GUEST LECTURE · RTL TO SIGNOFF

Hardware Verification in Practice

15 years of finding bugs before silicon does: from UVM to AI-orchestrated flows

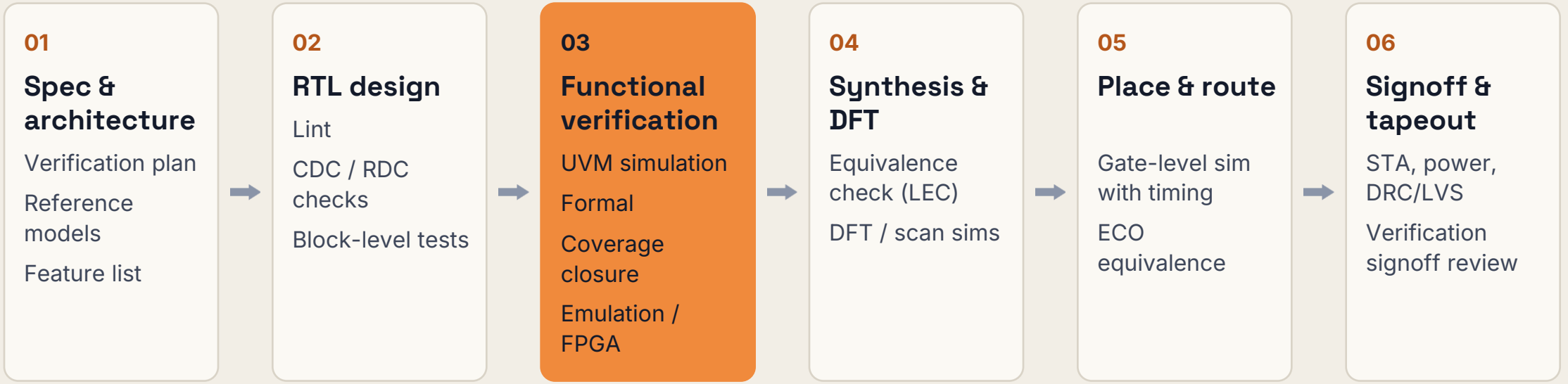
Vikash Kumar

Senior Verification Architect, Arm · IEEE Senior Member

November 19, 2026

THE BIG PICTURE

Verification runs through the whole RTL-to-signoff flow



Every stage a bug escapes makes it more expensive. After tapeout, it costs a respin.

THE CORE PROBLEM

Why verification is hard

70%

Up to 70% of a chip design cycle goes to verification

The scope explodes

There is no way to cover every combination of inputs and states.

Hardware is concurrent

Bugs hide in the timing between events, not in one event.

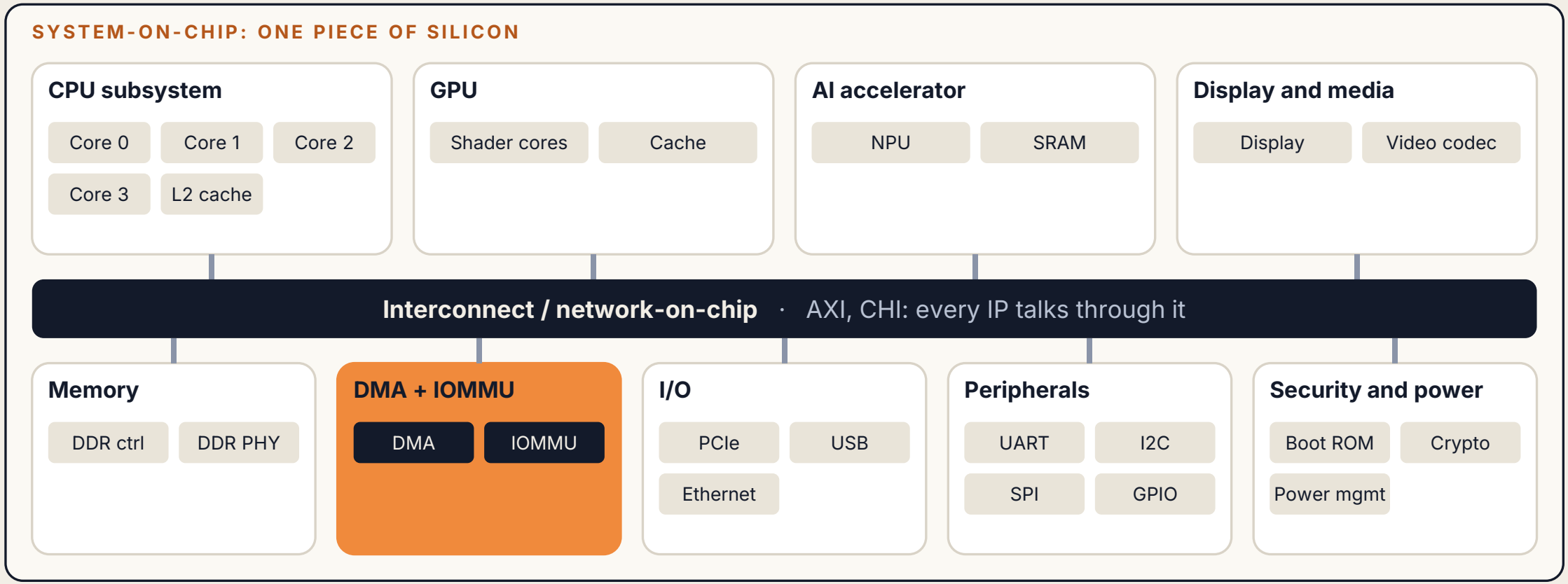
The spec is open to interpretation

A bug can be in the RTL, the testbench or the spec itself.

We can’t test everything, so we choose what to test and measure what we covered.

WHAT WE ARE VERIFYING

What is an SoC? Many IP blocks on one chip



Every box is an IP block, often from a different team or company. The SoC is all of them, working together.

A COMMON QUESTION

Why not just verify the SoC directly?

Too slow

A full-SoC simulation runs far slower than a block test. Far fewer tests per night.

Hard to reach corners

From the top level you can't easily force one FIFO full or one error deep inside a block.

Hard to debug

A failure could come from any of hundreds of blocks and millions of signals.

Too late

The SoC comes together late. Block verification starts months earlier.

Reuse

One IP ships in many SoCs. Verify it once, deeply, and reuse its testbench.

So: bottom-up

Block, then subsystem, then SoC. Each level finds a different kind of bug.

HOW CHIPS ARE BUILT

From IP block to subsystem to SoC

LEVEL 1

IP / block

One function: a DMA engine, an IOMMU, a UART, a memory controller.

Verify: every feature, register and corner case, with full control.

LEVEL 2

Subsystem

Several IPs plus an interconnect: for example CPUs, cache, DMA and IOMMU working together.

Verify: how blocks talk: interfaces, address maps, ordering, clocks and resets.

LEVEL 3

SoC

Subsystems plus the network-on-chip, memory, I/O and power management.

Verify: end-to-end use cases: boot, real software, power modes.

Each level assumes the level below is already clean.

THE VERIFICATION PLAN

How a verification plan is built

01
Read the spec
Spec, micro-architecture, protocol standards

→

02
Extract features
Every requirement becomes one line item

→

03
Categorize and number
Group the items and give each a unique ID

→

04
Map tests
Each item gets one or more tests

→

05
Map coverage
Each item gets coverage that proves it was exercised

→

06
Track to signoff
Status of every item, reviewed with design

The plan is the contract: if it's not in the plan, it won't be verified.

THE VERIFICATION PLAN

Every item gets a category and an ID

FN
Functional

Transfer modes, burst sizes, channels, descriptors

REG
Registers and config

Reset values, access types, illegal writes

ERR
Error handling

Error responses, timeouts, bad descriptors

RST
Reset, clock, power

Reset mid-transfer, clock gating, power states

PERF
Performance

Bandwidth, latency, outstanding requests

STR
Stress and concurrency

Everything at once, back-pressure, many seeds

DMA-RST-004 block · category · number: one ID links the spec, the tests, the coverage and the bugs

THE VERIFICATION PLAN

A slice of a real-style vplan: item, test, coverage, status

ID	Feature	Test	Coverage	Status
DMA-FN-001	Memory-to-memory copy	dma_basic_copy	Size × alignment	Pass · 100%
DMA-REG-003	Register reset values	reg_hw_reset	Register model	Pass · 100%
DMA-ERR-002	Error response on a read	dma_rd_err_rand	Error type × channel	Pass · 92%
DMA-RST-004	Reset during a transfer	dma_rst_mid_xfer	Reset × transfer state	Fail · bug filed
DMA-RST-007	Admin reset, read pending	dma_admin_rst_rd	Assertion: read completes	Fail · bug filed
DMA-STR-001	Everything at once	dma_kitchen_sink	Channel × error × reset	Running · 500 seeds

Signoff means every line has a passing test and closed coverage.

FROM PLAN TO TESTBENCH

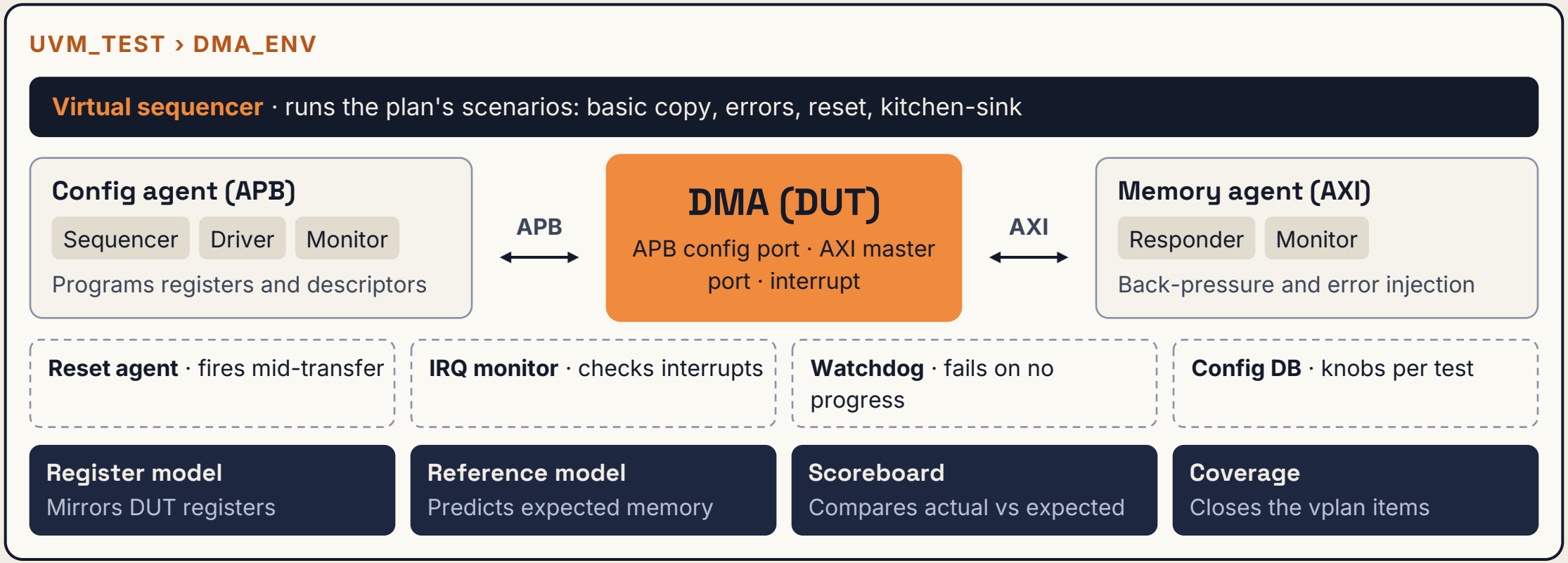
Design the testbench so it can execute every line of the plan

	THE PLAN NEEDS		SO THE TESTBENCH MUST HAVE
FN	Every transfer mode works	→	Sequences per mode, plus a reference model and scoreboard
REG	Registers behave per spec	→	A UVM register model with built-in reset and access tests
ERR	Errors are handled safely	→	A memory responder that injects errors and delays
RST	Reset at any moment	→	A reset agent that fires anytime; the TB flushes itself
PERF	Bandwidth and latency targets	→	Monitors that measure throughput and latency
STR	Everything at once, no hangs	→	A virtual sequence for stress, plus a hang watchdog

Build the testbench from the plan, not from the RTL.

WHAT IT LOOKS LIKE

A UVM testbench for a DMA engine



BUG STORY 1

A surprise reset in the middle of a data transfer

WHAT HAPPENED

Reset arrived while a burst was in flight. After reset, the block came back with stale state and the next transfer failed.

WHY TESTS MISSED IT

Every test applied reset once, at time zero, with the design idle. Nobody reset it while it was busy.

HOW WE CAUGHT IT

Random reset injection during live traffic, with the scoreboard and expected queues reset along with the design.

LESSON **Reset is a feature. Verify it in the middle of traffic, not just at time zero.**

BUG STORY 2

A DMA that hangs waiting for a read that will never finish

WHAT HAPPENED

An admin reset command arrived. The DMA waited for its outstanding read before responding, but the IOMMU had already dropped that read and entered reset.

WHY TESTS MISSED IT

Tests reset the DMA and the IOMMU cleanly. None reset the responder while a DMA read was still in flight.

HOW WE CAUGHT IT

Random responder resets with DMA reads outstanding, plus a watchdog that fails any test that stops making progress.

LESSON **Never wait forever on the other side. Every request needs a way out.**

BUG STORY 3

The kitchen-sink stress test: everything at once

WHAT HAPPENED

Every feature passed on its own. The bug showed up only when many features ran together.

WHY TESTS MISSED IT

The test plan was written feature by feature. No single test combined traffic, errors, interrupts and resets.

HOW WE CAUGHT IT

One stress test that throws everything together: all channels, maximum traffic, random errors and back-pressure, many seeds.

LESSON **Features that pass alone can fail together. Always run a kitchen-sink test.**

STIMULUS

Constrained random: explore widely, but stay legal

Directed tests

Each scenario written by hand. Precise, but only finds the bugs you thought of.

Pure random

Random values everywhere. Explores a lot, but mostly illegal or useless stimulus.

Constrained random

Random within the legal rules, weighted toward corners. Finds bugs nobody imagined.

```
class dma_xfer extends uvm_sequence_item;
  rand bit [31:0] src, dst;
  rand int unsigned len;
  // legal: word-aligned, 1 byte to 4 KB
  constraint legal_c {
    len inside {[1:4096]};
    src[1:0] == 0; dst[1:0] == 0;
  }
  // push toward the corners
  constraint corner_c {
    len dist {1:=10, [2:4095]:/80, 4096:=10};
  }
endclass
```

Same test, new seed, new scenario: randomness finds what you didn’t imagine.

MEASUREMENT

Coverage: proof that the tests did what the plan asked

TYPE 1

Code coverage

Line, branch, condition, toggle and FSM. Automatic. Shows RTL that never ran, but not features that are missing.

TYPE 2

Functional coverage

Covergroups written from the vplan: sizes, modes and crosses like reset × transfer state.

TYPE 3

Assertion coverage

SVA properties exercised and passing: handshake rules, every request gets a completion.

```
covergroup dma_xfer_cg;
  cp_len : coverpoint len {
    bins one = {1}; bins max = {4096}; }
  cp_rst : coverpoint rst_in_state;
  x_rst  : cross cp_rst, cp_len;
endgroup
```

Signoff targets

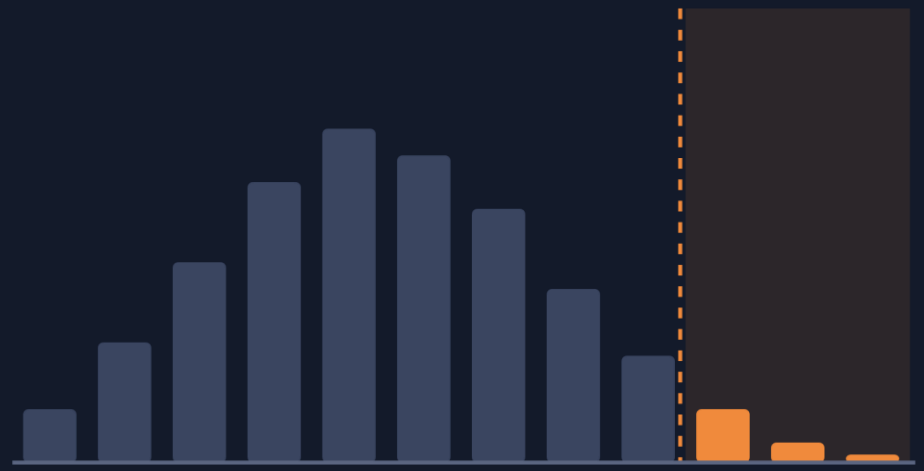
Functional: every vplan item closed

Code: near 100%, every exclusion reviewed

Assertions: all exercised, none failing

SIGNOFF

How do we know verification is complete?



New bugs found per week. Signoff waits for the flat tail (orange).

- **Plan closed**
  Every vplan item has a passing test and closed coverage
- **Coverage closed**
  Functional, code and assertion targets met; exclusions reviewed
- **Regressions stable**
  Clean nightly runs across many seeds, for weeks in a row
- **Bug rate flat**
  New bugs per week near zero; no open critical bugs
- **Beyond simulation**
  Formal proofs, gate-level sim, real software on emulation
- **Reviewed and signed**
  Design, verification and architecture agree on every waiver

You never prove there are no bugs. You prove the risk is low enough to tape out.

AI-ASSISTED VERIFICATION

Where AI helps across the verification flow

01

Verification plan

AI: extracts requirements from the spec

You: review and approve

02

Testbench

AI: generates UVM structure from interface specs

You: architect and refine

03

Stimulus

AI: proposes sequences and constraints

You: decide what matters

04

Coverage

AI: explains holes and suggests tests

You: judge real vs unreachable

05

Regression

AI: clusters failures, summarizes logs

You: root-cause and fix

06

Signoff

AI: drafts status reports

You: make the call

AI drafts and accelerates. The engineer still decides and signs off.

AI-ASSISTED VERIFICATION · STAGE 1

AI for the verification plan

01
Spec and standards
Design spec, micro-architecture, protocol rules

→

02
AI extracts
Every "shall" becomes a candidate requirement

→

03
Draft vplan
Category, ID, test idea, coverage idea

→

04
Engineer review
Accept, fix or reject every line

SPEC SENTENCE (ILLUSTRATIVE)

"On an admin reset, the DMA shall complete or abort all outstanding requests before reporting reset done."

→

AI DRAFT

DMA-RST-007 · Admin reset, read pending
Test: dma_admin_rst_rd · Cover: reset × outstanding reads

Question: what if the responder never answers?

AI-ASSISTED VERIFICATION · STAGE 2

AI for testbench generation: the VEGA flow

INPUT
Interface specs
IP-XACT description plus a metadata file

→

SICA
Interface contracts
Signals, protocols and rules for each port

→

SLTG + LLM
Generate UVM
Env, agents, sequences, scoreboard

→

CHECK
Compile and run
Errors fed back until the testbench is clean

```
$ python vega_llm_tbgen.py \
    --ipxact ahb2apb_bridge.xml \
    --meta ahb2apb_meta.json \
    --output_dir ahb2apb_bridge_llm_uvmtb
```

The engineer still owns

The testbench architecture

The hard checkers and reference model

Review of every generated file

AI-ASSISTED VERIFICATION · STAGES 4–5

AI for coverage, regression and debug

Failure clustering

Group thousands of failures by signature, so hundreds of fails become a few bugs.

Log summaries

Read huge sim logs and point to the first error and its likely cause.

First-fail triage

Find which design change or seed first broke a passing test.

Coverage holes

Explain why a bin is never hit and suggest a constraint or sequence.

Smarter seeds

Rerun the tests and seeds most likely to find new bugs first.

Engineer decides

AI ranks and explains. You confirm the root cause and the fix.

AI-ASSISTED VERIFICATION

Using AI safely in verification

AI IS GOOD AT

Speed and breadth

- Drafting plans, tests and reports
- Generating boilerplate code
- Reading long specs and logs
- Not skipping requirements

ALWAYS KEEP A CHECK

Trust, but verify

- Invented features: trace every line to the spec
- Generated code: must compile, run and pass
- Never let AI grade its own work
- Confidential IP: approved tools only

NEXT **Agentic flows that generate, run, check and retry, with engineers supervising. My ICLAD 2026 paper studies inference-time scaling in these workflows.**