Forvis: A Formal RISC-V ISA Specification

A Reading Guide

Rishiyur S. Nikhil
Bluespec, Inc.

Revision: January 29, 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)
RZtso	RISC-V Optional TSO Weak 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 [4]
- "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 [3]

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

Thanks ...

- to the original creators of RISC-V for making all this possible in the first place.
- to Bluespec, Inc. for supporting this work.
- to the RISC-V Foundation for recognizing the importance of formal specs and constituting the ISA Formal Specification Technical Group.
- to the members of the RISC-V Foundation's ISA Formal Specification Technical Group with whom we have wonderful weekly discussions on this topic.

Contents

This page is intentionally blank.

“This is not a pipe”. René Magritte, 1929.

https://en.wikipedia.org/wiki/The_Treachery_of_Images[1]

[1] Image from Wikipedia and used with same “fair use” rationale: https://en.wikipedia.org/wiki/File:MagrittePipe.jpg

1 Introduction

Forvis is a formal specification of the RISC-V Instruction Set Architecture (ISA), and is written in an "extremely elementary" subset of the functional programming language Haskell. The spec is located in the `src/` directory of this GitHub repository:
`https://github.com/rsnikhil/Forvis_RISCV-ISA-Spec`

This document is merely a set of "temporary training wheels" to help read the spec (Haskell source code files), for those new to Forvis, Haskell, ISAs, or RISC-V. For many people (especially those familiar with Haskell), this document may be unnecessary; just viewing Fig. 1 to see how the files are organized may be enough preparation to jump into the code. The code has copius comments and can be read on its own. In any case, this document is just an assist—it should only be read side-by-side with the code opened in your favorite code browser.[2]

Fig. 1 shows an overview of many of the source files.[3]

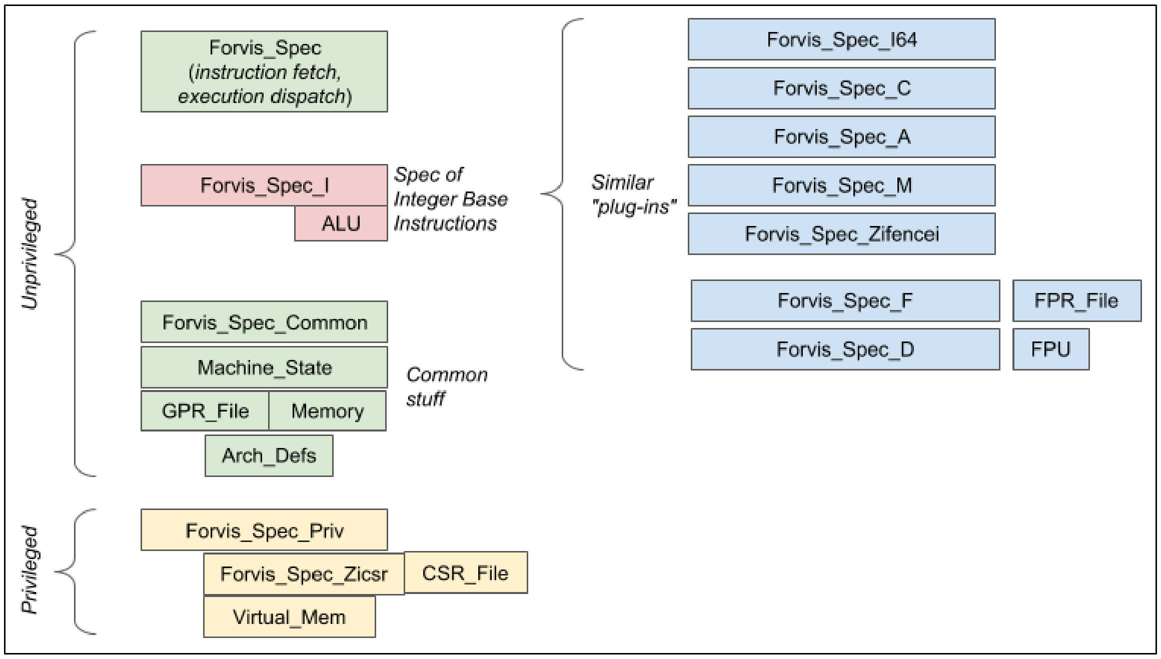

Figure 1: Overview of Forvis files/modules

- The modules shown in green are common resources used across all instructions.
- `Forvis_Spec_I`, shown in red, is the base integer instruction set. The modules shown in blue are for the other extensions. All of these files follow exactly the same internal organization so understanding `Forvis_Spec_I` should help in understanding them.
- The modules shown in yellow are the Privileged Architecture.

Please skip to Sec.2 to dive into the technical stuff. The rest of Sec. 1 contains optional detail and background context.

[2] All code fragments in this document are in fact automatically extracted from the actual spec code.
[3] For every Haskell module `Foo` in the figure, you'll find a file `src/Foo.hs` in the repository.

1.1 What is Forvis?

Forvis cover all these standard RISC-V features and extensions:

RV32I and RV64I	Base integer instruction sets for RV32 and RV64
Extension C	Compressed (16-bit) instructions
Extension A	Atomic memory operations
Extension M	Integer multiply and divide instructions
Extension F	Single-precision Floating Point instructions
Extension D	Double-precision Floating Point instructions
Extension Zicsr	CSR (Control and Status Registers) read and write instructions
Extension Zifencei	FENCE.I instruction
Privileged Architecture	Machine, Supervisor and User privilege levels Sv32, Sv39 and Sv48 Virtual Memory schemes Exceptions (Traps and Interrupts)

The Forvis spec is written in the pure functional programming language Haskell [5] (cf. `haskell.org`) because of its clean and precise mathematical semantics.

We have deliberately restricted ourselves to an extremely elementary subset of Haskell so that, along with this reading guide, it is easily accessible to people who may be totally unfamiliar with Haskell and who may have no interest in learning Haskell. Hopefully, familiarity with types and functions in almost any other programming language should be adequate background. In any case, we have also included a small "cheat sheet" in Appendix A for the Haskell subset we use.

Using extremely elementary Haskell will also make it easier for authors of new ISA extensions to extend Forvis to cover their ISA extensions, even if they are unfamiliar with Haskell. Appendix B provides some guidance on how to extend Forvis.

Using extremely elementary Haskell will also make it easy to parse and connect to other tools, such as proof assistants, theorem provers, and so on. A parser for Forvis is under development to facilitate this.

1.2 Sequential and Concurrent Interpretation as a RISC-V simulator

Forvis can be directly compiled in Haskell and executed as a RISC-V simulator where it, in turn, executes RISC-V machine code produced by hand or by RISC-V assemblers and compilers (the README in the GitHub repository has directions for how to do this). This models ***sequential***, one-instruction-at-a-time, single-hart semantics, and is adequate for many (if not most) use-cases.

In contrast, a ***concurrent*** interpretation is required to model some implementations with advanced micro-architectural features such as pipelining, separate instruction and data channels to memory, caches, store buffers, TLBs in MMUs, VLIW and super-scalarity, out-of-order execution, multi-harts cores and multi-cores. One may at first imagine these to be implementation details that would/should be invisible in an abstract semantics, but in fact their effects are *deliberately* made visible[4] in user programs in certain controlled ways, resulting in spec features such as "fence" instructions and weak memory models.

[4] For hardware performance reasons.

The Forvis source code, instead of being run through Haskell, can also be run through an aternative *concurrent* interpreter. The basic spec source codes (all the files of form `Forvis_Spec_`X`.hs`) remain untouched and are used as-is. What changes is the infrastructure framework within which that code is interpreted:

- register files change to allow concurrent, out-of-order access;
- the memory model changes to allow concurrent, out-of-order access;
- the outer fetch-and-execute loop changes to allow concurrent fetches and executions; etc.

At time of writing, this alternate concurrent interpreter for the same spec source code is still in development. It is based on ideas from "implicit dataflow" languages and interpretation (cf. "Implicit Parallel Programming in *pH*"[2]).

1.3 Optional background context: Forvis goals

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`). The original English-language specs for RISC-V are:

- *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*, Andrew Waterman and Krste Asanovic, Document Version 20181106-Base-Ratification, November 6, 2018. [6]

- *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*, Andrew Waterman and Krste Asanovic (eds.), Document Version 20181203-Base-Ratification, December 3, 2018. [7]:

Forvis is a formal specification of those specs, 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: RV32I (32-bit integers) and RV64I (64-bit integers) (an RV128I 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).
  - 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.

2 Common Stuff (shared by the various instruction specs)

Our tour will go somewhat bottom-up with respect to Fig. 1, first covering some "common stuff" used by all instructions, then the base integer instruction set, then the top-level instruction fetch and execute semantics.

Reminder: please only read this document side-by-side with the actual code.

2.1 File `Arch_Defs.hs`: basic architectural definitions

2.1.1 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)
```

2.1.2 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 hardware as bit vectors of fixed size. 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 this spec, we take care of 32-bit and 64-bit overflow explicitly (inside module `ALU`).

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
...more...
```

This Haskell function 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 sequential simulator, and are purely meant to improve the performance (speed) of the simulator. In accordance with the their semantic unimportance, we write these `INLINE` annotations *below* the corresponding function.

2.1.3 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[5], 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 because they are only used locally there (for example, in a section labelled "Sub-opcodes for 'I' instructions" in file `Forvis_Spec_I.hs`).

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

2.1.4 Exception Codes

We define a type synonym for exception codes, and the values of all the standard exception codes 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...
```

and 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...
```

[5]I am grateful to my colleague Joe Stoy for 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").

2.1.5 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 result-expressions like this:

```
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
```

2.2 File `GPR_File.hs`: General Purpose Registers

This module 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 we only ever access it via the API functions in this file, i.e., we treat it like an *abstract data type*. Here is the representation and constructor:

line 32 GPR_File.hs

```
newtype GPR_File = GPR_File  (Data_Map.Map  GPR_Addr  GPR_Val)

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 address (0..31) with 0 (arbitrarily 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.[6]

Note: we do not here model accesses to the register file that concurrent, interleaved, and returning results out of order. This is fine for sequential interpretation, but will have to be enriched for concurrent interpretation.

[6]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.

2.3 File `Memory.hs`: Memory

This module implements a model of memory. We represent it using Haskell's `Data_Map.Map` type, which is an associative map (like a Python "dictionary") that associates addresses with values). This representation choice is purely internal to this module; for all other modules it is an abstract data type accessible only via the API we export from this module (so we can freely change the internal representation, in future, if we wish).

Here is the representation and constructor:

line 33 Memory.hs
```
-- This is a private internal representation that can be changed at
-- will; only the exported API can be used by clients.
-- We choose 32-bit data to cover the most common accesses in RV32 and RV64,
-- and since AMO ops are either 32b or 64b
-- 'f_reserved_addr' is the address range reserved by an 'LR' (Load Reserved) op.

data Mem = Mem { f_dm            :: Data_Map.Map  Integer  Integer,
                 f_reserved_addr :: Maybe (Integer, Integer)
               }

mkMem :: [(Integer, Integer)] -> Mem
mkMem  addr_byte_list =
  let
    addr_word_list  = addr_byte_list_to_addr_word_list  addr_byte_list
  in
    Mem  {f_dm            = Data_Map.fromList  addr_word_list,
          f_reserved_addr = Nothing }
```

[In anticipation of supporting the 'A' ISA option (atomics), we also have a field that "remembers" the address of the most recent Load-Reserved instruction, to be matched against the next Store-Conditional instruction.]

The contructor's argument is a list of (address, byte) pairs, and it initializes the data map with those contents. As a practical simulation-speed consideration, we represent memory in 32-bit words (even though it is byte-addressable) since most accesses are at 32-bit or 64-bit granularity (even in RV64, instruction accesses are at 32-bit granularity).

Later in the file we see the API to read memory:

line 102 Memory.hs
```
mem_read :: Mem -> InstrField -> Integer -> (Mem_Result, Mem)
mem_read    mem    funct3        addr =
...more...
```

and to write memory:

line 142 Memory.hs
```
mem_write :: Mem -> InstrField -> Integer -> Integer -> (Mem_Result, Mem)
mem_write    mem    funct3        addr       stv =
...more...
```

In both cases, the first argument is the memory itself, the second argument (`funct3` is the same 3-bit value in the original LOAD or STORE instruction indicating the size of the access (byte, halfword, word or doubleword). The third argument is the memory byte-address and the `mem_write` function has a fourth argument which is the store-value.

The internal details are not too interesting other; they're doing some bit-manipulation to accommodate the fact that our representation is in 4-byte words, and the access size may be for 1, 2, 4 or 8 bytes.

The last fragment of the function checks if the access is aligned:

line 129 Memory.hs

```
    if (is_LOAD_aligned  funct3  addr) then
      (Mem_Result_Ok  u64, mem)
    else
      (Mem_Result_Err exc_code_load_addr_misaligned,  mem)
```

and returns an exception result if so. A future version of this spec will make this a parameter, since an implementation is allowed to handle misaligned accesses directly (and not return an exception).

Later in the file you will also see the function `mem_amo` that handles read-modify-write operations in the "A" (atomics) extension, but you can ignore it for now while we focus on the base Integer instruction set.

Note: we do not model caches, or write-buffers, or any such hardware implementation artifact here. This is fine for sequential interpretation, but will have to be enriched for concurrent interpretation.

2.4 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.]

2.4.1 Handling RV32 and RV64 simultaneously

Although hardware implementations are typically either RV32 systems or RV64 systems, 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.). RV32 has a smaller integer instruction set than RV64, and limits calculations on values to 32-bit arithmetic.

2.4.2 Machine State

We define a new type representing a *complete* "machine state". The representation is a record (or struct).

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
                }
```

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.[7]

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).

This record-with-fields representation is a choice purely internal to this module. Clients of this module only access it via the `mstate_`*function* API that follows.[8]

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,
```

[7]We have not yet discussed FPRs, CSRs and MMIO, but they can be ignored for now while we focus on the base Integer instruction set.

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

```
                              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,
                              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
```

The `misa` argument is passed down to the CSR register file constructor; it can be ignored for now. The `addr_byte_list` is passed down into the memory constructor to intialize memory.

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

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

For those unfamiliar with functional programming, it is sometimes startling to see something as "large" as a machine state passed as an argument and returned as a result, but rest assured this is fine for our spec; these are just like functions in mathematics.

What follows in the file 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 :: Integer -> Machine_State -> Machine_State
mstate_pc_write    val        mstate = mstate { f_pc = val }
```

The `mstate_pc_read` 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.

Many of the API functions, such as those to read and write GPRs, FPRs or CSRs merely invoke 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). For a sequential one-instruction-at-a-time interpretation, without multiple paths to memory, with just one hart, it is fine to treat them as no-ops (more accurately, as the identity function on the machine state).

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.

2.5 File `Forvis_Spec_Common.hs`: Common "instruction-finishing" functions

Although there are dozens of different instruction opcodes (hundreds, if we count ISA extensions), there are only five or six ways in which they all "finish"– possibly write a value to a destination register, possibly increment the PC by 2 or 4, or write a new value into the PC, possibly increment the MINSTRET (number of instructions retired) register, and so on.

Another possibility is to trap, which does standard things like storing a cause in the MCAUSE register, storing the current PC in the xEPC register, deciding whether the next PC should come from the MTVEC, STVEC or UTVEC register (taking into account delegation in the MIDELEG and MEDELEG registers), manipulate the MSTATUS register in a certain stylized manner ("pushing" the interrupt-enable and privilege stacks), and so on.

Rather than replicate these few patterns in each instruction's semantic function, we collect them in this file in standard functions and just invoke them from each instruction's semantic function. 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 instruction 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 :: GPR_Addr -> Integer -> Bool -> Machine_State -> Machine_State
finish_rd_and_pc_incr    rd          rd_val     is_C    mstate =
  let mstate1 = mstate_gpr_write  rd  rd_val  mstate
      pc      = mstate_pc_read    mstate1
      delta   = if is_C then 2 else 4
      mstate2 = mstate_pc_write  (pc + delta)   mstate1
      mstate3 = incr_minstret     mstate2
  in
    mstate3
```

3 File `Forvis_Spec_I`: Base Integer Instruction Specs

We are now ready to look at this module which covers the whole RV32 Integer instruction set. The organization of the code in this module is also followed in the modules for other extensions (I64, C, A, M, Zifencei, F, D, Priv, Zicsr):

- Declaration of a type `Instr_I`, a data structure representing all the instructions in this group, along with each one's logical fields. This is a Haskell "algebraic data type". These are like "abstract syntax trees" for instructions in this group.
- Definitions of sub-opcodes for instructions in group I. These are values of fields in a 32-bit instruction that collectively refine it to a more specific instruction opcode.
- A decode function `decode_I` of type:
  ```
      decode_I :: RV -> Instr_32b -> Maybe Instr_I
  ```
  that takes a a 32-bit instruction and returns a result that is either:
  - `Nothing`: this is not an instruction in this group,
  - or `Just adt`: this is an instruction in this group, and `adt` is a value of the algebraic data type `Instr_I`, 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_I` 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_LUI`, `exec_AUIPC`, `exec_JAL`, ..., one per opcode, describing the semantics of that particular kind of instruction.

3.1 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 a 20-bit immediate value),
- *or* a `AUIPC` instruction, in which case it has two fields of type `GPR_Addr` and `InstrField` (the destination register rd and a 20-bit immediate value),
- *or* a `JAL` instruction, in which case it has two fields of type `GPR_Addr` and `InstrField` (the destination register rd and a 21-bit 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 a 12-bit immediate value),
- ... and so on."

Note, the width information (20-bit, 21-bit, 12-bit) is just a comment and is not enforced by type-checking.

3.2 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.

3.3 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 more 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 decode function essentially abtracts away lower-level details of how fields are laid out in 32-bit parcels and returns a higher-level, more abstract view of type `Instr_I`.

The decode 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`

3.4 The type of each I-instruction semantic function

The functions describing the semantics of each I instruction (to follow, shortly) all have the same type. In such a situation, it's good to give a name to this type (using a type-synonym):

line 257 Forvis_Spec_I.hs

```
type Spec_Instr_I = Bool -> Instr_I -> Machine_State -> Machine_State
--                  is_C    instr_I    mstate           mstate'
```

The first argument is a boolean (we'll consistently use the variable `is_C` for this) 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 for a 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`.

The second argument is the instruction in its decoded form; the third argument is the machine state, and the final outcome after executing the instruction is the new machine state.

3.5 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 261 Forvis_Spec_I.hs

```
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` guarantees that it can only be a valid I instruction.

3.6 Semantics of each I instruction

This is the meat of instruction semantics: what exactly does each kind of instruction do? There is one `exec_`*FOO* function for each opcode *FOO*. We examine excerpts of a few of them. The first I instruction, LUI, is very simple:

line 316 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  rd  rd_val  is_C  mstate
  in
    mstate1
```

The second argument is the pattern `(LUI rd imm20)` that is matched against the instruction, and gives us bindings for the `rd` and `imm20` fields as a result. There is no chance that this pattern fails, since this function is only called by the dispatcher `exec_instr_I` (above) when it is this kind of instruction.

It uses the 20-bit immedate to calculate a value `rd_val` to save in the destination register `rd`, and calls the standard "finish" function described in Sec. 2.5 to write the destination register and increment the PC, and returns the final machine state.

The `JALR` instruction does a bit more:

line 370 Forvis_Spec_I.hs

```
exec_JALR :: Spec_Instr_I
exec_JALR  is_C  (JALR  rd  rs1  imm12)  mstate =
  let
    misa   = mstate_csr_read  csr_addr_misa   mstate
    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  rs1  mstate

    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  rd  rd_val  new_pc'  mstate
              else
                finish_trap  exc_code_instr_addr_misaligned  new_pc'  mstate
  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. If MISA.C is active, then it cannot be misaligned since we just cleared the least-significant bit. If MISA.C is not active, then it is aligned only if bits [1:0] are 0.

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 487 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
```

It invokes `mstate_vm_read`, which is in file `Virtual_Mem.hs` (see Sec. 6.5), which does all memory reads. By examining `mstate`, that function 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. 2.1.5), indicating that it's ok (and contains the appropriate payload), or an error (and contains the appropriate exception code). In the case of an access fault, the `is_instr` boolean allows the memory system to determine if it should return an instruction fetch access fault or a load access fault.

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

line 593 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 invokes 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.

line 635 Forvis_Spec_I.hs

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

Near the end of the file we have the semantics of ECALL:

line 667 Forvis_Spec_I.hs

```
exec_ECALL :: Spec_Instr_I
exec_ECALL    is_C  (ECALL)  mstate =
  let
    priv = mstate_priv_read  mstate
    exc_code | priv == m_Priv_Level = exc_code_ECall_from_M
             | priv == s_Priv_Level = exc_code_ECall_from_S
             | priv == u_Priv_Level = exc_code_ECall_from_U
             | True                 = error ("Illegal priv " ++ show (priv))
    tval = 0
    mstate1 = finish_trap  exc_code  tval  mstate
  in
    mstate1
```

It decides the exception code depending on the current privilege level, and then finishes with a trap, supplying that exception code, and 0 for the "trap value" (that goes into the xTVAL register).

3.7 File `ALU.hs`: A pure integer ALU

The module `ALU` represents the "pure integer ALU", i.e., pure functions of one or two inputs representing the various integer 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.

4 File `Forvis_Spec_Instr_Fetch.hs`: Instruction Fetch

An instruction fetch can have three possible outcomes, which are expressed in this data type:

line 40 Forvis_Spec_Instr_Fetch.hs
```
data Fetch_Result = Fetch       Integer     -- Normal 32-bit instr
                  | Fetch_C     Integer     -- C (compressed) 16-bit instr
                  | Fetch_Trap  Exc_Code    -- e.g., misaligned access, page fault, etc.
```

The instruction fetch function takes a machine state and returns a 2-tuple: a `Fetch_Result` and a new machine state:

line 46 Forvis_Spec_Instr_Fetch.hs
```
instr_fetch :: Machine_State -> (Fetch_Result, Machine_State)
```

Note, while we may think of an instruction fetch as a pure "read", we return a potentially changed machine state because even a read can have side effects.[9]

Instruction fetch must be done carefully, avoiding spurious exceptions. Consider this scenario: the PC is pointing at the last two bytes of a virtual memory page. We do not know whether those two bytes contain (i) a C instruction, or (ii) the first two bytes of a normal 32-bit instruction, with PC+2 pointing at its remaining two bytes. In case (i), control flow (branches, traps) may never take us to PC+2, and so we should not unnecessarily read the two bytes there, in case that read results in a virtual memory exception, being on the next page. Thus, we should fetch the latter two bytes *only if necessary*.

The code for the `instr_fetch` function is structured to handle this. It 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, which may of course return either an exception or a 32-bit instruction:

line 58 Forvis_Spec_Instr_Fetch.hs
```
      let
        -- Read 4 instr bytes
        -- with virtual-to-physical translation if necessary.
        (result1, mstate1) = read_n_instr_bytes  mstate  4  pc
      in
        case result1 of
          Mem_Result_Err  exc_code -> (let
                                         tval    = pc
                                         mstate2 = finish_trap  exc_code  tval  mstate1
                                      in
                                         (Fetch_Trap  exc_code,  mstate2))
          Mem_Result_Ok  u32       -> (Fetch  u32,  mstate1)
```

If 'C' is supported, it first reads 2 bytes from memory and, if that is not an exception, checks if it encodes a 'C' instruction:

line 72 Forvis_Spec_Instr_Fetch.hs
```
      let
        -- 16b and 32b instructions; read 2 instr bytes first
        -- with virtual-to-physical translation if necessary.
        (result1, mstate1) = read_n_instr_bytes  mstate  2  pc
      in
```

[9]For example, a virtual memory access can change an 'Accessed' bit in a page table entry; a read may be from an I/O device where reads can have side-effects; if we model caches for concurrent interpretation, cache state can change; if we model performance counters, those could change.

```
        case result1 of
          Mem_Result_Err  exc_code -> (let
                                          tval    = pc
                                          mstate2 = finish_trap  exc_code  tval  mstate1
                                       in
                                         (Fetch_Trap  exc_code, mstate2))

          Mem_Result_Ok   u16_lo ->
            if is_instr_C  u16_lo then
              -- Is a 'C' instruction; done
              (Fetch_C  u16_lo,  mstate1)
            else
              (let
                  -- Not a 'C' instruction; read remaining 2 instr bytes
                  -- with virtual-to-physical translation if necessary.
                  -- Note: pc and pc+2 may translate to non-contiguous pages.
                  (result2, mstate2) = read_n_instr_bytes  mstate  2  (pc + 2)
...more...
```

If it is not a C instruction, then it is the first 2 bytes of a 32-bit instruction; it reads 2 more bytes from memory and returns it as a full 32-bit instruction.

5 File `Forvis_Spec_Execute.hs`: Top-level execute function

The function `exec_instr_32b` executes exactly one instruction. It takes an instruction and machine state and returns the updated machine state resulting from executing that instruction (the second component of the output 2-tuple, a `String`, is just for printing out instruction traces during simulation, and has no semantic significance). It first calls `decode_I` (and similar decode functions for other ISA extensions) to decide whether it is an I instruction and, if so, obtains the abstract version of the instruction (of type `Instr_I`).

line 60 Forvis_Spec_Execute.hs

```
exec_instr_32b :: Instr_32b -> Machine_State -> (Machine_State, String)
exec_instr_32b    instr_32b    mstate =
  let
    rv   = mstate_rv_read  mstate
    misa = mstate_csr_read  csr_addr_misa  mstate
    frm  = mstate_csr_read  csr_addr_frm   mstate
    is_C = False

    dec_I         = decode_I         rv        instr_32b
    dec_Zifencei  = decode_Zifencei  rv        instr_32b
    dec_Zicsr     = decode_Zicsr     rv        instr_32b
    dec_I64       = decode_I64       rv        instr_32b
...more...
```

The long list of nested `case` expressions that follow simply dispatch to the appropriate execution function; for example if it is an `Instr_I`, it invokes `exec_instr_I`, and so on.

line 84 Forvis_Spec_Execute.hs

```
    case dec_I of
      Just  instr_I -> (exec_instr_I  is_C  instr_I  mstate,
                        show  instr_I)
      Nothing ->
        case dec_I64 of
          Just instr_I64 -> (exec_instr_I64  is_C  instr_I64  mstate,
                             show  instr_I64)
...more...
```

At the end of the list of case expressions, i.e., if none of the decoders identified the instruction, it raises an illegal instruction exception:

line 124 Forvis_Spec_Execute.hs

```
                                        -- Illegal instruction trap, since does
                                        -- not decode to any 32b instr
                                        let
                                          tval    = instr_32b
                                          mstate1 = finish_trap  exc_code_illegal_instruction
                                                                 tval
                                                                 mstate
                                        in
                                          (mstate1, "Illegal instr 0x" ++
                                                    showHex instr_32b "")
```

The rest of the file is straightforward: the function `exec_instr_16b` is just like `exec_instr_32b` which we just examined, except for 16-bit C (compressed) instructions.

6 Privileged Architecture

6.1 `Arch_Defs.hs`: Privilege levels

The privileged architecture instroduces three 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
```

6.2 `Forvis_Spec_Priv.hs`: Privileged architecture instructions

This file follows the standard file organization for each extension/feature:

- An algebraic data type `Instr_Priv` for the abstract, decoded view of privileged architecture instructions;
- Definitions of sub-opcodes for privileged architecture instructions, followed by a decode function `decode_Priv`
- An execution-dispatch function `exec_instr_Priv` from decoded instructions to individual semantic functions;

- and, finally, a series of individual semantic functions `exec_xRET` (for MRET/SRET/ URET), `exec_WFI` and `exec_SFENCE_VMA`.

6.3 File `Forvis_Spec_Zicsr.hs`: CSR-GPR atomic swap instructions

This is the spec for CSR read and write instructions CSRRW, CSRRWI, CSRRS, CSRRSI, CSRRC, and CSRRCI. These are technically not part of the privileged architecture but form their own "Zicsr" extension, but we place this section here here because they are most heavily used in privileged code.

These instructions atomically read a CSR and place the result in a GPR while taking another GPR value (or an immediate value) and using it to write a new value into a CSR.

6.4 File `CSR_File.hs`: Control and Status Registers

Privileged architecture instructions interact heavily with the CSR register file. Like the GPR regsiter file, this module's private representation choice is to use Haskell's `Data.Map`.

The four most-significant bits of a CSR address encode permissions–whether or not a CSR can be read/written at a particular privilege level. The `csr_permission` function computes this permission, returning None, RO (read-only) or RW (read and write), which constitute the `CSR_Permission` data type.

The API functions `csr_read` and `csr_write` functions are the general-purpose access functions for CSRs. These functions do not check access permissions; they are only invoked after permissions have been checked. Mostly they just access the `Data.Map`, but several CSR addresses are merely restricted "views" of other CSRs, so these are treated specially before the general lookup/update.

For example,

- `USTATUS` and `SSTATUS` are restricted views of `MSTATUS`
- `UIE` and `SIE` are restricted views of `MIE`
- `UIP` and `SIP` are restricted views of `MIP`
- `FRM` and `FFLAGS` are restricted views of `FCSR`

The file continues with with definitions of CSR addresses and their reset values, organized into User, Supervisor and Machine privilege groups. After these general-purpose definitions, the file continues with more detailed support for key specific CSRs.

6.4.1 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 495 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 501 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 515 CSR_File.hs

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

6.4.2 The MSTATUS, SSTATUS and USTATUS CSRs

Another key CSR is MSTATUS ("Machine Mode Status"). The code for this section begins with definitions for the integer bit positions of its fields. Not all fields are used, and more restricted views of this CSR are known as SSTATUS (Supervisor Mode Status) and USTATUS (User Mode Status). Some "mask" definitions encode these views.

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.

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.

6.4.3 Miscellaneous CSR definitions

The next several sections have definitions for:

- The MTVEC, STVEC and UTVEC CSRs: functions to extract the "mode" and "base" fields of the MTVEC (Machine-level trap vector) CSR. The same functions can also be applied to the STVEC and UTVEC for the Supervisor-level and User-level versions.

- The MIP, SIP and UIP CSRs: symbolic names for bit positions in the MIP (Machine-level interrupt pending) CSR. It also defines masks for SIP and UIP which are restricted views of the same register at Supervisor and User privilege levels, respectively.
- The MCAUSE "Exception Cause" CSR: definition of the bit position that distinguishes interrupts from synchronous traps, for RV32 and RV64 modes, and a predicate to decide whether an exception code is for an interrupt of synchronous trap.
- The FCSR "floating point status" CSR: definitions for bit positions in this CSR.

6.4.4 Interrupt-pending test and WFI-resumption test

The function `csr_interrupt_pending` determines if the CSR state allows taking an interrupt at the current privilege (specifically, CSRs MISA, MSTATUS, MIP, MIE, MIDELEG, and SIDELEG).

It takes into account the current privilege, whether an interrupt is pending in the CSR MIP, whether interrupts are delegated to a lower privilege level (in CSRs MIDELEG and SIDELEG), and whether interrupts are enabled at the appropriate level (in CSRs MSTATUS/ SSTATUS/ USTATUS and MIE/ SIE/ UIE). If an interrupt cannot be taken, it returns `Nothing`; otherwise it returns `Just` *ec*, where *ec* is the exception code describing the kind of interrupt.

The function `csr_wfi_resume` returns a boolean indicating whether the hart is allowed to resume execution if it is currently waiting at a WFI (Wait For Interrupt) instruction. The resumption condition is weaker than the condition used by the previous function that decides whether we can actually take an interrupt, because it does not take into account current privilege, delegation, global (MSTATUS) interrupt-enable fields, and so on. WFI instructions are typically used inside a loop where, if the interrupt is not actually taken, we just come back to wait again at the WFI.

6.5 File `Virtual_Mem.hs`: Virtual Memory

This module contains essentially all the code to support virtual memory, covering the Sv32, Sv39 and Sv48 virtual memory schemes.

Memory access begins in the semantic functions for various memory access instructions:

- `exec_LOAD` and `exec_STORE` in module `Forvis_Spec_I`;
- `exec_AMO` in module `Forvis_Spec_A`;
- `exec_FLW` and `exec_FSW` in module `Forvis_Spec_F`;
- and `exec_FLD` and `exec_FSD` in module `Forvis_Spec_D`.
- (Compressed memory access instructions, in module `Forvis_Spec_C` expand to one of the above).

These functions call `mstate_vm_read`, `mstate_vm_write` and `mstate_vm_amo` in the current file, `Virtual_Mem.hs`. Each of these functions first calls `fn_vm_is_active` to check whether or not we are currently running in a mode that requires virtual-to-physical translation. A subtlety is that the current effective privilege may be modified by the MSTATUS.MPRV and MSTATUS.MPP bits.

If virtual memory is active, `mstate_vm_read`, `mstate_vm_write` and `mstate_vm_amo` then call `vm_translate` which is described later in the file, to translate the virtual address, returning either a successful translation into a physical address or an exception (such as a page fault or access fault). Note, translation returns an updated memory state because it may modify the "access" and "dirty" bits in a page table entry. Translation is done with a "page table walk", which is specified in the following function:

```
line 196 Virtual_Mem.hs
    -- This function 'ptw' is the recursive Page Table Walk
    -- 'ptn_pa' is the address of a a Page Table Node at given 'level'
    ptw :: Machine_State -> Integer -> Int ->  (Mem_Result, Machine_State)
    ptw    mstate           ptn_pa     level =
...more...
```

This is a recursive function that descends the hierarchy of the page table, taking into account that any level can point at a leaf page (page, megapage, gigapage or terapage), and taking into account the protection bits in Page Table Entries (PTEs).

Hardware implementations typically implement a TLB (Translation Look-aside Buffer) to cache page-table entries for faster future access, avoiding repeated page table walks. However, that is primarily a performance optimization, and for simplicity we ignore it here in the semantics and perform the full page-table walk on each memory access.[10]

The rest of the file contains various help-functions for `vm_translate`.

7 File `Forvis_Spec_Interrupts.hs`: Interrupts

This module contains two functions related to interrupts. The `take_interrupt_if_any` function can be applied between the execution of any two instructions:

```
line 24 Forvis_Spec_Interrupts.hs
mstate_take_interrupt_if_any :: Machine_State -> (Maybe Exc_Code, Machine_State)
mstate_take_interrupt_if_any    mstate =
...more...
```

It first uses the function `csr_interrupt_pending` (described in Sec. 6.4.4) to determine if the current machine state allows taking an interrupt and, if so, which kind of interrupt.

If so, `take_interrupt_if_any` actually "takes the interrupt", i.e., it applies `mstate_upd_on_trap` to update the PC so that the next instruction fetched will be from the interrupt handler, with CSRs holding appropriate values for the handler.

If no interrupt can be taken now, it returns `Nothing` and an unchanged machine state. The second function in the file is:

```
line 60 Forvis_Spec_Interrupts.hs
mstate_wfi_resume :: Machine_State -> Bool
mstate_wfi_resume    mstate =
...more...
```

which merely calls `csr_wfi_resume` (also described in Sec. 6.4.4) to check if the CPU can resume from a WFI (Wait For Interrupt) state. Note, WFI does not actuall take an interrupt (hence this function does not return any updated machine state), it merely resumes at the next instruction, which may take the interrupt (using `mstate_take_interrupt_if_any`).

[10] In the concurrent interpretation, cacheing of PTEs in TLBs must be modeled because it does become observable and interacts with the SFENCE.VMA instruction.

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

In Sec. 1.2 we discussed how the same spec code can be *interpreted* in two different ways, sequential or concurrent. The file `Run_Program_Sequential.hs` contains the sequential interpreter, including various hooks to help in simulation (such as printing out instruction traces). The main function is this:

```
line 50 Run_Program_Sequential.hs
run_loop :: Int       -> (Maybe Integer) -> Machine_State -> IO (Int, Machine_State)
run_loop    maxinstrs  m_tohost_addr      mstate = do
...more...
```

The `maxinstrs` argument is a simulation aid to limit the number of instructions executed (stopping runaway simulations). The `m_tohost_addr` argument is another simulation aid; it stops simulation if the CPU writes to a particular location called `m_tohost_addr`; this facility is used in certain test programs. The `mstate` argument is the initial machine, when this function is first called, from the function `run_program_from_files` in `Main_Run_Program.hs`.

The result returned by `run_loop` is of type `(Int,Machine_State)`. The first component is a value written into the `to_host` location if that facility is used to signal termination, otherwise it is 0. The second component is the final machine state.

If we removed the simulation aids, the type of `run_loop` would be:

```
      Machine_State -> IO Machine_State
```

i.e., it simply transforms the machine state by repeatedly executing instructions.[11] `run_loop` is literally a loop. The last line of the function:

```
line 136 Run_Program_Sequential.hs
                         run_loop  maxinstrs  m_tohost_addr  mstate5))
```

shows a tail-recursive call to itself; this is how loops are expressed in Haskell.

The body of `run_loop` does the following (if you scan through the body ignoring the clutter of simulation aids like printing debug messages, detecting termination and self-loops, etc.):

- it calls `mstate_io_tick` to "advance" the conceptually concurrent processes in I/O devices,
- it moves console terminal input from the console into the UART buffer,
- it calls the `fetch_and_execute` (later in this file) to fetch and execute one instruction,
- it moves console terminal output from the UART buffer to the console,
- and loops.

9 Brief Description of Other Spec Source Files

The list below is in alphabetic order.

9.1 `Bit_Utils.hs`

This contains utilities for bit manipulation, including bit-field extraction and concatenation, sign- and zero-extension, truncation, conversion, etc. that are relevant for these semantics.

[11] The `IO` type constructor is the Haskell mechanism by which we can do IO within this loop, such as printing instruction traces, and reading and writing the console terminal.

9.2 `Forvis_Spec_A.hs`

Spec for the “A” ISA extension for Atomic memory operations (Load Reserved, Store Conditional, Atomic swap, add, and/or/xor, min/max).

9.3 `Forvis_Spec_C.hs`

Spec for the “C” ISA extension for compressed (16-bit) instructions.

9.4 `Forvis_Spec_D.hs`

Spec for the “D” ISA extension for double-precision floating point operations.

9.5 `Forvis_Spec_F.hs`

Spec for the “F” ISA extension for single-precision floating point oeprations.

9.6 `Forvis_Spec_I64.hs`

The RV64 base Integer instruction set.

9.7 `Forvis_Spec_M.hs`

Spec for the “M” extension for integer multiply/divide operations.

9.8 `Forvis_Spec_Zifencei.hs`

Spec for the “Zifencei” extension for the FENCE.I instruction.

9.9 `FPR_File.hs`

The floating point register file used by the “F” and “D” extensions.

9.10 `FPU.hs`

the floating-point analog of the integer `ALU` module, representing a “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.

9.11 `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.

10 Brief Description of Other Non-Spec Source Files

These files are not part of the spec *per se*. They They represent additional infrastructure to attach the spec (which just represents a CPU) with a minimal but complete "system". Collectively, they can be compiled and linked as a Haskell executable that is a RISC-V simulator, for the *sequential* interpretation. System elements include a boot ROM, a memory, a memory-mapped timer (MTIME, MTIMECMP), a memory-mapped place to request software interrupts (MSIP), a serial communication device for the console (UART), etc. These are sometimes collectively referred to as a "platform".

The list below is in alphabetic order.

10.1 `Address_Map.hs`

specifies the range of addresses for supported main memory, boot ROM memory, memory-mapped I/O, and specific memory-mapped addresses for MTIME. MTIMECMP, MSIP, the UART (serial console communications device), etc.

10.2 `Elf.hs`

This contains functions for reading ELF files representing RISC-V executable programs. Assemblers, compilers and linkers (such as gcc and llvm) produce ELF files as their output (see also `Read_Mem_Hex.hs` below).

10.3 `Main.hs`

This is the "main" program compiled by the Haskell compiler. It contains several alternative use-cases, one of which is statically chosen by editing this file and ensuring that we comment-out all but one alternative.

- `Main_RunProgram.hs`: a free-running RISC-V simulator. Reads RISC-V binaries (ELF or Mem Hex), initializes architecture state and memory, and calls `Run_Program_Sequential` to run the loaded program, up to a specified maximum number of instructions, performing console I/O, timer interrupts, and more.
- `Main_TandemVerifier.hs`: a program that verifies that a detailed instruction trace produced by some other RISC-V implementation (hardware or software) is a legal instruction trace. Please contact the author for more information about this use case [Note: this option is not up-to-date as of January 2019; it will be brought up-to-date].
- `Main_Test_Virtual_Mem.hs`: This is just a bit of test scaffolding to test `Virtual_Mem.hs` on its own. It builds a small custom page table and accesses various virtual addresses in memory to check that address translation and protection are working as expected.

10.4 `MMIO.hs`

implements the memory-mapped I/O, including MTIME, MTIMECMP, MSIP, and the console UART, and the read and write API to access them. Since IO conceptually "runs" concurrently with the CPU, it contains a function `mmio_tick_mtime` that is called periodically from the top-level simulation loop in order to "advance" these processes.

10.5 `Read_Hex_File.hs`

This contains functions for reading "Hex Memory" files for initial memory contents. Hex Memory files are standard for hardware description languages such as Verilog, SystemVerilog and VHDL. They are text files describing memory contents in hexadecimal format, one entry per memory location. (see also `Elf.hs` below).

10.6 `Run_Program_Sequential.hs`

This 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.

10.7 `UART.hs`

This is a model of the popular National Semiconductor NS16550A UART. A UART (Universal Asynchronous Receiver/Transmitter) is a serial communication device through which the CPU communicates with a console terminal.

References

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

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

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

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

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

[6] 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.

[7] 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.

A Haskell cheat sheet for reading Forvis code

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:

```
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)

B Extending Forvis to model new ISA features

This is a brief guide for those who are extending the RISC-V ISA spec with new ISA extensions (such as instructions for vector processing, bit-manipulation, managed languages, cryptography, security and so on), and would like to extend Forvis to specify these new extensions formally.

Please refer to Fig 1 on page 1. You will see that it is already modularlized according to the standard extensions (C, A, M, F, D and Zifencei). These are almost like "plug-ins" because they all follow the same pattern, but they are not truly dynamic plug-ins because we wish to compile it with static type-checking and optimization. Essentially, you'd follow the same pattern for your new extension.

In the following, we will use "FOO" as the name for the new extension.

B.1 Possibly extend the machine state with new architectural state

Some new instructions operate on existing architectural state (PC, GPRs, FPRs, CSRs), in which case nothing needs to be done here.

If FOO adds new CSRs, add their definitions to `CSR_File.hs`, following the existing definitions as examples. Examples there include CSRs with complex bit-field manipulations and multiple views via different addresses, such as MSTATUS/ SSTATUS/ USTATUS and MIE/ SIE/ UIE.

If FOO has other new architectural state, ultimately it must be added as a field in the `Machine_State` data type. Rather than adding multiple items to that type, we recommend that you create a new module for that state type, say `FOO_state`, by analogy with `GPR_File.hs`, `FPR_File.hs`, and `CSR_File.hs`. I.e., define a data type `FOO_state` to represent the state, and API functions to create that state and to read/write the state.

Then, add a single field `f_foo :: FOO_State` to `Machine_State`. Augment the `mkMachine_State` constructor to create and intialize the FOO field (like "`f_gprs=mkGPR_File`").

In `Machine_State.hs`, add API forwarding functions for FOO. For example, the `mstate_gpr_read` function simply forwards to the `gpr_read` function in the `GPR_File` module.

B.2 Possibly add a new major opcode for FOO

If FOO uses a new major opcode (least-significant 7 bits of an instruction), add the definition to `Arch_Defs.hs` in a style similar to the opcodes already defined there:

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...
```

Not all extensions need a new major opcode. For example, the “M” extension uses `opcode_OP` and `opcode_OP_32` from the base integer instruction set.

B.3 Write your main semantics module, `Forvis_Spec_FOO.hs`

We suggest you use `Forvis_Spec_I` as a reference, and organize your semantics file along similar lines. Define an algebraic data type `Instr_FOO` representing abstract decoded instructions of FOO, similar to:

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...
```

Define sub-opcodes for instructions in FOO, similar to:

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...
```

Define a decode function for instructions in FOO, similar to:

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...
```

Define the type of the FOO semantic functions, similar to:

line 257 Forvis_Spec_I.hs

```
type Spec_Instr_I = Bool -> Instr_I -> Machine_State -> Machine_State
--                  is_C    instr_I    mstate           mstate'
```

Define an execution-dispatch function, similar to:

line 261 Forvis_Spec_I.hs
```
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...
```

that dispatches to individual semantic functions for each instruction in FOO. Finally, define specification functions for each new instruction in FOO, similar to:

line 316 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  rd  rd_val  is_C  mstate
  in
    mstate1
```

B.4 Add a decode clause and an execute clause to `Forvis_Spec_Execute`

In `Forvis_Spec_Execute` add a decode clause for FOO, similar to:

line 74 Forvis_Spec_Execute.hs
```
    dec_M         = decode_M         rv        instr_32b
```

and add an execution-dispatch clause to the large nested case expression, similar to:

line 93 Forvis_Spec_Execute.hs
```
            case dec_M of
              Just instr_M -> (exec_instr_M  is_C  instr_M  mstate,
                               show  instr_M)
              Nothing ->
...more...
```

Functionally, it does not matter where you insert this in the nested case statement because the decodes should all be mutually exclusive, i.e., no 32-bit pattern should successfully decoded by two different extensions. This ensures that no more than one clause of the nested case expression will fire, no matter what the ordering of nesting.

However, from a simulation point of view, placing frequent instruction extensions earlier in the case expression list may provide some (small) speed improvement.

B.5 General advice

Try to encapsulate your extension FOO as much as possible, i.e., as much as possible, everything about FOO should be confined to its own files, and there should be minimal leakage of information about FOO into other files. Some cross references are unavoidable:

- FOO state has to be incorporated into `Machine_State`.
- FOO instructions will likely use definitions in `Arch_Defs` and `Forvis_Common` and perhaps `Virtual_Mem` and/or `Memory`.
- `Forvis_Spec_Execute` has to be extended with FOO's decoder and a dispatch to FOO's execute function.

Except for these, encapsulate information as locally as possible.

C Extending the Forvis system with new I/O devices or accelerators

This is not a technically an extension of Forvis itself, since Forvis is only concerned with the CPU (ISA semantics). Nevertheless, some people may want to use Forvis as an environment within which to experiment with a formal spec of a new I/O device or accelerator which, to Forvis, just looks like memory-mapped I/O (MMIO).

We suggest using `UART.hs` as a template to follow.

C.1 Creating a module for FOO

For your new I/O device FOO, create a new module `FOO` in file `FOO.hs`.

Define a data type representing the "state" of your I/O device, similar to:

line 92 UART.hs

```
data UART_NS16550A = UART_NS16550A {
  f_rbr_a :: String,
  f_rbr_b :: String,    -- 0    Read-only

  f_thr_a :: String,
  f_thr_b :: String,    -- 0    Write-only

  f_ier   :: Integer,     -- 1
...more...
```

This will contain FOO's device registers and local memories, etc. Define a constructor function for FOO, similar to:

line 121 UART.hs

```
mkUART :: UART_NS16550A
mkUART =  UART_NS16550A { f_rbr_a = "",
                          f_rbr_b = "",
...more...
```

Define the "memory read" function for FOO, similar to:

line 176 UART.hs

```
uart_read :: UART_NS16550A -> Integer -> (Integer, UART_NS16550A)
uart_read  uart  addr_offset =
...more...
```

and the "memory write" function for FOO, similar to:

line 228 UART.hs

```
uart_write :: UART_NS16550A -> Integer -> Integer -> UART_NS16550A
uart_write  uart  addr_offset  val =
...more...
```

The module may contain other functions specific to FOO. For example, the UART contains functions for enqueueing incoming characters and dequeueing outgoing characters.

C.2 Integrating FOO into the “system”

In `Address_Map.hs`, you will have to add the address-range for FOO, similar to:

line 67 Address_Map.hs
```
-- Other programs and tests use a UART for console I/O
addr_base_UART = 0xC0000000              :: Integer
addr_size_UART = 0x80                    :: Integer
```

and add it to the list of MMIO address ranges, similar to:

line 82 Address_Map.hs
```
-- Supported MMIO address ranges
mmio_addr_ranges :: [ (Integer, Integer) ]
mmio_addr_ranges = [ (addr_htif_console_out,  (addr_htif_console_out + 8)),
                     (addr_base_UART,         (addr_base_UART        + addr_size_UART)),
...more...
```

In `MMIO.hs` you will have to import your new module:

line 23 MMIO.hs
```
import UART
```

add the device type as a new field in the `MMIO` data type:

line 41 MMIO.hs
```
  -- UART for console I/O
  f_uart :: UART_NS16550A
```

and add the constructor to the `mkMMIO` constructor:

line 53 MMIO.hs
```
                f_uart          = mkUART
```

If FOO raises interrupts, it will have to be dealt with like this:

line 80 MMIO.hs
```
mmio_has_interrupts :: MMIO -> (Bool, Bool, Bool)
mmio_has_interrupts  mmio =
  let
    eip  = uart_has_interrupt  (f_uart  mmio)
...more...
```

In the “memory read” function `mmio_read`, add a dispatch for FOO:

line 142 MMIO.hs
```
  -- UART
  else if ((addr_base_UART <= addr) && (addr < (addr_base_UART + addr_size_UART))) then
    let
      uart       = f_uart  mmio
      (v, uart') = uart_read  uart  (addr - addr_base_UART)
      mmio'      = mmio { f_uart = uart' }
    in
      (Mem_Result_Ok  v,  mmio')
```

In the "memory write" function `mmio_write`, add a dispatch for FOO:

```
line 199 MMIO.hs
  -- UART
  else if ((addr_base_UART <= addr) && (addr < (addr_base_UART + addr_size_UART))) then
    let
      uart  = f_uart  mmio
      uart' = uart_write  uart  (addr - addr_base_UART)  val
      mmio' = mmio {f_uart = uart'}
    in
      (Mem_Result_Ok  0, mmio')
```

In `MMIO.hs` you will also see UART-specific API functions for console input and output to the UART that are simply forwarded to the UART. For the UART, these functions are called from the the outer `Run_Program_Sequential.hs` to communicate to the console terminal. FOO may need analogous forwarding API functions (pure accelerators typically do not need this).

C.3 Running an I/O device concurrently with the CPU

Some devices are not passive–they may run some "process" concurrently with the CPU. For example the timer accessed via the MTIME address ticks upwards constantly, independent of the CPU. The UART device moves data to/from its buffers and console terminal. An accelerator will typically compute something while the CPU busy-waits or does something else.

This kind of concurrent activity is driven by periodic calls to:

```
line 445 Machine_State.hs
mstate_io_tick :: Machine_State -> Machine_State
mstate_io_tick  mstate =
...more...
```

which, in turn, "runs" I/O devices by by calling:

```
line 61 MMIO.hs
mmio_tick :: MMIO -> MMIO
mmio_tick  mmio =
```

You will need to hook FOO's concurrent processes into `mmio_tick`, i.e., insert a call to `FOO_tick` which performs a slice of work for FOO.