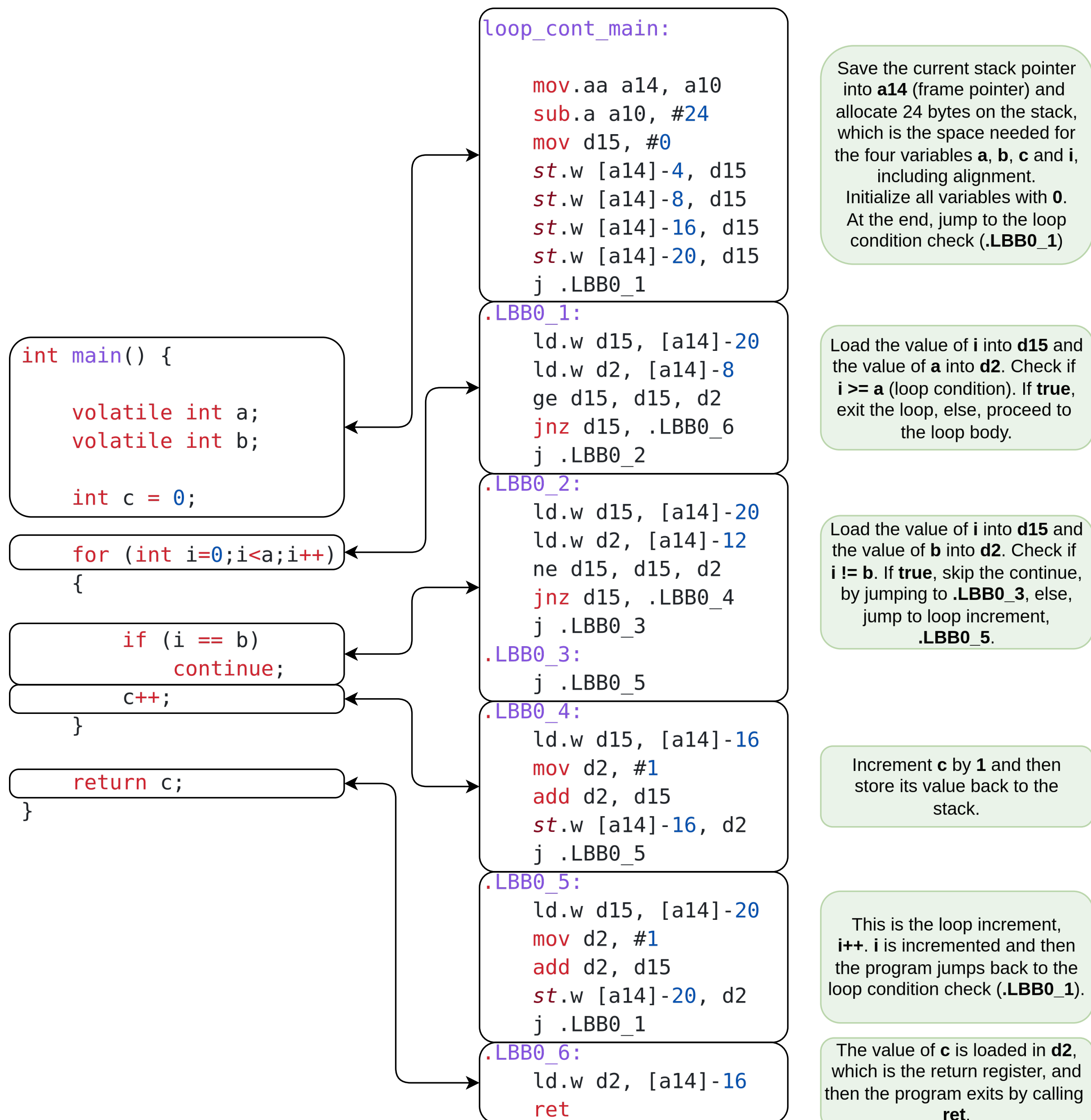

Figure 6.4: Assembly code generated for the `loop_cont` test case

6

CHAPTER 7

Conclusion and Future Work

Everyone you will ever meet knows something you don't.

Bill Nye

The **LLVM Backend Development Kit** (LBDK) and the **LLVM Development Environment** (LDE) were introduced as tools to simplify and accelerate the development of LLVM Backends. By automating the initial setup process, providing reusable templates for multiple LLVM release versions, and including convenience scripts, the framework reduces the complexity and time required for backend development. A proof-of-concept LLVM backend implementation for the TriCore TC1.6 Architecture successfully demonstrated the framework's effectiveness not only in the initial setup of the environment, but also in guiding developers through the development process of the backend.

This contribution addresses a long-standing challenge in the LLVM community, particularly within the backend development domain — the high barrier to entry. The evaluation demonstrated the framework's effectiveness in reducing this barrier, allowing developers to concentrate on target-specific implementation details rather than investing significant time and effort in setting up environments or manually structuring a backend, which typically requires an in-depth understanding of LLVM's internal architecture.

Additionally, this thesis provides a detailed developer guide for writing LLVM backends, addressing a current gap in existing resources. Some guides, such as the Official LLVM documentation on writing LLVM backends [1], lack sufficient depth, offering high-level information and few hands-on examples, which can leave developers unsure of how to proceed. Others, like the Cpu0 LLVM Backend tutorial [32] (which is over 700 pages long), are overly complex and detailed, making it challenging for newcomers to follow and understand the next steps in the development process.

Both resources are valuable sources of information, but in our experience, they fall short of addressing the specific needs of new LLVM developers. Beginners often require a guide that is both clear and structured. The developer guide, combined with the template provided in this thesis, creates an easy-to-follow process for developing LLVM backends.

It provides developers with clear steps and practical examples for implementing each stage, making the development process more straightforward and manageable. By doing so, this thesis bridges the gap in existing resources, offering a comprehensive yet accessible guide with a clear and structured approach to backend development.

While the framework provided in this thesis significantly improves the LLVM backend development process, there are several opportunities for **future research**:

- The **Skeleton** template could be extended to support more common patterns that are applicable across a wide range of backends, such as basic instruction selection, register allocation stubs, or common calling conventions. This would make the framework more versatile and reduce the effort required to adapt it to different architectures.
- Incorporating improved testing mechanisms could enhance the reliability of the generated backends. For example, integrating unit tests for backend components or providing a framework for automated validation of generated code could streamline the debugging process and ensure higher-quality results.
- Expanding the documentation and tutorials to cover more advanced use cases would further support developers, especially those new to LLVM. This could include detailed examples of implementing specific backend features or addressing common challenges encountered during development.
- Another future work area could involve addressing the issues identified during the evaluation of the TriCore backend. Resolving these challenges, such as incomplete instruction support or register allocation for function calls, would enhance the backend's functionality and reliability. Additionally, this effort would result in the TriCore architecture having a fully-featured open-source compiler, which would be highly valuable not only in research environments but also in broader applications, enabling further innovation and collaboration.

In conclusion, this thesis contributes a practical and impactful solution to the LLVM community by addressing the challenges of backend development. The LBDK and LDE enable developers to create custom LLVM backends more efficiently, as demonstrated by the proof-of-concept for the TriCore TC1.6 architecture. With further enhancements and extensions, the framework has the potential to become an important tool for compiler developers, driving innovation and enabling the creation of high-quality backends for a wide range of architectures.

List of Figures

2.1 Structure of a generic compiler 6
2.2 Basic LLVM Architecture based on a three-stage design 8
2.3 Retargetable LLVM Architecture 8
2.4 LLVM Code Generation Process 13

3.1 Overview of the generation process for the LDE 28
3.2 Skeleton class hierarchy 30
3.3 LLVM Development Environment 33

4.1 How to develop an LLVM backend using the LDE 47

5.1 How to use the LDE to develop any LLVM Backend 52

6.1 Before and after: Simplified backend development process 64
6.2 Assembly code generated for the `many_funcs` test case 69
6.3 Assembly code generated for the `mult_ret` test case 70
6.4 Assembly code generated for the `loop_cont` test case 71

List of Tables

4.1 Modifications needed to integrate a custom backend into LLVM 42
4.2 Modifications needed to integrate a custom backend into CLANG 43
4.3 Modifications needed to integrate a custom backend into LLD 43

6.1 The collection of sample C code files and their compilation result 66

List of Listings

4.1 Python Build Invoke Task 38
4.2 Example of a command to invoke an Invoke task 38
4.3 Example of help message for an invoke task 39
4.4 Invoke command example for the `setup_new_environment` task 40
4.5 Example for configuring the build for `X86` and `RISCV` 40
4.6 Invoke command example for the `add_new_target` task 41
4.7 Invoke command example for the `build_environment` task 45
4.8 Invoke command example for the `compile_samples` task 46

5.1 Command to create the LLVM Development Environment 53
5.2 Build command for the TriCore environment 53
5.3 TriCore register definition in Tablegen 54
5.4 Define TriCore Data Registers 54
5.5 TriCore Extended Registers 54
5.6 Defining register classes in TableGen 55
5.7 Generic instruction format for a TriCore instruction 55
5.8 Generic instruction format for a 32-bit TriCore instruction 56
5.9 Specialized instruction format for the ABS type of TriCore instructions . 56
5.10 `ADDrr` and `ADD_Arr` instructions definitions 56
5.11 Calling convetion definitions for TriCore 58

List of Abbreviations

API Application Programming Interface.
AST Abstract Syntax Tree.

CLI Command Line Interface.
CPU Central Processing Unit.

DAG Directed Acyclic Graph.
DSP Digital Signal Processor.

ELF Executable and Linkable Format.

GCC GNU Compiler Collection.
GPR General Purpose Register.

IDE Integrated Development Environment.
IR Internal/Intermediate Representation.
IS Instruction Selection.
ISA Instruction Set Architecture.

JIT Just-In-Time Compilation.

LBDK LLVM Backend Development Kit.
LDE LLVM Development Environment.
LLVM Low Level Virtual Machine.

MCU Microcontroller Unit.
MVT Machine Value Type.

PC Program Counter.
PCXI Previous Context Information.
PSW Program Status Word.

RISC Reduced Instruction Set Architecture.

SP Stack Pointer.
SSA Single Static Assignment.

Bibliography

[1] C. Lattner. LLVM documentation - writing and LLVM backend. Accessed: 2024-10-25. [Online]. Available: https://llvm.org/docs/WritingAnLLVMBackend.html → Cited on pages 2, 3, 15, 21, 36, 46, 63, and 73.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Pearson/Addison Wesley, 2006. → Cited on pages 5, 6, 7, 10, 11, and 13.

[3] T. Douglas, *Introduction to Compilers and Language Design*. University of Notre Dame, 2023. [Online]. Available: https://compilerbook.org → Cited on pages 5, 6, 7, and 10.

[4] P. E. Ceruzzi, *A history of modern computing*. MIT Press, Cambridge, Massachusetts, London, 2003. → Cited on page 5.

[5] J. E. Sammet, *Programming Languages: History and Fundamentals*. Prentice Hall, 1969. → Cited on page 5.

[6] C. Lattner, "The architecture of open source applications (volume 1)," 2012, accessed: 2024-10-25. [Online]. Available: https://aosabook.org/en/v1/llvm.html → Cited on pages 7, 8, 9, 13, and 14.

[7] ——, "LLVM: An infrastructure for multi-stage optimization," Master's thesis, University of Illinois at Urbana-Champaign, Computer Science Dept., 2002. → Cited on pages 7 and 8.

[8] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86. → Cited on pages 7 and 9.

[9] B. C. Lopes and R. Auler, *Getting Started with LLVM Core Libraries*. Packt Publishing Ltd, 2014. → Cited on pages 8 and 11.

[10] C. Lattner. LLVM documentation. Accessed: 2024-10-25. [Online]. Available: https://llvm.org/docs/ → Cited on page 9.

[11] K. Nacke and A. Kwan, *Learn LLVM 17: A beginner's guide to learning LLVM compiler tools and core libraries with C++*. Packt Publishing Ltd, 2024. → Cited on pages 9, 15, 16, 18, and 48.

[12] J. Schlamelcher, T. Goodfellow, B. Kebianyor, and K. Gruettner, “Extending Clang/LLVM with custom instructions using tablegen – an experience report,” in *MBMV 2024; 27. Workshop*, 2024, pp. 204–213. → Cited on page 9.

[13] D. Bokan, M. Đukić, M. Popović, and N. Četić, “Adjustment of GCC compiler frontend for embedded processors,” in *2014 22nd Telecommunications Forum Telfor (TELFOR)*, 2014, pp. 983–986. → Cited on page 10.

[14] R. Nitu, E. Staniloiu, C. Creteanu, and R. Rughinis, “Building an interface for the D compiler library,” in *2021 20th RoEduNet Conference: Networking in Education and Research (RoEduNet)*, 2021, pp. 1–6. → Cited on page 10.

[15] S. Algaraibeh, C. Jeffery, T. Soule, and T. Dousay, “A parsing technique for enhancing compiler syntax error messages for student programmers,” in *2024 IEEE Frontiers in Education Conference (FIE)*, 2024, pp. 1–7. → Cited on page 10.

[16] P. Osmialowski, “How the flang frontend works: Introduction to the interior of the open-source fortran frontend for LLVM,” in *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM-HPC’17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3148173.3148183 → Cited on page 10.

[17] G. Reedy, “Design and implementation of a scala compiler backend targeting the low level virtual machine,” Master’s thesis, The University of New Mexico, 2014. → Cited on page 10.

[18] D. van Oorschot and M. Huisman, “LLVM IR,” in *Fundamental Approaches to Software Engineering: 27th International Conference, FASE 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings*, vol. 14573. Springer Nature, 2024, p. 290. → Cited on page 11.

[19] (2024) LLVM developer policy). Accessed: 2024-11-15. [Online]. Available: https://llvm.org/docs/DeveloperPolicy.html → Cited on pages 11 and 36.

[20] D. Khaldi, Y. Luo, B. Yu, A. Sotkin, B. Morais, and M. Girkar, “Extending LLVM IR for DPC++ matrix support: A case study with intel® advanced matrix extensions (intel® AMX),” in *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, 2021, pp. 20–26. → Cited on page 12.

[21] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, “Obfuscator-LLVM – software protection for the masses,” in *2015 IEEE/ACM 1st International Workshop on Software Protection*, 2015, pp. 3–9. → Cited on page 12.

[22] K. Jingu, K. Shigenobu, K. Ootsu, T. Ohkawa, and T. Yokota, “An implementation of LLVM pass for loop parallelization based on IR-level directives,” in *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)*, 2018, pp. 501–505. → Cited on page 12.

[23] Y. Yan, Z. Pan, L. Yu, and T. Wang, "Research on the influencing factors of LLVM IR optimization effect," in *2023 IEEE 3rd International Conference on Information Technology, Big Data and Artificial Intelligence (ICIBA)*, vol. 3, 2023, pp. 756–763. → Cited on page 12.

[24] H. Karacalı, E. Cebel, and N. Donum, "Optimizing LLVM IR: Transforming multiplication to addition for enhanced execution efficiency," in *2024 9th International Conference on Computer Science and Engineering (UBMK)*, 2024, pp. 1098–1103. → Cited on page 12.

[25] C. Erhardt, "Design and implementation of a TriCore backend for the LLVM compiler framework," Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2009. → Cited on pages 13, 17, 19, 20, 25, 52, and 59.

[26] C. Lattner. LLVM documentation - the LLVM target-independent code generator. Accessed: 2025-02-14. [Online]. Available: https://llvm.org/docs/CodeGenerator.html → Cited on pages 14, 15, 16, 17, 18, and 19.

[27] D. R. Koes and S. C. Goldstein, "Near-optimal instruction selection on dags," in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 45–54. [Online]. Available: https://doi.org/10.1145/1356058.1356065 → Cited on page 17.

[28] A. J. Root, M. B. S. Ahmad, D. Sharlet, A. Adams, S. Kamil, and J. Ragan-Kelley, "Fast instruction selection for fast digital signal processing," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, ser. ASPLOS '23. New York, NY, USA: Association for Computing Machinery, 2024, p. 125–137. [Online]. Available: https://doi.org/10.1145/3623278.3624768 → Cited on page 17.

[29] R. Daly, C. Donovick, C. Terrill, J. Melchert, P. Raina, C. Barrett, and P. Hanrahan, "Efficiently synthesizing lowest cost rewrite rules for instruction selection," in *Proceedings of the 24th Conference on Formal Methods in Computer-Aided Design (FMCAD 2024)*. TU Wien Academic Press, 2024, pp. 8–17. [Online]. Available: https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_7 → Cited on page 17.

[30] T. C. d. S. Xavier, G. S. Oliveira, E. D. d. Lima, and A. F. d. Silva, "A detailed analysis of the LLVM's register allocators," in *2012 31st International Conference of the Chilean Computer Science Society*, 2012, pp. 190–198. → Cited on page 18.

[31] B. Chichereau, S. Vialle, and P. Carribault, "Fully integrated quantum method for classical register allocation in LLVM," in *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, vol. 02, 2024, pp. 293–299. → Cited on page 18.

[32] C. Chung-Shu, "Tutorial: Creating an LLVM Backend for the Cpu0 Architecture," 2024, Accessed: 2024-12-10, Version: 12.0.15.4. [Online]. Available: https://jonathan2251.github.io/lbd/TutorialLLVMBackendCpu0.pdf → Cited on pages 19, 20, 21, and 73.

[33] J. Kolek, Z. Jovanović, N. Šljivić, and D. Narančić, "Adding microMIPS backend to the LLVM compiler infrastructure," in *2013 21st Telecommunications Forum Telfor (TELFOR)*, 2013, pp. 1015–1018. → Cited on pages 19 and 21.

[34] L. Naimi, H. Abdelmalek, and A. Jakimi, "A DSL-based approach for code generation and navigation process management in a single page application," *Procedia Computer Science*, vol. 231, pp. 299–304, 2024, 14th International Conference on Emerging Ubiquitous Systems and Pervasive Networks / 13th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (EUSPN/ICTH 2023). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050923022196 → Cited on page 23.

[35] L. Luhunu and E. Syriani, "Comparison of the expressiveness and performance of template-based code generation tools," in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 206–216. [Online]. Available: https://doi.org/10.1145/3136014.3136021 → Cited on page 23.

[36] J. P. Smith, *Entity Framework core in action*. Simon and Schuster, 2021. → Cited on page 24.

[37] K. D. Jayaraman and E. Siddharth, "Harnessing the power of entity framework core for scalable database solutions," *Journal of Quantum Science and Technology (JQST)*, vol. 2, p. Jan(151–171), Jan. 2025. [Online]. Available: https://jqst.org/index.php/j/article/view/156 → Cited on page 24.

[38] I. T. AG, *TriCore V1.6 Core architecture; 32-bit Unified Processor Core; Volume 2*. Infineon Technologies AG, 2012-05. → Cited on pages 25, 53, 55, and 56.

[39] ——, *TriCore V1.6 Core architecture; 32-bit Unified Processor Core; Volume 1*. Infineon Technologies AG, 2012-05. → Cited on pages 25, 26, and 53.

[40] D. Oka and T. Matsuki, "A security assessment study and trial of Tricore-powered automotive ECU," in *CODE BLUE 2014*, 12 2014. → Cited on page 26.

[41] B. Binder, M. Asavoae, F. Brandner, B. Ben Hedia, and M. Jan, "Scalable detection of amplification timing anomalies for the superscalar tricore architecture," in *Formal Methods for Industrial Critical Systems*, M. H. ter Beek and D. Ničković, Eds. Cham: Springer International Publishing, 2020, pp. 151–169. → Cited on page 26.

[42] I. T. AG, *TriCore 32-bit Unified Processor Core Embedded Applications Binary Interface (EABI)*. Infineon Technologies AG, 2007-02. → Cited on pages 54, 57, and 58.

[43] D. Traussnig, “Control-flow graph validation of ELF images,” Master’s thesis, Technische Universitaet Graz, 2023. → Cited on pages 65 and 66.