TU WIEN Informatics

Spezifikation der PowerPC-Architektur in VADL

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Kevin Schlosser
Matrikelnummer 12119892

an der Fakultät für Informatik

der Technischen Universität Wien

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

Wien, 6. April 2026

Kevin Schlosser	Andreas Krall

TU WIEN Informatics

Specification of the PowerPC Architecture in VADL

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Kevin Schlosser
Registration Number 12119892

to the Faculty of Informatics

at the TU Wien

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

Vienna, April 6, 2026

Kevin Schlosser	Andreas Krall

Erklärung zur Verfassung der Arbeit

Kevin Schlosser

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, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 6. April 2026

Kevin Schlosser

Acknowledgements

I would like to thank Prof. Andreas Krall for his guidance, feedback, and excellent communication throughout this project. I would also like to thank the entire VADL team for their support, with special thanks to Florian Freitag for helping me get started and Johannes Zottele for his assistance with the various issues I encountered during development.

Kurzfassung

Die manuelle Entwicklung von Befehlssatz-Simulatoren ist zeitaufwändig und fehleranfällig. OpenVADL löst dieses Problem, indem Simulatoren automatisch aus einer High-Level-Prozessorspezifikation generiert werden. Im Rahmen dieser Arbeit werden die Ausdrucksstärke und Leistung von OpenVADL evaluiert, indem ein Simulator für die skalare Festkomma-Teilmenge der 64-Bit Power-Befehlssatzarchitektur implementiert wird. Power ist eine RISC-Architektur mittlerer Komplexität mit zwei Ausführungsmodi, mehreren modusabhängigen Flags sowie verschiedenen Befehlsvarianten, was sie zu einem geeigneten Maßstab für diese Evaluierung macht. Der generierte Simulator wird anschließend gegen QEMU getestet: zum einen auf Korrektheit durch den Vergleich von Befehlsergebnissen, zum anderen auf Leistung durch die Messung der Ausführungsgeschwindigkeit. Die Ergebnisse zeigen, dass der generierte Simulator für die meisten Befehle eine mit QEMU vergleichbare Leistung erzielt.

Abstract

Building instruction set simulators manually is time-consuming and error-prone. OpenVADL addresses this by generating simulators automatically from a high-level processor specification. In this thesis, the expressiveness and performance of OpenVADL are evaluated by implementing a simulator for the scalar fixed-point subset of the 64-bit Power Instruction Set Architecture. Power is a RISC architecture of moderate complexity, featuring two execution modes, several mode-dependent flags, and multiple instruction variants, making it a suitable benchmark for this evaluation. The generated simulator is then tested against QEMU: for correctness by comparing instruction results and for performance by measuring execution speed. Results show that the generated simulator achieves performance similar to QEMU for most instructions.

Contents

CHAPTER 1

Introduction

Instruction Set Simulators (ISSs) are important tools in processor development and software engineering. They allow programs to run on an architecture before physical hardware exists and serve as reference implementations for testing and verification. However, building one manually is time-consuming, and the effort grows with architectural complexity, as does the likelihood of specification errors. Achieving high simulation performance through techniques such as dynamic binary translation adds further complexity to this process.

Processor Description Languages (PDLs) offer an alternative by allowing processors to be described at a high level, from which tools such as simulators, assemblers, and compilers can be generated automatically. Vienna Architecture Description Language (VADL) is one such language, developed at TU Wien, with OpenVADL serving as its open-source implementation.

This thesis evaluates OpenVADL through the implementation of a simulator for the 64-bit Power Instruction Set Architecture (ISA) Scalar Fixed-Point Compliancy Subset (SFS). Power ISA is a good fit for this evaluation, as its two execution modes, multiple mode-dependent flag registers, and structured system of instruction variants introduce additional complexity to the specification.

The generated simulator is validated against QEMU (an open-source machine emulator and virtualizer) by cross-checking outputs on automatically generated test cases covering every instruction. Performance is evaluated separately by measuring execution speed across a set of handwritten assembly programs.

The remainder of this thesis is organized as follows. Chapter 2 introduces VADL and the Power ISA, along with an overview of related work. Chapter 3 covers language features and the structure of a processor specification. Chapter 4 presents the PPC64 implementation. Validation is covered in Chapter 5; Chapter 6 then evaluates VADL's

macro system and the performance of the generated simulator. Chapter 7 concludes the thesis.

CHAPTER 2

Background

This chapter introduces VADL and the Power ISA, and concludes with an overview of related work.

2.1 VADL

VADL is a PDL developed as a research project at TU Wien. From a single processor specification, a VADL implementation can generate assemblers, compilers, linkers, synthesizable hardware descriptions, and functional and cycle-approximate ISSs. There are two implementations: a proprietary original and OpenVADL, its open-source counterpart. OpenVADL's source code is available on GitHub.

A VADL processor specification is divided into sections, as shown in Listing 2.1. Only the sections relevant to the artifacts to be generated need to be defined.

instruction set architecture: Defines architectural elements and instructions.

application binary interface: Defines calling conventions and register usage.

assembly description: Defines the syntax of assembly instructions.

micro architecture: Describes the processor pipeline.

processor: Contains processor initialization code.

```
instruction set architecture RV64IM  = {}

application binary interface ABI for RV64IM = {}

assembly description Assemble implements RV64IM for ABI = {}

micro architecture FiveStage for RV64IM = {}

processor CPU implements RV64IM with FiveStage for ABI = {}
```

Listing 2.1: VADL Specification Structure

2.2 Power ISA

2.2.1 History

In the mid-1970s, IBM developed the IBM 801, widely considered the first modern Reduced Instruction Set Computer (RISC) processor by removing many memory-targeting instructions in favor of faster register operations. Further work led to the first POWER processor and its ISA, whose lineage eventually split into two branches:

- POWER — a high-performance server platform developed by IBM.
- PowerPC — a variant created jointly by Apple, IBM, and Motorola for consumer devices.

Growing fragmentation between the branches prompted a unification effort, resulting in the Power ISA. Its first release, version 2.03, unified the two under a common specification [Pow06]. Both predecessor architectures were subsequently deprecated, though many tools still refer to the 32- and 64-bit Power ISA by the legacy PowerPC-derived names PPC and PPC64. Since 2013, the ISA has been maintained by the OpenPOWER Foundation, now part of the Linux Foundation.

2.2.2 Modern Power ISA

The Power ISA defines a 64-bit, bi-endian RISC architecture with two execution modes: 64-bit and 32-bit, switchable at runtime. Pure 32-bit implementations are derived by restricting execution to 32-bit mode while halving the width of general-purpose registers. Similarly, both 64-bit and 32-bit implementations may restrict operation to a single endian mode.

Power ISA version 3.0c introduced compliancy subsets, allowing implementations to support only specific portions of the ISA [Ope20]. The four subsets, in increasing size, are SFS, Scalar Fixed-Point + Floating-Point Compliancy Subset (SFFS), Linux Compliancy Subset (LCS), and AIX Compliancy Subset (ACS). Each defines its own list

of optional features, including Single Instruction, Multiple Data (SIMD) instructions, power management, and atomic memory operations. Instruction counts range from roughly 130 in the SFS to over 1,000 in the ACS, varying further depending on which optional features are supported. All implementations share exactly 32 General-Purpose Registers (GPRs).

2.3 Related Work

VADL and OpenVADL were formally introduced in a 2024 paper that presents the language's core features, processor specification structure, and artifact generators, alongside benchmark results for RISC-V and AArch64 ISS implementations [FHH+25]. It compares VADL's feature set to other PDLs, highlighting its clear separation of ISA and microarchitectural concerns as a distinguishing feature. The paper serves as the primary reference for Chapter 3 and covers language aspects not included in the current PPC64 specification.

QEMU is an open-source machine emulator and virtualizer that uses dynamic binary translation, converting target instructions into host native code at runtime to achieve high simulation performance [Bel05]. It supports full-system emulation of multiple architectures, including PPC64, and serves as the reference simulator for validating and benchmarking the simulator presented in this thesis. The low-level implementation of QEMU allows for high performance but demands considerable development effort, a trade-off that VADL aims to address.

LISA, short for Language for Instruction Set Architectures, is a PDL developed at RWTH Aachen University that, similarly to VADL, generates software tools and hardware implementations from a single processor description [PHZM99]. The syntax for defining instructions and declaring processor resources shares many parallels with VADL. One notable difference is that the instruction behavior in LISA is written in C, a low-level procedural language, whereas VADL takes a higher-level functional approach.

Genesys-Pro is a constraint-based test program generation tool developed by IBM for functional processor verification [AAF+04]. It generates tests by solving a constraint satisfaction problem for each instruction, making it considerably more powerful than the random input generation approach used in this thesis. Genesys-Pro has been used to verify multiple IBM POWER processors.

ISSs are typically written by referencing the architecture manual and implementing the described logic manually. A different approach, demonstrated in [BHJ+11], is to generate the simulator directly from the manual itself. The work parses the ARMv6 reference manual to recover binary encodings, pseudo-code, and assembly syntax from which a simulator is generated automatically. PowerPC is also discussed as a target, though the authors note that its use of non-ASCII characters in the pseudo-code makes extraction more difficult. Only ISS generation is covered, without addressing the other artifacts PDLs like VADL can produce.

CHAPTER 3

Processor Specification in OpenVADL

This chapter covers the type system, common definitions, and the structure of an architecture specification, and concludes with a description of VADL's macro system.

3.1 Type System

3.1.1 Types

VADL provides five built-in types. There are three vector types: `Bits`, `UInt`, and `SInt`. `Bits` represents a bit vector, while `UInt` and `SInt` represent unsigned and two's-complement signed vectors, respectively. Vector types require a constant width parameter specified in angle brackets (e.g. `Bits<32>`). Tensors are defined by including multiple size parameters.

Bits are accessed via indexing with parentheses (e.g. `a(0)`), where index zero represents the least significant bit. Note that this is the opposite of the Power ISA convention, where index zero refers to the most significant bit. Bit ranges are extracted using slice notation (e.g. `a(7..0)`). The remaining types are `Bool` and `Str`, the latter being restricted to instruction assembly definitions and the macro system.

3.1.2 Casting

Explicit type casting is done using the `as` operator. Casting preserves the bit representation when the target type has the same width as the source. Narrower casts truncate the value, while widening behavior depends on the source type: `UInt` zero-extends, `SInt` sign-extends, and `Bits` sign-extends only when cast to `SInt`; otherwise, it zero-extends.

For arithmetic and bitwise operations (excluding shifts and rotations), VADL implicitly casts `Bits<N>` operands to match typed operands: operations involving `SInt<N>` produce `SInt<N>` results, while those involving `UInt<N>` produce `UInt<N>` results. Mixing `UInt` and `SInt` operands without explicit casting raises an error, as both operands must have compatible types.

3.2 Common Definitions

VADL provides a set of common definitions that are available across all sections of an architecture specification: constants, type aliases, enumerations, and format definitions. Examples of each are shown in Listing 3.1.

3.2.1 Constants

A constant definition binds a fixed value, given as a constant expression, to an identifier. The value is typically a literal in decimal, hexadecimal (prefix `0x`), or binary (prefix `0b`) notation. Single quotes can be inserted between digits to improve readability.

3.2.2 Type Aliases

A type alias binds an existing type to an identifier, making it available under a shorter or more descriptive name. This can greatly improve readability, especially for frequently used types in format definitions. Vector types require at least one size parameter.

3.2.3 Enumerations

Enumerations assign integer values to a list of names within a shared namespace, starting at zero by default. Any entry can be explicitly assigned a value via a constant expression, with subsequent entries incrementing from there. Values are accessed using the enumeration name followed by `::` and the entry name.

3.2.4 Formats

Formats describe the structure of instructions and registers, providing easier access to fields and reducing the likelihood of mistakes. VADL supports two styles for defining formats. The first defines fields sequentially, starting from the most significant bit, with each field specifying only its type. The second uses bit slice notation to explicitly specify bit positions for each field. The latter is necessary for non-contiguous fields but can be used for any format definition. Fields must not overlap.

Formats can also contain access functions that compute derived values from the format's fields. Both notations, along with access functions, are shown in Listing 3.1.

```
constant maxU64  : Bits<64> = 0xFFFF'FFFF'FFFF'FFFF

using Instr  = Bits<32>
using OpCode = Bits<6>
using XIndex = Bits<5>
using DWord  = Bits<64>
using SDWord = SInt<64>

enumeration SPRIndex : Bits<10> =
{ XER = 0b00000'00001       // 1
, LR  = 0b00000'01000       // 8
, CTR                       // 9
, TBL = 0b01000'01100       // 268
, TBH                       // 269
, TAR = 0b11001'01111       // 815
}

format DFormA : Instr =
{ opcd    : OpCode
, rs      : XIndex
, ra      : XIndex
, imm     : Bits<16>
, immS = imm as SDWord                // imm sign extended
}

format DXForm : Instr =
{ opcd [31..26]            : OpCode
, rs   [25..21]            : XIndex
, imm  [15..6, 20..16, 0]  : Bits<16>
, rsv  [5..1]              : Bits<5>
, shImmS = imm as SDWord << 16        // imm sign extended and shifted
}

// add operation, if carry -> result zero
function addZCarry (a : DWord, b : DWord) -> DWord =
  let res, flags = VADL::adds(a, b) in
    if flags.carry
    then 0
    else res

// generates a wrapping 64-bit mask
function mask (start : Bits<6>, end : Bits<6>) -> DWord =
  let lo = VADL::lsl(maxU64, start) in
    let hi = VADL::lsr(maxU64, 63 - end) in
      if start <= end
      then lo & hi
      else lo | hi
```

Listing 3.1: Common definitions

3.2.5 Functions

Functions can read from registers and memory, but cannot write to them. Recursive calls and higher-order functions are not supported.

A set of built-in functions is provided through the `VADL` namespace, covering most

operations needed for instruction definitions. Some built-ins return both a result and a status structure containing four boolean flags: zero, carry, overflow, and negative. To unpack both values, the built-in must be called from within a `let` statement. Listing 3.1 shows two example function definitions. The `addZCarry` function uses the `adds` built-in, which returns a status structure alongside the result to check whether an addition caused a carry.

3.3 Specification Structure

An architecture specification for the generation of a functional simulator requires at least two sections: `instruction set architecture` and `processor`.

3.3.1 ISA section

The ISA section forms the core of a VADL specification. By convention, it starts with a list of common definitions, followed by declarations for registers, the Program Counter (PC), and memory. These are then followed by format and instruction definitions. Listing 3.2 provides examples of register, PC, and memory declarations. A complete ISA section defining three PPC64 instructions is shown in Listing 3.3.

3.3.1.1 Registers

VADL supports the declaration of both individual registers and dynamically indexable register files. Register files can be defined in two ways: the first uses a mapping from an index type to a register type via the -> relation symbol, which limits the total number of registers to a power of two. The second allows any number of registers by using a vector type.

Registers can be given aliases. They can have different annotations and even types from the source register, as long as the bit width stays the same. In some RISC architectures, such as RISC-V and PPC64, index zero of the GPR file has special semantics in some instructions: reads always return zero, and writes are discarded. This behavior can be expressed by combining the `zero` annotation with an alias register, allowing both the full register file and the register file with a hardwired zero to be accessed. Annotations applied to a register or alias are inherited by all of its aliases.

3.3.1.2 Program Counter

Every VADL specification requires exactly one PC, defined separately from regular registers. By default, the PC points to the current instruction being executed and is automatically incremented by the instruction size after each instruction that does not explicitly modify it. Some architectures use different conventions, where the PC points to the next instruction or even the one after that. These behaviors can be expressed using the `next` or `next next` annotations.

```
1
2  using XIndex  = Bits<5>
3  using Reg32   = Bits<32>
4  using Reg64   = Bits<64>
5  using Address = Bits<64>
6  using Byte    = Bits<8>
7
8  register X : XIndex -> Reg64          // 32x 64-bit registers (relation)
9  [zero : XZ(0)]
10 alias register XZ = X                 // XZ(0) = 0
11 alias register XW = X(*)(31..0)       // lower 32 bits of X registers
12 register XV : Reg64<32>               // 32x 64-bit registers (vector)
13 register CR : Reg32                   // single 32-bit register
14 alias register CRF : Bits<8><4>  = CR // 8x4-bit fields in CR
15
16 [next]                                // points to next instruction address
17 program counter PC : Reg64
18
19 [bigEndian]
20 memory MEM : Address -> Byte
21
22 instruction EXAMPLE : InstrForm = {
23     X(0) := 1 as Bits<64>              // write 64-bit value to X(0)
24     MEM<4>(0x10) := 1 as Bits<32>      // write 32-bit value to MEM at 0x10
25 }
26 encoding EXAMPLE = ...
27 assembly EXAMPLE = ...
```

Listing 3.2: Register, PC and Memory Declarations

Within instruction definitions, the addresses of the current, next, and two ahead instructions can be accessed through the PC's built-in methods: `.current`, `.next` , and `.nextnext`.

3.3.1.3 Memory

A memory declaration follows the same structure as a relational register file declaration but uses the `memory` keyword instead. The relation maps from address type to memory cell type. Every VADL specification must include at least one memory declaration. Various annotations are available to customize memory characteristics. By default, memory stores both data and instructions, uses a little-endian byte order, and performs no address translation.

Memory cells are accessed by placing the address in parentheses (e.g. `mem(0x100)`). A size parameter in angle brackets can be added to access multiple cells at once (e.g. `mem<4>(0x100)` accesses four cells).

3.3.1.4 Instructions

Each instruction definition consists of three components linked by a common identifier: the `instruction` definition defines the behavior, the `encoding` definition assigns values for fixed fields, and the `assembly` definition determines how the instruction appears in disassembler or compiler output.

instruction Instruction behavior is written in a functional style and expects a single statement. If multiple statements are needed, they must be grouped into a block statement using curly brackets. Internally, all reads occur at the start of instruction execution, and all writes occur at the end. Consequently, read-after-write operations within a single instruction are not possible for registers and memory.

encoding Encoding definitions assign values to fixed fields, such as opcodes. Every field in the instruction format should appear either in the instruction behavior or in the encoding definition.

assembly The assembly definition consists of a single string expression. Parentheses combine multiple comma-separated strings into one. The built-in functions `sdec`, `udec`, and `hex` convert a value into its signed decimal, unsigned decimal, or hexadecimal string representation, respectively.

3.3.1.5 Exceptions

Exceptions represent exceptional behavior in instructions. They resemble functions but can modify registers and memory and do not return a value. An exception consists of a single statement; multiple statements must be grouped using curly brackets. Exceptions are defined using the `exception` keyword and invoked with the `raise` statement. The exceptional behavior can also be written directly after the raise statement, though this prevents the reuse of the exception. Listing 3.4 shows an exception definition.

3.3.2 Processor section

The processor section contains initialization code. Starting values for registers and the PC can be defined in a block following the `reset` keyword. Listing 3.5 shows the processor section from the PPC64 specification.

3.4 Macro System

Defining each instruction variant manually quickly becomes impractical. A single arithmetic instruction in PPC64 can have up to four variants through combinations of the overflow (`O`) and record (`.`) suffixes, each requiring a separate definition with a large amount of shared code. While functions can be used to reuse some behaviors, they cannot abstract over syntax elements like encodings or assembly definitions. VADL's macro

```
instruction set architecture PPC64SFS = {
  using Bit    = Bits<1>
  using OpCode = Bits<6>
  using XIndex = Bits<5>
  using Ext10  = Bits<10>
  using Instr  = Bits<32>

  program counter PC : Bits<64>
  memory MEM : Bits<64> -> Bits<8>
  register X : XIndex -> Bits<64>

  format XFormA : Instr =
  { opcd    : OpCode
  , rs      : XIndex
  , ra      : XIndex
  , rb      : XIndex
  , extopcd : Ext10
  , rc      : Bit
  }

  instruction AND : XFormA =
    X(ra) := VADL::and(X(rs), X(rb))
  encoding AND = { opcd = 0b011111, extopcd = 0b00000'11100, rc = false }
  assembly AND = ("AND ", udec(ra), ",", udec(rs), ",", udec(rb))

  instruction OR : XFormA =
    X(ra) := VADL::or(X(rs), X(rb))
  encoding OR = { opcd = 0b011111, extopcd = 0b01101'11100, rc = false }
  assembly OR = ("OR ", udec(ra), ",", udec(rs), ",", udec(rb))

  instruction XOR : XFormA =
    X(ra) := VADL::xor(X(rs), X(rb))
  encoding XOR = { opcd = 0b011111, extopcd = 0b01001'11100, rc = false }
  assembly XOR = ("XOR ", udec(ra), ",", udec(rs), ",", udec(rb))
}
```

Listing 3.3: Complete ISA Section

system addresses this by operating directly on syntax elements, allowing shared patterns to be defined once and significantly reducing the overall specification size.

3.4.1 Macro Definition

Macros are declared using the `model` keyword. Each macro accepts zero or more syntax elements as arguments and produces exactly one syntax element. Both arguments and output require explicit syntax types. Common syntax element types include `Bool` (a boolean value), `Bin` (a binary value), `Str` (a string value), `Ex` (an expression), `Stat` or `Stats` (one or multiple statements), `Encs` (a list of encoding assignments, or `none`), and `IsaDefs` (`instruction`, `encoding`, and `assembly` definitions). Macro arguments and calls are accessed using the `$` prefix. Semicolons are used to separate macro arguments. Listing 3.6 demonstrates how a single macro definition can generate the three instructions that were defined separately in Listing 3.3.

```
exception Privilege () = {
  SRR0 := PC
  SRR1 := (MSR(63..31), 0 as Nibble, MSR(26..22), 0 as Bits<3>, 1 as Bit, 0
      as Bits<2>, MSR(15..0))
  PC   := 0x0000'0000'0000'0700
}
raise Privilege // example call
```

Listing 3.4: Exception Definition

```
processor Power implements PPC64SFS = {
  reset = {
    PC    := 0x0000'0000'0000'0100
    X(3)  := 0x0000'0000'1FE0'0000
    MSR   := 0x0000'0000'0000'3000
  }

  [ firmware ]
  [ base: 0x00000000 ]
  memory region [RAM] DRAM in MEM
}
```

Listing 3.5: Processor Definition

3.4.2 Records

To minimize the number of arguments, VADL supports syntax type composition via record definitions. Elements within a record are accessed using the argument name, followed by a period and the field name. Records can also contain other records. Listing 3.6 shows how three different syntax types are combined into one using the record `AsmOpExt`. Record initialization is performed by enclosing all arguments in parentheses and separating them with semicolons, as in macro calls.

3.4.3 Lexical macros

`AsId` and `AsStr` are lexical macros used for string and identifier manipulation. `AsStr` and `AsId` both take multiple identifiers and strings as input, producing a concatenated string or identifier as output. Lexical macros are particularly useful when a string, such as an instruction mnemonic, is needed in different forms; for example, as both an instruction identifier and as a string in the `assembly` definition. In Listing 3.6, `AsId` converts the mnemonic strings into unique identifiers for each instruction.

```
instruction set architecture PPC64SFS = {
  ...
  record AsmOpExt (asm : Str, op : Id, ext : Bin)

  model XFormLogicalInstr (x : AsmOpExt) : IsaDefs = {
    instruction AsId($x.asm) : XFormA =
      X(ra) := VADL::$x.op(X(rs), X(rb))
    encoding AsId($x.asm) =
      {opcd = 0b011111, extopcd = $x.ext, rc = false}
    assembly AsId($x.asm) =
      ($x.asm, " ", udec(ra), ",", udec(rs), ",", udec(rb))
  }
  $XFormLogicalInstr(("AND"; and; 0b00000'11100))
  $XFormLogicalInstr(("OR" ; or ; 0b01101'11100))
  $XFormLogicalInstr(("XOR"; xor; 0b01001'11100))
}
```

Listing 3.6: ISA Section with Macros

```
model CS (c : Bool, stat : Stat) : Stat = {
  match : Stat ( $c = true => $stat; _ => {} )
}
$CS($lk; LR := PC.next) // example call: only update LR if $lk is set
```

Listing 3.7: Macro Example

3.4.4 Match macro

Match macros enable conditional logic within the macro system by selecting different syntax elements based on comparisons. Each case tests for equality or inequality between two syntax elements and returns the corresponding result upon a successful match. A default case must be provided at the end using an underscore as a placeholder. Listing 3.7 shows one of the most frequently used macros in the PPC64 specification: `CS`, which conditionally inserts a statement based on a boolean value. The advantage over an if statement is that the specification with expanded macros is significantly easier to read and debug.

CHAPTER 4

PPC64 Implementation

This chapter presents VADL's PPC64 implementation, covering the implementation scope, flag and instruction macros, and deviations from the QEMU reference simulator.

4.1 Scope

The implementation targets the big endian 64-bit SFS from Power ISA 3.1b (Power10), covering all 132 base instructions and their variants, as well as some additional overflow variants from larger subsets [Ope21]. It supports both 64-bit and 32-bit software, starting in 32-bit compatibility mode by default. Setting the SF bit in the Machine State Register (MSR) switches the processor to 64-bit mode.

The specification covers a total of 41 registers:

32 × 64-bit General-Purpose Registers (GPRs): Used for general computation.

1 × 32-bit Condition Register (CR): Contains eight fields for storing comparison results.

3 × Machine Registers:

- **64-bit Machine State Register (MSR):** Controls processor behavior and indicates processor capabilities. The VADL simulator implements two bits: PR (privilege state) and SF (32-bit/64-bit execution state).
- **64-bit Save/Restore Register (SRR) 0:** Saves the PC during interrupts.
- **64-bit Save/Restore Register (SRR) 1:** Saves part of the MSR during interrupts.

5 × Special-Purpose Registers (SPRs):

32-bit Fixed-Point Exception Register (XER): Stores arithmetic flags.

64-bit Link Register (LR): Holds the return address for function calls and acts as a branch target.

64-bit Count Register (CTR): Holds loop counters and can be used as an indirect branch target.

64-bit Time Base (TB): Provides a continuously incrementing timestamp.

64-bit Target Address Register (TAR): Used exclusively as a branch target.

4.2 Flag Handling

SFS instructions may access two flag registers, CR and XER, with up to nine flag bits potentially set per instruction. Several of these bits behave differently depending on the active execution mode. All CR and XER updates are handled through a small set of five macros that centralize this logic and account for the architecture's special cases.

4.2.1 Condition Register

The CR consists of eight 4-bit fields for storing comparison results. Comparison instructions can target any of these fields, while fixed-point arithmetic instructions always write to field zero. Each field follows the same structure: the upper three bits encode a comparison result (less than, greater than, or equal to), and the least significant bit stores the value of `XER.so` after instruction completion. Non-move instructions modify at most one field at a time.

Since CR accesses mostly target a single field, it is implemented as a register file of eight 4-bit registers (`CRF`) rather than a single 32-bit register, making individual field accesses more efficient.

Three macros handle `CRF` updates: `SetCRBase`, `SetCR0`, and `SetCRCmp`. Only `SetCR0` and `SetCRCmp` appear in instruction definitions. Listing 4.1 shows all macros alongside the register and alias definitions.

SetCRBase This macro contains the core logic for updating CR fields. It accepts a destination field expression `dst`, two comparison operands `a` as a `TwoArgs` record, and an expression to set the summary overflow bit `so`.

SetCR0 Performs a signed comparison of the instruction's result against zero and always updates field zero. The `MSR.sf` bit controls the comparison width. When set to zero, `result` is truncated to 32 bits and then sign extended back to 64 bits, ensuring the same result as a 32-bit comparison. `SetCR0` takes two arguments: `rc`, a boolean that indicates whether the instruction variant updates the CR, and `so`, an expression for the summary overflow bit. If `rc` is false, the macro returns an empty statement () instead of a `CRF` assignment (see Listing 3.7 for the `CS` macro definition).

```
using Nibble  = Bits<4>
using Word    = Bits<32>
using SWord   = SInt<32>
using SDWord  = SInt<64>
using UWord   = UInt<32>
using UDWord  = UInt<64>

register CRF  : Nibble<8>

record TwoArgs (arg1 : Ex, arg2 : Ex)

model SetCRBase (dst : Ex, a : TwoArgs, so : Ex) : Stat = {
  let a1 = $a.arg1 in
    let a2 = $a.arg2 in
      CRF($dst) := (a1 < a2, a1 > a2, a1 = a2, $so)
}

model SetCR0 (rc : Bool, so : Ex) : Stat = {
  $CS($rc;
    $SetCRBase(0;
    ( let sfmask = MSR.sf as SDWord in
        ((result          as SDWord) &  sfmask) |
        ((result as SWord as SDWord) & ~sfmask)
    ; 0
    ); $so)
  )
}

model SetCRCmp (a : TwoArgs, ctype : Id) : Stat = {
  $SetCRBase(crft;
  ( $a.arg1 as $ctype
  ; $a.arg2 as $ctype
  ); XER_SO)
}
```

Listing 4.1: CR Declaration and Update Macros

SetCRCmp This macro sets the CR for comparison instructions. The `crft` identifier selects which field to update. The macro takes two arguments: `a`, the two values being compared, and `ctype`, the type to use for the comparison. In the specification, the comparison type is always one of `SWord`, `SDWord`, `UWord`, or `UDWord`, covering signed and unsigned 32- and 64-bit comparisons. Since comparison instructions do not update the XER, the `so` argument passed to `SetCRBase` is set directly to `XER_SO`.

4.2.2 Fixed-Point Exception Register

The XER contains five flag bits that track arithmetic exceptions: CA, OV, CA32, OV32, and SO. The CA and OV flags track carry and overflow for operations in the current processor mode (32-bit carry and overflow in 32-bit mode, 64-bit carry and overflow in 64-bit mode). The CA32 and OV32 bits always track carry and overflow for the lower 32 bits. The SO bit is the summary overflow flag, which is set whenever the OV flag is

set and can only be cleared by move instructions. The remaining bits form two reserved fields and a field used for string operations, which are not part of the SFS. To improve performance, the specification implements XER as six separate registers: one for each of the five flags and one holding the remaining fields.

XER updates are handled by two macros: `SetXERBase` and `SetFlags`. `SetXERBase` is exclusively called by `SetFlags`.

SetXERBase This macro updates the required XER fields based on the carry enable (`ce`) and overflow enable (`oe`) boolean flags. The OV32 bit is set by the `ov32` argument, while the OV and SO bits are provided by the `ov` and `so` identifiers, respectively. The CA and CA32 bits are set directly from two flag status structures passed by identifier. This structure is necessary because the overflow bits are commonly set through instruction-specific conditions, while the carry bits are always set through carry flags (with the sole exception of `ADDEX`).

SetFlags `SetFlags` is used in instructions where both the CR and XER need to be set, combining the `SetXERBase` and `SetCR0` macros. The boolean arguments `oe`, `ce`, and `rc` have the same meaning as in those macros. Custom overflow expressions can be specified using `ov` and `ov32`. When overflow is enabled, `ov` and `so` are precomputed in let statements. The computed value for SO can then be passed to the `SetCR0` macro to avoid a read after write conflict.

4.3 Instructions

Some Power ISA instructions have variants that modify the behavior of the base instruction. A variant appends a suffix to the base mnemonic:

- `C` — updates the carry flags in the XER
- `O` — updates the overflow flags in the XER
- `.` — updates a field in the CR
- `L` — saves the next instruction address to the LR before execution
- `A` — computes the branch target using absolute rather than relative addressing

For SFS instructions, `C`, `O`, and `.` can appear together, as can `L` and `A`.

4.3.1 Instruction Macro Structure

Instruction macros are used to define instructions and their variants. Macros that directly contain `instruction`, `assembly`, and `encoding` definitions appear in two types: base macros, called directly to generate single instructions, and extended macros (ending

```
using Reg32 = Bits<32>
using Bit   = Bits<1>

register XER_BASE    : Reg32 // non-flag fields

register XER_SO      : Bit
register XER_OV      : Bit
register XER_CA      : Bit
register XER_OV32    : Bit
register XER_CA32    : Bit

model SetXERBase (oe : Bool, ce : Bool, ov32 : Ex) : Stat = {
  {
    $CS($oe; {
      XER_SO   := so
      XER_OV   := ov
      XER_OV32 := $ov32
    })
    $CS($ce; {
      XER_CA   := (flags64.carry && MSR.sf) || (flags32.carry && ~MSR.sf)
      XER_CA32 := flags32.carry
    })
  }
}

model SetFlags (oe : Bool, ce : Bool, rc : Bool, ov32 : Ex, ov : Ex) : Stat
    = {
  match : Stat
  ( $oe = true => let ov = $ov in
                    let so = XER_SO | ov in {
                      $SetXERBase(true; $ce; $ov32)
                      $SetCR0($rc; so)
                    }
  ; _          => {
                    $SetXERBase(false; $ce; $ov32)
                    $SetCR0($rc; XER_SO)
                  }
  )
}
```

Listing 4.2: XER Declaration and Update Macros

in `Ext`), which take extra boolean arguments to control which variant of an instruction to generate.

Each `Ext` macro has a corresponding variant macro that calls it once for every combination of variant arguments, producing all variants for a given base instruction, including the base instruction itself.

Sections 4.3.2 and 4.3.3 present examples of base and extended instruction macros.

```
record AsmOpOpcd     (asm : Str, op : SymEx, opcd : Bin)

// instructions: andi., andis., ori, oris, xori, xoris
model DFormLogicalInstr (x : AsmOpOpcd, arg2 : Id, rc : Bool) : IsaDefs = {
  instruction AsId($x.asm, $rId($rc)) : DFormA =
    let result = VADL::$x.op(X(rs), $arg2) in {
      X(ra) := result
      $SetCR0($rc; XER_SO)
    }
  encoding AsId($x.asm, $rId($rc)) =
    {opcd = $x.opcd}
  assembly AsId($x.asm, $rId($rc)) =
    ($x.asm, $rStr($rc), " ", udec(ra), ",", udec(rs), ",", udec(imm))
}

$DFormLogicalInstr(("ANDI" ; and; 0b011100); immU  ; true )
$DFormLogicalInstr(("ANDIS"; and; 0b011101); shImmU; true )
$DFormLogicalInstr(("ORI"  ; or ; 0b011000); immU  ; false)
...
```

Listing 4.3: DForm Logical Instructions Macro Example

4.3.2 DForm Logical Instruction Macro Example

DForm logical instructions are special in PPC64: typically, if a CR-updating record variant exists for one instruction in a family, it exists for all, and a trailing period usually appears only in such variants. Both rules are broken for DForm logical instructions, as every instruction is a base instruction with no variants; yet some, but not all, carry a trailing period. For example, `ANDI.` exists without an `ANDI` counterpart, while `ORI` has no `ORI.` variant. Since no variants exist, one macro layer is enough to define all instructions concisely.

`DFormLogicalInstr`, shown in Listing 4.3, is called directly to produce each instruction definition. Because the mnemonic (`asm`) is used both in the `assembly` definition and as part of a unique identifier, periods cannot appear in the string (as in `andi.` and `andis.`). In the `assembly` definition, the period is generated conditionally by the `rStr` macro based on `rc`. In the identifier, `rId` works the same way but generates an underscore instead, keeping the identifier valid (see Listing 4.5 for all string macro definitions). `AsId` then converts the resulting string into an identifier.

In the `instruction` definition, a `let` statement is required to pass the operation result to `SetCR0` as `result`.

4.3.3 XOForm Add/Sub Instruction Macro Example

Listing 4.4 shows two macros, `XOFormAddSubInstrExt` and `XOFormAddSubInstr`, which together generate a total of 40 instruction variants (10 base instructions) in the specification.

`XOFormAddSubInstrExt` receives three arguments: a record with the base mnemonic and instruction parameters (`r`); a boolean for an overflow variant (`oe`); and a boolean for a variant that sets the CR (`rc`). `XOFormAddSubInstr` generates all variants by calling `XOFormAddSubInstrExt` four times, once for each combination of `oe` and `rc`.

The final mnemonic is assembled by combining the base mnemonic with a suffix generated by the `orStr` macro from the boolean arguments. The unique identifier is built the same way via `AsId`, using `orId` in place of `orStr`, which uses an underscore instead of a period to represent `rc` (see Listing 4.5).

In the instruction behavior, two results are computed: one with 32-bit operands and one with 64-bit operands. While it is always the 64-bit result that gets written to the result register, the 32-bit status structure is needed to correctly set arithmetic flags. Flag updates are handled by the `SetFlags` macro, which uses the `oe` and `rc` boolean parameters to control whether overflow flags and the condition register are updated (see Listing 4.2).

This example also illustrates how the carry suffix is implemented. Because it is not available for all base instructions in the class, carry is implemented by adjusting the parameters in the `XOFormAddSubInstrRec` record rather than being implemented through the variant macro layer.

4.4 Limitations

The VADL PPC64 simulator differs from the QEMU reference in several ways.

The TB register can be read using the `MFSPR` and `MFTB` instructions, but it always returns zero. VADL lacks an annotation for auto-incrementing registers, and adding an explicit increment to every instruction definition is impractical. Programs that use the TB register as a timing source will, therefore, produce incorrect results.

Only a subset of SPRs is implemented. Using `MTSPR` or `MFSPR` with an unimplemented SPR index will produce unexpected behavior.

The `MTMSR` instruction may trigger an interrupt or change processor behavior when certain MSR bits are written. The VADL simulator only implements the PR and SF bits, so writing to any other bit may produce different results between the simulators. For example, the LE bit, which controls the endian mode, has no effect on the generated simulator.

```
record TwoArgs       (arg1 : Ex, arg2 : Ex)
record AsmOpExt      (asm : Str, op : SymEx, ext : Bin)
record XOFormAddSubInstrRec (x : AsmOpExt, a : TwoArgs, enc : Encs, ca : Ex,
     ce : Bool, rbasm : Ex)

model XOFormAddSubInstrExt (r : XOFormAddSubInstrRec, oe : Bool, rc : Bool)
    : IsaDefs = {
  instruction AsId($r.x.asm, $orId($oe; $rc)) : XOForm =
    let result32, flags32 = VADL::$r.x.op($r.a.arg1 as Word,
                                          $r.a.arg2 as Word, $r.ca) in
      let result, flags64 = VADL::$r.x.op($r.a.arg1         ,
                                          $r.a.arg2         , $r.ca) in {
        X(rs) := result
        $SetFlags($oe; $r.ce; $rc; flags32.overflow;
            (flags64.overflow && MSR.sf) || (flags32.overflow && ~MSR.sf))
      }
  encoding AsId($r.x.asm, $orId($oe; $rc)) =
    {opcd = 0b011111, oe = $oe, extopcd = $r.x.ext, rc = $rc, $r.enc}
  assembly AsId($r.x.asm, $orId($oe; $rc)) =
    ($r.x.asm, $orStr($oe; $rc), " ", udec(rs), ",", udec(ra), $r.rbasm)
}

// instructions: add[o][.], addc[o][.], adde[o][.], subf[o][.], subfc[o][.],
     subfe[o][.], addme[o][.], addze[o][.], subfme[o][.], subfze[o][.]
model XOFormAddSubInstr (r : XOFormAddSubInstrRec) : IsaDefs = {
  $XOFormAddSubInstrExt($r; false; false)
  $XOFormAddSubInstrExt($r; false; true )
  $XOFormAddSubInstrExt($r; true ; false)
  $XOFormAddSubInstrExt($r; true ; true )
}

$XOFormAddSubInstr(("ADD" ; addc; 0b1000'01010); (X(ra); X(rb)); none ;
    false ; false; (",", udec(rb))))
$XOFormAddSubInstr(("ADDC"; addc; 0b0000'01010); (X(ra); X(rb)); none ;
    false ; true ; (",", udec(rb))))
...
```

Listing 4.4: XOForm Add/Sub Instructions Macro Example

```
model rStr (r : Bool) : Str = { match : Str ( $r = true => "."; _ => "" ) }
model rId  (r : Bool) : Str = { match : Str ( $r = true => "_"; _ => "" ) }
model lStr (l : Bool) : Str = { match : Str ( $l = true => "L"; _ => "" ) }
model orStr (o : Bool, r : Bool) : Str = {
  match : Str
  ( $o = true => match : Str ( $r = true => "O."; _ => "O" )
  ; _         => match : Str ( $r = true => "." ; _ => ""  )
  )
}
model orId  (o : Bool, r : Bool) : Str = {
  match : Str
  ( $o = true => match : Str ( $r = true => "O_"; _ => "O" )
  ; _         => match : Str ( $r = true => "_" ; _ => ""  )
  )
}
model laStr (l : Bool, a : Bool) : Str = {
  match : Str
  ( $l = true => match : Str ( $a = true => "LA"; _ => "L" )
  ; _         => match : Str ( $a = true => "A" ; _ => ""  )
  )
}
```

Listing 4.5: String Macros

CHAPTER 5

Validation

Validation is performed using VADL's cosimulator, which runs test programs concurrently in both the generated simulator and QEMU, reporting any discrepancies. Small test programs are automatically generated for each instruction variant and fed into the cosimulator. Section 5.1 describes the test setup, and Section 5.2 explains why some instructions are not tested exhaustively.

5.1 Setup

5.1.1 Configuration

Cosimulator configuration is handled through architecture-specific TOML files, each specifying the run commands for both simulators and defining the register mappings used during comparison. For two registers, the CR and the XER, a custom slice mapping is used to accommodate their differing implementations. The PPC64 configuration file is named `ppc64_config.toml`.

5.1.2 Test Case Generation

Test cases are generated using Java. For each instruction variant, a fixed number of tests is generated with different instruction parameters. If an instruction reads from a register or memory, that location is first filled with random values. Half of all tests run in 32-bit compatibility mode; the other half run in 64-bit mode, activated by inserting a `trap` instruction at the start of the test code to set the SF bit. Tests for each instruction are written to a YAML file and copied into a Docker container that already contains the built cosimulator.

5.1.3 Running the Cosimulator

A script inside the container parses the YAML file, generates temporary assembly files, compiles them, and runs the cosimulator for each test. Since the VADL simulator and QEMU do not share a common way of terminating program execution, QEMU's Stop on Trigger plugin is used for both simulators, setting a termination address to which all tests branch at the end. The cosimulator generates one output file per test, recording any differences found between the two simulators. These are then parsed and displayed by the Java code.

5.2 Limitations

Most instructions are tested for all possible input values, with a few exceptions where only a limited set of parameters is used. These are defined separately in their own Java class to distinguish them from the other tests.

The branch instructions `B[L][A]`, `BC[L][A]`, `BCCTR[L]`, `BCLR[L]`, and `BCTAR[L]` always use the exit address as their branch target since all tests must end there, and branching to uninitialized memory could cause unexpected behavior. For `B[L][A]` and `BC[L][A]`, the target is tested with both absolute and relative addressing.

Since only a subset of SPRs is implemented, `MTSPR` and `MFSPR` are tested exclusively with the indices of those registers. The TB register is excluded, as it is read-only, and a read would return different values in each simulator.

While `MFMSR` is fully implemented, `MTMSR` requires special handling. Certain MSR fields, when modified by `MTMSR`, trigger behavior in QEMU that the VADL simulator does not implement. For this reason, these fields are always set to zero, which is their initial value. The affected fields are IR (bit 5), TE (bits 9 and 10), PR (bit 14), and EE (bit 15).

The `MFTB` instruction behaves identically to `MFSPR`, meaning it can read any SPR and store the value in any GPR. However, the GNU Compiler Collection (GCC) PPC64 compiler only allows `MFTB` in its single-argument form, where it always reads the TB register. Because of this limitation, `MFTB` is excluded from the instruction tests.

CHAPTER 6

Evaluation

Two evaluations were conducted: one assessing how effectively VADL's macro system reduces code duplication, and another comparing the performance of the generated PPC64 simulator against its QEMU reference implementation.

6.1 Macro System Evaluation

When the `expand-macros` option is passed to VADL's `iss` command, an additional specification file is generated alongside the simulator, in which all macros have been fully expanded. Comparing the specification against its expanded counterpart, after removing comments and empty lines from both, shows a more than threefold difference in size: 1,238 lines versus 3,900. This is also reflected at the instruction level: while the original specification contains 48 instruction definitions, the expanded version contains 215, with each instruction macro or macro group expanding into more than four definitions on average.

The macro system proves effective at reducing the code needed to specify flag register updates. Looking at reads and writes to the CR and XER registers, the original specification contains 86 accesses in total, while the expanded version contains 671, nearly an eightfold increase.

6.2 Performance Evaluation

Performance was evaluated by comparing the execution time of the generated simulator against QEMU across a set of benchmarks. Each benchmark measures a single instruction by executing it many times in a loop. The loop count was chosen so that each benchmark runs for approximately 10 seconds in the QEMU simulator. An example benchmark file is shown in Listing 6.1.

Figure 6.1 shows the relative execution time of each tested instruction, where lower values indicate faster execution in the VADL simulator. Most instructions perform very similarly across both simulators; however, some groups show significant deviations.

Instructions with the `.` suffix update the CR and are referred to as record-form instructions. These perform particularly well in the VADL simulator. As seen with `ADD.` and `MULHW.`, their relative execution time is noticeably faster than that of their base counterparts. This comes down to a difference in how the two simulators handle instruction variants.

The VADL specification includes unique instruction definitions for every instruction variant, producing one translation function each. QEMU, on the other hand, merges most variants into a single shared translation function and uses conditional branches on instruction encoding fields to select the appropriate behavior. For record-form variants, the relevant branch is annotated with an `unlikely` compiler hint, which introduces considerable overhead.

The other two variant suffixes present in the tested instructions, `O` and `E`, do not see the same benefit. Their performance relative to the base instruction is significantly worse. This is mostly due to an inefficient calculation of 32-bit overflow and carry flags in the VADL simulator.

Listing 4.4 shows the macro responsible for generating definitions for the `ADD[E][O][.]` and `SUBF[E][O][.]` instructions. Some variants require the `flags32` and `flags64` status structures to set overflow and carry flags in the CR and XER. Because VADL has no built-in way to obtain both 32-bit and 64-bit flags from a single operation, two separate arithmetic operations are needed to produce both status structures. This is wasteful: `result32` goes unused, and part of the flag calculation is duplicated as a result. Both VADL and QEMU compute overflow flags by producing a 64-bit intermediate value where each bit indicates overflow at that position. The difference is that QEMU does this once and extracts both OV and OV32 from it, while VADL performs the same computation twice, once for each flag.

Implementing QEMU's approach manually in VADL would undermine the purpose of using a high-level PDL. A better path forward would be to improve code generation in OpenVADL to eliminate this kind of duplicated logic. Alternatively, the existing flag operations could be extended to support generating multiple flag structures for different bit widths from a single built-in operation.

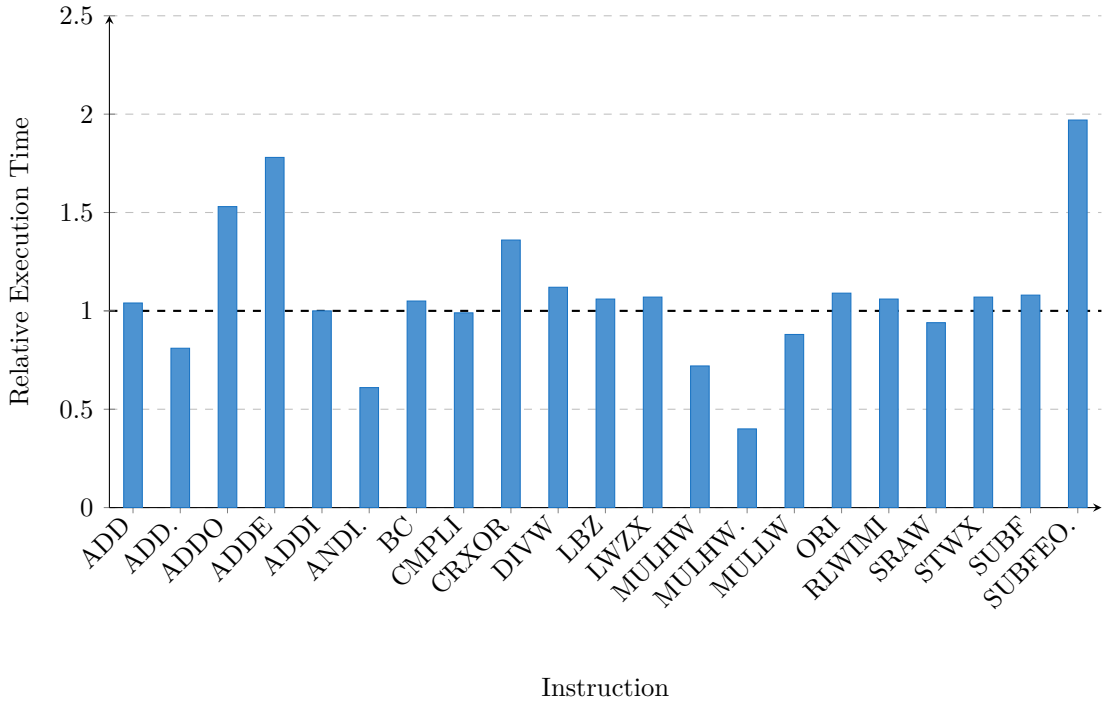

Figure 6.1: Execution Time of the VADL Simulator Relative to QEMU (lower is better)

```
1  .machine power10
2  .globl _start
3  .section .text
4  _start:
5      nop
6      lis       3, 0x0048     # loop counter
7
8  loop:
9      cmpli     0, 0, 3, 0
10     beqa      0xfc          # branch to exit if loop counter equals zero
11
12     .rept 10000             # reduce influence of loop instructions
13         addi 1, 1, 1
14     .endr
15
16     addi      3, 3, -1      # loop counter--
17     b         loop
```

Listing 6.1: Performance Test Example

CHAPTER 7

Conclusion

VADL is expressive enough to handle the complexity of the Power ISA SFS, and its macro system proves to be well suited for architectures of this scale, significantly reducing the amount of code required in a processor specification. Correctness was validated, and performance was evaluated against the QEMU PPC64 simulator. Execution speed was similar across most tested instructions, though further optimization or additional features would be needed to match QEMU in all cases.

Overall, the results indicate that OpenVADL can produce correct and efficient simulators with less effort than a hand-written implementation.

Overview of Generative AI Tools Used

Overleaf AI Assist was used throughout this thesis to rephrase sentences and detect grammar issues. No custom prompts were used.

Acronyms

ACS AIX Compliancy Subset. 4, 5

CR Condition Register. 17–20, 22, 23, 27, 29, 30

CTR Count Register. 18

GCC GNU Compiler Collection. 28

GPR General-Purpose Register. 5, 10, 17, 28

ISA Instruction Set Architecture. 1, 3–5, 7, 10, 13, 15, 17, 20, 33

ISS Instruction Set Simulator. 1, 3, 5

LCS Linux Compliancy Subset. 4

LR Link Register. 18, 20

MSR Machine State Register. 17, 23, 28

PC Program Counter. 10–12, 17

PDL Processor Description Language. 1, 3, 5, 30

RISC Reduced Instruction Set Computer. 4, 10

SFFS Scalar Fixed-Point + Floating-Point Compliancy Subset. 4

SFS Scalar Fixed-Point Compliancy Subset. 1, 4, 5, 17, 18, 20, 33

SIMD Single Instruction, Multiple Data. 5

SPR Special-Purpose Register. 17, 23, 28

SRR Save/Restore Register. 17

TAR Target Address Register. 18

TB Time Base. 18, 23, 28

VADL Vienna Architecture Description Language. 1, 3–5, 7, 8, 10–12, 14, 17, 23, 27–31, 33

XER Fixed-Point Exception Register. 18–21, 27, 29, 30

Bibliography

[AAF+04] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv. Genesys-pro: innovations in test program generation for functional processor verification. *IEEE Design & Test of Computers*, 21(2):84–93, 2004.

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 41, USA, 2005. USENIX Association.

[BHJ+11] Frédéric Blanqui, Claude Helmstetter, Vania Joloboff, Jean-François Monin, and Xiaomu Shi. Designing a cpu model: from a pseudo-formal document to fast code, 2011.

[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.

[Ope20] OpenPOWER Foundation. *Power ISA Version 3.0C*. OpenPOWER Foundation, May 2020.

[Ope21] OpenPOWER Foundation. *Power ISA Version 3.1B*. OpenPOWER Foundation, September 2021.

[PHZM99] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. Lisa—machine description language for cycle-accurate models of programmable dsp architectures. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC '99, page 933–938, New York, NY, USA, 1999. Association for Computing Machinery.

[Pow06] Power.org. *Power ISA Version 2.03*. Power.org, September 2006. Archived version; original published by Power.org.