End-to-end formal ISA verification of RISC-V processors with riscv-formal

Clifford Wolf

About RISC-V

- RISC-V is an Open Instruction Set Architecture (ISA)
- Can be freely used for any purpose
- Many implementations are available
  - from many different vendors
  - using different micro architectures
  - and under all kinds of licenses (commercial and free).

- Compatibility between those cores is key
- And so is having bug-free implementations
- riscv-formal can help achieve both of those goals

https://riscv.org/

About hardware model checking

- Hardware model checking is defined as
  - Given a circuit, a set of initial states, and assumptions, are ...
    - certain (bad) states reachable? (safety properties)
    - certain (good) states bound to be reached? (liveness properties)
  - In this presentation we focus on checking of safety properties.
  - Hardware model checking is part of the larger field of formal hardware verification

- Bounded and unbounded safety checking
  - Bounded methods: Only consider traces of up to a maximum length.
  - Unbounded methods: Consider an unlimited number of steps.
  - In this presentation we focus on bounded methods (aka “bug hunting”)

- Tools for model checking of HDL designs
  - There are a few prohibitively priced commercial tools.
  - By default riscv-formal uses the FOSS SymbiYosys flow.

http://symbiyosys.readthedocs.io/

End-to-end Verification

- Historically most checkable safety properties have been small, fine-grain properties of a system.
- It is up to the verification engineer to make sure that the combined properties ensure the overall correct functionality of a system.
- In end-to-end verification we describe the desired overall functionality directly.

- Pro: This makes the description of the desired behavior portable between implementations.
- Pro: This also makes it relatively easy to match the formal description of the desired behavior against a spec.
- Con: Proving large end-to-end properties is computationally expensive and in some cases simply impossible.

- Improvements in solver technologies in the last decade have made it possible to prove larger and more complex properties, enabling formal verification of end-to-end properties in some cases.

riscv-formal

- riscv-formal is a framework for formal end-to-end verification of RISC-V cores against the ISA spec.
- riscv-formal is not a formally verified RISC-V core! Instead it is a tool that can be used to formally verify existing cores.
- riscv-formal uses bounded methods (i.e. it's primary function is "bug hunting"), some parts generalize to complete proofs with some cores, but that isn't the primary goal.

- The following work needs to be done to integrate a new RISC-V core with riscv-formal:
  - Add the RVFI trace port to your core (usually done as optional feature)
  - Configure riscv-formal to match your core, such as:
    - set the number of RVFI channels (for superscalar cores)
    - set the correct ISA variant (like rv32i, rv32ic, rv64i, ...)
    - tell riscv-formal if the core can load/store misaligned data

https://github.com/cliffordwolf/riscv-formal

RISC-V Formal Interface (RVFI)

- RVFI is an output-only trace port for RISC-V cores.
- A core must implement RVFI to be verifiable with riscv-formal
- In addition to formal checks, riscv-formal contains a generator for (synthesizable) monitor cores that can be used with RVFI in simulation and emulation testing.
- There are RVFI draft proposals for how to support F/D/Q extensions and CSRs

Width	Port
NRET	rvfi_valid
NRET * 64	rvfi_order
NRET * ILEN	rvfi_insn
NRET	rvfi_trap
NRET	rvfi_halt
NRET	rvfi_intr
NRET * 5	rvfi_rs1_addr
NRET * 5	rvfi_rs2_addr
NRET * XLEN	rvfi_rs1_rdata
NRET * XLEN	rvfi_rs2_rdata

Width	Port
NRET * 5	rvfi_rd_addr
NRET * XLEN	rvfi_rd_wdata
NRET * XLEN	rvfi_pc_rdata
NRET * XLEN	rvfi_pc_wdata
NRET * XLEN	rvfi_mem_addr
NRET * XLEN/8	rvfi_mem_rmask
NRET * XLEN/8	rvfi_mem_wmask
NRET * XLEN	rvfi_mem_rdata
NRET * XLEN	rvfi_mem_wdata

NRET = number of RVFI channels

Verification Strategy

Two kinds of formal proofs are used to verify a RISC-V processor using riscv-formal:

(1) Instruction checks

- Prove that the retired instruction word (`rvfi_insn`) matches the reported pre/post state transition.
- This is one independent proof for each RISC-V instruction and RVFI channel.
- Using the instruction models in `riscv-formal/insns/insn_*.v`.
- Instruction checks look at all RVFI ports, but only check one time step.

(2) Consistency checks

- Prove that the sequence of state transitions is consistent. For example:
  - A register write followed by a register read must read back the previously written value.
  - The values of `rvfi_pc_wdata` and `rvfi_pc_rdata` of consecutive instructions must match.
  - Instruction reordering must be causal. It's impossible to read a value from a register before it's being written.
- Consistency checks look only at a few RVFI ports, but correlate values from different time steps.

See `riscv-formal/checks/` for Verilog source code of all checks

Anatomy of Instruction Checks

- Checker input ports:
  - RVFI Signals
  - Additional `check` input

- Bounded model check for N cycles
- Run for N cycles unconstrained (beside core reset)
- Then enable the checker for one single cycle (`check=1`)
- Individual check for each instruction and RVFI port
- Verilog defines are used to configure depth, instruction and RVFI port
- Compressed instructions and fused instructions are treated like separate instructions

- See `riscv-formal/insns/` for instruction models
- The riscv-formal instruction models are formally verified against spike (riscv-isa-sim)

Example riscv-formal instruction model (1/2)

```
// DO NOT EDIT -- auto-generated from riscv-formal/insns/generate.py

module rvfi_insn_addi (
  input                                 rvfi_valid,
  input [`RISCV_FORMAL_ILEN   - 1 : 0] rvfi_insn,
  input [`RISCV_FORMAL_XLEN   - 1 : 0] rvfi_pc_rdata,
  input [`RISCV_FORMAL_XLEN   - 1 : 0] rvfi_rs1_rdata,
  input [`RISCV_FORMAL_XLEN   - 1 : 0] rvfi_rs2_rdata,
  input [`RISCV_FORMAL_XLEN   - 1 : 0] rvfi_mem_rdata,

  output                                spec_valid,
  output                                spec_trap,
  output [                      4 : 0] spec_rs1_addr,
  output [                      4 : 0] spec_rs2_addr,
  output [                      4 : 0] spec_rd_addr,
  output [`RISCV_FORMAL_XLEN   - 1 : 0] spec_rd_wdata,
  output [`RISCV_FORMAL_XLEN   - 1 : 0] spec_pc_wdata,
  output [`RISCV_FORMAL_XLEN   - 1 : 0] spec_mem_addr,
  output [`RISCV_FORMAL_XLEN/8 - 1 : 0] spec_mem_rmask,
  output [`RISCV_FORMAL_XLEN/8 - 1 : 0] spec_mem_wmask,
  output [`RISCV_FORMAL_XLEN   - 1 : 0] spec_mem_wdata
);

...
```

Example riscv-formal instruction model (2/2)

```
...

  // I-type instruction format
  wire [`RISCV_FORMAL_ILEN-1:0] insn_padding = rvfi_insn >> 32;
  wire [`RISCV_FORMAL_XLEN-1:0] insn_imm = $signed(rvfi_insn[31:20]);
  wire [4:0] insn_rs1    = rvfi_insn[19:15];
  wire [2:0] insn_funct3 = rvfi_insn[14:12];
  wire [4:0] insn_rd     = rvfi_insn[11: 7];
  wire [6:0] insn_opcode = rvfi_insn[ 6: 0];

  // ADDI instruction
  wire [`RISCV_FORMAL_XLEN-1:0] result = rvfi_rs1_rdata + insn_imm;
  assign spec_valid = rvfi_valid && !insn_padding && insn_funct3 == 3'b 000 &&
                                                     insn_opcode == 7'b 0010011;
  assign spec_rs1_addr = insn_rs1;
  assign spec_rd_addr = insn_rd;
  assign spec_rd_wdata = spec_rd_addr ? result : 0;
  assign spec_pc_wdata = rvfi_pc_rdata + 4;

  // default assignments
  assign spec_rs2_addr = 0;
  assign spec_trap = 0;
  assign spec_mem_addr = 0;
  assign spec_mem_rmask = 0;
  assign spec_mem_wmask = 0;
  assign spec_mem_wdata = 0;
endmodule
```

Anatomy of Consistency Checks

- Checker input ports:
  - RVFI Signals
  - Additional `reset` and `check` inputs
  - Optionally an additional `trigger` input

- Bounded model check for N+M cycles
- Run for N cycles with checker in reset
- Then M cycles with checker not in reset
- In one of those M cycles `trigger` is driven high
- In the last of those M cycles `check` is driven high

- Verilog defines are used to configure N, M, and trigger position

Examples of Consistency Checks

- pc_fwd, pc_bwd
  - Make sure that `rvfi_pc_wdata` of insn K is equal to `rvfi_pc_rdata` of insn K+1 (unless insn K+1 has a high `rvfi_intr` signal)
- reg
  - Make sure that the value read from a register equals the value previously written (if any write was observed in the M cycles window)
- liveness
  - Make sure that for each insn K (retired at trig=1) there is a next insn K+1 within the M cycles window (unless insn K has a high `rvfi_halt` signal)
- unique
  - Make sure that for each insn K (retired at trig=1) there is no other insn using the same index K within the M cycles window
- causal
  - Make sure that the instruction reordering does not violate causality: When two instructions depend on each other because the first writes a value to a reg and the second reads that value from the reg, then the two instructions must be retired in-order.

What bugs can riscv-formal find?

- Hard to give a complete list, but for example
  - Incorrect single-threaded instruction semantics
  - Any bugs in bypassing/forwarding or pipeline interlock
  - Reordering gone wrong with respect to registers
  - Bugs where execution freezes (may require fairness constraints)
  - Some bugs related to memory interface and ld/st/fetch

- Bugs we can’t detect (yet :)
  - Things not covered by current RVFI (like CSRs and F/D/Q)
  - Anything related to concurrency between hearts

Processors currently supported

- PicoRV32
  - https://github.com/cliffordwolf/picorv32
  - A small RV32IMC implementation (M/C optional)
  - RVFI support enabled by `` `define RISCV_FORMAL ``
  - RV32IC variant of the core is fully verified

- RISC-V Rocket
  - RVFI branch on https://github.com/freechipsproject/rocket-chip
  - Checks small RV32I implementation by default
  - RV32IC/RV64I/RV64IC support is under construction

- VexRiscv
  - https://github.com/SpinalHDL/VexRiscv
  - A small RV32I implementation written in SpinalHDL

- See `riscv-formal/cores/` for core support scripts

Adding support for a new RISC-V processor to riscv-formal

- Adding a new core
  - Create a `riscv-formal/cores/<core-name>/` directory
  - Write a wrapper module that instantiates the core under test and abstract models of necessary peripherals (usually just memory)
  - Write a `genchecks.cfg` config file for the new core
  - See `cores/picorv32/` and `cores/rocket/` for examples

- Running the checks
  - `cd riscv-formal/cores/<core-name>/`
  - `python3 ../../checks/genchecks.py`
  - `make -C checks -j$(nproc)`

Cut-Points, Blackboxes, and other Abstractions

- Abstractions are used in formal verification to replace a complex problem with a more general simpler problem.
- The simples abstraction is cutpoints:
  - Disconnect the driver for a net, making the net unconstrained
  - Obviously this simplifies the problem: The original driver may now be optimized away.
  - The new problem is more general: If the proof succeeds that means that the properties also hold for the original problem.
- Blackboxing is like creating cut points, but for all outputs of a hierarchical entity.
- Examples for other abstractions:
  - Replace actual counter with `counter > $past(counter)` assumption
  - ALU but only one type of operation is functional and for all other operations the ALU output is unconstrained

Abstractions with riscv-formal

- Abstractions can be used extensively with riscv-formal because the large verification task is split into many smaller problems
- For each of those smaller problems other pieces of the core under test can be abstracted away

- For example consider an instruction checker:
  - The register file can be completely backboxed
  - Cutpoints can be added to bypassing paths

- And on the other hand for consistency checkers:
  - The ALU can be blackboxed for all consistency checkers
  - Some also allow blackboxing the register file and bypass paths

- None of those abstractions were possible if we’d use one single large checker for everything.

Effects of Abstractions

1.) In some cases:

- Simplifying the proof because large parts of the circuit under test can be ignored

2.) In other cases:

- Effectively increasing the BMC depth because some parts of the state space can be reached in a smaller number of cycles.

The 1st effect often only helps speeding up failing checks where a counter example must be produced. It is not uncommon for tests that pass to run with the same speed with and without abstractions in place.

The 2nd effect can cause proofs to become slower after the abstraction has been added. Therefore in some cases when an abstraction is added the BMC depth must also be reduced.

Determining ideal BMC depths

- Finding the right BMC depth setting is hard:
  - Too deep and the BMC will not complete within reasonable time.
  - Too shallow and important parts of the state space will not be reached.

- Solution #1: Use a separate formal check with SystemVerilog cover() statements to figure out what depth is necessary to include traces with certain properties. See `cover.sv in riscv-formal/cores/*/` for some examples.

- Solution #2: Add bugs to your design (one at a time) and see which BMC depth is sufficient to find them.

- In some cases it might even be necessary to combine deep BMC checks with restrictions with a shallow BMC check without restrictions in order to achieve the desired state space coverage.

Results

- So far riscv-formal has found bugs in
  - PicoRV32
  - RISC-V Rocket
  - VexRiscv
  - ISA Spec
  - Spike

- Most of them fall in one of the following categories
  - Clearing the LSB of the addition result in JALR (← single most common bug)
  - Decoding of reserved compressed instructions and hints
  - Bugs that need “weird timings” (e.g. bugs in bypassing)
  - Reset bugs

Future Work

- Support for more ISA extensions
  - Next on list: M/F/D/Q/A and RV64
  - Also support for (at least some) CSRs
  - Both require extensions to RVFI
- Support for more cores
  - But slowly, because more cores means less flexibility
  - Talk to me if you want to see your core supported
- Better integration with non-free tools (maybe :)

Thanks!

Questions?

Keywords:

- RISC-V, riscv-formal
- Yosys, SymbiYosys
- End-to-end Verification
- Hardware Model Checking
- Bug Hunting, BMC
- RISC-V Formal Interface (RVFI)
- RVFI Instruction Checks
- RVFI Consistency Checks
- RISC-V Instruction Models
- PicoRV32, RISC-V Rocket
- Cut-Points and Blackboxes
- Abstractions

Slides, relevant links and examples:

http://www.clifford.at/papers/2017/riscv-formal/