Formal semantics of RISCV

(Joint work with the members of the formal committee, Ian Clester, Samuel Gruetter, Andrew Wright and Adam Chlipala)

Thomas Bourgeat

MIT

7th May, 2018
What do we mean by formal spec?

- **spec**: Readable by humans that would like to work with RISCV.

- **formal**: Interpretation of the spec should be unique.

- **machine checkable**: Usable in proof assistants to prove the semantics of software. Or to prove the hardware by proving that anything that can happen in an hardware implementation could show up in the spec.
What do we mean by formal spec?

- *spec*: Readable by humans that would like to work with RISC-V.
- *formal*: Interpretation of the spec should be unique.
What do we mean by formal spec?

- **spec**: Readable by humans that would like to work with RISCV.
- **formal**: Interpretation of the spec should be unique.
- **machine checkable**: Usable in proof assistants to prove the semantics of software.
What do we mean by formal spec?

- **spec**: Readable by humans that would like to work with RISC-V.
- **formal**: Interpretation of the spec should be unique.
- **machine checkable**: Usable in proof assistants to prove the semantics of software. Or to prove the hardware by proving that anything that can happen in an hardware implementation could show up in the spec.
Hopes for our formal spec

- It should be possible to execute it.

- The maximum of it should be usable as documentation for a software or hardware developper.

- The SPEC should be written in a style usable by various proof assistants.

- It should be possible to use the formal spec to do some form of model checking.

- It should be usable by an undergrad student literate in formal methods: the spec should be modular. At least as modular as the extension mecanism of RISCV.
Hopes for our formal spec

- It should be possible to execute it.
- The maximum of it should be usable as documentation for a software or hardware developer.
Hopes for our formal spec

- It should be possible to execute it.
- The maximum of it should be usable as documentation for a software or hardware developer.
- The SPEC should be written in a style usable by various proof assistants.
Hopes for our formal spec

- It should be possible to execute it.
- The maximum of it should be usable as documentation for a software or hardware developer.
- The SPEC should be written in a style usable by various proof assistants.
- It should be possible to use the formal spec to do some form of model checking.
Hopes for our formal spec

- It should be possible to execute it.
- The maximum of it should be usable as documentation for a software or hardware developer.
- The SPEC should be written in a style usable by various proof assistants.
- It should be possible to use the formal spec to do some form of model checking.
- It should be usable by an undergrad student literate in formal methods: the spec should be modular. At least as modular as the extension mechanism of RISCV.
Users often want to be able to abstract over some level of details.

- There are different versions of the ISA RV32I, RV64G, ...
Users often want to be able to abstract over some level of details.

- There are different versions of the ISA RV32I, RV64G, . . .
- Are we doing misaligned accesses in HW or SW?
Users often want to be able to abstract over some level of details.

- There are different versions of the ISA RV32I, RV64G, ... 
- Are we doing misaligned accesses in HW or SW? *From the point of view of the usermode software writer, the behavior is fully specified.*
Users often want to be able to abstract over some level of details.

- There are different versions of the ISA RV32I, RV64G, ...
- Are we doing misaligned accesses in HW or SW? *From the point of view of the usermode software writer, the behavior is fully specified.*
- "Write Any Read Legal" CSRs: what happens when one writes an illegal value?
Users often want to be able to abstract over some level of details.

- There are different versions of the ISA RV32I, RV64G, . . .
- Are we doing misaligned accesses in HW or SW? *From the point of view of the usermode software writer, the behavior is fully specified.*
- "Write Any Read Legal" CSRs: what happens when one writes an illegal value? *Really?*
Single-core memory model problems: when the TLB has stalled values, or when the ICache holds stalled instructions modified by a store earlier. *This is real non-determinism.*
Single-core memory model problems: when the TLB has stalled values, or when the ICache holds stalled instructions modified by a store earlier. *This is real non-determinism.* *Though the determinism is fairly easy to restore using a single fence instruction.*
The memory model problems: The new level of abstraction is at an higher level (e.g mutual exclusion for locks, software transactional memory, c++11 atomics etc...).
The memory model problems: The new level of abstraction is at an higher level (e.g mutual exclusion for locks, software transactional memory, c++11 atomics etc...).

▶ The modeling of non-determinism becomes a research subject on its own.
Ongoing effort:

- https://github.com/cliffordwolf/riscv-formal by Clifford Wolf in SystemVerilog
- https://github.com/rsnikhil/RISCV_ISA_Formal_Spec_in_BSV by Rishiyur Nikhil in BSV.
- Cambridge also developing a model in Sail.
MIT’s contribution to this effort

- Written in Haskell in a monadic style.
MIT’s contribution to this effort

- Written in Haskell in a monadic style.
- The original point is that the following is left unimplemented:
  - getRegister
  - setRegister
  - loadWord
  - storeWord
  - getCSRField
  - getPC
  - ...

We will show how several implementation of those functions allows us to achieve different objectives discussed earlier.
MIT’s contribution to this effort

- Written in Haskell in a monadic style.
- The original point is that the following is left unimplemented:
  - getRegister
  - setRegister
  - loadWord
  - storeWord
  - getCSRField
  - getPC
  - ...

- We will show how several implementation of those functions allows us to achieve different objectives discussed earlier.
The code we present here

- The backbone code independent of the implementations of those functions.
- We define Minimal32/64
- Add putChar getChar
- Add TLB
Current state

Let’s take a look at the code!
Execute

```haskell
execute (lui rd imm20) = do
    setRegister rd (fromImm imm20)
execute (auipc rd oimm20) = do
    pc <- getPC
    setRegister rd (fromImm oimm20 + pc)
execute (jal rd jimm20) = do
    pc <- getPC
    let newPC = pc + (fromImm jimm20)
    if (mod newPC 4 /= 0)
        then raiseException 0 0
    else (do
            setRegister rd (pc + 4)
            setPC newPC)
execute (jalr rd rs1 oimm12) = do
    x <- getRegister rs1
    pc <- getPC
    let newPC = (x + fromImm oimm12) .&. (complement 1)
    if (mod newPC 4 /= 0)
        then raiseException 0 0
    else (do
            setRegister rd (pc + 4)
            setPC newPC)
execute (beq rs1 rs2 sbimm12) = do
    x <- getRegister rs1
    y <- getRegister rs2
    pc <- getPC
    when (x == y) (do
            let newPC = (pc + fromImm sbimm12)
            if (mod newPC 4 /= 0)
                then raiseException 0 0
            else setPC newPC)
execute (bne rs1 rs2 sbimm12) = do
    x <- getRegister rs1
    y <- getRegister rs2
    pc <- getPC
    when (x /= y) (do
            let addr = (pc + fromImm sbimm12)
            if (mod addr 4 /= 0)
                then raiseException 0 0
            else setPC addr)
execute (blt rs1 rs2 sbimm12) = do
    x <- getRegister rs1
    y <- getRegister rs2
    pc <- getPC
    when (x < y) (do
            let addr = (pc + fromImm sbimm12)
            if (mod addr 4 /= 0)
                then raiseException 0 0
            else setPC addr)
```
type Tlb = (Map.Map MachineInt (MachineInt, Int))

instance (RiscvProgram s t, MachineWidth t) => RiscvProgram (TlbState s) t where
  getRegister r = lift (getRegister r)
  setRegister r v = lift (setRegister r v)
  loadByte a = lift (loadByte a)
  loadHalf a = lift (loadHalf a)
  loadWord addr = lift (loadWord addr)
  loadDouble a v = lift (loadDouble a)
  storeByte a v = lift (storeByte a v)
  storeHalf a v = lift (storeHalf a v)
  storeWord addr val = lift (storeWord addr val)
  storeDouble a v = lift (storeDouble a v)
  getCSRField f = lift (getCSRField f)
  setCSRField f v = lift (setCSRField f v)
  getPC = lift getPC
  setPC v = lift (setPC v)
  getPrivMode = lift getPrivMode
  setPrivMode v = lift (setPrivMode v)
  commit = lift commit
  endCycle = lift endCycle

inTlb accessType a = do
  mode <- fmap (getMode (getCSRField Field.MODE))
  tblLevels <- get
  let founds = fmap (\idx tblevel -> Map.lookup (getVPN mode a (idx+2)) tblevel) tblLevels
      return . listToMaybe . catMaybes $ fmap (\idx found -> case found of
        Just (pte,level) ->
          let abit = testBit pte 6
              dbit = testBit pte 7
          in
            if (abit || (accessType == Store && ~ dbit))
              then Nothing
              else Just $ translateHelper mode a pte level
        Nothing -> Nothing) founds

  addTlb addr pte level = do
    mode <- fmap (getMode (getCSRField Field.MODE))
    tblLevels <- get
    put . fmap (\idx tblevel ->
      if (idx == level -1)
        then Map.Insert (getVPN mode addr level) (pte, level) tblevel
        else tblevel
      ) $ tblLevels
    FlushTlb = put (Map.\, Map.\, Map.\, Map.\)

---

tlb.hs Bot L68 Git:clash099 (Haskell company ElDoc)
Embedding the spec in a proof assistant!

Let's take a look at
https://github.com/samuelgruetter/riscv-coq
Execute in Coq

| Decode.Andi rd rs1 imm12 =>
| Bind (getRegister rs1) (λ x => setRegister rd (and x (fromImm imm12)))
| Decode.Slli rd rs1 shamt6 =>
| Bind (getRegister rs1) (λ x => setRegister rd (sll x shamt6))
| Decode.Srli rd rs1 shamt6 =>
| Bind (getRegister rs1) (λ x => setRegister rd (srli x shamt6))
| Decode.Sral rd rs1 shamt6 =>
| Bind (getRegister rs1) (λ x => setRegister rd (sra x shamt6))
| Decode.Add rd rs1 rs2 =>
| Bind (getRegister rs1) (λ x =>
  Bind (getRegister rs2) (λ y => setRegister rd (x + y)))
| Decode.Sub rd rs1 rs2 =>
| Bind (getRegister rs1) (λ x =>
  Bind (getRegister rs2) (λ y => setRegister rd (x - y)))
| Decode.Sll rd rs1 rs2 =>
| Bind (getRegister rs1) (λ x =>
  Bind (getRegister rs2) (λ y => setRegister rd (sll x (regToShamt y))))
| Decode.Slt rd rs1 rs2 =>
| Bind (getRegister rs1) (λ x =>
  Bind (getRegister rs2) (λ y =>
    setRegister rd (if x < y : B then one else zero)))
| Decode.Sltu rd rs1 rs2 =>
| Bind (getRegister rs1) (λ x =>
  Bind (getRegister rs2) (λ y =>
    setRegister rd (if (ltu x y) : B then one else zero)))
| Decode.Xor rd rs1 rs2 =>
| Bind (getRegister rs1) (λ x =>
  Bind (getRegister rs2) (λ y => setRegister rd (xor x y)))
| Decode.Or rd rs1 rs2 =>
| Bind (getRegister rs1) (λ x =>
  Bind (getRegister rs2) (λ y => setRegister rd (or x y)))
| Decode.Srl rd rs1 rs2 =>
| Bind (getRegister rs1) (λ x =>
  Bind (getRegister rs2) (λ y => setRegister rd (srl x (regToShamt y))))
| Decode.Sra rd rs1 rs2 =>
| Bind (getRegister rs1) (λ x =>
  Bind (getRegister rs2) (λ y => setRegister rd (sra x (regToShamt y))))
| Decode.And rd rs1 rs2 =>
| Bind (getRegister rs1) (λ x =>
  Bind (getRegister rs2) (λ y => setRegister rd (and x y)))

*** ExecuteI.v *** 69% L166 Git-master (Coq company dependency yas hs Outl Holes company)
An other benefits of being in a proof assistant
Instrumenting the spec as a circuit

- Clash is a compiler allowing one to compile haskell to circuit.

- We can craft a special implementation, a function that describes the update function with a simple memory interface (100 lines of code and 100s later we get verilog for the combinational function).

- Inputs: register file, instruction, pc, data from load if any.

- Outputs: new register file, new pc, store address and store data if any, load address if any.

- Model checked by Clifford Wolf (Yosys-SMTBMC), against riscv-formal!

- Note: CSR file not hooked up yet in this implementation.
Instrumenting the spec as a circuit

- Clash is a compiler allowing one to compile haskell to circuit.
- We can craft a special implementation, a function that describes the update function with a simple memory interface (100 lines of code and 100s later we get verilog for the combinational function)
Instrumenting the spec as a circuit

- Clash is a compiler allowing one to compile haskell to circuit.
- We can craft a special implementation, a function that describes the update function with a simple memory interface (100 lines of code and 100s later we get verilog for the combinatinal function)
  - Inputs: register file, instruction, pc, data from load if any.
  - Outputs: new register file, new pc, store address and store data if any, load address if any.
Instrumenting the spec as a circuit

- Clash is a compiler allowing one to compile haskell to circuit.
- We can craft a special implementation, a function that describes the update function with a simple memory interface (100 lines of code and 100s later we get verilog for the combinational function)
  - Inputs: register file, instruction, pc, data from load if any.
  - Outputs: new register file, new pc, store address and store data if any, load address if any.
- Model checked by Clifford Wolf (Yosys-SMTBMC), against riscv-formal!
Instrumenting the spec as a circuit

- Clash is a compiler allowing one to compile haskell to circuit.
- We can craft a special implementation, a function that describes the update function with a simple memory interface (100 lines of code and 100s later we get verilog for the combinational function)
  - Inputs: register file, instruction, pc, data from load if any.
  - Outputs: new register file, new pc, store address and store data if any, load address if any.
- Model checked by Clifford Wolf (Yosys-SMTBMC), against riscv-formal!
- Note: CSR file not hooked up yet in this implementation.
Some fun with this spec

- Instead of fetching the next instruction from memory we could hardcode a program:
Some fun with this spec

- Instead of fetching the next instruction from memory we could hardcode a program:
- We can design a riscv machine partially evaluated on a program (written in ASM) that computes a GCD.
Some fun with this spec

- Instead of fetching the next instruction from memory we could hardcode a program:
- We can design a riscv machine partially evaluated on a program (written in ASM) that computes a GCD.
module GcdExpr where
import Prelude

gcdState addr = if (addr ≡ 0) then 17719
  else if (addr ≡ 4) then 17847
  else if (addr ≡ 8) then 4556179
  else if (addr ≡ 12) then 337155
  else if (addr ≡ 16) then 370051
  else if (addr ≡ 20) then 20971759
  else if (addr ≡ 24) then 17847
  else if (addr ≡ 28) then 8750483
  else if (addr ≡ 32) then 10854435
  else if (addr ≡ 36) then 4259311727
  else if (addr ≡ 40) then 33882723
  else if (addr ≡ 44) then 33915491
  else if (addr ≡ 48) then 10877539
  else if (addr ≡ 52) then 361107
  else if (addr ≡ 56) then 329107
  else if (addr ≡ 60) then 165139
  else if (addr ≡ 64) then 4276088943
  else if (addr ≡ 68) then 1084589491
  else if (addr ≡ 72) then 4267700335
  else if (addr ≡ 76) then 361747
  else if (addr ≡ 80) then 32871
  else if (addr ≡ 84) then 0
else 0
This is not a serious way to design hardware. Though the fact that this is possible is an interesting datapoint regarding possible usage of this spec.
Future work:

- More complete version of the spec.
- Hook-up CSRs for model-checking
- Write an implementation that generates ISA constraints, that used in tandem with the axiomatic memory model will allows one to generate all legal behavior of an ASM snippet (*first loop free, without VM and selfmodifying code*)
Thank you!

Questions?

▶ You can check the code at:
   github.com/mit-plv/riscv-semantics/tree/clash099