Industry Perspectives on
Open Source Hardware IP
Verification Challenges

Michael Thompson, Covrado
What is “Hardware IP”? 

- Hardware Intellectual Property
  - That’s not particularly helpful…

- For the purposes of today’s discussion, “Hardware IP” shall mean:
  - A synthesizable (RTL) description of a functional unit.
  - Typically has well defined interfaces that follow an industry standard protocol.
  - Examples include a PCI-Express end-point, embedded microprocessor core or DRAM controller.
Why Use Hardware IP?

- Industrial teams use Hardware IP to reduce costs:
  - Documentation, Design, Verification, Synthesis, Place&Route and Timing Closure effort can all be reduced.

- None of these costs can be eliminated by using Hardware IP.

- Verification remains a thorny issue:
  - How to assess the quality of the Hardware IP?
  - Must still verify the usage of the Hardware IP within the host ASIC/FPGA.
Use of Hardware IP in Industry

- Use of Hardware IP is now well established in Industry:
  - Multiple vendors selling “silicon proven” IP.
  - ASIC and FPGA development teams have come to trust third-party IP.

- Successful teams understand that using IP reduces and changes development effort, it does not eliminate it.
  - Verification issues persists.

- Hardware IP vendors have learned how to support their customer’s verification:
  - Quality Documentation.
  - On-site support engineers.
  - Strict control of the implementation: virtually all commercial IP is “closed source”.
Why Use Open Source Hardware IP?

- Reduce Costs:
  - This is the primary reason to use *any* IP, open source or not.

- Take advantage of an ‘eco-system’ to reduce barriers to entry:
  - Access to engineering talent with relevant skills.
  - Hardware and software tools.

- Ability to make custom modifications:
  - This is unique to Open Source.

- Desire to contribute to the Open Source Community:
  - Not a strong motivation in Industry.
Industry use of Open Source Hardware IP

- Use of Open Source Hardware IP is not well established in Industry:
  - This is in contrast to Open Source Software which is widely used in Industry.

- Barriers to Adoption:
  - Perceived poor quality documentation.
  - Lack of direct vendor support tends to shrink the cost/benefit of using IP.
  - Perceived low quality verification.
The Verification Problem

- Its helpful to think of them as *challenges*, not *problems*.
  - All designs have functional defects (bugs).
  - It is impossible to know where the bugs are apriori.
  - No known method of demonstrating that all bugs have been found.
  - It cost more to find and fix the bugs than it does to create them.

- Gotta get it right the first time.
  - Schedule impact: an ASIC re-spin takes weeks to months.
  - Dollar cost of an ASIC re-spin is extreme.
The Cost of Bugs

- There is lots of data that shows software bug costs increase over the life-cycle.

- Hardware has a similar curve.
  - Absolute costs of bugs are much higher for hardware.
  - Cost of an ASIC re-spin can easily be $> 10^6$ dollars.
State of the Verification in Industry Today

- No known method to find all bugs.

- Multiple techniques are used:
  - Simulation, emulation, prototyping, formal methods, code reviews.

- Multiple techniques to measure verification completeness:
  - Compliance test-plans, code coverage, functional coverage.
Open Source Hardware has specific Challenges

- Two of the (many) reasons for the success of Open Source Software:
  - The cost of finding defects is spread across a wide developer base:
    - How many eyeballs have looked at the source code for gcc?
  - The cost of fixing defects is acceptable. Produces a large number of high-quality open source software projects.

- Open Source Hardware is unique:
  - Much smaller developer base. Reduces the quantity and quality of open source hardware projects.
  - Much higher costs to fix bugs.
In a Nutshell

- Verification is not a solved problem.
  - An Open Source Hardware IP provider needs to recognize this and take it seriously.

- The cost of defects in silicon means that Open Source Hardware IP must support “first time right” designs.
  - Industrial teams demand the same quality of Open Source Hardware IP as commercially available (typically closed source) Hardware IP.

- Open Source Hardware IP must address its unique situation and provide:
  - Quality documentation.
  - A support model for Open Source Hardware IP.
  - Industrial-grade verification completion metrics.
  - A method for users to re-verify any IP they modify.
Are We There Yet?!?

- The RISC-V community has a large set of Cores readily available:
  - Several, including RI5CY are already in silicon.
  - Several Industrial teams already have RISC-V based hardware available.

- As far as I am aware, none of the RISC-V cores available to the Open Source community are fully verified and “production-ready”:
  - RI5CY has known bugs (see GitHub Issue #132).
  - Completeness of RISC-V verification efforts is not known:
    - The above RI5CY bug is not attributed to any specific version of RI5CY!
Getting There From Here

- Start with a minimal set of Open Source Hardware IP:
  - The OpenHW Group will focus on RI5CY and Ariane cores first.
  - Execute a ‘complete’ verification effort on these cores, fix the bugs and make the updates available to the community.

- Verification Environment(s) should be developed with both Industry and Open Source requirements in mind:
  - Use of Industrial-grade tools and methodologies.
  - Ensure all verification code (“VIP”) is Open Source.
  - Specific emphasis on support for user-verification and user-modification.

- Strict Configuration management and Revision Control is essential:
  - Must have the ability to cross references issues to versions.
A RISC-V Open Source Simulation Environment

- The simulation environment is written in SystemVerilog and uses the Universal Verification Methodology.

- Constrained Random generation of Instructions using the Google Instruction Stream Generator:
  - Implemented in SystemVerilog using the Universal Verification Methodology library.
  - Search for “RISC-V Processor Verification Platform”.

- Reference Model based on the Imperas Instruction Set Simulator (implemented in C).

- Checking and functional coverage implemented in SystemVerilog.
In Place Today:
A RISC-V Hardware IP Open Source Testbench

Google Cloud

Open Source SystemVerilog
UVM
RISC-V
Instruction
Stream
Generator

GCC/LLVM

RISC-V.c

RISCV.elf

RTL Simulation

RISC-V RTL & memory

Imperas ISS (cpu+memory)

Metrics.log

compare

Imperas.log

Image source: © Imperas Software Ltd
Constrained Random Generation

- The effort required to write a set of tests to fully verify a modern IP unit is prohibitive:
  - Requires a large set of tests.
  - Small changes to RTL will impact a large set of tests.

- Current practise is to define a set of rules (constraints) that defines all legal (and illegal) input stimulus:
  - Significant reduction in the number of tests to write.
  - Small changes to RTL will impact a small set of constraints.

- Constrained random generation of stimulus is well supported by multiple HDLs such as SystemVerilog and SystemC.
Constrained Random Instruction Generation

- Not a new idea:
Constrained Random Instruction Generation 101

- First, define a randomizable “Instruction Class”:

```verilog
class riscv_instr_base extends uvm_object;

rand riscv_instr_group_t group;
rand riscv_instr_format_t format;
rand riscv_instr_category_t category;
rand riscv_instr_name_t instr_name;
rand bit [11:0] csr;
rand riscv_reg_t rs2;
rand riscv_reg_t rs1;
rand riscv_reg_t rd;
rand bit [31:0] imm;
rand imm_t imm_type;
rand bit [4:0] imm_len;
r rand bit is_pseudo_instr;
rand bit aq;
r rand bit rl;
```

- Add constraints to define the rules for a legal instruction:

```verilog
class riscv_rand_instr extends riscv_instr_base;

riscv_instr_gen_config cfg;
// Some additional reserved registers
riscv_reg_t reserved_rd[];

// uvm_object_utils(riscv_rand_instr)
constraint category@c {
  soft category inside (LOAD, STORE, SHIFT, ARITHMETIC, LOGICAL, BRANCH, COMPARE, CSR, SYSTEM, SYNCH);
}

constraint instr@c {
  solve instr name before imm;
  solve instr name before rs1;
  solve instr name before rs2;
  instr name inside (riscv_instr_pkg::unsupported_instr);
  group inside (riscv_instr_pkg::supported_isa);
  // Avoid using any special purpose register as rd, those registers are reserved for special instructions
  rd inside (cfg reserved regs);
  if(reserved rd size) > 0 {
    rd inside (reserved rd);
  }
  // Compressed instruction may use the same CSR for both rs1 and rd
  if(group inside (RV32C, RV64C, RV128C, RV32FC, RV32HC)) {
    rd inside (cfg reserved regs reserved rd);
  }
  // Below instructions will modify stack pointer, not allowed in normal instruction stream.
  // It can be used in stack operation instruction stream.
  instr name inside (C NOP, C NOP, C ADDI8SP);
  // Avoid using reserved registers as rs1 (base address)
  if(category inside (LOAD, STORE)) {
    rs1 inside (reserved rd, cfg reserved regs, ZERO);
  }
  if(cfg enable sfence) {
    instr name in SFENCE_VMA;
  }
  if(cfg no fence) {
    instr name inside (FENCE, FENCEI, SFENCE_VMA);
  }
  // Add Support C ADDI4SPN
  instr name in C ADDI4SPN;
}
```
Other Verification Environments

- **Formal Verification:**
  - Top-level interfaces and internal pipeline are both candidates for formal methods.
  - Again, all code for the formal environment must be written in SystemVerilog and will be Open Source.

- **Prototyping:**
  - A critical technique for system-level verification.
  - How to publish a lab prototype as an Open Source project?
Supporting Open Source Verification

- Everything on the ‘Testbench’ slide is implemented as Open Source code using non-proprietary languages:
  - SystemVerilog is IEEE-1800.
  - UVM is IEEE-1800.2
  - C is as Open Source as it gets.

- The EDA tools *are* proprietary...
  - These are the tools used by Industry.
  - The environment is not locked into any single EDA vendor’s tools.
Lots of Work Remaining...

- Complete a RISC-V Compliance specification for selected Cores.
  - Leverage the work being done by the RISC-V Foundation Compliance Task Force.

- Functional Coverage model for the Compliance spec.
  - Write it!
  - Close it!

- Integrate all components into a complete end-to-end UVM verification environment.

- Prototype verification flow for User extensions.

- Documentation!
Wrapping it Up

- Open Source Hardware IP seems to be an idea who’s time has come.

- To be successful, Open Source Hardware IP must address the concerns of Industrial development teams:
  - Cost and Quality.

- Unique Open Source Hardware IP capabilities could be game-changing:
  - Access to community knowledge, experience and tools.
  - Ability to produce open or closed source variants.
Thank You!
Background
RI5CY Issues
32-bit core

Hetalani Salaheddin, 4th Sep 2019
MPP of MSTATUS CSR written wrongly

Github issue #132

- While the core is running in U-mode
- LW instruction accesses an address where reads are not allowed is executed - raises load access fault (LAF) exception
- Followed by a fetch from an address that is outside the instruction match region - raises instruction access fault (IAF) exception
- The IAF is served before the LAF and the privilege level changes to M-mode again
- When the LAF is handled, all associated CSRs are updated correctly!
- Except MSTATUS, where the MPP field sets the privilege to M-mode
- Thus, the preceding LAF thinks it was executed in M-mode while it was in U-mode.
MPP of MSTATUS CSR written wrongly
Github issue #132

- IAF – Instruction access fault
- LAF – Load access fault

LW instruction causing LAF in U mode
Instruction fetch causing an IAF
Privilege moves to M mode and IAF is served
LAF is handled where MPP of MSTATUS CSR written with M mode