

Verification Planning



Writing the *specification* for the verification process.

- **What** will be verified?
- **How** will it be verified?

At the end of these 2 lectures you will know:

- What a verification strategy and plan may contain.
- How to get this information.

NOTE: The role of Formal Verification will be discussed later.

Verification Strategy AND Plan

The **Verification Strategy** defines:

- *How verification is to be achieved.*
- Including decisions on verification levels, e.g. module/system level,
- functional/formal verification,
- partitioning of the testbench,
- Golden Vectors, etc.

The **Verification Plan** on the other hand:

- *Adds the detail, of how that strategy is to implemented.*
- For *functional verification* (as opposed to formal) the plan would focus on:
 - a list of features, testcases, checks and coverage.

Role of the Verification Planning

To specify the Verification Process.

Traditionally - verification is an ad-hoc process.

- Verification done as time allows.
- Small designs done on FPGA for 1st product (pre-verify),
 - 💡 “Visibility” problems when things go wrong: How to identify the bug? - It’s buried deep down in logic!
- then implement as ASIC for cost reduction.
- 💡 Thorough Verification Planning has only recently been introduced into Design Verification!


Verification Plan

The plan determines WHAT is to be verified.

- It starts from the **Specification!** Why?
- If, and only if, it is in the plan will (/has) it be(en) verified.
- **For 1st time success**, every feature must be identified, under what conditions verification should be performed plus the expected response.

Verification plan documents **all features** (core and optional ones).

- **Prioritize** features.
- Risk can consciously be mitigated to bring in the schedule or reduce costs.

Verification Plan Review meetings include verification engineers, designers and system architects:  long&tall engineer!

Defining 1st time Success

Verification plan defines:

- Which verification scenarios (testbenches/testcases) must be written.
- How complex they need to be.
- Their dependencies (and hence priorities). (Examples in calc1?)
- Variations of Inputs (single items/sequences etc).
- Compliance: Does design comply with standards?
 - AMBA/PI bus protocols standards
- Topology and configuration attributes.
- Corner cases.
- Error conditions.
- ...



Think out of the box!

(Orange test)

Topology and Configuration - Examples

Topology: Environment the design is connected to.

- Devices surrounding the DUV.
 - number of memory banks, their type, their size, ...
 - bus topology, number of slaves
 - ...

Configuration:

- Possible operation modes.
- Controllable parameters in DUV.
 - delays, word size, ...
 - ...



Testbench can *randomize* different scenarios.

Corner Cases and Error Conditions

Corner Case Examples:

- Timing
 - All ports receive an operation at the same time.
 - Multiple interrupts at the same time.
 - Multiple masters requesting the bus at the same time.
- Buffer overflow/underflow.
- Interrupted transfers through priority schemes.

Error Conditions: Don't assume environment is always well-behaved!

- Bad input data, e.g. bad op-code, bad parity, bad length of packet.
- Bus protocol violations.
- Timeouts.

Identify stimulus likely to create error conditions (as opposed to illegal stimulus - for which the DUV response is not specified).

Using the Verification Plan

From the plan, a detailed schedule can be produced:

- Number of **resources** it will take:
 - people
 - computing power
- **Tradeoffs:** more people/machines vs number of tasks/simulation time
- Number of **tasks** that needs to be performed.
- Time it will take with current/proposed resources.



Define your **Verification Team** based on the plan.

Verification Strategy and Plan

Team owns these, incl verification eng, architects and designers!

 Anybody who identifies deficiencies should have input so they are fixed.

Industry has used plans to schedule projects for decades.

■ Learn from **Software projects!**

Adopt *project management methods* from SW projects: continuous re-assessment of priorities and strategy, as our ability to plan with real accuracy is error prone!

Remember:

- The job of the RTL designer is to code RTL.
- The responsibility of the RTL designer is to ensure a working design.
- **The job of the verification engineer is to ...**

... be confident when signing here:

Release to Production

I certify that
this design is
completely verified
and no re-spin
will be required.

X

Verification Levels

Verification can be performed at various granularities:

- **Designer/Unit/Sub-unit**
- **ASIC/FPGA/Re-usable component**
- **System/sub-system/SoC**
- **Board**

How to decide what levels?

Tradeoffs!

(Nothing comes for free.)

Lower levels - smaller pieces:

- Offer more control and observability.
- But more effort to construct environments.
- Easier to drive into interesting conditions and state combinations.

Higher levels:

- Integration of smaller partitions is implicitly verified at the cost of lower control and observability,
- but less effort because fewer environments.

Whatever level is decided:

- The interfaces should remain stable! *debug block*
- (• If sthg is moving and interfaces change - record this every time!)

Each verifiable component should have a specification (document).

Unit/Designer Level

- Usually quite **small**.
- **Interfaces and functionality** tend to **change quite often**.
- Often don't have independent specification associated with them.
- **Environment (if one)** is usually **ad hoc**.
 - Not intended to gain full coverage, just enough to verify basic operation (no boundary conditions).

High number of units in design usually means it isn't reasonable to do unit level due to high effort for environments.



Design for Verification: Design to facilitate verification!

- **Partition** the design so the units have features that are fully contained.

Once a unit is verified, the higher levels can assume it is correct.

So, then what remains to be verified at higher levels?

Reusable Component Level

- Verify component independent of any particular use.
 - Extra challenge for verification team!
 - **Save verification time:**
 - by encoding/encapsulating interfaces (BFMs) to the re-usable component
 - **Components will not be reused if users do not have confidence in them!**
- (• You don't re-use what you don't trust.)

Electronic System Level

What is a system? (Everyone's definition is a little different.)

“A system is a logical partition composed of independently verified components.”

Verification at this level focuses on interaction!

- (• Not on functions encapsulated in a single piece.)
 - Components tend to work well in isolation.
 - 💡 **Problems occur at integration, i.e. at system level!**

A large ASIC may take this approach:

- (• Assuming that all pieces that make up the ASIC are specified, documented, and fully verified.)
 - Assume that individual components are functionally correct.
 - **Testcase** defines the system.
 - 💡 **If the testcase does not exercise a piece, that piece probably does not need to be in the system.**

Current Practices for Verifying a System

Designer Level

- Verification of a module (or a few small modules)

Unit Level

- Verification of a group of modules

Chip/Element

- Verification of an entire logical function (such as a processor, memory controller, Ethernet MAC, etc)

Electronic System Level

- Multiple chip verification
- Sometimes done at the board level.
- Can utilize a mini operating system/platform.

In an ideal world...

Every module would have perfect verification performed.

- All permutations would be verified based on legal inputs.
- All outputs checked on the small pieces of the design.

Unit, chip, and system level would then only need to check interconnections.

- Ensure that modules use correct input/output assumptions and protocols.

This is not realistic!

Why?

So, in reality...

Module verification across an entire system is not feasible.

- Businesses can't support the development/verification expense.
- There are not enough skilled verification engineers.

Verification team leaders must take reasonable tradeoffs:

- Focus on areas of risk and maximum ROI.
 - Concentrate on **electronic system level**.
 - Designer level on riskiest macros.
 - * Consider degree of control/visibility for module level.



Verification strategies/plans help to make informed decisions.

Verification Strategies - HOW

Must decide the following:

- **Box:** White box, black box or grey box?
- **Level of abstraction** where majority of verification takes place.
- Types of **testcases/inputs**.
- **Checking:** How to verify the response?
- If we can't do it exhaustively, we need to be selective.
 - **How random will it be?**
 - **Depends on functionality to be verified.**

Higher level of abstraction - coarser grained verification:

- Less detailed control over timing.
- Less coordination of stimulus and response.
- Easier to generate lots of stimulus and check long responses.

■ **Processor verification:** cycle level vs architectural level

Driving and Checking - The core of the plan.

Exploiting randomization:

- Which inputs to drive?
- How to verify the response?

Random Verification

Which inputs to drive? So far, only **directed testing**.

Random Verification:

- 💡 Does not mean randomly applying 0s and 1s to input.

WHY?

Random Verification

Which inputs to drive? So far, only **directed testing**.

Random Verification:

- 💡 Does not mean randomly applying 0s and 1s to input.
- Still must produce valid stimulus!

WHY?

Sequence of operations and content of data is random.

- What can be randomized when writing into a buffer?

This creates conditions that one does not think of.

- Hit **corner cases**.
- **Unexpected** conditions.
- **Reduces bias** from the verification engineer.
 - Things we understand well vs those we don't.
- Create **background noise**: ■ Interactions not part of main test.

Need (at least) **repeatability!**

Random Verification

Which inputs to drive? So far, only **directed testing**.

Random Verification:

- 💡 Does not mean randomly applying 0s and 1s to input. WHY?
- Still must produce valid stimulus!

Sequence of operations and content of data is random.

■ What can be randomized when writing into a buffer?

This creates conditions that one does not think of.

- Hit **corner cases**.
- **Unexpected** conditions.
- **Reduces bias** from the verification engineer.
 - Things we understand well vs those we don't.
- Create **background noise**: ■ Interactions not part of main test.

Need (at least) **repeatability!** 💡 Use same random seed!

Random Verification (cntd.)

Can be complex to specify and code.

- Need high-level languages. ■ e-language, Vera, c++ testbuilder
- Not so useful without **constraints**.

Must be a **fair representation** of what the **operating conditions** are for the design.

- Must consider distributions and ranges.

More complicated to determine expected response.

WHY?

If input is randomized, how do you know WHAT was verified?

With proper constraints, a random environment can produce directed tests. (The converse is not true.)

Random Verification (cntd.)

Can be complex to specify and code.

- Need high-level languages. ■ e-language, Vera, c++ testbuilder
- Not so useful without **constraints**.

Must be a **fair representation** of what the **operating conditions** are for the design.

- Must consider distributions and ranges.

More complicated to determine expected response.

WHY?

- Use score boarding!

If input is randomized, how do you know WHAT was verified?

With proper constraints, a random environment can produce directed tests. (The converse is not true.)

Random Verification (cntd.)

Can be complex to specify and code.

- Need high-level languages. ■ e-language, Vera, c++ testbuilder
- Not so useful without **constraints**.

Must be a **fair representation** of what the **operating conditions** are for the design.

- Must consider distributions and ranges.

More complicated to determine expected response.

WHY?

- Use score boarding!

If input is randomized, how do you know WHAT was verified?

- Use automatic coverage tools!

With proper constraints, a random environment can produce directed tests.

(The converse is not true.)

Random Verification - Methodology

1. Write and run **random tests** first.

2. Measure coverage. **Identify coverage holes/gaps.**

💡 In practice, a large proportion of the coverage is achieved by driving randomized input.

3. **Add constraints to target these holes.**

4. Measure coverage. **Identify coverage holes/gaps.**

5. **Write “directed tests” to fill the remaining gaps.**

💡 In practice, getting the last 20% coverage can cost 80% of verification effort.

More automation is needed!

Can all holes be filled?

Random Verification - Methodology

1. Write and run **random tests** first.


2. Measure coverage. **Identify coverage holes/gaps.**

 In practice, a large proportion of the coverage is achieved by driving randomized input.

3. **Add constraints to target these holes.**

4. Measure coverage. **Identify coverage holes/gaps.**

5. **Write “directed tests” to fill the remaining gaps.**

 In practice, getting the last 20% coverage can cost 80% of verification effort.

More automation is needed!

Can all holes be filled?

 **Be inquisitive!** If it is very difficult to fill a hole, check whether this hole **can** be filled. If in doubt, **try to prove that it can't!**

Verifying the Response

Deciding how to apply stimulus is usually easy part.

- (• You have control of its timing and content.)

Deciding on the response is hard part.

- How do you determine the expected response?
- How do you determine whether design provided response you expected?

In practice:



Detect errors and stop simulation as early as possible!

- Error identified when it takes place. Easier to debug on full DUV state.
- No simulation cycles wasted. Saves memory.

⇒ **Concentrate on the 1st failure - not all of them.**

Types of Checking

Data checks: Verify the *correctness* of data.

- Is this a legal output?
- Has the payload data been modified while passing through the router?

Temporal checks: Verify *timing* of protocols.

- Has master received an acknowledgement within 3 cycles of request?
- Is the response issued 1 cycle after the request?
- Is this a legal transaction sequence?

Both checks are best done:

- On a high abstraction level. (not the bit level)
- Declaratively, i.e. with expressive languages.

Methods of Checking I

Predict and Check:

- Create stimulus.
- Run stimulus against independent reference model (**the spec**).
- Save predicted results.
- Re-run stimulus on simulation model (**the design**).



Compare results.

■ Used in processor verification and chip verification.

++ Does not require run-time interaction with model.

++ **Reference model** tends to have **high re-use value**.

++ Quick to set up. Can be employed at early stages.

-- Can be difficult to identify bug source.

-- Results have to be predictable from the testcase alone.

-- Input may have to be restricted to achieve results.

Methods of Checking II

Trace and Check later:

- Create tracers that observe and record system activity.
- Gather trace info during simulation.
- After simulation, analyze traces for rule violations.

■ Used in system and protocol verification.

- ++ Don't spend sim time checking results.
- ++ No constraints on stimulus generation.
- ++ Reusable tracers and checkers for standard interfaces.
- ++ Allows correlation between communicating parts of DUV.
- Bugs must propagate to an observed interface.
- Wasted sim time as tests run to completion.
- Requires run-time interaction with the DUV.

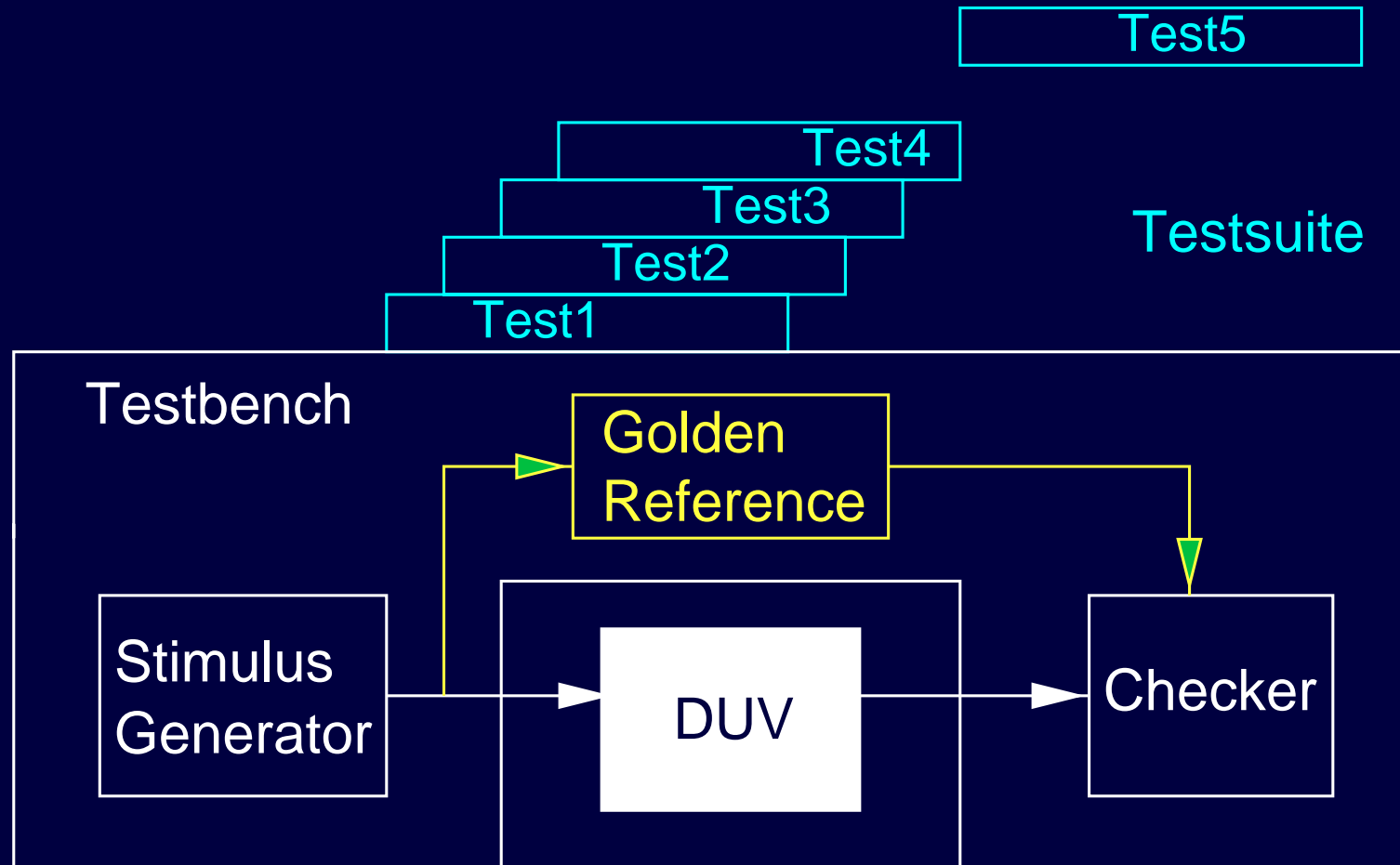
Methods of Checking III

Check as you go:

- Create **environment to simulate, observe and predict behaviour** of simulation model.
- As simulation is running, **environment compares** expected behaviour with observed behaviour.
- Stop testcase when differences occur.
- **Used in unit/chip/subsystem verification.**
- ++ Highly effective.
- ++ Stops test as soon as bug occurs. **No wasted sim cycles.**
- ++ Error messages in environment can be very specific.
- **Environment code is GREATER than DUV HDL code.**
- Requires run-time interaction with the DUV.
- **Danger that environment replicates the design.**
- Design-specific environment code: **low re-use value!**

Self-Checking Testbenches

Use Self-Checking Testbenches!



Mandatory for *random* input generation!

Self-Checking Testbenches

Traditional: Let simulation generate a set of outputs which can then be compared to a set of reference outputs.

 **Comparison decides whether the output is correct.**

(Tools: Scripting languages like Perl are very useful here.)

Self-checking Testbenches are harder (than this) to write.

- **Constantly check** for expected behaviour.
- All checks are part of the testbench.
 - Written once, rather than for each test case.
 - **Real-time checking**, while the test is running.
 - **All checks are active for every test case.**
 - Stop simulation immediately when a check fails.
- **Very hard to code self-checks for all 'generic' error conditions.**
 - It can help to turn some checks off during error conditions.

The Verification Planning Process

- From **Specification** to Features
- From **Features** to Testcases
- From **Testcases** to **Testbenches**

 Creating a verification plan is much easier with a good template document.

- But given the diversity of designs there is no such 'ideal template' for all projects. :(
- Design Verification unit web page has one template from Verification Guild.

Specification to Features

1st step in planning is to identify features to test/verify:

- **Study** design specification - review with architects and designers.
 - Any implementation-specific issues?
- **Label** each feature with short description.
- Ideally, spec and verification plan are **cross referenced**.
- **Describe at which level** the function will be verified.
 - Unit-level features: Bulk of verification.
 - System-level features: Connectivity, flow control, interoperability.
 - * **Question if a feature can be done at unit level!**
- Run all tests on RTL and gate-level simulation.
 - (– This checks the synthesis tool!)
- **Concentrate on functional errors**, not tool induced errors.
- Unconnected nets often arise from hand-stitched designs. Linting!

Features to Testcases

Prioritize the features:

- **Must-have** features: Verify thoroughly. They "gate" tape-out.
- Less important features receive less attention.

Helps project managers to make informed decisions.

Put features into groups: These become the testcases.

- Similar features may require same configuration, granularity.

If a feature does not have a testcase assigned, it is not verified.

Label each **testcase**, give a short **description**, cross-reference to feature list.

Not always 1-1 though!

To close coverage: Some testcases might be added late.

- Document them as such.
- These often don't fall into a group of features.

Define dependencies: This helps prioritize!

Features to Testcases (cntd.)

Driving: Specify testcase stimulus or constraints for random.

When to generate stimulus?

- **On-the-fly:**

- Easy to react to the current state of the DUV.
 - * Makes reaching corner cases easier.
- Easier concurrency control over multiple channels.
- Small memory footprint.

- **Pre-run**

- Can be easier for complex sequences with inter-dependencies.
- May be compulsory. (WHERE?)

- **Mixed: Pre-run and On-the-fly**

Features to Testcases (cntd.)

Driving: Specify testcase stimulus or constraints for random.

When to generate stimulus?

- **On-the-fly:**

- Easy to react to the current state of the DUV.
 - * Makes reaching corner cases easier.
- Easier concurrency control over multiple channels.
- Small memory footprint.

- **Pre-run**

- Can be easier for complex sequences with inter-dependencies.
- May be compulsory. (WHERE?)
 - Assembler program: Must first be compiled and loaded!

- **Mixed: Pre-run and On-the-fly**

Features to Testcases (cntd.)

Driving: Specify testcase stimulus or constraints for random.

When to generate stimulus?

- **On-the-fly:**

- Easy to react to the current state of the DUV.
 - * Makes reaching corner cases easier.
- Easier concurrency control over multiple channels.
- Small memory footprint.

- **Pre-run**

- Can be easier for complex sequences with inter-dependencies.
- May be compulsory. (WHERE?)
 - Assembler program: Must first be compiled and loaded!

- **Mixed: Pre-run and On-the-fly**

- Pre-run: sequence of processor instructions
- On-the-fly: interrupts

Testcases to Testbenches

Testcases naturally fall into groups.


- similar configuration or same abstraction level

These groups are **testbenches**.

 **Good practice to cross-reference testbenches with testcases.**

Assign testbenches to verification engineers.

Don't forget to: **Verify the testbench!**


-  Use a broken model for each feature to be verified.
- Organize regular peer reviews.
- Provide logging messages in testbenches.
 - Log file documents what was simulated/checked.

Tests into Regression Suites

How can you make sure that **no new bugs** are introduced during design?

Regression suites:


- dedicated set of tests
- simulated periodically,
- especially after major design/environment changes

 Ideally, a regression suite should be *comprehensive* and *efficient*!
WHY?

Which tests should be added to a regression suite?

- Tests which identify hard-to-find bugs.
- Tests which yield high coverage.

Remember: Verification Independence!

 It is important to avoid making the same assumptions as the designer.

The *Verification Engineer* must *focus on the functional requirements* of the design *as described in the specification*.