TU WIEN Informatics

Cycle-Accurate Simulator Generator for the VADL Processor Description Language

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Ing. Hermann Schützenhöfer, BSc.
Matrikelnummer 1226141

an der Fakultät für Informatik

der Technischen Universität Wien

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

Wien, 8. Dezember 2020

Hermann Schützenhöfer	Andreas Krall

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

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

TU WIEN Informatics

Cycle-Accurate Simulator Generator for the VADL Processor Description Language

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Ing. Hermann Schützenhöfer, BSc.
Registration Number 1226141

to the Faculty of Informatics

at the TU Wien

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

Vienna, 8th December, 2020

Hermann Schützenhöfer Andreas Krall

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

Erklärung zur Verfassung der Arbeit

Ing. Hermann Schützenhöfer, 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, 8. Dezember 2020

Hermann Schützenhöfer

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

Acknowledgements

I want to thank my family, my friends and all colleagues at work who continuously supported me over the years.

And I also want to thank Prof. Andreas Krall for this opportunity to work on the VADL processor description language.

The section about the state of the art for Processor Description Languages (PDLs) in 2.1 has been written together with Alexander Graf (e1429203@stundent.tuwien.ac.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 WIEN Bibliothek Your knowledge hub

Kurzfassung

Ein Simulator ist eine Software, welche die Ausführung eines Programms, dass für eine unterschiedliche Plattform oder Architektur erstellt wurde, nachahmen kann. Ein Simulator kann also beispielsweise ein *RISC-V* Programm auf einem Linux Host ausführen und ist damit gerade während der Entwicklung eines Mikroprozessors, bevor die Hardware noch verfügbar wird, ein besonders nützliches Tool. Ein Simulator kann nun entweder per Hand geschrieben oder automatisch aus einer abstrakten Beschreibung des Prozessor-Modells durch einen Simulator-Generator erzeugt werden. Ein solches abstraktes Modell eines Prozessors kann in einer Prozessorbeschreibungssprache (PDL) beschrieben werden, welche typischerweise die Spezifikation der Instruction Set Architecture (ISA), der Application Binary Interface (ABI) sowie der Mikroarchtektur (MIA) beinhaltet. Ein Simulator der nur die Auswirkungen von ausgeführten Instruktionen reproduziert, wird üblicherweise als Funktionssimulator oder Instruction Set Simulator (ISS) bezeichnet. Ein Cycle-Accurate Simulator (CAS) kann auch die Anzahl der ausgeführten Prozessorzyklen ermitteln, wozu üblicherweise auch das Pipelinemodell simuliert werden muss.

Das Ziel dieser Arbeit lag darin einen ISS und CAS Generator für die neu geschaffene *Vienna Architecture Description Language (VADL)* bereitzustellen. Ein flexibles Framework zur Codeerzeugung wurde in *Xtend* zu diesem Zweck erstellt, welches einen C++ Simulator generieren kann und die Basis für beide Simulator-Generator Implementierungen bildet. Eine VADL Spezifikation für *RISC-V* wurde für die Evaluierung verwendet, welche die grundlegenden Integer Funktionen (I), Multiplikation und Division (M), Kontroll- und Statusregister (Zicsr) und komprimierte Instruktionen (C) in 32 und 64 Bit unterstützt hat. Sechs ISS Varianten und eine weitere für den CAS mit einer 5-stage Inorder-Pipeline wurden von dieser *RISC-V* Spezifikation abgeleitet und für Auswertungen und Tests verwendet. Für diese Simulator Varianten wurden jeweils mehre Benchmarks, wie etwa MiBench-Automotive, MiBench-Network and Dhrystone durchgeführt und deren Ergebnisse einer empirischen Analyse unterzogen und mit dem handgeschriebenen SWERV-ISS verglichen.

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

Abstract

A simulator is a software that can mimic the execution of a program, which has been created for a different platform or architecture. While a simulator can be used for example to run a *RISC-V* program on a Linux host machine, it is an especially useful tool during the development of a micro processor before the hardware is available. A simulator can either be implemented manually or automatically created from an abstract processor model by a simulator generator. An abstract model of a processor can be described by a Processor Description Language (PDL), which typically includes a specification of the Instruction Set Architecture (ISA), Application Binary Interface (ABI) and also the Micro Architecture (MiA). When a simulator only reproduces the effects of the simulated instructions it is commonly called a functional simulator or Instruction Set Simulator (ISS). A Cycle-Accurate Simulator (CAS) can also provide the number of executed cycles, which typically requires the simulation of the pipeline model.

The aim of this work was to provide an ISS and CAS generator for the newly created *Vienna Architecture Description Language (VADL)*. A flexible code generation framework has been created in *Xtend* for this purpose, which can emit a C++ simulator and builds the foundation for both simulator generator implementations. A VADL specification for *RISC-V* has been used for evaluation, which supported the base integer function set (I), multiplication and division (M), control and status registers (Zicsr) and compressed instructions (C) in 32 and 64 bits. Six ISS variants and one for CAS with a 5-stage inorder pipeline have been derived from this RISC-V specification and used for evaluation. A variety of benchmarks has been conducted for these generated simulator variants, including MiBench-Automotive, MiBench-Network and Dhrystone and the presented in an empirical comparison against the hand written SWERV-ISS.

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

Contents

Kurzfassung v

Abstract vi

Contents vii

1 Introduction 1
- 1.1 Instruction-Set and Cycle-Accurate Simulation . . . 1
- 1.2 RISC-V . . . 1
- 1.3 Motivation . . . 2
- 1.4 Problem Statement . . . 3
- 1.5 Aim of the Work . . . 3
- 1.6 Methodological Approach . . . 4
- 1.7 Organization of the Work . . . 5

2 State of the Art 6
- 2.1 Processor Description Language . . . 6
- 2.2 Instruction Set- and Cycle-Accurate Simulation . . . 12
- 2.3 Retargetable Simulation . . . 21

3 Simulator Generation 22
- 3.1 Code Generation Framework . . . 22
- 3.2 Structure of the Generated Simulator . . . 26
- 3.3 ELF Files . . . 29
- 3.4 Instruction Decoding . . . 30
- 3.5 System Calls . . . 33

4 Instruction Set Simulator Generator 37
- 4.1 Structure of the generated Simulator . . . 37
- 4.2 ISS Components . . . 37
- 4.3 Simulation Step . . . 41
- 4.4 Interactive Mode . . . 44
- 4.5 Termination . . . 46

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

5 Cycle-Accurate Simulator Generator **47**
5.1 Structure of the generated Simulator . . . 47
5.2 Example Pipeline Model with 6 Stages . . . 48
5.3 CAS components . . . 50
5.4 MiA Behaviour . . . 57
5.5 Instruction Partitioning . . . 65
5.6 Interactive Mode . . . 71

6 Evaluation **74**
6.1 Methodology . . . 74
6.2 Functionality and RISC-V Compliance . . . 75
6.3 ISS Performance . . . 77
6.4 CAS Performance . . . 84

7 Future Work **86**
7.1 Functionality . . . 86
7.2 Performance . . . 87
7.3 Usability . . . 88
7.4 Testing . . . 88

8 Conclusion **89**

List of Figures **91**

List of Tables **92**

Acronyms **93**

Bibliography **95**

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

CHAPTER 1

Introduction

1.1 Instruction-Set and Cycle-Accurate Simulation

An Instruction Set Simulator (ISS) is a software which can simulate the execution of programs intended for a given Instruction Set Architecture (ISA). A simulator can be an invaluable tool during development of a micro processor, especially when the hardware under design is not yet available. And a simulator can also give valuable insights into a micro processor implementation, by providing execution metrics, which are otherwise difficult to obtain. An ISS hereby mimics the functional effects that the execution of a instruction has on stateful components, like the memory or register files, and is often called a functional simulator. While limiting a simulation to only functional effects can yield good simulation speed, it also ignores many aspects of modern processor architectures and is therefore limited in the metrics it can provide. A more detailed simulation model, which also emulates the processor pipeline, is provided by a so called Cycle-Accurate Simulator (CAS), which comes at the cost of slower execution speeds.

1.2 RISC-V

RISC-V is a modular Instruction Set Architecture (ISA), which started as a project at UC Berkeley in 2010. Today, after ten years, RISC-V is a well known open and free ISA, with its global non profit organization called *RISC-V International*, counting hundreds of member organizations [risb]. Its open specification is hereby divided into two volumes, describing an unprivileged ([risc]) and privileged architecture ([risd]) and can be freely used for academic and commercial purposes. Following a modular design, the RISC-V specification is based on a small integer base (ie. RV32I/RV64I) that can be extended with additional modules to add features like multiplication, floating point or vector computation. While the specification was considered as immature for commercial

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

use in its early years, it has improved over time and today a considerable number of cores and System-on-a-Chips (SOCs) are available [risa].

1.3 Motivation

Various types of computing devices have become ubiquitous in our daily lives. But designing processors for such devices is a challenging task that requires trade-offs, not only in technical aspects but also development time or financial costs. And while the transistor count and complexity of processors has increased dramatically over the years, the overall product lifespan decreased at the same time. Under high time-to-market pressure and fierce market competition it has become vital to optimize the development process of computing devices wherever possible, while still exploring the design options to improve the characteristics of the final product.

Processor Description Languages (PDLs) are specialized tools, which are designed to describe the structural aspects of a processor, its behaviour or often both. A processor description, specified in a PDL, can be used to generate various artifacts like a compiler or simulator, and therefore, is a key component to enable rapid prototyping and the exploration of design options during the development of a new processor design. A specification written in such a PDL should be expressive, free of ambiguities, easy to create, adapt and maintain. Satisfying such language design objectives while also supporting a broad range of processor families and hardware components is quite ambitious, and will eventually result in trade-offs. Therefore, various PDLs exists like LISA, *nML* or *Sail*, which are tailored to a range of use cases and will often be of limited use when different conditions apply. The achievable benefits of using a PDL are at risk if only a single crucial hardware component or technique is not supported or can't be expressed by other means. So while various PDLs exist today, there are still single hardware features or combinations of them, which are not expressible with reasonable effort.

1.3.1 Vienna Architecture Description Language (VADL)

The newly created *VADL* processor description language will fill some of these remaining gaps. VADL hereby aims to be an expressive and extensible high level description language, suitable to describe processor designs for a wide range of RISC, CISC, and VLIW based architectures. With planned characteristic features like the support for instructions with multiple output values (ie. load and increment) and superscalar processors, it has the potential to push boundaries of today's processor description languages. While the VADL language is being designed at a first step with a focus on creating a compiler and simulator generator, as well as to enable hardware synthesis, its main design goal is to be an extensible language suited for many more usage scenarios.

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

1.4 Problem Statement

A key part of the VADL infrastructure will be a simulator generator that can emit a cycle-accurate simulator for a given VADL processor description. Such a simulator emulates the processor cycle-per-cycle based on the described micro architecture and can give invaluable insights to assert the performance of a processor specification. A cycle-accurate simulation has to include various aspects of a processor like memory access, pipeline behaviour and hazards.

1.5 Aim of the Work

The expected outcome of this work is the creation of a cycle-accurate simulator generator for the VADL processor description language. This goal does include a basic setup of the VADL development environment. The VADL language design will be incrementally developed and improved over the course of this project, to ensure that all necessary aspects for implementing the simulator generator can be expressed. But it is also the case, that many design and implementation aspects of a CAS generator do also apply to an ISS generator. The major difference is, that an ISS doesn't have to consider pipeline aspects and hazards and therefore, creating such a simulator generator in advance can give valuable insights into the design and problem domain of the later task at hand.

The desired outcome can, therefore, be summarized in the three following goals:

- **Work on the VADL processor description language**
  The VADL processor description language will be implemented from scratch based on the xText framework. The main goal of the language design is to provide a flexible but extensible high level abstraction, that can describe the behaviour and structure of a processor and its components. Separation of different language aspects and especially the re-usability of described components are hereby desired language properties. The VADL language will be designed from the beginning to support superscalar processor architectures.

- **Creation of an ISS generator**
  This generator will be able to create a functional simulator in *C++* for a micro processor description specified in the VADL language. Such a micro processor description will mainly consist of an Instruction Set Architecture (ISA) description and some constructs from the Application Binary Interface (ABI). The generated simulator will emulate programs provided in Executable and Linkable Format (ELF) and provide additional execution metrics like the number of executed instructions. The generated simulator will be based on interpreter techniques.

- **Creation of a CAS generator**
  The CAS generator will be build on the foundation of the ISS generator and shares many aspects of its general design and code base. The major implementation difference will be a more sophisticated simulation model, which is necessary to express

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

the pipeline design that is expressed by the MiA section of VADL descriptions. The generated simulator is also based on *C++* interpreter techniques and will emulate programs which are being provided by ELF files.

1.6 Methodological Approach

The methodological approach consists of the following steps:

1. **Literature Review and First VADL Draft**
   The first step of this project was to review existing literature on processor description languages to get a deeper understanding of the problem domain and potential design problems. This task also included to look into existing PDLs, instruction set architectures and processor pipeline designs.

   The next step was to set up the project environment, while also getting used to work with in the *XText environment* and the *Xtend programming language*. A first VADL language draft for describing ISAs was created after that and has successfully been used to express a *RISC-V* based processor architecture, including load-, store-, branch- and arithmetic-instructions. This early practical approach was a key aspect to identify shortcomings of the first implementation draft and used to improve the language design accordingly. Additionally the VADL language has been extended to also support the description of a Micro Architecture (MiA).

2. **Implementation of an Instruction Set Simulator Generator**
   After the initial language design phase an ISS generator has been build, which could emit an interpreter based simulator for the previously in VADL specified *RISC-V* processor architecture. The ability to execute programs in the generated simulator was a tremendous help to detect and resolve various specification errors in the created VADL *RISC-V* specification.

3. **Implementation of a Cycle-Accurate Simulator Generator**
   An extensible code generation framework has been extracted from the previously created implementation, before proceeding to the next project step. This approach significantly reduced the necessary development work for the CAS generator, by reusing many of the common and already created code generation functionalities. The emitted CAS is also based on interpretative simulation techniques and able to simulate various in-order pipeline designs for RISC architectures, which has been tested with various VADL specifications.

4. **Evaluation and Documentation of the Results**
   The last step was to summarize the project and its results, including examples for multiple VADL based processor descriptions in the final thesis.

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

1.7 Organization of the Work

The remainder of this work is structured as follows: Chapter 2 gives an introduction into the state of the art of ISS and CAS. This chapter contains an overview of various considerations and techniques for implementing various types of simulators and is the foundation for better understanding design decisions and also for possible enhancements in the future work section of Chapter 7. Chapter 3 will give an overview over the general structure of the simulator generator framework and describe the implementation details on instruction decoding, system calls and parsing ELF files, which are being used by both simulator generators. The following chapter will then extend on this foundation and describe the implementation of the ISS in chapter 4 and the CAS in chapter 5. Chapter 6 will then contain an evaluation section for both implemented simulator generators. The remaining two chapters describe possible ways to move forward with the current implementation in chapter 7 and concluding the work with a short summary in chapter 8.

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

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

CHAPTER 2

State of the Art

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 behaviour. Based on a high-level description of a processor architecture in a PDL, it is not only possible to automatically create various artefacts 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 artefacts 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 behaviour 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 artefacts.

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 artefacts like a cycle-accurate simulator, which needs detailed information about the Micro Architecture (MiA).

The second category are the **behavioural 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 behavioural 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 behavioural information and are therefore suitable to emit all possible artefacts 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 **behavioural** and **mixed** 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

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

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 **behavioural** 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 **behavioural** 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 **behavioural 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 behavioural 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 paths description, and the memory subsystem. A components specification defines the RTL components of the processor architecture, like pipeline units, functional units,

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

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 behavioural 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 **behavioural PDL**. However the behaviour 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 **behavioural** 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 semantics (using C/C++ like constructs), binary representation and assembly syntax. The semantics of the instruction are split into various section definitions, which describe

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

among other things their effect on the processor, simulation behaviour and the correlation to compiler instructions. One reason for the different behavioural 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 behaviour. 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 behaviour 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 machine cycles. An instruction set can be described in nML by a grammar that consists of *AND-rules* and *OR-rules*, which describe compositions

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

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 behaviour 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 behaviour 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 behaviour 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 [APF+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 functional and imperative language, which does support loops and recursion, but no

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

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 **behavioural PDL**.

2.2 Instruction Set- and Cycle-Accurate Simulation

2.2.1 Introduction

Simulation, in the context of this document, refers to the process of executing a program that was originally intended for a different system by a so called simulator. The system executing the simulator is hereby often referred to as *host* and the intended target system of the simulated program is often called *guest*. A simulator is hereby a software that is able to emulate the behaviour of the guest system and the simulated program code at a certain level of detail, depending on the simulation use case. But to simulate the execution of a program it is at least required to represent some aspects of the Instruction Set Architecture (ISA) of the target platform. This includes abstractions of various hardware components like register files and memory, as well as the semantics of each instruction defined in the ISA.

The most basic simulator performs a functional simulation of the executed program and is called an Instruction Set Simulator (ISS). By emulating one single instruction in each simulation step, this type of simulator doesn't have to consider pipeline hazards or other complex aspects of the Micro Architecture (MiA). By omitting such details from the simulation, an ISS has the potential for the best performance, at the cost of limiting the execution metrics that can be gathered during the simulation of the executed program. Examples of metrics that are commonly collected by an ISS are the number of executed instructions, system calls or taken branches.

While a purely functional simulation is already a very useful tool, it is not able to sufficiently characterise the runtime behaviour of a program that is being executed on a pipelined processor architecture. But with the simulation of a pipelined processor model, there are multiple overlapping instructions that are being processed concurrently, which also introduces data- and pipeline hazards. A simulator considering these aspects is commonly referred to as *cycle-accurate* and will simulate a single *clock-cycle* per simulation step. And while adding aspects like the pipeline to the simulation has commonly adverse effects on implementation complexity and execution speed, it also allows to get better insights in the real execution behaviour of the target platform. This is mostly due to additional metrics which can be collected by a *cycle-accurate simulator*, like for example the number of executed cycles and pipeline stalls. Such measurements can be essential information and a key component to optimize hardware and software components.

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

Rarely a simulator is based on an even more detailed timing model, which referred to as *phase-accurate*.

2.2.2 Use Cases

A major use case for simulators is the emulation of different ISAs on a host system. Emulating different guest systems is hereby especially useful during the development phase of new processors and processor architectures, while the hardware is not yet available or to allow an easier transition when architectures are being changed[BHK13]. [Wag15] adds to this point the aspect of providing backwards compatibility to previous hardware platforms, like *Rosetta*[ros], which was part of *Mac OS X* to emulate the previously used *PowerPC* architecture.

As mentioned before, simulators are able to gather various metrics and performance measurements from the executed programs. This can often easily be done by augmenting the executed code and these metrics can give crucial insights, necessary for optimization[BHK13].

But a simulator can also be a valuable debugging tool, especially if equivalent tools are not yet available in the guest system during development. By combining a simulator with a visualization it is also quite usable for educational purposes, like[GM19]

A retargetable simulator is also an invaluable component for design space exploration[MD11].

2.2.3 Instruction Decoding

Determining the type of an instruction from a given bit sequence is called *instruction decoding* and can have a major influence on the execution performance of a simulator. Instruction decoding is done by matching significant bits from the fetched instruction value against opcode patterns, that are defined to unambiguously identify an ISA instruction. While only some of the bits of the instruction are being used in the instruction decoding and considered as significant for this purpose, the other remaining bits will contain the operand and immediate values necessary for executing the instruction itself. In the context of instruction decoding these bits are sometimes named as *don't care* bits.

Further development of an architecture will often add new instructions and therefore also introduce changes to the instruction decoding process. While hand crafted instruction decoders can certainly be optimized specifically for a given architecture and therefore have the best performance prospects, their implementation can be error prone due to continuous changes and complex decoding logic. But automatic generation of an efficient instruction decoder from a given description is often not trivial, but can reduce overall implementation efforts and risks.

Algorithms to create an instruction decoder are commonly based on decision trees and therefore mostly differ on how to organize and split encoding entries into partitions. Such an example algorithm can be found in [The01], which is based solely on the machine code patterns and will only consider significant bits. But how to handle the don't care

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

bits can also be used for optimizations as described by the *unbound bit expansion* from [KA01]. This optimization can treat blocks of significant bits separated by a few don't care bits as one contiguous bit sequence by enumerating all possible values for the don't care bits and therefore reduce the depth of the decision tree. [KA01] also found that switch based decoder could outperform table based decoders in their tests.

With an increasing number of supported instructions by a processor there also is a tendency to end up with a more complex instruction encoding, due to the limited amount of bits to represent all distinct instruction types. When value exclusions patterns are additionally used to encode instructions then this is called an irregular encoding [OT16]. An example of this can be found for example in the compressed instruction format for *RISC-V*, where *C.MV* and *C.JR* are encoded by the same significant bits and are being distinguished by the encoded operand value of *rs2* ([risc]). The given bit sequence is decoded as *C.JR*, when the *rs2* value is zero and otherwise as *C.MV*. [FMP13] describes two algorithms to handle such decoding rules based on a reduced ordered binary decision diagram, which both can express the decoding of the *ARM V7* instruction set. [OT16] describes another approach capable of creating efficient instruction decoders for *ARM V7*, based on a description of significant bits and optional exclusion conditions.

The instruction decoding process can also depend on the current state of the processor. Such global state can be used to for example to switch between different sets of instruction decoding rules to support varying instruction widths. [RCS09] describe an approach to create multiple instruction decoders for a single simulator, in the context to support 32 and 16 bit decoders for *ARM V5*.

For *X86* there also exists an additional challenge in the so called *prefixes*, which can modify the following instruction decoding in significant ways. This includes for example changing the operand sizes or type of the decoded instruction itself . And even worse, while these prefixes can be optional or mandatory depending on the instruction being decoded, their order also has to be taken into consideration in some cases. Dealing with these constraints is certainly a challenge and one approach is for example the generic decoder specification language described in [SKS12].

For VLIW architectures, there also exist various encoding schemes, which aim to reduce the effects of NOPs on the code size. *Fixed-overhead*, *distributed*, and *template-based* encodings are described in [PKH+11].

Caching

It is also possible to reduce the instruction decoding overhead in a simulator by caching previously decoded instruction sequences. This is for example described by [Bed90] for instructions that are being executed multiple times in a threaded code interpreter or by [Mag97] who implemented a variant of the former threaded code interpreter for the *SimICS* ([MCE+02]) instruction set simulator. [Mag97] hereby reports that his refined implementation allowed to add extensive profiling mechanisms for instruction caches and control flow, while maintaining similar execution performance compared to the previous

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

implementation. [RCS09] cached decoded instructions and also multiple values from fetch address into a buffer, which resulted in a small improvement in execution performance. But caching the instruction decoding results are also found in recent papers. For example in [LIB15], which describes the implementation of such a caching mechanism for the *JIT compiler* of *Pydgin* by marking the decoding function as *trace elidable*.

But self modifying code is a general concern, whenever decoded instructions are being cached. Various strategies exist to detect and handle these changes. Like special coherence primitives to signal that a code cache has to be invalidated or performed tag checks against the instruction addresses to detect code modifications. A survey of detection and implementation strategies to support self modifying code in the context of ISS can be found in [Kep09].

2.2.4 Delegation of System Calls and Full System Emulation

Modern computer systems commonly isolate critical services, that are provided by the kernel of the operating system from less privileged user processes due to security concerns. Such services can contain up to hundreds of essential features to perform for example communication and process management tasks or accessing files and devices. A program running with user privileges can now request these functionalities from the operating system by issuing a system call. While implementations can differ for each operating system, a system call will usually trigger a switch of the processor context. The actual request will then be processed with higher access privileges and after it has been finished, the execution will be returned to the calling program.

A simulator without system call support would not have access to these essential functionalities and therefore would be quite limited in the applications it could execute. And like described by [BHK13] there exist two common ways of implementation. First by delegating each system call from the simulated program to the host operating system which runs the simulator. Depending on the system architecture and operating system of the host and guest system, it can be necessary to translate the system calls, their parameters and return values accordingly. The other much more sophisticated approach is called *full-system simulation* and attempts to simulate large parts of the architecture of the guest system. Full system emulation often include the necessity to deal with multiple complex components in the simulation like accessed devices, the memory model, interrupts and exceptions [Wag15].

Using delegation, like for example in [ALE02], is less work intensive to implement, but is not able to run an operating system and therefore, can't reproduce the same level of precision as *full-system simulation*. This precision difference is shown for example in [CLSL02]. But the overall implementation effort can be reduced by building on top of existing system emulators, like the open source project Quick Emulator (QEMU), which is usable for full-system and user-mode emulation for a variety of systems, including *X86* and *ARM*[Bel05]. The Micro Architectural and System Simulator (MARSS) *X86*

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

simulator is one example for this[PACG11]. And there also exist a variant of MARSS for *RISC-V*[mar].

2.2.5 Simulation Speed and Accuracy

While using simulators can be tremendously useful for the aforementioned use cases, they are also subject of two major concerns. First the simulation speed and secondly the achieved accuracy of the obtained runtime metrics. A simulation is after all still an abstraction of the system and therefore can have limited accuracy in predicting its behaviour. And so there is also a trade-off between creating a more detailed, but potentially slower simulator, on cost of better accuracy for the collected runtime measurements.

Many techniques have been proposed to increase the effective speed of simulators as described in more detail in 2.2.6. But [YKS+05] and [YL06] also give valuable recommendations besides technical improvements on the simulator to achieve better accuracy and tackling speed concerns. First the authors give examples from literature that every simulator implementation should be thoroughly evaluated, otherwise the gathered metrics can't be fully trusted. Secondly they also stress that setting appropriate parameters in the simulation for the simulated hardware components like memory is crucial to obtain meaningful and accurate results. Thirdly these papers also describe and summarize techniques to reduce long running benchmarks and input sets to a more tolerable runtime if the simulator would otherwise not have sufficient speed.

[CLSL02] reported on the significant impact of operating system effects on the simulator accuracy. According to the authors this also applies for benchmarks like the *SPECInt2000* that have been considered to be executed predominantly in user-mode.

2.2.6 Simulation Model

Interpretive Simulation

The most basic simulator can be realized as an interpreter, which repeats a fetch-decode-execute loop. A description for such an implementation can be found for example by the name of a *classical interpreter* in [Kli81]. This paper also compares a classical interpreter against an implementation using threaded code, introduced by [Bel73] and indirect threaded code, introduced by [Dew75]. In these reported results the direct threaded code interpreter had a performance advantage over the other two methods. A more detailed comparison on threaded and indirect code implementation techniques can be found in [Ert02], and [EG01] gives some additional insights on how to reduce branch misprediction penalties on virtual machine interpreters.

Another optimization can be to combine common instruction combinations into new *superinstructions*. This technique as described by [Pro95] can reduce the fetch-decode overhead and can also enable further code simplifications. [CEG07] found that interpreter performance was greatly limited by branch target mispredictions and tested the effects of replicating instructions in a VM in combination and separately with *superinstructions*.

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

Both methods can hereby be applied at interpreter build-time or during the runtime of the interpreter and showed best results when applied together. [RSS15] report that indirect branch prediction is nowadays less of a problem for interpreters and attribute these findings to hardware improvements, especially in the area of indirect branch predictors like the ITTAGE ([Sez11], [SM06]). The authors of [RSS15] therefore consider instruction replication no longer necessary to reduce the indirect branch misprediction penalty of interpreters.

Compiled Simulation and Static Binary Translation (SBT)

[MAF91] originally described the technique of *compiled simulation*, which translates a given application program into a specialised simulator that is only capable of executing this one application. This implementation was heavily based on *in-line macro expansion* of the *C compiler* and the semantics of the *C programming language*, to translate the assembler input application into a C program. The C compiler could then apply static optimizations to the created C program, which resulted in improved execution speeds, at the cost of its compilation time. Additionally the translation for a given program also moves the cost of instruction decoding from run-time to compilation time. But this approach also comes with some limitations on the emitted code size and the exclusion of self modifying code, because the application code has to be fully known when the specialised simulator is being created. The limitations of emitted code size have later been removed by a refined technique described by [BARA04], which partitions the code into regions which are being contained in separate functions.

Another variant of compiled simulation is described by [RMD03]. While instruction decoding is performed at compile time, the created simulator can still detect code changes at runtime and even has the ability to decode and execute the modified code with an interpreter. Therefore, this approach can achieve better execution speed for most programs, without limitations on the executed programs.

[FKH07] describe a cycle-accurate simulator approach for a VLIW processor, which translates a target program based on its basic blocks into an equivalent C program. This focus on larger compilation units and the ability to specialize the execution paths by basic block duplication, results in execution times that are orders of magnitude faster than those of interpreters. This implemented simulator can also switch from compiled code to an interpreter and vice versa to handle self-modifying code and indirect branch targets, for which the jump address is not yet known at compile time.

Another approach is Static Binary Translation (SBT), which will emit native machine code, in contrast to emitted high level code by *compiled simulation*. Otherwise SBT faces the same challenges and yields the same potential benefit than *compiled simulation*. One SBT implementation targeting specifically embedded applications for *ARM* based on the LLVM compiler infrastructure is described in [SCHY12]. The authors argue that self modifying code is rarely used in embedded applications and therefore no concern and also describe their approach, for code discovery in the context of variable length instruction

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

encodings, and handling indirect branches. With some implemented optimizations they report significant runtime improvements compared to benchmarks emulated with QEMU.

Dynamic Binary Translation (DBT)

When the machine code generation is performed during simulator runtime, than the translation technique is called Dynamic Binary Translation (DBT). Typically this is implemented by a combination of an interpreter with a JIT compiler. Usually the simulation starts by interpreting the instructions, while also collecting profiling data to detect frequently executed code fragments. To reduce the compilation time, only often executed code pieces will then be translated into native machine code by the JIT compiler. In contrast to SBT, this technique can also handle self modifying code. And even if only a limited set of compiler optimizations can eligibly being begin used in this context, without having a detrimental effect of the simulation, this technique still yields very good execution performance in practice. [BHK13] lists *register allocation* as the most important optimization in this context, followed by *liveness analysis* to remove obsolete computations.

Some variants of DBT exist, depending on the size of the code fragment they are targeted at. [Wag15] gives a good overview on block-, trace- and region-based translation methods. By compiling bigger portions of the code, more of the control flow is becoming visible during the compilation. While this gives more optimization opportunities, it can also have a significant impact on the compilation overhead of the simulation.

But the size of compiled code fragments don't necessarily has to be static. [BFKR09] describe a LLVM based JIT simulation infrastructure, which can create a CAS from an ADL specification. While this implementation starts to only compile basic blocks it can later switch to compiled regions with non-linear code for often executed code blocks.

But advanced speculative optimization techniques can also be used in context of DBT. One example is [DGGL17], which describes a trace-based DBT approach to execute *AArch32* code on an *AArch64 instruction set*, with speculative optimizations to efficiently handle different addressing modes.

A similar approach to DBT is described for the *Pydgin ADL* by [LIB15]. This project is based on *RPython*, which provides infrastructure to implement dynamic languages and provides support for *JIT compilation*, based on a restricted subset of *Python*. An ISS described in *Pydgin ADL* can be translated to C code with an added *meta-tracing* based *JIT compiler* to achieve fast simulator performance. The *meta-tracing JIT compiler* hereby analyzes traces on how the interpreter is interpreting the bytecode and not working on direct bytecode traces. With additional *meta-tracing annotation* to guide and optimize the compilation the authors could report high simulator performance between 90 and 750 MIPS for selected *SPEC CINT 2006* benchmarks. A description for *SMIPS* and *ARMv5* is mentioned in [LIB15] and more details for a *RISC-V* simulator are given in [ILB16].

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

Hardware Supported Simulation

Differences between the host and guest system can have a tremendous effect on the simulation implementation effort and the obtained simulation speeds. One such example is given by [BHK13] describing a host with 32 bit register values trying to simulate a 40 bit DSP. Each simulated operation in this scenario would have a significant overhead due to this system mismatch. Therefore, depending on the simulation task at hand, it could be a quite effective to rely on dedicated hardware in parts of the simulation. One such example is a simulation platforms based on Field Programmable Gate Arrays (FPGA). While this has a quite different focus than this work, please refer to [BHK13] for an overview on this particular topic.

Cycle-Accurate Simulators

A simulator model which is capable of cycle-accurate simulation does not only have to preserve the semantics of the instructions, when being executed in a pipeline, but also has to consider various effects like pipeline and data hazards. Features like out-of-order execution or supporting various hardware components also increase the complexity and can have a detrimental effect on the maintainability of the codebase and its execution speed. Supporting a model which has been tailored to the use case at hand, can for this reason be beneficial and some implementations, like [You07], even provide multiple alternative simulation cores. Due to such considerations and various possible approaches to the topic, there exist different techniques for modelling and executing cycle-accurate simulators.

[ZPM96] modelled the cycle-accurate model by partitioning the simulated instructions into schedulable operations and modelled their timing constraints by a modified *Gantt chart*. [PHZM99] replaces this mechanism with a timing model, where the designer can explicitly assign the operations to fine grained control steps, at the level of instructions, clock cycles or phases. Operations can also explicitly trigger common pipeline operations, like flushes and stalls.

[PMHH09] describes an interpretive technique, which has been used for the implementation of the Instruction Set Architecture C (ISAC) language. This technique is based on a formal model, which as the authors claim, enables an easier validation of the simulator. Their model is based on finite automata and created from a description of the instruction set and a behaviour- and timing model. The authors of this paper report good runtime performance in tests against a *MIPS* and VLIW architecture, due to static scheduling for events of the implemented timing model.

Models based on *petri nets* are quite suitable for representing pipelined architectures[Raz87]. But depending on the way the model is represented it could suffer from an exponentially growth in size. [RD05] describe a modified approach which the authors call *reduced colored petri nets* and show an alternative way to express data hazards. This paper reports significant runtime improvements of the implementation of a cycle-accurate-simulator

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

compared to *SimpleScalar*. [YWZW16] used a *colored peri net* to model a cycle accurate simulator for a VLIW architecture.

PTLSim is a cycle-accurate full-system simulator for the *X86-64 architecture*, which is working at the micro-operation level and has been implemented by [You07]. The full-system emulation support is based on the *Xen Hypervisor* and the simulator implementation provides a *co-simulation feature*, which allows to switch between native execution and the cycle-accurate simulation during runtime. This has the advantage that program code can be executed natively until the start of a benchmark, before the simulation has to switch to the slower cycle-accurate simulation mode. MARSS is a full-system multicore X86-64 simulator, based on a modified version of *PTLSim*. This implementation uses QEMU instead of the *Xen Hypervisor* and has MMX support [PACG11].

The open source gem5 simulator project also allows hardware modelling at a cycle level for multiple architectures like X86, ARM and RISC-V and has been build by a large community of contributors over nearly a decade and therefore can showcase how a software system can continuously evolve in this domain[LPAA+20].

Cycle-Approximate and Cycle-Count-Accurate Simulation

Estimation techniques can also be an alternative for slow cycle-accurate simulation techniques in some circumstances and under the condition that the estimated cycles are close to cycle-accurate but also can be computed at much lower cost. One such approach is described by [HAG08], which is based on automatically generated transaction level models and suitable for multicore designs. This technique annotates basic blocks with estimated delays, that considers the number of cycles based on an operation schedule, cache-miss delays and branch mispredictions. The authors of this paper report an average error rate for the predicted cycles of 8%, but the error rate is often higher for single test cases. Another approach is described by [Fra08], which is based on a regression model that is trained with data from an ISS and CAS and should then be able to estimate the number of cycles only from the ISS inputs.

Another alternative are cycle-count-accurate simulation techniques, which try to accurately simulate the effects of a pipeline while eliminating all unnecessary details from the simulation. One such example is found in [LCWT11], which is a technique that computes the pipeline execution behaviour for all basic blocks of a program. The execution behaviour is then combined with the control flow information during a static analysis for the executed program to obtain a complete model of all possible pipeline execution behaviours (PEBs). This model allows to efficiently compute the effects of a basic block while preserving a cycle-accurate result at much faster speeds. But this technique is only suitable for the simulation of inorder pipelines. A similar approach that can also be used for superscalar processors is described by [LDT13]. This technique determines the timing states for a single instruction for the involved stages and can then separately simulate the functional effects of the instruction similar to a ISS. But the adaptation of

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

this instruction-oriented approach for an out-of-order execution model is only described, without showing any results.

2.3 Retargetable Simulation

While a simulator can be created for just one specific target architecture, it is generally a desirable feature to support a broader range of possible targets. This ability to cope with different processors and processor architectures is typically called retargetability and requires some form of specification of the target system and its behaviour. Using ADLs seems like a natural choice in this context [Wag15]. Based on such specifications it is possible to automatically generate a stand-alone simulator or at least some components for an existing simulator framework.

Simulators that have only limited support for retargetability features, can potentially be very fast, because they can be specialized for specific target platforms [MD11]. Parametrization is hereby the most basic form of retargetability, which can be used to alter simple settings like the number of registers for example. Creating a retargetable simulator with support for a wide variety of architectures and features, typically comes at the cost of a more general implementation, which can't provide the same optimizations as hand written simulators[BHK13].

Retargetable simulators have in general the same benefits, concerns and limitations as described in 2.2, but with additional challenges that parts or even a complete simulator is being created by a code generator.

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

CHAPTER 3

Simulator Generation

3.1 Code Generation Framework

3.1.1 Introduction

The VADL project supports the generation of an ISS and a CAS from a given VADL micro processor and micro architecture description. Despite the implementation differences, there still exist many common aspects that apply to both simulator generators and are therefore encapsulated in a common framework. This section describes this shared code generation framework, its structure and all common aspects which apply to both simulator generators in general. The subsequent chapters will then continue to extend on this foundation, giving further specific implementation details.

3.1.2 XText and XTend

Projects based on code generation can quickly become hard to maintain, especially when a project grows in size. By building the VADL project based on *Xtext* ([xteb]), this also enabled the use of the statically-typed *Xtend* programming language ([xtea]) throughout the project. Code written in *Xtend* will be compiled into *Java source code* and can directly interact with *Java sources* without any further obstacles. But the real advantage of using *Xtend* are language features like multiple dispatch, switch expressions and property support, which are not yet available in the Java programming language. These features combined with the integrated template engine make the *Xtend* programming language especially useful for creating code generators. Writing the whole simulator generator infrastructure purely in *Xtend* instead Java, therefore, avoided much of the otherwise necessary boilerplate code.

Listing 3.1 shows a basic example of how *Xtend* is being used in the framework to generate a *C++* enumeration, containing all instruction names from a VADL description.

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

The InstructionTypeGenerator from this example provides a generate(ctx) method, which will render the *C++* output. The content between the triple single quotes is a *Xtend template*, which allows to mix static output text with dynamic evaluated expressions, contained in guillemets («»). Dynamic expressions can be used to reference values like string constant NAMESPACE or the listInstructions method in the example.

```
class InstructionTypeGenerator <T extends SimulatorGeneratorContext>
    implements SimulatorFragmentGenerator<T> {

  override generate(T ctx) '''
    #include <ostream>
    namespace «NAMESPACE» {

    typedef enum {
      UNKNOWN_INSTRUCTION,
      OP_INSTRUCTION,
      «listInstructions(ctx)»
    } «INSTRUCTION_ID» ;
  }
  '''

  def listInstructions(T ctx) {
    ctx.isa.instructions.values()
          .map[instruction | instruction.name]
          .join(",\n")
  }

}
```

Listing 3.1: Xtend Example of a Generator

3.1.3 Simulator Generator Framework

The simulator generator framework emits a standalone *C++* based simulator with *CMake* build files. Each generated simulator is hereby assembled from static resource files, which are being copied from a template folder, and also files that are dynamically created from VADL descriptions. This high level overview is shown in Figure 3.1.

Including files and folders from a static template directory is hereby an easy mechanism to include external or internal libraries and distribute license information into a generated simulator.

Each dynamically created file is generated by a single *generator class*, which only has to implement the generate(ctx) method specified in Listing 3.2. All dynamically generated files are enumerated in a single *simulator generator class*, that basically associates the emitted filenames to a generator, which provides the file content. While parameters can be passed to generator classes to customize their behaviour, it is also possible to control which files are being generated at all in the simulator generator class. This is a simple,

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

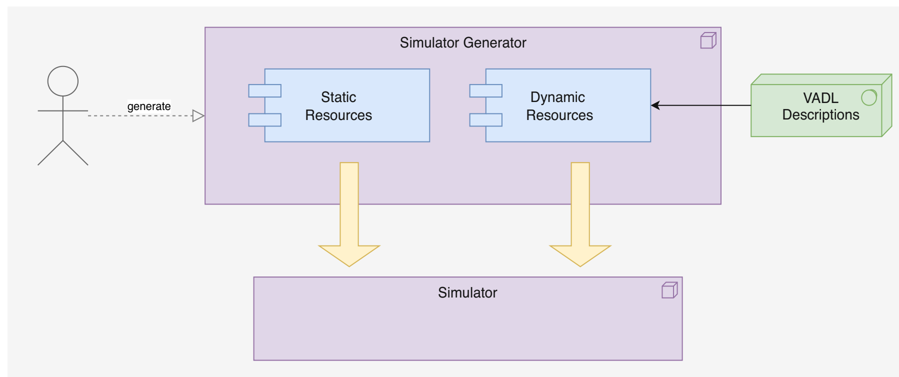

Figure 3.1: Basic Simulator Generator Overview

yet powerful mechanism that can be used to generate different classes, depending on the target environment and parameter values, and also very practical in its application to encourage code reuse.

While a generator class is intended to create a complete file, there also exists *fragment generators*, which are being used to generate common or shared code, which can be utilized in multiple generator implementations. An example for such shared code would be a static utility method for sign extension or the instruction decode logic, which is dynamically generated from VADL descriptions. To allow utmost flexibility, the fragment generator classes do not necessarily have to implement a shared interface, and are only identified by their naming scheme. And additionally fragment generators can also be useful to further split up a generator implementation, which has become to large. An example for a simulator generator is given in Listing 3.3 and a generator is show in Listing 3.1.

```
interface SimulatorFragmentGenerator<T extends SimulatorGeneratorContext> {
        def String generate(T ctx)
}
```

Listing 3.2: Generator Interface

```
class BasicSimulatorGenerator
    extends BaseGenerator<BasicSimulatorGeneratorContext> {

  new (BaseProcessorModel processor) {
    super(createSimulatorGeneratorContext(processor))

    // list files which have to be created for Basic Simulator
    add(MAIN_CPP, new MainGenerator())
    if (isLinuxEnvironment()) {
      val name = ‘Processor‘
```


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

```
        add(PROCESSOR_HPP, new ProcessorHppGenerator(name))
        add(PROCESSOR_CPP, new ProcessorCppGenerator(name, PROCESSOR_HPP))
      }
      //...
    }
}
```

Listing 3.3: Simulator Generator Example

CMake is being used as build tool for the generated simulator. The necessary classes are hereby automatically generated by the base class of the simulator generator that lists all dynamically created files. A *CMakeLists.txt* file is being generated for each folder of the generated simulator, containing a listing of all C++ files. The content of the CMake files is being created by using *Xtend templates* and can therefore be easily adapted if necessary.

Simulator Generator Context and Utility Classes

A code generator requires access to varying data values, depending on the code that should be emitted. While limiting data access to a bare minimum is a good programming practice, the number of passed through arguments in the generator methods is increased dramatical during the first prototyping. Some methods even had more than 10 arguments and at such a point it becomes a clear hindrance for code changes and improvements. Therefore, a context object was created, which encapsulates the access to common resources and making most previously done parameter passing obsolete.

This simulator generator context object is being created in the previously shown simulator generator class in Listing 3.3. With the use of generics, the context can be extended and provide additional data values depending on the simulator generator type.

A context always provides access to the micro processor model and various utility classes, which can be further customized during the initialization of the context object if necessary.

The first utility class is the `VariableTypeProvider`, which can return a C++ type name, that is suitable to store a signed or unsigned value of a given bit length, without unnecessarily wasting memory. For example a decoded unsigned 12 bit value will get the C++ type `uint16_t` assigned. The available data types are hereby statically registered in the utility class and can easily be changed if necessary. If no suitable type is available for a requested bit length, then an exception is being thrown and making the problem instantly visible to the VADL user. Continuously using this utility class also makes future changes to C++ types transparent throughout the simulator generator.

A second important utility class is the `IdentifierProvider`, which can create valid C++ variable names from a given identifier string. Each identifier string is furthermore associated to the generated C++ variable name, and will consequently return the same value again for the stored identifier string. While this utility class primarily filters out special characters to avoid C++ compile errors, this is also useful to change variable naming schemes and provide unique temporary variable names. Additionally, this class

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

can also help to avoid naming conflicts between identifiers, that are being specified in VADL descriptions and internal variable names used by the simulator generator. But variable names that are being used by the generator have to be registered with the IdentifierProvider which then will consequently add an index to each conflicting variable name accordingly. Multiple identifiers which would get the same variable name assigned after filtering special characters will also be indexed by this utility class.

Origin of Generated Code and Constants

It is a major concern for the long term maintainability to reproduce the origin of generated code in the simulator generator code base. While errors will probably be detected during the execution of programs by the simulator, their origin can be an erroneous VADL description or a bug at various points of the VADL code base. Debugging the code generation can be a good starting point to further isolate the exact problem area and is therefore such a crucial aspect for maintainability.

The strict structure of the code generation framework and that all generated filenames are listed with their content provider in a single class should help to quickly find the code origin. But sometimes it is not so easy to reduce a problem during debugging to a specific file. Let's assume for example that a wrong pipeline flush signal is being send to the cycle-accurate-simulator and it is not clear which component is actually issuing this signal. In such cases there exists a second mechanism, which can be used to find all relevant code generation fragments by searching over the constants, which are being used in the code base.

Constants are being used to define specific values like internal processor states throughout the simulator generator, but are also being used for public method names, that are being called from different components. By using constant values in the code base, allows to quickly refactor various generated names if necessary, but also helps to locate where those values are actually being used in the simulator generator logic.

3.2 Structure of the Generated Simulator

Both generated simulator types create a main.cpp class as single entry point. The implementation in this class is responsible for parsing the arguments, instantiating a processor implementation, reading the ELF input file into simulator memory and starting the simulation. The generated processor implementation is hereby instantiating all necessary components, like memory, register files and flags, and providing an execution loop, which will run until a stop signal has been issued.

The generated C++ simulator contains *CMake* build files, which provide debug- and release build targets. The simulator also supports program arguments to issue an instruction trace for simulated instructions or even start the simulation in an interactive mode, which is described in detail in the next section (3.2.1). The commands to build

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

and execute the ISS are given in Listing 3.4. The commands for the CAS are the same, except for the simulator executable, which is named ca_gen.

```
cmake .
make release
./isa_simulator elf_executable
```

Listing 3.4: Commands to Build and Start a Generated Simulator

3.2.1 Instruction Trace and Interactive Mode

During the development of the ISS it became quite apparent that debugging is one of the major implementation concerns. While the first compiled test programs for RISC-V, like the one in Listing 3.5, were small in size, this changed dramatically when programs have been successfully linked with the C++ standard library.

Printing out the executed instruction trace to the console or an output file was one of the first measures to improve the debugging capabilities for the ISS. An example trace for the simple addition program in Listing 3.5 is shown in Listing 3.6. The instruction trace shows one executed instruction per line, starting with an ascending instruction number, followed by the address where the instruction has been fetched from. After that the instruction name is printed with the decoded fields from the instruction format and decoded immediates. The printed parameter values are sorted by name and *ANSI escape codes* are being used to colour the instruction trace when printed to console to improve the readability.

A handwritten simulator can easily be optimized to generate a more readable instruction trace, because it can be adjusted to the most suitable information for each instruction. The [Rah] simulator would for example show for the first ADDI of the shown instruction trace the information: addi x2, x2, −0x20. This is on one hand more compact by following the assembler syntax but also gives more concise information on how the register destination (rd) is actually being used as index for the X register file. While the output of the instruction trace can also be improved by performing an analysis how decoded values are being used, like for example as register file index. But already the relatively simple instruction decoding of RISC-V shows that anomalies can occur, like for the CSRRCI instruction, where the bit sequence specifying the first source register index (rs1) is being used as 5 bit immediate value instead.

```c
int main() {
   int a=3, b=4;
   return a+b;
}
```

Listing 3.5: Simple C program (add.c)

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

```
#1 10054 - ADDI: rd: 2 (0x2), rs1: 2 (0x2), imm: -32 (0xffffffe0)
#2 10058 - SW: rs1: 2 (0x2), rs2: 8 (0x8), imm: 28 (0x1c)
#3 1005c - ADDI: rd: 8 (0x8), rs1: 2 (0x2), imm: 32 (0x20)
#4 10060 - ADDI: rd: 15 (0xf), rs1: 0 (0x0), imm: 3 (0x3)
#5 10064 - SW: rs1: 8 (0x8), rs2: 15 (0xf), imm: -20 (0xffffffec)
#6 10068 - ADDI: rd: 15 (0xf), rs1: 0 (0x0), imm: 4 (0x4)
#7 1006c - SW: rs1: 8 (0x8), rs2: 15 (0xf), imm: -24 (0xffffffe8)
#8 10070 - LW: rd: 14 (0xe), rs1: 8 (0x8), imm: -20 (0xffffffec)
#9 10074 - LW: rd: 15 (0xf), rs1: 8 (0x8), imm: -24 (0xffffffe8)
#10 10078 - ADD: rd: 15 (0xf), rs1: 14 (0xe), rs2: 15 (0xf)
#11 1007c - ADDI:rd: 10 (0xa), rs1: 15 (0xf), imm: 0 (0x0)
#12 10080 - LW: rd: 8 (0x8), rs1: 2 (0x2), imm: 28 (0x1c)
#13 10084 - ADDI: rd: 2 (0x2), rs1: 2 (0x2), imm: 32 (0x20)
#14 10088 - JALR: rd: 0 (0x0), rs1: 1 (0x1), imm: 0 (0x0)
```

Listing 3.6: Instruction Trace for add.c

Another problem of using a simple instruction trace is that it doesn't contain dynamic values, like the content of register files and memory locations. While this information is absolutely necessary for debugging, it also would excessively increase the output size of the trace. To still give an user access to dynamic values and even more additional information, an interactive simulator mode has been added, similar to the ones found in handwritten simulators like [Rah]. This mode can be started by the simulator option -i and allows the user to execute the program line by line and to print values from all stateful components like register files and memory locations. The actual output of the interactive mode differs a lot between the ISS and CAS and will be described in detail in the following chapters 4 and 5.

But it is noteworthy at this point, how this interactive mode is actually implemented. While the first implemented version for the ISS contained a single method to simulate each ISA instruction, those methods have just been duplicated with additional debug facilities for the interactive mode. These additional methods, combined with a modified version of the main interpreter loop already was a working implementation for the interactive debug mode. But the major drawback was that the generated main processor class nearly doubled in size, and while inconvenient when reading through the generated code, it also significantly decreased the simulator speed for the standard mode. This was assumedly caused by increased instruction cache pressure, but was not analysed in more detail, due to time constraints. To avoid the detrimental effect on the simulation speed, the interactive mode is now implemented in an additional InteractiveProcessor class, which contains the duplicated code and has no effect on the simulator speed in standard mode any more. Additionally this design also has the advantage, that a user can manually add debug outputs to the InteractiveProcessor implementation for debugging purposes, without breaking the standard mode implementation.

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

3.3 ELF Files

3.3.1 ELF File Format

The Executable and Linkable Format (ELF) is a common binary file standard which can encode multiple different formats, including object and executable files. Each ELF file starts with a header, defining basic global attributes for the file content, like the system architecture, ABI and that it is encoded for example in 2's complement and little endian. The header also provides start addresses for the program and section header, in which the actual data are being organized. While the program header consists of information to create the process image, the sections of the ELF file do contain the actual code and data of the executable. The available sections differ depending on the type of file which is being encoded in ELF. For example an executable file will contain sections describing how the segments should be loaded into the operating system memory, while object files contain sections that are relevant for the linker. A specification for the ELF file format can be found in [C+95] and the homepage at [elfb] gives a very practical introduction based on a hello world program and also shows various tools for working with ELF files.

3.3.2 ELFIO Parser Library

Creating a parser for the ELF file format is a erroneous and rather time consuming task. Therefore, the simulator relies on an open-source C++ library called *ELFIO*, which is being available under the *MIT licence*. The source code is available under [ELFa], with the tutorial and user manual found at [elfc]. This library is only being used in the entry class of the generated simulator to parse the binary input file, and can therefore easily replaced by a different or self created parser if the need will ever arise.

The example in Listing 3.7 shows the code steps to load an ELF executable and copy the loadable segments into the memory of the simulator. By using the *ELFIO* library the handling of the input file format is mostly transparent for the simulator implementation and only a single writeFetchMemory(address, value) interface has to be provided to initialize the simulator memory.

```
// read ELF file
elfio reader;
if (!reader.load(filename)) {
  cerr << " failed to load ELF file " << filename << endl;
  return -1;
}

// read all loadable segments into memory
for (int index = 0; index < reader.segments.size(); index++) {
  auto *seg = reader.segments[index];
  const char* data = seg->get_data();
  if (seg->get_type() = PT_LOAD) {  // only loadable segments
    auto vaddr = seg->get_virtual_address();
    auto size = seg->get_file_size();
```


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

```
    for (size_t i = 0; i < size; ++i)  {
      processor->writeFetchMemory(vaddr + i, data[i]);
    }
  }
}
```

Listing 3.7: Loading ELF sections into memory

3.4 Instruction Decoding

3.4.1 VADL description

The instruction decoder is generated from all instruction encoding and instruction format definitions for a given VADL micro processor description. An example for an instruction format specification is hereby given in Listing 3.8, which shows a subset of the base instruction formats for RISC-V. A format is defined by its overall bit length and a list of fields, that associate a name with a sequence of bits from the format. Each VADL instruction definition also has to be assigned to an instruction format, which makes the instruction format fields available, when the actual instruction encoding is being defined, like in the example in Listing 3.9. For RISC-V most of the instruction format fields are contiguously defined, like for example the *opcode* from the shown example formats, which is stored in the 7 least significant bits. But it is also possible to define format fields that are not contiguous like the *offset* of the J-Type, which is a concatenation of the most significant bit (31), followed by the bits 19 to 12, then adding bit 20 and bits 30 to 21, in this given order. Bit ranges specified in the brackets are always inclusive, that means that the *offset* has a total length of 20 bits.

The encoding shown in Listing 3.9 also represents a subset of actual RISC-V encodings for two I-Type instructions (LB, LH) and two R-Type Instructions (ADD, SLTU). Specifying the instruction encoding bit patterns in binary format typically corresponds to the definition found in the specification of the ISA. The additional apostrophe, like for example in 0b000'0011 is only for readability and is filtered out during parsing of the VADL file.

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

format I_TYPE : bit<32> = {
  imm             [31..20]
  rs1             [19..15]
  funct3          [14..12]
```


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

```
  rd              [11..7]
  opcode          [6..0]
}

format J_TYPE : bit<32> = {
  offset          [31, 19..12, 20, 30..21]
  rd              [11..7]
  opcode          [6..0]
}
```

Listing 3.8: R-, I- and J-Type Instruction Format for RISC-V

```
encoding LB = { opcode=0b000'0011, funct3=0b000 }
encoding LH = { opcode=0b000'0011, funct3=0b001 }
encoding ADD = { opcode=0b011'0011, funct3=0b000, funct7=0b000'0000 }
encoding SLTU = { opcode=0b011'0011, funct3=0b011, funct7=0b000'0000 }
```

Listing 3.9: Instruction Encoding Example

3.4.2 Simulator Generator Implementation

The instruction decoder for both simulator types is created by the generator class named DecodeInstructionFragmentGenerator and its output for the example definitions from RISC-V is shown in Listing 3.10. The implementation works by partitioning instructions similar to a binary decision tree. The recursive algorithm works by finding the next format field, that can be used to decode the majority of the remaining instructions. This format field is then parsed into a local variable and a switch statement is emitted to split the affected remaining instructions, based on this variable. Only if an instruction is completely decoded by all its defined encoding fields, then a return statement is being issued, otherwise the decode method is being called in a recursive descent, decoding increasingly more format fields for the current branch of the descent.

It is important to note that the name of the instruction format fields is not used in the algorithm, but only the bit index positions. This avoids naming conflicts and inefficiencies, when instruction formats would for example identify the same bit index positions by different names.

The implemented algorithm detects the relevance of the opcode for the given RISC-V example in Listing 3.10. Therefore, the opcode is parsed first and used to partition the instructions with a switch statement. The instructions *LB* and *LH* share the same opcode value and are only distinguished by the additional format field funct3. The process works similar for the *ADD* instruction, but here funct3 and funct7 have also to be considered in the decoding process.

```
InstructionId Processor::decodeInstruction32(const uint32_t instr) {
        uint8_t rs1, funct3, funct6, funct7, opcode;
        uint16_t imm;
```


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

```
        opcode = instr & 0x7f;
        switch(opcode) {
                // ...
                case 0x3:
                        funct3 = (instr >> 12) & 0x7;
                        switch(funct3) {
                                case 0x0: return LB;
                                case 0x1: return LH;
                                // ...
                        }
                        break;
                case 0x33:
                        funct3 = (instr >> 12) & 0x7;
                        switch(funct3) {
                                case 0x0:
                                        funct7 = (instr >> 25);
                                        switch(funct7) {
                                                case 0x0: return ADD;
                                                // ...
                                        }
                                        break;
                                case 0x3:
                                        return SLTU;
                        }
                        break;
                        // ...
        }
        return UNKNOWN_INSTRUCTION;
}
```

Listing 3.10: Example Instruction Decoder for RISC-V

3.4.3 Irregular Instruction Encodings

The described algorithm does also work for irregular instruction encodings, which can be found in the compressed RISC-V instruction set specification. An example is shown in Listing 3.11, where both instructions are encoded by the same opcode and funct4 values, but *C.JR* is only detected, when the rs2 value is zero, which is prohibited for *C.MV*. The implementation of the algorithm works for these cases, because all specified format fields in the encoding are recursively and eagerly being processed until no ones are left. Therefore, *C.JR* with the additional condition of rs2 being zero, is always processed before *C.MV* in this case.

```
encoding C_JR = { opcode = 0b10, funct4 = 0b1000,  rs2 = 0b00000 }
encoding C_MV = { opcode = 0b10, funct4 = 0b1000 }
```

Listing 3.11: Example of Irregular Instruction Encoding for RISC-V

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

3.4.4 Multiple Instruction Format Widths

The compressed instruction set for RISC-V uses an instruction width of 16 bit instead of 32 bits. The instruction decoder will emit one decoder method per instruction format width, specified in the given VADL description. And calls to these decoder methods are being issued in ascending order, until the instruction has been successfully detected or causes an exception due to an unknown instruction format.

3.4.5 Limitations of the Current Implementation

The current instruction decoder is already able to decode a wide range of possible instruction encodings, but lacks support for some advanced decoding features. First the current decoder can't access the state of a processor and therefore doesn't support switching between different instruction-set encodings at runtime. Very complex encoding schemes, like the prefixes from X86, are also not supported and would require additional features like optional encoding arguments and even to consider the argument order. And compressing schemes for multiple issued instructions, which are common to reduce code size for VLIWarchitectures, are also not supported yet.

3.5 System Calls

3.5.1 VADL description

The generated simulators have basic system call support by relying on delegation to the host operating system. That means that system calls from the guest system are being issued to and executed by the operating system of the host. Depending on the operating system and architecture of the host and guest system, a translation of arguments, return values and system call numbers may be necessary. The ABI section of a VADL description specifies the structure of a system call for the guest system. An example for RISC-V is shown in Listing 3.12. Each system call hereby consists of a number identifying the system call, very similar to a method name. For the example the system call number is stored in the registerfile X at index 17. The operands for the syscall are passed through the registerfile from index 10 to 15 and the return values will be contained after the call in index 10 and 11.

```
system call = X[17] : ( X{10..15} ) -> X{10..11}
```

Listing 3.12: VADL SysCall Defintion for RISC-V

3.5.2 Instruction Trace

The instruction sequence to issue a system call for simulated RISC-V guest system is shown in Listing 3.13. The instructions from number #598 to #602 are hereby setting the operands to value zero, while the instruction with number #603 specifies the system

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

call number to value 80. The special RISC-V *ECALL* instruction is then requesting to execute the prepared system call from the operating system and the instruction with #605 can then already access the result value stored in register file at index 10.

```
// ...
#598 20418 – ADDI: rd: 12 (0xc), rs1: 0 (0x0), imm: 0 (0x0)
#599 2041c – ADDI: rd: 11 (0xb), rs1: 2 (0x2), imm: 0 (0x0)
#600 20420 – ADDI: rd: 13 (0xd), rs1: 0 (0x0), imm: 0 (0x0)
#601 20424 – ADDI: rd: 14 (0xe), rs1: 0 (0x0), imm: 0 (0x0)
#602 20428 – ADDI: rd: 15 (0xf), rs1: 0 (0x0), imm: 0 (0x0)
#603 2042c – ADDI: rd: –15 (0x11), rs1: 0 (0x0), imm: 80 (0x50)
#604 20430 – ECALL
#605 20434 – ADDI: rd: 8 (0x8), rs1: 10 (0xa), imm: 0 (0x0)
```

Listing 3.13: Example Instruction Trace for a SysCall on RISC-V

3.5.3 Simulator Generator Implementation

The simulator framework creates a separate component class to handle the syscall delegation between guest and host system, which provides a generic interface do execute a syscall by calling `void emulate()`. The processor, which has been instantiated by the simulator main class, is only using the syscall component and therefore doesn't contain any syscall specific logic. This degree of separation of concerns was one of the design goals for system calls due to their system specific nature.

The syscall component is actually being rendered by the `LinuxSysCallCppGenerator` and `LinuxSysCallHppGenerator` generators, which provide the general structure of the class, but filling in the implementation details by relying on the `SysCallDescription`. The interface for such a description is shown in Listing 3.14, with the additional internal data structure, that defines the supported syscalls. This interface provides three basic functionalities that generators need for creating their implementation. First a list of C++ includes that are being used in the delegation mechanism. Typically it is more convenient to call available C++ wrapper methods from the standard library than to directly interact with the system calls from the current host system. An example for this would be the `int fcntl(int fd, int cmd, ... /* arg */)` method, which is provided by the *fcntl.h*. The second method of the interface will provide the list of supported syscalls. And the last method of the interface is a fallback, which provides one generic syscall that just passes the given parameters to the host system. This mechanism can handle delegation for very simple cases, but is probably not directly usable for most cases. The reason for that is the necessary conversion between parameters, because file descriptors and memory locations have to be converted between the simulator memory and the simulator process, which is running on the host system.

```
interface SysCallDescriptions {
        def List<String> getNecessaryImports();
        def List<SysCallDescription> getSysCalls();
```


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

```
        def SysCallDescription getGenericSysCall();
}

@Data
class SysCallDescription {
        int number
        String name
        List<Integer> parameters
        String callSyntax
        String code
}
```

Listing 3.14: SystemCallDescription Interace and Internal Data Structure

3.5.4 Defining a SysCall Delegation for Write

But to show a concrete example of a syscall implementation for the simulator generator, will probably make this point much clearer. For this we have to look into the class LinuxAsmGenericSysCalls, which provides an implementation for a SysCallDescriptions. Listing 3.15 shows the definition for the system for writing to a file descriptor, which is also used for printing a value to the console. The syscall in this example is registered with number 64 and the text 'ssize_t write(int fd, const void *buf, size_t count); constitutes the signature of the C++ wrapper method from the *man pages* that should be called. The signature is used to count the number of used arguments and provide these values as local variables starting from p1. Basically the simple number of parameters is sufficient to create this syscall description, but writing out the signature is nonetheless a good practice, because it also gives additional information about the argument types and names which helps to document the functionality and improves readability. The remaining code in the template is the rendered C++ code that executes the syscall on the host system with the given parameters. C++ pointers and file descriptors need to be translated by provided helper methods, as can be seen in the example. A pointer from the simulator memory is hereby just translated to the address in the host system. But in the case of file descriptor objects, the simulated application is only allowed to see the previously opened descriptors and not those the simulator uses for itself.

```
add(64, 'ssize_t write(int fd, const void *buf, size_t count);',
  '''
  const int fd = translateFd(p1);
  if (fd < 0) {
    return fd;
  }

  size_t addr;
  int rcMemoryRead = translateMemoryAddress(p2, addr);
  if (!rcMemoryRead) {
    return -1;
  }
  return write(fd, (void*) addr, p3);
```


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

```
  """
)
```

Listing 3.15: SystemCall Example for write

And there is also a practical point of view to be considered in the implementation of delegating system calls. A simulated application will probably try to close its resources, like `stdout` before exiting. But the simulator currently uses the same `stdout` for practical reasons and would then also not be able to print any message after this point. The current syscall delegation will therefore not close any file descriptors, for which `fd <= 2`. But the overall syscall framework should be quite extensible if the need ever arises for a future version.

3.5.5 Defining a SysCall Delegation for Exit

One last example for a special system call is shown in Listing 3.16. This system exit call can't be delegated to the host, because it would shut down the simulator process itself. But it also shows that the processor component is accessible from within the delegation mechanism, if necessary.

```
add(93, 'void exit(int status);',
   """
  processor.shutdown(p1);
  return p1;
   """
)
```

Listing 3.16: SystemCall Example for exit

3.5.6 Limitation of the Current Implementation

Currently only a subset of the most important syscalls is supported, but this limited set can quite easily be extended. And it is also noteworthy that the syscall numbers from a RISC-V Linux guest application will currently match the ones from the Linux host system the simualtor was tested against. Therefore, the syscall numbers have not been translated. But if this is not the case in a future application scenario, then the syscall numbers have to be translated between guest and host system, which should be done in two steps. First, translate the guest system number to an internal representation and from this internal representation in a second step to the actual syscall number from the host system.

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

CHAPTER 4

Instruction Set Simulator Generator

4.1 Structure of the generated Simulator

The schematic structure of the generated ISS simulator is depicted in Figure 4.1. This class diagram shows the simulator entry point in the Main class, which parses the program arguments and instantiates a processor object. A processor class provides an implementation for the IProcessor interface and contains various components, that are being generated according to the VADL description from which the ISS has been generated. But not all supported processor components have to be present in a processor implementation, which depends on which components are being described in the ISA. The RISC-V base description for example, doesn't doesn't specify register flags. The actual interface of the components is different for each component type, which will be described in the following section in more detail. The processor implementation also relies on a separate SysCall component for issuing system calls, which are being delegated to the host system, which runs the simulator.

4.2 ISS Components

Each supported component can be implemented as static resource or dynamically generated from a VADL description, by a previously described *generator class*.

While register, memory and flag components are all abstractions to access data from within a processor implementation, they can quite differ in their requirements and therefore also in their interfaces. To preserve the maximum flexibility for development, these components do not share a common interface. And because there is currently only one implementation for each of these component types at the time of this writing, there is

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

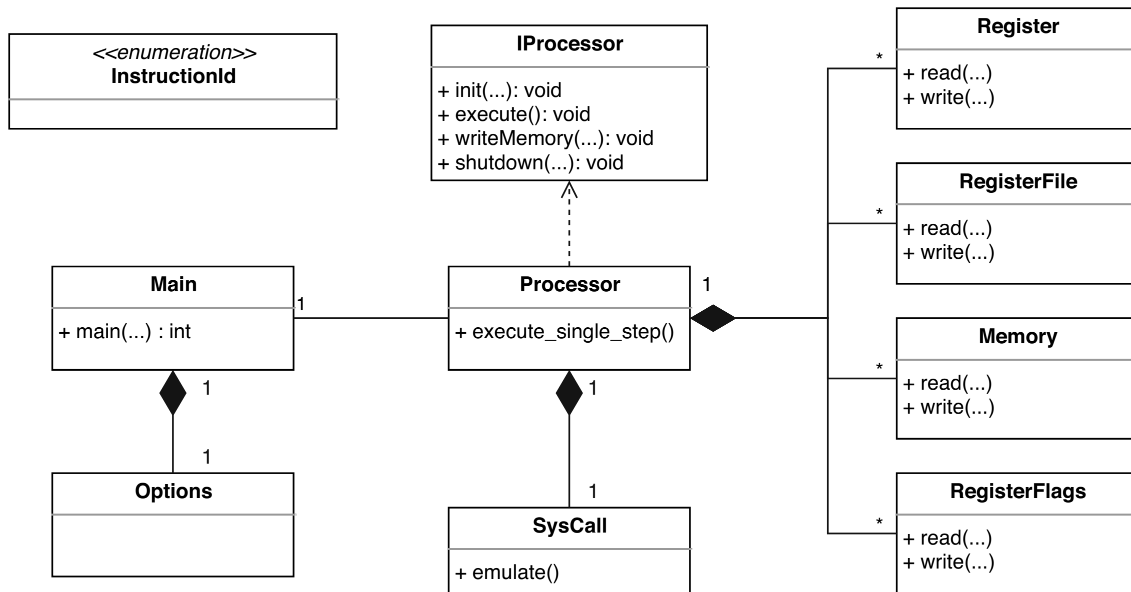

Figure 4.1: ISS Simulator Class Diagram

no explicit interface or abstract class defined for them either, to keep the implementation as simple as possible. But if multiple different memory components or register definitions exists at some point, then a common interface for each component type should certainly be specified.

Components are defined in separate class files and the processor implementation also has to know how to use them. This includes to generate the C++ includes for the used components, their instantiation by calling the specific constructor with mandatory parameters, and how the component object is actually being accessed.

An example of the generated component definition code on basis of the VADL description from Listing 4.1 is given in Listing 4.2. This example VADL definition for the 32 bit base ISA of RISC-V, defines a memory component, a program counter and a register file by the name X with 2^5 associated integer values. The memory component in this example is byte-addressable and has an address space of 32 bits. All register values in this case have a specified length of 32 bits. The annotation $\mathrm{X}(0) = 0$ specifies the first register value of X as so called zero register, which ignores assignments and will always return the integer value zero.

The emitted code for accessing these components for an example VADL instruction definition from Listing 4.3 is given in Listing 4.9. While a register read like X.read(rs1) only has the index as argument and can be written as an expression, this is handled differently for accessing a memory component. Reading a memory value is always emitted as a statement, which stores the return value into a temporary variable of the specified

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

length. While this small implementation detail requires that the abstract syntax tree (AST), which represents the code of the instruction has to be slightly altered, it allows to encapsulates the memory access logic completely into the memory component. Even if currently the memory is always assumed to be encoded as little-endian, it should help to add support for endianess in a future version of the simulator generator.

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

program counter PC : bit<32>

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

Listing 4.1: ISA Component Definition in VADL

```
class Processor : public IProcessor {
  Memory MEM;
  RegisterFile X;
  Register<uint32_t> PC;
  SysCall sysCall;

  // Constructor
  Processor(...) :
    IProcessor(),
    MEM("MEM", /*size*/ 0xffffffffULL),
    X("X"),
    PC("PC"),
    sysCall(...) {}

  //...
}
```

Listing 4.2: Example of Defining Components for a RISC-V description

```
// Load Halfword
instruction LH : I_TYPE = {
  let addr = unsigned( X(rs1) + ImmediateI ) in {
    X(rd) := sext(MEM(addr)[15..0], 32)
  }
}
```

Listing 4.3: RISC-V Example Instruction Definition in VADL

```
inline void Processor::exec_LH(const uint32_t instr, const uint32_t addr) {
  const uint8_t rs1 = (instr >> 15) & 0x1f;
  const uint8_t rd = (instr >> 7) & 0x1f;
  const uint16_t imm = (instr >> 20);
```


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

```
  const uint32_t ImmediateI = ((((((uint32_t) imm) &0xfff) ^0x800) - 0x800)
      & (0xfffff << 12)) | imm;
  const uint32_t addr2 = (int32_t) X.read(rs1) + (int32_t) ImmediateI;

  uint32_t t4 = 0;
  MEM.read4Byte(addr2, t4);
  X.write(rd, t4);
  PC.write(addr + 4);
}
```

Listing 4.4: Emited Code for the RISC-V Instruction LH (Load Halfword)

The logic to access and initialize components is currently hard wired in the used generator classes and has to be adapted accordingly, when the interface of a component is being changed. Breaking a component interface in the generated code can typically by detected quite easily during the compilation of the generated simulator. Such errors are also easy to fix, because the code which emits the component access, is encapsulated in only very few helper classes like SimulatorCodeGenerationUtil.

While this way of component implementation and usage is certainly on a very low abstraction level, which also has some disadvantages, it allows the greatest flexibility to adapt the code and also allows to use various C++ features, like templates, without restrictions. An example of this can be seen in the implementation of the register component from a static resource in Listing 4.5, where the template parameter is being used to support different register data types.

```
template<typename T>
class Register {
    private:
        const std::string name;
        T data;
    public:
        Register(const std::string& name) : name(name), data(0) {}

        const T read() const {
            return data;
        }

        bool write(const T value) {
            data = value;
            return true;
        }
};
```

Listing 4.5: Register Component defined as Static Resource

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

4.3 Simulation Step

The emitted ISS mimics the effects of a single instruction in each step. The implementation is realised as a switch-based interpreter, which executes a main loop that repeats a simple fetch-decode-execute pattern, until the processor is shutdown. The corresponding code can be seen in Listing 4.6, which relies on the decoding logic described in section 3.4.

```
void Processor::execute() {
        while (running) {
                execute_single_step();
        }
}

void Processor::execute_single_step() {
    instruction_counter++;

    // fetch next address
    const uint32_t addr = PC.read();
    uint32_t instr;
    MEM.read4Byte(addr, instr);

    // decode instruction
    InstructionId decodedInstruction = UNKNOWN_INSTRUCTION;
    decodedInstruction = decodeInstruction32(instr);

    // execute instruction
    if (decodedInstruction != UNKNOWN_INSTRUCTION) {
      switch(decodedInstruction) {
        case ADD:
          exec_ADD(instr, addr);
          return;

        // ...

        case LH:
          exec_LH(instr, addr);
          return;

        // ...
      }
      return;
    }
    cerr << "unknown instruction: " << std::bitset<32>(instr) << endl;
    shutdown(-1);
}
```

Listing 4.6: ISS Main Loop

The behaviour to simulate the effects of an instruction is hereby emitted in a separate method, which can typically be heavily optimized during compilation of the simulator by a C++ compiler. One example of the emitted code for a VADL instruction definition of a

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

RISC-V load halfword instruction from Listing 4.3 has already been shown in Listing 4.9 and will now be discussed in more detail. The instruction method receives the program counter and instruction bit sequence as method arguments. The method body consists of three blocks. First the instruction format fields are being decoded from the instruction bit sequence. Then an optional immediate value is being computed and finally the actual behaviour of the instruction is being emitted.

For this example of a LH instruction, the associated instruction format is the *I-Type*, which has been previously shown in Listing 3.8. Only the instruction format fields that are actually being used in the VADL instruction behaviour description, will be decoded. This is a minor optimization for instructions like the RISC-V *ECALL* and *EBREAK*, which are associated to the same *I-Type* as the load-halfword instruction, but do not use any of the instruction format fields. The emitted code to retrieve the bit values for the format fields is created by the same utility methods already described in Listing 3.8.

The second block can contain code to assign temporary values, based on the previously decoded format fields. This mechanism is typically used to apply sign extension to decoded format values. An example VADL definition to specify a temporary value is shown in Listing 4.7. Temporary values can be used by their name in the VADL instruction definition, along with other values from the instruction format. As before, a temporary value is only being decoded if it is actually being used in the instruction definition. This allows to define multiple temporary values for a specific instruction format, which are only being used in a subset of the instructions, without the need to worry about a potential performance impact. The example in Listing 4.7 uses the `sext (...)` built-in to perform the sign extension on the `imm` value. Bit masks that are used by the `sext (...)` built-in at runtime are pre calculated during the generation of the simulator. Immediate values can also be created by concatenating multiple bit sequences, which is shown in the Listing 4.8. The emitted C++ code is directly following this specification by creating two separate bit sequences that are then being combined into a single value by a *logical OR* operator. The first bit sequence in this example provides the upper 31 bits, while the least significant bit is set to binary zero.

The third block contains the code to simulate the described behaviour of the instruction. The first instruction is the address calculation, which adds the value of the register file at index `rs1` with the the sign extended immediate value and stores it into a local variable `addr2`. The original name of the variable has automatically been changed during code emission to avoid a name clash with the method argument of the same name, which has been described in section 3.1.3. The memory access expression has been rewritten to a statement, reading two bytes into a temporary variable `t0`, which is sign extended by the `sext (...)` built in, before the value is written to the register file. While sign extension can be expressed by using temporary variables or using this built-in keyword, it depends on the use case of which of these options is more fitting to use. And having various options to define such aspects in the processor description, is one of the strengths of the VADL language.

The last instruction `PC.write(addr + 4)` of this block is added by the simulator generator,

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

to increase the program counter by the format width of the instruction. This code is only added if the program counter is not being altered by the instruction behaviour. Otherwise an additonally added comparison between the current and previous program counter value will ensure, that this value is only being increased if the branch has not been taken.

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

Listing 4.7: Sign-Extension for the I-Type Immediate Value

```
immediate ImmediateB : B_TYPE -> Word = [ sext( imm, 31 ), 0b0 ]
```

Listing 4.8: Sign-Extension for the B-Type Immediate Value

```
inline void Processor::exec_LH(const uint32_t instr, const uint32_t addr) {
  const uint8_t rs1 = (instr >> 15) & 0x1f;
  const uint8_t rd = (instr >> 7) & 0x1f;
  const uint16_t imm = (instr >> 20);

  const uint32_t ImmediateI = ((((((uint32_t) imm) & 0xfff) ^ 0x800) - 0x800)
      & (0xfffff << 12)) | imm;

  const uint32_t addr2 = (int32_t) X.read(rs1) + (int32_t) ImmediateI;
  uint16_t t0 = 0;
  MEM.read2Byte(addr2, t0);
  X.write(rd, sext16To32(t0));
  PC.write(addr + 4);
}
```

Listing 4.9: Emitted Code for the RISC-V Instruction LH (Load Halfword)

The VADL description for a RISC-V instruction to load a word, instead of a previously shown half-word, looks quite similar. The only difference is that the value assigned to the register is actually 32 bits wide (`X(rd) := sext(MEM(addr)[31..0], 32)`). The emitted code is therefore also quite similar, with the only differences being shown in Listing 4.10. The memory access is now reading 32 bit into a temporary variable `t0` and the previously shown sign extension built in is not being applied any more, because it has been detected as obsolete.

```
...
uint32_t t0 = 0;
MEM.read4Byte(addr2, t0);
X.write(rd, t0);
PC.write(addr + 4);
```

Listing 4.10: Emitted Code for the RISC-V Instruction LW (Load Word)

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

4.4 Interactive Mode

Debugging in the context of simulated programs is quite a challenge. This is primarily due to the high number of executed instructions and the resulting large amount of log outputs, but also a consequence of the low level nature of the problem domain. Minor errors, like for example in the definition of a format field, can have adverse effects many hundreds of simulated instructions later. Each error can also manifest itself in various ways, depending on the simulated program, starting at wrong computed values leading to an endless loop or even having no relevant effect at all. The implemented interactive mode of the simulator is intended as main debugging tool for all of these situations. But the actual implementation also depends on the simulation model and therefore differs quite a lot between the ISS and CAS. This section starts by describing the interactive mode in the context of the ISS, while the later section in the CAS chapter will focus primarily on these differences.

4.4.1 User Interface

The interactive mode is being started by providing the simulator with the *-i* parameter at start up. This parameter ensures that the main entry class will instantiate the interactive processor implementation, which has been created with a modified main loop, which asks the user after each logical step for the next action to perform. The default action can be executed by pressing the *enter key* and will simulate the execution of a single instruction. An example output for the previously used add.c program from Listing 3.5 is shown in Listing 4.2.

The output is split into multiple sections. The first one, on top of the screen, shows the most recently executed instructions. Each line contains the instruction number, followed by the fetch address of the instruction and the instruction values. The representation of the instruction values is hereby using the assembly definition, extended by the name of the format fields. While this output variant differs from the printable instruction trace, it is more compact while preserving the most valuable information. The primary intention of this change was to improve the readability, which is also reinforced by the use of colours.

After a line break, the last executed instruction is being printed. In the given example this is the load word instruction with the instruction number 8, and its VADL behaviour code is also being printed. This feature was a big improvement, while creating the initial RISC-V VADL specifications, because it allowed to easily spot mistakes without looking up the actual definition in quite a few cases.

The next block in the output shows the data that have been read or altered, during the instruction execution. In the case of the load word example, this output will contain the sign extended temporary value for ImmediateI. Output values are hereby being printed as unsigned integer and hex value, which helps the readability, but also helps to spot errors, like values that have not been sign extended. For altered values like the assignment to

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

the register file at index 14, the previous zero value is also being printed. Each access to a resource value is being printed in the order it has been executed by the simulator. While this will typically show redundant information, it can help to detect data hazards from wrong execution sequences in the simulator implementation or the VADL definition.

The last section on the screen shows the next instruction, which is going to be executed in a simulation step. This feature has been added for usability reasons, after testing the first version of the interactive mode. Even the small lookup of one instruction gives valuable insights into which resources are currently being altered and used in the next instruction. This also avoids many resource lookups during a debugging session and helps to navigate through the instruction trace without overstepping.

```
#1      10054:          ADDI rd:2,rs1:2,imm:4064
#2      10058:          SW rs2:8,imm:28(rs1:2)
#3      1005c:          ADDI rd:8,rs1:2,imm:32
#4      10060:          ADDI rd:15,rs1:0,imm:3
#5      10064:          SW rs2:15,imm:4076(rs1:8)
#6      10068:          ADDI rd:15,rs1:0,imm:4
#7      1006c:          SW rs2:15,imm:4072(rs1:8)

#8      10070:          LW rd:14,imm:4076(rs1:8)

let addr = X[rs1] + ImmediateI {
        let t14 = MEM(addr) {
                X[rd] := sext(t14)
        }
}

=> ImmediateI := 4294967276 (0xffffffec)

=> X[8]: 4294967248 (0xffffffd0)
=> MEM[0xffffffbc]: 3 (4B, 0x3)
X[14] := 3 (prev:0, 0x0)
PC := 65652 (prev: 65648, 0x10070)


=>      #9      10074:          LW rd:15,imm:4072(rs1:8)
>
```

Figure 4.2: ISS Interactive Mode

4.4.2 Commands

The interactive mode of the ISS supports basic commands for proceeding through the simulation and displaying various information, which will be briefly described in the

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

following. The available options can also being looked up by typing in help.

For performing a single simulation step, there exists the already mentioned default option by pressing the *enter key*. Multiple simulation steps can be issued at once by the step command, followed by the number of steps. Similarly, there is also a command to, which execute simulation steps until the given instruction counter has been reached. For debugging it is also often useful to run through the simulation until an instruction has been fetched from a given memory location. This can be done by the until command, followed by the memory address.

While read and altered information is being printed after each executed instruction, it is also possible to query the state of components. This is done for a register or register file by simply issuing the name of the component from the VADL description. For the previously given example in Listing 4.1 this would be PC and X. To query the value of a memory component it is necessary to issue the component name of the VADL description, followed by the number of bytes (1,2,4,8) and the memory address in hexadecimal. Currently the state of components can only be displayed in the interactive processor and not altered, but it would not be difficult to add this feature in a future simulator version.

4.5 Termination

The ISS has to detect when a simulated program has been finished to stop its execution. If an executed program will issue a system call to exit, as described in section 3.5.5, then this is easy to detect and handle the termination process. In these cases, the simulator just omits the system call and sends an internal shutdown signal instead.

But if a program, like the one from Listing 3.5, has been compiled without operating system support, then it will not contain any such system calls. The program code for this add.c example is shown in Listing 3.6. The last instruction from this RISC-V example is a jump and link instruction ($JALR$) to the return address, which had been initialized with zero when starting the simulator. The memory address zero doesn't contain a valid instruction and this will, as a consequence also stop the simulator process with an unknown instruction exception. This is not ideal, but the possible alternative implementation options are also quite limited. The simulator could be extended with an additional argument to limit the number of instructions or specify an explicit stop address. Another variant is described by [Rah], which introduces a global variable with a fixed name. If the program wants to stop the simulator, it simply has to alter the global variable value. All of these options depend on or alter the simulated program and are therefore currently not supported.

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

CHAPTER 5

Cycle-Accurate Simulator Generator

The CAS generator is also based on the earlier introduced code generator framework and employs a similar structure as the ISS generator. Implementation differences result mostly from the more detailed simulation model and additional components described by the MiA.

Section 5.1 will introduce the general structure of the simulator generator. The remaining chapter will then describe various implementation aspects on basis of a 6-stage in-order pipeline architecture, which is being described in section 5.2. Section 5.3 will then introduce CAS components to manage processor state, instruction retirement and to represent the pipeline register and pipeline stages. Section 5.4 will describe how behaviour code can be added to pipeline stages and the following section 5.5 describes how instructions are being partitioned over multiple pipeline stages.

VADL descriptions are again shown alongside the presented examples.

A primary goal of the VADL language design was to avoid redundant definitions whenever possible. Chapter 5.5 shows how this design goal has been implemented, by partitioning the defined instruction behaviour onto pipeline stages. Section 5.5 will go into more detail how pipeline hazards are being handled in the generated simulator. And the last section 5.6 will briefly introduce the interactive mode of the CAS.

5.1 Structure of the generated Simulator

The class diagram of the generated CAS is at first glance quite similar to the ISS, as can be seen in Figure 5.1. The main differences are the additional components to represent pipeline register and pipeline stages, and two new components to manage the processor

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

state and the instruction retirement process. While the former two component types can be expected to be found in a pipeline model, the latter two have been added to reduce cyclic references between the processor implementation and its components. And this consolidation of processor state management and instruction retirement code also reduces the implementation complexity, which was otherwise aggregated in a few classes.

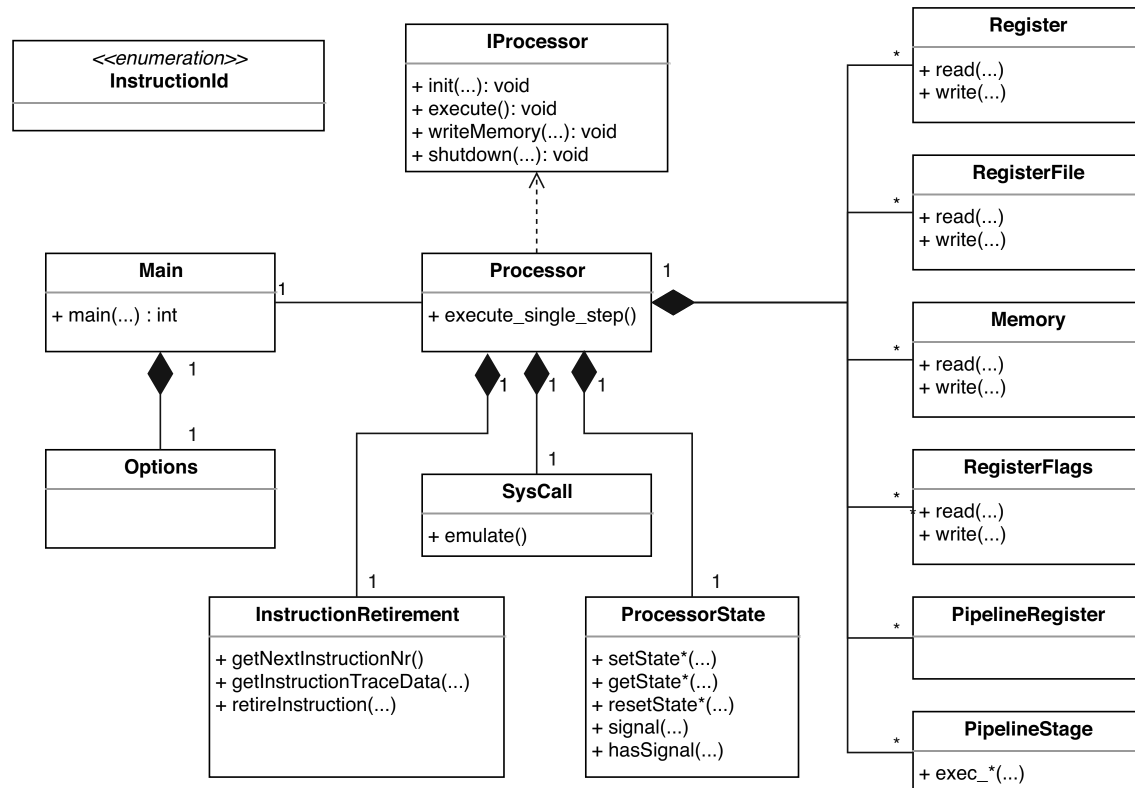

Figure 5.1: CAS Simulator Class Diagram

5.2 Example Pipeline Model with 6 Stages

While various pipeline architectures can be expressed in VADL, it is much easier to explain the underlying implementation and code generator by a concrete example. For this purpose a simple in-order RISC-V pipeline architecture has been chosen, which is essentially described in various references like in chapter 4 of [PH17] or the appendix C of [HP17]. While the cited references mostly describe the pipeline architecture on basis of a 5-stage pipeline, the remaining example pipeline is slightly altered to a 6-stage pipeline model. The reason behind this decision is the more clear distinction between the generation of the program counter and the instruction fetch behaviour and does not alter any other aspects.

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

A schematic model of the used example pipeline architecture is depicted in Figure 5.2. This in-order pipeline represents a linear data path from the leftmost pipeline stage called *PC-Gen* to the rightmost one named *WB*. A pipeline register is hereby a data storage, which separates two pipeline stages from each other. When a pipeline stage writes values into an outgoing pipeline register, these changes are only becoming visible in the next clock cycle. This is an essential mechanism to provide correct results in a pipelined computation model by providing stable data inputs during the computations of a clock cycle. The behaviour logic in the emitted simulator is represented by the pipeline stage components and each stage has a defined responsibility. The *PC-Gen* will provide the program counter value from which the following *Instruction Fetch Stage (IF)* can load the program instruction to execute. The *Instruction Decode Stage (ID)* stage will then identify the instruction type and decode its operand values. The *Execution Stage (EX)* will afterwards perform arithmetic or branch computations and the final two stages are then responsible to write these results back into the memory (*Memory Access Stage (MA)*) and the register files (*Write Back (WB)*).

Access to components is typically restricted in a pipelined model to specific pipeline stages. Memory access for example, is only allowed in this scenario to fetch instructions in the *ID stage* and to access data values in the *MA stage*. Resource conflicts can occur when a component is being used from multiple pipeline stages and have to be avoided by implicit synchronisation mechanisms or by skilful pipeline design.

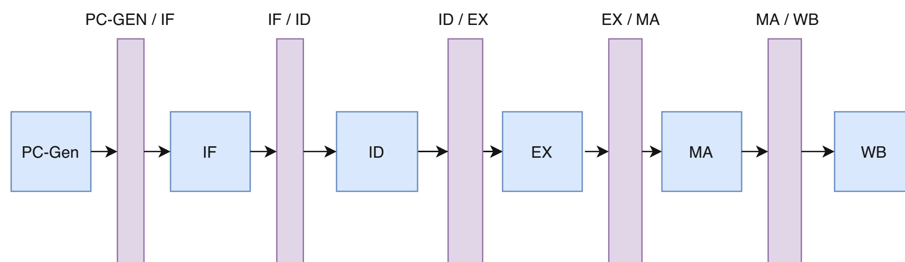

Figure 5.2: High Level View of the Pipeline Model

VADL Pipeline Description

To describe this 6-stage pipeline in VADL, it is necessary to define the pipeline stages first, which are being shown in Listing 5.1. Each pipeline stage is associated with a unique name and can also contain nested components, like the program counter in the *PC_GEN stage*.

While the pipeline stage definition in the MiA introduces new components, this is not the case for the PC register which already has been declared in the VADL ISA definition by the same name. When a component is also being specified in the MiA, this can be seen as an instantiation mechanism to provide additional implementation details, which are not present in the more abstract ISA representation. Assume for an instance a memory

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

component that is only modelled in the ISA as a generic data storage, while the MiA also requires additional implementation details like memory latency or the number of access ports.

Instantiating a rather basic component, like the register value in this case, will not provide additional implementation details for the component itself. But is a convenient way to define access restrictions, because nested components will per default only be accessible in the enclosing pipeline stage.

```
pipeline stage PC_GEN {
  register PC
}
pipeline stage IF
pipeline stage ID
pipeline stage EX
pipeline stage MA
pipeline stage WB
```

Listing 5.1: Pipeline Stage Definition in VADL

After the pipeline stages have been defined, the pipeline structure can be specified in a second step. This is done by listing each data path as a linear sequence of pipeline stages, which is shown in Listing 5.2. Each data path also has an associated name, which is mainly intended for documentation and debugging purposes.

While the used in-order pipeline example only consists of a single data path, it should be noted that the specification for superscalar pipeline architectures is not final yet. While it should be quite possible to extend the current approach to also define superscalar and other complex pipeline models, it could also be quite different in the future VADL versions.

```
pipeline structure:
        P1: PC_GEN -> IF -> ID -> EX -> MA -> WB
```

Listing 5.2: Pipeline Description in VADL

5.3 CAS components

The components used by the CAS are generated by the same code generation techniques as described for the ISS. But the CAS relies on its own generator classes and uses a separate template folder to specify static resources. This makes it possible to adapt components in various ways and alter their implementation significantly to the earlier described ISS counterparts.

A pipelined processor can process multiple overlapping instructions in its pipeline stages concurrently at the same time. It has to avoid all data hazards, which would not have occurred when executing the same instructions in sequence.

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

The design of the involved hardware components ensures that value changes do not result in any data hazards, which would not have occurred when executing the same instructions sequentially. The current simulator implementation does ensure that the order in which pipeline stages are being processed doesn't change the computation outcome. This is a consideration from a software development standpoint to delay a design decision as far as possible to preserve all options. Computing the pipeline stages in reverse order can make the buffering of the pipeline registers obsolete and therefore can probably achieve a better runtime performance. But for the current version it still holds that any operation on a register file component will not become visible until the next processor cycle. This feature is implemented by buffering the written values and only assign them if a commit is being issued by the processor main loop.

5.3.1 Processor State Component and Pipeline Control Instructions

A processor state component manages, as the name implies, the state of a processor implementation, but also the signal handling between components. This component allows to issue general signals like SYSCALL or SHUTDOWN for the processor, which are then being processed after each simulated clock cycle. But it also manages the state of the pipeline stage components, which can be either ACTIVE, STALL or FLUSH. While a processor stage will initially be set to ACTIVE at each iteration of the main simulator loop, it can be changed by VADL *stall* and *flush* build-ins, like can be seen in the listed pipeline stage behaviour code in the Listings 5.7 and 5.9.

A stalled pipeline stage will pause its current computation to resume it at the next processor clock cycle, with the same pipeline register values as its input. Stalled pipeline stages will pass a NOP instruction to successor pipeline stages, which is also often called a pipeline bubble. A flushed pipeline stage will behave quite similar to the described stall behaviour and will also emit a pipeline bubble. But the difference is that the computation is not being resumed on the next clock cycle, but instead abandoned.

Stalling can occur for various reasons in a pipeline, like for a example that the value of requested memory address is not yet available and will typically also stall its predecessor pipeline stages in a ripple effect. A pipeline flush on the other hand, mainly occurs in the case of a branch misprediction. This means that the pipeline is at least partially filled with obsolete instructions, which are being purged from the pipeline. While a stalled pipeline stage will typically also effect the state of its predecessors, this is not the case for flushed pipeline stages, which can accept a new instruction on the next clock cycle.

The described states for pipeline stages have an order of precedence, which means that a received flush signal will overwrite a stall signal, but not the other way around.

5.3.2 Pipeline Registers

Pipeline registers are being used to buffer values between pipeline stages. These structures are represented in the simulator as data container objects, comprised of a list of primitive data values quite similar to a C++ struct. The actual data values for each pipeline

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

register depend on the information, which has to be passed through the various pipeline stages to simulate the behaviour of an instruction. While pipeline register components can be defined explicitly in the VADL description, these structures will also be created automatically by the simulator generator if no such definition exists.

One reason to specify a pipeline register in VADL is to alter its name, which would otherwise be generated per default from the adjacent pipeline stage names, like for example REG_ID_EX. Data transfer values which are being used to pass information through the pipeline to implement the instruction behaviour, are automatically derived from the VADL description and added to the necessary pipeline register.

But pipeline stages can be extended by additional global behaviour code to implement for example branching-, forwarding- or hazard logic. If such features are being implemented over multiple pipeline stages, then shared data values have to be transferred between the stages by using pipeline register values. And this is the primary reason to explicitly specify pipeline register components, because it allows to add additional data values, which then can be accessed in the VADL behaviour code. An example for an explicit VADL definition of a pipeline register can be seen in Listing 5.3. This shows how to specify the name of the pipeline register between the EX and MA stage to REG_EX_MA. The branch_taken flag and branch_addr are also explicitly defined and added to the emitted pipeline register attributes.

```
pipeline register REG_EX_MA between EX and MA {
  // set to 1 if a branch has been taken
  branch_taken bit<1>

  // branch target address
  branch_addr bit<32>
}
```

Listing 5.3: Adding Additional Data Values to a Pipeline Register

The emitted code for the pipeline register between the *ID* and *EX* stage of the RISC-V description is shown in Listing 5.4. This generated pipeline register contains the instructionId, which identifies the decoded instruction, but also various data values. These values include the register source index (rs1, rs2), the destination index (rd), the program counter (PC), from which the decoded instruction has been fetched and an decoded immediate value with the name immediate. Along with the read register values X_rs1_ and X_rs2_ for the source register index rs1 and rs2. While instructions will typically only use a subset of these values, the pipeline register has to contain the superset of all data values used by any of the supported instruction.

```
class PipelineRegister_REG_ID_EX {
public:
        InstructionId instructionId;
        uint64_t instructionNr;
        uint8_t rs2;
```


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

```
        uint8_t rs1;
        uint8_t rd;
        uint32_t PC;
        uint32_t immediate;
        uint32_t X_rs1_;
        uint32_t X_rs2_;
...
}
```

Listing 5.4: Pipeline Register Example

Data Transfer Logic

Changes to values of a pipeline register should only be visible after the current clock cycle to avoid data hazards and race conditions. This is quite similar to the previously described commit mechanism for register files, but actually also has to take the state of the adjacent pipeline stages into consideration. The processor implementation of the simulator instantiates each pipeline register type like the above listed PipelineRegister_REG_ID_EX twice, to represent the values of the current and the next upcoming clock cycle. This pipeline registers usage is being depicted in Figure 5.3 and shows the current pipeline register values in purple colour, which serve as input values to the pipeline stages. Computation results in a pipeline stage can safely be written to the outgoing pipeline registers, which are being shown in green colour in the example figure. This separation between input and output values ensures that parallel computations in adjacent pipeline stages will not cause any race conditions and is a key aspect that pipeline stage computations do not depend on a specific order.

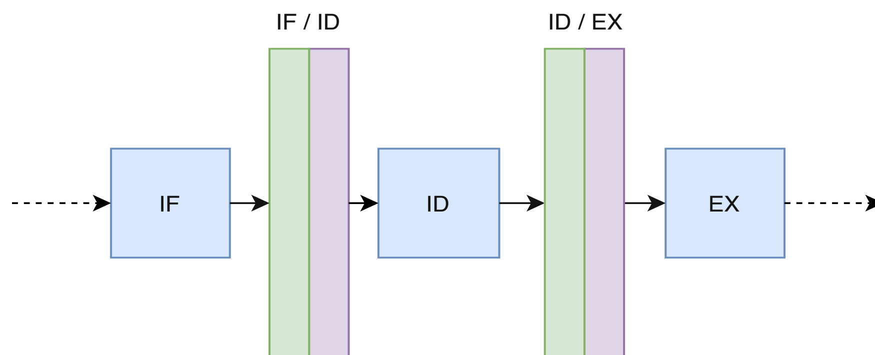

Figure 5.3: Pipeline Register used to buffer values between Pipeline Stages

The generated pipeline register components provide two methods to manage their data values. The first is the resetValues method, which will reset a component to a neutral state, which corresponds to a pipeline bubble. The second one called transferValues (...) will copy the data values from one pipeline register to another, which is applied to switch incoming and outgoing pipeline register values, when no stall or flush has occurred. These methods

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

are being used by the processor implementation to manage the state of pipeline registers, depending on the state of the adjacent pipeline stages. A code example for one pipeline register is shown in Listing 5.5.

```
if (processorState.getStateEX() != STALL) {
  if (processorState.getStateID() == ACTIVE) {
    outREG_ID_EX.transferValues(inREG_ID_EX);
  } else {
    inREG_ID_EX.resetValues();
   }
}
```

Listing 5.5: Forward, Stall and Flush Logic for the REG_ID_EX' Pipeline Register

Instruction Type Information

The type of a processed instruction will be detected during the instruction decoding process. In a pipelined model this is typically done in separate pipeline stages, after the instruction fetch has been performed. The code generator therefore also has to take the order of the pipeline stages and phases into account. The instructionId, which identifies the decoded instruction type is accessible as local variable during the instruction decoding stage and afterwards through incoming pipeline register values. Early stages like the PC_GEN and IF in the described 6-stage pipeline have no information on the processed instruction type yet.

Additionally, any information that is only available before the instruction decoding process has to be passed through the pipeline, until the instruction decoding can actually decide if the value is required or not. For RISC-V, all jump instructions are relative to the fetch address, which therefore has to be preserved through the IF stage in all cases. After the instruction decoding stage, only the jump instructions will pass the fetch address along, while it will be ignored for all other instructions.

5.3.3 Pipeline Stages

While the ISS can depend on a single execute method to simulate each instruction, this is no longer the case for the CAS, where the same instruction behaviour has to be executed sequentially over the course of multiple pipeline stages. This requires that the instruction behaviour code has to be transformed into transferable pieces and partitioned over multiple pipeline stages. An algorithmic approach for this task has been implemented, which requires that the component access permissions are being specified, along side with an assignment of all instruction statements to a pipeline stage. While this approach is described in detail in section 5.5, it can only resolve situations where data values are passed forward through the pipeline, but which is also the majority of the cases. But the current algorithm fails, when information has to be passed back to a predecessor pipeline stage, which typically occurs for branching- and data hazard logic. But these implementation details can be complemented by attaching additional behaviour code to

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

pipeline stages. This enables a VADL user to add code that can't be automatically derived and even express advanced features for the MiA. But the consequence is that the emitted code for the pipeline stages will be stitched together from automatically partitioned and explicitly defined code fragments. This requires clear rules that define the sequence in which code fragments are being emitted, and will be described in the remaining section. The following section 5.4 will then describe how to use specify additional behaviour code with practical code examples.

Clock Signal

First, there is another noteworthy type of MiA component in this context, which defines a clock signal for the pipeline model. The simulator generator will not directly emit a component from this VADL definition, but instead will use this to split the simulated clock cycle into sequential phases. An example VADL definition can be seen in Listing 5.6, which defines a begin and end phase. Additional behaviour code can now be assigned to pipeline stages at one of these specific phases, which provides a simple mechanism to specify the order in which behaviour code has to be executed during a clock cycle. This mechanism can be effectively used to perform the register file write back at the beginning of a clock phase, right before the instruction decode is being performed to avoid data hazards.

```
clock signal with BEGIN=1, END=0
```

Listing 5.6: Clock Signal description in VADL

5.3.4 Code Layout

The Figure 5.4 shows the code layout for a pipeline stage. A separate execute method will hereby be emitted for each defined clock phase, containing the global and statement specific code fragments.

The global code fragment, depicted in green colour, is always emitted first. This block consists of the MiA behaviour code fragments that are being assigned to a specific clock phase and global data transfer code. The latter is used to preserve data values in the pipeline registers, before the instruction decoding determined the instruction type.

The second block is depicted in yellow and contains the statement specific logic. This code starts with a switch statement to branch to the code for the decoded instruction and consists of three code blocks. First the instruction statements, that have been derived from partitioning the instruction behaviour code over the pipeline stages. The second block contains additional statements, which have been defined in the MiA behaviour code for specific instructions. And finally the specific transfer code move the data values for the decoded instruction through pipeline 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 WIEN Bibliothek Your knowledge hub

The third code block is the epilog, which will only be emitted for the last clock phase. This is a special code section that can only access the outgoing pipeline register values and can be used to change the pipeline stage outcome.

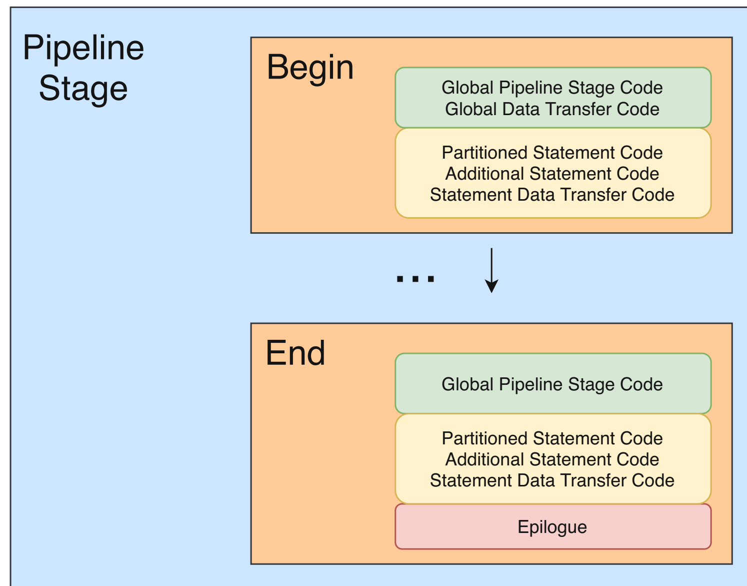

Figure 5.4: Pipeline Stage Code Layout

5.3.5 Code Generator Data Model

To generate the simulator code directly from the internal VADL model is usually a convenient and efficient approach. But if the complexity of the generated output exceeds a certain threshold it becomes difficult to maintain and one of this rare cases was the creation of the pipeline stage code layout. The reason for this is that the code consists of various independent code fragments, which should only be emitted when certain conditions apply. This led to many nested if-statements and a high cyclomatic complexity in the generator code. Introducing separate data objects, which contained only the prebuild code fragments that should be emitted, tremendously improved the code quality. The data objects are hereby created in the PipelineStageExecFragmentGenerator class, which will return a single data object containing all relevant global and specific code fragments listed by instructions and phases.

5.3.6 Instruction Retirement Component

The ability to emit an instruction trace, like described in section 3.2.1, is an important debugging tool for the generated simulator. But in contrast of the ISS, where all printed data values are well accessible at a single code location, this is not the case for the CAS. Data values are only being passed through the pipeline until their last usage in a pipeline

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

stage, which makes them typically inaccessible at the end of a pipeline data path. But instructions can also be flushed from the pipeline during earlier pipeline stages after a branch misprediction and should certainly not appear on the instruction trace in these cases.

CAS simulator implementations can avoid this problem by using a data structure which represents the decoded instruction with all its data values. Instead of relying on pipeline registers it would now be sufficient to pass a pointer to this data structure through the pipeline. But the CAS simulator has deliberately been designed with this limitation in mind to model a simulation that tends more to the hardware side and can have better synergies with the VADL hardware synthesis.

The instruction retirement component is now a simple data collector, which compensates for the described data accessibility problems, while also providing additional debugging and utility support. This component keeps track of code metrics like the number of instructions or consecutively fetched NOPs and also provides a unique sequence number for fetched instructions. This strictly increasing instruction number can be used in log files or the interactive mode to improve the readability, when the same instruction type is being issued multiple times. A final stage of a pipeline data path will issue a retireInstruction (...) call to this component and passing the instruction number and instruction type as arguments. The retirement component stores a cache of relevant values for the instructions that are currently being processed in the pipeline and also contains the program code to add the retired instruction to the instruction trace log file. The data cache consists of an array of a C++ structure, which can store the relevant value for all ISA instructions. The cache is being accessed by the modulo division of the instruction number by the cache size. Preserving additional logging information in the instruction cache slightly reduces the performance of the simulator and can be disabled in the InstructionRetirementGenerator.

5.4 MiA Behaviour

5.4.1 Fetching and Decoding of Instructions

Instruction fetching and decoding is not directly described by the VADL instruction definitions itself. But for the ISS this missing implementation details can be derived from the VADL format definitions and the program counter, which is specified in the ABI. But deriving the fetch and decode logic for a pipelined processor is much more of a challenge and would probably fail in at least some scenarios. Therefore, the necessary behaviour is being specified explicitly for the implemented MiA, which is done by adding additional program code to pipeline stages.

An example VADL fetch behaviour code is shown in Listing 5.7, which is assigned to previously defined END clock phase. The corresponding instruction decoding code is displayed in Listing 5.8. This given example shows how built-in keywords are being used to read a memory address and pass the instrValue value to another pipeline 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 WIEN Bibliothek Your knowledge hub

through the use of explicitly defined pipeline register values. When the memory read operation would fail at runtime, then this will be indicated by the available flag. The shown implementation will then stall the *IF stage*, based on this flag for the current clock cycle. Processing of the memory read operation will then be resumed at the following clock cycle and can be stalled again, until the memory value would eventually be available. The decode instruction keyword of the ID stage works in a similar manner and will provide a valid flag, to indicate that the instruction decoding has failed. While the current VADL behaviour code only has limited error handling abilities, this will certainly be extended in a future version.

```
IF @ END {
  let (available, instrValue) = MEM.read(REG_PCGEN_IF.PC) in {
    if (available != 0) {
      REG_IF_ID.instr := instrValue
    } else {
      stall()
    }
  }
}
```

Listing 5.7: Instruction-Fetch Pipeline Stage Specification

```
ID @ END {
  let (isValid, instructionId) = decodeInstruction( REG_IF_ID.instr ) in {}
}
```

Listing 5.8: Instruction-Decode Pipeline Stage Specification

5.4.2 Branching

The branching logic for the CAS, also has to be specified by adding additional behaviour code. This reliefs the partitioning approach from dealing with backedges in the data paths and therefore simplifies many implementation aspects of the simulator generator. But the explicit definition of the branching logic also preserves a great deal of flexibility to integrate future components like *branch target buffers* or *branch target address caches*, which are currently not supported yet in VADL.

The Listing in 5.9 shows how a *branch not taken* strategy can be specified for the PC_GEN stage. In the case of a misprediction, when a branch has been taken, then the three successor pipeline stages have to be flushed, which can be described by using the built-in *flush* instruction. The program counter is also being increased by the format width, which is 4 bytes in the case of the shown *RV32IM* specification. While this VADL code specifies how the branch strategy is being implemented, it is also necessary to set the branch taken flag and to compute the branch target address.

To do that it is first necessary to define both values in a pipeline register, with the code that has already been shown in Listing 5.3. The computation of both values is being

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

performed in context of this example in the *EX stage* and by using naming conventions it is possible to avoid additional code for most cases. Listing 5.10 shows the specification for a RISC-V *branch if equal* instruction. This code listing uses of the branch_taken identifier in a let variable and will therefore be automatically assigned to the outgoing pipeline register value of the same name.

The computed branch target address is hereby the value that is being assigned to the program counter. Creating another let expression with the name branch_addr would work similar to assign the pipeline register value, but also reduce the readability of the VADL code. While the section 5.5 will describe the instruction partitioning process in more detail, this mechanism will also divide the shown VADL code into smaller instructions, whenever possible. This is done by rewriting the AST and introducing a new let expression for the right hand side of an assignment, which separates the assignment from the computation of the assigned value. While this is primarily done with the intention to move the instruction behaviour more freely between different pipeline stages, it also conveniently introduces an identifier for the right hand side of an assignment instruction. This identifier can then be altered by VADL computation assignments and can be used for the current example as follows: *compute assignment* to PC *as* branch_addr *in* EX. This assignment rule will ensure that the assignment value for PC will be computed in the *EX stage* and stored in a newly introduced let variable with the name branch_addr, which will then be assigned to the pipeline register attribute of the same name.

```
PC_GEN @ END {
  if (REG_EX_MA.branch_taken != 0) {
    IF.flush()
    ID.flush()
    EX.flush()
    PC := REG_EX_MA.branch_addr + 4
    REG_PCGEN_IF.PC := REG_EX_MA.branch_addr
  } else {
    PC := PC + 4
  }
}
```

Listing 5.9: PC-Gen Pipeline Stage Specification

```
instruction BEQ : B_TYPE = {
  let branch_taken = (X(rs1) = X(rs2)) in {
    if branch_taken {
      PC := PC + ImmediateB
    }
  }
}
```

Listing 5.10: VADL Specification of the RISC-V Branch-If-Equal Instruction

But assigning pipeline register attribute values by matching identifier names is not always

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

an option. One example of this is the unconditional RISC-V branching instruction *jump-and-link*, which is shown in Listing 5.11. While the value assignment results in the expected outcome in the standard case, it does not in the case of a branch misprediction. Inserting an additional let instruction can be avoided in this case by manually adding the code fragment from Listing 5.12, which will add the necessary assignment for the unconditional branching instructions.

```
instruction JAL : J_TYPE = {
  X(rd) := PC + 4
  PC := PC + ImmediateJ
}
```

Listing 5.11: VADL Specification of the RISC-V Jump-And-Link Instruction

```
EX append @ BEGIN for JAL, JALR {
        REG_EX_MA.branch_taken := 1
}
```

Listing 5.12: Set branch_taken Flag for Unconditional Jump Instructions

The emitted C++ code for this example is displayed in Listing 5.13 and consists of code fragments assigned to the *begin* and *end phase* of the clock cycle. Both of these code sections will only be executed if the state of the pipeline stage is active and not being stalled for example. Note that the transfer of the current program counter value in the *begin phase* into the outgoing pipeline register REG_PCGEN_IF is a special case. Until the instruction type hasn't been decoded, it is not known if this value will later be needed in the pipeline and therefore always has to be transported to the instruction decode stage. Also note that the listed VADL behaviour code in Listing 5.9 also changes the program counter value in the case that a branch has been taken in the *end phase*. Therefore it is necessary to manually adjust the value in this case by the instruction REG_PCGEN_IF.PC := REG_EX_MA.branch_addr.

```
class PipelineStage_PC_GEN : protected IPipelineStage
  ...

  void exec_BEGIN() {
    if (processorState.getStatePC_GEN() != ACTIVE) { return; }
    outREG_PCGEN_IF.PC = PC.read();
  }

  void exec_END() {
    if (processorState.getStatePC_GEN() != ACTIVE) { return; }
    outREG_PCGEN_IF.PC = PC.read();

    // global code for pipeline stage
    if (inREG_EX_MA.branch_taken != 0) {
      processorState.setStateIF(FLUSH);
```


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

```
      processorState.setStateID(FLUSH);
      processorState.setStateEX(FLUSH);
      PC.write(inREG_EX_MA.branch_addr + 4);
      outREG_PCGEN_IF.PC = inREG_EX_MA.branch_addr;
    } else {
      PC.write(PC.read() + 4);
    }
  }
}
```

Listing 5.13: Emitted C++ Code for the Pipeline Stage PipelineStage_PC_GEN

```
pipeline register REG_PCGEN_IF between PC_GEN and IF {
  PC
}
```

Listing 5.14: Explicit Pipeline Register Definition for REG_PCGEN_IF

5.4.3 Pipeline- and Data Hazards

The execution of multiple instructions can overlap in a pipelined processor design, which can cause various types of data hazards. While this is preventable by stalling the pipeline stages, it is certainly not desirable for the resulting degradation of pipeline throughput. Techniques like *forwarding* can significantly reduce the requirement to stall for Read-After-Write (RAW) hazards, which commonly are a consequence of reading register values, while the relevant write back operations of earlier instructions have not been executed yet.

An example RAW hazard for the described 6-stage pipeline architecture can be seen by the first two lines of Listing 3.6. The ADDI instruction in line 1 writes the computation result to the register X2, which is being used as a source operand for the following SW instruction. Implementing a *forwarding* techniques proofs quite effective to avoid this kind of data hazards and the next section will describe how it can be specified in VADL.

The code in Listing 3.6 contains an extraordinarily potential for data hazards, because it has been compiled with –O0 to avoid code folding optimizations. While an optimizing compiler can certainly reduce the number of potential data hazards in a compiled program, it is still necessary to handle these cases for a CAS to correctly simulate any meaningful program.

5.4.4 Forwarding Example for a 6-Stage RISC-V Pipeline

Forwarding or *operand forwarding*, is an optimization technique to avoid pipeline stalls by transferring computation results to earlier pipeline stages. Writing back computed values into a register file for the described 6-stage pipeline architecture is being performed at the WB pipeline stage. But values for register source operands are already being read after the instruction decoding process at the *ID stage* and transferred to successor pipeline

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

stages through pipeline registers. Operand forwarding can now replace stale data values in the pipeline register with the more recent computation results.

Describing a forwarding logic in VADL works similar like shown before, by assigning code fragments to pipeline stages. The difference is that defined code fragments can not only be added to a specific phases of the clock cycle, but can be restricted for certain instructions, which can be seen in Listing 5.15. The code fragment of this example is added to code for the *EX stage* for the ADD, ADDI, LH, LW instructions. It is possible to assign multiple code fragments for the same instruction, pipeline stage and phase of the clock cycle, in which case all fragments are being concatenated in the order in which they are defined in the VADL specification. The shown `epilog` keyword assigns the code fragment automatically to the last defined phase of the clock cycle and the fragment will also be emitted at the very end of the emitted code blocks, despite the order in which the epilog is defined in the VADL file. The epilog is meant to specify additional behaviour code that can alter the outcome of a pipeline stage and will also only access the outgoing pipeline register values.

The forwarding logic shown in Listing 5.15 can directly look up the value of the outgoing register destination `rd`. If the register destination targets the zero register `X0` then no value has to be forwarded. Otherwise the register destination has to be compared to the source operands of the previous pipeline register values and be replaced if necessary. The same forwarding logic is necessary for the *MA stage*, with the difference that operand forwarding from the earlier *EX stage* takes precedence.

```
EX epilog for ADD,ADDI,LH,LW,...  {

   if ( REG_EX_MA.rd != 0 ) {
     if ( REG_EX_MA.rd = REG_ID_EX.rs1 ) {
       REG_ID_EX.X_rs1_ := REG_EX_MA.result
     }

     if ( REG_EX_MA.rd = REG_ID_EX.rs2 ) {
       REG_ID_EX.X_rs2_ := REG_EX_MA.result
     }
   }
}

MA epilog for ADD,ADDI,LH,LW,... {
   if ( REG_MA_WB.rd != 0 ) {
     if ( REG_MA_WB.rd = REG_ID_EX.rs1 ) {
       /** only forward this value if EX_MA doesn't didn't forward */
       if (REG_EX_MA.rd != REG_ID_EX.rs1) {
         REG_ID_EX.X_rs1_ := REG_MA_WB.result
       }
     }

     if ( REG_MA_WB.rd = REG_ID_EX.rs2 ) {
       /** only forward this value if EX_MA doesn't didn't forward */
       if (REG_EX_MA.rd != REG_ID_EX.rs2) {
```


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

```
        REG_ID_EX.X_rs2_ := REG_MA_WB.result
      }
    }
  }
}
```

Listing 5.15: Forwarding Logic for the 6-Stage RISC-V Pipeline Example

While forwarding is a very useful technique it can't solve every type of potential data hazard. A corner case for the discussed 6-stage pipeline is the load instruction, where the resulting value from the memory access is only available after the *MA stage*. Concurrent computations in the *EX stage* would still be based on the previous values and yield wrong results. An instruction trace with this problem can be found in Listing 3.6 at lines 9 and 10, which uses the same register file value X15. While operand forwarding doesn't work for this particular case, it can be easily be resolved by inserting a pipeline bubble with the code fragment from Listing 5.16.

```
EX epilog for LW, LH, LB, LBU, LHU {
  if (REG_EX_MA.rd = REG_ID_EX.rs1) {
    ID.stall()
  }
  if (REG_EX_MA.rd = REG_ID_EX.rs2) {
    ID.stall()
  }
}
```

Listing 5.16: Hazard Detection Logic for the 6-Stage RISC-V Pipeline Example

5.4.5 System Calls

While the actual delegation mechanism of a system call works as described for the ISS, it is still required to finish the processing of already issued instructions in advance. Typically the last instructions which are being issued before a system call, will set up the system call arguments by initializing register file values. Waiting until all previous instructions have been retired will ensures that the necessary initializations have been completed, before a system call is being delegated to the operating system. A related problem also occurs for the system call results, which will be available in the register file only after the system call has been resolved. If another instruction would be decoded in parallel, it would also potentially fetch the wrong results, similar like in the forwarding examples from before.

One possibility to implement this behaviour in VADL is being shown in Listing 5.17. This behaviour fragment uses the *isIdle* keyword to stall all system calls until the successor pipeline stages have successfully finished to process the previously issued instructions.

The additional necessary VADL definition in Listing 5.18 specifies that the actual signal processing for the system calls is being issued in the *EX stage*. The second code fragment

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

that is being attached to the *EX stage* in Listing 5.17 finally stalls the instruction decoding until the system call has been resolved.

```
ID epilog for ECALL, EBREAK {
  if (EX.isIdle() = 0) {
    stall()
  }
  if (MA.isIdle() = 0) {
    stall()
  }
  if (WB.isIdle() = 0) {
    stall()
  }
}

EX epilog for ECALL, EBREAK {
  ID.stall()
}
```

Listing 5.17: Stall Logic for System Calls in the 6-Stage RISC-V Pipeline Example

```
computation assignments:
        generate syscall for ECALL, EBREAK in EX
```

Listing 5.18: Define at which Pipeline Stage a System Call should be issued

The generated C++ code for the *EX stage* will hereby only issue the processor signal for the system with the code from Listing 5.19. The resolution of the system call will then occur in the main processor loop, after finalising the previous clock cycle, which can be seen in Listing 5.20.

```
case EBREAK:
case ECALL: {
  processorState.signal(SIGNAL_SYSCALL);
}
```

Listing 5.19: Emitted C++ Code to Issue a System Call in a PipelineStage

```
if (processorState.hasSignal(SIGNAL_SYSCALL)) {
  sysCall.emulate();
  syscall_counter++;
  processorState.signal(SIGNAL_NONE);
}
```

Listing 5.20: Emitted C++ Code to Resolve a System Call in the Processor

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

5.5 Instruction Partitioning

The definition of functional instruction behaviour in VADL is part of the ISA description and is strictly separated from the MiA definition. The specification of a MiA in VADL is actually derived from an ISA and ABI specification and therefore it is possible to access the instruction behaviour in the MiA, but not the other way around. This separation of concepts is one of the major design aspects in VADL and a cornerstone to maintain its flexibility and maintainability in the future. But to implement a CAS, the instruction behaviour code has to be split up and being distributed over the defined pipeline stages.

Assume the example of an arithmetic addition of two register values, for which the VADL definition is being shown in Listing 5.21. The desired output for this instruction definition, given the 6-stage pipeline architecture, is that the instruction decoding is being done in the *ID stage*. But the *ID stage* also will have to transfer both source register values to the *EX stage*, where the arithmetic addition should be performed. The resulting value of this operation will then just be routed through the *MA stage* and finally be assigned to the register destination in the *WB stage*.

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

Listing 5.21: VADL Specification of the RISC-V ADD Instruction

Achieving a partitioning like this, tears down to the tasks of splitting the instruction behaviour into small and moveable code fragments before assigning them to a pipeline stage, while also using pipeline register attributes to transfer necessary input and output values between these fragments. This was designed with the intention to minimize the necessary VADL description code and resulted in the specification of access and computation grants. While access grants define coarse grained read and write privileges to MiA components, the computation grants allow a more fine tuned assignment of code fragments to pipeline stages.

5.5.1 Access Grants

VADL Access grants determine if a component can be accessed from program code within a pipeline stage and phase. The modelled access privileges for the previously described 6-stage pipeline are being depicted in Figure 5.5. Register components are hereby displayed in colour green and the memory component in colour red.

Access privileges can be modelled in VADL in one of two ways. First by defining that a component belongs to a pipeline stage as a nested component. An example of this is the PC register in the code in Listing 5.22, which defines this register as a nested component for the *PC_GEN stage*. A pipeline stage hereby has full access privileges to its nested components. The second way is to specify read or write access grants to components, which is also being shown in the same code listing. Each access grant is limited to a

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

specified clock phase, but it is possible to define grants for all clock phases of a pipeline stage if necessary, by adding multiple entries in the VADL specification. The defined grants in Listing 5.22 correspond to the depiction in Figure 5.5.

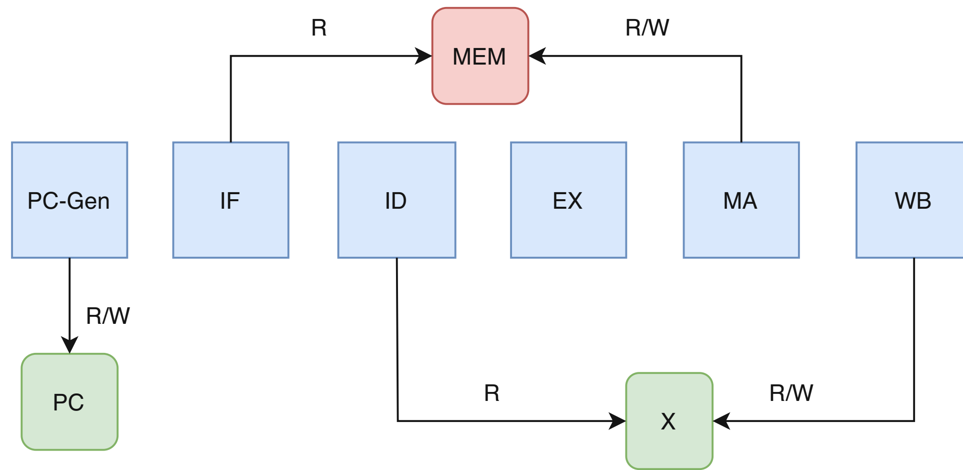

Figure 5.5: CAS Component Access for 6-Stage Pipeline Architecture

```
pipeline stage PC_GEN {
  register PC
}

access grants:
  IF:
    read MEM @ END

  ID:
    read X @ END

  MA:
    write MEM @ END
    read MEM @ END

  WB:
    write X @ BEGIN
```

Listing 5.22: VADL Definition of Access Grants

5.5.2 Computation Grants

Computation grants associate statements from VADL instruction definitions to pipeline stages. While the previously shown VADL code for the arithmetic addition in Listing 5.21 does only consist of a single line, its computation should be performed in the *EX 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 WIEN Bibliothek Your knowledge hub

but the result should only be written into the register file at the later *WB stage*. The corresponding VADL computation assignment rule is shown in Listing 5.23 and specifies that the computation of any assignment to the register file X should be computed in the EX stage and stored in a temporary variable by the name of result. This computation assignment in combination with the previously defined access grants, where the register file can only be altered from within the pipeline stage WB, is enough information to successfully apply the partitioning algorithm for this example.

```
computation assignments:
  compute assignment to X as result in EX
```

Listing 5.23: VADL Computation Grants for the ADD Instruction

But lets discuss a more complicated example with the load halfword instruction from Listing 4.3, for which the computation assignments are shown in Listing 5.24. The first line assigns the computation of the immediate value to the instruction decode stage and also specifies a name for the temporary variable. The two remaining computation rules will assign the right hand side of the address computation to the *EX stage* and the right hand side of the assignment to the register value of X to the *MA stage*. Note that these two computation assignments will also only affect the listed instructions, like LH, while the previously shown Listing in 5.23 was not restricted to a specific instruction. If multiple computation assignments exist for a single component or a single let variable name, then the more specific one will be used for the instruction partitioning algorithm. In the case of the shown example this means that the more general rule for the assignment of the register X is being ignored for the LH instruction, because a more specific one exists in Listing 5.24. The emitted C++ code from these definitions for the involved pipeline stages can be seen in Listing 5.25.

```
computation assignments:
  compute ImmediateI as immediate in ID

  compute let assignment to addr for LB, LH, LW, LBU, LHU as addr in EX
  compute assignment to X for LB, LH, LW, LBU, LHU as result in MA
```

Listing 5.24: VADL Computation Grants for the LH Instruction

```
pipeline stage ID:
  const uint8_t rs1 = (instr >> 15) & 0x1f;
  const uint8_t rd = (instr >> 7) & 0x1f;
  const uint16_t imm = (instr >> 20);
  const uint32_t ImmediateI = ...;
  const int32_t immediate = ImmediateI;

pipeline stage EX:
  const uint32_t addr =
    (int32_t) inREG_ID_EX.X_rs1_ + (int32_t) inREG_ID_EX.immediate;
```


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.

```
pipeline stage MA:
  uint16_t t2 = 0;
  MEM.read2Byte(inREG_EX_MA.addr, t2);
  const int32_t result = sext16To32(t2);

pipeline stage WB:
  X.write(inREG_MA_WB.rd, inREG_MA_WB.result);
```

Listing 5.25: Emitted C++ Pipeline Stage Code for LH Instruction

5.5.3 Partitioning Instructions Over a Data Path

The partitioning algorithm is being applied for each defined VADL pipeline data path. The first step is hereby to split the instruction behaviour code into smaller code fragments. The second step is to assign each statement to a specific pipeline stage according to the specified *access grants* and *computation assignments*. The third and final step is to ensure that the input values are accessible for each assigned statement.

Split Instruction Behaviour Code

Splitting the instruction behaviour code into smaller fragments, improves the degree of freedom in which instruction behaviour can be partitioned over pipeline stages. The instruction behaviour code is being split up by rewriting its internal AST representation for assignment-, let- and if-statements. This behaviour has already been briefly mentioned in the section describing the branching logic in section 5.4.2.

For assignment- and let statements this is a simple computation step, which replaces the right hand side of the statement by introducing a new enclosing let statement. Every usage of the right hand side in the originating statement as well as in nested statements is then replaced by the introduced let variable name. The name of the let variable can be altered by *computation assignments*, which greatly improves the readability for values that have to be transferred to other pipeline stages. For the previously shown arithmetic instruction from Listing 5.21, the changed code version is printed in Listing 5.26. Please note this code listing can also be directly written in VADL, but it would not be as concise as the original version.

```
instruction ADD : R_TYPE = {
  let result = X(rs1) + X(rs2) in {
          X(rd) := result
  }
}
```

Listing 5.26: Altered VADL Specification of the RISC-V ADD Instruction

The remaining statement to discuss is the if-statement, that occurs for example in the behaviour code of the *set-less-than* instruction (SLT), which is shown in Listing 5.27.

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

Partitioning of an if-statement has two major concerns in this context. How to partition nested statements and if the *then* and the *else* branch have to be moved to the same pipeline stage. A solution for both concerns is to rewrite if-statement into guarded instructions. This will split up each nested statement for the *then* and also for the *else* branch into separate statements. While a guarded instruction statement currently only exists for the internal representation of the simulator generator, its code would look like the depiction from Listing 5.28.

Rewritten instruction behaviour code can hereby certainly seem less efficient. This is due to introduced temporary variables and guarded instructions, which will often be emitted as two separate *if-statements* instead of the originally written *if-then-else* command. But each common C++ compiler should be able to optimize these minor details without any problems.

```
instruction SLT : R_TYPE = {
  let result = (X(rs1) < X(rs2)) in {
    if result {
      X(rd) := 1
    } else {
      X(rd) := 0
    }
  }
}
```

Listing 5.27: VADL Specification of the RISC-V Set-Less-Than Instruction (SLT)

```
instruction SLT : R_TYPE = {
  let t1 = (X(rs1) < X(rs2)) in {
    let result = t1 in {
      when( result , X(rd) := 1)
      when( not(result), X(rd) := 0)
  }}
}
```

Listing 5.28: Internal Representation for the Rewritten SLT Instruction

Assign Statements to Pipeline Stages

While *computation assignments* also have been discussed earlier in the section 5.5.2, there still remains a noteworthy concern to discuss. The current implementation expects a complete and valid assignment of all instruction behaviour statements to a specific pipeline stage. While a heuristic algorithm could probably be implemented to create valid assignments, even if some configuration entries are missing, there is no such mechanism in place yet. And there is also no feedback mechanism yet, which would point out missing or erroneous configuration entries in this context, other than the currently written log outputs.

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

An example for a valid log output for the SLT instruction is shown in Listing 5.29 and shows in which pipeline stage and clock phase each statement has been assigned, but also gives valuable information on the transferred instruction values. Values marked with the *eval* keyword will only be used in current pipeline stage, while all entries with *evalAndTransfer* will also be transferred through pipeline registers to the next pipeline stage. Each *transfer value* will be passed through to the next pipeline stage, but would also be accessible for the statements in the current pipeline stage.

This log output example shows that the SLT instruction only uses the source operands rs1 and rs2 in the *ID stage*. But the accessed register values for these source operands are also being transferred to the *EX stage*. The *EX stage* will then compute the result value and transfer it to the next pipeline stage, along with the passed through value for rd. The *MA stage* has no statements assigned to it and will only route register values trough for the SLT instruction. The last *WB stage* contains two guarded instruction values, which will write the value zero or one to the register file at index rd, depending on the result condition value.

```
partitioned instruction: SLT
pipeline stage: PC_GEN
pipeline stage: IF
pipeline stage: ID
  value: evalAndTransfer: rd
  value: evalAndTransfer: X(rs1)
  value: eval: rs1
  value: evalAndTransfer: X(rs2)
  value: eval: rs2
pipeline stage: EX
  statement (BEGIN): let result = (X(rs1) < X(rs2)) in (...)
  value: evalAndTransfer: result
  value: transfer: rd
pipeline stage: MA
  value: transfer: result
  value: transfer: rd
pipeline stage: WB
  statement (BEGIN): if ( result ) then write(X(rd)) := 1
  statement (BEGIN): if ( (!result) ) then write(X(rd)) := 0
```

Listing 5.29: Internal Representation for the Rewritten SLT Instruction

The instruction partitioning is quite sensitive to altered VADL specifications. Verifying that the algorithms produced the expected outcome is unfortunately work intensive, because the log outputs contain entries for all instructions and data path. One hint that something went wrong can be log entries like unallocated statements: *write*(PC) := branch_addr, which shows that some statements couldn't be assigned to a pipeline stage. But in this case from the JALR instruction this is still valid, because the branch logic was explicitly defined by a backedge, which can't be automatically handled by the partitioning algorithms and was actually covered by the added MiA behaviour 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 WIEN Bibliothek Your knowledge hub

Transfer Input Values

After each statement has been assigned to a pipeline stage, the partitioning algorithm can start to determine the origin of required input values. This process starts at the final pipeline stage, until all assigned statements have been processed and then repeats this procedure for the predecessor pipeline stage, until there are no more remaining predecessors.

The algorithm will iterate through all statements, which have been assigned to a pipeline stage and will try to determine a source for each input operand.

An input value can be directly accessed in a pipeline stage, if:

- it is contained in an incoming pipeline register
- it is computed in a let statement
- it is a format field and this pipeline stage and clock phase performs the instruction decoding
- it is an buffered value that is directly derived from a format field and can be computed at this pipeline stage
- it can be read from a memory or register file component

When an input value is not contained in an incoming pipeline register, but can be evaluated in the current pipeline stage by these rules, then this value will be marked with the *eval* keyword. If the input value can't be accessed in the local pipeline stage, then a recursive search through the predecessor pipeline stage is initiated. This search stops at the first location in the pipeline that can provide the required input value. The data value in this stage will be marked with *evalAndTransfer* and for all intermediate pipeline stages the same data value will also be marked by the *transfer* keyword.

The order in which statements are being processed by this algorithm, has to correspond to their order in which they occurred in the original instruction behaviour code. The step in which statements are being applied to pipeline stages therefore has to preserve this property.

The results of this algorithm for the *SLT instruction* can be seen in the log output in Listing 5.29.

5.6 Interactive Mode

The interactive mode of the generated CAS resembles the already described one from the ISS, with the difference that each simulation step will only advance the simulation by a single clock cycle and not necessarily retire an instruction at each step. An example output for the same addition program from Listing 3.5 is being shown in Figure 5.6.

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 WIEN Bibliothek Your knowledge hub

The interactive mode again lists the latest retired instructions at the beginning of its output, but is then followed by basic metrics like the current clock cycle, the number of retired instructions and if a processor signal has been issued.

The next output block lists the pipeline stages by their names and prints their current state as active, stalled or flushed. If a pipeline stage contains an instruction it is being printed with its unique instruction number, the memory address it has been fetched from, its type and its decoded parameters. The last line lists the currently retired instruction from this clock cycle, if one exists.

```
#1 10054 - ADDI: rd: 2 (0x2) rs1: 2 (0x2) ImmediateI: -32 (0xffffffe0)
#2 10058 - SW: rs1: 2 (0x2) rs2: 8 (0x8) ImmediateS: 28 (0x1c)
#3 1005c - ADDI: rd: 8 (0x8) rs1: 2 (0x2) ImmediateI: 32 (0x20)
#4 10060 - ADDI: rd: 15 (0xf) rs1: 0 (0x0) ImmediateI: 3 (0x3)
#5 10064 - SW: rs1: 8 (0x8) rs2: 15 (0xf) ImmediateS: -20 (0xffffffec)
#6 10068 - ADDI: rd: 15 (0xf) rs1: 0 (0x0) ImmediateI: 4 (0x4)
#7 1006c - SW: rs1: 8 (0x8) rs2: 15 (0xf) ImmediateS: -24 (0xffffffe8)

cycle: 12       retired instr: 7                signal: none
PC_GEN (#stall):        -
IF (#stall):    -
ID (#stall): (#10)      10078 - ADD: rd: 15 (0xf) rs1: 14 (0xe) rs2: 15 (0xf)
EX (#active): (#9)      10074 - LW: rd: 15 (0xf) rs1: 8 (0x8) ImmediateI: -24 (0xffffffe8)
MA (#active): (#8)      10070 - LW: rd: 14 (0xe) rs1: 8 (0x8) ImmediateI: -20 (0xffffffec)
WB (#active): (#7)      1006c - SW: rs1: 8 (0x8) rs2: 15 (0xf) ImmediateS: -24 (0xffffffe8)
#8 10070 - LW: rd: 14 (0xe) rs1: 8 (0x8) ImmediateI: -20 (0xffffffec)
>
```

Figure 5.6: CAS Interactive Mode

The CAS interactive mode also uses colours to improve the readability and provides equal navigation and lookup mechanisms like described for the ISS interactive mode. But there are two noteworthy additional command options. First there are the names of the pipeline stages, which will print the emitted program code for the current instruction that is being processed in that pipeline stage in this clock cycle. An example can be seen in Figure 5.7, which prints the code fragment of the LW instruction that has been assigned to the EX pipeline stage. The second new command options are the names of the pipeline registers, which will print the values of the pipeline register at the current and next clock cycle.

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

```
cycle: 12        retired instr: 7                signal: none
PC_GEN (#stall):         -
IF (#stall):     -
ID (#stall): (#10)      10078 - ADD: rd: 15 (0xf) rs1: 14 (0xe) rs2: 15 (0xf)
EX (#active): (#9)      10074 - LW: rd: 15 (0xf) rs1: 8 (0x8) ImmediateI: -24 (0xffffffe8)
MA (#active): (#8)      10070 - LW: rd: 14 (0xe) rs1: 8 (0x8) ImmediateI: -20 (0xffffffec)
WB (#active): (#7)      1006c - SW: rs1: 8 (0x8) rs2: 15 (0xf) ImmediateS: -24 (0xffffffe8)
EX:
LW@BEGIN
        const uint32_t addr = (int32_t) inREG_ID_EX.X_rs1_ + (int32_t) inREG_ID_EX.immediate;
```

Figure 5.7: CAS Interactive Mode listing the Code Fragment of the LW Instruction for the EX Pipeline 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 WIEN Bibliothek Your knowledge hub

CHAPTER 6

Evaluation

6.1 Methodology

6.1.1 VADL Specifications

The evaluation of the ISS and CAS generators has been primarily conducted on the basis of a VADL specification for the unprivileged ISA *RISC-V* specification[risc], which also has been used in various examples throughout the previous chapters. This specification describes the *RISC-V* 32 bit base integer function set (*RV32I*) with extensions for multiplication and division (*M*), the control and status registers (Zicsr) and the compressed instruction set (*C*).

Three different ISS variants have been generated from this VADL specification and named after the supported modules (V32/I, V32/IM, V32/IMC). Additionally three corresponding 64 bit ISS have created from an extended 64 bit VADL specification (V64/I, V64/IM, V64/IMC). The 64 bit specification is currently still less mature than the 32 bit variant and produced wrong results for very few of the used benchmarks. This will be discussed in more detail during the compliance section 6.2 and it will explicitly stated throughout this section when some results are being omitted.

A similar situation applies for the CAS, which can currently only be generated for the RV32I base module. This results from a problem in the CAS generator implementation, which can't yet generate the necessary division by zero and integer overflow cases for the *DIV* and *REM* instructions according to the *RISC-V* specification. While this problem should be resolved quite soon, only a limited test set for the CAS with a 5-stage pipeline has been conducted in the meantime.

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

6.1.2 Test System

All conducted tests have been performed on a 6 core / 12 thread AMD Ryzen™5 3600X with a base clock of 3.8Ghz and a boost clock of up to 4.4Ghz. The used operating system was a 64 bit Linux with kernel 5.8.0-29 and all generated simulators have been compiled with the gcc 9.3 compiler and -O3 if not stated otherwise.

6.1.3 Benchmarks and Tools

Runtime benchmarks have been conducted by using the tool hyperfine, which controlled and recorded the repeated execution of benchmarks[hyp]. Each benchmark has hereby been executed 10 times after a warm up phase, while all background processes have been disabled as far as possible. When a benchmark allowed to adjust its runtime, by setting an iteration count for example, then a reasonable value has been chosen to avoid unnecessary long run times.

As performance benchmark a self written *fibonacci* program has been used along side the Dhrystone V2.2[dhr] and MiBench V1.0[mib]. All benchmark programs have been compiled with the *RISC-V GNU Compiler Toolchain* in version 10.2[gnu] and an -O3 flag if not stated otherwise.

It was necessary to slightly adapt the code of the Dhrystone benchmark, because it still used an older *C syntax* for passing method arguments, which was not supported by the used compiler toolchain. Additionally the internal timing infrastructure of this benchmark was also disabled, because it could not be compiled without additional efforts and was not used for benchmarking anyway.

From MiBench all benchmarks from the automotive and network modules have been used, without *susan* and *patricia*, which did not seem to work out of the box and have therefore been omitted due to time constraints.

For a direct comparison of the ISS performance the handwritten *RISC-V* SWERV-ISS has been used [Rah]. This ISS can emit the number of retired instructions, but doesn't seem to count the number of taken branches for the simulated program, which are therefore omitted from the shown results.

Host performance counters like executed cycles, instructions, branches and missed-branches have been measured with the Linux tool *perf_events*.

6.2 Functionality and RISC-V Compliance

The primary concern for a generated simulator is that it can correctly reproduce the computation results of an executed program. Manual tests with relatively simple programs have been conducted repeatedly throughout the initial development stages of the ISS to get a basic confidence in the generated simulator code, which helped to detect various programming and *RISC-V* specification errors. And this collection of test programs has

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

constantly been extended over time when new features have been implemented in the simulator generator code base. But even small test programs that often only consist of a few lines of *C* code, can already be compiled to machine code with hundreds or even thousands of instructions and can quickly becomes very tedious during debugging. Features like an instruction trace or the interactive execution mode have been added very early as a consequence to reduce debugging efforts whenever possible. Building the simulator generator with a specific and well known ISA like *RISC-V* also tremendously reduced debugging efforts, because it allowed to use well matured simulators like SWERV-ISS ([Rah]) to get reference instruction traces for comparison. It is noteworthy to mention that program traces can deviate between different simulators when the simulated heap and stack are being initialised differently, for example with and without alignments for passed program arguments.

But even a very thorough manual testing approach will probably not detect subtle problems that occur only in few specific situations. Assume for example a jump instruction that will omit one of most significant bits for a 20 bit immediate value during instruction decoding. Such an error is especially hard to detect because it still works for most cases and will therefore probably be detected during the run of larger programs that are especially time consuming to debug.

To avoid such pitfalls it is very advisable to execute **compliance tests** for each defined instruction, which can methodically test various bit values and corner cases and also can ensure that the ISA specification has been understood correctly. But efficiently writing such tests requires expertise and is also very time consuming. Therefore the project from the *RISC-V Compliance Task Group* at [BM] has been used for this purpose.

The compiled test programs from this project contain two *ELF-symbols* (beginSignature and endSignature) which specify a memory region that has to be emitted after the test run and will be compared against the output of a reference implementation. The generated ISS and CAS simulators can also emit this test signature by providing the –s parameter with a signature filename at startup.

Running these compliance tests led to some modifications in the VADL specification for the *RISC-V* ISA, especially for corner cases and probably avoided future debugging efforts.

But the used compliance project for *RISC-V* is currently primarily focussed on the 32 bit specification and only has rudimentary support for 64 bit at the time of this writing. Additionally the project always requires the additional instructions for the control and status registers (Zicsr), even when only the base integer instruction set is being tested. And the test approach from these compliance tests is of course not exhaustive in respect that only a carefully selected and specified set of input values is being used throughout the test cases.

The generated RISC-V ISS simulators do still pass all compliance checks of this project for RV32I, M, C and Zicsr, with the exception of four test cases of the base integer instruction set. These four failing tests have been created with the assumption of a trap

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

handling behaviour that is probably not required by the *RISC-V* specification and also topic of a discussion in the issue tracker of the project.

6.3 ISS Performance

6.3.1 Generation Time

It typically took less than 250ms from parsing the VADL specification for *RISC-V* until the ISS or CAS have been generated and written to disk. And creating even multiple simulators at once from a given specification did only marginally increased the generation time by about 50ms per simulator.

6.3.2 Fibonacci

The recursive computation of the 30th fibonacci number has been used as self written benchmark to get a first impression of the execution performance for the generated ISS and is shown in Listing 6.1.

Compiling this benchmark for different *RISC-V* modules (I, M, C) for 32 and 64 bits produces different compilation targets that only use a limited set of instructions and can be used to get some insights for the effect of the instruction decoding and effects of the compressed instruction set on the runtime performance. Note that the fibonacci30 program does not contain any multiplication or division operations. Therefore the code for the emitted RV32I and RV32IM compilation targets is identical, while the RV32IMC compilation target will also contain compressed instructions.

```
uint32_t fibonacci(uint32_t v) {
    if (v <= 1) return v;
    return fibonacci(v - 1) + fibonacci(v - 2);
}
```

Listing 6.1: Recursive Fibonacci Compuatation

The Table 6.1 lists the fibonacci30 execution metrics for the VADL ISS V32/IMC and V64/IMC compared to the handwritten *SWERV-ISS* at different compilation targets that have been compiled with -O3. The Figure 6.1 additional illustrates the execution runtime for these ISS. The *SWERV-ISS* simulator can execute 32 and 64 bit programs and shows quite balanced runtime results throughout all benchmarks. The VADL ISS execution times are about 6% slower for 32 bit and about 11.5% slower for 64 bit than the *SWERV-ISS*, as long as no compressed instructions are being involved. With compressed instructions the V32/IMC has about a 12% faster runtime than the *SWERV-ISS* and for 64 the runtime is improved roughly by 14%.

The apparent reason for this discrepancy is the different way of how compressed instructions are being handled by the simulator generator as described in section 3.4.4. When multiple instruction formats are being specified the instruction decoder first tries to only

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

ISS	Compilation Target	#instructions	#branches	Runtime mean/stddev (ms)	MIPS
V32/IMC	RV32I	25.284.993	1.864.006	330.51 (7.61)	76.50
SWERV	RV32I	25.284.993	-	312.40 (7.70)	80.93
V32/IMC	RV32IM	25.284.993	1.864.006	331.02 (3.84)	76.39
SWERV	RV32IM	25.284.993	-	311.00 (4.07)	81.30
V32/IMC	RV32IMC	25.284.993	1.864.006	258.50 (3.17)	97.81
SWERV	RV32IMC	25.284.993	-	294.73 (9.14)	85.79
V64/IMC	RV64I	25.491.164	1.864.004	350.49 (4.00)	72.73
SWERV	RV64I	25.491.164	-	309.08 (4.06)	82.47
V64/IMC	RV64IM	25.491.164	1.864.004	351.52 (2.44)	72.52
SWERV	RV64IM	25.491.164	-	314.64 (8.63)	81.01
V64/IMC	RV64IMC	25.491.171	1.864.006	266.80 (2.78)	95.54
SWERV	RV64IMC	25.491.171	-	309.94 (2.05)	82.25

Table 6.1: Execution metrics of the fibonacci30 benchmark for the VADL and SWERV ISS at different compilation targets.

decode the smaller instruction format. And if such a compressed instruction is detected it is processed in a separate switch statement that only lists the cases for the compressed instructions. The *SWERV-ISS* on the other hand handles the differences between compressed and standard instructions in the decoding stage and produces intermediate data holder objects that are afterwards processed in a single switch statement. The approach of the VADL ISS therefore presents a shortcut which results in the runtime advantage if the majority of instructions are stored in the compressed format. The fraction of compressed instructions for different optimization levels can be found in Table 6.3 and lies between 73% and 90% for the fibonacci30 benchmark.

Table 6.2 lists the fibonacci30 run times for VADL ISS with matching compilation targets. The difference between V32/IM with RV32IM and V32/IMC with RV32IMC is about 6.3%, which can be attributed to the additional (unused) instructions for the V32/IMC ISS, which increase the code size and add additional complexity for instruction decoding and additional cases that have to be considered for the switch based interpreter.

Table 6.3 shows a comparison for the fibonacci30 benchmark at different optimization levels. This illustrates the very significant reduction of executed instructions and branches between O0 and O2/O3, which also has a strong impact on the overall runtime for the benchmarks. The simulation performance on the other hand is only slightly reduced and has a minor tendency to run slower on the more optimized code in this example.

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

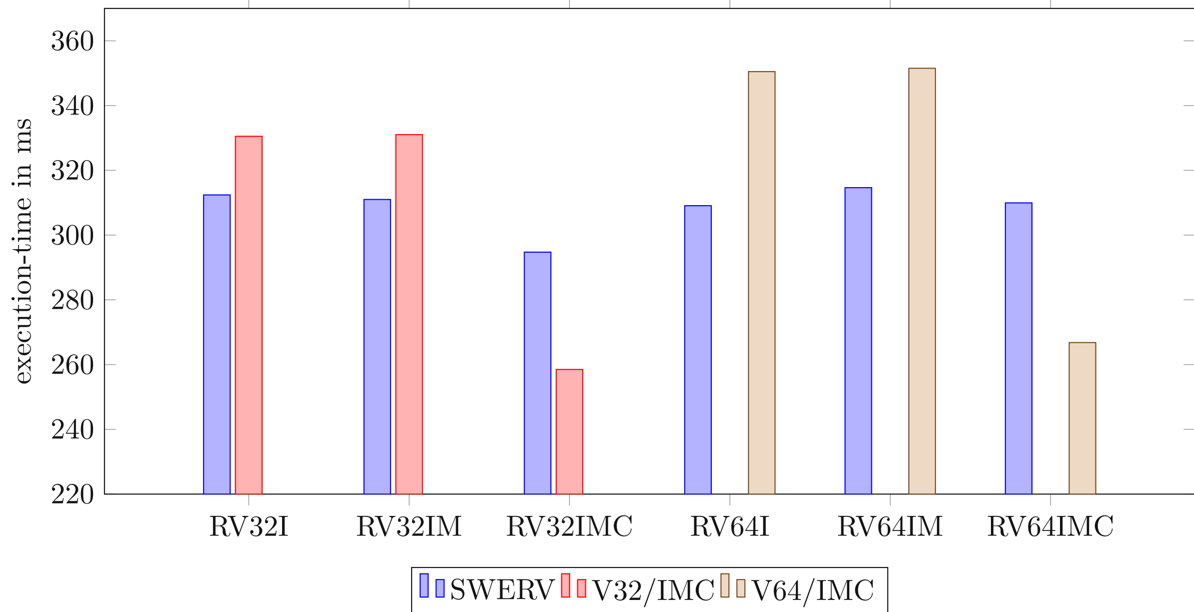

Figure 6.1: Execution times of the fibonacci30 benchmark for the VADL and SWERV ISS at different compilation targets.

ISS	***Compilation Target***	***Runtime Mean/stddev (ms)***	***MIPS***
V32/I	RV32I	305.47 (3.41)	82.77
V32/IM	RV32IM	310.15 (4.79)	81.52
V32/IMC	RV32IMC	258.50 (3.17)	97.81
V64/I	RV64I	340.12 (3.13)	74.95
V64/IM	RV64IM	352.58 (2.47)	72.30
V64/IMC	RV64IMC	266.80 (2.78)	95.54

Table 6.2: VADL ISS execution times of the fibonacci30 benchmark with the VADL ISS when the specification matches the compilation targets.

GCC O-level	***#instructions***	***#branches***	***compressed instructions***	***Runtime Mean/stddev (ms)***	***MIPS***
O0	57.889.794	8.077.651	79.07%	549.89 (2.95)	105.28
O1	40.388.295	6.731.381	90.00%	365.67 (2.29)	110.45
O2	22.236.228	1.695.036	73.06%	227.17 (2.56)	97.88
O3	25.284.993	1.864.006	76.52%	257.35 (2.73)	98.25

Table 6.3: Metrics for the fibonacci30 benchmark for the V32/IMC ISS with the compilation target RV32IMC at different GCC O-levels.

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

6.3.3 MiBench

Table 6.4 shows the results for the 32 and 64 bit VADL ISS and *SWERV-ISS* for the MiBench-Network (dijkstra) and MiBench-Automotive (qsort, basicmath, bitcount) at different compilation targets. Note that the shown compilation target matches the VADL ISS variant again and therefore should represent the best case for the achievable runtime in this setting. The first observation in this table is that the qsort and basicmath benchmarks significantly require more instructions when the compilation targets doesn't support multiplication and division instructions. The best overall run times are again achieved for the VADL ISS when the compilation target supports the compressed instruction set, with a best mark for bitcount at 64 bit of nearly 114 MIPS. A general tendency can also be seen that more instructions per second can be executed for the bitcount benchmark than for qsort. The SWERV-ISS shows overall much better runtime results compared to the VADL ISS in these benchmarks. But the difference strongly varies between the benchmarks and while there is only a slight advantage in the case of the dijkstra benchmark, there is a very significant difference for the qsort, basicmath and bitcount benchmarks.

6.3.4 Dhrystone

The Dhrystone benchmark stopped after only a 1/10 of the workload when it was run with a 64 bit VADL ISS, which is probably due to a still undetected error in the definition of the 64 bit instruction set. But the benchmark did also stop to early in a fifth of all test cases when executed with the *SWERV-ISS*, which could also indicate that there is still some problem with the compiled code itself. Table 6.5 shows the results for the 32 bit VADL ISS, which did run to the end without any problems.

6.3.5 Host/Guest Performance Counters

Tables 6.6 and 6.7 are showing the performance counters for the host and guest for different benchmarks. Very noteworthy is the branch miss rate of the host system, which is rarely more than 1% for any of the used benchmarks. The factor of executed instruction between the host and guest system is for the VADL ISS for 32 and 64 bit relatively constant between 1:130 - 1:147. The ratio results for the *SWERV-ISS* fluctuate between 1:54 for the 32 bitcount benchmark up to 1:154 for 64 bit dijkstra, which also fits the previously measured run times quite well.

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

Test	*ISS*	*Comp. Target*	*#instr.*	*#branches*	*Runtime mean/stddev* (s)	*MIPS*
dijkstra	V32/I	RV32I	151.812.617	28.255.647	1.642 (.0173)	92.47
dijkstra	V32/IM	RV32IM	144.891.854	26.571.656	1.534 (.0077)	94.47
dijkstra	V32/IMC	RV32IMC	144.871.854	26.561.656	1.511 (.0092)	95.86
dijkstra	V64/I	RV64I	166.069.289	29.790.469	1.947 (.0212)	85.32
dijkstra	V64/IM	RV64IM	153.840.634	26.571.591	1.823 (.0107)	84.40
dijkstra	V64/IMC	R64IMC	153.820.634	26.571.591	1.667 (.0083)	92.27
dijkstra	SWERV	RV32I	151.812.775	-	1.529 (.0124)	99.30
dijkstra	SWERV	RV32IM	144.891.938	-	1.483 (.0058)	97.67
dijkstra	SWERV	RV32IMC	144.871.938	-	1.463 (.0067)	99.01
dijkstra	SWERV	RV64I	166.069.441	-	1.589 (.0070)	104.50
dijkstra	SWERV	RV64IM	153.840.789	-	1.516 (.0068)	101.51
dijkstra	SWERV	R64IMC	153.820.789	-	1.507 (.0055)	102.09
qsort	V32/I	RV32I	1.934.054.534	414.685.842	23.951 (.1448)	80.75
qsort	V32/IM	RV32IM	528.335.940	232.805.060	7.573 (.0096)	69.77
qsort	V32/IMC	RV32IMC	528.035.940	66.961.856	6.244 (.0266)	84.57
qsort	V64/I	RV64I	1.021.805.969	29.790.469	12.60 (.0713)	81.12
qsort	V64/IM	RV64IM	345.544.559	47.823.387	5.103 (.0206)	67.72
qsort	V64/IMC	R64IMC	345.244.567	47.823.387	4.348 (.0112)	79.40
qsort	SWERV	RV32I	1.934.054.559	-	13.538 (.0869)	142.85
qsort	SWERV	RV32IM	528.335.965	-	4.774 (.01555)	110.66
qsort	SWERV	RV32IMC	528.035.965	-	4.592 (.0117)	115.00
qsort	SWERV	RV64I	1.021.538.149	-	6.927 (.0236)	147,48
qsort	SWERV	RV64IM	344.994.141	-	3.234 (.0124)	106.68
qsort	SWERV	R64IMC	344.694.146	-	3.139 (.0167)	109.78
basicmath	V32/I	RV32I	2.907.582.013	569.626.153	36.816 (.1802)	78.98
basicmath	V32/IM	RV32IM	1.299.929.784	146.366.327	17.858 (.0553)	72.79
basicmath	V32/IMC	RV32IMC	1.299.926.165	146.279.665	16.046 (.0292)	81.01
basicmath	V64/I	RV64I	1.975.560.716	392.622.667	25.486 (.2852)	77.52
basicmath	V64/IM	RV64IM	1.054.329.536	139.437.694	14.570 (.0387)	72.36
basicmath	V64/IMC	R64IMC	1.054.329.610	139.356.455	12.914 (.0490)	81.64
basicmath	SWERV	RV32I	2.907.582.109	-	21.327 (.0551)	136.33
basicmath	SWERV	RV32IM	1.299.929.880	-	12.331 (.0449)	105.41
basicmath	SWERV	RV32IMC	1.299.926.239	-	12.144 (.0470)	107.03
basicmath	SWERV	RV64I	1.929.808.639	-	13.879 (.0342)	139.04
basicmath	SWERV	RV64IM	1.026.175.360	-	8.816 (.0392)	116.40
basicmath	SWERV	R64IMC	1.026.175.358	-	8.745 (.0831)	117.35
bitcount	V32/I	RV32I	495.398.061	79.118.552	5.203 (.0415)	95.21
bitcount	V32/IM	RV32IM	495.336.922	79.103.948	5.192 (.0705)	95.401
bitcount	V32/IMC	RV32IMC	495.336.902	79.103.910	4.455 (.0158)	111.19
bitcount	V64/I	RV64I	496.076.207	8.035.1071	5.595 (.0367)	88.67
bitcount	V64/IM	RV64IM	495.883.526	80.304.176	5.679 (.0503)	87.32
bitcount	V64/IMC	R64IMC	495.879.173	80.303.553	4.353 (.0260)	113.91
bitcount	SWERV	RV32I	495.491.291	-	2.671 (.0106)	185.47
bitcount	SWERV	RV32IM	495.412.791	-	2.678 (.0324)	184.98
bitcount	SWERV	RV32IMC	495.416.611	-	2.667 (.0421)	185.77
bitcount	SWERV	RV64I	495.447.823	-	2.686 (.0093)	184.46
bitcount	SWERV	RV64IM	495.407.890	-	2.679 (.0088)	184.93
bitcount	SWERV	R64IMC	495.402.299	-	2.684 (.0115)	184.59

Table 6.4: Execution metrics of MiBench (network and automotive) for different variants of the VADL ISS and *SWERV-ISS* at different compilation targets.

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

ISS	*Compilation Target*	*#instructions*	*#branches*	*Runtime Mean/stddev* (s)	*MIPS*
V32/IMC	RV32I	1.110.208.785	80.034.687	14.08 (.126)	78.84
V32/IMC	RV32IM	955.150.092	60.020.424	11.97 (.037)	79.78
V32/IMC	RV32IMC	1.060.232.461	66.625.537	10.81 (.067)	98.12

Table 6.5: Dhrystone benchmark results for the 32 bit VADL ISS at different compilation targets.

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

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

ISS	*Benchmark*	*#H-instr*	*#H-cycles*	*#H-branches*	*#H-branch misses*	*#H-branch miss rate [%]*	*#G-instr*	*#G-branches*	*H:G instr*
V32/IMC	fibonacci30	3.383.107.165	1.106.346.161	553.294.969	2.308.650	0.417	25.284.993	1.864.006	1 : 134
V32/IMC	MiBench-dijkstra	21.152.704.409	6.533.914.182	3.527.288.076	2.287.956	0.065	144.871.918	26.561.662	1 : 146
V32/IMC	MiBench-qsort	73.464.377.231	27.503.165.663	12.455.150.072	77.464.315	0,622	528.035.940	66.961.856	1 : 139
V32/IMC	MiBench-basicmath	182.733.919.944	70.410.163.563	30.884.483.657	313.561.945	1.015	1.299.926.165	146.279.665	1 : 140
V32/IMC	MiBench-bitcount	64.525.153.052	19.775.457.225	10.372.516.674	9.091.520	0.087	495.336.902	79.103.910	1 : 130
SWERV	fibonacci30	3.864.594.484	1.294.360.247	755.219.136	3.683.183	0.487	25.284.993	-	1 : 153
SWERV	MiBench-dijkstra	18.441.449.424	6.397.056.483	3.577.729.816	20.505.592	0.573	144.871.837	-	1 : 127
SWERV	MiBench-qsort	44.304.693.578	20.141.808.765	8.835.958.408	104.605.387	1.183	528.035.864	-	1 : 83
SWERV	MiBench-basicmath	126.855.551.909	53.420.922.266	25.392.532.181	282.696.233	1.113	1.299.926.138	-	1 : 98
SWERV	MiBench-bitcount	26.978.649.500	11.722.479.432	5.406.495.604	11.355.361	0.210	495.416.926	-	1 : 54

Table 6.6: Host and Guest Performance counters for the RV32IMC compilation target.

ISS	*Benchmark*	*#H-instr*	*#H-cycles*	*#H-branches*	*#H-branch misses*	*#H-branch miss rate [%]*	*#G-instr*	*#G-branches*	*H:G instr*
V64/IMC	fibonacci30	3.457.445.496	1.147.196.619	555.065.695	3.408.444	0.614	25.491.171	1.864.006	1 : 135
V64/IMC	MiBench-dijkstra	22.544.402.869	7.354.406.949	3.775.759.210	2.847.372	0,075	153.820.699	26.571.598	1 : 147
V64/IMC	MiBench-qsort	48.599.542.888	19.227.237.152	8.177.183.641	78.485.319	0,959	345.244.567	47.823.387	1 : 141
V64/IMC	MiBench-basicmath	147.628.316.062	57.714.647.918	24.716.433.397	211.173.456	0,854	1.054.329.610	139.356.455	1 : 140
V64/IMC	MiBench-bitcount	64.676.547.183	19.346.542.971	10.424.654.141	13.658.208	0.131	495.882.041	80.304.135	1 : 130
SWERV	fibonacci30	3.925.665.654	1.334.949.244	775.954.057	5.175.516	0,667	25.491.171	-	1 : 154
SWERV	MiBench-dijkstra	18.914.308.746	6.629.622.361	3.684.547.721	20.028.241	0.544	153.820.701	-	1 : 123
SWERV	MiBench-qsort	32.810.907.242	14.280.260.617	6.515.148.316	61.646.298	0,946	344.694.058	-	1 : 95
SWERV	MiBench-basicmath	96.729.824.885	38.802.383.092	19.313.157.123	174.673.022	0.904	1. 026.175.270	-	1 : 94
SWERV	MiBench-bitcount	28.011.072.319	11.875.068.958	5.618.398.086	12.480.473	0.222	495.403.265	-	1 : 57

Table 6.7: Host and Guest Performance counters for the RV64IMC compilation target.

Benchmark	*#retired instructions*	*#cycles*	*CPI*	*Runtime Mean/stddev* (s)	*MIPS*
fibonacci30	25.284.992	29.446.016	0.8587	1.33 (.0134)	19.00
MiBench-dijkstra	151.812.616	225.779.727	0.6724	8.63 (.0489)	17.59
MiBench-qsort	1.934.054.533	2.778.740.269	0.6960	119.01 (.6823)	16.25
MiBench-basicmath	2.907.582.012	4.081.497.606	0.7123	174.34 (.4301)	16.68
MiBench-bitcount	495.611.102	653.928.510	0.7579	24.56 (.0834)	20.18
dhrystone	1.110.211.275	1.335.284.726	0.8314	68.43 (.5879)	16.22

Table 6.8: Benchmarks for a CAS 5-stage pipeline for compilation target RV32I.

6.4 CAS Performance

Table 6.8 shows the measurements for various benchmarks for a 5-stage RISC-V pipeline and RV32I compilation target. AS expected the performance is much lower than the ISS at approximately one fifth of previously seen run times for the same compilation target. The performance counter values can be found in the Table 6.9 and also show a cache miss ratio of less than 1% for the host. But the ratio between the executed host and guest instruction is also much worse and between 1:656 and 1:808. The simulated inorder pipeline has a cycle-per-instruction value between 67% and 83%.

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

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

Benchmark	***#H-instr***	***#H-cycles***	***#H-branches***	***#H-branch misses***	***#H-branch miss rate [%]***	***#G-instr***	***#G-cycles***	***H:G instr***
fibonacci30	16.595.443.137	5.785.056.136	3.339.033.309	30.804.436	0.922	25.284.992	29.446.016	1 : 656
MiBench-dijkstra	122.616.470.576	38.150.363.234	24.656.220.547	63.451.697	0.257	151.812.680	225.779.806	1 : 808
MiBench-qsort	1.490.346.469.475	562.977.336.231	300.821.054.711	2.634.354.344	0.875	1.934.007.685	2.778.684.739	1 : 770
MiBench-basicmath	2.171.687.293.799	770.214.013.202	441.047.218.948	3.372.700.338	0.764	2.907.582.012	4.081.497.606	1 : 747
MiBench-bitcount	367.419.766.810	109.624.060.300	74.574.885.481	69.747.542	0,094	509.377.291	672.299.766	1 : 721

Table 6.9: Host and Guest Performance counters for the V32/I/P5 CAS and compilation target RV32I.

CHAPTER 7

Future Work

While a working ISS and CAS can already be generated from a given VADL specification, there are certainly limitations to the current implementation and some desirable features are still missing. Possible new features and improvements for the current implementation can broadly be categorized into the aspect of functionality, performance, usability and test enhancements and will be discussed throughout the remaining chapter.

7.1 Functionality

Floating point support was recently added to the VADL language and has yet to be incorporated in both simulator generators. The ability to model **custom exceptions and traps** is a key feature that is necessary to model a **privileged user mode** and will be added in a future VADL version, but also probably require at least some changes in the simulator generator implementation.

Instruction decoding and **instruction fetching** is already supported for many real world cases, but will also need some extensions to handle complex ISAs like X86 and ARM or compressed instruction encodings for VLIW architectures.

While **delegation of system calls** to the host systems has already been implemented for core functionalities, there will certainly be some additional efforts required in the future, to execute a wider range of software applications. An additional support for a **full system emulation** is also desirable to increase the CAS accuracy, but will certainly require some efforts, even when build on top of an existing framework.

The CAS generator in its current form does only support *inorder pipeline architectures* and needs to be adapted accordingly for **superscalar** and **out-of-order execution**. The VADL language will also have to be adapted in this regard together with the support for additional and extended MiA components like **memory**, **caches** and **branch predictors**, among many others.

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

7.2 Performance

While the performance of the emitted ISS is already comparable in some benchmarks to the handwritten SWERV-ISS, it has not yet been optimized due to time constraints. The state of the art describes promising optimizations like **direct threaded code** or **decoded instructions caches**. Such techniques can probably be incorporated into the generated simulator without spending a lot of effort, but then also problem of self-modifying-code has to be considered. And while there are in general a few reasonable approaches to this problem, there is, as so often, no clear best solution for every situation as described by this survey [Kep09].

But optimizations can also be achieved by more efficient **instruction decoding schemes** or even by small improvements of the generated C++ code. Removing additional debug information and checks from the standard simulator and using the additional interactive mode for the creation of instruction traces and debug outputs is also a possible option for further improvements. Various types of **compilation techniques** can certainly achieve much better run times, but will also results in much higher implementation efforts than most other possible measures.

The evaluated CAS for the 5-stage RISC-V pipeline showed less satisfying results than the ISS variants of about 16 to 20 MIPS on current hardware, but this runtime performance will even decrease further with the simulation of additional and more complex MiA components and the support for superscalar and out-of-order execution. While the primary goal should certainly be to implement the missing functionalities first, it will also be a critical first step to reassess the performance situation of the CAS right after to be able to efficiently steer necessary optimization efforts.

While some optimizations for the ISS like faster instruction decoding can also profit the CAS, this will probably not be enough to make a significant improvement of the execution times.

Further and more significant improvements can probably be achieved by removing not strictly necessary details from the simulation. Take for example the currently simulated pipeline registers, which are often not directly modelled in manually written CAS. These simulators typically use a data structure to represent the instruction with all decoded values and only passes a pointer trough the pipeline. This reduces the overhead and can also be combined with the support for threaded code or decoded instruction caches. The need to buffer values in a secondary pipeline register could also be removed from the current implementation by processing the pipeline stages in reverse order or by sorting the executed operations accordingly.

The cycle-count accurate techniques like described by [LDT13] can also be an interesting option.

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

7.3 Usability

The user feedback for detected problems during the generation of a simulator is currently very limited. It is only possible to write warnings to the console or stop the generation with an error message by throwing an exception. While most errors should be caught earlier by checks and directly displayed to the user in the *Eclipse environment*, this will not be the possible in all complex cases and also not for bugs. Providing a better feedback mechanism for the user in the *Eclipse environment*, which also can be enriched with context-relevant information, could help to distinguish between VADL specification errors and implementation bugs.

Assume for example that the transformation from AST to CRM fails due to an unknown symbol, then it would be interesting to also get a list of available symbols with the error message or even a representation of the AST itself. Providing context-relevant information to a user through a log file or even graphically on the user interface also improves the reproducibility of bugs, because this information can easily be attached to a bug report.

7.4 Testing

The debugging of the generated simulators were without a doubt the most time consuming and continuously occurring tasks during the creation of the simulator generator project. The implementation of the interactive mode significantly improved the efficiency of this debugging process, but it still remains a manual and time consuming effort.

A often recurring debugging pattern is to find the first deviation of a faulty instruction trace. This is typically achieved by comparing the trace to a correct version that can be obtained from a reference simulator. The next step is to inspect this code location with its current register and memory values and then continuously going back in the trace to find the position at which some of these values have been computed incorrectly.

But the interactive mode currently does not keep the necessary information to **backtrack** to previously executed instructions and therefore can only be started multiple times and directed by the user to stop the instruction execution at an earlier stage. Additionally this debugging method is not practicable for bigger programs which would run minutes or even hours. Adding a limited backtracking ability for even a few hundred instructions in the interactive mode could therefore significantly improve the debugging experience. And this is especially the case when this feature would be combined with a **snapshot mechanisms** to restore the memory and register content for a specific point in the program execution, which would be an especially useful improvement for large programs.

While some debugging efforts will probably always be inevitable, it is an overall better approach to avoid this situations whenever possible. This can be achieved for example by extensive **unit testing** and should include tests for each instruction that is being defined in VADL, by using for example the feature to export test signatures.

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

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

CHAPTER 8

Conclusion

This work introduced a flexible code generation framework written in *Xtend*, which was the basis to create an ISS and CAS generator for the VADL language. This code generation framework can include static resources and dynamically created content and therefore provides a maintainable and adaptable solution for the generation of a C++ based simulator.

An ISS simulator is created from the ISA and ABI description of a VADL specification. This description includes the general components of a processor like memory and register files, but also defines the instruction behaviour and encoding. The ISS generator uses this information to create a C++ based simulator with additional build files.

The CAS simulator works in a similar fashion, but also needs an additional specification of the MiA, which describes the pipeline model and additional behaviour for instruction fetching, decoding and more advanced mechanisms like forwarding. One especially noteworthy implementation detail is the partitioning of ISA instructions over the pipeline stages. This is performed by using the definition of access privileges to processor components and by assigning computations statements to pipeline stages in the VADL specification.

Both simulator generators can reuse code for common tasks like instruction decoding, sign extension and the creation of debug outputs, to name a few. Program arguments can also be passed to the executed program. And both simulators also support the same debug mechanisms, which include performance counters, various kinds of verbose debug outputs, the printing of an instruction trace and an interactive mode.

A VADL specification for RISC-V modules for the base integer function set (I), multiplication and division (M), control and status registers (Zicsr) and compressed instructions (C) has been created during the runtime of this project for 32 and 64 bits. While this work only contributed a portion of the overall efforts for these RISC-V VADL module

specifications, it certainly provided substantial efforts for compliance testing and the resolution of specification errors.

Multiple ISSs have been created from this VADL specification and were tested in various benchmarks against the handwritten *SWERV* simulator. The handwritten simulator showed overall better results, but the generated VADL ISS variants could show competitive results in at least some of the benchmarks Most of these benchmarks could also be executed with 64 bits, but failed in very few cases, which is probably attributed to minor specification errors in the 64 bit VADL instruction definitions. We are quite confident that these problems can be resolved quickly when 64 bit compliance tests are available to thoroughly test these instruction definitions.

A CAS has also been created from a VADL specification with a 5-stage pipeline model for RV32I and tested against the same benchmarks as the ISS variant. While the executed benchmarks delivered only a mediocre performance, this can certainly be improved in a future version.

And while the current implementation does not yet support features like floating point arithmetic or out-of-order execution, it can still be seen as a good starting point to add all these desired features in a future version.

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

List of Figures

3.1 Basic Simulator Generator Overview 24

4.1 ISS Simulator Class Diagram 38
4.2 ISS Interactive Mode 45

5.1 CAS Simulator Class Diagram 48
5.2 High Level View of the Pipeline Model 49
5.3 Pipeline Register used to buffer values between Pipeline Stages 53
5.4 Pipeline Stage Code Layout 56
5.5 CAS Component Access for 6-Stage Pipeline Architecture 66
5.6 CAS Interactive Mode 72
5.7 CAS Interactive Mode listing the Code Fragment of the LW Instruction for the EX Pipeline Stage 73

6.1 Execution times of the fibonacci30 benchmark for the VADL and SWERV ISS at different compilation targets. 79

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

List of Tables

6.1 Execution metrics of the fibonacci30 benchmark for the VADL and SWERV ISS at different compilation targets. 78
6.2 VADL ISS execution times of the fibonacci30 benchmark with the VADL ISS when the specification matches the compilation targets. 79
6.3 Metrics for the fibonacci30 benchmark for the V32/IMC ISS with the compilation target RV32IMC at different GCC O-levels. 79
6.4 Execution metrics of MiBench (network and automotive) for different variants of the VADL ISS and *SWERV-ISS* at different compilation targets. 81
6.5 Dhrystone benchmark results for the 32 bit VADL ISS at different compilation targets. 82
6.6 Host and Guest Performance counters for the RV32IMC compilation target. 83
6.7 Host and Guest Performance counters for the RV64IMC compilation target. 83
6.8 Benchmarks for a CAS 5-stage pipeline for compilation target RV32I. . . 84
6.9 Host and Guest Performance counters for the V32/I/P5 CAS and compilation target RV32I. 85

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

Acronyms

ABI Application Binary Interface. v, vi, 3, 29, 33, 57, 65, 89

ADL Architecture Description Language. 6, 18, 21

AST abstract syntax tree. 39, 59, 68, 88

CAS Cycle-Accurate Simulator. v, vi, 1, 3–5, 8, 11, 18, 20, 22, 27, 28, 44, 47, 48, 50, 54, 56–58, 61, 65, 66, 71–74, 76, 77, 84–87, 89–92

CISC Complex Instruction Set Computer. 2

DBT Dynamic Binary Translation. 18

DSP Digital Signal Processor. 19

ELF Executable and Linkable Format. 3–5, 26, 29

FPGA Field Programmable Gate Arrays. 19

HDL Hardware Description Language. 6

ILP Instruction-Level Parallelism. 8

ISA Instruction Set Architecture. v, vi, 1, 3, 4, 7–9, 11–13, 28, 30, 37–39, 49, 50, 57, 65, 74, 76, 86, 89

ISAC Instruction Set Architecture C. 19

ISDL Instruction Set Description Language. 7–9

ISS Instruction Set Simulator. v, vi, 1, 3–5, 7, 11, 12, 15, 20, 22, 27, 28, 37, 38, 41, 44–47, 50, 54, 56, 57, 63, 71, 72, 74–82, 84, 86, 87, 89–92

ITTAGE Indirect Target TAgged GEometric Length Predictor. 17

JIT Just-In-Time. 18

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

LISA Language for Instruction Set Architecture. 2, 9–11

LLVM Low Level Virtual Machine. 17

MARSS Micro Architectural and System Simulator. 15, 16, 20

MiA Micro Architecture. vi, 4, 7, 12, 47, 49, 50, 55, 57, 65, 70, 86, 87, 89

MIMOLA Machine Independent Microprogramming Language. 10

MIPS Millions of Instructions per Second. 18

MMX Multi Media Extension. 20

NOP no operation. 14, 51, 57

PDL Processor Description Language. iv, vi, 2, 4, 6–12

QEMU Quick Emulator. 15, 18, 20

RADL Retargetable Architecture Description Language. 11

RAW Read-After-Write. 61

RISC Reduced Instruction Set Computer. 2, 4

RTL Register Transfer Level. 7–10

SBT Static Binary Translation. 17, 18

SOC System-on-a-Chip. 2, 8

VADL Vienna Architecture Description Language. iv–vi, 2–4, 22–26, 30, 33, 37–39, 41–52, 55–63, 65–70, 74, 76–82, 86, 88–92

VLIW Very Long Instruction Word. 2, 14, 17, 19, 20, 33, 86

VM Virtual Machine. 16

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

Bibliography

[ALE02] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[APF+19] Alasdair Armstrong, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, Peter Sewell, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E Gray, et al. ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS. 2019.

[BARA04] M. Bartholomeu, R. Azevedo, S. Rigo, and G. Araujo. Optimizations for compiled simulation using instruction type information. In *16th Symposium on Computer Architecture and High Performance Computing*, pages 74–81, 2004.

[Bed90] Robert Bedichek. Some efficient architecture simulation techniques, winter 1990 USENIX conference, 1990.

[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.

[Bel73] James R Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[Bel05] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[BFKR09] Florian Brandner, Andreas Fellnhofer, Andreas Krall, and David Riegler. Fast and accurate simulation using the LLVM compiler framework. In *Proceedings of the 1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, RAPIDO*, volume 9, pages 1–6, 2009.

[BHK13] Florian Brandner, R. Nigel Horspool, and Andreas Krall. DSP instruction set simulation. In Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems*, pages 945–974. Springer, 2013.

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

[BM] Jeremy Bennett and Lee Moore. RISC-V Compliance Task Group. https://github.com/riscv/riscv-compliance. [Online; accessed 22-November-2020].

[C+95] TIS Committee et al. Tool interface standard (TIS) executable and linking format (ELF) specification version 1.2, 1995.

[CEG07] Kevin Casey, M Anton Ertl, and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(6):37–es, 2007.

[CLSL02] Harold W Cain, Kevin M Lepak, Brandon A Schwartz, and Mikko H Lipasti. Precise and accurate processor simulation. In *Workshop on Computer Architecture Evaluation using Commercial Workloads, HPCA*, volume 8, 2002.

[Dew75] Robert BK Dewar. Indirect threaded code. *Communications of the ACM*, 18(6):330–331, 1975.

[DGGL17] Amanieu D'Antras, Cosmin Gorgovan, Jim Garside, and Mikel Luján. Low overhead dynamic binary translation on ARM. *SIGPLAN Not.*, 52(6):333–346, June 2017.

[dhr] dhrystone. https://github.com/Keith-S-Thompson/dhrystone.git. [Online; accessed 21-November-2020].

[EG01] M Anton Ertl and David Gregg. The behavior of efficient virtual machine interpreters on modern architectures. In *European Conference on Parallel Processing*, pages 403–413. Springer, 2001.

[ELFa] ELFIO. https://github.com/serge1/ELFIO. [Online; accessed 30-September-2020].

[elfb] ELFIO-hello world example. https://cirosantilli.com/elf-hello-world. [Online; accessed 30-September-2020].

[elfc] ELFIO-manual. http://elfio.sourceforge.net/elfio.pdf. [Online; accessed 30-September-2020].

[Ert02] Martin Anton Ertl. *Threaded code variations and optimizations:(extended version)*. Citeseer, 2002.

[FKH07] Stefan Farfeleder, Andreas Krall, and Nigel Horspool. Ultra fast cycle-accurate compiled emulation of inorder pipelined architectures. *Journal of Systems Architecture*, 53(8):501–510, 2007.

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

[FMP13] N. Fournel, L. Michel, and F. Pétrot. Automated generation of efficient instruction decoders for instruction set simulators. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 739–746, 2013.

[Fra08] Björn Franke. Fast cycle-approximate instruction set simulation. In *Proceedings of the 11th international workshop on Software & compilers for embedded systems*, pages 69–78, 2008.

[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.

[GM19] Roberto Giorgi and Gianfranco Mariotti. WebRISC-V: a web-based education-oriented RISC-V pipeline simulation environment. In *Proceedings of the Workshop on Computer Architecture Education*, pages 1–6, 2019.

[gnu] RISC-V GNU Compiler Toolchain. `https://github.com/riscv/riscv-gnu-toolchain`. [Online; accessed 21-November-2020].

[HAG08] Yonghyun Hwang, Samar Abdi, and Daniel Gajski. Cycle-Approximate Retargetable Performance Estimation at the Transaction Level. DATE '08, page 3–8, New York, NY, USA, 2008. Association for Computing Machinery.

[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.

[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 (ASIPs) 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.

[HP17] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017.

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

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

[hyp] hyperfine. `https://github.com/sharkdp/hyperfine`. [Online; accessed 21-November-2020].

[ILB16] Berkin Ilbeyi, Derek Lockhart, and Christopher Batten. Pydgin for RISC-V: A fast and productive instruction-set simulator. In *Extended Abstract for Presentation at the 3rd RISC-V Workshop*, 2016.

[KA01] Rajeev Krishna and Todd Austin. Efficient software decoder design. *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, 2001.

[Kep09] D. Keppel. How to Detect Self-Modifying Code During Instruction-Set Simulation. 2009.

[Kli81] Paul Klint. Interpretation techniques. *Software: Practice and Experience*, 11(9):963–973, 1981.

[LCWT11] Chen-Kang Lo, Li-Chun Chen, Meng-Huan Wu, and Ren-Song Tsay. Cycle-count-accurate processor modeling for fast and accurate system-level simulation. In *2011 Design, Automation & Test in Europe*, pages 1–6. IEEE, 2011.

[LDT13] Pei-Chia Patty Lin, Evason Du, and Ren-Song Tsay. A fast and accurate instruction-oriented processor simulation approach. In *2013 International Symposium onVLSI Design, Automation, and Test (VLSI-DAT)*, pages 1–5. IEEE, 2013.

[LIB15] Derek Lockhart, Berkin Ilbeyi, and Christopher Batten. Pydgin: generating fast instruction set simulators from simple architecture descriptions with meta-tracing jit compilers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267. IEEE, 2015.

[LPAA+20] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris,

Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. The gem5 Simulator: Version 20.0+, 2020.

[MAF91] Christopher Mills, Stanley C Ahalt, and Jim Fowler. Compiled instruction set simulation. *Software: Practice and Experience*, 21(8):877–889, 1991.

[Mag97] Peter S Magnusson. Efficient instruction cache simulation and execution profiling with a threaded-code interpreter. In *Proceedings of the 29th conference on Winter simulation*, pages 1093–1100, 1997.

[mar] MARSS-RISCV: Micro-architectural system simulator for RISC-V. `https://github.com/bucaps/marss-riscv`. [Online; accessed 26-September-2020].

[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.

[MCE+02] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

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

[mib] mibench. `http://vhosts.eecs.umich.edu/mibench/`. [Online; accessed 21-November-2020].

[OT16] K. Okuda and H. Takeyama. Decision tree generation for decoding irregular instructions. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1592–1597, 2016.

[PACG11] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSS: A full system simulator for multicore x86 CPUs. In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1050–1055, 2011.

[PH17] David A. Patterson and John L. Hennessy. Computer organization and design RISC-V edition: The hardware software interface. 2017.

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

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

[PKH+11] Zdeněk Prikryl, Jakub Kroustek, Tomáš Hruška, Dušan Kolář, Karel Masařík, and Adam Husár. Design and simulation of high performance parallel architectures using the ISAC language. *GSTF Journal on Computing*, 1(2), 2011.

[PMHH09] Zdenek Prikryl, Karel Masarík, Tomáš Hruška, and Adam Husár. Fast cycle-accurate interpreted simulation. In *2009 10th International Workshop on Microprocessor Test and Verification*, pages 9–14. IEEE, 2009.

[Pro95] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. POPL '95, page 322–332, New York, NY, USA, 1995. Association for Computing Machinery.

[Rah] Joseph Rahmeh. Western digital's open source RISC-V SweRV instruction set simulator. `https://github.com/westerndigitalcorporation/swerv-ISS`. [Online; accessed 12-September-2020].

[Raz87] Rami R Razouk. The use of Petri nets for modeling pipelined processors. 1987.

[RCS09] Tahiry Ratsiambahotra, Hugues Cassé, and Pascal Sainrat. A versatile generator of instruction set simulators and disassemblers. In *2009 International Symposium on Performance Evaluation of Computer & Telecommunication Systems*, volume 41, pages 65–72. IEEE, 2009.

[RD05] Mehrdad Reshadi and Nikil Dutt. Generic pipelined processor modeling and high performance cycle-accurate simulator generation. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*, DATE '05, page 786–791, USA, 2005. IEEE Computer Society.

[risa] Listing of RISC-V cores and SOCs. `https://github.com/riscv/riscv-cores-list`. [Online; accessed 3-October-2020].

[risb] RISC-V. `https://riscv.org/`. [Online; accessed 3-October-2020].

[risc] RISC-V instruction set manual: ISA specification (volumen I). `https://github.com/riscv/riscv-isa-manual/releases/download/draft-20200727-8088ba4/riscv-spec.pdf`. [Online; accessed 20-September-2020].

[risd] RISC-V instruction set manual: Privileged architecture (volumen II). `https://github.com/riscv/riscv-isa-manual/releases/`

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

download/draft-20200727-8088ba4/riscv-privileged.pdf. [Online; accessed 03-October-2020].

[RMD03] M. Reshadi, P. Mishra, and N. Dutt. Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*, pages 758–763, 2003.

[ros] Rosetta. https://www.apple.com/rosetta/index.html. [Online; accessed 22-September-2020].

[RSS15] Erven Rohou, Bharath Narasimha Swamy, and André Seznec. Branch prediction and the performance of interpreters: Don't trust folklore. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, page 103–114, USA, 2015. IEEE Computer Society.

[SCHY12] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. LLBT: An LLVM-based static binary translator. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '12, page 51–60, New York, NY, USA, 2012. Association for Computing Machinery.

[Sez11] André Seznec. A 64-Kbytes ITTAGE indirect branch predictor. In *JWAC-2: Championship Branch Prediction*, San Jose, United States, June 2011. JILP.

[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.

[SKS12] Alexander Sepp, Julian Kranz, and Axel Simon. GDSL: A generic decoder specification language for interpreting machine language. *Electronic Notes in Theoretical Computer Science*, 289:53 – 64, 2012. Third Workshop on Tools for Automatic Program Analysis (TAPAS' 2012).

[SM06] A. Seznec and P. Michaud. A case for (partially) TAgged GEometric history length branch prediction. *Journal of Instruction-level Parallelism - JILP*, 8, 01 2006.

[The01] Henrik Theiling. Generating decision trees for decoding binaries. In *Proceedings of the 2001 ACM SIGPLAN workshop on Optimization of middleware and distributed systems*, pages 112–120, 2001.

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

[Wag15] Harry Wagstaff. From high level architecture descriptions to fast instruction set simulators. 2015.

[xtea] Xtend. `https://www.eclipse.org/xtend/`. [Online; accessed 27-September-2020].

[xteb] Xtext. `https://www.eclipse.org/Xtext/`. [Online; accessed 27-September-2020].

[YKS+05] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and comparing prevailing simulation techniques. In *11th International Symposium on High-Performance Computer Architecture*, pages 266–277, 2005.

[YL06] J. J. Yi and D. J. Lilja. Simulation of computer architectures: simulators, benchmarks, methodologies, and recommendations. *IEEE Transactions on Computers*, 55(3):268–280, 2006.

[You07] M. T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *2007 IEEE International Symposium on Performance Analysis of Systems Software*, pages 23–34, 2007.

[YWZW16] Lei Yang, Lei Wang, Xing Zhang, and DongLin Wang. An approach to build cycle accurate full system VLIW simulation platform. *Simulation Modelling Practice and Theory*, 67:14 – 28, 2016.

[ZPM96] Vojin Zivojnovic, Stefan Pees, and Heinrich Meyr. LISA-machine description language and generic machine model for HW/SW co-design. In *VLSI Signal Processing, IX*, pages 127–136. IEEE, 1996.

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