A Reading Guide to FORVIS: A Formal RISC-V ISA Specification

Rishiyur S. Nikhil
Bluespec, Inc.

Revision: June 27, 2018

Abbreviations, acronyms and terminology and links

CSR	Control and Status Register
FPR	Floating Point Register
GPR	General Purpose Register
Hart	Hardware Thread. Not to be confused with software threads such as POSIX threads, "pthreads", and processes. A hart has, in hardware, its own PC and fetch unit, and can work concurrently with other harts
ISA	Instruction Set Architecture
PC	Program Counter
RVWMO	RISC-V Weak Memory Ordering (default memory model)
spec	Specification
Sv32	Virtual Memory System in RV32 systems
Sv39	Virtual Memory System in RV64 systems
Sv48	Optional additional Virtual Memory System in RV64 systems
WMM	Weak Memory Model

For more information on terminology and concepts, and information on RISC-V, we recommend these fine books:

- "The RISC-V Reader: An Open Architecture Atlas", by Patterson and Waterman [5]
- "Computer Architecture: A Quantitative Approach", by Hennessy and Patterson [1]
- "Computer Organization and Design: The Hardware/Software Interface" (RISC-V Edition) by Patterson and Hennessy [4]

and the RISC-V Foundation web site: `https://riscv.org`

Acknowledgments

Thanks to the original creators of RISC-V for making all this possible in the first place.

Thanks to Bluespec, Inc. for supporting this work.

Thanks to the RISC-V Foundation for constituting the ISA Formal Specification Technical Group.

Thanks to the members of the RISC-V Foundation's ISA Formal Specification Technical Group with whom we have had many wonderful discussions on a weekly basis that have inspired and clarified this work.

Contents

This page intentionally left blank

1 Introduction

This is a reading guide to FORVIS, a formal RISC-V ISA specification written in "extremely elementary" Haskell. It can be executed as a RISC-V simulator which, in turn, executes RISC-V binaries.

This is a work-in-progress, one of several similar concurrent efforts within the "ISA Formal Specification" Technical Group constituted by The RISC-V Foundation (`https://riscv.org`). We welcome your feedback, comments and suggestions.[1]

Forvis corresponds to these original English-text specs:

- *The RISC-V Instruction Set Manual, Volume I: User Level ISA*, Andrew Waterman and Krste Asanovic (eds.), Document Version 2.2, May 7, 2017. [7]
- *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Privileged Architecture Version 1.10*, Andrew Waterman and Krste Asanovic (eds.), Document Version 1.10, May 7, 2017. [8]:

1.1 Forvis goals

Forvis is a formal specification of the RISC-V Instruction Set Architecture, i.e., it is written in a precise, unambiguous language (here, Haskell) without regard to hardware implementation considerations; clarity and precision are paramount concerns. In contrast, specs written a natural language such as English are often prone to ambiguity, inconsistency and incompleteness. Further, a formal spec can be parsed and processed automatically, connecting to other formal analysis and transformation tools. In addition to precision and completeness, Forvis also has these goals:

- **Readability:** This spec should be readable by people who may be completely unfamiliar with Haskell or other formal specification languages. Examples of our target audience:
  - Compiler writers targeting RISC-V, as a reference explaining the instructions they generate.
  - RISC-V CPU hardware designers, as a refernce explaining the instructions interpreted by their designs.
  - Students studying RISC-V.
  - Designers of new RISC-V ISA extensions, who may want to extend these specs to include their extensions.
  - Users of formal methods, who wish to prove properties (especially correctness) of compilers and hardware designs.
- **Modularity:** RISC-V is one of the most modular ISAs. It supports:
  - A couple of base ISAs: RV32 (32-bit) and RV64 (64-bit) (an RV128 base is under development)
  - Numerous extensions, such as M (Integer Multiply/Divide), A (Atomic Memory Ops), F (single precision floating point), D (double precision floating point), C (compressed 16b insructions), E (embedded).

[1]Forvis, and this document, are available at: `https://github.com/rsnikhil/RISCV-ISA-Spec`

  - An optional Privilege Architecture, with M (machine) and optional S (supervisor) and U (user) privilege levels.
  - Implementation options, such as whether misaligned memory accesses are handled or cause a trap, whether interrupt delegation is supported or not, etc.

  Implementations can combine these flexibly in a 'mix-and-match' manner. Some of these options can coexist in a single implementation, and some may be dynamically switched on and off. Forvis tries to capture all these possibilities.

- **Concurrency and non-determinism:** RISC-V, like most modern ISAs, has opportunities for concurrency and legal non-determinism. For example, even in a single hart (hardware thread), it is expected that most implementations will have pipelined (concurrent) fetch and execute units, and that the instructions returned by the fetch unit may be unpredictable after earlier code that writes to instruction memory, unless mediated by a FENCE.I instruction. RISC-V has a Weak Memory Model, so that in a multi-hart system, memory-writes by one hart may be "seen" in a different order by another hart unless mediated by FENCE and AMO instructions. In particular, different implementations, and even different runs of the same program on the same implementation, may return different results from reading memory on different runs.

- **Executabality:** Forvis constitutes an "operational" semantics (as opposed to an "axiomatic" semantics). The spec can actually be executed as a Haskell program, representing a RISC-V "implementation", i.e., it can execute RISC-V binaries. The README file in the code repository explains how to execute the code.

1.1.1 Extension for concurrent behavior and weak memory models

Although it is convenient to directly execute this Haskell code as a Haskell program, thereby giving us a sequential RISC-V simulator for free, the code (specifically, the file `Forvis_Spec.hs`) can also be treated as a generic functional program with an alternate interpretation (non-Haskell, and changing what we mean by the "Machine State" that is an argument to each spec function).

Such an alternate interpreter can demonstrate all kinds of concurrencies (e.g., due to out-of-order execution, pipelining, different kinds of speculation, and more) and non-deterministic interaction with weak memory models. We believe it can describe the complete range of concurrent behaviors seen in actual implementations (and more concurrent behaviors not seen in practical implementations).

Describing this alternate interpretation is planned as a follow-up document. We have a general idea of how this concurrent interpreter works but are still working out the details. The concurrency is not exposed in the spec text, but is implicit in the data flow. The central ideas come from "implicit dataflow" computation (cf. "Implicit Parallel Programming in *pH*"[3]).

1.2 About the choice of Haskell, and the level of Haskell features used

We chose to use the well-known programming language Haskell [6] because it is a pure functional language, with no side effects. ISA specs are sometimes hard to read because of hidden state, and their updates by side-effect are hard to keep track of; in our Haskell code, all state is visible and all updates can be seen explicitly as recomputation of new state.

Forvis spec code is written in "extremely elementary" Haskell so that it is readable by people who may be totally unfamiliar with Haskell and who may have no interest in learning Haskell. It uses a *very* small, extremely simple subset of Haskell[2] (just simple types, function definition and function application) and none of the features that may be even slightly unfamiliar to the audience (no Currying/partial-application, lambda-expressions, laziness, typeclasses, monads, etc.) For those without prior exposure to Haskell, this document explains the minimal Haskell notation necessary to read the Forvis spec code.

Using extremely simple Haskell will also make it easier for authors of new ISA extensions to extend these specs to cover their ISA extensions, even if they are unfamiliar with Haskell.

Using extremely simple Haskell will also make it easy to parse and connect to other tools, such as proof assistants, theorem provers, and so on (including the alternate "concurrent" interpreter described at the end of the next section).

1.2.1 The code in this document is real

This document is produced with a kind of "literate programming" process (Knuth 1984 [2]). The Forvis spec *is* the collection of Haskell source code files, and this document is just a reading guide. All the code fragments herein are automatically extracted from the actual Haskell code during document production. As a reading guide, this document is not meant to be read on its own, but as an accompaniment to perusing the actual source code.

1.3 How to read the spec code

As mentioned earlier, the Forvis spec is Haskell source code. This document is just a reading guide, and contains code fragments automatically extracted from the actual source code. This document is not meant to be read on its own, but as a reference for clarification and commentary while you are reading the actual code.

For all readers, whether familiar with Haskell or not, this guide will help you navigate the source code; reading the code and files in the presented order may help you absorb the code most quickly.

Readers familiar with Haskell can skip the following sub-section.

1.3.1 Basic Haskell concepts and notation

Haskell is a pure functional language: everything is expressed as pure mathematical functions from arguments to results, and composition of functions. There is no sequencing, and no concept of updatable variables (traditional "assignment statement")

Each Haskell file is a Haskell module and has the form:

[2] We believe that the Haskell used here is simple enough that only minor syntactic transformation would be needed to render it into some other functional language such as SML, OCaml, or Scheme.

```
module module-name where
import another-module-name
...
import another-module-name
...
constant-or-function-or-type-definition
...
constant-or-function-or-type-definition
...
```

Comments begin with "--" and extend through the end of the line.

Haskell relies on "layout" to convey text structure, i.e., indentation instead of brackets and semi-colons. A constant definition looks like this:

```
foo :: type
foo =  value-expression
```

A function definition looks like this:

```
fn :: arg-type -> ... -> arg-type -> resul-type
fn arg ... arg = function-body-expression
```

Note: in Haskell, function arguments, both in definitions and in applications, are typically just juxtaposed and not enclosed in parentheses and commas, thus:

```
fn arg ... arg
```

instead of:

```
fn ( arg, ..., arg )
```

A definition like this:

```
type Instr = Word32
```

just defines a new type *synonym* (`Instr`) for an existing type (`Word32`); this is done just for readability.

A definition like this:

```
data newtype = ...
```

defines a new type; these will be explained as we go along.

For readability, large expressions are sometimes deconstructed using "`let`" expressions to provide meaningful names to intermediate sub-expressions, define local help-functions, etc. For example, instead of:

```
x + f y z - g a b c
```

we may write, equivalently:

```
let
    tmp1 = f y z
    tmp2 = g a b c
    result = x + tmp1 + tmp2
in
    result
```

Conditional expressions may be written using `if-then-else` which can of course be nested:

```
x = if cond-expr1
```

```
        then expr1
        else if cond-expr2
             then expr2
             else expr3
```

or using `case` which can also be nested:

```
x = case cond-expr1 of
         True ->  expr1
         False -> case cond-expr2 of
                       True -> expr2
                       False -> expr3
```

or may be folded into a definition:

```
x | cond-expr1 = expr1
  | cond-expr2 = expr2
  | True       = expr3
```

The following table shows some operators in Haskell and their counterparts in C, where the notations differ.

Haskell	C	
`not x`	`! x`	Boolean negation
`x /= y`	`x != y`	Not-equals operator
`x .&. y`	`x & y`	Bitwise AND operator
`x .|. y`	`x | y`	Bitwise OR operator
`complement x`	`~ x`	Bitwise complement
`shiftL x n`	`x << n`	Left shift
`shiftR x n`	`x >> n`	Right shift (arith if x is signed, logical otherwise)

2 File Arch_Defs.hs: basic architectural definitions

2.1 Base ISA type

The following defines a data type `RV` with two possible values, `RV32` and `RV64`. It is analogous to an "enum" declaration in C, defining a family of constants. The `deriving` clause says that Haskell can automatically extend the equality operator `==` to work on values of type `RV`, and that Haskell can automatically extend the `show()` function to work on such values, producing Strings `"RV32"` and `"RV64"`, respectively.

line 25 Arch_Defs.hs

```
data RV = RV32
        | RV64
        deriving (Eq, Show)
```

2.2 Instruction Fields

Below, we define the type `Instr` and `Instr_C` to be more readable synonyms for Haskell's `Word32` and `Word16` types (32-bit and 16-bit unsigned integers).

We use 32-bits here for all instruction fields, even though in practice they have fewer bits.

line 39 Arch_Defs.hs

```
type Instr   = Word32
type Instr_C = Word16

type InstrField = Word32

-- General-purpose registers

type GPR_Addr = InstrField

-- CSRs

type CSR_Addr = InstrField
```

We define a number of help-functions to extract fields from an instruction. For example, the function `ifield_opcode` takes an instruction as argument and returns the "bit slice" from bits 0 through 6 inclusive (equivalent to Verilog's `instr[6:0]`. Similarly, we have definitions for other fields of interest.

line 55 Arch_Defs.hs

```
ifield_opcode  :: Instr -> InstrField
ifield_opcode  instr = bitSlice instr  6  0

ifield_funct3  :: Instr -> InstrField
ifield_funct3  instr = bitSlice instr  14  12

ifield_rd      :: Instr -> InstrField
ifield_rd      instr = bitSlice instr  11  7

ifield_rs1     :: Instr -> InstrField
ifield_rs1     instr = bitSlice instr  19  15
```

RISC-V instructions come in a few standard formats. For example, the "B"-type format (for BRANCH instructions) consists of an opcode, a 3-bit function code, two source registers rs1 and rs2, and a 12-bit immediate value assembled out of various bits in the instruction. We define a function `ifields_B_type` that, given an instruction, returns these fields:

line 111 Arch_Defs.hs

```
ifields_B_type :: Instr -> (InstrField, InstrField, InstrField, InstrField, InstrField)
ifields_B_type  instr =
  let imm12  = ( shift (bitSlice instr  31  31) 11  .|.
                 shift (bitSlice instr   7   7) 10  .|.
                 shift (bitSlice instr  30  25)  4  .|.
                 shift (bitSlice instr  11   8)  0  )
      rs2    = ifield_rs2     instr
      rs1    = ifield_rs1     instr
      funct3 = ifield_funct3  instr
```

```
     opcode = ifield_opcode  instr
  in
    (imm12, rs2, rs1, funct3, opcode)
```

Similar functions are defined for the other standard formats. These instructions use `bitSlice`, shift functions, bitwise OR (`.|.`) to extract the fields.

2.3 Exception Codes

We define a type for exception codes, and the values of all the standard exception codes:

line 171 Arch_Defs.hs

```
type Exc_Code = Word64

exc_code_u_software_interrupt      :: Exc_Code;    exc_code_u_software_interrupt      =  0;
exc_code_s_software_interrupt      :: Exc_Code;    exc_code_s_software_interrupt      =  1;
```

...

line 185 Arch_Defs.hs

```
exc_code_instr_addr_misaligned     :: Exc_Code;    exc_code_instr_addr_misaligned     =  0;
exc_code_instr_access_fault        :: Exc_Code;    exc_code_instr_access_fault        =  1;
```

2.4 Memory responses

We define a type `Mem_Result` for responses from memory. This may be `Mem_Result_Ok` (successful), in which case it returns a value (irrelevant for STORE instructions, but relevant for LOAD, load-reserved, store-conditional, and AMO ops). Otherwise it is a `Mem_Result_Err`, in which case it returns an exception code (such as misalignment error, an access error, or a page fault.)

line 234 Arch_Defs.hs

```
-- Either Ok with value, or Err with an exception code

data Mem_Result = Mem_Result_Ok   Word64
                | Mem_Result_Err  Exc_Code
                deriving (Show)
```

When returning a result, we construct expressions like these:

`Mem_Result_Ok` *value-expression*
`Mem_Result_Err` *exception-value-expression*

When fielding a result, we deconstruct it using a case-expression like this:

```
case mem-result of
  Mem_Result_Ok v   -> use v in an expression
  Mem_Result_Err ec -> use ec in an expression
```

2.5 Privilege Levels

RISC-V defines 3 standard privileve levels: Machine, Supervisor and User:

line 242 Arch_Defs.hs
```
-- Machine, Supervisor and User

type Priv_Level = InstrField

m_Priv_Level :: Priv_Level;    m_Priv_Level  = 3
s_Priv_Level :: Priv_Level;    s_Priv_Level  = 1
u_Priv_Level :: Priv_Level;    u_Priv_Level  = 0
```

3 File Machine_State.hs: architectural and machine state

[Reminder: this is for the simple, sequential, one-instruction-at-a-time interpreter. The concurrent interpreter has a substantially different machine state.]

3.1 Handling RV32 and RV64 simultaneously

Although each hardware implementation will typically be either an RV32 system or an RV64 system, the spec encompasses implementations that can simultaneously support both. For example, machine-privilege code may run in RV64 mode while supervisor- and user-privilege code may run in RV32 mode. There is also a future RV128 being defined.

In Forvis, which covers RV32 and RV64 and their simultaneous use, we represent everything using 64 bits. The semantics of each instruction are defined to be governed by the current RV setting which is available in the architectural state (specifically, MISA.MXL, MSTATUS.SXL, MSTATUS.UXL, etc.). An RV32 setting can render some instructions illegal, and limits calculations on values to be done with 32-bit arithmetic.

3.2 Machine State

We define a new type representing a complete "machine state", which is just a record or struct. The first few fields represent a RISC-V hart's basic architectural state: a Program Counter, General Purpose Registers, Control-and-Status Registers, and the current privilege level at which it is running: This is followed by two fields representing memory and memory-mapped I/O devices, and finally by fields that are not semantically relevant and are used just for redundant information, simulation configuration options, simulation state, gathered statistics, and so on.

line 36 Machine_State.hs
```
data Machine_State =
  Machine_State { -- Architectural state
                  f_pc   :: Word64,
                  f_gprs :: GPR_File,
                  f_csrs :: CSR_File,
                  f_priv :: Priv_Level,

                  -- Memory and mory mapped IO
```

```
                 f_mem  :: Mem,
                 f_mmio :: MMIO,

                 -- Implementation options
                 f_mem_addr_ranges :: [(Word64, Word64)],    -- list of (addr_start, addr_lim)

                 -- For convenience and debugging only; no semantic relevance
                 f_rv        :: RV,   -- redundant copy of info in CSR MISA
                 f_verbosity :: Int,
                 f_run_state :: Run_State
               }
```

This record-with-fields is a purely internal representation choice in this module. Clients of this module only access it via the `mstate_`*function* API that follows.[3]

The following function is a constructor that returns a new machine state.

line 84 Machine_State.hs

```
-- Make a Machine_State, given initial PC and memory contents
mkMachine_State :: RV -> Word64 -> [(Word64,Word64)] -> ([(Int, Word8)]) -> Machine_State
mkMachine_State  rv  initial_PC  addr_ranges  addr_byte_list =
  Machine_State {f_pc   = initial_PC,
                 f_gprs = mkGPR_File,
                 f_csrs = mkCSR_File  rv,
                 f_priv = m_Priv_Level,

                 f_mem             = mkMem  addr_byte_list,
                 f_mmio            = mkMMIO,
                 f_mem_addr_ranges = addr_ranges,

                 f_rv        = rv,
                 f_verbosity = 0,
                 f_run_state = Run_State_Running}
```

Two typical API functions on the machine state are these, to read and write the PC.

line 103 Machine_State.hs

```
mstate_pc_read :: Machine_State -> Word64
mstate_pc_read  mstate = f_pc mstate

mstate_pc_write :: Machine_State -> Word64 -> Machine_State
mstate_pc_write  mstate  val = mstate { f_pc = val }
```

In the API function `mstate_gpr_write`, we ensure that if we are in RV32 mode, we sign-extend the lower 32 bits of the written value, before using the "raw" `gpr_write` function:

line 128 Machine_State.hs

```
mstate_gpr_write :: Machine_State -> GPR_Addr -> Word64 -> Machine_State
mstate_gpr_write  mstate  reg  val =
  let
    rv      = f_rv  mstate
```

[3]Haskell has export-import mechanisms to enforce this external invisibility of our representation choice, but we have omitted them to avoid clutter.

```
    val1 | rv == RV32 = signExtend  val  32
         | rv == RV64 = val
    gprs    = f_gprs  mstate
    gprs'   = gpr_write  gprs  reg  val1
    mstate' = mstate { f_gprs = gprs' }
  in
    mstate'
```

"Raw" reads and writes on the GPR file are described in the file `GPR_File.hs`.

4 File Forvis_Spec.hs: the ISA spec

The entire spec is essentially in this one file. The major sections are:

- A function `instr_fetch` expressing instruction-fetch, returning a regular 32-bit instruction, or 'C' compressed 16-bit instruction, or an instruction-fetch fault.
- A large number of functions named `spec_`*OPCODE* describing the semantics of all RISC-V instructions.
- A small number of functions name `finish_`*scheme* capturing the few common schemes by which instructions finish (write Rd, increment PC, increment MINSTRET, trap, ...)
- A function `mstate_upd_on_trap` (which is perhaps the most intricate) that updates the machine state for a trap. It computes the new privilege level, new PC, new MSTATUS, new MEPC/SEPC/UEPC, new MCAUSE/SCAUSE/UCAUSE, and new MTVAL/STVAL/UTVAL based on whether it was an interrupt or synchronous trap, the current privilege level, the MSTATUS register, MIP and MIE registers, MIDELEG and MEDELEG registers, MTVEC/STVEC/UTVEC
- A function `exec_instr` (and it's counterpart `exec_instr_C` for 'C' compressed instructions) that uses all the `spec_`*OPCODE* to update the machine state by executing exactly one instruction.
- A function `take_interrupt_if_any` that checks the machine state to see if an interrupt is pending and updates the machine state if so (to be poised at the trap vector).

4.1 Instruction fetch

The start of the code for instruction fetch looks like this:

`line 24 Forvis_Spec.hs`

```
-- Instruction fetch
-- This function attempts an insruction fetch based on the current PC.

-- It first attempts to read 2 bytes only, in case the next
-- instruction is a 'C' (compressed) instruction. This may trap; if
-- not, we can decide if it's a C instruction, and read the next 2
-- bytes if it is not a C instruction; this, too, may trap.
```

```
data Fetch_Result = Fetch_Trap  Exc_Code
                  | Fetch_C     Word16
                  | Fetch       Word32
                  deriving (Show)

instr_fetch :: Machine_State -> (Fetch_Result, Machine_State)
instr_fetch  mstate =
```

The `instr_fetch` function takes the current machine state as argument, and attempts to read and instruction from memory, returning a 2-tuple: a `Fetch Result` and the updated machine state.

The `Fetch_Result` can indicate that there was a fault (such as a memory access fault or page fault) during the attempted memory-read, or that the instruction is a 16-bit instruction from the 'C' ISA extension, or that the instructionis a regular 32-bit instruction. In all cases, there could have been a change in the machine state, and hence it returns the updated machine state.

The code that follows the above excerpt first reads 2 bytes from memory, and checks if it encodes a possible 'C' instruction. If not, it then reads 2 more bytes from memory and returns it as a full 32-bit instruction. Of course, either of these two reads can fault, and this is the reason we read two bytes at a time: the first read may succeed with a 'C' instruction, in which case we do not want to encounter a fault for reading two more bytes which may be unnecessary in the program flow.

4.2 General structure of spec instruction-semantics functions

The spec is written as a collection of functions, generally one per major opcode (7 least-significant bits of an instruction). Each function has the following structure:

```
spec_OPCODE :: Machine_State -> Instr -> (Bool, Machine_State)
spec_OPCODE mstate  instr =
    -- Instr fields:  X-type
     ... extract instruction fields from standard X format ...

    -- Decode check
    is_legal =  ... check if legal OPCODE and fields ...

    -- Semantics
    mstate1 =  ... instruction-specific semantics based on  mstate ...

    mstate2 =  ... small family of ``finish'' functions
  in
    (is_legal, mstate2)
```

The function takes a machine state and an instruction as arguments, and returns a 2-tuple: a Boolean and a new machine state. The Boolean result indicates whether the instruction is a legal OPCODE instruction (it's 7 least-significant bits code for OPCODE, and other constraints on other fields are met, such as must-be-zero).

If the Boolean result is True, the second component of the 2-tuple result is the updated state due to execution of the instruction.

4.2.1 Example: the ADD instruction

The ADD instruction is handled by the following function:

line 546 Forvis_Spec.hs
```
spec_OP :: Machine_State -> Instr -> (Bool, Machine_State)
spec_OP    mstate       instr =
  let
    -- Instr fields: R-type
    (funct7, rs2, rs1, funct3, rd, opcode) = ifields_R_type  instr

    -- Decode check
    is_ADD   = ((funct3 == funct3_ADD)  && (funct7 == funct7_ADD))
    is_SUB   = ((funct3 == funct3_SUB)  && (funct7 == funct7_SUB))
```

...

line 564 Forvis_Spec.hs
```
    is_legal = ((opcode == opcode_OP)
                && (is_ADD
                    || is_SUB
```

It first extracts the fields of an R-type instruction. Then we define some booleans like `is_ADD` that check the sub-opcodes of the instruction. Then we compute `is_legal` to checks the opcode and all the sub-opcodes covered by this function.

The next few lines express the semantics of this family of instructions. We read the Rs1 and Rs2 registers:

line 577 Forvis_Spec.hs
```
    -- Semantics
    rs1_val = mstate_gpr_read  mstate  rs1
    rs2_val = mstate_gpr_read  mstate  rs2
```

and we compute the value to be stored in the Rd register:

line 585 Forvis_Spec.hs
```
    rd_val | is_ADD  = cvt_s64_to_u64  ((cvt_u64_to_s64  rs1_val) + (cvt_u64_to_s64  rs2_val))
           | is_SUB  = cvt_s64_to_u64  ((cvt_u64_to_s64  rs1_val) - (cvt_u64_to_s64  rs2_val))
           | is_SLT  = if ((cvt_u64_to_s64  rs1_val) < (cvt_u64_to_s64  rs2_val)) then 1 else 0
           | is_SLTU = if (rs1_val < rs2_val) then 1 else 0
```

Note that ADD, SUB and SLT treat the register values as signed values, whereas SLTU treats them as unsigned values. Finally, we invoke the "finish" function that writes a value to Rd and increments PC by 4 (and increments MINSTRET), see discussion in Sec. 4.3.

line 601 Forvis_Spec.hs
```
    mstate1 = finish_rd_and_pc_plus_4  mstate  rd  rd_val
  in
    (is_legal, mstate1)
```

4.3 Standard "finish" functions

All instructions "finish" in one of a few common ways, and these are captured as functions. For example: this function captures the common finish of all ALU instructions, which write a result value `rd_val` to the GPR `rd`, increment the PC by 4, and increment the MINSTRET (instructions retired) counter.

line 1428 Forvis_Spec.hs

```
finish_rd_and_pc_plus_4 :: Machine_State -> GPR_Addr -> Word64 -> Machine_State
finish_rd_and_pc_plus_4  mstate  rd  rd_val =
  let mstate1 = mstate_gpr_write  mstate  rd  rd_val
      pc      = mstate_pc_read    mstate1
      mstate2 = mstate_pc_write   mstate1  (pc + 4)
      mstate3 = incr_minstret     mstate2
  in
    mstate3
```

4.4 Interrupts

line 1674 Forvis_Spec.hs

```
take_interrupt_if_any :: Machine_State -> (Bool, Machine_State)
take_interrupt_if_any  mstate =
```

The `take_interrupt_if_any` function can be applied between any two instruction executions. It uses the function `fn_interrupt_pending` that examines MSTATUS, MIP, MIE and the current privilege level to check if there is an interrupt is pending and the hart is ready to handle it. If so, it applies `mstate_upd_on_trap` to update the machine state, which it returns along with True. Otherwise, it returns False and the unchanged machine state.

4.5 Sequential (one-instruction-at-a-time) interpretation

The sequential interpreter has a machine state `M` as described in Sec. 3, and a list *spec_fns* of spec functions as described in the previous section, i.e., each having the type:

```
Machine_State -> Instr -> (Bool, Machine_State)
```

The interpreter performs the following, forever:

> It uses the memory-access API function `mstate_mem_read` to read an instruction from `M`. It then applies each function from *spec_fns*, one by one until one of them returns `(True,M')`, i.e., one of them successfully decodes and executes the instruction.
>
> If all the functions in *spec_fns* return `(False,...)`, the interpreter applies the `finish_trap` function to `M` with the `ILLEGAL_INSTRUCTION` exception code to produce the next state `M'`.

5 Files GPR_File.hs (General Purpose Registers) and CSR_File.hs (Control and Status Registers)

`GPR_File.hs` implements a file of general-purpose registers, and the API functions `gpr_read` and `gpr_write`; it is simple enough that we do not discuss it further here.

CSR_File.hs implements a file of Control and Status registers, and the API functions csr_read and csr_write. The main subtlety here is that the distinct CSR addresses refer to "views" of the same underlying register with various restrictions:

- USTATUS and SSTATUS are restricted views of MSTATUS
- UIE and SIE are restricted views of MIE
- UIP and SIP are restricted views of MIP

The functions mstatus_stack_fields and mstatus_upd_stack_fields encapsulate reading and writing the "stack" in the MSTATUS register containing the "previous privilege", "previous interrupt enable" and "interrupt enable" fields. This stack is pushed on traps/interrupts, and popped on URET/SRET/MRET instructions.

The function fn_interrupt_pending was mentioned earlier in Sec. 4.4; it analyzes the MSTATUS, MIP, MIE and current privilege level to decide whether a machine/supervisor/user external/software/timer interrupt is pending, and if so, which one.

6 File Virtual_Mem.hs: Virtual Memory

Essentially all the code to support virtual memory is in the file Virtual_Mem.hs.

There are four broad classes of memory access: instruction fetch, loads, stores, and AMOs. The function fn_vm_is_active checks whether the effective address computed in each kind of memory access is a virtual memory address that needs to be translated to a physical memory address. It examines the current privilege level, and the value in the "mode" field of CSR SATP. It also takes into account that if MSTATUS.MPRV is set, then loads, stores and AMOs should be regarded as occurring at the privilege level MSTATUS.MPP instead of the current privilege.

line 46 Virtual_Mem.hs

```
fn_vm_is_active :: Machine_State -> Bool -> Bool
fn_vm_is_active  mstate  is_instr =
```

In file Forvis_Spec.hs, in the spec functions for the four classes of memory access (instr_fetch, spec_LOAD, spec_STORE and spec_AMO), the code first invokes fn_vm_is_active to check if virtual-to-physical address translation is required. If so, it then invokes the function vm_translate to perform the translation. This function can return a memory access fault or page fault, or a successful translation with a physical address. We use the Memory_Result type to return this range of results. The vm_translate function may also modify machine state ("access" and "dirty" bits in page tables, internal cache- and TLB-tracking state, etc.), and so the machine state is both an argument and a result of the function.

line 77 Virtual_Mem.hs

```
vm_translate :: Machine_State -> Bool  ->  Bool  -> Word64 -> (Mem_Result, Machine_State)
vm_translate    mstate           is_instr  is_read  va =
```

7 Other source code files

`Main.hs` is a driver program that just dispatches to one of two use-cases, `Main_RunProgram.hs` (free-running) or `Main_TandemVerifier.hs` (Tandem Verification).

`Main_RunProgram.hs` reads RISC-V binaries (ELF), initializes architecture state and memory, and calls `RunProgram` to run the loaded program, up to a specified maximum number of instructions.

`Run_Program.hs` contains the FETCH-EXECUTE loop, along with some heuristic stopping-conditions (maximum instruction count, detected self-loop, detected non-zero write into `tohost` memory location, etc.

`Main_TandemVerifier.hs` sets up Forvis to be a slave processs to a tandem verifier, receiving commands on stdin and sending responses on stdout. The commands allow a tandem verifier to initialize architecture state, execute 1 or more instructions, and query archtectural state. Responses include tandem verification packets which the verifier can use to check an implementation.

`Memory.hs` implements a memory model with read, write and AMO functions.

`MMIO.hs` implements the memory-mapped I/O system. Currently it is very primitive, merely recording console I/O in a String in the `Machine_State`.

`Mem_Ops.hs` defines instruction field values that specify the type and size of memory operations. These are duplicates of defs in `Forvis_Spec.hs` where they are in the specs of LOAD, STORE and AMO instructions. They are repeated here because this information is also needed in `Memory.hs`, `MMIO.hs` and other places.

`Bit_Manipulation.hs` contains utilities for bit manipulation, including sign- and zero-extension, truncation, conversion, etc. that are relevant for these semantics.

`Elf.hs` and `Read_Hex_File.hs` are not part of the semantics per se; the executable uses them to read ELF files and "Hex Memory" files, respectively.

References

[1] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach (6th Edition)*. Morgan Kaufman, 2017.

[2] D. E. Knuth. Literate Programming. *The Computer Journal. British Computer Society.*, 27(2):97–11, 1984.

[3] R. S. Nikhil and Arvind. *Implicit Parallel Programming in* pH. Morgan Kaufman, Inc., 2001.

[4] D. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface (RISC-V Edition)*. Morgan Kaufmann, 2017.

[5] D. Patterson and A. Waterman. *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, 2017.

[6] S. Peyton Jones (Editor). *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 5 2003. haskell.org.

[7] A. Waterman and K. Asanovic (eds.). The RISC-V Instruction Set Manual. Technical Report Document Version 2.2, The RISC-V Foundation (`riscv.org`), May 7 2017.

[8] A. Waterman and K. Asanovic (eds.). The RISC-V Instruction Set Manual. Technical Report Document Version 1.10, The RISC-V Foundation (`riscv.org`), May 7 2017.