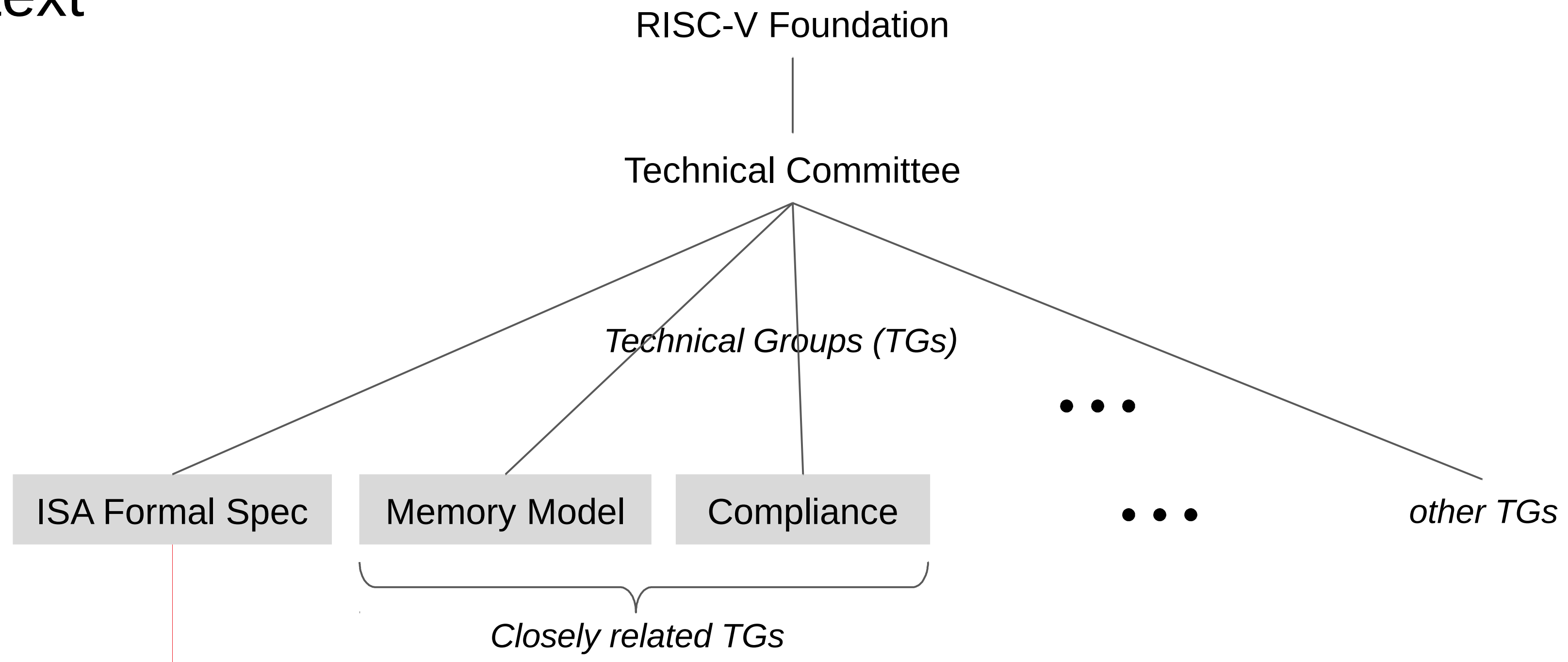
Formal Specification of the RISC-V Instruction Set Architecture

Niraj Sharma and Rishiyur S. Nikhil
{niraj.sharma, nikhil} at bluespec.com

RISC-V Workshop at IIT Madras, Chennai, India
Thursday, July 19, 2018

Context

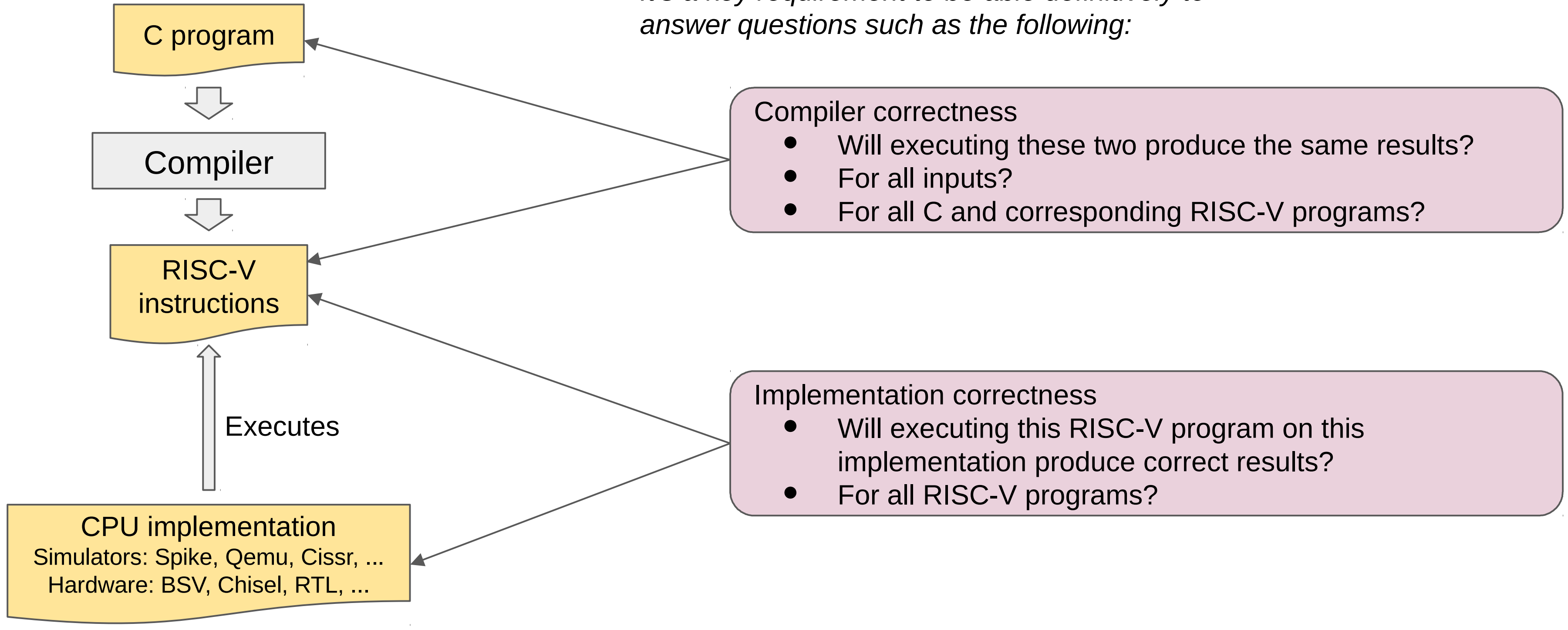


This talk:

- The challenges (and how the TG is exploring multiple approaches)
- Examples from one of these approaches

Of What Use is an ISA Formal Spec?

It's a key requirement to be able definitively to answer questions such as the following:


C program
Compiler
RISC-V instructions
Executes
CPU implementation
Simulators: Spike, Qemu, Cissr, ...
Hardware: BSV, Chisel, RTL, ...

Compiler correctness
- Will executing these two produce the same results?
- For all inputs?
- For all C and corresponding RISC-V programs?

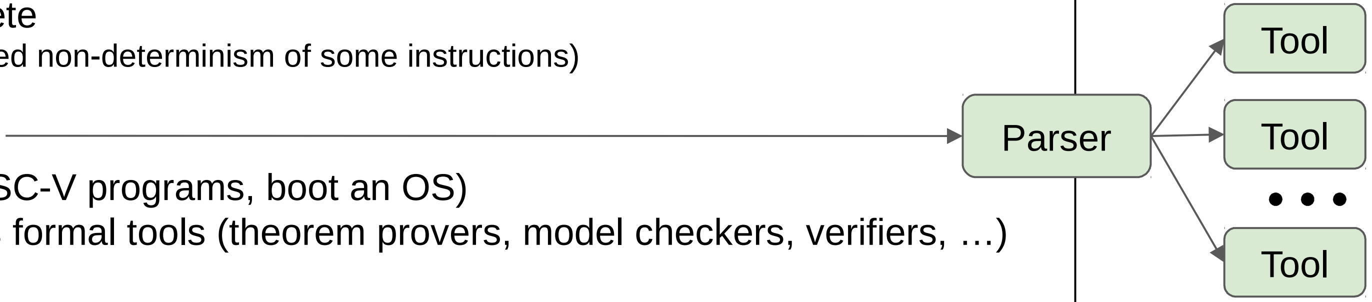
Implementation correctness
- Will executing this RISC-V program on this implementation produce correct results?
- For all RISC-V programs?

There are many other uses as well; the unifying theme is: ***Provable Correctness***

What is an ISA Formal Spec?

Goals:

- Clear and understandable to the human reader
  - Not cluttered with implementation considerations; no micro-architectural detail
  - Accessible to the practicing engineer: implementers of hardware, simulators, compilers, OSs, libraries of concurrent data structures, ...
- Precise and complete
  - (including allowed non-determinism of some instructions)
- Machine readable
- Executable (run RISC-V programs, boot an OS)
- Usable with various formal tools (theorem provers, model checkers, verifiers, …)



English-language text specs, and instruction-set simulators (Spike, Qemu, Cissr) can be regarded as specs, but they typically do not meet many of these goals.

Challenge 1: Universality

The spec must capture every possible legal RISC-V implementation:

- All base instruction sets (RV32, RV64) and every standard extension (I, M, A, F, D, C, …)
- All Privilege Combinations (M, M+U, M+S+U)
- All allowed implementation choices. Examples:
    - Misaligned memory access may be supported using traps, and may not be atomic
    - CSR SATP.MODE is a WARL field (Write Any, Read Legal). Which legal value?
    - Sv48 is optional but "Implementations that support Sv48 should also support Sv39"
    - An implementation can write the CSR MISA.C bit ("compressed instructions"); what happens to MTVEC register if it is only 2-byte aligned when MISA.C is written?

However, to be used for verification of a particular implementation,
the spec must be configurable/constrained to a particular set, to match the implementation:

- Choice of base instruction set(s) and extensions
- Implementation choices

This argues for extreme parameterization of the spec.
Completeness and parameterizability are in tension with readability.

Challenge 2: Extensibility

There are more standard ISA extensions currently being developed by various RISC-V Foundation Technical Groups:

- Vector, Bit Manipulation, Crypto, …

And no doubt there will be even more in the future.

The formal spec should be extensible by each Task Group developing a new standard extension.

Groups creating non-standard and proprietary extensions may also wish to extend the formal spec for their own purposes.

Challenge 3: Concurrency and Weak Memory Models

The simplistic view of an ISA is:

- it executes one instruction at a time
- each instruction effects a machine state transition *atomically* (registers, memory)

But most implementations are more complex:

- Intra-hart concurrency: pipelining, speculation, out-of-order execution, separate instruction and data channels
- Vertical memory-system concurrency: TLBs, multi-level caches
- Horizontal concurrency: multithreading (multi-hart), multicore, shared memory access

Weak Memory Models are quite different from idealized "Sequential Consistency":

- Even in a single-hart system: need for FENCE.I, FENCE, SFENCE.VMA
- In multi-hart systems, the RISC-V Weak Memory Model allows a LOAD to return different results on different implementations (within a certain allowed range of legal results).

The Formal ISA Spec must capture the allowed non-determinisms.
And, again, this is in tension with readability.

The "Formal ISA Spec Technical Group" is addressing these challenges

There are no precedents in the literature that attack the breadth and depth of these challenges, and so the RISC-V Foundation, and this TG, are breaking new ground here.
The TG is exploring multiple approaches concurrently:

- MIT
  - In Haskell, connecting to Coq formal tools in particular.

  https://github.com/mit-plv/riscv-semantics
- U. Cambridge and SRI International
  - SAIL DSL (domain specific language), based on most experience in addressing the concurrency challenge.

  https://github.com/rems-project/sail
- Bluespec: ***Forvis*** ("Formal RISC-V ISA spec")
  - In "Extremely Elementary" Haskell for extreme readability.
  - Same spec code has simple one-instruction-at-a-time and concurrent readings.

  https://github.com/rsnikhil/RISCV-ISA-Spec

  } Examples follow
- Clifford Wolf
  - In Verilog (!).

  https://github.com/cliffordwolf/riscv-formal
- Galois
  - In Haskell (details and public link not available to date).

All these efforts are Free and Open Source, and are on github.com.

We invite you to download them and experiment with them.

And, please provide feedback!

The remainder of this talk will show examples from one of these efforts:

- Bluespec: ***Forvis*** ("Formal RISC-V ISA spec")
    - In "Extremely Elementary" Haskell for extreme readability.
    - Same spec code has simple one-instruction-at-a-time and concurrent readings.

https://github.com/rsnikhil/RISCV-ISA-Spec

FORVIS ("Formal RISC-V ISA Spec") Design Principles

https://github.com/rsnikhil/RISCV-ISA-Spec

- **Extreme readability**: written in "Extremely Elementary" Haskell
  - Clarity from purely functional, mathematical style: each instruction's semantics explained as `Machine_State -> Instr -> Machine_State` function, no hidden side effects.
  - Easy read for those who do not know and may not want to learn Haskell.
    Avoids Haskell features that are unfamiliar to most people (typeclasses, monads, higher-order functions, Currying and partial application, …). It is practically a "Python subset" of Haskell, except with Haskell's strong type-checking and limited use of Haskell's pattern-matching and Haskell's better suitability for use by formal tools.
  - Trivial automatic translation into other languages.
- **Familiar style** of classic instruction-set manuals
  - "One page" per instruction, showing coding, legality conditions, and semantics.
- **Same code for sequential and concurrent reading**
  - Sequential: simple, one-instruction-at-a-time. Adequate for many use cases.
    - Direct execution as a Haskell program.
  - Concurrent: to describe allowable non-determinism and interaction with Weak Memory Model
    - Execution with an alternate (non-Haskell) interpreter.

FORVIS example (all instr specs follow this "one page" style)

```
opcode_JAL = 0x6F :: InstrField    -- 7'b_11_011_11

spec_JAL :: Machine_State -> Instr -> (Bool, Machine_State)
spec_JAL    mstate       instr =
  let
    -- Instr fields: J-type
    (imm20, rd, opcode) = ifields_J_type  instr

    -- Decode check
    is_legal = (opcode == opcode_JAL)

    -- Semantics
    pc     = mstate_pc_read  mstate
    rd_val = pc + 4

    x_u64   = zeroExtend_u32_to_u64  imm20
    y_u64   = shiftL  x_u64  1                -- offset imm20 is in multiples of 2 bytes
    z_u64   = signExtend  y_u64  21           -- sign-extend 21 lsbs
    new_pc  = cvt_s64_to_u64 ((cvt_u64_to_s64  z_u64) + (cvt_u64_to_s64  pc))
    aligned = ((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
    (is_legal, mstate1)
```

https://github.com/rsnikhil/RISCV-ISA-Spec/src/Forvis_spec.hs

FORVIS example (all instr specs follow this "one page" style)

```
opcode_JAL = 0x6F :: InstrField    -- 7'b_11_011_11

spec_JAL :: Machine_State -> Instr -> (Bool, Machine_State)
spec_JAL    mstate           instr =
  let
    -- Instr fields: J-type
    (imm20, rd, opcode) = ifields_J_type  instr

    -- Decode check
    is_legal = (opcode == opcode_JAL)

    -- Semantics
    pc     = mstate_pc_read  mstate
    rd_val = pc + 4

    x_u64   = zeroExtend_u32_to_u64  imm20
    y_u64   = shiftL  x_u64  1             -- offset imm20 is in multiples of 2 bytes
    z_u64   = signExtend  y_u64  21        -- sign-extend 21 lsbs
    new_pc  = cvt_s64_to_u64 ((cvt_u64_to_s64  z_u64) + (cvt_u64_to_s64  pc))
    aligned = ((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
    (is_legal, mstate1)
```

Major opcode

Every opcode's spec has this type

Result: Machine state after JAL

Result: True if 'instr' is a legal JAL

Arg: 32b instruction

Arg: Machine state before JAL

https://github.com/rsnikhil/RISCV-ISA-Spec/src/Forvis_spec.hs

FORVIS example (all instr specs follow this "one page" style)

```
opcode_JAL = 0x6F :: InstrField    -- 7'b_11_011_11

spec_JAL :: Machine_State -> Instr -> (Bool, Machine_State)
spec_JAL    mstate       instr =
  let
    -- Instr fields: J-type
    (imm20, rd, opcode) = ifields_J_type  instr

    -- Decode check
    is_legal = (opcode == opcode_JAL)

    -- Semantics
    pc     = mstate_pc_read  mstate
    rd_val = pc + 4

    x_u64   = zeroExtend_u32_to_u64  imm20
    y_u64   = shiftL  x_u64  1               -- offset imm20 is in multiples of 2 bytes
    z_u64   = signExtend  y_u64  21          -- sign-extend 21 lsbs
    new_pc  = cvt_s64_to_u64 ((cvt_u64_to_s64  z_u64) + (cvt_u64_to_s64  pc))
    aligned = ((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
    (is_legal, mstate1)
```

Apply standard decoding function for J-Type encoding

Pattern-match to bind J-type fields

https://github.com/rsnikhil/RISCV-ISA-Spec/src/Forvis_spec.hs

FORVIS example (all instr specs follow this "one page" style)

```
opcode_JAL = 0x6F :: InstrField    -- 7'b_11_011_11

spec_JAL :: Machine_State -> Instr -> (Bool, Machine_State)
spec_JAL    mstate       instr =
  let
    -- Instr fields: J-type
    (imm20, rd, opcode) = ifields_J_type  instr

    -- Decode check
    is_legal = (opcode == opcode_JAL)

    -- Semantics
    pc     = mstate_pc_read  mstate
    rd_val = pc + 4

    x_u64   = zeroExtend_u32_to_u64  imm20
    y_u64   = shiftL  x_u64  1                -- offset imm20 is in multiples of 2 bytes
    z_u64   = signExtend  y_u64  21           -- sign-extend 21 lsbs
    new_pc  = cvt_s64_to_u64 ((cvt_u64_to_s64  z_u64) + (cvt_u64_to_s64  pc))
    aligned = ((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
    (is_legal, mstate1)
```

Legality check.
(Other instructions can have more complex checks)

https://github.com/rsnikhil/RISCV-ISA-Spec/src/Forvis_spec.hs

FORVIS example (all instr specs follow this "one page" style)

```
opcode_JAL = 0x6F :: InstrField    -- 7'b_11_011_11

spec_JAL :: Machine_State -> Instr -> (Bool, Machine_State)
spec_JAL    mstate       instr =
  let
    -- Instr fields: J-type
    (imm20, rd, opcode) = ifields_J_type  instr

    -- Decode check
    is_legal = (opcode == opcode_JAL)

    -- Semantics
    pc      = mstate_pc_read  mstate
    rd_val = pc + 4

    x_u64   = zeroExtend_u32_to_u64  imm20
    y_u64   = shiftL  x_u64  1                 -- offset imm20 is in multiples of 2 bytes
    z_u64   = signExtend  y_u64  21            -- sign-extend 21 lsbs
    new_pc  = cvt_s64_to_u64 ((cvt_u64_to_s64  z_u64) + (cvt_u64_to_s64  pc))
    aligned = ((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
    (is_legal, mstate1)
```

JAL behavior.
We use explicit sign- and zero-extensions, explicit conversions from unsigned to signed, etc.; nothing hidden, nothing mysterious.

https://github.com/rsnikhil/RISCV-ISA-Spec/src/Forvis_spec.hs

FORVIS example (all instr specs follow this "one page" style)

```
opcode_JAL = 0x6F :: InstrField    -- 7'b_11_011_11

spec_JAL :: Machine_State -> Instr -> (Bool, Machine_State)
spec_JAL    mstate       instr =
  let
    -- Instr fields: J-type
    (imm20, rd, opcode) = ifields_J_type  instr

    -- Decode check
    is_legal = (opcode == opcode_JAL)

    -- Semantics
    pc     = mstate_pc_read  mstate
    rd_val = pc + 4

    x_u64   = zeroExtend_u32_to_u64  imm20
    y_u64   = shiftL  x_u64  1                -- offset imm20 is in multiples of 2 bytes
    z_u64   = signExtend  y_u64  21           -- sign-extend 21 lsbs
    new_pc  = cvt_s64_to_u64 ((cvt_u64_to_s64  z_u64) + (cvt_u64_to_s64  pc))
    aligned = ((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
    (is_legal, mstate1)
```

A small suite of standard "finishes". Here:
- normal: update Rd and PC
- trap: update trap CSRs and update PC to CSR MTVEC

Final result

https://github.com/rsnikhil/RISCV-ISA-Spec/src/Forvis_spec.hs

FORVIS example (all instr specs follow this "one page" style)

```
opcode_JAL = 0x6F :: InstrField    -- 7'b_11_011_11

spec_JAL :: Machine_State -> Instr -> (Bool, Machine_State)
spec_JAL    mstate       instr =
  let
    -- Instr fields: J-type
    (imm20, rd, opcode) = ifields_J_type  instr

    -- Decode check
    is_legal = (opcode == opcode_JAL)

    -- Semantics
    pc     = mstate_pc_read  mstate
    rd_val = pc + 4

    x_u64   = zeroExtend_u32_to_u64  imm20
    y_u64   = shiftL  x_u64  1                 -- offset imm20 is in multiples of 2 bytes
    z_u64   = signExtend  y_u64  21            -- sign-extend 21 lsbs
    new_pc  = cvt_s64_to_u64 ((cvt_u64_to_s64  z_u64) + (cvt_u64_to_s64  pc))
    aligned = ((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
    (is_legal, mstate1)
```

We hope that with this level of training in reading the spec, anyone will be able to read and understand all the instructions' specs, even if they have never seen Haskell code before.
(Please give us your feedback!)

https://github.com/rsnikhil/RISCV-ISA-Spec/src/Forvis_spec.hs

FORVIS Misc. comments

https://github.com/rsnikhil/RISCV-ISA-Spec

- **Instruction Fetch**: captures the following subtlety:
  - If supporting the 'C' standard extension ("compressed instructions"),
    and PC is 2 bytes away from page boundary,
    then first fetch 2 bytes and to decide whether it's a 16b 'C' instr or normal 32b instr
    and only fetch the next 2 bytes if necessary,
    to avoid a possible page-fault/memory-access trap on the latter 16b.

- **Sequential execution wrapper** (not part of the formal spec)**:**
  - Outer level Haskell function to attach semantic functions to memory and devices, and invoke loop function "run_program" that repeatedly calls fetch and execute
    - The whole spec can be compiled and run as a Haskell program *(please ask for demo)*.
  - Simulation artifacts:
    - Stop on desired instruction limit or detection of self-loop or write to <tohost> location ...
    - Dump instruction trace
    - Relay terminal input and output to Machine State

- **Concurrent execution wrapper** (not yet available; targeting December 2018):
  - Machine state is a 'tree of instruction instances'
  - Instruction children are all possible next-instructions (fall-through, branch, trap, speculation, …)
    - Sequential case is special case, where each instr has exactly one child
  - Tree-growth is concurrent with instruction execution (models pipelining, speculation)
  - Underlying concurrent dataflow from producers to consumers (models pipelining, out-of-order execution)

Conclusion: Status and Plans

Please see Niraj during the breaks or at the Bluespec booth for a demo of Forvis executing RISC-V programs.

- **By Aug 31**:
  - Complete core functionality:
    - Base ISAs: RV32I and RV64I (including simultaneous support for both in a single implementation)
    - Standard extensions: M (int mul/div), A (atomics), FD (floating pt), C (compressed)
    - Privilege specs M, S, U
      - For S: Virtual Memory schemes Sv32, Sv39, Sv48
    - Major milestone: boot Linux kernel
      - SRI/U.Cambridge (SAIL) model has already achieved this.
      - MIT (Haskell) and Bluespec (Haskell) models are close to achieving this.
  - Deliver this to Compliance Group for use as "Golden Reference Model".
- **Fall 2018**:
  - Complete parameterization by all implementation choices.
  - General cleanup and documentation, "delivery" to RISC-V Foundation.
  - Final report on sequential spec at December RISC-V Summit.
- **Fall 2018 and beyond**:
  - Work on concurrent version of specs (SRI/U.Cambridge SAIL model already started)
  - Merge ISA Formal Spec Group with Memory Model Group and integrate the two models