Formal Verification of RISC-V cores with riscv-formal

Clifford Wolf
CTO, Symbiotic EDA

http://www.clifford.at/papers/2018/riscv-formal/
About assertion based formal verification (formal ABV)

• Assertion based verification (ABV)
  – Uses SystemVerilog assertions to check for invariant during simulation
  – Usually used in combination with functional coverage to ensure all interesting cases are being simulated

• Formal ABV
  – Replaces simulation with formal methods
    • (This is effectively like simulating all possible traces.)
  – Formal assumptions are used to limit the scope of the traces considered
  – In case of a failure a (VCD) simulation trace is generated
  – No functional coverage is necessary because all possible traces are being considered by a formal proof
module hello (  
    input clk, rst,  
    output [3:0] cnt  
);  
    reg [3:0] cnt = 0;  
    
    always @(posedge clk) begin  
        if (rst)  
            cnt <= 0;  
        else  
            cnt <= cnt + 1;  
    end  

`ifdef FORMAL  
    always *@ assume (cnt != 10);  
    always *@ assert (cnt != 15);  
`endif  
endmodule

```plaintext
[options]
mode prove
depth 10

[engines]
smtbmc z3

[script]
read_verilog -formal hello.sv
prep -top hello

[files]
hello.sv
```
Hello World

$ sby -f hello.sby
SBY 14:45:35 [hello] Removing directory 'hello'.
SBY 14:45:35 [hello] Copy 'hello.sv' to 'hello/src/hello.sv'.
SBY 14:45:35 [hello] engine_0: smtbmc z3
...
...
...
SBY 14:45:35 [hello] engine_0.induction: finished (returncode=0)
SBY 14:45:35 [hello] engine_0: Status returned by engine for induction: PASS
SBY 14:45:36 [hello] engine_0.basecase: finished (returncode=0)
SBY 14:45:36 [hello] engine_0: Status returned by engine for basecase: PASS
SBY 14:45:36 [hello] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:00 (0)
SBY 14:45:36 [hello] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:00 (0)
SBY 14:45:36 [hello] summary: engine_0 (smtbmc z3) returned PASS for induction
SBY 14:45:36 [hello] summary: engine_0 (smtbmc z3) returned PASS for basecase
SBY 14:45:36 [hello] DONE (PASS, rc=0)
Formal ABV for safety properties: Are the bad states reachable from the initial states?

- Reachable states (implicit)
- Assertions (explicit)
- Unreachable non-bad states. Many of those usually implies a difficult proof.
Abstractions are used in formal verification to replace a complex problem with a more general simpler problem.

The simplest 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
- Multiplier that is unconstrained except $0 \times x = x \times 0 = 0$ and $1 \times x = x \times 1 = x$
Availability of various EDA tools for students, hobbyists, enthusiasts

- **FPGA Synthesis**
  - Free to use:
    - Xilinx Vivado WebPack, etc.
  - Free and Open Source:
    - Yosys + Project IceStorm
    - VTR (Odin II + VPR)

- **HDL Simulation**
  - Free to use:
    - Xilinx XSIM, etc.
  - Free and Open Source:
    - Icarus Verilog, Verilator, etc.

- **Formal Verification**
  - Free to use:
    - ???
  - Free and Open Source:
    - ???

.. and people in the industry are complaining they can't find any verification experts to hire!
About Symbiotic EDA

- We build Open Source EDA tools
  - Commercial focus on formal verification
  - But we are best known for our FPGA tool-chains

- We offer commercial versions of our tool suite
  - With SystemVerilog and VHDL support
  - We also offer trainings and commercial support

- And we create formal verification IP
  - Such as riscv-formal
HDL features in Yosys (Open Source) and Symbiotic EDA Suite (Commercial)

- **Yosys**
  - Verilog 2005
  - Memories / Arrays
  - Immediate `assert()`, `assume()`, and `cover()`
  - checkers, rand [const] regs
  - Special attributes:
    - `anyconst`, `anyseq`, `allconst`, `allseq`, `gclk`

- **Symbiotic EDA Suite**
  - Everything in Yosys
    + SystemVerilog 2012
    + VHDL 2008
    + Concurrent `assert()`, `assume()`, and `cover()`
    + SVA Properties
“Formal first” vs. traditional use of formal methods

Most formal tools are priced and advertised for the traditional use case.
Formal First $\rightarrow$ designing better
digital circuits faster and cheaper

- **Formal First** is a set of design methodologies focusing on using formal methods during development, as early as possible.
  - Target user base is design engineers, not verification engineers

- Not necessarily for creating complete correctness proofs. Instead run simple BMC for “low hanging fruits” safety properties, such as
  - standard bus interfaces like AXI/APB/etc.
  - simple data flow analysis to catch reset issues and/or pipeline interlocking problems
  - use cover() statements to replace hard-to-write one-off test benches for trying things with the design under test
    - Can be as simple as: `always @(posedge i_clk) cover(o_wb_ack);`

- Formal methods can help to find a vast range of bugs sooner and produces shorter (and thus easier to analyze) counter example traces.

- Let’s not limit our thinking to “formal is for XYZ”! Formal is a set of fairly generic technologies that have applications everywhere in the design process!
  - But we cannot unleash the full potential formal has to offer unless we make sure that every digital design and/or verification engineer has access to formal tools. (Like each of those people has access to HDL simulators.)
Formal First

- Here are a few example use cases for formal tools during the development phase of a new circuit:
  - Verification of embedded “sanity check” assertions
    - E.g. “write and read pointers never point to the same element after reset”
  - Verification of standardized interface using standardized “off-the-shelf” formal properties
    - E.g. standardized bus interfaces such as AXI.
  - Using cover statements to create test benches quickly.
    - E.g. cover “done signal goes high (some time after reset)”
  - Using cover statements during debugging to make sense of trace data from FPGA based test runs.
    - E.g. cover “done signal goes high while NAK is active”
    - Or assert “done signal never goes high while NAK is active”
  - Note that this are the same techniques that are employed in the traditional use case for formal.
  - This is similar to how simulators are used by design and verification engineers alike.
  - Nobody would claim that simulators are “only for verification (of few very special designs)”. 
About riscv-formal

- riscv-formal is a formal verification IP for RISC-V processors
  - Ongoing development, currently support RV32/64IMC
  - Current focus of development is improved support for priv spec and CSRs

- With riscv-formal we focus on bounded model check (BMC)
  - Usual depth is 10-50 cycles (depending on micro-arch)
  - Effective depth can be increased by using abstract init states

- The core under test just needs to support the riscv-formal interface (RVFI)
  - RVFI is a simple trace port that can be added easily to an existing core
  - RVFI is output-only, thus formal equivalence checks can extend a proof for the RVFI-enabled core to the version of the core without RVFI
  - riscv-formal is an end-to-end black-box approach. Any RISC-V processor that implements RVFI can be checked with riscv-formal

- riscv-formal is not simply one large formal check. Instead, it's a few 100 individual proofs, each relatively small. This yields much better performance than one large monolithic proof ever could.
Simplified anatomy of a riscv-formal check

riscv-formal is essentially a library of a few 100 such checks

Different wrapper for each core
RISC-V Formal Interface (RVFI)

- Outputs a packet for each retired instruction
  - Usually that packet is generated in the write-back stage

- Supports an arbitrary number of channels
  - Necessary for supporting superscalar cores

- Instructions can be output in an arbitrary order
  - Each packet is tagged with an instruction index (rvfi_order)
  - That instruction index must correspond to the program order

- riscv-formal works with any core that implements RVFI
RVFI Basic Signals

- Basic RVFI signals
  - output [NRET  - 1 : 0] rvfi_valid // 1 in a cycle with a packet
  - output [NRET * 64 - 1 : 0] rvfi_order // insn index in program order
  - output [NRET * ILEN - 1 : 0] rvfiInsn // instruction word
  - output [NRET  - 1 : 0] rvfi_trap // 1 if the instruction traps
  - output [NRET  - 1 : 0] rvfi_halt // 1 if the instruction may halt
  - output [NRET  - 1 : 0] rvfi_intr // 1 if first insn in intr handler
  - output [NRET * 2 - 1 : 0] rvfi_mode // 0=U, 1=S, 2=Reserved 3=M

- NRET = Number of RVFI channels
- ILEN = Maximum instruction length supported by the core (min 32)
RVFI Basic Signals

- Basic RVFI signals for program counter

  output [NRET * XLEN - 1 : 0] rvfi_pc_rdata // old program counter
  output [NRET * XLEN - 1 : 0] rvfi_pc_wdata // new program counter

- XLEN = 32 or 64

- pc_rdata = address of this instruction
- pc_wdata = address of next instruction
RVFI Basic Signals

- Basic RVFI signals for register file

  output [NRET * 5 - 1 : 0] rvfi_rs1_addr // address of rs1/rs2
  output [NRET * 5 - 1 : 0] rvfi_rs2_addr
  output [NRET * XLEN - 1 : 0] rvfi_rs1_rdata // data read from rs1/rs2
  output [NRET * XLEN - 1 : 0] rvfi_rs2_rdata
  output [NRET * 5 - 1 : 0] rvfi_rd_addr // address of rd
  output [NRET * XLEN - 1 : 0] rvfi_rd_wdata // data written to rd

- Unused fields simply use addr=0 and data=0 (consistent with x0/zero)
RVFI Basic Signals

- Basic RVFI signals for memory access

  \[
  \text{output } [\text{NRET} \times XLEN - 1 : 0] \text{ rvfi\_mem\_addr} \quad // \text{address of memory access}
  \]
  \[
  \text{output } [\text{NRET} \times XLEN/8 - 1 : 0] \text{ rvfi\_mem\_rmask} \quad // \text{byte-enable for read}
  \]
  \[
  \text{output } [\text{NRET} \times XLEN/8 - 1 : 0] \text{ rvfi\_mem\_wmask} \quad // \text{byte-enable for write}
  \]
  \[
  \text{output } [\text{NRET} \times XLEN - 1 : 0] \text{ rvfi\_mem\_rdata} \quad // \text{data read from memory}
  \]
  \[
  \text{output } [\text{NRET} \times XLEN - 1 : 0] \text{ rvfi\_mem\_wdata} \quad // \text{data written to memory}
  \]

- When the Verilog define `RISCV_FORMAL_ALIGNED_MEM` is set, `rvfi\_mem\_addr` must point to an XLEN-aligned address. Otherwise `rvfi\_mem\_addr` points directly to the accessed memory location.

- For instructions that don’t access memory, use `rmask=0` and `wmask=0`. 

RVFI Signals for CSRs

- For each (non-shadow) CSR we add 4 additional RVFI signals:
  
  \[
  \text{output} \ [\text{NRET} \times \text{XLEN} - 1 : 0] \ \text{rvfi}_\text{csr}_<\text{csrname}>_\text{rmask} \ // \ \text{bitmask: bits observed}
  \]
  
  \[
  \text{output} \ [\text{NRET} \times \text{XLEN} - 1 : 0] \ \text{rvfi}_\text{csr}_<\text{csrname}>_\text{wmask} \ // \ \text{bitmask: bits written}
  \]
  
  \[
  \text{output} \ [\text{NRET} \times \text{XLEN} - 1 : 0] \ \text{rvfi}_\text{csr}_<\text{csrname}>_\text{rdata} \ // \ \text{CSR data bits observed}
  \]
  
  \[
  \text{output} \ [\text{NRET} \times \text{XLEN} - 1 : 0] \ \text{rvfi}_\text{csr}_<\text{csrname}>_\text{wdata} \ // \ \text{CSR data bits written}
  \]

- Which CSRs are supported by the core under test is signaled using Verilog defines. For each supported CSR we define
  
  \[
  \text{RISCV}_\text{FORMAL}_\text{CSR}_<\text{CSRNAME}>
  \]

- See riscv-formal docs for details.

- Note: CSR support in riscv-formal is currently under development.
Alternative Arithmetic Operations

• Some arithmetic operations are hard to verify using black-box methods. (multiply, divide)
  – For those operations we define “alternative operations” that can be used during verification.
  – The Verilog define RISCV_FORMAL_ALTOPS is used to signal the use of those alternative operations.

• This requires providing “drop-in” replacements for the relevant Verilog modules (see for example rocket MulDiv drop-in module in `<riscv-formal>/cores/rocket/`).
  – The drop-in replacement must be an abstraction of the actual module with respect to control signals.
  – With respect to the data path the drop-in replacement must implement the “alternative operation”.

• Note that with alternative operations riscv-formal will only verify the data paths to and from the arithmetic unit. An extra proof is required to check the data path of the arithmetic unit in isolation.

• See RVFI documentation for details.
RVFI and F/D/Q ISA extensions

output [NRET * 5 - 1 : 0] rvfi_frs1_addr // register addresses
output [NRET * 5 - 1 : 0] rvfi_frs2_addr
output [NRET * 5 - 1 : 0] rvfi_frs3_addr
output [NRET * 5 - 1 : 0] rvfi_frd_addr

output [NRET - 1 : 0] rvfi_frs1_rvalid // there’s no floating point
output [NRET - 1 : 0] rvfi_frs2_rvalid // zero register, so we need
carved valid signals
output [NRET - 1 : 0] rvfi_frs3_rvalid
output [NRET - 1 : 0] rvfi_frd_wvalid

output [NRET * FLEN - 1 : 0] rvfi_frs1_rdata // data read and/or written
output [NRET * FLEN - 1 : 0] rvfi_frs2_rdata
output [NRET * FLEN - 1 : 0] rvfi_frs3_rdata
output [NRET * FLEN - 1 : 0] rvfi_frd_wdata

output [NRET * XLEN - 1 : 0] rvfi_csr_fcsr_rmask // fcsr
output [NRET * XLEN - 1 : 0] rvfi_csr_fcsr_wmask
output [NRET * XLEN - 1 : 0] rvfi_csr_fcsr_rdata
output [NRET * XLEN - 1 : 0] rvfi_csr_fcsr_wdata

Note: F/D/Q is work in progress
External AMOs

• Atomic Memory operations with rd=x0 may not actually return the old value to the core.
  − The atomic operation could be performed entirely in the external memory fabric without the core actually having knowledge of neither old nor new value.
  − Thus it would not be possible for the core to populate rvfi_mem_[rw]data correctly.

• Cores that have this issue may set RISCV_FORMAL_EXTAMO to signal that they implement the following additional RVFI signal:
  
  output [NRET - 1 : 0] rvfi_mem_extamo

• When rvfi_mem_extamo is set, rvfi_mem_wdata carries the rs2 value used with the atomic instruction instead of the new value in the memory location. rvfi_mem_rmask is all-zeros in this case.

• Note: This feature is work in progress.
Skipped Instructions

- Consider the instruction sequence on the right
  - If \( t3 \) is nonzero, the core might decide to simply skip the add instruction.
  - But the RVFI spec requires the add instruction to be retired with its correct output value \( t0 \).

```
add t0,t1,t2
beqz t3,label
sub t0,t1,t3
label:
```

- A core that can skip instructions like this can signal via RISCV_FORMAL_SKIP that it implements an addition RVFI signal:
  
  ```
  output [NRET - 1 : 0] rvfi_skip
  ```

- The register value written by an instruction with rvfi_skip active is not checked by riscv-formal.
- No non-skipped instruction may ever observe the value written by a skipped instruction.
- Note: This feature is work in progress.
Fused Instructions

• A core may retire multiple fused instructions in a single RVFI packet.
  - This is necessary if instruction fusing will hide intermediate results that become unavailable to
    the RVFI generator because of the instruction fusing.

• As far as riscv-formal is concerned those fused instructions are just longer instructions.
  - This means a core with support for instruction fusion needs to set a larger ILEN parameter.
  - For shorter (un-fused) instructions the upper (unused) bits of r$v$fi$_$insn must be set to zeros.

• Note: No core currently supported by riscv-formal uses this feature.
Verification Strategy

- riscv-formal is not one large check, it’s many small ones
  - Each check only uses some of the RVFI signals
  - Each check allows for blackboxing different parts of the core under test
  - Each check allows for different abstractions being used in the core under test
  - Thus those small checks are much faster than one large check could ever be

- There are two categories of riscv-formal checks:
  - Instructions checks
  - Consistency checks
Instruction Checks

- There is one instruction check for each RISC-V instruction and RVFI channel.
- They assume that the core retires:
  - The type of instruction the check is for.
  - On the RVFI channel the check is for.
  - In a given cycle N after reset (= bounds of check).

- They check that:
  - The instruction in rvfi_insn is consistent with.
  - The state transition described in the other RVFI signals in that RVFI packet.

- I.e. an instruction check only checks one RVFI packet on one RVFI channel in one cycle.
- Thus most of the things that hold persistent inter-instruction state, such as the register file, can be black-boxed or replaced with abstractions.
Consistency Checks

• In addition to instruction checks there is a handful of consistency checks in riscv-formal.
  − They check if the sequence of packets on the RVFI interface is internally consistent.

• For example, there is are checks to make sure that
  − a register read observes the value previously written (or read)
  − there are no instruction indices missing (rvfi_order)
  − rvfi_pc_wdata matches rvfi_pc_rdata of the next instruction, unless the next instruction has rvfi_intr set.

• i.e. consistency checks look at larger sequences of RVFI packets spread out over time, but each one of them only looks at a few of the RVFI signals
• Usually large parts of the core can be abstracted away of blackboxed for a given consistency check. The most obvious example for that would be the entire ALU.
Ex. `rvfi_pc_{fwd,bwd}_check.sv`

- Checks that
  - `rvfi_pc_wdata` in instruction $K$ equals
  - `rvfi_pc_rdata` in instruction $K+1$,
  - unless instruction $K+1$ has `rvfi_intr` set.
  - ($rvfi_order = K, K+1$)

- Remember: Instructions can be retired out of order on RVFI.
  - `rvfi_pc_fwd_check`: assumes instruction $K+1$ (for any $K$) is retired in cycle $N$ (= bounds of check),
    and asserts that a previously retired instruction $K$ has a matching `rvfi_pc_wdata`
  - `rvfi_pc_bwd_check`: assumes instruction $K$ (for any $K$) is retired in cycle $N$,
    and asserts that a previously retired instruction $K+1$ has a matching `rvfi_pc_rdata`

- We run a separate instance of this check for each RVFI channel.
  - The assumption and assertion for instruction $K+1$ (fwd) or $K$ (bwd) applies to that channel.
  - The “search” backwards for the matching instruction is always performed on all channels.
• Find the code on GitHub: https://github.com/SymbioticEDA/riscv-formal

• `<riscv-formal>/checks/`
  - Verilog code for riscv-formal checks, and also some other Verilog files

• `<riscv-formal>/insns/`
  - RISC-V ISA semantics used by instruction checks

• `<riscv-formal>/monitor/`
  - RVFI monitor core (for checking RVFI stream in simulation or FPGA-based testing)

• `<riscv-formal>/cores/<core-name>/`
  - Cores currently supported (not all are part of the public repo)

• `<riscv-formal>/tests/`
  - Additional tests to verify riscv-formal itself, for example formal verification against spike (official ISA sim, written in C++) and against the MIT RISC-V formal spec (Haskell)
Supported cores (excerpt)

- **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**
  - Full-featured RISC-V implementation
  - Version of Rocket with RVFI is not upstream yet

- **VexRiscv**
  - A small RV32I implementation written in SpinalHDL

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

$ git clone https://github.com/SymbioticEDA/riscv-formal
$ cd riscv-formal/cores/picorv32
$ cat README

$ wget -O picorv32.v https://raw.githubusercontent.com/..../picorv32.v

$ python3 .././checks/genchecks.py
  Reading checks.cfg.
  Creating checks directory.
  Generated 76 checks.

$ make -C checks -j$(nproc)

More details:
  → demo at the end of this presentation
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
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
  - Rocket
  - VexRiscv
  - RI5CY
  - (other cores)
  - ISA Spec
  - Spike

• Most of these bugs 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: F/D/Q/A
  - Support for CSRs, U-mode, S-mode

- Support for more cores
  - But slowly, because more cores mean less flexibility
  - Talk to me if you want to see your core supported

- Better integration with non-free tools (maybe :)

Formal Verification of RISC-V cores with riscv-formal

Clifford Wolf
CTO, Symbiotic EDA

http://www.clifford.at/papers/2018/riscv-formal/
OSDA – Open Source Design Automation
Friday Workshop at DATE 2019

• Topics include:
  - Open-Source Tools, IPs, Languages, and Methodologies
  - Future directions for the open-source FPGA movement
  - Discussions on licenses, funding, and commercialization

http://osda.gitlab.io
short demo