Feasibility of Vector Instruction-Set Semantics Using Abstract Monads

by

Arthur Reiner De Belen

S.B. Computer Science and Engineering, MIT, 2023

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2024

Authored by:	Arthur Reiner De Belen Department of Electrical Engineering and Computer Science August 9, 2024
Certified by:	Adam Chlipala Arthur J. Conner (1888) Professor of Computer Science, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair, Master of Engineering Thesis Committee

Feasibility of Vector Instruction-Set Semantics Using Abstract Monads

by

Arthur Reiner De Belen

Submitted to the Department of Electrical Engineering and Computer Science on August 9, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Formalizations of instruction-set semantics help establish formal proofs of correctness of both hardware designed to implement these instruction sets and the software implemented against this specification. One such prior work[1] formalizes a specification of a subset of the RISC-V instruction-set architecture using a general-purpose language, Haskell, using its monad and typeclass support to abstract over effects. Another member of the same family is the RISC-V V extension, which specifies instructions for operating on multiple data elements in a single instruction, which is useful for domains with high levels of data parallelism, such as graphics rendering and machine learning. In this work I examine the question of whether the same prior work can be extended to formalize the semantics of the vector extension. I answer this question with a tentative "yes", backed by a partial specification in Haskell of a small but nontrivial subset of this vector extension, a translation of the same specification into Coq using hs-to-coq[2], and work towards demonstrating the utility of this specification.

Thesis supervisor: Adam Chlipala
Title: Arthur J. Conner (1888) Professor of Computer Science

[1]Bourgeat et al., 2023, doi:10.1145/3607833.
[2]https://github.com/plclub/hs-to-coq

Acknowledgments

This thesis would not have been possible without my thesis advisor, Prof. Adam Chlipala, as well as his PhD student Samuel Gruetter, for their guidance in both the technical and nontechnical aspects of this project and of my journey through research over the past couple of years.

I also extend my gratitude to my academic advisor, Prof. Mengjia Yan, for supervising my academic journey throughout my Bachelor's and Master's.

Finally, my gratitude to the friends that have supported me from before MIT and at MIT, and, last but certainly not least, to my mother, whose sacrifices I owe my college education to, and so much more.

Contents

List of Tables

Chapter 1

Background

Much work in recent years has gone into formally specifying correctness of systems down to machine code, such as (a) CakeML, a verified compiler of a subset of Standard ML to x86 [1]; (b) CompCert, a commercially available C compiler [2]; and (c) the Verified Software Toolchain, a software stack building on CompCert's sequential logic to provide formalized semantics for a dialect of C supporting shared-memory concurrency [3]. The correctness proofs of the software in these systems are expressed against an operational semantics for the instruction-set architecture they target, as the instruction-set architecture (ISA) defines how a processor supporting this instruction set is expected to behave.

Two prior works on formalizing instruction-set semantics, both for subsets of the RISC-V ISA family, take two different approaches on how to express them. One such work by Armstrong et al. [4] formalized sequential semantics for a broad subset of several different instruction sets (RISC-V, CHERI-MIPS, and Armv8-A) in a custom domain-specific language (DSL), Sail, which is itself compilable to both emulators able to boot operating systems and to definitions compatible with proof assistants Isabelle/HOL, HOL4, and Coq.

Although said formalization for RISC-V has been adopted by the RISC-V International standards organization [5], Bourgeat et al. in their work formalizing a subset of just the RISC-V ISA family [6] bring up a potential concern about writing custom DSLs for each abstraction in the hardware-software stack, which is the need to write translators between specification languages (i.e., the DSLs for each abstraction) and verification languages, whose number can grow quadratically with respect to the number of languages.

Bourgeat et al. in the same work demonstrate that it is possible to write instruction-set semantics using a preexisting general-purpose programming language, Haskell, to avoid the quadratic problem, although the authors admit it is a tradeoff space due to losing the advantages of being able to customize the DSL used to their specific use case, specifying ISA semantics. The same work was also able to use preexisting tools to convert from Haskell to Coq, for theorem proving about software; as well as from Haskell to Verilog, a hardware description language which can be used for hardware model-checking. Bourgeat et al. cite two key features of Haskell which enabled this formalization, which were monads and type classes.

[7] introduced monads as a way to make managing side effects in functional programming more tenable. Instead of modifying types to also include side-effect information and modifying functions that use those types as well, monads were defined as a constrained

triple of a type constructor `M` and two methods with signatures `unitM :: a -> M a` and `bindM :: M a -> (a -> M b) -> M b`, and [6] use monads to abstract over effects such as nondeterministic execution, input/output, and exceptions. Meanwhile, [8] introduced type classes as a way to define a type together with a set of associated operations that type must support. [6] uses type classes in two different ways. First, they define a `MachineWidth` type class to abstract over both 32-bit and 64-bit register values. Second, and more importantly, they define a `RiscvMachine` type class to represent a RISC-V processor to abstract over the primitives that are needed to represent state and control flow, but also abstracting over the choice of monad and register width.

However, [6] had implemented specifications for only some of the extensions of the RISC-V ISA family, namely the I (Integer), M (Integer Multiply/Divide), Zicsr (Control and Status Registers), A (Atomics), and F (Single Floating-Point) extensions. One of the extensions not covered by this prior work is the V (Vector) extension.

[9] describes how the primary difference between a vector instruction and a scalar instruction is that the former specifies more operations within an instruction than the latter, because emulating vector instructions using scalar instructions would require specifying control flow and memory access-related bookkeeping that would be implicit in the vector instruction. According to RISC-V International [10], version 1.0 of the RISC-V Vector Extension aims to improve software power efficiency through the instruction set's density of operations per instruction and mentions that managing power use and data movement may be important considerations in fields like artificial intelligence, machine learning, and computer vision.

The V extension has been formalized in Sail as a continuation of the work by Armstrong et al. [5], which supports the claim that the custom DSL could be used for formalizing it. This thesis seeks to answer the analogous question of whether a general-purpose language can be used to specify the same extension, by building off of the model produced by [6], given the difference between the V extension and other extensions previously formalized. I tentatively answer "yes" based on progress towards formalization of a minimal interesting fragment of this ISA extension in Haskell and Coq, and I present this progress along with future work to be done on this formalization and possible limitations thereof.

The remainder of this thesis will be structured as follows. First, I will briefly introduce the programming model of the RISC-V V ISA, focusing on the significant technical aspects that differentiate it from previously formalized RISC-V extensions. Then, I will explain the progress made towards the specification in Haskell and Coq. Next, I will explain possible limitations and issues encountered in this specification. Finally, I will conclude with discussion of future work in terms of fully formalizing this extension and technical aspects of the extension that have not been covered by this work but may be of note.

Chapter 2

Implementation

2.1 RISC-V V Extension Programming Model

From the RISC-V V specification [11], this extension introduces 32 additional vector registers of an implementation-defined bitwidth, as well as 7 control-and-status registers. The three most relevant control-and-status registers (CSRs) to this thesis' scope are `vl`, `vlenb`, and `vtype`. The first two of these have simpler definitions than the last. `vl` holds the number of data elements that a vector instruction will write to with its results, while `vlenb` holds the number of bytes that a vector register has. The reason why these would be useful to keep track of is that, as supported by RISC-V International's article on the extension [10], the assembly written in this extension is meant to be "agnostic" of the actual bitwidth of the vector registers. This does not imply that one execution of a single vector instruction should always consume or produce the same amount of data on a processor with 128-bit vector registers and a processor with 65536-bit vector registers, but that the code for, say, looping through two arrays of a large size, adding them, and writing back to memory should work with the same code across both processors, although the number of times that each vector instruction would be run would differ.

The third CSR, `vtype`, is comprised of multiple fields whose values affect the semantics of most of the instructions in the extension. Table 2.1 lists the names of fields relevant to instruction semantics, with a brief summary of each one's functionality.

2.2 Scope and Overview

There are on the order of hundreds of instructions in the V extension [11], for operations on integers, fixed-point, and floating-point data. For scope reasons, the entirety of the extension could not be formalized, so this work selected the instructions listed in table 2.2 to specify first, as a proof-of-concept. These instructions together demonstrate critical aspects of the vector ISA, which are moving data into and out of vector registers and doing computation on data within vector registers.

The scope of formalization of these instructions consists of two major halves. First, the RISC-V formalization by [6], written in Haskell, was extended to also decode and execute the instructions previously mentioned. Then, the same formalization, once extended, was

Table 2.1: Fields of the `vtype` control and status register and their functionality.

Field name	Description
`vsew` (Vector Selected Element Width)	Each vector register is assumed to contain multiple data elements of the same width. This field encodes that assumed bitwidth, for computing, e.g. `vl`.
`vlmul` (Vector Register Grouping)	A single vector instruction may treat a group of contiguous vector registers as a single operand. This field encodes the number of vector registers per group.
`vma` and `vta` (Vector Tail Agnostic and Vector Mask Agnostic)	In some cases, an instruction may use a mask to only write certain elements and not others, or an instruction only writes to the first portion of bits of a register and not the remainder. These flags define whether the processor must leave the bits not written to as they were, or if it is allowed to arbitrarily overwrite them with 1's.

Table 2.2: The instructions of the RISC-V V ISA formalized in this thesis

Type of instruction	Specific instruction(s)
Configuration-Setting Instructions	• `vsetvli rd, rs1, vtypei` • `vsetivli rd, uimm, vtypei` • `vsetvl rd, rs1, rs2`
Vector Unit-Stride Instructions	• `vle<eew>.v vd, (rs1), vm` • `vse<eew>.v vs3, (rs1), vm`
Vector Load/Store Whole Register Instructions	• `vl<nf>e<eew>.v vd, (rs1)` • `vs<nf>e<eew>.v vs3, (rs1)`
Vector Single-Width Integer Add	• `vadd.vv vd, vs2, vs1, vm`

translated into Coq using the same tool [6] used, `hs-to-coq` [12], and progress was made towards adapting the translated definitions into theorems usable by the Bedrock2 compiler, a formally verified compiler first introduced in [13] for a C-like language written in Coq which compiles down to RISC-V assembly.

The following three sections will outline the following. First, more concrete background will be provided on the RISC-V formalization by [6], to provide context for this thesis' implementations. Then, the following two sections will outline the implementations in more detail in Haskell and Coq, respectively. An artifact containing the Haskell implementation, the Coq translation, and the translation tool used is available at [14].

2.3 Overview of Prior RISC-V Specification by Bourgeat et al. [6]

This section will first discuss in more detail the Haskell specification of [6]. Then, it will go over the mechanics of the translation from Haskell to Coq, and finally will go over the parts of the specification in Coq not directly derived from Haskell.

First, on the Haskell side, as mentioned previously a RISC-V machine is modelled as an instance of a type class whose type constructor takes two parameters, a monad and an instance of type class `MachineWidth`, which represents the type of data stored in registers. Listing 2.1 shows this type signature as well as some of the relevant methods instances of this type class are required to implement. Note that in said listing, `CSRField` represents a field of one of the control and status registers.

```
class (Monad p, MachineWidth t) => RiscvMachine p t | p -> t where
  getRegister  :: Register -> p t
  setRegister  :: Register -> t -> p ()
  loadByte :: SourceType -> t -> p Int8
  storeByte :: SourceType -> t -> Int8 -> p ()
  getCSRField :: CSRField -> p MachineInt
  unsafeSetCSRField :: (Integral s) => CSRField -> s -> p ()
  -- other methods
```

Listing 2.1: Haskell `RiscvMachine` abbreviated signature.

The execution of a given instruction is then specified as in Listing 2.2. Note that the actual execution semantics per instruction is delegated to helper functions which are defined for each of the extensions formalized and a sentinel value for failed decoding (lines 4 through 13). This code listing also demonstrated the do-notation used in Haskell to chain monadic actions, in this case being the execution (lines 3-13), followed by the getting and setting of CSRFields `Field.MCycle` and `Field.MInstRet`.

```
1 execute :: (RiscvMachine p t) => Instruction -> p ()
2 execute inst = do
3   case inst of
4      IInstruction     i      -> I.execute   i
5      MInstruction     i      -> M.execute   i
```

```
      AInstruction    i       -> A.execute   i
      FInstruction    i       -> F.execute   i
      I64Instruction  i       -> I64.execute i
      M64Instruction  i       -> M64.execute i
      A64Instruction  i       -> A64.execute i
      F64Instruction  i       -> F64.execute i
      CSRInstruction  i       -> CSR.execute i
        InvalidInstruction   i -> raiseExceptionWithInfo 0 2 i
   cycles <- getCSRField Field.MCycle
   setCSRField Field.MCycle (cycles + 1)
   instret <- getCSRField Field.MInstRet
   setCSRField Field.MInstRet (instret + 1)
```

Listing 2.2: Haskell `execute` definition.

As for more detail about the translation, the Haskell definitions of decoding bytes into instructions and the files defining execution are translated automatically into Coq using hs-to-coq [12], while other files were manually translated. However, the translating tool isn't always able to identify the intended translation of the Haskell constructs used in the semantics, either because the constructs, while built-in to one of Haskell's standard libraries, aren't supported by the tool; or because the types involved are defined by the specification. Hence, the translation uses manually written edit files which specify certain translation patterns to be used. The snippet in Listing 2.3 is from this thesis' edit files, to give a concrete example. Lines 2 through 12 and 22 through 25 of this listing refer to converting references helper functions defined in the vector execution semantics in Haskell into corresponding helper methods in Coq, while lines 14 through 19 refer to renaming Haskell's standard-library methods and types into their Coq equivalents. Note that `Int8` and `Word8`, both used in Haskell to represent bytes, are mapped to the same type in Coq, `w8`, which is indicated by how the translations in lines 22 through 25 map converting between the two Haskell functions into identity functions in Coq.

```
1
2  rewrite forall x y, Spec.ExecuteV.take_machineInt x y = List.upto x y
3  skip Spec.ExecuteV.take_machineInt
4
5  rewrite forall x y, Spec.ExecuteV.drop_machineInt x y = List.from x y
6  skip Spec.ExecuteV.drop_machineInt
7
8  rewrite forall x, Spec.ExecuteV.length_machineInt x = Coq.ZArith.BinInt.Z.of_nat (length x)
9  skip Spec.ExecuteV.length_machineInt
10
11 rewrite forall x y, Spec.ExecuteV.index_machineInt x y = List.get x y
12 skip Spec.ExecuteV.index_machineInt
13
14 rename type GHC.Maybe.Maybe = option
15 rename value GHC.Maybe.Just = Some
16 rename value GHC.Maybe.Nothing = None
17
```

```
rename type GHC.Int.Int8 = w8
rename type GHC.Word.Word8 = w8


rewrite  forall , Spec.ExecuteV.int8_toWord8 = (fun x => x)
skip  Spec.ExecuteV.int8_toWord8
rewrite  forall , Spec.ExecuteV.word8_toInt8 = (fun x => x)
skip  Spec.ExecuteV.word8_toInt8
```

Listing 2.3: Snippet of edit file for vector extension semantics annotating specific translations for translation tool to use

Lastly, 4 of the key definitions in the Coq semantics that aren't directly derived from Haskell are listed in Listing 2.4. `RiscvProgram` is most closely a direct, manual translation of the `RiscvMachine` definition in Haskell. `mcomp_sat` in `PrimitivesParams` represents a predicate attached to executions defining when a postcondition is satisfied by a monadic computation. `spec_Bind` and `spec_Return` are specific definitions for `mcomp_sat` for the two primitives of a monad, `Bind` and `Return`, and lastly class `Primitives` provides similar definitions for primitive operations of a RISC-V machine.

```
Class RiscvProgram{M}{t}`{Monad M}`{MachineWidth t} := mkRiscvProgram {
   getRegister :  Register  -> M t;
   setRegister :  Register  -> t -> M unit;
  loadByte    : SourceType -> t -> M w8;
   storeByte    : SourceType -> t -> w8 -> M unit;
  getCSRField : CSRField -> M MachineInt;
  setCSRField : CSRField -> MachineInt -> M unit;
    (* Other methods omitted *)
}

Class PrimitivesParams(Machine: Type) := {
    mcomp_sat: forall {A: Type}, M A -> Machine -> (A -> Machine -> Prop) -> Prop;
    (* Other methods omitted *)
  }.

Class mcomp_sat_spec{Machine: Type}(p: PrimitivesParams Machine): Prop := {
    spec_Bind{A B: Type}: forall ( initialL : Machine) (post: B -> Machine -> Prop)
                                 (m: M A) (f : A -> M B),
        ( exists mid: A -> Machine -> Prop,
            mcomp_sat m initialL mid /\
            ( forall  a middle, mid a middle -> mcomp_sat (f a) middle post)) <->
        mcomp_sat (Bind m f) initialL post;

    spec_Return{A: Type}: forall ( initialL : Machine)
                                 (post: A -> Machine -> Prop) (a: A),
        post a  initialL  <->
        mcomp_sat (Return a) initialL post;
  }.
```

```

Class  Primitives (primitives_params:  PrimitivesParams  RiscvMachine):  Prop := {
    #[global]  mcomp_sat_ok :: mcomp_sat_spec primitives_params;

    spec_getRegister:   forall   ( initialL  :  RiscvMachine) (x:  Register )
                                 (post:  word -> RiscvMachine -> Prop),
        ( valid_register  x /\
         match map.get initialL.(getRegs) x with
         | Some v => post v initialL
         | None => forall v,  is_initial_register_value  v -> post v  initialL
         end) \/
        (x = Register0  /\ post  (word.of_Z 0)  initialL )  ->
        mcomp_sat (getRegister x)  initialL   post;

    spec_loadByte: spec_load 1 (Machine.loadByte (RiscvProgram := RVM)) Memory.loadByte;

    spec_storeByte: spec_store 1  (Machine.storeByte  (RiscvProgram := RVM)) Memory.storeByte;
    (* Other lemmas omitted *)
  }.
```

Listing 2.4: Abbreviated definitions for Coq semantics.

2.4 Haskell Specification

This section will first discuss how the Haskell specification was tested before translation into Coq and then will give a brief overview of implementation.

The implementation in Haskell was evaluated for correctness by simulating RISC-V V assembly which was produced by compiling test cases written in C using the GCC compiler. These test cases used compiler intrinsics available for using RISC-V vector instructions directly and were adapted from [15], which specifies these intrinsics and provided examples for them.

The steps taken to adapt the previous Haskell specification by [6] to the vector instructions listed in the previous section include modifying the parser to recognize the binary encoding of these instructions, modifying the machine representation to support the state that these instructions read from and write to, and writing out the actual execution.

Besides the definition of additional control and status registers as mentioned earlier, the primary modification to the type class defined by [6] to represent a RISC-V machine was the addition of two more primitives, as shown in Listing 2.5. These represent getting and setting the contents of a vector register. A vector register is represented as an array of bytes for convenience, as the specification's semantics divide the contents of a vector register into data elements whose width is always byte-aligned and is determined by the instruction and/or control and status registers.

```
class (Monad p, MachineWidth t) => RiscvMachine p t | p -> t where
  -- other previously defined methods
  getVRegister :: VRegister -> p [Int8]
  setVRegister :: VRegister -> [Int8] -> p ()
```

Listing 2.5: Machine definition edits in Haskell.

To demonstrate these primitives in action, Listing 2.6 shows the Haskell definition of pairwise addition between vectors (starting at line 32 of the listing) as well as a couple of helper functions. Seen again in this listing is the do-notation mentioned previously for chaining monadic computation. Of note in this listing is that at lines 34 through 40, CSR register values are being read and are affecting how the instructions are being executed; for example, line 63 executes conditionally on whether the CSR register `vtypei` has been set to be mask-agnostic or not.

Lines 42-44, 47 and 51-54 demonstrate the setup in computing which vector register and which part of the vector register correspond to a single data element being operated on from each of the two operand vector registers. This computation is necessary because the selected element width, which is the width of the data element, is read from `Field.VSEW` and isn't a constant across all executions of the instruction. Line 47 in particular also gives range bounds over the total number of data elements to operate on, where the lower bound is given by CSR field `Field.VStart`, and the upper by CSR field `Field.VL`. The latter's role is discussed earlier in the thesis, while `Field.VStart` serves as a record of the "loop counter" while this instruction executes. The use of `Field.VStart` is more relevant in the context of instructions being interrupted midway through, which is outside the scope of this thesis but is permitted by the specification.

Also of note is the use of the `combineBytes` helper function at lines 58 and 59 of the listing. This helper function had already existed previously, but it used the `Word8` datatype to represent a byte, whereas the machine primitive for loading and storing bytes to and from memory were defined with signatures `loadByte :: SourceType -> t -> p Int8` and `storeByte :: SourceType -> t -> Int8 -> p ()` , respectively, necessitating the `int8_toWord8` conversion function. `combineBytes` at line 58, as well as the definitions of helper functions `getVRegisterElement` and `setVRegisterElement` at lines 1 and 16 respectively, all take advantage of and demonstrate the utility of representing a vector register as a list of bytes, as the selected element width is configurable in multiples of bytes. `getVRegisterElement` and `setVRegisterElement` are effectively methods for taking a slice of an array and replacing a contiguous segment of an array of bytes.

Finally, lines 62, 63, and 66 represent the persistent effects of the vector instruction's execution on state. Lines 62 and 63 update the destination vector register based on the mask bit set in the instruction and the mask-agnostic policy defined by CSR field `Field.VMA`. Meanwhile, line 66 updates the tail end of the last destination vector register based on the tail-agnostic policy defined by CSR field `Field.VTA`. A concrete example of when this tail portion may be executed is if `Field.VLMul` is in effect $\frac{1}{2}$, which may be of use when a future vector operation is planned which may output data elements that are wider than their inputs. In this scenario, the back half of the vector register would be the tail in this operation and would be updated as mentioned previously.

```
getVRegisterElement :: forall p t. (RiscvMachine p t) => MachineInt -> VRegister ->
    MachineInt -> p [Int8]
getVRegisterElement eew baseReg eltIndex =
  if (eew == 1 || eew == 2 || eew == 4 || eew == 8)
  then
    do
      vlenb <- getCSRField Field.VLenB
      vregValue <- getVRegister baseReg
      let value = take_machineInt ( eew) (drop_machineInt ( (eltIndex * eew)) vregValue) in
        if (length_machineInt value) == ( eew)
        then return value
        else raiseException 0 2
  else
    raiseException 0 2


setVRegisterElement :: forall p t. (RiscvMachine p t) => MachineInt -> VRegister ->
    MachineInt -> [Int8] -> p ()
setVRegisterElement eew baseReg eltIndex value =
  if (eew == 1 || eew == 2 || eew == 4 || eew == 8)
  then
    do
      vlenb <- getCSRField Field.VLenB
      vregValue <- getVRegister baseReg
      let newVregValue =
            (take_machineInt ( (eltIndex * eew)) vregValue) ++
            (value) ++
            (drop_machineInt ( ((eltIndex + 1) * eew)) vregValue) in
        if (length_machineInt newVregValue) == (length_machineInt vregValue)
        then (setVRegister baseReg newVregValue)
        else raiseException 0 2
   else raiseException 0 2

execute (Vaddvv vd vs1 vs2 vm) =
  do
     vstart <- getCSRField Field.VStart
    vlmul <- getCSRField Field.VLMul
    vlenb <- getCSRField Field.VLenB
    vl <- getCSRField Field.VL
    vma <- getCSRField Field.VMA
    vta <- getCSRField Field.VTA
    vsew <- getCSRField Field.VSEW
    vmask <- getVRegister 0
    let eew = 2 ^ (fromMaybe 0 (translateWidth_Vtype vsew))
        maxTail = computeMaxTail vlmul vlenb (eew)
        eltsPerVReg = (vlenb * 8) `quot` (eew)
      in
```

```haskell
    do
      forM_ [vstart ..( vl-1)]
        (\i ->
           do
             let
               realVd  = vd + ( (i `quot` eltsPerVReg))
               realVs1  = vs1 + ( (i `quot` eltsPerVReg))
               realVs2  = vs2 + ( (i `quot` eltsPerVReg))
                realEltIdx  = (i `rem` eltsPerVReg)
             vs1value <- getVRegisterElement (eew `quot` 8) realVs1 realEltIdx
             vs2value <- getVRegisterElement (eew `quot` 8) realVs2 realEltIdx
             let
               vs1Element = ((combineBytes :: [Word8] -> MachineInt) (map int8_toWord8
                   vs1value))
               vs2Element = ((combineBytes :: [Word8] -> MachineInt) (map int8_toWord8
                   vs2value))
               vdElement = vs1Element + vs2Element
             ( (setCSRField Field.VStart i))
             when (vm == 0b1 || (testVectorBit vmask i)) (setVRegisterElement (eew `quot` 8)
                 realVd  realEltIdx  (map word8_toInt8 (splitBytes (eew) vdElement)))
             when (vm == 0b0 && (not (testVectorBit vmask i) && (vma == 0b1))) (
                 setVRegisterElement (eew `quot` 8) realVd realEltIdx (replicate_machineInt (eew
                  `quot` 8) (complement (zeroBits))))
             setCSRField Field.VStart i
        )
      when (vta == 0b1) (forM_ [vl..(maxTail-1)]
                          (\i ->
                             let realVd  = vd + ( (i `quot` eltsPerVReg))
                                 realEltIdx  = (i `rem` eltsPerVReg)
                             in do
                               setVRegisterElement (eew `quot` 8) realVd  realEltIdx  (
                                   replicate_machineInt  (eew `quot` 8) (complement (zeroBits))
                                   )))
      setCSRField Field.VStart 0b0
```

Listing 2.6: Haskell code for vector-vector add implementation.

2.5 Coq Translation

The Haskell specification, once tested on the aforementioned test cases, was then translated into Coq using preexisting automatic translation tool `hs-to-coq` [12]. However, this translation was not fully automatic, for a couple of reasons.

First, `hs-to-coq` in translating does not necessarily know how to translate all the Haskell constructs used into Coq, and thus specific find-and-replace patterns were developed for the translation. For example, `Int8` and `Word8` in Haskell were both translated as the same type in Coq, so the aforementioned `int8_toWord8` conversion function was manually annotated as being equivalent to the identity function in Coq. Besides this, some of Haskell's standard constructs for working with monads (such as `Data.Traversable.forM` and `Data.Foldable.forM_`) were not recognized by the translator because [6] defined their own monad type for the Coq translation, and so the manual annotation for those two functions would be to mark as equivalent to utility functions that were written in Coq.

Second, the automatic translation only covered directly translating the specification of the decoding and execution of the instruction and did not, for example, update the Coq representation of a RISC-V machine to also have vector registers. Thus, this too was manually updated for the Coq equivalent of the base RISC-V machine type class, and the preexisting proofs about instances of this machine were partially updated as well.

The Coq translation of the modified Haskell semantics as contained in the `riscv-coq` subfolder of the previously mentioned artifact was successfully compiled by resolving most proofs, except for proofs less directly relevant to semantics, e.g. encoding and decoding being inverses. The Coq definitions listed in Listing 2.4 were also modified to support the equivalent of the operations added in Listing 2.5. Listing 2.7 presents the signature for the Coq translation of `getVRegister` and `setVRegister`, as well as how `mcomp_sat`, the definition of a monadic computation satisfying a postcondition, was defined for these primitives. `spec_getVRegister` and `spec_setVRegister` can be interpreted as stating that if the vector register argument to these primitives is within the 32 vector registers, and either the value read from or written to the vector register along with the rest of the machine state satisfy the postcondition, then the respective operations' execution will result in state that satisfies the postcondition.

```
Class RiscvProgram{M}{t}`{Monad M}`{MachineWidth t} := mkRiscvProgram {
  (* Unmodified lines omitted *)
  getVRegister : VRegister -> M (list w8);
  setVRegister : VRegister -> (list w8) -> M unit;
};

Class Primitives (primitives_params: PrimitivesParams RiscvMachine): Prop := {
(* Unmodified lines omitted *)
    spec_getVRegister: forall ( initialL : RiscvMachine) (x: Register)
                              (post: ( list w8) -> RiscvMachine -> Prop),
        ( valid_vregister x /\
         match map.get initialL.(getVRegs) x with
         | Some v => post v initialL
         | None => forall v, is_initial_vregister_value v -> post v initialL
         end) ->
        mcomp_sat (getVRegister x) initialL post;

    spec_setVRegister: forall ( initialL : RiscvMachine) (x: VRegister) (v: list w8)
                              (post: unit -> RiscvMachine -> Prop),
        ( valid_vregister x /\ valid_vregister_value v /\ post tt (withVRegs (map.put initialL.(
            getVRegs) x v) initialL )) ->
         mcomp_sat (setVRegister x v) initialL post;

(* Unmodified lines omitted *)
}.
```

Listing 2.7: Computation-satisfies-postcondition predicate for new primitives.

However, I was unable to adapt the Coq translation into theorems suitable for interoperation with the compiler theorems for Bedrock2 in terms of precondition-postcondition semantics. In the next chapter I will discuss the reasons why this was not completed as well as other limitations and difficulties encountered in the Haskell and Coq implementations.

Chapter 3

Limitations and Difficulties

In spite of the partial success to which the specification of the selected vector instructions in Coq and Haskell were executed, I identify two main difficulties that were encountered in this translation.

3.1 Different, Incompatible Numeric Types

First, unlike the extensions previously formalized by [6], the type of data element that a vector register contains is not fixed. In other words, a 128-bit vector register may hold 16 8-bit elements or 2 64-bit elements, and the intended interpretation is up to the context of the function being executed.

For the previous formalizations, a single type class `MachineWidth` was sufficient to represent a register value, and arithmetic could be done solely between members of that type class, before being converted back to bytes if being written to memory. However, given this flexibility in the vector specification, conversion between a vector register as an array of bytes and as an array of data elements has to be defined for every allowed data element type. This includes 8, 16, 32, and 64-bit integer types, as well as, though not covered within this thesis, fixed-point and floating-point values. Care will also need to be taken to ensure that overflow and edge cases for arithmetic for each of these types is properly handled in Coq and Haskell. An additional concern in terms of arithmetic edge cases is that the vector extension supports multiple variations of some arithmetic operations, depending on behavior on overflow. For example, the vector add formalized in this thesis is supposed to wrap around on overflow (although this wasn't tested empirically in Haskell), while there are also variants of the same for data-element widening (i.e., the output is twice the bitwidth of the inputs) and add-with-carry. Prior work on building a live-verification framework atop Bedrock2 also observed the complication of supporting multiple distinct numeric types simultaneously [16], noting Coq's lack of subtyping as another contributing factor.

Aside from this, much of the Haskell formalization relied on manipulating lists of bytes; however, a source of friction on this was that Haskell's functions with lists and integers (e.g., indexing into a list, taking the first few elements) used a different integer type (`Integer`) than the integer type used to store integers in the machine's control and status registers (`MachineInt`, which is an alias for Haskell's `Int64`). The effects of this can be seen in

Listing 2.6, where helper functions with suffix `_machineInt` (e.g. `replicate_machineInt`, `drop_machineInt`) are used. These helper functions are wrappers around the Haskell standard-library functions on lists, using `fromIntegral` to convert between the integer types; however, `fromIntegral` is not supported by the translation tool from Haskell to Coq, hence these helper functions were also manually rewritten for the Coq translation.

3.2 Control-and-Status Registers Affect Semantics

The other primary difficulty encountered in implementing this specification in Coq is that unlike the extensions previously formalized by [6], the values read from and written to the control and status registers affect the execution of instructions. This can be seen in Listing 2.4, where under `Primitives`, specifications for reading from and writing to a register are defined, as well as loading and storing from memory. However, no equivalent was previously defined for `Machine.getCSRField` and `Machine.unsafeSetCSRField`, the CSR methods of type-class `RiscvMachine` listed in Listing 2.1.

This hinders the use of the Coq specification in the Bedrock2 compiler because the view that the compiler proofs had of a RISC-V machine prior to the formalization of the vector extension did not track either vector registers or CSRs. This additional state in the model of a RISC-V processor pervades through other theorems that use the RISC-V machine due to type mismatches and tactics not being strongly typed. Besides this, new semantics have to be defined for when an abstract machine's execution of binding the returned value of a CSR register satisfies a given postcondition, which may be more nuanced than at first glance since CSRs are not unique to the vector extension and thus may have other semantic significance elsewhere.

Chapter 4

Future Work and Conclusion

Besides completing the formalization of the vector instructions mentioned in this thesis, there are many more instructions in the same vector extension to be formalized. In this section I will go over interesting technical details of the vector instructions which may be worth exploring in a future extension of this thesis, followed by a conclusion.

Earlier in this thesis, it was mentioned that mask-agnostic and tail-agnostic flags were available for configuration. Currently, the specification is implemented deterministically, i.e. if the mask-agnostic flag is set, then all data elements not written to will be overwritten, which does conform to the specification but may be too specific, as a processor technically has the leeway to leave the elements undisturbed or leave some undisturbed and overwrite others.

Besides this, currently the specification is hardcoded to 64-bit vector registers, with a smallest data element supported of 8 bits. However, both the vector bit length and the smallest data element supported can both be higher and are implementation-defined, so parametrizing the specification over these values would be useful.

Finally, it would be of use to actually examine whether Bedrock2 can use these instructions to reduce code size while preserving verifiable correctness, through implementing a verified vectorization phase in the compiler.

In conclusion, based on progress towards formalizing a subset of the RISC-V vector extension using Haskell, it has been shown that the past work of [6] in formalizing RISC-V using abstract monads can be extended to vector instructions, although not without caveats particular to vector instructions and the RISC-V V extension.

References

[1] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. "CakeML: a verified implementation of ML". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: Association for Computing Machinery, 2014, pp. 179–191. ISBN: 9781450325448. DOI: 10.1145/2535838.2535841. URL: https://doi.org/10.1145/2535838.2535841.

[2] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand. "CompCert-a formally verified optimizing compiler". In: *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. 2016.

[3] A. W. Appel. "Verified Software Toolchain: (Invited Talk)". In: *European Symposium on Programming*. Springer. 2011, pp. 1–17.

[4] A. Armstrong et al. "ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS". In: *Proceedings of the 46th Annual ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL '19. Cascais, Portugal: Association for Computing Machinery, 2019. DOI: 10.1145/3290384. URL: https://doi.org/10.1145/3290384.

[5] RISC-V International. *RISCV Sail Model*. URL: https://github.com/riscv/sail-riscv.

[6] T. Bourgeat, I. Clester, A. Erbsen, S. Gruetter, P. Singh, A. Wright, and A. Chlipala. "Flexible Instruction-Set Semantics via Abstract Monads (Experience Report)". In: *Proc. ACM Program. Lang.* 7.ICFP (Aug. 2023). DOI: 10.1145/3607833. URL: https://doi.org/10.1145/3607833.

[7] P. Wadler. "The essence of functional programming". In: *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. 1992, pp. 1–14.

[8] P. Wadler and S. Blott. "How to make ad-hoc polymorphism less ad hoc". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 60–76. ISBN: 0897912942. DOI: 10.1145/75277.75283. URL: https://doi.org/10.1145/75277.75283.

[9] R. Espasa, M. Valero, and J. E. Smith. "Vector architectures: past, present and future". In: *Proceedings of the 12th International Conference on Supercomputing*. 1998, pp. 425–432.

[10] R.-V. International. *RISC-V Vector Processing is Taking Off | SiFive*. 2022. URL: https://riscv.org/blog/2022/06/risc-v-vector-processing-is-taking-off-sifive/ (visited on 07/31/2024).

[11] RISC-V International. *riscv-v-spec*. URL: https://github.com/riscv/riscv-v-spec.

[12] The Penn PL Club. *hs-to-coq*. URL: https://github.com/plclub/hs-to-coq.

[13] A. Erbsen, S. Gruetter, J. Choi, C. Wood, and A. Chlipala. "Integration verification across software and hardware for a simple embedded system". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 604–619. ISBN: 9781450383912. DOI: 10.1145/3453483.3454065. URL: https://doi.org/10.1145/3453483.3454065.

[14] Arthur De Belen. *Thesis artifact*. URL: https://github.com/0adb/deBelen-reinerdb-meng-eecs-2024-artifact.

[15] RISC-V Non-ISA Specifications. *rvv-intrinsic-doc*. URL: https://github.com/riscv-non-isa/rvv-intrinsic-doc/tree/v0.11.x.

[16] S. Gruetter, V. Fukala, and A. Chlipala. "Live Verification in an Interactive Proof Assistant". In: *Proc. ACM Program. Lang.* 8.PLDI (June 2024). DOI: 10.1145/3656439. URL: https://doi.org/10.1145/3656439.