Verification and Debugging for High-Assurance RISC-V

Rishiyur S. Nikhil
Co-founder and CTO, Bluespec, Inc.

RISC-V International Conference at IIT Madras
April 2, 2017

www.bluespec.com

Overview

CPUs and SoCs are complex designs
and correctness is crucial.

Many techniques are used in concert for high assurance.

- Correct-by-Construction
- Direct remote GDB
- Tandem Verification
- Seamless FPGA acceleration
- BlueCheck (hardware QuickCheck) automated testing
- Formal Models and Proofs of CPUs and Memory Systems

The Bluespec “RISC-V Factory” already incorporates many of these techniques (and will eventually incorporate all of them)

Correct-by-Construction

In software development, it is now taken for granted that High-Level Languages (HLLs) are essential for developing robust, complex systems.

Many properties of HLLs contribute to robustness by construction, by eliminating many bugs statically, that would only be caught during execution if using lower-level languages (which, in hardware, may be too late!):

- A powerful type system:
  - Expressive user-defined types (structs, arrays, tagged unions, enums, ...)
  - Type parameterization (polymorphism/generics/…)
  - Strong type-checking
- Powerful value parameterization
  - Higher-order functions with parameters of any data type, including functions and modules
- Object-oriented interfaces (methods, not wires)
- Behavioral abstractions to manage composition and concurrency
  - Object-oriented interfaces (methods, not wires)
  - FSMs
  - Atomic transactions
- Formal semantics: simple, clear, unambiguous

(Note: these features are needed for synthesizable designs, not just for modeling and simulation)

For hardware design, Bluespec BSV has all these features; Chisel has many of them; SystemVerilog has some of them.

Overview

CPUs and SoCs are complex designs and correctness is crucial.

Many techniques are used in concert for high assurance.

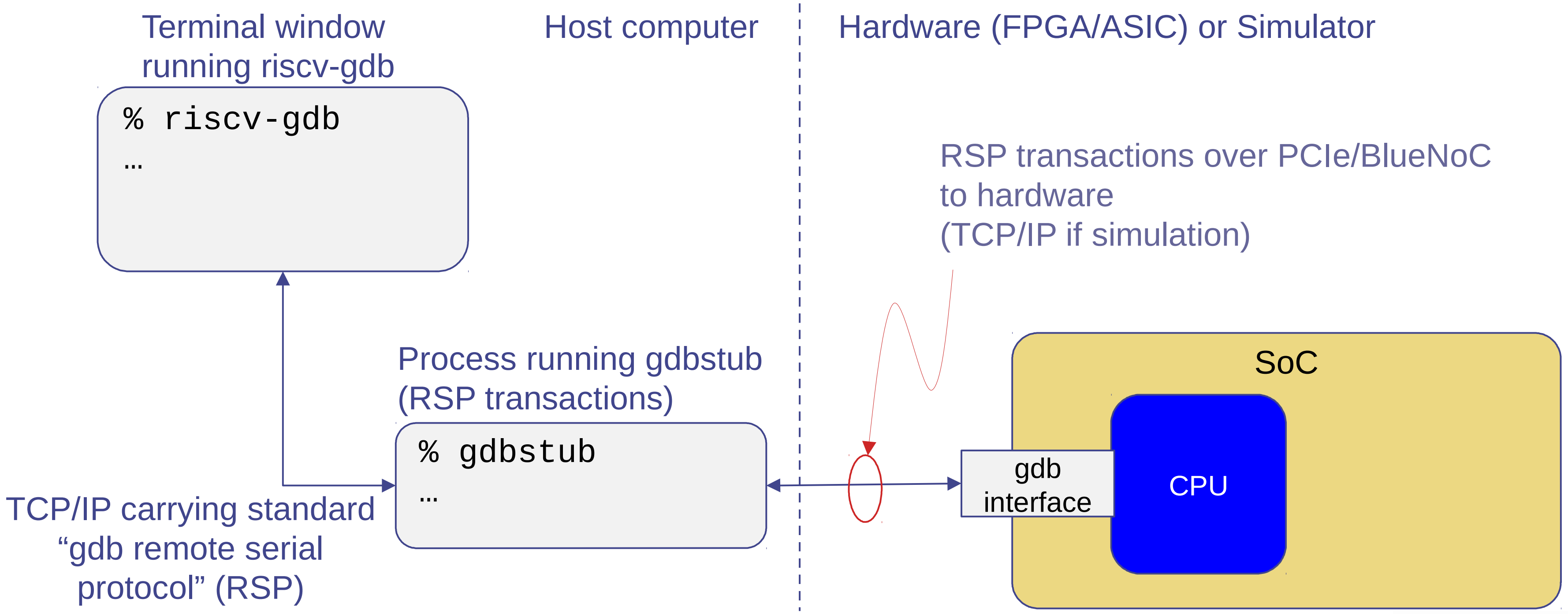
- Correct-by-Construction
- ➡ Direct remote GDB
- Tandem Verification
- Seamless FPGA acceleration
- BlueCheck (hardware QuickCheck) automated testing
- Formal Models and Proofs of CPUs and Memory Systems

bluespec™

Direct Remote GDB

- GDB is often run remotely, i.e., to debug a process on a remote computer
  - Uses a remote "gdb stub" that receives standard GDB RSP (Remote Serial Protocol) (r/w registers, mem, PC; set/remove breakpoints; continue, step, …)
- Normally, all this is done completely in software
  - Perfectly adequate if your CPU/HW system/OS kernel are all stable
- But what if you're developing a CPU/system/OS kernel that are not yet stable?

- We've developed a "gdb stub" *in hardware* hooking directly into the *CPU hardware*
  - Gives us "a rock to stand on"
  - Can be used with 3rd-party CPUs, not just those from Bluespec, Inc.


Terminal window running riscv-gdb
% riscv-gdb
…
Host computer
Hardware (FPGA/ASIC) or Simulator
RSP transactions over PCIe/BlueNoC to hardware (TCP/IP if simulation)
Process running gdbstub (RSP transactions)
% gdbstub
…
SoC
CPU
gdb interface
TCP/IP carrying standard "gdb remote serial protocol" (RSP)

Overview

CPUs and SoCs are complex designs
and correctness is crucial.

Many techniques are used in concert for high assurance.

- Correct-by-Construction
- Direct remote GDB
- ➡ Tandem Verification
- Seamless FPGA acceleration
- BlueCheck (hardware QuickCheck) automated testing
- Formal Models and Proofs of CPUs and Memory Systems

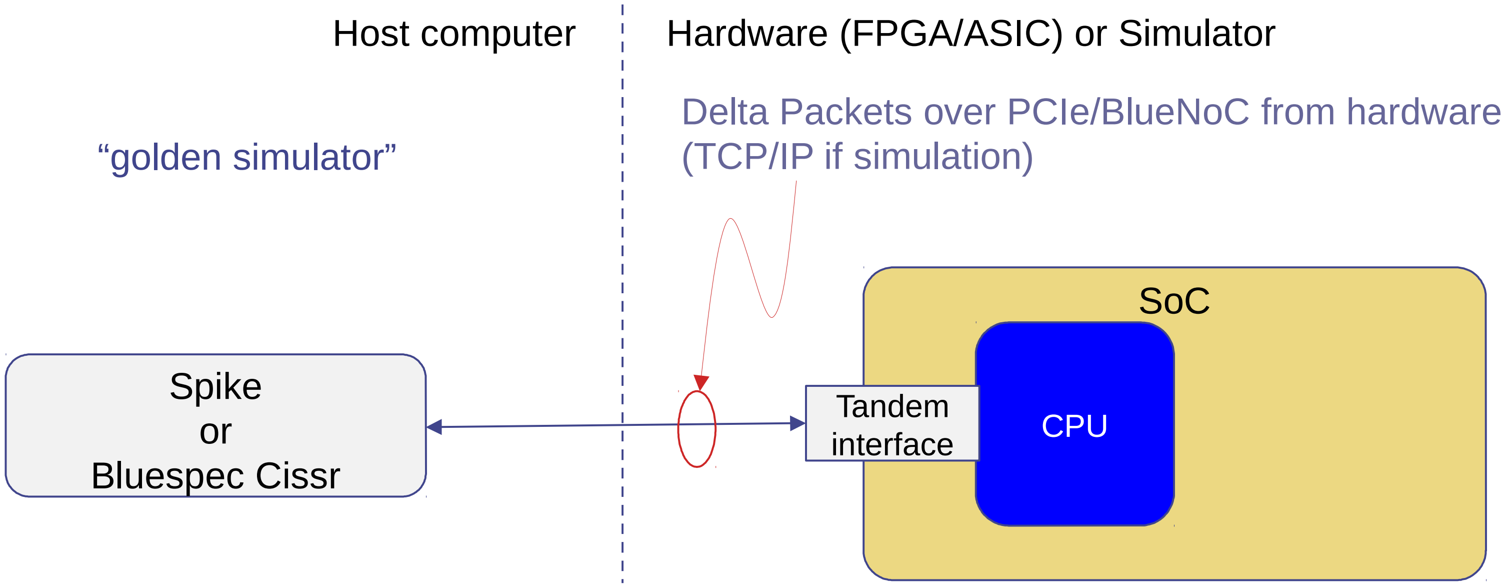
Tandem Verification (1)

CPU pipeline bugs can be very difficult to find

- May only be triggered on certain instruction combinations which may only occur after millions/billions of instructions
- Effect of bug may be latent for further millions of instructions (e.g., bad value stored and reloaded much later)

Bluespec builds-in “*Tandem Verification*” into each of its CPUs

- Observation: each instruction only makes small “delta” to architectural state (reg write, mem store value)
- CPU is instrument to emit a “delta packet” on each instruction
- This delta is checked against a “golden simulator” running in tandem

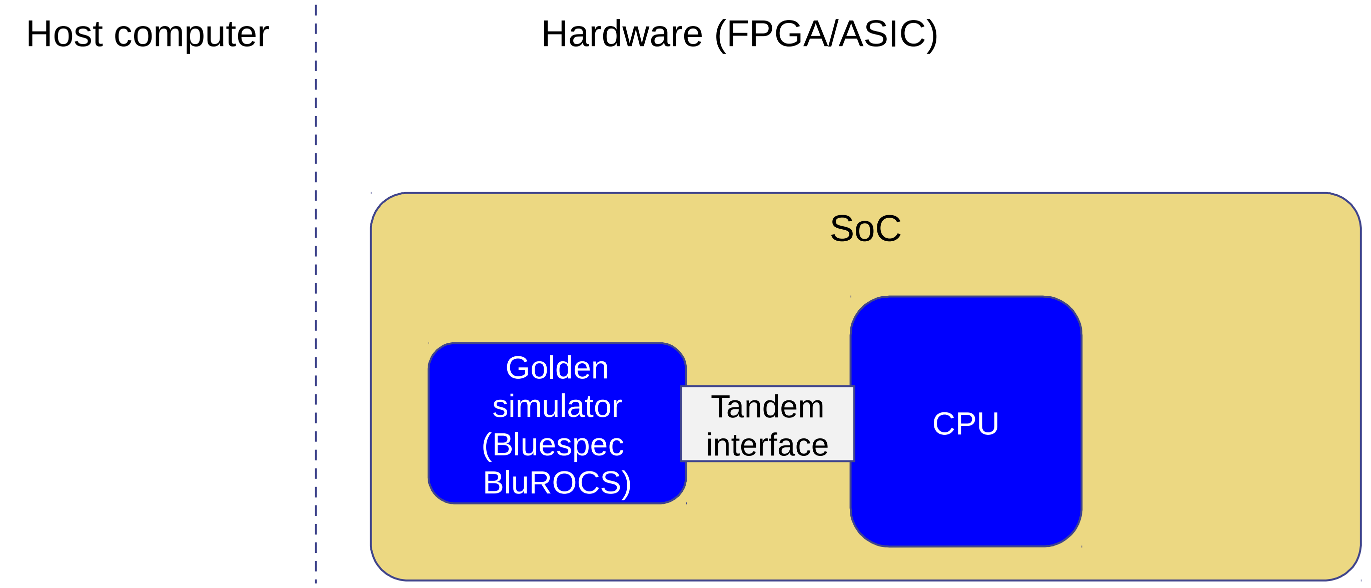


Host computer
Hardware (FPGA/ASIC) or Simulator
“golden simulator”
Delta Packets over PCIe/BlueNoC from hardware (TCP/IP if simulation)
SoC
Spike or Bluespec Cissr
Tandem interface
CPU

It's also been our observation that Tandem Verification provides more coverage, more quickly, than directed asm ISA tests.

Tandem Verification (2)

- Tandem verification as shown on the previous slide (with a software simulator like Spike or Bluespec Cissr) is of course bottlenecked by the speed of the communicating delta packets from HW to the host.
  - (It's still worth it's weight in gold!)
- But this can be significantly accelerated by using, instead, a synthesizable golden simulator running on an FPGA adjacent to the CPU



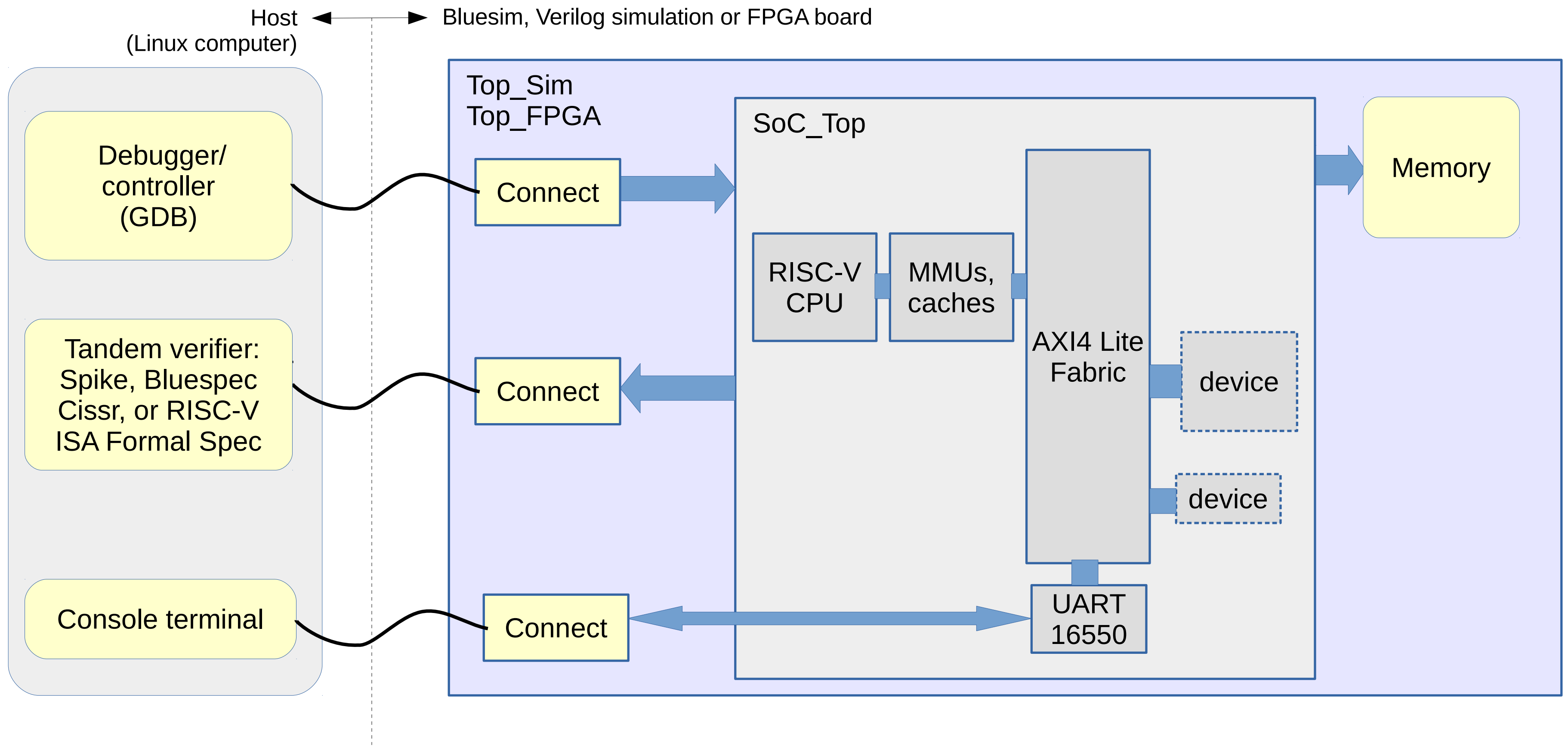
Overview

CPUs and SoCs are complex designs and correctness is crucial.

Many techniques are used in concert for high assurance.

- Correct-by-Construction
- Direct remote GDB
- Tandem Verification
- Seamless FPGA acceleration
- BlueCheck (hardware QuickCheck) automated testing
- Formal Models and Proofs of CPUs and Memory Systems

Seamless FPGA acceleration



Bluespec RISC-V Factory:

Identical setup (host-side view) for simulation and for FPGA execution.

- Simulation (Bluesim/Verilog) uses memory and UART models, TCP or PTY connections
- FPGA execution uses DRAM memory and actual UART, connections to host over PCIe

Overview

CPUs and SoCs are complex designs and correctness is crucial.

Many techniques are used in concert for high assurance.

- Correct-by-Construction
- Direct remote GDB
- Tandem Verification
- Seamless FPGA acceleration
- BlueCheck (hardware QuickCheck) automated testing
- Formal Models and Proofs of CPUs and Memory Systems

Bluecheck (hardware Quickcheck) automated tests

QuickCheck was developed in the Haskell world.

cf. “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs”,

K.Claessen and J.Hughes, Proc. Intl. Conf. on Functional Programming (ICFP), 2000

It has become a standard part of Haskell program verification, and has been ported and used in many other programming languages.

BlueCheck is the first implementation of QuickCheck in a Hardware Design Language,

and the first that is *synthesizable* (so, can run *much* larger tests on FPGAs).

BlueCheck is just a library—nothing is built-in to Bluespec BSV

(relies on BSV's high-level language features: typeclasses, monadic collections, first-class types, first-class resets, atomic transactions, ...)

BlueCheck is an open-source library in Bluespec BSV, developed by Matthew Naylor and Simon Moore at U.Cambridge (UK).

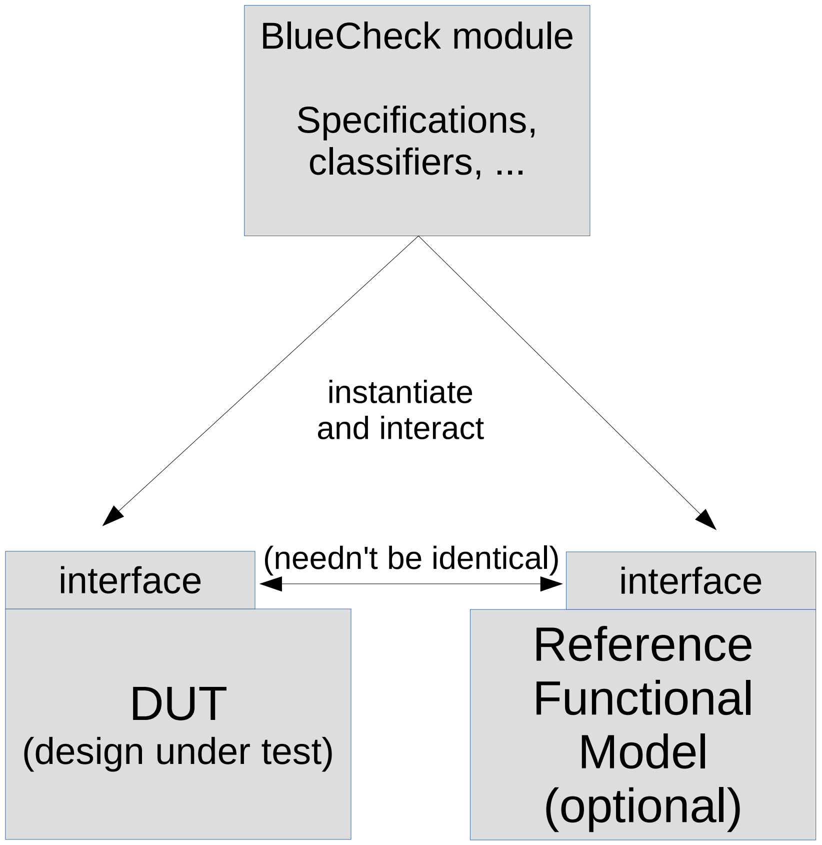
cf. “A Generic Synthesizable Test Bench”,

M.Naylor and S.Moore,

Proc. 13th ACM/IEEE Intl. Conf. of Methods and Models for System Design

(MEMOCODE), University of Texas at Austin, USA, Sep 21-23, 2015

Anatomy of a BlueCheck test system



BlueCheck specification elements
- Equivalences (between activities on DUT and model)
- Properties and Assertions (boolean predicates)
- Optional customization of input randomizers

From these declarative specs, BlueCheck *automatically*
- generates randomized tests
- tests the properties
- shrinks failures to minimal examples

DUT
- Is a black box (no internal visibility to BlueCheck)
- Can be BSV code, and/or Verilog/VHDL imported with a BSV wrapper

Reference model
- Is a black box (no internal visibility to BlueCheck or to compiler)
- Can be BSV code, or arbitrary model imported with a BSV wrapper, in C, Verilog/VHDL, or in some formal specification language (executable)
- Can even be 2nd instance of DUT (e.g., for algebraic property testing)

Example specification elements (1)

```
equiv (“push”,     dut_stack.push,     reference_stack.push);
equiv (“pop”,      dut_stack.pop,      reference_stack.pop);
equiv (“isEmpty”, dut_stack.isEmpty, reference_stack.isEmpty);
```

User-chosen name for this element of the spec

Two arbitrary expressions of the same type. Here:

• push:	stack_elem_type → Action
• pop:	ActionValue #(stack_elem_type)
• isEmpty:	Bool

What is a specification element? (2)

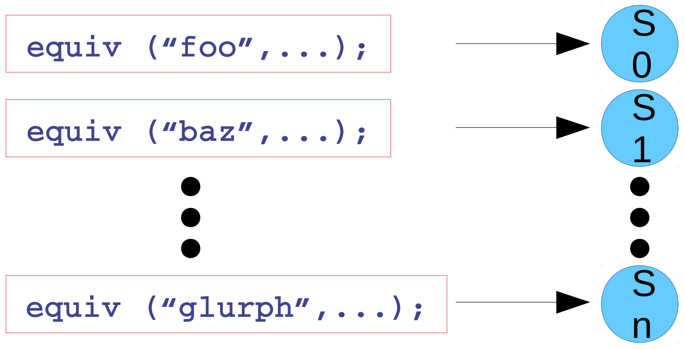
```
equiv ("foo",    f,    g);
```

In general, these are two arbitrary expressions, not necessarily of the form *module.method* shown earlier.

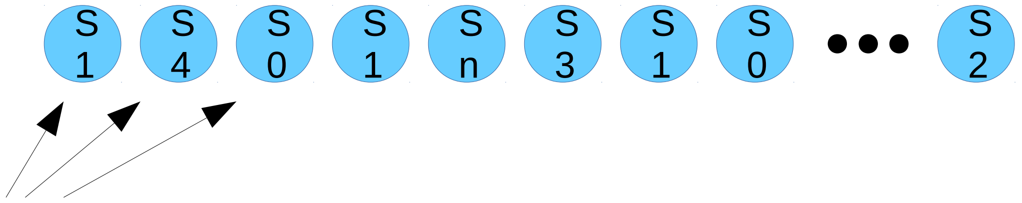
- The type, in general, is an arbitrary-arity function with arbitrary argument and result value types:
  - t1 → t2 → … → tn → any_value_type
  - t1 → t2 → … → tn → Action
  - t2 → t2 → … → tn → ActionValue #(any_value_type)
  - t1 → t2 → … → tn → Stmt
- BlueCheck will generate randomizers for each argument, and check all returned values for equality
- For each 'equiv' statement, BlueCheck generates a "state" for a state machine in which it evaluates (executes) the two expressions in parallel by applying the two functions to the same random inputs
  - Atomically, in a single rule/clock, for Action/ActionValue types
  - Multiple rules/clocks for Stmt types, using the StmtFSM "par" construct

What is a test sequence?

Each specification element is some "computation", represented internally as a "BlueCheck state":



A test sequence is a random walk amongst these states.



In each state, the computation(s) of that spec element are applied to random arguments, and the results (if any) are compared for equality.

- Actually, there is a state only for each "side-effecting" computation (Action/ActionValue/Stmt)
- Pure functions are applied and checked in every state

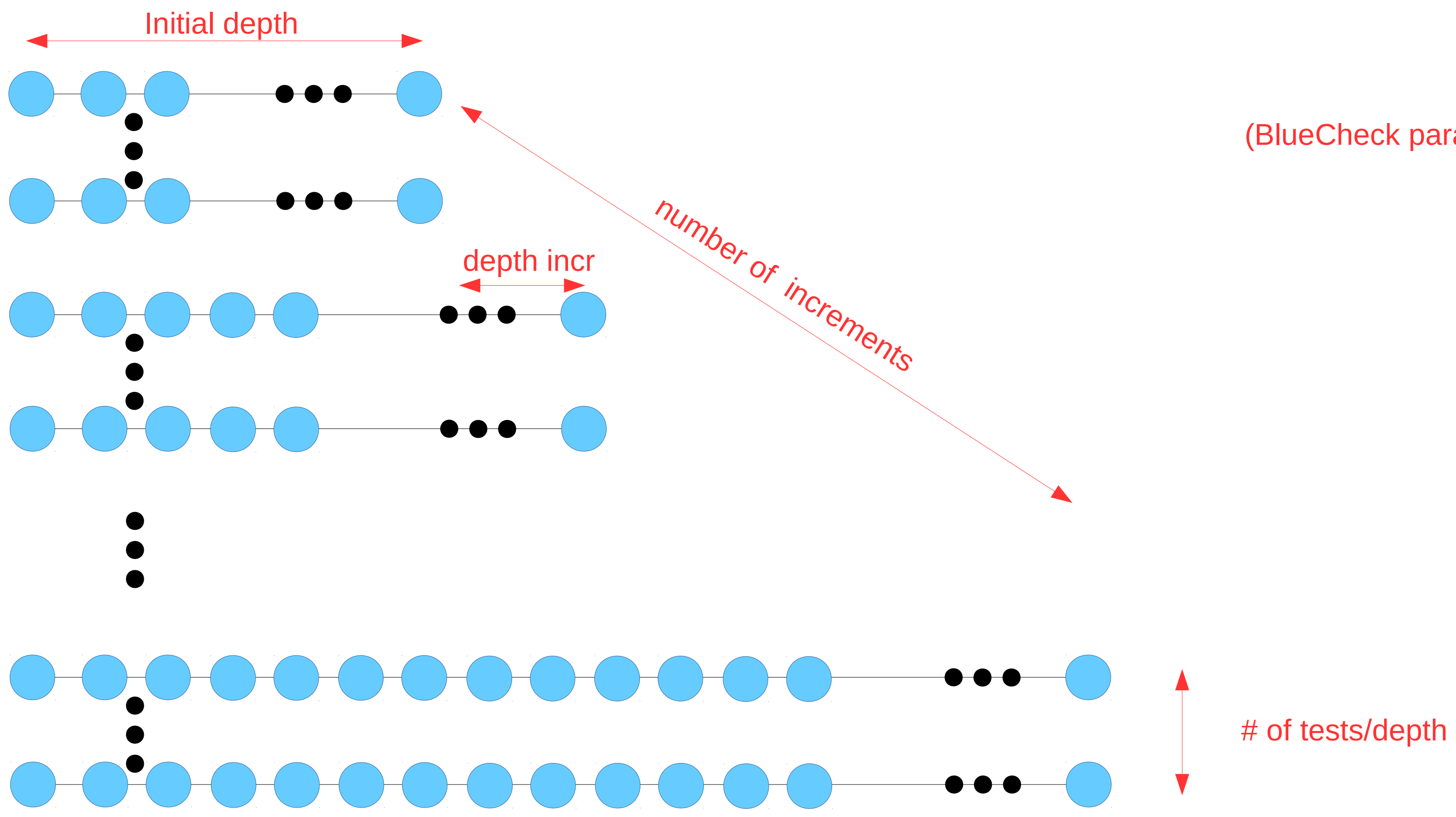
(BSV's powerful type system distinguishes pure and side-effecting expressions)

Multiple sequences: Iterative Deepening (“ID”)

By setting BlueCheck parameters, you can get it to run a single random sequence of specified depth (as illustrated in last slide)

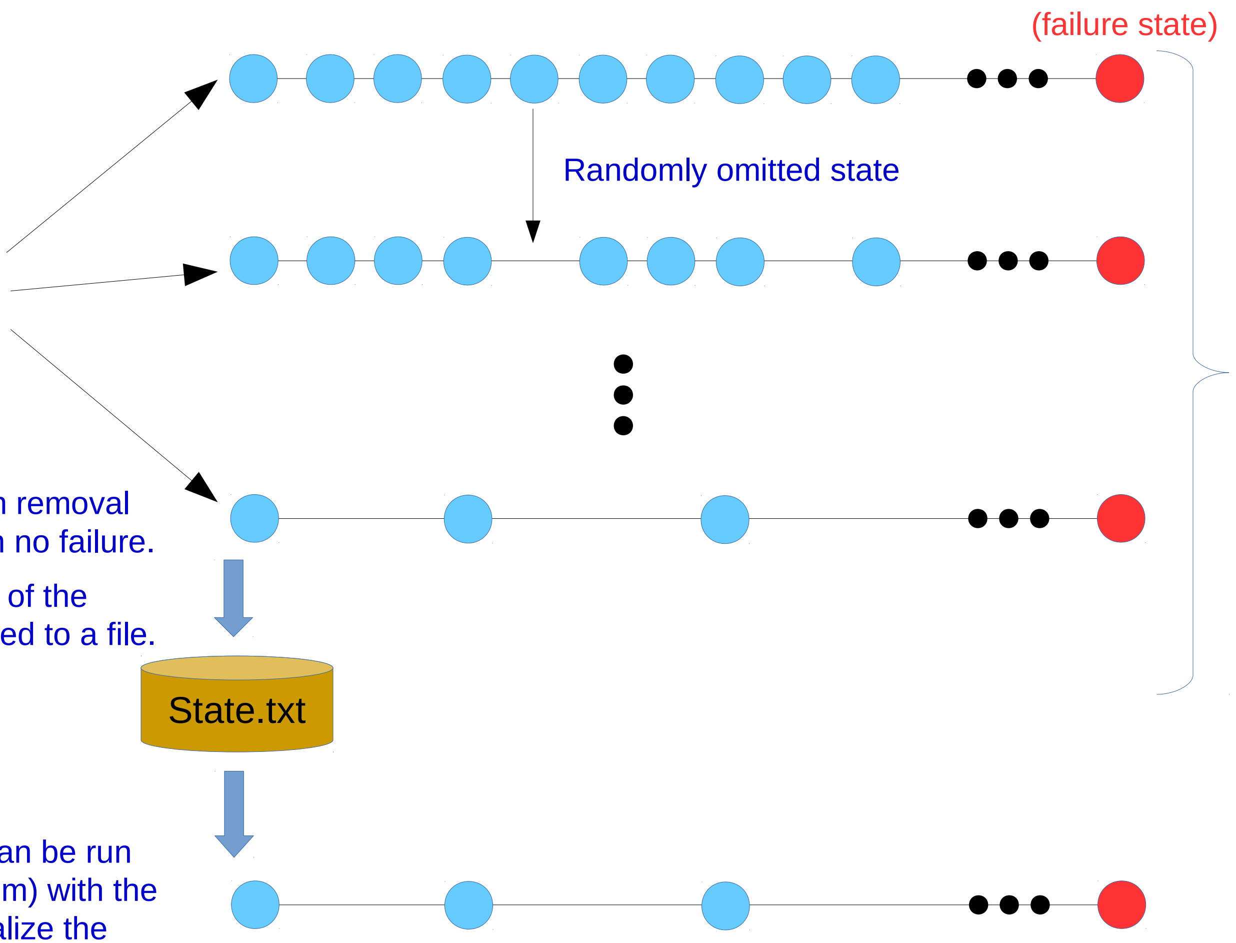
More often, one runs multiple randomized sequences, with Iterative Deepening

(each sequence starts at a different random state)



"Shrinking" and replay

When BlueCheck detects a failure, it repeatedly re-runs the sequence, randomly discarding states, trying to create a minimal-length sequence that still fails



Re-runs are done with the same initial state of the randomizers

All this can be done on an FPGA (see "synthesizability" discussion)

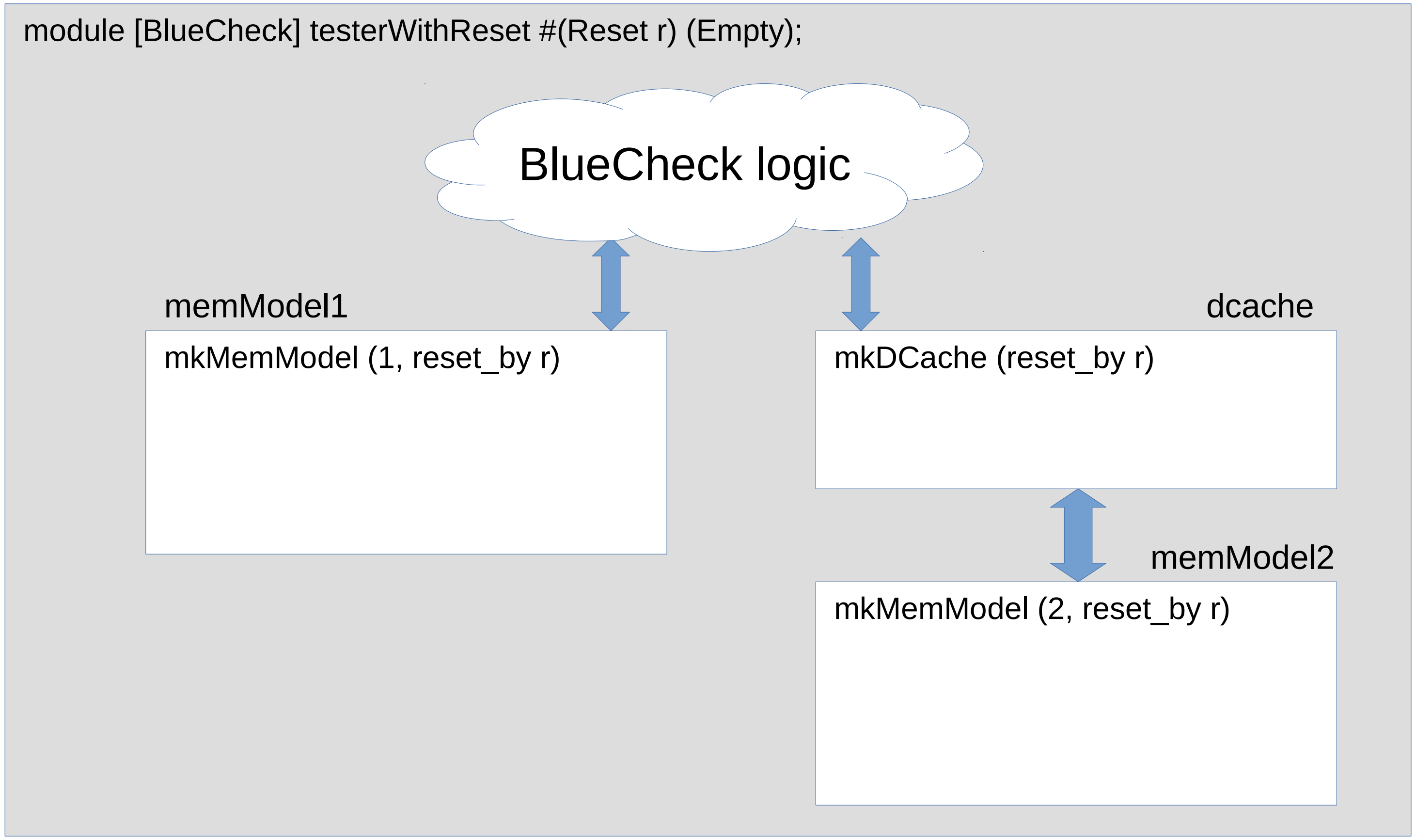
Shrinking stops when removal of any state results in no failure.

Then, the initial state of the randomizers is dumped to a file.

The same program can be run (Bluesim or Verilog sim) with the `+replay` flag to initialize the randomizers in the same way, to reproduce the failure.

This shorter sequence can be run at high trace verbosity to diagnose the problem.

Example: testing a Cache Memory (see code)


module [BlueCheck] testerWithReset #(Reset r) (Empty);
BlueCheck logic
memModel1
mkMemModel (1, reset_by r)
dcache
mkDCache (reset_by r)
memModel2
mkMemModel (2, reset_by r)

The interfaces to mkMemModel and mkDCache are not identical.
Instead, in the BlueCheck testbench, we generate random “unified requests”,
and we have “transactors” that convert it into the mkMemModel and mkDCache requests.

The latencies of the model and the DUT may be quite different.
The DUT's latencies are variable (e.g., cache hit vs. miss)

Overview

CPUs and SoCs are complex designs
and correctness is crucial.

Many techniques are used in concert for high assurance.

- Correct-by-Construction
- Direct remote GDB
- Tandem Verification
- Seamless FPGA acceleration
- BlueCheck (hardware QuickCheck) automated testing
- Formal Models and Proofs of CPUs and Memory Systems

Formal Models and Formal Verification (1)

CPU Specification

How do you know if your CPU is “correct”?

- Today's answer: does it compute the same result as the Spike simulator?

Unfortunately this is not a very satisfactory answer, for several reasons

- Simulators are not the most readable of “specifications”
- Simulators have other considerations (e.g., simulation speed) that may severely impact clarity
- Simulators are often written in C/C++, and are not very useful as inputs to formal tools for formal verification

A formal spec emphasizes precision, completeness, clarity

- Is usually written in a formal spec language that itself has formal semantics
- and is suitable as an input to formal tools for formal verification
- (e.g., in Coq, L3, Event B, Bluespec BSV, ...)

The RISC-V Foundation has a Technical Committee group working on producing a Formal Spec for the RISC-V ISA (Chair: Rishiyur Nikhil)

Groups (of which I am aware) working on Formal Verification of RISC-V exist at MIT, Galois Inc., SRI, and Bluespec, Inc.

Formal Models and Formal Verification (2)

Memory Model Specification

A memory model is an *implementation-independent* spec that defines what *re-orderings* are allowed between LOADs and STOREs.

- Loose enough to allow CPU/system implementors to exploit re-orderings of memory operations for high performance,
- but sufficiently precise, predictable, and *implementation-independent*, to be usable by machine-code generators (compilers or humans), targeting unknown implementations of RISC-V CPUs.

Memory models are amongst the most difficult features of computer systems to specify.

- Of course in systems that are multi-threaded, multi-core, and possibly with heterogeneous CPUs,
- But even in single-threaded processors:
  - “Coherence” between Instruction-access and Data-access
  - “Coherence” between MMU structures and caches

The RISC-V Foundation has a Technical Committee group working on producing a Formal Spec for the RISC-V Memory Model

There are many, many groups (unrelated to RISC-V) working on the topic of memory models (e.g., see conferences POPL, ISCA, ...)

Summary

CPUs and SoCs are complex designs
and correctness is crucial.

Many techniques are used in concert for high assurance.

- Correct-by-Construction
- Direct remote GDB
- Tandem Verification
- Seamless FPGA acceleration
- BlueCheck (hardware QuickCheck) automated testing
- Formal Models and Proofs of CPUs and Memory Systems

At Bluespec, Inc. we are using or exploring all these techniques.

Thank you!