TU WIEN Informatics

Optimizing the OpenVADL ISS generator

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Informatik

eingereicht von

Florian Malicky
Matrikelnummer 12325228

an der Fakultät für Informatik

der Technischen Universität Wien

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

Wien, 4. November 2026

Florian Malicky	Andreas Krall

TU WIEN Informatics

Optimizing the OpenVADL ISS generator

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Informatics

by

Florian Malicky
Registration Number 12325228

to the Faculty of Informatics

at the TU Wien

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

Vienna, November 4, 2026

Florian Malicky	Andreas Krall

Erklärung zur Verfassung der Arbeit

Florian Malicky

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 „Übersicht verwendeter Hilfsmittel“ 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, 4. November 2026

Florian Malicky

Danksagung

Ich möchte mich bei Prof. Andreas Krall für seine Betreuung, Unterstützung, sein zügiges Feedback beim Verfassen dieser Arbeit und für die Möglichkeit, beim OpenVADL-Projekt mitzuarbeiten, bedanken. Weiters bedanke ich mich bei Johannes Zottele für die Unterstützung beim Arbeiten mit der OpenVADL-Codebase, das ausgezeichnete Feedback und die vielen Code-Reviews. Ich möchte mich auch bei Florian Freitag für seine konstruktiven Inputs und Hilfe beim Arbeiten mit dem OpenVADL-Frontend bedanken.

Acknowledgements

I would like to thank Prof. Andreas Krall for his guidance, support, prompt feedback during the writing of this thesis and the opportunity to work on the OpenVADL project. Furthermore, I would like to thank Johannes Zottele for his help in working with the OpenVADL codebase, his excellent feedback, and his numerous code reviews. Finally, I would like to thank Florian Freitag for his constructive input and help with working on the OpenVADL frontend.

Kurzfassung

OpenVADL ist eine quelloffene Implementierung der Prozessorbeschreibungssprache VADL (Vienna Architecture Description Language), die die Generierung einer Compiler-Toolchain sowie weiterer für den Prozessorentwurf notwendiger Werkzeuge ermöglicht. Aus einer einzigen Spezifikation lassen sich ein Compiler, Assembler, Linker, Instruction Set Simulator sowie weitere zugehörige Werkzeuge und Artefakte erzeugen. Die Generierung aus einer *single source of truth* vermeidet eine langwierige, fehleranfällige manuelle Implementierung und erlaubt zugleich eine schnelle Exploration des Entwurfsraums. Diese Arbeit konzentriert sich auf die Optimierung und funktionale Erweiterung des generierten Instruction Set Simulators, der auf dem Open-Source-Simulator QEMU basiert. Sie beschreibt die Implementierung von Unterstützung für *big-endian* und *bi-endian* Memory, für Program Counter, die auf General Purpose Register abgebildet sind, sowie für grundlegende Gleitkomma-Befehle. Zudem werden verschiedene Optimierungen und Änderungen am Generator erläutert, durch die der Instruction Set Simulator – im Vergleich zu manuell implementierten Simulatoren – sowohl korrekt als auch effizient arbeitet. Abschließend behandelt die Arbeit die Erweiterung des OpenVADL Frontends, um die Spezifikation von Gleitkomma-Operationen zu ermöglichen.

Abstract

OpenVADL is an open source implementation of the Processor Description Language VADL (Vienna Architecture Description Language), which enables the generation of a compiler toolchain and tools necessary for processor design. It can generate a compiler, assembler, linker, Instruction Set Simulator and other associated tools and artifacts from a single specification. Generating them from a *single source of truth* avoids slow, error-prone manual implementation and enables rapid design space exploration. This thesis focuses on the optimization and feature extension of the generated Instruction Set Simulator, which is based on the open source simulator QEMU. It describes the addition of support for big- and bi-endian memory, Program Counters that alias General Purpose Registers and basic floating-point instructions. It also describes a few optimizations and changes to the generator that make the Instruction Set Simulator both correct and efficient when compared to handwritten simulators. Lastly, the thesis also covers the extension of the OpenVADL frontend to allow the specification of floating-point operations.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**

2 Background and Previous Work **3**
2.1 Introduction to OpenVADL 3
2.2 The QEMU ISS generator 5
2.3 Floating Point Arithmetic 8

3 Implementation **13**
3.1 Big- and Bi-endian memory 13
3.2 Translation Block state registers 15
3.3 PC register alias 20
3.4 Side effect ordering 21
3.5 Floating point 26

4 Evaluation **31**
4.1 AArch32 31
4.2 RISC-V F/D extension 31

5 Future Work **33**

Overview of Generative AI Tools Used **35**

List of Figures **37**

List of Tables **39**

List of Algorithms **41**

Acronyms **43**

CHAPTER 1

Introduction

OpenVADL is an open source implementation of the Vienna Architecture Description Language (VADL), a Processor Description Language (PDL). It allows the specification of processor architectures and can generate various artifacts, such as an assembler, linker, LLVM Compiler Backend (LCB), hardware, Cycle-Approximate Simulator (CAS) and an Instruction Set Simulator (ISS). This enables rapid design space exploration [FHH+25].

In Chapter 2, we will take a closer look at how the ISS works. The goal is that the generated ISS is on par with hand-written simulators in terms of simulation speed. Eventually, it should be able to simulate instructions of various kinds, such as vector and Floating Point (FP) instructions. In Chapter 3, we will go over a few optimizations and feature additions for the ISS. Furthermore, we will go over the implementation of support for basic FP operations for the language frontend and ISS. Finally, these changes are evaluated in Chapter 4. This thesis builds on Johannes Zottele's work and his thesis *Automatic Generation of a QEMU based Instruction Set Simulator for OpenVADL* [Zot26].

CHAPTER 2

Background and Previous Work

Readers of this thesis are expected to have a good understanding of compilers and assembler languages [TC22, ALSU06]. They should be familiar with concepts such as just-in-time (JIT) compilation, Dynamic Binary Translation (DBT) and graph intermediate representations in compilers such as Control Flow Graphs (CFGs) and dependency graphs.

2.1 Introduction to OpenVADL

VADL allows the specification of processor architectures. A specification is split into six parts: an Instruction Set Architecture (ISA), an Application Binary Interface (ABI), a Micro Architecture (MiA), a Processor Definition (PD), a User Mode Emulation (UME) definition, and an Assembly Definition (AD). To generate an artifact, OpenVADL requires a certain subset of these parts, depending on the artifact. For example, to generate the ISS, it requires an ISA and a PD [FHH+25]. Figure 2.1 shows a general overview of the architecture of OpenVADL. For this thesis, only the Frontend, VADL Intermediate Architecture Model (VIAM) and ISS are relevant.

For a more detailed description of the OpenVADL ISS generator and VIAM, please read Johannes Zottele's thesis *Automatic Generation of a QEMU based Instruction Set Simulator for OpenVADL* [Zot26]. For a description of OpenVADL's parsing and frontend, please read Florian Freitag's thesis *An efficient compiler front-end for OpenVADL* [Fre26].

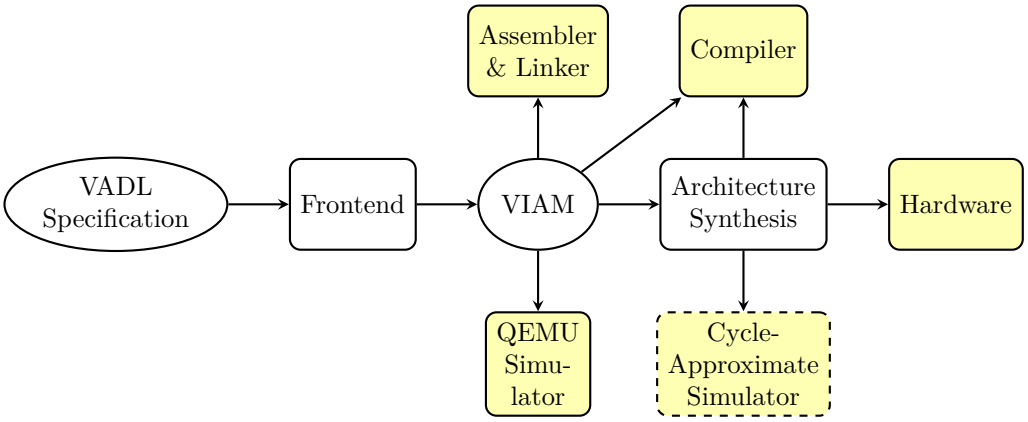


Figure 2.1: Overview of the OpenVADL architecture. Generated artifacts are in yellow. The Cycle-Approximate Simulator is not yet implemented [FHH+25].

```
instruction set architecture RV32I = {

    constant Size = 32          // architecture size is 32 bit

    using Byte  = Bits<8>       // 8 bit Byte
    using Inst  = Bits<32>      // instruction word type
    using Regs  = Bits<Size>    // register word type
    using Index = Bits<5>       // 5 bit register index
    using Addr  = Regs          // address type

    [ zero : X(0) ]             // register 0 is always zero
    register X : Index -> Regs  // integer register file
    memory MEM : Addr -> Byte   // byte addressed memory
    program counter PC : Addr   // current instruction

    format Itype : Inst =       // immediate instruction format
    { imm    : Bits<12>
    , rs1    : Index
    , funct3 : Bits<3>
    , rd     : Index
    , opcode : Bits<7>
    , immS = imm as SInt<Size>  // sign-extended immediate
    }

    instruction ADDI : Itype = X(rd) := X(rs1) + immS

    // further ISA definitions omitted
}
```

Listing 2.1: Example of a VADL ISA definition [Zot26].

2.1.1 Instruction Set Architecture Specification

In VADL, the ISA describes the architecture state, behavior, encoding and assembly representation of instructions [FHH+25]. Listing 2.1 shows what such a specification can look like. The common OpenVADL definitions `constant` and `using` can be used to define architectural constants and type aliases. The `format` definition is used to define the structure of instruction words through named fields. Format types can also be used to describe the structure of state and control registers. The processor state can be described using `register` definitions, as well as `memory` and `program counter` definitions. These definitions can be annotated to specify special behavior, such as the zero-register in the example. The `instruction` definition is used to describe the behavior of an instruction [Zot26].

2.1.2 Processor Definition

The Processor Definition is the second specification that OpenVADL needs, in addition to the ISA, to generate an ISS. It describes the concrete platform used by the ISS and provides information that is needed to start and run programs, such as reset behavior and memory regions. Listing 2.2 shows a PD with a reset procedure and two memory regions. The `MROM` region contains a small reset vector, which is specified with writes to the memory specified in the ISA [Zot26].

```
1  [ htif ]
2  processor Virt implements RV64I = {
3
4    reset = {
5      PC := 0x1000
6    }
7
8    [ firmware ]
9    [ base: 0x80000000 ]
10   memory region [RAM] DRAM in MEM
11
12   memory region [ROM] MROM in MEM = {
13     MEM<4>(0x1000) := 0x800002B7 // lui t0, 0x8000014
14     MEM<4>(0x1004) := 0x00028067 // jr t0
15   }
16 }
```

Listing 2.2: Processor Definition with reset behavior and memory regions [Zot26].

2.2 The QEMU ISS generator

Quick EMUlator (QEMU) is a generic and open source machine emulator and virtualizer [QEM26a, Bel05].

OpenVADL can generate a QEMU frontend, which integrates with the QEMU system to produce a functional ISS. This requires an Instruction Set Architecture (ISA) specification and a processor specification [FHH+25].

QEMU uses Dynamic Binary Translation (DBT) to translate target instructions to host instructions. Our frontend translates target code to Tiny Code Generator (TCG) instructions, QEMU's intermediate representation. A backend can then translate these into host instructions. Our generated frontend contains an instruction decoder, which is emitted by the VADL Decode Tree (VDT) generator. The decoder calls translation functions, which translate the target code to TCG code. One translation function is generated per target instruction.

```
static bool trans_addi(DisasContext *ctx, arg_addi *a) {
  trace_rv64im_instr_trans(__func__);
  TCGv_i64 reg_x_rd_dest = dest_x(ctx, a->rd);
  TCGv_i64 reg_x_rs1     = get_x(ctx, a->rs1);
  TCGv_i64 const_immS_n3 = tcg_constant_i64(a->immS);
  tcg_gen_add_i64(reg_x_rd_dest, reg_x_rs1, const_immS_n3);
  tcg_gen_mov_i64(reg_x_rd_dest, reg_x_rd_dest);
  return true;
}
```

Listing 2.3: Translation function for the RISC-V ADDI instruction.

We distinguish between the following execution phases:

- **Generation time**: The phase in which the QEMU frontend is generated by the OpenVADL compiler.
- **Translation time**: The phase during QEMU execution where the frontend generates TCG operations.
- **Runtime/Execution time**: The phase in which the translated native instructions are executed.

Furthermore, to define during which phase a value or some information is known, we use the following terms:

- **Static**: the value is known at translation time (e.g. instruction operands, immediate values)
- **Dynamic**: the value is only known at execution time (e.g. register contents)

CPU state

To keep track of the CPU state of the target architecture, QEMU maintains a struct, which stores register values and other information about the architectural state. Registers are usually accessed via a global TCG variable, which is associated with the in-memory location in the CPU state struct. During execution, register values are kept in host registers if possible, but sometimes must be spilled back to their canonical in-memory location. Both the canonical in-memory location and the TCG variable are called the TCG global of a register.

QEMU main execution loop

Target instructions are translated in bounded sequences of target instructions, so-called Translation Blocks (TBs), which are executed as a unit. TBs are similar to basic blocks, as both must end after branch instructions. TBs, however, must also end after instructions that raise exceptions and instructions that change certain parts of the CPU state (see Section 3.2).

When the QEMU main execution loop encounters a section of target code, it first checks if there already exists a compiled TB for that section. If not, it compiles the section until the TB must end. Once a TB exists for the section, it is executed. After the TB has finished execution, QEMU either jumps to another TB (which is called chaining) or returns to the main execution loop.

Larger TBs are generally better for performance, since fewer switches between the translated code and the execution loop are necessary. Larger TBs also expose more opportunities for QEMU's optimizations, which it performs on the TCG instructions.

TB Chaining

Returning to the main execution loop after a TB has finished execution is costly because the CPU state of the simulated program must be saved and the TB for the next instruction sequence must be looked up. QEMU therefore allows TBs to be chained together, which means that at the end of a TB there is a TCG instruction which directly jumps to the next TB. QEMU provides two jumpslots per TB, which can store addresses of TBs to jump to. Chaining to another TB is only possible if the Program Counter (PC) and relevant parts of the CPU state are always the same when entering that other TB. For example, after a direct jump (meaning the target address is static), the TB can be chained to the TB of the target address. After an indirect jump (i.e., jumping to a dynamic address), however, the TB must end and return to the main execution loop, as the next PC is not consistent across executions of the TB.

Helper functions

Some operations are inconvenient to express in TCG instructions. QEMU provides the means to call helper functions from TCG code in that case. Helper functions are regular

C functions that are compiled directly to host instructions as part of QEMU. They can then be called by emitting TCG call instructions, which jump out of the dynamically generated host code into the compiled C function. By default, before and after calling a helper, all TCG globals must be stored in their canonical location in the CPU state struct and later restored, such that the helper functions can read and modify them. Helper functions that do not access globals can be flagged to avoid this overhead. When helper functions have no side effects, the TCG compiler can even remove them entirely in some cases [QEM26c].

2.3 Floating Point Arithmetic

Floating Point (FP) arithmetic has a long history in computing. Today it is the most common method used to approximate real numbers in computers. The IEEE 754 standard is an attempt to standardize FP arithmetic by specifying its semantics in a strict but useful way. It defines various FP formats, rounding modes, operations and handling of special cases and exceptions. As of today, most ISAs support some version of FP arithmetic. Even though most try to adhere to IEEE 754, due to shortcomings of the standard and the backward compatibility requirement of many ISAs, most implementations have different semantics, which makes accurate and performant simulation of FP arithmetic challenging [IEE19, BJMM23, ZBJ+23].

2.3.1 IEEE 754

Since its introduction in 1985, the IEEE 754 standard has been expanded and changed. Each new version builds upon its predecessor, but the versions differ only in some details. IEEE 754-2019 is the most recent version and defines several binary and decimal formats. The most important formats are `binary32` and `binary64`. Figure 2.2 and Figure 2.3 show the binary encoding of the two formats.

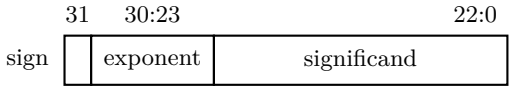


Figure 2.2: IEEE 754 binary32 layout

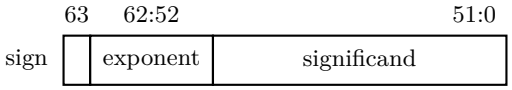


Figure 2.3: IEEE 754 binary64 layout

Each encoding consists of a single sign bit s, an exponent e of length E and a significand m of length M. Table 2.1 shows the different kinds of values that the encoding can represent.

The values of normal and subnormal numbers are calculated in almost the same way, but the normal values assume an implicit Most Significant Bit (MSB) of 1 in front of the significand, while subnormal values assume an implicit 0. Subnormal values have been introduced to close the gap between the smallest normal value and zero. Infinite values represent all values that are too big or too small to be represented by normal, subnormal or zero values. Operations on infinite values are supposed to still return reasonable results

Type	Condition	Value
Normal number	$1 \leq e \leq 2^E - 2$	$(-1)^s \times (1.m_{M-1}m_{M-2}\ldots m_0)_2 \times 2^{e-bias}$
Subnormal number	$e = 0 \land m \neq 0$	$(-1)^s \times (0.m_{M-1}m_{M-2}\ldots m_0)_2 \times 2^{1-bias}$
Zero	$e = 0 \land m = 0$	-0 if $s = 1$, $+0$ otherwise
Infinity	$e = 2^E - 1 \land m = 0$	$(-1)^s \times \infty$
NaN	$e = 2^E - 1 \land m \neq 0$	

Table 2.1: Types of IEEE 754 Floating Point values. Note that $bias = 2^{E-1} - 1$.

(e.g. the result of $-\infty \times -1$ should be $+\infty$). NaN values represent erroneous values and results of operations that do not make sense (e.g. $\infty \times 0$ or sqrt(-1)). Since for NaN values, only the exponent has a fixed value, it can have a payload in its significand. That payload cannot be zero, however, since that would make the value an infinite value. IEEE 754-2019 specifies that there should be two kinds of NaN values: Quiet NaN (qNaN) and Signaling NaN (sNaN). When an operation takes an sNaN, it raises an *Invalid Operation* exception and converts it to a qNaN, though there are exceptions to this. The exact encoding of the two NaN variants was only formally standardized in IEEE 754-2008. Because of that, many ISAs encode NaN values differently. IEEE 754-2008 specifies that the MSB of the significand must be 1 for qNaN values and 0 for sNaN values. IEEE 754 also recommends propagating NaN payloads through operations [IEE19].

Rounding modes

When an operation is performed, the exact result may not be representable in the chosen encoding. In that case, the rounding mode defines which representable value is delivered instead. If a result has to be rounded, the result is called inexact. Table 2.2 lists all rounding modes the standard defines [IEE19].

Float exceptions

Table 2.3 lists the five float exceptions that IEEE 754 defines. Some ISAs define additional float exceptions, which are not specified in the standard, such as the *Input Denormal* exception from AArch32 [ARM13, IEE19].

2.3.2 Other Floating Point formats

Beyond IEEE 754, new specialized FP formats are being introduced, which are tailored to suit new kinds of workloads (such as image processing and deep learning). These minifloat and microfloat formats sacrifice precision and dynamic range for a smaller memory representation and more energy efficiency (this is known as approximate or transprecision computing). Some formats, such as `bfloat16`, trade precision for more dynamic range by having a larger exponent and a smaller significand field [MSB+22, KMM+19, CFBS25].

Rounding mode	Description
Round to nearest, ties to even	Delivers the representable value closest to the exact result. If two representable values are equally close, the one with an even Least Significant Bit (LSB) is delivered.
Round to nearest, ties to away	Delivers the representable value closest to the exact result. If two representable values are equally close, the one with the larger magnitude is delivered.
Round towards zero	Delivers the closest representable value which is not greater in magnitude than the exact result.
Round towards $+\infty$	Delivers the closest representable value which is not smaller than the exact result.
Round towards $-\infty$	Delivers the closest representable value which is not greater than the exact result.

Table 2.2: IEEE 754 Floating Point rounding modes

Exception	Description
Invalid Operation	The operands are invalid for the operation (e.g. $0 \times \infty$).
Division by Zero	The result of an operation is exact and infinite (e.g. $0/0$).
Overflow	The result of an operation is inexact and infinite.
Underflow	The result of an operation is subnormal and not zero.
Inexact	The result of an operation is inexact.

Table 2.3: IEEE 754 Floating Point exceptions. Note that the exact conditions for when exceptions are raised are more complicated and are more closely specified in the standard [IEE19].

Many SIMD extensions of modern ISAs relax the requirements of IEEE 754 to achieve better performance or to reduce memory usage. This includes things like flushing subnormal values to zero (operations with subnormal values take significantly more cycles than with normal values), which was already supported by ARM's VFP architecture [ARM05] or mixed precision operations (for example vector widening FP instructions in the RISC-V V extension, whose operands have a smaller FP format than the result [RIS26b, Chapter 31]).

There are also attempts to develop alternatives to the IEEE 754 formats. Unums are an alternative FP format that allows the exponent to vary in length. Numbers near 1 have more accuracy this way, while numbers near 0 and very large numbers have less accuracy. This increases precision in common calculations, which rarely use very small or very large numbers. Despite that, Unums have not yet been adopted by the industry. Posits (Type III Unums) are the latest unum format introduced [GY17].

2.3.3 Floating Point instruction emulation

There is a wide variety of approaches to simulate FP instructions, all with different goals and constraints. There is a clear tradeoff between simulation speed and accuracy. Simulating FP instructions such that all behavior of the target ISA is exactly replicated may be too slow and unnecessary for some simulators. Our generated ISS, however, must behave exactly as described by the VADL specification. There are generally two approaches to simulating FP instructions, although mixed approaches are also common:

1. Simulation using integer and bit operations (usually slower, but capable of simulating exact behavior).
2. Mapping target instructions to host FP instructions (faster, but more restricted and in many cases not applicable).

2.3.4 Floating Point libraries

The Berkeley SoftFloat library is an IEEE 754-compliant simulation library relying purely on integer instructions. As a result, it is precise but slow [Hau18]. QEMU uses and adapts it for its own use case. It adds support for more non-IEEE 754 formats, such as x86 80-bit extended precision. It also adds a fast path, which maps target instructions to host instructions in many cases, making it precise and fast for most emulated software [QEM19].

FloppyFloat is a library specifically built for speeding up FP simulation in ISSs. It utilizes both integer and host FP instructions, similar to QEMU, but manages to use the host FP path in more cases. It achieves speedups of more than fivefold compared with Berkeley SoftFloat for individual instructions [ZBL25].

FlexFloat is an FP library for transprecision computing. It allows users to simulate custom float formats by specifying exact mantissa and exponent lengths. It also leverages host FP instructions and truncation to speed up simulation, which makes it more than twice as fast as Berkeley SoftFloat in some cases [TMB20].

The FP library CPFloat specializes in simulating low-precision float arithmetic. It offers built-in support for many low-precision formats, rounding modes and mathematical functions while providing a considerable speedup over other libraries [FM23].

SoftPosit is a C library that can be used to emulate arithmetic with the proposed Posit number format [Leo20].

GNU MPFR is an integer-based arbitrary-precision binary FP library, which provides correct rounding and exceptions. It is used by many other software projects as a math library or as a reference for testing their own libraries [FHL+07].

Compilers such as GCC and Clang use integer-based FP runtime libraries to enable running FP operations on targets that do not support hardware FP instructions. GCC's

runtime library `libgcc` uses `soft-fp`, a library which supports FP exceptions, rounding modes, and target-specific NaN values. It covers many standard and extended IEEE 754 FP formats [Fre]. Clang's runtime library `compiler-rt` is quite similar and also contains heavily optimized target-specific assembly routines [LLV].

CHAPTER 3

Implementation

3.1 Big- and Bi-endian memory

An ISA defines, among other properties, the representation of data in memory. A key aspect of this representation is endianness, which determines in what order bytes are stored in memory. Generally, three different cases must be considered. Little-endian stores lower bytes first in memory, whereas big-endian stores upper bytes first in memory. In bi-endian systems, memory can be read and written in both little- and big-endian, usually depending on a control register or the instruction.

Our solution for OpenVADL is to introduce an annotation, which can be applied to memory definitions. It defines the endianness used by all read and write operations on that memory. The annotation can optionally contain a condition, which makes the memory bi-endian. The condition is used to decide what endianness is used. Listing 3.1 shows how the annotation is used to define that the memory is little-endian if and only if the `le` bit of the `MSR` register is set.

```
[ little endian : MSR.le = 1 ]
memory MEM : Address -> Byte

[ execution state : sf, pr, le ]
register MSR : MSRFormat

format MSRFormat : Reg64 =
{ // ...
, sf : Bit
, pr : Bit
, le : Bit
}
```

Listing 3.1: Bi-endian memory annotation and execution state annotation in the 64-bit PowerPC specification [Sch26].

The following section describes the implementation of endianness for the ISS. The implementation for other artifacts of OpenVADL is not covered in this thesis.

3.1.1 VIAM pass

To apply the annotation to the actual memory operations (i.e., memory reads and writes), the new ISS-specific VIAM pass `IssApplyMemoryEndiannessPass` is introduced. By default, all memory operations are little-endian. If a big-endian annotation on a memory definition is present, all operations interacting with that memory are changed to operate in big-endian. If the annotation is bi-endian, the pass inserts a new ISS-specific VIAM node `IssStaticEndianConditionNode`, which represents the bi-endian condition result. It then duplicates the memory operation node and uses the new bi-endian condition node to decide which of the two memory operation nodes to use. This way, all nodes are either little- or big-endian. For read nodes, a `SelectNode` is inserted, which depends on the condition and selects between the two new read nodes (see Figure 3.1). Since write nodes are side effects, control flow nodes need to be inserted for an if-else construct, where each of the two branches contains one of the write nodes (see Figure 3.2). Note that round nodes are expression nodes in the data dependency graph of the VIAM and rectangular nodes are control flow nodes. Black edges are data dependencies and red edges are control flow edges.

3.1.2 Bi-endian condition constraints

During simulation, the ISS needs to evaluate the condition. The naive approach would be to evaluate the condition for every encountered `IssStaticEndianConditionNode` at runtime, which means that the branching also needs to happen at runtime. That would lower the performance of the ISS for memory instructions, which is why the constraint that the condition may only depend on TB-state registers is enforced (see Section 3.2). These registers are considered static at translation time, which means that the decision

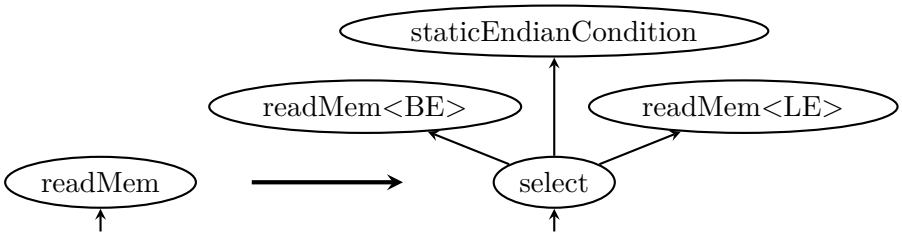


Figure 3.1: Transformation of a bi-endian memory read node into two duplicate read nodes, whose endianness is known during compilation.

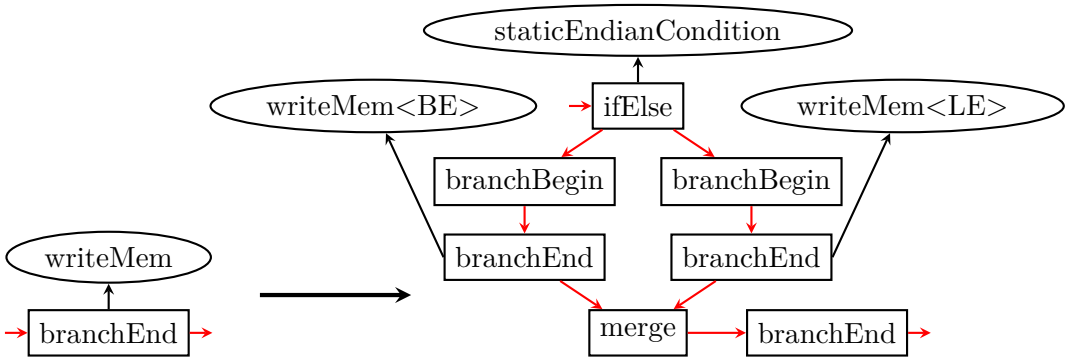


Figure 3.2: Transformation of a bi-endian memory write node into two duplicate write nodes, whose endianness is known during compilation.

about which endianness to use can be made at translation time. When starting a new TB, that condition is evaluated once (see Listing 3.4). Listing 3.5 shows the use of the static endian condition in a translation function. The static endian condition is also used when loading instructions, where memory byte order is also relevant. OpenVADL currently only supports one memory declaration, which means that at most one endianness condition is present, but this method also works for multiple memory declarations.

3.1.3 Memory region initialization

A problem that remains is memory region initialization in bi-endian memory. Register contents cannot be read during that phase in QEMU. The solution is to take the register values from the CPU reset procedure, where the registers used in the condition must be explicitly set to some constant value.

3.2 Translation Block state registers

In many ISAs, instruction behavior depends on control registers or other special-purpose registers. These registers dictate if and how certain operations are performed (e.g. privilege mode, memory endianness, bitsize of operations). Reading these status registers and performing a runtime check for each affected instruction is costly. The value of

these registers usually does not change very often or is always the same for a given code section. The simulation performance can be improved by reading these registers once at the start of a TB, and then treating them as static during translation. This also means that a TB must end when these registers change. The register contents must be written into the TB-state, and the corresponding static values must be provided during translation. When a TB is looked up, QEMU not only looks for the correct PC, but also checks whether the current CPU state matches the values of these registers stored in the TB-state. Going forward, these registers will be called TB-state registers, though usually only parts of the registers are stored in the TB-state.

This method is already used by most handwritten QEMU frontends [QEM26c] and improves the performance of certain instructions significantly (for example the integer arithmetic instructions of our PowerPC specification [Sch26]).

In OpenVADL, the new register annotation `[ execution state ]` is added, which marks parts of or whole registers as TB-state registers. In certain places (like the bi-endian memory condition), it is enforced that all involved registers are TB-state registers. Listing 3.2 shows how the annotation can be used to mark a whole register to be TB-state. In Listing 3.1, the annotation is used to mark only three bits of a register to be TB-state. Due to the limited space provided by QEMU in the TB-state, it is only possible to annotate 32 bits. Listing 3.3 and Listing 3.4 show how our generated code saves and restores TB-state registers from the TB-state.

```
1 [ execution state ]
2 register R : Bits<16>
```

Listing 3.2: Annotating a whole TB-state register.

Read operations on TB-state registers are analyzed by the ISS-specific VIAM pass `IssRegisterAccessLoweringPass`. When a read only accesses annotated bits, then the read node is converted to an `IssStaticReadRegNode`, which is later lowered to a translation-time register read. But if a read operation accesses unannotated parts of the register, the entire operation must remain a runtime register read. Listing 3.5 shows a translation function that contains a translation-time register read from the translation context.

```
1  static inline void cpu_get_tb_cpu_state(CPUPPC64SFSState *env,
2        vaddr *pc, uint64_t *cs_base, uint32_t *pflags) {
3    *pc = env->PPC64SFS_PC;
4    *cs_base = 0;
5
6    uint32_t flags = 0;
7    uint64_t off = 0;
8
9    flags |= ((env->msr >> 63) & 0x1ULL) << off; // sf
10   off += 1;
11   flags |= ((env->msr >> 14) & 0x1ULL) << off; // pr
12   off += 1;
13   flags |= ((env->msr >>  0) & 0x1ULL) << off; // le
14   off += 1;
15   *pflags = flags;
16 }
```

Listing 3.3: Encoding the values of TB-state registers from Listing 3.1 into the TB-state (see lines 9 until 14) in `cpu.h`.

```
1  static void ppc64sfs_tr_init_disas_context(DisasContextBase *db,
2        CPUState *cs) {
3    DisasContext *ctx = container_of(db, DisasContext, base);
4    CPUPPC64SFSState *env = cpu_env(cs);
5    PPC64SFSCPUClass *mcc = PPC64SFS_CPU_GET_CLASS(cs);
6    PPC64SFSCPU *cpu = PPC64SFS_CPU(cs);
7
8    ctx->env = env;
9
10   uint32_t flags = ctx->base.tb->flags;
11   uint64_t off = 0;
12   ctx->tb_state.msr = 0;
13   ctx->tb_state.msr |= ((flags >> off) & 0x1ULL) << 63; // sf
14   off += 1;
15   ctx->tb_state.msr |= ((flags >> off) & 0x1ULL) << 14; // pr
16   off += 1;
17   ctx->tb_state.msr |= ((flags >> off) & 0x1ULL) <<  0; // le
18   off += 1;
19
20   ctx->endian_cond = VADL_equ(
21       VADL_slice((ctx->tb_state.msr), 1, 0, 0), 1,
22       ((uint8_t) 0x1 ), 1
23   );
24 }
```

Listing 3.4: Restoration of the values of TB-state registers (from Listing 3.1) from the TB-state (see lines 10 until 18) and calculation of the bi-endian condition (see line 20) in `translate.c`.

```
1  static bool trans_stb(DisasContext *ctx, arg_stb *a) {
2    trace_ppc64sfs_instr_trans(__func__);
3    TCGv_i64 tmp_n32_0 = tcg_temp_new_i64();
4    TCGv_i64 tmp_n10_0 = tcg_temp_new_i64();
5    TCGv_i64 reg_x_rs = get_x(ctx, a->rs);
6    TCGv_i64 reg_x_ra = get_xz(ctx, a->ra);
7    TCGv_i64 const_n9_n9 = tcg_constant_i64(VADL_concat(
8        VADL_sextract(VADL_slice((ctx->tb_state.msr), 1, 63, 63),1),
9        32, ((uint32_t) 0xffffffff ), 32
10   ));
11   TCGv_i64 const_immS_n2 = tcg_constant_i64(a->immS);
12   if ((ctx->endian_cond)) {
13     tcg_gen_add_i64(tmp_n10_0, reg_x_ra, const_immS_n2);
14     tcg_gen_and_i64(tmp_n10_0, tmp_n10_0, const_n9_n9);
15     tcg_gen_extract_i64(tmp_n32_0, reg_x_rs, ((uint32_t) 0x0 ),
16         ((uint32_t) 0x8 ));
17     tcg_gen_qemu_st_i64(tmp_n32_0, tmp_n10_0, 0, MO_SIGN|MO_8|MO_LE);
18   } else {
19     tcg_gen_add_i64(tmp_n10_0, reg_x_ra, const_immS_n2);
20     tcg_gen_and_i64(tmp_n10_0, tmp_n10_0, const_n9_n9);
21     tcg_gen_extract_i64(tmp_n32_0, reg_x_rs, ((uint32_t) 0x0 ),
22         ((uint32_t) 0x8 ));
23     tcg_gen_qemu_st_i64(tmp_n32_0, tmp_n10_0, 0, MO_SIGN|MO_8|MO_BE);
24   }
25
26   return true;
27 }
```

Listing 3.5: Translation function for the PowerPC STB (store byte) instruction. Line 8 shows the use of the `sf` bit from the TB-state saved register `MSR` in a static (translation-time) expression. Line 12 shows the use of the static endian condition to decide at translation-time which TCG memory operation to emit (see lines 17 and 23).

With this change, it is also necessary to change when TBs are ended and chained. In particular, when a TB-state register is written to, the TB must end, because its static value stored in the translation context becomes stale. If the written value is dynamic during translation time, chaining is not possible, and it is necessary to return to the main execution loop and perform a lookup for the next TB.

3.2.1 Translation-time TB chaining and ending

During translation, operations that are only executed under certain conditions may be encountered. These conditions can be static or dynamic. In case of a dynamic condition, it must be conservatively assumed that these operations can always be executed. If executing such an operation would cause the current TB to end, the TB must always be ended. If, however, the condition is static (meaning all values the condition depends on are known at translation time), it can be decided during translation whether the TB must be ended.

Up until now, static conditions have been treated the same as dynamic conditions. The AArch32 instruction set shows how that leads to a performance loss. In that ISA, the PC is part of the GPRs [ARM05], which means that any instruction that can write to a GPR can cause a jump. Whether the PC is written depends on an instruction operand, which means that the condition is static. Our old implementation assumes that the PC write can always happen and ends TBs after every instruction writing to a GPR.

The new implementation defers the decision whether to end the TB to translation time in some instruction translation functions. It does this by keeping track of a state during translation. The state is modified by the translation function whenever a relevant TCG instruction is emitted. The state contains three predicates that are initially set to the following truth values: `end_tb = false`, `can_chain = true` and `jmpslt_free = true`.

Direct jump

When the translation function emits a direct jump, it sets `end_tb = true`. When `can_chain` and `jmpslt_free` are both `true`, it emits a chain instruction using jumpslot 1 and sets `jmpslt_free = false`. Otherwise, it emits a TCG instruction that returns to the main execution loop and performs a TB lookup.

Indirect jump

The translation function sets `end_tb = true` and emits a TCG instruction that returns to the main execution loop and performs a TB lookup.

Exception raise

The translation function sets `end_tb = true` and emits a TCG helper call, which handles the exception.

TB-state write

The translation function sets `end_tb = true`. If the written value is dynamic at translation time, it also sets `can_chain = false`.

Instruction end

If the instruction contains an unconditional jump or exception raise, then the translation function does not need to emit any more TCG instructions to end the TB, since the end can never be reached. Otherwise, if the instruction contains any jumps or writes, the state must be read. If `end_tb` is not set, no instructions are emitted. Otherwise, if `can_chain` is set, a chain instruction using jumpslot 0 is emitted. If `can_chain` is not set, an exit instruction is emitted, which returns to the main execution loop.

All this causes a slight performance loss during translation. So emitting this logic into translation functions must be avoided whenever possible. In particular, if the state is

never read, it does not need to be tracked. When a direct jump is emitted, the state is always read. But for the instruction end, the state is only read if no unconditional jump or unconditional exception raise is present in the instruction.

3.3 PC register alias

As mentioned previously, the PC is a GPR in the AArch32 ISA and should be specifiable in OpenVADL. A PC should be accessible both explicitly and implicitly (via the GPR). Listing 3.6 shows how this can be done using a program counter alias.

```
register R : Bits<4> -> Bits<32>

[next next]
alias program counter PC : Addr = R(15)

/* ... */ {
  PC         // explicitly reads the next next PC
  PC.current // explicitly reads the current PC
  R(15)      // implicitly reads the next next PC
}
```

Listing 3.6: Reading the PC explicitly via an alias and implicitly.

In many ISAs, reading the PC yields an address that points not to the current, but to the next instruction (or the instruction after that), as in ARM [ARM05]. In OpenVADL, there are two language features to describe that behavior: annotating a PC definition (see Listing 3.7) or using a subcall when reading the PC (see Listing 3.8). When using both, the subcall overwrites the offset of the annotation. An important detail is that the address stored inside the PC always points to the current instruction.

```
[ next ]
program counter PC : Bits<32>
```

Listing 3.7: Annotating the PC to yield the address of the next instruction.

```
R := PC.next
```

Listing 3.8: Using a PC subcall to read the address of the next instruction.

The frontend applies the subcall offsets while lowering to VIAM. The offsets depend on the length of the current instruction, which is sometimes not known during compilation (e.g. in exception procedures). Therefore a new expression node, `InstructionWidthNode`, is added, which represents the width of the current instruction in bytes. This node must then be handled by the artifact generators.

To apply the annotation offsets, it must be determined which read operations access the PC. The artifact-agnostic VIAM-pass `CounterAccessResolvingPass` looks at register

read/write nodes and analyzes the indices used to access register files. When the indices are constant, the pass can trivially determine whether the operation accesses the PC. When the indices are not constant, the pass uses a technique similar to the one used by the `IssApplyMemoryEndiannessPass` (see Section 3.1.1). It duplicates the node and inserts a check that compares the indices of the operation with the indices of the PC and decides whether to use the old read/write or the duplicate. Later passes can then safely assume that the first operation will never touch the PC, while the duplicate operation is always a PC read/write.

The implicit offsets are then applied by the artifact-agnostic VIAM-pass `PcOffsetPass`, also using the `InstructionWidthNode`. In instruction procedures, where the instruction width is known, this pass also resolves all occurrences of `InstructionWidthNode`. Other occurrences cannot currently be handled correctly by the ISS. They are being replaced by the concrete instruction width of the first instruction in the ISA, which means that only ISAs with a single instruction width are correctly compiled. To support multiple different instruction lengths, the length of the current instruction must be passed to exception handlers somehow (see Section 5.0.1).

3.4 Side effect ordering

Target instructions may cause several kinds of side effects during execution. These side effects may also be conditional (e.g. PC changes in conditional branch instructions) or may be executed in loops (e.g. write operations to register chunks in vector instructions). The semantics of VADL dictate that all side effects must be executed during an instruction, regardless of order.

During ISS generation, PC changes are lowered to jumps, which exit the instruction. This is problematic if other side effects are scheduled after the PC change. All PC changes (and all side effects which exit instructions in general) must therefore be moved behind all other side effects.

Furthermore, memory read and write operations may trigger I/O events when accessing MMIO memory, which requires the TB to end (the exact reason for that can be found in QEMU's documentation [QEM26b]). Whether the memory operation accesses an MMIO region is only known at execution time, which means that QEMU jumps out of the dynamically generated code, re-compiles the TB and re-executes the current instruction. That means that any changes to global register variables happening before the triggering memory operation incorrectly persist into the re-execution. To avoid this, changes to global register variables before any memory operations must be disallowed. Memory writes are side effects, and thus can be reordered. Memory reads, on the other hand, are not side effects, and their scheduling is handled separately (see Section 5.0.4).

3.4.1 Side effect reordering pass

The problem of side effect ordering is unique to the ISS; therefore, an ISS-specific VIAM pass called `IssSideEffectReorderingPass` is introduced, which analyzes and restructures the CFG of instruction behaviors such that all side effects are executed in the correct order. Side effects are grouped into three types, and a rank is assigned to each type (see Table 3.1).

Type	Rank	Description
MEM	0	Memory write operations. These must be scheduled before other side effects.
SE	1	Side effects that are not in MEM or PC (e.g. normal register writes).
PC	2	Side effects causing an instruction exit (direct and indirect jumps, raising exceptions). These must be scheduled after other side effects.

Table 3.1: Side effect types and ranks for reordering

The input of the pass is a CFG, which can contain `if`, `if-else` and `forall` blocks. These blocks can be nested and can contain side effects. Side effects are at this point not fully scheduled and are stored in unordered lists at branch ends (see Listing 3.9). A block may contain one or more branches, of which only one is executed. A branch in turn may contain any number of blocks. The side effects of a branch are only those stored at the branch end, not the ones inside the blocks nested within the branch. The side effects of a block are all side effects of all branches and blocks inside the block.

After reordering, the CFG should not be unnecessarily large and all side effects must still be executed under the same conditions and for the same number of loop iterations as in the original CFG. The pass first restructures the CFG such that no sibling blocks are conflicting. Sibling blocks are blocks inside the same branch. Then, the side effects inside each block can be scheduled such that they are ordered, which is done at a later stage by the `SideEffectSchedulingPass`.

Two blocks are conflicting if, regardless of their order, no ordering of the side effects is possible (e.g. the two blocks on lines 6 and 12 in Listing 3.9 are conflicting). To resolve conflicting blocks, side effects must be extracted out of them until they are no longer conflicting. Extracting a side effect means duplicating its containing blocks, and moving the side effect into the duplicates (see Listing 3.10, Listing 3.11 and Algorithm 3.1). Since each extraction introduces new control flow operations, the number of extractions must be minimized.

```
1  if (...) {
2    forall (...) {
3      side_effects: { SE }
4    }
5    if (...) { // type rank interval: 0..2
6      side_effects: { MEM, PC }
7    } else {
8      side_effects: { SE, PC }
9    }
10   if (...) { // type rank interval: 0..1
11     side_effects: { MEM, SE }
12   }
13   side_effects: { SE }
14 }
```

Listing 3.9: Example input control flow to the `IssSideEffectReorderingPass`. Side effects are stored as unordered lists at branch ends. The blocks on line 6 and 12 are conflicting.

```
1 // before extraction
2 if (a) { forall (b) { side_effects: { MEM, SE, PC } } }
3
4 // after extracting MEM
5 if (a) { forall (b) { side_effects: { SE, PC } } }
6 if (a) { forall (b) { side_effects: { MEM } } }
```

Listing 3.10: Example of a side effect extraction. Containing blocks must be duplicated and the extracted side effect placed into the duplicates.

```
1  // before extraction
2  if (a) {
3    if (b) { side_effects: { MEM, SE } }
4    if (c) { side_effects: { MEM, SE } }
5  }
6
7  // after extraction
8  if (a) {
9    if (c) { side_effects: { MEM } }
10   if (b) { side_effects: { MEM, SE } }
11   if (c) { side_effects: { SE } }
12 }
```

Listing 3.11: Example of a side effect extraction to resolve a conflict. Only the inner blocks are duplicated, because the conflict happens on the inner branch level.

To compute which blocks are conflicting, the interval that the side effect types in a block span is determined. The interval simply spans from the least type rank to the biggest type rank in the block (see Listing 3.9 for examples). Two blocks with intervals $a..b$ and $c..d$ conflict if and only if $a < d \wedge c < b$. Furthermore, all `forall` blocks with more than one side effect type conflict with themselves.

Finding the best extraction decisions is computationally expensive and depends on the exact goal (smallest number of control flow nodes, best runtime performance). We therefore decide to use a heuristic approach, which is more than sufficient for typical instruction CFGs: When two or more blocks are conflicting, the block that contains the most side effects is kept. Keeping blocks with smaller intervals is also preferred, because this usually allows more of the other blocks to be kept. Also, when side effects need to be extracted out of a block, one side effect type can always be left inside. We decide to keep the side effect type that has the most occurrences in the block (keeping the type which requires the fewest blocks to be duplicated would also be an option). Keeping a block with the interval 0..2 is only possible if no side effects of type SE exist in the containing branch and no other blocks contain side effects of type SE.

Algorithm 3.2 shows the procedure for reordering the side effects inside a branch. When reordering the CFG of an instruction behavior, ReorderBranch is simply called for the base branch with $depth = 0$.

Algorithm 3.1: Side Effect Extraction

```
1  procedure ExtractOutOf(block, blocks, depth)
2      keptType ← most common type in block;
3      foreach sideEffect in block do
4          if sideEffect_type ≠ keptType then
5              sideEffectDepth ← branch depth of sideEffect;
6              levelsToDuplicate ← sideEffectDepth − depth;
7              newBlock ← DuplControlFlow(levelsToDuplicate, sideEffect);
8              MoveSideEffect(sideEffect, block, newBlock);
9              Add(blocks, newBlock);
10         end
11     end
12 end
```

Algorithm 3.2: Branch Reordering

```
1  procedure ReorderBranch(branch, depth)
2      blocks ← FindBlocksIn(branch);   ▷ Sorted descending by side effect count
3      foreach block in blocks do
4          if block is a forall-block ∧ |block_types| > 1 then
5              ExtractOutOf(block, blocks, depth);
6          end
7      end
8      keptBlocks ← init empty list;                    ▷ Select which blocks to keep
9      memSeUsed ← sePcUsed ← false;
10     foreach block in blocks do
11         if block_interval is MEM..SE then
12             memSeUsed ← true;
13             Add(keptBlocks, block);
14             break;
15         end
16     end
17     foreach block in blocks do
18         if block_interval is SE..PC then
19             sePcUsed ← true;
20             Add(keptBlocks, block);
21             break;
22         end
23     end
24     if ¬memSeUsed ∧ ¬sePcUsed ∧ branch_sideEffects contains no SE then
25         foreach block in blocks do
26             if block_interval is MEM..PC ∧ no other block contains SE then
27                 Add(keptBlocks, block);
28                 break;
29             end
30         end
31     end
32     foreach block in blocks \ keptBlocks do      ▷ Extract out of all other blocks
33         ExtractOutOf(block, blocks, depth);
34     end
35     foreach block in keptBlocks do                ▷ Recurse on the kept blocks
36         foreach b in block_branches do
37             ReorderBranch(b, depth + 1);
38         end
39     end
40     ReorderBlocks(blocks);
41 end
```

3.5 Floating point

This section describes the prototype implementation of Floating Point (FP) support for the OpenVADL frontend and ISS. It covers all features needed to fully specify the RISC-V extensions `F` and `D`, which add 32- and 64-bit FP support respectively [RIS26b, IEE19]. The following features are implemented and configurable:

- IEEE 754-2019 float types `binary32` and `binary64`.
- All five IEEE 754-2019 rounding modes.
- Sticky (accrued) and non-sticky float exception flags.
- OpenVADL-built-ins for all RISC-V float instructions that cannot easily be expressed using existing OpenVADL features.

Everything else is either hardcoded or not yet implemented.

3.5.1 Frontend support

To specify float operations, new built-in float functions are introduced. A wide range of float operations should eventually be supported. Given the number of existing formats and operation semantics, we decide against introducing separate built-ins per format and semantics. Instead, functions for the basic operations are introduced, and the semantics and input/result formats are determined at the callsite by the types of the arguments and type parameters.

Float Type

The new built-in type `FloatType` is a subtype of `DataType` and holds information about a float format. Listing 3.12 shows how to declare a new `FloatType`. The `float-type` definition must be annotated to attach information about the float format. The annotation `[ IEEE : <size> ]` makes the defined float type represent one of the IEEE 754 binary formats.

```
1 [ IEEE : 32 ]
2 float-type Binary32
3
4 [ IEEE : 64 ]
5 float-type Binary64
```

Listing 3.12: Definition of the two IEEE 754 float types binary32 and binary64 (the annotation adds both encoding and semantic information).

Internally, `FloatType` is a subtype of `DataType`, but not a subtype of `BitsType`. Implicit type conversions between `FloatType` and other types are not possible. This prevents programmers from using `FloatType` expressions in places where `BitsType` or integers are expected. Implicit conversion from `BitsType` to `FloatType` is nonsensical, since there is no information about a float format, which the `FloatType` needs. `FloatType` can only be explicitly cast to `BitsType` of the same bit length. This is because there is no single useful way to truncate or extend a `FloatType`. `FloatType` can be explicitly cast from `BitsType` of an equal or higher bit length. When casting from a longer `BitsType`, the value is truncated to the length of the `FloatType`.

Built-in Type Parameters

VADL's type system supports inferring the result type of built-in functions from the input types. But that is not sufficient here, since for some float functions (e.g. conversion functions), the result format cannot be inferred from the input formats. Because of that, we make some built-in functions take type parameters. Listing 3.13 shows the signatures of two exemplary float built-in functions and how to use them. Type parameters must be listed before the parentheses in angle brackets.

```
1  // built-in function signatures
2  function fadd( a : FloatType, b : FloatType, rm : Bits<3> ) -> FloatType
3  function fcvt< T : FloatType | SInt<N> | UInt<N> >
4      ( a : FloatType | SInt<N> | UInt<N>, rm : Bits<3> ) -> T
5
6  // 32-bit float addition
7  VADL::fadd(rs1 as Binary32, rs2 as Binary32, rm)
8
9  // conversion from 64-bit float to 32-bit signed integer
10 using SInt32 = SInt<32>
11 VADL::fcvt<SInt32>(rs1 as Binary64, rm)
```

Listing 3.13: Parameterization of float built-ins using type parameters in angle brackets.

3.5.2 Float exception flags

Many ISAs store float exception flags in status registers. The flags themselves can either be sticky or non-sticky. Sticky means that the flags are only ever set but never unset by float operations (though these flags can usually be overwritten with a special instruction). For example, PowerPC has both sticky and non-sticky float exception flags [Fou21]. In OpenVADL, registers can be annotated to specify where float exception flags are stored (see Listing 3.14).

```
[ sticky fe flag invalid     : nv ]
[ sticky fe flag div_by_zero : dz ]
[ sticky fe flag overflow    : of ]
[ sticky fe flag underflow   : uf ]
[ sticky fe flag inexact     : nx ]
register FCSR : FCsrFormat

format FCsrFormat : Bits<32> =
{ reserved [31..8]
, frm      [7..5]  // Rounding mode
, nv       [4]     // Float exception flag: Invalid operation
, dz       [3]     // Float exception flag: Division by zero
, of       [2]     // Float exception flag: Overflow
, uf       [1]     // Float exception flag: Underflow
, nx       [0]     // Float exception flag: Inexact
}
```

Listing 3.14: Float exception flag annotations in the RISC-V F and D extension specification.

3.5.3 Float support for the ISS

QEMU simulates FP operations using integer and host FP instructions. QEMU specifically uses a modified version of the Berkeley SoftFloat library [Hau18, QEM19] to also use the hardware FP instructions of the host in some cases. Using the softfloat approach is generally slower, but often difficult to avoid, since the semantics of host and target may differ and acquiring certain information (e.g. float exception flags) is difficult or inefficient using hardware instructions. When the rounding mode is *round to nearest, ties to even* and certain float exception flags are already set (and do not need to be recomputed), then QEMU uses hardfloat. This is why sticky flags are generally better for performance in the long run, since once they are set (and usually they are set pretty fast), hardfloat can be used [ZBJ+23]. For most ISAs, QEMU's approach is more than sufficient in terms of performance. Therefore, we will also use it for OpenVADL's ISS.

QEMU's float library can only be called from helper functions, and there are no FP TCG instructions, so a helper is generated for each used float built-in. First, all instruction graphs are analyzed and it is determined which float built-ins are called. For each built-in, a set of all used tuples of argument types and type parameters is collected. Afterward, it is possible to generate helper functions only for existing tuples. Listing 3.15 shows what such a generated helper function looks like.

```
// helper.h
DEF_HELPER_FLAGS_3(binary64_32_fcvtfs, TCG_CALL_NO_RWG,
                   i64, env, i64, i64)

// helper.c
uint64_t helper_binary64_32_fcvtfs(CPURV64IDState *env,
                                   uint64_t rs1, uint64_t rm) {
    float_status *s = &env->fp_status;
    set_float_rounding_mode(rm, s);
    uint32_t result = float64_to_int32(rs1, s);
    return result;
}
```

Listing 3.15: Generated C helper function for the 64-bit float to 32-bit signed integer conversion built-in call in Listing 3.13. The helper does not modify any TCG globals, so it is flagged with `TCG_CALL_NO_RWG` (no read/write globals).

Most of QEMU's float functions take a pointer to a `float_status` value, which contains information about how the function is to be performed. Things like rounding, NaN propagation and NaN representation can be configured this way. It also stores float exception flags, which are treated as sticky, meaning that flags are only computed when they are not set, which speeds up the operations significantly [ZBJ+23].

Our generated translation functions then emit calls to these helper functions. The parameters of the helper functions include not only the float operands but also the rounding mode, though that could be subject to change once more parameters (e.g. for NaN or subnormal handling) need to be added.

Lazy float exception flags

QEMU uses a different encoding for float exception flags than most ISAs. Most float operations can set float exception flags. It is beneficial to determine which flags are already set before performing a float operation, because hardfloat can be used if certain flags do not need to be computed. Therefore, flags should also be read before most float operations. A naive approach would be to convert between QEMU's encoding and that of the target ISA before and after each float operation. This is terrible for performance, however, and single-operation benchmarks run about 50% slower than the upstream RISC-V implementation this way. We therefore use the same approach as the upstream implementation, which stores the flags in the CPU state using QEMU's encoding and only converts them when the value is actually required in the target encoding. In RISC-V, the flags can only be read and modified by accessing the `FCSR` register, which happens much less frequently than regular float operations [RIS26b].

To achieve this, the concept of lazy registers is introduced for the ISS generator. These registers contain lazy parts, which are not stored in the TCG global, but are sourced from other parts of the CPU state. When a register only contains lazy parts, the TCG

global can even be removed entirely. Reading and modifying these registers becomes more difficult. When the value is read, a helper function needs to be called, which constructs the register value. When the value is written, a similar helper function is needed to deconstruct the written value.

For the float exception flags, the parts of registers containing these flags are considered lazy. When a register value is read, the flags stored in QEMU's format are converted and inserted into the lazy parts of the register value. When a value is written, the lazy parts are extracted and converted back. Sticky float flags are stored in `env->fp_status` and non-sticky flags are stored in `env->ns_fe_flags` (see Listing 3.16).

Since TCG code can no longer access lazy registers through their globals, a helper function call needs to be emitted every time such a register is read or written. This does slow down certain instructions significantly, but overall, the performance of the simulator increases (see Section 4.2).

```
// cpu.h
struct CPUArchState {
  uint64_t x[32];
  uint64_t f[32];
  uint64_t fcsr;
  uint64_t pc;

  // ...

  float_status fp_status; // float status and sticky FE flags
  uint16_t ns_fe_flags;   // non-sticky FE flags
}
```

Listing 3.16: Example CPU state containing fields for sticky and non-sticky float exception (FE) flags. `ns_fe_flags` is only emitted when the ISA contains non-sticky flags.

CHAPTER 4

Evaluation

4.1 AArch32

The ISS generated from our specification of the AArch32 instruction set (Armv6) was benchmarked using the Embench IoT benchmark suite (version 1.0) [Emb]. For AArch32, the suite was compiled using GCC 14.2.1 and newlib, and FP instructions using software emulation.

The results in Figure 4.1 show that our generated ISS is on par with QEMU's handwritten simulator in most benchmarks. Our simulator even performs up to 20% better in some cases, but also over 20% worse in others. This is likely due to the TCG instruction selection of OpenVADL, which is not always optimal. The TB ending and chaining decisions made by the ISS generated by OpenVADL are very close to the ones made by the handwritten implementation. The handwritten simulator is also more complex and supports more features, which may explain why it performs worse than OpenVADL's ISS in some benchmarks.

4.2 RISC-V F/D extension

The ISS generated from our 64-bit RISC-V specification including the I base instruction set and the M, F and D extensions (RV64IMD) passes the RISC-V Architectural Certification Tests [RIS26a]. We ran Embench benchmarks containing a significant single- and double-precision FP workload for it (see Figure 4.2). They were compiled using GCC 16.1.0 and newlib.

The `st` benchmark performs double-precision FP calculations and the `minver` benchmark performs single-precision FP calculations. The `nbody` benchmark was excluded due to its short runtime. The ISS generated by OpenVADL simulates FP instructions very similarly

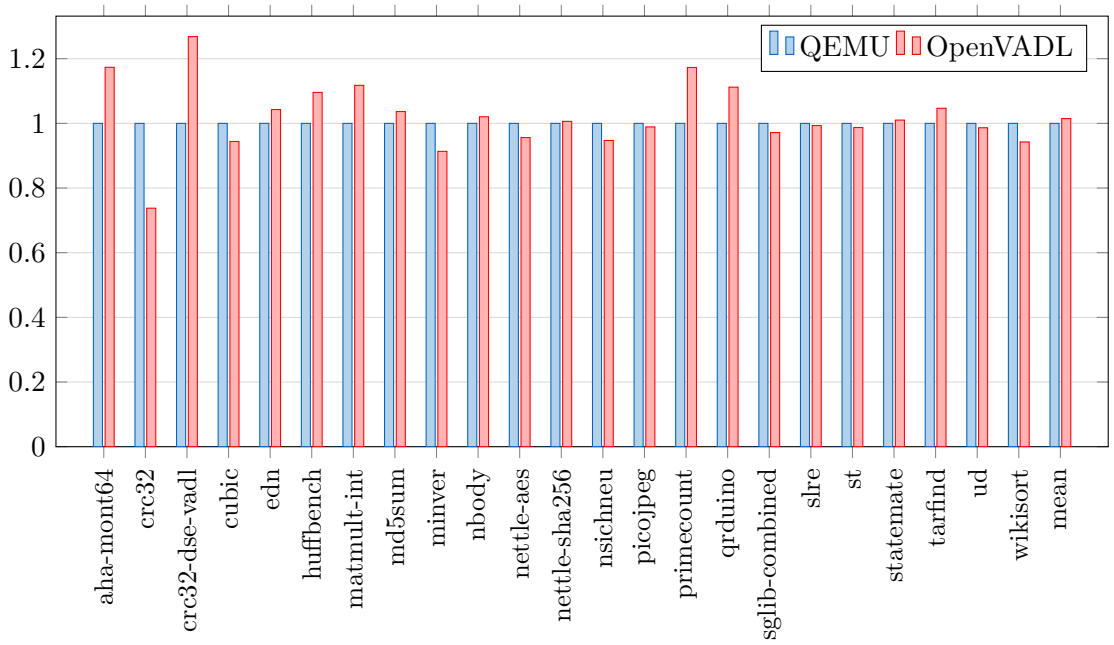


Figure 4.1: Runtimes of Embench benchmarks on the ISS generated by OpenVADL for AArch32 relative to QEMU (lower is better).

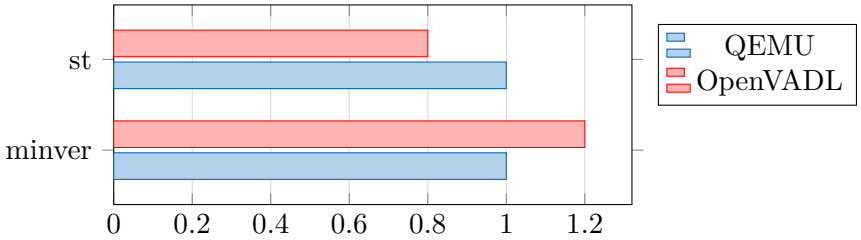


Figure 4.2: Runtimes of Embench benchmarks with significant single- and double-precision FP workloads on the ISS generated by OpenVADL for RV64IMD relative to QEMU (lower is better).

to the handwritten simulator. A key difference is how the two simulators handle NaN-boxing. The handwritten simulator uses helper functions while OpenVADL uses TCG instructions. The latter requires branching TCG instructions and is slower. NaN-boxing is necessary when single-precision instructions are performed with double-precision float registers, such as in the `minver` benchmark.

CHAPTER 5

Future Work

5.0.1 Variable instruction widths

Many ISAs have instructions that are of different lengths. ARM's T32 instruction set is one example [ARM13]. To fully support generating an ISS for such instruction sets, not only an appropriate instruction decoder is needed, but variable instruction lengths must also be considered when calculating offsets from the PC. Handling the `InstructionWidthNode` introduced in Section 3.3 in exception procedures is required for that.

5.0.2 Static and dynamic expression analysis

Expressions are static if they can be fully evaluated at translation time. This means that they do not need to be compiled to TCG instructions. Expressions can also be partially static. Recognizing which parts of an expression are static can therefore benefit the performance of the generated ISS in the following ways:

- Fewer TCG instructions are generated.
- Static expressions are compiled by the C compiler, which can employ more sophisticated optimizations than the TCG optimizer.
- The TB chaining and ending logic introduced in Section 3.2.1 benefits from static conditions.

Currently only the trivial case where expressions are completely static is recognized. A straightforward improvement would be to take apart `if` blocks with conjunct conditions into nested `if` blocks (see Listing 5.1).

```
// the whole condition is currently conservatively treated as dynamic
if (static && dynamic) { ... }

// separate into
if (static) { if (dynamic) { ... } }
```

Listing 5.1: Example separation of a partially static condition to expose optimization opportunities.

This analysis can be especially beneficial if the programmer who writes the ISA specification does not have a deep understanding of the generated ISS. Expressions can, of course, be structured by the programmer in a way that keeps static and dynamic parts separate.

5.0.3 Broad float support

This thesis only covers the implementation of very basic FP instruction support, for which the generated ISS is efficient. While many additional features can theoretically be expressed using existing OpenVADL features, the generation of an efficient ISS is not guaranteed. This includes, for example, custom NaN values and flushing denormal values to zero. QEMU's float library provides functionality for many common FP features, but is not very configurable beyond that. To be able to support a broad range of formats and features, OpenVADL will have to either have its own FP library or use existing ones.

5.0.4 Scheduling Memory Loads before other Side Effects

As mentioned in Section 3.4, memory operations should not occur after write operations to register globals, because that causes incorrect behavior in some special cases. The side effect reordering takes care of memory write operations, but cannot handle memory read operations, because these are not side effects. Memory read operations can have multiple users, which may be scheduled in different places in the CFG. They are therefore currently scheduled at the last control node that dominates all user nodes. The scheduling logic must be modified in a way that also considers register writes.

Furthermore, the ISS generator currently performs a liveness analysis on the TCG variables used in an instruction and uses graph coloring to reassign variables to operations in order to reduce the number of TCG variables used. This in turn reduces register pressure, but allows existing register global variables to be used as temporary storage [Zot26]. This is fine in the general case, but can be a source of error if a register global is written to before a memory operation. A solution would be to exclude register global variables from the reassignment when memory operations are present in the instruction.

Overview of Generative AI Tools Used

Generative AI tools were used to generate Figure 2.1, to assist in creating figures and plots and to spell-check this thesis. For recreating figures, prompts similar to the example shown in the following listing were used:

```
Recreate the attached image of a figure from an older OpenVADL paper in latex and print out the latex code. Use a minimal amount of packages.
```

To create templates for figures, prompts similar to the one shown in the following listing were used. The generated LaTeX code was usually heavily modified by hand afterwards.

```
Using latex and tikz, create a figure containing two example control flow graphs.
Nodes in the CFG should be boxes and edges should be directional arrows colored red.
Between the two CFGs, draw a thick black arrow from left to right.
```

To spell-check this thesis, the following prompt was used:

```
The attached PDF file is my bachelor thesis in computer science. Please check the german and english text for spelling mistakes, incorrect punctuation and grammatical errors.
Create a table with a row for each error:
- The first column should contain the page number of the error.
- The second should contain the original sentence, with the contained error highlighted in bold.
- The third column should contain the corrected sentence, with the changed parts highlighted in bold.
For long sentences, only the part containing the error must be shown.
```

The following models were used: OpenAI's ChatGPT (GPT-6) and Google's Gemini (3.1 Pro).

List of Figures

2.1 Overview of the OpenVADL architecture 4
2.2 IEEE 754 binary32 layout 8
2.3 IEEE 754 binary64 layout 8

3.1 Transformation of bi-endian memory read node 15
3.2 Transformation of bi-endian memory write node 15

4.1 Embench benchmarks of AArch32 relative to QEMU 32
4.2 Embench benchmarks of RV64IMD relative to QEMU 32

List of Tables

2.1 Types of IEEE 754 Floating Point values 9
2.2 IEEE 754 Floating Point rounding modes 10
2.3 IEEE 754 Floating Point exceptions 10

3.1 Side effect types and ranks for reordering 22

List of Algorithms

3.1 Side Effect Extraction 24
3.2 Branch Reordering 25

Acronyms

ABI Application Binary Interface. 3

AD Assembly Definition. 3

CAS Cycle-Approximate Simulator. 1, 4

CFG Control Flow Graph. 3, 22, 24, 34

DBT Dynamic Binary Translation. 3, 6

FP Floating Point. xv, 1, 8–12, 26, 28, 31, 32, 34

GPR General Purpose Register. xi, xiii, 19, 20

ISA Instruction Set Architecture. 3–6, 8–11, 13, 15, 19–21, 27–30, 33, 34

ISS Instruction Set Simulator. xi, xiii, xv, 1, 3, 5–7, 11, 14, 16, 21, 22, 26, 28, 29, 31–34

JIT just-in-time. 3

LCB LLVM Compiler Backend. 1

LSB Least Significant Bit. 10

MiA Micro Architecture. 3

MMIO Memory-Mapped I/O. 21

MSB Most Significant Bit. 8, 9

NaN Not a Number. 9, 29, 44

OpenVADL OpenVADL. xi, xiii, xv, 1, 3–6, 13–16, 20, 26–28, 31, 32, 34

PC Program Counter. xi, xiii, xv, 7, 16, 19–21, 33

PD Processor Definition. 3, 5

PDL Processor Description Language. xiii, 1

QEMU Quick EMUlator. xi, xiii, xv, 5–8, 11, 15, 16, 21, 28–32, 34, 37

qNaN Quiet NaN. 9

SIMD Single Instruction Multiple Data. 10

sNaN Signaling NaN. 9

TB Translation Block. xv, 7, 14–19, 21, 31, 33

TCG Tiny Code Generator. 6–8, 18, 19, 28–34

UME User Mode Emulation. 3

VADL Vienna Architecture Description Language. xi, xiii, 1, 3–6, 11, 21, 27, 43, 44

VDT VADL Decode Tree. 6

VIAM VADL Intermediate Architecture Model. 3, 14, 16, 20–22

Bibliography

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson, 2nd edition, 2006. https://www.pearson.com/en-us/subject-catalog/p/compilers-principles-techniques-and-tools/P200000003472/9780321486813. Accessed: 2026-11-04.

[ARM05] ARM Limited. *ARM Architecture Reference Manual*, 2005. https://support.arm.com/documentation/ddi0100/i/. Accessed: 2026-11-04.

[ARM13] ARM Limited. *Arm Architecture Reference Manual for A-profile architecture*, 2013. https://support.arm.com/documentation/ddi0487/. Accessed: 2026-11-04.

[Bel05] Fabrice Bellard. QEMU: A Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, Anaheim, CA, USA, 2005. USENIX. https://www.usenix.org/legacy/event/usenix05/tech/freenix/full_papers/bellard/bellard.pdf. Accessed: 2026-11-04.

[BJMM23] Sylvie Boldo, Claude-Pierre Jeannerod, Guillaume Melquiond, and Jean-Michel Muller. Floating-point arithmetic. *Acta Numerica*, 32:203–290, 2023. doi:10.1017/S0962492922000101.

[CFBS25] Brian Chmiel, Maxim Fishman, Ron Banner, and Daniel Soudry. FP4 All the Way: Fully Quantized Training of LLMs, 2025. doi:10.48550/arXiv.2505.19115.

[Emb] Embench Task Group. Embench: A modern embedded benchmark suite. https://www.embench.org/. Accessed: 2026-10-30.

[FHH+25] Florian Freitag, Linus Halder, Simon Himmelbauer, Christoph Hochrainer, Benedikt Huber, Benjamin Kasper, Niklas Mischkulnig, Michael Nestler, Philipp Paulweber, Kevin Per, Matthias Raschhofer, Alexander Ripar, Tobias Schwarzinger, Johannes Zottele, and Andreas Krall. The Vienna Architecture Description Language, 2025. doi:10.48550/arXiv.2402.09087.

[FHL+07] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2):13–es, June 2007. doi:10.1145/1236463.1236468.

[FM23] Massimiliano Fasi and Mantas Mikaitis. CPFloat: A C library for simulating low-precision arithmetic. *ACM Trans. Math. Softw.*, 49(2), June 2023. doi:10.1145/3585515.

[Fou21] OpenPOWER Foundation. *Power ISA Version 3.1B*. OpenPOWER Foundation, September 2021. `https://openpowerfoundation.org/specifications/isa/`. Accessed: 2026-11-04.

[Fre] Free Software Foundation. Software floating point in GCC. `https://gcc.gnu.org/wiki/Software_floating_point`. Accessed: 2026-10-11.

[Fre26] Florian Freitag. An efficient compiler front-end for OpenVADL. Master's thesis, TU Wien, Vienna, Austria, 2026. In preparation.

[GY17] John L. Gustafson and Isaac T. Yonemoto. Beating Floating Point at its Own Game: Posit Arithmetic. *Supercomputing Frontiers and Innovations*, 4(2):71–86, April 2017. doi:10.14529/jsfi170206.

[Hau18] John R. Hauser. Berkeley SoftFloat Release 3e. `https://github.com/ucb-bar/berkeley-softfloat-3`, January 2018. Accessed: 2026-10-14.

[IEE19] IEEE Standard for Floating-Point Arithmetic. IEEE Standard Std 754-2019, IEEE, 2019. Revision of IEEE Std 754-2008. doi:10.1109/IEEESTD.2019.8766229.

[KMM+19] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. A Study of BFLOAT16 for Deep Learning Training, 2019. doi:10.48550/arXiv.1905.12322.

[Leo20] S. H. Leong. SoftPosit, March 2020. Zenodo. doi:10.5281/zenodo.3709035.

[LLV] LLVM Project. compiler-rt Runtime Library. `https://compiler-rt.llvm.org/`. Accessed: 2026-10-11.

[MSB+22] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. FP8 Formats for Deep Learning, 2022. doi:10.48550/arXiv.2209.05433.

[QEM19] QEMU Project. Features/Softfloat. https://wiki.qemu.org/Features/Softfloat, June 2019. Accessed: 2026-10-14.

[QEM26a] QEMU Project. About QEMU. https://www.qemu.org/docs/master/about/index.html, October 2026. Accessed: 2026-10-14.

[QEM26b] QEMU Project. *TCG Instruction Counting*, 2026. Available at: https://www.qemu.org/docs/master/devel/tcg-icount.html. Accessed: 2026-10-14.

[QEM26c] QEMU Project. *Translator Internals*, 2026. https://www.qemu.org/docs/master/devel/tcg.html. Accessed: 2026-10-14.

[RIS26a] RISC-V International. RISC-V Architectural Certification Tests. https://github.com/riscv/riscv-arch-test, 2026. Accessed: 2026-10-09.

[RIS26b] RISC-V International. *The RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture*, January 2026. https://docs.riscv.org/reference/isa/v20260120/unpriv/unpriv-index.html. Accessed: 2026-11-04.

[Sch26] Kevin Schlosser. Specification of the PowerPC Architecture in VADL. https://www.complang.tuwien.ac.at/vadl/papers/SchlosserFinal.pdf, April 2026. Bachelor's thesis, Technische Universität Wien. Accessed: 2026-11-04.

[TC22] Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2022. doi:10.1016/C2014-0-01395-0.

[TMB20] Giuseppe Tagliavini, Andrea Marongiu, and Luca Benini. FlexFloat: A Software Library for Transprecision Computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(1):145–156, 2020. doi:10.1109/TCAD.2018.2883902.

[ZBJ+23] Niko Zurstraßen, Nils Bosbach, Jan Moritz Joseph, Lukas Jünger, Jan Henrik Weinstock, and Rainer Leupers. Efficient RISC-V-on-x64 Floating Point Simulation. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*, pages 558–565, 2023. doi:10.1109/ICCD58817.2023.00090.

[ZBL25] Niko Zurstraßen, Nils Bosbach, and Rainer Leupers. FloppyFloat: An Open Source Floating Point Library for Instruction Set Simulators. In *2025 Design, Automation & Test in Europe Conference (DATE)*, pages 1–6, 2025. doi:10.23919/DATE64628.2025.10992803.

[Zot26] Johannes Zottele. Automatic Generation of a QEMU based Instruction Set Simulator for OpenVADL. Master's thesis, TU Wien, Vienna, Austria, 2026. In preparation.