RISC-V Verif Generators

A Configurable ISA warrants a Configurable Verification Environment

Neel Gala
InCore Semiconductors

Configurable ISA ⇨ Configurable Verification

Configurable Verification

As designs become more configurable and fragmented, the number of possible **instances to verify grows exponentially** - making verification a bottleneck.
This warrants a verification methodology which includes a **scalable test-harness**, configurable and parameterized tests with a high level of quality assurance. **Automating test-generation** is the key here.

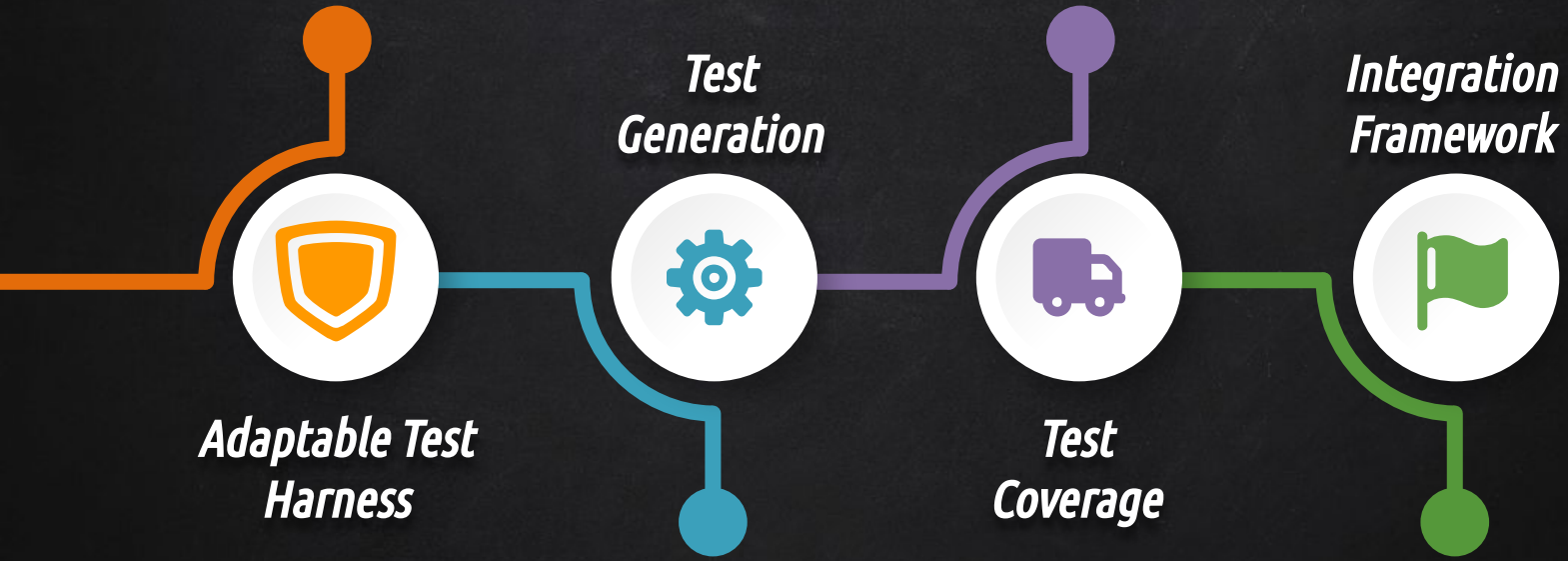
Configurable ISA

Open, Flexible and Modular ISAs like **RISC-V** along with its rich and diverse ecosystem are key to realizing the next generation solutions

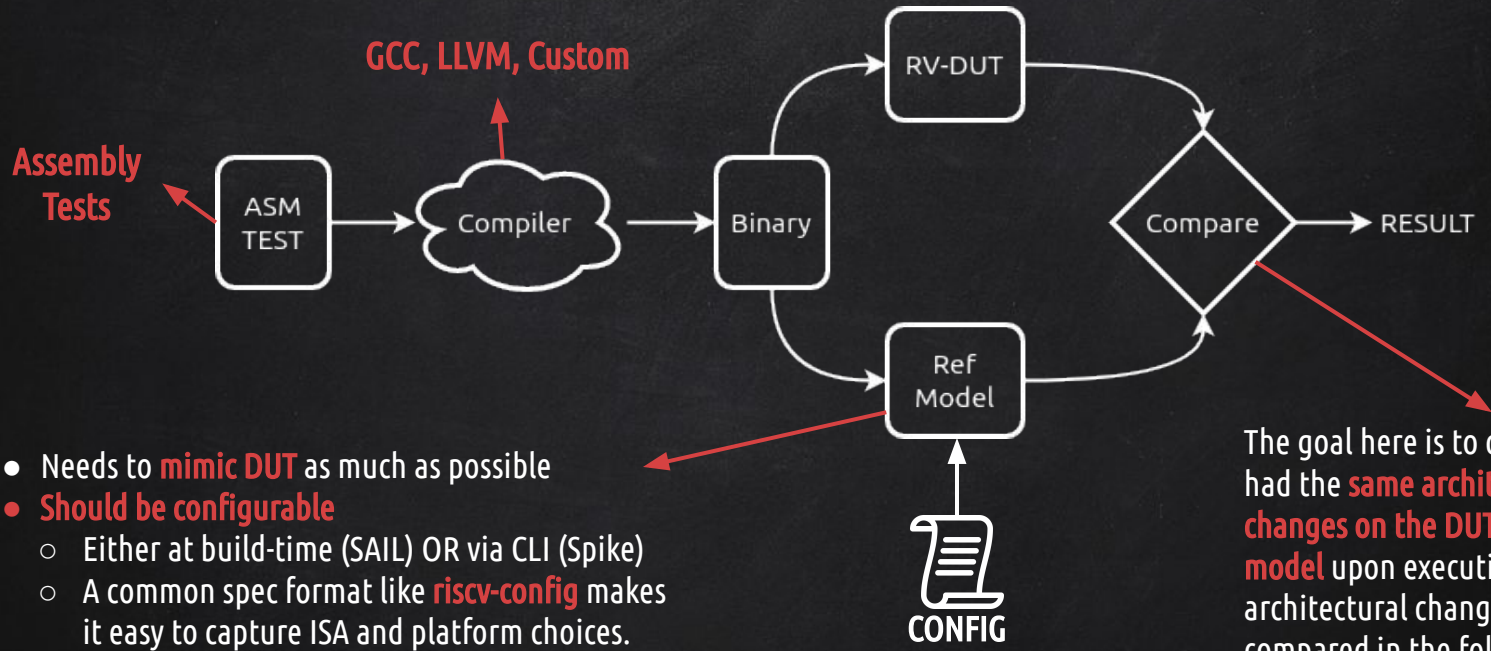
Configurable Cores

The growth and rapid acceptance of High-Level HDLs like BSV, Chisel, Clash, Spinal HDL,etc. has enabled creation of **extremely configurable, modular and scalable** micro-architectures.
Thus, in the new world of RISC-V, **Configurable Core Generators** (like Chromite, Rocket, etc) are becoming the new norm.

RISC-V Verification Generators



An Adaptable Test Harness



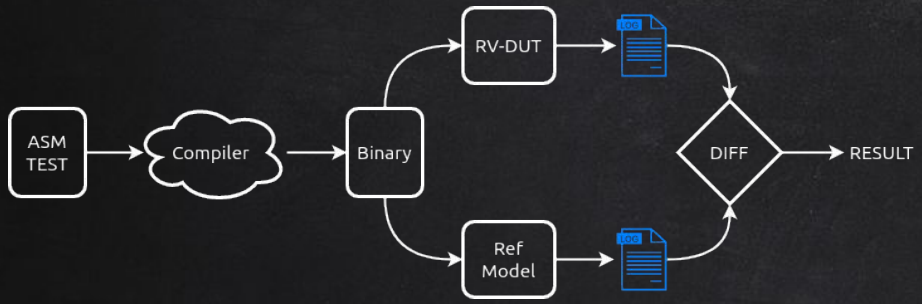
- Needs to **mimic DUT** as much as possible
- **Should be configurable**
  - Either at build-time (SAIL) OR via CLI (Spike)
  - A common spec format like **riscv-config** makes it easy to capture ISA and platform choices. Using **riscv-config to configure the model is a future-ready scalable approach** (check SAIL)
- Should be proven against an accepted model
- Sample Reference Models:
  - **SAIL, Spike, Whisper, etc.**

The goal here is to check if the test had the **same architectural state changes on the DUT and the reference model** upon execution. The architectural changes thus can be compared in the following ways

- **Log Based**
- **Step-n-Compare**

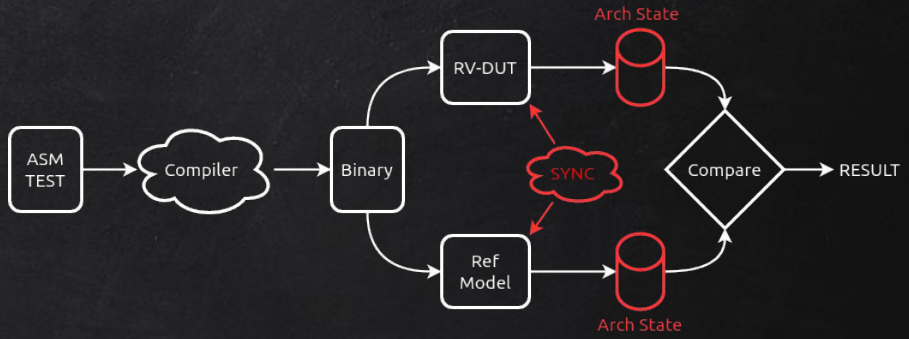
An Adaptable Test Harness

Comparison Modes



LOGs / SIGNATURE BASED

- Logs capture the state changes at the end of each instruction or events (traps, etc)
- Signatures capture a specific memory region and dump it out at the end of the test
- Pros:
  - Easy to set up - requires minimal scripts
  - Fast simulation speeds
- Cons:
  - Log sizes can explode quickly
  - Difficult to debug failures
  - Log formats need to match
  - Certain tests which depend on uncore (counters, timers, etc) cannot be used in this mode



STEP-N-COMPARE BASED

- Synchronize the DUT and the REF to step through one single instruction or a single event and then compare the relevant architectural states of both
- Pros:
  - Easier to debug failures
  - Event synchronization for uncore possible
  - Delay sensitive tests can be used
- Cons:
  - Slow simulation speeds
  - Requires REF model to be wrapped in an API so that state can be synchronized based on DUT events
  - Requires a more complex setup
  - Language can become restrictive

An Adaptable Test Harness

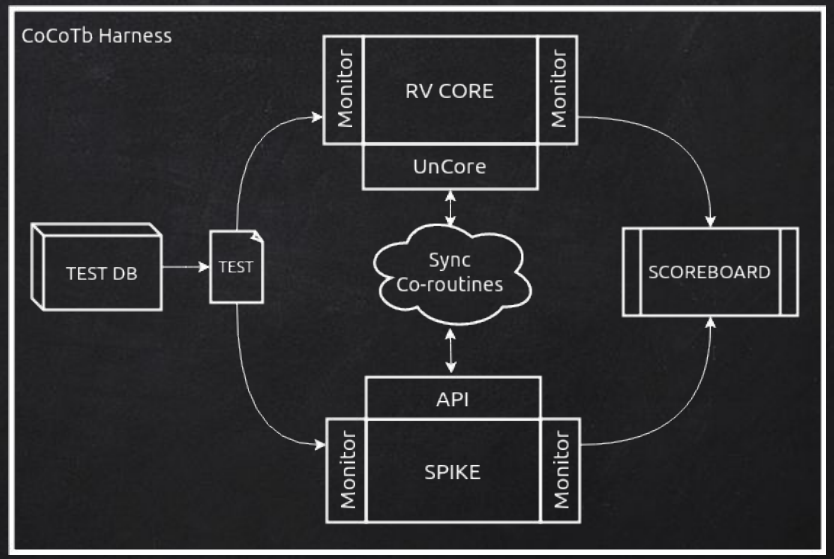
Step-n-Compare with CoCoTb and Spike

CoCoTb

- ↪ Library for digital logic verification in Python
- ↪ Coroutine co-simulation testbench
- ↪ Python interface to RTL simulators
- ↪ User-friendly alternative to Verilog/VHDL
- ↪ Simulator Agnostic Test Env

Spike API

- Build Spike as a shareable object binary
- Create a C API:
  - To initialize and kill spike
  - To read csrs, regfiles, memory, etc
  - To write csrs, regfiles, memory, etc
- Use SWIG to import C API in python world of CoCoTb
- Spike itself might require a few changes - like disabling its own clint, setting of xeip, etc.



Sync Coroutines

- UnCore on DUT includes: Memory, Debug, CLINT, PLIC, etc.
- Synchronization for:
  - To step spike once only when DUT has committed an instruction
  - Events like interrupts from DUT-uncore to be communicated to spike
  - Writes to CSRs with complex WARL behaviors need to be communicated to spike.

Test Generation

As the features of the ISA and the micro-Arch increase, the tests required to verify those features also grow exponentially.
Key is to focus on **Automated Test Generation** as much as possible.

ISA Compliance Tests

These tests focus on checking if the design has interpreted and implemented the spec correctly.

These tests primarily focus on positive testing and corner cases derived from the ISA spec only.

Can be treated as a solid smoke-suite for initial days of verification

Random Tests

These tests focus on generating a sequence of random instructions. Aim to improve initial confidence in the design very quickly.

The sequences and randomness is typically controlled by the user and thus capable of covering a wide variety of corner cases from ISA and micro-Arch with minimal effort.

Micro-Arch Tests

Primary focus here is to test the micro architectural features of the core such as Caches, Branch Predictors, Bypass Logic, etc.

These contribute heavily to the functional coverage closure.

Test Generation

ISA Compliance Tests

RISC-V CTG - A python based coverage driven test-generator.

- Coverpoints are defined as constraints on the user-visible architectural state
- Coverpoints are defined in high-level python eval strings
- Generated Tests are compliant with Compliance Test Format Spec.
- Features:
  - Supports are ratified unpriv extensions
  - Supports various sequence generation
  - Multi-threaded

```
rfmt_op_comb:
  'rs1 == rs2 != rd': 0
  'rs1 == rs2 == rd': 0
  'rs1 != rs2 == rd': 0
  'rs1 == rd != rs2': 0
  'rs1 != rs2 != rd': 0
```

riscv-ctg.readthedocs.io

Random Tests

Various stress test Generators Available:

- **AAPG**:
  - Python based
  - Uses a config file to define pseudo random constraints
  - Supports all ratified extensions
  - Several micro-arch features like cache thrashing, raw depth, recursion, etc
  - Allows one to define a custom snippet to repeat randomly
  - Custom trap handler support
- **RISC-V Torture:**
  - Scala based
  - Doesn't support rv32 off-the-shelf. Needs defining sequences
- **CSMITH** (Random C program Generator)
- **Test-Float** (Random FP Generator)

AAPG: gitlab.com/shaktiproject/tools/aapg
TORTURE: github.com/ucb-bar/riscv-torture
TEST-FLOAT:github.com/ucb-bar/berkeley-testfloat-3

Micro-Arch Tests

UATG - A plugin based python tool to build and maintain parameterized ASM tests focusing on micro-architectural features of a core.

Each Python plugin acts as a separate test-generator - while UATG fills in the commons and glue logic

The parameters can come from riscv-config OR a custom micro-arch yaml - allowing the tests to be parameterized at a much higher level of abstraction.

uatg.readthedocs.io

CHALLENGE : Need to standardize on common collaterals for easier use.

Test Coverage

ISA Coverage - Intent is to define coverpoints based on the ISA defined architectural states: registers, memory, csrs, instruction combinations, etc.

RISCV-ISAC: Given a set of coverpoints and an execution trace of a test run on a model (SAIL), ISAC can provide a report indicating in detail which of those coverpoints were covered by the test/application - thereby providing a ISA quality of the test.

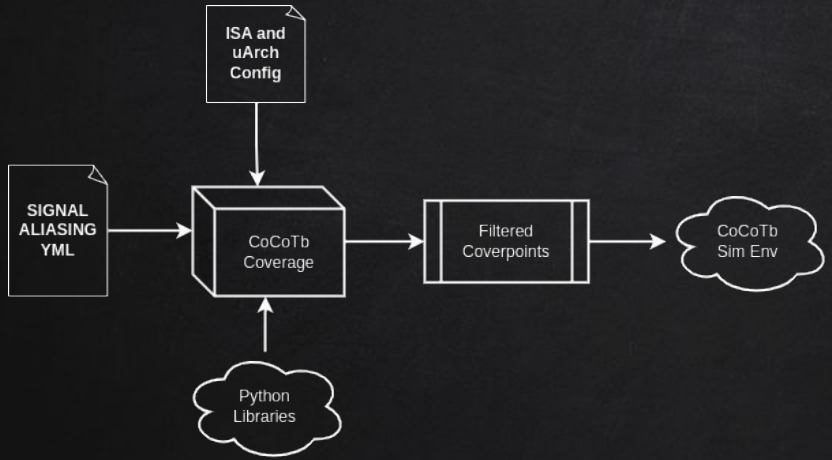
- ✘ Supports multiple models - SAIL, Spike
- ✘ Support all ratified extensions
- ✘ Provides a data propagation report for each test

[Link : riscv-isac.readthedocs.io](riscv-isac.readthedocs.io)

Test Coverage

Functional Coverage: As we shift from static cores to core-generators Functional Coverage becomes a major challenge:

- ✘ Due to abstraction of HLDHLs the signals names at RTL can change quickly
- ✘ Certain signals/coverpoints/properties exist only under certain configurations of ISA and micro-architectural features



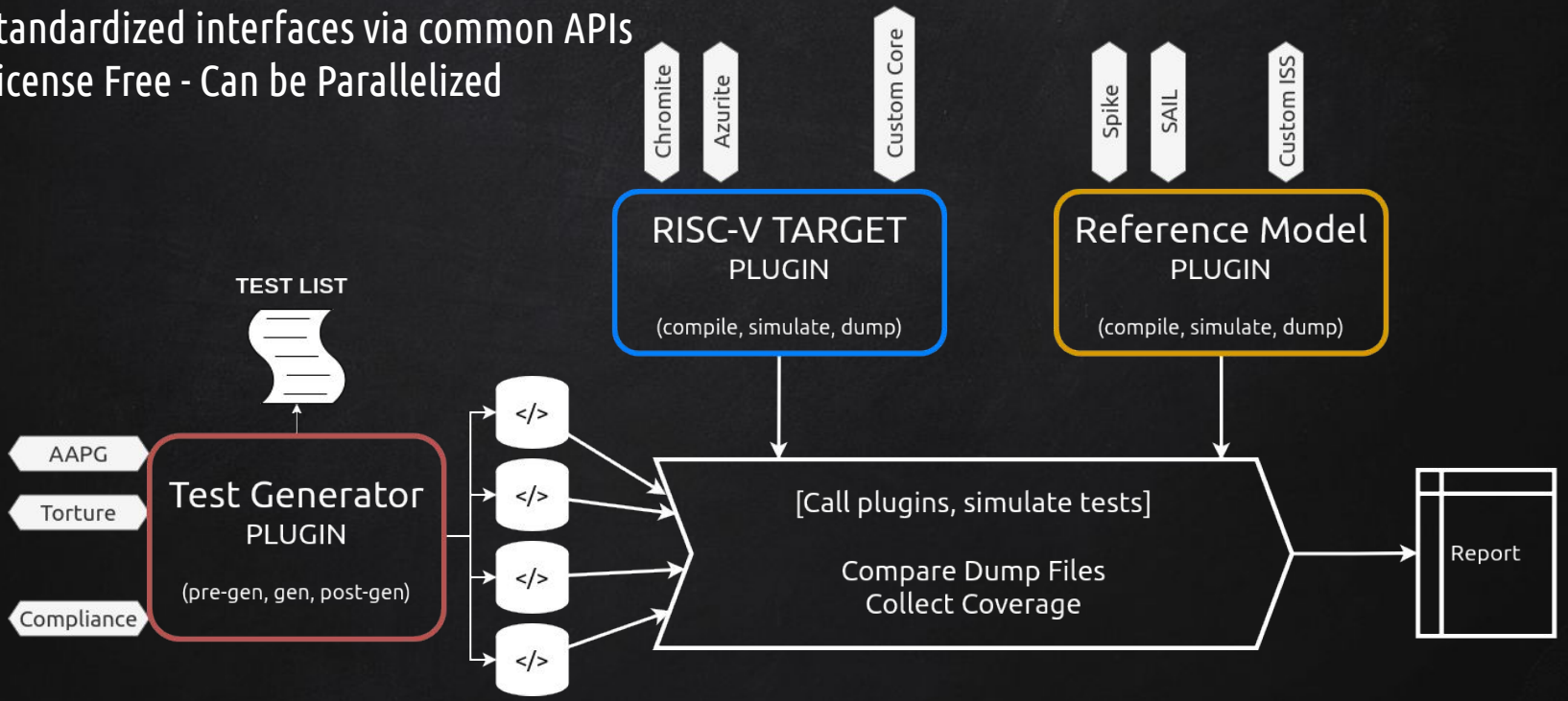

```
walked = []
temp = (1 << n_ones) - 1
for loop_var in range(bit_width):
    if temp <= (1 << bit_width) - 1:
        if not invert:
            if signed:
                walked.append(twos(temp, bit_width))
            else:
                walked.append(temp)
        elif invert:
            if signed:
                walked.append(
                    twos(temp ^ ((1 << bit_width) - 1), bit_width))
            else:
                walked.append(temp ^ ((1 << bit_width) - 1))
        temp = temp << 1
    else:
        break
return walked
```

Link : gitlab.com/shaktiproject/core-py-verif/-/tree/dev

RIVER-CORE : An umbrella framework

- ✘ Scalable and Simple
- ✘ Isolates Generation, SIMs and Models
- ✘ Standardized interfaces via common APIs
- ✘ License Free - Can be Parallelized

Link : river-core.readthedocs.io/en



THANKS

Contact info:

Neel Gala (CTO) : neelgala@incoresemi.com