Verification of a RISC-V system with multiple cores

Oscar Palomar[1], Roberto Ignacio Genovese[1], Iván Díaz Ortega[1], Albert Aguiler[1], Abdul Rauf[2]

[1]Barcelona Supercomputing Center

[2]ARM

Abstract

Verifying complex RISC-V systems, particularly out-of-order cores and their cache hierarchies, presents significant challenges due to the need for thorough functional coverage and reference model validation. This paper presents the verification strategy for a RISC-V system composed of multiple cores and its cache hierarchy. We have implemented a stand-alone UVM environment at the core-level that is then reused at the system-level testbench. Our approach employs a modified version of Spike ISS as a reference model to ensure correct execution and validation of test cases. We use multiple directed tests (riscv-tests, compliance), along with randomly generated binaries using riscv-dv. This work contributes a reusable UVM testbench, an enhanced reference model, and a set of directed and randomized tests that significantly improve coverage and bug detection in RISC-V system verification.

Introduction

Faced with the challenge to verify a complex RISC-V design, as part of the BZL[1] (Barcelona Zettascale Lab) project, we have developed multiple environments and extended existing RISC-V tools to suit our needs.

The methodology we follow for design verification is the Universal Verification Methodology (UVM). We have created a number of UVM environments that use co-simulation (with the RISC-V instruction set simulator Spike [2]) to check the correctness of the design. We have created environments for the scalar cores and for the CPU-subsystem. In general, their specifications and interfaces are well-defined and stable, facilitating the verification work.

Leveraging the reference model, we rely heavily on constrained random testing. More specifically, we use random binary generators (mostly, riscv-dv [3]) extensively to verify the designs and uncover many bugs. Code and functional coverage is collected and analysed to identify corner cases in the random tests, to be targeted with changes in the configuration of the generator and/or directed tests.

Additionally, we implement a set of assertions (SVAs), mostly in the interfaces of selected modules.

The following sections present the environment at the core and the system-level. We then discuss the reference model and the tests, and the changes needed in tools to support the verification of the system.

Core-UVM

Core-uvm is a verification environment for RISC-V cores. The DUT is the core itself, excluding all the cache hierarchy. It incorporates Spike as reference model for co-simulation. We have support for three different designs, one in-order (Sargantana) and two out-of-order cores (Lagarto KA and Lagarto OX).

The following interfaces are used to drive or sample signals from the DUT:

- complete_if: Completed (committed/retired) instructions state. This interface has as many entries as instructions can be completed in a cycle. It includes PC of the instruction, value written in the destination register, content of multiple CSRs, etc.
- int_if: Interrupt state signals. Injects interrupt requests to the core, and detect when it traps.
- dc_if: Data cache interface signals.
- ic_if: Instruction cache interface signals.

The Icache, Dcache and Interrupt agents monitor and drive the corresponding interfaces, properly responding to the core requests/responses, through dedicated UVM agents. The Complete agent monitors the complete interface to track execution of instructions, sending them to the scoreboard, which compares them with the outcome of Spike, used as reference model. Note that in a few cases, we force the value of the DUT into Spike.

The simulation of a test typically consists in first preloading a binary into the memory model and Spike and then respond to core requests until a certain position in memory is written to signal the end of test. Early termination occurs on scoreboard mismatches, timeout (no completed instruction in a long time) or assertions. External, timer or software interrupts are randomly generated.

One of the relevant agents of the design is the instruction manager. Firstly, it unifies the instruction state captured from complete_if in time and format so that it can be processed in the environment for the multiple cores. Also, the instruction manager is in charge of filtering known errors in the design so that they don't reach the scoreboard. Furthermore, instruction manager is in charge of filtering false errors, such as time related RISC-V CSRs. This is necessary due to the different nature of simulation and RTL.

CPU-Subsystem UVM

In the next higher level, the DUT is the system with three cores (one of each kind) and three levels of cache. However, note that although possible, the cores are not meant to be used simultaneously, so it is effectively a single-core system at run-time. The interfaces of the system are two AXI-M ports (one for memory requests, another for peripherals), one AXI-S port (for DMA), interrupt inputs and a JTAG interface. The main components of the testbench are:

- Core-uvm instance configured in passive mode.
- AXI - Slave Memories to emulate main memory, SRAM and bootrom. Modelled using the axi-mem component from the Pulp project [4].
- JTAG - VIP to verify JTAG protocol and RISCV debug module.
- AXI - Master VIP to support verification of DMA bridge and be able to exchange and access different design blocks of RTL through DMA.
- PLIC / CLINT UVM agents to generate external, timer and software interrupts at the subsystem level and verify its handling by the RTL.
- UART model. An AXI crossbar from the Pulp project is used to connect both an SRAM and the UART to the peripheral bus.

Reference Model

Our basic reference model is the RISC-V instruction set simulator Spike [2]. It has been modified to provide SystemVerilog DPI calls that interact with the UVM environment and to model implementation specific details of the designs. We have implemented a configuration flag (core_type) to indicate which design to model. Core_type=Standard follows the vanilla ISA specifications and upstream spike implementation. Examples of specific behaviour are reset values of some CSRs or which vector instructions are supported. We also modified it to allow the preloading of a bootrom from an .elf file. This provides us the flexibility we need to use Spike with different environments and designs.

The main DPI call implemented is to execute the next instruction (and return the simulator's state). This is called whenever the core commits one instruction. We also have DPI calls to force results into Spike (due to a few known-bugs of floating point rounding errors, values of hpm counters, etc.), to change the external interrupt signal (in order to mimic the core's acknowledge of the interrupt), and to perform a TLB walk without triggering exceptions (used to model VIPT caches in core-uvm).

Tests

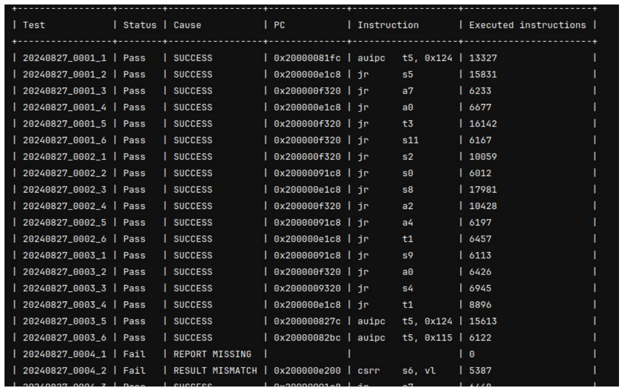
We have used a collection of regression tests composed of the riscv-isa-tests, the riscv-arch-tests (formerly compliance) and internal RVV ISA tests. Also a number of UVM directed tests for verifying the JTAG and DMA interfaces.

Nevertheless, the bulk of our tests come from the random binary generator riscv-dv [3]. We have modified it to provide full support of rvv 1.0.0, vector memory instructions generation with changing values of SEW and vector length. We have also added options to further modify the kinds of instructions to generate and provide finer control of the randomization of key registers (ra, sp, tp, etc.). Apart from this, we have fixed several issues in the code of the tool regarding aspects affecting the exception and trap handlers, while also expanding the customization of the trap delegation system.

Concerning the types of tests that we generated we can divide them in three different types:

- Basic scalar tests, which contains mainly the usual scalar instructions together with many different memory operation cases.
- Basic vector tests, similar to the basic scalar ones, but with emphasis on vector instructions and memory operations.
- Atomic instructions stress tests, which focus on generating randomized atomic instructions.

Figure below shows an example of a run with multiple randomly generated binaries. All binaries are generated with a similar amount of static instructions but the execution of exceptions and interrupts leads significant differences in the number of executed instructions.



```
+-----------------+--------+-----------------+--------------+------------------+----------------------+
| Test            | Status | Cause           | PC           | Instruction      | Executed instructions |
+-----------------+--------+-----------------+--------------+------------------+----------------------+
| 20240827_0001_1 | Pass   | SUCCESS         | 0x20000081fc | auipc   t5, 0x124 | 13327                |
| 20240827_0001_2 | Pass   | SUCCESS         | 0x200000e1c8 | jr      s5       | 15831                |
| 20240827_0001_3 | Pass   | SUCCESS         | 0x200000f320 | jr      a7       | 6233                 |
| 20240827_0001_4 | Pass   | SUCCESS         | 0x200000e1c8 | jr      a0       | 6677                 |
| 20240827_0001_5 | Pass   | SUCCESS         | 0x200000f320 | jr      t3       | 16142                |
| 20240827_0001_6 | Pass   | SUCCESS         | 0x200000f320 | jr      s11      | 6167                 |
| 20240827_0002_1 | Pass   | SUCCESS         | 0x200000f320 | jr      s2       | 10059                |
| 20240827_0002_2 | Pass   | SUCCESS         | 0x20000091c8 | jr      s0       | 6012                 |
| 20240827_0002_3 | Pass   | SUCCESS         | 0x200000e1c8 | jr      s8       | 17981                |
| 20240827_0002_4 | Pass   | SUCCESS         | 0x200000f320 | jr      a2       | 10428                |
| 20240827_0002_5 | Pass   | SUCCESS         | 0x20000091c8 | jr      a4       | 6197                 |
| 20240827_0002_6 | Pass   | SUCCESS         | 0x200000e1c8 | jr      t1       | 6457                 |
| 20240827_0003_1 | Pass   | SUCCESS         | 0x20000091c8 | jr      s9       | 6113                 |
| 20240827_0003_2 | Pass   | SUCCESS         | 0x200000f320 | jr      a0       | 6426                 |
| 20240827_0003_3 | Pass   | SUCCESS         | 0x2000009320 | jr      s4       | 6945                 |
| 20240827_0003_4 | Pass   | SUCCESS         | 0x200000e1c8 | jr      t1       | 8896                 |
| 20240827_0003_5 | Pass   | SUCCESS         | 0x200000827c | auipc   t5, 0x124 | 15613                |
| 20240827_0003_6 | Pass   | SUCCESS         | 0x20000082bc | auipc   t5, 0x115 | 6122                 |
| 20240827_0004_1 | Fail   | REPORT MISSING  |              |                  | 0                    |
| 20240827_0004_2 | Fail   | RESULT MISMATCH | 0x200000e200 | csrr    s6, vl   | 5387                 |
```

Our initial set of regression jobs used riscv-tests [5] and a few other directed tests proved to be necessary but not sufficient to detect buggy updates in the code, that would cause a high number of failing tests in the following runs. As a solution, we created a batch of random tests to complement the regression. The initial batch was later improved by selecting a handful of random tests that had uncovered difficult bugs. This approach ended up being much robust.

Conclusions

We have managed to verify a complex design with multiple cores and a cache hierarchy (currently under fabrication). Careful design and implementation of the core UVM has allowed using it for multiple cores, and reuse it in the system level. We have benefitted greatly of tools like Spike, riscv-dv or riscv-tests, adapting them as needed. We plan to contribute the changes that are not only relevant for our design, but may be beneficial to the community.

Acknowledgements

This project is promoted by the Ministry for Digital Transformation and the Civil Service, within the framework of the Recovery, Transformation and Resilience Plan - Funded by the European Union - NextGenerationEU.

References

[1] Barcelona Zettascale Lab https://bzl.es

[2] Spike RISC-V ISA Simulator https://github.com/riscv-software-src/riscv-isa-sim.

[3] RISCV-DV https://github.com/chipsalliance/riscv-dv

[4] A. Kurth et al., "An Open-Source Platform for High-Performance Non-Coherent On-Chip Communication," in IEEE Transactions on Computers, vol. 71, no. 8, pp. 1794-1809, 1 Aug. 2022, doi: 10.1109/TC.2021.3107726.

[5] RISCV-Tests https://github.com/riscv-software-src/riscv-tests/