TU WIEN Informatics

Compiler Backend Generation using the VADL Processor Description Language

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Alexander Graf, BSc
Matrikelnummer 01429203

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: ao. Univ. Prof. Dipl.-Ing. Dr. Andreas Krall

Wien, 13. April 2021

Alexander Graf	Andreas Krall

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.
TU Bibliothek Your knowledge hub WIEN

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 Informatics

Compiler Backend Generation using the VADL Processor Description Language

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Alexander Graf, BSc
Registration Number 01429203

to the Faculty of Informatics

at the TU Wien

Advisor: ao. Univ. Prof. Dipl.-Ing. Dr. Andreas Krall

Vienna, 13th April, 2021

Alexander Graf	Andreas Krall

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.
TU Bibliothek Your knowledge hub WIEN

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

Alexander Graf, 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, 13. April 2021

Alexander Graf

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

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

I could not have finished this work without the help and guidance of my advisor and chairman of this project Andreas Krall, which has supported me on multiple occasions and assisted me patiently during the writing process. I also want to thank everyone involved in this research project for giving me advice. I particularly want to mention my colleagues Herman Schützenhofer, who co-authored the sections regarding processor description languages and Christoph Hochrainer, who has written parts of the retargetable compiler sections and implemented parts of the compiler generator.

In addition, I want to thank friends and family for their patience. I want to give special thanks to my parents, especially my mother, for supporting me all these years during my time at university.

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

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 Nutzung von digitalen Geräten hat sich in den letzten Jahrzehnten drastisch erhöht. Eine Vielzahl von Alltagsgegenständen, sei es nun ein Smartphone oder ein Auto, haben einen Prozessor verbaut. Zusätzlich sind Konsumenten gewohnt, dass diese Geräte immer schneller und besser werden. Dies hat zur Folge, dass Hardwareproduzenten immer kleinere und leistungsstärkere Prozessoren auf den Markt bringen müssen. Infolgedessen haben sich sogenannte Application Specific Instruction Set Processors (ASIPs) etabliert, da diese Art von Prozessoren mit den Leistungsansprüchen und dem erhöhten Entwicklungsaufwand gut mithalten können. Jedoch werden dafür ein großes Maß an Fachwissen und zusätzliche Werkzeuge für das Evaluieren des Prozessordesigns benötigt. Um die Markteinführung und gleichzeitig den sich wiederholenden Entwicklungsaufwand zu reduzieren, werden Processor Description Languages (PDLs) verwendet. Diese Beschreibungssprachen erlauben es dem Benutzer, einen Prozessor auf einer höheren Abstraktionsebene zu modellieren und gleichzeitig die notwendigen Werkzeuge zu erzeugen.

Diese Arbeit stellt die Beschreibungssprache Vienna Architecture Description Language (VADL) vor und beschreibt, wie diese dazu verwendet werden kann, um ein LLVM Compiler Back-end zu generieren. Nach der Verarbeitung und Analyse einer VADL Spezifikation werden Instruction Selection Patterns und weitere LLVM Artefakte erzeugt.

Der Großteil dieser Arbeit beschäftigt sich mit den Methoden dieses Compilergenerators und den Sprachelementen, die für die Konfiguration eines Compilers notwendig sind.

Die Arbeit wurde anhand einer VADL Spezifikation für den **RV32IM** Teil des **RISCV** Befehlssatz evaluiert. Der generierte Compiler wurde mit dem von LLVM zur Verfügung gestellten Compiler verglichen. Die Programme haben dabei im Schnitt um rund 12,11% schlechter abgeschnitten und 16,51% mehr Assembler-Befehle erzeugt. Die Ergebnisse sind dennoch beachtlich, da sowohl VADL als auch der Compilergenerator noch in den Kinderschuhen stecken und keine wesentlichen Optimierungen implementiert wurden. Dies lässt darauf schließen, dass in Zukunft deutlich bessere Ergebnisse zu erwarten sind.

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

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

The use of digital devices has dramatically increased in the last decades. Many day-to-day devices, such as smartphones or cars, embed some form of a processor. In addition, consumers expect their devices to become smaller and faster with each generation. This enforces hardware producers to create smaller and better performing processors in a short period of time. Application Specific Instruction Set Processors (ASIPs) have shown to meet the performance demands and increased development output, but require additional application knowledge and tooling for evaluating the processor design. To minimize time-to-market and repetitive work, Processor Description Languages (PDLs) are used. Instead of designing the processor and all its tools individually, such languages provide a higher abstraction level for designing embedded processors and generators that automatically produce all necessary tools.

This work introduces the Vienna Architecture Description Language (VADL), a mixed PDL, and discusses how it is used to automatically generate a LLVM compiler backend. After extracting and analyzing the processor information from the VADL specification, instruction selection patterns and other LLVM specifications are generated. Most of this thesis focuses on the generator methodologies and language elements required for this retargeting process.

The approach is evaluated by specifying the **RISCV** subset **RV32IM** in VADL and comparing the automatically generated compiler to the open source LLVM compiler. On average, programs perform only 12.11% worse and produce 16.51% more instructions for the assembly output, which is remarkable considering that the generator does little to no target-specific optimizations. Furthermore, both VADL and the compiler generator are still in an early stage of development. Much better results can be expected in the future.

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

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

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

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

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 1

Introduction

The last decades have shown a tremendous raise in ASIP use [MD11]. The design process of these ASIPs involves a lot of repetitive work while exploring the processor design space. To minimize repetition, an Architecture Description Language (ADL) can be used. In the context of processor design it is also called Processor Description Language (PDL). PDLs allow to design a processor on a higher abstraction level and generate all the necessary components for the exploration phase automatically. A compiler is generated to compile the desired application for the designed Instruction Set Architecture (ISA). To further investigate the behavior of the processor, an automatically generated simulator is used to simulate the target machine and to execute the compiled application, collecting information regarding its performance, like energy consumption, execution speed and cycle stalls. According to this data the processor description can be adopted and the process starts again. Actual hardware can then be generated in form of a RTL model.

1.1 Motivation

Nowadays a variety of different PDLs exist, ranging from simple language extensions that tweak an existing ISA to languages that are capable of describing complex concurrent processor architectures [MD11, HL10]. However, these languages usually concentrate either on one or two of the three mentioned objectives, that are compiler-, simulator- or hardware generation. To leverage all the functionality used for ASIP design, mixed languages are used.

1.2 Problem statement

PDLs are an integral part in processor design and help minimizing costs and time-to-market. For a PDL to be useful, little to no user interaction for the generation from a

processor description is desired. Therefore, it will be evaluated how much of a compiler can be generated without user intervention, using a modern compiler framework.

1.3 Aim of this work

This master thesis is part of a larger research project conducted by the compiler and languages group at TU Wien, which focuses on designing a PDL, capable of describing the ISA and Microarchitecture (MIA) of a processor. From this description a compiler, a cycle-accurate simulator and a hardware model will be generated. This work will focus on the first part, the compiler. Additionally, a large portion of this work is dedicated to design this PDL, so that a LLVM[1] backend can be generated from it. This project will develop a proof of concept rather than a finished product and therefore focuses on the generation of compiler backends of Reduced Instruction Set Computer (RISC) architectures, especially RISCV. For this architecture a comparable performance of the generated code should be achieved (compared to the open source backend provided by LLVM).

1.4 Structure of this work

This work is split into six chapters. The state-of-the-art chapter gives an overview of techniques and methods currently applied to the problem statement. The next chapter will give an overview of the PDL called VADL, focusing on the parts most relevant for compiler generation. After that, the implementation will be discussed in more detail. The future work chapter will provide insight to the implementation, as well as a short overview of possible future extensions. Then the results of the conducted benchmarks are presented. At the end, a short summary of this thesis is given.

[1] 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 Bibliothek Your knowledge hub WIEN

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

State of the Art

This chapter will give a detailed tour of the different parts of PDLs and retargetable compilers. Then the relation between the retargetability process and PDLs is shown.

2.1 Processor Description Language

The following three subsections are based in large parts on the excellent introduction to this topic by Mishra and Dutt [MD11].

2.1.1 Introduction

A Processor Description Language (PDL) is a specialized Architecture Description Language (ADL), which is capable of describing a processor architecture, including its structural components and instruction behavior. Based on a high-level description of a processor architecture in a PDL, it is not only possible to automatically create various artifacts like a compiler or simulator, it can also allow to perform hardware synthesis and various test and validation tasks. A development process that uses these abilities to generate artifacts in its tool-chain, can significantly reduce the overall implementation efforts for creating new and enhanced processor architectures. By allowing rapid design exploration this also ensures the quality of created processor designs under given constraints like power consumption, chip area and manufacturing cost.

While a processor architecture can certainly be expressed in any programming language, a PDL based solution has the advantage that it was specifically build to express architectural abstractions of this problem domain. Additionally a PDL is commonly quite capable of capturing specific and complex hardware features like synchronization, which can otherwise be hard to express in traditional programming languages. Therefore, using a Hardware Description Language (HDL) would in this case be a better choice, because such languages are also quite suitable to express hardware features. But these languages

commonly only provide a lower abstraction level than a PDL and extracting the instruction behavior from such a processor architecture description can therefore be nearly impossible. While programming languages, HDLs and PDLs have clearly some common ground, the latter has an advantage when it comes to express architectural aspects and as a consequence also makes it easier to extract various kinds of information, that are necessary for automatically generating artifacts.

Ideally a PDL allows to create a complete and formal specification of a processor architecture, which is expressive, easy to understand and maintain and does not contain any redundant or ambiguous content. With the additional aim to support a wide range of instruction set architecture and micro-architecture designs, it becomes rather impossible to fulfill all of these ideals to a full extend. Due to this reason, there exist a number of different PDLs today, which can be classified by the content they are capturing or by the objective they have been created for.

2.1.2 Content-based classification

PDLs can be assigned into one of three categories, based on the content they can represent.

First there are the **structural PDLs**, which have, as the name implies, a focus on describing the structural aspects of the processor architecture. This type of PDLs is typically providing lower level abstractions like the RTL, which allows to create a very detailed description of hardware features and components, while still preserving a certain level of abstraction. Members of this category like for example *MIMOLA* are quite suitable to perform hardware synthesis and emit artifacts like a cycle-accurate simulator, which needs detailed information about the Microarchitecture (MIA).

The second category are the **behavioral PDLs**, which define the instructions and the semantics according to the Instruction Set Architecture (ISA), while omitting the description of hardware details. Members of this category, like ISDL, are very suitable to generate a compiler or an Instruction-Set Simulator (ISS).

While *structural- and behavioral PDLs* both have a distinct area of application, there also exists the attempt of combining both of them with **mixed PDLs**. Members of this third category attempt to capture structural and behavioral information and are therefore suitable to emit all possible artifacts from a processor architecture description. PDLs following this approach, like *LISA* or *nML*, can vastly differ in their implementation and supported feature sets.

2.1.3 Objective-based classification

Similar to the previous section it is also possible to classify PDLs based on their target objectives. These objectives are **compilation**, **simulation**, **synthesis** and **validation**.

PDLs having the objective to generate tools for **compilation** focus on retargeting an existing compiler using target machine information. This can reduce the overall amount of source code necessary to support a new target machine. Both **behavioral** and **mixed**

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

PDLs provide this information, such as instruction-set, resources and resource conflicts that can be used to parameterize a retargetable compiler. The types of retargetability can be further categorized based on the amount of detail provided by PDLs, the phases of a modern retargetable compiler and the particular architecture abstraction.

Although **structural PDLs** in general are not suitable for compiler generation, some attempts show promising results at extracting **behavioral** information from **structural** processor descriptions [BEK07]. However some additional meta information must be provided to fully support the generation of a compiler.

Simulation of a processor operates on different levels of abstraction. They can operate on the lowest level, considering timing information of various hardware components. This type of simulation is performed by cycle-accurate and phase-accurate simulators. For this type both **structural** and **mixed** PDLs are good choices. Whereas simulation on a higher abstraction level, usually considers only instruction-set information, which is done using instruction-set simulation. This type uses information provided by both **behavioral** and **mixed** PDLs. As for compilers, this approach can also use retargetable simulators and parameterize them to support different targets.

The synthesis of hardware needs detailed information about hardware components of a processor model, which makes **structural PDLs** and **mixed PDLs** suitable for this objective. Although **behavioral PDLs**, such as ISDL are also capable of hardware synthesis. Usually the synthesis process generates RTL descriptions in VHDL or Verilog.

The validation of a processor is an important task of the design process, which helps to find errors in the specification. Several PDLs are therefore capable of functional verification. Both **structural** and **mixed** PDLs are used for test generation. Which often apply techniques, such as property or equivalence checking, as well as simulation based approaches.

The following PDLs were chosen to give further insights on design decisions about the different kinds of PDLs.

2.1.4 Expression

Expression is a mixed PDL with Lisp-like syntax [HGG+08, MD11]. These two works describe the language as following. Focusing on SOC architectures the description is used to retarget a CAS and a compiler which optimizes for ILP. The behavioral view is split into operations specification, instruction description and operation mappings. Instruction set information provided here is used to retarget both compiler and simulator. An operation regarding the ISA is defined using opcodes, operands, semantics and binary format. An instruction definition describes how several operations can be parallelized by assigning them to different functional unit slots. Operation mappings can be used to associate compiler operations to target operations (instruction selection pattern) or can be used for mapping target operations to target operations (target optimization). The structural view is split into components specification, pipeline and data-transfer

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

paths description, and the memory subsystem. A components specification defines the RTL components of the processor architecture, like pipeline units, functional units, storage elements, ports or bus connections. Whereas the pipeline and data-transfer paths description defines the netlist of the processor, allowing to specify the units that build a pipeline, as well as describing valid data-transfers. The memory subsystem describes storage elements in more detail. The information provided in this view is used to extract the connectivity information for the simulator and reservation tables for the compiler.

2.1.5 ISDL

This summary is based on [MD11, HHD97]. The Instruction Set Description Language (ISDL) is a behavioral PDL and focuses on compiler- and assembler generation, as well as hardware synthesis. A specification is depicted of the instruction word format, global definitions, storage resources, instruction set, constraints and optional details about an architecture. The instruction word format defines the parts of the binary representation of an instruction. In the global definitions section, tokens, non-terminals and split functions can be defined. Tokens can be used to describe several components, like register and memory bank names, as well as immediate constants and correspond to the assembly syntax. These tokens can be grouped together, if they are syntactically related. Non-terminals can be used for defining rules, which group syntactically unrelated tokens together or to define syntax combinations of instructions. It is also possible to annotate these rules with C code. Split functions can be used to extract fields of the instruction word from long bit fields. These mechanisms can be used in several other definition sections to model instructions or the assembly syntax for example. Storage resources correspond to structural components, although this somehow contradicts the previous statement that ISDL is a **behavioral PDL**. However the behavior of a processor, i.e. its instructions, cannot be defined without the resources on which they operate (memory register, etc). The instruction set is defined in terms of operations, which can be executed simultaneously by a single instruction. Operations contain the assembly mnemonic, operands, binary representation, semantic in form of a RTL, costs and timing information. Constraints are boolean rules and can be defined in regard of a data path, bit fields or the assembler syntax. It is also possible to provide additional information for compiler optimizations.

2.1.6 LISA

This subsection is based on the works of [MD11, SHN+02, HL10, HKN+01]. The Language for Instruction Set Architecture (LISA) provides different abstractions of a processor, allowing to specify both the **behavioral** and **structural** information. The language supports a variety of architectures, for which it is possible to generate a compiler, assembler, linker, simulator, profiler and a hardware description. A LISA specification is split into the following models and constructs. It is possible to define structural components like registers and memories, containing bit widths, ranges and aliases. Operations can be defined to model the instructions of the ISA, including their

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

semantics (using C/C++ like constructs), binary representation and assembly syntax. The semantics of the instruction are split into various section definitions, which describe among other things their effect on the processor, simulation behavior and the correlation to compiler instructions. One reason for the different behavioral definitions seems to be the use of C/C++ constructs which make it hard to extract certain types of information. It is also possible to define detailed timing information and to model the pipelining behavior. In addition, tools for the processor designer are provided to help during the design phases and enable the user to capture information not directly modeled in a LISA specification.

2.1.7 MIMOLA

The following summary is based on [MD11, Mar84, Mar86]. The Machine Independent Microprogramming Language (MIMOLA) is a structural PDL, which has been designed for synthesis and is also capable of simulation. The general approach is centered around the idea of high-level synthesis, where a set of typical application programs written in a high-level programming language is used as input for synthesis. Two variants of a *Mimola Software System* (MSS1 and MSS2) have been developed around this idea and the remaining summary in this section will focus on MSS2, which was used for academic research until the early 90s and consists of multiple separate tools. A design specification in MSS2 consists generally of four parts. A typical set of application programs, a set of replacement rules to translate used high level language elements into equivalent RTL elements, a description of execution frequencies and hardware resources. High-level input programs could also be provided in *Pascal* instead of MIMOLA by using a precompiler. A typical design flow starts with the architectural synthesis by providing these necessary inputs. Various tools of the MSS2 can then be used for manual adaptations and design space exploration. The MSS2 tool chain is based around a *Lisp* like internal representation called *TREEEMOLA*, which can be enriched with various types of data and also be expressed in the MIMOLA language. The tool chain does also support mechanisms like the creation of multiple implementation variants for IF-statements and can also delay decisions to choose the most appropriate one for the given hardware design. MSS2 does also support a retargetable compiler and automatic creation of test programs on the register-transfer level.

2.1.8 nML

This summary is based on [FVF95, MD11]. A nML processor specification consists of a structural description of the target machine called a skeleton and the execution behavior on basis of register-transfer instructions. nML can therefore be classified as a **mixed PDL**. The skeleton describes the processor state by defining static and transitory storage components but also functional units, storage aliases, constants and enumeration types. Memory and register components are called static in this context, because they will store values until explicitly overwritten. Values written to transitory storage elements like buses and pipeline registers on the other hand will only be available for a specified number 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 Bibliothek Your knowledge hub WIEN

machine cycles. An instruction set can be described in nML by a grammar that consists of *AND-rules* and *OR-rules*, which describe compositions and alternatives. Each derivation from this grammar represents a single instruction, which significantly reduces the amount of necessary description for a typical processor. Additional grammar attributes are used to specify the behavior on an register-transfer level (action-attribute), the assembly syntax (syntax-attribute) and the binary encoding (image-attribute). Memory and register addressing modes can also be specified by a special *mode-rule*. The handling of control-, data- and structural hazards for pipelined processor models can also be specified by the designer. The nML toolchain consists of a retargetable C-compiler called *Chess*, a retargetable CAS generator named *Checkers*, the hardware description language generator *Go* and a retargetable test-program generator called *RISK*.

2.1.9 RADL

This subsection is based on the paper by Siska [Sis98]. The Retargetable Architecture Description Language (RADL) shares common traits with languages like *nML* and LISA and therefore can also be categorized as **mixed PDL**. RADL is focused on the generation of cycle- and phase accurate simulators and its key feature is its explicit event based description of the pipeline model. The used description technique allows as the author claims, to intuitively describe various features like delay slots, interrupts, hardware loops and data hazards. But it also allows the support of sub-pipelines and inter-pipeline control and communication. The pipeline behavior in RADL is described by a strategy table, which specifies the expected stall and flush operations in relation to occurring signals. Each strategy consists of a control signal, which specifies when the strategy can be applied, the effected target pipeline stage and the expected behavior in form of an instruction stall or flush action. If multiple strategies are simultaneously applicable then the first one in the order of the pipeline specification is being selected. And a default strategy to fetch and decode the next instruction is used when no other strategy is applicable. Signals also have to be declared in RADL as either simple or composite signal. The latter hereby supports additional boolean expressions that can be built up from previously defined signals. Pipeline stages can also be partitioned into multiple phases, to support multiple pipelines that run at different but synchronized clock cycles. And while pipeline registers also have to be declared manually, they do support a default copy semantic to move values from each pipeline register to its predecessor to reduce additional specification efforts. But the author doesn't disclose an evaluation for a generated simulator, which makes it impossible to asses the effectiveness of the approach against similar PDLs.

2.1.10 Sail

This short summary is based on the work by [ABC+19]. Sail is a language to formally describe the semantics of an ISA. The automatic generation of documentation and an *OCaml* and *C* based ISS are supported as well as the automatic creation of definitions for various proof-assistants like *Isabelle*, *HOL4* and *Coq*. The Sail language is a first-order

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

functional and imperative language, which does support loops and recursion, but no higher order functions. An exception mechanism as well as support for arbitrary-precision rational numbers has been added to enable the support for the ARMv8 architecture. ISA descriptions in Sail have been created for ARMv8.3-A, RISC-V, MIPS and Cherry-MIPS and evaluated by booting operating systems like Linux or FreeBSD. The extensive specification for ARM has hereby been derived from a machine readable version of the architecture specification language *ASL*. Sail has a primary focus to express ISA specifications with limited abilities to define structural components of a processor and can therefore probably be categorized as **behavioral PDL**.

2.2 Retargetable Compilers

It is assumed that the reader is familiar with the basic concepts of compilers. This section is mainly based on the book by Leupers and Marwedel about retargetable compiler technologies for embedded systems [LM01] and the book by Aho et al. about compiler principles, techniques and tools [ALSU06].

A retargetable compiler separates the target dependent and independent components from each other. The common idea is to provide a relatively easy way to modify the target-dependent back-end and to reuse the target-independent front- and mid-end. This design makes it easier to support new ISA specifications for an existing compiler. Furthermore, retargetablity comes in different levels of configurability and complexity. Mishra et al. [MD11] categorize the configuration levels regarding PDLs into parameter-based, structure-based and behavior-based.

In general, code quality suffers under the usage of a retargetable compiler. Leupers and Marwedel stated, that it is always a trade off between offering a broad target range and providing good optimizations. Broad target ranges ensure cost-efficient exploration of the design space, which plays a major role in embedded system development.

2.2.1 Representatives

One of the first retargetable compilers is LCC [Fra91]. The target specific components of the LCC compiler include a configuration header file, target specific interface functions and code generation rules. The paper states that roughly 377 to 522 lines of code are needed to adapt those components to produce a new compiler back-end. The end product is a tightly coupled ANSI C front-end and a back-end connected by an interface and shared data structures. Although LCC performs no global optimization, it emits local code that is comparable to generally available alternatives at that time.

Another well established Compiler for General Purpose Processors (GPPs) is GCC [S+20]. It supports a variety of programming languages and targets a lot of common GPP back-ends. The compiler is not designed for broad retargeting and is therefore limited to similar processor families as already supported. The code generation uses an Intermediate Representation (IR) called RTL (Register Transfer Language). In contrast to other

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

IRs, for example as with LLVM [LA04], RTL is very low level and already consists of machine-specific instruction patterns. In this language, the instructions to be output are described in an algebraic form, stating the semantics of the instructions. The mapping between the high level IRs of GCC and RTL is done in machine description files. This task requires a notable amount of knowledge about RTL and GCC internals. Research showed that popular architectures share similarities in their machine description files and (partially) automating the generation process of these files could be a viable option in the future [PK13]. Furthermore, the compiler toolchain comes with a variety of support software, like debugger, linker and assembler, which eases the development task drastically. In conclusion, GCC is a great choice if the target family is supported, otherwise it becomes very difficult to manage retargeting or achieving decent code quality.

A more narrow target range is supported by the retargetable code generator system Marion [BHE91]. It was specifically designed for RISC architectures. The authors emphasize that in contrast to CISC, RISC architectures implement most operations only one way and exposes the pipeline and functional unit cost to the code generator. That is the reason why Marion strongly focuses on retargetable instruction scheduling and global register allocation. The compiler uses the lcc front-end and its own back-end. The back-end is configured by providing a machine description language called Maril. The description is separated in three sections, a resource declaration, a runtime model and an instruction set description. Although the target scope is narrowed, the authors stated that Marion cannot handle all details perfectly, especially for "complex" Reduced Instruction Set Computers (RISCs).

Moving away from General Purpose Processors (GPPs) and into the field of Application Specific Instruction Set Processors (ASIPs), we take a quick look at two retargetable compilers, CBC [FHKM94, FVF95] and Chess [LCGDM94, VPGLDM94]. Both are based on the mixed PDL nML [FVF95] and focus on retargetable compilers for Digital Signal Processor (DSP) architectures. Chess is the newer of both and it uses a lot of ideas from CBC. One huge difference is that Chess's action attributes of nML operations are specified by calls to primitive C library routines instead of using the builtin nML language elements for describing behavior, as done by CBC. This makes Chess more powerful and no longer bound to some limitations by nML. The retargetable compiler class for DSPs have to overcome new problems different to the "classical" code generation. Now an architecture has several functional units that work in parallel, which requires a special instruction compacting of partial instruction after the code selection phase. Furthermore, both compilers apply *data routing* to find a optimal routing path for signals in the CDFG. This task includes register allocation, which is regarded a very important optimization for DSPs. While CBC performs data routing in combination with instruction scheduling only on a local level, Chess performs both with a global search mechanism.

Another specialized framework containing a retargetable compiler is Trimaran [CGH+04]. It is a compiler infrastructure for supporting state of the art research in compiling for ILP and VLIW-like architectures. The target space of Trimaran is narrowed down to the HPL-PD [EPI00] parametric processor. HPL-PD parameter space includes 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

number and types of functional units, registers files, operation latencies and descriptors that specify instruction formats and behavior of memory and operations. TRIMARAN uses a machine-description (MDES) database to save the specification of an architecture. The user provides this information in a human-readable, high-level machine description language (HMDES). A big difference to the other approaches is, that the retargetable compiler infrastructure is not generated from the description or parses it directly, but queries the needed information from the MDES database. For this the framework uses a MDES Query System (MQS) with a procedural interface. TRIMARAN is a highly advanced and ILP specialized framework that can not only generate optimized compilers but also a detailed ISS. Together with the ability to easily parameterize processors, this makes it a attractive tool for research in back-end optimizations of ILP architectures.

One of the biggest competitor to GCC when it comes to (retargetable) compiler frameworks is the LLVM project [LA04]. In contrast to GCC, LLVM was always designed with the intention of being highly modular and providing a quick way to include new components into the framework. For now we focus on the LLVM back-end, as this is the most interesting aspect for retargetability. The framework already supports a variety of processor architectures (e.g. x86, RISC-V and HEXAGON), which makes it easy to take an existing back-end and extend it to ones needs. Besides providing glue and very target specific code fragments in form of C++ methods, the main description of the back-end happens in the TABLEGEN[1] language. TABLEGEN files provide records of domain-specific information about a target architecture (e.g. registers or instructions). It allows inline C++ on certain sections, has hierarchical classes to reduce code duplication and is generally a broad, powerful DSL. In contrast to TRIMARAN, LLVM generates C++ files directly from the TABLEGEN files. These files, together with the rest of the components provided by the framework, are composed to the final compiler.

Until now, we only took full compiler frameworks into consideration. Retargeting is not limited to compilers, but can also be applied to compiler components as seen by PROPAN [KÖ3, Kä00]. The PROPAN system has been developed as a retargetable framework for high-quality, machine-dependent postpass optimisations and analyses on assembly level. The advantage of a postpass system is that it can easily be integrated in existing systems without making complicated changes to the compiler, while making a huge difference in code quality. Using it is as simple as providing the assembler input program and an architecture description file. PROPAN's target description is provided in the hardware description language TDL. TDL has a modular structure and is composed of a specification of the hardware resources, a description of the instruction set, a constraint section, an assembly section, and a pipeline section. The decision to make another description language was driven by the requirement of needing different levels of abstractions of the possible irregular architecture and the ability to easily generate integer linear programming constraints out of it. The framework produces a phase-coupled code optimizer, that performs integrated global instruction scheduling, register reassignment and resource allocation by utilizing an integer linear programming solver.

[1] https://llvm.org/docs/TableGen/

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

The paper stated that the approach of using integer linear programming is superior than the conventional graph-based approaches as it almost always produces an exact solution [KÖ3].

The subject area of retargetable compilers and techniques is gigantic and it is impossible to cover everything here. Nevertheless, the examples should give a good insight into the matter and explain the scientific and economic interest. More examples and further discussions about retargetable compilers can be found in the book by Leupers and Marwedel [LM01].

2.3 Retargetable Compiler Phases

This section will give a rough overview of the retargetable phases of a compiler [MD11], in addition to the information required to automatically generate them. Each subsection will contain a short summary of features necessary for a PDL to express this information.

2.3.1 Instruction selection

This section is based on the book by Blindell [Bli16]. For more information on instruction selection and related techniques we recommend reading the book and its referenced literature. Instruction selection is the phase that maps the mid-level machine-independent IR to the low-level machine-dependent IR. The compiler selects the machine-dependent instructions, based on semantically equivalent machine-independent instructions. Since machine-dependent instructions might perform differently, the objective is to find a minimum cover of the program execution graph. The most common approaches for solving this problem are *tree covering*, *DAG covering* and *graph covering*. *Tree covering* is efficient for most common instructions, but fails to capture instructions supporting instruction level parallelism [LM96]. Whereas *DAG* and *graph covering* are able to directly support this kind of complex instructions [Ert99] [LM96].

To automatically retarget an instruction selector from a PDL, it should be able to express all available machine instructions and their semantics [HL10]. This semantics can be used to generate the patterns needed by the covering algorithm [CHL+05]. In addition the information to calculate costs for selecting instructions should be either given implicitly or explicitly to achieve good code quality.

2.3.2 Register allocation

This section is based on the description of Aho et al. [ALSU06]. Registers are the fastest computational unit on the target machine. They hold intermediate values or variables needed for computations. Values not held by registers need to reside in the much slower memory. The utilization of the registers can be split into two sub-tasks. *Register allocation*, during which we select the set of variables that will reside in registers at each point in the program and *register assignment*, during which we pick the specific

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

register that a variable will reside in. If no register can be mapped to a variable, spilling code must be inserted.

The most obvious information that has to be provided are the register files and the special register classes. This can be especially interesting in architectures with heterogeneous register structures. Furthermore, register allocation has to respect calling conventions or specially reserved registers. Fortunately, the resource definitions and constraints can be easily parameterized and must only be filled with the according values provided by a PDL.

2.3.3 Instruction scheduling

Instruction scheduling is an additional optimization phase, which builds upon instruction selection. The basic idea is to reorder instructions to minimize overall execution time. There are several sophisticated approaches for instruction scheduling. However many of them use a dependency graph to represent dependencies among instructions and detailed timing information. Therefore, for a PDL to support automatic instruction scheduler retargeting it has to provide this information in one form or another. PDLs modeling the pipeline structure and the instruction flow seem to be a good fit for this retargeting process [MD11].

2.3.4 Code emission

Code emission is the final phase of a compiler that translates the internal representation of code into a machine understandable binary format (object code). Such object code formats include PE (Windows), Mach-O (MacOS) and ELF (Linux). Since the machine code gets usually executed by an operating system, it must conform to the calling conventions for supported system calls. These various formats are usually split into different parts to incorporate the necessary information, such as symbols, relocations or machine code of the program. Besides the information retrieved by the ISA, namely instruction encoding, register encoding and symbol encoding, behavior has to be provided on how to deal with relocation, relaxation and misalignments.

2.3.5 Linking

Linking is used to combine and link several object files that were generated by the code emission phase together. This method was introduced to separate parts of a program into smaller pieces for better modularity. In addition, linking can perform optimization tasks depending on information, that is only available during linking (e.g. relocation, relaxation). Linking is a very complex task and therefore won't be discussed in much detail here. Please refer to [Lev00] for a thorough guide through this subject. The most interesting part for a PDL is the handling of symbols, i.e. how are symbols of functions relocated to point to the correct location in the object file.

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

For a PDL to support automatic linker generation, information about the relocation of symbols and relaxation of instructions is required.

This chapter should have provided the concepts and methods used by PDLs and retargetable compilers. The following chapters will build on this information and describe the PDL VADL and how it is used to generate a compiler backend for LLVM.

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

CHAPTER 3

Vienna Architecture Description Language

This chapter is dedicated to introduce the mixed PDL called Vienna Architecture Description Language (VADL), capable of automatic compiler-, simulator- and hardware generation. Every VADL specification is divided into ISA, MIA, ABI and a processor model combining these parts. In the following only the parts directly related to the compiler generation are presented, that is the ISA, the ABI and the processor model.

Please note that the VADL code examples provided in this chapter are only a current snapshot of the language specification. It is likely that the syntax will change in the course of this project.

3.1 ISA specification

The ISA specification contains the functional description of a processor architecture, providing instructions and structural resource elements like registers or register files to model the computational view. VADL provides four integral constructs for resource elements that can be used by instructions: registers, register files, memory components and immediate definitions.

3.1.1 Register definition

A register is a structural processor resource capable of holding values in form of bits. Registers can be given a name and define the size of bits they are able to hold.

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

The following snippet shows the definition of the **Y** register:

Listing 3.1: VADL register definition.

```
1 register Y : bit<32>
```

Every instruction can use this resource to store or read 32 bit in or from the register.

3.1.2 Program counter definition

The program counter is a special register needed by every VADL specification for program execution and can be defined in a similar way to a register definition:

Listing 3.2: VADL program counter definition.

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

3.1.3 Register file definition

Most RISC architectures use a high amount of registers to accomplish better execution speeds. However, defining lots of registers manually can be quite cumbersome. If registers share the same properties, they can be defined as a register file, combining several registers into one file, that can be accessed using indices. The next code snippet shows the definition of the register file **X**:

Listing 3.3: VADL ISA definition of the register file X.

```
1 [ X(0) = 0 ]
2 register file X: bit<5> -> bit<32>
```

In contrast to a register, two bit sizes separated by "->" are provided. The first range defines the index space, i.e. how many registers can be accessed using this register file ($2^5 = 32$). The second range defines the size of each register inside the register file and is equivalent to the bit size of a single register. The equation inside brackets over the register file definition, is called an annotation. Annotations can be used in several places to give a syntactically unified way of defining properties. This particular annotation defines the constant zero register $X(0)$, i.e. the register at position 0 of the register file is hardwired to the value 0.

3.1.4 Memory definition

A memory defines an addressable space of the processor, capable of holding values. Its definition is similar to a register file, however the keyword **memory** is used. The next

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

code snippet shows the general purpose memory of the **RISCV** implementation used to store and load results:

Listing 3.4: VADL ISA definition of the memory MEM.

```
1 memory MEM : bit<32> -> bit<8>
```

Memory definitions are important for the compiler generator to distinguish load and store operations from simple arithmetic or logical computations.

3.1.5 Instruction format definition

The instruction format defines the bit structure of an instruction and can be seen as blue print of instructions sharing this structure. The following snippet shows the instruction format definition of the **R_TYPE** of **RISCV**:

Listing 3.5: VADL ISA definition of the R_TYPE instruction format.

```
1 format R_TYPE : bit<32> =
2 { funct7 [31..25]
3 , rs2 [24..20]
4 , rs1 [19..15]
5 , funct3 [14..12]
6 , rd [11..7]
7 , opcode [6..0]
8 }
```

This instruction format is depicted of 32 bit and further divided into 6 sections (format fields), that can be used by concrete instruction definitions to refer to certain bits. Each format field in the definition has the following form: *name bits*. For example the name **funct7** refers to the bits 31 to 25 of an instruction implementing this format.

3.1.6 Immediate definition

An immediate definition can be used to define a custom bit sequence using the fields of an instruction format.

The following definition specifies the immediate **ImmediateI**, representing the 32 bit sign extension of the *imm* field of the instruction format **I_TYPE**:

Listing 3.6: VADL ISA definition of an immediate.

```
1 immediate ImmediateI : I_TYPE -> 32 = sext(imm, 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 Bibliothek Your knowledge hub WIEN

This particular immediate can only be used by instructions corresponding to the **I_TYPE** format.

3.1.7 Instruction definition

A concrete instruction definition can be built upon an instruction format and defines a computation that can be performed by the processor. The computation is defined by statements and expressions using resource elements and the format fields defined by the used instruction format. Note that these statements and expressions are less powerful compared to a general purpose language, because it should be possible for a compiler-, simulator- and hardware- generator to model their semantics.

The following snippet shows the **ADD** instruction definition, which uses the previously defined instruction format:

Listing 3.7: VADL ISA definition of the ADD instruction.

```
1 instruction ADD : R_TYPE = {
2    X(rd) := X(rs1) + X(rs2)
3 }
```

This definition contains the entire semantics description of an instruction and is therefore used for instruction selection pattern generation. In this example a simple addition (+) is performed, using the values stored in the register file **X** at positions *rs1* and *rs2*, storing the result back at the position *rd*. Note that the instruction definition does not define parameters and return types, this information is given implicitly. All instruction format fields that are used for a reading access like *rs1* and *rs2* are considered input parameters. Output parameters are handled analogous, i.e. format fields accessed during a write (LHS of an assignment) are considered output parameters. These parameters are important for the compiler generator, since they describe the variable parts of an instruction. They are filling the missing parts for the binary instruction encoding and are the operands used by instruction selection patterns.

More complex examples of instructions will be discussed in the implementation chapter.

3.1.8 Instruction encoding

Parts of an instruction like its opcode are always the same. These hard-coded values can be set with the instruction encoding construct. An instruction encoding can access the format fields of an instruction and assign a binary value to it.

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

The following construct sets the hard-coded portion of the **RISCV** instruction **ADD**:

Listing 3.8: VADL ISA definition of the ADD instruction encoding.

```
encoding ADD =
{ opcode = 0b011'0011
, funct3 = 0b000
, funct7 = 0b000'0000
}
```

The binary representation of this instruction can be illustrated as following:

funct7	rs2	rs1	funct3	rd	opcode
0000000	XXXXX	XXXXX	000	XXXXX	0110011

Table 3.1: Binary representation of instruction ADD.

An "X" is a placeholder for an actual bit value. Format fields that are depicted of "X" values, will be called instruction format parameters from now on. Fields with hard-coded values assigned, will be called instruction encoding values from now on. Instruction format parameters and instruction encoding values are used to generate the binary code emitting function for the compiler backend, but more information on this topic will be presented in the next chapter.

3.1.9 Assembly syntax

The last important construct of an instruction is its assembly syntax.

The following snippet shows the assembly syntax of the **ADD** instruction:

Listing 3.9: VADL ISA definition of the ADD instruction assembly syntax.

```
assembly ADD = {
   mnemonic ' ' rd ',' rs1 ',' rs2
}
```

The assembly syntax can be constructed using any format field provided by the instruction, using arbitrary strings to separate these values. The keyword *mnemonic* is referring to the name of the instruction, i.e. **ADD**. A programmer could use the **ADD** instruction with its assembly syntax like this: ADD 3, 4, 5.

The information provided by this construct is merely used to generate assembly code and to some extend to generate an assembler. However only the generation of assembly code is part of this thesis. The assembler will be discussed briefly in Chapters 4 and 6.

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

3.2 Assembly functions

Before starting with the next constructs, lets discuss the assembly construct they are trying to describe. For code reusing purposes it is possible to define functions, that can be called instead of writing the same instruction sequence over and over again. Because of the complexity of this concept, additional constructs like a call stack, call stack frame and program counter are used. These constructs are discussed more thoroughly in the following subsections.

3.2.1 Program counter

The machine executing the assembly code maintains the program counter, a register holding the address of the currently executing instruction. After this instruction has finished, it will be automatically incremented to execute the next instruction. In case of function calls, the handling of the program counter gets more complicated. Calling a function will jump to its location, which can be any arbitrary place of the assembly file. After the function has finished the program should continue executing where it left, i.e. the next instruction after the program counter before the function was called. But since the program counter will point to an arbitrary location this will not work. To overcome this problem, the so-called return address is used. The return address stores the address of the next instruction before the call was executed, so it can return to that point after the call has finished. It is the responsibility of the call and return instructions to maintain the correct value of the return address and adjusting the program counter accordingly.

3.2.2 Call stack

The call stack is a data structure used to support nested function calls and proper variable scoping, features that can be found in many higher level languages such as C. A call stack is made of so-called frames, each of which corresponds to a particular function call. Each frame stores information about the function call, like the values of a particular register before the call or during the call (local variables), the address of the previous frame and the return address.

Lets consider the following function *foo* in C:

Listing 3.10: C function call.

```
1 int add(int a, int b) {
2    return a + b;
3 }
4
5 int foo() {
6    return add(1, 2);
7 }
```


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

The corresponding *RISCV* assembly output of function *foo* looks like this:

Listing 3.11: Assembly function definition of foo.

```
1  # Allocate space on the stack for new stack frame
2  addi sp, sp, -16
3
4  # Spill callee-saved registers ra and fp
5  sw ra, 12(sp)
6  sw fp, 8(sp)
7
8  # Let the frame pointer point to the new frame
9  addi fp, sp, 16
10
11 # Load first function argument
12 addi a0, zero, 1
13
14 # Load second function argument
15 addi a1, zero, 2
16
17 # Jump to function definition of add
18 call add
19
20 # Restore callee-saved registers ra and fp
21 lw fp, 8(sp)
22 lw ra, 12(sp)
23
24 # Deallocate stack frame
25 addi sp, sp, 16
26
27 # Return from function
28 ret
```

The first steps performed by a function are part of its *prologue*, whereas the last steps correspond to the *epilogue*. Everything in between models the actual behavior of the function, the example being a call of the function *add*.

3.2.3 Calling a function

Before calling a parameterized function, its arguments must be loaded into special argument registers as can be seen in lines 12 and 15. In case that more arguments than argument registers are present, the values are stored on the stack. Normally the caller stores caller-saved registers to the stack prior to the function call.

Definition 3.2.1 (Caller-saved register)**.** A caller saved register does not preserve its value across calls. It is the responsibility of the caller to save and restore the values of such a register immediately before and after the call, if the values must be preserved.

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

3.2.4 Function prologue

The prologue can be seen as the push function of the call stack, which initializes a new stack frame and puts it on top of the stack.

Have a look at the first instruction of the previous code snippet. The **addi** instruction initializes a new frame on the stack with the size of 16 bytes. Note that this is done by decreasing the stack pointer, since the stack starts at high addresses and grows down. If it would grow upwards, the stack pointer would be increased. The next two instructions are used to persist the callee-saved registers *ra* (return address) and *fp* (frame pointer).

Definition 3.2.2 (Callee-saved register)**.** A callee-saved register must be preserved across calls. It is the callee's responsibility to save and restore the value of such a register.

Note that *ra* as defined by the ABI [ris] of **RISCV** is actually a caller-saved register, but gets unconditionally stored alongside callee-saved registers.

After persisting the current frame pointer, it is set to point to the end of the new frame. Frame and stack pointer are just a way of implementing the call stack. They are needed so the current active frame can be addressed. Strictly speaking it is only necessary to use a frame pointer if a frame contains variable length variables, which makes it impossible to refer to local variables using only the stack pointer [ALSU06].

3.2.5 Function epilogue

The epilogue can be seen as the pop function of the call stack and therefore as the inverse of the prologue, rewinding the last frame from the stack and thereby restoring the state before the function call. Lines 21 and 22 restore the callee-saved registers, whereas line 25 resets the stack-pointer, i.e. deallocating the frame. The last instruction returns from the function.

The following section will introduce the elements of VADL used to describe the function call process as part of an ABI specification.

3.3 ABI specification

An ABI of a processor architecture describes how parts of the ISA are used to perform computations on the call stack. It defines the calling conventions, special registers and the layout of the stack frame.

As of now, a VADL specification is not capable of describing every aspect of an ABI, only register alias names, calling conventions, basic relocation definitions and sequences that define how parts of a call are structured are supported. Therefore, the compiler generator assumes standard behavior for the missing parts where possible or omits a feature. Possible extensions that are needed for describing missing features will be discussed in Chapter 6.

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

3.3.1 Special register and pointer definitions

These definitions are simple assignments of registers defined in the ISA to predefined pointers and registers.

Currently these definitions are supported:

- stack pointer
- return address
- global pointer
- frame pointer
- return values
- function arguments
- caller saved registers
- callee saved registers

The special meaning of each assigned register is given by the components that use them (compiler generator, simulator generator, etc.).

The following code snippet shows the special register definitions of the 32 bit **RISCV** specification according to the **ILP32** ABI:

Listing 3.12: VADL ABI definition of special registers and pointers.

```
return address: X[1]
stack pointer: X[2] aligned by bit<128>
global pointer: X[3]
frame pointer: X[8]
return value: X[10..11]
function argument: X[10..17]
caller saved: X[1], X[5..7], X[10..17], X[28..31]
callee saved: X[2], X[8..9], X[18..27]
```

Note that the *global pointer* is currently not used by the compiler generator.

3.3.2 Register alias names

Register alias names can be assigned to any register defined by an ISA specification. These aliases can then be used to refer to registers. The only value for the compiler generator is that the assembler can produce more readable code.

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

The next snippet shows some of the register aliases defined by **ILP32**:

Listing 3.13: VADL ABI definition for register aliases.

```
register names = {
   X[0] -> ^zero
   X[1] -> ra
   X[2] -> sp
   X[3] -> gp
   X[4] -> tp
}
```

Every register can be assigned to one or more identifiers, that can be used interchangeable with the actual register. If more than one alias is needed more than one entry can be used inside the register name's block. The information provided by this construct is interesting for assembly code emission, assembler and disassembler, since programmers tend to use the more readable register alias instead of the actual name.

3.3.3 Call sequence

A call sequence defines a list of instructions that are needed to call a function. Most of the time only one instruction is needed, however ISAs like **RISCV** need two instructions in most cases.

The following snippet shows the instruction sequence needed to perform a call in **RISCV** if the address of the function does not fit in 20 bit:

Listing 3.14: VADL ABI definition of the call sequence.

```
call sequence( bit<32> symbol ) = {
   AUIPC 1, 0
   JALR 1, 0, 1
}
```

As one can see a **AUIPC** and **JALR** pair is needed. Note that the sequence needs a relocation to actually work, but was omitted, since relocation support is in a too early stage.

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

3.3.4 Return sequence

This sequence of instructions is needed to define how to return from a function call.

The following code is used for **RISCV**:

Listing 3.15: VADL ABI definition of the return sequence.

```
1 return sequence = {
2    JALR 1, 0, 1
3 }
```

3.3.5 Nop sequence

A NOP defines how to stall one processor cycle. Although this has nothing to do with an ABI, it is placed here until a better solution is found.

Listing 3.16: VADL ABI definition of the nop sequence.

```
1 nop sequence = {
2    ADDI 0, 0, 0
3 }
```

3.3.6 Relocation definition

A relocation is used in combination with a symbol. Global symbols of function definitions and constants are normally not known until link time. Therefore, an ABI defines relocations to calculate a symbol's address. These relocations can range from simple bit manipulations to a more complex control flow, which defines how to relax an instruction sequence.

For now only relocations that can be expressed through an expression are considered. The **RISCV** relocations **R_RISCV_HI20** and **R_RISCV_LO12_I** as defined by its ABI [ris] can be represented as following:

Listing 3.17: VADL relocation definitions.

```
1 relocation R_RISCV_HI20(bit<32> symbolValue) -> bit<32> = (
     symbolValue + sext(0x800, 32)) >> 12
2 relocation R_RISCV_LO12_I(bit<32> symbolValue) -> bit<32> =
     symbolValue - (R_RISCV_HI20(symbolValue) << 12)
```

Note that relocations cannot be expressed for Position Independent Code (PIC) right now, which would need additional support for Procedure Linkage Table (PLT) and Global Offset Table (GOT).

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

3.4 Processor model

The processor model combines all parts of a VADL description and gives them a contextual meaning. It is used as the entry point for every generation task performed by the VADL tool-chain.

The following snippet shows the definition of a generic processor implementing the **RISCV** standard **RV32I** using the **ILP32** ABI definition:

Listing 3.18: VADL ABI definition of a micro processor.

```
1 micro processor RV32I_ILP32 implements RV32I with ILP32
```

Note that it is possible for a processor to implement more than one ISA.

This concludes the most important VADL language constructs used for compiler generation. The next chapter will go into more detail on how they are used for compiler generation.

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.

CHAPTER 4

Implementation

This chapter is divided into four parts, describing the implementation of VADL, the tooling used for generating a C compiler and the analysis performed by the compiler generator along with the methods of actually generating the compiler.

4.1 VADL

The language implementation is build on top of Xtext[1], a DSL framework for Eclipse[2]. It uses a grammar and the LL(*) parser generator ANTLR[3] to generate a complete tool-chain for the described language: a lexer, a parser and an IDE. The IDE is build using the Rich Client Platform (RCP) framework, also used by the Eclipse JAVA IDE.

Using the generated artifacts of Xtext, VADL adds additional components, making it more easy to adapt and extend the existing code base. Passes are used to perform transformations and verification on data models, which can be combined and executed using the pass manager. These passes contain the logic, which transforms the VADL source code to the parse tree and AST, from which more concrete data models can be generated. The compiler generator currently builds upon the Common Resource Model (CRM), which is a thin wrapper of the parse tree. Note that this model was only used because of time constraints during the implementation phase and has proven to be insufficient for more complex generation tasks. It will therefore be replaced with a more suitable model in the near future.

[1] https://www.eclipse.org/Xtext/
[2] https://www.eclipse.org/
[3] https://www.antlr.org/

4.2 C Compiler & LLVM

The compiler generator is the main part of this thesis. It is responsible to generate a C compiler from a VADL processor model. This task is achieved by using the existing retargetable compiler framework LLVM and retargeting it according to a processor specification.

The general LLVM tool-chain for the C programming language family (clang) is composed of the following phases and components [cla]:

- Preproccessing of macros
- Parsing of pure C files
- IR generation
- Compiler backend
- Assembler
- Linker

The first three items are part of clangs front-end and hardly need any target specific information.

The remaining items are target specific and are needed to generate an executable. This work focuses on the ELF format for executables, which are run on a Linux[4] based operating system.

The compiler backend is defined and generated using a so-called target definition. This target definition handles all the necessary steps of the compiler backend like instruction selection, register allocation, assembly and object code emission.

LLVM provides several documents [llvf, llve] helping new developers to write a backend. In addition, the works of [Bur19, ES10, Gol17, CS] give a good introduction on the composition of a backend. Following this literature an LLVM target can be split roughly into code generation and code emission. Code generation focuses on instructions, machine functions, basic blocks and the SelectionDAG, whereas code emission uses assembly and object files with labels and directives. The SelectionDAG is the most important representation for the compiler generator and directly used for instruction selection, the following section will therefore give a short summary of its structure.

[4] https://www.linuxfoundation.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 Bibliothek Your knowledge hub WIEN

4.2.1 SelectionDAG

A SelectionDAG as defined by the official documentation [sela], represents the original program in form of a DAG and can be used for instruction selection. Its nodes correspond to operations and operands, each of which contain an opcode indicating the type of the node. The edges are defined as pair, containing the node it points to and the used value. Furthermore these values are annotated with their corresponding machine value type, e.g. i32 (32 bit integer) etc. There are two different kinds of dependencies incorporated into the SelectionDAG, that of data flows and control flows. Data flow edges only contain simple values, such as integers or floating points, whereas control flow edges are used for values of type "Other" and are represented by so-called "chains". "Chains" provide an ordering between operations having side-effects, such as branching instructions and function calls.

The following two subsections will give a short overview on the files depicting a target, omitting build files or currently unsupported files. The presented files follow the structure and conventions of existing backends, such as **RISCV** and **MIPS**.

4.2.2 Code generation

The code generation part consists of several target resource definitions, used to model target specific behavior and code generation. Resources consist of target registers and target instructions, in addition to special operand types such as immediate types. Note that TableGen files have the "td" extension.

TargetRegisterInfo.td Defines the registers provided by the target machine.

TargetCallingConv.td This file defines the calling conventions using the register definitions of **TargetRegisterInfo.td**. It is possible to define registers that are used for arguments, return types, callee-saved registers, etc.

TargetInstrFormat.td Instruction formats combine properties of similar instructions, such as operands, their binary structure and assembly syntax, acting as blue prints for concrete instruction definitions. Note that this file serves merely the purpose of code reuse and adding readability for hand-written targets. For code generation it is useful to directly define this information for each instruction in the next file.

TargetInstrInfo.td This file contains the concrete instruction definitions using the formats defined in **TargetInstrFormat.td** and instruction selection patterns.

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

Target.td This file includes the **TargetRegisterInfo.td**, **TargetInstrInfo.td** and **TargetCallingConv.td** files to form a target and processor model. It is used to generate several C++ include files using the TableGen tool. The most important files generated are the following:

- **TargetGenAsmWriter.inc** Implements convenience methods to print the assembly syntax of instructions.
- **TargetGenCallingConv.inc** Defines the calling conventions of the target.
- **TargetGenDAGISel.inc** Implements the instruction selection code.
- **TargetGenInstrInfo.inc** Implements the C++ representation of instructions.
- **TargetGenRegisterInfo.inc** Implements the C++ representation of registers.
- **TargetGenMCCodeEmitter.inc** Implements convenience methods to print the binary representation of instructions.
- **TargetGenSubtargetInfo.inc** Defines target information.
- **TargetGenMCPseudoLowering.inc** Implements lowering code from pseudo instructions into target instructions.

TargetRegisterInfo.cpp & TargetRegisterInfo.h These files provide additional information about registers of a target. It defines constant registers, reserved registers, the frame pointer and how to eliminate a frame index. To do this it must include the generated **TargetGenRegisterInfo.inc** file.

TargetInstrInfo.cpp & TargetInstrInfo.h These files define special behavior, such as spilling and restoring values from the stack or copying physical registers, using the instructions defined in **TargetGenInstrInfo.inc**.

TargetISelDAGLowering.cpp & TargetISelDAGLowering.h These files handle the lowering of the SelectionDAG into a legal form, which can be used for instruction selection. That includes the lowering of function call sequences into target specific call pseudo nodes defined in **TargetInstrInfo.td**.

TargetISelDAGToDAG.cpp & TargetISelDAGToDAG.h These files handle instruction selection of the legalized SelectionDAG, using the selection method of **TargetGenDAGISel.inc**. Only special instruction selection rules that cannot be generated, must be defined here.

TargetFrameLowering.cpp & TargetFrameLowering.h These two files handle the frame lowering and the function prologue and epilogue.

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

TargetMCInstLower.cpp This is a convenience class, transforming the code generation representation of instructions into the code emission representation.

TargetAsmPrinter.cpp The asm printer is used for lowering instructions from the code generation abstraction to the code emission abstraction, used by the next part. For this it uses the helper class **TargetMCInstLower.cpp**.

TargetObjectFile.cpp & TargetObjectFile.h Defines the object file used for code emission in the next part.

TargetMachine.cpp & TargetMachine.h These files register the target machine to LLVM, so it can be used by various tools like **llc**. Note that tools like **clang** handle target registration elsewhere. The target machine defines the data layout [llvb], subtargets and LLVM passes that should be supported.

4.2.3 Code emission

Files used for code emission are located in the **MCTargetDesc** folder.

TargetFixupKinds.h A fixup is used by LLVM to patch certain values during assembly or linking time. It can be used to implement relocations, relaxations and assembler modifiers. This file defines the supported fixups of the target.

TargetObjectWriter.cpp There are different object writers for every supported binary format of LLVM. The main purpose of this file is to implement the lowering from fixups to relocations.

TargetStreamer.cpp & TargetStreamer.h The target streamer handles the code emission of both assembly and object code.

TargetAsmBackend.cpp & TargetAsmBackend.h The asm backend instantiates the object writer that is used to emit code and defines how fixups are applied. Additionally, everything related to assembling is defined here.

TargetInstPrinter.cpp & TargetInstPrinter.h The instruction printer defines how instructions of the target are printed to an assembly file. Most of the logic is already generated from **TargetInstrInfo.td** and must only be included here. If operands or instructions need special handling, they are handled here as well.

TargetMCAsmInfo.cpp & TargetMCAsmInfo.h Here meta-information about the assembly can be defined, such as code pointer size, comment string, debug-, exception-support, etc.

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

TargetMCCodeEmitter.cpp The code emitter is used to emit the binary instruction encoding to the object file. Most of the logic for this is already generated from **TargetInstrInfo.td** using TableGen. However special cases need to be taken care of here.

TargetMCExpr.cpp & TargetMCExpr.h These files are used to define special expressions for fixups and assembly symbols.

TargetMCTargetDesc.cpp & TargetMCTargetDesc.h Registers the phases of the backend used for code emission.

The following sections of this chapter will discuss how these files are generated.

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.

4.3 Compiler Backend generation

The implementation of the compiler generator focuses on the parts of the target that do not correspond to the assembler, since most work for the assembler and the linker will be done in another project part.

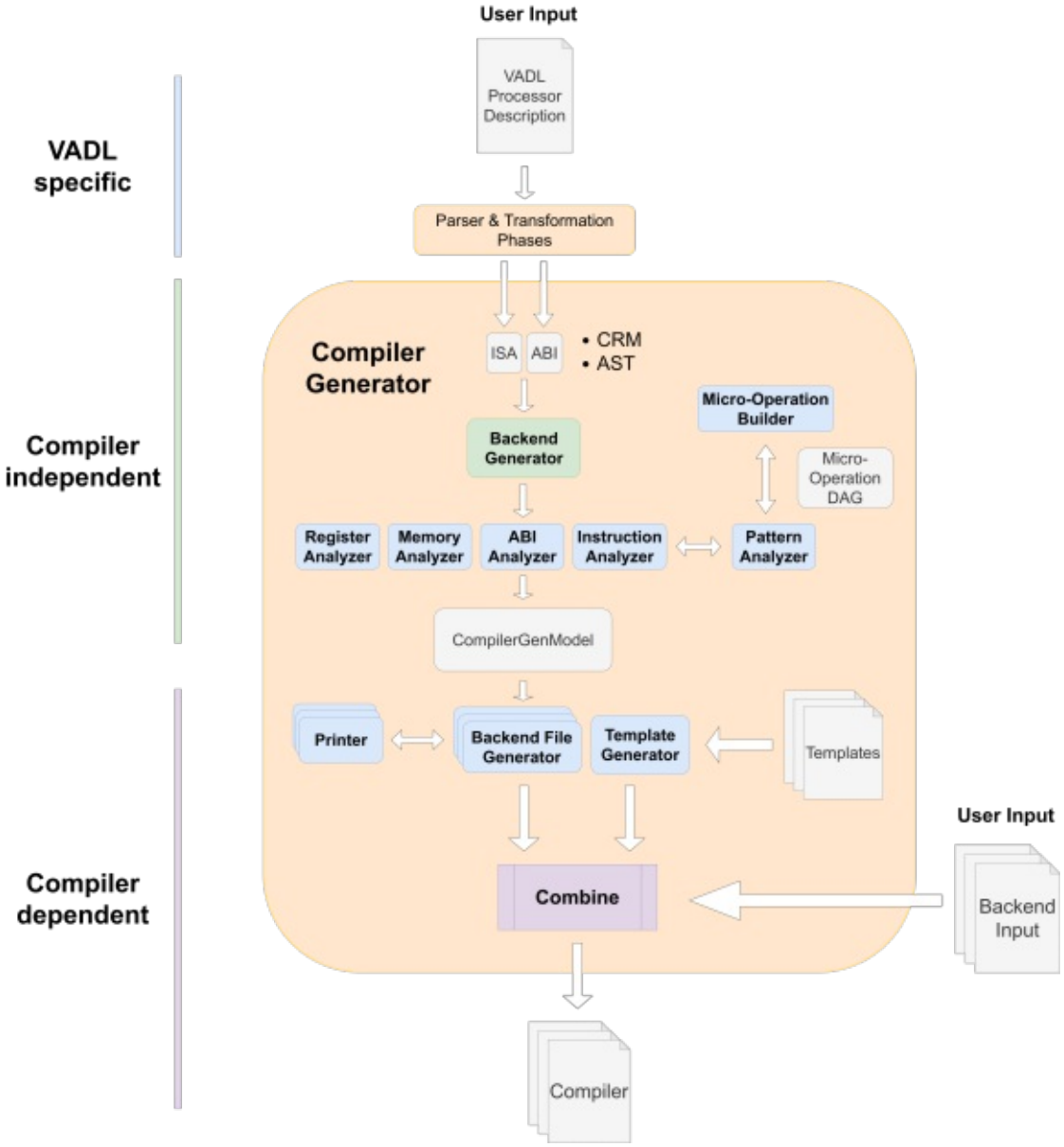


Figure 4.1: Architectural view of the compiler generator

Figure 4.1 shows the overall structure of the compiler generator. It is split into two

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

parts to keep the core logic independent of a concrete compiler implementation. The first part consists of static analysis of a VADL specification to extract the information, that is not explicitly given, but can be derived implicitly. Using the CRM and the extracted information, a compiler generator internal data model (CompilerGenModel) is constructed. This data model should be completely decoupled from any particular compiler implementation like LLVM.

On this data model a compiler framework specific implementation, for now only LLVM, is used to generate the compiler artifacts described in the previous section. This is accomplished by so-called file generators and printers, which take the implementation independent data model and generate implementation dependent files. Each file generator inherits from **BackendFileGenerator**, which takes the **CompilerGenModel** as input and writes content depending on this data in one file of the compiler backend. The convention is that each file generator is named like the file it generates with an additional prefix like LLVM, indicating its compiler infrastructure. File generators are only used if the code that needs to be generated is more complex. Note that some parts of the LLVM backend are treated as boilerplate for now, because of missing information in VADL. Each boilerplate file, is provided as template, containing valid source code annotated with special characters that get replaced with actual values.

4.4 Analysis and data creation

Analysis on the CRM is conducted to infer information, that is implicitly given by a VADL specification and necessary for the compiler generation. The following analyzers are currently implemented:

4.4.1 Register analyzer

Registers are the simplest resource of a processor description and contain all the information already explicitly present in the VADL specification. This information is however split among different parts of the specification and may be encoded in an impractical way. The register analyzer collects information about registers and register files from all locations, and generates the compiler generator's internal register definitions. Each data entry constructed can either be a single register with a name, several aliases and bit length, or a register file containing a list of registers. A constant register is a special type adding a hardwired constant value to a simple register. Most of this information is already provided by the CRM, the information about aliases is collected from the ABI definition.

4.4.2 Memory analyzer

A memory definition contains its name, input and output bit length. It is currently only used as operand type for instructions to determine, whether it is a store or load instruction. All the information can be directly used from the CRM.

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

4.4.3 ABI analyzer

This analyzer extracts all special registers from an ABI specification, using the same registers defined by the register analyzer and wraps them into a dedicated ABI model to preserve the special meaning of them. The semantics described inside the ABI specification are handled by the pattern analyzer.

4.4.4 Instruction analyzer

The instruction analyzer operates on the ISA specification and constructs an internal model for each instruction, combining the format, binary encoding and assembly syntax. Thus making it easier for the compiler generator to generate one artifact instead of multiple ones. Furthermore the parameters of an instruction are inferred and added to this model. Consider the following example:

Listing 4.1: VADL definition of the ADD instruction.

```
1 instruction ADD : R_TYPE = {
2    X(rd) := X(rs1) + X(rs2)
3 }
```

Here the instruction fields **rd**, **rs1** and **rs2** are used to access registers inside the register file **X** and are categorized into read or write. An access wraps the data type of the accessed resource, for example the register file **X** with its bit length and the instruction format field as index. These resource accesses are used as parameters of the instruction and to determine the particular kind of the instruction.

Currently only four kinds of instructions are known to the instruction analyzer:

Branch instructions Instructions that assign a value to the program counter (LHS of an assignment), alter the program execution and are therefore considered to be of kind branch.

Load instructions Instructions that read a value from a memory location (RHS of an assignment), are considered to be of kind load.

Store instructions Instructions that assign a value to a memory location (LHS of an assignment), are considered to be of kind store.

General instruction Every other instruction, i.e. neither branch, load or store instructions are considered regular and do not need special handling for now.

The previous example has shown the use of registers, if however an immediate value is used instead of a register, additional information must be obtained. Have a look at the **ADDI** instruction using the immediate value **ImmediateI**:

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

Listing 4.2: VADL definition of the ADDI instruction.

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

The immediate value is defined using VADL's immediate mechanism:

Listing 4.3: VADL definition of the ImmediateI immediate.

```
1 immediate ImmediateI : I_TYPE -> Word = sext(imm, 32)
```

This immediate definition uses the **imm** field of the instruction format **I_TYPE**, to form its 32 bit sign extension. Note that all format fields used inside an immediate definition, that are also referred by the assembly syntax (assembler parameters), are considered instruction parameters. It is necessary to use the bit length of the immediate definition for parameters, since the calculation performed by the instruction requires the type specified by the immediate (32 bit) and not the type of **imm**, i.e. 12 bit. In terms of the compiler, this means that the operation used during instruction selection must contain an immediate value matching the sign extension of **imm** to be selected. The part of the immediate corresponding to **imm**, will then be used for the binary encoding of **ADDI**. This information is encoded into so-called instruction operand types, containing the name of the immediate definition and its bit composition, used to generate an operand type and predicate for instruction selection.

The semantics of an instruction directly corresponds to instruction selection patterns and is therefore handled by the pattern analyzer.

4.4.5 Pattern analyzer

The pattern analyzer generates instruction selection patterns, either directly from the semantics section of an instruction definition or by using algebraic transformations on instructions.

Given that the statements and expressions used in the semantics definition of an instruction can be mapped to operations supported by a compiler, one-to-one (1 statement) and some many-to-one (>1 statements) patterns can be generated without additional analysis.

The following subsections will introduce the data model and methods used to generate patterns from instruction definitions.

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

Micro-operation DAG

This model is inspired by the micro-operations used by Ceng, Hohenauer, Leupers et al. [HL10] [CHL+05]. They also establish the relations of said micro-operations and target instructions, e.g. one-to-one, one-to-many and many-to-one, which will be used throughout this work. The name of micro-operations is still used, since the idea of mapping the semantics of instructions into smaller operations that can be performed by a compiler is the same. However the operations used here are tailored around the statements and expressions supported by VADL and try to maintain a surjective relation to the nodes of a SelectionDAG to minimize conversation overhead.

Now let this version of a micro-operation DAG m of an instruction i be a tuple $\langle M, E \rangle$, where M is the set of supported micro-operations (vertices) and E the set of edges connecting micro-operations with each other. Note that terminal nodes correspond to expressions or statements and leaf nodes to simple resource accesses.

An edge $e = \langle j \in \mathbb{N}, m_1, m_2 \rangle \in E$ connects the micro-operations m_1 and m_2, encoding the information that m_2 is the j-th operand of m_1. The sets M and E are determined according to the statements of instruction i.

Let S be the set of statements defined in i. $\forall s_k \in S$ construct a micro-operation m_k and the operands according to the following rules.

To keep the rule definitions simple the following assumptions are made:

- Let bindings are replaced with the actual value
- Special cases are omitted
- Micro-operation nodes may contain only a subset of the actual information of the implementation

Assignment with binary expression If s_k is an assignment $a = \langle lhs, rhs \rangle$, where rhs corresponds to a binary expression and lhs corresponds to a write access to a register $r = \langle name, bits \rangle$ or register file $rf = \langle name, bits, index \rangle$, construct the following micro-operations and operands:

- $m_k = op$
- $m_r = \langle name, bits \rangle$ or $m_{rf} = \langle name, bits, index \rangle$
- $e_{k,1} = \langle 1, m_k, m_r \rangle$ or $e_{k,1} = \langle 1, m_k, m_{re} \rangle$.
- Left operand m_a of m_k by applying the rules
- Right operand m_b of m_k by applying the rules
- $e_{k,2} = \langle 2, m_k, m_a \rangle$

TU Bibliothek Your knowledge hub WIEN
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.

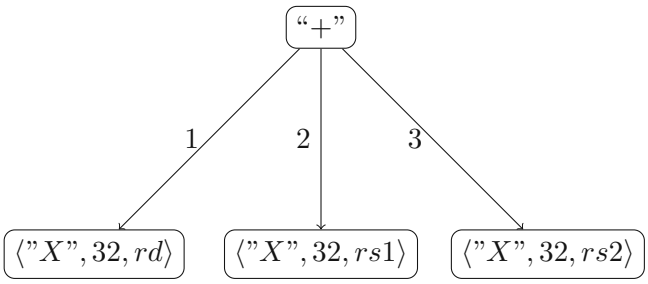
- $e_{k,3} = \langle 3, m_k, m_b \rangle$

Example:

Listing 4.4: Assignment with binary expression in VADL.

```
1 X(rd) := X(rs1) + X(rs2)
```

- $m_1 = \text{"}+\text{"}$
- $m_{rf} = \langle \text{"}X\text{"}, 32, rd \rangle$
- $e_{1,1} = \langle 1, m_1, m_{rf} \rangle$
- $m_a = \langle \text{"}X\text{"}, 32, rs1 \rangle$
- $m_b = \langle \text{"}X\text{"}, 32, rs2 \rangle$
- $e_{1,2} = \langle 2, m_1, m_a \rangle$
- $e_{1,3} = \langle 3, m_1, m_b \rangle$



Assignment to memory If s_k is an assignment $a = \langle lhs, rhs \rangle$, where lhs corresponds to a write access of a memory $m = \langle name, bits, index \rangle$, construct the following micro-operations and operands:

- $m_k = store$
- $m_{mem} = \langle name, bits, index \rangle$
- $e_{k,1} = \langle 1, m_k, m_{mem} \rangle$
- m_{rhs} by applying the rules
- $e_{k,2} = \langle 2, m_k, m_{rhs} \rangle$

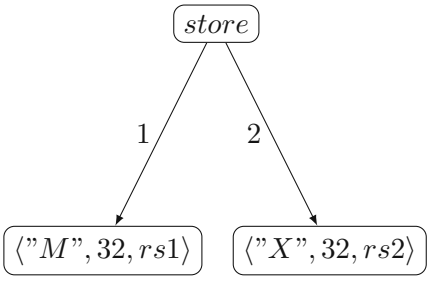
Example:

Listing 4.5: Assignment to memory in VADL.

```
1 MEM<4>(rs1) := X(rs2)[31..0]
```


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.

- $m_1 = store$
- $m_2 = \langle "M", 32, rs1 \rangle$
- $m_3 = \langle "X", 32, rs2 \rangle$
- $e_{1,1} = \langle 1, m_1, m_2 \rangle$
- $e_{1,2} = \langle 2, m_1, m_3 \rangle$



Assignment from memory If s_k is an assignment $a = \langle lhs, rhs \rangle$, where rhs corresponds to a read access of a memory $m = \langle name, bits, index \rangle$ and lhs corresponds to a write access to a register $r = \langle name, bits \rangle$ or register file $rf = \langle name, bits, index \rangle$, construct the following micro-operations and operands:

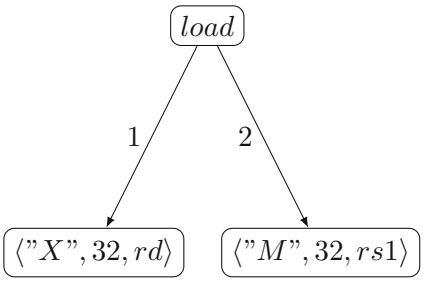
- $m_k = load$
- $m_{mem} = \langle name, bits, index \rangle$
- $m_r = \langle name, bits \rangle$ or $m_{rf} = \langle name, bits, index \rangle$
- $e_{k,1} = \langle 1, m_k, m_{mem} \rangle$
- $e_{k,2} = \langle 2, m_k, m_r \rangle$ or $e_{k,2} = \langle 2, m_k, m_{re} \rangle$.

Example:

Listing 4.6: Assignment from memory in VADL.

```
1 X(rd) := MEM<4>(rs1)
```

- $m_1 = load$
- $m_2 = \langle "X", 32, rd \rangle$
- $m_3 = \langle "M", 32, rs1 \rangle$
- $e_{1,1} = \langle 1, m_1, m_2 \rangle$
- $e_{1,2} = \langle 2, m_1, m_3 \rangle$



If/else: Setting a flag If s_k is an if/else construct $c = \langle condition, if, else \rangle$ with both if and else blocks containing a single assignment, such that $if = \langle lhs_1, v_1 \rangle$, $else = \langle lhs_2, v_2 \rangle$, $lhs_1 = lhs_2$, $v_1 \neq v_2$, $v_{1,2} \in \{0, 1\}$ and lhs is a write access to a register or register file, construct the following micro-operations and operands:

- $m_k = set$

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 Your knowledge hub WIEN

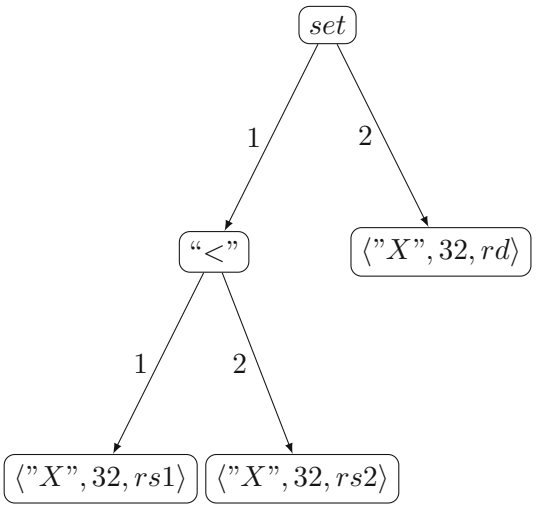
- $m_{condition}$ by applying the rules
- $m_r = \langle name, bits \rangle$ or $m_{rf} = \langle name, bits, index \rangle$
- $e_{k,1} = \langle 1, m_k, m_{condition} \rangle$
- $e_{k,2} = \langle 2, m_k, m_r \rangle$ or $e_{k,2} = \langle 2, m_k, m_{re} \rangle$.

Example:

Listing 4.7: If/else: Setting a flag in VADL.

```
1 if (unsigned(X(rs1)) < unsigned(X(rs2))) then
2    X(rd) := 1
3 else
4    X(rd) := 0
```

- $m_1 = set$
- $m_2 =$ ”$<$”
- $m_3 = \langle$”X”$, 32, rs1 \rangle$
- $m_4 = \langle$”X”$, 32, rs2 \rangle$
- $m_5 = \langle$”X”$, 32, rd \rangle$
- $e_{1,1} = \langle 1, m_1, m_2 \rangle$
- $e_{1,2} = \langle 2, m_1, m_5 \rangle$
- $e_{2,1} = \langle 1, m_2, m_3 \rangle$
- $e_{2,2} = \langle 2, m_2, m_4 \rangle$



If: Setting the program counter If s_k is an if block $if = \langle condition, statement \rangle$ containing a single assignment, such that $statement = \langle PC, rhs \rangle$ and rhs contains an immediate inside the expression, construct the following micro-operations and operands:

- $m_k = branch$
- $m_{condition}$ by applying the rules
- m_{rhs} by applying the rules
- $e_{k,1} = \langle 1, m_k, m_{condition} \rangle$
- $e_{k,2} = \langle 2, m_k, m_{rhs} \rangle$

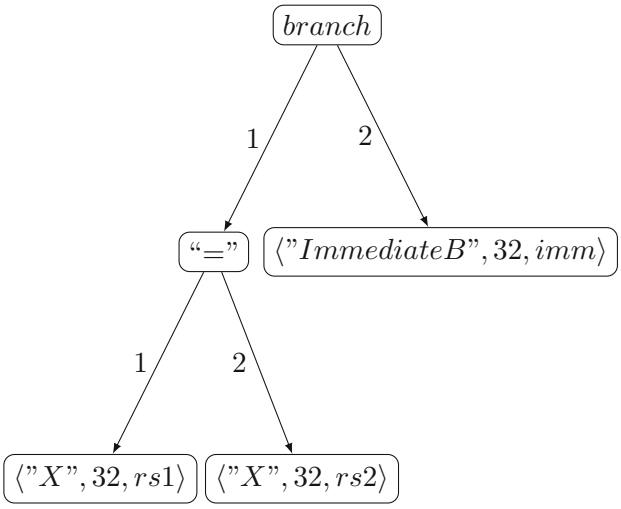
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Example:

Listing 4.8: If/else: Setting the program counter in VADL.

```
1 if (X(rs1) = X(rs2)) then
2    PC := PC + ImmediateB
```

- $m_1 = branch$
- $m_2 =$ ”$=$”
- $m_3 = \langle$”X”$, 32, rs1\rangle$
- $m_4 = \langle$”X”$, 32, rs2\rangle$
- $m_5 = \langle$”$ImmediateB$”$, 32, imm\rangle$
- $e_{1,1} = \langle 1, m_1, m_2\rangle$
- $e_{1,2} = \langle 2, m_1, m_5\rangle$
- $e_{2,1} = \langle 1, m_2, m_3\rangle$
- $e_{2,2} = \langle 2, m_2, m_4\rangle$



Expression For an expression $expr$ contained in s_k build an expression node as follows.

If $expr = \langle op, rhs, lhs\rangle$ $op \in \{+, -, *,\ div,\ rem,\ lshift,\ rshift,\ and,\ or,\ xor\}$:

- $m_1 = op$
- m_2 by recursively applying the expression rule
- m_3 by recursively applying the expression rule
- $e_{1,1} = \langle 1, m_1, m_2\rangle$
- $e_{1,2} = \langle 2, m_1, m_3\rangle$

If $expr = \langle name, bits\rangle$ is a register access, construct $m_r = \langle name, bits\rangle$. If $expr = \langle name, bits, index\rangle$ is a register file, construct $m_{rf} = \langle name, bits, index\rangle$. If $expr$ is a memory access, construct m_{mem} analogous to the register file.

Unsupported statement pattern Any other statement pattern currently not supported will construct a dummy node $m_k = dummy$ with an operand $e_{parent,j} = \langle j, m_{parent}, m_k\rangle$ if m_k is the j-th child of m_{parent}. These dummy nodes are used to propagate a missing implementation to the user and will either skip the generation part

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

of the instruction i containing such a dummy node or add a comment in the generated backend artifact.

Note that these rules and micro-operation definitions only account for a handful of possible statement patterns, since VADL is rapidly evolving and new statements are being added constantly.

Instruction selection pattern

An instruction selection pattern p can be represented as a tuple of two micro-operation DAGs $\langle m_1, m_2 \rangle$, where m_1 represents the target-independent operation and m_2 represents the target-dependent instruction. Both m_1 and m_2 use the same micro-operation data structure, the difference is given by the type of the operands and operations. Where m_1 uses a micro-operation associated to "+" with a virtual register and immediate placeholder, m_2 might use an instruction **ADDI** with the register file **X** and immediate type **ImmediateI**. From now on lets refer to the LHS of a pattern, i.e. m_1 in the above example as micro-operation matcher. A micro-operation matcher describes the set of valid values for a specific micro-operation like "+", "-" or a 32 bit register. The RHS of a pattern will be referred to as matched target representation and consists of one or more instruction call micro-operation nodes using concrete resource types.

Generating instruction selection patterns

Instruction selection patterns are used to map a higher level programming language such as C to target specific instructions of a hardware target. For a complete compiler implementation, all language features must be mapped to hardware instructions. However another level of abstraction in form of an IR can be added to support a variety of programming languages instead of just a single one. Therefore, every supported language must be lowered into the IR form first, as already done by most compiler frameworks, including LLVM. That IR can then be used for instruction selection. Note that the generated IR usually differs, depending on which source language was used. Although programming languages share some constructs, it is vital to match every IR constellation emitted by such a language. To keep things simple, this work focuses on matching LLVM instructions generated by the C language front-end. LLVM uses the SelectionDAG as IR during instruction selection, for which instruction selection patterns must be generated. The idea behind this is to represent parts of the SelectionDAG, that must be matched, using a micro-operation DAG (micro-operation matcher).

The compiler generator currently implements the following matchers:

- Arithmetic operations (+, -, etc.)
- Logical operations (and, or, etc.)
- Load/Store (sext, zext)

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

- Set condition code (set if equal, etc.)
- Conditional branches (branch if equal, etc.)
- Integer register operands
- Immediate operands

Simple one-to-one and many-to-one patterns can be directly generated from instruction definitions. Each instruction semantics will be translated into a micro-operation DAG according to the already mentioned rules and corresponds to a specific part of the SelectionDAG that can be matched. An additional step verifies that the resulting micro-operation DAG is needed to cover an IR operation of the compiler, by matching the micro-operations of the instruction with the set of micro-operation matchers.

Definition 4.4.1 (Matching)**.** Two micro-operations m_1 and m_2 match, if they have the same tree structure. For operands, i.e. leaf nodes, the value of m_2 must be valid for the type specified in m_1. Note that matcher nodes of m_1 can also match sub tree structures of m_2.

Matching is performed on a list of micro-operation matchers. A successful match constructs an instruction selection pattern $p = \langle m, m_i \rangle$ for the instruction i, where m_i is an instruction call micro-operation of i and m is the matched element. The operands of m_i are the same as of m.

This approach is quite capable of generating both simple and more complex patterns, however it is not possible to generate one-to-many patterns as already stated by [CHL+05]. One-to-many patterns need additional information about how computations correlate to target instructions. A rather simple method for providing this information and generating one-to-many, as well as additional one-to-one and many-to-one patterns is provided by transformations [CHL+05].

Semantics transformation

Before discussing the generation of one-to-many patterns, consider following motivating example as already mentioned by [CHL+05, HL10]. Computer programs often contain operations that cannot be directly matched with the instructions of an ISA, but have a supported equivalent representation. The **neg** operation, used to get the negative representation in 2' complement, can be represented as $(not\ x) + 1$, reassembling an one-to-many pattern.

Arithmetic relations like the one above can be used to generate additional instruction selection patterns.

Lets consider a transforming relation as used by the VADL compiler generator, to be a unary function $t : M \to M$, which takes a micro-operation DAG and transforms it into

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.

another micro-operation DAG. Function t should only be applicable to micro-operation nodes that allow it to successfully trigger. This applicability check is performed by an unary function $a : M \rightarrow Bool$, using the set $n \in \mathcal{P}(M)$ of micro-operations needed for building the new micro-operation. Therefore, a transformation is represented as triple $T = \langle t, a, n \rangle$.

The following algorithm is currently used for transforming micro-operation DAGs:

Algorithm 4.1: Transformation of a micro-operation DAG.

```
1  Function transform(M: MicroOperationDAG, Transformations:
     List<Transformation>, supportedOperations: List<MicroOperation>) :
     List<MicroOperationDAG> is
2      if children(M) = ∅ then
3          return List.of(M)
4      end
5      childVariants ← newList
6      foreach child ∈ children(M) do
7          childVariants.add(transform(child))
8      end
9      variants ← newList
10     permutations ← permute(childVariants)
11     foreach permutation ∈ permutations do
12         newM ← copy(M)
13         newM.children ← permutation
14         foreach transformation ∈ Transformations do
15             if applicable(transformation, newM) and
                neededOperations(transformation, newM) ∈ supportedOperations
               then
16                 toBeTransformed ← copy(newM)
17                 variants.add(transformation.transform(toBeTransformed))
18             end
19         end
20     end
21     return variants
22 end
```

This algorithm follows a bottom-up approach for inplace DAG rewriting based on patterns

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

(transformations), and can therefore be seen as form of tree pattern matching [HO82] without additional techniques used for covering [Pro92]. For each permutation of child variants obtained by transformations a new copy of the current DAG is generated. The children of this copy are substituted with the permutations, i.e. for the transformations t_1, t_2 and t_a, t_b of the first and second child, respectively, the permutations t_1, t_a, t_1, t_b, t_2, t_a and t_2, t_b are formed. Any applicable transformation is applied on the copy and the result is added to the return value. Transformations are only applied once for each node, to ensure the termination of this algorithm, since a transformation could introduce new transformations that can be applied, leading to a possible endless sequence of transformations.

For each already generated instruction selection pattern, the transformation is applied on the target independent part (LHS) and checked whether this exact micro-operation DAG is already handled by another pattern - if not - it is added to the existing patterns. Should operands change, i.e. a variable is assigned a concrete value, then the RHS will be adapted accordingly as well. Note that changes of operands, i.e. leaf nodes, must also be propagated on the transformed micro-operation DAG itself to ensure a correct transformation. Consider a transformation setting the value of the operand $rs1$ to zero in some part of the DAG, but the same transformation doesn't trigger in another part also containing $rs1$. Now the transformed micro-operation DAG is in an inconsistent state, containing a variable that won't be present for matching.

This mechanism is needed to generate one-to-many operations, but can also be used to generate variations of existing one-to-one and many-to-one patterns. Consider a simple example, where the instruction **setlt**, which sets a bit if its left operand is less than its right operand, is given but not its counterpart **setgt**, as is the case for RISCV [WLPA16]. **Setgt** can be represented by **setlt** if both operands are swapped, i.e. $x < y$ implies $y > x$, and can therefore be matched by **setlt y x**. This example should illustrate how simple - yet useful patterns - can be generated using existing patterns and simple algebraic properties.

Now consider a more complex example, which uses a many-to-one pattern and transforms it to a one-to-one pattern and thereby introduces a useful variant in case the many-to-one pattern is to specific. The following pattern is generated for a load instructions **LH** of **RV32IM**:

Listing 4.9: TableGen instsruction selection pattern for sextloadi16.

```
1 def : Pat<(sextloadi16 (add X:$rs1, ImmediateI:$imm), (LH X:$rs1,
      ImmediateI:$imm)>;
```

This pattern is matched if the result of an addition (+) of a value inside a register and an immediate value is used as address to load a value from memory, using the target-specific instruction **LH**. This directly mirrors the semantics of the **LH** instruction, however it is rather uncommon that the SelectionDAG contains such an operation sequence and

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

therefore this pattern will not be considered often. It is more likely that a value from memory is loaded directly. By applying a simple transformation this pattern can be used to match more common input. Since the addition (+) has the zero element 0, adding 0 to some variable x will return variable x. The rule of the zero element can be used to transform the many-to-one pattern from above into the following one-to-one pattern:

Listing 4.10: Transformed TableGen instruction selection pattern for sextloadi16.

```
1 def : Pat<(sextloadi16 X:$rs1), (LH X:$rs1, 0)>;
```

This pattern allows to match an arbitrary 16 bit load from memory and helps to increase completeness of the compiler generator.

These examples should have provided motivation and given an idea of how useful an adequate transformation system for generating additional instruction selection patterns can be.

4.5 LLVM backend generation

Building upon the data provided by the previous section, a LLVM backend is generated. The following subsections will provide more details on the crucial parts of the backend.

4.5.1 Register definitions

The register and register files emitted by the register analyzer are used to generate register definitions in TableGen.

Lets have a look on how a register is defined inside a LLVM backend:

Listing 4.11: TableGen definition of a hardware register.

```
1 def X0 : Register<"X0"> {
2    let Namespace = "TARGET_NAME";
3    let HWEncoding{4-0} = 0;
4    let AltNames = ["zero"];
5 }
```

The above code snippet shows the definition of the register **X0**. The identifier of it can be used to refer to the register inside the backend and the string representation is used for the assembler output. There are several additional things that can be appended to the definition, but for now only the register aliases and hardware encoding are defined. The hardware encoding is used during code emission to address this exact register in the binary instruction encoding, whereas register aliases are used to output more readable assembler output.

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

Note that registers can only be defined individually, which is is also the case for register files. The previously defined register **X0** is part of the register file **X**, however LLVM does not use register files, but rather register classes to group related registers together. A register can be added to a register class as following:

Listing 4.12: TableGen definition of a register class.

```
1 def X : RegisterClass<"TARGET_NAME", [i32], 32, (add X0)>;
```

The first parameter defines the namespace (name of the generated target) in which the register class is valid. The second parameter is the list of value types of the included registers of a class, in this case a 32 bit integer. The next parameter is the alignment, which is followed by the list of registers that belong to the register class. Note that here the list is defined using the **add** followed by a list of registers (separated by ",") enclosed by two braces.

These register definitions are contained in *TargetRegisterInfo.td*. Additional information about constant registers is provided in *TargetRegisterInfo.cpp* and *TargetRegisterInfo.h*, but will be omitted here since it is rather simple C++ code.

4.5.2 Calling conventions

Calling conventions are defined in *CallingConv.td* using TableGen. Callee-saved registers, which will be used to generate spilling and restoring code can be defined like this:

Listing 4.13: TableGen definition of callee-saved registers of ILP32.

```
1 def CSR_ILP32 : CalleeSavedRegs<(add X2, X8, X9, X18, X19, X20, X21,
    X22, X23, X24, X25, X26, X27)>;
```

Here the callee-saved registers according to RISCV's ILP32 ABI are defined. It is convention to start these definitions with a **CSR_** prefix, which stands for Callee Saved Registers. As one can see, the new definition inherits from the **CalleeSavedRegs** class, which takes the list of callee-saved registers as input. All registers, previously defined, can be accessed using their identifiers name.

LLVM only supports spilling callee-saved registers natively. Caller-saved registers must be handled individually, without the help of the framework. Sometimes, special registers like the return address are defined as caller-saved, as for example by *RISCV*. These caller-saved registers will be treated as callee-saved registers, spilling them unconditionally in the prologue. This approach reduces the efforts for the compiler generator, since only little adjustments to the spilling mechanism are needed. In the future a dedicated definition should be used to support proper handling of caller-saved registers.

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

The next listing shows how to define a rule, assigning function parameters of a certain type to argument registers.

Listing 4.14: TableGen definition of calling conventions

```
1 def CC_ILP32 : CallingConv<[
2    CCIfType<[i32], CCAssignToReg<[X10, X11, X12, X13, X14, X15, X16,
        X17]>>
3 ]>;
```

This definition starts with a prefix **CC**, which stands for Calling Convention and inherits from **CallingConv**, taking a list of conditional statements, i.e. **CCIfType**. This condition takes a list of types (32 bit integer) for which the action **CCAssignToReg** should be applied, assigning the values to a list of registers. The return values can be assigned in a similar way:

Listing 4.15: TableGen definition of return calling conventions

```
1 def RetCC_ILP32 : CallingConv<[
2    CCIfType<[i32], CCAssignToReg<[X10, X11]>>
3 ]>;
```

The only difference is the prefix **RetCC**. It is important that the names of the definitions follow this pattern, so code will be generated correctly. The information for these definitions is directly contained in the data model generated by the ABI analyzer.

Instruction definitions

The instructions emitted by the instruction analyzer are used to generate TableGen instruction definitions in *TargetInstrInfo.td*. This file will be used to generate the *TargetGenInstrInfo.inc* include file.

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

The following instruction (**ADD**) from the **RV32IM_ILP32** VADL specification will be used to illustrate such an instruction definition:

Listing 4.16: TableGen definition of an instruction format.

```
def ADD : Instruction {
   let Namespace = "RV32IM_ILP32";

   bits<5> rs2;
   bits<5> rs1;
   bits<5> rd;
   bits<3> funct3 = 0b000;
   bits<7> funct7 = 0b0000000;
   bits<7> opcode = 0b0110011;

   let AsmString = "ADD $rd,$rs1,$rs2";
   let OutOperandList = (outs X:$rd);
   let InOperandList = (ins X:$rs1, X:$rs2);

   let Size = 4;
   field bits<32> Inst;

   let mayLoad = 0;
   let mayStore = 0;

   let Inst{ 31 - 25 } = funct7;
   let Inst{ 24 - 20 } = rs2;
   let Inst{ 19 - 15 } = rs1;
   let Inst{ 14 - 12 } = funct3;
   let Inst{ 11 - 7 } = rd;
   let Inst{ 6 - 0 } = opcode;
}
```

All instructions must inherit the **Instruction** type, which defines several fields used by the tool-chain. For more information please refer to */llvm/include/llvm/Target/Target.td*.

The first field of the definition defines the namespace of the instruction, which will be used in the generated C++ file.

The following definitions describe the fields of an instruction:

```
bits<5> rs2;
bits<5> rs1;
bits<5> rd;
bits<3> funct3 = 0b000;
bits<7> funct7 = 0b0000000;
bits<7> opcode = 0b0110011;
```


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

Fields **rs2**, **rs1** and **rd** define the parts of the format that are not known until compile time, i.e. the operands of the instruction, whereas fields **funct3**, **funct7** and **opcode** define the constant parts of the instruction and usually correspond to opcodes.

The next section defines how the operands are used:

```
1 let AsmString = "ADD $rd,$rs1,$rs2";
2 let OutOperandList = (outs X:$rd);
3 let InOperandList = (ins X:$rs1, X:$rs2);
```

The *AsmString* field provides the assembly syntax of the instruction used for assembly code emission, referring to the operands using a "$". The *OutOperandList* and *InOperandList* categorize the operands into outputs and inputs, using the value preceding the ":" as type and the "$" to refer to the operand.

```
1 let Size = 4;
2 field bits<32> Inst;
```

The *Size* field defines the size of the instruction in bytes, i.e. 4 byte. Based on the size, field *Inst* instantiates the bits, i.e. $8 * \ size = 32$, used to assign the corresponding fields to it.

Next certain properties that can be used for optimizations by LLVM are defined:

```
1 let mayLoad = 0;
2 let mayStore = 0;
```

The current implementation only supports a handful of properties, namely *mayLoad*, *mayStore*, *isBranch* and *isTerminator*. For a complete list of possible values please also refer to */llvm/include/llvm/Target/Target*.

mayLoad This flag is set for load instructions.

mayStore This flag is set for store instructions.

isBranch This flag is set for branch instructions.

isTerminator This flag is currently only set for branch instructions, since every branch is also a terminator of a basic block.

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

The following fields are used for binary code emission:

```
let Inst{ 31 - 25 } = funct7;
let Inst{ 24 - 20 } = rs2;
let Inst{ 19 - 15 } = rs1;
let Inst{ 14 - 12 } = funct3;
let Inst{ 11 - 7 } = rd;
let Inst{ 6 - 0 } = opcode;
```

Note that these assignments correspond to the instruction format defined by VADL.

Custom operands Consider the *ADDI* instruction already shown in the previous section:

```
instruction ADDI : I_TYPE = {
   X(rd) := X(rs1) + ImmediateI
}
```

The immediate *ImmediateI* defines a bit sequence using the *imm* operand of *ADDI*:

```
immediate ImmediateI : I_TYPE -> Word = sext(imm, 32)
```

In the case of *ImmediateI*, the 32 bit sign extension of *imm* is constructed. Therefore, the operation described by *ADDI* operates on 32 bit. An instruction selection pattern for *ADDI* would thus map a 32 bit integer addition (+) to an instruction only supporting a 12 bit operand. Since this is not possible, a custom immediate operand must be generated.

For each instruction operand type provided by the instruction analyzer a corresponding immediate operand is generated.

These operands are all similar, therefore the operand corresponding to *ImmediateI* is shown as reference:

```
def ImmediateI : Operand<i32>, ImmLeaf<i32, [{
   if ( !(-2048 <= (int32_t) Imm && (int32_t) Imm <= 2047) ) {
      return false;
   }
   return true;
   }]> {
   let EncoderMethod = "getImmFromImmediateI";
}
```

This operand defines a predicate, which checks whether the 32 immediate value fits in a signed 12 bit integer, using its upper and lower bound. Note that the above snippet

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

simplifies the range check, since the compiler generator currently checks whether the 32 bit integer corresponds to the binary structure depicted by the immediate definition. The *EncoderMethod* field refers to a generated C++ method in *TargetMCCodeEmitter.cpp* used for binary code emission and extracts the 12 bit corresponding to the instruction operand *imm*.

4.5.3 Instruction selection patterns

Each instruction selection pattern found by the pattern analyzer is used to write an equivalent TableGen pattern to *TargetInstrInfo.td*.

Lets have a look at a simple pattern definition used for the **ADD** instruction.

Listing 4.17: TableGen definition of a simple instruction selection pattern.

```
1 def : Pat<(add X:$rs1, X:$rs2), (ADD X:$rs1, X:$rs2)>;
```

This pattern is split into a LHS and a RHS. The LHS describes a part of the input target-independent SelectionDAG, i.e. an *add* operation. If this part is matched, it is replaced with the RHS, i.e. the logical equivalent target-dependent SelectionDAG, using the target-specific instruction *ADD*. In this example the target-independent *add* using two integer registers is mapped to the integer addition (+) of the target architecture. Note that target-independent operations like *add* and *sub* infer their type using their operands, i.e. the same *add* is used for register and immediate values. Another important aspect of the target-independent operations is that most of them obey the associative, commutative and distributive laws, reducing the need of repetitive patterns.

It is also possible to use concrete values instead of parameters like this:

Listing 4.18: TableGen definition of a simple instruction selection pattern.

```
1 def : Pat<(ImmediateI:$imm, (ADDI X0, ImmediateI:$imm)>;
```

This pattern is used to load a 12 immediate value into a register, by utilizing the *ADDI* instruction and the register **X0**.

TableGen instruction selection patterns are generated by traversing the micro-operation DAG of both LHS and RHS and printing them.

4.6 SelectionDAG legalizing

The form of the SelectionDAG is affected by the input program, mapping rules for certain languages (e.g. C to IR) and the optimizations activated. This can lead to the use of unsupported operations and therefore will lead to errors during instruction selection.

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

A SelectionDAG leading to such errors is called illegal. Legalizing is the process of transforming an illegal SelectionDAG into a legal one, i.e. only containing operations and operands that are supported by a particular target. This section will discuss how the compiler generator can automatically emit code that legalizes an illegal SelectionDAG. However, only illegal instructions from mapping rules and optimizations are considered, since illegal instructions introduced by a specific program are much harder to legalize. Just imagine a program using floating point operations and compiling it with a target that only supports integer operations. Legalizing such a program is not possible by using target specific instructions. Note that LLVM automatically replaces unsupported floating operations with calls to a soft float library, if not told otherwise.

The compiler generator maintains a map, of every supported operation and instructions that were associated with it, allowing it to take legalizing actions according to this information.

Legalizing can either be performed by instruction selection patterns in *TargetInstrInfo.td* or as custom C++ code in *TargetISelLowering.cpp*.

LLVM supports three types of actions for legalizing operations without defining extra patterns, e.g. promoting, expanding and custom [selb].

4.6.1 Promoting

Promoting allows unmatched operations of a smaller type to be lifted to an instruction using a bigger type. This can be accomplished by looking at semantically equivalent operations that were matched, for example all load operations for integer registers, and check which integer registers are unsupported. So if for example only integer registers for 8, 16 and 32 bit values are supported, 1 bit values can be promoted to use the 8 bit instruction.

4.6.2 Expanding

Expanding is similar to the transformation mechanism described in the last section. LLVM defines rules on how operations can be expanded into a set of other instructions, applying them if an expanding rule was specified for a particular operation. The compiler generator must therefore decide whether to use an additional instruction selection pattern emitted by its own transformation mechanism or by using an expansion rule.

4.6.3 Custom

In cases where neither promoting nor expanding is sufficient, a custom rule can be created, using C++ methods. These methods can usually access all the information provided by the node and the SelectionDAG. This form of legalization is quite hard for the compiler generator, since the information on how to generate these methods can not always be found by static analysis and must be given explicitly by the programmer.

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

In addition, some custom rules need extra code in other LLVM phases to work properly and are only used as a “hack” to work around a missing feature.

The following questions arise for supporting custom rules:

- Should the compiler generator generate these rules automatically using static analysis or ask the user to provide information for this?
- What rules are good enough if generated automatically?
- Does VADL need a generic language construct to describe such rules or define special constructs for most common problems?

Currently only a handful of simple custom rules, related to symbols are supported. To support more complex rules an adequate solution must be found in the future.

4.6.4 Function calls

A function call is the only construct that always needs special lowering. The lowering can be split into the following subtasks:

- Lower call sequence
- Lower call frame
- Emit code

Lower call sequence

This lowering is divided into three smaller parts, which correspond to the calling conventions of a target, while entering and leaving a function, as well as calling a function [ES10].

Entering a function When entering a function, its formal arguments, i.e. parameters must be loaded into virtual registers or into the stack, which is implemented by the **LowerFormalArguments** method of *TargetISelLowering.cpp*. The method uses the calling conventions defined in *TargetCallingConv.td* and inserts additional copy and load nodes into the SelectionDAG.

Leaving a function When leaving a function the return value must be moved into a physical register or the stack, which is implemented in the **LowerReturn** and **CanLowerReturn** methods of *TargetISelLowering.cpp*. Right now, only returning the value in a single register is supported. In addition a return flag is emitted, which will be matched against a target-dependent return instruction during instruction selection.

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

This return flag is defined in *TargetInstrInfo.td* as following:

Listing 4.19: SelectionDAG node definition of a return flag.

```
1 def TargetName_ret_flag : SDNode<"TargetNameISD::RET_FLAG", SDTNone,
    [SDNPHasChain, SDNPOptInGlue, SDNPVariadic]>;
```

Calling a function When calling a function the arguments must be loaded to the argument registers or into the call stack. Additionally start and end nodes of the sequence are lowered to a target-dependent type and a call flag is emitted. The lowering is implemented in the **LowerCall** method of *TargetISelLowering.cpp*.

The start and end nodes for the sequence must be defined in *TargetInstrInfo.td* before they can be used:

Listing 4.20: SelectionDAG node definitions for the call sequence.

```
1 def SDT_CallSeqStart : SDCallSeqStart<[SDTCisVT<0, i32>, SDTCisVT<1,
    i32>]>;
2 def SDT_CallSeqEnd : SDCallSeqEnd<[SDTCisVT<0, i32>, SDTCisVT<1, i32
    >]>;
3
4 def callseq_start : SDNode<"ISD::CALLSEQ_START", SDT_CallSeqStart, [
    SDNPHasChain, SDNPOutGlue]>;
5 def callseq_end : SDNode<"ISD::CALLSEQ_END", SDT_CallSeqEnd, [
    SDNPHasChain, SDNPOptInGlue, SDNPOutGlue]>;
```

The first two lines define type profiles for the call sequence start and end, using predefined type profiles, defining that 0 return operands and 2 input operands are required for nodes using this profile. The two *SDTCisVT* argument definitions, define that the two input operands must be of type i32. According to the definition found in *ISDOpcodes.h* these operands correspond to the size of the call frame part, which must be set up within the sequence pair (start and end) and the part of the call frame prepared prior to the start of the sequence. Furthermore, these operands must be constants and their sum must correspond to the total frame size. The values of these operands are set in the previously mentioned **LowerCall** method. These two type profiles are used to define the custom start and end nodes. *SDNPHasChain*, *SDNPOutGlue* and *SDNPInGlue* are node properties. A list of avaiable properties can be found in *SDNodeProperties.td*.

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

Additionally the following pseudo nodes must be defined:

Listing 4.21: SelectionDAG definitions of call stack pseudo instructions.

```
def ADJCALLSTACKDOWN : Instruction {
   let Defs = [X2]; // X2 = stack pointer
   let OutOperandList = (outs);
   let InOperandList = (ins i32imm:$amt1, i32imm:$amt2);
   let Pattern = [(callseq_start timm:$amt1, timm:$amt2)];
}

def ADJCALLSTACKUP : Instruction {
   let Defs = [X2]; // X2 = stack pointer
   let OutOperandList = (outs);
   let InputOperandList = (ins i32imm:$amt1, i32imm:$amt2);
   let Pattern = [(callseq_end timm:$amt1, timm:$amt2)];
}
```

They will be matched with **callseq_start** and **callseq_end** nodes and will be used during frame lowering 4.6.4. Note that the operands must define the same size as the operands of **callseq_start** and **callseq_end**.

The call flag is defined as follows:

Listing 4.22: SelectionDAG node definitions of the call flag.

```
def SDT_TargetNameCall : SDTypeProfile<0, -1, [SDTCisVT<0, i32>]>;

def TargetName_call : SDNode<"TargetNameISD::CALL",
   SDT_TargetNameCall, [SDNPHasChain, SDNPOptInGlue, SDNPOutGlue,
   SDNPVariadic]>;
```

This node will be used to emit the target-dependent call instruction during code emission.

Note that the current implementation does only support passing arguments in registers or in the stack. Variable length arguments, struct passing, etc. is currently not explicitly handled. In addition, function calls currently only work with a frame pointer, even if it is not needed. Chapter 6 will discuss these parts in more detail.

For more details of the implementation of the C++ methods please have a look at the backends provided by LLVM.

Lower call frame

The call frame contains information about each function during a call and is handled in four different phases [ES10] during frame lowering in *TargetFrameLowering.cpp*.

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

Function prologue Before each function call, the stack pointer must be adjusted to make room for the new function call frame, i.e. allocate enough space on the stack for callee-saved registers, old stack pointer, etc. In addition, the new frame pointer must be set. The prologue is performed by the **emitPrologue** method of *TargetFrameLowering.cpp*. Note that, saving of callee-saved registers is done in the **spillCalleeSavedRegister** method, which delegates the actual store action to the **storeRegToStackSlot** method of *TargetInstrInfo.cpp*. As indicated earlier, special caller-saved registers such as the return address are handled as unconditionally spilled callee-saved registers. Therefore, they are added in *TargetCallingConv.td* as callee-saved and marked for spilling in the **determineCalleeSavedRegister** method of *TargetFrameLowering.cpp*. It should be noted, that a custom approach would be beneficial in the future to give the generator more freedom to adapt to new features.

Function epilogue The epilogue can be seen as reverse prologue, therefore performing the inverse steps of the prologue in reverse order. The analog methods are **emitEpilogue**, **restoreCalleeSavedRegisters** and **loadRegFromStackSlot**.

Eliminate frame pseudo instructions The previously discussed pseudo instructions **ADJCALLSTACKDOWN** and **ADJCALLSTACKUP**, are used to adjust the frame size, if it wasn't known during compile-time, which usually happens if variable sized objects are contained in the frame [ES10]. The compiler generator as of now does not handle variable sized objects and thus only erases these pseudo instructions. This lowering is implemented by the **eliminateCallFramePseudoInstr** method of *TargetFrameLowering.cpp*.

Eliminate frame index Until now all operations accessing sections of a frame, refer to the index of a frame slot and an offset [ES10]. The purpose of eliminating the frame index is to calculate the actual address of the word the operation is referring to and replace the frame index with a register holding this address. This is done in the **eliminateFrameIndex** method in *TargetRegisterInfo.cpp*.

The next subsection deals with the code emission of the lowered call and return instructions.

Emit Code for call and return instructions

To actually emit code for call and return instructions, special pseudo instructions are used. This is done to handle possible relocations of the call symbol and custom return sequences that cannot be directly matched by an instruction selection pattern.

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

The following pseudo instructions where generated for the **RV32IM** VADL specification and are located in *TargetInstrInfo.td*:

Listing 4.23: SelectionDAG definitions for return and call pseudo instructions.

```
def PseudoRET : Instruction, PseudoInstExpansion<(JALR X1, 0, X1)> {
   let isBarrier = 1;
   let isReturn = 1;
   let isTerminator = 1;
   let InOperandList = (ins);
   let OutOperandList = (outs);
   let Pattern = [(RV32IM_ILP32_ret_flag)];
}

def PseudoCall : Instruction {
   let isCall = 1;
   let Defs = [X1];
   let AsmString = "call $func";
   let InOperandList = (ins call_symbol:$func);
   let OutOperandList = (outs);
}
```

The *PseduoRET* instruction is used to match the previously mentioned return flag and therefore does not define any operands. An additional *PseduoInstExpansion* is used to generate arbitrary instruction sequences, that otherwise wouldn't be possible to be represented as an instruction selection pattern, i.e. "JALR X1, 0, X1" used to represent a return instruction does contain operands as opposed to the return flag and therefore does not allow to be directly matched by the flag. Although this mechanism is rather straight forward, it introduces yet another special case for the compiler generator to take care of. It is therefore suggested to treat returns similar to call instructions in the future, leading to more unified code and removing special cases that need to be maintained. Note that this instruction sequence corresponds to the return sequence of the ABI specification.

The *PseudoCall* instruction is used to match the previously mentioned call flag and therefore contains the call symbol as only operand. An additional assembly definition is given with *AsmString* to support assembly code emission. Note that the compiler generator currently always writes "call $func" and should be changed by either concatenating the instructions of the ABI call sequence or by using a dedicated pseudo instruction (in VADL).

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

The patterns used for selecting the pseudo instruction look like this:

Listing 4.24: Instruction selection patterns for selecting various function calls.

```
1 def : Pat<(RV32IM_ILP32_call tglobaladdr:$func), (PseudoCall
     tglobaladdr:$func)>;
2 def : Pat<(RV32IM_ILP32_call texternalsym:$func), (PseudoCall
     texternalsym:$func)>;
```

Note that only global addresses and external symbols are currently supported by the generator.

Definition 4.6.1 (Global address)**.** An address to a global value.

Definition 4.6.2 (External symbol)**.** A global symbol defined in a different linking unit, e.g. a library call.

The pseudo call node will be lowered during object code emission to concrete target-specific instructions.

4.7 Code emission

Code emission is responsible for lowering LLVM's code generation abstraction down to machine layer abstractions, used to either emit assembly or object code [llva]. This layer operates on assembly and object files using concepts like labels, directives and instructions.

4.7.1 Lowering

The *TargetAsmPrinter.cpp* and *TargetMCInstLower.cpp* files, lower the more general abstraction of instructions (MachineInstr) to MCInst instructions, which are used as input for assembly printing and binary encoding.

4.7.2 Assembly code emission

Most of the assembly code emission is automatically generated by TableGen, using the assembly strings provided by the instruction definitions. The generated code located in *TargetGenAsmWriter.inc* is included in *TargetInstPrinter.cpp* and used for printing the instructions into the assembly file. Register operands are printed according to the **getRegisterName** method, generated from the register definitions in *TargetRegisterInfo.td*, whereas immediate values are printed without additional processing. Meta-information like assembly directives and comment syntax is described in *TargetMCAsmInfo.cpp*. Special symbols and expressions, for example the function call symbol and assembly modifier are represented as **MCTargetExpr**, which are defined in *TargetMCExpr.cpp*, and implement their own print method.

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

4.7.3 Object code emission

A C Compiler traditionally emits assembly files and relies on a dedicated assembler to produce object code. For this the binary encoding of instructions and relocations are emitted into object code format, such as ELF or Mach-O. LLVM supports the use of an external assembler and direct object code emission [llve]. For the direct object code emission, a so-called integrated assembler is used, to omit the extra overhead of parsing assembly files. Cook provides a detailed introduction on how to implement the components for the integrated assembler [llvd]. Basically, the functionality of the assembler is located in *TargetAsmBackend.cpp* and can be used to either generate a native or integrated assembler. In addition, most of the work for writing an object file is already implemented by object streamers.

Machine binary code

TableGen uses the instruction definitions defined in *TargetInstrInfo.td* to generate the *TargetMCCodeEmitter.inc* file used by *TargetMCCodeEmitter.cpp* to encode incoming MCInst instructions into binary code and write them to the object file. The only exceptions are function calls, which need special treatment, as previously mentioned. Pseudo call instructions are lowered into concrete target instructions and encoded directly before writing them to the object file.

Relocations

Relocations are used to patch values that are currently unknown to the system, like the location of an external symbol, and represented using so-called **Fixups**. For each relocation a corresponding fixup in *TargetFixupKinds.h* is defined. The assembler tries to patch the values that are annotated with fixups, if the information is already available or treat the values as 0 and emit a corresponding relocation in the object file [llvd]. Currently the only fixup that is supported by the generator is a relocation, which uses the symbol offset of the call sequence. This fixup is applied during the lowering of the function call and propagated until the **TargetMCExpr** for the symbol is annotated accordingly.

4.8 Linking

Linking is the final step to produce an executable file and responsible to patch relocations contained in the object file. A linker can therefore be considered an integral part of a compiler and must be generated for a new processor model, since no currently available linker will support it.

LLVM supports a linker for all major object formats. To support a new target, LLVM's linker lld[5] must be extended. The changes are rather simple and only a handful of source

[5] https://lld.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 Bibliothek Your knowledge hub WIEN

files in the **lld** project must be adapted for basic functionality:

- Command-line argument parsing (*lld/ELF/InputFiles.cpp*)
- Target initialization (*Driver.cpp*)
- Relocation patching (*lld/ELF/Target.cpp*, *lld/ELF/Target.h* and *lld/ELF/Arch/ActualTargetName.cpp*)
- Adding the target to the build (*lld/ELF/CMakeLists.txt*)

The generation and patching of these files is not implemented right now, and must be taken care of in addition to better relocation support.

4.9 Direct user definitions

Sometimes, especially in this early stage of the compiler generator, it is necessary to manually add additional code for testing new features or supporting some missing features to get a working compiler. Therefore, a simple mechanism, build upon textual replacement, was implemented to allow hand-written code to be appended to the automatically generated LLVM backend. For this textual replacement an input folder must be created in the VADL source directory, reassembling the structure of the generated backend. Each file, matching a file of the generated backend, will be appended to the end of the generated file. Although this is rather restrictive, it has shown to be efficient enough for now. Since this is a work in progress it is likely that this mechanism will be replaced by VADL builtin functionality in the future. However, such a mechanism is undoubtedly needed and should also be provided.

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

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 5

Evaluation

The compiler generator was evaluated by describing the **RISCV** subset **RV32IM** in VADL version *2021–01–15*. This processor architecture was chosen as RISC representative, since it is simpler than **ARM** and more sophisticated than **MIPS**, giving a good insight on the capabilities of the compiler generator.

5.1 Benchmarks

The benchmarks were consulted to test the performance of a generated compiler compared to the reference implementation of **RISCV** of LLVM. Both backends were compiled as part of LLVM using a release build.

The following performance measurements were used for evaluation:

- Assembly instruction count
- Number of executed instructions
- Number of executed branches
- Simulation time

Furthermore, the expressiveness of VADL and the compiler generator was evaluated based on the additional manual work necessary for the generated compiler to be able of compiling the benchmarks. The following measurements were used:

- Number of hand-written instruction selection patterns
- Number of hand-written C++ lines

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

The following table lists all software and hardware components that were used to perform the benchmarks:

Component	Name	Version
OS	Arch Linux x86_64	5.4.69–1–lts
CPU	AMD Ryzen 5 2600X (12) @ 3.600GHz	–
Compiler	LLVM	10.0.0
Compiler	GCC	10.2.0
C Stdlib (ELF, RISCV)	riscv-gnu-toolchain	e3e8e28
Assembler (ELF, RISCV)	riscv-gnu-toolchain	e3e8e28
Linker (ELF, RISCV)	riscv-gnu-toolchain	e3e8e28
Benchmark Suite	Dhrystone (OVPSIM)	v20200708.0
Benchmark Suite	MiBench	1.0
Instruction set simulator	VADL generated	V0.3

Table 5.1: Benchmark system information

5.1.1 General setup

Clang was only used to generate assembly code, since neither assembler nor linker for the generated backend are currently available. This code was assembled and linked into an ELF executable using the *riscv-gnu-toolchain*. C libraries and startup code were pre-compiled and linked manually. Since the generated backend currently doesn't fully support variable length arguments, minor adoptions of the source code where performed. Some library calls, such as *printf* where replaced by wrapper functions using the exact number of parameters for each occurrence and moved into an external file, which was then compiled by the reference implementation and linked against the object code produced by the generated backend. This is only done to verify that the compiled code is correct and all *printf* wrappers are removed from the final benchmark, to avoid false results. All programs were compiled with the highest optimization level "O3".

5.1.2 Test programs

Overall, six test programs from the **Dhrystone**[1] and **MiBench**[2] benchmark suites were used, ranging from solving simple arithmetic and graph problems to sort and encryption algorithms. Note that because of time constraints and missing compiler features, only a handful of programs from **MiBench** were used, e.g. *basicmath*, *dijkstra*, *qsort*, *sha* and *stringsearch*.

[1] https://github.com/riscv/riscv-ovpsim
[2] http://vhosts.eecs.umich.edu/mibench/

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

Results

All programs where executed by the ISS 10 times in a row, taking the average of all performance measures to ensure that changes in the computer's workload did not effect the results. Table 5.2 shows all runtime performance measurements with each row corresponding to a test program possibly having an additional *small* or *large* suffix, indicating its input size. The first part of a row presents the results of the LLVM upstream target, followed by the results of the generated target. The values of the generated target are annotated by the relative increase (in red) or decrease (in blue) to the reference implementation.

Program	Target	Instructions	Branches	Simulation time (in seconds)
dhrystone	riscv32	1115004398	150000727	22.178
	RV32IM_ILP32	1320004420 +18.39%	255000725 +70%	25.032 +12.87%
basicmath small	riscv32	1305669912	151012027	28.944
	RV32IM_ILP32	1125653310 -13.79%	140905199 -6.69%	24.783 -14.38%
basicmath large	riscv32	55395278481	7889427355	1180.874
	RV32IM_ILP32	54846546369 -0.99%	7859501423 -0.38%	1168.726 -1.03%
dijkstra small	riscv32	36771777	4148885	0.778
	RV32IM_ILP32	43345422 +17.88%	8667863 +108.92%	0.984 +26.41%
dijkstra large	riscv32	167860473	19301848	3.453
	RV32IM_ILP32	200987322 +19.73%	42151931 +118.38%	4.498 +30.26%
qsort small	riscv32	17609489	2486613	0.348
	RV32IM_ILP32	17988656 +2.15%	2553825 +2.70%	0.373 +7.40
qsort large	riscv32	214340901	22874674	5.013
	RV32IM_ILP32	218409146 +1.90%	23428534 +2.42%	5.077 +1.27%
sha small	riscv32	14444323	814602	0.287
	RV32IM_ILP32	16827393 +16.50%	839044 +3.00%	0.348 +21.25%
sha large	riscv32	150362170	8474694	3.040
	RV32IM_ILP32	175177591 +16.50%	8729207 +3.00%	3.670 +20.71%
stringsearch small	riscv32	64709	15447	0.002
	RV32IM_ILP32	80098 +23.78%	15730 +1.83%	0.003 +20.79%
stringsearch large	riscv32	1460277	352168	0.038
	RV32IM_ILP32	1815969 +24.36%	358826 +1.89%	0.041 +7.72%

Table 5.2: Runtime performance results.

The programs compiled by the generated compiler usually perform between 1.90% and 24.36% more instructions and up to 118.38% more branches, but do not perform significantly worse than up to 30.26% in total simulation time. Only the **basicmath** program performs better for both small and large input. The gains seem however to decline with increasing input. The explanation for branches is simple, the generated compiler does not consider any branch optimizations on BBs. In addition, **RISCV** does not contain any conditional move instructions, which is directly opposing LLVM's efforts of optimizing small basic blocks. Both the upstream and generated target replace these moves with custom basic blocks, but only the upstream target adds additional optimizations on top of them.

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

The next metrics about the produced assembly files should give more insights regarding the results.

Program	Target	Assembly instructions	Basic block (BB) labels	Branch instructions to BBs
dhrystone	riscv32	285	20	24
	RV32IM_ILP32	474 +66.32%	32 +60.00%	39 +62.50%
basicmatch small	riscv32	1101	10	11
	RV32IM_ILP32	718 -34.79%	15 +50.00%	15 +36.36%
basicmath large	riscv32	1213	11	12
	RV32IM_ILP32	810 -33.22%	17 +54.55%	17 +41. 67%
dijkstra small	riscv32	406	25	30
	RV32IM_ILP32	499 +22.91%	44 +76.00%	50 + 66.67%
dijkstra large	riscv32	363	26	31
	RV32IM_ILP32	458 +26.17%	46 +76.92%	52 +67.74%
qsort small	riscv32	68	6	6
	RV32IM_ILP32	75 +10.29%	7 +16.67%	8 +33.33%
qsort large	riscv32	128	6	9
	RV32IM_ILP32	156 +21.88%	10 +66.67%	14 + 55.56%
sha	riscv32	579	21	25
	RV32IM_ILP32	754 +30.22%	36 +71.43%	41 +64.00%
stringsearch small	riscv32	149	15	19
	RV32IM_ILP32	187 +25.50%	22 +46.67%	33 +73.68%
stringsearch large	riscv32	149	15	19
	RV32IM_ILP32	189 +26.85%	22 +46.67%	33 +73.68%

Table 5.3: Assembly statistics of the compiled programs.

Table 5.3 shows the total number of instructions, number of BBs and number of jump instructions to BBs. These statistics were obtained by parsing the output assembly files and associating every line according to following rules:

- Lines not starting with "." (directive), "#" (comment) or end with ":" (label) are considered instructions
- Lines ending with a ":" and containing "LBB" (LLVM's convention for a BB) are considered a label corresponding to a basic block
- Instructions that contain a label to a basic block are considered jumps to a BB

All programs, which perform more instructions do have more assembly instructions and branches to basic blocks. It is not surprising to see the high increase of BB labels and jumps directing to them, considering the general increase of executed branches. Investigating the assembly files more closely has lead to the conclusion that most of the additional instructions correspond to the function prologue and epilogue, i.e. the spilling and restoring of registers. Often both spilling and restoring is performed, although not necessary. **Basicmath** is the only program producing less assembly instructions, which seems to directly correlate to the better runtime performance, despite still having more branches than the reference implementation. Looking closely at the assembly file, the answer is simple: the function called most often *SolveCubic* has only 349 instead 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 Bibliothek Your knowledge hub WIEN

816 instructions. The reason for the bigger amount of instructions seems to lie in the additional load instructions performed by the upstream target at the end of each library function call (soft float). Since the generated compiler does not explicitly handle this case, it seems to have a little advantage here, but in return might be more unreliable in some cases. Future development therefore should further investigate this behavior and take according steps to generate proper handling of function calls.

For the generated compiler to actually work for the provided benchmark programs, additional manual work was required, including 48 instruction selection patterns and approximately 300 lines of custom C++ code. Most of this code was re-used from the upstream target, if the code was target-independent, i.e. can be used for every target without specific dependencies to **RISCV**. Almost all patterns are only derivations of already generated patterns and it shouldn't be too hard generating them, after proper support for them is added. The custom C++ code corresponds to lowering of the SelectionDAG and is rather simple. Once VADL and the compiler generator support better lowering mechanisms, it should be no problem to generate this code. Further information about this topic is presented in the next chapter.

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

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.

CHAPTER 6

Future work

As for now only basic functionality is provided by the implementation of both VADL and the compiler generator. This section will list some of the features that couldn't be implemented as part of this thesis. It is currently possible to generate a LLVM backend for basic integer RISC architectures, however several things must still be done manually. The following sections concentrate on the features necessary to automate these hand-written parts. Support for VLIW architectures or floating-point operations is part of additional works.

6.1 Instruction selection

It is currently necessary to write some additional instruction selection patterns. This section will list some examples and discuss the problem of generating them.

Consider the pattern definition of **RV32IM** (adopted from the RISCV backend) for loading a 32 bit immediate value into a register:

Listing 6.1: Loading 32-bit integers in RISCV.

```
1 def : Pat<(simm32:$imm), (ADDI (LUI (HI20 imm:$imm)), (LO12Sext imm:
    $imm))>;
```

This pattern uses the instruction pair *ADDI* and *LUI*, to first load the upper 20 bit of the 32 bit immediate value, after which the lower 12 bit are loaded. Therefore, the node transformations *HI20* and *LO12Sext* are used.

Listing 6.2: Node transformations HI20 and LO12Sext as defined by the RISCV LLVM backend.

```
1 def HI20 : SDNodeXForm<imm, [{
2    return CurDAG->getTargetConstant(((N->getZExtValue() + 0x800) >>
        12) & 0xfffff, SDLoc(N), N->getValueType(0));
3 }]>;
4
5 def LO12Sext : SDNodeXForm<imm, [{
6    return CurDAG->getTargetConstant(SignExtend64<12>(N->getZExtValue
        ()), SDLoc(N), N->getValueType(0));
7 }]>;
```

Both the pattern and node transformations can be generated in a rather straight forward manner if somehow defined, i.e. there is an equivalent VADL definition. Generating this kind of instruction sequence solely from existing instruction definitions is rather hard, since this would need a sophisticated algorithm to combine instructions (e.g. *ADDI* and *LUI*) based on an input semantics, in addition to find the correct transformations (e.g. *HI20* and *LO12Sext*) to make them fit together. Therefore, it seems to be of practical use to let the designer of an ISA define such instruction sequences in a generic way.

A similar problem is introduced by patterns for unconditional jumps. Some ISAs do not define dedicated instructions for unconditional jumps, but parameterize other instructions to have the same outcome. Consider the currently hand-written pattern for unconditional jumps for the **RV32IM** target:

Listing 6.3: Instruction selection pattern for an unconditional branch in RISCV.

```
1 def PseudoBR : PseudoInstr<(ins ImmediateJ:$imm20), (outs), [(br bb:
    $imm20)]>, PseudoInstExpansion<(JAL X0, ImmediateJ:$imm20)>;
```

The instruction sequence “JAL X0, ImmediateJ:$imm20” in combination with the *PseudoInstExpansion* mechanism is used to match unconditional branches to a basic block within a 20 bit offset. To actually support the automatic generation of this pattern, the previously mentioned approach of defining this sequence explicitly could be used. However, this simple example could also be generated using the transformation mechanism. Therefore, it shouldn’t be a problem to support these kinds of patterns in the future.

The next category of hand-written patterns concerns the selection of a frame index. The SelectionDAG contains a variety of loads/stores and moves from the stack and must be matched by target-specific instructions. These patterns are not different to general load/store/move patterns, expect that they need an additional complex pattern (can be considered boilerplate) for selecting a frame index as 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 Bibliothek Your knowledge hub WIEN

Consider the following pattern for storing a value to a specific frame index into memory:

Listing 6.4: Instruction selection pattern to store from a frame index.

```
1 def : Pat<(store X:$rs2, SelectFrameIndexAddress:$rs1), (SW X:$rs2, 0,
      SelectFrameIndexAddress:$rs1)>;
```

This pattern is completely similar to the general store pattern:

Listing 6.5: Instruction selection pattern to store a value from an integer register.

```
1 def : Pat<(store X:$rs2, X:$rs1, (SW X:$rs2, 0, X:$rs1)>;
```

The only difference is the type of *rs1*, which is changed from *SelectFrameIndexAddress* to *X*. The complex pattern *SelectFrameIndexAddress* is only needed to select a frame index, which will be destroyed during frame lowering. Thus, supporting these kinds of patterns is only a matter of incorporating the frame pointer to micro-operation DAGs and automatically generating additional patterns for each load/store/move pattern using a frame index instead of a register or register file.

Other patterns must be written by hand, because they are currently unsupported, but could easily be supported by an additional transformation or native support in micro-operations.

These two patterns must be written by hand, since they are currently not directly supported by a micro-operation:

Listing 6.6: Instruction selection patterns for multiplication.

```
1 def : Pat<(mulhs X:$rs1, X:$rs2), (MULH X:$rs1, X:$rs2)>;
2 def : Pat<(mulhu X:$rs1, X:$rs2), (MULHSU X:$rs1, X:$rs2)>;
```

mulhs and *mulhu* are simple multiplications, only returning the upper half of the result, either signed or unsigned. Supporting this kind of patterns is only a matter of adding additional micro-operations for common scenarios.

The rest of the hand-written patterns are variants of existing patterns and could be automatically generated by implementing additional transformations for the transformation mechanism mentioned in Chapter 4.

6.2 Use information provided by the MiA

Currently only the ISA and ABI of a processor are considered, ignoring useful scheduling information, which could be used to generate a schedule model [Est] in LLVM.

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

In addition, information on the RTL level of instructions could be used to create precise cost functions (currently no cost functions are used) for instruction selection patterns and therefore improve the quality of matches.

6.3 Matching support for complex instructions

Instruction selection patterns are primarily generated from the semantics definition of instructions, which means that the statements and expressions depicting the semantics must be matched to generate according micro-operation DAGs. For now only a handful of common scenarios is supported, but as the focus shifts to highly optimized processor architectures, more complex instruction definitions will be defined and should also be supported by the compiler generator.

For example the use of several if/else blocks to encode different addressing modes based on the parameters of an instruction, could be matched to yield different instruction selection patterns for each provided condition.

It will be necessary to gradually add support for these kinds of instructions.

6.4 SelectionDAG Legalizing

Operations needing legalization can differ across the various optimization levels of LLVM. Optimization level O3 for example, will produce **setcc** operations to remove small basic blocks used for conditional assignments, and must be selected with conditional move instructions However, some targets do not support conditional move instructions and must replace these *setcc* operations with basic blocks and supported branch instructions. For this particular problem a simple template for inserting basic blocks can be used for targets not supporting conditional instructions. A future legalization mechanism should therefore also support LLVM's basic block inserter mechanism.

6.5 Function calls

The compiler generator supports only the bare minimum necessary for handling function calls, thereby not supporting features like variable length arguments or how registers should be loaded from or stored to the stack. Since this behavior is defined by the ABI of an architecture, VADL should add dedicated language constructs to model the missing parts to provide better support for the compiler generation.

A future VADL version should encode information on how to:

- Spill/restore callee-saved registers
- Spill/restore caller-saved registers
- Handle variable length arguments

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

- Handle various C data types like structs etc
- Handle arguments exceeding the arguments registers capacity
- Use a call stack frame in general

6.6 Relocations

The current relocation supported by VADL is rather limited. Relocations can only be used in combination with the call sequence construct, leaving out arbitrary sequences like load and stores. They are furthermore restricted to simple expressions and omit the possibility to model conditional relaxations. Therefore, a more generic way of using relocations either directly in instruction definitions or a construct that assigns an arbitrary instruction sequence to relocations, is needed. After implementing such a mechanism it should be not too difficult to adapt the current compiler generator to emit code for these relocations.

6.7 Exception Handling & Interrupt support

LLVM supports the handling of unexpected behavior, such as hardware interrupts and exceptions [llvc]. Supporting these features, would need additional instruction selection patterns for trap instructions, as well as special assembly directives that must be emitted alongside the function call sequence to handle the recovering and rewinding of a stack frame. It will be interesting to support it in the future, when VADL adds appropriate support for exceptions.

6.8 Assembler

To fully support the compilation from C source files down to executables, an assembler is needed. Fortunately, it is possible to generate a standalone assembler from a LLVM target definition that can be used with clang. Some of the work is already done however parsing of assembly files and proper relocation support are still missing.

6.9 Linker

The linker is yet another missing component of the compilation process, but can be generated alongside clang. Therefore, additional code for the definition of relocation types and the patching thereof must be generated for *lld*. The work done towards the linker by the compiler generator is very minimal. It currently generates a git[1] patch file[2] containing source changes for the **RV32IM** target and can be applied to a specific

[1] https://git-scm.com/

[2] The content of the patch file corresponds to the output of the git diff command.

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

commit of the LLVM code base. This was done to evaluate the steps necessary for supporting a linker.

Note the files necessary for both the assembler and linker are documented in the respective sections of the implementation chapter and must be implemented to finally support a complete compiler tool-chain.

6.10 Optimizations

The compiler generator does not implement any form of optimizations. To become competitive it is without question that optimizations must be supported in the future. However, it seems that there are only a few architecture independent optimizations, that could be implemented, i.e. using constant register instead of loads/moves or merging basic blocks, etc. An approach that optimizes backends based on their processor architecture seems currently more promising, so some optimizations would only be applied to the RISC family and others only for VLIW architectures.

6.11 Patch and compile C standard library

During the evaluation of the compiler generator compiling actual C code has shown to be quite cumbersome. One of the headaches that were faced, was the support for a pre-compiled C standard library that conforms to both the ABI and ISA of the generated processor. For the evaluation target it was possible to find such a pre-compiled library, however that is only the case for already known architectures. The purpose of this project is to define new processor architectures with unknown ISAs and therefore there will be no pre-compiled C standard library that can be linked by clang. The source code of the library must also be patched to correspond to the hardware limitations of the new processor architecture. To provide a better user experience it could be beneficial if future developments would also consider the generation of a patched C standard library and cross-compile it with the generated compiler.

6.12 User feedback

While developing a processor description in VADL, it might be the case that some features needed for the compiler generator are missing. Therefore, it would be useful to prompt helpful warning and error messages, indicating features that might not be supported by the current specification, making it easier for designers to adapt the processor specification accordingly. There is already a mechanism in place to support generic user feedback for both the development in the VADL IDE and using the commandline tooling, but isn't currently supported for the various generators of VADL. This mechanism uses a custom logger implementation that can log messages and assign them to the respecting code location in the VADL specification. This logger should be used in future developments.

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

CHAPTER 7

Conclusion

This work introduced VADL, a mixed PDL, capable of automatically generating a C compiler from a processor specification. It aims to separate the integral parts of a processor description, e.g. ISA, MIA and ABI without introducing too much redundancies for the connections thereof. The compiler generator currently focuses on the ISA and ABI to extract the target-specific information required for retargeting the LLVM compiler framework, but is not limited to them. Future versions of VADL might use details of the MIA to generate a sophisticated scheduling model of the processor or to annotate instruction selection patterns.

The methodologies and techniques used throughout this work have shown to be capable of the basic needs for automatic compiler generation, however lead to the conclusion that more sophisticated methods are needed in the future. Chapter 6 presents some of the immediate objectives that should be implemented, but with advancing development of VADL these objectives might shift in the long run. It is undoubtedly necessary to increase the effort on analyzing the presented components of VADL to extract more information required by the compiler generator, in addition to more generic language constructs for adding semantics.

Although results show worse performance of the compiled programs, they promise a lot of possible improvements. VADL is still under heavy development and most of the required adjustments are already in work. After allowing to declare the hand-written parts of the compiler to be explicitly defined or implicitly extracted, the focus will shift towards optimizations. The idea is to extend generic target optimizations and introduce special optimizations for processor families, allowing to get competitive performance for common architectures without extra complexity.

This will make VADL in the context of compiler generation an interesting research topic for years to come.

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

List of Figures

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

List of Tables

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

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

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

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 Algorithms

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.

Acronyms

ABI Application Binary Interface. 15, 22–26, 34, 35, 47, 48, 58, 71, 72, 74, 75, 79

ADL Architecture Description Language. 1, 3

ASIP Application Specific Instruction Set Processor. ix, xi, 1, 10

AST Abstract Syntax Tree. 27

BB Basic Block. 65, 66

CAS Cycle-Accurate Simulator. 5, 8

CDFG Control-Data Flow Graph. 10

CISC Complex Instruction Set Computer. 10

CRM Common Resource Model. 27, 34

DAG Direct Acyclic Graph. 12, 29, 37, 42–45, 52, 71, 72

DSP Digital Signal Processor. 10

ELF Executable and Linkable File. 13, 28, 60, 64

GOT Global Offset Table. 25

GPP General Purpose Processor. 9, 10

HDL Hardware Description Language. 3, 4

IDE Integrated Development Environment. 74

ILP Instruction-Level Parallelism. 5, 10, 11

IR Intermediate Representation. 9, 10, 12, 42, 43, 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 Bibliothek Your knowledge hub WIEN

ISA Instruction Set Architecture. 1, 2, 4–6, 8, 9, 13, 15–19, 22–24, 26, 35, 43, 70, 71, 74, 75, 79

ISDL Instruction Set Description Language. 4–6

ISS Instruction-Set Simulator. 4, 8, 11, 65

LHS Left Hand Side. 18, 35, 42, 45, 52

LISA Language for Instruction Set Architecture. 6–8

LLVM Low Level Virtual Machine. ix, xi, 28, 31, 34, 42, 46, 47, 50, 53, 54, 56, 59–61, 63, 65, 66, 69, 71–75

Mach-O Mach Object. 13, 60

MIA Microarchitecture. 2, 4, 15, 75

MIMOLA Machine Independent Microprogramming Language. 7

NOP No operation. 25

PDL Processor Description Language. ix, xi, 1–10, 12–15, 75

PE Portable Executable. 13

PIC Position Independent Code. 25

PLT Procedure Linkage Table. 25

RADL Retargetable Architecture Description Language. 8

RCP Rich Client Platform. 27

RHS Right Hand Side. 35, 42, 45, 52

RISC Reduced Instruction Set Computer. 2, 10, 16, 63, 69, 74

RTL Register-Transfer Level. 1, 4–7, 71

SOC System-on-chip. 5

VADL Vienna Architecture Description Language. ix, xi, 2, 14–19, 22–28, 34–43, 49, 51, 54, 58, 61, 63, 67, 69, 70, 72–75, 79

VLIW Very Long Instruction Word. 10, 69, 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 Bibliothek Your knowledge hub WIEN

Bibliography

[ABC+19] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[BEK07] Florian Brandner, Dietmar Ebner, and Andreas Krall. Compiler generation from structural architecture descriptions. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 13–22, 2007.

[BHE91] David G. Bradlee, Robert R. Henry, and Susan J. Eggers. The Marion System for Retargetable Instruction Scheduling. *SIGPLAN Not.*, 26(6):229–240, May 1991.

[Bli16] Gabriel Hjort Blindell. *Instruction Selection - Principles, Methods, and Applications*. Springer, 2016.

[Bur19] Michal Bureš. Optimalizace ASM kódu pro DLX procesor pomocí LLVM systému. 2019.

[CGH+04] Lakshmi N. Chakrapani, John Gyllenhaal, Wen-mei W. Hwu, Scott A. Mahlke, Krishna V. Palem, and Rodric M. Rabbah. Trimaran: An Infrastructure for Research in Instruction-Level Parallelism. In *Proceedings of the 17th International Conference on Languages and Compilers for High Performance Computing*, LCPC'04, page 32–41, Berlin, Heidelberg, 2004. Springer-Verlag.

[CHL+05] Jianjiang Ceng, Manuel Hohenauer, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, and Gunnar Braun. C compiler retargeting based on instruction semantics models. In *Design, Automation and Test in Europe*, pages 1150–1155. IEEE, 2005.

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

[cla] Assembling a Complete Toolchain. https://clang.llvm.org/docs/Toolchain.html. [Online; accessed November 2020].

[CS] Chen Chung-Shu. Tutorial: Creating an LLVM Backend for the Cpu0 Architecture. https://jonathan2251.github.io/lbd/index.html. [Online; accessed March 2021].

[EPI00] VLIW EPIC. HPL-PD architecture specification: Version 1.1. 2000.

[Ert99] M Anton Ertl. Optimal code selection in DAGs. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 242–249, 1999.

[ES10] Christoph Erhardt and Dipl-Inf Fabian Scheler. *Design and implementation of a tricore backend for the llvm compiler framework*. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2010.

[Est] Dave Estes. SchedMachineModel: Adding and Optimizing a Subtarget. https://llvm.org/devmtg/2014-10/Slides/Estes-MISchedulerTutorial.pdf. [Online; accessed November 2020].

[FHKM94] Andreas Fauth, Günter Hommel, Alois Knoll, and Carsten Müller. Global code selection for directed acyclic graphs. In Peter A. Fritzson, editor, *Compiler Construction*, pages 128–142, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[Fra91] Christopher W. Fraser. A Retargetable Compiler for ANSI C. *SIGPLAN Not.*, 26(10):29–43, October 1991.

[FVF95] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. In *Proceedings the European Design and Test Conference. ED TC 1995*, pages 503–507, 1995.

[Gol17] Connor Jan Goldberg. The Design of a Custom 32-bit RISC CPU and LLVM Compiler Backend. 2017.

[HGG+08] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Design, Automation, and Test in Europe*, pages 31–45. Springer, 2008.

[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.

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

[HKN+01] Andreas Hoffmann, Tim Kogel, Achim Nohl, Gunnar Braun, Oliver Schliebusch, Oliver Wahlen, Andreas Wieferink, and Heinrich Meyr. A novel methodology for the design of application-specific instruction-set processors (ASIP1s) using a machine description language. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(11):1338–1354, 2001.

[HL10] Manuel Hohenauer and Rainer Leupers. *C Compilers for ASIPs*. Springer, 2010.

[HO82] Christoph M Hoffmann and Michael J O'Donnell. Pattern matching in trees. *Journal of the ACM (JACM)*, 29(1):68–95, 1982.

[KÖ3] Daniel Kästner. TDL: A Hardware Description Language for Retargetable Postpass Optimizations and Analyses. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, GPCE '03, page 18–36, Berlin, Heidelberg, 2003. Springer-Verlag.

[Kä00] Daniel Kästner. PROPAN: A Retargetable System for Postpass Optimisations and Analyses. In *Languages, Compilers, and Tools for Embedded Systems*, volume 1985 of *LNCS*, pages 63–80. Association for Computing Machinery, 01 2000.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[LCGDM94] Dirk Lanneer, Marco Cornero, Gert Goossens, and Hugo De Man. Data routing: A paradigm for efficient data-path synthesis and code generation. In *Proceedings of the 7th International Symposium on High-Level Synthesis*, ISSS '94, page 17–22, Washington, DC, USA, 1994. IEEE Computer Society Press.

[Lev00] John R Levine. *Linkers and Loaders*. Morgan Kaufmann, 2000.

[llva] Code emission. `https://llvm.org/docs/CodeGenerator.html#code-emission`. [Online; accessed November 2020].

[llvb] Data Layout. `https://llvm.org/docs/LangRef.html#data-layout`. [Online; accessed November 2020].

[llvc] Exception Handling in LLVM. `https://www.llvm.org/docs/ExceptionHandling.html`. [Online; accessed November 2020].

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

[llvd] Howto: Implementing LLVM Integrated Assembler. https://www.embecosm.com/appnotes/ean10/html/index.html. [Online; accessed November 2020].

[llve] The LLVM Target-Independent Code Generator. https://llvm.org/docs/CodeGenerator.html. [Online; accessed November 2020].

[llvf] Writing an LLVM Backend. https://llvm.org/docs/WritingAnLLVMBackend.html. [Online; accessed November 2020].

[LM96] Rainer Leupers and Peter Marwedel. Instruction selection for embedded DSPs with complex instructions. In *Proceedings EURO-DAC'96. European Design Automation Conference with EURO-VHDL'96 and Exhibition*, pages 200–205. IEEE, 1996.

[LM01] Rainer Leupers and Peter Marwedel. *Retargetable compiler technology for embedded systems: tools and applications*. Springer Science & Business Media, 2001.

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

[Mar86] P. Marwedel. A New Synthesis Algorithm for the MIMOLA Software System. In *23rd ACM/IEEE Design Automation Conference*, pages 271–277, 1986.

[MD11] Prabhat Mishra and Nikil Dutt. *Processor description languages*. Elsevier, 2011.

[PK13] Saravana Perumal P and Amey Karkare. Retargeting GCC: Do We Reinvent the Wheel Every Time? *CoRR*, abs/1309.7685, 2013.

[Pro92] Todd A Proebsting. Simple and efficient BURS table generation. *ACM SIGPLAN Notices*, 27(7):331–340, 1992.

[ris] RISC-V ELF psABI specification. https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md. [Online; accessed March 2021].

[S+20] Richard Stallman et al. *Using the GNU Compiler Collection*. Gnu Press Boston, 2020.

[sela] Introduction to SelectionDAGs. https://www.llvm.org/docs/CodeGenerator.html#introduction-to-selectiondags. [Online; accessed November 2020].

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

[selb] The SelectionDAG Legalizing Phase. https://llvm.org/docs/WritingAnLLVMBackend.html#the-selectiondag-legalize-phase. [Online; accessed November 2020].

[SHN+02] Oliver Schliebusch, Andreas Hoffmann, Achim Nohl, Gunnar Braun, and Heinrich Meyr. Architecture implementation using the machine description language LISA. In *Proceedings of ASP-DAC/VLSI Design 2002. 7th Asia and South Pacific Design Automation Conference and 15h International Conference on VLSI Design*, pages 239–244. IEEE, 2002.

[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.

[VPGLDM94] Johan Van Praet, Gert Goossens, Dirk Lanneer, and Hugo De Man. Instruction set definition and instruction selection for ASIPs. In *Proceedings of 7th International Symposium on High-Level Synthesis*, pages 11–16. IEEE, 1994.

[WLPA16] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanović. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1. 2016.

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