



Fast Simulation and Performance Estimation for Vector Co-Processors

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Daniel Schloms, Bsc

Registration Number 11701253

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr.-Ing. Dipl.-Ing. Daniel Müller-Gritschneider

Vienna, 1st December, 2025

Daniel Schloms

Daniel Müller-Gritschneider



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

Erklärung zur Verfassung der Arbeit

Daniel Schloms, 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.

Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang "Overview of Used AI Tools" habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 1. Dezember 2025

Daniel Schloms



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

Acknowledgements

I want to thank Daniel Müller-Gritschneider and Parker Jones for their supervision and guidance. As this thesis builds upon Parkers work, his insights proved to be invaluable. Additionally, I want to highlight the people maintaining the various tools used in this thesis, namely Philipp van Kempen, Conrad Foik, Johannes Geier, and Johannes Kappes, who helped me out a lot with various questions, feature requests, and energy drinks. Lastly, special thanks go to my family, Karin, Bernd, and Jana, and my girlfriend, Katharina, for their continuous support in stressful times.



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

Kurzfassung

Applikationsspezifische Prozessoren haben verschiedenste Anforderungen an Größe, Effizienz, Geschwindigkeit, Features, etc. Um Hardware für einen speziellen Bereich zu entwickeln, wird der Prozess der Design Space Exploration (DSE) verwendet, um schnell verschiedene Entwicklungsentscheidungen auszutesten und gewünschte Eigenschaften zu erfüllen. RISC-V ist eine Befehlssatzarchitektur, dessen modularer Aufbau mit verschiedenen Erweiterungen und Konfigurationen für diesen Zweck gut geeignet ist, es existiert jedoch Verbesserungspotential im Bereich der Software für die Entwicklung. Es fehlen Tools zwischen funktionalen Befehlssatzsimulatoren (in Folge Instruction Set Simulator (ISS)), welche zwar schnell sind, jedoch keine Auskunft über die Performanz der Hardware geben, und Hardwaresimulatoren, die sowohl Verhalten, als auch notwendige Taktzyklen wiedergeben, aber dafür langsam simulieren und eine detaillierte Hardwarebeschreibung benötigen. Für funktionale ISSs gibt es Methoden, die eine Abschätzung von Taktzyklen ermöglichen, und in dieser Arbeit wird der Ansatz untersucht, Instruktionen mit zusätzlichen Informationen zu versehen, um gleichzeitig die Funktionalität, sowie das zeitliche Verhalten mithilfe eines vereinfachten Prozessormodells zu simulieren. ETISS ist ein ISS, für welchen dieser Ansatz bereits erfolgreich für skalare Prozessorkerne eingesetzt wurde. Dabei wurde CorePerfDSL verwendet, eine Sprache, die dazu dient, das zeitliche Verhalten eines Prozessorkerns zu modellieren. Nun gibt es jedoch viele Programme, welche von Single Instruction, Multiple Data (SIMD) Instruktionen profitieren können, z.B. im Bereich des maschinellen Lernens, somit wäre die Abschätzung der Performanz von Vektor Co-Prozessoren ein logischer nächster Schritt. Daher wurde für diese Arbeit ein CorePerfDSL Modell eines skalaren RISC-V CV32E40P Kerns mit einer Vektoreinheit implementiert. Anschließend wurden sowohl Simulationsgeschwindigkeiten, als auch die Güte der Schätzung der Taktzyklen untersucht, indem resultierenden Werte mit einem anderen gängigen funktionalen ISS, sowie einer vollen Hardwaresimulation verglichen wurden. Es wurde gezeigt, dass es mithilfe von ETISS und CorePerfDSL (mit Änderungen) möglich ist, SIMD Instruktionen schnell zu simulieren und deren zeitliches Verhalten mit geringem Fehler zu schätzen.



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

Abstract

Application-specific processors have vastly different requirements regarding area, efficiency, speed, and features, among other properties. To develop hardware that is tailored for specific purposes, a process called Design Space Exploration (DSE) is used to examine different design decisions and satisfy those particular demands. RISC-V is an instruction set architecture whose modular concept with different extensions and architectural configurations lends itself well for that purpose, however, there are still potential improvements to be made for tools used in the development process. A gap has been identified between behavioral-only Instruction Set Simulator (ISS), which are fast, but do not consider any hardware timing information, and full Register-Transfer Level (RTL) simulators, which require a detailed hardware implementation and are quite slow. There are some techniques for behavioral ISS that enable cycle-approximate timing estimations, with this thesis focussing on annotating instructions with additional information to simulate both behavior and a simplified timing model of a processor core in parallel. ETISS is an ISS for which this approach has been successfully used for single scalar cores by using CorePerfDSL, a language designed to model the timing behavior of a core. However, as many applications can greatly benefit from Single Instruction, Multiple Data (SIMD) instructions, such as machine learning tasks, estimating performance of vector co-processors would be a logical next step. Hence, for this thesis, a CorePerfDSL model for a scalar RISC-V CV32E40P scalar core with a Vector Processing Unit (VPU) has been implemented, and simulation results with regard to speed and accuracy were compared with a popular behavioral-only ISS and an RTL simulator. It was shown that, with some adaptations, performance estimation for SIMD instructions can be done in a fast and accurate manner using ETISS and CorePerfDSL.



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

Contents

Kurzfassung	vii
Abstract	ix
Contents	xi
List of Acronyms	1
List of Terms	3
1 Introduction	5
1.1 Motivation	5
1.2 Research Questions & Goals	6
1.3 Contributions	7
1.3.1 Functional Simulation	7
1.3.2 Timing Model	7
1.3.3 Accuracy- and Simulation Speed Measurements	8
2 Background & Related Work	9
2.1 RISC-V	9
2.1.1 V Extension	10
2.2 Instruction Set Simulators	12
2.3 Toolchain	13
2.3.1 ETISS	13
2.3.2 Verilator	13
2.3.3 CoreDSL & M2-ISA-R	14
2.3.4 CorePerfDSL & M2-ISA-R-Perf	14
2.4 Vicuna	16
2.5 Initial State	16
2.5.1 CoreDSL Model	16
2.5.2 ETISS & Softvector	17
2.5.3 Performance Model	17
2.6 Related Work	17
2.6.1 Spike	17
	xi

2.6.2	RISC-V VP++	17
2.6.3	QEMU	18
2.6.4	The Argument for ETISS with Performance Estimation	18
3	Performance Simulation Extensions for Vector Coprocessors	21
3.1	Functional Simulation	21
3.1.1	Overview	21
3.1.2	Proposed Architecture	21
3.1.3	Implementation	23
3.1.4	Potential Improvements	24
3.2	Performance Estimation	25
3.2.1	Overview	25
3.2.2	Reference	26
3.2.3	Analysis with Regard to CorePerfDSL	27
3.2.3.1	Parallel Pipelines	27
3.2.3.2	Parameterized Pipeline Architectures	30
3.2.3.3	Further Pipelining of Vector Instructions	34
3.2.3.4	Dependencies Between Parallel Pipelines	37
4	Experimental Results	41
4.1	Functional Simulation Results	41
4.2	Performance Comparison Workflow	41
4.2.1	Traces	43
4.2.2	Measurements	43
4.3	Performance Estimation Results	44
4.3.1	Configurations	44
4.3.2	Simple Tests	44
4.3.3	ML Inference Workloads	48
4.3.4	Discussion	49
4.4	Simulation Speed	50
4.4.1	ETISS JIT-Compiler	51
4.4.2	Results	51
4.4.3	Discussion	55
4.4.4	Performance of Vector Instructions	55
5	Conclusion and Future Work	57
5.1	Conclusion	57
5.2	Future Work	57
	List of Figures	61
	List of Tables	63
	Bibliography	65

List of Acronyms

- ADL** Architecture Description Language. 5, 6
- ALU** Arithmetic Logic Unit. 14–16, 27, 31, 34, 44
- AOT** Ahead-Of-Time. 12
- ASL** Arm Architecture Specification Language. 5
- CISC** Complex Instruction Set Computer. 9
- CPI** Cycles Per Instruction. 6, 8, 18, 42–44, 49, 50, 57
- CPU** Central Processing Unit. 9, 12
- CSR** Control and Status Register. 15, 16
- DAG** Directed Acyclic Graph. 38, 39
- DFS** Depth-First Search. 38
- DSE** Design Space Exploration. vii, ix, 5, 17, 19, 26, 57
- DSL** Domain-Specific Language. 6, 14, 19
- FPU** Floating-point Unit. 16
- HDL** Hardware Description Language. 5, 14, 26, 42
- ISA** Instruction Set Architecture. 5–7, 9, 12–14, 17–19, 41, 57
- ISS** Instruction Set Simulator. vii, ix, 5, 6, 11–13, 17, 18, 22, 57
- JIT** Just-In-Time. 12–14, 41, 58
- LSU** Load-Store Unit. 16, 33, 34

- MIPS** Mega/Million Instructions Per Second. 17–19, 51
- ML** Machine Learning. 19, 48–51
- PC** Program Counter. 13, 43
- RISC** Reduced Instruction Set Computer. 9
- RTL** Register-Transfer Level. ix, 6, 8, 9, 11, 13, 14, 26, 29, 30, 34, 35, 41–44, 48, 55, 57
- SIMD** Single Instruction, Multiple Data. vii, ix, 6, 10, 25, 41, 48
- SISD** Single Instruction, Single Data. 6
- VP** Virtual Prototype. 17, 18
- VPU** Vector Processing Unit. ix, 6, 7, 12, 16, 26–30, 37, 41, 57, 58, 61
- WCET** Worst-Case Execution Time. 16
- XIF** Core-V Extension Interface. 17

List of Terms

EEW The effective element width for a vector instruction, can differ from SEW. 10, 12

EMUL The effective multiplicity of a vector register group, can differ from LMUL. 10

LMUL The multiplicity of a vector register group. 3, 10, 25, 34–36, 41, 44

NOP Null operation - an instruction that does nothing. 44

SEW The selected element width for a vector instruction. 3, 10, 23, 25, 41, 44, 55

VLEN The length of a vector register in bits. 8, 10–12, 30–33, 44, 49, 51–55, 58, 63



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

Introduction

1.1 Motivation

In the realm of hardware development, many tasks, such as profiling, verification, and Design Space Exploration (DSE), pose a particular challenge. To achieve meaningful results, a design has to be either manufactured outright, or be tested with a multitude of simulation tools, which are sometimes not ideal for the task at hand. As a specific example, which is also the focus of this thesis, one can look at DSE for digital logic, processor cores that implement some Instruction Set Architecture (ISA) in particular. Here, the problem can be viewed from two sides. On one hand, there is the ISA and the full specification surrounding it. In this instance, DSE can mean adding instructions or changing their behavior, adding new registers, etc. Such a specification can be implemented in an Architecture Description Language (ADL), which often comes with additional tooling for simulation or verification. Examples of ADLs are the Arm Architecture Specification Language (ASL) [1], the Sail ISA specification language [2], or CoreDSL [3], which is used in this thesis. To perform functional simulations of code running on some ISA, one can either build a bespoke Instruction Set Simulator (ISS) for that architecture specifically (e.g. RISC-V VP++ [4]), or use the specification to generate code for simulation, as is the case with the Extendable Translating Instruction Set Simulator (ETISS) [5], which will be simulator used for this thesis. As the name suggests, ETISS is a simulator for ISAs that requires the architecture to be provided by the user (in C++), which can be generated from CoreDSL with a tool called M2-ISA-R [6].

On the other hand, there is the actual implementation in hardware. Apart from correctly adhering to the ISA specification, the design can be changed to achieve many different metrics, such as speed, minimal chip area, or efficiency, which often contradict each other. Digital logic is generally implemented in a Hardware Description Language (HDL), with popular examples being VHDL, Verilog, and SystemVerilog. For those languages, there are a host of simulation tools, both open-source and proprietary, as well as using different

approaches, such as compiled vs. interpreter-based. For full and detailed verification, testing, and profiling, there is no way around using such an RTL simulator or even synthesizing the design. A popular tool for this purpose, and one which is used in this thesis, is Verilator [7], which can translate RTL designs into C++ or SystemC to be simulated via a user-provided test bench. However, initially, not all metrics might be of equal importance, and exploring a more abstracted view at the pipeline level can be more interesting to get an idea about Cycles Per Instruction (CPI) or the impact of hazards. Here lies a problem in that, even then, either a fully detailed implementation, e.g. in VHDL, is required, or the simulation tool needs to have hardware implementation details baked into it. Not only does this lead to longer simulation times, but it also contradicts the need to be able to rapidly change the architecture to explore different higher-level decisions, such as the layout of the processor pipeline, or lane- and register widths.

1.2 Research Questions & Goals

The missing middle ground between full hardware simulation and functional simulation has led to the creation of a performance estimation plugin for ETISS, among other tools that aim to solve the same problem. This plugin is used in conjunction with a Domain-Specific Language (DSL) called CorePerfDSL and an accompanying code generation tool (M2-ISA-R-Perf) to estimate performance metrics based on a simplified microarchitectural description of a processor core [8] [9]. While CoreDSL describes behavior, CorePerfDSL is used to model hardware at the microarchitectural level. This can be done by specifying stages, which contain so-called *microactions*, that can be chained together to form a pipeline. Microactions model delays within a stage, and connector models can be used to further describe dependencies between instructions. A CorePerfDSL model can then be used to generate C++ code with M2-ISA-R-Perf for ETISS through the aforementioned performance estimation plugin. During simulation, this plugin uses the timing model to calculate pipeline stage timings for each instruction. The resulting metrics are not perfectly cycle-accurate compared to a Register-Transfer Level (RTL) simulation of the respective core, but the goal is to accept a reasonably small deviation in cycles to gain a significant simulation speedup. Currently, the described toolchain can be used to simulate and profile programs for scalar processor cores with Single Instruction, Single Data (SISD) instructions, with multi-issue and parallel pipeline capabilities for CorePerfDSL in active development. However, since many ISAs feature Single Instruction, Multiple Data (SIMD) capabilities, ISSs and performance estimators should reflect this. Therefore, the task of this thesis is to answer the following research questions:

- How do we integrate efficient functional simulation of RISC-V vector instructions using the Architecture Description Language (ADL) CoreDSL?
- How to model the performance of Vector Processing Units (VPUs) using the capabilities of the micro-architecture description language CorePerfDSL?

- Which possible extensions to CorePerfDSL are required to achieve more accurate VPU performance models?

The approach on VPU modeling shall be evaluated with the RISC-V Vicuna 2.0 co-processor [10] [11]. It currently implements a subset ISA of the RISC-V vector extension, namely `Zve32f`, and the `Zvfh` extension for vector half-precision floating-point operations. To achieve this, the following implementation tasks have to be done:

- Extend ETISS to be able to simulate at least `Zve32f` instructions (potentially `Zvfh` as well)
- Model Vicuna 2.0 as accurately as possible in CorePerfDSL
- Find a way to profile and compare programs on ETISS and Verilator

1.3 Contributions

1.3.1 Functional Simulation

The base requirement for performance estimation is that the behavior of relevant ISAs can be simulated in the first place. To achieve this, a previous `RV32IMACFD` CoreDSL model with minimal vector instruction support was extended to include all instructions specified in `Zve32d`. This model, however, defers the behavioral implementation to external functions, as vector instructions pose new challenges that are often more suited for general purpose programming languages. A basic architecture for the actual implementation of RISC-V vector instructions, that aims to be easily extensible, is proposed and implemented within a C++ library called *Softvector*. As with the CoreDSL model, an initial version of this library was already available with support for a small subset of vector instructions, and was subsequently extended to include all of `Zve32d`, and reworked with the newly proposed architecture. To test for correctness, an existing testing suite was used [12].

1.3.2 Timing Model

To model parallel VPUs, new behaviors and requirements not found in scalar cores were identified, for which new solution strategies and modeling techniques, including extensions to CorePerfDSL, have been proposed. This was done by examining the Vicuna 2.0 vector co-processor, and a number of the aforementioned modeling techniques were then implemented for initial validation. The final model is based on an existing model of a `CV32E40P` 4-stage RISC-V core, which is one of the possible scalar cores in the actual Vicuna 2.0 project, the other being the Ibex core.

1.3.3 Accuracy- and Simulation Speed Measurements

To show that performance estimation - with the specified toolchain in particular - is a viable strategy for our use case, two key metrics need to be observed. Firstly, it needs to be shown that a CorePerfDSL model can deliver accurate cycle estimations for non-trivial programs. This was done by comparing the results of performance estimation with a cycle-accurate RTL simulation for a selection of machine learning tasks. It was shown that, for the chosen programs, the performance estimation plugin for ETISS can deliver accurate cycle predictions. This applies to arbitrary hardware configurations, as estimated instruction times, and thus CPI, followed respective changes reported by Verilator simulations for different hardware parameters. Secondly, any inaccuracies need to be justified with a large enough simulation speedup. Using the previously mentioned benchmarks, results show that a large speedup can be achieved, especially for larger hardware configurations (e.g. longer VLEN).

Background & Related Work

2.1 RISC-V

RISC-V is an ISA that was originally developed at the University of California, Berkeley. As the name suggests, it specifies a Reduced Instruction Set Computer (RISC) architecture, as opposed to a Complex Instruction Set Computer (CISC) architecture, such as x86. In his doctoral thesis [13], which also serves as a design document for the ISA, Waterman justifies the development of RISC-V. The first issue with existing ISAs is that many of them are proprietary, especially more widespread ones, for instance x86 and ARM. This complicates academic use due to restrictions on sharing RTL implementations and creates a higher barrier to entry for anyone wanting to use these technologies commercially. RISC-V, in contrast, "is free and open with a permissive license for use by anyone in all types of implementations"[14]. This is reflected in, for example, the Instruction Set Manual [15] and the Vector Extension Specification [16], as both repositories contain a Creative Commons Attribution 4.0 International license. Secondly, they consider alternative ISAs too inflexible, unnecessarily complex, or find that they contain technical issues. In the aforementioned thesis, Waterman also lists specific shortcomings of other ISAs that were considered for adoption. What separates RISC-V in this regard is the focus on modularity. Only a small base ISA is mandated, the choice being between RV32I, RV64I, and RV32E, while most instructions are grouped into optional extensions. Combined with an effort to enable custom extensions, RISC-V cores can be tailored for specific applications to a very high degree, from efficient, small area embedded systems to desktop CPUs.

2.1.1 V Extension

The Vector (V) extension adds Single Instruction, Multiple Data (SIMD) capabilities to a RISC-V core through vector instructions, which operate on vector registers [16]. A vector register can hold a variable number of elements, depending on its size (VLEN) and the element width, e.g. a 128-bit register can hold 16 8-bit values or 2 64-bit values. Legal element widths are determined by the implemented sub-extensions, which will be discussed further down. Additionally, multiple vector registers can be combined into a register group, or only a partial register can be used. Most instructions can be performed either masked or unmasked, where a masked instruction will interpret `v0` as an array of bits determining which elements to skip. Each vector instruction operates according to a vector configuration (`vtype`) set by the user at runtime via special configuration-setting instructions. This configuration includes:

- Vector Length: The number of vector elements to be processed.
- Multiplicity (LMUL): If equal or greater than 1, how many registers to use, if fractional, how much of a register to use.
- Selected Element Width (SEW): The width of vector elements in bit. Together, SEW, LMUL, and VLEN imply the maximum number of processed elements in a single vector instruction, with $V_{MAX} = LMUL * \frac{VLEN}{SEW}$.
- Mask policy: Whether masked elements in the target register group must retain their old value, or if they can be overwritten.
- Tail policy: If the vector length is less than the maximal number of elements for the current configuration, all additional elements at the end are called *tail elements*. As with the mask policy, the tail policy determines whether tail elements retain their old values, or if they can be overwritten.

Both LMUL and SEW are usually the actual values used, but some instructions use a different *effective* value for one or more register groups. For instance, a widening arithmetic instruction, such as `vwadd.vv`, has an Effective Element Width (EEW) of $2 * SEW$ for its target registers, and, since the result element width is doubled, but the amount of result elements is the same as the number of source operands, the Effective LMUL (EMUL) of the target register group is also $2 * LMUL$.

As of version 1.0, the vector extension can be further divided into subsets for embedded processors, and it offers some additional extensions on its own. An overview of the different subsets and their dependencies can be found in Figure 2.1. An arrow denotes that the origin extension depends upon the sub-extension it points to. To illustrate this, the full V extension requires that the Zve64d, D, Zve64f, Zve32f, F, Zve64x, Zve32x, and Zicsr extensions are implemented, as well as a minimum vector register length of 128 (Zv1128b extension). If one were to add Zvfh, Zfhmin would also be required. Note that Zve* denotes a proper sub-extension that may specify new

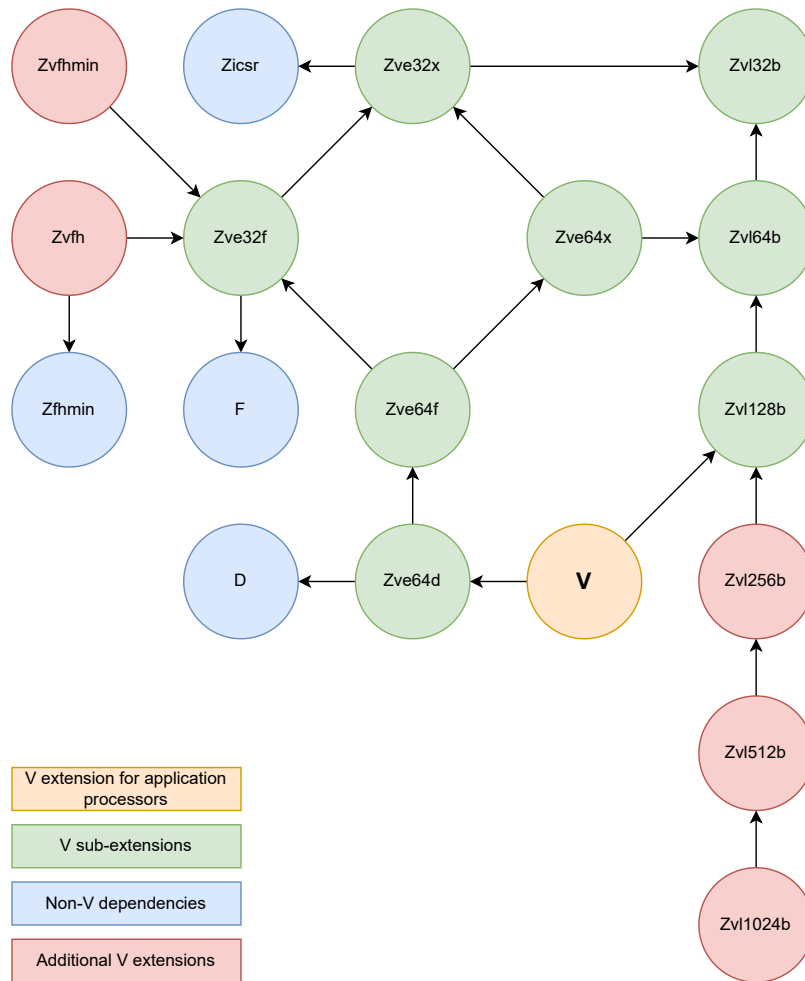


Figure 2.1: Dependency graph for the V extension

instructions or support for a wider range of element widths, while $Zvl*$ only describes the minimal VLEN, with the V and $Zve*$ extensions requiring some minimal VLEN. Implementations can offer a larger VLEN than required, with 1024-bit vector registers ($Zvl1024b$) being the largest ones named in the specification. Both $Zvfmin$ and Zvf are additional extensions that add vector instructions for half-precision floating-point values. Additional constraints for vector extensions can be found in Tables 2.1 and 2.2, where FP32 and FP64 denote support for single-precision and double-precision vector floating point instructions, respectively.

Regarding adaption of the vector extension, popular ISSs such as QEMU or Spike support the latest version (v1.0) of the specification. Open-source RTL implementations of full

Extension	Minimum VLEN	Supported EEW	FP32	FP64
Zve32x	32	8, 16, 32	No	No
Zve32f	32	8, 16, 32	Yes	No
Zve64x	64	8, 16, 32, 64	No	No
Zve64f	64	8, 16, 32, 64	Yes	No
Zve64d	64	8, 16, 32, 64	Yes	Yes

Table 2.1: Brief description of zve^* sub-extensions, taken from [16].

Extension	Minimum VLEN
Zvl32b	32
Zve64b	64
Zve128b	128
Zve256b	256
Zve512b	512
Zve1024b	1024

Table 2.2: Minimal VLEN for each zvl^* sub-extension, taken from [16].

cores or VPU with vector support include Ara [17], T1 [18], Boom/Ocelot [19] [20], and Vicuna 2.0 [11], among others. There also exist some commercially available CPUs that implement the vector extension, notably the Xuantie C910 [21] and SpacemiT K1 [22] processors.

2.2 Instruction Set Simulators

Instruction Set Simulators (ISSs) are tools that take in a program compiled for a given target ISA and execute them on a (usually different) host ISA. This functionality has a wide range of applications, from validation and evaluation of a new ISA or extensions thereof, running programs on multiple architectures without requiring the actual hardware, to emulating video game consoles. ISSs employ different techniques to achieve this, each which certain advantages over others. Overall, we can differentiate between two basic approaches, interpretation-based simulation, and compilation-based simulation [23]. In the first approach, instructions are fetched one by one, decoded, dispatched, and then executed, where the simulated processor state is updated. Here, implementation is comparatively easy, but fetching, decoding, and dispatching each instruction incurs a performance penalty. However, this does not necessarily mean that interpreting ISSs are always slower than compiling ones, as will be seen in Section 2.6. Compilation-based simulators on the other hand transform a target machine instruction into host machine instructions to be executed natively. This can either be done Ahead-Of-Time (AOT) (*statically compiled simulation*) or Just-In-Time (JIT) (*dynamically compiled simulation*), by translating target code into an intermediate representation, e.g. C code, first and then compiling that code for the host architecture. While the JIT approach introduces a

runtime overhead due to compilation, caching already translated and compiled instructions mitigates this issue, especially in programs where blocks of instructions are executed many times, as is the case with most programs. Some ISSs, notably ETISS and QEMU, use another standard optimization for dynamic binary translation, in that whole blocks of instructions are translated and cached at once [5] [24]. If a series of instructions, starting from some Program Counter (PC), contains no branches or jumps, that block will always be executed, barring any interrupts. Therefore, if, for the current PC, no cached block is available, all instructions up to the next control flow instruction are combined into a *translation block* and compiled together. Furthermore, these blocks can be set up in a way such that any PC within that block can be used as a branch target without recompilation. The compiled approach eliminates the need for the fetch, decode, and dispatch step completely in statically compiled simulations, while instructions in dynamically compiled simulations only have to be fetched, decoded, and translated if no translation block containing that instruction has been compiled yet, or if the respective translation block has been evacuated from the cache.

2.3 Toolchain

2.3.1 ETISS

ETISS is the main ISS of interest in this thesis [5]. The simulator itself is architecture-agnostic and needs an architecture implementation to simulate an instruction set. This architecture can be generated from CoreDSL, a language used to describe ISAs, using a tool called M2-ISA-R. As mentioned in Section 2.2, ETISS uses dynamic binary translation, where the generated architecture contains C code that is compiled Just-In-Time into blocks of instructions during runtime. This allows for high simulation speeds, as these blocks can be cached and reused. For JIT-compilation one has the option to choose between GCC, TCC, and Clang/LLVM, with GCC and TCC currently working out of the box. Differences in performance between TCC and GCC will be briefly discussed in Section 4.4.1. ETISS is also extensible with different kinds of pre-existing or user-defined plugins, e.g. to print an instruction trace, or to gather performance metrics. Lastly, it features a GDB server, which allows for debugging of simulated programs with GDB.

Comparing ETISS to alternative simulators, the defining feature and key advantage is its flexibility. Most ISSs are built for specific ISAs, and adding new architectures or instructions is often cumbersome for users. Using the CoreDSL workflow, which will be further discussed in Section 2.3.3, makes it comparatively easy to simulate arbitrary ISAs, allowing for fast testing and verification with only a modest performance penalty due to the effectiveness of dynamic binary translation.

2.3.2 Verilator

Verilator is a program used for simulating digital logic that takes in a Verilog RTL description and translates ("verilates") the design into either C++ or SystemC [7]. The

resulting files can then be used in a user-written wrapper or test bench that drives the translated model and compiled into an executable simulator. Other RTL simulation tools use a similar approach, e.g. GHDL, and compiling HDL code down to a native executable generally beats out interpreter-based simulators such as ModelSim. In fact, Verilator claims as 200-1000x speedup over interpreted simulators when also using multithreading capabilities. However, for small designs and comparatively short simulation times, interpreters might even be preferable, as compilation times have to be factored in.

Verilator has been used to simulate programs for the development of Vicuna 2.0 [11]. For many hardware configurations and programs, this is reasonably fast and also offers the choice to generate a signal trace, but when using larger designs, i.e. wider lane widths or vector registers, long-running programs come with a large simulation time.

2.3.3 CoreDSL & M2-ISA-R

CoreDSL is a Domain-Specific Language (DSL) that is designed for hardware modeling, specifically processor cores at the ISA level [25] [3]. It follows a C-style syntax and allows for the definition of cores (keyword `Core`) and instruction sets (keyword `InstructionSet`). A core can provide multiple instruction sets, which in turn can be extensions of one or more instruction sets. For example, the RISC-V core RV32IMACFDV provides the V (vector) instruction set, among others, which is built up from smaller instruction sets (see Figure 2.1). Both cores and instruction sets can define an architectural state. There, one can define registers, constants, and aliases, but also external devices or IO ports. An instruction set that extends another one will inherit its architectural state. An actual instruction can be implemented in the `instructions` section of an instruction set with its binary encoding, the assembly format, and the behavior of the instruction. Furthermore, it is possible to declare functions that can be called in the behavioral definition of an instruction. Those functions can be defined internally in CoreDSL, or marked as `extern` and treated as a black box. As the CoreDSL model is used to generate C++ code in this thesis, external functions are practical for handling more complex behaviors, such as those of vector instructions. In that case, the behavior is implemented in the previously mentioned external `Softvector` library.

M2-ISA-R is a translation tool used in conjunction with ETISS [6]. It takes in a CoreDSL description and translates it into a C++ architecture for the simulator. Furthermore, it translates instructions from CoreDSL syntax into C to be JIT-compiled during simulation.

2.3.4 CorePerfDSL & M2-ISA-R-Perf

While CoreDSL is used to describe behavior, CorePerfDSL is a DSL used to model a microarchitecture, and more specifically its timing [8] [9]. A core is generally modeled at the pipeline level, where one can define stages and chain them together to create said pipeline. Additionally, stages can contain subpipelines, which allows for parallel use of resources, such as a multiply unit and an ALU, and multiple top-level pipelines can also run in parallel. To ascribe delays to actions performed in a stage, such as using a divider,

stages can contain microactions, in the following denoted with the prefix `uA`. In simple cases, such as using an ALU, these can be simple constant delays that are added in that particular stage before being able to leave the stage. For instructions that incur dynamic delays, such as vector instructions, where the size of a vector register group can change during runtime, external models in the form of a `ResourceModel` can be used. They are written in C++ and can use information such as the contents of Control and Status Registers (CSRs) or custom trace values, such as the width of a vector load/store, to set dynamic delays. Additionally, different dependencies or hazards have to be handled. This can also be achieved by an external model, this time via a `ConnectorModel`. Instead of adding a delay to stages, they can receive and return timestamps that are set or requested in some microaction. By adding internal tracking variables and potentially trace values (such as register indices), one can now model various hazards and even branch prediction.

As a reference, the syntax for defining microactions, stages, and (parallel) pipelines, as well as subpipelined stages, can be found in Listings 2.1, 2.2, and 2.3.

```

1 // Microaction with flat delay
2 Resource Flat_Delay (<delay>)
3 Microaction uA_Flat_Delay (Flat_Delay)
4
5 // Microaction with dynamic delay from ResourceModel
6 ResourceModel dynamicDelayModel (
7     link: "path-to-resource-model.h",
8     trace : {<used trace values>}
9 )
10 Resource Dynamic_Delay (dynamicDelayModel)
11 Microaction uA_Dynamic_Delay (Dynamic_Delay)
12
13 // Microactions that get/set dependencies
14 // through a ConnectorModel
15 Resource Z_Setter (<delay or resource model>)
16 Connector {Y, Z}
17 ConnectorModel connectorModel (
18     link: "path-to-connector-model.h",
19     trace: {<used trace values>},
20     connectorIn: Y,
21     connectorOut: Z
22 )
23 Microaction uA_Get_Y (Y)
24 Microaction uA_Set_Z (Z_Setter -> Z)
25 Microaction uA_Get_Y_Set_Z (Y -> Z_Setter -> Z)

```

Listing 2.1: Syntax for defining microactions

```

1 Stage Stage_A (uA_A_1, uA_A_2, ..., uA_A_n)
2 Stage Stage_B (uA_B_1, uA_B_2, ..., uA_B_n)
3 ...
4 Stage Stage_Z (uA_Z_1, uA_Z_2, ..., uA_Z_n)

```

Listing 2.2: Syntax for defining stages

```
1 Pipeline Pipe_1 (Stage_A_1 -> Stage_B_1 -> ... -> Stage_Z_1)
2 Pipeline Pipe_2 (Stage_A_2 -> Stage_B_2 -> ... -> Stage_Z_2)
3 Pipeline Parallel_Pipe (Pipe_1 | Pipe_2)
4 Stage Subpipelined_stage (Pipe_1, Pipe_2)
```

Listing 2.3: Syntax for defining pipelines and subpipelined stages

M2-ISA-R-Perf is the corresponding translation tool that converts a CorePerfDSL model into C++ to be used in ETISS [26]. First and foremost, it generates scheduling functions for the performance estimation plugin for each instruction described in the CorePerfDSL model. During simulation of an instruction, the corresponding scheduling function is executed, and all stage timings are updated accordingly.

2.4 Vicuna

Vicuna is a Vector Processing Unit (VPU) that implements a subset of the RISC-V V extension [10]. It was designed to be timing-predictable, which benefits Worst-Case Execution Time (WCET) analysis and makes this implementation a sensible choice for real-time applications. This property also makes Vicuna a good target to examine performance estimation for vector co-processors. The original Vicuna implementation did not include vector floating point instructions, and the design has since been extended in Vicuna 2.0 to include the Zve32f and Zvfh extensions, which specify single- and half-precision vector floating point operations respectively [11]. The VPU is set up with a configurable number of unit pipelines, each housing a number of functional units, such as the ALU, multiplier, or LSU. All functional units in a unit pipeline share an unpacking- and a packing stage for reading and writing vector registers. Instructions are issued and retired in order, but two instructions that use different unit pipelines can run concurrently, assuming no dependencies between instructions. In this thesis, the dual pipeline configuration is used, which includes Pipe 1 with the vector LSU and the element unit for element-wise instructions (e.g. reduction instructions), and Pipe 2 for the ALU, multiplier, divider, FPU, and slide unit for vector slide instructions (e.g. vslideup.vx). A diagram is provided later in Figure 3.3.

2.5 Initial State

2.5.1 CoreDSL Model

A CoreDSL description is needed to generate the behavior in C++ for ETISS. Initially, the RV32IMACFD instruction set was given, with support for a minimal subset of vector instructions. Instructions were already not directly implemented in CoreDSL, but in a C++ library called Softvector. Vector CSRs were already fully implemented and did not require any changes.

2.5.2 ETISS & Softvector

The initial Softvector version matched the CoreDSL implementation, in that the instructions specified in CoreDSL were already implemented. At that point, instructions were implemented in a way that allowed for arbitrary element sizes by using byte-wise algorithms for addition, multiplication, etc. This quickly proved to be infeasible, since many instructions would require special algorithms instead of native arithmetic to be implemented, and if the need for elements larger than 64 bits arises, `__int128` or larger custom integer types should be preferred.

2.5.3 Performance Model

The starting point for the CorePerfDSL model was the CV32E40P core that was already implemented and tested in [9]. This works out nicely, as Vicuna 2.0 can use either the Ibex core or the CV32E40X core, which is a fork of CV32E40P that also implements the XIF that can be used to interface with co-processors. While single scalar pipelines, such as with CV32E40P, could already be modeled well, an obvious missing feature in the language was the possibility to describe parallel pipelines.

2.6 Related Work

In the context of this thesis, the related work consists of other tools that implement an ISS, performance estimation, or both. A key point to make here is why using a new solution is even desirable as opposed to existing solutions. Since the goal is fast DSE for processor microarchitectures, both an ISS and an accompanying performance estimator are necessary. There are a number of simulators that offer both, but even then there are trade-offs to consider, such as cycle accuracy, simulation speed, or ease of reconfigurability. A list of popular tools and key points regarding their capabilities is given below.

2.6.1 Spike

Spike is a RISC-V only behavioral ISA simulator maintained by RISC-V International [27]. It is often used as a reference for implementing both RISC-V ISSs and actual cores, and supports numerous extensions, including the full vector extension v1.0. While it is interpreter-based, it is still very fast, with Schlägl et al. reporting simulation speeds of up to around 366 MIPS on an Intel Core i7-10700 for the `zip test` benchmark [28]. Spike offers hosted simulation through the RISC-V Proxy Kernel (`pk`) extension [29], which allows for execution of system calls. As mentioned, this ISS only targets RISC-V, but offers some flexibility by allowing for easy integration of custom instructions.

2.6.2 RISC-V VP++

RISC-V VP++ is a RISC-V only Virtual Prototype (VP) that is based on the earlier RISC-V VP [30] [31]. It includes a translating ISS powering emulation of different system

configurations, with smaller configurations being used for running bare-metal programs, while larger configurations are able to run full operating systems, including a GUI. For the original VP, efforts have been made to include cycle-approximate performance estimation, which can be found in [32]. The HiFive1 board with an E31 RISC-V core was chosen as a reference target, which features a single-issue in-order 5-stage pipeline implementing the RV32IMAC architecture [33]. Performance is estimated by assigning a fixed number of execution cycles to each instruction. This approach results in accurate predictions for many benchmarks, usually below a 10% error in CPI. However, more recent attempts have proven to not only deliver better results for a fully in-order pipeline (CV32E40P), but also report comparable accuracy for the CVA6 core, which features out-of-order execution capabilities [9] [34]. Apart from the extensive set of features, another strength of this simulator is its speed. Using a number of optimization strategies for translation-based simulation, RISC-V VP++ is capable of reaching into the hundreds of MIPS on an Intel Core i7-10700, with a top speed of around 407 MIPS for the `zip test` benchmark [28]. Concerning flexibility, adding new instructions seems to demand a little more effort than Spike or ETISS, and, as with Spike, the project was designed to target RISC-V only.

2.6.3 QEMU

QEMU is a popular emulator and virtualizer with a compiling ISS for a wide range of host and target ISAs, including RISC-V with the full vector extension v1.0 [24]. It includes full system emulation, including peripherals, and allows running full operating systems. The dynamic binary translation techniques are similar to that of ETISS, with one defining difference being the immediate representation of translated code. Instead of C code, QEMU translates instructions into sequences of pre-defined *micro operations*, including handwritten translation recipes. During simulation, the *dynamic code generator*, generated from these micro operations by a tool called `dyngen`, generates a full function from multiple micro operations to translate an instruction. QEMU offers very high simulation speeds, with some results in thesis showing between 600 and 700 MIPS. Additionally, speeds for scalar programs are even better, with a longer running scalar version of the `vw` benchmark (see Section 4) pushing MIPS into the thousands. However, flexibility seems to be limited, as adding new architectures or instructions is not as straight forward as with other alternatives. Additionally, QEMU is not designed to estimate cycle counts.

2.6.4 The Argument for ETISS with Performance Estimation

Additional ISSs with cycle estimation were also discussed in [9], such as `gem5` [35], `GVSoc` [36], or `HARMLESS` [37]. Foik et al. argue that none of them fall within the targeted "sweet-spot" of relatively fast, relatively accurate, and easily reconfigurable. Apart from modeling the microarchitecture to some degree, using statistical methods is another possible strategy to predict performance. One such attempt has been made in [38] using regression analysis. A distinct disadvantage of this approach is the need for a priori

knowledge about performance for training. Furthermore, the cycle accuracies reported in that paper fall short of what can be achieved with alternative approaches.

Considering the advantages and disadvantages of the simulators mentioned before, the point for using ETISS with the performance estimation plugin can be made. While ETISS by itself is not the fastest candidate in this list, a quick test running a long scalar programs (20 iterations of the vww ML benchmark, around 2 billion instructions) still shows around 27 and 144 MIPS at the top end, with and without performance estimation respectively, on an AMD Ryzen 9 9950X 16-Core Processor. Combined with the fact that accompanying tools allow for easy retargeting of arbitrary architectures written in a simple DSL makes ETISS a sensible choice for DSE for ISAs.



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

Performance Simulation Extensions for Vector Coprocessors

3.1 Functional Simulation

3.1.1 Overview

The first goal in this thesis is for ETISS to be able to execute instructions specified in the RISC-V vector extension. For the most part, instructions are implemented in a CoreDSL model and translated to executable C code by M2-ISA-R. But while CoreDSL by itself is expressive enough to model most instructions, including vector instructions, some of them, especially those dealing with floating point values, introduce additional complexity that is more easily handled in an external C++ library. Furthermore, while the initial goal is correctness and adherence to the specification, C++ features can be leveraged to cut down on code size and potentially introduce optimizations that are not possible to express in CoreDSL. Such a library in the form of Softvector was already available, which implemented a limited number of arithmetic instructions, as well as some of the load and store instructions. Hence, the decision was made to fully implement all vector functionality in the Softvector library.

3.1.2 Proposed Architecture

Excluding configuration- and memory instructions, vector instructions represent simple operations performed over a number of elements in a vector register group. Given that there exist additional RISC-V extensions that also operate on vector registers, such as the vector cryptography extensions, we want to limit the implementation effort of single instructions and aim to apply future optimizations to many instructions at once.

Looking at the vector instruction implementation in RISC-V VP++ [4], such an approach can be found. When an instruction is decoded, a simple function containing a scalar operation, think adding or dividing two elements, is passed to a generic iterating loop. As mentioned earlier, this ISS is highly performant, despite employing interpretation-based simulation, so employing a similar technique should be a good choice for ETISS to start with. A simplified version of a possible setup is illustrated in Figure 3.1. A function is

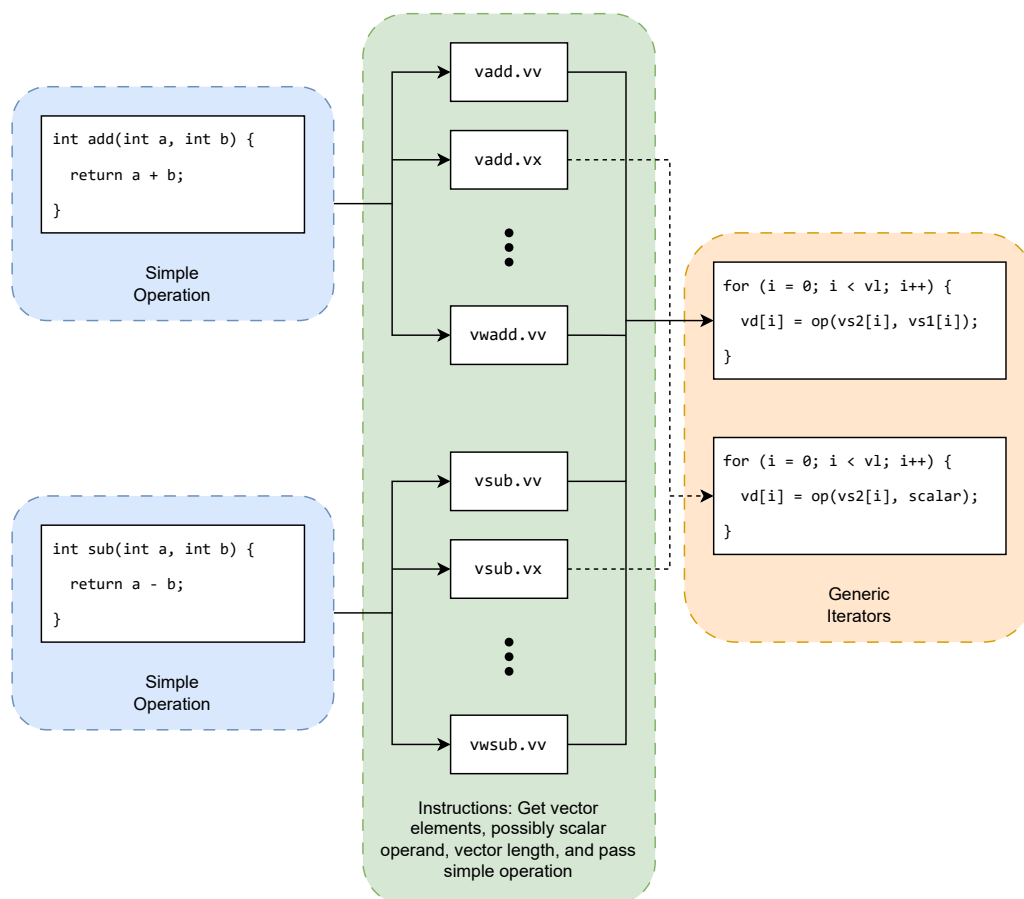


Figure 3.1: Proposed vector instruction implementation

written for each instruction, and a respective simple operation is assigned to it. This function prepares elements from the vector registers according to the current vector configuration and passes them to a fitting iteration function, together with its simple operation. In C++ this can be done in multiple ways, e.g. via function pointers or `std::function`. While each instruction still needs its own top level function to avoid unnecessary decoding, each simple operation and iterator can be used for potentially many instructions, thus cutting down on implementation size. Both simple operations

and iterators can also be generalized further to include more instructions, if so desired. For instance, in addition to regular vectorized addition, there exist `vadc` (add-with-carry) instructions, which also take in a carry bit from the mask register, i.e. the mask register is now used as data instead of masking elements. The basic single element addition can be extended to include an optional carry bit, and when iterating over the vector elements, one can differentiate between using the respective mask bit as a carry bit or a masking bit. Also, if an optimization is now found in a shared function, this instantly applies to all instructions that use it.

3.1.3 Implementation

Vector integer arithmetic instructions represent the majority of instructions (not counting the various iterations of some vector load and store instructions) and also the simplest to implement. The basic structure that is used is demonstrated graphically in Figure 3.2. In `Softvector`, each instruction has a corresponding function that takes in a pointer to the vector field, i.e. an array of bytes of length $\frac{VLEN}{8} * 32$, and all the required information to execute said instruction, including source and target registers, and the vector configuration. In a helper function, vector objects (`RVVector`) containing proper elements can then be retrieved from the vector field, where the elements are generated according to the current SEW. These vectors contain all elements of the whole vector register group, not necessarily just those of a single vector register. As described in Figure 3.1, a generic iteration function then executes the relevant innermost function (e.g. a simple addition), which is passed in as an argument, and the results are written back to an `RVVector`, which also updates the vector field.

Initially, already available vector instructions were implemented in a way that allowed for arbitrary vector element widths via byte-wise algorithms. However, this results in a considerable performance decrease and complicates implementation immensely. Additionally, the specification states that the full RISC-V vector extension shall support element widths of 8, 16, 32, and 64 bits. As a result, the computation of single vector elements was changed to use fixed-width integer types (`(u)int64_t`).

Floating point operations need to adhere to the IEEE 754-2008 standard for floating point arithmetic. This is handled by the Berkeley `SoftFloat` library, which provides all the required functionality for all vector floating point instructions [39]. Hence, the implementation of vector floating point instructions can be done in the same way as vector integer arithmetic instructions, where an element-wise function is passed to a generic iteration function. This approach is repeated for all instructions that can be similarly grouped together, such as vector mask instructions, while a few unique instructions are simply handled separately.

Vector loads and stores are special, as they need to access the underlying memory system. This is solved by passing a function for reading or writing to memory, which comes from ETISS itself. Therefore, the initial external functions are implemented within ETISS, which call the `Softvector` load and store functions while passing in a function pointer to

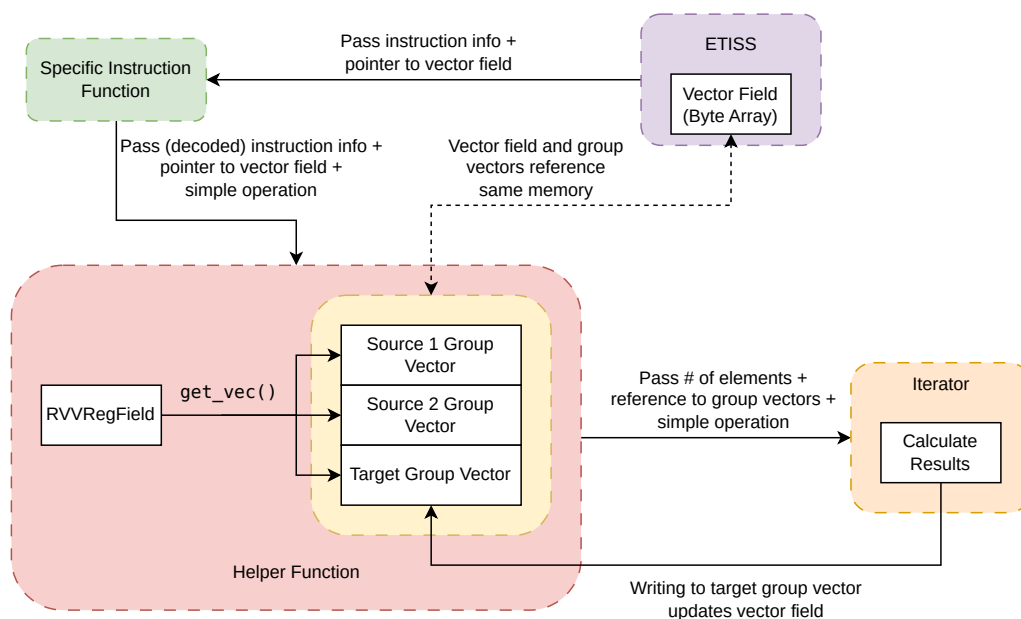


Figure 3.2: Basic execution flow in Softvector

its read and write functions. M2-ISA-R implements special attributes for this purpose. An external function annotated with `[[etiss_needs_arch]]` gets handles to the architectural state and the underlying system, where memory access is implemented.

3.1.4 Potential Improvements

As the focus thus far was to achieve a correct implementation that adheres to the specification, optimization has been neglected. This results in poor performance for vector instructions, as will be seen later in Section 4.4.4. Without citing specific numbers, profiling a vectorized benchmark shows that specific parts of Softvector have a large impact on runtime. In particular, the underlying representation of the vector registers and its elements seems to be rather inefficient as related functions, constructors, and destructors take up a significant percentage of simulation time. It stands to reason that using more basic ways to access the byte array representing the vector registers would perform much better. To test this, a single instruction in `vadd.vv` has been re-implemented in CoreDSL that uses basic arithmetic and masking to read and write vector elements. Here, vector elements are deserialized by simply shifting in bytes from the vector field, and results are written back analogously.

```

1 element = 0;
2 for (i = 0; i < nBytes; i++) {
3   element |= V[baseAddress + i] << (i * 8);
4 }

```

Further improvements could also be achieved in Softvector by also foregoing the transformation into vector objects and simply casting the vector field pointer according to SEW instead of concatenating bytes.

```

1 // Example:   vadd.vv
2 // V:        the vector field byte array
3 switch (sew_bytes)
4 {
5     case 1:
6         e1 = static_cast<uint8_t *>(V) [e1_index];
7         e2 = static_cast<uint8_t *>(V) [e2_index];
8         static_cast<uint8_t *>(V) [result_index] = e1 + e2;
9         break;
10    case 2:
11        e1 = static_cast<uint16_t *>(pV) [e1_index];
12        e2 = static_cast<uint16_t *>(pV) [e2_index];
13        static_cast<uint16_t *>(V) [result_index] = e1 + e2;
14        break;
15    ...
16 }

```

Both approaches show a massive performance improvement compared to the initial Softvector version, as seen in Section 4.4.4. However, for these simple experiments the impact of using simple operations in a generic iterator was not yet examined properly, but it seems to come with a small performance penalty compared to the examples shown here. Additionally, the actual vector computations could be improved as well. Most arithmetic instructions compute a number of independent results and could therefore benefit from parallelizing the workload, especially with multiple target registers, i.e. when LMUL is greater than 1. Possible options here are using multithreading libraries such as OpenMP, if parallelism outweighs the induced overhead, or leveraging SIMD capabilities of the simulation host. When compilers support it, using the `simd` extension introduced in the C++26 standard could also be a good choice. Vector loads and stores can also benefit from some improvements. In testing, it was observed that vector load and store instructions commonly access contiguous memory, i.e. mostly unit-strided or whole-register memory instructions are generated by the compiler, but currently, memory is read or written for each vector element separately. Therefore, a fast path with only a single read or write should be provided for access to contiguous memory. This could also be extended to strided and indexed memory instructions, however, this functionality has would need to be implemented in ETISS first.

3.2 Performance Estimation

3.2.1 Overview

By itself, ETISS only performs a behavioral simulation of an input program, so while the computational results are the same as on a real core, no timing information can be gathered. One could emulate the hardware in code, or perform a hardware simulation

with tools like Verilator or GHDL to get this information, but this contradicts the goal of rapid DSE, as both iteration and simulation times can be quite long. Instead, ETISS can use a performance estimation plugin, that simulates the timing behavior of a simplified pipeline model. As described in Section 2.3.4, a pipeline in CorePerfDSL is made up of a series of stages that contain microactions. Instructions then flow through the pipeline, taking a path specified by all the microactions the instruction executes. This behavior is implemented in *scheduling functions*, which are generated from the CorePerfDSL description for each instruction. Within it, microactions that belong to the corresponding instruction are "executed", i.e. all stages the instruction goes through are updated with a timestamp that is computed from all delays and dependencies in that stage. Also, all dependencies for subsequent instructions that can come from that instruction are set, such as register writes. With that, the timing behavior of a target program is evaluated in a simplified manner.

3.2.2 Reference

Building a timing model means working towards a model of the actual hardware description that includes information about delays, stalls, and dependencies between stages and instructions, but keeping complexity to a minimum, as not to rebuild the HDL description. Distilling down the RTL description of Vicuna 2.0, as well as taking in an already established description of the CV32E40P scalar core, results in the simplified pipeline model seen in Figure 3.3. In the actual hardware, scalar instructions will only

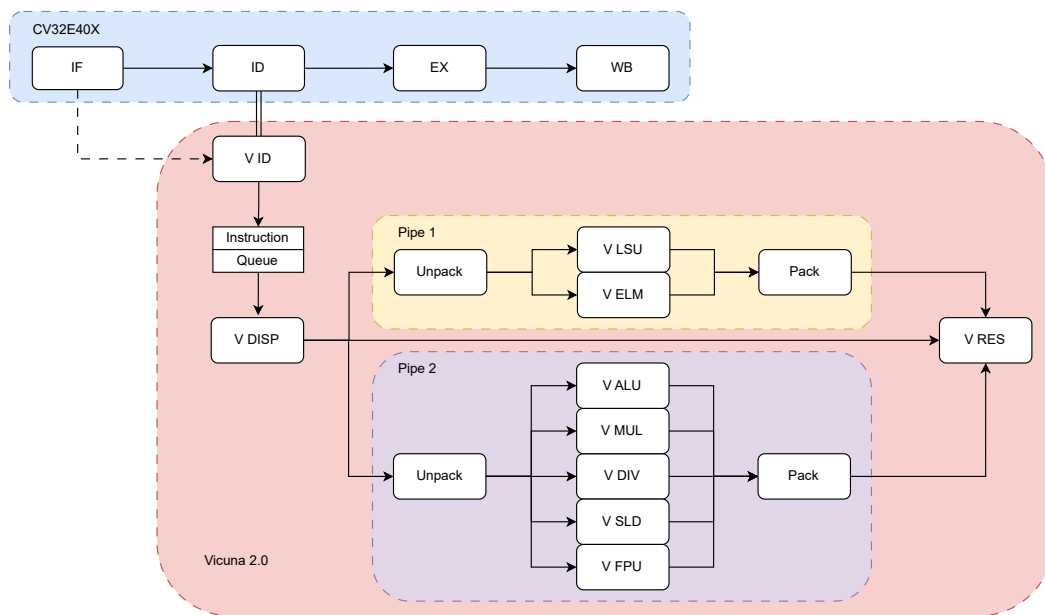


Figure 3.3: Simplified view of the CV32E40X with the Vicuna 2.0 VPU

take the scalar CV32E40X pipeline, while vector instructions go through both the scalar

and the Vicuna 2.0 pipeline. Vector instructions are offloaded in the scalar Decode stage (ID), entering the Vector Decoder (V ID) immediately, which is represented by the connection between ID and V ID. The arrow going from IF to V ID represents an adaption for CorePerfDSL, which will be discussed in Section 3.2.3.1. From there, they are put into an instruction buffer, after which they are dispatched to one of two unit pipelines (Pipe 1 and Pipe 2) housing the actual computation and memory units. From the Unpack stages, operands are fed into a functional unit, e.g. the vector ALU, and result values are re-packed in the Pack stage and written to the respective vector registers. During execution, vector loads and stores, as well as vector instructions that produce a scalar result, such as `vmv.x.s`, will stall the scalar pipeline until they complete, all other instructions do not stall the scalar core. The complexity of modeling the VPU comes from the vector unit pipelines. For one, while the whole design generally could be considered in-order, Pipe 1 and Pipe 2 can execute vector instructions in parallel. The resulting stalling logic for conflicts is extensive and thus hard to accurately describe in CorePerfDSL. Additionally, in contrast to the scalar CV32E40X core, where an instruction will only ever occupy a single stage, vector instructions are pipelined further within their respective unit pipelines. This issue is further discussed in Section 3.2.3.3. With a broad idea of the pipeline flow, the following sections deal with how the aforementioned issues can be handled in CorePerfDSL, currently unresolved problems, and some proposed solutions.

3.2.3 Analysis with Regard to CorePerfDSL

To answer the research questions regarding VPU modeling posed in Section 1.2, new requirements stemming from the addition of said hardware have to be identified, such that they can be matched against the capabilities of CorePerfDSL.

3.2.3.1 Parallel Pipelines

So far, CorePerfDSL allowed for the description of parallelism at the stage level via subpipelining. An example of how this looks like can be found in Figure 3.4. A subpipelined stage contains multiple pipelines, in this example subpipelines 1, 2, and 3. These subpipelines can consist of a single stage or multiple stages. To enable concurrent execution, a subpipelined stage has to be annotated with a capacity, referring to the depth of the optional instruction buffer, as otherwise any instruction would have to wait until the previous instruction has left the subpipelined stage. Adding an output buffer allows subpipelines to accept a new instruction on completion of the previous one, even if the following stage is not free yet. Lastly, subpipelines can block each other, where an occupied subpipeline prevents another subpipeline from accepting an instruction. An existing example of subpipelining is the CVA6 core from [9], where the subpipelining mechanism is also explained in more detail. Here, the execute stage is subpipelined, meaning that the ALU, multiplication unit, and divider unit can run in parallel. When looking at Figure 3.3, one can see that the functional units within Pipe 1 and Pipe 2 can

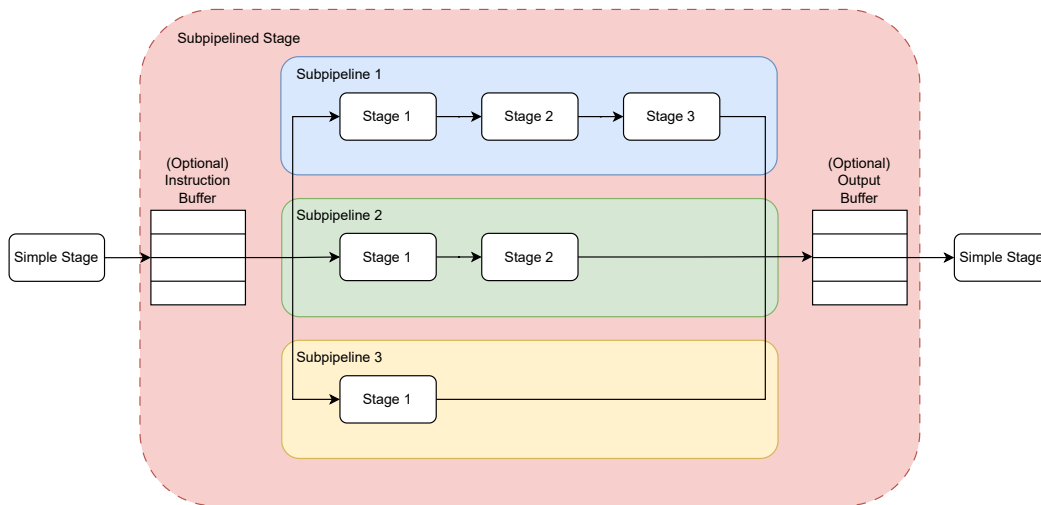


Figure 3.4: An example of a pipeline with a subpipelined stage

be implemented within a subpipelined stage. As only one unit within a unit pipeline can be active at a time anyway, buffering of instructions is not necessary.

However, the scalar core and the parallel Vicuna 2.0 co-processor cannot be modeled as subpipelines within a large stage. Vector instructions pass through both the scalar core and the VPU, but do not necessarily retire from both units at the same time, both properties contradicting the subpipelining paradigm.

Extending CorePerfDSL with Parallel Pipelines Parallel co-processors cover a wide range of functionalities and are not limited to the VPU use case. Therefore, modeling capabilities for concurrent modules should be reflected clearly in CorePerfDSL itself, and not necessarily handled with external models. A solution is proposed with a new definition of a parallel pipeline in CorePerfDSL. The constraints here are relatively simple. An instruction must be able to take both paths in a parallel pipeline, and no synchronization is forced at the end of the pipelines, i.e. a pipeline that is forked this way cannot be joined at the end. An example of parallel pipelines in CorePerfDSL is illustrated in Figure 3.5. While instructions pass through parallel pipelines independently, dependencies can still be introduced via connector models. In contrast to the previous subpipelining, instructions in different parallel pipelines do not have to retire in order, and cannot flow into a common stage at the end. This pipeline contains 2 common stages at the beginning, through which all instructions flow. The pipeline then splits into 2 parallel pipelines, and instructions can be modeled to take either one or both paths. One can also see that parallel pipelines cannot share a stage after splitting. After proposal of this idea, the language was extended by C. Foik, one of the authors of CorePerfDSL. In addition to the already mentioned features, further nesting of parallel pipelines was also introduced, with the same rules applying as before.

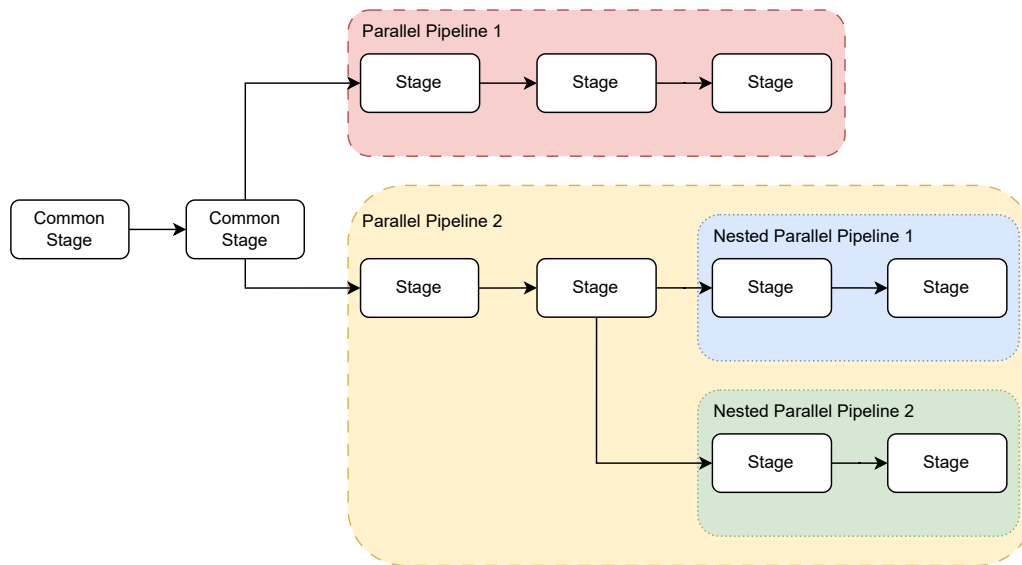


Figure 3.5: An example for parallel pipelines

Implementation of the Vicuna 2.0 Pipeline With a way to model parallel pipelines in place, one needs to figure out how to model the offloading of instructions onto a parallel unit, while also piping them through the main scalar core. As seen in Figure 3.3, vector instructions are offloaded from ID to V ID instantly. In the first iteration, this was modeled by combining ID and V ID into a single stage, where both the scalar core and the VPU were connected to the common ID stage as follows:

```

1 Pipeline Scalar (EX -> WB)
2 Pipeline Vector (InstructionQueue -> DISP -> ... )
3 Pipeline Parallel (Scalar | Vector)
4 Pipeline Full (IF -> ID -> Parallel)

```

This naive approach, however, does not work correctly with how stage timings are calculated, as an instruction can only leave ID if both the subsequent scalar and vector stage are available. If the scalar core is stalled, but the VPU can accept instructions, they will not be offloaded this way. This constraint can be avoided by letting vector instructions go into V ID directly from IF, as illustrated with the dashed arrow in Figure 3.3. This deviates from the actual RTL model, but since no actual decoding needs to be done for timing purposes, and no timing dependencies are violated, this works out well and can be written in CorePerfDSL as follows:

```

1 Pipeline Scalar (ID -> EX -> WB)
2 Pipeline Vector (V_ID -> InstructionQueue -> DISP -> ... )
3 Pipeline Parallel (Scalar | Vector)
4 Pipeline Full (IF -> Parallel)

```

During testing, and using this setup, stage timings were observed to be very accurate when compared to RTL simulations, with most errors stemming from incorrectly implemented stalls or delay calculations for functional units, instead of the base architecture.

3.2.3.2 Parameterized Pipeline Architectures

For the scalar cores that have been examined previously (CV32E40P, CVA6), the hardware configuration has been more or less fixed. In contrast, a specific Vicuna 2.0 VPU is instantiated with a number of additional parameters. An obvious example is the VLEN, which can be any power of 2 between 64 and 65536, already offering a lot more choices than with scalar register lengths. Additionally, lane widths of unit pipelines, as well as the actual unit pipeline configuration, are flexible.

In regard to performance estimation, these parameters impact both timing calculations and, in some cases, the pipeline structure itself. As an actual example, in the Unpack stages, shift registers with varying length have to be modeled. Additionally, they must also handle stalls resulting from vector load and store instructions or when an instruction is still speculative. Within the Unpack stages, vector registers are read, so before traversing the shift register, but after dispatching, read dependencies need to be resolved. In general, a shift register can be described in CorePerfDSL by chaining simple stages together, while stalls and read dependencies can be modeled with a connector model and corresponding signal. However, the number of stages in the Unpack shift registers of both Pipe 1 and Pipe 2 (see Figure 3.3) are dependent on the hardware configuration and change with VLEN. This would require the use of multiple CorePerfDSL descriptions, one for each VLEN, as the language currently offers no way to indicate that instructions should skip stages.

Proposed Modeling Techniques for Flexible Architectures Two approaches have been identified to solve this problem. The first solution is to use more elaborate external models that can handle different hardware parameters. For instance, the aforementioned shift registers can be moved into an external connector model and set up with a specific hardware configuration at runtime. To provide an example of how external models can be leveraged to model complex and configuration-dependent logic, the implementation of this specific example will be discussed in the next paragraph.

A general solution can be found by extending CorePerfDSL to include pre-processing of the design. Instead of elaborating a new pipeline for each hardware parameter, only relevant parts of the pipeline would need to be adapted. Using the previously mentioned shift registers as an example, with C pre-processor syntax:

```
1 Pipeline ShiftRegister (
2   Stage_1
3   -> Stage_2
4   -> Stage_3
5   #ifdef Use_Stage_4
6   -> Stage_4
7   #endif
```

```

8 #ifdef Use_Stage_5
9     -> Stage_5
10 #endif
11     ...
12 )
13
14 Vector_Instruction (
15     ...
16     uA_Dispatch,
17     uA_Shift_1,
18     uA_Shift_2,
19     uA_Shift_3,
20 #ifdef Use_Stage_4
21     uA_Shift_4,
22 #endif
23 #ifdef Use_Stage_5
24     uA_Shift_5,
25 #endif
26     ...
27 )

```

The need for delay computations could also be eliminated in some cases. For instance, the cost of computing a single vector register through the vector ALU, which depends on the hardware parameters VLEN and the width of the unit pipeline, has to be calculated in an external resource model, as we do not want to specify a core for each possible combination and thus pass these values to the performance estimator at runtime. However, once set, these values are constant, and so is the resulting delay. If such constants can be resolved at translation time, these delays could be moved from an external model into a static delay for the respective resource.

```

1 #define VLEN 512
2 #define PIPE_2_WIDTH 32
3 #define ALU_DELAY (VLEN / PIPE_2_WIDTH)
4
5 Resource V_ALU (ALU_DELAY)
6 Stage V_ALU_Stage (V_ALU)
7 ...
8 Vector_ALU_Instruction (
9     ...
10    uA_Dispatch,
11    uA_Unpack,
12    uA_V_ALU,
13    uA_Pack,
14    ...
15 )

```

Thus, pre-processing eliminates the need for elaborate external models handling many combinations of hardware parameters and passing in hardware parameters at runtime. Optionally, CorePerfDSL would not need to include any pre-processing features, or only a limited subset, as existing tools, such as the C pre-processor, could be used instead. Keep in mind that in the previous example, the C pre-processor would not perform integer

arithmetic in a `#define` statement, so `ALU_DELAY` would still need to be resolved during translation.

Implementation of Shift Registers This paragraph discusses how the previously mentioned shift registers can be fully modeled with the help of an external connector model. The starting point are two `CorePerfDSL` stages representing each real Unpack stage, one to enter the shift register, and one to leave it:

```

1 Stage {
2     Pipe_1_Unpack_Enter (...),
3     Pipe_1_Unpack_Leave (...),
4     Pipe_2_Unpack_Enter (...),
5     Pipe_2_Unpack_Leave (...)
6 }

```

They incur no flat delays, only calculating delays from connector models, and ensure sequential execution of the microactions that calculate shift register delays and retrieve the time the shift register is left. Optionally, a third stage could be inserted before to handle read dependencies, but they can also be handled during the first microaction. In the Connector Model, the shift register itself is modeled as a vector of timestamps, which represent the times at which an instruction enters a particular shift register stage. The number of possible stages is set to the maximum for all configurations, but depending on the `VLEN`, not all stages are used. When calculating the shift register timestamps, a starting time is computed from the maximum of the stage enter time and the last read dependency, at which the first stage of the shift register is valid. A basic diagram of how shift register cycles are calculated can be found in Figure 3.6, using a vector store as an example. Vector stores read one or more vector registers, with the base register named `vs3` in RISC-V assembly. Therefore, when entering Pipe 1 Unpack, the first stage of the shift register is entered at $Stage1EnterTime = \max(DispatchCycle, vs3ReadyCycle)$. From there, the wait time for the next instruction entering Pipe 1 Unpack is calculated with enough cycles to process the current instruction in its respective unit, which also includes checking for a stall. In the given example, the store instruction is dispatched at cycle 100, but it can only start unpacking operands when the source register `vs3` is ready at cycle 105. The wait time for the next instruction entering Pipe 1 is then calculated with $WaitTime = EMUL * \frac{VLEN}{PIPE_1_WIDTH}$. Stalls are handled with 2 timestamps through a connector model, which store the start and the end of a stall. A stall is experienced for a cycle c if it falls within the beginning and the end of that stall:

$$StallStart \leq c < StallEnd \iff CycleInStall(c)$$

When traversing through the shift register, the enter time for stage i ($Cycle(Stage_i)$) is computed as follows:

$$Cycle(Stage_i) = \begin{cases} Cycle(Stage_{i-1}) + 1, & \text{if } CycleInStall(Cycle(Stage_{i-1})) \\ StallEnd, & \text{otherwise} \end{cases}$$

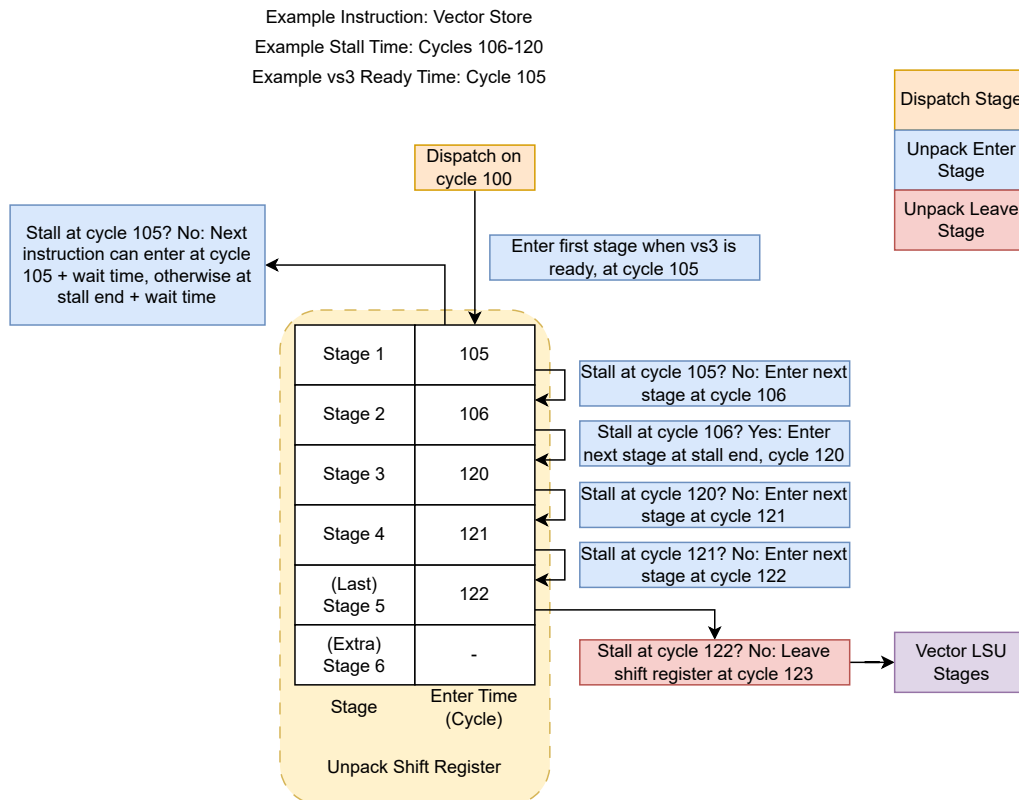


Figure 3.6: Example of how the shift registers work

Looking at the example, one can see that the second stage can be entered in the next cycle after entering the first stage, as $CycleInStall(105)$ evaluates to false. However, the third stage cannot be entered in the following cycle, as a stall was registered by a previous instruction which lasts from cycle 106 to cycle 120. As $CycleInStall(Stage_2)$ is true, $Cycle(Stage_3)$ is set to the end of that stall, and unpacking can proceed for further stages uninterrupted. When performing the microaction to leave Pipe 1 Unpack, the final timestamp is retrieved from the last stage for the current hardware configuration and checked for a stall one last time, before proceeding to the functional unit, in this case the vector LSU.

In Pipe 2, stalls are currently not propagated to the shift register, so a flat delay, depending on VLEN, from the starting time is computed instead of using the shift register model in `Pipe_1_Unpack_Enter`. The delay is then simply retrieved in the second microaction for leaving the shift register. This simplified implementation for Pipe 2 seems to have little impact on accuracy, so a full implementation is left for future work.

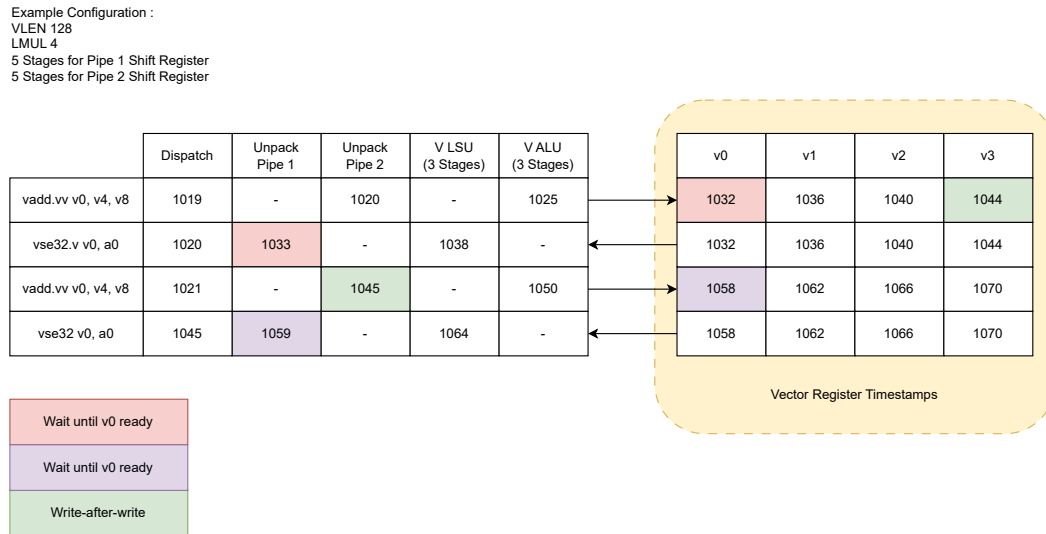


Figure 3.7: Example stage and vector register timings for vector ALU instructions and vector stores

3.2.3.3 Further Pipelining of Vector Instructions

Another concept that is not encountered in the scalar case is the splitting and further pipelining of single instructions. A vector instruction will perform reading and unpacking, computations or memory requests, and writing to vector registers or memory simultaneously if the instruction covers multiple vector registers, i.e. LMUL is greater than 1. Currently, this cannot be accurately modeled in the sense that an instruction is only ever imagined to be in a single stage at a time, but the timing outcome can still be calculated correctly in most instances.

Current Implementation The current naive approach is to combine register packing and computation in a single stage. There, register write times are calculated in a simple loop that increments a running timestamp by the number of cycles required to process a single set of source registers, and applies that timestamp to the respective target register. This works in the general case, as arithmetic instructions cannot stall and, with some exceptions, always complete register writes before they would cause a delay due to read dependencies. An example is given in Figure 3.7, where entering cycles and vector register timestamps for a sequence of instructions are examined. Looking at the vector register timestamps, one can see that writes are modeled closely after the RTL implementation, where a vector ALU instruction (in this case `vadd.vv`) completes register writes one after the other, at different times. After the first instruction writes to `v0`, the subsequent store instruction can start unpacking operands, and since the ALU takes at most the same amount of cycles to process a register as a memory instruction, other registers do not need to be considered as dependencies. Hence, after reaching the vector LSU,

the store simply takes an additional $EMUL * \frac{VLEN}{MEM_WIDTH} = 16$ cycles to complete, while in reality, unpacking and dependency checking is done for each set of input registers, i.e. 4 times for an LMUL of 4.

Vector instructions that use the vector division unit (`vdiv`, `vdivu`, `vrem`, `vremu`) are an exception here. A division takes a number of cycles for each element, so a subsequent vector store can process the first register after it is done, but then needs to wait until the next register completes before unpacking further. A simple solution for this is to set the same final timestamp for all target registers, as it is highly likely that all computed registers are used afterward anyway. This approach is illustrated in Figure 3.8. Here, `v0` through `v3` are given the same timestamp after both divisions, so

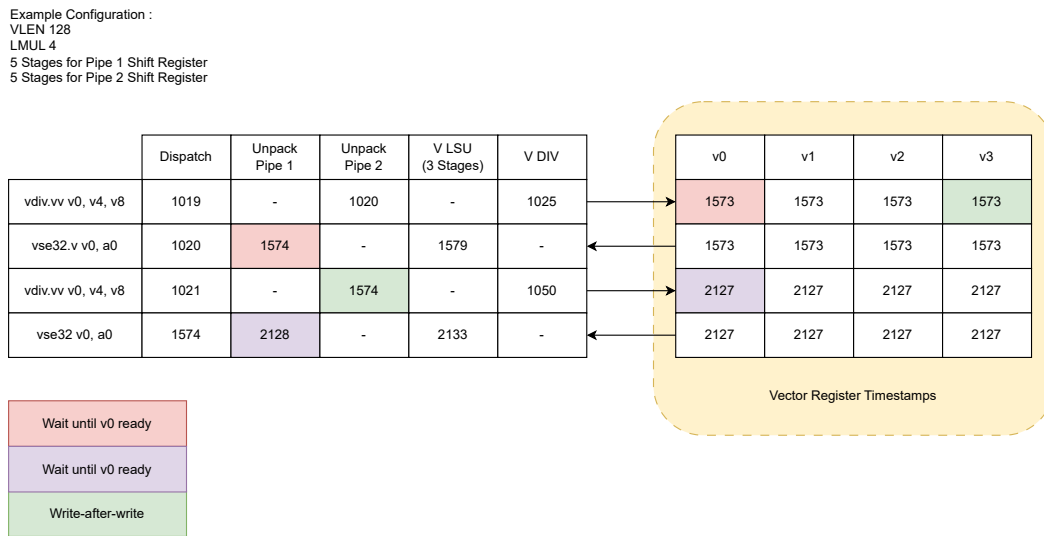


Figure 3.8: Example stage and vector register timings for vector DIV instructions and vector stores

a store has to wait until the division is fully completed, instead of starting when only the first register is available. As previously mentioned, this generally leads to accurate cycle estimation, as it would be unusual to only require a partial result after a division. That example executed on ETISS with performance estimation deviates from the RTL simulation only by a total 14 cycles. However, by simply changing the LMUL before a store, cycle prediction becomes very inaccurate, as demonstrated in Figure 3.9. Compared to Verilator, performance estimation now deviates by a total 260 cycles, which roughly accounts for the two extra registers (i.e. 8 divisions with 32 cycles each, including some extra cycles) not needed by the last store.

Proposed Modeling Techniques for Pipelined Vector Instructions While currently not implemented, two possible solutions have been devised for this problem. Firstly, the external connector model handling the unit pipeline timing logic could be

3. PERFORMANCE SIMULATION EXTENSIONS FOR VECTOR COPROCESSORS

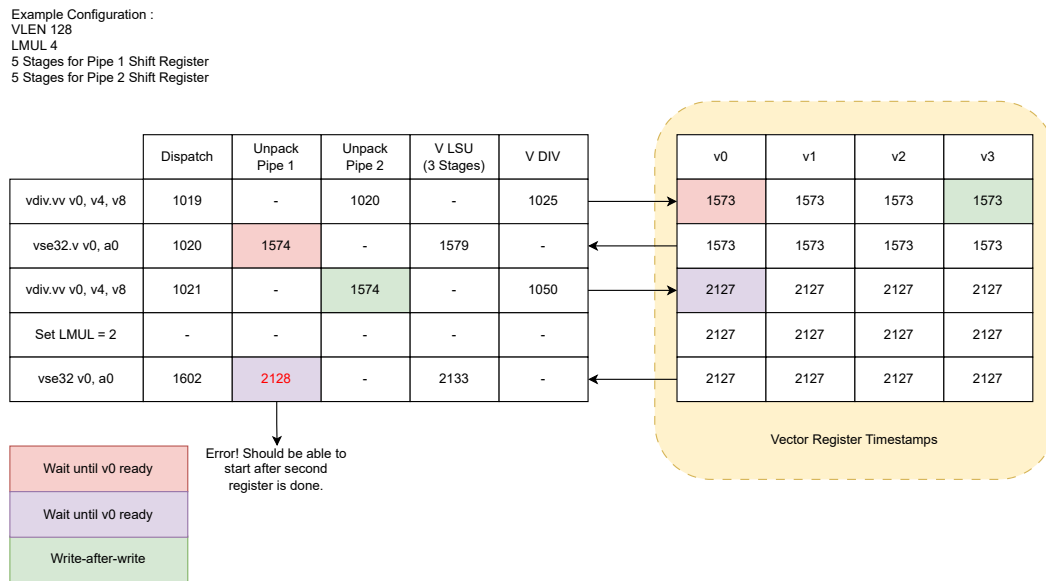


Figure 3.9: Example stage and vector register timings for vector DIV instructions and vector stores, including a change in LMUL

extended further. Checking read dependencies, traversing through the shift register, and stall checking has to be simulated for all registers for the current register group configuration. After unpacking a register, a timestamp is set to signal the following unit stage when it can actually start processing that register. A diagram of how this would look like is provided in Figure 3.10. Instead of only waiting for the first set of registers and then proceeding with the unit stage, an additional set of timestamps for each potential set of source registers is introduced, which takes in timestamps from the final Unpack shift register stage. Handling read dependencies and unpacking is then done LMUL times, and after each round the final timestamp is set. The functional unit stage then just adds the specific delay ("Cycles per Register") for the respective operation and configuration to the previously set timestamps, after which vector register times can be set.

Alternatively, CorePerfDSL itself could be extended to enable the splitting up of instructions. One possibility would be to actually split instructions into a sequence of sub-instructions, as shown in Figure 3.11. A special microaction or stage could split an instruction into a number of sub-instructions, depending on the current instruction configuration, and place them into a buffer. These sub-instructions would then be processed just like regular instructions, but would only use a single set of source and target registers, instead of a register group. This could be implemented by allowing scheduling functions to loop over parts of a pipeline, essentially shifting the logic to loop over a register group one level higher.

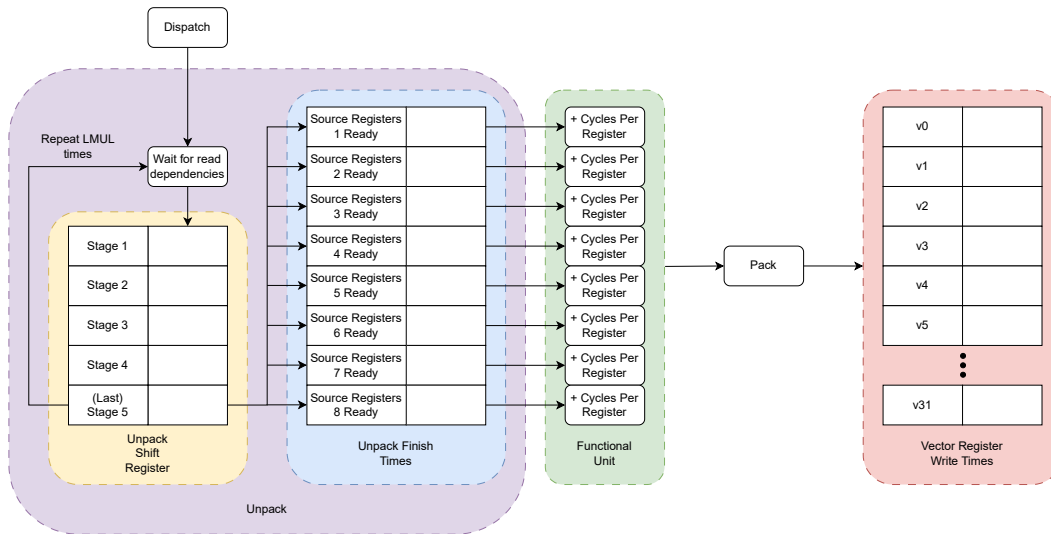


Figure 3.10: Simulating multiple rounds of unpacking

3.2.3.4 Dependencies Between Parallel Pipelines

Dependencies between the parallel scalar core and the VPU, in general, work the same as with a single pipeline, as microactions set in one pipeline can still be read in the other pipeline. The main use case when modeling Vicuna 2.0 is to express the co-processors' ability to stall the scalar core. As mentioned previously, vector loads and stores, as well as instructions that move values from a vector register into a scalar register, halt the scalar core until completion. Stalling vector instructions occupy the scalar Writeback stage, so we only have to introduce a single dependency through a connector model, which all instructions read. The corresponding timestamp is simply set whenever a stalling instruction completes. Thanks to the mathematical model behind CorePerfDSL, the "back-pressure" automatically applies the stall to previous stages. Looking at Figure 3.3, we connect V RES to EX via a connector model. As the relevant instructions retire from V RES, they set a timestamp at which a subsequent instruction can exit EX. However, there is a yet unresolved issue that can occur for instructions that both set and require the same timestamp, e.g. a vector load stalls EX from the VPU, but also needs to wait for stalls while going through the scalar core. As it is up to M2-ISA-R-Perf in which order stages of parallel pipelines are processed, we cannot guarantee whether setting or reading is done first.

Proposed Modeling Techniques for Inter-Pipeline Dependencies In the case where a timestamp has to be read first before it is set, this problem can be circumvented, if it appears, by checking whether that timestamp was already read, and buffering it if that has not happened yet, as demonstrated in Figure 3.12. A stalling vector instruction passes through both parallel pipelines and both reads and writes the `WbFreeTime` timestamp.

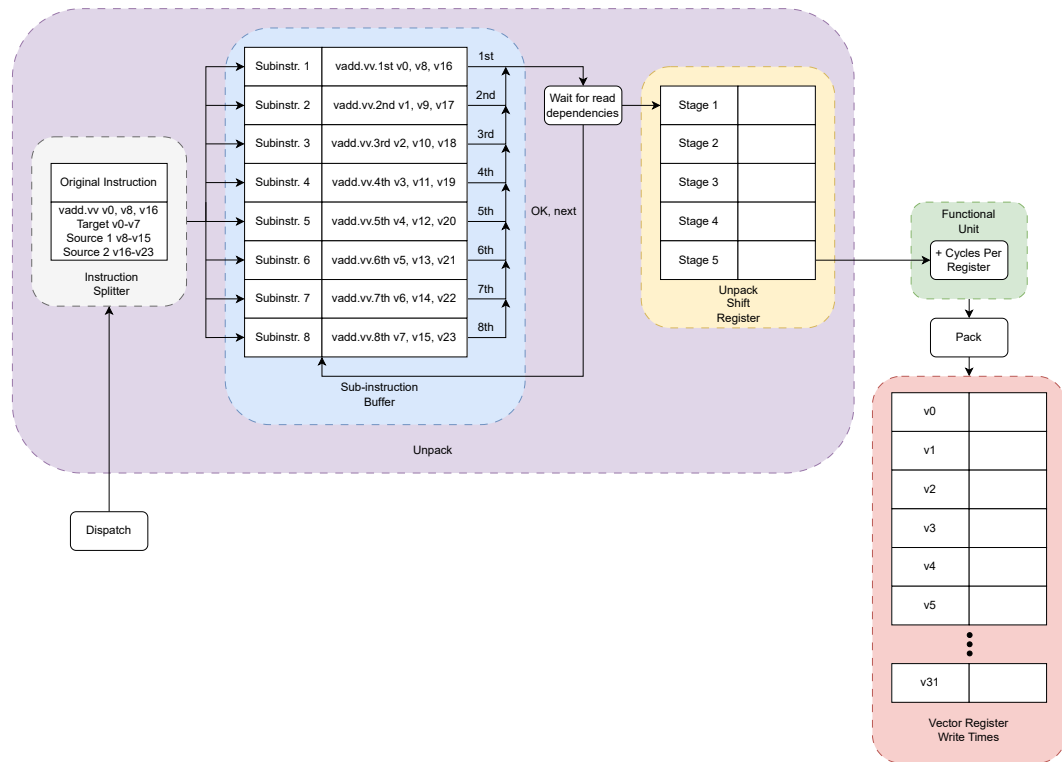


Figure 3.11: Splitting instructions into sub-instructions

Logically, when passing through EX, the timestamp coming from the previous stalling instruction is needed, however, CorePerfDSL might generate a scheduling function where V RES is processed before EX, and the instruction would erroneously stall itself in EX. To guarantee that the correct timestamp is consumed in EX, `WbFreeTime` is copied before being overwritten, and the order of operations is determined by using a flag. The read operation in EX would then return either the regular or the buffered timestamp, depending on whether it was already written by the same instruction.

The other case, where setting the timestamp has to happen first, is non-trivial, as reading the value can not be deferred to a later point. Currently, the only alternative is to find a substitute dependency that is already available from a previous instruction. Properly handling inter-pipeline dependencies, where the same timestamp is read and written in different parallel pipelines, is a feature that is missing in CorePerfDSL. A possible solution would be to generate a stage ordering through a list of stage dependencies for each stage. For any stage, this list would contain its predecessor (if it is not a root stage) and any stages in parallel pipelines explicitly listed as dependencies. If no logical rules are violated, e.g. no two stages can be mutual dependencies, then these lists represent a Directed Acyclic Graph (DAG), from which an ordering can be found via Depth-First Search (DFS), using the last stage of each parallel pipeline as starting nodes. An example

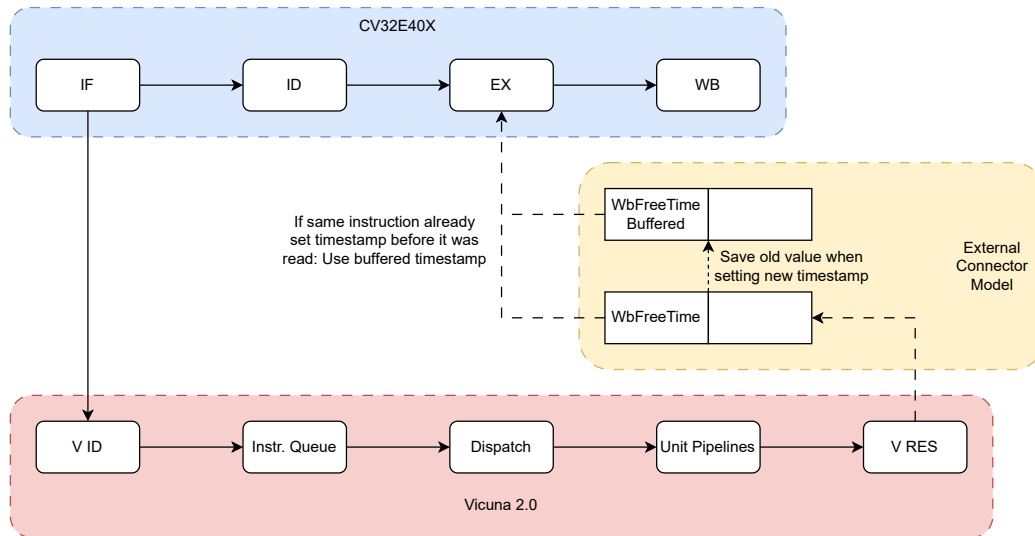


Figure 3.12: Solution to enforce temporal order in case read before write

using the CV32E40X and Vicuna 2.0 is shown in Figure 3.13. The only additional

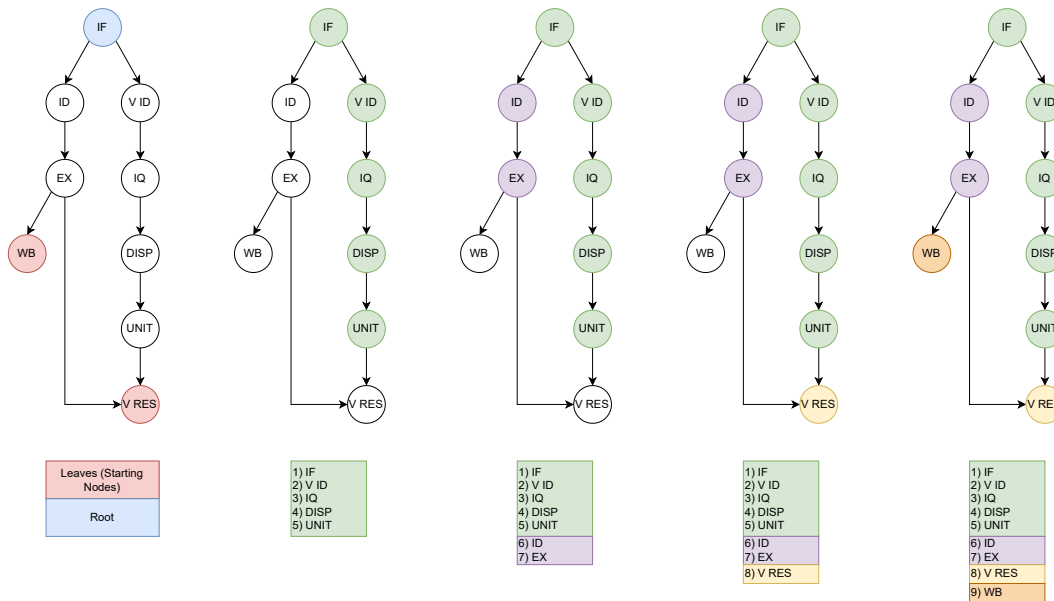


Figure 3.13: Generating a stage ordering for CV32E40X + Vicuna 2.0

dependency introduced in this example is that EX has to be processed before V RES, and since there is now an edge in the corresponding DAG, EX will always be ordered before V RES.



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

Experimental Results

4.1 Functional Simulation Results

To see whether the implementation of a vector instruction in ETISS adheres to the specification, it is easiest to compare computation results against a reference. Initially, handwritten examples were used, however, covering the many different combinations of LMUL, SEW, and number of vector elements is infeasible to do for the many instructions introduced in the vector extension. Luckily, there exists a testing repository for RISC-V vector instructions in [12] that generates an extensive number of test cases for each instruction programmatically. This works by implementing a custom instruction in Spike, an ISA simulator for RISC-V that is often used as a reference implementation, that generates reference results in an initial pass, and then making those results directly available in the final test binary. Running these test binaries in ETISS showed no errors for instructions in `Zve32d`.

4.2 Performance Comparison Workflow

With ETISS now supporting RISC-V vector instructions and the VPU timing model set up, RISC-V programs with SIMD instructions can be simulated and their performance on a CV32E40X with Vicuna 2.0 estimated. The performance metrics can then be compared to a cycle-accurate RTL simulation of the same program using Verilator. A diagram giving an overview of the main toolchain and the resulting workflow can be found in Figure 4.1. To generate a C++ architecture for ETISS, as well as the C code for each instruction used in JIT-compilation, M2-ISA-R requires a CoreDSL model and generates the aforementioned architecture and code. The specific performance estimator is generated from a CorePerfDSL timing model, as well as the respective CoreDSL description. The generated code is then used to compile ETISS, which is now capable of simulating the architecture and instructions specified in CoreDSL, as well as performing

4. EXPERIMENTAL RESULTS

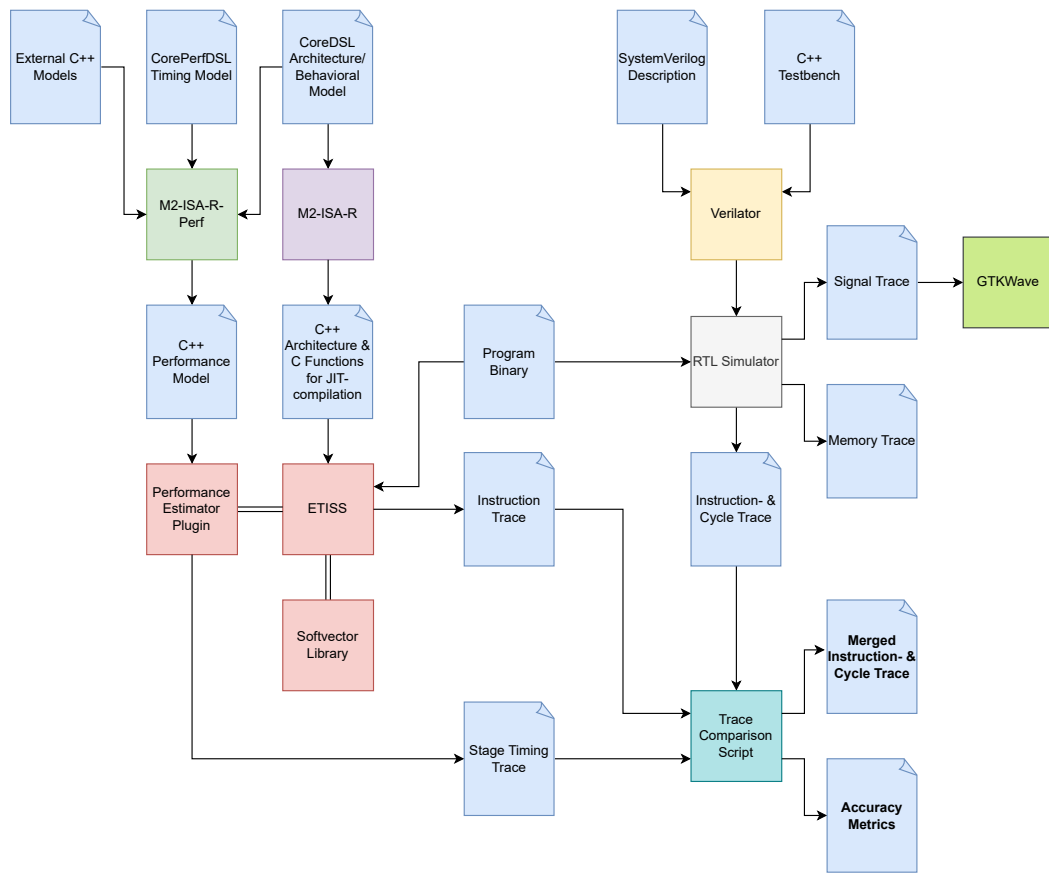


Figure 4.1: View of the comparison workflow

performance estimation. ETISS also includes the Softvector library, which implements RISC-V vector instructions, so the CoreDSL model does not need to implement those, but can refer to the library via external functions. A program binary can then be given to the ETISS executable as an input, and ETISS will simulate that program. An instruction trace is received as an output, as well as a stage timing trace and the estimated CPI from the performance estimator plugin.

On the RTL side, an HDL description, as well as a user-defined test bench are used by Verilator to generate an executable simulator. In this case, the test bench can receive a program binary as an input, which is reflected in the resulting RTL simulator. That program can then be simulated, and a signal trace, memory trace, and an instruction trace including cycle information for each instruction are received as an output. The signal trace can be inspected with GTKWave, which is useful for refining the CorePerfDSL model. The instruction trace and timing information coming from ETISS, as well as the instruction trace from the RTL simulation are then fed into a trace comparison script,

which merges all information into a final combined trace and calculates the CPI for both simulators, and how much the timing model deviates from the full hardware model.

4.2.1 Traces

Apart from the differences in terminating a program, the startup code for ETISS and the Verilator simulator is different. This means that one has to find a way to specify where the comparison should start and where to end. The first idea was to use a library to compare sequences, such as Python's `diff`lib. While this works for shorter traces, comparing longer-running programs is infeasible. Furthermore, when the address layout is not the same for both programs, as it was in the beginning before aligning the linker scripts for ETISS and the Verilator simulator, control flow instructions will often be different while the functionality of the program is the same, thus only matching parts of the trace. Another approach that was used previously in the Verilator main file to ignore startup code, was to rely on the compiler to put the main function at the same address each time. The problem with this is that, while it seemingly works well for regular builds, debug builds change the placement of the main function, and an approach that guarantees that measuring starts at the same point is preferred. This led to the final approach, using two functions that mark the beginning and the end of measurement. When preventing any optimization or inlining of these functions, they will generate corresponding labels in the assembly, and one can search for those labels. These addresses can then be used when analyzing the instruction traces to cut out unwanted startup- or return code that might differ for both targets.

4.2.2 Measurements

The base measurement is not the total amount of cycles or CPI, as positive and negative errors in the timing model can cancel out, but delays between two subsequent instructions. As both vector and scalar instructions are modeled to go through the scalar core, the scalar Writeback stage was chosen as the measuring point, or more precisely, the cycle in which an instruction enters the Writeback stage. This approach works out nicely, as this can easily be done for both the RTL simulation and the performance estimation. Verilator makes signals marked `public` via a comment available in the main simulation file, so by checking if the program counter changes in the writeback stage at any given time, for any cycle that registers a change, the corresponding instruction, PC, and cycle are written to the instruction trace, as well as the difference to the previous instruction. The performance estimator plugin records stage timings for each instruction anyway, so no additional work needs to be done, and the difference in cycles between two subsequent instructions entering the Writeback stage can simply be read from the resulting stage timing trace.

4.3 Performance Estimation Results

As mentioned previously, while CPI accuracy is important, a more meaningful question is whether the difference in cycles between two instructions is the same in the estimation as in the RTL simulation. Given cycles c_i^{etiss} and c_i^{rtl} , for ETISS and Verilator respectively, with i naming the instruction index, we want to minimize

$$|(c_{i+1}^{etiss} - c_i^{etiss}) - (c_{i+1}^{rtl} - c_i^{rtl})|$$

for all pairs of instructions i and $i + 1$. A simple metric to strive for would be the *average deviation per instruction* (ADI), which can be defined as

$$\frac{\sum_i |(c_{i+1}^{etiss} - c_i^{etiss}) - (c_{i+1}^{rtl} - c_i^{rtl})|}{\#instructions}$$

4.3.1 Configurations

The timing model is set up in a way that allows for estimation of different hardware configurations, namely different VLENs and lane widths for Pipe 2 (VLANE_WIDTH), referring back to Figure 3.3. Currently, Pipe 1 has a fixed width, as well as the memory interface, as at the time of writing, there were still some issues present in the RTL model regarding the memory interface, and each additional parameter also causes an exponential increase in configurations. The base architecture is `rv32im_zve32x`, meaning integer-only up to 32 bit wide elements. Vector floating point instructions are not included here, as they incur the same delays as regular vector ALU operations anyway, same with vector slide instructions. The chosen VLENs are 64, 128, 256, 512, and 1024, as they correspond with the named vector register length extensions in [16], and those are combined with the respective legal lane widths $\in (32, 64, 128)$, with $\text{VLANE_WIDTH} \leq \frac{\text{VLEN}}{2}$.

4.3.2 Simple Tests

These initial tests cover very simple patterns that, in order for real workloads to be estimated accurately, should already be close to their Verilator counterparts. They test the patterns by themselves, so between each sequence of instructions for any given SEW and LMUL combination, there is ample space in the form of NOPs, such that they do not interfere with each other. Results for these synthetic examples can be found in Tables 4.1, 4.2, 4.3, 4.4, 4.5, and 4.6.

VLEN	VLANE_W	CPI ETISS	CPI Verilator	Error	ADI
64	32	2.6060	2.6060	0.0000 %	0.0000
128	32	3.2826	3.2826	0.0000 %	0.0000
128	64	3.2500	3.2337	0.5042 %	0.0163
256	32	4.7826	4.7826	0.0000 %	0.0000
256	64	4.7174	4.7174	0.0000 %	0.0000
256	128	4.6848	4.6685	0.3492 %	0.0163
512	32	7.7826	7.7826	0.0000 %	0.0000
512	64	7.6522	7.6522	0.0000 %	0.0000
512	128	7.6196	7.6196	0.0000 %	0.0000
1024	32	13.7826	13.7826	0.0000 %	0.0000
1024	64	13.5870	13.5870	0.0000 %	0.0000
1024	128	13.5217	13.5217	0.0000 %	0.0000

Table 4.1: Results for simple pattern load, `vadd.vv`, store

VLEN	VLANE_W	CPI ETISS	CPI Verilator	Error	ADI
64	32	3.3438	3.3438	0.0000 %	0.0000
128	32	4.3164	4.3164	0.0000 %	0.0000
128	64	4.2578	4.2344	0.5535 %	0.0234
256	32	6.4609	6.4609	0.0000 %	0.0000
256	64	6.3672	6.3672	0.0000 %	0.0000
256	128	6.3203	6.2969	0.3722 %	0.0234
512	32	10.7734	10.7734	0.0000 %	0.0000
512	64	10.5859	10.5859	0.0000 %	0.0000
512	128	10.5273	10.5273	0.0000 %	0.0000
1024	32	19.3984	19.3984	0.0000 %	0.0000
1024	64	19.1055	19.1055	0.0000 %	0.0000
1024	128	19.0117	19.0117	0.0000 %	0.0000

Table 4.2: Results for simple pattern load, `vmul.vv`, store

VLEN	VLANE_W	CPI ETISS	CPI Verilator	Error	ADI
64	32	31.4688	31.5117	-0.1364 %	0.0430
128	32	60.6719	60.7070	-0.0579 %	0.0352
128	64	31.9609	31.9492	0.0367 %	0.0117
256	32	119.1719	119.1953	-0.0197 %	0.0234
256	64	61.7500	61.7734	-0.0379 %	0.0234
256	128	33.0391	33.0156	0.0710 %	0.0234
512	32	236.1719	236.1875	-0.0066 %	0.0156
512	64	121.3281	121.3438	-0.0129 %	0.0156
512	128	63.9062	63.9219	-0.0244 %	0.0156
1024	32	470.1719	470.1836	-0.0025 %	0.0117
1024	64	240.4844	240.4961	-0.0049 %	0.0117
1024	128	125.6406	125.6523	-0.0093 %	0.0117

Table 4.3: Results for simple pattern load, `vdiv.vv`, store

VLEN	VLANE_W	CPI ETISS	CPI Verilator	Error	ADI
64	32	3.2391	3.2391	0.0000 %	0.0000
128	32	4.5815	4.5815	0.0000 %	0.0000
128	64	4.5815	4.5815	0.0000 %	0.0000
256	32	7.4701	7.4701	0.0000 %	0.0000
256	64	7.4701	7.4701	0.0000 %	0.0000
256	128	7.4701	7.4701	0.0000 %	0.0000
512	32	13.2962	13.2962	0.0000 %	0.0000
512	64	13.2962	13.2962	0.0000 %	0.0000
512	128	13.2962	13.2962	0.0000 %	0.0000
1024	32	24.9484	24.9484	0.0000 %	0.0000
1024	64	24.9484	24.9484	0.0000 %	0.0000
1024	128	24.9484	24.9484	0.0000 %	0.0000

Table 4.4: Results for simple pattern load, `vredsum.vs`, store

VLEN	VLANE_W	CPI ETISS	CPI Verilator	Error	ADI
64	32	1.9593	1.9593	0.0000 %	0.0000
128	32	2.2907	2.2907	0.0000 %	0.0000
128	64	2.2209	2.1860	1.5957 %	0.0349
256	32	2.9535	2.9535	0.0000 %	0.0000
256	64	2.8140	2.8140	0.0000 %	0.0000
256	128	2.7442	2.7093	1.2876 %	0.0349
512	32	4.2791	4.2791	0.0000 %	0.0000
512	64	4.0000	4.0000	0.0000 %	0.0000
512	128	3.8605	3.8605	0.0000 %	0.0000
1024	32	6.9302	6.9302	0.0000 %	0.0000
1024	64	6.3721	6.3721	0.0000 %	0.0000
1024	128	6.0930	6.0930	0.0000 %	0.0000

Table 4.5: Results for simple pattern `vmv.v.i, store`

VLEN	VLANE_W	CPI ETISS	CPI Verilator	Error	ADI
64	32	2.4106	2.4106	0.0000 %	0.0000
128	32	3.1111	3.1111	0.0000 %	0.0000
128	64	3.1111	3.1111	0.0000 %	0.0000
256	32	4.6763	4.6763	0.0000 %	0.0000
256	64	4.6763	4.6763	0.0000 %	0.0000
256	128	4.6763	4.6763	0.0000 %	0.0000
512	32	7.8454	7.8454	0.0000 %	0.0000
512	64	7.8454	7.8454	0.0000 %	0.0000
512	128	7.8454	7.8454	0.0000 %	0.0000
1024	32	14.1836	14.1836	0.0000 %	0.0000
1024	64	14.1836	14.1836	0.0000 %	0.0000
1024	128	14.1836	14.1836	0.0000 %	0.0000

Table 4.6: Results for simple pattern `load, store`

4.3.3 ML Inference Workloads

To demonstrate that it is possible to accurately estimate proper workloads that can utilize SIMD, 3 Machine Learning (ML) inference benchmarks have been run, and the results compared to their RTL counterparts. Inference means feeding data into an already trained model and then receiving an output, for example for classification tasks. The three benchmarks are `toycar` (anomaly detection), `aww` (keyword spotting), and `vww` (visual wake words), which are also used in some publications regarding the RISC-V vector extension, e.g. in [40] or [11]. For many applications, inference can make use of SIMD instructions. The input to a neuron in a fully connected layer, that is used in `toycar` for instance, is just the sum of products of outputs from the previous layer with a weight vector, plus a bias. Hence, the inputs to a layer can be calculated using `vmacc` instructions. Results for the ML benchmarks can be found in Tables 4.7, 4.8, and 4.9.

VLEN	VLANE_W	CPI ETISS	CPI RTL	Error	ADI	# Instr.
64	32	2.9470	3.0182	-2.3600 %	0.0712	950997
128	32	3.5179	3.5236	-0.1624 %	0.0057	587766
128	64	2.9949	2.8853	3.7994 %	0.1154	587766
256	32	4.3366	4.4521	-2.5938 %	0.1155	414096
256	64	3.3130	3.3205	-0.2260 %	0.0075	414096
256	128	2.9725	2.8967	2.6185 %	0.0834	414096
512	32	5.7960	5.6689	2.2406 %	0.1871	325618
512	64	4.0476	3.9206	3.2399 %	0.1871	325618
512	128	3.5041	3.3566	4.3954 %	0.1666	325618
1024	32	7.1432	7.1634	-0.2809 %	0.0201	282058
1024	64	4.8559	4.9927	-2.7390 %	0.1368	282058
1024	128	4.0439	4.0640	-0.4952 %	0.0201	282058

Table 4.7: Results for `toycar`

VLEN	VLANE_W	CPI ETISS	CPI Verilator	Error	ADI	# Instr.
64	32	1.6575	1.6649	-0.4443 %	0.0074	31459285
128	32	1.6546	1.6627	-0.4841 %	0.0081	29016385
128	64	1.5796	1.5744	0.3322 %	0.0125	29016385
256	32	1.7474	1.7559	-0.4862 %	0.0086	27607588
256	64	1.6407	1.6468	-0.3730 %	0.0062	27607588
256	128	1.5989	1.5969	0.1262 %	0.0096	27607588
512	32	1.9633	1.9818	-0.9295 %	0.0185	26906876
512	64	1.7932	1.7996	-0.3545 %	0.0064	26906876
512	128	1.7361	1.7424	-0.3619 %	0.0063	26906876
1024	32	2.3517	2.3887	-1.5508 %	0.0371	26554756
1024	64	2.1238	2.1485	-1.1496 %	0.0247	26554756
1024	128	2.0286	2.0351	-0.3179 %	0.0065	26554756

Table 4.8: Results for aww

VLEN	VLANE_W	CPI ETISS	CPI RTL	Error	ADI	# Instr.
64	32	1.7988	1.8097	-0.6015 %	0.0110	78807060
128	32	1.8192	1.8313	-0.6630 %	0.0123	71298379
128	64	1.7222	1.7164	0.3387 %	0.0160	71298379
256	32	1.8705	1.8806	-0.5377 %	0.0102	68974858
256	64	1.7491	1.7565	-0.4179 %	0.0075	68974858
256	128	1.7018	1.7000	0.1137 %	0.0114	68974858
512	32	2.0393	2.0571	-0.8631 %	0.0179	69103622
512	64	1.8690	1.8758	-0.3604 %	0.0069	69103622
512	128	1.8146	1.8212	-0.3586 %	0.0066	69103622
1024	32	2.1835	2.2042	-0.9382 %	0.0208	74900400
1024	64	2.0293	2.0466	-0.8458 %	0.0174	74900400
1024	128	1.9718	1.9769	-0.2598 %	0.0052	74900400

Table 4.9: Results for vww

4.3.4 Discussion

As seen with both the simple tests and the ML inference benchmarks, creating a timing model that is accurate and reflects changes in CPI is possible to do with ETISS and CorePerfDSL. Looking only at the CPI error, the results are already quite good, with an absolute deviation of less than 5 % across all benchmarks. On the other hand, there is still room for improvement, as some combinations of `VLANE_WIDTH` and `VLEN` cause comparatively higher errors and ADI, which hints at the fact that the true correlation between those parameters and the instruction timings have not fully been captured in the timing model. However, even with good results for chosen test cases and only a small average deviation per instruction, one still has to be careful to make general assumptions for arbitrary input programs. For one, these inference workloads often show the same

repeating patterns of (vector) assembly instructions done many times. As an example, `toyCar` repeats the pattern

```
loop:
vle8.v      v10, (a3)
vle8.v      v11, (a2)
vsext.vf4   v12, v10
vsext.vf4   v14, v11
vadd.vx     v10, v14, s10
vadd.vx     v12, v12, s9
vmacc.vv    v8, v10, v12
add         a3, a3, s2
sub         a1, a1, s2
add         a2, a2, s2
bnez        a1, loop
```

that results from calculating values in a fully connected network for a large part of the program. Arbitrary programs could have instruction sequences that incur stalls due to dependencies that are present in hardware, but not yet properly modeled in the timing model. Additionally, scalar instructions can be modeled much more easily, and thus do not contribute heavily to the error. Depending on the ratio of scalar to vector instructions, errors due to the vector timing model could be drowned out. Still, both the low CPI error and ADI for the chosen benchmarks and simple tests show that, with further refinement of CorePerfDSL and the Vicuna 2.0 timing model, using ETISS and the performance estimator plugin can be a sensible approach for cycle-approximate modelling of vector co-processors.

4.4 Simulation Speed

While cycle-accuracy is an important metric, and a CorePerfDSL is certainly easier to implement than a full hardware description, the case still has to be made for using an estimation-based approach from the perspective of simulation speed. To get an idea where ETISS stands in this regard, the previously used ML benchmarks have also been used to measure simulation times for ETISS with and without performance estimation, Verilator, and QEMU, a behavioral-only simulator. Both ETISS and Verilator have been run without writing traces. The results can be found in the following Tables 4.10, 4.11, and 4.12. The instruction counts listed refer to the number of instructions used for accuracy measurements, so they do not include startup- and return code. However, the difference in total executed instructions for the ML benchmarks is insignificant. Timing itself was performed with the Bash builtin version of GNU `time`, which reports wall clock time (*real*), user space time (*user*), and kernel space time (*sys*) and provides millisecond accuracy, and user space time was used for comparison. Simulations with ETISS and QEMU were performed 10 times to get an average execution time, from which the average

simulation speed in MIPS was calculated. Since Verilator simulation times are already quite large, small deviations in execution times only have a negligible impact on MIPS, so those were only performed once. For both ETISS and QEMU the Pipe 2 lane width (referring back to Figure 3.3) is irrelevant, as QEMU does not take in such a parameter, and it should have no effect on the execution speed of performance estimation, as it is only used as a constant in delay calculations. For Verilator, the same lane width configurations as in the accuracy measurements are used, i.e. all legal lane widths $\in (32, 64, 128)$ for a particular VLEN. Furthermore, simulations for a VLEN of 64 have been omitted, as QEMU only accepts values $\in (128, 256, 512, 1024)$. All measurements were performed on an AMD Ryzen 9 9950X 16-Core Processor.

4.4.1 ETISS JIT-Compiler

ETISS offers the choice of either TCC, GCC, or LLVM for JIT-compilation, with TCC and GCC currently working out of the box. For longer programs, GCC should win out due to better optimization, while TCC seems to offer faster compilation. GCC also seems to incur a large initial startup delay, as for all tested programs ETISS internally reports faster simulation times for GCC, while TCC outperforms GCC in total program runtime for all ML benchmarks. Hence, for all results reported in this thesis, TCC is used.

4.4.2 Results

Results for measurements of simulation speeds can be found in Tables 4.10, 4.11, and 4.12.

4. EXPERIMENTAL RESULTS

Simulator	VLEN	VLANE_W	(Avg.) Time	(Avg.) MIPS	# Instr.
QEMU	128	-	0.0070s	83.9666	587766
ETISS	128	-	0.8342s	0.7046	587766
ETISS + Perf. Est.	128	-	0.9189s	0.6396	587766
Verilator	128	32	9.9340 s	0.0592	587766
Verilator	128	64	9.2520 s	0.0635	587766
QEMU	256	-	0.0065s	63.7071	414096
ETISS	256	-	0.7735s	0.5354	414096
ETISS + Perf. Est.	256	-	0.8622s	0.4803	414096
Verilator	256	32	10.3970 s	0.0398	414096
Verilator	256	64	9.0670 s	0.0457	414096
Verilator	256	128	8.8700 s	0.0467	414096
QEMU	512	-	0.0066s	49.3361	325618
ETISS	512	-	0.7450s	0.4371	325618
ETISS + Perf. Est.	512	-	0.8252s	0.3946	325618
Verilator	512	32	13.9350 s	0.0234	325618
Verilator	512	64	11.7010 s	0.0278	325618
Verilator	512	128	10.5980 s	0.0307	325618
QEMU	1024	-	0.0070s	40.2940	282058
ETISS	1024	-	0.7265s	0.3882	282058
ETISS + Perf. Est.	1024	-	0.8098s	0.3483	282058
Verilator	1024	32	21.8590 s	0.0129	282058
Verilator	1024	64	17.5400 s	0.0161	282058
Verilator	1024	128	15.5030 s	0.0182	282058

Table 4.10: Simulation speeds for toy-car

Simulator	VLEN	VLANE_W	(Avg.) Time	(Avg.) MIPS	# Instr.
QEMU	128	-	0.0415s	699.1900	29016385
ETISS	128	-	2.3239s	12.4861	29016385
ETISS + Perf. Est.	128	-	2.8344s	10.2372	29016385
Verilator	128	32	242.0770 s	0.1199	29016385
Verilator	128	64	250.1820 s	0.1160	29016385
QEMU	256	-	0.0538s	513.1522	27607588
ETISS	256	-	1.9192s	14.3849	27607588
ETISS + Perf. Est.	256	-	2.3740s	11.6291	27607588
Verilator	256	32	277.5040 s	0.0995	27607588
Verilator	256	64	280.4320 s	0.0984	27607588
Verilator	256	128	297.7710 s	0.0927	27607588
QEMU	512	-	0.0748s	359.7176	26906876
ETISS	512	-	1.7469s	15.4026	26906876
ETISS + Perf. Est.	512	-	2.1676s	12.4132	26906876
Verilator	512	32	396.2010 s	0.0679	26906876
Verilator	512	64	398.8520 s	0.0675	26906876
Verilator	512	128	390.0000 s	0.0690	26906876
QEMU	1024	-	0.1185s	224.0908	26554756
ETISS	1024	-	1.7305s	15.3451	26554756
ETISS + Perf. Est.	1024	-	2.1594s	12.2973	26554756
Verilator	1024	32	657.8670 s	0.0404	26554756
Verilator	1024	64	628.7600 s	0.0422	26554756
Verilator	1024	128	605.8690 s	0.0438	26554756

Table 4.11: Simulation speeds for aww

4. EXPERIMENTAL RESULTS

Simulator	VLEN	VLANE_W	(Avg.) Time	(Avg.) MIPS	# Instr.
QEMU	128	-	0.1093s	652.3182	71298379
ETISS	128	-	5.0685s	14.0670	71298379
ETISS + Perf. Est.	128	-	6.2097s	11.4818	71298379
Verilator	128	32	637.3250 s	0.1119	71298379
Verilator	128	64	646.4430 s	0.1103	71298379
QEMU	256	-	0.1455s	474.0540	68974858
ETISS	256	-	3.5693s	19.3245	68974858
ETISS + Perf. Est.	256	-	4.5892s	15.0298	68974858
Verilator	256	32	734.6930 s	0.0939	68974858
Verilator	256	64	734.3930 s	0.0939	68974858
Verilator	256	128	779.8680 s	0.0884	68974858
QEMU	512	-	0.2060s	335.4545	69103622
ETISS	512	-	2.9797s	23.1915	69103622
ETISS + Perf. Est.	512	-	3.9427s	17.5270	69103622
Verilator	512	32	1042.5340 s	0.0663	69103622
Verilator	512	64	1049.8260 s	0.0658	69103622
Verilator	512	128	1038.8970 s	0.0665	69103622
QEMU	1024	-	0.3320s	225.6036	74900400
ETISS	1024	-	2.8104s	26.6512	74900400
ETISS + Perf. Est.	1024	-	3.7953s	19.7350	74900400
Verilator	1024	32	1747.3800 s	0.0429	74900400
Verilator	1024	64	1720.5410 s	0.0435	74900400
Verilator	1024	128	1674.6550 s	0.0447	74900400

Table 4.12: Simulation speeds for vww

4.4.3 Discussion

Generally speaking, the results reflect the capabilities of each simulator. ETISS and QEMU, both using dynamic binary translation, benefit from longer running programs, where the translation overhead matters less. Looking at ETISS with performance estimation, the speedup over Verilator is less 10x for the shorter `toyCar` benchmark, but more than 100x for `aww` and `vww` for larger VLENs. Interestingly enough, the performance of ETISS seems to increase with a larger VLEN, whereas QEMU becomes slower. While no explanation can be given as to why the performance of QEMU decreases, ETISS seems to suffer from insufficient optimization of vector instructions. Hence, if the ratio of scalar instructions to vector instructions rises, as is the case with a larger VLEN, ETISS becomes faster. This would also explain why ETISS executes `vww` the fastest for a VLEN of 1024. As the compiler generally tries to use the vector registers efficiently, it does not vectorize parts of the code that were compiled to vector instructions for smaller VLENs, instead compiling them to scalar instructions instead, for which simulation speeds are much higher. This "benefit" would obviously be a regression for actual hardware, and programs with vectorizable code have different optimal hardware configurations depending on commonly used data types or width and structure of neural network layers, among many other application-specific factors, which is examined in [11]. The performance of vector instructions specifically will be discussed further in the following Section 4.4.4.

4.4.4 Performance of Vector Instructions

While results from the previous Subsection show that ETISS with performance offers a great speedup compared to the RTL simulation, the simulation speed is still less than expected. To get an idea of whether Softvector is the bottleneck here, an example program that performs one billion iterations of `vadd.vv` was used to test implementations of that instruction in Softvector and pure CoreDSL, using TCC as the JIT-compiler. The version using CoreDSL only was found to be roughly 10x faster than the Softvector version, implying that for future work, much of Softvector could either be reimplemented in CoreDSL, or more consideration should be given to optimizations within the library. One example of a Softvector optimization was given in Section 3.1.4, where implementation of `vadd.vv` was also simplified by just casting the pointer to the vector field to the right data type, depending on SEW. This version performed even better than the CoreDSL implementation in some instances, with one test case showing a more than 30 % decrease in simulation time. However, these results are not definitive, as not enough testing was done for concrete statements, and both the CoreDSL version and the improved Softvector do not use the generic iteration approach shown in 3.1. Initial testing showed that passing functions to iterators incurs a measurable performance penalty compared to a compact implementation, where everything is calculated in one place. Again, the extent of this performance decrease and how it grows with increased program run time is left for future work.

Comparing `vww` with and without vectorization (~70 Million vs. ~100 Million instructions

4. EXPERIMENTAL RESULTS

respectively), the scalar version exhibits around a 4.5x speedup. That being said, QEMU suffers from the same problem. The same comparison shows that QEMU can execute the scalar version of vww roughly 10x faster than the vectorized version.

Conclusion and Future Work

5.1 Conclusion

To summarize this thesis, the ISS ETISS has been extended to fully support the RISC-V V vector extension. With the Softvector library implementing the behavior of vector instructions, it has been shown how ISA extensions can be integrated into ETISS efficiently with CoreDSL and external libraries.

Regarding performance estimation, with the inclusion of parallel pipelines into CorePerfDSL, it can be shown that VPUs can be modeled in a way such that estimated cycle counts are close to those of full RTL simulations while decreasing simulation times considerably. Results for benchmarks performed in this thesis (`toyCar`, `aww`, and `vww`) show a less than 5 % error in CPI for all benchmarks and hardware configurations. Additionally, key issues have been identified regarding model complexity and in turn potential improvements for the Vicuna 2.0 timing model and CorePerfDSL itself.

Lastly, considering the flexibility of ETISS and the performance estimator in addition to the increased simulation speed over Verilator, the point for using this approach for DSE for VPUs can be made, as we show that this ISS is capable of approximating cycle counts with high accuracy while retaining a significant advantage in simulation speed.

5.2 Future Work

Given that vector instructions seem to have a large negative impact on performance, the Softvector library should be examined further. One option would be to rewrite vector instructions in pure CoreDSL, or to profile Softvector itself and look for optimizations there, as the focus so far was on correctness over speed. Replacing the current approach of representing vector registers with special vector objects with a more basic approach seems to yield immediate improvements. Additionally, one could look into why using

GCC as the JIT-compiler performs worse than TCC, and if that disadvantage disappears for very long-running programs, think billions of instructions.

To further verify the accuracy of the Vicuna 2.0 CorePerfDSL model, more extensive testing is necessary. For one, only a subset of possible hardware configurations has been examined for this thesis. This includes different vector unit pipeline configurations, as computational units can be split up further, as well as additional VLENs, pipeline widths, and memory interface widths. Furthermore, the memory system has been kept very simple so far, assuming static delays of a single cycle and no caches. Currently, there is also a lack of coverage for different instruction patterns. As mentioned in Section 4.3.4, many programs execute only a limited number of patterns of vector instructions, which can hide model errors. Covering a wide range of execution patterns would require a testing framework that programmatically generates test cases similar to the testing suite mentioned in Section 4.1.

Finally, some additions to CorePerfDSL and the timing model could help to capture many complexities found in Vicuna 2.0. As of now, it is generally intended to have a separate description for each hardware configuration. However, the VPU introduces a number of parameters, each increasing the number of possible configurations exponentially. Those parameters can not only impact delay calculations, but also the pipeline structure itself, as, for example, VLEN has an impact on the length of the register unpacking pipeline. While there are ways around reproducing the same model for each possible configuration, e.g. supplementing external models with a hardware configuration at runtime, being able to instantiate different models via a dedicated mechanism in CorePerfDSL itself would simplify implementation. The timing behavior of concurrent processing of single vector instructions in multiple stages could be verified most easily by extending the external model that handles register unpacking. This approach was discussed in the first point in Paragraph 3.2.3.3 and promises a more accurate model for a modest amount of implementation effort and additional complexity.

Overview of Generative AI Tools Used

No AI tools were used in the writing of this thesis.



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

List of Figures

2.1	Dependency graph for the V extension	11
3.1	Proposed vector instruction implementation	22
3.2	Basic execution flow in Softvector	24
3.3	Simplified view of the CV32E40X with the Vicuna 2.0 VPU	26
3.4	An example of a pipeline with a subpipelined stage	28
3.5	An example for parallel pipelines	29
3.6	Example of how the shift registers work	33
3.7	Example stage and vector register timings for vector ALU instructions and vector stores	34
3.8	Example stage and vector register timings for vector DIV instructions and vector stores	35
3.9	Example stage and vector register timings for vector DIV instructions and vector stores, including a change in LMUL	36
3.10	Simulating multiple rounds of unpacking	37
3.11	Splitting instructions into sub-instructions	38
3.12	Solution to enforce temporal order in case read before write	39
3.13	Generating a stage ordering for CV32E40X + Vicuna 2.0	39
4.1	View of the comparison workflow	42



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

List of Tables

2.1	Brief description of Zve^* sub-extensions, taken from [16].	12
2.2	Minimal VLEN for each Zvl^* sub-extension, taken from [16].	12
4.1	Results for simple pattern load, <code>vadd.vv, store</code>	45
4.2	Results for simple pattern load, <code>vmul.vv, store</code>	45
4.3	Results for simple pattern load, <code>vdiv.vv, store</code>	46
4.4	Results for simple pattern load, <code>vredsum.vs, store</code>	46
4.5	Results for simple pattern <code>vmv.v.i, store</code>	47
4.6	Results for simple pattern load, <code>store</code>	47
4.7	Results for <code>toycar</code>	48
4.8	Results for <code>aww</code>	49
4.9	Results for <code>vww</code>	49
4.10	Simulation speeds for <code>toycar</code>	52
4.11	Simulation speeds for <code>aww</code>	53
4.12	Simulation speeds for <code>vww</code>	54



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

Bibliography

- [1] *Architecture Specification Language Reference*. [Online]. Available: <https://developer.arm.com/documentation/ddi0626/00bet2/?lang=en> (visited on 10/22/2025).
- [2] A. Armstrong, T. Bauereiss, B. Campbell, *et al.*, “ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS”, en, *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–31, Jan. 2019, ISSN: 2475-1421. DOI: 10.1145/3290384. [Online]. Available: <https://dl.acm.org/doi/10.1145/3290384> (visited on 10/22/2025).
- [3] *Minres/CoreDSL*, original-date: 2020-09-23T10:28:34Z, Feb. 2025. [Online]. Available: <https://github.com/Minres/CoreDSL> (visited on 03/31/2025).
- [4] M. Schlägl, C. Hazott, and D. Große, “RISC-V VP++: Next generation open-source virtual prototype”, in *Workshop on Open-Source Design Automation, 2024*.
- [5] D. Mueller-Gritschneider, M. Dittrich, M. Greim, K. Devarajegowda, W. Ecker, and U. Schlichtmann, “The Extendable Translating Instruction Set Simulator (ETISS) Interlinked with an MDA Framework for Fast RISC Prototyping”, in *2017 International Symposium on Rapid System Prototyping (RSP)*, ISSN: 2150-5519, Oct. 2017, pp. 79–84. [Online]. Available: <https://ieeexplore.ieee.org/document/8547814> (visited on 03/18/2025).
- [6] J. Kappes, R. Kunzelmann, K. Emrich, C. Foik, D. Mueller-Gritschneider, and W. Ecker, “Effective Processor Model Generation from Instruction Set Simulator to Hardware Design”, in *2023 IEEE Nordic Circuits and Systems Conference (NorCAS)*, Oct. 2023, pp. 1–7. DOI: 10.1109/NorCAS58970.2023.10305465. [Online]. Available: <https://ieeexplore.ieee.org/document/10305465> (visited on 09/17/2025).
- [7] *Veripool*. [Online]. Available: <https://www.veripool.org/verilator/> (visited on 10/20/2025).
- [8] C. Foik, D. Mueller-Gritschneider, and U. Schlichtmann, “CorePerfDSL: A Flexible Processor Description Language for Software Performance Simulation”, in *2022 Forum on Specification & Design Languages (FDL)*, ISSN: 1636-9874, Sep. 2022, pp. 1–8. DOI: 10.1109/FDL56239.2022.9925665. [Online]. Available: <https://ieeexplore.ieee.org/document/9925665> (visited on 01/31/2025).

- [9] C. Foik, R. Kunzelmann, D. Mueller-Gritschneider, and U. Schlichtmann, “Flexible Generation of Fast and Accurate Software Performance Simulators From Compact Processor Descriptions”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 11, pp. 4130–4141, Nov. 2024, Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, ISSN: 1937-4151. DOI: 10.1109/TCAD.2024.3445255. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/10745761> (visited on 01/31/2025).
- [10] M. Platzer and P. Puschner, “Vicuna: A Timing-Predictable RISC-V Vector Co-processor for Scalable Parallel Computation”, in *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, B. B. Brandenburg, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), ISSN: 1868-8969, vol. 196, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 1:1–1:18, ISBN: 978-3-95977-192-4. DOI: 10.4230/LIPIcs.ECRTS.2021.1. [Online]. Available: <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2021.1> (visited on 01/31/2025).
- [11] J. P. Jones, P. Van Kempen, and D. Mueller-Gritschneider, “Vicuna2.0: RISC-V Embedded Vector Unit with Half-Precision Floating-Point Support for TinyML”, in *2025 Austrochip Workshop on Microelectronics (Austrochip)*, ISSN: 2689-8144, Sep. 2025, pp. 69–72. DOI: 10.1109/Austrochip67945.2025.11183718. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/11183718> (visited on 10/15/2025).
- [12] *Chipsalliance/riscv-vector-tests*, original-date: 2022-11-03T14:51:50Z, Jan. 2025. [Online]. Available: <https://github.com/chipsalliance/riscv-vector-tests> (visited on 01/31/2025).
- [13] A. S. Waterman, “Design of the RISC-V Instruction Set Architecture”, en, Ph.D. dissertation, UC Berkeley, 2016. [Online]. Available: <https://escholarship.org/uc/item/7zj0b3m7> (visited on 03/11/2025).
- [14] *RISC-V FAQ*. [Online]. Available: <https://riscv.org/about/faq/> (visited on 03/12/2025).
- [15] *Riscv/riscv-isa-manual: RISC-V Instruction Set Manual*. [Online]. Available: <https://github.com/riscv/riscv-isa-manual> (visited on 03/12/2025).
- [16] *Riscvarchive/riscv-v-spec: Working draft of the proposed RISC-V V vector extension*. [Online]. Available: <https://github.com/riscvarchive/riscv-v-spec> (visited on 03/12/2025).
- [17] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, “Ara: A 1-ghz+ scalable and energy-efficient risc-v vector processor with multiprecision floating-point support in 22-nm fd-soi”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 530–543, 2020. DOI: 10.1109/TVLSI.2019.2950087.

- [18] *Chipsalliance/t1*, original-date: 2022-09-18T00:09:31Z, Oct. 2025. [Online]. Available: <https://github.com/chipsalliance/t1> (visited on 10/30/2025).
- [19] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, “Sonicboom: The 3rd generation berkeley out-of-order machine”, May 2020.
- [20] *Tenstorrent/riscv-ocelot: Ocelot: The Berkeley Out-of-Order Machine With V-EXT support*. [Online]. Available: <https://github.com/tenstorrent/riscv-ocelot> (visited on 10/30/2025).
- [21] C. Chen, X. Xiang, C. Liu, *et al.*, “Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline Out-of-Order 64-bit High Performance RISC-V Processor with Vector Extension : Industrial Product”, in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, May 2020, pp. 52–64. DOI: 10.1109/ISCA45697.2020.00016. [Online]. Available: <https://ieeexplore.ieee.org/document/9138983> (visited on 10/30/2025).
- [22] *Banana Pi BPI-F3 SpacemiT K1 RISC-V chip datasheet | BananaPi Docs*. [Online]. Available: https://docs.banana-pi.org/en/BPI-F3/SpacemiT_K1_datasheet (visited on 10/30/2025).
- [23] J. Zhu and D. Gajski, “An ultra-fast instruction set simulator”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, no. 3, pp. 363–373, Jun. 2002, ISSN: 1557-9999. DOI: 10.1109/TVLSI.2002.1043339. [Online]. Available: <https://ieeexplore.ieee.org/document/1043339> (visited on 10/16/2025).
- [24] F. Bellard, “QEMU, a fast and portable dynamic translator”, in *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA: USENIX Association, Apr. 2005. [Online]. Available: <https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator>.
- [25] *Coredsl 2.0*. [Online]. Available: <https://www.minres.com/work/coredsl/> (visited on 07/02/2025).
- [26] *Tum-ei-eda/M2-ISA-R-Perf*. [Online]. Available: <https://github.com/tum-ei-eda/M2-ISA-R-Perf/tree/main> (visited on 03/19/2025).
- [27] *Riscv-software-src/riscv-isa-sim*, original-date: 2011-08-26T20:00:24Z, Mar. 2025. [Online]. Available: <https://github.com/riscv-software-src/riscv-isa-sim> (visited on 03/12/2025).
- [28] M. Schlägl and D. Große, “Fast Interpreter-Based Instruction Set Simulation for Virtual Prototypes”, [Online]. Available: https://ics.jku.at/files/2025DATE_Fast_Interpreter-based_ISS.pdf (visited on 03/25/2025).
- [29] *Riscv-software-src/riscv-pk*, original-date: 2011-08-26T20:02:36Z, Oct. 2025. [Online]. Available: <https://github.com/riscv-software-src/riscv-pk> (visited on 10/28/2025).

- [30] V. Herdt, D. Große, P. Pieper, and R. Drechsler, “RISC-V based virtual prototype: An extensible and configurable platform for the system-level”, *Journal of Systems Architecture*, vol. 109, p. 101756, Oct. 2020, ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2020.101756. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1383762120300503> (visited on 10/28/2025).
- [31] M. Schlägl, C. Hazott, and D. Große, “RISC-V VP++: Next generation open-source virtual prototype”, in *Workshop on Open-Source Design Automation*, 2024. [Online]. Available: https://ics.jku.at/files/2024OSDA_RISCV-VP-plusplus.pdf (visited on 03/12/2025).
- [32] V. Herdt, D. Große, and R. Drechsler, “Fast and Accurate Performance Evaluation for RISC-V using Virtual Prototypes”, in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, ISSN: 1558-1101, Mar. 2020, pp. 618–621. DOI: 10.23919/DATE48585.2020.9116522. [Online]. Available: <https://ieeexplore.ieee.org/document/9116522> (visited on 03/31/2025).
- [33] SiFive, *SiFive E31 Core Complex Manual*, en. [Online]. Available: https://starfivetech.com/uploads/e31_core_complex_manual_21G1.pdf (visited on 10/28/2025).
- [34] F. Zaruba and L. Benini, “The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, Nov. 2019, ISSN: 1557-9999. DOI: 10.1109/TVLSI.2019.2926114.
- [35] N. Binkert, B. Beckmann, G. Black, *et al.*, “The gem5 simulator”, *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011, ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. [Online]. Available: <https://dl.acm.org/doi/10.1145/2024716.2024718> (visited on 10/29/2025).
- [36] N. Bruschi, G. Haugou, G. Tagliavini, F. Conti, L. Benini, and D. Rossi, “Gvsoc: A highly configurable, fast and accurate full-platform simulator for risc-v based iot processors”, in *2021 IEEE 39th International Conference on Computer Design (ICCD)*, 2021, pp. 409–416. DOI: 10.1109/ICCD53106.2021.00071.
- [37] R. Kassem, M. Briday, J.-L. Béchenec, Y. Trinquet, and G. Savaton, “Instruction Set Simulator Generation Using HARMLESS, a New Hardware Architecture Description Language”, in *Proceedings of the 2nd International Conference on Simulation Tools and Techniques for Communications, Networks and Systems*, Mar. 2009, p. 24. DOI: 10.1145/1537614.1537646.
- [38] B. Franke, “Fast cycle-approximate instruction set simulation”, in *Proceedings of the 11th international workshop on Software & compilers for embedded systems*, ser. SCOPES '08, New York, NY, USA: Association for Computing Machinery, Mar. 2008, pp. 69–78, ISBN: 978-1-4503-7843-7. DOI: 10.1145/1361096.1361109. [Online]. Available: <https://dl.acm.org/doi/10.1145/1361096.1361109> (visited on 03/31/2025).

- [39] J. R. Hauser, *Berkeley SoftFloat*. [Online]. Available: <https://www.jhauser.us/arithmetric/SoftFloat.html> (visited on 10/20/2025).
- [40] P. van Kempen, J. P. Jones, D. Mueller-Gritschneider, and U. Schlichtmann, “muRISCV-NN: Challenging Zve32x Autovectorization with TinyML Inference Library for RISC-V Vector Extension”, in *Proceedings of the 21st ACM International Conference on Computing Frontiers: Workshops and Special Sessions*, ser. CF '24 Companion, New York, NY, USA: Association for Computing Machinery, Jul. 2024, pp. 75–78, ISBN: 979-8-4007-0492-5. DOI: 10.1145/3637543.3652878. [Online]. Available: <https://dl.acm.org/doi/10.1145/3637543.3652878> (visited on 10/20/2025).