Informatics

Flexible Generierung von Enwicklerwerkzeugen mit VADL

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Tobias Schwarzinger, BSc.
Matrikelnummer 11778257

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Wien, 5. Dezember 2022

Tobias Schwarzinger | Andreas Krall

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
Bibliothek Your knowledge hub

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Informatics

Flexible Generation of Low-Level Developer Tools with VADL

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Tobias Schwarzinger, BSc.
Registration Number 11778257

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Vienna, 5th December, 2022

Tobias Schwarzinger   Andreas Krall

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Tobias Schwarzinger, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 5. Dezember 2022

Tobias Schwarzinger

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

First and foremost, I want to thank my parents, who unconditionally supported me throughout my studies. None of this would have been possible without their financial support and life-long encouragement.

Next, I thank Carina, my brothers, and my friends. Composing this thesis was challenging, technically and emotionally. Thanks for always having an open ear and enduring me on my bad days.

Special thanks to my advisor Prof. Andreas Krall for allowing me to work on this exciting project. Furthermore, I want to thank him for his invaluable advice throughout the writing process. Moreover, I want to thank the VADL team for helping me developing the prototype and sharing their knowledge.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Die Entwicklung von Domain-spezifischen Mikroprozessoren ist ein herausforderndes und ressourcenintensives Unterfangen. Einer der wichtigsten Schritte dabei ist es, ein effizientes Prozessordesign zu finden. Idealerweise können Hardwarearchitekt:innen während dieses Prozesses schnell Ideen probieren und evaluieren. Die Vienna Architecture Description Langauge (VADL) unterstützt Ingenieure und Ingenieurinnen während dieses Prozesses. Das Ziel der Sprache ist es, eine kontinuierliche Verfeinerung der Prozessorarchitektur zu erlauben. Während dieses Prozesses, können Generatoren die Information in einer VADL Spezifikation nutzen, um Simulatoren, Compiler und Hardwareschemata zu generieren.

Vor dieser Arbeit konnte der Compilergenerator keinen Assembler und Linker erzeugen. Daher war es Nutzer:innen nicht möglich, C oder Assembly Programme in ausführbare Dateien zu übersetzen. Obwohl dieser Umstand durch handgeschriebene Werkzeuge überbrückt werden kann, verlangsamt sich die Designexploration erheblich. Das Ziel dieser Arbeit war es den Compilergenerator so zu erweitern, dass dieser einen Assembler und Linker (Low-Level Entwicklerwerkzeuge) erzeugen kann.

In dieser Arbeit entwickelten wir einen Prototyp, der erfolgreich Low-Level Entwicklerwerkzeuge für RISC-ähnliche Architekturen erzeugen kann. Beide Komponenten unterstützen komplexe Funktionen sowie die das Schreiben und Anwenden von Relocations. Um diese Anforderungen zu unterstützen, erweiterten wir VADL um ein Assembly-Beschreibungselement. Weiters generiert unser Prototyp automatisch Parserregeln aus der Formatierungsfunktion einer Instruktion. Darüber hinaus implementierten wir auch zwei Generierungsstrategien für Relocations welche es Benutzer:innen erlauben, Symbole in manchen Instruktionen zu verwenden, ohne Relocations manuell anlegen zu müssen. Unser Beitrag ermöglicht eine effizientere Designexploration, indem es Nutzer:innen für den Prozessor zugeschnittene Entwicklungswerkzeuge zur Verfügung stellt.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

Developing microprocessors tailored to a domain-specific problem is a challenging and resource-intensive task. A crucial step is coming up with an efficient processor design. Rapid design exploration allows hardware architects to quickly test and evaluate new ideas, ranging from specialized instructions to unique pipeline structures. The Vienna Architecture Description Language (VADL) supports engineers during this process. Its goal is to allow continuous refinement of a processor's architecture. During this process, generators can leverage the information in the VADL specification to emit simulators, compilers, and hardware schematics.

However, prior to this work, the compiler toolchain generator could not emit an assembler and linker. As a result, users could not generate executables from C or assembly programs. While this gap can be bridged by manually writing the necessary software, this significantly slows down the design exploration process. This additional overhead is most apparent when introducing new instructions or altering the architecture's encoding. This work aimed to alleviate this issue by extending the compiler tool generator with an assembler and a linker – low-level developer tools.

In this work, we developed a prototype that successfully generates a sophisticated assembler and linker for RISC-like architectures. Both components support complex features such as emitting and applying relocations. To facilitate this, we extended VADL with an assembly description element. Furthermore, our program automatically generates parser rules from pretty printers to reduce the specification burden of users. Lastly, we implemented two relocation generation strategies that allow passing symbols to some instructions without manually defined relocations. This contribution allows a more streamlined design exploration process by providing low-level developer tools tailored to the architecture.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**
1.1 Motivation and Problem Statement 1
1.2 Aim of this Work 2
1.3 Methodological Approach 2
1.4 Research Questions 4
1.5 Structure of this Work 5

2 Preliminaries **7**
2.1 Vienna Architecture Description Language 7
2.2 Low-Level Developer Tools 16
2.3 Parsing 21
2.4 Pretty Printers and their Inversions 24
2.5 LLVM 26
2.6 Testing Compilers 29

3 State of the Art **31**

4 Implementation **37**
4.1 Language Design 37
4.2 Grammar inference 45
4.3 Immediate Representation 50
4.4 Tool Generation 54
4.5 Testing 66

5 Evaluation **69**
5.1 Expressiveness 69
5.2 Tool Performance 71
5.3 Assembly Language Study 73

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.4 Grammar Rule Generation 78
5.5 Flexibility 79

6 Future Work **83**

7 Conclusions **85**

A Appendix **87**
A.1 VADL Intermediate Representation - Further Information 87
A.2 Parser Semantics Data Structures 88
A.3 Evaluation Data 89
A.4 Study Grammars 93
A.5 Test Programs for Evaluation 98

List of Figures **99**

List of Tables **101**

Listings **103**

Acronyms **105**

Bibliography **107**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

Low-level developer tools, such as assemblers and linkers, build the foundation of a modern compiler toolchain. Therefore, these executables must be correct and provide sufficient capabilities. Otherwise, the correctness and completeness of all depending tools (e.g., high-level language compilers) are impeded. The Vienna Architecture Description Language (VADL) is a Processor Description Language (PDL) that allows the generation of developer tools, hardware, and simulators from a single specification. This versatility allows for the re-use of concepts in different parts of the system. For example, the description of the assembler syntax may also be helpful in the simulator to print the executed commands. Most PDLs only focus on one or two of the artifacts that can be generated from VADL.

An Application Specific Instruction Set Processor (ASIP) is a processor tailored to a specific problem domain by including instructions that may not make sense in general-purpose processors. As a result, engineers can design more efficient algorithms in this domain by leveraging these instructions. Contrary to an Application Specific Circuit (ASIC), the nature of an ASIP still allows programmability, thus improving the end product's flexibility. The downside of both techniques is the high initial engineering cost. The goal of VADL is to decrease this expenditure by providing hardware designers with a high-level specification language that is powerful and intuitive. VADL specifications capture the information necessary for generating a versatile tool suite. Engineers can test the performance of new processor designs on the Instruction Set Simulator (ISS) and the Cycle-Accurate Simulator (CAS) before actual hardware needs to be manufactured. Furthermore, a compiler toolchain can be generated for every specification, reducing the initial costs of developing software for the newly designed ASIP. Finally, hardware designers can use the emitted schematics to speed up the development process of the final product.

Prior to this work, the generated compiler toolchain lacked support for many essential developer tools. These tools include an assembler and a linker. Fortunately, engineers could still get assembly code from the previous code generator version. However, they relied on a manually adjusted and compiled tool suite to generate executable files from the emitted assembly code. The lack of these tools mitigated much of the engineering cost reduction that the generated compiler toolchain promised.

1.2 Aim of this Work

This thesis aims at implementing a powerful generator for low-level developer tools based on VADL. Furthermore, we aim to make specifying the assembler syntax and semantics, the central part of this work, as simple as possible. The aspiration is that an engineer that sees the specification for the first time can understand it. Additionally, the language shall be flexible enough to express a variety of assembly languages. All of that shall be achieved without putting too much specification burden on the engineers. Lastly, we want to gain confidence in the correctness of our tools and thus make them usable in the industry. Given this thesis's results, generators that build upon VADL shall generate correct low-level developer tools for several processors that perform similarly to handcrafted alternatives. We expect our implementation to be flexible enough to support common assembly languages.

1.3 Methodological Approach

This section describes the methodological approach. We carry out this work in five different phases. While some phases may overlap, each has a distinguished goal that we strive to achieve.

1.3.1 Onboarding

VADL is a large project with thousands of commits. We need to familiarize ourselves with the characteristics of the system. We do this by implementing small tasks that scratch all components this thesis needs to improve upon. The first task is introducing a new expression type and then migrating the old assembly printing implementation to general-purpose functions. After that, we implement a basic parser. While doing these tasks, we will gain a good understanding of the parts that need adjustment. Furthermore, we can identify the most challenging problems that require the most attention in this work. Using this knowledge as a foundation, we can create a methodology and formulate research questions. All of this has already been completed by us. The research questions are presented in Section 1.4.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

1.3.2 Developing a Prototype

We develop a high-fidelity prototype for generating low-level developer tools. We intertwine this process with answering the research questions. Once we want to develop a feature that relates to a research question, we will take a break and answer said question to the necessary extent. After acquiring the necessary knowledge, we will develop a proof of concept for the feature and review its applicability in practice. This approach allows us to gain the necessary background information before developing a feature while still meeting the research project's milestones. Furthermore, we can react better to changes in requirements, similarly to agile development [BBVB+01]. After this phase, a working generator for the *RV32I* processor is available.

1.3.3 Study on assembly languages

After we have a working prototype with an $LL(1)$-parser, we want to examine other languages to determine if such a system is sufficient for our purpose. We choose the following assembly languages as targets of this study: *AArch64*, *MIPS*, *RISC-V*, *Hexagon* (very long instruction words (VLIW)), and *Intel x86* (CISC). The study results will guide our decision if the prototype meets our expectations, needs extensions, or if an out-of-the-box approach (e.g., ANTLR[PQ95]) is favorable.

1.3.4 Evaluation

We start the evaluation by implementing support for a VLIW architecture. This task shall give us an idea of how flexible our solution is when a new concept is introduced to VADL. Furthermore, we analyze how much specification work is required to create a functioning assembler and linker. We will evaluate the automated grammar rule generation using a set of formatting functions for the processors used in step 3. Lastly, we will benchmark the executables and compare the performance of our RISC-V assembler with the open-source RISC-V assembler that ships with LLVM. If we chose to use a different parser generator in step 3, we might compare our initially generated parser with the more sophisticated approach.

1.3.5 Correctness

We will implement a comprehensive testing system for the implemented generator and the resulting binaries. These tests shall execute automatically. How these tests will be facilitated depends on the findings of the research questions. Note that the initial prototype implementation and the tests created during the evaluation cover a large part of this phase.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

1.4 Research Questions

This section lists the research questions we derived during this thesis's onboarding phase. Most contributions of this work are associated with at least one of these questions.

RQ 1: *What language constructs can be used to concisely specify assembler-related properties while adhering to the design philosophy of VADL?*

Because VADL has numerous specification elements, each part shall adhere to a common design philosophy so that the language is consistent across different components. A syntax for describing an assembler has to be designed within that philosophy. It shall be intuitive and expressive. Common problems in constructing an assembler shall be easily expressible without losing generality. Such problems include, but are not limited to, instruction syntax, modifiers, number representations, and multiple assembly syntaxes. All of this shall be expressible with a concise specification. If an assembler syntax has special needs, the specification required shall deteriorate gracefully. We evaluate the expressibility of the proposed syntax by specifying the assembler for a set of processors.

RQ 2: *Given a forward function that formats an instruction to a string, to what extent can we automatically infer grammar rules describing the inverse transformation to a parser generator?*

A user should only need to specify a property once. The user has already specified the instruction syntax by writing a function that formats an instruction to a string. It shall not be necessary to define this format again for the assembler parser. Instead, the system shall be able to infer parsing rules from the formatting functions. The posed problem is a modified version of the automatic program inversion problem. Instead of directly emitting an inverse to a function, our algorithm shall be able to build a grammar rule that describes said inverse to a parser generator. We will evaluate our grammar rule inference by examining how many rules can be automatically inferred for a set of predefined processors.

RQ 3: *Assuming a parser is only used for assembly languages, how powerful does the parsing algorithm need to be? Especially to what extent is LL(1) sufficient to parse assembly languages?*

To the best of our knowledge, no work analyzes the grammar of modern assembly languages. Therefore, we will conduct a small study that evaluates how powerful our generated parser has to be. For now, a left-to-right parser with a look ahead of one token ($LL(1)$-Parser) will be sufficient for our purpose. Whether or not this is true will be evaluated on the processors described in Section 1.3.

RQ 4: *Allowing complex assembly concepts like very large instruction words, how much concept-specific code needs to be written in the generator?*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Designers of processor architectures should not be limited by their software tools. Some assembly constructs are specific to certain processor features. Constructs like these are usually not encountered while handling "standard" assembly code. While we try to design our specification language flexibly, some concepts may need dedicated support. One such feature is VLIW. By examining this feature's impact on the generators' code, we want to estimate how many changes are required to support a new construct.

RQ 5: *How can we establish confidence in the correctness and completeness of the generated tools?*

Correctness is paramount for a compiler toolchain. A bug in the assembler or linker could cause a malfunction in a production system. We want to find a way that gives us high confidence in the correctness of our solution. By analyzing how many and what kind of bugs our approach detects, we can get a sense of how effective the solution is.

1.5 Structure of this Work

Chapter 2 introduces the reader to the background necessary to understand the remaining thesis. Furthermore, it gives an overview of VADL and introduces the posed problems. Using this knowledge, Chapter 3 gives an overview of other PDLs and their approach to describing assembly language. After that, Chapter 4 discusses how the prototype is implemented in the LLVM Compiler Backend (LCB) generator. Then Chapter 5 discusses our findings and the performance of our implementation. Finally, Chapter 6 gives ideas for further improvements, while Chapter 7 summarizes this thesis.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 2

Preliminaries

This section will introduce the necessary background for understanding the remaining thesis. We assume exposure to programming languages and a high-level understanding of how a computer works. Moreover, a basic understanding of formal language theory may be beneficial, but not necessary.

2.1 Vienna Architecture Description Language

The Vienna Architecture Description Language is a mixed PDL (explained in Chapter 3). Based on a VADL specification, generators can emit a compiler toolchain, different types of simulators, and hardware schematics. The goal is to leverage overlaps between the different artifacts and eliminate redundant specification work – fewer specification results in less space for errors. This section starts by giving a coarse overview of VADL and the types of specification elements it contains. Furthermore, we will explain different data structures and summarise how our generators use them. Finally, specification elements essential for this work will be discussed in detail.

Keep in mind that we present the current snapshot of VADL. Even though we will cover numerous language elements, this is not an exhaustive list. Also, remember that VADL is still under active development. Therefore, the syntax and semantics of the presented code snippets may change over time. We even had to rewrite this section because some parts of the specification changed during this writing process.

The definition of a processor consists of multiple layers that build upon each other. The Instruction Set Architecture (ISA) builds the foundation of a processor description. Its purpose is to define the set of available instructions, the main memory, and the set of available registers. Once specified, the Application Binary Interface (ABI) defines calling conventions, gives alignment information and defines register aliases. Furthermore, users can specify the structure of system calls. A microprocessor implements an ISA with a

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

particular ABI. This element defines the startup behavior that allows the generation of simulators. Furthermore, a micro-architecture element can be defined to support the generation of hardware schematics. Finally, an additional assembly description element can be associated with an ABI to customize the assembler and linker of the generated compiler toolchain.

Our generators process a VADL specification in multiple stages. They transform the source code into a Concrete Syntax Tree (CST) with the help of Xtext[1], a framework for developing domain-specific languages. Next, the system transforms the CST into an Abstract Syntax Tree (AST). In this phase, referenced VADL modules are loaded and merged into a single specification. Furthermore, symbol resolution takes place, and a powerful type inference system annotates all nodes with their corresponding type. Therefore, it is a significant milestone to have a valid AST representation when introducing a new language construct. After the first round of validations, the AST is transformed into the VADL Intermediate Representation (VIR) - the central intermediate representation of the VADL generators. We refrain from explaining the data structures in detail because this will be part of another work. However, Section A.1 introduces all instructions explicitly mentioned in this thesis.

During the transformation from the AST to the VIR, all elements are associated with a microprocessor. Furthermore, we introduced further data structures to store artifact-specific information. The Generic Compiler Backend (GCB) is a compiler-specific representation essential for this work. For example, we store further information about immediates, instruction operands, and registers. However, there is a push to move more and more of the GCB back into the VIR because some of the analysis still proved beneficial for other components. Orthogonal to the compiler backend, the Register Transfer Level (RTL) enables the generation of hardware schematics. The simulators do not have a specialized intermediate representation (IR). They derive an in-memory representation of C++ routines directly from the VIR to model the simulators' behavior. Note that other backends also use this infrastructure to generate C++ code.

Because most analyses happen on the VIR, we need a powerful representation to model behavior. On the one hand, we have *functions* that model pure computations consisting of a single expression. On the other hand, *processes* model sequential computations with access to the state of the processor. One of the essential usages of functions is to model immediates, while one crucial task of processes is to define instruction semantics. Note that both of these constructs are loop-free and support no recursion. Therefore, VADL is not Turing-complete. This design decision is intended to make some analysis possible or more straightforward. The code within a function and process is represented using single static assignment (SSA) form [RWZ88].

[1] https://www.eclipse.org/Xtext/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2.1.1 Instruction Set Architecture

An ISA is an abstraction of a machine that can execute a sequence of instructions. Such a machine is called an implementation for the defined architecture. In reality, the complexity and nature of different implementations vary greatly. For example, simple instruction set simulators and highly optimized silicon may operate over the same instruction sequence. The standardized ISA makes this possible. This section will present the essential definitions that make up an ISA in VADL. Note that we will not list constructs we consider irrelevant for this thesis. Listing 2.1 shows the primary definition of an ISA. All other elements discussed in this section are children of this element.

```
1 instruction set architecture ISA = {
2     // ...
3 }
```

Listing 2.1: An Instruction Set Architecture

Defining Resources

An essential part of a computer is the main memory. The address width is one of the most crictical aspects of this component. Listing 2.2 defines a 4 GiB memory. In processes, `MEM` can be used as a function that takes a 32-bit address as input and returns the corresponding byte from the memory location. Access to this memory element is type-checked to catch erroneous specifications during design time. The `using`-declaration seen in this example is a type alias.

```
1 using Address = Bits<32>
2 memory MEM : Address -> Byte
```

Listing 2.2: Definition of Memory

Similarly to a memory definition, a register file is modeled as a function that takes an index as input and returns the value of a register. The input parameter type determines the number of addressable registers within the file. Listing 2.3 defines a register file X with 16 addressable 32-bit registers. Furthermore, this listing also declares the 32-bit register `R`. Annotations can restrict the value range of a register to model zero registers as seen in, for example, the RISC-V ISA [WLP+14].

```
1 using IntegerRegister = Bits< 32 >
2 register R : IntegerRegister
3 register file X : Bits< 4 > -> IntegerRegister
```

Listing 2.3: Definition of Registers and Register Files

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The program counter (PC) is a special-purpose register that keeps track of the program sequence. Listing 2.4 shows the definition of a standard PC. VADL supports three different kinds of PCs. During instruction execution, each kind differs in the register's value. The value of a "current" PC (the default) points to the start of the current instruction word. In the same way, a "next" PC points to the start of the next instruction word, i.e., the end of the current instruction word. The 32-bit ARM specification uses a special PC kind. A "next next" PC will always point to the end of the following instruction word. In this case, the following instruction word must be of the same size as the currently executed instruction. Most architectures will only use "current" and "next". An annotation on the definition of the PC may indicate its kind.

```
1 program counter PC : Bits<32>
```

Listing 2.4: Definition of a Program Counter

Defining Instruction Formats

Most architectures feature different types of instructions (e.g., arithmetic, load). Often, each type has a different layout. VADL allows a user to define several *instruction formats*. Listing 2.5 shows the definition of the `JType` instruction format of the RISC-V processor family. It consists of a list of *format fields* that have associated ranges, effectively dividing the instruction word[2] into multiple values. VADL computes the size of each format field based on the size of ranges. In this case, the `opcode` field has a type of `Bits< 7 >`. It is possible to store a field non-contiguously. The `offset` field is an example of that.

```
1 format JType : Bits< 32 >=
2 { offset [31, 19..12, 20, 30..21]
3 , rd     [11..7]
4 , opcode [6..0]
5 }
```

Listing 2.5: Definition of an Instruction Format

In VADL, custom immediates capsule the difference between the physical value of a format field and the semantic value used in an instruction's behavior. This concept is best explained with an example. The `JAL` instruction of the RISC-V ISA has a format field `offset` that consists of 20-bits. The semantic value corresponding to this bit vector is a 32-bit wide unsigned integer. We can construct the semantic value by sign-extending the format field to 31 bits and shifting this value by one. Listing 2.6 shows how this can be modeled in VADL. We extend the instruction type from the last example with the custom immediate `ImmediateJ`. Currently, an immediate definition consists of three

[2] In this thesis, instruction word refers to the bit string encoding a single instruction.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

components – an encoder, a decoder, and a predicate. While the encoder transforms the semantic value into the format field, the decoder acts as a natural inverse. The predicate checks whether a given value is a valid instance for this immediate. The code generator needs this functionality during instruction selection and machine-dependent optimizations. In this example, the compiler checks whether a given offset is small enough to use a relative jump[3]. The compiler must resort to an absolute jump if the predicate does not hold. In addition to the code generator, the assembly parser uses the predicate to check the validity of assembly operands. In the future, the VADL team wants to synthesize the encoder and predicate automatically from the decoder definition. Furthermore, the decoder function may be inferred from the instruction semantics, thus enabling users to write specifications without explicitly defining immediates.

```
format Jtype : BaseInstructionFormat =
{ offset [31, 19..12, 20, 30..21]
, rd     [11..7]
, opcode [6..0]
, ImmediateJ = ( offset as SInt<31>, 0b0 )
: predicate
    { ImmediateJ =>
        if ImmediateJ(0) = 0b0 then
            match ImmediateJ(31..20) with
            { 0x000 => true
            , 0xfff => true
            , _     => false
            }
        else
            false
    }
: encode
    { offset => ImmediateJ(20..1)
    }
}
```

Listing 2.6: JType with Immediates

The difference between these two representations leads to two different "views" on immediates. In VADL, a user should only think in format fields. Therefore, returning to the last example, `offset` always refers to the 20-bit format field value. The semantic value can be accessed by calling the immediate directly. However, the internal data structures in the LCB exclusively use the semantic value. This circumstance makes it easier and less error-prone to model the behavior of instructions. While this leads to a clean specification (no mixing of immediate views) and better implementation, it creates a mismatch at the boundaries of the LCB. Section 4.4.1 discusses this problem in detail.

[3]Note that this particular check only works if the offset is a known value. When using symbols, the linker is responsible for this optimization.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Defining Instructions

Instruction definitions are at the heart of the ISA as they capture a processor's capabilities. VADL distinguishes between three different types of instructions. A *machine instruction* is an actual operation that the processor can execute. Therefore, these instructions need to be encoded and decoded. A *pseudo instruction* only exists in the assembly code and thus cannot be encoded in an object file. Nevertheless, an ISA standardizes these instructions. Lastly, compiler instructions are internal to the code generator; in other words, they cannot be used in assembly code. As a result, these definitions do not have an assembly printing function. Otherwise, this instruction type behaves similarly to a pseudo instruction.

Listing 2.7 defines the `ADDI` instruction of the RISC-V ISA. This instruction is based on the `Itype` format. Therefore, all format fields and immediates of this definition are available in the instruction semantics. The behavior of the `ADDI` instructions adds a constant to a register and saves the result in a potentially different register. This particular definition makes use of a destination register index (`rd`), a source register index (`rs1`), and a custom immediate (`ImmediateI`). The code within the braces describes the semantics of the instruction, which is modeled by a process. Furthermore, the expression `X( i )` indexes the register file `X` with `i`.

```
1 instruction ADDI : Itype =
2 {
3     X( rd ) := X( rs1 ) + ImmediateI
4 }
```

Listing 2.7: Definition of a Machine Instruction

Pseudo and compiler instructions are similar in some regards. Because a processor cannot execute them, they do not have a process that describes the instruction semantics. However, the designers need to specify an *expansion* for these instructions. Listing 2.8 shows the definition of a pseudo and a compiler instruction. The body of the instructions specifies the expansion to actual machine instructions. Chapter 4 discusses when the LCB expands pseudo and compiler instruction. The annotations define these instructions as "sequences". This information instructs the compiler to use this instruction for special purposes, for example, loading a constant. Usually, compiler instructions only make sense in conjunction with a sequence annotation.

```
1 [ return sequence ]
2 pseudo instruction RET =
3 {
4     JALR{ rs1 = 1, rd = 0, imm = 0 }
5 }
6
7 [ constant sequence ]
8 compiler instruction LOAD32S( rd : Index, imm : SInt<32> ) =
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

```
9  {
10     LUI { rd=rd , imm20=hi20( imm )          }
11     ADDI{ rd=rd , rs1=rd , imm=lo12( imm ) }
12 }
```

Listing 2.8: Definition of Pseudo and Compiler Instructions

As mentioned, machine instructions must be encoded to create an object file. Some format fields of the instruction format may be constant, while others depend on the operands of a particular instruction. Listing 2.9 depicts the encoding for the `ADDI` instruction. Constants can be assigned to the fields defined in the instruction format. In this example, the opcode field has the value 19. Unbound format fields build the set of parameters.

```
1 encoding ADDI =
2 { opcode = 0b001'0011
3 , funct3 = 0b000
4 }
```

Listing 2.9: Definition of an Instruction Encoding

To define an instruction's assembly representation, a user has to specify an assembly printing function. The purpose of this function is to format an instruction word into a human-readable form. It has access to all non-constant format fields of the instruction. Furthermore, it can contain calls to auxiliary functions. The `AssemblyInliningPass` inlines these function calls later in the generator's pipeline. This step is necessary for the grammar rule inference described in Section 4.2. Listing 2.10 displays the printing function of the `ADDI` instruction of the RISC-V ISA. Note that the tuple syntax represents an n-ary concatenation operator. The `mnemonic` keyword is transformed into a string constant referring to the mnemonic of the current instruction during the AST to VIR translation. This construct helps to define proper printing functions in macros. The built-in functions `register` and `decimal` are used to print registers and immediate operands. Section 4.1.1 discusses these constructs in detail. To the best of our knowledge, PDLs never expressed assembly syntax in the form of general-purpose formatting functions. Even though this fits well into the overall philosophy of VADL, it introduces more complexity than a simple string pattern.

```
1 assembly ADDI = (mnemonic, " ", register( rd ), ",", register( rs1 ),
2     ",", decimal( imm ))
```

Listing 2.10: Assembly Printing Function

Lastly, we would like to mention other definitions that are used in the VLIW context. In VADL, designers can build subsets of instructions using the *operations* construct.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Furthermore, a *group* definition defines a bundle – a set of instruction instances with explicit instruction-level parallelism (ILP). These groups are annotated with constraints that may make use of the aforementioned operations. By analyzing these restrictions, a system can decide whether a given bundle is valid. Furthermore, the compiler uses the same information to create valid bundles in the first place.

2.1.2 Application Binary Interface

The ABI describes the interface between two binary programs. For example, a user program that wants to call into a library needs to adhere to certain restrictions that the interface imposes. Users must specify these restrictions so the compiler toolchain can produce correctly working binaries. Listing 2.11 shows the primary definition of an ABI. Note that this element always depends upon an ISA. All other elements described in this section are children of this element.

```
1 application binary interface ABI for ISA =
2 {
3     // ...
4 }
```

Listing 2.11: Application Binary Interface Definition

The alias register construct associates a name with a register or an index in a register file. Listing 2.12 shows an example of assigning the name `zero` to the first register in register file `X`. Note that the right-hand side of this assignment can also be another alias register.

```
1 alias register zero = X(0)
```

Listing 2.12: Alias Registers

Listing 2.13 shows the definition of a calling convention. In the first two lines, the set of caller and callee saved registers are defined. These sets describe who is responsible for preserving which register values over a subroutine invocation. Furthermore, we define a set of registers that allow efficient argument passing. If a routine needs more than seven argument registers, the caller will push the remaining values on the stack. Lastly, we define the register that contains the return address and the set of return value registers.

```
1 caller saved = [ ra, a{0..7}, t{0..6} ]
2 callee saved = [ sp, fp, s{0..11} ]
3
4 function argument = a{0..7}
5
6 return address = ra
7 return value = a{0..1}
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Listing 2.13: Calling Conventions

The ABI also specifies important system registers. To illustrate this, Listing 2.14 contains the necessary specification to define the frame, stack, and global pointer. The following list explains these registers.

- The frame pointer always refers to the base of the current stack frame in the call stack. Often, this register is just a convenience for programmers as they can easily refer to arguments and local variables with fixed offsets. However, when supporting dynamic stack allocations, this register is essential.
- The stack pointer points to the top of the stack. The value of this register may change during the execution of a stack manipulation operation, such as push and pop. The alignment annotation instructs the compiler to align the stack pointer to a 16-byte boundary on procedure entry.
- Finally, the global pointer can be used to relax access to global variables. For example, assuming the RISC-V Unix ABI [WLP+14, Chapter 18], loading a 32-bit address requires two instructions. However, using the global pointer allows us to load addresses in the vicinity of the global pointer with a single instruction, for example, by adding an offset to the global pointer.

```
1 frame pointer = fp
2
3 [ alignment : Bits< 128 > ]
4 stack pointer = sp
5
6 global pointer = gp
```

Listing 2.14: System Registers

2.1.3 Micro Processor

The definitions within the microprocessor are currently not relevant to this thesis. Therefore, we will not cover the supported definitions in detail. The microprocessor definition mainly contains information relevant to the simulators. For example, it contains processes for exception handling, startup code, and processor firmware. Listing 2.15 shows the primary definition of a RISC-V processor. This element connects all other definitions, thus allowing the generation of an LLVM backend.

```
1 [ target = "rv32i" ]
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
2 [ description = "32-bit RISC-V Integer" ]
3 micro processor CPU implements RV32I with ILP32 =
4 {
5     // ...
6 }
```

Listing 2.15: Micro Processor Definition

2.1.4 Micro Architecture

To allow the generation of hardware schematics from a specification, VADL supports defining a micro-architecture for a given microprocessor. Generally speaking, a user can describe the pipeline structure in this section. While this is yet to be relevant for the LCB, a generator may deduce compiler-relevant information from the architecture's structure. For example, we could infer the instruction delay used for instruction scheduling from the pipeline description. Listing 2.16 defines a micro-architecture for the CPU microprocessor.

```
1 micro architecture CPU_5_stage implements CPU =
2 {
3     // ...
4 }
```

Listing 2.16: Micro Architecture Definition

2.2 Low-Level Developer Tools

In this work, "Low-Level Developer Tools" refers to software programs at the end of a compiler toolchain. We use the canonical compiler process depicted in Figure 2.1 to clarify this. Some terms in this paragraph will be introduced later. The goal of this paragraph is to provide an overview. We cover the details in the following sections. The "Source Files" in the figure represent a program written in a high-level language (e.g., C). The code generator ingests the input program and produces assembly files. While the assembly printer is part of this work, we do not consider the code generator a low-level developer tool. The assembler reads the assembly files and emits a relocatable object file per input file. Lastly, the linker combines the relocatable object files into a single executable object file. The linker script describes the structure of the output file to the program. For example, a developer can choose at what address parts of the program should be loaded. Note that often multiple steps in this process are combined to limit overhead. For example, more complex compilers tend to skip emitting assembly by directly generating object files.

Historically, the output of an assembler or compiler is called object code. *Object files* bundle this code together. Multiple formats for storing such bundles exist. However, this

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

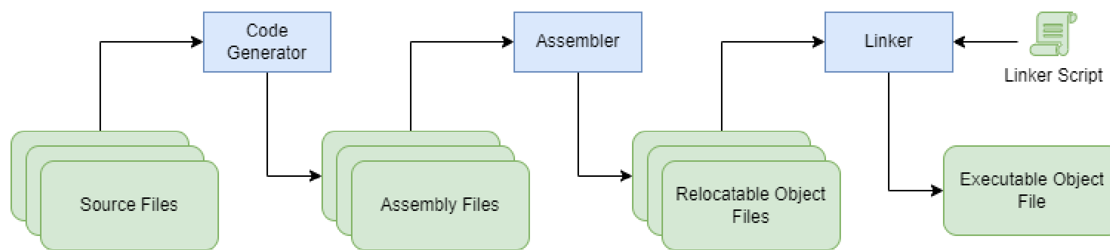

Figure 2.1: Canonical Compiler Process

work will focus on the Executable and Linkable Format (ELF). We will discuss three different flavors of object files. First, *relocatable* object files are usually the product of an assembler or compiler. They contain the machine code of an assembly unit but are not executable. Partly, this is because symbols (e.g., a routine name) within relocatable objects files are unresolved; in other words, format fields containing symbols are zero[4]. Second, *executable* object files are usually created by merging (i.e., linking) multiple relocatable files into a single binary. During this step, the linker resolves referenced symbols. Section 2.2.3 discusses this process in detail. Lastly, the third flavor is *shared libraries*. This work will primarily deal with the former two types.

2.2.1 Executable and Linking Format

This section discusses the basics of the ELF and is primarily based on the book "Linkers and Loaders"[Lev99] and the System V ABI specification [MHJM13]. The ELF originated in the initial System V ABI as a successor to the prevailing "a.out" format. Its purpose was to improve the support for modern system features such as dynamic linking. The format's popularity started gaining traction when Unix systems picked it up. Furthermore, the 86open project[5] chose ELF as the default binary format on Unix-like operating systems. Nowadays, ELF is one of the most used object file formats in computing.

An ELF file starts with a header that provides general information. We will not enumerate every header field, just the relevant ones in this work's context. The `class` header field specifies the size of addresses for a given ELF file. For example, is this a 32-bit or 64-bit object file? The address mode also influences other properties, like how many different relocation types an architecture can support. The `data` field indicates the endianness of the file and the `archtype` field specifies for which architecture this file is intended. Lastly, the `filetype` field encodes the type of the ELF file (e.g., relocatable or executable).

An ELF file has two opposing viewpoints. First, from the assembler's point of view, an ELF file consists of sections. A section consists of a name, additional headers, and

[4]Actually, in ELF, this depends on the used relocation entry type. REL entries may use this format field to store addends to the resolved symbol, while RELA entries store the addend in the relocations entry. Section 2.2.1 discusses this topic further.

[5]https://web.archive.org/web/20070311032337/http://www.telly.org/86open-faq

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

data. This construct is flexible enough to represent program code, data, and auxiliary information. The type of a section is stored in a header field and gives information about the section's content. For example, a `PROGBITS` section may contain machine code, while a `REL` section contains relocation information essential to the linker. The system reserves itself the usage of all sections starting with a dot. For example, the `.text` section usually contains the instruction sequence of the program. The System V ABI lists all special-purpose sections. Contrary to the assembler, the loader sees the ELF file as a set of segments that map into specific memory locations. Segments also have an associated header that, for example, indicates whether this segment is modifiable or not. For example, one segment may contain all object code, while another holds all the data. These segments may consist of multiple sections. The section and program header tables store all of this information. Note that the separation of viewpoints is not black and white. For example, the linker must know both viewpoints as it is responsible for mapping sections to segments.

Relocations are the primary communication channel between the assembler and the linker. This communication is necessary because some information is unknown during assembly and cannot be embedded directly in the machine code. For example, the assembler cannot know the address of a symbol. This circumstance is trivially true for external symbols as these are outside of the assembler's scope. However, less intuitively, symbols within the same assembly file cannot be addressed in most circumstances. On the one hand, absolute addressing does not work because the linker may move the symbol to a different location. On the other hand, while relative addressing may work in simple scenarios, necessary assumptions break when using, for example, linker relaxations or alignment directives. See Section 2.2.3 for further discussions. In both scenarios, the linker needs to know which instruction refers to which symbol. We use relocations to materialize this information. Now, if the assembler encounters a symbol usage, it emits a placeholder value in the unknown format field. Additionally, it allocates an entry in a relocation table with two crucial properties: the type of the relocation and the target symbol. During link-time, once the final layout of the file is determined, the linker resolves the addresses of the symbols and applies the necessary transformations.

ELF defines two formats for storing relocation data – `REL` and `RELA` entries. The difference is the handling of addends, in other words, the number that the linker adds to the resolved symbols' address. The former format stores the addends implicitly in the instruction word, while the latter stores the addends explicitly in the relocation entries themselves. As a result, `RELA` relocations consume more disk space but are easier to handle for the linker. Note that not all relocations are resolved in the same way. For example, sometimes, they are resolved as an absolute address. Other times they are resolved relative to the PC.

2.2.2 Assembler

Assembly languages are low-level programming languages that target specific computer architectures. Usually, the instructions in such languages mirror the hardware's instruc-

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

tions. However, as this file is not actual machine code, a program is required to create an object file from an assembly program – the *assembler*. We call this process “assembly”. On top of providing a text representation of machine code, most assembly languages support additional functionality. The following list contains commonly found features in assemblers.

- *Labels* mark specific points in an assembly file so other instructions can refer to them. This technique is preferred over simply using a hardcoded address due to the improved readability and the support for relocations. Jump instructions make heavy use of labels in real-world code.
- *Directives* are commands to the assembler. In many assembly languages, directives start with a dot. Some common commands include aligning a section, fixing the size of a constant in memory (e.g., 4 bytes), defining external symbols, and many more.
- *Expressions* allow passing complex calculations as an operand to an instruction. This technique is advantageous because the exact value of an immediate is often unknown until link time (e.g., $symb + 4$). To solve this issue, the assembler must provide a possibility to defer the evaluation of an expression. This issue can be addressed by accepting expressions with symbols as operands and emitting necessary information (relocations) in the resulting relocatable object file. Because the linker will already evaluate the expression, no superfluous instruction is executed at run-time to calculate the result of the expression.
- A computer architecture may have *multiple assembly languages*. One such example is the Intel and AT&T syntax for the x86 ISA. Therefore, many assemblers allow switching between these languages with a command line argument.
- *Pseudo instructions* may expand to multiple machine instructions. Sometimes, this instruction sequence depends on the inputs of the pseudo instruction. For example, a small immediate may be loaded with a single instruction, while a large immediate requires two instructions. The assembler is responsible for choosing the best instruction sequence.

2.2.3 Linker

The assembly process usually generates relocatable object files. A *linker* is a program that can combine multiple relocatable object files into a single executable object file. We call this process *linking*. This section is primarily based on the book “Linkers and Loaders” [Lev99].

There are two primary linking techniques – static linking and dynamic linking. A statically linked object file contains all referenced object code in a single file. Therefore, a device that implements the proper ISA and ABI can load and run this file independently. On

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

the other hand, a dynamically linked program can defer the linkage of symbols to the program load time. Such executables contain a list of dynamically linkable symbols in which each symbol is associated with a library. The loader can then read the referenced libraries and resolve the symbols while the program is already in memory. This technique blurs the line between a linker and a loader by moving linking tasks into the loading process. Because this work will solely deal with static linking, the term "linking" will refer to static linking in the remaining work.

One of the most important jobs of a linker is resolving symbols and applying relocations. Section 2.2.1 already discussed the background of this topic. In this work, applying the relocations and patching the corresponding instruction words will be our primary concern regarding linking. Note that modern linkers have an abundance of complex features that we do not list here because they are irrelevant to this work. Some may be out-of-scope, while the target-agnostic part of LLVM does the heavy lifting of other features the LCB supports.

Another target-specific component is *linker relaxation* or *link-time optimization (LTO)*. The goal is to leverage the additional information a linker has of other modules and the object files' final structure. Because this information is not available at compile-time, such techniques cannot be applied earlier. The following paragraph explains this concept with an example.

Let us assume the architecture at hand has two unconditional jumps, an absolute and a PC-relative one. Furthermore, in this architecture, executing an absolute jump requires two cycles, while a relative jump requires a single cycle. Unfortunately, our relative jump only supports small offsets (e.g., 1 MiB) to the PC, while the absolute jump can address the whole virtual memory. When a compiler for this ISA encounters a jump to an unknown routine, it will emit an absolute jump because it does not know how many bytes separate the jump instruction's address from the target address. However, the linker knows how far apart these two addresses are. Therefore, an LTO can replace the absolute jump with a relative jump, thus improving the program's performance. A similar optimization exists for RISC-V processors. The absolute jump usually is slower than a relative one because materializing arbitrary 32-bit addresses requires two instructions on the 32-bit RISC-V architecture.

Unfortunately, applying linker relaxations is not straightforward. Firstly, applying an LTO interacts with other relaxation candidates because the code size tends to shrink when linker relaxations are applied. For example, applying the relaxation from the last example frees 4 bytes per transformation, assuming a 32-bit instruction word. Such a code size reduction could then enable another jump LTO that was too far from the target address prior to this reduction. Other techniques, such as dead code elimination and only preserving selected caller-saved registers, promise further space savings [DBDSVP+04]. Thus, doing linker relaxations in an iterative process may prove beneficial. All of that has to be done without violating other program invariants, such as alignment directives.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2.2.4 Disassembler

The disassembler is the inverse of an assembler. It receives a series of bytes as input and outputs the assembly representing the machine code. Note that the disassembler cannot reproduce all constructs that an assembler supports. For example, the information on label positions in the original program may be lost in executable object files to save space. Its primary purpose is analyzing existing object files, reverse engineering a program, and supporting other debugging techniques.

2.3 Parsing

Parsing is the process of analyzing a text string and checking its validity against a set of syntax rules. Furthermore, the process derives a concrete syntax tree - a data structure representing the input hierarchically. We also refer to these structures as "parse trees". Depending on the complexity of the task, tools that ingest text-based specifications may use the CST as IR.

Generally, parsing is a well-studied and well-understood problem. As a result, many different approaches have proven to work well in practical settings. This section will present an overview of the design space of a text parsing system. Usually, the first decision is to write the parser by hand or use a parser generator that derives the parser from a formal specification. However, in this work, user input influences the language's syntax. Thus, we cannot write a parser upfront and must resort to a generated parser approach. The remaining Chapter is based on the book "Engineering a Compiler"[CT11, Chapter 3].

First, we will start by introducing a formal notion to describe parsing. A Context-Free Grammar (CFG) $L_G = (T, NT, S, P)$ is a quadruple that formally specifies the syntax of a language L. Firstly, T is the set of terminal symbols. Informally, they describe the output of the lexer – a token annotated with a token type. We also refer to a token as a word. Furthermore, we will use lowercase letters to indicate terminal rules. Secondly, NT is the set of non-terminal symbols used in productions to provide abstraction. Similar to terminal symbols, we will use uppercase letters to indicate non-terminal rules. Thirdly, S is the start symbol of the grammar. Lastly, P defines the set of allowed productions of the form $\alpha \to \beta$, where α is a single non-terminal symbol.

Using the definition of a grammar, we can define the parsing process more formally by using the notion of *derivation*. The derivation process always starts at a particular element – the start symbol. Each derivation step applies a single production to the string. We achieve this by replacing an occurrence of the left-hand side with the right-hand side of the production. The notion $\alpha \to \beta$ denotes that we can derive β from α in a single step. We use the symbol $\to^+$ to indicate that such a derivation can happen in one or more steps. Given a string $i \in T^*$ as input, the parser's goal is to check whether $S \to^+ i$. We can construct the parse tree from the applied production rules. The leaves of this tree will be the words of the input sentence.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

During the derivation process, we may encounter a string with two or more non-terminals. If that is the case, we must decide which non-terminal we would like to replace. There are two practically relevant strategies for choosing this non-terminal – left-most and right-most derivation. As their names indicate, we either expand the left-most or right-most non-terminal. Even if one of these strategies is applied, a grammar may permit multiple parse trees deriving the same sentence. We call such grammars *ambiguous*.

Non-deterministic pushdown automata can check whether a string is part of a context-free language [Cho56]. In other words, these automata can recognize these languages. Unfortunately, a parser implementing an arbitrary CFG may not have a linear computational complexity regarding the number of tokens. Fortunately, some sub-sets of CFGs are expressive enough to define general-purpose programming languages while still producing parsers with linear complexity. Because we do not need to parse a complex programming language, we will restrict ourselves to a sub-set with a straightforward implementation – $LL(1)$ grammars.

An $LL(1)$-parser is a **l**eft-to-right parser that uses **l**eft-most derivation. Because this family uses a top-down parsing approach, the algorithm builds the parse tree from the top by starting with the initial non-terminal symbol. Therefore, the initial parse tree has a single node. Now the parser's job is to expand this tree so that the fringe of the tree covers all input words. Such a tree is only possible if the string is a valid derivation of the start symbol. To reach such a state, the algorithm selects the left-most non-terminal symbol for expansion until the fringe of the tree has no non-terminal symbols. Once a non-terminal was chosen, the parser searches the set of productions for rules that can replace the selected symbol. At this point, the parser may run into the problem that it has multiple productions for the chosen non-terminal rule. One solution to this problem is to try all alternatives and reset the state if no derivation can be found. We call this technique *backtracking*. However, applying this technique may lead to longer parse times as the algorithm must keep track of all choices while trying out many productions. To speed up this process, we want a parser that always knows which production to apply – a *deterministic* parser.

The first step to solving this problem is to track what parts of the input string our algorithm already matched, in other words, which tokens are already leaves of the parse tree. As a result, the first word of the input string that the algorithm did not match yet, needs to be matched next. We refer to this word as a *lookahead* of one. To make use of this ability, the parser generator computes the `FIRST` set for each available production. This set contains the first words of all sentences derivable from the production's right-hand side. Intuitively, applying a production is futile if the lookahead token is not in the `FIRST` set of said production because the first word of the derivations can never match the current lookahead token. Note that the parser generator needs special handling for productions to the empty symbol ϵ. This element represents an empty sequence and does not match any word. Therefore, if this symbol is in the `FIRST` set of the right-hand side of a non-terminal, the generator must compute the `FOLLOW` set of the left-hand side of the non-terminal. We omit this description here because, in its current state, VADL does

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

not allow the use of ϵ.

Now we can create a deterministic parser by ensuring that for every rule application, there is at most one production with a `FIRST` set that contains the current lookahead. This approach can be extended by looking at k words instead of one word. We call this a $LL(k)$ parser. Furthermore, some approaches use arbitrary long lookaheads and can further execute semantic checks when choosing a production rule. We call such a system a predicated $LL(*)$ parser [PQ95]. However, in practice, $LL(1)$-parsers are still interesting because they have a straightforward implementation and language designers often ensure that their grammars are in $LL(1)$.

We have yet to discuss how to derive the input for the parser. Usually, the input consists of text characters and not tokens of T. To bridge this gap, a lexer (or scanner) can transform a string of characters into a string of tokens. In practical systems, all terminal symbols $t \in T$ are annotated with a regular expression. If a regular expression accepts a character string, it may be emitted together with the token type t. There are techniques to derive a lexer from the annotated terminal symbols [CT11, Chapter 2]. We do not discuss them here because the LCB uses the handcrafted LLVM lexer.

Lastly, we will discuss the relationships between the mentioned grammar types. In this paragraph, $G_1 \subset G_2$ denotes that grammars of type G_2 can model more languages than G_1. First, $LL(k)$ grammars are a real subset of CFG grammars. Thus $LL(k) \subset CFG$ holds. One source of this relation is that $LL(k)$ grammars cannot handle *left-recursion*. This problem arises when a grammar calls itself (indirectly) on the left-hand side of a production rule. Some algorithms [Moo00] try to address this problem. Furthermore, the relation $LL(k) \subset LL(k+1)$ [RS69] holds. However, while not true in general, many $LL(k)$ grammars can be modelled with an $LL(1)$ grammar. For example, the grammar in Listing 2.17 can be transformed to $LL(1)$ by hoisting the terminal x into a new helper non-terminal. This technique is called *left-factoring*. While the new grammar is $LL(1)$, writing such grammar is often tedious.

```
1  # LL(2)
2  A -> B | C ;
3  B -> x y ;
4  C -> x z ;
5
6  # LL(1)
7  A -> x T ;
8  T -> B | C ;
9  B -> y ;
10 C -> z ;
```

Listing 2.17: LL Grammars for a Simple Language

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2.4 Pretty Printers and their Inversions

A pretty printer transforms structured data into text while adhering to some notion of readability [HH95]. In the case of programming languages, this structured data is often referred to as AST. However, a single AST may have multiple representations in the concrete syntax (i.e., the source code). The goal of the pretty printer is to emit this data in a form that is pleasant to read. For example, most programs are easier to read with carefully set indentations than with no indents.

One of the first systems to explore this topic was Syn [Bou96]. The program aims to derive a language front-end from a Backus-Naur form (BNF) grammar. The generated output includes the code for the AST itself, a pretty printer, a lexer, and a parser. The grammar rule structure dictates the shape of the AST. Like other systems, Syn can derive the lexer and parser directly from the given grammar. Annotations embedded in the grammar instruct the pretty printer to format the grammar. For example, the rule `[<h 1> "a" "b"]` instructs the printer to insert one space between the two string literals. For the parser, this is equivalent to `"a" "b"`. Intuitively, the formatting annotation gets ignored. While this system can derive a parser and pretty printer, the grammar specification can become verbose by inserting the formatting rules.

Some systems similar to Syn fall into the BNF converter family. The first iteration [FR04] takes a labeled Backus-Naur form (LBNF) [FR03] grammar as input and can produce a parser and pretty printer for that language. Further work [DJ11] improved the existing solution by providing meta-programming capabilities. The generated program includes default implementations for a pretty printer. However, these default configurations cannot be customized.

Other efforts present an approach that does not rely on a specification that either fits pretty printing or parsing more naturally. Instead, they define the relationship between the AST and concrete syntax with a set of combinators that have two interpretations - one for each direction [RO10]. The only difference is applying the combinator's parser or pretty printing semantic. However, for function applications, this requires the usage of partial isomorphisms, in other words, invertible functions practicable for parsers. Furthermore, users may use syntax descriptions to model the parser's and printer's differences. For example, during parsing, we may want to accept arbitrary whitespace, while during printing, we want a single space.

The insight that "prettiness" is too complex to be inferred led to the implementation of FliPpr [MW13]. This system allows the developers to implement a pretty printer themselves. Then, the system derives a parser from this program using grammar-based program inversion [MMHT10]. Unfortunately, the parser only accepts the pretty printer's exact format without additional information. This behavior is undesirable as we often want to accept differently formatted input. To solve this problem, users must introduce "ugliness" into the pretty printer. To do so, the authors introduced the biased choice operator. This construct allows the definition of multiple alternative outputs. During printing, the program will always emit the first operand. However, this construct

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

gives the necessary information to the parser to accept other not-so-pretty alternatives. Additionally, FliPpr has special rules for handling whitespaces during the parsing process.

Alternatively, we can view this problem purely from the viewpoint of program inversion. Here we will consider the problem of injective string encoder inversion. Given a string encoder function γ, we want to derive a decoder function δ such that $\delta(y) = x$, iff $\gamma(x) = y$ for all x in the domain of f. In our case, x is the instruction word, while y is the assembly string. Note that this definition only works for injective functions, i.e., functions that do not map two inputs to the same output. This property often holds for pretty printers. However, all solutions to this problem suffer from the same problem already explained in the context of FliPpr. The inverted function γ will only accept the exact string that the pretty printer produces. Therefore, such a solution cannot handle semantically equivalent syntax. The solutions we present in the following paragraphs do not address this problem. However, with some adjustments, these approaches may lend themselves very well to generating a parser from a pretty printer.

We already mentioned one such example, grammar-based program inversion [MMHT10], the technique used to generate a parser in FliPpr. This approach splits the problem into two sub-problems. First, the system derives a grammar that approximates the evaluation tree of the original program. The second task is to find an efficient parsing method for the generated grammar. A valuable property of this approach is that the complexity of parsing the grammar gives an upper bound to the complexity of the inverse.

Another effort translates the problem of finding an inverse to a graph search [HVQ+12]. This process is split into multiple steps. First, the algorithm builds an equivalent SSA representation from the original program. Then, the system constructs a value search graph. This data structure explicitly models equivalency relations, thus enabling an effective approach to recovering values from the forward execution. This graph contains two types of nodes. Available nodes have a known value, while target nodes still need to be computed. Route graphs are sub-graphs of the value search graph that allow the computation of all target nodes. The algorithm selects a route graph based on the heuristic and then constructs the inverse program.

Qinheping et al. [HD17] describe a general-purpose solution inverting encoding functions operating over lists. Their solution can automatically infer the inverse functions of string encoders by leveraging symbolic transducers, i.e., automata with outputs. Moreover, their technique guarantees the inverse function's correctness by using syntax-guided program synthesis in conjunction with a state-of-the-art SMT solver (e.g., [MB08]). However, this approach presents a complex solution to the problem with significant dependencies.

Similarly to this solution, we could leverage program synthesis without symbolic transducers. Generally, a synthesis algorithm generates a program that fulfills a given specification. This specification could describe the inverse of the pretty printer to leverage this approach for our purpose. In theory, such an inverse can be generated by using a formula such as $\exists f^{-1} \in \mathcal{F}, \forall x \in \mathcal{D}, f^{-1}(f(x)) = x$. In other words, does a program (f^{-1}) exist in the set of all possible programs ($\mathcal{F}$) that inverts f for every value in the domain ($\mathcal{D}$) of f. Our

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

system then passes this specification to an SMT solver, with f being our pretty printer. However, in practice, this query will time out at some point due to the complexity of the request. Note that we also need to define the semantics of a program in a formal theory.

Counter Example Guided Inductive Synthesis (CEGIS) [SL08] tries to solve this efficiency problem. Implementing such an architecture requires two components, a component that generates a candidate (symbolic) program based on the set of learned counterexamples and a component that verifies if this program fulfills the whole specification. If the program does not adhere to the specification, a new counter-example is provided and added to the set of known input-output pairs. The premise is that the relatively small set of counter-examples covers the corner cases that the program has to implement. This approach tries to eliminate the all-quantifier in the synthesis step. A similar approach could work based on a formal grammar, not programs. Nevertheless, such a solution would significantly increase the complexity of our generator.

Another synthesis technique focuses on inferring a general solution from examples. The main issue is that the provided input is inherently ambiguous. For example, an infinite number of functions fulfill the requirement $f(0) = 0$. Recent efforts [ZLWG20] applied this approach to the problem of generating a regular grammar that accepts a given set of strings while rejecting another set. The authors could generalize the provided examples by using interactive user input to guide the synthesis process. In an approach tailored to our problem, programs like pretty-printers could generate an initial set of examples, which a domain expert refines in an interactive session.

2.5 LLVM

While initially designed as a research infrastructure for dynamic compilation techniques, LLVM [LA04] evolved into one of the most impactful compiler projects in industry and research. It is an umbrella for many sub-projects ranging from a C-Compiler to a tool that optimizes already-built executable object files. A list of sub-projects can be found on the official website[6]. This section will introduce the relevant parts of LLVM.

```
1 define i32 @adder(i32 %x, i32 %y) {
2 entry:
3     %tmp = add i32 %x, %y
4      ret i32 %tmp
5 }
```

Listing 2.18: LLVM IR Example

The most crucial sub-project, LLVM Core, is a retargetable compiler framework with a capable optimizer. This framework revolves around the LLVM-IR – an SSA-based [RWZ88] IR that abstracts away many, but not all, target-dependent properties of actual machine

[6] https://llvm.org/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

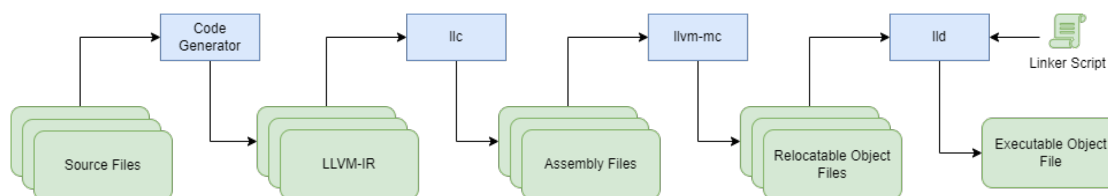

Figure 2.2: Canonical LLVM Compiler Process

code. This data structure has three different representations. First, the text-based representation is the only human-readable form of the IR as it is similar to a programming language. For example, Listing 2.18 defines a simple function that takes two input parameters and adds them together. A programmer may recognize many of the constructs used in this example without prior LLVM knowledge. The second form of the LLVM IR is a bitcode format that allows a denser encoding than text files. Users may switch between the two formats by using the LLVM assembler and disassembler. Lastly, the LLVM IR has a corresponding in-memory structure used by the optimizer.

Even though the textual representation of the IR looks a lot like a programming language, its intent differs from most. The goal of the LLVM-IR is to be a target for other compilers. A compiler for an LLVM-based language is responsible for transforming the source code to an equivalent LLVM-IR. From there on, LLVM will handle the rest of the compiler pipeline, including many optimizations and code-emission for multiple target processors. We refer to these supported architectures as *targets*. Such an architecture allows new languages to profit from a mature compiler infrastructure with comparatively low developer investment. This idea itself is not new. A similar approach, the Universal Computer Oriented Language (UNCOL) [Con58], was already proposed by Conway in 1958. However, LLVM is one of the first truly successful implementations of such an idea by supporting many industry-grade platforms, such as the LLVM C compiler (`clang`) and the reference Rust [MK14] compiler (`rustc`).

In the following paragraph, we will revisit the canonical compiler process from Section 2.2 for an LLVM-based compiler architecture. We start with code written in the source language. The compiler for that language (e.g., `clang`) is now responsible for emitting equivalent LLVM-IR. The LLVM Static Compiler (`llc`) can then ingest the code and generate assembly files for all processors that LLVM supports. Note that the `-S` flag has to be specified to output the assembly code. Otherwise, the compiler will emit an object file directly. The rest of the process is equivalent to the standard compilation process. However, we will still cover the remaining steps as we will discuss this in the context of the LLVM toolchain. The assembler in our toolchain is the LLVM Machine Code (`llvm-mc`) program. It is responsible for generating relocatable object files. Lastly, the LLVM Linker (`lld`) is the capable linker that will emit our executable program. Naturally, its purpose is to link the object files emitted by the assembler. Note that the LLVM Linker's command line interface is compatible to the GNU Linker.

Because LLVM supports many different architectures, developers tried to streamline

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

the process of adding and maintaining an architecture. One essential part is defining a standard interface for every architecture that other parts of LLVM can use. This software design allows defining target-dependent passes once and parameterizing them based on the information of the current target. However, defining all the necessary information in regular C++ files is cumbersome. Therefore, much of the target-dependent information the optimizer needs is defined in TableGen[7] files. TableGen is a declarative domain-specific language that allows the concise definition of records. An LLVM backend uses these files to define information, such as the set of available registers and the ISA instructions. From these definitions, a TableGen backend[8] emits several C++ files. Some files describe these records to the optimizer, while others implement large parts of the instruction encoding.

Internally, LLVM uses different data structures to represent the current code. This complexity arises because different aspects are either relevant or pragmatic in different circumstances. For example, inter-instruction dependencies are essential during instruction scheduling. However, during assembly, we mainly deal with a single instruction at a time. As a result, the two different tasks benefit from different data structures. The data structures used at the end of the toolchain are part of the `MachineCode` or `MC` layer. The remaining section introduces the most notable classes in this component. Note that all these classes or class hierarchies contain target-specific properties or logic.

An instance of the `MCInst` class represents a machine code instruction. Each instruction has an opcode that identifies the associated operation, for example, add two registers. Note that this opcode is LLVM internal and does not correspond to the opcode field commonly used in the instruction encodings. Furthermore, each instruction can have multiple `MCOperand` instances that represent the instruction's operands. An operand can contain an integer, a register, a floating-point value, a complex expression, or a sub-instruction.

The most versatile operand type contains an expression. To represent complex assembly expressions, we use the `MCExpr` class hierarchy. The subtypes of this hierarchy include a class for constant values, symbol values, and binary expressions. By composing objects of these classes together, we can express a variety of non-trivial terms. In addition to denoting a computation, each expression has the notion of being absolute and relocatable. If an expression is absolute, it will evaluate to a fixed constant value, independent of the context. For example, an expression with a single integer constant is absolute, but a symbol reference is not. This is because the object file layout, which is part of the context, influences the symbol's address and thus its value. Moreover, we consider an expression relocatable if we can rewrite it to another expression of the form $a - b + c$, where a and b are symbols and c is a constant.

In addition to the core data structures, the machine code layer consists of several components working with objects of the presented classes. Because the machine code

[7] https://llvm.org/docs/TableGen/ProgRef.html

[8] A TableGen backend is a generator that uses TableGen files as input.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

layer is not primarily relevant to optimizations[9], its primary use is emitting and ingesting different machine code representations. As a result, most components come in pairs – one part is responsible for reading and one for writing a particular file type.

The `MCInstPrinter` is the base class for target-specific instruction printer implementations. This class is responsible for emitting the assembly representation of a given `MCInst` to an output stream. The counterpart to the printer are classes that derive from the `MCTargetAsmParser` class. They have two responsibilities crucial to parsing an assembly file. Firstly, they need to transform an assembly string into a sequence of operands. Secondly, they need to match the sequence of operands to a particular instruction opcode. After matching the operand sequence, the matching phase creates the machine instruction instance. Furthermore, this class also supports the handling of target-specific directives. Usually, large parts of these two classes are autogenerated by supplying the `AsmString` property in the TableGen instruction definitions. However, because VADL allows using general-purpose functions instead of string templates, we cannot straightforwardly leverage these capabilities.

The target-specific sub-types of the `MCCodeEmitter` class can emit the binary encoded version of a given machine code instruction into an output stream. Additionally, this class generates a set of fixups while emitting the code. These fixups will become relocations in the resulting object file. Target-specific classes that extend the `MCDisassembler` class handle the other direction. The goal is to create the IR for a machine instruction from a byte array. Much of the encoding and decoding logic is autogenerated by supplying the mapping from format fields to instruction word ranges in the relevant TableGen files.

The last component, the linker, is technically not part of the regular LLVM machine code layer but a project on its own. Therefore, the data structures presented in this section do not exist there. As a result, we cannot leverage the disassembler and code emitter to apply the necessary relocations to a particular instruction word. Section 4.4.7 discusses the consequences of this circumstance. The most notable class in the linker implementation is `TargetInfo`. In our use case, the primary purpose of this class is to support applying relocations to an instruction word.

2.6 Testing Compilers

Due to the inherent complexity of compilers, ensuring correctness is a challenging task. This section gives an overview of techniques that help to strengthen confidence in the correctness of compilers. Even though these methods were not explicitly designed for low-level developer tools, some approaches may translate very well to assemblers and linkers. Furthermore, the LCB also generates a functioning compiler that we can test in conjunction with the binary tools. This section is largely based on a thorough study conducted by Chen et al. [CPP+20] on the testing of compilers.

[9]While LLVM does not perform optimizations at this level, some components can emit hints to the linker for link-time optimization.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Most approaches to compiler testing rely on test programs. The test compiles the program and then tests whether the compiled program captures the same semantics as the source program. The first problem is obtaining a good set of test programs, as they need to be valid without undefined behavior and diverse enough to cover large parts of the implementation. The oldest solution to this problem is manually crafting a set of tests. Most compilers make use of handcrafted test suites. One approach to creating such a test suite is to read every sentence of a language specification and write a test case if this line is testable. While this process is effective, it is work-intensive. Therefore, multiple paradigms surfaced that try to automate the test suite generation. Many approaches use a language's grammar to guide a program synthesis algorithm. One famous example of this category is Csmith [YCER11]. Note that such tools need complex rules to avoid undefined behavior. Other tools introduce mutations in existing programs to cover more language features. These mutations can preserve or alter the semantics of a program.

An essential part of running a test is to recognize undesired behavior. This is called the test-oracle problem. In the case of a handcrafted test suite, the developer may write a set of expectations. If the result does not meet the assertions, the test will fail. However, we do not have such a set of expectations for a generated program. Differential testing is a technique that requires at least two compilers that implement the same language specification. We can detect differences between the program's behavior by compiling the program on both versions. This difference is undesired as both compilers should produce programs with the same semantics, assuming the input program does not have undefined behavior. Note that we can do similar tests by, for example, using different optimization levels in a compiler. Alternatively, a test suite can generate multiple equivalent test programs and test whether they behave equally. This technique is an example of metamorphic testing.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 3

State of the Art

Creating an efficient processor design is a resource-intensive process that includes trying many architectural features and configurations. These choices may prove effective or ineffective while evaluating the new design. Unfortunately, this process may include laborious manual steps, such as searching an assembly code for possible applications of a new instruction. To allow a fast design exploration, we require a streamlined evaluation process. A promising approach to this problem is describing a processor's architecture in a domain-specific language – a PDL. From such a specification, a tool can generate, for example, an optimizing compiler that can leverage the new instructions that an engineer is currently evaluating. The remaining section is based on the introduction by Prabhat Mishra and Nikil Dutt [MD08].

To truly profit from using a PDL, a team must integrate it into their design process. Ideally, the language would be the central part of the process, similar to how programming languages are central to software design. Depending on the language's nature, programs can emit three categories of artifacts from this specification. We call such programs *generators*. Firstly, as already discussed in the example, one of these categories refers to *developer tools* that support the evaluation (and later development) process. In particular, many PDLs support the generation of compiler toolchains and simulators. Secondly, generators can emit *hardware schematics* to speed up the development process of the actual silicon. Lastly, generators may automatically produce a *test suite* to ensure the correctness of the generated or manually implemented artifacts.

One of the quintessential decisions in developing a PDL is choosing the abstraction level. Similar to programming languages, this decision will influence the whole development process. Often, this is a trade-off between abstraction and generality. For example, when a PDL has a low abstraction level by operating over an RTL specification, generators will have a hard time inferring high-level properties of the component. On the other hand, when a language chooses a high level of abstraction by representing high-level constructs (e.g., reorder buffers) as built-ins, a generator can infer a lot more about the

semantics of the processor. However, this may reduce the generality as some designs are baked into the language's design. One remedy to this problem is allowing refinement. For example, allowing users to specify the ISA on a high level while mapping the semantics to a lower-level specification detailing the hardware implementation.

Naturally, the information that a PDL captures must differ depending on its goal. For example, synthesizing configurable hardware components requires a different level of abstraction than a compiler generator. Therefore, choosing the right abstraction level for a new specification language is crucial. Based on these different requirements, we can classify PDLs based on two criteria: *content* and *objective*.

First, we will discuss content-based categorization. Here, we distinguish based on the information that the PDL captures. *Behavioral* languages capture the semantics of the implemented ISA. This information allows the generation of compilers and ISSs. However, synthesizing hardware will not be possible as more details are necessary, for example, the pipeline architecture. While a generator could make many assumptions to generate hardware from an ISA, the resulting schematics will be rigid due to the defaults. *Structural* languages solve this problem by capturing the components and interconnections of the circuit itself. Such a description allows the generation of a CAS and flexible hardware schematics. For example, an RTL representation may be used to specify these properties. Unfortunately, recovering the behavioral semantics from such a low-level representation is infeasible. Therefore, a specification language must incorporate a high-level behavioral view and a low-level structural view to support generating compilers and hardware schematics. *Mixed* PDLs implement this idea. As a result, such languages are particularly well suited for design automation. However, such an approach will inevitably lead to a more complex specification language.

Secondly, the objective-based classification argues over the goal behind a PDL. The first example, *compilation-oriented* languages, enable the generation of a compiler from the specification. The ISA is the central part of the specification for such languages, as the compiler needs to know the instruction semantics. Therefore, behavioral languages are well-suited. On the other hand, the primary goal of *synthesis-oriented* languages is the generation of hardware schematics – the primary use-case of structural PDLs. *Simulation-oriented* languages have multiple facets, as the simulator can operate on different levels of abstraction. For example, generating an ISS requires an ISA specification, while generating a CAS needs details about the implementation of the processor. A similar situation arises for *validation-oriented* languages. A behavioral or structural language is the better fit depending on the output. Note that mixed PDLs support all use cases.

One question remains, how can a team integrate PDLs into their design process? Obviously, the most crucial part is designing a processor in the specification language. However, more often than not, the first design can be improved. Mishra and Dutt [MD08] proposed a model with two essential feedback loops for improving an initial processor design. The first loop is between the simulator and the specification, while the second is between the hardware schematics and the specification. The former provides essential (cycle-accurate) benchmark figures for a set of chosen programs. Note that the tailored

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

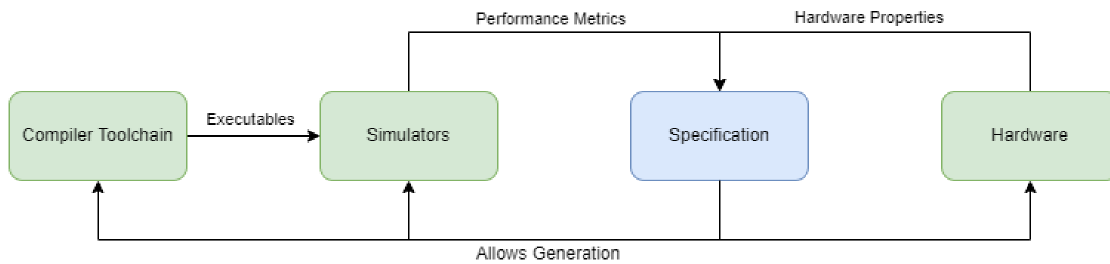

Figure 3.1: Feedback Loops in the Design Process

compiler toolchain is essential for producing optimized executables. The second feedback loop can be used to infer the maximum clock frequency of the generated hardware components. Furthermore, restrictions concerning the die size and production cost can be considered. These insights may lead to changes in the specification, which then triggers a new evaluation cycle. Figure 3.1 shows a simplified version of these feedback loops.

Lastly, we will illustrate the feedback loops with an example. In this scenario, we are developing a processor that should execute a particular set of programs efficiently. First, we will look at the problems and develop an "intuitive" idea for a processor design. Then, we will generate a CAS, a compiler toolchain, and hardware schematics from this initial specification effort. We start by optimizing our architecture for the given set of programs. To do so, we compile the benchmark programs with the generated compiler toolchain to get an executable file. We then execute our benchmark suite on the CAS. This step will give us the number of cycles necessary for executing the benchmark suite. While iterating through new designs, we also must consider the limitations of the generated hardware. By doing that, we can ensure that the die will fit on a predefined surface area and that the clock rate is high enough. For example, even if a highly complex instruction results in fewer CPU cycles, it may still be counterproductive if a reduced clock frequency is necessary.

These promises of a more efficient development process led to the creation of many PDLs. Some specialize in a particular aspect (e.g., simulation), while others try to cover all four objectives. The remaining section will present some well-known specification languages. While summarising the projects, we will pay special attention to the modeling of assembler and linker-related features, most notably the assembly syntax.

The heart of an *nML* [Fre91] specification is an attributed grammar [Knu05]. Such a grammar enables a user to attach auxiliary information to symbols, thus enriching the AST's information. In nML, the grammar itself defines the ISA of the processor. The start symbol of the grammar is the `instruction` symbol. Each string derivable from this symbol corresponds to a single instruction. Note that the grammar's language must be finite; otherwise, the ISA would contain infinite instructions. In other words, recursive rules are not allowed. Additionally, a user can define memory locations and registers outside of the grammar. Furthermore, nML defines a set of predefined annotations that generators can use. These annotations must be present at the root of the derivation tree

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

(the `instruction` rule). The *syntax* attribute specifies the assembly representation by defining an expression that evaluates to a string. The format built-in allows printing operands. It consists of a string similar to a `printf` template and a sequence of arguments (= operands). Using different formatting placeholders (e.g., `%s` for a string, `%b` for a binary formatted number) gives the user control over the assembler syntax. The encoding of instructions is defined similarly by the *image* attribute that evaluates to a string consisting solely of zeros and ones. This construct also uses the same formatting syntax as the syntax attribute. Lastly, the *action* attribute defines the semantics of an instruction. Furthermore, nML also allows the definition of structural components such as pipeline architecture. Thus, the language is a mixed PDL.

The goal of the *Instruction Set Description Langauge (ISDL)* [HHD97] is to describe the semantics and syntax of an ISA. The language specializes in VLIW processors. First, the language user has to define what instruction format fields exist and how they compose the instruction word (e.g., opcode). After specifying the general structure, the user has to define multiple properties for each instruction, such as the assembly syntax of the instruction and how the parameters relate to the format fields of the instruction word. ISDL uses a "grammar rule" that includes the mnemonic and operands to specify the syntax of an instruction. This rule can use predefined tokens (e.g., Register) and user-defined non-terminals. In this context, non-terminals are groupings of tokens with an optional action consisting of arbitrary C code. Because this code will be embedded into a lex/yacc [LMB92] generated parser, knowledge about these tools may be required. Furthermore, designers may specify bundle constraints to support the specification of VLIW architectures. Because ISDL aims to specify an architecture's semantics, it is a behavioral PDL. As a result, generators can emit an assembler, code generator, and simulator from a specification. The idea is that engineers can evaluate new designs quickly by analyzing the architecture's performance on the simulator.

A specification written with *Expression* [GHK+98] consists of Lisp-like expressions, thus keeping the language's syntax simple. Its primary use case is developing System-on-Chip (SOC) architectures and ILP processors. Users must specify the opcode, operands, and behavior to define an operation. For example, a simple operation may add two registers together. Furthermore, each operand in an operation is associated with a group that specifies valid operand types. However, by themselves, operations are not very useful. Users must define instructions to capture the ILP semantics of the processor. Each instruction consists of multiple slots that can be assigned to operands. The designer can specify constraints for each slot, for example, two operations may only operate on different functional units. Furthermore, the language supports operation mappings that define the lowering of compiler-internal operations to architecture-specific ones. Such patterns also may represent target-specific optimizations. In addition to the behavioral definitions, Expression allows for defining structural aspects such as bus connections, caches, and functional units. Thus, the language falls into the mixed PDL category. From a specification, generators can emit a CAS and retargetable compiler.

Modeling a processor is also possible in general-purpose programming languages, as

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

systems such as *SystemC* [Pan01], a set of C++ class libraries, show. However, because the goal is to describe the hardware architecture of a processor, the abstraction level of SystemC is relatively low compared to other languages presented in this section. Therefore, generating tools such as compilers is infeasible. As a result, evaluating a new ISA is a cumbersome process for SystemC users. To improve on that, *ArchC* [ARB+05] targets this user group by providing a familiar syntax and an abstraction level that allows the generation of developer tools. Furthermore, the ecosystem provides a co-verification process that allows comparing the behavior of a reference ArchC specification to a more refined specification. The idea is that including details in the specification may introduce erroneous behavior. Therefore, the co-verification strategy can detect errors by comparing the complex model to a simpler, hopefully correct, model. Users can set the assembly syntax and encoding by "calling" the respective setters on the instruction elements in the "constructor" of the ISA. This design decision shows the commitment to SystemC users. To specify the assembly syntax, the language uses a string template with formatting placeholders parameterized with operands. An instruction's semantics must be captured as a function of the pipeline architecture. For example, if a pipeline was defined, the instruction's behavior must be distributed across the different stages. Fortunately, the reference model can use an architecture with no pipeline, thus allowing designers to catch errors using the co-verification infrastructure. Because ArchC allows defining behavioral and structural elements, it is considered a mixed PDL.

The Language for Instruction Set Architecture (*LISA*) [PHZM99] was developed to support modern the architecture of a modern digital signal processor (DSP). Furthermore, the language supports superscalar architectures and VLIW processors. LISA is a source language for many artifacts, including a compiler, CAS, and hardware schematics. The language uses different models at different abstraction levels to support such diverse tools. For example, users can define operations in the instruction set model and pipeline behavior in the timing model. This versatility makes Likes a mixed PDL. Furthermore, the definitions are organized in sections, for example, the `CODING` section contains the instruction encoding. LISA uses a sequence consisting of string constants and operands to describe the assembly syntax. Users can annotate operands with formatting directives similar to format placeholders. C-like expressions can be used to define the semantics of an operation. In addition to LISA, The Retargetable Architecture Description Language (*RADL*) [Sis98] is another effort that synthesizes developer tools for complex DSPs.

Some languages, such as the Machine Independent Microprogramming Language (*MIMOLA*) [Mar84], incorporate representative processor workloads into the specification. These programs are transformed into an executable format which the system then uses to evaluate the processor design. The primary goal of the MIMOLA Software Systems (MSS), the generators operating on the specification language, is to synthesize efficient hardware schematics. Performance profiling tools provide execution frequencies per hardware component to guide the developers in this task. With this data, engineers can make informed decisions about future processor design modifications.

Retargetable compiler frameworks such as *LLVM* [LA04] also use string templates to

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

define the assembly syntax. Engineers may specify an `AsmString` in the instruction definition when implementing a new target architecture. The string contains the mnemonic and references to operands. This property is solely used to define the order and general layout of the assembler string. Emitting a particular operand (e.g., register, immediate) is facilitated by using general-purpose C++ functions.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 4

Implementation

This chapter discusses the implementation of the high-fidelity prototype. Our generator can emit a tailored LLVM target for the described microprocessor. Tools such as a code generator[1], assembler, disassembler, and linker are part of the target. First, Section 4.1 explains the newly introduced specification concepts. Secondly, Section 4.2 discusses the automatic grammar generation for assembly printing functions. Section 4.3 presents the used data structures, while Section 4.4 talks about the tool generation in detail. Lastly, Section 4.5 then finalizes this chapter by discussing our testing strategy.

4.1 Language Design

The assembly description element captures various aspects required for generating a fully-functional assembler and linker. This language construct depends on an ABI and, by extension, an ISA. All microprocessors that implement said ABI automatically include the assembly description. Defining an assembly description for a microprocessor is mandatory when generating an LCB. For each assembly flavor (e.g., x86 Intel, x86 AT&T), a separate assembly description definition needs to be specified. An assembly description may consist of up to three different sub-elements. The following sections describe these elements and necessary adaptations of the existing language. Some behavior can be changed with annotations to support additional flexibility. The generator falls back to a default if a section or annotation is missing.

4.1.1 New Expressions

Prior to this thesis, VADL did not use general-purpose functions to format assembly instructions. To support this use case, we were required to introduce new expression

[1]The code generator is not part of this work.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

types to the language. Mainly, we made instructions such as formatting a number explicit. This section gives an overview of the new constructs.

Because our approach is similar to FliPpr's [MW13], we face similar challenges. One such problem is instructing the parser to accept multiple versions. For example, in a given assembly language, a blank or a comma may separate assembly operands. The printer will always emit the canonical representation with commas. How is the grammar rule inference (see Section 4.2) supposed to know about this circumstance? We solve this problem similarly to FliPpr by introducing the biased choice operator. The printer will always emit the first operand of this construct, while the parser may accept all operands. Listing 4.1.1 shows the application of the biased choice operator in VADL. The semantics are equivalent to the semantics in FliPpr. Users may use a user-defined function to abstract the separator to avoid repeating the same choice operator.

```
1     assembly ADDI = (mnemonic, " ", register( rd ), choice( ",", " " )
2         , register( rs1 ), choice( ",", " "), decimal( imm ))
```

Listing 4.1.1 also contains the new constructs `register` and `decimal`. These built-in functions transform the format field into the assembly text representation. In addition to decimal, we also support binary, octal, and hexadecimal number representations. In addition to the printing primitives, the operands can be used in complex constructs such as conditionals. The printer must transform the operand to an integral value for that to work. Unfortunately, the LCB cannot guarantee this property for every operand. For example, if the immediate operand contains a complex expression with symbols, we cannot simply extract a number. The built-in formatting functions handle this gracefully by, for example, printing the symbol's name. The printer will emit an error if we cannot extract an integer from an operand used outside of the built-ins. It is the user's responsibility to ensure that for every instance of this instruction, the used operand only contains an integer. For example, in the processor's grammar, a user would use the `Integer` rule that only accepts integers instead of the `ImmediateOperand` rule to ensure this invariant.

4.1.2 Grammar

The grammar definition is the heart of the assembly description. It describes the overall structure of the assembly language and is the foundation for generating the assembly parser. A grammar consists of a set of terminal rules and non-terminal rules. The `DefaultGrammarInjectionPass` defines a set of default rules that are part of every definition. The defaults include the predefined terminal symbols. Because a user cannot define custom terminals, the alphabet the parser operates on (T) is the same for all generated backends. Currently, T is a subset of the terminal symbols that the default LLVM lexer emits. Therefore, the LCB can use the default lexical analysis of LLVM. In addition to the alphabet of the parser, the default grammar also contains a set of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Name	Description
`Statement`	This rule represents the start symbol of the grammar. All instructions need to be derivable from this symbol. The default implementation calls the Instruction non-terminal rule.
`Register`	Applying this rule consumes an identifier and converts the result into a register.
`ImmediateOperand`	Acts as an extension point for custom expressions. A specification should override this rule when a custom expression, such as a modifier, should be possible in an immediate operand.
`Identifier`	Equivalent to the identifier terminal.
`Expression`	The default implementation consists of a parser builtin that can handle complex expressions. Most specifications should not override this rule.
`Instruction`	If not overridden, this rule contains an alternative that covers all rules that produce an instruction.
`Integer`	A positive or negative number.
`Natural`	A positive number.
`Symbol`	Applying this rule consumes an identifier and interprets it as a symbol.

Table 4.1: Default Non-terminal Rules

non-terminal symbols. Table 4.1 describes all default non-terminal rules.

A user can specify custom non-terminal rules. Each rule has a name and a body, which is a tree that consists of *grammar elements*. The syntax and semantics of the basic building blocks are simple. Listing 4.1 shows four different grammar elements. The comment next to a rule specifies its semantics. These four constructs describe the syntax of the grammar. The remaining grammar elements give semantic meaning to parts of the grammar or work with the obtained data. In this work, we refer to the execution of the grammar element's semantics to a token sequence as *application*. Before discussing these essential parts, we need to discuss the type system.

```
1 grammar = {
2     Literal: "a" ; // Accepts the String 'a'
3     RuleReference: <Register> ; // Accepts the same inputs as Register
4     Alternative: [ "a", "b" ] ; // Accepts 'a' or 'b'
5     Sequence: ( "a" "b" ) ; // Accepts 'a' followed by 'b'
6 }
```

Listing 4.1: Basic Grammar Elments

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

An essential invariant for most VIR nodes is that they always have a type. Definition elements that other nodes do not reference (e.g., the whole grammar) usually have a type of `Void`. Nonetheless, each grammar element has a type associated that indicates what data is acquired by applying the element. While these types integrate into the general type system of VADL, the language defines a special-purpose type for describing grammar elements – the `AsmType`. It represents the primitive values that the parser collects from the input string. Note that the actual data structures representing each data type are highly generator-specific. For example, an elementary assembler generator that does not support symbols may use integers to represent operands. In contrast, complex generators that target a retargetable compiler must use the structures that the framework dictates. The following itemization lists all primitive types.

- **@constant** is a 64-bit signed integer. The default rules *Integer* or *Natural* produce this type.
- **@expression** represents a complex expression operand. It may refer to a single constant or an expression tree containing external symbols. VADL provides a built-in that can parse complex expressions. The default definition of the *Expression* rule makes use of this construct. In the LCB, this corresponds to an `MCExpr`.
- **@instruction** represents an entire machine or pseudo instruction. It is composed of at least one `@operand`. Note that the mnemonic is part of the operands. In the LCB, this type corresponds to an `MCInst`.
- **@modifier** holds a reference to a custom relocation. The modifier-mapping construct presented in Section 4.1.4 defines the relationship between the string and the actual relocation.
- **@operand** represents a machine operand. Once created, operands are used to construct instructions. The parser can convert most other primitives into an operand. In the LCB, this type corresponds to an `MCOperand`.
- **@register** represents a successfully matched register. Therefore, the parser already resolved the name to a particular register.
- **@string** represents a character string. Most terminals produce values of this type.
- **@symbol** represents a reference to a symbol, such as an assembly label.
- **@void** is the empty data type. No value is stored.

Like general-purpose programming languages, *casts* allow transitions between the different types. For example, one may cast an integer to a register index. The cast operations are type-checked on the AST level. As a result, all casts in a VIR grammar are valid. Attributes provide the ability to cast and transform the parsed data. The type system

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Source Type	Target Type	Description
@operand	@instruction	An instruction has to consist of at least one operand. These elements will make up the operand vector matched against the set of available instructions.
@register	@operand	Wrap the register number in an operand.
@constant	@operand	Wrap the constant in an operand.
@string	@operand	Wrap the string in an operand.
@expression	@operand	Wrap the expression in an operand.
@symbol	@operand	Wrap the symbol in an operand.
@modifier with @expression	@operand	Create a custom expression with the modifier relocation. Wrap the resulting expression in an operand.
@string	@register	The string represents a register name. We match this string against the available registers to retrieve the register number.
@constant	@register	Reinterpret the value as a register index.
@string	@modifier	The string represents a modifier name. Search for the correct relocation with the modifier mapping.
@string	@symbol	Reinterpret the value as a symbol name.
Any	@void	Drop all data from an element.

Table 4.2: Possible Casts with LCB Semantics

checks whether a cast can succeed. However, it does not guarantee successful casting operations. For example, the cast from @string to @register may fail if the string does not correspond to a known register. This approach is similar to casting reference types in many object-oriented languages. A cast that can never succeed is a compiler error. A cast that *may* succeed may become a runtime error during execution. Listing 4.2 demonstrates this concept. The Register rule calls the terminal rule IDENTIFIER, which produces a @string. Then, the result of the rule is cast to @register. Table 4.2 lists most of the available casts and gives their semantics in the context of the LCB. In addition to the casts from the table, we support the extraction of a single element from a map.

```
1 grammar = {
2     // Type @register
3     Register: <IDENTIFIER>@register;
4 }
```

Listing 4.2: Basic Grammar

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

VADL features a robust type inference algorithm. We extended the existing type inference to support the newly introduced grammar elements. Fortunately, most grammar elements have a trivial type inference rule. For example, a rule reference gets its type from the referenced rule. However, the type inference rule of a sequence may be noteworthy because a sequence needs to distinguish between relevant and irrelevant data. For example, the mnemonic is essential to the semantics, while the comma that separates the operands is not. Nevertheless, both elements have the same type (`@string`). Furthermore, a sequence contains multiple elements. How do we represent the aggregated data?

To address this problem, a user may indicate the significance of a value with a binding to a feature. Features allow a grammar element to have a value that consists of multiple named sub-elements, similar to how structs are composed of members. As a result, a sequence will only collect a parsed value if it is bound to feature, i.e., considered significant, thus separating relevant and irrelevant parts of the assembly string. The type of a sequence is a mapping of features to values – the `MapType`. Listing 4.3 demonstrates the usage of feature bindings in conjunction with sequences.

```
1 grammar = {
2     // Type { mod: @modifier, sym: @symbol }
3     ModifiedSymbol: mod=<Identifier>@modifier
4         "(" sym=<Identifier>@symbol ")";
5 }
```

Listing 4.3: Sequence with Feature Bindings

Alternatives describe mutually exclusive paths a parser can take. We also refer to these paths as branches. The formal pendant to this construct is multiple productions for the same non-terminal symbol. Recall that all productions are valid substitutions. Similarly, all branches are valid syntax alternatives that a parser can match. However, in an unambiguous grammar, only a single branch may lead to a successful derivation. Furthermore, in VADL, every branch of an alternative element has to be of the same type. This restriction allows straightforward and efficient operational semantics of an alternative. Section 4.3 discusses both issues further.

Sometimes it may be necessary to transform the parsed value. For example, we may need to invert the value of an integer if a minus precedes it. In VADL, designers can model such a use case with *transformation functions*. A transformation function is an arbitrary VADL function that takes the target type of the cast annotation as input and produces another value of the same type. For example, a function may take an `@integer` as input and produces another `@integer`. To be precise, the function must take the operational equivalent of the `AsmType` as input. This distinction is necessary because the VIR instructions do not have a semantic for an `AsmType`. The same relationship also applies to the return type of the transform function. Listing 4.4 shows an exemplary usage of transformation functions. The first branch of the alternative will negate the parsed constant because a minus precedes it. The generator creates a new function

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

AsmType	Operational Type
`@constant`	`SInt<64>`
`@string`	`String`
`@void`	`Void`

Table 4.3: Operational Types for Grammar Elements

that contains the code within the parenthesis to facilitate this transformation. The input parameter x refers to the annotated grammar element's operational value. The transformation function will be fully typed and optimized in the same way as a regular user-defined function. Table 4.3 maps some grammar types to their operational pendants. In the future, transformation functions may transform more complex assembly types. A full-fledged generator would support arbitrary functions operating over map types.

```
1 grammar = {
2     // Type @constant
3     Integer: [
4         (<MINUS> v=<INTEGER>)@constant(x => x*-1),
5         <INTEGER>
6     ];
7 }
```

Listing 4.4: Transformation Functions

Note that the grammar should not handle custom immediates with transformation, i.e., the parser should have the "format field view" discussed in Section 2.1.1. After successful parsing, the framework automatically applies the decoding function to the parsed immediate operands. Any associated predicates will be evaluated and checked beforehand to ensure we produce a valid immediate instance.

To conclude this section, we want to give a real-world example. The grammar definition shown in Listing 4.5 describes the syntax and semantics of the `LUI` instruction. This snippet assumes that the necessary modifiers and instructions are correctly defined.

```
1  grammar = {
2      // Example: LUI ra,%hi(main)
3      LUIInstruction: (
4          mnemonic='LUI'@operand
5          rd=<Register>@operand
6          ","
7          imm20=<ImmediateOperand>
8      )@instruction ;
9
10     // override immediate operand to allow modifiers
11     ImmediateOperand: [
12         <Expression>@operand,
13         ("%" reloc=<Modifier> "(" val=<Expression> ")")@operand
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
14     ] ;
15     Modifier: ["hi", "lo"]@modifier ;
16 }
```

Listing 4.5: Grammar Rule for LUI

4.1.3 Alias Directives

Directives are commands to the assembler embedded in assembler files. The LCB uses the directives natively supported by the LLVM framework. If users want different directive names, alias directives can map a new directive identifier to an already existing directive. Example 4.6 shows a possible mapping. The new alias directive ".word" is mapped to the existing LLVM directive ".4byte". This capability is convenient when the processor uses an established assembly syntax.

```
1 alias directives = {
2     ".word"  -> ".4byte",
3     ".hword" -> ".2byte"
4 }
```

Listing 4.6: Alias Directives

4.1.4 Modifier Mapping

Modifiers are value transformations used in assembly programs. Sometimes modifiers can be applied during assembly (constants) and sometimes during linking (symbols). A *modifier mapping* defines the relationship between the modifier and the relocation that implements the transformation. Listing 4.7 describes a subset of the modifiers defined by the RISC-V instruction set architecture. It maps the string used in the assembler language (e.g., `hi` and `lo`) to the relocations that implement them. This relationship allows the parser to map a string to a particular transformation (cast to `@modifier`). Furthermore, the printer can emit the correct assembly name.

```
1 modifiers =
2 {
3     "hi" -> RV32I::hi20,
4     "lo" -> RV32I::lo12
5 }
```

Listing 4.7: Modifier Mapping

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.1.5 Annotations

VADL supports annotations for all definitions. They provide supplementary information regarding the annotated specification element. The behavior of the generated low-level developer tools can be influenced by attributing the assembly description definition. Listing 4.8 contains examples of the supported annotations. In addition, the following list explains them briefly.

- **alignmentIsInBytes** indicates whether the `.align` directive is given in bytes (true) or as an exponent (false). When the alignment is given as an exponent, `.align 4` will align to a $2^4 = 16$ byte boundary. The default is `true`.
- **caseSensitive** indicates whether the parser should be case-sensitive. This behavior applies to all string comparisons in the parser (e.g., mnemonics, modifiers). The default is `false`.
- **commentString** sets the character that initiates a line comment. Note that C-style multiline comments are always possible. The default is `"#"`.
- **generateGrammarRules** allows users to disable the grammar generation for this assembly description.
- **relocationStrategy** dictates the relocation strategy when generating the ELF definitions for the architecture. See Section 4.4.2 for a thorough discussion.

```
1 [ alignmentIsInBytes = true ]
2 [ caseSensitive = false ]
3 [ commentString = ";" ]
4 [ generateGrammarRules = true ]
5 [ relocationStrategy = "performance" ]
6 assembly description ASM for ABI = {
7 ...
8 }
```

Listing 4.8: Example for Assembly Description Annotations

4.2 Grammar inference

Machine and pseudo instruction require an associated assembly printing function. They take the format fields of a concrete instruction instance as input and produce a string that represents it. Our generator also needs a grammar that describes the assembly language for the parser, the inverse of the assembly printer. Naturally, both definitions have a similar structure. The goal of the grammar inference system is to limit a user's specification burden by generating large parts of the grammar from the printing functions.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Section 2.4 introduces multiple approaches to this problem. Our approach is similar to FliPpr [MW13]. We let the user define the pretty printer and infer a parser from the formatting function. However, we can leverage the restrictions of VADL functions to use a more straightforward approach to program inversion. As of this writing, functions have no side effects and cannot be recursive. Furthermore, the size of possible inputs in printing functions is usually tiny compared to general program inversion. Given these circumstances, we were able to apply a simple scheme that proved successful in practice. See Section 5.4 for a detailed discussion of the results. We also considered implementing it the other way around, inferring the printing functions from the grammar definition. However, this approach combined poorly with the macro system of VADL.

We introduce the notion of slices – our immediate representation that contains the necessary information to generate a grammar rule. A slice represents a part of a string. Naturally, the different types of slices mirror the structure of the grammar elements. For example, an `AlternativeSlice` indicates that this part of the string has multiple possible structures. In addition to slices representing grammar elements, there is an `EmptySlice` that represents data that is not associated with a token of the input and a `BotSlice` that indicates an unparseable part of a string. Each slice contains metadata indicating the type and whether the slice is associated with a parameter or a VADL function. Each slice, except `BotSlice`, can generate a semantically equivalent grammar element.

Each VIR node has an associated slice semantic that represents a mapping from the node to a slice. Some nodes have native semantics encoded in our analysis pass, while others default to a scheme presented later in the section. Table 4.4 explains all native semantics in natural language. Note that some nodes take a vector of nodes n as input. The expression n_i denotes the i-th element of the operand vector. We decided against introducing a formal definition because we consider these semantics very intuitive, so formalisms would only hinder readability. We can emit grammar rules for many instruction formatting functions with only these eight semantics. For example, the assembly printing function defined in Listing 4.9 can be successfully analyzed solely with native slice semantics. Furthermore, note that this is the only special-purpose code required to support the biased choice operator in the parser.

```
1 assembly ADDI = ( "addi", " ", register( rd ), ",",
2                   register( rs1 ), ",", decimal( imm ) )
```

Listing 4.9: Assembly Formatting Function for RISC-V ADDI

We handle instructions without a native slice instruction with a generic constant evaluation approach. For this purpose, the algorithm requires an evaluation program that operates over the VIR. We denote the evaluation process as $\theta \vdash_f i \downarrow v$. This formula means that in function f, the instruction i evaluates to value v given the environment θ. Furthermore, let $\theta(x)$ denote the value of x in the environment θ. Fortunately, the instructions used for

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Node Type	Semantic
`AsmInstruction`(n)	If the operand type is formatted as a register, we emit a `RegisterSlice` with the metadata of n_1.
`CastInstruction`(n)	If we cast an integral value to a string, we emit an `ExpressionSlice` with the metadata of n_1. Otherwise, this instruction does not have a native slice semantic.
`ChoiceInstruction`(n)	Emit an `AlternativeSlice` that covers all n_i.
`ConcatInstruction`(n)	Create a `SequenceSlice` with all semantics of n_i as children.
`ConstInstruction`	If the constant is a `StringConstant`, use a `StringSlice` with the constant's value. Furthermore, we emit metadata if the string constant is the mnemonic of the currently processed instruction. Otherwise, this instruction does not have a native slice semantic.
`ParameterDefinition()`	Emit a `BotSlice` with the parameter name as metadata.
`ProbeInstruction`(n)	Use the slice from the parameter n_1.
`RetInstruction`(n)	Use the slice from the parameter n_1.

Table 4.4: Native Slice Semantics

assembly printing functions have no access to any state (e.g., global variables). Therefore, the environment only consists of the parameter assignment. This assignment has to be complete, i.e., each parameter has a known constant value. As a result, all instructions in the function have a value solely determined by the parameter assignment. Note that this only holds for pure VADL functions, not processes. However, the assembly printer only uses the former type.

In order to construct the evaluation program, we thought of two implementations. Both programs take a function definition and a parameter assignment θ as input. The first approach is to interpret the control flow of the function. The algorithm finds the solution once the interpreter evaluates the instruction i to a value v. While this works, the generator wastes time evaluating instructions that are not required. For example, even if instruction a comes before instruction b in the control flow, the value of a may not flow into b, thus making the computation unnecessary. In contrast, the second approach only evaluates instructions necessary for computing the value of i. Fortunately, the VIR stores operands as references to the instructions that produce them. This data layout allows a straightforward implementation of this algorithm by starting at the return instruction and recursively evaluating the operands. The fact that each assembly printing function only has a single return instruction makes this approach even more manageable. After considering both options, we implemented our evaluator with the latter strategy. The following paragraphs explain the role of this component in the context of an example.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
1 function WRegSP ( x : Index ) -> String =
2     if ( x = 31 ) then "wsp" as String else ( "w" + decimal( x ) )
3
4 assembly ADDI32 = ( "add", WRegSP( rd ), ",", WRegSP( rn ),
5                     ",", decimal( imm12 ) )
```

Listing 4.10: Assembly Formatting Function for more complex ADDI

The more complex formatting function shown in Listing 4.10 has instructions that do not have a native slice semantic. We will refer to this function as p in the following example. Before our generator can start inferring grammar rules, it must eliminate calls to other functions. This step is necessary because our evaluator only works locally in a single function. It is the responsibility of the `VirAssemblyInliningPass` to inline all function calls within assembly printing functions.

The evaluation of slice semantics starts at the single return instruction. The native slice semantics of the `RetInstruction` is equal to the semantics of its operand. For the `ADDI32` formatting function, this operand is a `ConcatInstruction`. The slice information of a concatenation is a sequence slice spanning all slices of the operands. Operands such as the string constants have a simple native slice semantic. However, at one point, our analysis will compute the slice information of one of the multiplexer instructions from the `WRegSP` function. This instruction does not have a native slice semantic. Therefore, the generator falls back to an evaluation-based strategy.

The algorithm starts by computing the parameter set of the multiplexer expression. We will refer to this instruction as m in the following paragraphs. This set contains all parameter definitions reachable by recursively following the operands of m. In other words, all parameters that influence the value of m. In order to keep tracing a concrete example, let us assume that m instruction corresponds to the expression `WRegSP(rd)`. The parameter set of this instruction is $\{$`rd`$\}$.

The algorithm now looks up the bit width of this format field and iterates through all possible values. In this case, rd has $2^5 = 32$ possible values. Then, the algorithm computes a parameter assignment for every value, setting operands not in the parameter set to 0, thus creating 32 different evaluation environments. For each environment θ_i, the generator calls the constant evaluator, which will compute $\theta_i \vdash_p m \downarrow v_i$. Because the concatenation produces a string, all operands must have a string type[2]. In this example, this constant may be the string "w1" when evaluated with $\theta_1(rd) = 1$. Once the algorithm has acquired v_i, it creates a string literal grammar element matching against v_i. Then, the algorithm combines this grammar element with a producer (explained in Section 4.3) generating the original value of rd in θ_i in a sequence element. Intuitively this construct matches against the result of the constant evaluation and emits the value

[2]This is because the string concatenation is only defined over other strings. If this instruction has a type other than string, the type inference would have failed earlier in the pipeline.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

that produced said output. Afterwards, the generator collects all 32 computed sequences into a single alternative slice, representing the slice information for m.

```
1  // Simplified version with fewer known registers
2  function formatCSRRegister( index : Bits< 12 > ) -> String =
3      match( index ) with
4      { 0x000 => "ustatus", 0x004 => "uie", 0x005 => "utvec"
5      , 0x040 => "uscratch", 0x041 => "uepc", 0x042 => "ucause"
6      , _ => hex( index )
7      } as String
8
9  assembly CSRRW = (mnemonic, ' ', register( rd ), ',',
10                         formatCSRRegister( imm ), ',', register( rs1 ))
```

Listing 4.11: Assembly Formatting Function for CSRRW

The proposed generation schema may derive unnecessary complex rules or rules that will lead to unintuitive error messages containing thousands of possible alternatives. A good example is the `CSRRW` instruction of the RISC-V ISA. Listing 4.11 shows the assembly printing function of this instruction. The idea is that a control and status register (CSR) may not have a name. However, for well-known registers, we want to emit the human-readable version, i.e., the string representation. Unfortunately, applying the presented approach will lead to an alternative with $2^{12} = 4096$ branches. Furthermore, our grammar would be too restrictive as we would no longer allow other representations for these registers. For example, defining the register index with a decimal or using a number for a well-known CSR is no longer possible. Note that designers can solve parts of this problem with the biased choice operand.

We address this issue with a relaxation technique which is currently very simple. During the constant evaluation, we track which instruction produced the final constant. Instructions that simply forward the result do not alter this metadata. For example, the return instruction at the end of a function will not erase the "source" of the constant. The current strategy is to collapse all constants produced by a number formatting instruction to a single call to the `Integer` non-terminal. While this solution is not perfect, it is sufficient to handle the CSR instruction in the RISC-V specification. Listing 4.12 shows the generated grammar with the relaxation strategy in place.

```
1  [{ value:@constant }]
2  Computed_0: [
3      ("ustatus" value=(produce_0.0)), ("uie" value=(produce_4.0)),
4      ("utvec" value=(produce_5.0)), ("uscratch" value=(produce_64.0)),
5      ("uepc" value=(produce_65.0)), ("ucause" value=(produce_66.0)),
6      value=<Integer>
7  ] ;
8
9  [@instruction]
10 CSRRSInstruction: [(
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
11      mnemonic='CSRRS' @operand
12      rd=<Register>@operand
13      ","
14      imm=[[<Computed_0>]@constant]@operand
15      ","
16      rs1=<Register>@operand
17  )]@instruction ;
```

Listing 4.12: Generated Parser Rules for CSRRW

Another problem was that our solution computed duplicate helper non-terminals (e.g., Computed_0) when applying the fall-back slice semantic for the multiple equivalent expression trees. These duplicates clutter the grammar definition, thus making debugging more cumbersome. By remembering what slices the algorithm has already computed, it can avoid unnecessary non-terminals. Furthermore, the solution needs to know whether two instructions must have the same semantics, thus allowing it to reuse already computed non-terminal symbols. Again, we use the fact that assembly printing functions cannot depend on any form of state. Thus equivalent instructions with equivalent (transitive) operands must produce the same value. The idea is that two instructions must have the same slice semantics if they always evaluate to the same value. An important aspect is that the name of a parameter definition is irrelevant, but its size is integral. The improved algorithm will reuse existing computed non-terminals if it encounters equivalent instructions.

Lastly, we would like to discuss a similar approach we tried for generating grammar rules. In this solution, every VIR node had a native slice semantic. Nodes without a corresponding implementation for this semantic caused the generator to abort the rule generation. Even though the VADL team tried to keep the semantics of the VIR as simple as possible, the complexity of the analysis surged due to the number of different instructions. Therefore, while working on analysing multiplexer instructions, we took our time to try another approach – the current implementation. Firstly, the new implementation is more straightforward than defining complex slice semantics for all instructions. Furthermore, in the old implementation, adding an instruction to the VIR required defining a new slice semantic. The new approach reuses the operational semantics of the instruction. Lastly, this architecture profits from all improvements to the constant evaluation component. However, we acknowledge that a more sophisticated approach may lead to better results.

4.3 Immediate Representation

Before we discuss the generation of the generator code, we need to discuss the immediate representation associated with the language design. The `AssemblerDescription` object bundles the information necessary for generating low-level developer tools. Each assembler description consists of multiple definitions resembling the elements described

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

in Section 4.1. While most classes representing this information are simple POJOs that mirror the information of the language structures, some of the grammar elements have invariants that simplify code generation. In this section, we want to briefly describe the IR's semantics and present any existing peculiarities.

A `Grammar` definition consists of multiple `NonterminalRules` and `TerminalRules`. Each rule is identified with a unique name. On the one hand, each terminal rule has an associated regular expression. Using this information, we can infer a terminal rule for a given string literal. We use this capability to check for ambiguities in the grammar definition. On the other hand, a non-terminal rule has a body that is a single `GrammarElement`.

"Grammar element" is an umbrella term used for elements on the right-hand side of a non-terminal rule definition. Each such element has an operational semantic associated with it. Intuitively, this semantic represents a parser implementation. The remaining section discusses different instances of such grammar elements and their semantics. For each IR element, we will elaborate on the operational semantics in natural language. Furthermore, we will give a computational description in Haskell[3], a well-known functional programming language. Even though the used data structures are straightforward, readers can find the definitions in Section A.2. However, the most important aspects will be discussed in this section. The `parse` function presented in Listing 4.13 models the semantics. Given a grammar element and a list of tokens, it returns a parsing result. The `Res` type can either represent success or failure. A successful parsing process also contains the obtained data and the remaining tokens. We define the semantics of each grammar element as cases of the `parse` function.

```
1 parse :: GrammarElement -> [ Token ] -> Res
```

Listing 4.13: Definition of the Parsing Function

A `StringLiteral` is one of the most simple grammar elements. It has one parameter that describes the expected string. Listing 4.14 shows the definition of the semantics. During parsing, we compare the current token's string value to the string stored in the literal. If the two values match, the result will be the string value itself. Otherwise, we emit an error. Another critical building block is the `RuleReference`. It allows applying another rule to the input. In other words, we call the method generated for the rule. The value of the rule reference is the same as the value produced by the rule application.

```
1 parse (NonterminalRule _ body) tokens = parse body tokens
2
3 parse (Literal expected) ((_, (DStr actual)):xs) =
4   if expected == actual
```

[3]https://www.haskell.org/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
5     then Ok (DStr actual) xs
6     else Err "Wrong String"
7 parse (Literal _) _ = Err "Token was no string"
```

Listing 4.14: Non-terminal Rule and String Literal Semantics

We must use constructs that combine multiple elements to model more complex derivations. For example, a `Sequence` chains multiple elements together. The system applies each element to the input and checks whether the element was parsed successfully or not. If there was an error during the application, the value of the sequence is the error itself. If all elements were processed successfully, the result of the sequence is the combination of all `MapTypes`. Recall that a `MapType` associates a key with a particular value. To hoist a standard value to a map type, we use the `Binding` construct. It is parameterized by a key and an inner grammar element. Once the algorithm applies the inner element successfully, the resulting value is associated with the key. Values not associated with a key are not part of the sequence value. This rule allows a user to drop semantically irrelevant parts of the assembly string (e.g., commas). Note that we validate that the keys in a `MapType` do not collide. Therefore, we do not have to handle this case in the operational semantics. Listing 4.15 provides a reference implementation of the sequence and binding semantics.

```
1 parse (Sequence []) tokens = Ok Void []
2 parse (Sequence (x:xs)) tokens =
3   let rem = (Sequence xs) in
4   case (parse x tokens) of
5     Ok (DKeys el1) ts1 ->
6       case (parse rem ts1) of
7         Ok (DKeys el2) ts2 -> Ok (DKeys (el1 ++ el2)) ts2
8         Ok _ ts2 -> Ok (DKeys el1) ts2
9         err -> err
10    Ok _ ts1 -> parse rem ts1
11    err -> err
12
13 parse (Bind to inner) ts =
14   case (parse inner ts) of
15     Ok val rem -> Ok (DKeys [(to, val)]) rem
16     err -> err
```

Listing 4.15: Sequence and Binding Semantics

An `Alternative` allows different branches in the application of a rule. Each branch represents a sub-element of the alternative. Listing 4.16 shows the semantics. The `choose_alt` function is responsible for selecting the best branch. The only requirement is that if it is possible to apply a branch successfully, then this branch has to be chosen. To give a more concrete example that is also highly relevant to the LCB, we will discuss a function that selects a branch for decisions in an $LL(1)$ grammar. Note that the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

restrictions for VADL grammars apply in this scenario. The function computes the FIRST set for every possible branch. Then it checks whether the head of the tokens list is compatible with an element in the FIRST set of the branch. In an unambiguous $LL(1)$ grammar, this condition should only hold for at most a single branch. This branch is returned and then applied in the semantics. In the case that no branch fulfills the requirements, an error is emitted.

```
parse (Alternative xs) tokens =
    case (choose_alt xs) of
        Just alt -> parse alt tokens
        _ -> Err "No branch is selectable"
```

Listing 4.16: Alternative Semantics

During the DefaultGrammarInjectionPass, our generator injects some special-purpose builtins that a user cannot create via the language. The LCB lowers some of these builtins to traditional grammar elements at the end of the pipeline. For example, a pass replaces the Instruction built-in with an alternative referencing all rules with a type of @instruction. Therefore, these builtins do not require an operational semantic. As of this writing, the only built-in that is not lowered is Expression. During application, this element must be able to construct complex assembly expressions. The complexity of these expressions is implementation-specific.

Finally, we have IR elements that operate on the data values from other grammar elements. The Binding construct is also part of this group. It was explained earlier due to the strong connection with the sequence element. Listing 4.17 contains the definition of the remaining grammar elements. One such data-focused element is the ValueProducer. This construct produces a value without consuming tokens. It does this by calling a function with zero parameters. Naturally, its value is equal to the value that it creates by evaluating said function. Note that this element is the only construct with no pendant in the specification language. Similarly, a ParsedValueTransform element can transform a single value into a different one by calling a function with the original value as an argument. The value of the element is the return value of the called function.

In a concrete implementation, the ParsedValueCast is the most complex element because it depends on the type of the cast. For example, the cast from a @constant to a @register is relatively straightforward, as we are just changing the interpretation of the value. Other casts, such as the cast from @string to @register, need to match the text against the set of available registers and register aliases. The table 4.2 gives a good impression of the logic that is necessary to implement the required casting behavior. However, recall that the data structures are implementation-specific. By extension, the casts also do not have a well-defined operational semantic. The only restriction is that the result has to be compatible with the definition of the target type. Therefore, in the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

casting semantics, we wrap the original data in a cast element. Evaluating this element will correspond to executing the cast operation in implementations.

```
parse (Cast to inner) ts =
  case (parse inner ts) of
    Ok result rem -> Ok (DCast to result) rem
    err -> err

parse (Transform fn inner) ts =
  case (parse inner ts) of
    Ok val rem -> Ok (fn val) rem
    err -> err

parse (Producer fn) ts = Ok (fn) ts
```

Listing 4.17: Semantics of Data Elements

VADL imposes an additional restriction on alternatives. Each branch has to have the same type. This way, the value of the alternative is equivalent to the value of the applied branch. No transformation into a different data type is necessary. If we had chosen to lift the restriction, we would have to use a type that describes multiple possible types – an `AlternativeType`. While this is possible in VADL, the algorithm would need to do runtime checks to determine which value is currently present in the element's value. For example, consider the following alternative, one branch produces a `@register` while the other creates a `@constant`. The result of the alternative is transformed to an `@operand`. Both casts are valid, thus the cast of the alternative type is valid too. Without grammar rewriting or other optimization techniques, the parser would have to check which concrete value the element holds. This check is necessary because the casting semantics depend on whether the register or constant branch was matched. Furthermore, this would require that the data type of variables must be flexible enough to have multiple values. With the restriction in place, each element has exactly one statically known value type, thus there is no alternative type in the whole grammar.

4.4 Tool Generation

Parallel to designing the specification language, we implemented a low-level developer tool generator in the LCB. Recall that this generator can emit a new target for LLVM. This work enhances the existing implementation by emitting the machine code layer for the target. In an LLVM target, this layer is responsible for processing assembly and object files. This section discusses the significant extensions to the LCB made during this thesis. Many correspond to a low-level developer tool presented in Section 2.2. Before we discuss the tools themselves, we shall give an overview of topics that pervade multiple tools, such as immediate handling and relocations.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

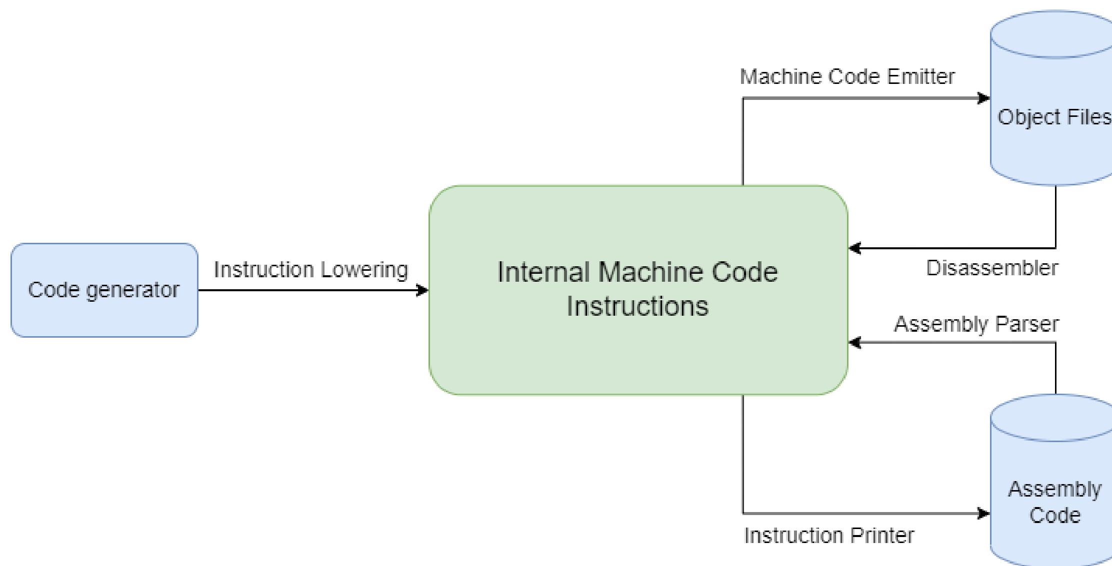

Figure 4.1: Internal and External Instruction Representations

4.4.1 Automatic handling of immediates

As discussed in Section 2.1.1, there are two views on immediate operands. This section presents a solution to bridge the gap between the two views. We consider the `MCInst` representation our internal system, while all other forms representing instructions are considered external. For example, the assembly code for a particular instruction is an external representation, while the in-memory data produced by the parser is considered internal. All immediate operands internal to the system are in the semantic view. Therefore, all tools that produce the internal data structures must be able to correct a possible mismatch. These tools represent the boundaries of our internal system. Figure 4.1 shows the different representations. The diagram shows the internal representation in green and all external systems in blue. The arrows correspond to border systems that either ingest (entry boundary) or create (exit boundary) external representations. By requiring a uniform internal representation, each boundary system is independent. As a result, for example, the code emitter does not have to check whether the data comes from the parser or the instruction lowering. Both types of data are represented and handled in the same way. An astute reader may have noticed that the linker is not part of the diagram. This is because the LLVM linker implementation does not use the `MCInst` internal representation. Still, the linker has to adhere to similar rules regarding immediate handling. Section 4.4.7 discusses this issue further.

We will use an example to illustrate the responsibilities at the system boundaries. Imagine an assembly file for a RISC-V processor that consists of a single `LUI` instruction. This instruction has a 20-bit wide format field. To get the semantic value, one needs to shift this format field by twelve bits and interpret the result as a 32-bit unsigned integer. Our task is to ingest the assembly file and then pretty print it. This task is facilitated by

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

converting all instructions into the internal machine code representation and emitting this data into another assembly file.

The assembly parser is the first relevant component in this scenario. It reads the line with the `LUI` instruction and constructs a vector of operands. The first operand contains the mnemonic, the second is the target register, and the third is the immediate operand. The parser is aware that it needs to shift the value of the `imm` operand if it is an integer value. However, the operand may also be a `hi` modifier which already produces a 32-bit integer. In this case, shifting would extend the value to a 44-bit integer, thus producing an invalid immediate value. If the operand requires shifting, our parser wraps the value in a custom expression class that contains the immediate decoding logic.

Now that the system obtained a canonical representation of the instruction, its task is to print it. Again, our instruction printer can encounter two scenarios: The `LUI` instruction is an integer or a complex expression containing a symbol. For the former case, the system can evaluate the operand to an integer and apply the encoding function of the immediate. In this case, it would shift the 32-bit value twelve bits to the right, thus obtaining the original 20-bit value that fits in the format field. In the latter case, the printer textually emits the expression tree.

Even though the process is slightly different for every entry boundary system, the rules for adjusting the immediate value are the same. The following enumeration lists the different scenarios that the boundary system may encounter. Note that some systems may have a simplified version. For example, the disassembler will not have to handle custom expressions.

1. The semantic value and the format field value coincide and no modifier is applied. In this case, no transformation is necessary. Note that our implementation still emits a custom expression representing this circumstance. However, the contained code will be a no-op.

2. The semantic value and the format fields value differ and no modifier is applied. The `LUI` example covers this case. The boundary system must wrap the operand in a custom expression that bridges the gap between the two immediate representations. Another approach could transform the respective integer value.

3. The operand has an associated modifier. In this case, the system uses the custom expression for that modifier. Because this relocation must return the semantic value of the operand, there is no need for another custom expression.

4.4.2 Relocations

In hand-crafted LLVM backends, relocations are carefully designed to be as efficient and expressive as possible. Being efficient is necessary because the number of relocation types per architecture is limited. The info field of a relocation entry consists of the symbol table index and the relocation type. In the LLVM infrastructure, the lowest byte of this

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Relocation Type	Description
NONE	Signals that no relocation is necessary.
32	The value of this relocation is emitted as 32-bit integer.
64	The value of this relocation is emitted as 64-bit integer.

Table 4.5: Default Relocations

field indicates the relocation type, while the remaining field points to the symbol table entry. As a result, each architecture may define up to 256 relocations. Note that this limit only applies to the 32-bit version of ELF. It should be noted that most hand-crafted backends also have a wider variety of supported relocation types (e.g., symbols in a procedure linkage table).

This limitation leads to an interesting trade-off – the number of relocations versus the information associated with a single relocation type. The following example illustrates this. The official RISC-V backend in LLVM defines the two relocation types `R_RISCV_LO12_I` and `R_RISCV_LO12_S`. Both of these relocations take the lower twelve bits of the relocatable value. The postfix indicates whether this is for an `IType` or `JType` instruction. Both instruction formats store the immediate operand in different bit ranges. With this knowledge, the linker can override the necessary bits without further analysis. Otherwise, the program must decode (parts of) the instruction word to infer the bit ranges. In this case, the designers decided to favor the amount of information over the number of relocations. Another backend may use a single relocation to represent both cases at the cost of linker performance.

Often, the backend designers can estimate how many relocation entries are needed. This information may guide them while considering the tradeoffs. However, we do not know how large a specification may become, so we need to to allocate relocations efficiently. Table 4.5 lists the default relocations that every LCB-generated target allocates.

Before deciding on how much information we want to implicitly associate with each relocation type, we need to find a solution to infer the rest of the information. For example, we need to know the format field that the relocation changes and the respective immediate encoding function. We close this gap by using a decoder in the linker. This decoder takes the instruction word as input and returns the instruction type. Therefore, the relocations only need enough information to determine the format field within the inferred instruction type. While this technique will hurt linking performance, it will allow us to keep the number of used relocations low. However, to provide users with additional flexibility, we implemented two different allocation strategies – conservative and performance. Chapter 5 discusses the performance tradeoffs.

```
1 ELF_RELOC(R_CPU_NONE, 0)
2 ELF_RELOC(R_CPU_32, 1)
3 ELF_RELOC(R_CPU_64, 2)
4 ELF_RELOC(R_RV32I_hi20_1, 3)
5 ELF_RELOC(R_RV32I_lo12_1, 4)
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
6 ELF_RELOC(R_CPU_SYMB_12_PCREL, 5)
7 ELF_RELOC(R_CPU_SYMB_20_PCREL, 6)
```

Listing 4.18: Relocations of the RV32I Processor with Conservative Strategy

Before discussing the two relocation strategies, we want to explain the architecture of our relocation system. First, and most importantly, we distinguish between two types of relocations – *regular* and *ELF* relocations. Regular relocations are the functions that users can define themselves. Their most important aspect is defining the transformation function. Now, a regular relocation may have multiple associated ELF relocations. Each one is connected with at least one machine instruction format field. An ELF relocation may encode further details about the format field, for example, the immediate and the encoding ranges. In the conservative relocation strategy, the ELF relocations will not encode additional information. Therefore, each regular relocation will have at most one ELF relocation[4]. However, in the performance relocation strategy, some relocations will have multiple ELF relocations, for example, representing the `lo12` relocation similar to the official RISC-V backend.

Additionally to these two types, the LCB has two passes that specifically handle relocations. One pass inserts default relocations to reduce the specification burden on the users, while the other pass generates ELF relocations based on the set of regular relocations and the available machine instructions. The `GcbRelocationInjectionPass` is responsible for creating default relocations for format fields that allow storing a reference to a symbol. In general, this will not be possible for absolute relocations because the format field would need to be 32-bits wide, assuming a 32-bit architecture. The `GcbElfRelocationGenerationPass` generates the necessary ELF relocations.

The LCB supports two primary types of relocations – absolute and PC relative. This distinction is necessary to steer the computation of the format field's value based on the symbol. The infrastructure passes this value into a target-specific function that patches the instruction word. With this architecture, computing the symbol's value is target-agnostic. However, this computation does not respect some architecture-specific properties. For example, in some architectures, the PC points to the end of the instruction word, while LLVM's infrastructure assumes it points to the start of the current instruction. As a result, linkers for these architectures would produce invalid PC-relative relocation values without further effort. We can obtain the correct value by either emitting an addend in the relocation entry or subtracting the same value in the target-specific part.

Both relocation strategies use the relocation type in their encodings. In this context, encoding refers to including information in the name and semantics of the relocation. For example, if the system encodes an immediate in a relocation, only format fields with this immediate will be associated with this relocation. Additionally, the conservative strategy

[4]A regular relocation may have no associated ELF relocation. This happens when no machine instruction has a fitting format field for a given regular relocation.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

encodes the width of the format field in the relocation. This information allows us to support multiple differently-sized immediate operands in a single instruction. Listing 4.18 shows the list of relocations for the RV32I processor with the conservative strategy. `R_CPU_SYMB_12_REL` is the automatically generated relative relocation for format fields 12-bit wide. However, this representation gives us no information about the format field's immediate type and encoding ranges.

In contrast to the conservative strategy, the performance strategy contains all the necessary information for relocating a value. Firstly, instead of only recording the bit-width of a format field, we use the whole set of encoding ranges. For example, `12_31` indicates that the format field is stored in the range `[31..12]`. Additionally, we also enrich the information by recording the immediate type. Listing 4.19 shows the list of relocations[5] for the RV32I processor with the performance strategy. Note that two different ELF relocations represent the `lo12` relocation. This strategy works in the current state of VADL so well that we chose the performance relocation strategy as default. The conservative strategy may become more and more useful once VADL supports more sophisticated relocations and thus needs more relocation entries. Architectures may switch to the conservative relocation strategy by specifying an annotation.

```
1 ELF_RELOC(R_CPU_NONE, 0)
2 ELF_RELOC(R_CPU_32, 1)
3 ELF_RELOC(R_CPU_64, 2)
4 ELF_RELOC(R_RV32I_hi20_1_ImmediateU_12_31, 3)
5 ELF_RELOC(R_RV32I_lo12_1_ImmediateS_25_31_7_11, 4)
6 ELF_RELOC(R_RV32I_lo12_1_ImmediateI_20_31, 5)
7 ELF_RELOC(R_CPU_SYMB_12_PCREL_ImmediateB_31_31_7_7_25_30_8_11, 6)
8 ELF_RELOC(R_CPU_SYMB_20_PCREL_ImmediateJ_31_31_12_19_20_20_21_30, 7)
```

Listing 4.19: Relocations of the RV32I Processor with Performance Strategy

4.4.3 Assembly Printer

The assembly printer is responsible for emitting the in-memory representation of an assembly file. Some parts of this process are target-agnostic such as emitting most directives. However, the most crucial part, printing the instructions, is target-dependent. In VADL, every instruction that may be present in an assembly file needs an assembly printing function. This general-purpose function maps an instruction word to a string representation. Before applying this function to the operands of an instruction, we need to apply immediate transformations, as discussed in Section 4.4.1.

Furthermore, compiler instructions need special attention while emitting the assembly code. While the stream of instructions may contain them, they do not have a string representation and, by design, should not be part of any assembly file. Because compiler

[5]The list contains shortened names to improve readability. All essential information is still included.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

instructions are a specialized type of pseudo instruction, calls to other instructions express their semantics. We call the process of translating compiler and pseudo instructions to their semantic equivalent “pseudo instruction expansion”. During this process, we may need to apply immediate transformations. Fortunately, this process has already been implemented in the generator. Note that we do not lower regular pseudo instructions in this component as they often improve the readability of the resulting assembly file.

```
1 function formatCSRRegister( index : Bits< 12 > ) -> String =
2     match( index ) with
3     { 0x000 => "ustatus"
4     , 0x004 => "uie"
5     , _ => hex( index )
6     } as String
7
8 assembly CSRR =
9     (mnemonic, register( rd ), ',', formatCSRRegister( csr ))
```

Listing 4.20: Complex Expression with Operand

A user may use the operands of a function for complex expressions (e.g., in a conditional). Listing 4.20 shows a simplified example of such a use case. Unfortunately, this feature comes with a big CAVEAT. Our evaluation strategy for such constructs is relatively straightforward. First, we wrap the `MCOperand` (e.g., `csr`) in a wrapper class. This class may be used directly in the number and register formatting builtins. Outside of these builtins, the printer evaluates the contained operand to the format field value it represents. However, such an evaluation is only possible for operands with a constant value or an absolute expression. In other words, if an expression contains a symbol, the instruction printer issues an error. A more sophisticated approach could hoist the types into a new domain that handles missing values. Such a solution requires defining additional VIR semantics in the hoisted domain. For example, comparing to a non-existing value could always return false or trigger the fallback branch of a match expression.

4.4.4 Disassembler

LLVM makes it relatively easy to emit a working disassembler. The TableGen definition of an instruction definition contains all the information necessary for decoding it. The disassembler needs to provide the decoding capabilities for operands. Decoding a register operand is relatively straightforward, consisting only of a validation step followed by emitting the correct register number. The pendant for immediate operands may be slightly more involved as the disassembler needs to apply the necessary immediate transformations.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.4.5 Assembly Parser

The assembly parser ingests assembly language and produces the corresponding in-memory representation. Looking at a particular architecture, we distinguish between two major parts of the parser – the target-agnostic and the target-specific. Fortunately, assembly languages tend to be similar so that we can offload much work to the target-agnostic part of the parser. These functionalities include but are not limited to, lexical analysis, handling most directives, and helper methods for parsing complex expressions. Furthermore, the target-agnostic part acts as the driver for the parsing process. Once the analysis encounters a target-specific construct (e.g., an instruction), the system calls into the respective target to analyze it in a target-dependent way.

At its core, the target-specific part of the assembler must be able to parse a single statement. LLVM separates this process into two steps. First, the parser transforms a statement into a vector of operands. Then, the parser uses this operand list to create a machine instruction. The grammar of the assembly description specifies the former step. The LCB generates a deterministic $LL(1)$ recursive descent parser using this specification. For each rule, the generator emits a different method. For example, the rule `Register` has a method in the parser class with the same name.

How the generator emits the body of the rule's method depends on the rule type. On the one hand, the implementation of a non-terminal rule is derived by visiting the grammar elements in the body of the rule and emitting the corresponding operational semantics as described in Section 4.3. On the other hand, each terminal rule checks whether the current token is of a particular type. This check is possible because each terminal symbol has precisely one associated `AsmToken`, i.e., a token that the lexer emits. Analogous to the semantics, the result of any rule application is the parsed value or an error. The cast to `@instruction` transforms the map holding different operands into the operand vector. Note that this vector also includes the mnemonic of the instruction, if present.

Once the parser obtains the operands, the algorithm matches the operands against the set of defined instructions $\mathcal{I}$. The goal of the matching algorithm is first to find a set $\mathcal{V} \subseteq \mathcal{I}$ that corresponds to legal instruction definitions regarding the given operands. Then, the algorithm picks a definition $v \in \mathcal{V}$ and constructs the correct in-memory data structure. The LLVM internal opcode of this `MCInst` solely depends on v. Note that more sophisticated versions may use an ordering over $\mathcal{V}$ that dictates which instruction definition the algorithm selects (e.g., choose compressed over regular version). The following paragraphs explain the solution in the context of the LCB.

The most involved of the matching process is obtaining the set of legal instruction definitions $\mathcal{V}$. Note that if $\mathcal{V} = \emptyset$, the system emits an error to the user. We use the following three predicates to determine whether an instruction definition is applicable given a vector of operands.

1. If present, the mnemonic of the instruction needs to match the mnemonic of the definition. For most architectures, this rules out the majority of candidates.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2. For every format field not assigned to a constant encoding, there has to be an operand with the respective name.
3. Given an absolute value v of an immediate operand o, its immediate encoding function e_o and decoding function d_o, the equality $d_o(e_o(v)) = v$ must hold. In other words, the value must "survive" an encoding and decoding round-trip. Furthermore, the parser checks the predicate of the operand against the decoded value.

Given the instruction definition v, the parser knows the instructions opcode and operand layout. In the internal representation, each instruction has a fixed operand order that may deviate from the assembly representation. Therefore, the system may re-order the operands according to the internal representation. Lastly, the program adjusts immediate operands as described in Section 4.4.1. This routine results in an instance of the `MCInst` class that represents the parsed machine code instruction.

The first and second requirements are very intuitive. However, the third predicate is also necessary for some architectures, as multiple instruction variants may share the same mnemonic and operand names. For example, an ISA may use an instruction with a compressed encoding if the operands only use small immediate values. As a result, the algorithm must not match the compressed instruction definition if the immediate value is too big. Similar compression schemes apply when using particular register indices. Currently, the LCB does not support such architectures. However, we see no reason why this would prove difficult. Therefore, the number of predicates will increase in future generator versions.

An essential aspect of our design is separating the specification from the actual implementation. We want to refrain from polluting our specification language with LLVM-specific constructs. Furthermore, this allows the generation of multiple generators. For example, one could compare the performance and ergonomics of a Binutils-based generator to the LCB. Furthermore, it allows us to separate the limitations of our implementation from the limitations of the specification. For example, the implementation presented in this section can only handle $LL(1)$ grammars, while other implementations may be able to handle $LL(k)$ grammars. However, using different constructs from the target system will make the interface between the generator and the target system more complex. This complexity arises from the necessity of inferring the structures necessary for the target system. For example, in LLVM, the syntax of an instruction definition is denoted by a string template. The implementation in the LCB would have been more straightforward, if VADL would require exactly this specification. However, this binds the specification language closer to LLVM, which we want to avoid.

We considered several design options for implementing the parser. One option was to leverage LLVM's parsing infrastructure based on string templates. Unfortunately, some parts of the parser are inherently target-dependent. Thus, we would still need a system for generating a (part of a) parser. Furthermore, we would then have to distinguish between constructs that the LLVM parser can handle and constructs that we have to

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

parse ourselves. This problem may have proven problematic in some contexts. Finally, altering the specification language to a more fitting representation of this problem was also not desired to avoid unnecessary coupling of VADL to LLVM. Even though this approach would have reduced the size of the parser code, we decided against it.

Another option was using a powerful out-of-the-box parser generator like ANTLR [PQ95]. This solution would have eliminated the need to develop the parser generator ourselves. Furthermore, it would have provided us with a predicated $LL(*)$ algorithm to handle more complex languages. However, integrating into a complex system is difficult, so the tool's disadvantages may outweigh its benefits. Furthermore, every external dependency incurs a long-term maintenance cost. For example, new developers need to learn another tool and some necessary features may require a costly major version update of the library. In addition, using our own parsing algorithm gives us precise control over the process. This flexibility may be required for future features. Lastly, $LL(1)$ is a simple parser implementation that can parse many common assembly language constructs. The recursive descent approach also allows designers to patch the generated LLVM code if they wish to bridge the gap to a new LCB version. While such a workaround is also possible with an ANTLR-based solution, this requires engineers to learn the basics of the parser generator. The existence of parser tables would make such an approach more cumbersome. For all these reasons, we decided against using such a system.

4.4.6 Machine Code Emitter

The machine code emitter transforms the in-memory representation of an assembly file into an object file. This work will focus on ELF object files. Section 2.2.1 already described the structure of such a file. As with most tools in the LLVM architecture, this task consists of a target-agnostic and target-dependent part. The central architecture-related issues are instruction encoding and fixup emission. A *fixup* is an LLVM concept that indicates that the tool cannot resolve the value of an operand yet. All fixups that cannot be made obsolete will become a relocation in the resulting object file. Because the LCB does not yet support any form of relaxation, it emits a relocation for all emitted fixups.

In VADL, only some instructions have an encoding definition. Analogous to the instruction printer, the code emitter may need to emit a compiler or pseudo instruction. In contrast to the printer, the code emitter must expand both types because a processor does not implement pseudo instructions. The only difference is that the code emitter keeps track of an offset during each expansion process. This value is necessary because a pseudo instruction may expand to multiple machine instructions. By using the offset, the assembler can associate fixups with instructions other than the first.

Like the disassembler discussed in Section 4.4.4, the LLVM infrastructure does much of the heavy lifting. In particular, it auto-generates code that emits the format fields of an `MCInst` at the correct ranges. Furthermore, it knows how to map constant format fields to the corresponding binary value. However, the generated code relies on a target-

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Expression Type	Fixup Type Description
SymbolRef	A symbol reference uses the generic relocation created by the default relocation injection. This case applies when symbols are used in operands without modifiers.
Target	Target-specific expressions come in two flavors. First, if the expression only represents an immediate, the system uses the fixup type from the inner expression. Otherwise, the expression itself contains the information of the fixup type.
Binary	First, the system tries to rewrite the binary expression as $a + c$ where a is a symbol, and c is a constant. If that is the case, the code emitter can emit a fixup with the normalized form. In the object file, a relocation entry with an addend of c represents this fixup. The linker emits an error if the expression cannot be rewritten to that form.

Table 4.6: Fixup Type Description for Expression Types

specific method to produce the correct value for dynamic format fields, i.e., operands. Furthermore, this method must also emit the necessary fixups if the operand's expression contains a symbol.

We start by discussing the calculation of the operand value that the assembler embeds in the resulting instruction word. This process differs depending on whether the operand represents an absolute or relocatable expression. Given an operand that is an absolute expression, the code emitter only needs to adjust the value according to the immediate rules of the LCB and embed it in the instruction word.

However, if this is not the case, the code emitter inserts a zero instead of the operand value[6]. The code emitter associates a fixup with the instruction to inform the linker of the missing symbol address. Currently, our generator can emit at most one fixup entry per instruction. We must lift this limitation once VADL supports relaxation so that the code emitter can indicate that an instruction is relaxable. Each fixup entry consists of an expression and a fixup type which indicates the type that LLVM uses for the relocation entry. Table 4.6 describes the fixup types of different expressions. Furthermore, Listing 4.21 shows three relocations in the disassembly of a relocatable object file. The first relocation is autogenerated by the LCB and is used to jump to a label. The latter two refer to the hi and lo user-defined relocations.

```
1    dc:  6f 00 00 00                         JAL  zero,0
2         000000dc:   R_CPU_SYMB_20_PCREL    .LBB1_3
3         ...
4    100:  b7 00 00 00                         LUI  ra,0x0
5         00000100:   R_RV32I_hi20_1          fibonacci
6    104:  e7 80 00 00                         JALR  ra,0(ra)
```

[6]Note that this is possible due to the usage of RELA relocation entries.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
7         00000104:   R_RV32I_lo12_1         fibonacci
```

Listing 4.21: Disassembly of RISC-V Instructions

4.4.7 Linker

Because the linker project is separate from LLVM's machine code infrastructure, the generator must implement some operations manually that are usually generated by a TableGen backend. One of the biggest ones is overriding a format field in the instruction word. Furthermore, our backend may require a decoder to infer the necessary information from the relocations. Additionally, the generator must emit files like the immediate implementations, once for the LLVM infrastructure and once for the linker.

First, if the architecture uses the conservative relocation strategy, we need a decoder to infer the missing information described in Section 4.4.2. Fortunately, VADL already has a decoder synthesizer used by the simulators and hardware schematics. In order to use this infrastructure, the generator creates a new dummy process that uses the VADL decoder built-in. Various existing passes will expand the decoder to a set of bit vector comparisons. Instead of executing the instruction similarly to the simulator, we will return an identifier for the decoded instruction. We then translate the process into a C++ procedure with the help of the object-oriented programming (OOP) infrastructure.

Furthermore, the linker also needs helper functions to patch the instruction words. In the machine code infrastructure, arranging the format fields is done by a TableGen backend. However, this code is based on different data structures and thus is unusable for this system. Conceptually, the most admirable solution would be invoking a decoder to get all format fields. This data structure could be adjusted and then re-encoded into the instruction word. However, this solution is more complex than necessary and may perform worse than the simple approach of overriding a single format field. To facilitate such operations, the LCB generates a helper class with a function for every instruction and format field pair. For example, the helper functions for the RISC-V ISA would contain a function with the name `Override_LUI_imm20`. Each function consists of only a single expression that composes multiple parts to the new instruction word. Because our generator uses `RELA` entries, we do not require functions that read the format field value. This functionality is unnecessary because the addends reside in the relocation table of the ELF file.

The target-agnostic part of LLVM covers many responsibilities of the linker. Our paramount goal is to apply the relocations to the instruction words. For every relocation entry, the driver program will invoke two methods. The first method returns the relocation type (e.g., absolute) that dictates how the relocated value should be computed. For example, the value for an absolute relocation needs to be calculated differently from a PC-relative relocation. Using this information, LLVM will call the second method with two important parameters. The new value of the relocation and the bytes containing the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

instruction word that needs patching. The procedure will then apply the transformation associated with a relocation to the given value. For example, the `hi` relocation from the RISC-V ISA will shift the value by twelve bits to the right.

Then the system, if necessary, invokes the decoder on the current instruction word. From the opcode of the instruction, the linker can infer the immediate encoding and the format field that needs adjustment. Using this information, it applies the immediate transformation to the value and uses the encoding helper class to override the format field in the byte array. Note that the program does not know the exact instruction definition it currently patches when using the performance strategy. Fortunately, the relocation implicitly dictates the immediate and the format field encoding. Thus, the linker may select any override function of a format field associated with the ELF relocation.

4.5 Testing

With the introduction of low-level developer tools in the LCB, we can leverage multiple testing techniques. We have three major types of tests. Firstly, we use unit tests that test the implementation of a single class or pass. The second type of test uses small specifications that the generator processes to an AST or VIR. We then do assertions on the intermediate representations. For example, we have multiple test specifications with different assembly printing functions that test the grammar rule inference. These two test types do not require a full-blown VADL specification usable by the LCB, thus making them essential during the first development steps.

Naturally, we need more extensive specifications for generating a compiler backend or a simulator, as a minimum amount of information is necessary. For example, generating an ISS requires a startup code definition. While the VADL team is working on producing a minimal specification that allows generating all artefacts, we resort to existing processor descriptions to test our implementation integratively. The most mature one is the RV32I processor. Integrative tests with the whole compiler toolchain represent the third and most powerful test type.

```
1  int begin_signature[] ={ 0x0, 0x0, 0x0, 0x0, 0x0, 0x0 };
2  int end_signature = 6;
3
4  typedef unsigned int uint32_t;
5  static uint32_t fibonacci( uint32_t v );
6
7  int main( void )
8  {
9      begin_signature[ 0 ] = fibonacci( 0 ); // => 0
10     begin_signature[ 1 ] = fibonacci( 1 ); // => 1
11     begin_signature[ 2 ] = fibonacci( 2 ); // => 1
12     begin_signature[ 3 ] = fibonacci( 3 ); // => 2
13     begin_signature[ 4 ] = fibonacci( 4 ); // => 3
14     begin_signature[ 5 ] = fibonacci( 5 ); // => 5
15 }
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
16
17 static uint32_t fibonacci( uint32_t n )
18 {
19     if( n <= 1 ) {
20         return n;
21     }
22     return fibonacci( n - 1 ) + fibonacci( n - 2 );
23 }
```

Listing 4.22: Example of a Test Program

Before the tests, we generate a LCB and compile the resulting LLVM target. This build gives us access to all the tools LLVM supports. We started by using these tools to make small test cases like re-printing an assembly file. Then we introduced tests that assemble and disassemble instructions to test the expansion of pseudo instructions and cover more components of the machine code layer. Lastly, we have multiple truly integrative tests in our test suite. The first type uses a C file as input and feeds it through the whole LLVM toolchain. We link the resulting object file together with two runtime files and execute this file on the simulator. We then check if the simulator generated the expected *signature*. In essence, the signature is the value of a global array at the end of the program. Listing 4.22 shows a simple test. Lastly, we use the same strategy to run a RISC-V compliance suite on multiple simulators. The primary idea is that if the simulators produced the correct output, the assembler and linker must have processed their respective inputs correctly.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 5

Evaluation

We extended the existing AArch64, MIPS IV, and RV32I (RISC-V) architectures with the language constructs described in Section 4.1 to evaluate our prototype. Furthermore, Section 5.5 discusses a VLIW prototype based on the Hexagon architectures. Before discussing the evaluation, Table 5.1 gives an overview of the specifications. Note that the number of instructions should only indicate the size and complexity of the specification. The Aarch64 specification defines specialized instructions based on properties such as the mode (32-bit or 64-bit) and extended register operations (e.g., shifting a register value before using it). This technique produces many instructions but allows for more succinct semantics per instruction definition.

5.1 Expressiveness

First, we would like to discuss the expressibility of the introduced language constructs. However, quantifying expressibility is problematic as many factors need to be considered. This evaluation uses the number of lines necessary to define an assembler and linker for a particular architecture as a proxy for expressibility. While this metric certainly has

Name	#Instructions	Description
RV32I	78	Baseline ISA primarily used for developing this prototype.
MIPS IV	106	A dollar sign precedes registers.
AArch64	1439	Relatively complex ISA with many instructions. Multiple instructions can have the same mnemonic. Pound symbol precedes immediates.
Hexagon	2	Small proof-of-concept for VLIW architectures.

Table 5.1: List of Specifications used for the Evaluation

problems (e.g., not every line has equivalent information content), it helps to get a rough idea. However, this metric has some ambiguities when it comes to counting lines. For example, VADL has a powerful macro system that allows users to reduce the amount of specification code by magnitudes. Is the original specification or the expanded concrete syntax used for counting?

To be as transparent as possible, we will divide the number of lines used into multiple categories. Not all of these categories are relevant to this work. Once we obtain the classification, we separate each type by whether it is highly relevant to low-level developer tools or not. This separation gives a rough idea of what part of the specification targets low-level developer tools. Unfortunately, clearly separating the categories is difficult as some things are essential to multiple components. For example, the encoding information is essential to the code emitter and disassembler. However, it is also paramount for the hardware design and all simulators (instruction decoding). In general, this is a positive thing as the premise of VADL, using the same information in multiple artifacts, bears fruits.

Furthermore, we will list the number of lines with and without expanded macros. We stripped comments and empty lines before counting the lines and tried to preserve formatting conventions between the hand-written and macro-expanded versions. We will use the RV32I architecture in this evaluation due to its maturity. However, note that the characteristics differ significantly between different architectures. For example, the RV32I does not use many macros. At the same time, the AArch64 specification expands to a file roughly 20 times as long as the original.

Figure 5.1 displays the number of lines per defined category. We refrain from explaining every category here. Section A.3 lists all data points and discusses them in detail. However, we would like to point out that the three categories directly associated with instructions profit from the macro system. Fortunately, this includes the assembly printing functions. In other architectures, the difference between macros and macro expanded specifications increases drastically. Leveraging this benefit for assembly definitions relies heavily on the grammar rule inference system. Suppose our approach fails to synthesize the grammar rules. In that case, each instruction created by a macro requires manually defining a corresponding production rule in the grammar, thus increasing the assembly-related specification code.

In this work, we will count alias registers, the assembly description, assembly printing functions, any used auxiliary formatting functions, instruction encoding, and relocation definitions towards the number of specification lines dedicated to defining low-level developer tools. Nevertheless, we want to emphasize that some of these constructs are essential to other generated artifacts. For example, leaving out the assembly printing functions will also impact the generation of a code generator. Figure 5.2 compares the number of lines devoted to low-level developer tools. This comparison shows that the assembly-related definitions profit disproportionately from the macro system[1]. By examining the line

[1]The macro system implementation is not part of this work.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

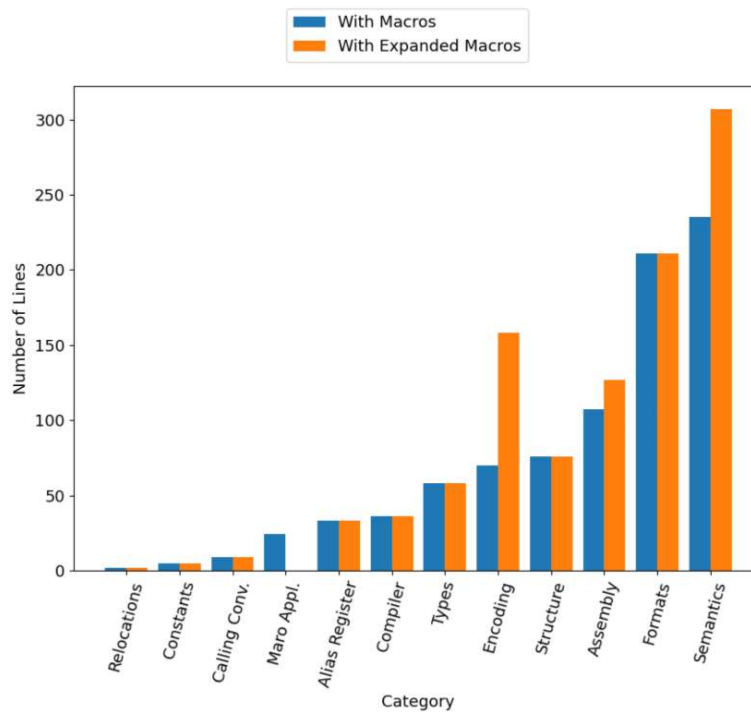

Figure 5.1: Comparison of Lines per Category

counts we conclude that the proposed language constructs are expressive enough, given a working grammar rule inference. This circumstance is highlighted by the fact that most lines are quintessential for other artifacts. Moreover, we recognize that a flawed grammar inference system seriously poses problems to the system's usability.

5.2 Tool Performance

This section evaluates the performance of the generated artifacts. Firstly, we chose the RISC-V specification to measure the tool performance. Table 5.2 contains information about the test system. The data points were acquired with the following methodology. Before measuring, we compiled LLVM once with the official RISC-V target and once with the LCB-generated one. The programs in the compliance suite act as benchmarks for this evaluation. For each benchmark, measurements indicate the time it takes the assemblers to emit the relocatable object file and the linker to emit the executable object file. To ensure the correctness of these processes, we use the same commands already in use in the continuous integration build. The tool hyperfine[2] was used to gather the data.

Before presenting the numbers, we want to recall that we only executed microbenchmarks. In practice, assembling and especially linking such small workloads is not the focus of

[2]https://github.com/sharkdp/hyperfine

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

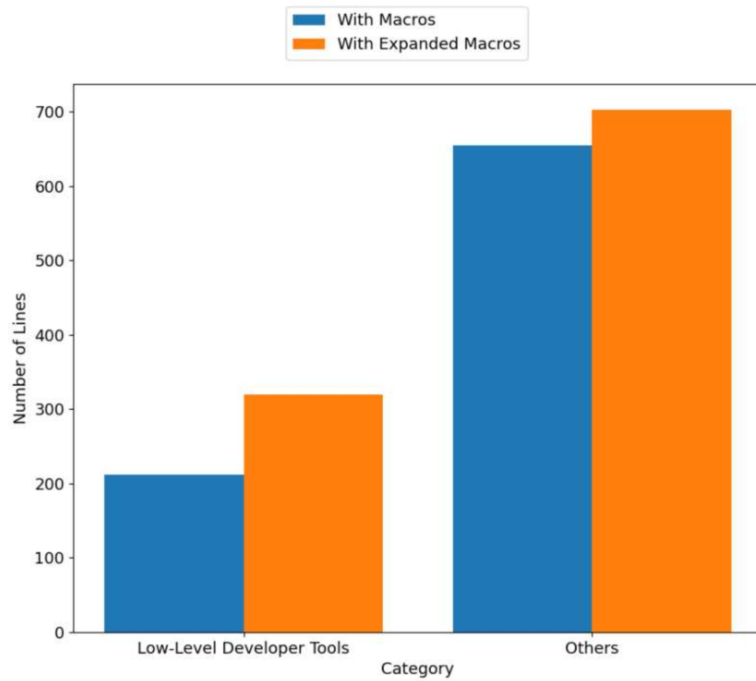

Figure 5.2: Lines devoted to Low-Level Developer Tools

Component	Name	Version
CPU	AMD Ryzen 9 3900X	-
OS	Ubuntu LTS (WSL)	22.04.1
Kernel	microsoft-standard-WSL2	5.15.68.1
Host OS	Windows 11	Build 22621.674
Compiler	g++	11.2.0
LLVM	LLVM	10.0.0
Compliance Suite	RISC-V Architecture Test SIG	2.7.4
Benchmark Tool	Hyperfine	1.12.0

Table 5.2: Test System used for Benchmarks

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

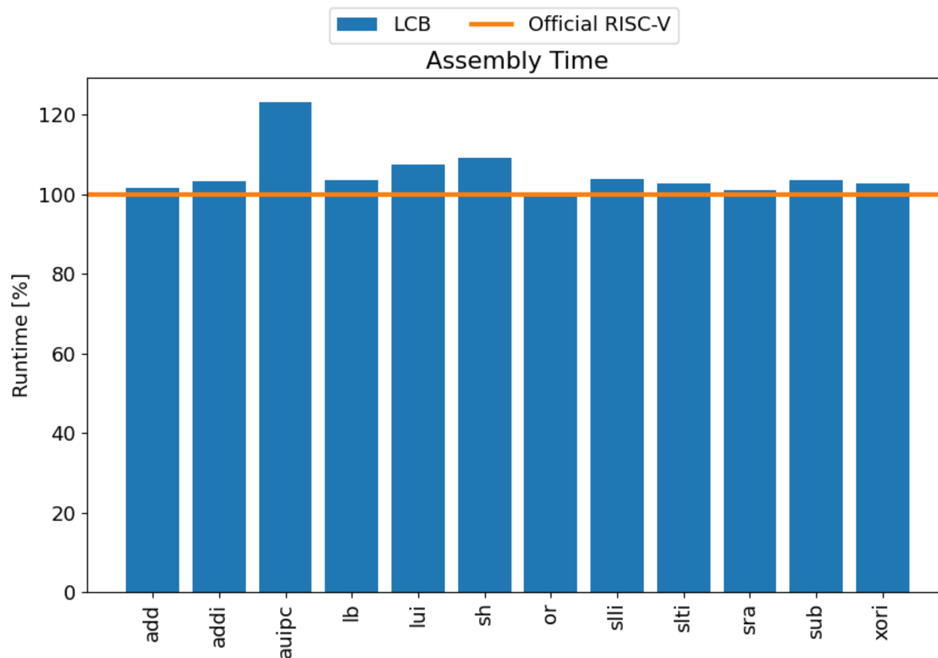

Figure 5.3: Assembly Benchmarks Relative to Official RISC-V

optimizations. As a result, we can only roughly estimate real-world performance from these benchmarks.

Figure 5.3 shows the assembly microbenchmark measurements. The horizontal axis corresponds to an assembly file that tests the compliance of a particular instruction. For example, the `add` compliance file mainly contains instances of the `ADD` instruction. The first sub-plot contains the runtimes of the LCB and the official RISC-V backend from LLVM. Our experiments showed that the LCB performs similarly to the official RISC-V backend.

Moreover, Figure 5.4 displays our measurements of the linking time. In our experiments, the LCB with performance relocations performs similarly to the official RISC-V backend. However, the conservative relocation mode performs worse in almost all benchmarks. We expected these results due to the increased overhead of decoding the instructions words prior to applying relocations. However, in our measurements, the loss in performance is still acceptable if large numbers of relocation types are necessary. Nevertheless, keep in mind that we used small test programs. Therefore, larger test files may make the gap between performance and conservative relocations more apparent.

5.3 Assembly Language Study

This section discusses the findings of the assembly language study. First, Section 5.3.1 describes the methodology and ideas behind our approach. Then, Section 5.3.2 discusses

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

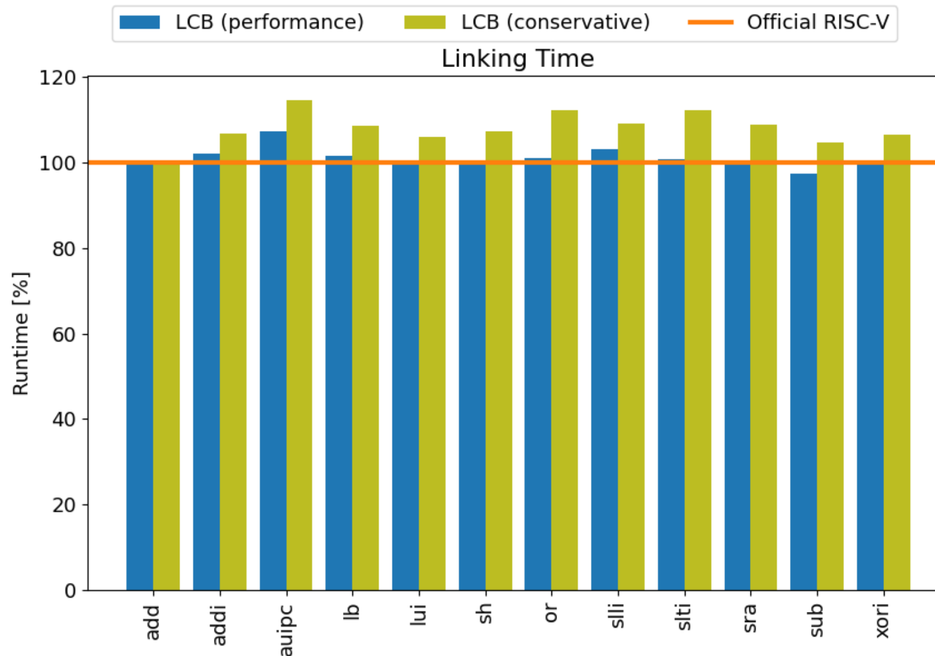

Figure 5.4: Linker Benchmarks Relative to Official RISC-V

our results. Lastly, Section 5.3.3 gives ideas for applying the findings to the implemented prototype.

5.3.1 Study Parameters

The target architectures for this study are AArch64, MIPS, RISC-V, Hexagon (VLIW), and Intel x86 (CISC). This study aims to find evidence of whether the assembly language of a given processor is in $LL(1)$ or not. A large test corpus is used as an approximation of the actual language. We assume that this corpus will have a test suite that covers most, if not all, relevant assembly constructs. As a result, if an $LL(1)$ grammar can capture the syntax of all test cases, it is likely that the actual language is also in $LL(1)$.

To be more specific, if the assembly language is in $LL(1)$, we can construct an $LL(1)$ grammar that will 1) recognize all valid test cases and 2) reject all syntactically invalid test cases. The distinction between syntactically and semantically invalid is crucial. In this study, the syntax only describes the structure of an assembly string, not whether an operand of a particular instruction should be a register or an immediate. Because most of the invalid tests of assembler test suites check semantic properties (e.g., immediate size), this study will omit them. We acknowledge that this approach may lead to us describing a different dialect of the assembly language. However, we took great care during the grammar construction to not overspecialize our grammar to the test set.

It should be mentioned that this study will operate in a slightly different setting as

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Architecture	# tests	# lines	# pos	% pos.
RISC-V	82	2089	82	100
MIPS	244	9051	244	100
AArch64	211	12643	209	99.05
Hexagon	81	1795	81	100
Intel x86	24	10799	23	95.83

Table 5.3: Results of Test Suites

the LCB parser. For example, the grammar in the study needs to handle labels, while LLVM's target-agnostic part handles this part for the LCB. By doing so, we decouple this study from LLVM's infrastructure. Even though, as a result, our study does not directly apply to our prototype, we consider this a more valuable contribution. The following list presents aspects different from the requirements from the LCB.

- The parsers in the study need to derive the whole assembly file, not only instructions. As a result, they must cope with all directives and labels.
- The grammars will not specify the operand names.
- The grammars will not specify the operand types.

This study uses version 3 of ANTLR to generate the parser and lexer for the languages. This version allows users to define maximum lookahead values for the parser. If a decision requires inspecting more tokens, a warning is shown to the user. By setting the maximum lookahead to one and ensuring the tool does not emit warnings, we constructed our $LL(1)$ grammars.

5.3.2 Results

The test corpus for this study was sourced from LLVM's official machine code layer test suite. Each test corresponds to a single assembly file. Note that some of them may contain hundreds of valid instructions. Because every file only consists of valid assembly strings, a parser describing the assembly language must be able to derive all tests. The test runner will mark a test case as invalid if the parser cannot derive the whole assembly file.

Table 5.3 presents the study's results. By comparing the total number of tests (# tests) against the number of tests that succeeded with the developed $LL(1)$ grammar (# pos), we can get an idea of how limited the grammar is in the context of a particular assembly language. The number of lines in the test cases (# lines) indicates the size of the test suite. Note that comments and empty lines have been removed.

Firstly, the limitations in the AArch64 test are due to the usage of macros. Our lexer and parser do not have a functioning macro system. Therefore, these failing tests are

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

not due to the $LL(1)$ restriction. Furthermore, the failing test case in the Intel x86 test suite is due to a limitation of the lexer. The assembly syntax allows defining hexadecimal numbers by appending a “H” to the end of the number (e.g., “123H”). This introduced a problem because the lexer expected the label reference “1f” to be a hexadecimal number. However, because no “H” followed this number, the lexer emitted an error. Therefore, the failing Intel x86 test is also not due to the $LL(1)$. We were able to derive all test files in the RISC-V, MIPS, and Hexagon test suites.

```
1 lw $31, ($29)
2 lw $31, (8 * 4) ^ (8 * 31)($29)
```

Listing 5.1: Two Variants of the MIPS LW Instruction

We found that most constructs in standard assembly languages can be parsed with $LL(1)$. However, sometimes this requires applying left-factoring, which results in complex rules. For example, Listing 5.1 shows two valid assembly strings of the LW instruction in the MIPS architecture. This construct allows offsetting the value of the register by an expression. The first grammar rule in Listing 5.2 is an intuitive grammar to this problem. Unfortunately, an expression may also start with a parenthesis. Once an $LL(1)$ parser encounters the first parenthesis, it cannot decide if it should accept this as part of an expression or as the parenthesis surrounding a register. Such a decision is only possible by looking at the following token and checking whether this is a register. To reduce the necessary lookahead, one can left-factor the grammar rule by hoisting the conflicting productions. Unfortunately, this results in complex rules, as this example shows.

```
1  # LL(2)
2  regoffset:
3          '(' ireg ')'
4      |   expression '(' ireg ')'
5      ;
6
7  # LL(1)
8  regoffset:
9          '(' (
10             ireg ')' |
11             expression ')' (binop binary)? '(' ireg ')'
12         )
13     |   modified (binop binary)? '(' ireg ')'
14     |   symbolref (binop binary)? '(' ireg ')'
15     |   unaryop? NUMBER (binop binary)? '(' ireg ')'
16     ;
```

Listing 5.2: ANTLR Grammar Snippet for LW Instruction

Another result from our study is that in standard assembly languages most problematic constructs arise from abbreviated syntaxes. In the last example, this is the optional

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

expression preceding the register value. As a result, a $LL(1)$ grammar can model most assembly instructions without complex rules if the abbreviated syntax is omitted. Similar problems arise with the optional "#" in the AArch64 assembly syntax. However, note that the example with the register offset is not due to an abbreviated syntax.

5.3.3 Applying the Results to the LCB

As already mentioned, the study does not represent the exact use case of the LCB. The parser in our prototype currently has problems with abbreviated syntaxes. However, this study shows that $LL(1)$ grammars can model many abbreviated assembly strings. This section discusses why the LCB does not support such constructs and how the results of this study can be applied to the assembly parser prototype.

Recall that LLVM divides the assembly parsing process into two steps: parsing and matching. In the current version of the LCB, the parsing step is responsible for assigning names to the parsed operands. Unfortunately, this makes it impossible to parse some abbreviated syntaxes with $LL(1)$, assuming operand renaming is not possible. Listing 5.3 shows two valid versions of the RISC-V `JALR` instruction. This instruction consists of two registers (`rd` and `rs1`) and one immediate, which is omitted here. There is no way for an $LL(1)$ parser positioned after the mnemonic to know which name it should assign to the first register. This is because the first operand may correspond to `rd` or `rs1`, depending on whether another register follows.

```
1 jalr x2
2 jalr ra, x2
```

Listing 5.3: Two Variants of the RISC-V JALR Instruction

However, by postponing the operand naming to the matching phase, the LCB could parse these constructs with an $LL(1)$ parser. The sole responsibility of the parsing phase would be to produce an operand vector without any metainformation, i.e., operand names. The generator can compute all possible operand sequences based on the assembly printing function. For example, in the case of the `JALR` instruction, there is one version with one register operand and one with two register operands. This information is then used in the matching phase to determine the operand names. The parser then assigns default values to all omitted operands. This would allow the parser to use an $LL(1)$ grammar while still allowing abbreviated syntaxes. If this approach is better suited for the problem at hand requires further investigation.

To summarize, while most assembly constructs can be modeled with $LL(1)$, we conclude that a user-facing specification should not burden a user with left-factoring the grammar manually. The main reason is the unintuitive rule structures resulting from the hoisting process. This problem can be solved with automatic grammar rewriting or using a more powerful parsing algorithm. However, because this limitation is only relevant to our

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Name	#Insts	#Inverted	Percent	# Rules
RV32I	74	74	100	75
MIPS IV	106	104	98.11	105
Aarch64	1439	1437	99.86	1446

Table 5.4: Number of Generated Grammar Rules

prototype and not the VADL specification, we defer this implementation to future work on the LCB.

5.4 Grammar Rule Generation

We evaluate the grammar rule generation based on the Aarch64, MIPS IV, and RV32I specifications. All instructions that have an assembly representation constitute our test set. In contrast to the table at the beginning of this chapter, this does not include compiler instructions. The grammar rule generation process is tasked with finding inverse rules for the test set. In this step, the system throws an exception if it cannot infer a grammar rule from an assembly printer. Using this fact, one can count how many rules the generator could successfully invert. Furthermore, our system counts the number of emitted grammar rules. This figure may be higher than the number of instructions due to generated helper non-terminals. We already validated the rules generated from the RV32I specification by using them in our test suite. To provide some quality assurance for other architectures, the generated grammar symbols are sampled and checked manually.

As expected, our approach inverted all instructions in the RV32I specification. Furthermore, it performs very well on the MIPS IV architecture. We inspected the generated rules and found that they are sensible. Handling the dollar prefix for registers proved to be unproblematic. However, our algorithm could not infer the slice information for two rules – `syscall` and `ebreak`. The problem is that they have an abbreviated syntax that omits the `code` field if it is zero, thus making use of a multiplexer instruction. However, this format field is 20-bits wide. Therefore, our approach needs to test $2^{20} = 1048576$ different values, which causes an out-of-memory exception. Fortunately, we can manually specify the rules for these two instructions to prevent the rule inference for the problematic instructions. Another alternative would be removing the abbreviated syntax in the printing function. Then all constructs have a native slice semantic, thus avoiding constant evaluation.

Lastly, we also tested the approach on the AArch64 specification. While most assembly printing functions were unproblematic, the system could not cope with two instruction definitions. The two errors originate from the inability to evaluate certain casting expressions. One could solve this problem by extending the constant evaluator. Furthermore, some problems arise from the specialized instructions described at the beginning of this chapter. Firstly, because many instructions have the same mnemonic, the generated grammar is not $LL(1)$ (without left-factoring). For example, this specification has 152

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

different versions of the ADDS instruction. In the alternative consisting of all instructions, these rules have overlapping FIRST sets. Furthermore, the matching process needs to know which options an instruction has (e.g., is a value shifted before adding). This information is currently not captured by the grammar. Therefore, the matcher cannot know which concrete instruction definition should be assigned to the operand vector.

Still, the high percentage of invertible assembly printing functions gives us confidence in the chosen architecture. Even though the concept of our rule generation is simple, it can generate parser rules for large parts of the test set. Still, designers must pay attention during their specification work to only use constructs supported by our system. For example, our implementation cannot handle a string that maps to multiple tokens (e.g., a comma followed by a parenthesis). Users are required to separate these tokens into two strings manually. During this evaluation, such constructs in the MIPS IV and AArch64 architectures had to be patched. However, we see no reason that a solution based on lexical analysis can address this problem. Such a system would also be beneficial for analyzing string constants provided by the constant evaluator. Furthermore, we found that the system that re-uses already generated non-terminals works well because the number of generated rules is only slightly higher than the number of instruction definitions.

Lastly, we would like to discuss how much computational power the inference process requires. This time is divided between the two most CPU-intensive passes – inlining auxiliary functions and generating the rules. All other introduced passes are negligible. We use the system from Section 5.2 to perform our measurements. Table 5.5 shows the acquired data split between inlining (Inl) and generating (Gen). Our solution performs reasonably well for MIPS IV and AArch64.

However, for the RISC-V architecture, the two passes consume almost a third of the execution time. According to our analysis, this is due to the large function that formats the CSRs. Our measurements showed that generating the corresponding alternative consumed more than 90% of the pass's runtime. Further analysis has to be done to pinpoint the problem. After that, we can improve the problem by, for example, implementing a better constant evaluator or using a more sophisticated approach. We also think the relative difference between the AArch64 and MIPS IV architecture is due to calling auxiliary formatting functions. Because the inlining phase was implemented prior to this work, we do not know of any improvements in the assembly inlining pass. Note that some pipeline parts that crashed were disabled while evaluating the AArch64 and MIPS IV specifications. Thus, in a fully functional pipeline, the relative runtime is slightly lower.

5.5 Flexibility

We evaluate the flexibility of our solution based on two experiments that add experimental low-level developer tools for two architectures – MIPS IV and the Hexagon architecture. MIPS IV already has an existing rather complex specification. We extend this specification with the necessary elements to describe the assembly. Then we will investigate which

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Name	Total (ms)	Inl (ms)	Inl (%)	Gen (ms)	Gen (%)
RV32I	5162	633	12.26	1637	31.71
MIPS IV	3160	4	0.013	141	4.46
AArch64	30137	1599	5.31	434	1.44

Table 5.5: Performance of Grammar Rule Generation

features of the LCB are missing and how many lines of specification code are necessary to support the new architecture. Furthermore, a new specification for the Hexagon architecture was created to evaluate the burden of adding a new concept – VLIW syntax support. We also wanted to conduct this experiment with the AArch64 architecture. However, the problems discussed in Section 5.4 prevented such an undertaking.

This section first discusses extending one existing architecture and then introducing a new concept. However, keep in mind that all architectures are highly experimental. The LCB has yet to emit a functioning code generator for these architectures. Fortunately, the VADL team pushed towards an architecture that allows a partial generation of the LCB. While it is still necessary to manually patch some files, this capability enables us to rapidly create a prototype without fulfilling all the code generator requirements. In our case, our generator emits an assembler and linker while omitting the compiler altogether. As a side-effect of the development stage, these architectures do not have a rigorous test suite comparable to the RV32I architecture.

The goal of every experiment is to parse an assembly file and then pretty-print the internal representation. This task represents a basic test case for the assembly parser and printer. Section A.5 lists the test programs. Unfortunately, at the time of this writing, we cannot test the code emitter and the disassembler because resolving the encoding does not work for these architectures. The VADL team hopes to support a full-fledged low-level developer tool suite for these architectures soon.

Firstly, the LCB was used to generate a MIPS IV assembler. This program was used to ingest and print a hello world assembly program. Listing A.8 depicts this program. Fortunately, the grammar rule generation works well for this architecture, thus significantly reducing the amount of work necessary. In summary, we had to add an alias directive for `.asciiz`, define a minimal subset of the ABI (register aliases), and provide the two overrides discussed in Section 4.2. Additionally, we had to fix some bugs in the generator's pipeline. Overall, the experiment was carried out within a few hours.

The second experiment adds limited VLIW support to the assembly parser and printer. In particular, we will define two instructions of the Hexagon architecture. In our prototype, all instructions enclosed within braces will be put into one bundle. Furthermore, the parser will wrap instructions not enclosed in braces into a bundle with a single instruction. Note that our implementation will not validate the emitted bundles. Supporting bundle validation would require two significant upgrades to the generator. First, the VIR must be able to model the whole operation and group semantics which are used in VADL to

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

model VLIWs. Secondly, the LCB must transform the bundle constraints into executable code. This component is currently in development because the generated simulators require a similar system.

Our experiment targets the small code snippet presented in Listing A.9. It only consists of two instructions. Firstly, a move that sign-extends the 16-bit immediate to 32-bits and saves it in the destination register. Secondly, the specification contains an `ADD` instruction that adds two registers together and saves the result in a third one.

We thought of three different language designs for this prototype. The first one models the bundle syntax with a built-in that the designer uses in the grammar's start symbol. Parameterizing the construct with, for example, the characters that start a bundle provides additional flexibility. As an alternative, the second approach captures this information in attributes, not in the grammar definition. The third design manifests the syntax purely in terms of grammar elements. Additional types would denote bundles and bundling operators. Furthermore, the grammar would need to support recursion or ϵ-productions. The third approach is the most complex, while the other two have similar complexity. In this prototype, we decided to implement the second idea because the complexity of the third approach outweighs its advantages. The newly introduced attribute is called `vliwSyntax`. One of its values describes the syntax of the Hexagon ISA. In a production-ready design, the language could separate this annotation into multiple ones, maybe even a dedicated language element.

Extending the LCB for this prototype took around 300 lines of code. First, the generator now emits a member that holds a state over multiple statements in the assembly parser. This state variable then indicates whether the current statement is enclosed in braces. If yes, the parser adds the instruction to the current bundle. If not, a new bundle specifically for this instruction is created and emitted. Note that the bundle is a regular `MCInst` with a particular opcode and sub-instructions as parameters. Furthermore, we added a similar mechanism to the assembly printer generator. When the printer encounters an instruction with this opcode, it emits all instructions inside the bundle enclosed in braces.

However, we encountered a problem during this experiment. Hexagon instructions have a peculiar format compared to other assembly languages. Instead of starting with the mnemonic, the two specified instructions start with the destination register (e.g., `R1=#1`). The syntax is similar to assigning a variable in general-purpose programming languages. Unfortunately, the combination of the $LL(1)$ restriction and the type restrictions on alternatives hinders us from specifying this grammar in the parser. In our experiment, we introduced an identifier in the front of both instructions so that the parser can choose the correct instruction with the first consumed token. Listing A.10 shows our adjusted version. We successfully conducted our experiment with the modified syntax.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 6

Future Work

While the assembler performance discussed in Chapter 5 is comparable to the official backends, there is still potential for improvement. For example, matching a register to a register number is still done with a chain of conditional statements making separate string comparisons. In addition, global grammar analysis could also yield considerable performance improvements. To elaborate, we currently cast the mnemonic to an operand and assign it to a field in a struct. This field is then read and written into a larger struct containing all operands of an instruction. The parser then appends all values in this struct to the operand vector. While the C++ compiler may remove some intermediate steps, removing these steps from the source code could significantly increase the assembly performance. Other approaches could integrate the parsing and matching phase to prevent duplicate analysis.

The LCB emits the target-specific code in the necessary LLVM files. To do this, we crafted code generators that emit the code structure required by the framework while incorporating the information from the VADL specification. As a result, we have copied large parts of the target-specific code base and enriched them with templating functionality. While this is a viable implementation strategy, it creates a strong coupling between the LCB and the used LLVM version. Therefore, upgrading to a new major version of LLVM may be laborious. One idea to mitigate this issue is to bundle all the code influenced by the specification in a single point. The new generated LLVM target then only calls into this wrapper when target information is necessary. This architecture would decouple the changes to LLVM from the C++ we generate based on the specification. Therefore, upgrading would mainly require changes to the LLVM side, which can be handled well with modern version control systems. Furthermore, because the LCB uses LLVM 10, such a refactoring could enable a straightforward upgrade to a newer version.

The current grammar generation works well when the assembly printing functions follow some guidelines. For example, printing the string "`, (`" needs to be done with two literals because this string consists of two lexer tokens. Unfortunately, as of this writing, the LCB

does not support splitting the tokens automatically. This limitation also holds for string constants acquired by the constant evaluation. Developing a solution to this problem in the grammar rule generation allows the user to write more concise specifications.

One of the most critical future improvements is lifting some restrictions on the grammar specification or the implementation itself. For example, while the evaluation shows that a $LL(1)$ grammar can define many common assembly constructs, it also shows its limitations – especially when we need to assign values to parameters. We think that extending our approach to $LL(k)$ or $LL(*)$ will lift many of the limitations. However, this also requires implementing recursive rules to model the additional lookahead required for parsing expressions. Fortunately, some solutions will work with the existing type system by only allowing recursive rules when the type does not change with each recursion.

An implementation with linker relaxation could improve the real-world performance of the generated binaries. For example, in the RV32I specification, we could speed up function invocations in the vicinity of the PC or global pointers. Furthermore, the LCB would need to support some options to enable and disable relaxation. Generating relaxation candidates by analyzing the instruction semantics is another interesting research question.

While we took care to cover all functionality with tests, our current test suite can still be extended. Unfortunately, at this point, the VADL specifications need to be more mature to compile many programs (e.g., missing floating point support). Nevertheless, once this circumstance changes, larger programs should be incorporated into the continuous integration process of the generators. Furthermore, we could also leverage advanced techniques such as input program generation to gain more test coverage. Lastly, we may adopt the test suites of other assemblers (e.g., LLVM's official RISC-V backend) to our infrastructure.

Naturally, future work may extend the set of specified processors. Similarly to the AArch64 and MIPS IV specifications, we may encounter some aspects that VADL cannot model or that the LCB still needs to implement. However, with every new addition, the language's feature set will become more powerful. Especially, successfully generating low-level developer tools for the complex AArch64 specification would be a milestone.

Lastly, future work could evaluate fundamentally different design choices. For example, it may prove beneficial to compute assembly operand sequences from the assembly printing instructions as described in Section 5.3.3. Such a system could remove the necessity of assigning operands to format fields by moving more logic to the matching phase of the parser. While the slice semantics approach is successful, a more sophisticated approach may improve the fallback semantics.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 7

Conclusions

In this work, we extended the VADL to support the generation of low-level developer tools. Furthermore, the LCB, a generator for an LLVM-based compiler backend, was extended to support the new language constructs. During the development process, new features were continuously incorporated into the RV32I specification. Our prototype assembled and linked all programs in an extensive compliance suite.

One of the main challenges was parsing programs for various assembly languages. We presented a specification allowing designers to describe an architecture's assembly language succinctly. Furthermore, we implemented a prototype parser generator for this description. While the prototype's performance was comparable to a handwritten LLVM backend, some limitations exist. In particular, restricting the input grammar to $LL(1)$ introduces problems with assigning names to operands during the parsing process. However, we give some ideas on extending the prototype to mitigate this issue.

To reduce the specification burden of users, an automatic grammar rule inference system was devised. Based on the assembly printing functions, this component effectively describes their inverses by emitting corresponding grammar rules. We evaluated our approach in three architectures. The algorithm was able to generate grammar rules for the vast majority of formatting functions.

Furthermore, two strategies for automatically generating relocations were proposed by us. One approach tries to minimize the amount of generated relocations while the other tries to improve the linking performance. We validated these expectations in an experiment.

Lastly, we conducted a study investigating to what extent $LL(1)$ suffices to parse assembly languages. It was found that this restriction can model most aspects of assembly languages. However, some abbreviated instruction versions require complex production rules. This is not necessary for more powerful parsing algorithms, thus making them more user-friendly.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

APPENDIX A

Appendix

A.1 VADL Intermediate Representation - Further Information

This section introduces selected VIR instructions that were explicitly mentioned in this work. Furthermore, we will give a short explanation of some facts stated in this thesis.

The `AsmInstruction` can format registers to strings. For this purpose, it uses the register definitions and any existing alias registers. The architectural register (file) is inferred from the instruction semantics.

The `CastInstruction` models VADL's complex casting semantics. For example, this concept can model all sign and zero extensions. Furthermore, it models casts from integers to string values which is essential to this work. In other words, we model the number formatting function (e.g., decimal) with this instruction.

The `ChoiceInstruction` implements the biased choice operator discussed in Section 4.1.1.

The `ConcatInstruction` is responsible for handling all sorts of concatenations. Initially, it was introduced to concatenate bit vectors. In this work, we extended the instruction to handle string concatenations.

The `ConstInstruction` holds a reference to a constant value. The idea is that this instruction allows modelling the value of a constant in the same way as any other value. Recall that the VIR stores operands as references to other instructions.

The `MuxInstruction` has three critical parts. A list of choices, a list of values, and a selector. There is a one-to-one mapping between choices and values. The instruction compares the selector's value to the list of choices. Once it finds a match, the multiplexer instruction obtains its value corresponding to the selected choice. There is also an optional fallback value. Note that, in actual hardware, all choices are evaluated simultaneously.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The ParameterDefinition holds the definition of a parameter, including its type.

The ProbeInstruction is another concept inspired by hardware components. The idea is that this is an explicit command to use the value of an operand at a given time. For example, we may have a process that takes two cycles. When probing the value in the first and the second cycle, both values can be used in the second cycle. This is comparable to saving the old value of a value in a helper variable.

The RetInstruction defines the return value of a function.

Assembly printing functions do only have one return statement: This is true because the function body only consists of a single expression. The IR models conditional expressions such as “match” and “if” with multiplexer instructions (MuxInstruction). Instead of branching to another control flow, we compute all possible values and select the correct value at the end. The VADL team chose this architecture because the representation is suited for generating hardware components. Listing A.1 shows the VIR code representing the WRegSP function displayed in Listing 4.10.

```
1  function @A64.WRegSP.1 (b5 %x) -> <0 x b8> = {
2    lbl %bb39:
3      %751 = const <1 x b8> "w"
4      %752 = probe b5 %x
5      %753 = cast.dec <0 x b8> %752
6      %754 = add <0 x b8> %751, %753
7      %755 = const <3 x b8> "wsp"
8      %756 = cast <0 x b8> %755
9      %757 = probe b5 %x
10     %758 = const u5 31           ;; '0x1f', '0b11111'
11     %759 = equ b %757, %758
12     %760 = const b 0            ;; '0x0', '0b0'
13     %761 = const b 1            ;; '0x1', '0b1'
14     %762 = mux <0 x b8> %759
15            , [b %761, <0 x b8> %756]
16            , [b %760, <0 x b8> %754]
17     ret <0 x b8> %762
18 }
```

Listing A.1: Internal VIR Code for the WRegSP Function

A.2 Parser Semantics Data Structures

Listing A.2 shows the data structures that are used for defining the parser semantics. Each element of the IR has a corresponding alternative in the GrammarElement sum type. The DataType describes the types of a VADL grammar. This data type is used in casts. The Data type describes the actual data that is gathered by the parser. We have to denote a cast with a special constructor as the cast semantics are implementation-specific. The Res data type either describes a piece of data in conjunction with the remaining

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

tokens, or an error message. Lastly, the `Token` type associates a data element with a token type.

```
1  data GrammarElement =
2    NonterminalRule String GrammarElement |
3    Sequence [GrammarElement] |
4    Alternative [GrammarElement] |
5    Bind String GrammarElement |
6    Literal String |
7    Cast DataType GrammarElement |
8    Transform (Data -> Data) GrammarElement |
9    Producer (() -> Data) |
10   Builtin String
11
12 data DataType = TStr | TInt | TReg | TMod | TKeys | TVoid| TInst
13   | TExpr | TSymb
14
15 data Data =
16   DStr String |
17   DKeys [(String, Data)] |
18   Void |
19   DCast DataType Data
20
21 data Res =
22   Ok Data [ Token ] |
23   Err String
24
25 type Token = (String, Data)
```

Listing A.2: Data Structure for Parser Semantics

A.3 Evaluation Data

This section gives further insights into the data collected during this work. In particular, it contains all measurements regarding expressibility and performance evaluation.

Table A.1 shows the line counts of the expressiveness evaluation with macros (# w/ M.) and with expanded macros (# w/o M.). Furthermore, each category is assigned to a meta category that indicates whether we count these lines towards the number of lines necessary to define low-level developer tools. Furthermore, the following list shortly describes the concepts that were counted towards a particular category.

- **Relocations**: Only relocation definitions. Modifier Mapping is counted towards the assembly category.
- **Constants**: Definition of constants, for example, word length.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Category	Meta Category	# w/ M.	# w/o M.
Relocations	Low-Level Dev.	2	2
Constants	Other	5	5
Calling Conventions	Other	9	9
Macro Applications	Other	24	0
Alias Register	Low-Level Dev.	33	33
Compiler	Other	36	36
Types	Other	58	58
Encoding	Low-Level Dev.	70	158
Structure	Other	76	76
Assembly	Low-Level Dev.	107	127
Formats	Other	211	211
Semantics	Other	235	307

Table A.1: Full Line Counts

- **Calling Conventions**: Defining special purpose registers (e.g., stack pointer) and sets of callee/caller-saved registers.
- **Macro Applications**: Applications of macros.
- **Alias Register**: Aliases for registers. We count this to the assembly meta category because most assembly code uses aliases instead of canonical names.
- **Compiler**: Contains sequence definition and custom LLVM patterns.
- **Types**: Type aliases and enumeration definitions.
- **Encoding**: Instruction encodings.
- **Structure**: Hardware structure (e.g., microarchitecture).
- **Assembly**: Assembly printing function and assembly definition.
- **Formats**: Formats for instructions and exceptions.
- **Semantics**: Instruction semantics.

Table A.2 shows the mean assembly duration for all assembler benchmarks. Table A.3 shows the mean linking duration for all benchmarks. An entry with "-" means that the program could not assemble or link the benchmark.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

Benchmark	Official	LCB
add-01.s	14.61	14.85
addi-01.s	12.61	13.04
and-01.s	14.43	14.65
andi-01.s	12.69	13.1
auipc-01.s	2.7	3.32
beq-01.s	–	66.96
bge-01.s	–	67.58
bgeu-01.s	–	84.86
blt-01.s	–	69.94
bltu-01.s	–	84.6
bne-01.s	–	70.35
fence-01.s	1.65	1.72
jal-01.s	–	234.92
jalr-01.s	–	3.43
lb-align-01.s	2.28	2.36
lbu-align-01.s	2.2	2.49
lh-align-01.s	2.21	2.32
lhu-align-01.s	2.23	2.33
lui-01.s	2.35	2.53
lw-align-01.s	2.19	2.3
or-01.s	14.89	14.85
ori-01.s	12.72	13.06
sb-align-01.s	2.44	2.69
sh-align-01.s	2.49	2.72
sll-01.s	3.64	3.84
slli-01.s	3.48	3.61
slt-01.s	14.73	14.76
slti-01.s	12.69	13.04
sltiu-01.s	15.53	15.57
sltu-01.s	17.59	17.66
sra-01.s	3.72	3.76
srai-01.s	3.43	3.59
srl-01.s	3.66	3.94
srli-01.s	3.42	3.7
sub-01.s	14.51	15.03
sw-align-01.s	2.41	2.67
xor-01.s	14.4	14.91
xori-01.s	12.92	13.27

Table A.2: Assembly Performance Data in ms

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Benchmark	Official	LCB perf.	LCB cons.
add-01.s	7.19	7.23	7.2
addi-01.s	7.22	7.37	7.7
and-01.s	7.19	7.14	7.6
andi-01.s	7.23	7.47	7.78
auipc-01.s	6.88	7.39	7.89
beq-01.s	–	8.27	10.63
bge-01.s	–	8.32	9.92
bgeu-01.s	–	8.36	9.85
blt-01.s	–	8.32	9.43
bltu-01.s	–	8.09	9.82
bne-01.s	–	7.92	9.37
fence-01.s	6.91	6.87	7.57
jal-01.s	–	8.48	8.86
jalr-01.s	–	7.02	7.4
lb-align-01.s	6.88	6.98	7.46
lbu-align-01.s	6.81	6.92	7.32
lh-align-01.s	7.0	6.9	7.33
lhu-align-01.s	6.87	6.92	7.41
lui-01.s	6.92	6.91	7.34
lw-align-01.s	6.98	6.96	7.38
or-01.s	7.04	7.12	7.9
ori-01.s	7.19	7.12	7.59
sb-align-01.s	7.05	7.0	7.46
sh-align-01.s	7.04	7.02	7.56
sll-01.s	6.99	6.96	7.31
slli-01.s	6.84	7.06	7.47
slt-01.s	7.26	7.1	7.61
slti-01.s	7.09	7.14	7.96
sltiu-01.s	7.16	7.22	8.16
sltu-01.s	7.21	7.34	7.85
sra-01.s	7.04	7.06	7.67
srai-01.s	6.97	6.95	7.54
srl-01.s	6.97	7.09	7.72
srli-01.s	7.02	6.94	7.4
sub-01.s	7.28	7.09	7.62
sw-align-01.s	7.49	7.02	7.43
xor-01.s	7.96	7.11	7.62
xori-01.s	7.16	7.16	7.63

Table A.3: Linking Performance Data in ms

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

A.4 Study Grammars

This section lists the grammars used for parsing the test suites. Note that some of these languages use a separate token type for registers (e.g., RISC-V), while others use identifiers for registers and symbols. This was done to show that both versions are viable. However, we found writing grammars not separating these concepts easier. Note that these specifications were designed to simply match all inputs. We did not put much effort into elegantly abstracting different concepts of the language.

```
1  statements: (statement)*   ;
2
3  statement
4    : NUMBER ':' EOS? |
5      IDENTIFIER (
6        (operand? (',' operand)*) EOS |
7        ':' EOS?
8      )
9    ;
10
11 operand:
12     reg
13   |      '(' (
14       reg ')' |
15       expression ')' (binop binary)? ('(' reg ')')?
16     )
17   | modified (binop binary)? ('(' reg ')')?
18   | symbol (binop binary)? ('(' reg ')')?
19   |   unaryop expression ('(' reg ')')?
20   | NUMBER ('b' | 'f')? (binop binary)? ('(' reg ')')?
21   |   type
22   ;
23
24 unaryop: ('~' | '!' | '-' | '+') ;
25 binop: ('<<' | '+' | '-' | '>>'| '*' | '|' | '||' | '&' | '&&'
26          | '^' | '/' | '%') ;
27 symbol: IDENTIFIER ('@' IDENTIFIER)? ;
28 type: '@' IDENTIFIER ;
29
30 expression: binary;
31 binary: unary (binop binary)? ;
32 unary: unaryop? term ;
33 term:  '(' expression ')'
34   | NUMBER ('b' | 'f')?
35   | symbol
36   | modified
37   ;
38 modified : '%' IDENTIFIER '(' expression ')' ;
39 reg: REG ;
```

Listing A.3: ANTLR3 Grammar for RISC-V Test Set

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
statements: (statement)*  ;

statement
  : NUMBER ':' |
    (IDENTIFIER | 'b' | 'f') (
      (operand? (',' operand)*) EOS |
      ':' |
      '=' expression EOS
    ) |
    EOS
  ;

operand:
    reg (
        ('(' reg ')')
      | '-' reg
      |   '[' (expression | reg) ']'
    )?
  |     '(' (
      reg ')' |
      expression ')' (binop binary)? ('(' reg ')')?
    )
  | modified (binop binary)? ('(' reg ')')?
  | symbol (binop binary)? ('(' reg ')')?
  |   unaryop expression ('(' reg ')')?
  | NUMBER ('b' | 'f')? (binop binary)? ('(' reg ')')?
  |   ('b' | 'f')
  |   type
  |   STRING
  |     FLOAT
  |   ('at' | 'fp' | 'arch') ('=' (IDENTIFIER PLUS? | reg | NUMBER))?
  ;

unaryop: ('~' | '!' | '-' | '+') ;
binop: ('<<' | '+' | '-' | '>>'| '*' | '|' | '||' | '&' | '&&' | '^'
        | '/' | '%') ;
symbol: IDENTIFIER ('@' IDENTIFIER)? ;
type: '@' IDENTIFIER ;

expression: binary;
binary: unary (binop binary)? ;
unary: unaryop? term ;
term:  '(' expression ')'
  | NUMBER ('b' | 'f')?
  | ('b' | 'f')
  | symbol
  | modified
  | FLOAT
  ;
modified : '%' IDENTIFIER '(' (expression | reg) ')' ;
reg: INTREG | WREG | FLOATREG ;
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Listing A.4: ANTLR3 Grammar for MIPS Test Set

```
statements: (statement)*  ;
statement
  : NUMBER ':' |
    '.loh' (IDENTIFIER | NUMBER) (operand? (',' operand)*) EOS |
    (IDENTIFIER  | immmodifier | 'b' | 'f') (
      ':' |
      (fstOperand? (',' operand)*) EOS
    ) |
    EOS
  ;
fstOperand: REG
     | OBJECT
     | fstExpression
     | address
     | type
     | string
     | regset
     ;
operand: REG
     | OBJECT
     | expression
     | address
     | type
     | string
     | regset
     ;

regset: '{' reg ('-' reg?)? (',' reg ('-' reg?)?)* '}'
        ('[' NUMBER ']')? ;
string: STRING ;
unaryop: ('~' | '!' | '-' | '+') ;
binop: ('<<' | '+' | '-' | '>>'| '*' | '|' | '||' | '&' | '&&' | '^'
       | '/' | '%') ;
symbol: (IDENTIFIER | 'f' | 'b');
modifier: IDENTIFIER ;
type: ('%' | '@') IDENTIFIER ;
address: '[' addressExpr (',' addressExpr)* ']' '!'? ;
addressExpr: (reg | expression | string '@' IDENTIFIER) ;

fstExpression: fstBinary;
fstBinary: fstUnary (binop binary)? ;
fstUnary: unaryop? fstTerm ;
fstTerm:  '(' expression ')'
  | symbol
  | immmodifier (NUMBER | FLOAT | '#' unary)?
  | (NUMBER ('f' | 'b')? | FLOAT | '#' unary)
  ;
expression: binary;
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
49 binary: unary (binop binary)? ;
50 unary: unaryop? term ;
51 term: '(' expression ')'
52   | (':' modifier ':')? symbol
53   | immmodifier (NUMBER | FLOAT | '#' unary)?
54   | (NUMBER ('f' | 'b')? | FLOAT | '#' unary)
55   ;
56 immmodifier
57   : 'lsl' | 'msl' | 'uxtx' | 'asr' | 'lsr' | 'uxtb' | 'uxth' | 'ror' |
58     'sxtw' | 'uxtw' | 'sxtb' | 'sxtx' | 'sxth' ;
59
60 reg: REG;
```

Listing A.5: ANTLR3 Grammar for AArchTest Set

```
1 statements: (statement)* ;
2
3 statement
4   : ('acquire' | 'release') 'lock' statement |
5     STRING ':' EOS |
6     NUMBER ':' EOS |
7     IDENTIFIER (
8       (operand? (',' operand)*) |
9       ':'
10    ) EOS |
11    EOS
12  ;
13
14 operand:
15        expression
16  ;
17 maskregmod: '{' IDENTIFIER '}' ;
18 unaryop: ('~' | '!' | '-' | '+' | 'not') ;
19 binop: ('<<' | '+' | '-' | '>>'| '*' | '|' | '||' | '&' | '&&' |
20        '^' | '/' | '%' | IDENTIFIER) ;
21 namedValue: (IDENTIFIER | STRING) ('@' IDENTIFIER)? (address)?;
22 type: '@' IDENTIFIER ;
23
24 expression: binary;
25 binary: unary (binop binary)? ;
26 unary: unaryop? term ('(' (IDENTIFIER | NUMBER) ')' | maskregmod+)?;
27 term: '(' expression ')'
28   | NUMBER ('b' | 'f')?
29   | address+
30   | namedValue
31   | ('ST' | 'st')
32   | (SIZE ('ptr' | 'PTR')?) term
33   | SEGMENTREG ':' (NUMBER? address?)
34   | regset
35   ;
36 regset : '{' IDENTIFIER ('-' IDENTIFIER)? '}' ;
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
address : '[' expression (',' expression)* ']' ;
```

Listing A.6: ANTLR3 Grammar for x86 Test Set

```
statements: (bundle)*  ;

bundle
  :   statement
  |   '{' statement* '}'
  ;

statement:
        IDENTIFIER (
            callArgs? (',' IDENTIFIER callArgs?)?
              (('|=' | '+=' | '=' | '^=' | '&=' | '-=') expression)? |
            oplist
        ) EOS |
        ('if' | 'IF') '(' expression ')' statement |
        EOS
  ;

unaryop: ('~' | '!' | '+' | '-') ;
binop: ('<<' | '+' | '-' | '>>'| '*' | '|' | '||' | '&' | '&&' |
        '^' | '/' | '%' | '++' | '!=') ;
type: '@' IDENTIFIER ;

expression: binary;
binary: unary (binop binary?)? ;
unary: unaryop? term ;
term:    IDENTIFIER (
    callArgs |
    selector
  )?
  | NUMBER
  | '#' (
    unaryop? (IDENTIFIER | NUMBER) |
    ('HI' | 'hi' | 'lo' | 'LO') ('(' unary ')')? |
    '(' expression ')'
      )
  | '##' unaryop? (IDENTIFIER | NUMBER)
  | STRING
  | type
  ;
callArgs: '(' oplist ')' callopt* selector? ;
callopt : ':' ('<<' NUMBER | IDENTIFIER);
oplist  :   expression (',' expression)* ;
selector: '.h' | '.tmp' | '.new' | '.b' | '.w' |
          '.uw' | '.uh' | '.ub';
```

Listing A.7: ANTLR3 Grammar for Hexagon Test Set

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

A.5 Test Programs for Evaluation

This section contains the listings for the test program evaluation.

```
1 # "Hello World" in MIPS assembly
2 # From: labs.cs.upt.ro/labs/so2/html/resources/nachos-doc/mipsf.html
3     .text
4     .globl      main
5 main:
6     li      $v0,4
7     la      $a0, msg
8     syscall 0
9     li      $v0,10
10    syscall 0
11    .data
12 msg:     .asciiz "Hello World!\n"
```

Listing A.8: Test Program for MIPS

```
1 {
2    R1=#1
3    R2=#2
4 }
5 R3=add(R1,R2)
```

Listing A.9: Test Program for Hexagon

```
1 {
2    mov R1=#1
3    mov R2=#2
4 }
5 add R3=add(R1,R2)
```

Listing A.10: Modified Test Program for Hexagon

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

List of Figures

2.1 Canonical Compiler Process 17
2.2 Canonical LLVM Compiler Process 27

3.1 Feedback Loops in the Design Process 33

4.1 Internal and External Instruction Representations 55

5.1 Comparison of Lines per Category 71
5.2 Lines devoted to Low-Level Developer Tools 72
5.3 Assembly Benchmarks Relative to Official RISC-V 73
5.4 Linker Benchmarks Relative to Official RISC-V 74

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Tables

4.1 Default Non-terminal Rules 39
4.2 Possible Casts with LCB Semantics 41
4.3 Operational Types for Grammar Elements 43
4.4 Native Slice Semantics 47
4.5 Default Relocations 57
4.6 Fixup Type Description for Expression Types 64

5.1 List of Specifications used for the Evaluation 69
5.2 Test System used for Benchmarks 72
5.3 Results of Test Suites 75
5.4 Number of Generated Grammar Rules 78
5.5 Performance of Grammar Rule Generation 80

A.1 Full Line Counts 90
A.2 Assembly Performance Data in ms 91
A.3 Linking Performance Data in ms 92

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Listings

2.1 An Instruction Set Architecture 9
2.2 Definition of Memory 9
2.3 Definition of Registers and Register Files 9
2.4 Definition of a Program Counter 10
2.5 Definition of an Instruction Format 10
2.6 JType with Immediates 11
2.7 Definition of a Machine Instruction 12
2.8 Definition of Pseudo and Compiler Instructions 12
2.9 Definition of an Instruction Encoding 13
2.10 Assembly Printing Function 13
2.11 Application Binary Interface Definition 14
2.12 Alias Registers 14
2.13 Calling Conventions 14
2.14 System Registers 15
2.15 Micro Processor Definition 15
2.16 Micro Architecture Definition 16
2.17 LL Grammars for a Simple Language 23
2.18 LLVM IR Example 26
4.1 Basic Grammar Elments 39
4.2 Basic Grammar 41
4.3 Sequence with Feature Bindings 42
4.4 Transformation Functions 43
4.5 Grammar Rule for LUI 43
4.6 Alias Directives 44
4.7 Modifier Mapping 44
4.8 Example for Assembly Description Annotations 45
4.9 Assembly Formatting Function for RISC-V ADDI 46
4.10 Assembly Formatting Function for more complex ADDI 48
4.11 Assembly Formatting Function for CSRRW 49
4.12 Generated Parser Rules for CSRRW 49
4.13 Definition of the Parsing Function 51
4.14 Non-terminal Rule and String Literal Semantics 51
4.15 Sequence and Binding Semantics 52

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.16 Alternative Semantics 53
4.17 Semantics of Data Elements 54
4.18 Relocations of the RV32I Processor with Conservative Strategy 57
4.19 Relocations of the RV32I Processor with Performance Strategy 59
4.20 Complex Expression with Operand 60
4.21 Disassembly of RISC-V Instructions 64
4.22 Example of a Test Program 66
5.1 Two Variants of the MIPS LW Instruction 76
5.2 ANTLR Grammar Snippet for LW Instruction 76
5.3 Two Variants of the RISC-V JALR Instruction 77
A.1 Internal VIR Code for the `WRegSP` Function 88
A.2 Data Structure for Parser Semantics 89
A.3 ANTLR3 Grammar for RISC-V Test Set 93
A.4 ANTLR3 Grammar for MIPS Test Set 94
A.5 ANTLR3 Grammar for AArchTest Set 95
A.6 ANTLR3 Grammar for x86 Test Set 96
A.7 ANTLR3 Grammar for Hexagon Test Set 97
A.8 Test Program for MIPS 98
A.9 Test Program for Hexagon 98
A.10 Modified Test Program for Hexagon 98

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Acronyms

ABI Application Binary Interface. 7, 8, 14, 15, 17, 18, 19, 37, 80

ASIC Application Specific Circuit. 1

ASIP Application Specific Instruction Set Processor. 1

AST Abstract Syntax Tree. 8, 13, 24, 33, 40, 66

BNF Backus-Naur form. 24

CAS Cycle-Accurate Simulator. 1, 32, 33, 34, 35

CFG Context-Free Grammar. 21, 22, 23

CSR control and status register. 49, 79

CST Concrete Syntax Tree. 8, 21

DSP digital signal processor. 35

ELF Executable and Linkable Format. 17, 18, 45, 57, 58, 59, 63, 65, 66

GCB Generic Compiler Backend. 8

ILP instruction-level parallelism. 14, 34

IR intermediate representation. 8, 21, 26, 27, 29, 51, 53, 88

ISA Instruction Set Architecture. 7, 9, 10, 12, 13, 14, 19, 20, 28, 32, 33, 34, 35, 37, 49, 62, 65, 66, 69, 81

ISDL Instruction Set Description Langauge. 34

ISS Instruction Set Simulator. 1, 32, 66

LBNF labeled Backus-Naur form. 24

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

LCB LLVM Compiler Backend. 5, 11, 12, 16, 20, 23, 29, 37, 38, 40, 41, 44, 52, 53, 54, 57, 58, 61, 62, 63, 64, 65, 66, 67, 73, 75, 77, 78, 80, 81, 83, 84, 85, 101

LTO link-time optimization. 20

OOP object-oriented programming. 65

PC program counter. 10, 18, 20, 58, 65

PDL Processor Description Language. 1, 5, 7, 13, 31, 32, 33, 34, 35

POJO Plain Old Java Object. 51

RTL Register Transfer Level. 8, 32

SOC System-on-Chip. 34

SSA single static assignment. 8, 25, 26

VADL Vienna Architecture Description Language. 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 16, 22, 29, 37, 38, 40, 42, 45, 46, 47, 50, 53, 54, 59, 62, 63, 64, 65, 66, 70, 78, 80, 83, 84, 85, 87, 88, 106

VIR VADL Intermediate Representation. 8, 13, 40, 42, 46, 47, 50, 60, 66, 80, 87, 88, 104

VLIW very long instruction words. 3, 5, 13, 34, 35, 69, 74, 80, 81

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Bibliography

[ARB+05] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. The ArchC architecture description language and tools. *International Journal of Parallel Programming*, 33(5):453–484, 2005.

[BBVB+01] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. 2001.

[Bou96] Richard J. Boulton. Syn: a single language for specifiying abstract syntax trees, lexical analysis, parsing and pretty-printing. Technical Report UCAM-CL-TR-390, University of Cambridge, Computer Laboratory, March 1996.

[Cho56] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[Con58] Melvin E Conway. Proposal for an UNCOL. *Communications of the ACM*, 1(10):5–8, 1958.

[CPP+20] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. *ACM Comput. Surv.*, 53(1), feb 2020.

[CT11] Keith D Cooper and Linda Torczon. *Engineering a compiler*. Morgan Kaufmann, 2nd edition, 2011.

[DBDSVP+04] Bruno De Bus, Bjorn De Sutter, Ludo Van Put, Dominique Chanet, and Koen De Bosschere. Link-time optimization of ARM binaries. *SIGPLAN Not.*, 39(7):211–220, jun 2004.

[DJ11] Jonas Duregård and Patrik Jansson. Embedded parser generators. volume 46, pages 107–117, 12 2011.

[FR03] Markus Forsberg and Aarne Ranta. Labelled BNF: a highlevel formalism for defining well-behaved programming languages. *Proceedings of the Estonian Academy of Sciences. Physics, Mathematics*, 52, 01 2003.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[FR04] Markus Forsberg and Aarne Ranta. BNF converter. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, page 94–95, New York, NY, USA, 2004. Association for Computing Machinery.

[Fre91] Markus Freericks. *The nML machine description formalism*. Leiter der Fachbibliothek Informatik, Sekretariat FR 5-4, 1991.

[GHK+98] Peter Grun, Ashok Halambi, Asheesh Khare, Vijay Ganesh, Nikil Dutt, and Alexandru Nicolau. Expression: An ADL for system level design exploration. Technical report, Citeseer, 1998.

[HD17] Qinheping Hu and Loris D'Antoni. Automatic program inversion using symbolic transducers. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 376–389, New York, NY, USA, 2017. Association for Computing Machinery.

[HH95] John Hughes and Chalmers Hogskola. The design of a pretty-printing library. 06 1995.

[HHD97] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. ISDL: An instruction set description language for retargetability. In *Proceedings of the 34th Annual Design Automation Conference*, DAC '97, page 299–302, New York, NY, USA, 1997. Association for Computing Machinery.

[HVQ+12] Cong Hou, George Vulov, Daniel Quinlan, David Jefferson, Richard Fujimoto, and Richard Vuduc. A new method for program inversion. In Michael O'Boyle, editor, *Compiler Construction*, pages 81–100, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[Knu05] Donald Ervin Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2:127–145, 2005.

[LA04] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis amp; transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004.

[Lev99] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann, 1999.

[LMB92] John R. Levine, Tony Mason, and Doug Brown. Lex & yacc, 2nd edition. 1992.

[Mar84] P. Marwedel. The MIMOLA design system: Tools for the design of digital processors. In *21st Design Automation Conference Proceedings*, pages 587–593, 1984.

[MB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[MD08] Prabhat Mishra and Nikil Dutt. Chapter 1 - introduction to architecture description languages. In Prabhat Mishra and Nikil Dutt, editors, *Processor Description Languages*, volume 1 of *Systems on Silicon*, pages 1–12. Morgan Kaufmann, Burlington, 2008.

[MHJM13] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System V application binary interface. *AMD64 Architecture Processor Supplement, Draft v0*, 99(2013):57, 2013.

[MK14] Nicholas D. Matsakis and Felix S. Klock. The Rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA, 2014. Association for Computing Machinery.

[MMHT10] Kazutaka Matsuda, Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. A grammar-based approach to invertible programs. In Andrew D. Gordon, editor, *Programming Languages and Systems*, pages 448–467, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Moo00] Robert C Moore. Removing left recursion from context-free grammars. In *1st Meeting of the North American Chapter of the Association for Computational Linguistics*, 2000.

[MW13] Kazutaka Matsuda and Meng Wang. FliPpr: A prettier invertible printing system. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 101–120, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Pan01] Preeti Ranjan Panda. SystemC: A modeling platform supporting multiple design abstractions. In *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS '01, page 75–80, New York, NY, USA, 2001. Association for Computing Machinery.

[PHZM99] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. LISA—machine description language for cycle-accurate models of programmable DSP architectures. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, pages 933–938, 1999.

[PQ95] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[RO10] Tillmann Rendel and Klaus Ostermann. Invertible syntax descriptions: Unifying parsing and pretty printing. *SIGPLAN Not.*, 45(11):1–12, sep 2010.

[RS69] D. J. Rosenkrantz and R. E. Stearns. Properties of deterministic top down grammars. In *Proceedings of the First Annual ACM Symposium on Theory of Computing*, STOC '69, page 165–180, New York, NY, USA, 1969. Association for Computing Machinery.

[RWZ88] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, 1988.

[Sis98] C. Siska. A processor description language supporting retargetable multi-pipeline DSP program development tools. In *Proceedings. 11th International Symposium on System Synthesis (Cat. No.98EX210)*, pages 31–36, 1998.

[SL08] Armando Solar-Lezama. *Program synthesis by sketching*. University of California, Berkeley, 2008.

[WLP+14] Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovic, Volume I User level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. The RISC-V instruction set manual. *Volume I: User-Level ISA, version*, 2, 2014.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery.

[ZLWG20] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L. Glassman. Interactive program synthesis by augmented examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, UIST '20, page 627–648, New York, NY, USA, 2020. Association for Computing Machinery.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub