Forvis: A Formal RISC-V ISA Specification

A Reading Guide

Rishiyur S. Nikhil
Bluespec, Inc.

Revision: January 20, 2019

***** DRAFT: this document is still being written *****

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 for 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.

A separate document describes how to build the spec as a simulator and have it execute RISC-V ELF binary. Another document describes how to extend this spec for new ISA extensions.

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: Unprivileged ISA*, Andrew Waterman and Krste Asanovic, Document Version 20181106-Base-Ratification, November 6, 2018. [7]
- *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*, Andrew Waterman and Krste Asanovic (eds.), Document Version 20181203-Base-Ratification, December 3, 2018. [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:
  - RISC-V Assembly Language programmers as a reference explaining the instructions they use.
  - 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)

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

- 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).
- 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 =` *value-expression* `::` *type*

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 printable Strings `"RV32"` and `"RV64"`, respectively.

line 23 Arch_Defs.hs

```
-- Future: add RV128

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

2.2 Key architectural types: instructions and registers

Througout the spec, we use Haskell's "unbounded integer" type (`Integer`) to represent values that are typically represented in bit vectors of fixed size in hardware. Unbounded integers are truly unbounded and have no limit such as the typical 32 bits or 64 bits found in most programming languages. Unbounded integers never overflow. In the spec, we take care of 32-bit and 64-bit overflow explicitly.

Below, we define Haskell "type synonyms" as more readable synonyms for Haskell's `Integer` type.

line 31 Arch_Defs.hs

```
-- These are just synonyms of 'Integer', for readability

type Instr_32b = Integer
type Instr_16b = Integer

type InstrField = Integer      -- various fields of instructions

-- General-purpose registers

type GPR_Addr = Integer        -- 5-bit addrs, 0..31
type GPR_Val  = Integer        -- 32-bit or 64-bit values

-- Floating-point registers

type FPR_Addr = Integer        -- 5-bit addrs, 0..31

-- CSRs

type CSR_Addr = Integer        -- 12-bit addrs, 0..0xFFF
```

Next, we have Haskell function that decides whether a particular instruction is a 32-bit instruction or a 16-bit (compressed) instruction, by testing its two least-significant bits.

line 53 Arch_Defs.hs

```
-- instruction or not ('C' instrs have 2 lsbs not equal to 2'b11)

is_instr_C :: Integer -> Bool
is_instr_C  u16 = ((u16 .&. 0x3) /= 0x3)

{-# INLINE is_instr_C #-}
```

Here, and everywhere in the spec, you can safely ignore the `INLINE` annotation. These are "pragmas" or "directives" to the Haskell compiler when we compiler this spec into a simulator, and are purely meant to improve the performance (speed) of the simulator. In accordance with the their unimportance to the semantics, we always write these `INLINE` annotations *below* the corresponding function.

2.3 Major Opcodes for 32-bit instructions

The 7 least-significant bits of a 32-bit instruction constitute its "major opcode". This section defines them for the base "I" instruction set.

line 62 Arch_Defs.hs

```
-- Major opcodes

-- ----------------
-- 'I' (Base instruction set)

opcode_LUI       = 0x37  :: InstrField    -- 7'b_01_101_11
opcode_AUIPC     = 0x17  :: InstrField    -- 7'b_00_101_11
opcode_JAL       = 0x6F  :: InstrField    -- 7'b_11_011_11
opcode_JALR      = 0x67  :: InstrField    -- 7'b_11_001_11
...more...
```

Later in the file, we see major-opcode definitions for the "A" (Atomics) extension[3], the "F" and "D" extensions (single-precision and double floating point).

Most instructions also have other fields that further refine the opcode; we call them "sub-opcodes". These are generally defined in the separate modules for each extention (for example, in a section labelled "Sub-opcodes for 'I' instructions" in file `Forvis_Spec_I.hs`) because they are only used locally there.

However, some sub-opcodes are used in multiple modules and are therefore defined in this file (`Arch_Defs.hs`). This is true of all the memory-operation sub-opcodes such as `funct3_LB`, `funct3_SB`, `msbs5_AMO_ADD`, etc., as well as `funct3_PRIV`.

2.4 Exception Codes

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

line 148 Arch_Defs.hs

```
type Exc_Code = Integer

-- Interrupt exception codes
exc_code_u_software_interrupt      =  0 :: Exc_Code
exc_code_s_software_interrupt      =  1 :: Exc_Code
...more...
```

... and for traps:

line 161 Arch_Defs.hs

```
-- Trap exception codes
exc_code_instr_addr_misaligned     =  0 :: Exc_Code
exc_code_instr_access_fault        =  1 :: Exc_Code
...more...
```

[3]I am grateful to my colleague Joe Stoy for introducing me to the etymology of the word "atomic". It can be read as "a+tom+ic". The "tom" (as in "tomography") means "cut", and the "a" negates it (→ "uncuttable").

2.5 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 210 Arch_Defs.hs

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

data Mem_Result = Mem_Result_Ok   Integer
                | Mem_Result_Err  Exc_Code
```

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*

2.6 Privilege Levels

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

line 218 Arch_Defs.hs

```
-- Machine, Supervisor and User

type Priv_Level = InstrField

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

3 File GPR_File.hs (General Purpose Registers)

`GPR_File.hs` implements a file of general-purpose registers.

We represent it using Haskell's `Data_Map.Map` type, which is an associative map (like a Python "dictionary") that associates register names with values). This representation choice is purely internal to this module because in the export list in the module header at the top of the file:

line 4 GPR_File.hs

```
module GPR_File (GPR_File,
                 mkGPR_File,
                 gpr_read,
                 gpr_write,
                 print_GPR_File) where
```

we mention the type `GPR_File` without exporting its internal representation. If, instead, we had said "`GPR_File(..)`", we'd expose its internal detail. Thus, we can freely change the representation to something else (and change the API functions accordingnly) without affecting any of the rest of the modules. In other words, `GPR_File` is an *abstract type* for the rest of the modules. Here is the representation and constructor:

line 37 GPR_File.hs

```
newtype GPR_File = GPR_File  (Data_Map.Map  InstrField  Integer)

mkGPR_File :: GPR_File
mkGPR_File = GPR_File (Data_Map.fromList (zip
                                          [0..31]
                                          (repeat (fromIntegral 0))))
```

The `zip` function constructs a listof intial values, associating each register name 0..31 with 0 (randomly chosen, since the spec does not specify the initial value of any register).

This is followed by the API functions `gpr_read` and `gpr_write`. The latter always writes 0 into GPR 0, so we can only ever read 0 from GPR 0.[4]

4 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.]

4.1 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 *unbounded* integers (Haskell's "`Integer`" type). 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 32-bit arithmetic.

4.2 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, floating-point regsiters, 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.

[4] In "`seq val1 (..)`" in the last line of `gpr_write`, only the part in parentheses is relevant, doing the actual GPR register file update; the rest is a wrapper that is merely a Haskell performance optimization for the simulator, concerned with Haskell's lazy evaluation regime.

Finally, we have fields that are not semantically relevant, but are needed or useful in simulation or formal reasoning, gathering statistics, etc., including a list of legal address ranges (memory load/store instructions should trap if accessing anything outside this range).

line 38 Machine_State.hs

```
data Machine_State =
  Machine_State { -- Architectural state
                  f_pc   :: Integer,
                  f_gprs :: GPR_File,
                  f_fprs :: FPR_File,
                  f_csrs :: CSR_File,
                  f_priv :: Priv_Level,

                  -- Memory and mory mapped IO
                  f_mem  :: Mem,
                  f_mmio :: MMIO,

                  -- Implementation options

                  -- Legal memory addresses: list of (addr_start, addr_lim)
                  f_mem_addr_ranges :: [(Integer, Integer)],

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

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.[5]

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

line 90 Machine_State.hs

```
-- Make a Machine_State, given initial PC and memory contents
mkMachine_State :: RV ->                      -- Initial RV32/RV64
                   Integer ->                 -- Initial value of misa
                   Integer ->                 -- Initial value of PC
                   [(Integer,Integer)] ->     -- List of legal memory addresses
                   ([(Integer, Integer)]) ->  -- Initial mem contents (addr-&-byte list)
                   Machine_State              -- result
mkMachine_State  rv  misa  initial_PC  addr_ranges  addr_byte_list =
  let
    mstate = Machine_State {f_pc   = initial_PC,
                            f_gprs = mkGPR_File,
                            f_fprs = mkFPR_File,
                            f_csrs = mkCSR_File  rv  misa,
                            f_priv = m_Priv_Level,

                            f_mem             = mkMem  addr_byte_list,
                            f_mmio            = mkMMIO,
```

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

```
                              f_mem_addr_ranges = addr_ranges,

                              f_rv                 = rv,
                              f_run_state          = Run_State_Running,
                              f_last_instr_trapped = False,
                              f_verbosity          = 0
                             }
  in
    mstate
```

All functions that "update" the machine state are written in purely functional style: the first argument is typically a machine state, and the final result is the new machine state. This will be evident in their type signatures:

somefunction `::  Machine_State ->` *...other arguments...* `-> Machine_State`

What follows is a series of API functions to read or update the machine state, such as the following to access and update the PC:

line 120 Machine_State.hs

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

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

The `mstate_pc_write` function just applies the `f_pc` field selector to the machine state to extract that field. The `mstate_pc_write` function uses Haskell's "field update" notation:

```
mstate { f_pc = val }
```

to construct (and return) a new machine state in which the `f_pc` field has the new value.

In many of the API functions, such as those to read and write GPRs, FPRs or CSRs, the function merely invokes the appropriate API of the corresponding component (GPR file, FPR file or CSR file).

In the API functions for read, write and atomic memory operations, such as `mstate_mem_read`, we check if the given address is a supported memory address and return an exception if not. Otherwise, we triage the address to determine if it is for actual memory or for a memory-mapped I/O device, and direct the request to the appropriate component.

The API functions for FENCE, FENCE.I and SFENCE.VMA are currenty no-ops in the spec since they only come into play when there is concurrency involving multiple paths to memory from one or more harts (hardware threads). The current specs is interpreted in a completely sequential, one-instruction-at-a-time manner; there is only one path to memory involving instructions and data; and there is only one hart.

The file ends with a number of functions to aid in simulation, to move console input and output between the machine state and the console, to "tick" IO devices (which logically run concurrently with the CPU), etc.

5 File Forvis_Spec_Common.hs: Common functions used by all the spec functions

All instructions "finish" in one of a few common ways, and these are captured as functions in this module. 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 or 2, depending on boolean `is_C`, which indicates whether the current instruction is a regular 32-bit insruction or a 16-bit C (compressed) instruction;
- and increment the MINSTRET (instructions retired) counter.

line 47 Forvis_Spec_Common.hs

```
finish_rd_and_pc_incr :: Machine_State -> GPR_Addr -> Integer -> Bool -> Machine_State
finish_rd_and_pc_incr    mstate           rd          rd_val     is_C =
  let mstate1 = mstate_gpr_write  mstate  rd  rd_val
      pc      = mstate_pc_read    mstate1
      delta   = if is_C then 2 else 4
      mstate2 = mstate_pc_write   mstate1  (pc + delta)
      mstate3 = incr_minstret     mstate2
  in
    mstate3
```

6 Instruction fetch

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

line 44 Forvis_Spec.hs

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

-- We do not blindly fetch 4 bytes, since the fetch of the latter 2
-- bytes may trap, which may not be relevant if the first 2 bytes are
-- a 'C' (compressed) instruction (which may trap or jump elsewhere).

-- So:
--   - We first attempt to read 2 bytes only
--   - This fetch-attempt may trap
--   - Else we check if it's a 'C' instruction; if so, we're done
--   - Else we attempt to read the next 2 bytes (remaining 2 bytes of a 32b instr)
--   -      This fetch-attempt may also trap

data Fetch_Result = Fetch_Trap  Exc_Code
                  | Fetch_C     Integer
                  | Fetch       Integer
                  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 body of the `instr_fetch` function first checks the 'C' flag in CSR MISA to see if compressed instructions are supported. If not, it reads a 4-byte (32-bit) instruction from memory. If 'C' is supported, it 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.

7 Execution specification

The execution spec is modularized in the same way as the ISA itself:

Haskell module (append `.hs` for filename)	Description
`Forvis_Spec_I`	Base Integer Instruction Set, R32 and RV64
`Forvis_Spec_I64`	Base Integer Instruction Set, RV64 only
`Forvis_Spec_M`	Integer Multiply/Divide instructions
`Forvis_Spec_A`	Atomic ops
`Forvis_Spec_C`	Compressed instructions
`Forvis_Spec_F`	Single-precision Floating Point instructions
`Forvis_Spec_D`	Double-precision Floating Point instructions
`Forvis_Spec_Priv`	Privileged Architecture instructions
`Forvis_Spec_Zicsr`	CSR instructions
`Forvis_Spec_Zifencei`	FENCE.I instruction
`Forvis_Spec_Common`	Shared functions for above

The module for each group X follows a similar plan:

- Declaration of a type `data Instr_X`, a data structure for the logical fields of instructions in the group X (a Haskell "algebraic data type"). These are like "abstract syntax trees" for instructions in the group.
- Definitions of sub-opcodes for instructions in group X. These are values of fields in the 32-bit instruction that collectively specify a particular instruction opcode.
- A decode function `decode_X` of type:
  ```
  decode_X :: RV -> Instr_32b -> Maybe Instr_X
  ```
  that takes a a 32-bit instruction and returns a result that is either:
  - `Nothing`: this is not an instruction in this group
  - `Just adt`: this is an instruction in this group, and `adt` is a value of type `Instr_X`, i.e., the logical view of the instruction.

The RV argument is because these functions serve for both the RV32 and RV64 base integer instructions, but some instructions are only valid in RV64.

- An execution-dispatch function exec_instr_X that dispatches each kind of instruction in the group to a specific execution function for that particular kind of instruction.
- A series of functions exec_OP1, exec_OP2, ..., one per opcode, describing the semantics of that kind of instruction.

Since all the modules follow this plan, we will not describe them all in detail here.

7.1 File Forvis_Spec_I: Spec for Integer Base Instruction Set

7.1.1 Algebraic Data Type for I instructions

The file begins with a Haskell data type declaration for the type Instr_I

line 30 Forvis_Spec_I.hs

```
data Instr_I = LUI    GPR_Addr  InstrField              -- rd,  imm20
             | AUIPC  GPR_Addr  InstrField              -- rd,  imm20

             | JAL    GPR_Addr  InstrField              -- rd,  imm21
             | JALR   GPR_Addr  GPR_Addr  InstrField    -- rd,  rs1, imm12
...more...
```

This should be read as follows: "A value of type Instr_I is

- *either* a LUI instruction, in which case it has two fields of type GPR_Addr and InstrField (the destination register rd and an immediate value),
- *or* a AUIPC instruction, in which case it has two fields of type GPR_Addr and InstrField (the destination register rd and an immediate value),
- *or* a JAL instruction, in which case it has two fields of type GPR_Addr and InstrField (the destination register rd and an immediate value),
- *or* a JALR instruction, in which case it has three fields of type GPR_Addr, GPR_Addr and InstrField (the destination register rd, the source register rs1, and an immediate value),
- ... and so on."

7.1.2 Sub-opcodes for I instructions

The next section defines values of other fields in a 32-bit instruction that further refine the group opcode into a specific opcode:

line 84 Forvis_Spec_I.hs

```
-- opcode_JALR sub-opcodes
funct3_JALR      = 0x0   :: InstrField    -- 3'b_000

-- opcode_BRANCH sub-opcodes
funct3_BEQ       = 0x0  :: InstrField     -- 3'b_000
funct3_BNE       = 0x1  :: InstrField     -- 3'b_001
...more...
```

These are values in the 3-bit field in bits [14:12] of a 32-bit instruction.

7.1.3 Decoder for I instructions

The next section defines the function `decode_I` whose arguments are `rv` (because some I instructions are only valid in RV64 and not in RV32) and a 32-bit instruction. The result is of type `Maybe Instr_I`, i.e., it is:

- *either* `Nothing`: this is not an I instruction,
- *or* `Just instr_I`: this is an I instruction, and the field `instr_I` is value of type `Instr_I`, the logical view of the instruction.

line 148 Forvis_Spec_I.hs

```
decode_I :: RV -> Instr_32b -> Maybe Instr_I
decode_I    rv    instr_32b =
  let
    -- Symbolic names for notable bitfields in the 32b instruction 'instr_32b'
    opcode  = bitSlice  instr_32b   6   0
    rd      = bitSlice  instr_32b  11   7
    funct3  = bitSlice  instr_32b  14  12
    rs1     = bitSlice  instr_32b  19  15
    rs2     = bitSlice  instr_32b  24  20
    funct7  = bitSlice  instr_32b  31  25
...more...
```

The first few lines of the function use `bitSlice` to extract bit-fields of the instruction. This is a help-function defined in `Bit_Utils.hs`, and `bitSlice x [`j`,`k`]` is equivalent to the Verilog/SystemVerilog bit-selection `x[`j`,`k`]`. Note that some field-extractions can involve some complex bit-shuffling, such as:

line 168 Forvis_Spec_I.hs

```
    imm21_J = ((    shiftL  (bitSlice  instr_32b  31  31)  20)
               .|. (shiftL  (bitSlice  instr_32b  30  21)   1)
               .|. (shiftL  (bitSlice  instr_32b  20  20)  11)
               .|. (shiftL  (bitSlice  instr_32b  19  12)  12))
```

The function finally defines the result `m_instr_I` (of type `Maybe Instr_I`) by dispatching on a series of conditions, each checking for a particular opcode:

line 198 Forvis_Spec_I.hs

```
    m_instr_I
      | opcode==opcode_LUI   = Just  (LUI    rd  imm20_U)
      | opcode==opcode_AUIPC = Just  (AUIPC  rd  imm20_U)

      | opcode==opcode_JAL                        = Just  (JAL   rd  imm21_J)
      | opcode==opcode_JALR, funct3==funct3_JALR = Just  (JALR  rd  rs1  imm12_I)
...more...
```

If none of the conditions match, it returns `Nothing`

7.1.4 Dispatcher for I instructions

The next function is simply a dispatcher that takes an `Instr_I` value and, based on the kind of I instruction it is, dispatches to a specific execution function for that kind of of instruction.

line 257 Forvis_Spec_I.hs
```
type Spec_Instr_I = Bool -> Instr_I -> Machine_State -> Machine_State
--                  is_C    instr_I    mstate           mstate'

exec_instr_I :: Spec_Instr_I
exec_instr_I  is_C  instr_I  mstate =
  case instr_I of
    LUI    rd   imm20      -> exec_LUI    is_C  instr_I  mstate
    AUIPC  rd   imm20      -> exec_AUIPC  is_C  instr_I  mstate

    JAL    rd   imm21      -> exec_JAL    is_C  instr_I  mstate
    JALR   rd   rs1  imm12 -> exec_JALR   is_C  instr_I  mstate
...more...
```

It uses the Haskell pattern-matching `case` statement to determine which kind of instruction it is, and invokes the appropriate function. Note, it has no check for illegal instructions; the fact that the argument is of type `Instr_I` means it can only be a valid I instruction.

Note that we've defined the type `Spec_Instr_I` to be a type-synonym for:
`Bool -> Instr_I -> Machine_State -> Machine_State`
because all our spec functions will have this common type.

The first argument, `is_C` is a boolean indicating whether the current instruction is a regular 32-bit instruction or a 16-bit (C, compressed) instruction. In RISC-V, each 16-bit instruction is defined as a "short form" for a specific corresponding 32-bit instruction. Thus, we define the semantics in one function, but use the parameter `is_C` to remember whether we're doing this for a 32-bit or 16-bit instruction. For almost all instructions, we either update the PC with the address of the next instruction, or we remember address of the next instruction (e.g., in JAL and JALR). This next-instruction-address may be the current PC +2 or +4, depending on `is_C`.

7.1.5 Execution semantics for each I instruction

There is one `exec_`*FOO* function for each opcode *FOO*. We examine excerpts of a few of them, for illustration.

The first I instruction, LUI, is very simple:

line 315 Forvis_Spec_I.hs
```
exec_LUI :: Spec_Instr_I
exec_LUI  is_C  (LUI  rd  imm20)  mstate =
  let
    xlen    = mstate_xlen_read  mstate
    rd_val  = sign_extend  32  xlen  (shiftL  imm20  12)
    mstate1 = finish_rd_and_pc_incr  mstate  rd  rd_val  is_C
  in
    mstate1
```

It uses the 20-bit immedate to calculate a value `rd_val` to save in the destination register `rd`, and calls a standard "finish" function to write the destination register and increment the PC.

The `JALR` instruction is a bit more complex:

line 369 Forvis_Spec_I.hs

```
exec_JALR :: Spec_Instr_I
exec_JALR  is_C  (JALR  rd  rs1  imm12)  mstate =
  let
    misa   = mstate_csr_read   mstate  csr_addr_misa
    xlen   = mstate_xlen_read  mstate
    pc     = mstate_pc_read    mstate
    rd_val = if is_C then pc + 2 else pc + 4

    s_offset = sign_extend  12  xlen  imm12

    rs1_val = mstate_gpr_read  mstate  rs1

    new_pc  = alu_add  xlen  rs1_val  s_offset
    new_pc' = clearBit  new_pc  0
    aligned  = if (misa_flag  misa  'C') then
                 True
               else
                 ((new_pc' .&. 0x3) == 0)

    mstate1 = if aligned then
                finish_rd_and_pc  mstate  rd  rd_val  new_pc'
              else
                finish_trap  mstate  exc_code_instr_addr_misaligned  new_pc'
  in
    mstate1
```

We see that that saved "return-address" (`rd_val` is calculated as PC+2 or PC+4 depending on `is_C`. The jump-target PC is initially calculated by adding the offset to the `rs1_val`. Then, per the spec, we clear its bit [0] to 0. Then we check if is is properly aligned, which decision itself depends on whether MISA.C is currently active or not. Finally, if the target is properly aligned, we finish normally (updating rd and incrementing PC), otherwise we finish by trapping with exception code `exc_code_instr_addr_misaligned`, and storing `new_pc'` in the TVAL CSR.

The functions for memory load instructions `exec_LB`, `exec_LH`, `exec_LW`, `exec_LD` all funnel back through a common `exec_LOAD` help-function. Let's focus on this excerpt:

line 486 Forvis_Spec_I.hs

```
    -- Read mem, possibly with virtual mem translation
    is_instr = False
    (result1, mstate1) = mstate_vm_read  mstate
                                         is_instr
                                         exc_code_load_access_fault
                                         funct3
                                         eaddr2
```

Its work is done by `mstate_vm_read`, which is in file `Virtual_Mem.hs`, which does all memory reads. By examining `mstate`, it can decide whether the address is a virtual or physical address, and do the translation if needed. During translation or subsequent memory access, it may encounter a fault. The final result is of type `Mem_Result` (described in Sec. 2.5), indicating that it's ok (and contains the appropriate payload, or an error (and contains the appropriate exception code).

Moving further down, the ADD instruction is handled by the following function:

line 592 Forvis_Spec_I.hs

```
exec_ADD  :: Spec_Instr_I
exec_ADD   is_C  (ADD    rd  rs1  rs2)  mstate =
  exec_OP  alu_add   is_C  rd  rs1  rs2  mstate
```

This just dispatches to the function `exec_OP` which is used by all the `opcode_OP` functions. To this common function, we pass the function `alu_add` (defined in file `ALU.hs`) to indicate the specific function to be performed on the operands.

The `exec_OP` function is simple. We read the Rs1 and Rs2 registers, perform the specified `alu_op`, and finish by updating the destination register `rd` and incrementing the PC (by 4 or 2, respectively, depending boolean `is_C`, which indicates whether the current instruction is a regular 32-bit instruction or a 16-bit C (compressed) instruction. The `finish` function will also increment MINSTRET (see discussion in Sec. 5).

line 634 Forvis_Spec_I.hs

```
exec_OP :: (Int -> Integer -> Integer -> Integer) ->
           Bool ->
           GPR_Addr ->
           GPR_Addr ->
           GPR_Addr ->
           Machine_State -> Machine_State
exec_OP    alu_op  is_C  rd  rs1  rs2  mstate =
  let
    xlen    = mstate_xlen_read  mstate
    rs1_val = mstate_gpr_read   mstate  rs1        -- read rs1
    rs2_val = mstate_gpr_read   mstate  rs2        -- read rs2
    rd_val  = alu_op  xlen  rs1_val  rs2_val       -- compute rd_val
    mstate1 = finish_rd_and_pc_incr  mstate  rd  rd_val  is_C
  in
```

7.2 Files ALU.hs and FPU.hs

The module `ALU` represents the "pure integer ALU", i.e., pure functions of one or two inputs representing the various interger arithmetic, logic and comparison operatinos of interest. Most of the functions are quite straightforward, invoking Haskell primitives to perform the actual operations.

All considerations of whether we are dealing with 32-bit or 64-bit input and output values, whether they are to be interpreted as signed or unsigned values, etc., are confined to this module. Outside this module, all values are "stored" as Haskell's `Integer` data type.

Analogously, the module `FPU` represents the "pure floating point ALU", encapsulating all floating-point arithmetic, logic, comparision and conversion operations. All considerations of single-precision vs. double-precision, conversions between these and integers etc., are confined to this module.

7.3 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 47 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 155 Virtual_Mem.hs

```
-- vm_translate    translates a virtual address into a physical address.
-- Notes:
--   - 'is_instr' is True if this is for an instruction-fetch as opposed to LOAD/STORE
--   - 'is_read'  is True for LOAD, False for STORE/AMO
--   - 1st component of tuple result is 'Mem_Result_Err exc_code' if there was a trap
--   -     and 'Mem_Result_Ok pa' if it successfully translated to a phys addr
--   - 2nd component of tuple result is new mem state,  potentially modified
--         (page table A D bits, cache tracking, TLB tracking, ...)

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

7.4 CSR_File.hs (Control and Status Registers)

`CSR_File.hs` implements a file of Control and Status registers. The module begins with definitions of reset values for CSRs at the User, Supervisor and Machine levels of privilege.

7.4.1 CSR addresses

The next few sections define the addresses of all the standard CSRs (Control and Status Registers), at User, Supervisor and Machine Privilege levels.

line 223 CSR_File.hs

```
csr_addr_ustatus    = 0x000 :: CSR_Addr
csr_addr_uie        = 0x004 :: CSR_Addr
csr_addr_utvec      = 0x005 :: CSR_Addr

csr_addr_uscratch   = 0x040 :: CSR_Addr
...more...
```

7.4.2 The MISA CSR

A key CSR is MISA ("Machine ISA Register"). The 2 most-significant bits are called `MXL` and it encodes the current "native" width of the ISA (width of PC and GPRs), which can be 32, 64 or 128 bits.

line 483 CSR_File.hs

```
-- Codes for MXL, SXL, UXL
xl_rv32  = 1 :: Integer
xl_rv64  = 2 :: Integer
xl_rv128 = 3 :: Integer
```

The lower 26 bits are named by letters of the alphabet, A-Z, with bit 0 being A and Bit 25 begin Z. The function `misa_flag`, when given the value in the MISA register and a letter of the alphabet (uppercase or lowercase), returns a boolean indicating whether the corresponding bit is set or not.

line 489 CSR_File.hs

```
-- Test whether a particular MISA 'letter' bit (A-Z) is set

misa_flag :: Integer -> Char -> Bool
misa_flag    misa       letter =
  if (     isAsciiUpper  letter) then
    (((shiftR  misa  ((ord letter) - (ord 'A'))) .&. 1) == 1)
  else if (isAsciiLower  letter) then
    (((shiftR  misa  ((ord letter) - (ord 'a'))) .&. 1) == 1)
  else
    error "Illegal argument to misa_flag"
```

This is followed by symbolic-name definitions for the integer bit positions of the 26 alphabets and the MXL field.

line 503 CSR_File.hs

```
misa_A_bitpos = 0 :: Int
misa_B_bitpos = 1 :: Int
misa_C_bitpos = 2 :: Int
...more...
```

7.4.3 The MSTATUS CSR

Another key CSR is MSTATUS ("Machine Mode Status"). The code starts with definitions for the integer bit positions of its fields.

This is followed by two help-functions that are used in defining the semantics of exceptions (interrupts and traps) and returns-from-exceptions. The least-significant 8 fields of MSTATUS represent a shallow "stack" of interrupt-enable and privilege bits. On an exception, we push new values on to this stack, and when we return-from-exception we pop the stack. The function `mstatus_stack_fields` extracts the stack, returning the fields as an 8-tuple: `(mpp,spp,mpie,spie,upie,mie,sie,uie)`. The inverse function, `mstatus_upd_stack_fields` takes an MSTATUS value and an 8-tuple of new stack values, and returns a new MSTATUS with the stack updated.

Not all fields in MSTATUS are used (they may be defined in future versions of the spec). The next few definitions describe "masks" that restrict an MSTATUS value to defined fields so that we do not disturb the undefined fields. Some of the fields of MSTATUS are visible as the SSTATUS (Supervisor Status) and USTATUS (User Status) at lower privilege levels. Masks for these views are also defined here.

The API functions `csr_read` and `csr_write`. The main subtlety here is that the certain 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. 7.5; 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.

7.5 Interrupts

line 244 Forvis_Spec.hs

```
mstate_take_interrupt_if_any :: Machine_State -> (Maybe Exc_Code, Machine_State)
mstate_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.

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

The sequential interpreter has a machine state `M` as described in Sec. 4, 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'`.

9 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).

10 Other source code files

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

Most of the remaining files are not part of ISA semantics, but infrastructure for building a "system": a boot ROM, a memory, and I/O devices such as a timer (MTIME and MTIMECMP), a software-interrupt location (MSIP), and a UART for console I/O.

`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.

`Addres_Map.hs` specifies the address map for the "system": the base address and address range for each memory and I/O device.

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

`MMIO.hs` implements the memory-mapped I/O system.

`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.

`UART.hs` is a model of the popular National Semiconductor NS16550A UART.

`Elf.hs` and `Read_Hex_File.hs` are functions for reading 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. The RISC-V Instruction Set Manual. Technical Report Document Version 20181106-Base-Ratification, The RISC-V Foundation (`riscv.org`), November 6 2018.

[8] A. Waterman and K. Asanovic. The RISC-V Instruction Set Manual. Technical Report Document Version 20181203-Base-Ratification, The RISC-V Foundation (`riscv.org`), December 3 2018.