Learn RISC-V CPU Implementation and **BSV**

(**BSV**: a High-Level Hardware Design Language)

Rishiyur S. Nikhil

L12: RISC-V: Functional Verification of CPUs

Reminders

Please git clone: `https://github.com/rsnikhil/Learn_Bluespec_and_RISCV_Design`
(git pull for latest version). Repsitory structure:

```
./Book_BLang_RISCV.pdf
  Slides/
      Slides_01_Intro.pdf
      Slides_02_ISA.pdf
      ...
  Exercises/
      Ex-03-A-Hello-World/
      Ex-03-B-Top-and-DUT/
      ...
  Code/
      src_Top/
      src_Drum/
      src_Fife/
      src_Common/
      ...
  Doc/Installing_bsc_Verilator_etc.{adoc,html}
```

- Slides and Exercise are numbered in sync with book Chapter numbers.
- For Exercises, please see Appendix E of the book. Some (not all) exercises have associated code in the `Exercises/` directory.

To compile and run the code for exercises, Drum and Fife, please make sure you have installed:

- *bsc* compiler (see `https://github.com/B-Lang-org/bsc`)
- Verilator compiler (see `https://www.verilator.org/`)

Chapter Roadmap

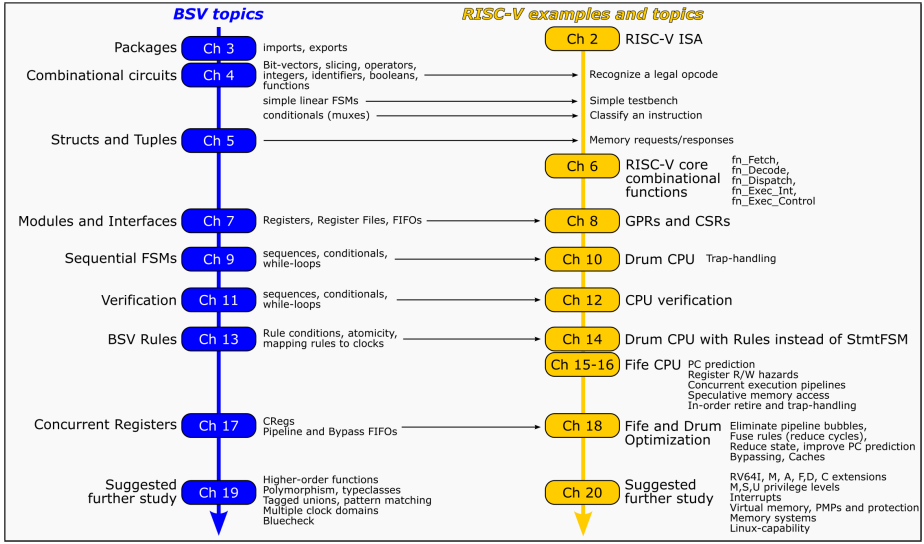


Table of Contents

Introduction

Functional Correctness *vs.* Performance Correctness

Functional Correctness:

> *"Does the CPU compute the correct answer?"*

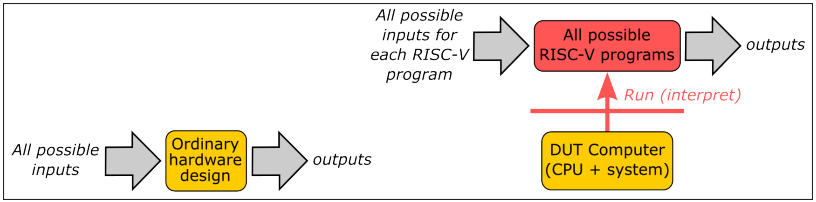
Performance Correctness:

> *"Does the CPU compute the result in an acceptable amount of time?"*

During CPU verification, typically the initial focus is on functional correctness. As the functionality stabilizes, the emphasis moves to performance correctness.

Complexity of CPU Verification

CPU verification is usually much harder than verification of ordinary (non-CPU) hardware designs because the space of possible inputs is the space of possible RISC-V programs multiplied by the space of possible inputs to each RISC-V program.



Determinacy of instruction traces

Fortunately, RISC-V program execution is *mostly deterministic*, *i.e.,* for a given RISC-V program and a given input to the program, the sequence of instructions executed (its "instruction trace") does not change from one run to the next, nor from one CPU implementation to another.

The only sources of non-deteriminism are:

- Interrupts (arrive at unpredictable times)
- Results of a few instructions, such as `rdcycle` and `rdtime` (depend on microarchitectural details)

This allows us to compare the insruction trace of a "trusted" execution platform with the instruction trace of the DUT. Any divergence indicates a potential bug in the DUT implementation.

(But, note, we also have to account for non-deteriminacy; we will discuss that later, under "asymmetric tandem verification".)

Trusted Functional Simulators

Trusted Functional Simulators ("Golden Reference Models")

A Trusted Functional Simulator or Golden Reference Model is a *simulator* for RISC-V instruction execution, which:

- can load and run RISC-V ELF binaries.
- at a minimum, produces an *instruction trace*, *i.e.,* a list of PCs of the instructions executed.

  By correlating these with the original ELF file, we know the instructions executed.

  The trace can produce more useful information than just the PCs, such as:

  - the instruction number (sequential number).
  - the instruction itself (can be verified against the ELF file to verify correctness of insruction-fetch)
  - the new values of any updated registers/CSRs
  - For Load/Store/AMO instructions, the effective memory address for any Load/Store/AMO instruction and the value loaded/stored
  - Trap information, if the instruction trapped
  - ...

Implementations of Trusted Functional Simulators

A Trusted Functional Simulator:

- is written primarily for clarity and maintainability (for trustworthiness!),
  and only secondarily for performance (simulation speed).
- is usually written in a software programming language (typically C/C++). This can be run as a standalone program, or "imported" into an HDL (**BSV**, Verilog, SystemVerilog, VHDL) to run in an HDL simulator. The purpose of running in an HDL simulator is *not* "cycle-accurary" (the simulator usually is by no means cycle-accurate with any hardware implementation), but easy interfacing to other system hardware components.
- is sometimes written in an HDL. Again, the purpose of running in an HDL simulator is *not* "cycle-accurary" (simulator "cycles" may have no relationship to any hardware implementation), but easy interfacing to other system hardware components.
- is sometimes written in a *synthesizable* HDL (**BSV**, Verilog, SystemVerilog, VHDL), in chich case it can be run on an FPGA, at potentially much greater speed than a software simulator.

  If run an FPGA, one has to provide some means to record the instruction trace on a host machine.

Trusted Functional Simulators: customizability and speed

Additional desirable properties of a trusted functional simulator:

- *Configurability/Customizability*: Ability to configure it for a particular RISC-V ISA: RV32I vs RV64I; optional M, A, F, D, C ISA extensions; optional M,S,U privilege levels; optional Sv32/Sv39/Sv48/Sv57 virtual memory schemes; optional multi-hart (multi-core); ...

  This enables the trusted functional simulator to be used for verification of different RISC-V hardware implementations, each of which typically makes specific choices amongst these opions.

  Configurability/Customizability

  For verifying RISC-V implementations that implement custom ISA extensions, the trusted functional simulator needs to be customizable also to be able to execute these extensions.

  Configurability/Customizability can be:

  - Static: recompile and rebuild a version of the simulator for a given set of option choices
  - Dynamic: a single simulator executable conforms to a given set of option choices specified in an input file.

- *Simulation Speed*: Ability to boot and run an operating system (*e.g.*, Linux) and applications within the OS. Ability to exploit multiple cores in the host computer for greater speed.

  Trusted functional simulators can typically boot Linux in minutes, sometimes in seconds. An HDL simulation of an actual RISC-V implementation can take days to simulate to the same point.

Existing Trusted Functional Simulators: Free and Open-source

Spike

The most famous and widely used trusted functional simulator.

- Continuously available since the earliest days of RISC-V. Continuously maintained, at high quality, up-to-date with all the latest ISA extensions.
- Written in C/C++.
  Fast (approx. 50-100 MIPS).
- Available at: `https://github.com/riscv-software-src/riscv-isa-sim`

Sail RISC-V Formal Model

A little less famous (to date), but equally important.

- The "official" formal specification for RISC-V, *i.e.,* the official definition of the semantics of the RISC-V ISA (a peer of the the textual documents for Unprivileged Spec, Privileged Spec, *etc.*).

  People who do *formal verification* of RISC-V CPU implementations and compilers use this model as the reference for proving correctness.
- Written in the Sail formal specification language. Compiled by the Sail compiler into an executable under Linux, MacOS, ...
  Reasonably fast (approx. 10-20 MIPS).
- Available at: `https://github.com/riscv/sail-riscv`

Other Functional Simulators

Qemu

Widely used.

- Prioritizes ability to model full systems (CPU,memory, devices), for RISC-V software development.
- Prioritizes extremely high speed for software development, exploiting JIT (Just-In-Time) dynamic compilation.
  Very fast (approx. 1 GIPS).
- Free, open-source.

There are probably many functional simulators within many companies in the RISC-V verification ecosystem, which may not be known publicly. Two of them are:

Synopsys ImperasDV

From Synopysis.

- The simulator was originally a product of the company Imperas, which was aquired by Synopsys in 2023.
- Imperas' simulator and modeling components are being integrated into Synopsys' VCS simulation and DV (Design for Verification) suites.

Cissr

From Bluespec, Inc.

- Can model caches, multi-hart, and some devices.
- Fast (speed compareable to Spike).

Test programs

Test programs

CPU verification involves verifying correct behavior over a very large suite of test programs.

- *Standard ISA Tests*: RVI (RISC-V International) maintains a set of several hundred standardized tests. These are small test programs, written in RISC-V Assembly Language, organized by ISA extension: RV32I, RV64I, A, M, F, D and C extensions, Machine/Supervisor/User mode, *etc.*

  Available at: `https://github.com/riscv-software-src/riscv-tests`

- *ACTs*: RVI (RISC-V International) is developing a set of tests called ACTs (Architecture Compatibility Tests). Each test is a RISC-V program that runs and produces a final "signature", which is a reliable hash of the final state (PC, registers, CSRs, memory, *etc.*). Over time, these will become the official "certification suite" for RISC-V.

  ```
  https://wiki.riscv.org/pages/viewpage.action?pageId=49872986
  https://github.com/riscv-non-isa/riscv-arch-test
  https://riscof.readthedocs.io/en/stable/
  ```

- *riscv-dv*: developed initially at Google; subsequent stewardship by the ChipsAlliance non-profit group. These tests generate "random" RISC-V programs (constrained to a specified ISA subset), run the program on a candidate implementation which should generate a trace, and compare against a trace from a trusted functional simulator:

  ```
  https://github.com/chipsalliance/riscv-dv
  ```

In addition, most organizations maintain their own collection of test programs and "regression suites" (programs that previously surfaced a bug in some implementation, which are continuously re-run to ensure that the bug has not resurfaced due to more recent changes).

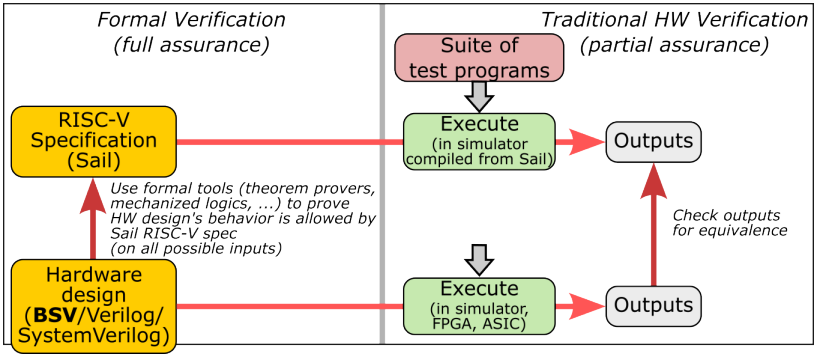
Levels of Assurance

Levels of Assurance

The "gold standard" for assurance that a design is fully correct (on all possible inputs) is *Formal Verification*.

This is a very difficult problem, and is still aspirational (there are some recent research successes, and the capability improves steadily).

In the meanwhile, we have been doing traditional HW verification.



"*Coverage*" is a quantifiable measure of how partial/full is the assurance of correctness:

- How many/what fraction of possible inputs are in the suite of test programs?
- What fraction of the source code lines in the design have been exercised by the suite of test programs?

Obviously, the higher these measures, the higher the level of assurance.

Levels of Assurance

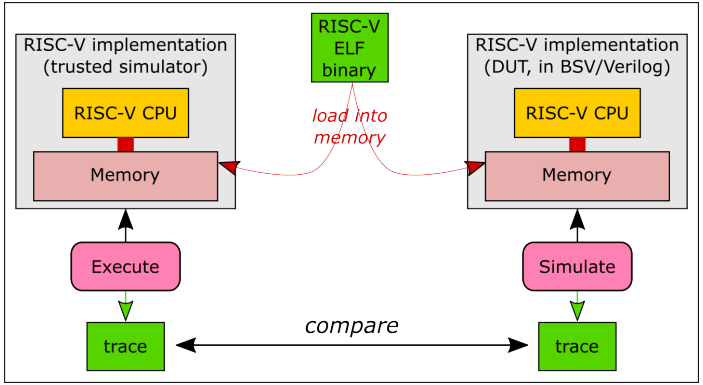
Here is a sequence of testing regimes with increasing simulation times but also increasing levels of assurance of the correctness of our CPU implementation:

- Run all "Standard ISA tests" mentioned.
- Run the ACTs and riscv-dv test suites.
- Run a small operating system (such as FreeRTOS or Zephyr).
  This will check correct handling of timer interrupts, and possibly memory maps and physical memory protection (PMPs).
- Boot the kernel of a full-service operating systsem (such as the Linux kernel).
- Run a standard distribution a full-service operating systsem (such as Debian Linux or Ubuntu Linux), *i.e.*, the OS kernel *plus* the pre-load of all the applications and service programs that come with distribution (including block devices and networking). Run applications under the OS.

The latter items are often executed on FPGA because simulation speed may be too slow.

Tandem Verification

Symmetric Tandem Verification



Run a RISC-V binary (ELF file) on two RISC-V implementations, both configured for exactly the same RISC-V ISA. The times taken to execute the program may be different.

Usually: one is a trusted functional simulator, the other is the DUT.

Each side produces an *instruction trace*, which we compare for equivalence.

Divergence implies a likely bug in the DUT implemenetation.

- At a minimum, the traces specify the sequence of PCs encountered during execution.
  More convenient: traces also specify instruction number, the instruction, updates to registers and CSRs, effective address in memory for Load/Store/AMO, value loaded/stored for Load/Store/AMO, ...
- Complication: non-determinism (interrupts, reading uninitialized memory, Store-Conditional success/failure, ...).
- Complication: modeling the DUT's devices (MMIO) in the reference simulator.

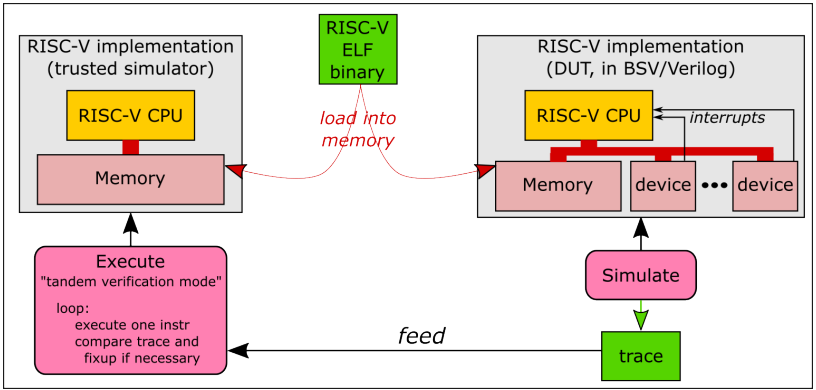
Symmetric Tandem Verification: notes

- Trace outputs can get very large (gigabytes).
- "Offline comparison": Run the two simulators independently and record the output traces. Separately, compare the traces.
- "Online comparison": Run the two simulators concurrently (as two concurrent processes);
  feed the trace outputs to the comparator running as another process that consumes both traces as they are produced.
  Nno need to record the traces, and can stop simulation as soon as divergence is detected.
- Since the trusted simulator's trace will be the same on repeated runs, it can be recorded once and just replayed into the (offline or online) comparator.

- Simulation time and trace sizes can be reduced if the trusted simulator and the DUT have "snapshot" capability:
  - Run the trusted simulator (fast) up to instruction N (*e.g.,* 100 million), then record a snapshot of the entire architectural state (all registers, CSRs, memory).
  - Pre-load the DUT's entire architecture state with the snapshot, then resume execution, thereby avoiding simullation time for executing the first N instructions.

Asymmetric and "full system" Tandem Verification (1/2)

If we can modify the trusted simulator to run in "verification mode", then Tandem Verification can handle non-determinism and devices (MMIO). (Details on next slide.)



Asymmetric and "full system" Tandem Verification (2/2)

The trusted simulator consumes the trace from the DUT simulation either in real time (online) or from a recording (offline) and, normally, compares each instruction from the trace with its own behavior (as in symmetric tandem verification).

- *Non-determinism: interrupts*: the DUT trace should include interrupt events:
  (a) when a bit in CSR MIP changes state ($0 \rightarrow 1$ and $1 \rightarrow 0$), and
  (b) when an interrupt is actually taken (responding to a 1 bits in MIP).

  For (a), the trusted simulator records the same change in its CSR MIP.

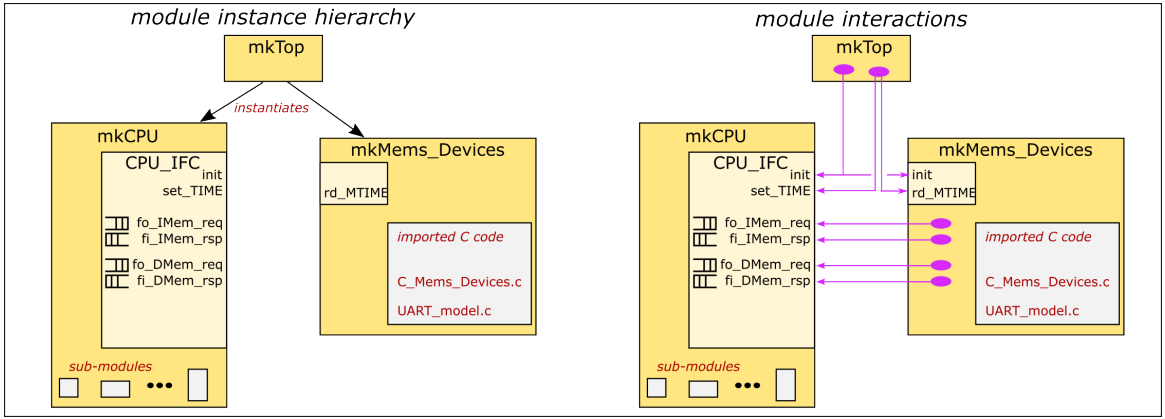
  For (b), the trusted simulator checks that it is legal to take this interrupt at this point in the trace and, if so, also takes the same interrupt at this point.
- *Non-determinism: Store-conditional success/fail result,* `rdtime` *and* `rdcycle` *instructions*: the DUT takes the value from the trace instead of its own.
- *Non-determinism: Reading uninitialized memory*: if the trusted simulator and the DUT disagree on a load-value, it could be because of uninitialized memory (and memory was not initialized identically in the two simulators).

  The verifier can issue a warning message, take the load-value from the DUT trace as the loaded value, and proceed.
- *Device (MMIO) Loads*: the trusted simulator takes the loaded value (or load exception) from the trace.
- *Device (MMIO) Stores*: the trusted simulator discards the stored value. If the trace indicates that the store response is an exception, the trusted simulator takes the same exception.

Testbench for Drum and Fife

Testbench for Drum and Fife (1/2)

We use exactly the same testbench for Drum and Fife (discussion on next slide):



Testbench for Drum and Fife (2/2)

- `mkMems_Devices` models everything outside the CPU (memory and devices).
  It is actually implemented in C and imported into a **BSV** wrapper using a standard mechanism provided by **BSV**.
- `mkTop` mostly just instantiates `mkCPU` (Drum or Fife) and `mkMems_Devices`, passing the former's memory interfaces into the latter.
- `mkTop` only performs the following:
  - invokes the CPU's `init` method once, at startup, to provide the CPU with the initial value of the PC (from which it should start fetching).
  - continually (on every clock) relays the value of real time from `mkMems_Devices` to the CPU, which places it in its CSR TIME. This is the value read by the `rdtime` instruction.
- `mkMems_Devices` continually dequeues memory requests from the CPU using its `fo_IMem_req` and `fo_DMem_req` methods. For each request it invokes C code to compute a response and returns it to the CPU using the using its `fi_IMem_rsp` and `fi_DMem_rsp` methods.
  The C code internally models memory as an ordinary C array of bytes, and it contains a model of a standard 16550 UART for character I/O.

Final comments on RISC-V Verification

- The functional verification question is:

  Does this CPU implementation always produce the correct answer, for any RISC-V program that we load into its memory?

  The performance verification question is:

  Does this CPU implementation produce its answer in the expected time, for any RISC-V program that we load into its memory?

  This chapter is only about functional verification

- Formal verification is the gold standard for functional verification, but the technology is not yet ready for production use. With today's technology it is routinely used for verifying simple properties and verifiying components.

- Until then, verification is done by running test programs and comparing program outputs and instruction traces with corresponding outputs and traces from a trusted functional simulator.

  This comparison is called "tandem verification", which can be online or offline, symmetric or asymmetric (the latter has advantages in dealing with non-determinism and devices (MMIO).

  "Coverage metrics" quantify the level of assurance.

End